

.

LAPORAN TUGAS KECIL 03
IF2211 STRATEGI ALGORITMA

**PENYELESAIAN PERMAINAN WORD LADDER MENGGUNAKAN
ALGORITMA UCS, GREEDY BEST FIRST SEARCH, DAN A***



Disusun oleh:

Benardo

13522055

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

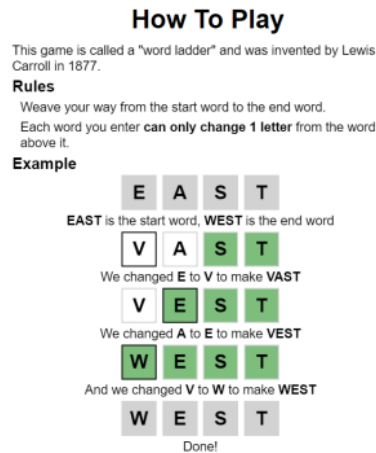
2024

DAFTAR ISI

| | |
|--|-----------|
| BAB I | |
| DESKRIPSI MASALAH..... | 3 |
| BAB II | |
| LANDASAN TEORI..... | 4 |
| 2.1 Algoritma Uniform Cost Search (UCS)..... | 4 |
| 2.2 Algoritma Greedy Best First Search..... | 5 |
| 2.3 Algoritma A* Search..... | 6 |
| BAB III | |
| IMPLEMENTASI PROGRAM..... | 7 |
| 3.1. Main.java..... | 7 |
| 3.2. WordLadderGUI.java (Bonus)..... | 7 |
| 3.3. WordLadderSolver.java..... | 11 |
| 3.4. ResultSolver.java..... | 15 |
| 3.5. Node.java..... | 16 |
| 3.6. DictionaryLoader.java..... | 16 |
| BAB IV | |
| ANALISIS DAN PENGUJIAN..... | 18 |
| 4.1. Pengujian 1..... | 18 |
| 1.1. Pengujian Algoritma Uniform Cost Search..... | 18 |
| 1.2. Pengujian Algoritma Greedy Best First Search..... | 22 |
| 1.3. Pengujian Algoritma A* Search..... | 27 |
| 1.4. Hasil Pengujian..... | 32 |
| 4.2. Analisis..... | 33 |
| 4.3. Penjelasan Bonus..... | 38 |
| BAB V | |
| PENUTUP..... | 39 |
| 5.1. Kesimpulan..... | 39 |
| 5.2. Saran..... | 39 |
| DAFTAR REFERENSI..... | 41 |
| LAMPIRAN..... | 42 |

BAB I

DESKRIPSI MASALAH



Gambar 1. Ilustrasi dan Peraturan Permainan Word Ladder

(Sumber: <https://wordwormdormdork.com/>)

Word ladder (juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragrams, laddergrams, atau word golf) adalah salah satu permainan kata yang terkenal bagi seluruh kalangan. Word ladder ditemukan oleh Lewis Carroll, seorang penulis dan matematikawan, pada tahun 1877. Pada permainan ini, pemain diberikan dua kata yang disebut sebagai start word dan end word. Untuk memenangkan permainan, pemain harus menemukan rantai kata yang dapat menghubungkan antara start word dan end word. Banyaknya huruf pada start word dan end word selalu sama. Tiap kata yang berdekatan dalam rantai kata tersebut hanya boleh berbeda satu huruf saja. Pada permainan ini, diharapkan solusi optimal, yaitu solusi yang meminimalkan banyaknya kata yang dimasukkan pada rantai kata. Pada tugas kali ini, kami disuruh untuk membuat suatu solver permainan ini dengan harapan kita menemukan solusi yang paling optimal untuk menyelesaikan permainan Word Ladder ini. Untuk menyelesaikan permasalahan ini, bisa menggunakan beberapa algoritma pencarian seperti algoritma Uniform Cost Search(UCS) algoritma Greedy Best First Search, dan Algoritma A* search.

BAB II

LANDASAN TEORI

2.1 Algoritma Uniform Cost Search (UCS)

Algoritma Uniform Cost Search (UCS) adalah teknik pencarian yang tidak mengandalkan informasi prediksi atau heuristik tentang tujuan, yang menjadikannya bagian dari kelompok pencarian tanpa informasi atau "uninformed search". Algoritma ini bekerja dengan mengiterasi setiap simpul dalam sebuah graf berbobot berdasarkan biaya terendah dari simpul awal. Simpul dan rute dalam graf diwakili dengan skor yang mencerminkan biaya untuk berpindah dari satu simpul ke simpul lain. Proses ini menggunakan struktur data berupa priority queue untuk mengatur simpul-simpul berdasarkan skor terkecil, memastikan bahwa simpul dengan biaya terendah selalu diproses terlebih dahulu.

Dalam penerapannya, UCS memulai dengan simpul awal yang ditentukan oleh pengguna dan mencari ke simpul tujuan, juga ditentukan oleh pengguna dan sering kali ditandai secara visual (misalnya dengan warna merah). Dari simpul awal, UCS memeriksa setiap simpul yang terhubung langsung berdasarkan skor atau biaya terendah. Simpul yang telah diperiksa akan dihapus dari priority queue dan skor yang terakumulasi dari simpul awal ke simpul tersebut dihitung. Proses ini terus berlanjut, dengan simpul baru yang diperiksa diambil dari depan priority queue sampai simpul tujuan ditemukan. Setelah simpul tujuan tercapai, pencarian dihentikan dan priority queue dikosongkan.

Kelebihan utama UCS adalah kemampuannya untuk selalu menemukan rute yang paling optimal berkat fokusnya pada pencarian simpul dengan biaya terendah terlebih dahulu. Namun, kelemahannya termasuk penggunaan memori yang tinggi dan proses yang lebih lambat dibandingkan dengan algoritma pencarian yang memanfaatkan informasi tambahan, seperti algoritma pencarian informed. Algoritma informed seperti A* mencapai efisiensi lebih tinggi dengan menggunakan heuristik untuk memperkirakan

biaya dari simpul saat ini ke tujuan, memungkinkan pencarian yang lebih cepat namun masih mempertahankan keoptimalan solusi serupa dengan UCS.

Secara keseluruhan, UCS adalah alat yang kuat dalam menentukan jalur terpendek dalam graf berbobot, ideal untuk aplikasi dimana keakuratan dan keoptimalan lebih diprioritaskan daripada kecepatan dan efisiensi penggunaan memori.

2.2 Algoritma Greedy Best First Search

Greedy Best-First Search adalah algoritma informed search yang menggunakan fungsi heuristik untuk mengevaluasi setiap simpul dalam graf. Tujuan utamanya adalah mencapai simpul tujuan dengan langkah paling optimal, memilih simpul yang paling menjanjikan berdasarkan estimasi heuristik. Dalam hal ini, fungsi evaluasi ($f(x)$) pada setiap simpul (x) adalah sama dengan nilai heuristik ($h(x)$).

Algoritma ini memprioritaskan ekspansi simpul yang tampak mendekati tujuan, berdasarkan perkiraan biaya terkecil dari simpul saat ini ke tujuan. Meskipun demikian, algoritma ini tidak menjamin penemuan jalur dengan biaya terendah, karena tidak memperhitungkan estimasi biaya dari simpul saat ini ke tujuan. Oleh karena itu, meskipun Greedy Best-First Search dapat menemukan jalur lebih cepat dari algoritma Uniform Cost Search (UCS) dalam hal waktu, tidak selalu menjamin penemuan jalur terpendek.

Kelemahan utama algoritma ini adalah kemungkinan terjebak pada solusi lokal yang optimal, terutama dalam graf yang kompleks atau ketika heuristik tidak akurat. Meskipun demikian, dengan menggunakan heuristik yang baik, algoritma ini dapat menemukan jalur lebih cepat, meskipun tidak selalu optimal. Greedy Best-First Search sering digunakan dalam penyelesaian permasalahan penentuan rute, namun dapat juga diterapkan pada berbagai domain permasalahan lainnya seperti pemecahan puzzle.

2.3 Algoritma A* Search

Algoritma A* merupakan gabungan antara Universal Cost Search (UCS) dan Greedy Best First Search, mengadopsi prinsip-prinsip keduanya. Dalam A*, fungsi biaya ($f(n)$) untuk mengevaluasi setiap simpul (n) adalah hasil penjumlahan antara biaya aktual dari simpul awal ke simpul tersebut ($g(n)$) dan estimasi biaya dari simpul tersebut ke simpul tujuan ($h(n)$). A* memilih untuk mengekspansi simpul dengan nilai $f(n)$ terendah, sehingga mencari jalur dengan biaya terendah dari simpul awal ke simpul tujuan.

Algoritma A* memerlukan heuristik yang admissible, artinya heuristik tersebut tidak pernah memberikan estimasi biaya yang lebih besar dari biaya sebenarnya. Hal ini penting untuk memastikan bahwa A* dapat menemukan jalur terpendek. Heuristik ini digunakan untuk mengarahkan pencarian ke arah yang benar, sehingga meningkatkan efisiensi pencarian jalur.

Dalam bidang ilmu komputer, A* dikenal luas karena keakuratannya dalam mencari jalur dan melintasi grafik. Algoritma ini sangat efisien dan efektif karena memanfaatkan keunggulan UCS dalam menemukan jalur dengan biaya minimal dan kecepatan Greedy Best First Search dalam mendekati tujuan. Dengan menggunakan fungsi heuristik yang tepat, A* mampu menemukan jalur dengan biaya terkecil dari simpul awal ke simpul tujuan, membuatnya menjadi pilihan yang populer untuk berbagai aplikasi, seperti pencarian jalur dalam permainan video, navigasi GPS, dan pemodelan jaringan logistik.

BAB III

IMPLEMENTASI PROGRAM

Berikut merupakan dari implementasi source code program, beserta penjelasan tiap class dan method yang diimplementasi.

3.1. Main.java

Main.java

```
import java.util.*;

import javax.swing.SwingUtilities;

public class Main {
    public static void main(String[] args) {
        Set<String> dictionary = DictionaryLoader.loadWordsFromFile("bin/dictionary.txt");

        SwingUtilities.invokeLater(() -> {
            WordLadderGUI frame = new WordLadderGUI(dictionary);
            frame.setVisible(true);
        });
    }
}
```

Di dalam file Main.java, terdapat class Main. Class Main mempunyai method main yang berfungsi untuk membaca file dictionary.txt yang terletak di folder bin, lalu menyimpan kata-kata di dalam sebuah Set bertipe string dengan nama dictionary. Lalu akan menginisiasi class WordLadderGUI untuk memanggil jalannya GUI untuk program ini.

3.2. WordLadderGUI.java (Bonus)

WordLadderGUI.java

```
import javax.swing.*;
import javax.swing.border.Border;
import javax.swing.border.EmptyBorder;

import java.awt.*;
import java.util.Collections;
import java.util.Set;

public class WordLadderGUI extends JFrame {
    private JTextField startWordField;
    private JTextField endWordField;
    private JButton ucsButton;
    private JButton greedyButton;
```

```

private JButton aStarButton;
private JLabel resultLabel;
private WordLadderSolver solver;
private JPanel outputPanel;

public WordLadderGUI(Set<String> dictionary) {
    solver = new WordLadderSolver(dictionary);

    setTitle("Word Ladder Game");
    setSize(600, 500);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    getContentPane().setBackground(Color.BLACK);
    setLayout(new BorderLayout(10, 10));
    ((JPanel) getContentPane()).setBorder(new EmptyBorder(10, 10, 10, 10));

    // Top Panel for Inputs
    JPanel containerPanel = new JPanel();
    containerPanel.setLayout(new BoxLayout(containerPanel, BoxLayout.Y_AXIS));
    containerPanel.setBackground(Color.BLACK);

    // Top Panel for Inputs
    JPanel inputPanel = new JPanel();
    inputPanel.setLayout(new GridLayout(2, 2, 5, 5));
    inputPanel.setBackground(Color.BLACK);
    JLabel startword = new JLabel("Start Word      :");
    startword.setForeground(Color.WHITE);
    inputPanel.add(startword);
    Border line = BorderFactory.createLineBorder(Color.DARK_GRAY);
    Border margin = BorderFactory.createEmptyBorder(5, 5, 5, 5); // top, left, bottom,
right padding
    Border compound = BorderFactory.createCompoundBorder(line, margin);
    startWordField = new JTextField();
    startWordField.setBorder(compound);
    startWordField.setBackground(Color.BLACK);
    startWordField.setForeground(Color.WHITE);
    inputPanel.add(startWordField);
    JLabel endword = new JLabel("End Word      :");
    endword.setForeground(Color.WHITE);
    inputPanel.add(endword);
    endWordField = new JTextField();
    endWordField.setBackground(Color.BLACK);
    endWordField.setForeground(Color.WHITE);

    endWordField.setBorder(compound);
    inputPanel.add(endWordField);
    containerPanel.add(inputPanel);

    containerPanel.add(Box.createVerticalStrut(20));

    // Button Panel for Solving Algorithms
    JPanel buttonPanel = new JPanel();
    buttonPanel.setBackground(Color.BLACK);
    ucsButton = createButton("Solve with UCS");
    greedyButton = createButton("Solve with Greedy BFS");
    aStarButton = createButton("Solve with A*");
    buttonPanel.add(ucsButton);
    buttonPanel.add(greedyButton);
    buttonPanel.add(aStarButton);
    containerPanel.add(buttonPanel);

```



```

add(containerPanel, BorderLayout.NORTH);

// Setup action listeners for each button
ucsButton.addActionListener(e → solve("UCS"));
greedyButton.addActionListener(e → solve("Greedy BFS"));
aStarButton.addActionListener(e → solve("A*"));

outputPanel = new JPanel();
outputPanel.setLayout(new BoxLayout(outputPanel, BoxLayout.Y_AXIS));
outputPanel.setBackground(Color.BLACK);
JScrollPane scrollPane = new JScrollPane(outputPanel);
scrollPane.setPreferredSize(new Dimension(580, 300));
add(scrollPane, BorderLayout.CENTER);
// Output Table for displaying the path

// Result Labels
resultLabel = new JLabel("Selected Algorithm: None      Execution Time: 0 ms
Visited Nodes: 0");
resultLabel.setHorizontalAlignment(JLabel.CENTER);
resultLabel.setForeground(Color.WHITE);
add(resultLabel, BorderLayout.SOUTH);
}

private JButton createButton(String text) {
    JButton button = new JButton(text);
    button.setBackground(Color.GREEN);
    button.setForeground(Color.BLACK);
    button.setFocusPainted(false);

    // Create a compound border to add padding
    Border line = BorderFactory.createLineBorder(Color.DARK_GRAY);
    Border margin = BorderFactory.createEmptyBorder(10, 15, 10, 15);
    Border compound = BorderFactory.createCompoundBorder(line, margin);
    button.setBorder(compound);
    return button;
}

private void solve(String algorithm) {
    String start = startWordField.getText().trim().toLowerCase();
    String end = endWordField.getText().trim().toLowerCase();

    if (start.length() ≠ end.length()) {
        JOptionPane.showMessageDialog(this, "Start word and end word must be the same
length.");
        return;
    }

    if (!solver.getDictionary().contains(start) || !solver.getDictionary().contains(end)){
        JOptionPane.showMessageDialog(this, "Invalid Input.");
        return;
    }

    Runtime runtime = Runtime.getRuntime();
    long startmemory = runtime.totalMemory() - runtime.freeMemory();
    System.out.println("Start : " + start);
    System.out.println("End : " + end);
    System.out.println("Algorithm : " + algorithm);
    SolverResult result;
    switch (algorithm) {
        case "UCS":

```

```

        result = solver.solveUCS(start, end);
        break;
    case "Greedy BFS":
        result = solver.solveUsingGreedy(start, end);
        break;
    case "A*":
        result = solver.solveUsingAStar(start, end);
        break;
    default:
        result = new SolverResult(Collections.emptyList(), 0, 0);
}

long finalMemory = runtime.totalMemory() - runtime.freeMemory();
long memoryUsed = (finalMemory - startmemory);

System.out.println("Memory Used : " + memoryUsed + " bytes");
updateResultDisplay(result, algorithm);
}

private void updateResultDisplay(SolverResult result, String algorithm) {
    outputPanel.removeAll();

    if (result.getPath().isEmpty()) {
        JOptionPane.showMessageDialog(this, "No path found.");
    } else {
        for (int i = 0; i < result.getPath().size(); i++) {
            JPanel wordPanel = new JPanel();
            wordPanel.setBackground(Color.BLACK);
            wordPanel.setForeground(Color.WHITE);
            wordPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
            String current = result.getPath().get(i);
            String prev = i != 0 ? result.getPath().get(i-1) : null;

            JLabel num = new JLabel((i + 1) + ". ");
            num.setForeground(Color.WHITE);
            wordPanel.add(num);

            for (int j = 0; j < current.length(); j++) {
                JLabel label = new JLabel(String.valueOf(current.charAt(j)));
                label.setForeground(Color.WHITE);
                if (prev != null && j < prev.length() && current.charAt(j) !=
prev.charAt(j)) {
                    String styledText = "<html><u><font color='green'
style='font-weight:bold;'>" + current.charAt(j) + "</font></u></html>";
                    label.setText(styledText);
                }
                wordPanel.add(label);
            }

            outputPanel.add(wordPanel);
        }

        outputPanel.revalidate();
        outputPanel.repaint();
    }

    resultLabel.setText("Selected Algorithm: " + algorithm +
        "
        Execution Time: " + result.getTimeTaken() + " ms" +
        "
        Visited Nodes: " + result.getVisitedNodes());
}

```

```
}  
}
```

Di dalam file WordLadderGUI.java di atas , terdapat kelas bernama WordLadderGUI. Kelas ini mempunyai fungsi untuk menginisiasi seluruh komponen visual yang terdapat dalam program GUI ini seperti Button , Input field, Output Field , dan Text untuk menunjukkan hasil pencarian. Terdapat konstruktor WordLadderGUI yang digunakan untuk menyusun letak seluruh komponen yang terdapat dalam program GUI , juga memasang logika untuk setiap button , sehingga ketika user melakukan penekanan button , akan memanggil fungsi solve sesuai dengan algoritma yang dipilih. Selain itu konstruktor ini juga memvalidasi inputan dari pengguna apakah sudah valid atau belum.terdapat juga method *createButton* yang berfungsi untuk membuat suatu button, method *solve* yang berfungsi untuk memvalidasi inputan dari user , lalu memanggil fungsi untuk mendapatkan hasil pencarian sesuai dengan pilihan algoritma user, method *updateResultDisplay* yang berfungsi untuk menampilkan semua hasil pencarian ke layar.

3.3. WordLadderSolver.java

WordLadderSolver.java

```
import java.util.*;  
  
public class WordLadderSolver {  
    private Set<String> dictionary;  
    private static final char[] ALPHABET = "abcdefghijklmnopqrstuvwxyz".toCharArray();  
  
    public WordLadderSolver(Set<String> dictionary) {  
        this.dictionary = dictionary;  
    }  
  
    public Set<String> getDictionary(){  
        return dictionary;  
    }  
  
    public SolverResult solveUCS(String start, String end) {  
        long startTime = System.currentTimeMillis();  
        if (!dictionary.contains(start) || !dictionary.contains(end)) {  
            return new SolverResult(Collections.emptyList(), 0, 0); // Start or end word not in  
dictionary  
        }  
  
        PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(node →  
node.cost));  
        Set<String> visited = new HashSet<>();  
        queue.add(new Node(start, null, 0));  
        visited.add(start);  
        int visitedCount = 0;  
        while (!queue.isEmpty()) {  
            Node current = queue.poll();
```

```

        visitedCount++;
        if (current.word.equals(end)) {
            long endTime = System.currentTimeMillis();
            return new SolverResult(constructPath(current), endTime - startTime,
visitedCount);
        }

        for (String neighbor : getNeighbors(current.word)) {
            if (!visited.contains(neighbor)) {
                visited.add(neighbor);
                queue.add(new Node(neighbor, current, current.cost + 1));
            }
        }
    }
    long endTime = System.currentTimeMillis();
    return new SolverResult(Collections.emptyList(), endTime - startTime, visitedCount); //
No solution found
}

    public SolverResult solveUsingGreedy(String start, String end) {
        long startTime = System.currentTimeMillis();
        if (!dictionary.contains(start) || !dictionary.contains(end)) {
            return new SolverResult(Collections.emptyList(), 0, 0); // Start or end word not in
dictionary
        }

        PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(node →
node.cost));
        Set<String> visited = new HashSet<>();
        queue.add(new Node(start, null, getHeuristic(start, end)));
        visited.add(start);
        int visitedCount = 0;
        while (!queue.isEmpty()) {
            Node current = queue.poll();
            visitedCount++;
            if (current.word.equals(end)) {
                long endTime = System.currentTimeMillis();
                return new SolverResult(constructPath(current), endTime -
startTime,visitedCount);
            }

            for (String neighbor : getNeighbors(current.word)) {
                if (!visited.contains(neighbor)) {
                    visited.add(neighbor);
                    queue.add(new Node(neighbor, current, getHeuristic(neighbor, end)));
                }
            }
        }
        long endTime = System.currentTimeMillis();
        return new SolverResult(Collections.emptyList(), endTime - startTime,visitedCount); //
No solution found
    }

    public SolverResult solveUsingAStar(String start, String end) {
        long startTime = System.currentTimeMillis();
        if (!dictionary.contains(start) || !dictionary.contains(end)) {
            return new SolverResult(Collections.emptyList(), 0, 0); // Start or end word not in
dictionary
        }

```

```

        PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(node →
node.cost + getHeuristic(node.word, end)));
        Set<String> visited = new HashSet<>();
        queue.add(new Node(start, null, getHeuristic(start, end)));
        visited.add(start);
        int visitedCount = 0;

        while (!queue.isEmpty()) {
            Node current = queue.poll();
            visitedCount++;
            if (current.word.equals(end)) {
                long endTime = System.currentTimeMillis();
                return new SolverResult(constructPath(current), endTime - startTime,
visitedCount);
            }

            for (String neighbor : getNeighbors(current.word)) {
                if (!visited.contains(neighbor)) {
                    visited.add(neighbor);
                    Node nextNode = new Node(neighbor, current, current.cost + 1 +
getHeuristic(neighbor, end));
                    queue.add(nextNode);
                }
            }
            long endTime = System.currentTimeMillis();
            return new SolverResult(Collections.emptyList(), endTime - startTime, visitedCount);
        }

        private int getHeuristic(String currentWord, String endWord) {
            int differenceCount = 0;
            for (int i = 0; i < currentWord.length(); i++) {
                if (currentWord.charAt(i) ≠ endWord.charAt(i)) {
                    differenceCount++;
                }
            }
            return differenceCount;
        }

        private List<String> getNeighbors(String word) {
            List<String> neighbors = new ArrayList<>();
            char[] chars = word.toCharArray();

            for (int i = 0; i < word.length(); i++) {
                char originalChar = chars[i];
                for (char c : ALPHABET) {
                    if (c ≠ originalChar) {
                        chars[i] = c;
                        String newWord = new String(chars);
                        if (dictionary.contains(newWord)) {
                            neighbors.add(newWord);
                        }
                    }
                }
                chars[i] = originalChar; // Restore original character
            }

            return neighbors;
        }

```

```

private List<String> constructPath(Node endNode) {
    LinkedList<String> path = new LinkedList<>();
    Node current = endNode;
    while (current != null) {
        path.addFirst(current.word);
        current = current.parent;
    }
    return path;
}
}

```

Dalam file `WordLadderSolver.java` di atas mempunyai fungsi untuk melakukan pencarian jalur dari kata awal ke kata akhir. Kelas ini mempunyai atribut `library` yang merupakan data kata-kata yang akan digunakan untuk memvalidasi kata tetangga yang valid. Terdapat beberapa method dalam kelas ini.

Method *`solveUCS`* adalah method untuk melakukan pencarian jalur terpendek dengan menggunakan algoritma UCS. Method ini dimulai dengan mengecek apakah kata awal dan kata akhir terdapat dalam `datastore` kata-kata, selanjutnya akan dilakukan inisiasi `priority queue` yang akan digunakan untuk menampung node yang akan dikunjungi. Selain itu, diinisiasi juga `set visited` yang digunakan untuk menampung node yang sudah dikunjungi. Setiap elemen akan di `prio queue` akan terurut berdasarkan `cost` dari node tersebut, `cost` yang rendah akan mendapatkan prioritas terlebih dahulu. `Cost` ini didapatkan dengan membandingkan kedalaman node tersebut dari kata awal. Pengecekan setiap elemen di `prio queue` akan dilakukan terus menerus sampai didapatkan node yang sama dengan node tujuan, baru dipanggil fungsi *`constructPath`* untuk mencari path yang menghubungkan kata awal dan kata tujuan.

Method *`solveUsingGreedy`* adalah method untuk melakukan pencarian jalur terpendek dengan menggunakan algoritma Greedy Best first search. Method ini dimulai dengan mengecek apakah di dalam `dictionary` terdapat kata awal dan kata akhir. Selanjutnya akan dilakukan inisiasi `priority queue` yang akan mengurutkan elemennya berdasarkan `cost` dari elemen tersebut, lalu dilakukan inisiasi juga `set` bertipe `string` untuk menampung node yang sudah dilakukan pengecekan. `Cost` dari setiap node akan didapatkan dengan fungsi *`getHeuristic`*. Fungsi ini akan mengecek jumlah perbedaan kata dari node sekarang dengan node tujuan. Semakin mirip dengan kata tujuan, maka `cost` semakin kecil, dan akan diprioritaskan di dalam `prioqueue`. Pengecekan akan dilakukan untuk semua elemen dalam `queue` sampai didapatkan node dengan kata yang sama.

Method *solveusingAstar* adalah method untuk melakukan pencarian jalur terpendek dengan menggunakan algoritma A* search . Method ini mempunyai cara kerja yang sama dengan *solveUCS* dan *solveUsingGreedy* , yang berbeda hanya fungsi yang digunakan untuk menghitung cost. Dalam pencarian dengan A* search , cost akan dihitung dengan melakukan penjumlahan ke dalam node dari node awal dengan perbedaan node ke node akhir, sehingga untuk menghitung cost sebuah node dapat dilakukan dengan menambah kedalaman ditambah dengan pemanggilan fungsi *getHeuristic*.

Method *getHeuristic* untuk menghitung cost suatu node ke node akhir. Terdapat juga method *getNeighbors* yang akan dilakukan untuk mencari tetangga dari sebuah node , yang kemudian akan dimasukkan ke dalam queue. Terakhir, method *constructPath* digunakan untuk mencari mem wrap parent-parent dari node yang sudah ditemukan.

3.4. ResultSolver.java

ResultSolver.java

```
import java.util.*;
class SolverResult {
    private List<String> path;
    private long timeTaken;
    private int visitedNodes;

    // Constructor to initialize the results
    public SolverResult(List<String> path, long timeTaken, int visitedNodes) {
        this.path = path;
        this.timeTaken = timeTaken;
        this.visitedNodes = visitedNodes;
    }

    public List<String> getPath() {
        return path;
    }

    public long getTimeTaken() {
        return timeTaken;
    }

    public int getVisitedNodes() {
        return visitedNodes;
    }
}
```

Dalam file ResultSolver.java , terdapat kelas ResultSolver yang akan digunakan untuk menyimpan hasil pencarian yang akan direturn untuk digunakan di GUI. Dalam kelas ini terdapat atribut List of path, lama waktu , dan jumlah node yang dikunjungi.

3.5. Node.java

Node.java

```
class Node implements Comparable<Node> {
    String word;
    Node parent;
    int cost;

    public Node(String word, Node parent, int cost) {
        this.word = word;
        this.parent = parent;
        this.cost = cost;
    }

    // Untuk PriorityQueue, yang membandingkan berdasarkan cost
    @Override
    public int compareTo(Node other) {
        return Integer.compare(this.cost, other.cost);
    }
}
```

Dalam kelas Node.java , terdapat kelas Node yang akan dilakukan untuk menginisiasi object node . Sebuah node terdiri dari atribut ,word (yang merupakan nama dari node tersebut), parent(menyimpan alamat dari parent node tersebut) , dan cost (yang merupakan biaya dari node ini).

3.6. DictionaryLoader.java

DictionaryLoader.java

```
import java.io.*;
import java.util.HashSet;
import java.util.Set;

public class DictionaryLoader {
    public static Set<String> loadWordsFromFile(String filename) {
        Set<String> words = new HashSet<>();
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = reader.readLine()) != null) {
                words.add(line.trim().toLowerCase());
            }
        } catch (IOException e) {
            System.err.println("Error reading from file: " + e.getMessage());
        }
        return words;
    }
}
```


Dalam file DictionaryLoader.java , terdapat kelas DictionaryLoader. Kelas ini digunakan untuk memuat kata-kata dalam file txt yang akan dijadikan data store kata-kata.

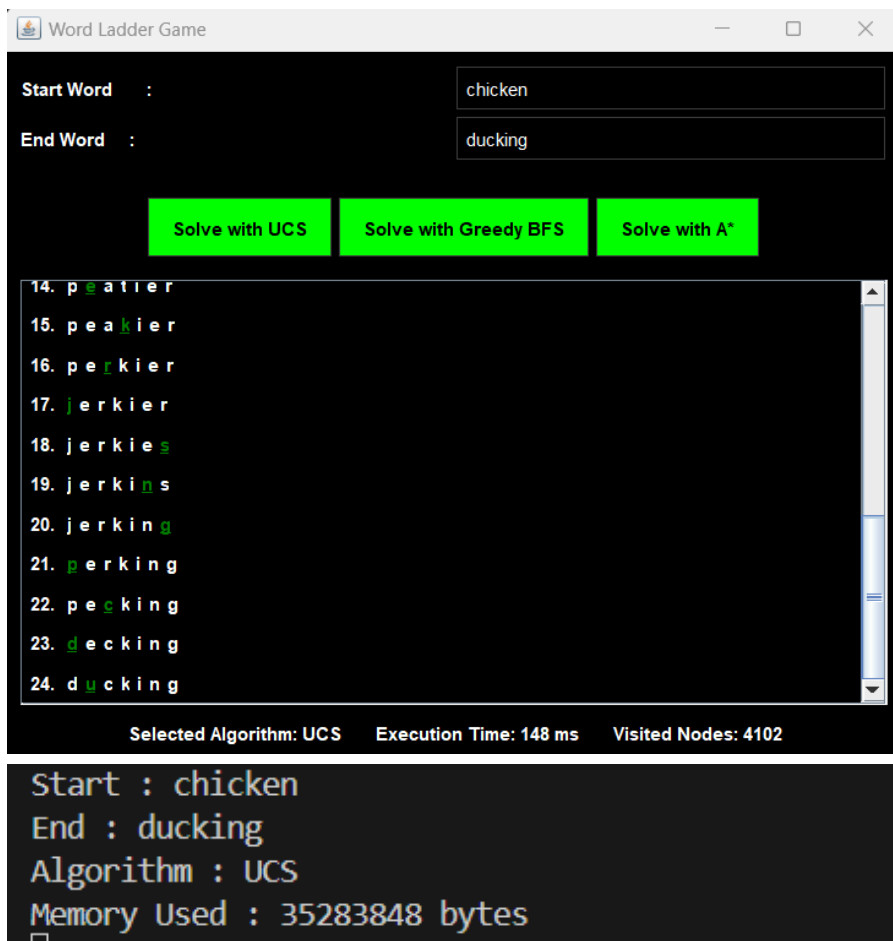
BAB IV

ANALISIS DAN PENGUJIAN

4.1. Pengujian 1

1.1. Pengujian Algoritma Uniform Cost Search

Tabel 1. Pengujian Algoritma Uniform Cost Search

| No. | Hasil Pengujian |
|-----|---|
| 1 |  <p>The screenshot displays the 'Word Ladder Game' interface. At the top, the 'Start Word' is 'chicken' and the 'End Word' is 'ducking'. Below these are three buttons: 'Solve with UCS' (highlighted in green), 'Solve with Greedy BFS', and 'Solve with A*'. The main area shows a list of 11 words in a ladder format, numbered 14 to 24. The words are: 14. p e a t i e r, 15. p e a k i e r, 16. p e k i e r, 17. j e r k i e r, 18. j e r k i e s, 19. j e r k i n s, 20. j e r k i n g, 21. p e r k i n g, 22. p e c k i n g, 23. d e c k i n g, and 24. d u c k i n g. At the bottom, a status bar shows 'Selected Algorithm: UCS', 'Execution Time: 148 ms', and 'Visited Nodes: 4102'. Below the screenshot, a text box contains the following information: Start : chicken, End : ducking, Algorithm : UCS, and Memory Used : 35283848 bytes.</p> |

2

Word Ladder Game

Start Word :

sorry

End Word :

thank

Solve with UCS

Solve with Greedy BFS

Solve with A*

1. s o r r y

2. s e r r y

3. t e r r y

4. t e a r y

5. t e a r e

6. t e a m s

7. t r a m s

8. t r a n s

9. t r a n k

10. t h a n k

Selected Algorithm: UCS

Execution Time: 135 ms

Visited Nodes: 6079

Start : sorry

End : thank

Algorithm : UCS

Memory Used : 32372072 bytes

3

Word Ladder Game

Start Word : stock

End Word : known

Solve with UCS

Solve with Greedy BFS

Solve with A*

1. stock

2. sto o k

3. sto o p

4. sto w p

5. sto w s

6. s n o w s

7. k n o w s

8. k n o w n

Selected Algorithm: UCS

Execution Time: 32 ms

Visited Nodes: 1996

Start : stock

End : known

Algorithm : UCS

Memory Used : 12582696 bytes

4

Word Ladder Game

Start Word :

End Word :

4. a **n** o l e
5. a n **i** l e
6. a n i **s** e
7. a **r** i s e
8. **p** r i s e
9. p r i s **s**
10. p r i e s
11. p r i e **d**
12. p r e e d
13. **b** r e e d
14. b r e a **d**

Selected Algorithm: UCS Execution Time: 25 ms Visited Nodes: 1548

Start : apple
End : bread
Algorithm : UCS
Memory Used : 9230096 bytes

5

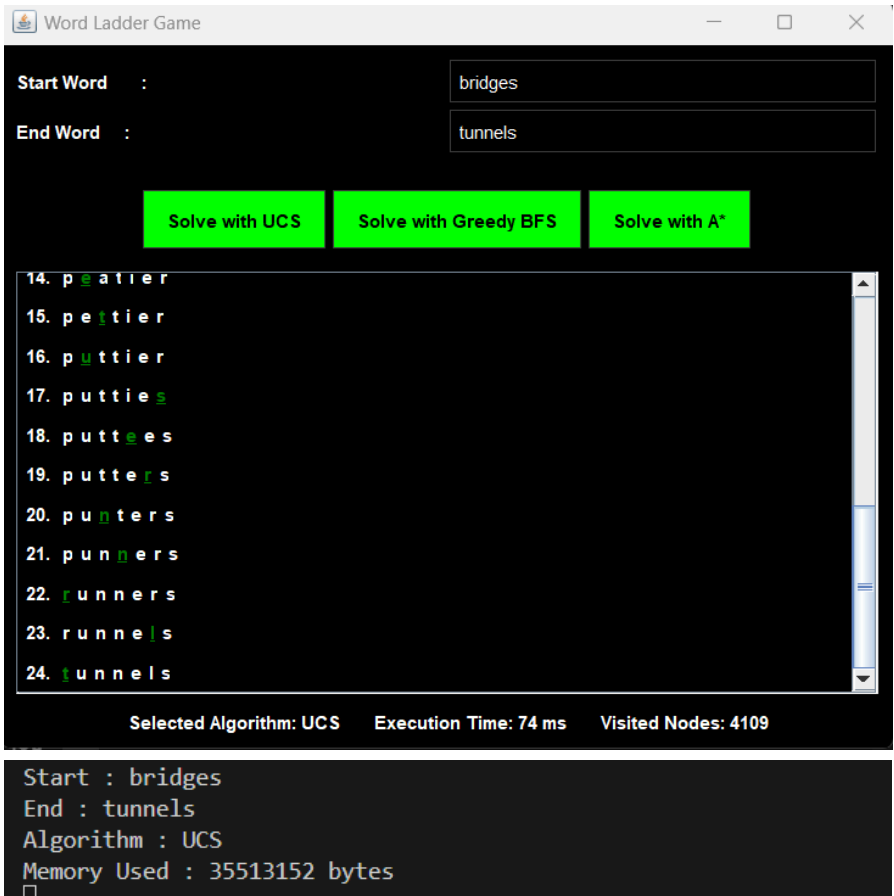
Word Ladder Game

Start Word :

End Word :

1. s t o n e
2. s **h** o n e
3. **p** h o n e
4. p h o n **y**
5. p e o n y
6. p e **n** n y
7. **b** e n n y
8. b o n n y
9. b o n e y
10. **m** o n e y

Selected Algorithm: UCS Execution Time: 63 ms Visited Nodes: 5045

| | |
|---|---|
| | <pre> Start : stone End : money Algorithm : UCS Memory Used : 31721736 bytes </pre> |
| 6 |  |

1.2. Pengujian Algoritma Greedy Best First Search

Tabel 2. Pengujian Algoritma Greedy Best First Search

| | |
|-----|-----------------|
| No. | Hasil Pengujian |
|-----|-----------------|

1

Word Ladder Game

Start Word : chicken

End Word : ducking

Solve with UCS Solve with Greedy BFS Solve with A*

86. p o c k i e r
87. p e c k i e r
88. p e r k i e r
89. j e r k i e r
90. j e r k i e s
91. j e r k i n s
92. j e r k i n g
93. p e r k i n g
94. p e c k i n g
95. d e c k i n g
96. d u c k i n g

Selected Algorithm: Greedy BFS Execution Time: 7 ms Visited Nodes: 469

Start : chicken
End : ducking
Algorithm : Greedy BFS
Memory Used : 4221960 bytes

2

Word Ladder Game

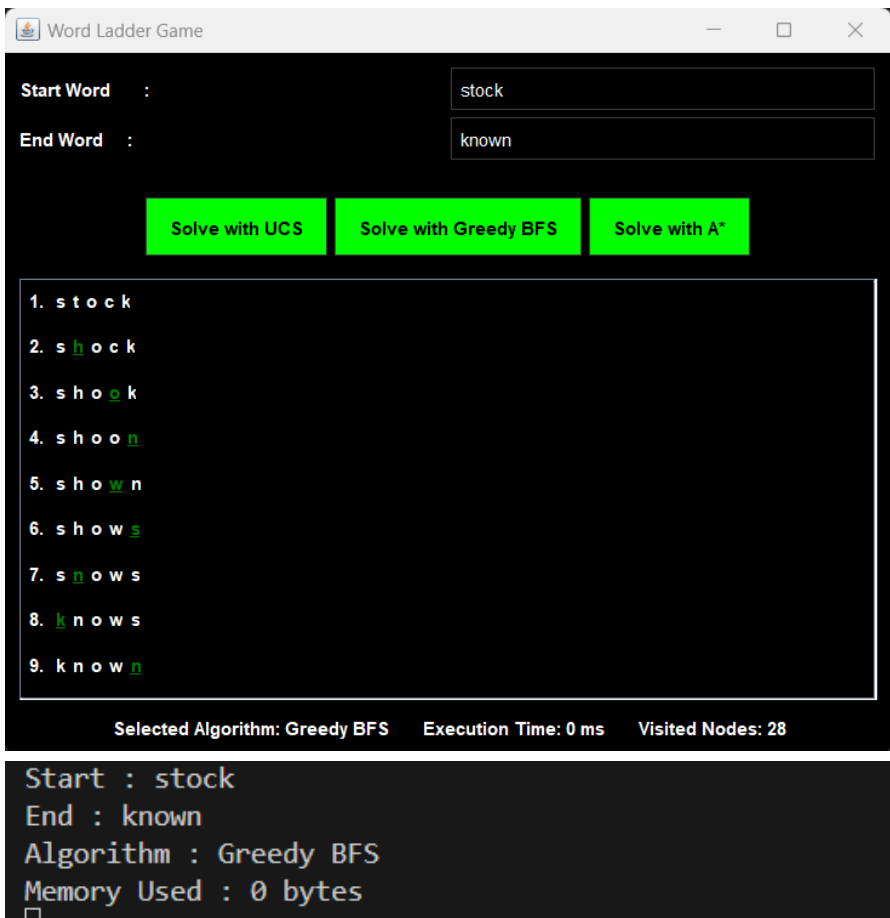
Start Word : sorry

End Word : thank

Solve with UCS Solve with Greedy BFS Solve with A*

1. s o r r y
2. s e r r y
3. t e r r y
4. t e a r y
5. t e a r s
6. t e a m s
7. t e a m s
8. t r a n s
9. t r a n k
10. t h a n k

Selected Algorithm: Greedy BFS Execution Time: 0 ms Visited Nodes: 13

| | |
|---|---|
| | <pre> Start : sorry End : thank Algorithm : Greedy BFS Memory Used : 0 bytes </pre> |
| 3 |  <p>The screenshot shows a web application titled "Word Ladder Game". It has two input fields: "Start Word" with the value "stock" and "End Word" with the value "known". Below these are three buttons: "Solve with UCS", "Solve with Greedy BFS" (which is highlighted in red), and "Solve with A*". A list of 9 words is displayed, showing the path from "stock" to "known" with one letter changing at each step: 1. stock, 2. shock, 3. shok, 4. shoon, 5. shown, 6. show, 7. snows, 8. knows, 9. know. At the bottom of the interface, it says "Selected Algorithm: Greedy BFS", "Execution Time: 0 ms", and "Visited Nodes: 28". Below the screenshot, the same text as in the first row is displayed.</p> <pre> Start : stock End : known Algorithm : Greedy BFS Memory Used : 0 bytes </pre> |

4

Word Ladder Game

Start Word :

End Word :

6. a n i l s
7. a i l s
8. a r i s
9. a e a s
10. u r e a s
11. u r e a l
12. u r i a l
13. t r i a l
14. t r i a d
15. t r e a d
16. b r e a d

Selected Algorithm: Greedy BFS Execution Time: 4 ms Visited Nodes: 40

Start : apple
End : bread
Algorithm : Greedy BFS
Memory Used : 0 bytes

5

Word Ladder Game

Start Word : stone

End Word : money

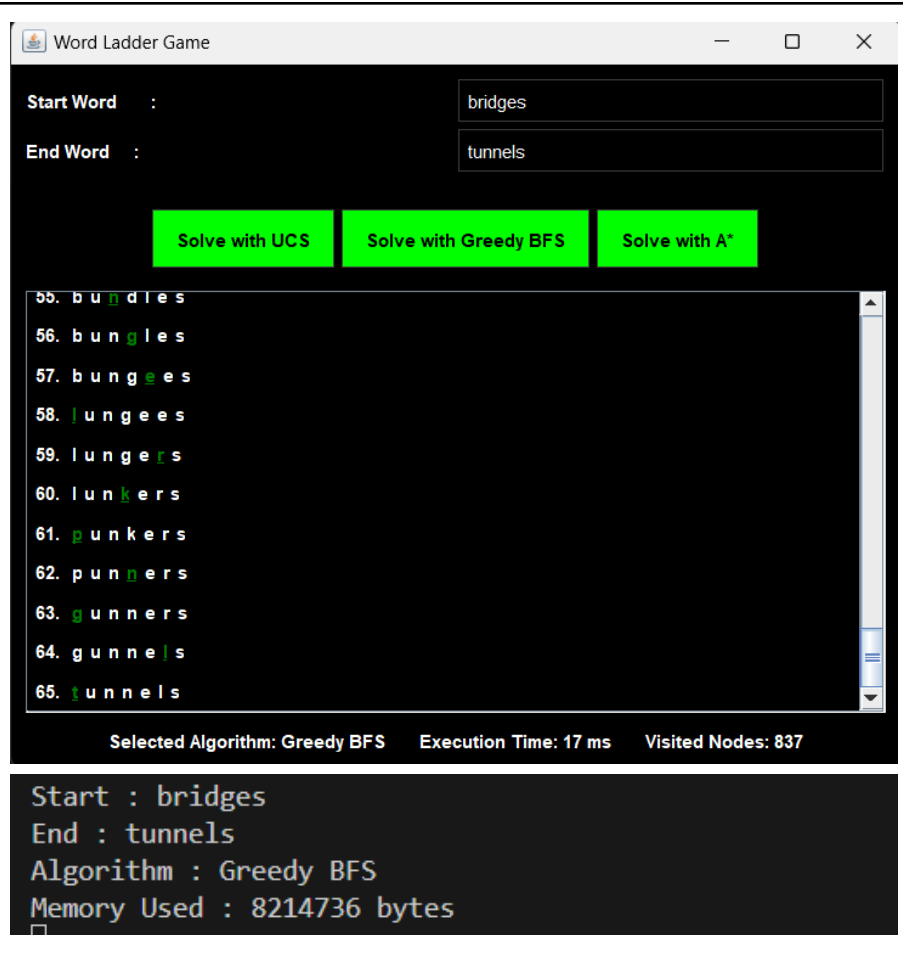
Solve with UCS Solve with Greedy BFS Solve with A*

14. s p i l l
15. s p i e l
16. s p i e r
17. s j i e r
18. s l y e r
19. f l y e r
20. f o y e r
21. t o y e r
22. t o n e r
23. t o n e y
24. m o n e y

Selected Algorithm: Greedy BFS Execution Time: 0 ms Visited Nodes: 143

Start : stone
End : money
Algorithm : Greedy BFS
Memory Used : 1046528 bytes

6



1.3. Pengujian Algoritma A* Search

Tabel 3. Pengujian Algoritma A*

| No. | Hasil Pengujian |
|-----|-----------------|
|-----|-----------------|

1

Word Ladder Game

Start Word : chicken

End Word : ducking

Solve with UCS Solve with Greedy BFS Solve with A*

14. b e a k i e r
15. p e a k i e r
16. p e r k i e r
17. j e r k i e r
18. j e r k i e s
19. j e r k i n s
20. j e r k i n g
21. p e r k i n g
22. p e c k i n g
23. d e c k i n g
24. d u c k i n g

Selected Algorithm: A* Execution Time: 68 ms Visited Nodes: 2316

Start : chicken
End : ducking
Algorithm : A*
Memory Used : 19787112 bytes

2

Word Ladder Game

Start Word : sorry

End Word : thank

Solve with UCS Solve with Greedy BFS Solve with A*

1. s o r r y
2. s e r r y
3. t e r r y
4. t e a r y
5. t e a r s
6. t e a m s
7. t r a m s
8. t r a n s
9. t r a n k
10. t h a n k

Selected Algorithm: A* Execution Time: 4 ms Visited Nodes: 280

| | |
|---|--|
| | <pre> Start : sorry End : thank Algorithm : A* Memory Used : 1875320 bytes </pre> |
| 3 | <div> <div>Word Ladder Game</div> <div> <div>Start Word : stock</div> <div>End Word : known</div> <div> <div>Solve with UCS</div> <div>Solve with Greedy BFS</div> <div>Solve with A*</div> </div> <div> <ol style="list-style-type: none"> 1. s t o c k 2. s t o <u>o</u> k 3. s t o o <u>p</u> 4. s t o <u>w</u> p 5. s t o w <u>s</u> 6. s <u>n</u> o w s 7. <u>k</u> n o w s 8. k n o w <u>n</u> </div> <div> <div>Selected Algorithm: A*</div> <div>Execution Time: 0 ms</div> <div>Visited Nodes: 170</div> </div> </div> <pre> Start : stock End : known Algorithm : A* Memory Used : 1184344 bytes </pre> </div> |

4

Word Ladder Game

Start Word :

End Word :

4. a n o l e
5. a n i l e
6. a n s e
7. a r i s e
8. p r i s e
9. p r e s e
10. p r e s
11. p r e s
12. b r e e s
13. b r e e d
14. b r e a d

Selected Algorithm: A* Execution Time: 3 ms Visited Nodes: 202

Start : apple
End : bread
Algorithm : A*
Memory Used : 1182344 bytes

5

Word Ladder Game

Start Word :

End Word :

1. s t o n e
2. s h o n e
3. s h o t e
4. s h o t s
5. s o o t s
6. s o o t y
7. s o o e y
8. h o o e y
9. h o n e y
10. m o n e y

Selected Algorithm: A* Execution Time: 20 ms Visited Nodes: 1020

| | |
|---|--|
| | <pre> Start : stone End : money Algorithm : A* Memory Used : 6291456 bytes □ </pre> |
| 6 | <div> <div>Word Ladder Game</div> <div> <div>Start Word : bridges</div> <div>End Word : tunnels</div> <div> <div>Solve with UCS</div> <div>Solve with Greedy BFS</div> <div>Solve with A*</div> </div> <div> <div>14. p e a t t i e r</div> <div>15. p e t t i e r</div> <div>16. p u t t i e r</div> <div>17. p u t t i e s</div> <div>18. p u t t e e s</div> <div>19. p u t t e r s</div> <div>20. p u n t e r s</div> <div>21. p u n n e r s</div> <div>22. g u n n e r s</div> <div>23. g u n n e l s</div> <div>24. t u n n e l s</div> </div> <div> <div>Selected Algorithm: A*</div> <div>Execution Time: 39 ms</div> <div>Visited Nodes: 2230</div> </div> </div> <pre> Start : bridges End : tunnels Algorithm : A* Memory Used : 18817744 bytes □ </pre> </div> |

1.4. Hasil Pengujian

| Percobaan | Parameter | <i>Uniform Cost Search (UCS)</i> | <i>Greedy Best First Search (GBFS)</i> | <i>A-Star (A*)</i> |
|--------------------|-----------------------------|----------------------------------|--|--------------------|
| Percobaan 1 | Jumlah Path | 24 | 96 | 24 |
| | Waktu Eksekusi (ms) | 148 | 7 | 68 |
| | Jumlah Node yang dikunjungi | 4102 | 469 | 2316 |
| | Memory Used | 35283848 | 4221960 | 19787112 |
| Percobaan 2 | Jumlah Path | 10 | 10 | 10 |
| | Waktu Eksekusi (ms) | 135 | 0 | 4 |
| | Jumlah Node yang dikunjungi | 5079 | 13 | 280 |
| | Memory Used | 32372072 | 0 | 1875320 |
| Percobaan 3 | Jumlah Path | 8 | 9 | 8 |
| | Waktu Eksekusi (ms) | 32 | 0 | 0 |
| | Jumlah Node yang dikunjungi | 1996 | 28 | 170 |
| | Memory Used | 12582696 | 0 | 1184344 |
| Percobaan 4 | Jumlah Path | 14 | 16 | 14 |
| | Waktu Eksekusi (ms) | 25 | 4 | 3 |
| | Jumlah Node yang dikunjungi | 1548 | 40 | 202 |
| | Memory Used | 9230096 | 0 | 1182344 |
| Percobaan 5 | Jumlah Path | 10 | 24 | 10 |
| | Waktu Eksekusi (ms) | 63 | 0 | 20 |

| | | | | |
|--------------------|-----------------------------|----------|---------|----------|
| | Jumlah Node yang dikunjungi | 5045 | 143 | 1020 |
| | Memory Used | 31721736 | 1046528 | 6291456 |
| Percobaan 6 | Jumlah Path | 24 | 65 | 24 |
| | Waktu Eksekusi (ms) | 74 | 17 | 39 |
| | Jumlah Node yang dikunjungi | 4109 | 837 | 2230 |
| | Memory Used | 35513152 | 8214736 | 18817744 |

4.2. Analisis

Dari hasil percobaan di atas, terlihat adanya perbedaan hasil, waktu, jumlah node yang dikunjungi, memory yang digunakan pada ketiga algoritma yang digunakan untuk memecah permainan Word Ladder, yaitu Uniform Cost Search (UCS), Greedy Best First Search (GBFS), dan A-Star (A*). Ada 2 fungsi yang digunakan untuk menentukan cost dari sebuah kata, yaitu $g(n)$ dan $f(n)$. Fungsi $g(n)$ merupakan jarak dari kata awal ke kata yang sedang di cek, dalam kasus ini adalah perbedaan huruf dari kata awal ke kata yang di cek. Fungsi $f(n)$ pada GBFS dengan fungsi $f(n)$ di A-star mempunyai perbedaan. Fungsi $f(n)$ pada GBFS didapatkan dari nilai heuristik ($h(n)$) yang menghitung banyak huruf yang berbeda dengan kata yang menjadi tujuan, sedangkan $f(n)$ untuk A-star merupakan hasil penjumlahan antara $g(n) + h(n)$. Di bawah ini akan dijelaskan secara lebih detail mengenai hasil analisa dari ketiga algoritma ini.

1. Hasil Analisa Uniform Cost Search (UCS)

Berikut adalah langkah-langkah algoritma UCS pada program ini. Pertama, node awal yang berisi kata awal akan dimasukan ke dalam queue prioritas, lalu akan dimulai pengambilan kata terdepan dalam queue prioritas. Selanjutnya, akan dicek apakah kata tersebut merupakan target atau bukan, jika iya, maka akan mengembalikan hasil, jika bukan, akan dilanjutkan dengan membangkitkan semua kemungkinan kata tetangga (kata yang berbeda 1 huruf), lalu tetangga yang belum dikunjungi akan dimasukkan ke dalam queue dengan cost sebesar $g(n)$ yakni yakni

jarak dari kata awal, yang bisa juga didapatkan dengan penambahan 1 pada node parentnya yang sedang di cek karena memiliki perbedaan kata 1 dengan node parentnya. Kata ini juga akan dimasukkan ke dalam set Visited yang menandakan node tersebut sudah dikunjungi , sehingga pada pengulangan selanjutnya jika ditemukan kata yang sama tidak dimasukkan lagi ke queue. Langkah di atas akan diulang-ulang sampai ditemukan kata tujuan atau sampai queue habis. Jika queue habis , maka akan menandakan tidak ada solusi untuk kata awal menuju kata tujuan..

Uniform Cost Search (UCS) cenderung memberikan solusi yang relatif optimal, ini dapat dilihat dari jumlah path yang dihasilkan saat menggunakan algoritma UCS ini. Dalam 6 kali percobaan , solusi yang didapatkan selalu lebih sedikit atau sama dengan algoritma lainnya. Hal ini menunjukkan bahwa algoritma UCS ini menghasilkan solusi yang relatif optimal. Namun ada beberapa kelemahan yang dimiliki oleh algoritma ini. Kelemahan tersebut adalah algoritma ini mempunyai waktu eksekusi lebih lama bila dibandingkan dengan algoritma lain , dapat dilihat dalam tabel hasil pengujian di atas, waktu eksekusi saat menggunakan algoritma UCS selalu lebih lama bila dibandingkan dengan kedua algoritma lainnya. Hal ini terjadi karena cara kerja algoritma UCS , yang selalu memprioritaskan node lain yang mempunyai cost terendah, dan cost ini dihitung dengan menghitung jarak dengan node awal. Pada percobaan kali ini, jarak antar kata ditentukan dengan menghitung perbedaan kata yang terdapat antara dua node, sehingga ini membuat jarak antar node parent dan anaknya akan selalu satu dan setiap anak memiliki jarak yang sama dengan node parentnya yaitu 1. Karena itu, ini membuat algoritma UCS dalam penerapan di permainan Word Ladder ini hampir sama dengan algoritma Breadth-First Search , dimana semua node dengan kedalaman 1 akan dicek terlebih dahulu , baru dilanjutkan ke node dengan kedalaman 2 , dan seterusnya sampai mencapai node yang ingin dicari. Hal ini berdampak ke waktu eksekusi karena pengecekan akan selalu dimulai dari kedalaman 1 , sehingga membuat jumlah node yang dikunjungi juga semakin banyak (dapat dilihat di tabel , jumlah node yang dikunjungi untuk algoritma UCS juga selalu lebih banyak bila dibandingkan dengan algoritma lainnya. Dari segi penggunaan memory, UCS juga menggunakan memory

yang lebih besar bila dibandingkan dengan kedua algoritma lainnya, hal ini karena jumlah node yang dikunjungi oleh algoritma UCS jauh lebih banyak.

2. Hasil Analisa Greedy Best First Search (GBFS)

Berikut adalah langkah-langkah algoritma GBFS pada program ini. Pertama, node awal yang berisi kata awal akan dimasukan ke dalam queue prioritas, lalu akan dimulai pengambilan kata terdepan dalam queue prioritas. Selanjutnya, akan dicek apakah kata tersebut merupakan target atau bukan, jika iya, maka akan mengembalikan hasil, jika bukan, akan dilanjutkan dengan membangkitkan semua kemungkinan kata tetangga (kata yang berbeda 1 huruf), lalu tetangga yang belum dikunjungi akan dimasukkan ke dalam queue dengan cost sebesar $f(n)$ yakni nilai heuristik dari kata tersebut (jarak ke kata tujuan), yang bisa juga didapatkan dengan mencari perbedaan dengan huruf kata tersebut dengan kata tujuan. Kata ini juga akan dimasukan ke dalam set Visited yang menandakan node tersebut sudah dikunjungi, sehingga pada pengulangan selanjutnya jika ditemukan kata yang sama tidak dimasukkan lagi ke queue. Langkah di atas akan diulang-ulang sampai ditemukan kata tujuan atau sampai queue habis. Jika queue habis, maka akan menandakan tidak ada solusi untuk kata awal menuju kata tujuan.

Pada percobaan yang sudah dilakukan, dapat dilihat bahwa algoritma GBFS ini mempunyai waktu eksekusi yang sangat cepat bila dibandingkan dengan kedua algoritma lainnya. Misalnya pada percobaan 1, waktu eksekusi saat menggunakan algoritma GBFS hanya sebesar 4 ms, sedangkan untuk UCS sebesar 37 ms dan untuk A* sebesar 19 ms. Nilai waktu tersebut tentunya sangat jauh berbeda dengan waktu kedua algoritma lainnya, sehingga bisa dikatakan GBFS mempunyai efisiensi yang paling tinggi. Hal ini karena cara kerja algoritma GBFS yang memprioritaskan node dengan jarak yang paling dekat dengan tujuan, sehingga akan lebih cepat untuk menemukan kata tujuan. Dapat dilihat juga dari jumlah node yang dikunjungi saat menggunakan algoritma GBFS ini selalu lebih sedikit jika dibandingkan dengan kedua algoritma lainnya. Namun, jika dilihat dari solusi yang dihasilkan oleh algoritma GBFS ini dalam 6 kali percobaan, 5 percobaan diantaranya mempunyai jumlah path yang sangat jauh berbeda dengan algoritma

UCS dan A* , misalnya pada percobaan 1 , dimana jumlah path dari solusi yang diperoleh dari algoritma GBFS ini menghasilkan 96 path , sedangkan algoritma UCS dan A* hanya menghasilkan 24 path. Hal ini menunjukkan bahwa pencarian solusi dengan menggunakan algoritma GBFS tidak menjamin menghasilkan solusi yang optimal. Tetapi, pada percobaan 2 , jumlah path yang dihasilkan oleh GBFS mempunyai jumlah yang sama dengan jumlah path dari algoritma UCS dan A* , hal ini menunjukkan bahwa hasil solusi dari GBFS tidak selalu menghasilkan solusi yang tidak optimal , namun terkadang juga bisa menghasilkan solusi yang optimal. Dari segi penggunaan memory, GBFS menggunakan memory yang paling kecil bila dibandingkan dengan kedua algoritma lainnya, hal ini karena jumlah node yang dikunjungi oleh algoritma GBFS yang jauh lebih sedikit.

3. Hasil Analisa Greedy A-star search (A*)

Berikut adalah langkah-langkah algoritma A* pada program ini. Pertama, node awal yang berisi kata awal akan dimasukan ke dalam queue prioritas, lalu akan dimulai pengambilan kata terdepan dalam queue prioritas .Selanjutnya , akan dicek apakah kata tersebut merupakan target atau bukan, jika iya, maka akan mengembalikan hasil, jika bukan , akan dilanjutkan dengan membangkitkan semua kemungkinan kata tetangga (kata yang berbeda 1 huruf) , lalu tetangga yang belum dikunjungi akan dimasukkan ke dalam queue dengan cost sebesar $f(n)$ yakni nilai heuristik dari kata tersebut (jarak ke kata tujuan) ditambah dengan $g(n)$ (jarak dengan node awal) , yang bisa juga didapatkan dengan mencari perbedaan dengan huruf kata tersebut dengan kata tujuan , lalu ditambah dengan cost node yang sedang dicek ditambah 1. Kata ini juga akan dimasukan ke dalam set Visited yang menandakan node tersebut sudah dikunjungi , sehingga pada pengulangan selanjutnya jika ditemukan kata yang sama tidak dimasukkan lagi ke queue. Langkah di atas akan diulang-ulang sampai ditemukan kata tujuan atau sampai queue habis. Jika queue habis , maka akan menandakan tidak ada solusi untuk kata awal menuju kata tujuan.

Dari hasil percobaan di atas , dapat dilihat bahwa solusi yang dihasilkan oleh algoritma A* search mempunyai jumlah path yang selalu sama dengan jumlah

path yang dihasilkan oleh algoritma UCS, sehingga bisa dibilang algoritma UCS dan A* menghasilkan solusi yang optimal. Namun terdapat perbedaan waktu eksekusi dan jumlah node yang dikunjungi antara UCS dan A*. Dimana dapat dilihat bahwa waktu eksekusi saat menjalankan A* selalu lebih cepat bila dibandingkan dengan UCS, namun masih tetap kalah dengan waktu eksekusi dari algoritma GBFS. Tetapi, hasil yang dihasilkan oleh A* ini merupakan hasil yang optimal, sedangkan hasil dari GBFS tidak menjamin optimal. Jumlah node yang dikunjungi oleh algoritma A* juga lebih sedikit daripada jumlah node dari algoritma UCS. Hal ini dapat membuktikan bahwa algoritma A* lebih efisien dari algoritma UCS. Penyebab algoritma A* bisa lebih efisien karena cara kerja algoritma A* yang memprioritaskan node dengan cost yang bergantung pada 2 aspek, yaitu jarak dari node awal dan jarak ke node akhir. Secara teoritis, memang algoritma A* pasti akan lebih efisien karena A* ini merupakan perpaduan antara GBFS dan UCS, sehingga tetap mencari solusi yang optimal seperti UCS, namun tetap memedulikan efisien seperti GBFS. Dari segi penggunaan memory, A* menggunakan memory yang lebih besar dari GBFS, tetapi lebih kecil dari UCS, hal ini karena jumlah node yang dikunjungi oleh algoritma A* yang lebih sedikit dari UCS namun lebih banyak dari GBFS.

Heuristik yang digunakan pada algoritma A* dikatakan admissible jika selalu memperkirakan biaya yang kurang dari atau sama dengan biaya sebenarnya untuk mencapai tujuan ($h(n) \leq h^*(n)$), di mana $h^*(n)$ adalah biaya sebenarnya atau biaya optimal untuk mencapai tujuan dari simpul n. Dengan kata lain, heuristik tersebut tidak pernah melebihi-lebihkan biaya untuk mencapai tujuan. Dalam kasus ini, heuristik ini menghitung jarak dari suatu kata ke kata tujuan dan jarak tersebut tidak pernah melebihi jarak sebenarnya, maka heuristik tersebut dapat dianggap admissible. Karena heuristics admissible, maka A* dijamin akan menemukan solusi yang optimal, A* tidak perlu untuk menjelajahi semua node pada kedalaman tertentu, tetapi juga memprioritaskan kata-kata yang lebih dekat dengan kata tujuan, sehingga pencarian dengan A* lebih efisien dari UCS.

4.3. Penjelasan Bonus

Bonus yang dikerjakan adalah GUI. GUI yang dibuat menggunakan Swing dan bahasa pemrograman Java. GUI yang dibuat ini menawarkan pengalaman pengguna yang interaktif dan menarik. Dengan latar belakang hitam dan teks putih yang kontras, serta tombol berwarna hijau yang mencolok, antarmuka terlihat menarik dan mudah dinavigasi. Pengguna dapat dengan mudah memasukkan kata awal dan kata akhir melalui input field yang tersedia, sambil memilih algoritma pencarian yang diinginkan dengan menggunakan tombol yang sesuai. Setelah pencarian selesai, hasilnya ditampilkan secara terstruktur dalam panel output, dengan setiap langkah dari solusi ditampilkan dengan jelas. Perubahan dari satu kata ke kata berikutnya ditandai dengan huruf yang dicetak tebal dan berwarna hijau, memudahkan pengguna untuk melihat bagaimana kata-kata berubah dari satu langkah ke langkah berikutnya. Informasi hasil yang lengkap, seperti algoritma yang digunakan, waktu eksekusi pencarian, dan jumlah node yang dikunjungi, juga ditampilkan di bagian bawah antarmuka, memberikan pengguna pemahaman yang lebih baik tentang proses pencarian yang dilakukan. Dengan responsif yang baik, GUI memberikan umpan balik yang cepat terhadap setiap tindakan pengguna, memastikan pengalaman pengguna yang lancar dan efisien. Dengan demikian, GUI ini tidak hanya fungsional, tetapi juga meningkatkan pengalaman pengguna secara keseluruhan.

BAB V

PENUTUP

5.1. Kesimpulan

Dari pengujian analisis yang dilakukan terhadap ketiga algoritma yang diaplikasikan pada permainan Word Ladder, yaitu Uniform Cost Search (UCS), Greedy Best First Search (GBFS), dan A-Star (A*), terdapat beberapa kesimpulan penting yang dapat diambil. Algoritma UCS meskipun cenderung lambat dalam waktu eksekusi dan membutuhkan penggunaan memori yang besar, namun secara konsisten menghasilkan solusi yang optimal ditandai dengan jumlah path yang minimal dalam semua percobaan. Sementara itu, GBFS menonjol karena efisiensi waktu yang cepat dan penggunaan memori yang minimal. Namun, GBFS tidak selalu menghasilkan solusi yang optimal, terkadang menghasilkan solusi dengan jumlah path yang jauh lebih banyak dibandingkan dengan UCS dan A*, hal ini menunjukkan bahwa efisiensinya mungkin berakibat pada pengorbanan keakuratan solusi. Di sisi lain, A* menawarkan keseimbangan yang baik antara kecepatan, penggunaan memori, dan keoptimalan solusi. Algoritma ini secara konsisten menghasilkan solusi dengan jumlah path yang optimal yang mirip dengan solusi dari UCS dan memiliki waktu eksekusi yang lebih baik daripada UCS tetapi tidak secepat GBFS. A* juga efisien dalam mengurangi jumlah node yang dijelajahi sehingga mengurangi penggunaan memori.

Oleh karena itu, dalam memilih algoritma yang sesuai untuk pengaplikasian word ladder solver ini, terdapat beberapa faktor yang harus dipertimbangkan yaitu keoptimalan, kecepatan, penggunaan memori, dan keakuratan. Jika kecepatan menjadi aspek yang krusial dan dipentingkan, maka GBFS bisa menjadi solusi. Jika diperlukan jaminan untuk memperoleh jalur terpendek yang mutlak, maka UCS bisa menjadi pilihan. Namun, jika keseimbangan terhadap semua aspek yang dipentingkan, maka A* bisa dijadikan pilihan.

5.2. Saran

Terdapat beberapa hal yang dapat diperhatikan untuk meningkatkan program ini lebih lanjut. Pertama, dapat dipertimbangkan untuk melakukan optimasi pada algoritma

yang digunakan. Mungkin ada alternatif-alternatif yang lebih efisien atau teknik optimasi tertentu yang dapat diterapkan untuk meningkatkan kinerja program secara keseluruhan. Selanjutnya, penambahan fitur-fitur tambahan dapat menjadi nilai tambah yang signifikan. Misalnya, dukungan untuk bahasa-bahasa tambahan, fitur penyimpanan permainan untuk melanjutkan di lain waktu, atau visualisasi tambahan untuk memperjelas hasil pencarian. Selain itu, perbaikan pada antarmuka pengguna juga dapat memberikan pengalaman pengguna yang lebih baik. Dan terakhir, meningkatkan keterbacaan kode dengan memperbaiki struktur kode, menambahkan dokumentasi yang lebih lengkap, dan meningkatkan keterbacaan kode secara keseluruhan juga dapat membantu dalam pemeliharaan dan pengembangan masa depan. Dengan mempertimbangkan saran-saran ini, program dapat terus ditingkatkan dalam berbagai aspek untuk memberikan pengalaman pengguna yang lebih baik dan meningkatkan fungsionalitas serta keamanan program secara keseluruhan..

DAFTAR REFERENSI

- Munir, Rinaldi. Penentuan rute (Route/Path Planning) - Bagian 1. Diakses 5 Mei 2024.
<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf>
- Munir, Rinaldi. Penentuan rute (Route/Path Planning) - Bagian 2. Diakses 5 Mei 2024.
<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf>

LAMPIRAN

Link Github: https://github.com/Benardo07/Tucil3_13522055

Tabel Kelayakan Program

| Poin | Ya | Tidak |
|---|----|-------|
| 1. Program berhasil dijalankan. | ✓ | |
| 2. Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma UCS | ✓ | |
| 3. Solusi yang diberikan pada algoritma UCS optimal | ✓ | |
| 4. Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma Greedy Best First Search | ✓ | |
| 5. Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma A* | ✓ | |
| 6. Solusi yang diberikan pada algoritma A* optimal | ✓ | |
| 7. [Bonus]: Program memiliki tampilan GUI | ✓ | |