

LAPORAN TUGAS KECIL 03

IF2211 STRATEGI ALGORITMA

PENYELESAIAN PERMAINAN WORD LADDER MENGGUNAKAN

ALGORITMA UCS, GREEDY BEST FIRST SEARCH, DAN A*



Disusun oleh:

Wilson Yusda (13522019)

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2024

DAFTAR ISI

DAFTAR ISI.....	1
DAFTAR GAMBAR.....	4
DAFTAR TABEL.....	6
BAB I	
DESKRIPSI MASALAH.....	7
BAB II	
LANDASAN TEORI.....	8
Algoritma UCS.....	8
Algoritma GBFS.....	9
Algoritma A*.....	9
BAB III	
IMPLEMENTASI PROGRAM.....	11
Rancangan Keseluruhan Program.....	11
Inisialisasi dan Cara Penggunaan Program.....	14
Penerapan Algoritma UCS.....	15
Penerapan Algoritma GBFS.....	17
Penerapan Algoritma A*.....	19
Perancangan Tampilan Sistem.....	22
BAB IV	
ANALISIS DAN PENGUJIAN.....	30
Pengujian Algoritma UCS.....	30
Test Case 1 UCS.....	30
Test Case 2 UCS.....	32
Test Case 3 UCS.....	34
Test Case 4 UCS.....	36
Test Case 5 UCS.....	37
Test Case 6 UCS.....	40
Pengujian Algoritma GBFS.....	42
Test Case 1 GBFS.....	42
Test Case 2 GBFS.....	44
Test Case 3 GBFS.....	46
Test Case 4 GBFS.....	48
Test Case 5 GBFS.....	50
Test Case 6 GBFS.....	52
Pengujian Algoritma A*.....	54
Test Case 1 A*.....	54
Test Case 2 A*.....	56

Test Case 3 A*	58
Test Case 4 A*	60
Test Case 5 A*	62
Test Case 6 A*	64
Pengelompokan Data	65
Analisis Data	66
BAB V	
KESIMPULAN DAN SARAN	70
Kesimpulan	70
Saran	71
BAB VI	
DAFTAR PUSTAKA	72
BAB VII	
LAMPIRAN	73
Github Repository	73
Kelayakan Program	73

DAFTAR GAMBAR

Gambar 1.1 Gambar Word Ladder.....	7
Gambar 3.1.1 Source Code WordLadder.....	12
Gambar 3.1.2 Source Code DictionaryLoader.....	13
Gambar 3.3.1 Source Code WordLadderUCS.....	16
Gambar 3.4.1 Source Code WordLadderGreedy.....	18
Gambar 3.5.1 Source Code WordLadderAStar.....	21
Gambar 3.6.1 Source Code main-view.fxml I.....	23
Gambar 3.6.2 Source Code main-view.fxml II.....	24
Gambar 3.6.3 Source Code Main.....	25
Gambar 3.6.4 Source Code MainController I.....	26
Gambar 3.6.4 Source Code MainController II.....	28
Gambar 3.6.5 Souce Code style.css.....	29
Gambar 4.1.1.1 Test Case 1 UCS Tampilan 1.....	31
Gambar 4.1.1.2 Test Case 1 UCS Tampilan 2.....	31
Gambar 4.1.2.1 Test Case 2 UCS Tampilan 1.....	32
Gambar 4.1.2.2 Test Case 2 UCS Tampilan 2.....	33
Gambar 4.1.3.1 Test Case 3 UCS Tampilan 1.....	34
Gambar 4.1.3.2 Test Case 3 UCS Tampilan 2.....	35
Gambar 4.1.4.1 Test Case 4 UCS Tampilan 1.....	36
Gambar 4.1.4.2 Test Case 4 UCS Tampilan 2.....	37
Gambar 4.1.5.1 Test Case 5 UCS Tampilan 1.....	38
Gambar 4.1.5.2 Test Case 5 UCS Tampilan 2.....	39
Gambar 4.1.6.1 Test Case 1 UCS Tampilan 1.....	40
Gambar 4.1.6.2 Test Case 1 UCS Tampilan 2.....	41
Gambar 4.2.1.1 Test Case 1 GBFS Tampilan 1.....	42
Gambar 4.2.1.2 Test Case 1 GBFS Tampilan 2.....	43
Gambar 4.2.2.1 Test Case 2 GBFS Tampilan 1.....	44
Gambar 4.2.2.2 Test Case 2 GBFS Tampilan 2.....	45
Gambar 4.2.3.1 Test Case 3 GBFS Tampilan 1.....	46
Gambar 4.2.3.2 Test Case 3 GBFS Tampilan 2.....	47
Gambar 4.2.4.1 Test Case 4 GBFS Tampilan 1.....	48
Gambar 4.2.4.2 Test Case 4 GBFS Tampilan 2.....	49
Gambar 4.2.5.1 Test Case 5 GBFS Tampilan 1.....	50
Gambar 4.2.5.2 Test Case 5 GBFS Tampilan 2.....	51
Gambar 4.2.6.1 Test Case 6 GBFS Tampilan 1.....	52
Gambar 4.2.6.2 Test Case 6 GBFS Tampilan 2.....	53

Gambar 4.3.1.1 Test Case 1 A* Tampilan 1.....	54
Gambar 4.3.1.2 Test Case 1 A* Tampilan 2.....	55
Gambar 4.3.2.1 Test Case 2 A* Tampilan 1.....	56
Gambar 4.3.2.2 Test Case 2 A* Tampilan 2.....	57
Gambar 4.3.3.1 Test Case 4 A* Tampilan 1.....	58
Gambar 4.3.3.2 Test Case 4 A* Tampilan 2.....	59
Gambar 4.3.4.1 Test Case 4 A* Tampilan 1.....	60
Gambar 4.3.5.2 Test Case 4 A* Tampilan 2.....	61
Gambar 4.3.5.1 Test Case 5 A* Tampilan 1.....	62
Gambar 4.3.5.2 Test Case 5 A* Tampilan 2.....	63
Gambar 4.3.6.1 Test Case 6 A* Tampilan 1.....	64
Gambar 4.3.6.2 Test Case 6 A* Tampilan 2.....	65

DAFTAR TABEL

Tabel 4.1 Tabel Pengelompokan Data.....	65
Tabel 7.1 Tabel Kelayakan Program.....	73

BAB I

DESKRIPSI MASALAH

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

How To Play

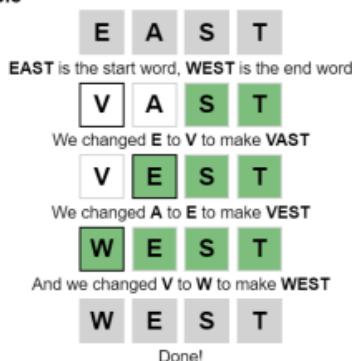
This game is called a "word ladder" and was invented by Lewis Carroll in 1877.

Rules

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

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

Example



Gambar 1.1 Gambar Word Ladder

BAB II

LANDASAN TEORI

Algoritma UCS

Uniform Cost Search (UCS) adalah algoritma pencarian yang mengutamakan jalur dengan biaya terendah tanpa mempertimbangkan estimasi jarak ke tujuan. Dalam konteks UCS, fungsi evaluasi $f(n)$ sama dengan $g(n)$ yang merupakan biaya dari titik awal ke node n . Ini berarti UCS memperluas node berikutnya dengan biaya kumulatif terendah yang terakumulasi dari titik start. Algoritma ini menggunakan struktur data priority queue untuk menyimpan semua node yang perlu dieksplorasi, di mana node dengan biaya terendah selalu dieksplorasi terlebih dahulu. UCS efektif dalam menemukan solusi dengan biaya minimum dan merupakan pendekatan yang ideal dalam situasi di mana semua langkah memiliki biaya yang bervariasi tetapi tujuan utama adalah meminimalkan biaya total perjalanan.

Dalam implementasinya, UCS akan menginisialisasi antrian prioritas dengan simpul awal dan biaya nol. Saat algoritma berlangsung, setiap simpul yang dikeluarkan dari antrian akan diperiksa apakah itu merupakan simpul tujuan. Jika tidak, algoritma akan melanjutkan dengan mengeksplorasi tetangganya, yang masing-masing ditambahkan ke dalam antrian prioritas dengan biaya kumulatif dari simpul awal hingga tetangga tersebut. Hal krusial dalam UCS adalah pemeliharaan daftar simpul yang telah dikunjungi beserta biaya minimum untuk mencapainya. Jika sebuah simpul dijumpai lagi dengan biaya yang lebih rendah, biaya ini akan diperbarui dalam daftar, dan simpul tersebut akan ditambahkan kembali ke dalam antrian untuk dieksplorasi lagi. Proses ini berlangsung terus-menerus sampai tujuan ditemukan atau semua kemungkinan jalur telah dieksplorasi. Ini membuat UCS sangat efektif dalam menemukan solusi yang optimal, namun penggunaannya dapat memakan banyak sumber daya dan waktu terutama dalam graf yang besar dan kompleks. UCS mirip dengan Breadth-First Search (BFS) dengan beberapa modifikasi kunci dalam implementasi pada program ini. BFS adalah strategi unweighted yang memperluas node dalam urutan lebar terlebih dahulu, menjelajahi semua node pada kedalaman d sebelum melanjutkan ke node pada kedalaman $d+1$.

Algoritma GBFS

Greedy Best-First Search (GBFS) adalah algoritma pencarian yang menggunakan pendekatan serakah untuk mencapai tujuan dengan cara yang cepat dengan memprioritaskan jalur yang tampaknya mendekati tujuan pada setiap langkahnya. Berbeda dengan Uniform Cost Search yang memfokuskan pada biaya terendah, GBFS mengarahkan pencarinya berdasarkan heuristik atau estimasi jarak dari simpul saat ini ke tujuan, tanpa mempertimbangkan total biaya dari titik awal. Dalam GBFS, fungsi evaluasi $f(n)$ hanya tergantung pada fungsi heuristik $h(n)$ yang mengestimasikan jarak terkecil dari node n ke tujuan. Dengan kata lain, $f(n) = h(n)$. Ini membuat GBFS sangat cepat dalam menemukan tujuan karena secara eksklusif mengikuti jalur yang tampak paling menjanjikan berdasarkan heuristik yang diberikan. Namun, kelemahan utama dari pendekatan ini adalah ia tidak menjamin penemuan jalur dengan biaya minimum atau bahkan jalur yang optimal karena bisa dengan mudah terjebak oleh jalan buntu atau terkecoh oleh heuristik yang menyesatkan, mengabaikan biaya total perjalanan.

Heuristik dalam GBFS harus dirancang dengan cermat untuk memastikan bahwa ia informatif dan dapat diandalkan. Misalnya, dalam masalah pencarian rute, heuristik yang sering digunakan adalah jarak garis lurus (straight-line distance) dari simpul saat ini ke tujuan. Ini dikenal sebagai heuristik "haversine" dalam konteks geospasial. Heuristik ini tidak selalu menjamin penemuan jalur terpendek, namun efisiensi waktu yang ditawarkannya membuat algoritma ini sangat berguna untuk pencarian solusi dalam situasi di mana kecepatan lebih penting daripada ketepatan optimal, atau di mana lingkungan pencarian tidak terlalu kompleks. Dalam implementasi program ini, digunakan $h(n)$ berupa jumlah huruf yang berbeda dari target.

Algoritma A*

Algoritma A* (A-star) adalah teknik pencarian lintas yang efektif dan efisien yang menggabungkan aspek-aspek terbaik dari algoritma Uniform Cost Search (UCS) dan Greedy Best-First Search (GBFS). Algoritma ini dirancang untuk menemukan jalur terpendek dari titik awal ke titik tujuan dalam sebuah graf dengan cara yang optimal dan efisien. A* menggunakan fungsi biaya $f(n) = g(n) + h(n)$ untuk setiap simpul n di dalam graf, di mana $g(n)$ adalah biaya sebenarnya dari simpul awal ke simpul n , dan $h(n)$ adalah heuristik yang memperkirakan biaya

terendah dari n ke tujuan. Heuristik ini harus admissible, artinya tidak pernah memperkirakan biaya yang lebih tinggi dari biaya sebenarnya, dan idealnya harus konsisten (monotonik).

Dalam A*, heuristik memainkan peran kunci dalam efisiensi algoritma. Heuristik yang sering digunakan adalah jarak Euclidean, Manhattan dalam ruang geometris, yang memberikan estimasi langsung jarak "segaris" dari simpul saat ini ke tujuan, asalkan simpul-simpul tersebut diatur dalam grid atau ruang serupa, ataupun jumlah huruf yang berbeda dalam penyelesaian *word ladder*. Heuristik ini membantu A* secara efektif meminimalkan jumlah simpul yang dieksplorasi dalam proses mencari jalur optimal.

Algoritma A* memulai pencarinya dengan simpul awal, memperluas simpul dengan biaya $f(n)$ terendah terlebih dahulu. Dengan demikian, ia mencampur pendekatan serakah dari GBFS, yang mencari untuk meminimalkan heuristik $h(n)$, dengan pendekatan yang lebih hati-hati dan metrik dari UCS, yang mengutamakan jalur dengan biaya kumulatif terendah $g(n)$. Selama pencarian, A* terus memperbarui jalur ke setiap simpul jika menemukan cara yang lebih murah untuk mencapainya, memastikan bahwa setiap simpul hanya dieksplorasi pada waktu yang paling optimal. Ketika tujuan ditemukan, A* dapat dengan cepat merekonstruksi jalur terpendek dengan mengikuti tautan orang tua dari simpul tujuan kembali ke simpul awal. Untuk perbandingan secara menyeluruh, dapat dilihat pada analisis data pada bagian 4 dibawah.

BAB III

IMPLEMENTASI PROGRAM

Rancangan Keseluruhan Program

Program dibuat dengan menggunakan bahasa pemrograman java dengan spesifikasi JDK 17 , terutama untuk WSL, namun dapat menggunakan versi 21 untuk windows. Versi yang dibutuhkan terbilang spesifik untuk versi ini saja karena penggerjaan didukung dengan tampilan GUI Java FX yang dibangun dengan Maven, dimana dalam pembangunannya dispesifikasikan untuk menggunakan JDK dan Maven yang sudah di atur untuk versi 17 pada pom.xml . Program didasari pada sebuah kelas utama yaitu WordLadder dan menerapkan konsep OOP untuk memberikan turunan kepada berbagai jenis penerapan algoritma.

```
● ● ●
1 package main.tucil_13522019;
2 import java.io.IOException;
3 import java.util.Set;
4 import java.util.*;
5 public class WordLadder {
6     protected Set<String> dictionary;
7     protected String start;
8     protected String end;
9     public WordLadder(Set<String> dictFile) throws IOException {
10         this.dictionary = dictFile;
11     }
12     public List<String> getNeighbors(String word) {
13         List<String> neighbors = new ArrayList<>();
14         char[] chars = word.toCharArray();
15         for (int i = 0; i < chars.length; i++) {
16             char originalChar = chars[i];
17             for (char c = 'a'; c <= 'z'; c++) {
18                 if (c != originalChar) {
19                     chars[i] = c;
20                     String newWord = new String(chars);
21                     if (dictionary.contains(newWord)) {
22                         neighbors.add(newWord);
23                     }
24                 }
25             }
26             chars[i] = originalChar;
27         }
28         // System.out.println(neighbors);
29         return neighbors;
30     }
31 }
32
```

Gambar 3.1.1 Source Code WordLadder

WordLadder sendiri memiliki fungsi yang sudah dideklarasikan yaitu `getneighbours` dimana fungsi ini menerima sebuah string dan akan mengembalikan kumpulan string yang dapat dijadikan acuan sebagai *branch* yang dihasilkan dari sebuah kata dalam berbagai iterasi posisi

huruf susunan kata tersebut. Variasi dari hasil pencarian *node* dapat dimulai dengan melakukan modifikasi pada fungsi *getneighbours* dimana untuk tipe yang berbeda, seperti List atau Set akan memberikan hasil yang berbeda mengingat cara penyimpanan yang juga berbeda. Dalam program ini akan digunakan List karena setelah pengetesan beberapa *test case* terlihat bahwa List lebih optimal dalam implementasinya. Untuk tiap turunan, akan memiliki fungsi *findLadder* dan kelas *Node* tersendiri mengingat konsep pencarian serta *f(n)*, *g(n)*, dan *h(n)* untuk tiap algoritma bervariasi.



```
1 package main.tucil_13522019;
2 import java.io.BufferedReader;
3 import java.io.FileReader;
4 import java.io.IOException;
5 import java.util.HashSet;
6 import java.util.Set;
7 public class DictionaryLoader {
8     private Set<String> dictionary;
9
10    public static Set<String> loadDictionary(String dictFile) throws IOException {
11        Set<String> dict = new HashSet<>();
12        try (BufferedReader reader = new BufferedReader(new FileReader(dictFile))) {
13            String line;
14            while ((line = reader.readLine()) != null) {
15                dict.add(line.trim().toLowerCase());
16            }
17        }
18        return dict;
19    }
20 }
```

Gambar 3.1.2 Source Code DictionaryLoader

Bagian lainnya yaitu *DictionaryLoader* yang dibuat terpisah agar sewaktu inisialisasi hanya perlu dipanggil sekali dan memperingan kinerja program. Terakhir terdapat kelas *Main* dan *MainController* yang merupakan bagian dari GUI. *Main* hanya berfungsi sebagai wadah pembentukan aplikasi, sedangkan untuk interaksi (UX) dari program dibuat pada *MainController*. Dalam folder *resources* juga terdapat *main-view.fxml*. File ini bekerja layaknya HTML dan dibuat dengan menggunakan Scene Builder untuk memberikan UI yang menarik.

Inisialisasi dan Cara Penggunaan Program

Program dimulai dengan menginisialisasi GUI serta objek objek seperti Dictionary dan tiap kelas WordLadder. Kemudian aplikasi akan meminta input kata awal dan kata akhir dan memberikan pula pilihan algoritma serta tombol pencarian. Untuk setiap pilihan, akan dipanggil fungsi findLadder dari masing masing kelas yang sesuai dengan pilihan algoritma.

Adapun spesifikasi yang perlu dimiliki untuk menjalankan program ini yakni:

1. *Code Editor* dengan *ekstension* yang sesuai
2. Java , JDK yang sudah disesuaikan dengan PATH di *environment variables*. (Spesifik JDK 17 jika ingin dilakukan dalam WSL, dapat dilakukan dengan sudo apt update dan sudo apt install openjdk-17-jdk).
3. Maven yang sudah disesuaikan ke dalam PATH *environment variables*.
4. Disarankan untuk menggunakan *command prompt* diluar *Code Editor*. Jika ingin menggunakan *Code Editor*, dapat merestart *Code Editor* ataupun perangkat mengingat *Code Editor* terkadang tidak menyimpan perubahan PATH.

Sebagai penanda, dapat melakukan pengecekan java dan maven secara mandiri terlebih dahulu pada *command prompt* sebelum menjalankan program. Panduan instalasi maven dapat diakses pada video berikut: https://youtu.be/YTvlb6eny_0?si=7SQohZXGTd-K52lo

Langkah penjalanan dan penggunaan program yaitu:

1. Memastikan spesifikasi diatas sudah terpenuhi
2. Mengarahkan *directory* ke folder src (sebelum main)
3. Menjalankan *command* mvn clean javafx:run untuk menampilkan GUI
4. Untuk CLI, dapat menjalankan command ./run.bat untuk menggunakan CLI.
5. Memasukkan kata awal dan kata tujuan dengan memperhatikan format yakni:
 - ◆ Kedua kata tidak boleh kosong
 - ◆ Kedua kata harus memiliki panjang yang sama
 - ◆ Kedua kata harus terdaftar dalam kamus yang digunakan aplikasi
6. Pilih algoritma yang ingin digunakan dan jalankan dengan menekan tombol yang pencarian.

Penerapan Algoritma UCS

Penerapan algoritma UCS dilakukan pada kelas WordLadderUCS. Dalam kelas ini, dideklarasikan findLadder serta Node yang telah dikhususkan untuk algoritma UCS. Deklarasi node pada UCS memiliki tambahan yakni berupa *cost*. Hal ini f(n) pada algoritma UCS hanya bergantung pada g(n), dimana g(n) dalam hal ini berupa jarak dari *root* ke *node*.



```
1 package main.tucil_13522019;
2 import java.io.IOException;
3 import java.util.*;
4 import java.util.AbstractMap.SimpleEntry;
5
6 public class WordLadderUCS extends WordLadder {
7
8     public WordLadderUCS(Set<String> dictFile) throws IOException {
9         super(dictFile);
10    }
11
12    public SimpleEntry<List<String>, Integer> findLadder(String start, String end) {
13        if (!dictionary.contains(start) || !dictionary.contains(end)) {
14            return new SimpleEntry<>(Collections.emptyList(), 0);
15        }
16
17        PriorityQueue<Node> frontier = new PriorityQueue<>(Comparator.comparingInt(n -> n.cost));
18        Map<String, Integer> visited = new HashMap<>();
19        int nodesVisited = 0;
20
21        frontier.add(new Node(start, null, 0));
22
23        while (!frontier.isEmpty()) {
24            Node current = frontier.poll();
25            nodesVisited++;
26
27            if (current.word.equals(end)) {
28                return new SimpleEntry<>(buildPath(current), nodesVisited);
29            }
30            visited.put(current.word, current.cost);
31
32            for (String neighbor : getNeighbors(current.word)) {
33                if (!visited.containsKey(neighbor) || current.cost + 1 < visited.get(neighbor)) {
34                    visited.put(neighbor, current.cost + 1);
35                    frontier.add(new Node(neighbor, current, current.cost + 1));
36                }
37            }
38        }
39
40        return new SimpleEntry<>(Collections.emptyList(), nodesVisited);
41    }
42    private List<String> buildPath(Node endNode) {
43        LinkedList<String> path = new LinkedList<>();
44        Node current = endNode;
45        while (current != null) {
46            path.addFirst(current.word);
47            current = current.parent;
48        }
49        return path;
50    }
51    private static class Node {
52        String word;
53        Node parent;
54        int cost;
55        Node(String word, Node parent, int cost) {
56            this.word = word;
57            this.parent = parent;
58            this.cost = cost;
59        }
60    }
61 }
62 }
```

Gambar 3.3.1 Source Code WordLadderUCS

Terlihat bahwa dalam UCS, *Node* memiliki atribut kata *node* itu sendiri, *Node parent*, dan juga *cost* dimana *cost* menandakan $f(n)$ yang sudah mencakup $f(n) = g(n)$. Fungsi *buildPath* membangun jalur yang telah dikunjungi berdasarkan masukkan *endNode*.

Secara jelas, tahapan pencarian dari *findLadder* dapat diuraikan dalam poin di bawah:

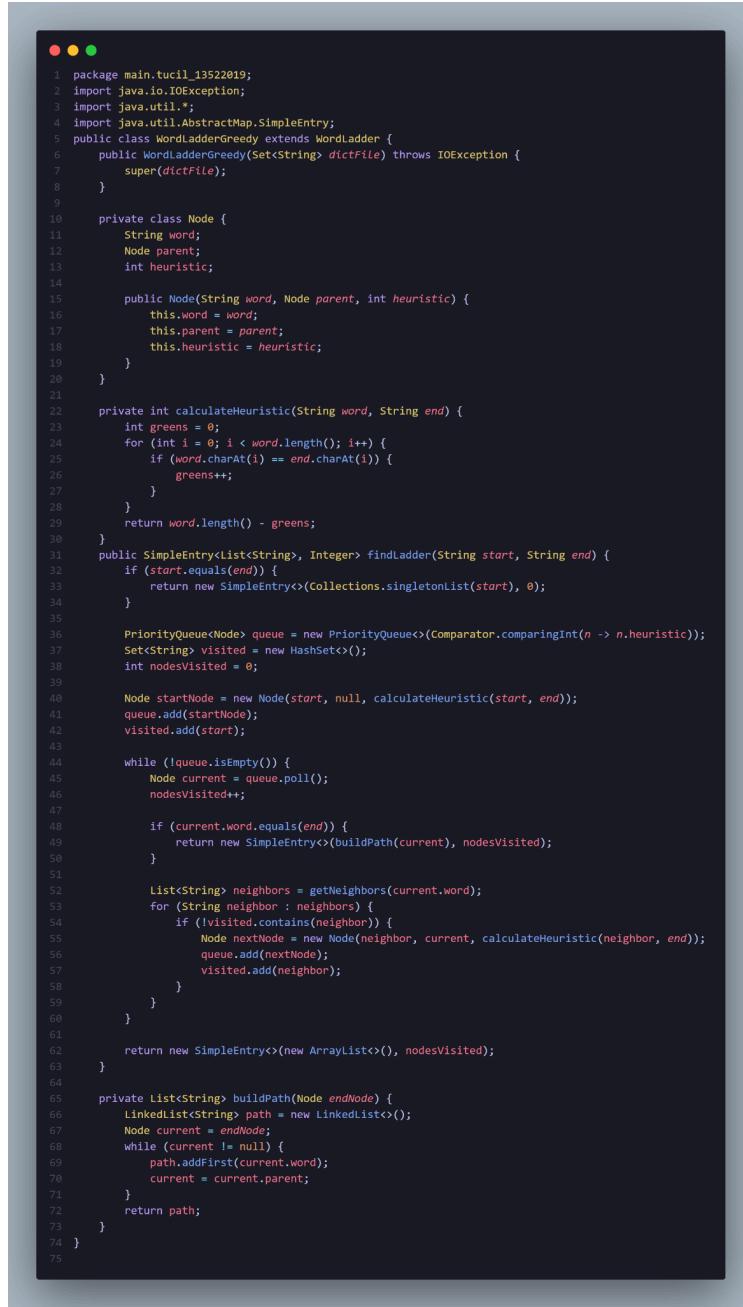
1. Implementasi menggunakan Priority Queue yang berfungsi untuk menyimpan *node* yang akan dieksplorasi. Implementasi komparasi menggunakan perbandingan *cost* pada *node* sehingga memastikan *node* dengan *cost* paling rendah akan dieksplorasi terlebih dahulu.
2. Dibuat pula sebuah Map yang menyimpan node yang dikunjungi serta value berupa biaya terendah yang diperlukan untuk mencapai node tersebut. Hal ini untuk mencegah pemeriksaan yang berulang pada node sehingga dilabel sebagai telah dikunjungi.
3. Menambahkan node awal ke Priority Queue dengan biaya 0 kemudian melakukan iterasi selama Priority Queue tidak kosong dan mengambil node dengan biaya terendah pada setiap iterasi.
4. Jika node yang diambil merupakan node tujuan (kata akhir), jalur dari kata awal ke kata akhir dibangun menggunakan *buildPath* dan dikembalikan sebagai hasil, dimana sudah dipastikan pasti merupakan jalur dengan efisiensi maksimal akibat pemilihan node dengan biaya terendah.
5. Jika bukan, iterasi melalui setiap tetangga dari kata saat ini yang diperoleh dari fungsi *getNeighbors*.
6. Jika tetangga belum dikunjungi atau biaya untuk mencapainya melalui node saat ini lebih rendah dari biaya yang diketahui sebelumnya, perbarui atau set biaya di *visited* dan tambahkan ke Priority Queue.
7. Terakhir, fungsi *buildPath* membangun jalur dari node tujuan kembali ke node awal dengan mengikuti tautan parent dari setiap node, dimulai dari node tujuan hingga mencapai null (node awal).

Melihat dari tahapan pencarian, pencarian UCS memiliki kesamaan dengan algoritma BFS. Kesamaan yang dihasilkan yakni berupa sistem peng-ekspansi menurut tingkatan serta pencarian yang bersifat meluas. Dalam kasus Word Ladder dengan biaya langkah yang seragam, UCS akan

beroperasi serupa dengan BFS karena setiap langkah memiliki bobot yang sama, dan pilihan node untuk ekspansi akan mengikuti urutan yang sama dengan BFS.

Penerapan Algoritma GBFS

Penerapan algoritma Greedy Best First Search diterapkan dalam kelas WordLadderGreedy. Dalam GBFS, $f(n) = h(n)$, dimana $h(n)$ menandakan *heuristic* dari konsep pencarian. Dalam penggerjaan program ini, ditentukan bahwa $h(n)$ yang optimal yaitu berupa jumlah huruf yang berbeda dari target. Pengambilan konsep ini mengingat pada permainan asli, dimana sewaktu user memberikan tebakan, sistem memberikan penanda mengenai ketepatan pemilihan huruf. Baik dalam posisi yang salah (warna kuning) maupun posisi benar (warna hijau) untuk huruf yang terkandung dalam jawaban, dapat memberikan arahan bagi user dalam memilih huruf yang seharusnya. Oleh karena itu, dalam perancangan GBFS, digunakan $h(n)$ berupa jumlah huruf yang berbeda untuk disesuaikan dengan konsep pemilihan *heuristic*.



```
1 package main.tucil_13522019;
2 import java.io.IOException;
3 import java.util.*;
4 import java.util.AbstractMap.SimpleEntry;
5 public class WordLadderGreedy extends WordLadder {
6     public WordLadderGreedy(Set<String> dictFile) throws IOException {
7         super(dictFile);
8     }
9
10    private class Node {
11        String word;
12        Node parent;
13        int heuristic;
14
15        public Node(String word, Node parent, int heuristic) {
16            this.word = word;
17            this.parent = parent;
18            this.heuristic = heuristic;
19        }
20    }
21
22    private int calculateHeuristic(String word, String end) {
23        int greens = 0;
24        for (int i = 0; i < word.length(); i++) {
25            if (word.charAt(i) == end.charAt(i)) {
26                greens++;
27            }
28        }
29        return word.length() - greens;
30    }
31    public SimpleEntry<List<String>, Integer> findLadder(String start, String end) {
32        if (start.equals(end)) {
33            return new SimpleEntry<>(Collections.singletonList(start), 0);
34        }
35
36        PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(n -> n.heuristic));
37        Set<String> visited = new HashSet<>();
38        int nodesVisited = 0;
39
40        Node startNode = new Node(start, null, calculateHeuristic(start, end));
41        queue.add(startNode);
42        visited.add(start);
43
44        while (!queue.isEmpty()) {
45            Node current = queue.poll();
46            nodesVisited++;
47
48            if (current.word.equals(end)) {
49                return new SimpleEntry<>(buildPath(current), nodesVisited);
50            }
51
52            List<String> neighbors = getNeighbors(current.word);
53            for (String neighbor : neighbors) {
54                if (!visited.contains(neighbor)) {
55                    Node nextNode = new Node(neighbor, current, calculateHeuristic(neighbor, end));
56                    queue.add(nextNode);
57                    visited.add(neighbor);
58                }
59            }
60        }
61
62        return new SimpleEntry<>(new ArrayList<>(), nodesVisited);
63    }
64
65    private List<String> buildPath(Node endNode) {
66        LinkedList<String> path = new LinkedList<>();
67        Node current = endNode;
68        while (current != null) {
69            path.addFirst(current.word);
70            current = current.parent;
71        }
72        return path;
73    }
74 }
```

Gambar 3.4.1 Source Code WordLadderGreedy

Deklarasi *Node* dalam konsep GBFS sedikit berbeda dari sisi penamaan karena GBFS mengandalkan $h(n)$ sehingga terdapat atribut *heuristic* dan fungsi *calculateHeuristic* yang menentukan nilai *heuristic* berdasarkan jumlah kata yang berbeda. Kemudian terdapat *buildPath* yang membangun jalur sesuai dengan masukkan *node* akhir. Secara jelas, langkah-langkah penentuan urutan pencarian kata pada *findLadder* dapat diuraikan secara berikut:

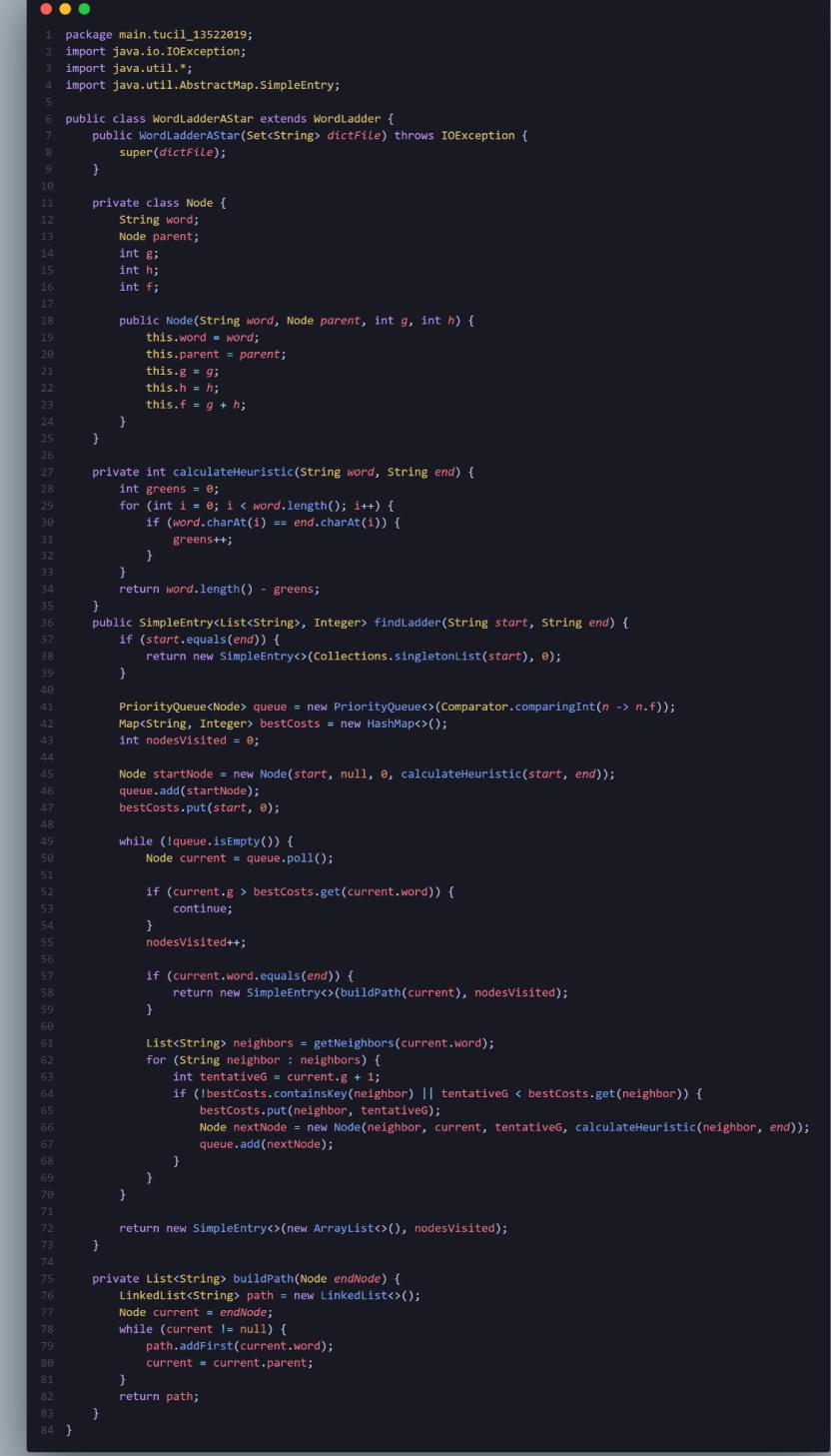
1. Inisialisasi dimulai dengan menggunakan Priority Queue dalam menyimpan node, dimana dalam program diatur untuk dibandingkan sesuai nilai dari *heuristic*.
2. Dibuat sebuah set untuk menyimpan kata yang telah dikunjungi (akibat merupakan penggeraan lanjut dari UCS, diketahui bahwa cukup mengingat node yang telah dikunjungi sehingga bebas jika ingin digunakan set ataupun map) dan dicatat node yang dikunjungi dalam variabel lain.
3. Selama antrian tidak kosong, ambil node dari antrian yang memiliki heuristik terendah.
4. Periksa apakah node tersebut adalah kata tujuan. Jika ya, bangun dan kembalikan jalur dari kata awal ke kata tujuan menggunakan metode buildPath. Jika bukan kata tujuan, dapatkan semua tetangga yang valid dari kata tersebut menggunakan metode getNeighbors. Penyusunan dijamin sesuai karena build path menyesuaikan dengan *node* terakhir dan dalam pemilihan *node* sudah ditentukan sesuai dengan perbandingan $h(n)$.
5. Untuk setiap tetangga, hitung heuristik mereka terhadap kata tujuan. Jika tetangga belum dikunjungi, tambahkan ke antrian prioritas dan tandai sebagai dikunjungi.
6. Jika kata tujuan ditemukan, bangun jalur kembali ke kata awal dengan mengikuti referensi parent dari setiap node.

GBFS menentukan pengambilan node dengan melihat nilai dari *heuristic*. Algoritma terlihat memilih jalur yang secara heuristik terlihat menjanjikan tetapi sebenarnya lebih panjang atau lebih rumit dibandingkan jalur lain yang lebih sederhana atau lebih pendek. Selain itu, secara keseluruhan, GBFS bergantung pada cara penilaian *heuristic* sehingga keoptimalan algoritma mengarah pada pengambilan *heuristic*. Meskipun begitu, GBFS lebih condong memberikan hasil secara lebih cepat, namun bukan berupa jawaban paling optimal akibat konsep pengambilan jalur yang hanya bergantung pada 1 jalur dan hanya mengandalkan *heuristic* yang ditentukan.

Penerapan Algoritma A*

Penerapan algoritma A* ditentukan dalam kelas WordLadderAStar. Sesuai dengan penjelasan mengenai $f(n)$ dari A*, dimana $f(n) = g(n) + h(n)$, A* menggabungkan konsep dari UCS serta *heuristic* dari GBFS. $g(n)$ merupakan nilai dari jarak dari *root* ke *node*, sedangkan $h(n)$ merupakan nilai *heuristic* yang ditandai dengan menghitung jumlah huruf yang berbeda ketika dibandingkan dengan target. Pengambilan *heuristic* ini memiliki alasan yang sama dengan

penentuan *heuristic* pada algoritma GBFS, dimana disesuaikan dengan cara bermain dari permainan *word ladder* ini sendiri



```
1 package main.tucil_13522019;
2 import java.io.IOException;
3 import java.util.*;
4 import java.util.AbstractMap.SimpleEntry;
5
6 public class WordLadderAStar extends WordLadder {
7     public WordLadderAStar(Set<String> dictFile) throws IOException {
8         super(dictFile);
9     }
10
11    private class Node {
12        String word;
13        Node parent;
14        int g;
15        int h;
16        int f;
17
18        public Node(String word, Node parent, int g, int h) {
19            this.word = word;
20            this.parent = parent;
21            this.g = g;
22            this.h = h;
23            this.f = g + h;
24        }
25    }
26
27    private int calculateHeuristic(String word, String end) {
28        int greens = 0;
29        for (int i = 0; i < word.length(); i++) {
30            if (word.charAt(i) == end.charAt(i)) {
31                greens++;
32            }
33        }
34        return word.length() - greens;
35    }
36    public SimpleEntry<List<String>, Integer> findLadder(String start, String end) {
37        if (start.equals(end)) {
38            return new SimpleEntry<>(Collections.singletonList(start), 0);
39        }
40
41        PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(n -> n.f));
42        Map<String, Integer> bestCosts = new HashMap<>();
43        int nodesVisited = 0;
44
45        Node startNode = new Node(start, null, 0, calculateHeuristic(start, end));
46        queue.add(startNode);
47        bestCosts.put(start, 0);
48
49        while (!queue.isEmpty()) {
50            Node current = queue.poll();
51
52            if (current.g > bestCosts.get(current.word)) {
53                continue;
54            }
55            nodesVisited++;
56
57            if (current.word.equals(end)) {
58                return new SimpleEntry<>(buildPath(current), nodesVisited);
59            }
60
61            List<String> neighbors = getNeighbors(current.word);
62            for (String neighbor : neighbors) {
63                int tentative = current.g + 1;
64                if (!bestCosts.containsKey(neighbor) || tentative < bestCosts.get(neighbor)) {
65                    bestCosts.put(neighbor, tentative);
66                    Node nextNode = new Node(neighbor, current, tentative, calculateHeuristic(neighbor, end));
67                    queue.add(nextNode);
68                }
69            }
70        }
71
72        return new SimpleEntry<>(new ArrayList<>(), nodesVisited);
73    }
74
75    private List<String> buildPath(Node endNode) {
76        LinkedList<String> path = new LinkedList<>();
77        Node current = endNode;
78        while (current != null) {
79            path.addFirst(current.word);
80            current = current.parent;
81        }
82        return path;
83    }
84 }
```

Gambar 3.5.1 Source Code WordLadderAStar

Terlihat bahwa dibandingkan dengan algoritma lain, *Node* memiliki atribut tambahan yakni *f,g*, dan *h* dimana masing masing menyatakan $f(n)$, $g(n)$, dan $h(n)$. *calculateHeuristic* menghitung $h(n)$ yaitu dengan menghitung jumlah huruf yang berbeda dari *end word*. *buildPath* berfungsi untuk membangun jalur yang sudah dihubungkan dengan menerima masukkan *node* akhir. Adapun cara kerja dari pencarian *findLadder* dengan algoritma A* dapat diuraikan dalam poin dibawah:

1. Inisialisasi Priority Queue yang diurutkan berdasarkan nilai *f* dan dicatat biaya terbaik untuk mencapai node dalam Map dengan nama *bestCosts*.
2. Proses berlanjut selama Priority Queue tidak kosong, dimana iterasi mencakup pengambilan *node* dengan nilai terendah dari Priority Queue.
3. Cek apakah biaya *g* dari node yang diambil lebih besar dari biaya terbaik yang tercatat di *bestCosts*. Jika ya, lewati iterasi ini karena jalur yang lebih baik sudah ditemukan.
4. Jika kata node saat ini sama dengan kata akhir (*end*), konstruksi jalur dari node ini kembali ke node awal menggunakan *buildPath* dan keluar dari loop. Penyusunan dijamin sesuai karena build path menyesuaikan dengan *node* terakhir dan dalam pemilihan *node* sudah ditentukan sesuai dengan perbandingan $h(n)$. Jika queue menjadi kosong sebelum kata akhir ditemukan, ini menandakan tidak ada jalur yang mungkin.
5. Selama belum bertemu kata akhir, dilakukan ekspansi tetangga, dengan menggunakan fungsi *getNeighbors* untuk menemukan semua kata yang bisa dijangkau dari kata saat ini dengan mengubah satu huruf pada satu waktu.
6. Untuk setiap tetangga, hitung *g* tentatif, yaitu *g* dari node saat ini ditambah satu (karena setiap langkah berarti satu perubahan huruf).
7. Jika tetangga belum tercatat di *bestCosts* atau *g* tentatif lebih kecil dari nilai yang ada di *bestCosts* untuk tetangga tersebut, maka perbarui *bestCosts* dan buat node baru untuk tetangga dengan parent adalah node saat ini dengan *h* yang dihitung menggunakan *calculateHeuristic*.
8. Hitung *f* dengan menjumlahkan nilai *g* dan *h* dan tambahkan node ke Priority Queue dan ulangi loop sampai bertemu solusi.

Efektivitas algoritma A* bergantung kepada penentuan *heuristic* yang tepat. Salah satu syarat *heuristic* yang baik adalah bersifat *admissible*. Heuristik dianggap admissible jika setiap estimasi yang diberikan tidak lebih dari biaya nyata untuk mencapai tujuan. Dalam konteks Word Ladder, heuristik ini menghitung jumlah perubahan huruf yang pasti harus dilakukan untuk mengubah word menjadi end. Ini tidak pernah meng-overestimate jumlah perubahan yang diperlukan karena setiap perubahan huruf yang tidak cocok memang harus diubah untuk mencapai kata target. Oleh karena itu, heuristik ini admissible. Untuk kecocokan *heuristic*, dapat melihat pada bagian analisis dibawah.

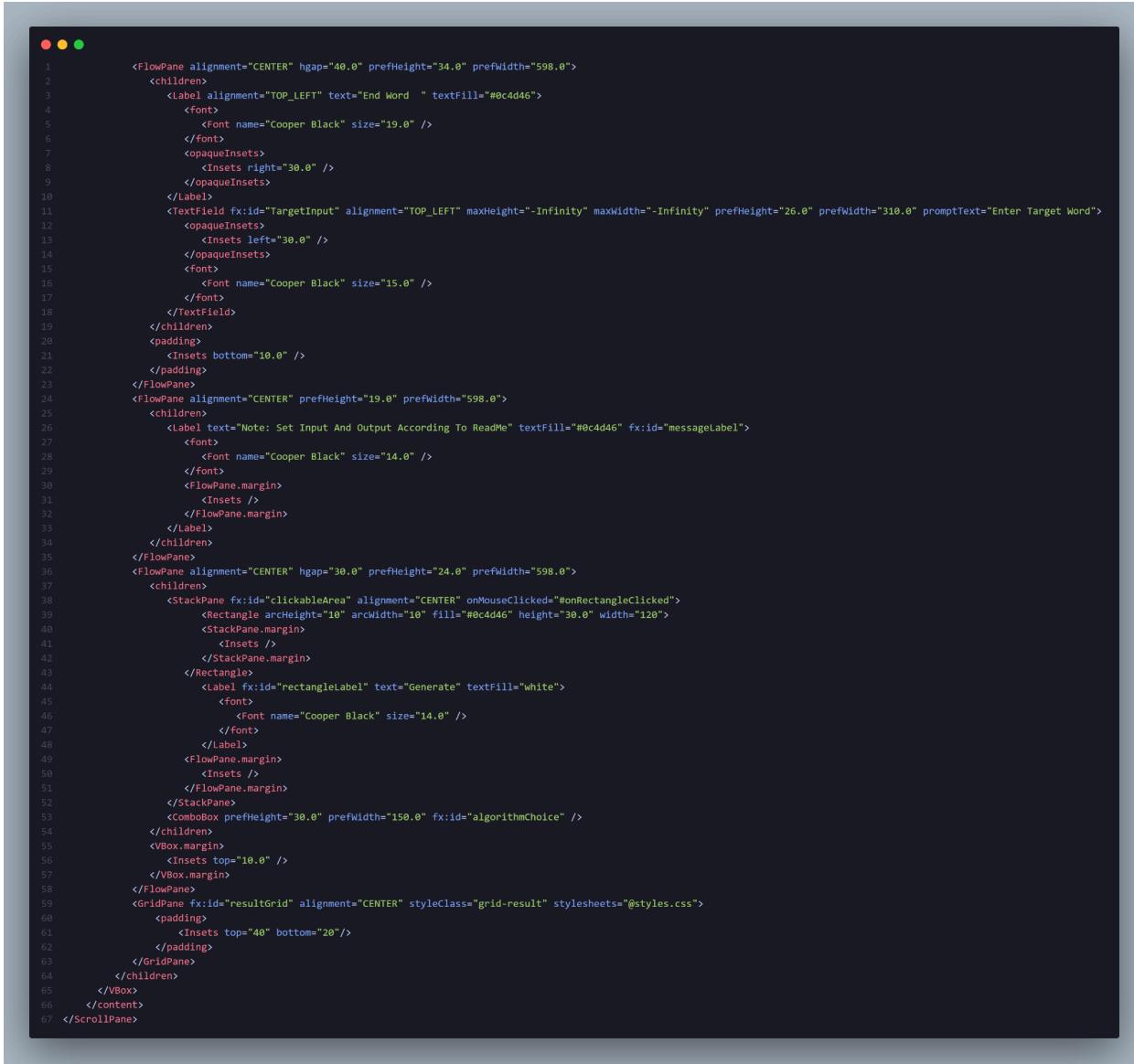
Secara teoritis, algoritma A* cenderung lebih efisien daripada Uniform Cost Search (UCS) dalam kasus Word Ladder, terutama berkat penggunaan heuristik yang admissible dan informatif. Ini memberikan A* kemampuan untuk mencapai tujuan dengan lebih cepat dengan menghindari eksplorasi jalur yang kurang menjanjikan, yang mungkin tetap dieksplorasi oleh UCS. A* menggunakan heuristik untuk memprioritaskan node yang berpotensi lebih dekat ke tujuan, secara teoritis mengurangi jumlah node yang perlu dieksplorasi. UCS tidak menggunakan heuristik dan bekerja dengan prinsip ekspansi node berdasarkan biaya terendah dari titik start hingga titik saat ini. Ini menyebabkan UCS menjelajahi semua kemungkinan jalur tanpa panduan heuristik, yang bisa sangat tidak efisien terutama jika solusi atau tujuan berada jauh dari titik start.

Perancangan Tampilan Sistem

Perancangan dibangun dengan menggunakan *Build Tools* Maven dan bahasa pemrograman Java, lebih tepatnya Java FX. 4 file utama perancangan GUI yakni berupa main-view.fxml, Main.java, MainController.java dan style.css.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <?import javafx.geometry.Insets?>
4 <?import javafx.scene.control.ComboBox?>
5 <?import javafx.scene.control.Label?>
6 <?import javafx.scene.control.ScrollPane?>
7 <?import javafx.scene.control.TextField?>
8 <?import javafx.scene.layout.ColumnConstraints?>
9 <?import javafx.scene.layout.FlowPane?>
10 <?import javafx.scene.layout.GridPane?>
11 <?import javafx.scene.layout.RowConstraints?>
12 <?import javafx.scene.layout.StackPane?>
13 <?import javafx.scene.layout.VBox?>
14 <?import javafx.scene.shape.Rectangle?>
15 <?import javafx.scene.text.Font?>
16
17 <ScrollPane fitToHeight="true" fitToWidth="true" stylesheets="@styles.css" xmlns="http://javafx.com/javafx/21" xmlns:fx="http://javafx.com/fxml/1" fx:controller="main.tucil_13522019.MainController" >
18   <content>
19     <VBox alignment="TOP_CENTER" prefHeight="497.0" prefWidth="619.0" styleClass="main">
20       <padding>
21         <Insets bottom="10" left="10" right="10" top="10" />
22       </padding>
23       <children>
24         <Label alignment="CENTER" contentDisplay="CENTER" text="Word Ladder Game" textFill="#0c4d46">
25           <font>
26             <Font name="Cooper Black" size="25.0" />
27           </font>
28           <opaqueInsets>
29             <Insets />
30           </opaqueInsets>
31           <padding>
32             <Insets bottom="20.0" />
33           </padding>
34         </Label>
35         <FlowPane alignment="CENTER" hgap="40.0" prefHeight="10.0" prefWidth="598.0">
36           <children>
37             <Label alignment="TOP_LEFT" text="Start Word" textFill="#0c4d46">
38               <font>
39                 <Font name="Cooper Black" size="19.0" />
40               </font>
41               <opaqueInsets>
42                 <Insets right="30.0" />
43               </opaqueInsets>
44               <padding>
45                 <Insets left="10.0" />
46               </padding>
47             <Label>
48               <TextField fx:id="StartInput" alignment="TOP_LEFT" maxHeight="-Infinity" maxWidth="-Infinity" prefHeight="26.0" prefWidth="318.0" promptText="Enter Start Word">
49                 <opaqueInsets>
50                   <Insets left="30.0" />
51                 </opaqueInsets>
52                 <FlowPane.margin>
53                   <Insets right="15.0" />
54                 </FlowPane.margin>
55               <font>
56                 <Font name="Cooper Black" size="15.0" />
57               </font>
58             </TextField>
59           </children>
60           <padding>
61             <Insets bottom="20.0" />
62           </padding>
63         </FlowPane>
```

Gambar 3.6.1 Source Code main-view.fxml I

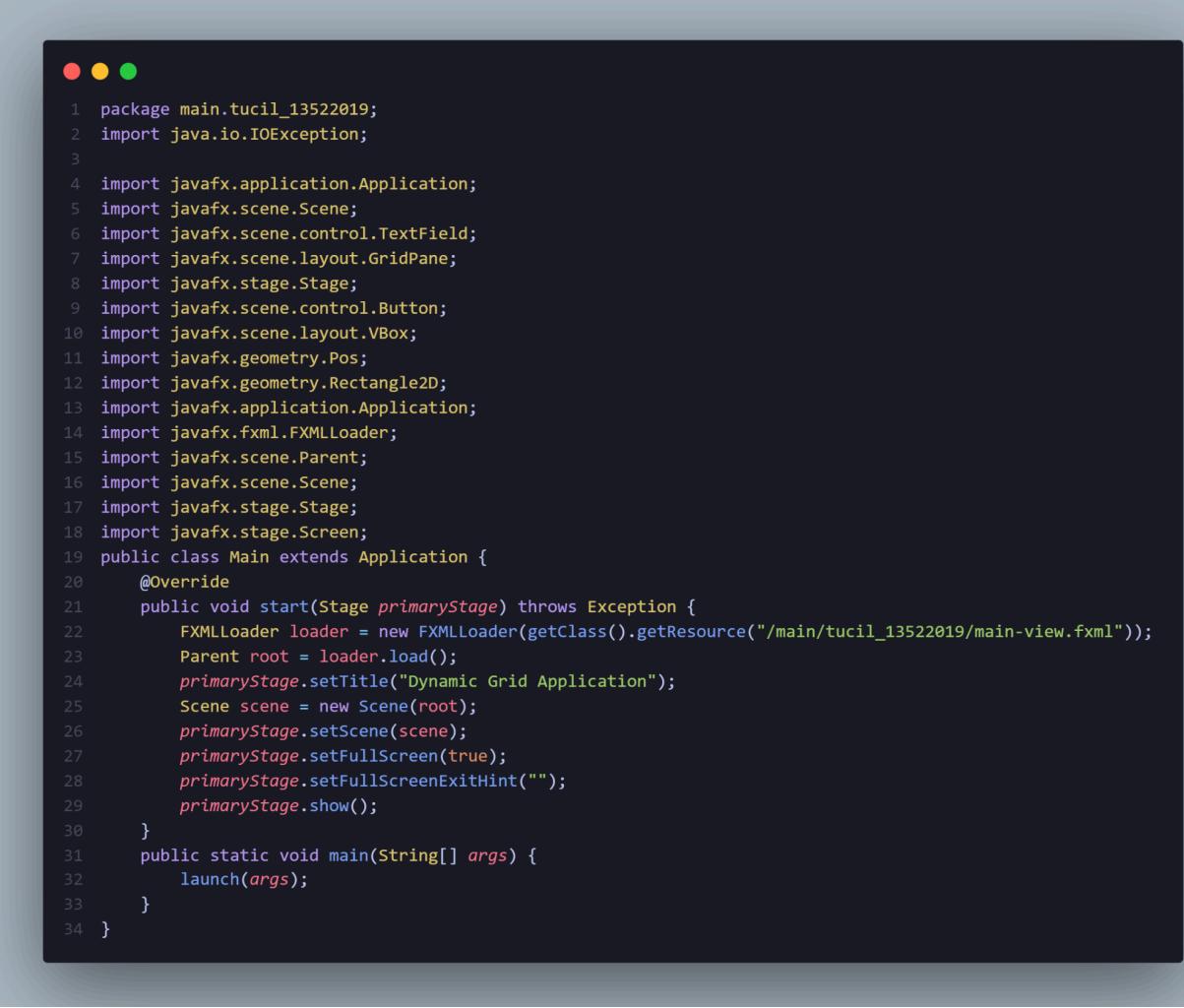


```
1      <FlowPane alignment="CENTER" hgap="40.0" prefHeight="34.0" prefWidth="598.0">
2          <children>
3              <Label alignment="TOP_LEFT" text="End Word" textFill="#0c4d46">
4                  <font>
5                      <font name="Cooper Black" size="19.0" />
6                  </font>
7                  <opaqueInsets>
8                      <Insets right="30.0" />
9                  </opaqueInsets>
10             </Label>
11             <TextField fx:id="TargetInput" alignment="TOP_LEFT" maxHeight="-Infinity" maxWidth="-Infinity" prefHeight="26.0" prefWidth="310.0" promptText="Enter Target Word">
12                 <opaqueInsets>
13                     <Insets left="30.0" />
14                 </opaqueInsets>
15                 <font>
16                     <font name="Cooper Black" size="15.0" />
17                 </font>
18             </TextField>
19         </children>
20         <padding>
21             <Insets bottom="10.0" />
22         </padding>
23     </FlowPane>
24     <FlowPane alignment="CENTER" prefHeight="19.0" prefWidth="598.0">
25         <children>
26             <Label text="Note: Set Input And Output According To ReadMe" textFill="#0c4d46" fx:id="messageLabel">
27                 <font>
28                     <font name="Cooper Black" size="14.0" />
29                 </font>
30                 <FlowPane.margin>
31                     <Insets />
32                 </FlowPane.margin>
33             </Label>
34         </children>
35     </FlowPane>
36     <FlowPane alignment="CENTER" hgap="30.0" prefHeight="24.0" prefWidth="598.0">
37         <children>
38             <StackPane fx:id="clickableArea" alignment="CENTER" onMouseClicked="#onRectangleClicked">
39                 <Rectangle arcHeight="10" arcWidth="10" fill="#0c4d46" height="30.0" width="120">
40                     <StackPane.margin>
41                         <Insets />
42                     </StackPane.margin>
43                 </Rectangle>
44                 <Label fx:id="rectangleLabel" text="Generate" textFill="white">
45                     <font>
46                         <font name="Cooper Black" size="14.0" />
47                     </font>
48                 </Label>
49                 <FlowPane.margin>
50                     <Insets />
51                 </FlowPane.margin>
52             </StackPane>
53             <ComboBox prefHeight="30.0" prefWidth="150.0" fx:id="algorithmChoice" />
54         </children>
55         <VBox.margin>
56             <Insets top="10.0" />
57         </VBox.margin>
58     </FlowPane>
59     <GridPane fx:id="resultGrid" alignment="CENTER" styleClass="grid-result" stylesheets="@styles.css">
60         <padding>
61             <Insets top="40" bottom="20"/>
62         </padding>
63     </GridPane>
64     </children>
65   </VBox>
66 </content>
67 </ScrollPane>
```

Gambar 3.6.2 Source Code main-view.fxml II

Main-view.fxml merancang struktur dari aplikasi. Dalam implementasinya digunakan berbagai komponen seperti *Scroll Pane*, *FlowPane*, *Stack Pane*, *Label*, *ComboBox*, *TextField*, dan *GridPane*. *Scroll Pane* menginisialisasi laman yang bersifat *scrollable* sehingga cocok dijadikan sebagai root atau *container* dari aplikasi. *FlowPane* memiliki konsep seperti *flex* dalam CSS *website* dimana berbagai komponen dapat disusun secara linear horizontal. *StackPane* menerapkan konsep *absolute* CSS pada objek didalamnya namun tidak bagi objek diluar, sehingga dapat menimpa berbagai komponen menjadi satu kesatuan. *StackPane* berguna dalam perancangan tombol yang di-customize sendiri. *Label* merupakan keluaran dalam bentuk tampilan text pada GUI. *ComboBox* merupakan *dropdown* yang memiliki pilihan sesuai kriteria

program. Sesuai dengan namanya, *Text Field* merupakan tempat penerimaan input dan *GridPane* membuat *grid* dengan ukuran yang disesuaikan. Pembuatan keseluruhan struktur dibuat dengan *Scene Builder* untuk memudahkan dan membuat struktur menjadi lebih rapi.

A screenshot of a Java code editor showing the Main.java file. The code is written in Java and uses JavaFX libraries to create a GUI. It includes imports for Application, Scene, TextField, GridPane, Stage, Button, VBox, Pos, Rectangle2D, and FXMLLoader. The class Main extends Application and overrides the start method to load an FXML file named main-view.fxml, set its title to "Dynamic Grid Application", and make it full-screen. The main method launches the application.

```
1 package main.tucil_13522019;
2 import java.io.IOException;
3
4 import javafx.application.Application;
5 import javafx.scene.Scene;
6 import javafx.scene.control.TextField;
7 import javafx.scene.layout.GridPane;
8 import javafx.stage.Stage;
9 import javafx.scene.control.Button;
10 import javafx.scene.layout.VBox;
11 import javafx.geometry.Pos;
12 import javafx.geometry.Rectangle2D;
13 import javafx.application.Application;
14 import javafx.fxml.FXMLLoader;
15 import javafx.scene.Parent;
16 import javafx.scene.Scene;
17 import javafx.stage.Stage;
18 import javafx.stage.Screen;
19 public class Main extends Application {
20     @Override
21     public void start(Stage primaryStage) throws Exception {
22         FXMLLoader loader = new FXMLLoader(getClass().getResource("/main/tucil_13522019/main-view.fxml"));
23         Parent root = loader.load();
24         primaryStage.setTitle("Dynamic Grid Application");
25         Scene scene = new Scene(root);
26         primaryStage.setScene(scene);
27         primaryStage.setFullScreen(true);
28         primaryStage.setFullScreenExitHint("");
29         primaryStage.show();
30     }
31     public static void main(String[] args) {
32         launch(args);
33     }
34 }
```

Gambar 3.6.3 Source Code Main

Main.java hanya memanggil dan membangun GUI dengan fxml yang sudah tersedia dan mengatur berbagai komponen seperti pemberian nama GUI dan penentuan ukuran layar.

```
 1 package main.tucil_13522019;
 2
 3 import java.io.IOException;
 4 import java.util.AbstractMap.SimpleEntry;
 5 import java.util.List;
 6 import java.util.Set;
 7
 8 import javafx.fxml.FXML;
 9 import javafx.geometry.Pos;
10 import javafx.scene.control.TextField;
11 import javafx.scene.layout.ColumnConstraints;
12 import javafx.scene.layout.GridPane;
13 import javafx.scene.layout.RowConstraints;
14 import javafx.scene.layout.StackPane;
15 import javafx.scene.shape.Rectangle;
16 import javafx.scene.text.Font;
17 import javafx.scene.control.ComboBox;
18 import javafx.scene.control.Label;
19 public class MainController {
20     private static WordLadder wordLadder;
21     private static WordLadderGreedy wordLadderGreedy;
22     private static WordLadderUCS wordLadderUCS;
23     private static WordLadderAStar wordLadderAStar;
24     private static Set<String> dict;
25     String selectedAlgorithm;
26     String startWord;
27     String endWord;
28     SimpleEntry<List<String>, Integer> result;
29     @FXML
30     private ComboBox<String> algorithmChoice;
31     @FXML
32     private Label messageLabel;
33     @FXML
34     private TextField StartInput;
35     @FXML
36     private TextField TargetInput;
37     @FXML
38     private StackPane clickableArea,ResultArea;
39     @FXML
40     private TextField widthInput;
41     @FXML
42     private GridPane resultGrid;
43     @FXML
44     private GridPane grid;
45     @FXML
46     private Label rectangleLabel;
47     @FXML
48     private void onRectangleClicked() {
49         startWord = StartInput.getText().trim().toLowerCase();
50         endWord = TargetInput.getText().trim().toLowerCase();
51         selectedAlgorithm = algorithmChoice.getValue();
52         if (startWord.isEmpty() || endWord.isEmpty()) {
53             messageLabel.setText("Error: Both fields must be filled.");
54             return;
55         }
56         if (startWord.length() != endWord.length()) {
57             messageLabel.setText("Error: Words must be of the same length.");
58             return;
59         }
60         if (!dict.contains(startWord) || !dict.contains(endWord)) {
61             messageLabel.setText("Error: Both start and end words must be in the dictionary.");
62             return;
63         }
64         messageLabel.setText("Enjoy the results below!!!");
65         findPath();
66     }
67 }
```

Gambar 3.6.4 Source Code MainController I

```

1  @FXML
2  private void initialize(){
3      try{
4          algorithmChoice.getItems().addAll("UCS", "GBFS", "A*");
5          dict = DictionaryLoader.loadDictionary("src/main/resources/main/tucil_13522019/Dict.txt");
6          wordLadderGreedy = new WordLadderGreedy(dict);
7          wordLadderUCS = new WordLadderUCS(dict);
8          wordLadderAStar = new WordLadderAStar(dict);
9      }catch (IOException e){
10         messageLabel.setText("Dictionary Not Set Properly");
11     }
12 }
13 @FXML
14 private void findPath() {
15     Runtime runtime = Runtime.getRuntime();
16     long startMemoryUsage = runtime.totalMemory() - runtime.freeMemory();
17     long startTime = System.nanoTime();
18     try {
19         switch (selectedAlgorithm) {
20             case "UCS":
21                 result = wordLadderUCS.findLadder(startWord, endWord);
22                 break;
23             case "GBFS":
24                 result = wordLadderGreedy.findLadder(startWord, endWord);
25                 break;
26             case "A*":
27                 result = wordLadderAStar.findLadder(startWord, endWord);
28                 break;
29             default:
30                 messageLabel.setText(String.format("Please Select An Algorithm"));
31                 break;
32         }
33         long endTime = System.nanoTime();
34         long duration = (endTime - startTime) / 1_000_000;
35         long endMemoryUsage = runtime.totalMemory() - runtime.freeMemory();
36         long memoryUsed = (endMemoryUsage - startMemoryUsage);
37         List<String> path = result.getKey();
38         int nodesVisited = result.getValue();
39         if (path.isEmpty()) {
40             messageLabel.setText(String.format("No ladder found (%d nodes visited. Time taken: %d ms. Memory used: %d bytes)", nodesVisited, duration, memoryUsed));
41             return;
42         }
43         updateResultGrid(path);
44         messageLabel.setText(String.format("Found ladder with %d nodes visited. Time taken: %d ms. Memory used: %d bytes", nodesVisited, duration, memoryUsed));
45     } catch (Exception e) {
46         messageLabel.setText(String.format("Please Select An Algorithm"));
47     }
48 }
49 private void updateResultGrid(List<String> results) {
50     resultGrid.getChildren().clear();
51     if (results.isEmpty()) return;
52     int numRows = results.size();
53     int numCols = results.stream().mapToInt(String::length).max().orElse(0) + 1;
54     resultGrid.getColumnConstraints().clear();
55     ColumnConstraints numberColumn = new ColumnConstraints(50);
56     resultGrid.getColumnConstraints().add(numberColumn);
57     for (int col = 0; col < numCols - 1; col++) {
58         ColumnConstraints cc = new ColumnConstraints(80);
59         resultGrid.getColumnConstraints().add(cc);
60     }
61     resultGrid.getRowConstraints().clear();
62     for (int row = 0; row < numRows; row++) {
63         RowConstraints rc = new RowConstraints(80);
64         resultGrid.getRowConstraints().add(rc);
65     }
66     for (int i = 0; i < numRows; i++) {
67         String word = results.get(i);
68         TextField rowLabel = new TextField(String.valueOf(i + 1));
69         rowLabel.setEditable(false);
70         rowLabel.setAlignment(Pos.CENTER);
71         rowLabel.setMinWidth(100);
72         rowLabel.setMinHeight(80);
73         rowLabel.setFont(Font.font("Cooper Black", 36));
74         rowLabel.setStyle("-fx-background-color: transparent; -fx-text-fill: #2A8278;");
75         resultGrid.addRowLabel(0, i);
76         for (int j = 0; j < word.length(); j++) {
77             char letter = word.charAt(j);
78             TextField textField = new TextField(String.valueOf(letter).toUpperCase());
79             textField.setEditable(false);
80             textField.setAlignment(Pos.CENTER);
81             textField.setMinWidth(80);
82             textField.setMinHeight(80);
83             textField.setFont(Font.font("Cooper Black", 36));
84             if (endWord.length() > j && letter == endWord.charAt(j)) {
85                 textField.setStyle("-fx-background-color: #38CB92; -fx-text-fill: white;");
86             } else if (endWord.contains(String.valueOf(letter))) {
87                 textField.setStyle("-fx-background-color: #ffdf00; -fx-text-fill: white;");
88             } else {
89                 textField.setStyle("-fx-background-color: #2A8278; -fx-text-fill: white;");
90             }
91             resultGrid.add(textField, j + 1, i);
92         }
93     }
94 }

```

Gambar 3.6.4 Source Code MainController II

Main.controller mencakup pembuatan interaksi dari GUI dengan user. Fungsi initialize selalu dijalankan sewaktu memulai program, dimana fungsi ini menginisialisasi semua *static*, mulai dari kamus sampai dengan masing masing permainan word ladder. Kemudian terdapat onRectangleClicked yang langsung menerima semua masukan dari komponen input dan *ComboBox*, melakukan validas, dan jika tepat, mencari solusi. Fungsi findPath memanggil fungsi findLadder dari komponen game sesuai dengan masukan dari *ComboBox* dan memberikan tampilan dari sisi *label* dan juga mengubah *grid* dengan *updateGrid*. *updateGrid* mengubah *grid* sesuai dengan hasil dari path, seperti menentukan panjang dan lebar, warna, serta isi dari *grid* tersebut.

Terakhir yaitu style.css yang dibuat untuk mempermudah *styling* secara langsung tanpa mengubah fxml (*styling* juga dapat dilakukan pada fxml).



```
1 .main {
2     -fx-background-color: #FBEBD8;
3 }
4 .grid-result {
5     -fx-font-size: 42px;
6     -fx-hgap: 40;
7     -fx-vgap: 40;
8 }
9 .letter-input {
10    -fx-alignment: center;
11    -fx-text-alignment: center;
12    -fx-padding: 10;
13 }
14
15
16 .text-field {
17     -fx-control-inner-background: #2A8278;
18     -fx-prompt-text-fill: #CCCCCC;
19
20 }
21 .combo-box {
22     -fx-background-color: #2A8278;
23     -fx-text-fill: white;
24 }
25
26 .combo-box .list-cell {
27     -fx-background-color: #0c4d46;
28     -fx-text-fill: white;
29     -fx-font-family: 'Cooper Black';
30 }
31
```

Gambar 3.6.5 Souce Code style.css

BAB IV

ANALISIS DAN PENGUJIAN

Pengujian Algoritma UCS

Test Case 1 UCS

Word Ladder Game

Start Word crime

End Word apple

Found ladder with 5265 nodes visited. Time taken: 28 ms. Memory used: 32931352 bytes

Generate ucs

1	C	R	I	M	E
2	P	R	I	M	E
3	P	R	I	S	E
4	A	R	I	S	E
5	A	N	I	S	E

The screenshot displays a 'Word Ladder Game' interface. At the top, it shows the 'Start Word' as 'crime' and the 'End Word' as 'apple'. Below this, a message indicates that a ladder was found with 5265 nodes visited, taking 28 ms and using 32931352 bytes of memory. There are two buttons: 'Generate' and 'ucs'. The main area shows a 5x5 grid of squares representing the word ladder. Each row corresponds to a step in the ladder, numbered 1 to 5. The columns represent the letters of the words: Step 1 (C, R, I, M, E), Step 2 (P, R, I, M, E), Step 3 (P, R, I, S, E), Step 4 (A, R, I, S, E), and Step 5 (A, N, I, S, E). The letter 'P' in the second row is highlighted in yellow, indicating it is the current node being processed.

Gambar 4.1.1.1 Test Case 1 UCS Tampilan 1

4	A	R	I	S	E
5	A	N	I	S	E
6	A	N	I	L	E
7	A	N	O	L	E
8	A	M	O	L	E
9	A	M	P	L	E
10	A	P	P	L	E

Gambar 4.1.1.2 Test Case 1 UCS Tampilan 2

Test Case 2 UCS

Word Ladder Game

Start Word

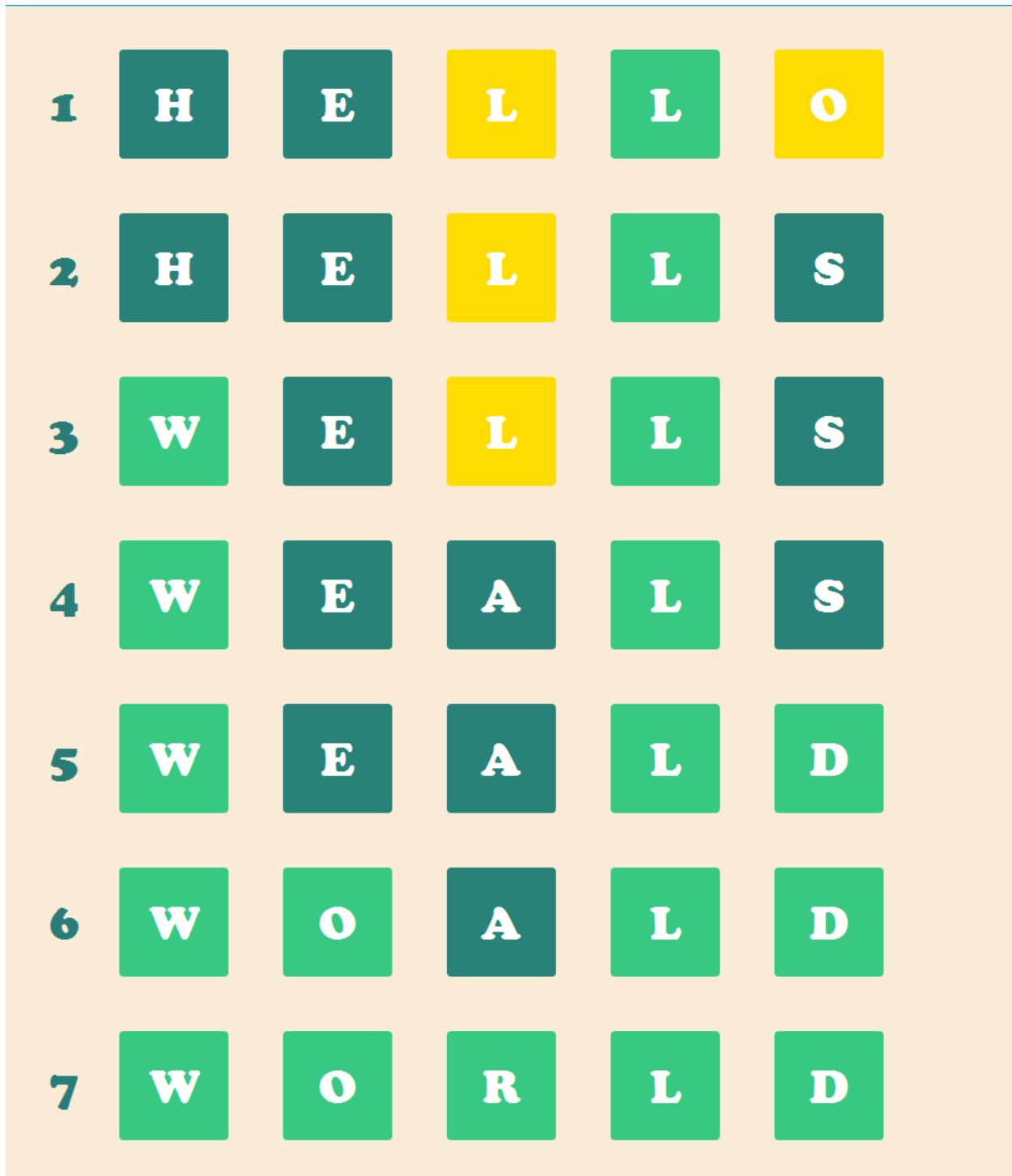
End Word

Found ladder with 1927 nodes visited. Time taken: 11 ms. Memory used: 12446088 bytes

Generate ucs

1	H	E	L	L	O
2	H	E	L	L	S
3	W	E	L	L	S
4	W	E	A	L	S
5	W	E	A	L	D

Gambar 4.1.2.1 Test Case 2 UCS Tampilan 1



Gambar 4.1.2.2 Test Case 2 UCS Tampilan 2

Test Case 3 UCS

Word Ladder Game

Start Word

End Word

Found ladder with 2495 nodes visited. Time taken: 19 ms. Memory used: 15728640 bytes

Generate UCS

1	N	I	G	H	T
2	N	I	G	H	S
3	S	I	G	H	S
4	S	I	N	H	S
5	S	I	N	E	S

Gambar 4.1.3.1 Test Case 3 UCS Tampilan 1

4	S	I	N	H	S
5	S	I	N	E	S
6	P	I	N	E	S
7	P	I	K	E	S
8	P	I	K	E	R
9	P	O	K	E	R
10	P	O	W	E	R

Gambar 4.1.3.2 Test Case 3 UCS Tampilan 2

Test Case 4 UCS

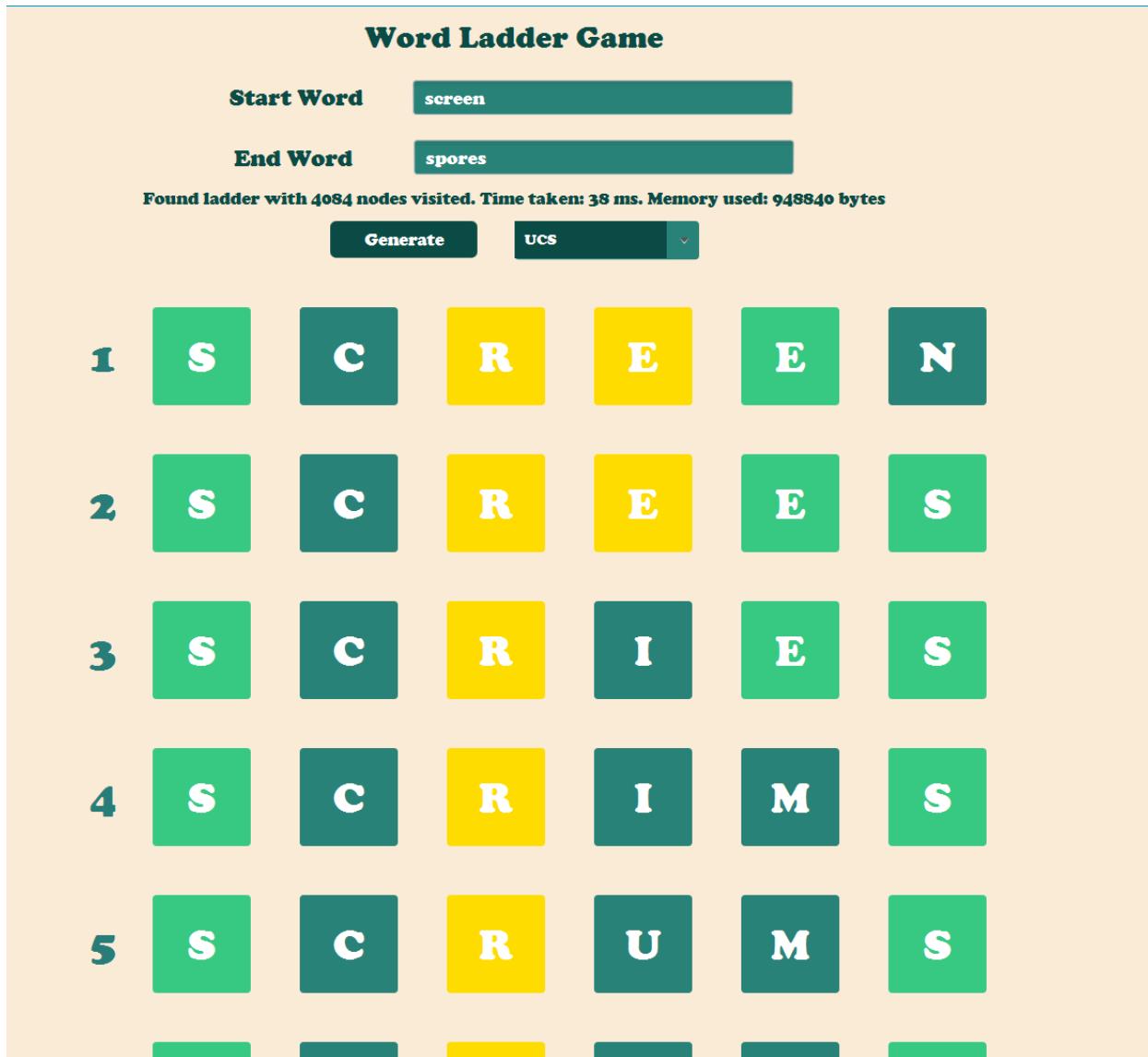


Gambar 4.1.4.1 Test Case 4 UCS Tampilan 1

6	S	U	R	G	E	R
7	S	U	R	F	E	R
8	S	U	F	F	E	R
9	D	U	F	F	E	R
10	D	O	F	F	E	R
11	C	O	F	F	E	R
12	C	O	F	F	E	E

Gambar 4.1.4.2 Test Case 4 UCS Tampilan 2

Test Case 5 UCS



5	S	C	R	U	M	S
6	S	T	R	U	M	S
7	S	T	R	U	T	S
8	S	T	O	U	T	S
9	S	P	O	U	T	S
10	S	P	O	R	T	S
11	S	P	O	R	E	S

Gambar 4.1.5.2 Test Case 5 UCS Tampilan 2

Test Case 6 UCS

Word Ladder Game

Start Word cloth

End Word fruit

Found ladder with 5955 nodes visited. Time taken: 55 ms. Memory used: 2342528 bytes

Generate ucs

1	C	L	O	T	H
2	C	L	O	T	S
3	B	L	O	T	S
4	B	L	A	T	S
5	B	R	A	T	S

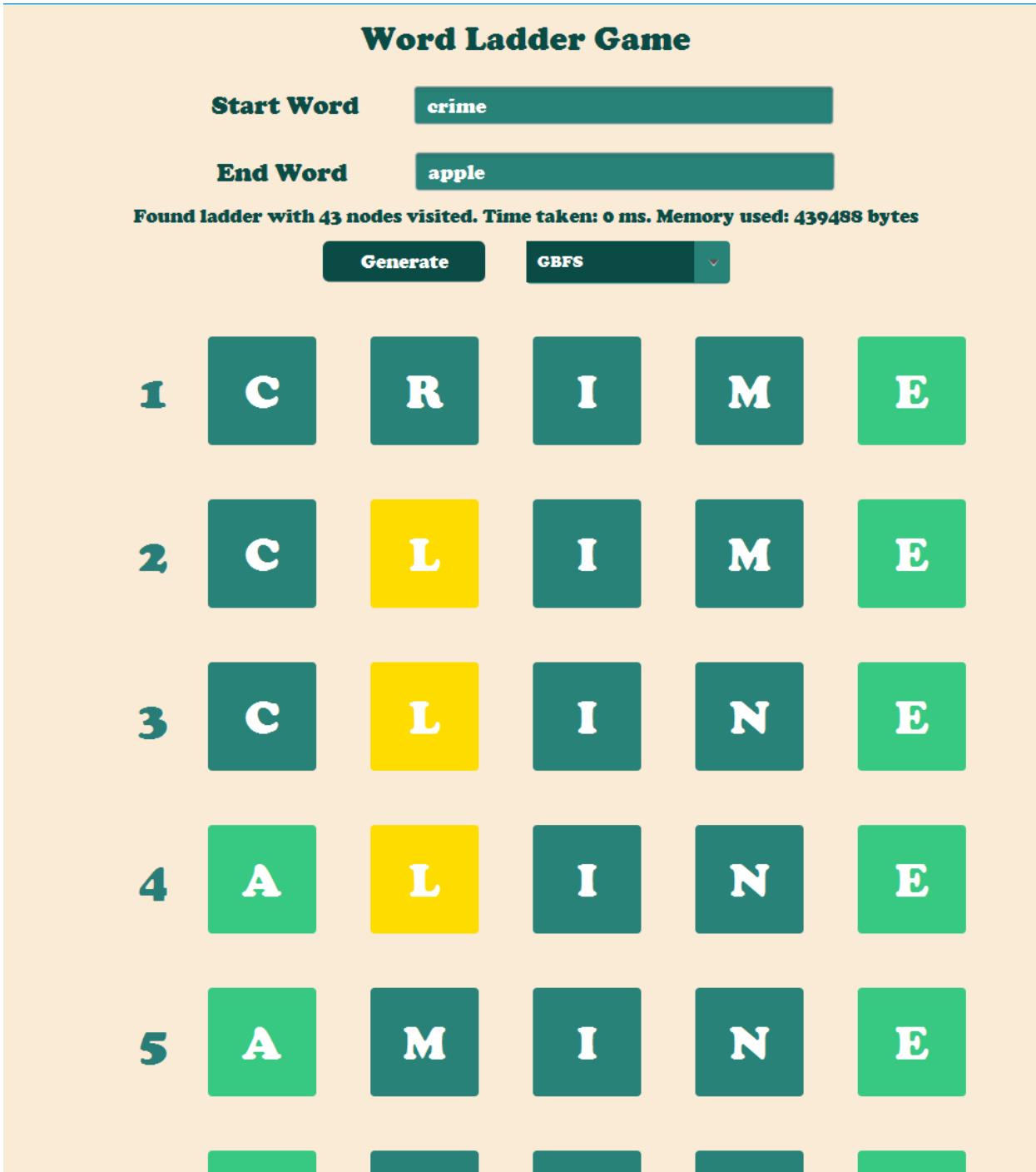
Gambar 4.1.6.1 Test Case 1 UCS Tampilan 1

4	B	L	A	T	S
5	B	R	A	T	S
6	B	R	A	N	S
7	B	R	A	N	T
8	B	R	U	N	T
9	B	R	U	I	T
10	F	R	U	I	T

Gambar 4.1.6.2 Test Case 1 UCS Tampilan 2

Pengujian Algoritma GBFS

Test Case 1 GBFS



Gambar 4.2.1.1 Test Case 1 GBFS Tampilan 1

7	A	B	I	D	E
8	A	B	O	D	E
9	A	N	O	D	E
10	A	N	O	L	E
11	A	M	O	L	E
12	A	M	P	L	E
13	A	P	P	L	E

Gambar 4.2.1.2 Test Case 1 GBFS Tampilan 2

Test Case 2 GBFS

Word Ladder Game

Start Word hello

End Word world

Found ladder with 390 nodes visited. Time taken: 2 ms. Memory used: 2558600 bytes

Generate GBFS

1	H	E	L	L	O
2	H	O	L	L	O
3	H	O	L	L	Y
4	C	O	L	L	Y
5	C	O	O	L	Y

Gambar 4.2.2.1 Test Case 2 GBFS Tampilan 1

16	W	I	L	E	S
17	W	I	L	L	S
18	W	E	L	L	S
19	W	E	A	L	S
20	W	E	A	L	D
21	W	O	A	L	D
22	W	O	R	L	D

Gambar 4.2.2.2 Test Case 2 GBFS Tampilan 2

Test Case 3 GBFS

Word Ladder Game

Start Word night

End Word power

Found ladder with 18 nodes visited. Time taken: 0 ms. Memory used: 524288 bytes

Generate GBFS

1	N	I	G	H	T
2	B	I	G	H	T
3	B	I	G	O	T
4	B	E	G	O	T
5	B	E	G	E	T

Gambar 4.2.3.1 Test Case 3 GBFS Tampilan 1

6	B	E	S	E	T
7	R	E	S	E	T
8	R	O	S	E	T
9	R	O	S	E	D
10	P	O	S	E	D
11	P	O	S	E	R
12	P	O	W	E	R

Gambar 4.2.3.2 Test Case 3 GBFS Tampilan 2

Test Case 4 GBFS

Word Ladder Game

Start Word

End Word

Found ladder with 27 nodes visited. Time taken: 0 ms. Memory used: 524288 bytes

Generate **GBFS** ▾

1	S	C	R	E	E	N
2	S	C	R	E	E	S
3	S	A	R	E	E	S
4	S	A	R	G	E	S
5	B	A	R	G	E	S

Gambar 4.2.4.1 Test Case 4 GBFS Tampilan 1

11	P	O	N	G	E	D
12	P	O	N	G	E	E
13	C	O	N	G	E	E
14	C	O	N	G	E	R
15	C	O	N	F	E	R
16	C	O	F	F	E	R
17	C	O	F	F	E	E

Gambar 4.2.4.2 Test Case 4 GBFS Tampilan 2

Test Case 5 GBFS

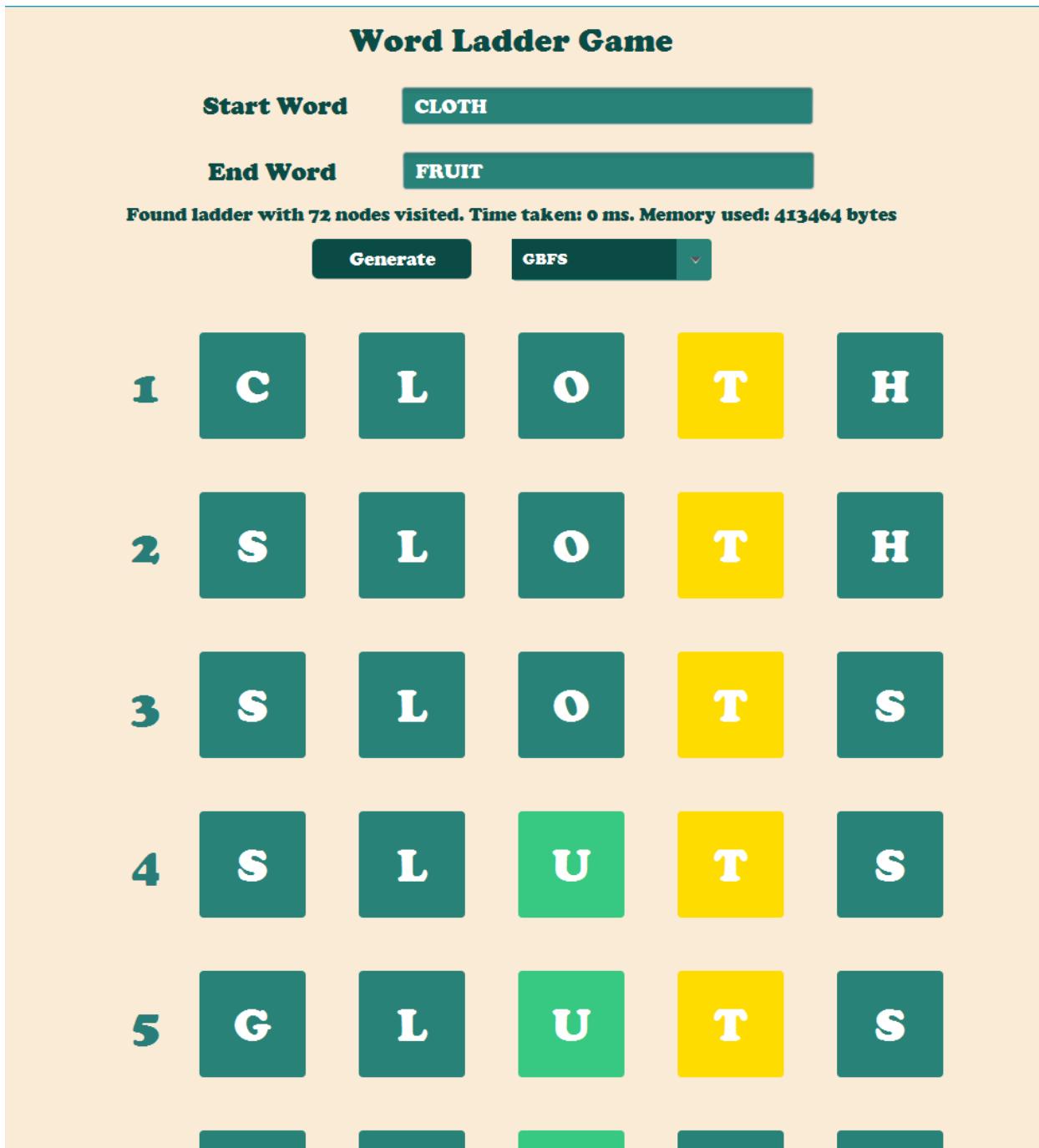


Gambar 4.2.5.1 Test Case 5 GBFS Tampilan 1

10	S	P	O	I	L	S
11	S	P	O	O	L	S
12	S	P	O	O	K	S
13	S	T	O	O	K	S
14	S	T	O	R	K	S
15	S	T	O	R	E	S
16	S	P	O	R	E	S

Gambar 4.2.5.2 Test Case 5 GBFS Tampilan 2

Test Case 6 GBFS



Gambar 4.2.6.1 Test Case 6 GBFS Tampilan 1

16 **G** **R** **I** **F** **T**

17 **G** **R** **A** **F** **T**

18 **G** **R** **A** **N** **T**

19 **G** **R** **U** **N** **T**

20 **B** **R** **U** **N** **T**

21 **B** **R** **U** **I** **T**

22 **F** **R** **U** **I** **T**

Gambar 4.2.6.2 Test Case 6 GBFS Tampilan 2

Pengujian Algoritma A*

Test Case 1 A*



Gambar 4.3.1.1 Test Case 1 A* Tampilan 1

4	A	R	I	S	E
5	A	N	I	S	E
6	A	N	I	L	E
7	A	N	O	L	E
8	A	M	O	L	E
9	A	M	P	L	E
10	A	P	P	L	E

Gambar 4.3.1.2 Test Case 1 A* Tampilan 2

Test Case 2 A*

Word Ladder Game

Start Word hello

End Word world

Found ladder with 49 nodes visited. Time taken: 0 ms. Memory used: 135888 bytes

Generate A* ↴

1	H	E	L	L	O
2	H	E	L	L	S
3	W	E	L	L	S
4	W	E	A	L	S
5	W	E	A	L	D

Gambar 4.3.2.1 Test Case 2 A* Tampilan 1

1	H	E	L	L	O
2	H	E	L	L	S
3	W	E	L	L	S
4	W	E	A	L	S
5	W	E	A	L	D
6	W	O	A	L	D
7	W	O	R	L	D

Gambar 4.3.2.2 Test Case 2 A* Tampilan 2

Test Case 3 A*

Word Ladder Game

Start Word night

End Word power

Found ladder with 106 nodes visited. Time taken: 1 ms. Memory used: 690784 bytes

Generate A* ▾

1	N	I	G	H	T
2	N	I	G	H	S
3	S	I	G	H	S
4	S	I	N	H	S
5	S	I	N	E	S
<hr/>					

Gambar 4.3.3.1 Test Case 4 A* Tampilan 1

4	S	I	N	H	S
5	S	I	N	E	S
6	P	I	N	E	S
7	P	I	K	E	S
8	P	I	K	E	R
9	P	O	K	E	R
10	P	O	W	E	R

Gambar 4.3.3.2 Test Case 4 A* Tampilan 2

Test Case 4 A*

Word Ladder Game

Start Word screen

End Word coffee

Found ladder with 265 nodes visited. Time taken: 2 ms. Memory used: 2028224 bytes

Generate A* ↴

1	S	C	R	E	E	N
2	S	C	R	E	E	S
3	S	A	R	E	E	S
4	S	A	R	G	E	S
5	M	A	R	G	E	S

Gambar 4.3.4.1 Test Case 4 A* Tampilan 1

6	M	A	N	G	E	S
7	M	A	N	G	E	R
8	M	O	N	G	E	R
9	C	O	N	G	E	R
10	C	O	N	F	E	R
11	C	O	F	F	E	R
12	C	O	F	F	E	E

Gambar 4.3.5.2 Test Case 4 A* Tampilan 2

Test Case 5 A*

Word Ladder Game

Start Word screen

End Word spores

Found ladder with 183 nodes visited. Time taken: 1 ms. Memory used: 1284840 bytes

Generate A*

1	S	C	R	E	E	N
2	S	C	R	E	E	S
3	S	C	R	I	E	S
4	S	C	R	I	M	S
5	S	C	R	U	M	S

Gambar 4.3.5.1 Test Case 5 A* Tampilan 1

5	S	C	R	U	M	S
6	S	T	R	U	M	S
7	S	T	R	U	T	S
8	S	T	O	U	T	S
9	S	P	O	U	T	S
10	S	P	O	R	T	S
11	S	P	O	R	E	S

Gambar 4.3.5.2 Test Case 5 A* Tampilan 2

Test Case 6 A*

Word Ladder Game

Start Word cloth

End Word fruit

Found ladder with 552 nodes visited. Time taken: 3 ms. Memory used: 3775512 bytes

Generate A* ▾

1	C	L	O	T	H
2	C	L	O	T	S
3	B	L	O	T	S
4	B	L	A	T	S
5	B	R	A	T	S

Gambar 4.3.6.1 Test Case 6 A* Tampilan 1



Gambar 4.3.6.2 Test Case 6 A* Tampilan 2

Pengelompokkan Data

Tabel 4.1 Tabel Pengelompokkan Data

Nomor	Algoritma	Waktu (ms)	Memory (byte)	Panjang Solusi (kata)	Banyak node yang dikunjungi

1	UCS	28	32931352	10	5265
1	GBFS	0	439488	13	43
1	A*	4	3862944	10	558
2	UCS	11	12446088	7	1927
2	GBFS	2	2558600	22	390
2	A*	2	135888	7	49
3	UCS	19	15728640	10	2495
3	GBFS	0	524288	12	18
3	A*	1	690784	10	106
4	UCS	37	39323648	12	5324
4	GBFS	0	524288	17	27
4	A*	2	2028224	12	265
5	UCS	38	948840	11	4084
5	GBFS	0	414816	16	39
5	A*	1	1284840	11	183
6	UCS	55	2342528	10	5955
6	GBFS	0	413464	22	72
6	A*	3	3775512	10	552

Analisis Data

Dalam proses pengetesan, terdapat adanya ketidak-konsisten dalam pengukuran waktu serta memory mengingat kondisi perangkat sewaktu perjalanan program berbeda beda akibat dilakukan dalam waktu yang berbeda sehingga memberikan hasil yang tidak selaras untuk tiap *test case*. Namun, terdapat beberapa poin yang dapat kita ambil dengan melihat tabel diatas:

1. Algoritma GBFS akan selalu memiliki panjang lintasan yang terpanjang. Hal ini mengingat bahwa GBFS tidak memperhatikan kebenaran ataupun ke-optimalan penentuan kata, namun hanya berfokus pada *heuristic* yang dimiliki. Disisi lain, akibat

konsepnya yang terus terang, GBFS umumnya memiliki *runtime* yang paling pendek, bahkan sampai dibawah 0,1 ms sehingga tidak terukur. GBFS cenderung mengunjungi node paling sedikit akibat pendekatan yang bersifat satu arah membuat GBFS tidak perlu mengekspansi pencarian ke node dari branch lain.

2. Algoritma UCS memiliki ukuran *path* yang sama dengan A*. Hal ini dikarenakan pencarian meluas yang memperhitungkan optimalitas jawaban sehingga memberikan jawaban yang optimal. Namun akibat pembawaan yang meluas dan tanpa memilah (hanya mengandalkan $g(n)$) , menyebabkan UCS cenderung memiliki *runtime* yang paling lama serta konsumsi memori yang sangat tinggi. Terakhir, UCS memiliki jumlah node dikunjungi paling banyak. Hal ini diakibatkan oleh pendekatan yang berfokus ekspansi pencarian pada seluruh *branch* sehingga membuat Algoritma UCS selalu memiliki jumlah *node* dikunjungi paling banyak.
3. Algoritma A* menerapkan kedua konsep poin diatas dan disatukan. Oleh karena itu, A* juga memiliki jawaban yang optimal, dimana terlihat dalam tabel A* memiliki jawaban dengan jumlah percobaan terpendek. Perlu diingat bahwa konsep pemilihan *node* pada algoritma ini juga bergantung pada *heuristic*, namun dalam program ini , dapat dibilang memberikan hasil optimal. A* memiliki efisiensi waktu yang lebih baik dari UCS akibat dari penentuan *heuristic* ($f(n) = g(n) + h(n)$) , namun lebih lambat dari GBFS akibat mengevaluasi lebih banyak node. Selain itu, dalam pengukuran memori, hal ini bervariasi, dimana tergantung dari persoalan yang diberikan. Pendekatannya yang lebih terarah sering membuatnya lebih memori efisien daripada UCS. A* cenderung menyimpan node yang relevan berdasarkan biaya dan heuristik, yang mengurangi bloat memori dibandingkan UCS, namun tidak konsisten dan bergantung pada persoalan jika dibandingkan pada GBFS. Sama halnya dengan node yang dikunjungi, A* umumnya mengunjungi node lebih banyak dari GBFS, namun terdapat kasus dimana A* tidak perlu terlalu luas dalam mencari jawaban sehingga membuat A* mengunjungi lebih sedikit *node* dari GBFS dan UCS.

Dari analisis dan data, jika ingin dirangkum dan diurutkan, maka akan didapat 4 *cluster* sebagai berikut:

1. Kecepatan Eksekusi

Urutan pertama: Greedy Best-First Search (GBFS) cenderung paling cepat karena GBFS memprioritaskan node yang paling "menjanjikan" berdasarkan heuristik yang hanya memperhitungkan jarak dari node ke tujuan. Ini seringkali membuatnya bergerak langsung menuju target tanpa mempertimbangkan path yang telah dilewati, yang bisa cepat tapi tidak selalu efektif.

Urutan kedua: A* menggabungkan pendekatan UCS dan GBFS dengan menggunakan fungsi $f(n) = g(n) + h(n)$ di mana $g(n)$ adalah biaya dari awal dan $h(n)$ adalah heuristik ke tujuan. Ini memungkinkan A* untuk mencari lebih efisien dari GBFS karena menghindari jalan memutar yang mahal, namun biasanya lebih lambat dari GBFS karena mengevaluasi lebih banyak node.

Urutan ketiga: UCS beroperasi dengan mengiterasi semua jalur yang memungkinkan dan selalu memperluas node dengan "biaya" terendah, tanpa memperhitungkan heuristik ke tujuan. Ini membuatnya seringkali paling lambat karena tidak memiliki panduan heuristik untuk mengarahkan pencarinya.

2. Penggunaan Memori

Urutan pertama: Karena UCS memeriksa semua jalur yang memungkinkan, ia dapat mempertahankan banyak node dalam antrian, terutama dalam pohon pencarian yang luas atau dalam, menghasilkan penggunaan memori yang tinggi.

Urutan kedua/ketiga: GBFS bisa jadi lebih hemat memori dari UCS karena tidak menyimpan semua jalur yang memungkinkan dengan sama rata. Namun, jika heuristic tidak efektif, GBFS bisa berakhir menjelajah lebih banyak node daripada A*.

Urutan kedua/ketiga : Meskipun A* umumnya menggunakan memori lebih banyak dari GBFS dalam beberapa kasus, pendekatannya yang lebih terarah sering membuatnya lebih memori efisien daripada UCS (sesuai dengan data). A* cenderung menyimpan node yang relevan berdasarkan biaya dan heuristik, yang mengurangi bloat memori dibandingkan UCS.

3. Optimalitas Solusi

Urutan Pertama/Kedua: UCS dijamin menemukan solusi paling optimal tetapi dengan biaya waktu dan memori.

Urutan Pertama/Kedua: A* dijamin menemukan jalur optimal asalkan heuristiknya adalah admissible (tidak pernah melebih-lebihkan jarak sebenarnya ke tujuan). Akibat

dari hasil yang sama dengan UCS, maka kita dapat mengatakan bahwa pemilihan *heuristic* sudah tepat dan sesuai dengan standar yang tepat.

Urutan Ketiga: Solusi yang dihasilkan oleh GBFS tidak dijamin optimal karena fokus utama pada heuristik bisa mengarahkan pencarian menjauh dari jalur optimal.

4. Jumlah node yang dikunjungi

Urutan pertama: UCS mengunjungi node paling banyak akibat konsep pencarian yang berakar akar sehingga menambah peluang untuk mengunjungi suatu *node*.

Urutan kedua: A* memiliki konsep *branching* yang mirip dengan UCS namun mengandalkan *heuristic* sehingga *node* yang dikunjungi akan lebih sedikit dari UCS.

Urutan ketiga : GBFS hanya berfokus pada penggunaan *heuristic* dalam pencarian jawaban. Hal ini membuat GBFS cenderung mengunjungi node yang lebih sedikit karena langsung bergerak menuju tujuan berdasarkan estimasi jarak paling dekat saja.

BAB V

KESIMPULAN DAN SARAN

Kesimpulan

Dalam menghadapi tantangan menemukan jalur optimal dalam permainan Word Ladder, tiga algoritma pencarian—*Uniform Cost Search* (UCS), *Greedy Best-First Search* (GBFS), dan A*—menawarkan pendekatan yang berbeda dengan kelebihan dan kelemahan masing-masing. UCS adalah model yang sangat *robust* yang menjamin penemuan jalur terpendek melalui ekspansi seragam semua node berdasarkan biaya kumulatif dari titik start, tanpa mempertimbangkan estimasi jarak ke tujuan. Meskipun efektif dalam menjamin solusi yang optimal, UCS sering kali kurang efisien karena tidak memanfaatkan heuristik untuk memprioritaskan node yang paling mungkin mendekatkan solusi, sehingga mengakibatkan penelusuran yang lebih luas dan pemrosesan yang lebih intensif.

Di sisi lain, GBFS menggunakan heuristik untuk secara agresif mencari jalur menuju tujuan dengan memprioritaskan node yang secara heuristik paling dekat dengan tujuan, mengabaikan biaya total yang telah dikeluarkan untuk mencapai node tersebut. Pendekatan ini dapat lebih cepat mencapai tujuan tetapi tidak selalu menjamin penemuan jalur yang paling efisien atau paling pendek, terutama dalam kasus di mana heuristik mungkin menyesatkan. A*, menggabungkan kekuatan dari kedua algoritma sebelumnya, menawarkan solusi yang lebih seimbang. Dengan menggunakan fungsi biaya $f = g + h$, di mana g adalah biaya dari start dan h adalah heuristik ke tujuan, A* mampu meminimalkan jumlah node yang tidak relevan yang dieksplorasi dan mengarahkan pencarian secara efektif menuju tujuan, menjadikannya pilihan yang lebih disukai ketika keseimbangan antara kecepatan dan akurasi adalah prioritas utama.

Meskipun begitu, pemilihan algoritma yang tepat harus didasarkan pada analisis mendalam tentang karakteristik masalah yang dihadapi, termasuk memahami kompleksitas masalah, keterbatasan sumber daya, dan persyaratan kinerja. Memilih algoritma yang paling cocok dapat secara signifikan meningkatkan efektivitas solusi, mengurangi sumber daya yang digunakan, dan mempercepat waktu pencarian, sambil tetap memastikan bahwa hasil yang diperoleh adalah yang terbaik yang mungkin dicapai dalam kondisi yang diberikan.

Saran

Saran dalam penggerjaan tugas besar ini mencakup agar mencari tahu terlebih dahulu mengenai berbagai *Build Tools* dari suatu bahasa pemrograman untuk memastikan bekerja dengan sesuai pada berbagai Code Editor. Selain itu, akan lebih baik jika disediakan sebuah *benchmark* untuk mendukung analisis yang lebih teratur, diakibatkan karena sering kali pengumpulan data bersifat rancu , yang diakibatkan oleh perangkat sendiri.

BAB VI

DAFTAR PUSTAKA

- [1] Munir, Rinaldi. Penentuan rute (Route/Path Planning) - Bagian 1. Diakses 4 Mei 2024.
<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf>
- [2] Munir, Rinaldi. Penentuan rute (Route/Path Planning) - Bagian 2. Diakses 4 Mei 2024.
<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf>

BAB VII

LAMPIRAN

Github Repository

https://github.com/Razark-Y/Tucil3_13522019

Kelayakan Program

Tabel 7.1 Tabel Kelayakan Program

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 optim	✓	
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	✓	