

**LAPORAN TUGAS KECIL III IF2211 STRATEGI ALGORITMA
SEMESTER II 2023/2024**

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

Ibrahim Ihsan Rasyid 13522018



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

DAFTAR ISI

DAFTAR ISI	2
BAB I	3
BAB II	4
A. Teori Singkat Algoritma UCS, Greedy Best First Search, dan A*	4
B. Algoritma UCS, Greedy Best First Search, dan A* dalam Menyelesaikan Persoalan	4
BAB III	7
A. Desain Program dalam Bahasa Java	7
B. Implementasi Algoritma UCS	12
C. Implementasi Algoritma Greedy Best First Search	13
D. Implementasi Algoritma A*	14
E. Implementasi Graphical User Interface (GUI) Program	15
BAB IV	18
A. Uji Coba	18
B. Analisis Hasil Uji Coba	28
LAMPIRAN	29

BAB I

DESKRIPSI MASALAH

Word ladder (juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragraphs, laddergrams, atau word golf) adalah salah satu permainan kata yang terkenal bagi seluruh kalangan. Word ladder ditemukan oleh Lewis Carroll, seorang penulis dan matematikawan, pada tahun 1877. Pada permainan ini, pemain diberikan dua kata yang disebut sebagai start word dan end word. Untuk memenangkan permainan, pemain harus menemukan rantai kata yang dapat menghubungkan antara start word dan end word. Banyaknya huruf pada start word dan end word selalu sama. Tiap kata yang berdekatan dalam rantai kata tersebut hanya boleh berbeda satu huruf saja. Pada permainan ini, diharapkan solusi optimal, yaitu solusi yang meminimalkan banyaknya kata yang dimasukkan pada rantai kata. Berikut adalah ilustrasi serta aturan permainan.

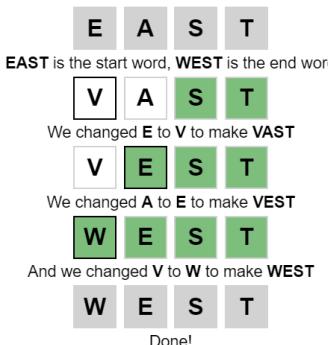
How To Play

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

Rules

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

Example



Gambar 1.1. Ilustrasi Permainan Word Ladder
(Sumber: <https://wordwormdormdork.com/>)

BAB II

ANALISIS PERSOALAN DENGAN ALGORITMA

A. Teori Singkat Algoritma UCS, Greedy Best First Search, dan A*

Algoritma Uniform Cost Search (UCS) adalah salah satu algoritma pencarian tanpa informasi (uninformed search atau blind search) yang menggunakan biaya kumulatif terendah untuk mencari jalur dari suatu node sumber ke node tujuan dalam graf berbobot. Algoritma UCS masuk dalam algoritma uninformed search karena bekerja tanpa mempertimbangkan keadaan simpul atau ruang pencarian. Rumus penentuan prioritas untuk menentukan simpul ekspan dari simpul hidup adalah $f(n) = g(n)$, dengan $g(n)$ adalah biaya menuju simpul n.

Algoritma Greedy Best First Search adalah salah satu algoritma pencarian yang menggunakan fungsi heuristik untuk menentukan jalur yang akan dilalui dalam pencarinya. Fungsi heuristik yang biasa digunakan dalam pencarian dengan Greedy Best First Search adalah seberapa “dekat” setiap node berikutnya ke tujuan, lalu algoritma akan memilih simpul dengan perhitungan heuristik terkecil. Rumus penentuan prioritas untuk menentukan simpul ekspan dari simpul hidup adalah $f(n) = h(n)$, dengan $h(n)$ adalah estimasi jarak dari simpul n ke simpul tujuan.

Algoritma A* adalah algoritma pencarian yang mampu menemukan jalur terpendek dari simpul awal ke simpul tujuan sambil meminimalkan biaya total dari jalur yang diambil untuk mencapai tujuan. Rumus penentuan prioritas untuk menentukan simpul ekspan dari simpul hidup adalah $f(n) = g(n) + h(n)$, dengan $g(n)$ adalah biaya menuju simpul n dan $h(n)$ adalah estimasi jarak dari simpul n ke simpul tujuan.

B. Algoritma UCS, Greedy Best First Search, dan A* dalam Menyelesaikan Persoalan

Persoalan permainan word ladder dapat diperlakukan sebagai sebuah graf berbobot. Sebuah kata yang memiliki definisi dianggap sebagai sebuah simpul, dan tiap kata yang memiliki perbedaan tepat satu huruf dianggap sebagai simpul tetangganya. Sebagai contoh, kata “mine” dan “mind” bertetangga karena memiliki tepat satu huruf yang berbeda yaitu “e” dan “d”. Namun, kata “minz” bukan tetangga dari “mine” karena kata “minz” tidak terdefinisi (bahkan tidak termasuk simpul dalam graf).

Algoritma UCS akan memeriksa simpul yang memiliki rute dari simpul awal terlebih dahulu serta biaya dari rute simpul tersebut. Dalam menentukan simpul ekspan yang akan diambil, setiap simpul hidup dimasukkan ke dalam priority queue yang diurutkan berdasarkan biaya yang terkecil, yaitu banyaknya perbedaan huruf dari kata awal. Apabila sebuah simpul ekspan diperiksa, maka dibuat lagi simpul hidup dari simpul

ekspan tersebut. Pencarian dilakukan hingga simpul tujuan tercapai atau priority queue kosong yang menandakan bahwa jalur tidak ditemukan

Algoritma Greedy Best First Search akan memeriksa simpul dengan jarak menuju simpul tujuan yang terkecil, yaitu banyaknya perbedaan huruf dengan simpul tujuan. Setiap simpul yang dikunjungi, algoritma tidak akan melakukan backtracking dan akan melupakan simpul-simpul sebelumnya. Dengan demikian, ada kemungkinan bahwa pencarian dengan Greedy Best First Search tidak menemukan solusi karena terjebak dalam optimum lokal.

Algoritma A* melakukan pencarian dengan menggabungkan dua faktor pada UCS dan Greedy Best First Search, yaitu biaya dari simpul awal dan jarak menuju simpul tujuan. Sama seperti UCS, algoritma akan menggunakan priority queue dengan mekanisme pemilihan simpul ekspan yang serupa. Namun, pengurutan dilakukan berdasarkan jumlah biaya dan jarak.

Pada ketiga algoritma pencarian yang akan diimplementasikan, ketiganya menggunakan fungsi $f(n)$ untuk menentukan prioritas simpul hidup yang diambil sebagai simpul ekspan. Ketiga algoritma memiliki definisi fungsi tersendiri, yaitu $g(n)$, $h(n)$, atau keduanya. Dalam konteks persoalan penyelesaian word ladder, $g(n)$ didefinisikan sebagai banyaknya perbedaan huruf dari kata yang diperiksa dengan kata awal, sedangkan $h(n)$ adalah banyaknya perbedaan huruf dari kata yang diperiksa dengan kata tujuan.

Heuristik yang admissible berarti estimasi yang diberikan pada setiap simpul tidak melebihi biaya sebenarnya untuk mencapai solusi. Pada kasus penyelesaian permainan word ladder, heuristik yang digunakan oleh algoritma A* tidak selalu admissible. Hal ini dikarenakan heuristik yang digunakan merupakan hamming distance antara kata yang diperiksa dengan kata tujuan. Perbedaan satu huruf tidak menandakan bahwa algoritma mendekat satu langkah menuju tujuan.

Dalam penyelesaian permainan word ladder, algoritma UCS dapat dikatakan sama dengan BFS karena biaya dari tiap simpul yang bertetangga adalah sama. Hal ini membuat pembobotan pada UCS tidak terlalu signifikan apabila setiap akan memeriksa simpul lain biayanya selalu sama.

Diketahui bahwa fungsi pada algoritma A* ikut mempertimbangkan heuristik dari setiap perhitungan, sehingga dalam menentukan simpul ekspan, kata yang lebih dekat dengan simpul tujuan akan lebih diprioritaskan. Berbeda dengan UCS, algoritma UCS hanya mempertimbangkan biaya dari simpul awal ke simpul yang diperiksa. Jadi, secara teoretis, algoritma A* lebih efisien dibandingkan dengan algoritma UCS

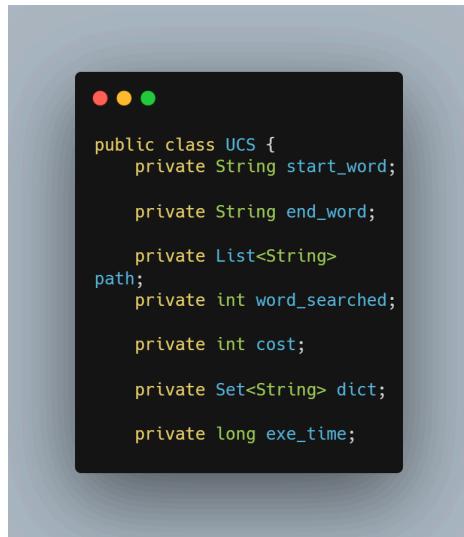
Algoritma Greedy Best First Search akan selalu berusaha mencari simpul dengan heuristik terdekat dengan simpul tujuan. Oleh karena itu, setiap langkah yang diambil selalu mendekatkan kepada simpul tujuan. Namun, algoritma ini akan mencarinya dengan buta tanpa melihat apakah akan ada jalurnya ketika simpul tersebut dikunjungi. Contoh kasus ini yaitu dari kata ‘flying’ ke ‘create’. Jalur yang paling mungkin diambil oleh algoritma adalah flying → frying → crying, lalu algoritma terjebak dalam simpul ‘crying’. Jadi, secara teoretis, apabila algoritma Greedy Best First Search berhasil menemukan solusinya, maka solusi tersebut optimal meskipun rentan terjebak dalam optimum lokal.

BAB III

IMPLEMENTASI PADA PROGRAM

A. Desain Program dalam Bahasa Java

Pada program yang kami buat, kami membuat kelas UCS, GBFS, dan AStar yang dikelompokkan ke dalam package algorithm. Setiap kelas tersebut memiliki atribut yang serupa, yaitu sebagai berikut



```
public class UCS {
    private String start_word;
    private String end_word;
    private List<String> path;
    private int word_searched;
    private int cost;
    private Set<String> dict;
    private long exe_time;
```

Gambar 3.1 Atribut Kelas UCS, GBFS, dan AStar

Sedangkan konstruktor dari masing-masing kelas juga serupa, yaitu sebagai berikut



```
public UCS(String start, String end, Set<String> dict) throws Exception {
    start_word = start.toUpperCase();
    end_word = end.toUpperCase();
    this.dict = dict;
    if (start_word.length() != end_word.length()) {
        Exception e = new WordLengthNotSameException();
        throw e;
    } else if (!(dict.contains(start_word.toUpperCase()) && dict.contains(end_word.toUpperCase()))) {
        Exception e = new WordNotFoundException();
        throw e;
    } else {
        path = search();
    }
    cost = getCost(end_word);
}
```

Gambar 3.2. Konstruktor kelas

Setiap kelas memiliki method search yang menjalankan algoritma pencarian (akan dijelaskan lebih lanjut pada subbab masing-masing). Jadi, ketika sebuah instance dari kelas tersebut dibuat, secara otomatis akan dijalankan algoritma pencarinya. Selain itu, terdapat beberapa method yang serupa pula sebagai berikut



```
// Membuat path dari start ke end dengan map dari pasangan simpul kata yang bertetangga
public List<String> makePath(String start, String end, Map<String, String> parentMap) {
    List<String> path = new ArrayList<>();
    String current = end;

    while (!current.equals(start)) {
        path.add(0, current);
        current = parentMap.get(current);

        if (current == null) {
            break;
        }
    }

    path.add(0, start);
    return path;
}
```

Gambar 3.3. Fungsi makePath untuk membentuk jalur



```
// Membuat list of simpul anak dari sebuah simpul kata
public List<String> getChildNode(String word) {
    List<String> child_node = new ArrayList<>();

    for (int i=0; i<word.length(); i++) {
        char[] chars = word.toCharArray();
        for (char c='A'; c<'Z'; c++) {
            chars[i] = c;
            String new_word = new String(chars).toUpperCase();
            if (dict.contains(new_word)) {
                child_node.add(new_word);
            }
        }
    }

    return child_node;
}
```

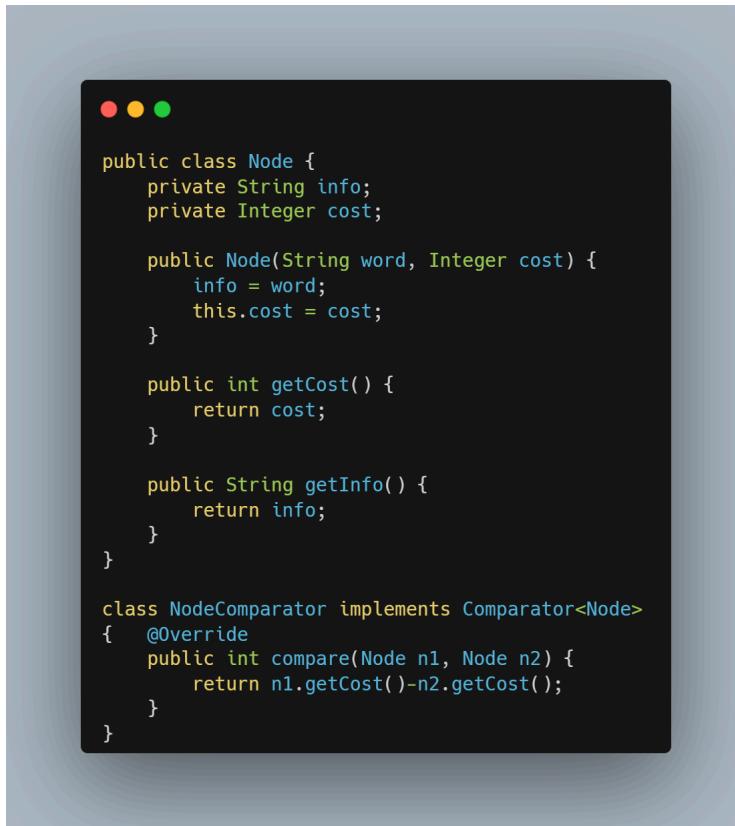
Gambar 3.4. Fungsi getChildNode untuk menghasilkan simpul anak dari suatu simpul kata



```
// Menampilkan hasil pencarian dengan GUI
public String getResult() {
    String res = "";
    if (path != null) {
        res += end_word + " found from " + word_searched + " words in " + exe_time + " ms.\n";
        res += "Path of search:\n";
        for (int i=0; i<path.size(); i++) {
            if (i != 0) {
                res += " -> ";
            }
            res += path.get(i);
        }
        res += "\n";
    } else {
        res += "No path found from " + start_word + " to " + end_word + "\n";
    }
    return res;
}
```

Gambar 3.5 Fungsi getResult untuk mendapatkan sebuah String untuk menampilkan hasil pada GUI

Supaya comparator pada priority queue dapat digunakan dengan baik, kami membuat sebuah kelas Node. Berikut adalah implementasinya



```
public class Node {
    private String info;
    private Integer cost;

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

    public int getCost() {
        return cost;
    }

    public String getInfo() {
        return info;
    }
}

class NodeComparator implements Comparator<Node>
{
    @Override
    public int compare(Node n1, Node n2) {
        return n1.getCost()-n2.getCost();
    }
}
```

Gambar 3.6 Kelas Node dan NodeComparator

Selain itu, kami membuat kelas FileHandler untuk membaca dan menyimpan kamus kata-kata yang valid. Berikut adalah implementasinya



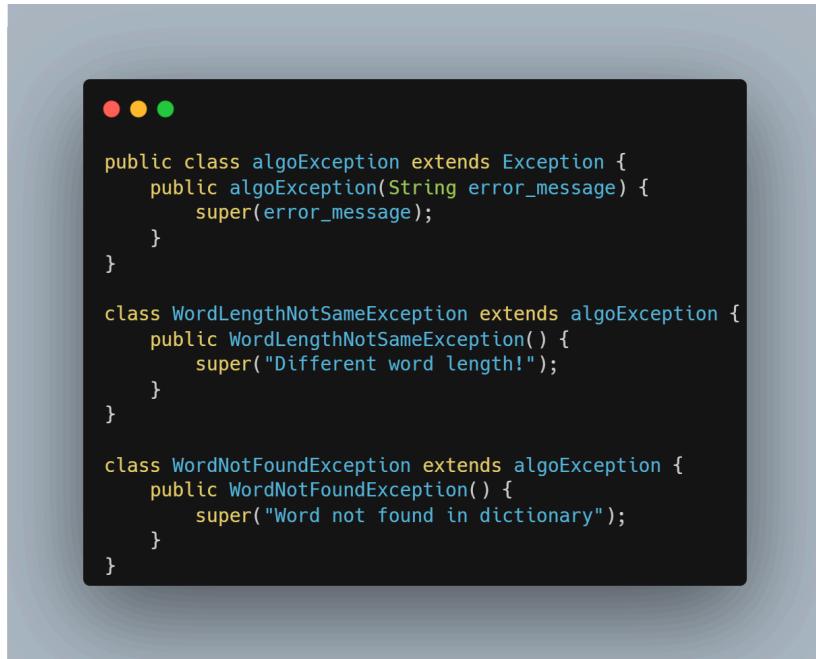
```
● ● ●

public class FileHandler {
    public Set<String> readDict(String file_path) throws IOException {
        Set<String> dict = new HashSet<>();
        try (BufferedReader br = new BufferedReader(new FileReader(file_path))) {
            String line;
            while ((line = br.readLine()) != null) {
                dict.add(line.trim());
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
        return dict;
    }

    // Trim dictionary to only store word with certain length
    public Set<String> trimDict(Set<String> dict, int length) {
        Set<String> new_dict = new HashSet<>();
        for (String s:dict) {
            if (s.length() == length) {
                new_dict.add(s);
            }
        }
        return new_dict;
    }
}
```

Gambar 3.7 Kelas FileHandler untuk menerima kamus kata

Untuk exception handling, kami mendefinisikan exception-exception untuk menghasilkan pesan kesalahan khusus yang bergantung pada kondisi. Berikut implementasinya



```
public class algoException extends Exception {
    public algoException(String error_message) {
        super(error_message);
    }
}

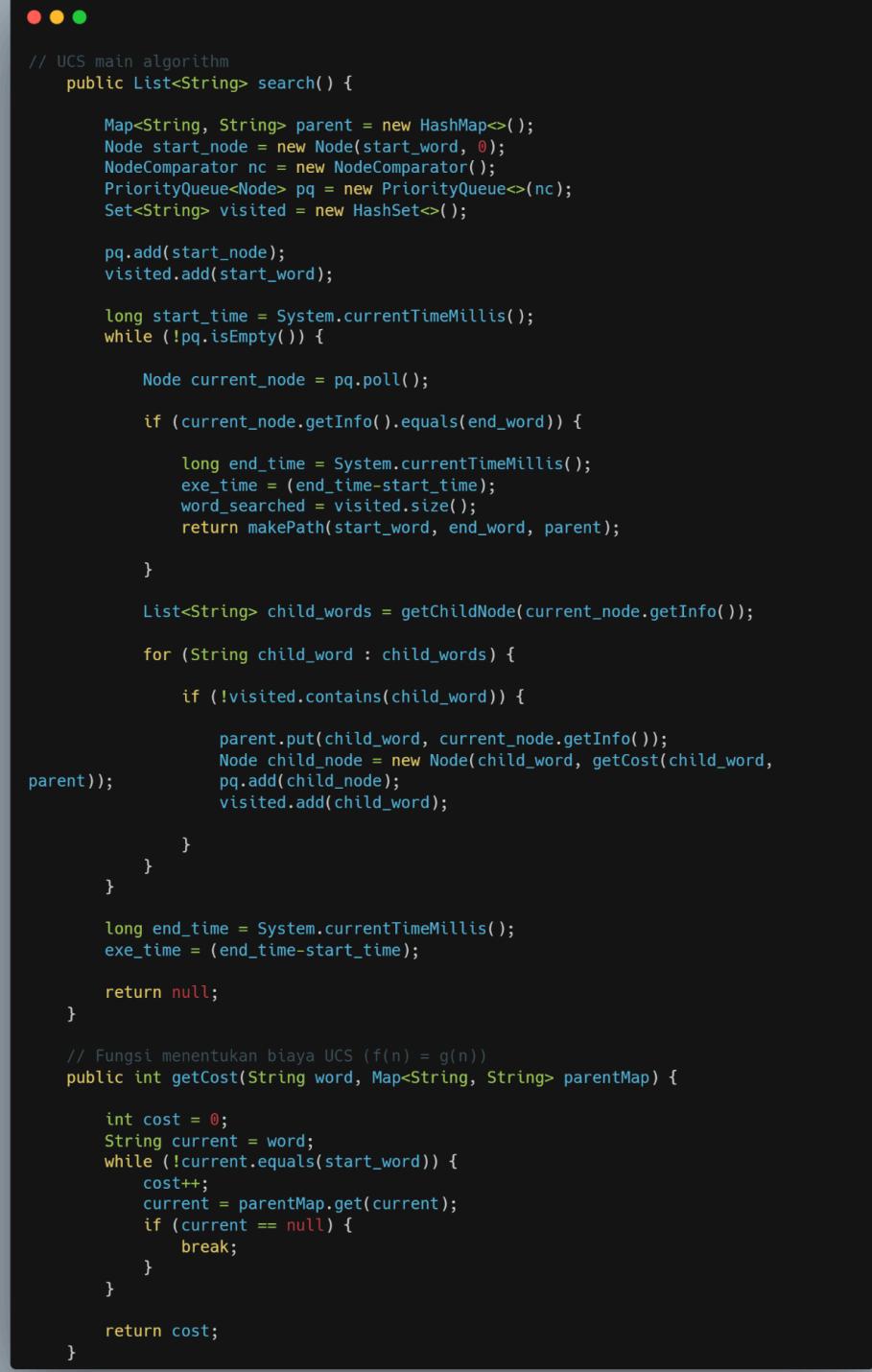
class WordLengthNotSameException extends algoException {
    public WordLengthNotSameException() {
        super("Different word length!");
    }
}

class WordNotFoundException extends algoException {
    public WordNotFoundException() {
        super("Word not found in dictionary");
    }
}
```

Gambar 3.8 Kelas Exception untuk kasus masukan yang tidak tepat

B. Implementasi Algoritma UCS

Berikut adalah implementasi algoritma UCS dalam pencarian rute.



```
// UCS main algorithm
public List<String> search() {

    Map<String, String> parent = new HashMap<>();
    Node start_node = new Node(start_word, 0);
    NodeComparator nc = new NodeComparator();
    PriorityQueue<Node> pq = new PriorityQueue<>(nc);
    Set<String> visited = new HashSet<>();

    pq.add(start_node);
    visited.add(start_word);

    long start_time = System.currentTimeMillis();
    while (!pq.isEmpty()) {

        Node current_node = pq.poll();

        if (current_node.getInfo().equals(end_word)) {

            long end_time = System.currentTimeMillis();
            exe_time = (end_time - start_time);
            word_searched = visited.size();
            return makePath(start_word, end_word, parent);
        }

        List<String> child_words = getChildNode(current_node.getInfo());
        for (String child_word : child_words) {

            if (!visited.contains(child_word)) {

                parent.put(child_word, current_node.getInfo());
                Node child_node = new Node(child_word, getCost(child_word,
                parent));
                pq.add(child_node);
                visited.add(child_word);

            }
        }
    }

    long end_time = System.currentTimeMillis();
    exe_time = (end_time - start_time);

    return null;
}

// Fungsi menentukan biaya UCS (f(n) = g(n))
public int getCost(String word, Map<String, String> parentMap) {

    int cost = 0;
    String current = word;
    while (!current.equals(start_word)) {
        cost++;
        current = parentMap.get(current);
        if (current == null) {
            break;
        }
    }

    return cost;
}
```

Gambar 3.9 Fungsi Utama algoritma UCS

C. Implementasi Algoritma Greedy Best First Search

Berikut adalah implementasi algoritma Greedy BFS dalam pencarian rute.



```
// Greedy Best First Search main algorithm
public List<String> search() {
    Map<String, String> parent = new HashMap<>();
    Node start_node = new Node(start_word, 0);
    NodeComparator nc = new NodeComparator();
    PriorityQueue<Node> pq = new PriorityQueue<>(nc);
    Set<String> visited = new HashSet<>();

    pq.add(start_node);
    visited.add(start_word);

    long start_time = System.currentTimeMillis();
    while (!pq.isEmpty()) {
        Node current_node = pq.poll();

        if (current_node.getInfo().equals(end_word)) {
            long end_time = System.currentTimeMillis();
            exe_time = (end_time - start_time);
            word_searched = visited.size();
            return makePath(start_word, end_word, parent);
        }

        List<String> child_words = getChildNode(current_node.getInfo());
        for (String child_word : child_words) {
            if (!visited.contains(child_word) && getHeuristic(child_word) <
                getHeuristic(current_node.getInfo())) {
                parent.put(child_word, current_node.getInfo());
                Node child_node = new Node(child_word, getHeuristic(child_word));
                pq.add(child_node);
                visited.add(child_word);
            }
        }
    }

    long end_time = System.currentTimeMillis();
    exe_time = (end_time - start_time);

    return null;
}

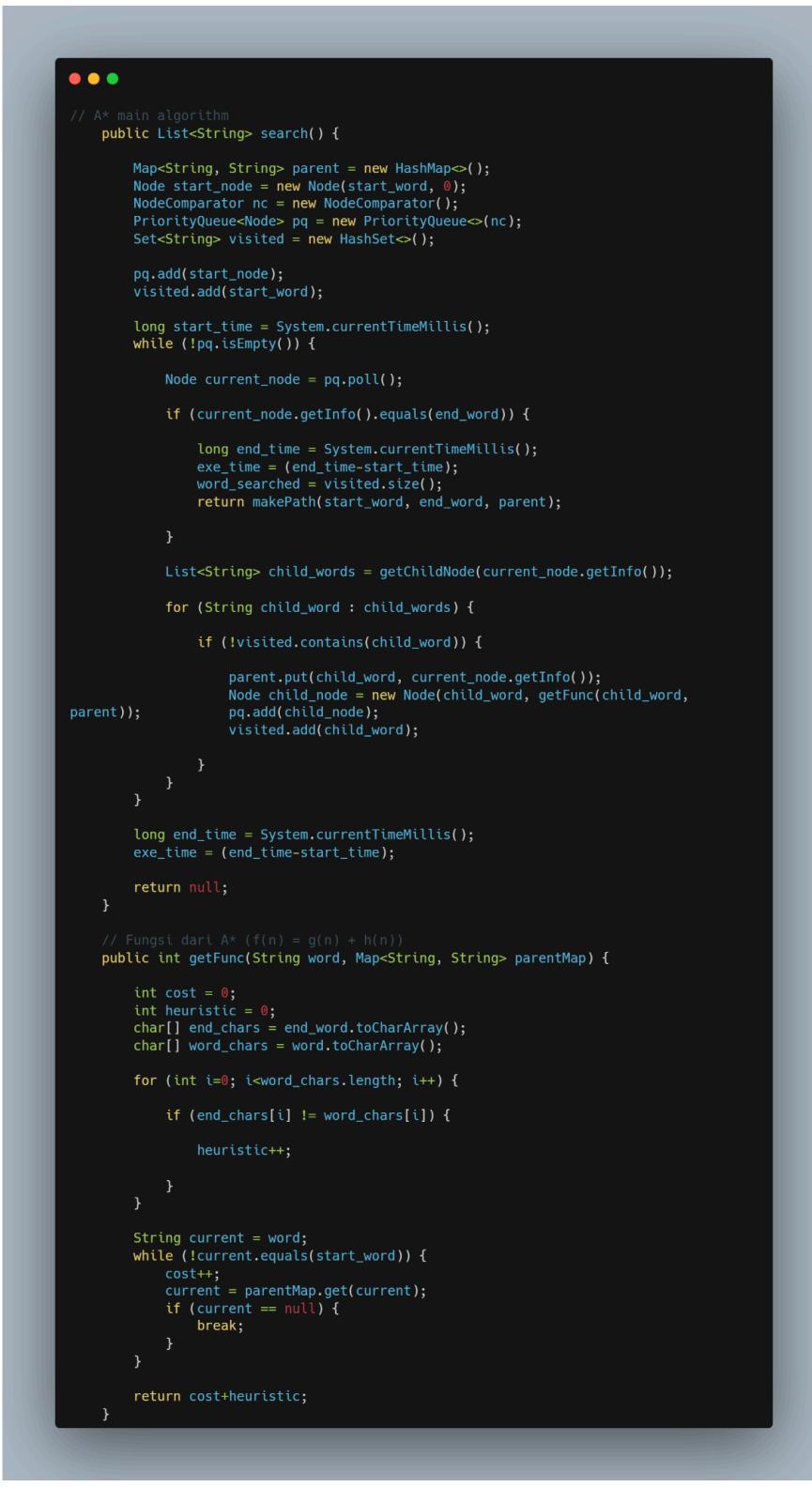
// Fungsi heuristik Greedy BFS (f(n) = h(n))
public int getHeuristic(String word) {
    int heuristic = 0;
    char[] end_chars = end_word.toCharArray();
    char[] word_chars = word.toCharArray();

    for (int i=0; i<word_chars.length; i++) {
        if (end_chars[i] != word_chars[i]) {
            heuristic++;
        }
    }
    return heuristic;
}
```

Gambar 3.10 Fungsi utama algoritma Greedy Best First Search

D. Implementasi Algoritma A*

Berikut adalah implementasi algoritma A* dalam pencarian rute



```
// A* main algorithm
public List<String> search() {
    Map<String, String> parent = new HashMap<>();
    Node start_node = new Node(start_word, 0);
    NodeComparator nc = new NodeComparator();
    PriorityQueue<Node> pq = new PriorityQueue<>(nc);
    Set<String> visited = new HashSet<>();

    pq.add(start_node);
    visited.add(start_word);

    long start_time = System.currentTimeMillis();
    while (!pq.isEmpty()) {

        Node current_node = pq.poll();

        if (current_node.getInfo().equals(end_word)) {

            long end_time = System.currentTimeMillis();
            exe_time = (end_time - start_time);
            word_searched = visited.size();
            return makePath(start_word, end_word, parent);
        }

        List<String> child_words = getChildNode(current_node.getInfo());
        for (String child_word : child_words) {
            if (!visited.contains(child_word)) {

                parent.put(child_word, current_node.getInfo());
                Node child_node = new Node(child_word, getFunc(child_word,
                    parent));
                pq.add(child_node);
                visited.add(child_word);

            }
        }
        long end_time = System.currentTimeMillis();
        exe_time = (end_time - start_time);
    }
    return null;
}

// Fungsi dari A* (f(n) = g(n) + h(n))
public int getFunc(String word, Map<String, String> parentMap) {
    int cost = 0;
    int heuristic = 0;
    char[] end_chars = end_word.toCharArray();
    char[] word_chars = word.toCharArray();

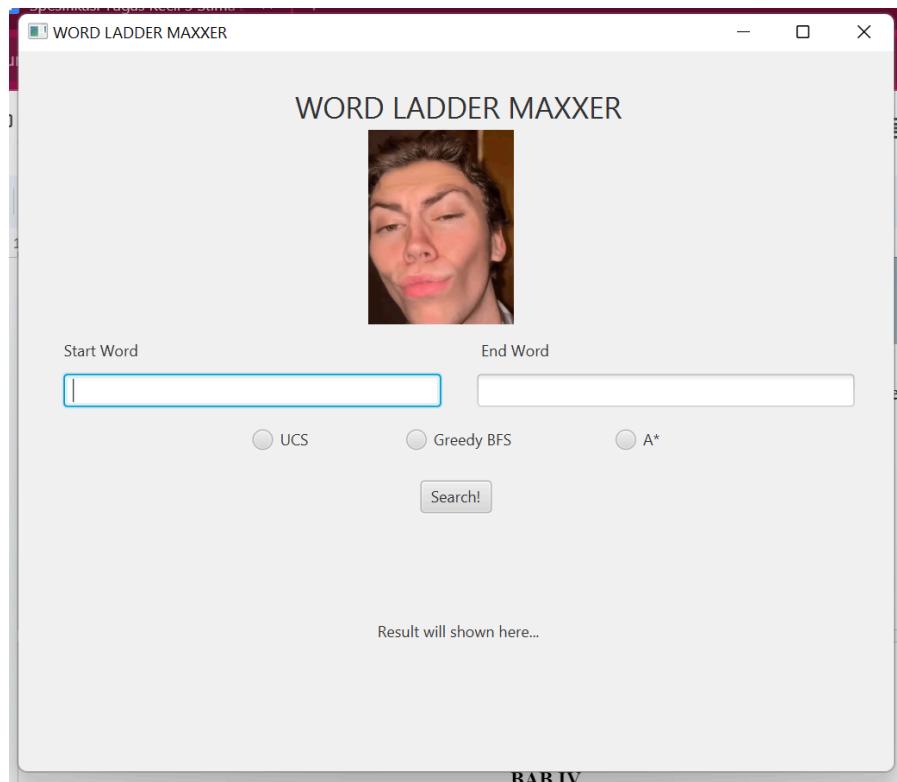
    for (int i=0; i<word_chars.length; i++) {
        if (end_chars[i] != word_chars[i]) {
            heuristic++;
        }
    }

    String current = word;
    while (!current.equals(start_word)) {
        cost++;
        current = parentMap.get(current);
        if (current == null) {
            break;
        }
    }
    return cost+heuristic;
}
```

Gambar 3.11 Fungsi utama algoritma A*

E. Implementasi Graphical User Interface (GUI) Program

Pada tugas kecil ini, kami mengerjakan bonus berupa implementasi GUI menggunakan JavaFX. Berikut adalah tampilan GUI yang kami buat.



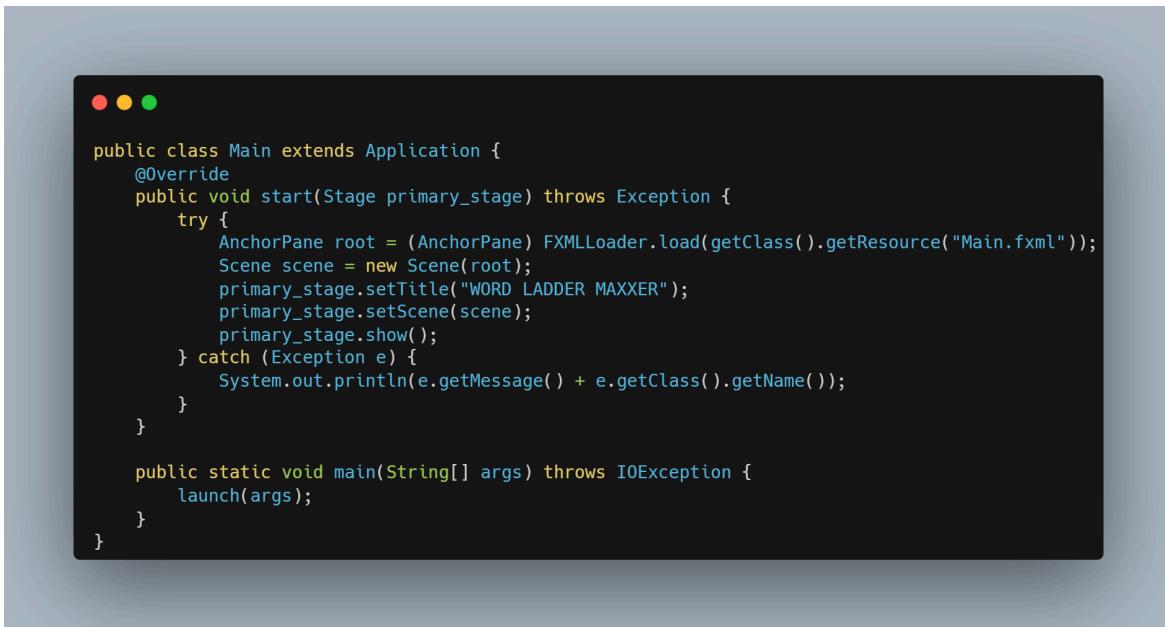
Gambar 3.12 Contoh ilustrasi GUI

Desain dari JavaFX dibangun menggunakan SceneBuilder yang menghasilkan file Main.fxml. Untuk mengontrol event-event yang terjadi di GUI, dibuat sebuah file MainSceneController.java yang berisi berikut

```
public class MainSceneController {  
  
    @FXML  
    private Label titleLabel;  
  
    @FXML  
    private ImageView imageView;  
    private Image image = new Image(getClass().getResourceAsStream("mewing-  
mew.png"));  
    @FXML  
    private Label resultLabel;  
  
    @FXML  
    private RadioButton rbUCS, rbGBFS, rbAStar;  
    private ToggleGroup algo;  
  
    @FXML  
    private Button searchButton;  
  
    @FXML  
    private TextArea taResult;  
  
    @FXML  
    private TextField tfEnd;  
  
    @FXML  
    private TextField tfStart;  
  
    @FXML  
    void displayImage() {  
        imageView.setImage(image);  
    }  
  
    @FXML  
    void setToggle() {  
        rbUCS.setToggleGroup(algo);  
        rbGBFS.setToggleGroup(algo);  
        rbAStar.setToggleGroup(algo);  
    }  
  
    @FXML  
    void search(ActionEvent event) throws Exception {  
        resultLabel.setWrapText(true);  
        try {  
            FileHandler fh = new FileHandler();  
            Set<String> dict = fh.readDict("src/dictionary/dictionary.txt");  
            Stage mainWindow = (Stage) titleLabel.getScene().getWindow();  
            String start_word = tfStart.getText();  
            String end_word = tfEnd.getText();  
            if (rbUCS.isSelected()) {  
                UCS algoUCS = new UCS(start_word, end_word, dict);  
                resultLabel.setText(algoUCS.getResult());  
            }  
            if (rbAStar.isSelected()) {  
                AStar algoAStar = new AStar(start_word, end_word, dict);  
                resultLabel.setText(algoAStar.getResult());  
            }  
            if (rbGBFS.isSelected()) {  
                GBFS algoGBFS = new GBFS(start_word, end_word, dict);  
                resultLabel.setText(algoGBFS.getResult());  
            }  
        } catch (Exception e) {  
            resultLabel.setText(e.getMessage());  
        }  
    }  
}
```

Gambar 3.13 Kelas MainClassController untuk handle event pada GUI

Dengan diimplementasikannya GUI pada program ini, maka berikut adalah fungsi main dari program



```
public class Main extends Application {
    @Override
    public void start(Stage primary_stage) throws Exception {
        try {
            AnchorPane root = (AnchorPane) FXMLLoader.load(getClass().getResource("Main.fxml"));
            Scene scene = new Scene(root);
            primary_stage.setTitle("WORD LADDER MAXXER");
            primary_stage.setScene(scene);
            primary_stage.show();
        } catch (Exception e) {
            System.out.println(e.getMessage() + e.getClass().getName());
        }
    }

    public static void main(String[] args) throws IOException {
        launch(args);
    }
}
```

Gambar 3.14 Program utama

BAB IV

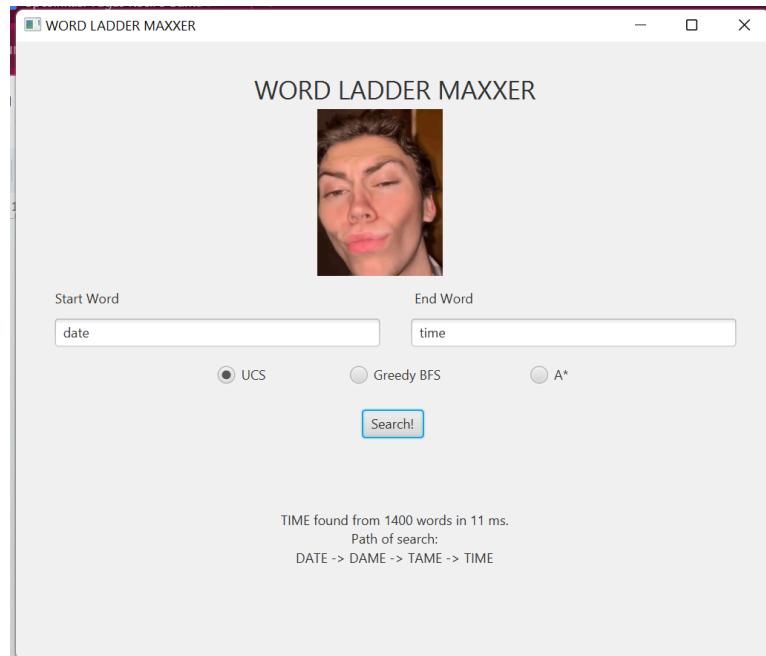
UJI COBA DAN ANALISIS

A. Uji Coba

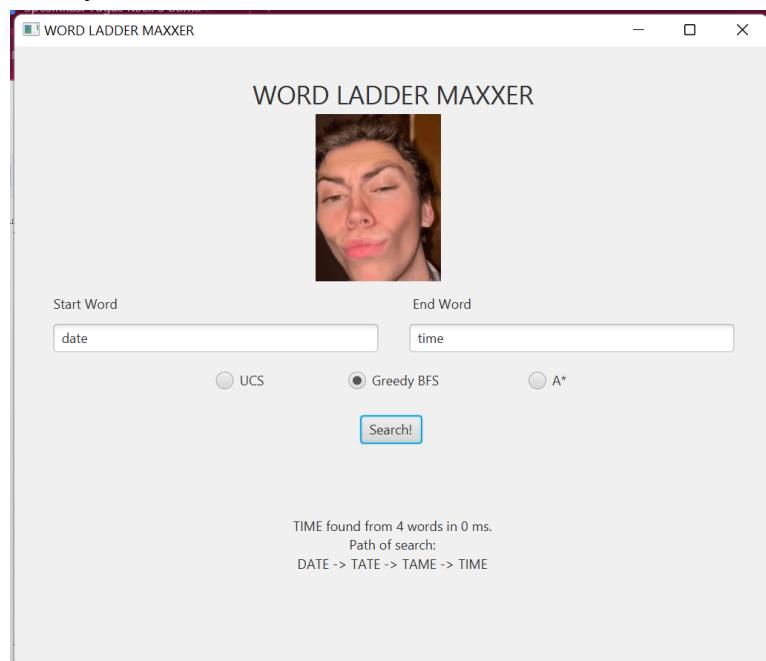
Uji coba akan dilakukan menggunakan test case yang dipakai oleh ketiga algoritma

1. Test Case 1 (date - time)

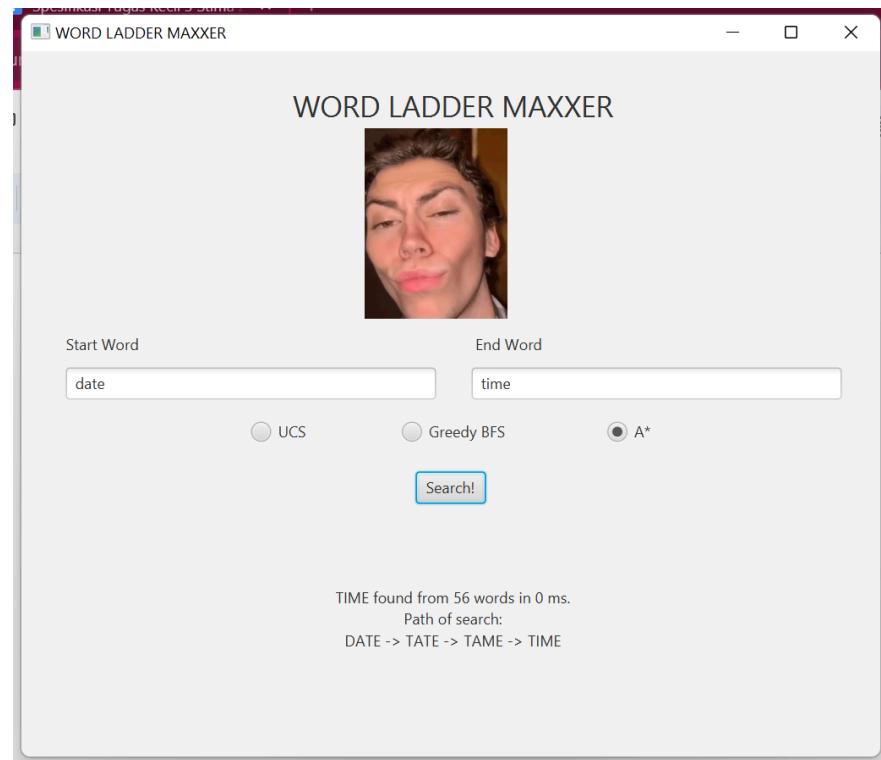
- UCS



- Greedy Best First Search

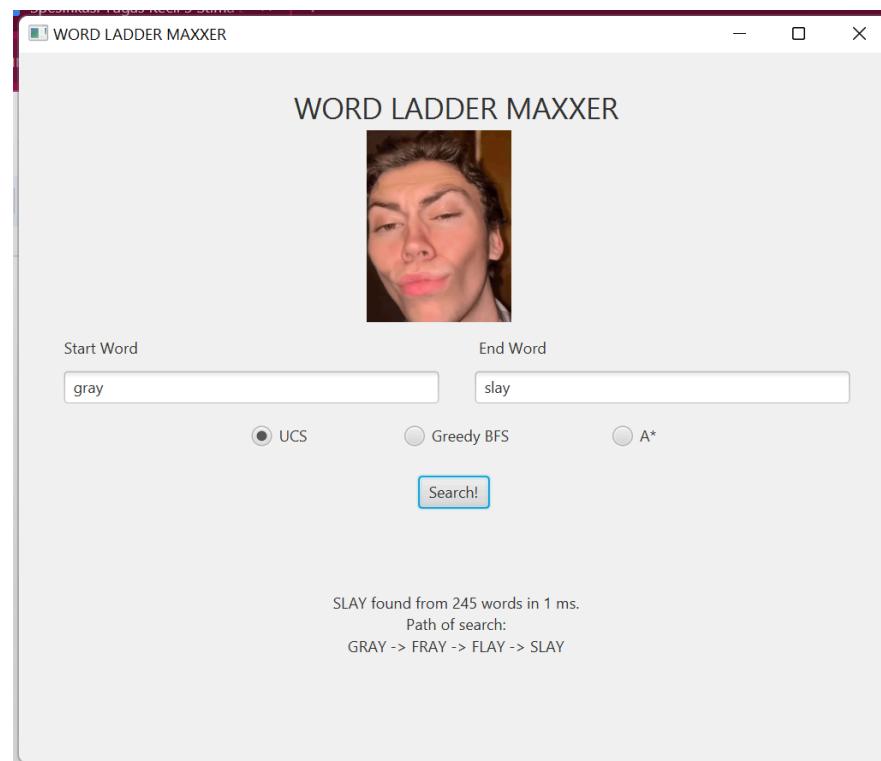


- A*

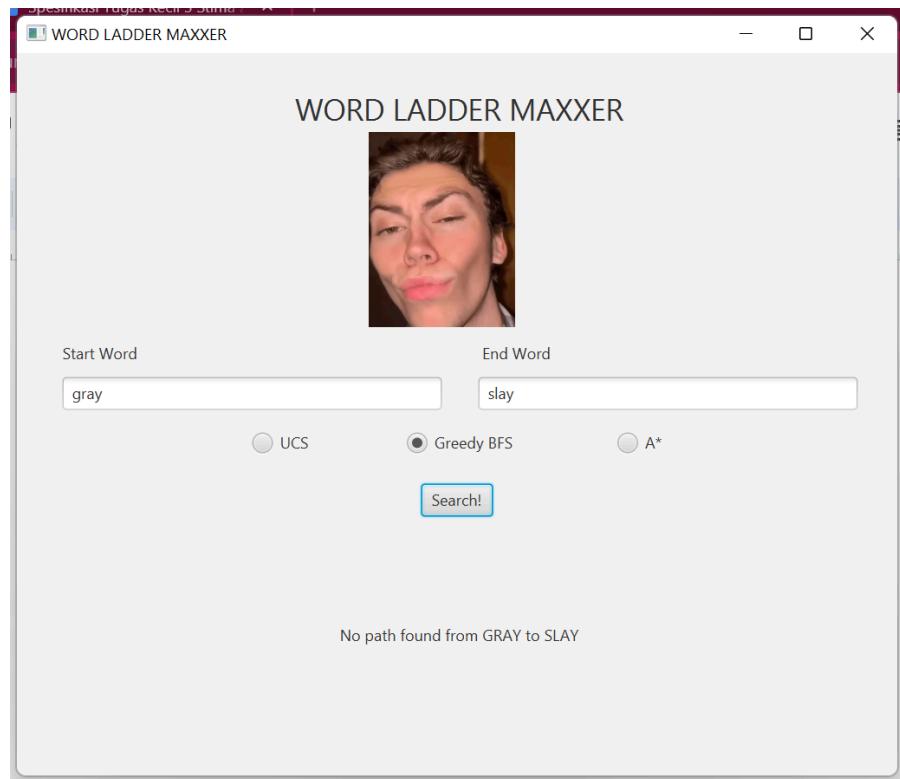


- 2. Test Case 2 (gray - slay)

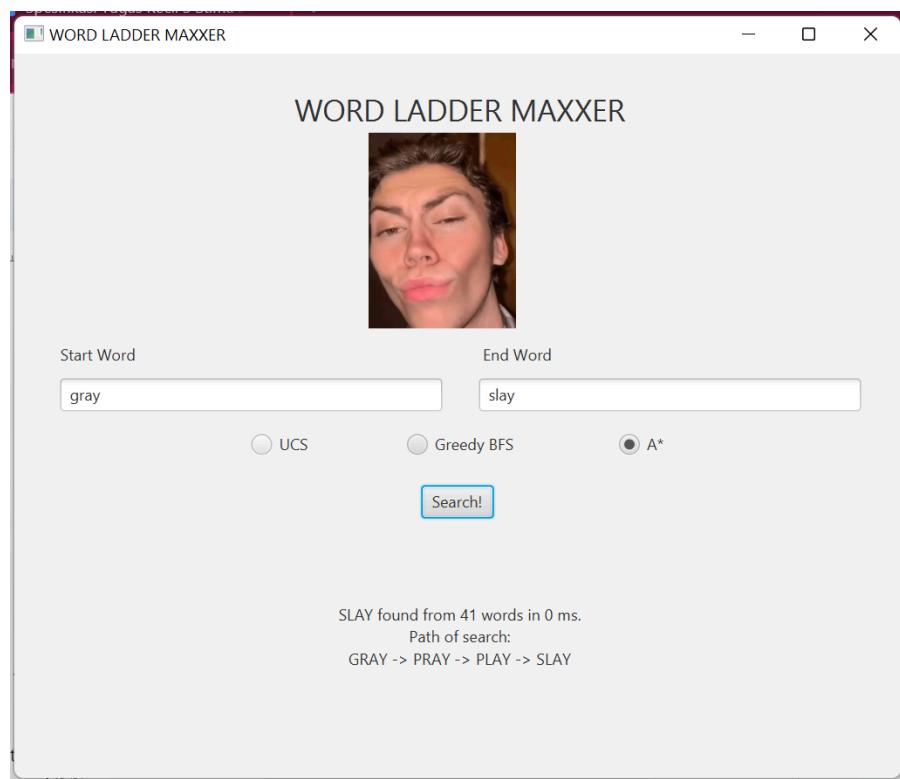
- UCS



- Greedy Best First Search

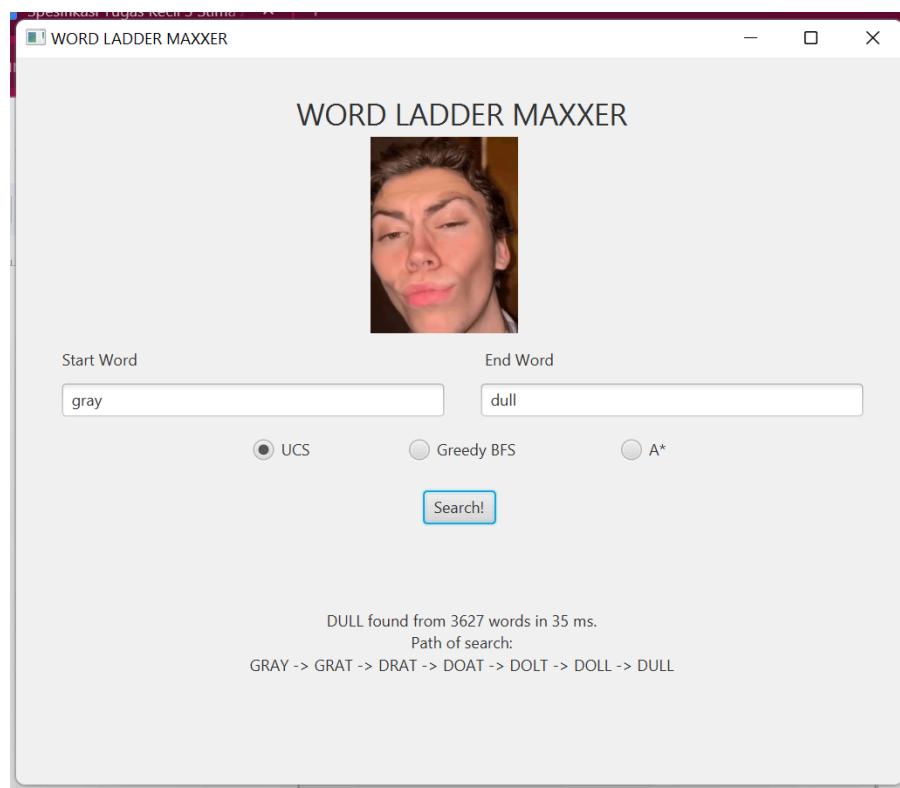


- A*

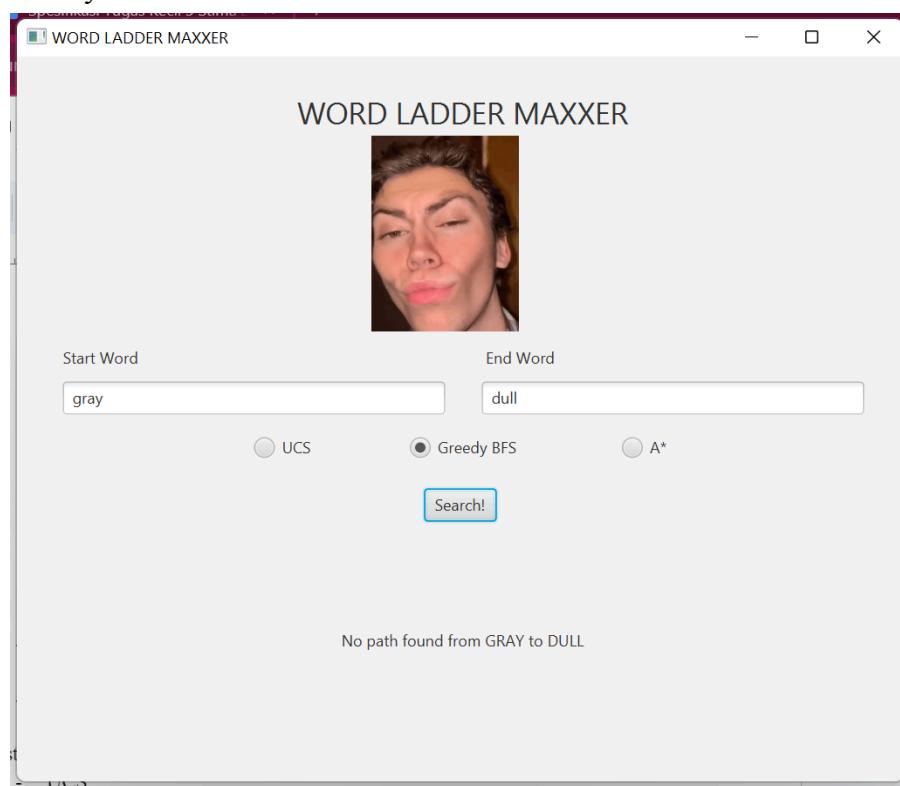


3. Test Case 3 (gray - dull)

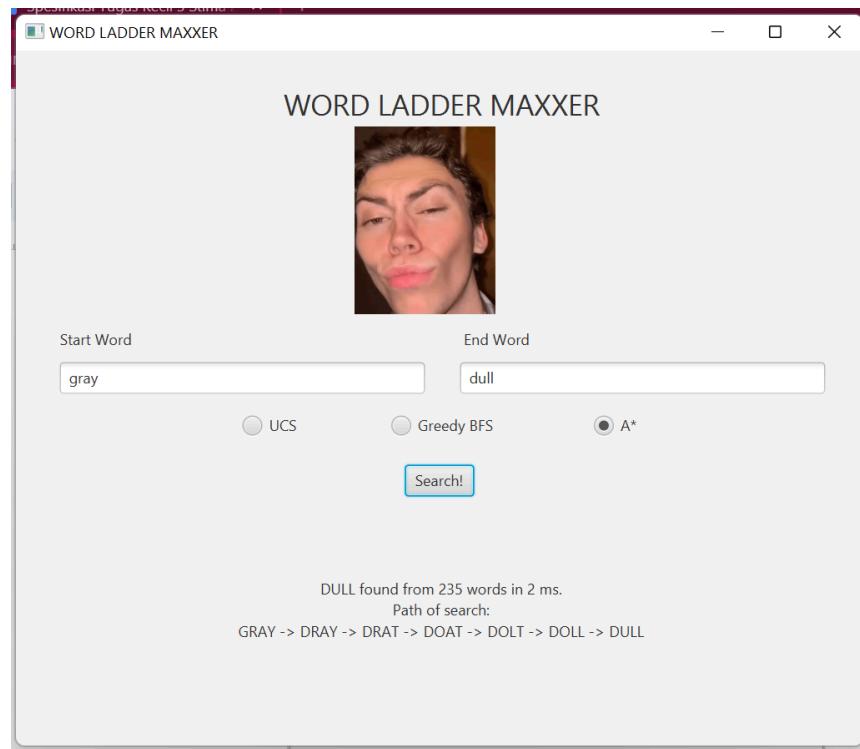
- UCS



- Greedy Best First Search

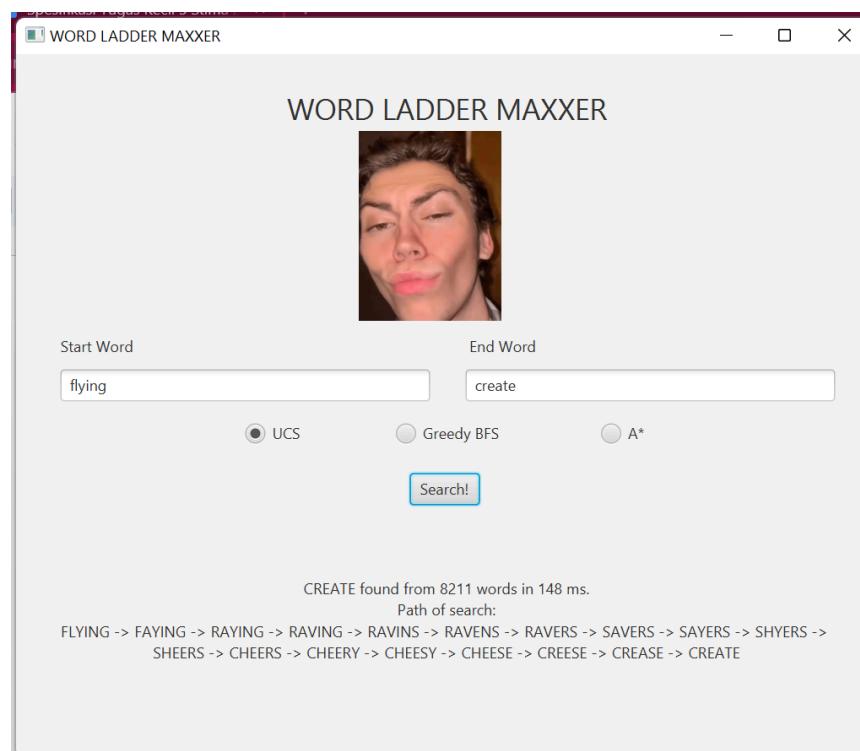


- A*

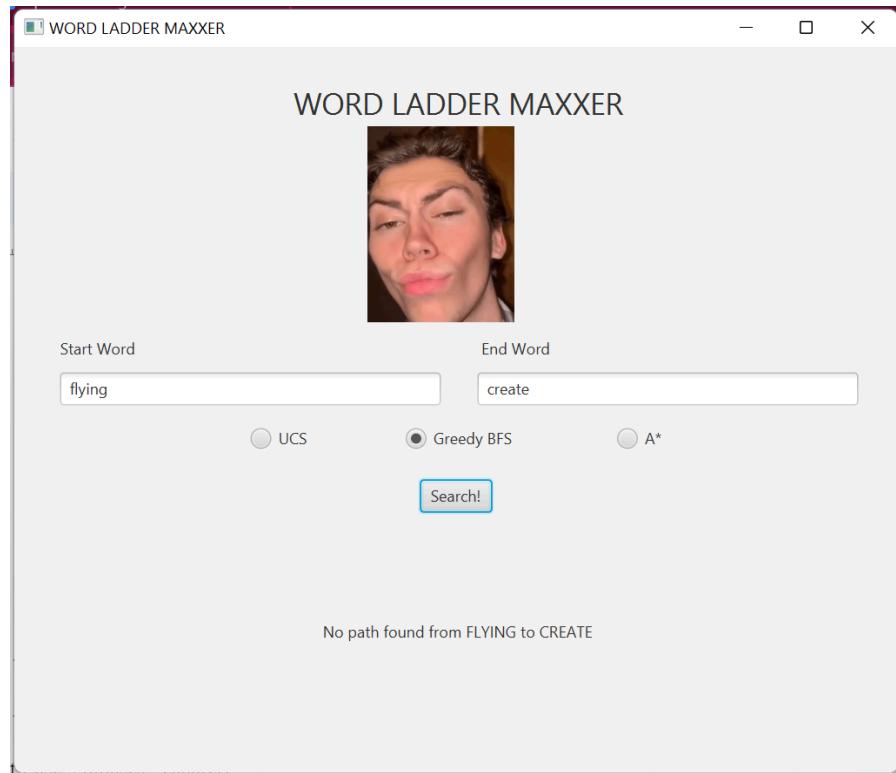


4. Test Case 4 (flying - create)

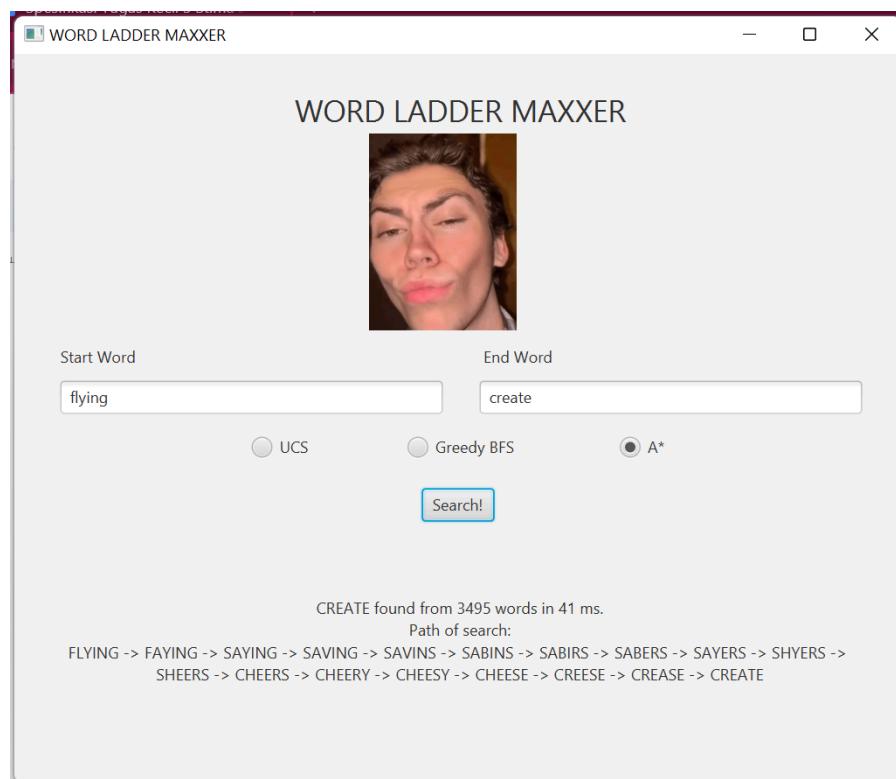
- UCS



- Greedy Best First Search

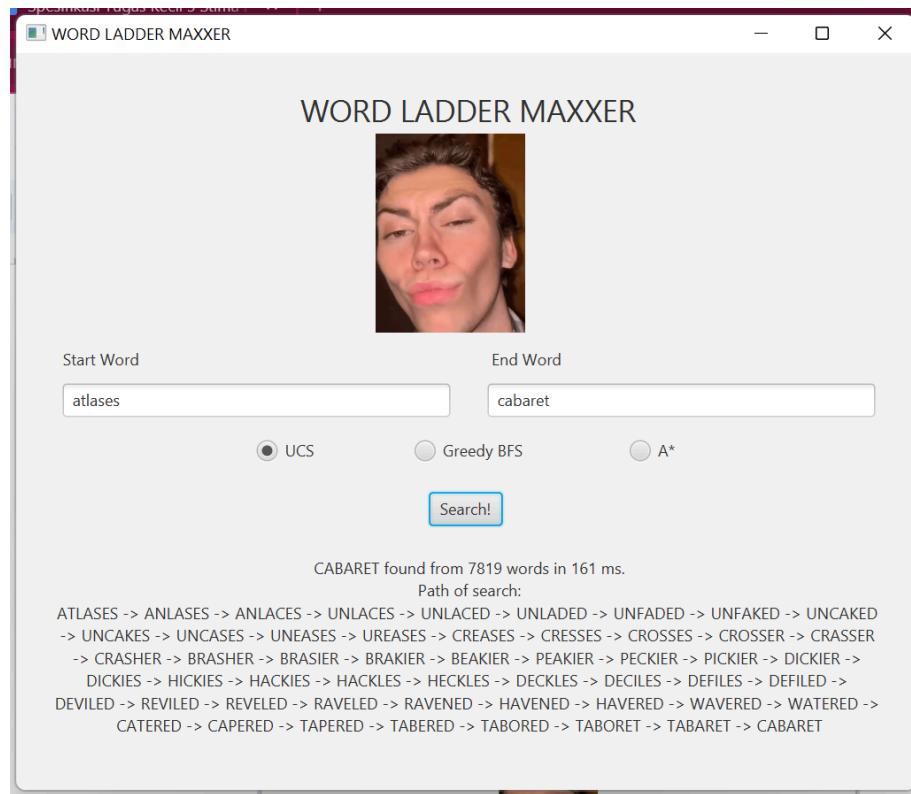


- A*

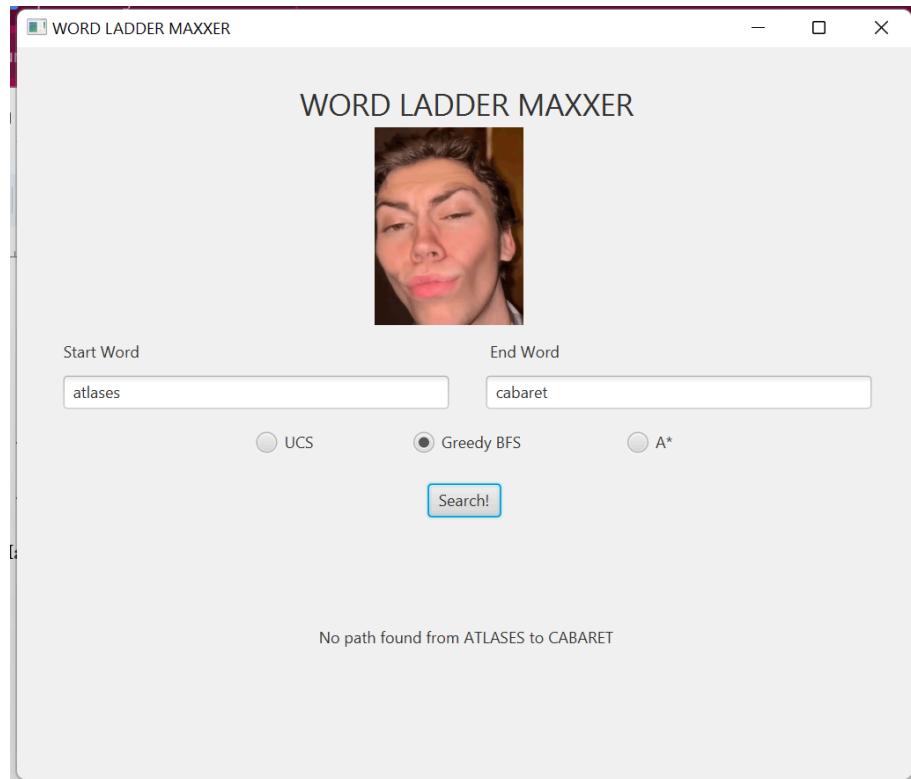


5. Test Case 5 (atlases - cabaret)

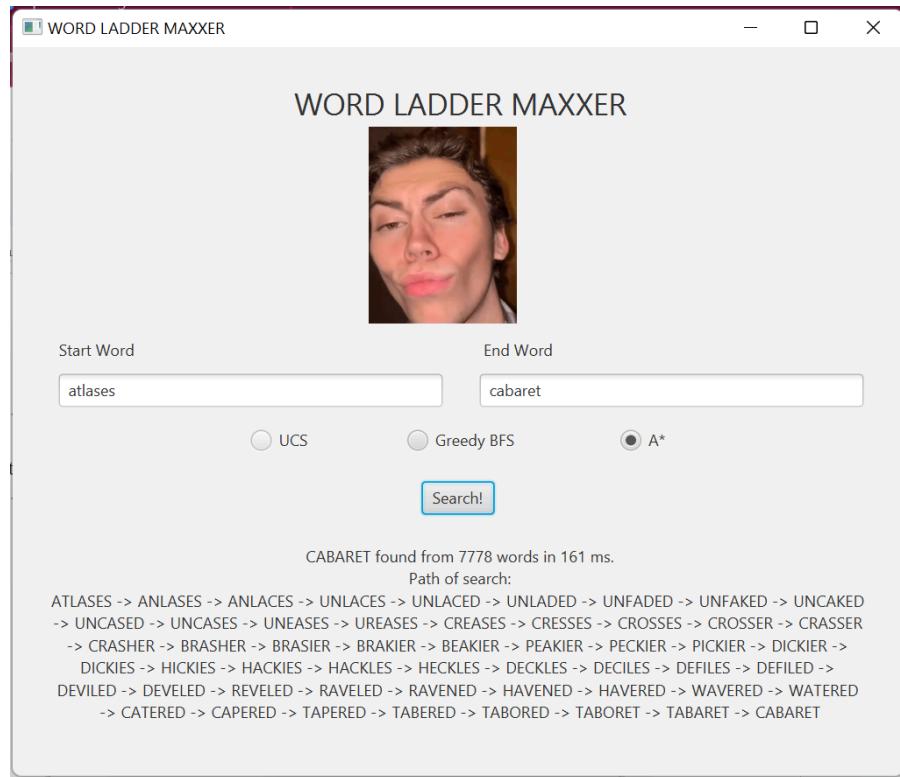
- UCS



- Greedy Best First Search

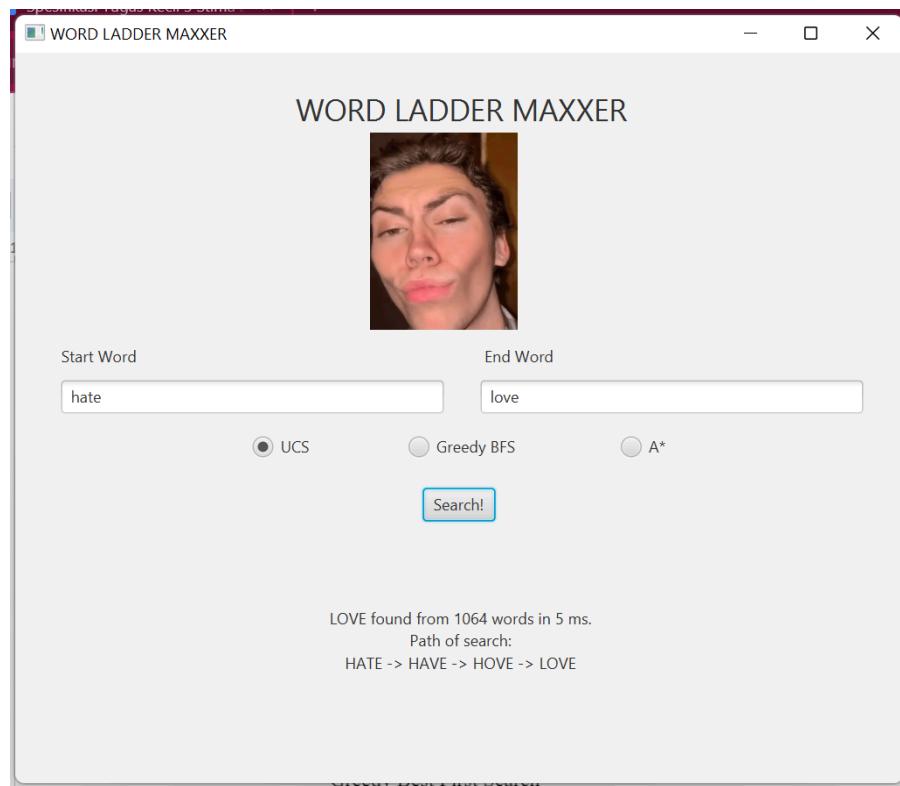


- A*

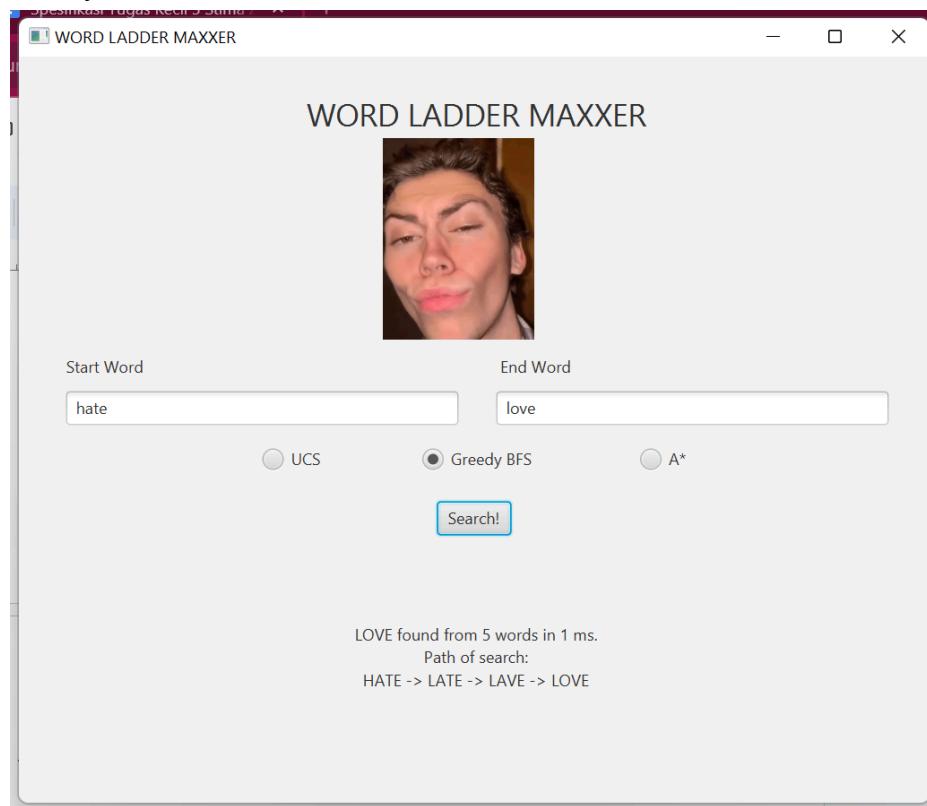


6. Test Case 6 (hate - love)

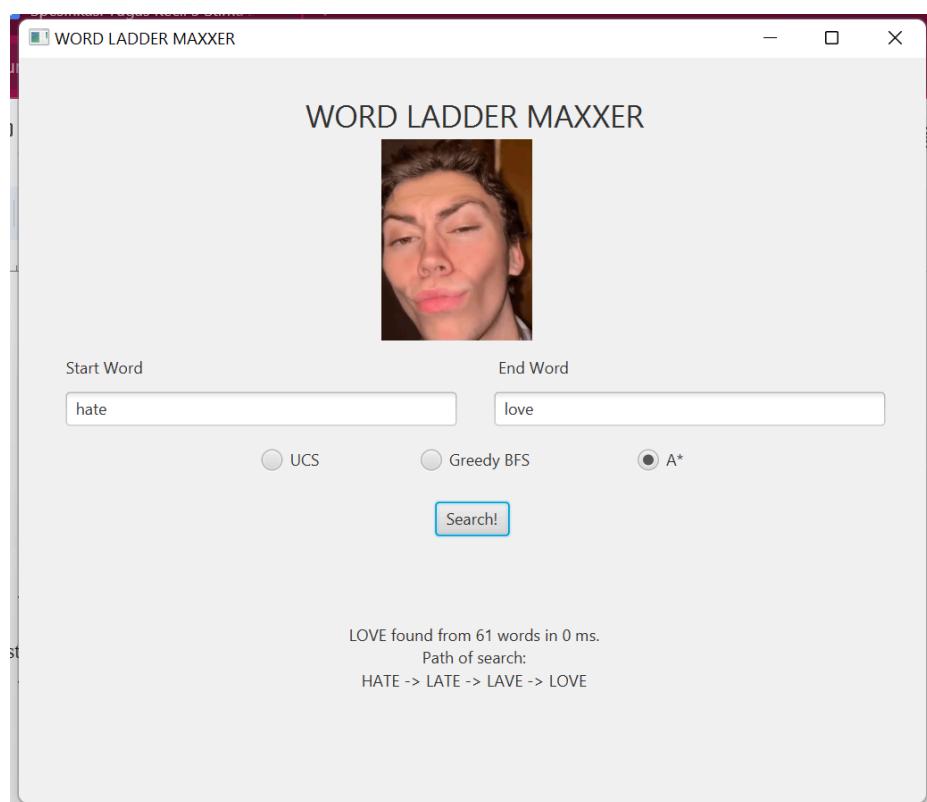
- UCS



- Greedy Best First Search

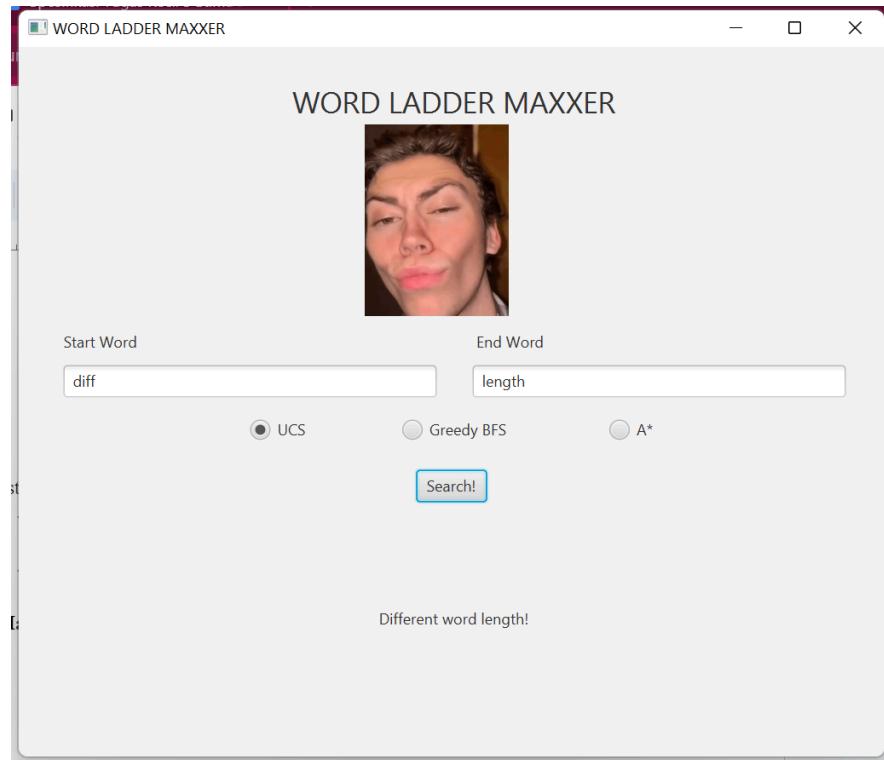


- A*

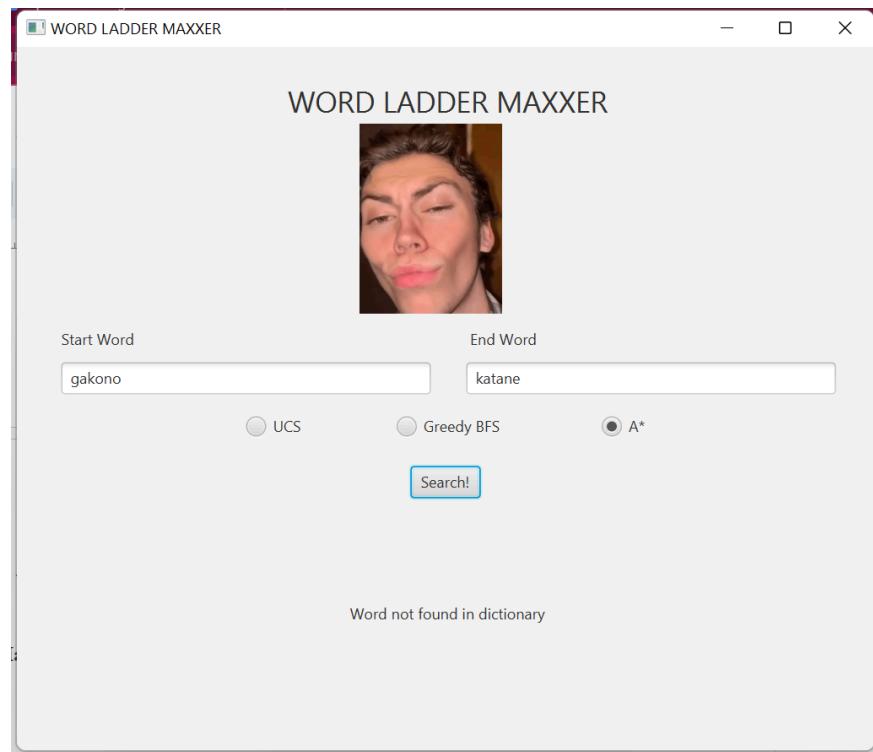


7. Test Case 7 (Masukan Tidak Valid)

- Panjang kata berbeda



- Kata tidak terdapat dalam kamus



B. Analisis Hasil Uji Coba

Pada program ini, setiap algoritma telah berhasil diimplementasikan dan dapat menemukan solusinya. Untuk kasus Greedy Best First Search yang tidak menemukan solusi pada test case 2, 3, 4, dan 5, kemungkinan besar algoritma terjebak dalam optimum lokal, sehingga ketika sudah mengunjungi satu simpul yang dekat dengan simpul tujuan, algoritma gagal untuk mencari jalur ke simpul tujuan karena tidak ada simpul tetangganya yang lebih dekat atau simpul tersebut tidak bertetangga dengan simpul tujuan.

Dari hasil uji coba, dapat dilihat bahwa hasil pada algoritma A* selalu lebih cepat dan pemeriksaan simpul dilakukan lebih sedikit, terutama pada test case dengan jalur yang pendek (kurang lebih 5). Hal ini sesuai dengan analisis secara teoretis bahwa algoritma A* lebih efisien dibandingkan dengan algoritma UCS. Namun, hasil tidak terlalu berarti apabila simpul tujuan sangat jauh dari simpul awal, contohnya pada test case 5 yang merupakan permainan terpanjang dalam persoalan word ladder (sumber: <http://datagenetics.com/blog/april32019/index.html>). Jadi, algoritma A* lebih efisien dibandingkan algoritma UCS

Dari hasil uji coba, dapat dilihat juga bahwa apabila algoritma Greedy Best First Search berhasil mendapatkan solusi, solusi yang didapatkan selalu optimal dengan simpul yang dikunjungi minimal dan waktu eksekusi yang cepat. Namun, seperti pada paragraf pertama, algoritma Greedy Best First Search sangat rentan untuk terjebak dalam optimum lokal yang menjadi kekurangan dari algoritma ini.

LAMPIRAN

- Checklist Kelengkapan Tugas

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

- Tautan Repository Github Implementasi Program :
https://github.com/ibrahim-rasyid/Tucil3_13522018