

## **TUGAS KECIL 3 IF2211 STRATEGI ALGORITMA**

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



**Disusun oleh:**

Shazya Audrea Taufik

13522063

**Program Studi Teknik Informatika  
Sekolah Teknik Elektro dan Informatika  
Institut Teknologi Bandung  
2024**

## DAFTAR ISI

<b>DAFTAR ISI.....</b>	<b>1</b>
<b>BAB I ANALISIS DAN IMPLEMENTASI.....</b>	<b>2</b>
1.1 Algoritma Uniform Cost Search (UCS).....	2
1.2 Algoritma Greedy Best First Search (GBFS).....	3
1.3 Algoritma A Star Search.....	4
1.4 Analisis Teori.....	5
<b>BAB II ALUR PROGRAM.....</b>	<b>7</b>
<b>BAB III SOURCE CODE.....</b>	<b>9</b>
3.1 Algorithm.java.....	9
3.1.1 Source Code.....	9
3.1.2 Class.....	11
3.1.3 Method.....	12
3.2 Node.java.....	12
3.2.1 Source Code.....	12
3.2.2 Class.....	13
3.2.3 Method.....	13
3.3 WordsDatabase.java.....	13
3.3.1 Source Code.....	13
3.3.2 Class.....	14
3.3.3 Method.....	14
3.4 WordLadderGUI.java.....	14
3.4.1 Source Code.....	14
3.4.2 Class.....	17
3.4.3 Method.....	17
3.5 Result.java.....	18
3.5.1 Source Code.....	18
3.5.2 Class.....	18
3.5.3 Method.....	18
3.6 Main.java.....	18
3.6.2 Class.....	19
3.6.3 Method.....	19
<b>BAB IV HASIL PENGUJIAN.....</b>	<b>20</b>
4.1 Test Case 1.....	20
4.2 Test Case 2.....	20
4.3 Test Case 3.....	20
4.4 Test Case 4.....	20
4.5 Test Case 5.....	21
4.6 Test Case 6.....	21

4.7 Test Case 7.....	21
4.8 Test Case 8.....	22
4.9 Test Case 9.....	22
4.10 Test Case 10.....	22
<b>BAB V ANALISIS PERBANDINGAN SOLUSI.....</b>	<b>24</b>
<b>BAB VI LAMPIRAN.....</b>	<b>25</b>
6.1 Link Repository.....	25
6.2 Tabel Kelengkapan.....	25

## BAB I

### ANALISIS DAN IMPLEMENTASI

#### 1.1 Algoritma Uniform Cost Search (UCS)

Uniform Cost Search (UCS) adalah algoritma pencarian jalur terpendek dalam ruang pencarian berbobot di mana biaya setiap langkah antara node berbeda. Algoritma ini mirip dengan algoritma Breadth First Search (BFS), tetapi dengan mempertimbangkan biaya dari setiap langkah.

Langkah penggunaan algoritma UCS pada metode penyelesaian dari masalah ini adalah sebagai berikut:

1. Inisialisasi struktur data yang dibutuhkan:
  - priorityQueue: menyimpan node yang akan dicek selanjutnya. Node yang memiliki nilai *cost* terendah (dalam hal ini,  $F_n$  terendah) akan diambil terlebih dahulu.
  - visited: Set yang menyimpan kata-kata yang sudah dieksplorasi agar tidak dieksplorasi lagi.
  - nodesVisited: Variabel untuk menghitung jumlah node yang telah dikunjungi.
2. Tambahkan node awal ke dalam priorityQueue dengan *cost* 0.
3. Selama priorityQueue tidak kosong:
  - Ambil node teratas dari priorityQueue.
  - Tambahkan 1 ke jumlah nodesVisited.
  - Periksa apakah kata dari node saat ini sama dengan kata akhir. Jika iya, maka solusi telah ditemukan. Buat objek Result yang berisi *path* yang ditemukan menggunakan metode wordPath dan jumlah node yang dikunjungi, lalu kembalikan hasilnya.
  - Jika kata dari node saat ini belum pernah dikunjungi, kata tersebut ditandai sebagai *visited* dan untuk setiap tetangga dari kata ini, jika belum dikunjungi, maka tetangga tersebut akan masuk ke priorityQueue dengan menambahkan 1 ke biaya dari node saat ini ( $F_n + 1$ ).
4. Jika tidak ada solusi yang ditemukan (yaitu priorityQueue kosong), buat objek Result yang berisi daftar kosong sebagai *path* dan jumlah node yang dikunjungi, lalu kembalikan hasilnya.

## 1.2 Algoritma Greedy Best First Search (GBFS)

Greedy Best First Search (GBFS) adalah algoritma pencarian jalur yang mirip dengan algoritma Best First Search (BFS), tetapi dengan strategi lebih "greedy" dalam pemilihan node berikutnya untuk dieksplorasi. GBFS memilih node berikutnya berdasarkan *heuristic value* dari node tersebut, dengan tujuan mencapai node tujuan secepat mungkin.

1. Inisialisasi struktur data yang dibutuhkan:
  - priorityQueue: *Priority Queue* menyimpan node yang akan dicek berikutnya. Node-node ini diurutkan berdasarkan *heuristic value* dari masing-masing node. Node dengan *heuristic value* yang lebih rendah akan diberikan prioritas lebih tinggi.
  - visitedMap: *map* yang menyimpan nilai heuristik terbaik yang telah dikunjungi untuk setiap kata yang telah dieksplorasi sebelumnya.
  - nodesVisited: Variabel untuk menghitung jumlah node yang telah dikunjungi.
2. Tambahkan node awal ke dalam priorityQueue dengan *heuristic value* 0.
3. Selama priorityQueue tidak kosong:
  - Ambil node teratas dari priorityQueue.
  - Tambahkan 1 ke jumlah nodesVisited.
  - Periksa apakah kata dari node saat ini sama dengan kata akhir. Jika iya, maka solusi telah ditemukan. Buat objek Result yang berisi *path* yang ditemukan menggunakan metode wordPath dan jumlah node yang dikunjungi, lalu *return* hasilnya.
  - Jika kata dari node saat ini belum pernah dikunjungi atau memiliki heuristic value yang lebih baik dari yang telah dikunjungi sebelumnya, kata tersebut ditandai sebagai *visited* dengan heuristic value yang baru.
  - Untuk setiap tetangga yang belum dieksplorasi atau memiliki heuristic value yang lebih baik dari yang telah dikunjungi sebelumnya, tambahkan tetangga tersebut ke dalam antrian prioritas dengan *heuristic value* 0. Ini berarti tetangga tersebut akan memiliki prioritas yang lebih tinggi untuk dieksplorasi berikutnya.
4. Jika antrian prioritas kosong tanpa menemukan solusi, ini menunjukkan bahwa tidak ada jalur yang memungkinkan antara node awal dan akhir.

### 1.3 Algoritma A Star Search

A\* (A-star) adalah salah satu algoritma pencarian jalur dalam graf berbobot. Algoritma ini menggabungkan dua strategi, yaitu informed search (pencarian berinformasi) dan best-first search (pencarian terbaik pertama), untuk menggabungkan keunggulan dari kedua pendekatan tersebut.

1. Inisialisasi struktur data yang dibutuhkan:
  - mapHeuristic: map yang menyimpan nilai heuristik dari setiap kata pada kamus.
  - priorityQueue: *Priority Queue* menyimpan node yang akan dicek berikutnya. Node-node ini diurutkan berdasarkan nilai  $f(n)$  (jumlah cost dengan nilai heuristik) dari masing-masing node. Node dengan nilai  $f(n)$  yang lebih rendah akan diberikan prioritas lebih tinggi.
  - costMap: map yang menyimpan biaya aktual untuk mencapai setiap kata dari node awal.
2. Hitung nilai heuristik dari kata awal dan masukkan nilai tersebut ke mapHeuristic.
3. Masukkan node awal ke priorityQueue beserta nilai  $f(n)$  nya.
4. Selama priorityQueue tidak kosong:
  - Ambil node teratas dari priorityQueue.
  - Tambahkan 1 ke jumlah nodesVisited.
  - Periksa apakah kata dari node saat ini sama dengan kata akhir. Jika iya, maka solusi telah ditemukan. Buat objek Result yang berisi *path* yang ditemukan menggunakan metode wordPath dan jumlah node yang dikunjungi, lalu *return* hasilnya.
  - Jika kata dari node saat ini belum pernah dikunjungi atau memiliki heuristic value yang lebih baik dari yang telah dikunjungi sebelumnya, kata tersebut ditandai sebagai *visited* dengan heuristic value yang baru.
  - Untuk setiap tetangga dari kata terkini, hitung  $g(n)$ ,  $h(n)$ , dan  $f(n)$ . Jika tetangga tersebut memiliki *cost* lebih rendah atau belum dikunjungi, tambahkan tetangga tersebut ke dalam antrian prioritas. Masukkan nilai heuristik ke mapHeuristic dan masukkan nilai cost ke costMap.
5. Jika antrian prioritas kosong tanpa menemukan solusi, ini menunjukkan bahwa tidak ada jalur yang memungkinkan antara node awal dan akhir.

## 1.4 Analisis Teori

$f(n)$  adalah biaya total yang diperlukan untuk mencapai node  $n$  dari node awal.  $g(n)$  adalah biaya aktual untuk mencapai node  $n$  dari node awal.  $h(n)$  adalah fungsi heuristik yang digunakan untuk memperkirakan biaya yang tersisa dari suatu node  $n$  ke goal node. Fungsi  $h(n)$  dikatakan admissible jika nilainya tidak pernah melebihi biaya aktual dari node  $n$  ke goal node ( $h(n) \leq h^*(n)$  dengan  $h^*(n)$  adalah biaya aktual).

```
private int heuristicValue (String currentWord, String endWord) {  
    int hValue = 0;  
    for (int i = 0; i < currentWord.length(); i++) {  
        if (currentWord.charAt(i) != endWord.charAt(i)) {  
            hValue++;  
        }  
    }  
    return hValue;  
}
```

Jika setiap transformasi yang valid didefinisikan sebagai perubahan satu karakter, maka fungsi heuristik ini bisa dianggap *admissible*, karena jumlah perubahan karakter yang diperlukan akan selalu lebih besar dari atau sama dengan jumlah perbedaan karakter antara dua kata. Fungsi ini tidak melebih-lebihkan biaya aktual. Dalam konteks permainan ini (perubahan hanya diizinkan satu karakter pada satu waktu dan ke kata-kata perantara yang valid), maka fungsi heuristik ini dianggap *admissible*.

Dalam word ladder, UCS memprioritaskan simpul berdasarkan biaya total dari simpul awal ke simpul saat ini, sementara BFS hanya mempertimbangkan jumlah langkah dari simpul awal tanpa memandang *cost*. BFS akan menelusuri simpul per level, artinya semua node pada kedalaman tertentu dieksekusi sebelum berpindah ke lapisan berikutnya. UCS mengutamakan node dengan biaya terendah. Dengan biaya seragam pada setiap perubahan, UCS mirip BFS. Namun, jika ada biaya berbeda untuk setiap perubahan huruf, maka urutan simpul yang dieksekusi oleh UCS dapat berbeda dari BFS. Secara umum, dalam skenario Word Ladder, dengan biaya seragam, UCS akan menghasilkan urutan node dan jalur yang dihasilkan sama dengan BFS. Namun, jika biaya langkah bervariasi, urutan dan hasilnya dapat berbeda.

Secara teoritis, algoritma A\* umumnya lebih efisien dibandingkan dengan Uniform Cost Search (UCS) dalam kasus word ladder. Uniform Cost Search (UCS) bekerja dengan mengeksplorasi node yang memiliki biaya terendah dari root ke node saat ini. Setiap langkah dipertimbangkan berdasarkan biaya kumulatifnya dari node awal. A\* menggabungkan dua faktor

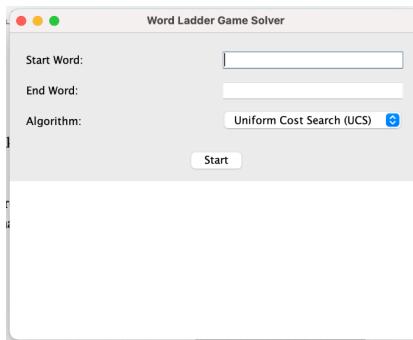
dalam eksplorasinya, yaitu biaya kumulatif dari root ke node saat ini ( $g(n)$ ) dan heuristik untuk mengestimasi jarak ke tujuan ( $h(n)$ ). Dengan demikian, A\* cenderung lebih cepat karena menggunakan heuristik untuk memandu eksplorasi lebih cepat ke solusi akhir dengan biaya total yang lebih rendah. Dalam konteks word ladder, dengan heuristik yang baik, A\* dapat memotong banyak eksplorasi yang tidak diperlukan dibandingkan dengan UCS, yang harus memeriksa setiap cabang tanpa panduan heuristik tersebut.

Secara teoritis, algoritma Greedy Best First Search (GBFS) tidak menjamin solusi optimal karena GBFS tidak mempertimbangkan total biaya kumulatif dari awal. Algoritma ini dapat memilih jalur yang mungkin terlihat lebih dekat ke tujuan menurut heuristik, tetapi sebenarnya memiliki biaya total yang lebih tinggi. Hal ini menunjukkan bahwa jalur yang dihasilkan dengan GBFS tidak selalu jalur terpendek atau optimal.

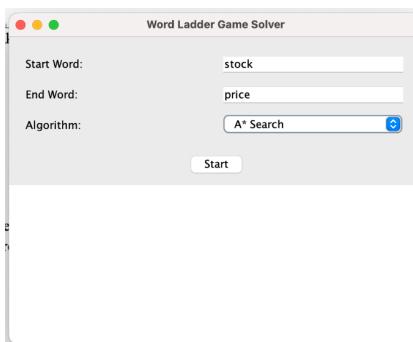
## BAB II

### ALUR PROGRAM

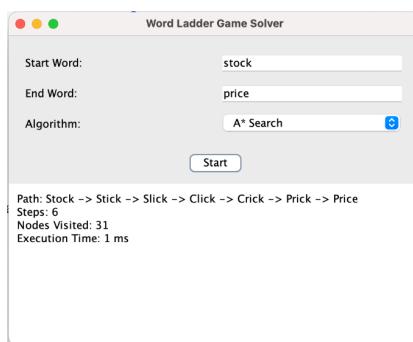
1. Akan muncul tampilan utama



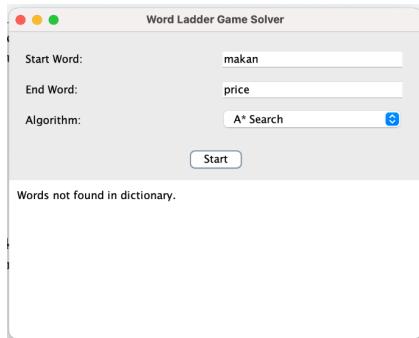
2. Pengguna memasukkan kata awal dan kata akhir serta memilih pilihan algoritma pada dropdown pilihan (UCS, GBFS, atau A\*).



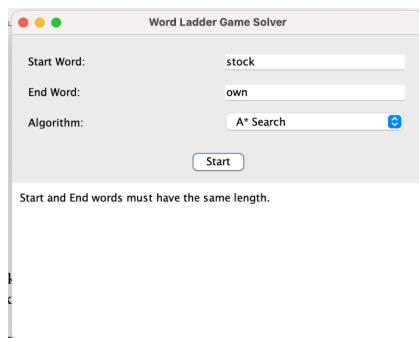
3. Pengguna menekan tombol 'Start'. Jika pengguna memasukkan kata yang ada pada dictionary, memiliki panjang sama, dan juga hanya mengandung huruf, maka hasil akan muncul.



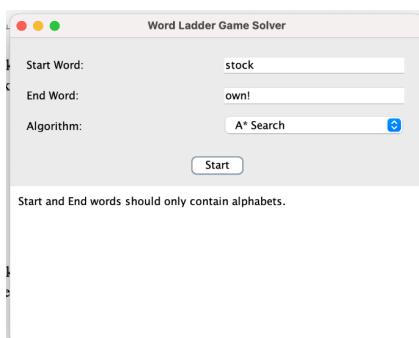
4. Jika pengguna memasukkan kata yang tidak ada pada kamus, maka akan muncul pesan error.



5. Jika pengguna memasukkan kata awal dan kata akhir yang panjangnya tidak sama, maka akan muncul pesan error.



6. Jika pengguna memasukkan kata dengan *special characters* atau angka, maka akan muncul pesan error.



## BAB III

### SOURCE CODE

#### 3.1 Algorithm.java

##### 3.1.1 Source Code

```
import java.util.*;
public class Algorithm {
    public Result solveAStar (String start, String end, WordsDatabase
dictionary) {
        Map<String, Integer> mapHeuristic = new HashMap<>();
        PriorityQueue<Node> priorityQueue = new
PriorityQueue<>(Comparator.comparingInt(Node::getFn));
        Map<String, Integer> costMap = new HashMap<>();
        int startH = heuristicValue(start, end);
        int nodesVisited = 0;
        mapHeuristic.put(start, startH);
        priorityQueue.offer(new Node(start, null, startH));
        costMap.put(start, 0);
        while (!priorityQueue.isEmpty()) {
            Node currentNode = priorityQueue.poll();
            int currentG = costMap.get(currentNode.getWord());
            nodesVisited++;
            if (currentNode.getWord().equals(end)) {
                return new Result(wordPath(currentNode), nodesVisited);
            }
            for (String neighbor: neighboursList(currentNode.getWord(),
dictionary)) {
                int currentCost = currentG + 1;
                int neighborH = heuristicValue(neighbor, end);
                int fValue = currentCost + neighborH;
                if (!costMap.containsKey(neighbor) || currentCost <
costMap.get(neighbor)) {
                    mapHeuristic.put(neighbor, neighborH);
                    priorityQueue.offer(new Node(neighbor, currentNode,
fValue));
                    costMap.put(neighbor, currentCost);
                }
            }
            return new Result(Collections.emptyList(), nodesVisited);
        }
        public Result solveGBFS (String start, String end, WordsDatabase
dictionary) {
            PriorityQueue<Node> priorityQueue = new
PriorityQueue<>(Comparator.comparingInt(node -> heuristicValue(node.getWord(),
end)));
            Map<String, Integer> visitedMap = new HashMap<>();
```

```

priorityQueue.offer(new Node(start, null, 0));
int nodesVisited = 0;
while (!priorityQueue.isEmpty()) {
    Node currentNode = priorityQueue.poll();
    nodesVisited++;
    if (currentNode.getWord().equals(end)) {
        return new Result(wordPath(currentNode), nodesVisited);
    }
    if (!visitedMap.containsKey(currentNode.getWord()) ||
visitedMap.get(currentNode.getWord()) > heuristicValue(currentNode.getWord(),
end)) {
        visitedMap.put(currentNode.getWord(),
heuristicValue(currentNode.getWord(), end));
        for (String neighbor : neighboursList(currentNode.getWord(),
dictionary)) {
            int neighborHeuristic = heuristicValue(neighbor, end);
            if (!visitedMap.containsKey(neighbor) ||
visitedMap.get(neighbor) > neighborHeuristic) {
                priorityQueue.offer(new Node(neighbor, currentNode,
0));
            }
        }
    }
    return new Result(Collections.emptyList(), nodesVisited);
}
public Result solveUCS(String start, String end, WordsDatabase dictionary)
{
    PriorityQueue<Node> priorityQueue = new
PriorityQueue<>(Comparator.comparingInt(Node::getFn));
    Set<String> visited = new HashSet<>();
    int nodesVisited = 0;
    priorityQueue.add(new Node(start, null, 0));
    while (!priorityQueue.isEmpty()) {
        Node currentNode = priorityQueue.poll();
        nodesVisited++;
        if (currentNode.getWord().equals(end)) {
            return new Result(wordPath(currentNode), nodesVisited);
        }
        if (!visited.contains(currentNode.getWord())) {
            visited.add(currentNode.getWord());
            for (String neighbor : neighboursList(currentNode.getWord(),
dictionary)) {
                if (!visited.contains(neighbor)) {
                    priorityQueue.add(new Node(neighbor, currentNode,
currentNode.getFn() + 1));
                }
            }
        }
    }
}

```

```

        }
    }
    return new Result(Collections.emptyList(), nodesVisited);
}
private int heuristicValue (String currentWord, String endWord) {
    int hValue = 0;
    for (int i = 0; i < currentWord.length(); i++) {
        if (currentWord.charAt(i) != endWord.charAt(i)) {
            hValue++;
        }
    }
    return hValue;
}
private List<String> neighboursList(String word, WordsDatabase dictionary)
{
    List<String> neighbours = new ArrayList<>();
    char[] letters = word.toCharArray();
    for (int i = 0; i < letters.length; i++) {
        char before = letters[i];

        for (char c = 'A'; c <= 'Z'; c++) {
            if (c != before) {
                letters[i] = c;
                String after = new String(letters);
                if (dictionary.inDictionary(after)) {
                    neighbours.add(after);
                }
            }
        }
        letters[i] = before;
    }
    return neighbours;
}
private List<String> wordPath (Node node) {
    LinkedList<String> path = new LinkedList<>();

    while (node != null) {
        path.addFirst(node.getWord());
        node = node.getParent();
    }
    return path;
}
}

```

### 3.1.2 Class

Class	Keterangan
Algorithm	Terdiri dari metode penyelesaian UCS, GBFS,

	dan A* beserta metode penunjang lainnya.
--	--

### 3.1.3 Method

Method	Keterangan
public Result solveAStar (String start, String end, WordsDatabase dictionary)	Menyelesaikan permasalahan word ladder dengan algoritma A*. Parameternya adalah kata awal, kata akhir, dan kamus kata. Fungsi ini mengembalikan Result yang terdiri dari path dan jumlah node yang dikunjungi.
public Result solveGBFS (String start, String end, WordsDatabase dictionary)	Menyelesaikan permasalahan word ladder dengan algoritma GBFS. Parameternya adalah kata awal, kata akhir, dan kamus kata. Fungsi ini mengembalikan Result yang terdiri dari path dan jumlah node yang dikunjungi.
public Result solveUCS(String start, String end, WordsDatabase dictionary)	Menyelesaikan permasalahan word ladder dengan algoritma UCS. Parameternya adalah kata awal, kata akhir, dan kamus kata. Fungsi ini mengembalikan Result yang terdiri dari path dan jumlah node yang dikunjungi.
private int heuristicValue (String currentWord, String endWord)	Fungsi tambahan yang diperlukan GBFS dan A* untuk mencari nilai heuristik. Parameter nya adalah kata terkini dan kata akhir. Fungsi ini mengembalikan integer yang merupakan nilai heuristik.
private List<String> neighboursList(String word, WordsDatabase dictionary)	Fungsi tambahan yang dibutuhkan ketiga algoritma untuk mencari list tetangga dari suatu kata.
private List<String> wordPath (Node node)	Fungsi tambahan untuk mencari path kata.

## 3.2 Node.java

### 3.2.1 Source Code

```
public class Node {
    private String word;
    private Node parent;
    private int Fn;
    public Node(String word, Node parent, int Fn) {
        this.word = word;
        this.parent = parent;
        this.Fn = Fn;
```

```

    }
    public String getWord() {
        return word;
    }
    public Node getParent() {
        return parent;
    }
    public int getFn() {
        return Fn;
    }
}

```

### 3.2.2 Class

Class	Keterangan
Node	Membentuk node yang terdiri dari kata, parent node, dan nilai f(n). Memiliki atribut private String word, private Node parent, private int Fn

### 3.2.3 Method

Method	Keterangan
public Node(String word, Node parent, int Fn)	Membentuk node dengan word, parent, Fn pada parameter.
public String getWord()	Getter untuk kata pada node.
public Node getParent()	Getter untuk node parent.
public int getFn()	Getter untuk nilai f(n)

## 3.3 WordsDatabase.java

### 3.3.1 Source Code

```

import java.io.*;
import java.util.*;
public class WordsDatabase {
    public WordsDatabase(String filename) throws IOException {
        wordSet = new LinkedHashSet<>();
        String line;
        BufferedReader reader = new BufferedReader(new FileReader(filename));
        while ((line = reader.readLine()) != null) {
            wordSet.add(line.trim());
        }
        reader.close();
    }
    public boolean inDictionary(String word) {

```

```

        return wordSet.contains(word);
    }
    private Set<String> wordSet;
}

```

### 3.3.2 Class

Class	Keterangan
WordsDatabase	Untuk menyimpan semua kata pada dictionary.txt dalam wordSet. Memiliki atribut private Set<String> wordSet

### 3.3.3 Method

Method	Keterangan
public WordsDatabase(String filename) throws IOException	Membentuk node dengan input nama file.
public boolean inDictionary(String word)	Mengembalikan true/false berdasarkan apakah sebuah kata terdapat pada kamus kata.

## 3.4 WordLadderGUI.java

### 3.4.1 Source Code

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class WordLadderGUI extends JFrame {
    private JTextField startWordField;
    private JTextField endWordField;
    private JComboBox<String> algorithmDropdown;
    private JTextArea outputArea;
    public WordLadderGUI(WordsDatabase dictionary) {
        setTitle("Word Ladder Game Solver");
        setSize(500, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        setLayout(new BorderLayout());
        // Input
        JPanel inputPanel = new JPanel();
        inputPanel.setLayout(new BoxLayout(inputPanel, BoxLayout.Y_AXIS));
        inputPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

```

```

JPanel fieldsPanel = new JPanel(new GridLayout(3, 2, 10, 10));
fieldsPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10,
10));
fieldsPanel.add(new JLabel("Start Word:"));
startWordField = new JTextField();
fieldsPanel.add(startWordField);
fieldsPanel.add(new JLabel("End Word:"));
endWordField = new JTextField();
fieldsPanel.add(endWordField);
fieldsPanel.add(new JLabel("Algorithm:"));
algorithmDropdown = new JComboBox<>(new String[]{"Uniform Cost Search
(UCS)", "Greedy Best First Search (GBFS)", "A* Search"});
fieldsPanel.add(algorithmDropdown);
inputPanel.add(fieldsPanel);
// Tombol Start
JButton startButton = new JButton("Start");
startButton.setAlignmentX(Component.CENTER_ALIGNMENT);
startButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        String startWord = startWordField.getText().trim();
        String endWord = endWordField.getText().trim();
        if (startWord.isEmpty() || endWord.isEmpty()) {
            outputArea.setText("Please provide both Start and End
words.");
            return;
        }
        if (!isValidWord(startWord) || !isValidWord(endWord)) {
            outputArea.setText("Start and End words should only
contain alphabets.");
            return;
        }
        if (startWord.length() != endWord.length()) {
            outputArea.setText("Start and End words must have the same
length.");
            return;
        }
        int algorithmChoice = algorithmDropdown.getSelectedIndex() +
1;
        String result = run(startWord, endWord, algorithmChoice);
        outputArea.setText(result);
    }
});
inputPanel.add(Box.createVerticalStrut(10));
inputPanel.add(startButton);
add(inputPanel, BorderLayout.NORTH);
// Output Area
outputArea = new JTextArea(10, 30); // Adjust initial rows and

```

```

columns as necessary
        outputArea.setEditable(false);
        outputArea.setLineWrap(true);
        outputArea.setWrapStyleWord(true);
        JScrollPane scrollPane = new JScrollPane(outputArea);
        scrollPane.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
        add(scrollPane, BorderLayout.CENTER);
        setVisible(true);
    }
    private String run(String start, String end, int algorithmChoice) {
        try {
            WordsDatabase dictionary = new WordsDatabase("dictionary.txt");

            start = start.toUpperCase();
            end = end.toUpperCase();

            if (!dictionary.inDictionary(start) ||
!dictionary.inDictionary(end)) {
                return "Words not found in dictionary.";
            }

            long startTime = System.currentTimeMillis();
            List<String> path;
            Algorithm algo = new Algorithm();
            switch (algorithmChoice) {
                case 1:
                    path = algo.solveUCS(start, end, dictionary);
                    break;
                case 2:
                    path = algo.solveGBFS(start, end, dictionary);
                    break;
                case 3:
                    path = algo.solveAStar(start, end, dictionary);
                    break;
                default:
                    return "Invalid";
            }
            long endTime = System.currentTimeMillis();

            if (path == null || path.isEmpty()) {
                return "No path found between " + start + " and " + end + "\n" +
+ "Nodes Visited: " + (nodesVisited) + "\n" + "Execution Time: " + (endTime -
startTime) + " ms";
            }
            for (int i = 0; i < path.size(); i++) {
                path.set(i, capitalizeWord(path.get(i)));
            }
        }
    }
}

```

```

        return "Path: " + String.join(" -> ", path) + "\n" + "Steps: " +
(path.size() - 1) + "\n" + "Nodes Visited: " + (nodesVisited) + "\n" +
"Execution Time: " + (endTime - startTime) + " ms";
    } catch (IOException e) {
        return "Failed to load the dictionary. Error: " + e.getMessage();
    }
}
private String capitalizeWord(String word) {
    if (word == null || word.isEmpty()) {
        return word;
    }
    return word.substring(0, 1).toUpperCase() +
word.substring(1).toLowerCase();
}
private boolean isValidWord(String word) {
    Pattern pattern = Pattern.compile("^[a-zA-Z]+$");
    Matcher matcher = pattern.matcher(word);
    return matcher.matches();
}
}
}

```

### 3.4.2 Class

Class	Keterangan
WordLadderGUI	Untuk membuat GUI dari word ladder solver. Memiliki atribut private JTextField startWordField, private JTextField endWordField, private JComboBox<String> algorithmDropdown, private JTextArea outputArea.

### 3.4.3 Method

Method	Keterangan
public WordLadderGUI (WordsDatabase dictionary)	Terdiri dari segala layout pada GUI dan juga untuk meng-run program.
private String run(String start, String end, int algorithmChoice)	Memanggil algoritma atau langkah-langkah yang diperlukan untuk mencari path pada word ladder.
private String capitalizeWord(String word)	Mengubah semua kata pada path agar dalam bentuk title case.
private boolean isValidWord(String word)	Mengecek apakah input kata tidak mengandung angka ataupun special character.

## 3.5 Result.java

### 3.5.1 Source Code

```
import java.util.*;
public class Result {
    private List<String> path;
    private int visitedNodes;
    public Result(List<String> path, int visitedNodes) {
        this.path = path;
        this.visitedNodes = visitedNodes;
    }

    public List<String> getPath() {
        return path;
    }
    public int getVisitedNodes() {
        return visitedNodes;
    }
}
```

### 3.5.2 Class

Class	Keterangan
Result	Untuk menyimpan path dan nodeVisited sebagai hasil dari ketiga algoritma. Terdiri dari private List<String> path dan private int visitedNodes

### 3.5.3 Method

Method	Keterangan
public Result(List<String> path, int visitedNodes)	Konstruktor Result.
public List<String> getPath()	Getter dari path.
public int getVisitedNodes()	Getter dari visited nodes.

## 3.6 Main.java

```
import java.io.*;
import javax.swing.JOptionPane;
import javax.swing.SwingUtilities;
public class Main {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            try {
                WordsDatabase dictionary = new
```

```

        WordsDatabase("dictionary.txt");
            new WordLadderGUI(dictionary);
        } catch (IOException e) {
            JOptionPane.showMessageDialog(null, "Failed to load the
dictionary. Error: " + e.getMessage());
        }
    });
}
}

```

### 3.6.2 Class

Class	Keterangan
Main	Berisi main program

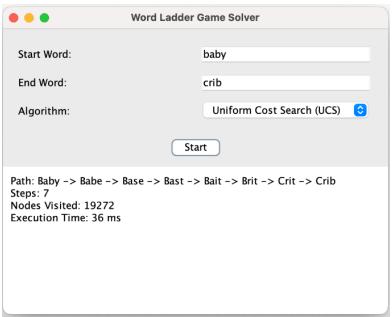
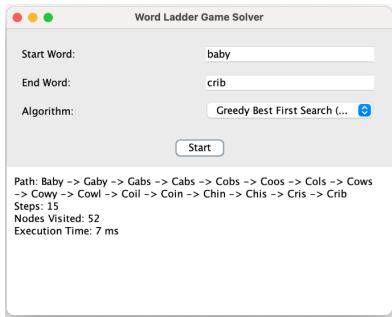
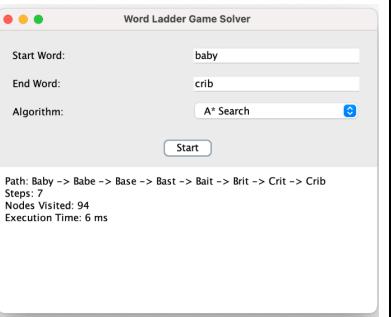
### 3.6.3 Method

Method	Keterangan
public static void main(String[] args)	Memanggil WordLadderGUI agar program bisa dijalankan

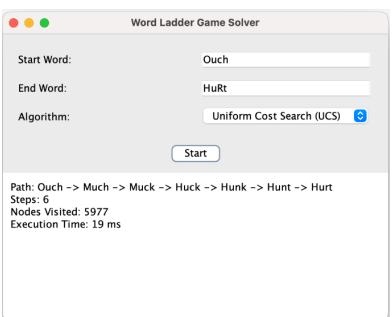
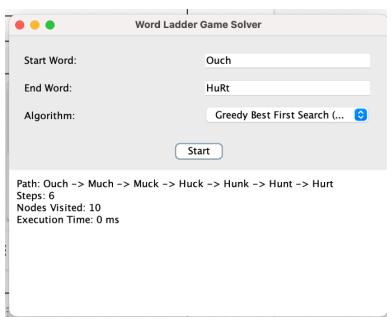
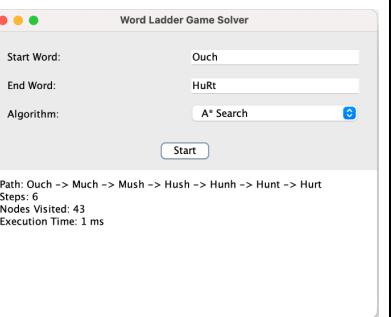
## BAB IV

### HASIL PENGUJIAN

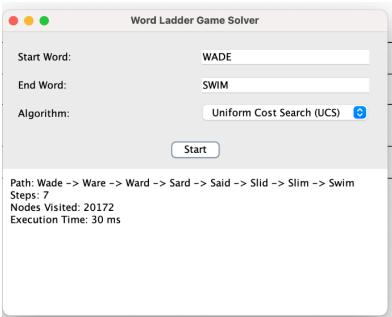
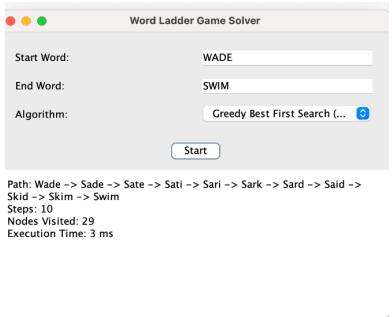
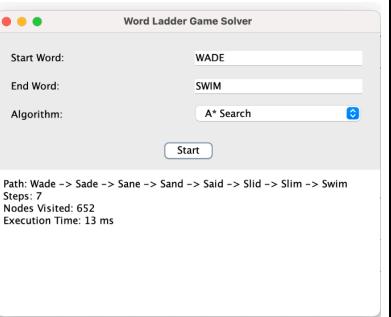
#### 4.1 Test Case 1

UCS	GBFS	A*
 <p>Start Word: baby End Word: crib Algorithm: Uniform Cost Search (UCS)</p> <p>Path: Baby -&gt; Babe -&gt; Base -&gt; Bast -&gt; Bait -&gt; Brit -&gt; Crit -&gt; Crib Steps: 7 Nodes Visited: 19272 Execution Time: 36 ms</p>	 <p>Start Word: baby End Word: crib Algorithm: Greedy Best First Search ...</p> <p>Path: Baby -&gt; Gaby -&gt; Gabs -&gt; Cabs -&gt; Cobs -&gt; Coos -&gt; Cols -&gt; Cows -&gt; Cowy -&gt; Cowl -&gt; Coil -&gt; Coin -&gt; Chin -&gt; Chis -&gt; Cris -&gt; Crib Steps: 52 Nodes Visited: 52 Execution Time: 7 ms</p>	 <p>Start Word: baby End Word: crib Algorithm: A* Search</p> <p>Path: Baby -&gt; Babe -&gt; Base -&gt; Bast -&gt; Bait -&gt; Brit -&gt; Crit -&gt; Crib Steps: 7 Nodes Visited: 94 Execution Time: 6 ms</p>

#### 4.2 Test Case 2

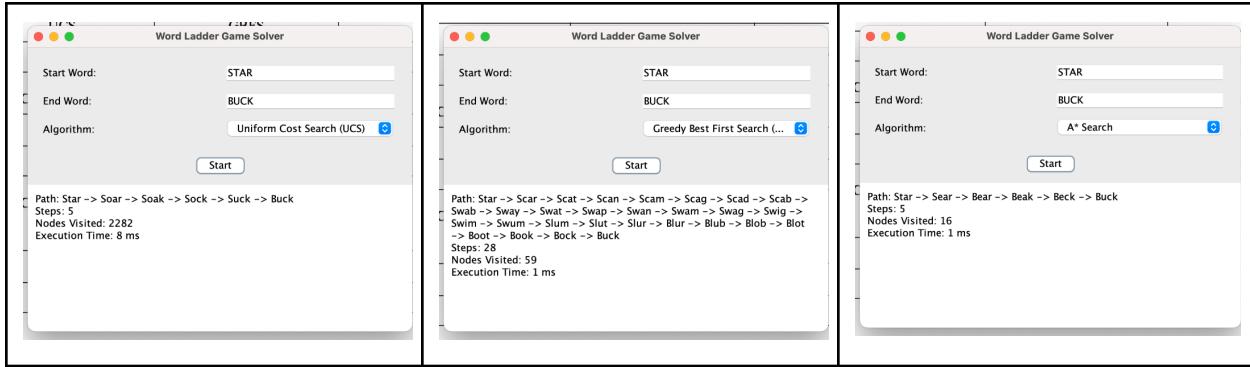
UCS	GBFS	A*
 <p>Start Word: Ouch End Word: HuRt Algorithm: Uniform Cost Search (UCS)</p> <p>Path: Ouch -&gt; Much -&gt; Muck -&gt; Huck -&gt; Hunk -&gt; Hunt -&gt; Hurt Steps: 6 Nodes Visited: 5977 Execution Time: 19 ms</p>	 <p>Start Word: Ouch End Word: HuRt Algorithm: Greedy Best First Search ...</p> <p>Path: Ouch -&gt; Much -&gt; Muck -&gt; Huck -&gt; Hunk -&gt; Hunt -&gt; Hurt Steps: 10 Nodes Visited: 10 Execution Time: 0 ms</p>	 <p>Start Word: Ouch End Word: HuRt Algorithm: A* Search</p> <p>Path: Ouch -&gt; Much -&gt; Muck -&gt; Huck -&gt; Hunk -&gt; Hunt -&gt; Hurt Steps: 6 Nodes Visited: 43 Execution Time: 1 ms</p>

#### 4.3 Test Case 3

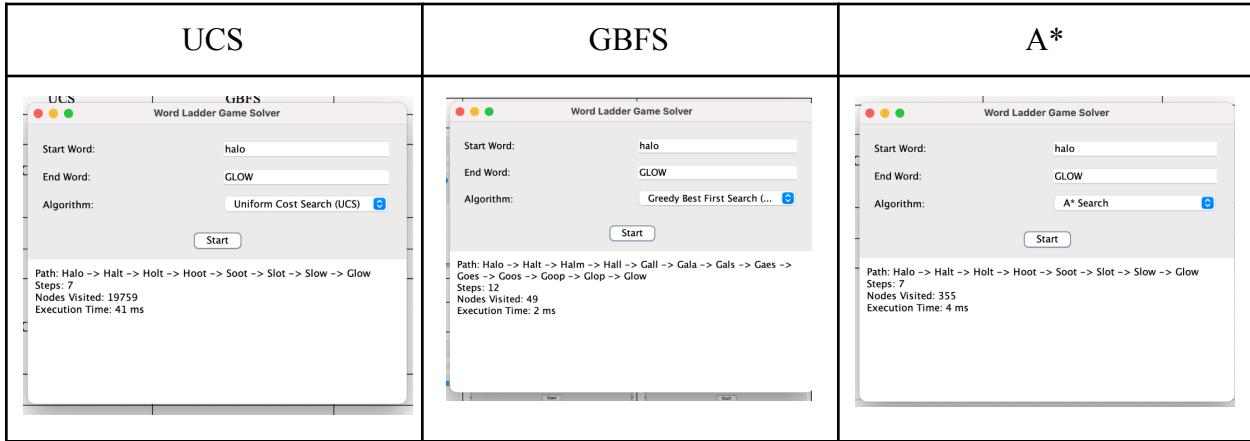
UCS	GBFS	A*
 <p>Start Word: WADE End Word: SWIM Algorithm: Uniform Cost Search (UCS)</p> <p>Path: Wade -&gt; Ware -&gt; Ward -&gt; Sard -&gt; Said -&gt; Slid -&gt; Slim -&gt; Swim Steps: 7 Nodes Visited: 20172 Execution Time: 30 ms</p>	 <p>Start Word: WADE End Word: SWIM Algorithm: Greedy Best First Search ...</p> <p>Path: Wade -&gt; Sade -&gt; Sate -&gt; Sat -&gt; Sari -&gt; Sark -&gt; Sard -&gt; Said -&gt; Skid -&gt; Skim -&gt; Swim Steps: 10 Nodes Visited: 29 Execution Time: 3 ms</p>	 <p>Start Word: WADE End Word: SWIM Algorithm: A* Search</p> <p>Path: Wade -&gt; Sade -&gt; Sane -&gt; Sand -&gt; Said -&gt; Slid -&gt; Slim -&gt; Swim Steps: 7 Nodes Visited: 652 Execution Time: 13 ms</p>

#### 4.4 Test Case 4

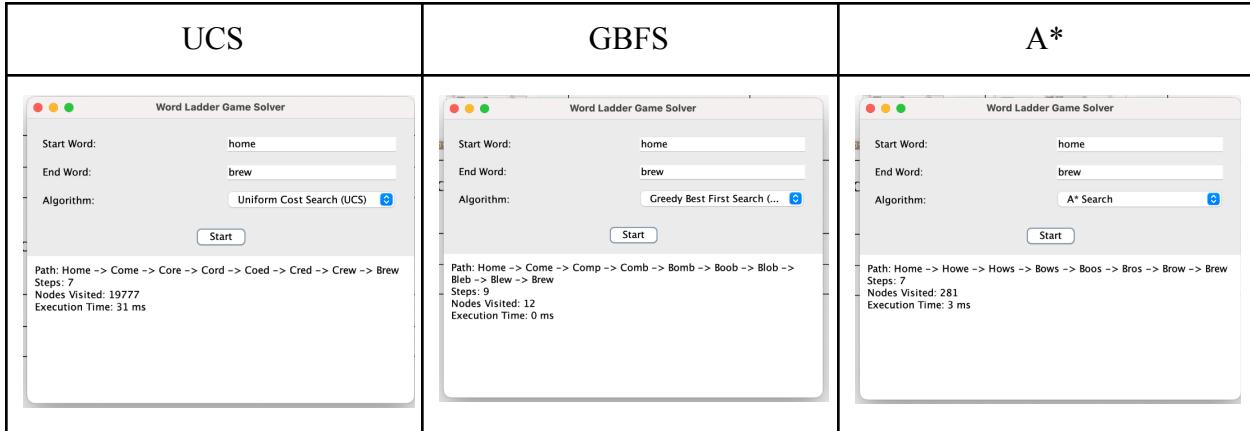
UCS	GBFS	A*



#### 4.5 Test Case 5

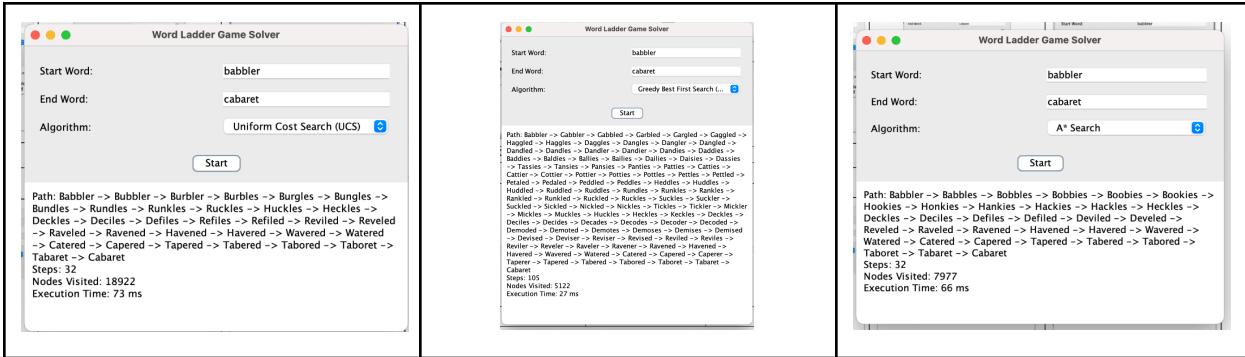


#### 4.6 Test Case 6



#### 4.7 Test Case 7





## 4.8 Test Case 8

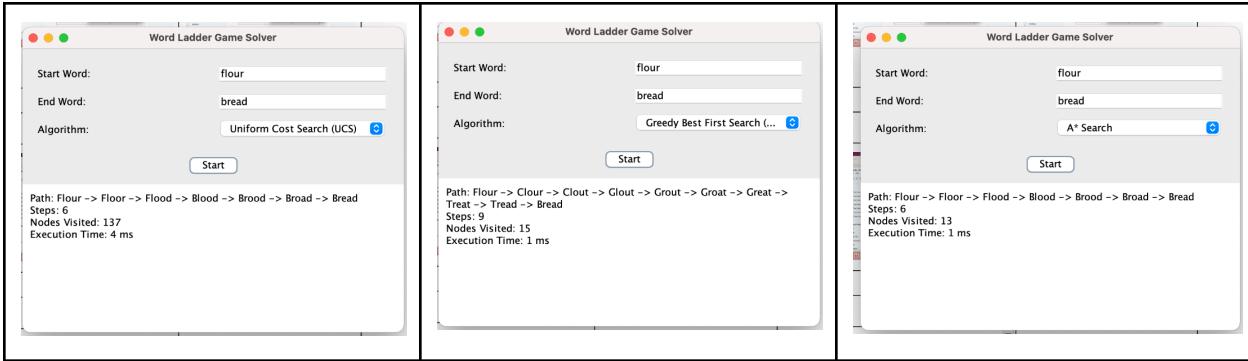
UCS	GBFS	A*
<p>No path found between COLORFUL and DELICATE Nodes Visited: 1 Execution Time: 12 ms</p>	<p>No path found between COLORFUL and DELICATE Nodes Visited: 1 Execution Time: 3 ms</p>	<p>No path found between COLORFUL and DELICATE Nodes Visited: 1 Execution Time: 1 ms</p>

## 4.9 Test Case 9

UCS	GBFS	A*
<p>Path: Kettle -&gt; Settle -&gt; Settee -&gt; Setter -&gt; Better -&gt; Belter -&gt; Bolter -&gt; Bolder -&gt; Holder Steps: 8 Nodes Visited: 3887 Execution Time: 15 ms</p>	<p>Path: Kettle -&gt; Fettle -&gt; Settle -&gt; Settee -&gt; Setter -&gt; Better -&gt; Belter -&gt; Bolter -&gt; Bolder -&gt; Holder Steps: 9 Nodes Visited: 11 Execution Time: 0 ms</p>	<p>Path: Kettle -&gt; Settle -&gt; Settee -&gt; Setter -&gt; Better -&gt; Belter -&gt; Bolter -&gt; Bolder -&gt; Holder Steps: 8 Nodes Visited: 43 Execution Time: 2 ms</p>

## 4.10 Test Case 10

UCS	GBFS	A*



## **BAB V**

### **ANALISIS PERBANDINGAN SOLUSI**

Berdasarkan 10 pengujian yang dilakukan, terdapat beberapa hal yang dapat disimpulkan mengenai ketiga algoritma, yaitu:

#### 1. Optimalitas

Dalam konteks optimalitas, algoritma UCS dan A\* menghasilkan path terpendek atau solusi yang paling optimal. GBFS gagal menghasilkan path terpendek, kecuali pada test case 2. Hal ini dikarenakan GBFS yang tidak mempertimbangkan total biaya kumulatif. Algoritma ini memilih jalur yang mungkin ‘terlihat’ lebih dekat ke tujuan menurut nilai heuristik, tetapi bisa saja memiliki biaya total yang lebih tinggi. Urutan optimalitas adalah UCS = A\* > GBFS, dimana UCS dan A\* paling optimal.

#### 2. Waktu eksekusi

Pada kebanyakan test case (kecuali test case 4, 8, dan 10 dimana A\* lebih cepat atau sama), GBFS cenderung lebih cepat atau memiliki waktu eksekusi yang lebih kecil daripada algoritma lainnya. GBFS lebih cepat menemukan solusi karena hanya bergantung pada heuristik yang memperkirakan jarak ke tujuan. Urutan kecepatan berdasarkan waktu eksekusi pada bagian hasil pengujian adalah GBFS > A\* > UCS, dimana GBFS paling cepat dan UCS paling lambat. UCS cenderung lebih lambat karena perlu mencari semua node tanpa perkiraan heuristik.

#### 3. Memori

GBFS paling hemat memori. Hal ini dapat dilihat dari banyaknya node yang dikunjungi oleh masing-masing algoritma. Pada kebanyakan test case (kecuali test case 4 dan 10 dimana A\* memiliki jumlah visited nodes lebih sedikit), GBFS memiliki jumlah *visited node* paling sedikit. Jika diurutkan dari yang paling hemat memori, maka urutannya adalah GBFS > A\* > UCS.

Secara keseluruhan, Algoritma A\* terbukti lebih efisien baik dari segi waktu eksekusi maupun penggunaan memori, sambil tetap memberikan solusi optimal. UCS memakan waktu paling lama dan juga menggunakan memori besar karena mengeksplorasi semua node. Sementara, GBFS gagal untuk mendapatkan hasil optimal.

## BAB VI

### LAMPIRAN

#### 6.1 Link Repository

[https://github.com/zyaaa-aaa/Tucil3\\_13522063.git](https://github.com/zyaaa-aaa/Tucil3_13522063.git)

#### 6.2 Tabel Kelengkapan

Poin	Ya	Tidak
1. Program berhasil dijalankan.	<input checked="" type="checkbox"/>	<input type="checkbox"/>
2. Program dapat menemukan rangkaian kata dari <i>start word</i> ke <i>end word</i> sesuai aturan permainan dengan algoritma UCS.	<input checked="" type="checkbox"/>	<input type="checkbox"/>
3. Solusi yang diberikan pada algoritma UCS optimal.	<input checked="" type="checkbox"/>	<input type="checkbox"/>
4. Program dapat menemukan rangkaian kata dari <i>start word</i> ke <i>end word</i> sesuai aturan permainan dengan algoritma <i>Greedy Best First Search</i> .	<input checked="" type="checkbox"/>	<input type="checkbox"/>
5. Program dapat menemukan rangkaian kata dari <i>start word</i> ke <i>end word</i> sesuai aturan permainan dengan algoritma A*.	<input checked="" type="checkbox"/>	<input type="checkbox"/>
6. Solusi yang diberikan pada algoritma A* optimal.	<input checked="" type="checkbox"/>	<input type="checkbox"/>
7. [Bonus]: Program memiliki tampilan GUI.	<input checked="" type="checkbox"/>	<input type="checkbox"/>