

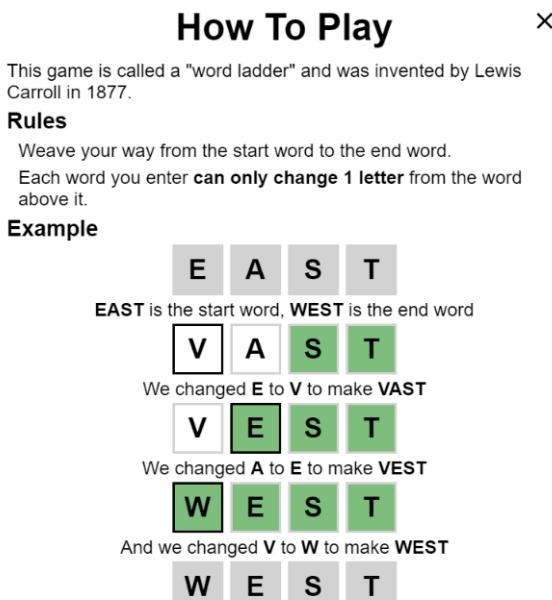
Penyelesaian Permainan Word Ladder Menggunakan Algoritma UCS, Greedy Best First Search, dan A*

Disusun oleh
Farhan Nafis Rayhan - 13522037

Deskripsi Masalah

1.1. Word Ladder

Word ladder (juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragraphs, 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.



[Privacy](#)
[Privacy Policy](#)

Gambar 1.1. Illustrasi permainan Word Ladder
(Sumber: <https://wordwormdormdork.com/>)

1.2. Word Ladder Solver

Pada permasalahan kali ini, saya ingin membuat sebuah solver permainan tersebut dengan harapan kita dapat menemukan solusi paling optimal untuk menyelesaikan permainan Word Ladder ini. Solver ini akan dibuat dalam bentuk program *offline* berbasis *Command Line Interface (CLI)*. Program akan menggunakan algoritma pencarian rute agar menemukan solusi optimal.

Analisis Algoritma

2.1. Route Planning

Route planning merupakan salah satu permasalahan yang sering muncul pada dunia informatika. Permasalahan ini melibatkan pencarian suatu *path* atau rute yang memberikan suatu *cost* (harga) paling minimal atau *profit* (keuntungan) paling optimal. Persoalan *route planning* sering digambarkan sebagai suatu graph, dan akan dilakukan penjelajahan dari simpul awal menuju sebuah simpul tujuan. Penjelajahan dilakukan dengan cara memilih suatu simpul untuk ditelusuri cabangnya, dengan suatu aturan pemilihan tertentu.

Algoritma Route Planning sering diklasifikasi menjadi 2 jenis, yaitu *Uninformed Search* serta *Informed Search*. Apabila pencarian dilakukan tanpa informasi tambahan dari luar, maka ia termasuk *Uninformed Search*. Algoritma yang termasuk dalam tipe ini adalah BFS, DFS, IDS, dan yang akan digunakan pada program ini adalah *UCS (Uniform Cost Search)*. Sedangkan *Informed Search* melibatkan adanya pengetahuan/informasi heuristik yang digunakan dalam mencari rute. Contoh dari jenis ini adalah *Greedy Best First Search* serta *A**, dimana keduanya akan digunakan pada program ini.

Secara teori, ketiga algoritma yang digunakan akan memiliki kompleksitas waktu dan ruang yang sama yaitu $O(b^m)$. Pada hal ini, b merupakan *branching factor*, yaitu rata-rata banyaknya tetangga pada setiap simpul. Sedangkan, m adalah kedalaman pencarian, atau maksimum banyaknya langkah pada permainan ini. Kompleksitas ruang bernilai sama karena program perlu menyimpan state dari setiap simpul yang hidup. Namun secara realita, beberapa algoritma bisa saja berjalan lebih baik dari kompleksitas waktu tersebut, karena banyak simpul yang dikunjungi dapat jauh lebih sedikit pada beberapa algoritma

Ketiga algoritma yang akan digunakan menggunakan pencarian yang serupa tetapi tidak sama. Pada setiap langkahnya, simpul yang dipilih untuk

ditelusuri cabangnya ditentukan melalui fungsi evaluasi $f(n)$. Dengan fungsi evaluasi, dapat ditentukan urutan prioritas dari simpul yang dapat ditelusuri selanjutnya. Letak perbedaan utama antara ketiga algoritma ini adalah bagaimana kita mengevaluasi nilai $f(n)$ suatu simpul. Akibatnya, untuk implementasi ketiga algoritma ini kita dapat menggunakan algoritma yang serupa dengan struktur data yang sama, yaitu *Priority Queue*.

2.2. Analisis Algoritma

2.2.1. Uniform Cost Search

Pada dasarnya Algoritma UCS merupakan kombinasi antara BFS dan IDS. letak perbedaan utama antara algoritma ini dengan kedua algoritma adalah *cost* tidak harus berupa kedalaman pencarian. Pada algoritma ini, digunakan fungsi evaluasi $f(n) = g(n)$, dimana nilai $g(n)$ suatu simpul adalah jumlah seluruh *cost* pada simpul yang dilalui untuk mencapai simpul n dari simpul awal. Algoritma ini termasuk *Uninformed Search* karena fungsi evaluasi masih sistematis dan tidak memerlukan suatu pengetahuan heuristik.

Untuk pencarian solusi permainan *Word Ladder*, digunakan *cost* berupa banyaknya langkah untuk mencapai kata tujuan dari kata awal, dimana kita ingin meminimalkan *cost* pencarian. Karena $g(n)$ merupakan jumlah seluruh *cost* untuk mencapai simpul n , nilai dari $g(n)$ akan sama dengan kedalaman simpul n pada pencarian. Ingat bahwa simpul yang dipilih pada setiap langkah adalah yang memiliki *cost*, atau kedalaman minimum. Sehingga urutan pembangkitan path adalah satu-persatu simpul sampai sebuah kedalaman ditelusuri sepenuhnya. Jadi, dapat disimpulkan bahwa **algoritma UCS pada permainan ini akan sama seperti pencarian pada BFS**.

2.2.2. Greedy Best First Search

Algoritma ini merupakan salah satu algoritma *Informed Search* yang pertama diciptakan. Pada Greedy BFS, digunakan pula fungsi evaluasi $f(n) = h(n)$, tetapi $h(n)$ disini merupakan fungsi heuristik. $h(n)$ merupakan suatu fungsi yang mengestimasi *total cost* dari simpul n menuju simpul tujuan. Hal ini mengakibatkan *Greedy*

BFS menelusuri simpul yang terlihat menjanjikan, yaitu terlihat paling dekat menuju tujuan berdasarkan heuristik.

Untuk kasus permainan *Word Ladder* ini, dipilih heuristik berupa *hamming distance*, atau jarak diantara 2 kata dengan panjang sama. Definisi *hamming distance* sendiri adalah banyaknya huruf yang terletak pada posisi sama namun berbeda simbolnya. Sebagai contoh, *hamming distance(Green, Grape) = 3*. Pada permainan ini akan didefinisikan $h(n)$ berupa besarnya hamming distance antara kata pada simpul n dengan kata tujuan.

Namun, amati bahwa algoritma ini memiliki banyak kelemahan. Penggunaan fungsi heuristik tanpa pembatas yang sistematis mengakibatkan pencarian yang dapat terjebak pada *local minima*. Hal ini juga berakibat pada penelusuran yang memfokuskan pada cabang yang terlihat paling menjanjikan, tetapi belum tentu cabang tersebut berujung pada simpul tujuan. Pada *Word Ladder*, algoritma ini sangat mungkin lebih memilih menelusuri kata yang memiliki hamming distance terkecil, padahal belum tentu apabila hamming distance sudah kecil, akan ada jalur yang membawanya pada tujuan dari kata tersebut. Hal ini lah yang mengakibatkan **algoritma Greedy BFS belum tentu menemukan solusi yang paling optimal.**

2.2.3. A*

Algoritma ini merupakan salah satu optimasi dari Greedy BFS. Pada algoritma ini, digunakan pula fungsi heuristik $h(n)$, tetapi $g(n)$ pun ikut menjadi konsiderasi untuk simpul yang dipilih. Fungsi evaluasi yang digunakan secara formal dapat ditulis sebagai

$$f(n) = g(n) + h(n)$$

Sehingga, bisa dibilang bahwa algoritma A* adalah algoritma pencarian heuristik yang lebih sistematis, karena tidak akan mengekspansi jalur yang telah terlalu mahal.

Algoritma A* pada program *Word Ladder Solver* menggunakan fungsi $h(n)$ dan $g(n)$ seperti yang telah didefinisikan sebelumnya. Pada setiap langkah pencarian, algoritma ini memilih simpul dengan $f(n)$ minimum, yaitu estimasi banyaknya langkah permainan minimum yang diperlukan untuk mencapai kata akhir apabila melewati simpul n. Definisikan $h^*(n)$ sebagai langkah

minimal sebenarnya untuk mencapai tujuan apabila melewati n. Amati bahwa memilih hamming distance sebagai heuristik akan mengakibatkan $h(n) \leq h^*(n)$, hal ini disebabkan hamming distance memberikan banyaknya huruf yang belum sesuai, dengan kata lain perlu diubah satu-persatu hingga menjadi kata tujuan. Maka, tidak mungkin jika *cost* sebenarnya pada langkah optimal lebih kecil dari hamming distance. **Akibatnya, fungsi heuristik $h(n)$ pada kasus ini bisa dibilang admissible, yang mengakibatkan solusi yang ditemukan oleh A* pasti optimal.**

Sadari juga bahwa perbedaan utama antara algoritma ini dengan UCS adalah penggunaan heuristik $h(n)$. Hal ini memberikan informasi kepada pencarian mengenai simpul mana yang lebih baik untuk dijelajahi terlebih dahulu saat banyak simpul memiliki nilai $g(n)$ yang sama. Akibatnya, pencarian akan selalu diarahkan lebih dekat menuju simpul tujuan, tetapi tidak akan ditelusuri terlalu jauh apabila sudah mahal. Berbeda dengan UCS yang tidak memiliki urutan penelusuran yang jelas apabila banyak simpul memiliki nilai $g(n)$ yang sama. Pada kasus ini algoritma UCS akan menelusuri simpul yang jauh lebih banyak bahkan sebelum menemukan cabang yang akan membawa pada solusi. Jadi, bisa disimpulkan bahwa secara teori seharusnya **algoritma A* akan lebih efektif dibandingkan algoritma UCS.**

2.3. Implementasi Algoritma

Program ini mengimplementasikan algoritma UCS, Greedy BFS, dan A*. Ketiga algoritma menggunakan struktur data yang sama yaitu *priority queue*. Karena letak perbedaan ketiga algoritma ini hanya pada perhitungan $f(n)$, penjelasan implementasi ketiga algoritma akan dijelaskan bersamaan sebagai berikut:

2.3.1. Algoritma Utama

1. Masukkan simpul kata awal kepada priority queue yang masih kosong, tandai bahwa simpul ini sudah dikunjungi, catat *parent* nya sebagai kosong.

2. Selama priority queue belum kosong, keluarkan simpul yang terletak paling depan (prioritas tertinggi), simpul ini merupakan simpul ekspan
3. Bangkitkan semua simpul yang berupa kata valid dan hanya berbeda 1 huruf dari simpul ekspan. Hal ini dilakukan dengan cara merubah setiap huruf satu-persatu dari huruf A-Z, periksa apabila ia terdapat dalam database kata yang valid, dan apabila ia belum dikunjungi, simpul ini dibangkitkan sebagai simpul hidup.
4. Periksa apakah simpul ini merupakan simpul kata tujuan, apabila iya, hentikan pencarian.
5. Kalkulasi nilai $f(n)$ dari simpul ini. Implementasi perhitungan akan berbeda beda bergantung pada algoritma yang digunakan. Nilai $g(n)$ diperoleh dengan menambahkan 1 pada $g(n)$ simpul ekspan. Nilai $h(n)$ diperoleh dengan memanggil algoritma hamming distance untuk kata pada simpul ini dengan kata tujuan.
6. Semua simpul hidup yang baru dibangkitkan dicatat bahwa telah dikunjungi, catat pada simpul ini bahwa *parent*-nya adalah simpul ekspan saat ini. Lalu masukkan simpul ini ke dalam priority queue.
7. Semua simpul di dalam priority queue akan diurutkan kembali sehingga yang terletak paling depan adalah yang memiliki prioritas tertinggi (nilai $f(n)$ minimal).
8. Algoritma akan diulang kembali dengan mengeluarkan simpul hidup paling depan priority queue sebagai simpul ekspan baru. Hal ini dilakukan sampai kata tujuan ditemukan atau priority queue kosong, apabila ini terjadi bisa dipastikan solusi tidak ditemukan.

Seperti yang telah dijelaskan sebelumnya, nilai $h(n)$ ditentukan melalui hamming distance antara kata pada simpul n dengan kata tujuan. Penjelasan algoritma Hamming Distance adalah sebagai berikut:

2.3.2. Algoritma Hamming Distance

1. Periksa apakah kata yang ingin dihitung hamming distance-nya memiliki panjang kata yang sama dengan kata tujuan, apabila tidak algoritma berhenti.
2. Buat variabel counter yang mulanya 0.
3. Periksa apakah huruf awal kedua kata sama, tambahkan counter apabila ternyata tidak.
4. Lanjutkan pada huruf selanjutnya di kedua kata, tambahkan counter apabila kedua huruf tidak sama.
5. Iterasi diteruskan sampai huruf terakhir di kedua kata.
Hamming Distance adalah nilai akhir variabel counter.

Setelah simpul terakhir ditemukan, jalur solusi perlu di konstruksi kembali untuk ditampilkan. Hal ini dapat dilakukan dengan algoritma berikut:

2.3.3. Algoritma Rekonstruksi Solusi

1. Mulai dari simpul tujuan, catat simpul *parent* yang tertulis pada simpul tersebut. Masukkan simpul ini pada sebuah larik.
2. Lakukan sebuah perulangan selama *parent* yang tertulis tidak kosong.
3. Lakukan pengecekan pada simpul yang sebelumnya dicatat sebagai *parent*, lihat siapakah *parent*-nya, lalu masukkan kedalam larik.
4. Ulangi proses tersebut sampai ditemukan simpul yang *parent*-nya kosong, ia pasti merupakan simpul awal. Masukkan simpul awal ke dalam larik.
5. Balikkan urutan pada larik ini, diperoleh urutan dari *path* solusi.

Implementasi dalam bahasa Java

3.1. Repository Program

Repository program ini dapat diakses melalui pranala berikut:

https://github.com/Farhannr28/Tucil3_13522037

Penulis memilih untuk menggunakan Bahasa Java karena performanya yang cukup cepat baik untuk program pencarian rute. Selain itu Java juga dilengkapi dengan berbagai package yang bisa memudahkan pemrograman. Program ini tidak mengimplementasikan GUI, dan hanya sebatas Command Line Interface. Salah satu package yang digunakan pada

program ini adalah java.io yang memungkinkan pembacaan dari file berekstensi .txt serta melakukan print berwarna yang menambah nilai estetika program.

Untuk daftar kata bahasa inggris yang valid, saya menggunakan dictionary yang disediakan pada dokumentasi Oracle. Alasan digunakannya daftar ini adalah karena dokumentasi ini memiliki kata yang cukup lengkap dalam kamus bahasa inggris namun tidak terlalu banyak kata yang kurang lazim. Daftar kata ini dapat diakses melalui pranala <https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/dictionary.txt>

3.2. Struktur Data

Program yang diimplementasikan menggunakan 5 buah struktur data utama. Dimana 2 struktur data yang digunakan algoritma telah tersedia dalam oleh bahasa java dalam package java.util, yaitu *HashMap* dan *Priority Queue*. Sedangkan terdapat 3 struktur data lain yang diimplementasikan sendiri yaitu *Trie*, *Adjacency List*, dan *SearchNode*

3.2.1. SearchNode

Utamanya, program menyimpan informasi mengenai sebuah simpul dalam struktur data ini. Struktur ini merepresentasikan simpul dalam graf pencarian. Struktur ini terletak pada class **SearchNode.java** dengan attribute nilai g, nilai h, kata yang direpresentasikan, serta reference menuju simpul *parent*.

3.2.2. HashMap

Map merupakan struktur data berbentuk *key-value pair*, dimana sebuah value disimpan dengan sebuah key sebagai identitasnya. Pada program ini, saya memilih menggunakan *HashMap* karena implementasinya yang menggunakan lebih sedikit menggunakan memori dibanding jenis Map lainnya yang disediakan Java. Untuk program ini, *HashMap* akan menyimpan pasangan dengan sebuah kata yang direpresentasikan sebagai key dan nilai f sebagai valuenya. Implementasi dan penggunaan struktur ini terdapat dalam file **Algorithm.java** dengan nama **m**.

Sejatinya struktur ini menyimpan nilai f terkecil saat suatu kata dihidupkan simpulnya. Hal ini bertujuan agar suatu program dapat mengetahui apabila simpul sudah dikunjungi, dan apakah dengan mengunjunginya lagi akan diperoleh hasil yang lebih optimal. Karena apabila simpul yang sama dihidupkan padahal nilai f nya lebih besar, akan terjadi penelusuran ulang yang percuma. Sedangkan jika nilai f baru lebih kecil dibanding di Hash Map, akan optimal apabila simpul ditelusuri lagi dan nilai f yang disimpan oleh Hash Map untuk kata tersebut harus diperbarui.

3.2.3. Priority Queue

Struktur ini merupakan salah satu jenis queue yang mengurutkan elemen didalamnya berdasarkan suatu prioritas, dimana elemen dengan prioritas tertinggi akan diletakan di posisi paling depan. Seperti sebuah Queue, elemen yang dapat dikeluarkan hanyalah elemen paling depan Priority Queue. Pada program, struktur ini digunakan untuk menyimpan struktur SearchNode, yaitu simpul pada persoalan. Setiap SearchNode akan diurutkan berdasarkan nilai f terkecil, sebagaimana pada ketiga algoritma yang membutuhkan pengurutan simpul sehingga yang dipilih selalu simpul dengan f terkecil. Penggunaan struktur ini terletak pada **Algorithm.java** dengan nama **pq**.

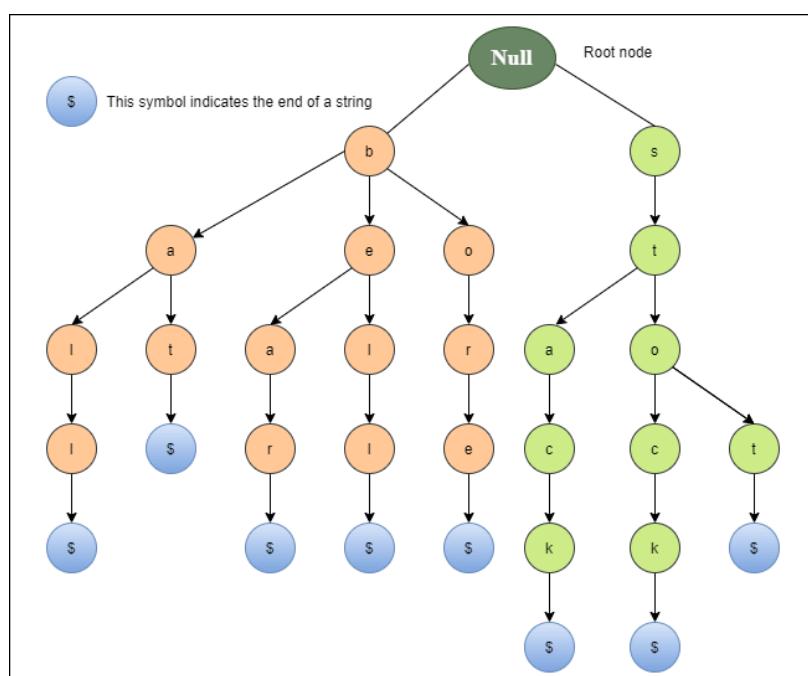
3.2.4. Adjacency List

Struktur data ini merupakan salah satu struktur yang paling sering digunakan pada graf, karena ia mampu merepresentasikan hubungan antar simpul. Struktur ini menyimpan sebuah larik berisi seluruh simpul lainnya yang bertetangga dengan dirinya. Untuk program ini, saya menggunakan sebuah HashMap dengan kata sebagai key dan larik kata sebagai valuenya. Sehingga, struktur ini dapat dengan cepat mengembalikan semua kata valid yang berbeda hanya satu huruf dari suatu kata tertentu. Tujuan dibuatnya struktur ini adalah supaya saat melakukan pencarian, program tidak perlu berulang kali mencari kata apa saja yang dapat ia kunjungi sebelumnya pada setiap simpul ekspan. Adjacency list akan dibuat cukup sekali yaitu saat program dimulai dan terus disimpan dan

digunakan sampai program dihentikan. Implementasi struktur data ini terdapat pada *class Adjacency.java*

3.2.5. Trie

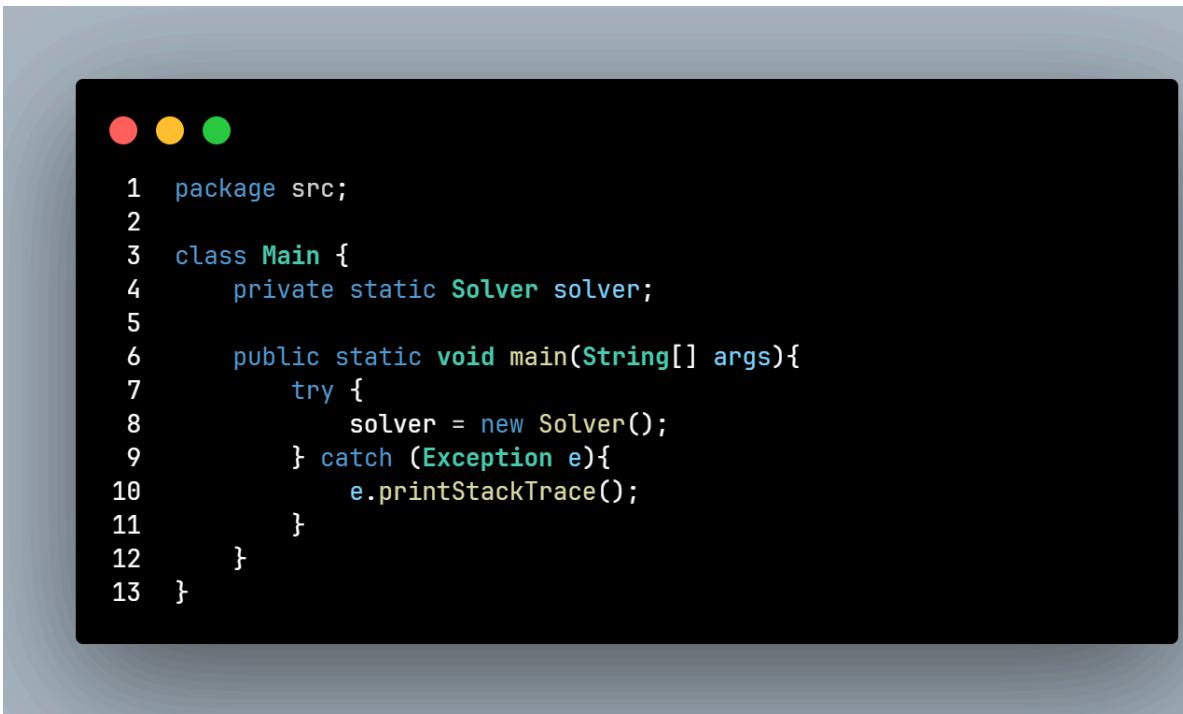
Diambil dari kata *retrieval*, struktur data ini memungkinkan pengembalian suatu kata dari *dictionary* secara efisien. Trie berbentuk struktur tree terurut yang setiap simpulnya menyimpan suatu *character*. Dengan memanfaatkan prefix, setiap kata dalam *dictionary* dapat ditemukan dengan waktu linier terhadap panjang kata, dengan cara melakukan perjalanan pada trie dengan setiap langkahnya menuju pada huruf selanjutnya di kata tersebut. Alasan digunakannya struktur ini adalah untuk mengetahui apakah suatu kata terletak pada *dictionary* yang digunakan secara sangat efisien, karena proses ini sangat dibutuhkan berulang kali selama program berlangsung. Sama seperti adjacency list, struktur ini dibuat cukup sekali yaitu saat program dimulai dan terus disimpan dan digunakan sampai program dihentikan. Implementasi dari simpul trie terletak pada *class TrieNode.java* dan struktur utama trie nya sendiri pada *class Trie.java*.



Gambar 3.1. Illustrasi Struktur Data Trie
(Sumber: <https://www.javatpoint.com/>)

3.3. Source Code

3.3.1. Main.java



```
 1 package src;
 2
 3 class Main {
 4     private static Solver solver;
 5
 6     public static void main(String[] args){
 7         try {
 8             solver = new Solver();
 9         } catch (Exception e){
10             e.printStackTrace();
11         }
12     }
13 }
```

3.3.2. Solver.java

```
import java.io.IOException;
import java.util.ArrayList;

public class Solver {
    private static IO io;
    private static FileReader fr;
    private static Trie tr;
    private static Adjacency adj;
    private ArrayList<String> solution;
    private long calculationDuration;
    private int numVisited;

    public Solver() throws InterruptedException, IOException{
        io = new IO();
        fr = new FileReader();
        fr.readFile();
    }
}
```

```

        tr = new Trie();
        tr.createTrie(fr.getWordList());
        adj = new Adjacency();
        Thread loadingThread = new Thread(() -> {
            try {
                adj.createAdjacencyMap(fr.getWordList(), tr);
            } catch (Exception e) {
                System.err.println("Background function was
interrupted.");
            }
        });
        loadingThread.start();
        io.loadingScreen(loadingThread);
        io.showTitle();
        io.firstGreeting();
        this.startProgram();
    }

    public void startProgram(){
        boolean restart = true;
        while (restart){
            io.askInputs(tr);
            solution = new ArrayList<String>();
            long startTime, endTime;
            if (io.getSelection() == 1){
                Algorithm algorithm = new UCS(io.getOrigin(),
io.getTarget());
                startTime = System.currentTimeMillis();
                algorithm.doAlgorithm(adj);
                endTime = System.currentTimeMillis();
                solution = algorithm.getSolutionPath();
                numVisited = algorithm.getNodeVisited();
                algorithm = null;
            } else if (io.getSelection() == 2){
                Algorithm algorithm = new
GreedyBestFirstSearch(io.getOrigin(), io.getTarget());
                startTime = System.currentTimeMillis();
                algorithm.doAlgorithm(adj);
                endTime = System.currentTimeMillis();
                solution = algorithm.getSolutionPath();
                numVisited = algorithm.getNodeVisited();
                algorithm = null;
            }
        }
    }
}

```

```

        } else {
            Algorithm algorithm = new AStar(io.getOrigin(),
io.getTarget());
            startTime = System.currentTimeMillis();
            algorithm.doAlgorithm(adj);
            endTime = System.currentTimeMillis();
            solution = algorithm.getSolutionPath();
            numVisited = algorithm.getNodeVisited();
            algorithm = null;
        }
        calculationDuration = endTime - startTime;
        io.printSolution(solution);
        io.printSearchData(numVisited, calculationDuration,
solution.size());
        if (io.askToRestart()) {
            io.nextGreeting();
        } else {
            io.closeProgram();
            restart = false;
        }
    }
}
}

```

3.3.3. IO.java

```

package src;

import java.util.*;
import java.io.IOException;

public class IO {

    /* ATTRIBUTES */
    private Scanner scanner = new Scanner(System.in);
    private PrintColor pc;
    private String origin;
    private String target;
    private long randomNumber;
    private Integer algorithmSelection;
    private int threadSleepTime = 200;
}

```

```
/* CONSTRUCTOR METHOD*/
public IO(){
    pc = new PrintColor();
}

/* clearScreen Method */
public static void clearScreen() throws InterruptedException,
IOException {
    String osName =
System.getProperty("os.name").toLowerCase();
    if (osName.contains("windows")) {
        new ProcessBuilder("cmd", "/c",
"cls").inheritIO().start().waitFor();
    } else {
        new
ProcessBuilder("clear").inheritIO().start().waitFor();
    }
}

/* Getter */
public String getOrigin(){
    return origin.toLowerCase();
}

public String getTarget(){
    return target.toLowerCase();
}

public int getSelection(){
    return algorithmSelection;
}

/* PRINT METHODS */
public void loadingScreen(Thread backgroundThread) throws
InterruptedException, IOException{
    pc.PrintBlue();
    clearScreen();
    System.out.print("Program Initializing...\\n" +
"[===== 10%\\n");
    Thread.sleep(threadSleepTime);
    clearScreen();
}
```

```
System.out.print("Program Initializing...\\n" +
    "[====] 20%\\n");
Thread.sleep(threadSleepTime);
clearScreen();
System.out.print("Program Initializing...\\n" +
    "[=====] 30%\\n");
Thread.sleep(threadSleepTime);
clearScreen();
System.out.print("Program Initializing...\\n" +
    "[=====] 40%\\n");
Thread.sleep(threadSleepTime);
clearScreen();
System.out.print("Program Initializing...\\n" +
    "[=====] 50%\\n");
Thread.sleep(threadSleepTime);
clearScreen();
System.out.print("Program Initializing...\\n" +
    "[=====] 60%\\n");
Thread.sleep(threadSleepTime);
clearScreen();
System.out.print("Program Initializing...\\n" +
    "[=====] 70%\\n");
Thread.sleep(threadSleepTime);
clearScreen();
System.out.print("Program Initializing...\\n" +
    "[=====] 80%\\n");
backgroundThread.join();
clearScreen();
System.out.print("Program Initializing...\\n" +
    "[=====] 90%\\n");
Thread.sleep(threadSleepTime);
clearScreen();
System.out.print("Program Initializing...\\n" +
    "[=====] 100%\\n");
Thread.sleep(threadSleepTime);
clearScreen();
pc.PrintReset();
}

public void showTitle(){
    pc.PrintCyan();
    System.out.println();
}
```



```

    pc.PrintReset();
    System.out.println();
    System.out.println(">>> Welcome to Word Ladder Solver");
    System.out.println(">>> This solver will find optimum path
in a word ladder game for any 2 word in the English dictionary");
    System.out.println();
}

public void askInputs(Trie tr){
    randomNumber = (System.currentTimeMillis()/1000) % 4;
    askOrigin();
    while (!tr.search(this.getOrigin())){
        System.out.println(pc.RED + "!!! " + this.origin+
doesn't exists in Dictionary, pick another word!" + pc.DEFAULT);
        pc.PrintGreen();
        System.out.print("??? ");
        this.origin = scanner.nextLine();
        pc.PrintReset();
        this.origin = this.origin.toUpperCase();
        System.out.println();
    }
    askTarget();
    boolean isValid;
    if (!tr.search(this.getTarget())){
        System.out.println(pc.RED + "!!! " + this.target +
doesn't exists in Dictionary, pick another word!" + pc.DEFAULT);
        isValid = false;
    } else if (this.origin.length() != this.target.length()){
        System.out.println(pc.RED + "!!! Word length doesn't
match, pick another ending word!" + pc.DEFAULT);
        isValid = false;
    } else {
        isValid = true;
    }
    while (!isValid){
        pc.PrintGreen();
        System.out.print("??? ");
        this.target = scanner.nextLine();
        pc.PrintReset();
        this.target = this.target.toUpperCase();
    }
}

```

```

        System.out.println();
        if (!tr.search(this.getTarget())){
            System.out.println(pc.RED + "!!! " + this.target +
" doesn't exists in Dictionary, pick another word!" + pc.DEFAULT);
            isValid = false;
        } else if (this.origin.length() != this.target.length()){
            System.out.println(pc.RED + "!!! Word length
doesn't match, pick another ending word!" + pc.DEFAULT);
            isValid = false;
        } else {
            isValid = true;
        }
    }

    askSelection();
}

public void askOrigin(){
    if (this.randomNumber == 0){
        System.out.println(">>> What would the starting word
be: ");
    } else if (this.randomNumber == 1){
        System.out.println(">>> Please input the origin word:
");
    } else if (this.randomNumber == 2){
        System.out.println(">>> Where would we start the
search from: ");
    } else {
        System.out.println(">>> Enter a word to begin our
search: ");
    }
    pc.PrintGreen();
    System.out.print("??? ");
    this.origin = scanner.nextLine();
    pc.PrintReset();
    this.origin = this.origin.toUpperCase();
    System.out.println();
}

public void askTarget(){
    if (this.randomNumber == 0){

```

```

        System.out.println(">>> Good!, " + pc.GREEN +
this.origin + pc.DEFAULT + " is the starting word");
        System.out.println(">>> Now what would the ending word
be: ");
    } else if (this.randomNumber == 1){
        System.out.println(">>> Sublime!, " + pc.GREEN +
this.origin + pc.DEFAULT + " is the origin word");
        System.out.println(">>> Next, Please input the ending
word: ");
    } else if (this.randomNumber == 2){
        System.out.println(">>> Our search will start from " +
pc.GREEN + this.origin + pc.DEFAULT);
        System.out.println(">>> Where will we end our search:
");
    } else {
        System.out.println(">>> " + pc.GREEN + this.origin +
pc.DEFAULT + " is entered, excellent!");
        System.out.println(">>> Now enter the word to end
with: ");
    }
    pc.PrintGreen();
    System.out.print("??? ");
    this.target = scanner.nextLine();
    pc.PrintReset();
    this.target = this.target.toUpperCase();
    System.out.println();
}

public void askSelection(){
    if (this.randomNumber == 0){
        System.out.println(">>> Great!, " + pc.GREEN +
this.target + pc.DEFAULT + " is the end");
        System.out.println();
        System.out.println(">>> Now what algorithm would be
used: ");
    } else if (this.randomNumber == 1){
        System.out.println(">>> Superb!, " + pc.GREEN +
this.target + pc.DEFAULT + " is the ending word");
        System.out.println();
        System.out.println(">>> Please choose an algorithm to
be used: ");
    } else if (this.randomNumber == 2){

```

```

        System.out.println(">>> Our search will end at " +
pc.GREEN + " " + this.target + pc.DEFAULT);
        System.out.println();
        System.out.println(">>> Before we begin our search,
pick one of the following algoritm:");
    } else {
        System.out.println(">>> " + pc.GREEN + " " + this.target + pc.DEFAULT + " is sucessfully entered");
        System.out.println();
        System.out.println(">> Next, an algorithm for the
searching process:");
    }
    pc.PrintPurple();
    System.out.println("1. Uniform Cost Search");
    System.out.println("2. Greedy Best First Search");
    System.out.println("3. A*");
    pc.PrintReset();
    System.out.println();
    if (this.randomNumber == 0){
        System.out.println(">>> What algorithm number would be
used: ");
    } else if (this.randomNumber == 1){
        System.out.println(">>> Please input the chosen
algorithm number: ");
    } else if (this.randomNumber == 2){
        System.out.println(">>> Pick a number according to the
algorithm: ");
    } else {
        System.out.println(">>> Enter an algorithm number: ");
    }
    pc.PrintGreen();
    System.out.print("??? ");
    this.algorithmSelection =
Integer.parseInt(scanner.nextLine());
    pc.PrintReset();
    System.out.println();
    while (this.algorithmSelection < 0 ||
this.algorithmSelection > 3){
        System.out.println(pc.RED + "!!! Algorithms selection
is Invalid, Pick a number between 1 and 3" + pc.DEFAULT);
        pc.PrintGreen();
        System.out.print("??? ");
    }
}

```

```

        this.algorithmSelection =
Integer.parseInt(scanner.nextLine());
        pc.PrintReset();
        System.out.println();
    }

    String[] algoList = {"Uniform Cost Search", "Greedy Best
First Search", "A*"};

    if (this.randomNumber == 0){
        System.out.println(">>> " + pc.PURPLE +
algoList[this.algorithmSelection-1] + pc.DEFAULT + " would be
used");

        System.out.println(">>> Fantastic!, we can finally
start the search");
    } else if (this.randomNumber == 1){
        System.out.println(">>> " + pc.PURPLE +
algoList[this.algorithmSelection-1] + pc.DEFAULT + " is the
chooseen Algorithm");

        System.out.println(">>> Please be patient while the
solver is starting");
    } else if (this.randomNumber == 2){
        System.out.println(">>> You selected " + pc.PURPLE +
algoList[this.algorithmSelection-1] + pc.DEFAULT);

        System.out.println(">>> Marvelous!, let's start our
search for a solution");
    } else {
        System.out.println(">>> " + pc.PURPLE +
algoList[this.algorithmSelection-1] + pc.DEFAULT + " has been
entered as the search Algorithm");

        System.out.println(">>> We can finally begin!");
    }
    System.out.println();
}

public void printSolution(ArrayList<String> sol){
    if (sol.isEmpty()){
        pc.PrintYellow();
        System.out.println(">>> There is no solution found for
this word ladder!");

        pc.PrintReset();
    } else {
        System.out.println(">>> Building your Ladder!");
        System.out.println();
    }
}

```



```

public boolean askToRestart(){
    String input;
    System.out.println(">>> Do you want to try another word
ladder " + pc.BRIGHTBLUE + "(Yes/No) " + pc.DEFAULT);
    pc.PrintGreen();
    System.out.print("??? ");
    input = scanner.nextLine();
    pc.PrintReset();
    System.out.println();
    if (input.equals("YES") || input.equals("Yes") ||
input.equals("yes") || input.equals("Y") || input.equals("y")){
        return true;
    } else {
        return false;
    }
}

public void closeProgram(){
    pc.PrintBlue();
    System.out.println("Thank you for using Word Ladder
Solver, see you next time!");
    System.out.println("Closing Program...");
    pc.PrintReset();
}
}

```

3.4. Coordinate.java

```

package src;

public class Coordinate implements Comparable<Coordinate> {
    int col;
    int row;

    Coordinate(int c, int r){
        this.col = c;
        this.row = r;
    }
}

```

```
@Override
public int compareTo(Coordinate other) {
    int result = Integer.compare(this.col, other.col);
    if (result != 0) {
        return result;
    }
    return Integer.compare(this.row, other.row);
}
```

3.3.4. FileReader.java

```
 1 package src;
 2
 3 import java.io.*;
 4 import java.util.*;
 5
 6 public class FileReader {
 7     private Scanner sc;
 8     private String[] wordList;
 9     private int wordCount = 80368;
10
11     public FileReader(){
12         wordList = new String[wordCount];
13         readFile();
14     }
15
16     public String[] getWordList(){
17         return wordList;
18     }
19
20     public void readFile(){
21         int i = 0;
22         String word;
23         try{
24             sc = new Scanner(new File("res/words.txt"));
25             while (sc.hasNextLine()){
26                 word = sc.nextLine();
27                 wordList[i] = word;
28                 i++;
29             }
30             sc.close();
31         } catch (IOException e){
32             e.printStackTrace();
33         }
34     }
35 }
36
```

3.3.6. TrieNode.java

```
● ● ●  
1 package src;  
2  
3 public class TrieNode {  
4     TrieNode[] child;  
5     boolean isEnd;  
6  
7     public TrieNode() {  
8         child = new TrieNode[26];  
9         isEnd = false;  
10    }  
11 }
```

3.3.7. Trie.java

```
package src;  
  
public class Trie {  
  
    /* Attributes */  
    private TrieNode root;  
  
    /* Constructor */  
    public Trie() {  
        root = new TrieNode();  
    }  
  
    /* Methods */  
    public void insert(String word) {  
        TrieNode curr = root;  
        char[] charArray = word.toCharArray();  
        for (char ch : charArray) {  
            int i = ch - 'a';  
            if (curr.child[i] == null) {
```

```
        curr.child[i] = new TrieNode();
    }
    curr = curr.child[i];
}
curr.isEnd = true;
}

public boolean search(String word) {
    TrieNode curr = root;
    char[] charArray = word.toCharArray();
    for (char ch : charArray) {
        int i = ch - 'a';
        if (curr.child[i] == null) {
            return false;
        }
        curr = curr.child[i];
    }
    return curr.isEnd;
}

public void createTrie(String[] arr){
    for (int i=0; i<arr.length; i++){
        this.insert(arr[i]);
    }
}
}
```

3.3.8. Adjacency.java

```
1 package src;
2
3 import java.util.*;
4
5 public class Adjacency {
6     private HashMap<String, String[]> map;
7
8     public Adjacency(){
9         map = new HashMap<String, String[]>();
10    }
11
12    public String[] getAdjacency(String node){
13        return map.get(node);
14    }
15
16    public void createAdjacencyMap(String[] arr, Trie tr){
17        String newWord;
18        String[] tempArr = new String[53]; //Placeholder
19        String[] putArr;
20        int count;
21        for (String word : arr){
22            char[] charArray = word.toCharArray();
23            count = 0;
24            for (int i=0; i<charArray.length; i++){
25                for (int j=0; j<25; j++){
26                    charArray[i] = (charArray[i] == 'z') ? 'a' : (char)(charArray[i]+1);
27                    newWord = String.valueOf(charArray);
28                    if (tr.search(newWord)){
29                        tempArr[count] = newWord;
30                        count++;
31                    }
32                }
33                charArray[i] = (charArray[i] == 'z') ? 'a' : (char)(charArray[i]+1);
34            }
35            putArr = new String[count];
36            System.arraycopy(tempArr, 0, putArr, 0, count);
37            map.put(word, putArr);
38        }
39    }
40}
41
```

3.3.9. PrintColor.java

```
package src;

public class PrintColor {
    public final String DEFAULT = "\u001B[39m";
    public final String RED = "\u001B[31m";
    public final String GREEN = "\u001B[32m";
```

```

public final String YELLOW = "\u001B[33m";
public final String BRIGHTBLUE = "\u001B[34m";
public final String PURPLE = "\u001B[35m";
public final String CYAN = "\u001B[36m";

public void PrintReset(){
    System.out.print(DEFAULT);
}

public void PrintYellow(){
    System.out.print(YELLOW);
}

public void PrintCyan(){
    System.out.print(CYAN);
}

public void PrintGreen(){
    System.out.print(GREEN);
}

public void PrintBlue(){
    System.out.print(BRIGHTBLUE);
}

public void PrintPurple(){
    System.out.print(PURPLE);
}

public void PrintRed(){
    System.out.print(RED);
}
}

```

3.3.10. Algorithm.java

```

package src;

import java.util.*;

abstract class Algorithm {
    private PriorityQueue<SearchNode> pq;
}

```

```

private Map<String, Integer> m;
private ArrayList<String> solutionPath;
private String start;
private String end;
private int nodeVisited;

public Algorithm(String _start, String _end){
    pq = new PriorityQueue<SearchNode>();
    m = new HashMap<String, Integer>(); // <id, f>
    solutionPath = new ArrayList<>();
    start = _start;
    end = _end;
}

public ArrayList<String> getSolutionPath(){
    return solutionPath;
}

public int getNodeVisited(){
    return nodeVisited;
}

public int hammingDistanceToEnd(String x){
    int res = 0;
    for (int i=0; i<x.length(); i++){
        if (x.charAt(i) != end.charAt(i)){
            res++;
        }
    }
    return res;
}

public abstract int calculateG(SearchNode n);

public abstract int calculateH(SearchNode n);

public void doAlgorithm(Adjacency adj){
    SearchNode startNode = new SearchNode(start, null);
    startNode.setG(0);
    startNode.setH(calculateH(startNode));
    pq.add(startNode);
}

```

```

m.put(start, startNode.getF());

String[] arr = new String[53];
SearchNode curr = new SearchNode();
SearchNode neighborNode = new SearchNode();
Integer tempInteger;
boolean found = false;
nodeVisited = 0;

while (!pq.isEmpty() && !found) {
    curr = pq.poll();
    nodeVisited++;
    arr = adj.getAdjacency(curr.getID());
    for (String neighbor : arr){
        neighborNode = new SearchNode(neighbor, curr);
        neighborNode.setG(calculateG(neighborNode));
        neighborNode.setH(calculateH(neighborNode));
        if (neighbor.equals(end)){
            solutionPath =
constructSolution(neighborNode);
            found = true;
            break;
        }
        if (m.containsKey(neighbor)){
            tempInteger =
Integer.valueOf(neighborNode.getF());
            if (m.get(neighbor) > tempInteger){
                m.put(neighbor, tempInteger);
                pq.add(neighborNode);
            }
        } else {
            m.put(neighbor, neighborNode.getF());
            pq.add(neighborNode);
        }
    }
}

public ArrayList<String> constructSolution(SearchNode end){
    ArrayList<String> sol = new ArrayList<String>();
    SearchNode curr = end;
    while (curr != null){

```

```
        sol.add(curr.getID().toUpperCase());
        curr = curr.getParent();
    }
    Collections.reverse(sol);
    return sol;
}
}
```

3.3.11. UCS.java

```
public class UCS extends Algorithm{

    UCS(String _start, String _end) {
        super(_start, _end);
    }

    @Override
    public int calculateG(SearchNode n) {
        return n.getParent().getG()+1;
    }

    @Override
    public int calculateH(SearchNode n) {
        return 0;
    }
}
```

3.3.12. GreedyBestFirstSearch.java

```
● ● ●  
1 package src;  
2  
3 public class GreedyBestFirstSearch extends Algorithm{  
4  
5     GreedyBestFirstSearch(String _start, String _end){  
6         super(_start, _end);  
7     }  
8  
9     @Override  
10    public int calculateG(SearchNode n){  
11        return 0;  
12    }  
13  
14    @Override  
15    public int calculateH(SearchNode n){  
16        return hammingDistanceToEnd(n.getID());  
17    }  
18}
```

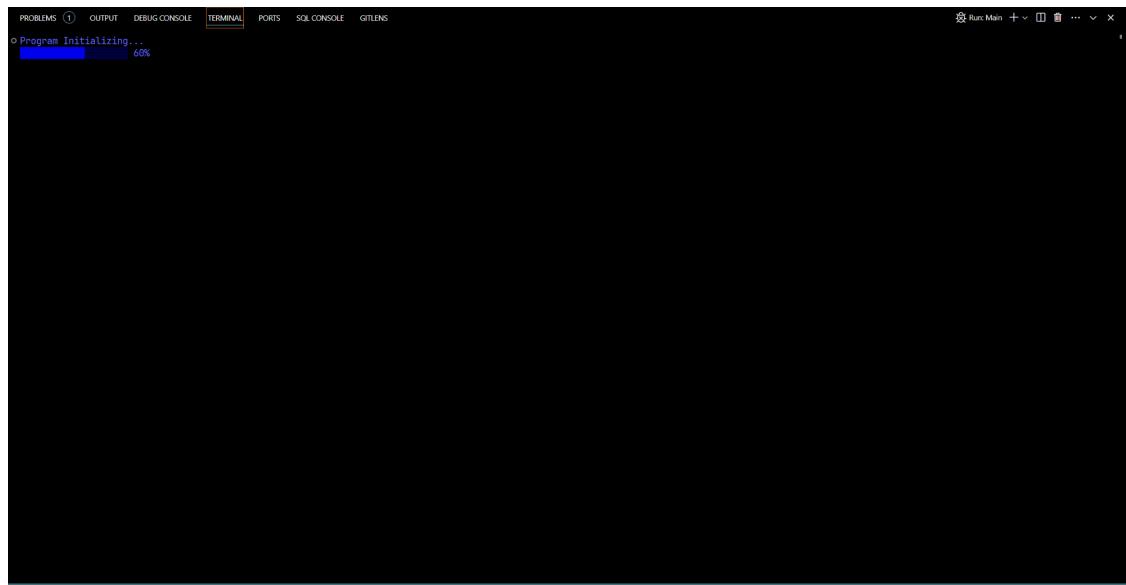
3.3.13 AStar.java

```
1 package src;
2
3 public class AStar extends Algorithm{
4
5     AStar(String _start, String _end){
6         super(_start, _end);
7     }
8
9     @Override
10    public int calculateG(SearchNode n){
11        return n.getParent().getG()+1;
12    }
13
14    @Override
15    public int calculateH(SearchNode n){
16        return hammingDistanceToEnd(n.getID());
17    }
18 }
```

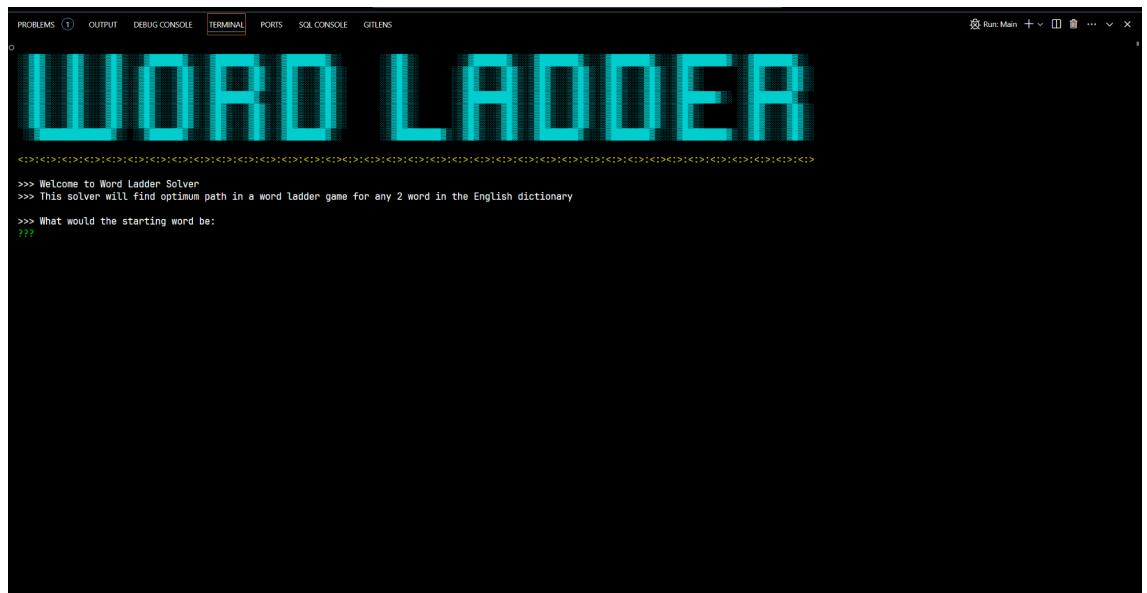
Pengujian

4.1. Tampilan Awal

4.1.1. Loading Screen



4.1.2. Judul Program



4.2. Test Case

4.2.1. POOL → MILK

4.2.1.1. UCS

4.2.1.2. Greedy BFS

4.2.1.3. A*

```
>>> Welcome back!, what Word Ladder game would you try this time

>>> Where would we start the search from:
?? pool

>>> Our search will start from POOL
>>> Where will we end our search:
?? milk

>>> Our search will end at MILK

>>> Before we begin our search, pick one of the following algoritm:
1. Uniform Cost Search
2. Greedy Best First Search
3. A*

>>> Pick a number according to the algorithm:
?? 3

>>> You selected A*
>>> Marvelous!, let's start our search for a solution

>>> Building your Ladder!
[[ P O O L
  M O O L
  M O L L
  M I L L
  M I L K
  ]]

>>> Number of nodes visited are 6 Nodes
>>> The path found is 5 words long
>>> Time taken to find solution is 0 miliseconds
```

4.2.2. CHANGE → COMEDO

4.2.2.1. UCS

```
>>> Where will we end our search:  
??? comedo  
  
>>> Our search will end at COMEDO  
  
>>> Before we begin our search, pick one of the following algorithm:  
1. Uniform Cost Search  
2. Greedy Best First Search  
3. A*  
  
>>> Pick a number according to the algorithm:  
??? 1  
  
>>> You selected Uniform Cost Search  
>>> Marvelous!, let's start our search for a solution  
  
>>> Building your Ladder!  
  


|  |             |
|--|-------------|
|  | C H A N G E |
|  | C H A N G S |
|  | C H A N T S |
|  | C H I N T S |
|  | C H I N E S |
|  | C H I N E D |
|  | C O I N E D |
|  | C O N N E D |
|  | C O N N E R |
|  | C O N G E R |
|  | C O N G E S |
|  | C O N I E S |
|  | C O N I N S |
|  | C O N I N G |
|  | H O N I N G |
|  | H O M I N G |
|  | H O M I N Y |
|  | H O M I L Y |
|  | H O M E L Y |
|  | C O M E L Y |
|  | C O M E D Y |
|  | C O M E D O |

  
>>> Number of nodes visited are 8274 Nodes  
>>> The path found is 22 words long  
>>> Time taken to find solution is 19 miliseconds
```

4.2.2.2. Greedy BFS

```
>>> Fantastic!, we can finally start the search
>>> Building your Ladder!
[[[ C H A N G E
    C H A N G S
    C H A N T S
    C H I N T S
    C H I N E S
    C H I N E D
    C O I N E D
    C O N N E D
    C O N K E D
    C O O K E D
    C O O E E D
    C O O E E S
    C O O E R S
    C O M E R S
    C O M E T S
    C O M P T S
    C O M P O S
    C O M B O S
    C O M B E S
    C O M B E D
    C O M P E D
    C O P P E D
    C O P P E R
    C O P I E R
    C O P I E S
    C O L I E S
    C O L I N S
    C O N I N S
    C O N I N G
    C O M I N G
    H O M I N G
    H O M I N Y
    H O M I L Y
    H O M E L Y
    C O M E L Y
    C O M E D Y
    C O M E D O

>>> Number of nodes visited are 337 Nodes
>>> The path found is 37 words long
>>> Time taken to find solution is 1 miliseconds
```

4.2.2.3. A*

```
>>> Good!, CHANGE is the starting word
>>> Now what would the ending word be:
???. comando

>>> Great!, comando is the end

>>> Now what algorithm would be used:
1. Uniform Cost Search
2. Greedy Best First Search
3. A*

>>> What algorithm number would be used:
???. 3

>>> A* would be used
>>> Fantastic!, we can finally start the search

>>> Building your Ladder!


C H A N G E  
C H A N G S  
C H A N T S  
C H I N T S  
C H I N E S  
C H I N E D  
C O I N E D  
C O N N E D  
C O N N E R  
C O N G E R  
C O N G E S  
C O N I E S  
C O N I N S  
C O N I N G  
C O M I N G  
H O M I N G  
H O M I N Y  
H O M I L Y  
H O M E L Y  
C O M E L Y  
C O M E D Y  
C O M E D O


```

4.2.3. MOTION → ACTION

4.2.3.1. UCS

4.2.3.2. Greedy BFS

```
>>> Welcome back!, what Word Ladder game would you try this time

>>> Enter a word to begin our search:
??? motion

>>> MOTION is entered, excellent!
>>> Now enter the word to end with:
??? action

>>> ACTION is sucessfully entered

>>> Next, an algorithm for the searching process:
1. Uniform Cost Search
2. Greedy Best First Search
3. A*

>>> Enter an algorithm number:
??? 2

>>> Greedy Best First Search has been entered as the search Algorithm
>>> We can finally begin!

>>> There is no solution found for this word ladder!

>>> Number of nodes visited are 8403 Nodes
>>> The path found is 0 words long
>>> Time taken to find solution is 13 miliseconds
```

4.2.3.3. A*

4.2.4. GIMLETS → TREEING

4.2.4.1. UCS

```
    G I M L E T S
    G I B L E T S
    R I B L E T S
    R I L L E T S
    B I L L E T S
    B I L L E R S
    B A I L E R S
    B A I L E E S
    B A I L I E S
    B A L L I E S
    B U L L I E S
    C U L L I E S
    C O L L I E S
    C O D L I N S
    C O O D I N G
    C O O D I N G
    G O O D I N G
    G O A D I N G
    G R A D I N G
    G R A Y I N G
    G R E Y I N G
    G R E E I N G
    T R E E I N G

>>> Number of nodes visited are 5794 Nodes
>>> The path found is 25 words long
>>> Time taken to find solution is 15 miliseconds
```

4.2.4.2. Greedy BFS

```
>>> Building your Ladder!
```

	GIMLETS
	GIBLETS
	GIBBETS
	GIBBERS
	JIBBERS
	JOBBERS
	COBBERS
	COMBERS
	CAMBERS
	CAMPERS
	TAMPERS
	TAPPERS
	TOPPERS
	LOPPERS
	LOOPERS
	LOOTERS
	TOOTERS
	TOTTERS
	TETTERS
	TEETERS
	TEENERS
	TENNERS
	TENDERS
	TEDDERS
	WEDDERS
	WADDERS
	WARDERS
	WARDENS
	WARRENS
	BARRENS
	BARRELS
	PARRELS
	PARCELS
	CARCELS
	CARPELS
	CARPETS
	CARNETS
	CORNETS
	CORNERS
	CORKERS
	CONKERS
	CONFERS
	COFFERS
	COFFEES
	TOFFEES
	TOFFIES
	TAFFIES
	BAFFIES
	BIFFIES
	BIFFINS
	TIFFINS
	TIFFING
	RIFFING

```
R E F F I N G
R E E F I N G
R E E K I N G
S E E K I N G
S E E M I N G
T E E M I N G
T E A M I N G
T E A S I N G
T E N S I N G
T E N T I N G
T E S T I N G
T A S T I N G
T A T T I N G
T O T T I N G
D O T T I N G
D O A T I N G
C O A T I N G
C R A T I N G
G R A T I N G
G R A C I N G
T R A C I N G
T R U C I N G
T R U E I N G
T R E E I N G

>>> Number of nodes visited are 354 Nodes
>>> The path found is 77 words long
>>> Time taken to find solution is 2 miliseconds
```

4.2.4.3. A*

```
>>> Building your Ladder!

G I M L E T S
G I G L E T S
W I G L E T S
W I L L E T S
W I L L E R S
T I L L E R S
T I L T E R S
T I T T E R S
T A T T E R S
P A T T E R S
P A T T E N S
L A T T E N S
L A T T I N S
M A T T I N S
M A T T I N G
M A L T I N G
M O L T I N G
M O A T I N G
C O A T I N G
C R A T I N G
P R A T I N G
P R A Y I N G
P R E Y I N G
P R E E I N G
T R E E I N G

>>> Number of nodes visited are 4193 Nodes
>>> The path found is 25 words long
>>> Time taken to find solution is 9 miliseconds
```

4.2.5. KISS → BABY

4.2.5.1. UCS

```
W W W U L H U U E R

>>> Welcome to Word Ladder Solver
>>> This solver will find optimum path in a word ladder game for any 2 word in the English dictionary

>>> Enter a word to begin our search:
???
>>> kiss

>>> KISS is entered, excellent!
>>> Now enter the word to end with:
???
>>> baby

>>> BABY is sucessfully entered

>>> Next, an algorithm for the searching process:
1. Uniform Cost Search
2. Greedy Best First Search
3. A*
>>> Enter an algorithm number:
???
>>> 1

>>> Uniform Cost Search has been entered as the search Algorithm
>>> We can finally begin!

>>> Building your Ladder!
[[ K I S S
  K I N S
  K I N E
  K A N E
  B A N E
  B A B E
  B A B Y ]]

>>> Number of nodes visited are 2466 Nodes
>>> The path found is 7 words long
>>> Time taken to find solution is 27 miliseconds
```

4.2.5.2. Greedy BFS

```
>>> Welcome back!, what Word Ladder game would you try this time

>>> Enter a word to begin our search:
???
>>> kiss

>>> KISS is entered, excellent!
>>> Now enter the word to end with:
???
>>> baby

>>> BABY is sucessfully entered

>>> Next, an algorithm for the searching process:
1. Uniform Cost Search
2. Greedy Best First Search
3. A*
>>> Enter an algorithm number:
???
>>> 2

>>> Greedy Best First Search has been entered as the search Algorithm
>>> We can finally begin!

>>> Building your Ladder!
[[ K I S S
  M I S S
  M A S S
  B A S S
  B A T S
  B A T E
  B A B E
  B A B Y ]]

>>> Number of nodes visited are 10 Nodes
>>> The path found is 8 words long
>>> Time taken to find solution is 0 miliseconds
```

4.2.5.3. A*

4.2.6. START → ENDOW

4.2.6.1. UCS

4.2.6.2. Greedy BFS

```
>>> Welcome back!, what Word Ladder game would you try this time  
  
>>> Enter a word to begin our search:  
???
```

start

```
>>> START is entered, excellent!  
>>> Now enter the word to end with:  
???
```

endow

```
>>> ENDOW is sucessfully entered  
  
>> Next, an algorithm for the searching process:  
1. Uniform Cost Search  
2. Greedy Best First Search  
3. A*
```

A

```
>>> Enter an algorithm number:  
???
```

2

```
>>> Greedy Best First Search has been entered as the search Algorithm  
>>> We can finally begin!
```

```
>>> There is no solution found for this word ladder!
```

```
>>> Number of nodes visited are 7396 Nodes  
>>> The path found is 0 words long  
>>> Time taken to find solution is 79 miliseconds
```

4.2.6.3. A*

```
WORD LADDER

>>> Welcome to Word Ladder Solver
>>> This solver will find optimum path in a word ladder game for any 2 word in the English dictionary

>>> Enter a word to begin our search:
?? start

>>> START is entered, excellent!
>>> Now enter the word to end with:
?? endow

>>> ENDOW is sucessfully entered

>>> Next, an algorithm for the searching process:
1. Uniform Cost Search
2. Greedy Best First Search
3. A*

>>> Enter an algorithm number:
?? 3

>>> A* has been entered as the search Algorithm
>>> We can finally begin!

>>> There is no solution found for this word ladder!

>>> Number of nodes visited are 7520 Nodes
>>> The path found is 0 words long
>>> Time taken to find solution is 72 miliseconds
```

Analisis Solusi

Ketiga algoritma yang digunakan, yaitu Uniform Cost Search, Greedy Best First Search, dan A*, menghasilkan hasil pencarian yang berbeda. Mulai dari rute solusi, optimalitas, durasi eksekusi, serta efektivitas memori. Pada bab ini, setiap aspek tersebut akan dianalisis bagi setiap algoritma.

5.1. Optimalitas

Untuk permasalahan optimalitas solusi, tersedia dibawah ini berupa tabel panjang rute yang dihasilkan masing-masing algoritma untuk setiap test case.

Test Case	Panjang Solusi		
	Uniform Cost Search	Greedy Best First Search	A*
1	5	5	5
2	22	37	22
3	0	0	0
4	25	77	25
5	7	8	7
6	0	0	0

Perhatikan bahwa untuk semua testcase, Uniform Cost Search serta A* selalu memberikan solusi dengan panjang kata yang sama. Hal ini diakibatkan karena kedua algoritma tersebut memang sudah optimal. Tetapi, perhatikan bahwa untuk Greedy Best First Search, selain pada testcase pertama ia selalu memberikan solusi yang lebih panjang dibanding kedua testcase lainnya. Hal ini diakibatkan karena memang Greedy BFS bukanlah solusi yang selalu optimal. Penyebab utama terjadinya hal ini adalah karena Greedy BFS sudah terjebak pada minimum lokal, sehingga ia tidak mencoba memperhatikan cabang lainnya yang mungkin

berakhir lebih optimal. Bisa disimpulkan bahwa hanya menggunakan fungsi heuristik seperti pada algoritma Greedy BFS tidak bisa menjamin solusi yang paling optimal.

5.2. Waktu Eksekusi

Amati tabel yang menggambarkan waktu eksekusi masing-masing algoritma dibawah ini

Test Case	Waktu Pencarian (millisecond)		
	Uniform Cost Search	Greedy Best First Search	A*
1	2	1	0
2	19	1	10
3	19	13	9
4	15	2	9
5	27	0	4
6	68	79	72
Rata-Rata	25	16	17,33

Secara rata-rata, dapat dilihat bahwa Uniform Cost Search adalah algoritma paling lambat dengan perbedaan yang cukup besar. Hal ini sesuai dengan teori yang sudah dibahas pada bagian Analisis Algoritma. Sedangkan untuk Greedy BFS, ia memiliki rata-rata kecepatan tertinggi walaupun tidak optimal.

Penyebabnya adalah apabila Greedy BFS sudah memilih solusi optimum lokal, ia akan terus mendekati optimum lokal tersebut tanpa memperhatikan solusi lainnya, yang akhirnya mempercepat jalannya algoritma. Akibatnya, bisa disimpulkan bahwa penggunaan heuristik yang tepat dapat mempercepat algoritma Route Planning.

5.3. Efektivitas Memori

Diberikan tabel dibawah ini yang menggambarkan aproksimasi penggunaan memori masing-masing algoritma untuk setiap test case.

Test Case	Estimasi Penggunaan Memori (Byte)		
	Uniform Cost Search	Greedy Best First Search	A*
1	845040	16384	24576
2	2097152	1050624	1046528
3	1046528	2097152	2097152
4	1240688	234048	1240688
5	1251232	671904	711584
6	2391376	2097152	1425248
Rata-Rata	1478669	1027877	1090962

Sebelumnya perlu disadari bahwa data diatas adalah hasil aproksimasi menggunakan perbedaan penggunaan memori Java Virtual Machine (JVM). Data memori tersebut tidak akurat namun cukup baik dalam menggambarkan perbandingan memori masing-masing algoritma. Perlu diingat pula bahwa Java juga memiliki *garbage collector*, akibatnya pengecekan memori sangat tidak konsisten dan tidak akurat. Kalkulasi dibantu dengan menggunakan class bawaan java yang tersedia dalam package java.util. Tepatnya menggunakan method totalMemory() dan freeMemory() dari class Runtime.

Secara rata-rata, jelas bahwa UCS memiliki penggunaan memori yang terbesar. Hal ini diakibatkan karena banyaknya simpul yang perlu dikunjungi UCS memang lebih banyak dibandingkan 2 algoritma lainnya. Perlu diingat bahwa implementasi ketiga algoritma sangat serupa dan menggunakan priority queue, akibatnya perbedaan penggunaan memori ketiga algoritma pun tidak jauh berbeda. Namun, bisa disimpulkan bahwa besarnya penggunaan memori tetap berbanding lurus dengan banyaknya simpul yang dikunjungi oleh sebuah algoritma.

Penutup

6.1. Kesimpulan

Pada tugas kecil 3 IF2211 Strategi Algoritma Semester 2 Tahun Ajaran 2023/2024 ini, Saya membuat sebuah program Java yang dapat menyelesaikan permainan *Word Ladder*. Permainan diselesaikan dengan memanfaatkan 3 buah algoritma yaitu Uniform Cost Search, Greedy Best First Search, dan A*. Saya merasa belajar cukup banyak selama penyelesaian tugas ini, termasuk mendalami pembuatan program *CLI (command line interface)* dengan baik serta penggunaan paradigma *OOP* dalam bahasa Java.

Tugas ini juga sangat membantu saya dalam memahami lebih algoritma Route Planning yang digunakan, dan saya merasa telah berhasil mengimplementasikan dengan baik algoritma tersebut sebagai sebuah program yang efektif dan efisien. Saya telah memperoleh kesimpulan bahwa hasil performa ketiga algoritma sudah sesuai dengan teori yang dibahas pada bab Analisis Algoritma.

6.2. Saran

Saran pada saya sendiri antara lain:

1. Sebaiknya jangan memaksa menggunakan struktur data kompleks yang pada kenyataanya tidak meningkatkan efisiensi secara signifikan
2. Lebih memprioritaskan spesifikasi program terlebih dahulu dibanding menghabiskan terlalu banyak waktu pada tampilan program
3. Mempertimbangkan untuk mengimplementasikan bonus GUI dibandingkan menggunakan tampilan CLI yang terlalu kaku
4. Lebih memperhatikan diskusi pada Q&A agar tidak terlewat informasi penting

6.3. Lampiran

1. Repository Program: https://github.com/Farhannr28/Tucil3_13522037
2. Tabel Checklist

Poin	Ya	Tidak
1. Program berhasil dijalankan.	✓	

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

6.4. Daftar Pustaka

- Rinaldi Munir. Penentuan Rute (Bagian 1). Diakses pada 1 Mei 2024 dari <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf>
- Rinaldi Munir. Penentuan Rute (Bagian 2). Diakses pada 1 Mei 2024 dari <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf>
- Geeksforgeeks. Uniform-Cost Search (Dijkstra for large Graphs). Diakses pada 3 Mei 2024 dari <https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/>
- Geeksforgeeks. Greedy Best First Search Algorithm. Diakses pada 3 Mei 2024 dari <https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/>
- Geeksforgeeks. A* Search Algorithm. Diakses pada 3 Mei 2024 dari <https://www.geeksforgeeks.org/a-search-algorithm/>
- Geeksforgeeks. Trie Insert and Search. Diakses pada 2 Mei 2024 dari <https://www.geeksforgeeks.org/trie-insert-and-search/>