

**LAPORAN TUGAS KECIL 3**  
**MATA KULIAH IF2211 STRATEGI ALGORITMA**  
**Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding**



**Dosen Pengampu:**  
Dr. Nur Ulfa Maulidevi, S.T, M.Sc.

**Disusun Oleh:**

Abdullah Farhan	13523042
Adhimas Aryo Bimo	13523052

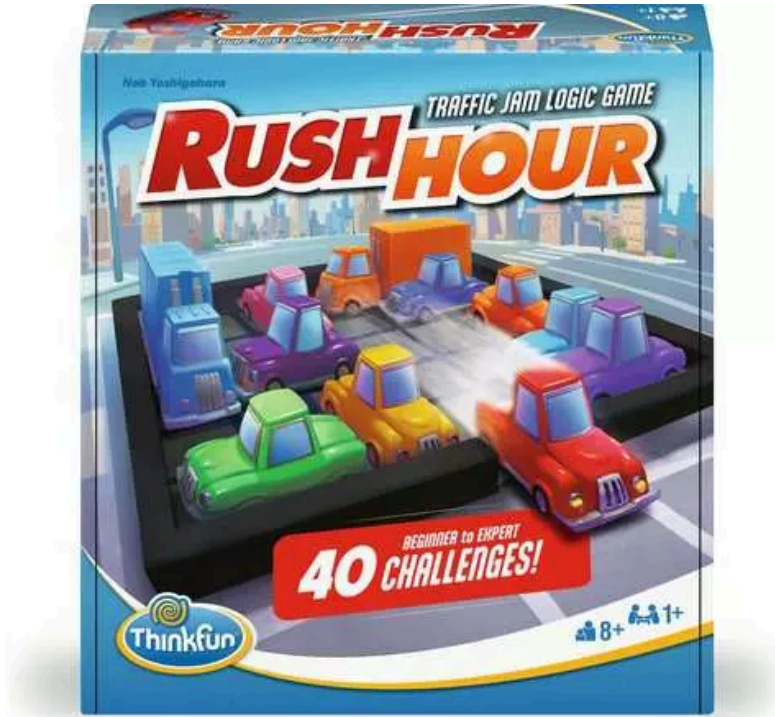
**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA**  
**INSTITUT TEKNOLOGI BANDUNG**  
**MAY 2025**

## DAFTAR ISI

<b>DAFTAR ISI.....</b>	<b>1</b>
<b>BAB I LATAR BELAKANG.....</b>	<b>1</b>
I.I. Deskripsi Tugas.....	1
I.II. Spesifikasi.....	4
<b>BAB II LANDASAN TEORI.....</b>	<b>5</b>
II.I. Uniform Cost Search (UCS).....	5
II.II. Greedy Best First Search (GBFS).....	5
II.III. Algoritma A*.....	6
II.IV. Iterative Deepening A* (IDA*).....	6
II.V. Genetic Algorithm (GA).....	7
<b>BAB III ANALISIS PEMECAHAN MASALAH.....</b>	<b>10</b>
III.I. Pemetaan Masalah.....	10
III.II. Fungsi Heuristik.....	11
III.III. Algoritma UCS dalam Program.....	11
III.IV. Algoritma GBFS dalam Program.....	13
III.V. Algoritma A* dalam Program.....	14
III.VI. Algoritma IDA* dalam Program.....	16
III.VII. Algoritma GA dalam program.....	17
<b>BAB IV ANALISIS DAN PERCOBAAN.....</b>	<b>22</b>
IV.I. Inisialisasi Percobaan.....	22
IV.II. Percobaan dengan UCS.....	23
IV.III. Percobaan dengan GBFS.....	23
IV.IV. Percobaan dengan A*.....	26
IV.V. Percobaan dengan IDA*.....	28
IV.VI. Percobaan dengan GA.....	29
IV.VII. Analisis Algoritma.....	31
IV.VIII. Analisis Kompleksitas.....	32
1. Uniform Cost Search (UCS).....	32
2. Greedy Best First Search (GBFS).....	33
3. A* Search.....	34
4. Iterative Deepening A* (IDA*).....	34
5. Genetic Algorithm (GA).....	35
<b>LAMPIRAN.....</b>	<b>36</b>

## BAB I LATAR BELAKANG

### I.I. 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 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.

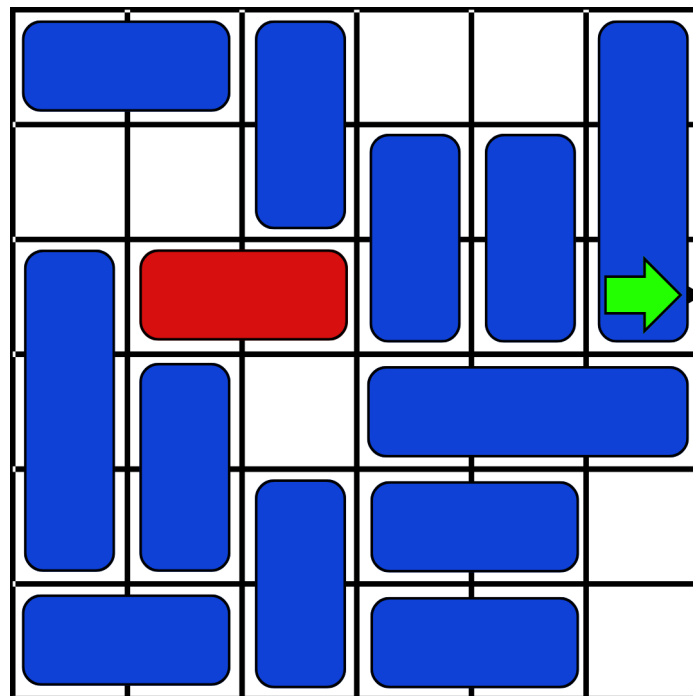
*Papan* terdiri atas *cell*, yaitu sebuah *singular point* dari papan. Sebuah *piece* akan menempati *cell-cell* pada papan. Ketika permainan dimulai, semua *piece* telah diletakkan di dalam papan dengan konfigurasi tertentu berupa lokasi *piece* dan *orientasi*, antara *horizontal* atau *vertikal*.

**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.

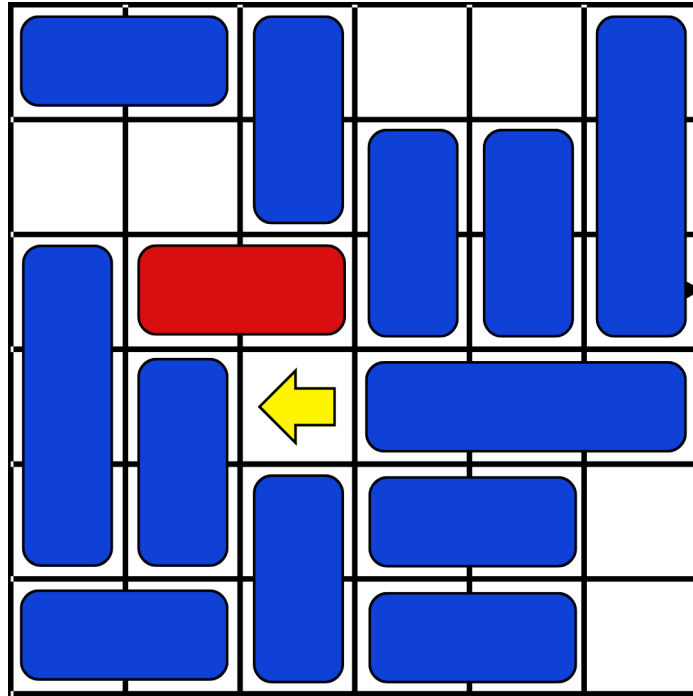
Ilustrasi kasus :

Diberikan sebuah *papan* berukuran 6 x 6 dengan 12 *piece* kendaraan dengan 1 *piece* merupakan *primary piece*. *Piece* ditempatkan pada *papan* dengan posisi dan orientasi sebagai berikut.



Gambar 2. Awal Permainan Game Rush Hour

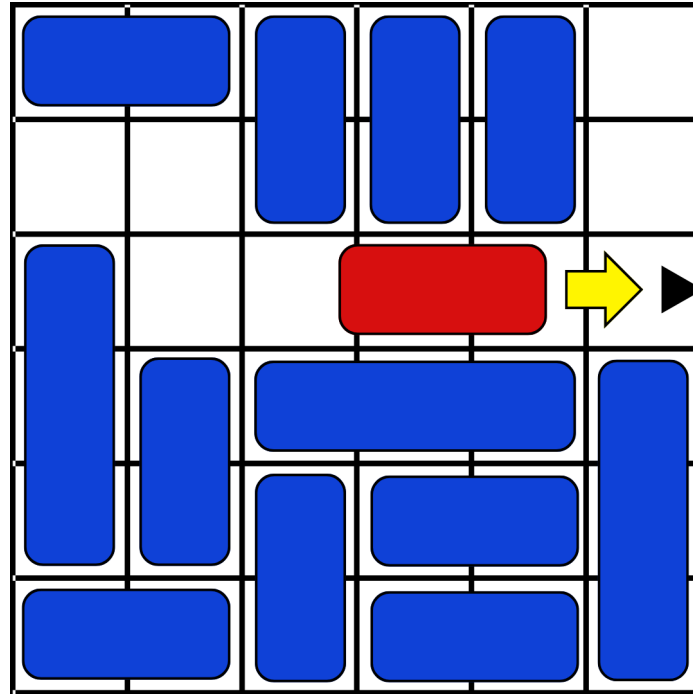
Pemain dapat menggeser-geser *piece* (termasuk *primary piece*) untuk membentuk jalan lurus antara *primary piece* dan *pintu keluar*.



Gambar 3. Gerakan Pertama Game Rush Hour

Gambar 4. Gerakan Kedua Game Rush Hour

Puzzle berikut dinyatakan telah selesai apabila *primary piece* dapat digeser keluar papan melalui *pintu keluar*.



Gambar 5. Pemain Menyelesaikan Permainan

Agar lebih jelas, silahkan amati video cara bermain berikut:

▶ The New Rush Hour by ThinkFun!

Anda juga dapat melihat gif berikut untuk melihat contoh permainan [Rush Hour Solution](#).

## I.II. Spesifikasi

1. Buatlah program sederhana dalam bahasa **C/C++/Java/Javascript** yang mengimplementasikan **algoritma pathfinding Greedy Best First Search, UCS (Uniform Cost Search), dan A\*** dalam menyelesaikan permainan Rush Hour.
2. Tugas dapat dikerjakan **individu atau berkelompok** dengan anggota **maksimal 2 orang** (sangat disarankan). Boleh lintas kelas dan lintas kampus, tetapi **tidak boleh sama** dengan anggota kelompok pada **tugas kecil Strategi Algoritma sebelumnya**.
3. Algoritma *pathfinding* minimal menggunakan **satu heuristic** (2 atau lebih jika mengerjakan *bonus*) yang ditentukan sendiri. Jika mengerjakan *bonus*, *heuristic* yang digunakan ditentukan berdasarkan input pengguna.
4. Algoritma dijalankan secara terpisah. Algoritma yang digunakan ditentukan berdasarkan Input pengguna.

## BAB II LANDASAN TEORI

### II.I. Uniform Cost Search (UCS)

Uniform Cost Search adalah algoritma pencarian jalur yang mencari jalur dengan total biaya terendah dari simpul awal ke simpul tujuan, tanpa menggunakan estimasi heuristik. UCS merupakan versi uninformed search dari algoritma A\*, karena hanya menggunakan biaya aktual yang telah ditempuh ( $g(n)$ ).

UCS akan menjamin solusi optimal selama semua biaya lintasan tidak negatif. Akan tetapi, hal tersebut membuat algoritma ini menjadi lambat karena tidak menggunakan heuristik dan menjelajahi banyak simpul sebelum mencapai tujuan.

Fungsi Evaluasi UCS adalah sebagai berikut:

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

Di mana:

- $g(n)$  adalah total biaya dari simpul awal ke simpul  $n$ .

Langkah-langkah UCS :

1. Masukkan simpul awal A ke dalam *priority queue* dengan biaya 0
2. Ambil simpul dengan biaya terkecil dari queue
3. Jika simpul tersebut adalah G (tujuan), selesai.
4. Jika tidak, perluas simpul dan tambahkan tetangganya ke dalam queue, dengan total biaya kumulatifnya.
5. Ulangi langkah 2–4.

### II.II. Greedy Best First Search (GBFS)

Greedy Best First Search adalah algoritma pencarian jalur yang hanya mempertimbangkan estimasi jarak dari simpul saat ini ke tujuan (heuristik), tanpa memperhatikan biaya dari simpul awal. GBFS berusaha untuk "*serakah*", langsung memilih simpul yang tampak paling dekat ke tujuan berdasarkan nilai heuristik  $h(n)$ .

Fungsi Evaluasi GBFS adalah sebagai berikut:

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

Di mana:

- $h(n)$  adalah estimasi jarak dari simpul  $n$  ke tujuan.

Langkah-Langkah Algoritma GBFS

1. Masukkan simpul awal (A) ke dalam open list
2. Pilih simpul dari open list dengan nilai  **$h(n)$**  terkecil.

3. Jika simpul tersebut adalah tujuan (G), selesai.
4. Jika tidak, perluas simpul tersebut (tambahkan tetangganya ke open list).
5. Ulangi langkah 2–4.

### II.III. Algoritma A\*

A\* adalah algoritma pencarian yang menggabungkan kelebihan UCS (memperhatikan biaya sebenarnya) dan GBFS (menggunakan heuristik). A\* mencari jalur dengan total estimasi biaya terkecil dari awal ke tujuan.

Fungsi Evaluasi Algoritma A\* adalah sebagai berikut:

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

Di mana:

- $h(n)$  adalah estimasi jarak dari simpul n ke tujuan.
- $g(n)$  adalah biaya sebenarnya dari simpul awal ke n

Langkah-langkah Algoritma A\* :

1. Masukkan simpul awal ke open list dengan  $f(n) = g(n) + h(n)$ .
2. Ambil simpul dengan  $f(n)$  terendah.
3. Jika itu simpul tujuan, selesai.
4. Perluas simpul, hitung  $g(n)$ ,  $h(n)$ , dan  $f(n)$  untuk tiap tetangganya.
5. Jika tetangga belum di-explore atau  $f(n)$  lebih kecil, masukkan ke open list.
6. Ulangi.

### II.IV. Iterative Deepening A\* (IDA\*)

Iterative Deepening A\* (IDA\*) adalah algoritma pencarian jalur yang menggabungkan kelebihan A\* dengan teknik iterative deepening. IDA\* mencari jalur dengan total estimasi biaya terkecil dari awal ke tujuan, dengan menggunakan teknik iterative deepening untuk meningkatkan efisiensi.

Fungsi Evaluasi IDA\* adalah sebagai berikut:

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

Di mana:

- $h(n)$  adalah estimasi jarak dari simpul n ke tujuan
- $g(n)$  adalah biaya sebenarnya dari simpul awal ke n

Langkah-Langkah Algoritma IDA\* :

1. Tentukan batas awal (threshold) untuk biaya estimasi.



2. Jalankan algoritma Depth-Limited Search (DLS) dengan batas biaya estimasi yang ditentukan.
3. Jika DLS menemukan solusi, selesai.
4. Jika DLS tidak menemukan solusi, tingkatkan batas biaya estimasi dan ulangi langkah 2.
5. Ulangi langkah 2-4 sampai solusi ditemukan atau batas biaya estimasi melebihi nilai maksimum.

DLS adalah algoritma pencarian yang membatasi kedalaman pencarian dengan menggunakan batas biaya estimasi. DLS mencari jalur dengan total estimasi biaya terkecil dari awal ke tujuan, dengan menggunakan batas biaya estimasi yang ditentukan.

Fungsi Evaluasi DLS adalah sebagai berikut:  $f(n) = g(n) + h(n)$  di mana:  $h(n)$  adalah estimasi jarak dari simpul  $n$  ke tujuan.  $g(n)$  adalah biaya sebenarnya dari simpul awal ke  $n$

Langkah-Langkah Algoritma DLS :

- A. Masukkan simpul awal ke dalam stack dengan biaya estimasi 0.
- B. Ambil simpul dengan biaya estimasi terendah dari stack.
- C. Jika simpul tersebut adalah tujuan, selesai.
- D. Jika tidak, perluas simpul dan tambahkan tetangganya ke dalam stack, dengan total biaya kumulatifnya.
- E. Ulangi langkah 2-4 sampai stack kosong atau batas biaya estimasi melebihi nilai maksimum.

Dengan menggunakan teknik iterative deepening, IDA\* dapat meningkatkan efisiensi pencarian dengan membatasi kedalaman pencarian dan meningkatkan batas biaya estimasi secara bertahap.

## **II.V. Genetic Algorithm (GA)**

Genetic Algorithm (GA) adalah algoritma pencarian dan optimasi yang terinspirasi oleh prinsip-prinsip evolusi biologis, seperti seleksi alam, pewarisan genetik, rekombinasi (crossover), dan mutasi. GA digunakan untuk menyelesaikan masalah kompleks dengan

ruang pencarian yang sangat besar, di mana pendekatan brute-force atau algoritma konvensional kurang efisien.

GA tidak memerlukan pengetahuan awal tentang solusi optimal dan bekerja dengan populasi solusi (bukan satu solusi tunggal), yang secara bertahap berkembang menuju solusi terbaik melalui proses evolusi.

GA menggunakan *fungsi fitness* untuk mengevaluasi seberapa baik suatu solusi (individu) dalam menyelesaikan masalah yang diberikan.

Fungsi Evaluasi Genetic Algorithm adalah sebagai berikut:

$$f_{fitness}(x)$$

Di mana:

- $x$  adalah representasi individu (solusi kandidat) dalam populasi.
- $f_{fitness}(x)$  mengembalikan skor fitness dari individu tersebut.
- Nilai  $f_{fitness}(x)$  dapat berupa nilai maksimisasi (semakin besar semakin baik) atau minimisasi (semakin kecil semakin baik, tergantung tujuan optimasi).

Langkah-langkah algoritma GA adalah sebagai berikut :

1. Inisialisasi Populasi  
Buat populasi awal secara acak yang berisi sejumlah individu (solusi potensial).
2. Evaluasi Fitness  
Hitung nilai fitness untuk setiap individu dalam populasi.
3. Seleksi  
Pilih individu-individu terbaik berdasarkan nilai fitness untuk menjadi "induk" (parent). Teknik seleksi bisa berupa:
  - Roulette Wheel Selection
  - Tournament Selection
  - Rank Selection
4. Crossover (Rekombinasi)  
Gabungkan pasangan induk untuk menghasilkan anak (offspring) baru. Crossover dilakukan untuk menukar bagian gen antar induk, dengan tujuan mewariskan sifat baik dari kedua orang tua.
5. Mutasi  
Lakukan perubahan acak kecil pada beberapa individu untuk menjaga keragaman populasi dan menghindari konvergensi prematur.

6. Pembentukan Populasi Baru

Bentuk populasi generasi berikutnya dari hasil crossover dan mutasi.

7. Pengulangan (Iterasi)

Ulangi langkah 2–6 untuk beberapa generasi sampai salah satu kondisi berikut tercapai:

- Solusi optimal ditemukan
- Jumlah generasi maksimum tercapai
- Nilai fitness tidak meningkat secara signifikan dalam beberapa generasi terakhir

## BAB III ANALISIS PEMECAHAN MASALAH

### III.I. Pemetaan Masalah

Tujuan dari permainan ini adalah untuk menemukan solusi langkah-langkah agar mobil tujuan dapat mencapai titik keluar yang ditentukan. Mobil dipastikan hanya dapat bergerak sesuai dengan orientasinya yang berarti mobil tidak dapat melakukan arah gerak belok ataupun miring. Berarti dapat dipastikan bahwa mobil target akan selalu mengarah ke posisi akhir sesuai dengan orientasinya.

Dari pengetahuan tersebut dapat dipecah tiap gerakan mobil yang valid menjadi suatu kondisi (*State*) dari langkah yang dipilih. Misal dalam kondisi awal, terdapat berbagai gerakan yang mungkin, kita dapat memilih berbagai gerakan tersebut secara acak dan melakukan pengembalian kondisi jika melakukan kesalahan atau tidak menemukan solusi pada langkah tersebut. Oleh karena itu, algoritma *pathfinding* dapat diterapkan untuk menentukan langkah-langkah tersebut. Solusi yang dihasilkan adalah kumpulan langkah untuk mencapai target, maka untuk mendapatkan solusi yang optimal langkah tersebut merupakan langkah yang paling sedikit untuk mencapai titik keluar.

Dalam memetakan persoalan tersebut, fungsi pencarian langkah akan melakukan pencarian pada setiap kondisi (*State*) yang memiliki langkah valid. Dalam hal ini kami menyimpan kondisi yang valid dalam sebuah list yang dibentuk dalam metode *generateSucc()*. List tersebut akan dipakai pada metode pencarian. Untuk menentukan solusi yang optimal, tiap *State* memiliki *cost*-nya masing-masing. *cost* tersebut akan menjadi penentu *State* mana yang akan dipilih dalam pencarian solusi. *State* yang dipilih adalah *State* yang memiliki *cost* terkecil dari berbagai kemungkinan *State* yang valid.

Berikut struktur data *State* dalam program :

```
public class State {
    public Map<Character, Vehicle> vehicles;
    public int tot_rows;
    public int tot_cols;
    public char[][] board;
    public State parent;
    public String move;
    public int cost;
    public int exitRow, exitCol;
    public int heuristicCost;
    public String methode;
    ...
}
```

terdapat 2 *cost* yang kami simpan, *heuristicCost* dan *cost*. *heuristicCost* dihitung berdasarkan *methode* heuristik yang dipilih, sedangkan *cost* merupakan jumlah langkah dari *State* awal hingga ke *State* tersebut.

### III.II. Fungsi Heuristik

Dalam algoritma tertentu yang menggunakan heuristik dalam fungsi evaluasinya, kami membuat beberapa fungsi heuristik untuk menentukan nilai estimasi heuristik. Terdapat 5 fungsi heuristik yang kami gunakan, yakni :

1. Menghitung Jarak Manhattan  
Mengestimasi jarak dari posisi saat ini ke titik tujuan berdasarkan jarak Manhattan (selisih absolut pada sumbu x dan y).
2. Menghitung Kendaraan  
Menghitung banyaknya kendaraan yang menghalangi lintasan kendaraan target menuju titik keluar.
3. Menggabungkan poin 1 dan 2  
Mengombinasikan jarak Manhattan dengan jumlah kendaraan penghalang untuk memperoleh estimasi yang lebih akurat.
4. Menghitung Jarak Chebyshev  
Menggunakan jarak Chebyshev, yaitu jarak maksimum dari perbedaan koordinat x dan y antar dua titik.
5. Melakukan Pembobotan pada poin 1,2,3 dan 4  
Menggabungkan keempat heuristik sebelumnya dengan memberikan bobot tertentu pada masing-masing komponen untuk menghasilkan estimasi heuristik yang seimbang dan lebih informatif.

### III.III. Algoritma UCS dalam Program

Kode dalam program adalah sebagai berikut:

```
package main;

public class UCS extends BaseSearch {
    public PriorityQueue<State> queue;
    public Set<State> visited;
    public State goalState;

    public UCS(State initState) {
        super(initState);
        this.queue = new PriorityQueue<>(Comparator.comparingInt(s ->
```

```

s.cost));
    this.visited = new HashSet<>();
    this.goalState = null;
}

@Override
public void search() {
    queue.add(initState);
    while (!queue.isEmpty()) {
        State currState = queue.poll();
        visitedNodesCount++;

        if (currState.isGoal(currState.vehicles.get('P'))) {
            goalState = currState;
            return;
        }

        processState(currState);
    }
}

@Override
protected void processState(State currState) {
    if (visited.contains(currState)) {
        return;
    }
    visited.add(currState);

    List<State> succ = currState.generateSucc();
    for (State s : succ) {
        if (!visited.contains(s)) {
            queue.add(s);
        }
    }
}

public State getGoalState() {
    if (goalState == null)
        search();
    return goalState;
}

@Override
public int getVisitedNodesCount() {
    return visitedNodesCount;
}
}

```

Langkah-langkah algoritma :

1. UCS menginisialisasi *PriorityQueue* dengan memasukkan *initState* (Kondisi Awal). *PriorityQueue* akan mengurutkan *State* berdasarkan nilai *cost* terkecil
2. Memilih *State* terkecil dari *PriorityQueue*
3. Mengecek apakah *State* tersebut sudah mencapai goal, jika sudah berhenti.

4. Membuat kemungkinan *State* yang mungkin pada fungsi *processState*
5. Mengecek apakah *State* tersebut sudah dikunjungi, jika belum masukkan *State* tersebut ke *PriorityQueue*.
6. Ulangi langkah 2-5

### III.IV. Algoritma GBFS dalam Program

Kode dalam program adalah sebagai berikut:

```
package main;

import java.util.*;

public class GBFS extends BaseSearch {
    public PriorityQueue<State> queue;
    public Set<State> visited;
    public State goalState;

    public GBFS(State initState) {
        super(initState);
        this.queue = new PriorityQueue<>(Comparator.comparingInt(s ->
s.heuristicCost));
        this.visited = new HashSet<>();
        this.goalState = null;
    }

    @Override
    public void search() {
        queue.add(initState);
        while (!queue.isEmpty()) {
            visitedNodesCount++;
            State current = queue.poll();
            if (current.isGoal(current.vehicles.get('P'))) {
                goalState = current;
                return;
            }
            processState(current);
        }
    }

    @Override
    public void processState(State currState) {
        if (visited.contains(currState)) {
            return;
        }
        visited.add(currState);

        // Generate successors
        List<State> succ = currState.generateSucc();
        for (State s : succ) {
            if (!visited.contains(s)) {
                queue.add(s);
            }
        }
    }
}
```

```

    }
}

public State getGoalState() {
    if (goalState == null)
        search();
    return goalState;
}

@Override
public int getVisitedNodesCount() {
    return visitedNodesCount;
}
}

```

Langkah-langkah algoritma :

1. GBFS menginisialisasi *PriorityQueue* dengan memasukkan *initState* (Kondisi Awal). *PriorityQueue* akan mengurutkan *State* berdasarkan nilai *heuristicCost* terkecil
2. Memilih *State* terkecil dari *PriorityQueue*
3. Mengecek apakah *State* tersebut sudah mencapai goal, jika sudah berhenti.
4. Membuat kemungkinan *State* yang mungkin pada fungsi *processState*
5. Mengecek apakah *State* tersebut sudah dikunjungi, jika belum masukkan *State* tersebut ke *PriorityQueue*.
6. Ulangi langkah 2-5

### III.V. Algoritma A\* dalam Program

Kode dalam program adalah sebagai berikut:

```

public class AStar extends BaseSearch {
    private PriorityQueue<State> openSet;
    private Set<State> closedSet;
    private State goalState;

    public AStar(State initState) {
        super(initState);
        this.openSet = new PriorityQueue<>(Comparator.comparingInt(s ->
s.cost + heuristic(s)));
        this.closedSet = new HashSet<>();
    }

    private int heuristic(State state) {
        return state.getHeuristicCost(state.methode);
    }

    @Override

```



```

public void search() {
    openSet.add(initState);

    while(!openSet.isEmpty()) {
        State current = openSet.poll();
        visitedNodesCount++;

        if(current.isGoal(current.vehicles.get('P'))) {
            goalState = current;
            return;
        }

        processState(current);
    }
}

@Override
protected void processState(State current) {
    if(closedSet.contains(current)) return;

    closedSet.add(current);
    for(State successor : current.generateSucc()) {
        if(!closedSet.contains(successor)) {
            successor.cost = current.cost + 1 + heuristic(successor);
            openSet.add(successor);
        }
    }
}

public State getGoalState() {
    if(goalState == null) search();
    return goalState;
}

@Override
public int getVisitedNodesCount() {
    return visitedNodesCount;
}
}

```

Langkah-langkah algoritma :

1. A\* menginisialisasi *PriorityQueue* dengan memasukkan *initState* (Kondisi Awal). *PriorityQueue* akan mengurutkan *State* berdasarkan nilai *cost* + heuristic(s) terkecil (dengan s adalah method heuristics nya)
2. Memilih *State* terkecil dari *PriorityQueue*
3. Mengecek apakah *State* tersebut sudah mencapai goal, jika sudah berhenti.
4. Membuat kemungkinan *State* yang mungkin pada fungsi *processState*
5. Mengecek apakah *State* tersebut sudah dikunjungi, jika belum masukkan *State* tersebut ke *PriorityQueue*.
6. Ulangi langkah 2-5

### III.VI. Algoritma IDA\* dalam Program

Kode dalam program adalah sebagai berikut:

```
package main;
import java.util.Stack;

public class IDAStar extends BaseSearch {
    private int threshold;
    private State goalState;
    private String heuristicMethod;

    public IDAStar(State initState) {
        super(initState);
        this.threshold = initState.getHeuristicCost(initState.methode);
        this.heuristicMethod = initState.methode;
    }

    @Override
    public void search(){
        while (true){
            int result = DLS(initState, 0);
            if (result == -1) {
                goalState = null;
                return;
            } else if (result == Integer.MIN_VALUE) {
                return;
            }
            threshold = result;
        }
    }

    private int DLS(State state, int g){
        visitedNodesCount++;
        int f = g + state.getHeuristicCost(heuristicMethod);

        if (f > threshold) return f;
        if (state.isGoal(state.vehicles.get('P'))){
            goalState = state;
            return Integer.MIN_VALUE;
        }

        int min = Integer.MAX_VALUE;
        for (State succ : state.generateSucc()){
            int res = DLS(succ, g+1);
            if (res == Integer.MIN_VALUE) return Integer.MIN_VALUE;
            if (res < min) min = res;
        }
        return min;
    }

    @Override
```

```

protected void processState(State current) {
}
public State getGoalState() {
    return goalState;
}

@Override
public int getVisitedNodesCount() {
    return visitedNodesCount;
}
}

```

Langkah-langkah Algoritma :

1. IDA\* menginisialisasi nilai threshold menggunakan heuristic dari *initState*, yaitu *initState.getHeuristicCost(initState.methode)*. Threshold ini berfungsi sebagai batas atas dari nilai  $f = g + h$  yang boleh dijelajahi pada iterasi saat ini.
2. Melakukan pencarian mendalam terbatas (DLS - Depth-Limited Search) dimulai dari *initState* dengan level  $g = 0$ .
3. Pada setiap node State yang dikunjungi:
  - a. Hitung  $f = g + h$  di mana  $g$  adalah depth (langkah dari awal), dan  $h$  adalah nilai heuristic dari State tersebut.
  - b. Jika  $f > \text{threshold}$ , kembalikan nilai  $f$  ke pemanggil untuk dijadikan threshold baru pada iterasi selanjutnya.
  - c. Jika State adalah goal (*state.isGoal(state.vehicles.get('P'))*), simpan state tersebut sebagai *goalState* dan kembalikan sinyal bahwa goal telah ditemukan.
4. Generate semua successor dari State saat ini (*state.generateSucc()*) dan rekursifkan fungsi DLS untuk masing-masing successor dengan  $g+1$ .
5. Jika semua successor telah dieksplorasi namun tidak ada yang mencapai goal, ambil nilai minimum dari semua  $f$  yang melebihi threshold dan jadikan itu threshold baru untuk iterasi berikutnya.
6. Jika goal ditemukan (return Integer.MIN\_VALUE), hentikan pencarian
7. Ulangi proses dari langkah 2 menggunakan threshold baru, sampai goal ditemukan atau tidak ada State lagi yang dapat dieksplorasi (tidak ditemukan solusi).

### III.VII. Algoritma GA dalam program

Kode dalam program adalah sebagai berikut:

```

package main;

import java.util.*;

```

```

public class GA {
    private static final int POPULATION_SIZE = 30;
    private static final double MUTATION_RATE = 0.1;
    private static final int MAX_GENERATIONS = 20;
    private static final int STALL_GENERATIONS = 2;

    private State initState;

    public GA(State initState) {
        this.initState = initState;
    }

    public HeuristicSolution evolveHeuristic() {
        List<HeuristicIndividual> population = initializePopulation();
        evaluatePopulationFitness(population);

        int lastImprovement = 0;
        int bestNodesVisited = Integer.MAX_VALUE;
        HeuristicSolution bestSolution = null;

        for(int gen=0; gen < MAX_GENERATIONS; gen++) {

            HeuristicSolution currentBest = getBestSolution(population);
            if (currentBest != null && currentBest.nodesVisited <
bestNodesVisited) {
                bestNodesVisited = currentBest.nodesVisited;
                lastImprovement = gen;
                bestSolution = currentBest;
            }

            if (gen - lastImprovement >= STALL_GENERATIONS) {
                break;
            }

            population = createNewGeneration(population);
            evaluatePopulationFitness(population);
        }

        return bestSolution != null ? bestSolution :
getBestSolution(population);
    }

    private List<HeuristicIndividual> initializePopulation() {
        List<HeuristicIndividual> population = new ArrayList<>();
        for(int i=0; i<POPULATION_SIZE; i++) {
            population.add(new HeuristicIndividual());
        }
        return population;
    }

    private void evaluatePopulationFitness(List<HeuristicIndividual>
population) {

```

```

        population.forEach(ind -> {
            AStar astar = new AStar(initState.copy());
            astar.search();
            ind.setFitness(1.0 / (astar.getVisitedNodesCount() + 1));
            ind.setSolution(new HeuristicSolution(
                astar.getVisitedNodesCount(),
                astar.getGoalState()
            ));
        });
    }

    private List<HeuristicIndividual>
createNewGeneration(List<HeuristicIndividual> population) {
        population.sort(Comparator.comparingDouble(ind ->
ind.getFitness()));
        Collections.reverse(population);

        List<HeuristicIndividual> newPopulation = new ArrayList<>();
        newPopulation.add(population.get(0).copy());
        newPopulation.add(population.get(1).copy());

        while(newPopulation.size() < POPULATION_SIZE) {
            HeuristicIndividual parent1 = selectParent(population);
            HeuristicIndividual parent2 = selectParent(population);
            HeuristicIndividual child = crossover(parent1, parent2);
            mutate(child);
            newPopulation.add(child);
        }

        return newPopulation;
    }

    private HeuristicIndividual selectParent(List<HeuristicIndividual>
population) {
        return population.stream()

.sorted(Comparator.comparingDouble(HeuristicIndividual::getFitness).reversed
())

        .limit(3)
        .findFirst()
        .get();
    }

    private HeuristicIndividual crossover(HeuristicIndividual p1,
HeuristicIndividual p2) {
        HeuristicIndividual child = new HeuristicIndividual();
        p1.getHeuristics().forEach(h ->
            child.setWeight(h, Math.random() < 0.5 ? p1.getWeight(h) :
p2.getWeight(h))
        );
        return child;
    }

    private void mutate(HeuristicIndividual ind) {

```

```

        if(Math.random() < MUTATION_RATE) {
            String randomHeuristic = ind.getRandomHeuristic();
            ind.setWeight(randomHeuristic, Math.random());
        }
    }

    private HeuristicSolution getBestSolution(List<HeuristicIndividual>
population) {
        return population.stream()

.max(Comparator.comparingDouble(HeuristicIndividual::getFitness))
        .get()
        .getSolution();
    }

    public static class HeuristicIndividual {
        private Map<String, Double> weights;
        private double fitness;
        private HeuristicSolution solution;

        public HeuristicIndividual() {
            weights = new HashMap<>();
            weights.put("MANHATTAN", Math.random());
            weights.put("BLOCKED", Math.random());
            weights.put("COMBINEDMB", Math.random());
            weights.put("CHEBYSHEV", Math.random());
        }

        public HeuristicIndividual copy() {
            HeuristicIndividual copy = new HeuristicIndividual();
            copy.weights = new HashMap<>(this.weights);
            copy.fitness = this.fitness;
            copy.solution = this.solution;
            return copy;
        }

        public double getWeight(String heuristic) { return
weights.get(heuristic); }
        public void setWeight(String h, double w) { weights.put(h, w); }
        public List<String> getHeuristics() { return new
ArrayList<>(weights.keySet()); }
        public String getRandomHeuristic() {
            return getHeuristics().get((int)(Math.random()*weights.size()));
        }
        public double getFitness() { return fitness; }
        public void setFitness(double fitness) { this.fitness = fitness; }
        public HeuristicSolution getSolution() { return solution; }
        public void setSolution(HeuristicSolution solution) { this.solution
= solution; }
    }

    public static class HeuristicSolution {
        public final int nodesVisited;
    }

```

```

        public final State solutionState;

        public HeuristicSolution(int nodes, State state) {
            nodesVisited = nodes;
            solutionState = state;
        }
    }
}

```

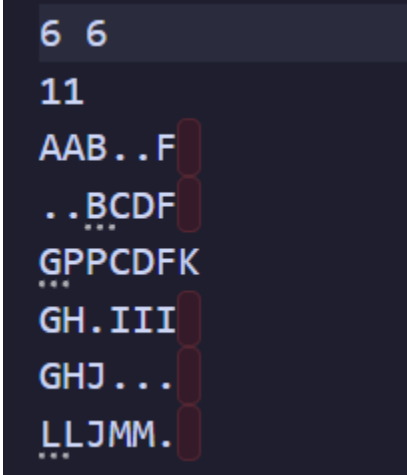
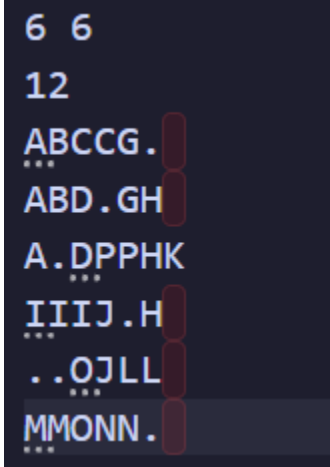
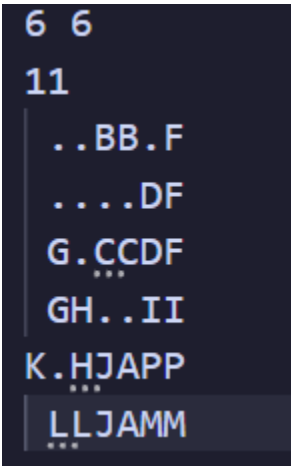
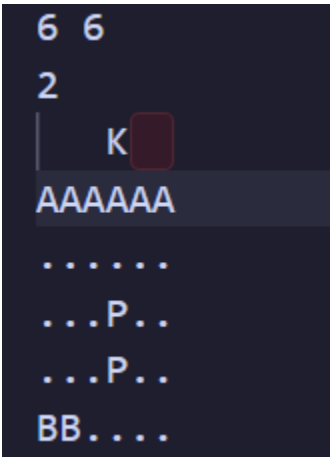
Langkah-langkah algoritma:

1. GA menginisialisasi populasi awal sebanyak POPULATION\_SIZE (30) individu melalui fungsi initializePopulation(). Setiap individu memiliki bobot acak untuk 4 heuristik berbeda: MANHATTAN, BLOCKED, COMBINEDMB, dan CHEBYSHEV.
2. Evaluasi fitness setiap individu dalam populasi menggunakan algoritma A\* melalui fungsi evaluatePopulationFitness():
  - a. Untuk setiap individu, jalankan algoritma A\* dengan state awal
  - b. Hitung nilai fitness sebagai  $1/(\text{jumlah node yang dikunjungi} + 1)$
  - c. Semakin sedikit node yang dikunjungi, semakin tinggi nilai fitness
3. Jalankan proses evolusi selama maksimal MAX\_GENERATIONS (20) generasi:
  - a. Catat solusi terbaik saat ini (individu dengan fitness tertinggi)
  - b. Jika tidak ada perbaikan setelah STALL\_GENERATIONS (2), hentikan proses evolusi
4. Untuk setiap generasi, buat populasi baru melalui fungsi createNewGeneration():
  - a. Pertahankan 2 individu terbaik (elitism)
  - b. Untuk sisa populasi:
    - i. Pilih 2 parent menggunakan fungsi selectParent() (tournament selection)
    - ii. Lakukan crossover antara 2 parent untuk menghasilkan child
    - iii. Mutasi child dengan probabilitas MUTATION\_RATE (0.1)
    - iv. Tambahkan child ke populasi baru
5. Evaluasi fitness populasi baru
6. Ulangi langkah 3-5 hingga kondisi berhenti terpenuhi:
  - a. Mencapai MAX\_GENERATIONS
  - b. Tidak ada perbaikan setelah STALL\_GENERATIONS
7. Kembalikan solusi terbaik yang ditemukan selama proses evolusi

## BAB IV ANALISIS DAN PERCOBAAN

### IV.I. Inisialisasi Percobaan

Pada percobaan ini kami menggunakan 4 jenis *test case* untuk setiap algoritma. *Test case* tersebut berada pada tabel berikut :

Tabel Test Case	
 <pre>6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.</pre>	 <pre>6 6 12 ABCCG. ... ABD.GH A.DPPHK .IIIJ.H ..OJLL MMONN.</pre>
<p>Test Case 1 (mudah) Atau dapat akses di link: <a href="https://github.com/ryonlunar/Tucil3_13523042_13523052/blob/main/test/in/t1.txt">https://github.com/ryonlunar/Tucil3_13523042_13523052/blob/main/test/in/t1.txt</a></p>	<p>Test Case 2 (sulit) Atau dapat akses di link: <a href="https://github.com/ryonlunar/Tucil3_13523042_13523052/blob/main/test/in/t2.txt">https://github.com/ryonlunar/Tucil3_13523042_13523052/blob/main/test/in/t2.txt</a></p>
 <pre>6 6 11 ..BB.F ....DF G.CCDF GH..II K.HJAPP LLJAMM</pre>	 <pre>6 6 2 K AAAAAA ..... ...P.. ...P.. BB....</pre>
<p>Test Case 3 (sulit) Atau dapat akses di link: <a href="https://github.com/ryonlunar/Tucil3_13523042_13523052/blob/main/test/in/t3.txt">https://github.com/ryonlunar/Tucil3_13523042_13523052/blob/main/test/in/t3.txt</a></p>	<p>Test Case 4 (no solution) Atau dapat akses di link: <a href="https://github.com/ryonlunar/Tucil3_13523042_13523052/blob/main/test/in/t4.txt">https://github.com/ryonlunar/Tucil3_13523042_13523052/blob/main/test/in/t4.txt</a></p>



Jika ingin membuat file secara manual tolong masukkan spasi (yang bertanda merah) pada file .txt Anda. Jika tidak akan memicu error ketika parsing.

#### IV.II. Percobaan dengan UCS

Fungsi evaluasi pada algoritma UCS secara umum telah dijelaskan pada bagian Landasan Teori. Untuk kasus ini, fungsi evaluasi tersebut menggunakan *cost* sebagai nilai pada fungsi tersebut. *cost* sendiri merupakan langkah yang telah ditempuh dari *State* awal hingga *State* saat ini.

$$f(n) = cost(n)$$

GIF output dari masing-masing percobaan dapat dilihat di link berikut :

[https://github.com/ryonlunar/Tucil3\\_13523042\\_13523052/tree/main/test/out/UCS](https://github.com/ryonlunar/Tucil3_13523042_13523052/tree/main/test/out/UCS)

Percobaan	Input	Output
1 (mudah)	Test Case 1	Runtime: 1.3190 detik Visited Nodes: 15399 Path Length: 6
2 (sulit)	Test Case 2	Runtime: 0.4720 detik Visited Nodes: 11626 Path Length: 51
3 (sulit)	Test Case 3	Runtime: 0.7390 detik Visited Nodes: 15399 Path Length: 6
4 (no solution)	Test Case 4	Runtime: 0.0300 detik Visited Nodes: 34 Path Length: 0 No solution found

#### IV.III. Percobaan dengan GBFS

Fungsi evaluasi pada algoritma GBFS secara umum telah dijelaskan pada bagian Landasan Teori. Untuk kasus ini, fungsi evaluasi tersebut menggunakan heuristik  $h(n)$  sebagai nilai pada fungsi tersebut. Heuristik ini merupakan estimasi atau perkiraan biaya dari *State* saat ini menuju *State* tujuan. Nilai ini digunakan untuk mengarahkan pencarian menuju tujuan dengan cara memilih *State* yang tampak paling dekat berdasarkan estimasi tersebut.

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

Dalam hal ini, kami memiliki beberapa fungsi heuristik yang telah dijelaskan pada bagian Fungsi Heuristik.

GIF output dari masing-masing percobaan dapat dilihat di link berikut :

[https://github.com/ryonlunar/Tucil3\\_13523042\\_13523052/tree/main/test/out/GBFS](https://github.com/ryonlunar/Tucil3_13523042_13523052/tree/main/test/out/GBFS)

Test Case	Heuristik	Output
1	Manhattan	Runtime: 0.0660 detik Visited Nodes: 193 Path Length: 69
1	Blocked	Runtime: 0.0020 detik Visited Nodes: 26 Path Length: 7
1	CombineMB	Runtime: 0.0030 detik Visited Nodes: 28 Path Length: 9
1	Chebyshev	Runtime: 0.1030 detik Visited Nodes: 1337 Path Length: 653
1	Evolve	Runtime: 0.0050 detik Visited Nodes: 119 Path Length: 33
2	Manhattan	Runtime: 0.0990 detik Visited Nodes: 5284 Path Length: 202
2	Blocked	Runtime: 0.0650 detik Visited Nodes: 4943 Path Length: 162
2	CombineMB	Runtime: 0.0860 detik Visited Nodes: 4995 Path Length: 191
2	Chebyshev	Runtime: 0.0910 detik Visited Nodes: 2188 Path Length: 613
2	Evolve	Runtime: 0.0880 detik Visited Nodes: 5164 Path Length: 254

3	Manhattan	Runtime: 0.4540 detik Visited Nodes: 5330 Path Length: 1619
3	Blocked	Runtime: 0.1370 detik Visited Nodes: 6030 Path Length: 20
3	CombineMB	Runtime: 0.1370 detik Visited Nodes: 6025 Path Length: 20
3	Chebyshev	Runtime: 1.3320 detik Visited Nodes: 8791 Path Length: 4373
3	Evolve	Runtime: 0.1830 detik Visited Nodes: 1531 Path Length: 80
4	Manhattan	Runtime: 0.0100 detik Visited Nodes: 34 Path Length: 0 No solution found.
4	Blocked	Runtime: 0.0070 detik Visited Nodes: 34 Path Length: 0 No solution found.
4	CombineMB	Runtime: 0.0060 detik Visited Nodes: 34 Path Length: 0 No solution found.
4	Chebyshev	Runtime: 0.0040 detik Visited Nodes: 34 Path Length: 0 No solution found
4	Evolve	Runtime: 0.0040 detik Visited Nodes: 34 Path Length: 0 No solution found.

#### IV.IV. Percobaan dengan A\*

Fungsi evaluasi pada algoritma A\* secara umum telah dijelaskan pada bagian Landasan Teori. Untuk kasus ini, fungsi evaluasi tersebut menggunakan kombinasi antara cost dan heuristik, yaitu  $g(n)$  dan  $h(n)$ , sebagai nilai pada fungsi tersebut. Nilai  $g(n)$  merupakan biaya atau langkah yang telah ditempuh dari *State* awal hingga *State* saat ini, sedangkan  $h(n)$  adalah estimasi biaya dari *State* saat ini menuju *State* tujuan. Kombinasi ini membuat A\* bersifat optimal dan efisien dalam mencari jalur terpendek.

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

Dalam hal ini, kami memiliki beberapa fungsi heuristik yang telah dijelaskan pada bagian Fungsi Heuristik.

GIF output dari masing-masing percobaan dapat dilihat di link berikut :

[https://github.com/ryonlunar/Tucil3\\_13523042\\_13523052/tree/main/test/out/Astar](https://github.com/ryonlunar/Tucil3_13523042_13523052/tree/main/test/out/Astar)

Test Case	Heuristik	Output
1	Manhattan	Runtime: 0.0680 detik Visited Nodes: 181 Path Length: 6
1	Blocked	Runtime: 0.0200 detik Visited Nodes: 34 Path Length: 5
1	CombineMB	Runtime: 0.0220 detik Visited Nodes: 96 Path Length: 5
1	Chebyshev	Runtime: 0.1570 detik Visited Nodes: 1337 Path Length: 653
1	Evolve	Runtime: 0.0260 detik Visited Nodes: 82 Path Length: 6
2	Manhattan	Runtime: 0.3230 detik Visited Nodes: 9996 Path Length: 52
2	Blocked	Runtime: 0.1840 detik Visited Nodes: 8816 Path Length: 52
2	CombineMB	Runtime: 0.2310 detik Visited Nodes: 9801

		Path Length: 51
2	Chebyshev	Runtime: 0.0980 detik Visited Nodes: 2188 Path Length: 613
2	Evolve	Runtime: 0.2270 detik Visited Nodes: 9421 Path Length: 51
3	Manhattan	Runtime: 0.3620 detik Visited Nodes: 8096 Path Length: 6
3	Blocked	Runtime: 0.0500 detik Visited Nodes: 631 Path Length: 6
3	CombineMB	Runtime: 0.0860 detik Visited Nodes: 2551 Path Length: 6
3	Chebyshev	Runtime: 1.3550 detik Visited Nodes: 8791 Path Length: 4373
3	Evolve	Runtime: 0.1960 detik Visited Nodes: 5200 Path Length: 6
4	Manhattan	Runtime: 0.0000 detik Visited Nodes: 34 Path Length: 0 No solution found.
4	Blocked	Runtime: 0.0010 detik Visited Nodes: 34 Path Length: 0 No solution found.
4	CombineMB	Runtime: 0.0000 detik Visited Nodes: 34 Path Length: 0 No solution found.
4	Chebyshev	Runtime: 0.0010 detik Visited Nodes: 34 Path Length: 0

		No solution found.
4	Evolve	Runtime: 0.0010 detik Visited Nodes: 34 Path Length: 0 No solution found.

#### IV.V. Percobaan dengan IDA\*

Fungsi evaluasi pada algoritma IDA\* secara umum telah dijelaskan pada bagian Landasan Teori. IDA\* (Iterative Deepening A\*) menggunakan prinsip yang sama dengan A\*, yaitu memanfaatkan kombinasi antara cost dan heuristik dalam fungsi evaluasinya. Namun, berbeda dengan A\*, IDA\* melakukan pencarian secara iteratif dengan batas ambang (threshold) yang semakin meningkat berdasarkan nilai  $f(n)$ . Nilai evaluasi dihitung dengan menambahkan biaya yang telah ditempuh  $g(n)$  dan estimasi ke tujuan  $h(n)$ .

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

Dalam hal ini, kami memiliki beberapa fungsi heuristik yang telah dijelaskan pada bagian Fungsi Heuristik.

GIF output dari masing-masing percobaan dapat dilihat di link berikut :

[https://github.com/ryonlunar/Tucil3\\_13523042\\_13523052/tree/main/test/out/IDA](https://github.com/ryonlunar/Tucil3_13523042_13523052/tree/main/test/out/IDA)

Test Case	Heuristik	Output
1	Manhattan	Runtime: 0.0350 detik Visited Nodes: 2361 Path Length: 6
1	Blocked	Runtime: 0.0090 detik Visited Nodes: 1362 Path Length: 5
1	CombineMB	Runtime: 0.0020 detik Visited Nodes: 151 Path Length: 6
1	Chebyshev	Runtime: 0.0760 detik Visited Nodes: 31947 Path Length: 5
1	Evolve	Runtime: 0.0250 detik Visited Nodes: 1680 Path Length: 6
2	Manhattan	Terlalu Lambat

2	Blocked	Terlalu Lambat
2	CombineMB	Terlalu Lambat
2	Chebyshev	Terlalu Lambat
2	Evolve	Terlalu Lambat
3	Manhattan	Terlalu Lambat
3	Blocked	Terlalu Lambat
3	CombineMB	Terlalu Lambat
3	Chebyshev	Terlalu Lambat
3	Evolve	Terlalu Lambat
4	Manhattan	Terlalu Lambat
4	Blocked	Terlalu Lambat
4	CombineMB	Terlalu Lambat
4	Chebyshev	Terlalu Lambat
4	Evolve	Terlalu Lambat

#### IV.VI. Percobaan dengan GA

Fungsi evaluasi pada algoritma GA secara umum telah dijelaskan pada bagian Landasan Teori. Untuk kasus ini, fungsi evaluasi tersebut menggunakan fungsi fitness sebagai nilai pada fungsi tersebut. Fitness merepresentasikan seberapa baik suatu solusi (individu/kromosom) dalam menyelesaikan permasalahan. Nilai fitness digunakan untuk menentukan peluang suatu individu untuk dipilih dalam proses reproduksi seperti seleksi, crossover, dan mutasi. Semakin tinggi nilai fitness, semakin besar kemungkinan individu tersebut akan dipertahankan dan berkembang dalam generasi berikutnya.

$$f(x) = fitness(x)$$

Dalam hal ini, fungsi fitness yang digunakan telah dijelaskan pada bagian Genetic Algorithm.

GIF output dari masing-masing percobaan dapat dilihat di link berikut :

[https://github.com/ryonlunar/Tucil3\\_13523042\\_13523052/tree/main/test/out/GA](https://github.com/ryonlunar/Tucil3_13523042_13523052/tree/main/test/out/GA)

Test Case	Heuristik	Output
-----------	-----------	--------

1	Manhattan	Runtime: 1.0230 detik Visited Nodes: 181 Path Length: 7
1	Blocked	Runtime: 0.1300 detik Visited Nodes: 34 Path Length: 6
1	CombineMB	Runtime: 0.3980 detik Visited Nodes: 96 Path Length: 6
1	Chebyshev	Runtime: 10.5980 detik Visited Nodes: 1337 Path Length: 654
1	Evolve	Runtime: 0.3280 detik Visited Nodes: 82 Path Length: 7
2	Manhattan	Runtime: 31.0720 detik Visited Nodes: 9996 Path Length: 53
2	Blocked	Runtime: 27.3120 detik Visited Nodes: 8816 Path Length: 53
2	CombineMB	Runtime: 31.5960 detik Visited Nodes: 9801 Path Length: 52
2	Chebyshev	Runtime: 31.5960 detik Visited Nodes: 9801 Path Length: 52
2	Evolve	Runtime: 49.3890 detik Visited Nodes: 9421 Path Length: 52
3	Manhattan	Runtime: 29.0230 detik Visited Nodes: 8096 Path Length: 7
3	Blocked	Runtime: 1.8220 detik Visited Nodes: 631 Path Length: 7



3	CombineMB	Runtime: 7.5560 detik Visited Nodes: 2551 Path Length: 7
3	Chebyshev	Runtime: 130.0440 detik Visited Nodes: 8791 Path Length: 4374
3	Evolve	Runtime: 21.1360 detik Visited Nodes: 5200 Path Length: 7
4	Manhattan	Runtime: 0.1520 detik Visited Nodes: 33 Path Length: 0 No solution found.
4	Blocked	Runtime: 0.1440 detik Visited Nodes: 33 Path Length: 0 No solution found
4	CombineMB	Runtime: 0.1240 detik Visited Nodes: 33 Path Length: 0 No solution found.
4	Chebyshev	Runtime: 0.0910 detik Visited Nodes: 33 Path Length: 0 No solution found.
4	Evolve	Runtime: 0.0930 detik Visited Nodes: 33 Path Length: 0 No solution found.

#### IV.VII. Analisis Algoritma

Akibat percobaan yang terlalu banyak, untuk output langkah-langkah tersebut dapat diakses pada github melalui link berikut :

[https://github.com/ryonlunar/Tucil3\\_13523042\\_13523052/tree/main/test/out](https://github.com/ryonlunar/Tucil3_13523042_13523052/tree/main/test/out)

Dalam algoritma pencarian / *pathfinding*  $f(n)$  merupakan fungsi evaluasi yang menentukan seberapa baik sebuah node untuk dieksplorasi selanjutnya. dalam fungsi evaluasi ( $f(n)$ ) terdapat  $g(n)$  yang merupakan biaya dari kondisi awal (*initial state*) ke node  $n$  (ke *state* saat ini). Hal ini merupakan biaya atau *cost* yang telah ditempuh.

Dalam menerapkan algoritma A\*, kami menggunakan beberapa jenis pendekatan heuristik. Heuristik yang diimplementasikan pada A\* akan admissible jika tidak melebihi estimasi biaya minimum sebenarnya. Dalam hal ini heuristik yang kami gunakan terdapat Manhattan Distance, Blocked Vehicle, kombinasi Manhattan Distance dan Blocked Vehicle, Chebyshev, dan Weighted dari Chebyshev, Manhattan Distance, dan Blocked Vehicle. Untuk Manhattan Distance dan Blocked Vehicle pasti admissible. Manhattan Distance admissible karena mobil hanya bergerak secara horizontal dan vertikal sehingga tidak memperhitungkan rintangan secara berlebihan. Blocked Vehicle juga admissible karena untuk mencapai target tentu tidak boleh ada kendaraan yang berada di depan target.

Untuk Chebyshev tidak admissible. Hal ini karena Chebyshev menghitung gerakan diagonal pada grid sehingga bisa terjadi biaya berlebih dalam menghitung nilai heuristik. Untuk kombinasi Manhattan + Blocked Vehicle dan Weighing. Komponen-komponen yang admissible jika digabungkan belum tentu admissible. Penjumlahan dapat mengakibatkan perhitungan tersebut menjadi *overestimate* dari sesungguhnya.

Pada penyelesaian Rush Hour ini, metode UCS yang digunakan menggunakan *cost* yang sama untuk semua aksi. Karena setiap gerakan memiliki bobot yang sama, maka UCS akan memiliki langkah pembangkitan yang sama dengan BFS. UCS dan BFS akan mengunjungi node dalam urutan yang sama sehingga *path* yang dihasilkan juga sama panjang dan optimal.

Algoritma A\* yang diterapkan pada persoalan ini secara teoritis lebih efisien dibanding UCS, jika heuristik yang digunakan informatif dan admissible. UCS mencari solusi optimal tanpa panduan ke *goal*, hanya berdasarkan biaya tempuh sejauh ini. A\* akan memanfaatkan panduan tersebut dari heuristik yang digunakan sehingga dapat mengurangi jumlah node yang dieksplorasi dan membuat A\* menjadi lebih efisien.

Secara teoritis, GBFS yang diimplementasikan pada persoalan ini tidak menjamin solusi optimal. GBFS hanya menggunakan heuristik tanpa mempertimbangkan *cost* yang telah ditempuh sejauh ini. Oleh karena itu, memilih jalur yang optimum lokal belum tentu memberikan hasil yang mengarah ke optimum global. Akibatnya, GBFS dapat melewati jalur optimal dan berhenti pada solusi sub-optimal.

#### IV.VIII. Analisis Kompleksitas

##### 1. Uniform Cost Search (UCS)

- Kompleksitas Waktu: UCS memiliki kompleksitas waktu sebesar  $O(b^d)$ , di mana  $b$  adalah branching factor dan  $d$  adalah depth dari solusi optimal. Karena UCS

mengeksplorasi semua node berdasarkan cost terkecil tanpa menggunakan heuristik, ia bisa sangat lambat pada kasus dengan depth besar.

```
PriorityQueue<State> queue = new
PriorityQueue<>(Comparator.comparingInt(State::getCost));
...
while (!queue.isEmpty()) {
    State current = queue.poll();
    ...
    for (State neighbor : current.generateNextStates()) {
        ...
        queue.add(neighbor);
    }
}
```

- Kelebihan: UCS menjamin solusi optimal jika cost dari setiap langkah  $\geq 0$ .
- Kelemahan: Tidak efisien dalam ruang eksplorasi besar karena eksplorasi tidak diarahkan.

## 2. Greedy Best First Search (GBFS)

- Kompleksitas Waktu: Bergantung pada kualitas heuristik. Dalam kasus terbaik bisa sangat cepat, tetapi kompleksitas bisa menjadi  $O(b^m)$  di mana  $m$  adalah kedalaman maksimal karena GBFS tidak menjamin eksplorasi optimal.

```
PriorityQueue<State> queue = new
PriorityQueue<>(Comparator.comparingInt(State::getHeuristi
c));
...
while (!queue.isEmpty()) {
    State current = queue.poll();
    ...
    for (State neighbor : current.generateNextStates()) {
        ...
        queue.add(neighbor);
    }
}
```

- Kelebihan: Cepat jika heuristik cukup baik.
- Kelemahan: Tidak menjamin solusi optimal. Sangat bergantung pada heuristik.

### 3. A\* Search

- Kompleksitas Waktu: Dalam kasus terburuk bisa  $O(b^d)$  tetapi lebih efisien dibanding UCS jika heuristik admissible dan konsisten.

```
PriorityQueue<State> queue = new
PriorityQueue<>(Comparator.comparingInt(s -> s.getCost() +
s.getHeuristic()));
...
while (!queue.isEmpty()) {
    State current = queue.poll();
    ...
    for (State neighbor : current.generateNextStates()) {
        ...
        queue.add(neighbor);
    }
}
```

- Kelebihan: Menjamin solusi optimal bila heuristik admissible. Umumnya lebih cepat dari UCS.
- Kelemahan: Konsumsi memori tinggi karena menyimpan semua state dalam open list.

### 4. Iterative Deepening A\* (IDA\*)

- Kompleksitas Waktu: Biasanya sedikit lebih besar dari A\* karena melakukan pencarian berulang-ulang, namun tetap  $O(b^d)$ .

```
public void search(){
    while (true){
        int result = DLS(initState, 0);
        ...
    private int DLS(State state, int g){
        ...
        int f = g + state.getHeuristicCost(heuristicMethod);
        ...
    }
}
```

- Kelebihan: Konsumsi memori lebih kecil dari A\* karena menggunakan DFS dan tidak menyimpan semua node di memori.
- Kelemahan: Lebih lambat dari A\* karena eksplorasi berulang dengan batas cost yang meningkat.

## 5. Genetic Algorithm (GA)

- Kompleksitas Waktu: Tergantung pada jumlah populasi dan generasi. Kompleksitas bisa  $O(p * g * e)$  dengan  $p$  jumlah populasi,  $g$  jumlah generasi, dan  $e$  evaluasi tiap individu.

```
for (int gen = 0; gen < MAX_GENERATION; gen++) {  
    ...  
    evaluatePopulation(population);  
    List<Chromosome> selected = selection(population);  
    ...  
    mutate(selected);  
}
```

- Kelebihan: Cocok untuk problem besar dan ruang solusi yang tidak dapat dicari secara eksplisit.
- Kelemahan: Tidak menjamin solusi optimal. Bisa stagnan pada solusi lokal tergantung desain fitness dan operator genetic.

## LAMPIRAN

Link Github : [https://github.com/ryonlunar/Tucil3\\_13523042\\_13523052](https://github.com/ryonlunar/Tucil3_13523042_13523052)

Link Laporan :

No.	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 <b>.txt</b> dan menyimpan solusi berupa print board tahap per tahap dalam berkas <b>.txt</b>	✓	
5	<b>[Bonus]</b> Implementasi algoritma pathfinding alternatif (Disclaimer: Kadang IDA* masih terlalu lama ketika di run)	✓	
6	<b>[Bonus]</b> Implementasi 2 atau lebih heuristik alternatif	✓	
7	<b>[Bonus]</b> Program memiliki GUI	✓	
8	Program dan laporan dibuat (kelompok) sendiri	✓	