

Tugas Kecil 3 IF2211 Strategi Algoritma
Penyelesaian Permainan Word Ladder Menggunakan
Algoritma UCS, Greedy Best First Search, dan A*
Semester II Tahun 2023/2024



Disusun oleh

Sa'ad Abdul Hakim (13522092)

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG
2024

Daftar Isi

A. Analisis dan Implementasi Dalam Algoritma UCS, Greedy Best First Search, dan A*	3
B. Source Code Program	6
C. Test Case	17
D. Analisis Perbandingan Solusi UCS, Greedy Best First Search, dan A*	28
E. Penjelasan Implementasi Bonus	29
Lampiran	30

A. Analisis dan Implementasi Dalam Algoritma UCS, Greedy Best First Search, dan A*

1. Uniform Cost Search (UCS)

Uniform Cost Search (UCS) adalah metode penelusuran graf yang mempertimbangkan biaya atau cost yang terkait dengan simpul awal. Algoritma ini memulai pencarian dari root node, kemudian dilanjutkan ke node-node selanjutnya. Simpul yang diekspansi merupakan simpul dengan harga (cost) kumulatif terendah. Ongkos kumulatif simpul n didefinisikan sebagai $g(n)$, yaitu jarak dari akar ke simpul n . UCS menjamin optimalitas dalam pencarian solusi, meskipun kompleksitas waktu algoritmanya dapat menjadi masalah, terutama dalam graf dengan struktur yang kompleks. Meskipun demikian, UCS tetap menjadi pilihan yang disukai untuk aplikasi dimana keoptimalan jalur solusi adalah prioritas utama.

Langkah-langkah dalam mengimplementasi algoritma Uniform Cost Search dalam permainan World Ladder adalah sebagai berikut :

1. Pertama, kata awal atau *start word* akan dimasukkan ke dalam sebuah priority queue kosong dalam bentuk node dengan cost 0.
2. Kemudian priority queue tersebut akan di cek dan dikeluarkan elemen terdepannya.
3. Jika elemen terdepan tersebut adalah kata tujuan atau *end word* maka proses pencarian akan dihentikan.
4. Jika elemen terdepan tersebut bukan merupakan kata tujuan, maka kata tersebut akan disimpan dalam sebuah set visited dan kemudian akan dicari seluruh tetangga dari elemen tersebut yaitu yang memiliki perbedaan 1 kata dari elemen tersebut serta belum pernah dikunjungi atau tidak ada dalam set visited.
5. Seluruh tetangga elemen tersebut kemudian akan dibuat menjadi sebuah node dengan cost ditambah satu dari cost elemen terdepan yang tadi dikeluarkan.
6. Node-node tersebut kemudian akan ditambahkan ke dalam priority queue yang tadi sudah dibuat.
7. Langkah 2-6 akan terus dilakukan hingga mencapai kondisi nomor 3 atau priority queue kosong yang berarti tidak ada ladder yang merupakan solusi dari pencarian.

2. Greedy Best First Search (GBFS)

Greedy Best-First Search (GBFS) adalah algoritma pencarian graf yang menggunakan pendekatan heuristik untuk mengeksplorasi simpul yang memiliki nilai heuristik paling rendah terlebih dahulu. Berbeda dengan UCS, GBFS tidak mempertimbangkan biaya kumulatif dari simpul akar ke simpul saat ini, melainkan hanya memperhatikan nilai heuristik dari simpul tersebut. Pada program ini, nilai heuristik tersebut adalah cost dari kata tersebut. Simpul yang dipilih untuk dieksplorasi selanjutnya adalah simpul dengan biaya atau cost terendah. Biaya simpul n didefinisikan sebagai $h(n)$, yaitu jumlah huruf yang berbeda dari huruf pada kata tujuan pada posisi yang sama. Meskipun GBFS dapat

memberikan solusi dengan cepat dalam beberapa situasi, algoritma ini tidak menjamin pencarian solusi terpendek atau optimal, dan bisa saja terperangkap dalam optimum lokal. Meskipun demikian, GBFS sering digunakan di aplikasi di mana kecepatan eksekusi menjadi lebih penting daripada keoptimalan solusi.

Langkah-langkah dalam mengimplementasi algoritma Greedy Best First Search dalam permainan World Ladder adalah sebagai berikut :

1. Pertama, kata awal atau *start word* akan dimasukkan ke dalam sebuah priority queue kosong dalam bentuk node dengan cost adalah jumlah huruf yang berbeda dengan huruf pada kata tujuan pada posisi yang sama.
2. Kemudian priority queue tersebut akan di cek dan dikeluarkan elemen terdepannya.
3. Jika elemen terdepan tersebut adalah kata tujuan atau *end word* maka proses pencarian akan dihentikan.
4. Jika elemen terdepan tersebut bukan merupakan kata tujuan, maka kata tersebut akan disimpan dalam sebuah set visited dan kemudian akan dicari seluruh tetangga dari elemen tersebut yaitu yang memiliki perbedaan 1 kata dari elemen tersebut serta belum pernah dikunjungi atau tidak ada dalam set visited.
5. Seluruh tetangga elemen tersebut kemudian akan dibuat menjadi sebuah node dengan cost adalah jumlah huruf yang berbeda dengan huruf pada kata tujuan pada posisi yang sama.
6. Node-node tersebut kemudian akan ditambahkan ke dalam priority queue yang tadi sudah dibuat.
7. Langkah 2-6 akan terus dilakukan hingga mencapai kondisi nomor 3 atau priority queue kosong yang berarti tidak ada ladder yang merupakan solusi dari pencarian.

3. A* Search

A* (A-star) adalah algoritma pencarian yang menggunakan pendekatan gabungan antara heuristik dan biaya aktual untuk menemukan jalur terpendek dalam sebuah graf. Pada program ini, nilai heuristik tersebut adalah cost dari kata tersebut. Algoritma ini dimulai dari simpul awal dan secara iteratif mengeksplorasi simpul-simpul yang memiliki perkiraan biaya terendah (biaya atau cost yang terkait dengan simpul awal ditambah dengan nilai heuristik) hingga mencapai simpul tujuan. Biaya atau cost dari sebuah simpul n didefinisikan sebagai $f(n) = g(n) + h(n)$, di mana $g(n)$ adalah jarak dari akar ke simpul n . A* menjamin pencarian solusi yang optimal serta memastikan bahwa jalur yang ditemukan adalah yang terpendek. Algoritma ini sering menjadi pilihan yang optimal dalam aplikasi di mana kecepatan dan keoptimalan solusi sama-sama penting.

Langkah-langkah dalam mengimplementasi algoritma A*(A-star) dalam permainan World Ladder adalah sebagai berikut :

1. Pertama, kata awal atau *start word* akan dimasukkan ke dalam sebuah priority queue kosong dalam bentuk node dengan cost adalah jumlah huruf yang berbeda dengan huruf pada kata tujuan pada posisi yang sama.

2. Kemudian priority queue tersebut akan di cek dan dikeluarkan elemen terdepannya.
3. Jika elemen terdepan tersebut adalah kata tujuan atau *end word* maka proses pencarian akan dihentikan.
4. Jika elemen terdepan tersebut bukan merupakan kata tujuan, maka kata tersebut akan disimpan dalam sebuah set visited dan kemudian akan dicari seluruh tetangga dari elemen tersebut yaitu yang memiliki perbedaan 1 kata dari elemen tersebut serta belum pernah dikunjungi atau tidak ada dalam set visited.
5. Seluruh tetangga elemen tersebut kemudian akan dibuat menjadi sebuah node dengan cost adalah jarak dari akar ke simpul n ditambah jumlah huruf yang berbeda dengan huruf pada kata tujuan pada posisi yang sama.
6. Node-node tersebut kemudian akan ditambahkan ke dalam priority queue yang tadi sudah dibuat.
7. Langkah 2-6 akan terus dilakukan hingga mencapai kondisi nomor 3 atau priority queue kosong yang berarti tidak ada ladder yang merupakan solusi dari pencarian.

Dalam algoritma UCS, GBFS, dan A* terdapat sebuah fungsi yang bertanda $f(n)$, fungsi tersebut adalah fungsi evaluasi yang berisi nilai estimasi total biaya dari sebuah path yang melalui sebuah simpul node dan mengarah simpul tujuan. Nilai fungsi tersebut digunakan untuk menghitung biaya dari setiap node yang ada. Semakin kecil nilai fungsi tersebut maka prioritas node tersebut akan menjadi lebih tinggi untuk dicek karena node tersebut akan berada dalam posisi yang terdepan jika memiliki nilai evaluasi yang lebih kecil. Pada algoritma Uniform Cost Search (UCS) nilai evaluasi fungsinya sama dengan nilai sebuah fungsi bertanda $g(n)$, fungsi tersebut adalah fungsi yang berisi nilai biaya dari simpul akar hingga simpul saat ini atau simpul n yang dalam permasalahan ini berisi jarak dari simpul akar ke simpul ke n .

Heuristik yang digunakan dalam algoritma A* memenuhi sifat admissible. Heuristik yang digunakan memiliki sifat admissible jika estimasi biaya dari simpul saat ini ke simpul tujuan tidak boleh melebihi-lebihkan dari biaya sebenarnya dari simpul tersebut ke simpul tujuan. Dengan kata lain, heuristik harus selalu menghasilkan nilai yang kurang dari atau sama dengan biaya sebenarnya dari simpul saat ini ke simpul tujuan. Hal tersebut untuk memastikan bahwa algoritma A* dapat menemukan jalur optimal. Jika heuristik tidak admissible, A* tidak lagi menjamin optimalitas solusi. Pada program ini, heuristik yang digunakan dalam algoritma A* memenuhi sifat admissible.

Pada dasarnya, algoritma Uniform Cost Search (UCS) memiliki kesamaan dengan algoritma Breadth-First Search (BFS) dalam urutan pembangkitan node. Hal ini karena pada UCS, biaya $f(n)$ dari sebuah simpul n dihitung sebagai kedalaman $g(n)$ dari simpul tersebut. Dalam konteks Word Ladder, ini berarti bahwa saat melakukan ekspansi simpul, simpul-simpul baru akan memiliki biaya yang sama dengan kedalaman atau depth mereka yang ditambah satu. Dengan kata lain, simpul-simpul dengan kedalaman yang sama akan memiliki biaya yang sama. Sehingga, urutan pemilihan simpul dalam antrian prioritas pada UCS akan serupa dengan urutan pemilihan

pada BFS, di mana simpul-simpul pada kedalaman yang sama diekspansi terlebih dahulu sebelum melanjutkan ke kedalaman berikutnya. Oleh karena itu, pada kasus Word Ladder, algoritma UCS dan BFS mungkin akan menghasilkan urutan pembangkitan simpul yang sama, karena kedua algoritma ini mempertimbangkan kedalaman simpul dalam mengeksplorasi graf.

Secara teoritis, algoritma A* lebih efisien dibandingkan dengan UCS karena A* menggunakan heuristic untuk mengukur jarak dari simpul saat ini ke simpul tujuan, sedangkan UCS hanya menggunakan kedalaman simpul dalam pemilihan langkah selanjutnya. Dengan memanfaatkan heuristic, A* dapat memilih simpul-simpul yang lebih mungkin mengarah ke tujuan, sehingga mengurangi jumlah simpul yang perlu dieksplorasi dan meminimalkan waktu yang dibutuhkan untuk menemukan jalur solusi. Dengan demikian, A* cenderung dapat menghindari penjelajahan jalur yang tidak diperlukan, membuatnya lebih efisien dalam menemukan solusi pada kasus Word Ladder.

Secara teoritis, algoritma Greedy Best First Search (GBFS) tidak menjamin solusi optimal karena menggunakan pendekatan algoritma greedy. Algoritma GBFS dapat terjebak di local minima dan gagal menemukan solusi, karena hanya mempertimbangkan nilai heuristic dari setiap simpul dalam pemilihan langkah selanjutnya. Oleh karena itu, meskipun GBFS dapat menemukan solusi dengan cepat dalam beberapa kasus, tidak ada jaminan bahwa solusi yang ditemukan akan optimal.

B. Source Code Program

node.java

Kelas node merupakan representasi simpul dalam struktur data yang digunakan untuk menyelesaikan masalah Word Ladder atau permasalahan lain yang melibatkan pencarian jalur di graf. Setiap objek node menyimpan informasi tentang kata yang direpresentasikan oleh simpul tersebut (word), simpul induknya (parent), dan biaya simpul saat ini (cost). Terdapat tiga konstruktor yang memungkinkan inisialisasi objek node dengan berbagai kombinasi argumen. Metode distancefromroot() digunakan untuk menghitung jarak atau kedalaman simpul dari simpul awal ke simpul saat ini dengan menghitung jumlah simpul induknya secara rekursif.

```
public class node {
    public String word;
    public node parent;
    public int cost;

    public node(String word) {
        this.word = word;
        this.parent = null;
        this.cost = 0;
    }
}
```

```

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

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

public int distancefromroot() {
    int count = 0;
    node current = this;
    while (current.parent != null) {
        count++;
        current = current.parent;
    }
    return count;
}
}

```

util.java

Kelas util berisi kumpulan metode utilitas yang digunakan dalam pemrosesan kata dan pencarian jalur pada struktur data graf. Metode reconstructpath() digunakan untuk merekonstruksi jalur dari simpul akhir ke simpul awal dalam bentuk daftar kata. Metode getneighbors() mengembalikan himpunan tetangga kata dari kata yang diberikan, berdasarkan perubahan satu huruf. Metode isWordInDictionary() digunakan untuk memeriksa apakah kata tertentu ada dalam kamus kata yang diberikan. Terakhir, metode getWordsWithLength() mengembalikan himpunan kata dengan panjang tertentu dari kamus kata.

```

public class util {
    public static List<String> reconstructpath(node node) {
        List<String> path = new ArrayList<>();
        while (node != null) {
            path.add(node.word);
            node = node.parent;
        }
    }
}

```

```

        Collections.reverse(path);
        return path;
    }

    public static Set<String> getneighbors(String word, Set<String>
dictionary) {
        Set<String> neighbors = new HashSet<>();
        for (int i = 0; i < word.length(); i++) {
            char[] chars = word.toCharArray();
            for (char c = 'a'; c <= 'z'; c++) {
                chars[i] = c;
                String newWord = new String(chars);
                if (!newWord.equals(word) && dictionary.contains(newWord))
{
                    neighbors.add(newWord);
                }
            }
        }

        return neighbors;
    }

    public static boolean isWordInDictionary(String word) {
        try {
            BufferedReader reader = new BufferedReader(new
FileReader("../test/dictionary.txt"));
            String line = reader.readLine();

            while (line != null) {
                if (line.equals(word)) {
                    reader.close();
                    return true;
                }
                line = reader.readLine();
            }
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

```



```

        return false;
    }

    public static Set<String> getWordsWithLength(int length) {
        Set<String> words = new HashSet<>();
        try {
            BufferedReader reader = new BufferedReader(new
FileReader("../test/dictionary.txt"));
            String line = reader.readLine();

            while (line != null) {
                if (line.length() == length) {
                    words.add(line);
                }
                line = reader.readLine();
            }
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return words;
    }
}

```

greedy.java

Kelas greedy bertugas menemukan jalur terpendek antara dua kata dalam masalah Word Ladder menggunakan pendekatan Greedy Best-First Search (GBFS). Metode hammingDistance() menghitung jumlah huruf yang berbeda antara huruf pada kata yang diterima dari parameter dengan huruf pada kata tujuan pada posisi yang sama antara dua string. Metode findShortestLadder() menerima kata awal, kata akhir, kamus kata, dan objek IntegerWrapper untuk menghitung jumlah simpul yang dikunjungi. Utilitas seperti util.reconstructpath() dan util.getneighbors() digunakan dalam proses pencarian. Jika jalur tidak ditemukan, metode mengembalikan daftar dengan satu elemen berisi pesan "No ladder found".

```

public class greedy {
    public static int hammingDistance(String str1, String str2) {
        if (str1.length() != str2.length()) {
            throw new IllegalArgumentException("Strings must have equal
length");
        }
    }
}

```

```

    }
    int distance = 0;
    for (int i = 0; i < str1.length(); i++) {
        if (str1.charAt(i) != str2.charAt(i)) {
            distance++;
        }
    }
    return distance;
}

    public static List<String> findShortestLadder(String startWord, String
endWord, Set<String> dictionary, IntegerWrapper countnode) {
        PriorityQueue<node> pq = new
PriorityQueue<>(Comparator.comparingInt(o -> o.cost));
        Set<String> visited = new HashSet<>();
        node startNode = new node(startWord, hammingDistance(startWord,
endWord));

        pq.offer(startNode);

        while (!pq.isEmpty()) {
            node current = pq.poll();
            String currentWord = current.word;

            if (currentWord.equals(endWord)) {
                countnode.value = visited.size()+1;
                return util.reconstructpath(current);
            }

            visited.add(currentWord);

            for (String neighbor : util.getneighbors(currentWord,
dictionary)) {
                if (!visited.contains(neighbor)) {
                    node neighborNode = new node(neighbor, current,
hammingDistance(neighbor, endWord));
                    pq.offer(neighborNode);
                }
            }
        }
    }

```

```

    }

    countnode.value = visited.size()+1;
    return Collections.singletonList("No ladder found");
}
}

```

ucs.java

Kelas ucs bertanggung jawab untuk menemukan jalur terpendek antara dua kata dalam masalah Word Ladder menggunakan pendekatan Uniform Cost Search (UCS). Metode findShortestLadder() menerima kata awal, kata akhir, kamus kata, dan objek IntegerWrapper untuk menghitung jumlah simpul yang dikunjungi. Algoritma UCS digunakan untuk mencari jalur terpendek dengan mempertimbangkan biaya kumulatif dari setiap simpul ke kata akhir. Metode ini menggunakan PriorityQueue untuk memprioritaskan simpul dengan biaya terendah, dan menggunakan utilitas seperti util.reconstructpath() dan util.getneighbors() dalam proses pencarian. Jika jalur tidak ditemukan, metode mengembalikan daftar dengan satu elemen berisi pesan "No ladder found".

```

public class ucs {
    public static List<String> findShortestLadder(String startWord, String
endWord, Set<String> dictionary, IntegerWrapper countnode) {
        PriorityQueue<node> pq = new
PriorityQueue<>(Comparator.comparingInt(o -> o.cost));
        Set<String> visited = new HashSet<>();
        node startNode = new node(startWord);

        pq.offer(startNode);

        while (!pq.isEmpty()) {
            node current = pq.poll();
            String currentWord = current.word;

            if (currentWord.equals(endWord)) {
                countnode.value = visited.size()+1;
                return util.reconstructpath(current);
            }

            visited.add(currentWord);

```

```

        for (String neighbor : util.getneighbors(currentWord,
dictionary)) {
            if (!visited.contains(neighbor)) {
                node neighborNode = new node(neighbor, current,
current.cost + 1);
                pq.offer(neighborNode);
            }
        }
    }

    countnode.value = visited.size()+1;
    return Collections.singletonList("No ladder found");
}
}

```

Astar.java

Kelas Astar digunakan untuk menemukan jalur terpendek antara dua kata dalam masalah Word Ladder menggunakan algoritma A* Search. Metode findShortestLadder() menerima kata awal, kata akhir, kamus kata, dan objek IntegerWrapper untuk menghitung jumlah simpul yang dikunjungi. Algoritma A* digunakan untuk mencari jalur terpendek dengan mempertimbangkan biaya kumulatif dan heuristik dari setiap simpul ke kata akhir. Metode ini menggunakan PriorityQueue untuk memprioritaskan simpul dengan biaya terendah ditambah heuristik terendah, dan menggunakan utilitas seperti util.reconstructpath() dan util.getneighbors() dalam proses pencarian. Jika jalur tidak ditemukan, metode mengembalikan daftar dengan satu elemen berisi pesan "No ladder found".

```

public class Astar {
    public static List<String> findShortestLadder(String startWord, String
endWord, Set<String> dictionary, IntegerWrapper countnode) {
        PriorityQueue<node> pq = new
PriorityQueue<>(Comparator.comparingInt(o -> o.cost));
        Set<String> visited = new HashSet<>();
        node startNode = new node(startWord,
greedy.hammingDistance(startWord, endWord));

        pq.offer(startNode);

        while (!pq.isEmpty()) {
            node current = pq.poll();

```

```

        String currentWord = current.word;

        if (currentWord.equals(endWord)) {
            countnode.value = visited.size()+1;
            return util.reconstructpath(current);
        }

        visited.add(currentWord);

        for (String neighbor : util.getneighbors(currentWord,
dictionary)) {
            if (!visited.contains(neighbor)) {
                node neighborNode = new node(neighbor, current,
greedy.hammingDistance(neighbor, endWord));
                neighborNode.cost += neighborNode.distancefromroot();
                pq.offer(neighborNode);
            }
        }
        countnode.value = visited.size()+1;
        return Collections.singletonList("No ladder found");
    }
}

```

Main.java

Kelas Main adalah penghubung antara pengguna dan algoritma pencarian untuk menyelesaikan masalah Word Ladder. Metode main() memulai aplikasi, sedangkan metode actionPerformed() memproses input pengguna dan menjalankan pencarian. Seluruhnya, kelas Main menyediakan antarmuka yang intuitif dan efisien untuk menemukan solusi dalam masalah Word Ladder.

```

public class Main {
    private JFrame frame;
    private JTextField startField;
    private JTextField endField;
    private JComboBox<String> algorithmComboBox;
    private JButton findButton;
    private JTextArea resultArea;
    private JLabel nodeCountLabel;
    private JLabel timeLabel;
}

```

```

private JLabel memoryLabel;
private JLabel pathlengthLabel;

static class IntegerWrapper {
    public int value;

    public IntegerWrapper(int value) {
        this.value = value;
    }
}

public Main() {
    frame = new JFrame("Word Ladder Solver");
    frame.setSize(400, 400);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setLayout(null);

    JLabel startLabel = new JLabel("Start Word:");
    startLabel.setBounds(20, 20, 100, 20);
    frame.add(startLabel);

    startField = new JTextField();
    startField.setBounds(120, 20, 200, 20);
    frame.add(startField);

    JLabel endLabel = new JLabel("End Word:");
    endLabel.setBounds(20, 50, 100, 20);
    frame.add(endLabel);

    endField = new JTextField();
    endField.setBounds(120, 50, 200, 20);
    frame.add(endField);

    JLabel algorithmLabel = new JLabel("Algorithm:");
    algorithmLabel.setBounds(20, 80, 100, 20);
    frame.add(algorithmLabel);

    algorithmComboBox = new JComboBox<>(new String[]{"UCS", "GBFS",
"A*"});

```

```

algorithmComboBox.setBounds(120, 80, 200, 20);
frame.add(algorithmComboBox);

findButton = new JButton("Find Shortest Ladder");
findButton.setBounds(120, 110, 200, 30);
frame.add(findButton);

resultArea = new JTextArea();
JScrollPane scrollPane = new JScrollPane(resultArea);
scrollPane.setBounds(20, 150, 360, 150);
frame.add(scrollPane);

nodeCountLabel = new JLabel("");
nodeCountLabel.setBounds(20, 310, 200, 20);
frame.add(nodeCountLabel);

timeLabel = new JLabel("");
timeLabel.setBounds(220, 310, 200, 20);
frame.add(timeLabel);

memoryLabel = new JLabel("");
memoryLabel.setBounds(20, 330, 200, 20);
frame.add(memoryLabel);

pathlengthLabel = new JLabel("");
pathlengthLabel.setBounds(220, 330, 200, 20);
frame.add(pathlengthLabel);

findButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String startWord = startField.getText().toLowerCase();
        if (!util.isWordInDictionary(startWord)) {
            resultArea.setText("Kata awal tidak ditemukan dalam kamus.\n" + "Silahkan masukkan kata yang valid.");
            return;
        }

        String endWord = endField.getText().toLowerCase();

```

```

        if (!util.isWordInDictionary(endWord) ||
startWord.length() != endWord.length()){
            if (!util.isWordInDictionary(endWord)) {
                resultArea.setText("Kata tujuan tidak ditemukan
dalam kamus.\n" + "Silahkan masukkan kata yang valid.");
            } else {
                resultArea.setText("Panjang kata tujuan tidak sama
dengan panjang kata awal.\n" + "Silahkan masukkan kata yang valid.");
            }
            return;
        }

        String algorithm = (String)
algorithmComboBox.getSelectedItem();

        IntegerWrapper countnode = new IntegerWrapper(0);
        List<String> ladder = new ArrayList<>();
        long startTime = System.nanoTime();
        long endTime = 0;

        Set<String> words =
util.getWordsWithLength(startWord.length());

        switch (algorithm) {
            case "UCS":
                ladder = ucs.findShortestLadder(startWord,
endWord, words, countnode);
                break;
            case "GBFS":
                ladder = greedy.findShortestLadder(startWord,
endWord, words, countnode);
                break;
            case "A*":
                ladder = Astar.findShortestLadder(startWord,
endWord, words, countnode);
                break;
        }
        endTime = System.nanoTime();
        long duration = endTime - startTime;
        double seconds = duration / 1_000_000_000.0;

```



```

        resultArea.setText("Shortest Ladder:\n" +
String.join("\n", ladder));
        nodeCountLabel.setText("Node Count: " + countnode.value);
        timeLabel.setText("Execution Time: " + seconds + " s");
        pathlengthLabel.setText("Path Length: " + (ladder.size() -
1));

        MemoryMXBean memoryBean =
ManagementFactory.getMemoryMXBean();
        MemoryUsage heapMemoryUsage =
memoryBean.getHeapMemoryUsage();
        long usedMemory = heapMemoryUsage.getUsed();
        memoryLabel.setText("Used Memory: " + usedMemory + "
bytes");
    }
});

    frame.setVisible(true);
}

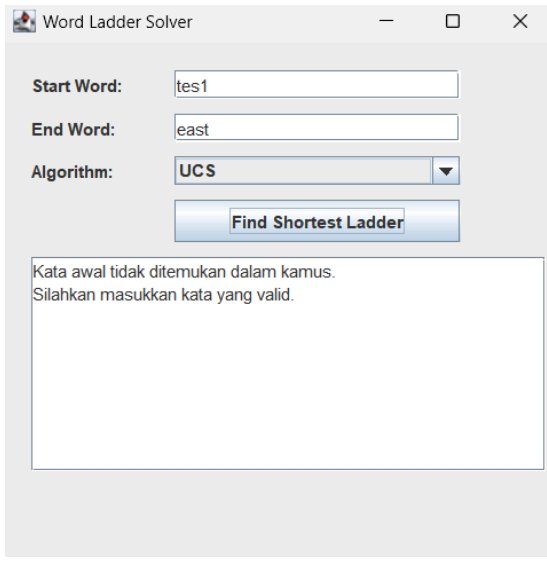
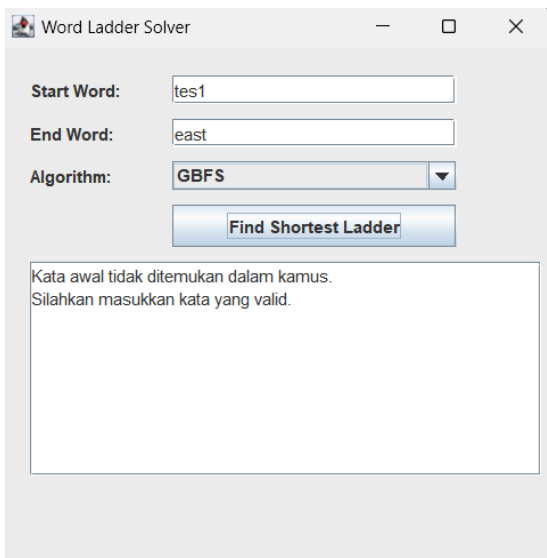
public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new Main();
        }
    });
}
}

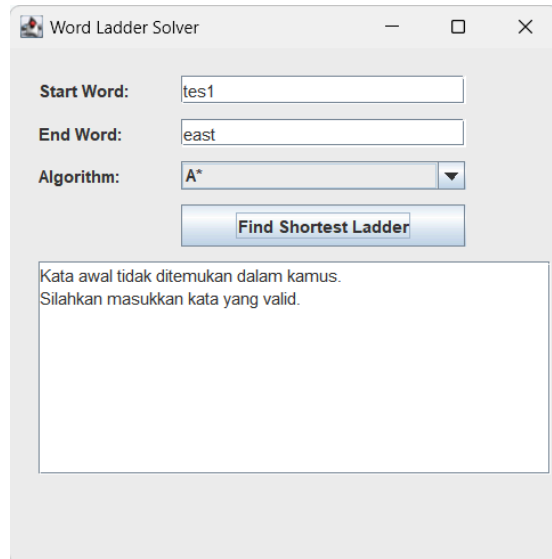
```

C. Test Case

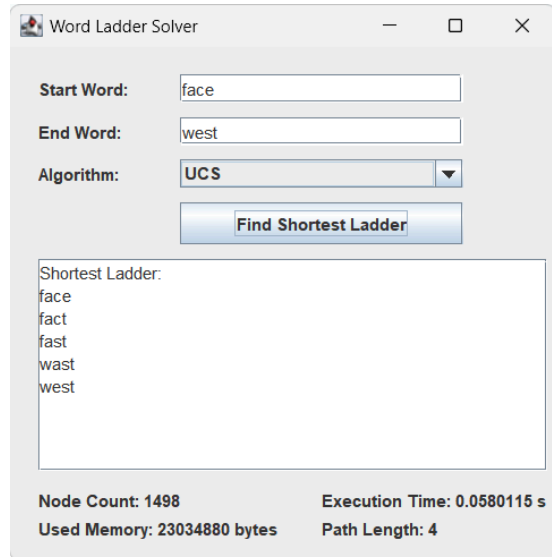
Test Case 1

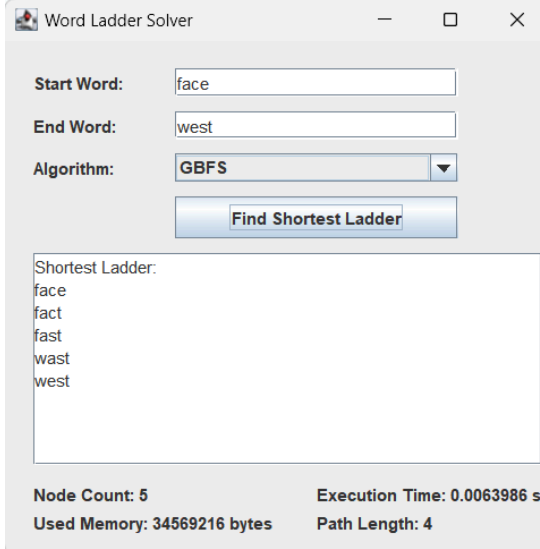
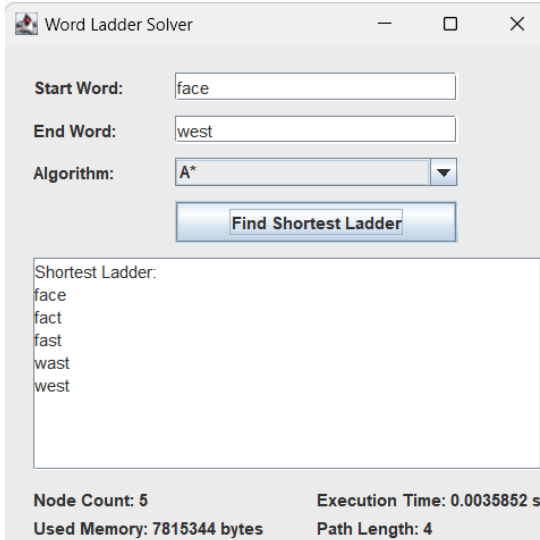
Masukkan	Algoritma	Hasil
----------	-----------	-------

<p>Start Word : tes1 End Word : east</p>	<p>Uniform Cost Search (UCS)</p>	
	<p>Greedy Best First Search (GBFS)</p>	

	A*	
--	----	--

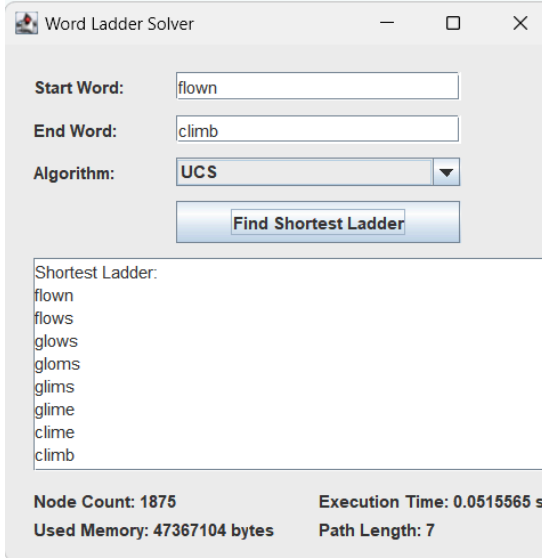
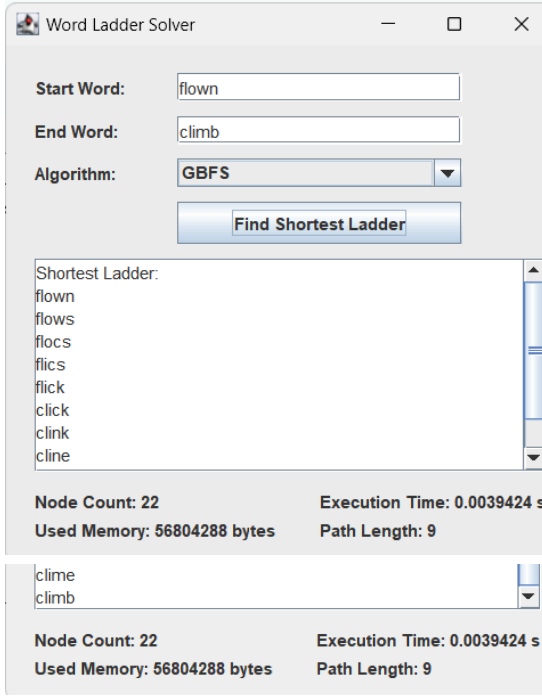
Test Case 2

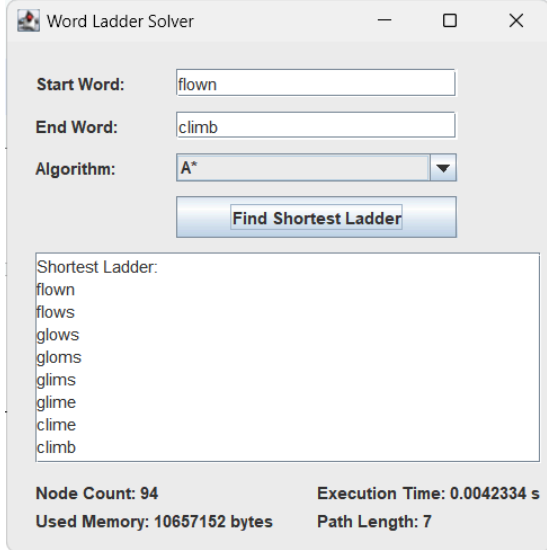
Masukkan	Algoritma	Hasil
Start Word : face End Word : west	Uniform Cost Search (UCS)	

	Greedy Best First Search (GBFS)	 <p>Word Ladder Solver</p> <p>Start Word: <input type="text" value="face"/></p> <p>End Word: <input type="text" value="west"/></p> <p>Algorithm: GBFS</p> <p><input type="button" value="Find Shortest Ladder"/></p> <p>Shortest Ladder:</p> <pre>face fact fast wast west</pre> <p>Node Count: 5 Execution Time: 0.0063986 s</p> <p>Used Memory: 34569216 bytes Path Length: 4</p>
	A*	 <p>Word Ladder Solver</p> <p>Start Word: <input type="text" value="face"/></p> <p>End Word: <input type="text" value="west"/></p> <p>Algorithm: A*</p> <p><input type="button" value="Find Shortest Ladder"/></p> <p>Shortest Ladder:</p> <pre>face fact fast wast west</pre> <p>Node Count: 5 Execution Time: 0.0035852 s</p> <p>Used Memory: 7815344 bytes Path Length: 4</p>

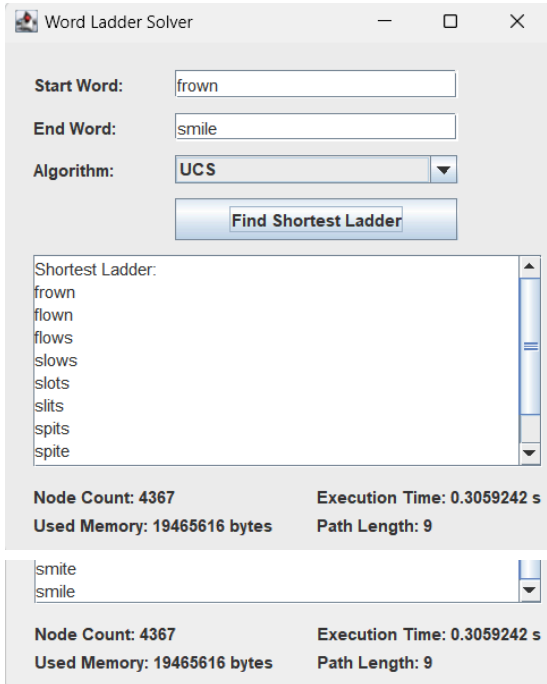
Test Case 3

Masukkan	Algoritma	Hasil
----------	-----------	-------

<p>Start Word : flown End Word : climb</p>	<p>Uniform Cost Search (UCS)</p>	 <p>The screenshot shows the 'Word Ladder Solver' window with the 'Start Word' set to 'flown' and 'End Word' set to 'climb'. The 'Algorithm' is set to 'UCS'. The 'Find Shortest Ladder' button is visible. The 'Shortest Ladder' list contains the following words: flown, flows, glows, gloms, glims, glime, clime, and climb. The statistics at the bottom are: Node Count: 1875, Execution Time: 0.0515565 s, Used Memory: 47367104 bytes, and Path Length: 7.</p>
	<p>Greedy Best First Search (GBFS)</p>	 <p>The screenshot shows the 'Word Ladder Solver' window with the 'Start Word' set to 'flown' and 'End Word' set to 'climb'. The 'Algorithm' is set to 'GBFS'. The 'Find Shortest Ladder' button is visible. The 'Shortest Ladder' list contains the following words: flown, flows, flocs, flics, flick, click, clink, and cline. The statistics at the bottom are: Node Count: 22, Execution Time: 0.0039424 s, Used Memory: 56804288 bytes, and Path Length: 9.</p>

	A*	
--	----	--

Test Case 4

Masukkan	Algoritma	Hasil
Start Word : frown End Word : smile	Uniform Cost Search (UCS)	

Greedy Best First Search (GBFS)

Word Ladder Solver

Start Word: frown

End Word: smile

Algorithm: GBFS

Find Shortest Ladder

Shortest Ladder:

- frown
- flown
- blown
- brown
- grown
- growl
- prowl
- prows

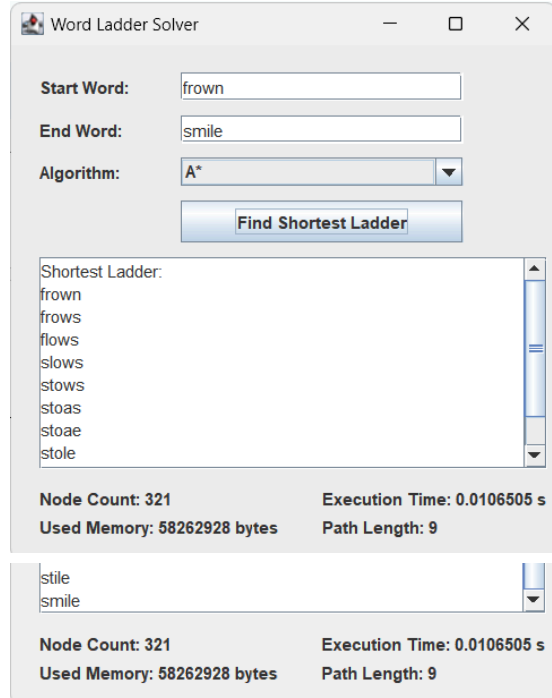
Node Count: 56 Execution Time: 0.0041889 s
Used Memory: 32048528 bytes Path Length: 22

- vrows
- frows
- frogs
- frigs
- trigs
- trips
- tripe
- tribe
- trine

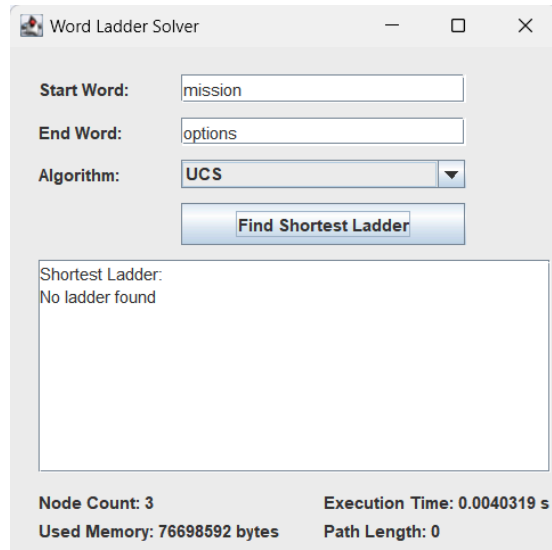
Node Count: 56 Execution Time: 0.0041889 s
Used Memory: 32048528 bytes Path Length: 22

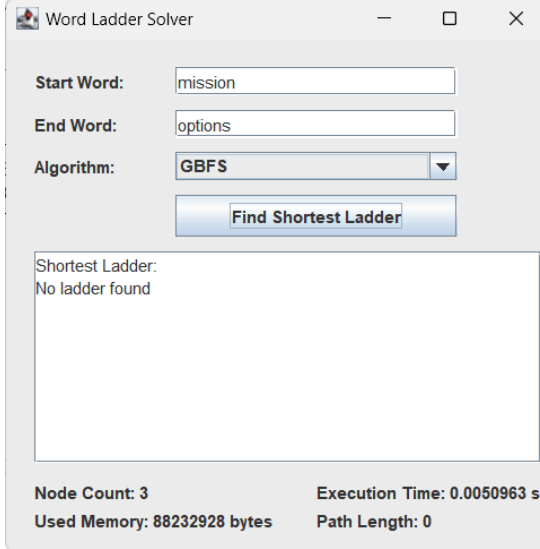
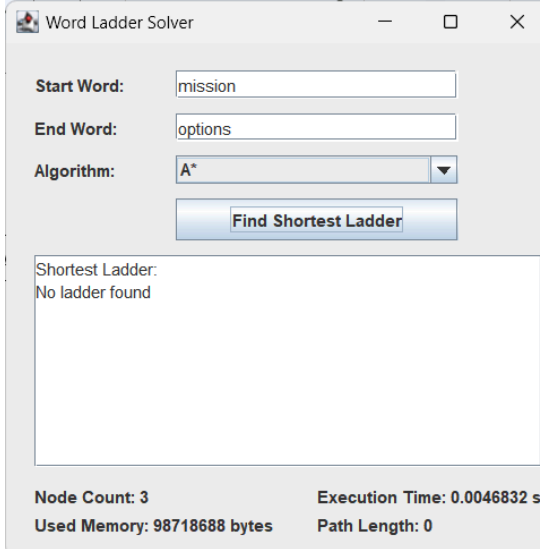
- thine
- shine
- seine
- spine
- spile
- smile

Node Count: 56 Execution Time: 0.0041889 s
Used Memory: 32048528 bytes Path Length: 22

	A*	 <p>The screenshot shows the 'Word Ladder Solver' window. The 'Start Word' is 'frown', 'End Word' is 'smile', and the 'Algorithm' is set to 'A*'. A 'Find Shortest Ladder' button is present. Below it, a list of words in the shortest ladder is shown: frown, frows, flows, slows, stows, stoas, stoe, stole. At the bottom, statistics are displayed: Node Count: 321, Execution Time: 0.0106505 s, Used Memory: 58262928 bytes, Path Length: 9.</p>
--	----	---

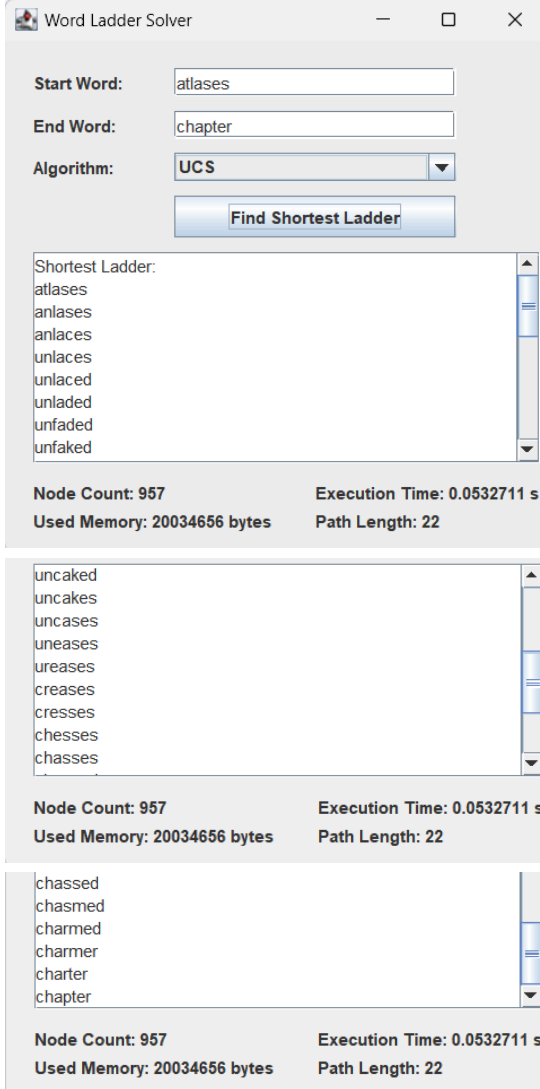
Test Case 5

Masukkan	Algoritma	Hasil
Start Word : mission End Word : options	Uniform Cost Search (UCS)	 <p>The screenshot shows the 'Word Ladder Solver' window. The 'Start Word' is 'mission', 'End Word' is 'options', and the 'Algorithm' is set to 'UCS'. A 'Find Shortest Ladder' button is present. Below it, the text 'No ladder found' is displayed. At the bottom, statistics are displayed: Node Count: 3, Execution Time: 0.0040319 s, Used Memory: 76698592 bytes, Path Length: 0.</p>

	Greedy Best First Search (GBFS)	
	A*	

Test Case 6

Masukkan	Algoritma	Hasil
----------	-----------	-------

<p>Start Word : atlases End Word : chapter</p>	<p>Uniform Cost Search (UCS)</p>	 <p>The screenshot shows the 'Word Ladder Solver' application window. It has three input fields: 'Start Word' (atlases), 'End Word' (chapter), and 'Algorithm' (UCS). A 'Find Shortest Ladder' button is present. Below the inputs, there are three scrollable lists of words, each preceded by a status bar showing 'Node Count: 957', 'Used Memory: 20034656 bytes', 'Execution Time: 0.0532711 s', and 'Path Length: 22'.</p> <p>Shortest Ladder:</p> <ul style="list-style-type: none"> atlases anlases anlaces unlaces unlaced unladed unfaded unfaked <p>Word List 1:</p> <ul style="list-style-type: none"> uncaked uncakes uncases uneases ureases creases resses chesses chasses <p>Word List 2:</p> <ul style="list-style-type: none"> chassed chasmed charmed charmer charter chapter
--	----------------------------------	---

Greedy Best First Search
(GBFS)

Word Ladder Solver

Start Word:

atlasses

End Word:

chapter

Algorithm:

GBFS

Find Shortest Ladder

Shortest Ladder:

atlasses
anlases
anlases
unlases
enlases
inlases
inlaced
unlaced

Node Count: 120

Execution Time: 0.0071553 s

Used Memory: 42054752 bytes

Path Length: 38

unladed
unfaded
unfazed
unrazed
unrated
undated
unbated
unbased
uncased

Node Count: 120

Execution Time: 0.0071553 s

Used Memory: 42054752 bytes

Path Length: 38

uncases
uneases
ureases
creases
creates
created
cheated
cheater
cheaper

Node Count: 120

Execution Time: 0.0071553 s

Used Memory: 42054752 bytes

Path Length: 38

cheeper
cheeped
cheeked
cheesed
cheeses
chesses
chasses
chassed
chasmed

Node Count: 120

Execution Time: 0.0071553 s

Used Memory: 42054752 bytes

Path Length: 38

		<div> <div> charmed charted charter chapter </div> <div> Node Count: 120 Used Memory: 42054752 bytes </div> <div> Execution Time: 0.0071553 s Path Length: 38 </div> </div>
	A*	<div> <div> <div>Word Ladder Solver</div> <div> <div>Start Word: atlases</div> <div>End Word: chapter</div> <div>Algorithm: A*</div> <div>Find Shortest Ladder</div> </div> <div> Shortest Ladder: atlases anlases anlaces unlaces unlades unladed unfaded unfaked </div> <div> Node Count: 216 Used Memory: 63026272 bytes </div> <div> Execution Time: 0.0090999 s Path Length: 22 </div> </div> <div> <div> uncaked uncakes uncases uneases ureases creases cresses chesses chasses </div> <div> Node Count: 216 Used Memory: 63026272 bytes </div> <div> Execution Time: 0.0090999 s Path Length: 22 </div> </div> <div> <div> chasmed charmed charted charter chapter </div> <div> Node Count: 216 Used Memory: 63026272 bytes </div> <div> Execution Time: 0.0090999 s Path Length: 22 </div> </div> </div>

D. Analisis Perbandingan Solusi UCS, Greedy Best First Search, dan A*

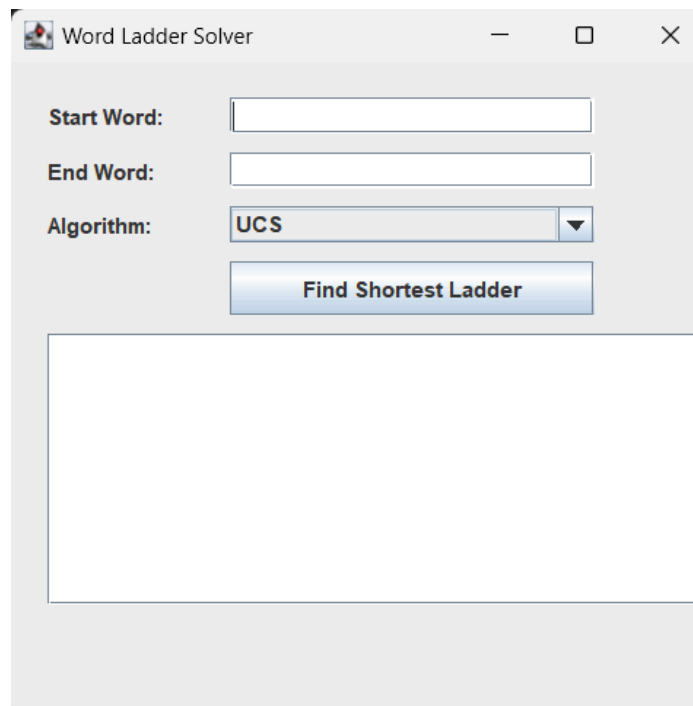
Test Case	UCS			GBFS			A*		
	Panjang Solusi	Waktu Eksekusi	Jumlah Memori	Panjang Solusi	Waktu Eksekusi	Jumlah Memori	Panjang Solusi	Waktu Eksekusi	Jumlah Memori

		(s)	(bytes)		(s)	(bytes)		(s)	(bytes)
1	-	-	-	-	-	-	-	-	-
2	4	0.058	23034880	4	0.006	34569216	4	0.003	7815344
3	7	0.005	47367104	9	0.003	56804288	7	0.004	10657512
4	9	0.305	19465616	22	0.004	32048528	9	0.010	58262928
5	0	0.004	76698592	0	0.005	88232928	0	0.004	98718688
6	22	0.053	20034656	38	0.007	42054752	22	0.009	63026272

Berdasarkan tabel perbandingan hasil test case diatas, dapat dilihat bahwa algoritma A* adalah algoritma terbaik dalam mencari solusi karena dapat menghasilkan solusi yang optimal serta memiliki waktu eksekusi yang cepat. Algoritma UCS dan A* selalu menghasilkan hasil yang optimal hanya saja waktu eksekusi algoritma UCS jauh lebih lambat. Algoritma GBFS hampir selalu memiliki waktu eksekusi yang lebih cepat dibandingkan dengan algoritma lainnya, tetapi solusi yang dihasilkan tidak selalu optimal. Jumlah memori yang dibutuhkan untuk ketiga algoritma berbeda-beda bergantung dengan panjang kata yang diuji. Untuk kata dengan banyak huruf yang sedikit atau kata yang tidak terlalu panjang, algoritma A* membutuhkan memori yang paling sedikit diikuti dengan algoritma UCS baru kemudian GBFS. Sedangkan untuk kata dengan banyak huruf yang lumayan besar, algoritma UCS membutuhkan memori paling sedikit diikuti dengan algoritma GBFS baru kemudian algoritma A*.

E. Penjelasan Implementasi Bonus

Dalam kelas Main, digunakan pustaka Java Swing untuk membuat antarmuka pengguna (GUI). Antarmuka ini terdiri dari JFrame sebagai jendela utama yang menampung komponen-komponen GUI lainnya. Terdapat JLabel untuk menampilkan teks statis seperti "Start Word:", "End Word:", dan sebagainya. Pengguna dapat memasukkan kata awal dan akhir melalui JTextField. Untuk memilih algoritma pencarian, digunakan JComboBox dengan opsi "UCS", "GBFS", dan "A*". Sebuah JButton memulai pencarian jalur terpendek, dan hasilnya ditampilkan dalam JTextArea yang dapat digulir menggunakan JScrollPane. Digunakan juga SwingUtilities.invokeLater() untuk memastikan pembuatan GUI berlangsung di dalam thread UI utama.



Lampiran

Link Repository Github : https://github.com/saadabha/Tucil3_13522092

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	✓	