Tugas Kecil 3 IF 2211

Strategi Algoritma: Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding

Dibuat Oleh:
William Gerald Briandelo - 13222061

Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
Bandung
2025

Pseudocode:

```
Procedure ParseBoard(filename, exit)
Kamus Lokal:
    br              : BufferedReader
    dims            : array of string
    A, B            : integer
    line            : string
    kpos            : integer
    board           : char[][]

Algoritma:
    br ← new BufferedReader(new FileReader(filename))
    dims ← split(br.readLine().trim(), whitespace)
    A ← toInteger(dims[0])
    B ← toInteger(dims[1])
    br.readLine()
    board ← new char[A][B]
    for i dari 0 hingga A – 1 do
        line ← br.readLine()
        if length(line) = B then
            board[i] ← toCharArray(line)
        else if length(line) = B + 1 and contains(line, 'K') then
            kpos ← indexOf(line, 'K')
            exit.r ← i
            if kpos = 0 then
                exit.c ← -1
                for j dari 0 hingga B – 1 do
                    board[i][j] ← charAt(line, j + 1)
            else
                exit.c ← B
                for j dari 0 hingga B – 1 do
                    board[i][j] ← charAt(line, j)
            endif
        else
            error("Invalid row length")
        endif
    endfor
    detectOrientation(board)
    return board


Procedure detectOrientation(board)
Kamus Lokal:
    R, C            : integer
    map             : map<char, list of (int, int)>
    coords          : list of (int, int)
    sameRow, sameCol : boolean

Algoritma:
    R ← rows(board)
    C ← cols(board)
    map ← empty map
    for i dari 0 hingga R – 1 do
```

```
        for j dari 0 hingga C – 1 do
            if board[i][j] ≠ '.' and board[i][j] ≠ 'K' then
                append map[board[i][j]] dengan (i, j)
            endif
        endfor
    endfor
    for tiap entry (p, coords) dalam map do
        sameRow ← all coords have same row as coords[0]
        sameCol ← all coords have same col as coords[0]
        if not (sameRow or sameCol) then error("Piece not linear") endif
        orientation[p] ← sameRow
    endfor


Procedure MovePiece(state, p, dir, exit)
Kamus Lokal:
    di, dj    : integer
    b         : char[][]
    coords    : list of (int, int)
    br, bc    : integer
    nr, nc    : integer
    toExit    : boolean

Algoritma:
    (di, dj) ← directionDelta(dir)
    b ← deepCopy(state.board)
    coords ← positions of p in state.board
    br ← extremeRow(coords, di)
    bc ← extremeCol(coords, dj)
    nr ← br + di
    nc ← bc + dj
    toExit ← (nr = exit.r and nc = exit.c)
    if outOfBounds(nr, nc, state) and not (p = 'P' and toExit) then return null endif
    if inBounds(nr, nc, state) and state.board[nr][nc] ≠ '.' then return null endif
    for tiap (r, c) dalam coords do b[r][c] ← '.' endfor
    for tiap (r, c) dalam coords do
        if inBounds(r + di, c + dj, state) then
            b[r + di][c + dj] ← p
        endif
    endfor
    return new State(b)


Procedure Expand(node, exit, hfunc)
Kamus Lokal:
    neigh  : list of Node
    s      : State
    p      : char
    d      : Direction
    nxt    : State

Algoritma:
    neigh ← empty list
    s ← node.state
    for i dari 0 hingga s.rows – 1 do
```

```
      for j dari 0 hingga s.cols – 1 do
         p ← s.board[i][j]
         if p = '.' or p = 'K' then continue endif
         for tiap d dalam {UP, DOWN, LEFT, RIGHT} do
            if invalidDirection(p, d) then continue endif
            nxt ← MovePiece(s, p, d, exit)
            if nxt ≠ null then
               cost ← node.cost + 1
               h ← (hfunc ≠ null) ? hfunc.eval(nxt, exit) : 0
               append neigh dengan new Node(nxt, node, new Move(p, d), cost, h)
            endif
         endfor
      endfor
   endfor
   return neigh


Procedure Search(start, exit, hfunc, algo)
Kamus Lokal:
   frontier : priority queue of Node
   seen     : set of State
   comp     : comparator

Algoritma:
   comp ← comparatorBy(algo)
```

Uniform Cost Search (UCS) adalah algoritma pencarian yang termasuk ke dalam kategori uninformed search, artinya algoritma ini tidak memiliki pengetahuan tambahan tentang posisi tujuan. UCS menggunakan strategi ekspansi berdasarkan biaya total dari simpul awal ke simpul saat ini, yang dilambangkan dengan g(n). Algoritma ini mengembangkan simpul dengan g(n) terkecil terlebih dahulu, sehingga menjamin solusi optimal apabila semua biaya langkah sama atau positif. UCS cocok digunakan jika informasi tentang jarak atau estimasi ke tujuan tidak tersedia.

Greedy Best First Search (Greedy BFS) adalah algoritma informed search yang menggunakan fungsi heuristik h(n) untuk memperkirakan jarak dari suatu simpul ke tujuan. Greedy hanya mempertimbangkan nilai h(n) dan selalu mengembangkan simpul yang memiliki nilai h(n) terkecil. Meskipun sering kali lebih cepat karena fokus ke arah tujuan, algoritma ini tidak mempertimbangkan biaya dari simpul awal ke simpul saat ini, sehingga tidak menjamin solusi optimal.

A* adalah gabungan dari UCS dan Greedy BFS. Algoritma ini mengembangkan simpul berdasarkan nilai f(n) = g(n) + h(n), yaitu jumlah dari biaya sebenarnya dari awal (g(n)) dan estimasi biaya ke tujuan (h(n)). A* berusaha menyeimbangkan antara eksplorasi dan eksploitasi: ia mengejar tujuan dengan panduan heuristik namun tetap mempertimbangkan jalur termurah sejauh ini. Jika fungsi h(n) yang digunakan adalah admissible (tidak melebih-lebihkan jarak ke tujuan) dan consistent, maka A* menjamin solusi optimal.

g(n) adalah biaya sebenarnya dari simpul awal ke simpul n. Ini mencerminkan jarak aktual atau jumlah langkah yang dibutuhkan untuk mencapai simpul n dari posisi awal. Sedangkan, f(n) adalah total estimasi biaya dari simpul awal ke tujuan melewati simpul n. Dalam A*, f(n) dihitung sebagai f(n) = g(n) + h(n), sedangkan dalam UCS hanya f(n) = g(n), dan dalam Greedy hanya f(n) = h(n).

Heuristik Manhattan dan Euclidean yang digunakan admissible. Heuristik admissible tidak boleh melebihi biaya sesungguhnya menuju tujuan. Dalam kasus Rush Hour, heuristik Manhattan dan Euclidean mengukur jarak dari mobil 'P' ke posisi keluar tanpa mempertimbangkan rintangan di jalan. Karena itu, estimasinya selalu kurang dari atau sama dengan biaya aktual untuk mencapai tujuan, sehingga memenuhi kriteria admissibility.

UCS berperilaku sama seperti BFS dalam Rush Hour karena setiap langkah memiliki biaya yang sama, seperti pada permainan ini di mana tiap pergerakan dihitung sebagai satu langkah. Karena BFS juga mengembangkan simpul berdasarkan kedalaman atau jumlah langkah dari awal, maka urutan pengembangan simpul dan jalur solusi yang ditemukan UCS dan BFS akan sama dalam kondisi biaya seragam.

Secara teoritis, A* lebih efisien dibandingkan UCS dalam banyak kasus karena menggunakan heuristik untuk membatasi ruang pencarian. UCS mengembangkan banyak simpul yang mungkin tidak relevan, karena tidak tahu arah ke tujuan. Sementara A* menggunakan h(n) untuk memperkirakan arah terbaik dan memfokuskan pencarian. Jadi, jika heuristik yang digunakan akurat, A* akan lebih cepat menemukan solusi dibandingkan UCS, terutama pada masalah yang kompleks seperti Rush Hour.

Greedy Best First Search tidak menjamin solusi optimal. Karena hanya mempertimbangkan nilai heuristik h(n) tanpa memperhatikan biaya dari awal (g(n)), algoritma ini bisa saja memilih jalur yang terlihat menjanjikan tapi ternyata lebih panjang secara keseluruhan. Dalam Rush Hour, bisa terjadi Greedy memilih memindahkan mobil ke arah tujuan lebih cepat tapi harus berputar banyak kali, dibandingkan jalur yang lebih langsung namun dengan nilai h(n) lebih tinggi.

Source Code:
RushHour.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.*;

public class RushHour {
    enum Direction { UP, DOWN, LEFT, RIGHT }

    static class Move {
```

```
        char piece;
        Direction dir;
        Move(char piece, Direction dir) {
            this.piece = piece;
            this.dir = dir;
        }
        @Override
        public String toString() {
            return piece + "-" + dir;
        }
    }

    static class State {
        char[][] board;
        int rows, cols;
        State(char[][] b) {
            rows = b.length;
            cols = b[0].length;
            board = new char[rows][cols];
            for (int i = 0; i < rows; i++)
                System.arraycopy(b[i], 0, board[i], 0, cols);
        }
        @Override
        public boolean equals(Object o) {
            return (o instanceof State) && Arrays.deepEquals(board,
((State)o).board);
        }
        @Override
        public int hashCode() {
            return Arrays.deepHashCode(board);
        }
    }

    static class Node {
        State state;
        Node parent;
        Move move;
        int cost;
        double h;
```

```
        Node(State s, Node p, Move m, int cost, double h) {
            this.state = s;
            this.parent = p;
            this.move = m;
            this.cost = cost;
            this.h = h;
        }
        double f() { return cost + h; }
    }

    interface Heuristic { double eval(State s, Exit exit); }

    static class Manhattan implements Heuristic {
        public double eval(State s, Exit exit) {
            for (int i = 0; i < s.rows; i++)
                for (int j = 0; j < s.cols; j++)
                    if (s.board[i][j] == 'P')
                        return Math.abs(i - exit.r) + Math.abs(j -
exit.c);
            return Double.MAX_VALUE;
        }
    }

    static class Euclidean implements Heuristic {
        public double eval(State s, Exit exit) {
            for (int i = 0; i < s.rows; i++)
                for (int j = 0; j < s.cols; j++)
                    if (s.board[i][j] == 'P')
                        return Math.hypot(i - exit.r, j - exit.c);
            return Double.MAX_VALUE;
        }
    }

    static class Exit { int r, c; }

    static Map<Character, Boolean> orientation = new HashMap<>();

    static char[][] parseBoard(String filename, Exit exit) throws
IOException {
```

```
        try (BufferedReader br = new BufferedReader(new
FileReader(filename))) {
            String[] dims = br.readLine().trim().split("\\s+");
            int A = Integer.parseInt(dims[0]), B =
Integer.parseInt(dims[1]);
            br.readLine();
            char[][] board = new char[A][B];
            for (int i = 0; i < A; i++) {
                String line = br.readLine();
                if (line.length() == B) {
                    board[i] = line.toCharArray();
                } else if (line.length() == B + 1 && line.indexOf('K')
!= -1) {
                    int kpos = line.indexOf('K');
                    exit.r = i;
                    if (kpos == 0) {
                        exit.c = -1;
                        for (int j = 0; j < B; j++) board[i][j] =
line.charAt(j + 1);
                    } else if (kpos == B) {
                        exit.c = B;
                        for (int j = 0; j < B; j++) board[i][j] =
line.charAt(j);
                    } else {
                        throw new IllegalArgumentException("Invalid K
position at row " + i);
                    }
                } else {
                    throw new IllegalArgumentException("Row " + i + "
length invalid");
                }
            }
            detectOrientation(board);
            return board;
        }
    }

    static void detectOrientation(char[][] board) {
        int R = board.length, C = board[0].length;
```

```
        Map<Character, List<int[]>> map = new HashMap<>();
        for (int i = 0; i < R; i++)
            for (int j = 0; j < C; j++) {
                char ch = board[i][j];
                if (ch == '.' || ch == 'K') continue;
                map.computeIfAbsent(ch, k -> new ArrayList<>()).add(new
int[]{i, j});
            }
        for (Map.Entry<Character, List<int[]>> e : map.entrySet()) {
            char ch = e.getKey();
            List<int[]> coords = e.getValue();
            boolean sameRow = coords.stream().allMatch(rc -> rc[0] ==
coords.get(0)[0]);
            boolean sameCol = coords.stream().allMatch(rc -> rc[1] ==
coords.get(0)[1]);
            if (!(sameRow || sameCol))
                throw new IllegalArgumentException("Piece '" + ch + "'
not linear");
            orientation.put(ch, sameRow);
        }
    }

    static boolean isGoal(State s, Exit exit) {
        if (exit.c == -1 || exit.c == s.cols) {
            int j = exit.c == -1 ? 0 : s.cols - 1;
            return s.board[exit.r][j] == 'P';
        } else {
            return s.board[exit.r][exit.c] == 'P';
        }
    }

    static List<Node> expand(Node node, Exit exit, Heuristic hfunc) {
        List<Node> neigh = new ArrayList<>();
        State s = node.state;
        for (int i = 0; i < s.rows; i++) {
            for (int j = 0; j < s.cols; j++) {
                char p = s.board[i][j];
                if (p == '.' || p == 'K') continue;
                for (Direction d : Direction.values()) {
```

```
                    boolean horiz = orientation.get(p);
                    if ((horiz && (d == Direction.UP || d ==
Direction.DOWN)) ||
                        (!horiz && (d == Direction.LEFT || d ==
Direction.RIGHT)))
                        continue;
                    State nxt = movePiece(s, exit, p, d);
                    if (nxt != null) {
                        double h = (hfunc != null) ? hfunc.eval(nxt,
exit) : 0;
                        neigh.add(new Node(nxt, node, new Move(p, d),
node.cost + 1, h));
                    }
                }
            }
        }
        return neigh;
    }

    static State movePiece(State s, Exit exit, char p, Direction dir) {
        int di = dir == Direction.DOWN ? 1 : dir == Direction.UP ? -1 :
0;
        int dj = dir == Direction.RIGHT ? 1 : dir == Direction.LEFT ? -1
: 0;
        char[][] b = new char[s.rows][s.cols];
        for (int r = 0; r < s.rows; r++) System.arraycopy(s.board[r], 0,
b[r], 0, s.cols);

        List<int[]> coords = new ArrayList<>();
        for (int r = 0; r < s.rows; r++)
            for (int c = 0; c < s.cols; c++)
                if (s.board[r][c] == p) coords.add(new int[]{r, c});

        int br = coords.stream().mapToInt(rc -> rc[0])
                .reduce((a, b1) -> di > 0 ? Math.max(a, b1) : di < 0 ?
Math.min(a, b1) : a)
                .getAsInt();
        int bc = coords.stream().mapToInt(rc -> rc[1])
                .reduce((a, b1) -> dj > 0 ? Math.max(a, b1) : dj < 0 ?
```

```
Math.min(a, b1) : a)
                .getAsInt();

        int nr = br + di, nc = bc + dj;
        boolean toExit = (nr == exit.r && nc == exit.c);
        if (nr < 0 || nr >= s.rows || nc < 0 || nc >= s.cols) {
            if (!(p == 'P' && toExit)) return null;
        } else if (s.board[nr][nc] != '.') {
            return null;
        }

        for (int[] rc : coords) b[rc[0]][rc[1]] = '.';
        for (int[] rc : coords) {
            int r2 = rc[0] + di, c2 = rc[1] + dj;
            if (r2 >= 0 && r2 < s.rows && c2 >= 0 && c2 < s.cols) {
                b[r2][c2] = p;
            }
        }
        return new State(b);
    }

    static Node search(State start, Exit exit, Heuristic hfunc, String
algo) {
        Comparator<Node> comp;
        switch (algo) {
            case "UCS":
                comp = Comparator.comparingInt(n -> n.cost);
                break;
            case "Greedy":
                comp = Comparator.comparingDouble(n -> n.h);
                break;
            default:
                comp = Comparator.comparingDouble(Node::f);
                break;
        }
        PriorityQueue<Node> frontier = new PriorityQueue<>(comp);
        frontier.add(new Node(start, null, null, 0, (hfunc != null ?
hfunc.eval(start, exit) : 0)));
        Set<State> seen = new HashSet<>();
```

```
        while (!frontier.isEmpty()) {
            Node cur = frontier.poll();
            if (seen.contains(cur.state)) continue;
            seen.add(cur.state);
            if (isGoal(cur.state, exit)) return cur;
            frontier.addAll(expand(cur, exit, hfunc));
        }
        return null;
    }

    static Node backtrack(Node node, Exit exit, Set<State> visited) {
        if (isGoal(node.state, exit)) {
            return node;
        }
        visited.add(node.state);
        Node best = null;

        for (Node child : expand(node, exit, null)) {
            if (visited.contains(child.state)) continue;
            Node sol = backtrack(child, exit, visited);
            if (sol != null) {
                if (best == null || sol.cost < best.cost) {
                    best = sol;
                }
            }
        }

        visited.remove(node.state);
        return best;
    }

    static void printSolution(Node goal, PrintWriter pw) {
        List<Node> path = new ArrayList<>();
        for (Node n = goal; n != null; n = n.parent) path.add(n);
        Collections.reverse(path);

        pw.println("Papan Awal:");
        System.out.println("Papan Awal:");
```

```
        printBoard(path.get(0).state, pw);
        printBoard(path.get(0).state);
        for (int i = 1; i < path.size(); i++) {
            String header = "\nGerakan " + i + ": " + path.get(i).move;
            pw.println(header);
            System.out.println(header);
            printBoard(path.get(i).state, pw);
            printBoard(path.get(i).state);
        }
    }

    static void printBoard(State s, PrintWriter pw) {
        for (char[] row : s.board) {
            pw.println(new String(row));
        }
    }

    static void printBoard(State s) {
        for (char[] row : s.board) {
            System.out.println(new String(row));
        }
    }

    public static void main(String[] args) throws IOException {
        Scanner in = new Scanner(System.in);
        System.out.print("Masukkan nama file input: ");
        String fn = in.nextLine();

        Exit exit = new Exit();
        char[][] board = parseBoard(fn, exit);
        State start = new State(board);

        System.out.println("Pilih algoritma:");
        System.out.println(" 1. Greedy Best-First Search");
        System.out.println(" 2. Uniform-Cost Search (UCS)");
        System.out.println(" 3. A* Search");
        System.out.println(" 4. Backtracking (recursive)");
        int ch = in.nextInt();
```

```
        Heuristic h = null;
        String algo;
        if (ch == 1 || ch == 3) {
            System.out.print("Pilih heuristik: 1. Manhattan  2.
Euclidean: ");
            h = (in.nextInt() == 1) ? new Manhattan() : new Euclidean();
        }
        algo = (ch == 1 ? "Greedy" : ch == 2 ? "UCS" : ch == 3 ? "A*" :
"Backtracking");

        long startTime = System.nanoTime();
        Node sol;
        if (ch == 4) {
            Set<State> visited = new HashSet<>();
            sol = backtrack(new Node(start, null, null, 0, 0), exit,
visited);
        } else {
            sol = search(start, exit, h, algo);
        }
        long endTime = System.nanoTime();

        String outFile = "solusi_" + fn;
        try (PrintWriter pw = new PrintWriter(outFile)) {
            if (sol != null) {
                System.out.println("\nSolusi Ditemukan! (" + algo +
")");
                pw.println("Solusi Ditemukan! (" + algo + ")");
                printSolution(sol, pw);
            } else {
                System.out.println("Tidak ada solusi.");
                pw.println("Tidak ada solusi.");
            }
            double elapsed = (endTime - startTime) / 1e6;
            String timeMsg = String.format("Waktu eksekusi: %.2f ms",
elapsed);
            System.out.println(timeMsg);
            pw.println(timeMsg);
        }
```

```
        in.close();
    }
}
```

GUI.java:

```
import javax.swing.*;
import javax.swing.filechooser.FileNameExtensionFilter;
import java.awt.*;
import java.awt.event.*;
import javax.swing.Timer;
import java.io.*;
import java.util.*;

public class GUI extends JFrame {
    private JPanel boardPanel;
    private JButton loadButton, solveButton;
    private JComboBox<String> algoBox, heuristicBox;
    private File currentFile;
    private java.util.List<char[][]> solutionStates;
    private Timer animationTimer;
    private int animIndex = 0;

    public GUI() {
        super("Rush Hour Solver");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(600, 700);
        setLayout(new BorderLayout());

        JPanel controlPanel = new JPanel();
        loadButton = new JButton("Load Board");
        solveButton = new JButton("Solve");
        solveButton.setEnabled(false);
        algoBox = new JComboBox<>(new
String[]{"Greedy","UCS","A*","Backtracking"});
        heuristicBox = new JComboBox<>(new
String[]{"Manhattan","Euclidean"});
        heuristicBox.setEnabled(false);
```

```
        controlPanel.add(loadButton);
        controlPanel.add(new JLabel("Algorithm:"));
        controlPanel.add(algoBox);
        controlPanel.add(new JLabel("Heuristic:"));
        controlPanel.add(heuristicBox);
        controlPanel.add(solveButton);
        add(controlPanel, BorderLayout.NORTH);

        boardPanel = new JPanel() {
            @Override
            protected void paintComponent(Graphics g) {
                super.paintComponent(g);
                if (solutionStates != null && animIndex <
solutionStates.size()) {
                    drawBoard(g, solutionStates.get(animIndex));
                }
            }
        };
        add(boardPanel, BorderLayout.CENTER);

        loadButton.addActionListener(e -> loadBoardFile());
        algoBox.addActionListener(e -> {
            boolean needHeuristic = algoBox.getSelectedIndex() == 0 ||
algoBox.getSelectedIndex() == 2;
            heuristicBox.setEnabled(needHeuristic);
        });
        solveButton.addActionListener(e -> startSolve());

        setVisible(true);
    }

    private void loadBoardFile() {
        JFileChooser chooser = new JFileChooser();
        chooser.setFileFilter(new FileNameExtensionFilter("Text Files",
"txt"));
        if (chooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION)
{
            currentFile = chooser.getSelectedFile();
```

```
            solveButton.setEnabled(true);
        }
    }

    private void startSolve() {
        try {
            RushHour.Exit exit = new RushHour.Exit();
            char[][] board =
RushHour.parseBoard(currentFile.getAbsolutePath(), exit);
            RushHour.State start = new RushHour.State(board);

            String algo = (String) algoBox.getSelectedItem();
            RushHour.Heuristic h = null;
            if (algo.equals("Greedy") || algo.equals("A*")) {
                String hname = (String) heuristicBox.getSelectedItem();
                h = hname.equals("Manhattan") ? new RushHour.Manhattan()
: new RushHour.Euclidean();
            }

            long startTime = System.nanoTime();
            RushHour.Node goal;
            if (algo.equals("Backtracking")) {
                goal = RushHour.backtrack(new RushHour.Node(start, null,
null, 0, 0), exit, new HashSet<>());
            } else {
                goal = RushHour.search(start, exit, h, algo);
            }
            long endTime = System.nanoTime();
            double elapsedMs = (endTime - startTime) / 1e6;

            if (goal == null) {
                JOptionPane.showMessageDialog(this, String.format("No
solution found. (%.2f ms)", elapsedMs));
                return;
            }
            JOptionPane.showMessageDialog(this, String.format("Solution
found in %.2f ms.", elapsedMs));

            solutionStates = new ArrayList<>();
```

```
            RushHour.Node n = goal;
            while (n != null) {
                solutionStates.add(0, n.state.board);
                n = n.parent;
            }
            animIndex = 0;
            if (animationTimer != null && animationTimer.isRunning())
animationTimer.stop();
            animationTimer = new Timer(800, e -> animateStep());
            animationTimer.start();
        } catch (IOException ex) {
            JOptionPane.showMessageDialog(this, "Error loading board: "
+ ex.getMessage());
        }
    }

    private void animateStep() {
        if (animIndex >= solutionStates.size()) {
            animationTimer.stop();
            return;
        }
        boardPanel.repaint();
        animIndex++;
    }

    private void drawBoard(Graphics g, char[][] b) {
        int rows = b.length, cols = b[0].length;
        int w = boardPanel.getWidth(), h = boardPanel.getHeight();
        int cellW = w / cols, cellH = h / rows;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                g.setColor(Color.LIGHT_GRAY);
                g.fillRect(j*cellW, i*cellH, cellW, cellH);
                g.setColor(Color.BLACK);
                g.drawRect(j*cellW, i*cellH, cellW, cellH);
                char ch = b[i][j];
                if (ch != '.' ) {
                    g.setFont(new Font("Arial", Font.BOLD, cellH/2));
                    g.drawString(String.valueOf(ch), j*cellW + cellW/3,
```

```
i*cellH + cellH*2/3);
                }
            }
        }
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(GUI::new);
    }
}
```

Demonstrasi:
https://drive.google.com/drive/folders/1EeiZJPVnE2filj7Bxe1BFct7bligKgXI?usp=sharing

Lampiran:
https://github.com/Azekhiel/Tucil2_13222061

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