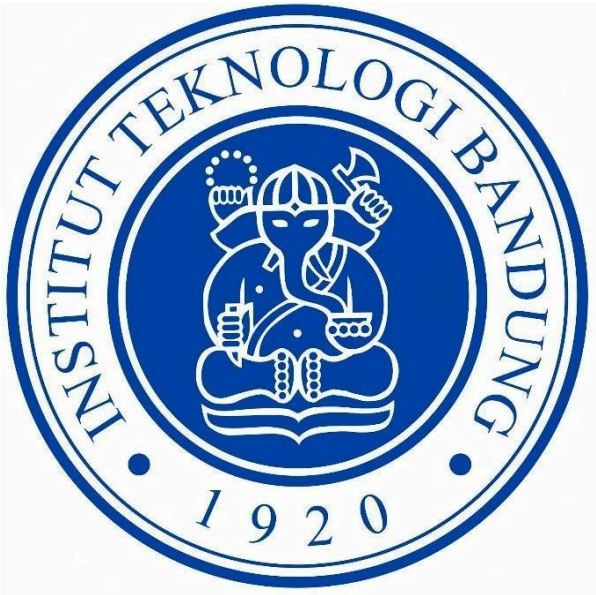
LAPORAN TUGAS KECIL

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

Disajikan untuk memenuhi salah satu tugas kecil Mata Kuliah IF2211 Strategi Algoritma yang diampu oleh:

Dr. Nur Ulfa Maulidevi, S.T., M.Sc.



Disusun oleh:

Nelsen Putra (13520130)

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2024

A. Analisis dan Implementasi Algoritma UCS

Algoritma *Uniform Cost Search* (UCS) adalah algoritma pencarian lintasan terpendek antara dua simpul pada sebuah graf dengan mempertimbangkan biaya (*cost*) terkecil untuk mencapai simpul tujuan dari simpul awal. Jadi, algoritma UCS ini akan memberikan solusi yang pasti optimal berdasarkan *cost* yang paling kecil. Dalam penyelesaian permain Word Ladder menggunakan algoritma UCS, terdapat rincian langkah-langkah yang ditempuh untuk menemukan solusi optimal, sebagai berikut.

1. Definisikan terlebih dahulu kata awal (*starting word*) untuk dijadikan sebagai simpul awal pencarian dan kata tujuan (*ending word*) yang dijadikan sebagai simpul akhir atau tujuan pencarian dalam permainan ini.
2. Lakukan pengecekan atau validasi terhadap masukan dari pengguna terhadap *starting word* dan *ending word*, apakah kedua kata memiliki panjang kata yang sama. Jika kedua kata memiliki panjang kata yang berbeda, maka algoritma tidak akan melanjutkan pencarian dan menghentikan langkahnya.
3. Definisikan sebuah *priority queue* yang mengurutkan komponen di dalamnya berdasarkan *cost* yang telah dilalui hingga simpul tertentu berdasarkan *cost* yang terendah hingga tertinggi. Maka dari itu, pemrosesan selanjutkan akan dilanjutkan berdasarkan *cost* yang terendah terlebih dahulu sehingga sesuai dengan prinsip dari algoritma UCS.
    a. Sebagai tambahan informasi bahwa *cost* dalam UCS dapat dihitung menggunakan $f(n) = g(n)$, dengan $g(n)$ merupakan *cost* dari jalan yang telah ditempuh ke suatu simpul dari simpul akar. Semisal, perjalanan sebuah algoritma A-B-C, maka $g(n)$ merupakan *cost* perjalanan A-B ditambah dengan *cost* perjalanan B-C. Di sini pula, *cost* yang kecil menandakan kedalaman perjalanan yang lebih pendek, yang berarti solusi perjalanan yang didapatkan akan diusahakan sesingkat mungkin oleh algoritma.
    b. Dalam permainan *Word Ladder* ini, bisa dikatakan bahwa *cost* perjalanan sebuah simpul dapat dihitung menggunakan *depth* dimana simpul tersebut dilalui pada sebuah perjalanan. Semisal, perjalanan algoritmanya adalah A-B-C, maka simpul C akan disimpul dengan $g(n) = 2$, karena *depth* 2. Penyimpanan informasi $g(n)$ ini akan sangat penting karena jika terjadi perjalanan lain seperti A-C-B, maka C masih dipertimbangkan karena memiliki *depth* yang lebih awal sehingga berkemungkinan menjadi lebih optimal. Namun, jika ditemukan simpul C di atas kedalaman 3, maka simpul C tersebut tidak akan dipertimbangkan lagi sebagai calon solusi.
    c. Dengan itu, maka didapatkan alasan mengapa dilakukan pengurutan pada *priority queue* dengan aturan pengurutan dari *depth* / $g(n)$ terkecil hingga terbesar. Jika dilihat pada algoritma yang dibuat untuk menyelesaikan permainan ini, $g(n)$ tidak lagi disebutkan secara eksplisit atau

didefinisikan, melainkan hanya digunakan informasi kedalaman yang telah disimpan pada class *Node* yang menandakan simpul-simpul.

d. *Priority queue* diinisialisasi dengan satu simpul / *Node* yang akan memicu perulangan yang terjadi pada langkah 4, yaitu dengan inisialisasi pemasukan *node* atau simpul kata mulai (*starting word*) ke dalam *priority queue*.

4. Selama *priority queue* tidak kosong, algoritma akan selalu mengunjungi simpul dengan *cost* yang paling rendah dan dilakukan pengecekan terhadap *node* tersebut apakah merupakan *goal node* atau bukan.
    a. Jika *node* sekarang merupakan *goal node*, maka algoritma akan berhenti dan mengembalikan simpul-simpul yang dilalui oleh algoritma untuk mencapai *goal node* tersebut.
    b. Jika *node* bukan merupakan *goal node*, maka algoritma akan dilanjutkan ke langkah selanjutnya.
5. Lakukan pembangkitan *node* anak dari *node* yang sedang dikunjungi oleh algoritma. Maka dari itu, *node* yang sekarang sedang dikunjungi dari oleh algoritma akan menjadi *parent node*. Dalam setiap pembangkitan anaknya dilakukan secara *bruteforce* untuk mencari kata-kata yang *valid* dalam kamus untuk dijadikan sebagai *node*. Jika *node* anak tersebut tidak pernah dikunjung maka *node* anak tersebut dimasukkan ke *priority queue* karena layak untuk dijadikan pertimbangan solusi. Setelah itu, maka algoritma akan kembali ke langkah 4.
6. Algoritma akan mengulangi langkah 4-5 hingga *priority queue* kosong atau algoritma menemukan solusi. Jika *priority queue* kosong, artinya tidak ada lintasan atau perjalanan yang mungkin bagi simpul awal untuk mencapai simpul tujuan.

B. Analisis dan Implementasi Algoritma *Greedy Best First Search*

Algoritma *Greedy Best First Search* (GBFS) adalah algoritma pencarian untuk mencari solusi atau lintasan yang mungkin untuk dilalui agar simpul awal dapat mencapai simpul akhir tanpa memedulikan apakah sebuah solusi akan optimal atau tidak. Algoritma ini menentukan langkah atau simpul yang akan dikunjungi selanjutnya secara heuristik. Heuristik yang digunakan untuk menyelesaikan permainan *Word Ladder* dengan algoritma GBFS ini adalah dengan mencari *hamming distance* terkecil antara satu simpul dengan simpul yang dikunjungi. Sebagai informasi tambahan, *hamming distance* pada algoritma ini berarti banyak perbedaan huruf antara satu kata dengan kata yang lain. Contoh kasusnya seperti “ABCD” dengan “EFGH”, maka *hamming distance*-nya adalah 4 karena terdapat 4 perbedaan huruf pada kedua kata tersebut pada posisi yang berurutan. Dalam penyelesaian permain Word Ladder menggunakan algoritma GBFS, terdapat rincian langkah-langkah yang ditempuh untuk menemukan solusi optimal, sebagai

berikut.

1. Definisikan terlebih dahulu kata awal (*starting word*) untuk dijadikan sebagai simpul awal pencarian dan kata tujuan (*ending word*) yang dijadikan sebagai simpul akhir atau tujuan pencarian dalam permainan ini.
2. Lakukan pengecekan atau validasi terhadap masukan dari pengguna terhadap *starting word* dan *ending word*, apakah kedua kata memiliki panjang kata yang sama. Jika kedua kata memiliki panjang kata yang berbeda, maka algoritma tidak akan melanjutkan pencarian dan menghentikan langkahnya.
3. Definisikan sebuah *priority queue* yang mengurutkan komponen di dalamnya berdasarkan *estimation cost h(n)* yang terendah hingga tertinggi (pengurutan berdasarkan *hamming distance* terendah hingga tertinggi). Maka dari itu, pemrosesan selanjutkan akan dilanjutkan berdasarkan *estimation cost heuristic* yang terendah terlebih dahulu sehingga sesuai dengan prinsip dari algoritma GBFS.
    a. Sebagai tambahan informasi bahwa *cost* dalam GBFS dapat dihitung menggunakan *f(n) = h(n)*, dengan *h(n)* merupakan estimasi *cost* yang diperlukan dari simpul n untuk menuju ke simpul tujuan, yang pada kasus ini *h(n)* dihitung menggunakan heuristik *hamming distance*. Semisal, perjalanan sebuah algoritma dengan kata awal “as” dan kata tujuan “if”, maka *h(n)* akan bernilai 2 karena *hamming distance* kedua kata tersebut 2.
    b. Algoritma tidak akan mempertimbangkan faktor lain sama sekali selain *heuristic hamming distance* untuk satu simpul menuju ke simpul lainnya. Oleh karena itu, jika bertemu dengan kasus kata awal “as” yang berkemungkinan mengunjungi simpul “is” (*hamming-distance : 1)*, “an (*hamming-distance : 1)*”, “ax (*hamming-distance : 1)*”, ataupun kata lain. Maka, penentuan pemrosesan yang mana terlebih dahulu akan ditentukan oleh *priority queue* itu sendiri, karena tidak dapat dipilih secara khusus semisal *is* karena sepertinya memiliki probabilitas yang tinggi untuk mencapai kata tujuan. Jadi, algoritma ini benar-benar secara buta mengandalkan *hamming distance* setiap simpul dalam pengambilan keputusannya.
    c. *Priority queue* diinisialisasi dengan satu simpul / *Node* yang akan memicu perulangan yang terjadi pada langkah 4, yaitu dengan inisialisasi pemasukan *node* atau simpul kata mulai (*starting word*) ke dalam *priority queue*.
4. Selama *priority queue* tidak kosong, algoritma akan selalu mengunjungi simpul dengan *cost* yang paling rendah dan dilakukan pengecekan terhadap *node* tersebut apakah merupakan *goal node* atau bukan.
    a. Jika *node* sekarang merupakan *goal node*, maka algoritma akan berhenti dan mengembalikan simpul-simpul yang dilalui oleh algoritma untuk

mencapai *goal node* tersebut.

b. Jika *node* bukan merupakan *goal node*, maka algoritma akan dilanjutkan ke langkah selanjutnya.

5. Lakukan pembangkitan *node* anak dari *node* yang sedang dikunjungi oleh algoritma. Maka dari itu, *node* yang sekarang sedang dikunjungi dari oleh algoritma akan menjadi *parent node*. Dalam setiap pembangkitan anaknya dilakukan secara *bruteforce* untuk mencari kata-kata yang *valid* dalam kamus untuk dijadikan sebagai *node*. Jika *node* anak tersebut tidak pernah dikunjungi, maka *node* anak tersebut dimasukkan ke *priority queue* karena layak untuk dijadikan pertimbangan solusi. Setelah itu, maka algoritma akan kembali ke langkah 4.
6. Algoritma akan mengulangi langkah 4-5 hingga *priority queue* kosong atau algoritma menemukan solusi. Jika *priority queue* kosong, artinya tidak ada lintasan atau perjalanan yang mungkin bagi simpul awal untuk mencapai simpul tujuan.

C. Analisis dan Implementasi Algoritma A*

Algoritma *A** adalah algoritma pencarian untuk mencari perjalanan atau solusi yang paling optimal atau jalur yang terpendek antara simpul awal dengan simpul tujuan pada sebuah graf. Berbeda dengan algoritma UCS dan GBFS, yang mana algoritma UCS mempertimbangkan *g(n)* dan algoritma GBFS yang mempertimbangkan *h(n*). Algoritma ini mempertimbangkan keduanya dalam pencarian perjalanan yang optimal dengan menggunakan konsep perhitungan *cost f(n) = g(n) + h(n)*. Dalam penyelesaian permainan *Word Ladder* ini dengan menggunakan algoritma *A** dapat dirincikan dengan langkah-langkah sebagai berikut.

1. Definisikan terlebih dahulu kata awal (*starting word*) untuk dijadikan sebagai simpul awal pencarian dan kata tujuan (*ending word*) yang dijadikan sebagai simpul akhir atau tujuan pencarian dalam permainan ini.
2. Lakukan pengecekan atau validasi terhadap masukan dari pengguna terhadap *starting word* dan *ending word*, apakah kedua kata memiliki panjang kata yang sama. Jika kedua kata memiliki panjang kata yang berbeda, maka algoritma tidak akan melanjutkan pencarian dan menghentikan langkahnya.
3. Definisikan sebuah *priority queue* yang mengurutkan komponen di dalamnya berdasarkan *cost f(n)* yang terendah hingga tertinggi. Maka dari itu, pemrosesan selanjutkan akan dilanjutkan berdasarkan simpul dengan *cost f(n)* yang terendah terlebih dahulu sehingga sesuai dengan prinsip dari algoritma A*.
    a. Sebagai tambahan informasi bahwa *cost* dalam A* dapat dihitung menggunakan *f(n) = g(n) + h(n)*, dengan *h(n)* merupakan estimasi *cost* yang diperlukan dari simpul n untuk menuju ke simpul tujuan dan *g(n)* merupakan *cost* yang telah dikumpulkan untuk mencapai simpul tertentu.

Pada kasus ini *h(n)* dihitung menggunakan heuristik *hamming distance* dan *g(n)* merupakan *depth* simpul tersebut ditemukan. Semisal, sebuah kata "fly" ditemukan pada simpul berkedalaman 1, maka *g(n)* = 1 dan ingin menuju ke kata tujuan "fit", maka *heuristic cost h(n)* = 2 dengan perhitungan *hamming distance*. Jadi, total *cost f(n)* pada simpul kata "fly" adalah *g(n) + h(n) = 1 + 2 = 3*.

b. Dengan konsep seperti ini, secara tidak langsung algoritma A* merupakan sebuah algoritma pencarian yang menggabungkan kedua algoritma UCS dan GBFS menjadi satu, sehingga solusi yang ditemukan akan menjadi optimal / terpendek seperti UCS, namun dengan kunjungan *node* yang lebih sedikit karena dibantu dengan bantuan *heuristic hamming distance* seperti pada algoritma GBFS.

c. *Priority queue* diinisialisasi dengan satu simpul / *Node* yang akan memicu perulangan yang terjadi pada langkah 4, yaitu dengan inisialisasi pemasukan *node* atau simpul kata mulai (*starting word*) ke dalam *priority queue*.

4. Selama *priority queue* tidak kosong, algoritma akan selalu mengunjungi simpul dengan *cost* yang paling rendah dan dilakukan pengecekan terhadap *node* tersebut apakah merupakan *goal node* atau bukan.
    a. Jika *node* sekarang merupakan *goal node*, maka algoritma akan berhenti dan mengembalikan simpul-simpul yang dilalui oleh algoritma untuk mencapai *goal node* tersebut.
    b. Jika *node* bukan merupakan *goal node*, maka algoritma akan dilanjutkan ke langkah selanjutnya.
5. Lakukan pembangkitan *node* anak dari *node* yang sedang dikunjungi oleh algoritma. Maka dari itu, *node* yang sekarang sedang dikunjungi dari oleh algoritma akan menjadi *parent node*. Dalam setiap pembangkitan anaknya dilakukan secara *bruteforce* untuk mencari kata-kata yang *valid* dalam kamus untuk dijadikan sebagai *node*. Jika *node* anak tersebut tidak pernah dikunjungi atau pernah dikunjungi namun dalam kedalaman yang lebih dalam, maka *node* anak tersebut dimasukkan ke *priority queue* karena layak untuk dijadikan pertimbangan solusi. Setelah itu, maka algoritma akan kembali ke langkah 4.
6. Algoritma akan mengulangi langkah 4-5 hingga *priority queue* kosong atau algoritma menemukan solusi. Jika *priority queue* kosong, artinya tidak ada lintasan atau perjalanan yang mungkin bagi simpul awal untuk mencapai simpul tujuan.

D. Analisis Perbandingan Ketiga Algoritma

Terlihat bahwa dari implementasi setiap algoritma yang dijelaskan pada bagian-bagian sebelumnya bahwa ketiga algoritma yang di-implementasi tentu saja

memiliki perbedaan atau cirinya masing-masing yang menyebabkan ketiganya merupakan algoritma yang berbeda. Sistem iterasi yang dilakukan untuk ketiga algoritma tersebut hampir sama, namun yang membedakan hanya berupa cara perhitungan *cost* yang dilakukan. Dalam algoritma *Uniform Cost Search* (UCS), perhitungan *cost* dilakukan dengan *f(n) = g(n)* dan algoritma *Best First Search* (GBFS), perhitungan *cost* dilakukan dengan *f(n) = h(n)*. Sedangkan, algoritma *A-Star* (A*) melakukan perhitungan *cost* dengan *f(n) = g(n) + h(n)*. Dengan deskripsi bahwa *f(n)* merupakan *total cost* yang dipertimbangkan oleh sebuah algoritma, *g(n)* merupakan *total cost* yang telah ditempuh untuk mencapai suatu simpul (pada persoalan kali ini, dapat ditandai dengan kedalaman sebuah simpul), dan *h(n)* merupakan *cost heuristic* estimasi yang dihitung dari simpul n menuju simpul akhir (*goal node*).

Pada algoritma A*, heuristik yang digunakan bersifat admissible untuk semua kasus karena heuristik yang digunakan merupakan *hamming distance* dan setiap kali langkah yang dilakukan algoritma hanya maksimal mengubah satu huruf pada kata simpul tersebut. Dengan itu, maka dapat disimpulkan bahwa *hamming distance* yang selalu ≥1 jika simpul belum mencapai *goal node* akan selalu lebih kecil dibandingkan dengan banyak langkah algoritma yang diperlukan untuk mencapai *goal node* yang sebenarnya. Oleh karena itu, algoritma A* dengan heuristik ini akan dijamin selalu bersifat *admissible* karena *h(n) <= h*(n)* karena *hamming distance* sudah merupakan estimasi yang sifatnya *best case* sehingga tidak mungkin akan terjadi overestimasi yang melebihi langkah yang sebenarnya untuk mencapai *goal node*. Karena konsistensi algoritma yang selalu *admissible*, maka algoritma ini bersifat ideal dan *optimistic*.

Jika dilakukan perbandingan antara algoritma UCS dan GBFS, yang jika dilihat pada pembahasan implementasi pada bagian sebelumnya menunjukkan bahwa langkah yang berbeda hanya terjadi pada langkah nomor 3. Ini menandakan bahwa kedua algoritma hampir memiliki mekanisme yang mirip, namun hanya sedikit perbedaan. Cara kedua algoritma tersebut membangkitkan anaknya cenderung sama, namun perbedaan terdapat dalam urutan *node-node* yang akan diproses selanjutnya oleh algoritma dalam *priority queue*. Jika *priority queue* kedua algoritma tersebut berbeda, maka sudah dapat dipastikan urutan pemrosesan serta *path* yang dihasilkan akan beda pula. Akibat UCS yang memperhatikan dan mementingkan pemrosesan *node* yang memiliki *depth* atau *cost g(n)* yang rendah, maka akan cenderung menghasilkan solusi *path* yang terpendek atau paling optimal dibandingkan dengan algoritma GBFS yang cenderung langsung mengunjungi sebuah *node* jika memiliki *hamming distance* atau *heuristic cost h(n)* yang rendah tanpa mempertimbangkan kedalaman solusi. Jadi, *path* yang dihasilkan kedua algoritma tersebut dipastikan berbeda dan algoritma UCS akan menghasilkan *path* yang lebih singkat dibandingkan dengan algoritma GBFS.

Secara teoritis, algoritma A* akan lebih efisien dibandingkan dengan algoritma UCS pada kasus *Word Ladder* ini, yang dapat dilihat pada visualisasi salah satu kasus yang membuktikan bahwa algoritma A* lebih unggul dibandingkan dengan algoritma

UCS. Algoritma UCS membutuhkan lebih banyak langkah daripada algoritma A* hanya dalam melakukan pencarian simpul pada kedalaman tertentu. Ini disebabkan A* sudah menggunakan konsep seperti UCS yang mempertimbangkan kedalaman, namun A* juga mempertimbangkan *heuristic cost*-nya sehingga tidak melakukan langkah yang sia-sia untuk mengunjungi *node* yang memiliki *hamming distance* lebih tinggi sehingga pencarian menjadi lebih jauh dari *goal node* yang menyebabkan langkah terbuang secara sia-sia. Dengan mempertimbangkan *h(n)*, algoritma A* berhasil melakukan penghematan langkah yang masif dibandingkan algoritma UCS sehingga algoritma A* lebih efisien.

Secara teoritis, algoritma *Greedy Best First Search* (GBFS) tidak menjamin solusi yang optimal seperti pada algoritma UCS dan A*. Ini disebabkan karena algoritma GBFS hanya mempertimbangkan *h(n)* pada pengambilan keputusannya sehingga tidak mempertimbangkan *g(n)* atau kedalaman solusi sehingga solusi menjadi tidak optimal dengan kedalaman yang tidak menentu. Algoritma ini sifatnya tidak mempedulikan apapun, hanya mementingkan ditemukan sebuah solusi dari simpul awal menuju simpul tujuan sehingga kemungkinan ini akan menghemat waktu eksekusi karena algoritma ini tidak terlalu menghabiskan waktunya untuk melakukan pertimbangan untuk melakukan kunjungan *node* lainnya untuk mendapatkan solusi optimal.

E. *Source Code* Program

Source code program *Word Ladder Solver* ditulis dalam bahasa Java dengan isi kode sebagai berikut.

1. File Dictionary.java

```
package utils;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashSet;

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

   public static void load_word(String filename) {
       try (BufferedReader reader = new BufferedReader(new
FileReader(filename))) {
           String line;
           while ((line = reader.readLine()) != null) {
               words.add(line.trim());
           }
       } catch (IOException e) {
```

```
            System.out.println("Failed to load dictionary: " +
e.getMessage());
            e.printStackTrace();
        }
    }

    public static boolean word_valid_checker(String word) {
        return words.contains(word);
    }

    public static void print_dictionary() {
        System.out.println(words.toString());
    }
}
```

Pada file ini, terdapat sebuah kelas bernama Dictionary yang berfungsi untuk menyimpan kata-kata yang telah didefinisikan pada sebuah file dictionary tertentu yang berada dalam format *text-file* dengan menggunakan *method load_word*. Kemudian, juga terdapat *method word_valid_checker* yang berfungsi untuk melakukan pengecekan apakah sebuah kata terdefinisi dalam kamus. Metode-metode serta atribut dalam kelas ini dibuat secara statik sehingga tidak perlu di instansiasi dan inisialisasi dictionary dapat dipanggil secara langsung melalui kelas tersebut.

2. File Utils.java

```
package utils;

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

public interface Utils {
    default List<String> find_word_possibility(String word) {
        List<String> valid_words = new ArrayList<>();
        char[] chars = word.toCharArray();
        for (int i = 0; i < chars.length; i++) {
            char original = chars[i];
            for (char c = 'a'; c <= 'z'; c++) {
                if (c != original) {
                    chars[i] = c;
                    String newWord = new String(chars);
                    if (Dictionary.word_valid_checker(newWord)) {
```

```
                        valid_words.add(newWord);
                    }
                }
            }
            chars[i] = original;
        }
        return valid_words;
    }

    default int count_mismatch_letter(String current, String target) {
        int count = 0;
        for (int i = 0; i < current.length(); i++) {
            if (current.charAt(i) != target.charAt(i)) {
                count += 1;
            }
        }
        return count;
    }

    default int count_same_letter(String current, String target) {
        int count = 0;
        for (int i = 0; i < current.length(); i++) {
            if (current.charAt(i) == target.charAt(i)) {
                count += 1;
            }
        }
        return count;
    }
}
```

File *Utils.java* memiliki *interface Utils* yang berisi metode-metode atau fungsi yang sering digunakan oleh beberapa kelas. Dengan adanya *interface* ini, fungsi tidak perlu didefinisikan secara berulang-ulang pada kelas yang berbeda, cukup *implements interface* ini. Pada *interface* ini terdapat metode *find_word_possibility* yang merupakan metode untuk melakukan *bruteforce* atau mencari kata yang mungkin dari sebuah kata, metode *count_mismatch_letter* merupakan metode untuk menghitung banyaknya huruf yang berbeda di antara dua kata dan metode *count_same_letter* untuk menghitung banyaknya huruf yang sama di antara dua kata. Ketiga fungsi ini merupakan fungsi yang sering dipanggil pada file dan kelas yang berbeda sehingga diabstraksi menjadi file tersendiri.

3. File WordLadder.java

```java
package algorithm;

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

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

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

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

   public long getExecutionTime() {
       return executionTime;
   }

   public int getNodesVisited() {
       return nodesVisited;
   }

   public void setValue(List<String> path, long executionTime, int
nodesVisited) {
       this.path = path;
       this.executionTime = executionTime;
       this.nodesVisited = nodesVisited;
   }

   public void reset() {
       this.path = new ArrayList<>();
       this.nodesVisited = 0;
       this.executionTime = 0;
   }
```

```
}
```

Kelas *WordLadder* ini dibuat dengan tujuan untuk abstraksi kode sehingga kelas ini dapat diwariskan kepada tiga kelas algoritma lainnya, seperti WordLadderAStar, WordLadderUCS, dan WordLadderGBFS. Pada kelas ini terdapat atribut *path* untuk menampung informasi solusi, atribut *executionTime* untuk menyimpan informasi lama waktu eksekusi program, dan atribut *nodesVisited* untuk menyimpan informasi berapa simpul yang pernah dikunjungi oleh sebuah algoritma ketika menjalani pencarian. Sisanya, merupakan metode-metode *setter getter* yang digunakan untuk melakukan akses atau modifikasi terhadap atribut *private* yang dimiliki objek tersebut.

4. File WordLadderUCS.java

```
package algorithm;

import java.util.*;
import utils.*;

public class WordLadderUCS extends WordLadder implements Utils {

   public WordLadderUCS(List<String> path, long executionTime, int
nodesVisited) {
       super(path, executionTime, nodesVisited);
   }

   public void print_queue(Queue<Node> queue) {
       Queue<Node> tempQueue = new PriorityQueue<>(queue);
       tempQueue.addAll(queue);

       while (!tempQueue.isEmpty()) {
           Node node = tempQueue.poll();
           System.out.print(node.word + " ");
           System.out.print(node.depth + " ");
       }
       System.out.println();
   }

   public void find_path_solution_UCS(String starting_word, String
target_word) {
       long startTime = System.nanoTime();
```

```
        if (starting_word.length() != target_word.length()) {
            this.setValue(new ArrayList<>(), 0, 0);
            return;
        }

        Queue<Node> priorityQueue = new
PriorityQueue<>(Comparator.comparingInt(node -> node.depth));
        Map<String, Boolean> visited = new HashMap<>();
        int nodesVisited = 0;

        priorityQueue.add(new Node(starting_word, null, 0));

        while (!priorityQueue.isEmpty()) {
            Node current_node = priorityQueue.poll();
            nodesVisited++;

            if (current_node.word.equals(target_word)) {
                List<String> temp = make_path_from_node(current_node);
                long endTime = System.nanoTime();
                this.setValue(temp, endTime - startTime, nodesVisited);
                return;
            }

            for (String neighbor :
find_word_possibility(current_node.word)) {
                if (!visited.containsKey(neighbor)) {
                    visited.put(neighbor, true);
                    priorityQueue.add(new Node(neighbor, current_node,
current_node.depth + 1));
                }
            }
        }

        long endTime = System.nanoTime();
        this.setValue(new ArrayList<>(), endTime - startTime,
nodesVisited);
    }


    private List<String> make_path_from_node(Node current) {
        List<String> path = new ArrayList<>();
```

```
        while (current != null) {
            path.add(0, current.word);
            current = current.parent;
        }
        return path;
    }

    class Node {
        String word;
        Node parent;
        int depth;

        public Node(String word, Node parent, int depth) {
            this.word = word;
            this.parent = parent;
            this.depth = depth;
        }
    }
}
```

Kelas *WordLadderUCS* merupakan kelas yang dibuat dengan tujuan untuk melakukan penyelesaian persoalan permainan *WordLadder* dengan menggunakan algoritma *Uniform Cost Search* (UCS). Kelas ini memiliki metode yang hampir sama dengan kelas *WordLadderAStar* dan merupakan turunan dari kelas *WordLadder*. Namun, terdapat perbedaan pada kelas *Node* yang terdapat pada kelas ini, karena *Node* pada kelas ini menyimpan informasi atribut *depth, word,* dan *parent*. Pada kelas ini, terdapat metode utamanya, yaitu *find_path_solution_UCS* yang merupakan *solver* utama permasalahan *WordLadder* dengan menerapkan algoritma UCS. Kelas ini juga melakukan implementasi *interface Utils*.

5. File WordLadderGBFS.java

```
package algorithm;

import utils.*;
import java.util.*;

public class WordLadderGBFS extends WordLadder implements Utils {

    public WordLadderGBFS(List<String> path, long executionTime, int
nodesVisited) {
```

```
        super(path, executionTime, nodesVisited);
    }

    public void print_queue(Queue<Node> queue) {
        Queue<Node> tempQueue = new PriorityQueue<>(queue);

        while (!tempQueue.isEmpty()) {
            Node node = tempQueue.poll();
            System.out.print(node.word + " ");
            System.out.print(node.depth + " ");
        }
        System.out.println();
    }


    public void find_path_solution_GBFS(String starting_word, String
target_word) {
        long startTime = System.nanoTime();
        if (starting_word.length() != target_word.length()) {
            this.setValue(new ArrayList<>(), 0, 0);
            return;
        }

        Queue<Node> priorityQueue = new
PriorityQueue<>(Comparator.comparingInt(node ->
-count_same_letter(node.word, target_word)));
        Map<String, Boolean> visited = new HashMap<>();
        int nodesVisited = 0;

        priorityQueue.add(new Node(starting_word, null, 0));

        while (!priorityQueue.isEmpty()) {
            Node current_node = priorityQueue.poll();
            nodesVisited++;

            if (current_node.word.equals(target_word)) {
                List<String> temp = make_path_from_node(current_node);
                long endTime = System.nanoTime();
                this.setValue(temp, endTime - startTime, nodesVisited);
                return;
            }
```

```
            for (String neighbor :
find_word_possibility(current_node.word)) {
                if (!visited.containsKey(neighbor)) {
                    visited.put(neighbor, true);
                    priorityQueue.add(new Node(neighbor, current_node,
current_node.depth + 1));
                }
            }
        }

        long endTime = System.nanoTime();
        this.setValue(new ArrayList<>(), endTime - startTime,
nodesVisited);
    }


    private List<String> make_path_from_node(Node current) {
        List<String> path = new ArrayList<>();
        while (current != null) {
            path.add(0, current.word);
            current = current.parent;
        }
        return path;
    }

    class Node {
        String word;
        Node parent;
        int depth;

        public Node(String word, Node parent, int depth) {
            this.word = word;
            this.parent = parent;
            this.depth = depth;
        }
    }
}
```

Kelas *WordLadderGBFS* adalah kelas yang dibuat dengan tujuan untuk menampung *solver* untuk menyelesaikan persoalan permainan *WordLadder* dengan menggunakan algoritma *Greedy Best First Search* (GBFS). Kelas ini

merupakan turunan dari kelas *WordLadder* dan mengimplementasikan *interface Utils*. Kelas ini menggunakan konstruktor *superclass*-nya, memiliki metode yang hampir sama dengan kelas *WordLadder* algoritma lainnya, dan memiliki kelas *Node* yang sama dengan kelas *WordLadderUCS*. Inti utama dari kelas ini yang menyimpan otak dari *solver* dengan menggunakan algoritma GBFS terdapat pada metode *find_path_solution_GBFS*.

6. File WordLadderAStar.Java

```
package algorithm;

import utils.Utils;
import java.util.*;

public class WordLadderAStar extends WordLadder implements Utils {

   public WordLadderAStar(List<String> path, long executionTime, int
nodesVisited) {
       super(path, executionTime, nodesVisited);
   }

   public void print_queue(Queue<Node> queue) {
       Queue<Node> tempQueue = new PriorityQueue<>(queue);
       tempQueue.addAll(queue);

       while (!tempQueue.isEmpty()) {
           Node node = tempQueue.poll();
           System.out.print(node.word + " ");
           System.out.print("f" + node.f);
           System.out.print("g" + node.g);
       }
       System.out.println();
   }

   private List<String> make_path_from_node(Node current) {
       List<String> path = new ArrayList<>();
       while (current != null) {
           path.add(0, current.word);
           current = current.parent;
       }
       return path;
   }
```

```
   public void find_path_solution_AStar(String starting_word, String
target_word) {
       long startTime = System.nanoTime();
       if (starting_word.length() != target_word.length()) {
           this.setValue(new ArrayList<>(), 0, 0);
           return;
       }

       PriorityQueue<Node> priority_queue = new
PriorityQueue<>(Comparator.comparingInt(node -> node.f));
       Map<String, Integer> visited = new HashMap<>();

       priority_queue.add(new Node(starting_word, null, 0,
count_mismatch_letter(starting_word, target_word)));

       while (!priority_queue.isEmpty()) {
           Node current_node = priority_queue.poll();

           if (current_node.g > visited.getOrDefault(current_node.word,
0)) {
               continue;
           }

           nodesVisited++;

           if (current_node.word.equals(target_word)) {
               long endTime = System.nanoTime();
               this.setValue(make_path_from_node(current_node), endTime
- startTime, nodesVisited);
               return;
           }

           for (String children :
find_word_possibility(current_node.word)) {
               if (!visited.containsKey(children) ||
visited.get(children) > current_node.g + 1) {
                   visited.put(children, current_node.g + 1);
                   priority_queue.add(new Node(children, current_node,
current_node.g + 1, current_node.g + 1 +
count_mismatch_letter(children, target_word)));
```

```
                }
            }
        }

        long endTime = System.nanoTime();
        this.setValue(new ArrayList<>(), endTime - startTime,
nodesVisited);
    }

    class Node {
        String word;
        Node parent;
        int g; // Cost dari start hingga node yang ini
        int f; // Cost g + (heuristic mismatch count)

        public Node(String word, Node parent, int g, int f) {
            this.word = word;
            this.parent = parent;
            this.g = g;
            this.f = f;
        }
    }
}
```

Kelas *WordLadderAStar* merupakan kelas yang dibuat dengan tujuan sebagai kelas yang menjadi *solver* permainan *WordLadder* dengan metode algoritma A*. Pada kelas ini terdapat konstruktor yang diambil dari *superclass*-nya. Terdapat juga metode *make_path_from_node* yang merupakan metode untuk menyusun kembali dari suatu *node* dan dihubungkan dengan *parent-parent* sebelumnya sehingga mengembalikan satu *list* yang utuh atau *path* yang ditempuh untuk mencapai simpul tersebut. Pada kelas ini juga terdapat sebuah kelas *Node* yang berbeda dengan kelas *Node* yang berada pada kelas lainnya. Ini disebabkan penyimpanan informasi atribut yang berbeda pada *Node* di kelas ini, yaitu menyimpan informasi nilai *cost f* dan *g* untuk mencapai sebuah *node* tertentu. Kelas ini melakukan implementasi *interface Utils*.

7. File Main.Java

```
import java.util.*;
import java.util.concurrent.TimeUnit;
```

```
import algorithm.WordLadderAStar;
import algorithm.WordLadderGBFS;
import algorithm.WordLadderUCS;
import utils.Dictionary;

public class Main {
   public static void print_choice() {
       System.out.println("Choose the algorithm you would like to
use!");
       System.out.println("1. Uniformed Cost Search (UCS) Algorithm");
       System.out.println("2. Greedy Best First Search Algorithm");
       System.out.println("3. A-Star Algorithm");
       System.out.println("4. Exit program");
   }

   public static String get_information(List<String> path, long exec,
int node_number) {
       long convert = TimeUnit.MILLISECONDS.convert(exec,
TimeUnit.NANOSECONDS);
       String resultText = "";
       if (path.isEmpty()) {
           resultText += "No path found\n";
       } else {
           resultText += "Result path: " + path.toString() + "\n";
       }
       resultText += "Execution time: " + convert + " ms.\n";
       resultText += "Total node visited: " + node_number + "
nodes.\n";
       return resultText;
   }


   public static void main(String[] args) {
       Scanner scan = new Scanner(System.in);
       Dictionary.load_word("./utils/words.txt");

       while (true) {
           System.out.println("Welcome to Word Ladder Game Solver!");
           System.out.println("This program is designed to tackle the
well-known Word Ladder Game.");
           System.out.println("You will only need to choose the
```

```
algorithm used to solve the game, enter a starting word, and lastly
enter an ending word.");

System.out.println("==================================================
=============");

            print_choice();

            System.out.print(">> ");
            int choice;
            try {
                choice = scan.nextInt();
                if (choice == 4) {
                    break;
                } else if (choice < 1 || choice > 3) {
                    System.out.println("Invalid choice. Please enter a
number between 1 and 4.");
                    continue;
                }
            } catch (InputMismatchException e) {
                System.out.println("Invalid input. Please enter a number
between 1 and 4.");
                scan.next();
                continue;
            }

            String starting_word = "";
            String ending_word = "";

            boolean valid_input = false;
            while (!valid_input) {
                System.out.println("Enter the starting word:");
                if (scan.hasNext()) {
                    starting_word = scan.next();
                    if (Dictionary.word_valid_checker(starting_word)) {
                        valid_input = true;
                    } else {
                        System.out.println("Word is not found in
dictionary. Input a valid english word.");
                    }
                } else {
```

```
                    System.out.println("Invalid input. Please enter a
valid word.");
                    scan.next();
                }
            }

            valid_input = false;
            while (!valid_input) {
                System.out.println("Enter the ending word:");
                if (scan.hasNext()) {
                    ending_word = scan.next();
                    if (Dictionary.word_valid_checker(ending_word)) {
                        valid_input = true;
                    } else {
                        System.out.println("Word is not found in
dictionary. Input a valid english word.");
                    }
                } else {
                    System.out.println("Invalid input. Please enter a
valid word.");
                    scan.next();
                }
            }

            if (starting_word.length() != ending_word.length()) {
                System.out.println("The starting word and the ending
word must have the same length.");
                continue;
            }

            switch (choice) {
                case 1:
                    WordLadderUCS solverUCS = new WordLadderUCS(new
ArrayList<>(), 0, 0);
                    solverUCS.find_path_solution_UCS(starting_word,
ending_word);

System.out.println(get_information(solverUCS.getPath(),
solverUCS.getExecutionTime(), solverUCS.getNodesVisited()));
                    break;
                case 2:
```

```
                    WordLadderGBFS solverGBFS = new WordLadderGBFS(new
ArrayList<>(), 0, 0);
                    solverGBFS.find_path_solution_GBFS(starting_word,
ending_word);

System.out.println(get_information(solverGBFS.getPath(),
solverGBFS.getExecutionTime(), solverGBFS.getNodesVisited()));
                    break;
                case 3:
                    WordLadderAStar solverAStar = new
WordLadderAStar(new ArrayList<>(), 0, 0);
                    solverAStar.find_path_solution_AStar(starting_word,
ending_word);

System.out.println(get_information(solverAStar.getPath(),
solverAStar.getExecutionTime(), solverAStar.getNodesVisited()));
                    break;
            }
        }

        scan.close();
    }
}
```

File Main.java ini hanya berisi logika pemrograman secara prosedural untuk meminta input kepada user dan kemudian memprosesnya berdasarkan algoritma yang dipilih oleh pengguna. Untuk melakukan pemrograman secara prosedural dibuat *method public static void main (String[] args)* yang berisi otak dari program ini yang telah menggabungkan semua algoritma menjadi satu. Main.java ini dapat dikompilasi dan dijalankan *bytecode*, dan jika *bytecode*-nya dijalankan maka akan muncul program berbasis CLI yang interaktif sehingga dapat digunakan oleh pengguna.

8. File WordLadderGUI.py

```
import javax.swing.*;

import algorithm.WordLadderAStar;
import algorithm.WordLadderGBFS;
import algorithm.WordLadderUCS;

import java.awt.*;
```

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.TimeUnit;

import utils.Dictionary;

public class WordLadderGUI extends JFrame implements ActionListener {
   private JTextField startingWordField, endingWordField;
   private JButton solveButton;
   private JTextArea resultArea;
   private JRadioButton jRadioButton1, jRadioButton2, jRadioButton3;
   private ButtonGroup algorithmGroup;

   public static String get_information(String algorithm_name,
List<String> path, long exec, int node_number) {
       long convert = TimeUnit.MILLISECONDS.convert(exec,
TimeUnit.NANOSECONDS);
       String resultText = "The selected algorithm is " +
algorithm_name + ". Here is the result.\n";
       if (path.isEmpty()) {
           resultText += "No path found\n";
       } else {
           resultText += "Path result: " + path.toString() + "\n";
       }
       resultText += "Execution time: " + convert + " ms.\n";
       resultText += "Total of nodes visited: " + node_number + "
nodes.\n";
       return resultText;
   }

   public WordLadderGUI() {
       setTitle("Word Ladder Solver");
       setSize(500, 400);
       setDefaultCloseOperation(EXIT_ON_CLOSE);
       setLocationRelativeTo(null);
       setLayout(new BorderLayout());

       JPanel formPanel = new JPanel(new GridBagLayout());
       formPanel.setBackground(new Color(247, 224, 146));
```

```
        GridBagConstraints c = new GridBagConstraints();
        c.insets = new Insets(10, 10, 10, 10);

        JLabel startingWordLabel = new JLabel("Starting Word:");
        startingWordLabel.setForeground(new Color(245, 130, 15));
        c.gridx = 0; c.gridy = 0;
        formPanel.add(startingWordLabel, c);

        startingWordField = new JTextField(15);
        startingWordField.setForeground(new Color(100, 65, 0));
        startingWordField.setBackground(new Color(255, 255, 224));
        c.gridx = 1;
        formPanel.add(startingWordField, c);

        JLabel endingWordLabel = new JLabel("Ending Word:");
        endingWordLabel.setForeground(new Color(245, 130, 15));
        c.gridx = 0; c.gridy = 1;
        formPanel.add(endingWordLabel, c);

        endingWordField = new JTextField(15);
        endingWordField.setForeground(new Color(100, 65, 0));
        endingWordField.setBackground(new Color(255, 255, 224));
        c.gridx = 1;
        formPanel.add(endingWordField, c);

        jRadioButton1 = new JRadioButton("UCS");
        jRadioButton2 = new JRadioButton("GBFS");
        jRadioButton3 = new JRadioButton("A*");
        jRadioButton1.setBackground(formPanel.getBackground());
        jRadioButton2.setBackground(formPanel.getBackground());
        jRadioButton3.setBackground(formPanel.getBackground());
        algorithmGroup = new ButtonGroup();
        algorithmGroup.add(jRadioButton1);
        algorithmGroup.add(jRadioButton2);
        algorithmGroup.add(jRadioButton3);
        JPanel radioPanel = new JPanel(new
FlowLayout(FlowLayout.CENTER));
        radioPanel.setBackground(new Color(255, 198, 64));
        radioPanel.add(jRadioButton1);
        radioPanel.add(jRadioButton2);
        radioPanel.add(jRadioButton3);
```

```
        c.gridx = 0; c.gridy = 2; c.gridwidth = 2;
        formPanel.add(radioPanel, c);

        solveButton = new JButton("Solve");
        solveButton.setBackground(new Color(245, 130, 15));
        solveButton.addActionListener(this);
        c.gridx = 0; c.gridy = 3; c.gridwidth = 2;
        formPanel.add(solveButton, c);

        resultArea = new JTextArea(5, 20);
        resultArea.setLineWrap(true);
        resultArea.setEditable(false);
        resultArea.setBackground(new Color(255, 255, 255));
        JScrollPane scrollPane = new JScrollPane(resultArea);
        scrollPane.setBorder(BorderFactory.createLineBorder(new
Color(255, 198, 64), 2));

        add(formPanel, BorderLayout.NORTH);
        add(scrollPane, BorderLayout.CENTER);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        Dictionary.load_word("./utils/words.txt");
        if (e.getSource() == solveButton) {
            String startingWord =
startingWordField.getText().toLowerCase();
            String endingWord = endingWordField.getText().toLowerCase();

            if (!Dictionary.word_valid_checker(startingWord)) {
                JOptionPane.showMessageDialog(this, "Invalid input
words. Starting word is not defined in the dictionary.");
                return;
            }

            if (!Dictionary.word_valid_checker(endingWord)) {
                JOptionPane.showMessageDialog(this, "Invalid input
words. Ending word is not defined in the dictionary.");
                return;
            }
```

```
            if (startingWord.length() != endingWord.length()) {
                JOptionPane.showMessageDialog(this, "Invalid input
length. The starting word and the ending word input must have the same
length.");
                return;
            }

            String selectedAlgorithm = getSelectedAlgorithm();

            if (selectedAlgorithm == "No algorithm selected") {
                JOptionPane.showMessageDialog(this, "Invalid algorithm
selection. Please select one of the algorithm provided.");
                return;
            }

            Runtime runtime = Runtime.getRuntime();
            long initialMemory = runtime.totalMemory() -
runtime.freeMemory();
            switch (selectedAlgorithm) {
                case "UCS":
                    WordLadderUCS solverUCS = new WordLadderUCS(new
ArrayList<>(), 0, 0);
                    solverUCS.find_path_solution_UCS(startingWord,
endingWord);
                    resultArea.setText(get_information("Uniformed Cost
Search", solverUCS.getPath(), solverUCS.getExecutionTime(),
solverUCS.getNodesVisited()));
                    break;
                case "GBFS":
                    WordLadderGBFS solverGBFS = new WordLadderGBFS(new
ArrayList<>(), 0, 0);
                    solverGBFS.find_path_solution_GBFS(startingWord,
endingWord);
                    resultArea.setText(get_information("Greedy Best
First Search", solverGBFS.getPath(), solverGBFS.getExecutionTime(),
solverGBFS.getNodesVisited()));
                    break;
                case "A*":
                    WordLadderAStar solverAStar = new
WordLadderAStar(new ArrayList<>(), 0, 0);
```

```
                    solverAStar.find_path_solution_AStar(startingWord,
endingWord);
                    resultArea.setText(get_information("A-Star",
solverAStar.getPath(), solverAStar.getExecutionTime(),
solverAStar.getNodesVisited()));
                    break;
            }
            long finalMemory = runtime.totalMemory() -
runtime.freeMemory();
            long memoryUsed = finalMemory - initialMemory;
            System.out.println("Memory used: " + memoryUsed + " bytes");
        }
    }

    private String getSelectedAlgorithm() {
        if (jRadioButton1.isSelected()) return "UCS";
        if (jRadioButton2.isSelected()) return "GBFS";
        if (jRadioButton3.isSelected()) return "A*";
        return "No algorithm selected";
    }



    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            WordLadderGUI gui = new WordLadderGUI();
            gui.setVisible(true);
        });
    }
}
```

File WordLadderGUI.java memiliki sebuah kelas yang bernama *WordLadderGUI* yang berisi atribut-atribut untuk melakukan inisialisasi objek-objek yang akan ditampilkan pada GUI. Untuk membuat GUI, penulis menggunakan *library Java Swing*. Pada kelas ini, terdapat konstruktor *WordLadderGUI* yang bertujuan untuk melakukan pembuatan *interface*-nya yang kemudian akan dilihat oleh pengguna, seperti komponen *box*, *input*, *button*, *radio-group*, dan komponen lainnya. Kemudian, terdapat metode *actionPerformed* yang bertujuan untuk meletakkan logika dari program ini yang membuat GUI yang dibuat menjadi memiliki arti. Pada metode *actionPerformed*, dilakukan inisialisasi seperti pembacaan kamus melalui kelas Dictionary, perlakuan pengecekan validasi *input* dari pengguna, dan melakukan proses berupa

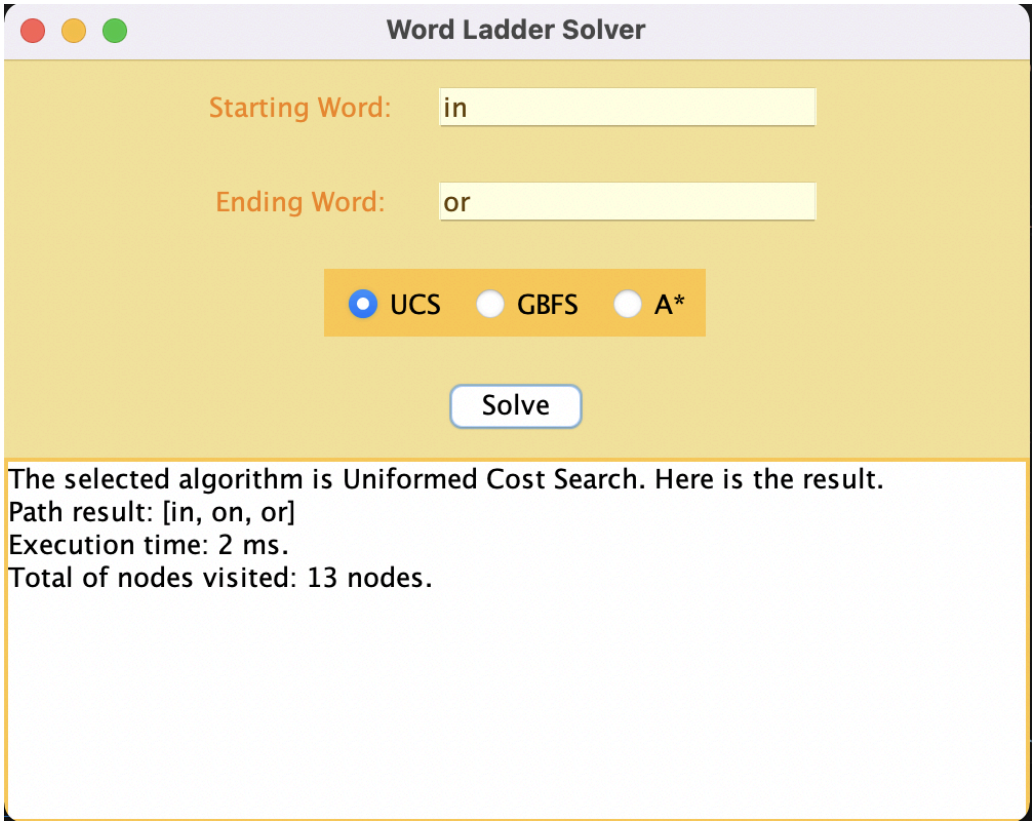
aksi pemrosesan informasi yang telah diterima setelah pengguna menekan *submit button*. Pada kelas ini juga terdapat *public static void main(String[] args)* yang berisi instansiasi *WordLadderGUI* dan GUI diatur menjadi *visible* sehingga dapat terlihat pada layar *desktop* pengguna.

F. *Screenshot* Program (Input dan Output)

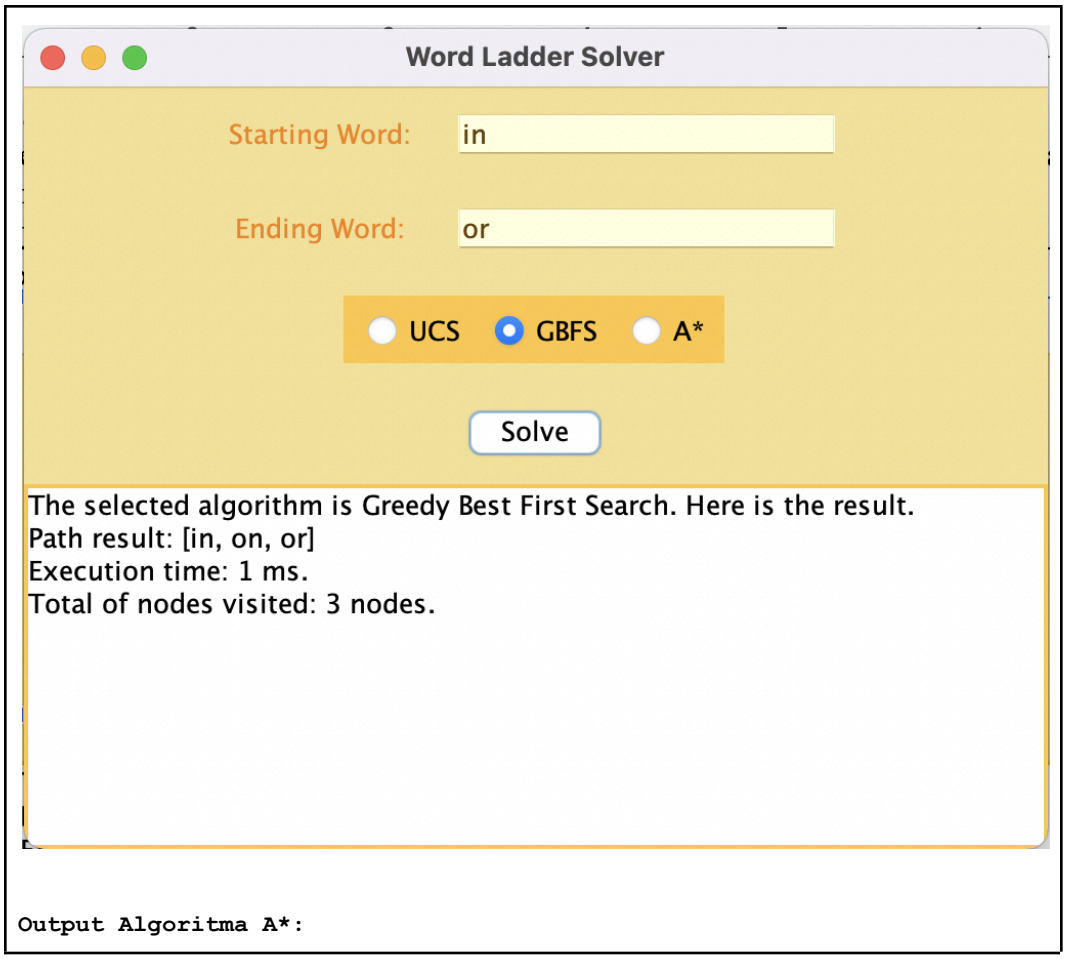
Berikut ini adalah hasil uji coba kode program berupa input yang digunakan dan output yang dihasilkan pada program untuk masing-masing algoritma

1. Uji Coba 1

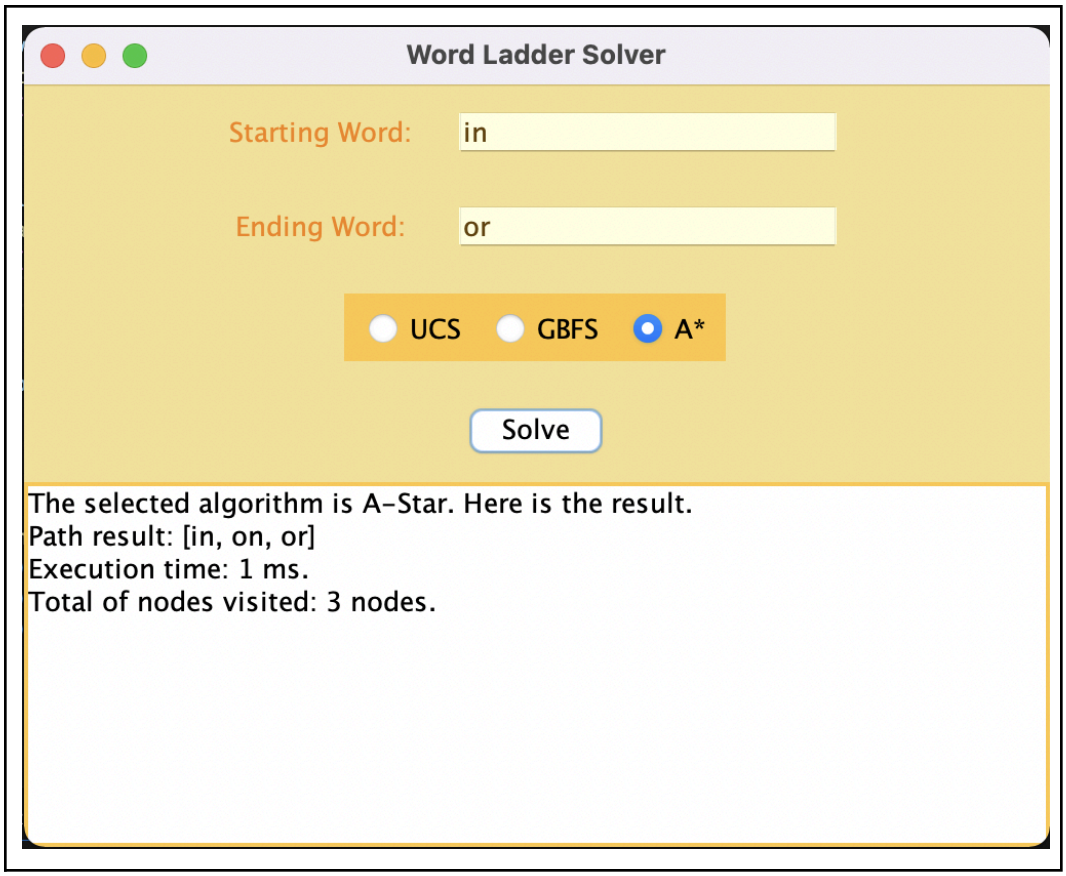
```
Output Algoritma UCS:
```




```
Output Algoritma GBFS:
```

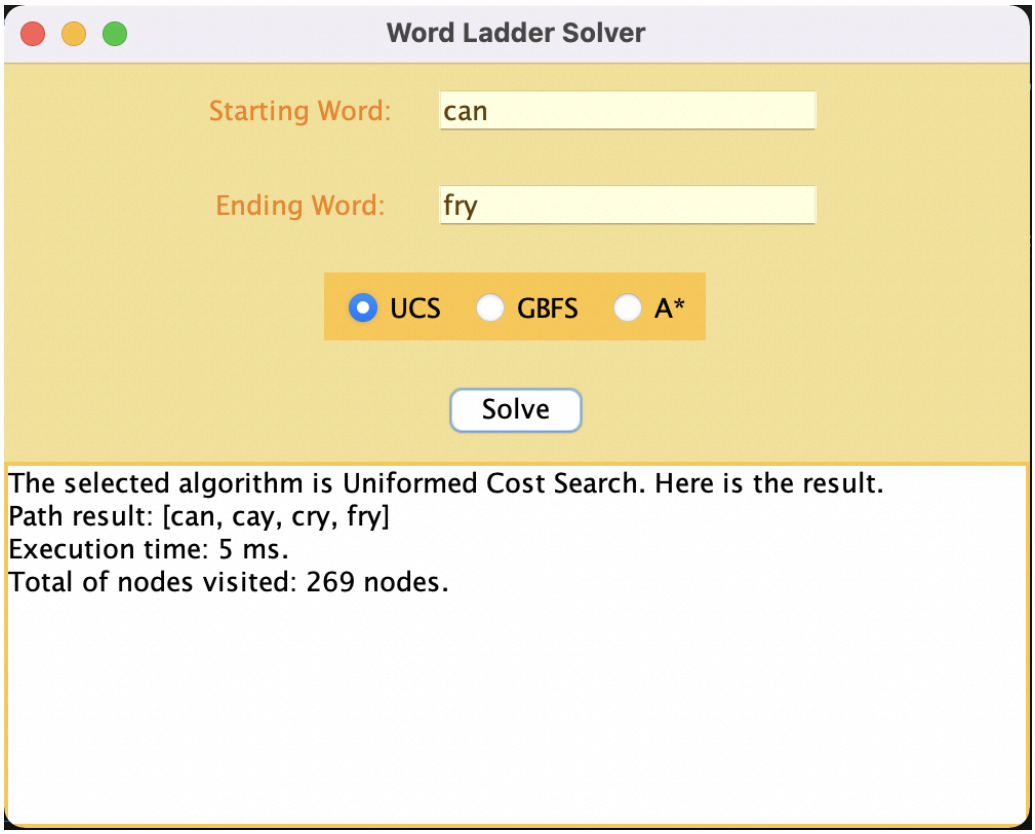


Output Algoritma A*:

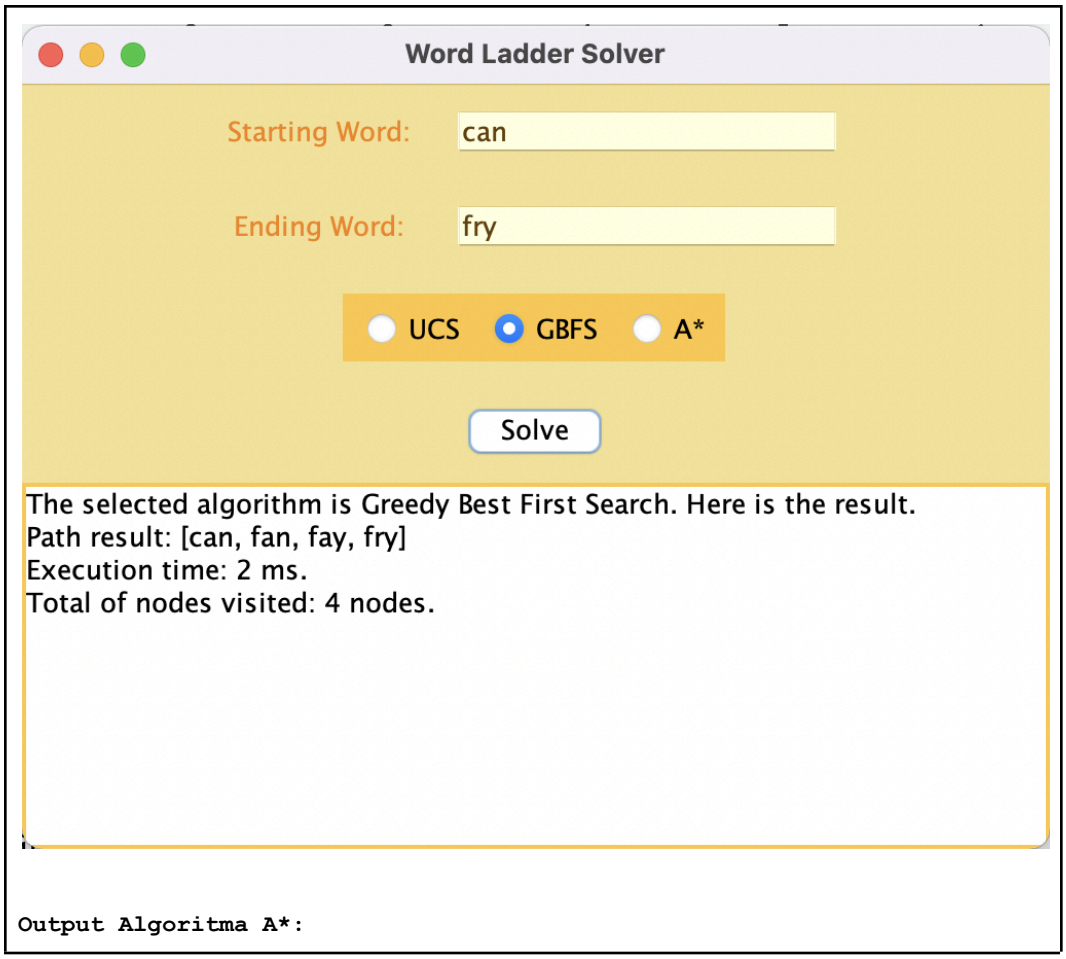


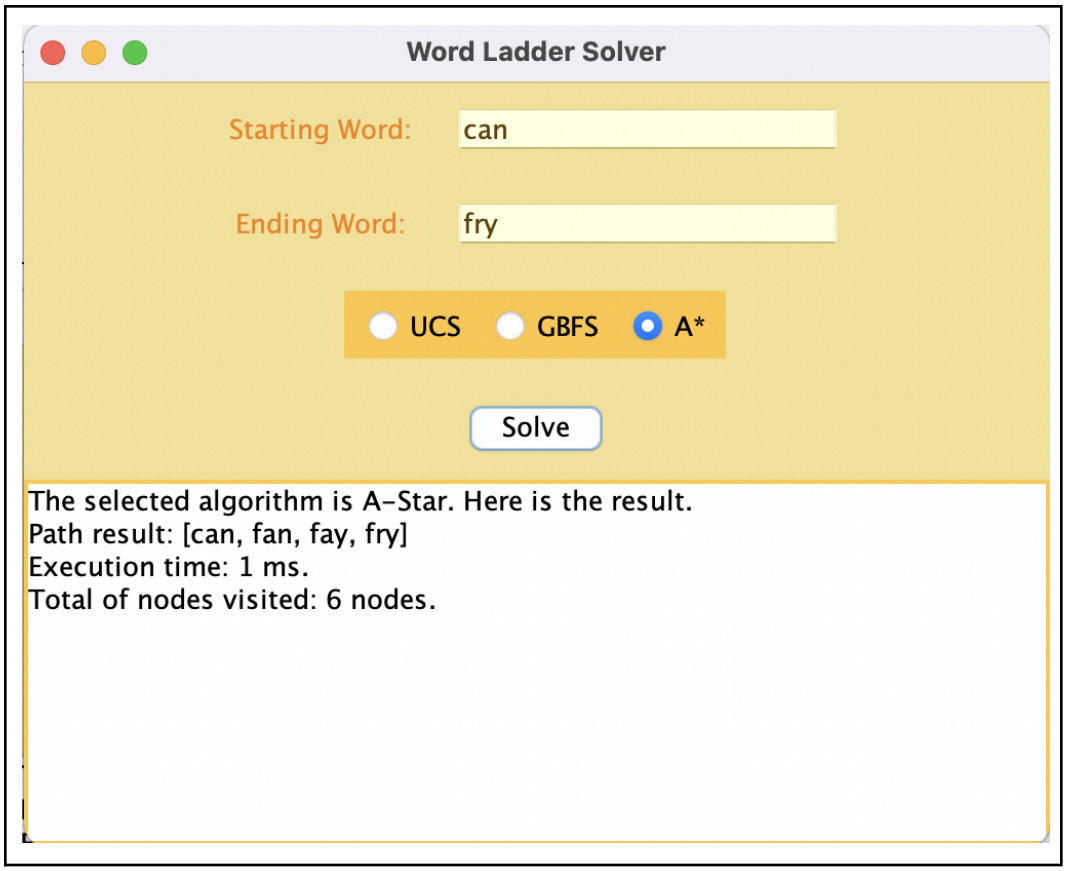
2. Uji Coba 2

```
Output Algoritma UCS:
```



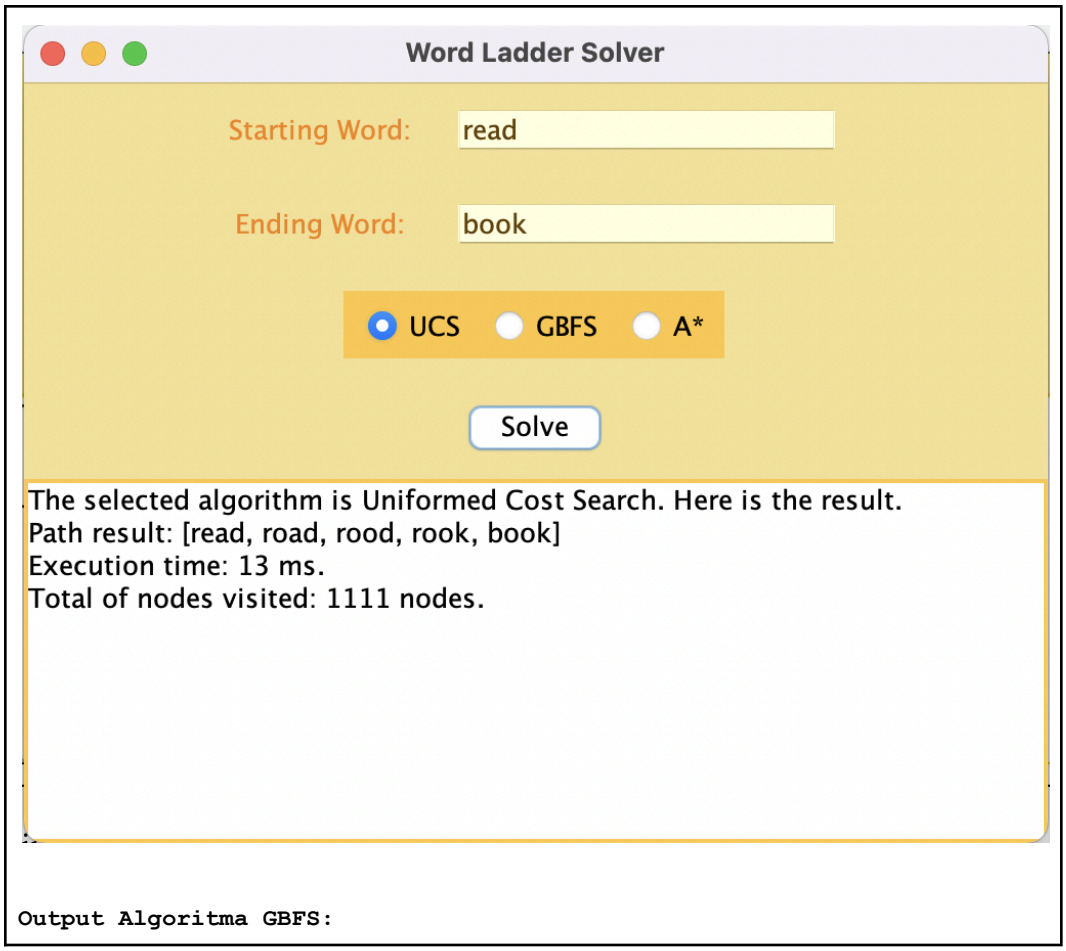

```
Output Algoritma GBFS:
```

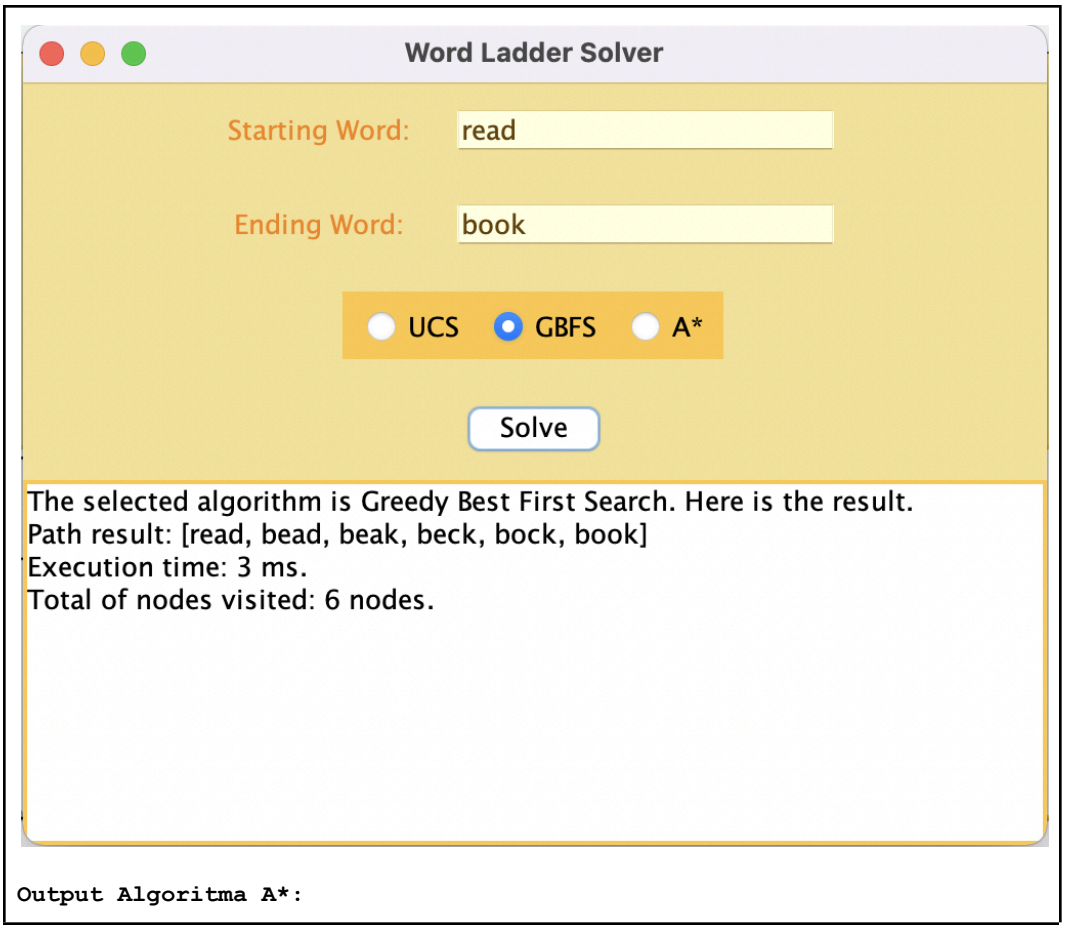

Word Ladder Solver
Starting Word:
can
Ending Word:
fry
UCS
GBFS
A*
Solve
The selected algorithm is Greedy Best First Search. Here is the result.
Path result: [can, fan, fay, fry]
Execution time: 2 ms.
Total of nodes visited: 4 nodes.
Output Algoritma A*:



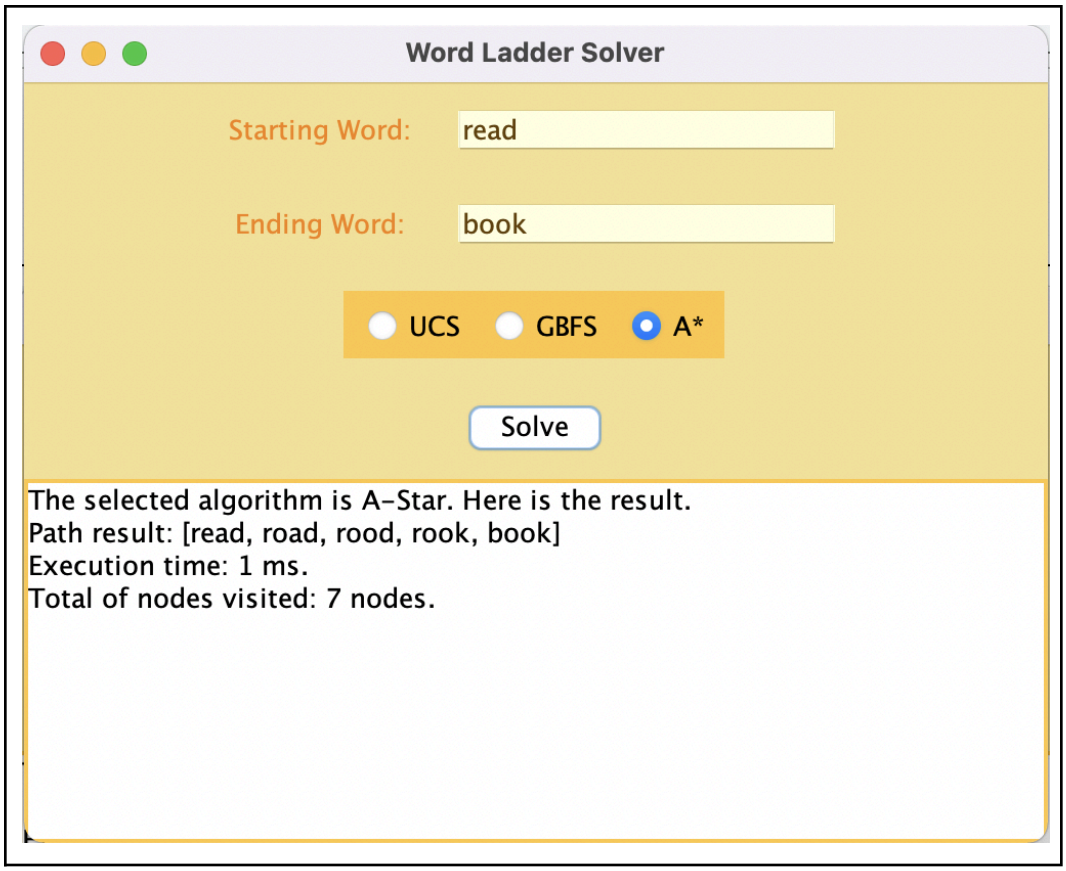
3. Uji Coba 3

```
Output Algoritma UCS:
```


Word Ladder Solver
Starting Word:
read
Ending Word:
book
UCS
GBFS
A*
Solve
The selected algorithm is Uniformed Cost Search. Here is the result.
Path result: [read, road, rood, rook, book]
Execution time: 13 ms.
Total of nodes visited: 1111 nodes.
Output Algoritma GBFS:

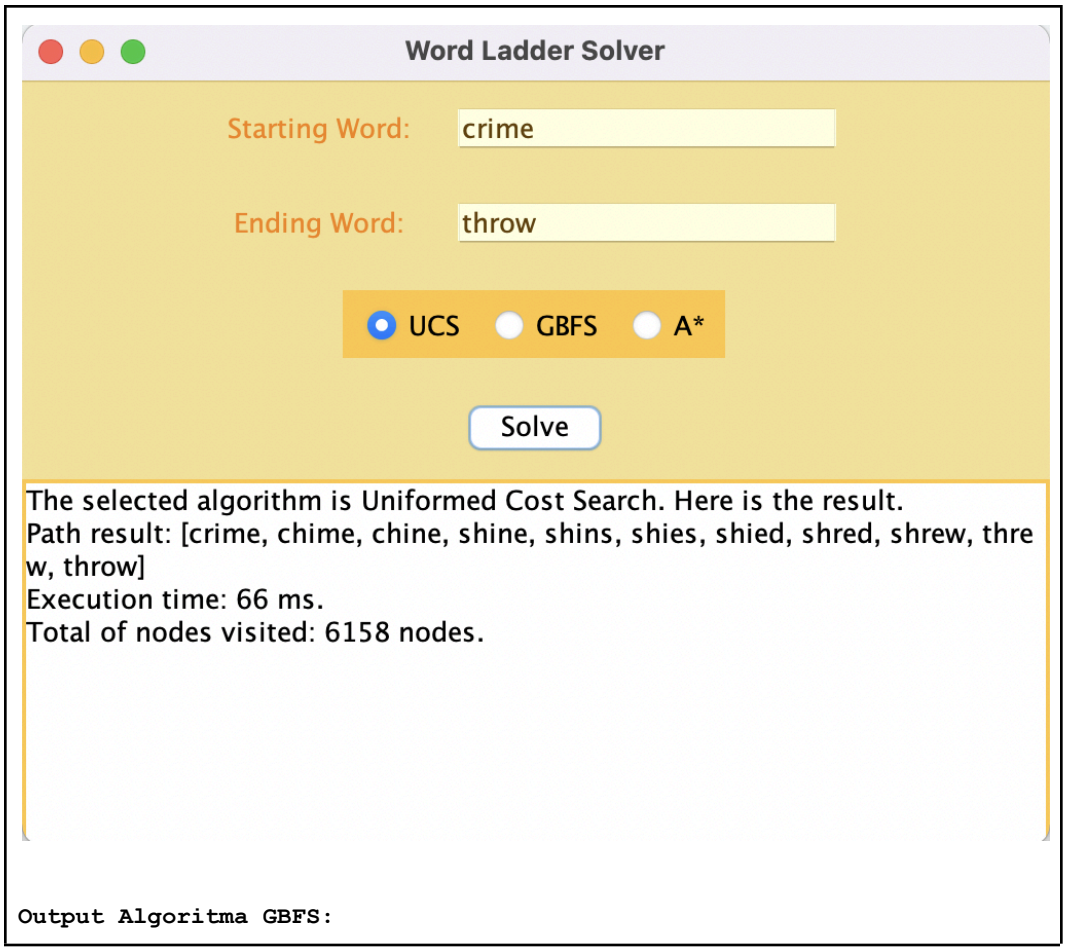


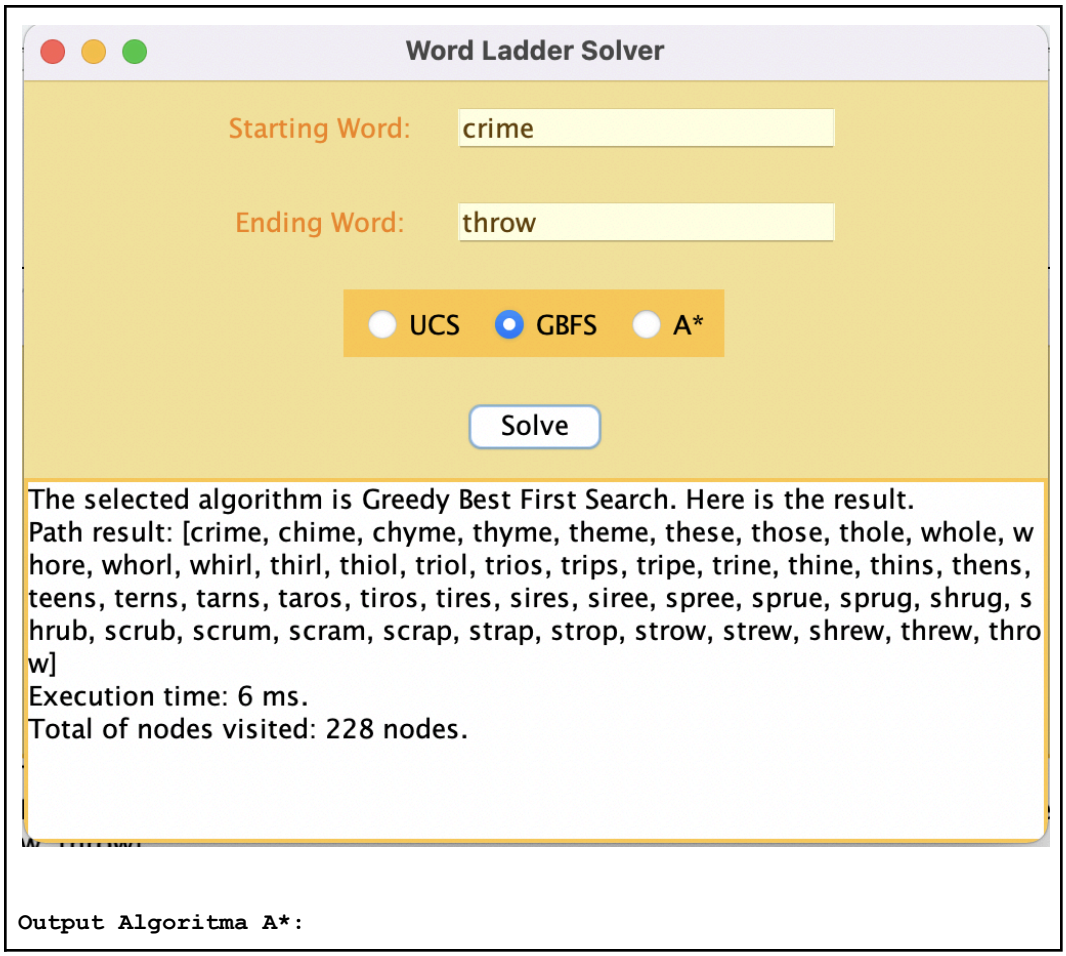
Output Algoritma A*:



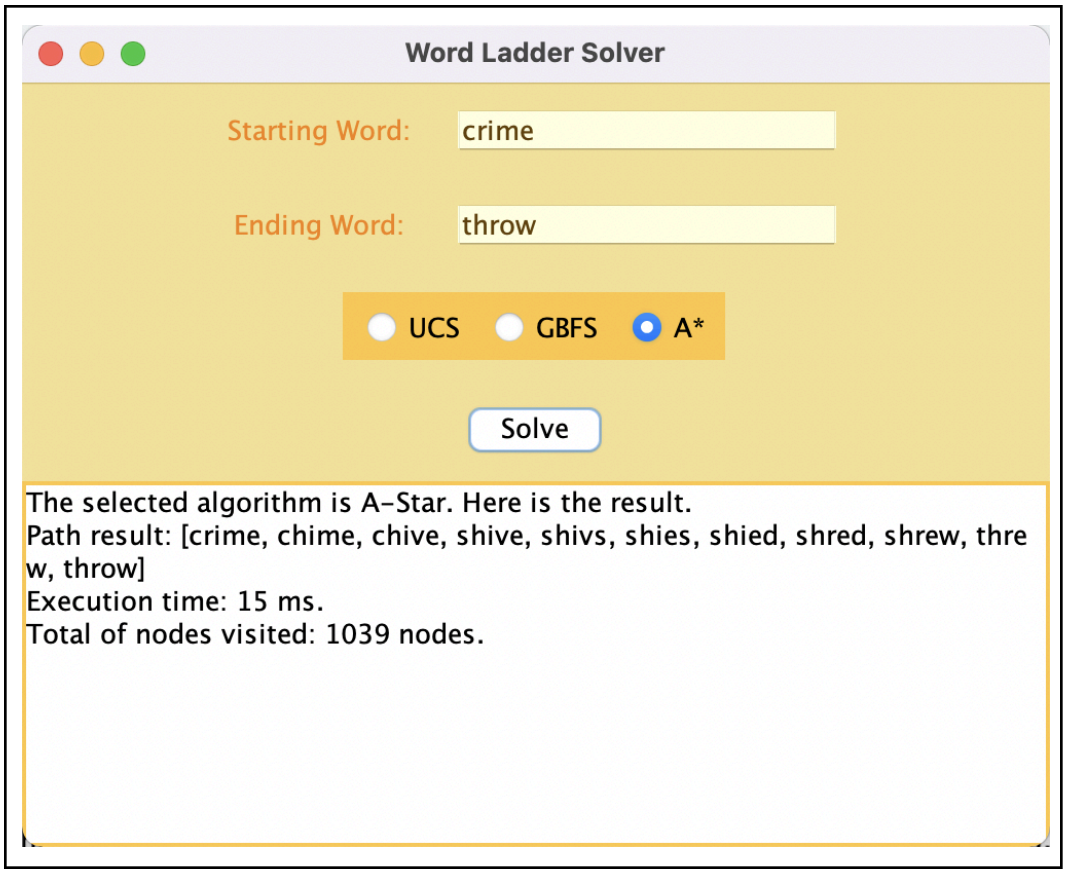
4. Uji Coba 4

```
Output Algoritma UCS:
```


Word Ladder Solver
Starting Word:
crime
Ending Word:
throw
UCS
GBFS
A*
Solve
The selected algorithm is Uniformed Cost Search. Here is the result.
Path result: [crime, chime, chine, shine, shins, shies, shied, shred, shrew, thre
w, throw]
Execution time: 66 ms.
Total of nodes visited: 6158 nodes.
Output Algoritma GBFS:

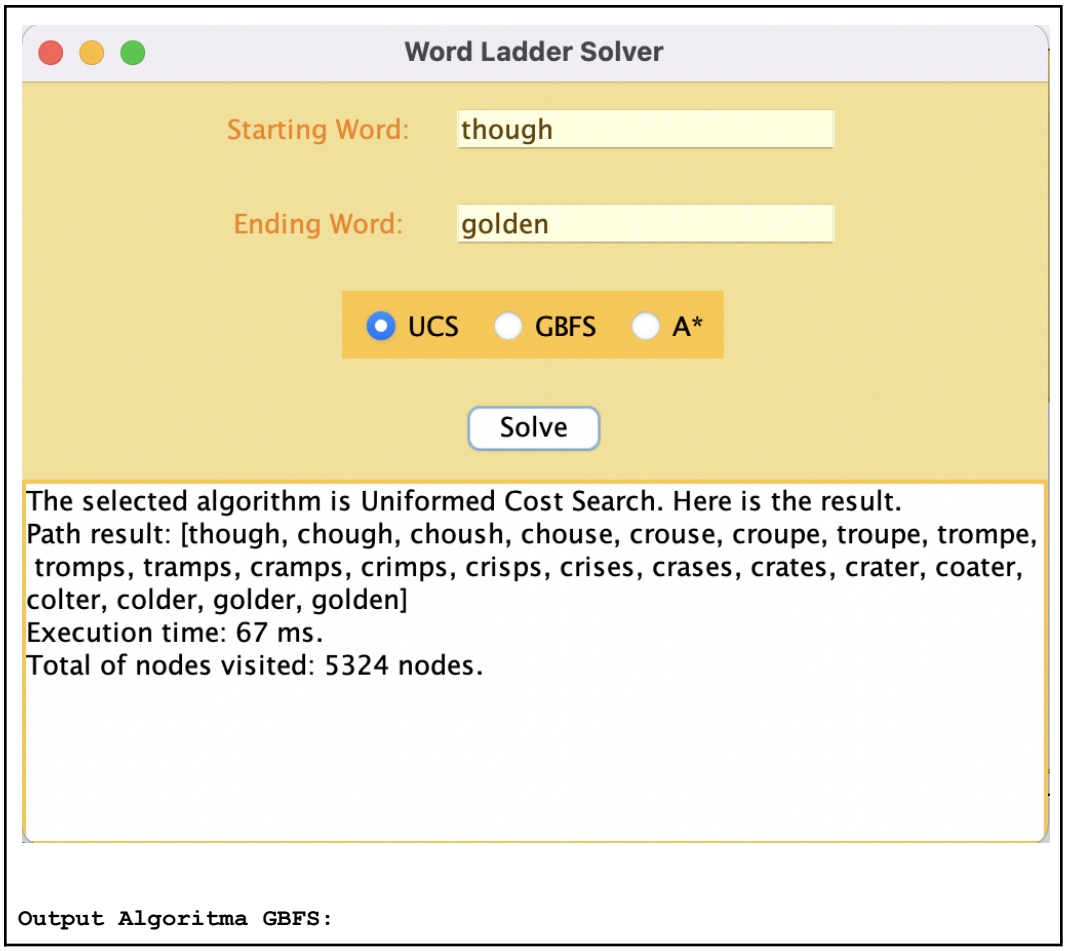


Output Algoritma A*:

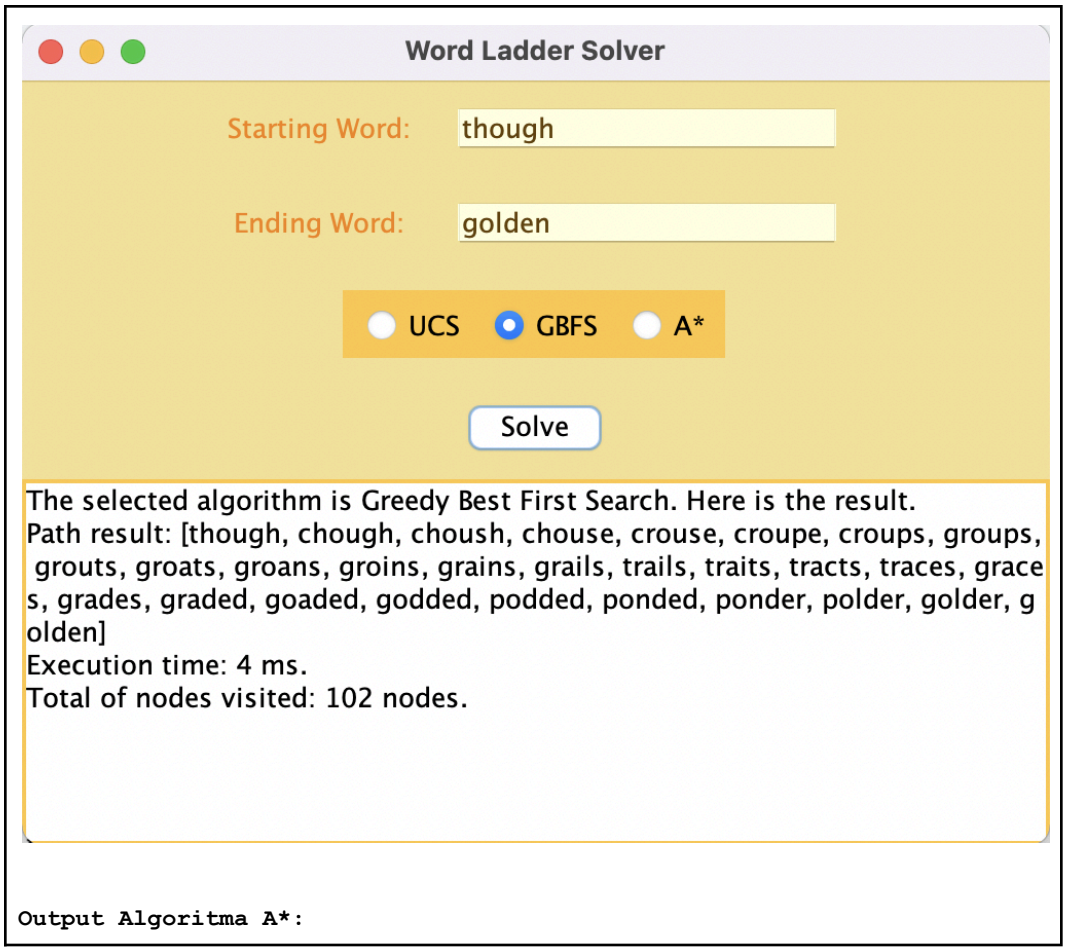


5. Uji Coba 5

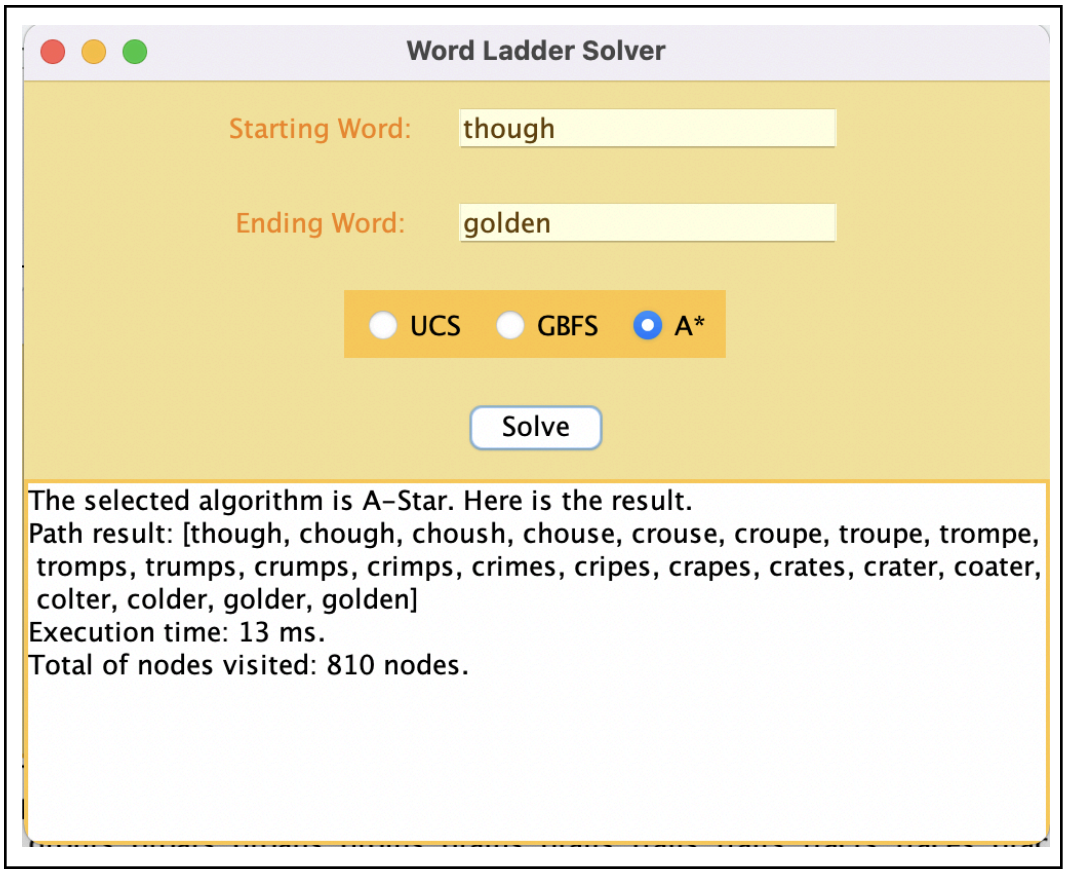
```
Output Algoritma UCS:
```




```
Output Algoritma GBFS:
```

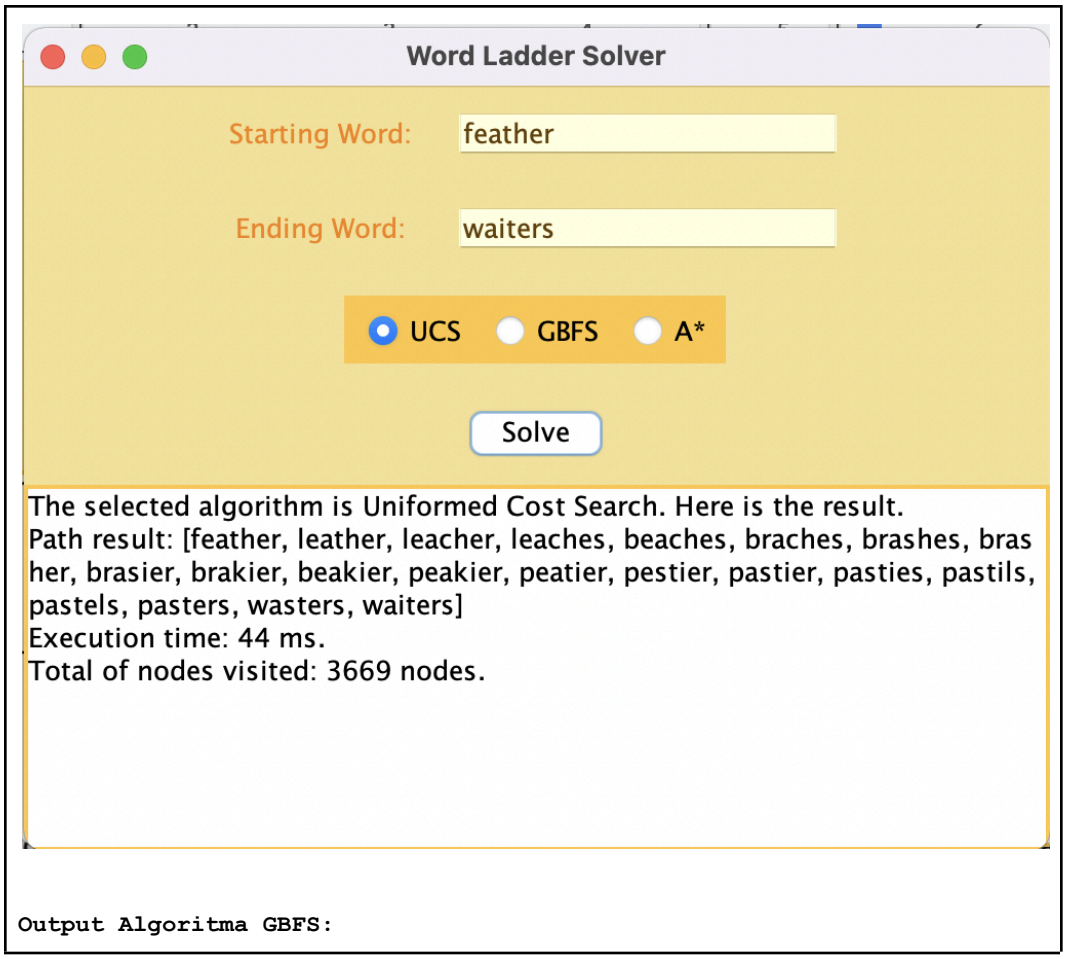


Output Algoritma A*:

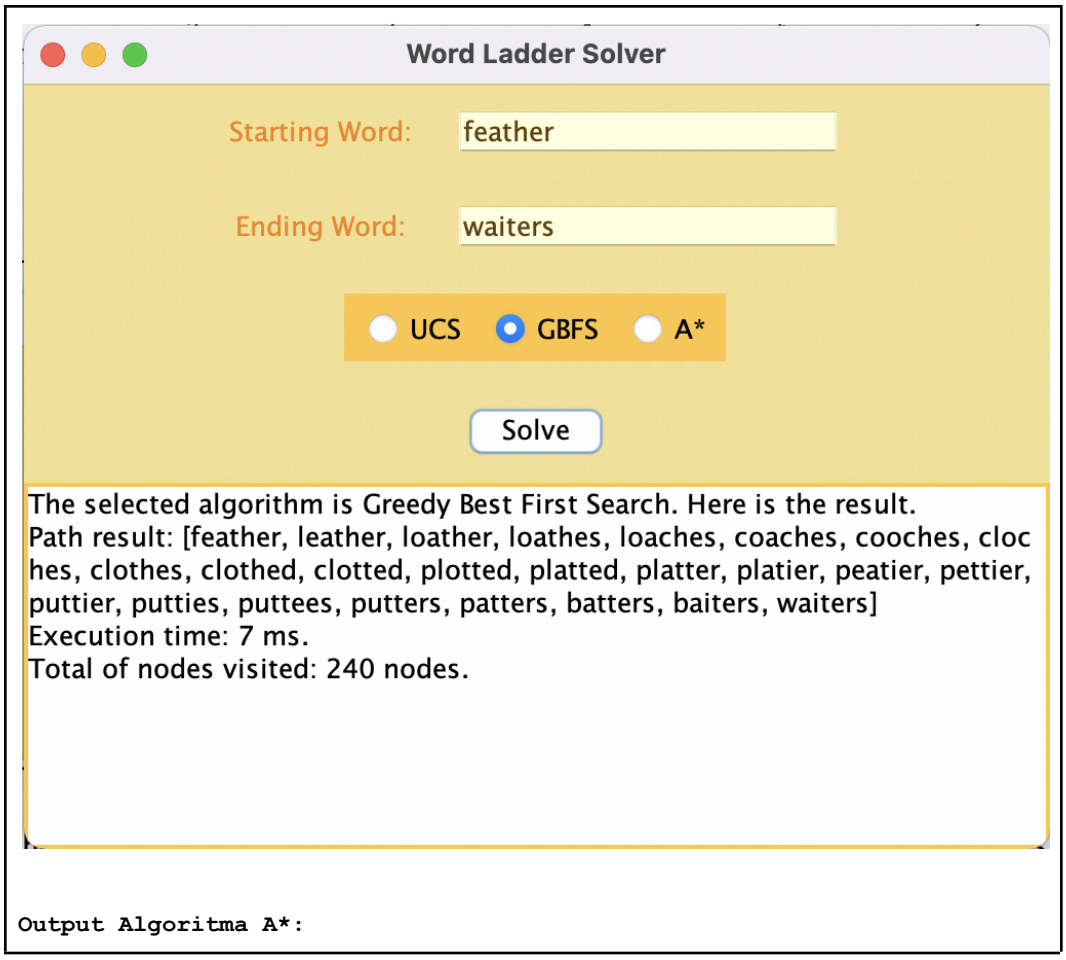


6. Uji Coba 6

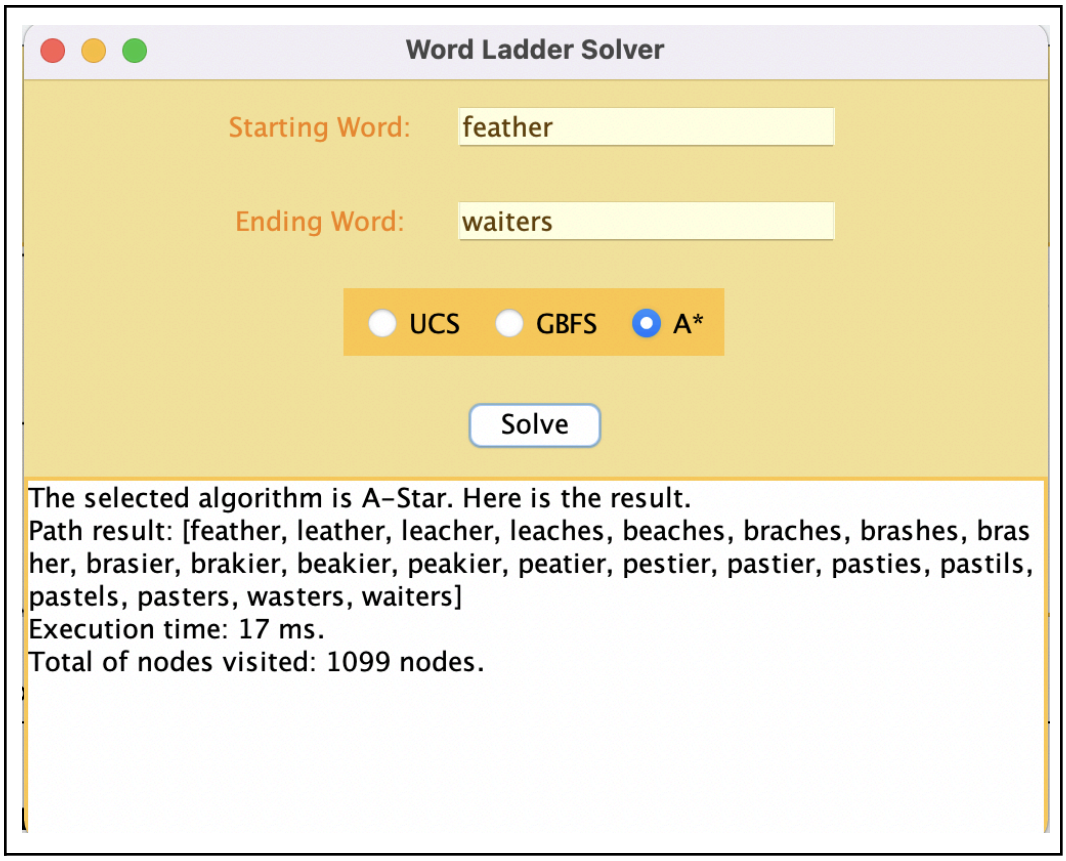
```
Output Algoritma UCS:
```




```
Output Algoritma GBFS:
```



Output Algoritma A*:



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

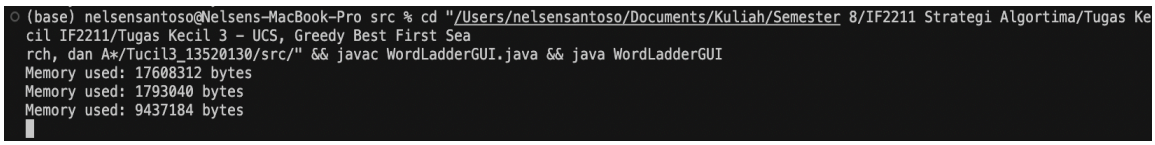
Berdasarkan uji coba yang telah dilakukan di atas, kita dapat menyimpulkan bahwa apabila dilihat dari segi optimalitas, terlihat jelas bahwa algoritma yang paling optimal dalam kasus penyelesaian permainan *WordLadder* ini adalah algoritma A* dengan optimal yang dimaksud berarti menghasilkan solusi yang paling optimal (perjalanan yang paling singkat dengan panjang lintasan paling sedikit) dan waktu eksekusi yang cepat pula. Didapat pula informasi bahwa algoritma *Uniform Cost Search* memiliki waktu eksekusi yang paling lama dibandingkan dengan ketiga algoritma, namun memiliki solusi yang optimal. Dalam hal ini, optimal berarti lintasannya tetap efisien dengan panjang lintasan sedikit. Sedangkan, untuk algoritma *Greedy Best First Search*, algoritma cenderung memiliki waktu eksekusi yang paling cepat dibandingkan ketiga algoritma yang dibandingkan, namun memiliki solusi yang kurang optimal, dimana panjang lintasan yang ditempuh cenderung lebih banyak dibandingkan dengan algoritma lainnya. Terakhir, algoritma *A-Star* yang memiliki solusi yang optimal dan memiliki waktu eksekusi yang berada di rata-rata perbandingan ketiga algoritma tersebut. Oleh karena itu, karena tujuan permainan *WordLadder* adalah mencari lintasan yang optimal, maka algoritma A* merupakan algoritma yang paling baik dan juga menyediakan waktu eksekusi yang relatif cepat.

Berdasarkan informasi yang diperoleh dari keenam uji coba tersebut, terdapat

pula korelasi yang cukup tinggi antara waktu eksekusi dengan banyaknya *node* yang dikunjungi oleh sebuah algoritma. Apabila data hasil percobaan untuk setiap algoritma dibuatkan graf yang membandingkan antara keduanya, akan dapat dibentuk garis linear dengan data *outlier* yang minim. Oleh karena itu, kedua variabel, yakni waktu eksekusi dan banyaknya *node* yang dilalui, bisa dikatakan saling terikat sehingga akan cenderung searah perkembangannya, bahwa makin besar banyaknya *node* yang dilalui, maka makin besar pula waktu eksekusinya.

Dari segi waktu eksekusi, algoritma *Greedy Best First Search* lebih efisien dibandingkan dengan algoritma lainnya. Hal ini disebabkan karena *node* yang dikunjungi oleh algoritma tersebut cenderung sedikit dibandingkan algoritma lainnya. Alasan mengapa *node* yang dikunjungi oleh algoritma GBFS cenderung sedikit ialah karena algoritma ini hanya mempertimbangkan *heuristic estimation cost*, sehingga tidak mempertimbangkan *node-node* lain yang mungkin memiliki solusi yang lebih optimal. Jadi, algoritma GBFS memiliki kelebihan dalam segi waktu eksekusi, namun kelemahannya terletak pada solusi yang kurang optimal, berupa panjang lintasan yang paling panjang di antara ketiganya.

Di samping itu, apabila analisis dilakukan dari penggunaan memori yang diperlukan untuk masing-masing algoritma, dapat dilihat gambar di bawah ini yang menunjukkan memori yang digunakan oleh algoritma UCS, *Greedy Best First Search*, dan A* secara berturut-turut dalam uji coba ke-6.



Dapat disimpulkan bahwa yang mengonsumsi memori terbesar hingga terkecil untuk melakukan eksekusi adalah algoritma UCS, algoritma A*, dan algoritma GBFS. Semua variabel saling berkorelasi dimulai dari waktu eksekusi, banyak *node* yang dikunjungi, dan juga memori yang dipakai. Ketiga atribut informasi tersebut menunjukkan keterhubungan korelasi yang positif sehingga dapat disimpulkan pula bahwa memori yang semakin besar dikonsumsi disebabkan pula akibat banyaknya *nodes* yang pernah dikunjungi oleh algoritma tersebut. Tingginya memori pada algoritma UCS ini disebabkan akibat pembangkitan *node* anak yang dilakukan secara terus menerus secara bertingkat secara *depth*, sehingga sifat dari UCS hampir sama dengan algoritma BFS, yang mana akan mengalami ekspansi memori yang menyebabkan ruang kompleksitasnya berkembang secara eksponensial. Berbeda dengan algoritma A*, yang mungkin juga melakukan pembangkitan anak secara bertingkat terhadap seluruh *node* pada setiap tingkatannya, namun algoritma ini tetap melakukan seleksi terhadap *node* yang dibandingkan berdasarkan *heuristic estimation cost*-nya sehingga tidak semua *node* dibangkitkan seperti UCS, namun masih bisa mendapatkan solusi optimal. Sedangkan, untuk algoritma GBFS, yang tidak memedulikan kedalaman pencariannya dan hanya berusaha untuk memenuhi keperluan *heuristic*-nya, maka sifatnya menjadi pencarian

yang bersifat vertikal (seperti DFS) sehingga algoritma ini akan memiliki ruang kompleksitas yang lebih kecil dibandingkan kedua algoritma lainnya.

H. Pranala

Berikut ini adalah alamat *repository* yang berisi kode program Tugas Kecil 3 IF2211 Strategi Algoritma milik Nelsen Putra (13520130):
https://github.com/nelsenputra/Tucil3_13520130

I. *Checklist*

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	✓	