LAPORAN TUGAS KECIL 3
IF2211 STRATEGI ALGORITMA

Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding

Disusun oleh:

Ferdinand Gabe Tua Sinaga	13523051
Henry Filberto Shenelo	13523108

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

Daftar Isi

BAB I
ALGORITMA PENENTUAN RUTE

A. Penjelasan Dasar

Algoritma penentuan rute merupakan algoritma yang digunakan untuk menemukan lintasan dari suatu titik awal ke titik tujuan dalam suatu ruang pencarian di dalam suatu graf atau peta. Pada permainan puzzle Rush Hour, titik awal adalah kondisi *board* pada awal permainan dan titik tujuan adalah ketik mobil merah berhasil mencapai *exit.* Beberapa algoritma dalam penentuan rute antara lain:

1. Uniform Cost Search (UCS)
2. Best-First Search
3. A*
4. Iterative Deepening Search (IDS)

B. Uniform Cost Search (UCS)

Uniform Cost Search (UCS) adalah algoritma pencarian yang memilih simpul dengan biaya total (cost) terkecil sejauh ini untuk dieksplorasi terlebih dahulu. Fungsi evaluasi f(n) adalah suatu fungsi yang melakukan evaluasi terhadap suatu simpul dan dijadikan sebagai *total cost* simpul tersebut. Dalam permainan Rush Hour, UCS menggunakan fungsi evaluasi f(n) = g(n), dengan g(n) adalah jumlah gerakan (perpindahan mobil/piece) dari posisi awal hingga posisi saat ini.

Berikut adalah penjelasan singkat aplikasi algoritma UCS dalam Rush Hour

1. Keadaan *board* awal dimasukkan ke dalam *priority queue* dengan g(n) = 0.
2. Program mencari semua kemungkinan gerakan mobil dari konfigurasi papan saat ini (posisi mobil pada grid).
3. Setiap konfigurasi baru yang dihasilkan dari satu gerakan mobil akan memiliki g(n) = g(parent) + 1. Hal ini dikarenakan g(n) berdasarkan jumlah gerakan yang telah dilakukan untuk mencapai konfigurasi sekarang.
4. Konfigurasi dengan g(n) paling kecil akan dicek terlebih dahulu.
5. Jika *board* adalah *goal state* (sudah mencapai exit) program berhenti.
6. Jika bukan *goal state*, board akan diekspansi menjadi semua kemungkinan *board* yang terbuat dari suatu gerakan.

UCS menjamin solusi optimal (jumlah gerakan minimum). Pada permainan Rush Hour, UCS dapat dikatakan berlaku seperti BFS. Hal ini dikarenakan pada Rush Hour, semua gerakan bernilai sama (dinilai sebagai satu gerakan), baik jika mobil bergerak satu *tile* atau *multiple tiles.* BFS pada dasarnya adalah UCS pada *weighted graph* dengan nilai *edge*nya sama. BFS adalah UCS yang diterapkan pada graf berbobot sama (uniform edge cost). Keduanya akan membangkitkan node dengan urutan yang sama dan menghasilkan *path* terpendek dalam permainan ini.

C. Greedy Best First Search

Greedy Best First Search (GBFS) menggunakan pendekatan berbasis heuristik murni, yaitu memilih node yang dievaluasi dengan suatu heuristik tertentu paling dekat ke *goal node*. Dalam permainan Rush Hour, fungsi evaluasi GBFS adalah f(n) = h(n), dengan h(n) merupakan estimasi cost dari *board* sekarang hingga mencapai exit.

Pada program yang dibuat, digunakan tiga jenis heuristik, yaitu BlockerOnly dan ManhattanDistance, dan gabungan keduanya.

a. BlockerOnly merupakan heuristik yang menghitung jumlah kendaraan lain yang menghalangi jalur langsung mobil utama (primary piece) menuju pintu keluar. Heuristik ini memperkirakan seberapa sulit jalan keluar dengan menghitung banyaknya mobil/hambatan yang harus dipindahkan.
b. ManhattanDistance adalah heuristik yang menghitung jarak Manhattan (jumlah langkah horizontal dan vertikal) antara posisi terdekat dari mobil utama dan posisi pintu keluar. Heuristik ini memberikan estimasi jarak minimal tanpa mempertimbangkan adanya hambatan di jalur tersebut.
c. Gabungan BlockerOnly dan ManhattanDistance digunakan dengan cara menjumlahkan hasil dari kedua heuristik tersebut, sehingga menghasilkan estimasi yang mempertimbangkan baik jarak ke tujuan maupun hambatan di jalur keluar.

Berikut adalah penjelasan singkat aplikasi algoritma GBFS dalam Rush Hour

1. Dari konfigurasi awal, cari semua kemungkinan *board* baru yang bisa dicapai dengan satu gerakan mobil.
2. Hitung nilai h(n) dari setiap *board* berdasarkan metode heuristik yang digunakan. Semuanya dimasukkan ke priority queue berdasarkan nilai f(n) nya dari yang terkecil.
3. Pilih *board* dengan h(n) terkecil dan ekspansi jika bukan *goal state.*
4. Jika *board* adalah *goal state* (sudah mencapai exit) program berhenti.
5. Lanjutkan terus hingga mencapai *goal state* atau semua node sudah dieksplorasi.

Kelemahan dari GBFS adalah GBFS tidak menjamin solusi optimal pada permainan Rush Hour. GBFS hanya mempertimbangkan estimasi jarak ke goal (h(n)) tanpa memperhatikan total cost hingga saat ini (g(n)). Algoritma ini bisa memilih jalur yang terlihat bagus secara heuristik tetapi sebenarnya memerlukan lebih banyak langkah untuk mencapai solusi, sehingga tidak selalu menghasilkan jumlah gerakan minimum. Heuristik tertentu mungkin akan menghasilkan performa yang lebih baik, tetapi suatu heuristik (kecuali telah dibuktikan secara matematis) tidak menjamin solusi optimal pada permainan ini.

D. Algoritma A*

A-Star (A*) dapat dikatakan menggabungkan UCS dan GBFS dengan mempertimbangkan jumlah langkah yang sudah diambil (g(n)) maupun estimasi cost ke tujuan menggunakan suatu heuristik (h(n)). Fungsi evaluasinya adalah f(n) = g(n) + h(n).

Berikut adalah penjelasan singkat aplikasi algoritma A* pada permainan Rush Hour

1. *Board* awal dimasukkan ke dalam priority queue dengan f(n) = 0 + h(n).
2. Setiap node akan diekspansi menjadi berbagai node yang terbentuk dari satu gerakan mobil. Setiap node yang diekspansi akan dihitung nilai f(n) nya dengan:
    a. g(n) = jumlah langkah dari *board* awal
    b. h(n) = nilai heuristik dari metode yang dipakai (seperti pada GBFS)

    dan dimasukkan ke dalam priority queue
3. Ambil node dengan priority queue dan ekspansi jika bukan *goal state*
4. Lakukan ini hingga mencapai *goal state* atau semua node sudah dieksplorasi.

Suatu fungsi heuristik, h disebut admissible jika estimasi *cost* dari state saat ini ke *goal state* selalu lebih kecil atau sama dengan *cost* asli (h*(n)) (h(n) ≤ h*(n)). Pada program ini, terdapat tiga algoritma:

a. ManhattanDistance merupakan heuristik yang admissible, karena hanya menghitung jarak langsung ke tujuan tanpa mempertimbangkan hambatan, sehingga selalu kurang dari atau sama dengan biaya sebenarnya.
b. BlockerOnly juga admissible karena setiap blocker memerlukan setidaknya satu langkah untuk dipindahkan, sehingga estimasi jumlah langkah yang dihitung oleh heuristik ini tidak akan melebihi biaya sebenarnya untuk mencapai goal state.
c. Gabungan BlockerOnly + ManhattanDistance tidak selalu admissible, karena hasil penjumlahan keduanya bisa melebihi biaya sebenarnya dalam beberapa kasus.

Secara teoritis, A* lebih efisien dibandingkan UCS, selama heuristik yang digunakan memberi informasi dan admissible. UCS harus mengeksplorasi semua kemungkinan berdasarkan jarak langkah (g(n)) tanpa informasi, sedangkan A* menggunakan estimasi (h(n)) agar lebih cepat ke arah goal. Namun, jika heuristik kurang baik, A* mungkin saja tidak lebih efisien, dan jika tidak admissible, A* tidak dijamin memberi solusi yang optimal. Dalam program ini, A* akan lebih efisien dengan penggunaan heuristik-heuristik di atas, walaupun untuk heuristik gabungan, tidak dijamin optimal.

E. Iterative Deepening Search

IDS adalah algoritma yang melakukan pencarian DFS berulang-ulang dengan batas kedalaman (depth limit) yang semakin bertambah, dimulai dari 0 hingga solusi ditemukan. IDS menjamin solusi optimal pada permainan Rush Hour, karena ia mencari semua solusi dengan jumlah langkah minimal terlebih dahulu sebelum menuju *depth* yang lebih tinggi.

Berikut adalah penjelasan singkat aplikasi algoritma A* pada permainan Rush Hour

1. Algoritma dimulai dari root state (konfigurasi awal board) dengan batas kedalaman awal (depth limit) = 0.
2. Lakukan pencarian depth-limited DFS hingga mencapai depth limit tersebut.
3. Jika goal state ditemukan, pencarian dihentikan, tetapi jika belum, akan dilakukan depth-limited DFS kembali dari awal dengan depth limit + 1.
4. Pencarian dihentikan jika semua node dieksplorasi atau solusi ditemukan.

Kelebihan dari IDS adalah IDS menghasilkan solusi yang optimal dan penggunaan memorinya efisien karena hanya menyimpan satu jalur dalam satu waktu. Kelemahannya adalah terdapat duplikasi dalam mengunjungi node dan tidak efisien jika depth tinggi. Dalam kasus permainan Rush Hour, pada konfigurasi papan yang memerlukan banyak langkah, IDS memakan waktu yang sangat lama.

F. Pseudocode Algoritma

1. Solver

```
class Solver:
    visited = Set()          # visited state
    final_path = List()      # Store solution path

    # Reconstruct path from goal to start
    def build_path(goal):
        while goal exists:
            final_path.add_first(goal)
            goal = goal.prev_state

    # Main solving template
    def solve(root, counter, mode)
```

2. UCS

```
function UCS(root):
    priority_queue = PriorityQueue() #prioritize low f(n)
    visited = set()
    root.total_cost = root.steps
    priority_queue.enqueue(root)

    while not priority_queue.is_empty():
        current = priority_queue.dequeue()

        if current.board in visited:
            continue
        visited.add(current.board)

        if current.is_goal():
            return reconstruct_path(current)

        for successor in current.get_successors():
            successor.total_cost = successor.steps  # g(n) only
            priority_queue.enqueue(successor)

    return "No solution"
```

3. Heuristic

```
function ManhattanHeuristic(state):
    primary_pos = state.get_primary_positions()
    exit_pos = state.get_exit_pos()
    return min(|x1 - x2| + |y1 - y2| for all (x1,y1) in
primary_pos)
```

```
function BlockerHeuristic(state):
    primary_pos, exit_pos, grid = state.get_path_data()
    blockers = 0
    for cell in path_to_exit(primary_pos, exit_pos):
        if grid[cell] blocks primary_piece:
            blockers += 1
    return blockers
```

```
function CombinedHeuristic(state):
    return ManhattanHeuristic(state) + BlockerHeuristic(state)
```

4. GBFS

```
function GBFS(root, heuristic_mode):
    heuristic = get_heuristic(heuristic_mode)
    priority_queue = PriorityQueue()  # Prioritizes by h(n) only
    visited = set()
    root.total_cost = heuristic.calculate(root)
    priority_queue.enqueue(root)

    while not priority_queue.is_empty():
        current = priority_queue.dequeue()

        if current.board in visited:
            continue
        visited.add(current.board)

        if current.is_goal():
            return reconstruct_path(current)

        for successor in current.get_successors():
            successor.total_cost = heuristic.calculate(successor)
# h(n) only
            priority_queue.enqueue(successor)

    return "No solution"
```

5. A*

```
function AStar(root, heuristic_mode):
    heuristic = get_heuristic(heuristic_mode)
    priority_queue = PriorityQueue()  # Prioritizes by f(n) = g(n)
+ h(n)
    visited = set()
    root.total_cost = root.steps + heuristic.calculate(root)
    priority_queue.enqueue(root)

    while not priority_queue.is_empty():
        current = priority_queue.dequeue()

        if current.board in visited:
            continue
        visited.add(current.board)

        if current.is_goal():
            return reconstruct_path(current)

        for successor in current.get_successors():
            successor.total_cost = successor.steps +
heuristic.calculate(successor)  # f(n)
            priority_queue.enqueue(successor)

    return "No solution"
```

6. IDS

```
function IDS(root):
    depth_limit = 0
    while True:
        result = DLS(root, depth_limit)
        if result == "FOUND":
            return reconstruct_path(goal_state)
        elif result == "FAILURE":
            return "No solution"
        depth_limit += 1

function DLS(node, limit):
    if node.is_goal():
        return "FOUND"
    if limit == 0:
        return "CUTOFF"

    cutoff_occurred = False
    for successor in node.get_successors():
        successor.prev_state = node
        result = DLS(successor, limit - 1)
        if result == "FOUND":
            return "FOUND"
        if result == "CUTOFF":
            cutoff_occurred = True

    return "CUTOFF" if cutoff_occurred else "FAILURE"
```

BAB II
SOURCE CODE

Tugas kecil ini diimplementasikan dengan menggunakan Bahasa pemrograman Java. Berikut adalah daftar class yang digunakan:

a. Animator.class
b. AStar.class
c. BlockerOnly.class
d. Board.class
e. CombinedHeuristic.class
f. GBFS.class
g. Heuristic.class
h. IDS.class
i. ManhattanDistance.class
j. Piece.class
k. PrioQueue.class
l. Result.class
m. Solver.class
n. State.class
o. UCS.class

Selain itu, daftar functions dan procedures utama yang digunakan adalah sebagai berikut.

1. run()
2. solve()
3. depthLimitedSearch()
4. State() *(constructor)*
5. getSuccessors()
6. isGoalState()
7. readInputFromFileGUI()
8. generateGrid()
9. movePiece()
10. initializeUI()
11. loadPuzzleFile()
12. solvePuzzle()
13. updateBoard()
14. replaySolution()
15. Piece.moveUp()/Down()/Left()/Right()
16. getFinalPath()

Berikut adalah *source code* dari program yang telah dibuat

1. File: Animator.java

```java
import java.awt.*;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.LinkedList;
import javax.swing.*;
import javax.swing.border.Border;

public class Animator extends JFrame {
    private int rows = 6;
    private int cols = 6;
    private Board curBoard;
    private int[] exitPos = null;

    private JSlider speedSlider;
    private JPanel boardPanel;
    private JLabel[][] cellLabels;
    private JButton startButton, replayButton, loadFileButton;
    private JButton saveButton;
    private JComboBox<String> algorithmSelector;
    private JLabel resultLabel;
    private JLabel titleLabel;
    private JPanel headerPanel;
    private JPanel footerPanel;
    private JComboBox<String> heuristicSelector;
    private JLabel nodesLabel;
    private JLabel timeLabel;

    private LinkedList<State> solutionSteps;
    private int currentStep = 0;
    private Timer animationTimer;
    private int countNode = 0;
    private double execTime = 0;

    private static final Color[] VEHICLE_COLORS = {
        new Color(r:0, g:0, b:255),
        new Color(r:0, g:128, b:0),
        new Color(r:255, g:255, b:0),
        new Color(r:0, g:255, b:255),
        new Color(r:128, g:0, b:128),
        new Color(r:255, g:192, b:203),
```

```
        new Color(r:255, g:192, b:203),
        new Color(r:165, g:42, b:42),
        new Color(r:0, g:255, b:0),
        new Color(r:70, g:130, b:180),
        new Color(r:255, g:215, b:0),
        new Color(r:0, g:128, b:128),
        new Color(r:75, g:0, b:130),
        new Color(r:240, g:230, b:140),
        new Color(r:32, g:178, b:170),
        new Color(r:218, g:112, b:214),
        new Color(r:50, g:205, b:50),
        new Color(r:147, g:112, b:219),
        new Color(r:210, g:105, b:30),
        new Color(r:0, g:206, b:209),
        new Color(r:60, g:179, b:113),
        new Color(r:70, g:130, b:180),
        new Color(r:100, g:149, b:237),
        new Color(r:0, g:191, b:255),
        new Color(r:106, g:90, b:205),
        new Color(r:154, g:205, b:50),
        new Color(r:95, g:158, b:160)

    };

    public Animator() {
        initializeUI();
    }
```

```
    private void initializeUI() {
        setTitle(title:"Rush Hour Solver");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new BorderLayout(hgap:10, vgap:10));
        getContentPane().setBackground(new Color(r:240, g:240, b:240));

        createHeaderPanel();
        createBoardPanel(this.rows, this.cols);
        createControlPanel();
        createFooterPanel();

        pack();
        setLocationRelativeTo(c:null);
        setVisible(b:true);
    }

    private void createHeaderPanel() {
        headerPanel = new JPanel();
        headerPanel.setBackground(new Color(r:100, g:149, b:237));
        headerPanel.setBorder(BorderFactory.createEmptyBorder(top:10, left:10, bottom:10, right:10));

        titleLabel = new JLabel(text:"RUSH HOUR SOLVER", SwingConstants.CENTER);
        titleLabel.setFont(new Font(name:"Arial", Font.BOLD, size:24));
        titleLabel.setForeground(Color.WHITE);

        headerPanel.add(titleLabel);
        add(headerPanel, BorderLayout.NORTH);
    }
```

```java
private void createBoardPanel(int rows, int cols) {
    boardPanel = new JPanel(new GridLayout(rows, cols, hgap:2, vgap:2));
    boardPanel.setBorder(BorderFactory.createEmptyBorder(top:10, left:10, bottom:10, right:10));
    boardPanel.setBackground(Color.DARK_GRAY);

    cellLabels = new JLabel[rows][cols];

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cellLabels[i][j] = new JLabel(text:"", SwingConstants.CENTER);
            cellLabels[i][j].setOpaque(isOpaque:true);
            cellLabels[i][j].setBackground(Color.LIGHT_GRAY);
            cellLabels[i][j].setPreferredSize(new Dimension(width:60, height:60));
            cellLabels[i][j].setFont(new Font(name:"Arial", Font.BOLD, size:16));

            Border finalBorder;

            if (exitPos != null && i == exitPos[0] && j == exitPos[1]) {
                String direction = (curBoard != null) ? curBoard.getExitDirection() : "UNKNOWN";
                Border normalBorder = BorderFactory.createLineBorder(Color.BLACK);
                Border exitMarker;

                switch (direction) {
                    case "UP":
                        exitMarker = BorderFactory.createMatteBorder(top:10, left:0, bottom:0, right:0, Color.RED);
                        break;
                    case "DOWN":
                        exitMarker = BorderFactory.createMatteBorder(top:0, left:0, bottom:10, right:0, Color.RED);
                        break;
                    case "LEFT":
                        exitMarker = BorderFactory.createMatteBorder(top:0, left:10, bottom:0, right:0, Color.RED);
                        break;
                    case "RIGHT":
                        exitMarker = BorderFactory.createMatteBorder(top:0, left:0, bottom:0, right:10, Color.RED);
                        break;
                    default:
                        exitMarker = BorderFactory.createEmptyBorder();
                }

                finalBorder = BorderFactory.createCompoundBorder(exitMarker, normalBorder);
            } else {
```

```java
                finalBorder = BorderFactory.createLineBorder(Color.BLACK);
            }

            cellLabels[i][j].setBorder(finalBorder);
            boardPanel.add(cellLabels[i][j]);
        }
    }

    add(boardPanel, BorderLayout.CENTER);
}
```

```
private void createControlPanel() {
    JPanel controlPanel = new JPanel();
    controlPanel.setLayout(new BoxLayout(controlPanel, BoxLayout.Y_AXIS));
    controlPanel.setBorder(BorderFactory.createEmptyBorder(top:10, left:10, bottom:10, right:10));
    controlPanel.setBackground(Color.WHITE);

    loadFileButton = createStyledButton(text:"Load Puzzle", new Color(r:70, g:130, b:180));
    loadFileButton.addActionListener(k -> loadPuzzleFile());

    algorithmSelector = new JComboBox<>(new String[]{"GBFS", "UCS", "A*", "IDS"});
    algorithmSelector.setMaximumSize(new Dimension(width:200, height:30));
    algorithmSelector.setAlignmentX(Component.CENTER_ALIGNMENT);
    algorithmSelector.addActionListener(e ->{
        replayButton.setEnabled(b:false);
        saveButton.setEnabled(b:false);}
    );

    heuristicSelector = new JComboBox<>(new String[]{
        "Heuristic 1 - Blocking Vehicles",
        "Heuristic 2 - Manhattan Distance",
        "Heuristic 3 - Combined"
    });
    heuristicSelector.setMaximumSize(new Dimension(width:200, height:30));
    heuristicSelector.setAlignmentX(Component.CENTER_ALIGNMENT);
    heuristicSelector.addActionListener(e ->{
        replayButton.setEnabled(b:false);
        saveButton.setEnabled(b:false);}
    );

    this.speedSlider = new JSlider(JSlider.HORIZONTAL, min:50, max:1000, value:500); // min=50, max=1000, initial=500
    speedSlider.setMajorTickSpacing(n:250);
    speedSlider.setMinorTickSpacing(n:50);
    speedSlider.setPaintTicks(b:true);
    speedSlider.setPaintLabels(b:true);
    speedSlider.setAlignmentX(Component.CENTER_ALIGNMENT);

    JLabel replaySpeed = new JLabel(text:"Animation Speed");
    replaySpeed.setAlignmentX(Component.CENTER_ALIGNMENT);

    startButton = createStyledButton(text:"Solve", new Color(r:34, g:139, b:34));
    startButton.setEnabled(b:false);
    startButton.addActionListener(k -> solvePuzzle());
```

```
    saveButton = createStyledButton(text:"Save Solution", new Color(r:255, g:140, b:0));
    saveButton.setEnabled(b:false);
    saveButton.addActionListener(e -> saveSolutionToFile());

    replayButton = createStyledButton(text:"Replay Solution", new Color(r:138, g:43, b:226));
    replayButton.setEnabled(b:false);
    replayButton.addActionListener(k -> replaySolution());

    controlPanel.add(loadFileButton);
    controlPanel.add(Box.createRigidArea(new Dimension(width:0, height:40)));
    controlPanel.add(algorithmSelector);
    controlPanel.add(Box.createRigidArea(new Dimension(width:0, height:10)));
    controlPanel.add(heuristicSelector);
    controlPanel.add(Box.createRigidArea(new Dimension(width:0, height:10)));
    controlPanel.add(startButton);
    controlPanel.add(Box.createRigidArea(new Dimension(width:0, height:10)));
    controlPanel.add(replayButton);
    controlPanel.add(Box.createRigidArea(new Dimension(width:0, height:10)));
    controlPanel.add(replaySpeed);
    controlPanel.add(Box.createRigidArea(new Dimension(width:0, height:5)));
    controlPanel.add(speedSlider);
    controlPanel.add(Box.createRigidArea(new Dimension(width:0, height:10)));
    controlPanel.add(saveButton);

    add(controlPanel, BorderLayout.EAST);
}

private void createFooterPanel() {
    footerPanel = new JPanel(new FlowLayout(FlowLayout.CENTER, hgap:20, vgap:10));
    footerPanel.setBackground(new Color(r:220, g:220, b:220));

    resultLabel = new JLabel(text:"Ready to solve!");
    resultLabel.setFont(new Font(name:"Arial", Font.BOLD, size:14));

    nodesLabel = new JLabel(text:"Nodes explored: 0");
    nodesLabel.setFont(new Font(name:"Arial", Font.PLAIN, size:12));

    timeLabel = new JLabel(text:"Time: 0 ms");
    timeLabel.setFont(new Font(name:"Arial", Font.PLAIN, size:12));

    footerPanel.add(resultLabel);
```

```
        footerPanel.add(nodesLabel);
        footerPanel.add(timeLabel);

        add(footerPanel, BorderLayout.SOUTH);
    }

    private JButton createStyledButton(String text, Color bgColor) {
        JButton button = new JButton(text);
        button.setAlignmentX(Component.CENTER_ALIGNMENT);
        button.setMaximumSize(new Dimension(width:200, height:35));
        button.setFont(new Font(name:"Arial", Font.BOLD, size:14));
        button.setBackground(bgColor);
        button.setForeground(Color.WHITE);
        button.setFocusPainted(b:false);
        return button;
    }


    private void loadPuzzleFile() {
        JFileChooser fileChooser = new JFileChooser();
        if (fileChooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
            File selectedFile = fileChooser.getSelectedFile();
            String filePath = selectedFile.getAbsolutePath();
            startButton.setEnabled(b:false);
            replayButton.setEnabled(b:false);
            try {
                this.curBoard = new Board();
                this.curBoard.readInputFromFileGUI(filePath);

                this.rows = curBoard.getRows();
                this.cols = curBoard.getColumns();
                this.exitPos = curBoard.getExitPos();

                resetBoard(this.rows, this.cols);
                updateBoard(this.curBoard);
                System.out.println("File path: " + filePath);
                resultLabel.setText(text:"Puzzle loaded!");
                replayButton.setEnabled(b:false);
                startButton.setEnabled(b:true);
                saveButton.setEnabled(b:false);
            } catch (FileNotFoundException e) {
                JOptionPane.showMessageDialog(this,
```

```
                    "File tidak ditemukan:\n" + e.getMessage(),
                    title:"File Error",
                    JOptionPane.ERROR_MESSAGE);
            saveButton.setEnabled(b:false);
            this.exitPos = null;
            remove(this.boardPanel);
            createBoardPanel(rows:6, cols:6);
            revalidate();
            repaint();
        } catch (IllegalArgumentException e) {
            JOptionPane.showMessageDialog(this,
                    "Gagal memuat file puzzle:\n" + e.getMessage(),
                    title:"Format Error",
                    JOptionPane.ERROR_MESSAGE);
            saveButton.setEnabled(b:false);
            this.exitPos = null;
            remove(this.boardPanel);
            createBoardPanel(rows:6, cols:6);
            revalidate();
            repaint();
        }
    }
}

private void solvePuzzle() {
    String algorithm = (String) algorithmSelector.getSelectedItem();
    resultLabel.setText("Solving using " + algorithm + "...");

    String heuristic = (String) heuristicSelector.getSelectedItem();
    int mode = 1;
    switch (heuristic) {
        case "Heuristic 1 - Blocking Vehicles" -> mode = 1;
        case "Heuristic 2 - Manhattan Distance" -> mode = 2;
        case "Heuristic 3 - Combined" -> mode = 3;
        default -> {
        }
    }

    switch (algorithm) {
        case "GBFS":
            GBFS g = new GBFS();
            Result res = g.run(this.curBoard, mode);
```

```
            this.solutionSteps = res.solutionStep;
            this.countNode = res.nodes;
            this.execTime = res.time;
            // this.curBoard = solutionSteps.getLast();
            replaySolution();
            break;
        case "A*":
            {
                AStar aStarAlgo = new AStar();
                Result result = aStarAlgo.run(this.curBoard, mode);
                this.solutionSteps = result.solutionStep;
                this.countNode = result.nodes;
                this.execTime = result.time;
                replaySolution();
                break;
            }
        case "UCS":
            {
                UCS ucsAlgo = new UCS();
                Result result = ucsAlgo.run(this.curBoard, -1);
                this.solutionSteps = result.solutionStep;
                this.countNode = result.nodes;
                this.execTime = result.time;
                replaySolution();
                break;
            }
        case "IDS":
            {
                IDS idsAlgo = new IDS();
                Result result = idsAlgo.run(this.curBoard);
                this.solutionSteps = result.solutionStep;
                this.countNode = result.nodes;
                this.execTime = result.time;
                replaySolution();
                break;
            }
        default:
            break;
    }


    SwingUtilities.invokeLater(() -> {
```

```
            int numberofSteps = solutionSteps.size();
            if (numberofSteps > 1) {
                numberofSteps -= 2; // remove initial and final state
            }
            if(this.solutionSteps.isEmpty()){
                resultLabel.setText(text:"No Solution found!");
            }
            else{
                resultLabel.setText("Solution found in "+numberofSteps+" steps!");
            }
            nodesLabel.setText("Nodes explored: " + this.countNode);
            timeLabel.setText("Time: "+this.execTime+" ms");
            replayButton.setEnabled(b:true);
            saveButton.setEnabled(b:true);

        });

    }

    private void replaySolution() {
        if (solutionSteps == null || solutionSteps.isEmpty()) {
            JOptionPane.showMessageDialog(this, message:"No solution to replay", title:"Error", JOptionPane.ERROR_MESSAGE);
            return;
        }
        currentStep = 0;
        int delay = this.speedSlider.getValue();
        //Ubah delay kalo kelambatan
        animationTimer = new Timer(delay, k -> {
            if (currentStep < solutionSteps.size()) {
                updateBoard(solutionSteps.get(currentStep).getCurrBoard());
                currentStep++;
            } else {
                animationTimer.stop();
            }
        });
        animationTimer.start();
    }


    private void updateBoard(Board boardInState) {
        char[][] grid = boardInState.generateGrid();
        int[] exitPos = boardInState.getExitPos();
```

```
        for (int i = 0; i < this.rows; i++) {
            for (int j = 0; j < this.cols; j++) {
                char cellValue = grid[i][j];
                if (cellValue != '.') {
                    if (cellValue == 'P') {
                        cellLabels[i][j].setBackground(Color.RED);
                    } else {
                        int vehicleId = cellValue - 'A';
                        cellLabels[i][j].setBackground(VEHICLE_COLORS[vehicleId % VEHICLE_COLORS.length]);
                    }
                    cellLabels[i][j].setText(String.valueOf(cellValue));
                } else {
                    cellLabels[i][j].setBackground(Color.LIGHT_GRAY);
                    if (i == exitPos[0] && j == exitPos[1]) {
                        cellLabels[i][j].setText(text:"EXIT");
                    }
                    else {
                        cellLabels[i][j].setText(text:"");
                    }
                }
            }
        }

    }

    private void resetBoard(int newRows, int newCols) {
        if (boardPanel != null) {
            remove(boardPanel);
        }
        createBoardPanel(newRows, newCols);

        revalidate();
        repaint();
    }

    private String getSolutionInString(){
        String res = "";

        for(int k = 0; k < this.solutionSteps.size(); k++){
            State currState = this.solutionSteps.get(k);
            Board b = currState.getCurrBoard();
            if(k==0){
```

```
                res+= "Kondisi awal papan \n";
            }
            else{
                res= res + "Gerakan " + k + ": "+ currState.getMovedPiece()
                + "-" + currState.getMoveDirection() + "\n";
            }
            char[][] grid = b.generateGrid();
            for (int i = 0; i < this.rows; i++) {
                for (int j = 0; j < this.cols; j++) {
                    res+=grid[i][j];
                }
                res+="\n";
            }

            res+="\n";
        }

        return res;
    }

    private void saveSolutionToFile() {
        if (solutionSteps.isEmpty()) {
            JOptionPane.showMessageDialog(this, message:"No solution available to save.", title:"Error", JOptionPane.ERROR_MESSAGE);
            return;
        }

        String fileName = JOptionPane.showInputDialog(this, message:"Enter the filename (without extension):", title:"Save Solution",
        if (fileName == null || fileName.trim().isEmpty()) {
            JOptionPane.showMessageDialog(this, message:"Filename cannot be empty.", title:"Error", JOptionPane.ERROR_MESSAGE);
            return;
        }

        if (!fileName.toLowerCase().endsWith(suffix:".txt")) {
            fileName += ".txt";
        }

        JFileChooser folderChooser = new JFileChooser();
        folderChooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
        int result = folderChooser.showSaveDialog(this);

        if (result == JFileChooser.APPROVE_OPTION) {
            File selectedFolder = folderChooser.getSelectedFile();
```

```
            File outputFile = new File(selectedFolder, fileName);

            try (PrintWriter writer = new PrintWriter(outputFile)) {
                writer.print(getSolutionInString());
                JOptionPane.showMessageDialog(this, "Solution saved to " + outputFile.getAbsolutePath(), title:"Success", JOptionPane.INFORMATION_MESSAGE);
            } catch (IOException ex) {
                JOptionPane.showMessageDialog(this, "Failed to save solution: " + ex.getMessage(), title:"Error", JOptionPane.ERROR_MESSAGE);
            }
        }
    }



Run | Debug
public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
        try {
            new Animator();
        } catch (Exception e) {
            e.printStackTrace();
        }
    });
}
}
```

2. File: Board.java

```java
public class Board {
    private int rows;
    private int columns;
    private int numPieces; // Number of pieces specified in the input file
    private final Map<Character, Piece> pieces = new HashMap<>();

    public static final char PRIMARY_PIECE = 'P';
    public static final char EXIT = 'K';
    private int[] exitPos = null;

    public Board() {
        // Empty constructor
    }

    public boolean readInputFromFile() {
        try (Scanner inputScanner = new Scanner(System.in)) {
            System.out.print(s:"Masukkan nama file: ");
            String filePath = "../test/" + inputScanner.nextLine().trim();

            try {
                try (Scanner fileScanner = new Scanner(new File(filePath))) {
                    if (!fileScanner.hasNextLine()) {
                        System.out.println(x:"Error: File kosong atau tidak valid.");
                        return false;
                    }

                    String[] dimensions = fileScanner.nextLine().split(regex:" ");
                    if (dimensions.length != 2) {
                        System.out.println(x:"Error: Baris pertama harus terdiri dari dimensi papan dalam bilangan bulat (A B).");
                        return false;
                    }

                    rows = Integer.parseInt(dimensions[0]);
                    columns = Integer.parseInt(dimensions[1]);

                    if (!fileScanner.hasNextLine()) {
                        System.out.println(x:"Error: Baris kedua harus ada untuk menyatakan jumlah kendaraan.");
                        return false;
                    }

                    numPieces = Integer.parseInt(fileScanner.nextLine().trim());

                    Map<Character, List<int[]>> pieceCoordinates = new HashMap<>();
```

```java
                    int currentRow = 0;
                    boolean validLine = true;
                    while (fileScanner.hasNextLine() && validLine) {
                        String line = fileScanner.nextLine();
                        validLine = processLine(line, currentRow, pieceCoordinates);
                        currentRow++;
                    }

                    if (!verifyBoardConstraints(pieceCoordinates)) {
                        System.out.println(x:"Papan tidak sesuai. Periksa file input");
                        return false;
                    }

                    normalizeCoordinates(pieceCoordinates);

                    if (!validateNormalizedCoordinates(pieceCoordinates)) {
                        System.out.println(x:"Error: Papan melebihi dimensi yang ditulis.");
                        return false;
                    }

                    createPieces(pieceCoordinates);
                }

                if ((pieces.size() - 1) != numPieces) {
                    System.out.println("Warning: Jumlah kendaraan dalam file (" +
                        (pieces.size() - 1) +
                        ") tidak sesuai dengan yang dinyatakan (" + numPieces + ")");
                }

                System.out.println(x:"Input berhasil diproses.");
                if (exitPos != null)
                    System.out.println("Posisi pintu keluar 'K': [" + exitPos[0] + "," + exitPos[1] + "]");
                else
                    System.out.println(x:"Pintu keluar 'K' tidak ditemukan.");

            } catch (FileNotFoundException e) {
                System.out.println(x:"File tidak ditemukan.");
            } catch (NumberFormatException e) {
                System.out.println(x:"Error parsing angka. Pastikan format file benar.");
```

```
            } catch (Exception e) {
                // Catch all other unexpected exceptions
                System.out.println("Terjadi kesalahan tak terduga: " + e.getMessage());
            }
        } catch (Exception e) {
            System.out.println("Terjadi kesalahan input dari pengguna: " + e.getMessage());
        }
        return true;
}


    private boolean processLine(String line, int row, Map<Character, List<int[]>> pieceCoordinates) {
        // if (row > rows) {
        //     System.out.println("Error: Jumlah baris dalam file melebihi batas (" + (rows + 1) + ").");
        //     return false;
        // }

        // if (line.length() > columns + 1) {
        //     System.out.println("Error: Panjang baris ke-" + row + " melebihi batas (" + (columns + 1) + ").");
        //     return false;
        // }

        for (int col = 0; col < line.length(); col++) {
            char ch = line.charAt(col);

            if (ch != ' ' && ch != '.') {
                pieceCoordinates.computeIfAbsent(ch, k -> new ArrayList<>()).add(new int[]{row, col});
                if (ch == EXIT) {
                    exitPos = new int[]{row, col};
                }
            }
        }
        return true;
    }

    private boolean verifyBoardConstraints(Map<Character, List<int[]>> pieceCoordinates) {
        // Verify there is exactly one exit 'K'
        List<int[]> exitCoords = pieceCoordinates.getOrDefault(EXIT, Collections.emptyList());
        if (exitCoords.size() != 1) {
            System.out.println("Error: Harus ada tepat satu 'K' pada papan, ditemukan: " + exitCoords.size());
            return false;
        }
```

```
        int[] exit = exitCoords.get(index:0);
        int kRow = exit[0];
        int kCol = exit[1];

        // Verify exit is on the border
        if (!(kRow == 0 || kRow == rows || kCol == 0 || kCol == columns)) {
            System.out.println("Error: 'K' harus berada di baris 0 atau " + rows + ", atau kolom 0 atau " + columns);
            return false;
        }

        // Verify there is at least one primary piece 'P'
        if (!pieceCoordinates.containsKey(PRIMARY_PIECE)) {
            System.out.println(x:"Error: Papan harus mengandung setidaknya satu 'P'");
            return false;
        }

        return true;
    }

    private boolean hasPieceAt(Map<Character, List<int[]>> pieceCoordinates, int targetRow, int targetCol) {
        for (Map.Entry<Character, List<int[]>> entry : pieceCoordinates.entrySet()) {
            if (entry.getKey() == 'K') continue;
            for (int[] coord : entry.getValue()) {
                if (coord[0] == targetRow && coord[1] == targetCol) {
                    return true;
                }
            }
        }
        return false;
    }

    public String getExitDirection() {
        int[] epos = exitPos;
        int[] ppos = pieces.get(key:'P').getPivot();
        String direction = pieces.get(key:'P').getDirection();

        if (direction.equals(anObject:"horizontal")) {
            if (epos[0] == ppos[0]) {
                if (epos[1] < ppos[1]) {
                    return "LEFT";
                } else if (epos[1] > ppos[1]) {
```

```
                return "RIGHT";
            }
        }
    } else if (direction.equals(anObject:"vertical")) {
        if (epos[1] == ppos[1]) {
            if (epos[0] < ppos[0]) {
                return "UP";
            } else if (epos[0] > ppos[0]) {
                return "DOWN";
            }
        }
    }

    return "UNKNOWN";
}


public String getExitDirection(Map<Character, List<int[]>> pieceCoordinates) {
    List<int[]> kCoords = pieceCoordinates.get(key:'K');
    if (kCoords == null || kCoords.isEmpty()) return "K not found";

    for (int[] coord : kCoords) {
        int r = coord[0];
        int c = coord[1];

        // (0,0)
        if (r == 0 && c == 0) {
            if (hasPieceAt(pieceCoordinates, targetRow:0, targetCol:1)) return "left";
            else return "up";
        }
        // (0, 1 up to columns-1)
        if (r == 0 && c > 0 && c < columns) {
            return "up";
        }

        // (0, columns)
        if (r == 0 && c == columns) {
            return "right";

        }
```

```
        if (c == 0 && r > 0 && r < rows) {
            return "left";
        }

        // (rows, 0)
        if (c == 0 && r == rows) {
            return "bottom";
        }

        // (rows, 1 up to columns-1)
        if (c > 0 && c < columns && r == rows) {
            return "bottom";
        }

        // (rows-1, columns)
        if (c == columns && r == rows - 1) {
            return "right";
        }
    }

    return "right";
}


private void normalizeCoordinates(Map<Character, List<int[]>> pieceCoordinates) {
    if (exitPos == null) {
        System.out.println(x:"Pintu keluar tidak ditemukan. Tidak bisa menormalisasi.");
        return;
    }

    // int exitRow = exitPos[0];
    // int exitCol = exitPos[1];

    String exitDirection = getExitDirection(pieceCoordinates);

    // Adjust all piece coordinates
    for (Map.Entry<Character, List<int[]>> entry : pieceCoordinates.entrySet()) {
        char piece = entry.getKey();
        if (piece == EXIT) continue;

        //coord[0] = row
```

```
        List<int[]> coords = entry.getValue();

        if (exitDirection.equals(anObject:"left")) { //left
            for (int[] coord : coords) {
                coord[1] -= 1;
            }
        } else if (exitDirection.equals(anObject:"up")) { //up
            for (int[] coord : coords) {
                coord[0] -= 1;
            }
        }
    }

    // Adjust exitPos once
    if (exitDirection.equals(anObject:"bottom")) { // bottom
        exitPos[0] -= 1;
    } else if (exitDirection.equals(anObject:"right")) { // right
        exitPos[1] -= 1;
    }
}

private boolean validateNormalizedCoordinates(Map<Character, List<int[]>> pieceCoordinates) {
    for (Map.Entry<Character, List<int[]>> entry : pieceCoordinates.entrySet()) {
        char piece = entry.getKey();
        if (piece == EXIT) continue;

        List<int[]> coords = entry.getValue();
        for (int[] coord : coords) {
            int r = coord[0];
            int c = coord[1];
            if (r < 0 || r >= rows || c < 0 || c >= columns) {
                return false;
            }
        }
    }
    return true;
}

private void createPieces(Map<Character, List<int[]>> pieceCoordinates) {
    for (Map.Entry<Character, List<int[]>> entry : pieceCoordinates.entrySet()) {
        char symbol = entry.getKey();
```

```
        if (symbol == EXIT) continue;

        List<int[]> positions = entry.getValue();
        String direction = determineDirection(positions);

        // Find the length and pivot (smallest coordinate)
        int length = positions.size();
        int[] pivot = findPivot(positions, direction);

        Piece piece = new Piece(symbol, direction, pivot, length);
        pieces.put(symbol, piece);
    }
}

private int[] findPivot(List<int[]> positions, String direction) {
    if (positions.isEmpty()) {
        return new int[]{0, 0};
    }

    int[] pivot = positions.get(index:0).clone();

    for (int[] pos : positions) {
        if (direction.equals(anObject:"horizontal")) {
            if (pos[1] < pivot[1]) {
                pivot = pos.clone();
            }
        } else { // vertical
            if (pos[0] < pivot[0]) {
                pivot = pos.clone();
            }
        }
    }

    return pivot;
}

public String determineDirection(List<int[]> coords) {
    if (coords.size() == 1) {
        return "horizontal"; // Default for single pieces
    }
```

```
    boolean allSameRow = true;
    int firstRow = coords.get(index:0)[0];
    for (int[] pos : coords) {
        if (pos[0] != firstRow) {
            allSameRow = false;
            break;
        }
    }

    if (allSameRow) {
        return "horizontal";
    }

    boolean allSameCol = true;
    int firstCol = coords.get(index:0)[1];
    for (int[] pos : coords) {
        if (pos[1] != firstCol) {
            allSameCol = false;
            break;
        }
    }

    if (allSameCol) {
        return "vertical";
    } else {
        System.out.println(x:"Warning: Piece has non-linear shape.");
        return "unknown";
    }
}

// Generate grid only when needed
public char[][] generateGrid() {
    char[][] grid = new char[rows][columns];

    // Initialize grid with empty spaces
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            grid[i][j] = '.';
        }
    }
```

```
    // Place pieces on the grid
    for (Piece piece : pieces.values()) {
        List<int[]> positions = piece.getPositions();
        for (int[] pos : positions) {
            int r = pos[0];
            int c = pos[1];
            if (r >= 0 && r < rows && c >= 0 && c < columns) {
                grid[r][c] = piece.getSymbol();
            }
        }
    }

    return grid;
}

public void printBoard() {
    char[][] grid = generateGrid();
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            System.out.print(grid[i][j]);
        }
        System.out.println();
    }
}

public void printPieces() {
    System.out.println("\nDaftar Piece, Koordinat, dan Arah:");
    for (Piece piece : pieces.values()) {
        System.out.print(piece.getSymbol() + " - Direction: " + piece.getDirection() +
                ", Pivot: [" + piece.getPivot()[0] + "," + piece.getPivot()[1] +
                "], Length: " + piece.getLength() + ": ");

        List<int[]> positions = piece.getPositions();
        for (int[] coord : positions) {
            System.out.print("[" + coord[0] + "," + coord[1] + "] ");
        }
        System.out.println();
    }
}
```

```
public boolean canMovePiece(char symbol, String direction) {
    Piece piece = pieces.get(symbol);
    if (piece == null) return false;

    String pieceDirection = piece.getDirection();
    if (pieceDirection.equals("horizontal") && (direction.equalsIgnoreCase("up") || direction.equalsIgnoreCase("down"))) {
        return false;
    }
    if (pieceDirection.equals("vertical") && (direction.equalsIgnoreCase("left") || direction.equalsIgnoreCase("right"))) {
        return false;
    }

    // Generate current grid to check for collisions
    char[][] grid = generateGrid();
    List<int[]> positions = piece.getPositions();

    switch (direction.toLowerCase()) {
        case "up" -> {
            for (int[] pos : positions) {
                int newRow = pos[0] - 1;
                if (newRow < 0) {
                    return false;
                }
                if (grid[newRow][pos[1]] != '.' && grid[newRow][pos[1]] != piece.getSymbol()) {
                    return false;
                }
            }
            return true;
        }
        case "down" -> {
            for (int[] pos : positions) {
                int newRow = pos[0] + 1;
                if (newRow >= rows) {
                    return false;
                }
                if (grid[newRow][pos[1]] != '.' && grid[newRow][pos[1]] != piece.getSymbol()) {
                    return false;
                }
            }
            return true;
        }
```

```
        case "left" -> {
            for (int[] pos : positions) {
                int newCol = pos[1] - 1;
                if (newCol < 0) {
                    return false;
                }
                if (grid[pos[0]][newCol] != '.' && grid[pos[0]][newCol] != piece.getSymbol()) {
                    return false;
                }
            }
            return true;
        }
        case "right" -> {
            for (int[] pos : positions) {
                int newCol = pos[1] + 1;
                if (newCol >= columns) {
                    return false;
                }
                if (grid[pos[0]][newCol] != '.' && grid[pos[0]][newCol] != piece.getSymbol()) {
                    return false;
                }
            }
            return true;
        }
        default -> {
            return false;
        }
    }
}

public Board movePiece(char symbol, String direction) {
    if (!this.canMovePiece(symbol, direction)) {
        System.out.println("Tidak bisa memindahkan kendaraan " + symbol + " ke arah " + direction);
        return null;
    }

    // Create a new board with the same dimensions and properties
    Board newBoard = new Board();
    newBoard.rows = this.rows;
    newBoard.columns = this.columns;
```

```
    newBoard.numPieces = this.numPieces;
    newBoard.exitPos = this.exitPos != null ? new int[]{this.exitPos[0], this.exitPos[1]} : null;

    // Copy all pieces except the one to be moved
    for (Map.Entry<Character, Piece> entry : this.pieces.entrySet()) {
        char pieceSymbol = entry.getKey();
        Piece originalPiece = entry.getValue();

        if (pieceSymbol != symbol) {
            // Deep copy of the piece
            int[] newPivot = originalPiece.getPivot().clone();
            Piece newPiece = new Piece(
                pieceSymbol,
                originalPiece.getDirection(),
                newPivot,
                originalPiece.getLength()
            );
            newBoard.pieces.put(pieceSymbol, newPiece);
        } else {
            // Create a new piece with updated position
            int[] newPivot = originalPiece.getPivot().clone();

            // Update pivot based on direction
            switch (direction.toLowerCase()) {
                case "up" -> newPivot[0] -= 1;
                case "down" -> newPivot[0] += 1;
                case "left" -> newPivot[1] -= 1;
                case "right" -> newPivot[1] += 1;
            }

            Piece newPiece = new Piece(
                symbol,
                originalPiece.getDirection(),
                newPivot,
                originalPiece.getLength()
            );
            newBoard.pieces.put(symbol, newPiece);
        }
    }

    return newBoard;
}
```

```java
public Map<Character, Piece> getPieces() {
    return pieces;
}

public int getRows() {
    return rows;
}

public int getColumns() {
    return columns;
}

public int[] getExitPos() {
    return exitPos;
}

public int getNumPieces() {
    return numPieces;
}

public List<int[]> getPrimaryPiecePosition() {
    Piece primary = pieces.get(PRIMARY_PIECE);
    if (primary != null) {
        return primary.getPositions();
    }
    return Collections.emptyList();
}

public Board removePrimaryPiece() {
    Board newBoard = new Board();
    newBoard.rows = this.rows;
    newBoard.columns = this.columns;
    newBoard.numPieces = this.numPieces - 1;
    newBoard.exitPos = this.exitPos != null ? new int[]{this.exitPos[0], this.exitPos[1]} : null;

    for (Map.Entry<Character, Piece> entry : this.pieces.entrySet()) {
        char pieceSymbol = entry.getKey();
        if (pieceSymbol != PRIMARY_PIECE) {
            Piece originalPiece = entry.getValue();
            int[] newPivot = originalPiece.getPivot().clone();
            Piece newPiece = new Piece(
                originalPiece.getDirection(),
                newPivot,
                originalPiece.getLength()
            );
            newBoard.pieces.put(pieceSymbol, newPiece);
        }
    }

    return newBoard;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Board board = (Board) o;
    return Arrays.deepEquals(this.generateGrid(), board.generateGrid());
}

@Override
public int hashCode() {
    return Arrays.deepHashCode(generateGrid());
}

public void readInputFromFileGUI(String filePath) throws FileNotFoundException {
    try (Scanner fileScanner = new Scanner(new File(filePath))) {
        if (!fileScanner.hasNextLine()) {
            throw new IllegalArgumentException(s:"Error: File kosong atau tidak valid.");
        }

        String[] dimensions = fileScanner.nextLine().split(regex:" ");
        if (dimensions.length != 2) {
            throw new IllegalArgumentException(s:"Error: Baris pertama harus terdiri dari dimensi papan (A B).");
        }

        try {
            rows = Integer.parseInt(dimensions[0]);
            columns = Integer.parseInt(dimensions[1]);
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException(s:"Error: Dimensi papan harus berupa bilangan bulat.");
        }
```

```
        if (!fileScanner.hasNextLine()) {
            throw new IllegalArgumentException(s:"Error: Jumlah kendaraan tidak ada atau tidak sesuai format.");
        }

        try {
            numPieces = Integer.parseInt(fileScanner.nextLine().trim());
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException(s:"Error: Jumlah kendaraan tidak ada atau tidak sesuai format.");
        }

        Map<Character, List<int[]>> pieceCoordinates = new HashMap<>();
        int currentRow = 0;

        while (fileScanner.hasNextLine()) {
            String line = fileScanner.nextLine();
            try {
                boolean validLine = processLine(line, currentRow, pieceCoordinates);
                if (!validLine) {
                    throw new IllegalArgumentException("Error: Format baris ke-" + currentRow + " tidak valid.");
                }
            } catch (IllegalArgumentException e) {
                throw new IllegalArgumentException("Error saat memproses baris ke-" + currentRow + ": " + e.getMessage());
            }
            currentRow++;
        }

        try {
            if (!verifyBoardConstraints(pieceCoordinates)) {
                throw new IllegalArgumentException(s:"Papan tidak sesuai format. Periksa file input.");
            }
        } catch (IllegalArgumentException e) {
            throw new IllegalArgumentException(s:"Papan tidak sesuai format. Periksa file input.");
        }

        try {
            String exitDirection = getExitDirection(pieceCoordinates);
            String primaryDir = determineDirection(pieceCoordinates.get(PRIMARY_PIECE));
            boolean match = (primaryDir.equals(anObject:"vertical") && (exitDirection.equals(anObject:"up") || exitDirection.equals(anObject:"bottom")))
                    || (primaryDir.equals(anObject:"horizontal") && (exitDirection.equals(anObject:"left") || exitDirection.equals(anObject:"right")));
            if (!match) {
                throw new IllegalArgumentException("Arah kendaraan utama 'P' (" +
                    primaryDir +
                    primaryDir +
                    ") tidak sesuai dengan arah pintu keluar (" + exitDirection + ").");
            }

            // if horizontal, all 'P' must be on the same row as exit; if vertical, same column
            List<int[]> primaryCoords = pieceCoordinates.get(PRIMARY_PIECE);
            int[] exit = pieceCoordinates.get(EXIT).get(index:0);
            if (primaryDir.equals(anObject:"horizontal")) {
                for (int[] coord : primaryCoords) {
                    if (coord[0] != exit[0]) {
                        throw new IllegalArgumentException(s:"Kendaraan utama 'P' harus berada pada baris yang sama dengan pintu keluar.");
                    }
                }
            } else if (primaryDir.equals(anObject:"vertical")) {
                for (int[] coord : primaryCoords) {
                    if (coord[1] != exit[1]) {
                        throw new IllegalArgumentException(s:"Kendaraan utama 'P' harus berada pada kolom yang sama dengan pintu keluar.");
                    }
                }
            }

        } catch (IllegalArgumentException e) {
            throw new IllegalArgumentException(e.getMessage());
        }

        try {
            normalizeCoordinates(pieceCoordinates);
        } catch (Exception e) {
            throw new IllegalArgumentException(s:"Error: Posisi piece di luar board.");
        }

        try {
            if (!validateNormalizedCoordinates(pieceCoordinates)) {
                throw new IllegalArgumentException(s:"Error: Posisi piece di luar board.");
            }
        } catch (IllegalArgumentException e) {
            throw new IllegalArgumentException(s:"Error: Posisi piece di luar board.");
        }

        try {
            createPieces(pieceCoordinates);
        } catch (Exception e) {
            throw new IllegalArgumentException(s:"Error saat membuat pieces");
        }
```

```
            if ((pieces.size() - 1) != numPieces) {
                throw new IllegalArgumentException("Jumlah kendaraan dalam file (" +
                    (pieces.size() - 1) + ") tidak sesuai dengan yang dinyatakan (" + numPieces + ")");
            }

            if (exitPos == null) {
                throw new IllegalArgumentException(s:"Pintu keluar 'K' tidak ditemukan.");
            }

        } catch (FileNotFoundException e) {
            throw new FileNotFoundException("File tidak ditemukan: " + filePath);
        } catch (Exception e) {
            throw new IllegalArgumentException(e.getMessage());
        }
    }
```

3. AStar.java

```
public class AStar extends Solver {
    public AStar() {
        super();
    }

    @Override
    protected void solve(State root, int[] counter, int mode) {
        Heuristic heuristic;
        switch (mode) {
            case 1 -> heuristic = new BlockerOnly();
            case 2 -> heuristic = new ManhattanDistance();
            case 3 -> heuristic = new CombinedHeuristic();
            default -> throw new IllegalArgumentException("Invalid mode: " + mode);
        }

        PrioQueue queue = new PrioQueue(capacity:100);
        root.setTotalCost(heuristic.calculate(root) + root.getCountSteps());
        queue.enqueue(root);

        while (!queue.isEmpty()) {
            State current = queue.dequeue();

            if (isVisited(current.getCurrBoard())) continue;
            addToVisited(current.getCurrBoard());
            counter[0] += 1;

            if (current.isGoalState()) {
                State finalState = current.removePrimaryPieceState();
                buildPath(finalState);
                return;
            }

            for (State succ : current.getSuccessors()) {
                succ.setTotalCost(heuristic.calculate(succ) + succ.getCountSteps());
                queue.enqueue(succ);
            }
        }
    }
```

4. CombinedHeuristic

```
public class CombinedHeuristic implements Heuristic {

    private final ManhattanDistance md = new ManhattanDistance();
    private final BlockerOnly bo = new BlockerOnly();

    @Override
    public int calculate(State state) {
        return md.calculate(state) + bo.calculate(state);
    }
}
```

5. GBFS.java

```java
public class GBFS extends Solver {

    public GBFS() {
        super();
    }

    @Override
    protected void solve(State root, int[] counter, int mode) {
        Heuristic heuristic;
        switch (mode) {
            case 1 -> heuristic = new BlockerOnly();
            case 2 -> heuristic = new ManhattanDistance();
            case 3 -> heuristic = new CombinedHeuristic();
            default -> throw new IllegalArgumentException("Invalid mode: " + mode);
        }

        PrioQueue queue = new PrioQueue(capacity:100);
        root.setTotalCost(heuristic.calculate(root));
        queue.enqueue(root);

        while (!queue.isEmpty()) {
            State current = queue.dequeue();

            if (isVisited(current.getCurrBoard())) continue;
            addToVisited(current.getCurrBoard());
            counter[0] += 1;

            if (current.isGoalState()) {
                State finalState = current.removePrimaryPieceState();
                buildPath(finalState);
                return;
            }

            for (State succ : current.getSuccessors()) {
                succ.setTotalCost(heuristic.calculate(succ));
                queue.enqueue(succ);
            }
        }
    }
```

6. Heuristic.java

```java
public interface Heuristic {
    int calculate(State state);
}
```

7. IDS.java

```java
public class IDS extends Solver {

    private static final int FOUND = 1;
    private static final int CUTOFF = 0;
    private static final int FAILURE = -1;

    public IDS() {
        super();
    }

    @Override
    protected void solve(State root, int[] counter, int mode) {
        int depthLimit = 0;

        while (true) {
            System.out.println(x:"Prosessing2....");
            int result = depthLimitedSearch(root, depthLimit, depth:0, counter);
            if (result == FOUND) {
                break;
            }
            if (result == FAILURE) {
                System.out.println(x:"All nodes explored. No solution.");
                break;
            }
            depthLimit++;
        }
    }
```

```java
    private int depthLimitedSearch(State current, int limit, int depth, int[] counter) {
        counter[0]++; // count node
        System.out.println(x:"Prosessing....");
        if (current.isGoalState()) {
            buildPath(current.removePrimaryPieceState());
            return FOUND;
        }

        if (depth == limit) {
            return CUTOFF;
        }

        boolean anyCutoff = false;
        for (State succ : current.getSuccessors()) {
            succ.setPrevState(current);
            int result = depthLimitedSearch(succ, limit, depth + 1, counter);
            if (result == FOUND) {
                return FOUND;
            }
            if (result == CUTOFF) {
                anyCutoff = true;
            }

        }

        return anyCutoff ? CUTOFF : FAILURE;
    }
```

8. BlockerOnly.java

```java
public class BlockerOnly implements Heuristic {

    @Override
    public int calculate(State state) {
        Board b = state.getCurrBoard();
        List<int[]> pos = b.getPrimaryPiecePosition();
        int[] exitPos = b.getExitPos();
        char[][] currBoard = b.generateGrid();

        int[] closest = pos.get(index:0);
        int minimalDistance = Integer.MAX_VALUE;

        for (int[] piecePos : pos) {
            int dist = Math.abs(piecePos[0] - exitPos[0]) + Math.abs(piecePos[1] - exitPos[1]);
            if (dist < minimalDistance) {
                minimalDistance = dist;
                closest = piecePos;
            }
        }

        String orientation = b.determineDirection(pos);
        int blockers = 0;

        if ("horizontal".equals(orientation)) {
            int row = closest[0];
            int startCol = closest[1];
            int endCol = exitPos[1];
            int step = (endCol > startCol) ? 1 : -1;

            char currentChar = currBoard[row][startCol];
            for (int col = startCol + step; col != endCol + step; col += step) {
                if (col >= 0 && col < currBoard[0].length) {
                    if (currBoard[row][col] != '.' && currBoard[row][col] != currentChar) {
                        blockers++;
                    }
                }
            }

        } else if ("vertical".equals(orientation)) {
            int col = closest[1];
            int startRow = closest[0];
            int endRow = exitPos[0];
            int step = (endRow > startRow) ? 1 : -1;

            char currentChar = currBoard[startRow][col];
            for (int row = startRow + step; row != endRow + step; row += step) {
                if (row >= 0 && row < currBoard.length) {
                    if (currBoard[row][col] != '.' && currBoard[row][col] != currentChar) {
                        blockers++;
                    }
                }
            }
        }

        return blockers;
    }
}
```

9. ManhattanDistance.java

```java
public class ManhattanDistance implements Heuristic {

    @Override
    public int calculate(State state) {
        Board b = state.getCurrBoard();
        List<int[]> pos = b.getPrimaryPiecePosition();
        int[] exitPos = b.getExitPos();

        int minimalDistance = Integer.MAX_VALUE;
        for (int[] piecePos : pos) {
            int dist = Math.abs(piecePos[0] - exitPos[0]) + Math.abs(piecePos[1] - exitPos[1]);
            minimalDistance = Math.min(minimalDistance, dist);
        }

        return minimalDistance;
    }
}
```

10. Piece.java

```java
public class Piece {
    private char symbol;
    private String direction;
    private int[] pivot; // [row, col] of the smallest coordinate
    private int length;

    public Piece(char symbol, String direction, int[] pivot, int length) {
        this.symbol = symbol;
        this.direction = direction;
        this.pivot = pivot;
        this.length = length;
    }

    public char getSymbol() {
        return symbol;
    }

    public String getDirection() {
        return direction;
    }

    public int[] getPivot() {
        return pivot;
    }

    public int getLength() {
        return length;
    }

    public List<int[]> getPositions() {
        List<int[]> positions = new ArrayList<>();
        if (direction.equals(anObject:"horizontal")) {
            for (int i = 0; i < length; i++) {
                positions.add(new int[]{pivot[0], pivot[1] + i});
            }
        } else { // vertical
            for (int i = 0; i < length; i++) {
                positions.add(new int[]{pivot[0] + i, pivot[1]});
            }
        }
        return positions;
    }
```

```java
    public void setSymbol(char symbol) {
        this.symbol = symbol;
    }

    public void setDirection(String direction) {
        this.direction = direction;
    }

    public void setPivot(int[] pivot) {
        this.pivot = pivot;
    }

    public void setLength(int length) {
        this.length = length;
    }

    public void moveUp() {
        pivot[0] -= 1;
    }

    public void moveDown() {
        pivot[0] += 1;
    }

    public void moveLeft() {
        pivot[1] -= 1;
    }

    public void moveRight() {
        pivot[1] += 1;
    }
}
```

11. PrioQueue.java

```java
public class PrioQueue {
    private State[] queue;
    private int size;
    private int capacity;

    public PrioQueue(int capacity) {
        this.capacity = capacity;
        this.queue = new State[capacity];
        this.size = 0;
    }

    public void enqueue(State state) {
        if (size == capacity) {
            resize();
        }
        queue[size] = state;
        size++;
        moveUp(size - 1);
    }


    private void resize() {
        capacity *= 2;
        State[] newQueue = new State[capacity];
        System.arraycopy(queue, srcPos:0, newQueue, destPos:0, size);
        queue = newQueue;
    }


    public State dequeue() {
        if (size == 0) {
            throw new IllegalStateException(s:"Queue is empty");
        }
        State root = queue[0];
        queue[0] = queue[size - 1];
        size--;
        moveDown(index:0);
        return root;
    }

    public boolean isEmpty() {
        return size == 0;
    }
```

```
private void moveUp(int index) {
    while (index > 0) {
        int parentIndex = (index - 1) / 2;
        if (queue[index].getTotalCost() < queue[parentIndex].getTotalCost()) {
            swap(index, parentIndex);
            index = parentIndex;
        } else {
            break;
        }
    }
}

private void moveDown(int index) {
    while (index < size) {
        int leftChildIndex = 2 * index + 1;
        int rightChildIndex = 2 * index + 2;
        int smallestIndex = index;

        if (leftChildIndex < size && queue[leftChildIndex].getTotalCost() < queue[smallestIndex].getTotalCost()) {
            smallestIndex = leftChildIndex;
        }
        if (rightChildIndex < size && queue[rightChildIndex].getTotalCost() < queue[smallestIndex].getTotalCost()) {
            smallestIndex = rightChildIndex;
        }
        if (smallestIndex != index) {
            swap(index, smallestIndex);
            index = smallestIndex;
        } else {
            break;
        }
    }
}

private void swap(int i, int j) {
    State temp = queue[i];
    queue[i] = queue[j];
    queue[j] = temp;
}
```

12. Result.java

```
public class Result {
    public LinkedList<State> solutionStep;
    public double time;
    public int nodes;

    public Result(double time, int nodes, LinkedList<State> solutionStep) {
        this.time = time;
        this.nodes = nodes;
        this.solutionStep = solutionStep;
    }
}
```

13. Solver.java

```java
public abstract class Solver {
    protected Set<Board> visited = new HashSet<>();
    protected LinkedList<State> finalPath = new LinkedList<>();

    public LinkedList<State> getFinalPath() {
        return finalPath;
    }

    public void addToVisited(Board b) {
        visited.add(b);
    }

    public boolean isVisited(Board b) {
        return visited.contains(b);
    }

    protected void buildPath(State goal) {
        State current = goal;
        while (current != null) {
            finalPath.addFirst(current);
            current = current.getPrevState();
        }
    }

    public Result run(Board board) {
        return run(board, -1);
    }

    public Result run(Board board, int mode) {
        State root = new State(board);
        int[] counter = new int[1];

        long startTime = System.nanoTime();
        solve(root, counter, mode);
        long endTime = System.nanoTime();

        double durationMs = (endTime - startTime) / 1_000_000.0;
        return new Result(durationMs, counter[0], getFinalPath());
    }

    protected abstract void solve(State root, int[] counter, int mode);
}
```

```java
    public String getMoveDirection() {
        return moveDirection;
    }

    public void setMoveDirection(String moveDirection) {
        this.moveDirection = moveDirection;
    }

    public char getMovedPiece() {
        return movedPiece;
    }

    public void setMovedPiece(char movedPiece) {
        this.movedPiece = movedPiece;
    }

    public int getTotalCost() {
        return totalCost;
    }

    public void setTotalCost(int totalCost) {
        this.totalCost = totalCost;
    }

    public int getCountSteps() {
        return countSteps;
    }

    public void setCountSteps(int countSteps) {
        this.countSteps = countSteps;
    }
```

14. State.java

```java
public class State {
    private State prevState;
    private Board currBoard;
    private String moveDirection;  // "Up", "Down", "Left", "Right"
    private char movedPiece;       // 'A', 'B', etc.
    private int totalCost;
    private int countSteps;

    public State(Board currBoard) {
        this.prevState = null;
        this.currBoard = currBoard;
        this.moveDirection = "";
        this.movedPiece = ' ';
        this.totalCost = 0;
        this.countSteps = 0;
    }

    public State(State prevState, Board currBoard, String moveDirection, char movedPiece, int countSteps) {
        this.prevState = prevState;
        this.currBoard = currBoard;
        this.moveDirection = moveDirection;
        this.movedPiece = movedPiece;
        this.totalCost = 0;
        this.countSteps = countSteps;
    }

    public State getPrevState() {
        return prevState;
    }

    public void setPrevState(State prevState) {
        this.prevState = prevState;
    }

    public Board getCurrBoard() {
        return currBoard;
    }

    public void setCurrBoard(Board currBoard) {
        this.currBoard = currBoard;
    }
```

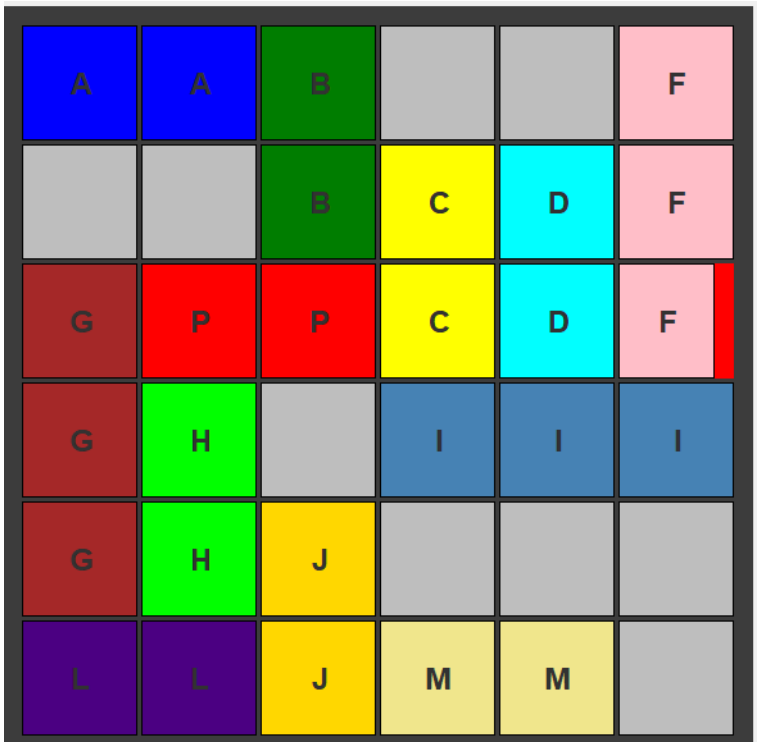
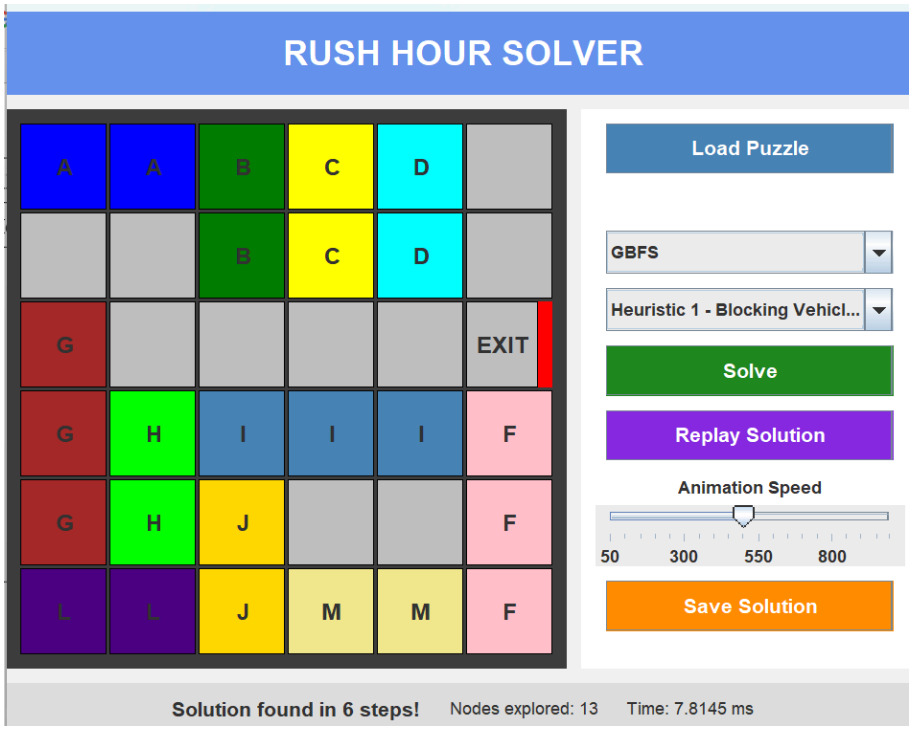
```java
    public boolean isGoalState() {
        Board board = this.currBoard;
        if (board == null) return false;

        int[] exitPos = board.getExitPos();
        if (exitPos == null) return false;

        List<int[]> primaryPositions = board.getPrimaryPiecePosition();
        for (int[] position : primaryPositions) {
            if (position[0] == exitPos[0] && position[1] == exitPos[1]) {
                return true; // Goal state reached
            }
        }
        return false;
    }

    public State removePrimaryPieceState() {
        Board newBoard = this.currBoard.removePrimaryPiece();
        return new State(
            this,
            newBoard,
            this.moveDirection,
            this.movedPiece,
            this.countSteps
        );
    }
```

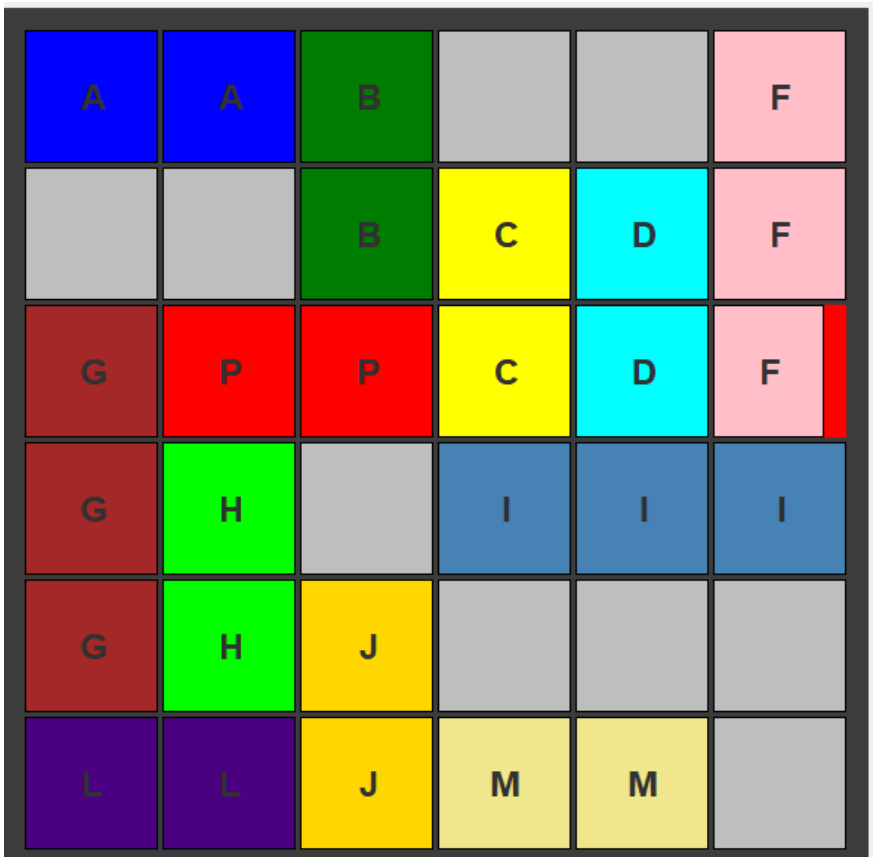
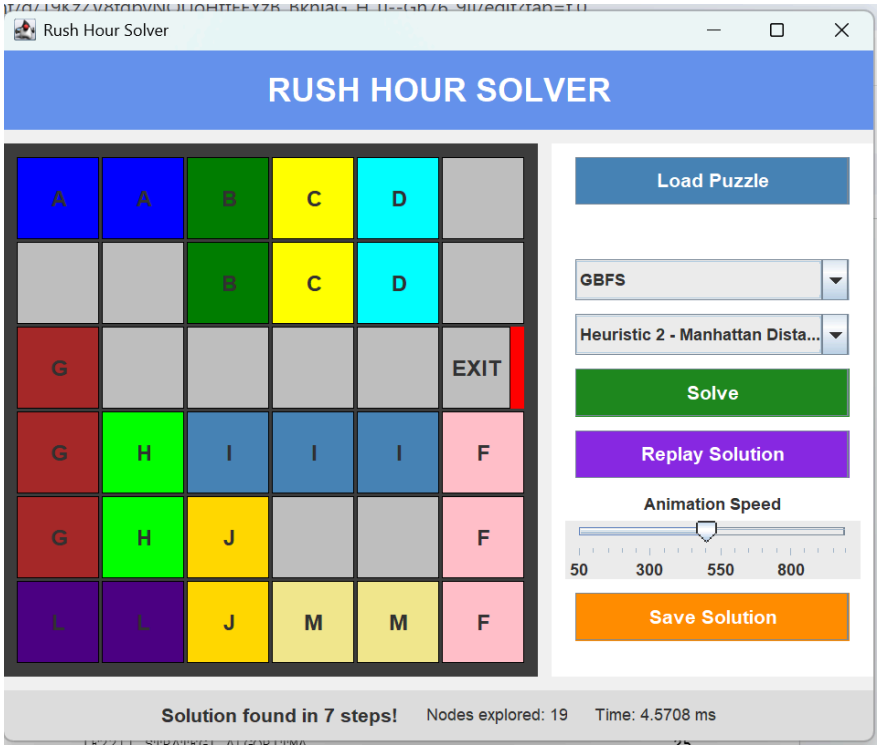
```java
    public List<State> getSuccessors() {
        Board currentBoard = this.currBoard;
        if (currentBoard == null) return Collections.emptyList();

        Map<Character, Piece> pieces = currentBoard.getPieces();
        List<State> successors = new ArrayList<>();

        for (Map.Entry<Character, Piece> entry : pieces.entrySet()) {
            char pieceSymbol = entry.getKey();
            Piece piece = entry.getValue();
            String pieceDirection = piece.getDirection();

            String[] directionsToTry;
            if (pieceDirection == null) {
                directionsToTry = new String[]{"up", "down", "left", "right"};
            } else {
                switch (pieceDirection.toLowerCase()) {
                    case "horizontal" -> directionsToTry = new String[]{"left", "right"};
                    case "vertical" -> directionsToTry = new String[]{"up", "down"};
                    default -> directionsToTry = new String[]{"up", "down", "left", "right"};
                }
            }

            for (String direction : directionsToTry) {
                Board tempBoard = currentBoard;
                while (tempBoard.canMovePiece(pieceSymbol, direction)) {
                Board newBoard = tempBoard.movePiece(pieceSymbol, direction);
                if (newBoard == null) {
                    break;
                }


                State newState = new State(
                    this,
                    newBoard,
                    direction,
                    pieceSymbol,
                    this.countSteps + 1
                );
                successors.add(newState);

                tempBoard = newBoard;
            }
        }
    }
        return successors;
    }
}
```

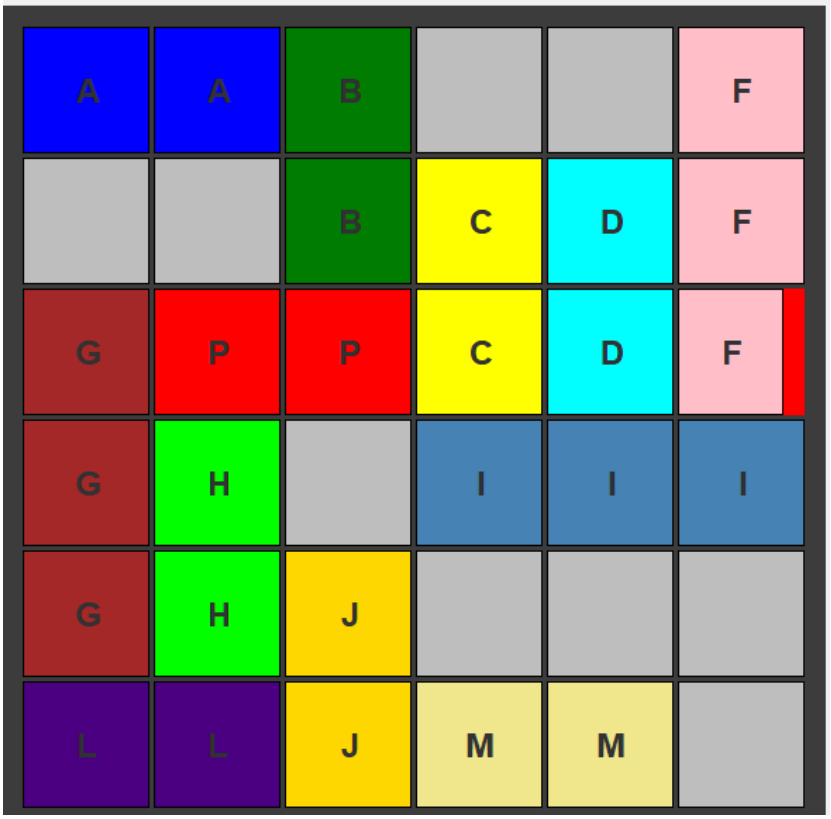
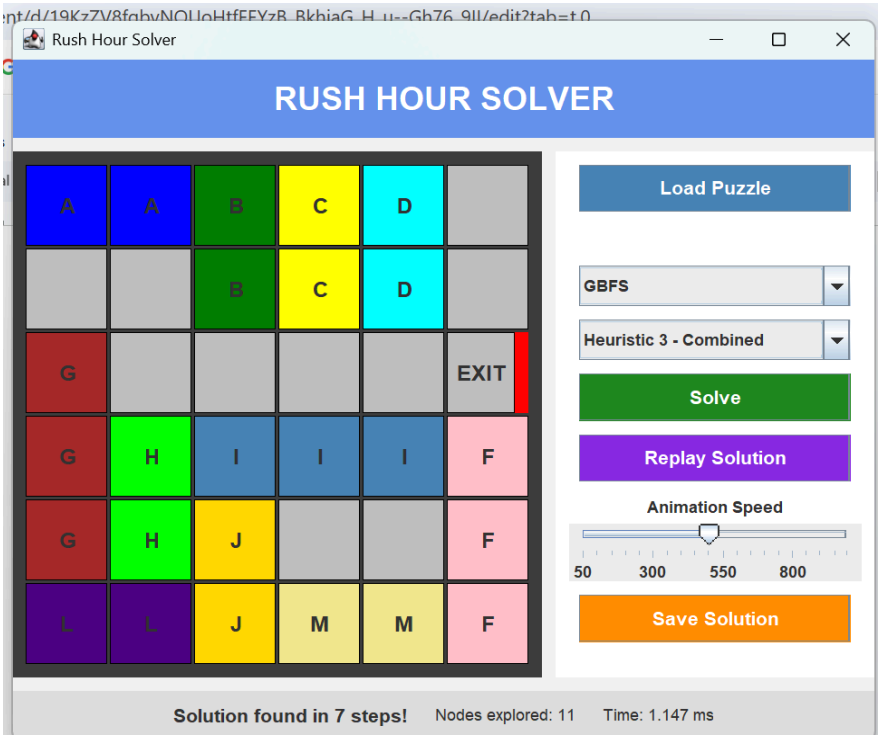
15. UCS.java

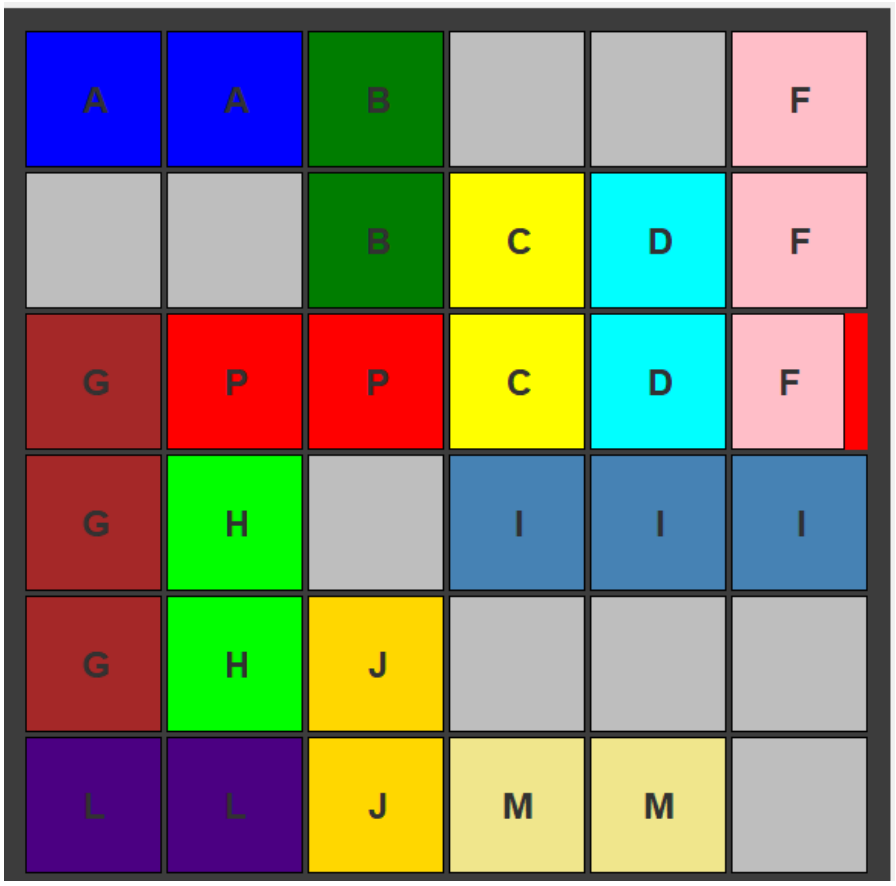
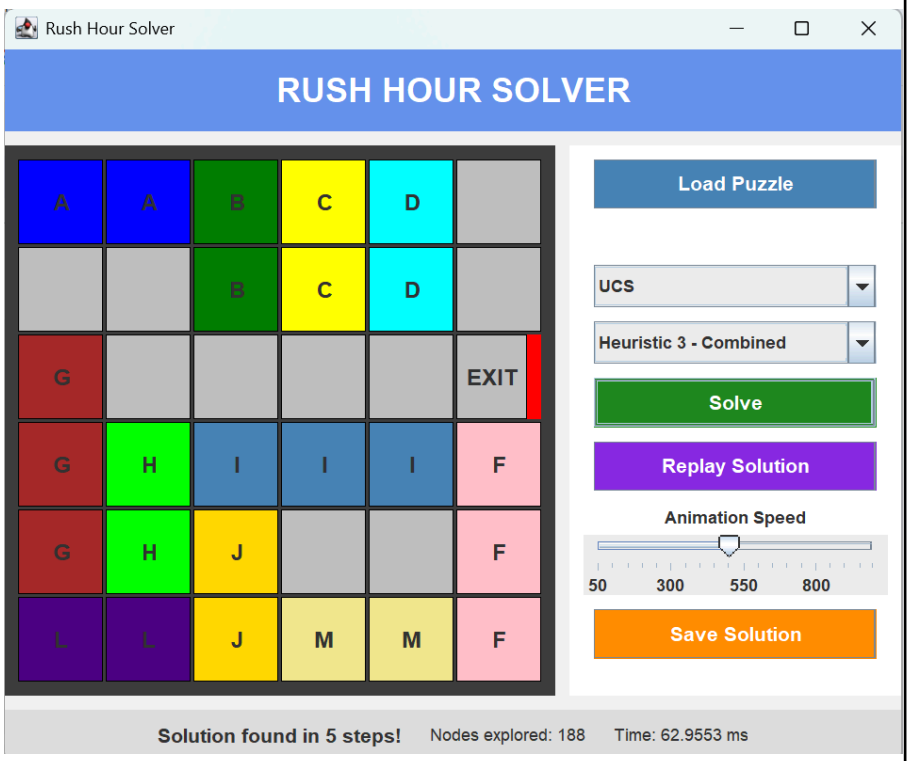
```java
public class UCS extends Solver{
    public UCS() {
        super();
    }

    @Override
    protected void solve(State root, int[] counter, int modeIngnored) {

        PrioQueue queue = new PrioQueue(capacity:100);
        root.setTotalCost(root.getCountSteps());
        queue.enqueue(root);

        while (!queue.isEmpty()) {
            State current = queue.dequeue();

            if (isVisited(current.getCurrBoard())) continue;
            addToVisited(current.getCurrBoard());
            counter[0] += 1;

            if (current.isGoalState()) {
                State finalState = current.removePrimaryPieceState();
                buildPath(finalState);
                return;
            }

            for (State succ : current.getSuccessors()) {
                succ.setTotalCost(succ.getCountSteps());
                queue.enqueue(succ);
            }
        }
    }
```

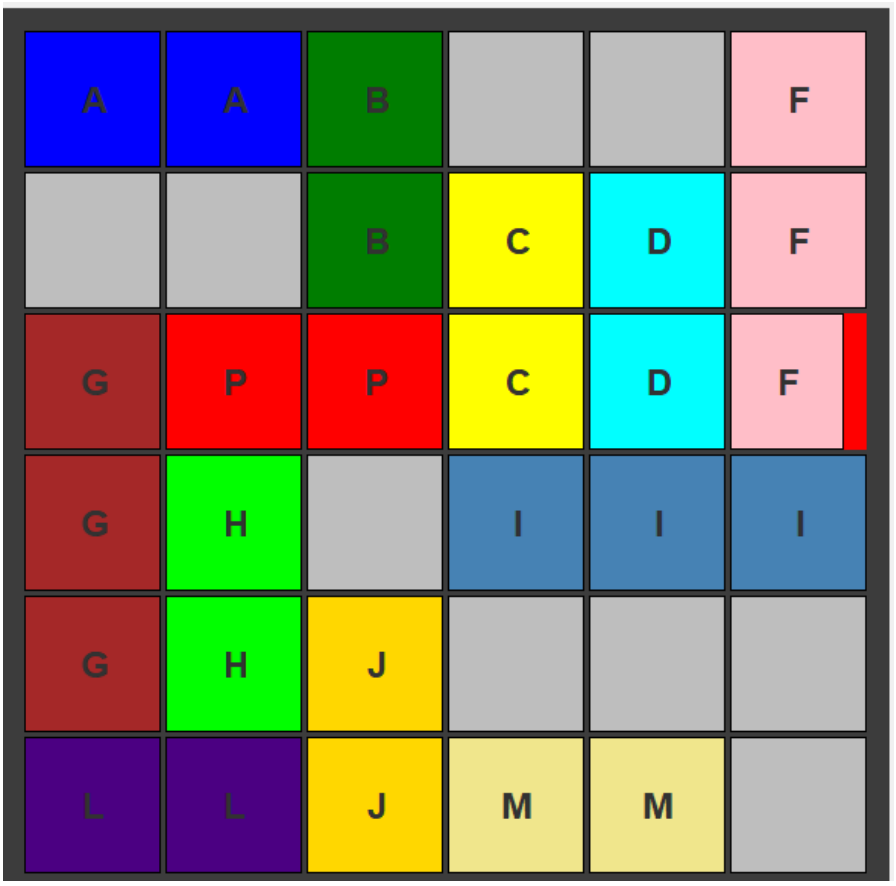
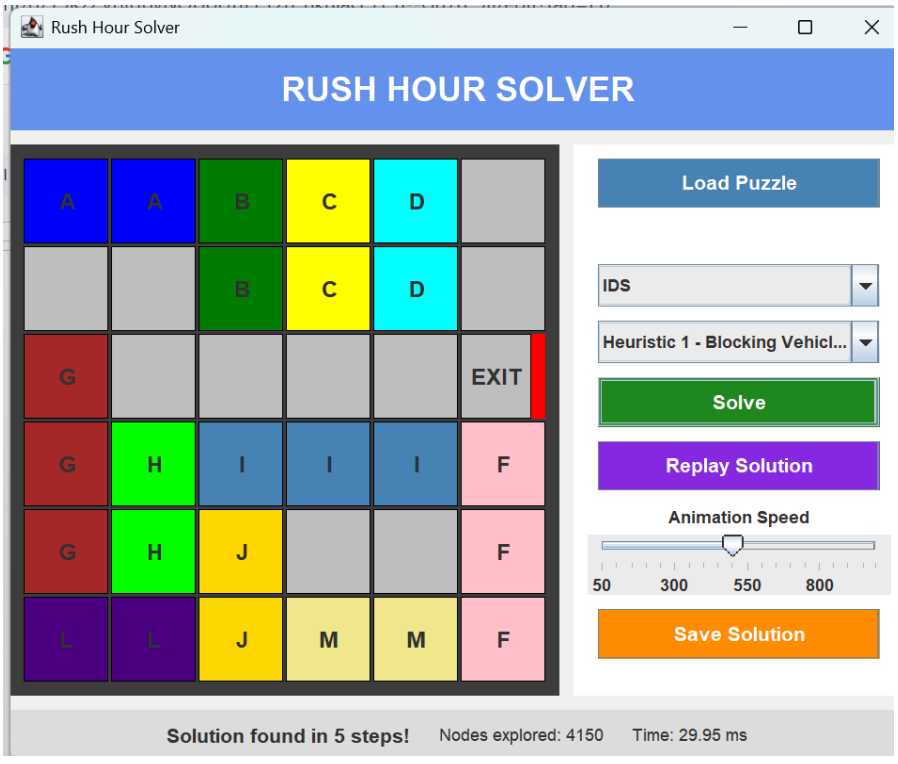
BAB III
SCREENSHOT HASIL TEST

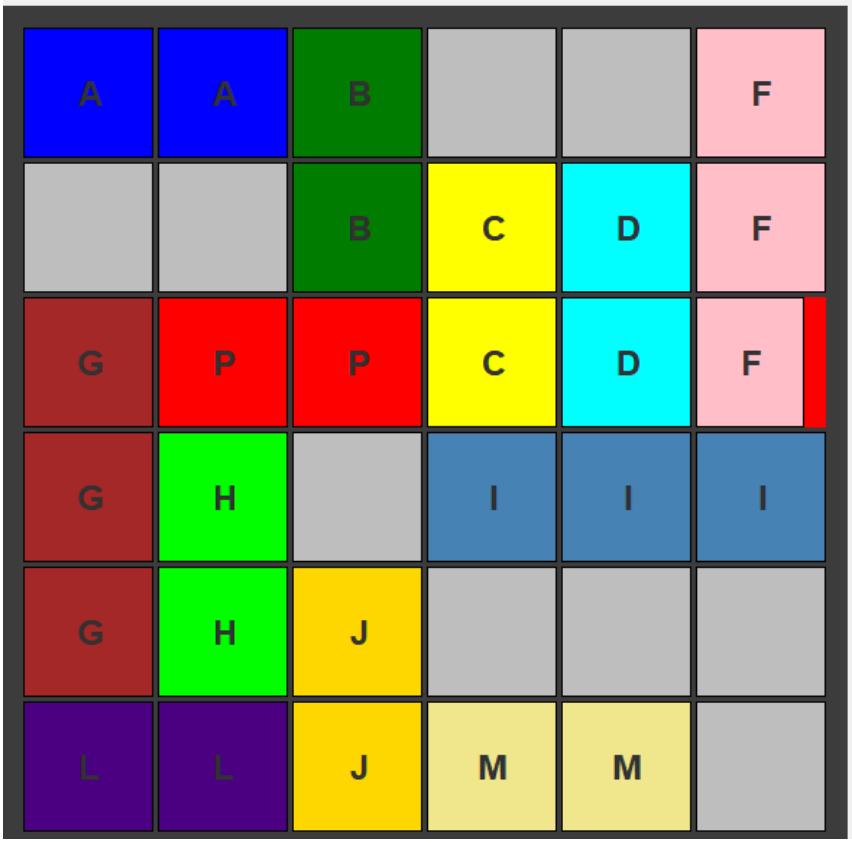
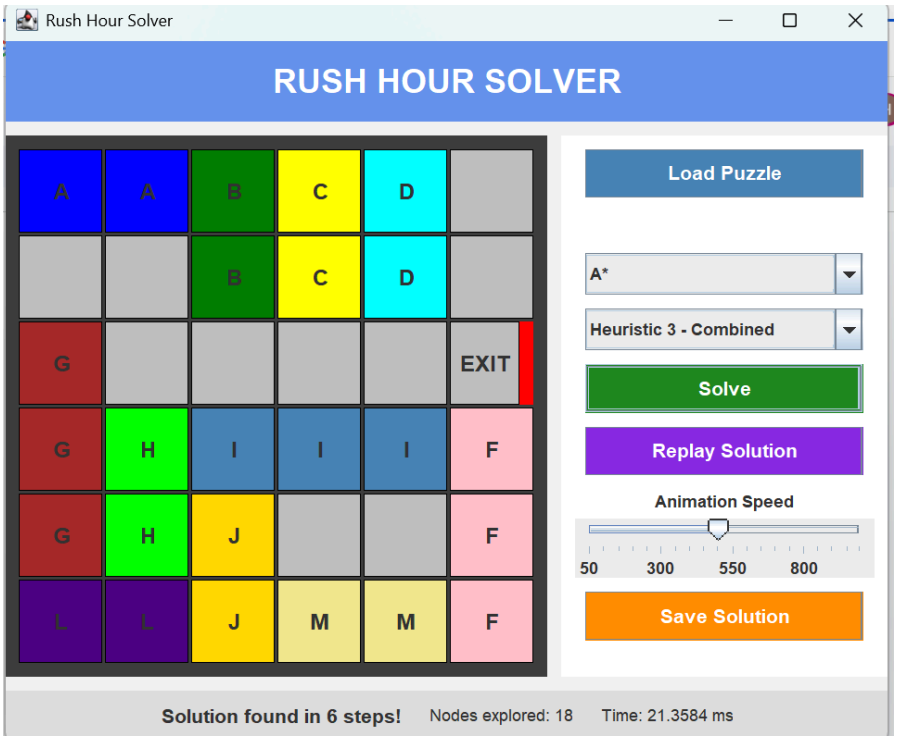
Algoritma: Greedy Best First Search	
Heuristic: Blocking Vehicle	
 	Input: tcspek.txt 6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.

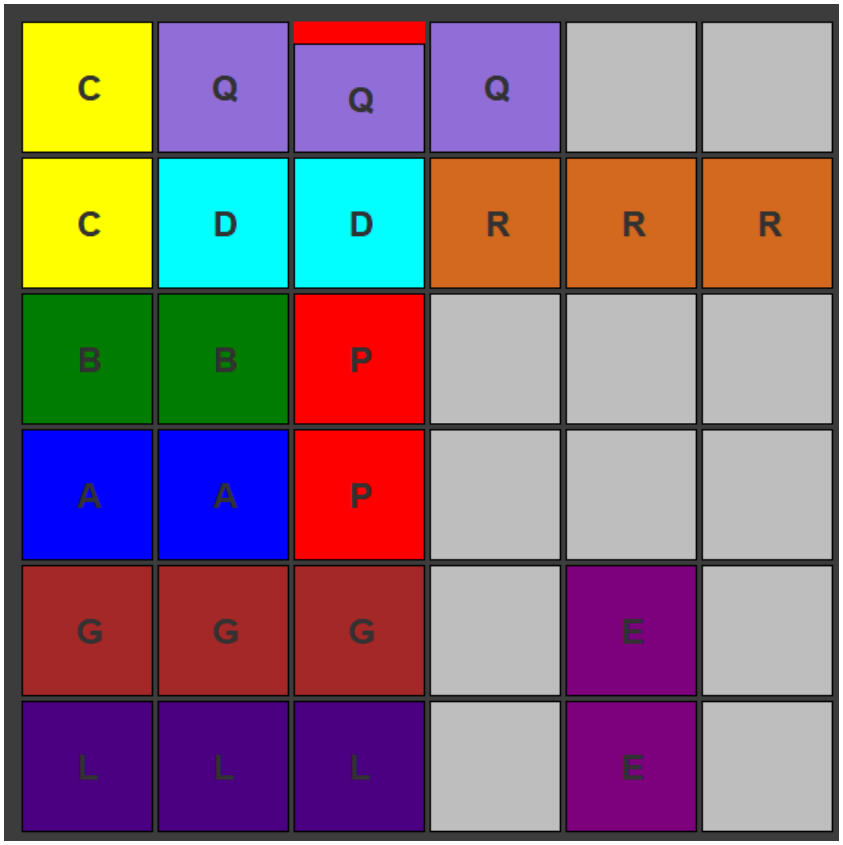
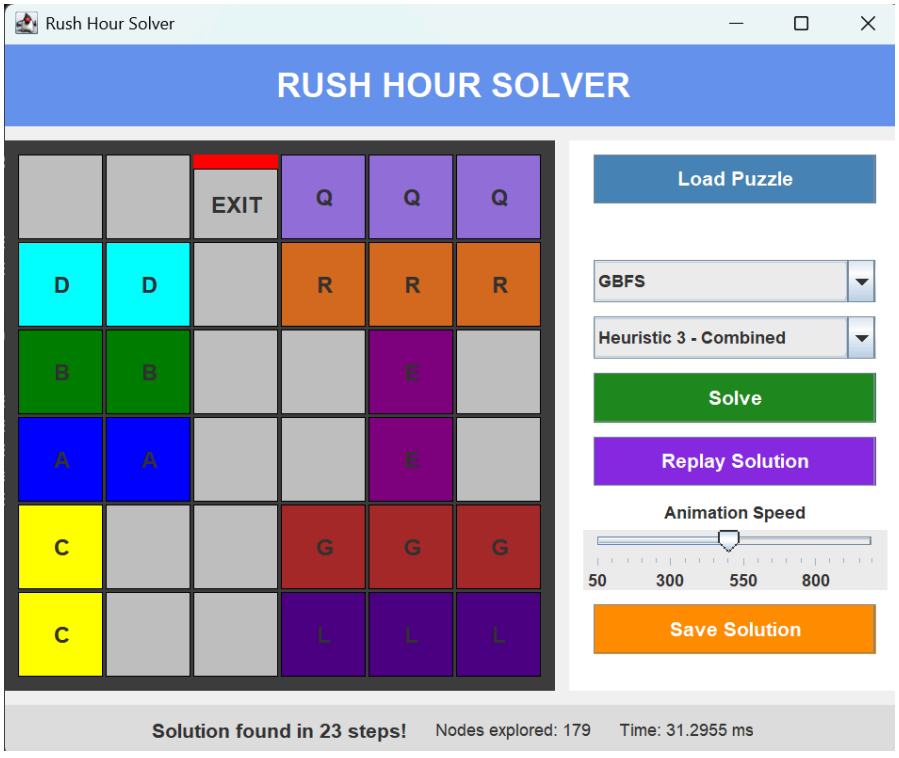
Algoritma: Greedy Best First Search	
Heuristic: Manhattan distance	
 	Input: tcspek.txt 6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.

Algoritma: Greedy Best First Search (GBFS)	
Heuristic: Combine	
 	Input: tcspek.txt 6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.

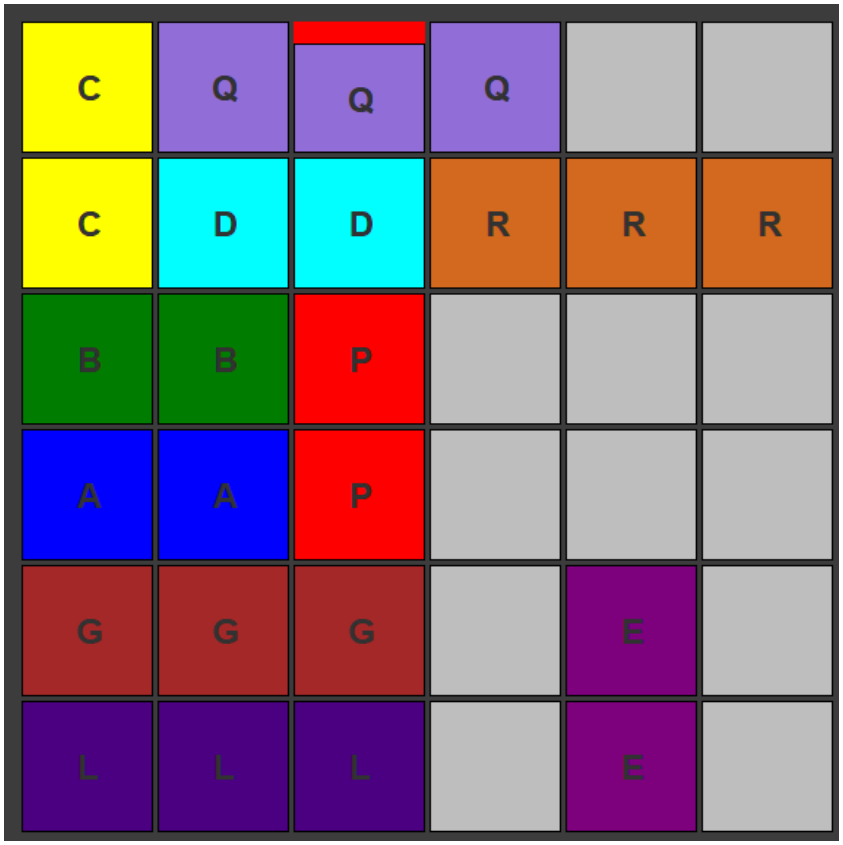
Algoritma: Uniform Cost Search (UCS)	
Heuristic: -	
 	Input: tcspek.txt 6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.

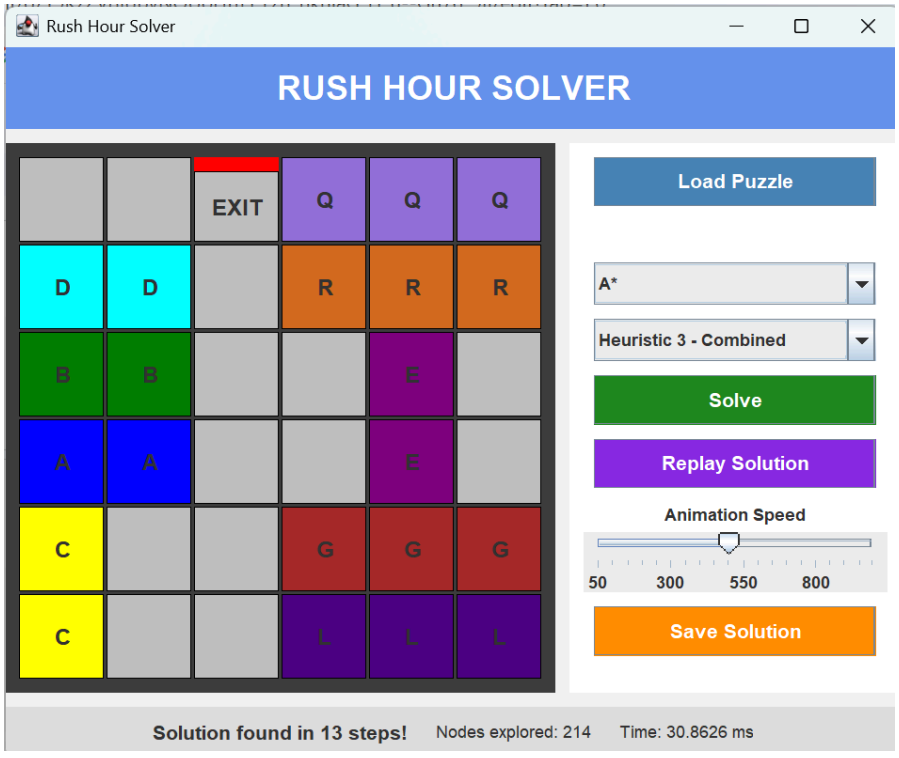
Algoritma: Iterative Deepening Search (IDS)	
Heuristic: -	
 	Input: tcspek.txt 6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.

Algoritma: A*	
Heuristic: Combined	
  	Input: tcspek.txt 6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.

Algoritma: GBFS	
Heuristic: Combine	
 	Input: tc3.txt 6 6 9 K CQQQ.. CDDRRR BBP... AAP... GGG.E. LLL.E.

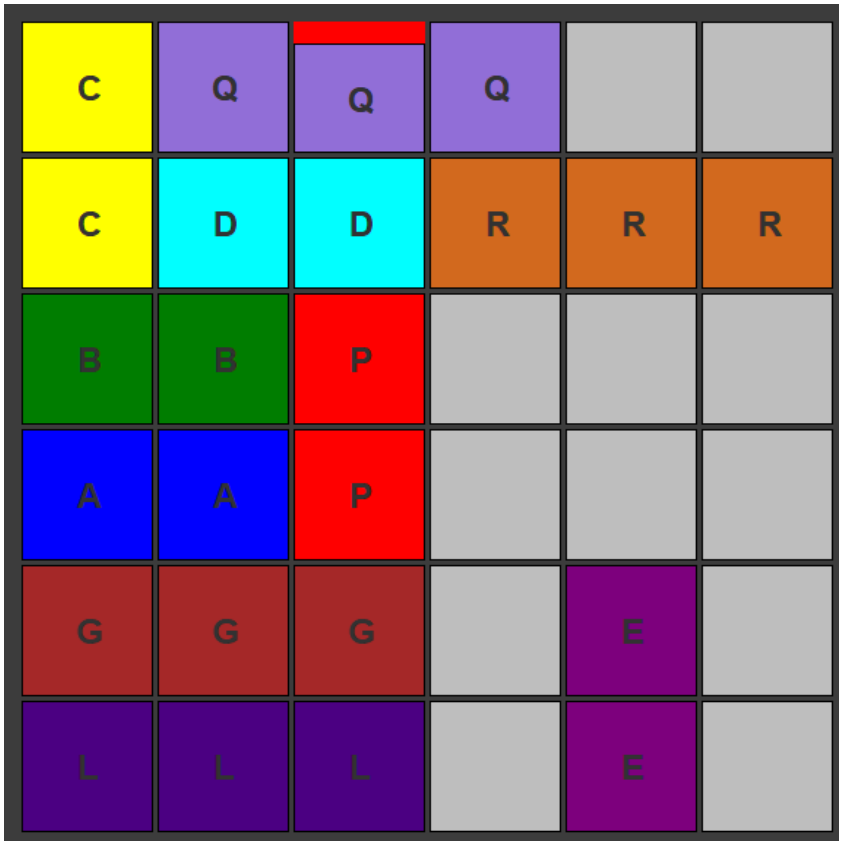
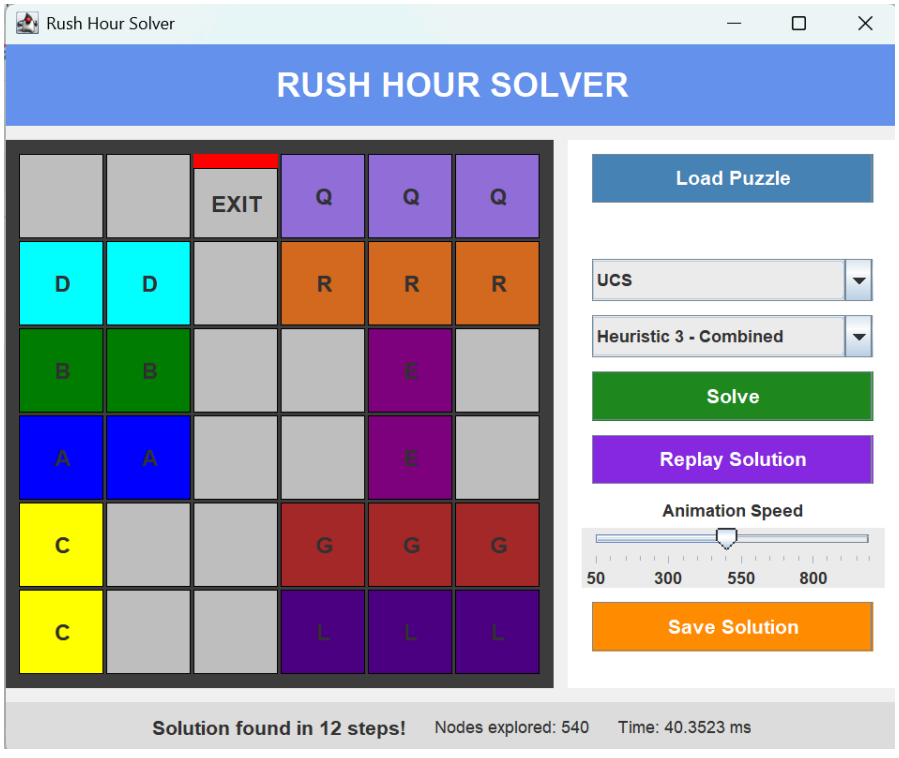
Algoritma: A*
Heuristic: Combine

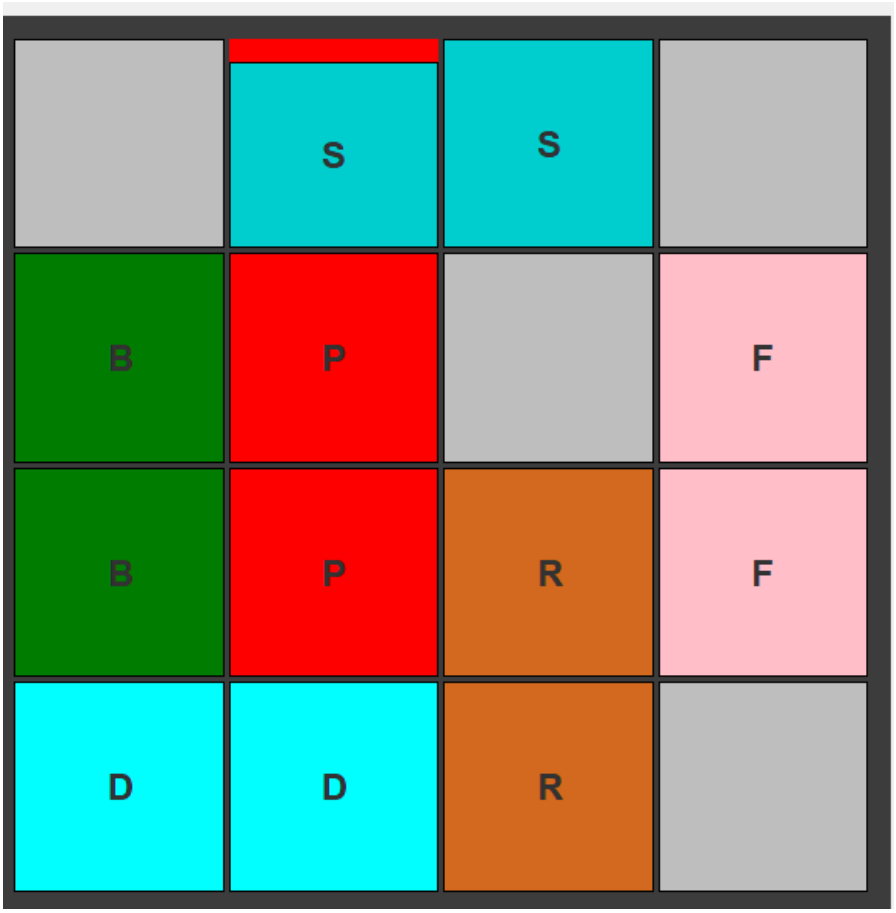
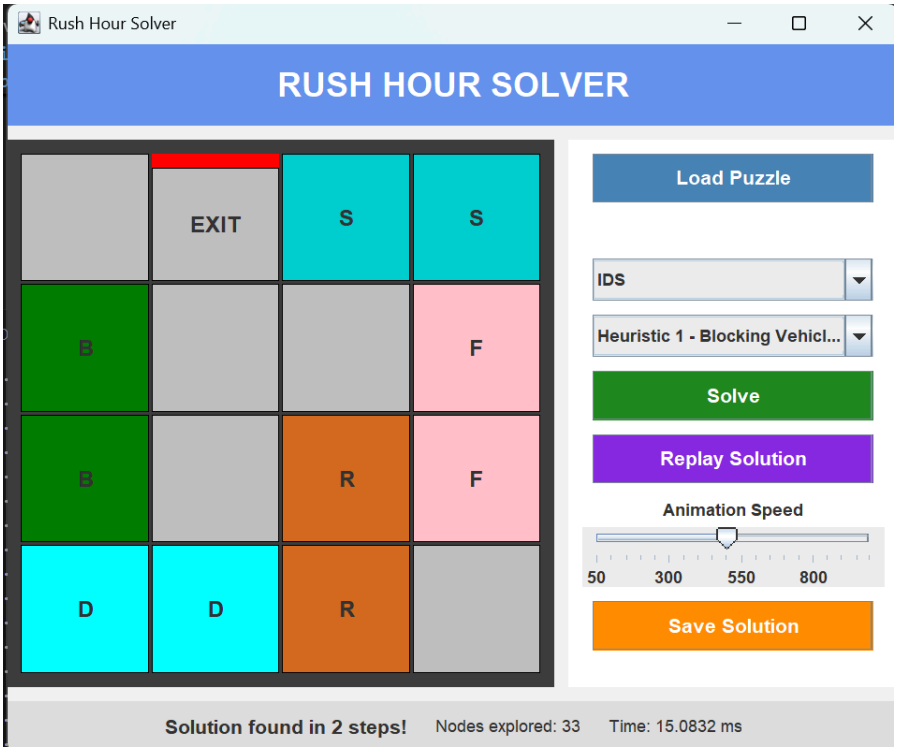


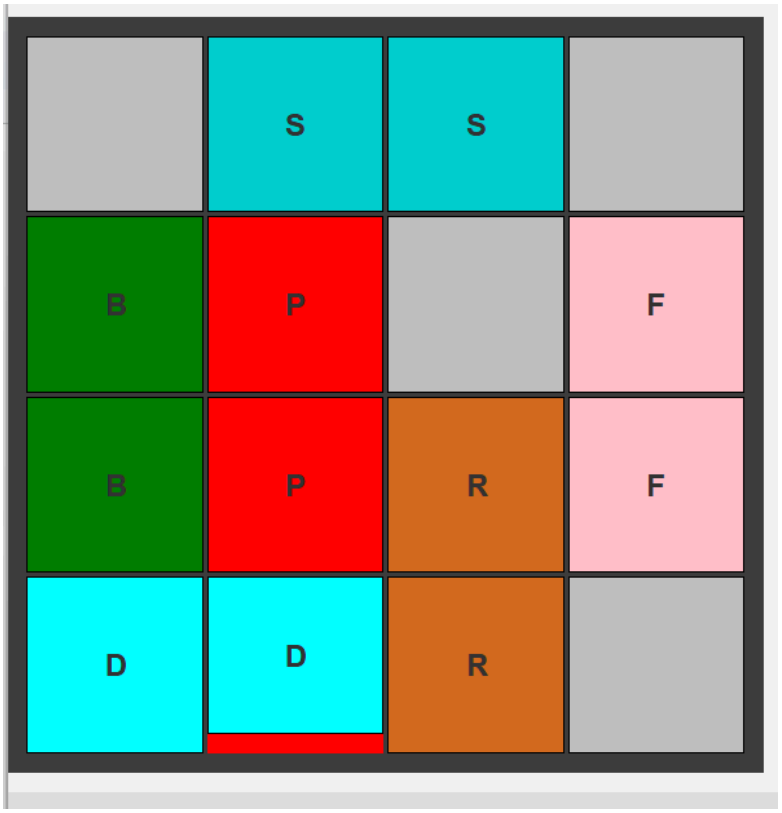
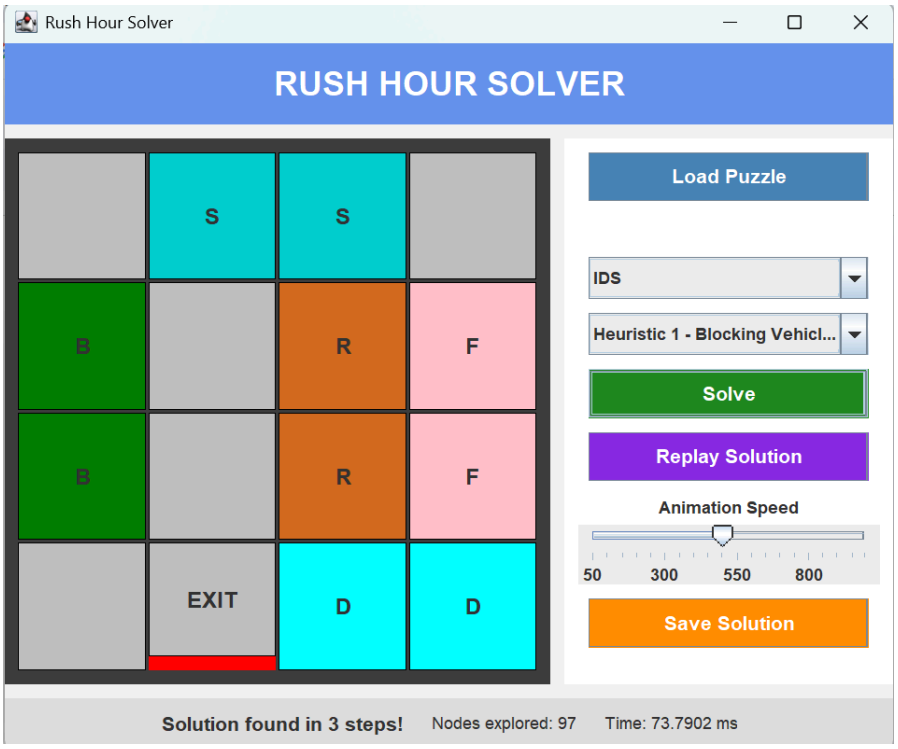


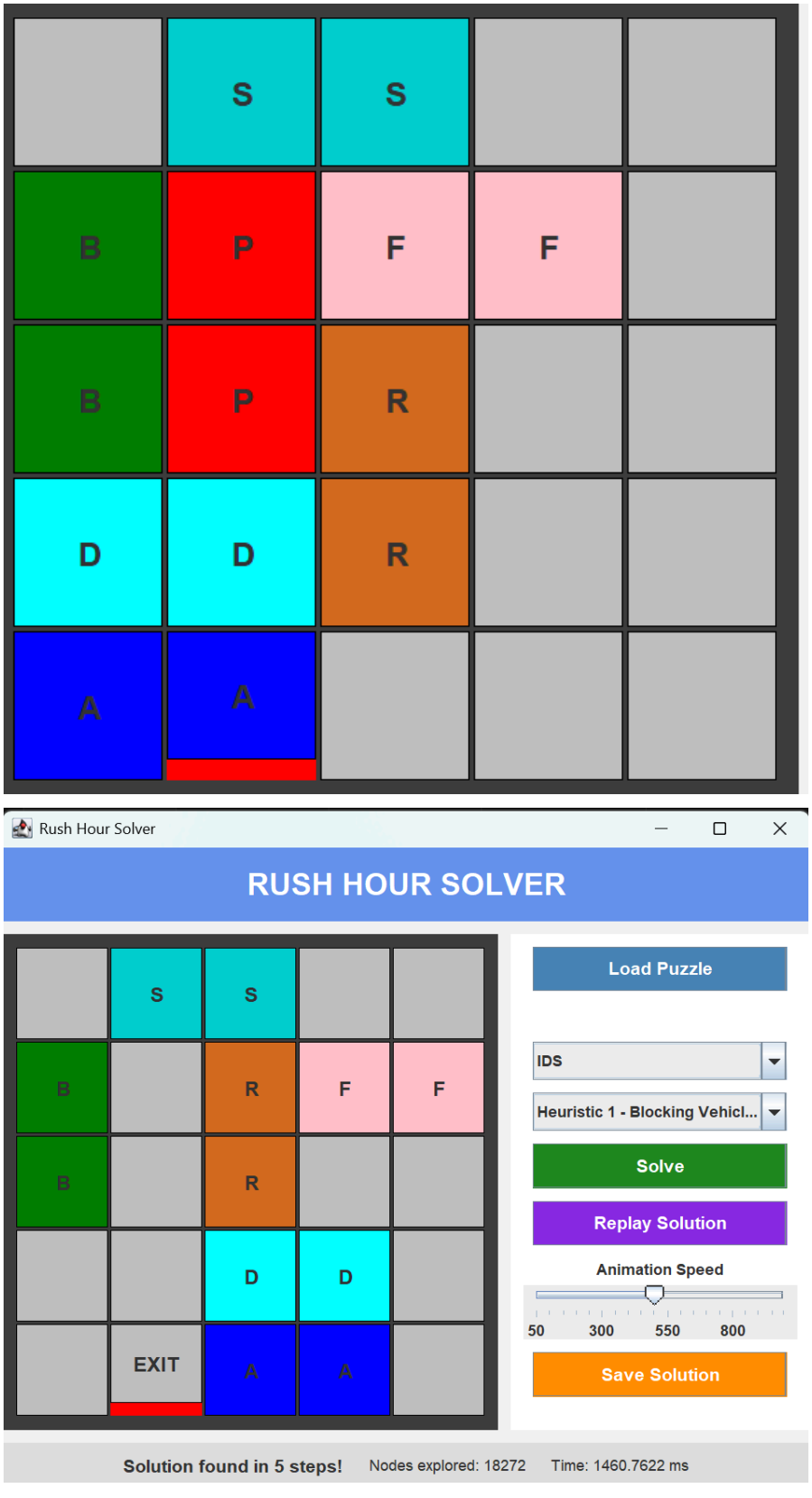
Input: tc3.txt

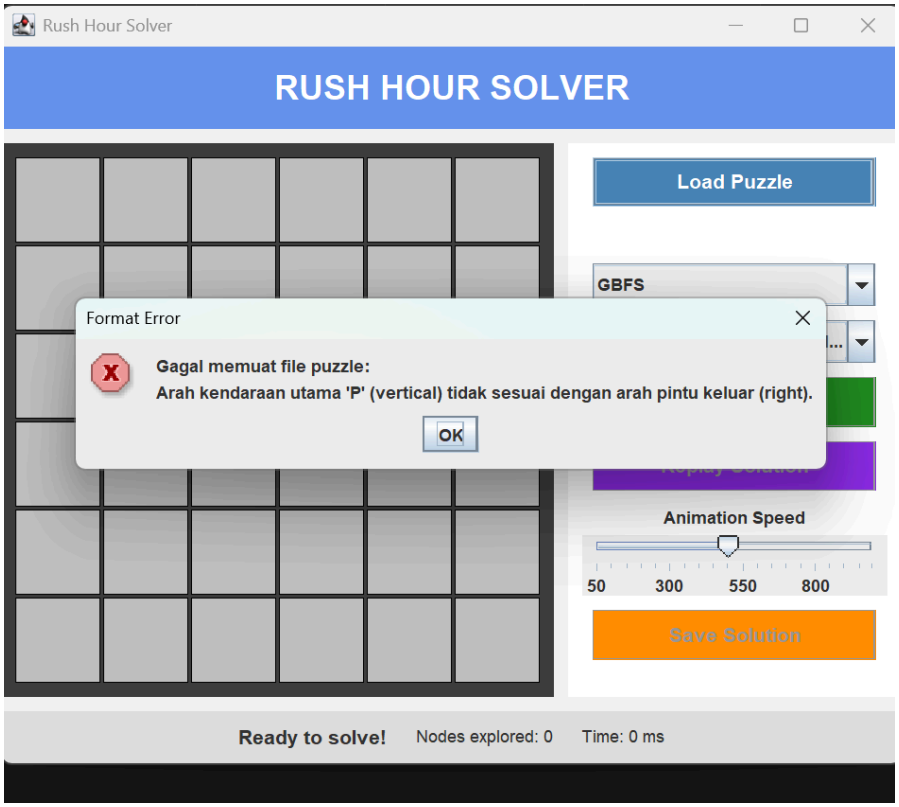
6 6
9
K
CQQQ..
CDDRRR
BBP...
AAP...
GGG.E.
LLL.E.

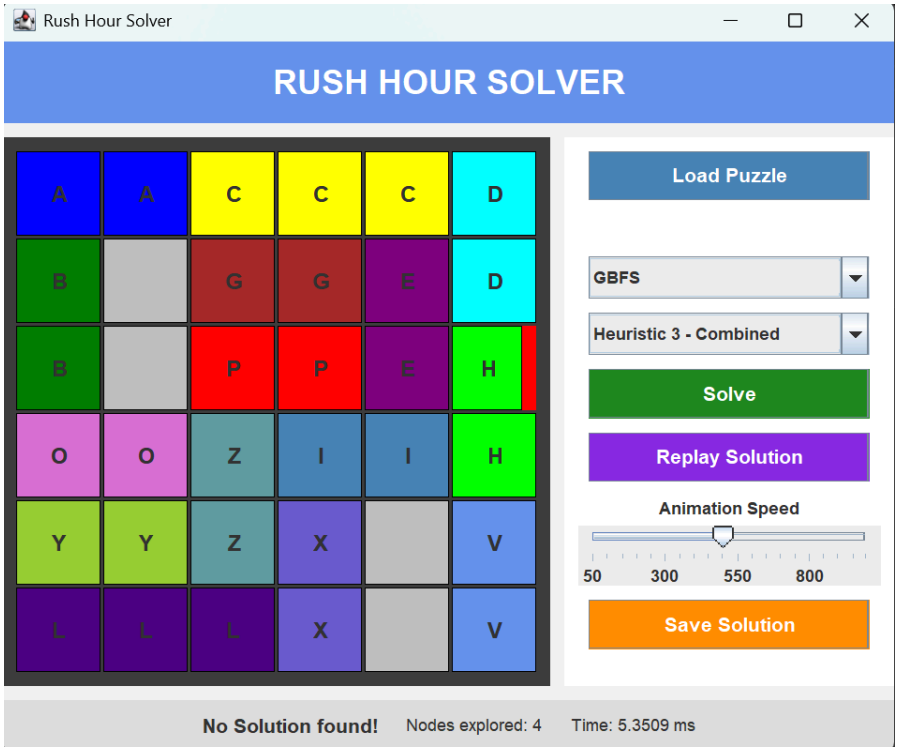
Algoritma: UCS	
Heuristic: -	
 	Input: tc3.txt 6 6 9 K CQQQ.. CDDRRR BBP... AAP... GGG.E. LLL.E.

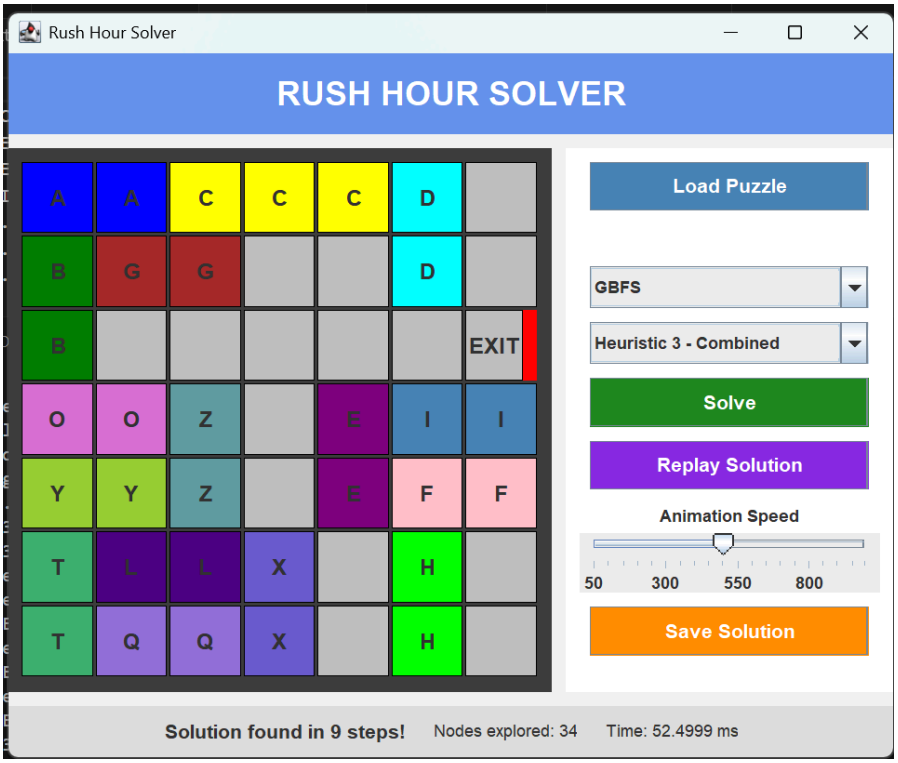
Algoritma: IDS	
Heuristic: -	
 	Input: test2.txt 4 4 5 K .SS. BP.F BPRF DDR.

Algoritma: IDS	
Heuristic: -	
 	Input: testIDX3.txt 4 4 5 .SS. BP.F BPRF DDR. K

Algoritma: IDS	
Heuristic: -	
	Input: tes3.txt 5 5 6 .SS.. BPFF. BPR.. DDR.. AA... K

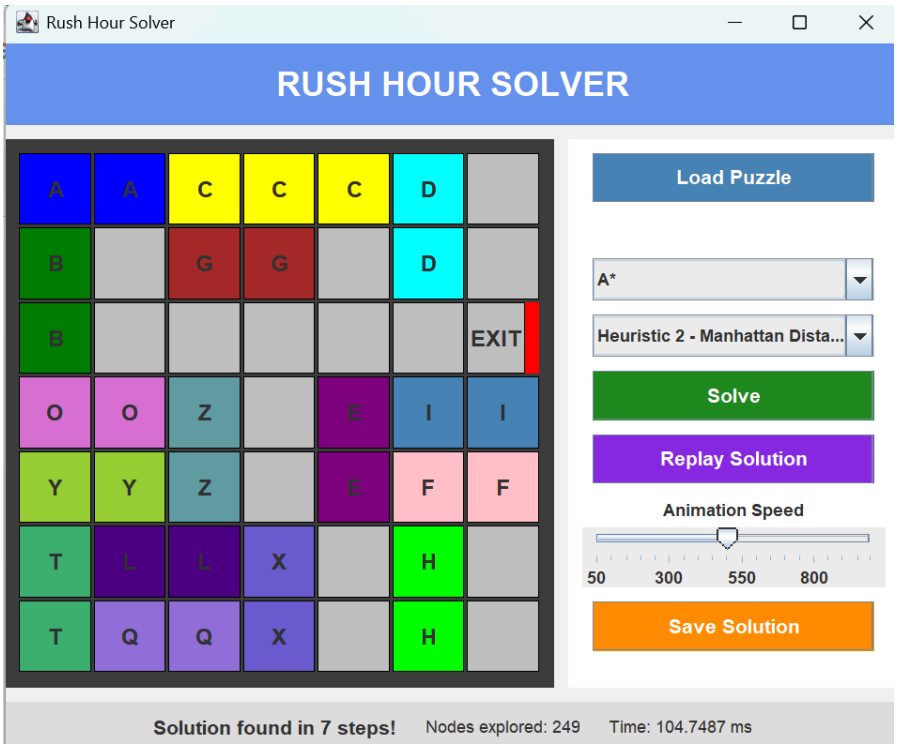
Algoritma: -	
Heuristic: -	
	Input: tcSlaah.txt 6 6 12 AABXXFK .PBCDF GP.CDF GH.III GHJ... LLJMM.

Algoritma: Bebas	
Heuristic: -	
	Input: nosolution.txt 6 6 14 AACCCD B.GGED B.PPEHK OOZIIH YYZX.V LLLX.V

Algoritma: GBFS	
Heuristic: Combine	
  	Input: tc77.txt 7 7 16 AACCCD. B.GGED. B.PPEH.K OOZIIH. YYZX.FF TLLX... TQQ....

Algoritma: A*

Heuristic: Manhattan Distance



Input: tc77.txt

7 7
16
AACCCD.
B.GGED.
B.PPEH.K
OOZIIH.
YYZX.FF
TLLX...
TQQ....

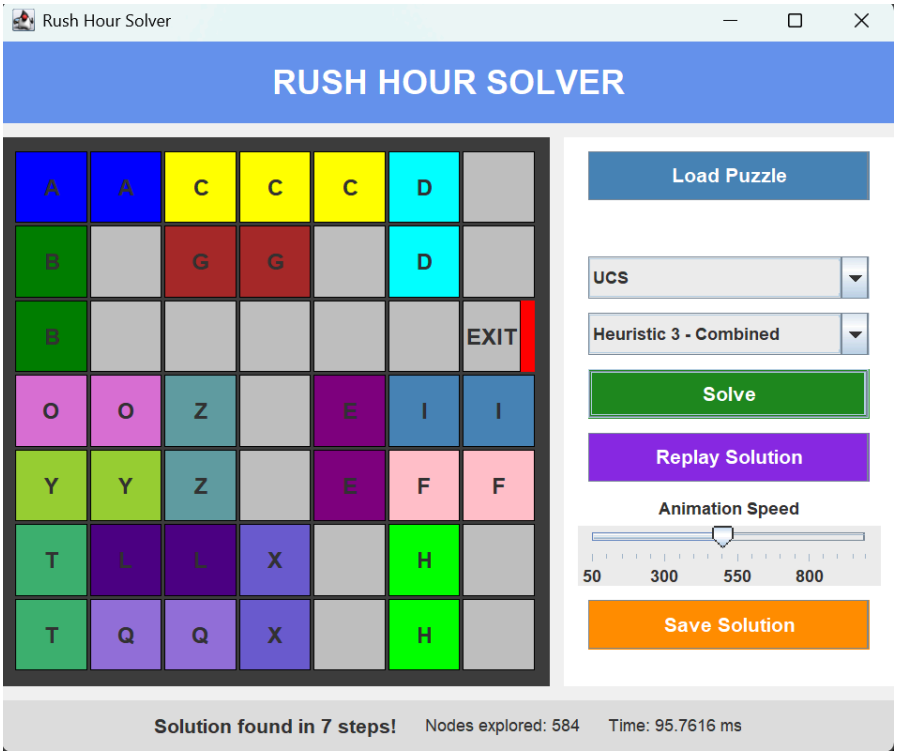
Contoh Output File:
Kondisi awal papan
AACCCD.
B.GGED.
B.PPEH.
OOZIIH.
YYZX.FF
TLLX...
TQQ....

Gerakan 1: X-down
AACCCD.
B.GGED.
B.PPEH.
OOZIIH.
YYZ..FF
TLLX...
TQQX…

…..

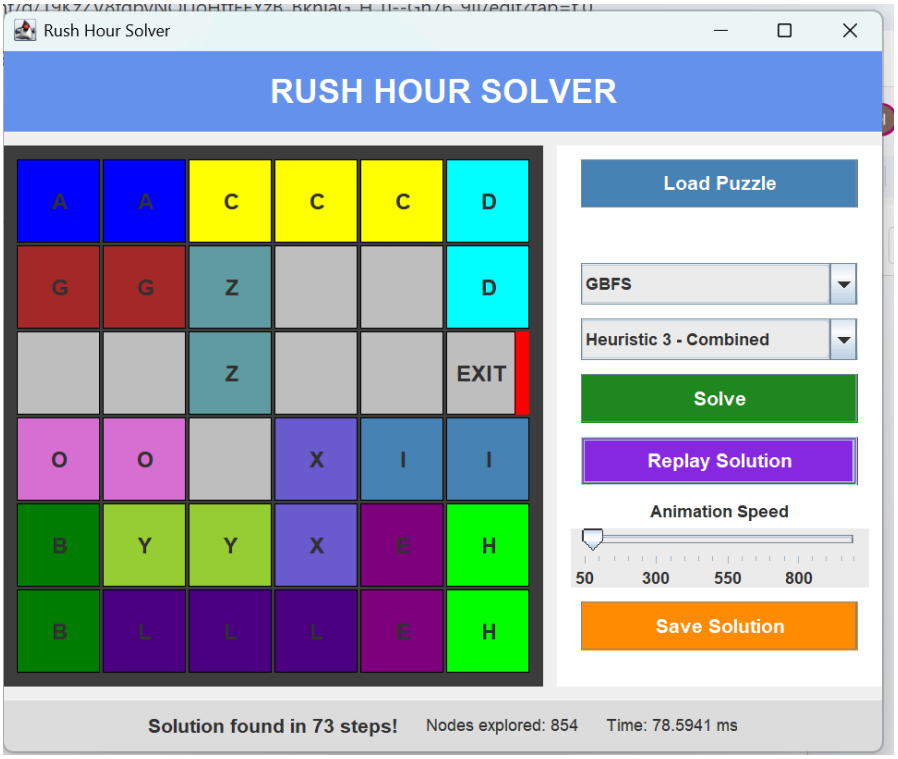
Gerakan 8: P-right
AACCCD.
B.GG.D.
B.....
OOZ.EII
YYZ.EFF
TLLX.H.
TQQX.H.

Algoritma: UCS
Heuristic: -

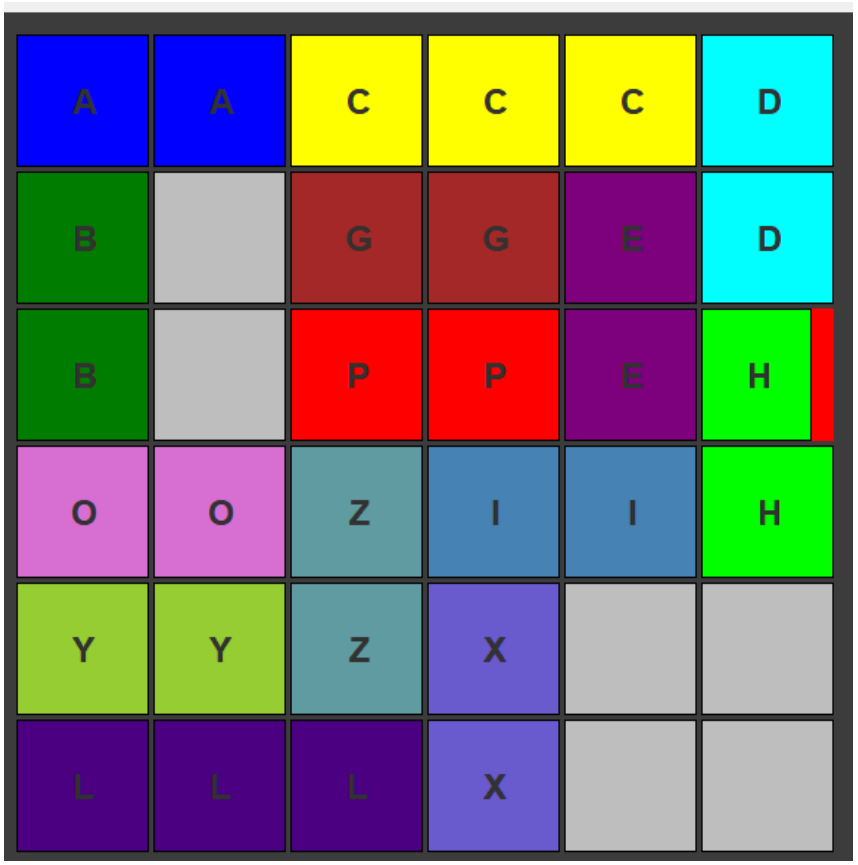


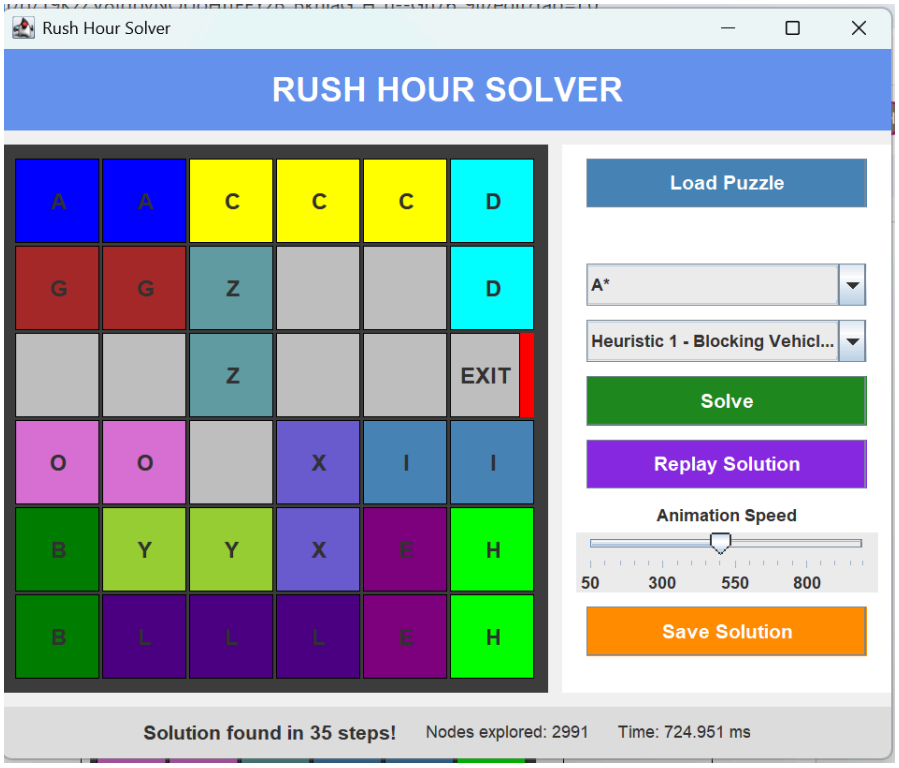
Input: tc77.txt

7 7
16
AACCCD.
B.GGED.
B.PPEH.K
OOZIIH.
YYZX.FF
TLLX...
TQQ....

Algoritma: GBFS	
Heuristic: Combine	
  	Input: hard.txt 6 6 13 AACCCD B.GGED B.PPEHK OOZIIH YYZX.. LLLX..

Algoritma: A*
Heuristic: Blocking Vehicle

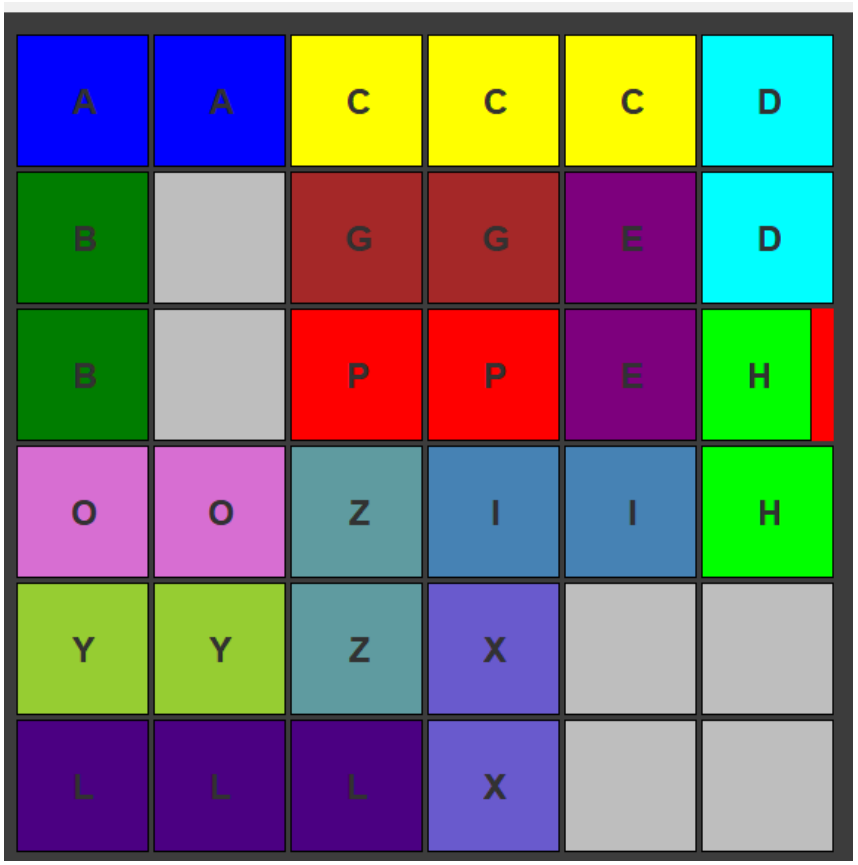


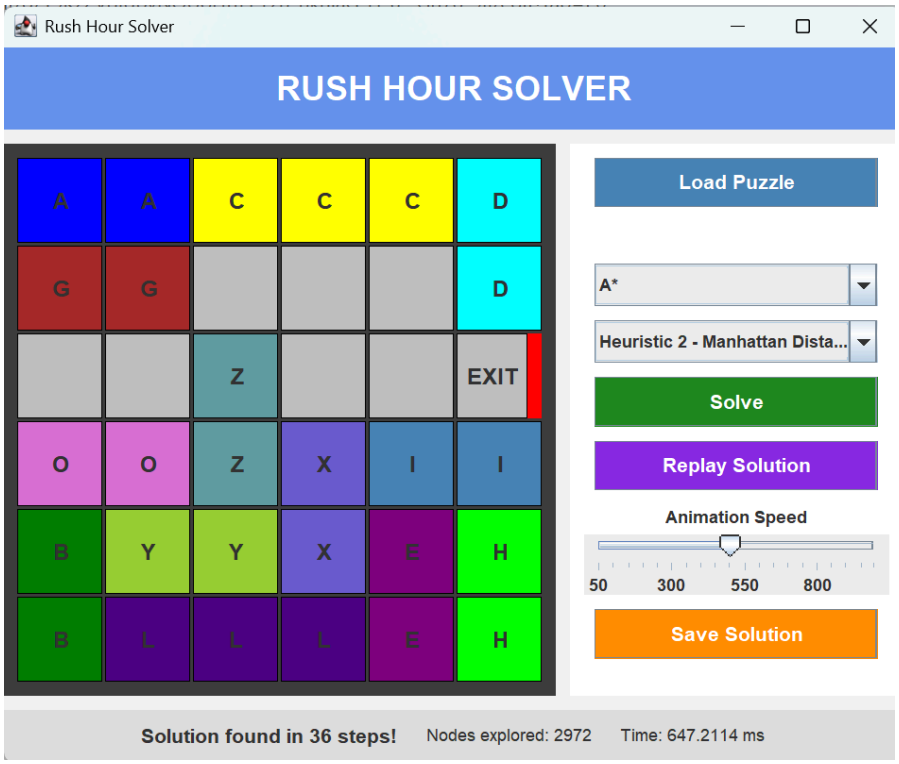


Input: hard.txt

6 6
13
AACCCD
B.GGED
B.PPEHK
OOZIIH
YYZX..
LLLX..

Algoritma: A*
Heuristic: Manhattan Distance

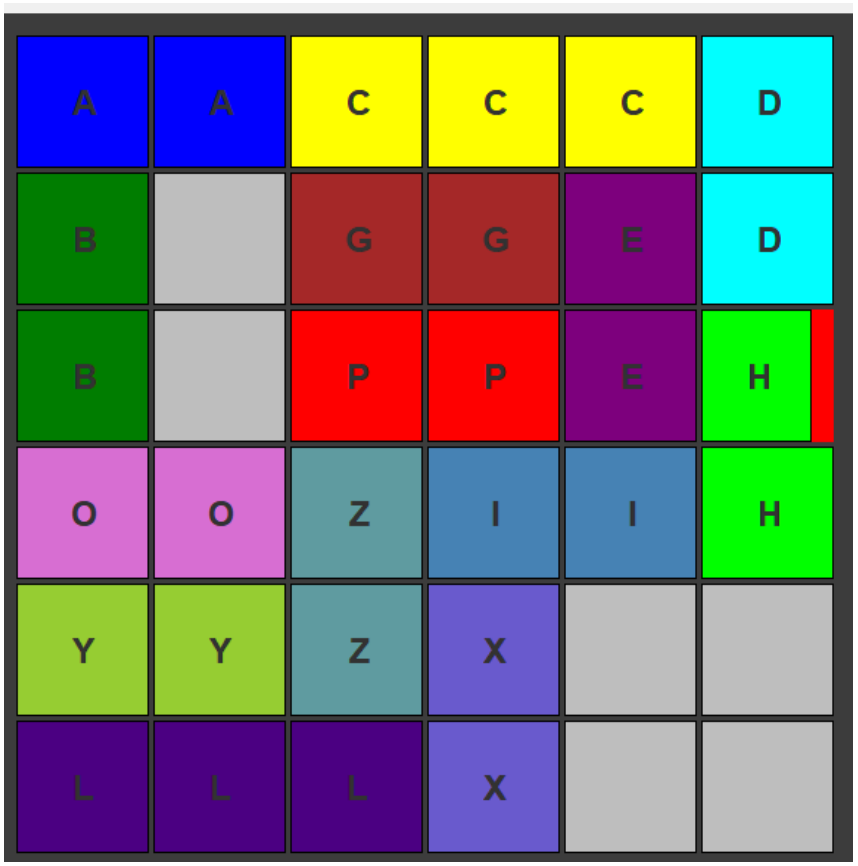


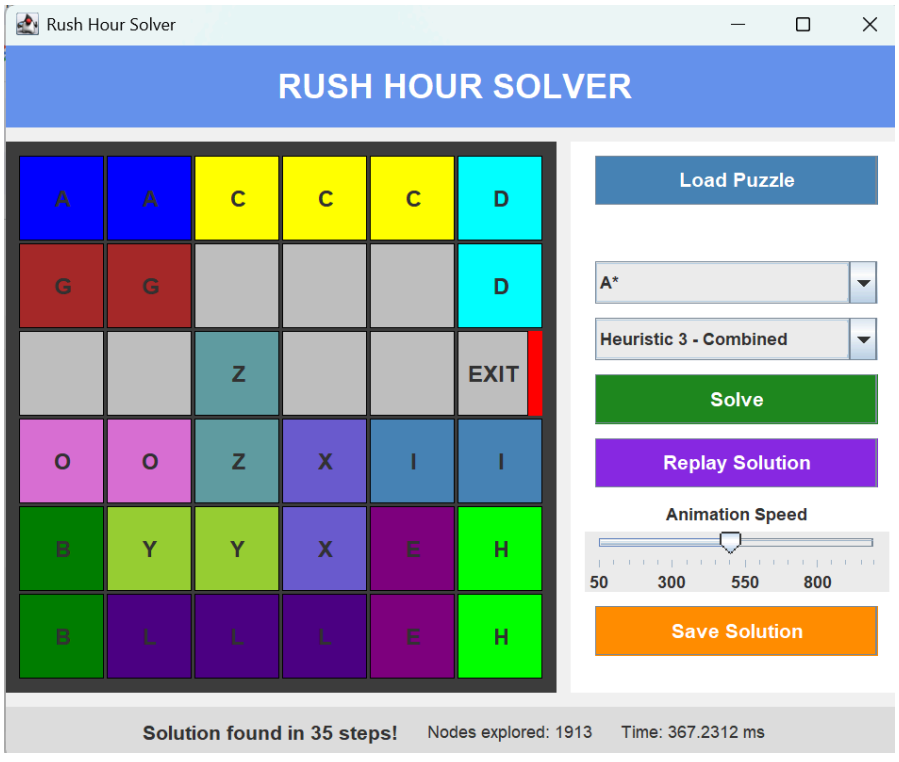


Input: hard.txt

6 6
13
AACCCD
B.GGED
B.PPEHK
OOZIIH
YYZX..
LLLX..

Heuristic: Combined

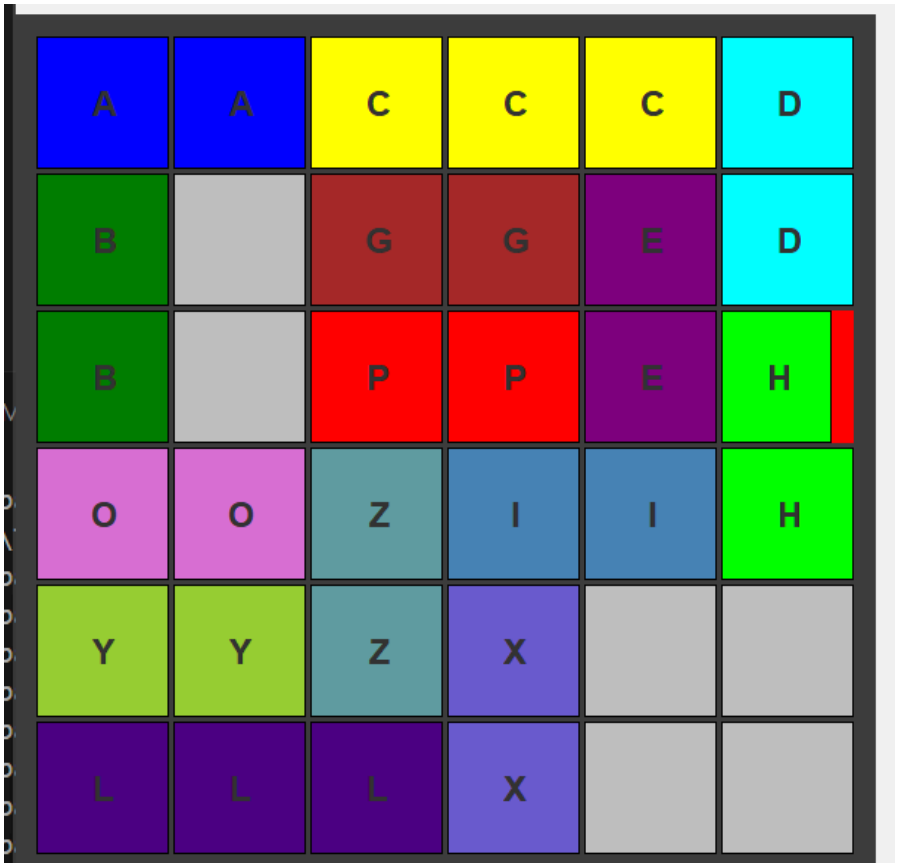


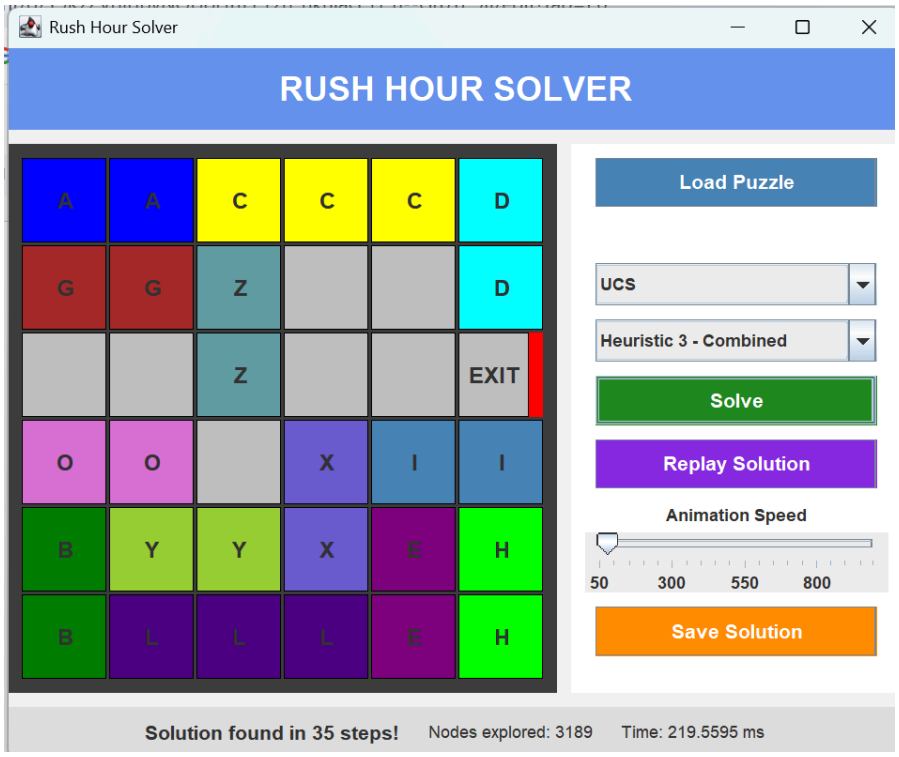


Input: hard.txt

6 6
13
AACCCD
B.GGED
B.PPEHK
OOZIIH
YYZX..
LLLX..

Algoritma: UCS
Heuristic: -





Input: hard.txt

6 6
13
AACCCD
B.GGED
B.PPEHK
OOZIIH
YYZX..
LLLX..

BAB IV
ANALISIS HASIL DAN ALGORITMA

A. Analisis Hasil

Berdasarkan hasil di atas, Greedy Best First Search (GBFS) memberikan hasil yang tidak optimal, tetapi cukup mengeksplorasi node dengan jumlah yang lebih sedikit dibandingkan Uniform Cost Search (UCS). Misalnya pada hasil testing tcspek.txt, GBFS menemukan hasil terbaik dengan heuristik BlockerOnly yaitu dengan 6 gerakan. Namun, dengan UCS, didapatkan dengan hanya 5 gerakan saja. Hal ini dikarenakan GBFS tidak memperhitungkan cost untuk mencapai state tertentu dan juga heuristik tidak dapat memastikan jalur yang dipilih adalah optimal, berbeda dengan UCS yang menjamin solusi optimal. Akan tetapi, UCS memerlukan 188 nodes untuk dieskplor dan waktu 62,9 ms, sedangkan GBFS hanya memerlukan sekitar 13 nodes dan waktu di bawah 8 ms. Artinya, GBFS lebih cepat dari UCS dalam menemukan suatu rute walaupun belum tentu yang terpendek.

Selain itu, dapat terlihat juga pada percobaan yang sama, bahwa pemilihan fungsi heuristik memengaruhi seberapa baik GBFS bekerja untuk menemukan lintasan terpendek. BlockerOnly menghasilkan rute yang lebih pendek dibandingkan ManhattanDistance karena pada ManhattanDistance mengabaikan kemungkinan jalurnya dihalangi oleh mobil lain, yang merupakan hal umum dalam permainan Rush Hour, sehingga BlockerOnly lebih cocok dan sesuai dengan permainan Rush Hour.

Pada penggunaan algoritma A*, terlihat bahwa hasilnya optimal selama admissible fungsi heuristiknya. Misalnya, pada testcase dengan heuristik ManhattanDistance pada input tc77.txt, A* menghasilkan jumlah gerakan yang sama dengan UCS, yaitu 7 gerakan. Di sisi lain, ketika menggunakan fungsi yang tidak dijamin admissible, A* terlihat tidak dijamin menghasilkan solusi optimal. Sebagai contoh, pada input tc3.txt, dengan menggunakan algoritma combine yang tidak admissible, didapatkan hasil 13 steps, sedangkan hasil optimalnya (dengan UCS) mendapatkan 12 steps.

A* juga lebih efisien dibandingkan UCS. Pada input tc77.txt, keduanya menghasilkan solusi optimal 7 gerakan, tetapi A* hanya perlu mengeksplor 249 nodes sedangkan UCS perlu mengeksplor 584 nodes. Hal ini membuktikan bahwa A* lebih efisien karena A* menggunakan *heuristic* untuk memperkirakan jarak ke tujuan, sehingga dapat memprioritaskan eksplorasi node yang lebih menjanjikan. Dengan demikian, A* mampu mengurangi jumlah node yang harus dieksplor tanpa mengorbankan optimalitas solusi.

Pada penggunaan algoritma IDS, algoritma ini mampu menghasilkan solusi yang optimal. Hal ini terlihat misalnya pada testcase dengan input tcspek.txt. UCS dan IDS memberikan jumlah langkah optimal (5). Akan tetapi, IDS kurang efisien karena perlu mengeksplor 4150 nodes dibandingkan dengan UCS yang hanya mengeksplor 188 nodes. Hal ini dikarenakan IDS berulang kali mengunjungi node yang sama sehingga kurang efisien, terutama jika jumlah langkah yang dibutuhkan besar.

B. Kompleksitas Algoritma

Pada algoritma UCS, UCS memiliki kompleksitas waktu $O\left(b^{1+\frac{C}{\varepsilon}}\right)$ dengan kompleksitas ruangnya juga sama yaitu $O\left(b^{1+\frac{C}{\varepsilon}}\right)$ dimana b adalah maksimum percabangan dari suatu simpul, C adalah cost dari solusi optimal, dan setiap gerakan costnya setidaknya ε . Pada permainan Rush Hour, karena setiap gerakan bernilai sama, UCS dapat dipandang sebagai BFS sehingga kompleksitas waktu dan kompleksitas ruangnya dapat disederhanakan menjadi $O\left(b^{d}\right)$ dengan d adalah kedalaman dari solusi terbaik.

Pada algoritma Greedy Best First Search, worst-case kompleksitas waktunya sama seperti DFS yaitu $O\left(b^{m}\right)$ dengan m adalah kedalaman maksimum dari ruang status. Pada umumnya, dengan heuristik yang baik, GBFS dapat memiliki kompleksitas waktu dan ruang yang lebih baik, yaitu $O\left(b^{D}\right)$ dengan D adalah kedalaman dari solusi.

Pada algoritma A*, worst-case kompleksitas waktu dan ruangnya adalah $O\left(b^{d}\right)$. Kompleksitasnya bergantung pada seberapa baik heuristik yang digunakan. Pada best-case, kompleksitas waktu dan ruangnya dapat menjadi $O(d)$.

Pada algoritma IDS, kompleksitas waktunya adalah $O\left(b^{d}\right)$ dan kompleksitas ruangnya adalah $O(bd)$.

C. Implementasi Bonus GUI

Implementasi antarmuka grafis (GUI) menggunakan pustaka bawaan Java, yaitu Swing. Desain UI dibagi menjadi beberapa bagian utama yaitu header, footer, dan body. Bagian *body* terdiri atas dua komponen penting, yaitu board panel dan control panel. Board Panel berfungsi untuk menampilkan kondisi terkini dari papan permainan (board) serta menunjukkan langkah-langkah penyelesaian puzzle. Papan ditampilkan dalam bentuk grid yang terdiri dari banyak sel. Sel yang berisi potongan (piece) akan diberi warna, sedangkan sel kosong akan berwarna abu-abu. Pendekatan ini digunakan untuk memudahkan pemain dalam mengidentifikasi potongan yang sedang bergerak. Sementara control panel berfungsi untuk memberikan kontrol kepada pengguna untuk mengatur kecepatan animasi, melakukan penyimpanan dan pemuatan permainan (*save and load*), serta menyelesaikan puzzle yang dimuat dari file .txt.

LINK REPOSITORY

https://github.com/henry204xx/Tucil3_13523051_13523108

CHECKLIST

No	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 serta menyimpan solusi 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	✓	