Laporan Tugas Kecil 3 IF2211 Strategi Algoritma

Sem II tahun 2024/2025

"Pemanfaatan Algoritma *PathFinding untuk Mencari Solusi Permainan Rush Hour*"

Disusun oleh :

Rafizan M S - 13523034
Salman Hanif - 13523056

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung

Bandung 2025

Daftar Isi

BAB I : Deskripsi Tugas

Gambar 1. Rush Hour Puzzle

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:

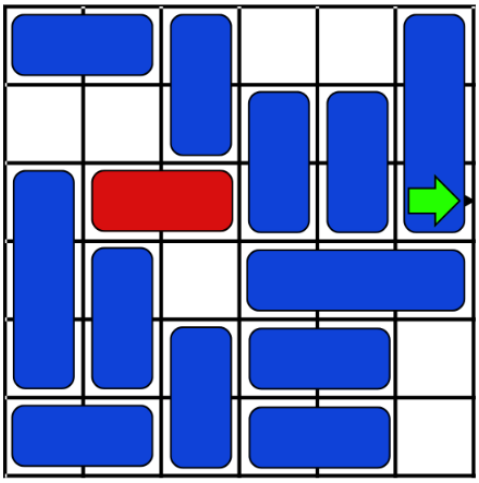
1. **Papan** – *Papan* merupakan tempat permainan dimainkan.
   Papan terdiri atas *cell*, yaitu sebuah *singular point* dari papan. Sebuah *piece* akan menempati *cell-cell* pada papan. Ketika permainan dimulai, semua *piece* telah diletakkan di dalam papan dengan konfigurasi tertentu berupa lokasi 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 dari Permainan game rush hour :



Gambar 2. Kondisi Awal Game Rush Hour

Pemain dapat menggeser-geser *piece* (termasuk *primary piece*) untuk membentuk jalan lurus antara *primary piece* dan *pintu keluar*. Puzzle berikut dinyatakan telah selesai apabila *primary piece* dapat digeser keluar papan melalui *pintu keluar*.

BAB II : Landasan Teori

Algoritma pathfinding adalah algoritma yang digunakan untuk menemukan jalur terbaik dari satu titik ke titik lainnya dalam suatu ruang pencarian (search space). Dalam konteks permainan, pathfinding memungkinkan agen atau objek dalam permainan mencari solusi untuk mencapai tujuan tertentu, meskipun terdapat hambatan atau rintangan di sekitarnya.

Pada permainan Rush Hour, tujuan utamanya adalah mengeluarkan mobil berwarna merah dari papan permainan melalui gerbang keluar di sisi kanan, dengan menggeser mobil-mobil lain yang menghalangi jalur. Karena banyaknya kemungkinan posisi dan gerakan mobil, pendekatan brute-force menjadi tidak efisien, dan algoritma pathfinding menjadi solusi untuk mencari jalur solusi secara lebih optimal dan terarah.

2.1 Algoritma UCS

Uniform Cost Search (UCS) adalah algoritma pencarian yang memilih jalur dengan biaya total terendah dari titik awal ke node tertentu. UCS adalah perluasan dari algoritma Breadth-First Search, namun memperhitungkan biaya kumulatif dari jalur yang telah ditempuh, bukan hanya kedalaman node.

UCS menggunakan struktur data antrian prioritas (*priority queue*), di mana setiap simpul (node) diberi prioritas berdasarkan biaya perjalanan total (g(n)). Node dengan nilai g(n) terendah akan diekspansi lebih dahulu.

- Termasuk dalam kategori algoritma *uninformed search*.
- Cocok untuk kasus dengan biaya langkah yang bervariasi.
- Tidak memperhitungkan jarak ke tujuan, hanya biaya sejauh ini.

Dalam *Rush Hour*, UCS dapat digunakan untuk menemukan solusi dengan jumlah langkah paling sedikit untuk memindahkan mobil merah ke pintu keluar. Setiap perpindahan satu langkah oleh satu mobil dianggap sebagai biaya tetap. UCS akan mencari urutan langkah dengan total biaya terendah hingga tujuan tercapai.

2.2 Algoritma Greedy Best First Search

Greedy Best First Search adalah algoritma pencarian yang memilih node yang terlihat paling dekat dengan tujuan berdasarkan fungsi heuristik h(n). Algoritma ini mengabaikan total biaya dari titik awal dan hanya memperhatikan estimasi jarak ke tujuan.

Node dengan nilai h(n) terkecil akan diprioritaskan untuk diekspansi. Heuristik h(n) bisa dirancang berdasarkan estimasi langkah minimal untuk mencapai tujuan dari node saat ini.

- Termasuk algoritma *informed search*.
- Fokus pada *heuristic* (fungsi h(n)).
- Cepat dalam menemukan solusi, namun tidak selalu optimal.

Dalam *Rush Hour*, heuristik h(n) dapat berupa jumlah mobil yang menghalangi jalan mobil merah ke pintu keluar. Greedy BFS bisa menemukan solusi dengan cepat, tetapi mungkin melewatkan solusi optimal karena tidak mengevaluasi keseluruhan biaya jalur.

2.3 Algoritma A*

A* adalah algoritma pencarian heuristik yang menggabungkan biaya perjalanan sejauh ini (g(n)) dan estimasi biaya ke tujuan (h(n)). Nilai totalnya dirumuskan sebagai f(n) = g(n) + h(n). A* merupakan algoritma yang sering digunakan dalam pencarian jalur optimal karena menggabungkan efisiensi dan ketepatan.

A* juga menggunakan *priority queue*, namun node diprioritaskan berdasarkan nilai f(n). Apabila fungsi heuristik h(n) yang digunakan *admissible* (tidak melebihi jarak sebenarnya) dan *consistent*, maka A* dijamin menemukan solusi optimal.

- *Informed search* yang optimal dan efisien (jika heuristik yang digunakan *admissible* dan *consistent*).
- Sering digunakan dalam pencarian jalur optimal di berbagai domain, termasuk game.

A* sangat cocok diterapkan pada permainan Rush Hour. Contoh heuristik yang digunakan dapat berupa:

- Jumlah mobil yang menghalangi jalur keluar.
- Estimasi jumlah gerakan minimum untuk memindahkan penghalang.

Dengan kombinasi g(n) dan h(n), A* mampu menemukan solusi dengan langkah paling efisien tanpa mengevaluasi seluruh kemungkinan.

2.4 Algoritma Beam Search

Beam Search adalah varian dari algoritma *Best First Search* yang membatasi jumlah simpul (node) yang dieksplorasi pada setiap tingkat pencarian. Berbeda dengan A* yang mengevaluasi semua kemungkinan berdasarkan nilai, Beam Search hanya mempertahankan k simpul terbaik pada setiap langkah, di mana k adalah *beam width* atau lebar balok pencarian yang ditentukan sebelumnya.

Beam Search menggunakan fungsi heuristik h(n) untuk memperkirakan jarak ke tujuan, namun tidak menyimpan semua simpul seperti A*. Hal ini membuat Beam Search lebih efisien dalam penggunaan memori, tetapi tidak menjamin solusi yang ditemukan adalah optimal, karena simpul yang menjanjikan bisa saja dibuang jika tidak masuk dalam k simpul terbaik pada suatu tingkat pencarian.

Beam Search cocok untuk kasus di mana memori terbatas atau pencarian solusi cepat lebih diutamakan daripada optimalitas absolut. Dalam konteks permainan Rush Hour, Beam Search dapat digunakan dengan heuristik yang sama seperti A*, misalnya:

- Jumlah kendaraan yang menghalangi jalan keluar kendaraan utama (P).
- Estimasi jumlah langkah minimal untuk memindahkan penghalang.

Secara umum, Beam Search menawarkan kompromi antara kecepatan pencarian dan kualitas solusi, dan cocok untuk eksperimen pembanding dengan BFS dan A* dalam menyelesaikan puzzle seperti Rush Hour.

2.5 Heuristik

Heuristik adalah fungsi penaksir yang digunakan dalam algoritma pencarian *informed* untuk memperkirakan seberapa dekat suatu simpul (node) terhadap tujuan. Dalam konteks permainan Rush Hour, heuristik yang baik dapat secara signifikan mempercepat pencarian solusi dengan mengarahkan algoritma pada jalur-jalur yang lebih menjanjikan.

2.5.1. Distance to Exit

Heuristik ini menghitung jumlah sel antara bagian belakang mobil merah dan pintu keluar di sisi kanan papan. Asumsinya adalah bahwa semakin dekat mobil merah dengan pintu keluar, semakin sedikit langkah yang diperlukan untuk mencapai tujuan.

2.5.2 Blocker Count

Heuristik ini menghitung jumlah kendaraan yang secara langsung menghalangi jalur mobil merah ke pintu keluar. Semakin banyak penghalang, maka semakin sulit mobil merah keluar.

2.5.3 Blocker Chaining

Heuristik ini lebih kompleks karena memperhitungkan *rantai penghalang*, yaitu kendaraan yang menghalangi penghalang utama mobil merah. Misalnya, jika sebuah mobil menghalangi mobil merah, tetapi mobil tersebut sendiri tidak bisa bergerak karena terhalang oleh mobil lain, maka semua mobil dalam rantai ini diperhitungkan.

BAB III : Analisis Algoritma Pathfinding

Dalam menyelesaikan permainan *Rush Hour*, digunakan berbagai algoritma pathfinding untuk mengeksplorasi jalur solusi. Analisis berikut membandingkan tiga algoritma utama, yaitu **Uniform Cost Search (UCS)**, **Greedy Best First Search**, dan **A*** berdasarkan definisi fungsi evaluasi, admissibility heuristik, efisiensi, dan optimalitas hasil solusi.

3.1 Definisi f(n) dan g(n)

Pengertian **g(n)** *biaya jalur* dari simpul awal (start node) hingga ke simpul saat ini (node n). Dalam konteks permainan *Rush Hour*, g(n) umumnya merepresentasikan jumlah total langkah yang telah dilakukan untuk mencapai konfigurasi papan tertentu. **f(n)** adalah *fungsi evaluasi* total dari sebuah node, yang digunakan untuk menentukan urutan ekspansi simpul oleh algoritma pencarian. Nilai f(n) tergantung pada algoritma yang digunakan:

- **UCS** : f(n) = g(n)
- **GBFS** : f(n) = h(n)
- **Beam** : f(n) = h(n)
- **A*** : f(n) = g(n) + h(n), dengan h(n) adalah heuristik estimasi biaya dari node n ke tujuan.

3.2 Heuristik pada A*

Heuristik **h(n)** dikatakan *admissible* jika tidak pernah melebihi biaya sebenarnya dari node n ke tujuan (dalam konteks minimisasi). Artinya, $h(n) \leq h^*(n)$, di mana $h^*(n)$ adalah biaya optimal sebenarnya dari n ke goal.

Contoh heuristik yang umum digunakan dalam permainan Rush Hour adalah:

- Jumlah mobil yang menghalangi jalur keluar.
- Jarak mobil merah ke pintu keluar.

Heuristik ini adalah admissible, karena:

- Ia tidak melebih-lebihkan jumlah langkah yang dibutuhkan.
- Ia mengabaikan kemungkinan langkah tambahan untuk memindahkan penghalang, sehingga nilainya cenderung *underestimate*.

Heuristik yang digunakan pada A* untuk permainan Rush Hour adalah admissible, sehingga A* tetap menjamin solusi optimal.

3.3 Apakah UCS sama dengan BFS pada Rush Hour

Secara umum:

- BFS adalah UCS dengan asumsi semua langkah memiliki biaya yang sama.

- Dalam Rush Hour, jika kita mendefinisikan bahwa setiap aksi (misalnya satu pergeseran mobil) memiliki biaya tetap (misalnya 1), maka UCS dan BFS akan mengeksplorasi node dengan urutan yang sama.

Namun secara teknis:

- UCS menggunakan *priority queue* berdasarkan nilai g(n).
- BFS menggunakan *queue biasa* (first-in-first-out) berdasarkan level.

Jika semua aksi memiliki biaya yang sama, maka UCS dan BFS akan membangkitkan node dalam urutan yang sama, dan menghasilkan jalur solusi yang sama panjang. Namun, implementasi mereka tetap berbeda dari sisi struktur data dan cara seleksi node.

3.4 Apakah UCS sama dengan BFS pada Rush Hour

Secara teoritis:

- UCS melakukan pencarian berdasarkan **biaya sejauh ini (g(n))** saja tanpa mempertimbangkan arah ke tujuan.
- A* menggunakan kombinasi **g(n) + h(n)**, sehingga lebih *terarah* menuju goal, dengan menghindari perluasan node yang tidak menjanjikan.

Secara teoritis, **A*** lebih efisien dibandingkan UCS dalam menyelesaikan *Rush Hour*, karena dapat mengurangi jumlah node yang diekspansi tanpa mengorbankan optimalitas, asalkan heuristik yang digunakan admissible.

3.5 Apakah UCS sama dengan BFS pada Rush Hour

Greedy Best First Search hanya menggunakan nilai heuristik h(n), tanpa memperhitungkan biaya langkah yang telah ditempuh (g(n)). Akibatnya:

- Ia mungkin memilih jalur yang terlihat dekat ke tujuan, tetapi ternyata memerlukan lebih banyak langkah.
- Tidak ada jaminan bahwa solusi yang ditemukan merupakan solusi dengan langkah paling sedikit.

Greedy Best First Search tidak menjamin solusi optimal pada permainan *Rush Hour*. Ia hanya mencari solusi tercepat (dalam waktu komputasi), bukan yang terbaik dalam hal jumlah langkah.

BAB IV : Source Code dan Pengujian

4.1 Source Code

Main.java

```java
import java.util.Scanner;

public class Main {
    private static final boolean debug = true;
    static Board initBoard;
    static Algorithm algorithm;
    static Heuristic heuristic;

    public static void configMenu(Scanner scanner) {…

    public static void algorithmMenu(Scanner scanner) {…

    public static void heuristicMenu(Scanner scanner) {…

    Run main | Debug main | Run | Debug
    public static void main(String[] args) {…
}
```

Solver.java

```java
public class Solver {
    // Algorithm;
    private static Algorithm algorithm;
    private static Heuristic heuristic;
    Board board; // Inisial state
    // prioqueue
    private final PriorityQueue<State> queue;
    // visited state
    private final Set<String> visited;

    public Solver(Board board, Algorithm algorithm, Heuristic heuristic) {…

    public static Algorithm getAlgorithm() {…

    public static Heuristic getHeuristic() {…

    // Mengeksplorasi semua kemungkinan gerakan dari state saat ini
    // dan menambahkannya ke antrian jika belum dikunjungi
    public void explore(State state) {…

    // Menggunakan algoritma UCS / GBFS / A* untuk menyelesaikan puzzle
    // Heuristic digunakan untuk menentukan urutan eksplorasi
    public void solve() {…

    private void printSolutionPath(State state) {…
}
```

State.java

```
4   public class State implements Comparable<State> {
5       private State parent;
6       Movement movement;
7       private final Board board;
8       int costSoFar;
9       int estimatedCostToGoal;
10
11      public static class Movement {···
32
33      public State(Board board, boolean isInitial, int costSoFar) {···
56
57      public Movement getMovement() {···
60
61      public State getParent() {···
64
65      public void setParent(State parent) {···
68
69      public Board getBoard() {···
72
73      public void setMovement(Movement movement) {···
76
77      public boolean equals(Board board) {···
80
81      public final int calculateManhattan() {···
89
90      public final int calculateBlocker() {···
129
130     public List<Movement> getAllPossibleMovements() {···
177
178     @Override
179     public int compareTo(State other) {···
198
199 }
200
```

Board.java

```
Board.java > ...
1   import java.util.*;
2
3   public class Board {
4       private final boolean debug = true;
5       private final int width;
6       private final int height;
7       private boolean[][] board;
8       private Piece primaryPiece;
9       private List<Piece> listOfPieces;
10      private int x_Exit;
11      private int y_Exit;
12
13  >   public Board() {···
22
23  >   public Board(int width, int height, int x_Exit, int y_Exit) {···
33
34  >   public Board clone() {···
40
41  >   public void printDebug(String message) {···
46
47  >   public boolean isBoardEqual(Board b) {···
61
62  >   public boolean isGoal() {···
95
```

```
133 >   public void addPiece(Piece piece, boolean isPrimary) {···
177
178 >   public void removePiece(char symbol) {···
198
199 >   public boolean isMovementPossible(char symbol, Direction direction, int distance) {···
282
283
284 >   public void updateBoard(Piece piece, Direction direction, int distance) {···
327
328 >   public void MovementPiece(char symbol, Direction direction, int distance) {···
332
333 >   public Piece getPiece(char symbol) {···
344
345 >   public List<Piece> getListOfPieces() {···
348
349 >   public void setListOfPieces(List<Piece> listOfPieces) {···
352
353 >   public void printBoard() {···
435
436 >   public void printBooleanBoard() {···
448
449 >   public String hashCodeSigma() {···
457
458 >   public void printColouredBoard(State.Movement m) {···
```

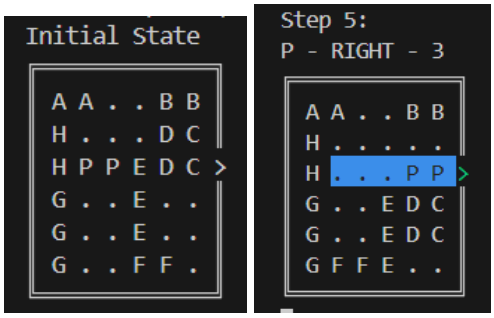
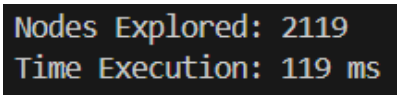
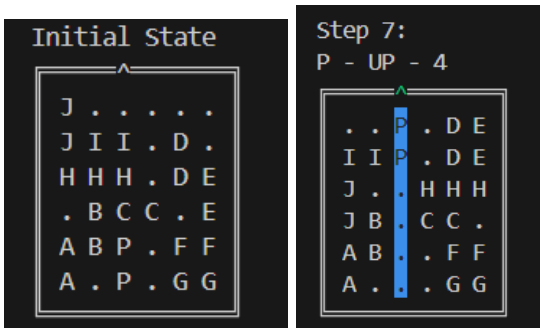
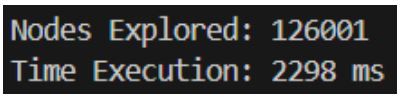
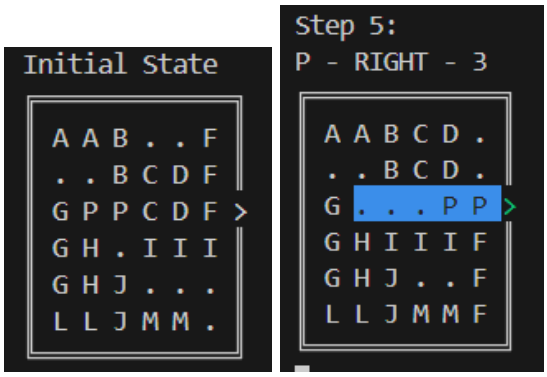
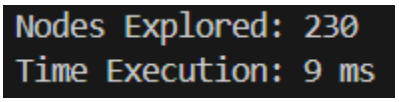
Piece.java

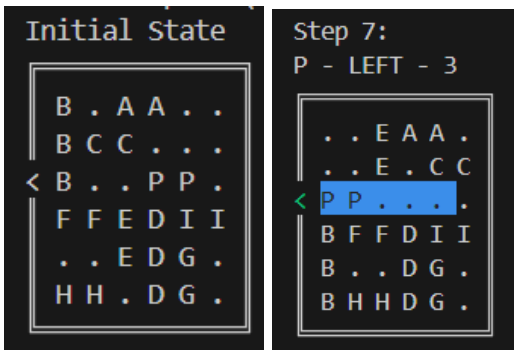
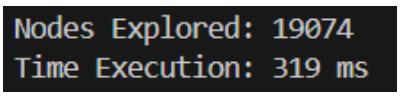
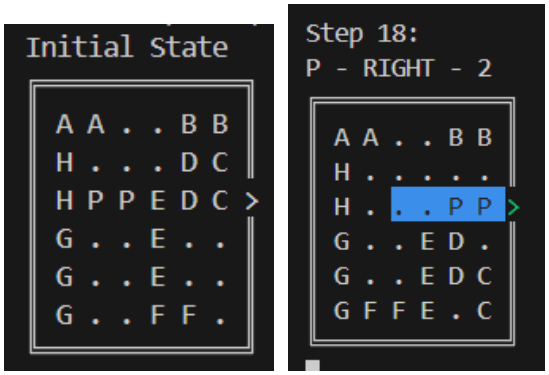
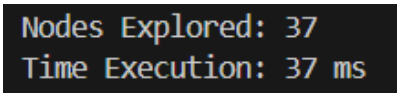
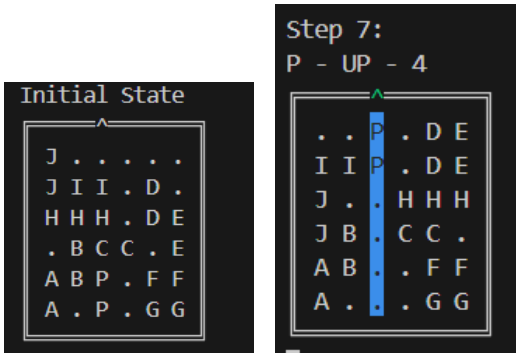
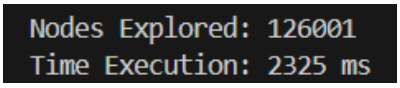
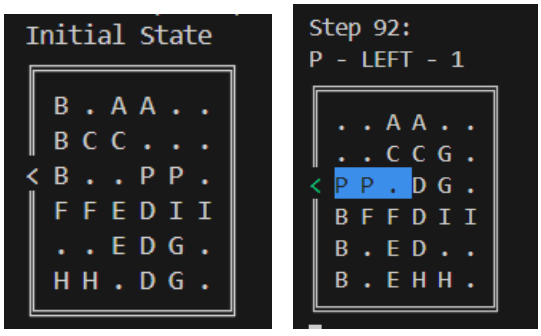
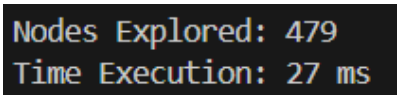
```java
public class Piece {
    private char symbol;
    private int x;
    private int y;
    private int size;
    private Orientation orientation;

    public Piece(char symbol, int x, int y, int size, Orientation orientation) {…

    public char getSymbol() {…

    public int getX() {…

    public int getY() {…

    public int getSize() {…

    public Orientation getOrientation() {…

    public boolean equal(Piece p) {…

    public void Movement(Direction m, int distance) {…

    public Piece clone() {…
}
```

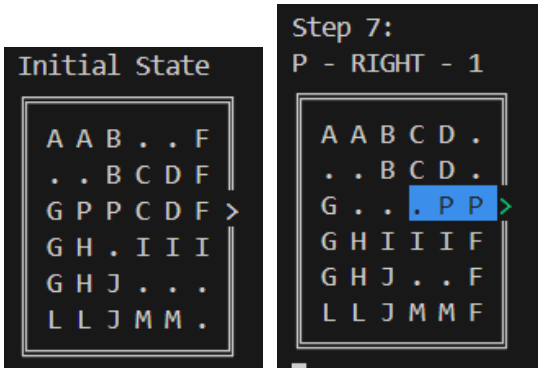
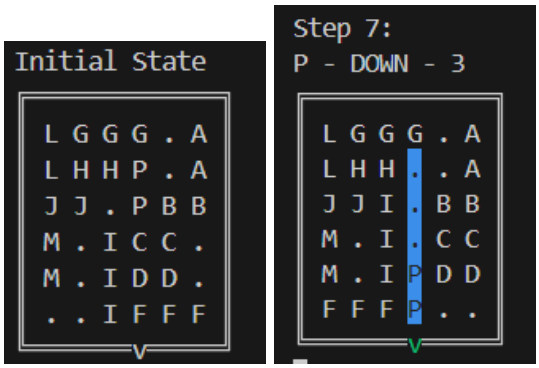
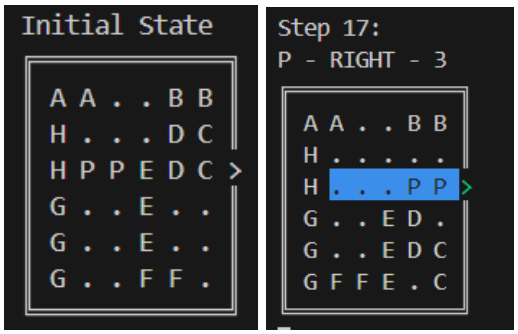
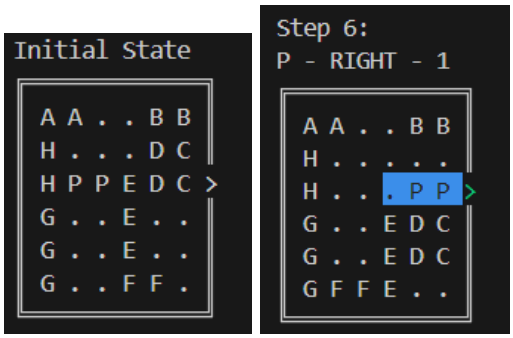
IOHandler.java

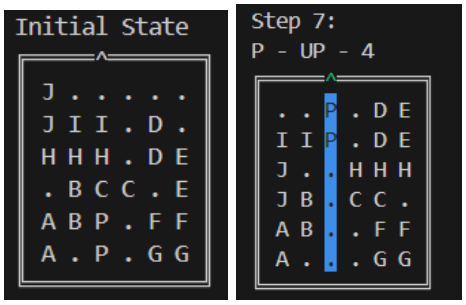
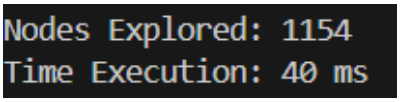
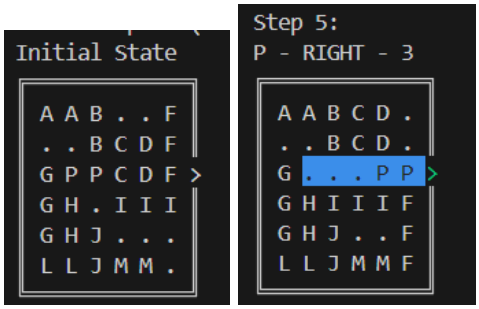
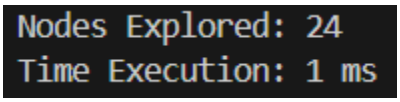
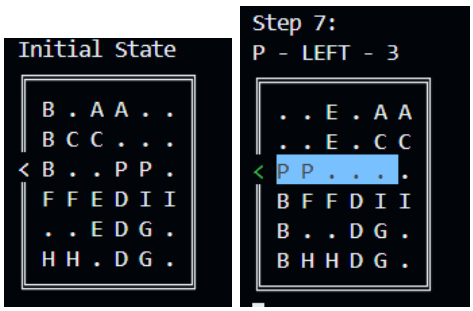
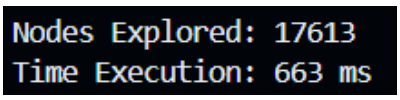
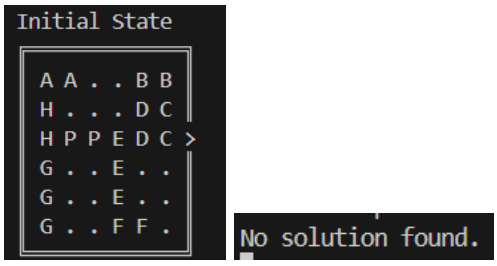
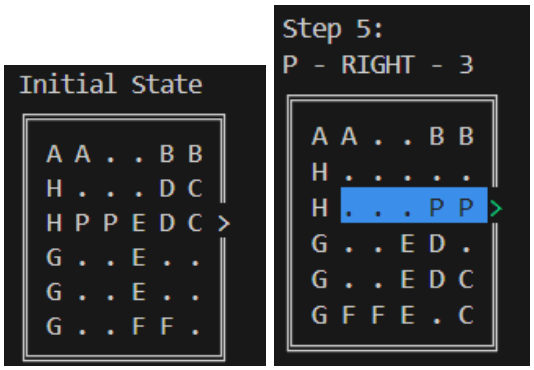
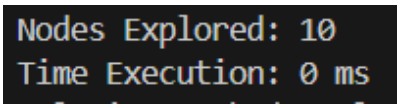
```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Arrays;
import java.util.List;

public class IOHandler {
    static boolean debug = true;

    public static void debugPrint(String message) {…

    public static void debugPrint(char[][] messages) {…

    public static Board inputFromFile(String filePath) {…
}
```

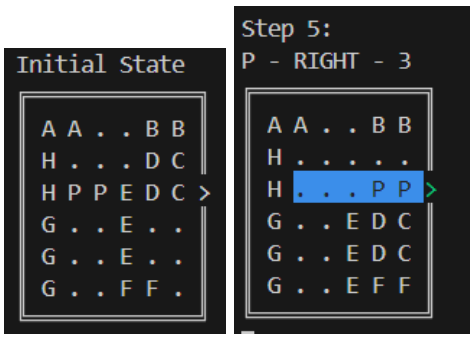
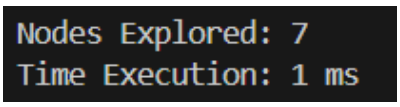
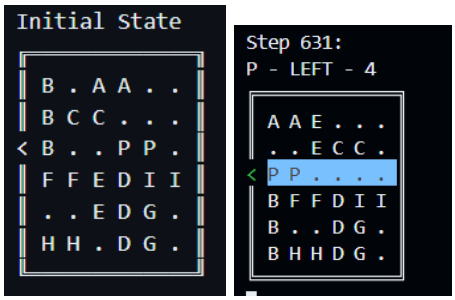
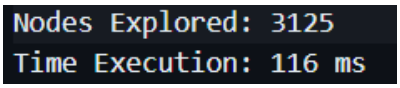
4.2 Testing

No	Algoritma & Heuristik	Input / Output	Node dikunjungi & Waktu Eksekusi
1	UCS		
2	UCS		
3	UCS		

4	**UCS**		
5	**GBFS / Distance to Exit**		
6	**GBFS / Distance to Exit**		
7	**GBFS / Blocker Count**		

8	GBFS / Blocker Count		Found a solution! Nodes Explored: 23 Time Execution: 2 ms
9	GBFS /Blocker Chaining		Nodes Explored: 11 Time Execution: 32 ms
10	GBFS /Blocker Chaining		Nodes Explored: 31 Time Execution: 25 ms
11	A* / Distance to Exit		Nodes Explored: 603 Time Execution: 51 ms

12	A* / Blocker Count		
13	A*/Blocker Chaining		
14			
15	Beam Search/5/ Blocker Count		-
16	Beam Search/10/ Blocker Count		

17	**Beam Search/999/ Blocker Count**		
18	**Beam Search/10/ Chain Blocking**		

BAB V : Hasil Analisis Algoritma

5.1 Analisis Algoritma Pathfinding

5.1.1 Algoritma UCS

- Selalu menghasilkan jalur yang optimal,
- Berguna jika mengutamakan rute termurah dibandingkan kecepatan mendapatkan rute,
- Lambat dan boros memori jika ruang pencariannya besar,
- Tidak menggunakan informasi arah ke tujuan

Kompleksitas waktu:

O(b^d)

Di mana:

- b = branching factor (jumlah kemungkinan gerakan per state)
- d = kedalaman solusi minimum

Ini setara dengan BFS dengan prioritas biaya.

5.1.2 Algoritma Greedy Best First Search

- Cepat karena mengincar tujuan sesuai heuristik,
- Efisien secara waktu pada banyak kasus,
- Lebih hemat memori,
- Rute yang didapat tidak terjamin optimal, mungkin bisa jauh dari optimal,
- Bisa terjebak jika heuristik buruk,
- Kompleksitas waktu:

  O(b^m)

  Di mana:
  - m = kedalaman maksimum pencarian (lebih dari kedalaman optimal karena GBFS bisa “salah jalan”)
  - Performa sangat tergantung pada kualitas heuristik.

5.1.3 Algoritma A*

- Optimal dan lengkap jika heuristik konsisten,
- Penjelajahan yang lebih efisien daripada UCS,
- Lebih memungkinkan mendapatkan jalur optimal dibanding GBFS,
- Konsumsi memori cukup besar,
- Bisa lambat jika heuristik tidak akurat,
- Kompleksitas waktu :

O(b^d)

Dalam kasus terburuk (heuristik buruk), tapi **lebih cepat dari UCS** jika heuristik baik.

Dapat mendekati:

O(|E|) atau O(nlogn)

Jika graf cukup kecil dan heuristik kuat.

5.1.4 Algoritma Beam Search

- Lebih hemat memori dibanding seluruh algoritma lainnya
- Cepat untuk permasalahan apapun,
- Tidak optimal dan tidak lengkap
- Sangat bergantung pada heuristik dan ukuran beamnya,
- Bisa gagal jika heuristik tidak akurat dan semua kandidat terbuang.
- Kompleksitas waktu :

  O(k.d)

  Di mana :

  - K = beam width
  - kedalaman maksimum pencarian

Lampiran

No	Poin	Ya	Tidak
1	Aplikasi berhasil dikompilasi tanpa kesalahan	✔	
2	Program berhasil dijalankan	✔	
3	Solusi yang diberikan program benar dan mematuhi aturan permainan	✔	
4	Program dapat membaca masukan berkas .txt dan menyimpan solusi berupa print board tahap per tahap dalam berkas .txt	✔	
5	[Bonus] Implementasi algoritma pathfinding alternatif	✔	
6	[Bonus] Implementasi 2 atau lebih heuristik alternatif	✔	
7	[Bonus] Program memiliki GUI		✔
8	Program dan laporan dibuat (kelompok) sendiri	✔	

~ Pranala Github : https://github.com/salmaanhaniif/Tucil3_13523034_13523056.git

Referensi

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf