

LAPORAN TUGAS KECIL 03
IF2211 STRATEGI ALGORITMA

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



Disusun oleh:

Albert

13522081

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

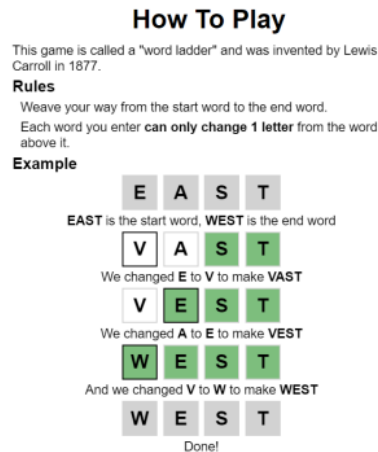
2024

DAFTAR ISI

DAFTAR ISI.....	1
BAB I	
DESKRIPSI MASALAH.....	2
BAB II	
LANDASAN TEORI.....	3
2.1 Algoritma Uniform Cost Search (UCS).....	3
2.2 Algoritma Greedy Best First Search.....	4
2.3 Algoritma A* Search.....	4
BAB III	
IMPLEMENTASI PROGRAM.....	6
3.1. Main.java.....	6
3.2. GUI.java.....	6
3.3. Loader.java.....	9
3.4. Node.java.....	10
3.5. SearchResult.java.....	11
3.6. WordLadderController.java.....	11
3.7. UCS1.java.....	12
3.8. GBFS1.java.....	13
3.9. AStar.java.....	14
BAB IV	
IMPLEMENTASI DAN PENGUJIAN.....	16
4.1. Pengujian.....	16
1.1. Pengujian Algoritma Uniform Cost Search.....	16
1.2. Pengujian Algoritma Greedy Best First Search.....	21
1.3. Pengujian Algoritma A* Search.....	27
4.2. Analisis.....	33
4.3. Penjelasan Bonus.....	36
BAB V	
PENUTUP.....	38
5.1. Kesimpulan.....	38
5.2. Saran.....	39
DAFTAR REFERENSI.....	40
LAMPIRAN.....	41

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. Pencarian solusi dari permainan Word Ladder 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), atau dikenal juga sebagai Dijkstra's Algorithm dalam kasus tertentu, merupakan metode pencarian yang digunakan untuk menemukan jalur terpendek dari titik awal ke titik tujuan tanpa mempertimbangkan informasi tambahan tentang keadaan node atau ruang pencarian pada graf berbobot berarah. Algoritma ini tidak menggunakan heuristik dalam proses pencariannya. Algoritma UCS beroperasi dengan cara memprioritaskan simpul yang memiliki '*cost*' atau biaya terendah dari titik awal hingga simpul tersebut. Setiap simpul yang dieksplorasi dari simpul awal diberikan prioritas berdasarkan biaya kumulatif dari simpul awal hingga simpul tersebut, bukan berdasarkan estimasi jarak hingga ke tujuan. Implementasi dari algoritma UCS biasanya menggunakan *priority queue* dimana setiap simpul yang dimasukkan akan diurutkan berdasarkan '*cost*' yang diperhitungkan.

Proses pencarian dalam UCS dalam permainan *word ladder* dimulai dengan memasukkan kata awal ke dalam *queue* dengan biaya nol. Setiap simpul yang dikeluarkan dari *queue* akan menghasilkan simpul baru, yang masing-masing mewakili kata hasil transformasi satu huruf dari kata sebelumnya. Setiap kata baru yang valid (yaitu, terdapat dalam kamus) akan ditambahkan kembali ke dalam *queue* dengan biaya yang diakumulasikan dari langkah sebelumnya ditambah satu. Proses ini terus berlanjut hingga kata tujuan ditemukan atau *queue* kosong, yang berarti tidak ada jalur yang ditemukan. Algoritma ini memastikan pencarian yang menyeluruh dan terstruktur, walaupun dengan biaya memori dan waktu yang bisa menjadi signifikan tergantung pada ukuran graf. Hal ini membuat UCS sangat efektif dalam menjamin bahwa jalur yang ditemukan adalah jalur dengan biaya total terendah, meskipun proses pencarian mungkin memerlukan waktu yang lebih lama dan memerlukan memori yang lebih besar karena harus menyimpan semua jalur yang mungkin dalam antrian prioritas.

2.2 Algoritma Greedy Best First Search

Algoritma *Greedy Best First Search* (GBFS) merupakan teknik pencarian heuristik yang menggunakan pendekatan '*greedy*' atau serakah untuk menavigasi graf. Algoritma ini memilih simpul yang paling menjanjikan untuk dieksplorasi atau yang lebih tampak untuk mendekati tujuan, berdasarkan estimasi biaya dari simpul tersebut hingga tujuan tanpa mempertimbangkan biaya total dari titik awal. Heuristik yang digunakan biasanya adalah jarak lurus (*straight-line distance*) dari simpul saat ini ke tujuan atau metrik lain yang mencerminkan perkiraan biaya minimum untuk mencapai tujuan. Dalam Word Ladder, GBFS mengevaluasi simpul berdasarkan seberapa dekat kata pada simpul tersebut dengan kata tujuan, yang diukur dengan jumlah huruf yang berbeda antara dua kata tersebut.

Langkah-langkah algoritma ini diawali dengan memasukkan kata awal ke dalam *PriorityQueue*, dengan huruf yang berbeda dari kata tujuan sebagai prioritas. Simpul yang paling menjanjikan yaitu yang paling mirip dengan kata tujuan dipilih terlebih dahulu untuk dieksplorasi. Setiap generasi kata baru yang valid akan diperiksa dan ditambahkan ke *queue* jika belum pernah dikunjungi. Walaupun cepat dalam mencapai solusi, GBFS dapat terjebak dalam *blind alley* atau jalan buntu, terutama ketika heuristik yang digunakan tidak menggambarkan kondisi sebenarnya dengan baik.

2.3 Algoritma A* Search

Algoritma A* (AStar) merupakan pengembangan dari algoritma pencarian UCS dan GBFS dengan menggabungkan keunggulan dari kedua algoritma tersebut. A* menggunakan fungsi penilaian $f(n) = g(n) + h(n)$, dimana $g(n)$ adalah biaya jalur dari titik awal ke simpul n , dan $h(n)$ adalah heuristik yang memperkirakan biaya dari simpul n ke tujuan. Heuristik yang digunakan dalam A* harus admissible (tidak boleh memperkirakan biaya yang lebih tinggi dari biaya sebenarnya) dan konsisten agar algoritma dapat menjamin pencarian jalur terpendek.

Dalam implementasi A* di dalam permainan *word ladder* ini, simpul awal dimasukkan ke dalam *PriorityQueue* dengan fungsi evaluasi yang mempertimbangkan kedua aspek biaya. Simpul dengan nilai $f(n)$ terkecil akan dieksplorasi terlebih dahulu. Setiap kali kata baru dihasilkan dan valid, akan dihitung biaya $g(n)$ baru dan diperbarui jika lebih kecil dari biaya sebelumnya untuk simpul tersebut. Proses ini berlanjut sampai kata tujuan ditemukan atau queue habis. Algoritma A* sangat efektif karena mengurangi jumlah simpul yang dieksplorasi dibandingkan dengan UCS

sambil menghindari risiko terjebak dalam jalan buntu seperti GBFS. A* dianggap sebagai salah satu algoritma pencarian jalur terbaik karena kemampuannya untuk secara efisien menemukan jalur terpendek dengan meminimalisir jumlah simpul yang dieksplorasi, berkat pendekatan optimalnya yang menggabungkan biaya aktual dan estimasi biaya ke tujuan. Algoritma ini sangat populer dalam berbagai aplikasi, termasuk dalam permainan, robotika, dan sistem navigasi.

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.*;

public class Main {
    public static void main(String[] args) {
        Gui.main(args);
    }
}
```

Di dalam file Main.java, terdapat class Main. *Class* Main memiliki method main yang akan memanggil fungsi jalannya program GUI.

3.2. Gui.java

Gui.java

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import javafx.scene.text.TextFlow;
import javafx.stage.Stage;
import java.util.*;

public class Gui extends Application {
    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Word Ladder Solver");

        // Main layout container
        VBox mainLayout = new VBox(20);
        mainLayout.setAlignment(Pos.TOP_CENTER);
        mainLayout.setPadding(new Insets(20));

        // Grid for user inputs
        GridPane inputGrid = new GridPane();
        inputGrid.setAlignment(Pos.CENTER);
        inputGrid.setVgap(10);
        inputGrid.setHgap(10);

        TextField startWordField = new TextField();
```

```

TextField endWordField = new TextField();
ComboBox<String> algorithmChoice = new ComboBox<>();
algorithmChoice.getItems().addAll("UCS", "GBFS", "AStar");
algorithmChoice.setValue("UCS");

inputGrid.add(new Label("Start Word:"), 0, 0);
inputGrid.add(startWordField, 1, 0);
inputGrid.add(new Label("End Word:"), 0, 1);
inputGrid.add(endWordField, 1, 1);
inputGrid.add(new Label("Algorithm:"), 0, 2);
inputGrid.add(algorithmChoice, 1, 2);

Button submitButton = new Button("Find Path");
HBox buttonBox = new HBox(submitButton);
buttonBox.setAlignment(Pos.CENTER);

// Labels for metrics
Label timeLabel = new Label("Time taken: -");
Label visitedLabel = new Label("Nodes visited: -");
HBox metricsBox = new HBox(20, timeLabel, visitedLabel);
metricsBox.setAlignment(Pos.CENTER);

ScrollPane scrollPane = new ScrollPane();
scrollPane.setFitToWidth(true);
VBox resultsBox = new VBox(10);
scrollPane.setContent(resultsBox);

mainLayout.getChildren().addAll(inputGrid, buttonBox, metricsBox, scrollPane);

submitButton.setOnAction(e → {
    String startWord = startWordField.getText().trim().toLowerCase();
    String endWord = endWordField.getText().trim().toLowerCase();

    resultsBox.getChildren().clear();
    timeLabel.setText("Time taken: -");
    visitedLabel.setText("Nodes visited: -");

    Set<String> dictionary = Loader.loadDictionary("src/dict.txt");
    if (dictionary.isEmpty() || !dictionary.contains(startWord) ||
!dictionary.contains(endWord)
        || startWord.length() ≠ endWord.length()) {
        resultsBox.getChildren().setAll(new Text("Invalid input or dictionary
issues."));
        return;
    }

    long startTime = System.currentTimeMillis();
    SearchResult result = WordLadderController.findPath(startWord, endWord,
dictionary,
        algorithmChoice.getValue());
    long endTime = System.currentTimeMillis();
    long timeTaken = endTime - startTime;

    resultsBox.getChildren().clear();
    timeLabel.setText(String.format("Time taken: %d ms", timeTaken));
    visitedLabel.setText(String.format("Nodes visited: %d",
result.getVisitedCount()));

    if (result.getPath() == null || result.getPath().isEmpty()) {
        resultsBox.getChildren().add(new Text("No path found."));
    } else {
        formatPathDisplay(result.getPath(), resultsBox);
    }
});
});

```



```

        Scene scene = new Scene(mainLayout, 600, 600);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    private void formatPathDisplay(List<String> path, VBox resultsBox) {
        if (path == null || path.isEmpty()) {
            return;
        }

        TextFlow initialWordFlow = createStyledTextFlow(path.get(0), "1. ");
        resultsBox.getChildren().add(initialWordFlow);

        for (int i = 1; i < path.size(); i++) {
            TextFlow textFlow = new TextFlow();
            textFlow.setStyle(
                "-fx-background-color: #f4f4f4; -fx-padding: 5px; -fx-border-color:
black; -fx-border-width: 2px;");

            String previousWord = path.get(i - 1);
            String currentWord = path.get(i);
            String indexText = (i + 1) + ". ";

            Text indexNode = new Text(indexText);
            indexNode.setStyle("-fx-font-weight: bold; -fx-font-size: 14px;");
            textFlow.getChildren().add(indexNode);

            for (int j = 0; j < currentWord.length(); j++) {
                char currentChar = currentWord.charAt(j);
                Text textNode = new Text(String.valueOf(currentChar));
                textNode.setStyle("-fx-font-size: 14px;");

                if (j ≥ previousWord.length() || currentChar ≠ previousWord.charAt(j)) {
                    textNode.setStyle("-fx-font-weight: bold; -fx-fill: red; -fx-font-size:
14px;");
                }

                textFlow.getChildren().add(textNode);
            }

            resultsBox.getChildren().add(textFlow);
        }
    }

    private TextFlow createStyledTextFlow(String word, String indexText) {
        TextFlow textFlow = new TextFlow();
        textFlow.setStyle(
            "-fx-background-color: #f4f4f4; -fx-padding: 5px; -fx-border-color: black;
-fx-border-width: 2px;");

        Text indexNode = new Text(indexText);
        indexNode.setStyle("-fx-font-weight: bold; -fx-font-size: 14px;");

        Text wordNode = new Text(word);
        wordNode.setStyle("-fx-font-size: 14px;");

        textFlow.getChildren().addAll(indexNode, wordNode);
        return textFlow;
    }

    public static void main(String[] args) {
        launch(args);
    }
}

```

Dalam file GUI.java terdapat *Class* Gui yang mengimplementasikan antarmuka pengguna grafis untuk visualisasi aplikasi *Word Ladder Solver* menggunakan JavaFX. Kelas ini akan memanggil fungsi dari Loader untuk membaca dan menyimpan data kamus. Kelas ini mengatur tampilan utama yang mencakup field untuk memasukkan kata awal dan akhir, pilihan algoritma, dan sebuah tombol untuk memulai pencarian. Ketika tombol diklik, akan dilakukan validasi masukan terlebih dahulu. Jika masukan valid, aplikasi akan memproses pencarian jalur kata dengan algoritma yang dipilih, mengukur waktu eksekusi, dan menampilkan jumlah simpul yang dikunjungi serta hasil pencarian dalam bentuk teks yang diformat di *ScrollPane*. Selain itu terdapat method *formatPathDisplay* dan *createStyledTextFlow* yang berfungsi untuk memberikan *styling* solusi yang ditampilkan.

3.3. Loader.java

Loader.java

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class Loader {
    public static Set<String> loadDictionary(String filePath) {
        try {
            List<String> lines = Files.readAllLines(Paths.get(filePath));
            return new HashSet<>(lines);
        } catch (Exception e) {
            System.err.println("Error reading dictionary file: " + e.getMessage());
            return new HashSet<>();
        }
    }

    public static void main(String[] args) {
        String filePath = "dict.txt";
        Set<String> dictionary = loadDictionary(filePath);
        System.out.println("Loaded " + dictionary.size() + " words.");
    }
}
```

Di dalam file Loader.java, terdapat *Class* Loader yang bertanggung jawab untuk memuat dan membaca file kamus berupa file txt yang berisi daftar kata yang digunakan dalam pencarian Word Ladder. Metode *loadDictionary* dalam kelas ini membaca file teks dari lokasi yang ditentukan, memasukkan setiap baris sebagai elemen dalam *Set<String>* untuk mengeliminasi duplikat dan memfasilitasi pencarian yang cepat. Kelas ini juga menangani pengecualian jika file tidak dapat dibaca.

3.4. Node.java

Node.java

```
import java.util.Set;
import java.util.ArrayList;
import java.util.List;

public class Node {
    private String word;
    private Node parent;
    private int cost;

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

    public String getWord() {
        return word;
    }

    public Node getParent() {
        return parent;
    }

    public int getCost() {
        return cost;
    }

    public void setParent(Node parent) {
        this.parent = parent;
    }

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

    public List<Node> generateSuccessors(Set<String> dictionary) {
        List<Node> successors = new ArrayList<>();
        char[] wordChars = word.toCharArray();
        for (int i = 0; i < word.length(); i++) {
            char originalChar = wordChars[i];
            for (char c = 'a'; c ≤ 'z'; c++) {
                if (c ≠ originalChar) {
                    wordChars[i] = c;
                    String newWord = new String(wordChars);
                    if (dictionary.contains(newWord)) {
                        successors.add(new Node(newWord, this, cost + 1));
                    }
                }
            }
            wordChars[i] = originalChar;
        }
        return successors;
    }
}
```

Dalam file Node.java terdapat Class Node yang digunakan untuk merepresentasikan setiap simpul dalam pencarian jalur Word Ladder. Setiap Node menyimpan kata, simpul induknya (untuk melacak jalur kembali ke sumber), dan biaya untuk mencapai simpul

tersebut dari kata awal. Terdapat metode setter dan getter untuk mengubah dan mengambil isi *attribute* dari kelas tersebut. Metode *generateSuccessors* menghasilkan semua kemungkinan kata selanjutnya yang valid dari kata saat ini dengan mengganti setiap huruf secara berurutan dan memeriksa keberadaan kata baru dalam kamus.

3.5. SearchResult.java

SearchResult.java

```
import java.util.*;

public class SearchResult {
    private List<String> path;
    private int visitedCount;

    public SearchResult(List<String> path, int visitedCount) {
        this.path = path;
        this.visitedCount = visitedCount;
    }

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

    public int getVisitedCount() {
        return visitedCount;
    }
}
```

Dalam file SearchResult.java terdapat Class SearchResult yang digunakan untuk menyimpan hasil dari pencarian Word Ladder. Karena searchResult hanya digunakan sebagai template solusi maka method yang ada di kelas ini hanya kontruktor dan getter saja. Kelas ini menyimpan jalur hasil pencarian dalam bentuk List<String> dan jumlah node yang dikunjungi selama pencarian. Ini memungkinkan data ini mudah diakses dan ditampilkan dalam GUI.

3.6. WordLadderController.java

WordLadderController.java

```
import java.util.*;

public class WordLadderController {
    public static SearchResult findPath(String startWord, String endWord, Set<String>
dictionary, String algorithm) {
        switch (algorithm.toLowerCase()) {
            case "ucs":
                return UCS1.findPath(startWord, endWord, dictionary);
            case "gbfs":
                return GBFS1.findPath(startWord, endWord, dictionary);
            case "astar":
                return AStar.findPath(startWord, endWord, dictionary);
            default:
                return new SearchResult(null, 0);
        }
    }
}
```

Dalam file `WordLadderController.java` terdapat Class `WordLadderController` bertindak sebagai pengendali yang menghubungkan GUI dengan logika pencarian Word Ladder. Kelas ini menangani logika untuk memilih dan menjalankan algoritma pencarian yang sesuai berdasarkan pilihan pengguna, dan mengembalikan hasil pencarian melalui `SearchResult`.

3.7. UCS1.java

UCS1.java

```
import java.util.*;

public class UCS1 {
    public static SearchResult findPath(String startWord, String endWord, Set<String>
dictionary) {
        if (startWord.equals(endWord)) {
            return new SearchResult(null, 0);
        }
        Queue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(Node::getCost));
        Set<String> visited = new HashSet<>();
        int visitedCount = 0;
        queue.add(new Node(startWord, null, 0));

        while (!queue.isEmpty()) {
            Node current = queue.poll();
            if (visited.add(current.getWord())) {
                visitedCount++;
                if (current.getWord().equals(endWord)) {
                    return new SearchResult(constructPath(current), visitedCount);
                }
                for (Node successor : current.generateSuccessors(dictionary)) {
                    if (!visited.contains(successor.getWord())) {
                        queue.add(successor);
                    }
                }
            }
        }
        return new SearchResult(null, visitedCount); // No path found
    }

    private static List<String> constructPath(Node node) {
        List<String> path = new ArrayList<>();
        while (node != null) {
            path.add(0, node.getWord());
            node = node.getParent();
        }
        return path;
    }
}
```

Di dalam file `UCS1.java`, terdapat class `UCS1` yang memiliki metode utama yaitu *findpath*, yang berfungsi untuk mencari jalur terpendek dari kata awal ke kata tujuan menggunakan algoritma *Uniform Cost Search* (UCS). Algoritma ini menggunakan *PriorityQueue* yang diurutkan berdasarkan biaya akumulatif (*cost*) dari node, memastikan bahwa *node* dengan biaya terkecil selalu diproses terlebih dahulu. Selama proses pencarian, setiap *node* yang dieksplorasi ditambahkan ke dalam set *visited* untuk menghindari pengulangan dan pengolahan berlebih. Jumlah node yang dikunjungi, yang ditandai dalam

variabel *visitedCount*, memberikan ukuran efisiensi dan kompleksitas pencarian. Class UCS1 juga memiliki fungsi *constructpath* untuk mencari semua *parent* dari *node* yang ditemukan. Hasil pencarian berupa jalur kata dan jumlah kata yang dikunjungi.

3.8. GBFS1.java

GBFS1.java

```
import java.util.*;

public class GBFS1 {
    public static SearchResult findPath(String startWord, String endWord, Set<String>
dictionary) {
        if (startWord.equals(endWord)) {
            return new SearchResult(null, 0);
        }
        Queue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(node →
heuristic(node, endWord)));
        Set<String> visited = new HashSet<>();
        int visitedCount = 0;
        queue.add(new Node(startWord, null, 0));
        visited.add(startWord);
        while (!queue.isEmpty()) {
            Node current = queue.poll();
            if (current.getWord().equals(endWord)) {
                return new SearchResult(constructPath(current), visitedCount);
            }

            visitedCount++;
            List<Node> successors = current.generateSuccessors(dictionary);
            for (Node successor : successors) {
                if (!visited.contains(successor.getWord())) {
                    visited.add(successor.getWord());
                    queue.add(successor);
                }
            }
        }
        return new SearchResult(null, visitedCount); // No path found
    }

    private static List<String> constructPath(Node node) {
        List<String> path = new ArrayList<>();
        while (node != null) {
            path.add(0, node.getWord());
            node = node.getParent();
        }
        return path;
    }

    private static int heuristic(Node node, String endWord) {
        int mismatchCount = 0;
        for (int i = 0; i < node.getWord().length(); i++) {
            if (node.getWord().charAt(i) != endWord.charAt(i)) {
                mismatchCount++;
            }
        }
        return mismatchCount;
    }
}
```

Di dalam file GBFS1.java, terdapat class GBFS1 mengimplementasikan algoritma *Greedy Best First Search*, yang berfokus pada pencarian jalur yang tampak paling menjanjikan menuju tujuan menggunakan heuristik. Class GBFS1 memiliki satu method pencarian yaitu *findpath* yang berfungsi untuk mencari jalur terpendek dari kata awal ke kata tujuan menggunakan algoritma *Greedy Best First Search* (GBFS). Algoritma ini juga menggunakan *PriorityQueue*, tetapi node diurutkan berdasarkan estimasi heuristik dari node ke tujuan, bukan biaya yang telah dikeluarkan. Heuristik dihitung sebagai jumlah huruf yang tidak cocok antara kata di node saat ini dan kata tujuan. HashSet digunakan untuk melacak setiap kata yang sudah dikunjungi. Setiap kata baru yang dihasilkan dan belum dikunjungi ditambahkan ke queue dan visited. Variabel *visitedCount* juga *diupdate* setiap kali node baru ditambahkan ke visited. Proses ini berlanjut hingga kata tujuan ditemukan atau queue kosong. Class GBFS juga memiliki fungsi *constructpath* untuk mencari semua *parent* dari *node* yang ditemukan. Hasil pencarian termasuk jalur kata dan jumlah kata yang dikunjungi.

3.9. AStar.java

AStar.java

```
import java.util.*;

public class AStar {
    public static SearchResult findPath(String startWord, String endWord, Set<String>
dictionary) {
        if (startWord.equals(endWord)) {
            return new SearchResult(null, 0);
        }
        Queue<Node> queue = new PriorityQueue<
(Comparator.comparingInt(node -> node.getCost() + heuristic(node,
endWord)));
        Set<String> visited = new HashSet<>();
        int visitedCount = 0;
        queue.add(new Node(startWord, null, 0));

        while (!queue.isEmpty()) {
            Node current = queue.poll();
            if (visited.add(current.getWord())) {
                visitedCount++;
                if (current.getWord().equals(endWord)) {
                    return new SearchResult(constructPath(current), visitedCount);
                }
                for (Node successor : current.generateSuccessors(dictionary)) {
                    int newCost = current.getCost() + 1;
                    if (!visited.contains(successor.getWord()) || newCost <
successor.getCost()) {
                        successor.setParent(current);
                        successor.setCost(newCost);
                        queue.add(successor);
                    }
                }
            }
        }
        return new SearchResult(null, visitedCount); // No path found
    }

    private static List<String> constructPath(Node node) {
        List<String> path = new ArrayList<>();
        while (node != null) {
```

```

        path.add(0, node.getWord());
        node = node.getParent();
    }
    return path;
}

private static int heuristic(Node node, String endWord) {
    int mismatchCount = 0;
    for (int i = 0; i < node.getWord().length(); i++) {
        if (node.getWord().charAt(i) != endWord.charAt(i)) {
            mismatchCount++;
        }
    }
    return mismatchCount;
}
}

```

Di dalam file Astar.java, terdapat class Astar yang mengimplementasikan algoritma A*, yang menggabungkan pendekatan UCS dan GBFS dengan menggunakan fungsi penilaian $f(n) = g(n) + h(n)$, dimana $g(n)$ adalah biaya aktual dari kata awal ke node saat ini dan $h(n)$ adalah heuristik dari node saat ini ke tujuan. Algoritma ini juga memanfaatkan *PriorityQueue* yang mengurutkan *node* berdasarkan nilai $f(n)$ yang lebih rendah. Seperti pada algoritma UCS dan GBFS, *node* yang telah dikunjungi ditambahkan ke dalam set *visited* untuk menghindari proses ulang, dengan *visitedCount* mencatat jumlah total node yang telah diproses. Setiap kata baru yang dihasilkan dan belum dikunjungi ditambahkan ke *queue* dan *visited*. Variabel *visitedCount* juga *diupdate* setiap kali *node* baru ditambahkan ke *visited*. Metode ini tidak hanya mengarah pada jalur yang efisien tetapi juga mengoptimalkan pencarian dengan menghindari jalur yang kurang menjanjikan berdasarkan estimasi biaya total dari awal hingga tujuan. Proses ini berlanjut hingga kata tujuan ditemukan atau *queue* kosong. Hasil pencarian berupa jalur kata dan jumlah kata yang dikunjungi.

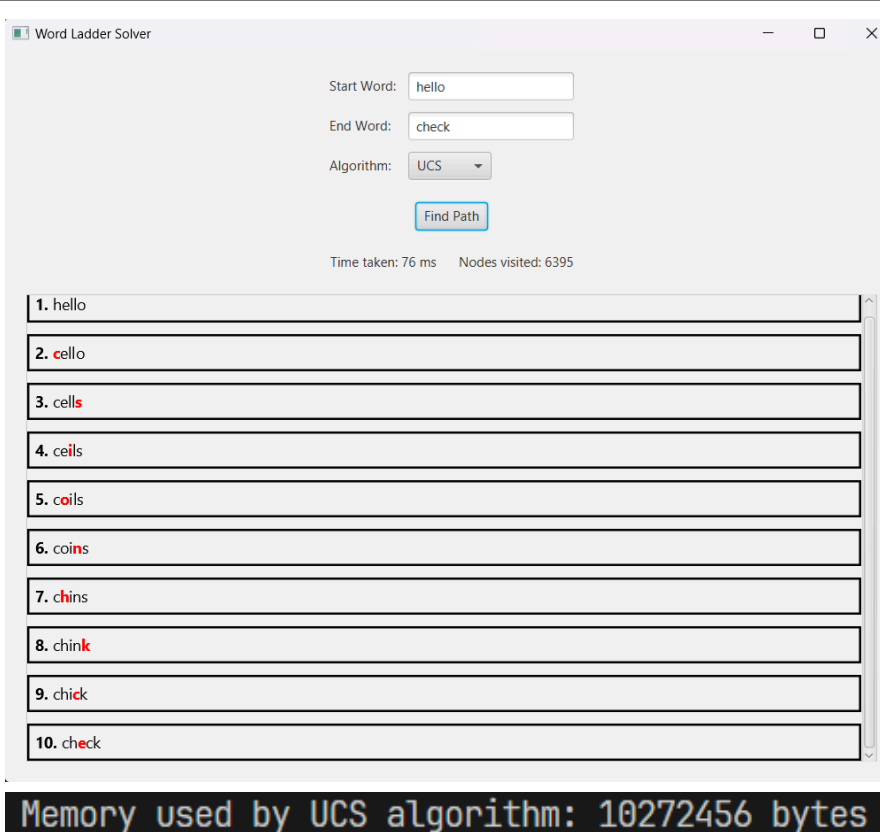
BAB IV

IMPLEMENTASI DAN PENGUJIAN

4.1. Pengujian

1.1. Pengujian Algoritma Uniform Cost Search

Tabel 1. Pengujian Algoritma Uniform Cost Search

No.	Hasil Pengujian
1	

2

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Time taken: 49 ms Nodes visited: 2518

1. hello
2. hell**s**
3. hea**l**s
4. wea**l**s
5. wea**l**d
6. wo**a**ld
7. wo**r**ld

Memory used by UCS algorithm: 16777216 bytes

3

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Time taken: 45 ms Nodes visited: 2129

1. work
2. wa**r**k
3. wa**r**n
4. wa**i**n
5. wa**i**r
6. wh**i**r

Memory used by UCS algorithm: 12172592 bytes

4

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Time taken: 31 ms Nodes visited: 5

1. whine
2. while

Memory used by UCS algorithm: 571912 bytes

5

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Time taken: 44 ms Nodes visited: 3173

1. mood
2. moon
3. poon
4. peon
5. pein
6. rein
7. ruin

Memory used by UCS algorithm: 17306552 bytes

6

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Time taken: 81 ms Nodes visited: 8288

1. charge
2. change
3. changs
4. chants
5. chints
6. chines
7. chined
8. coined
9. coiner
10. conner
11. conger
12. conges
13. conies
14. conins
15. coning
16. coming
17. homing
18. hominy
19. homily
20. homely

	<div> <div>21. comely</div> <div>22. comed^y</div> <div>23. comed^o</div> </div> <div>Memory used by UCS algorithm: 31942352 bytes</div>
7	<div> <div>Word Ladder Solver</div> <div> <div>Start Word: hell</div> <div>End Word: norm</div> <div>Algorithm: UCS</div> <div>Find Path</div> </div> <div>Time taken: 49 ms Nodes visited: 2872</div> <div> <div>1. hell</div> <div>2. her^l</div> <div>3. her^m</div> <div>4. d^erm</div> <div>5. d^orm</div> <div>6. n^orm</div> </div> </div> <div>Memory used by UCS algorithm: 15730688 bytes</div>
8	<div> <div>Word Ladder Solver</div> <div> <div>Start Word: zoom</div> <div>End Word: norm</div> <div>Algorithm: UCS</div> <div>Find Path</div> </div> <div>Time taken: 31 ms Nodes visited: 124</div> <div> <div>1. zoom</div> <div>2. d^oom</div> <div>3. d^or^m</div> <div>4. n^orm</div> </div> </div> <div>Memory used by UCS algorithm: 1257264 bytes</div>

1.2. Pengujian Algoritma Greedy Best First Search

Tabel 2. Pengujian Algoritma Greedy Best First Search

No.	Hasil Pengujian
1	<div><div>Word Ladder Solver</div><div>Start Word: <input type="text" value="hello"/></div><div>End Word: <input type="text" value="check"/></div><div>Algorithm: <input type="text" value="GBFS"/></div><div>Find Path</div><div>Time taken: 34 ms Nodes visited: 42</div><div><div>1. hello</div><div>2. cello</div><div>3. cells</div><div>4. culls</div><div>5. curls</div><div>6. curds</div><div>7. cards</div><div>8. caids</div></div><div><div>9. cains</div><div>10. chins</div><div>11. chics</div><div>12. chick</div><div>13. check</div></div><div>Memory used by GBFS algorithm: 811728 bytes</div></div>

2

Word Ladder Solver

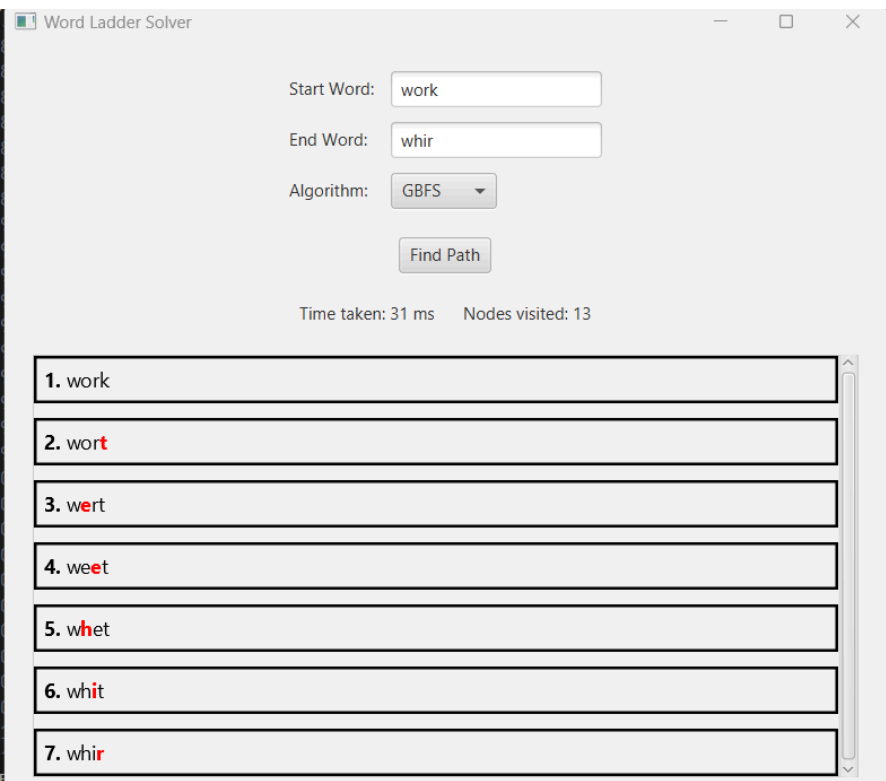
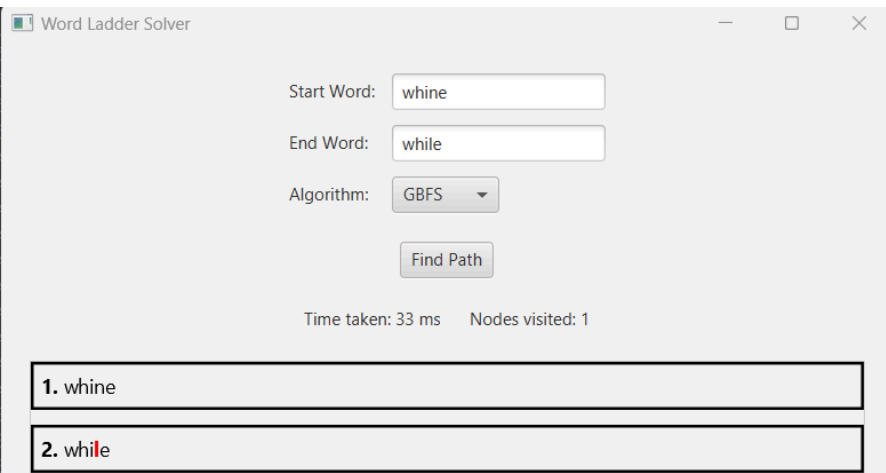
Start Word:

End Word:

Algorithm:

Time taken: 32 ms Nodes visited: 389

1. hello
2. hollo
3. holly
4. colly
5. cooly
6. wooly
7. woody
8. wordy
9. wormy
10. worms
11. warms
12. wares
13. wared
14. wired
15. wiled
16. wiles
17. wills
18. wells
19. weals
20. weald
21. wold
22. world

	<p>Memory used by GBFS algorithm: 2716856 bytes</p>
3	 <p>Word Ladder Solver</p> <p>Start Word: <input type="text" value="work"/></p> <p>End Word: <input type="text" value="whir"/></p> <p>Algorithm: <input type="button" value="GBFS"/></p> <p><input type="button" value="Find Path"/></p> <p>Time taken: 31 ms Nodes visited: 13</p> <ol style="list-style-type: none"> 1. work 2. wort 3. wert 4. weet 5. whet 6. whit 7. whir <p>Memory used by GBFS algorithm: 527256 bytes</p>
4	 <p>Word Ladder Solver</p> <p>Start Word: <input type="text" value="whine"/></p> <p>End Word: <input type="text" value="while"/></p> <p>Algorithm: <input type="button" value="GBFS"/></p> <p><input type="button" value="Find Path"/></p> <p>Time taken: 33 ms Nodes visited: 1</p> <ol style="list-style-type: none"> 1. whine 2. while <p>Memory used by GBFS algorithm: 531056 bytes</p>

5

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Time taken: 32 ms Nodes visited: 13

1. mood
2. moon
3. coon
4. coin
5. cain
6. rain
7. ruin

Memory used by GBFS algorithm: 575544 bytes

6

Word Ladder Solver

Start Word:

End Word:

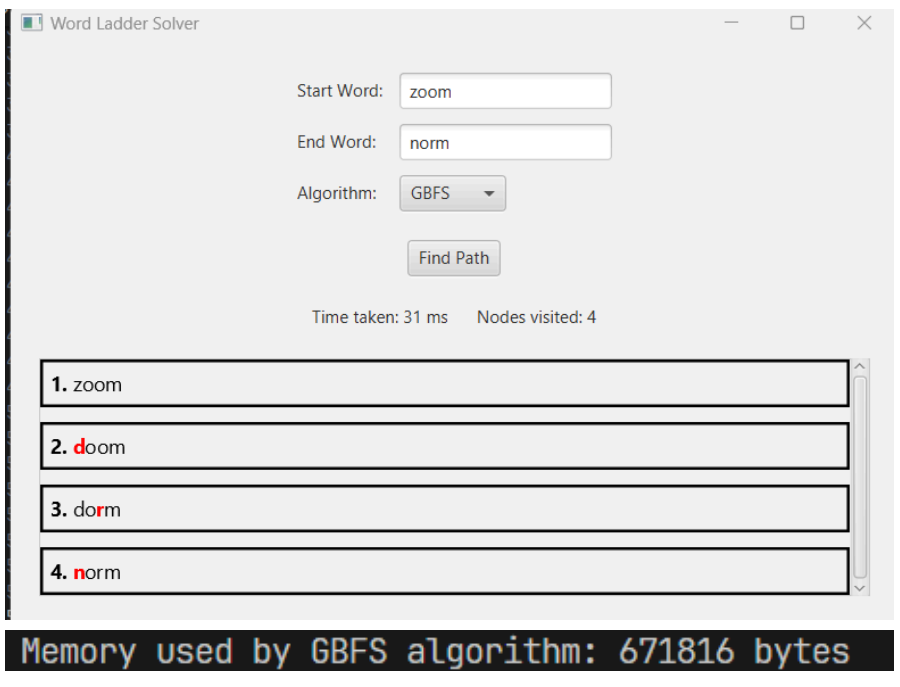
Algorithm:

Time taken: 32 ms Nodes visited: 164

1. charge
2. change
3. changs
4. clangs
5. clanks
6. clacks
7. clucks
8. crucks
9. cruces
10. cruses
11. causes
12. carses
13. corses
14. copses
15. copies
16. conies
17. conins
18. coning
19. coming
20. homing

	<div> <div>21. hominy</div> <div>22. homily</div> <div>23. homely</div> <div>24. comely</div> <div>25. comedy</div> <div>26. comedo</div> </div> <div>Memory used by GBFS algorithm: 1886160 bytes</div>
7	<div> <div>Word Ladder Solver</div> <div> <div>Start Word: hell</div> <div>End Word: norm</div> <div>Algorithm: GBFS</div> <div>Find Path</div> </div> <div>Time taken: 47 ms Nodes visited: 11</div> <div> <div>1. hell</div> <div>2. herl</div> <div>3. herm</div> <div>4. harm</div> <div>5. farm</div> <div>6. form</div> <div>7. norm</div> </div> <div>Memory used by GBFS algorithm: 786776 bytes</div> </div>

8



The screenshot shows a window titled "Word Ladder Solver". It has input fields for "Start Word" (zoom) and "End Word" (norm), and a dropdown menu for "Algorithm" set to "GBFS". A "Find Path" button is below. Below the button, it says "Time taken: 31 ms" and "Nodes visited: 4". A list of four words is shown in a scrollable box: 1. zoom, 2. doom (with 'd' in red), 3. dorm (with 'd' in red), and 4. norm (with 'n' in red). At the bottom, a black bar contains the text "Memory used by GBFS algorithm: 671816 bytes" in a monospace font.

1.3. Pengujian Algoritma A* Search

Tabel 3. Pengujian Algoritma A*

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

1

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Time taken: 48 ms Nodes visited: 569

1. hello
2. cello
3. cell*s*
4. ce*il*s
5. co*il*s
6. co*in*s
7. ch*in*s
8. chi*c*s
9. chick
10. check

Memory used by ASTAR algorithm: 4194304 bytes

2

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Time taken: 39 ms Nodes visited: 46

1. hello
2. hell**s**
3. hea**l**s
4. wea**l**s
5. wea**l**d
6. wo**a**ld
7. wo**r**ld

Memory used by ASTAR algorithm: 631832 bytes

3

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Time taken: 31 ms Nodes visited: 28

1. work
2. wor**t**
3. wa**r**t
4. wa**i**t
5. wa**i**r
6. wh**i**r

Memory used by ASTAR algorithm: 539256 bytes

4

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Time taken: 32 ms Nodes visited: 2

1. whine
2. while

Memory used by ASTAR algorithm: 527256 bytes

5

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Time taken: 32 ms Nodes visited: 121

1. mood
2. moon
3. loon
4. loin
5. lain
6. rain
7. ruin

Memory used by ASTAR algorithm: 1144320 bytes

6

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Time taken: 98 ms Nodes visited: 7000

1. charge
2. change
3. changs
4. chants
5. chints
6. chines
7. chined
8. coined
9. coiner
10. conner
11. conger
12. conges
13. conies
14. conins
15. coning
16. honing
17. homing
18. hominy
19. homily
20. homely

	<div> <div>21. comely</div> <div>22. comed^y</div> <div>23. comed^o</div> </div> <div>Memory used by ASTAR algorithm: 20356560 bytes</div>
7	<div> <div>Word Ladder Solver</div> <div> <div>Start Word: hell</div> <div>End Word: norm</div> <div>Algorithm: AStar</div> <div>Find Path</div> </div> <div>Time taken: 38 ms Nodes visited: 41</div> <div> <div>1. hell</div> <div>2. her^l</div> <div>3. her^m</div> <div>4. d^erm</div> <div>5. d^orm</div> <div>6. n^orm</div> </div> </div> <div>Memory used by ASTAR algorithm: 789432 bytes</div>
8	<div> <div>Word Ladder Solver</div> <div> <div>Start Word: zoom</div> <div>End Word: norm</div> <div>Algorithm: AStar</div> <div>Find Path</div> </div> <div>Time taken: 34 ms Nodes visited: 8</div> <div> <div>1. zoom</div> <div>2. d^oom</div> <div>3. d^orm</div> <div>4. n^orm</div> </div> </div> <div>Memory used by ASTAR algorithm: 784128 bytes</div>

4.2. Analisis

Dari hasil percobaan, data yang didapatkan akan disajikan dan dalam tabel berikut:

Parameter	Algoritma	TC 1	TC 2	TC 3	TC 4	TC 5	TC 6	TC 7
Waktu eksekusi (ms)	UCS	76	49	45	31	44	81	49
	GBFS	34	32	31	33	32	32	47
	A*	48	39	31	32	32	98	38
Panjang jalur yang didapat	UCS	10	7	6	2	7	23	6
	GBFS	13	22	7	2	7	26	7
	A*	10	7	6	2	7	23	6
Jumlah kata yang dicek	UCS	6395	2518	2129	5	3173	8288	2872
	GBFS	42	389	13	1	13	164	11
	A*	569	46	28	2	121	7000	41
Memori yang digunakan	UCS	10272456	16777216	12172592	571912	17306552	31942352	15730688
	GBFS	811728	2716586	527256	531056	575544	1886160	786776
	A*	4194304	631832	539256	527256	1144320	20356560	789432

Tabel 4.2.1 Data hasil percobaan

Dari hasil eksperimen yang dilakukan pada tiga algoritma pencarian dalam konteks *Word Ladder*, data yang dikumpulkan memberikan wawasan mendalam mengenai kinerja masing-masing algoritma berdasarkan beberapa parameter kunci: waktu eksekusi, panjang jalur yang ditemukan, jumlah kata yang dicek, dan penggunaan memori.

Waktu eksekusi yang tercatat menunjukkan perbedaan yang signifikan antar algoritma. *Uniform Cost Search* (UCS) cenderung memberikan waktu yang lebih lama dibandingkan dengan AStar, yang mencerminkan efisiensi penggunaan heuristik oleh AStar. Hal ini dapat dilihat dari lima dari tujuh percobaan di mana UCS memakan waktu paling lama. Ini menunjukkan bahwa UCS, meskipun akurat dalam menemukan jalur terpendek, memerlukan waktu yang lebih lama karena metode pencarian yang menyeluruh tanpa panduan heuristik untuk memprioritaskan jalur. Di sisi lain, *Greedy Best First Search* (GBFS) menunjukkan waktu eksekusi yang paling bervariasi, yang bisa sangat cepat dalam kondisi ideal ketika heuristik yang digunakan sangat akurat dalam memprediksi jarak ke tujuan. Hal

ini menjadikan GBFS sebagai algoritma yang efektif dalam kondisi spesifik, namun risikonya adalah ketidakpastian dalam pencapaian solusi yang optimal.

Dari segi panjang jalur yang ditemukan, UCS dan AStar sering kali memberikan hasil yang serupa, mengindikasikan bahwa keduanya cenderung mencapai solusi yang optimal dengan mempertimbangkan biaya kumulatif dan jarak yang ditempuh. GBFS, dengan fokus utamanya pada heuristik yang memperkirakan jarak langsung ke tujuan, cenderung menemukan jalur yang lebih panjang dalam beberapa kasus, seperti pada percobaan ketiga di mana GBFS memiliki waktu eksekusi yang sangat cepat tetapi dengan panjang jalur yang tidak optimal. Hal ini menegaskan bahwa GBFS mengorbankan akurasi demi kecepatan, menghasilkan solusi yang mungkin tidak efisien.

Parameter Jumlah Kata yang Dicek menggambarkan efisiensi algoritma dalam mengeksplorasi ruang solusi. UCS mengecek lebih banyak kata dibandingkan dengan AStar, menandakan sifat eksploratifnya yang luas tanpa panduan heuristik. Sebaliknya, AStar menunjukkan keefektifan yang tinggi dengan jumlah kata yang dicek jauh lebih sedikit, berkat integrasi heuristik yang memfokuskan pencarian hanya pada area yang paling berpotensi. GBFS, meskipun bisa sangat efisien dalam kasus-kasus dengan heuristik yang akurat, menunjukkan fluktuasi dalam jumlah kata yang dicek, yang mencerminkan variabilitasnya dalam kinerja tergantung pada keakuratan heuristik yang digunakan.

Analisis penggunaan memori merupakan indikator penting dalam mengukur efisiensi algoritma. Dari data yang diperoleh, terlihat bahwa algoritma *Uniform Cost Search* (UCS) cenderung menggunakan memori yang lebih banyak dibandingkan dengan dua algoritma lainnya, yaitu *Greedy Best-First Search* (GBFS) dan A*. Hal ini disebabkan oleh karakteristik UCS yang menyimpan banyak *node* dalam antrian untuk diproses, mirip dengan pendekatan yang diambil oleh algoritma *Breadth-First Search* (BFS). Dalam algoritma UCS, setiap *node* pada setiap tingkat dieksplorasi tanpa memprioritaskan kedekatan *node* tersebut terhadap solusi yang diharapkan. Ini berarti UCS harus mempertimbangkan setiap kemungkinan perpindahan tanpa memilah berdasarkan estimasi biaya, sehingga memori yang digunakan meningkat secara eksponensial. Konsumsi memori yang tinggi ini dapat dijelaskan lebih lanjut melalui mekanisme pembangkitan *node* yang terus-menerus dan ekspansif, dimana setiap *node* yang dibangkitkan disimpan dalam memori, menghasilkan ruang kompleksitas yang besar.

Sebaliknya, algoritma GBFS memprioritaskan *node* berdasarkan heuristik yang menilai *node* mana yang paling menjanjikan untuk mencapai solusi dengan cepat. Fokus pada heuristik ini mengurangi kebutuhan untuk menyimpan banyak *node* yang kurang relevan,

sehingga memori yang digunakan jauh lebih kecil dibandingkan UCS. GBFS mengutamakan efisiensi pencarian melalui pemilihan node yang optimal tanpa harus memproses setiap kemungkinan secara menyeluruh.

Algoritma A* mengambil pendekatan yang lebih seimbang. Meskipun menggunakan mekanisme pembangkitan node yang bertingkat seperti UCS, A* mengintegrasikan fungsi heuristik yang mengestimasi biaya dari setiap node ke solusi. Integrasi ini memungkinkan A* untuk memilah node yang dianggap tidak efisien dan mengurangi jumlah node yang harus disimpan dalam memori. Sehingga, meskipun A* cenderung menggunakan lebih banyak memori dibandingkan dengan GBFS, ia masih menggunakan memori yang lebih efisien dibandingkan dengan UCS.

Heuristik memegang peranan krusial dalam mengarahkan proses pencarian pada algoritma *Greedy Best First Search* (GBFS) dan A* (AStar), dengan menyediakan estimasi biaya dari node yang sedang dianalisis menuju tujuan. Fungsi heuristik ini sangat berpengaruh terhadap efisiensi dan efektivitas algoritma dalam menavigasi ruang pencarian dan mencapai solusi yang optimal. Heuristik dianggap *admissible* jika selalu meremehkan biaya sebenarnya untuk mencapai tujuan. Pada algoritma AStar untuk Word Ladder, jika heuristik yang digunakan adalah jumlah huruf yang berbeda, heuristik ini *admissible* karena tidak pernah memperkirakan biaya yang lebih tinggi dari biaya sebenarnya. Setiap perbedaan huruf pasti memerlukan setidaknya satu langkah perubahan, sehingga memenuhi kriteria *admissibility*.

Pada algoritma GBFS, heuristik dimanfaatkan untuk memilih *node* yang, baik berdasarkan perhitungan paling mendekati tujuan. Heuristik yang digunakan dalam konteks *Word Ladder* adalah jumlah huruf yang berbeda antara kata pada *node* saat ini dan kata tujuan. Contohnya, jika target adalah mengubah kata "gold" menjadi "lead", dan pilihan kata-kata kandidat adalah "goad", "load", dan "golf", heuristik akan menilai sebagai berikut:

"goad" -> 2 perbedaan (g->l, d->d)

"load" -> 1 perbedaan (g->l)

"golf" -> 2 perbedaan (d->f)

Dalam situasi ini, GBFS akan memilih "load" karena memiliki nilai heuristik yang paling rendah, yang menunjukkan jarak terdekat dari kata tujuan. Namun, ini tidak selalu menjamin bahwa jalur yang ditempuh adalah yang terpendek secara keseluruhan dari "gold" ke "lead", karena GBFS tidak memperhitungkan biaya total yang telah dikeluarkan.

Di sisi lain, AStar mengintegrasikan heuristik ke dalam proses pencarian dengan cara yang lebih lengkap. AStar tidak hanya mempertimbangkan biaya untuk mencapai *node*

tertentu ($g(n)$), dimana $g(n)$ adalah biaya jalur dari titik awal ke simpul n , tetapi juga menambahkannya dengan heuristik ($h(n)$), $h(n)$ adalah heuristik yang memperkirakan biaya dari simpul n ke tujuan, untuk memberikan estimasi total biaya ($f(n) = g(n) + h(n)$) dari awal hingga tujuan melalui *node* tersebut. Menggunakan contoh yang sama, AStar akan menghitung biaya sebenarnya untuk mencapai setiap kata ditambah dengan estimasi heuristik ke "*lead*". Jika dari "*gold*" ke "*goad*" telah ditempuh dengan biaya 1 langkah, dan heuristik dari "*goad*" ke "*lead*" adalah 2, maka $f(goad) = 1 + 2 = 3$. Proses yang sama akan diulang untuk kata-kata lain, dengan AStar memilih *node* yang memiliki nilai $f(n)$ terendah pada setiap langkah.

Secara teoritis, AStar lebih efisien dibandingkan dengan UCS dalam kasus *Word Ladder* karena menggunakan heuristik untuk membatasi dan memfokuskan pencarian, yang berpotensi mengurangi jumlah *node* yang harus dievaluasi secara signifikan dibandingkan UCS yang harus mengevaluasi lebih banyak kemungkinan karena kurangnya panduan heuristik.

GBFS tidak menjamin solusi optimal karena hanya memfokuskan pada *node* yang heuristiknya paling rendah tanpa mempertimbangkan biaya yang telah dikeluarkan. Ini bisa mengarah pada solusi yang cepat tetapi tidak selalu efisien atau ekonomis dalam hal jumlah langkah total yang diperlukan untuk mencapai tujuan.

Sementara UCS memberikan kepastian dalam menemukan jalur terpendek, penggunaannya mungkin tidak ideal ketika ada keterbatasan waktu dan memori. Dalam hal ini, AStar muncul sebagai pilihan yang lebih seimbang, menawarkan solusi yang cepat dan efisien tanpa mengorbankan secara signifikan penggunaan sumber daya atau kualitas solusi. GBFS, meskipun kadang dapat menjadi algoritma yang paling cepat, kekurangannya dalam konsistensi dan potensi untuk menemukan solusi kurang optimal membuatnya kurang diandalkan untuk aplikasi yang membutuhkan kepastian solusi optimal.

4.3. Penjelasan Bonus

Bonus yang dikerjakan yang bonus GUI. GUI dibuat dengan menggunakan framework JavaFX dan bahasa pemrograman Java. Antarmuka utama dari aplikasi ini dilengkapi dengan elemen-elemen berikut:

- Input Kata Awal dan Akhir: Pengguna dapat memasukkan kata awal (start word) dan kata akhir (end word) melalui dua field teks.

- Validasi Input: Sebelum memproses pencarian, aplikasi melakukan validasi untuk memastikan bahwa kedua kata masukan tidak kosong, terdapat dalam kamus, dan memiliki panjang yang sama.
- Pilihan Algoritma Pencarian: GUI menyediakan Dropdown yang memungkinkan pengguna memilih antara tiga algoritma pencarian yang berbeda: *Uniform Cost Search (UCS)*, *Greedy Best First Search (GBFS)*, dan *A**.
- Tombol 'Find Path': Setelah memasukkan kata dan memilih algoritma, pengguna dapat memulai pencarian jalur dengan mengklik tombol 'Find Path'.
- Display Hasil: Hasil dari pencarian ditampilkan dalam format yang mudah dibaca. Setiap kata di jalur yang ditemukan disajikan dengan angka urutan, dan huruf yang berubah di setiap langkah ditekankan dengan pewarnaan dan pemformatan teks untuk memberikan visualisasi yang jelas tentang transisi antar kata. GUI juga menampilkan waktu eksekusi dan jumlah simpul yang dikunjungi.

BAB V

PENUTUP

5.1. Kesimpulan

Dari analisis yang dilakukan terhadap hasil eksperimen dengan menggunakan tiga algoritma pencarian *Uniform Cost Search* (UCS), *Greedy Best First Search* (GBFS), dan A* (*AStar*)—dalam konteks permainan *Word Ladder* dapat disimpulkan.

Pertama, waktu eksekusi yang dicatat selama percobaan mengungkapkan perbedaan yang signifikan antara algoritma-algoritma tersebut. UCS, sementara dapat diandalkan dalam menemukan jalur terpendek, menunjukkan waktu eksekusi yang lebih lama dibandingkan dengan AStar, yang menggunakan heuristik untuk meningkatkan efisiensi pencariannya. GBFS, meskipun kadang-kadang mampu mencapai hasil yang sangat cepat, variabilitasnya tinggi dan tergantung pada akurasi heuristik yang digunakan.

Kedua, dalam hal panjang jalur, UCS dan AStar konsisten dalam mencapai solusi yang optimal, menggambarkan kemampuan mereka untuk mengintegrasikan secara efektif biaya kumulatif dan jarak yang ditempuh. Sebaliknya, GBFS cenderung menghasilkan jalur yang lebih panjang dalam beberapa kasus, menegaskan kembali bahwa algoritma ini mengorbankan keakuratan untuk kecepatan.

Ketiga, analisis jumlah kata yang dicek menunjukkan bahwa UCS melakukan eksplorasi yang paling luas, mencerminkan pendekatannya yang menyeluruh namun kurang efisien. AStar memiliki efisiensi yang tinggi, membatasi eksplorasi hanya pada node yang paling berpotensi berdasarkan kombinasi biaya sebenarnya dan heuristik. GBFS menunjukkan performa yang fluktuatif, yang sekali lagi tergantung pada kualitas heuristik yang diimplementasikan.

Keempat, penggunaan memori secara signifikan lebih tinggi pada UCS karena pendekatan yang mirip dengan BFS, di mana setiap kemungkinan perpindahan diperhitungkan tanpa seleksi berbasis heuristik. GBFS dan AStar menggunakan memori lebih efisien, dengan AStar memberikan keseimbangan terbaik antara kecepatan dan penggunaan sumber daya.

Dalam konteks heuristik, penggunaan heuristik *admissible* dalam AStar menjamin bahwa algoritma tidak hanya efisien tetapi juga efektif, menghasilkan solusi optimal dengan penggunaan sumber daya yang minimal. Sementara itu, GBFS,

dengan pendekatannya yang berbasis heuristik tanpa mempertimbangkan biaya yang telah dikeluarkan, tidak menjamin solusi optimal dan bisa jadi kurang dapat diandalkan untuk aplikasi yang membutuhkan kepastian hasil.

Secara keseluruhan, analisis ini menunjukkan bahwa dalam konteks permainan Word Ladder, A* umumnya menawarkan kinerja yang lebih superior dibandingkan dengan UCS dan GBFS. Ini menjadikan A* pilihan yang sangat cocok untuk mereka yang mencari keseimbangan terbaik antara kecepatan, akurasi, dan efisiensi penggunaan memori. Di sisi lain, GBFS bisa menjadi alternatif yang menarik untuk aplikasi yang memprioritaskan kecepatan dan efisiensi, dengan kesiapan untuk menerima risiko solusi yang mungkin suboptimal. Sedangkan UCS, dengan kemampuannya yang teruji dalam menghasilkan jalur terpendek, tetap relevan dan berharga dalam situasi di mana keakuratan adalah hal yang kritis, meskipun ini mungkin berarti pengorbanan terhadap waktu eksekusi dan konsumsi memori yang lebih besar.

5.2. Saran

Untuk meningkatkan efektivitas dan efisiensi pencarian Word Ladder, disarankan untuk mengembangkan fungsi heuristik yang lebih akurat untuk algoritma GBFS dan A*. Heuristik yang lebih tepat akan mengoptimalkan jumlah kata yang dicari dan penggunaan memori. Implementasi paralelisme dalam pencarian juga dapat mempercepat proses, terutama dengan memanfaatkan teknologi multithreading atau komputasi terdistribusi. Selain itu, menggunakan dataset yang lebih bervariasi akan memastikan bahwa program tetap adaptif dan relevan terhadap berbagai skenario penggunaan. Optimasi struktur data dan integrasi model pembelajaran mesin untuk prediksi jalur mungkin juga dapat meningkatkan akurasi dan efisiensi algoritma secara signifikan. Keseluruhan saran ini bertujuan untuk mengembangkan solusi yang lebih cepat, efisien, dan intuitif dalam aplikasi Word Ladder Solver.

DAFTAR REFERENSI

- Munir, Rinaldi. Penentuan rute (Route/Path Planning) - Bagian 1. Diakses 4 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 4 Mei 2024.
<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf>
- Trivusi. Apa itu Uniform-Cost Search? Pengertian dan Cara Kerjanya. Diakses 4 Mei 2024.
<https://www.trivusi.web.id/2022/10/apa-itu-algoritma-uniform-cost-search.html>

LAMPIRAN

Link Github: https://github.com/AlbertChoe/Tucil3_13522081

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	✓	