

Laporan Tugas Kecil 3 IF2211 Strategi Algoritma

Sem II tahun 2024/2025

“Pemanfaatan Algoritma *Uniform Cost Search, Greedy Best First Search* dan *A Star* untuk Pencarian Jalur dalam Permainan Rush Hour”



Disusun oleh :

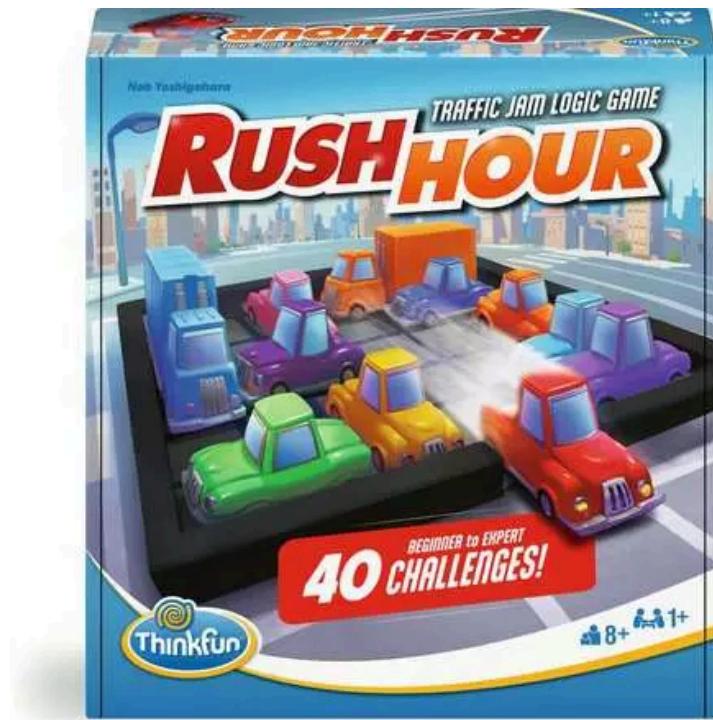
Guntara Hambali - 13523114

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2025

Daftar Isi

Daftar Isi.....	2
BAB I : Deskripsi Tugas.....	3
BAB II : Landasan Teori.....	7
2.1 Algoritma Uniform Cost Search.....	7
2.2 Algoritma Greedy Best First Search.....	8
2.3 Algoritma A Star.....	9
BAB III : Analisis Algoritma.....	10
3.1 Perbandingan Algoritma Uniform Cost Search, Greedy Best First Search, dan A Star	10
BAB IV : Source Code Program.....	12
4.1 Algoritma Utama.....	12
4.2 Kelas Board.....	14
4.3 Kelas Piece.....	17
4.4 Heuristik (Blocking Cars)	20
4.5 Kelas TreeNode.....	21
BAB V : Pengujian.....	23
5.1 Pengujian Algoritma UCS.....	24
5.2 Pengujian Algoritma GBFS.....	27
5.3 Pengujian Algoritma A Star.....	29
BAB VI : Analisis Hasil Pengujian.....	31
6.1 Analisis Algoritma UCS.....	31
6.2 Analisis Algoritma GBFS.....	31
6.3 Analisis Algoritma A Star.....	31
BAB VII : Bonus.....	33
7.1 GUI.....	33
7.2 Heuristik Tambahan Jarak Manhattan.....	33
BAB VIII : Lampiran.....	37

BAB I : Deskripsi Tugas



Gambar 1. Rush Hour Puzzle

(Sumber: <https://www.thinkfun.com/en-US/products/educational-games/rush-hour-76582>)

Rush Hour adalah sebuah permainan puzzle logika berbasis grid yang menantang pemain untuk menggeser kendaraan di dalam sebuah kotak (biasanya berukuran 6x6) agar mobil utama (biasanya berwarna merah) dapat keluar dari kemacetan melalui pintu keluar di sisi papan. Setiap kendaraan hanya bisa bergerak lurus ke depan atau ke belakang sesuai dengan orientasinya (horizontal atau vertikal), dan tidak dapat berputar. Tujuan utama dari permainan ini adalah memindahkan mobil merah ke pintu keluar dengan jumlah langkah seminimal mungkin.

Komponen penting dari permainan Rush Hour terdiri dari:

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.

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.

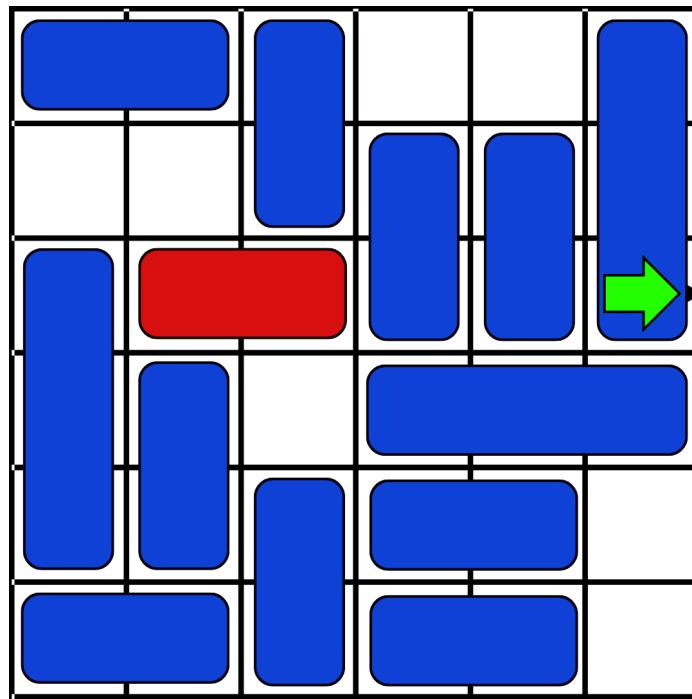
Primary Piece – Primary piece adalah kendaraan utama yang harus dikeluarkan dari papan (biasanya berwarna merah). Hanya boleh terdapat satu primary piece.

Pintu Keluar – Pintu keluar adalah tempat primary piece dapat digerakkan keluar untuk menyelesaikan permainan

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.

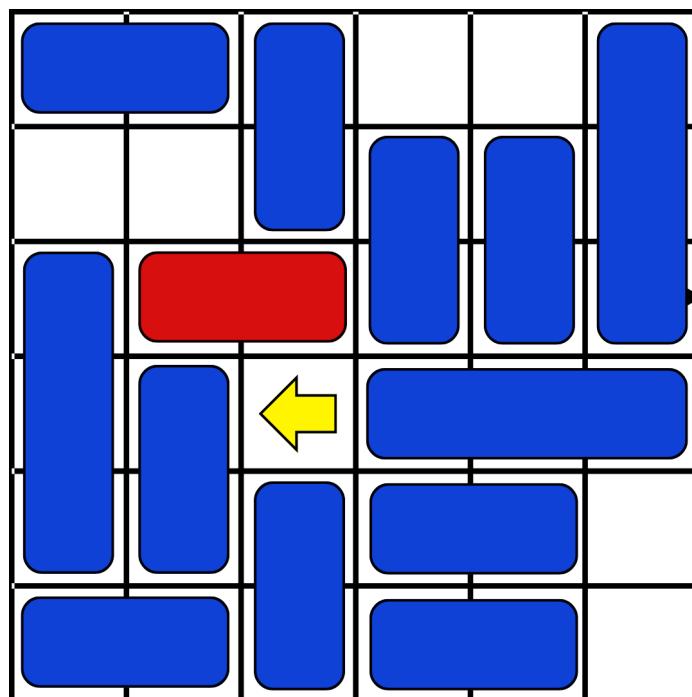
Ilustrasi kasus:

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.

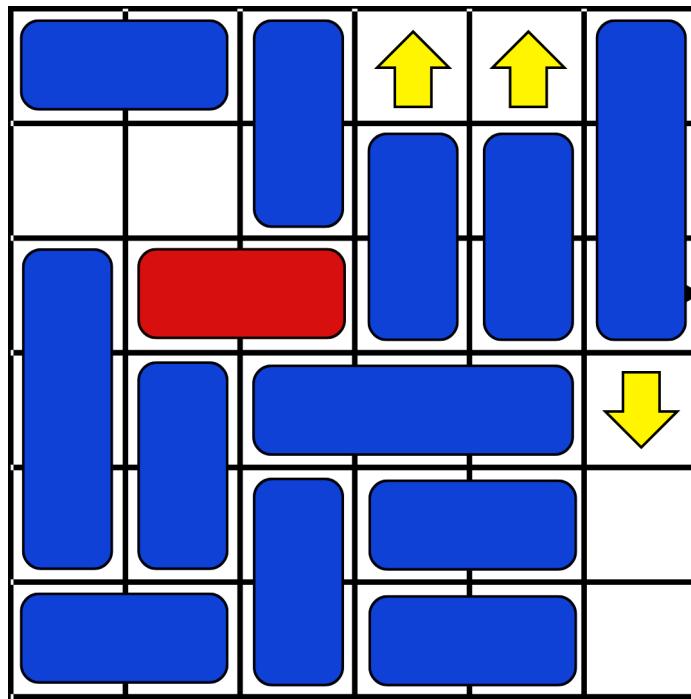


Gambar 2. Awal Permainan Game Rush Hour

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

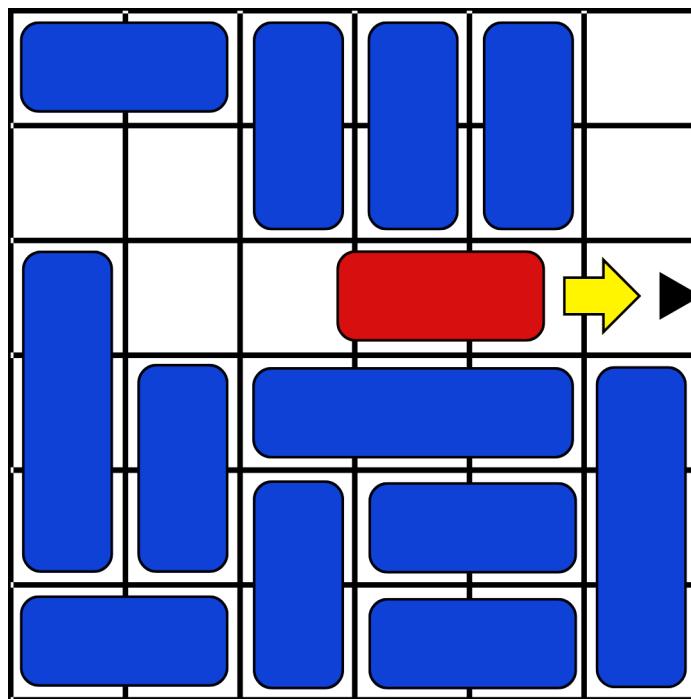


Gambar 3. Gerakan Pertama Game Rush Hour



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

BAB II : Landasan Teori

2.1 Algoritma Uniform Cost Search

Algoritma Uniform Cost Search (UCS) adalah algoritma yang digunakan untuk mencari jalan/rute (pathfinding) pada suatu state permasalahan. UCS sebenarnya mirip dengan BFS, yang membedakan ialah UCS mempertimbangkan biaya/cost dari suatu simpul n ke simpul tertentu dan mencari cost yang paling minimal. Secara matematis dinyatakan dengan:

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

dimana $f(n)$ adalah fungsi umum yang akan dikomparasikan dan $g(n)$ adalah biaya/cost yang telah dikeluarkan dari simpul awal ke simpul n. Dalam tugas ini, saya menyatakan biaya yang telah ditempuh untuk mencapai simpul n ($g(n)$) sebagai jumlah langkah yang harus ditempuh agar mencapai konfigurasi papan pada simpul tersebut. Berikut adalah algoritma yang saya tempuh dalam implementasi UCS untuk permainan Rush Hour:

1. Baca konfigurasi papan dari file .txt.
2. Buat simpul akar (TreeNode) dengan: cost = 0
3. Masukkan simpul akar ke dalam antrian prioritas (priority queue) berdasarkan cost-nya ($f(n) = g(n)$).
4. Selama antrian tidak kosong, lakukan langkah 5-8
5. Ambil simpul dengan $g(n)$ terkecil dari antrian.
6. Jika simpul sudah pernah dikunjungi, lewati.
7. Tandai simpul sebagai dikunjungi.
8. Jika simpul mencapai goal , cetak solusinya dan selesai.
9. Jika belum goal, lakukan langkah 9-14:
10. Bangkitkan semua anak yang mungkin dari board sekarang (kita sebut suksesor).
11. Untuk setiap suksesor yang belum dikunjungi:
12. Hitung cost dengan cara = parent.cost + 1
13. Buat simpul baru dan tambahkan ke antrian.
14. Jika antrian kosong dan goal tidak ditemukan, laporan bahwa tidak ada solusi.

2.2 Algoritma Greedy Best First Search

Algoritma Greedy Best First Search (GBFS) adalah algoritma yang digunakan untuk mencari jalan/rute (pathfinding) pada suatu state permasalahan. Disebut Greedy Best First Search memang karena pencarian node dilakukan secara greedy dengan mencari biaya heuristik paling sedikit. Berbeda dengan UCS yang mempertimbangkan $g(n)$. Secara matematis, fungsi perbandingan untuk algoritma A Star adalah:

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

dimana $f(n)$ adalah fungsi umum yang akan dikomparasikan dan $h(n)$ adalah biaya heuristik yang berarti estimasi biaya untuk mencapai simpul tujuan dari simpul n . Dalam tugas ini, saya menyatakan biaya heuristik ($h(n)$) dengan banyaknya piece yang menghalangi primary piece. Berikut adalah algoritma yang saya tempuh dalam implementasi GBFS untuk permainan Rush Hour:

1. Baca konfigurasi papan dari file .txt.
2. Buat simpul akar (TreeNode) dengan: cost = 0
3. Masukkan simpul akar ke dalam antrian prioritas (priority queue) berdasarkan cost-nya ($f(n) = h(n)$).
4. Selama antrian tidak kosong, lakukan langkah 5-8
5. Ambil simpul dengan $h(n)$ terkecil dari antrian.
6. Jika simpul sudah pernah dikunjungi, lewati.
7. Tandai simpul sebagai dikunjungi.
8. Jika simpul mencapai goal , cetak solusinya dan selesai.
9. Jika belum goal, lakukan langkah 9-14:
10. Bangkitkan semua anak yang mungkin dari board sekarang (kita sebut suksesor).
11. Untuk setiap suksesor yang belum dikunjungi:
12. Hitung heuristik dengan jumlah piece yang menghalangi atau dengan jarak Manhattan
13. Buat simpul baru dan tambahkan ke antrian prioritas (sama seperti langkah 3).
14. Jika antrian kosong dan goal tidak ditemukan, laporkan bahwa tidak ada solusi.

2.3 Algoritma A Star

Algoritma A Star juga adalah algoritma yang digunakan untuk mencari jalan/rute (pathfinding) pada suatu state permasalahan. A Star dapat diipandang sebagai gabungan dari 2 algoritma sebelumnya dalam hal fungsi yang digunakan untuk menghitung biaya. Secara matematis, dinyatakan dengan:

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

dimana $f(n)$ adalah fungsi umum yang akan dikomparasikan, $g(n)$ adalah biaya/cost yang telah dikeluarkan dari simpul awal ke simpul n , dan $h(n)$ adalah biaya heuristik. Berikut adalah algoritma yang saya tempuh dalam implementasi A Star untuk permainan Rush Hour:

1. Baca konfigurasi papan dari file .txt.
2. Buat simpul akar (TreeNode) dengan: cost = 0
3. Masukkan simpul akar ke dalam antrian prioritas (priority queue) berdasarkan cost-nya ($f(n) = g(n) + h(n)$).
4. Selama antrian tidak kosong, lakukan langkah 5-8
5. Ambil simpul dengan $g(n)$ terkecil dari antrian.
6. Jika simpul sudah pernah dikunjungi, lewati.
7. Tandai simpul sebagai dikunjungi.
8. Jika simpul mencapai goal , cetak solusinya dan selesai.
9. Jika belum goal, lakukan langkah 9-14:
10. Bangkitkan semua anak yang mungkin dari board sekarang (kita sebut suksesor).
11. Untuk setiap suksesor yang belum dikunjungi:
12. Hitung cost dengan cara = parent.cost + 1
13. Hitung heuristik dengan jumlah piece yang menghalangi atau dengan jarak Manhattan
14. Buat simpul baru dan tambahkan ke antrian berdasarkan jumlah dari biaya dan biaya heuristik (sama seperti langkah 3).
15. Jika antrian kosong dan goal tidak ditemukan, laporkan bahwa tidak ada solusi.

BAB III : Analisis Algoritma

3.1 Perbandingan Algoritma Uniform Cost Search, Greedy Best First Search, dan A Star

Algoritma UCS, Greedy Best First Search (GBFS), dan A Star merupakan 3 algoritma pencarian rute yang serupa, namun tak sama. Dalam pencarian rute, ketiganya juga memperhitungkan kuantitas dari biaya yang harus ditempuh untuk melewati suatu jalur. Ini tentu berbeda dari algoritma BFS dan DFS yang hanya melihat urutan pemrosesan dari simpul-simpul yang dikunjungi. Ketiga algoritma tersebut sebenarnya bisa diklasifikasikan lagi menjadi 2. UCS termasuk blind search, sementara GBFS dan A Star termasuk informed search.

Dasar pengklasifikasian tersebut adalah mengenai penghitungan biaya total dalam menempuh suatu rute ($f(n)$). Pada UCS, biaya total dihitung hanya berdasarkan biaya sejauh ini untuk mencapai simpul n ($g(n)$) oleh karena itu, secara matematis dinyatakan dengan $f(n) = g(n)$. Sementara itu, untuk GBFS, biaya total dihitung hanya berdasarkan estimasi heuristik untuk mencapai simpul tujuan dari simpul n ($h(n)$). Estimasi ini bersifat heuristik karena merupakan estimasi kasar atas biaya dari simpul n ke simpul tujuan. Secara matematis, dilambangkan dengan $f(n) = h(n)$. Di lain sisi, A Star merupakan penggabungan antara biaya sejauh ini untuk sampai ke simpul n ditambah biaya heuristik dari simpul n ke tujuan. Secara matematis dilambangkan dengan $f(n) = g(n) + h(n)$. Dari sini, dapat dilihat bahwa algoritma GBFS dan A Star sama-sama mempertimbangkan heuristik sebagai komponen biaya totalnya. Oleh sebab itulah, keduanya diklasifikasikan sebagai informed search, sementara UCS sebagai blind search.

Pada pencarian rute permainan Rush Hour ini, dapat diterapkan algoritma UCS dengan cost $g(n)$ dihitung sebagai jumlah langkah yang diperlukan untuk mencapai simpul n. Artinya, satu gerakan akan menambahkan cost sebanyak satu satuan. Atau jika dinyatakan secara matematis, menjadi

$$g(n) = g(n-1) + 1$$

Dimana $g(n-1)$ merupakan simpul sebelumnya dari simpul n (atau yang lebih tepat adalah parent dari simpul n). Rumusan ini dipilih karena kita ingin mencari sesedikit mungkin langkah yang bisa ditempuh untuk mencapai tujuan. Akan tetapi, dengan penggunaan fungsi biaya yang seperti ini, sebenarnya membuat algoritma UCS yang diterapkan sama saja seperti algoritma Best First Search BFS).

Seperti yang kita ketahui, BFS mencari jalur dengan step paling sedikit, sementara step paling sedikit belum tentu menghasilkan biaya paling sedikit. Saat membutuhkan biaya paling sedikit, di situ lah UCS dipakai. Akan tetapi, dalam rumusan biaya tadi, kita menghitung tiap langkah (step) sebagai 1 satuan cost. Ini berarti kita menyamakan cost dengan step (cost = step) yang artinya algoritma UCS yang diterapkan akan sama saja dengan algoritma BFS.

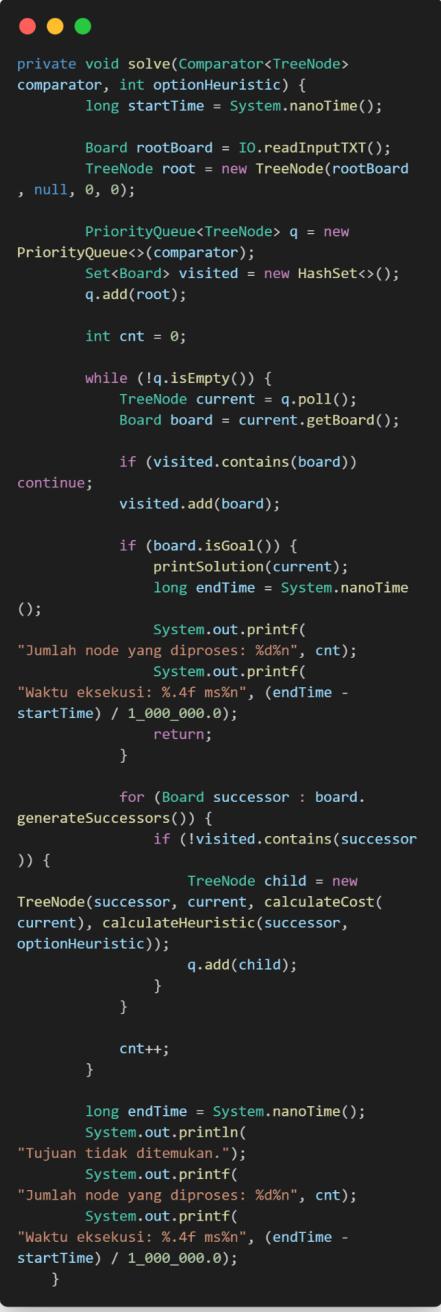
Pada pencarian rute permainan Rush Hour dengan algoritma GBFS dan A Star, saya menggunakan 2 heuristik, yaitu banyak piece yang menghalangi piece utama dan jarak manhattan. Kedua heuristik ini bersifat admissible. Banyak piece yang menghalangi piece utama bersifat admissible karena tidak mungkin banyak piece yang ada melebihi jumlah langkah sebenarnya untuk memindahkan piece utama. Banyak piece yang menghalangi maksimal akan sama dengan jumlah langkah yang sebenarnya. Sementara itu, untuk manhattan distance, definisinya adalah total jarak yang dibutuhkan oleh mobil-mobil penghalang untuk berpindah (tidak menghalangi lagi) piece utama. Tentu ini tidak akan melebihi banyak aslinya sehingga ini juga admissible.

Ketika kita algoritma A Star dan UCS dibandingkan, secara teoritis, algoritma A Star akan lebih efisien daripada algoritma UCS. Hal ini dapat terjadi tentu berkaitan dengan fungsi biayanya. Algoritma A Star dengan $f(n) = g(n) + h(n)$ memiliki informasi lebih banyak dibandingkan $f(n) = g(n)$ milik UCS. Hal ini dapat mempersempit ruang pencarian terhadap simpul-simpul yang ada sehingga simpul-simpul yang dikunjungi secara total akan berkurang. Akan tetapi, efisiensi yang seperti itu berlaku hanya jika fungsi heuristik yang digunakan bersifat admissible dan konsisten

Selain itu, algoritma GBFS juga perlu diperhatikan kaitannya dengan tujuan dari pencarian solusi optimal pada permainan Rush Hour ini, yaitu mencari langkah paling sedikit untuk piece utama bisa keluar. Sayangnya, GBFS tidak menjamin solusi optimal. Ini terjadi karena GNFS hanya memperhitungkan heuristik sebagai komponen perhitungan biaya totalnya. Padahal, seperti yang kita ketahui, heuristik hanya merupakan estimasi yang sifatnya intuitif saja.

BAB IV : Source Code Program

4.1 Algoritma Utama

No	Snapshot kode	Penjelasan
1.	 <pre>private void solve(Comparator<TreeNode> comparator, int optionHeuristic) { long startTime = System.nanoTime(); Board rootBoard = IO.readInputTXT(); TreeNode root = new TreeNode(rootBoard , null, 0, 0); PriorityQueue<TreeNode> q = new PriorityQueue<>(comparator); Set<Board> visited = new HashSet<>(); q.add(root); int cnt = 0; while (!q.isEmpty()) { TreeNode current = q.poll(); Board board = current.getBoard(); if (visited.contains(board)) continue; visited.add(board); if (board.isGoal()) { printSolution(current); long endTime = System.nanoTime (); System.out.printf("Jumlah node yang diproses: %d\n", cnt); System.out.printf("Waktu eksekusi: %.4f ms\n", (endTime - startTime) / 1_000_000.0); return; } for (Board successor : board. generateSuccessors()) { if (!visited.contains(successor)) { TreeNode child = new TreeNode(successor, current, calculateCost(current), calculateHeuristic(successor, optionHeuristic)); q.add(child); } } cnt++; } long endTime = System.nanoTime(); System.out.println("Tujuan tidak ditemukan."); System.out.printf("Jumlah node yang diproses: %d\n", cnt); System.out.printf("Waktu eksekusi: %.4f ms\n", (endTime - startTime) / 1_000_000.0); }</pre>	Prosedur ini merupakan prosedur utama untuk mencari rute menggunakan 3 algoritma. Ketiga algoritma menggunakan fungsi solve yang sama. Perbedaannya, adalah pada argumen Comparator untuk membandingkan prioritas dalam antrian prioritas berdasarkan f(n)

2.

```
import java.util.Comparator;

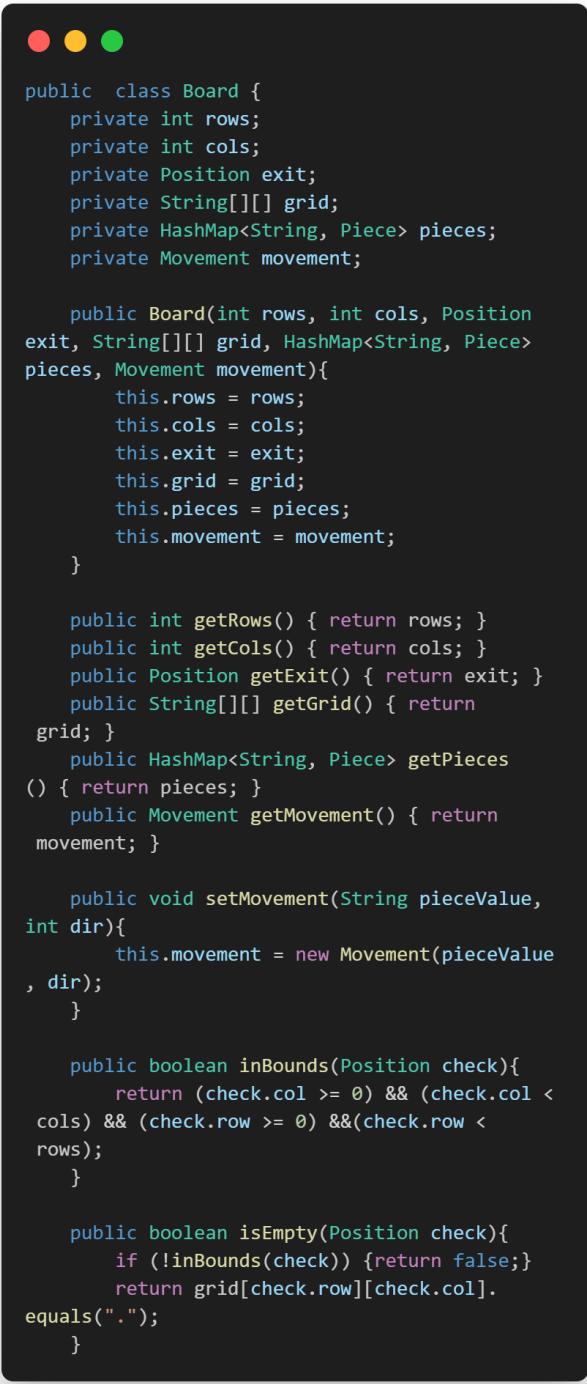
public class Evaluator {
    public static final Comparator<TreeNode> UCS
= Comparator.comparingInt(TreeNode::getCost);

    public static final Comparator<TreeNode> GBFS
= Comparator.comparingInt(n -> n.getHeuristic
());

    public static final Comparator<TreeNode>
AStar = Comparator.comparingInt(n -> n.getCost()
+ n.getHeuristic());
}
```

Kelas comparator yang
menghitung $f(n)$

4.2 Kelas Board

No	Snapshot kode	Penjelasan
1.	 <pre> public class Board { private int rows; private int cols; private Position exit; private String[][] grid; private HashMap<String, Piece> pieces; private Movement movement; public Board(int rows, int cols, Position exit, String[][] grid, HashMap<String, Piece> pieces, Movement movement){ this.rows = rows; this.cols = cols; this.exit = exit; this.grid = grid; this.pieces = pieces; this.movement = movement; } public int getRows() { return rows; } public int getCols() { return cols; } public Position getExit() { return exit; } public String[][] getGrid() { return grid; } public HashMap<String, Piece> getPieces() () { return pieces; } public Movement getMovement() { return movement; } public void setMovement(String pieceValue, int dir){ this.movement = new Movement(pieceValue , dir); } public boolean inBounds(Position check){ return (check.col >= 0) && (check.col < cols) && (check.row >= 0) &&(check.row < rows); } public boolean isEmpty(Position check){ if (!inBounds(check)) {return false;} return grid[check.row][check.col]. equals("."); } } </pre>	<p>Kelas board yang berisi prosedur prosedur untuk menentukan apakah sebuah board merupakan goal (isGoal), menghasilkan penerus/suksesor dengan gerakan-gerakan baru (generateSuccessor), mengaplikasikan perubahan/pergerakan suatu piece (applyMove).</p>

```
● ● ●

public boolean isGoal() {
    Piece primary = pieces.get("P");
    if (primary == null) return false;

    int r = primary.getRowPos();
    int c = primary.getColPos();

    for (int i = 0; i < primary.getSize(); i++)
    {
        if (!primary.getOrientation())
// horizontal
            if (exit.row == r && Math.abs(
exit.col - (c+i)) == 1) return true;
        } else { // vertical
            if (Math.abs(exit.row - (r+i))
== 1 && exit.col == c) return true;
        }
    }
    return false;
}

public List<Board> generateSuccessors() {
    List<Board> successors = new ArrayList
<>();

    for (Piece piece : pieces.values()) {

        for (int dir = 0; dir < 4; dir++) {
            List<Piece> movedVariants =
piece.getAllPossibleMoves(dir, this);
            for (Piece moved :
movedVariants) {
                Board newBoard = this.copy
();
                newBoard.setMovement(moved.
getValue(), dir);
                newBoard.applyMove(piece,
moved);
                successors.add(newBoard);
            }
        }
    }

    return successors;
}

private void applyMove(Piece oldPiece,
Piece newPiece) {
    // delete old
    List<Position> oldPos =
getOccupiedPositions(oldPiece);
    for (Position pos : oldPos) grid[pos.
row][pos.col] = ".";

    // add new
    List<Position> newPos =
getOccupiedPositions(newPiece);
    for (Position pos : newPos) grid[pos.
row][pos.col] = newPiece.getValue();

    // update in map
    pieces.put(String.valueOf(newPiece.
getValue()), newPiece);
}
}
```

```
● ● ●

private List<Position> getOccupiedPositions(
Piece piece) {
    List<Position> positions = new
ArrayList<>();
    int r = piece.getRowPos();
    int c = piece.getColPos();

    for (int i = 0; i < piece.getSize(); i++)
    ) {
        if (piece.getOrientation()) {
            positions.add(new Position(r +
i, c));
        } else {
            positions.add(new Position(r, c
+ i));
        }
    }

    return positions;
}

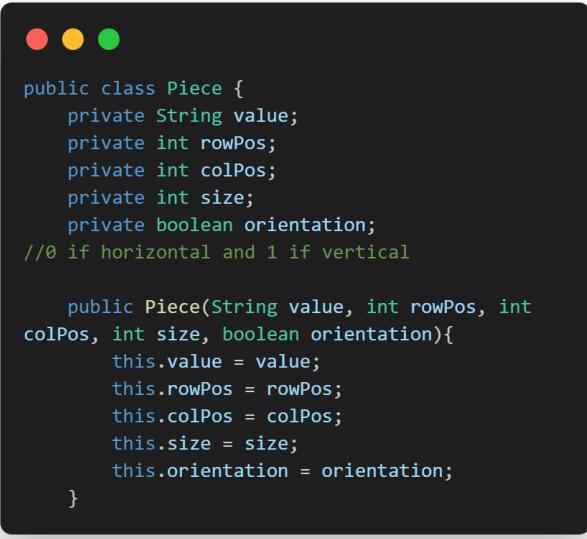
public Board copy() {
    Board b = new Board(this.rows, this.
cols, new Position(this.exit.row, this.exit.col
), new String[rows][cols], new HashMap<>(),
this.movement);

    for (Piece p : this.pieces.values()) {
        Piece clone = new Piece(p.getValue
(), p.getRowPos(), p.getColPos(), p.getSize(),
p.getOrientation());
        b.pieces.put(String.valueOf(p.
getValue()), clone);
    }

    for (int i = 0; i < rows; i++) {
        b.grid[i] = Arrays.copyOf(this.grid
[i], cols);
    }

    return b;
}
```

4.3 Kelas Piece

No	Snapshot kode	Penjelasan
1.	 <pre>public class Piece { private String value; private int rowPos; private int colPos; private int size; private boolean orientation; //0 if horizontal and 1 if vertical public Piece(String value, int rowPos, int colPos, int size, boolean orientation){ this.value = value; this.rowPos = rowPos; this.colPos = colPos; this.size = size; this.orientation = orientation; } }</pre>	Kelas piece yang berisi prosedur-prosedur untuk memeriksa apakah sebuah piece dapat digerakkan, menggerakkan piece, mencari semua gerakan piece yang mungkin.



```
public boolean canMove(int dir, Board board) {
    if (orientation == false) {
        if (dir == 0) { // left
            Position check = new Position
(rowPos, colPos - 1);
            return board.inBounds(check) &&
board.isEmpty(check);
        } else if (dir == 2) { // right
            Position check = new Position
(rowPos, colPos + size);
            return board.inBounds(check) &&
board.isEmpty(check);
        }
    } else { // VERTICAL
        if (dir == 1) { // up
            Position check = new Position
(rowPos - 1, colPos);
            return board.inBounds(check) &&
board.isEmpty(check);
        } else if (dir == 3) { // down
            Position check = new Position
(rowPos + size, colPos);
            return board.inBounds(check) &&
board.isEmpty(check);
        }
    }
    return false;
}

public Piece move(int dir) {
    int newRow = rowPos;
    int newCol = colPos;
    switch (dir) {
        case 0 -> newCol--; // left
        case 2 -> newCol++; // right
        case 1 -> newRow--; // up
        case 3 -> newRow++; // down
    }

    return new Piece(value, newRow, newCol
, size, orientation);
}
```

```
● ● ●

public List<Piece> getAllPossibleMoves(int dir
, Board board) {
    List<Piece> moves = new ArrayList<>();

    int deltaRow = 0;
    int deltaCol = 0;

    switch (dir) {
        case 0 -> deltaCol = -1; // left
        case 2 -> deltaCol = 1; // right
        case 1 -> deltaRow = -1; // up
        case 3 -> deltaRow = 1; // down
        default -> {
            return moves;
        }
        // arah tidak valid
    }

    if ((!orientation && (dir == 1 || dir
== 3)) || // horizontal tapi arah vertical
    (orientation && (dir == 0 || dir ==
2))) { // vertical tapi arah horizontal
        return moves;
    }

    int steps = 1;
    while (true) {
        Position check;
        if (!orientation) { // horizontal
            check = (dir == 0)
                ? new Position
            (rowPos, colPos - steps)
                : new Position
            (rowPos, colPos + size - 1 + steps);
        } else { // vertical
            check = (dir == 1)
                ? new Position(rowPos -
steps, colPos)
                : new Position(rowPos +
size - 1 + steps, colPos);
        }

        if (!board.inBounds(check) || !
board.isEmpty(check)) {
            steps--;
            break;
        }

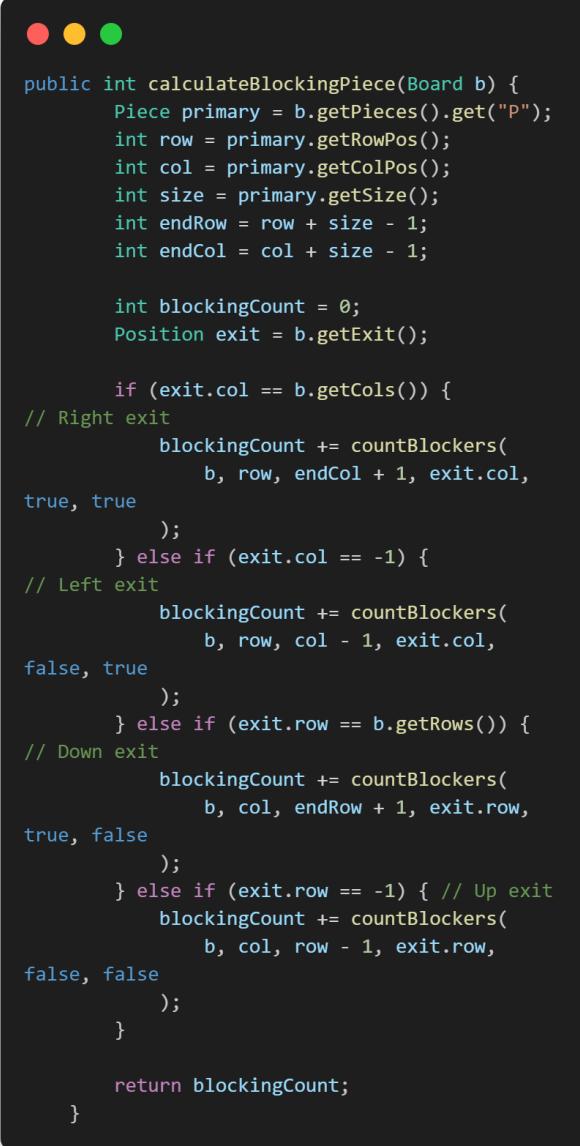
        steps++;
    }

    if (steps > 0) {
        int newRow = rowPos + deltaRow *
steps;
        int newCol = colPos + deltaCol *
steps;

        if (board.inBounds(new Position(
newRow, newCol))) {
            moves.add(new Piece(value,
newRow, newCol, size, orientation));
        }
    }
}

return moves;
}
```

4.4 Heuristik (Blocking Cars)

No	Snapshot kode	Penjelasan
1.	 <pre>public int calculateBlockingPiece(Board b) { Piece primary = b.getPieces().get("P"); int row = primary.getRowPos(); int col = primary.getColPos(); int size = primary.getSize(); int endRow = row + size - 1; int endCol = col + size - 1; int blockingCount = 0; Position exit = b.getExit(); if (exit.col == b.getCols()) { // Right exit blockingCount += countBlockers(b, row, endCol + 1, exit.col, true, true); } else if (exit.col == -1) { // Left exit blockingCount += countBlockers(b, row, col - 1, exit.col, false, true); } else if (exit.row == b.getRows()) { // Down exit blockingCount += countBlockers(b, col, endRow + 1, exit.row, true, false); } else if (exit.row == -1) { // Up exit blockingCount += countBlockers(b, col, row - 1, exit.row, false, false); } return blockingCount; }</pre>	Merupakan prosedur untuk menghitung berapa banyak blok tegak lurus piece utama yang menghalangi jalur exit.

2



```
private int countBlockers(Board b, int fixed,
int start, int end, boolean increasing, boolean
horizontal) {
    int count = 0;
    int step = increasing ? 1 : -1;

    for (int i = start; increasing ? i <
end : i > end; i += step) {
        String cell = horizontal ? b.
getGrid()[fixed][i] : b.getGrid()[i][fixed];
        if (!cell.equals(".")) && !cell.
equals("P")) {
            Piece blocker = b.getPieces().
get(cell);
            if (blocker.getOrientation() ==
horizontal) {
                count++;
            }
        }
    }

    return count;
}
```

Merupakan fungsi helper
untuk menghitung banyak
penghalang.

4.5 Kelas TreeNode

No	Snapshot kode	Penjelasan
----	---------------	------------

1.

```
● ● ●

public class TreeNode {
    private Board board;
    private TreeNode parent;
    private int cost;
    private int heuristic;

    public TreeNode(Board board, TreeNode
parent, int cost, int heuristic){
        this.board = board;
        this.parent = parent;
        this.cost = cost;
        this.heuristic = heuristic;
    }

    public Board getBoard() { return board; }
    public void setBoard(Board board) { this.
board = board; }

    public TreeNode getParent() { return
parent; }
    public void setParent(TreeNode parent) {
this.parent = parent; }

    public int getCost() { return cost; }
    public void setCost(int cost) { this.cost =
cost; }

    public int getHeuristic() { return
heuristic; }
    public void setHeuristic(int heuristic) {
this.heuristic = heuristic; }
}
```

Kelas TreeNode yang berisi state-state penting dari suatu simpul dalam sebuah rute. Terdapat atribut-atribut penting seperti board, parent node, cost hingga simpul tersebut, dan heuristic dari simpul tersebut.

BAB V : Pengujian

Test Case yang digunakan adalah sebagai berikut:

tc1	tc2
6 6 11 AAB..F .BCDF GPPCDFK GH.III GHJ... LLJMM.	6 6 12 .AABEE .DBFG PPDBFGK XYCCCH XYI..H RRI...
tc3	tc4
6 6 12 AABP.. .BPCD LLJJCD .IFFE .MI.GE .MHHGE K	6 6 12 AB.CC. ABD..E KFBDPPE F..HJJ FGGHZZ .II.LL

Berikut hasil pengujinya:

5.1 Pengujian Algoritma UCS

tc1

Rush Hour Puzzle Solver
By Guntara Hambali

Konfigurasi Board

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

Load from File

Algoritma

Algorithm: UCS

Heuristic: Blocking Cars

Solve Puzzle

Board

Statistik Pencarian

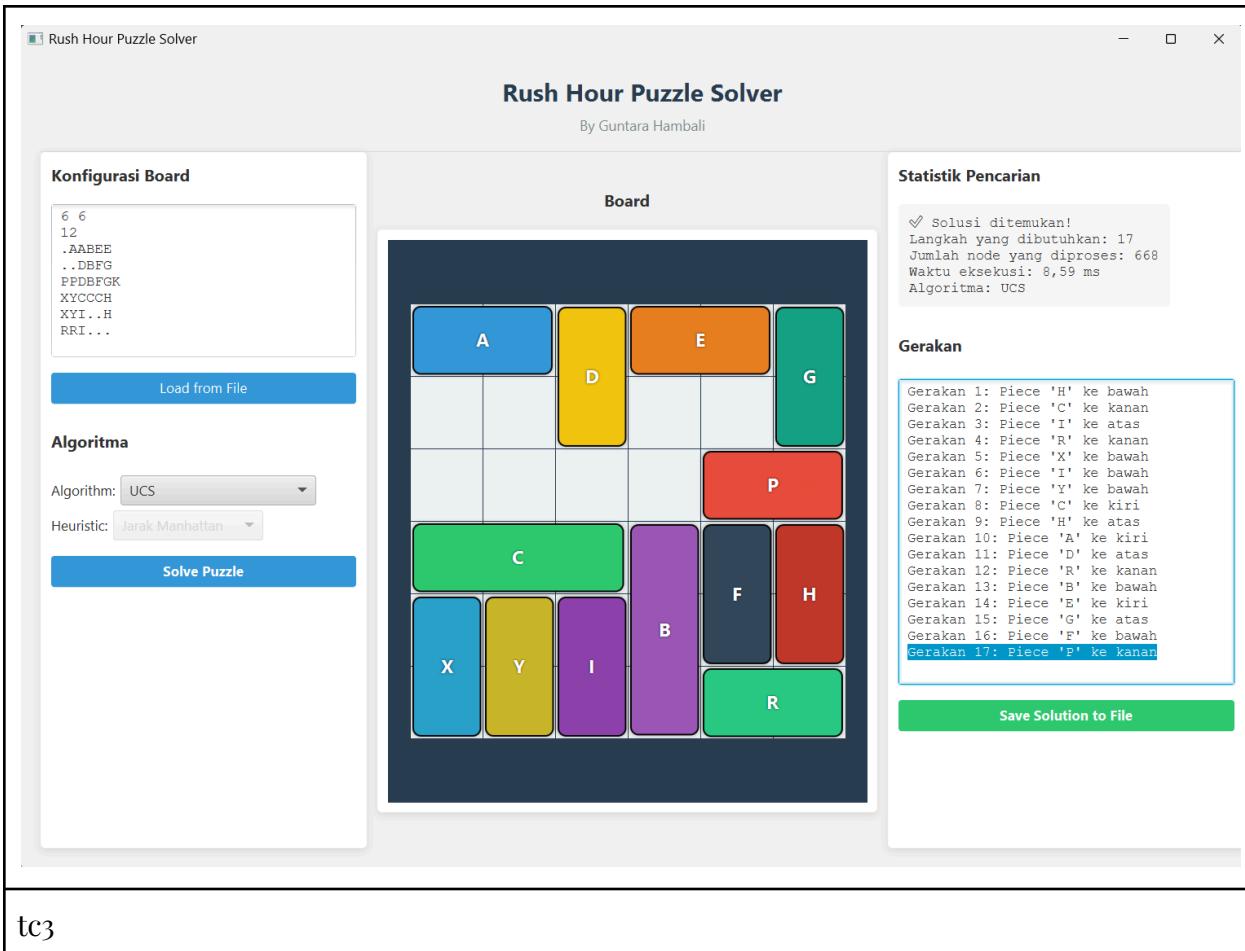
✓ Solusi ditemukan!
Langkah yang dibutuhkan: 5
Jumlah node yang diproses: 195
Waktu eksekusi: 33,52 ms
Algoritma: UCS

Gerakan

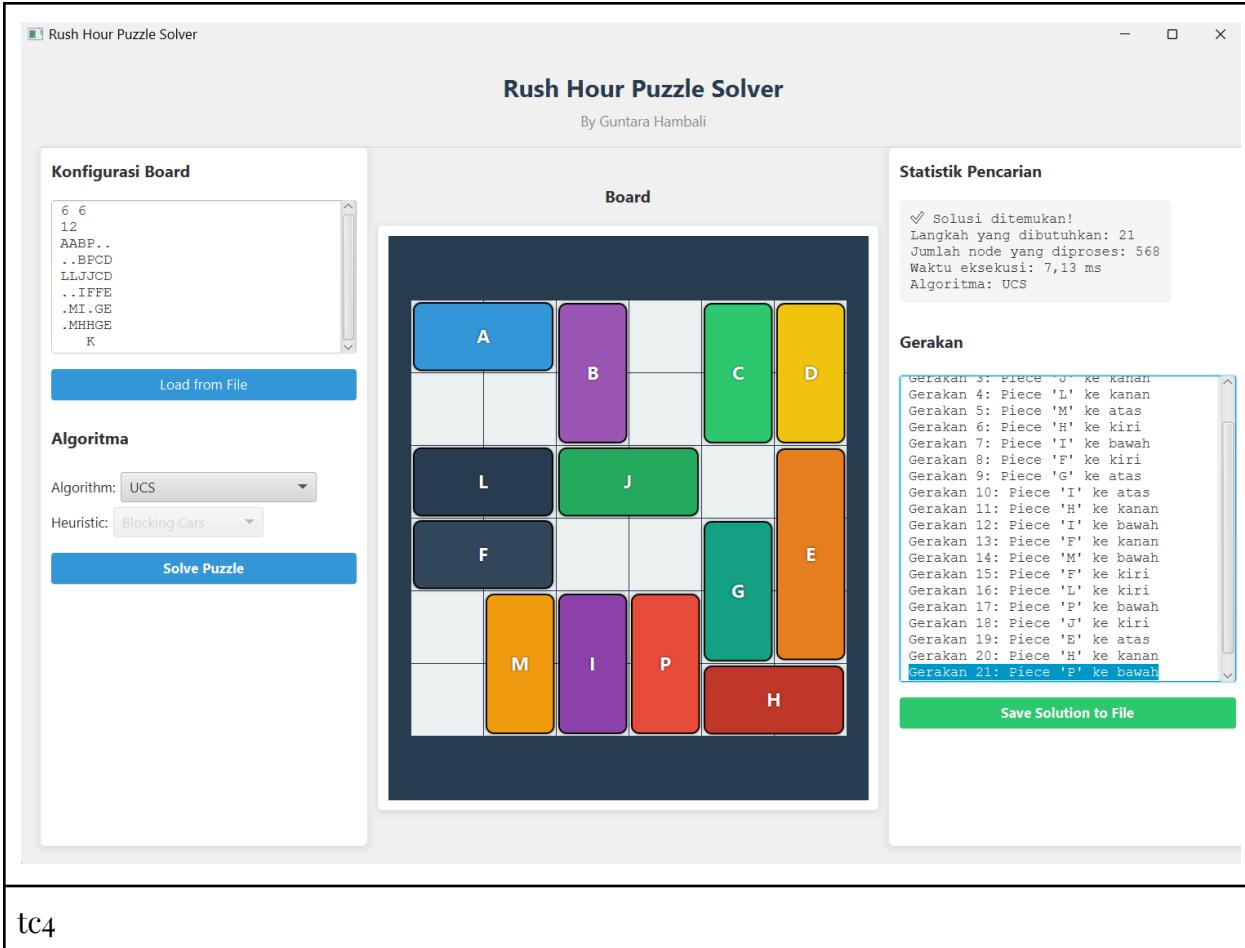
```
Gerakan 1: Piece 'D' ke atas
Gerakan 2: Piece 'I' ke kiri
Gerakan 3: Piece 'C' ke atas
Gerakan 4: Piece 'F' ke bawah
Gerakan 5: Piece 'P' ke kanan.
```

Save Solution to File

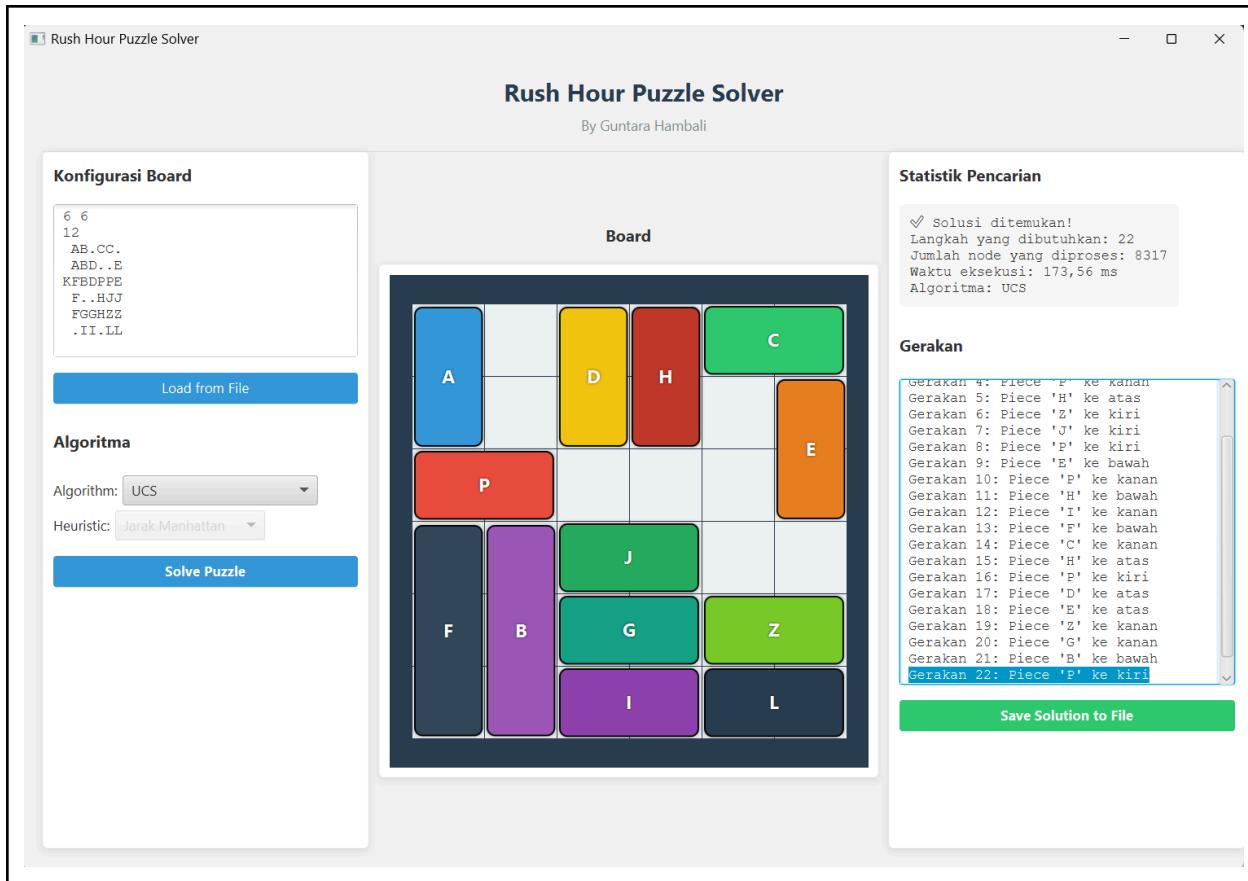
tc2



tc3

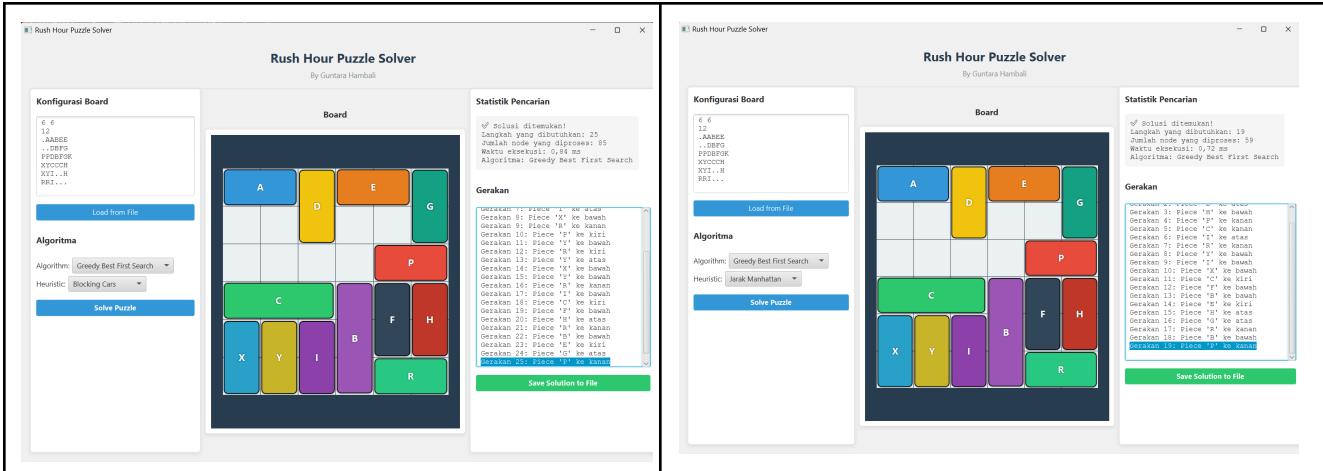


tc4



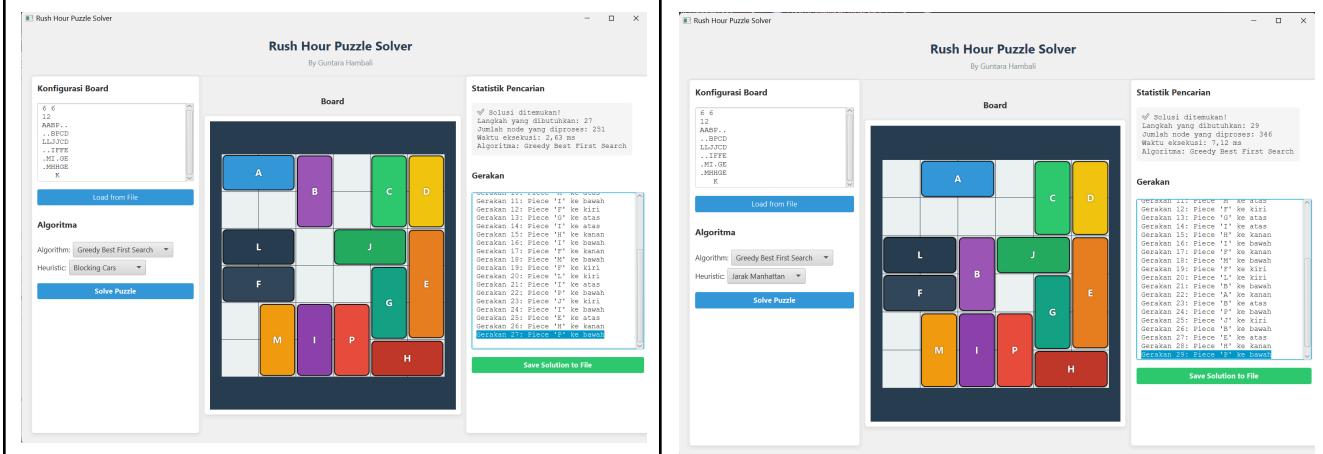
5.2 Pengujian Algoritma GBFS

tc1 (heuristik1)	tc1 (heuristik2)
<p>The configuration board shows the same initial state as the first screenshot. The "Algoritma" dropdown is set to "Greedy Best First Search" and the "Heuristic" dropdown is set to "Blocking Cars". The "Board" section shows the state after 6 moves. The "Statistik Pencarian" section shows:</p> <ul style="list-style-type: none"> Solusi ditemukan! Langkah yang dibutuhkan: 6 Jumlah node yang diproses: 12 Waktu eksekusi: 0,74 ms Algoritma: Greedy Best First Search <p>The "Gerakan" section lists moves 1 through 6. The "Save Solution to File" button is present at the bottom right.</p>	<p>The configuration board shows the same initial state. The "Algoritma" dropdown is set to "Greedy Best First Search" and the "Heuristic" dropdown is set to "Jarak Manhattan". The "Board" section shows the state after 6 moves. The "Statistik Pencarian" section shows:</p> <ul style="list-style-type: none"> Solusi ditemukan! Langkah yang dibutuhkan: 6 Jumlah node yang diproses: 13 Waktu eksekusi: 0,47 ms Algoritma: Greedy Best First Search <p>The "Gerakan" section lists moves 1 through 6. The "Save Solution to File" button is present at the bottom right.</p>
tc2 (heuristik1)	tc2 (heuristik2)



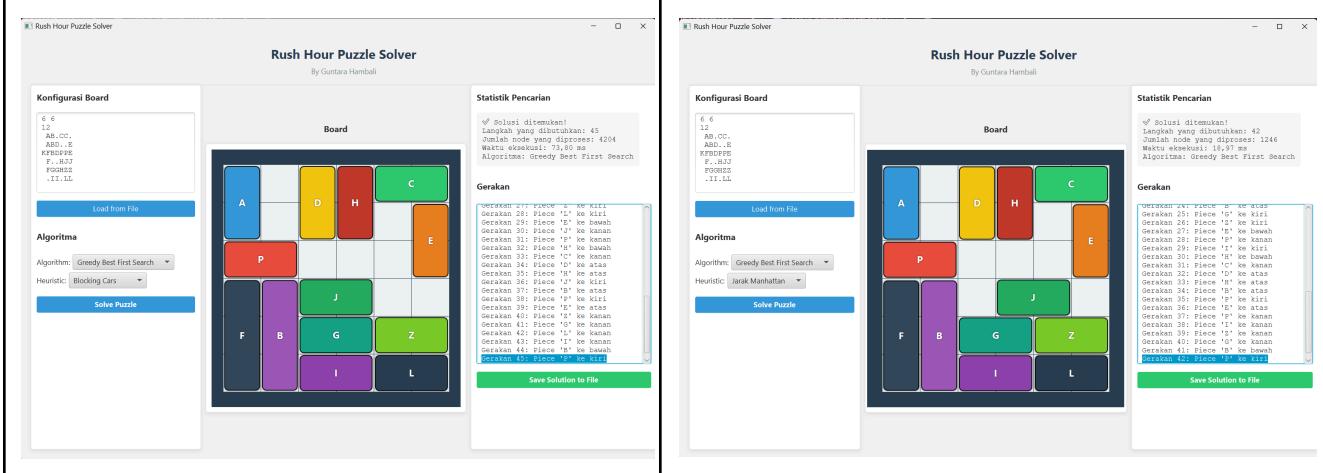
tc3 (heuristik1)

tc3 (heuristik2)

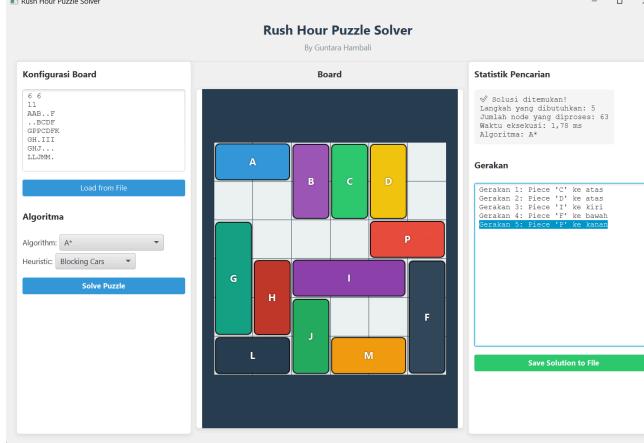
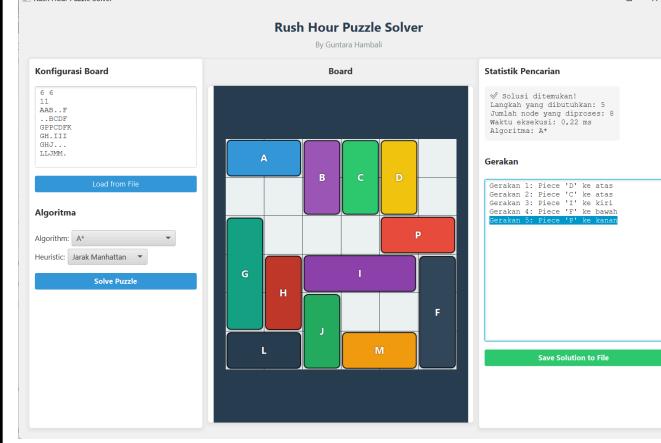
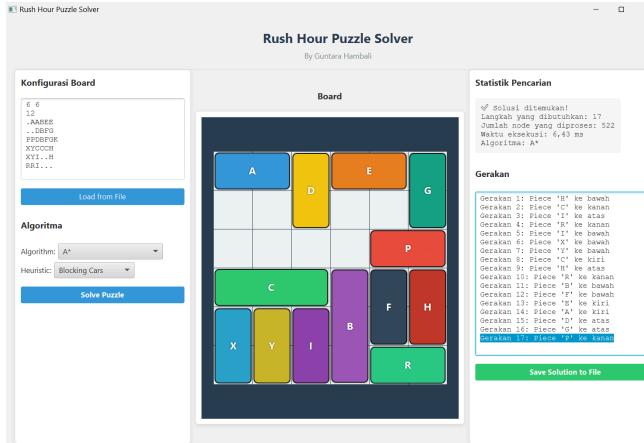
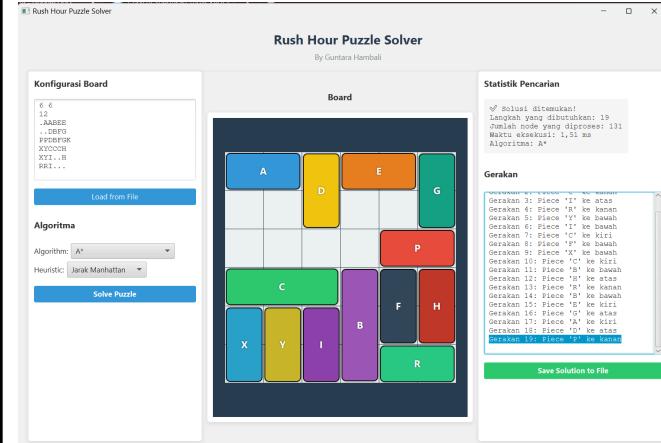


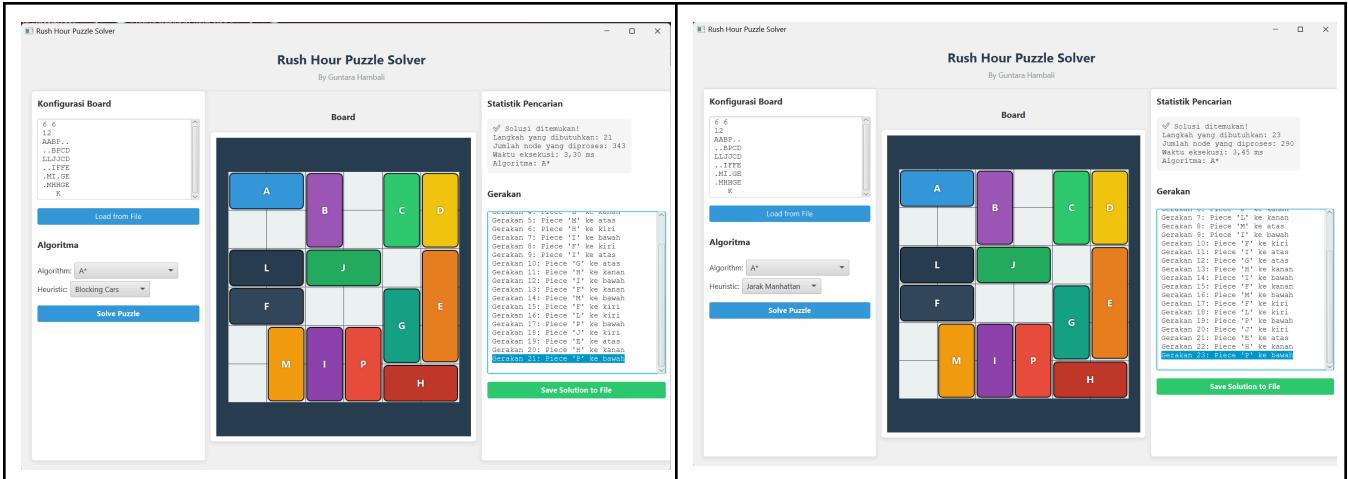
tc4 (heuristik1)

tc4 (heuristik2)



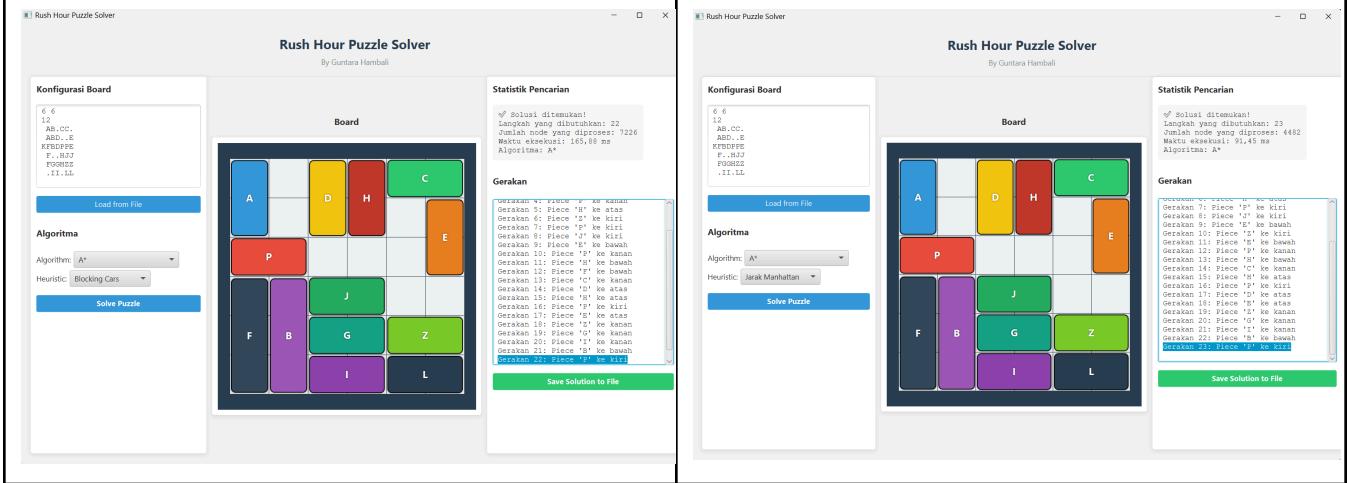
5.3 Pengujian Algoritma A Star

tc1 (heuristik1)	tc1 (heuristik2)
 <p>Rush Hour Puzzle Solver By Gunтарa Hamballi</p> <p>Konfigurasi Board</p> <pre>6 6 11 AAB...F ..BCDF ...CDEP GH...II OH...J LL...N.</pre> <p>Board</p>  <p>Statistik Pencarian</p> <pre># Solusi ditemukan! Langkah yang dibutuhkan: 5 Jumlah node yang diproses: 63 Waktu eksekusi: 1,97 ms Algoritma: A*</pre> <p>Gerakan</p> <pre>Gerakan 1: Piece 'C' ke atas Gerakan 2: Piece 'D' ke atas Gerakan 3: Piece 'I' ke kiri Gerakan 4: Piece 'J' ke bawah Gerakan 5: Piece 'A' ke bawah</pre> <p>Algoritma</p> <p>Algoritma: A* Heuristic: Blocking Cars</p> <p>Solve Puzzle</p> <p>Save Solution to File</p>	 <p>Rush Hour Puzzle Solver By Guntarah Hamballi</p> <p>Konfigurasi Board</p> <pre>6 6 11 AAB...F ..BCDF ...CDEP GH...II OH...J LL...N.</pre> <p>Board</p>  <p>Statistik Pencarian</p> <pre># Solusi ditemukan! Langkah yang dibutuhkan: 8 Jumlah node yang diproses: 8 Waktu eksekusi: 0,22 ms Algoritma: A*</pre> <p>Gerakan</p> <pre>Gerakan 1: Piece 'D' ke atas Gerakan 2: Piece 'C' ke atas Gerakan 3: Piece 'B' ke bawah Gerakan 4: Piece 'F' ke bawah Gerakan 5: Piece 'I' ke bawah</pre> <p>Algoritma</p> <p>Algoritma: A* Heuristic: Jarak Manhattan</p> <p>Solve Puzzle</p> <p>Save Solution to File</p>
tc2 (heuristik1)	tc2 (heuristik2)
 <p>Rush Hour Puzzle Solver By Guntarah Hamballi</p> <p>Konfigurasi Board</p> <pre>12 6 12 AAEE BBBD CCCF DDDE FFBBFGK XYCCDH RR...I</pre> <p>Board</p>  <p>Statistik Pencarian</p> <pre># Solusi ditemukan! Langkah yang dibutuhkan: 17 Jumlah node yang diproses: 522 Waktu eksekusi: 6,43 ms Algoritma: A*</pre> <p>Gerakan</p> <pre>Gerakan 1: Piece 'H' ke bawah Gerakan 2: Piece 'C' ke kanan Gerakan 3: Piece 'D' ke bawah Gerakan 4: Piece 'B' ke kanan Gerakan 5: Piece 'I' ke bawah Gerakan 6: Piece 'E' ke bawah Gerakan 7: Piece 'Y' ke bawah Gerakan 8: Piece 'X' ke kanan Gerakan 9: Piece 'G' ke bawah Gerakan 10: Piece 'R' ke kanan Gerakan 11: Piece 'F' ke bawah Gerakan 12: Piece 'P' ke bawah Gerakan 13: Piece 'B' ke kanan Gerakan 14: Piece 'Y' ke bawah Gerakan 15: Piece 'D' ke atas Gerakan 16: Piece 'F' ke bawah Gerakan 17: Piece 'Y' ke bawah</pre> <p>Algoritma</p> <p>Algoritma: A* Heuristic: Blocking Cars</p> <p>Solve Puzzle</p> <p>Save Solution to File</p>	 <p>Rush Hour Puzzle Solver By Guntarah Hamballi</p> <p>Konfigurasi Board</p> <pre>12 6 12 AAEE BBBD CCCF DDDE FFBBFGK XYCCDH RR...I</pre> <p>Board</p>  <p>Statistik Pencarian</p> <pre># Solusi ditemukan! Langkah yang dibutuhkan: 19 Jumlah node yang diproses: 131 Waktu eksekusi: 1,51 ms Algoritma: A*</pre> <p>Gerakan</p> <pre>Gerakan 1: Piece 'D' ke atas Gerakan 2: Piece 'C' ke atas Gerakan 3: Piece 'B' ke bawah Gerakan 4: Piece 'F' ke bawah Gerakan 5: Piece 'I' ke bawah Gerakan 6: Piece 'E' ke bawah Gerakan 7: Piece 'Y' ke bawah Gerakan 8: Piece 'X' ke kanan Gerakan 9: Piece 'G' ke bawah Gerakan 10: Piece 'R' ke kanan Gerakan 11: Piece 'F' ke bawah Gerakan 12: Piece 'P' ke bawah Gerakan 13: Piece 'B' ke kanan Gerakan 14: Piece 'Y' ke bawah Gerakan 15: Piece 'D' ke atas Gerakan 16: Piece 'F' ke bawah Gerakan 17: Piece 'A' ke kiri Gerakan 18: Piece 'B' ke atas Gerakan 19: Piece 'Y' ke bawah</pre> <p>Algoritma</p> <p>Algoritma: A* Heuristic: Jarak Manhattan</p> <p>Solve Puzzle</p> <p>Save Solution to File</p>



tc4 (heuristik1)

tc4 (heuristik2)



BAB VI : Analisis Hasil Pengujian

6.1 Analisis Algoritma UCS

Dalam penyelesaian Rush Hour, UCS menjelajahi seluruh jalur dari akar hingga goal berdasarkan urutan biaya terendah secara bertahap, yang secara teori membuatnya optimal. Dari hasil pengujian, UCS selalu menghasilkan solusi paling optimal dibanding 2 algoritma lainnya Namun, karena UCS tidak menggunakan arah atau panduan heuristik, simpul yang tidak relevan dengan solusi tetap dievaluasi selama memiliki biaya rendah, sehingga eksplorasi menjadi luas dan bisa sangat lambat. Kompleksitas waktu dan ruang UCS berada pada $O(b^d)$, dengan b sebagai *branching factor* dan d sebagai kedalaman solusi optimal. Meskipun secara teori kompleksitasnya sama dengan A*, UCS cenderung memproses lebih banyak simpul karena tidak memiliki arah pencarian yang spesifik. Akibatnya, pada puzzle seperti Rush Hour yang memiliki ruang keadaan besar dan solusi tersembunyi jauh dalam ruang pencarian, UCS sering kali menjadi tidak efisien kecuali ruang pencarian terbatas.

6.2 Analisis Algoritma GBFS

Dalam kasus Rush Hour, GBFS memilih simpul yang menurut heuristik paling dekat ke solusi dan mengabaikan jalur atau langkah sebelumnya. Karena hanya mengandalkan $h(n)$, GBFS bisa cepat dalam beberapa kasus, terutama jika heuristik sangat akurat. Namun, kompleksitas waktu dalam kasus terburuk berada pada $O(b^d)$, di mana b adalah branching factor dan d adalah kedalaman solusi yang ditemukan. Selain itu, dari hasil pengujian tampak bahwa GBFS hampir selalu tidak dapat menemukan solusi optimum walau sudah cukup mendekati. Ini terjadi karena GBFS dapat terjebak dalam optimum lokal jika heuristik tidak bagus. Kompleksitas ruang pun bisa mendekati $O(b^d)$, karena node yang dievaluasi dan disimpan bisa sangat banyak. Yang membedakan adalah GBFS bisa tampak sangat cepat karena eksplorasinya fokus pada simpul-simpul yang “menjanjikan,” tetapi sayangnya tidak menjamin solusi optimal. Akibatnya, efisiensi GBFS dalam Rush Hour sangat tergantung pada kualitas heuristik. Jika heuristik tidak bagus, GBFS bisa berkinerja lebih buruk daripada UCS.

6.3 Analisis Algoritma A Star

Dalam pencarian rute untuk permainan Rush Hour, A* sangat efektif jika heuristik yang digunakan adalah admissible dan konsisten. Secara kompleksitas waktu, A* memiliki orde $O(b^d)$, di mana b adalah jumlah cabang rata-rata dari suatu state (branching factor), dan d adalah kedalaman solusi optimal. Dari hasil pengujian, algoritma ini selalu mendapatkan hasil yang sama dengan UCS (terutama heuristik) yang merupakan solusi optimum. Akan tetapi, bisa dilihat di statistik hasil bahwa jumlah node yang dikunjungi berkurang drastis

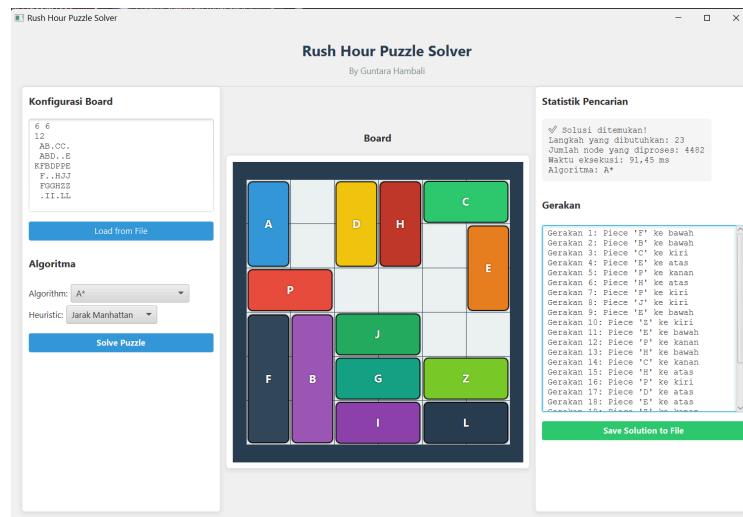
dibandingkan UCS. Ini terjadi karena A* harus menyimpan dan mengevaluasi semua simpul yang nilai $f(n)$ -nya lebih kecil dari solusi optimal untuk menjamin bahwa solusi yang ditemukan memang paling murah (optimal). Dari sisi ruang, A* juga memerlukan $O(b^d)$ memori karena seluruh simpul yang telah dikunjungi dan semua kandidat perlu disimpan selama proses pencarian. Ini membuat A* menuntut alokasi besar dari sisi memori. Dalam praktiknya, dengan heuristik yang baik, A* bisa mengurangi jumlah simpul yang dievaluasi secara drastis dibanding algoritma UCS.

BAB VII : Bonus

7.1 GUI

Saya mengerjakan GUI dengan pustaka JavaFX. Berikut tampilan GUI-nya. Ada beberapa komponen GUI yang penting seperti solve button dan juga input panel. Untuk solve button, ketika di-klik, saya akan memanggil method solve pada kelas GUI tsb yang implementasinya sama seperti method solve di kelas Board. Selain itu, untuk input, saya sesuaikan seperti input file txt. Ada beberapa fitur dari GUI ini, yaitu animasi pergerakan solusi, kemudian track gerakan di sebelah kanan agar pengguna dapat tetap melihat solusinya walau animasinya terus berjalan. Lalu, terdapat fitur statistik pencarian yang bergantung pada algoritma dan heuristic yang digunakan. Terdapat pula fitur untuk memasukkan konfigurasi papan (bisa dari input field, maupun file txt). Terdapat pula fitur untuk save solusi ke dalam file.

Berikut tampilan GUI



7.2 Heuristik Tambahan Jarak Manhattan

Berikut implementasi heuristik jarak Manhattan:

No	Snapshot kode	Penjelasan
----	---------------	------------

1.



```
public int calculateManhattanDistance(Board b) {
    Piece primary = b.getPieces().get("P");
    int row = primary.getRowPos();
    int col = primary.getColPos();
    int size = primary.getSize();
    int endRow = row + size - 1;
    int endCol = col + size - 1;

    Position exit = b.getExit();
    int totalManhattan = 0;

    if (exit.col == b.getCols()) {
        // Right exit
        totalManhattan +=
        sumBlockerDistances(b, row, endCol + 1, exit.
        col, true, true);
    } else if (exit.col == -1) {
        // Left exit
        totalManhattan +=
        sumBlockerDistances(b, row, col - 1, exit.col,
        false, true);
    } else if (exit.row == b.getRows()) {
        // Down exit
        totalManhattan +=
        sumBlockerDistances(b, col, endRow + 1, exit.
        row, true, false);
    } else if (exit.row == -1) { // Up exit
        totalManhattan +=
        sumBlockerDistances(b, col, row - 1, exit.row,
        false, false);
    }

    return totalManhattan;
}
```

Manhattan distance adalah heuristik yang sebenarnya modifikasi dari heuristik blocking cars. Tetapi, dibanding hanya mencari banyak mobil yang menghalangi, manhattan distance juga mencari jarak yang dibutuhkan oleh mobil penghalang untuk berpindah (tidak menghalangi lagi).

Lalu, jarak dari masing-masing block dijumlahkan dan itulah hasil dari jarak Manhattan.



```
private int sumBlockerDistances(Board b, int
fixed, int start, int end, boolean increasing,
boolean horizontal) {
    int sum = 0;
    int step = increasing ? 1 : -1;

    for (int i = start; increasing ? i <
end : i > end; i += step) {
        String cell = horizontal ? b.
getGrid()[fixed][i] : b.getGrid()[i][fixed];
        if (!cell.equals(".")) && !cell.
equals("P")) {
            Piece blocker = b.getPieces().
get(cell);
            int minMove =
calculateMinMoveToClear(b, blocker);
            sum += minMove;
        }
    }

    return sum;
}
```

Merupakan fungsi helper
untuk menjumlahkan jarak
yang dibutuhkan mobil
penghalang untuk berpindah
di sepanjang jalur keluar



```
private int calculateMinMoveToClear(Board b,
Piece piece) {
    int row = piece.getRowPos();
    int col = piece.getColPos();
    int size = piece.getSize();
    boolean isHorizontal = piece.
getOrientation();

    int minDist = Integer.MAX_VALUE;

    // left/up
    for (int offset = 1; offset <= 5;
offset++) {
        int r = isHorizontal ? row : row -
offset;
        int c = isHorizontal ? col - offset
: col;
        if (isValidMove(b, r, c, size,
isHorizontal)) {
            minDist = Math.min(minDist,
offset);
            break;
        }
    }

    // right/down
    for (int offset = 1; offset <= 5;
offset++) {
        int r = isHorizontal ? row : row +
offset;
        int c = isHorizontal ? col + offset
: col;
        if (isValidMove(b, r, c, size,
isHorizontal)) {
            minDist = Math.min(minDist,
offset);
            break;
        }
    }

    return (minDist == Integer.MAX_VALUE) ?
5 : minDist;
}
```

Merupakan fungsi helper yang berupaya menghitung jarak yang diperlukan **satu mobil penghalang** untuk membuka jalan bagi mobil utama

BAB VIII : Lampiran

Link repositori github: https://github.com/guntarahmbl/Tucil3_13523114

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