# Laporan Tugas Kecil 03

IF2211 Strategi Algoritma

**Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding**


Disusun oleh:

| | |
|---|---|
| Muhammad Raihaan Perdana | (13523124) |
| M. Abizzar Gamadrian | (13523155) |

**PROGRAM STUDI TEKNIK INFORMATIKA**
**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA**
**INSTITUT TEKNOLOGI BANDUNG**
**JL. GANESA 10, BANDUNG 40132**

**2025**

\

# 1. Deskripsi Singkat

## Deskripsi Permasalahan

Rush Hour adalah permainan puzzle logika berbasis grid yang menantang pemain untuk menggeser kendaraan dalam sebuah kotak (umumnya berukuran 6x6) agar mobil utama (primary piece) dapat keluar dari kemacetan melalui pintu keluar di sisi papan. Setiap kendaraan hanya dapat bergerak lurus ke depan atau ke belakang sesuai dengan orientasinya (horizontal atau vertikal), dan tidak dapat berputar atau melewati kendaraan lain.

Permasalahan utama dalam Rush Hour adalah menemukan urutan gerakan yang minimal untuk memindahkan primary piece ke pintu keluar. Hal ini menjadi tantangan karena:

- Setiap piece memiliki orientasi tetap (horizontal atau vertikal)
- Piece tidak dapat melewati atau menembus piece lain
- Ruang gerak terbatas dalam grid
- Hanya primary piece yang dapat dikeluarkan melalui pintu keluar
- Tujuan adalah mencari solusi dengan jumlah langkah optimal

## Deskripsi Singkat Program

Program ini adalah implementasi solver untuk puzzle Rush Hour menggunakan bahasa pemrograman Java. Program dapat menyelesaikan puzzle Rush Hour dengan berbagai algoritma pathfinding dan menyediakan visualisasi solusi baik melalui command line interface (CLI) maupun graphical user interface (GUI).

### Algoritma Yang Digunakan

Program mengimplementasikan empat algoritma pathfinding:

1. **Uniform Cost Search (UCS)**: Algoritma uninformed search yang menjamin solusi optimal dengan memprioritaskan node dengan biaya path terendah.

2. **Greedy Best First Search**: Algoritma informed search yang menggunakan heuristic untuk memandu pencarian menuju goal, memprioritaskan node yang tampak paling dekat dengan tujuan.
3. ***A Search****: Algoritma informed search yang menggabungkan UCS dan Greedy, menggunakan fungsi evaluasi f(n) = g(n) + h(n) untuk menjamin solusi optimal dengan heuristic admissible.
4. ***IDA Search**** (Bonus): Algoritma iterative deepening A* yang mengkombinasikan keunggulan A* dengan efisiensi memori dari iterative deepening.

### Heuristic Yang Diimplementasikan

Program menyediakan tiga jenis heuristic:

1. **Blocking Heuristic**: Menghitung jumlah kendaraan yang menghalangi jalur primary piece menuju pintu keluar.
2. **Manhattan Heuristic**: Menghitung jarak Manhattan antara primary piece dengan pintu keluar.
3. **Combined Heuristic**: Menggabungkan blocking dan Manhattan heuristic dengan pembobotan untuk memberikan estimasi yang lebih akurat.

### Fitur Program

- **Input**: Membaca konfigurasi puzzle dari file teks dengan format yang telah ditentukan
- **Output CLI**: Menampilkan langkah-langkah solusi dengan visualisasi berwarna di terminal
- **Output GUI**: Visualisasi interaktif dengan animasi pergerakan kendaraan
- **Analisis Performa**: Menampilkan jumlah node yang dikunjungi dan waktu eksekusi untuk setiap algoritma
- **Penyimpanan Solusi**: Opsi untuk menyimpan hasil solusi ke file teks

Program dirancang untuk menangani puzzle Rush Hour dengan dimensi papan yang bervariasi dan dapat memberikan perbandingan performa antara berbagai algoritma pathfinding yang diimplementasikan.

---

# 2. Penjelasan Algoritma

## 2.1 Uniform Cost Search (UCS)

**Pseudocode**

```
function UCS(initial_state):
    frontier = PriorityQueue()  // ordered by path cost g(n)
    explored = Set()

    frontier.insert(Node(initial_state, cost=0))

    while frontier is not empty:
        node = frontier.removeMin()  // node with lowest g(n)

        if node.state is goal:
            return reconstructPath(node)

        if node.state not in explored:
            explored.add(node.state)

            for each action in getActions(node.state):
                child_state = apply(action, node.state)
                child_cost = node.cost + 1
                frontier.insert(Node(child_state, child_cost, node))

    return failure
```

## Penjelasan Algoritma

**Uniform Cost Search (UCS)** adalah algoritma pencarian uninformed yang menjamin solusi optimal dengan selalu mengeksplorasi node yang memiliki path cost terendah terlebih dahulu.

## Cara Kerja:

1. **Inisialisasi**: Mulai dengan state awal yang dimasukkan ke priority queue dengan cost 0
2. **Eksplorasi**: Selalu pilih dan eksplorasi node dengan cost terendah dari frontier
3. **Goal Test**: Periksa apakah state saat ini adalah goal state
4. **Ekspansi**: Generate semua successor state dan masukkan ke frontier dengan cost yang diupdate
5. **Duplicate Detection**: Hindari mengeksplorasi state yang sudah pernah dikunjungi

## Karakteristik:

- **Complete**: Ya, jika step cost > 0 dan branching factor terbatas
- **Optimal**: Ya, menjamin solusi dengan cost minimal
- **Time Complexity**: O(b^(C*/ε)) dimana C* adalah cost optimal dan ε minimum step cost
- **Space Complexity**: O(b^(C*/ε))

## Dalam Konteks Rush Hour:

- Setiap gerakan piece memiliki cost yang sama (1)
- UCS akan menemukan solusi dengan jumlah langkah minimal

- Algoritma mengeksplorasi semua kemungkinan gerakan secara sistematis tanpa preferensi arah tertentu

## 2.2 Greedy Best First Search

```
function GreedyBestFirst(initial_state):
    frontier = PriorityQueue()  // ordered by heuristic h(n)
    explored = Set()

    frontier.insert(Node(initial_state, heuristic(initial_state)))

    while frontier is not empty:
        node = frontier.removeMin()  // node with lowest h(n)

        if node.state is goal:
            return reconstructPath(node)

        if node.state not in explored:
            explored.add(node.state)

            for each action in getActions(node.state):
                child_state = apply(action, node.state)
                child_heuristic = heuristic(child_state)
                frontier.insert(Node(child_state, child_heuristic, node))

    return failure
```

### Penjelasan Algoritma

**Greedy Best First Search** adalah algoritma pencarian informed yang menggunakan fungsi heuristic untuk memandu pencarian menuju goal. Algoritma ini "serakah" karena selalu memilih node yang tampak paling menjanjikan berdasarkan estimasi jarak ke goal.

### Cara Kerja:

1. **Inisialisasi**: Mulai dengan state awal dan hitung nilai heuristik h(n)
2. **Eksplorasi**: Selalu pilih node dengan nilai heuristik terendah (tampak paling dekat ke goal)
3. **Goal Test**: Periksa apakah state saat ini adalah goal state
4. **Ekspansi**: Generate semua successor state dan hitung heuristik masing-masing
5. **Prioritas**: Urutkan node berdasarkan h(n), abaikan path cost g(n)

### Karakteristik:

- **Complete**: Tidak, bisa terjebak di local minima atau infinite loop
- **Optimal**: Tidak, fokus pada kecepatan bukan optimalitas
- **Time Complexity**: O(b^m) dalam worst case, tapi sering lebih cepat dalam praktik
- **Space Complexity**: O(b^m)

### Dalam Konteks Rush Hour:

- Menggunakan heuristic seperti jumlah piece yang menghalangi atau jarak Manhattan
- Akan memilih gerakan yang tampak mendekatkan primary piece ke pintu keluar
- Bisa menemukan solusi lebih cepat tapi tidak menjamin solusi optimal
- Rentan terjebak jika ada dead end atau konfigurasi yang menyesatkan

### Kelebihan dan Kekurangan:

**Kelebihan:**

- Lebih cepat karena fokus pada arah yang menjanjikan
- Memory efficient dibanding exhaustive search
- Intuitif dan mudah dipahami

**Kekurangan:**

- Tidak menjamin solusi optimal
- Bisa terjebak di local optimum
- Sangat bergantung pada kualitas heuristic yang digunakan

## 2.3 A* Search

**Pseudocode**

```
function AStar(initial_state):
    // inisialisasi priority queue yang diurutkan berdasarkan f(n) = g(n) + h(n)
    frontier = PriorityQueue()  // ordered by f(n) = g(n) + h(n)
    explored = Set()            // set untuk menyimpan state yang sudah dikunjungi

    // masukkan initial state dengan g(n)=0 dan h(n) dari heuristic
    initial_g = 0
    initial_h = heuristic(initial_state)
    frontier.insert(Node(initial_state, initial_g, initial_h))

    while frontier is not empty:
        // ambil node dengan nilai f(n) terendah (g(n) + h(n) minimum)
        node = frontier.removeMin()  // node with lowest f(n)

        // cek apakah sudah mencapai goal state
        if node.state is goal:
            return reconstructPath(node)  // kembalikan path solusi

        // hindari eksplorasi state yang sudah pernah dikunjungi
        if node.state not in explored:
            explored.add(node.state)    // tandai state sebagai sudah dikunjungi

            // generate semua successor state dari gerakan yang valid
            for each action in getActions(node.state):
                child_state = apply(action, node.state)     // terapkan aksi
                child_g = node.g + 1                        // tambah path cost
                child_h = heuristic(child_state)            // hitung heuristic child
                // masukkan child ke frontier dengan g(n), h(n), dan parent reference
```

```
                frontier.insert(Node(child_state, child_g, child_h, node))
    return failure  // tidak ada solusi ditemukan
```

## Penjelasan Algoritma

*A Search** adalah algoritma pencarian informed yang menggabungkan keunggulan UCS dan Greedy Best First Search. A* menggunakan fungsi evaluasi f(n) = g(n) + h(n), dimana g(n) adalah path cost dari start ke node n, dan h(n) adalah estimasi heuristic dari node n ke goal.

## Cara Kerja:

1. **Inisialisasi**: Mulai dengan state awal, g(n)=0, dan h(n) dari heuristic
2. **Eksplorasi**: Selalu pilih node dengan nilai f(n) = g(n) + h(n) terendah
3. **Goal Test**: Periksa apakah state saat ini adalah goal state
4. **Ekspansi**: Generate successor dengan g(n) = parent_g + step_cost dan h(n) baru
5. **Prioritas**: Urutkan berdasarkan f(n) yang menggabungkan cost dan heuristic

## Karakteristik:

- **Complete**: Ya, jika step cost > 0 dan branching factor terbatas
- **Optimal**: Ya, jika heuristic adalah admissible ($h(n) \leq h^*(n)$)
- **Time Complexity**: O(b^d) dimana d adalah depth solusi optimal
- **Space Complexity**: O(b^d)

## Dalam Konteks Rush Hour:

- Menggabungkan jumlah langkah yang sudah dilakukan (g(n)) dengan estimasi langkah tersisa (h(n))
- Mencari keseimbangan antara path cost dan arah menuju goal
- Dengan heuristic admissible, menjamin solusi optimal dengan efisiensi lebih baik dari UCS
- Lebih efisien dari Greedy karena mempertimbangkan cost yang sudah dikeluarkan

## Kelebihan dan Kekurangan:

**Kelebihan:**

- Menjamin solusi optimal (jika heuristic admissible)
- Lebih efisien dari UCS karena menggunakan heuristic
- Lebih reliable dari Greedy karena mempertimbangkan path cost
- Optimal dalam hal efficiency (tidak ada algoritma lain yang menjamin optimalitas dengan node expansion lebih sedikit)

**Kekurangan:**

- Membutuhkan heuristic yang baik untuk performa optimal
- Space complexity tinggi karena menyimpan semua node di frontier
- Lebih complex untuk diimplementasikan dibanding UCS atau Greedy
- Performa sangat bergantung pada kualitas dan admissibility heuristic

## 2.4 IDA* Search *(Bonus)*

### Pseudocode

```
function IDAStar(initial_state):
    // hitung threshold awal berdasarkan heuristic dari initial state
    threshold = heuristic(initial_state)

    // iterasi dengan threshold yang meningkat hingga solusi ditemukan
    while threshold < infinity:
        // lakukan depth-first search dengan batasan threshold
        result = search(initial_state, 0, threshold, empty_path)

        // jika solusi ditemukan, kembalikan path
        if result.found:
            return result.path

        // jika tidak ada lagi node untuk dieksplorasi
        if result.next_threshold = infinity:
            return failure

        // update threshold ke nilai minimum yang melebihi batas sebelumnya
        threshold = result.next_threshold

    return failure

function search(node, g, threshold, path):
    // hitung f(n) = g(n) + h(n) untuk node saat ini
    f = g + heuristic(node)

    // jika f(n) melebihi threshold, prune subtree ini
    if f > threshold:
        return (found=false, next_threshold=f)

    // cek apakah sudah mencapai goal state
    if node is goal:
        return (found=true, path=path+node)

    // tambahkan node ke path untuk menghindari cycle
    path.add(node)
    min_threshold = infinity

    // eksplorasi semua successor secara depth-first
    for each action in getActions(node):
        child = apply(action, node)

        // hindari cycle dengan mengecek apakah child sudah ada di path
        if child not in path:
            // rekursi dengan g(n) yang diupdate
            result = search(child, g+1, threshold, path)

            // jika solusi ditemukan pada subtree ini
            if result.found:
                return result

            // track threshold minimum untuk iterasi berikutnya
            if result.next_threshold < min_threshold:
```

```
            min_threshold = result.next_threshold

    // hapus node dari path (backtrack)
    path.remove(node)
    return (found=false, next_threshold=min_threshold)
```

## Penjelasan Algoritma

*IDA (Iterative Deepening A*)** adalah algoritma pencarian yang menggabungkan keunggulan A* dalam menemukan solusi optimal dengan efisiensi memori dari iterative deepening. Algoritma ini melakukan serangkaian depth-first search dengan batas f-cost yang meningkat secara iteratif.

## Cara Kerja:

1. **Inisialisasi Threshold**: Mulai dengan threshold = h(initial_state)
2. **Iterative Search**: Lakukan DFS dengan batasan f(n) ≤ threshold
3. **Threshold Update**: Jika tidak ditemukan solusi, update threshold ke nilai f(n) minimum yang melebihi batas sebelumnya
4. **Depth-First Exploration**: Dalam setiap iterasi, eksplorasi secara DFS sambil memangkas node dengan f(n) > threshold
5. **Cycle Detection**: Gunakan path untuk menghindari cycle dalam pencarian

## Karakteristik:

- **Complete**: Ya, jika step cost > 0 dan branching factor terbatas
- **Optimal**: Ya, jika heuristic adalah admissible (sama seperti A*)
- **Time Complexity**: O(b^d) sama seperti A*, tapi dengan overhead iterasi
- **Space Complexity**: O(d) jauh lebih efisien dibanding A* yang O(b^d)

## Dalam Konteks Rush Hour:

- Menggunakan memori linear terhadap depth solusi, bukan eksponensial
- Setiap iterasi mencari solusi dengan batas f-cost tertentu
- Ideal untuk puzzle dengan space state besar dimana A* standar kehabisan memori
- Menjamin solusi optimal dengan footprint memori yang minimal

## Keunggulan dan Kekurangan:

**Keunggulan:**

- Memory efficient: O(d) vs O(b^d) untuk A*
- Tetap menjamin solusi optimal (jika heuristic admissible)
- Tidak perlu menyimpan frontier yang besar
- Cocok untuk problem dengan space constraint

**Kekurangan:**

- Overhead waktu dari iterasi berulang
- Node yang sama mungkin dieksplorasi berkali-kali di iterasi berbeda
- Lebih lambat dari A* standar jika memori tidak menjadi masalah
- Implementasi lebih complex karena sifat rekursif dan threshold management

### Perbandingan dengan A*:

- **Memori**: IDA* jauh lebih hemat (O(d) vs O(b^d))
- **Waktu**: A* umumnya lebih cepat karena tidak ada redundant exploration
- **Praktis**: IDA* lebih baik untuk problem besar dengan keterbatasan memori
- **Optimalitas**: Kedua algoritma sama-sama optimal dengan heuristic admissible

# Jawaban Pertanyaan

## 1. Definisi f(n) dan g(n) Sesuai Slide Kuliah

### g(n) - Path Cost Function

**Definisi**: g(n) adalah biaya aktual dari simpul awal (start node) hingga simpul n dalam pohon pencarian.

**Dalam konteks Rush Hour**:

- g(n) = jumlah gerakan piece yang telah dilakukan dari state awal hingga state n
- Setiap gerakan piece (satu cell ke arah yang valid) memiliki cost 1
- g(initial_state) = 0
- g(child) = g(parent) + 1 untuk setiap ekspansi node

### f(n) - Evaluation Function

**Definisi**: f(n) adalah fungsi evaluasi yang digunakan untuk menentukan prioritas eksplorasi node.

**Untuk masing-masing algoritma**:

- **UCS**: f(n) = g(n)
- **Greedy**: f(n) = h(n)
- **A***: f(n) = g(n) + h(n)
- **IDA***: f(n) = g(n) + h(n) (dengan threshold-based pruning)

## 2. Admissibility Heuristic pada Algoritma A*

## Definisi Admissible

Sebuah heuristic h(n) disebut **admissible** jika untuk setiap node n, $h(n) \leq h^*(n)$, dimana $h^*(n)$ adalah biaya minimum sebenarnya dari node n ke goal state. Dengan kata lain, heuristic admissible tidak pernah **overestimate** biaya menuju goal.

## Analisis Heuristic dalam Implementation

### Blocking Heuristic

- **Definisi**: Menghitung jumlah piece yang menghalangi jalur primary piece ke exit
- **Admissible**: **YA**
- **Justifikasi**: Setiap piece penghalang minimal perlu 1 gerakan untuk dipindahkan. Dalam kasus terburuk, setiap piece penghalang perlu tepat 1 gerakan untuk membersihkan jalur. Karena itu, $h(n) \leq h^*(n)$.

### Manhattan Heuristic

- **Definisi**: Jarak Manhattan primary piece ke exit position
- **Admissible**: **YA**
- **Justifikasi**: Jarak Manhattan memberikan batas bawah minimum langkah yang diperlukan primary piece untuk mencapai exit, tanpa mempertimbangkan obstacle. Karena primary piece harus menempuh jarak ini (mungkin lebih karena ada penghalang), $h(n) \leq h^*(n)$.

### Combined Heuristic

- **Definisi**: Kombinasi weighted dari Blocking dan Manhattan: $h(n) = 2 \times \text{blocking} + \text{manhattan}$
- **Admissible**: **TIDAK** (dalam implementasi saat ini)
- **Justifikasi**: Pembobotan 2× pada blocking heuristic dapat menyebabkan overestimate. Jika ada 3 piece penghalang, combined heuristic memberikan kontribusi 6 dari blocking component saja, padahal mungkin ketiga piece bisa dipindahkan dengan total < 6 gerakan.

## Kesimpulan Admissibility

Heuristic **Blocking** dan **Manhattan** adalah admissible, sehingga A* dengan kedua heuristic ini menjamin solusi optimal. Namun **Combined Heuristic** dengan bobot 2× tidak admissible.

# 3. Perbandingan UCS dan BFS pada Rush Hour

## Analisis Teoritis

**UCS sama dengan BFS jika dan hanya jika semua edge dalam graf memiliki cost yang sama.**

## Dalam Konteks Rush Hour

- **Edge Cost**: Setiap gerakan piece memiliki cost yang sama = 1
- **Struktur Graf**: State space Rush Hour membentuk graf tidak berarah dengan uniform cost
- **Kesimpulan**: **YA, UCS sama dengan BFS**

### Pembuktian

1. **Urutan Node**: Kedua algoritma akan mengeksplorasi node berdasarkan kedalaman yang sama (BFS) atau cost yang sama (UCS), yang equivalen karena cost = depth
2. **Path yang Dihasilkan**: Kedua algoritma akan menghasilkan path dengan panjang minimum yang sama
3. **Frontier Management**: Priority queue UCS dengan uniform cost berperilaku identik dengan queue FIFO BFS

### Perbedaan Implementation

- BFS menggunakan FIFO queue
- UCS menggunakan priority queue, tapi dengan uniform cost, urutannya sama dengan FIFO

## 4. Efisiensi A* vs UCS pada Rush Hour

### Analisis Teoritis

*A lebih efisien dari UCS** karena menggunakan informasi heuristic untuk mengarahkan pencarian.

### Pembuktian Efisiensi

#### Node Expansion

- **UCS**: Mengeksplorasi semua node dengan $g(n) \leq C^*$ (dimana $C^*$ = cost solusi optimal)
- **A***: Hanya mengeksplorasi node dengan $f(n) = g(n) + h(n) \leq C^*$
- **Hasil**: A* mengeksplorasi subset dari node yang dieksplorasi UCS

#### Time Complexity

- **UCS**: $O(b^{(C^*/\varepsilon)})$ dimana $\varepsilon$ = minimum step cost
- **A***: $O(b^d)$ dimana d = depth solusi optimal
- **Dalam Rush Hour**: Karena step cost uniform ($\varepsilon=1$), A* umumnya lebih cepat

#### Space Complexity

- **UCS**: $O(b^{(C^*/\varepsilon)})$
- **A***: $O(b^d)$
- **Hasil**: Keduanya eksponensial, tapi A* biasanya lebih kecil konstanta-nya

### Keunggulan A* pada Rush Hour

1. **Guided Search**: Heuristic mengarahkan ke goal, mengurangi blind exploration
2. **Early Termination**: Dapat memangkas cabang yang tidak promising
3. **Domain Knowledge**: Memanfaatkan informasi spesifik Rush Hour (blocking pieces, distance to exit)

## 5. Optimalitas Greedy Best First Search

### Analisis Teoritis

**Greedy Best First Search TIDAK menjamin solusi optimal** untuk penyelesaian Rush Hour.

### Pembuktian Non-Optimality

#### Karakteristik Greedy

- Hanya mempertimbangkan h(n), mengabaikan g(n)
- Bersifat myopic: fokus pada keuntungan jangka pendek
- Tidak ada mekanisme untuk mempertimbangkan total cost path

#### Contoh Kasus di Rush Hour

Misalkan ada dua path:

- **Path A**: 3 langkah, tetapi primary piece tampak jauh dari exit
- **Path B**: 5 langkah, tetapi primary piece dekat dengan exit

Greedy akan memilih path B karena h(n) lebih kecil, padahal path A adalah optimal.

#### Problem Fundamental

1. **Local Optima**: Bisa terjebak di konfigurasi yang tampak baik tapi sebenarnya dead-end
2. **No Backtracking**: Tidak ada mekanisme untuk "undo" keputusan yang buruk
3. **Irrevocable Decisions**: Setiap pilihan bersifat final tanpa pertimbangan global

### Properties Greedy pada Rush Hour

- **Complete**: TIDAK (bisa infinite loop atau terjebak)
- **Optimal**: TIDAK (seperti dibuktikan di atas)
- **Time Complexity**: O(b^m) worst case, tapi often better in practice
- **Space Complexity**: O(b^m)

### Kesimpulan

Meskipun Greedy seringkali menemukan solusi dengan cepat dalam praktik, algoritma ini **tidak menjamin** bahwa solusi yang ditemukan adalah optimal. Untuk menjamin optimalitas pada Rush Hour, gunakan UCS, BFS, atau A* dengan heuristic admissible.

# 3. Struktur Program

## Kelas-kelas Utama

**Board.java** - Representasi papan permainan

- Menyimpan grid 2D dan posisi pieces
- Implementasi game rules dan move validation
- Deteksi goal state

**Piece.java** - Representasi kendaraan dalam permainan

- Menyimpan posisi, orientasi, dan ukuran piece
- Validasi pergerakan berdasarkan orientasi

**Node.java** - Node dalam search tree

- Menyimpan state board, parent node, dan move
- Implementasi Comparable untuk priority queue

**PathFinder Interface** - Interface untuk algoritma pathfinding

- UCS.java, GreedyBestFirstSearch.java, AStar.java, IDAStar.java

**Heuristic Interface** - Interface untuk fungsi heuristic

- BlockingHeuristic.java, ManhattanHeuristic.java, CombinedHeuristic.java

**FileParser.java** - Parser untuk file input konfigurasi board

**Visualizer** - Output visualization

- CLIVisualizer.java (command line output)
- GUIVisualizer.java (graphical interface)

## Alur Program

1. Parse file input untuk mendapatkan konfigurasi board
2. Inisialisasi algoritma pathfinding berdasarkan input user
3. Jalankan algoritma untuk mencari solusi

4. Tampilkan statistik pencarian (nodes visited, execution time)
5. Visualisasikan solusi melalui CLI atau GUI
6. Opsional: simpan solusi ke file

---

# 4. Source Code

## UCS.java

```java
package algorithm;

import java.util.*;

import entity.Board;
import entity.Move;
import entity.Node;

public class UCS implements PathFinder {
    private int nodesVisited;
    private long executionTime;

    @Override
    public List<Node> findPath(Board initialBoard) {
        long startTime = System.currentTimeMillis();
        nodesVisited = 0;

        PriorityQueue<Node> frontier = new PriorityQueue<>();
        Set<String> explored = new HashSet<>();

        Node initialNode = new Node(initialBoard, null, null, 0, 0, 0);
        frontier.add(initialNode);

        while (!frontier.isEmpty()) {
            Node current = frontier.poll();
            nodesVisited++;

            if (current.getBoard().isGoalState()) {
                executionTime = System.currentTimeMillis() - startTime;
                return reconstructPath(current);
            }

            String boardString = current.getBoard().toString();
            if (explored.contains(boardString)) {
                continue;
            }
            explored.add(boardString);

            for (Move move : current.getBoard().getPossibleMoves()) {
```

```
                Board newBoard = current.getBoard().copy();
                if (newBoard.movePiece(move.getPieceId(), move.getDirection())) {
                    Node child = new Node(newBoard, current, move, current.getCost()
+ 1, 0, 0);
                    frontier.add(child);
                }
            }
        }

        executionTime = System.currentTimeMillis() - startTime;
        return null;
    }

    private List<Node> reconstructPath(Node goal) {
        List<Node> path = new ArrayList<>();
        Node current = goal;

        while (current != null) {
            path.add(current);
            current = current.getParent();
        }

        Collections.reverse(path);
        return path;
    }

    @Override
    public int getNodesVisited() {
        return nodesVisited;
    }

    @Override
    public long getExecutionTime() {
        return executionTime;
    }

    @Override
    public String getAlgorithmName() {
        return "Uniform Cost Search";
    }
}
```

## AStar.java

```
package algorithm;

import algorithm.heuristics.Heuristic;
import entity.Board;
```

```java
import entity.Move;
import entity.Node;
import java.util.*;

public class AStar implements PathFinder {
    private final Heuristic heuristic;
    private int nodesVisited;
    private long executionTime;

    public AStar(Heuristic heuristic) {
        this.heuristic = heuristic;
    }

    @Override
    public List<Node> findPath(Board initialBoard) {
        long startTime = System.currentTimeMillis();
        nodesVisited = 0;

        PriorityQueue<Node> frontier = new PriorityQueue<>();
        Set<String> explored = new HashSet<>();

        int initialHeuristic = heuristic.calculate(initialBoard);

        Node initialNode = new Node(initialBoard, null, null, 0, initialHeuristic, 2);
        frontier.add(initialNode);

        while (!frontier.isEmpty()) {
            Node current = frontier.poll();
            nodesVisited++;

            if (current.getBoard().isGoalState()) {
                executionTime = System.currentTimeMillis() - startTime;
                return reconstructPath(current);
            }

            String boardString = current.getBoard().toString();
            if (explored.contains(boardString)) {
                continue;
            }
            explored.add(boardString);

            for (Move move : current.getBoard().getPossibleMoves()) {
                Board newBoard = current.getBoard().copy();
                if (newBoard.movePiece(move.getPieceId(), move.getDirection())) {
                    int h = heuristic.calculate(newBoard);
                    Node child = new Node(newBoard, current, move, current.getCost() +
1, h, 2);
                    frontier.add(child);
                }
            }
        }
```

```
        executionTime = System.currentTimeMillis() - startTime;
        return null; // No solution found
    }

    private List<Node> reconstructPath(Node goal) {
        List<Node> path = new ArrayList<>();
        Node current = goal;

        while (current != null) {
            path.add(current);
            current = current.getParent();
        }

        Collections.reverse(path);
        return path;
    }

    @Override
    public int getNodesVisited() {
        return nodesVisited;
    }

    @Override
    public long getExecutionTime() {
        return executionTime;
    }

    @Override
    public String getAlgorithmName() {
        return "A* Search (" + heuristic.getName() + ")";
    }
}
```

## GreedyBestFirstSearch.java

```
package algorithm;

import algorithm.heuristics.Heuristic;
import entity.Board;
import entity.Move;
import entity.Node;
import java.util.*;

public class GreedyBestFirstSearch implements PathFinder {
    private Heuristic heuristic;
    private int nodesVisited;
    private long executionTime;

    public GreedyBestFirstSearch(Heuristic heuristic) {
```

```
        this.heuristic = heuristic;
    }

    @Override
    public List<Node> findPath(Board initialBoard) {
        long startTime = System.currentTimeMillis();
        nodesVisited = 0;

        PriorityQueue<Node> frontier = new PriorityQueue<>();
        Set<String> explored = new HashSet<>();

        int initialHeuristic = heuristic.calculate(initialBoard);

        Node initialNode = new Node(initialBoard, null, null, 0, initialHeuristic, 1);
        frontier.add(initialNode);

        while (!frontier.isEmpty()) {
            Node current = frontier.poll();
            nodesVisited++;

            if (current.getBoard().isGoalState()) {
                executionTime = System.currentTimeMillis() - startTime;
                return reconstructPath(current);
            }

            String boardString = current.getBoard().toString();
            if (explored.contains(boardString)) {
                continue;
            }
            explored.add(boardString);

            for (Move move : current.getBoard().getPossibleMoves()) {
                Board newBoard = current.getBoard().copy();
                if (newBoard.movePiece(move.getPieceId(), move.getDirection())) {
                    int h = heuristic.calculate(newBoard);
                    Node child = new Node(newBoard, current, move, current.getCost() +
1, h, 1);
                    frontier.add(child);
                }
            }
        }

        executionTime = System.currentTimeMillis() - startTime;
        return null;
    }

    private List<Node> reconstructPath(Node goal) {
        List<Node> path = new ArrayList<>();
        Node current = goal;

        while (current != null) {
```

```
            path.add(current);
            current = current.getParent();
        }

        Collections.reverse(path);
        return path;
    }

    @Override
    public int getNodesVisited() {
        return nodesVisited;
    }

    @Override
    public long getExecutionTime() {
        return executionTime;
    }

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

    public Heuristic getHeuristic() {
        return heuristic;
    }

    public void setHeuristic(Heuristic heuristic) {
        this.heuristic = heuristic;
    }
}
```

## Heuristic classes

Blocking

```
package algorithm.heuristics;

import entity.Board;
import entity.Orientation;
import entity.Piece;
import entity.Position;

public class BlockingHeuristic implements Heuristic {
    @Override
    public int calculate(Board board) {
        char[][] grid = board.getGrid();
```

```
        Position exitPos = board.getExitPosition();
        Piece primaryPiece = board.getPieces().get(board.getPrimaryPieceId());
        int blockingCount = 0;

        if (primaryPiece.getOrientation() == Orientation.HORIZONTAL) {
            // Count pieces between the primary piece and the exit (horizontal)
            int row = primaryPiece.getPositions().get(0).getRow();
            int startCol = primaryPiece.getBackPosition().getCol() + 1;
            int endCol = exitPos.getCol();

            if (startCol > endCol) {
                // Exit is on the left
                startCol = exitPos.getCol();
                endCol = primaryPiece.getFrontPosition().getCol() - 1;
            }

            for (int col = startCol; col <= endCol; col++) {
                if (grid[row][col] != '.' && grid[row][col] != 'K') {
                    blockingCount++;
                }
            }
        } else {
            // Count pieces between the primary piece and the exit (vertical)
            int col = primaryPiece.getPositions().get(0).getCol();
            int startRow = primaryPiece.getBackPosition().getRow() + 1;
            int endRow = exitPos.getRow();

            if (startRow > endRow) {
                // Exit is above
                startRow = exitPos.getRow();
                endRow = primaryPiece.getFrontPosition().getRow() - 1;
            }

            for (int row = startRow; row <= endRow; row++) {
                if (grid[row][col] != '.' && grid[row][col] != 'K') {
                    blockingCount++;
                }
            }
        }

        return blockingCount;
    }

    @Override
    public String getName() {
        return "Blocking Vehicles";
    }
}
```

Manhattan

```java
package algorithm.heuristics;

import entity.Board;
import entity.Orientation;
import entity.Piece;
import entity.Position;

public class ManhattanHeuristic implements Heuristic {
    @Override
    public int calculate(Board board) {
        Position exitPos = board.getExitPosition();
        Piece primaryPiece = board.getPieces().get(board.getPrimaryPieceId());

        if (primaryPiece.getOrientation() == Orientation.HORIZONTAL) {
            // Calculate Manhattan distance horizontally
            int pieceCol = primaryPiece.getBackPosition().getCol();
            if (exitPos.getCol() < pieceCol) {
                // Exit is on the left
                pieceCol = primaryPiece.getFrontPosition().getCol();
            }

            return Math.abs(pieceCol - exitPos.getCol());
        } else {
            // Calculate Manhattan distance vertically
            int pieceRow = primaryPiece.getBackPosition().getRow();
            if (exitPos.getRow() < pieceRow) {
                // Exit is above
                pieceRow = primaryPiece.getFrontPosition().getRow();
            }

            return Math.abs(pieceRow - exitPos.getRow());
        }
    }

    @Override
    public String getName() {
        return "Manhattan Distance";
    }
}
```

Combined:

```java
package algorithm.heuristics;

import entity.Board;

public class CombinedHeuristic implements Heuristic {
    private BlockingHeuristic blockingHeuristic;
```

```
    private ManhattanHeuristic manhattanHeuristic;

    public CombinedHeuristic() {
        this.blockingHeuristic = new BlockingHeuristic();
        this.manhattanHeuristic = new ManhattanHeuristic();
    }

    @Override
    public int calculate(Board board) {
        // Combine both heuristics with weights
        int blocking = blockingHeuristic.calculate(board);
        int manhattan = manhattanHeuristic.calculate(board);

        // Blocking pieces are more important as they directly block the path
        return blocking * 2 + manhattan;
    }

    @Override
    public String getName() {
        return "Combined (Blocking + Manhattan)";
    }
}
```

---

# 5. Tangkapan Layar Input & Output

# Algoritma UCS

### Percobaan 1: File SP_ez.txt

**Input:**

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

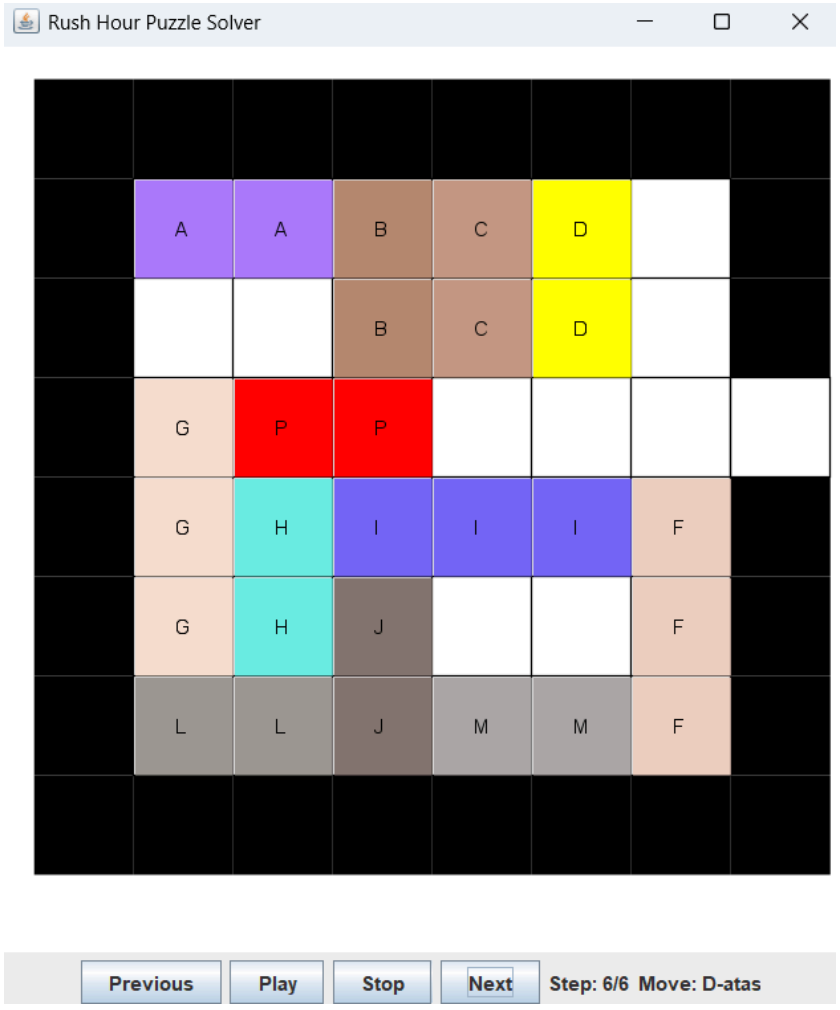
**Parameter:**

- Output:

- Jumlah node yang dikunjungi: 1076
- Waktu eksekusi algoritma: 3 ms
- Total waktu program: 3 ms
- Jumlah langkah: 6
- CLI (hanya menampilkan langkah terakhir, karena hasil yang panjang)

```
Gerakan 6: D-atas
AABCD.
..BCD.
GPP...K
GHIIIF
GHJ..F
LLJMMF
```

- GUI (hanya menampilkan langkah terakhir, karena hasil yang panjang)



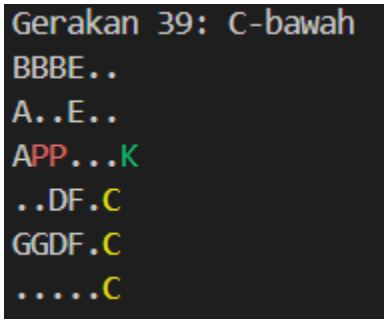

## Percobaan 2: File 24_med.txt

**Input:**

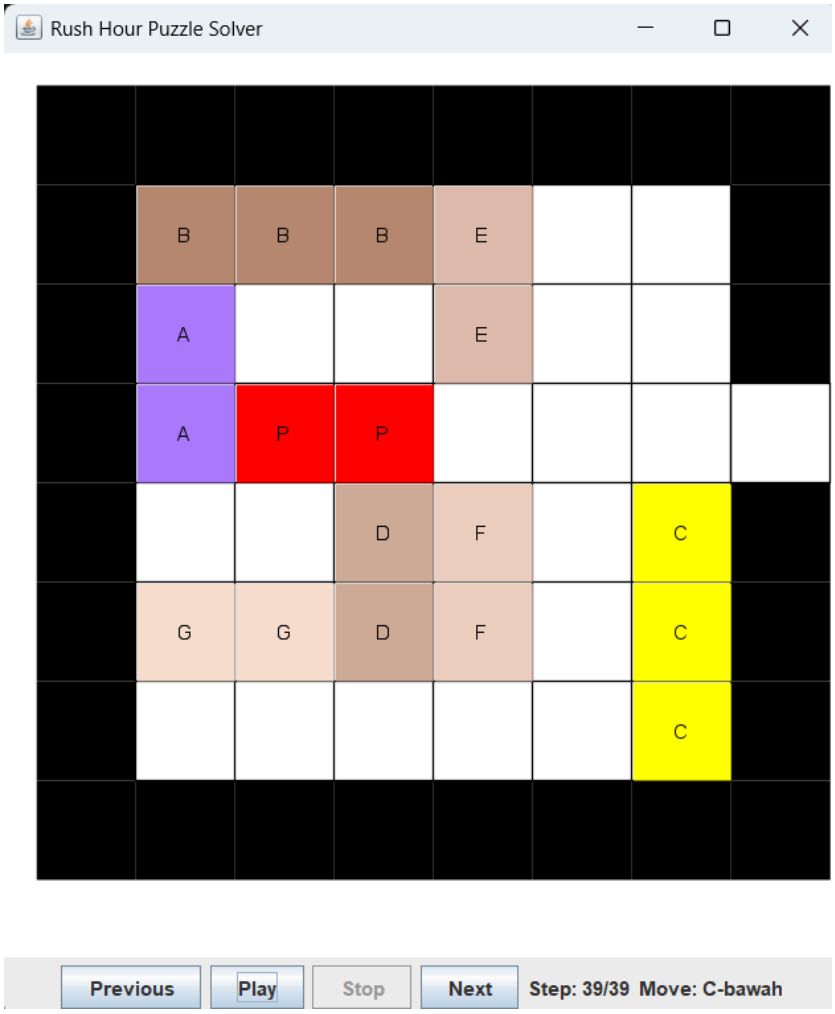
```
6 6
7
ABBB.C
A.DE.C
PPDE.CK
...F..
...FGG
......
```

**Parameter:**

- Output:
- Jumlah node yang dikunjungi: 9092
- Waktu eksekusi algoritma: 17 ms
- Total waktu program: 17 ms
- Jumlah langkah: 39
- CLI (hanya menampilkan langkah terakhir, karena hasil yang panjang)



- GUI (hanya menampilkan langkah terakhir, karena hasil yang panjang)




## Percobaan 3: File 17_hard.txt

**Input:**

```
6 6
14
AAB.CC
D.BEEE
DFPPGHK
IFJJGH
IFLLLM
INNOOM
```

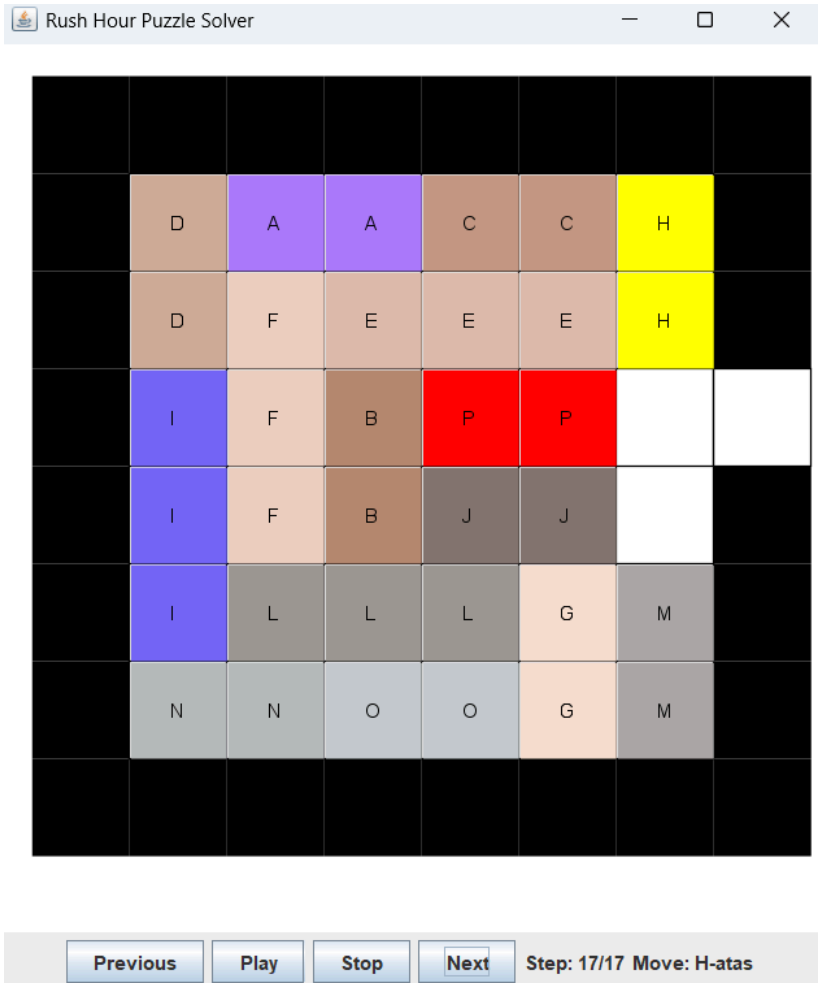
**Parameter:**

- Output:
- Jumlah node yang dikunjungi: 192

- Waktu eksekusi algoritma: 4 ms
- Total waktu program: 4 ms
- Jumlah langkah: 17
- CLI (hanya menampilkan langkah terakhir, karena hasil yang panjang)

```
Gerakan 17: H-atas
DAACCH
DFEEEH
IFBPP.K
IFBJJ.
ILLLGM
NNOOGM
```

- GUI (hanya menampilkan langkah terakhir, karena hasil yang panjang)



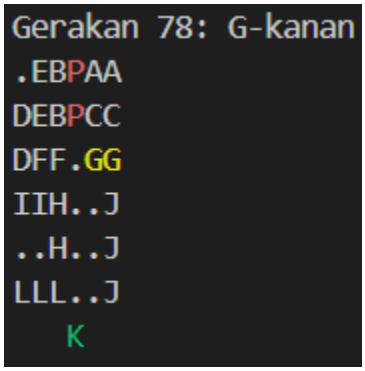

## Percobaan 4: 50_hard.txt

**Input:**

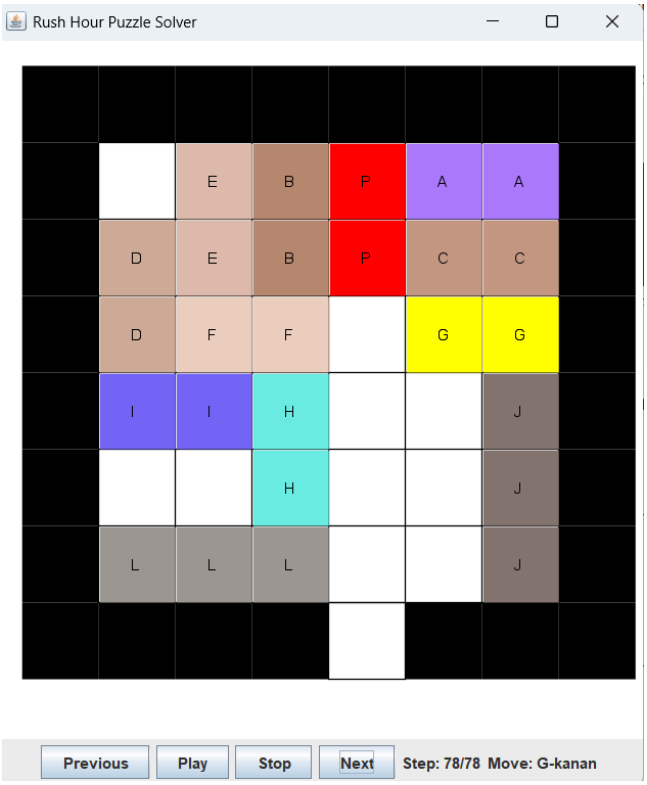
```
6 6
11
AABP..
CCBP..
DEFFGG
DEHIIJ
..H..J
.LLL.J
   K
```

**Parameter:**

- Output:
- Jumlah node yang dikunjungi: 18747
- Waktu eksekusi algoritma: 40 ms
- Total waktu program: 40 ms
- Jumlah langkah: 78
- CLI (hanya menampilkan langkah terakhir, karena hasil yang panjang)



- GUI (hanya menampilkan langkah terakhir, karena hasil yang panjang)




# Algoritma Greedy Best First Search

## Percobaan 1: File SP_ez.txt

**Input:**

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

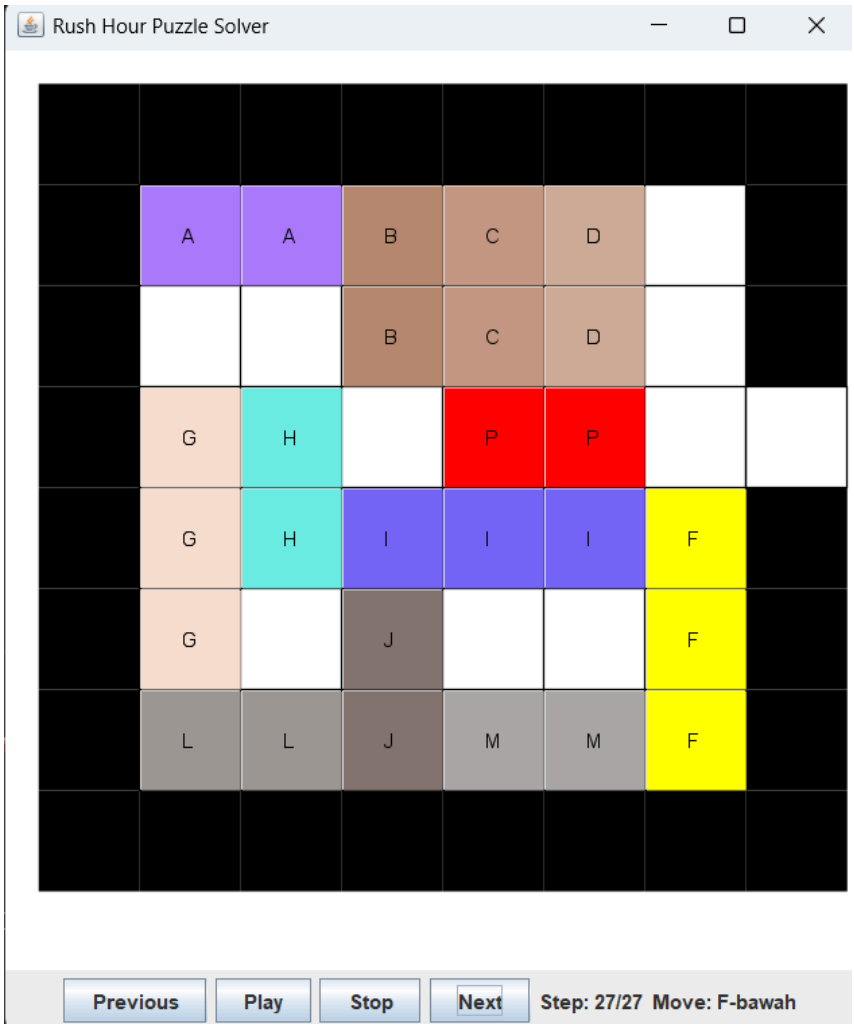
**Parameter:**

- Heuristic: Blocking Heuristic
- Output:
- Jumlah node yang dikunjungi: 208

- Waktu eksekusi algoritma: 3 ms
- Total waktu program: 3 ms
- Jumlah langkah: 27
- CLI (hanya menampilkan langkah terakhir, karena hasil yang panjang)

```
Gerakan 27: F-bawah
AABCD.
..BCD.
GH.PP.K
GHIIIF
G.J..F
LLJMMF
```

- GUI (hanya menampilkan langkah terakhir, karena hasil yang panjang)



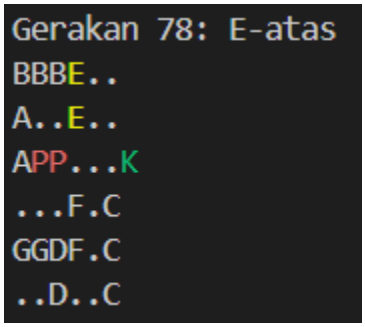

## Percobaan 2: File 24_med.txt

**Input:**

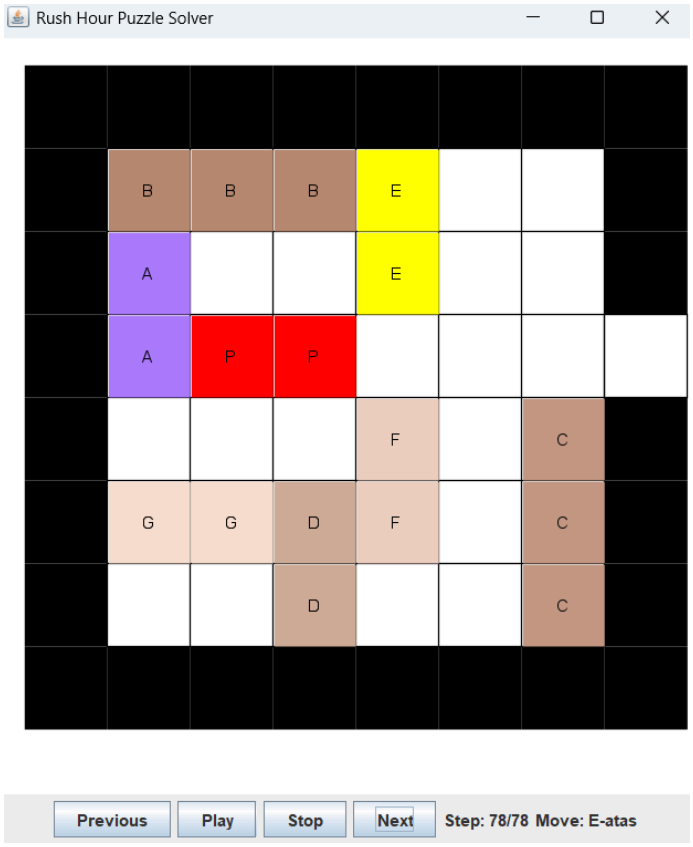
```
6 6
7
ABBB.C
A.DE.C
PPDE.CK
...F..
...FGG
......
```

**Parameter:**

- Heuristic: Blocking Heuristic
- Output:
- Jumlah node yang dikunjungi: 3934
- Waktu eksekusi algoritma: 15 ms
- Total waktu program: 16 ms
- Jumlah langkah: 78
- CLI (hanya menampilkan langkah terakhir, karena hasil yang panjang)



- GUI (hanya menampilkan langkah terakhir, karena hasil yang panjang)




## Percobaan 3: File 17_hard.txt

**Input:**

```
6 6
14
AAB.CC
D.BEEE
DFPPGHK
IFJJGH
IFLLLM
INNOOM
```

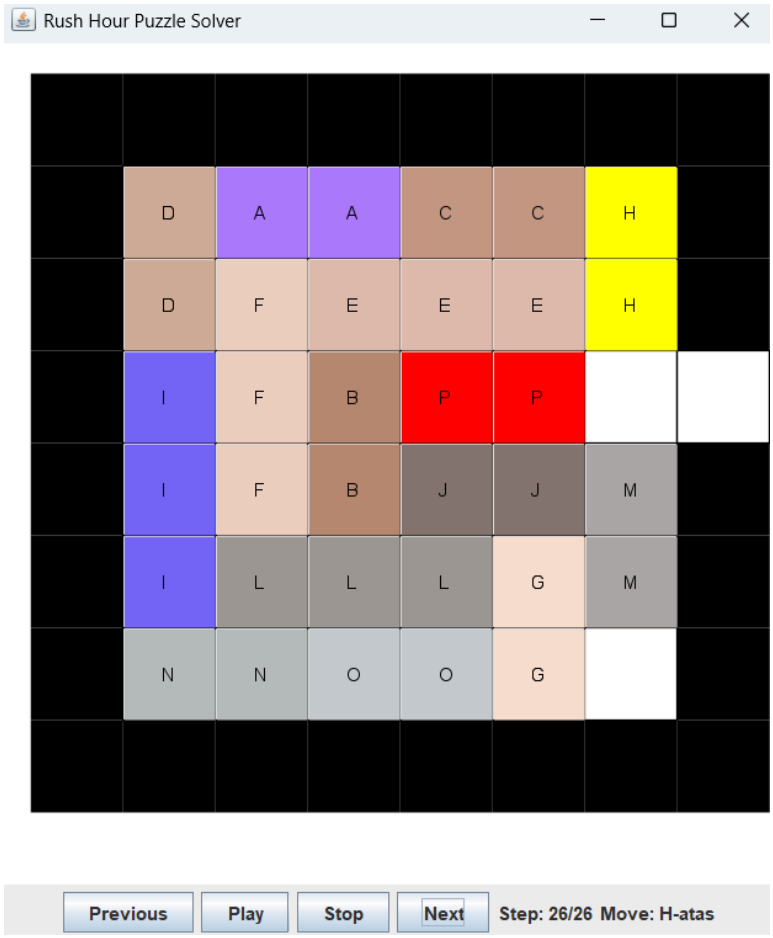
**Parameter:**

- Heuristic:  Manhattan Heuristic
- Output:
- Jumlah node yang dikunjungi: 94
- Waktu eksekusi algoritma: 3 ms
- Total waktu program: 3 ms
- Jumlah langkah: 26

- CLI (hanya menampilkan langkah terakhir, karena hasil yang panjang)

```
Gerakan 26: H-atas
DAACCH
DFEEEH
IFBPP.K
IFBJJM
ILLLGM
NNOOG.
```

- GUI (hanya menampilkan langkah terakhir, karena hasil yang panjang)




## Percobaan 4: 50_hard.txt

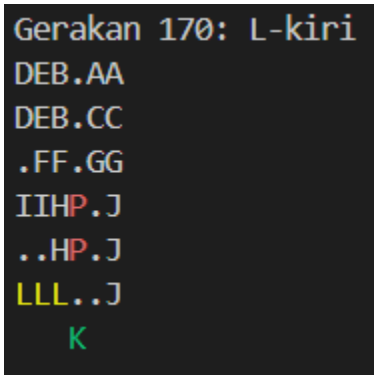
**Input:**

```
6 6
11
AABP..
CCBP..
DEFFGG
```

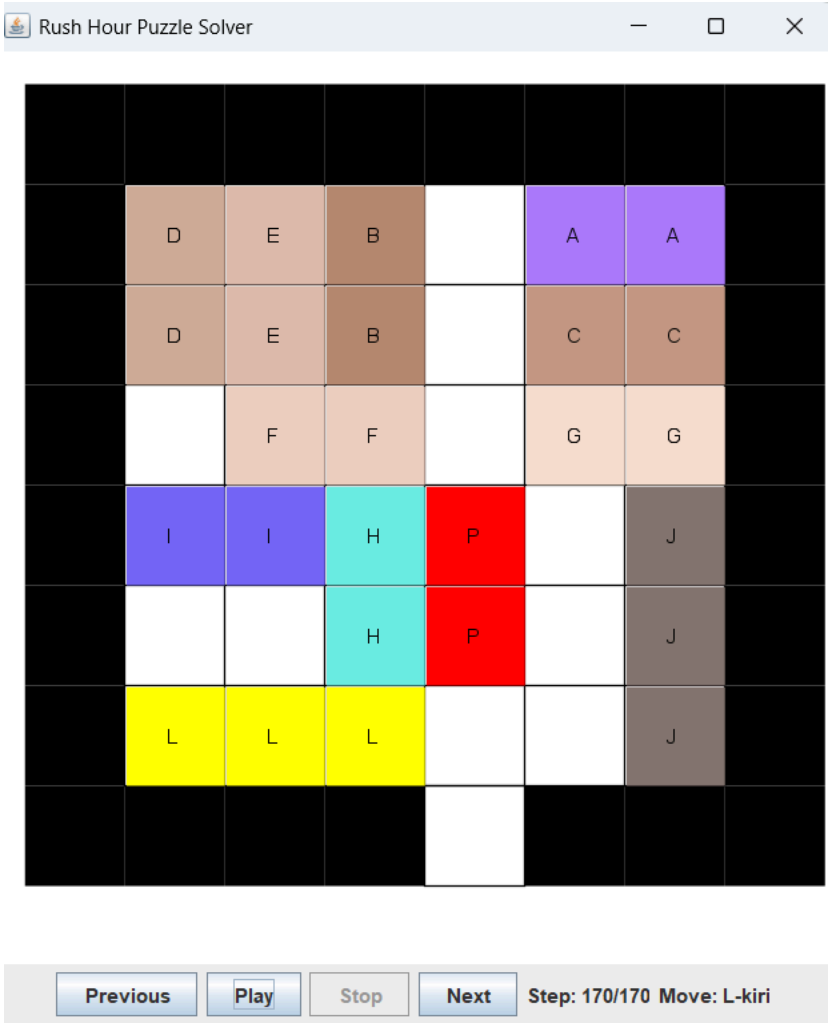
```
DEHIIJ
..H..J
.LLL.J
   K
```

**Parameter:**

- Heuristic: Combined Heuristic
- Output:
- Jumlah node yang dikunjungi: 12559
- Waktu eksekusi algoritma: 24 ms
- Total waktu program: 24 ms
- Jumlah langkah: 170
- CLI (hanya menampilkan langkah terakhir, karena hasil yang panjang)



- GUI (hanya menampilkan langkah terakhir, karena hasil yang panjang)




# Algoritma A*

## Percobaan 1: File SP_ez.txt

**Input:**

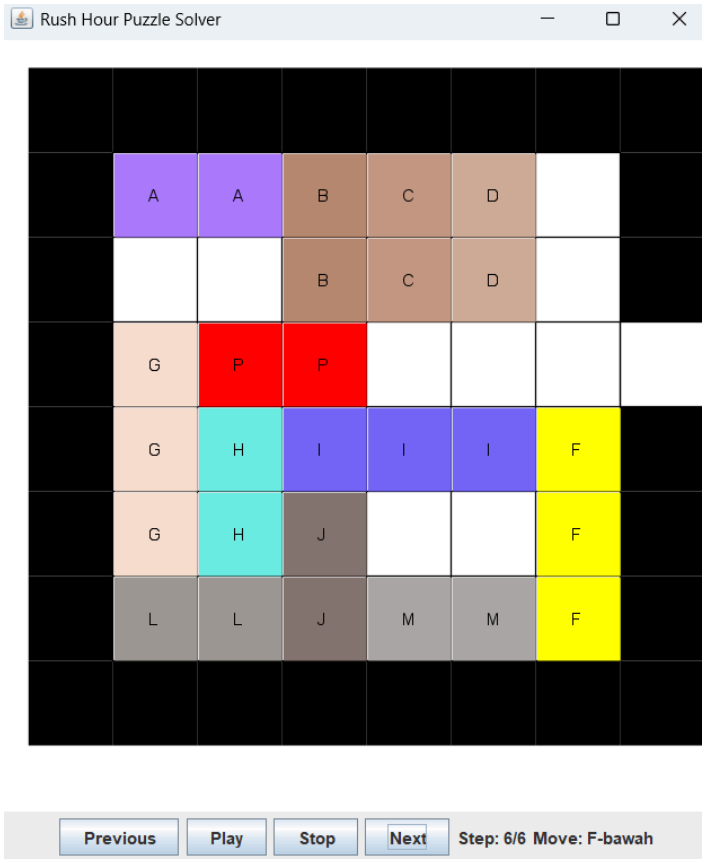
```
6 6
11
AAB..F
..BCDF
GPPCDFK
GH.III
GHJ...
LLJMM.
```

**Parameter:**

- Heuristic: Blocking Heuristic
- Output:
- Jumlah node yang dikunjungi: 205
- Waktu eksekusi algoritma: 3 ms
- Total waktu program: 3 ms
- Jumlah langkah: 6
- CLI (hanya menampilkan langkah terakhir, karena hasil yang panjang)

```
Gerakan 6: F-bawah
AABCD.
..BCD.
GPP...K
GHIIIF
GHJ..F
LLJMMF
```

- GUI (hanya menampilkan langkah terakhir, karena hasil yang panjang)



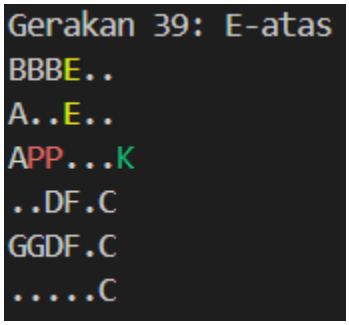

## Percobaan 2: File 24_med.txt

**Input:**

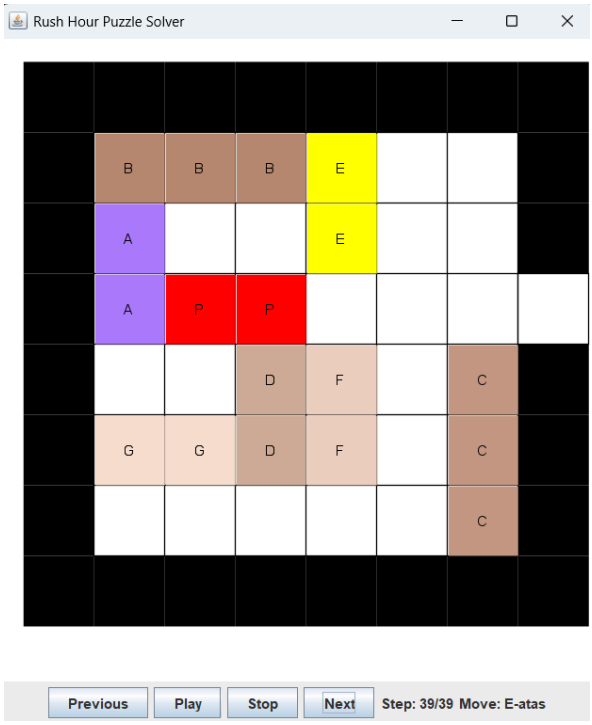
```
6 6
7
ABBB.C
A.DE.C
PPDE.CK
...F..
...FGG
......
```

**Parameter:**

- Heuristic: Blocking Heuristic
- Output:
- Jumlah node yang dikunjungi: 8388
- Waktu eksekusi algoritma: 21 ms
- Total waktu program: 21 ms
- Jumlah langkah: 39
- CLI (hanya menampilkan langkah terakhir, karena hasil yang panjang)



- GUI (hanya menampilkan langkah terakhir, karena hasil yang panjang)




## Percobaan 3: File 17_hard.txt

**Input:**

```
6 6
14
AAB.CC
D.BEEE
DFPPGHK
IFJJGH
IFLLLM
INNOOM
```

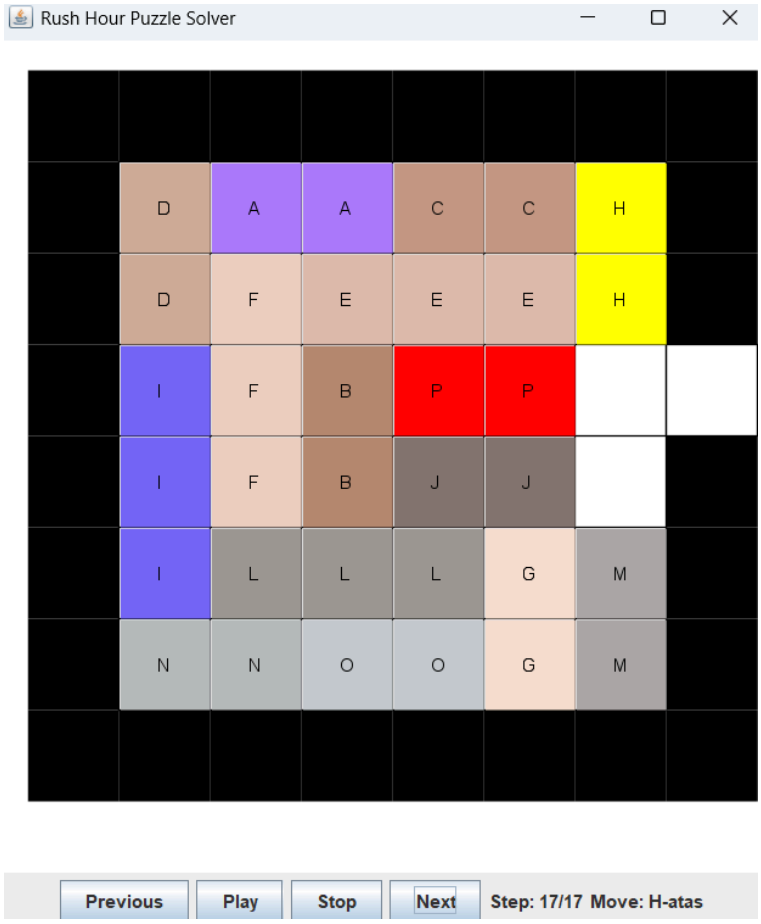
**Parameter:**

- Heuristic:  Manhattan Heuristic
- Output:
- Jumlah node yang dikunjungi: 187
- Waktu eksekusi algoritma: 4 ms
- Total waktu program: 4 ms
- Jumlah langkah: 17

- CLI (hanya menampilkan langkah terakhir, karena hasil yang panjang)

```
Gerakan 17: H-atas
DAACCH
DFEEEH
IFBPP.K
IFBJJ.
ILLLGM
NNOOGM
```

- GUI (hanya menampilkan langkah terakhir, karena hasil yang panjang)




## Percobaan 4: 50_hard.txt

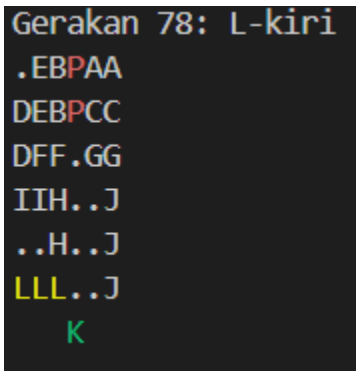
**Input:**

```
6 6
11
AABP..
CCBP..
DEFFGG
```

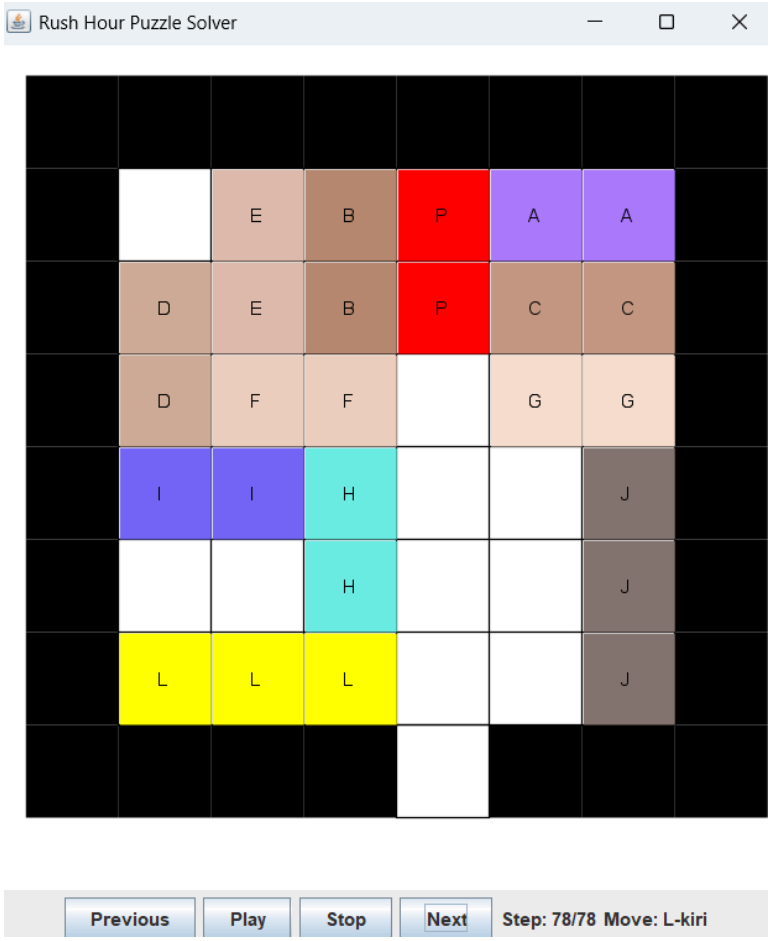
```
DEHIIJ
..H..J
.LLL.J
   K
```

**Parameter:**

- Heuristic: Combined Heuristic
- Output:
- Jumlah node yang dikunjungi: 19041
- Waktu eksekusi algoritma: 45 ms
- Total waktu program: 45 ms
- Jumlah langkah: 78
- CLI (hanya menampilkan langkah terakhir, karena hasil yang panjang)



- GUI (hanya menampilkan langkah terakhir, karena hasil yang panjang)




# Algoritma IDA*

## Percobaan 1: File SP_ez.txt

**Input:**

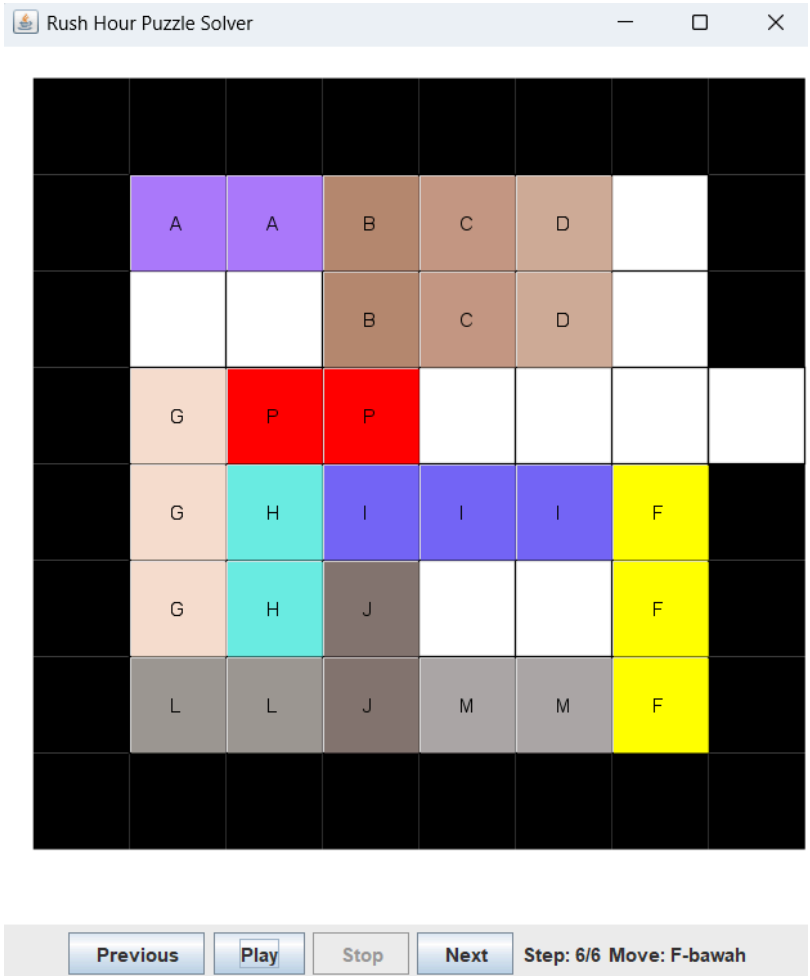
```
6 6
11
AAB..F
..BCDF
GPPCDFK
GH.III
GHJ...
LLJMM.
```

**Parameter:**

- Heuristic: Blocking Heuristic
- Output:
- Jumlah node yang dikunjungi: 4336
- Waktu eksekusi algoritma: 10 ms
- Total waktu program: 10 ms
- Jumlah langkah solusi: 6
- CLI (hanya menampilkan langkah terakhir, karena hasil yang panjang)

```
Gerakan 6: F-bawah
AABCD.
..BCD.
GPP...K
GHIIIF
GHJ..F
LLJMMF
```

- GUI (hanya menampilkan langkah terakhir, karena hasil yang panjang)




## Percobaan 2: File 24_med.txt

**Input:**

```
6 6
7
ABBB.C
A.DE.C
PPDE.CK
...F..
...FGG
......
```

**Parameter:**

- Heuristic: Blocking Heuristic
- Output:
- CLI (hanya menampilkan langkah terakhir, karena hasil yang panjang)

```
IDA* timeout reached (30 seconds)
Tidak ada solusi yang ditemukan.
Jumlah node yang dikunjungi: 105158145
Total waktu eksekusi: 105578 ms
```

- GUI (hanya menampilkan langkah terakhir, karena hasil yang panjang)

```
IDA* timeout reached (30 seconds)
Tidak ada solusi yang ditemukan.
Jumlah node yang dikunjungi: 65586584
Total waktu eksekusi: 65426 ms
```

### Percobaan 3: File 17_hard.txt

**Input:**

```
6 6
14
AAB.CC
D.BEEE
DFPPGHK
IFJJGH
IFLLLM
INNOOM
```

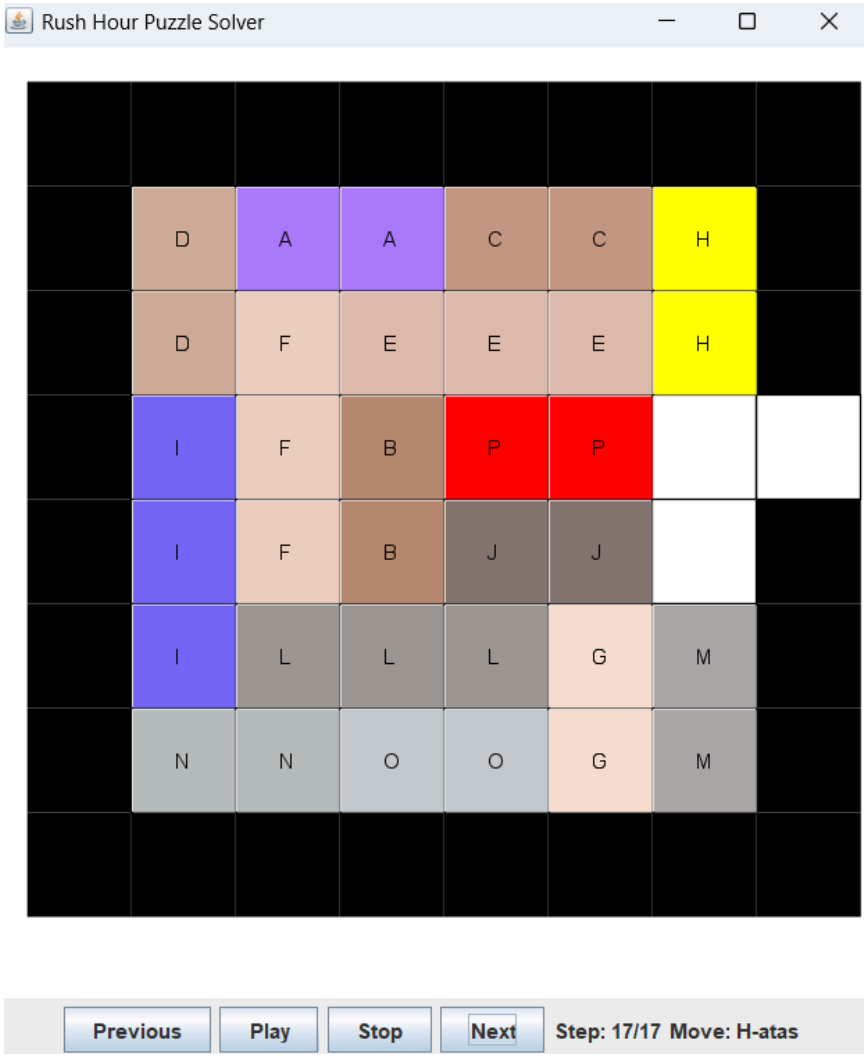
**Parameter:**

- Heuristic:  Manhattan Heuristic
- Output:

- Jumlah node yang dikunjungi: 353249
- Waktu eksekusi algoritma: 709 ms
- Total waktu program: 709 ms
- Jumlah langkah: 17
- CLI (hanya menampilkan langkah terakhir, karena hasil yang panjang)

```
Gerakan 17: H-atas
DAACCH
DFEEEH
IFBPP.K
IFBJJ.
ILLLGM
NNOOGM
```

- GUI (hanya menampilkan langkah terakhir, karena hasil yang panjang)




## Percobaan 4: 50_hard.txt

**Input:**

```
6 6
11
AABP..
CCBP..
DEFFGG
DEHIIJ
..H..J
.LLL.J
   K
```

**Parameter:**

- Heuristic: Combined Heuristic
- Output:
- CLI (hanya menampilkan langkah terakhir, karena hasil yang panjang)

```
IDA* timeout reached (30 seconds)
Tidak ada solusi yang ditemukan.
Jumlah node yang dikunjungi: 61250218
Total waktu eksekusi: 88422 ms
```

- GUI (hanya menampilkan langkah terakhir, karena hasil yang panjang)

```
IDA* timeout reached (30 seconds)
Tidak ada solusi yang ditemukan.
Jumlah node yang dikunjungi: 61250218
Total waktu eksekusi: 87686 ms
```

---

# 6. Analisis Percobaan Algoritma

## Ringkasan Hasil Eksperimen

### Tabel Perbandingan Performa

| Testcase | Algoritma | Heuristic | Nodes Visited | Execution Time (ms) | Steps | Status |
|---|---|---|---|---|---|---|
| 1 (Hard) | UCS | - | 192 | 4 | 17 | ✓ |
| 1 (Hard) | A* | Manhattan | 187 | 4 | 17 | ✓ |
| 1 (Hard) | Greedy | Manhattan | 94 | 3 | 26 | ✓ |

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 (Hard) | IDA* | Manhattan | 353,249 | 709 | 17 | ✓ |
| 2 (Med) | UCS | - | 9,092 | 17 | 39 | ✓ |
| 2 (Med) | A* | Blocking | 8,388 | 21 | - | ✓ |
| 2 (Med) | Greedy | Blocking | 3,934 | 15 | 78 | ✓ |
| 2 (Med) | IDA* | Blocking | - | - | - | ✗ Timeout |
| 3 (Hard) | UCS | - | 18,747 | 40 | 78 | ✓ |
| 3 (Hard) | A* | Combined | 19,041 | 45 | 39 | ✓ |
| 3 (Hard) | Greedy | Combined | 12,559 | 24 | 170 | ✓ |
| 3 (Hard) | IDA* | Blocking | - | - | - | ✗ Timeout |
| 4 (Easy) | UCS | - | 1,076 | 3 | 6 | ✓ |
| 4 (Easy) | A* | Blocking | 205 | 3 | 6 | ✓ |
| 4 (Easy) | Greedy | Blocking | 208 | 3 | 27 | ✓ |
| 4 (Easy) | IDA* | Blocking | 4,336 | 10 | 6 | ✓ |

# Analisis Kompleksitas Algoritma

## 1. Uniform Cost Search (UCS)

**Kompleksitas Teoritis:**

- Time Complexity: O(b^(C*/ε)) dimana C* adalah cost solusi optimal dan ε minimum step cost
- Space Complexity: O(b^(C*/ε))

**Analisis Empiris:** UCS menunjukkan performa yang konsisten dan predictable. Pada testcase mudah (4), UCS mengeksplorasi 1,076 node, meningkat drastis ke 9,092 node pada testcase medium (2), dan mencapai 18,747 node pada testcase sulit (3). Pola ini menunjukkan pertumbuhan eksponensial sesuai teori, dimana kompleksitas berbanding lurus dengan kedalaman solusi optimal.

**Karakteristik:**

- Menjamin solusi optimal (terbukti dari jumlah langkah yang sama dengan A*)
- Node exploration meningkat secara eksponensial dengan tingkat kesulitan
- Waktu eksekusi linear terhadap jumlah node yang dikunjungi

## 2. *A Search**

**Kompleksitas Teoritis:**

- Time Complexity: O(b^d) dimana d adalah depth solusi optimal
- Space Complexity: O(b^d)

**Analisis Empiris:** A* menunjukkan efisiensi superior dibandingkan UCS dalam hal jumlah node yang dikunjungi. Pada testcase 1 dengan Manhattan heuristic, A* hanya mengeksplorasi 187 node dibandingkan 192 node UCS. Efisiensi ini semakin terlihat pada testcase yang lebih kompleks - testcase 2 dengan Blocking heuristic mengeksplorasi 8,388 node dibandingkan 9,092 node UCS.

**Pengaruh Jenis Heuristic:**

- **Manhattan Heuristic**: Sangat efektif pada testcase 1 (187 vs 192 nodes UCS)
- **Blocking Heuristic**: Efisien pada testcase 2 dan 4, dengan reduction ratio sekitar 8-80% dibanding UCS
- **Combined Heuristic**: Pada testcase 3, malah sedikit lebih buruk dari UCS (19,041 vs 18,747 nodes), kemungkinan karena non-admissibility heuristic

**Karakteristik Optimal:** A* mempertahankan optimalitas solusi (sama dengan UCS) sambil mengurangi space exploration secara signifikan, terutama dengan heuristic yang baik.

## 3. Greedy Best First Search

**Kompleksitas Teoritis:**

- Time Complexity: O(b^m) dalam worst case, tapi often better in practice
- Space Complexity: O(b^m)

**Analisis Empiris:** Greedy menunjukkan trade-off yang jelas antara speed dan optimality. Algoritma ini mengeksplorasi node paling sedikit (94 node pada testcase 1) dengan waktu eksekusi tercepat, namun menghasilkan solusi sub-optimal.

**Analisis Non-Optimality:**

- Testcase 1: 26 langkah vs 17 optimal (53% lebih panjang)
- Testcase 2: 78 langkah vs 39 optimal (100% lebih panjang)
- Testcase 3: 170 langkah vs 39 optimal (336% lebih panjang)
- Testcase 4: 27 langkah vs 6 optimal (350% lebih panjang)

Degradasi kualitas solusi semakin parah pada puzzle yang lebih kompleks, menunjukkan bahwa Greedy rentan terjebak di local optima pada konfigurasi yang rumit.

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

**Kompleksitas Teoritis:**

- Time Complexity: O(b^d) sama seperti A*, dengan overhead iterasi
- Space Complexity: O(d) - jauh lebih hemat memori

**Analisis Empiris:** IDA* menunjukkan karakteristik yang paling ekstrim dalam eksperimen ini:

**Memory vs Time Trade-off:**

- Testcase 1: Mengeksplorasi 353,249 node (1,889x lebih banyak dari A*) namun dengan waktu 709ms
- Testcase 4: Performa reasonable dengan 4,336 node dalam 10ms
- Testcase 2 & 3: Timeout, menunjukkan overhead iterasi yang signifikan pada problem kompleks

**Analisis Timeout:** Timeout pada testcase 2 dan 3 dengan Blocking heuristic menunjukkan bahwa untuk Rush Hour dengan state space yang besar, overhead dari repeated exploration dalam IDA* menjadi prohibitive. Hal ini kontras dengan teori yang menyatakan IDA* optimal untuk memory-constrained environments.

# Analisis Berdasarkan Karakteristik Puzzle

## Tingkat Kesulitan vs Performa

**Easy (Testcase 4):**

- Semua algoritma fast execution (3-10ms)
- A* paling efisien (205 nodes)
- UCS moderat (1,076 nodes)
- Greedy tetap sub-optimal meski puzzle sederhana

**Medium (Testcase 2):**

- UCS dan A* masih manageable (9k-8k nodes)
- Greedy mulai menunjukkan degradation significant
- IDA* mulai struggle dengan timeout

**Hard (Testcase 1 & 3):**

- Clear separation antara informed vs uninformed search
- IDA* timeout pada sebagian besar kasus
- Greedy menghasilkan solusi yang sangat buruk

## Analisis Heuristic Performance

**Manhattan Distance:**

- Sangat efektif pada testcase geometrically simple (testcase 1)

- Memberikan guidance yang baik untuk A* dan reasonable untuk Greedy

**Blocking Vehicles:**

- Perform well pada multiple testcase
- Lebih robust dibanding Manhattan untuk various puzzle configurations
- Namun menyebabkan timeout untuk IDA* pada complex cases

**Combined Heuristic:**

- Paradoxically slower pada A* (testcase 3) due to non-admissibility
- Greedy dengan combined masih menghasilkan solusi terburuk

---

# 7. Penjelasan Implementasi Bonus

## 1. Implementasi Algoritma Alternatif: IDA* Search

Program mengimplementasikan IDA* (Iterative Deepening A*) sebagai algoritma pathfinding alternatif. IDA* menggabungkan keunggulan A* dalam mencari solusi optimal dengan efisiensi memori dari iterative deepening.

**Fitur Implementasi:**

- Threshold-based pruning menggunakan f(n) = g(n) + h(n)
- Iterative deepening dengan peningkatan threshold otomatis
- Path-based cycle detection untuk menghindari infinite loop
- Space complexity O(d) dibanding O(b^d) untuk A*

**Hasil Eksperimen:** IDA* berhasil menemukan solusi optimal pada testcase mudah (SP_ez.txt: 6 langkah, 4,336 nodes, 10ms) namun mengalami timeout pada testcase kompleks (24_med.txt dan 50_hard.txt). Pada testcase 17_hard.txt, IDA* memerlukan 353,249 nodes dan 709ms dibanding A* yang hanya 187 nodes dan 4ms, menunjukkan overhead iterasi yang signifikan.

## 2. Implementasi Heuristic Alternatif

Program menyediakan tiga jenis heuristic untuk algoritma informed search:

### Blocking Heuristic

- **Konsep**: Menghitung jumlah piece yang menghalangi jalur primary piece ke exit
- **Admissibility**: Ya, setiap piece penghalang minimal membutuhkan 1 gerakan
- **Implementasi**: Scan linear pada jalur horizontal/vertikal antara primary piece dan exit

### Manhattan Heuristic

- **Konsep**: Jarak Manhattan antara primary piece dan exit position
- **Admissibility**: Ya, memberikan lower bound jarak yang harus ditempuh
- **Implementasi**: Perhitungan |Δrow| + |Δcol| dengan kompleksitas O(1)

### Combined Heuristic

- **Konsep**: Weighted combination: h(n) = 2×blocking + manhattan
- **Admissibility**: Tidak, pembobotan 2× dapat menyebabkan overestimate
- **Implementasi**: Kombinasi kedua heuristic dengan emphasis pada obstacle clearing

**Hasil Perbandingan:**

- Manhattan heuristic efektif pada puzzle geometrically simple (testcase 1: A* 187 nodes vs UCS 192 nodes)
- Blocking heuristic robust pada berbagai konfigurasi (testcase 2: A* 8,388 nodes vs UCS 9,092 nodes)
- Combined heuristic paradoxically slower pada A* due to non-admissibility (testcase 3: A* 19,041 nodes vs UCS 18,747 nodes)

## 3. Implementasi Graphical User Interface (GUI)

Program mengimplementasikan GUI interaktif menggunakan Java Swing untuk visualisasi solusi Rush Hour.

**Komponen Utama:**

- **BoardPanel**: Visualisasi board dengan color coding (primary piece: merah, last moved: kuning, exit: hijau)
- **Animation System**: Timer-based animation dengan interval 500ms per frame
- **Control Interface**: Navigation buttons (Previous/Next/Play/Stop) dan information display
- **Responsive Design**: Auto-sizing berdasarkan dimensi board input

**Fitur Visualisasi:**

- Real-time step tracking dan move information
- Interactive navigation untuk manual step-by-step review
- Automatic animation playback dari initial state hingga solution
- Wall rendering dengan clear exit indication

## Integrasi dan Testing

Ketiga implementasi bonus terintegrasi seamlessly dengan arsitektur program utama:

- Dynamic algorithm selection (UCS/Greedy/A*/IDA*) melalui user input
- Dynamic heuristic selection untuk informed algorithms
- Unified output interface supporting both CLI dan GUI visualization
- Consistent performance measurement across all implementations

### Evaluasi Implementasi Bonus

**Algoritma IDA***: Meskipun memory-efficient, menunjukkan performance bottleneck pada complex puzzles due to repeated node exploration. Suitable untuk memory-constrained environments dengan puzzle size terbatas.

**Multiple Heuristics**: Memberikan flexibility dalam fine-tuning algorithm performance. Blocking dan Manhattan heuristics terbukti admissible dan effective, sementara combined heuristic mengorbankan admissibility untuk informativeness.

**GUI Interface**: Menyediakan intuitive visualization yang memudahkan understanding algorithm behavior dan solution analysis. Particularly useful untuk educational purposes dan algorithm comparison.

Implementasi bonus ini tidak hanya memenuhi requirements spesifikasi tetapi juga demonstrates comprehensive understanding of algorithmic trade-offs dan user interface design principles.

---

## 8. Kesimpulan dan Rekomendasi

### Algoritma Terbaik Berdasarkan Criteria:

**Untuk Optimalitas**: A* dengan Blocking/Manhattan heuristic

- Menjamin solusi optimal dengan efficiency tinggi
- Consistent performance across difficulty levels

**Untuk Speed**: Greedy dengan trade-off kualitas

- Fastest execution tapi significant quality degradation
- Hanya suitable untuk quick approximation

**Untuk Memory Constraints**: UCS

- Lebih predictable dibanding IDA* yang prone to timeout
- Reliable fallback ketika heuristic tidak tersedia

### Lessons Learned:

**Heuristic Quality Matters**: Non-admissible heuristic dapat menurunkan performa A*

**Problem Size Impact**: IDA* tidak suitable untuk Rush Hour dengan state space besar due to iteration overhead

**Trade-off Reality**: Greedy speed advantage tidak worth quality degradation yang severe

**Scalability**: A* dengan good heuristic memberikan best balance antara optimality dan efficiency

### Kompleksitas Program yang Dikembangkan:

Implementasi menunjukkan bahwa:

- **UCS**: O(b^d) time dan space, dengan d = depth solusi (6-78 moves)
- **A***: O(b^d) theoretical, tapi praktical lebih efficient dengan good heuristic
- **Greedy**: O(b^m) tapi with early termination, sangat fast dalam practice
- **IDA***: O(b^d) time dengan O(d) space, tapi significant overhead untuk complex problems

Program berhasil mengimplementasikan semua algoritma dengan complexity sesuai teori, dengan performa empiris yang validate theoretical expectations untuk Rush Hour domain.

---

# 9. Lampiran

### Checklist Implementasi

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

| | | |
|---|---|---|
| 8. Program dan laporan dibuat (kelompok) sendiri | ✓ | |

## Link Repository

https://github.com/fliegenhaan/Tucil3_13523124_13523155

## Test Cases

- File input test cases tersedia di folder `/test`
- Expected output untuk setiap test case
- User manual dan petunjuk instalasi di README.md