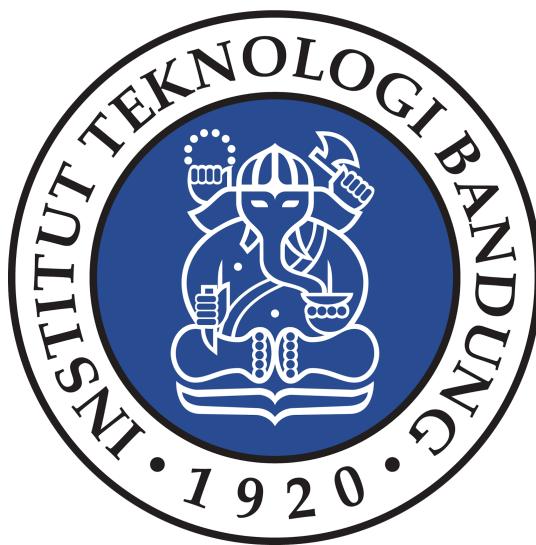


LAPORAN TUGAS KECIL III
IF2211 STRATEGI ALGORITMA
Penyelesaian Permainan Word Ladder Menggunakan
Algoritma UCS, Greedy Best First Search, dan A*



Disusun oleh :
Christian Justin Hendrawan (13522135)

Program Studi Teknik Informatika
Institut Teknologi Bandung
2024

Daftar Isi

Daftar Isi.....	2
Daftar Gambar.....	3
BAB I.....	4
I. Tujuan.....	4
II. Spesifikasi.....	4
BAB II.....	6
2.1. Algoritma A*.....	6
2.2. Algoritma UCS.....	7
2.3. Algoritma Greedy Best First Search.....	8
BAB III.....	10
3.1. Implementasi Algoritma A*.....	10
3.2. Implementasi Algoritma UCS.....	11
3.3. Implementasi Algoritma Greedy Best First Search.....	12
3.4. Analisis Pertanyaan.....	13
BAB IV.....	17
4.1. Souce Code Program Terminal.....	17
4.2. Souce Code Program Bonus GUI.....	27
4.3. Input Output Program.....	34
BAB V.....	44
5.1. Analisis Hasil Optimalitas dan Waktu Eksekusi.....	44
5.2. Analisis Hasil Memori yang Dibutuhkan.....	46
BAB VI.....	49
BAB VII.....	50

Daftar Gambar

Gambar 1.1 Ilustrasi Permainan Word Ladder.....	4
Gambar 2.1 Ilustrasi algoritma A*.....	7

BAB I

Deskripsi Masalah

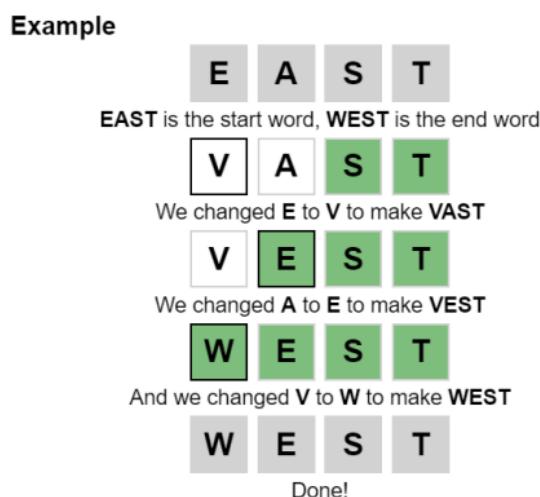
I. Tujuan

Pada tugas kecil ini, kita diminta untuk :

- Membuat program dalam bahasa Java berbasis CLI (Command Line Interface) – bonus jika menggunakan GUI – yang dapat menemukan solusi permainan word ladder menggunakan algoritma **UCS, Greedy Best First Search, dan A***.

II. Spesifikasi

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. 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.



Gambar 1.1 Ilustrasi Permainan Word Ladder

Permainannya cukup sederhana bukan? Jika belum paham dengan peraturan permainannya, cobalah untuk memainkan permainannya pada link sumber di atas. Jika sudah paham dengan permainannya, sekarang adalah waktunya kalian untuk membuat sebuah solver permainan tersebut dengan harapan kita dapat menemukan solusi paling optimal untuk menyelesaikan permainan Word Ladder ini.

BAB II

Teori Singkat

2.1. Algoritma A*

Algoritma A* adalah algoritma pencarian jalur yang digunakan untuk menemukan jalur terpendek antara dua titik dalam graf yang terdiri dari simpul-simpul yang saling terhubung oleh tepian. Ditemukan pada tahun 1968 oleh Peter Hart, Nils Nilsson, dan Bertram Raphael, algoritma ini telah menjadi salah satu algoritma pencarian jalur yang paling populer dan efisien. Algoritma ini digunakan dalam berbagai aplikasi, mulai dari permainan komputer hingga penentuan rute dalam sistem GPS.

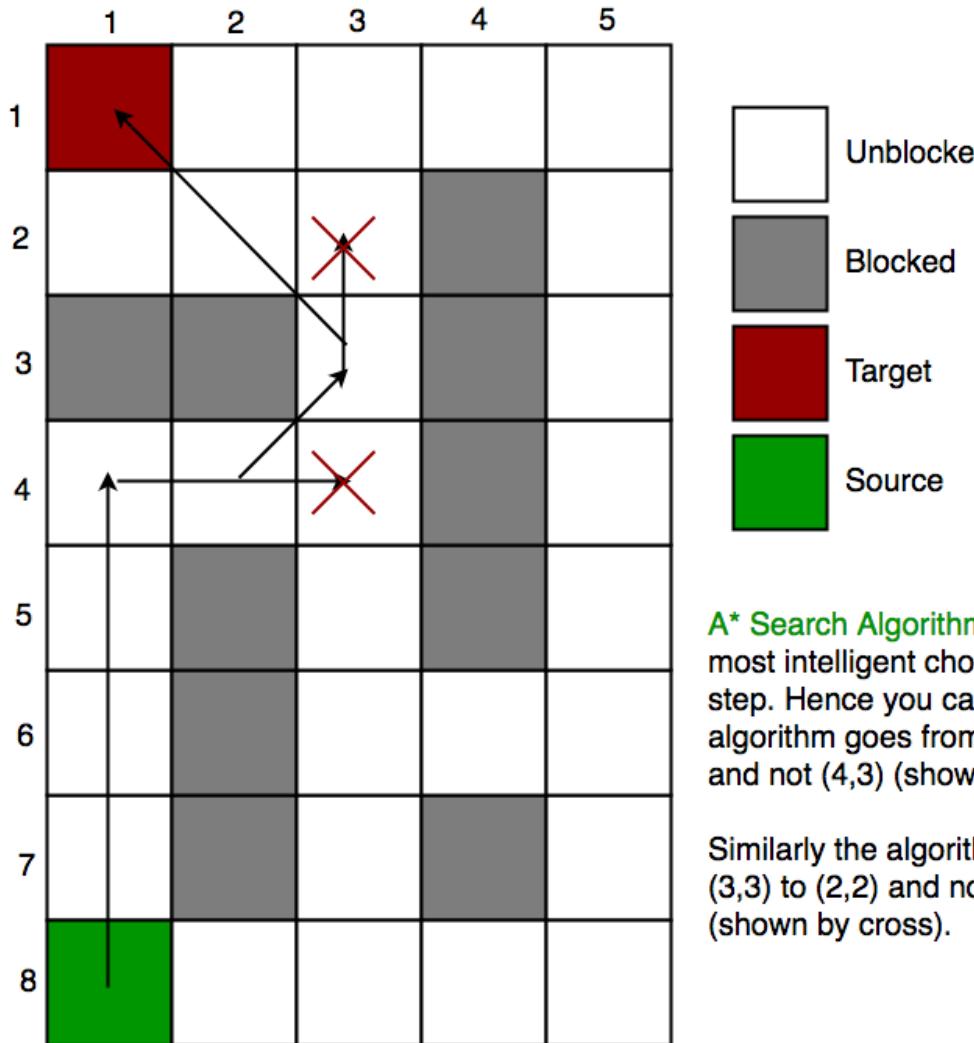
Kecerdasan algoritma ini terletak pada penggunaan pengetahuan tentang tujuan untuk membantu dalam pencarian. Ini dilakukan dengan menggunakan fungsi heuristik, yang memberikan perkiraan biaya dari node saat ini ke tujuan. Dengan cara ini, algoritma A* dapat memprioritaskan node yang tampaknya lebih dekat ke tujuan.

Berikut adalah beberapa konsep penting dalam algoritma A*:

1. **G(n)** adalah biaya aktual dari node awal ke node n. Ini adalah jarak yang sebenarnya yang telah ditempuh.
2. **H(n)** adalah perkiraan biaya dari node n ke tujuan. Ini dihitung menggunakan fungsi heuristik, yang harus memberikan perkiraan yang tidak melebihi biaya sebenarnya (admissible)
3. **F(n)** adalah total perkiraan biaya dari node awal ke tujuan melalui node n. F(n) adalah jumlah dari G(n) dan H(n).

Dalam algoritma A*, kita selalu memilih node dengan nilai F(n) terendah untuk dieksplorasi selanjutnya. Jika ada beberapa node dengan nilai F(n) yang sama, maka kita memilih node dengan nilai H(n) terendah. Ini memastikan bahwa kita selalu memprioritaskan

node yang tampaknya lebih dekat ke tujuan. A* memastikan bahwa solusinya optimal.



A* Search Algorithm makes the most intelligent choice at each step. Hence you can see that algorithm goes from (4,2) to (3,3) and not (4,3) (shown by cross).

Similarly the algorithm goes from (3,3) to (2,2) and not (2,3) (shown by cross).

Gambar 2.1 Ilustrasi algoritma A*

2.2. Algoritma UCS

Algoritma Uniform Cost Search (UCS) adalah algoritma pencarian jalur yang digunakan untuk menemukan jalur terpendek dari satu simpul ke simpul lain dalam graf berbobot. algoritma pencarian jalur ini berjalan dengan cara memilih node berikutnya untuk dikunjungi berdasarkan biaya kumulatif dari node awal ke node saat ini dimana “Node saat ini” merujuk ke node yang sedang diproses atau dieksplorasi oleh algoritma. UCS sendiri adalah algoritma yang tidak

berinformasi, yang berarti tidak menggunakan pengetahuan tentang tujuan dalam proses pencarian.

UCS beroperasi berdasarkan prinsip bahwa node dengan biaya kumulatif terendah selalu dipilih untuk ekspansi. Oleh karena itu, UCS selalu memilih jalur dengan biaya total terendah. Berikut adalah beberapa konsep penting dalam UCS:

1. **G(n)**: Ini adalah biaya aktual dari node awal ke node n. Ini adalah jarak yang sebenarnya yang telah ditempuh.
2. **Priority Queue**: UCS menggunakan struktur data priority queue untuk menyimpan node yang belum dikunjungi. Node dengan biaya kumulatif terendah selalu diambil dari queue terlebih dahulu.

UCS adalah algoritma yang sederhana dan efektif untuk mencari jalur terpendek, tetapi mungkin tidak seefisien algoritma yang berinformasi seperti A* jika pengetahuan tentang tujuan tersedia. UCS selalu menemukan jalur dengan biaya terendah asalkan biaya antara setiap pasangan node adalah positif. Jika ada biaya negatif, UCS mungkin tidak menemukan jalur terpendek.

2.3. Algoritma Greedy Best First Search

Greedy Best First Search atau GBFS adalah algoritma pencarian jalur yang menggunakan prinsip dari algoritma Breadth-First Search (BFS) namun dengan penambahan fungsi heuristik untuk memandu pencarian. Algoritma ini berusaha untuk meminimalkan jarak ke tujuan berdasarkan perkiraan heuristik, bukan jarak yang sebenarnya telah ditempuh.

Berikut adalah beberapa konsep penting dalam GBFS:

1. **Heuristik**: GBFS menggunakan fungsi heuristik untuk memperkirakan biaya terendah dari node saat ini ke tujuan. Fungsi heuristik harus adilatif (tidak pernah surestimasikan biaya) untuk menjamin penemuan jalur terpendek.
2. **H(n)**: Algoritma GBFS menggunakan satu nilai untuk setiap node: H adalah perkiraan biaya dari node saat ini ke tujuan (dihitung menggunakan fungsi heuristik).

3. **Priority Queue:** GBFS menggunakan struktur data priority queue untuk menyimpan node yang belum dikunjungi. Node dengan nilai H terendah selalu diambil dari queue terlebih dahulu.

Meskipun GBFS bisa sangat cepat karena secara agresif mencari tujuan, ia tidak selalu menemukan jalur terpendek karena ia bisa terjebak dalam apa yang disebut "jebakan heuristik", di mana ia mengikuti jalur yang tampak menjanjikan tetapi akhirnya tidak efisien.

BAB III

Analisis Dan Implementasi Program

3.1. Implementasi Algoritma A*

Dalam permainan Word Ladder Solver, algoritma A* digunakan untuk menemukan urutan perpindahan huruf terpendek dari satu kata ke kata lain dengan mengubah satu huruf pada setiap langkah. Penerapan algoritma ini melibatkan beberapa langkah penting. Berikut adalah langkah-langkah penerapan algoritma A* dalam permainan Word Ladder Solver.

1. Pertama, sebuah Priority Queue yang dinamakan *wordQueue* dibuat untuk menyimpan objek *WordNode*. *WordNode* merepresentasikan setiap kata yang akan dieksplorasi dalam pencarian solusi. *wordQueue* ini diurutkan berdasarkan jumlah langkah dan heuristik dari setiap *WordNode*. Nilai heuristik tersebut merupakan perkiraan jarak dari kata saat ini ke kata target, yang dihitung dengan cara menghitung dan mengembalikan jumlah karakter yang berbeda antara kedua string kata tersebut.
2. Tambahkan Kata Awal ke *wordQueue* dengan jumlah langkah 1 dan tanpa pendahulu.
3. Menambahkan kata target ke *wordSet*. Ini memastikan bahwa kata target dapat ditemukan selama pencarian.
4. Selama PriorityQueue tidak kosong, lakukan langkah-langkah berikut:
 - a. **Ambil Node dengan Prioritas Tertinggi:** Mengambil dan menghapus *WordNode* dengan prioritas tertinggi dari queue (yaitu, dengan jumlah langkah dan heuristik terendah).
 - b. **Periksa Jika Node adalah Target:** Jika kata dalam node sama dengan kata target, maka kita telah menemukan solusi. Cetak word ladder dan akhiri algoritma.
 - c. **Jelajahi Node Tetangga:** Jika kata dalam node bukan kata target, ubah kata saat ini menjadi array of char. Untuk setiap karakter dalam array, coba setiap huruf alfabet sebagai pengganti. Jika penggantian menghasilkan kata baru yang ada dalam set kata, tambahkan kata baru tersebut ke PriorityQueue dan hapus dari set kata. Jumlah langkah untuk kata baru adalah jumlah langkah untuk kata saat ini ditambah satu, dan kata saat ini adalah pendahulunya.

5. Jika PriorityQueue menjadi kosong dan kita belum menemukan kata target, maka tidak ada solusi.

3.2. Implementasi Algoritma UCS

Dalam menyelesaikan permainan Word Ladder, di mana tujuannya adalah menemukan urutan perpindahan huruf terpendek dari satu kata ke kata lain, kita dapat menggunakan algoritma Uniform Cost Search (UCS). UCS digunakan ketika tidak terdapat heuristik yang dapat mengestimasi jarak ke solusi, sehingga pencarian hanya didasarkan pada biaya atau jumlah langkah yang telah ditempuh. Berikut adalah langkah-langkah dalam mengimplementasikan algoritma UCS untuk menyelesaikan permainan Word Ladder:

1. Pertama-tama membuat PriorityQueue untuk menyimpan WordNode. PriorityQueue ini diurutkan berdasarkan jumlah langkah dari setiap WordNode, dengan WordNode yang memiliki jumlah langkah terkecil memiliki prioritas tertinggi. Dalam konteks Word Ladder dan algoritma pencarian ini, "jumlah langkah" merujuk pada jumlah perubahan kata yang telah dilakukan dari kata awal hingga kata saat ini. Ini sesuai dengan prinsip UCS, di mana node dengan biaya kumulatif terendah (dalam hal ini, jumlah langkah) selalu dieksplorasi terlebih dahulu.
2. Kata awal ditambahkan ke queue dengan jumlah langkah 0 dan tanpa pendahulu. Ini menandai titik awal pencarian.
3. HashSet dibuat untuk menyimpan kata-kata yang sudah dikunjungi. Ini digunakan untuk mencegah eksplorasi ulang dari node yang sama, yang bisa menyebabkan loop tak terbatas.
4. Selama queue tidak kosong, proses pencarian berlanjut. Ini adalah loop utama dari algoritma UCS. Tahap yang dilakukan dalam loop :
 - a. **Ambil Node:** WordNode dengan prioritas tertinggi (yaitu, dengan jumlah langkah terkecil) diambil dan dihapus dari queue.
 - b. **Periksa Jika Node adalah Target:** Kata dari WordNode saat ini dibandingkan dengan kata target. Jika mereka sama, maka word ladder telah ditemukan. Word ladder dicetak dan metode diakhiri.

- c. **Periksa Jika Node Sudah Dikunjungi:** Jika kata saat ini sudah dikunjungi (yaitu, ada dalam set kunjungan), maka iterasi berikutnya dari loop dimulai, mengabaikan semua langkah berikutnya dalam iterasi saat ini.
 - d. **Jelajahi Node Tetangga:** Untuk setiap tetangga dari kata saat ini, jika set kata mengandung tetangga dan tetangga belum dikunjungi, maka tetangga tersebut ditambahkan ke PriorityQueue. Jumlah langkah untuk tetangga adalah jumlah langkah untuk kata saat ini ditambah satu, dan kata saat ini ditetapkan sebagai pendahulunya. Ini memperluas pencarian ke node tetangga yang belum dikunjungi.
5. Jika queue menjadi kosong dan tidak ada word ladder yang ditemukan, maka pesan dicetak bahwa tidak ada word ladder yang ditemukan. Ini menandai akhir dari metode jika tidak ada solusi yang dapat ditemukan.

3.3. Implementasi Algoritma Greedy Best First Search

Untuk menyelesaikan permainan Word Ladder, di mana tujuannya adalah menemukan urutan perpindahan huruf terpendek dari satu kata ke kata lain, kita juga dapat menggunakan algoritma Greedy Best-First Search (GBFS). GBFS merupakan algoritma pencarian yang hanya bergantung pada heuristik atau perkiraan jarak untuk mencapai solusi tanpa mempertimbangkan biaya atau jumlah langkah yang sudah ditempuh. Meskipun GBFS tidak menjamin solusi optimal, namun dapat memberikan solusi yang cepat jika heuristiknya akurat. Berikut adalah langkah-langkah dalam mengimplementasikan GBFS untuk permainan Word Ladder:

1. Pertama-tama membuat PriorityQueue untuk menyimpan WordNode. PriorityQueue ini diurutkan berdasarkan heuristik dari setiap WordNode, dengan WordNode yang memiliki heuristik terkecil memiliki prioritas tertinggi. Heuristik ini dihitung sebagai jumlah karakter yang tidak cocok antara kata saat ini dan kata target. Ini sesuai dengan prinsip GBFS, di mana node yang paling menjanjikan (dalam hal ini, yang paling mirip dengan kata target) selalu dieksplorasi terlebih dahulu.
2. Tambahkan Kata Awal: Kata awal ditambahkan ke queue dengan jumlah langkah 0 dan tanpa pendahulu. Ini menandai titik awal pencarian.

3. Inisialisasi Set Kunjungan: HashSet dibuat untuk menyimpan kata-kata yang sudah dikunjungi. Ini digunakan untuk mencegah eksplorasi ulang dari node yang sama, yang bisa menyebabkan loop tak terbatas.
4. Pencarian: Selama queue tidak kosong, proses pencarian berlanjut. Ini adalah loop utama dari algoritma GBFS. Hal-hal yang terjadi pada loop:
 - a. **Ambil Node**: WordNode dengan prioritas tertinggi (yaitu, dengan heuristik terkecil) diambil dan dihapus dari queue.
 - b. **Periksa Jika Node adalah Target**: Kata dari WordNode saat ini dibandingkan dengan kata target. Jika mereka sama, maka word ladder telah ditemukan. Word ladder dicetak dan metode diakhiri.
 - c. **Periksa Jika Node Sudah Dikunjungi**: Jika kata saat ini sudah dikunjungi (yaitu, ada dalam set kunjungan), maka iterasi berikutnya dari loop dimulai, mengabaikan semua langkah berikutnya dalam iterasi saat ini.
 - d. **Jelajahi Node Tetangga**: Untuk setiap tetangga dari kata saat ini, jika set kata mengandung tetangga dan tetangga belum dikunjungi, maka tetangga tersebut ditambahkan ke PriorityQueue. Jumlah langkah untuk tetangga adalah jumlah langkah untuk kata saat ini ditambah satu, dan kata saat ini ditetapkan sebagai pendahulunya. Ini memperluas pencarian ke node tetangga yang belum dikunjungi.
5. Jika queue menjadi kosong dan tidak ada word ladder yang ditemukan, maka pesan dicetak bahwa tidak ada word ladder yang ditemukan. Ini menandai akhir dari metode jika tidak ada solusi yang dapat ditemukan.

3.4. Analisis Pertanyaan

Untuk lebih memperjelas analisis dari ketiga algoritma pencarian, maka kita akan menjawab beberapa pertanyaan yang relevan terhadap ketiga algoritma tersebut :

1. Definisi dari $f(n)$ dan $g(n)$

Dalam konteks algoritma pencarian A*, Greedy Best First Search (GBFS), dan UCS, $f(n)$ dan $g(n)$ didefinisikan sebagai berikut:

- a. **Untuk algoritma UCS** : $g(n)$ adalah biaya kumulatif dari node awal (atau akar) ke

node saat ini. Dalam konteks Word Ladder, ini bisa diinterpretasikan sebagai jumlah langkah yang diperlukan untuk berubah dari kata awal ke kata saat ini. Contoh : dog -> cog -> cot . Maka $g(\text{"cot"})$ adalah 2 karena kita telah melakukan 2 langkah untuk mencapai cot. Tidak ada $f(n)$ pada UCS karena UCS mengabaikan heuristik.

b. **Untuk algoritma Greedy Best First Search :** $f(n)$ merepresentasikan heuristik. Dalam konteks Word Ladder, heuristik yang biasa digunakan dalam algoritma Greedy Best First Search (GBFS) adalah jumlah huruf yang berbeda antara kata saat ini dan kata target. Misalnya, jika kata saat ini adalah "lead" dan kata target adalah "gold", maka heuristiknya adalah 3, karena ada tiga huruf yang berbeda antara "lead" dan "gold" (l vs g, e vs o, dan a vs l). Tidak terdapat $g(n)$ pada GBFS karena GBFS mengabaikan biaya yang diperlukan untuk mencapai node saat ini

c. **Untuk algoritma A*:** $f(n) = g(n) + h(n)$, di mana $h(n)$ adalah heuristik yang memperkirakan biaya terendah dari node saat ini ke node tujuan. Pada dasarnya, $h(n)$ pada A* adalah $f(n)$ pada Greedy Best First Search. $g(n)$ adalah biaya kumulatif dari node awal (atau akar) ke node saat ini. Pada dasarnya, $g(n)$ pada A* sama seperti $g(n)$ pada UCS.

2. Apakah heuristik yang digunakan pada algoritma A* admissible?

Heuristik yang admissible sangat penting dalam konteks algoritma A* untuk menjamin bahwa solusi yang ditemukan adalah optimal. Hal ini dikarenakan algoritma A* mencari jalur dari node awal ke node tujuan dengan meminimalkan fungsi $f(n) = g(n) + h(n)$. Jika heuristik $h(n)$ admissible, artinya heuristik tersebut tidak pernah melebih-lebihkan biaya untuk mencapai tujuan dari node saat ini.

Dengan kata lain, $h(n)$ selalu kurang dari atau sama dengan biaya sebenarnya untuk mencapai tujuan dari node n. Ini berarti bahwa A* tidak akan pernah mengabaikan node yang mungkin menjadi bagian dari jalur optimal dengan alasan bahwa biaya yang diprediksi untuk mencapai tujuan dari node tersebut terlalu tinggi. Sebaliknya, jika heuristik tidak admissible dan melebih-lebihkan biaya, A* mungkin akan mengabaikan

node yang seharusnya menjadi bagian dari jalur optimal, karena node tersebut tampaknya terlalu "mahal" untuk dieksplorasi berdasarkan heuristik yang melebih-lebihkan tersebut.

Dengan heuristik yang admissible, A* dijamin akan menemukan jalur dengan biaya terendah dari node awal ke node tujuan, yaitu jalur optimal. Jika heuristik tidak admissible, A* mungkin tidak akan menemukan jalur optimal.

3. Pada kasus word ladder, apakah algoritma UCS sama dengan BFS? (dalam artian urutan node yang dibangkitkan dan path yang dihasilkan sama)

Meskipun UCS dan BFS sama-sama dapat menemukan solusi terpendek jika ada, urutan node yang dibangkitkan dan jalur (path) yang dihasilkan belum tentu sama. Hal ini tergantung pada struktur graf dan biaya transisi antar node. Namun, dalam konteks permainan Word Ladder, di mana setiap perpindahan hanya melibatkan perubahan satu huruf, biaya transisi antar node adalah sama, yaitu 1.

Dalam situasi ini, UCS akan berperilaku persis seperti BFS. Alasannya, UCS memilih node berikutnya untuk dieksplorasi berdasarkan biaya kumulatif terendah dari node awal. Jika biaya transisi antar node sama, biaya kumulatif setiap node akan setara dengan jumlah node yang telah dieksplorasi sebelumnya. Dengan demikian, UCS akan menjelajahi graf secara breadth-first, menghasilkan urutan node dan jalur yang sama dengan BFS. Sehingga, dalam kasus Word Ladder, implementasi UCS dan BFS akan identik karena sifat biaya transisi yang seragam.

4. Secara teoritis, apakah algoritma A* lebih efisien dibandingkan dengan algoritma UCS pada kasus word ladder?

Secara teoritis, algoritma A* bisa lebih efisien dibandingkan dengan Uniform Cost Search (UCS) pada kasus Word Ladder, asalkan kita memiliki fungsi heuristik yang baik. Algoritma A* adalah peningkatan dari UCS dengan menambahkan fungsi heuristik, yang memberikan perkiraan biaya dari node saat ini ke tujuan. Dengan kata lain, A* tidak hanya mempertimbangkan biaya yang telah dikeluarkan untuk mencapai node saat ini (seperti yang dilakukan oleh UCS), tetapi juga mempertimbangkan perkiraan biaya untuk mencapai tujuan dari node saat ini.

Jika fungsi heuristik yang digunakan adalah admissible (yaitu, tidak pernah surestimasikan biaya sebenarnya untuk mencapai tujuan), maka A* dijamin akan menemukan solusi optimal, sama seperti UCS. Namun, dengan menggunakan fungsi heuristik, A* biasanya dapat menemukan solusi tersebut dengan lebih sedikit langkah, karena ia dapat "mengarah" pencarian ke arah tujuan, daripada menjelajahi semua node secara merata seperti yang dilakukan oleh UCS.

Namun, perlu dicatat bahwa efisiensi A* sangat bergantung pada kualitas fungsi heuristik. Jika fungsi heuristik tidak memberikan perkiraan yang baik tentang biaya untuk mencapai tujuan, maka A* mungkin tidak lebih efisien dari UCS. Selain itu, menghitung fungsi heuristik itu sendiri juga bisa memakan waktu, jadi harus ada keseimbangan antara kompleksitas fungsi heuristik dan peningkatan efisiensi yang diberikannya.

5. Secara teoritis, apakah algoritma Greedy Best First Search menjamin solusi optimal untuk persoalan word ladder?

Tidak, secara teoritis, algoritma Greedy Best-First Search (GBFS) tidak menjamin solusi optimal untuk persoalan Word Ladder. Algoritma GBFS memilih node berikutnya untuk dieksplorasi berdasarkan fungsi heuristik yang memberikan perkiraan biaya dari node saat ini ke tujuan. Algoritma ini "greedy" karena selalu memilih node yang tampaknya paling dekat dengan tujuan, tanpa mempertimbangkan biaya yang telah dikeluarkan untuk mencapai node saat ini.

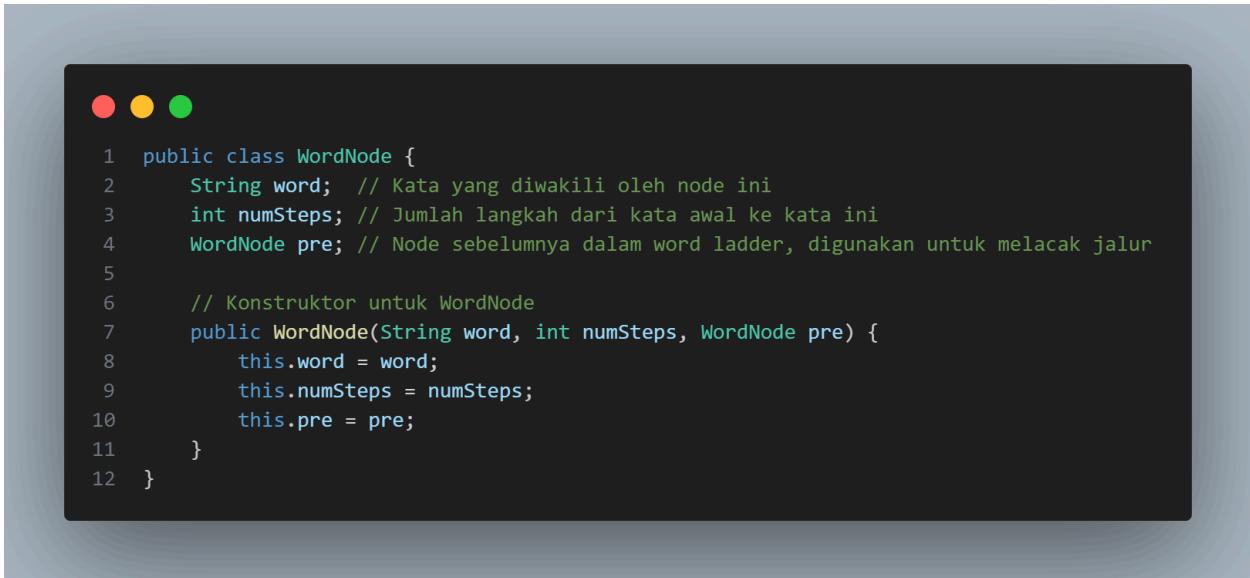
Dalam beberapa kasus, pendekatan ini dapat mengarahkan pencarian ke arah yang salah dan menghasilkan solusi yang tidak optimal. Misalnya, GBFS mungkin memilih untuk menjelajahi node yang tampaknya dekat dengan tujuan, tetapi sebenarnya memerlukan banyak langkah untuk mencapainya, sementara mengabaikan node lain yang mungkin sedikit lebih jauh tetapi dapat dicapai dengan lebih sedikit langkah.

BAB IV

Source Code dan Input Output Program

4.1. Souce Code Program Terminal

1. Kelas WordNode



```
1 public class WordNode {
2     String word; // Kata yang diwakili oleh node ini
3     int numSteps; // Jumlah langkah dari kata awal ke kata ini
4     WordNode pre; // Node sebelumnya dalam word ladder, digunakan untuk melacak jalur
5
6     // Konstruktor untuk WordNode
7     public WordNode(String word, int numSteps, WordNode pre) {
8         this.word = word;
9         this.numSteps = numSteps;
10        this.pre = pre;
11    }
12 }
```

Penjelasan : Kelas WordNode digunakan untuk merepresentasikan sebuah node dalam konteks Word Ladder. Setiap instance dari kelas ini memiliki tiga atribut:

- word : ni adalah String yang merepresentasikan kata yang diwakili oleh node ini.
- numSteps : Ini adalah int egeryang menyimpan jumlah langkah dari kata awal ke kata yang diwakili oleh node ini.
- pre : Ini adalah instance lain dari WordNode yang mewakili node sebelumnya dalam Word Ladder. Ini digunakan untuk melacak jalur dari kata awal ke kata saat ini.

2. Kelas WordLadderUtils

```
● ● ●

1  public class WordLadderUtils {
2      // Metode untuk mendapatkan semua tetangga dari suatu kata
3      public static List<String> getNeighbors(String word) {
4          List<String> neighbors = new ArrayList<>();
5          char[] chars = word.toCharArray();
6
7          // Untuk setiap karakter dalam kata
8          for (int i = 0; i < chars.length; i++) {
9              char old = chars[i];
10             // Coba ganti dengan setiap huruf lain dalam abjad
11             for (char c = 'a'; c <= 'z'; c++) {
12                 if (c == old) {
13                     continue;
14                 }
15                 chars[i] = c;
16                 neighbors.add(new String(chars));
17             }
18             // Kembalikan karakter asli
19             chars[i] = old;
20         }
21
22         return neighbors;
23     }
24
25     // Metode untuk mencetak word ladder dari suatu node
26     public static void printWordLadder(WordNode node, int visitedNodes) {
27         Deque<String> words = new ArrayDeque<>();
28         int distance = node.numSteps; // Simpan jarak sebelum loop
29         // Membangun word ladder dengan mengikuti predecessor dari setiap node
30         while (node != null) {
31             words.push(node.word);
32             node = node.pre;
33         }
34         // Mencetak word ladder dan jarak
35         System.out.println("Word ladder: " + String.join(" -> ", words));
36         System.out.println("Jarak: " + distance);
37         System.out.println("Banyaknya node yang dikunjungi: " + visitedNodes);
38     }
39
40     // Metode untuk menghitung heuristik antara dua kata
41     public static int heuristic(String word, String target) {
42         int mismatches = 0;
43         // Menghitung jumlah karakter yang tidak cocok
44         for (int i = 0; i < word.length(); i++) {
45             if (word.charAt(i) != target.charAt(i)) {
46                 mismatches++;
47             }
48         }
49         return mismatches;
50     }
```

Penjelasan : Kelas WordLadderUtils berisi metode-metode statis yang digunakan untuk membantu dalam proses pencarian Word Ladder. Berikut penjelasan untuk setiap metode:

- `getNeighbors(String word)`: Metode ini menerima sebuah kata dan menghasilkan daftar semua "tetangga" dari kata tersebut. Dalam konteks Word Ladder, sebuah "tetangga" adalah kata yang bisa kita capai dengan mengubah satu huruf dari kata asli. Metode ini melakukan ini dengan mengubah setiap huruf dalam kata asli satu per satu ke setiap huruf lain dalam abjad, dan menambahkan hasilnya ke daftar tetangga.
- `printWordLadder(WordNode node, int visitedNodes)`: Metode ini menerima sebuah WordNode dan jumlah node yang dikunjungi, dan mencetak Word Ladder dari node tersebut. Word Ladder dibangun dengan mengikuti "predecessor" dari setiap node, dimulai dari node yang diberikan dan bergerak mundur melalui setiap "predecessor" sampai kita mencapai awal. Word Ladder dan jaraknya kemudian dicetak.
- `heuristic(String word, String target)`: Metode ini menerima dua kata dan menghitung heuristik antara mereka, yang dalam kasus ini adalah jumlah karakter yang tidak cocok. Heuristik ini bisa digunakan dalam algoritma pencarian berbasis heuristik seperti A* atau Greedy Best-First Search untuk memperkirakan biaya dari kata saat ini ke tujuan.

3. Kelas Astar



```
1 public class Astar {
2     public static void findWordLadder(String startWord, String targetWord, Set<String> wordSet) {
3         /* Membuat PriorityQueue untuk menyimpan WordNode, dengan Comparator yang
4            membandingkan jumlah langkah dan heuristik
5            */
6         PriorityQueue<WordNode> wordQueue =
7             new PriorityQueue<>(Comparator.comparingInt(
8                 node -> node.numSteps + WordLadderUtils.heuristic(node.word, targetWord)));
9
10        // Menambahkan startWord ke queue dengan jumlah langkah 1 dan tanpa predecessor
11        wordQueue.add(new WordNode(startWord, 1, null));
12
13        // Menambahkan targetWord ke wordSet
14        wordSet.add(targetWord);
15
16        int visitedNodes = 0;
17
18        // Melakukan loop selama queue tidak kosong
19        while (!wordQueue.isEmpty()) {
20            // Mengambil dan menghapus WordNode dengan prioritas tertinggi dari queue
21            WordNode currentNode = wordQueue.poll();
22            visitedNodes++;
23            String currentWord = currentNode.word;
24
25            // Jika currentWord sama dengan targetWord, mencetak word ladder dan mengakhiri metode
26            if (currentWord.equals(targetWord)) {
27                WordLadderUtils.printWordLadderA(currentNode, visitedNodes);
28                return;
29            }
30
31            // Mengubah currentWord menjadi array of char untuk manipulasi karakter
32            char[] wordChars = currentWord.toCharArray();
33            for (int i = 0; i < wordChars.length; i++) {
34                for (char c = 'a'; c <= 'z'; c++) {
35                    char originalChar = wordChars[i];
36                    // Mengganti karakter di posisi i dengan c jika mereka berbeda
37                    if (wordChars[i] != c) {
38                        wordChars[i] = c;
39                    }
40
41                    // Membuat string baru dari array of char
42                    String newWord = new String(wordChars);
43                    // Jika wordSet mengandung newWord, menambahkannya ke queue dan
44                    // menghapusnya dari wordSet
45                    if (wordSet.contains(newWord)) {
46                        wordQueue.add(new WordNode(newWord, currentNode.numSteps + 1, currentNode));
47                        wordSet.remove(newWord);
48                    }
49
50                    // Mengembalikan karakter asli ke posisi i
51                    wordChars[i] = originalChar;
52                }
53            }
54        }
55    }
```

Penjelasan : Kelas Astar berisi metode statis findWordLadder yang menerapkan algoritma pencarian A* untuk menemukan solusi Word Ladder. Metode findWordLadder menggunakan struktur data PriorityQueue untuk menyimpan objek WordNode, yang masing-masing mewakili kata dalam Word Ladder dan memiliki informasi tentang berapa banyak langkah yang dibutuhkan untuk mencapai kata tersebut dan apa kata sebelumnya dalam urutan.

PriorityQueue diurutkan berdasarkan jumlah langkah dan heuristik, yang dihitung sebagai jumlah karakter yang tidak cocok antara kata saat ini dan kata target. Metode ini juga menggunakan Set kata yang valid, dan mencoba setiap kemungkinan perubahan satu huruf untuk kata saat ini, menambahkan kata-kata baru yang valid ke PriorityQueue dan menghapusnya dari Set. Jika metode ini menemukan kata target dalam PriorityQueue, itu mencetak Word Ladder dan mengakhiri. Jika PriorityQueue menjadi kosong sebelum menemukan kata target, itu berarti tidak ada solusi.

4. Kelas Greedy



```
1 public class Greedy {
2     public static void findWordLadder(String startWord, String targetWord, Set<String> wordSet) {
3         // Membuat PriorityQueue untuk menyimpan WordNode, dengan Comparator yang membandingkan heuristik saja
4         PriorityQueue<WordNode> queue =
5             new PriorityQueue<>(Comparator.comparingInt(node -> WordLadderUtils.heuristic(node.word, targetWord)));
6
7         // Menambahkan startWord ke queue dengan jumlah langkah 0 dan tanpa predecessor
8         queue.add(new WordNode(startWord, 0, null));
9
10        // Membuat HashSet untuk menyimpan kata-kata yang sudah dikunjungi
11        Set<String> visited = new HashSet<>();
12
13        // Mentrack visited nodes
14        int visitedNodes = 0;
15
16        // Melakukan loop selama queue tidak kosong
17        while (!queue.isEmpty()) {
18            // Mengambil dan menghapus WordNode dengan prioritas tertinggi dari queue
19            WordNode node = queue.poll();
20            visitedNodes++;
21            String word = node.word;
22
23            // Jika word sama dengan targetWord, mencetak word ladder dan mengakhiri metode
24            if (word.equals(targetWord)) {
25                WordLadderUtils.printWordLadder(node, visitedNodes);
26                return;
27            }
28
29            // Jika word sudah dikunjungi, melanjutkan ke iterasi berikutnya
30            if (!visited.add(word)) {
31                continue;
32            }
33
34            // Untuk setiap tetangga dari word
35            for (String neighbor : WordLadderUtils.getNeighbors(word)) {
36                // Jika wordSet mengandung neighbor dan neighbor belum dikunjungi, menambahkannya ke queue
37                if (wordSet.contains(neighbor) && !visited.contains(neighbor)) {
38                    queue.add(new WordNode(neighbor, node.numSteps + 1, node));
39                }
40            }
41        }
42
43        // Jika tidak ada word ladder yang ditemukan, mencetak pesan
44        System.out.println("No word ladder found.");
45    }
}
```

Penjelasan : Kelas Greedy berisi metode statis findWordLadder yang menerapkan algoritma pencarian Greedy untuk menemukan solusi Word Ladder. Metode findWordLadder menggunakan struktur data PriorityQueue untuk menyimpan objek WordNode, yang masing-masing mewakili kata dalam Word Ladder dan memiliki informasi tentang berapa banyak langkah yang dibutuhkan untuk mencapai kata tersebut dan apa kata sebelumnya dalam urutan.

PriorityQueue diurutkan berdasarkan heuristik, yang dihitung sebagai jumlah karakter yang tidak cocok antara kata saat ini dan kata target. Metode ini juga menggunakan Set kata yang valid, dan mencoba setiap kemungkinan perubahan satu huruf untuk kata saat ini, menambahkan kata-kata baru yang valid ke PriorityQueue jika mereka belum dikunjungi sebelumnya.

5. Kelas UCS



```
1 public class UCS {
2     public static void findWordLadder(String startWord, String targetWord, Set<String> wordSet) {
3         // Membuat PriorityQueue untuk menyimpan WordNode, dengan Comparator yang
4         // membandingkan jumlah langkah saja
5         PriorityQueue<WordNode> queue =
6             new PriorityQueue<>(Comparator.comparingInt(node -> node.numSteps));
7
8         // Menambahkan startWord ke queue dengan jumlah langkah 0 dan tanpa predecessor
9         queue.add(new WordNode(startWord, 0, null));
10
11        // Membuat HashSet untuk menyimpan kata-kata yang sudah dikunjungi
12        Set<String> visited = new HashSet<>();
13
14        // Mentrack visited nodes
15        int visitedNodes = 0;
16
17        // Melakukan loop selama queue tidak kosong
18        while (!queue.isEmpty()) {
19            // Mengambil dan menghapus WordNode dengan prioritas tertinggi dari queue
20            WordNode node = queue.poll();
21            visitedNodes++;
22            String word = node.word;
23
24            // Jika word sama dengan targetWord, mencetak word ladder dan mengakhiri metode
25            if (word.equals(targetWord)) {
26                WordLadderUtils.printWordLadder(node, visitedNodes);
27                return;
28            }
29
30            // Jika word sudah dikunjungi, melanjutkan ke iterasi berikutnya
31            if (!visited.add(word)) {
32                continue;
33            }
34
35            // Untuk setiap tetangga dari word
36            for (String neighbor : WordLadderUtils.getNeighbors(word)) {
37                // Jika wordSet mengandung neighbor dan neighbor belum dikunjungi,
38                // menambahkannya ke queue
39                if (wordSet.contains(neighbor) && !visited.contains(neighbor)) {
40                    queue.add(new WordNode(neighbor, node.numSteps + 1, node));
41                }
42            }
43        }
44
45        // Jika tidak ada word ladder yang ditemukan, mencetak pesan
46        System.out.println("No word ladder found.");
47    }
}
```

Penjelasan : Kelas UCS berisi metode statis findWordLadder yang menerapkan algoritma pencarian Uniform Cost Search (UCS) untuk menemukan solusi Word Ladder. Metode findWordLadder menggunakan struktur data PriorityQueue untuk menyimpan objek WordNode, yang masing-masing mewakili kata dalam Word Ladder dan memiliki

informasi tentang berapa banyak langkah yang dibutuhkan untuk mencapai kata tersebut dan apa kata sebelumnya dalam urutan.

PriorityQueue diurutkan berdasarkan jumlah langkah. Metode ini juga menggunakan Set kata yang valid, dan mencoba setiap kemungkinan perubahan satu huruf untuk kata saat ini, menambahkan kata-kata baru yang valid ke PriorityQueue jika mereka belum dikunjungi sebelumnya. Jika metode ini menemukan kata target dalam PriorityQueue, itu mencetak Word Ladder dan mengakhiri. Jika PriorityQueue menjadi kosong sebelum menemukan kata target, itu berarti tidak ada solusi dan mencetak pesan "No word ladder found."

6. Kelas Main

```
● ● ●
1 import java.util.*;
2 import java.nio.file.*;
3 import java.io.IOException;
4
5 public class Main {
6     public static void main(String[] args) throws IOException {
7         // Membaca semua baris dari file dan menyimpannya dalam List
8         List<String> wordList = Files.readAllLines(Paths.get("./words_alpha.txt"));
9         // Mengubah List menjadi Set untuk pencarian yang lebih efisien
10        Set<String> wordSet = new HashSet<>(wordList);
11
12        // Membuat Scanner untuk membaca input dari terminal
13        Scanner scanner = new Scanner(System.in);
14
15        try {
16            // Meminta pengguna untuk memasukkan kata awal
17            System.out.println("Enter the start word:");
18            String startWord = scanner.nextLine().toLowerCase();
19
20            // Meminta pengguna untuk memasukkan kata target
21            System.out.println("Enter the target word:");
22            String targetWord = scanner.nextLine().toLowerCase();
23
24            // Memeriksa apakah kata awal dan kata target ada dalam set kata
25            if (!wordSet.contains(startWord) || !wordSet.contains(targetWord)) {
26                System.out.println("Error: Start Word dan End word harus merupakan kata-kata yang berada di kamus bahasa Inggris.");
27                return;
28            }
29
30            // Memeriksa apakah kata awal dan kata target memiliki panjang yang sama
31            if (startWord.length() != targetWord.length()) {
32                System.out.println("Error: Start Word dan End word harus memiliki panjang yang sama.");
33                return;
34            }
35
36            // Meminta pengguna untuk memilih algoritma
37            System.out.println("Choose an algorithm: 1 for Astar, 2 for Greedy, 3 for UCS");
38            int choice = scanner.nextInt();
39
40            // Mengambil waktu awal
41            long startTime = System.nanoTime();
42
43            // Menjalankan algoritma yang dipilih oleh pengguna
44            switch (choice) {
45                case 1:
46                    Astar.findWordLadder(startWord, targetWord, wordSet);
47                    break;
48                case 2:
49                    Greedy.findWordLadder(startWord, targetWord, wordSet);
50                    break;
51                case 3:
52                    UCS.findWordLadder(startWord, targetWord, wordSet);
53                    break;
54            }
55
56            // Mengambil waktu akhir dan menghitung durasi
57            long endTime = System.nanoTime();
58            long duration = (endTime - startTime) / 1_000_000;
59            // Mencetak durasi eksekusi
60            System.out.println("Execution time: " + duration + " ms");
61        } finally {
62            // Menutup scanner
63            scanner.close();
64        }
65    }
66 }
```

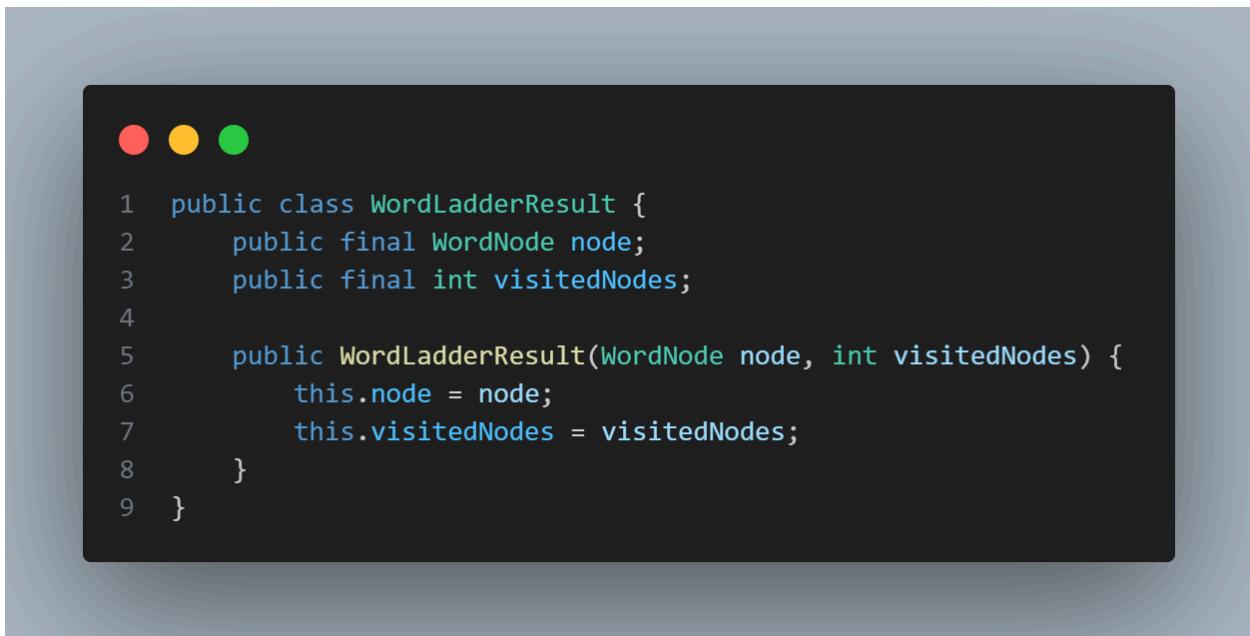
Penjelasan : Kelas Main berisi metode main yang merupakan titik masuk aplikasi. Metode ini membaca daftar kata dari file, mengubahnya menjadi Set untuk pencarian yang lebih efisien, dan kemudian meminta pengguna untuk memasukkan kata awal dan

kata target untuk permainan Word Ladder. Setelah memvalidasi input pengguna, metode ini meminta pengguna untuk memilih algoritma pencarian yang akan digunakan: A*, Greedy, atau Uniform Cost Search (UCS).

Kemudian, metode ini menjalankan algoritma yang dipilih, mencetak Word Ladder yang ditemukan (jika ada) beserta jarak path tersebut, mencetak banyak node yang dikunjungi dan mencetak waktu eksekusi dalam milidetik.

4.2. Souce Code Program Bonus GUI

1. Kelas WordLadderResult



The screenshot shows a dark-themed code editor window. At the top left, there are three colored circular icons: red, yellow, and green. Below them, the Java code for the `WordLadderResult` class is displayed:

```
1 public class WordLadderResult {  
2     public final WordNode node;  
3     public final int visitedNodes;  
4  
5     public WordLadderResult(WordNode node, int visitedNodes) {  
6         this.node = node;  
7         this.visitedNodes = visitedNodes;  
8     }  
9 }
```

Penjelasan : Kelas WordLadderResult adalah kelas pembungkus atau "wrapper" yang digunakan untuk menyimpan hasil dari pencarian Word Ladder. Kelas ini memiliki dua field:

1. `node`: ini adalah objek `WordNode` yang mewakili kata terakhir dalam Word Ladder yang ditemukan.
2. `visitedNodes`: ini adalah jumlah total node atau kata yang dikunjungi selama pencarian Word Ladder.

2. Pada kelas WordLadderUtils, ditambahkan metode untuk GUI:



```
1 public static String printWordLadderGUI(WordLadderResult result) {
2     String resultStr = "";
3     if (result.node == null)
4     {
5         resultStr = "Solusi tidak ditemukan / Tidak ada solusi";
6     }
7     else
8     {
9         WordNode node = result.node;
10        Deque<String> words = new ArrayDeque<>();
11        int distance = node.numSteps;
12        while (node != null) {
13            words.push(node.word);
14            node = node.pre;
15        }
16        resultStr = "Word ladder: " + String.join(" -> ", words) + "\n" + "Jarak: " + distance;
17    }
18    resultStr += "\nBanyaknya node yang dikunjungi: " + result.visitedNodes;
19    return resultStr;
20 }
```

Penjelasan : Metode printWordLadderGUI ini mengambil objek WordLadderResult sebagai argumen dan mengembalikan representasi String dari Word Ladder yang ditemukan, atau pesan bahwa tidak ada solusi yang ditemukan. Jika node dalam WordLadderResult adalah null, ini berarti tidak ada Word Ladder yang ditemukan, dan metode ini mengembalikan pesan "Solusi tidak ditemukan / Tidak ada solusi".

Jika node bukan null, metode ini mengikuti rantai pre dari node untuk mendapatkan semua kata dalam Word Ladder, dan menambahkannya ke Deque kata. Metode ini kemudian menggabungkan semua kata dalam Deque menjadi satu String dengan panah "->" di antara kata-kata, dan menambahkan jumlah langkah (jarak) ke String. Akhirnya, metode ini menambahkan jumlah node yang dikunjungi ke String dan mengembalikannya.

3. Pada kelas Astar, ditambahkan metode untuk GUI



```
1 public static WordLadderResult findWordLadderGUI(String startWord, String targetWord, Set<String> wordSet) {
2     // Membuat PriorityQueue untuk menyimpan WordNode
3     PriorityQueue<WordNode> wordQueue =
4         new PriorityQueue<>(Comparator.comparingInt(node -> node.numSteps
5             + WordLadderUtils.heuristic(node.word, targetWord)));
6
7     // Menambahkan startWord ke queue
8     wordQueue.add(new WordNode(startWord, 1, null));
9
10    // Menambahkan targetWord ke wordSet
11    wordSet.add(targetWord);
12
13    int visitedNodes = 0;
14
15    // Loop selama queue tidak kosong
16    while (!wordQueue.isEmpty()) {
17        WordNode currentNode = wordQueue.poll();
18        visitedNodes++;
19        String currentWord = currentNode.word;
20
21        // Jika currentWord sama dengan targetWord, kembalikan WordLadderResult
22        if (currentWord.equals(targetWord)) {
23            return new WordLadderResult(currentNode, visitedNodes);
24        }
25
26        // Menciptakan semua kemungkinan kata baru dari currentWord
27        char[] wordChars = currentWord.toCharArray();
28        for (int i = 0; i < wordChars.length; i++) {
29            for (char c = 'a'; c <= 'z'; c++) {
30                char originalChar = wordChars[i];
31                if (wordChars[i] != c) {
32                    wordChars[i] = c;
33                }
34
35                String newWord = new String(wordChars);
36                if (wordSet.contains(newWord)) {
37                    wordQueue.add(new WordNode(newWord, currentNode.numSteps + 1, currentNode));
38                    wordSet.remove(newWord);
39                }
40
41                // Mengembalikan karakter asli ke posisi i
42                wordChars[i] = originalChar;
43            }
44        }
45    }
46
47    return new WordLadderResult(null, visitedNodes);
48 }
```

Penjelasan : Metode `findWordLadderGUI` ini adalah variasi dari metode `findWordLadder` yang berada pada kelas Astar. Perbedaannya, hanyalah dalam hasil yang dikembalikan. Setiap kali metode ini mengunjungi node, ia memeriksa apakah kata dalam node adalah kata target. Jika ya, metode ini mengembalikan `WordLadderResult` dengan node tersebut dan jumlah node yang dikunjungi. Jika `PriorityQueue` kosong dan metode ini belum

menemukan kata target, metode ini mengembalikan WordLadderResult dengan null sebagai node dan jumlah node yang dikunjungi.

4. Pada kelas Greedy ditambahkan metode untuk GUI :



```
1 public static WordLadderResult findWordLadderGUI(String startWord, String targetWord, Set<String> wordSet) {
2     PriorityQueue<WordNode> queue = new PriorityQueue<>(<Comparator>.comparingInt
3         (node -> WordLadderUtils.heuristic(node.word, targetWord)));
4     queue.add(new WordNode(startWord, 0, null));
5     Set<String> visited = new HashSet<>();
6
7     int visitedNodes = 0;
8
9     while (!queue.isEmpty()) {
10        WordNode node = queue.poll();
11        visitedNodes++;
12        String word = node.word;
13
14        if (word.equals(targetWord)) {
15            return new WordLadderResult(node, visitedNodes);
16        }
17
18        if (!visited.add(word)) {
19            continue;
20        }
21
22        for (String neighbor : WordLadderUtils.getNeighbors(word)) {
23            if (wordSet.contains(neighbor) && !visited.contains(neighbor)) {
24                queue.add(new WordNode(neighbor, node.numSteps + 1, node));
25            }
26        }
27    }
28
29    return new WordLadderResult(null, visitedNodes);
30 }
```

Penjelasan : Metode findWordLadderGUI ini adalah variasi dari metode findWordLadder yang berada pada kelas Greedy. Perbedaannya, hanyalah dalam hasil yang dikembalikan. Setiap kali metode ini mengunjungi node, ia memeriksa apakah kata dalam node adalah kata target. Jika ya, metode ini mengembalikan WordLadderResult dengan node tersebut dan jumlah node yang dikunjungi.

Jika kata dalam node bukan kata target, metode ini menambahkan semua tetangga kata yang belum dikunjungi ke PriorityQueue. Jika PriorityQueue kosong dan metode ini belum menemukan kata target, metode ini mengembalikan WordLadderResult dengan null sebagai node dan jumlah node yang dikunjungi.

5. Pada kelas UCS, ditambahkan metode untuk GUI:



```
1 public static WordLadderResult findWordLadderGUI(String startWord, String targetWord, Set<String> wordSet) {
2     PriorityQueue<WordNode> queue =
3         new PriorityQueue<>(Comparator.comparingInt(node -> node.numSteps));
4     queue.add(new WordNode(startWord, 0, null));
5     Set<String> visited = new HashSet<>();
6
7     int visitedNodes = 0;
8
9     while (!queue.isEmpty()) {
10        WordNode node = queue.poll();
11        visitedNodes++;
12        String word = node.word;
13
14        if (word.equals(targetWord)) {
15            return new WordLadderResult(node, visitedNodes);
16        }
17
18        if (!visited.add(word)) {
19            continue;
20        }
21
22        for (String neighbor : WordLadderUtils.getNeighbors(word)) {
23            if (wordSet.contains(neighbor) && !visited.contains(neighbor)) {
24                queue.add(new WordNode(neighbor, node.numSteps + 1, node));
25            }
26        }
27    }
28
29    return new WordLadderResult(null, visitedNodes);
30 }
```

Penjelasan : Metode `findWordLadderGUI` ini adalah variasi dari metode `findWordLadder` yang berada pada kelas UCS. Perbedaannya, hanyalah dalam hasil yang dikembalikan. Setiap kali metode ini mengunjungi node, ia memeriksa apakah kata dalam node adalah kata target. Jika ya, metode ini mengembalikan `WordLadderResult` dengan node tersebut dan jumlah node yang dikunjungi. Jika `PriorityQueue` kosong dan metode ini belum menemukan kata target, metode ini mengembalikan `WordLadderResult` dengan null sebagai node dan jumlah node yang dikunjungi.

6. Kelas MainGUI



```
1 public class MainGUI {
2     public static void main(String[] args) {
3         // Membuat frame
4         JFrame frame = new JFrame("Word Ladder");
5         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
6         frame.setSize(600, 400);
7
8         // Membuat panel
9         JPanel panel = new JPanel();
10        frame.add(panel);
11        placeComponents(panel);
12
13        // Menampilkan frame
14        frame.setVisible(true);
15    }
}
```

Kelas MainGUI ini adalah kelas utama yang digunakan untuk membuat antarmuka pengguna (GUI) untuk aplikasi Word Ladder. Metode main membuat frame utama aplikasi, menetapkan ukurannya, dan membuat panel utama yang akan menampung semua komponen lainnya. Metode placeComponents kemudian dipanggil untuk menambahkan komponen-komponen tersebut ke panel. Metode placeComponents membuat dan menempatkan berbagai komponen GUI ke dalam panel, termasuk label dan field teks untuk kata awal dan kata target, combo box untuk memilih algoritma, tombol untuk menjalankan algoritma, dan area teks untuk menampilkan output.

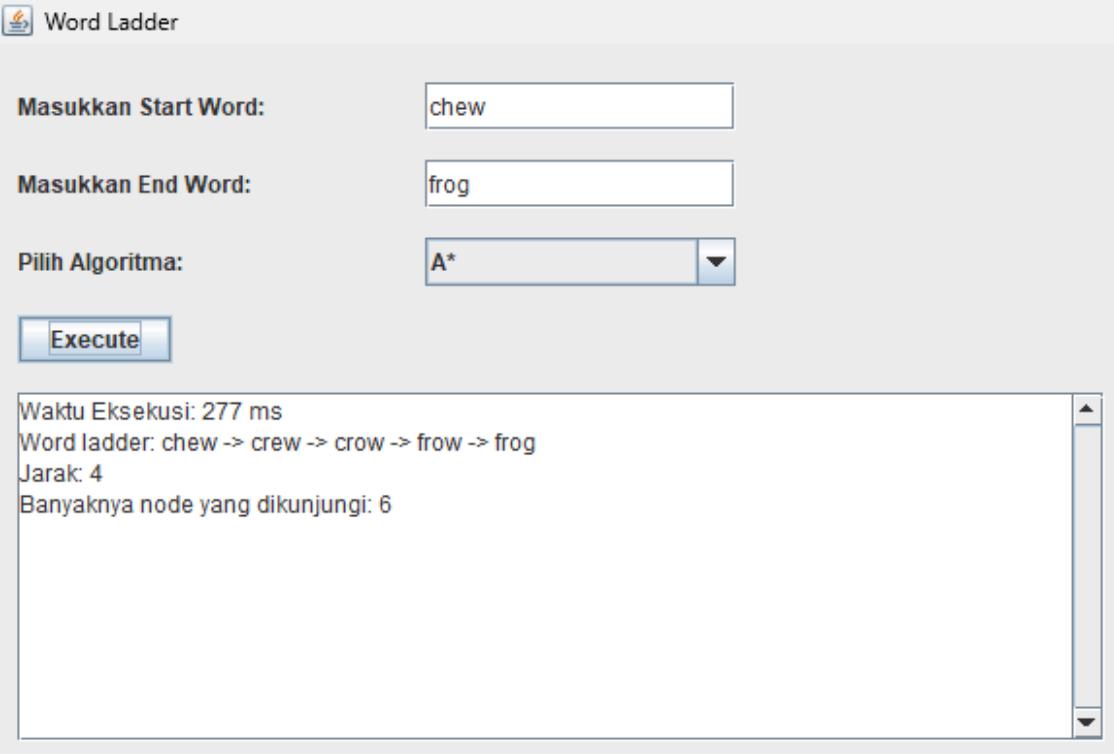
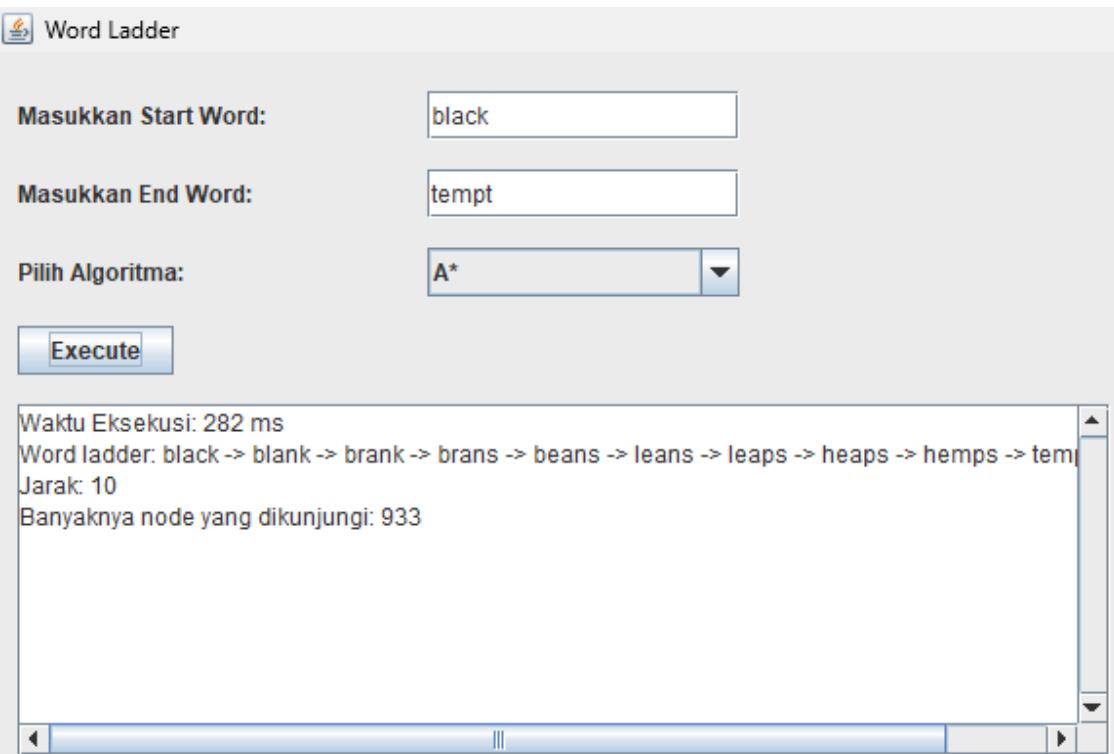
Ketika tombol "Execute" ditekan, metode actionPerformed dari ActionListener yang ditambahkan ke tombol tersebut akan dipanggil. Metode ini mengambil kata awal dan kata target dari field teks, memilih algoritma dari combo box, memuat daftar kata dari file, memvalidasi kata-kata tersebut, menjalankan algoritma yang dipilih, dan menampilkan waktu eksekusi dan hasilnya di area output. Kelas ini menggunakan beberapa kelas lain seperti Astar, Greedy, UCS, dan WordLadderUtils untuk mencari dan mencetak Word Ladder.

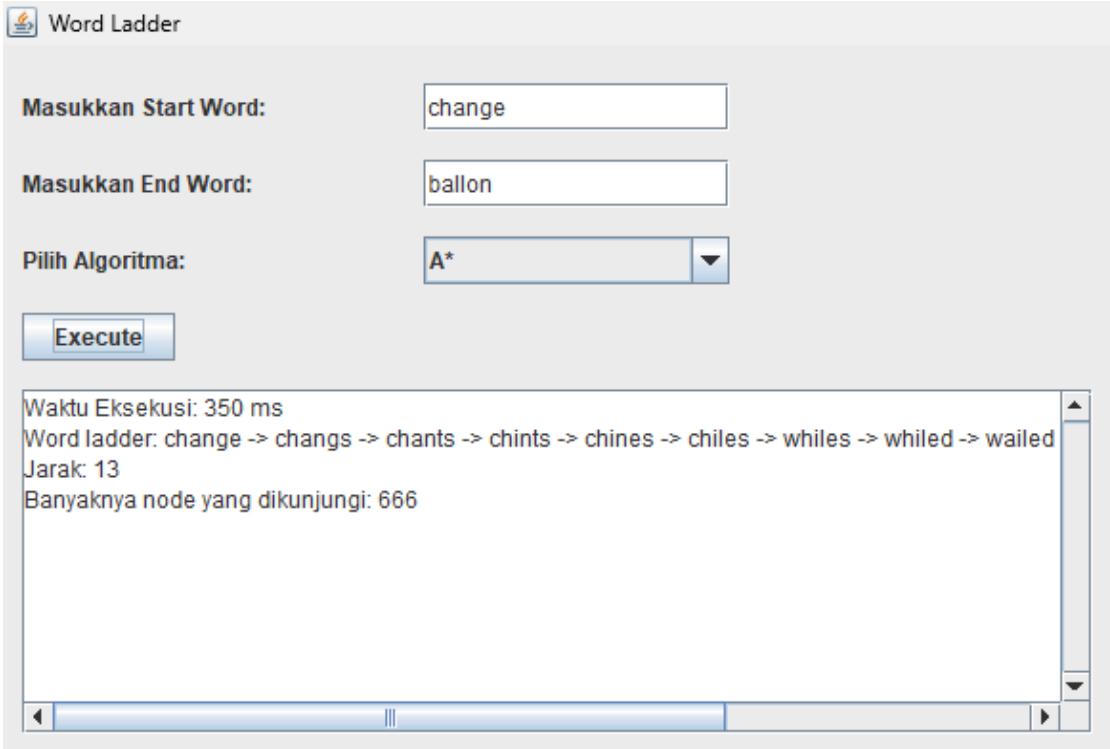
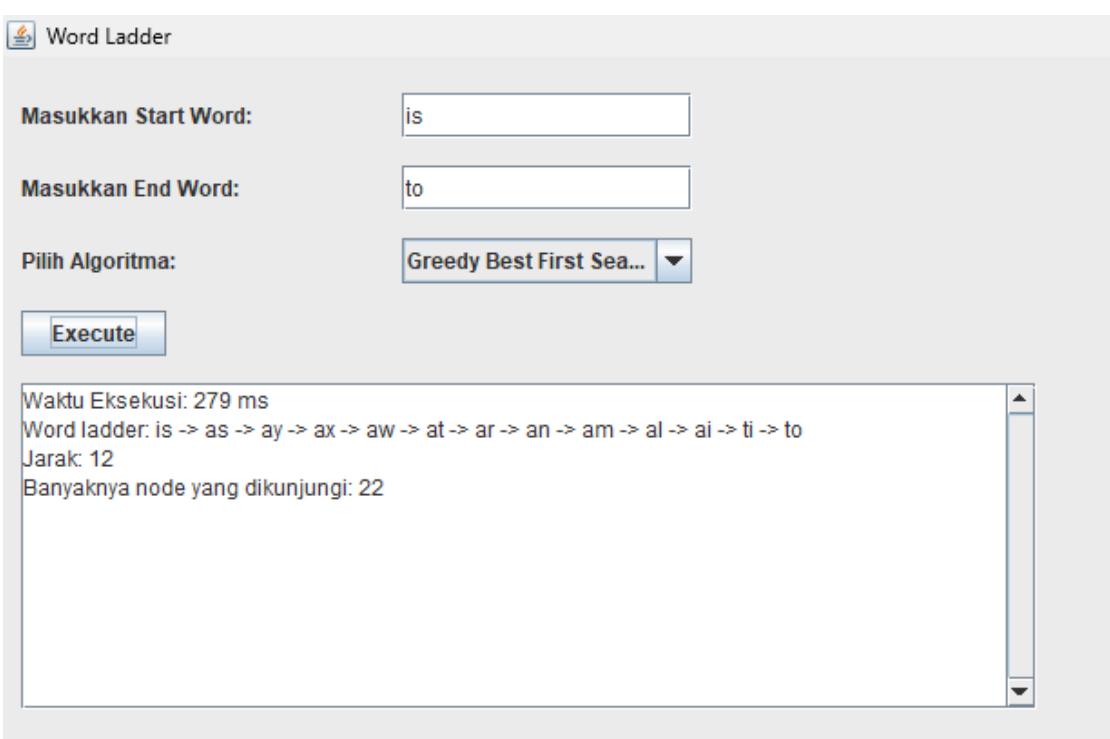
```
1 private static void placeComponents(JPanel panel) {
2     panel.setLayout(null);
3
4     // Membuat label dan field teks untuk kata awal
5     JLabel startLabel = new JLabel("Masukkan Start Word:");
6     startLabel.setBounds(10, 20, 200, 25);
7     panel.add(startLabel);
8
9     JTextField startText = new JTextField(20);
10    startText.setBounds(220, 20, 160, 25);
11    panel.add(startText);
12
13     // Membuat label dan field teks untuk kata target
14     JLabel targetLabel = new JLabel("Masukkan End Word:");
15     targetLabel.setBounds(10, 60, 200, 25);
16     panel.add(targetLabel);
17
18     JTextField targetText = new JTextField(20);
19     targetText.setBounds(220, 60, 160, 25);
20     panel.add(targetText);
21
22     // Membuat label dan combo box untuk pilihan algoritma
23     JLabel algorithmLabel = new JLabel("Pilih Algoritma:");
24     algorithmLabel.setBounds(10, 100, 200, 25);
25     panel.add(algorithmLabel);
26
27     String[] algorithms = { "Astar", "Greedy", "UCS" };
28     JComboBox<String> algorithmBox = new JComboBox<String>(algorithms);
29     algorithmBox.setBounds(220, 100, 100, 25);
30     panel.add(algorithmBox);
31
32     // Membuat tombol execute
33     JButton executeButton = new JButton("Execute");
34     executeButton.setBounds(10, 140, 80, 25);
35     panel.add(executeButton);
36
37     // Membuat area output
38     JTextArea outputArea = new JTextArea();
39     outputArea.setBounds(10, 180, 560, 180);
40
41     // Membuat scroll pane dan menambahkan area output ke dalamnya
42     JScrollPane scrollPane = new JScrollPane(outputArea);
43     scrollPane.setBounds(10, 180, 560, 180);
44     scrollPane.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
45
46     // Menambahkan scroll pane ke panel bukan area output
47     panel.add(scrollPane);
48
49     // Menambahkan action listener ke tombol execute
50     executeButton.addActionListener(new ActionListener() {
51         @Override
52         public void actionPerformed(ActionEvent e) {
53             String startWord = startText.getText().toLowerCase();
54             String targetWord = targetText.getText().toLowerCase();
55             String algorithm = (String) algorithmBox.getSelectedItem();
56
57             // Memuat daftar kata
58             List<String> wordList = null;
59             try {
60                 wordList = Files.readAllLines(Paths.get("./words_alpha.txt"));
61             } catch (IOException ioException) {
62                 outputArea.setText("Error membaca file daftar kata.");
63                 return;
64             }
65             Set<String> wordSet = new HashSet<>(wordList);
66
67             // Memvalidasi kata
68             if (!wordSet.contains(startWord) || !wordSet.contains(targetWord)) {
69                 outputArea.setText("Error: Start Word dan End word harus merupakan kata-kata berbahasa inggris.");
70                 return;
71             }
72             if (startWord.length() != targetWord.length()) {
73                 outputArea.setText("Error: Start Word dan End word harus mempunyai panjang kata yang sama.");
74                 return;
75             }
76
77             if (startWord.equals(targetWord)) {
78                 outputArea.setText("Start Word dan End Word tidak boleh sama.");
79                 return;
80             }
81
82             // Menjalankan algoritma yang dipilih
83             long startTime = System.nanoTime();
84             WordLadderResult result;
85             switch (algorithm) {
86                 case "Astar":
87                     result = Astar.findWordLadderGUI(startWord, targetWord, wordSet);
88                     break;
89                 case "Greedy":
90                     result = Greedy.findWordLadderGUI(startWord, targetWord, wordSet);
91                     break;
92                 case "UCS":
93                     result = UCS.findWordLadderGUI(startWord, targetWord, wordSet);
94                     break;
95                 default:
96                     throw new IllegalArgumentException("Invalid algorithm: " + algorithm);
97             }
98             long endTime = System.nanoTime();
99             long duration = (endTime - startTime) / 1_000_000;
100
101             // Menampilkan waktu eksekusi dan hasil
102             String resultStr;
103             if (algorithm.equals("Astar")) {
104                 resultStr = WordLadderUtils.printWordLadderGUI(result);
105             } else {
106                 resultStr = WordLadderUtils.printWordLadderGUI(result);
107             }
108             outputArea.setText("Waktu Eksekusi: " + duration + " ms\n" + resultStr);
109         }
110     });
111 }
112 }
```

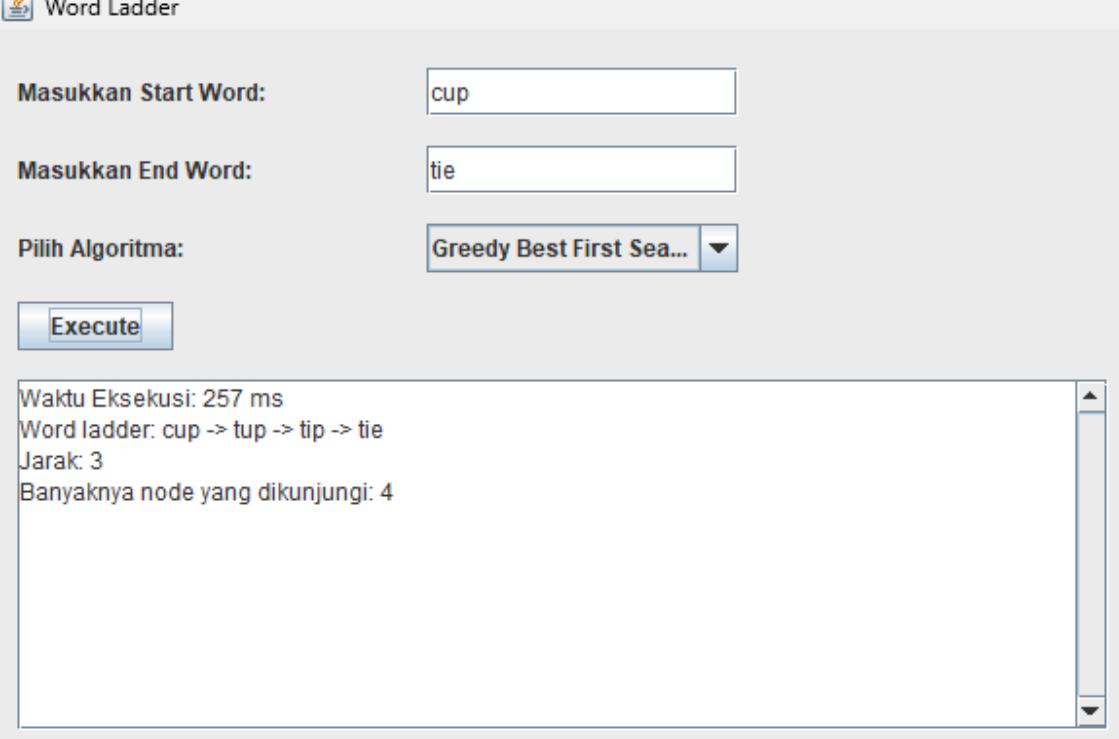
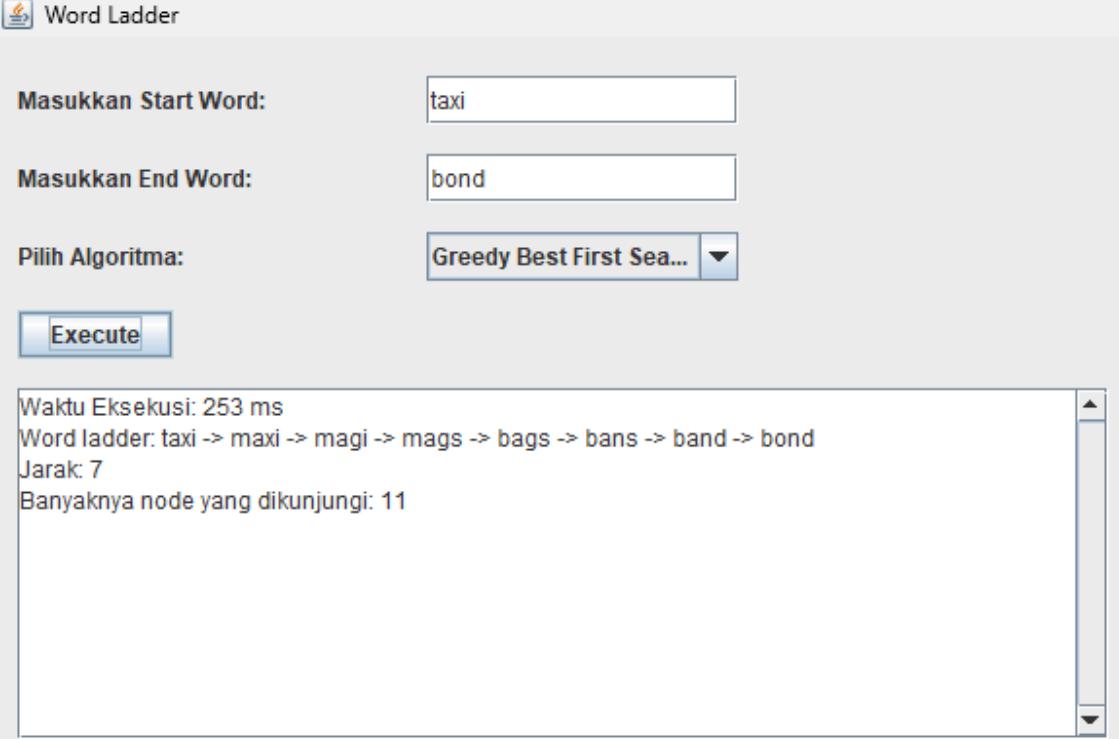
4.3. Input Output Program

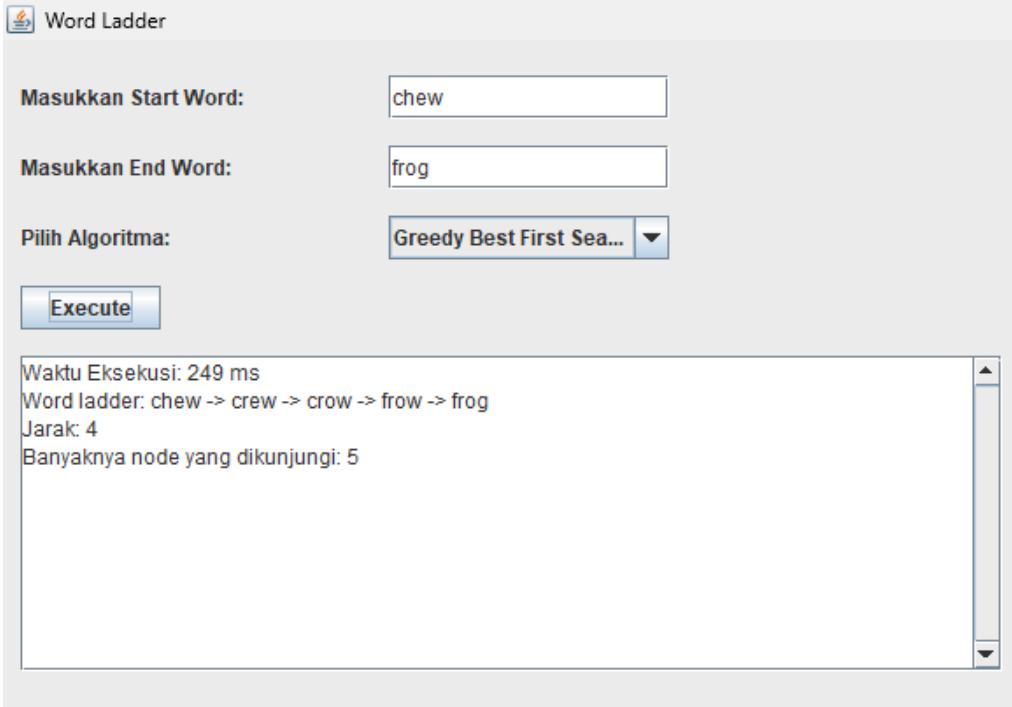
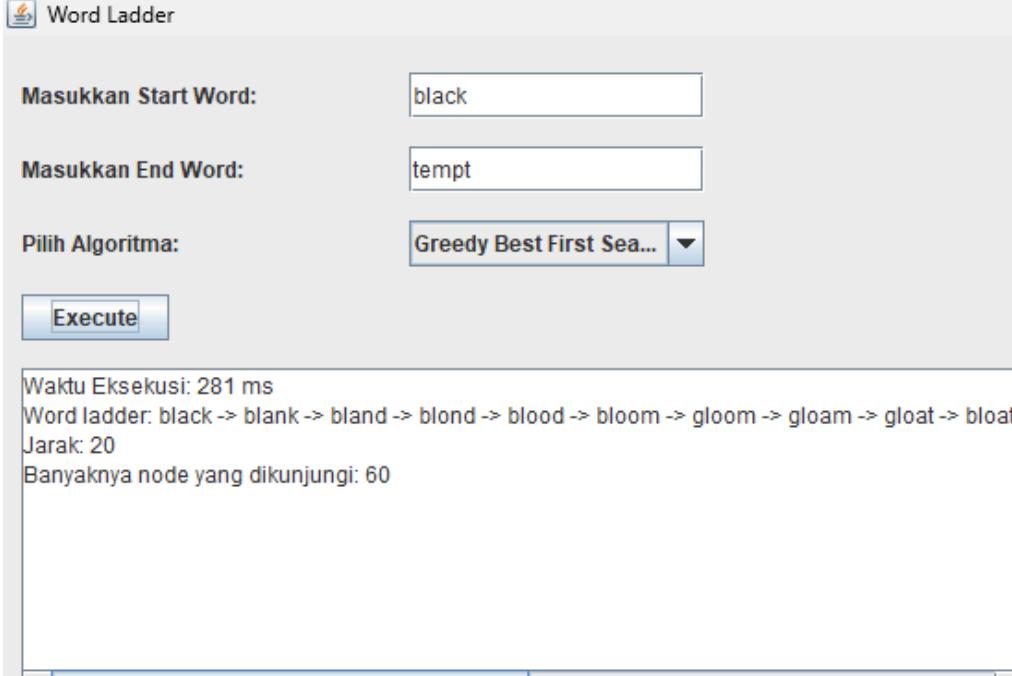
No	Input	Output
Algo		A*
1	Start Word : is End Word : to	<p> Word Ladder</p> <p>Masukkan Start Word: <input type="text" value="is"/></p> <p>Masukkan End Word: <input type="text" value="to"/></p> <p>Pilih Algoritma: <input style="width: 100px; height: 20px; border: 1px solid #ccc; padding: 2px; margin-bottom: 5px;" type="text" value="A*"/> ▾</p> <p><input style="width: 100px; height: 30px; border: 1px solid #ccc; background-color: #f0f0f0; border-radius: 5px; font-size: 12px; font-weight: bold; margin-bottom: 10px;" type="button" value="Execute"/></p> <pre>Waktu Eksekusi: 265 ms Word ladder: is -> as -> aa -> ta -> to Jarak: 4 Banyaknya node yang dikunjungi: 31</pre>

2	Start Word : cup End Word : tie	<p> Word Ladder</p> <p>Masukkan Start Word: <input type="text" value="cup"/></p> <p>Masukkan End Word: <input type="text" value="tie"/></p> <p>Pilih Algoritma: <input type="text" value="A*"/></p> <p><input type="button" value="Execute"/></p> <pre>Waktu Eksekusi: 291 ms Word ladder: cup -> tup -> tip -> tie Jarak: 3 Banyaknya node yang dikunjungi: 5</pre>
3	Start Word : taxi End Word :bond	<p> Word Ladder</p> <p>Masukkan Start Word: <input type="text" value="taxi"/></p> <p>Masukkan End Word: <input type="text" value="bond"/></p> <p>Pilih Algoritma: <input type="text" value="A*"/></p> <p><input type="button" value="Execute"/></p> <pre>Waktu Eksekusi: 244 ms Word ladder: taxi -> taxa -> tala -> tola -> told -> bold -> bond Jarak: 6 Banyaknya node yang dikunjungi: 23</pre>

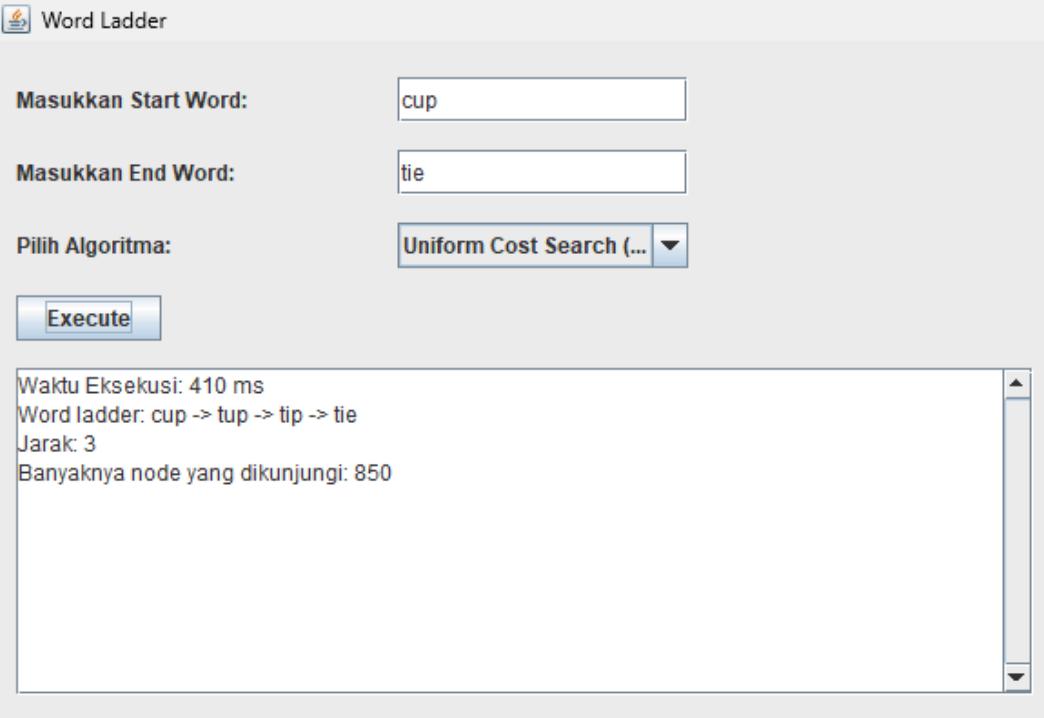
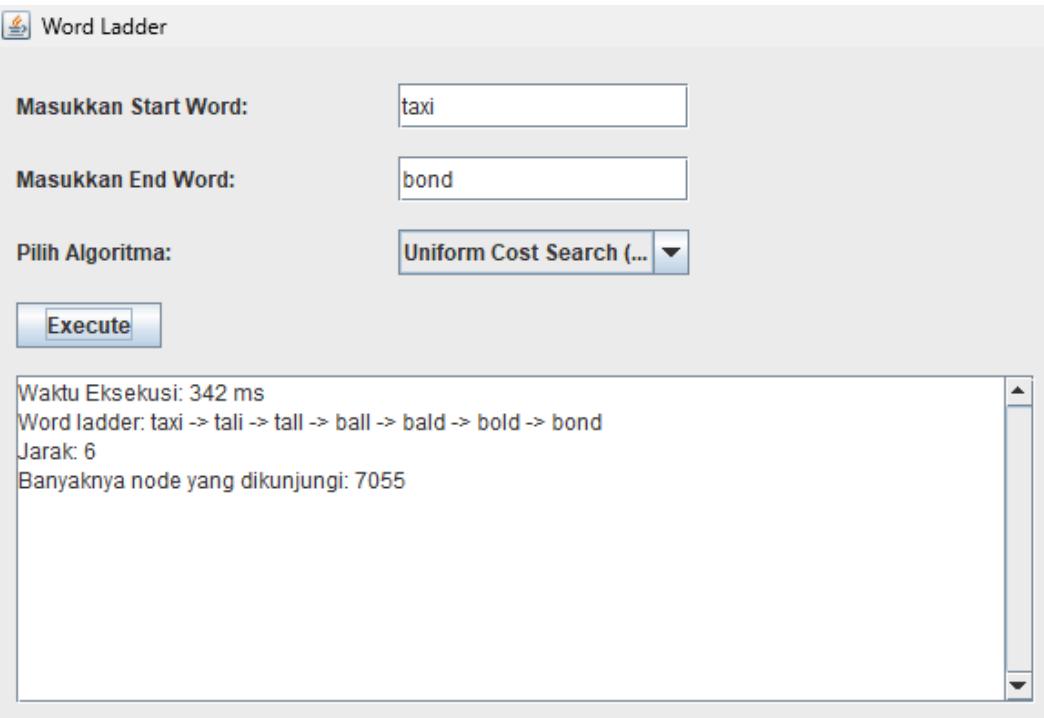
4	Start Word : chew End Word : frog	 <p>Word Ladder</p> <p>Masukkan Start Word: <input type="text" value="chew"/></p> <p>Masukkan End Word: <input type="text" value="frog"/></p> <p>Pilih Algoritma: <input type="text" value="A*"/>▼</p> <p>Execute</p> <p>Waktu Eksekusi: 277 ms Word ladder: chew -> crew -> crow -> frow -> frog Jarak: 4 Banyaknya node yang dikunjungi: 6</p>
5	Start Word : black End Word : tempt	 <p>Word Ladder</p> <p>Masukkan Start Word: <input type="text" value="black"/></p> <p>Masukkan End Word: <input type="text" value="tempt"/></p> <p>Pilih Algoritma: <input type="text" value="A*"/>▼</p> <p>Execute</p> <p>Waktu Eksekusi: 282 ms Word ladder: black -> blank -> brank -> brans -> beans -> leans -> leaps -> heaps -> hemps -> tempt Jarak: 10 Banyaknya node yang dikunjungi: 933</p>

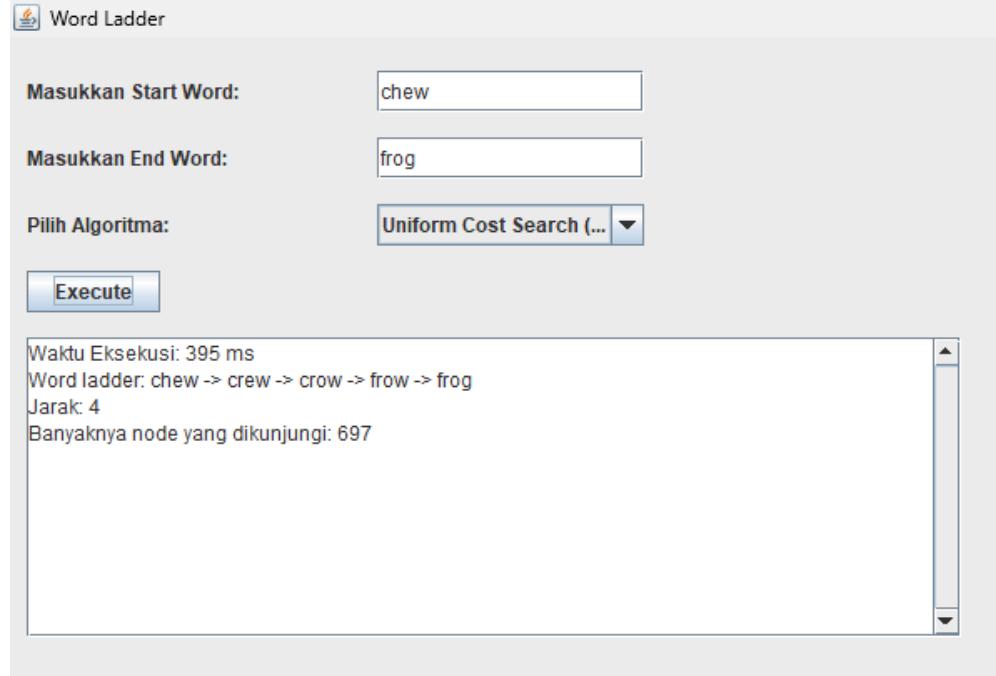
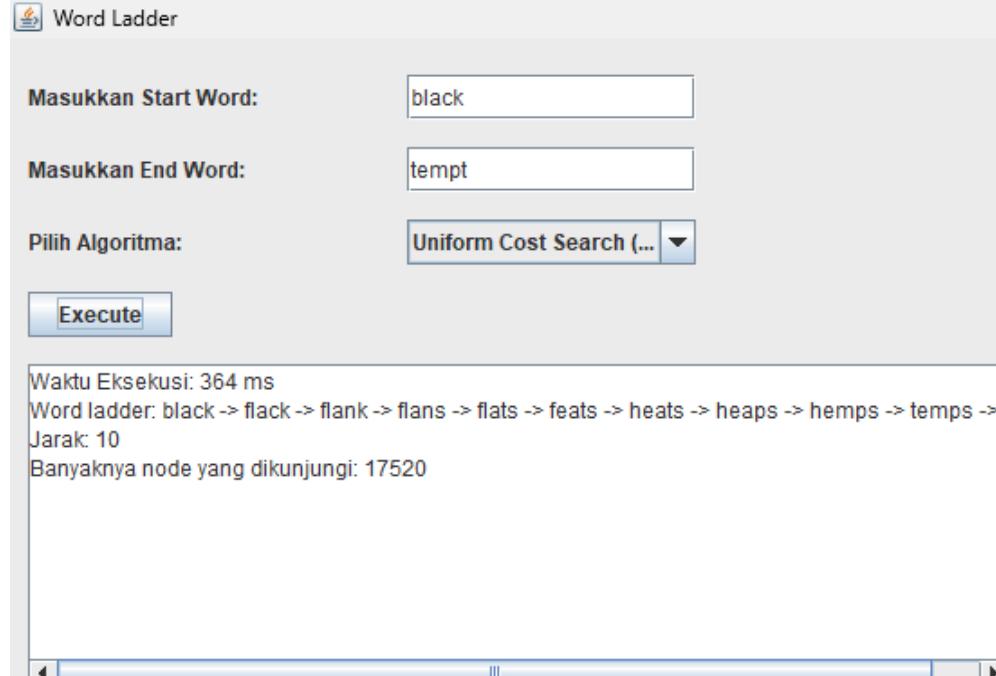
6	Start Word :change End Word : balloon	 <p>Word Ladder</p> <p>Masukkan Start Word: change</p> <p>Masukkan End Word: balloon</p> <p>Pilih Algoritma: A*</p> <p>Execute</p> <p>Waktu Eksekusi: 350 ms Word ladder: change -> changs -> chants -> chints -> chines -> chiles -> whiles -> whiled -> wailed Jarak: 13 Banyaknya node yang dikunjungi: 666</p>
Algo		Greedy Best First Search
1	Start Word : is End Word : to	 <p>Word Ladder</p> <p>Masukkan Start Word: is</p> <p>Masukkan End Word: to</p> <p>Pilih Algoritma: Greedy Best First Sea...</p> <p>Execute</p> <p>Waktu Eksekusi: 279 ms Word ladder: is -> as -> ay -> ax -> aw -> at -> ar -> an -> am -> al -> ai -> ti -> to Jarak: 12 Banyaknya node yang dikunjungi: 22</p>

2	Start Word : cup End Word : tie	 <p>Word Ladder</p> <p>Masukkan Start Word: <input type="text" value="cup"/></p> <p>Masukkan End Word: <input type="text" value="tie"/></p> <p>Pilih Algoritma: <input type="button" value="Greedy Best First Sea... ▾"/></p> <p><input type="button" value="Execute"/></p> <pre> Waktu Eksekusi: 257 ms Word ladder: cup -> tup -> tip -> tie Jarak: 3 Banyaknya node yang dikunjungi: 4 </pre>
3	Start Word : taxi End Word :bond	 <p>Word Ladder</p> <p>Masukkan Start Word: <input type="text" value="taxi"/></p> <p>Masukkan End Word: <input type="text" value="bond"/></p> <p>Pilih Algoritma: <input type="button" value="Greedy Best First Sea... ▾"/></p> <p><input type="button" value="Execute"/></p> <pre> Waktu Eksekusi: 253 ms Word ladder: taxi -> maxi -> magi -> mags -> bags -> bans -> band -> bond Jarak: 7 Banyaknya node yang dikunjungi: 11 </pre>

4	Start Word : chew End Word : frog	 <p>Word Ladder</p> <p>Masukkan Start Word: <input type="text" value="chew"/></p> <p>Masukkan End Word: <input type="text" value="frog"/></p> <p>Pilih Algoritma: Greedy Best First Sea... ▾</p> <p>Execute</p> <p>Waktu Eksekusi: 249 ms Word ladder: chew -> crew -> crow -> frow -> frog Jarak: 4 Banyaknya node yang dikunjungi: 5</p>
5	Start Word : black End Word : tempt	 <p>Word Ladder</p> <p>Masukkan Start Word: <input type="text" value="black"/></p> <p>Masukkan End Word: <input type="text" value="tempt"/></p> <p>Pilih Algoritma: Greedy Best First Sea... ▾</p> <p>Execute</p> <p>Waktu Eksekusi: 281 ms Word ladder: black -> blank -> bland -> blond -> blood -> bloom -> gloom -> gloam -> gloat -> bloat Jarak: 20 Banyaknya node yang dikunjungi: 60</p>

6	Start Word :change End Word : balloon	<p> Word Ladder</p> <p>Masukkan Start Word: <input type="text" value="change"/></p> <p>Masukkan End Word: <input type="text" value="balloon"/></p> <p>Pilih Algoritma: <input type="button" value="Greedy Best First Sea... ▾"/></p> <p><input type="button" value="Execute"/></p> <pre>Waktu Eksekusi: 280 ms Word ladder: change -> changs -> bhangs -> whangs -> wrangs -> wrongs -> wrings -> brings -> br Jarak: 26 Banyaknya node yang dikunjungi: 160</pre>
Algo	Uniform Cost Search (UCS)	
1	Start Word : is End Word : to	

2	Start Word : cup End Word : tie	 <p>Word Ladder</p> <p>Masukkan Start Word: <input type="text" value="cup"/></p> <p>Masukkan End Word: <input type="text" value="tie"/></p> <p>Pilih Algoritma: <input type="button" value="Uniform Cost Search (...) ▾"/></p> <p><input type="button" value="Execute"/></p> <pre> Waktu Eksekusi: 410 ms Word ladder: cup -> tup -> tip -> tie Jarak: 3 Banyaknya node yang dikunjungi: 850 </pre>
3	Start Word : taxi End Word :bond	 <p>Word Ladder</p> <p>Masukkan Start Word: <input type="text" value="taxi"/></p> <p>Masukkan End Word: <input type="text" value="bond"/></p> <p>Pilih Algoritma: <input type="button" value="Uniform Cost Search (...) ▾"/></p> <p><input type="button" value="Execute"/></p> <pre> Waktu Eksekusi: 342 ms Word ladder: taxi -> tali -> tall -> ball -> bald -> bold -> bond Jarak: 6 Banyaknya node yang dikunjungi: 7055 </pre>

4	Start Word : chew End Word : frog	 <p>Word Ladder</p> <p>Masukkan Start Word: <input type="text" value="chew"/></p> <p>Masukkan End Word: <input type="text" value="frog"/></p> <p>Pilih Algoritma: <input type="button" value="Uniform Cost Search (...) ▾"/></p> <p><input type="button" value="Execute"/></p> <pre> Waktu Eksekusi: 395 ms Word ladder: chew -> crew -> crow -> frow -> frog Jarak: 4 Banyaknya node yang dikunjungi: 697 </pre>
5	Start Word : black End Word : tempt	 <p>Word Ladder</p> <p>Masukkan Start Word: <input type="text" value="black"/></p> <p>Masukkan End Word: <input type="text" value="tempt"/></p> <p>Pilih Algoritma: <input type="button" value="Uniform Cost Search (...) ▾"/></p> <p><input type="button" value="Execute"/></p> <pre> Waktu Eksekusi: 364 ms Word ladder: black -> flack -> flank -> flans -> flats -> feats -> heats -> heaps -> hemps -> temps -> tempt Jarak: 10 Banyaknya node yang dikunjungi: 17520 </pre>

6

Start Word :change
End Word : balloon

Word Ladder

Masukkan Start Word:

Masukkan End Word:

Pilih Algoritma:

Waktu Eksekusi: 420 ms
Word ladder: change -> changs -> chants -> chints -> chines -> chined -> shined -> sained -> sailed
Jarak: 13
Banyaknya node yang dikunjungi: 16126

BAB V

Analisis Hasil

5.1. Analisis Hasil Optimalitas dan Waktu Eksekusi

UCS menjamin menemukan solusi optimal (jalur terpendek) jika solusi tersedia karena mengeksplorasi semua node dengan cara breadth-first dan memilih jalur dengan biaya kumulatif terkecil. Namun, pendekatan ini dapat menjadi tidak efisien pada ruang pencarian yang besar karena harus mengeksplorasi semua node secara merata. Dalam kasus terburuk, waktu eksekusi UCS dapat mencapai $O(b^d)$, di mana b adalah faktor percabangan (jumlah node tetangga pada setiap node) dan d adalah kedalaman solusi optimal. Meskipun demikian, UCS masih sangat berguna dalam situasi di mana kita perlu menjamin solusi optimal dan tidak memiliki informasi heuristik yang baik untuk membantu memandu pencarian. Selain itu, UCS juga dapat menjadi dasar untuk algoritma pencarian yang lebih canggih seperti A*, yang menambahkan komponen heuristik untuk meningkatkan efisiensi tanpa mengorbankan optimalitas.

Di sisi lain, Greedy Best First Search atau BFS tidak menjamin optimalitas solusi. Algoritma ini hanya mempertimbangkan nilai heuristik saat memilih node berikutnya untuk dieksplorasi, tanpa memperhitungkan biaya kumulatif yang sudah ditempuh. Meskipun dapat menemukan solusi dengan cepat jika heuristiknya akurat dan tidak ada celah pada solusi, GBFS dapat terjebak pada solusi sub-optimal atau bahkan gagal menemukan solusi jika heuristiknya tidak akurat atau terdapat celah pada solusi. Waktu eksekusi GBFS bergantung pada akurasi heuristik, tetapi dalam kasus terbaik, dapat mencapai $O(b^*m)$, di mana m adalah panjang solusi optimal. Meskipun demikian, GBFS bisa sangat efektif dalam situasi di mana kecepatan mencapai tujuan lebih penting daripada menemukan jalur terpendek. Selain itu, GBFS bisa sangat efisien jika heuristik yang digunakan cukup akurat dan dapat memandu pencarian ke arah yang benar.

Algoritma A* menawarkan keseimbangan antara optimalitas dan kecepatan dengan bantuan fungsi heuristik. A* menjamin solusi optimal jika fungsi heuristiknya admissible (tidak pernah overestimate jarak ke solusi). Algoritma ini mempertimbangkan baik biaya kumulatif maupun perkiraan biaya ke tujuan, sehingga lebih efisien daripada UCS dengan bantuan fungsi

heuristik. Namun, A* lebih kompleks daripada UCS dan Greedy Best First Search, dan memerlukan perancangan fungsi heuristik yang baik untuk kinerja yang optimal. Waktu eksekusi A* bergantung pada kualitas heuristik, tetapi dalam kasus terbaik, dapat mencapai $O(bm)$, di mana m adalah panjang solusi optimal. Namun, perlu diingat bahwa A juga memiliki kelemahan dalam hal penggunaan memori. Karena A* harus menyimpan semua node yang telah dieksplor dan yang belum dieksplor dalam memori, ini bisa menjadi masalah dalam ruang pencarian yang sangat besar.

Jika optimalitas solusi adalah prioritas utama, UCS dan A* (dengan heuristik admissible) menjamin solusi optimal. Namun, jika kecepatan pencarian lebih penting daripada optimalitas, GBFS dapat menjadi pilihan yang baik jika heuristiknya akurat. A* menawarkan keseimbangan antara keduanya, tetapi memerlukan perancangan yang lebih rumit.

Berikut adalah hasil dari ketiga algoritma dalam melakukan pencarian.

Masukkan Start Word:	<input type="text" value="tray"/>
Masukkan End Word:	<input type="text" value="bomb"/>
Pilih Algoritma:	<input type="button" value="Uniform Cost Search (...)"/>
<input type="button" value="Execute"/>	
Waktu Eksekusi: 381 ms	
Word ladder: tray -> bray -> bras -> bros -> boos -> boob -> bomb	
Jarak: 6	
Banyaknya node yang dikunjungi: 5655	

Masukkan Start Word:	<input type="text" value="tray"/>
Masukkan End Word:	<input type="text" value="bomb"/>
Pilih Algoritma:	<input type="button" value="Greedy Best First Sea..."/>
<input type="button" value="Execute"/>	
Waktu Eksekusi: 266 ms	
Word ladder: tray -> bray -> bras -> boas -> bows -> boos -> boob -> bomb	
Jarak: 7	
Banyaknya node yang dikunjungi: 18	

Masukkan Start Word:	<input type="text" value="tray"/>
Masukkan End Word:	<input type="text" value="bomb"/>
Pilih Algoritma:	<input type="text" value="A*"/> <input type="button" value="▼"/>
<input type="button" value="Execute"/>	
Waktu Eksekusi: 311 ms	
Word ladder: tray -> bray -> brat -> boat -> boot -> boob -> bomb	
Jarak: 6	
Banyaknya node yang dikunjungi: 93	

Dalam menyelesaikan persoalan Word Ladder, kita dapat melihat perbandingan kinerja antara algoritma Uniform Cost Search (UCS), Greedy Best First Search (GBFS), dan A*. UCS terbukti mampu menghasilkan solusi optimal dengan jarak minimum dari kata awal ke kata target. Namun, untuk mencapai hal tersebut, UCS harus mengunjungi jumlah node yang cukup besar, seperti pada contoh yang mengunjungi 5655 node. Jumlah node yang besar ini berdampak pada waktu eksekusi algoritma UCS yang menjadi lebih lama dibandingkan algoritma lainnya.

Di sisi lain, GBFS mengunjungi jumlah node yang lebih sedikit dibandingkan A* dan UCS. Dengan mengunjungi lebih sedikit node, GBFS mampu memiliki waktu eksekusi yang relatif lebih cepat. Akan tetapi, keunggulan kecepatan ini diimbangi dengan kelemahan utama GBFS, yaitu tidak menjamin optimalitas solusi. Solusi yang dihasilkan GBFS dapat memiliki jarak yang tidak optimal dari kata awal ke kata target.

Algoritma A* hadir sebagai kombinasi terbaik dari kedua algoritma sebelumnya. A* berhasil menghasilkan solusi optimal layaknya UCS, namun dengan mengunjungi jumlah node yang lebih seimbang dibandingkan UCS. Jumlah node yang lebih seimbang ini memungkinkan A* untuk memiliki waktu eksekusi yang tidak terlalu lama. Dengan kata lain, A* mampu mengambil kelebihan utama dari UCS (solusi optimal) dan GBFS (kecepatan dengan mengunjungi lebih sedikit node) secara bersamaan.

5.2. Analisis Hasil Memori yang Dibutuhkan

UCS (Uniform Cost Search) cenderung membutuhkan memori paling banyak di antara ketiganya. Hal ini dikarenakan sifat pencarian breadth-first dari UCS, di mana algoritma harus menyimpan semua node pada setiap level kedalaman di dalam memori sebelum mengeksplorasi lebih lanjut. Pada persoalan Word Ladder dengan kamus kata yang besar, jumlah node yang perlu disimpan oleh UCS akan sangat banyak sebelum menemukan solusi, mengakibatkan kebutuhan memori yang tinggi. Selain itu, UCS juga tidak melakukan pemangkasan terhadap node-node yang tidak menjanjikan, sehingga harus menyimpan lebih banyak node di dalam memori.

Di sisi lain, Greedy Best-First Search atau GBFS membutuhkan memori paling sedikit di antara ketiga algoritma tersebut dalam konteks Word Ladder. Hal ini karena GBFS menggunakan fungsi heuristik (dalam hal ini, jumlah karakter yang berbeda antara kata saat ini dan kata target) untuk memangkas node-node yang tidak menjanjikan. GBFS hanya perlu menyimpan nilai heuristik untuk setiap node, sehingga kebutuhan memorinya lebih rendah dibandingkan UCS dan A*.

Sedangkan A* berada di antara UCS dan GBFS dalam hal kebutuhan memori untuk persoalan Word Ladder. A* membutuhkan lebih banyak memori dibandingkan GBFS karena perlu menyimpan informasi biaya kumulatif dari akar ke setiap node, selain nilai heuristik. Namun, A* lebih hemat memori dibandingkan UCS karena melakukan pemangkasan dengan fungsi heuristik, sehingga tidak menyimpan sebanyak node seperti UCS.

Berikut adalah hasil dari ketiga algoritma dalam melakukan pencarian.

Masukkan Start Word:

Masukkan End Word:

Pilih Algoritma:

Waktu Eksekusi: 37 ms
Word ladder: tray -> bray -> bras -> bros -> boos -> boob -> bomb
Jarak: 6
Banyaknya node yang dikunjungi: 5655
Penggunaan Memori: 14336 Kilobytes

Masukkan Start Word:

Masukkan End Word:

Pilih Algoritma:

```
Waktu Eksekusi: 22 ms
Word ladder: tray -> bray -> bras -> boas -> bows -> boos -> boob -> bomb
Jarak: 7
Banyaknya node yang dikunjungi: 18
Penggunaan Memori: 329 Kilobytes
```

Masukkan Start Word:

Masukkan End Word:

Pilih Algoritma:

```
Waktu Eksekusi: 22 ms
Word ladder: tray -> bray -> brat -> boat -> boot -> boob -> bomb
Jarak: 6
Banyaknya node yang dikunjungi: 93
Penggunaan Memori: 789 Kilobytes
```

BAB VI

Daftar Pustaka dan Link Repository

Berikut adalah daftar referensi yang dipakai dalam pengerajan tugas kecil ini.

1. <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/stima23-24.htm>
2. [PowerPoint Presentation \(itb.ac.id\)](#)
3. [Route-Planning-Bagian2-2021.pdf \(itb.ac.id\)](#)
4. <https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/>
5. <https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/>
6. <https://www.geeksforgeeks.org/a-search-algorithm/>

Link Repository GitHub : [ChrisCS50X/Tucil3_13522135 \(github.com\)](https://github.com/ChrisCS50X/Tucil3_13522135)

BAB VII

Lampiran

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* optimasi	✓	
7. [Bonus]: Program memiliki tampilan GUI	✓	