

TUGAS KECIL 3
IF2211 STRATEGI ALGORITMA
Penyelesaian Permainan Word Ladder Menggunakan Algoritma
UCS, Greedy Best First
Search, dan A*



DISUSUN OLEH:
Jonathan Emmanuel Saragih **13522121**

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2024

DAFTAR ISI

DAFTAR ISI	1
BAB I	2
DESKRIPSI MASALAH DAN ALGORITMA	2
1.1 Algoritma Greedy Best-First Search	2
1.2 Algoritma Uniform Cost Search (UCS)	2
1.3 Algoritma A*(A-Star) Search serta Penjelasan $f(n)$ dan $g(n)$	2
1.4 Admissibility of Heuristic pada A*	3
BAB II	5
IMPLEMENTASI DAN ANALISIS ALGORITMA DALAM BAHASA JAVA	5
2.1 WordLadderSolver.java	5
2.2 DictionaryLoader.java	6
2.3 Main.java	6
BAB III	7
SOURCE CODE PROGRAM	7
3.1 Repository Program	7
3.2 Source Code Program	7
3.2.1. WordLadderSolver.java	7
3.2.2. DictionaryLoader.java	11
3.2.3. Main.java	12
BAB IV	15
UJI COBA PROGRAM	15
4.2 Set Percobaan Kedua	16
4.3 Set Percobaan Ketiga	17
4.4 Set Percobaan Keempat	19
4.5 Set Percobaan Kelima	21
4.6 Set Percobaan Keenam	22
BAB V	24
ANALISIS, SARAN, DAN KESIMPULAN	24
5.1 Analisis Efektivitas Algoritma	24
5.2 Saran	25
5.3 Kesimpulan	25
5.4 Refleksi	25
LAMPIRAN	27

BAB I

DESKRIPSI MASALAH DAN ALGORITMA

1.1 Algoritma Greedy Best-First Search

Algoritma Greedy Best-First Search merupakan teknik pada program yang akan melakukan pencarian yang menggunakan heuristik untuk mengarahkan pencarian ke tujuan dengan mengambil langkah yang tampaknya paling mendekatkan ke tujuan. Dalam algoritma ini, setiap langkah diambil berdasarkan pemilihan yang terbaik pada saat itu, tanpa mempertimbangkan langkah sebelumnya atau implikasi langkah berikutnya. Ini sering digunakan dalam masalah seperti pencarian jalur, optimasi, dan konfigurasi game.

Algoritma ini memiliki keunggulan yaitu kemampuan untuk menemukan solusi dengan cepat dalam banyak kasus karena algoritma ini akan langsung bergerak ke arah yang paling menguntungkan berdasarkan heuristiknya. Hal ini akan membuat algoritma menjadi sangat cepat dalam banyak skenario, terutama di ruang pencarian yang besar di mana langkah yang optimal pada setiap titik adalah langkah yang benar. Namun, kelemahannya adalah algoritma ini akan sulit untuk menemukan jalur terpendek atau solusi optimal karena hanya terfokus pada tujuan tanpa mempertimbangkan total usaha atau banyaknya langkah yang harus diambil.

1.2 Algoritma Uniform Cost Search (UCS)

Algoritma Uniform Cost Search adalah algoritma yang mengabaikan atau tidak memprioritaskan informasi mengenai lokasi tujuan dan hanya berfokus pada penjelajahan jalur dengan jarak terkecil atau usaha terendah. UCS adalah bentuk khusus dari Breadth-First Search yang menggunakan priority queue untuk memperluas node dengan biaya kumulatif terendah.

UCS efektif dalam menemukan jalur biaya minimum, namun algoritma ini memiliki kelemahan yaitu program bisa berjalan sangat lambat, terutama ketika ruang pencarian luas karena UCS secara sistematis akan menjelajah atau mengeksplorasi semua jalur yang mungkin tanpa memanfaatkan heuristik untuk mengarahkan pencarian.

1.3 Algoritma A*(A-Star) Search serta Penjelasan $f(n)$ dan $g(n)$

Algoritma A* (A-star) merupakan salah satu algoritma pencarian jalur yang paling terkenal dan banyak digunakan. Algoritma ini akan menggabungkan keunggulan dari algoritma Greedy Best-First Search dan Uniform Cost Search. Algoritma ini menggunakan fungsi penilaian $f(n) = g(n) + h(n)$ untuk setiap node n , di mana $g(n)$ adalah biaya dari titik awal ke n , dan $h(n)$ adalah heuristik—estimasi biaya dari n ke tujuan.

Algoritma A* memiliki keunggulan yaitu memiliki efisiensi yang tinggi dalam mencari jalur terpendek. Hal tersebut bisa terjadi karena algoritma ini mengintegrasikan informasi dari usaha yang telah dikeluarkan dan perkiraan jarak ke tujuan. Penggunaan heuristik yang baik dan admissible (tidak pernah overestimate biaya sebenarnya ke tujuan) membuat algoritma A* cepat dan optimal.

Dalam A*, digunakan $g(n)$, $f(n)$, dan $h(n)$ untuk membantu berejalannya algoritma ini. $g(n)$ sendiri memiliki definisi yaitu fungsi untuk menghitung biaya atau usaha total dari node awal ke node n . Biaya atau usaha ini adalah akumulasi dari semua yang dikeluarkan untuk mencapai node n dari node awal. $h(n)$ merupakan fungsi untuk menghitung heuristik yang memperkirakan biaya atau usaha terendah dari node n ke tujuan. Heuristik ini harus *admissible*, artinya tidak boleh overestimate biaya sebenarnya untuk mencapai tujuan dari n . Untuk $f(n)$ sendiri, $f(n)$ memiliki definisi yaitu sebagai fungsi evaluasi yang digunakan oleh A* untuk mengurutkan node-node dalam priority queue. Fungsi ini didefinisikan sebagai jumlah dari $g(n)$ dan $h(n)$.

$$f(n) = g(n) + h(n)$$

Dengan adanya fungsi tersebut, $f(n)$ memberikan estimasi total biaya minimum dari node awal melalui node n hingga mencapai tujuan. Node dengan nilai $f(n)$ terendah adalah yang pertama dieksplorasi, karena teorinya adalah jalur dengan biaya terendah yang diperkirakan menuju tujuan. Dengan adanya hal - hal tersebut, algoritma A* secara teori dianggap memiliki algoritma yang sangat efisien dan efektif dalam kasus pencarian jalur.

1.4 Admissibility of Heuristic pada A*

Admissibility of heuristic pada algoritma A* merupakan sifat yang memberikan kepastian bahwa heuristic yang digunakan tidak akan pernah melebihi biaya sebenarnya dari titik awal ke titik tujuan. Dengan kata lain, heuristic yang admissible akan selalu memperkirakan biaya terpendek yang sesungguhnya untuk mencapai solusi.

Admissibility of heuristic sangat penting dalam algoritma A* karena memastikan bahwa algoritma akan menemukan jalur optimal jika heuristic yang digunakan adalah admissible. Jika heuristic tidak admissible, maka A* tidak lagi menjamin optimalitas solusi yang ditemukan. Sebagai contoh, jika heuristic menghasilkan estimasi yang terlalu tinggi, A* mungkin mengabaikan jalur yang sebenarnya lebih efisien karena mengira bahwa jalur tersebut lebih mahal berdasarkan nilai heuristic yang diberikan.

1.5 Perbedaan UCS dan BFS

Dalam kasus Word Ladder, algoritma UCS (Uniform Cost Search) dan BFS (Breadth-First Search) tidaklah sama. Kedua algoritma ini memiliki perbedaan dalam cara memilih simpul berikutnya untuk dieksplorasi atau dilewati. Namun, jika implementasinya ditujukan untuk menemukan jalur terpendek antara dua kata dalam Word Ladder, kedua algoritma tersebut akan menghasilkan jalur yang sama jika digunakan dengan benar.

Perbedaan utama antara UCS dan BFS terletak pada cara kedua algoritma tersebut menangani jarak perpindahan antar simpul. BFS cenderung mengeksplorasi simpul secara berurutan berdasarkan kedalaman dalam graf, sementara UCS akan mempertimbangkan biaya dari simpul awal ke simpul saat ini. Dengan demikian, UCS lebih fleksibel karena memperhitungkan biaya perpindahan antar simpul.

1.6 Efisiensi Algoritma

Secara teoritis, algoritma A* akan memiliki algoritma program yang lebih efisien jika dibandingkan dengan algoritma UCS. Hal ini disebabkan oleh penggunaan heuristic pada algoritma A*, yang memungkinkan pencarian jalur terpendek dengan lebih cepat. Algoritma UCS (Uniform Cost Search) memerlukan pencarian ke semua kemungkinan jalur yang ada dari simpul awal ke simpul tujuan, tanpa mempertimbangkan informasi tambahan tentang lokasi relatif atau kemungkinan lebih jauh dari simpul tujuan. Hal ini menyebabkan algoritma UCS akan mengeksplorasi simpul berdasarkan biaya perpindahan antar simpul tanpa mempertimbangkan jarak sisa ke simpul tujuan, sementara itu algoritma A* menggunakan heuristic untuk memperkirakan biaya atau usaha yang diperlukan dari simpul saat ini ke simpul tujuan. Dengan memanfaatkan informasi ini, A* cenderung akan melakukan eksplorasi ke arah yang lebih menjanjikan. Dengan adanya hal - hal tersebut, algoritma A* cenderung lebih efisien.

Secara teoritis, dalam persoalan WordLadder, algoritma Greedy Best First Search tidak menjamin solusi optimal. Algoritma Greedy Best First Search memilih simpul berikutnya berdasarkan nilai heuristic saja, tanpa mempertimbangkan biaya atau usaha yang dikeluarkan sebelumnya dari simpul awal ke simpul saat ini. Dengan demikian, algoritma ini cenderung

memprioritaskan melewati jalur simpul yang paling dekat dengan simpul tujuan berdasarkan nilai heuristiknya.

Dalam konteks Word Ladder, algoritma Greedy Best First Search cenderung memilih simpul yang memiliki jumlah karakter yang paling sedikit berbeda dengan kata tujuan, tanpa mempertimbangkan jumlah langkah atau biaya total yang telah ditempuh. Karena itu, algoritma ini tidak dapat menjamin akan selalu menghasilkan jalur atau path yang optimal.

BAB II

IMPLEMENTASI DAN ANALISIS ALGORITMA DALAM BAHASA JAVA

Dalam pembuatan program WordLadder untuk tugas ini, saya menggunakan bahasa pemrograman Java untuk programnya dan menggunakan LinuxSwing untuk GUInya. Berikut adalah penjelasan lebih detail mengenai programnya.

2.1 WordLadderSolver.java

Program class ini merupakan program yang berisi implementasi dari algoritma Greedy, UCS, dan A*. Algoritma ini memiliki tanggungjawab untuk menyelesaikan permasalahan WordLadder dan mencari jalur dengan algoritma yang dibuat.

Tabel 1 Method Table WordLadderSolver.java

Methods	Description
findPath(String start, String end, String algorithm)	Metode ini menerima kata awal, kata akhir, dan algoritma yang akan digunakan untuk menemukan jalur. Berdasarkan algoritma yang dipilih, metode ini memanggil metode khusus yang sesuai.
uniformCostSearch(String start, String end)	Metode ini mengimplementasikan algoritma UCS untuk menemukan jalur terpendek antara dua kata.
greedyBestFirstSearch(String start, String end)	Metode ini mengimplementasikan algoritma Greedy Best First Search untuk menemukan jalur dengan memilih simpul berikutnya berdasarkan nilai heuristik.
aStar(String start, String end)	Metode ini mengimplementasikan algoritma A* untuk menemukan jalur dengan mempertimbangkan biaya dari simpul awal hingga simpul saat ini serta nilai heuristik.

Tabel 2 Variable Table bruteforce page

Class	Description
Node	Kelas ini merepresentasikan simpul dalam pencarian. Setiap simpul memiliki kata, simpul induk, dan biaya dari awal hingga simpul saat ini.

2.2 DictionaryLoader.java

Program kelas ini merupakan kelas yang memiliki tanggungjawab untuk memuat atau membaca kamus kata kata dari file teks.

Method	Description
loadDictionary(String filePath)	Metode ini membaca file kata-kata dan memuatnya ke dalam sebuah himpunan (Set).

2.3 Main.java

Program ini merupakan program interface user yang menggunakan grafis (GUI) untuk menggunakan WordLadder.

Metode	Description
main(String[] argos)	Metode utama untuk menjalankan program. Ini membuat jendela GUI dengan kolom input untuk kata awal, kata akhir, pilihan algoritma, dan tombol "Find Path"

BAB III

SOURCE CODE PROGRAM

3.1 Repository Program

Repository program dapat diakses melalui pranala *GitHub* berikut:
https://github.com/JonathanSaragih/Tucil3_13522121

3.2 Source Code Program

3.2.1. WordLadderSolver.java

```
package src;

import java.util.*;

public class WordLadderSolver {
    private Set<String> dictionary;
    private int nodesVisited;

    public WordLadderSolver(Set<String> dictionary) {
        this.dictionary = dictionary;
    }

    public List<String> findPath(String start, String end, String algorithm) {
        nodesVisited = 0; // Reset the visited nodes counter at the start of
each path find
        switch (algorithm.toLowerCase()) {
            case "ucs":
                return uniformCostSearch(start, end);
            case "greedy best first search":
                return greedyBestFirstSearch(start, end);
            case "astar":
                return aStar(start, end);
            default:
                return new ArrayList<>();
        }
    }

    private List<String> uniformCostSearch(String start, String end) {
        PriorityQueue<Node> openSet = new
PriorityQueue<>(Comparator.comparingInt(n -> n.cost));
        Map<String, Node> allNodes = new HashMap<>();
        Node startNode = new Node(start, null, 0);
        openSet.add(startNode);
        allNodes.put(start, startNode);

        while (!openSet.isEmpty()) {
            Node current = openSet.poll();
            nodesVisited++; // Count each node when it is expanded

            if (current.word.equals(end)) {
                return reconstructPath(current);
            }

            for (String neighbor : getNeighbors(current.word)) {
```



```

        int newCost = current.cost + 1;
        Node neighborNode = allNodes.get(neighbor);

        if (neighborNode == null || newCost < neighborNode.cost) {
            if (neighborNode == null) {
                neighborNode = new Node(neighbor, current, newCost);
                allNodes.put(neighbor, neighborNode);
                openSet.add(neighborNode);
            } else {
                openSet.remove(neighborNode);
                neighborNode.parent = current;
                neighborNode.cost = newCost;
                openSet.add(neighborNode);
            }
        }
    }

    return new ArrayList<>();
}

private List<String> greedyBestFirstSearch(String start, String end) {
    PriorityQueue<Node> openSet = new
PriorityQueue<>(Comparator.comparingInt(n -> heuristic(n.word, end)));
    Map<String, Node> allNodes = new HashMap<>();
    Node startNode = new Node(start, null, 0);
    openSet.add(startNode);
    allNodes.put(start, startNode);

    while (!openSet.isEmpty()) {
        Node current = openSet.poll();
        nodesVisited++; // Count each node when it is expanded

        if (current.word.equals(end)) {
            return reconstructPath(current);
        }

        for (String neighbor : getNeighbors(current.word)) {
            if (!allNodes.containsKey(neighbor)) {
                Node neighborNode = new Node(neighbor, current, current.cost
+ 1);

                openSet.add(neighborNode);
                allNodes.put(neighbor, neighborNode);
            }
        }
    }

    return new ArrayList<>();
}

private List<String> aStar(String start, String end) {
    PriorityQueue<Node> openSet = new PriorityQueue<>(
        Comparator.comparingInt(n -> n.cost + heuristic(n.word, end)));
    Map<String, Node> allNodes = new HashMap<>();
    Node startNode = new Node(start, null, 0);
    openSet.add(startNode);
    allNodes.put(start, startNode);
}

```

```

while (!openSet.isEmpty()) {
    Node current = openSet.poll();
    nodesVisited++; // Count each node when it is expanded

    if (current.word.equals(end)) {
        return reconstructPath(current);
    }

    for (String neighbor : getNeighbors(current.word)) {
        int newCost = current.cost + 1;
        Node neighborNode = allNodes.get(neighbor);

        if (neighborNode == null || newCost < neighborNode.cost) {
            if (neighborNode == null) {
                neighborNode = new Node(neighbor, current, newCost);
                allNodes.put(neighbor, neighborNode);
                openSet.add(neighborNode);
            } else {
                openSet.remove(neighborNode);
                neighborNode.parent = current;
                neighborNode.cost = newCost;
                openSet.add(neighborNode);
            }
        }
    }
}

return new ArrayList<>();
}

private List<String> getNeighbors(String word) {
    List<String> neighbors = new ArrayList<>();
    char[] chars = word.toCharArray();
    for (int i = 0; i < word.length(); i++) {
        char oldChar = chars[i];
        for (char c = 'a'; c <= 'z'; c++) {
            if (c == oldChar)
                continue;
            chars[i] = c;
            String newWord = new String(chars);
            if (dictionary.contains(newWord)) {
                neighbors.add(newWord);
            }
        }
        chars[i] = oldChar;
    }
    return neighbors;
}

private int heuristic(String word, String end) {
    int mismatchCount = 0;
    for (int i = 0; i < word.length(); i++) {
        if (word.charAt(i) != end.charAt(i))
            mismatchCount++;
    }
    return mismatchCount;
}

```

```

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

public int getNodesVisited() {
    return nodesVisited;
}

static class Node {
    String word;
    Node parent;
    int cost;

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

while (!openSet.isEmpty()) {
    Node current = openSet.poll();
    nodesVisited++;

    if (current.word.equals(end)) {
        return reconstructPath(current);
    }

    for (String neighbor : getNeighbors(current.word)) {
        int newCost = current.cost + 1;
        Node neighborNode = allNodes.get(neighbor);

        if (neighborNode == null || newCost < neighborNode.cost) {
            if (neighborNode == null) {
                neighborNode = new Node(neighbor, current, newCost);
                allNodes.put(neighbor, neighborNode);
                openSet.add(neighborNode);
            } else {
                openSet.remove(neighborNode);
                neighborNode.parent = current;
                neighborNode.cost = newCost;
                openSet.add(neighborNode);
            }
        }
    }

    return new ArrayList<>();
}

private List<String> getNeighbors(String word) {

```

```

        List<String> neighbors = new ArrayList<>();
        char[] chars = word.toCharArray();
        for (int i = 0; i < word.length(); i++) {
            char oldChar = chars[i];
            for (char c = 'a'; c <= 'z'; c++) {
                if (c == oldChar)
                    continue;
                chars[i] = c;
                String newWord = new String(chars);
                if (dictionary.contains(newWord)) {
                    neighbors.add(newWord);
                }
            }
            chars[i] = oldChar;
        }
        return neighbors;
    }

    private int heuristic(String word, String end) {
        int mismatchCount = 0;
        for (int i = 0; i < word.length(); i++) {
            if (word.charAt(i) != end.charAt(i))
                mismatchCount++;
        }
        return mismatchCount;
    }

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

    public int getNodesVisited() {
        return nodesVisited;
    }

    static class Node {
        String word;
        Node parent;
        int cost;

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

```

3.2.2. DictionaryLoader.java

```

package src;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashSet;
import java.util.Set;

public class DictionaryLoader {
    public static Set<String> loadDictionary(String filePath) throws IOException
    {
        Set<String> dictionary = new HashSet<>();
        try (BufferedReader reader = new BufferedReader(new
FileReader(filePath))) {
            String word;
            while ((word = reader.readLine()) != null) {
                dictionary.add(word.trim().toLowerCase());
            }
        }
        return dictionary;
    }
}

```

3.2.3. Main.java

```

package src;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.util.List;

public class Main {
    private static JTextField startWordField;
    private static JTextField endWordField;
    private static JComboBox<String> algorithmComboBox;
    private static JTextArea resultArea;

    public static void main(String[] args) {
        JFrame frame = new JFrame("Word Ladder Solver");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);
        frame.setLayout(new BorderLayout());
        frame.getContentPane().setBackground(Color.WHITE);

        JPanel inputPanel = new JPanel(new GridLayout(4, 2, 10, 10));
        inputPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
        inputPanel.setBackground(Color.WHITE);

        inputPanel.add(new JLabel("Start Word:"));
    }
}

```

```

startWordField = new JTextField();
inputPanel.add(startWordField);

inputPanel.add(new JLabel("End Word:"));
endWordField = new JTextField();
inputPanel.add(endWordField);

inputPanel.add(new JLabel("Algorithm:"));
String[] algorithms = { "UCS", "Greedy Best First Search", "AStar" };
algorithmComboBox = new JComboBox<>(algorithms);
inputPanel.add(algorithmComboBox);

JButton findPathButton = new JButton("Find Path");
findPathButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        findPath();
    }
});
inputPanel.add(findPathButton);

frame.add(inputPanel, BorderLayout.NORTH);

resultArea = new JTextArea();
resultArea.setEditable(false);
resultArea.setBackground(Color.LIGHT_GRAY);
resultArea.setFont(new Font("Arial", Font.PLAIN, 12));
frame.add(new JScrollPane(resultArea), BorderLayout.CENTER);

frame.setVisible(true);
}

private static void findPath() {
    String startWord = startWordField.getText().trim().toLowerCase();
    String endWord = endWordField.getText().trim().toLowerCase();
    String algorithm = (String) algorithmComboBox.getSelectedItem();

    if (!isValidWord(startWord) || !isValidWord(endWord) ||
startWord.length() != endWord.length()) {
        JOptionPane.showMessageDialog(null,
            "Please enter valid start and end words (only alphabetical
characters with the same length are allowed).",
            "Error", JOptionPane.ERROR_MESSAGE);
        return;
    }

    resultArea.setText("Searching...");

    try {
        long startTime = System.currentTimeMillis();
        WordLadderSolver solver = new
WordLadderSolver(DictionaryLoader.loadDictionary("src\\words.txt"));
        List<String> path = solver.findPath(startWord, endWord, algorithm);
        long endTime = System.currentTimeMillis();

        if (path.isEmpty()) {
            resultArea.setText("No path found.");
        } else {

```

```

        resultArea.setText("Path found: " + String.join(" -> ", path) +
"\n");
    }
    resultArea.append("Nodes visited: " + solver.getNodesVisited() +
"\n");
    resultArea.append("Time taken: " + (endTime - startTime) + " ms\n");
} catch (IOException ex) {
    JOptionPane.showMessageDialog(null, "Failed to load dictionary: " +
ex.getMessage(), "Error",
JOptionPane.ERROR_MESSAGE);
}
}

private static boolean isValidWord(String word) {
    return word.matches("[a-z]+");
}
}

```

BAB IV

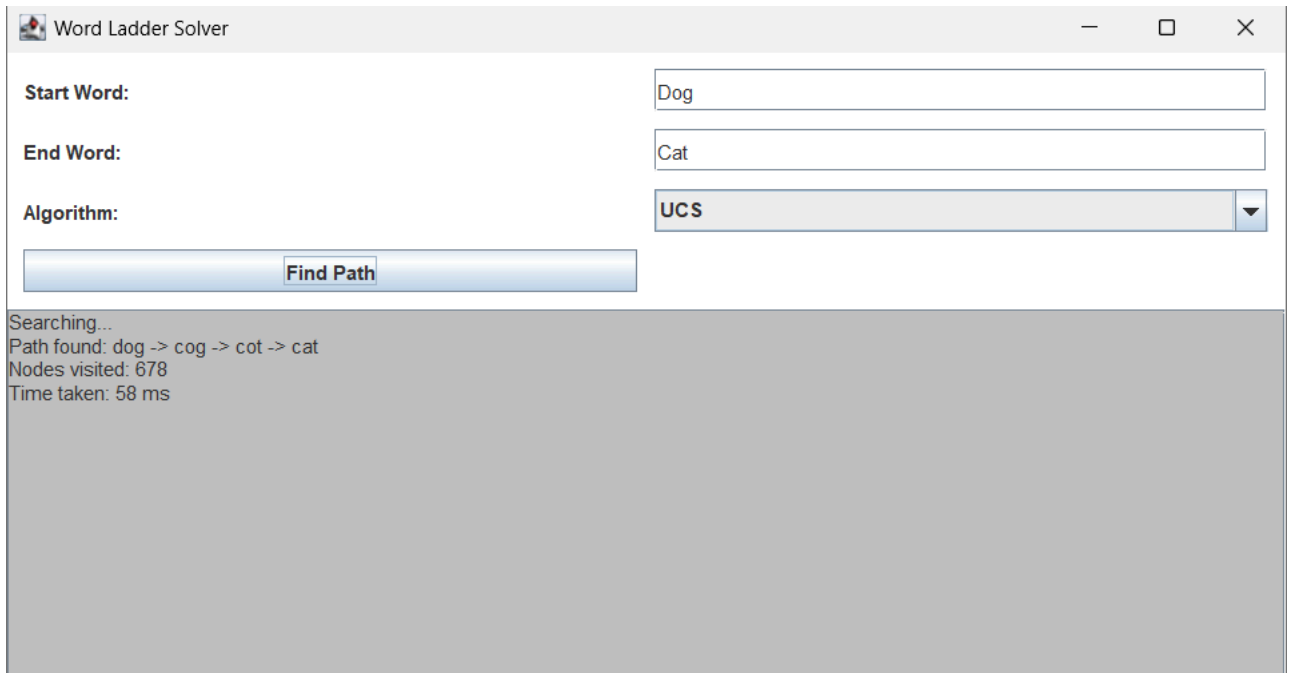
UJI COBA PROGRAM

Pada uji coba pada tugas ini, saya akan mencoba memasukkan testcase sebanyak 6 case untuk tiap algoritmanya. Akan dicatat keluaran serta lama waktu yang dibutuhkan program untuk mendapatkan jalur atau path yang dicari.

4.1 Set Percobaan Pertama

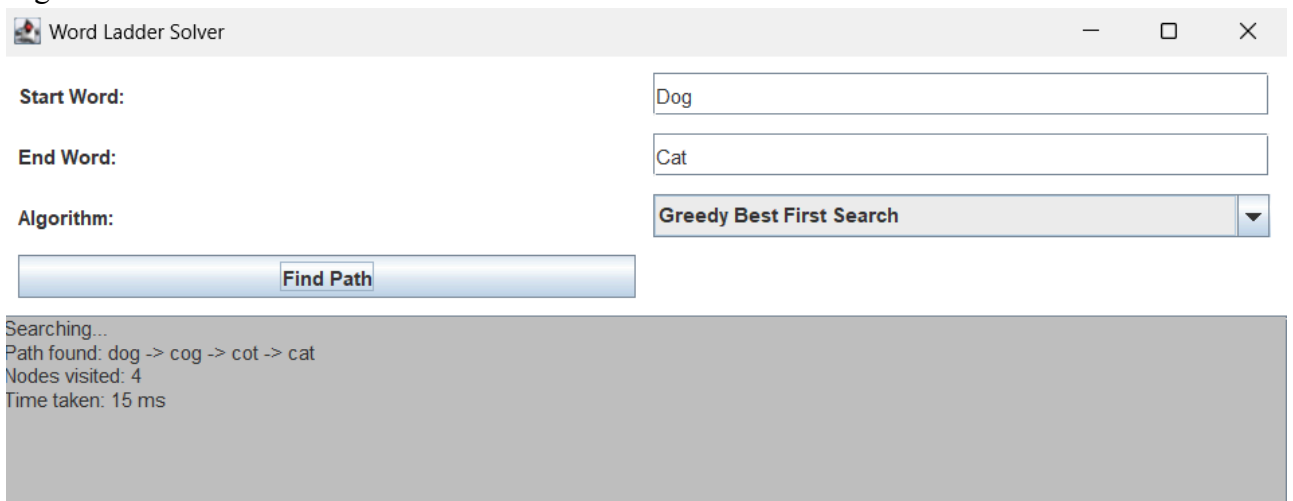
Testcase : Dog -> Cat

a) Algoritma UCS



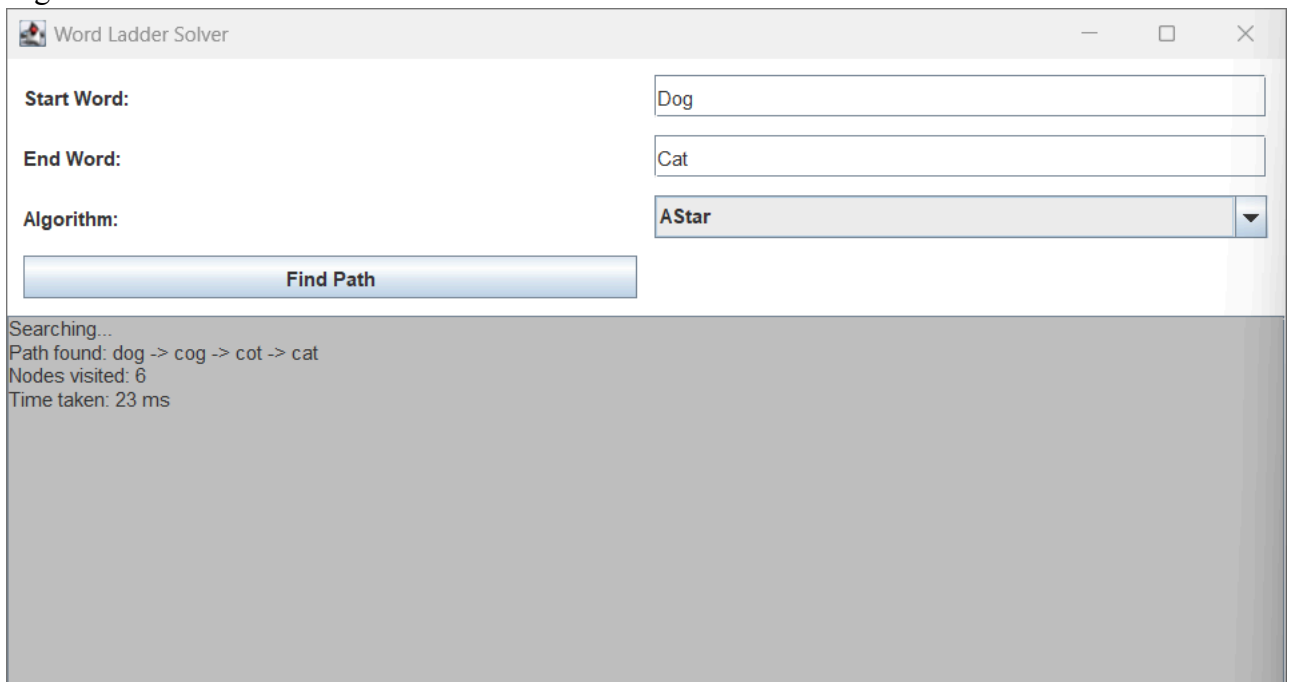
The screenshot shows a window titled "Word Ladder Solver". It has three input fields: "Start Word:" with "Dog", "End Word:" with "Cat", and "Algorithm:" with a dropdown menu set to "UCS". Below these is a "Find Path" button. The output area below the button displays the following text: "Searching...", "Path found: dog -> cog -> cot -> cat", "Nodes visited: 678", and "Time taken: 58 ms".

b) Algoritma GBFS



The screenshot shows the same "Word Ladder Solver" window, but the "Algorithm:" dropdown menu is now set to "Greedy Best First Search". The "Find Path" button is visible. The output area displays: "Searching...", "Path found: dog -> cog -> cot -> cat", "Nodes visited: 4", and "Time taken: 15 ms".

c) Algoritma A*



Word Ladder Solver

Start Word: Dog

End Word: Cat

Algorithm: AStar

Find Path

Searching...

Path found: dog -> cog -> cot -> cat

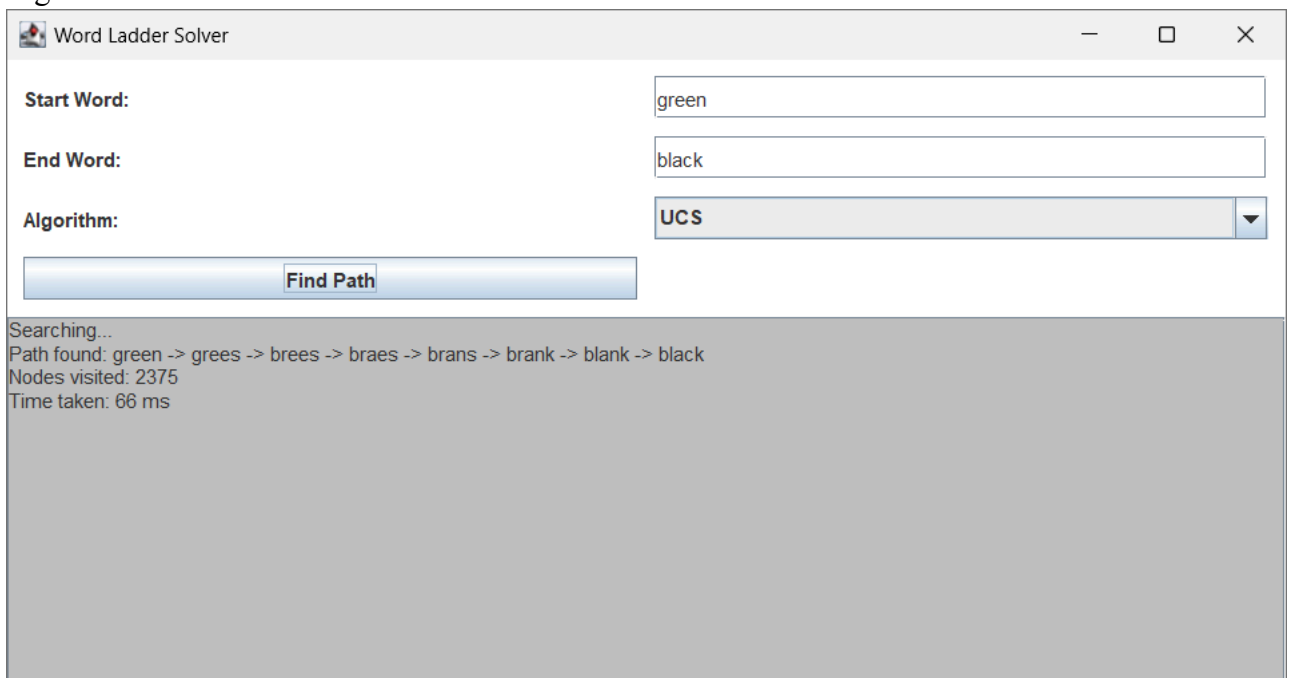
Nodes visited: 6

Time taken: 23 ms

4.2 Set Percobaan Kedua

Testcase : green, black

a) Algoritma UCS



Word Ladder Solver

Start Word: green

End Word: black

Algorithm: UCS

Find Path

Searching...

Path found: green -> grees -> breees -> braes -> brans -> brank -> blank -> black

Nodes visited: 2375

Time taken: 66 ms

b) Algoritma GBFS

Word Ladder Solver

Start Word:

End Word:

Algorithm: Greedy Best First Search

Searching...

Path found: green -> greek -> gleek -> gleed -> bleed -> blend -> bland -> blank -> black

Nodes visited: 15

Time taken: 23 ms

c) Algoritma A*

Word Ladder Solver

Start Word:

End Word:

Algorithm: AStar

Searching...

Path found: green -> grees -> breees -> braes -> brans -> brank -> blank -> black

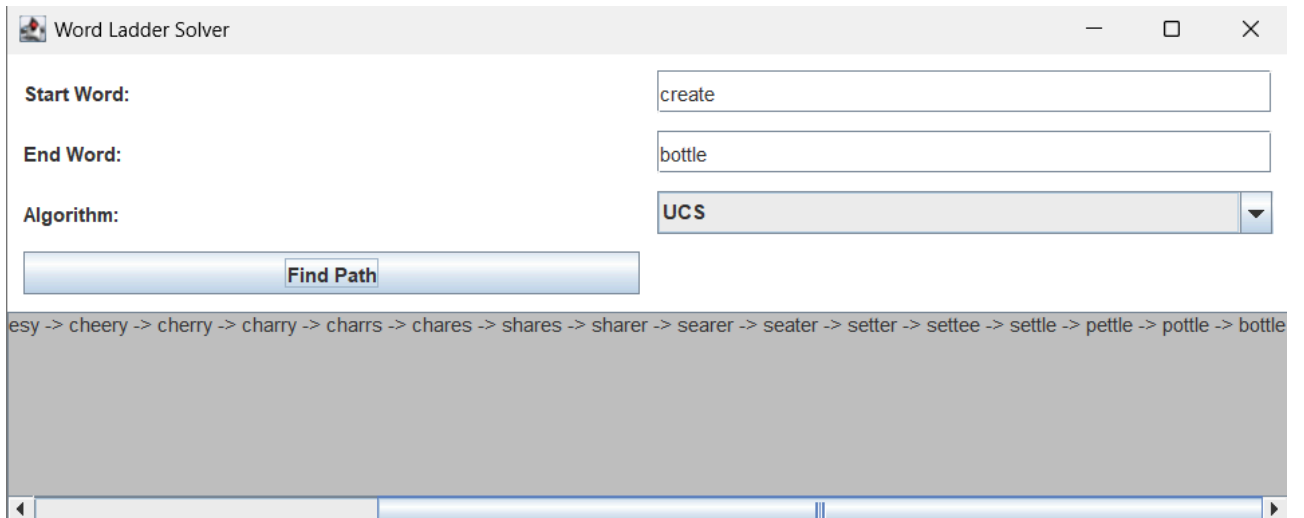
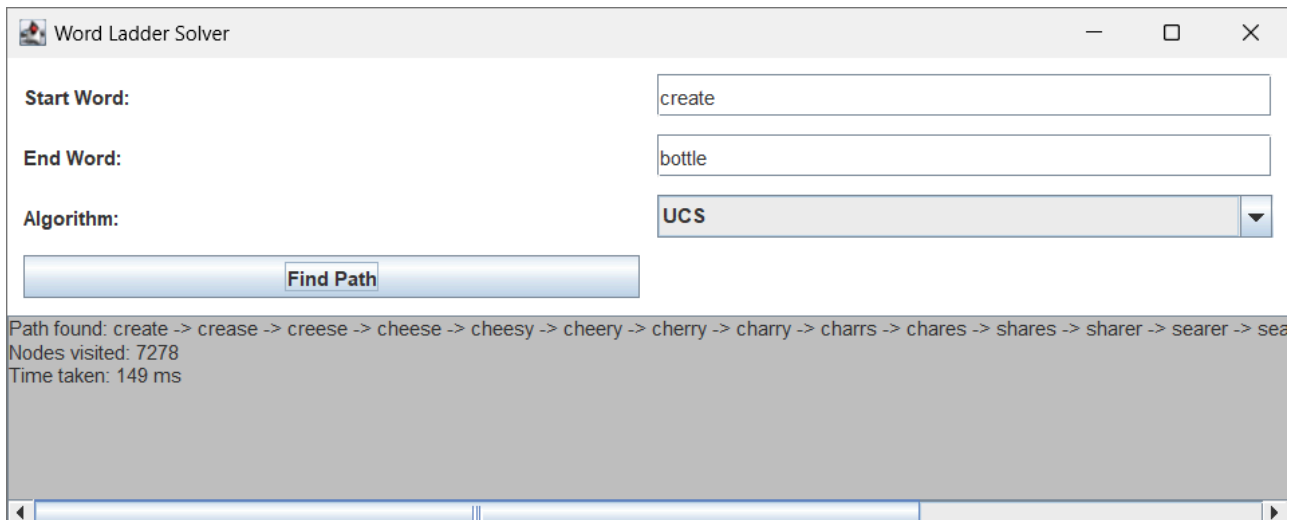
Nodes visited: 58

Time taken: 41 ms

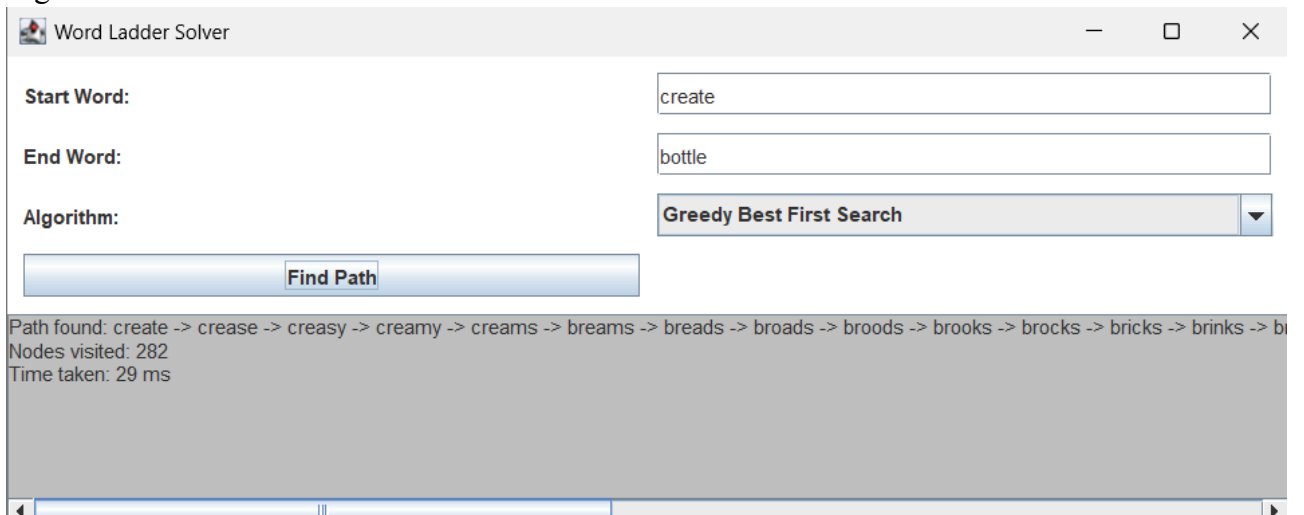
4.3 Set Percobaan Ketiga

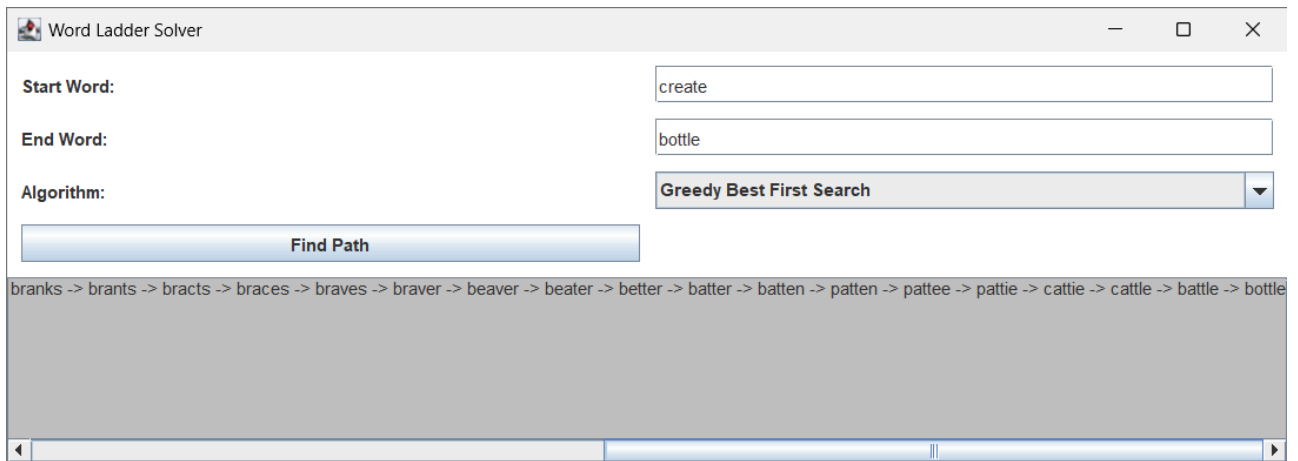
Testcase : create, bottle

a) Algoritma UCS

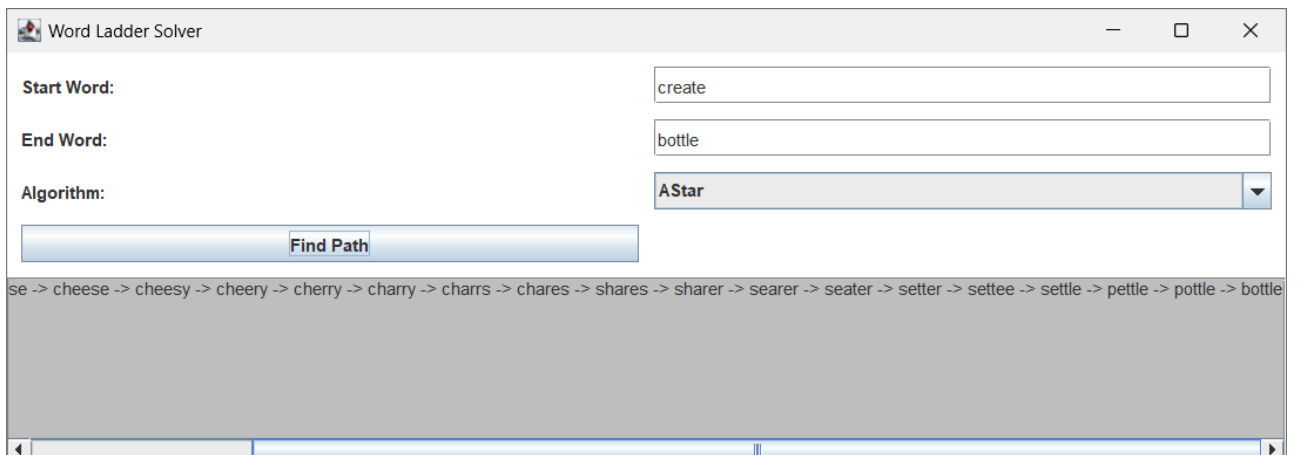
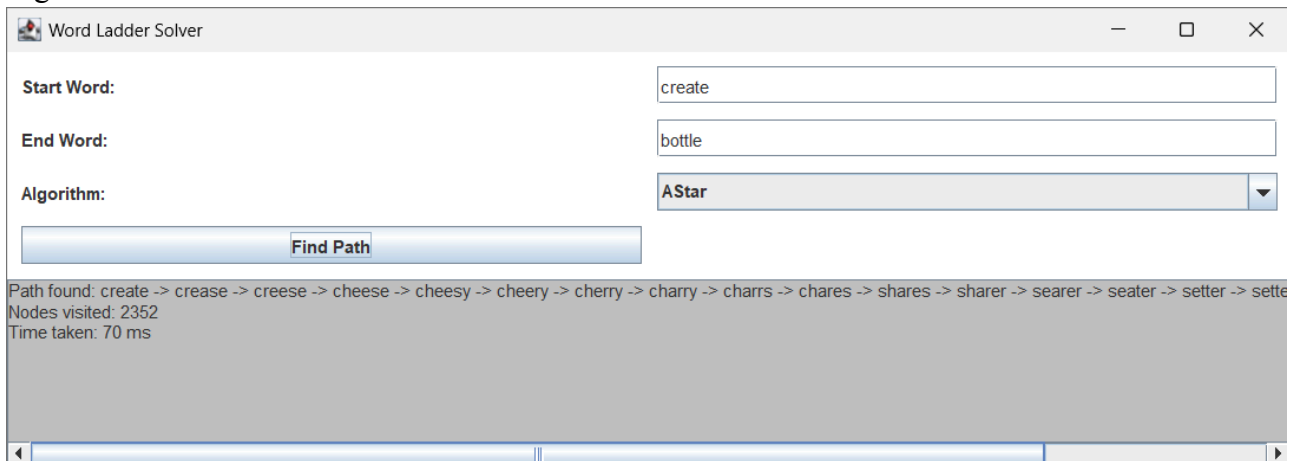


b) Algoritma GBFS





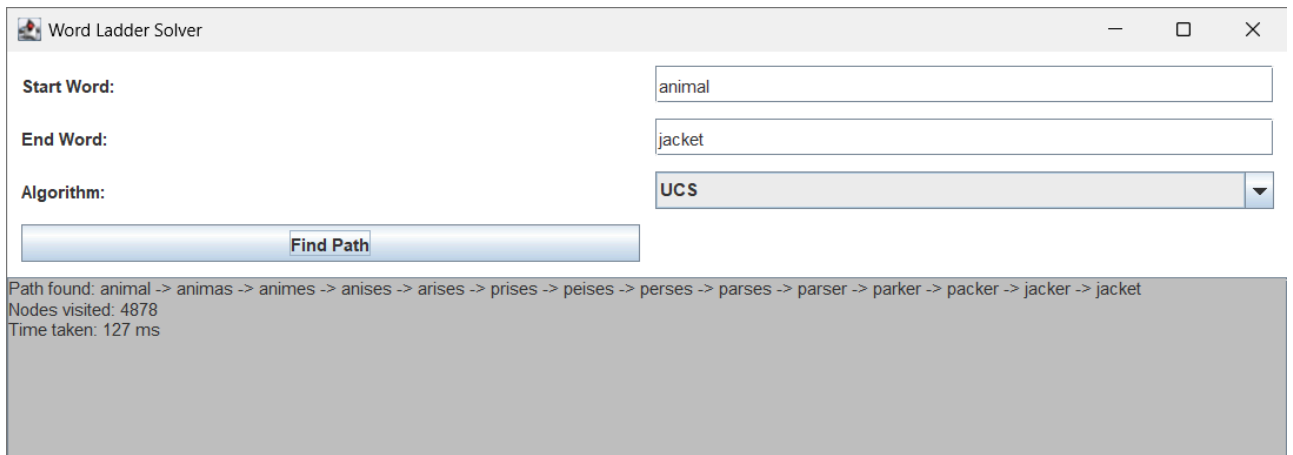
c) Algoritma A*



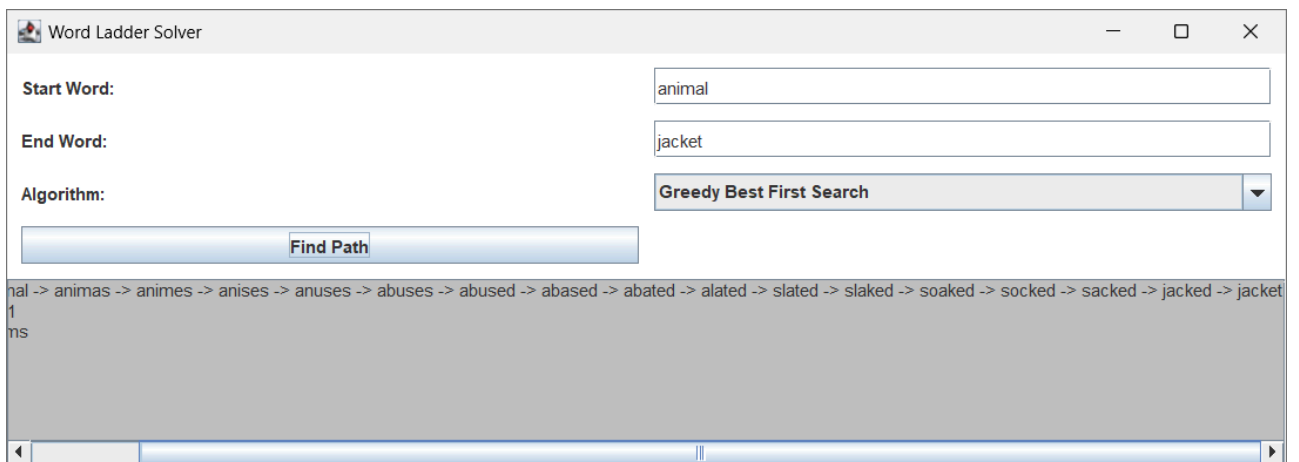
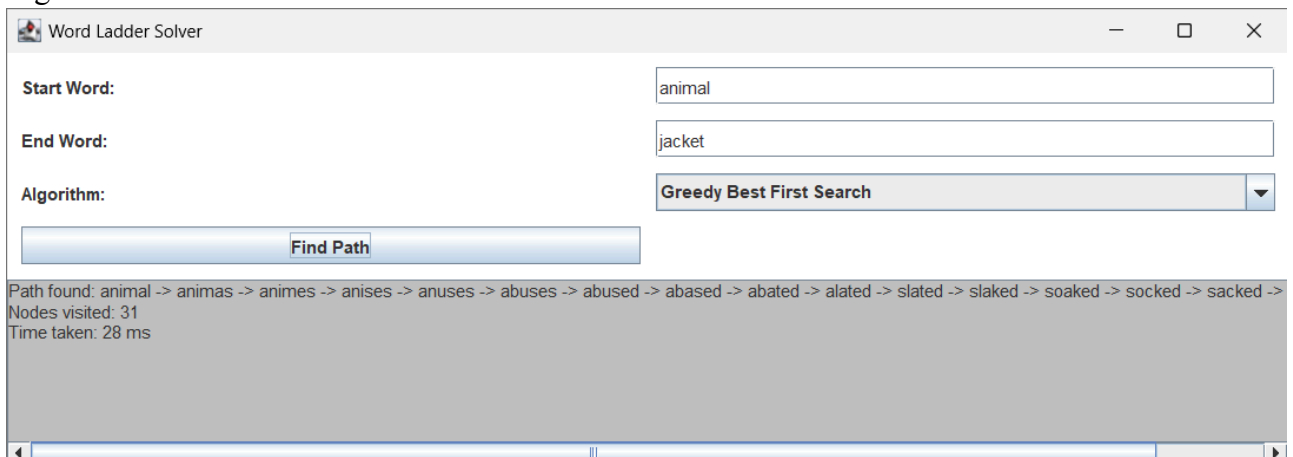
4.4 Set Percobaan Keempat

Testcase : Animal, jacket

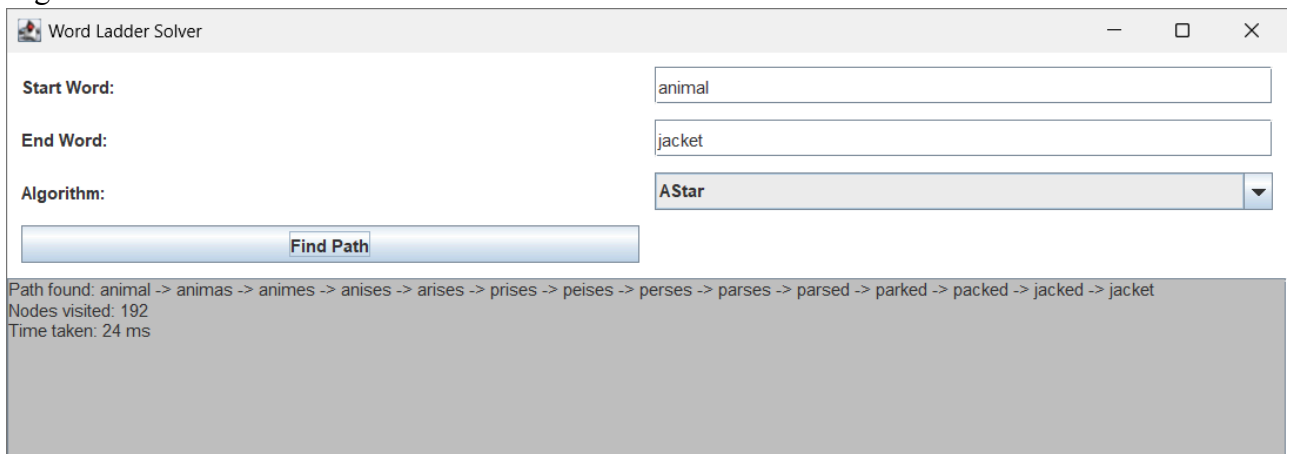
a) Algoritma UCS



b) Algoritma GBFS



c) Algoritma A*



Word Ladder Solver

Start Word: animal

End Word: jacket

Algorithm: AStar

Find Path

Path found: animal -> animas -> animes -> anises -> arises -> prises -> peises -> perses -> parses -> parsed -> parked -> packed -> jacked -> jacket

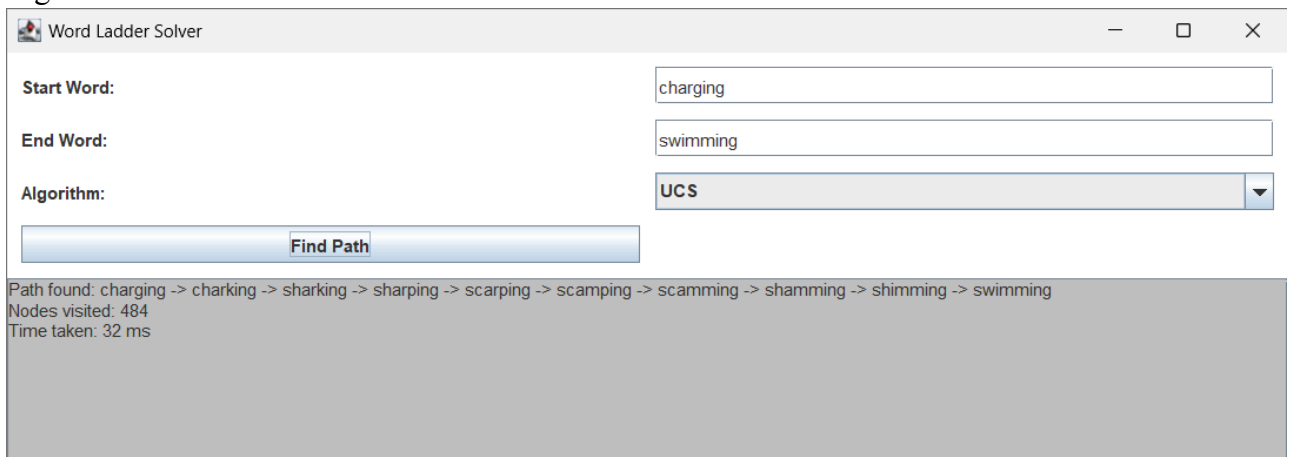
Nodes visited: 192

Time taken: 24 ms

4.5 Set Percobaan Kelima

Testcase : charging, swimming

a) Algoritma UCS



Word Ladder Solver

Start Word: charging

End Word: swimming

Algorithm: UCS

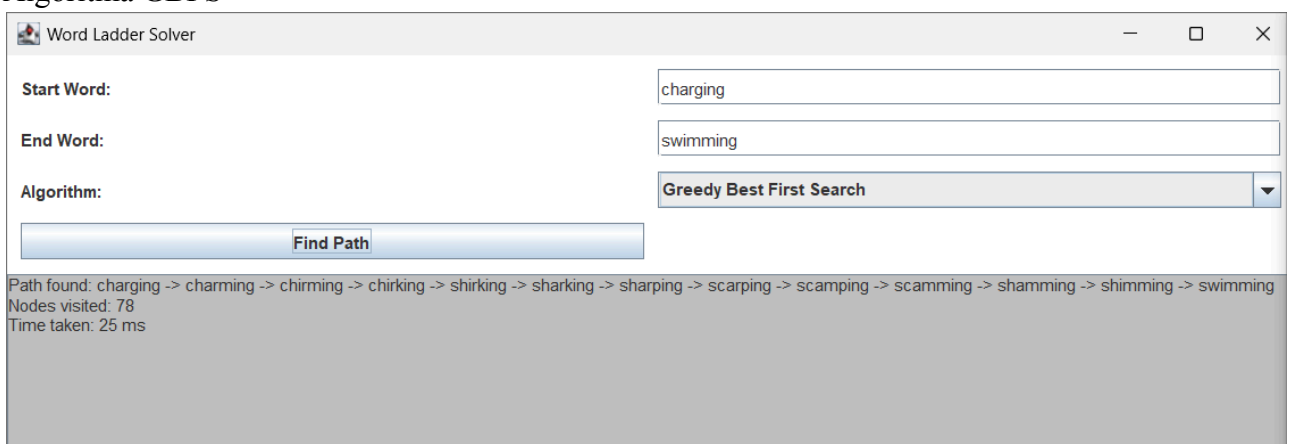
Find Path

Path found: charging -> charking -> sharking -> sharpening -> scarpening -> scampening -> scamming -> shamming -> shimming -> swimming

Nodes visited: 484

Time taken: 32 ms

b) Algoritma GBFS



Word Ladder Solver

Start Word: charging

End Word: swimming

Algorithm: Greedy Best First Search

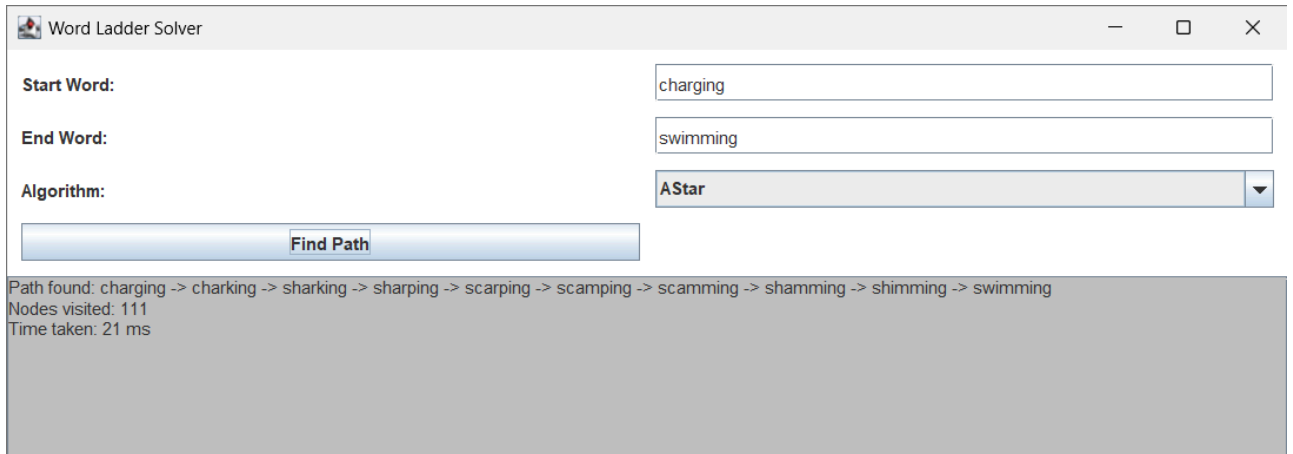
Find Path

Path found: charging -> charming -> chirring -> chirking -> shirking -> sharking -> sharpening -> scarpening -> scampening -> scamming -> shamming -> shimming -> swimming

Nodes visited: 78

Time taken: 25 ms

c) Algoritma A*



Word Ladder Solver

Start Word: charging

End Word: swimming

Algorithm: AStar

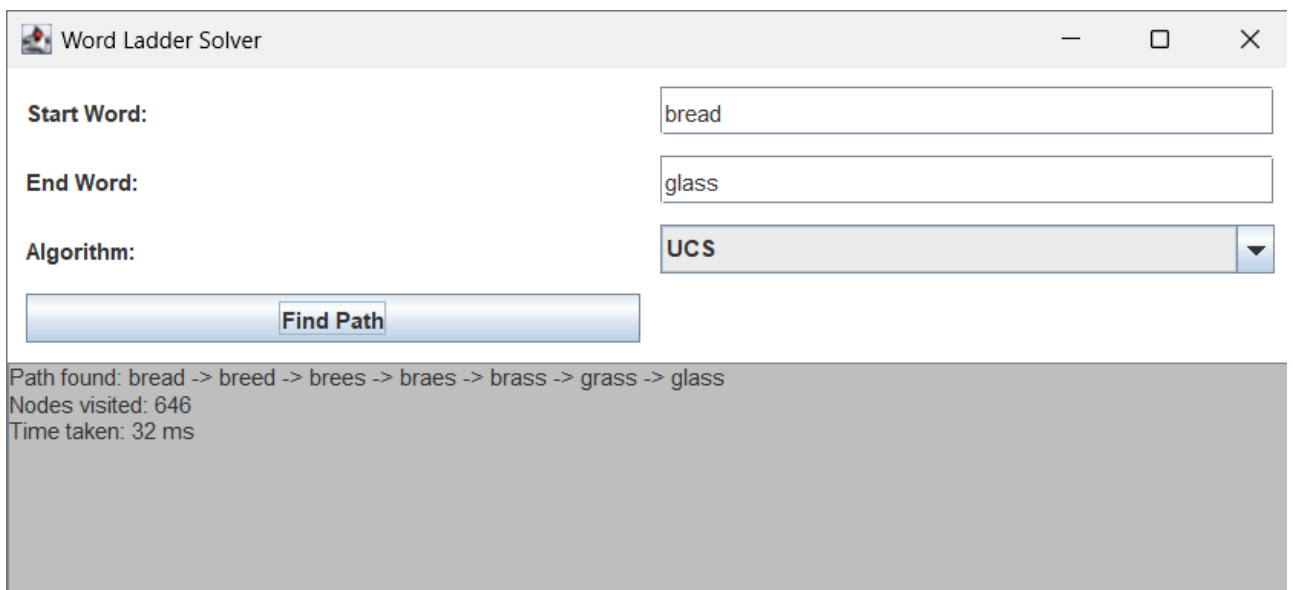
Find Path

Path found: charging -> charking -> sharking -> sharpening -> scarping -> scamping -> scamming -> shamming -> shimming -> swimming
Nodes visited: 111
Time taken: 21 ms

4.6 Set Percobaan Keenam

Testcase : bread, glass

a) Algoritma UCS



Word Ladder Solver

Start Word: bread

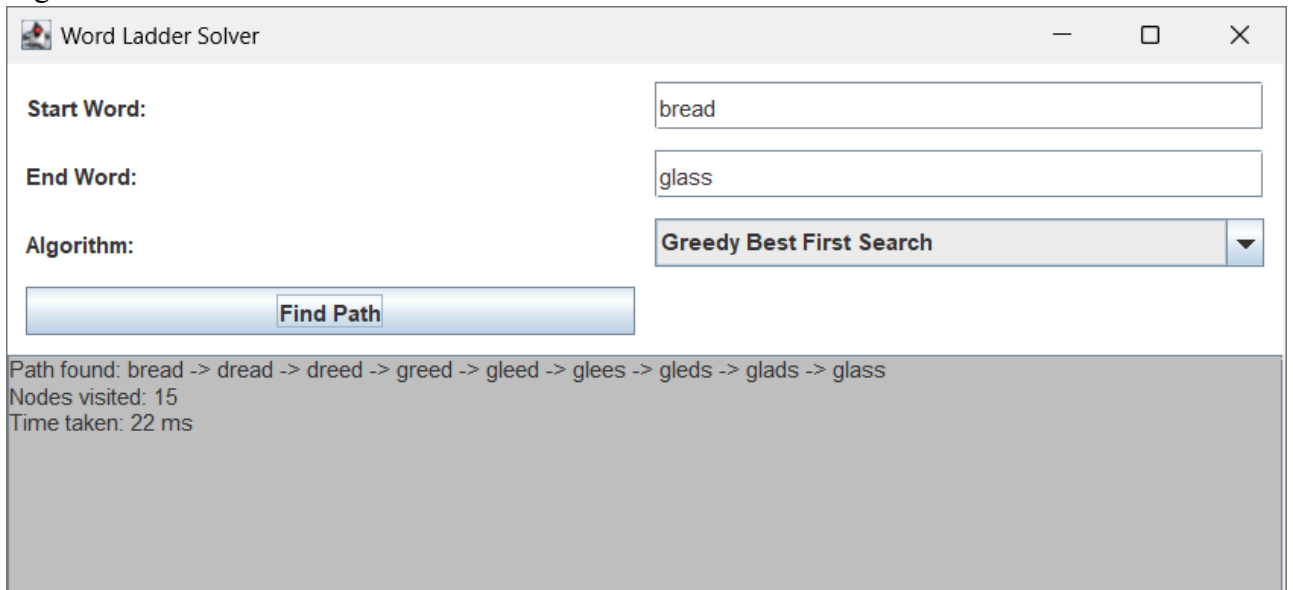
End Word: glass

Algorithm: UCS

Find Path

Path found: bread -> breed -> breez -> braes -> brass -> grass -> glass
Nodes visited: 646
Time taken: 32 ms

b) Algoritma GBFS



Word Ladder Solver

Start Word: bread

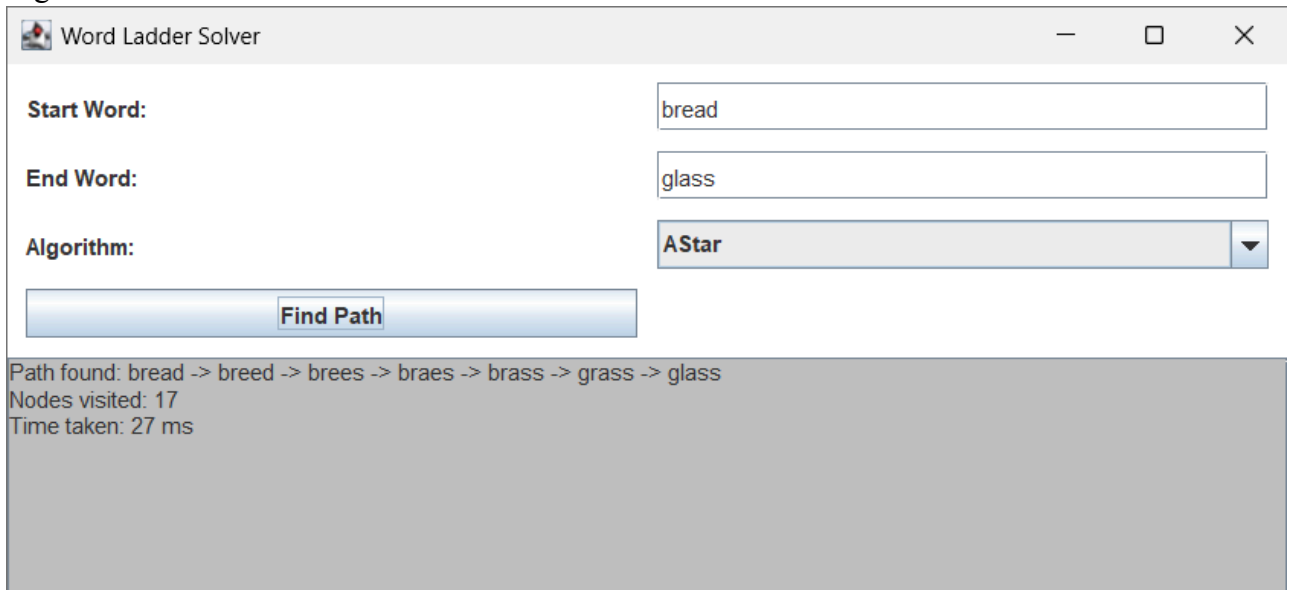
End Word: glass

Algorithm: Greedy Best First Search

Find Path

Path found: bread -> dread -> dreed -> greed -> gleed -> glees -> gleds -> glads -> glass
Nodes visited: 15
Time taken: 22 ms

c) Algoritma A*



Word Ladder Solver

Start Word: bread

End Word: glass

Algorithm: AStar

Find Path

Path found: bread -> breed -> breez -> braes -> brass -> grass -> glass
Nodes visited: 17
Time taken: 27 ms

BAB V

ANALISIS, SARAN, DAN KESIMPULAN

5.1 Analisis Efektivitas Algoritma

1. Optimalitas :

Dalam permasalahan optimalitas suatu algoritma, Algoritma UCS akan menemukan solusi optimal jika usaha atau jarak langkah konstan dan meningkat secara stabil. Algoritma UCS akan melakukan pencarian sampai menemukan solusi terpendek ke tujuan tanpa dipengaruhi oleh heuristik.

Untuk kasus algoritma Greedy Best First Search, algoritma ini akan menggunakan heuristik untuk membantu dalam mencari path atau jalur untuk dilalui. Dengan adanya heuristik, algoritma ini akan menjadi lebih cepat tetapi tidak selalu optimal. Algoritma ini cenderung memilih solusi lokal sehingga mungkin gagal menemukan jalur terpendek.

Untuk kasus A*, A* merupakan algoritma yang paling efisien dari ketiga algoritma yang dibuat dalam tugas ini. Hal tersebut terjadi karena menggabungkan kelebihan dari UCS (optimal) dan Greedy Best First Search (cepat). Algoritma A* menggunakan fungsi evaluasi $f(n) = g(n) + h(n)$ dimana $g(n)$ adalah biaya dari start ke n dan $h(n)$ adalah estimasi jarak atau usaha yang dikeluarkan dari n ke tujuan. Dengan adanya heuristik $h(n)$ yang admissible dan consistent, algoritma A* akan menghasilkan solusi yang optimal.

2. Waktu Eksekusi

Dalam lama waktu untuk mengeksekusi program, UCS tidak menggunakan heuristik, sehingga waktu yang dibutuhkan relatif lebih lama jika dibandingkan kedua algoritma lainnya. Algoritma Greedy Best First Search memiliki waktu yang lebih cepat dibandingkan UCS karena algoritma ini akan selalu melangkah ke arah tujuan paling dekat menurut heuristik. Namun, kelemahan algoritma ini adalah memiliki langkah atau jalur relatif lebih panjang. Untuk A*, waktu yang diperlukan cukup singkat karena fungsi ini memastikan pencarian tetap ada pada jalur optimal dengan memanfaatkan heuristik untuk mengurangi jumlah node yang dikunjungi.

3. Alokasi dan Penggunaan Memori

Dalam penggunaan memori saat program dijalankan, terdapat beberapa hal yang ditemukan. Pertama, UCS mungkin memerlukan lebih banyak memori karena menyimpan semua path yang terbuka dalam queue. Selanjutnya, Greedy Best First Search dan A* biasanya memiliki penggunaan memori yang lebih sedikit dibandingkan UCS, namun tetap saja, hal ini terjadi bergantung pada keberhasilan heuristik dalam membatasi eksplorasi. Namun, A* secara umum mungkin membutuhkan memori lebih banyak dibanding Greedy Best First Search jika heuristik tidak memotong banyak cabang.

4. Penjelasan mengenai GUI

Untuk UI yang digunakan dalam tugas ini menggunakan Java Swing. Dalam page UI tersebut, akan ada dua panel input untuk memasukkan kata awal dan kata akhir. Untuk memilih algoritma yang digunakan, akan ada pilihan dalam bentuk dropdown menu. Akan ada tombol yang bertuliskan "Find Path" untuk memulai pencarian jalur dari kata awal ke kata akhir menggunakan algoritma yang dipilih dari

dropdown menu sebelumnya. Setelah selesai mencari, akan ada beberapa informasi penting yang ditampilkan yaitu durasi pencarian, jumlah node yang dikunjungi, dan node yang dikunjungi. Jika jumlah huruf dari kedua kata yang dimasukkan tidak sama, akan ada keluaran atau output bahwa kata yang dimasukkan tidak valid.

5.2 Saran

1. Optimasi Algoritma:

Mencari dan memaksimalkan waktu proses dalam algoritma sehingga jawaban yang diperoleh menjadi lebih cepat didapatkan.

2. Interaktivitas Pengguna:

Mengembangkan visualisasi pada GUI sehingga pengguna lebih merasa lebih nyaman dalam menjalankan program serta menambahkannya fitur yang bisa ditambahkan seperti menambahkan rekomendasi kata sehingga pengguna bisa langsung memilih.

3. Analisis pengujian :

Melakukan analisis komparatif lebih mendalam terhadap ketiga algoritma diberbagai skenario pengujian untuk menilai keefektifan ketiga algoritma dalam berbagai kondisi dan mengevaluasi performapada platform JavaSwing atau website untuk memastikan jawabanyang diperoleh maksimal dan efisien.

5.3 Kesimpulan

Tugas kecil Strategi Algoritma yang ketiga ini merupakan tugas yang diberikan untuk meningkatkan kemampuan mahasiswa dalam mata kuliah strategi algoritma terutama dalam materi algoritma UCS, Greedy Best First Search, dan juga A*. Secara garis besar, algoritma Uniform Cost Search (UCS), mirip dengan pendekatan brute force, dimana algoritma ini akan mengeksplorasi semua jalur yang mungkin tanpa memprioritaskan jalur berdasarkan estimasi jarak ke tujuan (heuristik). Hal ini menjadikan UCS kurang efisien dalam hal waktu eksekusi, namun algoritma ini memiliki kelebihan yaitu dapat menemukan solusi yang paling akurat atau optimal karena tidak terpengaruh oleh heuristik yang mungkin bias. Di sisi lain, Greedy Best First Search menggunakan pendekatan heuristik yang memprioritaskan node yang tampak paling dekat dengan tujuan, mirip dengan konsep divide and conquer dalam memecah masalah menjadi bagian yang tampaknya paling menjanjikan. Namun, ini sering mengakibatkan Greedy Best First Search menemukan solusi lebih cepat tetapi tidak selalu menjamin solusi tersebut adalah yang terpendek, karena bisa saja terjebak pada optimal lokal. A*, yang merupakan algoritma yang menggabungkan kelebihan UCS dalam mencari solusi optimal dan kecepatan Greedy Best First Search. Dengan adanya gabungan kelebihan dua algoritma lainnya, hasil yang didapatkan memiliki hasil yang efisien dan singkat.

Dari hasil analisis dari hasil yang didapat dari ketiga algoritma program pada tugas ini, A* terbukti menjadi algoritma yang paling efisien dan efektif dalam memecahkan masalah Word Ladder jika dibandingkan dengan UCS dan Greedy Best First Search. Hal ini membuktikan mengenai pentingnya memilih strategi algoritma yang sesuai dengan kebutuhan spesifik dari masalah yang dihadapi, terutama dalam kasus yang membutuhkan keseimbangan antara kecepatan dan akurasi.

5.4 Refleksi

Tugas ini mengajarkan pentingnya pemilihan dan penerapan algoritma yang tepat dalam membuat suatu program. Tugas ini berhasil menunjukkan bagaimana teori algoritma dapat diterapkan dengan sukses dalam praktik, khususnya dalam materi Algoritma UCS, Greedy Best First Search, dan juga A*. Dari tugas ini juga ditekankan pentingnya optimasi dan efisiensi

dalam membuat suatu program, dan pentingnya memiliki GUI yang interaktif dan nyaman untuk digunakan.

Secara umum, tugas ini adalah sebuah ilustrasi yang sangat baik tentang bagaimana konsep teori dapat digabungkan dengan penerapannya dalam praktik membuat suatu program, tugas kecil stima kali ini juga menunjukkan efektivitas pengetahuan algoritma dalam memecahkan masalah secara efektif dan efisien. Tugas ini juga menekankan pentingnya rekayasa perangkat lunak yang berkualitas, dimana perencanaan yang teliti, desain yang matang, dan eksekusi yang hati-hati menjadi faktor utama dalam menciptakan solusi teknologi yang inovatif dan berpengaruh.

Saya juga memiliki keinginan untuk meningkatkan GUI menjadi lebih baik dan dikembangkan menjadi sebuah website sehingga pengguna dapat lebih nyaman dalam menggunakan program yang saya buat.

LAMPIRAN

Link atau pranala yang digunakan untuk membantu keberjalanan tugas ini :

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf>

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf>

Checklist Tugas Kecil 2 IF2211 Strategi Algoritma:

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	✓	