LAPORAN TUGAS KECIL III

IF2211 STRATEGI ALGORITMA

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

Disusun oleh:

13522116 Naufal Adnan

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

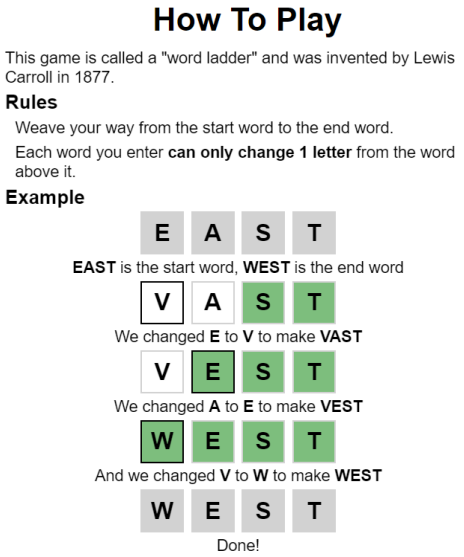
Institut Teknologi Bandung

2024

DAFTAR ISI

PENDAHULUAN

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 I
DASAR TEORI

Algoritma pencarian rute (*route planning*) merupakan algoritma yang digunakan untuk menemukan jalur terpendek atau optimal antara dua atau lebih titik dalam suatu jaringan. Jaringan tersebut biasanya digambarkan dalam bentuk suatu graf yang terhubung. Tujuannya adalah untuk menentukan jalur terbaik yang harus diambil dari titik awal ke titik tujuan, dengan mempertimbangkan berbagai faktor seperti jarak, waktu, biaya, atau kendala-kendala lainnya. Algoritma ini digunakan dalam berbagai aplikasi seperti navigasi GPS, logistik, pengiriman barang, pengiriman paket, robotika, dan pemetaan. Terdapat beberapa algoritma untuk pencarian rute di antaranya yang terkenal adalah Uniform Cost Search (UCS), Greedy Best First Search (GBFS), dan A-Star (A*).

1.1 *Uniform Cost Search* (UCS)

Uniform Cost Search (UCS) merupakan algoritma pencarian rute yang digunakan untuk menemukan jalur terpendek dengan mempertimbangkan biaya terendah dari suatu titik ke titik berikutnya, yang biasanya direpresentasikan dengan graf berbobot. Algoritma UCS hampir mirip dengan algoritma pencarian Breadth First Search (BFS) namun memiliki perbedaan. Algoritma BFS memperlakukan semua lintasan dengan biaya (*cost*) yang sama, sedangkan pada algoritma UCS untuk semua lintasan yang telah dieksplorasi akan dipertimbangkan besar biaya (*cost*) setiap simpul yang telah dibangkitkan untuk kemudian dipilih. *Cost* pada algoritma UCS biasanya ditentukan oleh biaya operasi terendah atau jarak terendah dari node awal ke node tetangganya ($g(n)$). Secara umum, langkah-langkah algoritma UCS adalah sebagai berikut.

1. Algoritma dimulai dengan menetapkan titik awal sebagai simpul awal. Biaya (cost) dari titik awal diatur menjadi 0, kemudian masukkan ke dalam *priority queue*.
2. Selama *priority queue* belum kosong, ambil simpul dengan biaya terendah.
3. Periksa simpul tersebut. Jika simpul merupakan simpul tujuan, maka kembalikan jalur yang sudah ditemukan.
4. Jika bukan, maka lakukan eksplorasi terhadap simpul saat ini untuk mendapatkan simpul tetangga dengan biaya yang sesuai.
5. Setelah mengekspansi simpul, algoritma memperbarui biaya dari setiap tetangga yang baru dieksplorasi. Biaya dari setiap tetangga dihitung dengan menambahkan biaya dari simpul induk ke biaya dari sisi yang menghubungkannya. Kemudian masukkan ke dalam *priority queue*.
6. Selama jalur belum ditemukan, ulangi kembali langkah kedua.
7. Algoritma berhenti ketika simpul tujuan telah ditemukan atau ketika himpunan simpul yang belum dieksplorasi kosong.

Kelebihan dari UCS adalah kemampuannya untuk menemukan jalur terpendek dalam graf berbobot dengan mempertimbangkan biaya dari setiap sisi secara individu. Namun, kelemahannya adalah algoritma ini bisa menjadi lambat jika biaya antara simpul-simpul yang

berdekatan sangat bervariasi, karena algoritma cenderung menjelajahi semua kemungkinan jalur dengan biaya yang semakin meningkat.

1.2 *Greedy Best First Search* (GBFS)

Greedy Best First Search (GBFS) adalah algoritma pencarian rute yang digunakan untuk menemukan jalur terpendek berdasarkan perhitungan heuristik tertentu. Hal ini berarti bahwa pada algoritma GBFS akan memilih simpul berikutnya untuk diekspansi berdasarkan perhitungan heuristik, yang umumnya dihitung berdasarkan jarak langsung ke tujuan ($f(n) = g(n)$). Algoritma GBFS mencoba untuk mencapai tujuan secepat mungkin dengan mengambil solusi terbaik pada setiap langkahnya (maksimum lokal) sehingga pada algoritma ini tidak melakukan backtracking. Tujuan dari algoritma GBFS ini adalah untuk memperluas pencarian ke arah yang kemungkinannya mendekati atau paling dekat dengan tujuan. Secara umum, algoritma GBFS memiliki langkah-langkah sebagai berikut.

1. Algoritma dimulai dengan menetapkan titik awal sebagai simpul awal dengan biaya sebesar biaya yang dihitung menggunakan fungsi *heuristic* yang digunakan.
2. Periksa simpul tersebut. Jika simpul merupakan simpul tujuan, maka kembalikan jalur yang sudah ditemukan.
3. Jika bukan, maka lakukan eksplorasi terhadap simpul saat ini untuk mendapatkan simpul tetangga dengan biaya yang sesuai.
4. Setelah mengekspansi simpul, algoritma memperbarui biaya dari setiap tetangga yang baru dieksplorasi. Biaya dari setiap tetangga dihitung dengan menghitung nilai *heuristic*.
5. Ambil simpul dengan nilai *heuristic* yang terbaik. Lalu ulangi kembali langkah kedua.
6. Algoritma berhenti ketika simpul tujuan telah ditemukan atau ketika simpul sudah tidak memiliki simpul tetangga untuk diekspansi.

Kelebihan dari GBFS adalah kemampuannya untuk menemukan jalur yang cepat dalam ruang pencarian yang besar. Akan tetapi, algoritma ini memiliki kelemahan yaitu tidak selalu dapat menemukan solusi jika tidak ada jalur yang langsung menuju ke tujuan. Hal ini berarti algoritma GBFS tidak dapat menjamin ditemukannya solusi jika jalur menuju tujuan tidak tersedia (*not complete*). GBFS cenderung dapat terjebak pada minimum lokal di sepanjang jalur pencarian. Hal ini terjadi karena algoritma hanya mempertimbangkan nilai heuristik lokal saat memilih simpul berikutnya untuk diekspansi, tanpa mempertimbangkan jalur keseluruhan. Keputusan yang dibuat oleh GBFS tidak dapat dibatalkan atau diubah. Setelah simpul diekspansi berdasarkan heuristik, tidak ada mekanisme untuk membatalkan atau mengubah keputusan tersebut (tidak ada *backtracking*).

1.3 A-Star (A*)

A* (A-star) adalah algoritma pencarian rute yang digunakan untuk menemukan jalur terpendek atau solusi optimal dalam graf berbobot dengan menggabungkan keunggulan dari algoritma pencarian UCS dan GBFS. Konsep dasar dari algoritma A* adalah untuk menghindari memperluas jalur yang mahal atau jalur yang biayanya sudah melebihi biaya

jalur terpendek yang sudah ditemukan. Untuk menentukan biaya, algoritma A* menggunakan fungsi evaluasi *f(n) = g(n) + h(n)*, di mana:

- *g(n)* adalah biaya yang sudah dikeluarkan untuk mencapai simpul n.
- *h(n)* adalah perkiraan biaya dari simpul n ke tujuan akhir.
- *f(n)* adalah perkiraan biaya total dari simpul awal melalui simpul n ke tujuan akhir.

Dengan menggunakan fungsi evaluasi ini, algoritma A* dapat memilih simpul untuk diekspansi berdasarkan kombinasi biaya yang sudah dikeluarkan (g(n)) dan estimasi biaya dari simpul saat ini ke tujuan akhir *(h(n))* sehingga akan efisien untuk ruang pencarian yang besar. Algoritma A* pada dasarnya akan menemukan solusi optimal selama *heuristic* yang dipakai *admissible*, yaitu nilai heuristik yang dipakai tidak *overestimate* dari nilai yang seharusnya. Secara umum algoritma A* memiliki langkah-langkah sebagai berikut.

1. Algoritma dimulai dengan menetapkan titik awal sebagai simpul awal dengan memberikan biaya (*cost*) menggunakan fungsi evaluasi *f(n)*, kemudian masukkan ke dalam *priority queue*.
2. Selama *priority queue* belum kosong, ambil simpul dengan nilai *f(n)* terendah.
3. Periksa simpul tersebut. Jika simpul merupakan simpul tujuan, maka kembalikan jalur yang sudah ditemukan.
4. Jika bukan, maka lakukan eksplorasi terhadap simpul saat ini untuk mendapatkan simpul tetangga dengan biaya yang sesuai.
5. Setelah mengekspansi simpul, algoritma memperbarui biaya dari setiap tetangga yang baru dieksplorasi. Biaya dari setiap tetangga dihitung dengan menggunakan fungsi evaluasi *f(n)*. Kemudian masukkan ke dalam *priority queue*.
6. Selama jalur belum ditemukan, ulangi kembali langkah kedua.
7. Algoritma berhenti ketika simpul tujuan telah ditemukan atau ketika himpunan simpul yang belum dieksplorasi kosong.

BAB II
IMPLEMENTASI

Persoalan permainan *Word Ladder* merupakan persoalan pencarian rute (*route planning*) untuk menemukan jalur terpendek atau optimal antara dua kata, dari kata awal menuju kata tujuan dengan mengganti sebuah karakter di setiap langkahnya. Karakter yang diganti harus tetap mempertahankan kata tersebut sehingga kata masih tetap terdapat di sebuah kamus (*dictionary*). Terdapat beberapa algoritma untuk pencarian rute pada permainan *Word Ladder* ini, di antaranya adalah Uniform Cost Search (UCS), Greedy Best First Search (GBFS), dan A-Star (A*).

2.1 Uniform Cost Search (UCS)

Pada implementasi algoritma UCS untuk permainan *Word Ladder* memiliki langkah-langkah sebagai berikut.

1. Algoritma dimulai dengan menetapkan kata awal sebagai simpul awal. Biaya (cost) dari simpul awal diatur menjadi 0, kemudian dimasukkan ke dalam *priority queue*.
2. Selama *priority queue* belum kosong, ambil simpul dengan biaya terendah. Jika terdapat dua atau lebih simpul memiliki biaya yang sama, maka akan dipilih indeks paling kecil yang diubah karakternya dengan urutan *leksikografis*. Misalkan dari ‘ABET’ maka akan lebih dipilih ‘ABED’ terlebih dahulu dibanding ‘ABET’, dan jika dari ‘DREE’ maka akan lebih dipilih ‘BREE’ terlebih dahulu dibanding ‘DREG’.
3. Tandai simpul sebagai telah dikunjungi lalu periksa simpul tersebut. Jika simpul merupakan simpul tujuan atau dalam hal ini kata tujuan, maka kembalikan jalur yang sudah ditemukan.
4. Jika bukan, maka lakukan eksplorasi terhadap simpul saat ini untuk mendapatkan simpul tetangga dengan biaya yang sesuai. Simpul diekspansi dengan melakukan penggantian satu karakter pada simpul saat ini dengan karakter dari ‘A’ hingga ‘Z’. Jika penggantian karakter membuat kata menjadi tidak terdapat di kamus (*dictionary*), maka abaikan kata tersebut.
5. Setelah mengekspansi simpul, algoritma memperbarui biaya dari setiap tetangga yang baru dieksplorasi. Biaya dari setiap tetangga dihitung dengan menambahkan biaya dari simpul induk ke biaya dari sisi yang menghubungkannya. Dalam hal ini biaya suatu simpul didefinisikan sebagai banyak langkah perubahan yang telah dilakukan (*depth*).
6. Jika simpul tetangga tersebut belum dikunjungi, maka simpul tetangga akan dimasukkan ke dalam *priority queue*. Jika sudah dikunjungi, maka abaikan.
7. Selama jalur belum ditemukan, ulangi kembali langkah kedua.
8. Algoritma berhenti ketika simpul tujuan telah ditemukan atau ketika himpunan simpul yang belum dieksplorasi kosong.

Algoritma UCS ini akan selalu complete selama memang ada solusinya karena dilakukan *backtracking* pada proses algoritma ini. Ekspansi yang dilakukan dengan menghitung biaya

berdasarkan kedalaman atau jarak langkah dari simpul induk membuat algoritma ini mirip seperti algoritma Breadth First Search namun dengan urutan leksikografis.

1.2 *Greedy Best First Search* (GBFS)

Pada implementasi algoritma GBFS untuk permainan *Word Ladder* memiliki langkah-langkah sebagai berikut.

1. Algoritma dimulai dengan menetapkan titik awal atau kata awal sebagai simpul awal. Biaya (cost) dari simpul awal diatur dengan menghitung nilai *heuristic* simpul tersebut ke simpul tujuan dengan menghitung berapa jumlah karakter yang berbeda pada simpul awal dengan kata tujuan.
2. Tandai simpul sebagai telah dikunjungi lalu periksa simpul tersebut. Jika simpul merupakan simpul tujuan atau dalam hal ini kata tujuan, maka kembalikan jalur yang sudah ditemukan.
3. Jika bukan, maka lakukan eksplorasi terhadap simpul saat ini untuk mendapatkan simpul tetangga dengan biaya yang sesuai. Simpul diekspansi dengan melakukan penggantian satu karakter pada simpul saat ini dengan karakter dari ‘A’ hingga ‘Z’. Jika penggantian karakter membuat kata menjadi tidak terdapat di kamus (*dictionary*), maka abaikan kata tersebut.
4. Setelah mengekspansi simpul, algoritma memperbarui biaya dari setiap tetangga yang baru dieksplorasi. Biaya dari setiap tetangga dihitung dengan menghitung nilai *heuristic* simpul tersebut ke simpul tujuan dengan menghitung berapa jumlah karakter yang berbeda pada simpul ekspansi dengan kata tujuan.
5. Ambil simpul dengan nilai *heuristic* paling kecil atau biaya operasi yang paling kecil.
6. Jika terdapat dua atau lebih simpul memiliki biaya yang sama, maka akan dipilih indeks paling kecil yang diubah karakternya dengan urutan *leksikografis.* Misalkan dari ‘ABET’ maka akan lebih dipilih ‘ABED’ terlebih dahulu dibanding ‘ABET’, dan jika dari ‘DREE’ maka akan lebih dipilih ‘BREE’ terlebih dahulu dibanding ‘DREG’.
7. Pilih simpul tersebut lalu ulangi kembali langkah kedua.
8. Algoritma berhenti ketika simpul tujuan telah ditemukan atau ketika simpul sudah tidak memiliki simpul tetangga untuk diekspansi.

Algoritma GBFS ini tidak akan menjamin *complete* karena tidak dilakukan *backtracking* pada proses algoritma ini. Biaya yang diperoleh dengan menghitung nilai *heuristic* simpul tersebut ke simpul tujuan dengan menghitung berapa jumlah karakter yang berbeda pada simpul awal dengan kata tujuan dapat membuat pencarian solusi terjebak pada maksimum lokal.

1.3 A-Star (A*)

Pada implementasi algoritma A* untuk permainan *Word Ladder* biaya dihitung dengan menggabungkan biaya pada algoritma UCS dan GBFS. Dapat dikatakan bahwa implementasi algoritma ini memiliki fungsi perhitungan biaya *f(n)* sebagai berikut.

f(n) = g(n) + h(n), di mana:

- *g(n)*: banyak langkah perubahan yang telah dilakukan (*depth*).

- *h(n)*: berapa jumlah karakter yang berbeda pada simpul ekspansi dengan kata tujuan
- *f(n)* adalah perkiraan biaya total dari simpul awal melalui simpul n ke tujuan akhir.

Pada implementasi algoritma A* ini mirip dengan UCS namun dengan perhitungan biaya yang berbeda. Secara matematis algoritma A* untuk permainan *Word Ladder* memiliki langkah-langkah sebagai berikut.

1. Algoritma dimulai dengan menetapkan kata awal sebagai simpul awal. Biaya (cost) dari simpul awal diatur menggunakan perhitungan *f(n)* seperti pada penjelasan di atas, kemudian dimasukkan ke dalam *priority queue*.
2. Selama *priority queue* belum kosong, ambil simpul dengan biaya terendah. Jika terdapat dua atau lebih simpul memiliki biaya yang sama, maka akan dipilih indeks paling kecil yang diubah karakternya dengan urutan *leksikografis*. Misalkan dari ‘ABET’ maka akan lebih dipilih ‘ABED’ terlebih dahulu dibanding ‘ABET’, dan jika dari ‘DREE’ maka akan lebih dipilih ‘BREE’ terlebih dahulu dibanding ‘DREG’.
3. Tandai simpul sebagai telah dikunjungi lalu periksa simpul tersebut. Jika simpul merupakan simpul tujuan atau dalam hal ini kata tujuan, maka kembalikan jalur yang sudah ditemukan.
4. Jika bukan, maka lakukan eksplorasi terhadap simpul saat ini untuk mendapatkan simpul tetangga dengan biaya yang sesuai. Simpul diekspansi dengan melakukan penggantian satu karakter pada simpul saat ini dengan karakter dari ‘A’ hingga ‘Z’. Jika penggantian karakter membuat kata menjadi tidak terdapat di kamus (*dictionary*), maka abaikan kata tersebut.
5. Setelah mengekspansi simpul, algoritma memperbarui biaya dari setiap tetangga yang baru dieksplorasi. Biaya dari setiap tetangga dihitung dengan menggunakan perhitungan *f(n)* seperti pada penjelasan di atas.
6. Jika simpul tetangga tersebut belum dikunjungi, maka simpul tetangga akan dimasukkan ke dalam *priority queue*. Jika sudah dikunjungi, maka abaikan.
7. Selama jalur belum ditemukan, ulangi kembali langkah kedua.
8. Algoritma berhenti ketika simpul tujuan telah ditemukan atau ketika himpunan simpul yang belum dieksplorasi kosong.

Algoritma A* ini akan selalu complete selama memang ada solusinya karena dilakukan *backtracking* pada proses algoritma ini. Algoritma ini juga akan efisien untuk ruang pencarian yang besar karena fungsi evaluasi ini pada algoritma A* memilih simpul untuk diekspansi berdasarkan kombinasi biaya yang sudah dikeluarkan (g(n)) dan estimasi biaya dari simpul saat ini ke tujuan akhir *(h(n))*. Algoritma A* pada dasarnya akan menemukan solusi optimal selama *heuristic* yang dipakai *admissible*, yaitu nilai *heuristik* yang dipakai tidak *overestimate* dari nilai yang seharusnya.

BAB III
SOURCE CODE

3.1 Bahasa dan Library

Program diimplementasikan dalam bahasa Java dengan menggunakan library java.Util.*, java.io.*, dan java.swing.* serta java.awt.* untuk implementasi GUI.

3.2 Struktur Program

Struktur program dibuat sedemikian rupa terdiri dari 4 folder utama. Folder bin berisi semua java class *file*, folder doc berisi dokumentasi laporan, folder src berisi *source code*, folder test berisi folder CLI untuk testing yang dilakukan dengan CLI dan folder GUI untuk testing yang dilakukan dengan GUI. File dictionary.txt berfungsi sebagai file kamus yang akan dimuat sebagai rujukan utama kamus. File ini dapat diganti dengan kamus yang diinginkan oleh pengguna namun dengan memperhatikan bahwa penamaan kamus baru haruslah bernama 'dictionary.txt' dan berada pada root directory yang sama dengan directory sebelumnya. File runCLI.bat dan runGUI.bat digunakan untuk melakukan kompilasi dan menjalankan program dalam OS Windows. Sementara file runCLI.sh dan runGUI.sh digunakan untuk melakukan kompilasi dan menjalankan program dalam OS Linux. File README.md berisi uraian tentang isi dari *repository*.

3.3 Implementasi Algoritma

3.3.1 File Dictionary.java

File Dictionary.java ini berisi sebuah class Dictioinary yang memiliki tanggung jawab untuk melakukan pemuatan kamus dari file dictionary.txt dan menempatkannya dalam sebuah HashSet untuk mempercepat pencarian. Berikut *source code* dari file Dictionary.java.

Dictionary.java

```java
package lib;

import java.io.*;
import java.util.*;

public class Dictionary {
    private HashSet<String> dictionary = new HashSet<>();

    public Dictionary(String fileName) {
        try {
            File file = new File(fileName);
            Scanner scanner = new Scanner(file);
            while (scanner.hasNextLine()) {
                String line = scanner.nextLine().trim();
                dictionary.add(line.toUpperCase());
            }
            scanner.close();
        } catch (FileNotFoundException e) {
            System.err.println("Error filename: " + e.getMessage());
            System.exit(status:1);
        }
    }

    public HashSet<String> getDictionary() {
        return dictionary;
    }
}
```

3.3.2 File Node.java

File Node.java ini berisi sebuah class Node yang memiliki tanggung jawab untuk membangun sebuah simpul dengan propertinya seperti kedalaman, biaya, kata, dan tetangga. Pada class ini juga memiliki tanggung jawab untuk membangun path dari suatu node menuju node akar dan equalWord untuk membandingkan dua buah string. Berikut *source code* dari file Node.java.

Node.java

```java
package lib;

import java.util.*;

public class Node {
    private int     depth;
    private int     cost;
    private String  word;
    private Node    prev;

    public Node(int depth, int cost, String word, Node prev) {
        setDepth(depth);
        setCost(cost);
        setWord(word);
        setPrev(prev);
    }

    // getter
    public int getDepth()   {return depth;}
    public int getCost()    {return cost;}
    public String getWord() {return word;}
    public Node getPrev()   {return prev;}

    // setter
    public void setDepth(int depth)     {this.depth = depth;}
    public void setCost(int cost)       {this.cost  = cost;}
    public void setWord(String word)    {this.word  = word;}
    public void setPrev(Node Prev)      {this.prev  = Prev;}

    public boolean equalWord(String word) {
        return getWord().equals(word);
    }

    public List<String> buildPath() {
        List<String> path = new ArrayList<>();
        Node currentNode = this;
        while (currentNode != null) {
            path.add(index:0, currentNode.getWord());
            currentNode = currentNode.getPrev();
        }
        return path;
    }
}
```

3.3.3 File NComparator.java

File NComparator.java ini berisi sebuah class NComparator yang memiliki tanggung jawab untuk membandingkan dua buah node berdasarkan urutan leksikografisnya. Berikut *source code* dari file Node.java.

NComparator.java

```java
package lib;

import java.util.*;

public class NComparator implements Comparator<Node> {
    @Override
    public int compare(Node a, Node b) {
        int cost = Integer.compare(a.getCost(), b.getCost());
        if (cost != 0) {
            return cost;
        } else {
            return a.getWord().compareTo(b.getWord());
        }
    }
}
```

3.3.4 File Util.java

File Util.java ini berisi sebuah class Util yang memiliki tanggung jawab untuk menghitung nilai *heuristic*. Berikut *source code* dari file Util.java.

Util.java

```java
package lib;

public class Util {
    public static int heuristicCost(String start, String end) {
        int cost = 0;
        for (int i = 0; i < start.length(); i++) {
            if (start.charAt(i) != end.charAt(i))
                cost ++;
        }
        return cost;
    }
}
```

3.3.5 File IWordLadder.java

File IWordLadder.java ini berisi sebuah interface IWordLadder yang memiliki sebuah method solver untuk selanjutnya diimplementasikan oleh ketiga algoritma, yaitu UCS, GBFS, dan A*. Berikut *source code* dari file IWordLadder.java.

Util.java

```java
package wordladder;

import java.util.*;

public interface IWordLadder {
    WordLadder solver(String start, String end, HashSet<String> dictionary);
}
```

3.3.6 File WordLadder.java

File WordLadder.java ini berisi sebuah class WordLadder yang memiliki sebuah tanggung jawab sebagai tipe class yang akan dikembalikan oleh masing-masing algoritma pencarian rute. Berikut *source code* dari file WordLadder.java.

Util.java

```java
package wordladder;

import java.util.*;

public class WordLadder {
    private List<String> path;
    private int nodesVisited;

    public WordLadder(List<String> path, int nodesVisited) {
        this.path = path;
        this.nodesVisited = nodesVisited;
    }

    public void setPathError(String error) {
        if (this.path == null) {
            this.path = new ArrayList<>();
        }
        this.path.add(error);
    }

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

    public int getNodesVisited() {
        return nodesVisited;
    }
}
```

3.3.7 File UCS.java

File UCS.java ini berisi sebuah class UCS yang mengimplementasikan interface IWordLadder dan memiliki sebuah method solver untuk menemukan solusi permainan *Word Ladder* dengan pencarian rute menggunakan algoritma UCS. Berikut *source code* dari file UCS.java.

UCS.java

```java
package wordladder;

import java.util.*;
import lib.*;

public class UCS implements IWordLadder {
    @Override
    public WordLadder solver(String start, String end, HashSet<String> dictionary) {
        HashSet<String> visited = new HashSet<>();
        PriorityQueue<Node> priorityQueue = new PriorityQueue<>(new lib.NComparator());
        Node currentNode = new Node(depth:0, Util.heuristicCost(start, end), start, prev:null);

        priorityQueue.offer(currentNode);
        visited.add(currentNode.getWord());

        int check = 0;
        while (!priorityQueue.isEmpty()) {
            currentNode = priorityQueue.poll();
            String word = currentNode.getWord();

            check++;
            if (currentNode.equalWord(end))
                return new WordLadder(currentNode.buildPath(), check);

            for (int i = 0; i < word.length(); i++) {
                for (char character = 'A'; character <= 'Z'; character++) {
                    if (word.charAt(i) != character) {
                        StringBuilder str = new StringBuilder(word);
                        str.setCharAt(i, character);
                        String tempWord = str.toString();

                        if (dictionary.contains(tempWord)) {
                            if (!visited.contains(tempWord)) {
                                int depth = currentNode.getDepth() + 1;
                                Node newNode = new Node(depth, depth, tempWord, currentNode);
                                priorityQueue.offer(newNode);
                                visited.add(tempWord);
                            }
                        }
                    }
                }
            }
        }

        return new WordLadder(path:null, check);
    }
}
```

3.3.8 File GBFS.java

File GBFS.java ini berisi sebuah class GBFS yang mengimplementasikan interface IWordLadder dan memiliki sebuah method solver untuk menemukan solusi permainan *Word Ladder* dengan pencarian rute menggunakan algoritma GBFS. Berikut *source code* dari file GBFS.java.

GBFS.java

```java
package wordladder;

import java.util.*;
import lib.*;

public class GBFS implements IWordLadder {
    @Override
    public WordLadder solver(String start, String end, HashSet<String> dictionary) {
        HashSet<String> visited = new HashSet<>();
        Node currentNode = new Node(depth:0, Util.heuristicCost(start, end), start, prev:null);
        int check = 0;
        while (true) {
            String word = currentNode.getWord();
            visited.add(word);
            check++;
            if (currentNode.equalWord(end))
                return new WordLadder(currentNode.buildPath(), check);

            int newCost = Integer.MAX_VALUE;
            String newWord = "";
            for (int i = 0; i < word.length(); i++) {
                for (char character = 'A'; character <= 'Z'; character++) {
                    if (word.charAt(i) != character) {
                        StringBuilder str = new StringBuilder(word);
                        str.setCharAt(i, character);
                        String tempWord = str.toString();

                        if (dictionary.contains(tempWord) && !visited.contains(tempWord)) {
                            int cost = Util.heuristicCost(tempWord, end);
                            if (cost < newCost) {
                                newCost = cost;
                                newWord = tempWord;
                            }
                        }
                    }
                }
            }

            if (newWord.equals(anObject:""))
                return new WordLadder(path:null, check);
            currentNode = new Node(currentNode.getDepth() + 1, newCost, newWord, currentNode);
        }
    }
}
```

3.3.9 File AStar.java

File AStar.java ini berisi sebuah class AStar yang mengimplementasikan interface IWordLadder dan memiliki sebuah method solver untuk menemukan solusi permainan *Word Ladder* dengan pencarian rute menggunakan algoritma AStar. Berikut *source code* dari file AStar.java.

AStar.java

```java
package wordladder;

import java.util.*;
import lib.*;

public class AStar implements IWordLadder {
    @Override
    public WordLadder solver(String start, String end, HashSet<String> dictionary) {
        HashSet<String> visited = new HashSet<>();
        PriorityQueue<Node> priorityQueue = new PriorityQueue<>(new lib.NComparator());
        Node currentNode = new Node(depth:0, Util.heuristicCost(start, end), start, prev:null);

        priorityQueue.offer(currentNode);
        visited.add(currentNode.getWord());

        int check = 0;
        while (!priorityQueue.isEmpty()) {
            currentNode = priorityQueue.poll();
            String word = currentNode.getWord();

            check++;
            if (currentNode.equalWord(end))
                return new WordLadder(currentNode.buildPath(), check);

            for (int i = 0; i < word.length(); i++) {
                for (char character = 'A'; character <= 'Z'; character++) {
                    if (word.charAt(i) != character) {
                        StringBuilder str = new StringBuilder(word);
                        str.setCharAt(i, character);
                        String tempWord = str.toString();

                        if (dictionary.contains(tempWord)) {
                            if (!visited.contains(tempWord)) {
                                int depth = currentNode.getDepth() + 1;
                                int cost = Util.heuristicCost(tempWord, end) + depth;
                                Node newNode = new Node(depth, cost, tempWord, currentNode);
                                priorityQueue.offer(newNode);
                                visited.add(tempWord);
                            }
                        }
                    }
                }
            }
        }

        return new WordLadder(path:null, check);
    }
}
```

3.3.10 File UI.java

File UI.java ini berisi sebuah class UI yang berisikan method-method static untuk keperluan tampilan pada CLI. Source code untuk file ini terlalu panjang dan dapat dilihat pada pranala github yang terdapat pada lampiran.

3.3.11 File CLI/Main.java

File Main.java ini berisi sebuah class Main yang berisikan program utama dari permainan *Word Ladder* ini untuk tampilan pada CLI.

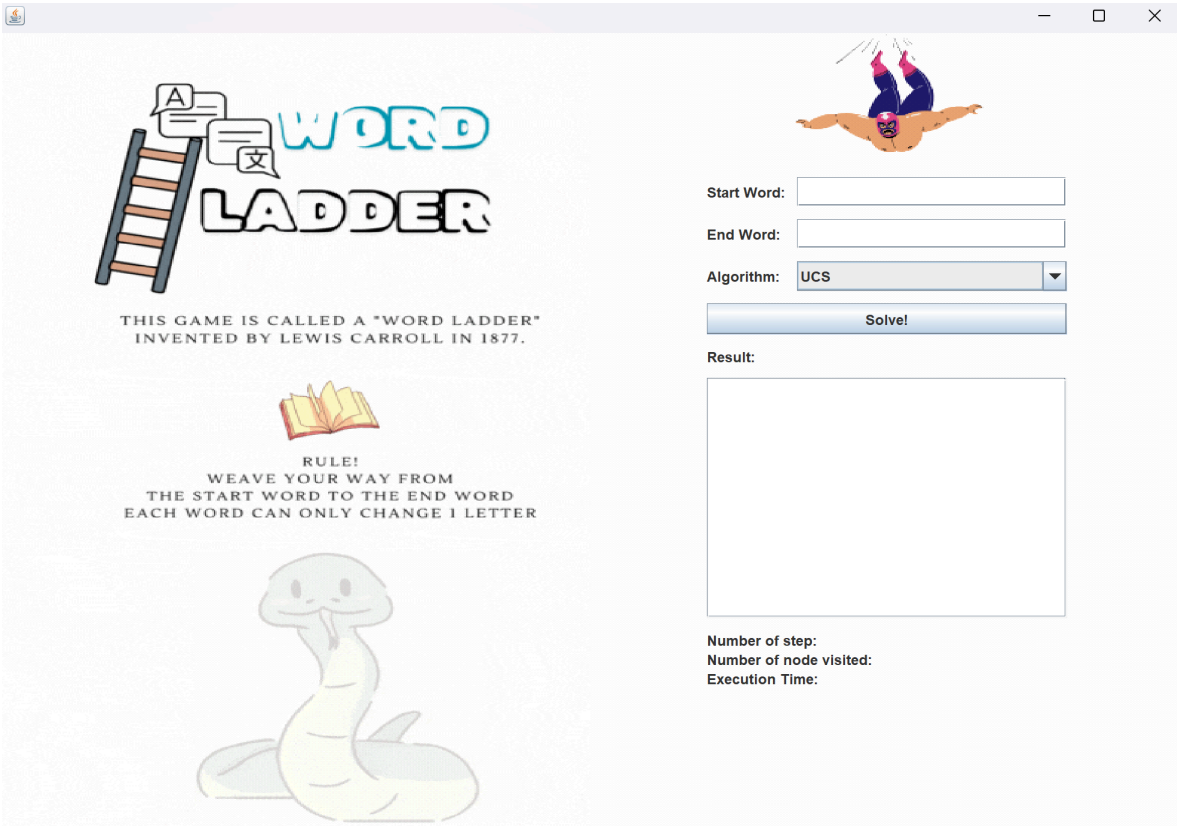
Main.java

```java
package CLI;

import java.io.IOException;
import java.util.*;
import wordladder.AStar;
import wordladder.GBFS;
import wordladder.UCS;
import wordladder.WordLadder;

public class Main {
    Run | Debug
    public static void main(String[] args) throws InterruptedException, IOException {
        boolean run = true;
        UI.startingLoading();
        lib.Dictionary Load = new lib.Dictionary(fileName:"dictionary.txt");
        HashSet<String> dictionary = Load.getDictionary();
        while (run) {
            UI.clearScreen();
            UI.printOpening();
            String start = UI.inputStartWord();
            while (!dictionary.contains(start)) {
                if (start.length() == 0) {
                    System.out.println(UI.ANSI_RED + "The Start Word cannot be blank!" + UI.ANSI_RESET);
                } else {
                    System.out.println(
                            UI.ANSI_RED + "The word " + start + " is not in our English dictionary!" + UI.ANSI_RESET);
                }
                start = UI.inputStartWord().toUpperCase();
            }

            String end = UI.inputEndWord();
            while (!dictionary.contains(end) || end.length() != start.length()) {
                if (end.length() == 0) {
                    System.out.println(UI.ANSI_RED + "The End Word cannot be blank!" + UI.ANSI_RESET);
                } else if (!dictionary.contains(end)) {
                    System.out.println(
                            UI.ANSI_RED + "The word " + end + " is not in our English dictionary!" + UI.ANSI_RESET);
                } else {
                    System.out.println(
                            UI.ANSI_RED + "The length of the start and end word doesn't match!" + UI.ANSI_RESET);
                }
                end = UI.inputEndWord().toUpperCase();
            }
            System.out.println("\nStart from " + UI.ANSI_RED + start + UI.ANSI_RESET + " To " + UI.ANSI_GREEN + end
                    + UI.ANSI_RESET + "\n");

            int choice = 0;
            boolean isValidInput = false;
            while (!isValidInput) {
                UI.choiceAlgorithm();
                if (UI.scan.hasNextInt()) {
                    choice = UI.scan.nextInt();
                    if (choice >= 1 && choice <= 3) {
                        isValidInput = true;
                    } else {
                        System.out.println(UI.ANSI_RED + "\nInvalid input!\nPlease enter a number between 1 and 3!\n"
                                + UI.ANSI_RESET);
                    }
```

```
                } else {
                    System.out
                            .println(UI.ANSI_RED + "\nInvalid input!\nPlease enter a valid integer!\n" + UI.ANSI_RESET);
                    UI.scan.next();
                }
            }

            UI.clearScreen();
            System.out.println("\nStart from " + UI.ANSI_RED + start + UI.ANSI_RESET + " To " + UI.ANSI_GREEN + end
                    + UI.ANSI_RESET + "\n");
            System.out.print(UI.ANSI_YELLOW + "Using ");

            WordLadder result = new WordLadder(path:null, nodesVisited:0);
            long startTime = System.nanoTime();
            switch (choice) {
                case 1:
                    System.out.println("Uniform Cost Search\n" + UI.ANSI_RESET);
                    UCS ucsSolver = new UCS();
                    result = ucsSolver.solver(start, end, dictionary);
                    break;

                case 2:
                    System.out.println("Greedy Best First Search\n" + UI.ANSI_RESET);
                    GBFS gbfsSolver = new GBFS();
                    result = gbfsSolver.solver(start, end, dictionary);
                    break;

                case 3:
                    System.out.println("A*\n" + UI.ANSI_RESET);
                    AStar aStarSolver = new AStar();
                    result = aStarSolver.solver(start, end, dictionary);
                    break;

                default:
                    break;
            }
            long endTime = System.nanoTime();
            double totalTime = (endTime - startTime) / 1e6;

            System.out.println(x:"Path: ");
            if (result.getPath() == null) {
                System.out.println(UI.ANSI_RED + "\nPath Not Found!" + UI.ANSI_RESET);
                System.out.println(x:"\nNumber of step: 0");
            } else {
                for (int i = 0; i < result.getPath().size(); i++) {
                    String str = UI.colorMatchingCharacters(result.getPath().get(i), end);
                    System.out.println((i + 1) + ". " + str);
                }
                System.out.println("\nNumber of step: " + (result.getPath().size() - 1));
            }
            System.out.println("Number of nodes visited: " + result.getNodesVisited());
            System.out.println("Time Execution: " + totalTime + " ms");
            System.out.print(UI.ANSI_YELLOW + "\n>> Press Enter key to continue..." + UI.ANSI_RESET);
            UI.scan.nextLine();
            UI.scan.nextLine();
        }
    }
}
```

BAB IV
EKSPERIMEN

Tampilan Utama GUI

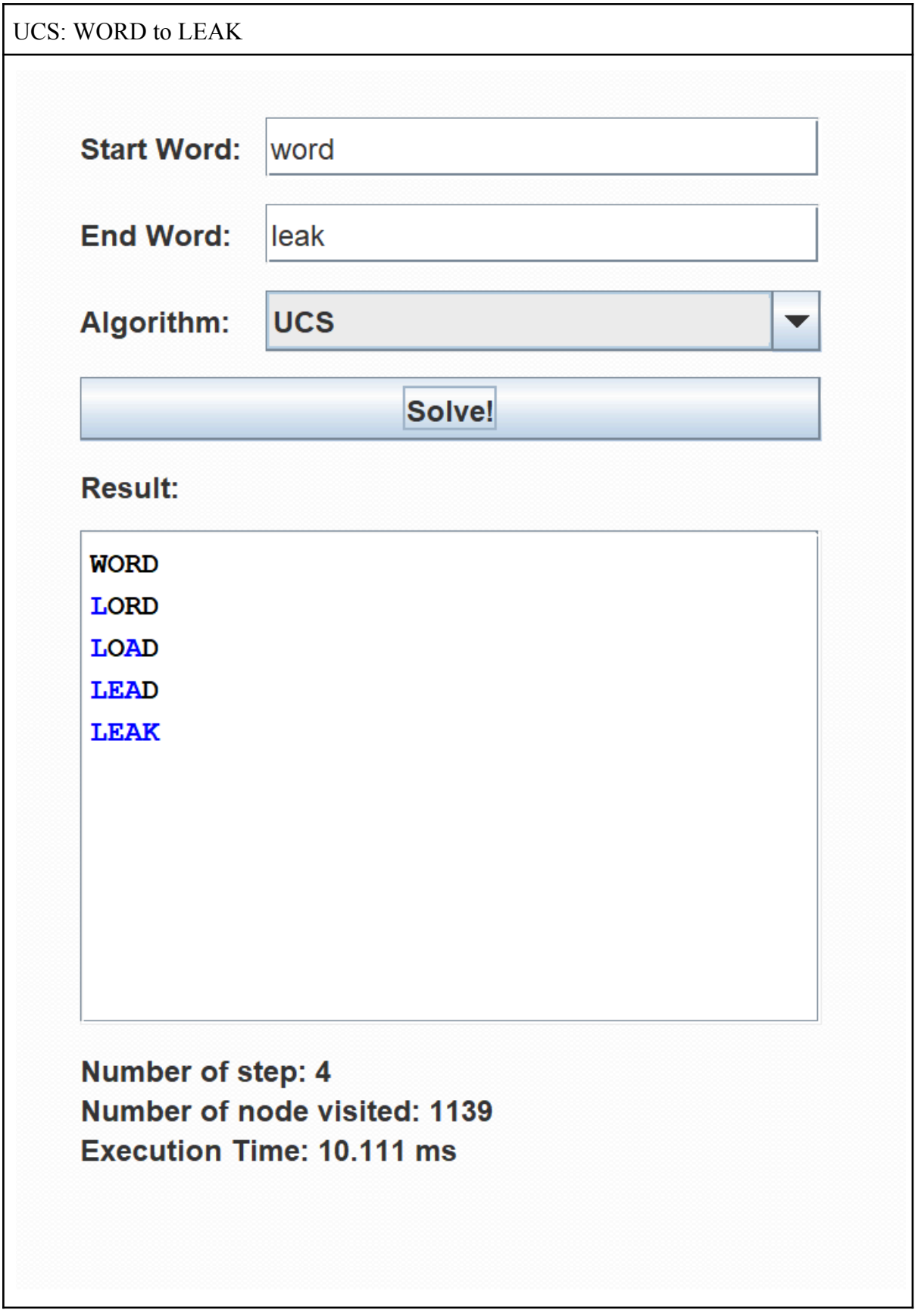


Tampilan Utama CLI

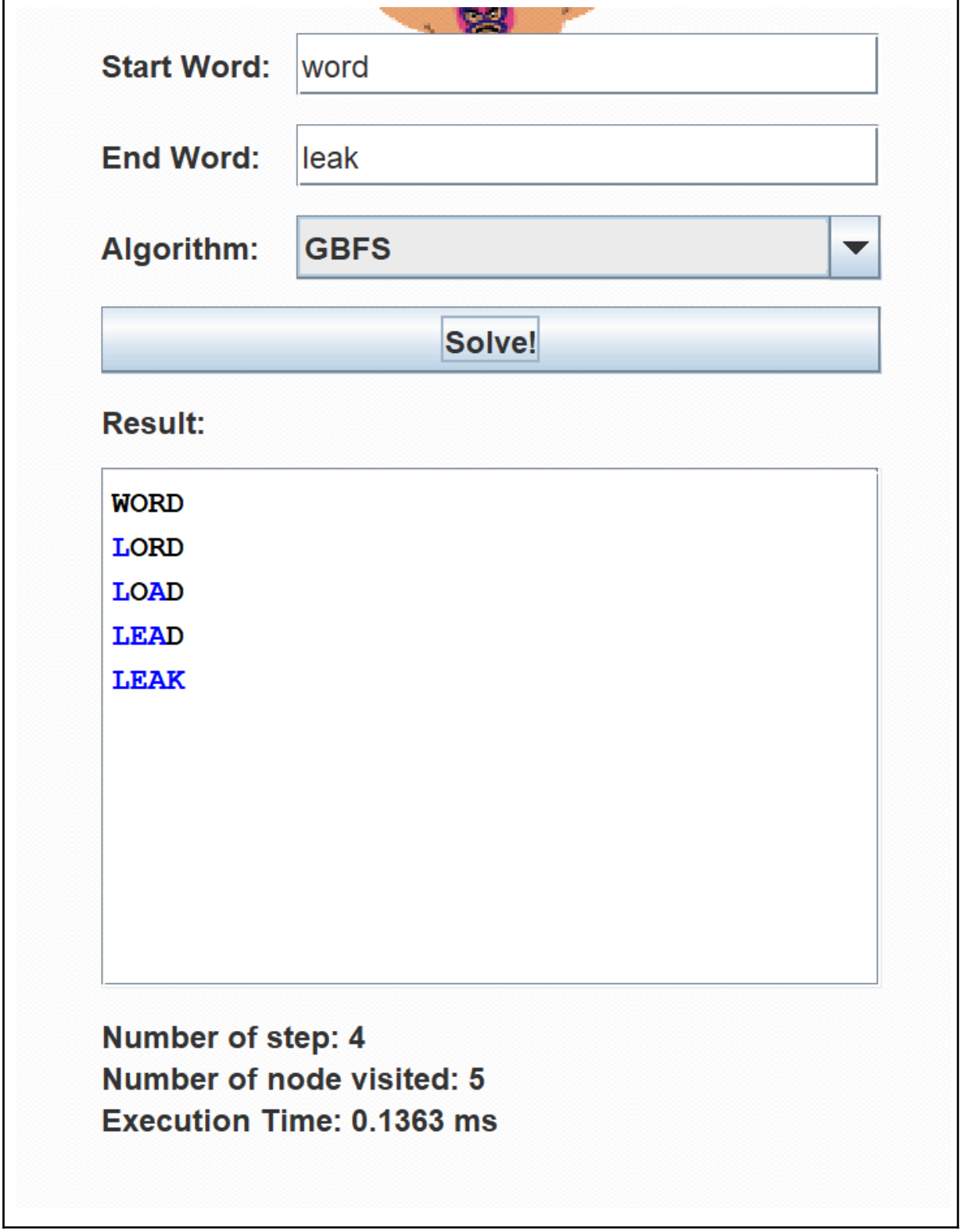
Pada testing dan eksperimen ini selanjutnya hanya akan ditunjukkan untuk tampilan GUI saja untuk menghemat tulisan. Percobaan tetap dilakukan di GUI dan CLI.

4.1 Test 1

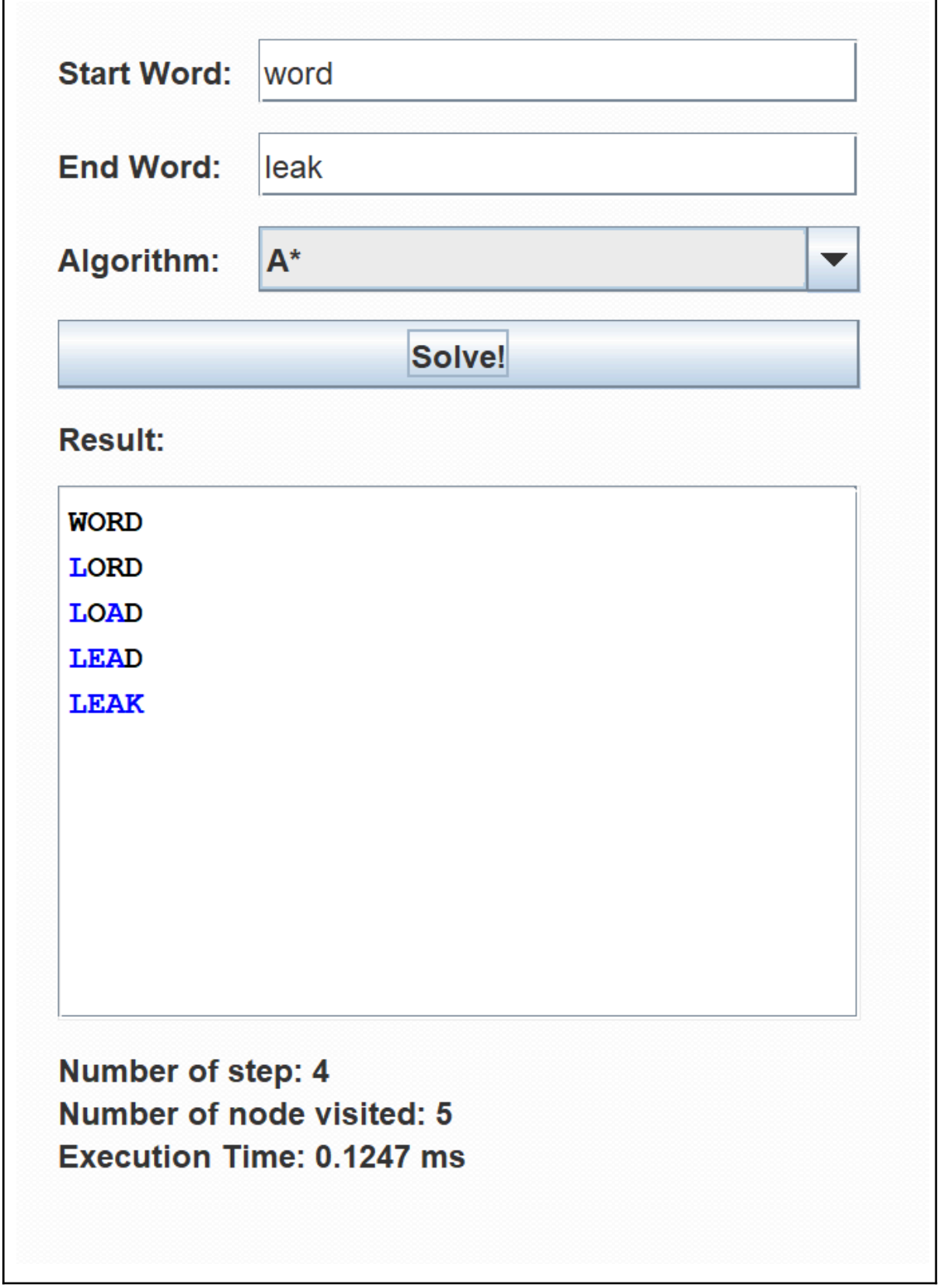
UCS: WORD to LEAK



GBFS: WORD to LEAK

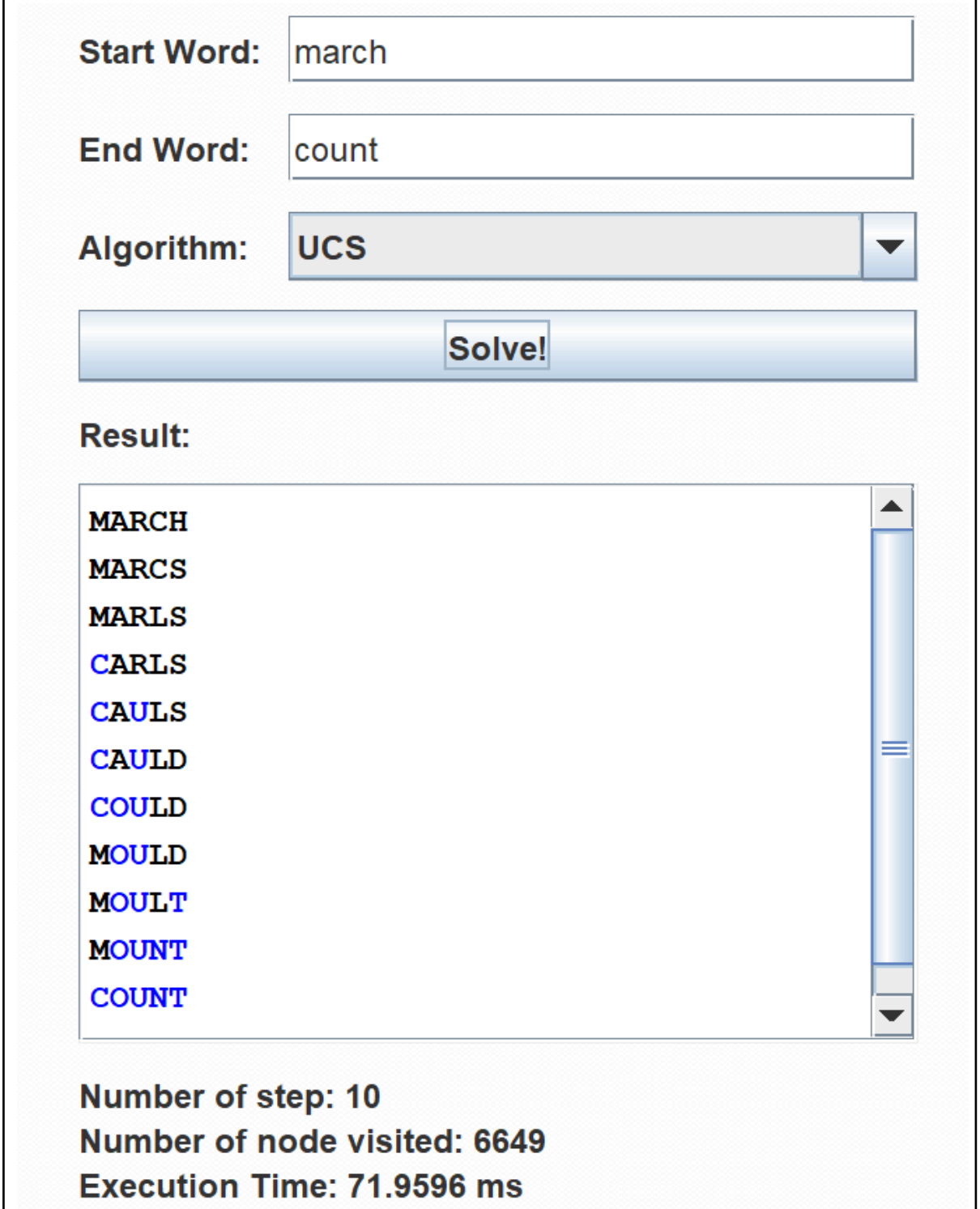


A*: WORD to LEAK

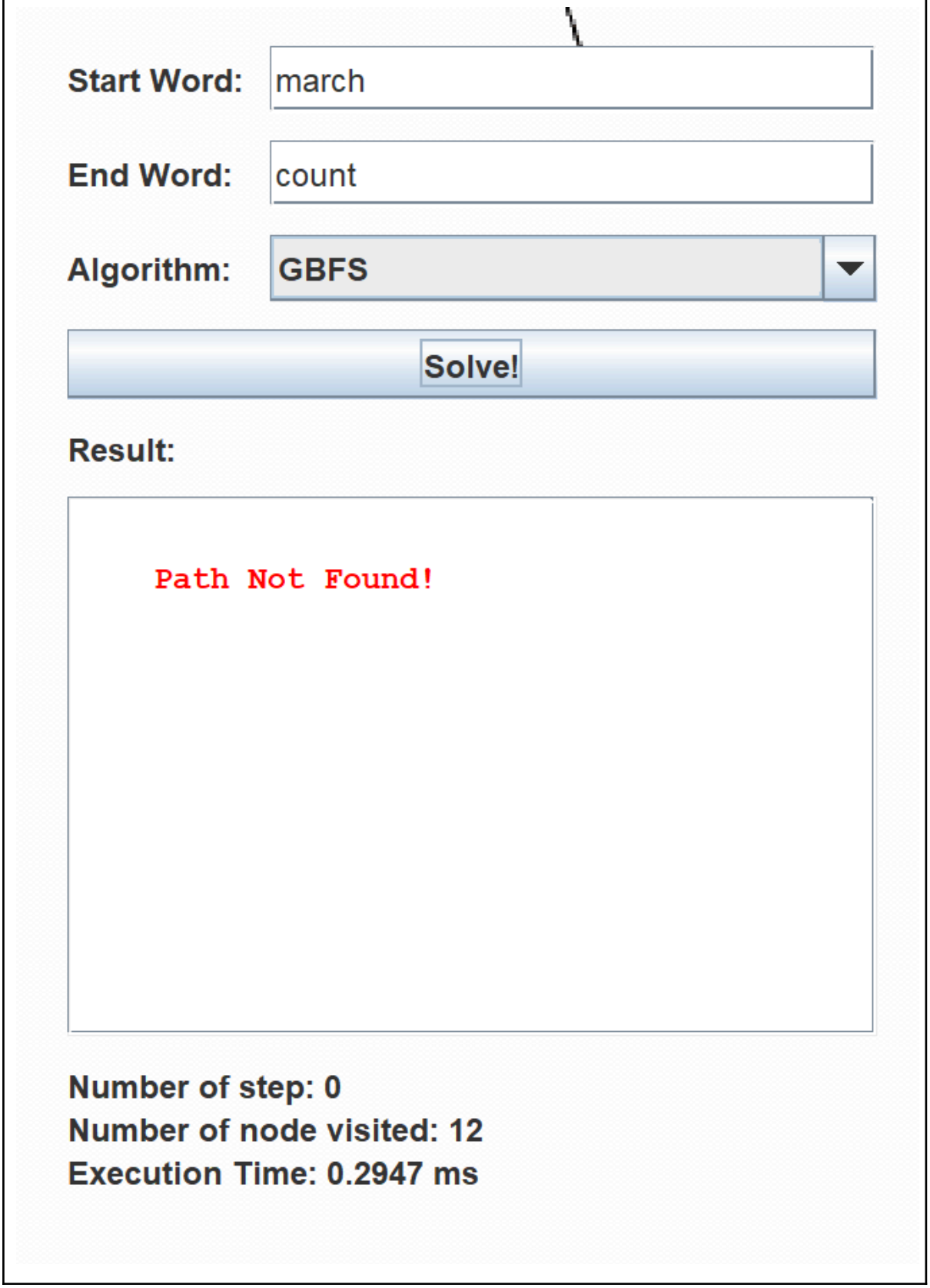


4.2 Test 2

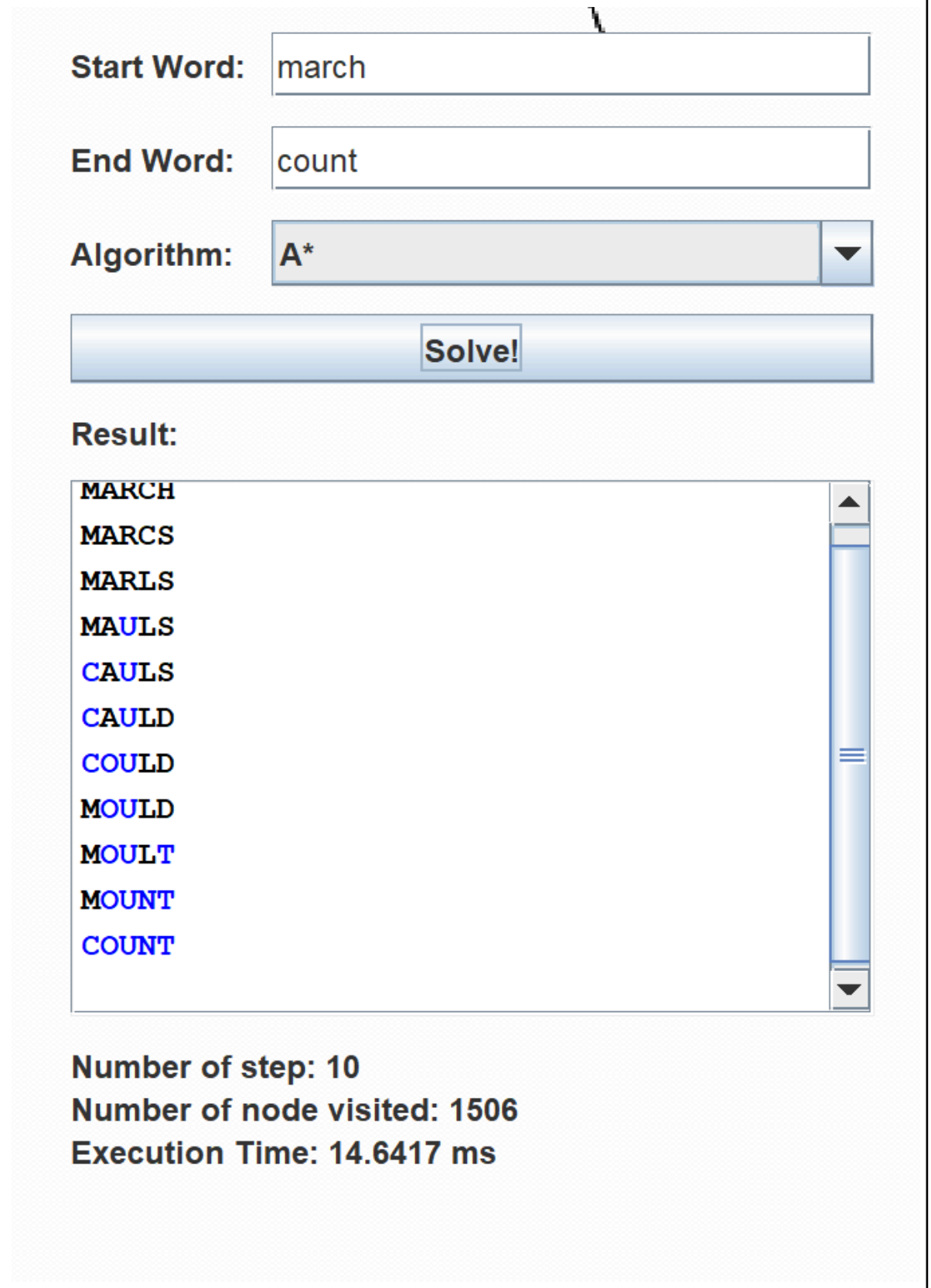
UCS: MARCH to COUNT



GBFS: MARCH to COUNT

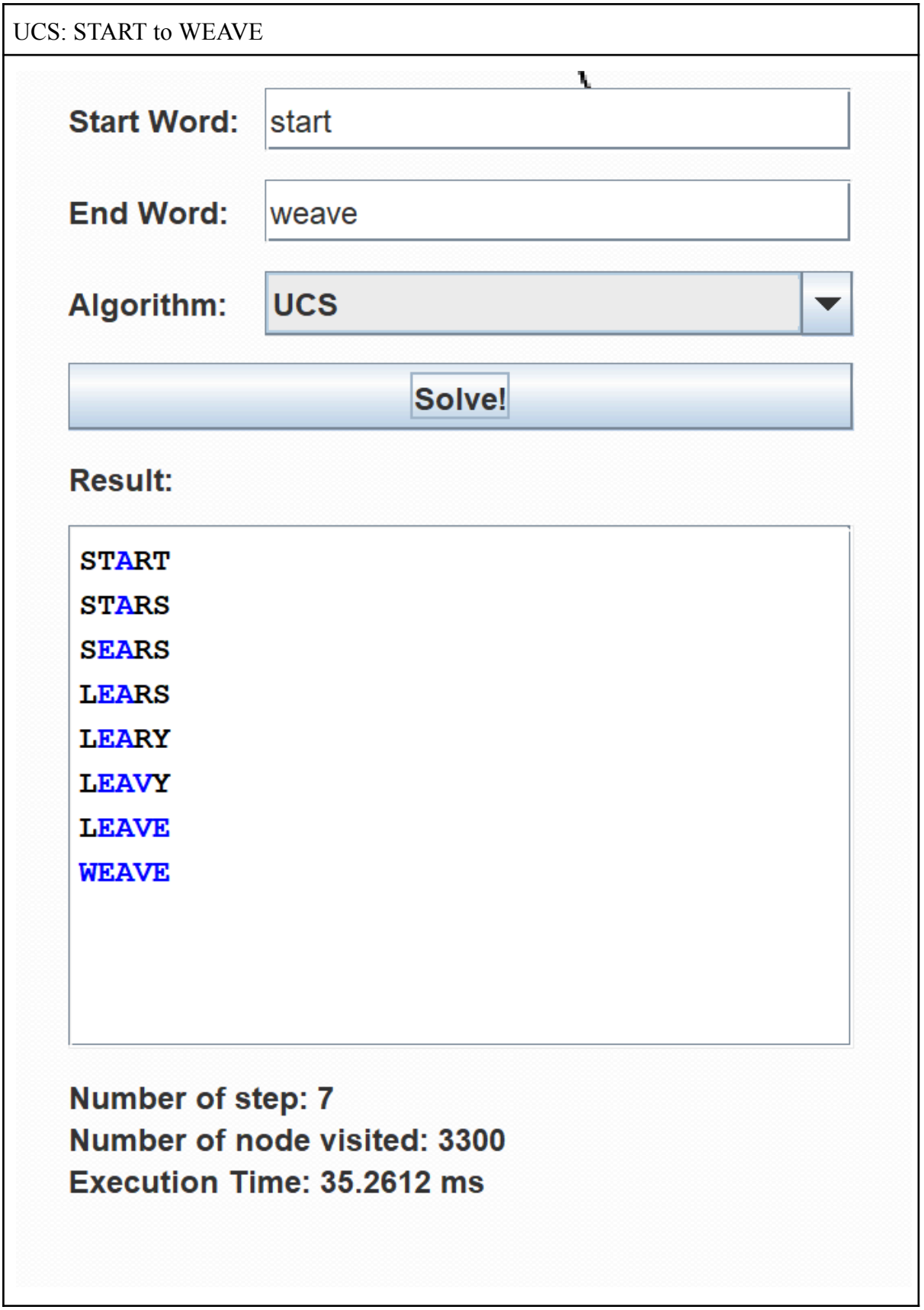


A*: MARCH to COUNT

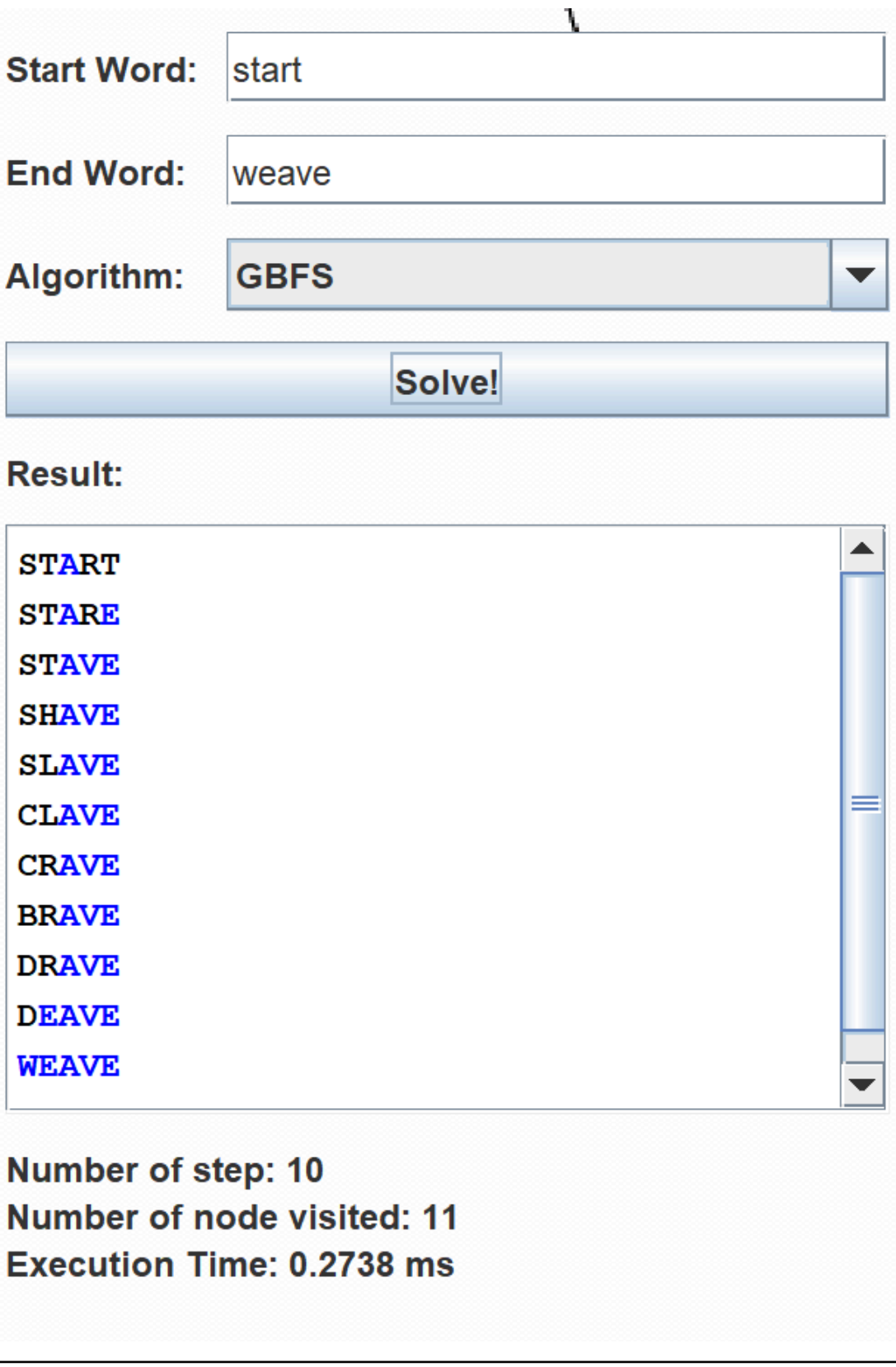


4.3 Test 3

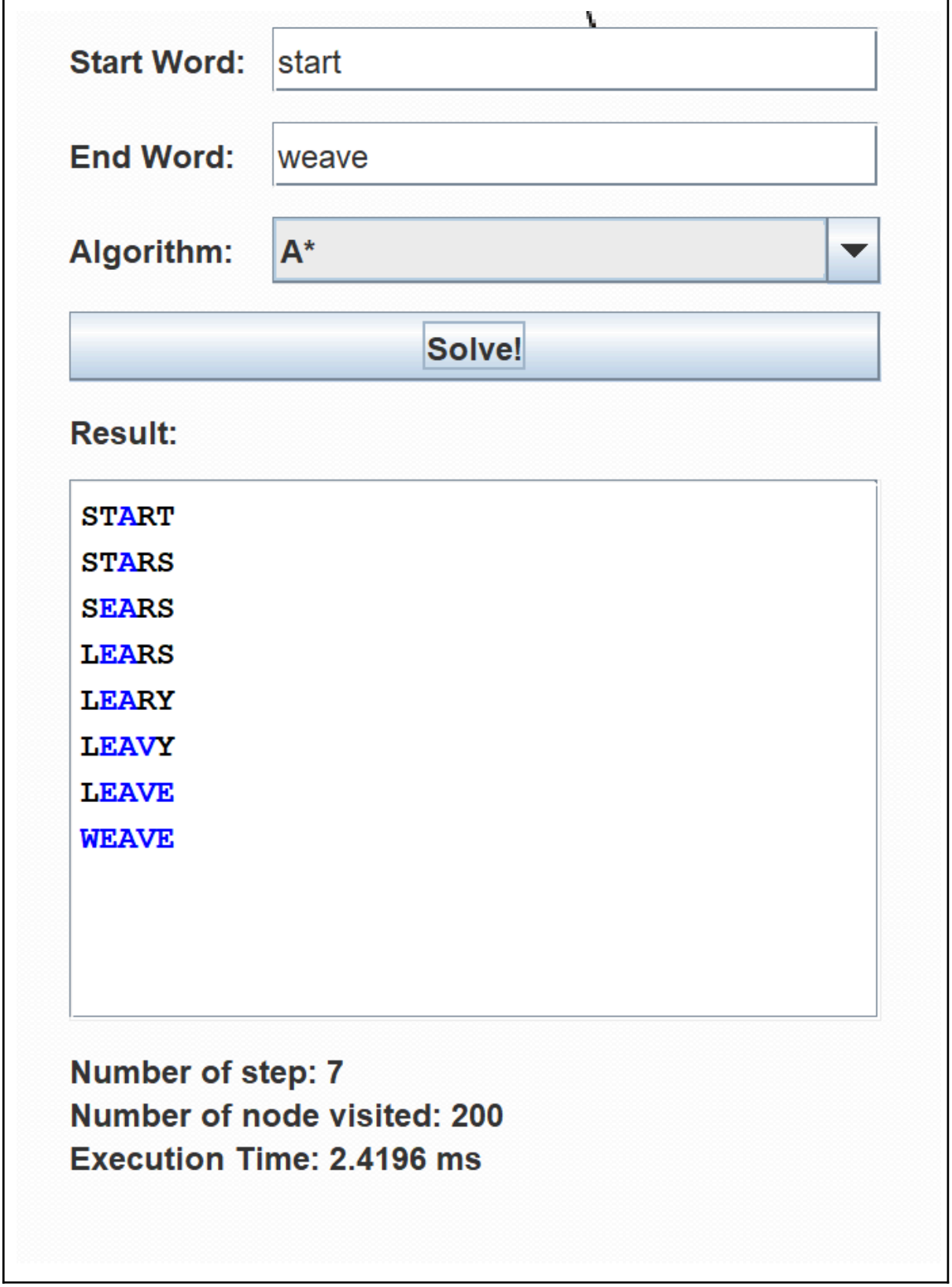
UCS: START to WEAVE



GBFS: START to WEAVE

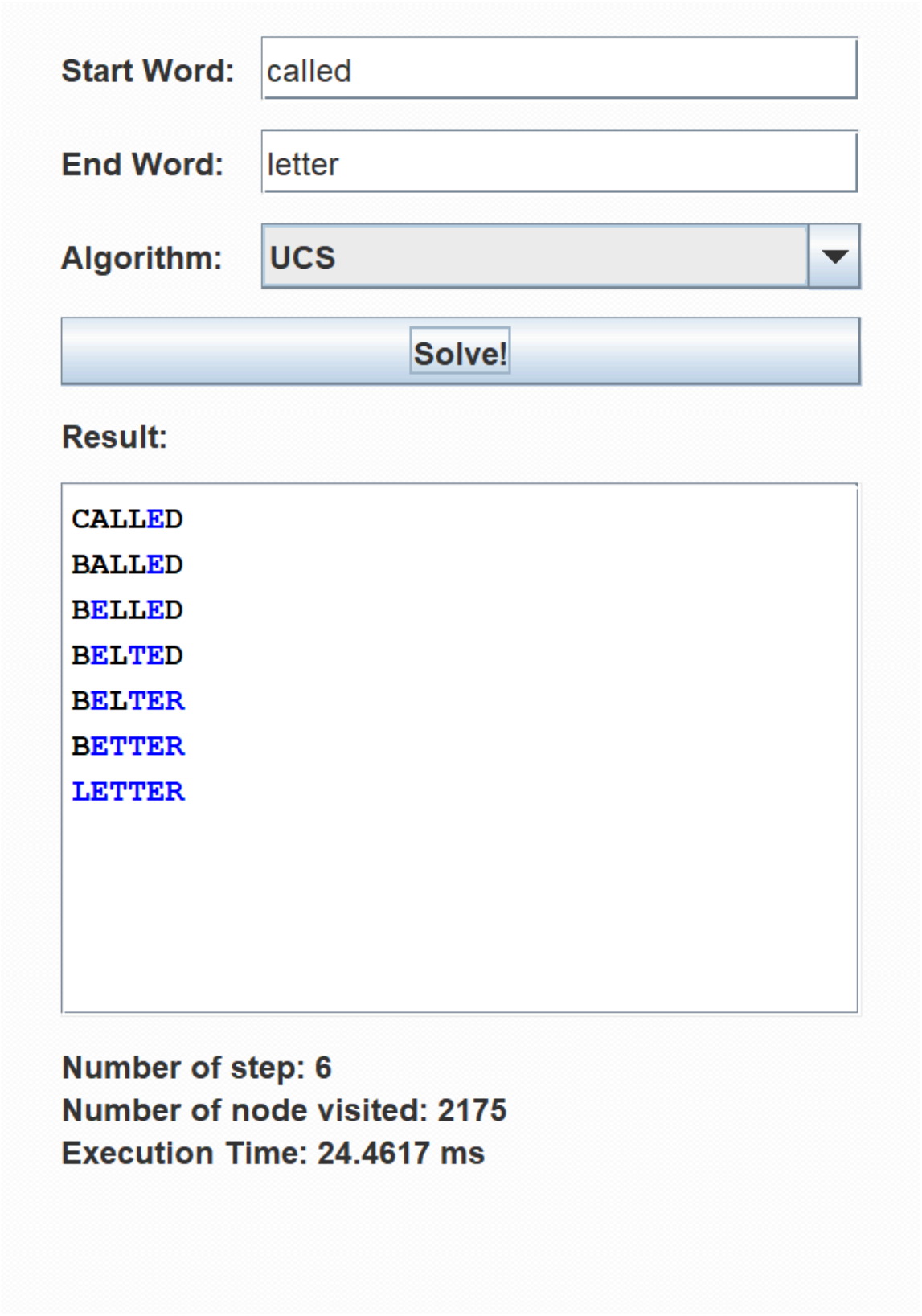


A*: START to WEAVE

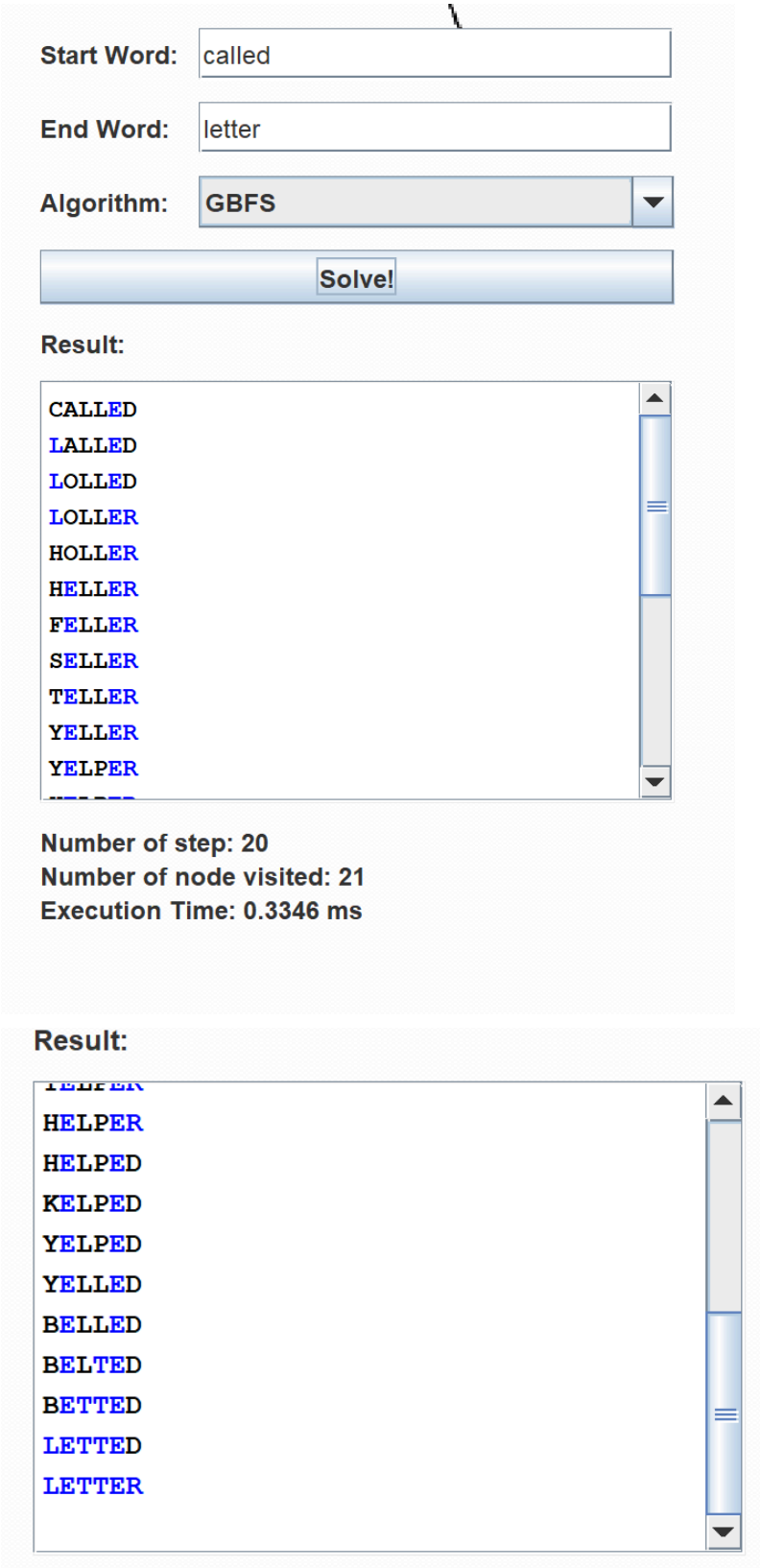


4.4 Test 4

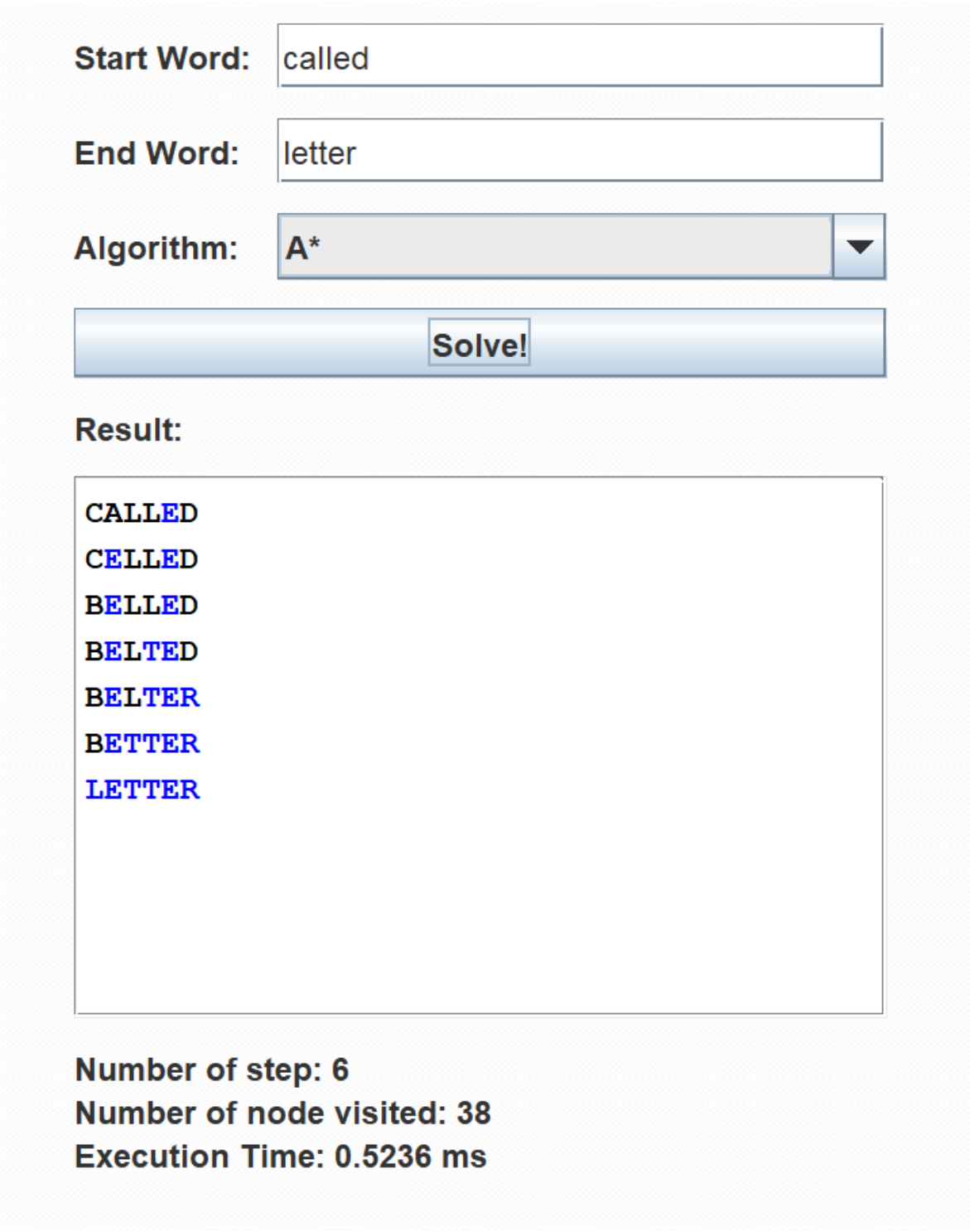
UCS: CALLED to LETTER



GBFS: CALLED to LETTER

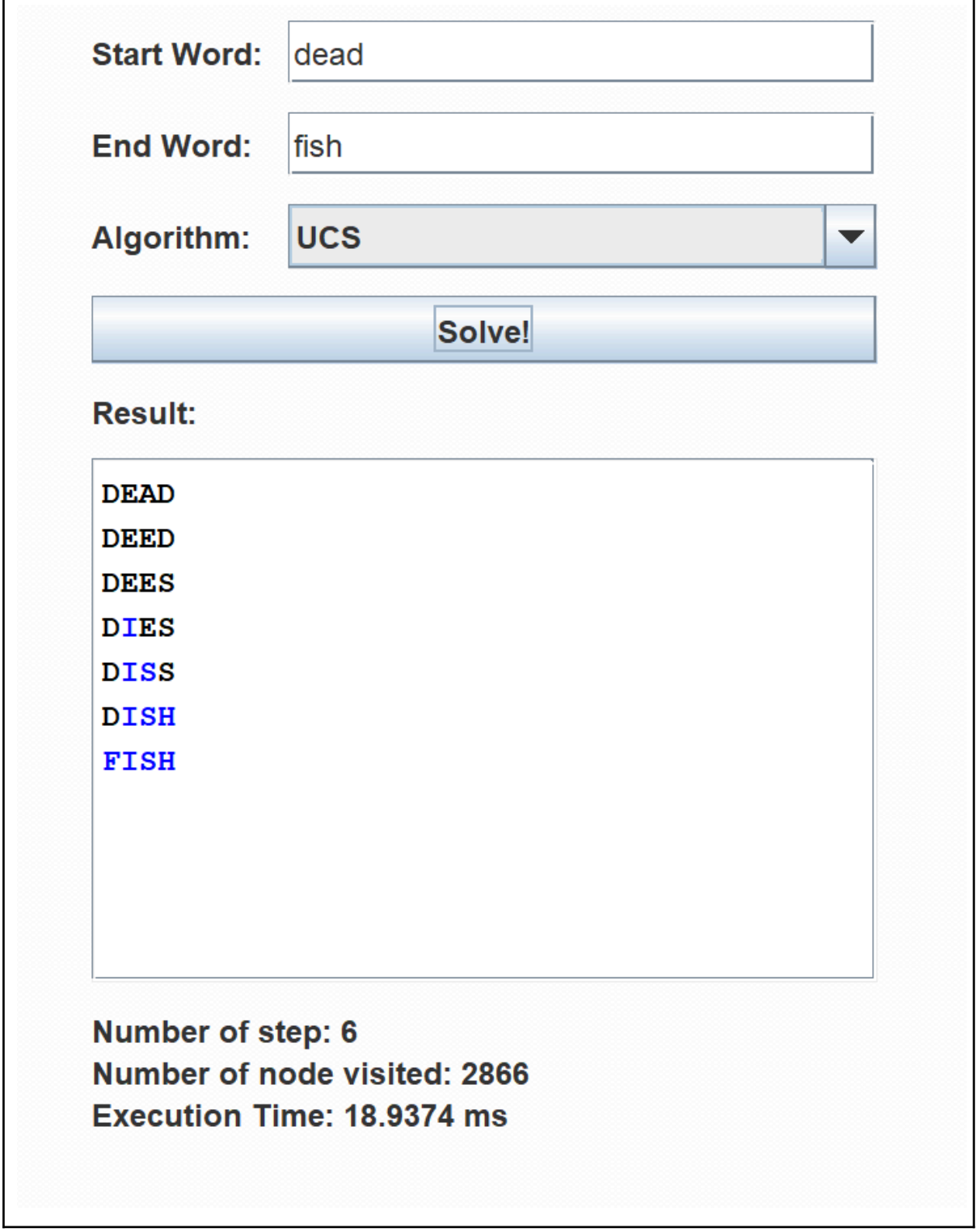


A*: CALLED to LETTER

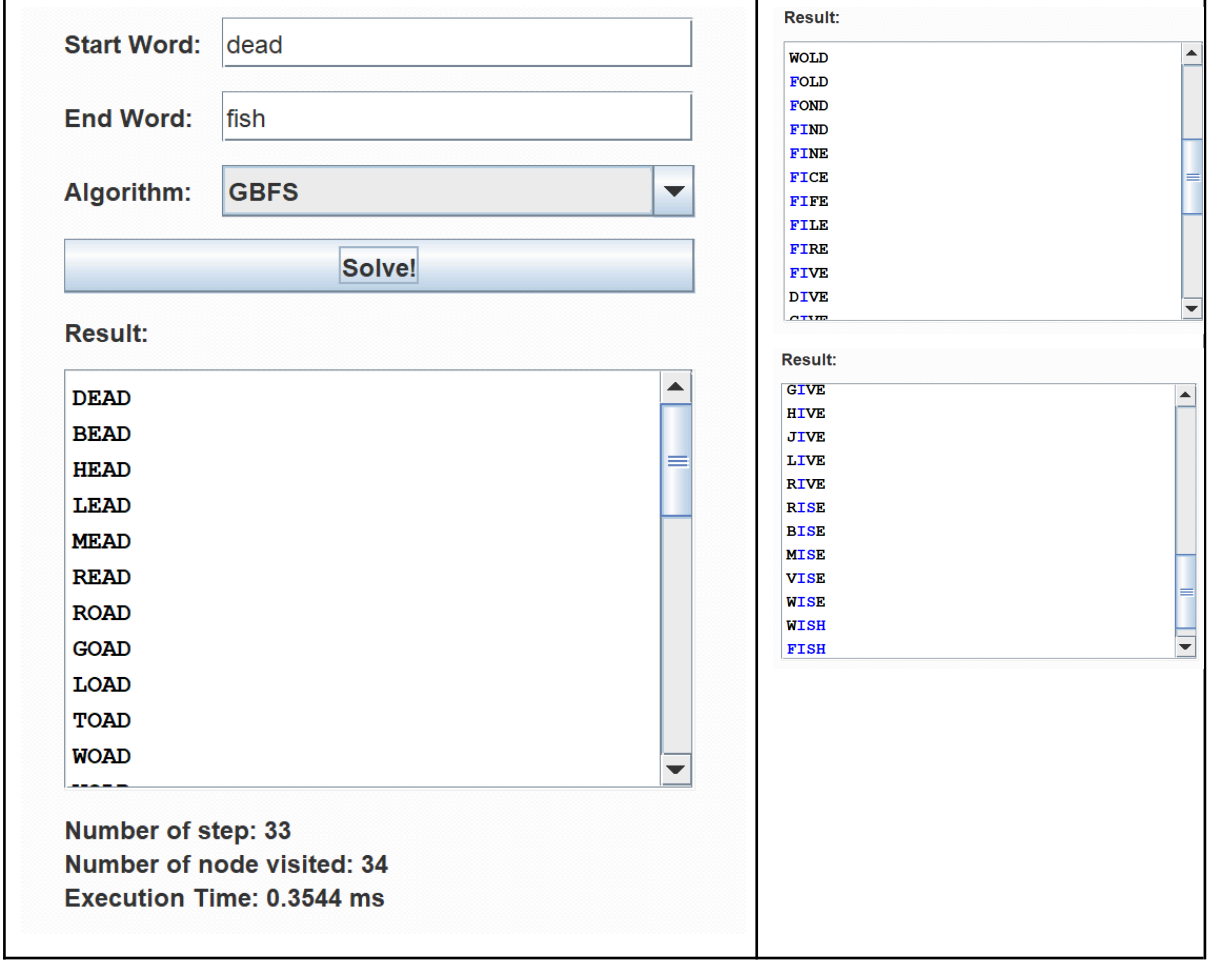


4.5 Test 5

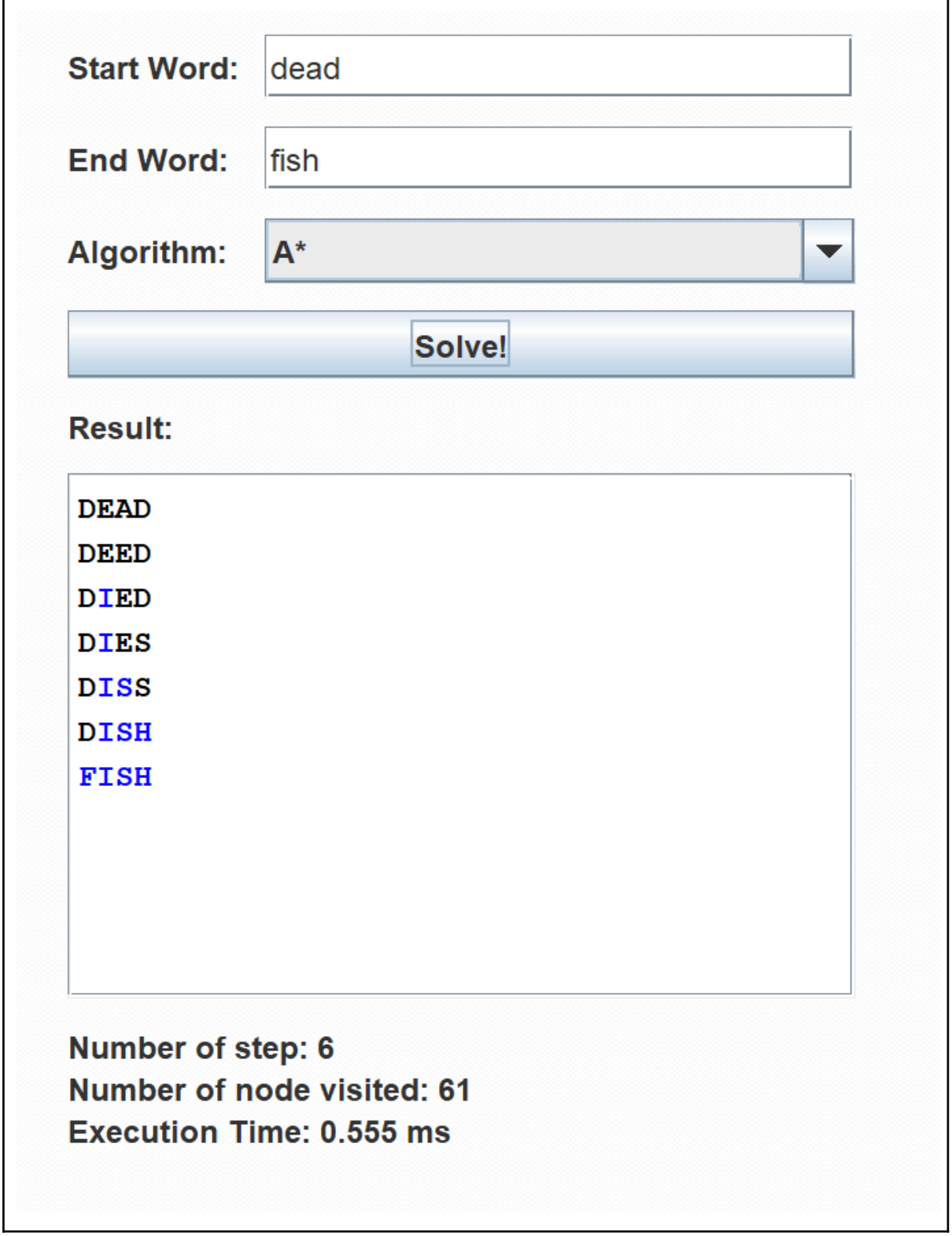
A*: DEAD to FISH



GBFS: DEAD to FISH

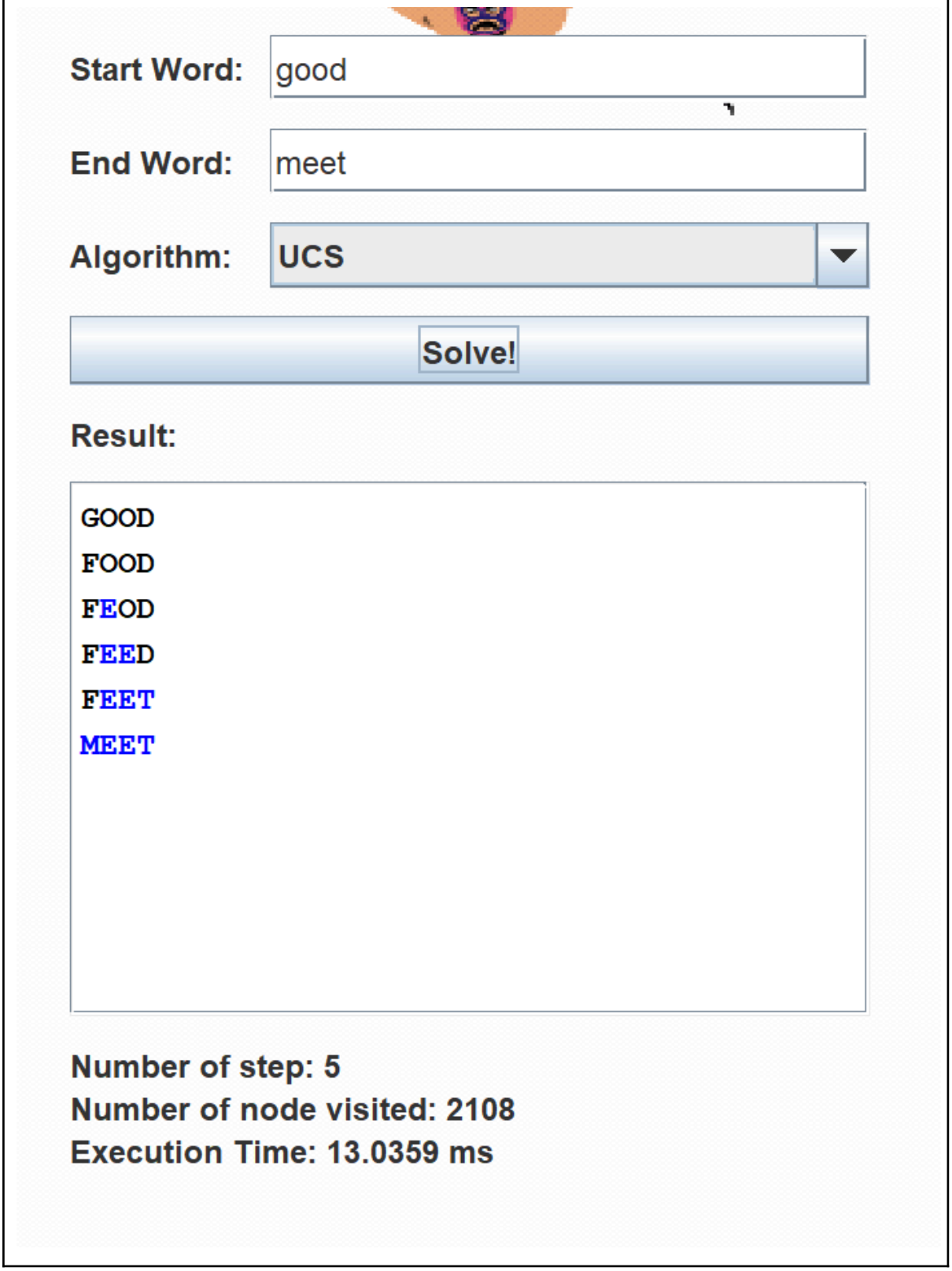


A*: DEAD to FISH

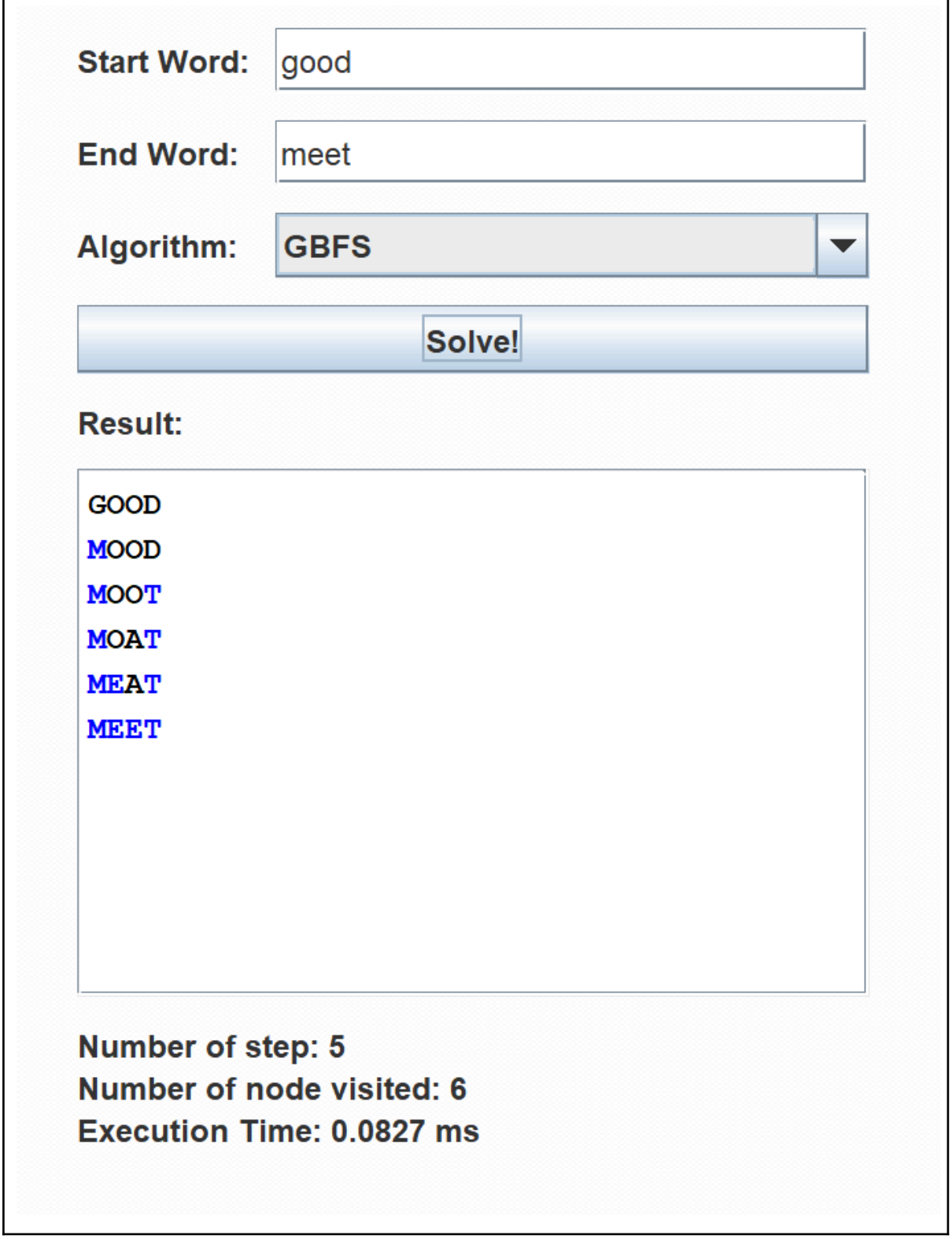


4.6 Test 6

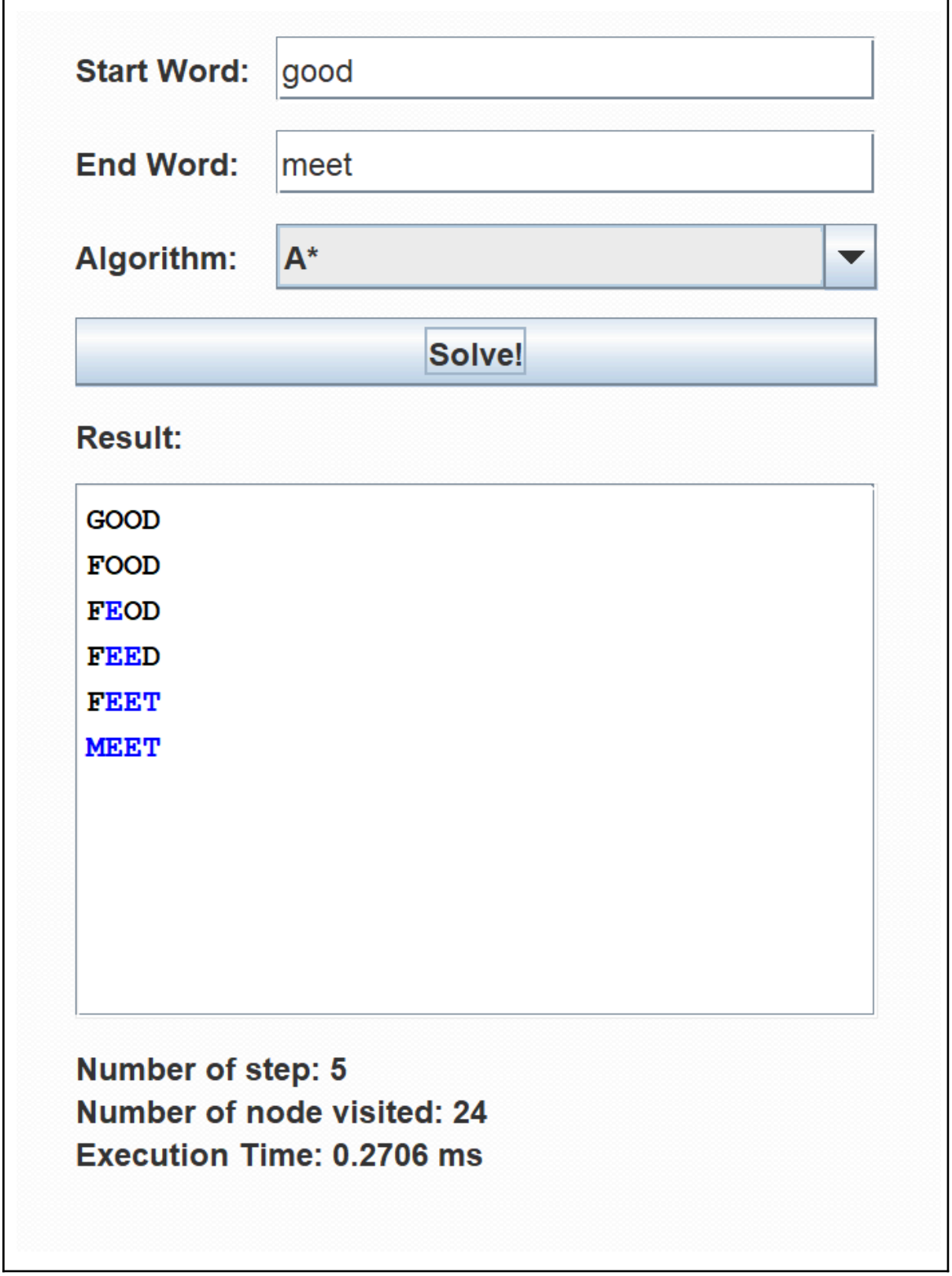
UCS: GOOD to MEET



GBFS: GOOD to MEET



A*: GOOD to MEET



BAB V
ANALISIS

5.1 Analisis Algoritma

Pada algoritma UCS, biaya yang digunakan untuk mencapai suatu simpul dihitung menggunakan fungsi *g(n)*, yaitu biaya yang sudah dikeluarkan untuk mencapai simpul *n*. Pada algoritma UCS untuk permainan *Word Ladder*, *g(n)* adalah jarak dari simpul kata awal ke simpul saat ini yang akan dilakukan pengecekan. Hal ini sama dengan banyaknya perubahan langkah yang telah dilakukan. Sementara pada algoritma GBFS, biaya pada tiap simpul dihitung menggunakan fungsi evaluasi fungsi *h(n)*, yaitu perkiraan biaya dari simpul n ke tujuan akhir. Pada algoritma GBFS untuk permainan *Word Ladder*, *h(n)* adalah nilai *heuristic* yang diperoleh dari perhitungan berapa banyak jumlah karakter yang berbeda pada simpul yang akan dicek dengan kata tujuan. Sedangkan pada algoritma A*, biaya pada tiap simpul dihitung menggunakan fungsi evaluasi *f(n)*, yaitu perkiraan biaya total dari simpul awal melalui simpul n menuju ke tujuan akhir. Fungsi evaluasi pada algoritma A* tersebut dihitung dengan menggabungkan biaya pada algoritma UCS dan GBFS. Dapat dikatakan bahwa implementasi algoritma ini memiliki fungsi perhitungan biaya *f(n)* sebagai berikut.
f(n) = *g(n)* + *h(n)*, di mana:

- *g(n)*: banyak langkah perubahan yang telah dilakukan (*depth*).
- *h(n)*: berapa jumlah karakter yang berbeda pada simpul ekspansi dengan kata tujuan

Penggunaan *heuristic* dengan cara menghitung berapa banyak jumlah karakter yang berbeda pada simpul yang akan dicek dengan kata tujuan merupakan *heuristic* yang *admissible*. *Heuristic* dikatakan *admissible* jika nilai *heuristik* yang dipakai tidak *overestimate* dari nilai yang seharusnya. Pada persoalan *Word Ladder*, misalkan diberikan kata GOOD menuju DONE, maka *heuristic* akan memiliki nilai 3 karena terdapat 3 karakter yang berbeda di indeks yang sama. Dalam kasus ini, *heuristic* tersebut tidak *overestimate* jumlah langkah yang sebenarnya diperlukan. Hal ini karena setiap perubahan karakter dianggap dapat langsung dilakukan tanpa mempertimbangkan keberadaan kata di dalam kamus yang sebenarnya dapat memberikan jalur yang lebih jauh. Nilai *heuristic* ini memenuhi kriteria *admissible* dengan asumsi untuk setiap perubahan karakter adalah langkah yang valid walaupun dalam praktiknya dapat membutuhkan langkah yang lebih banyak untuk mendapatkan kata yang valid. Namun, maksimal dari perbedaan karakter ini adalah sebanyak *n*, yaitu panjang karakter kata tersebut sehingga nilai *heuristic* (*h*) akan selalu di bawah atau sama dengan *n* (*h* <= *n*).

Pada kasus permainan Word Ladder, algoritma UCS melakukan ekspansi dengan menghitung biaya berdasarkan kedalaman atau jarak langkah dari simpul induk. Hal ini sama saja menganggap bahwa setiap perubahan satu karakter memiliki biaya operasi satu langkah sehingga setiap perubahan dari satu kata ke kata lain memiliki biaya yang sama. Secara teoritis hal ini akan membuat algoritma UCS ini mirip seperti algoritma Breadth First Search dengan urutan leksikografis di kedalaman yang sama. Kedua algoritma ini akan sama-sama

membangkitkan setiap simpul dengan kedalaman yang sama, menelusuri semua kemungkinan perubahan satu karakter pada level yang sama sebelum selanjutnya akan bergerak ke perubahan karakter yang berikutnya.

Pada kasus permainan Word Ladder, algoritma A* memilih simpul untuk diekspansi berdasarkan penggabungan biaya yang sudah dikeluarkan untuk mencapai suatu simpul *(g(n))* dan estimasi biaya dari simpul ke *n* menuju tujuan akhir *(h(n))*. Sementara algoritma UCS memilih simpul untuk diekspansi berdasarkan biaya yang sudah dikeluarkan untuk mencapai suatu simpul *(g(n))* saja. Hal ini membuat algoritma A* bekerja lebih efisien karena penelusuran rute akan berfokus pada pencarian jalur yang menjanjikan, sedangkan algoritma UCS melakukan eksplorasi di tiap kedalaman yang sama seperti *Breadth First Search*. Proses ini membuat simpul yang dikunjungi atau dilakukan pengecekan pada algoritma A* akan lebih sedikit dibandingkan algoritma UCS. Dengan nilai *heuristic* yang *admissible*, algoritma A* akan dijamin menemukan solusi yang optimal dengan memprioritaskan kata-kata yang telah mendekati dengan kata tujuan. Secara teoritis, pada kasus ini algoritma A* lebih efisien dibandingkan algoritma UCS, apalagi untuk ruang pencarian yang besar.

Algoritma GBFS tidak akan menjamin solusi optimal bahkan bisa saja solusi tidak didapatkan untuk persoalan *Word Ladder*. Hal ini karena nilai *heuristic* yang digunakan. Pada GBFS nilai *heuristic* dihitung berdasarkan berapa banyak jumlah karakter yang berbeda pada simpul yang akan dicek dengan kata tujuan. Perubahan karakter ini bisa saja membuat algoritma GBFS memilih simpul yang secara nilai *heuristic* memiliki biaya operasional yang rendah, namun pada kenyataannya jalan yang akan ditempuh menuju kata tujuan lebih panjang dengan mempertimbangkan keberadaannya di dalam kamus, atau mungkin sudah tidak ada kata berikutnya lagi di dalam kamus padahal belum mencapai kata tujuan (tidak ada solusi). Pada algoritma GBFS ini tidak ada proses *backtracking* sehingga simpul yang dibangkitkan selanjutnya selalu bertetanggaan dengan simpul saat ini. Simpul yang telah dikunjungi atau kata yang telah dikunjungi juga tidak akan dikunjungi kembali untuk menghindari algoritma berjalan terus-menerus menggunakan *backtracking.* Oleh karena itu, algoritma ini tidak menjamin solusi optimal bahkan bisa saja solusi tidak didapatkan untuk persoalan *Word Ladder*.

5.2 Analisis Eksperimen

Pada eksperimen yang telah dijalankan, terdapat enam kali test yang diberikan untuk masing-masing algoritma. Terdapat perbandingan yang dapat dilihat untuk menentukan kelebihan dan kekurangan masing-masing algoritma. Berikut tabel perbandingan hasil testing berdasarkan ketiga algoritma.

Kriteria	Algoritma	Test 1	Test2	Test 3	Test 4	Test 5	Test 6
Optimal	UCS	Ya	Ya	Ya	Ya	Ya	Ya
	GBFS	Ya	No Path	Tidak	Tidak	Tidak	Ya
	A*	Ya	Ya	Ya	Ya	Ya	Ya
Simpul dikunjungi (ruang memori)	UCS	1139	6649	3300	2175	2866	2108
	GBFS	5	12	11	21	34	6
	A*	5	1506	200	38	61	24
Waktu (ms)	UCS	10.111	71.9596	35.2612	24.4617	18.9374	13.0359
	GBFS	0.1363	0.2947	0.2738	0.3346	0.3544	0.0827
	A*	0.1247	14.6417	2.4196	0.5236	0.555	0.2706

Pada algoritma UCS, jalur yang ditemukan selalu optimal. Algoritma UCS ini akan selalu complete selama memang ada solusinya karena dilakukan *backtracking* pada proses algoritma ini. Ekspansi yang dilakukan dengan menghitung biaya berdasarkan kedalaman atau jarak langkah dari simpul induk membuat algoritma ini mirip seperti algoritma Breadth First Search dengan urutan leksikografis. Namun, kelemahannya adalah algoritma ini bisa menjadi lambat (paling lama) karena algoritma cenderung menjelajahi semua kemungkinan jalur dengan biaya yang semakin meningkat. Kompleksitas waktu untuk algoritma UCS adalah *O(b^d)* dengan *b* adalah *branching factor* dan *d* adalah kedalamannya. Ruang memori yang digunakan pun akan lebih banyak dibandingkan algoritma yang lainnya, yaitu dengan kompleksitas ruang *O(b^d)* dengan *b* adalah *branching factor* dan *d* adalah kedalamannya.

Pada algoritma GBFS memiliki kemampuan untuk menemukan jalur yang cepat dalam ruang pencarian yang besar. Akan tetapi, algoritma ini memiliki kelemahan yaitu tidak selalu dapat menemukan solusi sehingga algoritma GBFS tidak dapat menjamin ditemukannya solusi (*not complete*). Selain itu, GBFS cenderung dapat terjebak pada minimum lokal di sepanjang jalur pencarian yang membuat jalur yang diperoleh tidak optimal. Hal ini terjadi karena algoritma hanya mempertimbangkan nilai heuristik lokal saat memilih simpul berikutnya untuk diekspansi, tanpa mempertimbangkan jalur keseluruhan. Keputusan yang dibuat oleh GBFS tidak dapat dibatalkan atau diubah. Setelah simpul diekspansi berdasarkan heuristik, tidak ada mekanisme untuk membatalkan atau mengubah keputusan tersebut (tidak ada *backtracking*). Dalam hal ekspansi *node*, ruang memori yang digunakan pada algoritma ini paling efisien karena hanya mengambil simpul terbaik di setiap langkahnya sehingga memiliki kompleksitas waktu dan ruang yang sangat kecil yaitu *O(bd)* dengan *b* adalah *branching factor* dan *d* adalah kedalamannya. *Branching factor* tersebut diperlukan di setiap simpul untuk kemudian dievaluasi simpul mana yang paling bagus nilai *heuristicnya* untuk

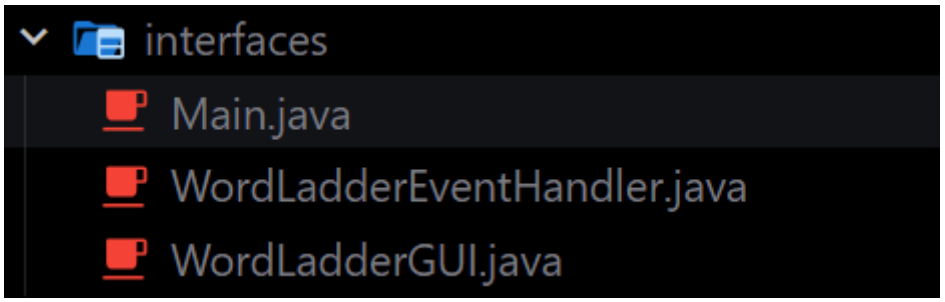
kemudian akan dilanjutkan pemeriksaan dan ekspansi *node.* Pada implementasinya setelah dipilih simpul yang terbaik di satu langkah, maka simpul tetangga tidak akan disimpan dalam *priority queue.*

Pada algoritma A*, jalur yang ditemukan selalu optimal. Algoritma A* ini akan selalu complete selama memang ada solusinya karena dilakukan *backtracking* pada proses algoritma ini. Waktu eksekusi algoritma ini sedikit lebih lama dibandingkan algoritma GBFS, namun jauh lebih cepat dibandingkan algoritma UCS. Algoritma A* menggabungkan keunggulan dari algoritma pencarian UCS dan GBFS dan akan efisien untuk ruang pencarian yang besar (memori) karena fungsi evaluasi ini pada algoritma A* memilih simpul untuk diekspansi berdasarkan kombinasi biaya yang sudah dikeluarkan (g(n)) dan estimasi biaya dari simpul saat ini ke tujuan akhir *(h(n))*. Ruang pencarian algoritma A* lebih baik dibandingkan dengan algoritma UCS sesuai dengan prinsip teoritisnya. Kompleksitas waktu dan ruang untuk algoritma A* sama halnya dengan algoritma UCS, yaitu *O(b^d)* dengan *b* adalah *branching factor* dan *d* adalah kedalamannya. Akan tetapi, dengan pendekatan *heuristic* yang *admissible*, maka algoritma A* akan menemukan jalur dengan lebih cepat dan mengurangi jumlah kunjungan ke *node* yang tidak diperlukan.

BAB V

PENJELASAN BONUS

Implementasi bonus berupa GUI. GUI dibuat dengan menggunakan Java Swing dari pustaka pemrograman Java. GUI pada program ini dapat menerima masukan Start Word, End Word, dan pilihan algoritma dalam bentuk *dropdown*. Selanjutnya tombol solve! dapat ditekan untuk melakukan proses algoritma. Hasil pencarian jalur akan ditampilkan pada kontainer result yang dilengkapi dengan pewarnaan karakter yang telah cocok dengan kata tujuan.



Folder interfaces berisikan semua file java yang bertanggung jawab untuk GUI. WordLadderEventHandler bertanggung jawab untuk menghubungkan input output dan proses algoritma menuju antarmuka. WordLadderGUI bertanggung jawab untuk memberikan dan mengatur tampilan antarmuka kepada pengguna. Sementara Main bertanggung jawab sebagai program utama untuk GUI.

DAFTAR PUSTAKA

[1] Munir, Rinaldi. 2021. Penentuan rute (Route/Path Planning) - Bagian 1. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian 1-2021.pd diakses pada 6 Mei 2024.

[2] Munir, Rinaldi. 2021. Penentuan rute (Route/Path Planning) - Bagian 2. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian 2-2021.pdf diakses pada 6 Mei 2024.

LAMPIRAN

Link Repository:

https://github.com/nanthedom/Tucil3_13522116

Check List:

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	✓	