

## **LAPORAN TUGAS KECIL 3**

### **IF2211 Strategi Algoritma**

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



Disusun oleh:

Nyoman Ganadipa Narayana (13522066)

**Program Studi Teknik Informatika  
Sekolah Teknik Elektro dan Informatika  
Institut Teknologi Bandung  
Jl. Ganesa 10, Bandung 40132**

<b>1</b>	<b>Deskripsi Permasalahan</b>	<b>3</b>
<b>2</b>	<b>Landasan Teori</b>	<b>4</b>
<b>2. 1</b>	<b>Algoritma Uniform Cost Search</b>	<b>4</b>
<b>2. 2</b>	<b>Algoritma Greedy Best First Search</b>	<b>4</b>
<b>2. 3.</b>	<b>Algoritma A*</b>	<b>5</b>
<b>2. 4.</b>	<b>Konteks Permainan Word Ladder</b>	<b>6</b>
<b>3</b>	<b>Analisis dan Implementasi</b>	<b>7</b>
<b>3. 2</b>	<b>Analisis dan Implementasi Algoritma UCS</b>	<b>7</b>
<b>3. 2</b>	<b>Analisis dan Implementasi Algoritma GBFS</b>	<b>7</b>
<b>3. 3</b>	<b>Analisis dan Implementasi Algoritma A*</b>	<b>7</b>
<b>4</b>	<b>Source Code</b>	<b>9</b>
<b>4. 1</b>	<b>Class Hierarchy</b>	<b>9</b>
<b>4. 2</b>	<b>Struktur Data Node</b>	<b>9</b>
<b>4. 3</b>	<b>Algoritma UCS</b>	<b>10</b>
<b>4. 4</b>	<b>Algoritma GBFS</b>	<b>13</b>
<b>4. 5</b>	<b>Algoritma A*</b>	<b>15</b>
<b>4. 6</b>	<b>Penjelasan Source Code dan Pemanggilan Algoritma</b>	<b>17</b>
<b>5</b>	<b>Testing</b>	<b>18</b>
<b>5. 2</b>	<b>Testing pada Algoritma UCS</b>	<b>19</b>
<b>5. 3</b>	<b>Testing pada Algoritma GBFS</b>	<b>23</b>
<b>5. 4</b>	<b>Testing pada Algoritma A*</b>	<b>27</b>
<b>5. 5</b>	<b>Testing Perbandingan Hasil dari Penggunaan A* dengan UCS</b>	<b>31</b>
<b>6</b>	<b>Analisis Data</b>	<b>33</b>
<b>7</b>	<b>Implementasi Bonus</b>	<b>35</b>
<b>8</b>	<b>Lampiran</b>	<b>38</b>
<b>8. 1</b>	<b>Pemenuhan Spesifikasi Tugas Kecil</b>	<b>38</b>
<b>8. 2</b>	<b>Pranala Github</b>	<b>38</b>

## 1 Deskripsi Permasalahan

*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.

### How To Play

This game is called a "word ladder" and was invented by Lewis Carroll in 1877.

#### Rules

Weave your way from the start word to the end word.  
Each word you enter can only change **1 letter** from the word above it.

#### Example



**Gambar 1.** Ilustrasi dan Peraturan Permainan *Word Ladder*

(Sumber: <https://wordwormdormdork.com/>)

Permainan ini cukup sederhana, dan sebagai programmer, akan dibuatkan solver dari permainan ini dengan 3 algoritma, yaitu UCS (Uniform Cost Search), GBFS (Greedy Best First Search), dan A\*.

## 2 Landasan Teori

### 2. 1 Algoritma *Uniform Cost Search*

Algoritma UCS (Uniform Cost Search), juga dikenal sebagai algoritma Dijkstra untuk *specific task* pencarian source dan target node dalam konteks *weighted graph*, adalah teknik pencarian yang digunakan untuk menemukan jalur terpendek dari titik awal ke titik tujuan dalam grafik yang berbobot dan terarah. Algoritma ini merupakan bagian dari keluarga algoritma pencarian berbasis grafik dan sangat berguna dalam peta dan pengaturan navigasi untuk menemukan jalur teroptimal.

Berikut adalah beberapa prinsip kerja dari algoritma UCS:

1. Algoritma ini tidak mempertimbangkan jumlah langkah dari titik awal, tetapi berfokus pada *cost* yang terakumulasi untuk mencapai setiap simpul. Ini adalah perbedaan utama antara UCS dan Breadth-first search (BFS), yang hanya menghitung langkah tanpa memperhatikan bobot lintasan.
2. UCS menggunakan *priority queue* untuk mengelola simpul yang akan dieksplorasi berdasarkan *cost* lintasan dari titik awal. Simpul dengan biaya terendah akan dieksplorasi lebih dulu.
3. Di setiap langkah, simpul dengan biaya total terendah dari titik awal diambil dari antrian prioritas, dan semua tetangganya yang dapat diakses akan di-*enqueue* kembali ke *priority queue* dengan biaya akumulasi *cost* yang diperbarui.
4. UCS mampu menangani siklus dalam grafik dengan memastikan bahwa setiap simpul hanya dikunjungi jika biaya untuk mencapai simpul tersebut lebih rendah dari biaya yang diketahui sebelumnya. Hal ini mencegah perulangan yang tidak perlu dan membantu dalam optimasi pencarian.
5. Pencarian berlanjut sampai tujuan dicapai, atau sampai semua jalur yang mungkin telah di-*visit* (*priority queue* kosong) – menunjukkan bahwa tidak ada jalur yang mungkin ke tujuan.

### 2. 2 Algoritma Greedy Best First Search

Algoritma Greedy Best-First Search (GBFS) adalah teknik pencarian heuristik yang digunakan untuk menemukan jalur dari titik awal ke titik tujuan dengan memilih untuk memperluas simpul yang paling "menjanjikan" atau terdekat dengan tujuan pada setiap langkah pencarian. Algoritma ini merupakan bagian dari keluarga algoritma pencarian yang menggunakan heuristik untuk memandu proses pencarian mereka agar lebih efisien dibandingkan dengan pendekatan yang tidak informasi seperti BFS atau DFS.

Berikut adalah beberapa prinsip kerja GBFS:

1. Berbeda dengan Uniform Cost Search yang berfokus pada biaya terendah yang terakumulasi dari titik awal, GBFS menggunakan fungsi heuristik untuk memperkirakan seberapa dekat atau seberapa baik sebuah simpul dibandingkan dengan tujuan. Pada setiap langkah, simpul yang memiliki nilai heuristik terendah (misalnya, jarak terpendek, diperkirakan dari simpul tersebut ke tujuan) dipilih untuk ekspansi selanjutnya.
2. Seperti UCS, GBFS juga menggunakan *priority queue* untuk mengelola simpul yang akan divisit. Simpul dengan nilai heuristik terbaik (misalnya terendah) akan dieksplorasi terlebih dahulu.
3. Simpul yang dipilih berdasarkan nilai heuristiknya kemudian diperluas, dan tetangganya ditambahkan ke *priority queue* dengan nilai heuristik masing-masing.
4. Meskipun GBFS lebih cepat dalam mencapai tujuan dalam banyak kasus, algoritma ini tidak selalu menjamin penemuan jalur yang optimal, karena bergantung pada kualitas dan akurasi fungsi heuristik yang digunakan.
5. Proses berlanjut sampai tujuan dicapai atau sampai tidak ada lagi simpul yang bisa divisit. Jika antrian prioritas kosong sebelum mencapai tujuan, maka tidak ada solusi yang ditemukan.

### 2. 3. Algoritma A\*

Algoritma A\* (A-star) adalah teknik pencarian yang efektif dan kuat yang digunakan untuk menemukan jalur terpendek dari titik awal ke titik tujuan dalam graf yang berbobot dan berarah. Algoritma ini menggabungkan kelebihan dari algoritma Greedy Best-First Search (GBFS) dengan strategi yang lebih sistematis seperti Uniform Cost Search (UCS), membuatnya tidak hanya cepat tetapi juga optimal dan lengkap.

Berikut adalah beberapa prinsip kerja dari algoritma A\*

1. Algoritma A\* menggunakan fungsi  $f(n)$  untuk setiap simpul  $n$ , yang merupakan estimasi dari biaya total (jarak terpendek) dari titik awal ke tujuan melewati simpul  $n$ . Fungsi ini didefinisikan sebagai  $f(n) = g(n) + h(n)$  dengan  $g(n)$  adalah biaya dari titik awal ke  $n$  dan  $h(n)$  adalah nilai heuristik dari  $n$  ke tujuan.
2.  $h(n)$  harus admissible, artinya harus tidak pernah memperkirakan biaya untuk mencapai tujuan lebih tinggi dari biaya sebenarnya (biaya sebenarnya menjadi upper bound). Contoh populer dari heuristik yang admissible adalah euclidean atau manhattan distance dalam pencarian jalur pada grid.
3. Penggunaan Antrian Prioritas: Seperti UCS dan GBFS, A\* menggunakan *priority queue* untuk memilih simpul yang memiliki nilai  $f(n)$  terendah untuk dieksplorasi selanjutnya.

4. Simpul dengan  $f(n)$  terendah di-*expand*, artinya semua tetangganya ditambahkan ke antrian prioritas dengan nilai  $f(n)$  yang baru dihitung berdasarkan  $g(n)$  dan  $h(n)$ .
5. A\* berhenti ketika tujuan ditemukan, atau ketika tidak ada simpul yang tersisa untuk dieksplorasi (yang menunjukkan tidak ada solusi). Karena menggunakan heuristik yang admissible, A\* menjamin penemuan jalur terpendek.

## 2. 4. Konteks Permainan *Word Ladder*

Dalam konteks permainan *word ladder*, setiap word dapat dijadikan sebagai simpul dalam suatu graf, dengan sisi yang menghubungkan dua simpul menyatakan bahwa kedua simpul tersebut memiliki perbedaan huruf sebanyak tepat 1 huruf. Sehingga graf yang dibentuk merupakan *undirected & unweighted graph* yang mengakibatkan algoritma UCS dan BFS akan menjadi algoritma yang sama dalam konteks ini.

### 3      Analisis dan Implementasi

#### 3.2    Analisis dan Implementasi Algoritma UCS

Algoritma ini merupakan algoritma yang menjamin solusi optimal. Karena pada setiap langkahnya kita akan menelusuri semua child node yang terhubung dengan memasukkannya terlebih dahulu ke sebuah antrian. Pada kasus ini, fungsi penilaian  $f$  akan hanya bergantung pada  $g$  yaitu cost suatu simpul dari simpul awal, dalam arti lain,  $g(n)$  hanya menunjukkan depth dari simpul  $n$ . Sehingga, sesuai dengan subbab 2.4, algoritma ini akan sama persis implementasinya dengan BFS karena graf yang ditelusuri merupakan graf tak berbobot. Dengan demikian pembangkitan node dan path yang dihasilkan pun akan sama persis.

Berikut adalah deskripsi detail setiap langkah algoritma UCS yang diimplementasikan.

1. Siapkan queue, dan map parent yang menyimpan parent dari suatu node saat node tersebut dikunjungi dalam pohon pencarian UCS.
2. Masukkan source ke dalam queue
3. Selama target belum pernah diekspan dan queue masih belum habis, lakukan:
  3. 1. Inisialisasi variabel size dengan jumlah elemen yang ada di queue sekarang.
  3. 2. Lakukan for loop sebanyak size:
    3. 2. 1 Ambil head dari queue – katakanlah current, tambahkan jumlah node yang diperiksa.
    3. 2. 2 Jika current adalah target, maka lanjut ke step no 5.
    3. 2. 3 Jika bukan, lakukan untuk setiap  $i$  yaitu posisi pada kata current, Inisialisasi kata baru yang posisi pada kata tersebut diubah menjadi huruf yang lain – katakanlah next.
    3. 2. 4. Jika kata tersebut merupakan kata bahasa Inggris yang valid, dan belum pernah ditaruh di queue, maka taruh kata tersebut di dalam queue kemudian set parent dari next menjadi current.
  3. 3 Lanjutkan proses (kembali ke no 3).
4. Jika kata target tidak pernah ditemukan, maka selesai.
5. Jika kata target ditemukan, inisialisasi list kosong – katakanlah solusi. Serta inisialisasi variabel current menjadi target.
6. Selama current bukan source, current ditambahkan ke dalam solusi, kemudian current di-set menjadi parent dari current.
7. Masukan source ke solusi.
8. *Reverse* urutan solusi.

### **3. 2 Analisis dan Implementasi Algoritma GBFS**

Algoritma ini merupakan algoritma yang urutan pemeriksaan nodenya sepenuhnya bergantung pada fungsi heuristik. Pada program yang telah dibuat, fungsi heuristik yang digunakan adalah jumlah karakter yang berbeda antara current node dengan target node, dengan memanfaatkan *priority queue* untuk menyimpan *adjacent nodes* dan pemilihan simpul selanjutnya, yang memprioritaskan berdasarkan sepenuhnya pada fungsi heuristik tersebut, maka priority queue mendapatkan solusi jauh lebih cepat dibandingkan algoritma UCS dan A\*. Namun, karena yang dipilih adalah hanya berdasarkan fungsi heuristik dan tidak melihat kondisi aktual, maka algoritma ini belum tentu mendapatkan solusi yang optimal.

Berikut adalah deskripsi detail setiap langkah algoritma GBFS yang diimplementasikan.

1. Siapkan priority queue yang memprioritaskan berdasarkan nilai heuristik, serta map parent sebagai map yang menyimpan parent dari suatu node.
2. Masukkan source ke dalam queue.
3. Selama target belum pernah diekspan, lakukan hal berikut
  3. 1 Ambil head dari queue – katakanlah current, tambahkan jumlah node yang diperiksa.
  3. 2 Jika current adalah target maka lanjut ke step no 5.
  3. 3 Untuk setiap posisi di current, coba ganti huruf tersebut menjadi huruf yang lain – katakanlah next.
  3. 4 Jika next merupakan kata bahasa inggris yang valid dan belum pernah ditaruh di queue, maka masukkan kata tersebut ke priority queue dengan memprioritaskan berdasarkan fungsi heuristiknya, serta menjadikan current menjadi parent dari next pada map.
  3. 5 Lanjutkan proses (kembali ke no 3).
4. Jika kata target tidak pernah ditemukan, maka selesai.
5. Jika kata target ditemukan, inisialisasi list kosong – katakanlah solusi. Serta inisialisasi variabel current menjadi target.
6. Selama current bukan source, current ditambahkan ke dalam solusi, kemudian current di-set menjadi parent dari current.
7. Masukan source ke solusi.
8. *Reverse* urutan solusi.

### **3. 3 Analisis dan Implementasi Algoritma A\***

Algoritma ini mengimplementasikan strategi UCS dan GBFS, mengambil kecepatan dan heuristik dari GBFS tetapi juga menjamin solusi optimal. Secara garis besar, algoritma ini menggunakan langkah yang sama dengan apa yang dilakukan pada UCS, namun, fungsi penilaian f yang digunakan bukan hanya bergantung pada kedalamannya (fungsi g) tetapi juga menggunakan fungsi heuristik h yang merupakan jumlah perbedaan huruf pada

posisi yang sama antara kata n dengan kata target, sehingga  $f(n) = g(n) + h(n)$ . Fungsi heuristik yang dipakai telah memenuhi syarat A\* yang merupakan *admissible*, yaitu fungsi heuristik yang digunakan harus selalu lebih kecil atau sama dengan langkah sebenarnya. Syarat *admissible* ini dapat dibuktikan melalui fakta bahwa jarak minimum antara dua kata tanpa harus melalui kata lain yang valid selalu lebih kecil sama dengan jarak minimum antara dua kata tetapi harus melalui kata lain yang valid, lebih khusus, yaitu karena adanya tambahan *constraint* sehingga diperlukan langkah-langkah yang tepat.

Algoritma ini dapat selalu lebih efisien dan lebih cepat daripada algoritma UCS karena menggunakan fungsi penilaian yang bergantung dengan fungsi heuristik, sehingga akan lebih memprioritaskan untuk mengekspansi simpul yang lebih dekat dengan solusi.

Berikut adalah deskripsi detail setiap langkah algoritma A\* yang diimplementasikan.

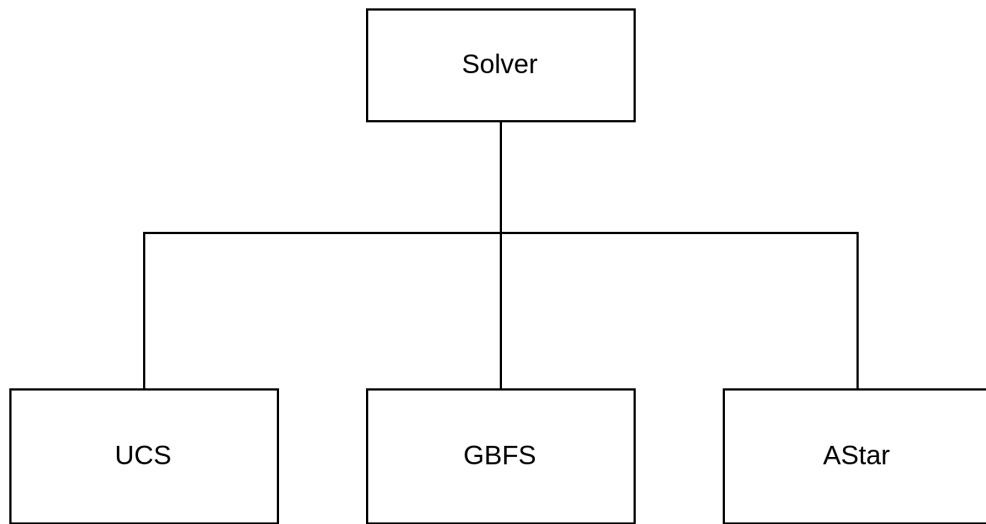
1. Siapkan priority queue yang memprioritaskan berdasarkan nilai evaluasi f map parent sebagai map yang menyimpan parent dari suatu node, dan map distance sebagai penyimpan depth dari suatu node.
2. Masukkan source ke dalam queue.
3. Selama target belum pernah diekspan, lakukan hal berikut
  3. 1 Ambil head dari queue – katakanlah current, tambahkan jumlah node yang diperiksa.
  3. 2 Jika current adalah target maka lanjut ke step no 5.
  3. 3 Untuk setiap posisi di current, coba ganti huruf tersebut menjadi huruf yang lain – katakanlah next.
  3. 4 Jika next merupakan kata bahasa inggris yang valid, maka lakukan hal berikut.
    3. 4. 1 Bandingkan nilai evaluasi sebelumnya (Jika belum pernah terevaluasi maka nilai evaluasi = nilai heuristik) dengan depth current + 1 + estimate.
    3. 4. 2 Jika next sudah pernah dimasukkan ke queue dan nilai evaluasi sebelumnya lebih jelek, maka lanjutkan prosesnya (kembali ke 3.3)
    3. 4. 3. Otherwise, maka artinya next memiliki evaluasi yang lebih bagus ataupun belum pernah dimasukkan ke queue, sehingga set map distance untuk next dengan distance dari current + 1, masukkan next ke priority queue dengan evaluasi yang baru, dan set parent dari next sebagai current.
  3. 5 Lanjutkan proses (kembali ke no 3).
4. Jika kata target tidak pernah ditemukan, maka selesai.
5. Jika kata target ditemukan, inisialisasi list kosong – katakanlah solusi. Serta inisialisasi variabel current menjadi target.

6. Selama current bukan source, current ditambahkan ke dalam solusi, kemudian current di-set menjadi parent dari current.
7. Masukan source ke solusi.
8. *Reverse* urutan solusi.

## 4 Source Code

### 4. 1 Class Hierarchy

Kode yang diimplementasikan untuk membuat program *logic* dari *word ladder solver* menggunakan hirarki kelas sebagai berikut.



**Gambar 2** Hirarki Kelas

### 4. 2 Struktur Data Node

Struktur data node yang digunakan pada algoritma GBFS dan A\* menyimpan kata dan fungsi penilaiannya, juga mengimplementasikan interface Comparable agar dapat dimasukkan ke *priority queue*.



```
public class Node implements Comparable<Node>{
    private String word;
    private Integer evaluation;

    public Node(String word, Integer evaluation) {
        this.word = word;
        this.evaluation = evaluation;
    }

    public String getWord() {
        return word;
    }

    public Integer getEvaluation() {
        return evaluation;
    }

    @Override
    public int compareTo(Node o) {
        if (this.evaluation < o.evaluation) {
            return -1;
        } else if (this.evaluation > o.evaluation) {
            return 1;
        } else {
            return this.word.compareTo(o.getWord());
        }
    }
}
```

**Gambar 3** Struktur Data Node

#### 4.3 Algoritma UCS

Berikut adalah implementasi algoritma UCS pada source code yang dibuat.

```
/*
 * This class is a solver for word ladder problem using BFS algorithm.
 * where BFS stands for Breadth First Search.
 */
public class UCS implements Solver{
    private boolean solved;
    private String source;
    private String target;
    private List<String> solution;
    private Map<String, Boolean> englishWordsMap;
    private Integer totalNodesVisited;
    private Long solveTime;

    public UCS(Map<String, Boolean> englishWordsMap) {
        solved = false;
        this.englishWordsMap = englishWordsMap;
        solution = new ArrayList<String>();
    }

    public boolean isSolved() {
        return solved;
    }

    public List<String> getSolution() throws Exception {
        if (isSolved()) {
            return solution;
        } else {
            throw new Exception("Solution not found!");
        }
    }

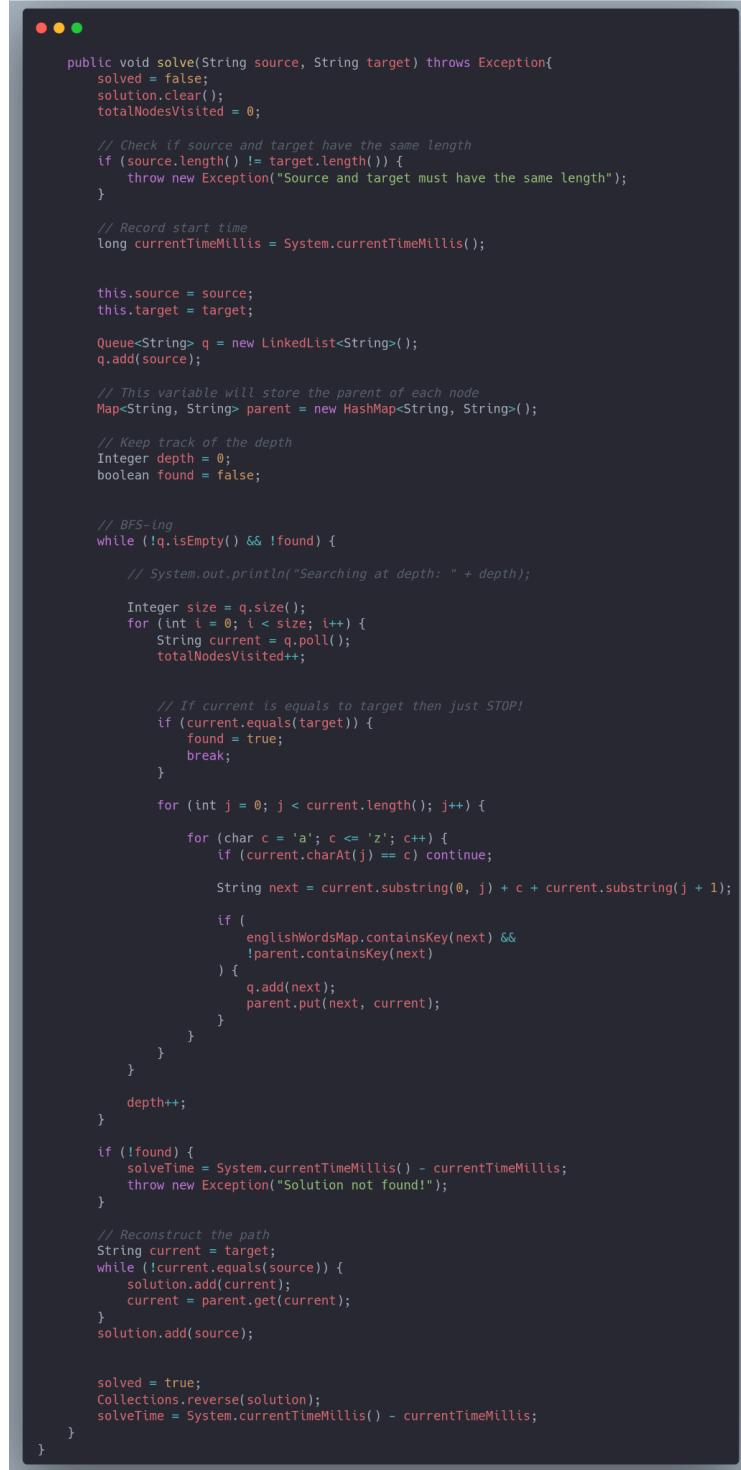
    public String getSource() {
        return source;
    }

    public String getTarget() {
        return target;
    }

    public Integer getTotalNodesVisited() {
        return totalNodesVisited;
    }

    public Long getSolveTime() {
        return solveTime;
    }
}
```

Gambar 4 Attribut dan Method Getter Setter



The screenshot shows a Java code editor with a dark theme. The code is a Java class named `Solver` that implements the Uniform Cost Search (UCS) algorithm to solve a word ladder puzzle. The code includes imports for `java.util.*`, `java.util.LinkedList`, `java.util.HashMap`, and `java.util.Queue`. It defines several private fields: `solved`, `solution`, `totalNodesVisited`, `current`, `parent`, `englishWordsMap`, and `solveTime`. The `solve` method takes `source` and `target` strings as parameters and throws an `Exception` if they are not of equal length. It records the start time, initializes variables, and sets up a queue `q` containing the source word. A `parent` map stores the parent of each node. The algorithm then enters a loop where it processes nodes from the queue until the target is found or all possibilities are exhausted. Inside the loop, it prints the current depth of search. For each node, it generates all possible words by changing one character at a time. If the generated word is in the English words map and has not been visited, it is added to the queue and its parent is updated. The loop continues until the target is found or the queue is empty. Finally, the solution path is reconstructed by backtracking through the parent map, starting from the target and adding words to the `solution` list until it reaches the source. The solved state is then set to true and the total time taken for the search is printed.

```
public void solve(String source, String target) throws Exception{
    solved = false;
    solution.clear();
    totalNodesVisited = 0;

    // Check if source and target have the same length
    if (source.length() != target.length()) {
        throw new Exception("Source and target must have the same length");
    }

    // Record start time
    long currentTimeMillis = System.currentTimeMillis();

    this.source = source;
    this.target = target;

    Queue<String> q = new LinkedList<String>();
    q.add(source);

    // This variable will store the parent of each node
    Map<String, String> parent = new HashMap<String, String>();

    // Keep track of the depth
    Integer depth = 0;
    boolean found = false;

    // BFS-ing
    while (!q.isEmpty() && !found) {
        // System.out.println("Searching at depth: " + depth);

        Integer size = q.size();
        for (int i = 0; i < size; i++) {
            String current = q.poll();
            totalNodesVisited++;

            // If current is equals to target then just STOP!
            if (current.equals(target)) {
                found = true;
                break;
            }

            for (int j = 0; j < current.length(); j++) {
                for (char c = 'a'; c <= 'z'; c++) {
                    if (current.charAt(j) == c) continue;

                    String next = current.substring(0, j) + c + current.substring(j + 1);

                    if (
                        englishWordsMap.containsKey(next) &&
                        !parent.containsKey(next)
                    ) {
                        q.add(next);
                        parent.put(next, current);
                    }
                }
            }
            depth++;
        }

        if (!found) {
            solveTime = System.currentTimeMillis() - currentTimeMillis;
            throw new Exception("Solution not found!");
        }
    }

    // Reconstruct the path
    String current = target;
    while (!current.equals(source)) {
        solution.add(current);
        current = parent.get(current);
    }
    solution.add(source);

    solved = true;
    Collections.reverse(solution);
    solveTime = System.currentTimeMillis() - currentTimeMillis;
}
```

**Gambar 5** Pencarian Solusi dengan UCS

#### 4. 4 Algoritma GBFS

Algoritma GBFS yang diimplementasikan merupakan varian dari GBFS yang menghandle cycle sehingga selalu mendapatkan solusi meskipun tidak optimal.



```
/*
 * This class is a solver for word ladder problem using GBFS algorithm.
 * Where GBFS stands for Greedy Best First Search.
 *
 * Here we will use the heuristic function to determine the next node to visit.
 * and my heuristic function is calculated based on how many minimum characters
 * are needed to change the current node to the target node without the constraint
 * the process must be a valid english word.
 *
 * This GBFS Implements variance of GBFS that always find a solution, even if it is not optimal!
 */
public class GBFS implements Solver {
    private boolean solved;
    private String source;
    private String target;
    private List<String> solution;
    private Map<String, Boolean> englishWordsMap;
    private Integer totalNodesVisited;
    private Double solveTime;

    /**
     * Constructor
     *
     * @param englishWordsMap is a map contains all the english words.
     */
    public GBFS(Map<String, Boolean> englishWordsMap) {
        solved = false;
        this.englishWordsMap = englishWordsMap;
        solution = new ArrayList<String>();
    }

    private Integer heuristic(String current){
        int count = 0;
        for (int i = 0; i < current.length(); i++) {
            if (current.charAt(i) != target.charAt(i)) {
                count++;
            }
        }
        return count;
    }

    public boolean isSolved() {
        return solved;
    }

    public List<String> getSolution() throws Exception {
        if (isSolved()) {
            return solution;
        } else {
            throw new Exception("Solution not found!");
        }
    }

    public String getSource() {
        return source;
    }

    public String getTarget() {
        return target;
    }

    public Integer getTotalNodesVisited() {
        return totalNodesVisited;
    }

    public Double getSolveTime() {
        return solveTime;
    }
}
```

Gambar 6 Attribut dan Method Getter Setter

```
/*  
 * Setup the solution.  
 *  
 * @param source  
 * @param target  
 * @throws Exception  
 */  
public void solve(String source, String target) throws Exception{  
    solved = false;  
    solution.clear();  
    totalNodesVisited = 0;  
  
    // Check if source and target have the same length  
    if (source.length() != target.length()) {  
        throw new Exception("Source and target must have the same length");  
    }  
  
    long currentTimeNanos = System.nanoTime();  
  
    this.source = source;  
    this.target = target;  
  
    // A priority queue to store child nodes to be visited  
    Queue<Node> pq = new PriorityQueue<Node>();  
    pq.add(new Node(source, heuristic(source)));  
  
    Map<String, String> parent = new HashMap<String, String>();  
  
    boolean found = false;  
  
    // Doing Greedy Best First Search  
    while (!pq.isEmpty()) {  
        Node current = pq.poll();  
        totalNodesVisited++;  
  
        if (current.getWord().equals(target)) {  
            found = true;  
            break;  
        }  
  
        String currentWord = current.getWord();  
        for (int i = 0; i < currentWord.length(); i++) {  
  
            for (char c = 'a'; c <= 'z'; c++) {  
                String next = currentWord.substring(0, i) + c + currentWord.substring(i + 1);  
  
                if (englishWordsMap.containsKey(next) && !parent.containsKey(next)) {  
                    pq.add(new Node(next, heuristic(next)));  
                    parent.put(next, currentWord);  
                }  
            }  
        }  
    }  
  
    if (!found) {  
        solveTime = (System.nanoTime() - currentTimeNanos)/1000000.0;  
        throw new Exception("Solution not found!");  
    }  
  
    // Reconstruct the path  
    String current = target;  
    while (current != source) {  
        solution.add(current);  
        current = parent.get(current);  
    }  
    solution.add(source);  
  
    solved = true;  
    Collections.reverse(solution);  
    solveTime = (System.nanoTime() - currentTimeNanos)/1000000.0;  
}
```

Gambar 7 Implementasi GBFS

## 4.5 Algoritma A\*

```
/*
 * This class implements the A* algorithm to solve the word ladder problem.
 * where here the heuristic function is the sum of the number of characters that
 * need to be changed to make the current node the target node and the depth of
 * the current node.
 */
public class AStar implements Solver {
    private boolean solved;
    private String source;
    private String target;
    private List<String> solution;
    private Map<String, Boolean> englishWordsMap;
    private Integer totalNodesVisited;
    private Long solveTime;

    public AStar(Map<String, Boolean> englishWordsMap) {
        solved = false;
        this.englishWordsMap = englishWordsMap;
        solution = new ArrayList<String>();
    }

    private Integer estimatedCostToGoal(String word) {
        int count = 0;
        for (int i = 0; i < word.length(); i++) {
            if (word.charAt(i) != target.charAt(i)) {
                count++;
            }
        }
        return count;
    }

    public boolean isSolved() {
        return solved;
    }

    public List<String> getSolution() throws Exception {
        if (isSolved()) {
            return solution;
        } else {
            throw new Exception("Solution not found!");
        }
    }

    public String getSource() {
        return source;
    }

    public String getTarget() {
        return target;
    }

    public Integer getTotalNodesVisited() {
        return totalNodesVisited;
    }

    public Long getSolveTime() {
        return solveTime;
    }
}
```

**Gambar 8** Attribute dan Method Getter Setter

```

● ● ●

public void solve(String source, String target) throws Exception {
    solved = false;
    solution.clear();
    totalNodesVisited = 0;

    // Check if source and target have the same length
    if (source.length() != target.length()) {
        throw new Exception("Source and target must have the same length");
    }

    long currentTimeMillis = System.currentTimeMillis();

    this.source = source;
    this.target = target;

    Queue<Node> pq = new PriorityQueue<Node>();
    pq.add(new Node(source, 0 + estimatedCostToGoal(source)));

    // This variable will store the parent of each node
    Map<String, String> parent = new HashMap<String, String>();

    // This variable will store the distance from source to the element
    Map<String, Integer> distance = new HashMap<String, Integer>();
    distance.put(source, 0);

    boolean found = false;
    while (!pq.isEmpty()) {
        Node current = pq.poll();
        totalNodesVisited++;

        if (current.getWord().equals(target)) {
            found = true;
            break;
        }

        String currentWord = current.getWord();
        for (int i = 0; i < currentWord.length(); i++) {

            for (char c = 'a'; c <= 'z'; c++) {
                if (currentWord.charAt(i) == c) continue;

                String next = currentWord.substring(0, i) + c + currentWord.substring(i + 1);

                if (englishWordsMap.containsKey(next)) {
                    Integer estimate = estimatedCostToGoal(next);

                    Integer oldEvaluation = ((distance.get(next) == null ? 0 : distance.get(next)) +
estimate;
                    Integer currentEvaluation = distance.get(currentWord) + 1 + estimate;

                    if (parent.containsKey(next) && oldEvaluation <= currentEvaluation) {
                        continue;
                    }

                    distance.put(next, distance.get(currentWord) + 1);
                    pq.add(new Node(next, distance.get(next) + estimate));
                    parent.put(next, currentWord);
                }
            }
        }
    }

    if (!found) {
        solveTime = System.currentTimeMillis() - currentTimeMillis;
        throw new Exception("Solution not found!");
    }

    // Reconstruct the path
    String current = target;
    while (current != source) {
        solution.add(current);
        current = parent.get(current);
    }
    solution.add(source);

    solved = true;
    Collections.reverse(solution);
    solveTime = System.currentTimeMillis() - currentTimeMillis;
}
}

```

**Gambar 9** Pencarian Solusi dengan Algoritma A\*

#### 4. 6 Penjelasan *Source Code* dan Pemanggilan Algoritma

Pada source code yang diberikan, masing-masing algoritma mengimplementasikan interface Solver serta menyimpan banyak attribut seperti solusi, waktu eksekusi, jumlah simpul yang di-visit, dan melemparkan exception jika tidak menemukan solusi. Jadi pemanggilan algoritma dari program utama, dapat dilakukan sebagai berikut.



```
import java.util.HashMap;
import java.util.Map;

import solver.*;

class Main {
    private static Map<String, Boolean> englishWordMap;

    private static void runCLI() {

        /**
         * Assume source, target, and englishWordMap have been setted up
         */

        source = source.toLowerCase();
        target = target.toLowerCase();

        Solver s;
        switch (algorithm) {
            case 1:
                s = new UCS(englishWordMap);
                break;
            case 2:
                s = new GBFS(englishWordMap);
                break;
            default:
                assert algorithm == 3;

                s = new AStar(englishWordMap);
                break;
        }

        try {
            s.solve(source, target);
            Integer solutionLength = s.getSolution().size();

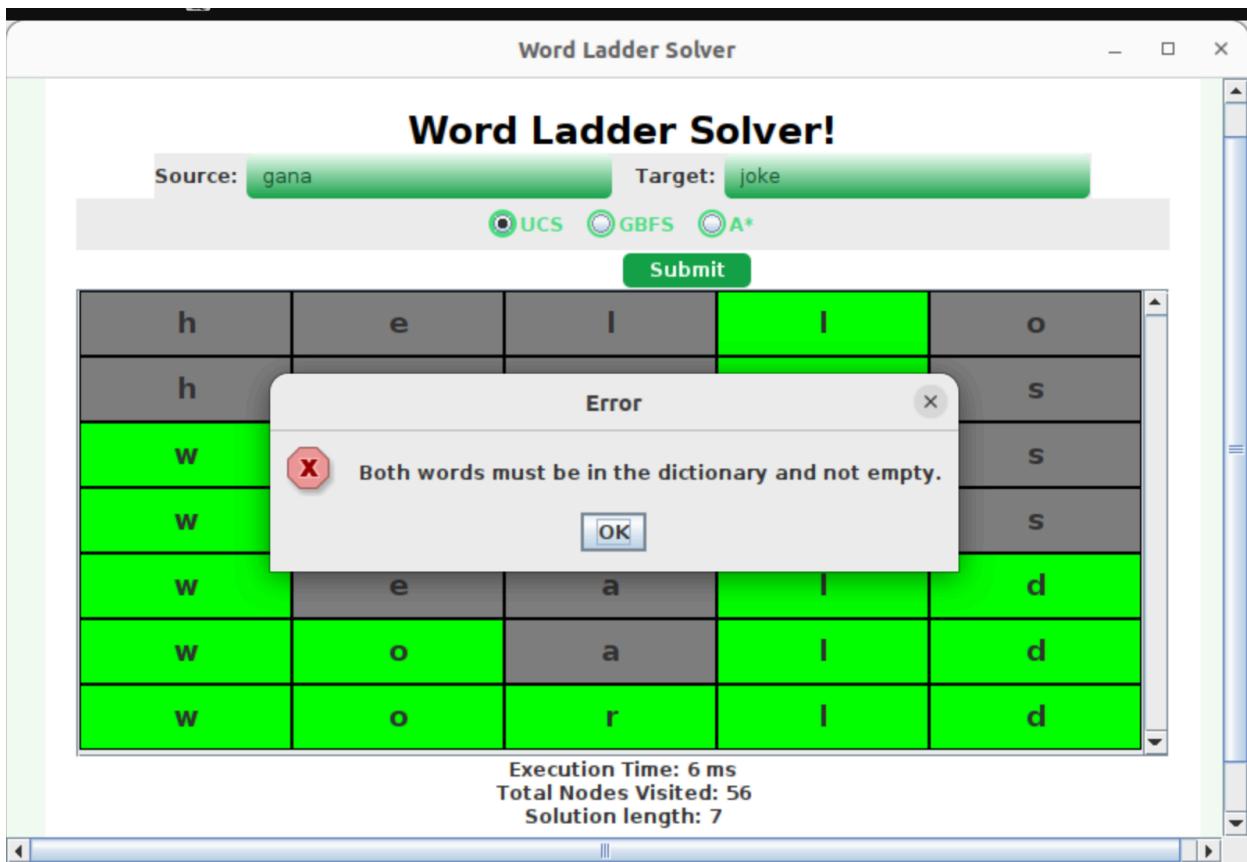
            for (int i = 0; i < solutionLength; i++) {
                System.out.println((i + 1) + ". " + s.getSolution().get(i));
            }

            System.out.println("Total nodes visited: " + s.getTotalNodesVisited());
            System.out.println("Solve time: " + s.getSolveTime() + " miliseconds.");
            System.out.println("Solution length: " + solutionLength);
        } catch (Exception e) {
            System.out.println("No solution found.");
            System.out.println("Total nodes visited: " + s.getTotalNodesVisited());
            System.out.println("Solve time: " + s.getSolveTime() + " miliseconds.");
        }
    }
}
```

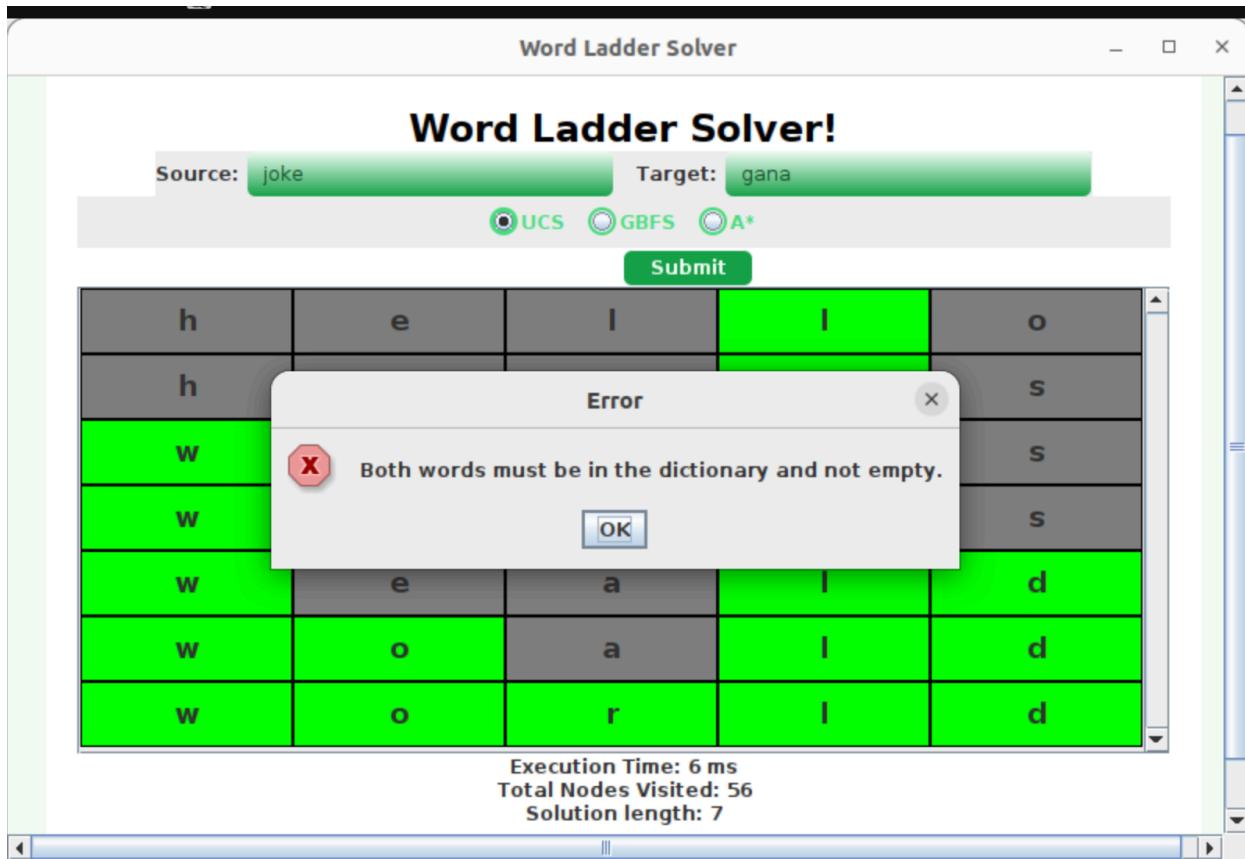
**Gambar 10** Pemanggilan Algoritma dari Program Utama

## 5 Testing

### 5.1 Invalid Input Testing



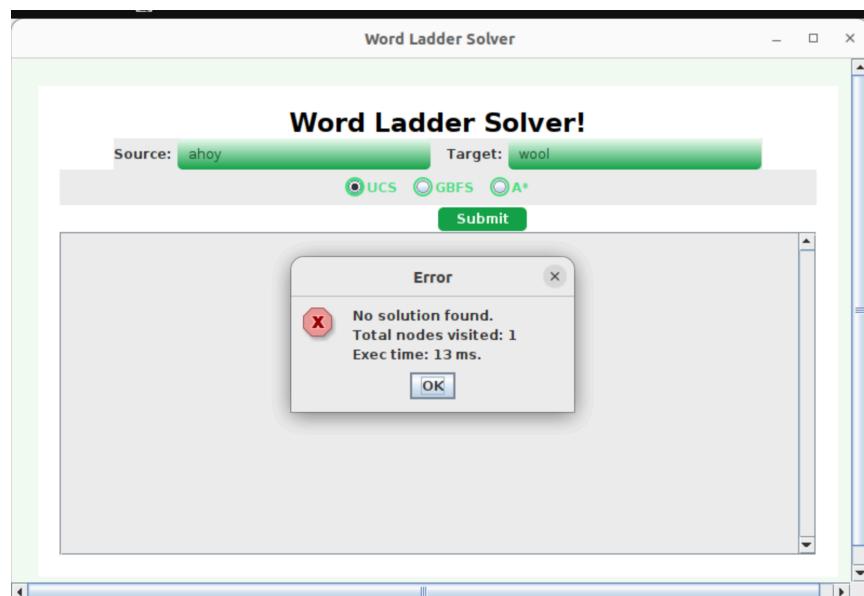
Gambar 11 Test Case 1



Gambar 12 Test Case 2

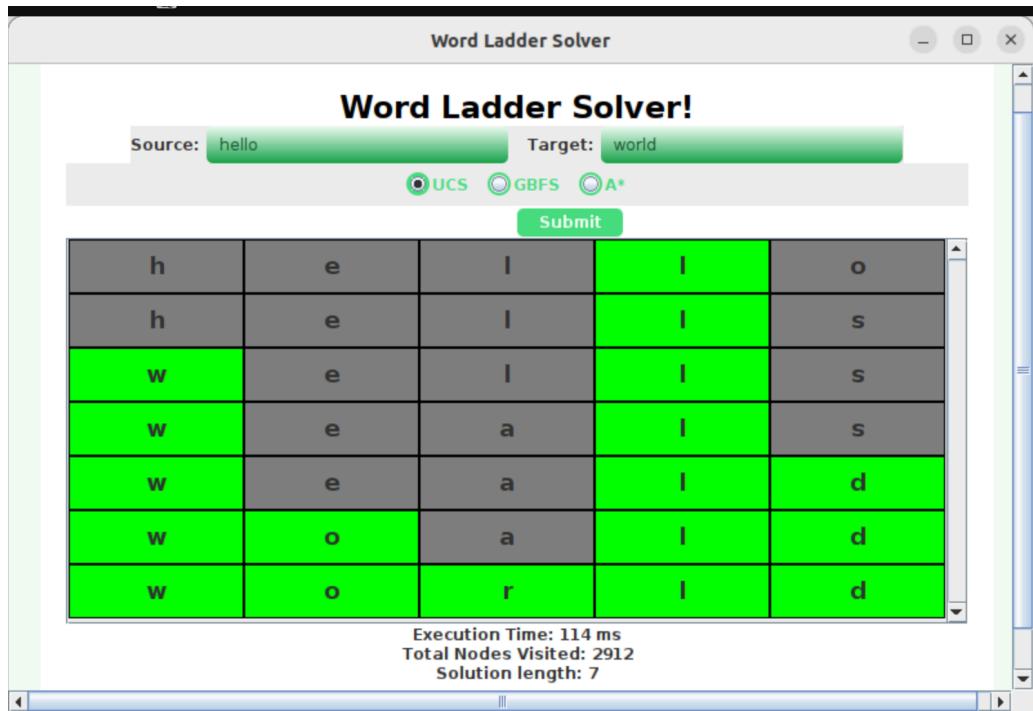
## 5. 2 Testing pada Algoritma UCS

1. Ahoy ke wool



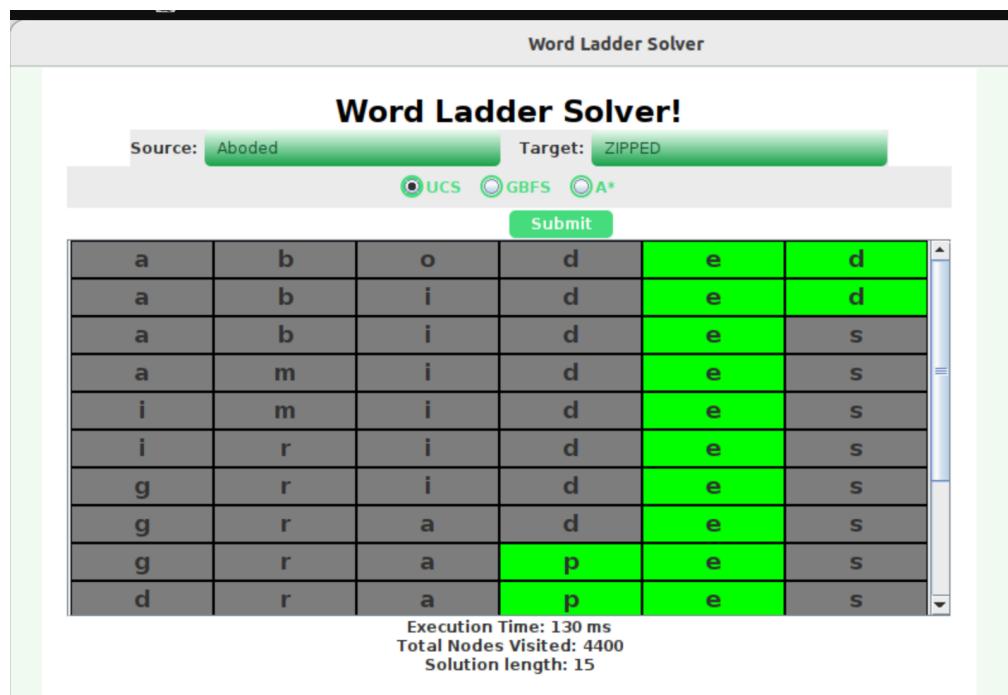
Gambar 13 Test Case Ahoy to Wool

2. hello ke world

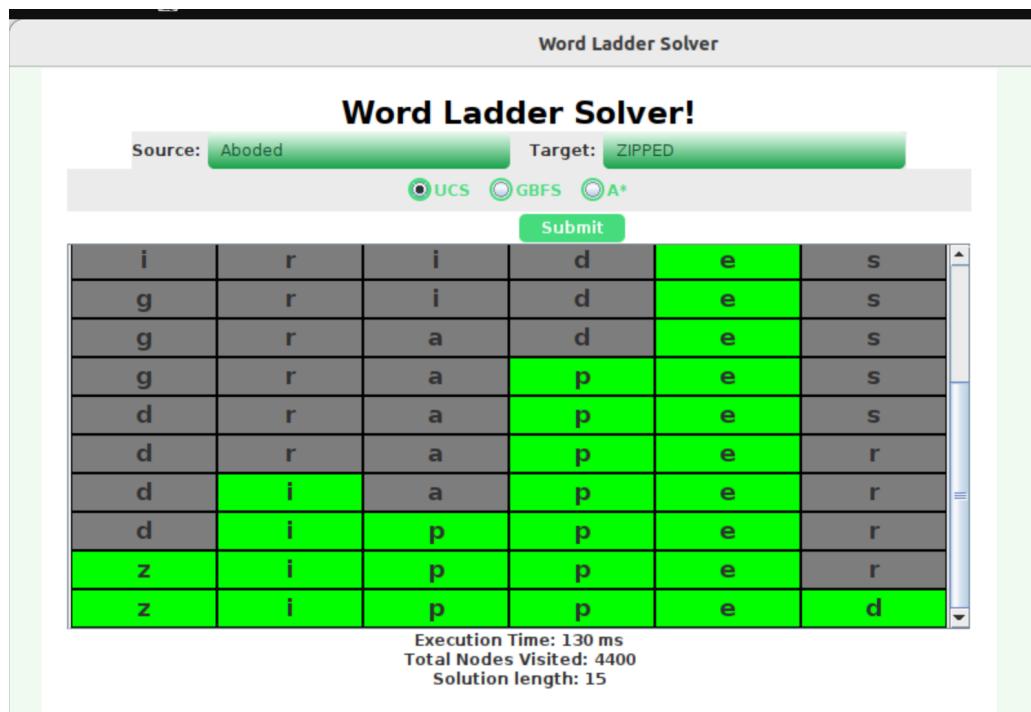


Gambar 14 Test Case hello to world

3. Aboded ke ZIPPED

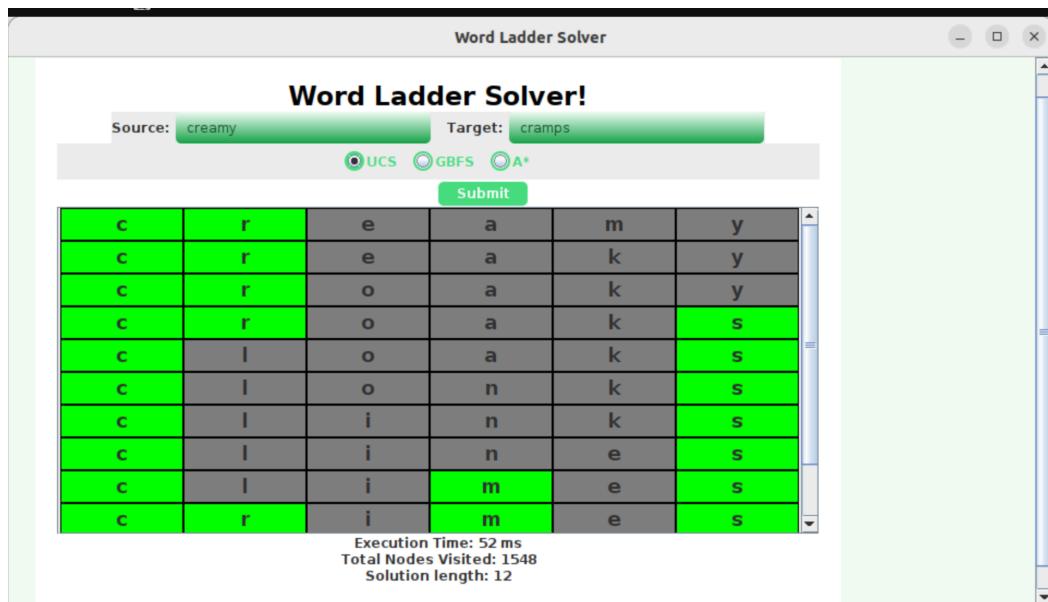


Gambar 15 Test Case aboded to ZIPPED

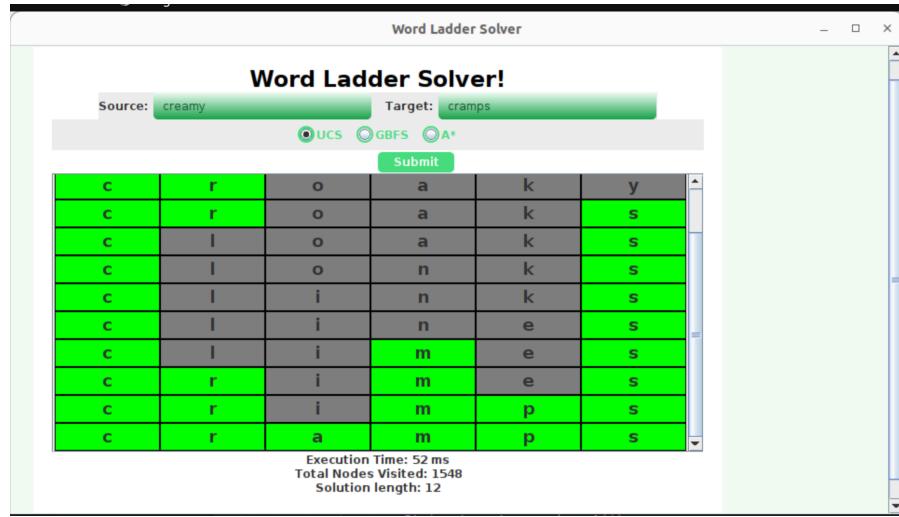


Gambar 16 Test Case aboded to ZIPPED

4. creamy ke cramps

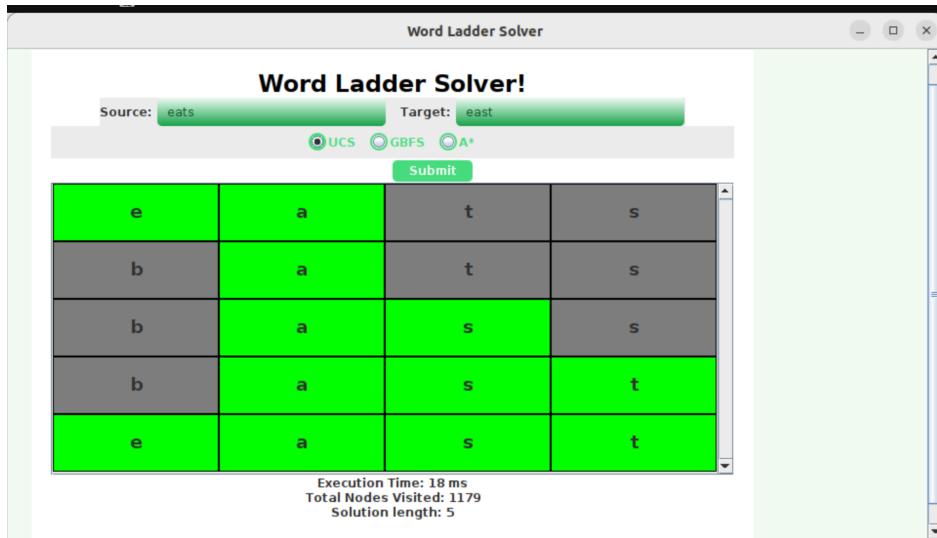


Gambar 17 Test Case creamy to cramps



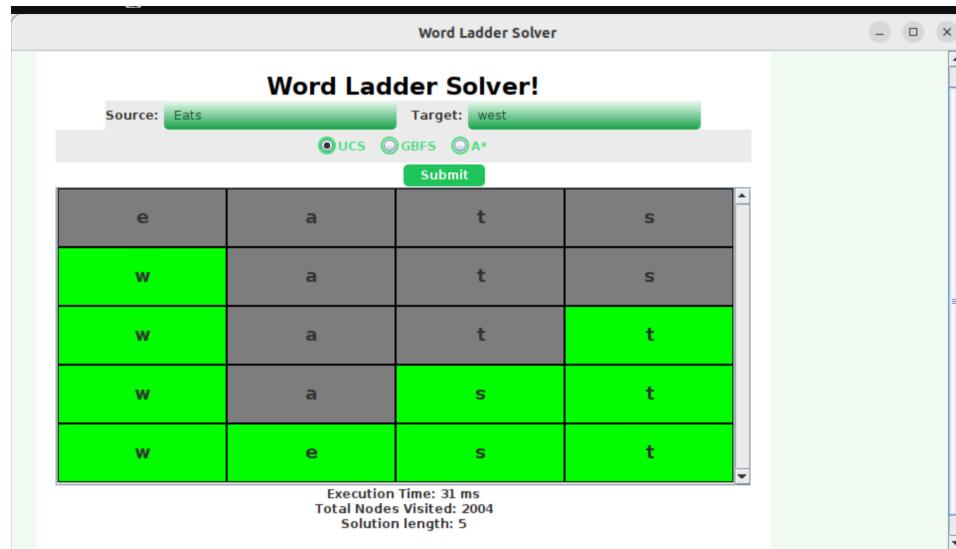
Gambar 18 Test Case creamy to cramps

5. eats ke east



Gambar 19 Test Case eats ke east

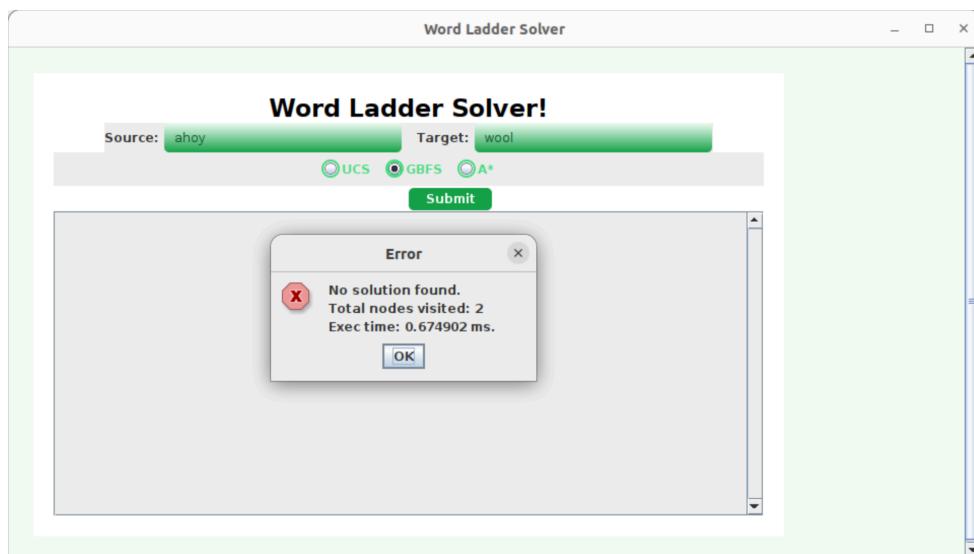
6. Eats ke west



Gambar 20 Test Case eats ke west

### 5. 3 Testing pada Algoritma GBFS

1. Ahoy ke wool



Gambar 21 Test Case ahoy ke wool

2. Hello ke world



Gambar 22 Test Case hello ke world



Gambar 23 Test Case hello ke world

### 3. Aboded ke zipped

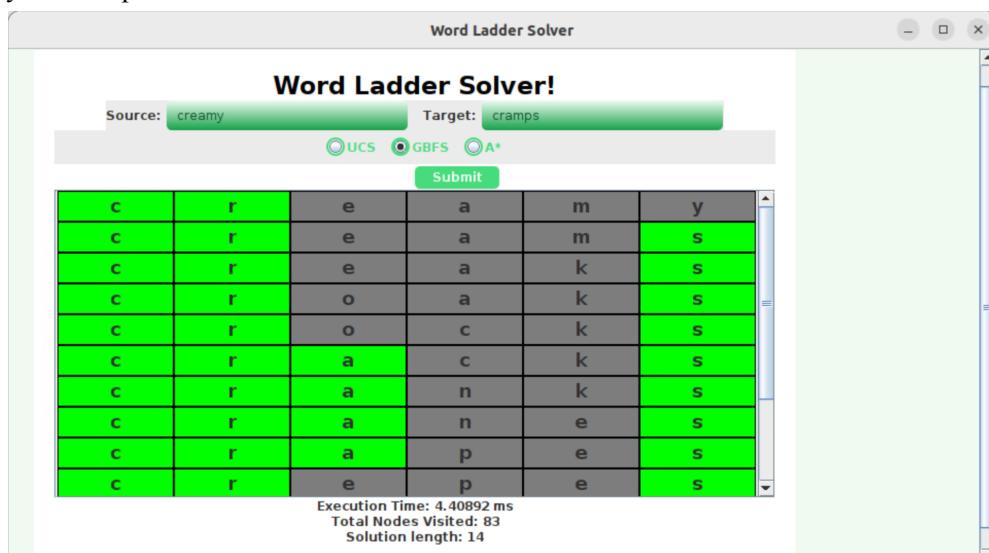


**Gambar 24** Test Case aboded ke zipped

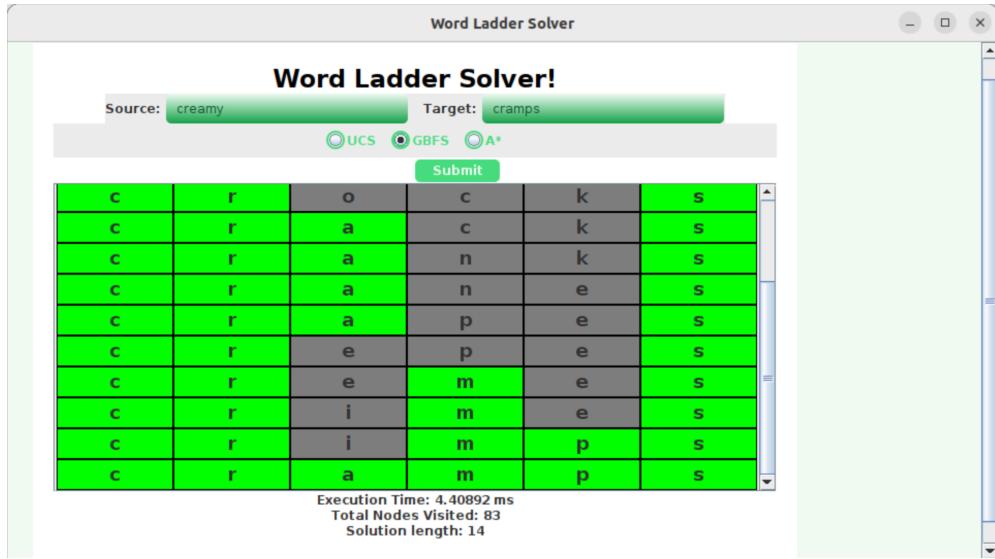


**Gambar 25** Test Case aboded ke zipped

#### 4. Creamy ke cramps

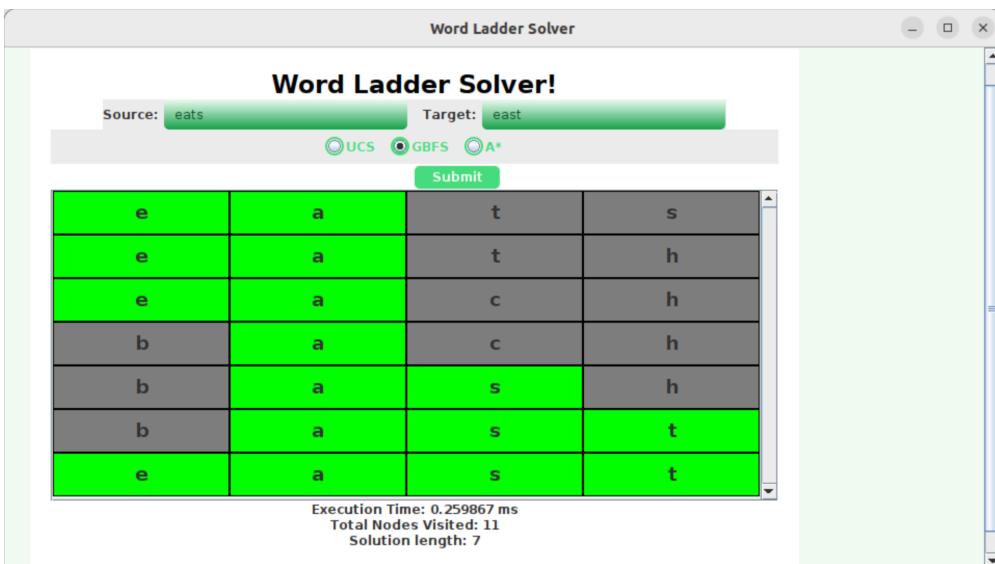


**Gambar 26** Test Case creamy ke cramps



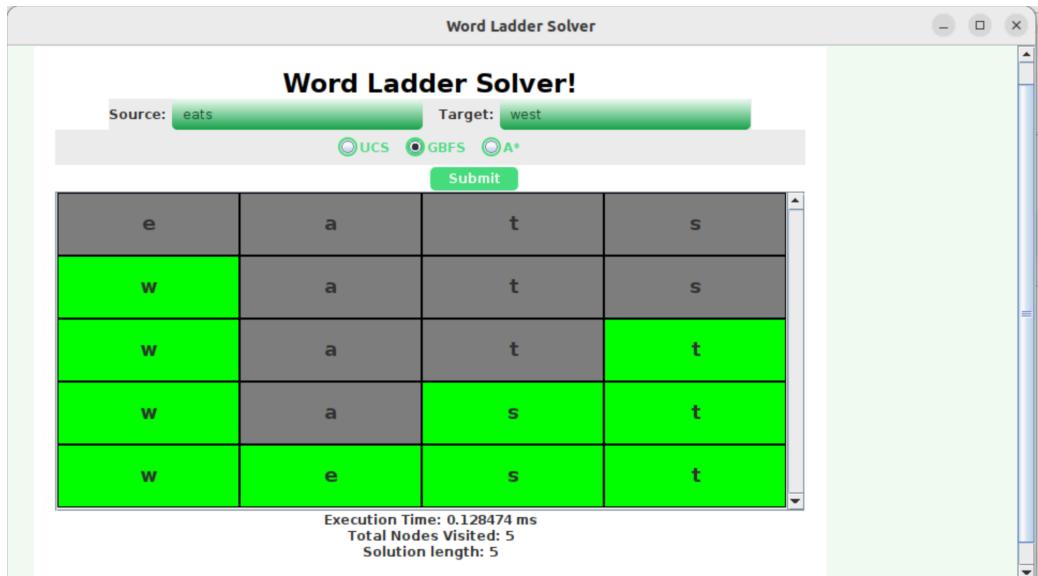
Gambar 27 Test Case creamy ke cramps

5. Eats ke east



Gambar 27 Test Case eats ke east

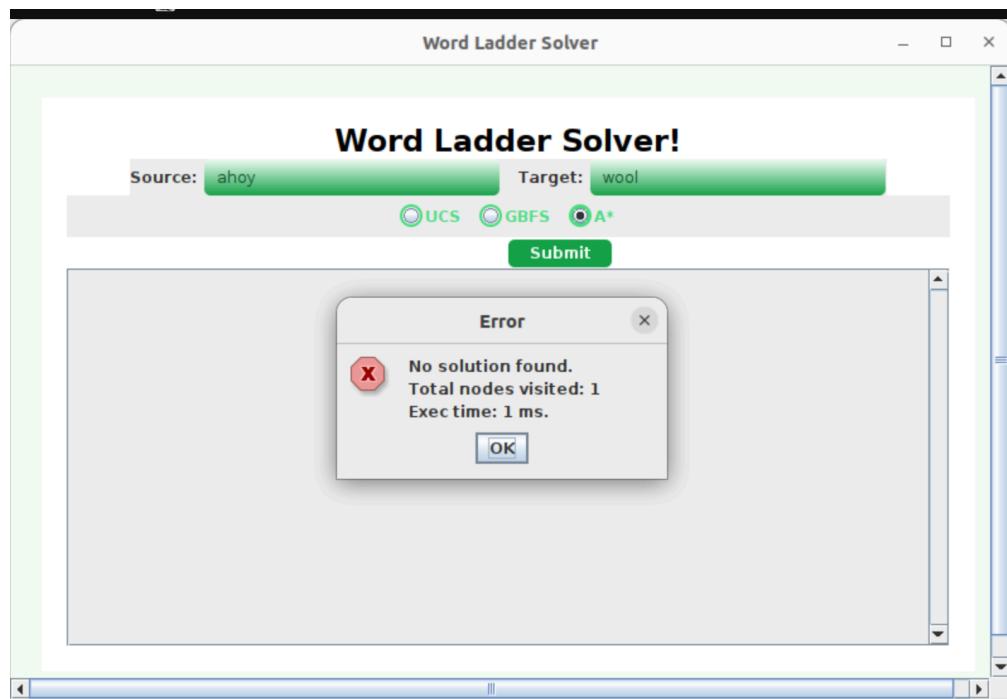
6. Eats ke west



Gambar 27 Test Case eats ke west

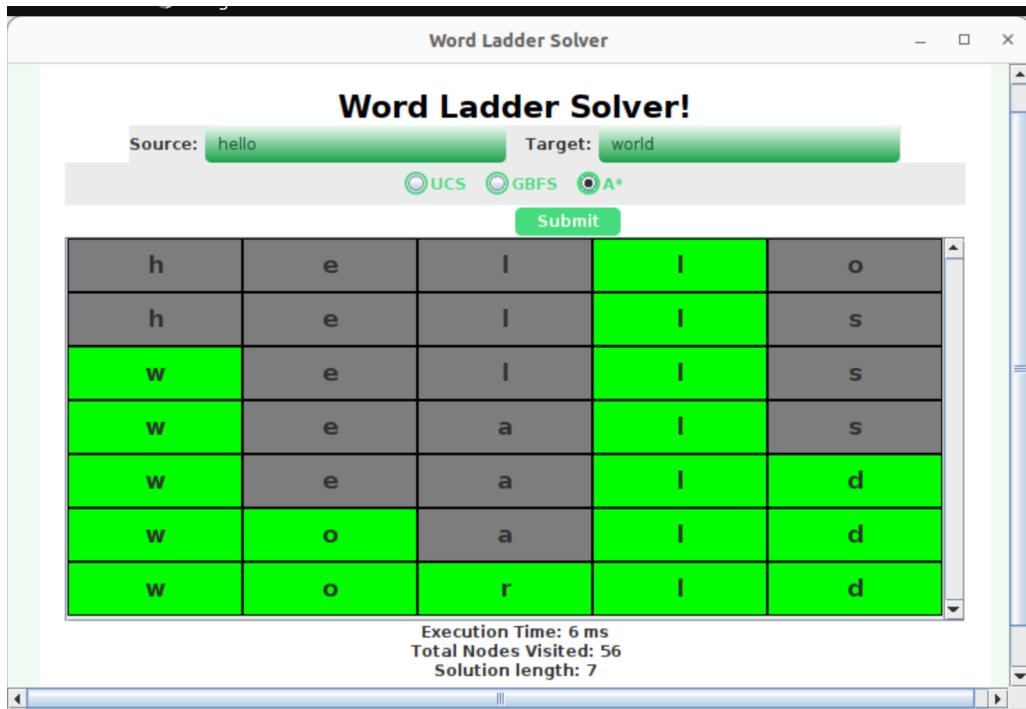
#### 5. 4 Testing pada Algoritma A\*

1. Ahoy ke wool



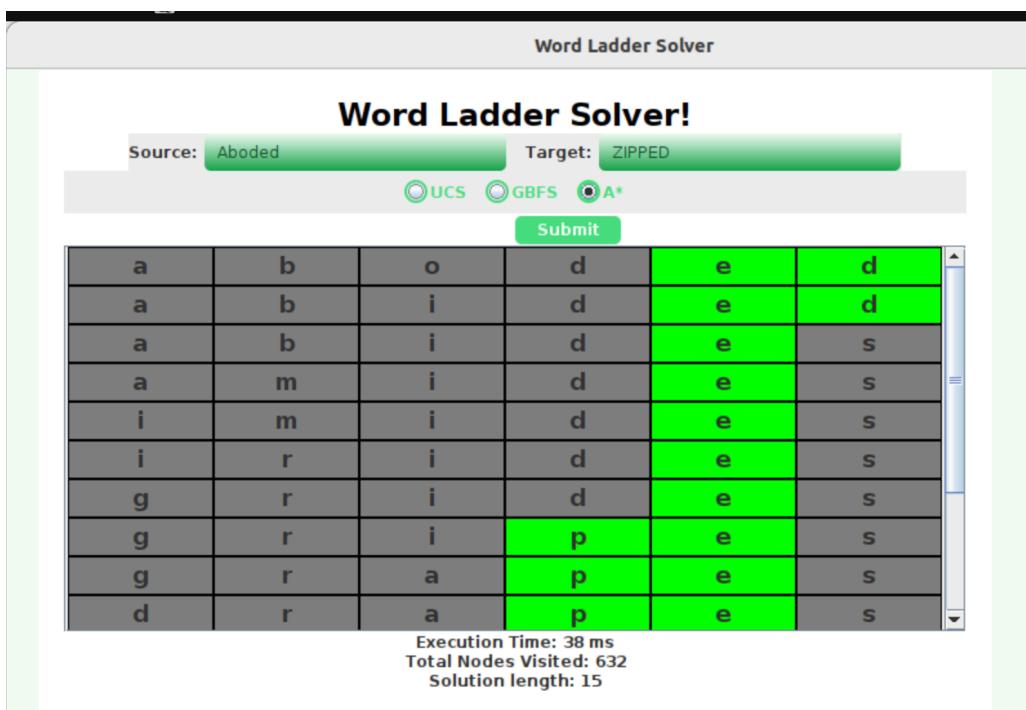
Gambar 28 Test Case ahoy ke wool

2. hello ke world

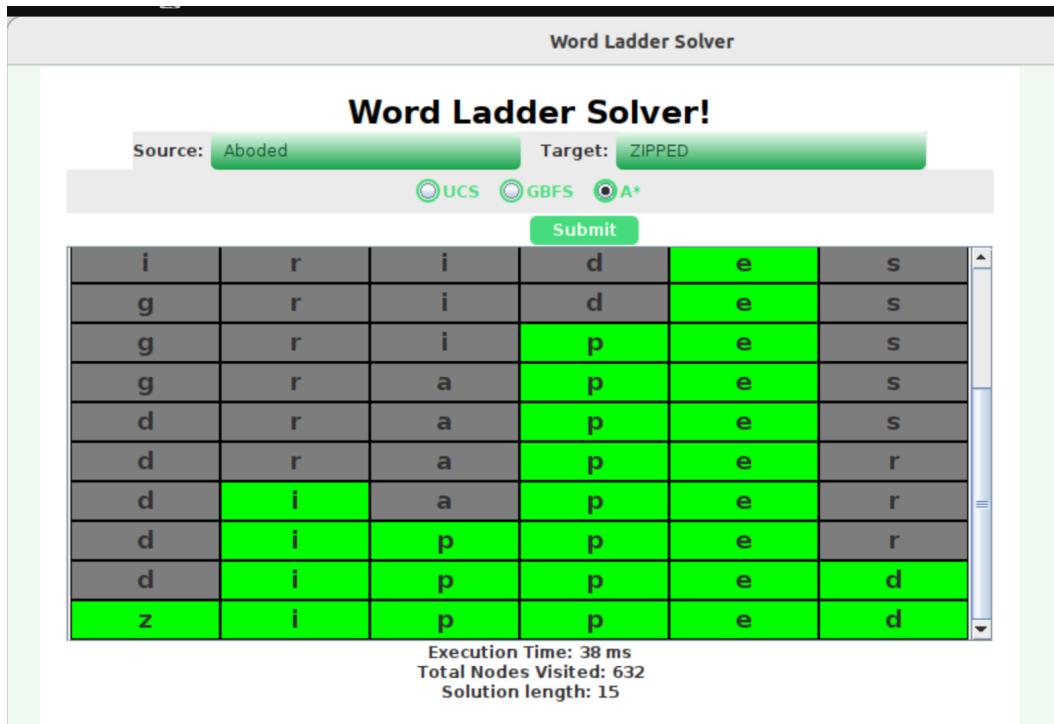


Gambar 29 Test Case hello ke world

### 3. Aboded ke ZIPPED

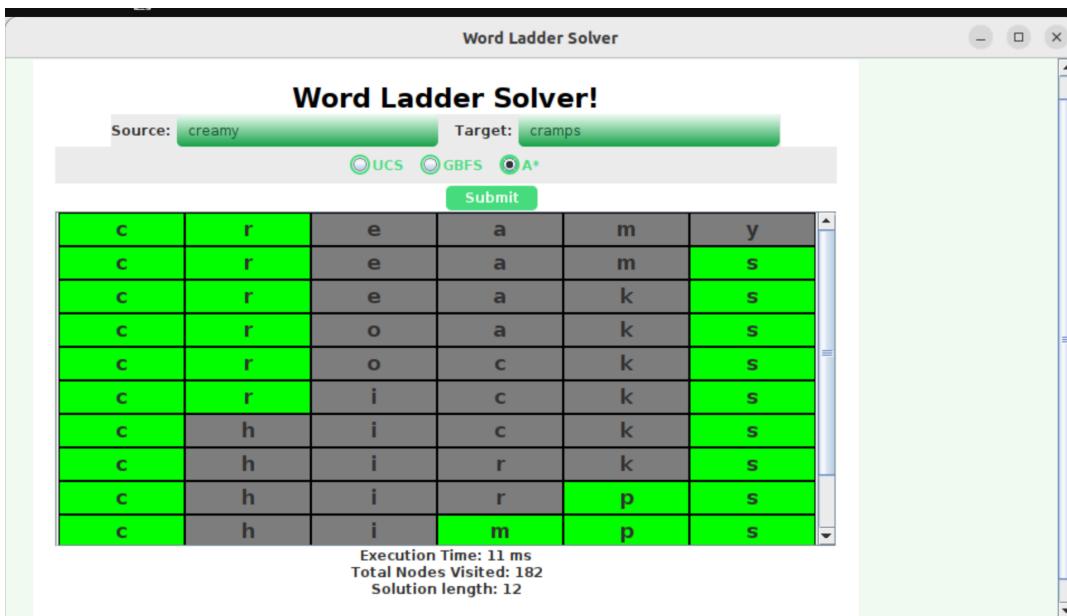


Gambar 30 Test Case aboded ke zipped

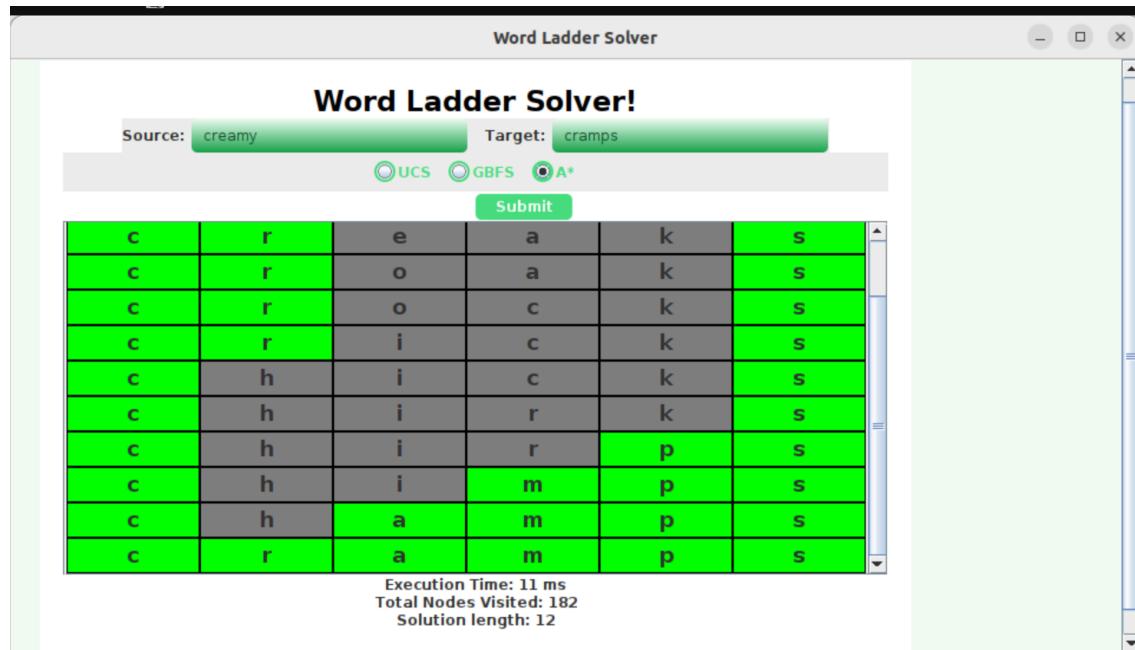


Gambar 31 Test Case aboded ke zipped

4. creamy ke cramps

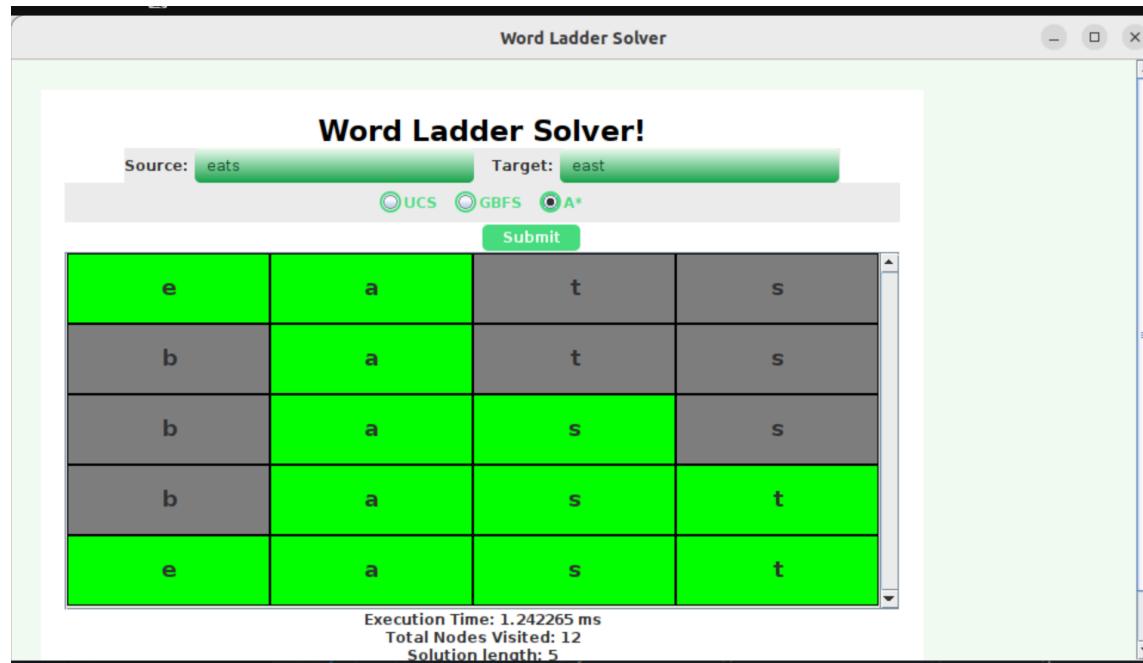


Gambar 32 Test Case creamy ke cramps



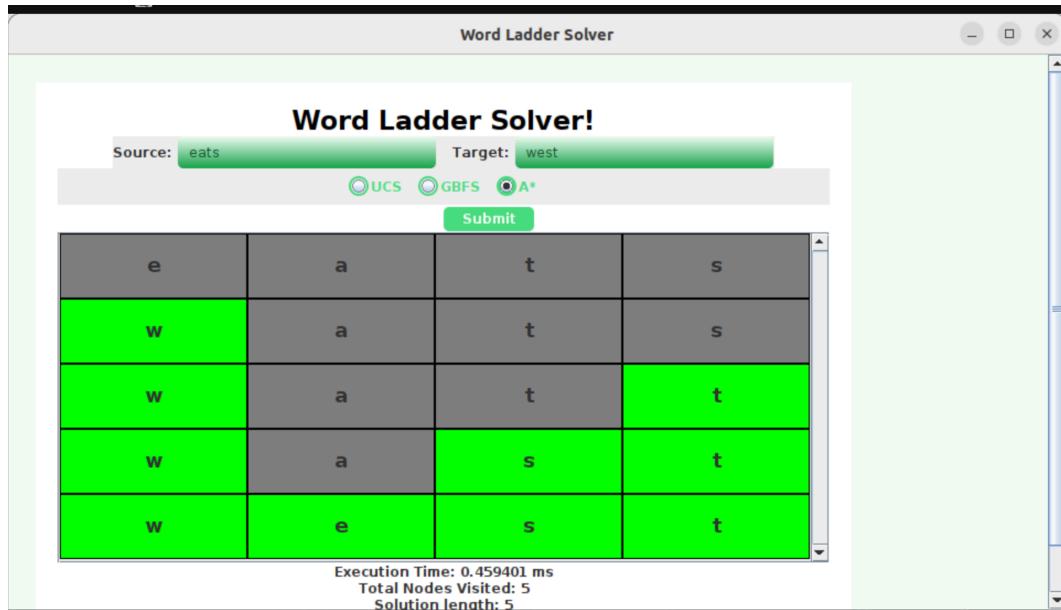
Gambar 33 Test Case creamy ke cramps

5. eats ke east



Gambar 34 Test Case eats ke east

6. eats ke west



**Gambar 35** Test Case eats ke west

## 5. 5 Testing Perbandingan Hasil dari Penggunaan A\* dengan UCS

Telah diimplementasikan testing yang terautomasi (*stress test*) dengan memilih dua kata random dan membandingkan hasilnya dengan menggunakan A\* dan UCS.

```
Checking between acmic and yoghs: PASSED! Both doesnt have solution
Checking between acne and yogh: PASSED! 10 10
Checking between acned and yoghs: PASSED! 10 10
Checking between acnes and yogee: PASSED! Both doesnt have solution
Checking between acock and yogee: PASSED! Both doesnt have solution
Checking between acold and yogas: PASSED! 14 14
Checking between acorn and yodle: PASSED! Both doesnt have solution
Checking between acre and yock: PASSED! 10 10
Checking between acred and yobbo: PASSED! Both doesnt have solution
Checking between acres and yobbo: PASSED! Both doesnt have solution
Checking between acrid and ylems: PASSED! Both doesnt have solution
Checking between act and yid: PASSED! 4 4
Checking between acta and yids: PASSED! 5 5
Checking between acted and yield: PASSED! Both doesnt have solution
Checking between actin and yeuky: PASSED! Both doesnt have solution
Checking between actor and yerks: PASSED! Both doesnt have solution
Checking between acts and yens: PASSED! 5 5
Checking between acute and yell: PASSED! 8 8
Checking between acyl and yell: PASSED! 11 11
Checking between acyls and yelks: PASSED! 11 11
Checking between ad and ye: PASSED! 3 3
Checking between adage and yelks: PASSED! Both doesnt have solution
Checking between adapt and yechy: PASSED! Both doesnt have solution
Checking between add and yea: PASSED! 6 6
Checking between addax and years: PASSED! Both doesnt have solution
Checking between added and years: PASSED! 9 9
Checking between adder and yearn: PASSED! 10 10
Checking between addle and yawps: PASSED! Both doesnt have solution
Checking between adds and yeah: PASSED! 7 7
Checking between adeem and yawns: PASSED! Both doesnt have solution
Checking between adept and yawed: PASSED! Both doesnt have solution
Checking between adieu and yarns: PASSED! Both doesnt have solution
Checking between adios and yarns: PASSED! 13 13
Checking between adit and yarn: PASSED! 8 8
Checking between adits and yarns: PASSED! 12 12
Checking between adman and yapon: PASSED! 14 14
Checking between admen and yapon: PASSED! 15 15
Checking between admit and yanks: PASSED! Both doesnt have solution
Checking between admix and yangs: PASSED! Both doesnt have solution
Checking between ado and yam: PASSED! 6 6
```

## 6      Analisis Data

Dalam bagian Testing, telah dilakukan 6 percobaan berbeda kepada masing-masing algoritma, dan didapatkan data pada tabel berikut.

	Algoritma	Total nodes visited	Execution time (ms)	Solution length
Ahoy -> wool	UCS	1	13	-
	GBFS	2	0.7	-
	A*	1	1	-
Hello -> world	UCS	2912	114	7
	GBFS	399	7	16
	A*	56	6	7
Aboded -> zipped	UCS	4400	130	15
	GBFS	24	1.2	16
	A*	632	38	15
Creamy -> cramps	UCS	1548	52	12
	GBFS	83	4.4	14
	A*	182	11	12
Eats -> east	UCS	1179	18	5
	GBFS	11	0.2	7
	A*	12	1.2	5
Eats -> west	UCS	2004	31	5
	GBFS	5	0.13	5
	A*	5	0.46	5

**Tabel 1** Data Hasil Test case

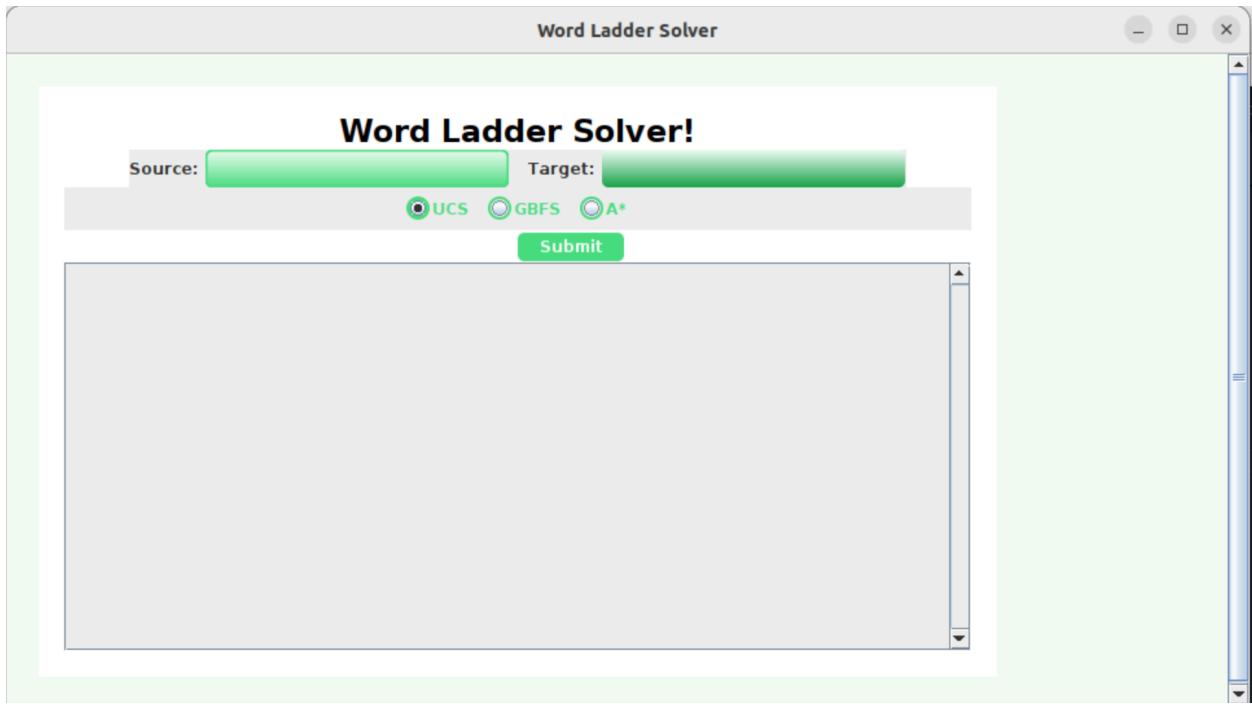
Dari tabel di atas, dapat diamati bahwa hasil panjang solusi antara algoritma UCS dan A\* selalu sama, namun A\* memiliki waktu eksekusi yang lebih cepat, hal ini karena A\* memiliki fungsi

heuristik yang mengakibatkan A\* lebih mementingkan untuk meng-expand node yang lebih dekat dengan tujuan sehingga *total nodes visited* dan waktu eksekusinya lebih sedikit dibandingkan dengan algoritma UCS. Data ini juga membuktikan bahwa fungsi heuristik yang digunakan sudah *admissible*. Namun optimalitas tidak ditunjukkan oleh greedy, algoritma greedy sering menunjukkan solusi yang lebih panjang dibandingkan solusi *shortest path*. Tetapi greedy memiliki waktu eksekusi dan jumlah simpul ekspan yang jauh lebih sedikit dibandingkan dengan algoritma A\*. Hal ini karena GBFS secara naif hanya mengekspansi simpul yang kelihatannya lebih dekat oleh fungsi heuristik.

Lebih lanjut, dapat diamati juga bahwa jumlah simpul yang terekspansi pada UCS secara konsisten selalu lebih besar dibandingkan dengan kedua algoritma lainnya, kemudian diikuti oleh A\* dan terakhir GBFS. Katakanlah setiap simpul memiliki *branching factor* b, maka batas atas kapasitas queue pada saat pemrosesan dapat mencapai  $b \cdot T$  dengan T adalah jumlah simpul yang diperiksa. Sehingga apabila jumlah simpul yang terekspansi untuk UCS > A\* > GBFS maka haruslah batas atas jumlah elemen pada queue menjadi  $E_{UCS} > E_{A^*} > E_{GBFS}$  dengan E adalah jumlah maksimal elemen pada queue pada saat pemrosesan. Akibatnya memori yang dibutuhkan untuk masing-masing algoritma dari terkecil ke terbesar adalah algoritma GBFS, A\*, kemudian UCS.

## 7      Implementasi Bonus

GUI diimplementasikan menggunakan java swing dengan tampilan awal sebagai berikut



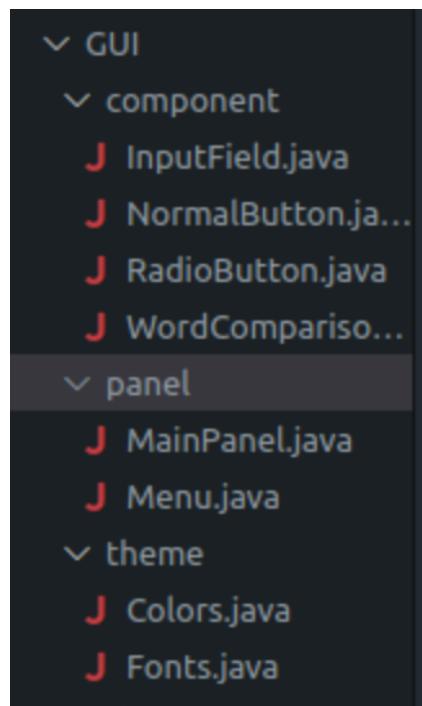
**Gambar 36** Tampilan Awal GUI

dan tampilan saat menunjukkan solusi sebagai berikut



Gambar 36 Tampilan Awal GUI

Implementasi GUI menggunakan telah prinsip modularitas dan *Maintainability* dengan membaginya menjadi 3 poin utama, yaitu *component*, *panel*, dan *theme*.



Gambar 37 Struktur folder GUI

Component terdiri atas komponen-komponen yang pada saat *development* atapun nantinya akan *reusable*, yaitu InputField, NormalButton, RadioButton, dan WordComparison. Panel terdiri dari Menu.java sebagai base class untuk panel-panel lainnya, dalam hal ini hanya MainPanel.java. Dan Theme sebagai penyimpan tema-tema yang dipakai berulang kali di dalam GUI, seperti color dan font.

## 8      *Lampiran*

### 8. 1 Pemenuhan Spesifikasi Tugas Kecil

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

### 8. 2 Pranala Github

[https://github.com/ganadipa/Tucil3\\_13522066](https://github.com/ganadipa/Tucil3_13522066)