

IF2211 Strategi Algoritma

Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding

Laporan Tugas Kecil 3

Disusun untuk memenuhi tugas mata kuliah IF2211 Strategi Algoritma

Semester II Tahun Akademik 2024/2025



Disusun oleh:

Natalia Desiany Nursimin - 13523157

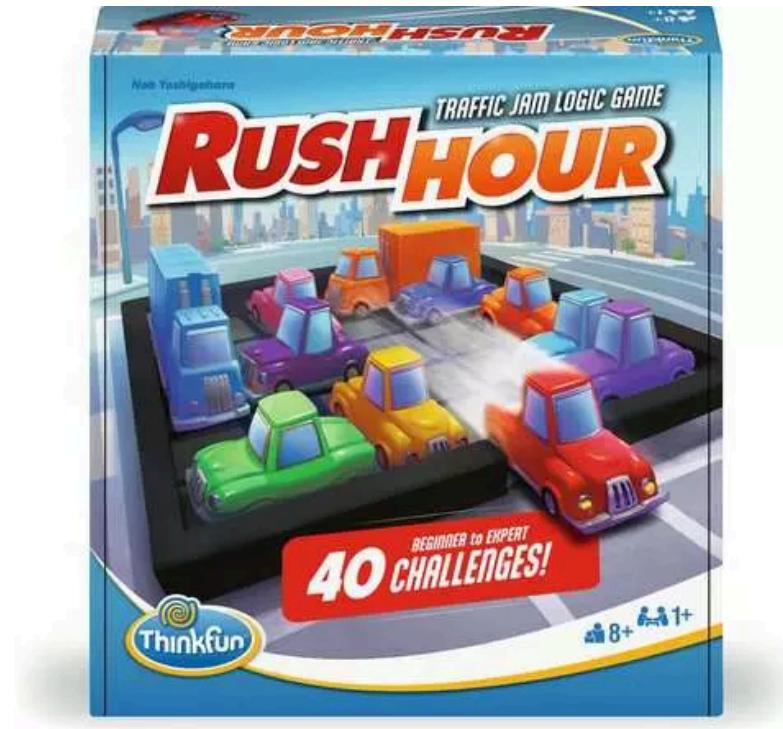
Anas Ghazi Al Ghifari - 13523159

**PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2025**

DAFTAR ISI

BAB I DESKRIPSI MASALAH.....	3
BAB II LANDASAN TEORI.....	12
BAB III PENJELASAN ALGORITMA.....	14
3.1. UCS.....	14
3.2. Greedy Best First Search.....	17
3.3. A*.....	19
3.4. Beam Search.....	22
3.5. Analisis Algoritma Pathfinding.....	25
3.5.2. Admissibility Heuristik pada algoritma A*.....	25
3.5.3. UCS vs BFS pada Rush Hour.....	26
3.5.5. Optimalitas Greedy Best First Search.....	27
BAB IV SOURCE PROGRAM.....	29
4.1. Board.....	30
4.2. Piece.....	40
4.3. Node.....	42
4.4. Solver.....	44
4.5. Solution.....	45
4.6. Main.....	53
BAB V PENGUJIAN DAN HASIL PERCOBAAN ALGORITMA PATHFINDING.....	71
5.1. Uji Coba Dengan Tangkapan Layar.....	71
5.1.1. Tabel Pengujian Algoritma Greedy Best First Search dengan Heuristic Manhattan Distance to Exit.....	71
5.1.2. Tabel Pengujian Algoritma Greedy Best First Search dengan Heuristic Blocking Vehicles Count.....	73
5.1.3. Tabel Pengujian Algoritma UCS (Uniform Cost Search).....	75
5.1.4. Tabel Pengujian Algoritma A* dengan Heuristic Manhattan Distance to Exit.....	77
BAB VI IMPLEMENTASI BONUS.....	80
6.1. Algoritma Alternatif (Beam Search).....	80
6.1.1. Kode Pengimplementasian Beam Search.....	80
6.1.2. Penjelasan Kode Beam Search.....	81
6.2. Heuristik Alternatif (Jarak Manhattan ke Exit dan Blocking Pieces).....	83
6.2.1. Kode Pengimplementasian Heuristik.....	83
6.2.2. Penjelasan Kode Heuristik.....	85
Pranala Repository:.....	87
Tabel Checklist Spesifikasi Program:.....	87

BAB I DESKRIPSI MASALAH



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.

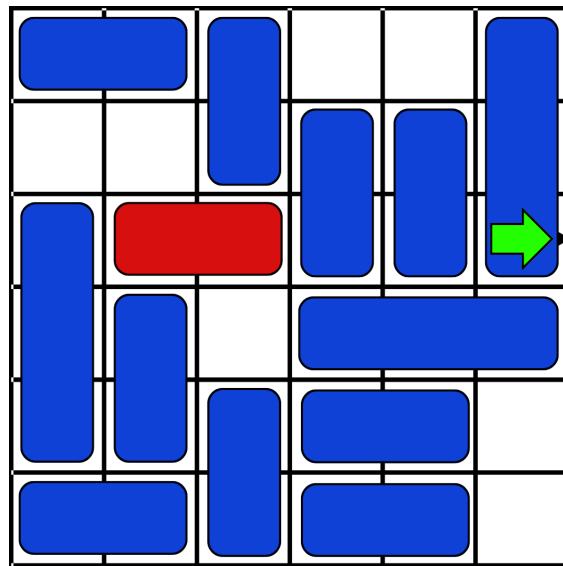
Papan terdiri atas *cell*, yaitu sebuah *singular point* dari papan. Sebuah *piece* akan menempati *cell-cell* pada papan. Ketika permainan dimulai, semua *piece* telah diletakkan di dalam papan dengan konfigurasi tertentu berupa lokasi *piece* dan *orientasi*, antara *horizontal* atau *vertikal*.

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.

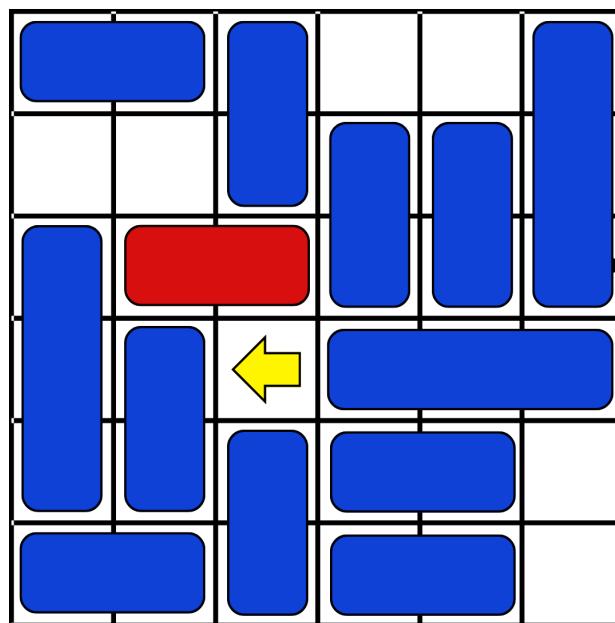
Ilustrasi kasus :

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.

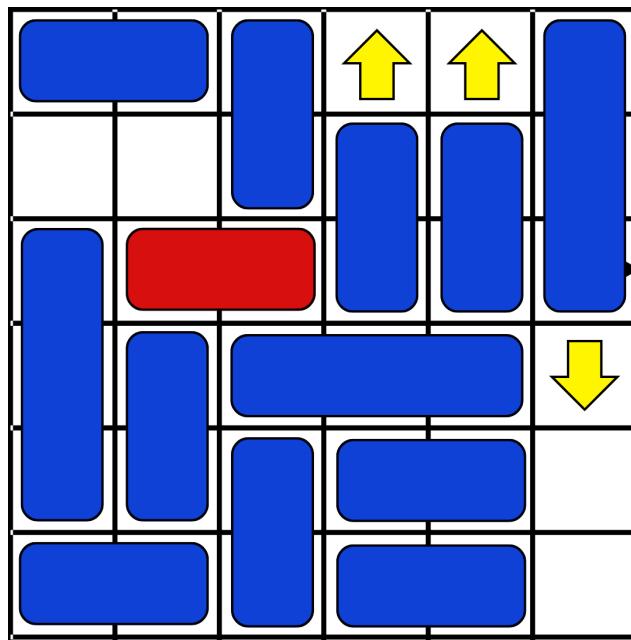


Gambar 2. Awal Permainan Game Rush Hour

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

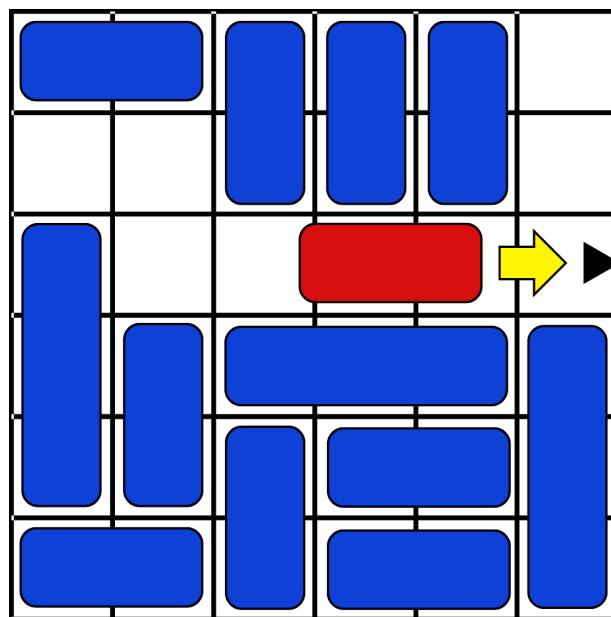


Gambar 3. Gerakan Pertama Game Rush Hour



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

Program Rush Hour Solver dirancang dengan pendekatan modular yang memungkinkan penggunaan berbagai algoritma pencarian untuk menemukan solusi optimal. Alur dari program ini adalah sebagai berikut:

1. **[INPUT]** **konfigurasi permainan/test case** dalam format ekstensi **.txt**. File *test case* tersebut berisi:
 1. **Dimensi Papan** terdiri atas dua buah variabel **A** dan **B** yang membentuk papan berdimensi $A \times B$
 2. **Banyak piece BUKAN** *primary piece* direpresentasikan oleh variabel integer **N**.
 3. **Konfigurasi papan** yang mencakup penempatan *piece* dan *primary piece*, serta lokasi *pintu keluar*. *Primary Piece* dilambangkan dengan huruf **P** dan pintu keluar dilambangkan dengan huruf **K**. *Piece* dilambangkan dengan huruf dan karakter selain *P* dan *K*, dan huruf/karakter berbeda melambangkan *piece* yang berbeda. *Cell* kosong dilambangkan dengan karakter ‘.’ (titik). (**Catatan:** ingat bahwa *pintu keluar* pasti berada di *dinding* papan dan sejajar dengan orientasi *primary piece*)

File .txt yang akan dibaca memiliki format sebagai berikut:

```
A B  
N  
konfigurasi_papan
```

Contoh Input

```
6 6  
11
```

AAB..F

..BCDF

GPPCDF**K**

GH.III

GHJ...

LLJMM.

keterangan: “K” adalah pintu keluar, “P” adalah primary piece, Titik (“.”) adalah cell kosong.

Contoh konfigurasi papan lain yang mungkin berdasarkan letak *pintu keluar* (X adalah *piece/cell random*)

K	XXX	XXX
XXX	XXXX	XXX
XXX	XXX	XXX
XXX		K

2. [INPUT] algoritma *pathfinding* yang digunakan
3. [INPUT] *heuristic* yang digunakan (**bonus**)
4. [OUTPUT] Banyaknya gerakan yang diperiksa (alias banyak ‘node’ yang dikunjungi)
5. [OUTPUT] Waktu eksekusi program
6. [OUTPUT] konfigurasi *papan* pada setiap tahap pergerakan/pergeseran. Output ini tidak harus diimplementasi apabila mengerjakan *bonus output GUI*. **Gunakan print berwarna**

untuk menunjukkan pergerakan *piece* dengan jelas. Cukup mewarnakan ***primary piece***, **pintu keluar**, dan ***piece yang digerakkan*** saja (boleh dengan *highlight* atau *text color*). Pastikan ketiga komponen tersebut memiliki warna berbeda.

Format sekuens adalah sebagai berikut:

Papan Awal

[konfigurasi_papan_awal]

Gerakan 1: [piece]-[arah gerak]

[konfigurasi_papan_gerakan_1]

Gerakan 2: [piece]-[arah gerak]

[konfigurasi_papan_gerakan_2]

Gerakan [N]: [piece]-[arah gerak]

[konfigurasi_papan_gerakan_N]

dst

Contoh Output

Papan Awal

AAB..F

..BCDF

G**PP**CDF**K**

GH.III

GHJ...

LLJMM.

Gerakan 1: I-kiri
AAB..F

..BCDF

GPPCDFK

GHIII

GHJ...

LLJMM.

Gerakan 2: F-bawah
AAB..I

..BCD**F**

GPPCDF**K**

GHIII**F**

GHJ...

LLJMM.

dst

Keterangan: **hanya sebagai contoh**. Pastikan output jelas dan mudah dimengerti. Warna dan highlight hanya untuk menunjukan perubahan.

7. [OUTPUT] animasi gerakan-gerakan untuk mencapai solusi (**bonus GUI**).

BAB II LANDASAN TEORI

Proses pencarian jalur atau *pathfinding* merupakan sebuah proses untuk menentukan urutan langkah terbaik dari suatu titik awal menuju titik tujuan dalam suatu sistem yang memiliki aturan dan batasan tertentu. Permasalahan *pathfinding* sering kita temui dalam kehidupan sehari-hari, seperti dalam mencari rute tercepat ke suatu lokasi, pengoptimalan pengiriman rute logistik, hingga dalam menentukan strategi dalam permainan. *Pathfinding* sangat penting karena berfokus pada efisiensi, baik dari segi waktu, jarak tempuh, maupun jumlah langkah menuju tujuan. Untuk membantu menyelesaikan masalah ini secara sistematis, dibutuhkan algoritma *pathfinding* yang mampu mengevaluasi berbagai kemungkinan dan memilih jalur terbaik berdasarkan seluruh informasi yang tersedia. Pendekatan ini sangat relevan, terutama pada masalah dengan banyak kemungkinan kombinasi, seperti pada permainan logika berbasis grid.

Salah satu contoh dari permasalahan *pathfinding* adalah permainan *Rush Hour*, sebuah puzzle berbasis logika dengan papan berukuran 6x6 yang berisi kendaraan dengan panjang dan orientasi berbeda. Kendaraan hanya dapat bergerak lurus sesuai dengan orientasinya, yaitu antara horizontal atau vertikal. Tujuan utama dari permainan ini adalah untuk mengeluarkan kendaraaan

utama (*primary piece*) dari kemacetan dengan menggeser kendaraan lain yang menghalangi jalannya, hingga mencapai pintu keluar di sisi papan. Tantangan utama yang dihadapi dalam permainan ini adalah keterbatasan ruang dan banyaknya kemungkinan gerakan. Permainan ini dapat dimodelkan sebagai permasalahan dalam ruang keadaan (*state space*), di mana setiap konfigurasi papan yang valid adalah sebuah *state*, dan perpindahan antar *state* terjadi setiap kali kendaraan digerakkan ke posisi yang *valid*. Solusi dicari dengan menelusuri jalur dari *initial state* (konfigurasi awal papan) menuju *goal state*, yaitu *primary piece* berada tepat di depan pintu keluar.

Untuk dapat menjelajahi *state space* secara efisien, digunakan algoritma pencarian dengan strategi eksplorasi tertentu. Secara umum, algoritma pencarian dibagi menjadi dua kelompok utama, yaitu *uninformed search* dan *informed search*. *Uninformed search* bekerja tanpa informasi tambahan selain struktur dasar masalah, dan melakukan eksplorasi sistematis. Contoh algoritma bertipe ini adalah *Uniform Cost Search* (UCS). Sebaliknya, *informed search* memanfaatkan *heuristic*, yaitu estimasi jarak dari *state* saat ini ke tujuan, untuk membimbing pencarian secara lebih terarah. Contoh algoritma bertipe ini adalah *Greedy Best First Search* dan *A**.

Heuristik berperan penting dalam algoritma *informed search* karena menentukan seberapa dekat suatu *state* terhadap solusi. Dalam konteks permainan *Rush Hour*, terdapat beberapa pendekatan heuristik yang umum digunakan. Salah satu pendekatan adalah heuristik berbasis jarak Manhattan, yaitu estimasi jarak horizontal dari posisi primary piece saat ini ke pintu keluar. Nilai ini merepresentasikan jumlah minimum langkah yang dibutuhkan jika tidak ada kendaraan lain yang menghalangi. Pendekatan lain adalah *blocking pieces heuristic*, yaitu pendekatan yang menghitung jumlah kendaraan yang secara langsung menghalangi jalur primary piece menuju pintu keluar. Heuristik ini menilai berapa banyak kendaraan yang perlu dipindahkan agar primary piece dapat mencapai tujuan. Kedua metode heuristik ini memberikan gambaran awal tentang kedekatan *state* saat ini dengan *goal state*, yang kemudian akan digunakan dalam proses pemilihan jalur pada algoritma.

Setiap algoritma memiliki karakteristik, kelebihan, dan keterbatasannya masing-masing dalam menyelesaikan permasalahan pencarian jalur. *Uniform Cost Search* (UCS) adalah algoritma *uninformed search* yang menggunakan pendekatan berbasis biaya untuk menemukan jalur dari titik awal ke tujuan dengan total biaya terendah. UCS mengembangkan simpul berdasarkan akumulasi biaya dari titik awal, dengan fungsi evaluasi $f(n) = g(n)$, di mana $g(n)$ adalah total biaya dari simpul awal menuju *state* n. UCS menjamin solusi optimal selama semua biaya langkah bersifat non-negatif dan bekerja dengan memperluas simpul berdasarkan urutan biaya kumulatif terkecil terlebih dahulu, sehingga eksplorasi dilakukan secara menyeluruh dan sistematis, meskipun dapat menjadi lambat pada ruang pencarian yang besar.

Sementara itu, *Greedy Best First Search* merupakan algoritma pencarian *informed search* yang menggunakan fungsi heuristik untuk memperkirakan jarak dari node saat ini ke tujuan, tanpa memperhitungkan biaya yang telah dikeluarkan sejauh ini. Algoritma ini menggunakan fungsi evaluasi $f(n) = h(n)$, di mana $h(n)$ adalah fungsi heuristik yang mengestimasi jarak atau langkah minimum dari node n ke goal. Pendekatan ini bersifat *greedy* karena selalu memilih node yang tampak paling dekat ke tujuan berdasarkan nilai heuristik tersebut, tanpa mempertimbangkan apakah jalur tersebut menghasilkan biaya total yang rendah secara keseluruhan. Dalam banyak kasus, algoritma ini mampu menemukan solusi dengan cepat karena fokus eksplorasinya lebih terarah. Namun, karena hanya mengandalkan estimasi ke depan dan mengabaikan biaya yang telah ditempuh, algoritma ini tidak menjamin ditemukannya solusi optimal dan dapat melewatkana jalur yang sebenarnya lebih baik.

A^* menggabungkan keunggulan dari UCS dan Greedy dengan memanfaatkan fungsi evaluasi $f(n) = g(n) + h(n)$, yang mempertimbangkan baik biaya nyata dari simpul awal ($g(n)$) maupun estimasi biaya menuju tujuan ($h(n)$). Jika heuristik yang digunakan bersifat *admissible* dan *consistent*, A^* mampu menemukan solusi optimal secara efisien.

Algoritma *Beam Search* merupakan variasi dari pencarian heuristik yang bertujuan menghemat sumber daya dengan membatasi jumlah simpul yang dievaluasi di setiap tingkat eksplorasi. Algoritma ini hanya mempertahankan sejumlah simpul terbaik pada setiap level pencarian berdasarkan nilai heuristik tertinggi, yang disebut *beam width*. Meskipun pendekatan ini mempercepat pencarian dan mengurangi penggunaan memori, ia tidak menjamin tercapainya solusi optimal karena tidak mengeksplorasi seluruh kemungkinan jalur yang ada.

BAB III PENJELASAN ALGORITMA

3.1. UCS

Uniform Cost Search (UCS) adalah sebuah algoritma pencarian berjenis *uninformed search* yang memperluas node berdasarkan biaya perjalanan terendah (*cost*) yang diperlukan untuk mencapai node tersebut. Algoritma ini menggunakan *priority queue* sebagai struktur data utama, yang memprioritaskan ekspansi node dengan nilai biaya terkecil. Dalam konteks permainan *Rush Hour*, biaya ini direpresentasikan oleh jumlah langkah atau pergeseran *piece* yang dilakukan agar *primary piece* dapat sampai ke pintu keluar. Algoritma ini memiliki kelebihan utama, yaitu kemampuannya untuk menjamin solusi yang ditemukan merupakan solusi optimal dengan biaya terkecil, selama setiap langkah memiliki biaya positif. Hal ini membuat algoritma UCS sangat sesuai untuk digunakan dalam permainan seperti *Rush Hour*, di mana tujuan akhirnya adalah untuk mengeluarkan *primary piece* dari papan dengan jumlah langkah

seminimal mungkin. Algoritma ini akan mengeksplorasi semua kemungkinan solusi berdasarkan urutan biaya dari yang paling rendah ke yang tertinggi, kemudian menjelajahi ruang solusi dan akan berhenti di saat berhasil mencapai tujuan dengan total biaya terkecil.

Secara umum, UCS bekerja dengan cara menginisialisasi sebuah *priority queue* yang berisi node awal, kemudian secara iteratif mengambil node dengan biaya terendah dari antrian untuk diperluas. Biaya yang digunakan dalam UCS merupakan biaya *real cost* dari node awal ke node tersebut, yang dilambangkan sebagai $g(n)$. Proses berlanjut hingga node tujuan ditemukan atau semua kemungkinan konfigurasi telah dieksplorasi. Untuk menghindari eksplorasi terhadap konfigurasi yang sudah pernah dikunjungi, UCS menggunakan sebuah *closed set* yang mencatat semua konfigurasi papan yang telah diekspansi. Ketika sebuah node diambil dari antrian, algoritma memeriksa apakah konfigurasi papan dari node tersebut adalah solusi. Jika iya, maka jalur dari keadaan awal hingga konfigurasi akhir tersebut akan dikembalikan sebagai solusi. Jika tidak, maka node tersebut akan diperluas dengan menghasilkan semua kemungkinan *legal moves* dari *pieces* pada papan, yang kemudian menghasilkan node-node *successors*.

Setiap successor yang dihasilkan akan kemudian diperiksa. Jika konfigurasinya belum pernah dikunjungi atau tidak ada dalam antrian, maka node tersebut langsung dimasukkan ke dalam priority queue. Namun, jika node tersebut sudah berada di dalam antrian tetapi memiliki biaya yang lebih tinggi, maka UCS akan mengganti node tersebut dengan versi baru yang memiliki biaya lebih rendah. Dengan strategi ini, UCS dapat memastikan bahwa setiap jalur menuju sebuah konfigurasi selalu diperbarui dengan jalur dengan biaya terkecil, sehingga solusi yang dihasilkan dapat memiliki total biaya minimum.

Pada permainan Rush Hour, setiap konfigurasi papan permainan direpresentasikan sebagai *state* yang perlu dieksplorasi. Gerakan yang mungkin dari setiap kendaraan dihitung berdasarkan aturan permainan, yaitu bergerak secara horizontal atau vertikal tergantung dengan orientasinya. Biaya setiap langkah diberi nilai 1, sehingga total biaya merupakan jumlah langkah yang telah dilakukan sejak konfigurasi keadaan awal hingga mencapai kondisi solusi.

Berikut adalah kode pengimplementasian UCS yang telah dibuat dalam konteks permainan *Rush Hour*:

```
✓ import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Set;

You, 2 days ago | 1 author (You)
✓ public class UniformCostSearch extends Solver {

    ✓ public UniformCostSearch(Board initialBoard) {
        super(initialBoard, heuristicChoice:0); // UCS tidak menggunakan heuristic
    }

    @Override
    ✓ public Solution solve() {
        // Antrian prioritas berdasarkan cost terendah
        PriorityQueue<Node> openSet = new PriorityQueue<>(Comparator.comparingInt(Node::getCost));
        Set<Board> closedSet = new HashSet<>(); // Jika sudah dikunjungi

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

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

            if (current.getBoard().isSolved()) {
                return new Solution(current.getPath());
            }

            ✓ if (closedSet.contains(current.getBoard())) {
                continue;| You, 2 days ago • feat: add UCS, basic solver and solution implem...
            }
            closedSet.add(current.getBoard());
        }
    }
}
```

```

        List<int[]> legalMoves = current.getBoard().getLegalMoves();
        for (int[] move : legalMoves) {
            int vehicleIdx = move[0];
            int moveAmount = move[1];

            Board newBoard = current.getBoard().applyMove(vehicleIdx, moveAmount);
            if (closedSet.contains(newBoard)) {
                continue;
            }

            Node successor = new Node(current, newBoard, vehicleIdx, moveAmount, heuristicChoice:0);

            boolean foundBetter = false;
            for (Node openNode : openSet) {
                if (openNode.getBoard().equals(newBoard) && openNode.getCost() <= successor.getCost()) {
                    foundBetter = true;
                    break;
                }
            }

            if (!foundBetter) {
                openSet.add(successor);
            }
        }
        return null; // Jika tidak ditemukan solusi
    }

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

```

Pengimplementasian algoritma UCS dilakukan melalui kelas `UniformCostSearch` yang merupakan turunan dari kelas abstrak `Solver`. Di dalam kelas ini, metode `solve()` mengimplementasikan inti dari algoritma UCS. *Priority queue* disini digunakan untuk menyimpan node-node yang akan diekspansi, dengan prioritas berdasarkan nilai biaya terendah. Setiap node berisi informasi mengenai konfigurasi board, jalur yang telah dilalui, serta total biaya. Algoritma akan terus mengevaluasi dan memperluas node dari priority queue hingga menemukan konfigurasi papan *primary piece* berhasil keluar yang mengakhiri permainan.

UCS memiliki kompleksitas waktu $O(b^{C^*/\epsilon})$, di mana:

- b adalah *branching factor* (jumlah rata-rata gerakan yang mungkin pada setiap langkah)
- C^* adalah biaya solusi optimal
- ϵ adalah biaya langkah terkecil (dalam konteks Rush Hour, $\epsilon = 1$ karena setiap langkah memiliki biaya 1)

3.2. Greedy Best First Search

Greedy Best First Search adalah sebuah algoritma pencarian berjenis *informed search* yang menggunakan fungsi heuristik $h(n)$ untuk memperkirakan jarak suatu *state* ke *goal state*. Tujuan utama dari algoritma ini adalah untuk mengejar solusi secepat mungkin berdasarkan estimasi paling dekat terhadap *goal*, meskipun tidak menjamin solusi yang ditemukan adalah yang paling optimal secara *cost*. Sesuai dengan namanya, algoritma ini bersifat *greedy* karena selalu memilih node yang menurut fungsi heuristik paling dekat dengan tujuan, tanpa mempertimbangkan biaya yang telah dikeluarkan untuk mencapai node tersebut (tidak mempertimbangkan $g(n)$).

Algoritma *Greedy Best First Search* bekerja dengan memanfaatkan fungsi heuristik $h(n)$ untuk menentukan node mana yang paling menjanjikan untuk diekspansi. Proses dimulai dengan menginisialisasi sebuah *priority queue* yang berisi node awal (konfigurasi papan awal). Selama antrian tersebut tidak kosong, algoritma akan terus mengambil node dengan nilai heuristik terkecil dari antrian. Jika node tersebut merupakan solusi, maka algoritma akan segera mengembalikan jalur yang membentuk solusi tersebut. Jika bukan, node tersebut akan ditandai sebagai telah dikunjungi dengan memasukkannya ke dalam *closed set*.

Selanjutnya, algoritma akan melakukan ekspansi terhadap node tersebut untuk menghasilkan semua kemungkinan konfigurasi sah berikutnya (*child nodes*). Untuk setiap konfigurasi baru yang dihasilkan, algoritma akan memeriksa apakah konfigurasi tersebut telah dikunjungi sebelumnya. Jika belum, algoritma akan menghitung nilai heuristiknya dan menambahkannya ke dalam *priority queue*. Algoritma ini secara keseluruhan hanya menggunakan informasi dari fungsi heuristik untuk memilih node yang akan diperluas, tanpa memperhatikan biaya akumulatif perjalanan dari node awal, sehingga tidak dapat menjamin solusi yang ditemukan merupakan solusi optimal.

Berikut adalah kode pengimplementasian *Greedy Best First Search* yang telah dibuat dalam konteks permainan *Rush Hour*:

```
csans13, yesterday | 1 author (csans13)
import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Set;

csans13, yesterday | 1 author (csans13)
public class GreedyBestFirstSearch extends Solver {

    public GreedyBestFirstSearch(Board initialBoard, int heuristicChoice) {
        super(initialBoard, heuristicChoice);
    }

    @Override
    public Solution solve() {
        // Antrian prioritas berdasarkan heuristic terendah
        PriorityQueue<Node> openSet = new PriorityQueue<>(Comparator.comparingInt(Node::getHeuristic));
        Set<Board> closedSet = new HashSet<>(); // Jika sudah dikunjungi

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

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

            if (current.getBoard().isSolved()) {
                return new Solution(current.getPath());
            }
            closedSet.add(current.getBoard());

            List<int[]> legalMoves = current.getBoard().getLegalMoves();
            for (int[] move : legalMoves) {
                int vehicleIdx = move[0];
                int moveAmount = move[1];

                Board newBoard = current.getBoard().applyMove(vehicleIdx, moveAmount);

                if (closedSet.contains(newBoard)) {
                    continue;
                }

                Node successor = new Node(current, newBoard, vehicleIdx, moveAmount, heuristicChoice);

                boolean alreadyInOpenSet = false;
                for (Node openNode : openSet) {
                    if (openNode.getBoard().equals(newBoard)) {
                        alreadyInOpenSet = true;
                        break;
                    }
                }

                if (!alreadyInOpenSet) {
                    openSet.add(successor);
                }
            }
        }
        return null; // Jika tidak ditemukan solusi
    }

    @Override
    public String getName() {
        return "Greedy Best First Search";
    }
}
```

Algoritma Greedy Best First Search diimplementasikan melalui kelas `GreedyBestFirstSearch`, yang merupakan turunan dari kelas abstrak `Solver`. Implementasi algoritma ini memanfaatkan struktur data *priority queue* yang diatur berdasarkan nilai heuristik terkecil, sehingga node yang dianggap paling dekat dengan solusi akan diekspansi terlebih dahulu.

- `getLegalMoves()` digunakan pada program untuk memperoleh semua pergerakan valid
- `applyMove()` digunakan pada program untuk menerapkan pergerakan tersebut ke papan permainan.
- `closedSet` digunakan pada program untuk menghindari eksplorasi konfigurasi yang sama secara berulang dengan menyimpan semua konfigurasi papan yang telah dikunjungi.

Greedy Best First Search memiliki kompleksitas waktu $O(b^m)$, di mana:

- b adalah *branching factor* (jumlah rata-rata gerakan yang mungkin pada setiap langkah)
- m adalah kedalam maksimal pencarian

Efektivitas algoritma ini sangat bergantung pada kualitas fungsi heuristik. Jika heuristik tidak sesuai, maka algoritma dapat mengeksplorasi jalur tidak efisien atau bahkan gagal menemukan solusi yang seharusnya dapat diperoleh.

3.3. A*

A* adalah algoritma pencarian berjenis *informed search* yang menggabungkan keunggulan antara *Uniform Cost Search* (UCS) dan *Greedy Best First Search*. A* menggunakan fungsi evaluasi, yaitu

$$f(n) = g(n) + h(n) , \text{di mana:}$$

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

Dengan menggabungkan kedua nilai ini, algoritma A* dapat bekerja dengan efektif karena mempertimbangkan biaya yang sudah dikeluarkan dan perkiraan biaya yang masih diperlukan untuk mencapai tujuan. Algoritma ini menjamin solusi optimal jika heuristik

memiliki sifat *admissible*, yaitu tidak melebih-lebihkan biaya sebenarnya dari node n ke tujuan, serta konsisten.

Proses kerja A* dilakukan dengan melibatkan pemilihan node dengan nilai $f(n)$ terkecil pada setiap langkah, kemudian mengekspansi node tersebut dengan menghasilkan semua tetangga yang valid. Algoritma ini dimulai dengan menginisialisasi *priority queue* menggunakan node awal sebagai titik mulai pencarian. Selama *priority queue* tidak kosong, algoritma akan terus mengambil node dengan nilai $f(n)$ terkecil untuk diproses. Jika node yang diambil tersebut merupakan tujuan yang dicari, maka algoritma akan segera mengembalikan solusi yang ditemukan. Selanjutnya, node tersebut ditandai sebagai sudah dikunjungi untuk menghindari pengecekan ulang. Algoritma kemudian melakukan ekspansi terhadap node tersebut untuk menghasilkan semua node tetangga yang valid. Setiap node tetangga yang diperoleh akan dihitung nilai $g(n)$ -nya, yaitu biaya dari node awal menuju node tetangga melalui node saat ini. Jika node tetangga tersebut belum pernah dikunjungi sebelumnya, atau nilai $g(n)$ yang baru lebih kecil dibandingkan nilai sebelumnya, maka nilai $g(n)$ dan $f(n)$ akan diperbarui. Setelah itu, node tetangga ditambahkan atau diperbarui di dalam *priority queue*. Dengan metode ini, A* dapat secara efisien mencari jalur terbaik menuju tujuan dengan mempertimbangkan biaya aktual dan estimasi biaya yang masih diperlukan.

Berikut adalah kode pengimplementasian A* yang telah dibuat dalam konteks permainan *Rush Hour*:

```
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Set;

csans13, yesterday | 1 author (csans13)
public class AStarSearch extends Solver {

    public AStarSearch(Board initialBoard, int heuristicChoice) {
        super(initialBoard, heuristicChoice);
    }

    @Override
    public Solution solve() {
        // Antrian prioritas berdasarkan  $f(n) = g(n) + h(n)$  terendah
        PriorityQueue<Node> openSet = new PriorityQueue<>();
        Set<Board> closedSet = new HashSet<>(); // Jika sudah dikunjungi

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

        while (!openSet.isEmpty()) { csans13, yesterday * feat: add GreedyBestFirstSearch and AStarSearch
            Node current = openSet.poll();
            nodesVisited++;

            if (current.getBoard().isSolved()) {
                return new Solution(current.getPath());
            }
            closedSet.add(current.getBoard());
        }
    }
}
```

```

        List<int[]> legalMoves = current.getBoard().getLegalMoves();
        for (int[] move : legalMoves) {
            int vehicleIdx = move[0];
            int moveAmount = move[1];

            Board newBoard = current.getBoard().applyMove(vehicleIdx, moveAmount);
            if (closedSet.contains(newBoard)) {
                continue;
            }

            Node successor = new Node(current, newBoard, vehicleIdx, moveAmount, heuristicChoice);

            boolean foundBetter = false;
            for (Node openNode : openSet) {
                if (openNode.getBoard().equals(newBoard) && openNode.getCost() <= successor.getCost()) {
                    foundBetter = true;
                    break;
                }
            }

            if (!foundBetter) {
                openSet.add(successor);
            }
        }
    }
    return null; // Jika tidak ditemukan solusi
}

@Override
public String getName() {
    return "A* Search";
}
}

```

- Algoritma menggunakan *priority queue* (`openSet`) yang menyimpan node berdasarkan nilai $f(n) = g(n)+h(n)$ terkecil
- Node awal dimasukkan ke dalam fungsi `openSet`.
- Selama `openSet` tidak kosong, maka algoritma akan mengambil node `current` dengan nilai terendah dari `openSet`. Jika node `current` sudah mencapai tujuan, maka jalur solusi dikembalikan. Node `current` akan kemudian ditandai sebagai sudah dikunjungi dengan memasukkannya ke `closedSet`. Untuk setiap gerakan, membentuk konfigurasi papan baru (`newBoard`). Jika `newBoard` sudah ada di `closedSet`, maka gerakan tersebut dilewati. Jika belum, membuat node `successor` dengan status `newBoard`.
- Kemudian, program akan memeriksa apakah `openSet` sudah ada node dengan papan yang sama dan biaya lebih baik.

- Jika tidak ada node lebih baik, maka program akan menambahkan successor ke `openSet` dan proses berulang sampai solusi ditemukan atau `openSet` kosong (solusi tidak ada).

3.4. Beam Search

Beam Search adalah sebuah algoritma pencarian heuristik yang merupakan varian Best-First Search dengan pembatasan memori. Algoritma ini hanya menyimpan k (beam width) node terbaik pada setiap level pencarian. Algoritma ini sangat berguna dalam permasalahan dengan ruang state yang sangat besar, karena dapat menghemat memori dan waktu dengan mengorbankan jaminan solusi optimal.

Algoritma *Beam Search* bekerja dimulai dengan node awal. Kemudian, pada setiap level semua node akan diekspansi untuk menghasilkan node anak. Node anak ini kemudian dievaluasi menggunakan fungsi heuristik, dan hanya k node terbaik dengan nilai heuristik terkecil akan dipilih untuk melanjutkan ke level berikutnya, sisanya akan dibuang. Proses ini berlanjut sampai solusi ditemukan atau tidak ada node yang tersisa.

Berikut adalah kode pengimplementasian *Beam Search* yang telah dibuat dalam konteks permainan *Rush Hour*:

```

import java.util.ArrayList;
import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Set;

You, 2 days ago | 1 author (You)
public class BeamSearch extends Solver {
    private int beamWidth;

    public BeamSearch(Board initialBoard, int heuristicChoice, int beamWidth) {
        super(initialBoard, heuristicChoice);
        this.beamWidth = beamWidth;
    }
}

```

```

@Override
public Solution solve() {
    // Menyimpan papan yang sudah dikunjungi agar tidak duplicate
    Set<Board> closedSet = new HashSet<>();

    // Pencarian dari level pertama
    List<Node> currentLevel = new ArrayList<>();
    currentLevel.add(new Node(initialBoard, heuristicChoice));

    while (!currentLevel.isEmpty()) {
        PriorityQueue<Node> nextLevel = new PriorityQueue<>(Comparator.comparingInt(Node::getHeuristic));

        // Iterasi seluruh node di level ini
        for (Node current : currentLevel) {
            nodesVisited++;

            // Jika sudah mencapai solusi, kembalikan path-nya
            if (current.getBoard().isSolved()) {
                return new Solution(current.getPath());
            }

            closedSet.add(current.getBoard());

            // Hasilkan semua possible moves
            List<int[]> legalMoves = current.getBoard().getLegalMoves();
            for (int[] move : legalMoves) {
                int vehicleIdx = move[0];
                int moveAmount = move[1];

                Board newBoard = current.getBoard().applyMove(vehicleIdx, moveAmount);

                // Jika papan baru sudah pernah dikunjungi, jangan diproses lagi
                if (closedSet.contains(newBoard)) continue;

                Node successor = new Node(current, newBoard, vehicleIdx, moveAmount, heuristicChoice);
                nextLevel.add(successor);
            }
        }

        // Seleksi node terbaik untuk level berikutnya
        currentLevel = new ArrayList<>();
        for (int i = 0; i < beamWidth && !nextLevel.isEmpty(); i++) {
            currentLevel.add(nextLevel.poll());
        }
    }

    // Jika tidak ditemukan solusi, kembalikan null
    return null;
}

@Override
public String getName() {
    return "Beam Search (width=" + beamWidth + ")";
}

```

- Program dimulai dari node awal yang disimpan di dalam `currentLevel`.

- Kemudian, `closedSet` akan dipanggil untuk menyimpan konfigurasi papan yang sudah pernah dikunjungi agar tidak diproses ulang.
- Setiap iterasi level pencarian, akan memanggil `nextLevel` sebagai `PriorityQueue` yang mengurutkan node berdasarkan nilai heuristik terkecil. Kemudian, semua node pada `currentLevel` akan diekspansi satu per satu. Untuk setiap node, algoritma akan memeriksa apakah kondisi solusi telah tercapai. Jika belum, algoritma menghasilkan semua gerakan legal yang memungkinkan dari node tersebut dan membentuk konfigurasi papan baru (`newBoard`) untuk setiap gerakan. Jika konfigurasi papan baru ini belum pernah dikunjungi sebelumnya, maka node penerus akan dibuat dan dimasukkan ke dalam `nextLevel`.
- Setelah seluruh node pada `currentLevel` diekspansi, algoritma hanya memilih sejumlah node terbaik sesuai dengan nilai heuristik, sebanyak `beamWidth`, dari `nextLevel` untuk menjadi `currentLevel` pada iterasi berikutnya.
- Proses ini akan berulang terus hingga solusi ditemukan atau tidak ada lagi node yang tersisa untuk dieksplorasi. Jika solusi tidak ditemukan, algoritma mengembalikan nilai null.

Beam Search memiliki kompleksitas waktu $O(b \times k \times d)$, di mana:

- b adalah *branch factor* (jumlah gerakan rata-rata)
- k adalah *beam width* (jumlah node dipertahankan tiap level)
- d adalah kedalaman solusi

Algoritma ini memiliki keunggulan utama, yaitu penggunaan memori yang terbatas karena hanya mempertahankan sejumlah node sesuai dengan beam width. Sehingga, sangat cocok digunakan pada ruang pencarian yang besar karena mampu menyingkat eksplorasi ke node-node yang dianggap paling menjanjikan berdasarkan heuristik. Namun, *Beam Search* juga memiliki kelemahan, yaitu tidak menjamin akan menemukan solusi yang optimal. Selain itu, dikarenakan algoritma ini bersifat tidak lengkap, maka algoritma ini bisa saja gagal menemukan solusi walaupun sebenarnya solusi itu ada. Kualitas solusi yang diperoleh sangat bergantung pada pemilihan fungsi heuristik serta nilai *beamwidth* yang digunakan.

3.5. Analisis Algoritma Pathfinding

3.5.1. Definisi $f(n)$ dan $g(n)$

Dalam konteks algoritma pencarian terinformasi (*informed search*), $g(n)$ dan $f(n)$ memiliki pengertian sebagai berikut:

1. $g(n)$: adalah biaya perjalanan sebenarnya dari node awal ke node n. Dalam kasus *Rush Hour*, $g(n)$ merepresentasikan jumlah langkah pergeseran *piece* yang telah dilakukan untuk mencapai konfigurasi papan saat ini dari konfigurasi papan awal.
2. $f(n)$: adalah perkiraan total biaya dari node awal ke tujuan melalui node n.

Penggunaan kedua fungsi ini berbeda berdasarkan algoritma yang digunakan, yaitu:

1. Pada algoritma Uniform Cost Search (UCS), nilai $f(n)$ didefinisikan sebagai $f(n) = g(n)$. UCS memprioritaskan eksplorasi node dengan biaya perjalanan terendah, tanpa mempertimbangkan estimasi jarak ke tujuan. Algoritma ini menjamin menemukan solusi optimal dengan cara sistematis mengeksplorasi jalur-jalur berdasarkan urutan biaya terendah.
2. Pada algoritma Greedy Best First Search, nilai $f(n)$ didefinisikan sebagai $f(n) = h(n)$, dimana $h(n)$ adalah fungsi heuristik yang mengestimasi biaya dari node n ke node tujuan. Algoritma ini sepenuhnya mengabaikan biaya yang telah dikeluarkan ($g(n)$) dan hanya fokus pada proyeksi jarak menuju tujuan.
3. Pada algoritma A*, nilai $f(n)$ didefinisikan sebagai $f(n) = g(n) + h(n)$, yaitu jumlah biaya aktual dari awal ke n ditambah estimasi biaya dari n ke goal. A* menggunakan keseimbangan antara langkah sejauh ini dan estimasi langkah menuju goal untuk menentukan node yang akan dieksplorasi selanjutnya.
4. Pada algoritma Beam Search, fungsi evaluasi biasanya menggunakan $f(n) = g(n) + h(n)$ seperti pada A*, namun dengan batasan penting, yaitu algoritma ini hanya mempertahankan k node terbaik pada setiap level kedalaman pencarian.

3.5.2. Admissibility Heuristik pada algoritma A*

Sebuah fungsi heuristik $h(n)$ dikatakan *admissible* jika $h(n) \leq h^*(n)$ untuk semua node n, di mana $h^*(n)$ adalah biaya sebenarnya dari node n ke tujuan. Sehingga, dapat disimpulkan bahwa fungsi heuristik *admissible* tidak pernah overestimasi biaya sebenarnya untuk mencapai

tujuan dan selalu memberikan nilai yang lebih kecil atau sama dengan biaya sebenarnya. Untuk *Rush Hour puzzle*, beberapa heuristik yang bisa digunakan adalah:

1. Distance to Exit: mengukur Jarak Manhattan (umlah langkah minimal) dari *primary piece* ke pintu keluar.
2. Blocking Pieces: menghitung jumlah kendaraan yang menghalangi *primary piece* menuju pintu keluar.

Heuristik *Distance to Exit* bersifat admissible karena mengukur jarak terpendek yang secara teoritis harus ditempuh oleh *primary piece* untuk mencapai pintu keluar, yaitu menggunakan jarak Manhattan. Dalam konteks permainan Rush Hour, *primary piece* hanya dapat bergerak satu langkah dalam satu gerakan dan tidak bisa melompati kendaraan lain. Oleh karena itu, jarak horizontal dari ujung *primary piece* ke pintu keluar memberikan batas bawah dari jumlah gerakan yang diperlukan, tanpa memperhitungkan hambatan dari kendaraan lain. Dikarenakan tipe heuristik ini tidak pernah melebih-lebihkan jumlah langkah minimum yang sebenarnya dibutuhkan, maka ia memenuhi syarat *admissibility*.

Sementara itu, heuristik *Blocking Pieces* juga dapat disebut *admissible* dengan asumsi bahwa setiap kendaraan (*piece*) yang menghalangi jalur primary piece ke pintu keluar memerlukan minimal satu gerakan untuk keluar dari jalur. Artinya, jumlah kendaraan penghalang memberikan estimasi konservatif terhadap jumlah aksi yang diperlukan. Meskipun dalam kenyataannya beberapa kendaraan mungkin memerlukan lebih dari satu langkah, atau bahkan pergeseran kendaraan lain untuk dapat dipindahkan, asumsi satu langkah per penghalang tetap membuat heuristik ini tidak melebih-lebihkan biaya aktual ke goal. Sehingga, masih dapat dikategorikan *admissible* jika syarat terpenuhi.

3.5.3. UCS vs BFS pada Rush Hour

Breadth-First Search (BFS) adalah algoritma pencarian yang bekerja dengan menjelajahi semua node pada kedalaman tertentu sebelum beralih ke node pada kedalaman berikutnya. Algoritma ini menggunakan antrian FIFO (*First-In-First-Out*) untuk menyimpan node yang akan dieksplorasi, menjamin menemukan solusi dengan jumlah langkah minimum. Sedangkan, Uniform Cost Search (UCS) adalah algoritma pencarian yang menjelajahi node berdasarkan biaya perjalanan dari node awal. Node dengan biaya terendah akan dieksplorasi terlebih dahulu.

Pada penyelesaian *Rush Hour*, algoritma UCS dan BFS akan menghasilkan urutan node yang dibangkitkan dan path yang sama jika semua gerakan memiliki biaya yang sama. Hal ini sesuai karena dalam *Rush Hour*, setiap gerakan menggeser *piece* dianggap memiliki biaya yang sama, yaitu 1. Dengan biaya yang sama persis untuk setiap gerakan, UCS akan memprioritaskan

node berdasarkan jumlah total gerakan untuk mencapai node tersebut ($g(n)$). Node dengan jumlah gerakan lebih sedikit akan selalu diproses terlebih dahulu, sama seperti perilaku BFS yang selalu mengeksplorasi node berdasarkan kedalaman. Dikarenakan $g(n)$ berbanding lurus dengan kedalaman node, UCS akan mengeksplorasi node dalam urutan yang sama persis seperti BFS.

Namun, jika permainan *Rush Hour* dimodifikasi di mana biaya gerakan berbeda-beda, seperti berdasarkan panjang kendaraan, maka UCS dan BFS akan menghasilkan urutan eksplorasi node yang berbeda dan berkemungkinan menghasilkan jalur solusi yang berbeda pula.

3.5.4. Efisiensi A* vs UCS pada Rush Hour

Secara teoritis, algoritma A* akan lebih efisien dibandingkan UCS pada penyelesaian *Rush Hour* dengan syarat bahwa fungsi heuristik yang digunakan harus bersifat *admissible* dan *consistent*. Hal ini dikarenakan algoritma UCS hanya mempertimbangkan $g(n)$, sehingga algoritma ini mengeksplorasi semua node dengan *cost minimum* terlebih dahulu tanpa arah yang jelas menuju tujuan. Sedangkan, algoritma A* menggunakan informasi heuristik dan mempertimbangkan $g(n) + h(n)$, sehingga cenderung mengeksplorasi lebih sedikit node untuk mencapai solusi. Jika $h(n)$ *admissible* dan *consistent*, A* dipastikan akan mengeksplorasi lebih sedikit node dibanding UCS dan bersifat optimal. Dalam konteks *Rush Hour*, heuristik tertentu seperti jarak ke pintu keluar atau jumlah kendaraan penghalang dapat memberikan informasi berguna untuk mengarahkan pencarian A*. Efisiensi A* dibandingkan UCS semakin meningkat seiring dengan semakin informatifnya heuristik, yaitu semakin mendekati biaya sebenarnya. Selain itu, juga akan menjadi lebih efisien berdasarkan semakin kompleksnya permasalahan (lebih banyak state yang mungkin).

3.5.5. Optimalitas Greedy Best First Search

Secara teoritis, algoritma Greedy Best First Search tidak menjamin solusi optimal untuk penyelesaian *Rush Hour*. Hal ini dikarenakan beberapa alasan, yaitu:

1. Greedy Best First Search hanya mempertimbangkan nilai heuristik $h(n)$ dan mengabaikan biaya sebenarnya $g(n)$ yang telah ditempuh.
2. Algoritma ini selalu memilih node yang tampaknya paling dekat dengan tujuan berdasarkan heuristik, tanpa mempertimbangkan berapa banyak langkah yang telah diambil.

3. Pendekatan algoritma ini bersifat "serakah" dan hanya fokus secara eksklusif pada heuristik, sehingga dapat menyebabkan algoritma terjebak dalam jalur suboptimal jika heuristik tidak sempurna atau terjebak dalam jalur yang awalnya tampak menjanjikan tetapi akhirnya membutuhkan banyak langkah untuk mencapai tujuan.

Dalam konteks *Rush Hour*, Greedy Best First Search mungkin memilih gerakan yang mengurangi jumlah kendaraan penghalang atau membawa *primary piece* lebih dekat ke pintu keluar, tetapi tidak mempertimbangkan total jumlah langkah yang diperlukan. Akibatnya, meskipun solusi ditemukan, jumlah langkah yang dihasilkan kemungkinan tidak minimal.

BAB IV SOURCE PROGRAM

Program pada tucil 3 ini ditulis dalam bahasa pemrograman Java dan mengimplementasikan berbagai algoritma pencarian untuk menyelesaikan permainan *Rush Hour Puzzle*, yaitu varian permainan papan yang menantang pemain untuk mengeluarkan *primary piece* dari papan dengan cara memindahkan kendaraan lain yang menghalangi jalan keluar. Program ini mengimplementasikan empat jenis algoritma, yaitu UCS, Greedy Best First Search, A* dan Beam Search. Selain itu, program yang dibuat juga mengimplementasikan dua pilihan heuristik, yaitu Jarak Manhattan ke Exit dan Blocking Pieces.

Program yang dibuat memiliki struktur direktori sebagai berikut:

```
Tucil3_13523157_13523159/ (Root Directory)
|
└── src/
    ├── AStarSearch.java
    ├── BeamSearch.java
    ├── Board.java
    ├── GreedyBestFirstSearch.java
    ├── Main.java
    ├── Node.java
    ├── Piece.java
    ├── Solution.java
    ├── Solver.java
    └── UniformCostSearch.java

    └── test/
        ├── testcase1.txt
        ├── testcase2.txt
        ├── testcase3.txt
        ├── testcase4.txt
        ├── testcase5.txt
        ├── testcase6.txt
        ├── testcase7.txt
        └── testcase8.txt

    └── bin/

    └── doc/
        └── Tucil3_K3_13523157_13523159.pdf
```

```
└── README.md
```

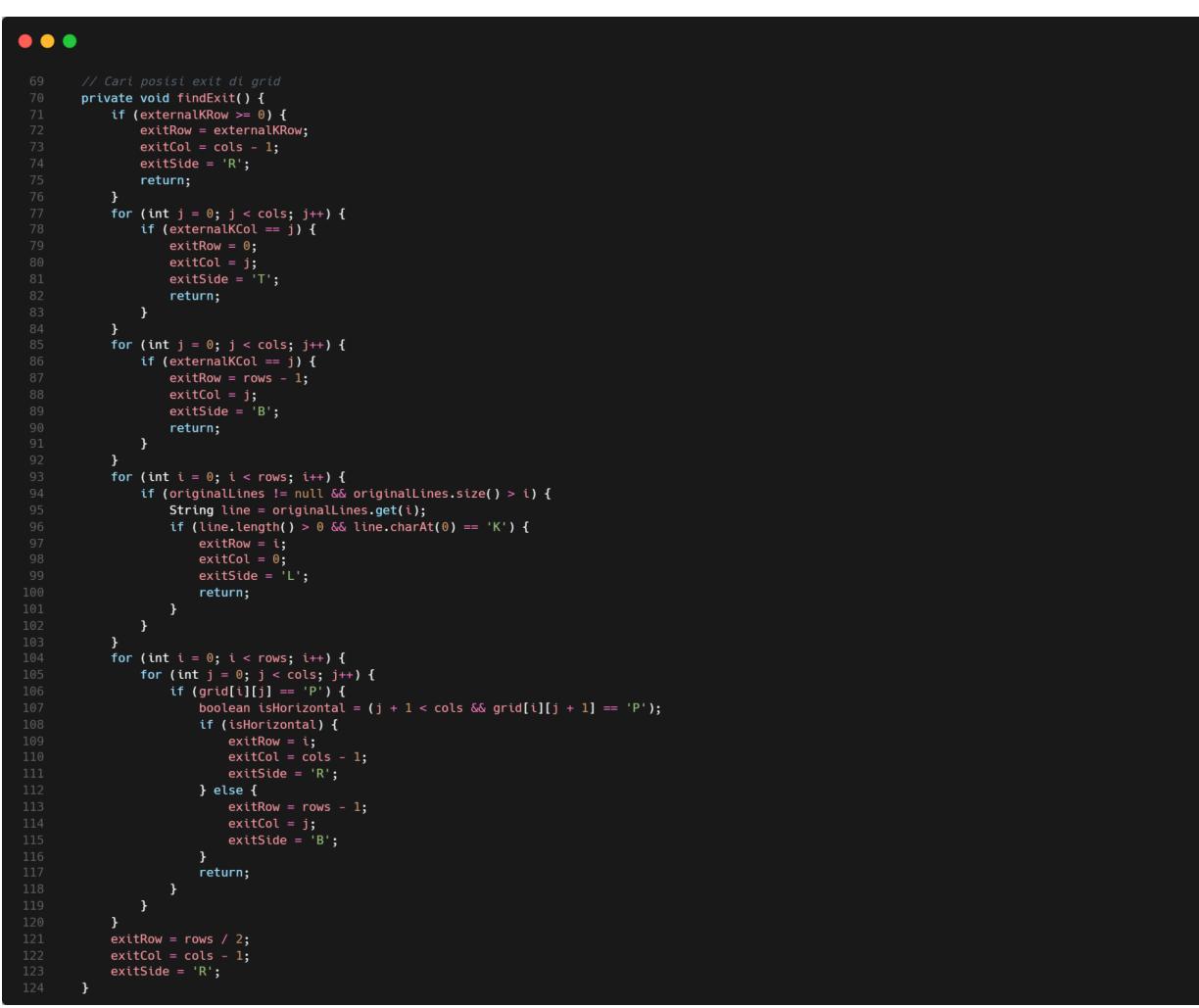
4.1. Board

```
 1 import java.util.ArrayList;
 2 import java.util.HashMap;
 3 import java.util.List;
 4 import java.util.Map;
 5
 6 public class Board {
 7     private int rows;
 8     private int cols;
 9     private List<Piece> pieces;
10     private char[][] grid;
11     private int exitRow;
12     private int exitCol;
13     private char exitSide; // 'T' untuk atas, 'R' untuk kanan, 'B' untuk bawah, 'L' untuk kiri
14     private Piece primaryPiece;
15     private List<String> originalLines;
16     private int externalKRow = -1;
17     private int externalKCol = -1;
18
19     // Constructor
20     public Board(char[][] grid, int rows, int cols) {
21         this.grid = new char[rows][cols];
22         this.rows = rows;
23         this.cols = cols;
24         this.originalLines = new ArrayList<>();
25         this.externalKRow = -1;
26         this.externalKCol = -1;
27         this.exitSide = 'R';
28         for (int i = 0; i < rows; i++) {
29             for (int j = 0; j < cols; j++) {
30                 this.grid[i][j] = grid[i][j];
31             }
32         }
33         pieces = new ArrayList<>();
34         findExit();
35         extractPieces();
36     }
37
38     // Buat copy dari board
39     public Board copy() {
40         char[][] newGrid = new char[rows][cols];
41         for (int i = 0; i < rows; i++) {
42             for (int j = 0; j < cols; j++) {
43                 newGrid[i][j] = grid[i][j];
44             }
45         }
46         Board newBoard = new Board(newGrid, rows, cols);
47         newBoard.pieces = new ArrayList<>();
48         for (Piece piece : pieces) {
49             newBoard.pieces.add(piece.copy());
50         }
51         for (Piece piece : newBoard.pieces) {
52             if (piece.isPrimary()) {
53                 newBoard.primaryPiece = piece;
54                 break;
55             }
56         }
57         newBoard.exitRow = this.exitRow;
58         newBoard.exitCol = this.exitCol;
59         newBoard.exitSide = this.exitSide;
60         newBoard.externalKRow = this.externalKRow;
61         newBoard.externalKCol = this.externalKCol;
62         newBoard.originalLines = this.originalLines;
63         if (pieces.size() != newBoard.pieces.size()) {
64             throw new IllegalStateException("Piece list size mismatch after copy: original=" + pieces.size() + ", copy=" + newBoard.pieces.size());
65         }
66         return newBoard;
67     }
}
```

Konstruktor *Board(char[][] grid, int rows, int cols)* bertugas untuk menginisialisasi objek Board, yaitu papan permainan Rush Hour. Parameter yang diterima adalah *grid* yang merupakan matriks karakter berisi representasi papan, serta *rows* dan *cols* yang berturut-turut merupakan

jumlah baris dan kolom papan. Di dalam konstruktor, isi *grid* disalin satu per satu ke atribut *this.grid*, lalu list pieces diinisialisasi sebagai array kosong.

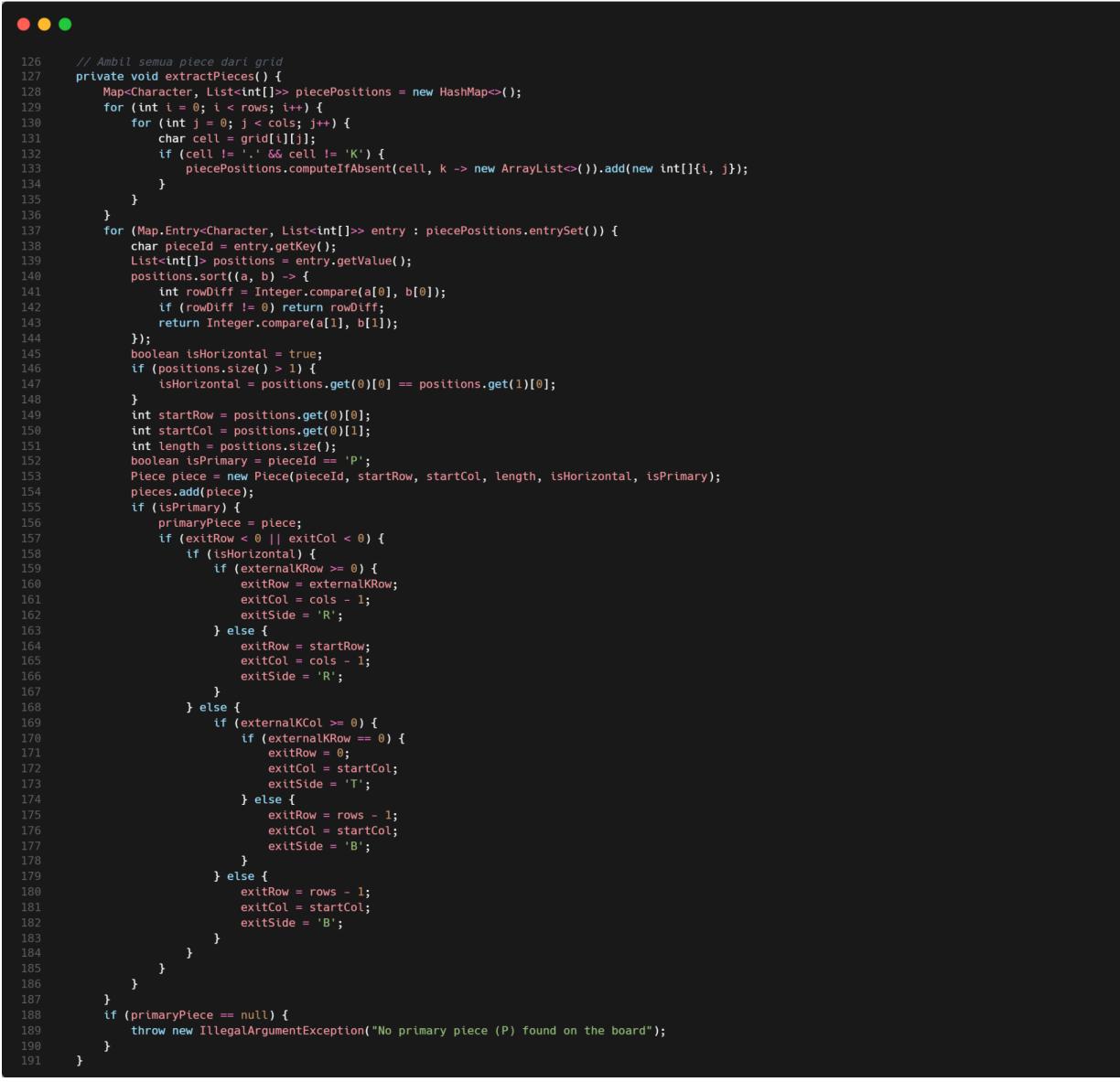
Method *copy()* berfungsi untuk membuat salinan papan permainan secara menyeluruh sehingga perubahan pada salinan tidak memengaruhi papan asli. Pertama-tama, isi *grid* disalin ke dalam *newGrid*. Kemudian, objek *Board* baru dibuat dengan *grid* tersebut. Daftar *pieces* disalin satu per satu dengan memanggil method *copy()* dari masing-masing objek *Piece*. Berikutnya adalah mencari dan menetapkan *primaryPiece*, serta menyalin nilai-nilai lain seperti *exitRow*, *exitCol*, *exitSide*, *externalKRow*, *externalKCol*, dan *originalLines*. Sebagai verifikasi, method ini memeriksa apakah jumlah *pieces* pada salinan sama dengan yang asli.



```
69 // Cari posisi exit di grid
70 private void findExit() {
71     if (externalKRow >= 0) {
72         exitRow = externalKRow;
73         exitCol = cols - 1;
74         exitSide = 'R';
75         return;
76     }
77     for (int j = 0; j < cols; j++) {
78         if (externalKCol == j) {
79             exitRow = 0;
80             exitCol = j;
81             exitSide = 'T';
82             return;
83         }
84     }
85     for (int j = 0; j < cols; j++) {
86         if (externalKCol == j) {
87             exitRow = rows - 1;
88             exitCol = j;
89             exitSide = 'B';
90             return;
91         }
92     }
93     for (int i = 0; i < rows; i++) {
94         if (originalLines != null && originalLines.size() > i) {
95             String line = originalLines.get(i);
96             if (line.length() > 0 && line.charAt(0) == 'K') {
97                 exitRow = i;
98                 exitCol = 0;
99                 exitSide = 'L';
100                return;
101            }
102        }
103    }
104    for (int i = 0; i < rows; i++) {
105        for (int j = 0; j < cols; j++) {
106            if (grid[i][j] == 'P') {
107                boolean isHorizontal = (j + 1 < cols && grid[i][j + 1] == 'P');
108                if (isHorizontal) {
109                    exitRow = i;
110                    exitCol = cols - 1;
111                    exitSide = 'R';
112                } else {
113                    exitRow = rows - 1;
114                    exitCol = j;
115                    exitSide = 'B';
116                }
117                return;
118            }
119        }
120    }
121    exitRow = rows / 2;
122    exitCol = cols - 1;
123    exitSide = 'R';
124}
```

Method *findExit()* bertugas menentukan lokasi pintu keluar dari papan permainan yang menjadi tujuan utama kendaraan utama. Dimulai dengan pengecekan *externalKRow*. Jika *externalKRow* belum ditentukan, method akan mencari nilai *externalKCol*. Jika *originalLines*

tersedia (berisi representasi asli papan), maka method mencari karakter 'K' di kolom paling kiri sebagai kemungkinan pintu keluar di sisi kiri papan. Jika tidak ditemukan posisi keluar eksplisit, method beralih mencari keberadaan kendaraan utama di grid.



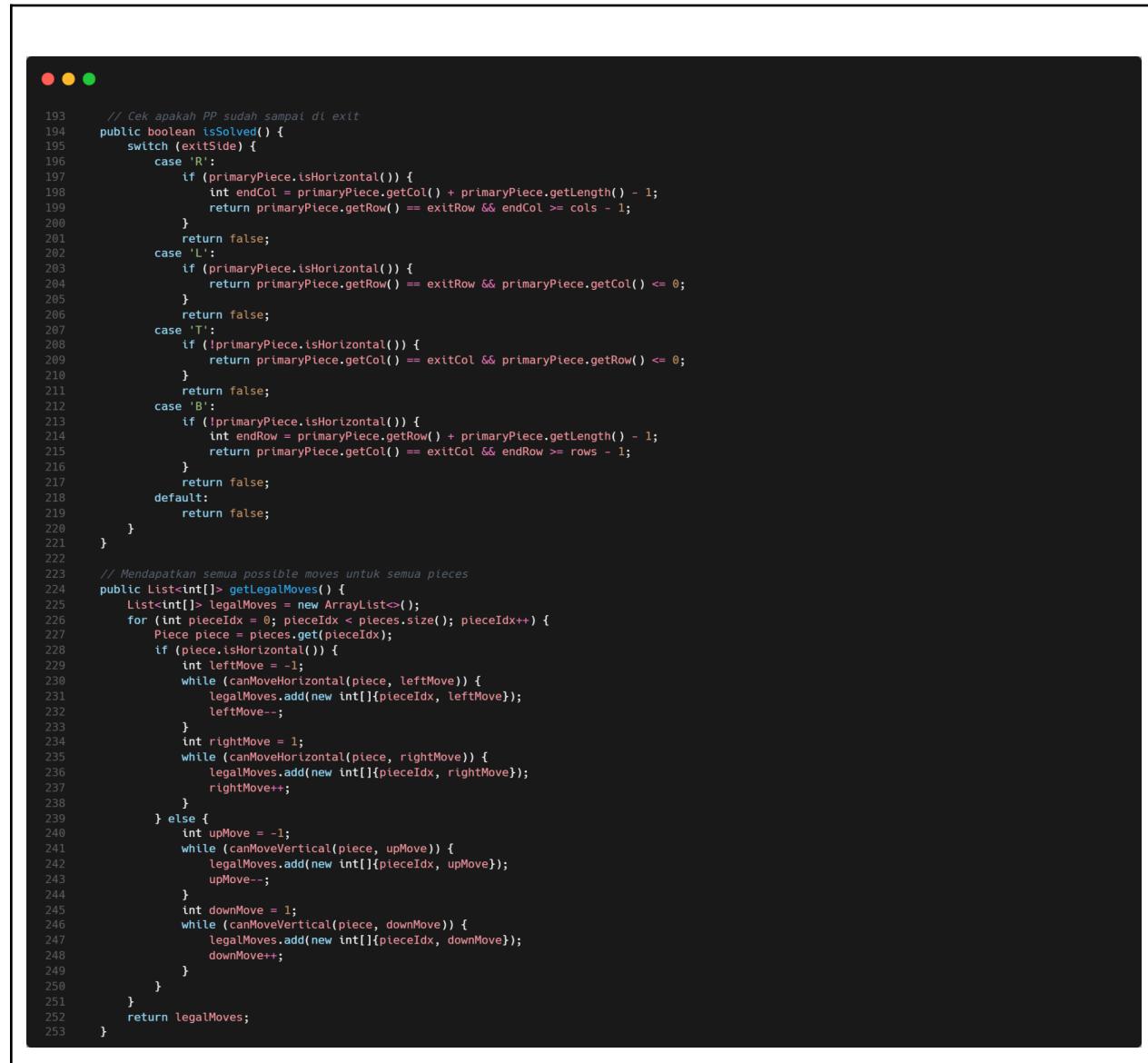
```

126     // Ambil semua piece dari grid
127     private void extractPieces() {
128         Map<Character, List<int[]>> piecePositions = new HashMap<>();
129         for (int i = 0; i < rows; i++) {
130             for (int j = 0; j < cols; j++) {
131                 char cell = grid[i][j];
132                 if (cell != '.' && cell != 'K') {
133                     piecePositions.computeIfAbsent(cell, k -> new ArrayList<>()).add(new int[]{i, j});
134                 }
135             }
136         }
137         for (Map.Entry<Character, List<int[]>> entry : piecePositions.entrySet()) {
138             char pieceId = entry.getKey();
139             List<int[]> positions = entry.getValue();
140             positions.sort((a, b) -> {
141                 int rowDiff = Integer.compare(a[0], b[0]);
142                 if (rowDiff != 0) return rowDiff;
143                 return Integer.compare(a[1], b[1]);
144             });
145             boolean isHorizontal = true;
146             if (positions.size() > 1) {
147                 isHorizontal = positions.get(0)[0] == positions.get(1)[0];
148             }
149             int startRow = positions.get(0)[0];
150             int startCol = positions.get(0)[1];
151             int length = positions.size();
152             boolean isPrimary = pieceId == 'P';
153             Piece piece = new Piece(pieceId, startRow, startCol, length, isHorizontal, isPrimary);
154             pieces.add(piece);
155             if (isPrimary) {
156                 primaryPiece = piece;
157                 if (exitRow < 0 || exitCol < 0) {
158                     if (isHorizontal) {
159                         if (externalKRow >= 0) {
160                             exitRow = externalKRow;
161                             exitCol = cols - 1;
162                             exitSide = 'R';
163                         } else {
164                             exitRow = startRow;
165                             exitCol = cols - 1;
166                             exitSide = 'R';
167                         }
168                     } else {
169                         if (externalKCol >= 0) {
170                             if (externalKRow == 0) {
171                                 exitRow = 0;
172                                 exitCol = startCol;
173                                 exitSide = 'T';
174                             } else {
175                                 exitRow = rows - 1;
176                                 exitCol = startCol;
177                                 exitSide = 'B';
178                             }
179                         } else {
180                             exitRow = rows - 1;
181                             exitCol = startCol;
182                             exitSide = 'B';
183                         }
184                     }
185                 }
186             }
187         }
188         if (primaryPiece == null) {
189             throw new IllegalArgumentException("No primary piece (P) found on the board");
190         }
191     }

```

Method *extractPieces()* berfungsi mendeteksi dan membentuk daftar semua kendaraan dari papan permainan berdasarkan informasi yang terdapat pada *grid*. Kendaraan diidentifikasi sebagai karakter selain '.' dan 'K'. Pertama-tama, method membentuk sebuah *Map<Character, List<int[]>>* bernama *piecePositions* yang menyimpan daftar koordinat untuk setiap karakter kendaraan. Proses ini dilakukan dengan memindai seluruh grid papan. Setelah semua posisi

kendaraan dikumpulkan, method melakukan iterasi atas setiap entri dalam map tersebut. Posisi-posisi kendaraan untuk setiap karakter diurutkan dari atas ke bawah dan kiri ke kanan, lalu arah kendaraan ditentukan berdasarkan apakah dua koordinat pertamanya berada pada baris yang sama (horizontal) atau kolom yang sama (vertikal). Jika kendaraan tersebut adalah kendaraan utama, atribut *primaryPiece* akan diatur ke objek kendaraan tersebut.



```

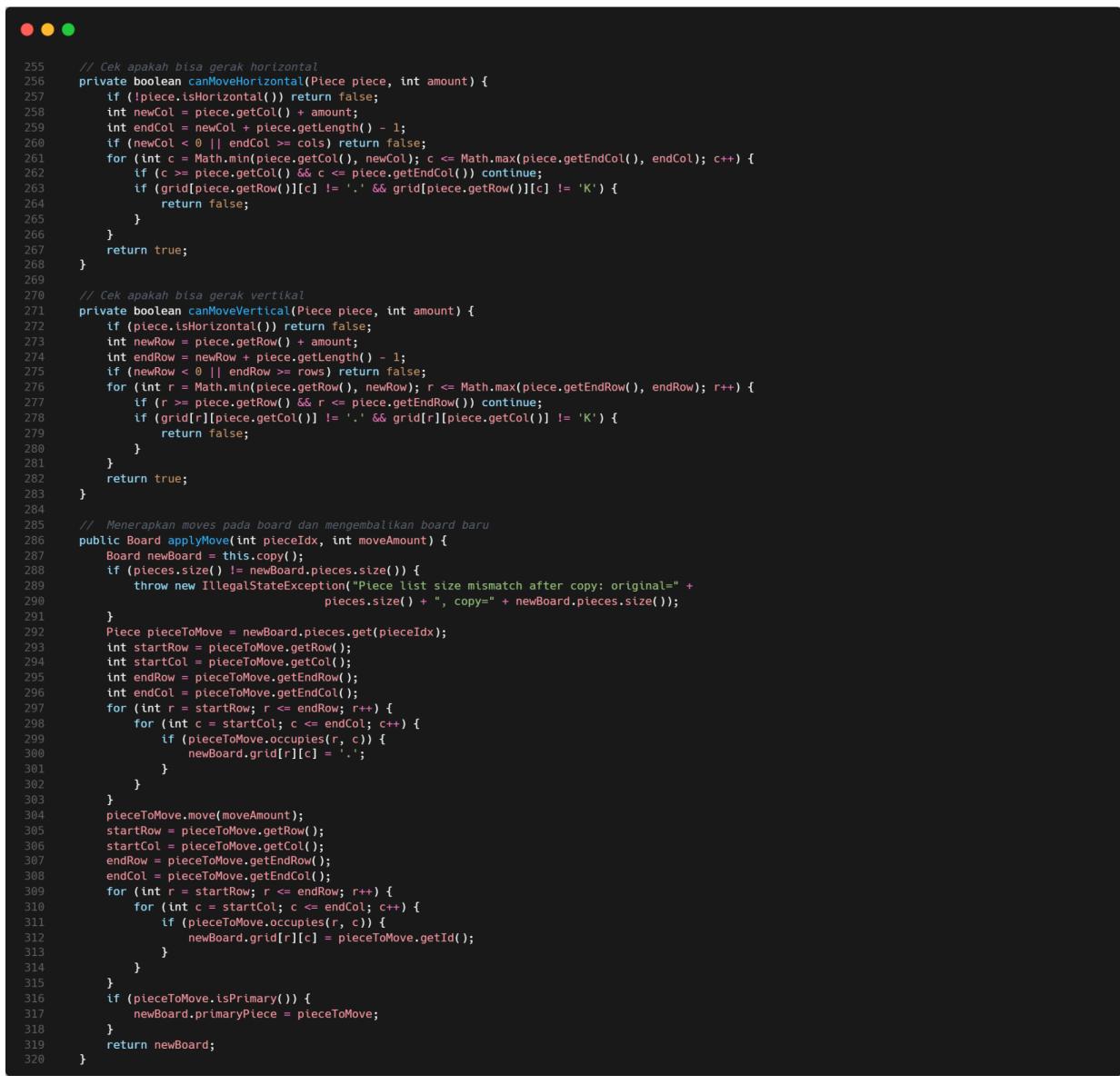
193     // Cek apakah PP sudah sampai di exit
194     public boolean isSolved() {
195         switch (exitSide) {
196             case 'R':
197                 if (primaryPiece.isHorizontal()) {
198                     int endCol = primaryPiece.getCol() + primaryPiece.getLength() - 1;
199                     return primaryPiece.getRow() == exitRow && endCol >= cols - 1;
200                 }
201                 return false;
202             case 'L':
203                 if (primaryPiece.isHorizontal()) {
204                     return primaryPiece.getRow() == exitRow && primaryPiece.getCol() <= 0;
205                 }
206                 return false;
207             case 'T':
208                 if (!primaryPiece.isHorizontal()) {
209                     return primaryPiece.getCol() == exitCol && primaryPiece.getRow() <= 0;
210                 }
211                 return false;
212             case 'B':
213                 if (!primaryPiece.isHorizontal()) {
214                     int endRow = primaryPiece.getRow() + primaryPiece.getLength() - 1;
215                     return primaryPiece.getCol() == exitCol && endRow >= rows - 1;
216                 }
217                 return false;
218             default:
219                 return false;
220         }
221     }
222
223     // Mendapatkan semua possible moves untuk semua pieces
224     public List<int[]> getLegalMoves() {
225         List<int[]> legalMoves = new ArrayList<>();
226         for (int pieceIdx = 0; pieceIdx < pieces.size(); pieceIdx++) {
227             Piece piece = pieces.get(pieceIdx);
228             if (piece.isHorizontal()) {
229                 int leftMove = -1;
230                 while (canMoveHorizontal(piece, leftMove)) {
231                     legalMoves.add(new int[]{pieceIdx, leftMove});
232                     leftMove--;
233                 }
234                 int rightMove = 1;
235                 while (canMoveHorizontal(piece, rightMove)) {
236                     legalMoves.add(new int[]{pieceIdx, rightMove});
237                     rightMove++;
238                 }
239             } else {
240                 int upMove = -1;
241                 while (canMoveVertical(piece, upMove)) {
242                     legalMoves.add(new int[]{pieceIdx, upMove});
243                     upMove--;
244                 }
245                 int downMove = 1;
246                 while (canMoveVertical(piece, downMove)) {
247                     legalMoves.add(new int[]{pieceIdx, downMove});
248                     downMove++;
249                 }
250             }
251         }
252         return legalMoves;
253     }

```

Method *isSolved()* menentukan apakah permainan Rush Hour telah mencapai kondisi selesai, yaitu ketika kendaraan utama telah mencapai posisi keluar. Pemeriksaan dilakukan berdasarkan *exitSide* yang bisa berupa kanan ('R'), kiri ('L'), atas ('T'), atau bawah ('B'). Untuk keluar di kanan, kendaraan utama harus horizontal dan bagian ujung kanannya sudah mencapai atau melebihi kolom terakhir papan. Untuk keluar di kiri, kendaraan juga harus horizontal, tetapi kolom awalnya harus sudah mencapai atau melewati sisi paling kiri. Untuk keluar di atas,

kendaraan harus vertikal dan bagian atasnya sudah di baris pertama. Sedangkan untuk keluar di bawah, kendaraan harus vertikal dan bagian bawahnya sudah menyentuh atau melewati baris terakhir. Jika kondisi-kondisi tersebut tidak terpenuhi sesuai arah keluar, method akan mengembalikan *false*.

Method *getLegalMoves()* menghasilkan daftar semua gerakan sah yang dapat dilakukan oleh semua kendaraan pada papan saat ini. Setiap gerakan diwakili sebagai *array integer [pieceIdx, offset]* dengan *pieceIdx* adalah indeks kendaraan dalam daftar *pieces*, dan *offset* menunjukkan arah serta jumlah langkah pergerakan. Untuk kendaraan horizontal, method mencoba bergerak ke kiri (dengan offset -1, -2, dst.) dan ke kanan (offset +1, +2, dst.) selama sel di tujuan masih kosong. Hal serupa dilakukan untuk kendaraan vertikal, yaitu mencoba bergerak ke atas dan ke bawah.



```

255     // Cek apakah bisa gerak horizontal
256     private boolean canMoveHorizontal(Piece piece, int amount) {
257         if (!piece.isHorizontal()) return false;
258         int newCol = piece.getCol() + amount;
259         int endCol = newCol + piece.getLength() - 1;
260         if (newCol < 0 || endCol >= cols) return false;
261         for (int c = Math.min(piece.getCol(), newCol); c <= Math.max(piece.getEndCol(), endCol); c++) {
262             if (c >= piece.getRow() && c <= piece.getEndRow()) continue;
263             if (grid[piece.getRow()][c] != '.' && grid[piece.getRow()][c] != 'K') {
264                 return false;
265             }
266         }
267         return true;
268     }
269
270     // Cek apakah bisa gerak vertikal
271     private boolean canMoveVertical(Piece piece, int amount) {
272         if (piece.isHorizontal()) return false;
273         int newRow = piece.getRow() + amount;
274         int endRow = newRow + piece.getLength() - 1;
275         if (newRow < 0 || endRow >= rows) return false;
276         for (int r = Math.min(piece.getRow(), newRow); r <= Math.max(piece.getEndRow(), endRow); r++) {
277             if (r >= piece.getRow() && r <= piece.getEndRow()) continue;
278             if (grid[r][piece.getCol()] != '.' && grid[r][piece.getCol()] != 'K') {
279                 return false;
280             }
281         }
282         return true;
283     }
284
285     // Menerapkan moves pada board dan mengembalikan board baru
286     public Board applyMove(int pieceIdx, int moveAmount) {
287         Board newBoard = this.copyOf();
288         if (pieces.size() != newBoard.pieces.size()) {
289             throw new IllegalStateException("Piece list size mismatch after copy: original=" +
290                     pieces.size() + ", copy=" + newBoard.pieces.size());
291         }
292         Piece pieceToMove = newBoard.pieces.get(pieceIdx);
293         int startRow = pieceToMove.getRow();
294         int startCol = pieceToMove.getCol();
295         int endRow = pieceToMove.getEndRow();
296         int endCol = pieceToMove.getEndCol();
297         for (int r = startRow; r <= endRow; r++) {
298             for (int c = startCol; c <= endCol; c++) {
299                 if (pieceToMove.occupies(r, c)) {
300                     newBoard.grid[r][c] = '.';
301                 }
302             }
303         }
304         pieceToMove.move(moveAmount);
305         startRow = pieceToMove.getRow();
306         startCol = pieceToMove.getCol();
307         endRow = pieceToMove.getEndRow();
308         endCol = pieceToMove.getEndCol();
309         for (int r = startRow; r <= endRow; r++) {
310             for (int c = startCol; c <= endCol; c++) {
311                 if (pieceToMove.occupies(r, c)) {
312                     newBoard.grid[r][c] = pieceToMove.getId();
313                 }
314             }
315         }
316         if (pieceToMove.isPrimary()) {
317             newBoard.primaryPiece = pieceToMove;
318         }
319         return newBoard;
320     }

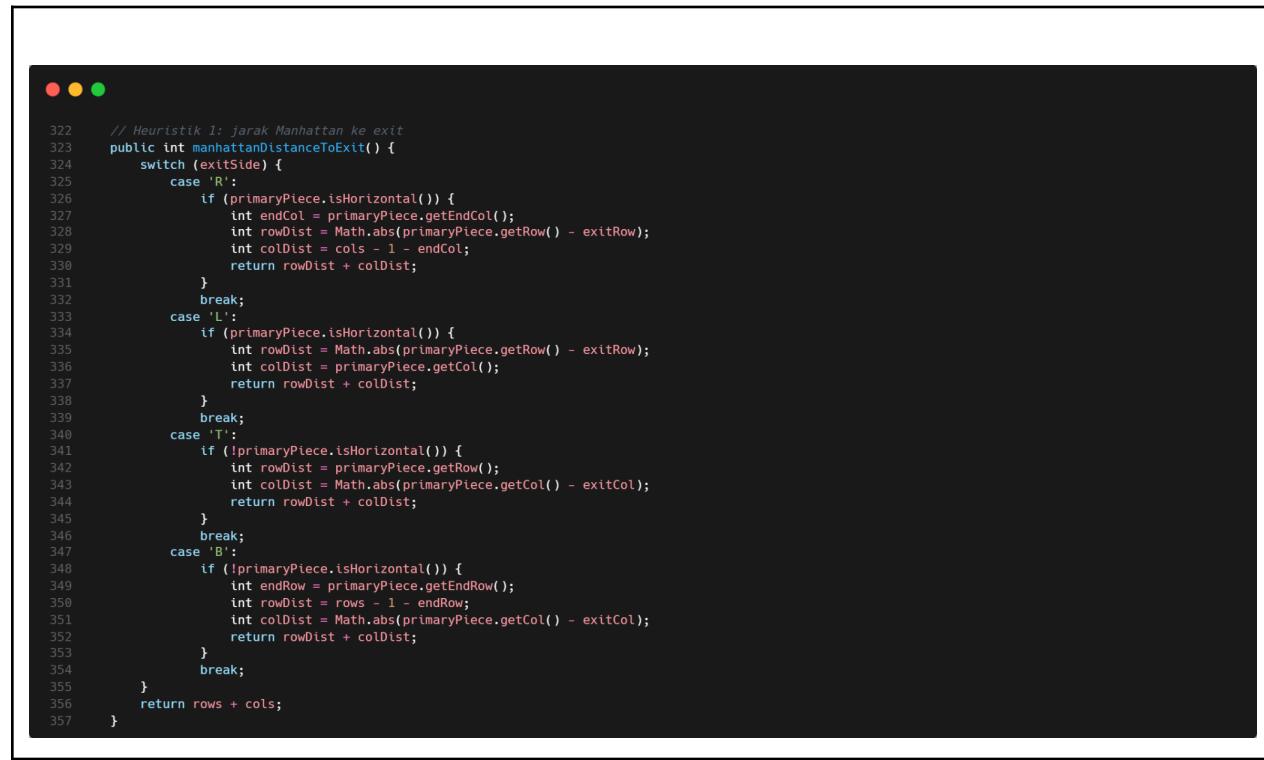
```

Method `canMoveHorizontal(Piece piece, int amount)` mengecek apakah sebuah kendaraan yang orientasinya horizontal bisa digerakkan sejauh `amount` kolom ke kiri (jika negatif) atau ke kanan (jika positif). Method memastikan kendaraan memang horizontal terlebih dahulu, lalu dihitung kolom awal baru (`newCol`) dan kolom akhirnya (`endCol`) setelah pergerakan. Jika pergerakan menyebabkan kendaraan keluar dari batas papan (`newCol < 0` atau `endCol >= cols`), gerakan tidak sah. Selanjutnya, method memeriksa semua sel di antara posisi awal dan posisi baru apakah kosong ('.') atau merupakan sel pintu keluar ('K').

Method `canMoveVertical(Piece piece, int amount)` memiliki fungsi yang mirip dengan `canMoveHorizontal()`, namun digunakan untuk kendaraan vertikal. Setelah memastikan

kendaraan tidak horizontal, method menghitung baris baru awal (*newRow*) dan akhir (*endRow*) yang akan ditempati kendaraan jika digerakkan. Jika pergerakan membuat kendaraan keluar dari batas atas atau bawah papan, hasilnya *false*. Kemudian, method memeriksa seluruh baris dalam jalur yang dilalui kendaraan (di kolom tetap), memastikan bahwa sel-sel tersebut kosong ('.') atau merupakan sel keluar ('K'). Jika tidak ada halangan, kendaraan bisa digerakkan ke arah vertikal sejauh nilai *amount*.

Method *applyMove(int pieceIdx, int moveAmount)* digunakan untuk menerapkan sebuah langkah (move) terhadap papan permainan dan mengembalikan papan baru hasil pergerakan tersebut.



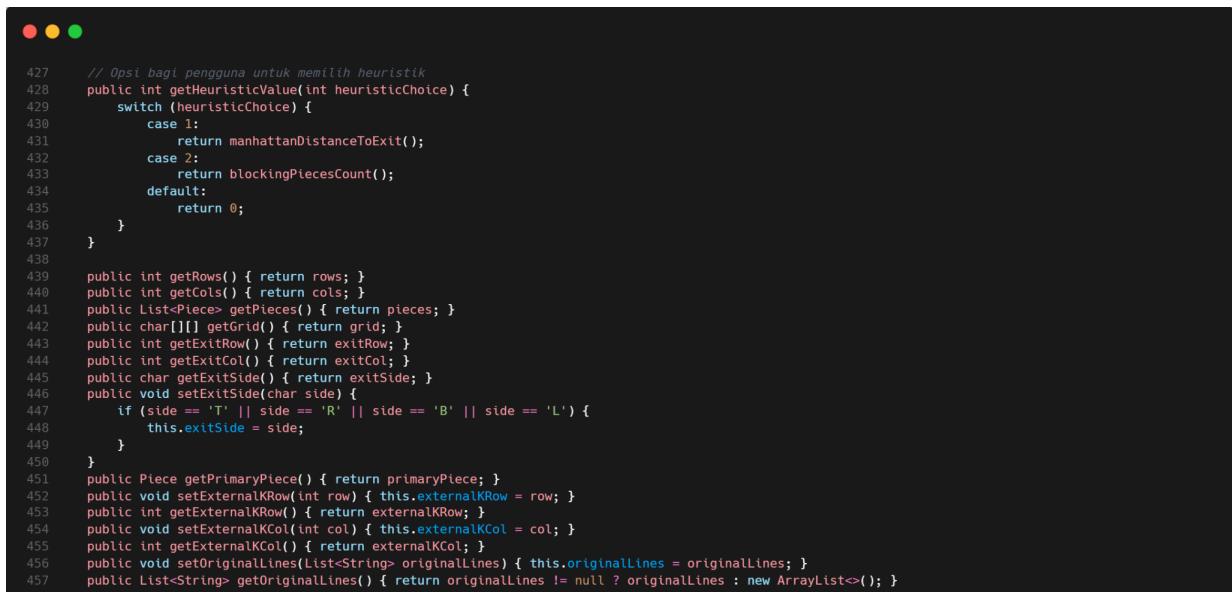
```
322 // Heuristik 1: jarak Manhattan ke exit
323 public int manhattanDistanceToExit() {
324     switch (exitSide) {
325         case 'R':
326             if (primaryPiece.isHorizontal()) {
327                 int endCol = primaryPiece.getEndCol();
328                 int rowDist = Math.abs(primaryPiece.getRow() - exitRow);
329                 int colDist = cols - 1 - endCol;
330                 return rowDist + colDist;
331             }
332             break;
333         case 'L':
334             if (primaryPiece.isHorizontal()) {
335                 int rowDist = Math.abs(primaryPiece.getRow() - exitRow);
336                 int colDist = primaryPiece.getCol();
337                 return rowDist + colDist;
338             }
339             break;
340         case 'T':
341             if (!primaryPiece.isHorizontal()) {
342                 int rowDist = primaryPiece.getRow();
343                 int colDist = Math.abs(primaryPiece.getCol() - exitCol);
344                 return rowDist + colDist;
345             }
346             break;
347         case 'B':
348             if (!primaryPiece.isHorizontal()) {
349                 int endRow = primaryPiece.getEndRow();
350                 int rowDist = rows - 1 - endRow;
351                 int colDist = Math.abs(primaryPiece.getCol() - exitCol);
352                 return rowDist + colDist;
353             }
354             break;
355     }
356     return rows + cols;
357 }
```

Method *manhattanDistanceToExit()* menghitung nilai heuristik berupa jarak Manhattan dari posisi kendaraan utama ke pintu keluar. Jarak Manhattan adalah jumlah dari selisih baris dan kolom antara dua posisi. Method ini menangani empat kemungkinan arah keluarnya papan: kanan ('R'), kiri ('L'), atas ('T'), dan bawah ('B'). Untuk arah kanan, kendaraan utama harus horizontal. Dihitung jarak vertikal ke baris pintu keluar dan jarak horizontal dari kolom ujung kendaraan ke tepi kanan papan. Untuk arah kiri, prinsipnya sama tapi dihitung dari kolom awal kendaraan ke sisi kiri papan. Jika pintu keluar berada di atas atau bawah, kendaraan utama harus vertikal, dan dihitung jarak baris ke tepi atas atau bawah, serta jarak kolom ke posisi pintu keluar.

```

359 // Heuristik 2: blocking pieces
360 public int blockingPiecesCount() {
361     int count = 0;
362     switch (exitSide) {
363         case 'R':
364             if (primaryPiece.isHorizontal()) {
365                 int row = primaryPiece.getRow();
366                 int startCol = primaryPiece.getEndCol() + 1;
367                 int endCol = cols - 1;
368                 for (int c = startCol; c <= endCol; c++) {
369                     if (grid[row][c] != '.' && grid[row][c] != 'K') {
370                         count++;
371                         while (c + 1 < cols && grid[row][c] == grid[row][c + 1]) {
372                             c++;
373                         }
374                     }
375                 }
376             break;
377         case 'L':
378             if (primaryPiece.isHorizontal()) {
379                 int row = primaryPiece.getRow();
380                 int startCol = 0;
381                 int endCol = primaryPiece.getCol() - 1;
382                 for (int c = startCol; c <= endCol; c++) {
383                     if (grid[row][c] != '.' && grid[row][c] != 'K') {
384                         count++;
385                         while (c + 1 <= endCol && grid[row][c] == grid[row][c + 1]) {
386                             c++;
387                         }
388                     }
389                 }
390             break;
391         case 'T':
392             if (!primaryPiece.isHorizontal()) {
393                 int col = primaryPiece.getCol();
394                 int startRow = 0;
395                 int endRow = primaryPiece.getRow() - 1;
396                 for (int r = startRow; r <= endRow; r++) {
397                     if (grid[r][col] != '.' && grid[r][col] != 'K') {
398                         count++;
399                         while (r + 1 <= endRow && grid[r][col] == grid[r + 1][col]) {
400                             r++;
401                         }
402                     }
403                 }
404             break;
405         case 'B':
406             if (!primaryPiece.isHorizontal()) {
407                 int col = primaryPiece.getCol();
408                 int startRow = primaryPiece.getEndRow() + 1;
409                 int endRow = rows - 1;
410                 for (int r = startRow; r <= endRow; r++) {
411                     if (grid[r][col] != '.' && grid[r][col] != 'K') {
412                         count++;
413                         while (r + 1 < rows && grid[r][col] == grid[r + 1][col]) {
414                             r++;
415                         }
416                     }
417                 }
418             }
419         }
420     }
421     return count;
422 }
423 }
```

Method `blockingPiecesCount()` merupakan fungsi heuristik kedua yang digunakan untuk memperkirakan seberapa sulit kendaraan utama mencapai pintu keluar dengan menghitung jumlah kendaraan lain yang menghalangi jalur menuju keluar. Implementasinya bergantung pada posisi pintu keluar dan orientasi kendaraan utama. Jika keluar di kanan ('R') atau kiri ('L'), kendaraan utama harus horizontal, dan method akan memindai sel-sel pada baris yang sama dari ujung kendaraan sampai ke tepi papan, menghitung kendaraan lain yang muncul dalam lintasan. Jika keluar di atas ('T') atau bawah ('B'), kendaraan utama harus vertikal, dan proses serupa dilakukan di kolom yang sesuai.



```
427     // Opsi bagi pengguna untuk memilih heuristik
428     public int getHeuristicValue(int heuristicChoice) {
429         switch (heuristicChoice) {
430             case 1:
431                 return manhattanDistanceToExit();
432             case 2:
433                 return blockingPiecesCount();
434             default:
435                 return 0;
436         }
437     }
438
439     public int getRows() { return rows; }
440     public int getCols() { return cols; }
441     public List<Piece> getPieces() { return pieces; }
442     public char[][] getGrid() { return grid; }
443     public int getExitRow() { return exitRow; }
444     public int getExitCol() { return exitCol; }
445     public char getExitSide() { return exitSide; }
446     public void setExitSide(char side) {
447         if (side == 'T' || side == 'R' || side == 'B' || side == 'L') {
448             this.exitSide = side;
449         }
450     }
451     public Piece getPrimaryPiece() { return primaryPiece; }
452     public void setExternalKRow(int row) { this.externalKRow = row; }
453     public int getExternalKRow() { return externalKRow; }
454     public void setExternalKCol(int col) { this.externalKCol = col; }
455     public int getExternalKCol() { return externalKCol; }
456     public void setOriginalLines(List<String> originalLines) { this.originalLines = originalLines; }
457     public List<String> getOriginalLines() { return originalLines != null ? originalLines : new ArrayList<>(); }
```

Method `getHeuristicValue(int heuristicChoice)` menyediakan opsi bagi pengguna untuk memilih heuristik mana yang akan digunakan untuk evaluasi keadaan papan saat ini. Parameter `heuristicChoice` menentukan heuristik yang dipakai.

Selain itu, kelas `Board` juga menyediakan berbagai `getter` dan `setter` untuk mengakses dan memodifikasi variabel-variabel privat. method-method seperti `getRows()`, `getCols()`, `getPieces()`, dan `getGrid()` mengembalikan jumlah baris, kolom, daftar potongan (`pieces`), dan representasi grid papan, secara berurutan. Method `getExitRow()`, `getExitCol()`, dan `getExitSide()` mengembalikan posisi dan sisi pintu keluar, sedangkan `setExitSide(char side)` memberikan kemampuan untuk mengubah sisi keluar asalkan nilai yang diberikan valid ('T', 'R', 'B', atau 'L').

Selain itu, ada method untuk mengakses dan mengubah variabel `externalKRow` dan `externalKCol`, yang kemungkinan merepresentasikan posisi eksternal kunci atau referensi khusus di papan. Variabel `originalLines` menyimpan representasi asli dari input papan dalam bentuk daftar string, dengan `setOriginalLines(List<String>)` untuk menetapkan dan `getOriginalLines()` untuk mendapatkan nilai tersebut, yang akan mengembalikan daftar kosong jika belum diinisialisasi.

```
459     @Override
460     public int hashCode() {
461         int hash = 7;
462         for (int i = 0; i < rows; i++) {
463             for (int j = 0; j < cols; j++) {
464                 hash = 31 * hash + grid[i][j];
465             }
466         }
467         return hash;
468     }
469
470     @Override
471     public boolean equals(Object obj) {
472         if (this == obj) return true;
473         if (obj == null || getClass() != obj.getClass()) return false;
474         Board other = (Board) obj;
475         if (rows != other.rows || cols != other.cols) return false;
476         for (int i = 0; i < rows; i++) {
477             for (int j = 0; j < cols; j++) {
478                 if (grid[i][j] != other.grid[i][j]) {
479                     return false;
480                 }
481             }
482         }
483         return true;
484     }
485
486     @Override
487     public String toString() {
488         StringBuilder sb = new StringBuilder();
489         for (int i = 0; i < rows; i++) {
490             for (int j = 0; j < cols; j++) {
491                 sb.append(grid[i][j]);
492             }
493             sb.append('\n');
494         }
495         return sb.toString();
496     }
497 }
```

Method `hashCode()` mengembalikan nilai `hash` dari objek `Board`, yang dihitung berdasarkan isi dari `grid`. Nilai `hash` ini digunakan terutama dalam struktur data seperti `HashSet` atau `HashMap` untuk membandingkan dan menyimpan objek dengan efisien.

Method `equals(Object obj)` digunakan untuk menentukan apakah dua objek `Board` identik secara logika. Perbandingan dilakukan dengan mengecek kesamaan ukuran (`rows` dan `cols`) dan kesamaan isi tiap elemen pada `grid`. Jika ada sel yang tidak cocok, method ini langsung mengembalikan `false`.

Method `toString()` memberikan representasi string dari objek `Board`. Ia membangun tampilan papan sebagai string multilini, di mana setiap baris diakhiri dengan karakter baris baru.

4.2. Piece

```
 1 public class Piece {
 2     private char id;
 3     private int row;
 4     private int col;
 5     private int length;
 6     private boolean isHorizontal;
 7     private boolean isPrimary;
 8
 9     // Konstruktor Piece
10    public Piece(char id, int row, int col, int length, boolean isHorizontal, boolean isPrimary) {
11        this.id = id;
12        this.row = row;
13        this.col = col;
14        this.length = length;
15        this.isHorizontal = isHorizontal;
16        this.isPrimary = isPrimary;
17    }
18
19    public Piece copy() {
20        return new Piece(id, row, col, length, isHorizontal, isPrimary);
21    }
22
23    // Getter id pada Piece
24    public char getId() {
25        return id;
26    }
27
28    // Getter baris awal Piece
29    public int getRow() {
30        return row;
31    }
32
33    // Getter kolom awal Piece
34    public int getCol() {
35        return col;
36    }
37
38    // Getter length Piece
39    public int getLength() {
40        return length;
41    }
42
43    public boolean isHorizontal() {
44        return isHorizontal;
45    }
46
47    public boolean isPrimary() {
48        return isPrimary;
49    }
```

Konstruktor *Piece(char id, int row, int col, int length, boolean isHorizontal, boolean isPrimary)* digunakan untuk menginisialisasi semua atribut ketika sebuah objek *Piece* dibuat. Dengan konstruktor ini, seluruh properti kendaraan langsung ditentukan saat instansiasi.

Method *copy()* menghasilkan salinan baru dari objek *Piece* saat ini. Ini berguna ketika algoritma pencarian perlu membuat papan baru tanpa mengubah keadaan asli. Selain itu, terdapat beberapa method *getter*, yaitu *getId()*, *getRow()*, *getCol()*, dan *getLength()*, yang masing-masing digunakan untuk mengambil nilai dari atribut *id*, *row*, *col*, dan *length*. Adapun method *isHorizontal()* yang mengembalikan informasi orientasi kendaraan, sedangkan *isPrimary()* memberitahukan apakah kendaraan tersebut adalah *primary piece*.

```
51     public int getEndRow() {
52         if (isHorizontal) {
53             return row;
54         } else {
55             return row + length - 1;
56         }
57     }
58
59     // Untuk mendapatkan kolom akhir Piece
60     public int getEndCol() {
61         if (isHorizontal) {
62             return col + length - 1;
63         } else {
64             return col;
65         }
66     }
67
68     // Menggerakkan Piece sejumlah amount
69     public void move(int amount) {
70         if (isHorizontal) {
71             col += amount;
72         } else {
73             row += amount;
74         }
75     }
76
77     // Mengecek apakah Piece menempati sel
78     public boolean occupies(int r, int c) {
79         if (isHorizontal) {
80             return r == row && c >= col && c <= col + length - 1;
81         } else {
82             return c == col && r >= row && r <= row + length - 1;
83         }
84     }
85
86     @Override
87     public String toString() {
88         return String.format("Piece %c at (%d,%d), length=%d, %s, %s",
89                             id, row, col, length,
90                             isHorizontal ? "horizontal" : "vertical",
91                             isPrimary ? "primary" : "non-primary");
92     }
93 }
```

Method `getEndRow()` digunakan untuk menentukan baris akhir dari suatu *piece*. Jika *piece* tersebut bergerak secara horizontal, baris akhirnya tetap sama dengan `row`. Namun, jika orientasinya vertikal, baris akhirnya dihitung sebagai `row + length - 1`

Method `getEndCol()` serupa dengan `getEndRow()`, namun digunakan untuk menghitung kolom akhir. Bila *piece* horizontal, kolom akhirnya adalah `col + length - 1`, sedangkan jika vertikal, kolom akhirnya sama dengan `col`.

Method `move(int amount)` digunakan untuk menggerakkan *piece* sejauh sejumlah `amount` langkah. Jika orientasinya horizontal, pergerakan dilakukan pada nilai `col`. Sebaliknya, untuk orientasi vertikal, perubahan terjadi pada nilai `row`. Nilai `amount` bisa positif (bergerak ke kanan/bawah) maupun negatif (bergerak ke kiri/atas)

Method `occupies(int r, int c)` mengecek apakah *piece* menempati sel tertentu di papan. Untuk *piece* horizontal, kondisi yang dicek adalah apakah barisnya sama dengan `row` dan kolomnya berada di antara `col` hingga `col + length - 1`. Untuk *piece* vertikal, kolomnya harus sama dengan `col`, dan barisnya berada dalam rentang `row` hingga `row + length - 1`.

Terakhir, method `toString()` memberikan representasi string dari suatu *piece*, mencakup informasi *id*, posisi (*row*, *col*), panjang (*length*), arah (*horizontal* atau *vertical*), serta status apakah *piece* tersebut *primary* atau tidak. Ini sangat membantu untuk keperluan debugging dan visualisasi internal keadaan *piece* saat permainan berjalan.

4.3. Node

```
1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class Node implements Comparable<Node> {
5     private Board board;
6     private Node parent;
7     private int pieceIdx; // index yang digerakkan
8     private int moveAmount; // banyak langkah yang digesek
9     private int cost; // jumlah langkah
10    private int heuristic;
11
12    // Node awal
13    public Node(Board board, int heuristicChoice) {
14        this.board = board;
15        this.parent = null;
16        this.pieceIdx = -1;
17        this.moveAmount = 0;
18        this.cost = 0;
19        this.heuristic = board.getHeuristicValue(heuristicChoice);
20    }
21
22    // Node hasil dari pergerakan
23    public Node(Node parent, Board board, int pieceIdx, int moveAmount, int heuristicChoice) {
24        this.board = board;
25        this.parent = parent;
26        this.pieceIdx = pieceIdx;
27        this.moveAmount = moveAmount;
28        this.cost = parent.cost + 1;
29        this.heuristic = board.getHeuristicValue(heuristicChoice);
30    }
31
32    public List<Node> getPath() {
33        List<Node> path = new ArrayList<Node>();
34        Node current = this;
35        while (current != null) {
36            path.add(0, current);
37            current = current.parent;
38        }
39        return path;
40    }
41
42    public int getFValue() {
43        return cost + heuristic;
44    }
45
46    public String getMoveDescription() {
47        if (parent == null) return "Initial state";
48        Piece piece = board.getPieces().get(pieceIdx);
49        String arah = piece.isHorizontal() ?
50            (moveAmount > 0 ? "kanan" : "kiri") :
51            (moveAmount > 0 ? "bawah" : "atas");
52        return String.format("%c-%s", piece.getId(), arah);
53    }
}
```

Konstruktor `Node(Board board, int heuristicChoice)` digunakan untuk membuat *Node* awal. Dalam hal ini, *parent* diset ke `null`, *pieceIdx* diatur ke `-1` sebagai penanda bahwa belum ada pergerakan, dan *moveAmount* bernilai `0`. Nilai *cost* juga dimulai dari nol. Nilai *heuristic* dihitung dengan memanggil method `getHeuristicValue` dari objek *board*, dengan parameter *heuristicChoice* yang menentukan jenis heuristik yang digunakan.

Konstruktor `Node(Node parent, Board board, int pieceIdx, int moveAmount, int heuristicChoice)` digunakan untuk membuat *Node* hasil dari suatu pergerakan. Ia menerima *Node* induk, *board* hasil pergerakan, indeks *piece* yang digerakkan, jumlah langkah perpindahan, dan

pilihan heuristik. Nilai *cost* bertambah satu dari *parent*, karena satu langkah telah diambil. Nilai *heuristic* kembali dihitung berdasarkan *board* baru.

Method *getPath()* menghasilkan urutan *Node* dari simpul awal hingga simpul saat ini. Ini dilakukan dengan melacak *parent* dari setiap *Node*, dimulai dari *this*, dan menambahkan setiap simpul ke list secara mundur (dimasukkan di posisi awal) agar urut dari awal ke akhir.

Method *getFValue()* mengembalikan penjumlahan antara *cost* dan *heuristic*, yang merupakan nilai evaluasi $f(n)$ dalam algoritma A*.

Method *getMoveDescription()* memberikan deskripsi string dari langkah yang diambil pada *Node* ini. Jika simpul merupakan simpul awal, maka dikembalikan "Initial state". Jika tidak, informasi arah gerakan diperoleh dengan memeriksa apakah *piece* bergerak horizontal atau vertikal dan apakah *moveAmount* bernilai positif atau negatif.

```
55     public Board getBoard() {
56         return board;
57     }
58
59     public Node getParent() {
60         return parent;
61     }
62
63     public int getPieceIdx() {
64         return pieceIdx;
65     }
66
67     public int getMoveAmount() {
68         return moveAmount;
69     }
70
71     public int getCost() {
72         return cost;
73     }
74
75     public int getHeuristic() {
76         return heuristic;
77     }
78
79     @Override
80     public int compareTo(Node other) {
81         return Integer.compare(this.getFValue(), other.getFValue());
82     }
83
84     @Override
85     public boolean equals(Object obj) {
86         if (this == obj) return true;
87         if (obj == null || getClass() != obj.getClass()) return false;
88         Node other = (Node) obj;
89         return board.equals(other.board);
90     }
91
92     @Override
93     public int hashCode() {
94         return board.hashCode();
95     }
96 }
```

Beberapa *getter method* disediakan untuk mengakses properti internal dari objek *Node*. Method *getBoard()* mengembalikan objek *Board* yang mewakili keadaan papan permainan pada simpul tersebut. Method *getParent()* memberikan referensi ke simpul induk dalam jalur pencarian. Method *getPieceIdx()* dan *getMoveAmount()* masing-masing mengembalikan indeks *piece* yang digerakkan serta jumlah pergeseran yang dilakukan. Method *getCost()* memberikan

total langkah dari simpul awal ke simpul ini, sementara *getHeuristic()* mengembalikan nilai estimasi jarak ke tujuan dari simpul ini.

Method *compareTo(Node other)* mengimplementasikan antarmuka *Comparable*, yang memungkinkan objek *Node* dibandingkan berdasarkan nilai *f(n)*, yaitu penjumlahan antara *cost* dan *heuristic*.

Method *equals(Object obj)* digunakan untuk membandingkan dua objek *Node*. Dua simpul dianggap sama jika keadaan *board*-nya identik. Untuk keperluan efisiensi dalam struktur data seperti *HashSet* atau *HashMap*, method *hashCode()* juga dioverride, dan menghasilkan *hash* berdasarkan *board* yang diasosiasikan dengan simpul tersebut.

4.4. Solver



```
1 public abstract class Solver {
2     protected Board initialBoard;
3     protected int heuristicChoice;
4     protected int nodesVisited;
5
6     public Solver(Board initialBoard, int heuristicChoice) {
7         this.initialBoard = initialBoard;
8         this.heuristicChoice = heuristicChoice;
9         this.nodesVisited = 0;
10    }
11
12    public abstract Solution solve();
13
14    public abstract String getName();
15
16    public int getNodesVisited() {
17        return nodesVisited;
18    }
19 }
```

Kelas Solver adalah kelas abstrak yang menjadi dasar bagi semua algoritma pencarian jalur dalam permainan Rush Hour ini. Kelas ini menyimpan beberapa atribut penting. Atribut *initialBoard* menyimpan keadaan awal papan permainan, sementara *heuristicChoice* menentukan heuristik mana yang digunakan dalam proses pencarian, jika algoritmanya memanfaatkan heuristik. Atribut *nodesVisited* berfungsi sebagai penghitung berapa banyak simpul (node) yang telah dikunjungi selama proses pencarian berlangsung.

Konstruktor *Solver(Board initialBoard, int heuristicChoice)* menginisialisasi semua atribut tersebut. Karena kelas ini bersifat abstrak, maka ia tidak dapat diinstansiasi secara langsung. Sebaliknya, kelas ini mendefinisikan dua method abstract yang wajib diimplementasikan oleh kelas turunannya.

Method *solve()* akan mengembalikan objek *Solution* yang berisi hasil pencarian dari papan awal hingga keadaan akhir. Sedangkan *getName()* digunakan untuk mengidentifikasi nama algoritma.

Method `getNodesVisited()` adalah getter yang mengembalikan nilai dari atribut `nodesVisited`. Nilai ini bisa digunakan untuk mengevaluasi efisiensi algoritma dalam menemukan solusi.

4.5. Solution

```
1 import java.io.FileWriter;
2 import java.io.IOException;
3 import java.util.List;
4
5 public class Solution {
6     private static final String RESET = Main.RESET;
7     private static final String BRIGHT_RED = Main.BRIGHT_RED;
8     private static final String BRIGHT_GREEN = Main.BRIGHT_GREEN;
9     private static final String BRIGHT_YELLOW = Main.BRIGHT_YELLOW;
10    private static final String BRIGHT_BLUE = Main.BRIGHT_BLUE;
11    private static final String BRIGHT_CYAN = Main.BRIGHT_CYAN;
12    private static final String BRIGHT_WHITE = Main.BRIGHT_WHITE;
13    private static final String BOLD = Main.BOLD;
14
15    private List<Node> path;
16
17    public Solution(List<Node> path) {
18        this.path = path;
19    }
20
21    public List<Node> getMoves() {
22        return path.subList(1, path.size());
23    }
24
25    public void printSteps() {
26        System.out.println(BOLD + BRIGHT_CYAN + "Papan Awal" + RESET);
27        printBoardWithFrame(path.get(0).getBoard(), -1);
28
29        for (int i = 1; i < path.size(); i++) {
30            Node move = path.get(i);
31            Piece piece = move.getBoard().getPieces().get(move.getPieceIdx());
32
33            String direction;
34            if (piece.isHorizontal()) {
35                direction = move.getMoveAmount() > 0 ? "right" : "left";
36            } else {
37                direction = move.getMoveAmount() > 0 ? "down" : "up";
38            }
39
40            System.out.println("\n" + BOLD + BRIGHT_CYAN + "Gerakan " + i + ":" + " "
41                            + BRIGHT_YELLOW + piece.getId() + "-" + direction + RESET);
42
43            printBoardWithFrame(move.getBoard(), move.getPieceIdx());
44        }
45    }
46
47    public void printStepsAnimated() {
48        System.out.println(BOLD + BRIGHT_CYAN + "Papan Awal" + RESET);
49        printBoardWithFrame(path.get(0).getBoard(), -1);
50
51        try {
52            for (int i = 1; i < path.size(); i++) {
53                Node move = path.get(i);
54                Piece piece = move.getBoard().getPieces().get(move.getPieceIdx());
55
56                String direction;
57                if (piece.isHorizontal()) {
58                    direction = move.getMoveAmount() > 0 ? "right" : "left";
59                } else {
60                    direction = move.getMoveAmount() > 0 ? "down" : "up";
61                }
62
63                Thread.sleep(800);
64
65                System.out.println("\n" + BOLD + BRIGHT_CYAN + "Gerakan " + i + ":" + " "
66                                + BRIGHT_YELLOW + piece.getId() + "-" + direction + RESET);
67
68                printBoardWithFrame(move.getBoard(), move.getPieceIdx());
69            }
70        } catch (InterruptedException e) {}
71    }
}
```

Konstruktor `Solution(List<Node> path)` menerima sebuah daftar lintasan solusi dan menyimpannya dalam `path`. Method `getMoves()` mengembalikan semua langkah solusi dari simpul pertama ke simpul terakhir, tetapi tidak menyertakan simpul awal.

Method `printSteps()` mencetak seluruh langkah solusi ke layar. Pertama-tama, papan awal dicetak menggunakan `printBoardWithFrame(Board, int)` dengan argumen `-1` untuk menandai tidak ada potongan yang disorot. Kemudian, untuk setiap langkah dalam solusi, potongan (`piece`) yang digerakkan diambil menggunakan indeks dari `Node`, dan arah gerakan ditentukan berdasarkan apakah potongan tersebut horizontal atau vertikal serta nilai `moveAmount`. Jika nilai `moveAmount` lebih besar dari nol, arah gerakannya ke kanan atau bawah. Jika lebih kecil dari nol, ke kiri atau atas. Informasi ini lalu dicetak ke layar bersama keadaan papan saat itu.

Method `printStepsAnimated()` memiliki perilaku serupa dengan `printSteps()`, tetapi menambahkan `Thread.sleep(800)` di antara langkah-langkah untuk memberikan efek animasi dengan jeda 800 milidetik.

```
73
74     private void printBoardWithFrame(Board board, int movedPieceIdx) {
75         char[][] grid = board.getGrid();
76         int rows = board.getRows();
77         int cols = board.getColumns();
78
79         char movedPieceId = (movedPieceIdx >= 8) ?
80             board.getPieces().get(movedPieceIdx).getId() + ' ';
81
82         Piece primaryPiece = null;
83         for (Piece piece : board.getPieces()) {
84             if (piece.isPrimary()) {
85                 primaryPiece = piece;
86                 break;
87             }
88         }
89         if (primaryPiece == null) {
90             throw new IllegalStateException("Primary piece not found");
91         }
92
93         int primaryRow = primaryPiece.getRow();
94         int primaryCol = primaryPiece.getCol();
95         boolean isHorizontal = primaryPiece.isHorizontal();
96
97         int externalKRow = board.getExternalKRow();
98         int externalKCol = board.getExternalKCol();
99         char exitSide = board.getExitSide();
100
101        if (exitSide == 'T' || exitSide == 'B') {
102            externalKRow = primaryRow;
103        } else if (exitSide == 'L' || exitSide == 'R') {
104            externalKRow = primaryCol;
105        }
106
107        boolean isSolved = board.isSolved();
108
109        if (exitSide == 'T') {
110            if (!isSolved) {
111                for (int index = 0; index < 4; index++) {
112                    System.out.print(" ");
113
114                    for (int j = 0; j < primaryCol; j++) {
115                        System.out.print(" ");
116                    }
117
118                    if (index == 0) {
119                        if (primaryPiece.getLength() == 3) {
120                            System.out.println(BRIGHT_RED + "P" + RESET);
121                            System.out.print(" ");
122                            for (int j = 8; j < primaryCol; j++) {
123                                System.out.print(" ");
124                            }
125                            System.out.println(BRIGHT_RED + "P (Berhasil Keluar)" + RESET);
126                        } else if (index == 1) {
127                            System.out.println(BRIGHT_RED + "P" + RESET);
128                        } else if (index == 2) {
129                            System.out.print(" ");
130                        } else if (index == 3) {
131                            System.out.println(BRIGHT_GREEN + "K" + RESET);
132                        }
133                    }
134                }
135            } else {
136                System.out.print(" ");
137
138                for (int i = 0; i < primaryCol; i++) {
139                    System.out.print(" ");
140                }
141
142                System.out.println(BRIGHT_GREEN + "K" + RESET);
143            }
144        }
145
146        if (exitSide == 'L' && isSolved) {
147            if (primaryPiece.getLength() == 3) {
```

```

148         System.out.print(" ");
149     }
150     System.out.print(" " + BRIGHT_CYAN + "p");
151 } else if (exitSide == 'L') {
152     System.out.print(" " + BRIGHT_CYAN + "p");
153 } else {
154     System.out.print(BRIGHT_CYAN + "p");
155 }
156
157 for (int j = 0; j < cols; j++) {
158     System.out.print(" ");
159 }
160 System.out.println("p" + RESET);
161
162 for (int i = R; i < rows; i++) {
163     if (exitSide == 'L' && i == primaryRow) {
164         if (isSolved) {
165             if (primaryPiece.getLength() == J) {
166                 System.out.print(BRIGHT_RED + "P" + RESET);
167             }
168             System.out.print(BRIGHT_RED + "PP" + BRIGHT_GREEN + "K" + RESET);
169         } else {
170             System.out.print(BRIGHT_GREEN + "K" + RESET);
171         }
172     } else {
173         if (exitSide == 'L' && (isSolved)) {
174             if (primaryPiece.getLength() == J) {
175                 System.out.print(" ");
176             }
177             System.out.print(" ");
178         } else if (exitSide == 'L') {
179             System.out.print(" ");
180         }
181     }
182
183     System.out.print(BRIGHT_CYAN + "I" + RESET);
184
185     for (int j = 0; j < cols; j++) {
186         char cell = grid[i][j];
187         if ((isSolved && exitSide == 'R') && j == externalRow && cell == 'P') ||
188             (isSolved && exitSide == 'L') && j == externalRow && cell == 'P') ||
189             (isSolved && exitSide == 'T') && j == externalRow && cell == 'P') ||
190             (isSolved && exitSide == 'B') && j == externalRow && cell == 'P')) {
191             System.out.print(BRIGHT_WHITE + "*" + RESET);
192         } else if (cell == 'P') {
193             System.out.print(BRIGHT_RED + "P" + RESET);
194         } else if (cell == movedPieceId) {
195             System.out.print(BRIGHT_YELLOW + cell + RESET);
196         } else if (cell == 'I') {
197             System.out.print(BRIGHT_WHITE + "*" + RESET);
198         } else if (cell == 'K') {
199             System.out.print(BRIGHT_WHITE + "*" + RESET);
200         } else {
201             System.out.print(BRIGHT_BLUE + cell + RESET);
202         }
203     }
204
205     if (exitSide == 'R' && i == primaryRow) {
206         System.out.print(BRIGHT_CYAN + "I" + RESET + BRIGHT_GREEN + "K" + RESET);
207         if (isSolved) {
208             if (primaryPiece.getLength() == 3) {
209                 System.out.print(BRIGHT_RED + "P" + RESET);
210             }
211             System.out.print(BRIGHT_RED + "PP" + "Baris" + Ketuar) + RESET);
212         }
213     } else if (exitSide == 'L' && i == primaryRow) {
214         System.out.print(BRIGHT_CYAN + "I" + RESET);
215     }
216
217     if (exitSide == 'T' && i == primaryRow && (isSolved)) {
218         System.out.print(BRIGHT_RED + " " + Baris + Ketuar) + RESET);
219     }
220
221     System.out.println();
222 }
223
224 if (exitSide == 'L' && isSolved) {
225     if (primaryPiece.getLength() == 3) {
226         System.out.print(" ");
227     }
228     System.out.print(" " + BRIGHT_CYAN + "K");
229 } else if (exitSide == 'L') {
230     System.out.print(" " + BRIGHT_CYAN + "K");
231 } else {
232     System.out.print(BRIGHT_CYAN + "K");
233 }
234
235 for (int j = 0; j < cols; j++) {
236     System.out.print(" ");
237 }
238 System.out.println("K" + RESET);
239
240 if (exitSide == 'B') {
241     if (isSolved) {
242         for (int index = 0; index < 4; index++) {
243             System.out.print(" ");
244
245         for (int i = 0; i < primaryCol; i++) {
246             System.out.print(" ");
247         }
248     }

```

```

249         if (index == 0) {
250             System.out.println(BRIGHT_GREEN + "K" + RESET);
251         } else if (index == 1) {
252             System.out.println();
253         } else if (index == 2) {
254             System.out.println(BRIGHT_RED + "P" + RESET);
255         } else if (index == 3) {
256             System.out.println(BRIGHT_RED + "P (Berhasil Keluar)" + RESET);
257             if (primaryPiece.getLength() == 3) {
258                 System.out.print(" ");
259                 for (int j = 0; j < primaryCol; j++) {
260                     System.out.print(" ");
261                 }
262             }
263             System.out.println(BRIGHT_RED + "P" + RESET);
264         }
265     } else {
266         System.out.print(" ");
267
268         for (int j = 0; j < primaryCol; j++) {
269             System.out.print(" ");
270         }
271
272         System.out.println(BRIGHT_GREEN + "K" + RESET);
273     }
274 }
275
276
277 if (movedPieceIdx >= 0) {
278     System.out.println(BRIGHT_RED + "■" + RESET + "Primary Piece (P) " +
279                         BRIGHT_GREEN + "■" + RESET + "Exit (K) " +
280                         BRIGHT_YELLOW + "■" + RESET + "Moved Piece (" + movedPieceIdx + ")");
281 } else {
282     System.out.println(BRIGHT_RED + "■" + RESET + "Primary Piece (P) " +
283                         BRIGHT_GREEN + "■" + RESET + "Exit (K) " +
284                         BRIGHT_BLUE + "■" + RESET + "Other Pieces");
285 }
286

```

Method *printBoardWithFrame(Board board, int movedPieceIdx)* bertanggung jawab untuk mencetak representasi visual papan permainan *Board* ke terminal, lengkap dengan bingkai dan elemen warna yang memperjelas status permainan. Method ini menerima parameter *board* dan *movedPieceIdx*, yang menunjukkan indeks potongan terakhir yang digerakkan jika ada.

Method mengambil *grid*, ukuran papan, serta posisi dan orientasi *primaryPiece*. Informasi tentang posisi keluar (*exitSide*, *externalKRow*, *externalKCol*) digunakan untuk menentukan lokasi simbol "K" sebagai penanda pintu keluar. Jika keadaan papan menunjukkan bahwa permainan telah selesai, *primaryPiece* ditampilkan seolah-olah telah keluar dari papan.

Method ini menyesuaikan output dengan posisi pintu keluar dan memberikan representasi khusus jika *primaryPiece* a telah berhasil keluar, termasuk penambahan keterangan (*Berhasil Keluar*). Warna digunakan untuk membedakan *primaryPiece* (*P*) dalam merah terang, potongan yang digerakkan dalam kuning, potongan lain dalam biru, dan pintu keluar (*K*) dalam hijau. Adapun legenda dicetak untuk membantu pengguna memahami makna dari simbol-simbol dan warna yang ditampilkan.

```
288     public void saveToFile(String filePath) throws IOException {
289         FileWriter writer = new FileWriter(filePath);
290
291         writer.write("Papan Awal\n");
292         char[][] grid = path.get(0).getBoard().getGrid();
293         int rows = path.get(0).getBoard().getRows();
294         int cols = path.get(0).getBoard().getCols();
295
296         int externalKRow = path.get(0).getBoard().getExternalKRow();
297         int externalKCol = path.get(0).getBoard().getExternalKCol();
298         char exitSide = path.get(0).getBoard().getExitSide();
299
300         StringBuilder save = new StringBuilder();
```

```

301     if (exitSide == 'T') {
302         for (int j = 0; j < externalKCol; j++) {
303             save.append(" ");
304         }
305         save.append("K\n");
306     }
307     for (int i = 0; i < rows; i++) {
308         if (exitSide == 'L') {
309             if (i == externalKRow) {
310                 save.append("K");
311             } else {
312                 save.append(" ");
313             }
314         }
315         for (int j = 0; j < cols; j++) {
316             save.append(grid[i][j]);
317         }
318         if (exitSide == 'R') {
319             if (i == externalKRow) {
320                 save.append("K");
321             }
322         }
323         save.append("\n");
324     }
325     if (exitSide == 'B') {
326         for (int j = 0; j < externalKCol; j++) {
327             save.append(" ");
328         }
329         save.append("K\n");
330     }
331
332     writer.write(save.toString());
333     writer.write("\n");
334
335     for (int index = 1; index < path.size(); index++) {
336         Node move = path.get(index);
337
338         String direction;
339         Piece piece = move.getBoard().getPieces().get(move.getPieceIdx());
340
341         if (piece.isHorizontal()) {
342             direction = move.getMoveAmount() > 0 ? "right" : "left";
343         } else {
344             direction = move.getMoveAmount() > 0 ? "down" : "up";
345         }
346
347         writer.write("Gerakan " + index + ": " + piece.getId() + "-" + direction + "\n");
348         grid = move.getBoard().getGrid();
349         boolean isSolved = move.getBoard().isSolved();
350
351         save = new StringBuilder();
352         if (exitSide == 'T') {
353             for (int j = 0; j < externalKCol; j++) {
354                 save.append(" ");
355             }
356             save.append("K\n");
357         }
358         for (int i = 0; i < rows; i++) {
359             if (exitSide == 'L') {
360                 if (i == externalKRow) {
361                     save.append("K");
362                 } else {
363                     save.append(" ");
364                 }
365             }
366             for (int j = 0; j < cols; j++) {
367                 if (!isSolved && grid[i][j] == 'P') {
368                     if ((exitSide == 'R' && i == externalKRow && j != cols - 1) ||
369                         (exitSide == 'L' && i == externalKRow && j != 0) ||
370                         (exitSide == 'T' && i == externalKCol && j != 0) ||
371                         (exitSide == 'B' && i == externalKCol && j != rows - 1)) {
372                         save.append('.');
373                     } else {
374                         save.append('P');
375                     }
376                 } else {
377                     save.append(grid[i][j]);
378                 }
379             }
380             if (exitSide == 'R') {
381                 if (i == externalKRow) {
382                     save.append("K");
383                 }
384             }
385             save.append("\n");
386         }
387         if (exitSide == 'B') {
388             for (int j = 0; j < externalKCol; j++) {
389                 save.append(" ");
390             }
391             save.append("K\n");
392         }
393
394         writer.write(save.toString());
395         writer.write("\n");
396     }
397
398     writer.close();
399 }
400 }
```

Method `saveToFile(String filePath)` menyimpan langkah-langkah solusi permainan ke dalam sebuah file teks di path yang ditentukan oleh parameter `filePath`. Method ini menghasilkan representasi papan awal dan setiap langkah perpindahan potongan ke dalam format teks sederhana yang mencerminkan posisi tiap potongan serta pintu keluar (K).

Pertama, method menulis label "Papan Awal", lalu mencetak papan awal dengan memperhatikan posisi pintu keluar berdasarkan nilai `exitSide` (atas, bawah, kiri, kanan). Posisi pintu keluar ditandai dengan huruf K yang ditambahkan secara manual sebelum atau sesudah baris/kolom papan sesuai letaknya.

Setelah itu, untuk setiap langkah dalam `path` mulai dari indeks 1, method mencatat deskripsi gerakan dalam format “Gerakan X: id-arah”, lalu menyusun kembali grid papan setelah gerakan tersebut. Jika `primaryPiece` (P) sudah mencapai pintu keluar, sebagian sel tempat `primaryPieceI` tersebut berada akan ditandai dengan titik (.) untuk membedakan `primaryPiece` yang sedang keluar melalui pintu keluar dengan yang masih berada di papan. Akhirnya, seluruh isi papan setelah setiap gerakan juga ditulis ke file. File kemudian ditutup setelah semua langkah disimpan.

4.6. Main

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Scanner;
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static final String RESET = "\u001B[0m";
    public static final String RED = "\u001B[31m";
    public static final String GREEN = "\u001B[32m";
    public static final String YELLOW = "\u001B[33m";
    public static final String BLUE = "\u001B[34m";
    public static final String PURPLE = "\u001B[35m";
    public static final String CYAN = "\u001B[36m";
    public static final String WHITE = "\u001B[37m";

    public static final String BRIGHT_RED = "\u001B[91m";
    public static final String BRIGHT_GREEN = "\u001B[92m";
    public static final String BRIGHT_YELLOW = "\u001B[93m";
    public static final String BRIGHT_BLUE = "\u001B[94m";
    public static final String BRIGHT_PURPLE = "\u001B[95m";
    public static final String BRIGHT_CYAN = "\u001B[96m";
    public static final String BRIGHT_WHITE = "\u001B[97m";

    public static final String BG_RED = "\u001B[41m";
    public static final String BG_GREEN = "\u001B[42m";
    public static final String BG_YELLOW = "\u001B[43m";
    public static final String BG_BLUE = "\u001B[44m";
    public static final String BG_PURPLE = "\u001B[45m";
    public static final String BG_CYAN = "\u001B[46m";

    public static final String BOLD = "\u001B[1m";
    public static final String UNDERLINE = "\u001B[4m";

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        boolean continueProgram = true;

        printWelcomeBanner();

        Board initialBoard = null;

        while (continueProgram) {
            try {
                if (initialBoard == null) {
```

```
if (initialBoard == null) {
    System.out.println(BOLD + BRIGHT_CYAN + "Please enter the input file path:" + RESET);
    System.out.print(BRIGHT_YELLOW + "► " + RESET);
    String filePath = scanner.nextLine();

    if (filePath.trim().isEmpty()) {
        System.out.println(BRIGHT_RED + "X Error: File path cannot be empty. Please try again." + RESET);
        continue;
    }

    System.out.println(BRIGHT_PURPLE + "\n✓ Loading puzzle file..." + RESET);

    try {
        long startReadTime = System.currentTimeMillis();
        initialBoard = readInputFile(filePath);
        long endReadTime = System.currentTimeMillis();

        System.out.println(BRIGHT_GREEN + "✓ File loaded successfully in " +
            (endReadTime - startReadTime) + "ms!" + RESET);
    } catch (IOException e) {
        System.out.println(BRIGHT_RED + "X Error loading file: " + e.getMessage() + RESET);
    }

    boolean validTryAgainChoice = false;
    while (!validTryAgainChoice) {
        System.out.println(BRIGHT_CYAN + "Do you want to try another file? (y/n)" + RESET);
        System.out.print(BRIGHT_YELLOW + "► " + RESET);
        String tryAgain = scanner.nextLine();

        if (tryAgain.equalsIgnoreCase("y")) {
            validTryAgainChoice = true;
        } else if (tryAgain.equalsIgnoreCase("n")) {
            validTryAgainChoice = true;
            continueProgram = false;
        } else {
            System.out.println(BRIGHT_RED + "X Invalid input. Please enter 'y' or 'n'." + RESET);
        }
    }
    continue;
}
}
```

```
System.out.println("\n" + BOLD + BRIGHT_CYAN + "● Initial Board:" + RESET);
printBoardWithFrame(initialBoard);

boolean validAlgorithmChoice = false;
int algorithmChoice = 0;

while (!validAlgorithmChoice) {
    System.out.println("\n" + BOLD + BRIGHT_CYAN + "● Select Algorithm:" + RESET);
    printAlgorithmMenu();

    System.out.print(BRIGHT_YELLOW + "► Enter your choice (1-4): " + RESET);
    String algorithmInput = scanner.nextLine();

    try {
        algorithmChoice = Integer.parseInt(algorithmInput);
        if (algorithmChoice >= 1 && algorithmChoice <= 4) {
            validAlgorithmChoice = true;
        } else {
            System.out.println(BRIGHT_RED + "X Invalid input. Please enter a number between 1 and 4." + RESET);
        }
    } catch (NumberFormatException e) {
        System.out.println(BRIGHT_RED + "X Invalid input. Please enter a valid number." + RESET);
    }
}

int heuristicChoice = 1;
if (algorithmChoice != 2) {
    boolean validHeuristicChoice = false;

    while (!validHeuristicChoice) {
        System.out.println("\n" + BOLD + BRIGHT_CYAN + "● Select Heuristic:" + RESET);
        printHeuristicMenu();

        System.out.print(BRIGHT_YELLOW + "► Enter your choice (1-2): " + RESET);
        String heuristicInput = scanner.nextLine();

        try {
            heuristicChoice = Integer.parseInt(heuristicInput);
            if (heuristicChoice >= 1 && heuristicChoice <= 2) {
                validHeuristicChoice = true;
            } else {
                System.out.println(BRIGHT_RED + "X Invalid input. Please enter either 1 or 2." + RESET);
            }
        } catch (NumberFormatException e) {
            System.out.println(BRIGHT_RED + "X Invalid input. Please enter a valid number." + RESET);
        }
    }
}
}
```

```

Solver solver;
String algorithmName = "";
String heuristicName = getHeuristicName(heuristicChoice);

int beamWidth = 3;
if (algorithmChoice == 4) {
    boolean validBeamWidth = false;

    while (!validBeamWidth) {
        System.out.println("\n" + BRIGHT_CYAN + "Enter beam width (recommended: 3-5):" + RESET);
        System.out.print(BRIGHT_YELLOW + "▶ " + RESET);
        String beamWidthInput = scanner.nextLine();

        try {
            beamWidth = Integer.parseInt(beamWidthInput);
            if (beamWidth >= 1) {
                validBeamWidth = true;
            } else {
                System.out.println(BRIGHT_RED + "X Beam width must be a positive integer." + RESET);
            }
        } catch (NumberFormatException e) {
            System.out.println(BRIGHT_RED + "X Invalid input. Please enter a valid number." + RESET);
        }
    }
}

switch (algorithmChoice) {
    case 1:
        algorithmName = "Greedy Best First Search";
        solver = new GreedyBestFirstSearch(initialBoard, heuristicChoice);
        break;
    case 2:
        algorithmName = "Uniform Cost Search (UCS)";
        solver = new UniformCostSearch(initialBoard);
        break;
    case 3:
        algorithmName = "A* Search";
        solver = new AStarSearch(initialBoard, heuristicChoice);
        break;
    case 4:
        algorithmName = "Beam Search (width=" + beamWidth + ")";
        solver = new BeamSearch(initialBoard, heuristicChoice, beamWidth);
        break;
    default:
        throw new IllegalArgumentException("Invalid algorithm choice");
}

System.out.println("\n" + BRIGHT_PURPLE + "● Running " + algorithmName +
    (algorithmChoice != 2 ? " with " + heuristicName + " heuristic" : "") +
    "... " + RESET);

printLoadingAnimation();

```

```

long startTime = System.currentTimeMillis();
Solution solution = solver.solve();
long endTime = System.currentTimeMillis();
long executionTime = endTime - startTime;

System.out.println("\n" + BOLD + BG_BLUE + "===== RESULTS =====" + RESET);
System.out.println(BOLD + BRIGHT_CYAN + "● Algorithm: " + RESET + BRIGHT_WHITE + algorithmName + RESET);

if (algorithmChoice != 2) {
    System.out.println(BOLD + BRIGHT_CYAN + "● Heuristic: " + RESET + BRIGHT_WHITE + heuristicName + RESET);
}

System.out.println(BOLD + BRIGHT_CYAN + "● Execution time: " + RESET + BRIGHT_WHITE + executionTime + " ms" + RESET);
System.out.println(BOLD + BRIGHT_CYAN + "● Nodes visited: " + RESET + BRIGHT_WHITE + solver.getNodesVisited() + RESET);

if (solution != null) {
    int moveCount = solution.getMoves().size();
    System.out.println(BOLD + BRIGHT_GREEN + "\n/ SOLUTION FOUND! " + RESET +
        BRIGHT_WHITE + "Number of moves: " + moveCount + RESET);

    System.out.println(BOLD + BG_BLUE + "===== SOLUTION STEPS =====" + RESET);

    solution.printStepsAnimated();

    boolean validSaveChoice = false;
    while (!validSaveChoice) {
        System.out.println("\n" + BRIGHT_CYAN + "● Do you want to save the solution to a file? (y/n)" + RESET);
        System.out.print(BRIGHT_YELLOW + "▶ " + RESET);
        String saveToFile = scanner.nextLine();

        if (saveToFile.equalsIgnoreCase("y") || saveToFile.equalsIgnoreCase("n")) {
            validSaveChoice = true;

            if (saveToFile.equalsIgnoreCase("y")) {
                boolean validOutputPath = false;
                while (!validOutputPath) {
                    System.out.println(BRIGHT_CYAN + "Enter output file path (.txt):" + RESET);
                    System.out.print(BRIGHT_YELLOW + "▶ " + RESET);
                    String outputPath = scanner.nextLine();

                    if (outputPath.trim().isEmpty()) {
                        System.out.println(BRIGHT_RED + "X Output path cannot be empty. Please try again." + RESET);
                    } else {
                        try {
                            solution.saveToFile(outputPath);
                            System.out.println(BRIGHT_GREEN + "✓ Solution saved to " + outputPath + RESET);
                            validOutputPath = true;
                        } catch (IOException e) {
                            System.out.println(BRIGHT_RED + "X Error saving file: " + e.getMessage() +
                                ". Please try another path." + RESET);
                        }
                    }
                }
            }
        }
    }
}

```

```

        }
    }
}
} else {
    System.out.println(BRIGHT_RED + "X Invalid input. Please enter 'y' or 'n'." + RESET);
}
}
} else {
    System.out.println(BOLD + BRIGHT_RED + "\nX NO SOLUTION FOUND!" + RESET);
    System.out.println(YELLOW + "The puzzle may be unsolvable or require more resources than available." + RESET);
}

boolean validContinueChoice = false;
while (!validContinueChoice) {
    System.out.println("\n" + BRIGHT_CYAN + "● Do you want to run another test? (y/n)" + RESET);
    System.out.print(BRIGHT_YELLOW + "▶ " + RESET);
    String continueChoice = scanner.nextLine();

    if (continueChoice.equalsIgnoreCase("y")) {
        validContinueChoice = true;
    } else if (continueChoice.equalsIgnoreCase("n")) {
        validContinueChoice = true;
        continueProgram = false;
    } else {
        System.out.println(BRIGHT_RED + "X Invalid input. Please enter 'y' or 'n'." + RESET);
    }
}

if (continueProgram) {
    if (initialBoard != null) {
        boolean validReuseChoice = false;
        while (!validReuseChoice) {
            System.out.println("\n" + BRIGHT_CYAN + "● Do you want to reuse the same puzzle file? (y/n)" + RESET);
            System.out.print(BRIGHT_YELLOW + "▶ " + RESET);
            String reuseChoice = scanner.nextLine();

            if (reuseChoice.equalsIgnoreCase("y")) {
                validReuseChoice = true;

                System.out.println(BRIGHT_GREEN + "✓ Reusing the same puzzle file." + RESET);

                System.out.println("\n" + BOLD + BRIGHT_CYAN + "● Initial Board:" + RESET);
                printBoardWithFrame(initialBoard);
            } else if (reuseChoice.equalsIgnoreCase("n")) {
                validReuseChoice = true;
                initialBoard = null;
                continue;
            } else {
                System.out.println(BRIGHT_RED + "X Invalid input. Please enter 'y' or 'n'." + RESET);
            }
        }
    }
}
}

```



```

// Print each line with different colors for a rainbow effect
String[] colors = {BRIGHT_RED, BRIGHT_YELLOW, BRIGHT_GREEN, BRIGHT_CYAN, BRIGHT_BLUE, BRIGHT_PURPLE};
for (int i = 0; i < banner.length; i++) {
    System.out.println(colors[i % colors.length] + banner[i] + RESET);
}

System.out.println("\n" + BRIGHT_CYAN + "IF2211 Strategi Algoritma - Tugas Kecil 3" + RESET);
System.out.println(YELLOW + "Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding" + RESET);
System.out.println(BRIGHT_GREEN + "-----" + RESET);
}

private static void printAlgorithmMenu() {
    System.out.println(BRIGHT_GREEN + "1. Greedy Best First Search" + RESET);
    System.out.println(BRIGHT_GREEN + "2. Uniform Cost Search (UCS)" + RESET);
    System.out.println(BRIGHT_GREEN + "3. A* Search" + RESET);
    System.out.println(BRIGHT_GREEN + "4. Beam Search (Bonus)" + RESET);
}

private static void printHeuristicMenu() {
    System.out.println(BRIGHT_BLUE + "1. Manhattan Distance to Exit" + RESET);
    System.out.println(BRIGHT_BLUE + "2. Blocking Vehicles Count" + RESET);
}

private static void printLoadingAnimation() {
    String[] frames = {"-", "+", "x", ":", "x", ":", "+", "-"};
    try {
        for (int i = 0; i < 20; i++) {
            System.out.print("\r" + BRIGHT_CYAN + frames[i % frames.length] +
                " Solving puzzle... " + (i * 5) + "%" + RESET);
            Thread.sleep(100);
        }
        System.out.println("\r" + BRIGHT_GREEN + "✓ Puzzle solved!" + RESET);
    } catch (InterruptedException e) {}
}

```

```
    private static void printBoardWithFrame(Board board) {
        char[][] grid = board.getGrid();
        int rows = board.getRows();
        int cols = board.getCols();

        Piece primaryPiece = null;
        for (Piece piece : board.get_pieces()) {
            if (piece.isPrimary()) {
                primaryPiece = piece;
                break;
            }
        }

        if (primaryPiece == null) {
            throw new IllegalStateException("Primary piece not found");
        }

        int primaryRow = primaryPiece.getRow();
        int primaryCol = primaryPiece.getCol();
        boolean isHorizontal = primaryPiece.isHorizontal();

        int externalKRow = board.getExternalKRow();
        int externalKCol = board.getExternalKCol();
        char exitSide = board.getExitSide();
        boolean isSolved = board.isSolved();

        if (exitSide == 'T' || exitSide == 'B') {
            externalKCol = primaryCol;
        } else if (exitSide == 'L' || exitSide == 'R') {
            externalKRow = primaryRow;
        }

        if (exitSide == 'T') {
            System.out.print(" ");

            for (int j = 0; j < primaryCol; j++) {
                System.out.print(" ");
            }

            if (isSolved) {
                System.out.println(BRIGHT_GREEN + "K " + BRIGHT_RED + "PP (Berhasil Keluar)" + RESET);
            } else {
                System.out.println(BRIGHT_GREEN + "K" + RESET);
            }
        }
    }
}
```

```
if (exitSide == 'L') {
    System.out.print(" " + BRIGHT_CYAN + "||");
} else {
    System.out.print(BRIGHT_CYAN + "||");
}

for (int j = 0; j < cols; j++) {
    System.out.print("=");
}
System.out.println("||" + RESET);

for (int i = 0; i < rows; i++) {
    if (exitSide == 'L' && i == primaryRow) {
        if (isSolved) {
            System.out.print(BRIGHT_GREEN + "K " + BRIGHT_RED + "PP " + RESET);
        } else {
            System.out.print(BRIGHT_GREEN + "K" + RESET);
        }
    } else {
        if (exitSide == 'L') {
            System.out.print(" ");
        }
    }
}

System.out.print(BRIGHT_CYAN + "||" + RESET);

for (int j = 0; j < cols; j++) {
    char cell = grid[i][j];
    if (cell == 'P') {
        System.out.print(BRIGHT_RED + cell + RESET);
    } else if (cell == '.') {
        System.out.print(BRIGHT_WHITE + "." + RESET);
    } else if (cell == 'K') {
        System.out.print(BRIGHT_WHITE + "." + RESET);
    } else {
        System.out.print(BRIGHT_BLUE + cell + RESET);
    }
}

if (exitSide == 'R' && i == primaryRow) {
    System.out.print(BRIGHT_CYAN + "||" + RESET + BRIGHT_GREEN + "K" + RESET);
    if (isSolved) {
        System.out.print(" " + BRIGHT_RED + "PP (Berhasil Keluar)" + RESET);
    }
} else {
    System.out.print(BRIGHT_CYAN + "||" + RESET);
}
```

```

        if (exitSide == 'L' && i == primaryRow && isSolved) {
            System.out.print(BRIGHT_RED + "(Berhasil Keluar)" + RESET);
        }

        System.out.println();
    }

    if (exitSide == 'L') {
        System.out.print(" " + BRIGHT_CYAN + "\u25aa");
    } else {
        System.out.print(BRIGHT_CYAN + "\u25aa");
    }

    for (int j = 0; j < cols; j++) {
        System.out.print("-");
    }
    System.out.println("\u25aa" + RESET);

    if (exitSide == 'B') {
        System.out.print(" ");

        for (int j = 0; j < primaryCol; j++) {
            System.out.print(" ");
        }

        if (isSolved) {
            System.out.print(BRIGHT_GREEN + "K " + BRIGHT_RED + "PP (Berhasil Keluar)" + RESET);
        } else {
            System.out.print(BRIGHT_GREEN + "K" + RESET);
        }
        System.out.println();
    }

    System.out.println(BRIGHT_RED + "■ " + RESET + "Primary Vehicle (P)  " +
        BRIGHT_GREEN + "■ " + RESET + "Exit (K)  " +
        BRIGHT_BLUE + "■ " + RESET + "Other Vehicles");
}

private static Board readInputFile(String filePath) throws IOException {
    try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
        String dimensionLine = reader.readLine();
        if (dimensionLine == null) {
            throw new IOException("File is empty");
        }

        String[] dimensions = dimensionLine.trim().split("\\s+");
        if (dimensions.length != 2) {
            throw new IOException("Invalid dimension format. Expected two numbers separated by space.");
        }
    }
}

```

```
int rows, cols;
try {
    rows = Integer.parseInt(dimensions[0]);
    cols = Integer.parseInt(dimensions[1]);
} catch (NumberFormatException e) {
    throw new IOException("Invalid dimension values. Must be integers.");
}

if (rows <= 0 || cols <= 0) {
    throw new IOException("Invalid board dimensions: must be positive");
}

String countLine = reader.readLine();
if (countLine == null) {
    throw new IOException("Unexpected end of file after dimensions");
}

int numNonPrimaryVehicles;
try {
    numNonPrimaryVehicles = Integer.parseInt(countLine.trim());
} catch (NumberFormatException e) {
    throw new IOException("Invalid number of non-primary vehicles. Must be an integer.");
}

if (numNonPrimaryVehicles < 0) {
    throw new IOException("Number of non-primary vehicles cannot be negative");
}

char[][] grid = new char[rows][cols];
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        grid[i][j] = '.';
    }
}

List<String> originalLines = new ArrayList<>();

int externalKRow = -1;
int externalKCol = -1;
char exitSide = 'R';
boolean topExit = false, leftExit = false, rightExit = false, bottomExit = false;

String line = reader.readLine();
if (line == null) throw new IOException("Unexpected end of file when reading grid");
originalLines.add(line);
```

```

if (line.trim().indexOf('K') >= 0 && line.trim().length() <= cols) {
    if (line.trim().length() > 1) {
        throw new IOException("Invalid board dimensions");
    }
    exitSide = 'T';
    externalKCol = line.indexOf('K');
    topExit = true;
    line = reader.readLine();
    if (line == null) throw new IOException("Unexpected end of file after top K row");
    originalLines.add(line);
}

int horizontalExit = 0;
for (int rowIdx = 0; rowIdx < rows; rowIdx++) {
    String currentLine = (rowIdx == 0) ? line : reader.readLine();
    if (currentLine == null) {
        throw new IOException("Invalid board dimensions");
    }

    String trimmed = currentLine.trim();
    leftExit = trimmed.startsWith("K");
    rightExit = trimmed.endsWith("K") && trimmed.length() > 1;

    int nonSpaceCount = 0;
    for (char c : currentLine.toCharArray()) {
        if (c != ' ') nonSpaceCount++;
    }
    if (leftExit) nonSpaceCount--;
    if (rightExit) nonSpaceCount--;

    if (nonSpaceCount < cols) {
        throw new IOException("Invalid board dimensions");
    }
    if (nonSpaceCount > cols) {
        throw new IOException("Invalid board dimensions");
    }

    if (rowIdx > 0) {
        originalLines.add(currentLine);
    }
}

int kIndex = currentLine.indexOf('K');

if (kIndex == 0) {
    exitSide = 'L';
    externalKRow = rowIdx;
    if (currentLine.length() > 1) {
        fillGridRow(grid, rowIdx, currentLine.substring(1));
    }
}
else if (kIndex > 0 && kIndex < cols) {
    throw new IOException("The exit (K) was found inside the grid");
}

```

```

    }

    else if (kIndex >= cols) {
        exitSide = 'R';
        externalKRow = rowIdx;
        fillGridRow(grid, rowIdx, currentLine);
    }

    else {
        fillGridRow(grid, rowIdx, currentLine);
    }

    if (leftExit) horizontalExit++;
    if (rightExit) horizontalExit++;

    if (horizontalExit > 1) {
        throw new IOException("More than one exit (K) found");
    }
}

String bottomLine = reader.readLine();
if (bottomLine != null && bottomLine.trim().length() > 0) {
    if (bottomLine.trim().indexOf('K') >= 0 && bottomLine.trim().length() <= cols) {
        if (bottomLine.trim().length() > 1) {
            throw new IOException("Invalid board dimensions");
        }
        originalLines.add(bottomLine);
        exitSide = 'B';
        externalKCol = bottomLine.indexOf('K');
        bottomExit = true;
        String afterBottomK = reader.readLine();
        while (afterBottomK != null && afterBottomK.trim().isEmpty()) {
            afterBottomK = reader.readLine();
        }
        if (afterBottomK != null) {
            throw new IOException("Invalid board dimensions");
        }
    } else {
        throw new IOException("Invalid board dimensions");
    }
}

if ((topExit && rightExit) ||
    (topExit && leftExit) ||
    (topExit && bottomExit) ||
    (rightExit && bottomExit) ||
    (leftExit && bottomExit)) {
    throw new IOException("Error: More than one exit (K) found");
}

```

```

int primaryPieceRow = -1;
int primaryPieceCol = -1;
boolean primaryIsHorizontal = false;

for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        if (grid[i][j] == 'P') {
            if (primaryPieceRow == -1) {
                primaryPieceRow = i;
                primaryPieceCol = j;
                if (j + 1 < cols && grid[i][j + 1] == 'P') {
                    primaryIsHorizontal = true;
                }
            }
        }
    }
}

if (primaryPieceRow >= 0 && primaryPieceCol >= 0) {
    if ((exitSide == 'L' || exitSide == 'R') && !primaryIsHorizontal) {
        throw new IOException("Exit orientation (horizontal) doesn't match primary piece orientation (vertical)");
    } else if ((exitSide == 'T' || exitSide == 'B') && primaryIsHorizontal) {
        throw new IOException("Exit orientation (vertical) doesn't match primary piece orientation (horizontal)");
    }
}

if ((primaryPieceRow != externalKRow) && primaryIsHorizontal) {
    throw new IOException("Exit (K) is not aligned in the same row as the primary piece (P)");
} else if ((primaryPieceCol != externalKCol) && !primaryIsHorizontal) {
    throw new IOException("Exit (K) is not aligned in the same column as the primary piece (P)");
}

boolean hasPrimary = primaryPieceRow >= 0;
if (!hasPrimary) {
    throw new IOException("No primary vehicle (P) found on the board");
}

if (externalKRow == -1 && externalKCol == -1) {
    throw new IOException("No exit (K) found at any edge");
}

int actualVehicleCount = 0;
boolean[] vehicleFound = new boolean[128];

for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        char c = grid[i][j];
        if (c != '.' && c != 'P' && c != 'K' && !vehicleFound[c]) {
            vehicleFound[c] = true;
            actualVehicleCount++;
        }
    }
}
}

```

```

        if (actualVehicleCount != numNonPrimaryVehicles) {
            System.out.println(YELLOW + "Warning: Number of non-primary vehicles in file (" +
                numNonPrimaryVehicles + ") doesn't match actual count (" +
                actualVehicleCount + ")" + RESET);
        }

        Board board = new Board(grid, rows, cols);
        board.setExternalKRow(externalKRow);
        board.setExternalKCol(externalKCol);
        board.setExitSide(exitSide);
        board.setOriginalLines(originalLines);

        return board;
    } catch (IOException e) {
        throw e;
    } catch (Exception e) {
        throw new IOException("Error reading file: " + e.getMessage(), e);
    }
}

private static void fillGridRow(char[][] grid, int rowIdx, String line) {
    int cols = grid[0].length;
    int gridColIdx = 0;

    for (int j = 0; j < cols; j++) {
        grid[rowIdx][j] = '.';
    }

    for (int j = 0; j < line.length(); j++) {
        char c = line.charAt(j);
        if (c == ' ') {
            continue;
        } else if (gridColIdx < cols) {
            grid[rowIdx][gridColIdx] = c;
            gridColIdx++;
        }
    }
}

private static String getHeuristicName(int choice) {
    switch (choice) {
        case 1:
            return "Manhattan Distance to Exit";
        case 2:
            return "Blocking Vehicles Count";
        default:
            return "Unknown";
    }
}

```

Alur Program Main.java

1. **Tampilan Banner Selamat Datang:** Program Dimulai dengan memanggil metode `printWelcomeBanner()` yang menampilkan banner selamat datang berupa ASCII art untuk nama aplikasi "RUSH HOUR PUZZLE SOLVER".
2. **Input File Puzzle:** Pengguna akan diminta untuk memasukkan path file input yang berisi konfigurasi puzzle Rush Hour. File dibaca menggunakan metode `readInputFile()` yang akan memvalidasi format file dan mengekstrak informasi seperti dimensi papan, jumlah kendaraan, dan posisi kendaraan serta pintu keluar. Jika terjadi kesalahan saat membaca file, pesan error ditampilkan dan pengguna diberi opsi untuk mencoba file lain.
3. **Menampilkan Papan Awal:** Setelah file berhasil dibaca, program memanggil metode `printBoardWithFrame()` untuk menampilkan konfigurasi papan awal dengan visualisasi berwarna. *Primary piece* ditampilkan dengan warna merah, pintu keluar dengan warna hijau, dan kendaraan lain dengan warna biru.
4. **Pemilihan Algoritma:** Program menggunakan metode `printAlgorithmMenu()` untuk menampilkan menu pilihan algoritma dalam format tabel berwarna. Pengguna diminta untuk memilih satu dari empat algoritma yang tersedia:
 - Greedy Best First Search
 - Uniform Cost Search (UCS)
 - A* Search
 - Beam Search (bonus)
5. **Pemilihan Heuristik:** Jika algoritma yang dipilih bukan UCS (yang tidak memerlukan heuristik), program memanggil metode `printHeuristicMenu()` untuk menampilkan pilihan heuristik. Pengguna dapat memilih:
 - Manhattan Distance to Exit
 - Blocking Vehicles Count
6. **Konfigurasi Tambahan:** Jika Beam Search dipilih, pengguna diminta untuk memasukkan beam width
7. **Eksekusi Algoritma:** Program menampilkan animasi loading sederhana saat menjalankan algoritma yang dipilih. Algoritma dieksekusi dan program mengukur waktu eksekusi dan jumlah node yang dikunjungi.
8. **Menampilkan Hasil:** Setelah selesai, program menampilkan hasil eksekusi, termasuk:
 - Nama algoritma dan heuristik yang digunakan
 - Waktu eksekusi dalam milidetik
 - Jumlah node yang dikunjungi
 - Apakah solusi ditemukan atau tidak

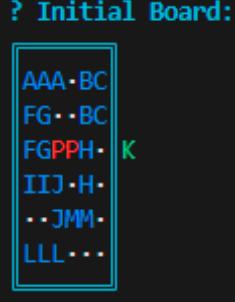
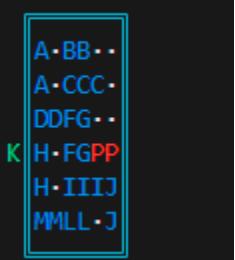
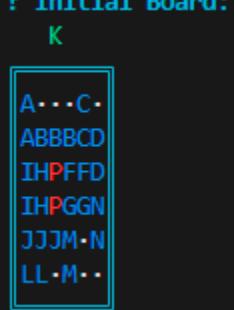
9. **Menampilkan Langkah-langkah Solusi:** Jika solusi ditemukan, program memanggil metode `solution.printStepsAnimated()` untuk menampilkan setiap langkah solusi secara berurutan. Setiap langkah menunjukkan pergerakan kendaraan dengan visualisasi papan yang diperbarui.
10. **Menyimpan Solusi:** Program memberikan opsi kepada pengguna untuk menyimpan solusi ke file teks. Jika pengguna memilih untuk menyimpan, program meminta path file output dan memanggil metode `solution.saveToFile()` untuk menulis hasil ke file.
11. **Opsi Lanjutan:** Setelah satu pengujian selesai, pengguna diberi opsi untuk:
 - Menjalankan pengujian lain dengan file puzzle yang berbeda
 - Menggunakan kembali file puzzle yang sama dengan algoritma atau heuristik berbeda
 - Mengakhiri program
12. **Penanganan Error:** Program memiliki penanganan error yang komprehensif untuk berbagai kemungkinan kesalahan, seperti input yang tidak valid, file yang tidak ditemukan, atau format file yang salah. Pesan error ditampilkan dengan warna merah.

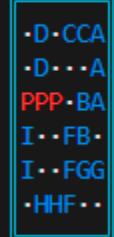
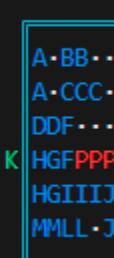
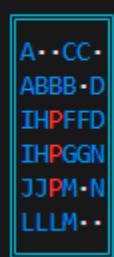
BAB V PENGUJIAN DAN HASIL PERCOBAAN

ALGORITMA PATHFINDING

5.1. Uji Coba Dengan Tangkapan Layar

5.1.1. Tabel Pengujian Algoritma Greedy Best First Search dengan Heuristic Manhattan Distance to Exit

No	Test case	Hasil pengujian
1.	? Initial Board: 	===== RESULTS ===== ? Algorithm: Greedy Best First Search ? Heuristic: Manhattan Distance to Exit ? Execution time: 1871 ms ? Nodes visited: 7754 ? SOLUTION FOUND! Number of moves: 88
2.	? Initial Board: 	===== RESULTS ===== ? Algorithm: Greedy Best First Search ? Heuristic: Manhattan Distance to Exit ? Execution time: 183 ms ? Nodes visited: 950 ? SOLUTION FOUND! Number of moves: 174
3.	? Initial Board: 	===== RESULTS ===== ? Algorithm: Greedy Best First Search ? Heuristic: Manhattan Distance to Exit ? Execution time: 63 ms ? Nodes visited: 478 ? SOLUTION FOUND! Number of moves: 69

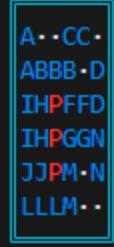
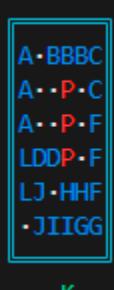
4.	<p>? Initial Board:</p> 	<p>RESULTS</p> <p>? Algorithm: Greedy Best First Search ? Heuristic: Manhattan Distance to Exit ? Execution time: 19 ms ? Nodes visited: 61</p> <p>? SOLUTION FOUND! Number of moves: 22</p>
5.	<p>? Initial Board:</p> 	<p>RESULTS</p> <p>? Algorithm: Greedy Best First Search ? Heuristic: Manhattan Distance to Exit ? Execution time: 147 ms ? Nodes visited: 999</p> <p>? SOLUTION FOUND! Number of moves: 80</p>
6.	<p>? Initial Board:</p> 	<p>RESULTS</p> <p>? Algorithm: Greedy Best First Search ? Heuristic: Manhattan Distance to Exit ? Execution time: 24 ms ? Nodes visited: 70</p> <p>? SOLUTION FOUND! Number of moves: 19</p>
7.	<p>? Initial Board:</p> 	<p>RESULTS</p> <p>? Algorithm: Greedy Best First Search ? Heuristic: Manhattan Distance to Exit ? Execution time: 34 ms ? Nodes visited: 92</p> <p>? SOLUTION FOUND! Number of moves: 46</p>

8.	<p>? Initial Board:</p>	<p>===== RESULTS =====</p> <p>? Algorithm: Greedy Best First Search ? Heuristic: Manhattan Distance to Exit ? Execution time: 18 ms ? Nodes visited: 47</p> <p>? SOLUTION FOUND! Number of moves: 20</p>
----	-------------------------	---

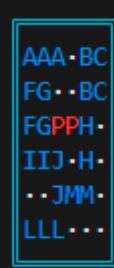
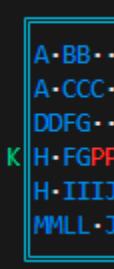
5.1.2. Tabel Pengujian Algoritma Greedy Best First Search dengan Heuristic Blocking Vehicles Count

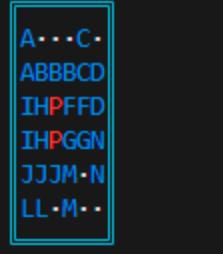
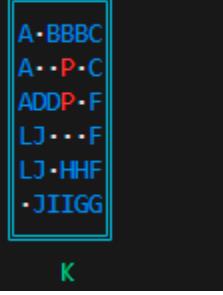
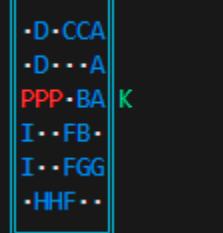
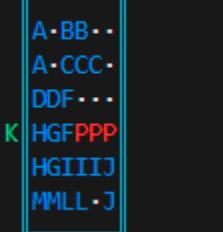
No	Test case	Hasil pengujian
1.	<p>? Initial Board:</p>	<p>===== RESULTS =====</p> <p>? Algorithm: Greedy Best First Search ? Heuristic: Blocking Vehicles Count ? Execution time: 1671 ms ? Nodes visited: 2790</p> <p>? SOLUTION FOUND! Number of moves: 48</p>
2.	<p>? Initial Board:</p>	<p>===== RESULTS =====</p> <p>? Algorithm: Greedy Best First Search ? Heuristic: Blocking Vehicles Count ? Execution time: 259 ms ? Nodes visited: 1625</p> <p>? SOLUTION FOUND! Number of moves: 86</p>

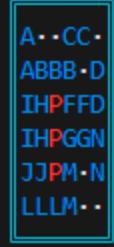
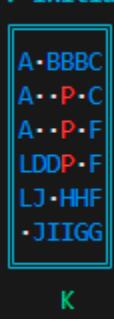
3.	<p>? Initial Board:</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td></td><td>K</td><td></td></tr> <tr><td>A</td><td>...C</td><td></td></tr> <tr><td>ABBB</td><td>CD</td><td></td></tr> <tr><td>IHP</td><td>FFD</td><td></td></tr> <tr><td>IHP</td><td>GGN</td><td></td></tr> <tr><td>JJJ</td><td>M-N</td><td></td></tr> <tr><td>LL</td><td>M..</td><td></td></tr> </table>		K		A	...C		ABBB	CD		IHP	FFD		IHP	GGN		JJJ	M-N		LL	M..		<p>————— RESULTS ————</p> <p>? Algorithm: Greedy Best First Search ? Heuristic: Blocking Vehicles Count ? Execution time: 54 ms ? Nodes visited: 501</p> <p>? SOLUTION FOUND! Number of moves: 75</p>
	K																						
A	...C																						
ABBB	CD																						
IHP	FFD																						
IHP	GGN																						
JJJ	M-N																						
LL	M..																						
4.	<p>? Initial Board:</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>A-BBBC</td><td></td></tr> <tr><td>A-..P-C</td><td></td></tr> <tr><td>ADDP-F</td><td></td></tr> <tr><td>LJ...F</td><td></td></tr> <tr><td>LJ-HHF</td><td></td></tr> <tr><td>.JIIGG</td><td></td></tr> <tr><td>K</td><td></td></tr> </table>	A-BBBC		A-..P-C		ADDP-F		LJ...F		LJ-HHF		.JIIGG		K		<p>————— RESULTS ————</p> <p>? Algorithm: Greedy Best First Search ? Heuristic: Blocking Vehicles Count ? Execution time: 22 ms ? Nodes visited: 56</p> <p>? SOLUTION FOUND! Number of moves: 20</p>							
A-BBBC																							
A-..P-C																							
ADDP-F																							
LJ...F																							
LJ-HHF																							
.JIIGG																							
K																							
5.	<p>? Initial Board:</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>.D-CCA</td><td></td></tr> <tr><td>.D-..A</td><td></td></tr> <tr><td>PPP-BA</td><td>K</td></tr> <tr><td>I-..FB-</td><td></td></tr> <tr><td>I-..FGG</td><td></td></tr> <tr><td>.HHF..</td><td></td></tr> </table>	.D-CCA		.D-..A		PPP-BA	K	I-..FB-		I-..FGG		.HHF..		<p>————— RESULTS ————</p> <p>? Algorithm: Greedy Best First Search ? Heuristic: Blocking Vehicles Count ? Execution time: 208 ms ? Nodes visited: 865</p> <p>? SOLUTION FOUND! Number of moves: 18</p>									
.D-CCA																							
.D-..A																							
PPP-BA	K																						
I-..FB-																							
I-..FGG																							
.HHF..																							
6.	<p>? Initial Board:</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>A-BB..</td><td></td></tr> <tr><td>A-CCC..</td><td></td></tr> <tr><td>DDF...</td><td></td></tr> <tr><td>K-HGFPP</td><td></td></tr> <tr><td>HGIJJ</td><td></td></tr> <tr><td>MMLL-J</td><td></td></tr> </table>	A-BB..		A-CCC..		DDF...		K-HGFPP		HGIJJ		MMLL-J		<p>————— RESULTS ————</p> <p>? Algorithm: Greedy Best First Search ? Heuristic: Blocking Vehicles Count ? Execution time: 19 ms ? Nodes visited: 32</p> <p>? SOLUTION FOUND! Number of moves: 13</p>									
A-BB..																							
A-CCC..																							
DDF...																							
K-HGFPP																							
HGIJJ																							
MMLL-J																							

7.	<p>? Initial Board:</p> 	<p>RESULTS</p> <p>? Algorithm: Greedy Best First Search ? Heuristic: Blocking Vehicles Count ? Execution time: 16 ms ? Nodes visited: 18</p> <p>? SOLUTION FOUND! Number of moves: 5</p>
8.	<p>? Initial Board:</p> 	<p>RESULTS</p> <p>? Algorithm: Greedy Best First Search ? Heuristic: Blocking Vehicles Count ? Execution time: 19 ms ? Nodes visited: 29</p> <p>? SOLUTION FOUND! Number of moves: 13</p>

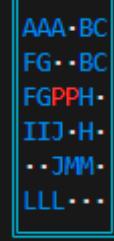
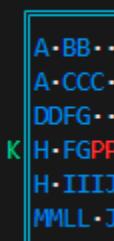
5.1.3. Tabel Pengujian Algoritma UCS (Uniform Cost Search)

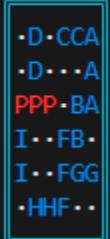
No	Test case	Hasil pengujian
1.	<p>? Initial Board:</p> 	<p>RESULTS</p> <p>? Algorithm: Uniform Cost Search (UCS) ? Execution time: 1542 ms ? Nodes visited: 11178</p> <p>? SOLUTION FOUND! Number of moves: 27</p>
2.	<p>? Initial Board:</p> 	<p>RESULTS</p> <p>? Algorithm: Uniform Cost Search (UCS) ? Execution time: 306 ms ? Nodes visited: 4126</p> <p>? SOLUTION FOUND! Number of moves: 41</p>

3.	<p>? Initial Board:</p> 	<p>————— RESULTS ————</p> <p>? Algorithm: Uniform Cost Search (UCS) ? Execution time: 79 ms ? Nodes visited: 678</p> <p>? SOLUTION FOUND! Number of moves: 31</p>
4.	<p>? Initial Board:</p> 	<p>————— RESULTS ————</p> <p>? Algorithm: Uniform Cost Search (UCS) ? Execution time: 36 ms ? Nodes visited: 264</p> <p>? SOLUTION FOUND! Number of moves: 16</p>
5.	<p>? Initial Board:</p> 	<p>————— RESULTS ————</p> <p>? Algorithm: Uniform Cost Search (UCS) ? Execution time: 357 ms ? Nodes visited: 1808</p> <p>? SOLUTION FOUND! Number of moves: 6</p>
6.	<p>? Initial Board:</p> 	<p>————— RESULTS ————</p> <p>? Algorithm: Uniform Cost Search (UCS) ? Execution time: 1214 ms ? Nodes visited: 2976</p> <p>? SOLUTION FOUND! Number of moves: 9</p>

7.	<p>? Initial Board:</p> 	<p>===== RESULTS =====</p> <p>? Algorithm: Uniform Cost Search (UCS)</p> <p>? Execution time: 28 ms</p> <p>? Nodes visited: 109</p> <p>? SOLUTION FOUND! Number of moves: 4</p>
8.	<p>? Initial Board:</p> 	<p>===== RESULTS =====</p> <p>? Algorithm: Uniform Cost Search (UCS)</p> <p>? Execution time: 27 ms</p> <p>? Nodes visited: 109</p> <p>? SOLUTION FOUND! Number of moves: 10</p>

5.1.4. Tabel Pengujian Algoritma A* dengan Heuristic Manhattan Distance to Exit

No	Test case	Hasil pengujian
1.	<p>? Initial Board:</p> 	<p>===== RESULTS =====</p> <p>? Algorithm: A* Search</p> <p>? Heuristic: Manhattan Distance to Exit</p> <p>? Execution time: 1566 ms</p> <p>? Nodes visited: 10205</p> <p>? SOLUTION FOUND! Number of moves: 28</p>
2.	<p>? Initial Board:</p> 	<p>===== RESULTS =====</p> <p>? Algorithm: A* Search</p> <p>? Heuristic: Manhattan Distance to Exit</p> <p>? Execution time: 310 ms</p> <p>? Nodes visited: 4010</p> <p>? SOLUTION FOUND! Number of moves: 41</p>

3.	<p>? Initial Board:</p> 	<p>RESULTS</p> <p>? Algorithm: A* Search ? Heuristic: Manhattan Distance to Exit ? Execution time: 88 ms ? Nodes visited: 689</p> <p>? SOLUTION FOUND! Number of moves: 31</p>
4.	<p>? Initial Board:</p> 	<p>RESULTS</p> <p>? Algorithm: A* Search ? Heuristic: Manhattan Distance to Exit ? Execution time: 27 ms ? Nodes visited: 193</p> <p>? SOLUTION FOUND! Number of moves: 17</p>
5.	<p>? Initial Board:</p> 	<p>RESULTS</p> <p>? Algorithm: A* Search ? Heuristic: Manhattan Distance to Exit ? Execution time: 108 ms ? Nodes visited: 520</p> <p>? SOLUTION FOUND! Number of moves: 7</p>
6.	<p>? Initial Board:</p> 	<p>RESULTS</p> <p>? Algorithm: A* Search ? Heuristic: Manhattan Distance to Exit ? Execution time: 392 ms ? Nodes visited: 1189</p> <p>? SOLUTION FOUND! Number of moves: 9</p>

7.

? Initial Board:

K
A..CC.
ABBB.D
IHPFFD
IHPGGN
JJPM.N
LLL.M..

===== RESULTS =====

? Algorithm: A* Search

? Heuristic: Manhattan Distance to Exit

? Execution time: 24 ms

? Nodes visited: 54

? SOLUTION FOUND! Number of moves: 4

8.

? Initial Board:

A.BBBC
A..P.C
A..P.F
LDDP.F
LJ.HHF
.JIIGG

K

===== RESULTS =====

? Algorithm: A* Search

? Heuristic: Manhattan Distance to Exit

? Execution time: 16 ms

? Nodes visited: 68

? SOLUTION FOUND! Number of moves: 11

BAB VI IMPLEMENTASI BONUS

Dalam program tugas kecil 3 mengenai penyelesaian puzzle rush hour menggunakan algoritma *pathfinding*, kami berhasil mengimplementasikan algoritma *Beam Search* sebagai salah satu algoritma alternatif dan mengimplementasikan dua tipe *heuristic*, yaitu jarak Manhattan ke exit dan *blocking pieces*.

6.1. Algoritma Alternatif (Beam Search)

6.1.1. Kode Pengimplementasian Beam Search

```
100,2 days ago | author (100)
✓ import java.util.ArrayList;
  import java.util.Comparator;
  import java.util.HashSet;
  import java.util.List;
  import java.util.PriorityQueue;
  import java.util.Set;

You, 2 days ago | 1 author (You)
✓ public class BeamSearch extends Solver {
    private int beamWidth;

    ✓ public BeamSearch(Board initialBoard, int heuristicChoice, int beamWidth) {
        super(initialBoard, heuristicChoice);
        this.beamWidth = beamWidth;
    }

    @Override
    ✓ public Solution solve() {
        // Menyimpan papan yang sudah dikunjungi agar tidak duplicate
        Set<Board> closedSet = new HashSet<>();

        // Pencarian dari level pertama
        List<Node> currentLevel = new ArrayList<>();
        currentLevel.add(new Node(initialBoard, heuristicChoice));

        while (!currentLevel.isEmpty()) {
            PriorityQueue<Node> nextLevel = new PriorityQueue<>(Comparator.comparingInt(Node::getHeuristic));

            // Iterasi seluruh node di level ini
            for (Node current : currentLevel) {
                nodesVisited++;

                // Jika sudah mencapai solusi, kembalikan path-nya
                if (current.getBoard().isSolved()) {
                    return new Solution(current.getPath());
                }

                closedSet.add(current.getBoard());
            }
        }
    }
}
```

```

        // Hasilkan semua possible moves
        List<int[]> legalMoves = current.getBoard().getLegalMoves();
        for (int[] move : legalMoves) {
            int vehicleIdx = move[0];
            int moveAmount = move[1];

            Board newBoard = current.getBoard().applyMove(vehicleIdx, moveAmount);

            // Jika papan baru sudah pernah dikunjungi, jangan diproses lagi
            if (closedSet.contains(newBoard)) continue;

            Node successor = new Node(current, newBoard, vehicleIdx, moveAmount, heuristicChoice);
            nextLevel.add(successor);
        }
    }

    // Seleksi node terbaik untuk level berikutnya
    currentLevel = new ArrayList<>();
    for (int i = 0; i < beamWidth && !nextLevel.isEmpty(); i++) {
        currentLevel.add(nextLevel.poll());
    }
}

// Jika tidak ditemukan solusi, kembalikan null
return null;
}

@Override
public String getName() {
    return "Beam Search (width=" + beamWidth + ")";
}
}

```

6.1.2. Penjelasan Kode Beam Search

Algoritma *Beam Search* merupakan algoritma pengembangan dari Best-First Search. Namun, berbeda dengan algoritma seperti A* atau Greedy yang mempertahankan semua node terbuka selama proses pencarian, Beam Search hanya menyimpan sejumlah terbatas node terbaik pada setiap level eksplorasi, berdasarkan nilai heuristik. Batas jumlah node yang dipertahankan ini dikenal biasanya sebagai *beam width*. Pendekatan ini memungkinkan Beam Search untuk menghemat penggunaan memori secara signifikan dan mempercepat proses pencarian, terutama pada ruang pencarian yang sangat besar. Namun, karena algoritma ini membuang node di luar beam, algoritma ini tidak menjamin solusi yang ditemukan merupakan solusi optimal.

Kode implementasi utama algoritma *Beam Search* terdapat dalam metode `solve()`. Pada awalnya, `currentLevel` diinisialisasi dengan Node yang dibentuk dari papan awal (`initialBoard`), kemudian dilakukan eksplorasi level demi level. Pada setiap iterasi, seluruh node pada `currentLevel` diperluas. Untuk setiap node `current`, dilakukan pemeriksaan apakah konfigurasi papan saat ini telah mencapai solusi melalui `current.getBoard().isSolved()`. Jika ya, maka

solusi dikembalikan dalam bentuk path. Jika belum, maka semua legal moves dari papan tersebut diambil menggunakan `current.getBoard().getLegalMoves()`, kemudian masing-masing gerakan diterapkan dengan `applyMove()`. Node hasil gerakan disimpan sebagai objek Node baru dan dimasukkan ke dalam *priority queue* `nextLevel`. Priority queue ini diurutkan berdasarkan nilai heuristik Node melalui `Comparator.comparingInt(Node::getHeuristic)`.

Terakhir, setelah semua successor dari level saat ini diproses, hanya node sejumlah input `beamWidth` dengan heuristik terbaik yang disimpan ke `currentLevel` untuk level selanjutnya. Hal ini dilakukan melalui loop yang mengambil elemen dari `nextLevel` menggunakan `poll()`. Untuk menghindari eksplorasi redundan, setiap konfigurasi papan yang telah dieksplorasi dimasukkan ke `closedSet` agar tidak diproses kembali.

6.2. Heuristik Alternatif (Jarak Manhattan ke Exit dan Blocking Pieces)

6.2.1. Kode Pengimplementasian Heuristik

```
// Heuristik 1: jarak Manhattan ke exit
public int manhattanDistanceToExit() {
    switch (exitSide) {
        case 'R':
            if (primaryPiece.isHorizontal()) {
                int endCol = primaryPiece.getEndCol();
                int rowDist = Math.abs(primaryPiece.getRow() - exitRow);
                int colDist = cols - 1 - endCol;
                return rowDist + colDist;
            }
            break;
        case 'L':
            if (primaryPiece.isHorizontal()) {
                int rowDist = Math.abs(primaryPiece.getRow() - exitRow);
                int colDist = primaryPiece.getCol();
                return rowDist + colDist;
            }
            break;
        case 'T':
            if (!primaryPiece.isHorizontal()) {
                int rowDist = primaryPiece.getRow();
                int colDist = Math.abs(primaryPiece.getCol() - exitCol);
                return rowDist + colDist;
            }
            break;
        case 'B':
            if (!primaryPiece.isHorizontal()) {
                int endRow = primaryPiece.getEndRow();
                int rowDist = rows - 1 - endRow;
                int colDist = Math.abs(primaryPiece.getCol() - exitCol);
                return rowDist + colDist;
            }
            break;
    }
    return rows + cols;
}
```

```

// Heuristik 2: blocking pieces
public int blockingPiecesCount() {
    int count = 0;
    switch (exitSide) {
        case 'R':
            if (primaryPiece.isHorizontal()) {
                int row = primaryPiece.getRow();
                int startCol = primaryPiece.getEndCol() + 1;
                int endCol = cols - 1;
                for (int c = startCol; c <= endCol; c++) {
                    if (grid[row][c] != '.' && grid[row][c] != 'K') {
                        count++;
                        while (c + 1 < cols && grid[row][c] == grid[row][c + 1]) {
                            c++;
                        }
                    }
                }
            }
            break;
        case 'L':
            if (primaryPiece.isHorizontal()) {
                int row = primaryPiece.getRow();
                int startCol = 0;
                int endCol = primaryPiece.getCol() - 1;
                for (int c = startCol; c <= endCol; c++) {
                    if (grid[row][c] != '.' && grid[row][c] != 'K') {
                        count++;
                        while (c + 1 <= endCol && grid[row][c] == grid[row][c + 1]) {
                            c++;
                        }
                    }
                }
            }
            break;
        case 'T':
            if (!primaryPiece.isHorizontal()) {
                int col = primaryPiece.getCol();
                int startRow = 0;
                int endRow = primaryPiece.getRow() - 1;
                for (int r = startRow; r <= endRow; r++) {
                    if (grid[r][col] != '.' && grid[r][col] != 'K') {
                        count++;
                        while (r + 1 <= endRow && grid[r][col] == grid[r + 1][col]) {
                            r++;
                        }
                    }
                }
            }
            break;
        case 'B':
            if (!primaryPiece.isHorizontal()) {
                int col = primaryPiece.getCol();
                int startRow = primaryPiece.getEndRow() + 1;
                int endRow = rows - 1;
                for (int r = startRow; r <= endRow; r++) {
                    if (grid[r][col] != '.' && grid[r][col] != 'K') {
                        count++;
                        while (r + 1 < rows && grid[r][col] == grid[r + 1][col]) {
                            r++;
                        }
                    }
                }
            }
            break;
    }
    return count;
}

```

```
// Opsi bagi pengguna untuk memilih heuristic
public int getHeuristicValue(int heuristicChoice) {
    switch (heuristicChoice) {
        case 1:
            return manhattanDistanceToExit();
        case 2:
            return blockingPiecesCount();
        default:
            return 0;
    }
}
```

6.2.2. Penjelasan Kode Heuristik

Dalam program yang dibuat, kami mengimplementasikan dua tipe heuristik, yaitu Jarak Manhattan ke Exit dan Blocking Pieces. Pilihan heuristik yang dibuat dikendalikan melalui parameter `heuristicChoice` yang diteruskan ke seluruh algoritma pencarian, yaitu Greedy, A*, Beam Search. Evaluasi heuristik dilakukan melalui metode `getHeuristicValue(int heuristicChoice)` di kelas Board.

Heuristik pertama yang diimplementasikan adalah Manhattan Distance to Exit, yang diimplementasikan pada metode `manhattanDistanceToExit()`. Fungsi ini menghitung jarak horizontal atau vertikal minimum dari ujung primary piece menuju pintu keluar berdasarkan posisi dan orientasi primary piece serta sisi tempat pintu keluar berada (kanan, kiri, atas, atau bawah). Untuk exit di kanan ('R'), misalnya, dihitung dari kolom terakhir primary piece ke tepi kanan papan. Nilai yang dikembalikan merupakan penjumlahan dari selisih posisi baris dan kolom saat ini terhadap posisi pintu keluar. Dikarenakan heuristik ini tidak memperhitungkan kendaraan penghalang, maka nilainya selalu berada pada batas bawah dari biaya sebenarnya, menjadikannya heuristik yang *admissible*.

Heuristik kedua adalah Blocking Pieces Count. Heuristik ini diimplementasikan dalam metode `blockingPiecesCount()`. Fungsi ini menghitung jumlah kendaraan unik yang secara langsung menghalangi jalur primary piece menuju pintu keluar. Proses perhitungannya dilakukan dengan memindai seluruh sel pada jalur *primary piece* hingga ke pintu keluar dan menghitung

kendaraan non-'' yang berbeda. Untuk menghindari perhitungan ganda pada kendaraan yang menempati lebih dari satu sel, loop while digunakan untuk melewati sel-sel yang masih merupakan bagian dari kendaraan yang sama. Nilai ini tidak akan melebih-lebihkan biaya minimum yang sebenarnya karena minimal satu langkah diperlukan untuk setiap kendaraan penghalang, sehingga heuristik ini juga *admissible*.

Kedua heuristik ini diintegrasikan dalam metode `getHeuristicValue(int heuristicChoice)`, yang dipanggil saat menghitung nilai heuristik. Saat pengguna memilih heuristik melalui input, yaitu 1 untuk Manhattan Distance dan 2 untuk Blocking Pieces), maka program akan mengatur nilai heuristiknya sesuai pilihan tersebut, yang akan digunakan dalam pengurutan prioritas di *priority queue* masing-masing algoritma seperti A*, Greedy Best First Search, dan Beam Search.

LAMPIRAN

Pranala Repository:

https://github.com/csans13/Tucil3_13523157_13523159.git

Tabel Checklist Spesifikasi Program:

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	✓	