LAPORAN TUGAS KECIL 3

PENYELESAIAN PERMAINAN WORD LADDER MENGGUNAKAN ALGORITMA UCS, GREEDY BEST FIRST SEARCH, DAN A*

Disusun oleh:
Venantius Sean Ardi Nugroho/13522078

Mata Kuliah IF2211 Strategi Algoritma
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
INSTITUT TEKNOLOGI BANDUNG

DAFTAR ISI

A. Algoritma Program

a. Pembangkitan Child Node

Dalam permainan *Word Ladder* , Anda diminta untuk menggunakan kata - kata yang berbeda 1 huruf dengan kata yang diberikan atau kata yang sebelumnya Anda input untuk mendekatkan Anda ke kata tujuan. Kata - kata yang berbeda 1 huruf tersebut didefinisikan sebagai *child node* dari suatu kata. Berikut adalah cara pembangkitannya :

1. Ambil suatu *state node* yang memiliki kata yang memiliki panjang N.
2. Mulai dari dari karakter pada indeks ke 0 tambahkan karakter tersebut dengan 1.
3. Bila kata yang dihasilkan dari penggantian tersebut merupakan kata dalam bahasa Inggris, maka akan ditambahkan pada array *child node* pada *state node*.
4. Bila bukan merupakan kata dalam bahasa Inggris, tidak melakukan apa.
5. Lakukan pergantian karakter sampai *rollback* ke karakter asal.
6. Lanjut ke karakter berikutnya dan ulangi hingga semua karakter sudah diperiksa satu - satu.

b. Algoritma *Uninformed Cost Search* (UCS)

Pada UCS, algoritma akan selalu menggunakan kata dengan *cost* terkecil, dimana *cost* disini berarti jumlah sisi yang perlu dilalui dari *root node* ke *expand node*. Algoritma ini termasuk *Uninformed Search* karena pada penentuan *cost* tidak ada fungsi heuristiknya. Untuk memudahkan implementasi algoritma ini, digunakan sebuah *priority queue*, pada kasus ini dinamakan “pq”. Berikut adalah algoritmanya:

1. Buat node dengan *cost* 0,*trail* sebagai array string yang berisi kata awal, dan kata awal yang dimasukkan.
2. Masukkan ke dalam pq.
3. Cek apakah panjang kata awal dan kata tujuan sama, bila sama lanjutkan, bila tidak, keluarkan exception.
4. Masuk ke dalam *main loop* yang akan berhenti bila *expand node* memiliki kata yang dicari.
5. Ambil *node* yang paling depan dari pq dan *assign* sebagai *expand node*.
6. Cek apakah kata yang ada pada *expand node* sama dengan kata tujuan, bila iya, keluar dari main loop, bila tidak lanjutkan.
7. Bangkitkan anak dari *expand node*.
8. Untuk tiap anak yang dibangkitkan, buatlah *state node* dengan kata baru yang dibangkitkan, *trail* sebagai *trail* pada *expand node* ditambah dengan kata yang dibangkitkan dan *cost* sebagai *cost* dari *expand node* ditambah satu.
9. Tambahkan tiap anak pada pq.
10. Ulangi dari langkah ke 5 sampai kata tujuan ditemukan.
11. Kembalikan trail.

c. Algoritma *Greedy Best First Search*

Algoritma *Greedy Best First Search* termasuk algoritma *Informed Search* yang berarti pada tiap *state* kita memiliki informasi mengenai *goal state*. Oleh karena itu, algoritma ini memiliki fungsi heuristik untuk menentukan *cost* dari tiap *node*. Perbedaan algoritma ini dengan algoritma lainnya adalah algoritma ini hanya mengandalkan fungsi heuristik (h(n)) untuk menentukan urutan traversal, dimana algoritma ini akan memilih *state node* dengan *cost* heuristik terkecil. Pada kasus penemuan solusi *word ladder* fungsi heuristiknya adalah banyak karakter yang berbeda pada indeks yang sama dari *expand node* dengan simpul tujuan. Berikut adalah algoritmanya :

1. Buat node dengan *cost* 0,*trail* sebagai array string yang berisi kata awal, dan kata awal yang dimasukkan.
2. Masukkan ke dalam pq.
3. Cek apakah panjang kata awal dan kata tujuan sama, bila sama lanjutkan, bila tidak, keluarkan exception.
4. Masuk ke dalam *main loop* yang akan berhenti bila *expand node* memiliki kata yang dicari.
5. Ambil *node* yang paling depan dari pq dan *assign* sebagai *expand node*.
6. Cek apakah kata yang ada pada *expand node* sama dengan kata tujuan, bila iya, keluar dari main loop, bila tidak lanjutkan.
7. Bangkitkan anak dari *expand node*.
8. Untuk tiap anak yang dibangkitkan, buatlah *state node* dengan kata baru yang dibangkitkan, *trail* sebagai *trail* pada *expand node* ditambah dengan kata yang dibangkitkan dan *cost* sebagai banyak karakter yang berbeda dari kata yang dibangkitkan dengan kata tujuan.
9. Tambahkan tiap anak pada pq.
10. Ulangi dari langkah ke 5 sampai kata tujuan ditemukan.
11. Kembalikan trail.

d. Algoritma A*

Algoritma A* juga termasuk algoritma *informed search*. Perbedaannya dengan algoritma *Greedy Best First Search* adalah selain menggunakan *heuristic cost*, algoritma ini juga menggunakan *actual cost* (cost dari *root node* ke *current node*, g(n)) pada penentuan *cost* suatu *node*. *Actual cost* dari suatu *state node* adalah berapa banyak sisi yang harus dilewati untuk sampai pada *node* tersebut, sedangkan *heuristic cost* nya sama dengan *heuristic cost* pada algoritma *Greedy Best First Search*. Bisa disimpulkan bahwa *cost* pada algoritma ini adalah *actual cost* ditambah dengan *heuristic cost*. Berikut adalah algoritmanya.

1. Buat node dengan *cost* 0,*trail* sebagai array string yang berisi kata awal, dan kata awal yang dimasukkan.
2. Masukkan ke dalam pq.
3. Cek apakah panjang kata awal dan kata tujuan sama, bila sama lanjutkan, bila tidak, keluarkan exception.

4. Masuk ke dalam *main loop* yang akan berhenti bila *expand node* memiliki kata yang dicari.
5. Ambil *node* yang paling depan dari pq dan *assign* sebagai *expand node*.
6. Cek apakah kata yang ada pada *expand node* sama dengan kata tujuan, bila iya, keluar dari main loop, bila tidak lanjutkan.
7. Bangkitkan anak dari *expand node*.
8. Untuk tiap anak yang dibangkitkan, buatlah *state node* dengan kata baru yang dibangkitkan, *trail* sebagai *trail* pada *expand node* ditambah dengan kata yang dibangkitkan dan *cost* sebagai *actual cost* ditambah dengan *heuristic cost*.
9. Tambahkan tiap anak pada pq.
10. Ulangi dari langkah ke 5 sampai kata tujuan ditemukan.
11. Kembalikan trail.

B. Analisis Algoritma

Bila diperhatikan, fungsi *cost* untuk semua algoritma yang dinyatakan di atas bisa dinyatakan sebagai fungsi f(n) = g(n) + h(n). Dimana f(n) adalah estimasi *cost* termurah untuk mendapatkan *solution path* yang melalui n. Pada algoritma UCS f(n) = g(n), pada algoritma *Greedy Best First Search*, f(n) = h(n) ,dan pada A* f(n) = h(n) + g(n). Pada kasus penemuan solusi *Word Ladder* g(n) didefinisikan sebagai jumlah perbedaan karakter dari kata awal ke kata n, sedangkan h(n) adalah jumlah karakter yang berbeda pada indeks yang sama dari kata n ke kata tujuan.Pada kasus *word ladder*, algoritma UCS sama dengan BFS Seperti yang telah disampaikan tadi pada pada algoritma UCS f(n) = g(n) dan cost dari anak yang memiliki parent sama seragam. Secara teoritis, algoritma A* lebih efisien dari UCS karena A* adalah ekstensi dari UCS yang memiliki heuristik. Nilai heuristik tersebut memang didesain untuk mendekatkan *node* dengan kata tujuan. Secara teoritis, *Greedy Best First Search* tidak menjamin solusi optimal untuk persoalan *Word Ladder* karena *Greedy Best First Search* tidak menjamin ditemukannya solusi. Bisa saja algoritma tersebut “terperangkap” dalam pencarian kata dengan membangkitkan anak yang membangkitkan anak lagi dengan kata yang sama dengan parentnya dan keduanya memiliki *heuristic cost* yang sama. pada kasus tersebut *Greedy Best First Search* akan *infinite loop.*

Definisi h(n) di atas bisa dibuktikan sebagai heuristik yang *admissible*. Untuk suatu heuristik dibilang *admissible*, heuristik tersebut harus memiliki sifat $h(n) \leq h^*(n)$, dimana h*(n) adalah cost sebenarnya yang diperlukan untuk mencapai *goal state* dari node n. Premis yang diperlukan adalah bahwa dalam permainan *Word Ladder* langkah minimum yang diperlukan untuk mencapai kata tujuan dari kata n adalah sebanyak beda karakter pada indeks yang sama dari kata n dengan kata tujuan. Hal tersebut dikarenakan pada permainan tersebut, langkah yang valid harus memiliki dua properti yaitu hanya berbeda 1 karakter dan merupakan kata dalam bahasa Inggris. Karena heuristik selalu mengasumsikan bahwa banyak langkah yang diperlukan adalah sebanyak langkah minimum yang diperlukan untuk mencapai kata tujuan dari kata n, dapat disimpulkan bahwa $h(n) \leq h^*(n)$ dan heuristik *admissible*.

C. Kode dan Penjelasan Program

a. StringCheck

```java
package State;

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

public class StringCheck {
    private HashSet<String> container;
    private String fileName;

    public StringCheck(String _fileName) {
        fileName = _fileName;
        container = new HashSet<>();
        try (BufferedReader br = new BufferedReader(new FileReader("src/State/" + fileName))) {
            String line;
            while ((line = br.readLine()) != null) {
                container.add(line.toUpperCase());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public boolean isValidWord(String word) {
        return container.contains(word);
    }
}
```

Gambar 1. StringCheck class

Fungsi utama pada class ini adalah untuk memeriksa apakah suatu kata adalah kata dalam bahasa Inggris yang valid. Class ini memiliki 2 buah atribut yaitu container yang merupakan *HashSet* yang menyimpan kata - kata yang di load dari file. Nama file tersebut terdapat pada atribut ke 2 yaitu fileName. Konstruktor pada class ini menerima String nama file yang akan dibaca untuk membuat container. Method yang digunakan untuk menentukan apakah katanya benar atau tidak adalah isValidWord(String word).

b. StateNode

```java
package State;

import java.util.*;

public class StateNode implements Comparable<StateNode> {
    private String name;
    private int cost;
    private ArrayList<String> trail;
    private ArrayList<String> children;

    public StateNode(String _name, int _cost, ArrayList<String> _trail, boolean
makeChild) {
        name = _name;
        cost = _cost;
        trail = _trail;
        if (makeChild) {
            children = makeChildren(name);
        } else {
            children = new ArrayList<String>();
        }
    }

    public StateNode(StateNode sn) {
        name = sn.name;
        cost = sn.cost;
        trail = sn.trail;
        children = sn.children;
    }

    // getter
    public String getName() {
        return name;
    }

    public int getCost() {
        return cost;
    }

    public ArrayList<String> getTrail() {
        return trail;
    }

    public ArrayList<String> getChildren() {
        return children;
    }

    // setter or modifier
    public void setName(String _name) {
        this.name = _name;
    }

    public void setCost(int _cost) {
        this.cost = _cost;
    }

    public ArrayList<String> addTrail(String s) {
        ArrayList<String> returnValue = trail;
        returnValue.add(s);
        return returnValue;
    }

    public void setChildren() {
        children = makeChildren(name);
    }

    // method
    public ArrayList<String> makeChildren(String _name) {
        StringCheck checker = new StringCheck("words_alpha.txt");
        ArrayList<String> children = new ArrayList<String>();

        for (int i = 0; i < _name.length(); i++) {
            for (int j = 1; j <= 26; j++) {
                StringBuilder sb = new StringBuilder(_name);
                char shiftedChar = (char) (((_name.charAt(i) - 'A' + j) % 26) + 'A');
                sb.setCharAt(i, shiftedChar);
                String possibleChild = sb.toString();
                if (checker.isValidWord(possibleChild) && j != 26) {
                    children.add(possibleChild);
                }
            }
        }
        return children;
    }

    public void printChildren() {
        for (int i = 0; i < children.size(); i++) {
            System.out.println(children.get(i).toString());
        }
    }

    public int compareTo(StateNode other) {
        if (cost == other.cost) {
            return 0;
        } else if (cost < other.cost) {
            return -1;
        } else {
            return 1;
        }
    }
}
```

Gambar 2. StateNode Class

Kelas StateNode adalah kelas yang digunakan untuk merepresentasikan sebuah state node pada algoritma path finding. Terdiri dari atribut name yang berisi kata yang menjadi perhatian, cost yang merupakan harga dari node tersebut (ditentukan sesuai dengan algoritma pencarian yang digunakan), trail yang menampung nama - nama node dari root ke node sekarang, dan children yang menampung nama - nama anak dari node tersebut.

Kelas ini memiliki konstruktor yang memungkinkan dibuat atau tidak dibangkitkannya anak pada suatu node. Kelas ini juga memiliki getter dan setter untuk atribut yang bersesuaian. Method makeChildren digunakan untuk mendapatkan anak - anak dari suatu node sedangkan printChildren digunakan untuk mem-print anak - anak dari suatu node, digunakan untuk keperluan testing. Kelas ini juga mengimplementasikan *interface* Comparable yang akan membandingkan berdasarkan cost dari *state node*.

c. Hasil

```java
package Hasil;
import java.util.*;

public class Hasil {
    public ArrayList<String> Path;
    public int ammountOfNode;

    public Hasil(ArrayList<String> _path,int amm) {
        Path = _path;
        ammountOfNode = amm;
    }

    public Hasil() {
        Path = new ArrayList<String>();
        ammountOfNode = 0;
    }
}
```

Gambar 3. Hasil Class

Kelas Hasil dibuat untuk menampung *return value* dari algoritma dan dibuat seperti sebuah *struct* dimana semua attributnya public. Path menampung solusi kata - kata yang digunakan untuk sampai ke kata tujuan dari kata awal. Atribut ammountOfNode menampung banyak *node* yang dilalui oleh algoritma untuk sampai ke kata tujuan.

d. UCS

```java
public class UCS {
    private Queue<StateNode> urutanTraversal;

    //constructor
    public UCS() {
        urutanTraversal = new PriorityQueue<StateNode>();
    }

    public Hasil find(String origin, String target) {
        int nodeAmmount = 0;
        ArrayList<String> temp = new ArrayList<String>();
        temp.add(origin);
        Hasil retVal = new Hasil(temp,nodeAmmount);
        StateNode seed = new StateNode(origin,0,temp,false);
        urutanTraversal.add(seed);
        boolean found = false;
        if(origin.length() == target.length()) {
            StateNode expandNode = new StateNode(urutanTraversal.poll());
            while(!found){
                if(nodeAmmount != 0) {expandNode = new StateNode(urutanTraversal.poll());}
                System.out.println(expandNode.getName());
                if(expandNode.getName().equals(target) ) {
                    found = true;
                } else {
                    nodeAmmount++;
                    expandNode.setChildren();
                    ArrayList<String> enChildren = expandNode.getChildren(); //expand node children
                    for(int i=0;i<enChildren.size();i++) {
                        ArrayList<String> tempTrail = new ArrayList<>(expandNode.getTrail());
                        tempTrail.add(enChildren.get(i));
                        StateNode child = new StateNode(enChildren.get(i),expandNode.getCost()+1,tempTrail,false);
                        urutanTraversal.add(child);
                    }
                }
            }
            retVal.Path = expandNode.getTrail();
            retVal.ammountOfNode = nodeAmmount;
        }
        return retVal;
    }
}
```

Gambar 5. UCS Class

Kelas UCS memiliki atribut urutanTraversal yang merupakan PriorityQueue<StateNode> yang dimanfaatkan dari java.util. Find merupakan *method* yang digunakan untuk mencari solusi, algoritma sudah dijelaskan di atas. Poin yang ingin *dihighlight* pada *method* find di UCS adalah cost untuk anaknya didefinisikan sebagai expandNode.getCost() +1.

e. GBFS

```java
public class GBFS {

    private Queue<StateNode> urutanTraversal;

    public GBFS() {
        urutanTraversal = new PriorityQueue<StateNode>();
    }

    public int heuristicCost(String current,String target) {
        int cost =0;
        for(int i=0; i < current.length();i++) {
            if(current.charAt(i) != target.charAt(i)) {
                cost++;
            }
        }
        return cost;
    }

    public Hasil find(String origin, String target) {
        int nodeAmmount = 0;
        ArrayList<String> temp = new ArrayList<String>();
        temp.add(origin);
        Hasil retVal = new Hasil(temp,nodeAmmount);
        int originalCost = heuristicCost(origin,target);
        StateNode seed = new StateNode(origin,originalCost,temp,false);
        urutanTraversal.add(seed);
        boolean found = false;
        if(origin.length() == target.length()) {
            StateNode expandNode = new StateNode(urutanTraversal.poll());
            while(!found) {
                if(nodeAmmount != 0) {expandNode = new StateNode(urutanTraversal.poll());}
                System.out.println(expandNode.getName());
                if(expandNode.getName().equals(target) ) {
                    found = true;
                } else {
                    nodeAmmount++;
                    expandNode.setChildren();
                    ArrayList<String> enChildren = expandNode.getChildren();
                    for(int i=0;i<enChildren.size();i++) {
                        ArrayList<String> tempTrail = new ArrayList<>(expandNode.getTrail());
                        tempTrail.add(enChildren.get(i));
                        int tempCost = heuristicCost(enChildren.get(i),target);
                        StateNode child = new StateNode(enChildren.get(i),tempCost,tempTrail,false);
                        urutanTraversal.add(child);
                    }
                }
            }
            retVal.Path = expandNode.getTrail();
            retVal.ammountOfNode = nodeAmmount;
        }
        return retVal;
    }
}
```

Gambar 6. GBFS Class

Kelas yang merupakan penerapan dari algoritma *Greedy Best First Search*.Kelas GBFS memiliki atribut urutanTraversal yang merupakan PriorityQueue<StateNode> yang dimanfaatkan dari java.util. Kelas ini memiliki method untuk mencari nilai heuristik dari suatu node, dengan spesifikasi sudah dijelaskan di bab sebelumnya. Find merupakan *method* yang digunakan untuk mencari solusi, algoritma sudah dijelaskan di atas. Poin yang ingin *dihighlight* pada *method* find di GBFS adalah cost untuk anaknya didefinisikan sebagai nilai heuristic dari node tersebut ke kata tujuan.

f. Astar

```java
public class Astar {
    private Queue<StateNode> urutanTraversal;

    public Astar() {
        urutanTraversal = new PriorityQueue<StateNode>();
    }

    public int heuristicCost(String current,String target) {
        int cost =0;
        for(int i=0; i < current.length();i++) {
            if(current.charAt(i) != target.charAt(i)) {
                cost++;
            }
        }
        return cost;
    }

    public Hasil find(String origin, String target) {
        int nodeAmmount = 0;
        ArrayList<String> temp = new ArrayList<String>();
        temp.add(origin);
        Hasil retVal = new Hasil(temp,nodeAmmount);
        int originalCost = heuristicCost(origin,target);
        StateNode seed = new StateNode(origin,originalCost,temp,false);
        urutanTraversal.add(seed);
        boolean found = false;
        if(origin.length() == target.length()) {
            StateNode expandNode = new StateNode(urutanTraversal.poll());
            while(!found) {
                if(nodeAmmount != 0) {expandNode = new StateNode(urutanTraversal.poll());}
                System.out.println(expandNode.getName());
                if(expandNode.getName().equals(target) ) {
                    found = true;
                } else {
                    nodeAmmount++;
                    expandNode.setChildren();
                    ArrayList<String> enChildren = expandNode.getChildren();
                    for(int i=0;i<enChildren.size();i++) {
                        ArrayList<String> tempTrail = new ArrayList<>(expandNode.getTrail());
                        tempTrail.add(enChildren.get(i));
                        int hc = heuristicCost(enChildren.get(i),target);
                        int gc = (expandNode.getCost() - heuristicCost(expandNode.getName(),target))+1;
                        StateNode child = new StateNode(enChildren.get(i),hc + gc,tempTrail,false);
                        urutanTraversal.add(child);
                    }
                }
            }
            retVal.Path = expandNode.getTrail();
            retVal.ammountOfNode = nodeAmmount;
        }
        return retVal;
    }
```

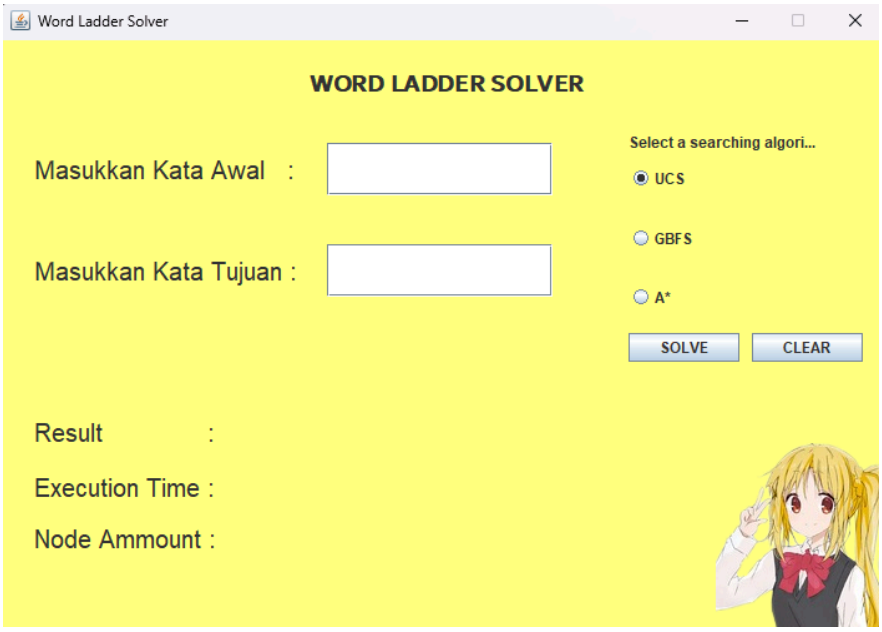
Gambar 6. GBFS Class

Kelas yang merupakan penerapan dari algoritma A*.Kelas A* memiliki atribut urutanTraversal yang merupakan PriorityQueue<StateNode> yang dimanfaatkan dari java.util. Kelas ini memiliki method untuk mencari nilai heuristik dari suatu node, dengan spesifikasi sudah dijelaskan di bab sebelumnya. Find merupakan *method* yang digunakan untuk mencari solusi, algoritma sudah dijelaskan di atas. Poin yang ingin *dihighlight* pada *method* find di GBFS adalah cost untuk anaknya didefinisikan sebagai nilai heuristic dari node tersebut ke kata tujuan.

g. FindingUI

Bonus yang dibuat, merupakan UI dari program ini yang dikembangkan menggunakan kakas Java Swing. Karena terlalu besar, diputuskan untuk tidak menaruh *snippet*-nya pada laporan ini. Source code dari kelas FindingUI bisa ditemukan di folder src pada repository. Kelas ini mengandung main method dari program ini. Beberapa komponen yang penting terdiri atas fieldTujuan dan fieldAwal yang merupakan objek

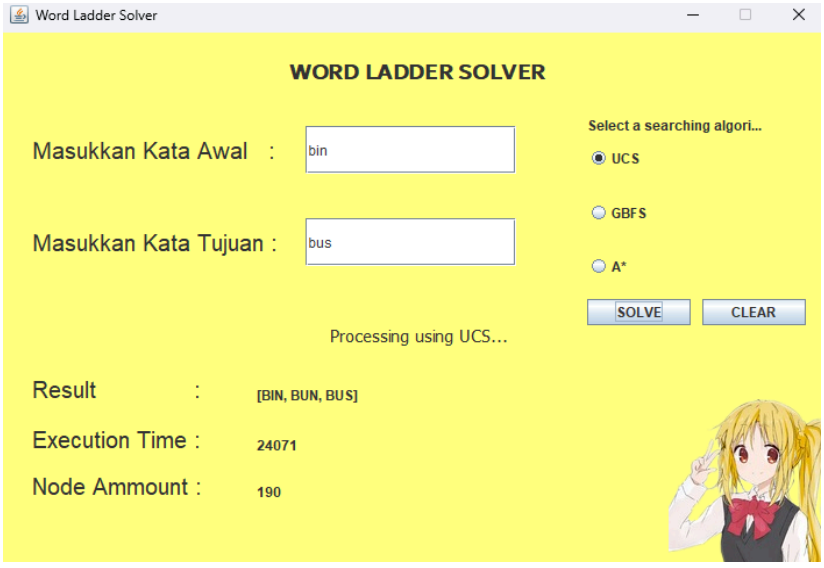
JTextField yang digunakan untuk menginput kata awal dan kata tujuan. Beberapa JRadioButton untuk memilih algoritma yang ingin dipakai untuk membuat solusi. Tombol solve yang akan memulai algoritma. Tombol clear yang akan mengosongkan semua field dan hasil. Output dari kelas ini adalah path dari kata awal ke kata tujuan, jumlah node, dan waktu eksekusi dalam milisecond.



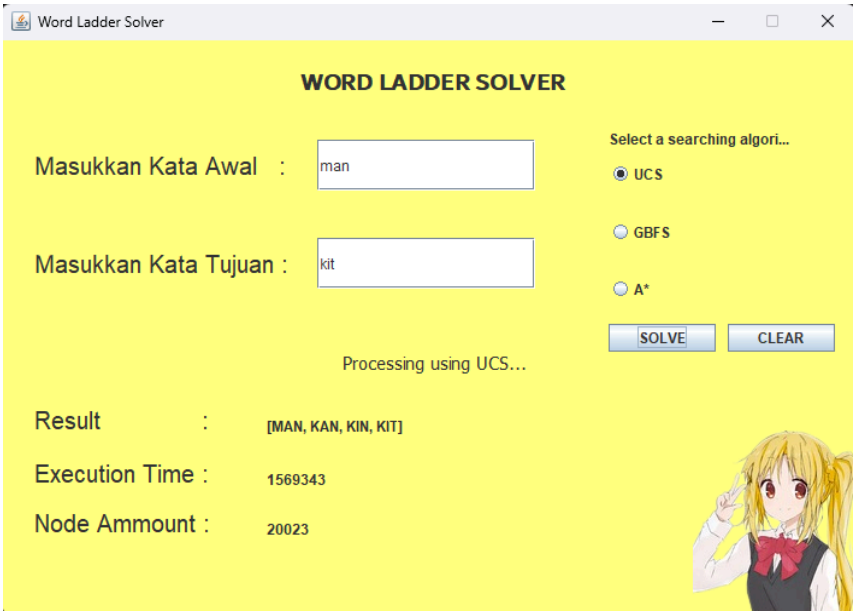
Gambar 7. Tampilan UI dalam Kondisi Kosong Featuring Nijika Ijichi

D. Hasil Testing

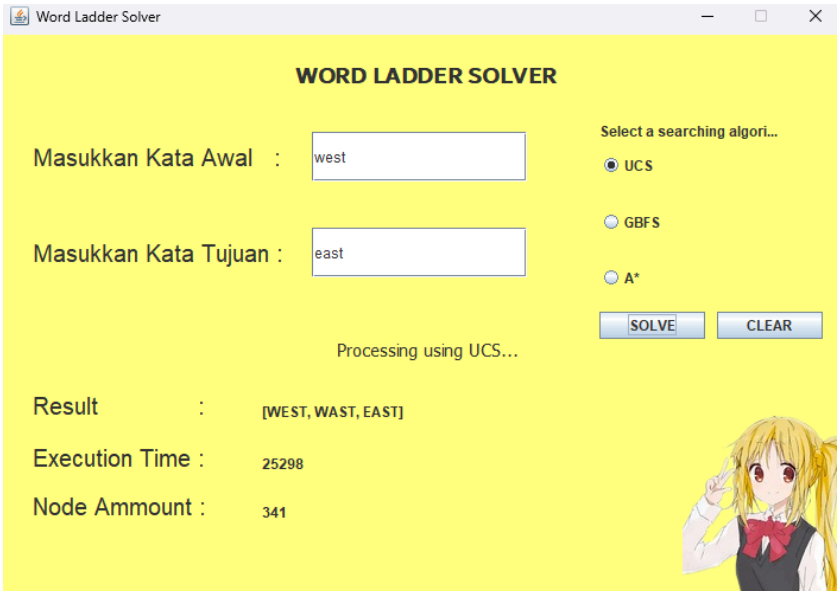
a. Test Algoritma UCS



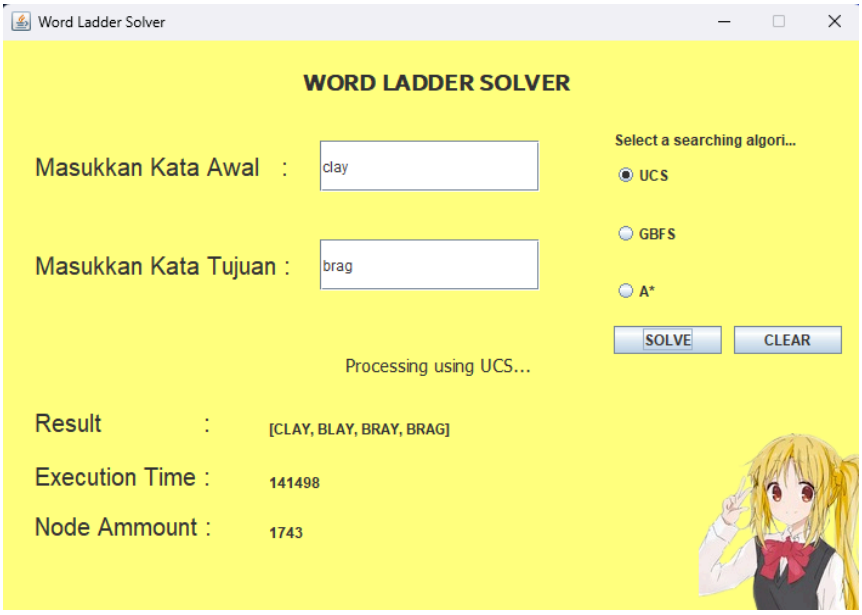
Gambar 8. Test 1 UCS



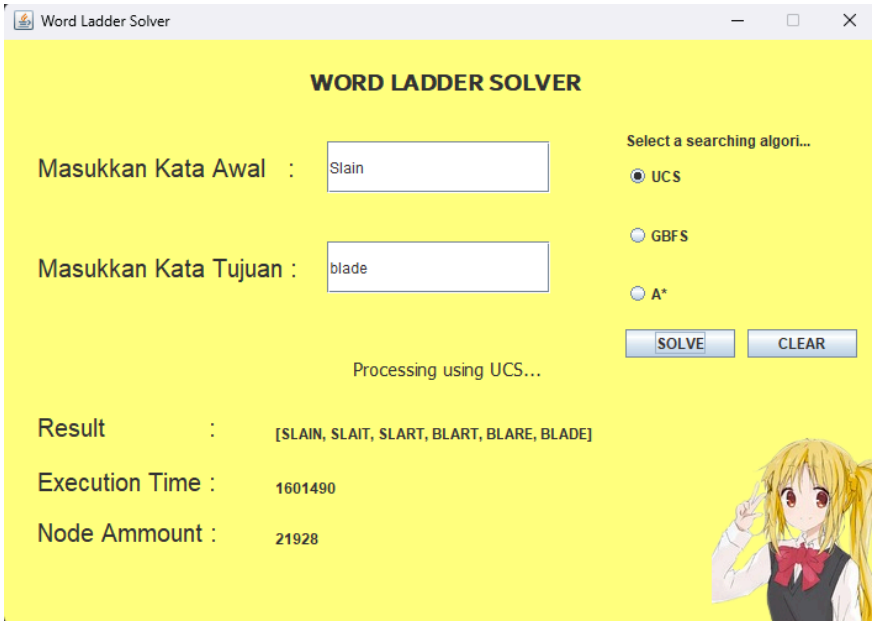
Gambar 9. Test 2 UCS



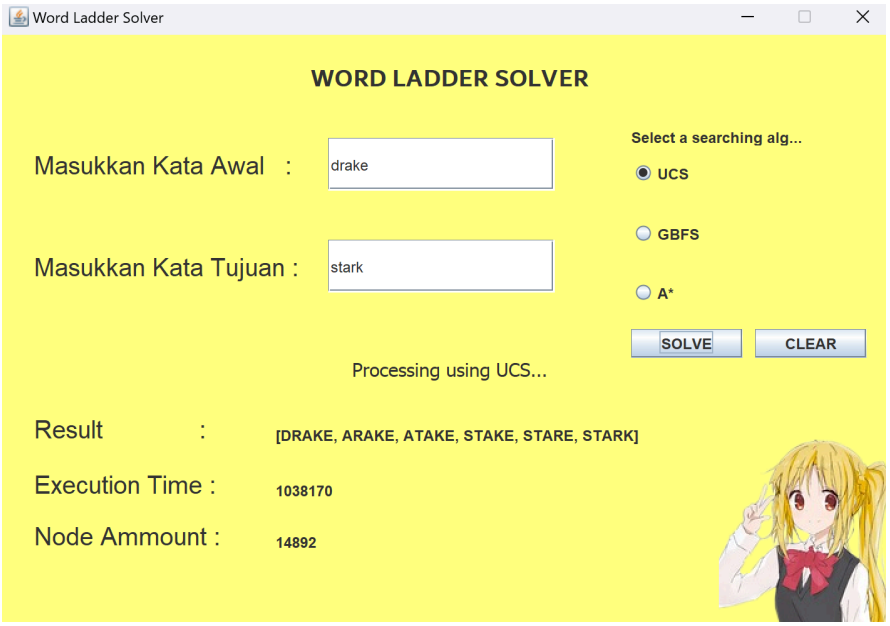
Gambar 10. Test 3 UCS



Gambar 11. Test 4 UCS

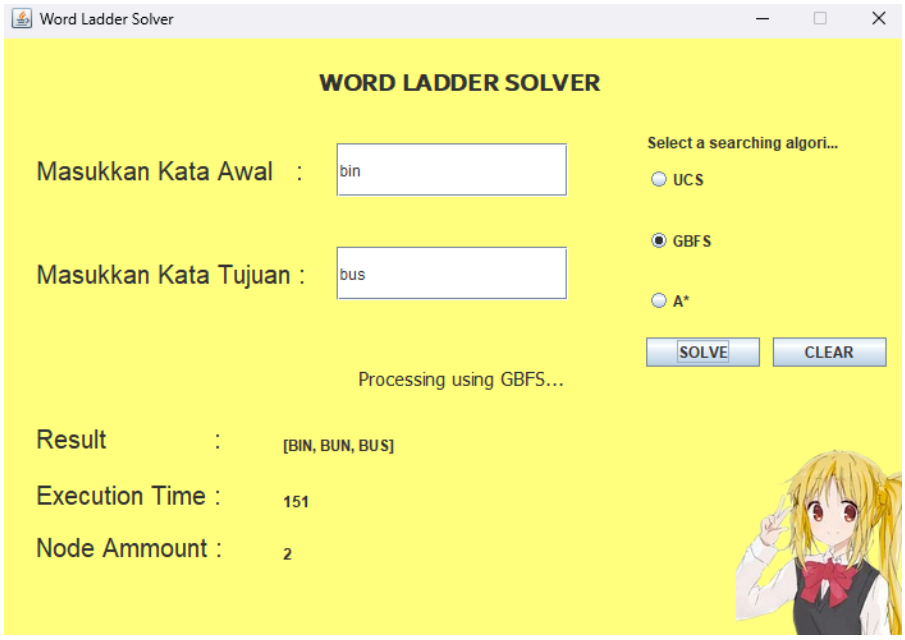


Gambar 12. Test 5 UCS

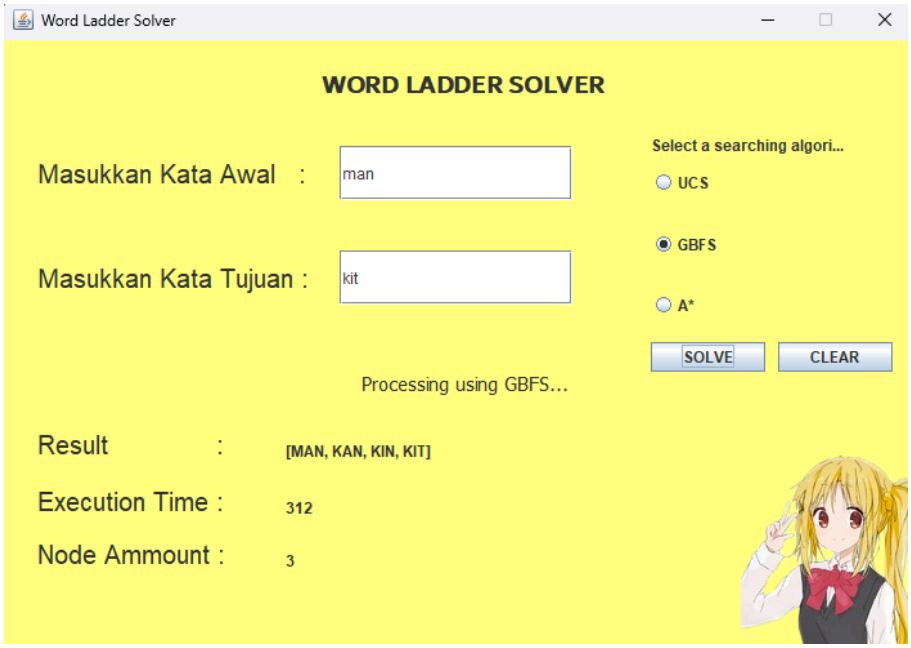


Gambar 13. Test 6 UCS

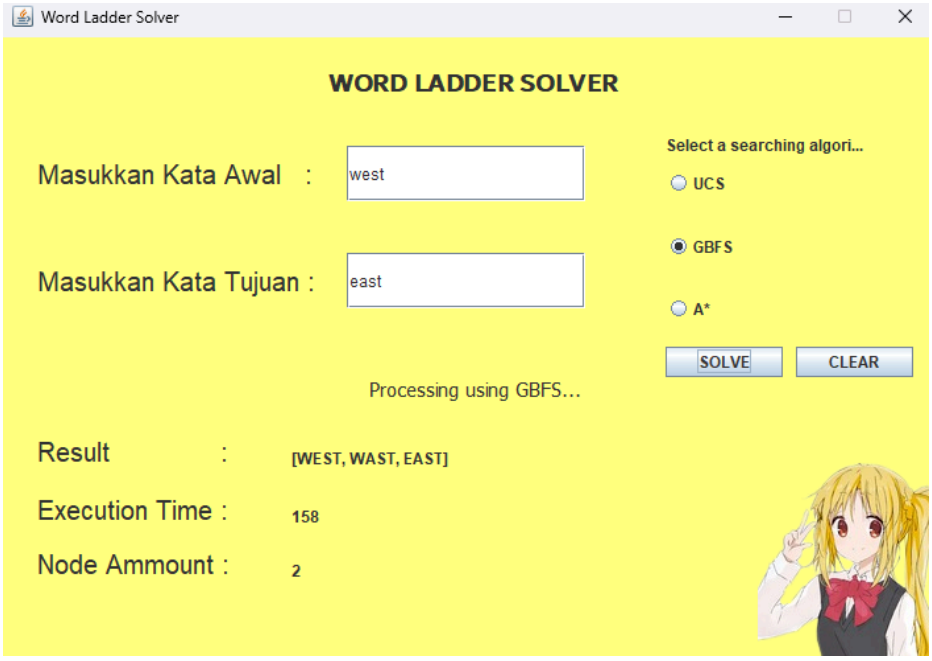
b. Test Algoritma Greedy Best First Search



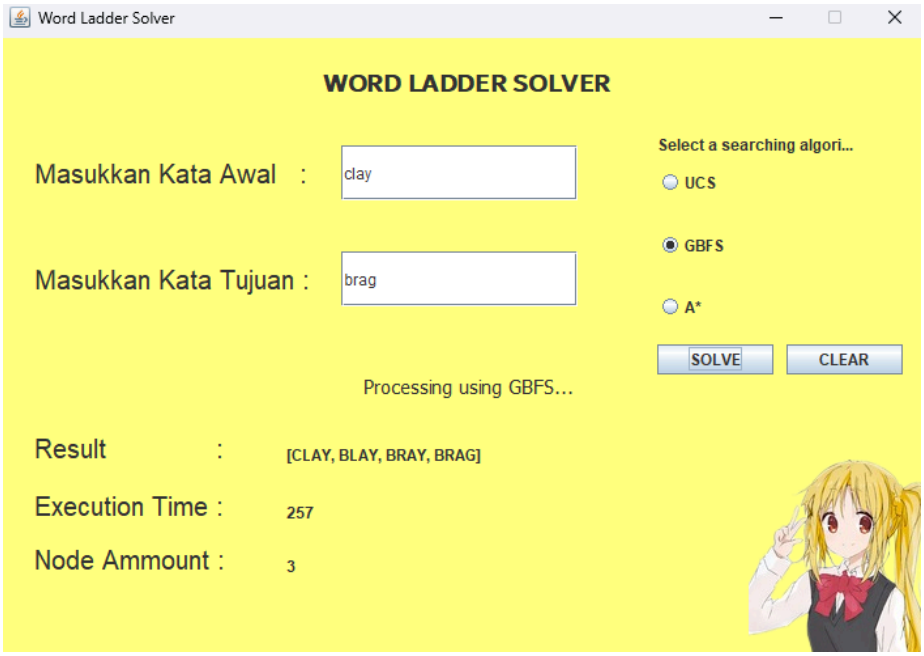
Gambar 14. Test 1 GBFS



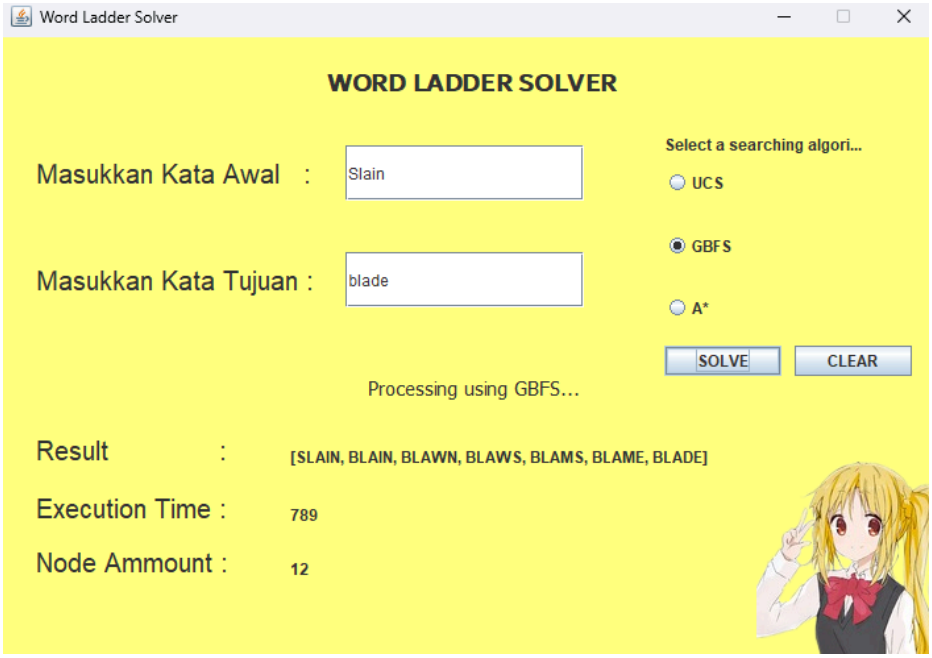
Gambar 15. Test 2 GBFS



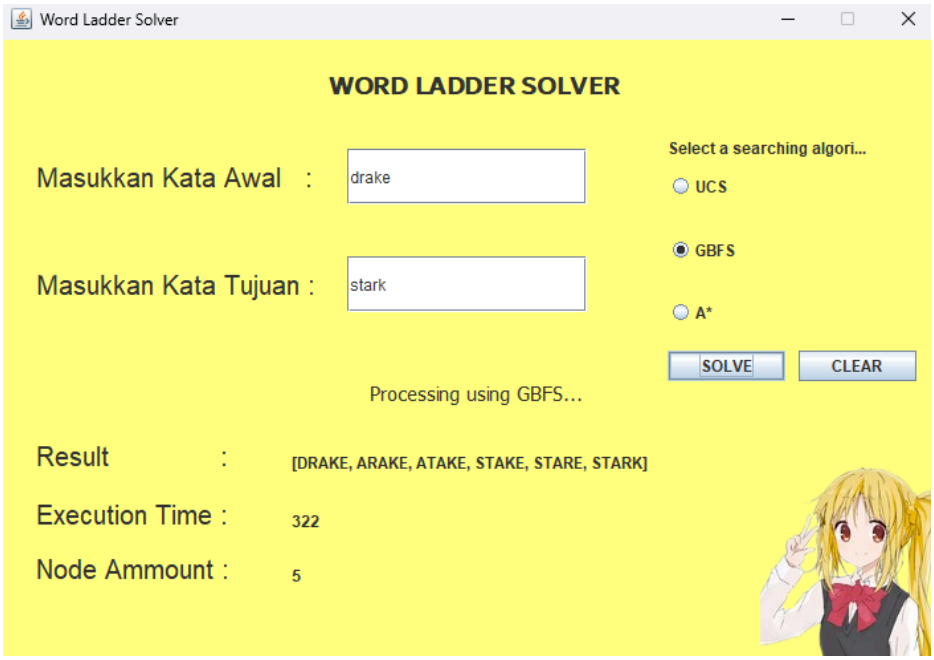
Gambar 16. Test 3 GBFS



Gambar 17. Test 4 GBFS

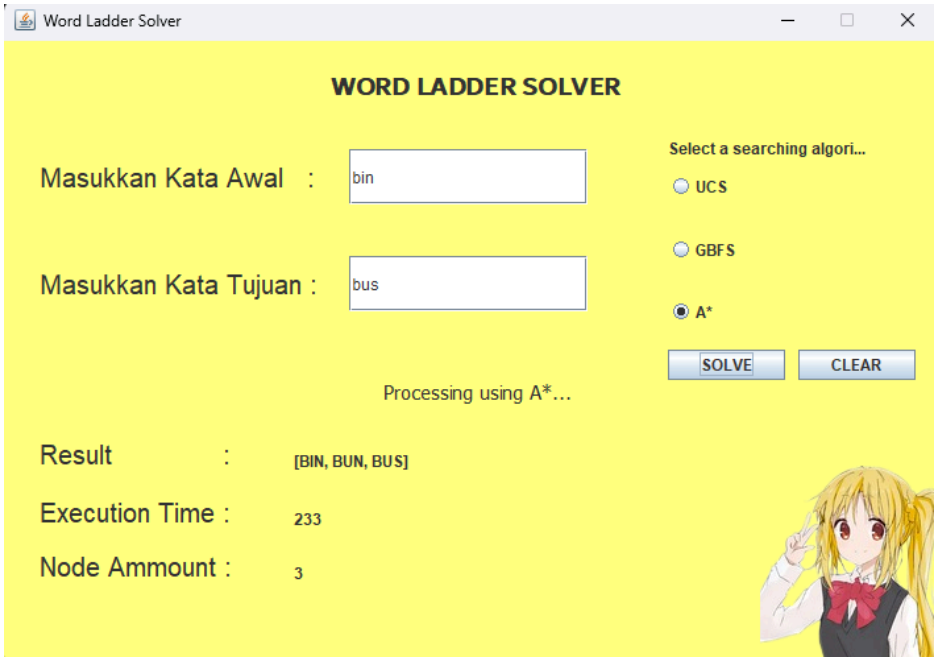


Gambar 18. Test 5 GBFS

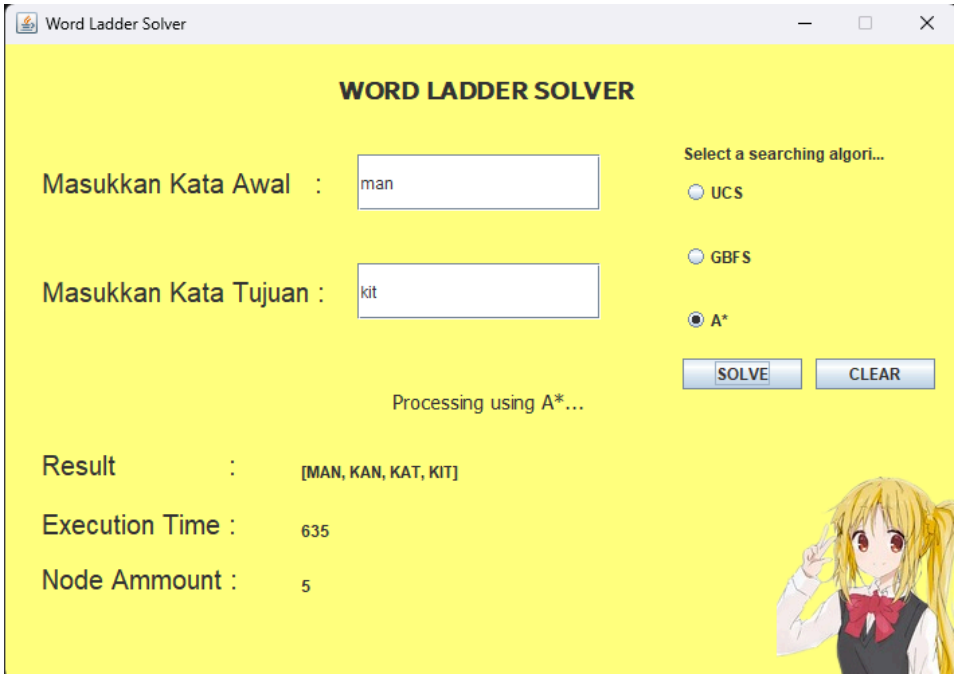


Gambar 19. Test 6 GBFS

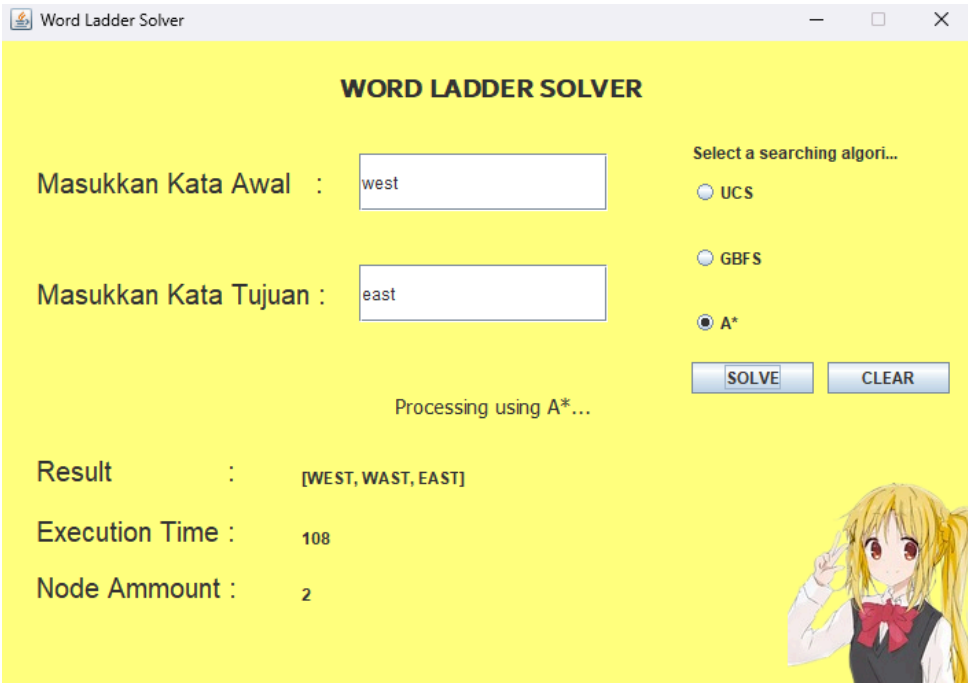
c. Test Algoritma A*



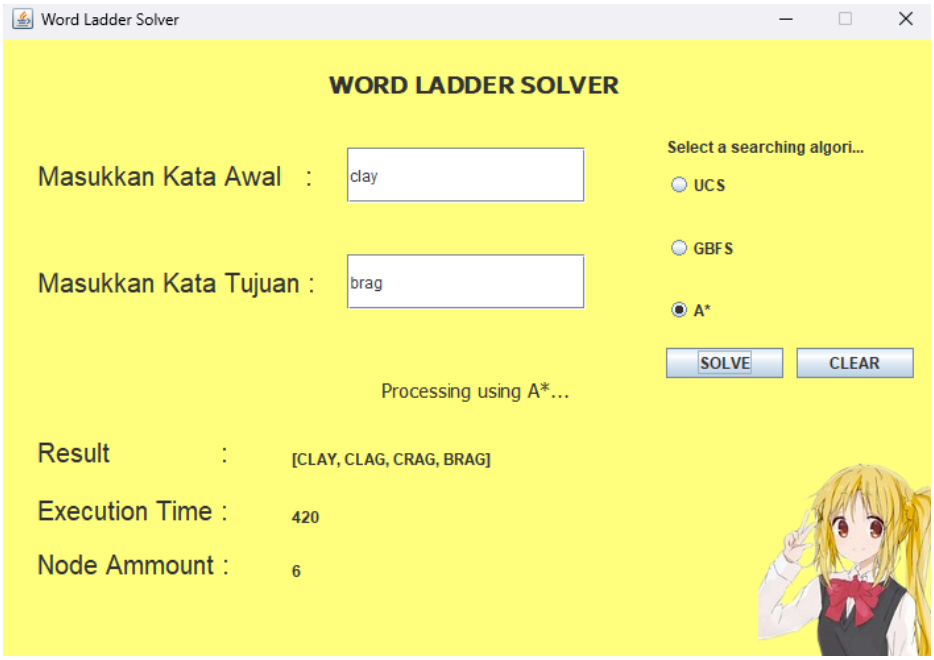
Gambar 20. Test 1 A*



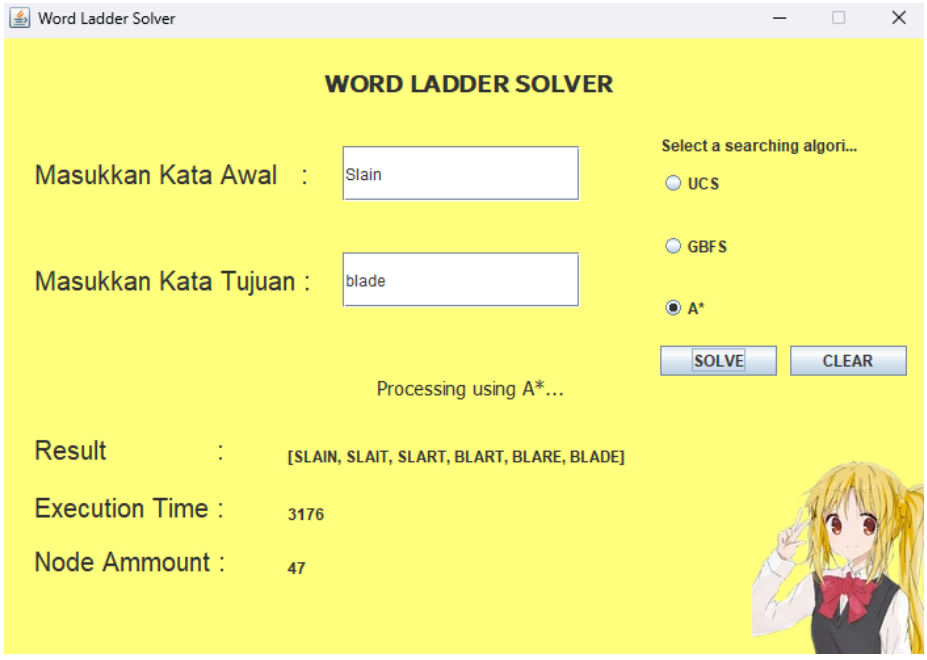
Gambar 21. Test 2 A*



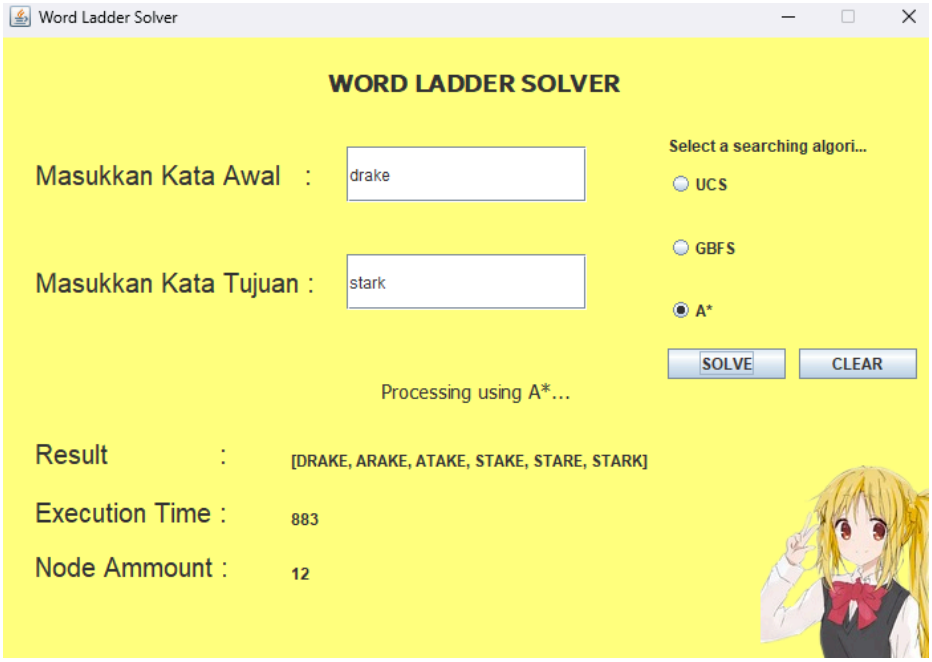
Gambar 22. Test 3 A*



Gambar 23. Test 4 A*

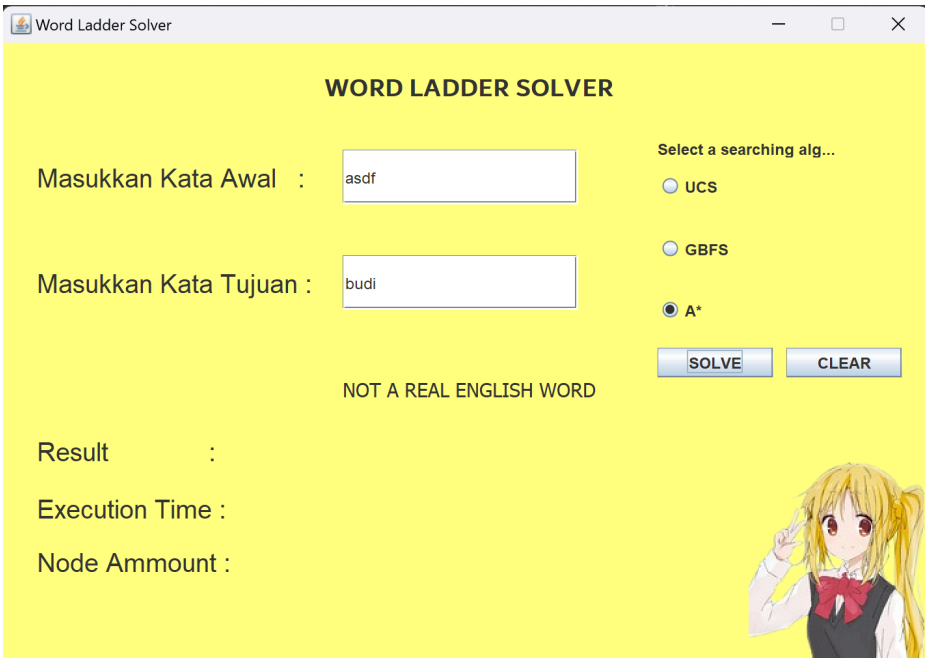


Gambar 24. Test 5 A*

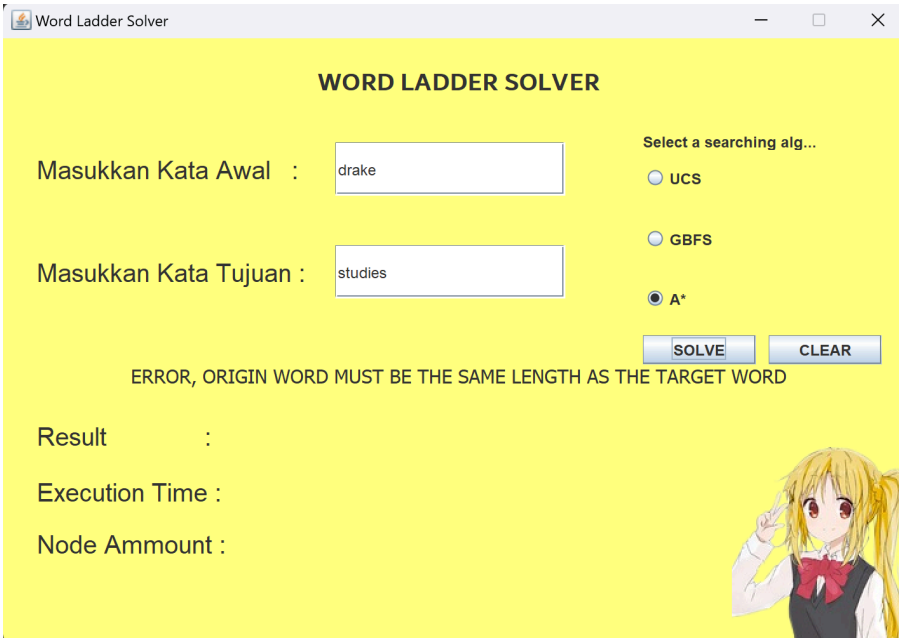


Gambar 25. Test 6 A*

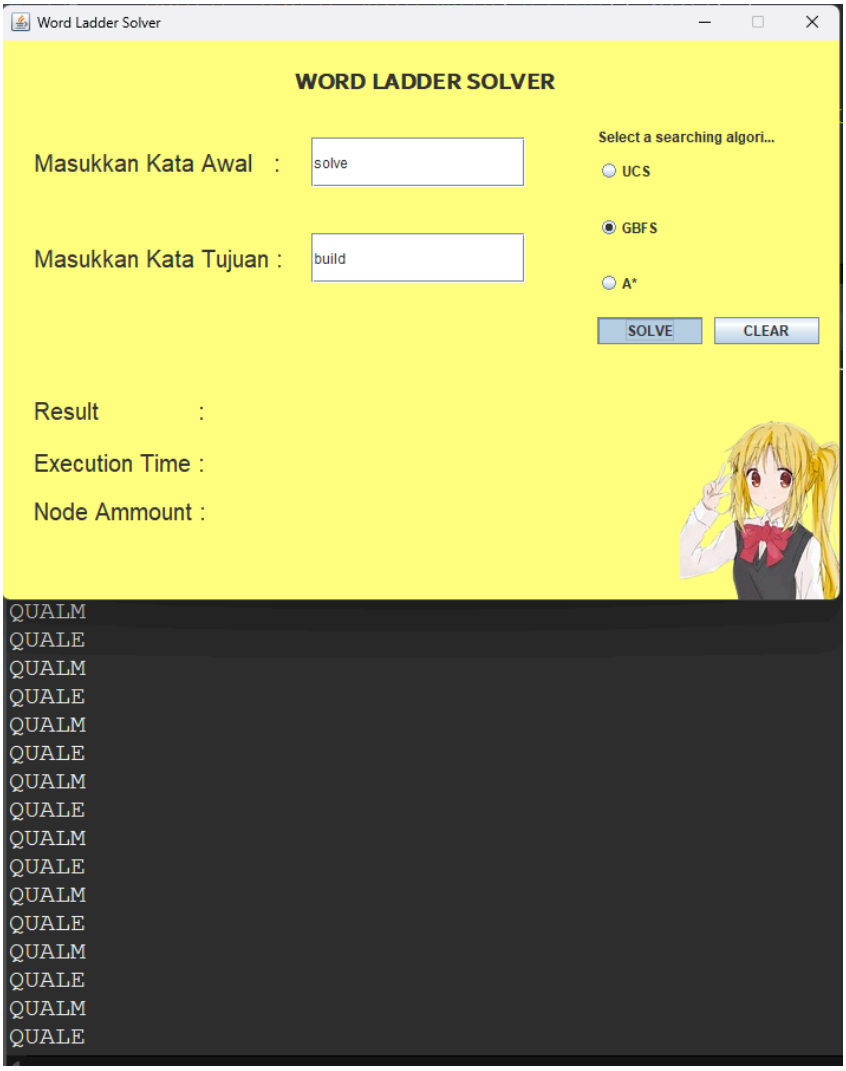
d. Special Case



Gambar 26. Kasus Memasukkan Kata yang Salah



Gambar 27. Kasus Memasukkan 2 Kata dengan Panjang yang Berbeda



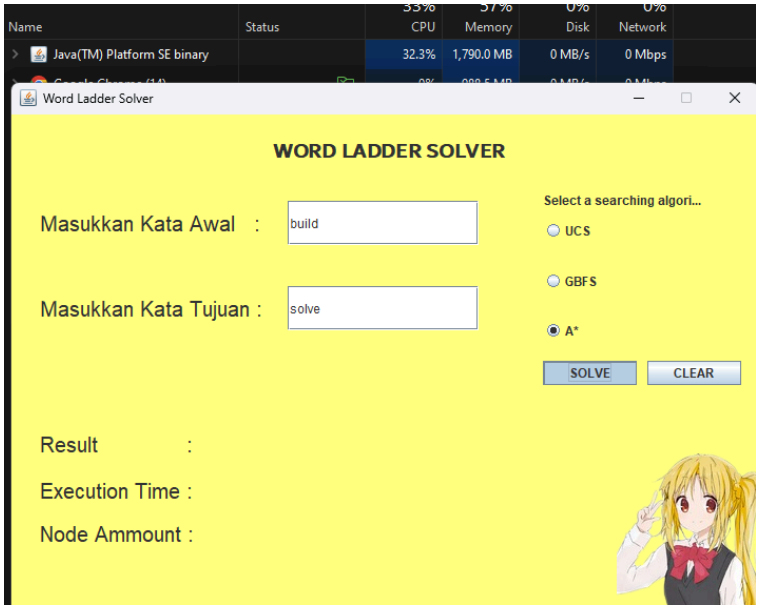
Gambar 28. Kasus GBFS Infinite Loop

E. Analisis Testing

Dari hasil testing terlihat bahwa algoritma UCS dan algoritma A* akan menghasilkan solusi yang optimal, sedangkan GBFS tidak selalu menghasilkan solusi yang optimal, atau bahkan tidak menemukan solusi sama sekali. Hal tersebut bisa diperhatikan pada test case 5 slain → blade , dimana pada algoritma UCS dan A* hanya memerlukan 4 kata antara untuk mencapai tujuan dari kata awal, sedangkan pada GBFS memerlukan 5 kata antara. Pada gambar 28 juga terlihat bahwa pada algoritma GBFS terdapat kemungkinan untuk *infinite loop*. Bila ditelusuri, alasan UCS selalu menghasilkan solusi yang optimal adalah karena pada dasarnya algoritma UCS ini sama persis dengan BFS dimana algoritma akan memeriksa *layer by layer*, serta algoritma A* akan selalu menghasilkan solusi yang optimal karena memiliki heuristik yang *admissible*.

Tetapi hal tersebut bukan berarti bahwa GBFS adalah algoritma yang jelek. Bila diperhatikan, GBFS memiliki rata - rata waktu eksekusi sebesar 331,5 millisecond. Rata

- rata waktu eksekusi A* adalah 909,167 millisecond sedangkan UCS adalah 771262 millisecond atau sekitar 12,8 menit. Hal tersebut sesuai dengan teori karena walau ketiga search algorithm memiliki *worst-case time complexity* yang eksponensial dengan *branching factor* sebagai basis, heuristik pada GBFS dan A* akan sangat membantu dalam memandu ke solusi.



Gambar 29. Penggunaan Memori Saat Algoritma Sedang Dijalankan

Terlihat pula bahwa Penggunaan memori saat algoritma sedang bekerja adalah sekitar 1700 sampai 2000 MB pada kondisi terburuknya.

F. Lampiran

Repository GitHub : https://github.com/Leaguemen/Tucil3_13522078.git

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	√	