Tugas Kecil 1 IF2211 Strategi Algoritma

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

Semester II Tahun 2023/2024

Disusun oleh

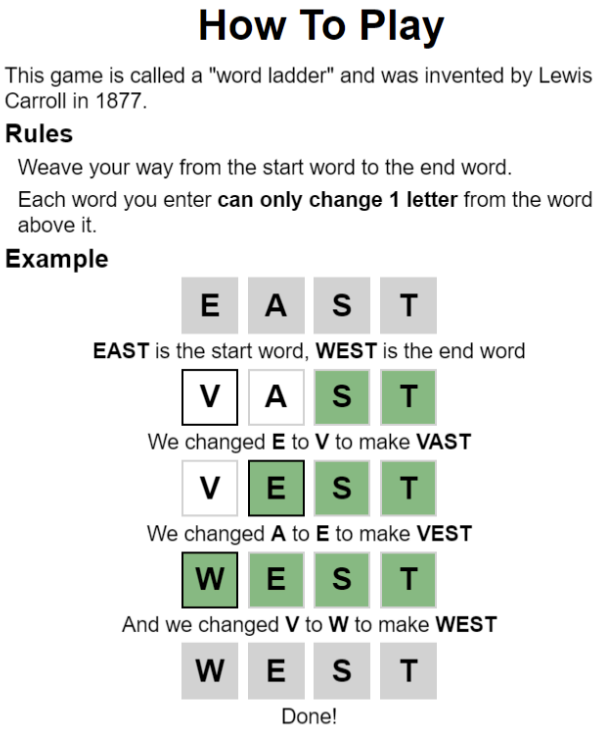
Shafiq Irvansyah : 13522003

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2024

Daftar Isi

I. Deskripsi Tugas

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.



Gambar 1. Ilustrasi dan Peraturan Permainan Word Ladder (Sumber: https://wordwormdormdork.com/)

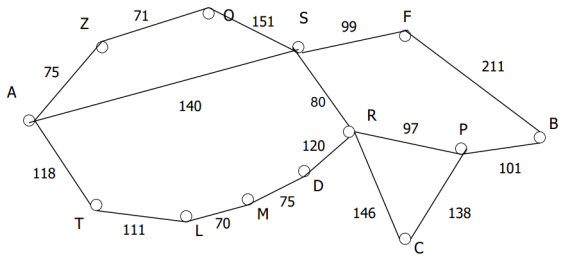
II. Landasan Teori

A. Uniform Cost Search

Uniform Cost Search adalah suatu algoritma pencarian rute yang menggunakan teknik pembentukan pohon ruang status. Perbedaan mendasar antara algoritma BFS dan DFS dengan Uniform Cost Search (UCS) terletak pada cara penanganan bobot atau cost di antara simpul-simpul. BFS dan DFS tidak mempertimbangkan bobot pada setiap sisi antara simpul-simpul, sementara UCS memperhitungkan bobot ini untuk menentukan rute terpendek atau tercepat. Pada BFS dan DFS, bobot dianggap sama untuk setiap sisi, misalnya 1, sehingga semua sisi memiliki bobot yang sama.

Simpul-E	Simpul Hidup
A	Z_{A-75}, T_{A-118}, S_{A-140}
Z_{A-75}	T_{A-118}, S_{A-140},O_{AZ-146}
T_{A-118}	S_{A-140},O_{AZ-146},L_{AT-229}
S_{A-140}	O_{AZ-146},R_{AS-220}, L_{AT-229},F_{AS-239},O_{AS-291}
O_{AZ-146}	R_{AS-220}, L_{AT-229}, F_{AS-239},O_{AS-291}
R_{AS-220}	L_{AT-229}, F_{AS-239},O_{AS-291}, $P_{ASR-317}$,$D_{ASR-340}$,$C_{ASR-366}$
L_{AT-229}	F_{AS-239},O_{AS-291},$M_{ATL-299}$, $P_{ASR-317}$,$D_{ASR-340}$,$C_{ASR-366}$
F_{AS-239}	O_{AS-291},$M_{ATL-299}$, $P_{ASR-317}$,$D_{ASR-340}$,$C_{ASR-366}$, $B_{ASF-450}$
O_{AS-291}	$M_{ATL-299}$, $P_{ASR-317}$,$D_{ASR-340}$,$C_{ASR-366}$, $B_{ASF-450}$
$M_{ATL-299}$	$P_{ASR-317}$,$D_{ASR-340}$,$D_{ATLM-364}$,$C_{ASR-366}$, $B_{ASF-450}$
$P_{ASR-317}$	$D_{ASR-340}$,$D_{ATLM-364}$,$C_{ASR-366}$, $B_{ASRP-418}$, $C_{ASRP-455}$, $B_{ASF-450}$
$D_{ASR-340}$	$D_{ATLM-364}$,$C_{ASR-366}$, $B_{ASRP-418}$, $C_{ASRP-455}$, $B_{ASF-450}$
$D_{ATLM-364}$	$C_{ASR-366}$, $B_{ASRP-418}$, $C_{ASRP-455}$, $B_{ASF-450}$
$C_{ASR-366}$	$B_{ASRP-418}$, $C_{ASRP-455}$, $B_{ASF-450}$
B	Solusi ketemu

- $g(n)$ = path cost from root to n



Gambar 2. Contoh Pencarian Secara UCS

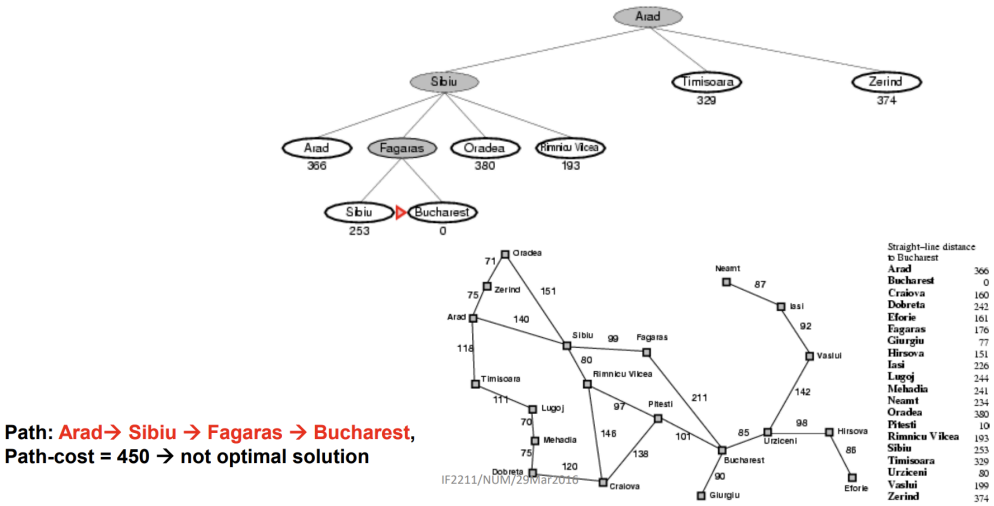
UCS menggunakan fungsi (g(n)) untuk mengevaluasi cost dari simpul yang diekspansi dan simpul-simpul tetangganya. Misalnya, dalam contoh grafik yang diberikan, jika kita mencari rute terendah dari simpul A ke simpul B, UCS akan menentukan simpul-A sebagai simpul-E awal, kemudian mengevaluasi simpul-simpul tetangga (misalnya AZ-75, AS-80, dan AT-118) berdasarkan urutan fungsi (g(n)). Proses ini berlanjut hingga simpul B ditemukan sebagai simpul-E.

UCS menjamin optimalitas dalam mencari rute dengan cost paling minimum, yang biasanya merupakan jarak antara dua simpul.

B. Greedy Best First Search

Greedy Best First Search (GBFS) adalah sebuah algoritma pencarian yang mengutamakan simpul dengan nilai heuristik terendah untuk diekspansi

selanjutnya. Dalam GBFS, setiap langkah keputusan didasarkan pada nilai heuristik lokal tanpa mempertimbangkan konsekuensi jangka panjang secara menyeluruh. Heuristik digunakan untuk mengevaluasi setiap simpul dalam ruang pencarian, memberikan perkiraan biaya atau jarak tersisa dari simpul saat ini ke simpul tujuan. Simpul dengan nilai heuristik terendah diberikan prioritas tertinggi untuk diekspansi, sehingga diharapkan dapat mendekati solusi optimal. Nilai heuristik ini memiliki istilah umum yaitu h(n). Meskipun GBFS dapat cepat dalam menemukan solusi, terutama dalam ruang pencarian yang luas, algoritma ini tidak menjamin solusi optimal karena sifat serakahnya. Keterbatasan tersebut membuat GBFS bisa tersesat dalam jalur yang tidak optimal atau terjebak dalam lokal minimum atau maksimum dalam beberapa kasus.



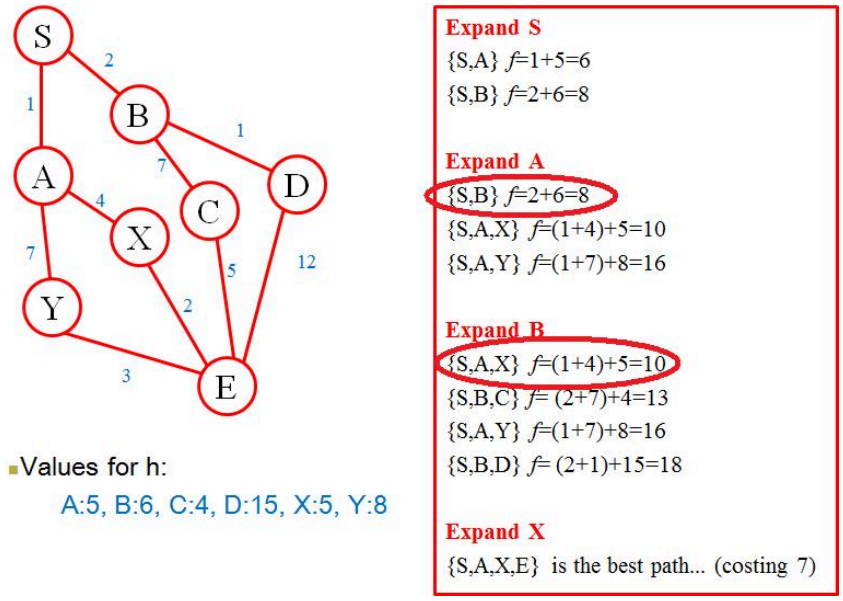
Gambar 3. Contoh Pencarian dengan GBFS

Dalam contoh tersebut, Algoritma GBFS menitikberatkan pada nilai heuristik h(n), yang merupakan garis lurus dari titik saat ini ke tujuan akhir. Meskipun selalu memilih nilai heuristik terendah, algoritma ini cenderung terjebak pada solusi yang hanya optimal secara lokal. Sebagai contoh, dalam pencarian rute dari Iasi ke Oradea, GBFS mungkin akan memilih rute menuju Neamt terlebih dahulu, meskipun jalur tersebut mengarah ke jalan buntu. Hal ini disebabkan oleh perbedaan jarak bersih antara Neamt dan Oradea yang lebih dekat daripada Vaslui dan Oradea.

C. A-Star (A*)

Algoritma pencarian A* adalah salah satu algoritma pencarian yang digunakan untuk menemukan jalur terpendek antara dua titik dalam graf, dengan mempertimbangkan biaya langkah dan nilai heuristik. A* menggabungkan dua komponen utama: biaya sejauh ini dari titik awal ke titik saat ini (dinyatakan dengan g(n) dan estimasi biaya dari titik saat ini ke titik akhir (dinyatakan dengan h(n) . Biaya total dari titik awal ke titik akhir (dinyatakan dengan f(n) dihitung sebagai jumlah dari g(n) dan h(n) . A* menggunakan fungsi heuristik untuk

memberikan perkiraan biaya terdekat dari setiap simpul ke titik akhir. Selama pencarian, A* secara iteratif memilih simpul dengan biaya total terendah f(n) untuk diekspansi berikutnya. Dengan memilih simpul-simpul dengan biaya total terendah, A* dapat menemukan jalur terpendek dengan mempertimbangkan biaya langkah dan estimasi biaya ke titik akhir. Algoritma ini dianggap optimal dan efisien jika fungsi heuristik memenuhi kriteria keoptimalan, seperti admissibility (heuristik tidak melebih-lebihkan biaya sebenarnya) dan konsistensi (heuristik tidak melebihi biaya dari satu simpul ke simpul lain ditambah biaya estimasi dari simpul lain ke tujuan).



Gambar 4. Contoh Pencarian A*

III. Analisis dan Implementasi

Dalam proses pemecahan masalah dan implementasinya, penulis menggunakan solusi yang memanfaatkan array untuk menyimpan path tiap node, visited node, dan informasi lainnya seperti pendekatan pada umumnya. Namun, berdasarkan pernyataan asisten dalam forum Q&A, waktu pemrosesan yang dilakukan oleh asisten terlalu cepat jika menggunakan array, karena pasti memerlukan banyak memory. Oleh karena itu, penulis mencari solusi alternatif dengan menggunakan hashmap untuk mendapatkan path solusi. Hashmap ini akan menyimpan kata sebagai kunci dan kata sebelumnya (parent-nya) sebagai value-nya. Hashmap dari suatu kata hanya akan diganti jika tidak ada entri hashmap untuk kata tersebut atau jika kata selanjutnya memiliki cost yang lebih murah, sehingga cost yang lebih mahal akan digantikan. Dengan pendekatan ini, program tidak perlu menyimpan path dari setiap node yang dihasilkan, tidak perlu mencatat node yang sudah dikunjungi, dan tidak perlu membentuk kelas node baru untuk implementasinya.

Dalam konteks pemrograman dan algoritma pencarian, f(n) merujuk pada fungsi evaluasi total dari suatu node n. Fungsi ini digunakan untuk menilai nilai atau biaya dari setiap node dalam proses pencarian jalur atau solusi optimal. Secara umum, f(n) dapat dianggap sebagai perkiraan total biaya atau nilai yang diperlukan untuk mencapai tujuan dari titik awal melalui node n. Ini bisa mencakup biaya aktual atau jarak yang sudah dilalui g(n) ditambah dengan perkiraan biaya yang diperlukan untuk mencapai tujuan dari node n h(n). Dengan demikian, f(n) memberikan gambaran tentang pentingnya node n dalam mencapai solusi optimal, dan digunakan untuk mengambil keputusan tentang langkah berikutnya dalam algoritma pencarian.

1. Uniform Cost Search

Pada UCS, *f(n) = g(n).* Dalam implementasi program, *g(n)* yang digunakan adalah panjang dari *path* yang sudah ditempuh hingga suatu titik tertentu. Implementasi tersebut dipilih karena *path* yang dianggap optimal adalah jika panjang *path* minimum.

a) Program akan memvalidasi terlebih dahulu *word* yang dimasukkan user, jika tidak valid maka user akan diminta ulang
b) Program memasukkan *word* awal kedalam *priority queue* dan memasukkan *g(n)* = 0 ke dalam *hashmap costSoFar* kemudian akan dilakukan proses looping yang akan berhenti jika *priority queue* habis atau *word* akhir sudah ditemukan
c) *Priority queue* otomatis mengsortir head queue berdasarkan *g(n)* terkecil, program akan men-*dequeue head queue* kemudian memprosesnya sebagai *current word*
d) Jika *current word* tidak sama dengan *word* akhir, maka akan dibangkitkan semua tetangga dari *current word*. Pada tiap pembangkitan semua tetangga dari *current word*, akan dimasukkan kedalam hashmap *cameFrom*, dimana key adalah tetangganya itu sendiri dan valuenya adalah *current word nya. g(n)* tetangganya adalah *g(n) current word* + 1, kemudian dimasukkan kedalam *hashmap*

costSoFar. Current word lanjut ke antrian berikutnya (ulangi tahap c–d hingga queue habis/ketemu).
Jika *current word* sama dengan *word* akhir, maka proses pencarian dihentikan. Program akan mengkonstruksi *path* dimulai dari *currentWord* dengan memanfaatkan *hashmap cameFrom.*

e) Program menampilkan visited node, lama waktu proses, dan *path*-nya jika ada.

2. Greedy Best First Search

Pada GBFS, *f(n) = h(n).* Dalam implementasi program, *h(n)* yang digunakan adalah jumlah karakter yang berbeda antara *word* akhir dengan *current word*. Implementasi tersebut dipilih karena pendekatan heuristik tersebut dipilih karena menganggap jumlah karakter yang berbeda setara dengan *best case scenario* dengan langkah yang mungkin diambil.

a) Program akan memvalidasi terlebih dahulu *word* yang dimasukkan user, jika tidak valid maka user akan diminta ulang
b) Program memasukkan *word* awal kedalam *priority queue* dan memasukkan *h(n)* = 0 ke dalam *hashmap costs*, kemudian akan dilakukan proses looping yang akan berhenti jika *priority queue* habis atau *word* akhir sudah ditemukan
c) *Priority queue* otomatis mengsortir head queue berdasarkan *h(n)* terkecil, program akan men-*dequeue head queue* kemudian memprosesnya sebagai *current word*
d) Jika *current word* tidak sama dengan *word* akhir, maka akan dibangkitkan semua tetangga dari *current word*. Pada tiap pembangkitan semua tetangga dari *current word*, akan dimasukkan kedalam hashmap *cameFrom*, dimana key adalah tetangganya itu sendiri dan valuenya adalah *current word nya. h(n)* tetangganya kemudian dimasukkan kedalam *hashmap costs. Current word* lanjut ke antrian berikutnya (ulangi tahap c–d hingga queue habis/ketemu)
Jika *current word* sama dengan *word* akhir, maka proses pencarian dihentikan. Program akan mengkonstruksi *path* dimulai dari *currentWord* dengan memanfaatkan *hashmap cameFrom.*
e) Program menampilkan visited node, lama waktu proses, dan *path*-nya jika ada.

3. A-Star

Pada A*, *f(n) = g(n) + h(n).* Dalam implementasi program, *g(n)* yang digunakan adalah panjang dari *path* yang sudah ditempuh hingga suatu titik tertentu dan *h(n)* yang digunakan adalah jumlah karakter yang berbeda antara *word* akhir dengan *current word*. Implementasi tersebut dipilih karena heuristik yang digunakan admissible. Heuristik *h(n)* tidak akan melebih-lebihkan biayanya karena menggambarkan biaya dalam skenario terbaik, sementara *g(n)* akan selalu mencerminkan kondisi aslinya.

a) Program akan memvalidasi terlebih dahulu *word* yang dimasukkan user, jika tidak valid maka user akan diminta ulang

b) Program memasukkan *word* awal kedalam *priority queue* dan memasukkan *f(n)* = 0 ke dalam *hashmap costTotal* dan *g(n)* ke dalam *hasmap costSoFar*. Kemudian akan dilakukan proses looping yang akan berhenti jika *priority queue* habis atau *word* akhir sudah ditemukan

c) *Priority queue* otomatis mengsortir head queue berdasarkan *f(n)* terkecil, program akan men-*dequeue head queue* kemudian memprosesnya sebagai *current word*

d) Jika *current word* tidak sama dengan *word* akhir, maka akan dibangkitkagn semua tetangga dari *current word*. Pada tiap pembangkitan semua tetangga dari *current word*, akan dimasukkan kedalam hashmap *cameFrom*, dimana key adalah tetangganya itu sendiri dan valuenya adalah *current word nya. g(n)* tiap tetangga yang merupakan *current word g(n)* + 1 dimasukkan ke dalam *hashamp costSoFar,* beserta juga dengan *f(n) = g(n) +h(n)* dari tetangga yang dimasukkan ke dalam *hashmap costs. Current word* lanjut ke antrian berikutnya (ulangi tahap c–d hingga queue habis/ketemu)

   Jika *current word* sama dengan *word* akhir, maka proses pencarian dihentikan. Program akan mengkonstruksi *path* dimulai dari *currentWord* dengan memanfaatkan *hashmap cameFrom.*

e) Program menampilkan visited node, lama waktu proses, dan *path*-nya jika ada.

IV. Kode Sumber

Kode sumber berikut hanya terdiri dari algoritma utama, tidak termasuk kode pembangun CLI dan GUI.

No.	Class
1.	**WordLadder.java** Berfungsi sebagai parent dari semua algoritma pencarian

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.util.*;

public class WordLadder {
    // Membaca file tree.txt dan membuat map dari string ke list of string
                    protected static Map<String, List<String>> wordMap =
createWordMap("test/tree.txt");

    private static Map<String, List<String>> createWordMap(String filename) {
        Map<String, List<String>> result = new HashMap<>();
        try {
            BufferedReader reader = new BufferedReader(new FileReader(filename));
            String line;
            while ((line = reader.readLine()) != null) {
                String[] words = line.split(" ");
                String root = words[0];
                        List<String> children = Arrays.asList(Arrays.copyOfRange(words, 1,
words.length));
                result.put(root, children);
            }
            reader.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return result;
    }

    // Mengecek apakah string valid (tidaka ada kareakter selain huruf)
    public static Boolean isStringValid(String word) {
        return wordMap.containsKey(word);
    }

}
```

2.	**UniformCostSearch.java** Pencarian dengan algoritma UCS

```
import java.util.*;

public class UniformCostSearch extends WordLadder {
   public static List<String> ucs(String start, String end) {
      // Jarak dari root sampai current, akan diambil yang paling kecil pada tiap
      Map<String, Integer> costSoFar = new HashMap<>();
      Map<String, String> cameFrom = new HashMap<>();
                                          PriorityQueue<String>      queue      =      new
PriorityQueue<>(Comparator.comparingInt(costSoFar::get));
      queue.add(start);
      costSoFar.put(start, 0);
      Integer count = 0;

      // Iterasi sampai queue kosong/ketemu
      while (!queue.isEmpty()) {
         count++;
         String current = queue.poll();
         // Kondisi Ketemu
         if (current.equals(end)) {
            List<String> path = reconstructPath(cameFrom, current);
            path.addFirst(count.toString());
            return path;
         }
         // Kondisi lanjut
         for (String next : wordMap.getOrDefault(current, Collections.emptyList())) {
            int newCost = costSoFar.get(current) + 1;
            if (!costSoFar.containsKey(next) || newCost < costSoFar.get(next)) {
               costSoFar.put(next, newCost);
               cameFrom.put(next, current);
               queue.add(next);
            }
         }
      }
      // Kondisi ga ketemu sama sekali
      List<String> path = new ArrayList<>();
      path.add(count.toString());
      return path; // Path not found
   }

    private static List<String> reconstructPath(Map<String, String> cameFrom, String
current) {
      List<String> path = new ArrayList<>();
      while (current != null) {
         path.add(current);
         current = cameFrom.get(current);
      }
      Collections.reverse(path);
      return path;
   }
```

	}
3	**BestFirstSearch.java** Pencarian dengan algoritma Greedy-Best First Search

```
import java.util.*;

public class BestFirstSearch extends WordLadder {
    // Fungsi untuk mencari jumlah karakter yang berbeda antara dua string, akan
    // digunakan sebagai nilai h(n)
    public static Integer getDiffChar(String curr, String target) {
        int count = 0;
        for (int i = 0; i < curr.length(); i++) {
            if (curr.charAt(i) != target.charAt(i)) {
                count++;
            }
        }
        return count;
    }

    public static List<String> bfs(String start, String end) {
        // Jarak dari current sampai end word berdasarkan perbedaan jumlah karakter,
        // akan digunakan sebagai nilai h(n)
        Map<String, Integer> costs = new HashMap<>();
        Map<String, String> cameFrom = new HashMap<>();
                                            PriorityQueue<String>      queue      =      new
PriorityQueue<>(Comparator.comparingInt(costs::get));
        queue.add(start);
        costs.put(start, 0);
        Integer count = 0;

        // Iterasi sampai queue kosong/ketemu
        while (!queue.isEmpty()) {
            count++;
            String current = queue.poll();

            // Kondisi Ketemu
            if (current.equals(end)) {
                List<String> path = reconstructPath(cameFrom, current);
                path.addFirst(count.toString());
                return path;
            }
            // Kondisi lanjut
            for (String next : wordMap.getOrDefault(current, Collections.emptyList())) {
                int newCost = getDiffChar(next, end);
                if (!costs.containsKey(next) || newCost < costs.get(next)) {
                    costs.put(next, newCost);
                    cameFrom.put(next, current);
```

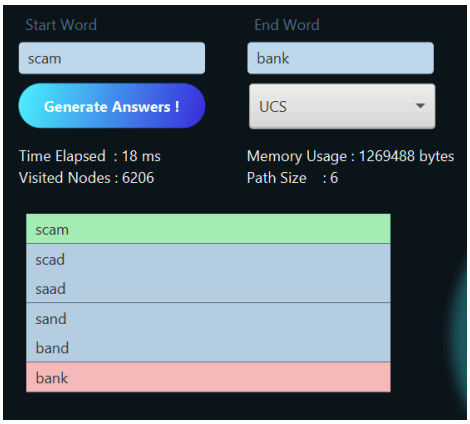
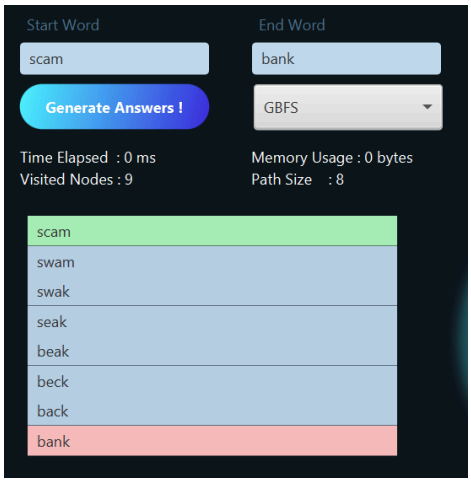
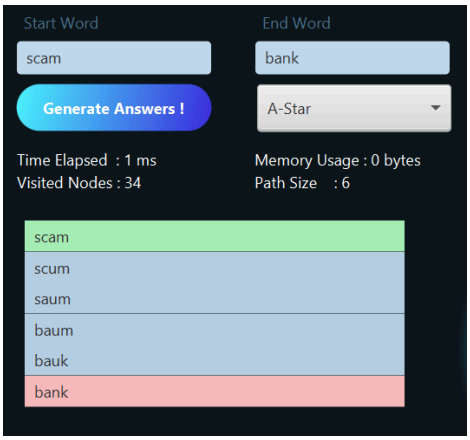
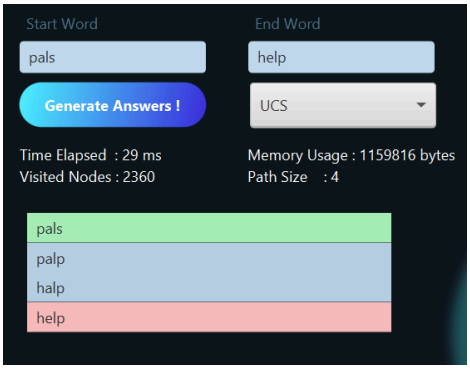
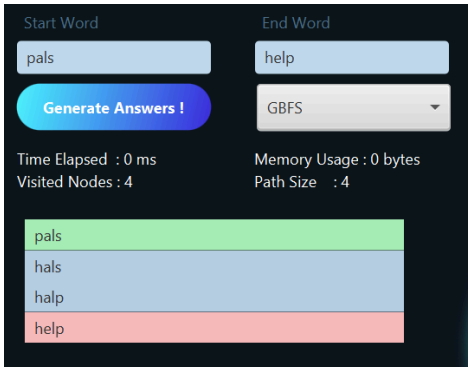
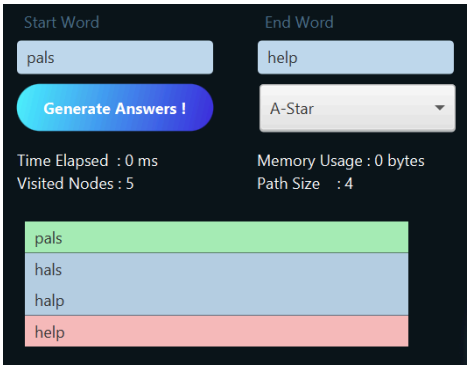
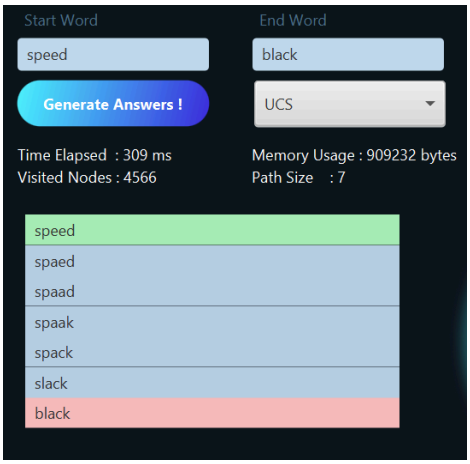
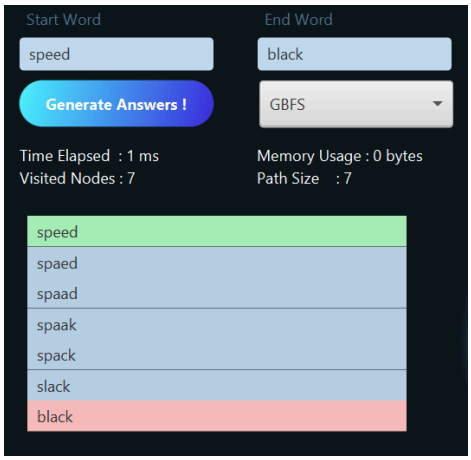
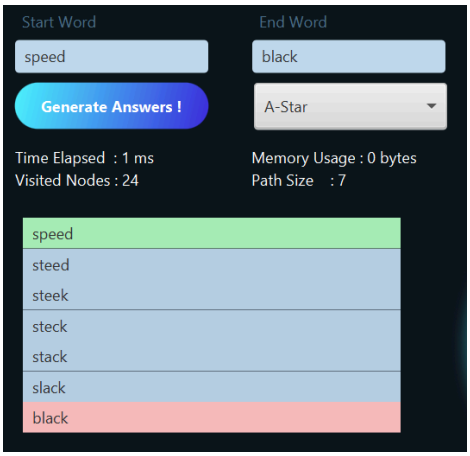
```
                queue.add(next);
            }
        }
    }
    // Kondisi ga ketemu sama sekali
    List<String> path = new ArrayList<>();
    path.add(count.toString());
    return path;
  }

   private static List<String> reconstructPath(Map<String, String> cameFrom, String current) {
    List<String> path = new ArrayList<>();
    while (current != null) {
      path.add(current);
      current = cameFrom.get(current);
    }
    Collections.reverse(path);
    return path;
  }
}
```

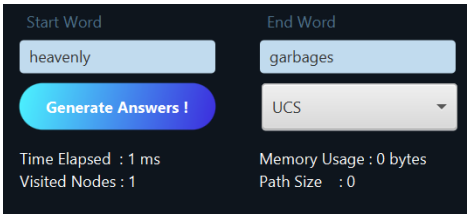
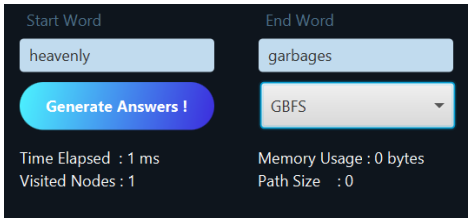
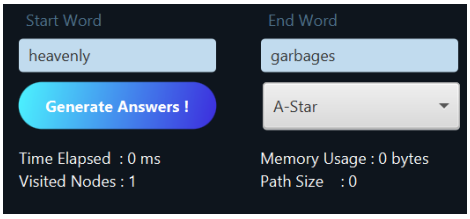
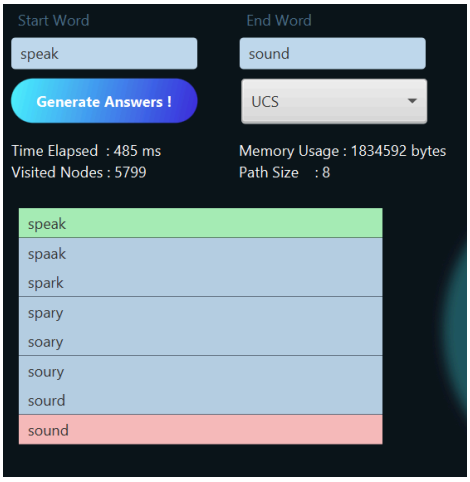
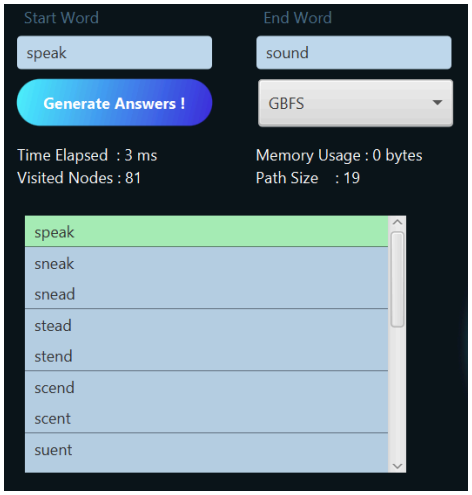
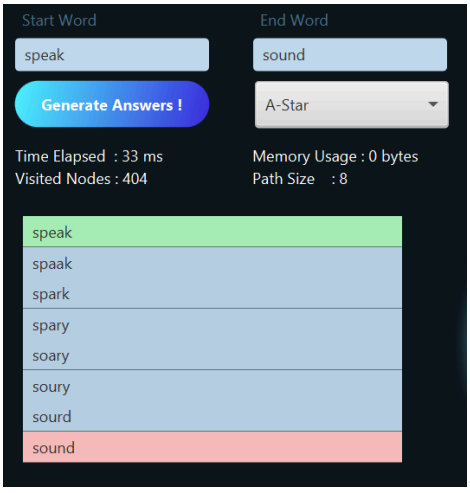
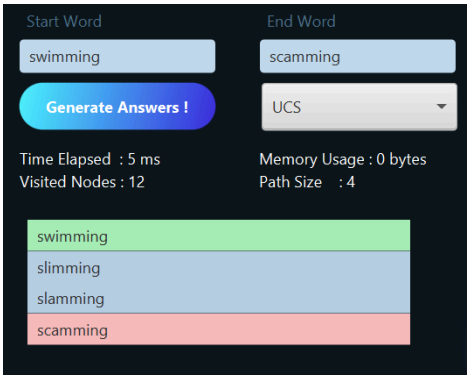
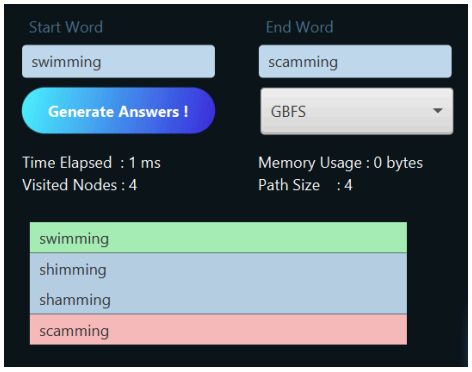
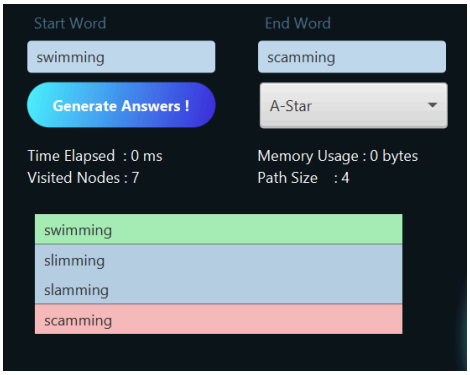
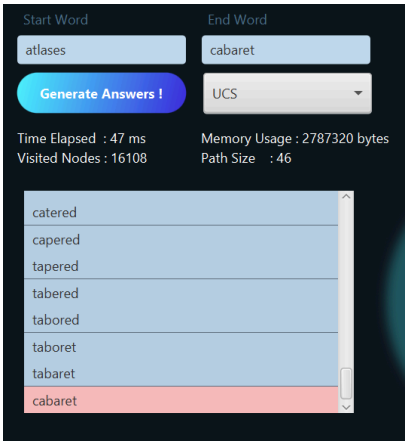
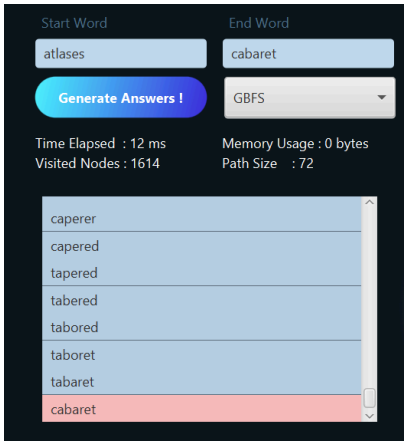
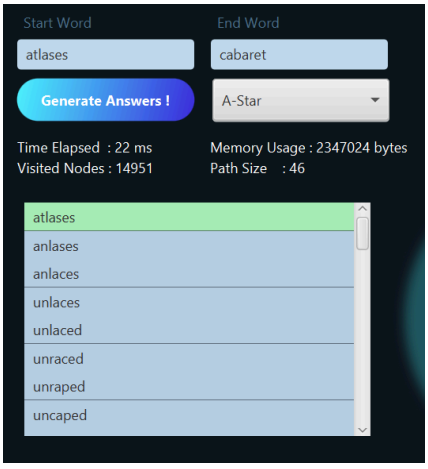
4 **AStar.java**
Pencarian dengan algoritma A*

```
import java.util.*;

class AStar extends WordLadder {
    // Fungsi untuk mencari jumlah karakter yang berbeda antara dua string, akan digunakan sebagai nilai h(n)
  public static Integer getDiffChar(String curr, String target) {
    int count = 0;
    for (int i = 0; i < curr.length(); i++) {
      if (curr.charAt(i) != target.charAt(i)) {
        count++;
      }
    }
    return count;

  }

  public static List<String> astar(String start, String end) {
    // Jarak dari root sampai current, akan diambil yang paling kecil pada tiap node
    Map<String, Integer> costSoFar = new HashMap<>();
    Map<String, Integer> costTotal = new HashMap<>();
    Map<String, String> cameFrom = new HashMap<>();
    // Mengurut dari costTotal yaitu g(n) + h(n)
                                        PriorityQueue<String>  queue  =  new
```

```
PriorityQueue<>(Comparator.comparingInt(costTotal::get));
        queue.add(start);
        costSoFar.put(start, 0);
        costTotal.put(start, getDiffChar(start, end));
        Integer count = 0;

        // Iterasi sampai queue kosong/ketemu
        while (!queue.isEmpty()) {

            count++;
            String current = queue.poll();
            // Kondisi Ketemu
            if (current.equals(end)) {
                List<String> path = reconstructPath(cameFrom, current);
                path.addFirst(count.toString());
                return path;
            }
            // Kondisi lanjut
            for (String next : wordMap.getOrDefault(current, Collections.emptyList())) {
                int newCost = costSoFar.get(current) + 1;
                int newCostTotal = newCost + getDiffChar(next, end);
                if (!costSoFar.containsKey(next) || newCostTotal < costTotal.get(next)) {
                    costSoFar.put(next, newCost);
                    costTotal.put(next, newCostTotal);
                    cameFrom.put(next, current);
                    queue.add(next);
                }
            }
        }
        // Kondisi ga ketemu sama sekali
        List<String> path = new ArrayList<>();
        path.add(count.toString());
        return path;
    }
    // Membentuk path dari start sampai end
     private static List<String> reconstructPath(Map<String, String> cameFrom, String
current) {
        List<String> path = new ArrayList<>();
        while (current != null) {
            path.add(current);
            current = cameFrom.get(current);
        }
        Collections.reverse(path);
        return path;
    }

}
```

V. Test Case

No.	UCS	GBFS	A-Star
1.			
2.			
3.			

4.	 Start Word heavenly End Word garbages Generate Answers ! UCS Time Elapsed : 1 ms Visited Nodes : 1 Memory Usage : 0 bytes Path Size : 0	 Start Word heavenly End Word garbages Generate Answers ! GBFS Time Elapsed : 1 ms Visited Nodes : 1 Memory Usage : 0 bytes Path Size : 0	 Start Word heavenly End Word garbages Generate Answers ! A-Star Time Elapsed : 0 ms Visited Nodes : 1 Memory Usage : 0 bytes Path Size : 0
5	 Start Word speak End Word sound Generate Answers ! UCS Time Elapsed : 485 ms Visited Nodes : 5799 Memory Usage : 1834592 bytes Path Size : 8 speak spaak spark spary soary soury sourd sound	 Start Word speak End Word sound Generate Answers ! GBFS Time Elapsed : 3 ms Visited Nodes : 81 Memory Usage : 0 bytes Path Size : 19 speak sneak snead stead stend scend scent suent	 Start Word speak End Word sound Generate Answers ! A-Star Time Elapsed : 33 ms Visited Nodes : 404 Memory Usage : 0 bytes Path Size : 8 speak spaak spark spary soary soury sourd sound
6	 Start Word swimming End Word scamming Generate Answers ! UCS Time Elapsed : 5 ms Visited Nodes : 12 Memory Usage : 0 bytes Path Size : 4 swimming slimming slamming scamming	 Start Word swimming End Word scamming Generate Answers ! GBFS Time Elapsed : 1 ms Visited Nodes : 4 Memory Usage : 0 bytes Path Size : 4 swimming shimming shamming scamming	 Start Word swimming End Word scamming Generate Answers ! A-Star Time Elapsed : 0 ms Visited Nodes : 7 Memory Usage : 0 bytes Path Size : 4 swimming slimming slamming scamming
7	 Start Word atlases End Word cabaret Generate Answers ! UCS Time Elapsed : 47 ms Visited Nodes : 16108 Memory Usage : 2787320 bytes Path Size : 46 catered capered tapered tabered tabored taboret tabaret cabaret	 Start Word atlases End Word cabaret Generate Answers ! GBFS Time Elapsed : 12 ms Visited Nodes : 1614 Memory Usage : 0 bytes Path Size : 72 caperer capered tapered tabered tabored taboret tabaret cabaret	 Start Word atlases End Word cabaret Generate Answers ! A-Star Time Elapsed : 22 ms Visited Nodes : 14951 Memory Usage : 2347024 bytes Path Size : 46 atlases anlases anlaces unlaces unlaced unraced unraped uncaped

VI. Analisis

Dari perspektif penemuan solusi yang optimal, algoritma UCS dan A* selalu menemukan solusi terbaik, sementara algoritma GBFS sering kali menghasilkan solusi yang suboptimal. Hal ini terbukti dari analisis panjang jalur yang dihasilkan dalam pengujian, di mana pada kasus terburuk, algoritma GBFS memiliki panjang jalur yang bahkan melebihi dua kali lipat dari solusi optimal (Test Case 5).

Secara teoritis, kompleksitas waktu untuk algoritma UCS, GBFS, dan A* search adalah O(b^m), di mana b adalah faktor percabangan (jumlah kata dengan panjang yang sama dalam kamus) dan m adalah kedalaman pohon (jarak dari kata awal ke kata akhir). Namun, kompleksitas waktu tidak memberikan informasi definitif tentang algoritma mana yang seharusnya lebih cepat jika kompleksitas yang dibandingkan sama. Ini karena kompleksitas waktu hanya menunjukkan seberapa cepat pertumbuhannya, bukan seberapa banyak pertumbuhannya. Namun dari hasil uji coba, didapatkan algoritma GBFS selalu menjadi yang paling unggul dalam konteks lama proses pencarian, serta algoritma A* selalu mengalahkan UCS, sehingga dapat disimpulkan bahwa tingkat kecepatan pencarian secara umum dapat diperoleh sebagai berikut: GBFS > A* > UCS.

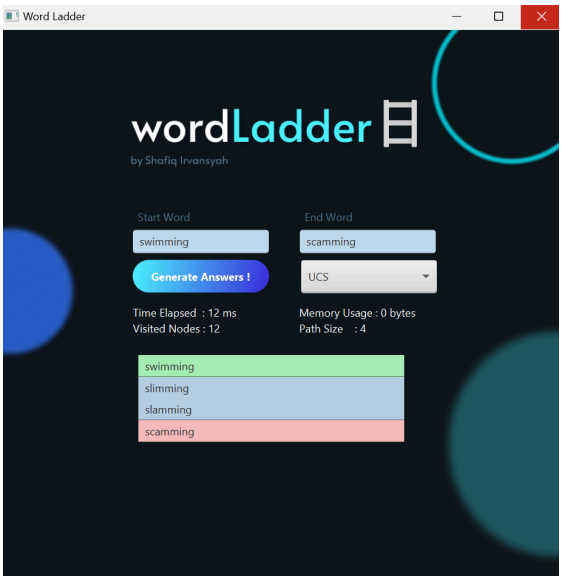
Dari perspektif penggunaan memori, pada seluruh uji coba terdapat pola konsisten, di mana penggunaan memori untuk algoritma UCS lebih tinggi daripada A* dan GBFS. Hal ini juga terbukti dengan jumlah node yang dikunjungi oleh UCS lebih banyak daripada A* dan GBFS. Dengan demikian, secara keseluruhan, urutan penggunaan memori untuk algoritma tersebut dapat disimpulkan sebagai berikut: UCS > A* > GBFS.

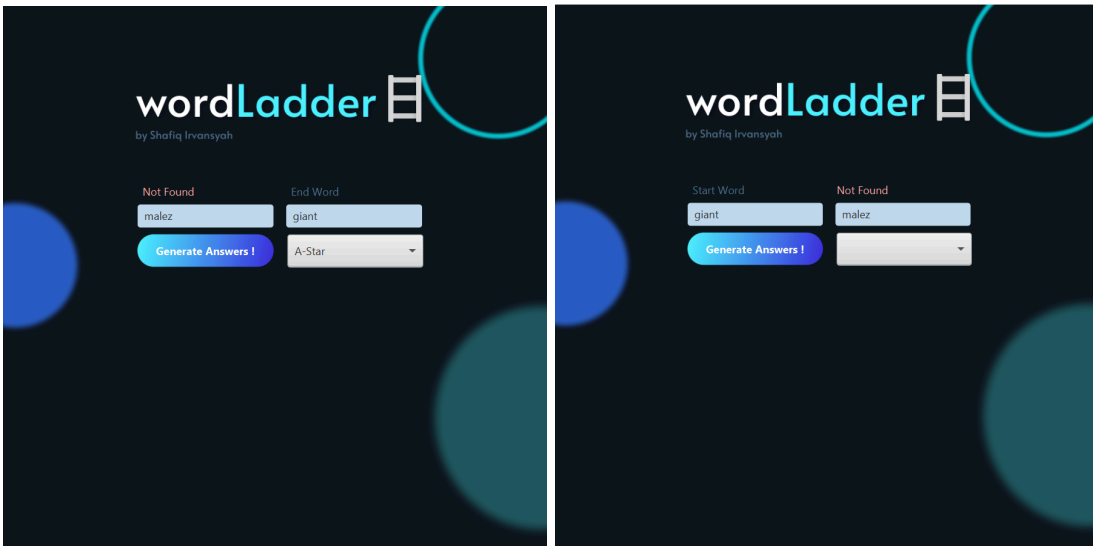
VII. Kesimpulan

Kesimpulannya, algoritma UCS dan A* secara konsisten mampu menemukan solusi optimal, sementara algoritma GBFS seringkali menghasilkan solusi suboptimal dengan panjang jalur yang melebihi dua kali lipat dari solusi optimal pada kasus terburuk. Meskipun kompleksitas waktu secara teoritis sama, uji coba menunjukkan bahwa GBFS memiliki keunggulan dalam waktu proses pencarian, diikuti oleh A* yang mengungguli UCS. Namun, dari segi penggunaan memori, UCS menunjukkan tingkat penggunaan memori yang lebih tinggi daripada A* dan GBFS, dengan jumlah node yang dikunjungi oleh UCS juga lebih banyak. Oleh karena itu, secara keseluruhan, urutan penggunaan memori untuk algoritma tersebut adalah UCS > A* > GBFS. Dengan demikian, algoritma UCS cocok digunakan ketika optimalisasi menjadi prioritas utama, GBFS cocok jika efisiensi waktu menjadi fokus utama, sementara A* menjadi pilihan yang ideal untuk mencapai solusi optimal dengan kecepatan yang relatif tinggi.

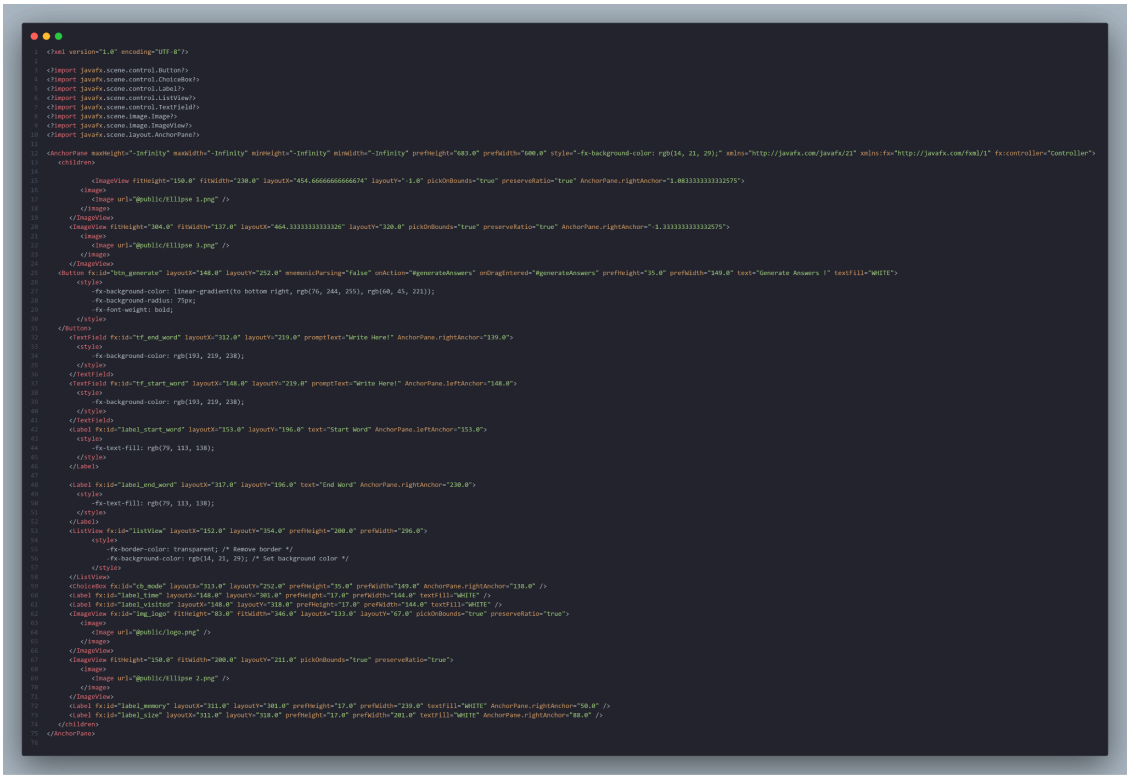
VIII. Implementasi Bonus

GUI




wordLadder
by Shafiq Irvansyah
Not Found
End Word
malez
giant
Generate Answers !
A-Star
wordLadder
by Shafiq Irvansyah
Start Word
Not Found
giant
malez
Generate Answers !

Source Code



IX. Lampiran

Repositori GitHub : https://github.com/shafiqIrv/Tucil3_13522003

X. Daftar Pustaka

Munir, Rinaldi. 2024. “Penentuan rute (Route/Path Planning) - Bagian 1”. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf
(Diakses pada 02 Mei 2024)

Munir, Rinaldi. 2024. “Penentuan rute (Route/Path Planning) - Bagian 2”. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf
(Diakses pada 02 Mei 2024)

No	Poin	Ya	Tidak
1	Program berhasil dijalankan.	☑	☐
2	Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma UCS	☑	☐
3	Solusi yang diberikan pada algoritma UCS optimal	☑	☐
4	Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma Greedy Best First Search	☑	☐
5	Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma A*	☑	☐
6	Solusi yang diberikan pada algoritma A* optimal	☑	☐
7	**[Bonus]:** Program memiliki tampilan GUI	☑	☐