

Laporan Tugas Kecil 3 IF2211 Strategi Algoritma  
Semester II tahun 2023/2024

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



Elbert Chailes – 13522045

PROGRAM STUDI TEKNIK INFORMATIKA  
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA  
INSTITUT TEKNOLOGI BANDUNG  
2023

# DAFTAR ISI

DAFTAR ISI.....	2
BAB I : DESKRIPSI MASALAH.....	4
BAB II : ALGORITMA UCS, GREEDY BEST FIRST SEARCH, A* UNTUK PENYELESAIAN PERMAINAN WORD LADDER.....	5
2.1 Algoritma Uniform Cost Search (UCS).....	5
2.2. Algoritma Greedy Best First Search (GBFS).....	7
2.3. Algoritma A-Star (A*).....	9
2.4. Analisis Teori.....	11
BAB III : IMPLEMENTASI PROGRAM DENGAN JAVA.....	14
1. Folder Utils.....	14
a. File Dictionary.java.....	14
b. File Utils.java.....	15
2. Folder algorithm.....	16
a. File WordLadder.java.....	16
b. File WordLadderAStar.java.....	17
c. File WordLadderUCS.java.....	19
d. File WordLadderGBFS.java.....	20
3. File Main.java.....	22
4. File WordLadderGUI.java (Bonus).....	22
BAB IV : EKSPERIMEN.....	25
4.1. Percobaan 1.....	25
4.1.1. Algoritma Uniform Cost Search (UCS).....	25
4.1.2. Algoritma Greedy Best First Search (GBFS).....	26
4.1.3. Algoritma A-Star (A*).....	26

4.2. Percobaan 2.....	27
4.2.1. Algoritma Uniform Cost Search (UCS).....	27
4.2.2. Algoritma Greedy Best First Search (GBFS).....	27
4.2.3. Algoritma A-Star (A*).....	28
4.3. Percobaan 3.....	28
4.3.1. Algoritma Uniform Cost Search (UCS).....	28
4.3.2. Algoritma Greedy Best First Search (GBFS).....	29
4.3.3. Algoritma A-Star (A*).....	29
4.4. Percobaan 4.....	30
4.4.1. Algoritma Uniform Cost Search (UCS).....	30
4.4.2. Algoritma Greedy Best First Search (GBFS).....	30
4.4.3. Algoritma A-Star (A*).....	31
4.5. Percobaan 5.....	31
4.5.1. Algoritma Uniform Cost Search (UCS).....	31
4.5.2. Algoritma Greedy Best First Search (GBFS).....	32
4.5.3. Algoritma A-Star (A*).....	32
4.6. Percobaan 6.....	33
4.6.1. Algoritma Uniform Cost Search (UCS).....	33
4.6.2. Algoritma Greedy Best First Search (GBFS).....	33
4.6.3. Algoritma A-Star (A*).....	34
BAB V : ANALISIS.....	35
LAMPIRAN.....	39
Github Repository.....	39
Tabel Spesifikasi.....	39

# BAB I : DESKRIPSI MASALAH

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

## How To Play

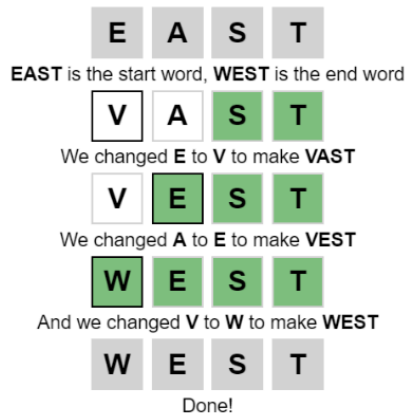
This game is called a "word ladder" and was invented by Lewis Carroll in 1877.

### Rules

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

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

### Example



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

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

# BAB II : ALGORITMA UCS, GREEDY BEST FIRST SEARCH, A\* UNTUK PENYELESAIAN PERMAINAN WORD LADDER

## 2.1 Algoritma *Uniform Cost Search* (UCS)

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

1. Definisikan terlebih dahulu kata awal (*starting word*) untuk dijadikan sebagai simpul awal pencarian dan kata tujuan (*ending word*) yang dijadikan sebagai simpul akhir atau tujuan pencarian dalam permainan ini.
2. Lakukan pengecekan atau validasi terhadap masukan dari pengguna terhadap *starting word* dan *ending word*, apakah kedua kata memiliki panjang kata yang sama. Jika kedua kata memiliki panjang kata yang berbeda, maka algoritma tidak akan melanjutkan pencarian dan menghentikan langkahnya.
3. Definisikan sebuah *priority queue* yang mengurutkan komponen di dalamnya berdasarkan *cost* yang telah dilalui hingga simpul tertentu berdasarkan *cost* yang terendah hingga tertinggi. Maka dari itu, pemrosesan selanjutnya akan dilanjutkan berdasarkan *cost* yang terendah terlebih dahulu sehingga sesuai dengan prinsip dari algoritma UCS.
  - a. Sebagai tambahan informasi bahwa *cost* dalam UCS dapat dihitung menggunakan  $f(n) = g(n)$ , dengan  $g(n)$  merupakan *cost* dari jalan yang telah ditempuh ke suatu simpul dari simpul akar. Semisal, perjalanan sebuah algoritma A-B-C, maka  $g(n)$  merupakan *cost* perjalanan A-B ditambah dengan *cost* perjalanan B-C. Di sini pula, *cost* yang kecil menandakan kedalaman perjalanan yang lebih pendek, yang berarti solusi perjalanan yang didapatkan akan diusahakan sesingkat mungkin oleh algoritma.

- b. Dalam permainan *Word Ladder* ini, bisa dikatakan bahwa *cost* perjalanan sebuah simpul dapat dihitung menggunakan *depth* dimana simpul tersebut dilalui pada sebuah perjalanan. Semisal, perjalanan algoritmanya adalah A-B-C, maka simpul C akan disimpul dengan  $g(n) = 2$ , karena *depth* 2. Penyimpanan informasi  $g(n)$  ini akan sangat penting karena jika terjadi perjalanan lain seperti A-C-B, maka C masih dipertimbangkan karena memiliki *depth* yang lebih awal sehingga berkemungkinan menjadi lebih optimal. Namun, jika ditemukan simpul C di atas kedalaman 3, maka simpul C tersebut tidak akan dipertimbangkan lagi sebagai calon solusi.
  - c. Dengan itu, maka didapatkan alasan mengapa dilakukan pengurutan pada *priority queue* dengan aturan pengurutan dari *depth* /  $g(n)$  terkecil hingga terbesar. Jika dilihat pada algoritma yang dibuat untuk menyelesaikan permainan ini,  $g(n)$  tidak lagi disebutkan secara eksplisit atau didefinisikan, melainkan hanya digunakan informasi kedalaman yang telah disimpan pada class *Node* yang menandakan simpul-simpul.
  - d. *Priority queue* diinisialisasi dengan satu simpul / *Node* yang akan memicu perulangan yang terjadi pada langkah 4, yaitu dengan inisialisasi pemasukan *node* / simpul kata mulai (*starting word*) ke dalam *priority queue*.
4. Selama *priority queue* tidak kosong, algoritma akan selalu mengunjungi simpul dengan *cost* yang paling rendah dan dilakukan pengecekan terhadap *node* tersebut apakah merupakan *goal node* atau bukan.
- a. Jika *node* sekarang merupakan *goal node*, maka algoritma akan berhenti dan mengembalikan simpul-simpul yang dilalui oleh algoritma untuk mencapai *goal node* tersebut.
  - b. Jika *node* bukan merupakan *goal node*, maka algoritma akan dilanjutkan ke langkah selanjutnya.
5. Lakukan pembangkitan *node* anak dari *node* yang sedang dikunjungi oleh algoritma. Maka dari itu, *node* yang sekarang sedang dikunjungi dari oleh algoritma akan menjadi *parent node*. Dalam setiap pembangkitan anaknya dilakukan secara *bruteforce* untuk mencari kata-kata yang *valid* dalam kamus untuk dijadikan sebagai *node*. Jika *node* anak tersebut tidak pernah dikunjungi atau pernah dikunjungi namun dalam kedalaman yang

lebih dalam, maka *node* anak tersebut dimasukkan ke *priority queue* karena layak untuk dijadikan pertimbangan solusi. Setelah itu, maka algoritma akan kembali ke langkah 4.

6. Algoritma akan mengulangi langkah 4-5 hingga *priority queue* kosong atau algoritma menemukan solusi. Jika *priority queue* kosong, artinya tidak ada lintasan atau perjalanan yang mungkin bagi simpul awal untuk mencapai simpul tujuan.

## 2.2. Algoritma *Greedy Best First Search* (GBFS)

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

1. Definisikan terlebih dahulu kata awal (*starting word*) untuk dijadikan sebagai simpul awal pencarian dan kata tujuan (*ending word*) yang dijadikan sebagai simpul akhir atau tujuan pencarian dalam permainan ini.
2. Lakukan pengecekan atau validasi terhadap masukan dari pengguna terhadap *starting word* dan *ending word*, apakah kedua kata memiliki panjang kata yang sama. Jika kedua kata memiliki panjang kata yang berbeda, maka algoritma tidak akan melanjutkan pencarian dan menghentikan langkahnya.
3. Definisikan sebuah *priority queue* yang mengurutkan komponen di dalamnya berdasarkan *estimation cost*  $h(n)$  yang terendah hingga tertinggi (pengurutan berdasarkan *hamming distance* terendah hingga tertinggi). Maka dari itu, pemrosesan selanjutnya akan dilanjutkan berdasarkan *estimation cost heuristic* yang terendah terlebih dahulu sehingga sesuai dengan prinsip dari algoritma GBFS.

- a. Sebagai tambahan informasi bahwa *cost* dalam GBFS dapat dihitung menggunakan  $f(n) = h(n)$ , dengan  $h(n)$  merupakan estimasi *cost* yang diperlukan dari simpul  $n$  untuk menuju ke simpul tujuan, yang pada kasus ini  $h(n)$  dihitung menggunakan heuristik *hamming distance*. Semisal, perjalanan sebuah algoritma dengan kata awal “as” dan kata tujuan “if”, maka  $h(n)$  akan bernilai 2 karena *hamming distance* kedua kata tersebut 2.
- b. Algoritma tidak akan mempertimbangkan faktor lain sama sekali selain *heuristic hamming distance* untuk satu simpul menuju ke simpul lainnya. Oleh karena itu, jika bertemu dengan kasus kata awal “as” yang berkemungkinan mengunjungi simpul “is” (*hamming-distance : 1*), “an” (*hamming-distance : 1*), “ax” (*hamming-distance : 1*), ataupun kata lain. Maka, penentuan pemrosesan yang mana terlebih dahulu akan ditentukan oleh *priority queue* itu sendiri, karena tidak dapat dipilih secara khusus semisal *is* karena sepertinya memiliki probabilitas yang tinggi untuk mencapai kata tujuan. Jadi, algoritma ini benar-benar secara buta mengandalkan *hamming distance* setiap simpul dalam pengambilan keputusannya.
- c. *Priority queue* diinisialisasi dengan satu simpul / *Node* yang akan memicu perulangan yang terjadi pada langkah 4, yaitu dengan inisialisasi pemasukan *node* / simpul kata mulai (*starting word*) ke dalam *priority queue*.
4. Selama *priority queue* tidak kosong, algoritma akan selalu mengunjungi simpul dengan *cost* yang paling rendah dan dilakukan pengecekan terhadap *node* tersebut apakah merupakan *goal node* atau bukan.
  - a. Jika *node* sekarang merupakan *goal node*, maka algoritma akan berhenti dan mengembalikan simpul-simpul yang dilalui oleh algoritma untuk mencapai *goal node* tersebut.
  - b. Jika *node* bukan merupakan *goal node*, maka algoritma akan dilanjutkan ke langkah selanjutnya.
5. Lakukan pembangkitan *node* anak dari *node* yang sedang dikunjungi oleh algoritma. Maka dari itu, *node* yang sekarang sedang dikunjungi oleh algoritma akan menjadi *parent node*. Dalam setiap pembangkitan anaknya dilakukan secara *bruteforce* untuk mencari kata-kata yang *valid* dalam kamus untuk dijadikan sebagai *node*. Jika *node* anak



tersebut tidak pernah dikunjungi atau pernah dikunjungi namun dalam kedalaman yang lebih dalam, maka *node* anak tersebut dimasukkan ke *priority queue* karena layak untuk dijadikan pertimbangan solusi. Setelah itu, maka algoritma akan kembali ke langkah 4.

6. Algoritma akan mengulangi langkah 4-5 hingga *priority queue* kosong atau algoritma menemukan solusi. Jika *priority queue* kosong, artinya tidak ada lintasan atau perjalanan yang mungkin bagi simpul awal untuk mencapai simpul tujuan.

### 2.3. Algoritma *A-Star* ( $A^*$ )

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

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

tersebut ditemukan. Semisal, sebuah kata “fly” ditemukan pada simpul berkedalaman 1, maka  $g(n) = 1$  dan ingin menuju ke kata tujuan “fit”, maka *heuristic cost*  $h(n) = 2$  dengan perhitungan *hamming distance*. Jadi, total *cost*  $f(n)$  pada simpul kata “fly” adalah  $g(n) + h(n) = 1 + 2 = 3$ .

- b. Dengan konsep seperti ini, secara tidak langsung algoritma A\* merupakan sebuah algoritma pencarian yang menggabungkan kedua algoritma UCS dan GBFS menjadi satu, sehingga solusi yang ditemukan akan menjadi optimal / terpendek seperti UCS, namun dengan kunjungan *node* yang lebih sedikit karena dibantu dengan bantuan *heuristic hamming distance* seperti pada algoritma GBFS.
  - c. *Priority queue* diinisialisasi dengan satu simpul / *Node* yang akan memicu perulangan yang terjadi pada langkah 4, yaitu dengan inisialisasi pemasukan *node* / simpul kata mulai (*starting word*) ke dalam *priority queue*.
4. Selama *priority queue* tidak kosong, algoritma akan selalu mengunjungi simpul dengan *cost* yang paling rendah dan dilakukan pengecekan terhadap *node* tersebut apakah merupakan *goal node* atau bukan.
    - a. Jika *node* sekarang merupakan *goal node*, maka algoritma akan berhenti dan mengembalikan simpul-simpul yang dilalui oleh algoritma untuk mencapai *goal node* tersebut.
    - b. Jika *node* bukan merupakan *goal node*, maka algoritma akan dilanjutkan ke langkah selanjutnya.
  5. Lakukan pembangkitan *node* anak dari *node* yang sedang dikunjungi oleh algoritma. Maka dari itu, *node* yang sekarang sedang dikunjungi oleh algoritma akan menjadi *parent node*. Dalam setiap pembangkitan anaknya dilakukan secara *bruteforce* untuk mencari kata-kata yang *valid* dalam kamus untuk dijadikan sebagai *node*. Jika *node* anak tersebut tidak pernah dikunjungi atau pernah dikunjungi namun dalam kedalaman yang lebih dalam, maka *node* anak tersebut dimasukkan ke *priority queue* karena layak untuk dijadikan pertimbangan solusi. Setelah itu, maka algoritma akan kembali ke langkah 4.
  6. Algoritma akan mengulangi langkah 4-5 hingga *priority queue* kosong atau algoritma menemukan solusi. Jika *priority queue* kosong, artinya tidak ada lintasan atau perjalanan yang mungkin bagi simpul awal untuk mencapai simpul tujuan.

## 2.4. Analisis Teori

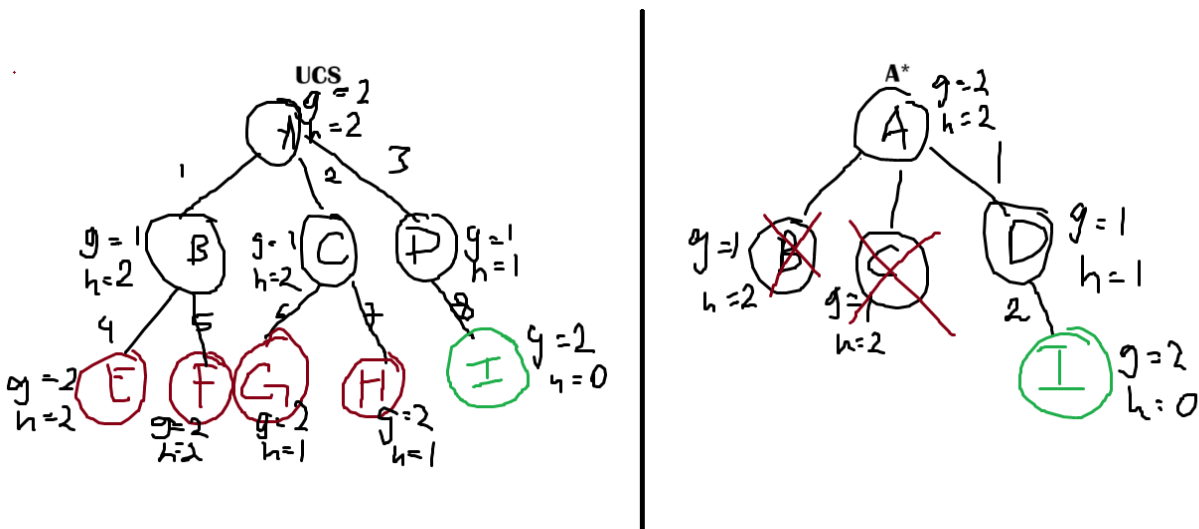
Terlihat bahwa dari implementasi setiap algoritma yang dijelaskan pada bagian-bagian sebelumnya bahwa ketiga algoritma yang di-implementasi tentu saja memiliki perbedaan atau cirinya masing-masing yang menyebabkan ketiganya merupakan algoritma yang berbeda. Sistem iterasi yang dilakukan untuk ketiga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

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

Jika dilakukan perbandingan antara algoritma UCS dan GBFS, yang jika dilihat pada pembahasan implementasi pada bagian sebelumnya menunjukkan bahwa langkah yang berbeda hanya terjadi pada langkah nomor 3. Ini menandakan bahwa kedua algoritma hampir memiliki mekanisme yang mirip, namun hanya sedikit perbedaan. Cara kedua algoritma tersebut membangkitkan anaknya cenderung sama, namun perbedaan terdapat dalam urutan *node-node* yang akan diproses selanjutnya oleh algoritma dalam *priority queue*. Jika *priority queue* kedua algoritma tersebut berbeda, maka sudah dapat dipastikan urutan pemrosesan serta *path* yang

dihasilkan akan beda pula. Akibat UCS yang memperhatikan dan mementingkan pemrosesan *node* yang memiliki *depth* atau *cost*  $g(n)$  yang rendah, maka akan cenderung menghasilkan solusi *path* yang terpendek atau paling optimal dibandingkan dengan algoritma GBFS yang cenderung langsung mengunjungi sebuah *node* jika memiliki *hamming distance* atau *heuristic cost*  $h(n)$  yang rendah tanpa mempertimbangkan kedalaman solusi. Jadi, *path* yang dihasilkan kedua algoritma tersebut dipastikan berbeda dan algoritma UCS akan menghasilkan *path* yang lebih singkat dibandingkan dengan algoritma GBFS.

Secara teoritis, algoritma A\* akan lebih efisien dibandingkan dengan algoritma UCS pada kasus *Word Ladder* ini, yang dapat dilihat pada visualisasi salah satu kasus yang membuktikan bahwa algoritma A\* lebih unggul dibandingkan dengan algoritma UCS pada lampiran **Gambar 2.4.1**. Terlihat bahwa algoritma UCS membutuhkan sebanyak 8 langkah untuk mencapai simpul I dari A, sedangkan algoritma A\* hanya membutuhkan sebanyak 2 langkah. Ini disebabkan A\* sudah menggunakan konsep seperti UCS yang mempertimbangkan kedalaman, namun A\* juga mempertimbangkan *heuristic cost*-nya sehingga tidak melakukan langkah yang sia-sia untuk mengunjungi *node* yang memiliki *hamming distance* lebih tinggi sehingga pencarian menjadi lebih jauh dari *goal node* yang menyebabkan langkah terbuang secara sia-sia. Dengan mempertimbangkan  $h(n)$ , algoritma A\* berhasil melakukan penghematan langkah yang masif dibandingkan algoritma UCS sehingga algoritma A\* lebih efisien.



**Gambar 2.4.1.** Perbandingan perjalanan antara algoritma UCS dan A\*

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

## BAB III : IMPLEMENTASI PROGRAM DENGAN JAVA

Algoritma *Uniform Cost Search* (UCS), *Greedy Best First Search* (GBFS), dan *A-Star* (A\*) seperti yang telah dijelaskan pada bab sebelumnya dapat diimplementasikan dengan menggunakan bahasa pemrograman Java. *Source code* program yang dibuat oleh penulis dapat diakses pada pranala *repository* yang telah dilampirkan pada lampiran laporan ini. *Source code* telah diimplementasikan secara modular dengan kode algoritma dan *utilities* yang dipakai dan dipisahkan pada *package*-nya masing-masing. Berikut merupakan penjelasan *file-file* yang terdapat pada *repository* berdasarkan dengan penjelasan rinciannya.

### 1. Folder *Utils*

Setiap kelas yang terdapat pada folder yang dipisah pada file masing-masing dibungkus menjadi satu package bernama *utils* sehingga dapat diakses oleh file yang terdapat pada folder atau *path* lainnya. Folder ini berisi kelas dan *method* yang akan digunakan untuk memenuhi kebutuhan dalam pembuatan algoritma serta pemuatan yang dibutuhkan untuk menjalankan program utama.

#### a. File *Dictionary.java*

```
package utils;

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

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

    public static void load_word(String filename) {
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = reader.readLine()) != null) {
                words.add(line.trim());
            }
        } catch (IOException e) {
            System.out.println("Failed to load dictionary: " + e.getMessage());
            e.printStackTrace();
        }
    }

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

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

**Gambar 3.1.** Cuplikan Kode Kelas *Dictionary*

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

**b. File *Utils.java***

```
package utils;

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

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

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

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

**Gambar 3.2.** Cuplikan kode *Interface Utils*

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

## 2. Folder *algorithm*

Folder *algorithm* berisi file-file yang mengandung kelas yang dibuat untuk menampung logika algoritma yang dibuat untuk menyelesaikan persoalan permainan ini. Pada folder ini terdapat beberapa file berikut.

### a. File *WordLadder.java*

```
package algorithm;

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

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

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

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

    public long getExecutionTime() {
        return executionTime;
    }

    public int getNodesVisited() {
        return nodesVisited;
    }

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

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



### **Gambar 3.3.** Cuplikan Kode Kelas *WordLadder*

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

#### **b. File *WordLadderAStar.java***

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

```

package algoritme;

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

public class WordLadderAStar extends WordLadder implements Utils {

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

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

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

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

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

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

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

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

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

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

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

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

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

```

**Gambar 3.4.** Cuplikan Kode Kelas *WordLadderAStar*

c. File *WordLadderUCS.java*

```
package algorithm;

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

public class WordLadderUCS extends WordLadder implements Utils {

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

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

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

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

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

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

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

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

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

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

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

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

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

**Gambar 3.5.** Cuplikan Kode Kelas *WordLadderUCS*

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

**d. File *WordLadderGBFS.java***

Kelas *WordLadderGBFS* adalah kelas yang dibuat dengan tujuan untuk menampung *solver* untuk menyelesaikan persoalan permainan *WordLadder* dengan menggunakan algoritma *Greedy Best First Search* (GBFS). Kelas ini merupakan turunan dari kelas *WordLadder* dan mengimplementasikan *interface Utils*. Kelas ini menggunakan konstruktor *superclass*-nya, memiliki metode yang hampir sama dengan kelas *WordLadder* algoritma lainnya, dan memiliki kelas *Node* yang sama dengan kelas *WordLadderUCS*. Inti utama dari kelas ini yang menyimpan otak dari *solver* dengan menggunakan algoritma GBFS terdapat pada metode *find\_path\_solution\_GBFS*.

```

package algoritme;

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

public class WordLadderGBFS extends WordLadder implements Utils {

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

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

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

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

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

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

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

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

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

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

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

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

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

```

**Gambar 3.5.** Cuplikan Kode Kelas *WordLadderGBFS*

### 3. File Main.java

```
boolean found = false;
switch (choice) {
    case 1:
        WordLadderUCS solverUCS = new WordLadderUCS(new ArrayList<>(), executionTime:0, nodesVisited:0);
        solverUCS.find_path_solution_UCS(starting_word, ending_word);
        if (!solverUCS.getPath().isEmpty()) {
            print_information(solverUCS.getPath(), solverUCS.getExecutionTime(), solverUCS.getNodesVisited());
            found = true;
        }
        break;
    case 2:
        WordLadderGBFS solverGBFS = new WordLadderGBFS(new ArrayList<>(), executionTime:0, nodesVisited:0);
        solverGBFS.find_path_solution_GBFS(starting_word, ending_word);
        if (!solverGBFS.getPath().isEmpty()) {
            print_information(solverGBFS.getPath(), solverGBFS.getExecutionTime(), solverGBFS.getNodesVisited());
            found = true;
        }
        break;
    case 3:
        WordLadderAStar solverAStar = new WordLadderAStar(new ArrayList<>(), executionTime:0, nodesVisited:0);
        solverAStar.find_path_solution_AStar(starting_word, ending_word);
        if (!solverAStar.getPath().isEmpty()) {
            print_information(solverAStar.getPath(), solverAStar.getExecutionTime(), solverAStar.getNodesVisited());
            found = true;
        }
        break;
}

if (!found) {
    System.out.println(x:"Path not found. So sadd :(");
}
```

**Gambar 3.6.** Cuplikan Kode Program Utama CLI Main.java

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

### 4. File WordLadderGUI.java (Bonus)

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

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

```
public WordLadderGUI() {
    setTitle(title:"Word Ladder Game");
    setSize(width:500, height:400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLocationRelativeTo(c:null);
    setLayout(new BorderLayout());

    JPanel formPanel = new JPanel(new GridBagLayout());
    formPanel.setBackground(new Color(r:230, g:240, b:255));
    GridBagConstraints c = new GridBagConstraints();
    c.insets = new Insets(top:10, left:10, bottom:10, right:10);

    JLabel startingWordLabel = new JLabel(text:"Starting Word:");
    startingWordLabel.setForeground(new Color(r:50, g:50, b:150));
    c.gridx = 0; c.gridy = 0;
    formPanel.add(startingWordLabel, c);

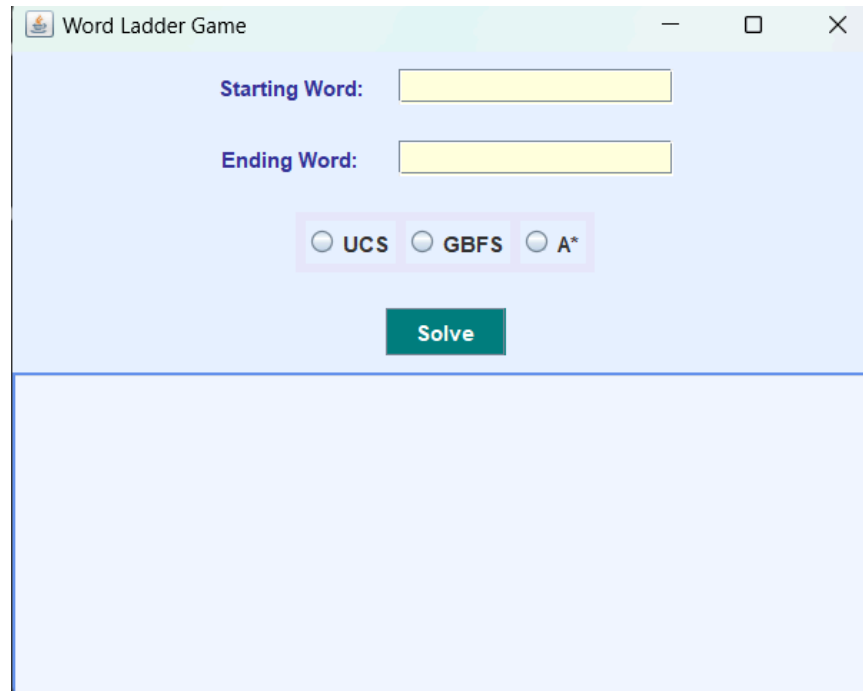
    startingWordField = new JTextField(columns:15);
    startingWordField.setForeground(new Color(r:0, g:100, b:0));
    startingWordField.setBackground(new Color(r:255, g:255, b:224));
    c.gridx = 1;
    formPanel.add(startingWordField, c);

    JLabel endingWordLabel = new JLabel(text:"Ending Word:");
    endingWordLabel.setForeground(new Color(r:50, g:50, b:150));
    c.gridx = 0; c.gridy = 1;
    formPanel.add(endingWordLabel, c);

    endingWordField = new JTextField(columns:15);
    endingWordField.setForeground(new Color(r:0, g:100, b:0));
    endingWordField.setBackground(new Color(r:255, g:255, b:224));
    c.gridx = 1;
    formPanel.add(endingWordField, c);

    JRadioButton1 = new JRadioButton(text:"UCS");
    JRadioButton2 = new JRadioButton(text:"GBFS");
    JRadioButton3 = new JRadioButton(text:"A*");
    JRadioButton1.setBackground(formPanel.getBackground());
    JRadioButton2.setBackground(formPanel.getBackground());
    JRadioButton3.setBackground(formPanel.getBackground());
    algorithmGroup = new ButtonGroup();
    algorithmGroup.add(JRadioButton1);
    algorithmGroup.add(JRadioButton2);
```

**Gambar 3.7.** Cuplikan Kode Program Utama GUI WordLadderGUI.java



**Gambar 3.8.** Tampilan GUI setelah *WordLadderGUI* dijalankan

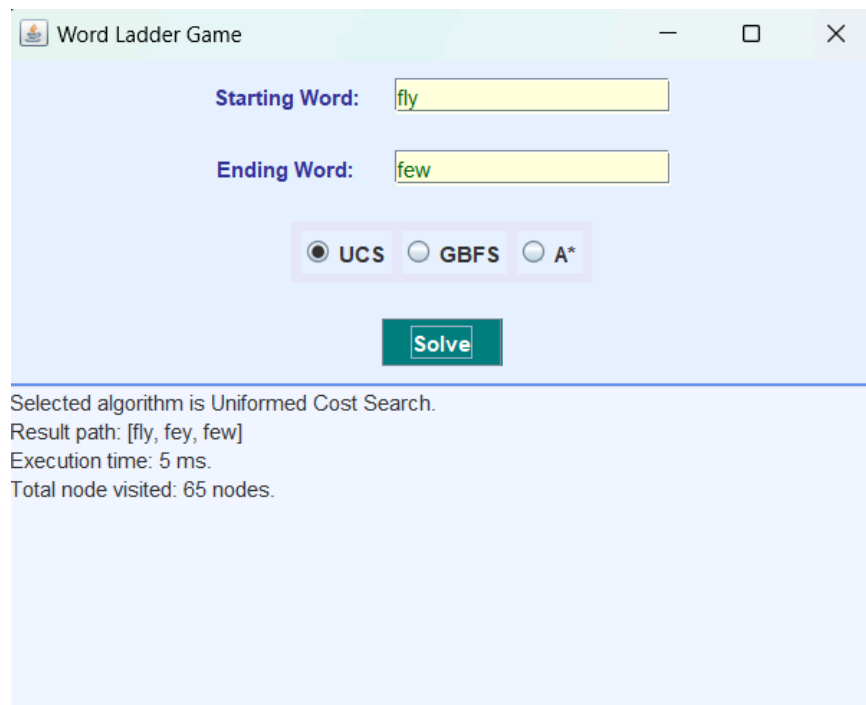


## BAB IV : EKSPERIMEN

Untuk setiap percobaan yang dilakukan, dibutuhkan informasi terkait *starting word*, *ending word*, dan juga algoritma yang dipilih oleh pengguna untuk menentukan jenis algoritma yang digunakan untuk menyelesaikan permainan *Word Ladder* ini. Untuk menguji keberhasilan program yang dibuat oleh penulis dan untuk dijadikan sebagai bahan analisis untuk bab selanjutnya, maka pengujian dan eksperimen akan dilampirkan sebagai berikut.

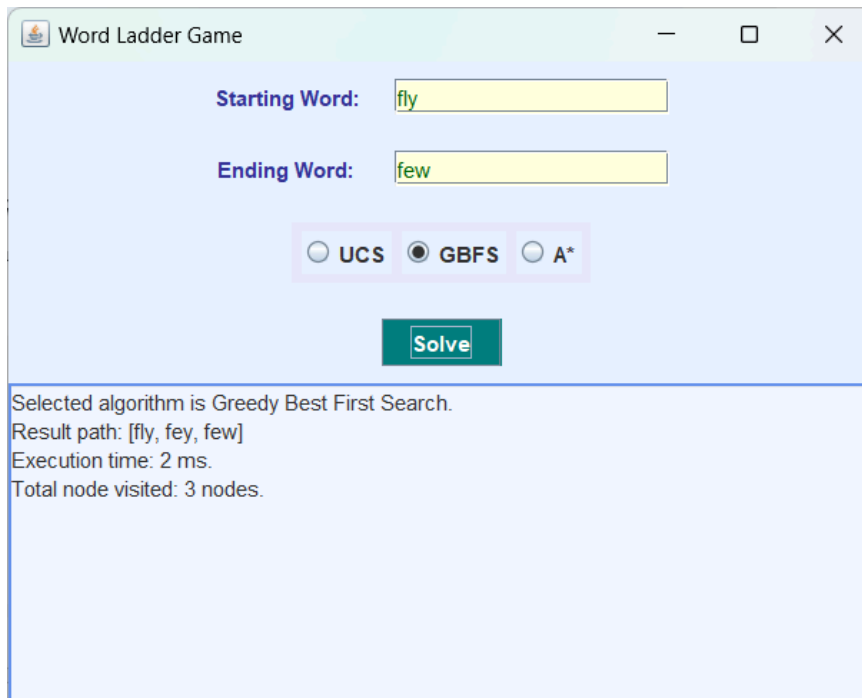
### 4.1. Percobaan 1

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



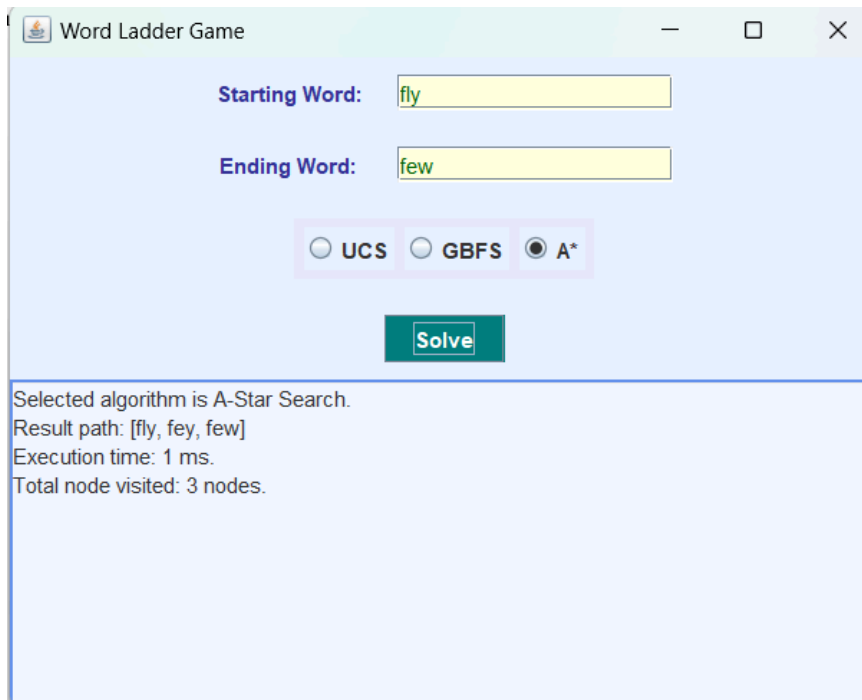
**Gambar 4.1.1.** Percobaan 1 dengan Algoritma UCS

#### 4.1.2. Algoritma *Greedy Best First Search* (GBFS)



**Gambar 4.1.2.** Percobaan 1 dengan Algoritma GBFS

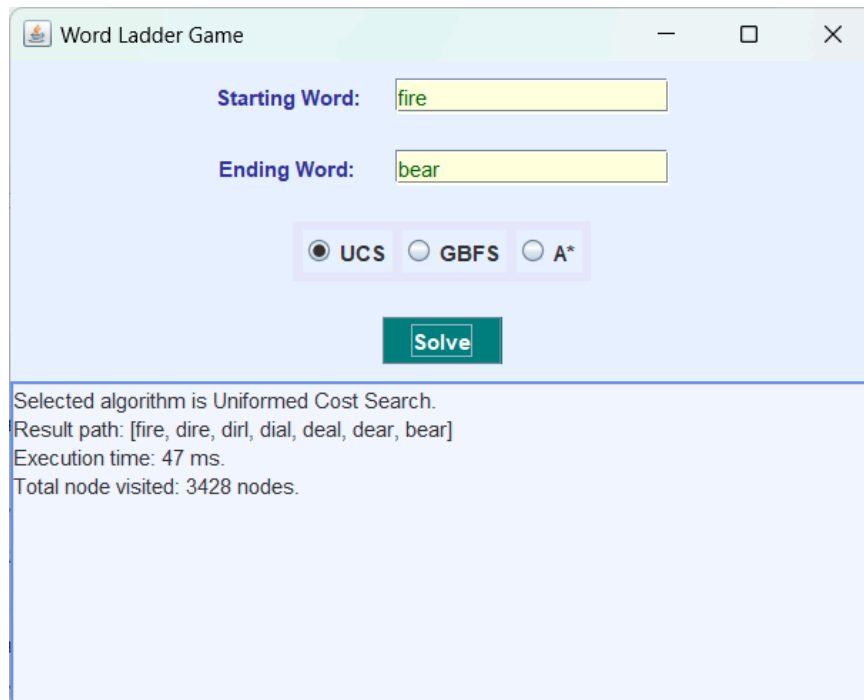
#### 4.1.3. Algoritma *A-Star* (A\*)



**Gambar 4.1.3.** Percobaan 1 dengan Algoritma A\*

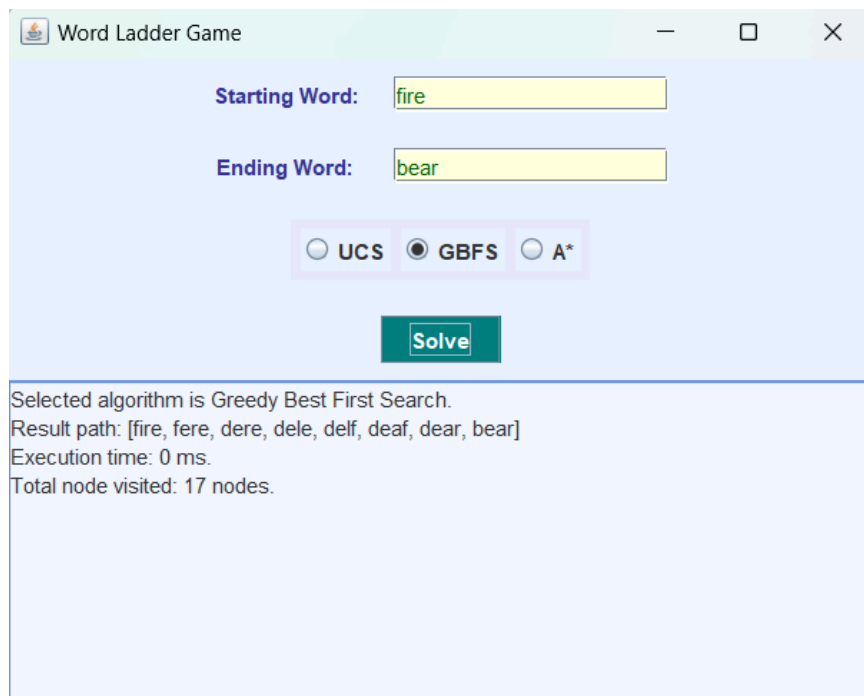
## 4.2. Percobaan 2

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



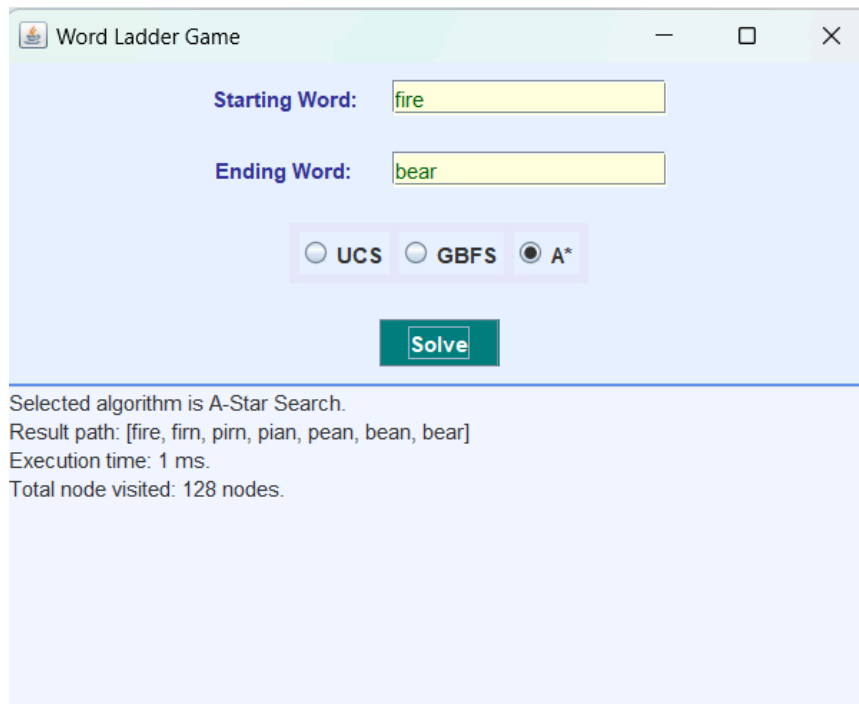
**Gambar 4.2.1.** Percobaan 2 dengan Algoritma UCS

### 4.2.2. Algoritma *Greedy Best First Search* (GBFS)



**Gambar 4.2.2.** Percobaan 2 dengan Algoritma GBFS

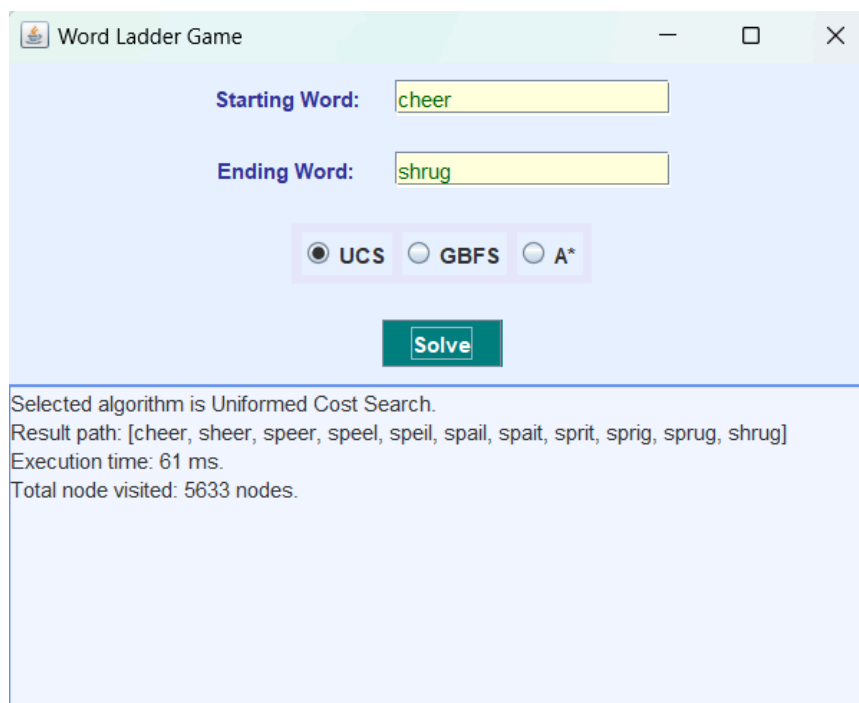
#### 4.2.3. Algoritma *A-Star* (A\*)



**Gambar 4.2.3.** Percobaan 2 dengan Algoritma A\*

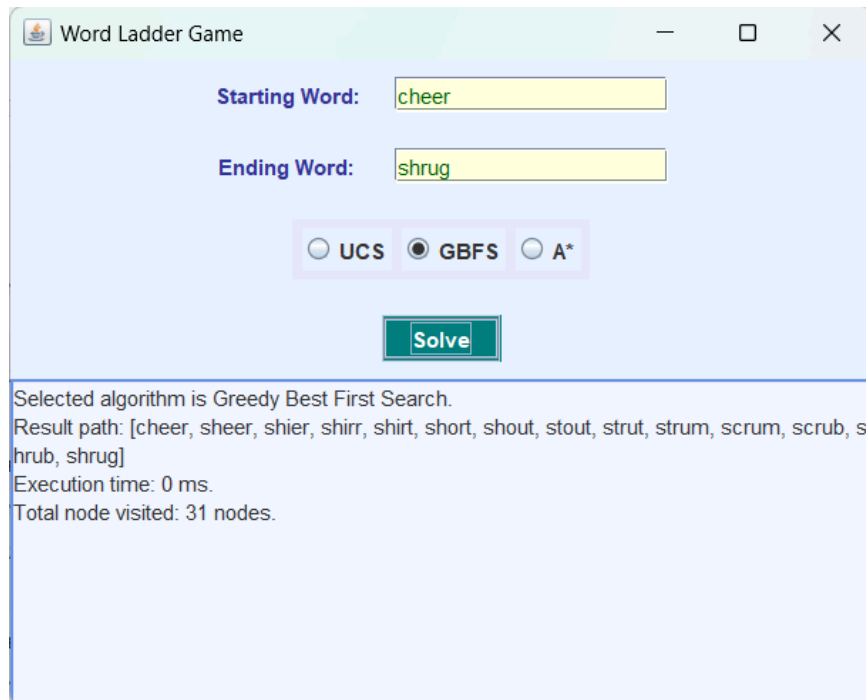
#### 4.3. Percobaan 3

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



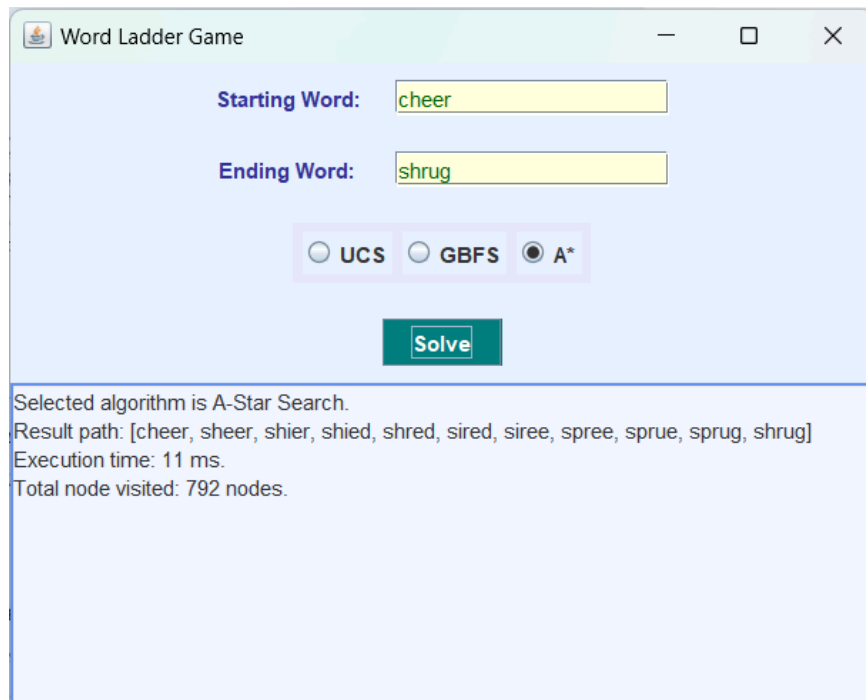
**Gambar 4.3.1.** Percobaan 3 dengan Algoritma UCS

#### 4.3.2. Algoritma *Greedy Best First Search* (GBFS)



**Gambar 4.3.2.** Percobaan 3 dengan Algoritma GBFS

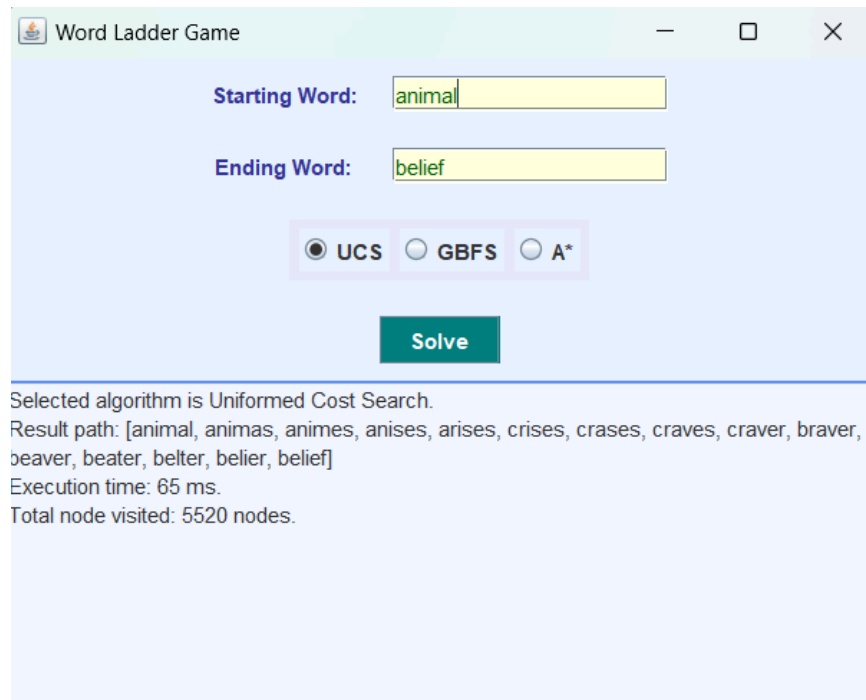
#### 4.3.3. Algoritma *A-Star* (A\*)



**Gambar 4.3.3.** Percobaan 3 dengan Algoritma A\*

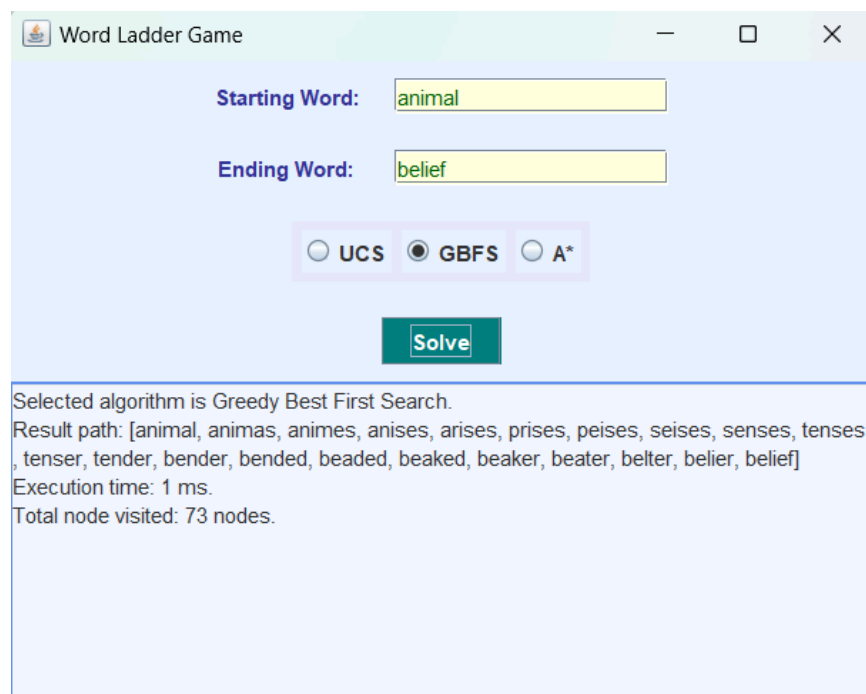
#### 4.4. Percobaan 4

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



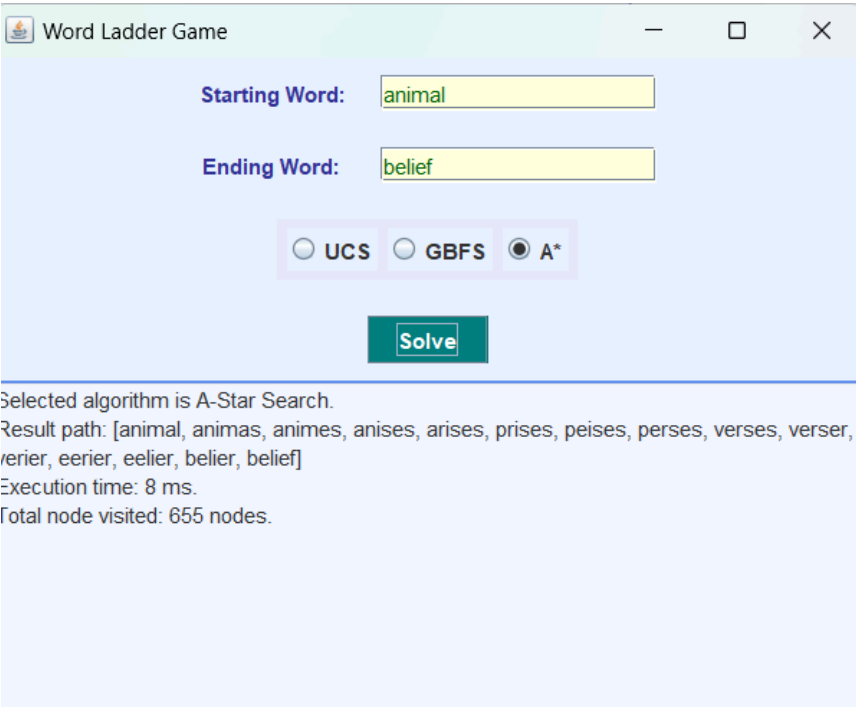
**Gambar 4.4.1.** Percobaan 4 dengan Algoritma UCS

##### 4.4.2. Algoritma *Greedy Best First Search* (GBFS)



**Gambar 4.4.2.** Percobaan 4 dengan Algoritma GBFS

#### 4.4.3. Algoritma *A-Star* (A\*)



Word Ladder Game

Starting Word: animal

Ending Word: belief

☐ UCS ☐ GBFS ☒ A\*

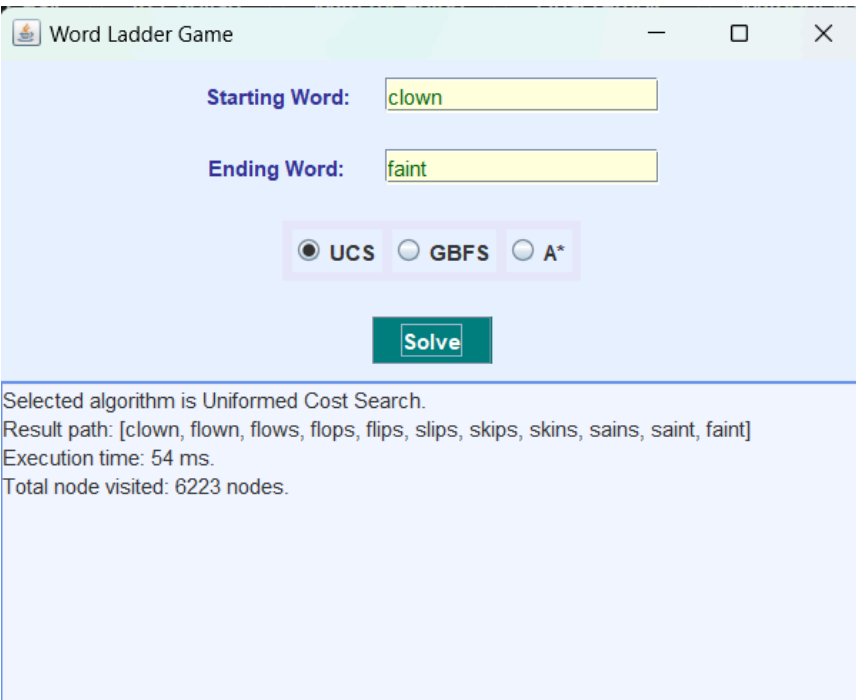
**Solve**

Selected algorithm is A-Star Search.  
Result path: [animal, animas, animes, anises, arises, prises, peises, perses, verses, verser, rerier, eerier, eelier, belief, belief]  
Execution time: 8 ms.  
Total node visited: 655 nodes.

**Gambar 4.4.3.** Percobaan 4 dengan Algoritma A\*

#### 4.5. Percobaan 5

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



Word Ladder Game

Starting Word: clown

Ending Word: faint

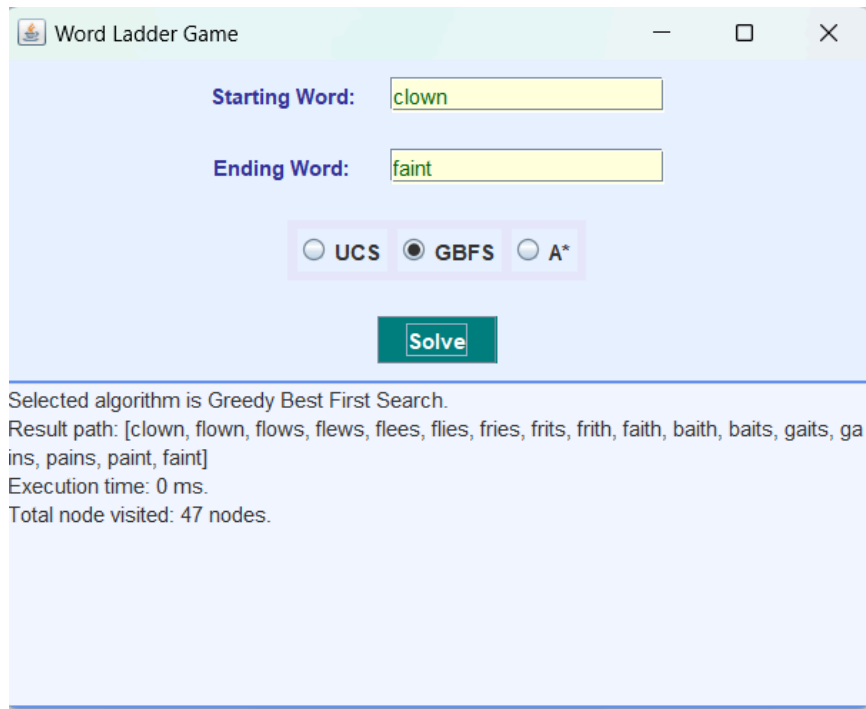
☒ UCS ☐ GBFS ☐ A\*

**Solve**

Selected algorithm is Uniformed Cost Search.  
Result path: [clown, flown, flows, flops, flips, slips, skips, skins, sains, saint, faint]  
Execution time: 54 ms.  
Total node visited: 6223 nodes.

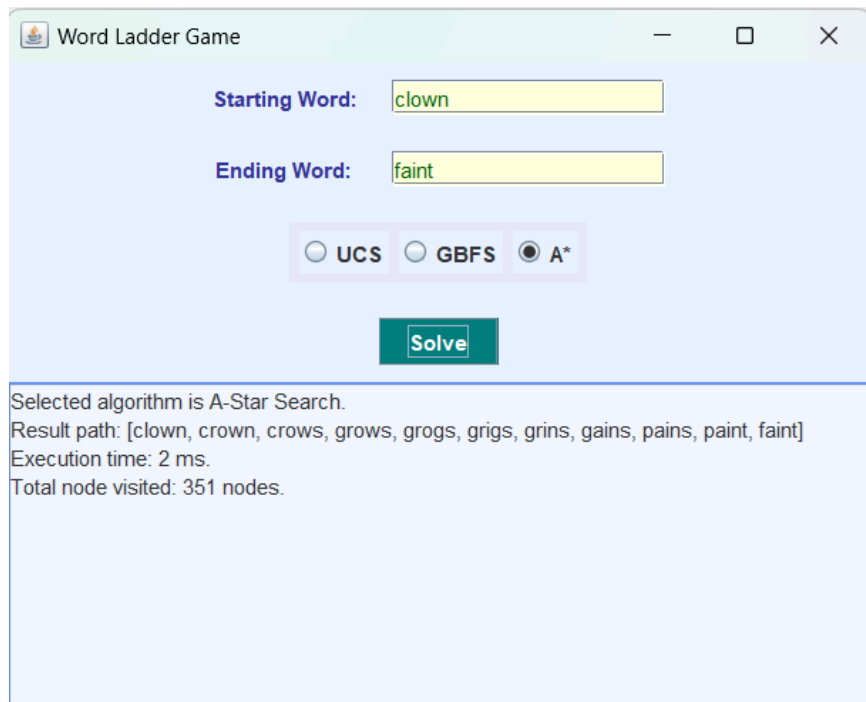
**Gambar 4.5.1.** Percobaan 5 dengan Algoritma UCS

#### 4.5.2. Algoritma *Greedy Best First Search* (GBFS)



**Gambar 4.5.2.** Percobaan 5 dengan Algoritma GBFS

#### 4.5.3. Algoritma *A-Star* (A\*)

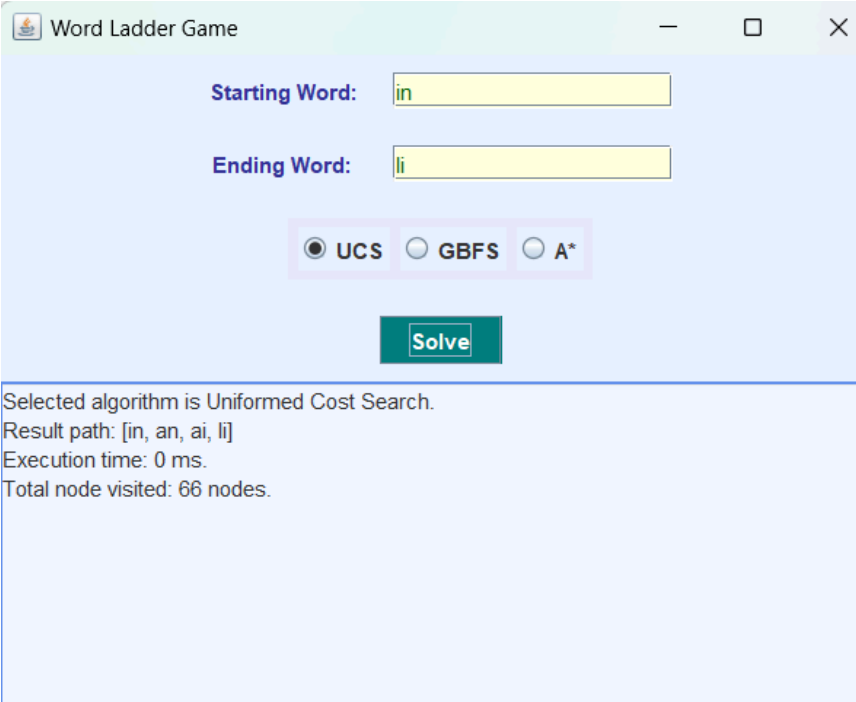


**Gambar 4.5.3.** Percobaan 5 dengan Algoritma A\*



## 4.6. Percobaan 6

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



Word Ladder Game

Starting Word:

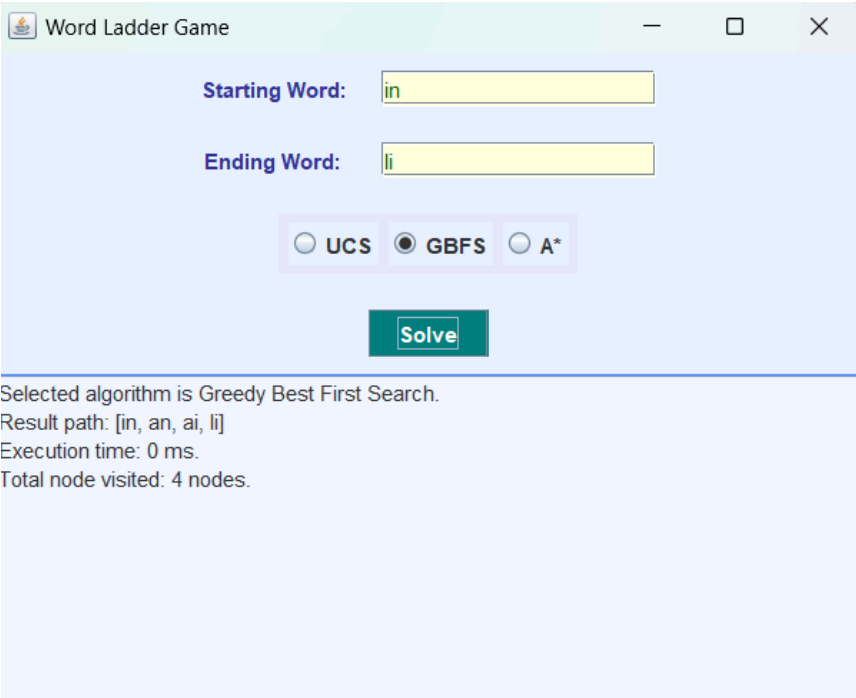
Ending Word:

☒ UCS ☐ GBFS ☐ A\*

Selected algorithm is Uniformed Cost Search.  
Result path: [in, an, ai, li]  
Execution time: 0 ms.  
Total node visited: 66 nodes.

**Gambar 4.6.1.** Percobaan 6 dengan Algoritma UCS

### 4.6.2. Algoritma *Greedy Best First Search* (GBFS)



Word Ladder Game

Starting Word:

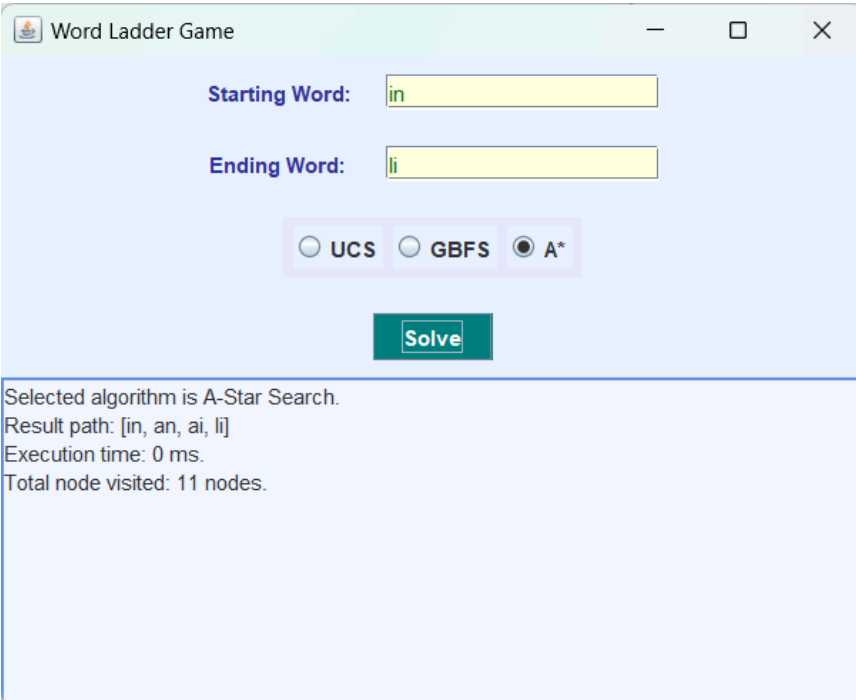
Ending Word:

☐ UCS ☒ GBFS ☐ A\*

Selected algorithm is Greedy Best First Search.  
Result path: [in, an, ai, li]  
Execution time: 0 ms.  
Total node visited: 4 nodes.

**Gambar 4.6.2.** Percobaan 6 dengan Algoritma GBFS

#### 4.6.3. Algoritma *A-Star* (A\*)



The screenshot shows a window titled "Word Ladder Game". It has two input fields: "Starting Word:" with the value "in" and "Ending Word:" with the value "li". Below these are three radio buttons for selecting an algorithm: "UCS", "GBFS", and "A\*". The "A\*" radio button is selected. A green "Solve" button is positioned below the radio buttons. The bottom half of the window displays the results of the search:

Selected algorithm is A-Star Search.  
Result path: [in, an, ai, li]  
Execution time: 0 ms.  
Total node visited: 11 nodes.

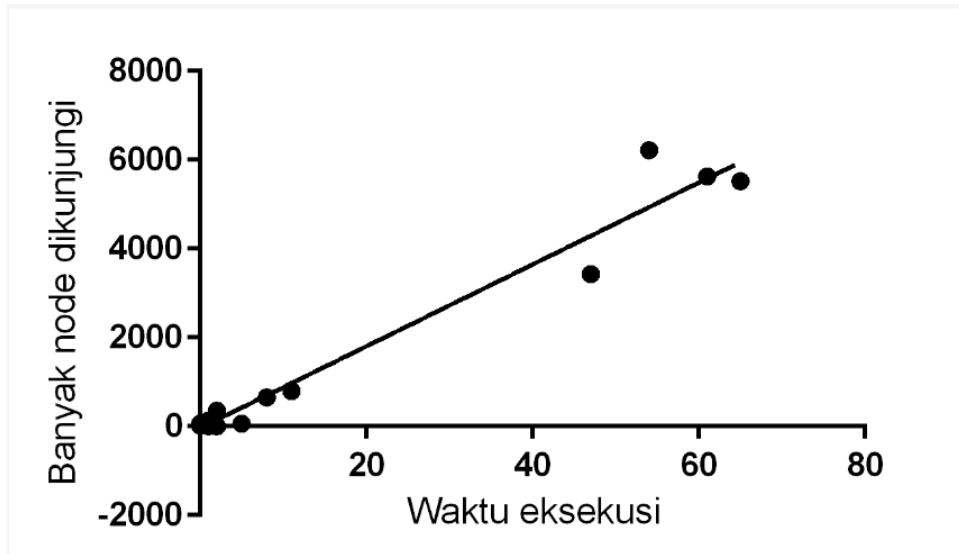
**Gambar 4.6.3.** Percobaan 6 dengan Algoritma A\*

## BAB V : ANALISIS

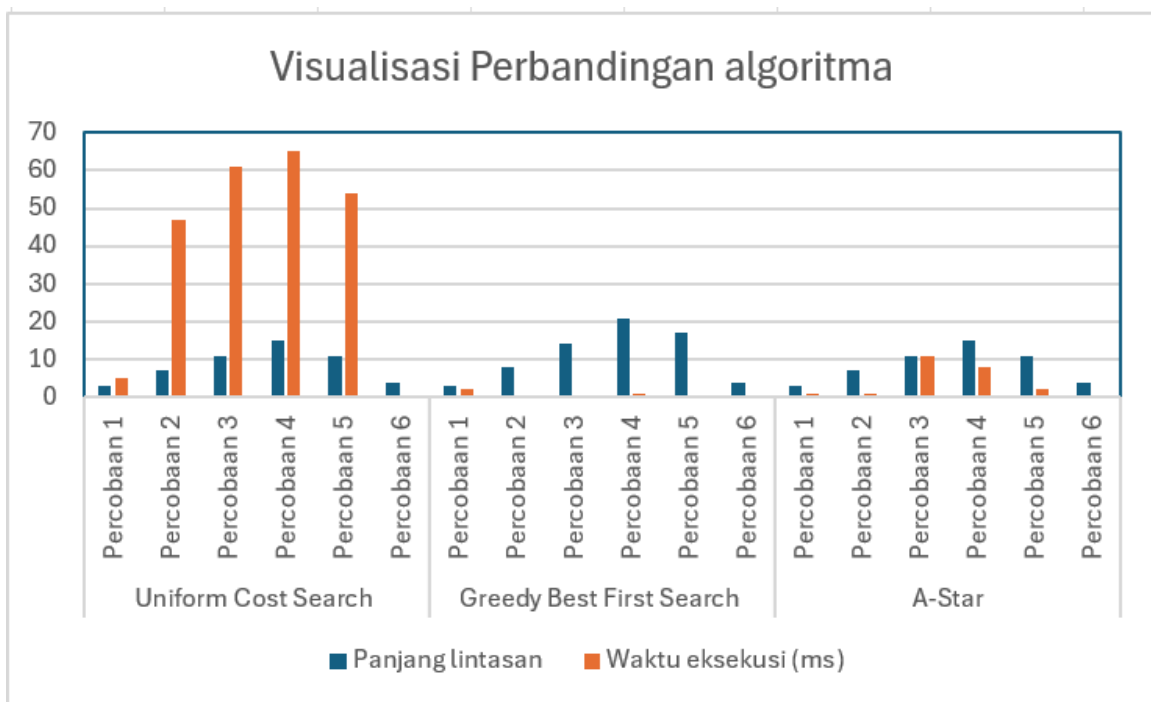
Dari hasil eksperimen, beberapa data diperoleh dari pengujian yang dilakukan. Data hasilnya disajikan dalam bentuk tabel dan divisualisasikan dalam gambar seperti berikut.

Jenis Algoritma	Nomor Percobaan	Panjang lintasan	Waktu eksekusi (ms)	Banyak <i>node</i> dikunjungi (node)
<i>Uniform Cost Search</i> (UCS)	Percobaan 1	3	5	65
	Percobaan 2	7	47	3428
	Percobaan 3	11	61	5633
	Percobaan 4	15	65	5520
	Percobaan 5	11	54	6223
	Percobaan 6	4	0	66
<i>Greedy Best First Search</i> (GBFS)	Percobaan 1	3	2	3
	Percobaan 2	8	0	17
	Percobaan 3	14	0	31
	Percobaan 4	21	1	73
	Percobaan 5	17	0	47
	Percobaan 6	4	0	4
<i>A-Star</i> (A*)	Percobaan 1	3	1	3
	Percobaan 2	7	1	128
	Percobaan 3	11	11	792
	Percobaan 4	15	8	655
	Percobaan 5	11	2	351
	Percobaan 6	4	0	11

**Tabel 5.1.** Tabulasi data hasil eksperimen



**Gambar 5.1.** Visualisasi Korelasi Waktu Eksekusi dan Banyak Node dikunjungi dengan nilai 0.92

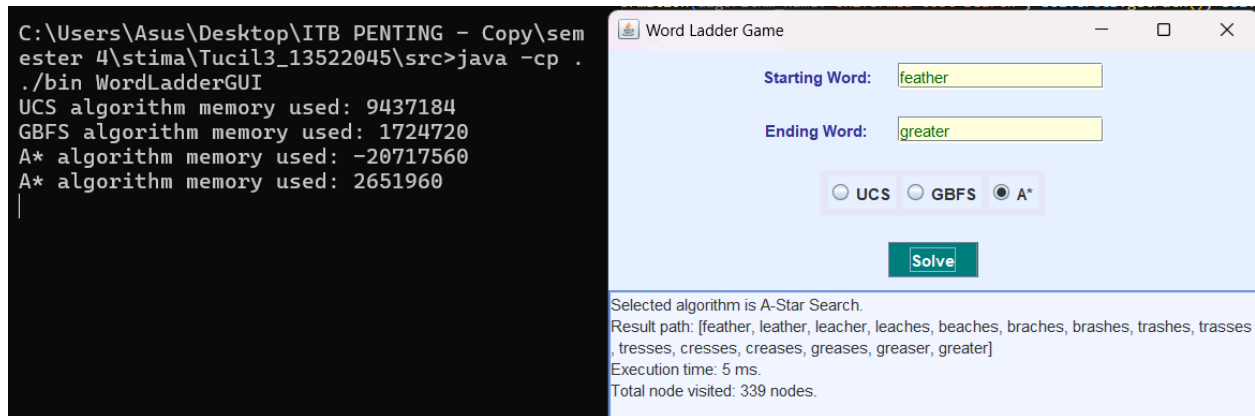


**Gambar 5.2.** Visualisasi perbandingan algoritma terhadap atribut informasi panjang lintasan dan waktu eksekusi

Dari segi optimalitas, dari hasil visualisasi data yang telah ditampilkan, terlihat jelas bahwa algoritma yang paling optimal dalam kasus penyelesaian permainan *WordLadder* ini adalah algoritma A\* dengan optimal yang dimaksud berarti menghasilkan solusi yang paling optimal (perjalanan yang paling singkat) dan waktu eksekusi yang cepat pula. Dari hasil visualisasi perbandingan ketiga algoritma pada **Gambar 5.2.**, terdapat informasi bahwa algoritma *Uniform Cost Search* memiliki waktu eksekusi yang paling lama dibandingkan dengan ketiga algoritma, namun memiliki solusi yang optimal. Sedangkan, untuk algoritma *Greedy Best First Search*, algoritma cenderung memiliki waktu eksekusi yang paling cepat dibandingkan ketiga algoritma yang dibandingkan, namun memiliki solusi yang kurang optimal (panjang lintasan yang ditempuh cenderung lebih banyak dibandingkan algoritma lainnya). Terakhir, algoritma *A-Star* yang memiliki solusi yang optimal dan memiliki waktu eksekusi yang berada di rata-rata perbandingan ketiga algoritma tersebut. Oleh karena itu, karena tujuan permainan *WordLadder* adalah mencari lintasan yang optimal, maka algoritma A\* merupakan algoritma yang paling baik dan juga menyediakan waktu eksekusi yang relatif cepat.

Sebagai tambahan informasi dari hasil visualisasi **Gambar 5.1.** bahwa informasi yang diperoleh adalah tingginya korelasi antara waktu eksekusi dengan banyaknya *node* yang dikunjungi oleh sebuah algoritma. Akibat korelasi yang tinggi, maka kedua variabel tersebut bisa dikatakan saling terikat sehingga akan cenderung searah perkembangannya.

Dari segi waktu eksekusi, sudah terlihat jelas pada hasil visualisasi, bahwa algoritma *Greedy Best First Search* mengungguli secara jauh dibandingkan algoritma lainnya. Ini disebabkan *node* yang dikunjungi oleh algoritma tersebut sedikit dibandingkan algoritma lainnya. Alasan mengapa *node* yang dikunjungi oleh algoritma GBFS cenderung sedikit dikarenakan algoritma ini hanya mempertimbangkan *heuristic estimation cost*, sehingga tidak mempertimbangkan *node-node* lain yang mungkin memiliki solusi yang lebih optimal. Jadi, algoritma GBFS memiliki kelebihan dalam segi waktu eksekusi, namun kurang dalam mengembalikan solusi yang optimal (berupa lintasan yang paling singkat).



**Gambar 5.3.** Visualisasi perbandingan pemakaian memori untuk melakukan eksekusi masing-masing algoritma untuk percobaan *feather* ke *greater*

Dari **Gambar 5.3.** , dapat dilihat bahwa yang mengonsumsi memori terbesar hingga terkecil untuk melakukan eksekusi adalah algoritma UCS, algoritma A\*, dan algoritma GBFS. Semua data berkorelasi dimulai dari waktu eksekusi, banyak *node* yang dikunjungi, dan juga memori yang dipakai. Ketiga atribut informasi tersebut menunjukkan keterhubungan korelasi yang positif sehingga dapat disimpulkan pula bahwa memori yang semakin besar dikonsumsi disebabkan pula akibat banyaknya *nodes* yang pernah dikunjungi oleh algoritma tersebut. Tingginya memori pada algoritma UCS ini disebabkan akibat pembangkitan *node* anak yang dilakukan secara terus menerus secara bertingkat secara *depth*, sehingga sifat dari UCS hampir sama dengan algoritma BFS, yang mana akan mengalami ekspansi memori yang menyebabkan ruang kompleksitasnya berkembang secara eksponensial. Berbeda dengan algoritma A\*, yang mungkin juga melakukan pembangkitan anak secara bertingkat terhadap seluruh *node* pada setiap tingkatannya, namun algoritma ini tetap melakukan seleksi terhadap *node* yang dibandingkan berdasarkan *heuristic estimation cost*-nya sehingga tidak semua *node* dibangkitkan seperti UCS, namun masih bisa mendapatkan solusi optimal. Sedangkan, untuk algoritma GBFS, yang tidak memedulikan kedalaman pencariannya dan hanya berusaha untuk memenuhi keperluan *heuristic*-nya, maka sifatnya menjadi pencarian yang bersifat vertikal (seperti DFS) sehingga algoritma ini akan memiliki ruang kompleksitas yang lebih kecil dibandingkan kedua algoritma lainnya.

# LAMPIRAN

## Github Repository

[https://github.com/ChaiGans/Tucil3\\_13522045](https://github.com/ChaiGans/Tucil3_13522045)

## Tabel Spesifikasi

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