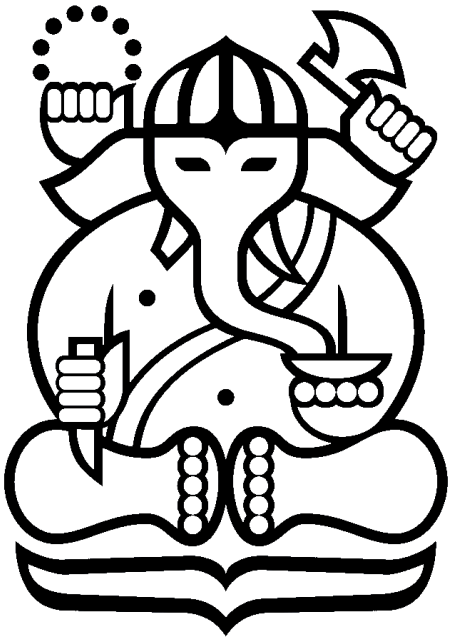
IF2211 Strategi Algoritma

Penyelesaian Permainan Word Ladder Menggunakan Algoritma UCS, Greedy Best First Search, dan A*

Laporan Tugas Kecil 3

Disusun untuk memenuhi tugas mata kuliah Strategi Algoritma
pada Semester 2 (dua) Tahun Akademik 2023/2024



Oleh
Valentino Chryslie Triadi (13522164)

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

DAFTAR ISI

BAB 1
DESKRIPSI TUGAS

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.

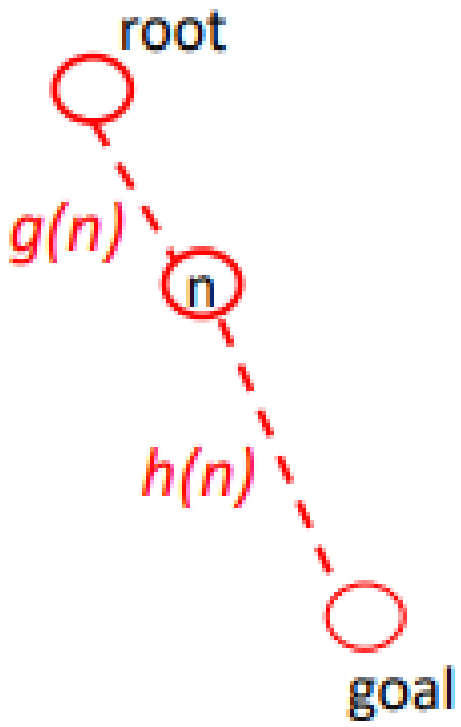
Gambar 1 *Ilustrasi dan Peraturan Permainan Word Ladder*

Permainannya cukup sederhana bukan? Jika belum paham dengan peraturan permainannya, cobalah untuk memainkan permainannya pada link sumber di atas. Jika sudah paham dengan permainannya, sekarang adalah waktunya kalian untuk membuat sebuah solver permainan tersebut dengan harapan kita dapat menemukan solusi paling optimal untuk menyelesaikan permainan Word Ladder ini.

BAB 2
LANDASAN TEORI

Penentuan rute atau *path planning* adalah salah satu contoh penerapan algoritma yang sangat dibutuhkan di kehidupan nyata. Salah satu contoh nyatanya adalah penentuan rute terpendek oleh aplikasi *google maps*. Di balik penentuan rute tersebut, ada beberapa algoritma yang bisa digunakan sesuai dengan kelebihan dan kekurangan masing-masing. Salah satu contohnya adalah algoritma *A**, algoritma *Uniform Cost Search* (UCS), dan algoritma *Greedy Best-First Search*.

Dalam menentukan rute, terdapat dua metode pencarian, yakni *Uninformed Search* dan *Informed Search*. *Uninformed Search* atau bisa disebut juga sebagai pencarian buta merupakan pencarian yang dilakukan tanpa adanya informasi tambahan terkait kondisi di luar status yang saat ini sedang dihadapi. *Uninformed Search* melakukan pencarian dengan mengandalkan informasi status yang terdapat pada setiap simpul dalam graf. Sedangkan *Informed Search* atau biasa disebut juga dengan *Heuristic Search* adalah pencarian yang menggunakan informasi tambahan yang spesifik terkait permasalahan yang dihadapi di samping dari definisi masalahnya itu sendiri. *Informed Search* memiliki keunggulan dalam hal optimalisasi biaya yang paling efisien. Dalam melakukan *Uninformed Search* ataupun *Informed Search* diketahui ada beberapa komponen fungsi yang digunakan, yaitu fungsi evaluasi atau bisa dinyatakan dalam notasi f(n). Fungsi evaluasi dapat dibedakan lagi menjadi beberapa konsep, seperti fungsi evaluasi *start-to-node* (g(n)) dan fungsi evaluasi node-to-end (h(n)). Fungsi evaluasi dapat berbeda-beda untuk tiap algoritma. Hal tersebut akan dibahas lebih lanjut dalam pembahasan algoritma.



Gambar 2 *Ilustrasi fungsi evaluasi heuristik*

Algoritma *Uniform Cost Search* atau UCS adalah salah satu bagian dari *Uninformed Search*. Algoritma ini memanfaatkan fungsi *start-to-node* sebagai fungsi evaluasi nya. Fungsi *start-to-node* adalah fungsi yang menghitung bobot sebuah rute dengan menghitung semua bobot atau *cost* dari titik awal hingga titik saat ini. Pada gambar 2.1, jika n merupakan posisi saat ini maka fungsi *start-to-node* dapat digambarkan oleh fungsi g(n). Algoritma ini memanfaatkan struktur data *priority queue* dengan pembobotan yang dikendalikan oleh fungsi *start-to-node*. Pada algoritma ini dikenal dua himpunan simpul, yaitu himpunan simpul hidup (himpunan simpul yang akan dilakukan pencarian) dan himpunan simpul mati (himpunan simpul yang sudah pernah dilakukan pencarian). Setiap kali algoritma ini melakukan pencarian, maka simpul yang ditemukan akan dimasukan kedalam himpunan simpul hidup dengan urutan sesuai dengan bobot terkecil dan memasukan simpul yang dicari kedalam himpunan simpul mati. Namun, apabila simpul yang ditemukan sudah termasuk ke dalam himpunan simpul mati, maka simpul yang ditemukan tersebut tidak akan dimasukan lagi kedalam himpunan simpul hidup.

Algoritma *Greedy Best-First Search* adalah salah satu bagian dari *Informed Search*. Algoritma ini memanfaatkan fungsi *node-to-end* sebagai fungsi evaluasi nya. Fungsi *node-to-end* adalah fungsi yang melakukan pembobotan berdasarkan informasi luar yang berhubungan dengan tujuan/permasalahan. Pada gambar 2.1, jika n merupakan posisi saat ini maka fungsi *node-to-end* dapat digambarkan oleh fungsi h(n). Pada algoritma ini, setiap kali dilakukan pencarian maka akan dilakukan evaluasi terhadap simpul yang ditemukannya. Simpul berikutnya yang dipilih adalah simpul dengan bobot evaluasi terbaik pertama. Algoritma ini memungkinkan untuk tidak ditemukan solusi akibat dari penentuan simpul yang selalu mengambil simpul terbaik pertama.

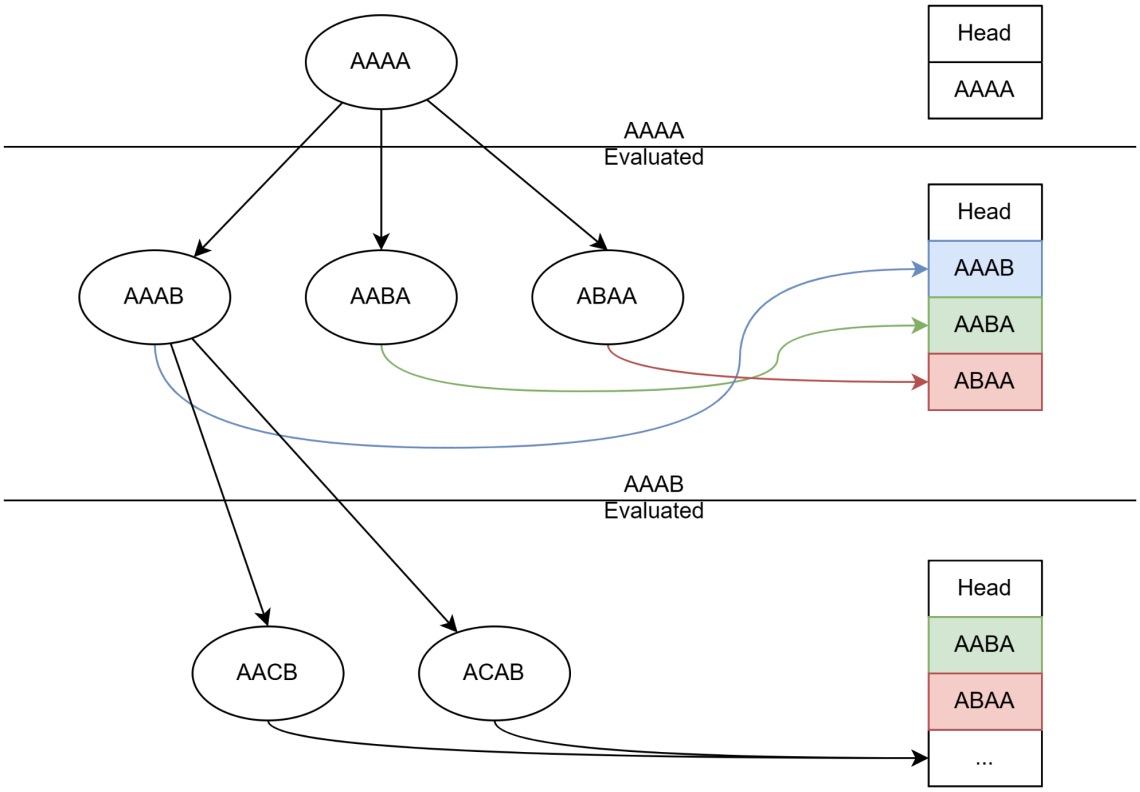
Algoritma *A** atau *A-Star* adalah salah satu bentuk dari *Informed Search*. Algoritma ini memanfaatkan gabungan dari fungsi *start-to-node* dan fungsi *node-to-end* sebagai fungsi evaluasinya. Cara kerja dari algoritma ini kurang lebih menyerupai dengan algoritma *Greedy Best-First Search* namun berbeda dalam pembobotan simpul dan juga pemilihan simpul terbaik. Jika ditemukan dua atau lebih simpul dengan bobot evaluasi yang sama, maka algoritma ini akan mengevaluasi kedua simpul dan mencari simpul anakan dari simpul kembar tersebut dengan bobot evaluasi yang terbaik.

BAB 3
IMPLEMENTASI DAN PENGUJIAN

3.1 Implementasi

3.1.1 Implementasi Algoritma UCS

Dalam mengimplementasikan algoritma UCS, diperlukan bantuan struktur data *queue* sebagai tempat penyimpanan himpunan simpul hidup. Alasan mengapa struktur data yang dipilih adalah *queue* bukan *priority queue* adalah karena fungsi evaluasi (f(n) = g(n)) dalam permasalahan ini yang hanya bergantung pada jumlah perbedaan huruf pada posisi yang sama dan peraturan permainan yang hanya memperbolehkan setiap simpul anak memiliki perbedaan satu huruf saja dengan simpul orang tua nya. Hal ini membuat bobot evaluasi dalam permasalahan ini hanya bergantung pada panjang rute dikurang satu (n-1) dan tidak diperlukan ukuran prioritas melainkan hanya diperlukan urutan dalam *queue*. Algoritma ini akan melakukan pengecekan pada rute yang terletak pada urutan terdepan *queue*. Setiap kali ditemukan simpul anak, maka semua rute baru (rute dengan simpul anak sebagai simpul akhir) akan dimasukan ke dalam *queue* jika simpul anak tersebut belum pernah dilakukan pengecekan atau masih belum terdaftar dalam himpunan simpul mati. Ada beberapa kondisi pemberhenti algoritma ini, yaitu ketika rute yang diperiksa sudah memiliki simpul akhir yang sesuai dengan tujuan (solusi ditemukan) dan kondisi saat *queue* sudah kosong.



Gambar 3.1.1 *Ilustrasi Penentuan Rute Terbaik pada Algoritma UCS*

Berikut adalah penjelasan langkah demi langkah implementasi algoritma UCS dalam bentuk kode:

Melakukan pengambilan rute pertama pada *queue*

```
List<String> currentPath = path.poll();
String tempWord = currentPath.get(currentPath.size() - 1);
```

Melakukan pengecekan kondisi pemberhenti (solusi ditemukan) saat simpul akhir dari rute adalah kata tujuan

```
if (tempWord.equals(endWord)) {
    ...
    // return Result
```

Menambahkan simpul kedalam himpunan simpul mati

```
checkedWord.add(tempWord);
```

Mencari simpul anak

```
List<String> linkingWordList = WordChecker.wordMap.get(tempWord);
```

Mengevaluasi semua simpul anak sesuai dengan *constraint* simpul anak yang sudah terdaftar dalam himpunan simpul mati tidak boleh dimasukan ke himpunan simpul hidup

```
for (String word : linkingWordList) {
    if (!checkedWord.contains(word)) {
        List<String> newPath = new ArrayList<>(currentPath);
        newPath.add(word);
        path.add(newPath);
    }
}
```

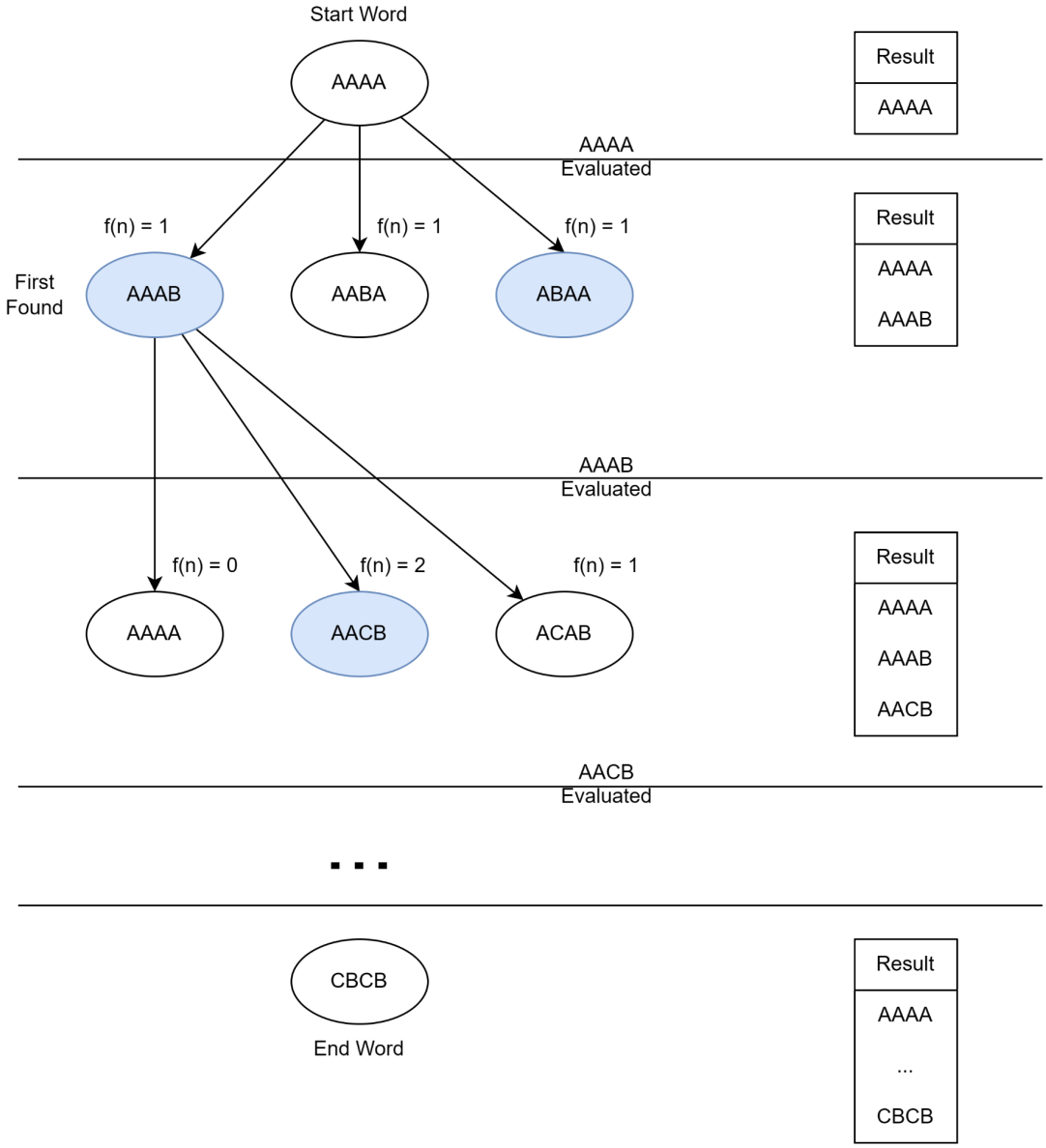
Menjalankan kondisi pemberhenti terakhir (tidak ditemukan solusi) saat queue sudah kosong

```
while (!path.isEmpty()) {
   ...
}

// No Solution
...
```

3.1.2 Implementasi Algoritma Greedy Best-First Search

Dalam mengimplementasikan algoritma *Greedy Best-First Search*, diperlukan bantuan fungsi evaluasi (f(n) = h(n)) yang akan mengembalikan bobot tiap simpul berdasarkan kemiripannya dengan simpul tujuan. Kemiripan yang dimaksud adalah banyaknya jumlah huruf dengan posisi yang sama antara simpul saat ini dan simpul tujuan. Simpul yang akan diambil sebagai simpul lanjutan dan yang akan dimasukan ke rute hasil adalah simpul yang memiliki kemiripan paling besar dan belum terdaftar dalam rute hasil. Apabila ditemukan lebih dari simpul anak yang memiliki bobot evaluasi sama, maka akan diambil simpul anak pertama dengan bobot terbaik. Pada Algoritma ini ada tiga kondisi pemberhenti, yaitu saat tidak ditemukan simpul anak (solusi tidak ditemukan), saat hasil evaluasi seluruh simpul anak tidak menemukan simpul terbaik (solusi tidak ditemukan), dan saat simpul saat ini sama dengan simpul tujuan (solusi ditemukan).



Gambar 3.1.2 *Ilustrasi Penentuan Rute Terbaik pada Algoritma Greedy Best-First Search*

Berikut adalah penjelasan langkah demi langkah implementasi algoritma *Greedy Best-First Search* dalam bentuk kode:

Menambahkan simpul saat ini ke result

```
this.result.add(tempWord);
```

Melakukan pencarian simpul anak

```
List<String> linkingWordList = WordChecker.wordMap.get(tempWord);
nodeChecked += linkingWordList.size();
```

Melakukan pengecekan kondisi pemberhenti (tidak ditemukan solusi) saat tidak ditemukan simpul anak

```
if (linkingWordList.isEmpty()) {
    ...
}
```

Melakukan evaluasi seluruh simpul anak yang ditemukan dan menjadikan simpul saat ini dengan hasil evaluasi terbaik dari seluruh simpul anak

```
tempWord = Utils.getFirstMostSimiliar(linkingWordList, endWord);
```

Melakukan pengecekan kondisi pemberhenti (tidak ditemukan solusi) saat evaluasi seluruh simpul anak tidak berhasil menemukan simpul terbaik

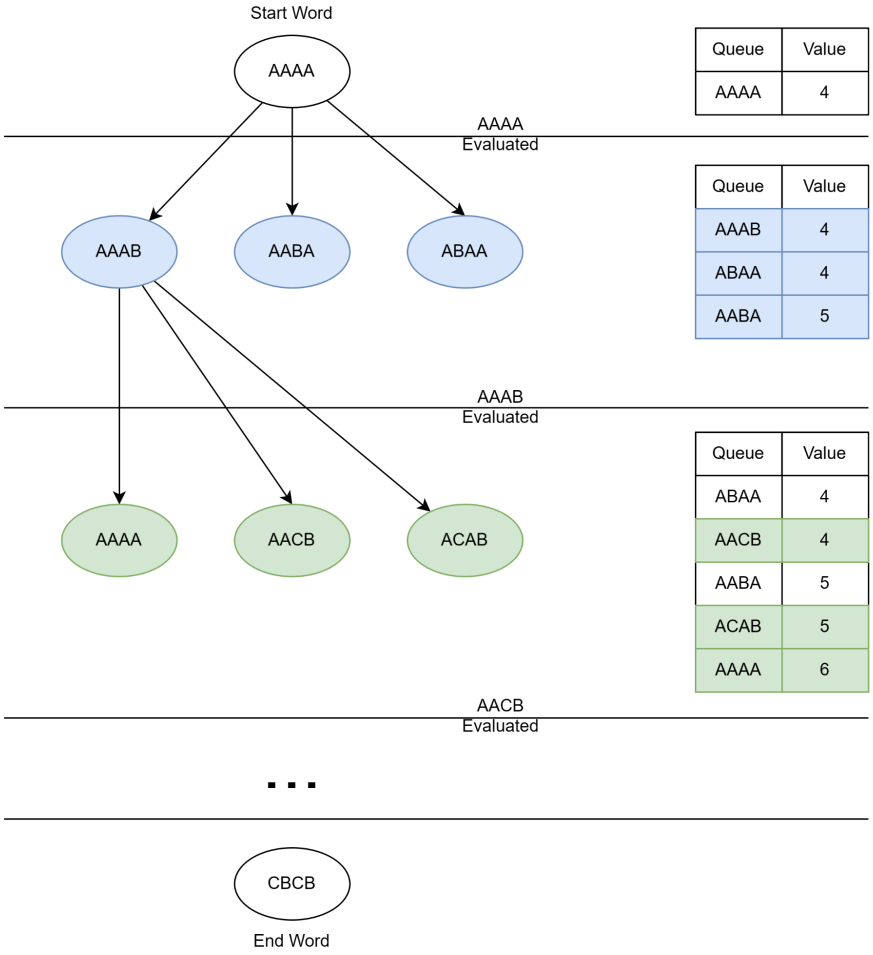
```
if (tempWord.equals("")) {
    ...
}
```

Menjalankan kondisi pemberhenti terakhir (ditemukan solusi) saat simpul anak yang sudah dievaluasi sama dengan simpul tujuan

```
while (!tempWord.equals(endWord)) {
    ...
}
// Solution Found
...
```

3.1.3 Implementasi Algoritma A*

Dalam mengimplementasikan Algoritma *A**, diperlukan bantuan struktur data *priority-queue* dengan pembobotan prioritas yang merupakan hasil dari fungsi evaluasi. Fungsi evaluasi (f(n)) pada algoritma kali ini menerapkan fungsi *start-to-node* (g(n)) dan fungsi *node-to-end* (h(n)). Fungsi *start-to-node* menghitung jumlah perbedaan huruf pada posisi yang sama dari simpul satu dengan simpul setelahnya dalam rute, hal tersebut dapat diefisienkan dengan melakukan perhitungan panjang rute dikurang satu (n-1) karena perbedaan huruf pada posisi yang sama antara simpul satu dengan simpul setelahnya dipastikan hanya satu. Fungsi *node-to-end* akan menghitung perbedaan huruf pada posisi yang sama antara simpul saat ini dengan simpul tujuan. Fungsi evaluasi adalah penjumlahan dari fungsi *start-to-node* dan fungsi *node-to-end* (f(n) = g(n) + h(n)). Konsep implementasi dari Algoritma A* kurang lebih menyerupai konsep implementasi algoritma UCS, yaitu melakukan pengecekan pada rute dengan urutan terdepan di *priority-queue* dan memasukan rute baru (rute lama ditambah simpul anak) kedalam *priority-queue*. Namun yang membedakan adalah saat memasukan rute kedalam *priority-queue*, dimana elemen di dalam *priority-queue* harus terurut berdasarkan bobot evaluasi dari yang terendah ke yang tertinggi sehingga pemasukan rute kedalam *priority-queue* tidak selalu di urutan paling belakang namun berdasarkan bobot dari rute tersebut. Ada tiga kondisi pemberhenti dalam implementasi algoritma ini, yaitu saat simpul akhir sama dengan simpul tujuan (solusi ditemukan), saat tidak ditemukan simpul anak (solusi tidak ditemukan), dan saat *priority-queue* sudah kosong (solusi tidak ditemukan).



Gambar 3.1.3 *Ilustrasi Penentuan Rute Terbaik pada Algoritma A**

Berikut adalah penjelasan langkah demi langkah implementasi algoritma *A** dalam bentuk kode:

Mengambil rute pertama dalam priority-queue dan mengambil simpul akhir dari rute

```
AStarNode currentPath = path.poll();
String tempWord = currentPath.getLastWord();
```

Melakukan pengecekan kondisi pemberhenti (ditemukan solusi) saat simpul akhir sama dengan simpul tujuan

```
if (tempWord.equals(endWord)) {
    ...
}
```

Melakukan pencarian simpul anak

```
List<String> linkingWordList = WordChecker.wordMap.get(tempWord);
```

Memasukan simpul saat ini ke himpunan simpul mati

```
checkedWord.add(tempWord);
```

Melakukan pengecekan kondisi pemberhenti (tidak ditemukan solusi) saat tidak ditemukan simpul anak

```
if (linkingWordList.isEmpty()) {
    ...
}
```

Melakukan evaluasi seluruh simpul anak yang belum terdaftar di himpunan simpul matidan menambahkan rute baru (rute lama ditambah simpul anak) ke priority-queue

```
for (String word : linkingWordList) {
    if (!checkedWord.contains(word)) {
       List<String> newPath = new ArrayList<>(currentPath.getPath());
       newPath.add(word);
       path.add(new AStarNode(newPath, getCost(newPath)));
    }
}
```

Kondisi pemberhenti (tidak ditemukan solusi) saat priority-queue sudah kosong

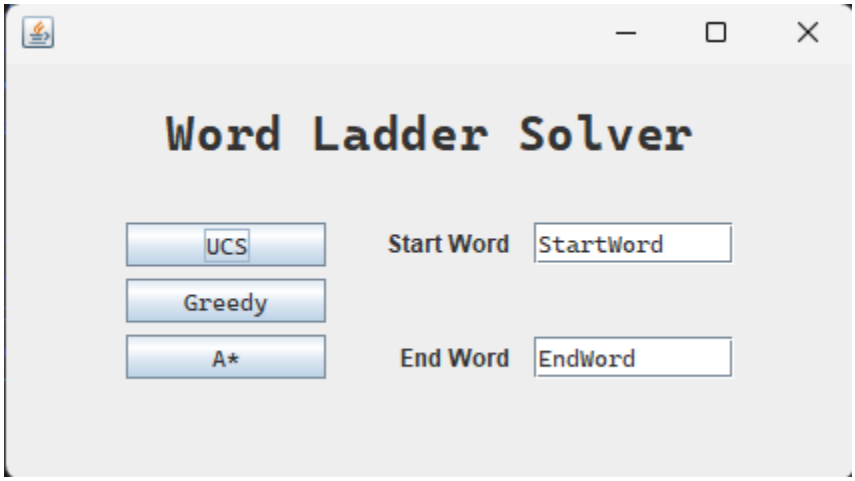
```
while (!path.isEmpty()) {
    ...
}

// No Solution
...
```

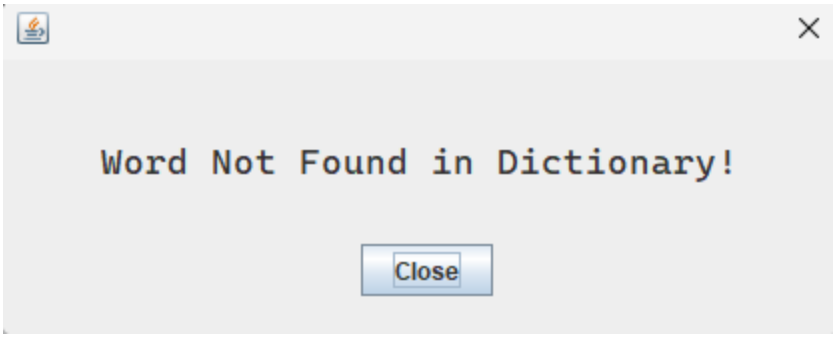
3.1.4 Implementasi Bonus GUI

Dalam mengimplementasikan bonus GUI, diperlukan beberapa perkakas seperti Java Swing dan Netbeans (IDE). Seluruh implementasi GUI dapat dilihat pada file *MainGUI.java*, *ResultUI.java*, dan *WordNotFound.java*. File *MainGUI.java* berisi dua *textarea* untuk pengguna memasukan kata awal dan kata akhir dan juga tiga tombol yang jika diklik akan mulai menjalankan algoritma UCS, *Greedy Best First Search*, atau *A**. File *WordNotFound.java* berisi dialog terkait validasi kata dalam kamus. File *ResultUI.java* berisi text mengenai informasi hasil berupa waktu eksekusi, jumlah simpul yang dievaluasi, dan jumlah memori yang digunakan. File tersebut juga akan menampilkan hasil pencarian dalam bentuk tabel.

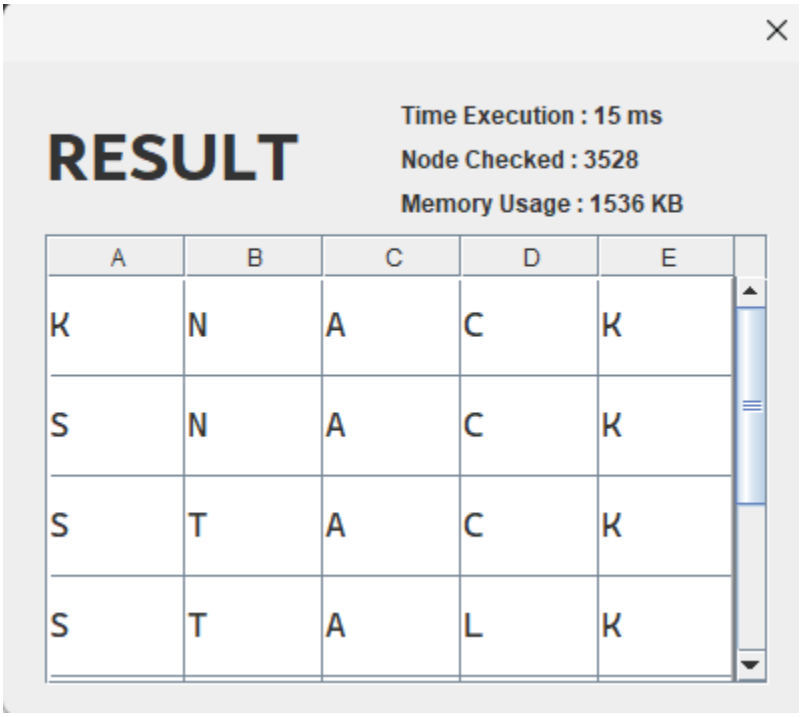
Berikut adalah tangkapan layar dari GUI:



Gambar 3.1.4.1 *Tangkapan Layar MainGUI*



Gambar 3.1.4.2 *Tangkapan Layar WordNotFoundUI*



Gambar 3.1.4.3 *Tangkapan Layar ResultUI*

3.2 Penjelasan Class dan Method

Tabel 3.2 *Tabel Penjelasan Class dan Method*

No	Class/Method	Keterangan
1	Algortihm (Interface)	Interface yang berisi abstract method yang diperlukan oleh UCS, Greedy, AStar
2	Result	Class untuk menyimpan hasil pencarian sebelum ditampilkan ke GUI
	String[][] getResult()	Mengembalikan result dalam bentuk matriks of character
	int getNodeChecked()	Mengembalikan jumlah simpul yang dicek
	String getTime()	Mengembalikan waktu eksekusi algoritma
	String getMemory()	Mengembalikan memory yang dipakai saat eksekusi algoritma
3	UCS	Class untuk mengimplementasikan algoritma UCS
	Result Solve()	Menjalankan algoritma pencarian UCS dan mengembalikan result
	void printResult()	Menampilkan hasil (path) ke layar
	String[][] getResult()	Memproses dan mengembalikan result (list of string) menjadi dalam bentuk matriks of character
4	Greedy	Class untuk mengimplementasikan algoritma Greedy
	Result Solve()	Menjalankan algoritma

		pencarian Greedy dan mengembalikan result
	void printResult()	Menampilkan hasil (path) ke layar
	String[][] getResult()	Memproses dan mengembalikan result (list of string) menjadi dalam bentuk matriks of character
5	AStarNode	Class untuk menyimpan informasi simpul
	List<String> getPath()	Mengembalikan path saat ini dalam bentuk list of string
	String getLastWord()	Mengembalikan huruf terakhir dari path saat ini dalam bentuk string
	int getCost()	Mengembalikan informasi bobot simpul
6	AStarNodeComparator	Class untuk melakukan perbandingan antar simpul
	int compare(AStarNode node1, AStarNode node2)	Mengembalikan nilai integer -1 jika bobot node1 lebih kecil dari bobot node2, 1 jika bobot node1 lebih besar dari bobot node2, dan 0 jika bobot node1 dan bobot node2 sama
7	AStar	Class untuk mengimplementasikan algoritma Greedy
	Result Solve()	Menjalankan algoritma pencarian A* dan mengembalikan result
	void printResult()	Menampilkan hasil (path) ke layar
	String[][] getResult()	Memproses dan mengembalikan result (list

		of string) menjadi dalam bentuk matriks of character
8	Utils	Class untuk menghandle fungsional yang dibutuhkan oleh algoritma
	int getSimiliarity(String word1, String word2)	Mengembalikan kemiripan word1 dengan word2
	String getFirstMostSimiliar(List<String> wordList, String word, List<String> bannedWord)	Mengembalikan string pertama di wordList yang paling mirip dengan word namun belum terdaftar di bannedWord
	int getDifferentCharacter(String word1, String word2)	Mengembalikan jumlah perbedaan huruf pada posisi character yang sama
9	WordChecker	Class untuk menghandle fungsional pencarian kata dalam dictionary
	void loadLength(int lengthWord)	Untuk melakukan load dictionary dengan panjang kata tertentu
	void load()	Untuk melakukan load dictionary untuk semua kata
	boolean isWordExist(String word)	Untuk melakukan pengecekan apakah word terdaftar di dictionary atau tidak
10	MainGUI	Class untuk GUI main menu, berisi 2 text field untuk input startword dan endword, dan 3 tombol untuk pilihan algoritma
	void initComponents()	Menginisiasi GUI main menu
	void jButton2ActionPerformed(...)	Action untuk menjalankan pencarian dengan algoritma UCS saat tombol jButton2

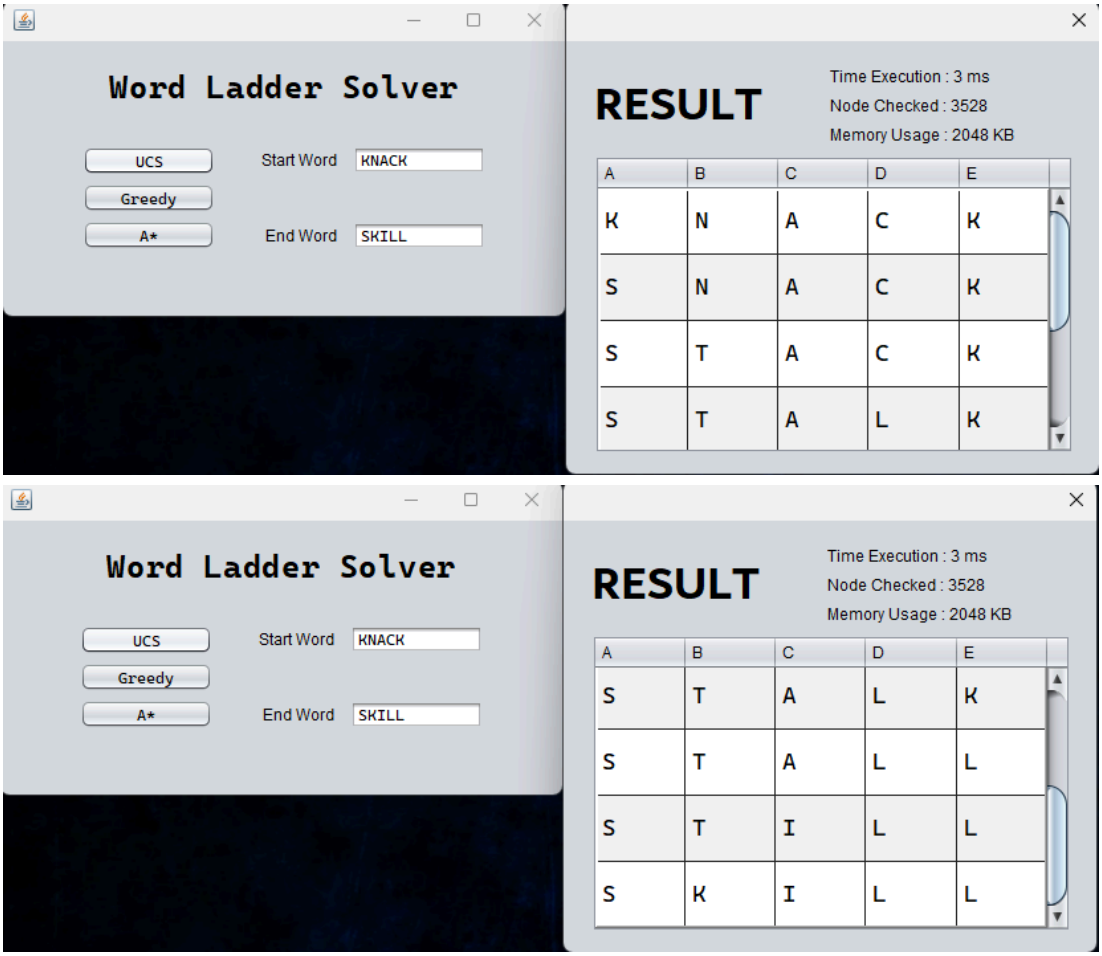
		dipencet
	void jButton3ActionPerformed(...)	Action untuk menjalankan pencarian dengan algoritma Greedy saat tombol jButton3 dipencet
	void jButton4ActionPerformed(...)	Action untuk menjalankan pencarian dengan algoritma A* saat tombol jButton4 dipencet
11	WordNotFoundUI	Class untuk GUI alert kata tidak ditemukan dalam dictionary
	Void jButton1ActionPerformed(...)	Untuk menutup window apabila jButton1 dipencet
	Void initComponents()	Menginisiasi GUI alert
12	ResultUI	Class untuk GUI menampilkan hasil
	Void setResult()	Untuk menampilkan hasil pada window
	Void initComponents()	Menginisiasi GUI hasil

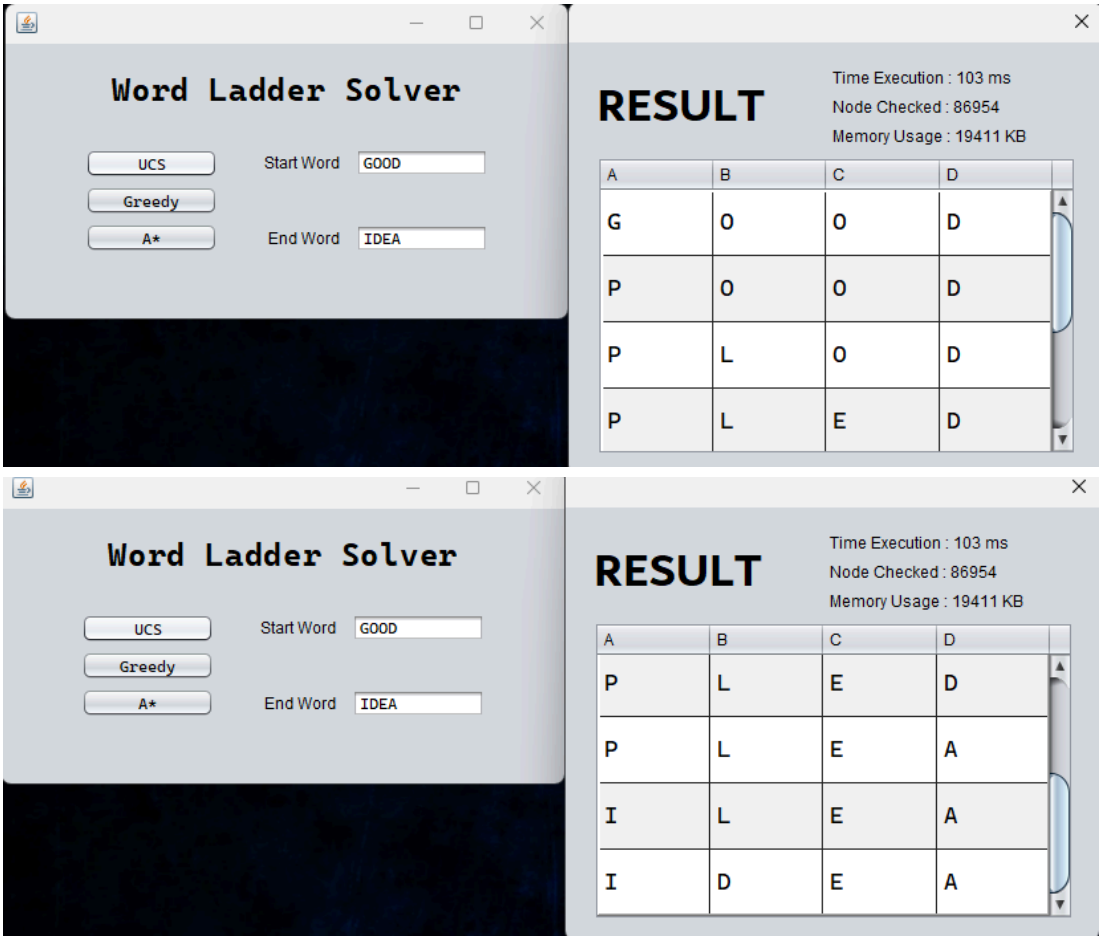
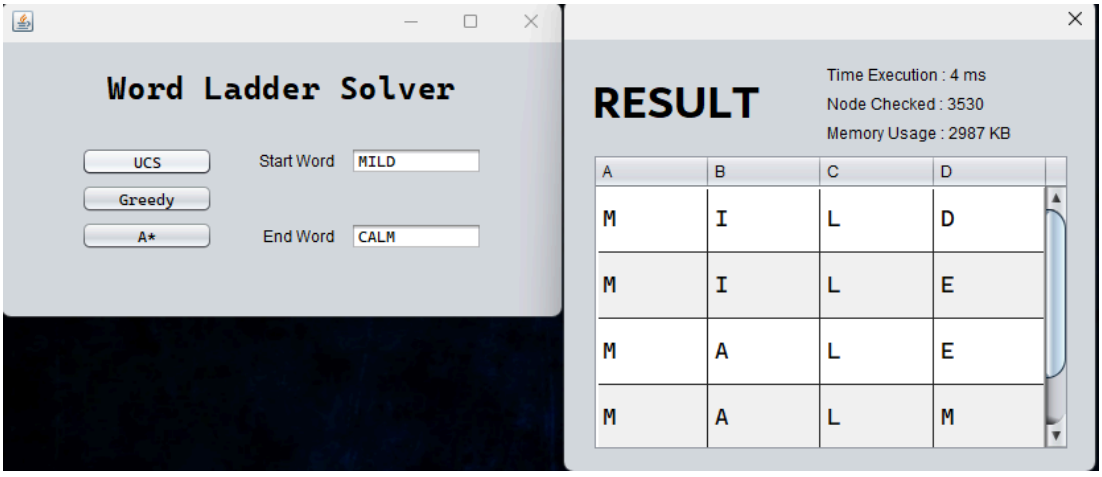
Keterangan: warna putih untuk class dan warna abu untuk method didalamnya

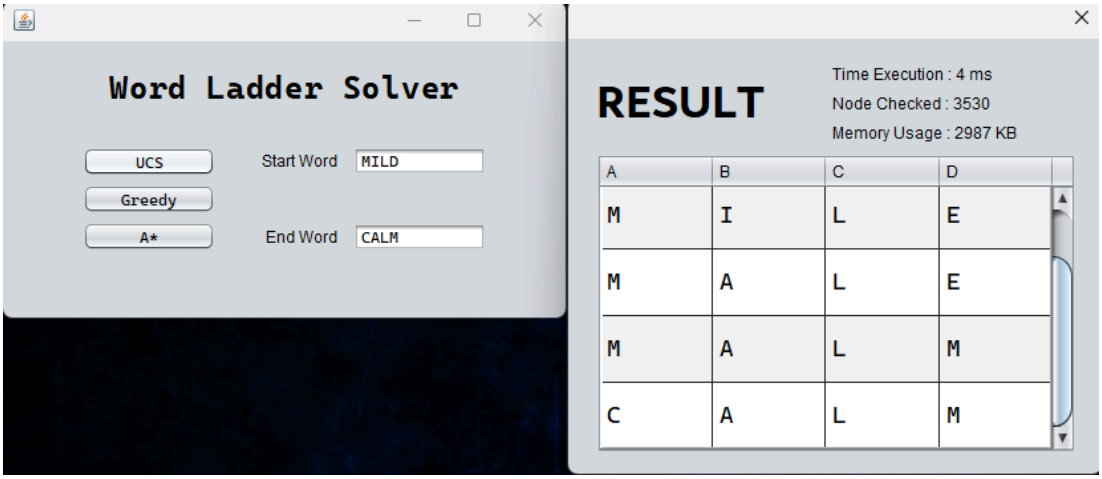
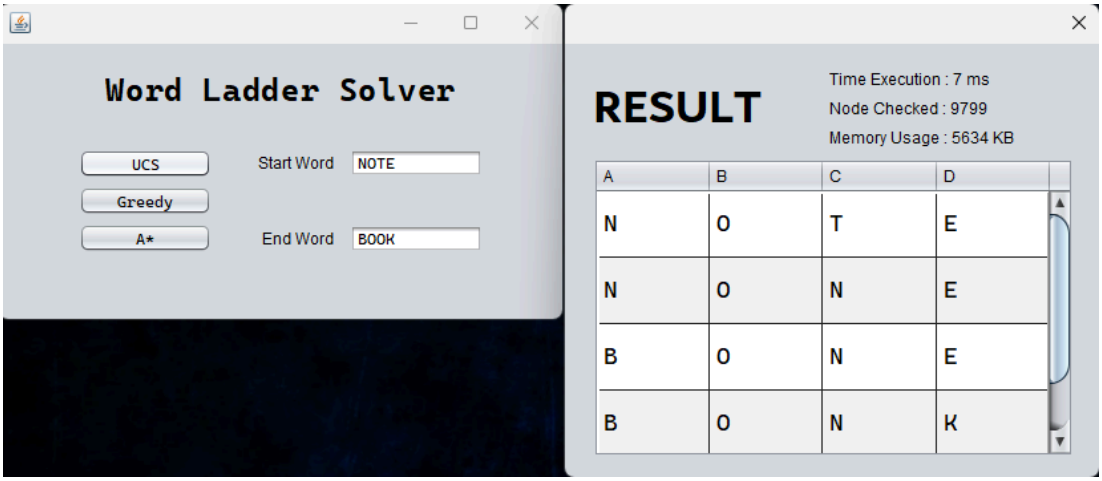
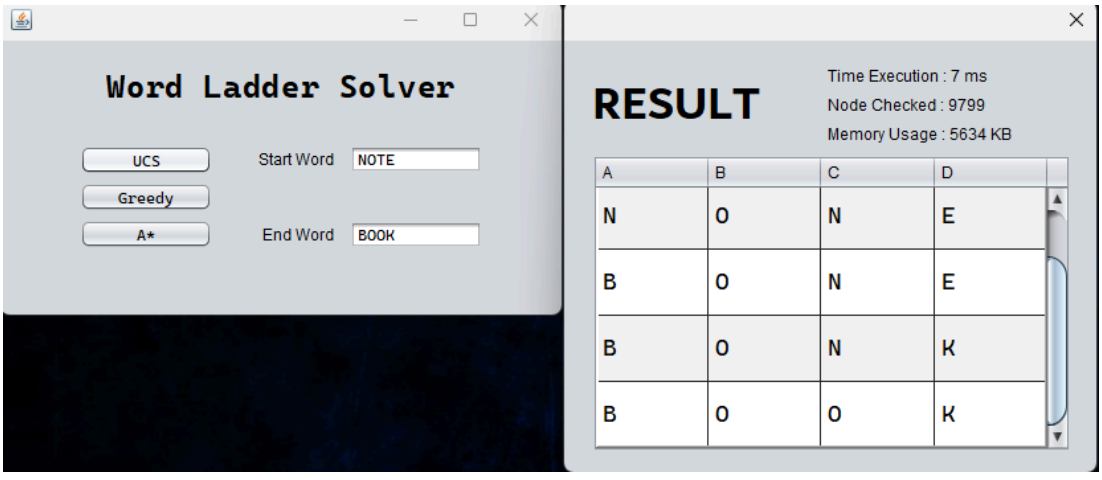
3.3 Pengujian

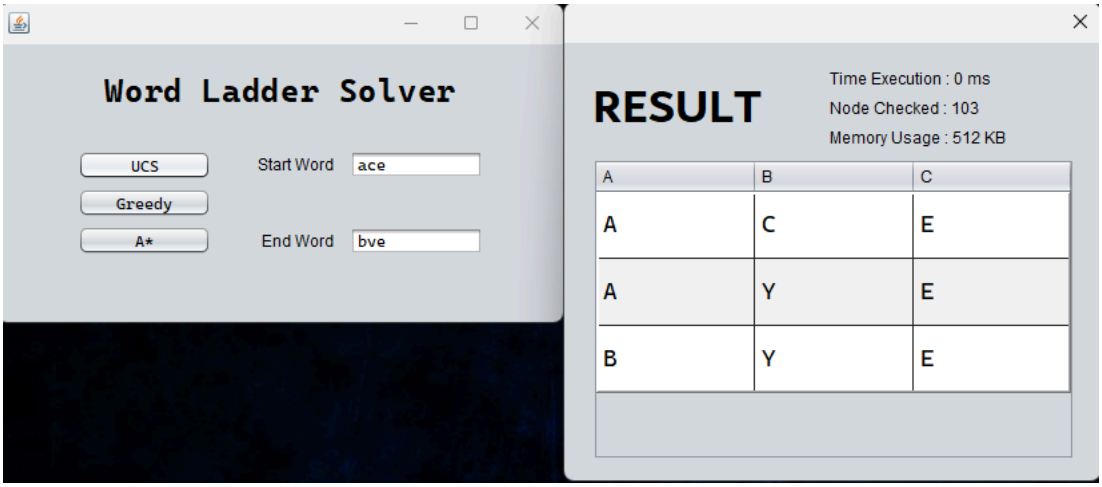
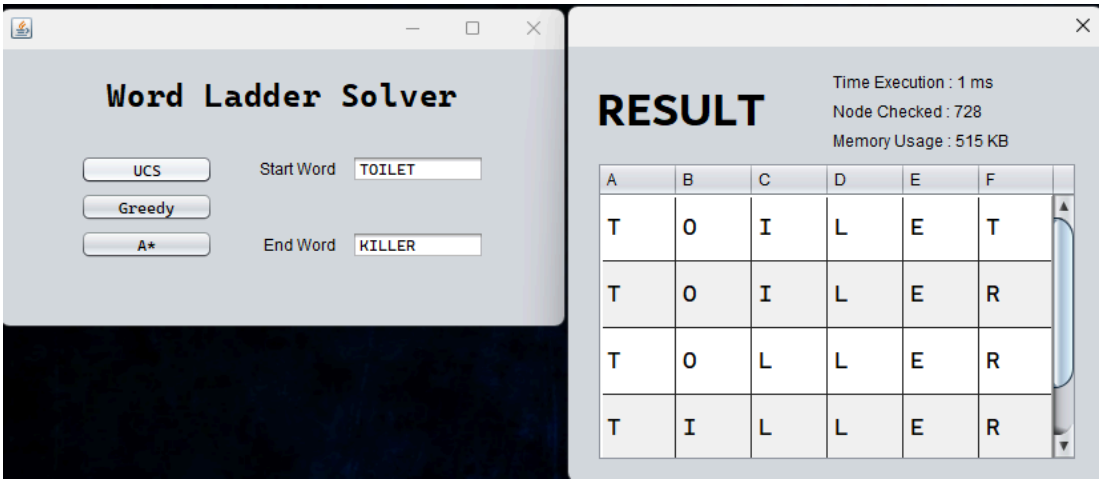
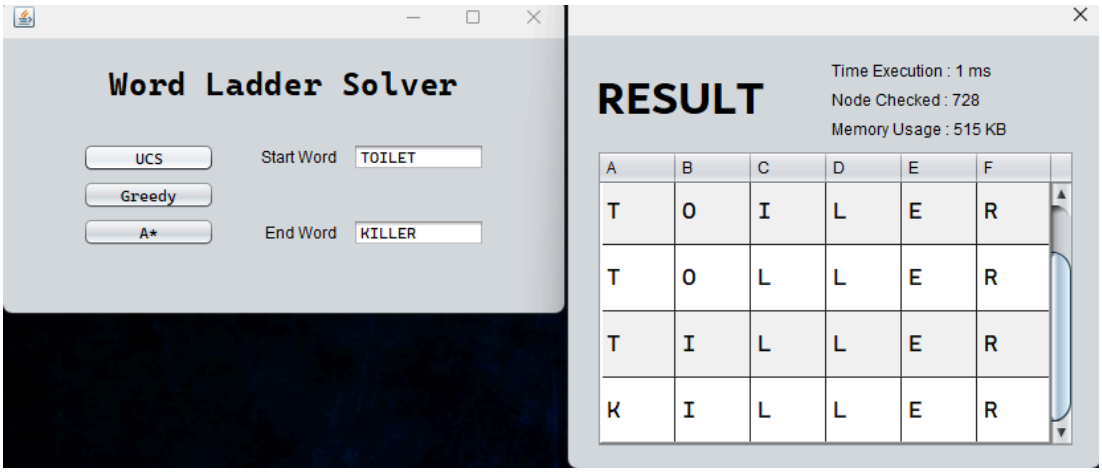
3.3.1 Pengujian Algoritma UCS

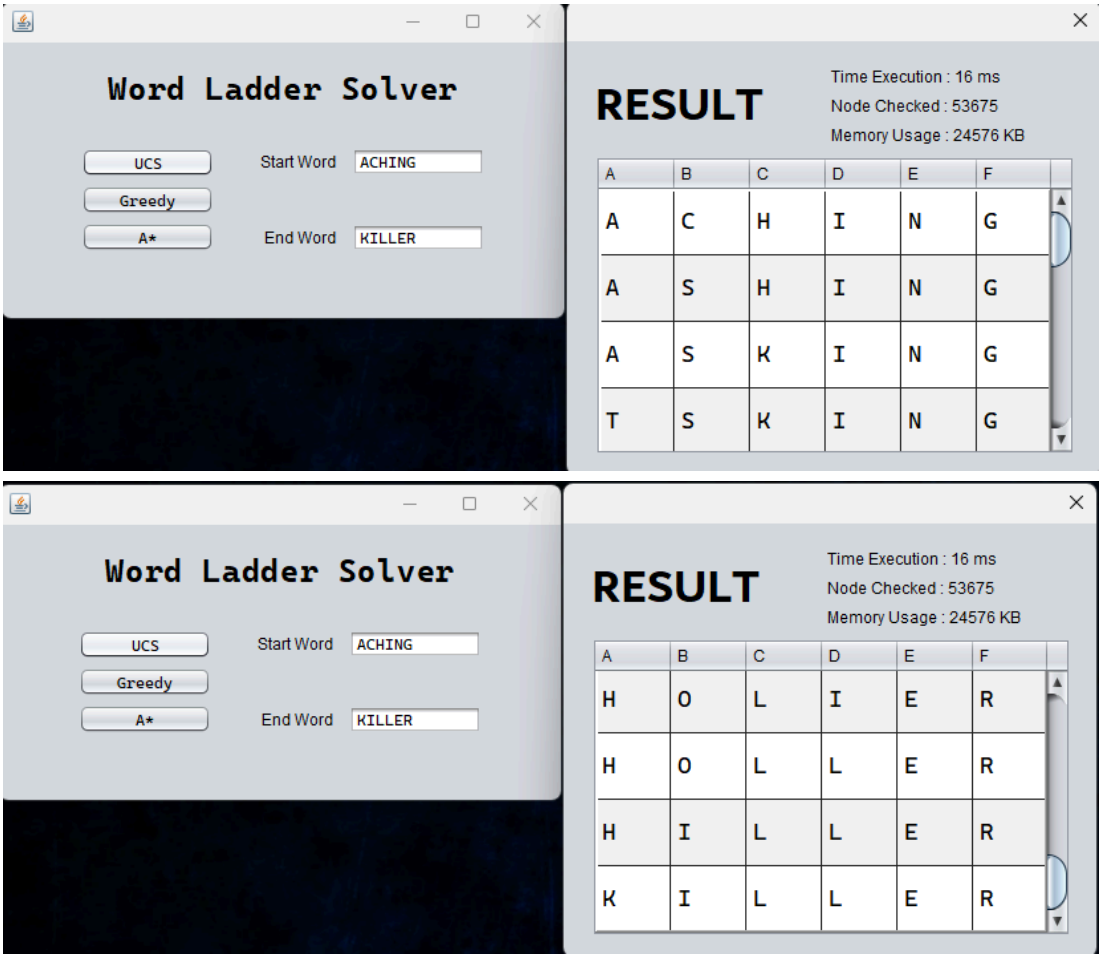
Tabel 3.3.1 *Tabel Pengujian Algoritma UCS*

No	Pengujian
1	

No	Pengujian
2	
3	

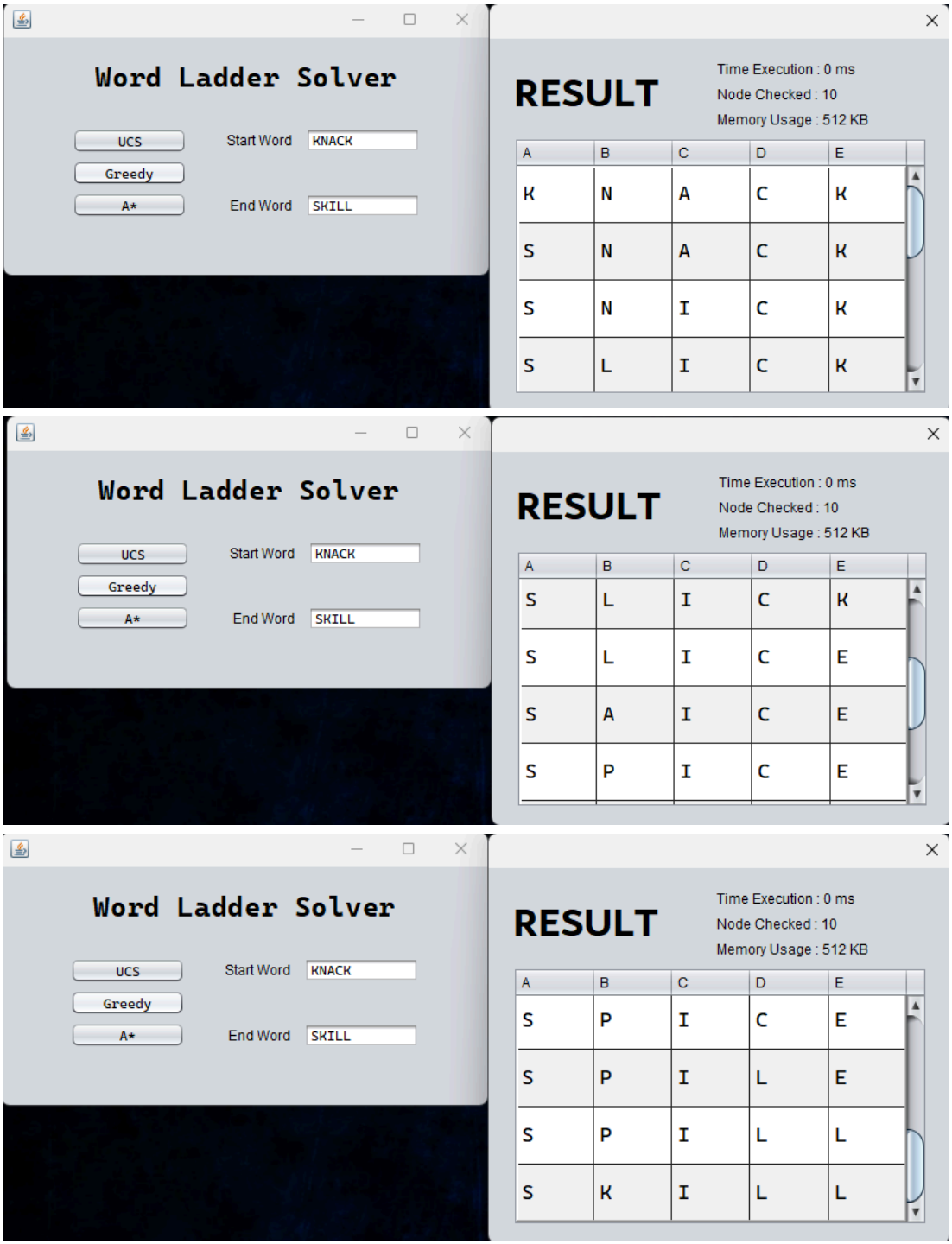
No	Pengujian
	
4	 

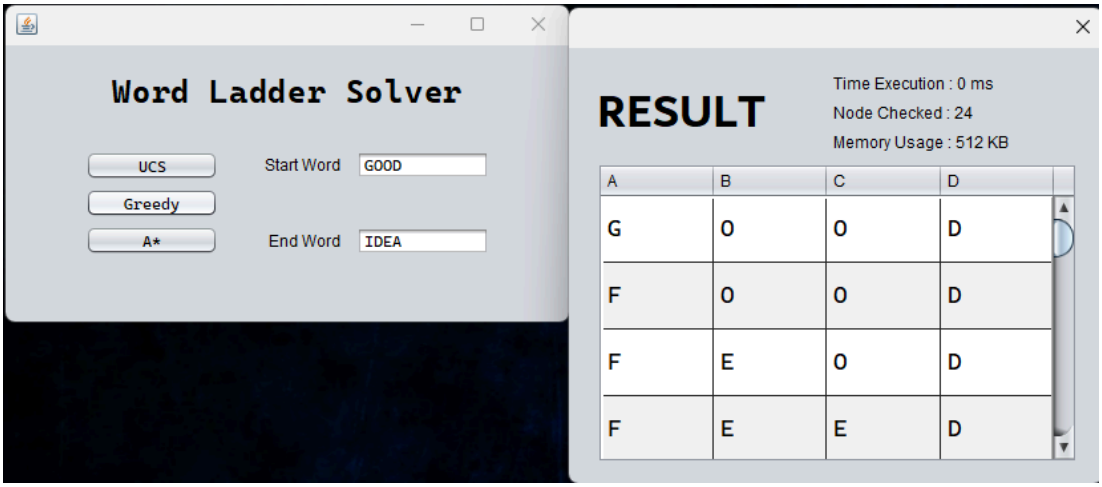
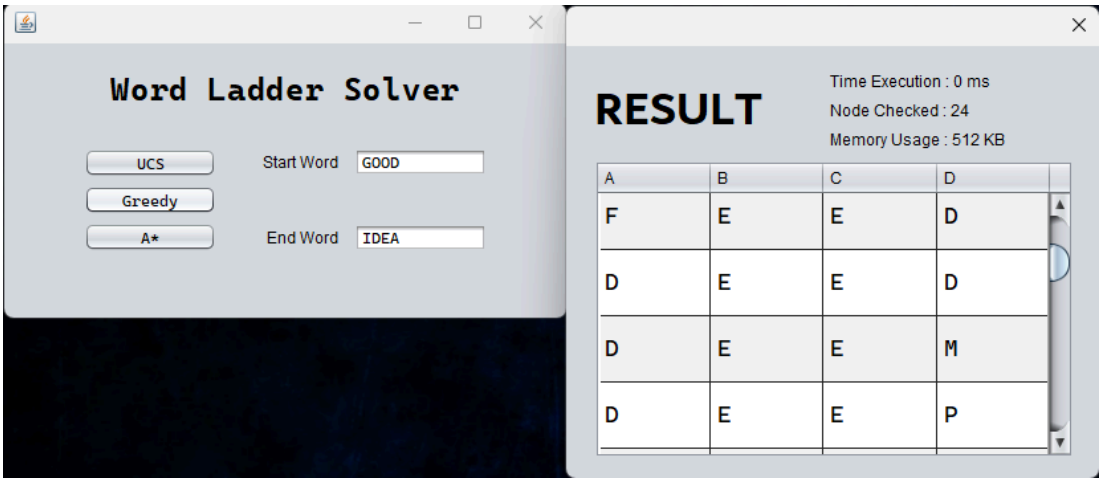
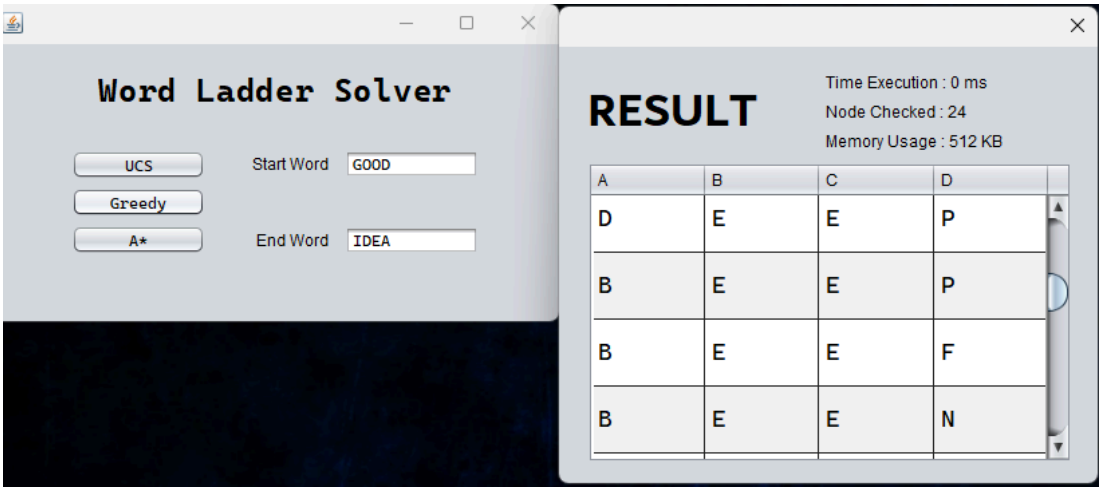
No	Pengujian
5	
6	 

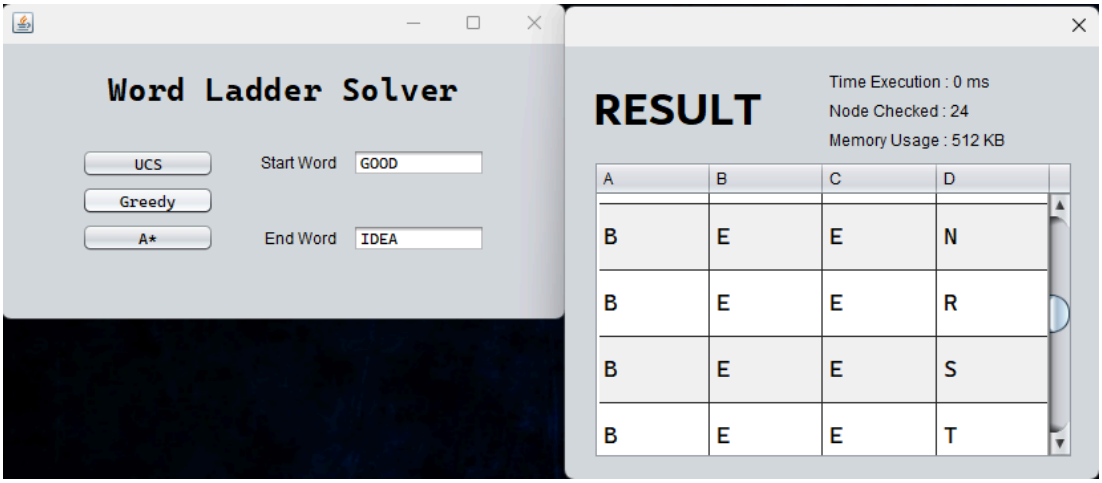
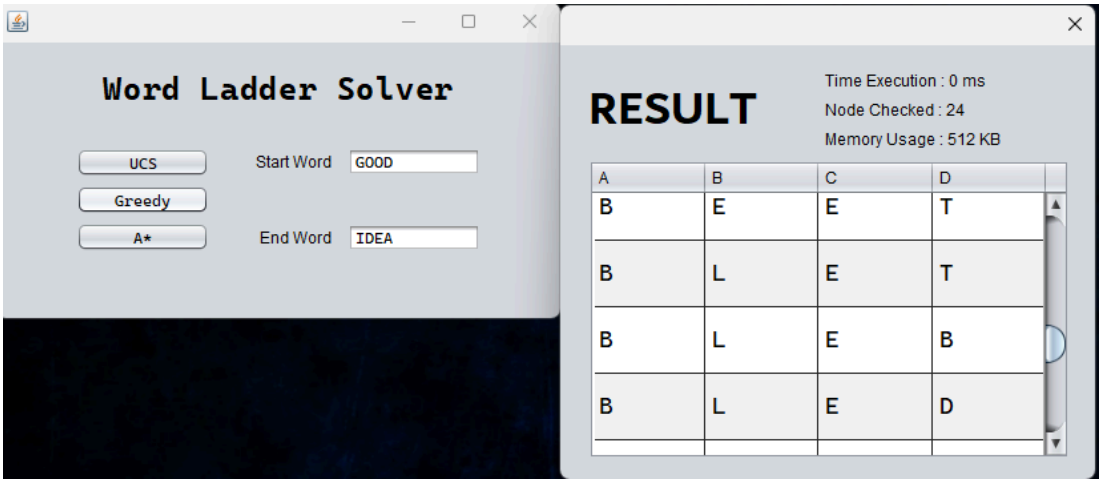
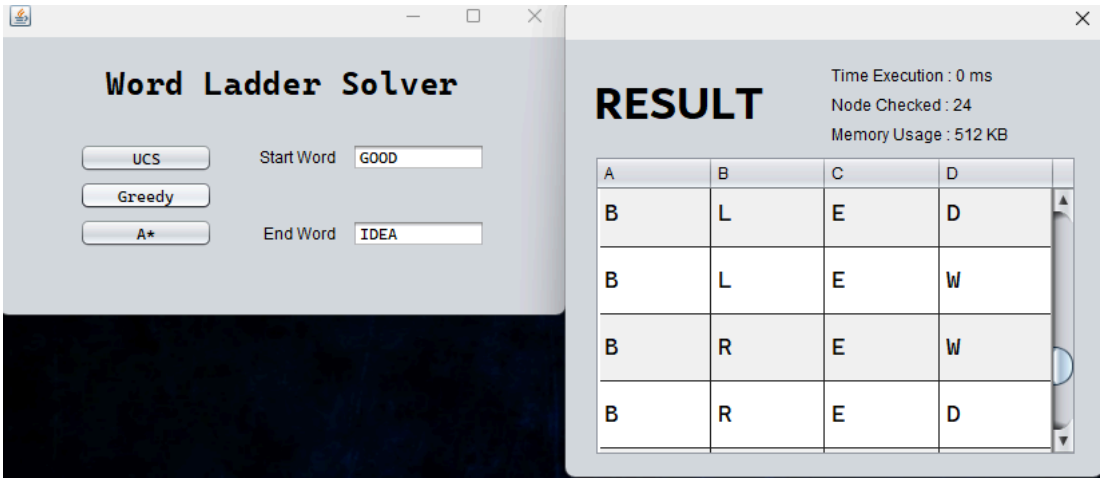
No	Pengujian
7	 Note: Karena path terlalu panjang, maka hasil pengujian ini hanya akan digunakan untuk analisis memory

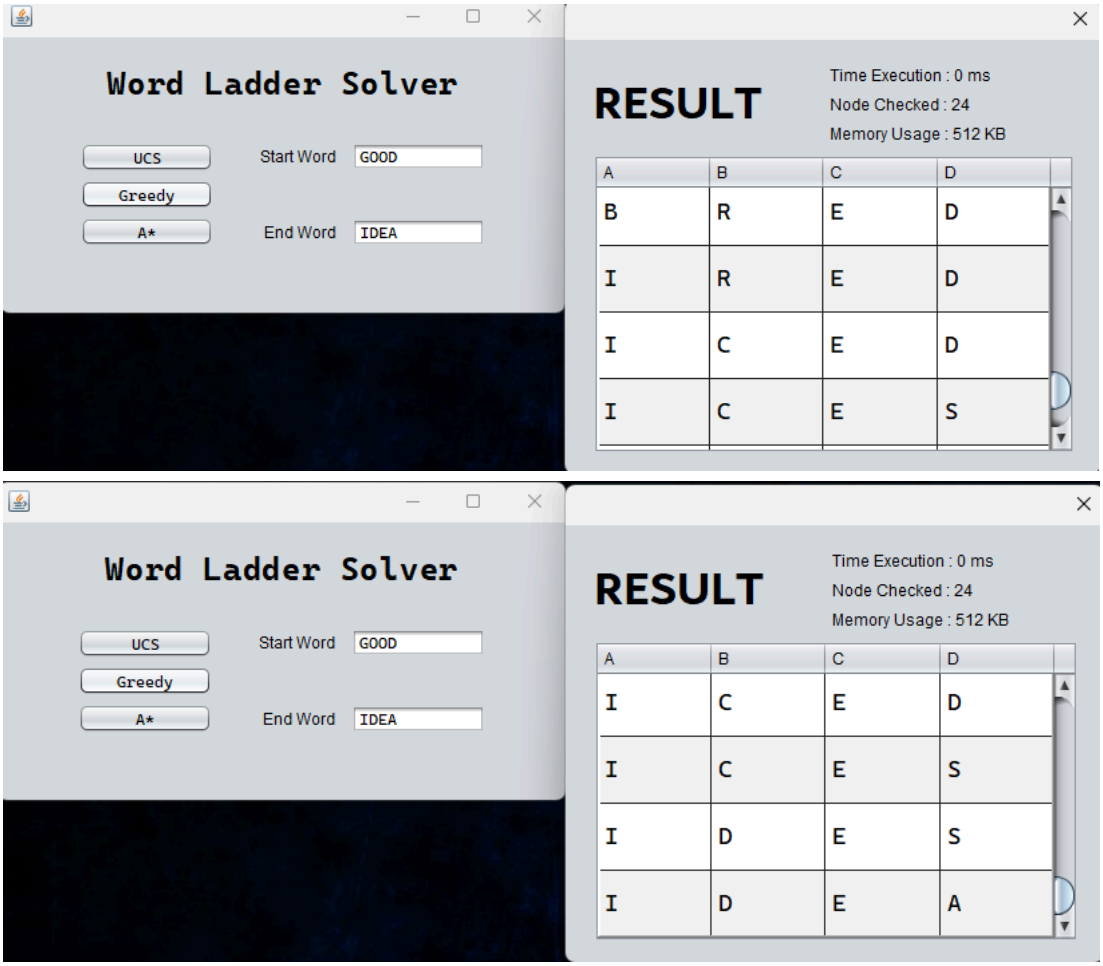
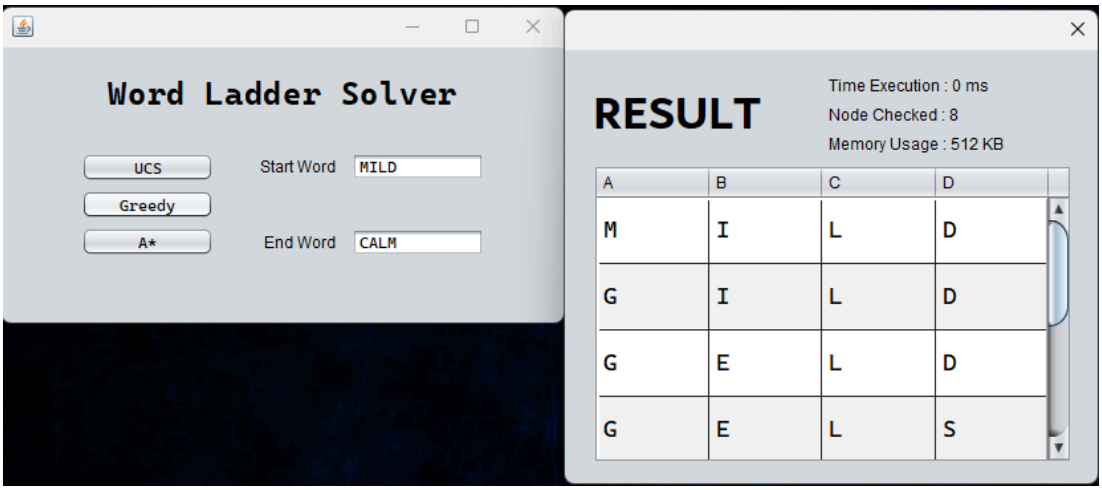
3.3.2 Pengujian Algoritma Greedy Best First Search

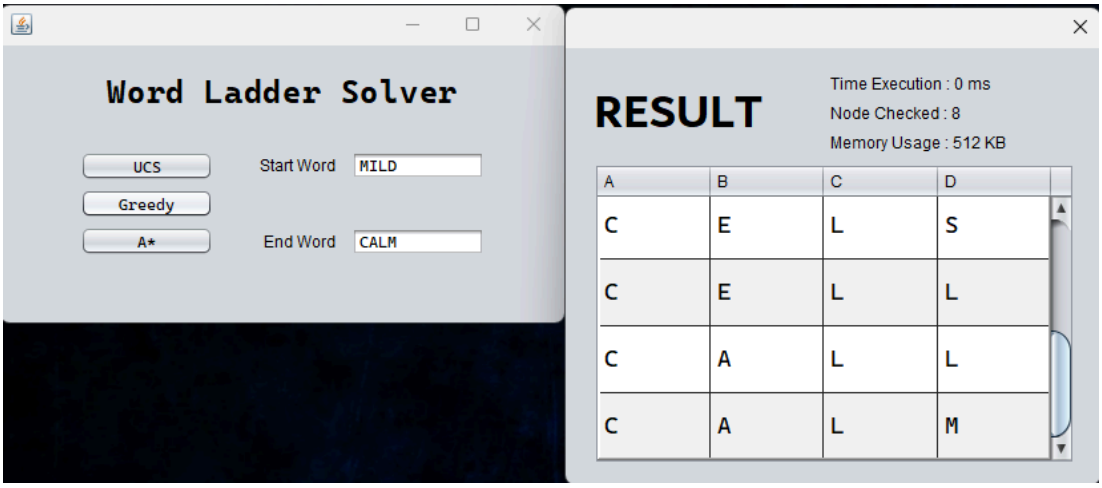
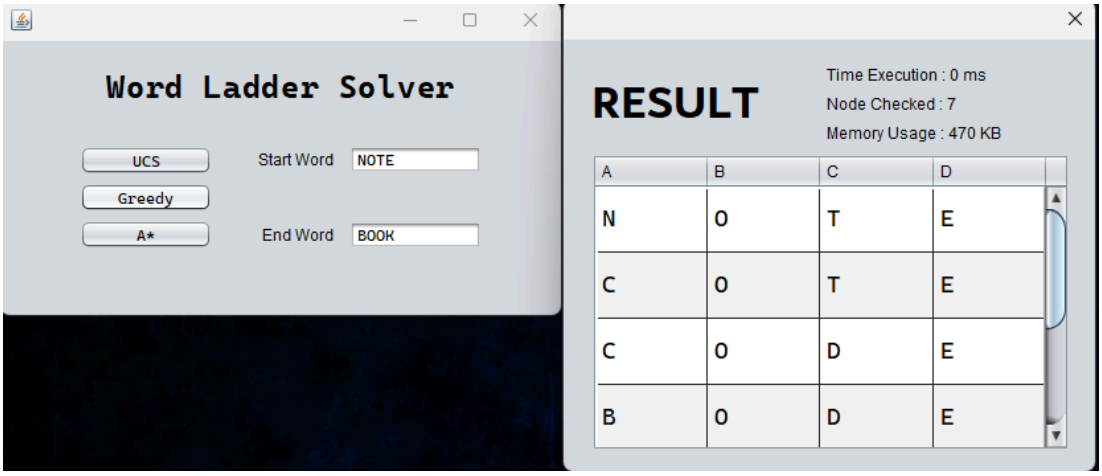
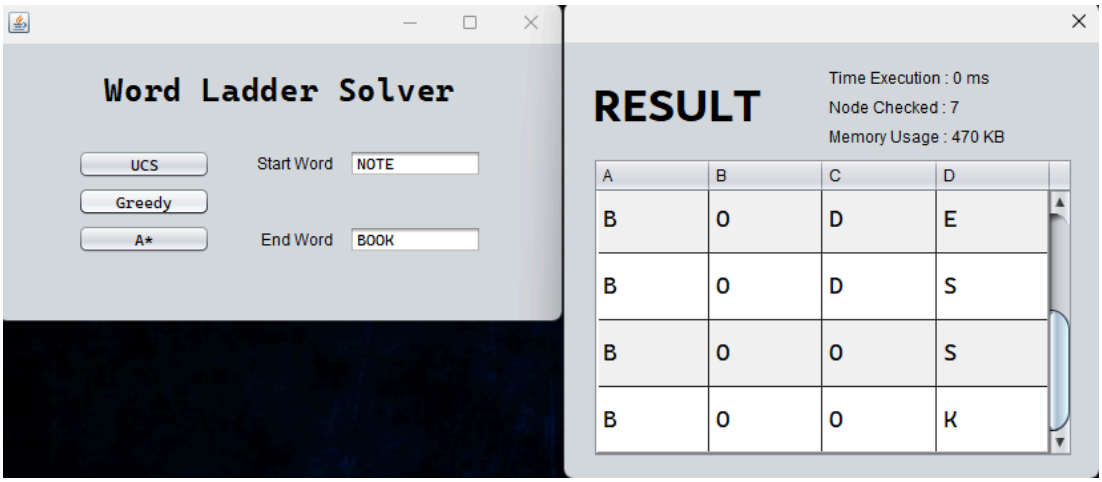
Tabel 3.3.2 *Tabel Pengujian Algoritma Greedy Best First Search*

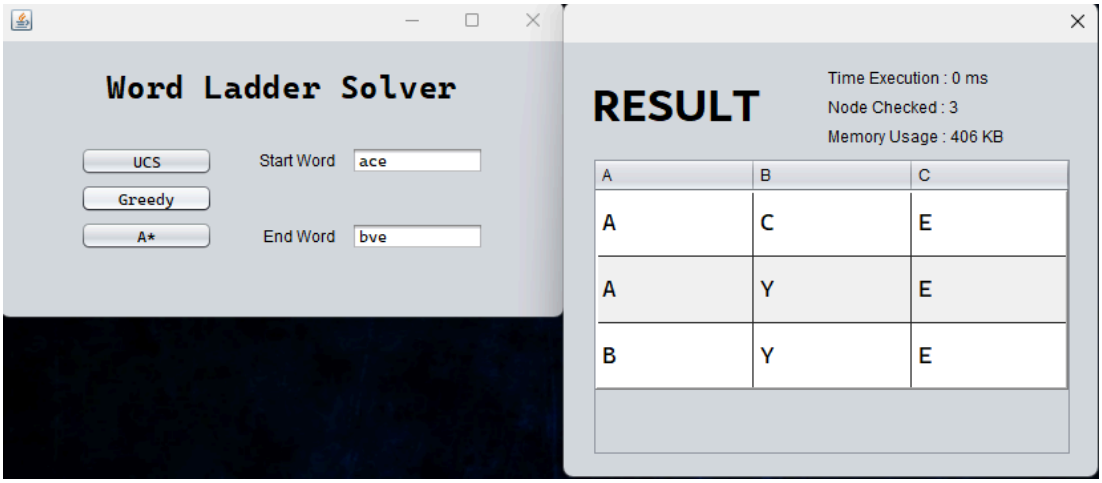
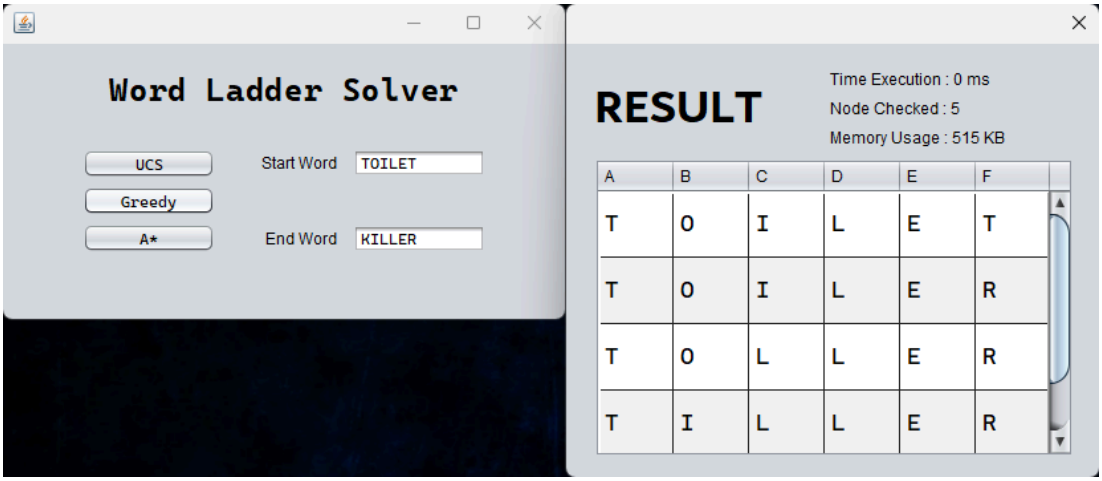
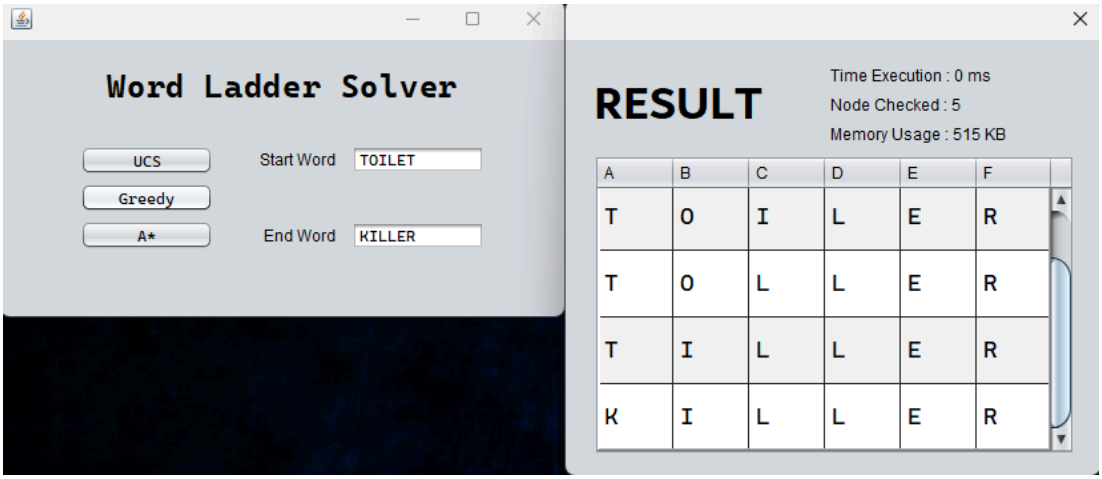
No	Pengujian
1	

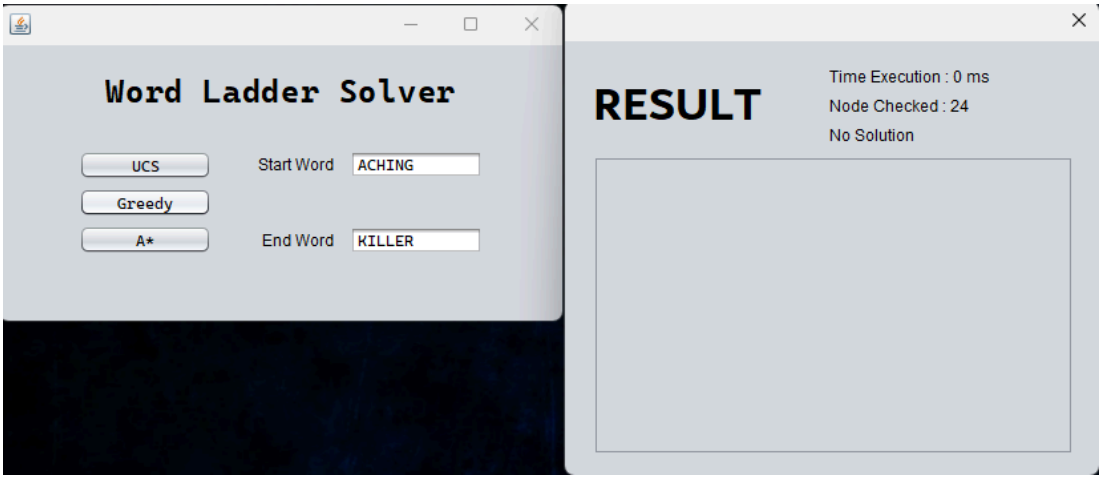
No	Pengujian
2	  

No	Pengujian
	  

No	Pengujian
	
3	

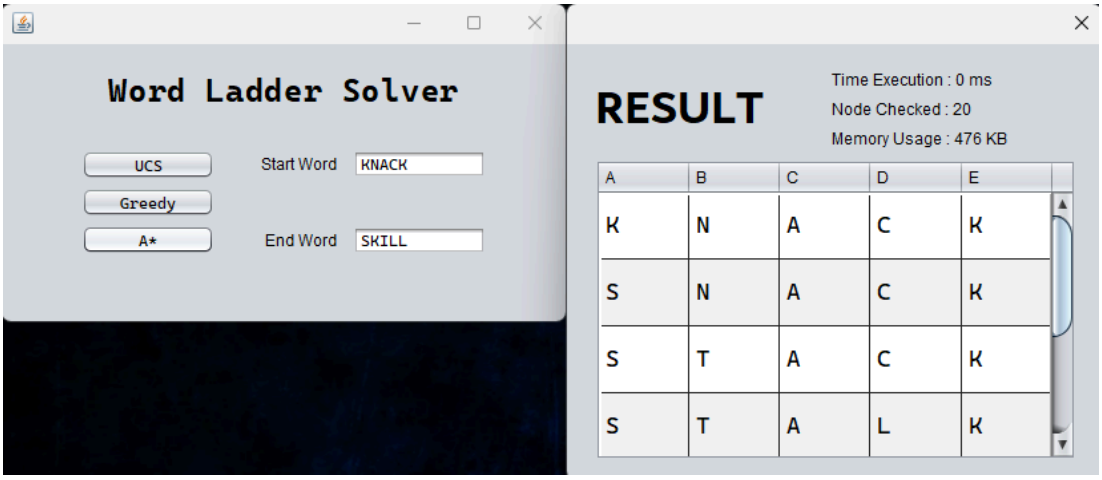
No	Pengujian
	
4	 

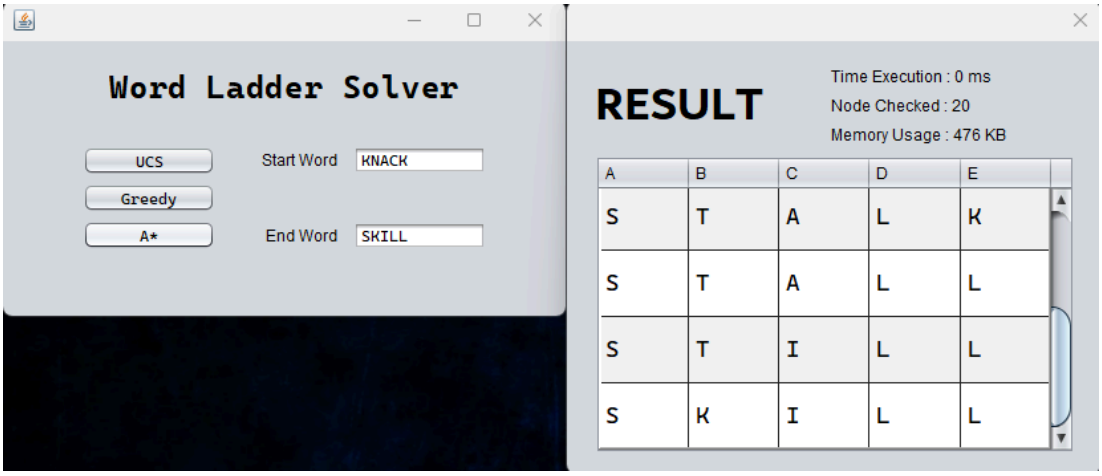
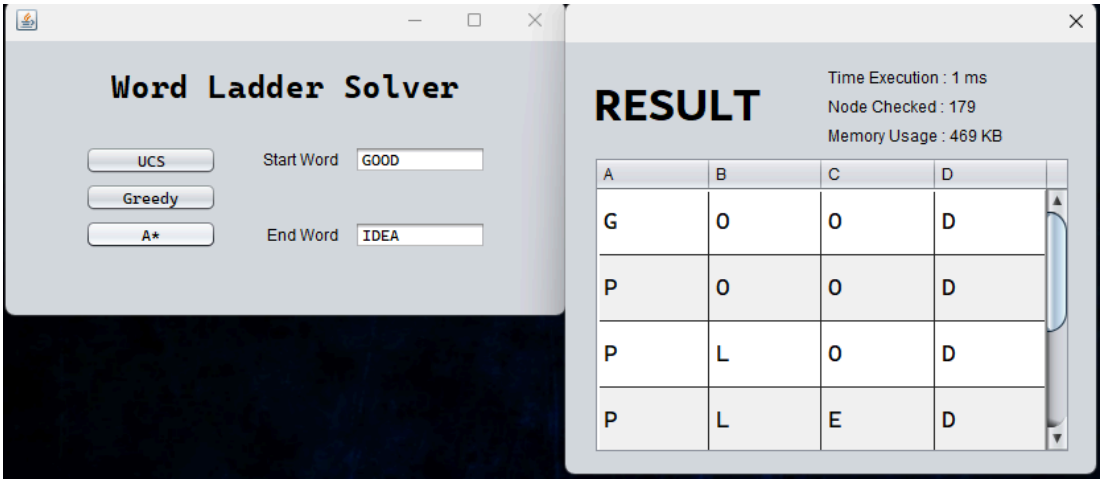
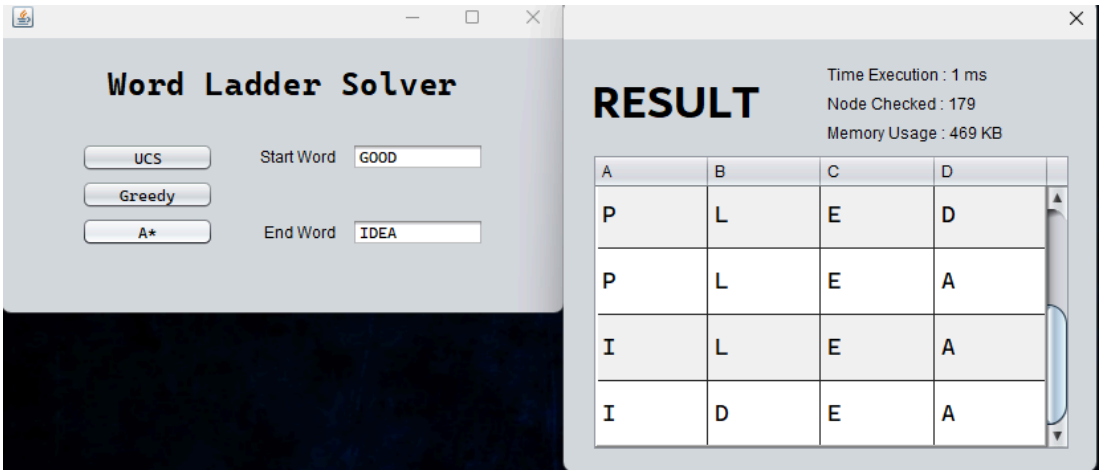
No	Pengujian
5	
6	 

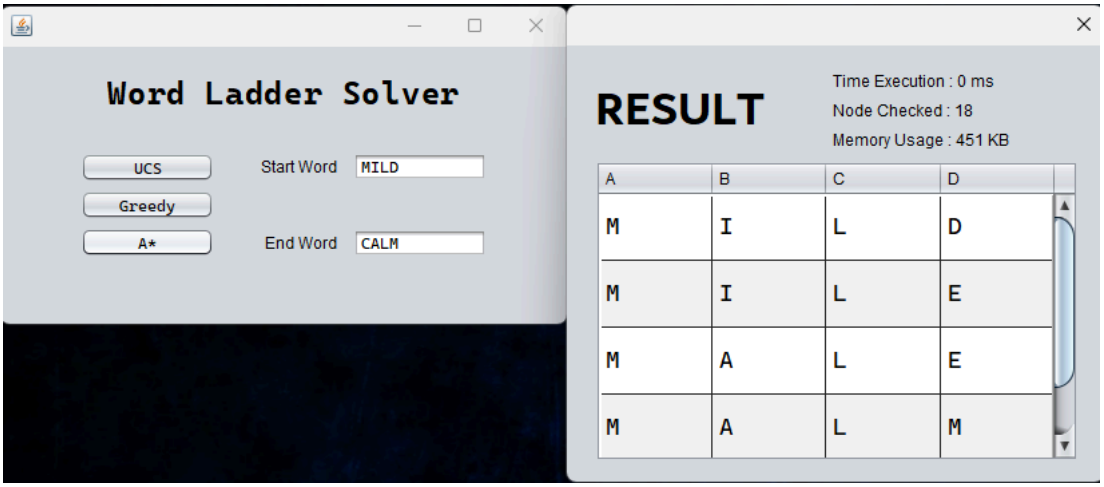
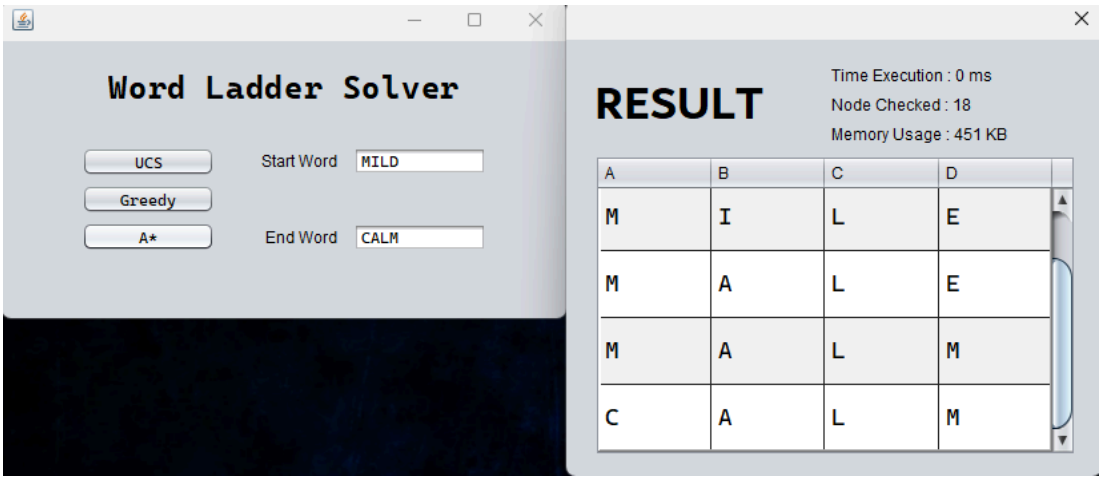
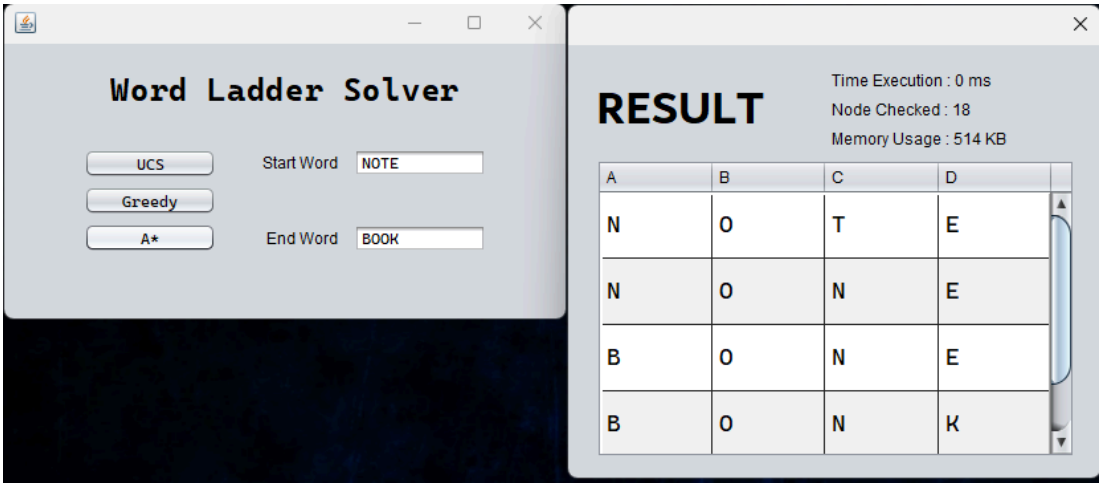
No	Pengujian
7	

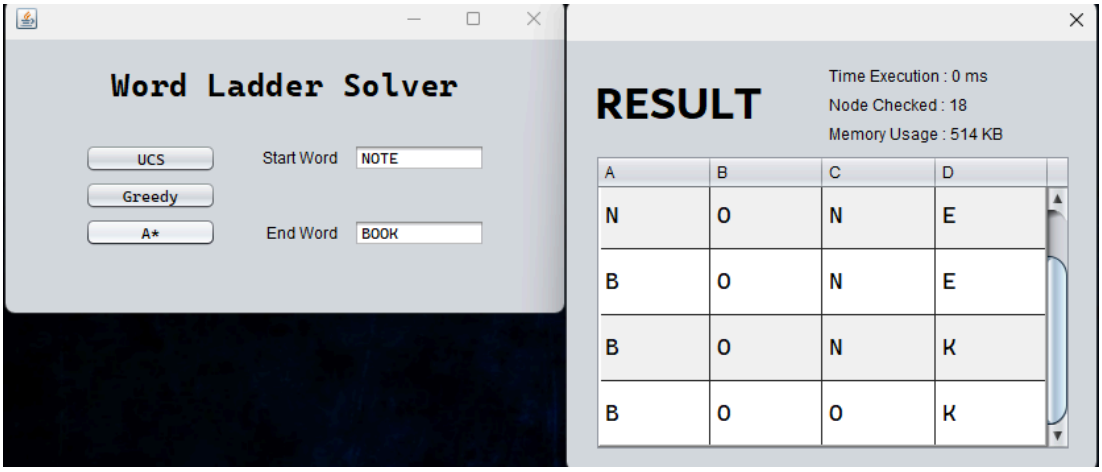
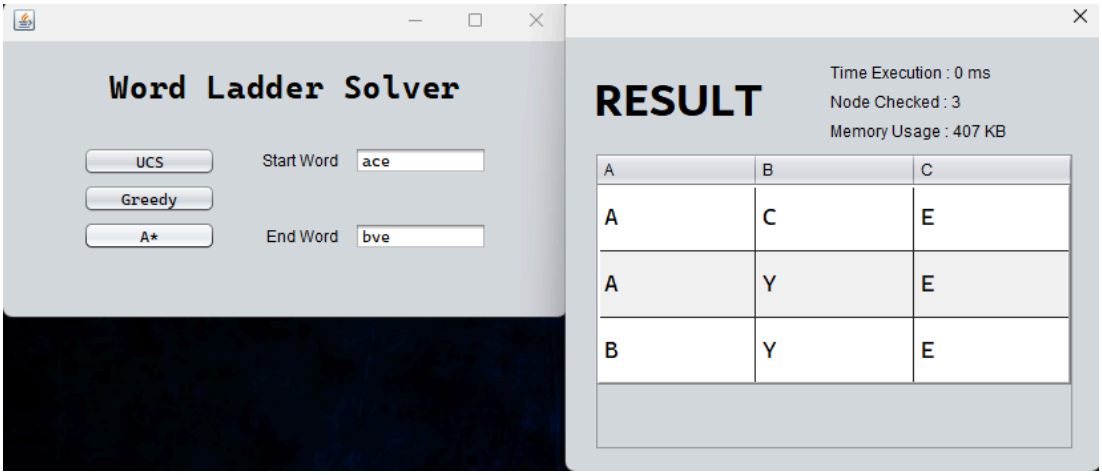
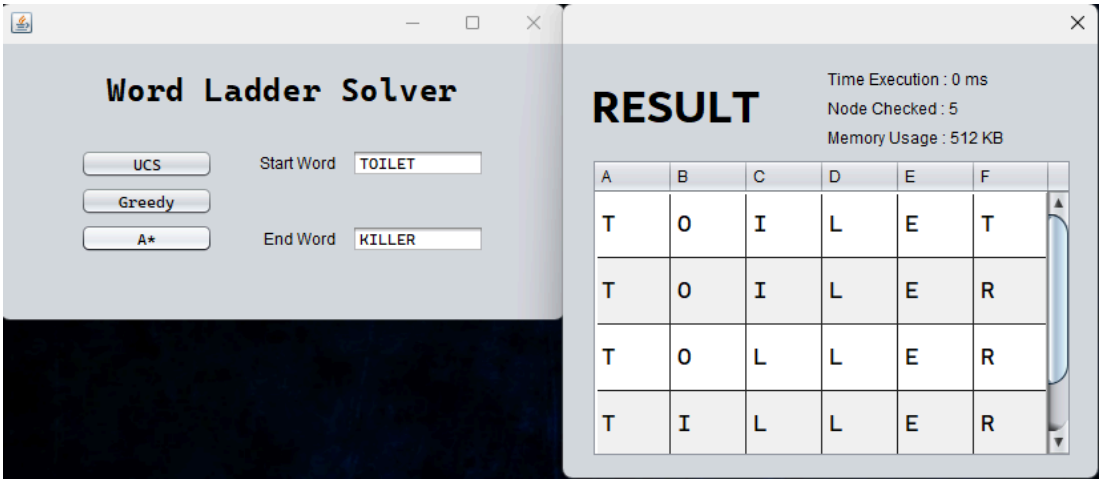
3.3.3 Pengujian Algoritma A*

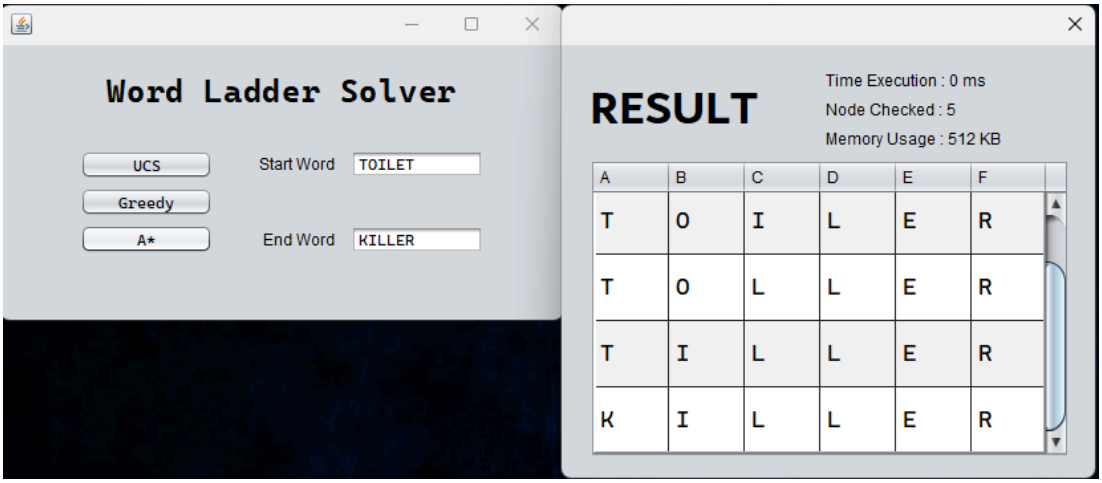
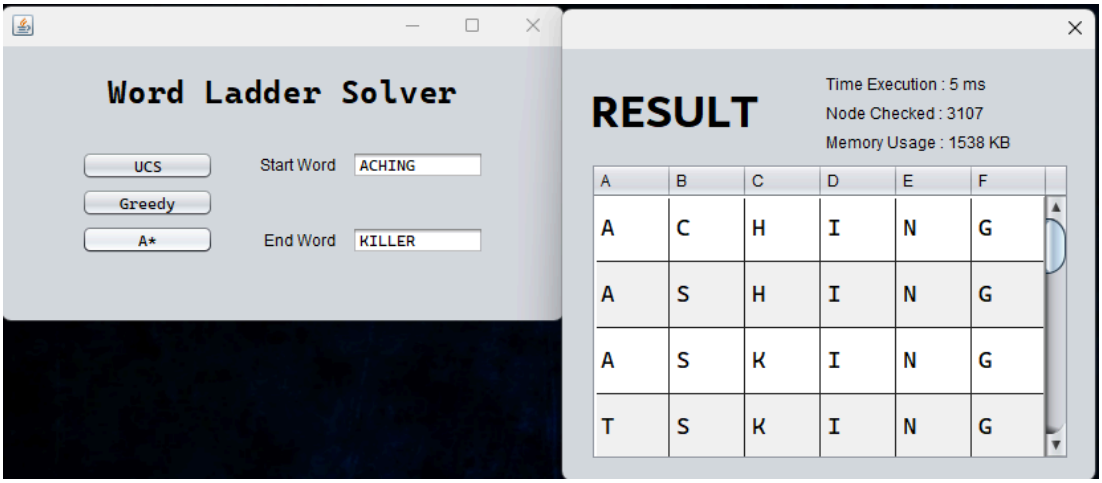
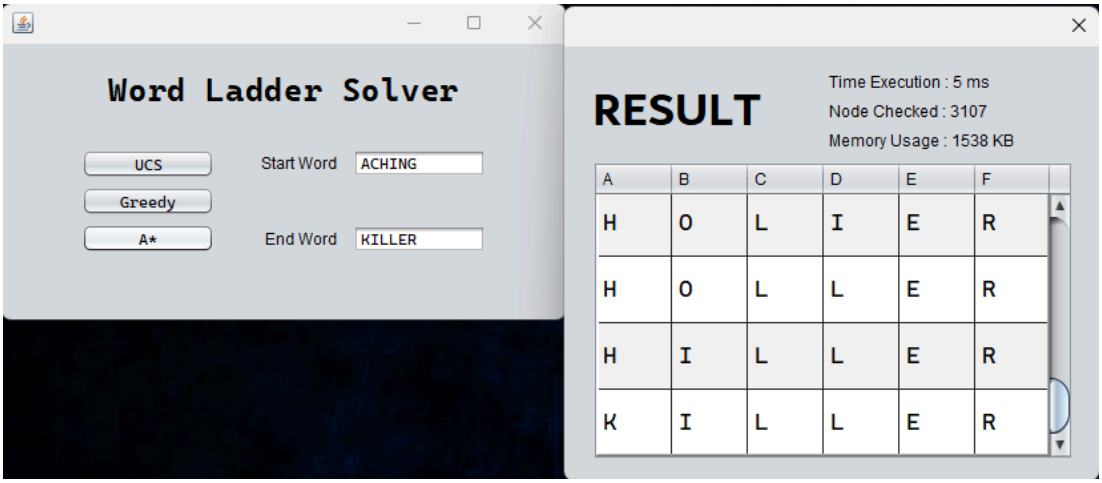
Tabel 3.3.3 *Tabel Pengujian Algoritma A**

No	Pengujian
1	

No	Pengujian
	
2	 

No	Pengujian
3	 
4	

No	Pengujian
	
5	
6	

No	Pengujian
	
7	  Note: Karena path terlalu panjang, maka hasil pengujian ini hanya akan digunakan untuk analisis memory

BAB 4
SOURCE CODE

4.1 Word Checker

```java
package com.mycompany.tucil3_13522164;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.List;
import java.util.ArrayList;
import java.util.Map;
import java.util.HashMap;
import java.util.Scanner;

public class WordChecker {
    public static Map<String, List<String>> wordMap = new HashMap<String, List<String>>();

    public static boolean isWordExist(String word) {
        return wordMap.containsKey(word);
    }

    public static void load() {
        wordMap = new HashMap<String, List<String>>();
        try {
            for (int i = 1; i <= 15; i++) {
                File file = new File("src/main/java/com/mycompany/tucil3_13522164/dict/data" + i + ".txt");
                Scanner scanner = new Scanner(file);
                while (scanner.hasNextLine()) {
                    String temp = scanner.nextLine();
                    Integer len = Integer.parseInt(scanner.nextLine());
                    List<String> list = new ArrayList<String>();
                    for (int j = 0; j < len; j++) {
                        list.add(scanner.nextLine());
                    }
                    if (!wordMap.containsKey(temp)) {
                        wordMap.put(temp, list);
                    }
                }
                scanner.close();
            }
        } catch (FileNotFoundException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }

    public static void loadLength(int lengthWord) {
        wordMap = new HashMap<String, List<String>>();
        try {
            File file = new File("src/main/java/com/mycompany/tucil3_13522164/dict/data" + lengthWord + ".txt");
            Scanner scanner = new Scanner(file);
            while (scanner.hasNextLine()) {
                String temp = scanner.nextLine();
                Integer len = Integer.parseInt(scanner.nextLine());
                List<String> list = new ArrayList<String>();
                for (int j = 0; j < len; j++) {
                    list.add(scanner.nextLine());
                }
                if (!wordMap.containsKey(temp)) {
                    wordMap.put(temp, list);
                }
            }
            scanner.close();
        } catch (FileNotFoundException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}

```

Gambar 4.1 *Source Code Word Checker*

Method *isWordExists(String word)* berguna untuk melakukan pengecekan *word* dalam dictionary, mengembalikan *true* jika word ada dalam dictionary dan kebalikannya. Method *load()* berfungsi untuk melakukan *parsing dictionary* dari folder dict dan memasukan ke dalam atribut wordMap. Method *loadLength(int lengthWord)* berfungsi untuk melakukan *parsing dictionary* dari file data{n}.txt dalam folder dict dimana n merupakan panjang kata yang dicek dan memasukan ke dalam atribut *wordMap*.

4.2 Result

```java
package com.mycompany.tucil3_13522164;

public class Result {
    private String[][] result;
    private int nodeChecked;
    private Long time;
    private int memory;

    Result(String[][] result, int nodeChecked, Long time, int memory) {
        this.result = result;
        this.nodeChecked = nodeChecked;
        this.time = time;
        this.memory = memory;
    }

    public String[][] getResult() {
        return result;
    }

    public int getNodeChecked() {
        return nodeChecked;
    }

    public String getTime() {
        return String.valueOf(time) + " ms";
    }

    public String getMemory() {
        return String.valueOf(memory) + " KB";
    }
}

```

Gambar 4.2 *Source Code Result*

Method *getResult(), getNodeChecked(), getTime(), getMemory()* adalah method untuk mengembalikan nilai dari atribut didalam class *Result*.

4.3 UCS

```java
package com.mycompany.tucil3_13522164;

import java.util.Queue;
import java.util.ArrayList;
import java.util.Set;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;

public class UCS implements Algorithm {
    private String startWord;
    private String endWord;
    private Long Time;
    private Queue<List<String>> path;
    private List<String> result;
    private Set<String> checkedWord;
    private int nodeChecked;

    public UCS(String startWord, String endWord) {
        this.startWord = startWord;
        this.endWord = endWord;
        this.Time = 0L;
        this.path = new LinkedList<>();
        this.result = new ArrayList<String>();
        this.checkedWord = new HashSet<String>();
        this.nodeChecked = 0;
    }

    public void printResult() {
        for (String temp : this.result) {
            System.out.println(temp);
        }
    }

    public String[][] getResult() {
        if (this.result.isEmpty()) {
            return new String[0][0];
        }
        String[][] resultArray = new String[this.result.size()][this.result.get(0).length()];
        for (int i = 0; i < this.result.size(); i++) {
            resultArray[i] = this.result.get(i).split("");
        }
        return resultArray;
    }
```

Gambar 4.3.1 *Source Code UCS-1*

```java
public Result solve() {
        // Activate Garbage Collector and Get Current Memory Usage
        System.gc();
        int memoryNow = (int) (Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()) / 1024;

        // Initialize Start Time and Path
        Long startTimeLong = System.currentTimeMillis();
        path.add(new ArrayList<String>() { {
            add(startWord);
        } });
        checkedWord.add(startWord);

        while (!path.isEmpty()) {

            // Get First Path
            List<String> currentPath = path.poll();
            String tempWord = currentPath.get(currentPath.size() - 1);
            this.nodeChecked++;

            // Check if the word is the solution is found
            if (tempWord.equals(endWord)) {
                this.result = currentPath;
                this.Time = System.currentTimeMillis() - startTimeLong;
                int memory = (int) (Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()) / 1024;
                System.gc();
                return new Result(getResult(), nodeChecked, this.Time, memory-memoryNow);
            }

            // Add to checked word
            checkedWord.add(tempWord);

            // Get next linked word
            List<String> linkingWordList = WordChecker.wordMap.get(tempWord);

            // Skip evaluation proccess if no linked word exist
            if (linkingWordList.isEmpty()) {
                continue;
            }

            // Evaluate each linked word
            for (String word : linkingWordList) {
                if (!checkedWord.contains(word)) {
                    List<String> newPath = new ArrayList<>(currentPath);
                    newPath.add(word);
                    path.add(newPath);
                }
            }
        }

        // No Solution
        this.result.clear();
        this.Time = System.currentTimeMillis() - startTimeLong;
        return new Result(getResult(), nodeChecked, this.Time, 0);
    }
}

```

Gambar 4.3.2 *Source Code UCS-2*

4.4 Greedy

```java

    public Result solve() {
        // Activate Garbage Collector and Get Current Memory Usage
        System.gc();
        int memoryNow = (int) (Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()) / 1024;

        // Initialize Start time
        String tempWord = startWord;
        Long startTimeLong = System.currentTimeMillis();

        while (!tempWord.equals(endWord)) {

            // Add to Result
            this.result.add(tempWord);

            // Get next linked word
            List<String> linkingWordList = WordChecker.wordMap.get(tempWord);

            // add node checked
            this.nodeChecked++;

            // No Solution
            if (linkingWordList.isEmpty()) {
                System.out.println("No Solution");
                this.Time = System.currentTimeMillis() - startTimeLong;
                return new Result(new String[0][0], this.nodeChecked, this.Time, 0);
            }

            // Get Most Similiar Children with EndWord but not in result
            tempWord = Utils.getFirstMostSimiliar(linkingWordList, endWord, result);

            if (tempWord.equals("")) {
                System.out.println("No Solution");
                this.Time = System.currentTimeMillis() - startTimeLong;
                return new Result(new String[0][0], this.nodeChecked, this.Time, 0);
            }
        }

        // Solution Found
        this.result.add(endWord);
        this.nodeChecked++;
        this.Time = System.currentTimeMillis() - startTimeLong;
        int memory = (int) (Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()) / 1024;
        System.gc();
        return new Result(getResult(), this.nodeChecked, this.Time, memory - memoryNow);
    }
}

```

Gambar 4.4.1 *Source Code Greedy-1*

```java
package com.mycompany.tucil3_13522164;

import java.util.List;
import java.util.ArrayList;

public class Greedy implements Algorithm {
    private String startWord;
    private String endWord;
    private List<String> result;
    private int nodeChecked;
    private Long Time;

    public Greedy(String startWord, String endWord) {
        this.startWord = startWord;
        this.endWord = endWord;
        this.result = new ArrayList<String>();
        this.nodeChecked = 0;
        this.Time = 0L;
    }

    public void printResult() {
        for (String temp : this.result) {
            System.out.println(temp);
        }
    }

    public String[][] getResult() {
        if (this.result.isEmpty()) {
            return new String[0][0];
        }
        int wordLength = this.result.get(0).length();
        String[][] result = new String[this.result.size()][wordLength];
        for (String temp : this.result) {
            for (int i = 0; i < wordLength; i++) {
                result[this.result.indexOf(temp)][i] = String.valueOf(temp.charAt(i));
            }
        }
        return result;
    }
```

Gambar 4.4.2 *Source Code Greedy-2*

4.5 A*

```java
package com.mycompany.tucil3_13522164;

import java.util.ArrayList;
import java.util.List;

public class AStarNode {
    private List<String> path;
    private int cost;

    public AStarNode(List<String> path, int cost) {
        this.path = new ArrayList<>(path);
        this.cost = cost;
    }

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

    public String getLastWord() {
        return path.get(path.size() - 1);
    }

    public int getCost() {
        return cost;
    }
}

```

Gambar 4.5.1 *Source Code A* Node*

```java
package com.mycompany.tucil3_13522164;

import java.util.Comparator;

public class AStarNodeComparator implements Comparator<AStarNode> {

    @Override
    public int compare(AStarNode node1, AStarNode node2) {
        if (node1.getCost() < node2.getCost()) {
            return -1;
        } else if (node1.getCost() > node2.getCost()) {
            return 1;
        }
        return 0;
    }
}

```

Gambar 4.5.2 *Source Code A* Node Comparator*

```java
package com.mycompany.tucil3_13522164;

import java.util.ArrayList;
import java.util.List;
import java.util.Queue;
import java.util.PriorityQueue;
import java.util.Set;
import java.util.HashSet;

public class AStar implements Algorithm {
    private String startWord;
    private String endWord;
    private Long time;
    private Queue<AStarNode> path;
    private AStarNode result;
    private int nodeChecked;
    private Set<String> checkedWord;

    public AStar(String startWord, String endWord) {
        this.startWord = startWord;
        this.endWord = endWord;
        this.time = 0L;
        this.path = new PriorityQueue<AStarNode>(new AStarNodeComparator());
        this.nodeChecked = 0;
        this.checkedWord = new HashSet<String>();
    }

    public void printResult() {
        if (result == null) {
            System.out.println("No Solution");
            return;
        }
        for (String temp : result.getPath()) {
            System.out.println(temp);
        }
    }

    public String[][] getResult() {
        if (result == null) {
            return new String[0][0];
        }
        List<String> resultPath = result.getPath();
        String[][] resultArray = new String[resultPath.size()][startWord.length()];
        for (int i = 0; i < resultPath.size(); i++) {
            resultArray[i] = resultPath.get(i).split("");
        }
        return resultArray;
    }
```

Gambar 4.5.3 *Source Code A*-1*

```java
public Result solve() {
        // Activate Garbage Collector and Get Current Memory Usage
        System.gc();
        int memoryNow = (int) (Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()) / 1024;

        // Initialize Start Time and Path
        Long startTimeLong = System.currentTimeMillis();
        List<String> startPath = new ArrayList<String>(){{
            add(startWord);
        }};
        path.add(new AStarNode(startPath, getCost(startPath)));

        while (!path.isEmpty()) {

            // Get First Path
            AStarNode currentPath = path.poll();

            // get last word in path
            String tempWord = currentPath.getLastWord();

            // add node checked
            nodeChecked++;

            // Check if the word is the solution is found
            if (tempWord.equals(endWord)) {
                this.result = currentPath;
                this.time = System.currentTimeMillis() - startTimeLong;
                int memory = (int) (Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()) / 1024;
                System.gc();
                return new Result(getResult(), nodeChecked, this.time, memory - memoryNow);
            }
            checkedWord.add(tempWord);

            // Get next linked word
            List<String> linkingWordList = WordChecker.wordMap.get(tempWord);

            // No Solution
            if (linkingWordList.isEmpty()) {
                System.out.println("No Solution");
                this.result = null;
                this.time = System.currentTimeMillis() - startTimeLong;
                return new Result(new String[0][0], nodeChecked, this.time, 0);
            }

            // Evaluate each linked word
            for (String word : linkingWordList) {
                if (!checkedWord.contains(word)) {
                    List<String> newPath = new ArrayList<>(currentPath.getPath());
                    newPath.add(word);
                    path.add(new AStarNode(newPath, getCost(newPath)));
                }
            }
        }

        // No Solution
        this.result = null;
        this.time = System.currentTimeMillis() - startTimeLong;
        return new Result(new String[0][0], nodeChecked, this.time, 0);
    }

    private int getCost(List<String> path) {
        int cost = path.size() - 1;
        return cost + Utils.getDifferentCharacter(path.get(cost), endWord);
    }
}
```

Gambar 4.5.4 *Source Code A*-2*

BAB 5
ANALISIS HASIL UJI

Berdasarkan cara mengimplementasi algoritma, dapat diketahui bahwa dari masing-masing algoritma (UCS, *Greedy Best First Search*, dan *A**) menggunakan pendekatan fungsi evaluasi yang berbeda-beda. Algoritma UCS menggunakan fungsi evaluasi dengan menghitung banyaknya perbedaan huruf pada posisi yang sama antara simpul satu dengan simpul lainnya dari suatu rute. Fungsi evaluasi tersebut tergolong ke dalam konsep fungsi evaluasi dari awal hingga posisi saat ini atau fungsi evaluasi *start-to-node*. Sehingga fungsi evaluasi yang digunakan oleh Algoritma UCS adalah g(n). Algoritma *Greedy Best First Search* menggunakan fungsi evaluasi dengan menghitung kemiripan atau banyaknya huruf yang sama dengan posisi yang sama juga antara simpul saat ini dengan simpul tujuan. Fungsi evaluasi tersebut tergolong ke dalam konsep fungsi evaluasi dari posisi saat ini dengan posisi akhir atau fungsi evaluasi *node-to-end*. Sehingga fungsi evaluasi yang digunakan oleh algoritma *Greedy Best First Search* adalah h(n). Algoritma *A** menggunakan fungsi evaluasi dengan menghitung banyaknya perbedaan huruf pada posisi yang sama antar simpul satu dengan simpul setelahnya dari rute saat ini dan menghitung juga perbedaan huruf pada posisi yang sama antara simpul akhir dalam rute saat ini dengan simpul akhir/tujuan. Fungsi evaluasi menghitung perbedaan huruf pada posisi yang sama antar simpul satu dengan simpul setelahnya dalam satu rute tergolong kedalam fungsi evaluasi *start-to-node* dan fungsi evaluasi menghitung perbedaan huruf pada posisi yang sama antara simpul saat ini dengan simpul tujuan tergolong kedalam fungsi evaluasi *node-to-end*. Sehingga algoritma *A** menggunakan fungsi evaluasi g(n) + h(n).

Berdasarkan hasil pengujian pada bab 3.3 khususnya pada tabel 3.2.3, didapatkan bahwa untuk setiap hasil pencarian menggunakan algoritma A* menghasilkan solusi yang memiliki bobot asli f(n) lebih besar dari bobot estimasi atau *true cost* f*(n) yang membuat seluruh solusi dari algoritma A* ini adalah solusi yang *admissible*. Sebagai contoh, pada pengujian nomor satu, dimana kata awal adalah “KNACK” dan kata tujuan adalah “SKILL”. Perhitungan estimasi atau *true cost* (f*(n)) pada pengujian tersebut menghasilkan bobot sebesar lima yang didapat dari perhitungan perbedaan huruf dari kata awal dengan kata akhir yaitu lima huruf berbeda dan juga jumlah perbedaan huruf dari awal hingga posisi saat ini

yang dalam hal ini bernilai nol. Sedangkan perhitungan bobot aslinya didapat untuk pengujian ini memiliki bobot enam, yang didapat dari perhitungan perbedaan kata saat ini ("SKILL") dengan kata akhir ("SKILL") yang bernilai nol ditambah dengan jumlah perbedaan huruf antara simpul satu dengan simpul setelahnya dari rute tersebut ("KNACK" → "SNACK" → "STACK" → "STALK" → "STALL" → "STILL" → "SKILL") yang berbobot enam. Sehingga didapat hasil bahwa bobot asli yang didapat akan lebih besar sama dengan bobot estimasi atau *true cost*. Hal tersebut berlaku juga untuk pengujian lainnya, sehingga dapat dikatakan bahwa heuristik yang digunakan oleh algoritma A* selalu *admissible*.

Pada kasus pencarian solusi *word ladder*, Algoritma UCS yang memiliki fungsi evaluasi berdasarkan jumlah perbedaan huruf antara simpul satu dengan simpul setelahnya memiliki kesamaan dengan algoritma BFS yang sama-sama melakukan pencarian dan menyimpan simpul anak ke urutan paling belakang *queue* dan memproses simpul dari urutan terdepan *queue*. Hal tersebut dapat terjadi karena simpul-simpul pada algoritma UCS memiliki perbedaan bobot satu pada setiap levelnya (berdasarkan peraturan permainan *word ladder*) yang dalam hal ini bisa digantikan dengan urutan pada *queue* saja. Apabila dimungkinkan untuk algoritma UCS melanggar peraturan dari permainan *word ladder* yaitu maksimal perbedaan huruf antar simpul satu dengan simpul setelahnya tidak harus satu, maka algoritma UCS tidak bisa disamakan dengan algoritma BFS.

Berdasarkan hasil pengujian pada bab 3.3 khususnya pengujian nomor 7, didapatkan bahwa memori yang terpakai saat melakukan pencarian dengan algoritma UCS adalah 53675 KB dan melakukan pengecekan 24576 simpul. Sedangkan memori yang terpakai saat menjalankan algoritma A* hanya sebesar 3107 KB saja dan melakukan pengecekan ke 1538 simpul saja. Hal ini sangat menggambarkan perbedaan efisiensi algoritma UCS dan algoritma A* dimana algoritma UCS yang merupakan *Uninformed Search* yang berarti tidak ada informasi tambahan terpaksa harus melakukan pengecekan lebih luas dan pada akhirnya harus menggunakan memori yang cukup besar. Sedangkan algoritma A* yang merupakan *Informed Search* atau *Heuristic Search* yang berarti memiliki informasi tambahan dalam melakukan pencarian sehingga hanya simpul kandidat terbaik yang akan dievaluasi dan akibatnya pencarian tidak seluas pencarian yang dilakukan oleh

algoritma UCS. Sehingga, dapat disimpulkan bahwa algoritma A* memiliki tingkat efisiensi lebih tinggi dibanding dengan algoritma UCS pada permasalahan pencarian solusi *word ladder*. Hal ini bersesuaian dengan teori, dimana A* menggunakan fungsi evaluasi heuristik dan juga memiliki tambahan informasi heuristik sedangkan UCS hanya menggunakan fungsi evaluasi dari awal hingga posisi saat ini dengan tanpa adanya informasi tambahan. Sehingga efisiensi algoritma A* akan lebih baik daripada algoritma UCS.

Pada hasil pengujian di bab 3.3 khususnya pengujian nomor 7, ditemukan bahwa algoritma *Greedy Best First Search* gagal dalam menemukan solusi rute. Hal tersebut dapat terjadi akibat dari algoritma *Greedy Best First Search* yang tidak bisa melakukan *backtracking* dan selalu mengambil simpul anak terbaik yang pertama kali ditemukan sebagai rute solusi, sehingga ada kemungkinan algoritma ini tersesat atau terjebak saat melakukan pencarian apabila rute solusi harus melalui simpul anakan terbaik yang bukan pertama kali ditemukan. Sehingga, algoritma *Greedy Best First Search* belum tentu memberikan solusi yang paling optimal pada permasalahan pencarian solusi *word ladder*. Berdasarkan teori, algoritma *Greedy Best First Search* melakukan pengambilan simpul selanjutnya berdasarkan simpul terbaik pada setiap iterasi-nya. Sehingga, algoritma ini memungkinkan terjadinya pencarian rute yang tersesat atau terjebak. Karena algoritma ini melarang adanya arus balik atau backtrack, maka saat terjadi pencarian yang tersesat atau terjebak maka solusi tidak akan ditemukan. Hal ini sesuai dengan teori algoritma *Greedy Best First Search* yang menyatakan bahwa algoritma ini dapat mengalami permasalahan seperti *not complete*, *get stuck with local minima,* dan *Irrevocable*. Sehingga algoritma ini tidak dapat memberikan solusi yang paling optimal pada permasalahan ini.

Berdasarkan keseluruhan pengujian, berhasil didapatkan kesimpulan bahwa dalam permasalahan pencarian solusi *word ladder*, algoritma *Greedy Best First Search* merupakan algoritma yang belum tentu bisa memberikan solusi terbaik namun memiliki efisiensi pencarian yang baik (dari segi memory dan jumlah node yang dicari) namun memiliki waktu eksekusi yang baik, akibat dari pencarian yang selalu mencari yang terbaik dan mengabaikan yang lain. Algoritma UCS merupakan algoritma yang dapat memberikan solusi yang optimal namun efisiensi pencarian dari algoritma ini masih kurang baik, dapat dilihat pada penggunaan memori dan jumlah titik yang dievaluasi masih sangat besar dibandingkan

algoritma *Greedy Best First Search* dan algoritma A*. Hal tersebut mengakibatkan waktu eksekusi untuk algoritma UCS juga tergolong cukup lama dibandingkan dengan alogoritma lainnya. Algoritma A* merupakan algoritma yang berhasil memberikan solusi optimal dengan efisiensi pencarian yang baik juga, dapat dilihat dari penggunaan memori yan kecil, jumlah titik yang dievaluasi yang sedikit, dan waktu eksekusi yang tergolong cepat. Selain itu, algoritma pencarian A* juga memiliki heuristik yang *admissible* dalam permasalahan pencarian solusi *word ladder*. Sehinga, dapat disimpulkan bahwa algoritma pencarian terbaik dalam permasalahan pencarian solusi *word ladder* adalah algoritma A*.

LAMPIRAN

Pranala repository: [ValentinoTriadi/Tucil3_13522164 (github.com)]

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	✓	