Tugas Kecil Strategi Algoritma 2024/2024
Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding

Disusun Oleh
Kenneth Ricardo Chandra / 13523022

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
JL. GANESHA 10, BANDUNG 40132
2025

BAB I
DESKRIPSI TUGAS

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

Komponen penting dari permainan Rush Hour terdiri dari:

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

2. **Piece –** *Piece* adalah sebuah kendaraan di dalam papan. Setiap *piece* memiliki *posisi*, *ukuran*, dan *orientasi*. *Orientasi* sebuah *piece* hanya dapat berupa horizontal atau vertikal–tidak mungkin diagonal. *Piece* dapat memiliki beragam *ukuran*, yaitu jumlah *cell* yang ditempati oleh *piece*. Secara standar, variasi *ukuran* sebuah *piece* adalah *2-piece* (menempati 2 *cell*) atau *3-piece* (menempati 3 *cell*). Suatu *piece* tidak dapat digerakkan melewati/menembus *piece* yang lain.

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

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

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

BAB II
DASAR TEORI

Algoritma

1. Greedy Best-First Search (GBFS)

Greedy Best-First Search (GBFS) merupakan salah satu algoritma pencarian informasi yang mengandalkan fungsi heuristik untuk memandu proses eksplorasi dari suatu ruang keadaan. Berbeda dengan algoritma pencarian buta (uninformed search), GBFS menggunakan informasi tambahan, yaitu perkiraan biaya dari node saat ini menuju node tujuan, untuk membuat keputusan tentang node mana yang akan diekspansi selanjutnya. Pada setiap langkah, GBFS akan memilih node yang memiliki nilai heuristik terkecil dari semua node yang telah diekspansi dan belum dikunjungi sepenuhnya. Nilai heuristik ini diinterpretasikan sebagai estimasi terdekat menuju gol. Algoritma ini bersifat "greedy" karena selalu memilih opsi yang tampak paling menjanjikan pada saat ini, tanpa mempertimbangkan apakah pilihan tersebut akan mengarah pada solusi optimal secara global.

Misalkan h(n) adalah fungsi heuristik yang mengestimasi biaya dari node n menuju node tujuan. GBFS akan selalu mengekspansi node n dari antrean prioritas (biasanya diimplementasikan sebagai *min-priority queue*) yang memiliki nilai h(n) terkecil.

Kelebihan:

- Efisiensi Waktu: GBFS cenderung lebih cepat daripada algoritma pencarian buta karena fokus pada arah yang menjanjikan, sehingga dapat menemukan solusi lebih cepat jika heuristiknya akurat.
- Penggunaan Memori Lebih Rendah: Dalam beberapa kasus, GBFS dapat memiliki kebutuhan memori yang lebih rendah dibandingkan algoritma pencarian lain yang perlu menyimpan seluruh ruang keadaan.

Kekurangan:

- Tidak Optimal: Karena sifatnya yang "greedy", GBFS tidak menjamin penemuan solusi optimal. Algoritma ini dapat terjebak dalam jalur yang terlihat baik di awal namun ternyata tidak mengarah ke solusi terbaik secara keseluruhan.
- Tidak Lengkap (Not Complete): GBFS dapat terjebak dalam *loop* tak terbatas jika terdapat siklus dalam graf dan fungsi heuristik tidak dapat menghindarinya.

2. A* Search

Algoritma A* adalah salah satu algoritma pencarian graf terbaik yang menggabungkan keunggulan Uniform Cost Search dan Greedy Best-First Search untuk menemukan jalur terpendek dari node awal ke node tujuan. A* mencapai optimalitas dan kelengkapan (completeness) dengan menggunakan fungsi evaluasi yang cerdas. A* menggunakan fungsi evaluasi f(n) untuk setiap node n, yang didefinisikan sebagai: $f(n)=g(n)+h(n)$ di mana:

- g(n) adalah biaya aktual dari node awal ke node n.
- h(n) adalah biaya estimasi (heuristik) dari node n ke node tujuan.

Algoritma A* selalu mengekspansi node n dari antrean prioritas yang memiliki nilai f(n) terkecil. Dengan demikian, A* menyeimbangkan antara biaya yang telah dikeluarkan (g(n)) dan estimasi biaya yang tersisa (h(n)).

Optimalitas A* terjamin jika fungsi heuristik h(n) memenuhi dua syarat:

1. Admissible Heuristic (Heuristik Admisibel): $h(n) \leq h^*(n)$ untuk setiap node n, di mana $h^*(n)$ adalah biaya aktual dari node n ke node tujuan. Dengan kata lain, heuristik tidak pernah melebih-lebihkan biaya sebenarnya.
2. Consistent Heuristic (Heuristik Konsisten) / Monotone: Untuk setiap node n dan setiap suksesor n' dari n, dan biaya langkah $c(n,n')$ dari n ke n', berlaku: $h(n) \leq c(n,n')+h(n')$ Jika heuristik konsisten, maka ia juga admisibel.

Kelebihan:

- Optimal: A* menjamin penemuan solusi optimal (jalur terpendek) jika heuristiknya admisibel.
- Lengkap (Complete): A* akan selalu menemukan solusi jika ada, asalkan ruang pencarian hingga.
- Efisiensi: A* jauh lebih efisien dibandingkan pencarian buta karena dipandu oleh heuristik.

Kekurangan:

- Penggunaan Memori: A* dapat membutuhkan memori yang besar karena harus menyimpan semua node yang telah diekspansi dalam memori untuk mencegah kunjungan berulang dan membangun jalur.
- Sensitif terhadap Kualitas Heuristik: Performa A* sangat bergantung pada kualitas fungsi heuristik. Heuristik yang buruk dapat mengurangi efisiensi atau bahkan mengarah pada pencarian yang tidak optimal.

3. Iterative Deepening A* (IDA*)

Iterative Deepening A* (IDA*) adalah varian dari algoritma A* yang dirancang untuk mengatasi masalah penggunaan memori yang tinggi pada A* standar, terutama pada ruang pencarian yang sangat besar. IDA* mengadaptasi prinsip Iterative Deepening Depth-First Search (IDDFS) dengan menggabungkan heuristik A*. IDA* bekerja dengan melakukan serangkaian pencarian *depth-first* iteratif, di mana setiap iterasi memiliki batas *f-cost* (biaya g(n)+h(n)) yang semakin meningkat. Pada setiap iterasi, algoritma melakukan pencarian *depth-first* hingga nilai f(n) dari node yang diekspansi melebihi batas *f-cost* saat ini. Jika node tujuan tidak ditemukan dalam batas tersebut, batas *f-cost* dinaikkan ke nilai f(n) minimum dari node yang "terpotong" (pruned) pada iterasi sebelumnya, dan proses diulang. Mirip dengan A*, IDA* menggunakan fungsi f(n)=g(n)+h(n). Namun, alih-alih menggunakan antrean prioritas global, IDA* menggunakan *depth-limited depth-first search* dengan batas yang berdasarkan nilai f(n).

Kelebihan:

- Optimal: IDA* juga menjamin penemuan solusi optimal jika heuristiknya admisibel.
- Penggunaan Memori yang Efisien: Keunggulan utama IDA* adalah penggunaan memori yang minimal karena sifatnya yang *depth-first*, hanya menyimpan jalur saat ini dalam stack. Ini membuatnya ideal untuk masalah dengan ruang keadaan yang sangat besar.
- Lengkap (Complete): IDA* akan selalu menemukan solusi jika ada.

Kekurangan:

- Redundant Computation: Kelemahan utama IDA* adalah *recomputasi* node yang berulang kali di berbagai iterasi. Hal ini dapat menyebabkan IDA* mengekspansi node yang sama berkali-kali, meskipun pada umumnya tidak terlalu signifikan dalam hal waktu dibandingkan keuntungan memorinya.
- Waktu Eksekusi Lebih Lama: Meskipun optimal dalam memori, IDA* seringkali lebih lambat daripada A* jika A* dapat dijalankan dengan memori yang tersedia.

4. Uniform Cost Search (UCS)

Uniform Cost Search (UCS) adalah algoritma pencarian yang dirancang untuk menemukan jalur dengan biaya terendah (bukan hanya jumlah langkah terpendek) dari node awal ke node tujuan dalam graf berbobot. UCS dapat dianggap sebagai kasus khusus dari A* di mana fungsi heuristik h(n) selalu nol untuk semua node, atau sebagai variasi dari Breadth-First Search (BFS) yang dipandu oleh cost kumulatif. UCS mengeksplorasi graf dengan selalu melakukan ekspansi node yang memiliki biaya kumulatif terkecil dari node awal. Biaya kumulatif g(n) adalah jumlah bobot tepi (edge weights) dari node awal hingga node n. Algoritma ini menggunakan antrean prioritas (min-priority queue) untuk menyimpan node-node yang akan diekspansi, diurutkan berdasarkan nilai g(n). Pada setiap langkah, UCS akan mengekspansi node n dari antrean prioritas yang memiliki nilai g(n) terkecil. f(n)=g(n) di mana g(n) adalah biaya jalur dari node awal ke node n.

Kelebihan:

- Optimal: UCS menjamin penemuan solusi optimal (jalur dengan biaya terendah) jika semua bobot tepi tidak negatif.
- Lengkap (Complete): UCS akan selalu menemukan solusi jika ada, asalkan setiap biaya langkah lebih besar dari suatu konstanta positif yang kecil dan graf tidak memiliki siklus dengan biaya nol atau negatif.

Kekurangan:

- Efisiensi Waktu: UCS bisa sangat lambat dalam graf dengan banyak jalur panjang atau biaya langkah kecil, karena ia harus menjelajahi semua jalur potensial hingga menemukan yang termurah. Dalam kasus terburuk, UCS akan menjelajahi seluruh ruang pencarian.
- Penggunaan Memori: Sama seperti BFS, UCS dapat membutuhkan memori yang besar karena harus menyimpan semua node yang telah diekspansi untuk mencegah kunjungan berulang dan membangun jalur.
- Tidak Memanfaatkan Heuristik: UCS tidak menggunakan informasi heuristik, sehingga tidak dapat dipandu menuju tujuan secara efisien. Hal ini membuatnya kurang efisien dibandingkan algoritma berpandu heuristik seperti A*.

Heuristik

1. Heuristik Manhattan Distance (Jarak Manhattan)

Heuristik Manhattan Distance, atau jarak blok kota, biasanya lebih relevan untuk masalah di mana objek dapat bergerak secara ortogonal (atas/bawah, kiri/kanan) pada grid, dan biaya setiap langkah adalah 1. Dalam konteks Rush Hour, heuristik ini dimodifikasi untuk fokus pada mobil merah.

Untuk Rush Hour, Manhattan Distance didefinisikan sebagai jumlah langkah minimum yang dibutuhkan mobil merah untuk mencapai kolom pintu keluar jika tidak ada kendaraan lain yang menghalanginya.

Asumsi dan Perhitungan:

Kita asumsikan mobil merah adalah satu-satunya kendaraan yang perlu dipindahkan.
Pintu keluar berada di ujung kanan grid.
Mobil merah hanya bisa bergerak secara horizontal.
Misalnya, jika mobil merah berada di kolom C_merah dan pintu keluar berada di kolom C_keluar, maka jarak Manhattan adalah:
$h(n)=C_{keluar} - C_{merah}$

Contoh Visual:
Bayangkan mobil merah (panjang 2 blok) berada di kolom 2 pada baris pintu keluar (misalnya baris 3). Pintu keluar ada di kolom 6.

| | | | | | |
| | | | | | |
| | M R | | | | Exit ->
Di sini, C_merah (posisi ujung kanan mobil merah) adalah 3 (jika dihitung dari 1). Pintu keluar ada di kolom 6.
Maka, h(n)=6−3=3. Mobil merah membutuhkan 3 langkah maju untuk keluar jika tidak ada halangan.

*Properti (untuk A):**

Admisibel: Ya. Karena heuristik ini menghitung langkah minimum tanpa adanya hambatan, ia tidak pernah melebih-lebihkan jumlah langkah sebenarnya yang diperlukan. Setiap langkah yang dibutuhkan mobil merah untuk bergerak ke depan setidaknya memerlukan satu gerakan.
Konsisten: Ya, jika pergerakan mobil merah hanya satu langkah per waktu dan biaya per langkah adalah konstan (misalnya 1).

2. Heuristik Blocking Cars (Mobil Penghalang)

Heuristik Blocking Cars adalah heuristik yang lebih spesifik untuk *Rush Hour* dan seringkali lebih informatif daripada sekadar Manhattan Distance, meskipun masih merupakan estimasi yang konservatif.

Heuristik Blocking Cars menghitung jumlah kendaraan (mobil atau truk) yang berada di jalur mobil merah dan secara langsung menghalangi pergerakannya menuju pintu keluar.

Asumsi dan Perhitungan:

- Identifikasi baris di mana mobil merah berada.
- Scan kolom-kolom di depan mobil merah (menuju pintu keluar).
- Hitung setiap kendaraan lain (selain mobil merah itu sendiri) yang menempati salah satu kolom di jalur mobil merah menuju pintu keluar.

Contoh Visual: Ambil contoh mobil merah (R) berada di baris yang sama dengan pintu keluar. Ada dua mobil lain (A dan B) yang menghalangi.

| | | | | | |
| | | | | | |
| | R A | | B | Exit ->

Di sini, mobil A dan mobil B keduanya berada di depan mobil merah. Maka, h(n)=2 (karena ada 2 mobil penghalang: A dan B). Setiap mobil ini harus bergerak setidaknya satu kali untuk menyingkir dari jalur mobil merah.

Properti (untuk A)*:

- Admisibel: Ya. Setiap mobil penghalang yang teridentifikasi setidaknya membutuhkan satu gerakan untuk dipindahkan dari jalur mobil merah. Bahkan mungkin memerlukan lebih dari satu gerakan jika ada mobil lain yang menghalangi mobil penghalang itu sendiri. Oleh karena itu, heuristik ini tidak pernah melebih-lebihkan biaya sebenarnya.
- Konsisten: Umumnya ya. Jika sebuah mobil penghalang bergerak, nilai heuristik bisa tetap sama (jika hanya bergeser dan masih di jalur) atau berkurang (jika pindah dari jalur). Jika berkurang, selisihnya tidak akan lebih dari biaya langkah (1).

BAB III
IMPLEMENTASI

Implementasi dilakukan dalam bahasa Java dan output CLI dengan membuat dua buah heuristik dan 4 algoritma yaitu dengan heuristik Manhattan Distance dan Blocking Cars atau obstacle. Algoritma yang digunakan merupakan Greedy Best First Search atau GBFS, A*, Uniform Cost Search atau UCS dan Iterative Deepening A* atau IDA*. Implementasi dilakukan dengan membuat sebuah board dengan piece yang akan berubah dengan node untuk membuat sebuah path dari board tersebut. Terdapat juga InputParser dan OutputWriter yang bertugas sebagai I/O dalam program ini.
Berikut implementasinya

Board.java

```
package view;

import java.util.*;
```

```java
public class Board {
    private char[][] grid;
    private Map<Character, Piece> pieceMap;
    private int exitRow, exitCol;

    public Board(char[][] initialGrid, Map<Character, Piece> pieces, int
exitRow, int exitCol) {
        this.grid = initialGrid;
        this.pieceMap = (pieces != null) ? new HashMap<>(pieces) : new
HashMap<>();
        this.exitRow = exitRow;
        this.exitCol = exitCol;
    }

    public boolean isGoal() {
        Piece main = pieceMap.get('P');
        if (main == null) return false;
        for (int offset = 0; offset < main.length; offset++) {
            int currX = main.x + (main.isHorizontal ? offset : 0);
            int currY = main.y + (main.isHorizontal ? 0 : offset);
            if (currX == exitRow && currY == exitCol) return true;
        }
        return false;
    }

    public String getBoardKey() {
        StringBuilder keyBuilder = new StringBuilder();
        List<Character> ids = new ArrayList<>(pieceMap.keySet());
        Collections.sort(ids);
        for (char ch : ids) {
            Piece p = pieceMap.get(ch);

keyBuilder.append(ch).append(p.x).append(',').append(p.y).append(';');
        }
        return keyBuilder.toString();
    }

    public Board createCopy() {
        int height = grid.length;
```

```
        int width = grid[0].length;
        char[][] copiedGrid = new char[height][width];

        for (int r = 0; r < height; r++) {
            copiedGrid[r] = Arrays.copyOf(grid[r], width);
        }

        Map<Character, Piece> copiedPieces = new HashMap<>();
        for (Map.Entry<Character, Piece> entry : pieceMap.entrySet()) {
            Piece original = entry.getValue();
            copiedPieces.put(entry.getKey(),
                new Piece(original.id, original.x, original.y,
original.length, original.isHorizontal, original.isPrimary));
        }

        return new Board(copiedGrid, copiedPieces, exitRow, exitCol);
    }

    public void shiftPiece(char pieceId, int moveAmount) {
        Piece target = pieceMap.get(pieceId);
        if (target == null) return;

        int proposedX = target.x + (target.isHorizontal ? moveAmount :
0);
        int proposedY = target.y + (target.isHorizontal ? 0 :
moveAmount);

        if (proposedX < 0 || proposedY < 0 ||
            (target.isHorizontal && proposedX + target.length >
grid[0].length) ||
            (!target.isHorizontal && proposedY + target.length >
grid.length)) {
            return;
        }
        for (int i = 0; i < target.length; i++) {
            int clearX = target.x + (target.isHorizontal ? i : 0);
            int clearY = target.y + (target.isHorizontal ? 0 : i);
            grid[clearY][clearX] = '.';
        }
```

```
        target.x = proposedX;
        target.y = proposedY;
        for (int i = 0; i < target.length; i++) {
            int fillX = target.x + (target.isHorizontal ? i : 0);
            int fillY = target.y + (target.isHorizontal ? 0 : i);
            grid[fillY][fillX] = target.id;
        }
    }

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

    public Map<Character, Piece> getPieceMap() {
        return pieceMap;
    }

    public int getExitRow() {
        return exitRow;
    }

    public int getExitCol() {
        return exitCol;
    }
}
```

Piece.java

```
package view;

public class Piece {
    public char id;
    public int x, y;
    public int length;
    public boolean isHorizontal;
```

```
    public boolean isPrimary;

    public Piece(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;
    }
}
```

Node.java

```
package view;

import java.util.*;

public class Node {
    public Board board;
    public Node parent;
    public String moveDesc;
    public int g;
    public int h;

    public Node(Board board, Node parent, String moveDesc, int g, int h)
{
        this.board = board;
        this.parent = parent;
        this.moveDesc = moveDesc;
        this.g = g;
        this.h = h;
    }

    public static List<Node> reconstructPath(Node goal) {
        LinkedList<Node> path = new LinkedList<>();
```

```
        Node curr = goal;
        while (curr != null) {
            path.addFirst(curr);
            curr = curr.parent;
        }
        return path;
    }

    public static List<Node> generateNextNodes(Node current) {
        List<Node> successors = new ArrayList<>();
        Board board = current.board;

        for (Map.Entry<Character, Piece> entry :
board.getPieceMap().entrySet()) {
            char pieceId = entry.getKey();
            Piece p = entry.getValue();

            if (p.isHorizontal) {
                for (int offset = 1; p.x - offset >= 0; offset++) {
                    int posX = p.x - offset;
                    int posY = p.y;
                    if (board.getGrid()[posY][posX] != '.' && !(posX ==
board.getExitRow() && posY == board.getExitCol()))
                        break;

                    Board copied = board.createCopy();
                    copied.shiftPiece(pieceId, -offset);
                    String move = "Geser " + pieceId + " ke kiri " +
offset;
                    successors.add(new Node(copied, current, move,
current.g + 1, 0));
                }

                for (int offset = 1; p.x + p.length - 1 + offset <
board.getGrid()[0].length; offset++) {
                    int posX = p.x + p.length - 1 + offset;
                    int posY = p.y;
                    if (board.getGrid()[posY][posX] != '.' && !(posX ==
board.getExitRow() && posY == board.getExitCol()))
```

```
                        break;

                    Board copied = board.createCopy();
                    copied.shiftPiece(pieceId, offset);
                    String move = "Geser " + pieceId + " ke kanan " +
offset;
                    successors.add(new Node(copied, current, move,
current.g + 1, 0));
                }
            } else {
                for (int offset = 1; p.y - offset >= 0; offset++) {
                    int posX = p.x;
                    int posY = p.y - offset;
                    if (board.getGrid()[posY][posX] != '.' && !(posX ==
board.getExitRow() && posY == board.getExitCol()))
                        break;

                    Board copied = board.createCopy();
                    copied.shiftPiece(pieceId, -offset);
                    String move = "Geser " + pieceId + " ke atas " +
offset;
                    successors.add(new Node(copied, current, move,
current.g + 1, 0));
                }

                for (int offset = 1; p.y + p.length - 1 + offset <
board.getGrid().length; offset++) {
                    int posX = p.x;
                    int posY = p.y + p.length - 1 + offset;
                    if (board.getGrid()[posY][posX] != '.' && !(posX ==
board.getExitRow() && posY == board.getExitCol()))
                        break;

                    Board copied = board.createCopy();
                    copied.shiftPiece(pieceId, offset);
                    String move = "Geser " + pieceId + " ke bawah " +
offset;
                    successors.add(new Node(copied, current, move,
current.g + 1, 0));
```

```
                }
            }
        }

        return successors;
    }
}
```

Algoritma yang digunakan
UCS.java

```
package algorithm;

import java.util.*;
import view.*;

public class UCS {

    public static List<Node> solve(Board initialBoard) {
        PriorityQueue<Node> openSet = new
PriorityQueue<>(Comparator.comparingInt(n -> n.g));
        Set<String> visited = new HashSet<>();
        long startTime = System.currentTimeMillis();

        Node startNode = new Node(initialBoard, null, null, 0, 0);
        openSet.add(startNode);

        while (!openSet.isEmpty()) {
            Node current = openSet.poll();

            if (current.board.isGoal()) {
                long endTime = System.currentTimeMillis();
                System.out.println("Solusi ditemukan dalam " + current.g
+ " langkah");
                System.out.println("Waktu pencarian: " + (endTime -
startTime) + " ms");
```

```
                return Node.reconstructPath(current);
            }

            String key = current.board.getBoardKey();
            if (visited.contains(key)) continue;
            visited.add(key);

            List<Node> nextNodes = Node.generateNextNodes(current);
            for (Node nextNode : nextNodes) {
                String nextKey = nextNode.board.getBoardKey();
                if (!visited.contains(nextKey)) {
                    nextNode.g = current.g + 1;
                    openSet.add(nextNode);
                }
            }
        }

        System.out.println("Tidak ada solusi");
        long endTime = System.currentTimeMillis();
        System.out.println("Waktu pencarian: " + (endTime - startTime) +
" ms");
        return null;
    }
}
```

GBFS.java

```
package algorithm;

import java.util.*;
import view.*;

public class GBFS {

    public static int calculateHeuristic(Board state, String type) {
        return switch (type.toLowerCase()) {
```

```
            case "manhattan" -> heuristicManhattan(state);
            case "blockingcars" -> heuristicBlockers(state);
            default -> throw new IllegalArgumentException("Unknown
heuristic type: " + type);
        };
    }

    public static int heuristicManhattan(Board state) {
        Piece primary = state.getPieceMap().get('P');
        if (primary == null) return 0;

        if (primary.isHorizontal) {
            return Math.abs(state.getExitCol() - (primary.x +
primary.length - 1));
        } else {
            return Math.abs(state.getExitRow() - (primary.y +
primary.length - 1));
        }
    }

    public static int heuristicBlockers(Board state) {
        Piece primary = state.getPieceMap().get('P');
        if (primary == null) return 0;

        int blockers = 0;
        if (primary.isHorizontal) {
            int y = primary.y;
            for (int x = primary.x + primary.length; x <=
state.getExitCol(); x++) {
                char cell = state.getGrid()[y][x];
                if (cell != '.' && cell != primary.id) blockers++;
            }
        } else {
            int x = primary.x;
            for (int y = primary.y + primary.length; y <=
state.getExitRow(); y++) {
                char cell = state.getGrid()[y][x];
                if (cell != '.' && cell != primary.id) blockers++;
            }
```

```
        }
        return blockers;
    }

    public static List<Node> solve(Board initialBoard, String
heuristicType) {
        PriorityQueue<Node> openSet = new
PriorityQueue<>(Comparator.comparingInt(n -> n.h));
        Set<String> visited = new HashSet<>();
        long startTime = System.currentTimeMillis();

        int h0 = calculateHeuristic(initialBoard, heuristicType);
        Node startNode = new Node(initialBoard, null, null, 0, h0);
        openSet.add(startNode);

        while (!openSet.isEmpty()) {
            Node current = openSet.poll();

            if (current.board.isGoal()) {
                long endTime = System.currentTimeMillis();
                System.out.println("Solusi ditemukan dalam " + current.g
+ " langkah");
                System.out.println("Waktu pencarian: " + (endTime -
startTime) + " ms");
                return Node.reconstructPath(current);
            }

            String key = current.board.getBoardKey();
            if (visited.contains(key)) continue;
            visited.add(key);

            for (Node nextNode : Node.generateNextNodes(current)) {
                String nextKey = nextNode.board.getBoardKey();
                if (!visited.contains(nextKey)) {
                    nextNode.h = calculateHeuristic(nextNode.board,
heuristicType);
                    nextNode.g = current.g + 1;
                    openSet.add(nextNode);
                }
```

```
            }
        }

        System.out.println("Tidak ada solusi");
        long endTime = System.currentTimeMillis();
        System.out.println("Waktu pencarian: " + (endTime - startTime) +
" ms");
        return null;
    }
}
```

AStar.java

```
package algorithm;

import java.util.*;
import view.*;

public class AStar {

    public static int calculateObstacleHeuristic(Board board, Piece
primaryPiece) {
        int obstacles = 0;
        int exitCol = board.getExitCol();
        int exitRow = board.getExitRow();
        int pieceX = primaryPiece.x;
        int pieceY = primaryPiece.y;

        for (Piece p : board.getPieceMap().values()) {
            if (p == primaryPiece) continue;

            if (p.isHorizontal && p.y == pieceY) {
                if ((pieceX < exitCol && p.x > pieceX && p.x < exitCol)
||
                    (pieceX > exitCol && p.x < pieceX && p.x + p.length
> exitCol)) {
```

```
                    obstacles++;
                }
            } else if (!p.isHorizontal) {
                if (p.x >= Math.min(pieceX, exitCol) && p.x <=
Math.max(pieceX, exitCol) &&
                    p.y <= pieceY && p.y + p.length > pieceY) {
                    obstacles++;
                }
            }
        }

        return obstacles;
    }

    public static int calculateManhattanHeuristic(Board board, Piece
primaryPiece) {
        int dx = Math.abs(primaryPiece.x - board.getExitCol());
        int dy = Math.abs(primaryPiece.y - board.getExitRow()) / 2;
        return dx + dy;
    }

    public static int calculateCombinedHeuristic(Board board, Piece
primaryPiece) {
        int obstacles = calculateObstacleHeuristic(board, primaryPiece);
        int manhattan = calculateManhattanHeuristic(board,
primaryPiece);
        return obstacles + manhattan * 2;
    }

    public static List<Node> solve(Board initialBoard, String
heuristicType) {
        PriorityQueue<Node> openSet = new
PriorityQueue<>(Comparator.comparingInt(n -> n.g + n.h));
        Set<String> visited = new HashSet<>();
        long startTime = System.currentTimeMillis();

        Piece primaryPiece = initialBoard.getPieceMap().get('P');

        int h0 = switch (heuristicType.toLowerCase()) {
```

```
            case "obstacle" -> calculateObstacleHeuristic(initialBoard,
primaryPiece);
            case "combined" -> calculateCombinedHeuristic(initialBoard,
primaryPiece);
            default -> calculateManhattanHeuristic(initialBoard,
primaryPiece);
        };

        Node startNode = new Node(initialBoard, null, null, 0, h0);
        openSet.add(startNode);

        while (!openSet.isEmpty()) {
            Node current = openSet.poll();

            if (current.board.isGoal()) {
                long endTime = System.currentTimeMillis();
                List<Node> solution = Node.reconstructPath(current);
                System.out.println("Solusi ditemukan dalam " + current.g
+ " langkah");
                System.out.println("Waktu pencarian: " + (endTime -
startTime) + " ms");
                return solution;
            }

            String boardKey = current.board.getBoardKey();
            if (visited.contains(boardKey)) continue;
            visited.add(boardKey);

            List<Node> neighbors = Node.generateNextNodes(current);
            for (Node nextNode : neighbors) {
                String nextKey = nextNode.board.getBoardKey();
                if (!visited.contains(nextKey)) {
                    Piece nextPrimary =
nextNode.board.getPieceMap().get('P');
                    int hNext = switch (heuristicType.toLowerCase()) {
                        case "obstacle" ->
calculateObstacleHeuristic(nextNode.board, nextPrimary);
                        case "combined" ->
calculateCombinedHeuristic(nextNode.board, nextPrimary);
```

```
                        default ->
calculateManhattanHeuristic(nextNode.board, nextPrimary);
                    };
                    nextNode.h = hNext;
                    openSet.add(nextNode);
                }
            }
        }

        System.out.println("Tidak ditemukan solusi");
        long endTime = System.currentTimeMillis();
        System.out.println("Waktu pencarian: " + (endTime - startTime) +
" ms");
        return null;
    }
}
```

IDA.java

```
package algorithm;

import java.util.*;
import view.*;

public class IDA {

    public static List<Node> solve(Board initialBoard, String
heuristicType) {
        long startTime = System.currentTimeMillis();

        Piece primaryPiece = initialBoard.getPieceMap().get('P');
        int h0 = getHeuristic(initialBoard, primaryPiece,
heuristicType);
        Node startNode = new Node(initialBoard, null, null, 0, h0);
        int threshold = h0;
```

```
        while (true) {
            Set<String> visited = new HashSet<>();
            Map<String, Integer> gScoreMap = new HashMap<>();

            Result result = search(startNode, threshold, heuristicType,
visited, gScoreMap);

            if (result.found) {
                long endTime = System.currentTimeMillis();
                System.out.println("Solusi ditemukan dalam " +
result.goalNode.g + " langkah");
                System.out.println("Waktu pencarian: " + (endTime -
startTime) + " ms");
                return Node.reconstructPath(result.goalNode);
            }

            if (result.minThreshold == Integer.MAX_VALUE) {
                long endTime = System.currentTimeMillis();
                System.out.println("Tidak ditemukan solusi");
                System.out.println("Waktu pencarian: " + (endTime -
startTime) + " ms");
                return null;
            }

            threshold = result.minThreshold;
        }
    }

    private static Result search(Node node, int threshold, String
heuristicType,
                                 Set<String> visited, Map<String,
Integer> gScoreMap) {
        int f = node.g + node.h;
        String key = node.board.getBoardKey();

        if (gScoreMap.containsKey(key) && gScoreMap.get(key) <= node.g)
{
            return new Result(false, null, Integer.MAX_VALUE);
        }
```

```
        gScoreMap.put(key, node.g);

        if (f > threshold) {
            return new Result(false, null, f);
        }

        if (node.board.isGoal()) {
            return new Result(true, node, f);
        }

        visited.add(key);
        int min = Integer.MAX_VALUE;

        List<Node> children = Node.generateNextNodes(node);

        for (Node child : children) {
            child.g = node.g + 1;
            Piece nextPrimary = child.board.getPieceMap().get('P');
            child.h = getHeuristic(child.board, nextPrimary,
heuristicType);

            Piece currentPrimary = node.board.getPieceMap().get('P');
            String moveDirection = "";
            if (nextPrimary.x > currentPrimary.x) {
                moveDirection = "ke kanan " + (nextPrimary.x -
currentPrimary.x);
            } else if (nextPrimary.x < currentPrimary.x) {
                moveDirection = "ke kiri " + (currentPrimary.x -
nextPrimary.x);
            } else if (nextPrimary.y > currentPrimary.y) {
                moveDirection = "ke bawah " + (nextPrimary.y -
currentPrimary.y);
            } else if (nextPrimary.y < currentPrimary.y) {
                moveDirection = "ke atas " + (currentPrimary.y -
nextPrimary.y);
            }
            child.moveDesc = "Geser P " + moveDirection;
        }
```

```
        children.sort(Comparator.comparingInt(c -> c.g + c.h));

        for (Node child : children) {
            String childKey = child.board.getBoardKey();
            if (visited.contains(childKey)) continue;

            Result result = search(child, threshold, heuristicType,
visited, gScoreMap);
            if (result.found) {
                return result;
            }

            if (result.minThreshold < min) {
                min = result.minThreshold;
            }

            visited.remove(childKey);
        }

        visited.remove(key);
        return new Result(false, null, min);
    }

    private static int getHeuristic(Board board, Piece primaryPiece,
String heuristicType) {
        if (primaryPiece == null) return 0;
        return switch (heuristicType.toLowerCase()) {
            case "obstacle" -> AStar.calculateObstacleHeuristic(board,
primaryPiece);
            case "combined" -> AStar.calculateCombinedHeuristic(board,
primaryPiece);
            default -> AStar.calculateManhattanHeuristic(board,
primaryPiece);
        };
    }

    private static class Result {
        boolean found;
        Node goalNode;
```

```
        int minThreshold;

        Result(boolean found, Node goalNode, int minThreshold) {
            this.found = found;
            this.goalNode = goalNode;
            this.minThreshold = minThreshold;
        }
    }
}
```

InputParser.java

```
package util;

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

public class InputParser {

    public static Board parseFile(String filePath) {
        try (BufferedReader reader = new BufferedReader(new
InputStreamReader(new FileInputStream(filePath), "UTF-8"))) {

            // Baca dimensi board
            String header = reader.readLine();
            if (header == null) {
                System.out.println("File kosong. Tidak bisa memproses
input.");
                return null;
            }

            String[] dimensions = header.trim().split("\\s+");
            int expectedRows = Integer.parseInt(dimensions[0]);
            int expectedCols = Integer.parseInt(dimensions[1]);
```

```
            // Baca jumlah piece
            String countLine = reader.readLine();
            int expectedPieceCount;
            try {
                expectedPieceCount = Integer.parseInt(countLine.trim());
            } catch (NumberFormatException e) {
                System.out.println("Jumlah piece harus berupa angka
valid.");
                return null;
            }

            // Baca representasi papan
            List<String> boardLines = new ArrayList<>();
            int maxWidth = 0;
            String line;
            while ((line = reader.readLine()) != null) {
                if (!line.matches("[A-Za-z\\.K ]*")) {
                    System.out.println("Karakter tidak valid ditemukan
dalam papan.");
                    return null;
                }
                boardLines.add(line);
                maxWidth = Math.max(maxWidth, line.length());
            }

            int exitCol = -1, exitRow = -1;
            for (int row = 0; row < boardLines.size(); row++) {
                String rowStr = boardLines.get(row);
                for (int col = 0; col < rowStr.length(); col++) {
                    if (rowStr.charAt(col) == 'K') {
                        exitCol = col;
                        exitRow = row;
                    }
                }
            }

            if (exitCol == -1 || exitRow == -1) {
                System.out.println("Tidak ditemukan posisi keluar (K)
dalam papan.");
```

```
                return null;
            }

            int actualRows = Math.max(expectedRows, boardLines.size());
            int actualCols = Math.max(expectedCols, maxWidth);

            char[][] grid = new char[actualRows][actualCols];
            for (int i = 0; i < actualRows; i++) {
                Arrays.fill(grid[i], ' ');
            }

            for (int i = 0; i < boardLines.size(); i++) {
                String rowStr = boardLines.get(i);
                for (int j = 0; j < rowStr.length(); j++) {
                    grid[i][j] = rowStr.charAt(j);
                }
            }

            Map<Character, Piece> pieceMap = new HashMap<>();
            Set<Character> processed = new HashSet<>();

            for (int y = 0; y < actualRows; y++) {
                for (int x = 0; x < actualCols; x++) {
                    char symbol = grid[y][x];
                    if (Character.isUpperCase(symbol) && symbol != 'K'
&& !processed.contains(symbol)) {
                        processed.add(symbol);
                        boolean horizontal = (x + 1 < actualCols &&
grid[y][x + 1] == symbol);
                        int length = 1;

                        if (horizontal) {
                            int nextX = x + 1;
                            while (nextX < actualCols && grid[y][nextX]
== symbol) {
                                length++;
                                nextX++;
                            }
                        } else {
```

```
                            int nextY = y + 1;
                            while (nextY < actualRows && grid[nextY][x]
== symbol) {
                                length++;
                                nextY++;
                            }
                        }

                        boolean isMain = (symbol == 'P');
                        pieceMap.put(symbol, new Piece(symbol, x, y,
length, horizontal, isMain));
                    }
                }
            }

            int otherPieceCount = 0;
            for (char key : pieceMap.keySet()) {
                if (key != 'P') otherPieceCount++;
            }

            if (otherPieceCount != expectedPieceCount) {
                System.out.println("Jumlah piece tidak sesuai dengan
angka yang tercantum.");
                return null;
            }

            return new Board(grid, pieceMap, exitCol, exitRow);

        } catch (IOException e) {
            System.out.println("Terjadi kesalahan saat membaca file: " +
e.getMessage());
            return null;
        }
    }
}
```

OutputWriter.java

```java
package util;

import java.awt.Point;
import java.io.*;
import java.util.*;
import view.*;

public class OutputWriter {

    private static final String RED_TEXT = "\u001B[31m";
    private static final String WHITE_BG = "\u001B[47m";
    private static final String BLACK_TEXT = "\u001B[30m";
    private static final String RESET_COLOR = "\u001B[0m";
    private static final String YELLOW_BG = "\u001B[43m";

    public static void displayBoard(char[][] grid) {
        for (char[] row : grid) {
            for (char ch : row) {
                switch (ch) {
                    case 'P' -> System.out.print(RED_TEXT + ch +
RESET_COLOR);
                    case 'K' -> System.out.print(WHITE_BG + BLACK_TEXT +
ch + RESET_COLOR);
                    default -> System.out.print(ch);
                }
            }
            System.out.println();
        }
    }

    public static Map<Character, Set<Point>> extractPositions(char[][]
grid) {
        Map<Character, Set<Point>> positionMap = new HashMap<>();
        for (int row = 0; row < grid.length; row++) {
            for (int col = 0; col < grid[row].length; col++) {
                char symbol = grid[row][col];
                if (symbol != '.') {
                    positionMap.computeIfAbsent(symbol, k -> new
```

```
HashSet<>()).add(new Point(col, row));
                }
            }
        }
        return positionMap;
    }

    public static void displayBoardWithHighlight(char[][] currentGrid,
char[][] prevGrid) {
        Set<Point> changed = new HashSet<>();

        if (prevGrid != null) {
            Map<Character, Set<Point>> prevPos =
extractPositions(prevGrid);
            Map<Character, Set<Point>> currPos =
extractPositions(currentGrid);

            for (char piece : currPos.keySet()) {
                Set<Point> oldPositions = prevPos.getOrDefault(piece,
Collections.emptySet());
                Set<Point> newPositions = currPos.get(piece);

                for (Point p : oldPositions) {
                    if (!newPositions.contains(p)) changed.add(p);
                }
                for (Point p : newPositions) {
                    if (!oldPositions.contains(p)) changed.add(p);
                }
            }
        }

        for (int row = 0; row < currentGrid.length; row++) {
            for (int col = 0; col < currentGrid[row].length; col++) {
                char ch = currentGrid[row][col];
                Point pos = new Point(col, row);
                boolean isChanged = changed.contains(pos);

                if (isChanged) {
                    switch (ch) {
```

```
                        case 'P' -> System.out.print(YELLOW_BG +
RED_TEXT + ch + RESET_COLOR);
                        case 'K' -> System.out.print(YELLOW_BG +
BLACK_TEXT + ch + RESET_COLOR);
                        default -> System.out.print(YELLOW_BG + ch +
RESET_COLOR);
                    }
                } else {
                    switch (ch) {
                        case 'P' -> System.out.print(RED_TEXT + ch +
RESET_COLOR);
                        case 'K' -> System.out.print(WHITE_BG +
BLACK_TEXT + ch + RESET_COLOR);
                        default -> System.out.print(ch);
                    }
                }
            }
            System.out.println();
        }
    }

    public static String convertBoardToString(char[][] grid) {
        StringBuilder builder = new StringBuilder();
        for (char[] row : grid) {
            for (char ch : row) {
                builder.append(ch);
            }
            builder.append(System.lineSeparator());
        }
        return builder.toString();
    }

    public static void writeLinesToFile(String filename, List<String>
lines) {
        try (PrintWriter out = new PrintWriter(new
FileWriter(filename))) {
            for (String line : lines) {
                out.println(line);
            }
```

```
            System.out.println("Berhasil menyimpan output ke " +
filename);
        } catch (IOException e) {
            System.err.println("Gagal menulis file: " + e.getMessage());
        }
    }

    public static void writeBoardToFile(String filename, char[][] grid)
{
        String boardStr = convertBoardToString(grid);
        try (PrintWriter out = new PrintWriter(new
FileWriter(filename))) {
            out.print(boardStr);
            System.out.println("Papan berhasil disimpan ke " +
filename);
        } catch (IOException e) {
            System.err.println("Gagal menyimpan papan ke file: " +
e.getMessage());
        }
    }

    public static void printSolution(List<Node> solutionPath) {
        if (solutionPath == null) {
            System.out.println("Solusi tidak ditemukan.");
            return;
        }

        System.out.println("Konfigurasi awal:");
        displayBoard(solutionPath.get(0).board.getGrid());

        for (int i = 1; i < solutionPath.size(); i++) {
            Node step = solutionPath.get(i);
            System.out.println("\nLangkah ke-" + i + ": " +
step.moveDesc);
            displayBoardWithHighlight(step.board.getGrid(),
solutionPath.get(i - 1).board.getGrid());
        }
    }
```

```
    public static void printSolution(List<Node> solutionPath, long
elapsedTimeMs) {
        if (solutionPath == null) {
            System.out.println("Solusi tidak ditemukan.");
            return;
        }
        System.out.println("Konfigurasi awal:");
        displayBoard(solutionPath.get(0).board.getGrid());
        for (int i = 1; i < solutionPath.size(); i++) {
            Node step = solutionPath.get(i);
            System.out.println("\nLangkah ke-" + i + ": " +
step.moveDesc);
            displayBoardWithHighlight(step.board.getGrid(),
solutionPath.get(i - 1).board.getGrid());
        }
        System.out.println("\nJumlah langkah: " + (solutionPath.size() -
1));
        System.out.println("Waktu pencarian: " + elapsedTimeMs + " ms");
    }

    public static void saveSolution(List<Node> solutionPath, String
outputPath) {
        if (solutionPath == null) {
            System.err.println("Tidak ada solusi untuk disimpan.");
            return;
        }

        List<String> output = new ArrayList<>();
        output.add("Solusi ditemukan dalam " + (solutionPath.size() - 1)
+ " langkah:");
        output.add("Papan awal:");

output.add(convertBoardToString(solutionPath.get(0).board.getGrid()));

        for (int i = 1; i < solutionPath.size(); i++) {
            Node step = solutionPath.get(i);
            output.add("");
            output.add("Langkah ke-" + i + ": " + step.moveDesc);
            output.add(convertBoardToString(step.board.getGrid()));
```

```
        }

        writeLinesToFile(outputPath, output);
    }

    public static void saveSolution(List<Node> solutionPath, String
outputPath, long elapsedTimeMs) {
        if (solutionPath == null) {
            System.err.println("Tidak ada solusi untuk disimpan.");
            return;
        }
        List<String> output = new ArrayList<>();
        output.add("Solusi ditemukan dalam " + (solutionPath.size() - 1)
+ " langkah:");
        output.add("Waktu pencarian: " + elapsedTimeMs + " ms");
        output.add("Papan awal:");

output.add(convertBoardToString(solutionPath.get(0).board.getGrid()));
        for (int i = 1; i < solutionPath.size(); i++) {
            Node step = solutionPath.get(i);
            output.add("");
            output.add("Langkah ke-" + i + ": " + step.moveDesc);
            output.add(convertBoardToString(step.board.getGrid()));
        }
        writeLinesToFile(outputPath, output);
    }
}
```

Main.java

```
package main;

import algorithm.*;
import java.util.*;
import view.*;
import util.*;
```

```java
public class Main {

    public static void main(String[] args) {
        try (Scanner scanner = new Scanner(System.in)) {

            System.out.print("Masukkan nama file input (contohnya
tc1.txt): ");
            String filename = scanner.nextLine();
            String inputPath = "test/" + filename;

            Board start = InputParser.parseFile(inputPath);
            if (start == null) {
                System.out.println("Gagal memuat papan.");
                return;
            }
            System.out.println("Papan berhasil dibaca.");

            int algoChoice = askForInt(scanner,
                "\nPilih algoritma pencarian:\n" +
                "1. Greedy Best First Search\n" +
                "2. Uniform Cost Search\n" +
                "3. A* Search\n" +
                "4. IDA* Search\n" +
                "Masukkan pilihan (1-4): ", 1, 4);

            List<Node> path = null;
            String heuristic = null;
            long startTime = System.currentTimeMillis();

            switch (algoChoice) {
                case 1 -> {
                    System.out.println("Greedy Best First Search
dipilih.");
                    heuristic = askGBFSHeuristic(scanner);
                    path = GBFS.solve(start, heuristic);
                }
                case 2 -> {
                    System.out.println("Uniform Cost Search dipilih.");
                    path = UCS.solve(start);
```

```
                }
                case 3 -> {
                    System.out.println("A* Search dipilih.");
                    heuristic = askAStarHeuristic(scanner);
                    path = AStar.solve(start, heuristic);
                }
                case 4 -> {
                    System.out.println("IDA* Search dipilih.");
                    heuristic = askAStarHeuristic(scanner);
                    path = IDA.solve(start, heuristic);
                }
            }

            long endTime = System.currentTimeMillis();
            long elapsedTimeMs = endTime - startTime;

            OutputWriter.printSolution(path, elapsedTimeMs);

            if (path != null) {
                System.out.print("\nApakah ingin menyimpan hasil ke
file? (y/n): ");
                String saveChoice =
scanner.nextLine().trim().toLowerCase();
                if (saveChoice.equals("y")) {
                    System.out.print("Masukkan nama file simpanan
(misal: solusi.txt): ");
                    String outputFile = scanner.nextLine().trim();
                    if (!outputFile.contains("/")) {
                        outputFile = "test/" + outputFile;
                    }
                    OutputWriter.saveSolution(path, outputFile,
elapsedTimeMs);
                }
            }
        }
    }

    private static int askForInt(Scanner scanner, String prompt, int
min, int max) {
```

```
        int choice = -1;
        while (choice < min || choice > max) {
            System.out.print(prompt);
            try {
                choice = Integer.parseInt(scanner.nextLine());
                if (choice < min || choice > max) {
                    System.out.println("Input tidak valid! Silakan
masukkan angka antara " + min + "-" + max + ".");
                }
            } catch (NumberFormatException e) {
                System.out.println("Input tidak valid! Silakan masukkan
angka antara " + min + "-" + max + ".");
            }
        }
        return choice;
    }

    private static String askGBFSHeuristic(Scanner scanner) {
        int hChoice = askForInt(scanner,
            "Pilih heuristik:\n" +
            "1. Manhattan Distance Heuristic\n" +
            "2. Blocking Cars Heuristic\n" +
            "Masukkan pilihan (1-2): ", 1, 2);
        return (hChoice == 2) ? "blockingcars" : "manhattan";
    }

    private static String askAStarHeuristic(Scanner scanner) {
        int hChoice = askForInt(scanner,
            "Pilih heuristik:\n" +
            "1. Manhattan Distance Heuristic\n" +
            "2. Obstacle Heuristic\n" +
            "3. Combined Heuristic\n" +
            "Masukkan pilihan (1-3): ", 1, 3);
        return switch (hChoice) {
            case 2 -> "obstacle";
            case 3 -> "combined";
            default -> "manhattan";
        };
    }
```

```
}
```

BAB IV
PENGUJIAN DAN ANALISIS

PENGUJIAN

Test Case 1 :

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

Hasil Pengujian Test Case 1

UCS	Langkah ke-5: Geser P ke kanan 4 AABCD. ..BCD. G...PP GHIIIF GHJ..F LLJMMF Jumlah langkah: 5 Waktu pencarian: 68 ms

GBFS + Manhattan	Langkah ke-136: Geser P ke kanan 1 ...AA. GHBC.. GHBC.PP GIIIDF ..J.DF LLJMMF Jumlah langkah: 136 Waktu pencarian: 3722 ms
GBFS + Blocking	Langkah ke-408: Geser P ke kanan 2 .AACD. GHBCD. GHB..PP GIII.F ..J..F LLJMMF Jumlah langkah: 408 Waktu pencarian: 656 ms
A* + Manhattan	Langkah ke-5: Geser P ke kanan 4 AABCD. ..BCD. G...PP GHIIIF GHJ..F LLJMMF Jumlah langkah: 5 Waktu pencarian: 816 ms
A* + Obstacle	Langkah ke-5: Geser P ke kanan 4 AABCD. ..BCD. G...PP GHIIIF GHJ..F LLJMMF Jumlah langkah: 5 Waktu pencarian: 1599 ms

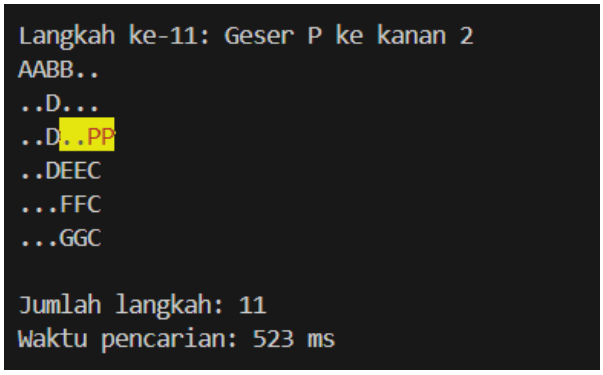
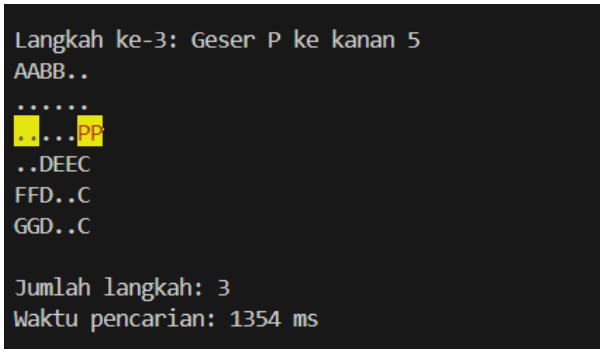
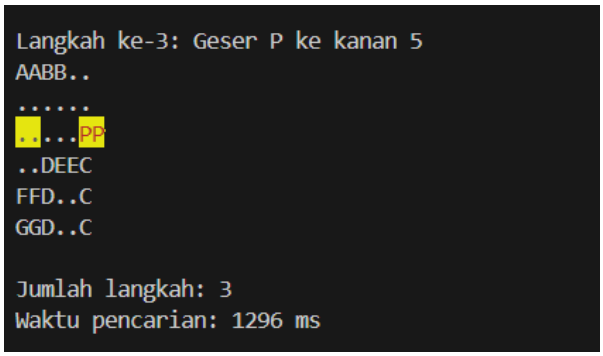
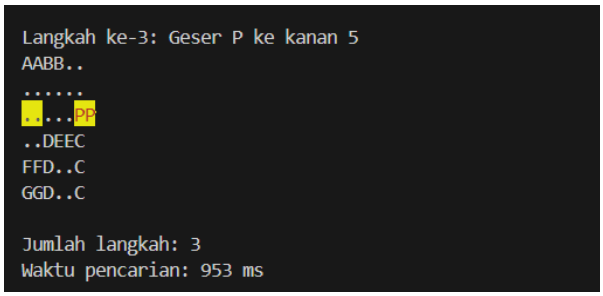
A* + Combined	Langkah ke-5: Geser P ke kanan 4 AABCD. ..BCD. G...PP GHIIIF GHJ..F LLJMMF Jumlah langkah: 5 Waktu pencarian: 954 ms
IDA* + Manhattan	Langkah ke-5: Geser P ke kanan 4 AABCD. ..BCD. G...PP GHIIIF GHJ..F LLJMMF Jumlah langkah: 5 Waktu pencarian: 797 ms
IDA* + Obstacle	Langkah ke-5: Geser P ke kanan 4 AABCD. ..BCD. G...PP GHIIIF GHJ..F LLJMMF Jumlah langkah: 5 Waktu pencarian: 913 ms
IDA* + Combined	Langkah ke-5: Geser P ke kanan 4 AABCD. ..BCD. G...PP GHIIIF GHJ..F LLJMMF Jumlah langkah: 5 Waktu pencarian: 1364 ms

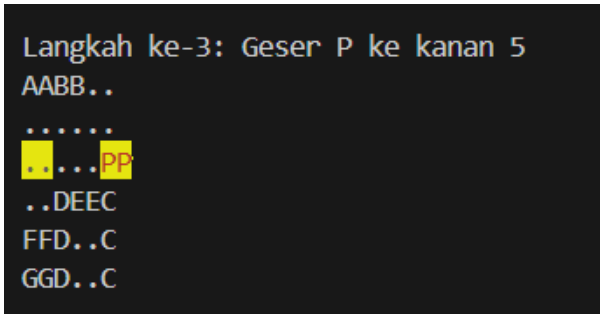
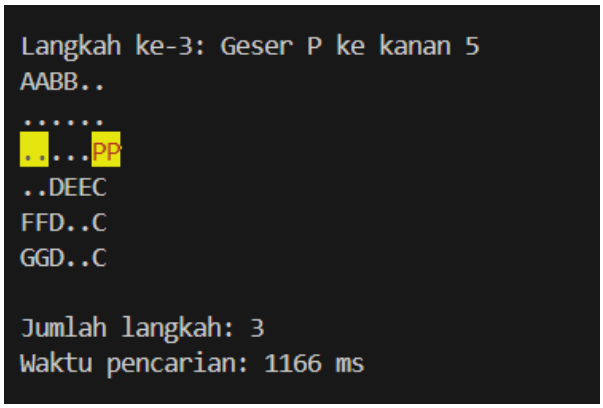
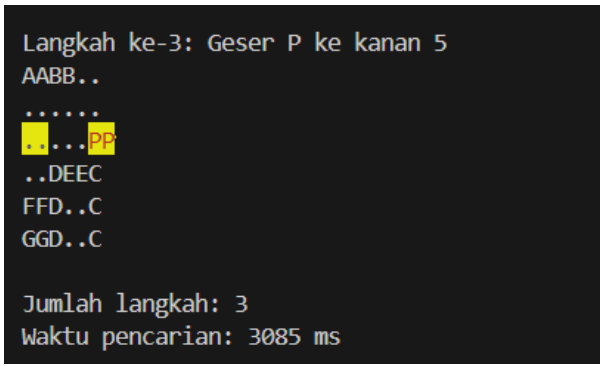
Test Case 2 :

```
6 6
7
AABB.C
..D..C
PPD..CK
..DEE.
FF....
GG....
```

Hasil Pengujian Test Case 2

UCS	Langkah ke-3: Geser P ke kanan 5 AABB.. ....PP ..DEEC FFD..C GGD..C Jumlah langkah: 3 Waktu pencarian: 63 ms
GBFS + Manhattan	Langkah ke-347: Geser P ke kanan 1 AABB.. ....D..PP EED..C ..DFFC ..GG.C Jumlah langkah: 347 Waktu pencarian: 1359 ms

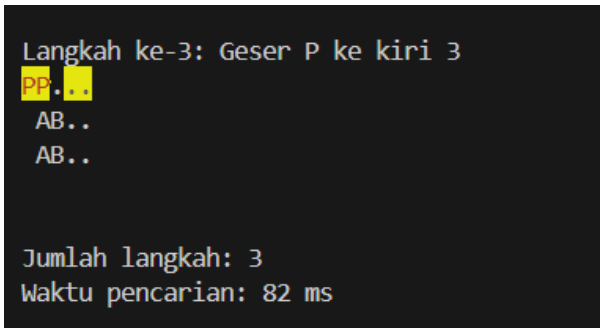
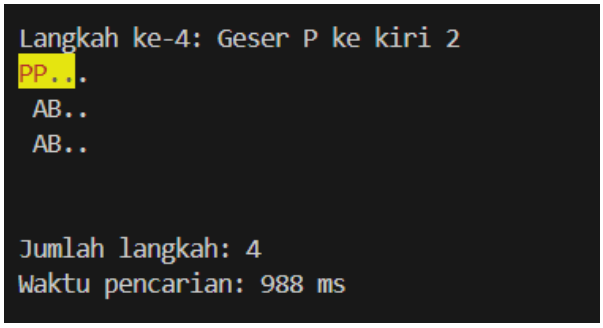
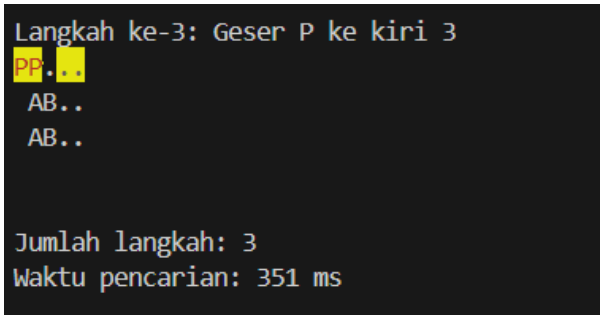
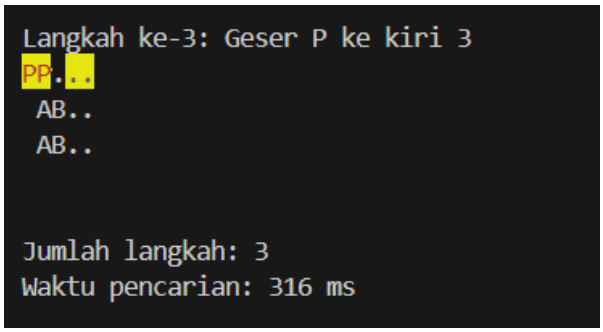
GBFS + Blocking	
A* + Manhattan	
A* + Obstacle	
A* + Combined	

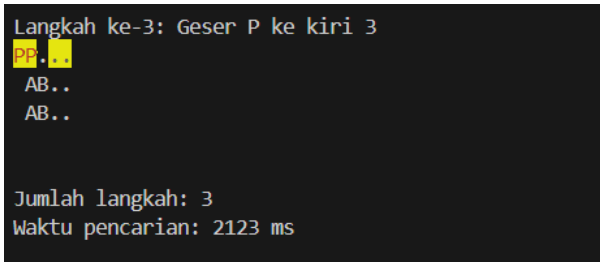
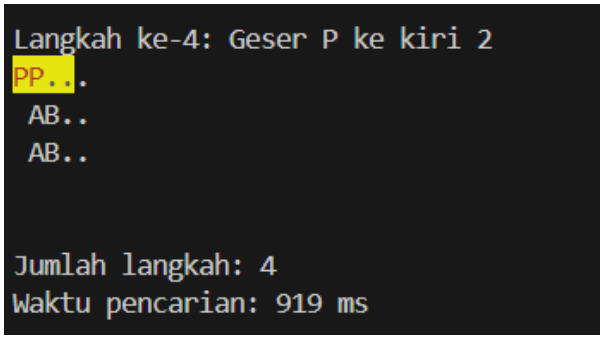
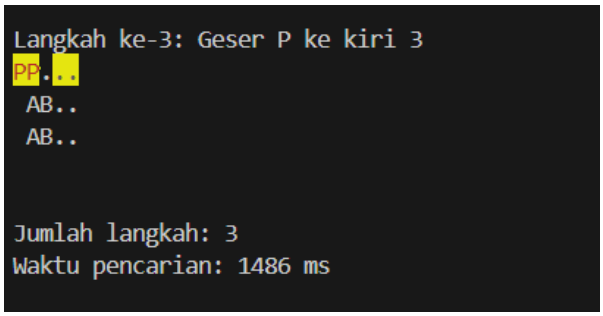
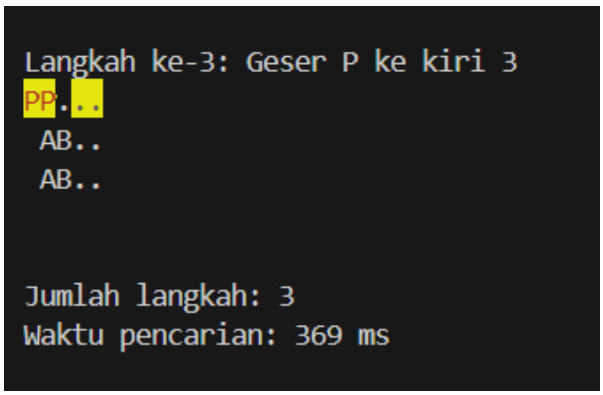
IDA* + Manhattan	
IDA* + Obstacle	
IDA* + Combined	

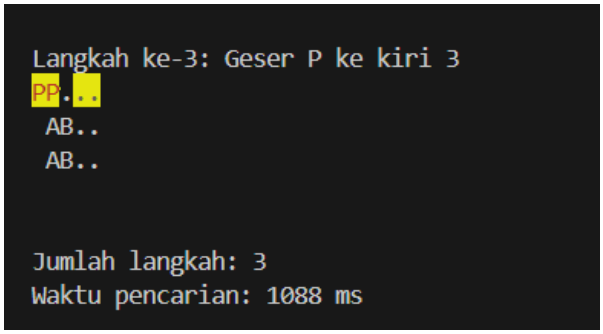
Test Case 3 :

```
4 3
2
KABPP
 AB..
 ....
```

Hasil Pengujian Test Case 3

UCS	
GBFS + Manhattan	
GBFS + Blocking	
A* + Manhattan	

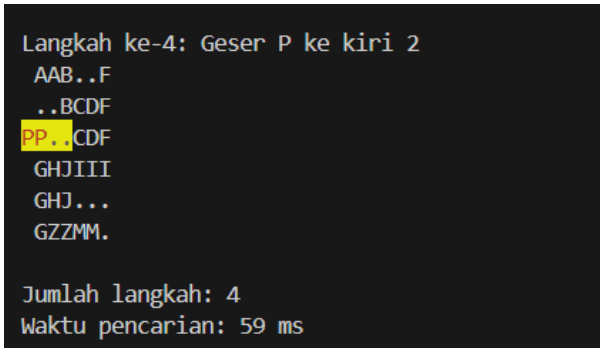
A* + Obstacle	
A* + Combined	
IDA* + Manhattan	
IDA* + Obstacle	

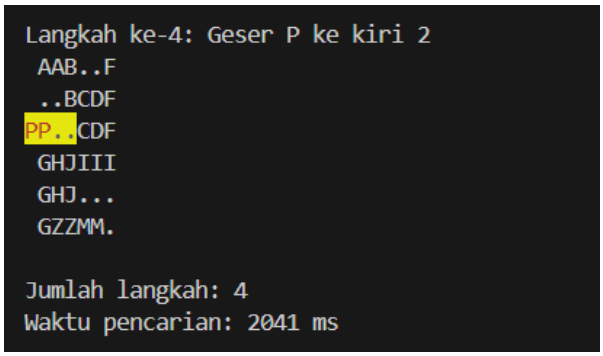
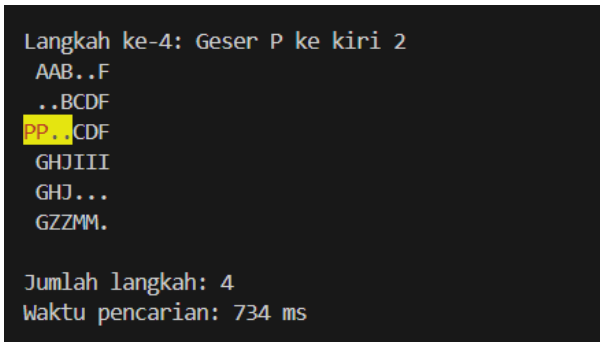
IDA* + Combined	

Test Case 4 :

```
6 6
11
 AAB..F
 ..BCDF
KGPPCDF
 GH.III
 GHJ...
 ZZJMM.
```

Hasil Pengujian Test Case 4

UCS	

GBFS + Manhattan	
GBFS + Blocking	
A* + Manhattan	
A* + Obstacle	

A* + Combined	Langkah ke-4: Geser P ke kiri 2 AAB..F ..BCDF PP..CDF GHJIII GHJ... GZZMM. Jumlah langkah: 4 Waktu pencarian: 838 ms
IDA* + Manhattan	Langkah ke-4: Geser P ke kiri 2 AAB..F ..BCDF PP..CDF GHJIII GHJ... GZZMM. Jumlah langkah: 4 Waktu pencarian: 579 ms
IDA* + Obstacle	Langkah ke-4: Geser P ke kiri 2 AAB..F ..BCDF PP..CDF GHJIII GHJ... GZZMM. Jumlah langkah: 4 Waktu pencarian: 367 ms
IDA* + Combined	Langkah ke-4: Geser P ke kiri 2 AAB..F ..BCDF PP..CDF GHJIII GHJ... GZZMM. Jumlah langkah: 4 Waktu pencarian: 1271 ms

Hasil Analisis

Dari percobaan yang telah dilakukan dapat dilihat bahwa UCS atau Uniform Cost Search merupakan algoritma dengan waktu pencarian dan langkah yang diambil paling sedikit. Hal ini disebabkan karena algoritma ini telah menghitung terlebih dahulu biaya total dari awal hingga akhir sehingga berhasil menjadi pathfinding terbaik.

Untuk GBFS atau Greedy Best First Search, untuk kasus yang rumit, akan membutuhkan jumlah waktu dan langkah yang cukup banyak. Hal ini terjadi karena GBFS cenderung mengabaikan jumlah langkah yang diambil dan justru mencoba untuk mengambil jalan yang paling menjanjikan secara heuristik. Dengan begitu, langkah yang diambil akan lebih banyak dan waktu yang berlalu sudah pasti lebih banyak. Maka dari itu, langkah yang diambil oleh GBFS belum tentu optimal seperti pada test case 1 dan 4 yang cukup rumit, dibanding test case 2 dan 3 yang jumlah langkahnya tidak berbeda jauh.

Untuk A*, akan hampir selalu memberikan hasil yang optimal karena dalam algoritmanya, A* akan memperhitungkan seberapa mahal atau cost dari langkah yang akan diambil selanjutnya sehingga dengan begitu, A* akan memilih opsi yang terpendek sehingga langkahnya pun akan sedikit

Untuk IDA, sebagai pengembangan dari A* juga akan memberikan hasil yang lumayan optimal. Hal ini disebabkan karena menggunakan basis dari A* yang akan memilih opsi untuk jalur terpendeknya, sehingga langkah yang dihasilkan sedikit.

Setiap algoritma memiliki kelebihan dan kekurangannya masing-masing, dan untuk Rush Hour ini, paling baik menggunakan algoritma UCS.

Dari heuristik yang digunakan, tergantung dari algoritmanya, akan menghasilkan waktu yang lebih pendek atau lama namun untuk langkahnya sendiri masih tidak jauh berbeda atau cenderung sama. Untuk Manhattan, karena hanya mengukur jarak ke tempat keluar, maka kemungkinan besar akan memakan waktu yang cukup lama karena tidak memperhitungkan blocking yang terdapat dalam rutenya. Untuk Blocking cars, bertujuan untuk mengurangi obstacle yang berada dalam rute sehingga cenderung akan lebih mudah mencari jalan keluar karena dengan berkurangnya obstacle, maka primary piece dapat bergerak lebih banyak.

BAB V
KESIMPULAN DAN SARAN

Kesimpulan

Melalui pengerjaan tugas besar ini, program penyelesaian puzzle Rush Hour dengan algoritma pathfinding telah berhasil diimplementasikan dan dibuat dengan bahasa pemrograman Java. Algoritma yang diimplementasikan berupa Uniform Cost Search, Greedy Best-First Search, A*, dan Iterative Deepening A* dan heuristik yang digunakan adalah Manhattan Distance dan Blocking Vehicle. Seluruh algoritma dan heuristik dapat memberikan solusi jika ada dan untuk algoritma terbaik, adalah UCS atau Uniform Cost Search yang dapat dilihat dari hasil percobaan.

Saran

Saran untuk penulis sendiri adalah untuk dapat belajar membagi waktu dengan lebih baik lagi agar tidak terjadi hal-hal yang tidak diinginkan

LAMPIRAN

Tautan Repository

https://github.com/KennethRicardoC/Tucil3_13523022

Tabel Checklist

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