

**LAPORAN TUGAS KECIL 3**  
**IF2211 STRATEGI ALGORITMA**  
**Penyelesaian Permainan Word Ladder Menggunakan Algoritma**  
**UCS, Greedy Best First Search, dan A\***



**Disusun Oleh:**  
Diana Tri Handayani  
13522104

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

## DAFTAR ISI

|   |           |
|---|-----------|
| <b>DAFTAR ISI.....</b>  | <b>1</b>  |
| <b>DAFTAR GAMBAR.....</b>   | <b>3</b>  |
| <b>DAFTAR TABEL.....</b>  | <b>5</b>  |
| <b>BAB I</b>  |           |
| <b>DESKRIPSI TUGAS.....</b>   | <b>6</b>  |
| 1.1. Deskripsi Persoalan.....   | 6         |
| 1.2. Spesifikasi Tugas.....   | 8         |
| 1.2.1. Input.....   | 9         |
| 1.2.2. Output.....  | 9         |
| 1.2.3. Bonus.....   | 9         |
| <b>BAB II</b>   |           |
| <b>LANDASAN TEORI.....</b>  | <b>10</b> |
| 2.1. Algoritma Penentuan Rute ( <i>Route/Path Planning</i> ).....                 | 10        |
| 2.2. Algoritma <i>Uniform Cost Search</i> (UCS).....                              | 10        |
| 2.3. Algoritma <i>Greedy Best First Search</i> .....                              | 11        |
| 2.4. Algoritma A*.....  | 13        |
| <b>BAB III</b>  |           |
| <b>IMPLEMENTASI PROGRAM.....</b>  | <b>15</b> |
| 3.1. Permainan Word Ladder dengan Algoritma <i>Uniform Cost Search</i> (UCS)..... | 15        |
| 3.2. Permainan Word Ladder dengan Algoritma <i>Greedy Best First Search</i> ..... | 17        |
| 3.3. Permainan Word Ladder dengan Algoritma A*.....                               | 17        |
| 3.4. Sistematisasi File.....  | 18        |
| 3.5. Struktur Kelas dan Struktur Data.....  | 19        |
| 3.6. Source Code Program.....   | 20        |
| 3.5.1. Node.java.....   | 20        |
| 3.5.2. UCS.java.....  | 21        |
| 3.5.3. GBFS.java.....   | 22        |
| 3.5.4. Astar.java.....  | 23        |
| 3.5.5. Triple.java.....   | 25        |
| 3.5.6. WordLadderGUI.java.....  | 27        |
| <b>BAB IV</b>   |           |
| <b>IMPLEMENTASI DAN PENGUJIAN.....</b>  | <b>31</b> |
| 4.1. Spesifikasi Komputer Uji Coba.....   | 31        |
| 4.2. Uji Coba.....  | 31        |
| 4.2.1. Hasil Uji Coba Panjang <i>Path</i> .....                                   | 32        |

|  |           |
|--|-----------|
| 4.2.2. Hasil Uji Coba Jumlah Node Terkunjungi..... | 33        |
| 4.2.3. Hasil Uji Coba Waktu Eksekusi.....          | 35        |
| 4.3. Analisis Hasil Uji Coba.....                  | 48        |
| 4.4. Bonus.....                                    | 48        |
| <b>BAB V</b>                                       |           |
| <b>PENUTUP.....</b>                                | <b>52</b> |
| 5.1. Kesimpulan.....                               | 52        |
| <b>DAFTAR PUSTAKA.....</b>                         | <b>53</b> |
| <b>LAMPIRAN.....</b>                               | <b>54</b> |
| Lampiran 1 Link Repository.....                    | 54        |
| Lampiran 2 Tabel Checklist Poin.....               | 54        |

## BAB I

### DESKRIPSI MASALAH

#### 1.1. Deskripsi Persoalan

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

#### How To Play

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

##### Rules

Weave your way from the start word to the end word.

Each word you enter **can only change 1 letter** from the word above it.

##### Example



**Gambar 1.1.** Ilustrasi dan Peraturan Permainan Word Ladder

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

## 1.2. Spesifikasi Tugas

Program yang dibangun memanfaatkan algoritma Uniform Cost Search (UCS), Greedy Best First Search, dan A\*. Ketiga algoritma tersebut akan dibandingkan satu sama lain baik dari sisi tingkat optimal solusi, besar memori yang digunakan, serta besar waktu yang dibutuhkan untuk menemukan solusi. Program ditulis dalam bahasa pemrograman Java berbasis Command Line Interface (CLI) dan *Graphical User Interface* (GUI) sebagai bonus.

### 1.2.1. Input

Pengguna perlu memasukkan:

- **Start word** sebagai kata yang menjadi acuan titik awal pencarian. Kata harus dalam bahasa inggris dan panjang kata  $> 1$  huruf.
- **End word** sebagai kata yang menjadi tujuan titik akhir pencarian. Kata harus dalam bahasa inggris dan panjang kata sama dengan panjang *start word*..
- **Algoritma pencarian** yang ingin digunakan. Algoritma yang dipilih dapat lebih dari satu sekaligus.

### 1.2.2. Output

Program menampilkan:

- **Path** yang dihasilkan dari start word ke end word (cukup 1 path).
- **Banyak node** yang dikunjungi.
- **Waktu eksekusi** pencarian setiap algoritma yang dipilih.

### 1.2.3. Bonus

Berikut adalah spesifikasi bonus dari program yang dibangun.

- Membuat *Graphical User Interface* (GUI) dalam bahasa Java dengan kakas yang dibebaskan.

## BAB II

### LANDASAN TEORI

#### 2.1. Algoritma Penentuan Rute (*Route/Path Planning*)

Algoritma penentuan rute (*route/path planning*) adalah teknik untuk menemukan rute terpendek atau paling optimal antara dua titik atau lokasi dalam suatu jaringan. Jaringan yang dimaksud biasanya seperti jalur penerbangan, jalan raya, atau suatu jaringan distribusi. Algoritma ini diwujudkan dalam banyak jenis algoritma dengan berbagai perbedaan pengambilan keputusan setiap langkah. Beberapa faktor yang membedakan dalam pertimbangan keputusan antara lain kompleksitas waktu, kompleksitas memori, keoptimalan, ataupun kemampuan dalam menangani kondisi khusus.

Algoritma penentuan rute diklasifikasikan sebagai *uniformed search* (pencarian tanpa informasi) atau *informed search* (pencarian dengan informasi). Hal tersebut dilakukan berdasarkan algoritma menggunakan informasi tambahan mengenai sifat struktural dari ruang pencarian atau tidak. Algoritma *uniformed search* sering juga disebut *blind search* diantaranya adalah *Breadth-First Search* (BFS), *Depth-First Search* (DFS), *Depth Limited Search* (DLS), *Iterative Deepening Search* (IDS), dan *Uniform Cost Search* (UCS). Sedangkan algoritma *informed search* atau sering disebut *heuristic search* adalah *Greedy Best First Search* dan  $A^*$ .

Dalam memilih algoritma penentuan rute, sangatlah penting untuk mempertimbangkan antara efisiensi, kompleksitas, dan keakuratan solusi yang diinginkan. Algoritma *uniformed search* dapat digunakan ketika tidak ada informasi tambahan yang tersedia tentang struktur masalah, sedangkan *informed search* seringkali digunakan ketika terdapat informasi tambahan dan diperlukan untuk meningkatkan efisiensi pencarian.

#### 2.2. Algoritma *Uniform Cost Search* (UCS)

Algoritma *Uniform Cost Search* (UCS) adalah algoritma pencarian graf yang digunakan untuk menemukan jalur dengan biaya terendah dari simpul awal ke simpul tujuan dari graf berbobot. Graf berbobot adalah graf yang memiliki nilai numerik pada setiap sisinya

untuk menunjukkan biaya atau jarak yang terkait dengan perjalanan dari satu simpul ke simpul lainnya.

Algoritma UCS sering digunakan dalam permasalahan pencarian rute dengan biaya minimum sebagai tujuan utama. Sedangkan, algoritma BFS dan IDS bertujuan untuk menemukan rute terpendek, tanpa memperhatikan *cost* atau biaya yang ditempuh dalam rute. Sehingga ketika jumlah langkah tidak sama dengan mewakili jumlah *cost* atau biaya perjalanan, algoritma BFS dan IDS menjadi tidak lagi relevan untuk mencari solusi optimal dengan biaya minimum. Oleh karena itu, algoritma UCS melakukan pertimbangan biaya setiap langkah yang dibuatnya untuk mendapatkan solusi optimal dengan biaya minimum. Biaya setiap langkah biasanya diwakili oleh fungsi  $g(n)$  dengan  $g(n)$  adalah biaya perjalanan dari titik awal hingga titik ke- $n$ . Meskipun algoritma UCS dapat menemukan jalur terpendek, efisiensi dan kompleksitasnya dapat bergantung pada struktur dan ukuran graf yang diproses.

### 2.3. Algoritma *Greedy Best First Search*

Algoritma *Greedy Best First Search* adalah algoritma pencarian graf yang digunakan untuk menemukan jalur terpendek atau jalur dengan biaya terendah dari simpul awal ke simpul tujuan dalam graf berbobot. Algoritma *Greedy Best First Search* menggunakan heuristik untuk mengevaluasi simpul-simpul dalam ruang pencarian. Heuristik adalah fungsi atau aturan yang memberikan perkiraan biaya atau jarak tersisa dari simpul saat ini ke simpul tujuan.

Algoritma ini menggunakan fungsi heuristik  $h(n)$  sebagai fungsi evaluasi  $f(n)$  untuk menentukan prioritas simpul yang akan dieksplorasi selanjutnya. Algoritma *Greedy Best First Search* akan memilih simpul dengan perkiraan biaya terendah sebagai langkah selanjutnya. Akan tetapi, keputusan itu tidak selalu membuat hasil paling baik dalam jangka panjang. Hal itu terjadi karena algoritma ini hanya mempertimbangkan heuristik lokal tanpa memperhatikan konsekuensi jangka panjang dari langkah tersebut.

Algoritma *Greedy Best First Search* sering digunakan dalam permasalahan yang heuristik tersedia secara efektif dan pencarian solusi yang cepat lebih diutamakan daripada menjamin solusi optimal. Sehingga dapat disimpulkan *Greedy Best First Search* meskipun

cenderung lebih cepat daripada algoritma lain tetapi tidak dapat menjamin solusi yang dihasilkan optimal jika heuristiknya tidak akurat.

#### 2.4. Algoritma A\*

Algoritma A\*, sama seperti algoritma UCS dan *Greedy Best First Search* digunakan untuk menemukan jalur terpendek atau jalur dengan biaya terendah dari simpul awal hingga ke simpul tujuan dalam graf berbobot. A\* termasuk dalam algoritma *informed search*, artinya menggunakan informasi tambahan mengenai sifat struktural dari ruang pencarian untuk mengarahkan pencarian tujuan.

Algoritma A\* menggunakan fungsi evaluasi  $F(n)$  untuk mengevaluasi setiap simpul  $n$  dalam ruang pencarian. Fungsi evaluasi mengandung dua komponen, yaitu:

- $g(n)$  = biaya aktual dari simpul awal hingga simpul  $n$  yang terakumulasi dari setiap langkah yang ditempuhnya.
- $h(n)$  = heuristik yang memberikan perkiraan biaya atau jarak tersisa dari simpul  $n$  hingga simpul tujuan.

Sehingga fungsi evaluasi  $F(n)$  dihitung dengan menjumlahkan  $g(n)$  dengan  $h(n)$  atau  $F(n) = g(n) + h(n)$ . Kemudian, algoritma A\* akan memilih simpul dengan nilai  $F(n)$  terendah sebagai langkah selanjutnya atau memprioritaskan simpul-simpul yang memiliki perkiraan biaya total (biaya aktual + heuristik) paling rendah.

Algoritma ini menjamin menemukan solusi optimal jika heuristiknya *admissible* yaitu tidak melebihi-lebihkan biaya sebenarnya ke tujuan dan/atau  $h(n) \leq h^*(n)$  dengan  $h^*(n)$  adalah biaya sesungguhnya untuk mencapai tujuan dari titik  $n$ . Namun, untuk kompleksitasnya dapat bervariasi tergantung pada struktur graf dan kualitas heuristik.



### BAB III

#### IMPLEMENTASI PROGRAM

##### 3.1. Permainan *Word Ladder* dengan Algoritma *Uniform Cost Search* (UCS)

Salah satu cara untuk menyelesaikan permainan word ladder adalah dengan menggunakan algoritma *Uniform Cost Search* (UCS). Pada algoritma ini, UCS dibuat sebagai sebuah kelas turunan Node. Node adalah kelas buatan yang memiliki atribut word dan parent yang merepresentasikan simpul dalam sebuah struktur data *tree*. Node memiliki *method getter* setiap atributnya dan *method* *getPath()* untuk mendapatkan jalur dari suatu simpul hingga ke simpul akarnya. *Method* *getPath()*, nantinya akan digunakan untuk menelusuri hasil pencarian dari kata awal hingga kata tujuan.

Sesuai dengan prinsip algoritma UCS yang memiliki fungsi  $f(n)$  atau  $g(n)$  yaitu biaya dari titik awal ke suatu titik- $n$ , kelas UCS juga memiliki atribut *cost* untuk menyimpan data biaya dari node awal ke node tersebut. Dalam kasus ini *cost* didapatkan dengan menjumlahkan *cost* dari *parent*-nya dengan satu. Di dalam kelas UCS terdapat *method* *findUCS()* yang akan menjalankan algoritma UCS secara garis besar. *Method* *findUCS()* memiliki parameter masukan berupa *endWord* atau kata tujuan berupa String dan dictionary berupa *Set<String>* yang menyimpan kata-kata dalam kamus, serta akan mengembalikan sebuah kelas Triple yaitu kelas buatan yang menyimpan 3 buah variabel sekaligus.

Berikut setiap langkah yang dilakukan algoritma UCS di dalam *method* *findUCS()*:

1. Diawali dengan inialisasi *PriorityQueue* (*queue*) untuk menyimpan node-node dalam antrian diurutkan secara menaik berdasarkan besar atribut *cost*. Inialisasi sebuah *HashSet* (*visited*) untuk menyimpan node yang pernah dikunjungi agar tidak terjadi *looping* dan terbentuk *tree*. Inialisasi juga dilakukan untuk variabel *iterations* digunakan dalam penghitungan iterasi atau jumlah node yang dikunjungi.
2. Memasukkan node saat ini ke dalam *queue* dan *visited*.
3. Melakukan *looping* hingga *queue* kosong atau node tujuan ditemukan. Di dalam *looping* dilakukan penambahan *iterations* setiap iterasinya. Lalu akan diambil node paling depan dari *queue* untuk dicek apakah node tersebut merupakan node tujuan. Jika merupakan node tujuan, *method* akan langsung *me-return* sebuah Triple yang berisikan jalur dari

node awal hingga node tujuan menggunakan *method* *getPath()*, jumlah simpul hidup atau panjang set *visited*, dan jumlah node dikunjungi atau banyak iterasi.

4. Apabila node yang diambil dari *queue* bukan node tujuan, akan dilanjutkan dengan pencarian *child* atau node yang memungkinkan menjadi *path* selanjutnya dari node tersebut menggunakan *method* *getChildren()*. *Method* *getChildren()* ini akan menerima parameter masukan berupa *Set<String> dictionary* dan *Set<String> visited*. Pada *method* ini akan dilakukan pengecekan pada pergantian setiap huruf dari sebuah kata yang tersimpan dalam atribut *word*. Apabila kata yang diganti hurufnya tersebut berada pada *dictionary* dan belum tersimpan pada set *visited*, maka akan dimasukkan pada sebuah List. Setelah semua kemungkinan pergantian satu huruf telah dicoba, List tersebutlah yang menjadi output dari *method* *getChildren()*.
5. List yang merupakan output dari *method* *getChildren()* tadi selanjutnya akan dimasukkan dalam *queue* untuk diproses lagi dalam *looping* nantinya.
6. Apabila *queue* hingga kosong namu node tidak ditemukan, *method* akan mengembalikan sebuah nilai Triple tetapi pada parameter yang seharusnya berisikan jalur hasil pencarian akan diisi oleh null.

Hal yang perlu diperhatikan dalam algoritma UCS adalah mengenai nilai  $g(n)$  atau biaya dari node awal ke suatu node. Dalam kasus Word Ladder, tidak ada variabel biaya yang ditentukan secara eksplisit dari satu kata ke kata yang lain yang perbedaannya hanya satu huruf. Sehingga, dapat diasumsikan variabel biaya untuk menuju ke sebuah kata dari kata yang lain dengan perbedaan hanya satu huruf adalah konstan yaitu satu. Dengan begitu, variabel biaya di sini sama dengan jumlah langkah yang diambil dalam pencarian. Seperti yang telah dibahas dalam dasar teori, ketika biaya sama dengan jumlah langkah, maka permasalahan yang diselesaikan dengan algoritma UCS ini sebenarnya sama dengan diselesaikan dengan algoritma BFS. Kesamaan di sini dalam artian untuk urutan pembangkitan node atau simpul hidup dan simpul ekspansi, serta *path* yang dihasilkannya sama. Karena pada hakikatnya UCS diciptakan karena BFS hanya dapat mengoptimalkan jumlah langkah tanpa memperhatikan jumlah biaya, sedangkan permasalahan ini jika jumlah langkah optimal maka jumlah biaya juga akan optimal.

### 3.2. Permainan *Word Ladder* dengan Algoritma *Greedy Best First Search*

Pencarian solusi permainan Word Ladder juga dapat dilakukan dengan menggunakan algoritma *Greedy Best First Search (GBFS)*. Algoritma GBFS juga diterapkan dalam sebuah kelas turunan Node (kelas yang sama dan telah dijelaskan pada 3.1). Sesuai dengan prinsip algoritma GBFS yang memiliki fungsi heuristik ( $h(n)$ ), kelas GBFS juga memiliki atribut *heuristic* untuk merepresentasikan perkiraan biaya dari titik- $n$  ke titik tujuan. Dalam kasus ini, fungsi heuristik didapatkan dengan menghitung jumlah perbedaan huruf dengan kata tujuan. Fungsi heuristik tersebut diimplementasikan dalam *method* `getHeuristic()`.

Untuk melakukan garis besar algoritma GBFS, dibentuk *method* `findGBFS()`. *Method* ini sebenarnya sangat mirip dengan *method* `findUCS()` milik kelas UCS. Hanya saja ketika pembentukan *PriorityQueue queue*, node yang disimpan diurutkan menaik berdasarkan besar atribut (*heuristic*). Hal tersebut untuk menyesuaikan dengan teori yang menyatakan bahwa UCS adalah pencarian rute berdasarkan biaya yang telah dilalui dan GBFS adalah pencarian rute berdasarkan perkiraan biaya yang akan dilalui.

Secara teoritis, algoritma Greedy Best First Search (GBFS) tidak menjamin solusi optimal untuk persoalan word ladder. Karena heuristik dalam GBFS tidak selalu akurat sebagai pembanding jumlah biaya yang akan diperlukan. GBFS dapat tertipu oleh informasi yang tidak lengkap atau tidak akurat dari heuristik. Contohnya ketika heuristik dua kata sama-sama bernilai 3 huruf yang berbeda dengan kata tujuan, akan tetapi kata pertama telah memakan biaya yang lebih besar daripada kata kedua. GBFS dapat terjebak untuk memilih kata pertama karena tidak memiliki informasi mengenai biaya yang telah dilalui dan menyebabkan solusi menjadi tidak optimal secara biaya.

### 3.3. Permainan *Word Ladder* dengan Algoritma $A^*$

Algoritma  $A^*$  dapat dibilang adalah kombinasi dari algoritma UCS dan algoritma GBFS. Dibuat dalam sebuah kelas sebagai turunan Node dan memiliki atribut tambahan berupa *cost* yang merepresentasikan biaya dari node awal hingga node tersebut dan *evaluasi* yang merepresentasikan besar fungsi evaluasi ( $F(n)$ ). Untuk mendapatkan besar fungsi evaluasi ( $F(n)$ ) atau atribut *evaluasi*, dalam kasus ini dilakukan dengan cara menjumlahkan nilai heuristik (didapatkan dengan cara yang sama pada *heuristic* GBFS) dan nilai *cost* (didapatkan dengan cara yang sama pada *cost* UCS).

Sama seperti sebelum-sebelumnya, garis besar algoritma A\* terbentuk dalam *method* findAstar(). *Method* ini juga sangat mirip dengan *method* findUCS() dan findGBFS(), perbedaannya ketika pembentukan PriorityQueue *queue*, node disimpan dan diurutkan secara menaik berdasarkan besar atribut *evalusi*-nya. Fungsi heuristik yang digunakan di sini dapat dikatakan *admissible* atau untuk setiap node, yaitu nilai heuristiknya ( $h(n)$ ) selalu lebih kecil atau sama dengan biaya sesungguhnya untuk menempuh node tujuan dari node- $n$  ( $h^*(n)$ ). Hal tersebut terjadi karena heuristik didapatkan dari menghitung perbedaan huruf dengan kata tujuan dan dalam satu langkah tidak mungkin mengubah lebih dari satu huruf. Sehingga, secara teori algoritma A\* menggunakan pencarian pohon atau *tree* akan menghasilkan solusi yang optimal.

Algoritma A\* secara teoritis juga akan lebih efisien dibandingkan algoritma UCS. Hal tersebut terjadi karena algoritma A\* memiliki heuristik yang informatif untuk memandu pencarian. Dalam kasus ini, algoritma A\* dapat memprioritaskan ekspansi dari kata-kata yang lebih dekat ke kata akhir, sehingga mengarah pada pencarian yang lebih efisien secara keseluruhan. Sedangkan, algoritma UCS melakukan pencarian secara melebar tanpa mempertimbangkan heuristik. Hal tersebut menyebabkan UCS akan mengeksplorasi semua kemungkinan tanpa memperhatikan langkah tersebut menuju ke arah yang benar atau tidak. Apalagi dalam kasus Word Ladder yang jumlah biaya dapat diwakili oleh jumlah langkah, memungkinkan akan menghabiskan banyak waktu untuk mengeksplorasi jalur-jalur yang tidak produktif.

#### 3.4. Sistematika File

```
Tucil3_13522104
|-- doc
|   |-- Tucil3_13522104.pdf
|-- src
|   |-- Astar.java
|   |-- dictionary.txt
|   |-- GBFS.java
|   |-- makefile
|   |-- Node.java
|   |-- Triple.java
|   |-- UCS.java
```

```
| L-- WorldLadderGUI.java  
|-- test  
| |-- test case  
| |-- test.png  
| L-- tampilan_awal.png  
L-- README.md
```

### 3.5. Struktur Kelas dan Struktur Data

Pada pengimplementasian algoritma, program dibuat dalam sebuah kelas-kelas, mengikuti konsep bahasa pemrograman Java. Terdapat kelas *Node* yang menyimpan atribut *word* dan *parent*. Lalu terdapat kelas *UCS*, *GBFS*, dan *Astar* yang ketiganya diimplementasikan sebagai kelas turunan dari *Node*. Kelas *UCS* memiliki atribut tambahan *cost*, kelas *GBFS* memiliki atribut tambahan *heuristic*, kelas *Astar* memiliki atribut tambahan *cost* dan *evaluasi*. Selain itu juga terdapat kelas *Triple* yang memiliki tiga atribut generik, digunakan sebagai kelas bantuan dalam mengimplementasikan kelas lainnya. Sedangkan untuk *method* main disimpan dalam kelas *WordLadderGUI*.

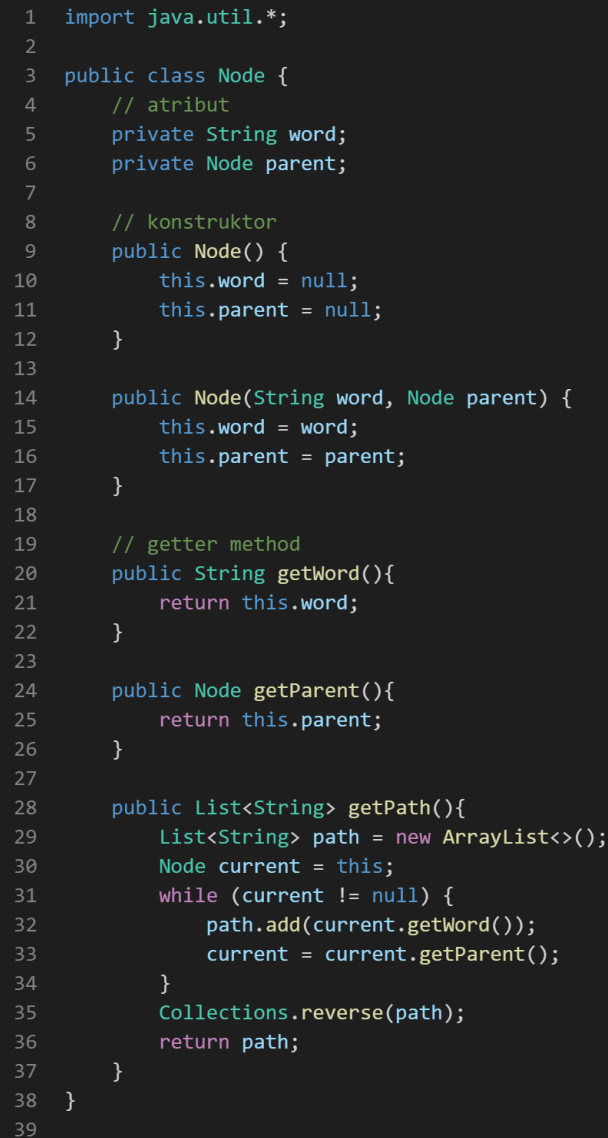
Struktur data yang dipakai pada implementasi kode adalah *PriorityQueue*, *HashSet*, dan *List*. *PriorityQueue* digunakan untuk mengurutkan menaik node berbobot secara otomatis. *HashSet* digunakan untuk menyimpan kumpulan kata dalam *dictionary* dan *visited* agar ketika dibutuhkan dalam pengecekan dalam berjalan dengan cepat dengan kompleksitas waktu cenderung konstan yaitu  $O(1)$ . Sedangkan untuk *List* digunakan untuk menyimpan daftar *children* dan daftar *path* yang dihasilkan. Ketiga struktur data tersebut telah tersedia dalam package *Java.util*.

```
1 class Point:
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6 class QuadraticBezier:
7     def __init__(self, points, iterations):
8         self.p0 = points[0]
9         self.p1 = points[1]
10        self.p2 = points[2]
11        self.iterations = iterations
12        self.elapsed_time = 0.0
13
14 class MultipointBezierDnC:
15     def __init__(self, points, iterations):
16         self.points = points # List of Point
17         self.iterations = iterations # integer
18         self.elapsed_time = 0.0
19
```

**Gambar 3.2.** Source Code “datatype.py”

### 3.6. Source Code Program

#### 3.6.1. Node.java



```
1  import java.util.*;
2
3  public class Node {
4      // atribut
5      private String word;
6      private Node parent;
7
8      // konstruktor
9      public Node() {
10         this.word = null;
11         this.parent = null;
12     }
13
14     public Node(String word, Node parent) {
15         this.word = word;
16         this.parent = parent;
17     }
18
19     // getter method
20     public String getWord(){
21         return this.word;
22     }
23
24     public Node getParent(){
25         return this.parent;
26     }
27
28     public List<String> getPath(){
29         List<String> path = new ArrayList<>();
30         Node current = this;
31         while (current != null) {
32             path.add(current.getWord());
33             current = current.getParent();
34         }
35         Collections.reverse(path);
36         return path;
37     }
38 }
39
```

**Gambar 3.3.** Source Code “Node.java”

### 3.6.2. UCS.java

```
1  import java.util.*;
2
3  public class UCS extends Node {
4      // atribut
5      private int cost;
6
7      // konstruktor
8      public UCS(String word, UCS parent) {
9          super(word, parent);
10         if (parent != null){
11             this.cost = parent.cost + 1;
12         } else {
13             this.cost = 0;
14         }
15     }
16
17     // getter method
18     public int getCost(){
19         return this.cost;
20     }
21
22     public List<UCS> getChildren(Set<String> dictionary, Set<String> visited) {
23         // inisialisasi
24         char[] chars = this.getWord().toCharArray();
25         List<UCS> children = new ArrayList<>();
26
27         // iterasi untuk setiap perubahan huruf
28         for (int i = 0; i < chars.length; i++) {
29             char originalChar = chars[i];
30             for (char c = 'a'; c <= 'z'; c++) {
31                 if (c == originalChar) {
32                     continue;
33                 }
34                 chars[i] = c;
35                 String newWord = new String(chars);
36                 // cek apakah valid dalam dictionary dan belum tercatat di visited
37                 if (!visited.contains(newWord) && dictionary.contains(newWord)) {
38                     UCS newUCS = new UCS(newWord, this);
39                     children.add(newUCS);
40                     visited.add(newWord);
41                 }
42             }
43             chars[i] = originalChar;
44         }
45         return children;
46     }
47
48     // main method
49     public Triple<List<String>, Integer, Integer> findUCS(String endWord, Set<String> dictionary) {
50         // inisialisasi antrian node (queue), node terkunjungi (visited), jumlah iterasi
51         Queue<UCS> queue = new PriorityQueue<>(Comparator.comparingInt(UCS::getCost).thenComparingInt(System::identityHashCode));
52         Set<String> visited = new HashSet<>();
53         int iterations = 0;
54
55         // masukan node ke dalam queue dan visited
56         queue.offer(this);
57         visited.add(this.getWord());
58
59         // looping hingga queue kosong atau node tujuan ditemukan
60         while (!queue.isEmpty()) {
61             iterations++;
62             UCS current = queue.poll();
63             if (current.getWord().equals(endWord)) {
64                 return new Triple<>(current.getPath(), visited.size(), iterations);
65             }
66             List<UCS> children = current.getChildren(dictionary, visited);
67             queue.addAll(children);
68         }
69         return new Triple<>(null, visited.size(), iterations);
70         // output: path, jumlah simpul hidup, jumlah simpul ekspansi (iterasi)
71     }
72 }
73
```

Gambar 3.4. Source Code “UCS.java”



### 3.6.3. GBFS.java

```
1  import java.util.*;
2
3  public class GBFS extends Node {
4      // atribut
5      private int heuristic;
6
7      // konstruktor
8      public GBFS(String word, String end, GBFS parent) {
9          super(word, parent);
10         this.heuristic = getHeuristic(word, end);
11     }
12
13     public int getHeuristic(String current, String end) {
14         // fungsi heuristik (jarak) antara kata saat ini dan kata akhir
15         int count = 0;
16         for (int i = 0; i < end.length(); i++) {
17             if (end.charAt(i) != current.charAt(i)) {
18                 count++;
19             }
20         }
21         return count;
22     }
23
24     public List<GBFS> getChildren(Set<String> dictionary, String end, Set<String> visited) {
25         // inialisasi
26         char[] chars = this.getWord().toCharArray();
27         List<GBFS> children = new ArrayList<>();
28
29         // iterasi untuk setiap perubahan huruf
30         for (int i = 0; i < chars.length; i++) {
31             char originalChar = chars[i];
32             for (char c = 'a'; c <= 'z'; c++) {
33                 if (c == originalChar) {
34                     continue;
35                 }
36                 chars[i] = c;
37                 String newWord = new String(chars);
38                 // cek apakah valid dalam dictionary dan belum tercatat di visited
39                 if (!visited.contains(newWord) && dictionary.contains(newWord)) {
40                     GBFS newGBFS = new GBFS(newWord, end, this);
41                     children.add(newGBFS);
42                     visited.add(newWord);
43                 }
44             }
45             chars[i] = originalChar;
46         }
47         return children;
48     }
49
50     // main method
51     public Triple<List<String>, Integer, Integer> findGBFS(String endWord, Set<String> dictionary) {
52         // inialisasi antrian node (queue), node terkoneksi (visited), jumlah iterasi
53         Queue<GBFS> queue = new PriorityQueue<GBFS>(Comparator.comparingInt(GBFS -> GBFS.heuristic));
54         Set<String> visited = new HashSet<>();
55         int iterations = 0;
56
57         // simpan startWord atau this ke dalam queue dan visited
58         queue.offer(this);
59         visited.add(this.getWord());
60
61         // looping hingga queue kosong atau node tujuan ditemukan
62         while (!queue.isEmpty()) {
63             iterations++;
64             GBFS current = queue.poll();
65             if (current.getWord().equals(endWord)) {
66                 return new Triple<>(current.getPath(), visited.size(), iterations);
67             }
68
69             List<GBFS> children = current.getChildren(dictionary, endWord, visited);
70             queue.addAll(children);
71         }
72
73         return new Triple<>(null, visited.size(), iterations);
74         // output: path, jumlah simpul hidup, jumlah simpul ekspansi (iterasi)
75     }
76 }
77
```

Gambar 3.5. Source Code “GBFS.java”

### 3.6.4. Astar.java

```
1 import java.util.*;
2
3 public class Astar extends Node {
4     // atribut
5     private int cost;
6     private int evaluasi;
7
8     // konstruktor
9     public Astar(String word, String end, Astar parent) {
10         super(word, parent);
11         if (parent != null){
12             this.cost = parent.cost + 1;
13         } else {
14             this.cost = 0;
15         }
16         this.evaluasi = getHeuristic(word, end) + cost;
17     }
18
19     // getter method
20     public int getEvaluasi(){
21         return this.evaluasi;
22     }
23
24     public int getHeuristic(String current, String end) {
25         // fungsi heuristik (jarak) antara kata saat ini dan kata akhir
26         int count = 0;
27         for (int i = 0; i < end.length(); i++) {
28             if (end.charAt(i) != current.charAt(i)) {
29                 count++;
30             }
31         }
32         return count;
33     }
34
35     public List<Astar> getChildren(Set<String> dictionary, String end, Set<String> visited){
36         // inisialisasi
37         char [] chars = this.getWord().toCharArray();
38         List<Astar> children = new ArrayList<>();
39
40         // iterasi untuk setiap pengubahan huruf
41         for (int i = 0; i < chars.length; i++){
42             char originalChar = chars[i];
43             for (char c = 'a'; c <= 'z'; c++){
44                 if (c == originalChar) {
45                     continue;
46                 }
47
48                 chars[i] = c;
49                 String newWord = new String(chars);
50                 // cek apakah valid dalam dictionary dan belum tercatat di visited
51                 if (!visited.contains(newWord) && dictionary.contains(newWord)) {
52                     Astar newAstar = new Astar(newWord, end, this);
53                     children.add(newAstar);
54                     visited.add(newWord);
55                 }
56             }
57             chars[i] = originalChar;
58         }
59         return children;
60     }
61
62     // main method
63     public Triple<List<String>, Integer, Integer> findAstar(String endWord, Set<String> dictionary){
64         // inisialisasi antrian node (queue), node terkunjungi (visited), jumlah iterasi
65         Queue<Astar> queue = new PriorityQueue<Astar>(Comparator.comparingInt(Astar::getEvaluasi).thenComparingInt(System::identityHashCode));
66         Set<String> visited = new HashSet<>();
67         int iterations = 0;
68
69         // simpan startWord atau this ke dalam queue dan visited
70         queue.offer(this);
71         visited.add(this.getWord());
72
73         // looping hingga queue kosong atau node tujuan ditemukan
74         while (!queue.isEmpty()){
75             iterations++;
76             Astar current = queue.poll();
77             if (current.getWord().equals(endWord)){
78                 return new Triple<>(current.getPath(), visited.size(), iterations);
79             }
80
81             List<Astar> children = current.getChildren(dictionary, endWord, visited);
82             queue.addAll(children);
83         }
84
85         return new Triple<>(null, visited.size(), iterations);
86         // output: path, jumlah simpul hidup, jumlah simpul ekspansi (iterasi)
87     }
88 }
89
```

Gambar 3.6. Source Code “Astar.java”

3.6.5. Triple.java



```
1 public class Triple<A, B, C> {  
2     // atribut  
3     public final A first;  
4     public final B second;  
5     public final C third;  
6  
7     // konstruktor  
8     public Triple(A first, B second, C third) {  
9         this.first = first;  
10        this.second = second;  
11        this.third = third;  
12    }  
13 }  
14
```

**Gambar 3.8.** Source Code “Triple.java”

### 3.6.6. WordLadderGUI.java

```
1  import javax.swing.*;
2  import javax.swing.border.EmptyBorder;
3  import javax.swing.text.BadLocationException;
4  import javax.swing.text.SimpleAttributeSet;
5
6  import java.awt.*;
7  import java.awt.event.*;
8  import java.io.BufferedReader;
9  import java.io.FileReader;
10 import java.io.IOException;
11 import java.util.*;
12 import java.util.List;
13
14 public class WordLadderGUI extends JFrame implements ActionListener {
15     // atribut
16     private JTextField startField, endField;
17     private JTextArea resultArea;
18     private JButton findButton;
19     private JCheckBox ucsCheckBox, gbfsCheckBox, aStarCheckBox;
20
21     // konstruktor
22     public WordLadderGUI() {
23         setTitle("Word Ladder Solver");
24         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25         setSize(800, 500);
26         setLayout(new BorderLayout());
27
28         // bagian input kata
29         JPanel inputPanel = new JPanel(new GridBagLayout());
30         inputPanel.setBackground(new Color(255, 192, 203));
31
32         GridBagConstraints gbc = new GridBagConstraints();
33         gbc.gridx = 0;
34         gbc.gridy = 0;
35         gbc.anchor = GridBagConstraints.WEST;
36         gbc.insets = new Insets(10, 10, 10, 10);
37
38         Font labelFont = new Font("Arial", Font.BOLD, 14);
39
40         JLabel labelKataAwal = new JLabel("Kata Awal:");
41         labelKataAwal.setFont(labelFont);
42         inputPanel.add(labelKataAwal, gbc);
43
44         gbc.gridy++;
45
46         JLabel labelKataAkhir = new JLabel("Kata Akhir:");
47         labelKataAkhir.setFont(labelFont);
48         inputPanel.add(labelKataAkhir, gbc);
49
50         gbc.gridy++;
51
52         // bagian pilih algoritma
53         JLabel labelAlgoritma = new JLabel("Pilih algoritma yang akan dijalankan:");
54         labelAlgoritma.setFont(labelFont);
55         inputPanel.add(labelAlgoritma, gbc);
56
57         gbc.gridx = 1;
58         gbc.gridy = 0;
59         gbc.fill = GridBagConstraints.HORIZONTAL;
60         gbc.weightx = 1.0;
61
```

```
1
2     startField = new JTextField();
3     startField.setFont(new Font("SansSerif", Font.PLAIN, 14));
4     inputPanel.add(startField, gbc);
5     startField.setBackground(new Color(255, 240, 245));
6     gbc.gridy++;
7     endField = new JTextField();
8     endField.setFont(new Font("SansSerif", Font.PLAIN, 14));
9     inputPanel.add(endField, gbc);
10    endField.setBackground(new Color(255, 240, 245));
11
12    gbc.gridy++;
13    ucsCheckBox = new JCheckBox("UCS");
14    gbfsCheckBox = new JCheckBox("Greedy Best-First");
15    aStarCheckBox = new JCheckBox("A*");
16    ucsCheckBox.setFont(new Font("SansSerif", Font.BOLD, 14));
17    gbfsCheckBox.setFont(new Font("SansSerif", Font.BOLD, 14));
18    aStarCheckBox.setFont(new Font("SansSerif", Font.BOLD, 14));
19    ucsCheckBox.setBackground(new Color(255, 192, 203));
20    gbfsCheckBox.setBackground(new Color(255, 192, 203));
21    aStarCheckBox.setBackground(new Color(255, 192, 203));
22
23    JPanel checkBoxPanel = new JPanel(new GridLayout(1, 3));
24    checkBoxPanel.setBackground(new Color(255, 192, 203));
25    checkBoxPanel.add(ucsCheckBox);
26    checkBoxPanel.add(gbfsCheckBox);
27    checkBoxPanel.add(aStarCheckBox);
28    inputPanel.add(checkBoxPanel, gbc);
29
30    // bagian button cari jalur
31    JPanel buttonPanel = new JPanel();
32    findButton = new JButton("Cari Jalur");
33    findButton.setFont(new Font("SansSerif", Font.BOLD, 14));
34    findButton.addActionListener(this);
35    buttonPanel.add(findButton);
36    buttonPanel.setBackground(new Color(255, 192, 203));
37    buttonPanel.setBorder(new EmptyBorder(0, 10, 10, 10));
38
39    // bagian result area
40    JPanel mainPanel = new JPanel(new BorderLayout());
41    mainPanel.setBackground(new Color(255, 192, 203));
42    mainPanel.add(inputPanel, BorderLayout.NORTH);
43    mainPanel.add(buttonPanel, BorderLayout.CENTER);
44
45    resultArea = new JTextArea("Hasil pencarian akan muncul di sini.");
46    resultArea.setEditable(false);
47    resultArea.setBackground(new Color(255, 240, 245)); // Memberikan warna latar belakang (Light Blue)
48    resultArea.setForeground(Color.BLACK); // Memberikan warna teks (Hitam)
49    resultArea.setMargin(new Insets(10, 10, 10, 10)); // Menambahkan padding
50    resultArea.setLineWrap(true); // Mengaktifkan pembungkusan baris
51    resultArea.setWrapStyleWord(true);
52    resultArea.setFont(new Font("Monospaced", Font.PLAIN, 14));
53
54    mainPanel.add(resultArea, BorderLayout.LINE_END);
55    add(mainPanel, BorderLayout.NORTH);
56    add(new JScrollPane(resultArea), BorderLayout.CENTER);
57 }
58
```

```
1
2 public void actionPerformed(ActionEvent e) {
3     if (e.getSource() == findButton) {
4         String start = startField.getText().toLowerCase();
5         String end = endField.getText().toLowerCase();
6
7         // Memastikan panjang kedua kata sama
8         if (start.length() != end.length()) {
9             JOptionPane.showMessageDialog(this, "Kata awal dan kata akhir harus memiliki panjang yang sama.");
10            return;
11        }
12
13        Set<String> words = loadDictionary("dictionary.txt");
14
15        // memastikan kata inputan tersedia di dictionary
16        if (!words.contains(start)){
17            JOptionPane.showMessageDialog(this, "Kata awal tidak terdaftar dalam kamus.");
18            return;
19        }
20        if (!words.contains(end)){
21            JOptionPane.showMessageDialog(this, "Kata akhir tidak terdaftar dalam kamus.");
22            return;
23        }
24
25        resultArea.setText("");
26
27        // pemanggilan method sesuai pilihan algoritma
28        long startTime, endTime;
29        if (ucsCheckBox.isSelected()) {
30            resultArea.append("UCS:\n");
31            startTime = System.nanoTime();
32            UCS startUCS = new UCS(start, null);
33            Triple<List<String>, Integer, Integer> ucsLadder = startUCS.findUCS(end, words);
34            printLadder(ucsLadder.first);
35            if (ucsLadder.first != null){
36                resultArea.append("Panjang path: " + (ucsLadder.first.size()-1) + "\n");
37            }
38            resultArea.append("Jumlah node dikunjungi (simpul ekspan): " + ucsLadder.third + "\n");
39            resultArea.append("Jumlah simpul hidup: " + ucsLadder.second + "\n");
40            endTime = System.nanoTime();
41            resultArea.append("Waktu eksekusi UCS: " + ((endTime - startTime) / 1000000) + " milidetik\n");
42        }
43        if (gbfsCheckBox.isSelected()) {
44            resultArea.append("\nGreedy Best-First:\n");
45            startTime = System.nanoTime();
46            GBFS startGBFS = new GBFS(start, end, null);
47            Triple<List<String>, Integer, Integer> gbfsLadder = startGBFS.findGBFS(end, words);
48            printLadder(gbfsLadder.first);
49            if (gbfsLadder.first != null){
50                resultArea.append("Panjang path: " + (gbfsLadder.first.size()-1) + "\n");
51            }
52            resultArea.append("Jumlah node dikunjungi (simpul ekspan): " + gbfsLadder.third + "\n");
53            resultArea.append("Jumlah simpul hidup: " + gbfsLadder.second + "\n");
54            endTime = System.nanoTime();
55            resultArea.append("Waktu eksekusi GBFS: " + ((endTime - startTime) / 1000000) + " milidetik\n");
56        }
57        if (aStarCheckBox.isSelected()) {
58            resultArea.append("\nA*:\n");
59            startTime = System.nanoTime();
60            Astar startAstar = new Astar(start, end, null);
61            Triple<List<String>, Integer, Integer> astarLadder = startAstar.findAstar(end, words);
62            printLadder(astarLadder.first);
63            if (astarLadder.first != null){
64                resultArea.append("Panjang path: " + (astarLadder.first.size()-1) + "\n");
65            }
66            resultArea.append("Jumlah node dikunjungi (simpul ekspan): " + astarLadder.third + "\n");
67            resultArea.append("Jumlah simpul hidup: " + astarLadder.second + "\n");
68            endTime = System.nanoTime();
69            resultArea.append("Waktu eksekusi A*: " + ((endTime - startTime) / 1000000) + " milidetik\n");
70        }
71        if (!ucsCheckBox.isSelected() && !gbfsCheckBox.isSelected() && !aStarCheckBox.isSelected()){
72            resultArea.append("Tidak ada algoritma pencarian yang dipilih.");
73        }
74    }
75 }
76
```

```
1
2 // method untuk mengubah file txt dictionary menjadi HashSet
3 public static Set<String> loadDictionary(String filename) {
4     Set<String> dictionary = new HashSet<>();
5     try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
6         String line;
7         while ((line = reader.readLine()) != null) {
8             dictionary.add(line.trim());
9         }
10    } catch (IOException e) {
11        e.printStackTrace();
12    }
13    return dictionary;
14 }
15
16 // method untuk mencetak hasil path
17 public void printLadder(List<String> ladder) {
18     if (ladder == null) {
19         resultArea.append("Tidak ada jalur yang ditemukan.\n");
20     } else {
21         resultArea.append("Jalur ditemukan:\n");
22         for (String word : ladder) {
23             resultArea.append("    [" + word + "]\n");
24         }
25     }
26 }
27
28 // main
29 public static void main(String[] args) {
30     SwingUtilities.invokeLater(() -> {
31         new WordLadderGUI().setVisible(true);
32     });
33 }
34 }
35
```

**Gambar 3.10.** Source Code “WordLadderGUI.java”

## BAB IV

### PENGUJIAN DAN ANALISIS

#### 4.1. Spesifikasi Komputer Uji Coba

Uji coba dilakukan pada perangkat dengan spesifikasi sebagai berikut.

**Tabel 4.1.1** Spesifikasi Perangkat Uji Coba

|          |                              |
|----------|------------------------------|
| Nama     | HP Pavilion Gaming Laptop 15 |
| Prosesor | AMD Ryzen 5 5600H            |
| RAM      | 16 GB                        |
| OS       | Microsoft Windows 11 HSL     |

#### 4.2. Uji Coba

Telah dilakukan uji coba pencarian solusi permainan *Word Ladder* menggunakan ketiga algoritma sekaligus. Kamus kata diambil dari [dictionary.txt](#). Berdasarkan program yang telah dibuat dan tampilan dalam sebuah GUI, berikut beberapa hasil yang didapatkan:

**Tabel 4.2.1** Tabel Hasil Uji Coba dengan Tangkapan Layar

| No. | Tangkapan Layar |               |
|-----|-----------------|---------------|
| 1.  | Start word: you | End word: cat |



The screenshot shows a window titled "Word Ladder Solver". The interface has a pink header area with input fields and checkboxes, and a light pink main area for output text.

**Input Fields:**

- Kata Awal:** you
- Kata Akhir:** cat

**Algorithm Selection:**

Pilih algoritma yang akan dijalankan: ☒ UCS ☒ Greedy Best-First ☒ A\*

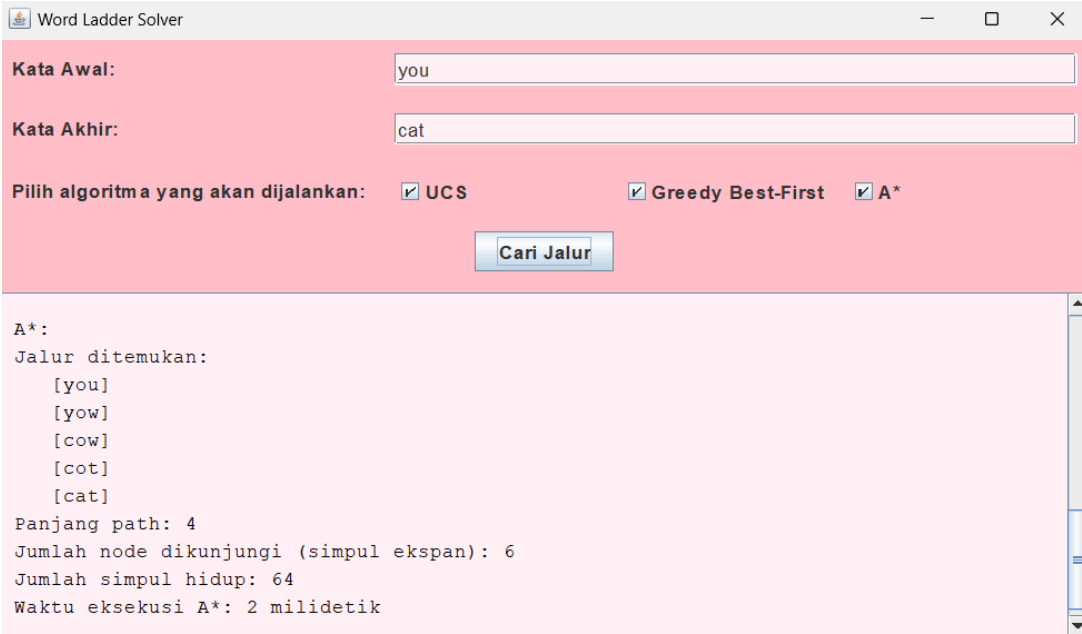
**Buttons:**

Cari Jalur

**Output Results:**

**UCS:**  
Jalur ditemukan:  
[you]  
[yon]  
[con]  
[can]  
[cat]  
Panjang path: 4  
Jumlah node dikunjungi (simpul ekspan): 477  
Jumlah simpul hidup: 835  
Waktu eksekusi UCS: 17 milidetik

**Greedy Best-First:**  
Jalur ditemukan:  
[you]  
[yow]  
[cow]  
[caw]  
[cat]  
Panjang path: 4  
Jumlah node dikunjungi (simpul ekspan): 6  
Jumlah simpul hidup: 70  
Waktu eksekusi GBFS: 3 milidetik

|                  |  |                  |                |
|------------------|--|------------------|----------------|
|                  |  <p>The screenshot shows a window titled "Word Ladder Solver". It has two input fields: "Kata Awal:" with the value "you" and "Kata Akhir:" with the value "cat". Below these are three checkboxes for algorithm selection: "UCS", "Greedy Best-First", and "A*", all of which are checked. A "Cari Jalur" button is positioned below the checkboxes. The output area at the bottom displays the following text:<br/>A*:<br/>Jalur ditemukan:<br/>[you]<br/>[yow]<br/>[cow]<br/>[cot]<br/>[cat]<br/>Panjang path: 4<br/>Jumlah node dikunjungi (simpul ekspan): 6<br/>Jumlah simpul hidup: 64<br/>Waktu eksekusi A*: 2 milidetik</p> |                  |                |
| 2.               | <table><tr><td data-bbox="329 926 881 997">Start word: ride</td><td data-bbox="881 926 1430 997">End word: pool</td></tr></table>  | Start word: ride | End word: pool |
| Start word: ride | End word: pool   |                  |                |

Word Ladder Solver



Kata Awal:

Kata Akhir:

Pilih algoritma yang akan dijalankan: ☒ UCS ☒ Greedy Best-First ☒ A\*

UCS:  
Jalur ditemukan:  
[ride]  
[rile]  
[rill]  
[pill]  
[poll]  
[pool]  
Panjang path: 5  
Jumlah node dikunjungi (simpul ekspan): 2747  
Jumlah simpul hidup: 3297  
Waktu eksekusi UCS: 30 milidetik

Greedy Best-First:  
Jalur ditemukan:  
[ride]  
[rode]  
[rods]  
[pods]  
[pols]  
[poll]  
[pool]  
Panjang path: 6  
Jumlah node dikunjungi (simpul ekspan): 11  
Jumlah simpul hidup: 108  
Waktu eksekusi GBFS: 0 milidetik

|                  |   |                  |                |
|------------------|---|------------------|----------------|
|                  |  <p>Word Ladder Solver</p> <p>Kata Awal: ride</p> <p>Kata Akhir: pool</p> <p>Pilih algoritma yang akan dijalankan: <input checked="" type="checkbox"/> UCS <input checked="" type="checkbox"/> Greedy Best-First <input checked="" type="checkbox"/> A*</p> <p>Cari Jalur</p> <p>A*:<br/>Jalur ditemukan:<br/>[ride]<br/>[rile]<br/>[rill]<br/>[pill]<br/>[poll]<br/>[pool]<br/>Panjang path: 5<br/>Jumlah node dikunjungi (simpul ekspan): 45<br/>Jumlah simpul hidup: 298<br/>Waktu eksekusi A*: 1 milidetik</p>  |                  |                |
| 3.               | <table><tr><td>Start word: draw</td><td>End word: rose</td></tr></table>  <p>Word Ladder Solver</p> <p>Kata Awal: draw</p> <p>Kata Akhir: rose</p> <p>Pilih algoritma yang akan dijalankan: <input checked="" type="checkbox"/> UCS <input checked="" type="checkbox"/> Greedy Best-First <input checked="" type="checkbox"/> A*</p> <p>Cari Jalur</p> <p>UCS:<br/>Jalur ditemukan:<br/>[draw]<br/>[drat]<br/>[doat]<br/>[dost]<br/>[dose]<br/>[rose]<br/>Panjang path: 5<br/>Jumlah node dikunjungi (simpul ekspan): 468<br/>Jumlah simpul hidup: 1460<br/>Waktu eksekusi UCS: 5 milidetik</p> | Start word: draw | End word: rose |
| Start word: draw | End word: rose  |                  |                |

|                   |   |                   |                 |
|-------------------|---|-------------------|-----------------|
|                   | <pre>Greedy Best-First: Jalur ditemukan:   [draw]   [braw]   [brae]   [bree]   [tree]   [tyee]   [tyke]   [ryke]   [rake]   [rase]   [rose]  Panjang path: 10 Jumlah node dikunjungi (simpul ekspan): 24 Jumlah simpul hidup: 149 Waktu eksekusi GBFS: 2 milidetik  A*: Jalur ditemukan:   [draw]   [drat]   [doat]   [dost]   [dose]   [rose]  Panjang path: 5 Jumlah node dikunjungi (simpul ekspan): 15 Jumlah simpul hidup: 87 Waktu eksekusi A*: 0 milidetik</pre> |                   |                 |
| 4.                | <table><tr><td data-bbox="329 1184 881 1251">Start word: black</td><td data-bbox="881 1184 1430 1251">End word: white</td></tr></table>   | Start word: black | End word: white |
| Start word: black | End word: white   |                   |                 |

Word Ladder Solver

Kata Awal:

black

Kata Akhir:

white

Pilih algoritma yang akan dijalankan:

☒ UCS

☒ Greedy Best-First

☒ A\*

Cari Jalur

UCS:

Jalur ditemukan:

[black]

[clack]

[click]

[clink]

[chink]

[chine]

[whine]

[white]

Panjang path: 7

Jumlah node dikunjungi (simpul ekspan): 2353

Jumlah simpul hidup: 3731

Waktu eksekusi UCS: 55 milidetik

Greedy Best-First:

Jalur ditemukan:

[black]

[blank]

[blink]

[clink]

[chink]

[chine]

[whine]

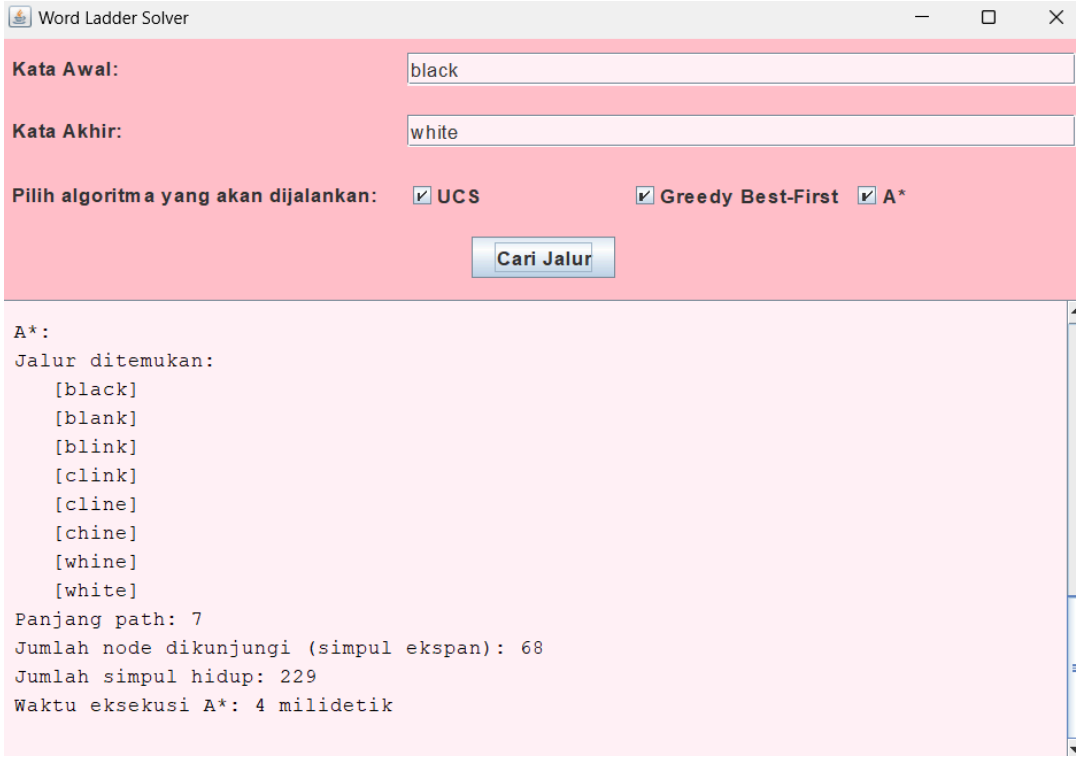
[white]

Panjang path: 7

Jumlah node dikunjungi (simpul ekspan): 9

Jumlah simpul hidup: 47

Waktu eksekusi GBFS: 1 milidetik

|                    |  |                    |                  |
|--------------------|--|--------------------|------------------|
|                    |  <p>The screenshot shows a window titled "Word Ladder Solver". It has two input fields: "Kata Awal:" with the value "black" and "Kata Akhir:" with the value "white". Below these are three checkboxes for algorithm selection: "UCS" (checked), "Greedy Best-First" (checked), and "A*" (checked). A "Cari Jalur" button is positioned below the checkboxes. The output area shows the A* search results:</p> <pre> A*: Jalur ditemukan:   [black]   [blank]   [blink]   [clink]   [cline]   [chine]   [whine]   [white] Panjang path: 7 Jumlah node dikunjungi (simpul ekspan): 68 Jumlah simpul hidup: 229 Waktu eksekusi A*: 4 milidetik </pre> |                    |                  |
| 5.                 | <table border="1"> <tr> <td>Start word: tonish</td> <td>End word: launch</td> </tr> </table>   | Start word: tonish | End word: launch |
| Start word: tonish | End word: launch   |                    |                  |

Word Ladder Solver

Kata Awal:

tonish

Kata Akhir:

launch

Pilih algoritma yang akan dijalankan:

☒ UCS

☒ Greedy Best-First

☒ A\*

Cari Jalur

UCS:

Jalur ditemukan:

[tonish]  
[monish]  
[mopish]  
[popish]  
[polish]  
[palish]  
[parish]  
[pariah]  
[parian]  
[partan]  
[tartan]  
[tartar]  
[tarter]  
[carter]  
[cartes]  
[carses]  
[corses]  
[horses]  
[horsts]  
[hoists]  
[joists]  
[joints]  
[points]  
[poinds]  
[pounds]  
[mounds]



```
[mounts]
[counts]
[county]
[bounty]
[bouncy]
[jouncy]
[jounce]
[jaunce]
[launce]
[launch]
Panjang path: 35
Jumlah node dikunjungi (simpul ekspan): 8397
Jumlah simpul hidup: 8399
Waktu eksekusi UCS: 148 milidetik

Greedy Best-First:
Jalur ditemukan:
[tonish]
[monish]
[mopish]
[popish]
[polish]
[palish]
[parish]
[pariah]
[parian]
[partan]
[tartan]
[tartar]
```

```
[tarter]
[tauter]
[tauted]
[sauted]
[sauced]
[saucer]
[sauger]
[mauger]
[mauler]
[mauled]
[wauled]
[wauked]
[jauked]
[jauped]
[yauped]
[yauper]
[pauper]
[pauser]
[causer]
[causes]
[caules]
[caulds]
[faulds]
[faulds]
[faulds]
[vaults]
[vaunts]
[taunts]
[taints]
[paints]
```

```
[points]
[poinds]
[pounds]
[founds]
[founts]
[counts]
[county]
[bounty]
[bouncy]
[jouncy]
[jounce]
[jaunce]
[launce]
[launch]
Panjang path: 54
Jumlah node dikunjungi (simpul ekspan): 1718
Jumlah simpul hidup: 3131
Waktu eksekusi GBFS: 64 milidetik

A*:
Jalur ditemukan:
    [tonish]
    [monish]
    [mopish]
    [popish]
    [polish]
    [palish]
    [parish]
    [pariah]
```

|                      |  |                      |                    |
|----------------------|--|----------------------|--------------------|
|                      | <pre>[parian] [partan] [tartan] [tartar] [tarter] [tauter] [touter] [router] [rouser] [rouses] [rousts] [jousts] [joists] [joints] [points] [pounds] [pounds] [mounds] [mounts] [counts] [county] [bounty] [bouncy] [bounce] [jounce] [jaunce] [launce] [launch] Panjang path: 35 Panjang path: 35 Jumlah node dikunjungi (simpul ekspan): 8332 Jumlah simpul hidup: 8357 Waktu eksekusi A*: 158 milidetik</pre> |                      |                    |
| 6.                   | <table><tr><td data-bbox="331 1289 883 1358">Start word: spelling</td><td data-bbox="883 1289 1430 1358">End word: coaching</td></tr></table>  | Start word: spelling | End word: coaching |
| Start word: spelling | End word: coaching   |                      |                    |

Word Ladder Solver

Kata Awal:

spelling

Kata Akhir:

coaching

Pilih algoritma yang akan dijalankan:

☒ UCS

☒ Greedy Best-First

☒ A\*

Cari Jalur

UCS:

Jalur ditemukan:

[spelling]  
[spalling]  
[sparling]  
[sparring]  
[scarring]  
[scarting]  
[scatting]  
[slatting]  
[blatting]  
[blasting]  
[boasting]  
[coasting]  
[coacting]  
[coaching]

Panjang path: 13  
Jumlah node dikunjungi (simpul ekspan): 750  
Jumlah simpul hidup: 772  
Waktu eksekusi UCS: 29 milidetik

Greedy Best-First:

Jalur ditemukan:

[spelling]  
[spalling]  
[sparling]  
[sparking]  
[sharking]  
[charking]  
[charging]  
[changing]  
[clanging]  
[slanging]  
[slanting]  
[slatting]  
[blatting]  
[blasting]  
[boasting]  
[coasting]  
[coacting]  
[coaching]

Panjang path: 17  
Jumlah node dikunjungi (simpul ekspan): 154  
Jumlah simpul hidup: 294  
Waktu eksekusi GBFS: 5 milidetik

```
A*:  
Jalur ditemukan:  
[spelling]  
[spalling]  
[sparling]  
[sparring]  
[scarring]  
[scarting]  
[scatting]  
[slatting]  
[blatting]  
[blasting]  
[boasting]  
[coasting]  
[coacting]  
[coaching]  
Panjang path: 13  
Jumlah node dikunjungi (simpul ekspan): 540  
Jumlah simpul hidup: 640  
Waktu eksekusi A*: 18 milidetik
```

#### 4.2.1. Hasil Uji Coba Panjang *Path*

Berikut perbandingan hasil uji coba dari parameter panjang path yang dihasilkan dari setiap algoritma:

**Tabel 4.2.2** Tabel Hasil Uji Coba dengan Parameter Panjang Path

| Nomor Uji Coba    | UCS  | GBFS | A*   |
|-------------------|------|------|------|
| 1.                | 4    | 4    | 4    |
| 2.                | 5    | 6    | 5    |
| 3.                | 5    | 10   | 5    |
| 4.                | 7    | 7    | 7    |
| 5.                | 35   | 54   | 35   |
| 6.                | 13   | 17   | 13   |
| <b>Rata-rata:</b> | 11.5 | 16.3 | 11.5 |

#### 4.2.2. Hasil Uji Coba Jumlah Node Terkunjungi

Berikut perbandingan hasil uji coba dari parameter jumlah node yang dikunjungi dari setiap algoritma:

**Tabel 4.2.3** Tabel Hasil Uji Coba dengan Parameter Jumlah Node

| <b>Nomor Uji Coba</b> | <b>UCS</b> | <b>GBFS</b> | <b>A*</b> |
|-----------------------|------------|-------------|-----------|
| 1.                    | 477        | 6           | 6         |
| 2.                    | 2747       | 11          | 45        |
| 3.                    | 468        | 24          | 15        |
| 4.                    | 2353       | 9           | 68        |
| 5.                    | 8397       | 1718        | 8332      |
| 6.                    | 750        | 154         | 540       |
| <b>Rata-rata:</b>     | 2532       | 332         | 1501      |

#### 4.2.3. Hasil Uji Coba Waktu Eksekusi

Berikut perbandingan hasil uji coba dari parameter waktu eksekusi dalam milidetik dari setiap algoritma:

**Tabel 4.2.4** Tabel Hasil Uji Coba dengan Parameter Waktu Eksekusi

| <b>Nomor Uji Coba</b> | <b>UCS</b> | <b>GBFS</b> | <b>A*</b> |
|-----------------------|------------|-------------|-----------|
| 1.                    | 17         | 3           | 2         |
| 2.                    | 30         | 0           | 1         |
| 3.                    | 5          | 2           | 0         |
| 4.                    | 55         | 1           | 4         |
| 5.                    | 148        | 64          | 158       |
| 6.                    | 29         | 5           | 18        |
| <b>Rata-rata:</b>     | 47.3       | 12.5        | 30.5      |

### 4.3. Analisis Hasil Uji Coba

Dari ketiga algoritma yang telah dibuat, didapatkan bahwa ketiganya berhasil menemukan solusi untuk permainan Word Ladder dengan waktu yang cukup singkat. Selain itu, didapatkan pula tiga parameter yang dapat dijadikan pembanding untuk perbedaan ketiga algoritma tersebut. Berikut adalah rata-rata ketiga parameter dari setiap algoritma berdasarkan uji coba yang telah dilakukan:

**Tabel 4.3.1** Tabel Hasil Rata-Rata Uji Coba dengan Ketiga Parameteri

| Parameter       | UCS       | GBFS      | A*        |
|-----------------|-----------|-----------|-----------|
| Panjang path    | 11.5 kata | 16.3 kata | 11.5 kata |
| Node dikunjungi | 2532 node | 332 node  | 1501 node |
| Waktu eksekusi  | 47.3 ms   | 12.5 ms   | 30.5 ms   |

Berdasarkan data hasil uji coba di atas didapatkan 3 kesimpulan. Pertama, panjang path algoritma UCS dan A\* selalu sama, serta algoritma GBFS panjang pathnya cenderung lebih banyak. Hal tersebut menunjukkan bahwa algoritma UCS dan A\* selalu menghasilkan solusi yang optimal secara panjang path atau jumlah biaya. Akan tetapi hal tersebut tidak berarti GBFS tidak memiliki kelebihan.

Kedua, node yang dikunjungi algoritma GBFS selalu menjadi yang paling sedikit, disusul dengan algoritma UCS. Sedangkan node terkunjungi terbanyak selalu pada algoritma UCS. Hal tersebut menunjukkan bahwa dengan adanya fungsi heuristik atau *informed search* pencarian rute menjadi lebih terarah dan tidak mengeksplorasi node-node yang tidak menuju node tujuan. Sehingga, memori yang digunakan pun dapat efisien dengan algoritma GBFS dan UCS. Oleh karena itu, dapat dikatakan bahwa algoritma UCS kurang efisien dalam hal kompleksitas memori.

Ketiga, berhubungan dengan banyaknya node yang dikunjungi, hal tersebut tentu saja juga akan memengaruhi waktu eksekusi tiap algoritma. Waktu eksekusi algoritma GBFS cenderung menjadi yang tercepat sesuai dengan jumlah node dikunjungi yang selalu paling sedikit. Disusul oleh algoritma A\* dan paling lama ditempati oleh algoritma UCS. Untuk

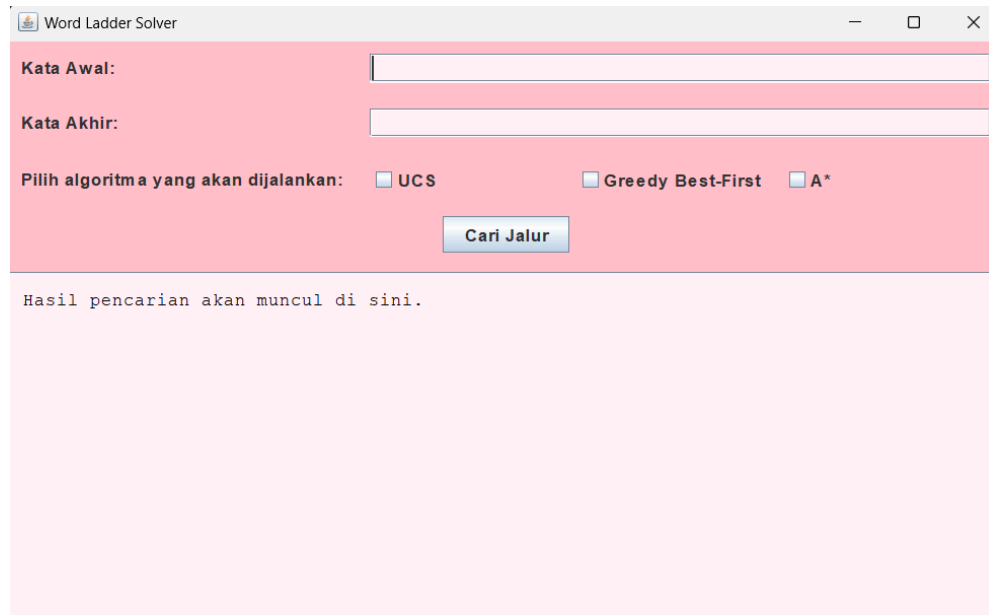


mengunjungi setiap node tentu saja membutuhkan waktu, meskipun begitu waktu yang dibutuhkan UCS rata-rata masih jauh di bawah satu detik.

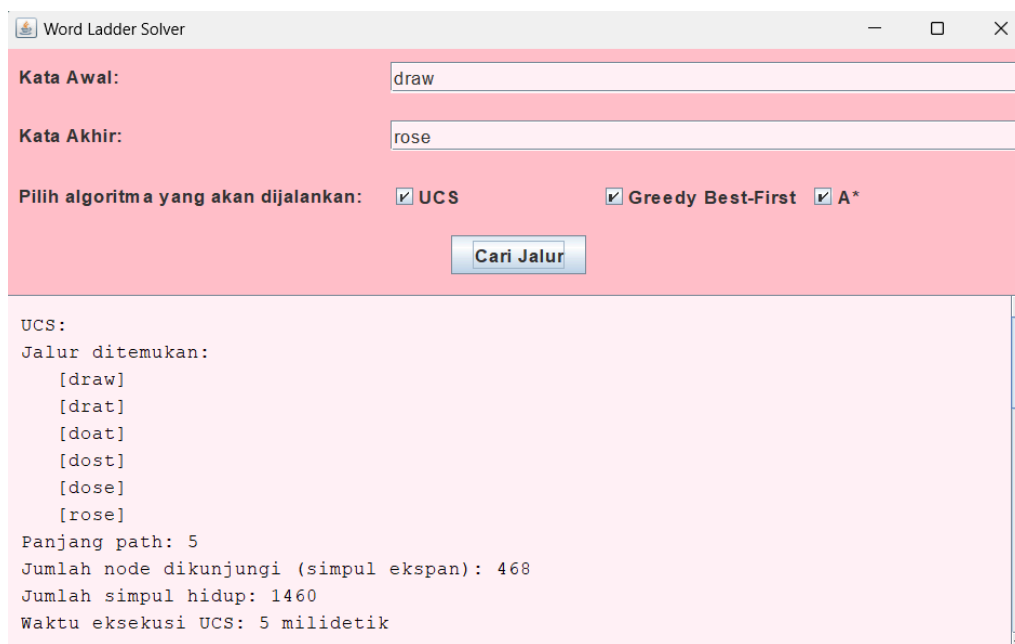
Dari ketiga kesimpulan di atas, didapatkan kesimpulan pula bahwa algoritma yang paling sesuai untuk permasalahan pada kasus kali ini adalah algoritma A\*. Karena solusi dari path yang dihasilkan selalu optimal, selain itu waktu eksekusi yang dibutuhkan juga cepat sesuai dengan jumlah node yang dikunjungi. Sehingga ruang memori yang dibutuhkan juga tidak terlalu besar. Dengan kata lain algoritma A\* yang memiliki hasil paling stabil dalam ketiga parameter tersebut. Algoritma GBFS lebih cocok dalam kasus-kasus yang hanya membutuhkan waktu yang sangat cepat dibandingkan konsisten dalam menghasilkan solusi yang optimal. Sedangkan algoritma UCS kurang cocok karena informasi bobot yang sama dengan jumlah langkah menyebabkan seperti algoritma BFS. Selain itu, UCS juga termasuk *informed search* atau tanpa fungsi heuristik sehingga kurang memiliki informasi untuk menentukan path yang mendekati titik tujuan. Di lain hal, hasil semua uji coba ini juga selaras dengan teori-teori yang telah dipaparkan sebelumnya.

#### 4.4. Bonus

Untuk memudahkan dan memberikan pengalaman yang baik bagi pengguna, telah dibuat *Graphical User Interface* (GUI) menggunakan bahasa Java dan *toolkit* Java Swing. GUI dibuat dalam satu panel sederhana dan memuat fitur-fitur sesuai kebutuhan. Pengguna dapat memasukkan kata awal dan kata akhir yang ingin dicari *path*-nya. Pengguna juga dapat memilih algoritma apa saja yang akan dilakukan, sehingga pengguna dapat melakukan lebih dari satu algoritma dalam satu waktu. Setelah itu untuk menjalankannya pengguna dapat menekan tombol yang bertuliskan ‘cari jalur’. Dengan waktu yang sangat cepat akan ditampilkan hasilnya berupa daftar *path*, panjang *path*, banyaknya node yang dikunjungi, jumlah simpul hidup yang dibuat, serta waktu eksekusi dari tiap-tiap algoritma yang dipilih.



**Gambar 4.4.1.** GUI Tampilan Awal



**Gambar 4.4.2.** GUI Tampilan Mengeluarkan Hasil Jalur

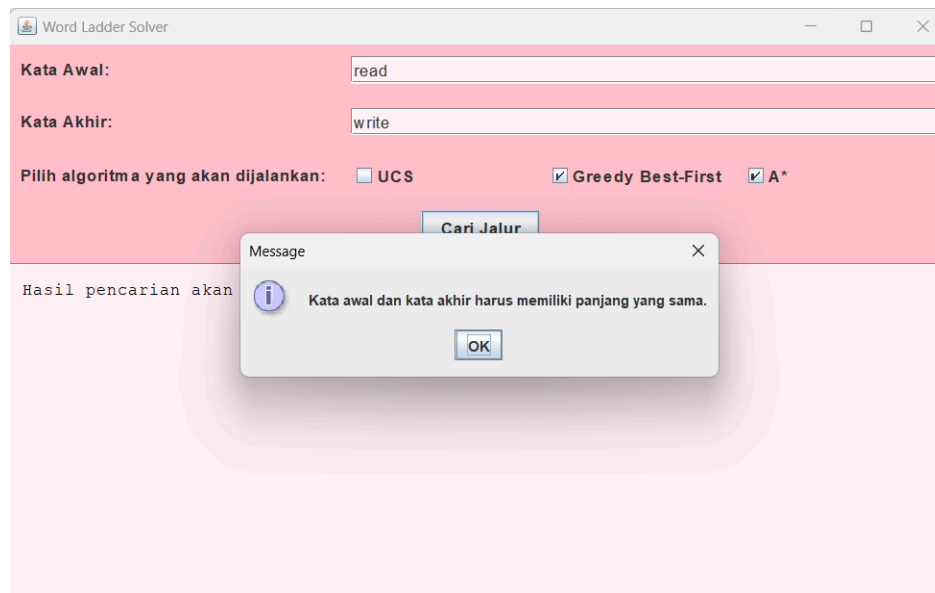
Selain itu, GUI yang dibuat juga telah mengatasi untuk hal-hal yang tidak diinginkan. Contohnya seperti masukan kata awal atau kata akhir tidak tersedia dalam *dictionary*, panjang kata awal dan kata akhir tidak sama, algoritma tidak ada yang terpilih, serta ketika tidak ditemukan path yang dihasilkan. Berikut adalah gambaran ketika terjadi hal yang tidak diinginkan:



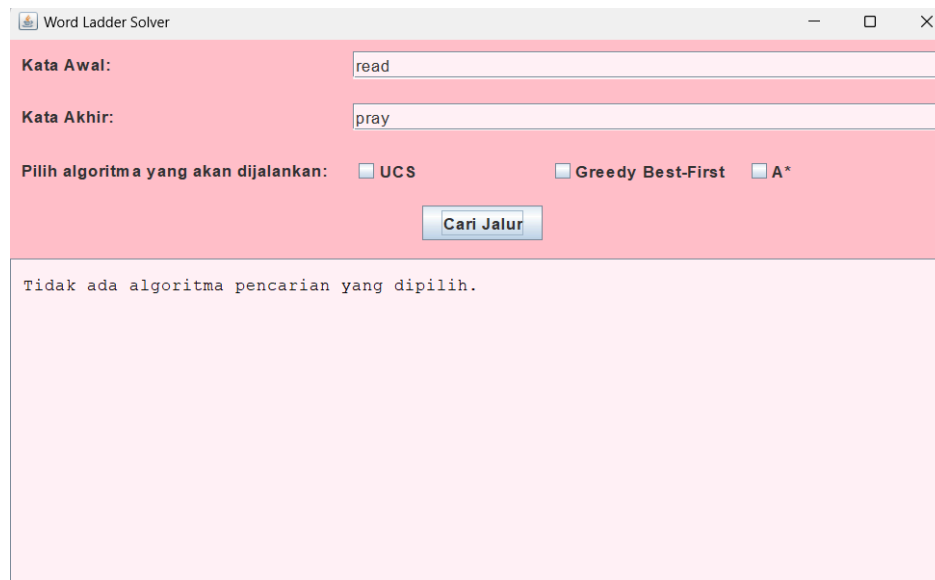
**Gambar 4.4.3.** GUI Ketika Kata Awal Tidak Terdaftar dalam Kamus



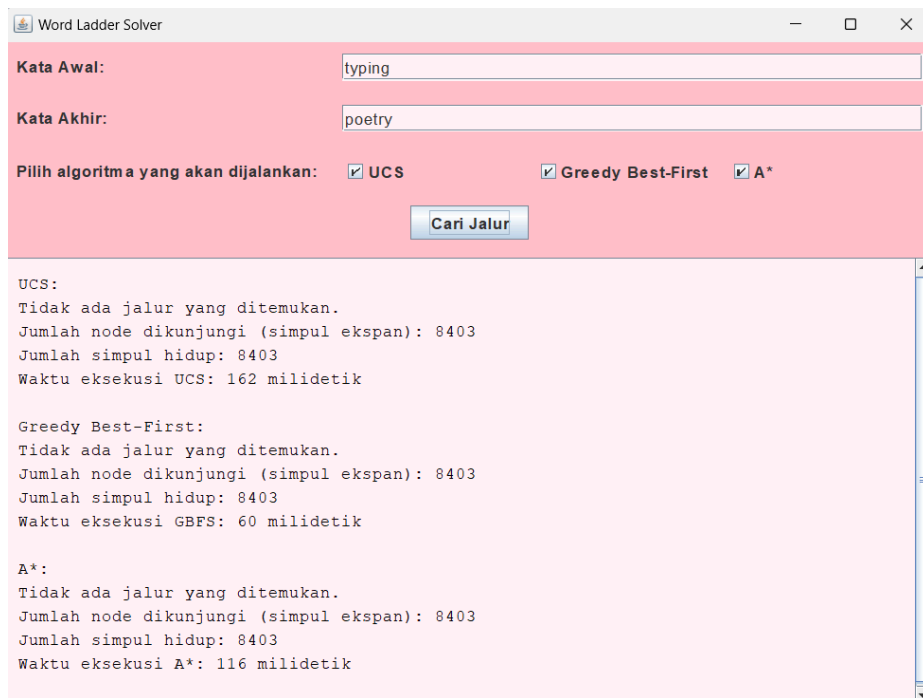
**Gambar 4.4.4.** GUI Ketika Kata Akhir Tidak Terdaftar dalam Kamus



**Gambar 4.4.5.** GUI Ketika Panjang Kata Awal dan Akhir Tidak Terdaftar Sama



**Gambar 4.4.6.** GUI Ketika Tidak Ada Algoritma yang Dipilih



**Gambar 4.4.7.** GUI Ketika Tidak Ada Jalur yang Ditemukan

Implementasi kode pembuatan GUI telah ditampilkan pada bagian Source Code Program atau dapat dilihat pada lampiran link github yang tersedia dan terletak pada file WordLadderGUI.java.

## **BAB V**

### **PENUTUP**

#### **5.1. Kesimpulan**

Algoritma *Uniform Cost Search* (UCS), *Greedy Best First Search*, dan A\* dapat mencari solusi permainan Word Ladder dengan baik. Meski tidak dapat dipungkiri juga masing-masing algoritma memiliki kelebihan dan kekurangan. Algoritma UCS selalu menghasilkan solusi yang optimal, tetapi algoritma ini memakan banyak waktu dan memori. Algoritma *Greedy Best First Search* selalu menghasilkan solusi dalam waktu yang sangat cepat dan memori yang sedikit, tetapi algoritma ini tidak selalu menghasilkan solusi yang optimal. Sedangkan algoritma A\* dapat dikatakan algoritma yang terbaik untuk permasalahan ini, karena solusi yang dihasilkan selalu optimal dan waktu serta memori yang dibutuhkan juga sedikit. Hal tersebut terjadi karena algoritma A\* mempertimbangkan fungsi biaya ( $g(n)$ ) dan fungsi heuristik ( $h(n)$ ) untuk mendapatkan fungsi evaluasi ( $F(n)$ ).

## DAFTAR PUSTAKA

- [1] Lecture Notes in Informed Heuristic Search, ICS 271 Fall 2008. Diakses pada Senin, 6 Mei 2024. Dilansir dari laman berikut :  
<http://www.ics.uci.edu/~dechter/courses/ics-271/fall-08/lecturenotes/4.InformedHeuristicSearch.ppt>
- [2] Maulidevi, NU. . (2021). *informatika.stei.itb.ac.id*, “Penentuan Rute (Route/Path Planning) Bagian 1: BFS, DFS, UCS, Greedy Best First Search”. Diakses Senin, 6 Mei 2024. Dilansir dari Laman Rinaldi Munir:  
<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf>
- [3] Maulidevi, NU. Munir, R. (2021). *informatika.stei.itb.ac.id*, “Penentuan Rute (Route/Path Planning) Bagian 2: Algoritma A\*”. Diakses Senin, 6 Mei 2024. Dilansir dari Laman Rinaldi Munir:  
<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf>
- [4] W3Schools. [www.w3schools.com](http://www.w3schools.com). “Java Tutorial”. Diakses pada Selasa, 7 Mei 2024. Dilansir pada laman berikut: <https://www.w3schools.com/java/>

## LAMPIRAN

### Lampiran 1 Link Repository

Repository untuk source code dan hasil kompilasi program ini dapat diakses melalui tautan berikut: [https://github.com/dianatrihy/Tucil3\\_13522104](https://github.com/dianatrihy/Tucil3_13522104)

### Lampiran 2 Tabel *Checklist* Poin

| No. | 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.  | <b>[Bonus]:</b> Program memiliki tampilan GUI  | ✓  |       |