

Tugas Kecil 3 IF2211 Strategi Algoritma

Semester II Tahun 2024/2025

Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding



Disusun oleh :

Ahmad Wafi Idzharulhaqq (13523131)

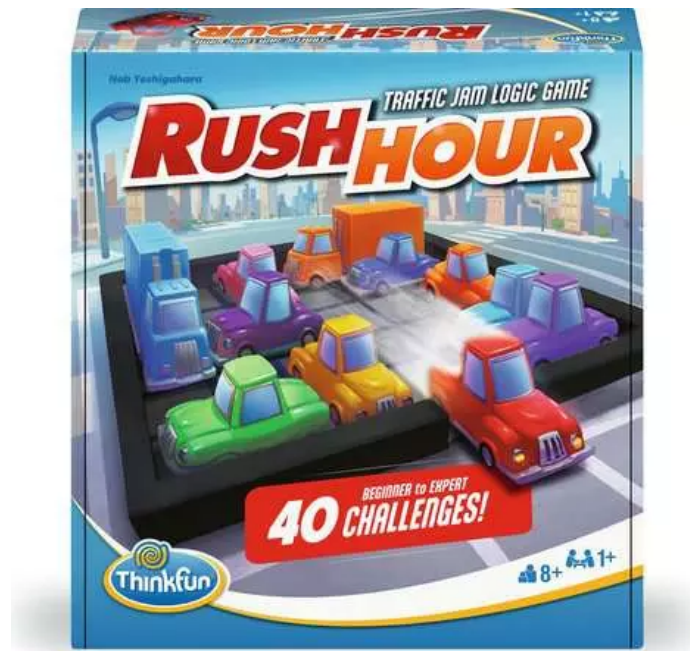
Lukas Raja Agripa (13523158)

Institut Teknologi Bandung

2025

Bab 1

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.

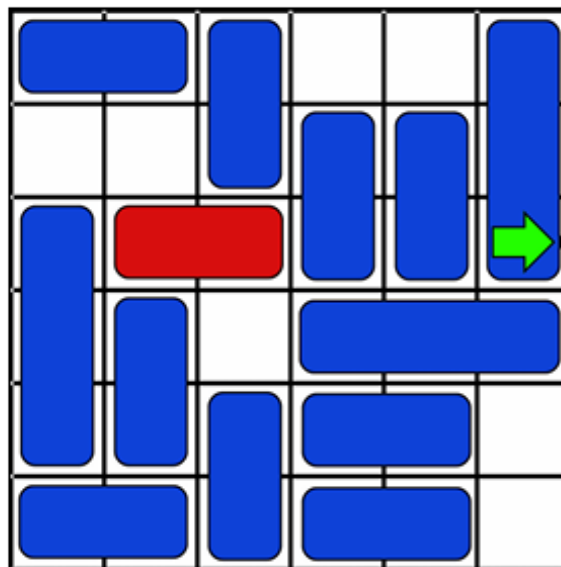
3. Primary Piece – *Primary piece* adalah kendaraan utama yang harus dikeluarkan dari papan (biasanya berwarna merah). Hanya boleh terdapat satu *primary piece*.

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.

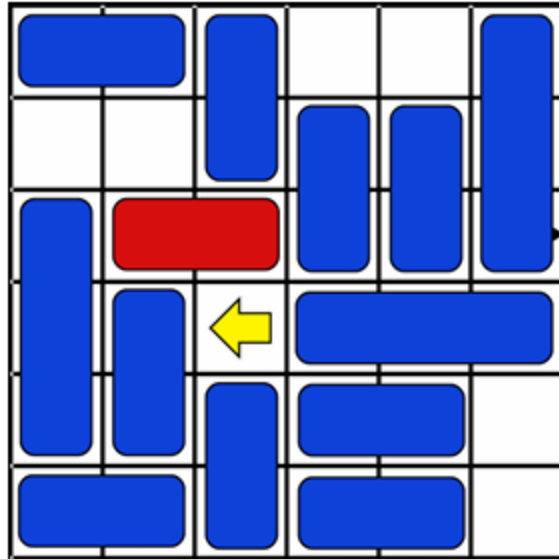
Ilustrasi kasus :

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.

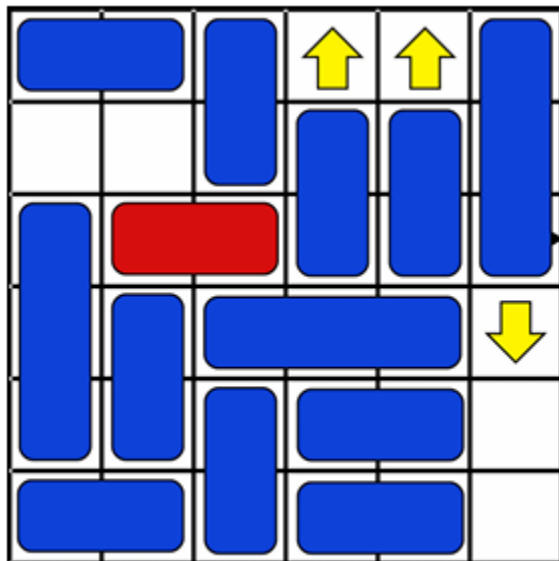


Gambar 2. Awal Permainan Game Rush Hour

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

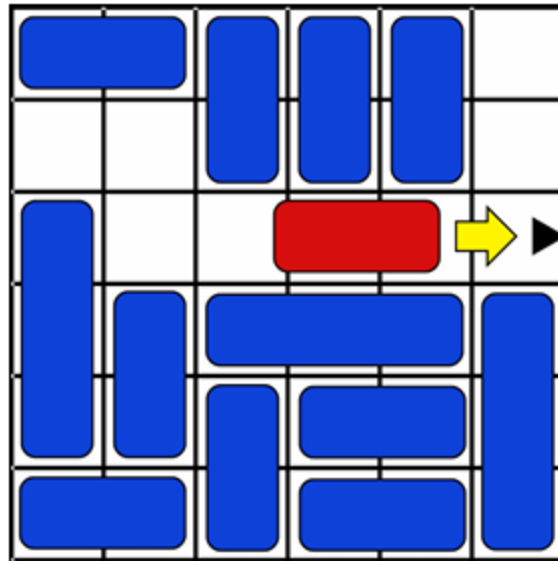


Gambar 3. Gerakan Pertama Game Rush Hour



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](#)

Bab 2

Landasan Teori

2.1 Greedy Best First Search

Greedy Best-First Search adalah algoritma pencarian yang memanfaatkan pendekatan heuristik untuk memperkirakan jarak suatu state ke tujuan, lalu memilih node dengan estimasi jarak paling kecil untuk diperluas terlebih dahulu. Berbeda dari UCS yang mempertimbangkan biaya riil dari awal, Greedy BFS hanya berfokus pada nilai heuristik $h(n)$ dan mengabaikan biaya yang telah dikeluarkan sejauh ini. Hal ini membuat Greedy BFS cenderung lebih cepat dalam menemukan solusi, meskipun solusi tersebut belum tentu optimal. Dalam penerapannya, pemilihan heuristik yang baik sangat memengaruhi kinerja algoritma. Sebagai contoh, dalam permainan Rush Hour, heuristik yang umum digunakan adalah jumlah kendaraan yang menghalangi jalur mobil utama ke pintu keluar.

2.2 Uniform First Search

Uniform Cost Search (UCS) merupakan algoritma pencarian tak terinformatika yang bekerja dengan cara memperluas node berdasarkan biaya kumulatif paling rendah dari titik awal. Russell & Norvig (2020) menjelaskan bahwa dalam konteks pencarian jalur, UCS memprioritaskan jalur dengan total cost terkecil menggunakan struktur data seperti antrian prioritas. Algoritma ini akan menjamin solusi optimal selama setiap langkah memiliki biaya positif. UCS tidak mempertimbangkan seberapa dekat suatu state dengan tujuan, melainkan hanya fokus pada akumulasi biaya perjalanan dari awal hingga titik tertentu. Pendekatan ini sangat cocok untuk masalah di mana setiap tindakan memiliki bobot yang berbeda. Namun, kekurangannya adalah waktu eksekusi dan penggunaan memori yang cukup besar jika ruang pencarian sangat luas.

2.3 A Star

A Star (A^*) adalah algoritma pencarian yang menggabungkan prinsip dari Uniform Cost Search dan Greedy Best-First Search. Algoritma ini menggunakan fungsi evaluasi $f(n) = g(n) + h(n)$, di mana $g(n)$ adalah biaya dari titik awal ke node saat ini, dan $h(n)$ adalah estimasi biaya dari node tersebut ke tujuan. Dengan memadukan dua komponen tersebut, A^* mampu menyeimbangkan antara eksplorasi jalur terpendek dan efisiensi pencarian menuju goal. A^* akan selalu menemukan solusi optimal jika heuristik yang digunakan bersifat admissible, yaitu tidak melebih-lebihkan jarak ke tujuan. Dalam konteks permainan seperti Rush Hour, A^* sangat efektif karena mampu mengarahkan pencarian dengan efisien tanpa mengorbankan keoptimalan. Algoritma ini banyak digunakan dalam berbagai aplikasi seperti pemetaan rute dan permainan puzzle karena kemampuannya dalam memberikan hasil terbaik dengan performa yang relatif cepat.

2.4 Simulated Annealing

Simulated Annealing merupakan algoritma pencarian berbasis probabilistik yang terinspirasi dari proses pendinginan logam dalam metalurgi. Algoritma ini dirancang untuk menghindari jebakan local optimum dengan cara sesekali menerima solusi yang lebih buruk secara terkontrol berdasarkan probabilitas yang bergantung pada parameter suhu (temperature) dan seberapa buruk solusi tersebut. Pada awalnya, algoritma memperbolehkan perubahan besar dan penerimaan terhadap solusi yang jauh lebih buruk, namun seiring penurunan suhu, ruang gerak dan toleransi terhadap solusi buruk semakin kecil. Dalam konteks permainan Rush Hour, Simulated Annealing bekerja dengan mengeksplorasi berbagai konfigurasi kendaraan, dan meskipun tidak menjamin solusi optimal, pendekatan ini efektif untuk menemukan solusi layak dalam waktu yang relatif cepat, terutama pada kasus dengan ruang pencarian sangat besar dan kompleks.

2.5 Iterative Deepening Search (IDS)

Iterative Deepening Search (IDS) adalah algoritma pencarian yang menggabungkan kelebihan Depth-First Search (DFS) dalam hal penggunaan memori dengan keoptimalan dan kelengkapan Breadth-First Search (BFS). IDS bekerja dengan cara melakukan DFS berulang-ulang dengan batas kedalaman yang meningkat secara bertahap hingga solusi ditemukan. Meskipun tampak redundant karena melakukan eksplorasi ulang dari akar setiap iterasi, IDS memiliki efisiensi ruang yang sangat baik karena hanya membutuhkan memori linear terhadap kedalaman pohon. Dalam penerapannya pada permainan Rush Hour, IDS mengeksplorasi semua kemungkinan konfigurasi kendaraan hingga kedalaman tertentu, memastikan bahwa solusi ditemukan secara sistematis tanpa mengorbankan memori yang berlebihan. Algoritma ini cocok untuk kasus di mana kedalaman solusi tidak diketahui dan ruang pencarian terlalu luas untuk BFS biasa.

Bab 3

Analisis Pemecahan Masalah

3.1 Analisis Permasalahan

Permasalahan utama dalam permainan *Rush Hour* adalah bagaimana menemukan urutan langkah paling efisien untuk memindahkan mobil merah (*primary piece*) dari posisi awal menuju pintu keluar pada papan permainan yang dipenuhi kendaraan lain. Setiap kendaraan memiliki arah gerak terbatas (hanya bisa maju atau mundur sesuai orientasinya), sehingga ruang gerak sangat sempit dan menghasilkan ruang pencarian yang kompleks. Dibutuhkan algoritma pencarian yang mampu menavigasi ruang solusi secara efisien, baik dari sisi waktu maupun akurasi solusi. Program yang dibangun harus dapat menemukan langkah minimal menuju solusi, atau menyesuaikan dengan heuristik yang dipilih.

3.2 Langkah - Langkah Pemecahan Masalah

- **Muat Data dan Inisialisasi**
Program menerima input file .txt yang berisi ukuran papan, jumlah kendaraan non-primary, dan matriks papan berisi representasi pieces dan ruang kosong.
- **Pembentukan Struktur Internal Data**
Dibentuk objek *Board* yang menyimpan posisi dan ukuran setiap kendaraan serta objek *Piece* yang berisi posisi (koordinat), orientasi, serta panjang setiap piece
- **Proses Pencarian Solusi**
Berdasarkan algoritma pilihan pengguna (UCS, Greedy BFS, atau A*), program melakukan traversal terhadap state-state papan. Hal ini dilakukan dengan ketentuan sebagai berikut:
 - Setiap node adalah konfigurasi papan
 - Gerakan piece menghasilkan state baru
 - Sistem menyimpan visited states untuk menghindari duplikasi langkah.

Dalam prosesnya, diterapkan Heuristik (khusus Greedy BFS dan A*) sesuai dengan masukan dari pengguna.

- **Bentuk dan Tampilkan Output**
Output yang ditampilkan berupa kondisi awal papan, tahapan dalam pencarian solusi, waktu eksekusi, dan banyak node yang ditempuh hingga mencapai solusi.

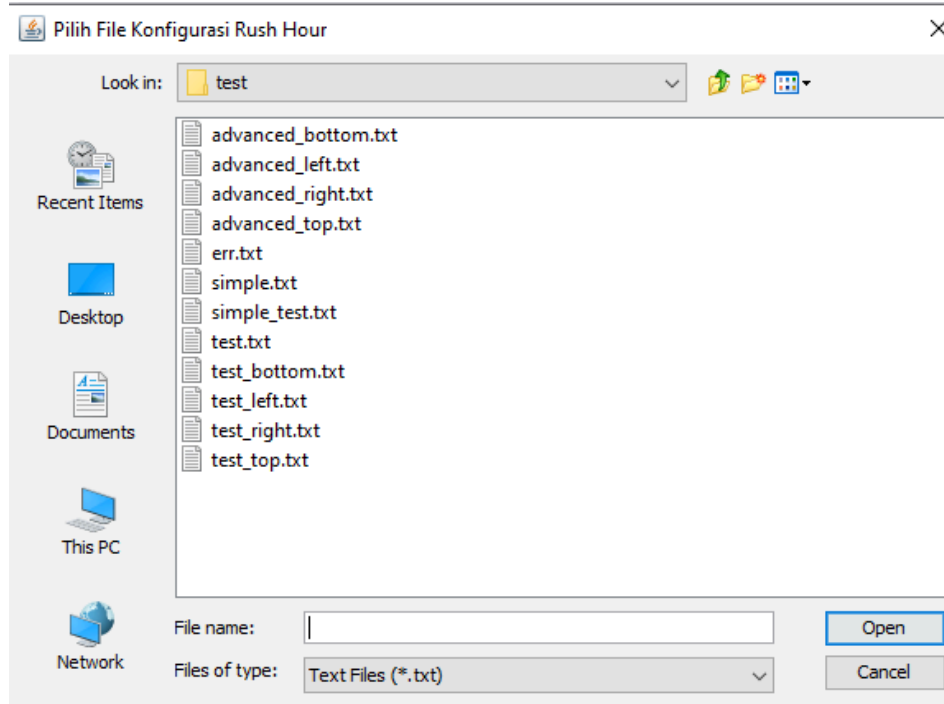
3.3 Proses Pemetaan Masalah

Dalam program ini, Masalah Rush Hour direpresentasikan sebagai graf tak eksplisit dengan karakteristik sebagai berikut:

- **Node (Simpul):** Konfigurasi lengkap posisi semua kendaraan.
- **Edge (Sisi):** Aksi pergeseran kendaraan satu langkah maju/mundur.
- **Start Node:** Konfigurasi awal dari input.
- **Goal Node:** Konfigurasi di mana primary piece berada tepat sebelum pintu keluar dan dapat digerakkan keluar papan.

3.3 Fitur Fungsional yang Dibangun

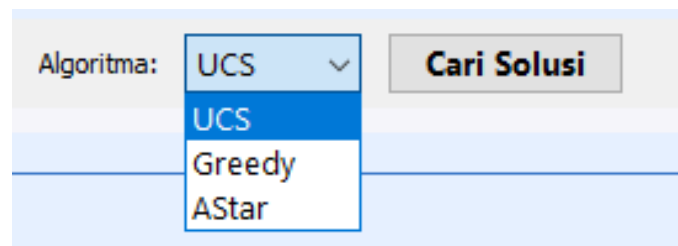
- **Import File .txt**
Pengguna dapat memilih file dengan format .txt dari direktori lokal untuk dilakukan pemrosesan dalam program. Hal ini memungkinkan pengguna untuk mencari solusi dari berbagai kasus dengan mudah.



Gambar 1: Menu Input File

- **Pemilihan Algoritma Pencarian Solusi**

Terdapat 3 algoritma utama yang bisa dipilih oleh pengguna untuk mencari solusi, yaitu Greedy BFS, A*, dan UCS. Pengguna dapat melakukan komparasi hasil dari ketiga algoritma ini secara langsung dengan mengganti pilihan algoritma pada file yang sama tanpa perlu melakukan impor file kembali.



Gambar 2: Menu Pemilihan Algoritma

- **Pemilihan Heuristik**

1. Manhattan Distance

Heuristik Manhattan Distance menghitung jarak antara posisi saat ini dari *primary piece* ke posisi pintu keluar. Jarak ini dihitung secara horizontal atau vertikal, tanpa mempertimbangkan rintangan di antaranya. Heuristik ini bersifat sederhana namun efektif dalam memberikan estimasi seberapa jauh *primary piece* dari tujuan.

2. Manhattan Blocking

Heuristik Manhattan Blocking memberikan penalti terhadap *pieces* yang menghalangi jalur langsung *primary piece* menuju pintu keluar. Dengan menghitung jumlah kendaraan yang menjadi penghalang di jalur tersebut, heuristik ini mendorong algoritma untuk memprioritaskan langkah-langkah yang membuka jalur utama. Pendekatan ini sangat berguna dalam mengarahkan solusi agar tidak memindahkan *piece* secara tidak perlu ke jalur utama.

3. Enhanced Heuristic

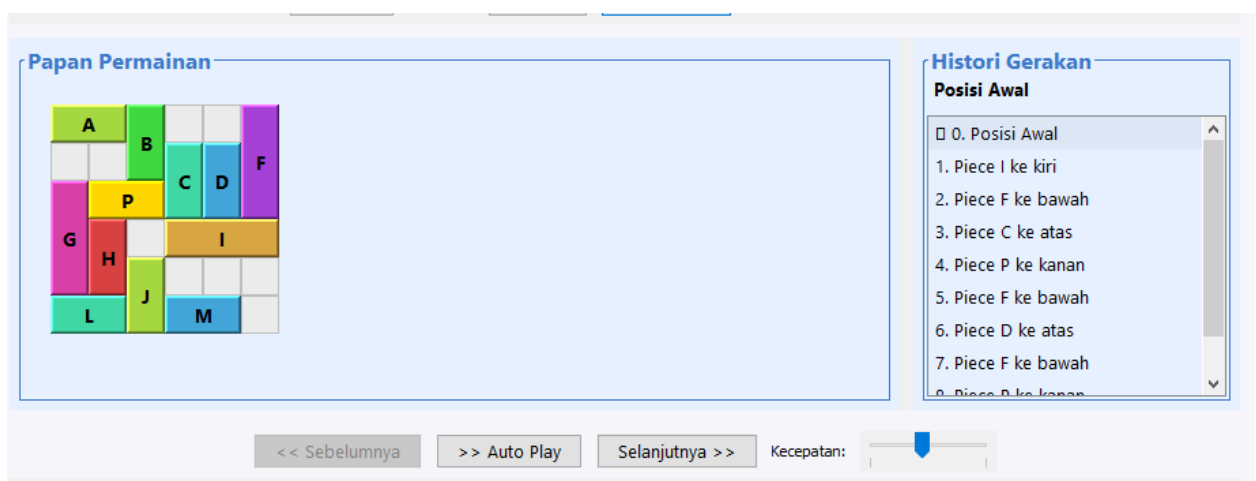
Enhanced Heuristic merupakan gabungan dari Manhattan Distance dan Manhattan Blocking. Heuristik ini memperhitungkan jarak *primary piece* ke tujuan sekaligus penalti dari *pieces* yang menghalangi jalur. Dengan demikian, algoritma tidak hanya diarahkan untuk mendekati tujuan, tetapi juga untuk mengurangi hambatan di sepanjang jalur. Pendekatan ini cenderung menghasilkan pencarian yang lebih efisien dan terarah.

4. Custom SA

Heuristik Custom SA digunakan secara khusus dalam algoritma Simulated Annealing. Heuristik ini mempertimbangkan atribut-atribut dari papan (*board*), posisi dan orientasi setiap *piece*, serta konstanta lain seperti temperatur awal, laju pendinginan, dan probabilitas penerimaan solusi. Heuristik ini dirancang agar mampu mengevaluasi kualitas solusi secara menyeluruh dan fleksibel, menyesuaikan dengan dinamika eksplorasi solusi khas dari Simulated Annealing.

- Tampilan Pencarian Solusi per Tahap

Proses pencarian ditampilkan secara bertahap dari kondisi awal hingga tercapai solusi. Apabila tidak ada solusi yang memungkinkan, akan ditampilkan keterangan bahwa case tersebut tidak memiliki solusi. Hal ini meningkatkan efisiensi waktu pengguna dalam menemukan solusi dari kasus tertentu.



Gambar 3: Tampilan Solusi Pencarian Solusi

- Pengaturan Kecepatan

Terdapat opsi pengaturan kecepatan untuk mengatur kecepatan solusi ditampilkan ke layar

Bab 4

Implementasi dan Pengujian

4.1 Implementasi Algoritma

Program Rush Hour Solver ini dikembangkan menggunakan bahasa pemrograman Java dengan menerapkan paradigma berorientasi objek. Untuk kebutuhan visualisasi dan input interaktif, digunakan komponen GUI menggunakan tools bawaan dari Java.

Algoritma UCS

Metode ini menghitung *cost kumulatif terkecil* ($g(n)$) dari *root state* ke state saat ini. Karena semua gerakan bernilai 1, UCS mencari solusi dengan jumlah langkah minimum.

```
private List<GameBoard> ucs(GameBoard initialBoard) {  
    PriorityQueue<Node> queue = new  
        PriorityQueue<>(Comparator.comparingInt(Node::getCost));  
    Set<String> visited = new HashSet<>();  
    queue.add(new Node(initialBoard, null, 0, 0));  
}
```

Pada tahap awal, dilakukan inisialisasi variabel berupa antrean prioritas berdasarkan cost dari *root state* ($g(n)$) serta set visited untuk menyimpan konfigurasi papan yang sudah dikunjungi. Root state atau node awal ditambahkan ke antrean.

```
while (!queue.isEmpty()) {  
    Node current = queue.poll();  
    nodesVisited++;  
    GameBoard currentBoard = current.getBoard();  
  
    if (currentBoard.isGoal()) {  
        return reconstructPath(current);  
    }  
  
    String boardState = boardToString(currentBoard);  
    if (!visited.contains(boardState)) {
```

```

        visited.add(boardState);
        for (GameBoard neighbor : currentBoard.getNeighbors()) {
            String neighborState = boardToString(neighbor);
            if (!visited.contains(neighborState)) {
                Node newNode = new Node(neighbor, current,
current.getCost() + 1, 0);
                queue.add(newNode);
                nodeMap.put(neighbor, newNode);
            }
        }
    }
    return null;
}

```

Pada setiap iterasi, apabila queue tidak kosong, node dengan cost terkecil diambil (node pada posisi head), lalu dilakukan penambahan nodeVisited dan dilakukan pemeriksaan apakah state saat ini sudah mencapai kondisi goal. Jika belum, seluruh neighbor (konfigurasi yang memungkinkan dari state saat ini) diekspansi dan dimasukkan ke antrian sebagai node baru jika belum pernah dikunjungi.

Semua langkah akan memiliki bobot yang sama, yaitu 1. Sehingga cost terhadap node neighbor yang akan dibentuk ditambah satu dari current cost. Apabila node yang saat ini dicek adalah Goal, dilakukan pengembalian dari pemanggilan fungsi reconstructPath untuk merekonstruksi jalur solusi. Sebaliknya, jika queue kosong dan tidak tercapai goal maka akan dikembalikan null (tidak ada solusi)

Algoritma Greedy Best First Search

Greedy BFS memilih node berdasarkan nilai heuristik $h(n)$ saja, tanpa mempertimbangkan cost dari root ($g(n)$). Ia memilih node yang tampak paling dekat ke solusi saja.

```

private List<GameBoard> greedyBestFirstSearch(GameBoard initialBoard) {
    PriorityQueue<Node> queue = new
PriorityQueue<>(Comparator.comparingInt(Node::getHeuristic));
    Set<String> visited = new HashSet<>();
    queue.add(new Node(initialBoard, null, 0, initialBoard.getHeuristic()));
}

```

Pada tahap awal, dilakukan inisialisasi variabel berupa antrian prioritas berdasarkan pendekatan heuristik yang digunakan ($h(n)$) serta set visited untuk menyimpan konfigurasi papan yang sudah dikunjungi. Root state atau node awal ditambahkan ke antrian.

```

while (!queue.isEmpty()) {
    Node current = queue.poll();
    nodesVisited++;
    GameBoard currentBoard = current.getBoard();

    if (currentBoard.isGoal()) {

```

```

        return reconstructPath(current);
    }

    String boardState = boardToString(currentBoard);
    if (!visited.contains(boardState)) {
        visited.add(boardState);
        for (GameBoard neighbor : currentBoard.getNeighbors()) {
            String neighborState = boardToString(neighbor);
            if (!visited.contains(neighborState)) {
                Node newNode = new Node(neighbor, current, 0,
neighbor.getHeuristic());
                queue.add(newNode);
                nodeMap.put(neighbor, newNode);
            }
        }
    }
    return null;
}

```

Selama antrian tidak kosong, algoritma mengambil konfigurasi dengan nilai heuristik terkecil, memeriksa apakah konfigurasi tersebut merupakan goal. Jika bukan, algoritma menghasilkan semua konfigurasi neighbor (semua konfigurasi yang memungkinkan). Setiap konfigurasi baru yang belum pernah dikunjungi akan dikonversi menjadi node dan dimasukkan ke dalam antrian. Proses ini terus berulang hingga solusi ditemukan atau semua kemungkinan telah dieksplorasi (tidak ada solusi).

Algoritma AStar (A*)

```

private List<GameBoard> greedyBestFirstSearch(GameBoard initialBoard) {
    PriorityQueue<Node> queue = new
    PriorityQueue<>(Comparator.comparingInt(Node::getHeuristic));
    Set<String> visited = new HashSet<>();
    queue.add(new Node(initialBoard, null, 0, initialBoard.getHeuristic()));
}

```

Pada tahap inisialisasi, queue dibuat dengan urutan berdasarkan $f(n) = g(n) + h(n)$ (dalam program disebut `getTotalCost()`). Set visited disiapkan untuk mencatat konfigurasi papan yang sudah pernah dikunjungi. Kemudian, node awal (berisi konfigurasi papan awal) dimasukkan ke dalam antrian dengan nilai $g = 0$ dan h dihitung menggunakan `initialBoard.getHeuristic()`.

```

while (!queue.isEmpty()) {
    Node current = queue.poll();
    nodesVisited++;
    GameBoard currentBoard = current.getBoard();
}

```

```

        if (currentBoard.isGoal()) {
            return reconstructPath(current);
        }

        String boardState = boardToString(currentBoard);
        if (!visited.contains(boardState)) {
            visited.add(boardState);
            for (GameBoard neighbor : currentBoard.getNeighbors()) {
                String neighborState = boardToString(neighbor);
                if (!visited.contains(neighborState)) {
                    Node newNode = new Node(neighbor, current,
current.getCost() + 1, neighbor.getHeuristic());
                    queue.add(newNode);
                    nodeMap.put(neighbor, newNode);
                }
            }
        }
    }
    return null;
}

```

selama antrian belum kosong, algoritma mengambil node dengan nilai total cost terkecil ($f(n) = g(n) + h(n)$) dari antrian. Setiap node yang diambil akan dicek apakah sudah merupakan goal state. Jika sudah, maka jalur menuju solusi akan dibentuk melalui `reconstructPath(current)` dan dikembalikan. Jika belum, algoritma akan mencatat konfigurasi papan saat ini sebagai sudah dikunjungi, lalu menghasilkan semua konfigurasi neighbor menggunakan `getNeighbors()`. Untuk setiap neighbor yang belum pernah dikunjungi, dibuat node baru dengan $g = \text{current.g} + 1$ dan $h = \text{heuristic}$ dari konfigurasi baru, kemudian node tersebut dimasukkan ke antrian. Proses ini berulang hingga goal ditemukan atau antrian kosong.

Simulated Annealing

```

private List<GameBoard> simulatedAnnealing(GameBoard initialBoard) {
    // Parameter SA
    double initialTemperature = 1000.0;
    double finalTemperature = 0.1;
    double coolingRate = 0.95;
    int maxIterations = 10000;
    int maxStagnation = 1000; // Maksimum iterasi tanpa improvement

    GameBoard currentBoard = initialBoard;
    GameBoard bestBoard = initialBoard;
    int currentCost = calculateCostForSA(currentBoard);
    int bestCost = currentCost;

    // Untuk menyimpan path terbaik yang ditemukan
    Map<GameBoard, GameBoard> parentMap = new HashMap<>();
    Map<GameBoard, Integer> costMap = new HashMap<>();

    parentMap.put(initialBoard, null);
}

```

```

costMap.put(initialBoard, 0);

double temperature = initialTemperature;
int stagnationCounter = 0;
int iterationsWithoutImprovement = 0;

```

Bagian awal fungsi simulatedAnnealing ini bertujuan untuk melakukan inisialisasi parameter dan state awal algoritma. Parameter seperti suhu awal (initialTemperature) dan akhir (finalTemperature), laju pendinginan (coolingRate), batas maksimum iterasi (maxIterations), serta batas stagnasi (maxStagnation) ditentukan untuk mengatur dinamika proses annealing. Selanjutnya, papan permainan awal (initialBoard) dijadikan sebagai state awal (currentBoard) sekaligus solusi terbaik sementara (bestBoard), dan biaya awal dihitung menggunakan fungsi calculateCostForSA. Untuk keperluan pelacakan jalur solusi, digunakan struktur parentMap dan costMap yang masing-masing menyimpan hubungan antar state dan biaya kumulatifnya. Terakhir, suhu saat ini, penghitung stagnasi, dan iterasi tanpa perbaikan diinisialisasi sebagai dasar kontrol terhadap alur iteratif algoritma.

```

for (int iteration = 0; iteration < maxIterations && temperature >
finalTemperature; iteration++) {
    nodesVisited++;

    // Cek apakah sudah mencapai goal
    if (currentBoard.isGoal()) {
        return reconstructPathFromMap(currentBoard, parentMap);
    }

    // Dapatkan neighbor secara random
    ArrayList<GameBoard> neighbors = currentBoard.getNeighbors();
    if (neighbors.isEmpty()) {
        break; // Tidak ada gerakan yang mungkin
    }

    GameBoard nextBoard =
neighbors.get(random.nextInt(neighbors.size()));
    int nextCost = calculateCostForSA(nextBoard);

    // Hitung delta cost (perbedaan cost)
    int deltaCost = nextCost - currentCost;

    // SA decision: terima state baru jika lebih baik, atau dengan
probabilitas tertentu jika lebih buruk
    boolean acceptMove = false;
    if (deltaCost < 0) {
        // State lebih baik, pasti diterima
        acceptMove = true;
        iterationsWithoutImprovement = 0;
    } else {
        // State lebih buruk, terima dengan probabilitas  $\exp(-\delta/T)$ 
        double probability = Math.exp(-deltaCost / temperature);

```

```

        if (random.nextDouble() < probability) {
            acceptMove = true;
        }
        iterationsWithoutImprovement++;
    }

    if (acceptMove) {
        // Update parent map untuk path reconstruction
        parentMap.put(nextBoard, currentBoard);
        costMap.put(nextBoard, costMap.getOrDefault(currentBoard, 0) +
1);

        currentBoard = nextBoard;
        currentCost = nextCost;
        stagnationCounter = 0;
        acceptedMoves++;

        // Update best solution jika ditemukan yang lebih baik
        if (currentCost < bestCost) {
            bestBoard = currentBoard;
            bestCost = currentCost;
            iterationsWithoutImprovement = 0;
        }
        else {
            stagnationCounter++;
            rejectedMoves++;
        }

        // Cooling schedule: kurangi temperature
        temperature *= coolingRate;

        // Early termination jika terlalu lama tidak ada improvement
        if (stagnationCounter > maxStagnation || iterationsWithoutImprovement
> maxStagnation) {
            break;
        }

        // Restart jika stuck di local optimum untuk waktu yang lama
        if (iterationsWithoutImprovement > maxStagnation / 2) {
            // Random restart dengan temperature yang lebih tinggi
            temperature = Math.max(temperature, initialTemperature * 0.1);
            currentBoard = getRandomValidState(initialBoard);
            currentCost = calculateCostForSA(currentBoard);
            iterationsWithoutImprovement = 0;
        }
    }
}

```

Pencarian solusi dilakukan selama suhu masih di atas batas minimum dan iterasi belum mencapai maksimum. Pada setiap iterasi, algoritma memeriksa apakah state saat ini merupakan solusi akhir; jika ya, maka jalur solusi langsung direkonstruksi dan dikembalikan. Jika belum, algoritma mengambil satu tetangga secara acak dan menghitung cost-nya, lalu menentukan apakah tetangga tersebut diterima sebagai state baru berdasarkan selisih cost (delta) dan suhu saat ini—dengan kemungkinan menerima solusi yang lebih buruk secara probabilistik untuk

menghindari jebakan local optimum. Jika state baru diterima, struktur pelacakan diperbarui dan solusi terbaik disimpan jika lebih optimal; jika tidak, penghitung stagnasi bertambah. Di akhir setiap iterasi, suhu dikurangi berdasarkan *cooling rate*, dan pencarian dihentikan lebih awal jika terlalu lama tanpa peningkatan. Untuk mengatasi kebuntuan di local optimum, algoritma melakukan *random restart* dari state acak jika diperlukan, sambil menjaga suhu tetap cukup tinggi untuk memperluas eksplorasi solusi.

```
// Jika tidak menemukan solusi optimal, kembalikan path ke best state yang
ditemukan
    if (bestBoard.isGoal()) {
        return reconstructPathFromMap(bestBoard, parentMap);
    }

    // Jika SA tidak menemukan solusi, coba fallback dengan hill climbing
    sederhana
    return hillClimbingFallback(bestBoard, parentMap, costMap);
}
```

Bagian akhir dari fungsi `simulatedAnnealing` ini menangani kondisi setelah proses iteratif selesai. Jika state terbaik yang ditemukan selama proses merupakan solusi yang valid (goal), maka algoritma langsung merekonstruksi dan mengembalikan jalur solusi tersebut menggunakan `parentMap`. Namun, jika Simulated Annealing gagal menemukan solusi yang mencapai goal, maka algoritma tidak langsung menyerah; sebagai gantinya, dilakukan *fallback* menggunakan strategi hill climbing sederhana. Strategi ini mencoba memperbaiki solusi dari state terbaik yang pernah ditemukan dengan cara mengeksplorasi tetangga yang memiliki cost lebih rendah secara deterministik. Pendekatan ini menjaga kemungkinan menemukan solusi meskipun SA mengalami stagnasi atau terjebak di local optimum.

Iterative Deepening Search

```
private List<GameBoard> iterativeDeepeningSearch(GameBoard initialBoard) {
    int depth = 0;
    while (true) {
        Set<String> visited = new HashSet<>();
        nodesVisited = 0;
        Node root = new Node(initialBoard, null, 0, 0);
        List<GameBoard> result = depthLimitedSearch(root, depth, visited);
        if (result != null) {
            return result;
        }
        depth++;
        if (depth > 1000) {
            return null;
        }
    }
}
```

Fungsi memulai pencarian dengan kedalaman awal `depth = 0` dan terus meningkatkannya satu per satu menggunakan loop tak hingga. Pada setiap iterasi, struktur `visited` diinisialisasi ulang

untuk memastikan pencarian bersih dari loop sebelumnya, lalu dilakukan pemanggilan `depthLimitedSearch`, yaitu pencarian DFS terbatas hingga kedalaman saat ini. Jika solusi ditemukan (hasil tidak null), maka jalur solusi langsung dikembalikan. Jika belum, kedalaman ditingkatkan dan proses diulang. Pembatasan kedalaman maksimum ($\text{depth} > 1000$) digunakan sebagai mekanisme *early termination* untuk mencegah infinite loop pada kasus yang tidak memiliki solusi atau eksplorasi sangat dalam.

Reconstructing Path

Reconstructing Path digunakan untuk membentuk jalur solusi per tahap dari posisi awal ke posisi akhir untuk ditampilkan kepada pengguna.

```
private List<GameBoard> reconstructPath(Node goalNode) {  
    List<GameBoard> path = new ArrayList<>();  
    Node current = goalNode;  
    while (current != null) {  
        path.add(current.getBoard());  
        current = current.getParent();  
    }  
    Collections.reverse(path);  
    return path;  
}
```

Di awal, fungsi membuat list kosong bernama `path`, berupa list yang akan menyimpan urutan konfigurasi papan dari solusi hingga awal. Lalu dilakukan inisialisasi `current` berupa `goalNode`, berupa node yang memenuhi solusi. Selama `current` tidak bernilai null, `current` node ditambahkan ke list `path` dan `current` diganti dengan `parent` dari node saat ini. Apabila `current` null (root state sudah masuk ke dalam list `path`), dilakukan `reverse` pada list `path`, dan dikembalikan ke luar fungsi.

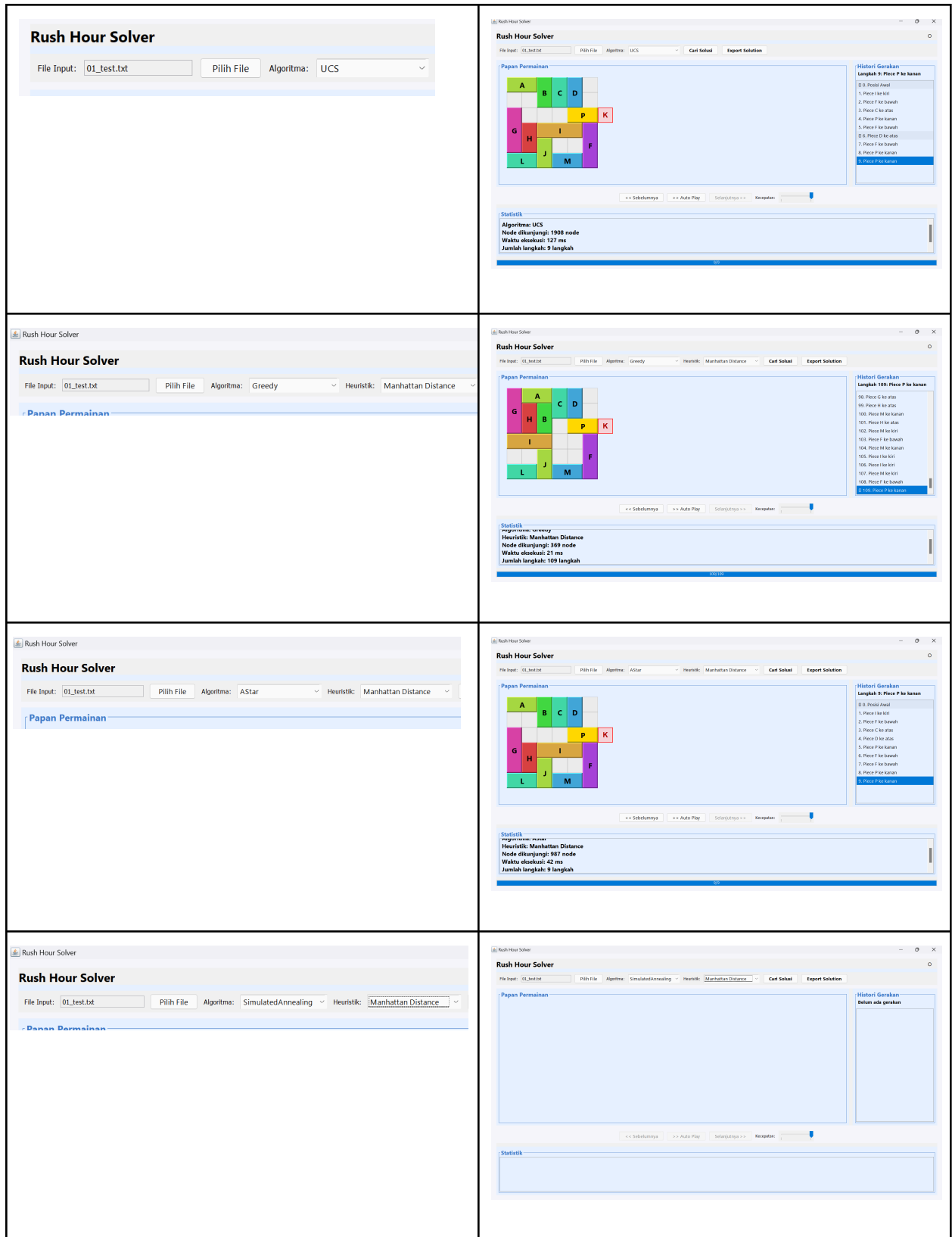
4.2 Pengujian Program

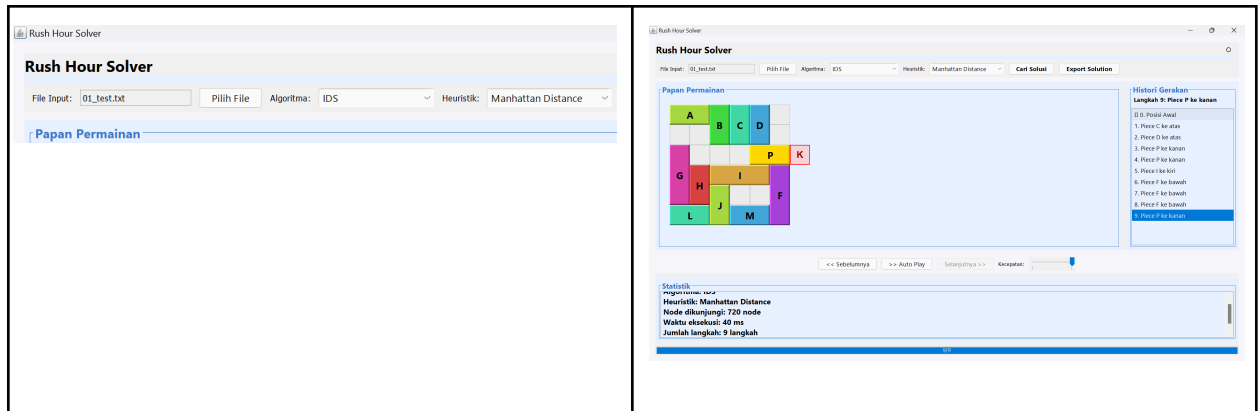
Pengujian dilakukan dengan menggunakan 4 file tes dengan isi sebagai berikut:

<pre> 6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM. </pre>	<pre> 6 6 9 AA.P.. BCCP.. BD.EEE .D.FFF .GGG.. .HHII. K </pre>
File Tes 1	File Tes 2
<pre> 6 6 13 K ABBCCD AEPFFD GEP.HH GEIIJJ NNNL.. ...LMM </pre>	<pre> 6 6 9 ..AABB .CCD.. .E.DFF K.E.PPG .E..HG .IIIHG </pre>
File Tes 3	File Tes 4

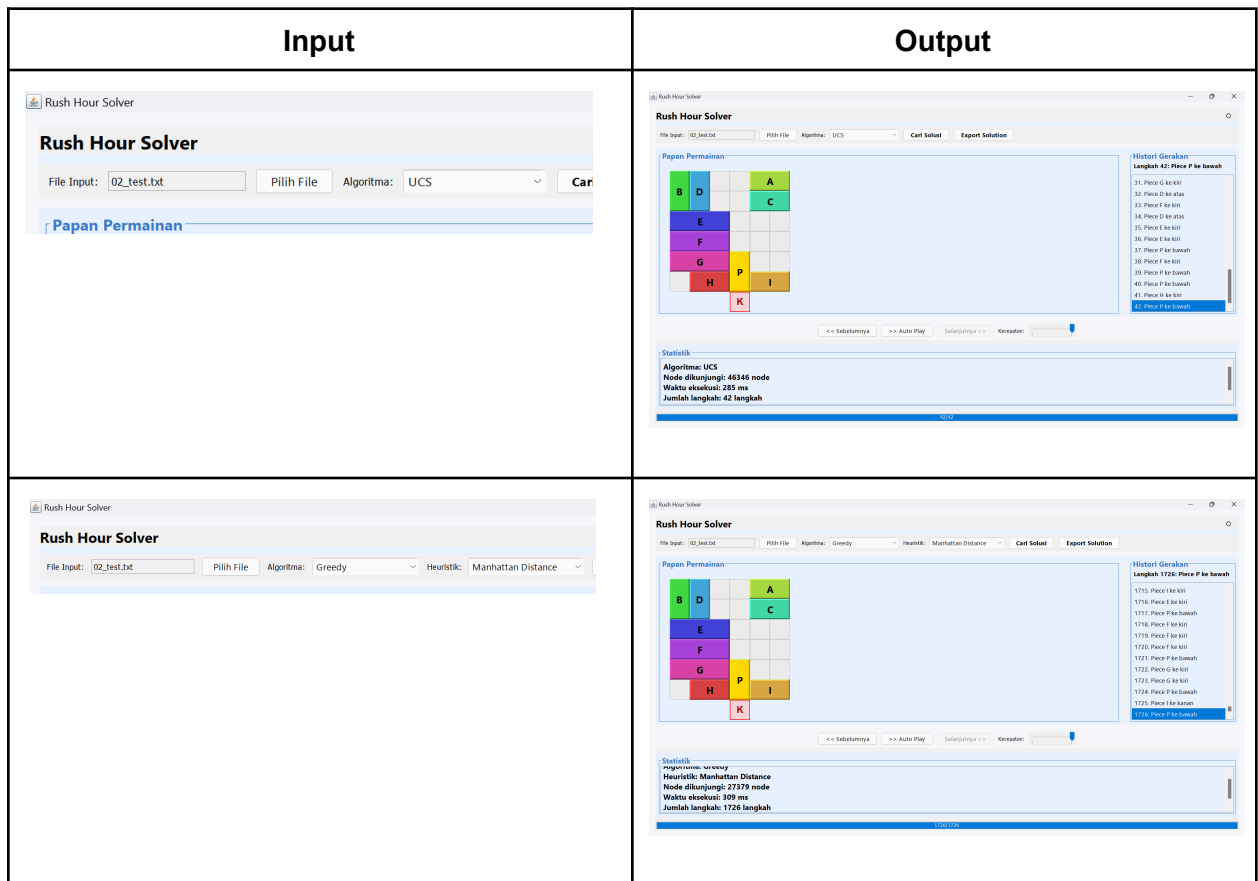
Test File #1

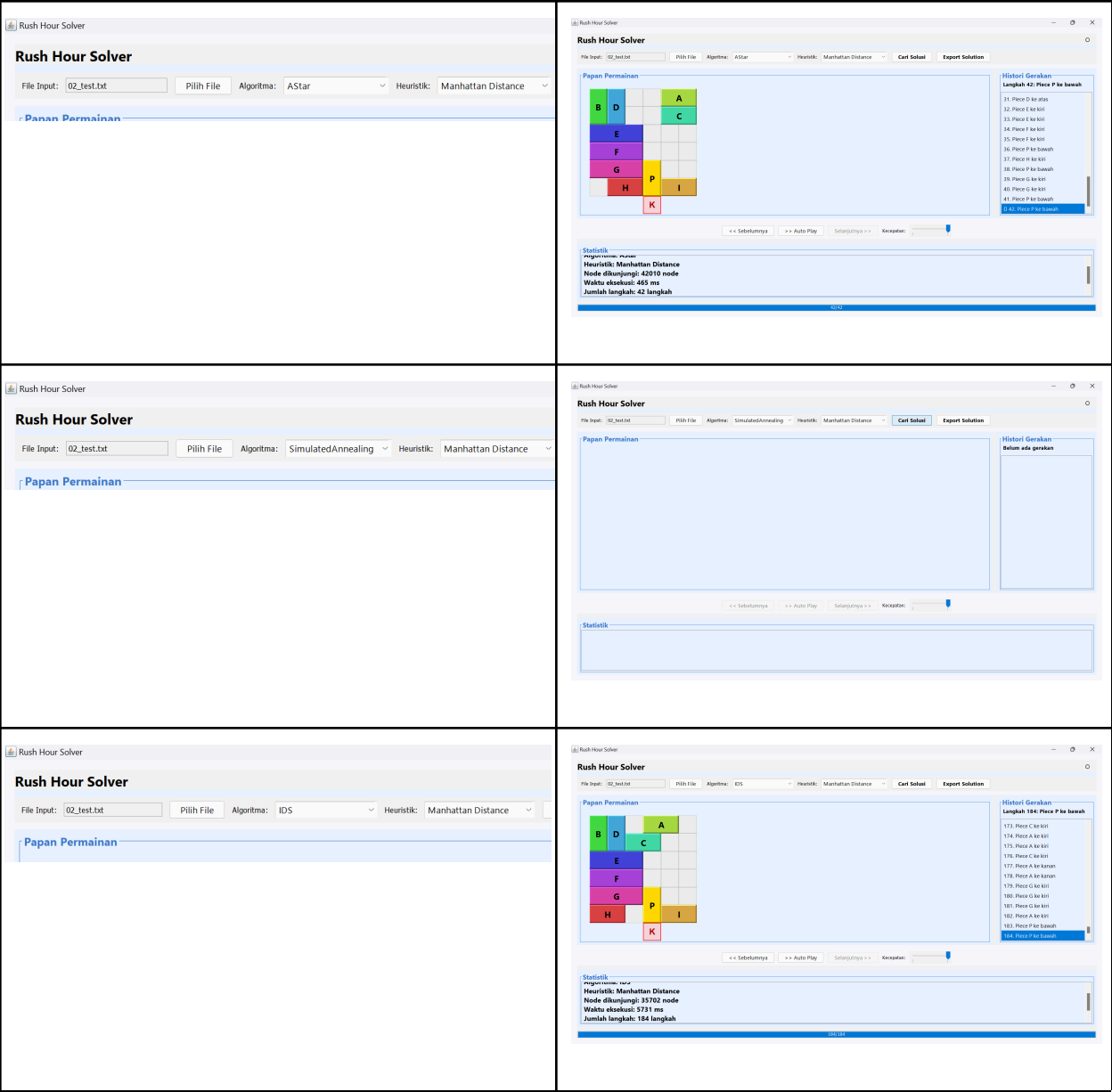
Input	Output
-------	--------





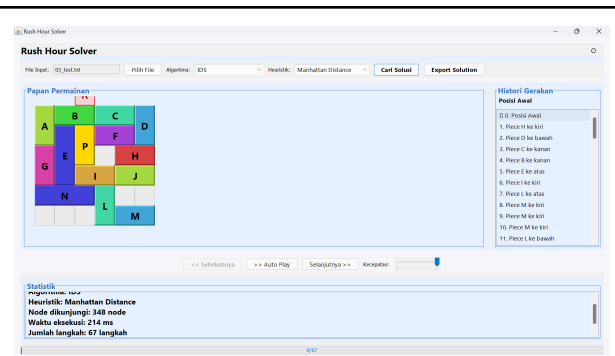
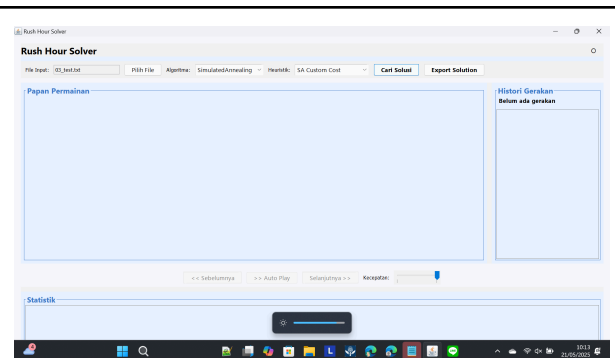
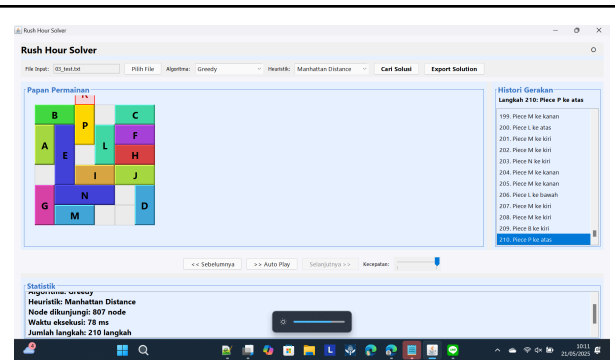
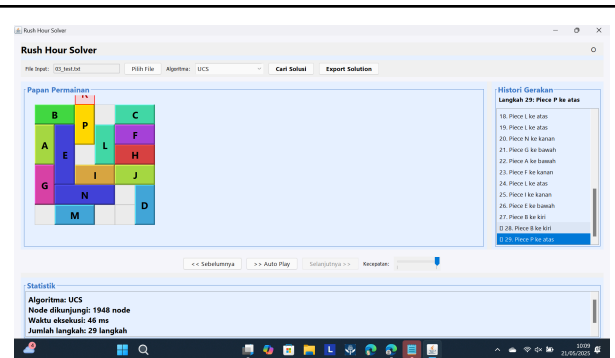
Test File #2



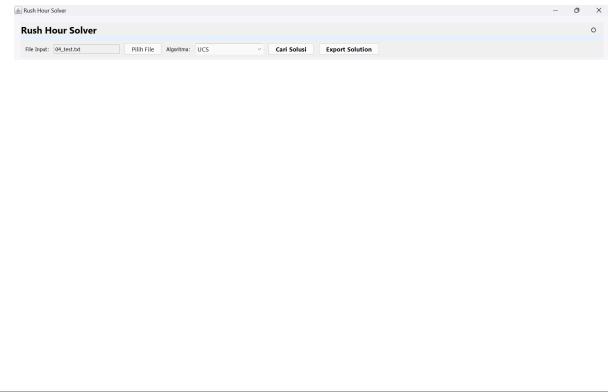
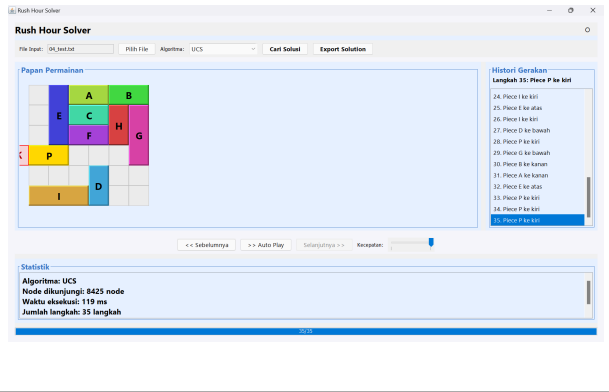
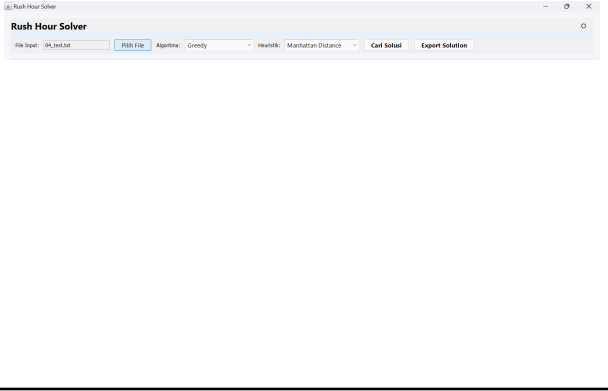
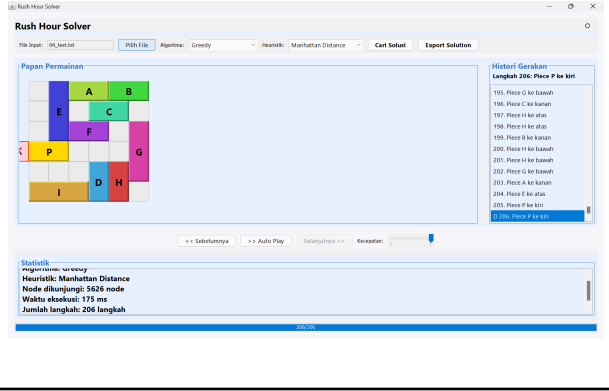
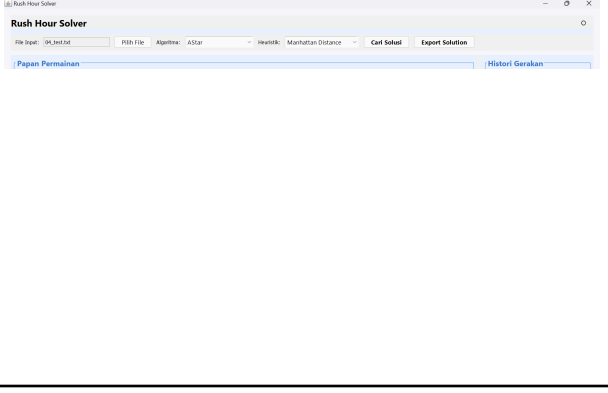
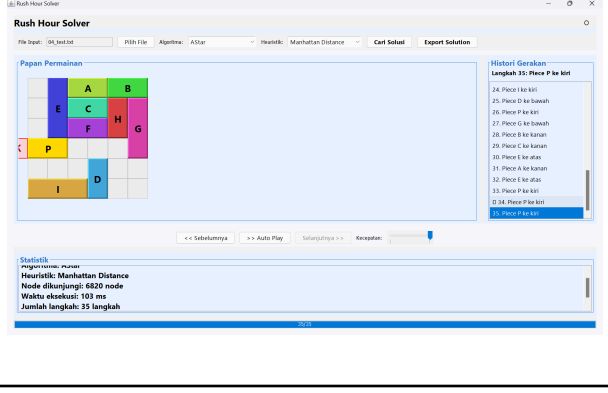


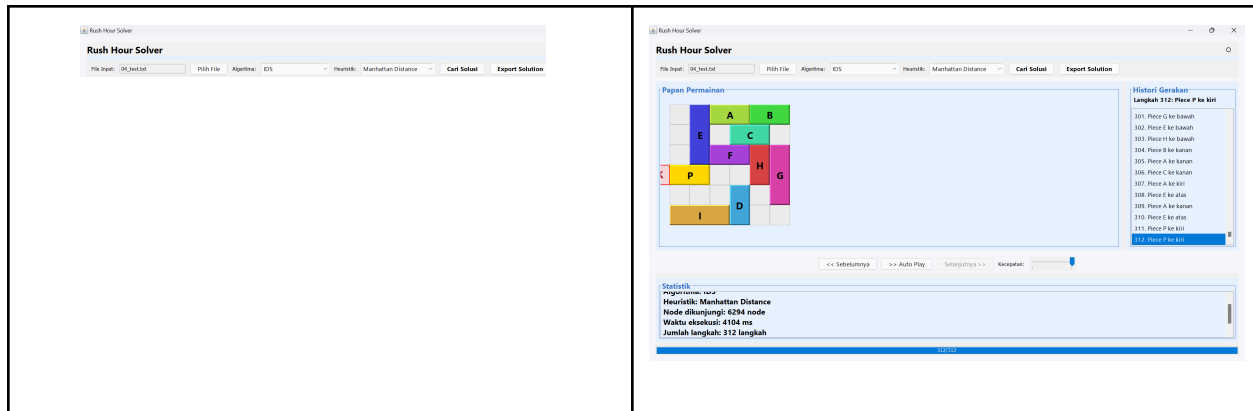
Test File #3

Input	Output
-------	--------



#Test File 4

Input	Output
	
	
	



4.3. Analisis Pengujian Program

Pada test file 1, didapatkan bahwa A* dapat menempuh langkah yang sama pendeknya dengan IDS dan Greedy Best First Search, namun dengan waktu yang paling cepat.

Pada test file 2, didapatkan bahwa A* dapat menempuh langkah yang paling pendek bersama Greedy Best First Search, namun dengan waktu yang lebih cepat.

Pada test file 3, didapatkan bahwa UCS dapat menempuh langkah yang paling pendek dibanding lainnya.

Pada test file 4, didapatkan UCS dan A* dapat menempuh langkah yang lebih pendek, namun A* memiliki waktu yang lebih cepat dibandingkan UCS.

Sementara Simulated Annealing, tidak dapat memunculkan solusi, karena konstanta temperatur dan lainnya yang masih bersifat *default*, masih bersifat kurang cocok di pengujian ini.

Dari keempat test file diatas, dinyatakan bahwa A* dapat memberikan solusi yang terbaik dalam menentukan langkah dan waktu yang efisien. Hal ini dikarenakan, A* mengkombinasikan dari UCS dan Greedy Best First Search, dengan bantuan beberapa heuristik. Sementara pada test file 3, didapatkan UCS yang lebih efisien dalam penentuan langkah, karena kebetulan UCS tidak memperhitungkan perhitungan heuristik, terutama Manhattan Blocking, dimana blocking sekitar tidak diperhitungkan dalam langkah, sementara A* akan memperhitungkan heuristik ini juga, sehingga menambah langkah, yang sebenarnya kurang perlu

Bab 5

Kesimpulan dan Saran

5.1 Kesimpulan

Berdasarkan implementasi dan pengujian terhadap berbagai algoritma pencarian dalam menyelesaikan permasalahan permainan Rush Hour, diperoleh beberapa temuan penting:

1. Efektivitas Algoritma Bervariasi Sesuai Karakteristik Masalah:
 - Uniform Cost Search (UCS) menjamin solusi optimal dengan jumlah langkah minimum karena mempertimbangkan biaya kumulatif dari awal. Namun, ia membutuhkan waktu eksekusi dan memori yang lebih besar.
 - Greedy Best-First Search (GBFS) lebih cepat dalam mencapai solusi karena hanya mengandalkan nilai heuristik, tetapi hasilnya tidak selalu optimal.
 - A* Search mampu menyeimbangkan eksplorasi dan efisiensi, menghasilkan solusi optimal lebih cepat dibanding UCS, asalkan heuristik yang digunakan bersifat admissible.
 - Simulated Annealing cukup baik dalam menjelajahi ruang solusi besar secara efisien, meski tidak menjamin keoptimalan. Algoritma ini menjadi solusi alternatif ketika metode deterministik gagal atau terlalu lambat.
 - Iterative Deepening Search (IDS) efektif dalam penggunaan memori dan sistematis dalam pencarian, namun kurang efisien pada ruang pencarian besar karena eksplorasi ulang yang berulang.
2. Heuristik Berperan Krusial pada Efisiensi Algoritma:
 - Kombinasi Manhattan Distance dan Blocking Heuristic (Enhanced Heuristic) terbukti memberikan performa pencarian yang paling seimbang dalam akurasi dan kecepatan.
 - Penggunaan heuristik yang tidak relevan atau terlalu sederhana dapat menyebabkan algoritma tersesat atau menghasilkan banyak langkah tidak efisien.
3. Fitur Interaktif Meningkatkan Usabilitas:
 - Adanya antarmuka grafis, input file eksternal, dan kontrol kecepatan visualisasi solusi mempermudah pengguna dalam memahami alur pencarian dan membandingkan efektivitas algoritma.

5.2 Saran

Untuk pengembangan lebih lanjut dan peningkatan kualitas aplikasi, berikut beberapa saran yang dapat dipertimbangkan:

1. Optimasi Struktur Data:
 - Gunakan struktur hashing atau bitmap encoding yang lebih efisien untuk representasi state, guna mempercepat pengecekan visited states dan mengurangi kompleksitas waktu.
2. Penambahan Algoritma Lain
 - Pertimbangkan integrasi algoritma seperti IDA*, Bidirectional Search, atau Genetic Algorithms untuk variasi pendekatan penyelesaian yang lebih adaptif terhadap kasus sulit.
3. Visualisasi Dinamis dan Penilaian Solusi:
 - Tambahkan metrik evaluasi solusi seperti jumlah langkah, waktu pencarian, dan memory footprint yang dapat dibandingkan antarmuka secara langsung.
 - Visualisasi animasi pemindahan kendaraan dalam GUI bisa memperjelas dinamika solusi yang terbentuk.
4. Penggunaan Benchmark Board:
 - Lakukan pengujian terhadap berbagai benchmark puzzles yang sudah dikenal dalam komunitas Rush Hour untuk validasi performa terhadap kasus nyata atau ekstrem.
5. Ekspor Laporan Solusi:
 - Tambahkan fitur untuk mengeksport solusi dalam format teks atau gambar sebagai dokumentasi hasil pencarian yang dapat digunakan untuk analisis atau pembelajaran lebih lanjut.

Lampiran

Github Repository: [rlukassa/Tucil3_13523131_13523158 at newfeat/GUI](https://github.com/rlukassa/Tucil3_13523131_13523158_at_newfeat/GUI)

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	✓	