

LAPORAN TUGAS KECIL 3 IF2211

STRATEGI ALGORITMA

Word Ladder Solver Menggunakan Uniform Cost Search, Greedy Best First Search, dan A* Search



Disusun Oleh:
Elijah Darrellshane Suryanegara 13522097

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2023/2024

Daftar Isi

BAB I	
DESKRIPSI MASALAH	2
BAB II	
LANDASAN TEORI	3
Uniform Cost Search (UCS)	3
Greedy Best-First Search (GBFS)	4
A* Search	5
BAB III	
IMPLEMENTASI	7
Analisis	7
Pros & Cons	7
$f(n)$, $g(n)$, dan $h(n)$	7
Heuristik pada A*	7
Efisiensi	8
Implementasi	9
SearchResult.java	9
UniformCostSearch.java	9
GreedyBestFirstSearch.java	12
AStarSearch.java	16
WordLadderSolverGUI.java	20
BAB IV	
PENGUJIAN	26
UCS	26
GBFS	32
A*	38
Tabel Pengujian	43
Hasil Pengujian	44
Execution Time	44
Path Length	44
Visited Nodes	44
Used Memory	44
Bonus	44
BAB V	
KESIMPULAN DAN SARAN	45
Kesimpulan	45
Saran	45
DAFTAR PUSTAKA	46
LAMPIRAN	46

BAB I

DESKRIPSI MASALAH

Word ladder (juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragrams, laddergrams, atau word golf) adalah salah satu permainan kata yang terkenal bagi seluruh kalangan. Word ladder ditemukan oleh Lewis Carroll, seorang penulis dan matematikawan, pada tahun 1877. Pada permainan ini, pemain diberikan dua kata yang disebut sebagai start word dan end word. Untuk memenangkan permainan, pemain harus menemukan rantai kata yang dapat menghubungkan antara start word dan end word. Banyaknya huruf pada start word dan end word selalu sama. Tiap kata yang berdekatan dalam rantai kata tersebut hanya boleh berbeda satu huruf saja. Pada permainan ini, diharapkan solusi optimal, yaitu solusi yang meminimalkan banyaknya kata yang dimasukkan pada rantai kata. Berikut adalah ilustrasi serta aturan permainan



Gambar 1. Ilustrasi dan Peraturan Permainan Word Ladder
(Sumber: <https://wordwormdormdork.com/>)

BAB II

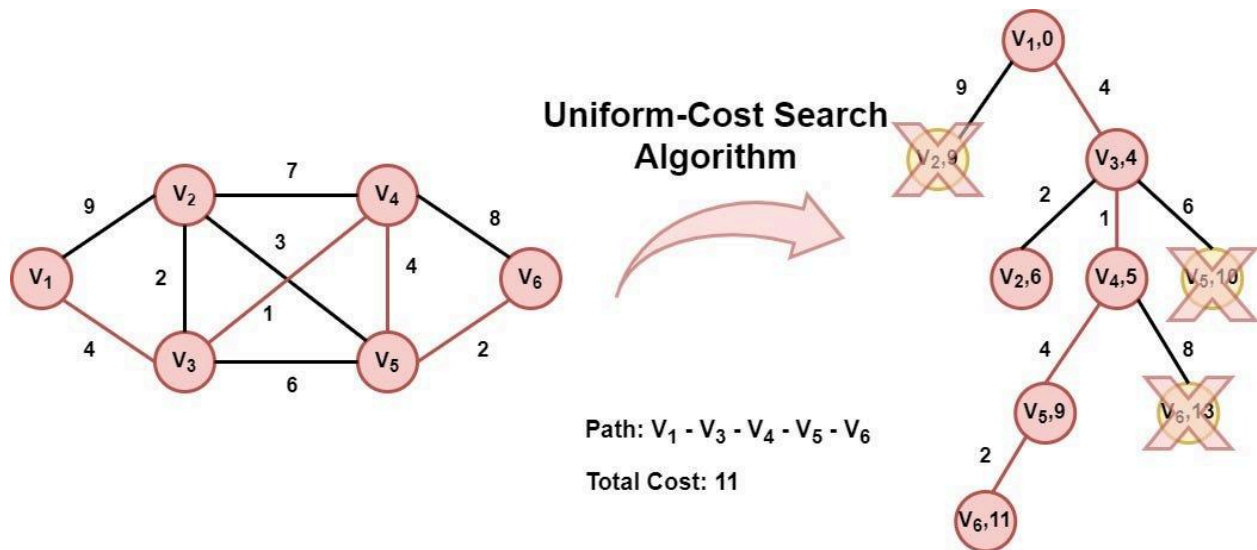
LANDASAN TEORI

Uniform Cost Search (UCS)

Uniform-Cost Search (UCS) adalah variasi dari algoritma Dijkstra. Algoritma UCS digunakan untuk menemukan jalur minimum dari node sumber ke node tujuan dalam sebuah graf berarah berbobot. Algoritma pencarian ini menggunakan pendekatan *brute force*, mengunjungi semua node berdasarkan bobot saat ini, dan menemukan jalur dengan biaya minimum dengan secara berulang memeriksa semua jalur yang mungkin. Kita menggunakan array *boolean visited* dan *priority queue* untuk menemukan biaya minimum. Node dengan biaya minimum memiliki prioritas tertinggi. Algoritma ini menggunakan *uninformed/blind search* karena tidak ada informasi sebelumnya tentang node.

Langkah-langkah:

1. Buat sebuah antrian prioritas, sebuah array *boolean visited* dengan ukuran jumlah node, dan sebuah variabel *min_cost* diinisialisasi dengan nilai maksimum. Tambahkan node sumber ke dalam antrian dan tandai sebagai telah dikunjungi.
2. Ambil elemen dengan prioritas tertinggi dari antrian. Jika node yang dihapus adalah node tujuan, periksa variabel *min_cost*. Jika nilai variabel *min_cost* lebih besar dari biaya saat ini, maka perbarui variabel tersebut.
3. Jika node yang diberikan bukanlah node tujuan, maka tambahkan semua node yang belum dikunjungi ke dalam antrian prioritas yang berdekatan dengan node saat ini.



Gambar 2. Ilustrasi Algoritma *Uniform Cost Search* (UCS)

(Sumber: <https://plainenglish.io/blog/uniform-cost-search-ucs-algorithm-in-python-ec3ee03fca9f>)

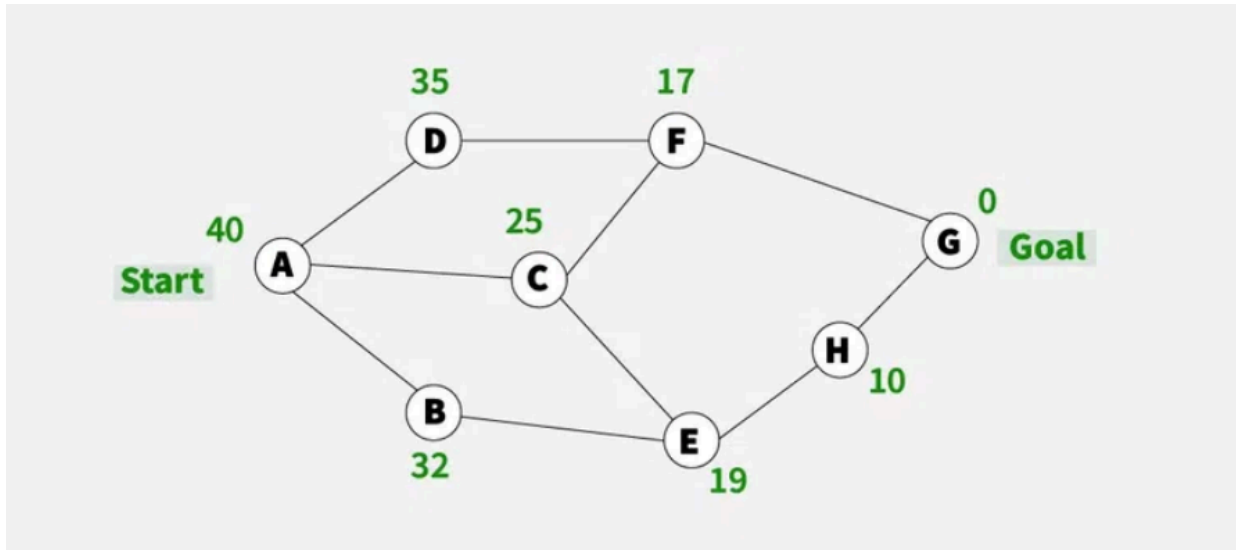
Greedy Best-First Search (GBFS)

Greedy best-first search adalah *informed search algorithm* yang mana fungsi evaluasi sama persis dengan fungsi heuristik, mengabaikan bobot tepi dalam sebuah graf berbobot karena hanya nilai heuristik yang dipertimbangkan. Dinamakan "*greedy*" karena pada setiap langkahnya mencoba mendekati tujuan sebanyak mungkin. Untuk mencari node tujuan, algoritma ini mengembangkan node yang paling dekat dengan tujuan seperti yang ditentukan oleh fungsi heuristik. Pendekatan ini mengasumsikan bahwa hal tersebut mungkin mengarah pada solusi dengan cepat. Namun, solusi dari pencarian *greedy best-first* mungkin tidak optimal karena mungkin ada jalur yang lebih pendek. Dalam algoritma ini, biaya pencarian minimal karena solusi ditemukan tanpa mengembangkan node yang tidak berada pada jalur solusi. Walau algoritma minimal, tetapi algoritma berpotensi tidak lengkap karena dapat mengarah ke jalan buntu.

Langkah-langkah:

1. Inisialisasi sebuah *tree* dengan *root node* sebagai node awal dalam *open list*.
2. Jika *open list* kosong dan *goal node* belum ditemukan, kembalikan gagal. Jika tidak, tambahkan *node* saat ini ke dalam daftar tertutup. Hapus *node* dengan nilai $h(x)$ terendah dari daftar terbuka untuk dieksplorasi.

3. Jika sebuah *child node* adalah goal node, kembalikan keberhasilan. Jika tidak, jika *node* belum ada di dalam *open list* maupun *closed*, tambahkan ke dalam *open list* untuk dieksplorasi.



Gambar 3. Ilustrasi Algoritma *Greedy Best-First Search* (GBFS)
(Sumber: <https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/>)

A* Search

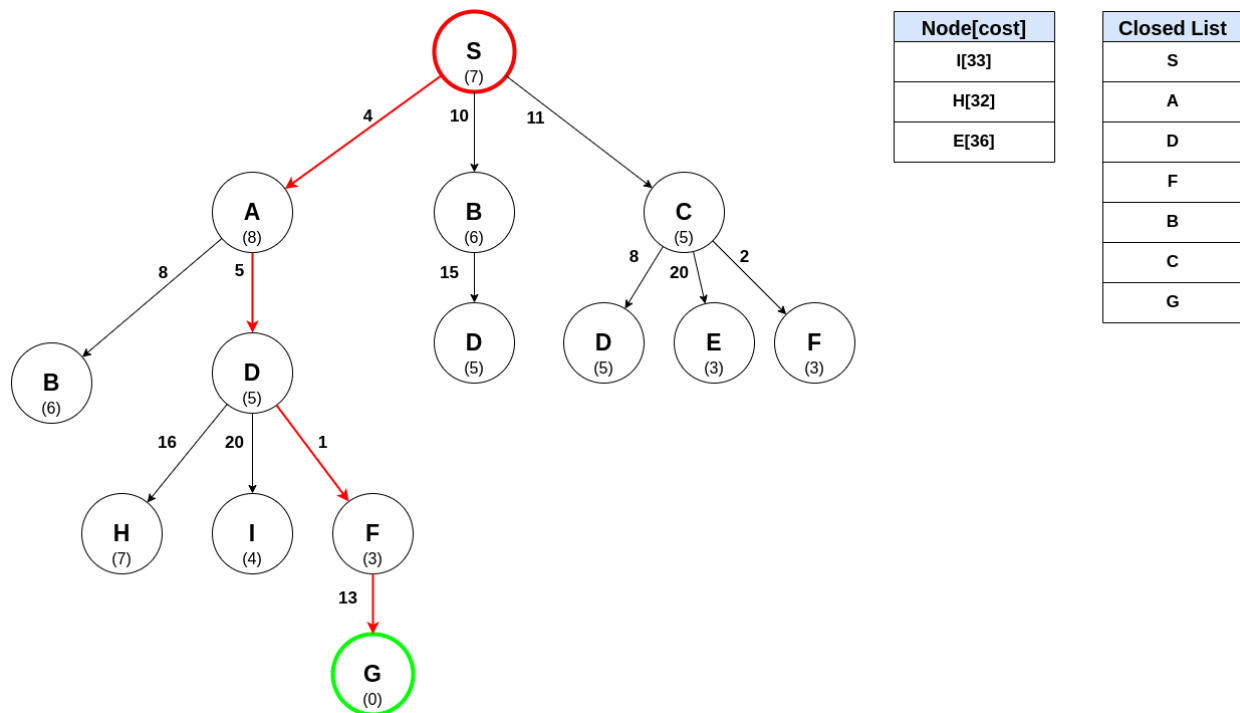
A* Search adalah algoritma pencarian terbaik yang diinformasikan yang secara efisien menentukan jalur biaya terendah antara dua node dalam sebuah graf berarah berbobot dengan bobot tepi non-negatif. Algoritma ini merupakan varian dari algoritma Dijkstra.

Diberikan sebuah graf berbobot dengan bobot tepi non-negatif, untuk menemukan jalur dengan biaya terendah dari sebuah node awal S ke sebuah node tujuan G, dua daftar digunakan:

1. *Open list*, diimplementasikan sebagai *priority queue*, yang menyimpan node-node selanjutnya yang akan dieksplorasi. Karena ini adalah *priority queue*, node kandidat yang paling menjanjikan (yang memiliki nilai terendah dari fungsi evaluasi) selalu berada di bagian atas. Awalnya, satu-satunya node dalam daftar ini adalah node awal S.
2. Daftar tertutup yang menyimpan node-node yang sudah dievaluasi. Ketika sebuah node berada dalam daftar tertutup, itu berarti jalur dengan biaya terendah ke node tersebut telah ditemukan.

Untuk menemukan jalur dengan biaya terendah, sebuah pohon pencarian dibangun dengan cara sebagai berikut:

1. Inisialisasi pohon dengan node akar yang merupakan node awal S.
2. Hapus node teratas dari daftar terbuka untuk dieksplorasi.
3. Tambahkan node saat ini ke dalam daftar tertutup.
4. Tambahkan semua node yang memiliki tepi masuk dari node saat ini sebagai node anak dalam pohon.
5. Perbarui biaya terendah untuk mencapai node anak.
6. Hitung fungsi evaluasi untuk setiap node anak dan tambahkan mereka ke dalam daftar terbuka.



Gambar 4. Ilustrasi Algoritma *Uniform Cost Search* (UCS)

(Sumber: <https://www.codecademy.com/resources/docs/ai/search-algorithms/a-star-search>)

BAB III IMPLEMENTASI

Analisis

a) Pros & Cons

Uniform Cost Search (UCS) adalah metode pencarian yang *complete*, dalam arti menjamin ditemukannya solusi yang optimal. Namun, menggunakan UCS menghasilkan manajemen memori yang buruk karena banyaknya node yang dikunjungi secara *redundant*. Di sisi lain, Greedy Best-First Search (GBFS) adalah metode pencarian yang lebih efisien dalam penggunaan memory, tetapi tidak *complete* karena hasilnya hanya ditinjau secara lokal. Dalam kasus ini, algoritma A* menjadi alternatif solusi yang memiliki manajemen memori yang lebih efisien seperti GBFS, tetapi tetap mempertahankan sifat *complete* dari UCS. *Best of both worlds*.

b) $f(n)$, $g(n)$, dan $h(n)$

UCS

$$f(n) = g(n)$$

GBFS

$$f(n) = h(n)$$

A*

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

$f(n)$: Total estimasi biaya mencapai tujuan melalui simpul n.

$g(n)$: Jumlah biaya dari node awal (*root*) ke node saat ini (*current*). Pada kasus Word Ladder, $g(n)$ didefinisikan sebagai jumlah langkah yang harus ditempuh untuk mencapai *current node*.

$h(n)$: Estimasi jarak atau biaya antara node saat ini (*current*) sampai node tujuan (*goal*). Pada kasus Word Ladder, nilai heuristik didapatkan dengan menghitung jumlah huruf yang berbeda antara kata yang sedang diproses saat ini dengan kata target.

c) Heuristik pada A*

Fungsi heuristik dalam Greedy Best-First Search (GBFS) dikatakan *admissible* jika memberikan perkiraan yang tidak melebihi biaya sebenarnya (*underestimate*) untuk mencapai tujuan. Jika kita lihat, heuristik yang digunakan pada Greedy Best-First Search (GBFS) yang kemudian juga digunakan pada A* didapatkan dengan menghitung Hamming distance, pada kasus ini dalam wujud jumlah huruf yang berbeda antara

current word dengan *goal word*. Mengingat hanya dapat terjadi perubahan pada 1 huruf per gerakan, dibutuhkan setidaknya n gerakan untuk n huruf yang berbeda. Dalam kata lain, heuristik yang didapat dari Hamming distance sudah merupakan nilai minimum dengan asumsi bahwa hanya dibutuhkan 1 gerakan per huruf yang berbeda untuk mencapai *goal word* dan ini belum tentu valid karena ada kemungkinan bahwa tidak ada kata valid yang mungkin dibentuk dari mengubah sebuah huruf pada *start word* menyesuaikan dengan huruf yang terdapat pada *goal word*. Karena dapat dipastikan bahwasanya tidak mungkin jumlah langkah yang optimal lebih kecil daripada nilai *heuristic*, dapat disimpulkan bahwa *heuristic* yang kita gunakan selalu *underestimate* sehingga tergolong *admissible*.

d) Efisiensi

Algoritma A* dianggap lebih efisien daripada Uniform Cost Search (UCS) dalam kasus Word Ladder. Hal ini disebabkan A* menggabungkan biaya yang telah dikeluarkan (cost from start) dan estimasi biaya untuk mencapai tujuan (heuristic cost to goal) untuk menentukan simpul mana yang akan dieksplorasi selanjutnya. Dalam konteks Word Ladder, UCS dan Breadth First Search (BFS) akan memberikan urutan simpul dan jalur yang identik karena priority queue diurutkan berdasarkan jumlah langkah yang hanya di-*increment*, sama seperti mengunjungi masing-masing kata sesuai urutannya pada tree menggunakan BFS.

UCS tidak memperhitungkan estimasi jarak dari simpul saat ini ke tujuan. UCS secara sistematis menjelajahi setiap kedalaman penuh sebelum beralih ke kedalaman berikutnya, yang dapat dianggap sebagai pendekatan brute-force karena menjelajahi semua kemungkinan tanpa preferensi. Di sisi lain, A* memprioritaskan simpul berdasarkan estimasi biaya total yang lebih rendah, memungkinkan algoritma untuk fokus pada jalur yang lebih menjanjikan. Hal ini mengurangi jumlah simpul yang perlu dieksplorasi, membuat A* lebih efisien dari UCS dalam situasi di mana prediksi heuristik dapat digunakan untuk mengarahkan pencarian.

GBFS sendiri dalam konteks Word Ladder sebenarnya mencapai efisiensi yang paling tinggi karena hanya memproses simpul-simpul yang mendekati *goal word*. Fungsi *heuristic* yang digunakan menjamin pemrosesan node-node yang redundant diabaikan sama sekali. Namun, hal ini sekaligus menjadi pedang bermata dua karena solusi bersifat

lokal dan dapat berimbas pada tidak ditemukannya solusi yang optimal pada kasus-kasus ketika pemrosesan awal melibatkan banyak kata antara sebelum digantikan dengan huruf-huruf dari *goal word*.

Implementasi

a) SearchResult.java

```
package com.henryofskalitz;
import java.util.*;

public class SearchResult {
    private List<String> foundPath;
    private int visitedWordsLength;

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

    public List<String> getFoundPath() {
        return foundPath;
    }

    public int getVisitedWordsLength() {
        return visitedWordsLength;
    }
}
```

SearchResult digunakan untuk menyimpan hasil pencarian jalur dan informasi terkait jumlah node yang dikunjungi. Kelas ini memiliki dua atribut yaitu foundPath yang merupakan daftar (*List*) dari *string* yang mewakili jalur yang ditemukan, dan visitedWordsLength yang merupakan panjang (*length*) dari kata-kata yang telah dikunjungi selama pencarian.

b) UniformCostSearch.java

```

package com.henryofskalitz;
import java.util.*;

public class UniformCostSearch {

    // Function to generate all possible words by changing one
    letter
    private static List<String> generateWords(String word,
Set<String> wordSet) {
        String alphabet = "abcdefghijklmnopqrstuvwxyz";
        List<String> words = new ArrayList<>();
        for (int i = 0; i < word.length(); i++) {
            for (char letter : alphabet.toCharArray()) {
                String newWord = word.substring(0, i) + letter +
word.substring(i + 1);
                if (!newWord.equals(word) &&
wordSet.contains(newWord)) {
                    words.add(newWord);
                }
            }
        }
        return words;
    }

    // Function to perform uniform cost search
    public static SearchResult findWithUCS(String startWord,
String endWord, Set<String> wordSet) {
        Set<String> visitedWords = new HashSet<>();
        PriorityQueue<Node> queue = new
PriorityQueue<>(Comparator.comparingInt(node -> node.cost));
        List<String> foundPath = new ArrayList<>();
        queue.offer(new Node(startWord, 0, null));

        while (!queue.isEmpty()) {
            Node currentNode = queue.poll();
            String currentWord = currentNode.word;

            if (currentWord.equals(endWord)) {
                // Construct the found path

```

```

        Node node = currentNode;
        while (node != null) {
            foundPath.add(0, node.getWord()); // Add word
at the beginning to maintain order
            node = node.getParent();
        }

        return new SearchResult(foundPath,
visitedWords.size());
    }

    if (!visitedWords.contains(currentWord)) {
        visitedWords.add(currentWord);
        List<String> neighbors =
generateWords(currentWord, wordSet);
        for (String neighbor : neighbors) {
            if (!visitedWords.contains(neighbor)) {
                queue.offer(new Node(neighbor,
currentNode.cost + 1, currentNode));
            }
        }
    }
}

return new SearchResult(null, visitedWords.size());
}

// Node class to represent word with its cost and parent
private static class Node {
    String word;
    int cost;
    Node parent;

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

```

```

// Getter and setter methods for word and cost
public String getWord() {
    return word;
}

public int getCost() {
    return cost;
}

public Node getParent() {
    return parent;
}
}

```

UniformCostSearch mengimplementasikan algoritma Uniform Cost Search, yang merupakan varian dari algoritma Dijkstra yang digunakan untuk menemukan jalur terpendek dalam sebuah graf berbobot. Dalam konteks ini, graf mewakili jaringan kata di mana setiap kata terhubung ke kata lain dengan mengubah satu huruf. Metode findWithUCS mengambil kata awal, kata akhir, dan sebuah set kata valid sebagai input, kemudian melakukan pencarian cost seragam untuk menemukan jalur terpendek dari kata awal ke kata akhir melalui kata-kata yang valid. Ini menjaga sebuah antrian prioritas dari simpul (kata) yang diurutkan berdasarkan biaya mereka, memperluas simpul untuk menjelajahi kata-kata tetangga sampai mencapai kata akhir atau menghabiskan semua kemungkinan. Metode generateWords menghasilkan kata-kata tetangga dengan mengganti satu huruf dalam kata yang diberikan dengan setiap huruf abjad, dan kelas Node mewakili sebuah kata beserta dengan biayanya dan kata induknya dalam pohon pencarian. Terakhir, itu mengembalikan objek SearchResult yang berisi jalur yang ditemukan dan jumlah kata yang telah dikunjungi.

c) GreedyBestFirstSearch.java

```

package com.henryofskalitz;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;

```

```

import java.util.Set;

public class GreedyBestFirstSearch {
    private static List<String> generateWords(String word,
Set<String> wordSet) {
        String alphabet = "abcdefghijklmnopqrstuvwxyz";
        List<String> words = new ArrayList<>();
        for (int i = 0; i < word.length(); i++) {
            for (char letter : alphabet.toCharArray()) {
                String newWord = word.substring(0, i) + letter +
word.substring(i + 1);
                if (!newWord.equals(word) &&
wordSet.contains(newWord)) {
                    words.add(newWord);
                }
            }
        }
        return words;
    }

    private static int countLetterDifference(String word1, String
word2){
        if (word1.length() != word2.length()) {
            return 0;
        }

        int count = 0;
        for (int i = 0; i < word1.length(); i++) {
            if (word1.charAt(i) != word2.charAt(i)) {
                count++;
            }
        }

        return count;
    }

    public static SearchResult findWithGBFS(String startWord,
String goalWord, Set<String> dictionary) {
        PriorityQueue<Node> priorityQueue = new PriorityQueue<>();

```

```

List<String> foundPath = new ArrayList<>();
Set<String> visitedWords = new HashSet<>();

int diff = countLetterDifference(startWord, goalWord);
priorityQueue.add(new Node(startWord, diff, null));

while (!priorityQueue.isEmpty()) {
    Node currentNode = priorityQueue.poll();
    String currentWord = currentNode.getWord();

    if (currentWord.equals(goalWord)) {
        // Construct the found path
        Node node = currentNode;
        while (node != null) {
            foundPath.add(0, node.getWord()); // Add word
at the beginning to maintain order
            node = node.getParent();
        }

        return new SearchResult(foundPath,
visitedWords.size());
    }

    // Add current word to visited words
    visitedWords.add(currentWord);

    // Generate neighboring words
    List<String> neighbors = generateWords(currentWord,
dictionary);
    for (String word : neighbors) {
        diff = countLetterDifference(word, goalWord);
        if (!visitedWords.contains(word)) {
            Node newNode = new Node(word, diff,
currentNode);
            priorityQueue.add(newNode);
        }
    }
}

```

```

        // If no path is found, return an empty list
        return new SearchResult(null, visitedWords.size());
    }

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

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

        // Getter and setter methods for word and cost
        public String getWord() {
            return word;
        }

        public int getCost() {
            return cost;
        }

        public Node getParent() {
            return parent;
        }

        @Override
        public int compareTo(Node other) {
            return Integer.compare(cost, other.cost);
        }
    }
}

```

GreedyBestFirstSearch mengimplementasikan algoritma Greedy Best First Search (GBFS), yang merupakan algoritma pencarian yang menggunakan pendekatan greedy untuk mencari solusi terdekat dari titik awal ke titik tujuan dalam graf berbobot. Dalam konteks ini, graf mewakili jaringan kata di mana setiap kata terhubung ke kata lain

dengan mengubah satu huruf. Metode `findWithGBFS` mengambil kata awal, kata tujuan, dan sebuah kamus kata valid sebagai input, kemudian melakukan pencarian greedy untuk menemukan jalur terpendek dari kata awal ke kata tujuan melalui kata-kata yang valid. Ini menjaga sebuah antrian prioritas dari simpul (kata) yang diurutkan berdasarkan perbedaan huruf antara kata saat ini dan kata tujuan, memilih simpul dengan perbedaan huruf terkecil. Metode `generateWords` menghasilkan kata-kata tetangga dengan mengganti satu huruf dalam kata yang diberikan dengan setiap huruf abjad, dan metode `countLetterDifference` menghitung jumlah perbedaan huruf antara dua kata. Kelas `Node` mewakili sebuah kata beserta dengan biayanya dan kata induknya dalam pohon pencarian, dan mengimplementasikan antarmuka `Comparable` untuk memungkinkan perbandingan berdasarkan biaya. Terakhir, itu mengembalikan objek `SearchResult` yang berisi jalur yang ditemukan dan jumlah kata yang telah dikunjungi.

d) `AStarSearch.java`

```
package com.henryofskalitz;
import java.util.*;

public class AStarSearch {

    private static int calculateCost(String word, String goalWord)
    {
        // Here, we can use the countLetterDifference method as
        the cost function
        return countLetterDifference(word, goalWord);
    }

    public static SearchResult findWithAStar(String startWord,
String goalWord, Set<String> dictionary) {
        PriorityQueue<Node> priorityQueue = new PriorityQueue<>();
        Map<String, Integer> costs = new HashMap<>();
        List<String> foundPath = new ArrayList<>();
        Set<String> visitedWords = new HashSet<>();

        // Initialize the start node with cost 0 and heuristic
        value
        priorityQueue.add(new Node(startWord, 0,
calculateCost(startWord, goalWord), null));
        costs.put(startWord, 0);
```

```

while (!priorityQueue.isEmpty()) {
    Node currentNode = priorityQueue.poll();
    String currentWord = currentNode.getWord();

    if (currentWord.equals(goalWord)) {
        // Construct the found path
        Node node = currentNode;
        while (node != null) {
            foundPath.add(0, node.getWord()); // Add word
at the beginning to maintain order
            node = node.getParent();
        }
        return new SearchResult(foundPath,
visitedWords.size());
    }

    // Add current word to visited words
    visitedWords.add(currentWord);

    // Generate neighboring words
    List<String> neighbors = generateWords(currentWord,
dictionary);
    for (String word : neighbors) {
        int newCost = costs.get(currentWord) + 1;
        if (!costs.containsKey(word) || newCost <
costs.get(word)) {
            int heuristic = calculateCost(word, goalWord);
            Node newNode = new Node(word, newCost,
heuristic, currentNode);
            priorityQueue.add(newNode);
            costs.put(word, newCost);
        }
    }
}

// If no path is found, return an empty list
return new SearchResult(null, visitedWords.size());
}

```

```

        private static List<String> generateWords(String word,
Set<String> wordSet) {
            String alphabet = "abcdefghijklmnopqrstuvwxyz";
            List<String> words = new ArrayList<>();
            for (int i = 0; i < word.length(); i++) {
                for (char letter : alphabet.toCharArray()) {
                    String newWord = word.substring(0, i) + letter +
word.substring(i + 1);
                    if (!newWord.equals(word) &&
wordSet.contains(newWord)) {
                        words.add(newWord);
                    }
                }
            }
            return words;
        }

        private static int countLetterDifference(String word1, String
word2) {
            if (word1.length() != word2.length()) {
                return 0;
            }

            int count = 0;
            for (int i = 0; i < word1.length(); i++) {
                if (word1.charAt(i) != word2.charAt(i)) {
                    count++;
                }
            }

            return count;
        }

        private static class Node implements Comparable<Node> {
            private String word;
            private int cost;
            private int heuristic;
            private Node parent;

```

```

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

        public String getWord() {
            return word;
        }

        // public int getCost() {
        //     return cost;
        // }

        // public int getHeuristic() {
        //     return heuristic;
        // }

        public Node getParent() {
            return parent;
        }

        @Override
        public int compareTo(Node other) {
            int f1 = cost + heuristic;
            int f2 = other.cost + other.heuristic;
            return Integer.compare(f1, f2);
        }
    }
}

```

AStarSearch mengimplementasikan algoritma A* Search yang merupakan algoritma pencarian yang menggunakan pendekatan kombinasi dari biaya aktual dan estimasi heuristik untuk mencari solusi terdekat dari titik awal ke titik tujuan dalam graf berbobot. Dalam konteks ini, graf mewakili jaringan kata di mana setiap kata terhubung ke kata lain dengan mengubah satu huruf. Metode findWithAStar mengambil kata awal,

kata tujuan, dan sebuah kamus kata valid sebagai input, kemudian melakukan pencarian A* untuk menemukan jalur terpendek dari kata awal ke kata tujuan melalui kata-kata yang valid. Ini menjaga sebuah antrian prioritas dari simpul (kata) yang diurutkan berdasarkan nilai $f = \text{biaya aktual} + \text{estimasi heuristik}$, dengan estimasi heuristik dihitung menggunakan metode `calculateCost`. Metode `generateWords` menghasilkan kata-kata tetangga dengan mengganti satu huruf dalam kata yang diberikan dengan setiap huruf abjad, dan metode `countLetterDifference` menghitung jumlah perbedaan huruf antara dua kata. Kelas `Node` mewakili sebuah kata beserta dengan biayanya, estimasi heuristiknya, dan kata induknya dalam pohon pencarian, dan mengimplementasikan antarmuka `Comparable` untuk memungkinkan perbandingan berdasarkan nilai f . Terakhir, itu mengembalikan objek `SearchResult` yang berisi jalur yang ditemukan dan jumlah kata yang telah dikunjungi.

e) `WordLadderSolverGUI.java`

```
package com.henryofskalitz;

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.FileChooser;
import javafx.stage.Stage;

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

public class WordLadderSolverGUI extends Application {
```

```

private TextField startWordField;
private TextField endWordField;
private ComboBox<String> algorithmComboBox;
private TextArea resultTextArea;

private Set<String> wordList;
File selectedFile;

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

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

    GridPane gridPane = new GridPane();
    gridPane.setPadding(new Insets(20, 20, 20, 20));
    gridPane.setVgap(10);
    gridPane.setHgap(10);

    Label startLabel = new Label("Start Word:");
    GridPane.setConstraints(startLabel, 0, 0);

    startWordField = new TextField();
    GridPane.setConstraints(startWordField, 1, 0);

    Label endLabel = new Label("End Word:");
    GridPane.setConstraints(endLabel, 0, 1);

    endWordField = new TextField();
    GridPane.setConstraints(endWordField, 1, 1);

    Label algorithmLabel = new Label("Algorithm:");
    GridPane.setConstraints(algorithmLabel, 0, 2);

    algorithmComboBox = new ComboBox<>();
    algorithmComboBox.getItems().addAll("Uniform Cost Search

```

```

(UCS)", "Greedy Best First Search (GBFS)", "A* Search");
    algorithmComboBox.setValue("Uniform Cost Search (UCS)");
// Default selection
    GridPane.setConstraints(algorithmComboBox, 1, 2);

    Button uploadButton = new Button("Upload Dictionary");
    GridPane.setConstraints(uploadButton, 0, 3);
    uploadButton.setOnAction(e -> uploadDictionary());

    Button solveButton = new Button("Solve");
    GridPane.setConstraints(solveButton, 1, 3);
    solveButton.setOnAction(e -> solve());

    resultTextArea = new TextArea();
    resultTextArea.setEditable(false);
    GridPane.setConstraints(resultTextArea, 0, 4, 2, 1);

    gridPane.getChildren().addAll(startLabel, startWordField,
endLabel, endWordField, algorithmLabel, algorithmComboBox,
uploadButton, solveButton, resultTextArea);

    Scene scene = new Scene(gridPane, 400, 300);
    primaryStage.setScene(scene);
    primaryStage.show();
}

private File uploadDictionary() {
    FileChooser fileChooser = new FileChooser();
    fileChooser.setTitle("Upload Dictionary File");
    selectedFile = fileChooser.showOpenDialog(null);
    if (selectedFile != null) {
        return selectedFile;
    }else{
        return null;
    }
}

public static Set<String> findWordsWithLength(File fileName,
int length) {

```

```

        Set<String> words = new HashSet<>();

        if(fileName == null){
            return null;
        }

        try (BufferedReader br = new BufferedReader(new
FileReader(fileName))) {
            String line;
            while ((line = br.readLine()) != null) {
                String[] wordList = line.split("\\s+");
                for (String word : wordList) {
                    if (word.length() == length) {
                        words.add(word);
                    }
                }
            }
        } catch (IOException e) {
            System.err.println("Error reading the file: " +
e.getMessage());
        }

        return words;
    }

    private void solve(){
        String startWord = startWordField.getText();
        String goalWord = endWordField.getText();
        String selectedAlgorithm = algorithmComboBox.getValue();

        wordList = findWordsWithLength(selectedFile,
startWord.length());

        if (wordList == null || wordList.isEmpty()) {
            resultTextArea.setText("Please upload a dictionary
first.");
            return;
        }
    }

```



```

        // Print the words found
        if (wordList.isEmpty()){
            resultTextArea.setText("No words found with length " +
startWord.length());
            return;
        }

        Runtime runtime = Runtime.getRuntime();
        runtime.gc();
        long startTime = System.nanoTime();
        long startMemory = runtime.totalMemory() -
runtime.freeMemory();
        SearchResult result = null;
        if(selectedAlgorithm == "Uniform Cost Search (UCS){
            result = UniformCostSearch.findWithUCS(startWord,
goalWord, wordList);
        }else if(selectedAlgorithm == "Greedy Best First Search
(GBFS)"){
            result = GreedyBestFirstSearch.findWithGBFS(startWord,
goalWord, wordList);
        }else if(selectedAlgorithm == "A* Search"){
            result = AStarSearch.findWithAStar(startWord,
goalWord, wordList);
        }
        long endTime = System.nanoTime();
        long afterMemory = runtime.totalMemory() -
runtime.freeMemory();

        long elapsedTimeInNanoseconds = endTime - startTime;
        double elapsedTimeInMilliseconds = (double)
elapsedTimeInNanoseconds / 1_000_000;
        long usedMemory = afterMemory - startMemory;

        // Extract the found path and visited words length from
the result
        List<String> foundPath = result.getFoundPath();
        int visitedWordsLength = result.getVisitedWordsLength();

```

```

        String resultString = "Using " + selectedAlgorithm +
"...\\n";
        if(foundPath == null){
            resultString += "No path found!";
        }else{
            // Handle the result as needed, such as displaying it
to the user
            resultString += String.join(" -> ", foundPath);
            resultString += "\\nVisited nodes: " +
visitedWordsLength;
            resultString += "\\nExecution time: " +
elapsedTimeInMilliseconds + " ms";
            resultString += "\\nUsed Memory: " + usedMemory / 8 + "
bytes";
        }

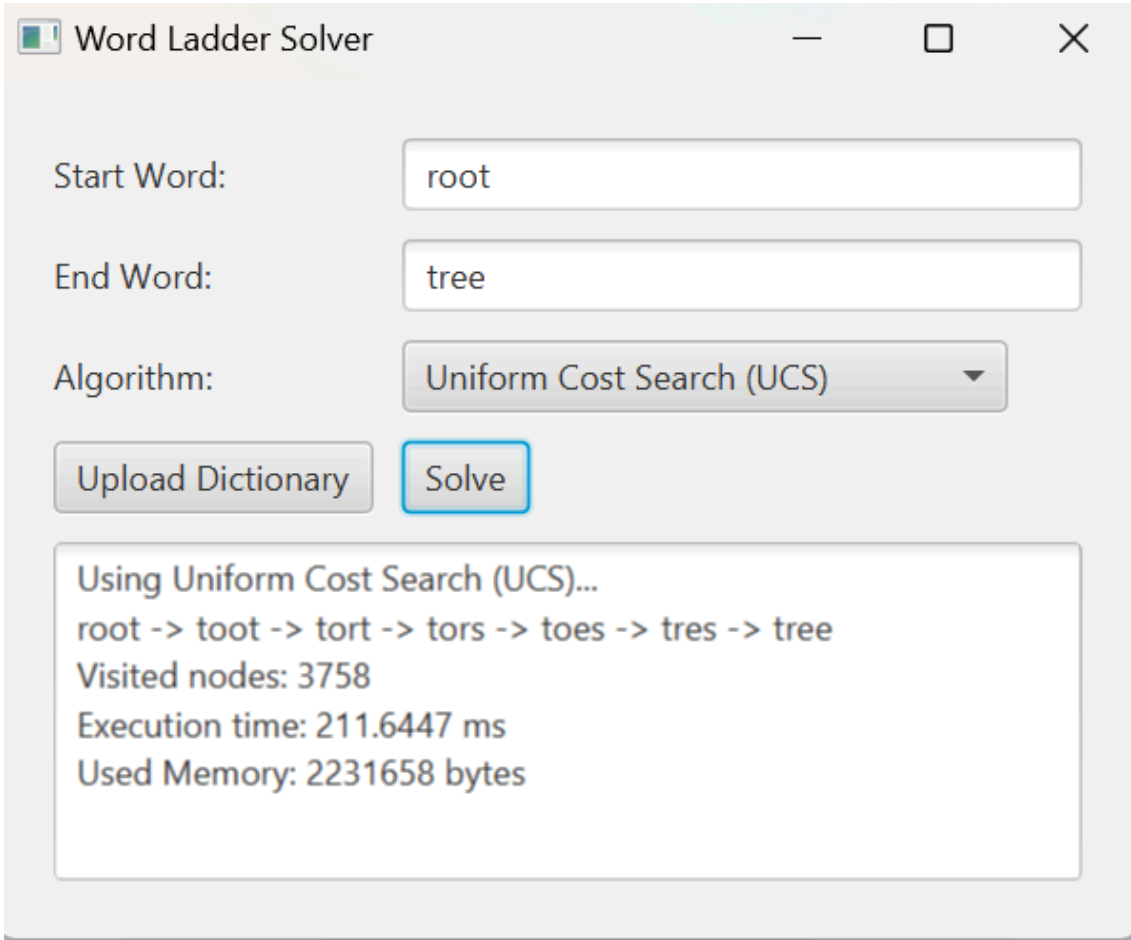
        resultTextArea.setText(resultString);
    }
}

```

WordLadderSolverGUI adalah antarmuka pengguna grafis (GUI) untuk menyelesaikan permasalahan tangga kata menggunakan algoritma pencarian seperti UCS, GBFS, dan A*. Antarmuka pengguna ini memungkinkan pengguna untuk memasukkan kata awal dan akhir, memilih algoritma pencarian, mengunggah kamus kata, dan menyelesaikan masalah tangga kata. Metode `uploadDictionary` digunakan untuk mengunggah kamus kata dari file yang dipilih pengguna. Metode `findWordsWithLength` membaca kamus kata dari file yang diunggah dan mencari kata-kata yang memiliki panjang tertentu. Metode `solve` digunakan untuk memulai proses penyelesaian masalah tangga kata dengan memanggil algoritma pencarian yang dipilih sesuai dengan input pengguna dan menampilkan hasilnya, termasuk jalur yang ditemukan, jumlah kata yang dikunjungi, waktu eksekusi, dan penggunaan memori.

BAB IV
PENGUJIAN

UCS

No	Tangkapan Layar
1	

2

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Using Uniform Cost Search (UCS)...
anger -> aiger -> airer -> firer -> fires -> feres -> heres -> herls -> hello
Visited nodes: 7474
Execution time: 162.7109 ms
Used Memory: 305744 bytes

3

Word Ladder Solver

Start Word: others

End Word: heroes

Algorithm: Uniform Cost Search (UCS)

Upload Dictionary Solve

Using Uniform Cost Search (UCS)...
No path found!

4

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Using Uniform Cost Search (UCS)...

blistering -> blustering -> flustering -> fluttering -> sluttering -> slattering -> slathering

Visited nodes: 35

Execution time: 2.5376 ms

Used Memory: 262144 bytes

5

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Using Uniform Cost Search (UCS)...

quirking -> quirting -> quilting -> quilling -> quelling -> fuelling -> fwelling -> swelling

Visited nodes: 1054

Execution time: 35.2074 ms

Used Memory: 3801344 bytes

6

Word Ladder Solver

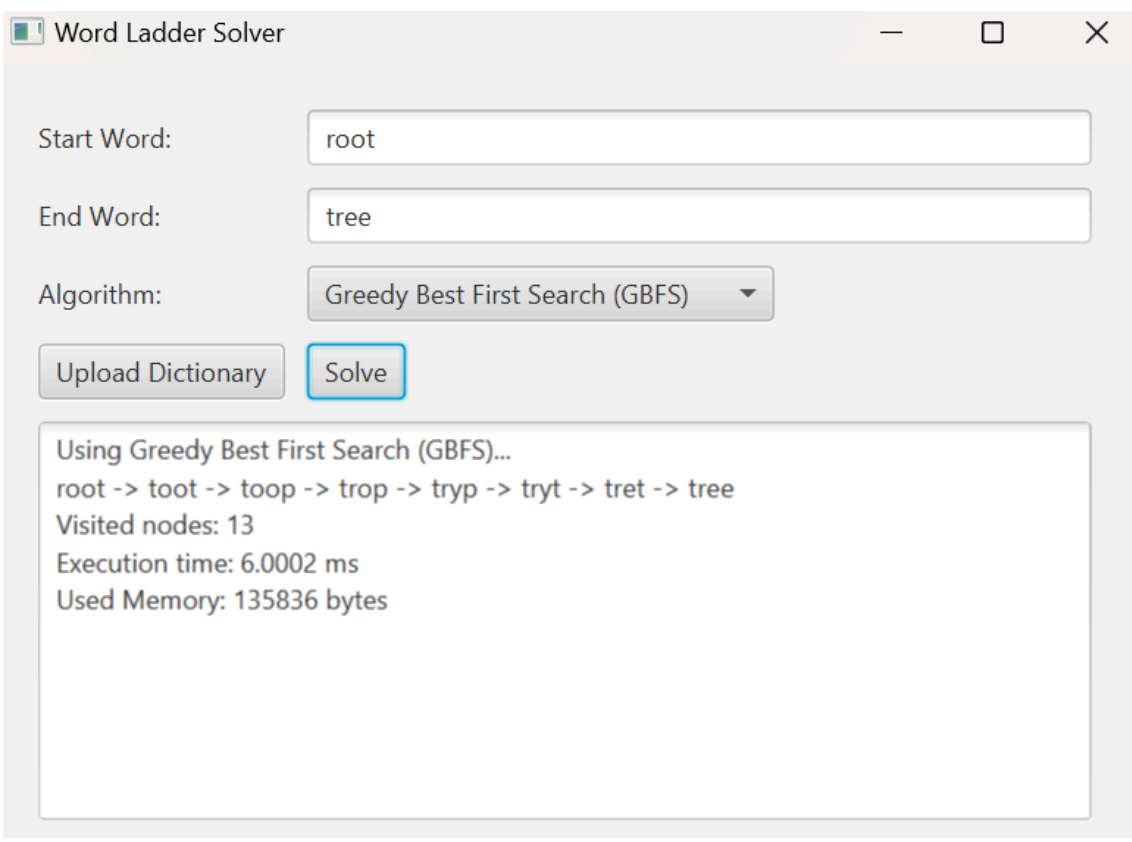
Start Word:

End Word:

Algorithm:

Using Uniform Cost Search (UCS)...
atlases -> anlases -> anlaces -> unlaces -> unlaced -> unraced ->
Visited nodes: 11444
Execution time: 359.3532 ms
Used Memory: 1574586 bytes

GBFS

No	Tangkapan Layar
1	 <p>The screenshot shows a window titled "Word Ladder Solver". It contains the following fields and controls:</p> <ul style="list-style-type: none">Start Word: A text input field containing "root".End Word: A text input field containing "tree".Algorithm: A dropdown menu set to "Greedy Best First Search (GBFS)".Buttons: "Upload Dictionary" and "Solve". The "Solve" button is highlighted with a blue border.Output Area: A text box displaying the solution path and performance metrics:<pre>Using Greedy Best First Search (GBFS)... root -> toot -> toop -> trop -> tryp -> tryt -> tret -> tree Visited nodes: 13 Execution time: 6.0002 ms Used Memory: 135836 bytes</pre>

2

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Using Greedy Best First Search (GBFS)...

anger -> aeger -> neger -> newer -> hewer -> hexer -> hexes -> heres -> herls -> hells

Visited nodes: 18

Execution time: 2.0377 ms

Used Memory: 131072 bytes

3

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Using Greedy Best First Search (GBFS)...
No path found!

4

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Using Greedy Best First Search (GBFS)...

blistering -> glistering -> glittering -> flittering -> fluttering -> sluttering -> scuttering -

Visited nodes: 16

Execution time: 1.5415 ms

Used Memory: 131072 bytes

5

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Using Greedy Best First Search (GBFS)...

quirking -> quirting -> quitting -> quilting -> quilling -> quelling -> duelling -> dwelling

Visited nodes: 261

Execution time: 14.5444 ms

Used Memory: 1572864 bytes

6

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Using Greedy Best First Search (GBFS)...

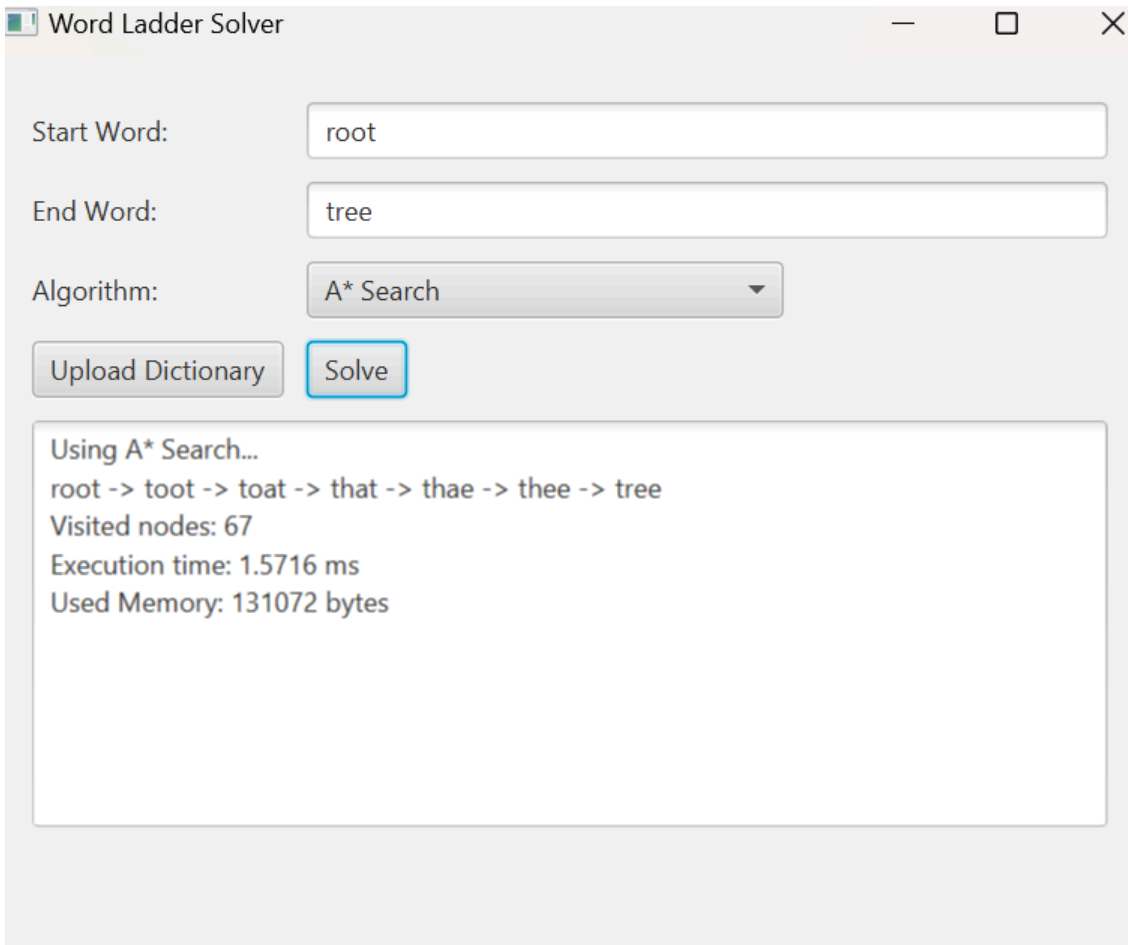
atlases -> anlases -> anlaces -> unlaces -> unlaced -> unlaved -> unlamed

Visited nodes: 1998

Execution time: 114.4247 ms

Used Memory: 2957797 bytes

A*

No	Tangkapan Layar
1	 <p>The screenshot shows a window titled "Word Ladder Solver" with standard Windows window controls (minimize, maximize, close). The interface includes three input fields: "Start Word:" with the value "root", "End Word:" with the value "tree", and "Algorithm:" with a dropdown menu set to "A* Search". Below these fields are two buttons: "Upload Dictionary" and "Solve". The "Solve" button is highlighted with a blue border. Below the buttons is a text area containing the following text: "Using A* Search...", "root -> toot -> toat -> that -> thae -> thee -> tree", "Visited nodes: 67", "Execution time: 1.5716 ms", and "Used Memory: 131072 bytes".</p>

2

Word Ladder Solver

Start Word:

anger

End Word:

hello

Algorithm:

A* Search

Upload Dictionary

Solve

Using A* Search...

anger -> aiger -> airer -> firer -> fires -> feres -> heres -> herls -> hells -> hello

Visited nodes: 253

Execution time: 6.661 ms

Used Memory: 655616 bytes

3

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Using A* Search...
No path found!

4

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Using A* Search...

blistering -> blustering -> flustering -> fluttering -> sluttering -> slattering -> slathering

Visited nodes: 20

Execution time: 1.8287 ms

Used Memory: 131072 bytes

5

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Using A* Search...

quirking -> quirting -> quilting -> quilling -> quelling -> duelling -> dwelling -> swellin

Visited nodes: 746

Execution time: 29.1567 ms

Used Memory: 2752768 bytes

6

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Using A* Search...

atlases -> anlases -> anlases -> unlaces -> unlaced -> unpaced -> unpaged -> uncaged

Visited nodes: 10905

Execution time: 499.0795 ms

Used Memory: 1886772 bytes

Tabel Pengujian

Parameter	Algoritma	Test Case					
		1	2	3	4	5	6
Execution Time (ms)	UCS	212	163	-	3	35	359
	GBFS	6	2	-	2	15	114
	A*	2	7	-	2	29	499
Path Length	UCS	7	10	0	9	21	48
	GBFS	8	11	0	13	28	101
	A*	7	10	0	9	21	48
Visited Nodes	UCS	3758	7474	-	35	1054	11444
	GBFS	13	18	-	16	261	1998

	A*	67	253	-	20	746	10905
Used Memory (bytes)	UCS	2231658	305744	-	262144	3801344	1574586
	GBFS	135836	131072	-	131072	1572864	2957797
	A*	131072	655616	-	131072	2752768	1886772

Hasil Pengujian

a) Execution Time

Didapatkan bahwa pencarian dengan GBFS lebih cepat dengan disusul tidak jauh oleh A*. Keduanya memiliki *execution time* yang jauh lebih cepat dibandingkan pencarian dengan UCS

b) Path Length

Didapatkan bahwa path length yang dihasilkan UCS dan A* sama dan lebih optimal dibandingkan path length yang dihasilkan dari pencarian melalui GBFS.

c) Visited Nodes

Didapatkan bahwa nodes yang dikunjungi GBFS jauh lebih sedikit dibandingkan dengan nodes yang dikunjungi oleh A* dan UCS walau perlu diperhatikan bahwa nodes yang dikunjungi oleh A* lebih sedikit dibandingkan UCS

d) Used Memory

Didapatkan bahwa memory yang digunakan oleh GBFS cenderung lebih “hemat” dibandingkan yang digunakan oleh A* dan UCS walau perlu diperhatikan bahwa terdapat inkonsistensi terhadap data ketika dijalankan berulang kali. Dengan demikian, lebih disarankan untuk penilaian terhadap *memory usage* bisa menggunakan *visited nodes* dengan hasil yang lebih konsisten.

Bonus

Bagian bonus dari tugas kecil ini adalah pengembangan antarmuka pengguna (GUI) yang dibuat menggunakan JavaFX yang di-build menggunakan Maven. Antarmuka ini dibuat untuk menerima input dari pengguna berupa kata awal (*start word*) dan kata akhir (*goal word*). Antarmuka ini juga dilengkapi dengan opsi pemilihan yang mewakili berbagai algoritma pencarian yang berbeda serta opsi untuk mengunggah *dictionary* yang akan digunakan. Input kemudian akan diproses lalu hasil dari pencarian akan ditampilkan di dalam antarmuka pengguna.

BAB V

KESIMPULAN DAN SARAN

Kesimpulan

Berdasarkan hasil pengujian yang telah dilakukan, algoritma A* dan UCS secara konsisten menunjukkan kemampuan untuk mencapai solusi yang efisien dalam hal panjang jalur yang dihasilkan. Kedua algoritma tersebut menghasilkan jumlah kata pada jalur yang paling sedikit dalam hampir semua kasus tes, memastikan bahwa jalur yang diambil adalah yang terpendek. A* menonjol dengan pendekatan yang lebih seimbang, memberikan efisiensi dalam jumlah kata yang diperiksa dan waktu eksekusi yang relatif cepat. Di sisi lain, UCS, meskipun efektif dalam mencapai jalur yang optimal, cenderung menggunakan lebih banyak memori dan waktu karena pendekatannya yang lebih dalam dan eksploratif. GBFS, yang menunjukkan waktu eksekusi tercepat, berisiko menghasilkan solusi yang kurang optimal karena jumlah kata pada jalur yang dihasilkan cenderung lebih banyak dan melakukan validasi yang lebih sedikit, menunjukkan efisiensi tetapi dengan ketepatan yang berkurang. Meskipun GBFS cepat, namun tidak selalu ideal ketika ketepatan solusi menjadi krusial, terutama dalam kasus yang membutuhkan solusi yang paling efisien atau optimal. Sebaliknya, A* menawarkan keseimbangan yang sangat baik antara kecepatan dan akurasi, menjadikannya pilihan yang lebih solid dalam berbagai kasus uji. Penggunaan memori oleh A* juga cenderung lebih efisien dibandingkan dengan UCS, membuatnya lebih cocok untuk aplikasi dengan sumber daya yang terbatas.

Saran

Peneliti menyarankan untuk mengeksplorasi pendekatan-pendekatan lain untuk menyelesaikan Word Ladder untuk bisa lebih memahami berbagai pendekatan untuk memecahkan sebuah masalah dan lebih memahami algoritma *search*. Peneliti juga menyarankan untuk mengembangkan lebih lagi GUI yang ditampilkan agar memiliki tampilan yang lebih menarik serta fitur yang lebih *advanced*. Terakhir, perlu ada penelitian juga terkait pengembangan Word Ladder Solver untuk bahasa yang lebih banyak selain bahasa Inggris.

DAFTAR PUSTAKA

<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>

<https://www3.cs.stonybrook.edu/~pfodor/courses/CSE114/L14-JavaFXBasics.pdf>

LAMPIRAN

Kode program dapat diakses pada *repository* Github berikut

<https://github.com/HenryofSkalitz1202/Tucil3Stima.git>

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