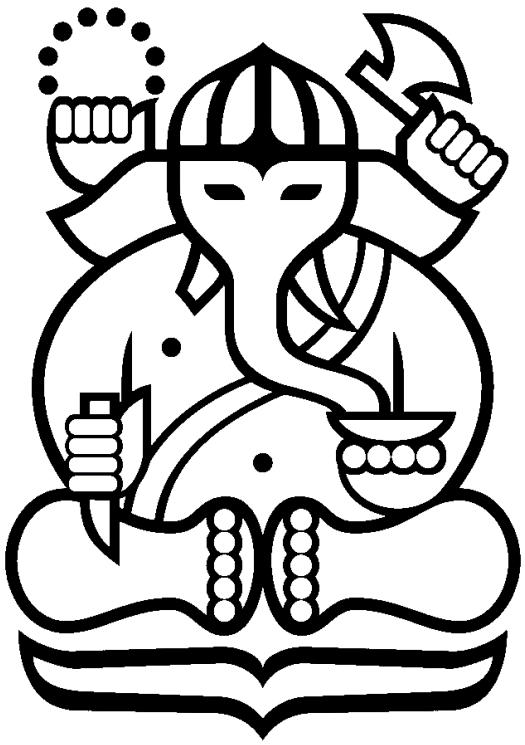
LAPORAN TUGAS KECIL III
IF2211 STRATEGI ALGORITMA

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



Disusun Oleh:

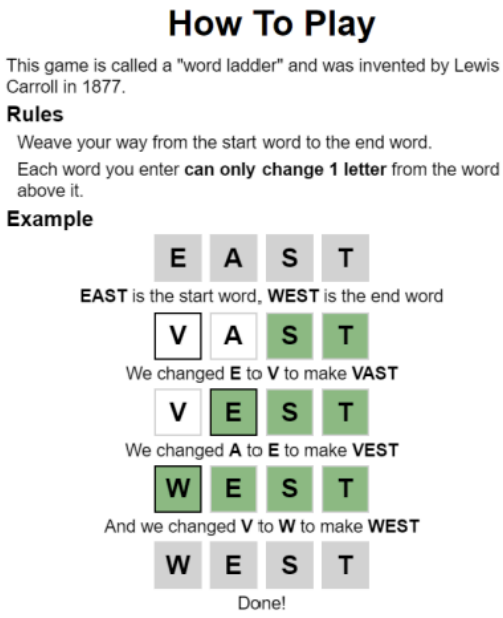
Zachary Samuel Tobing / 13522016

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

2024

Bab I
Latar Belakang

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



Gambar 1. Contoh Permainan
(Sumber: https://wordwormdormdork.com/)

Pada tugas ini, diminta membuat algoritma membuat program yang dapat menentukan solusi permainan ***word ladder*** dengan menggunakan algoritma **UCS**, **Greedy Best First Search**, dan **A***.

Bab II
Algoritma

2.1 UCS (Uniform Cost Search)

Algoritma Uniform Cost Search (UCS) adalah algoritma pencarian graf yang digunakan untuk menemukan jalur terpendek dalam graf berbobot. Algoritma ini mirip dengan algoritma Dijkstra, tetapi alih-alih menggunakan antrian prioritas berdasarkan total biaya untuk mencapai suatu simpul, UCS menggunakan antrian prioritas berdasarkan biaya aktual untuk mencapai setiap simpul dari simpul awal.

Berikut cara kerjanya:

1. Mulai dengan simpul awal dan antrian prioritas kosong.
2. Tambahkan simpul awal ke antrian prioritas dengan biaya 0.
3. Selama antrian prioritas tidak kosong, lakukan hal berikut:
    - Hapus simpul dengan biaya terendah dari antrian prioritas.
    - Jika simpul ini adalah simpul tujuan, maka jalur terpendek telah ditemukan.
    - Jika tidak, perluas simpul (ekspansi) dengan mempertimbangkan semua tetangganya menggunakan fungsi evaluasi g(n).
    - Fungsi evaluasi g(n) ini menghitung biaya untuk mencapai tetangga tersebut dari simpul awal melalui simpul saat ini.
    - Jika tetangga belum ada di dalam antrian prioritas atau jika biaya yang baru dihitung lebih rendah dari biaya yang ada untuk mencapai tetangga tersebut, perbarui biaya dan tambahkan/perbarui tetangga di dalam antrian prioritas.
4. Jika antrian prioritas menjadi kosong dan simpul tujuan belum tercapai, maka tidak ada jalur dari simpul awal ke simpul tujuan.

UCS memiliki sebuah fungsi evaluasi g(n) setiap kali sebuah rute diekspansi yang menghitung jarak dari simpul awal kepada simpul yang mau diekspansi, tergantung dari konteksnya, dapat berbagai macam dan umumnya semakin kecil nilainya semakin baik bagi penentuan simpul berikutnya.

2.1.1 Algoritma UCS yang Digunakan

Algoritma, atau lebih tepatnya fungsi g(n) yang digunakan sebagai fungsi evaluasi bagi

Algoritma UCS ini adalah **jumlah perubahan huruf** dari simpul awal. Dalam pencarian rute terdekat antara simpul awal dan simpul *target*, karena adanya batasan jumlah huruf yang dapat diubah sejumlah satu antar simpulnya, harus dipastikan bahwa jumlah perubahan kata/huruf berupa optimal, memiliki jumlah perubahan minimum.

Karena nilai setiap angka (jika dikonversi) dan aturan-aturan bahasa yang mungkin sebenarnya bisa membantu mempermudah pencarian, nilai-nilai ini tidak memberikan nilai yang pasti bahwa kata tersebut mendekati solusi (kata *target*) sehingga tidak dapat digunakan untuk algoritma UCS ini.

Berikut adalah lampiran fungsinya:

```
public int evalFuncG(int prevCost) {
    return prevCost + 1;
}
```

Gambar 2.1.1.1 Algoritma UCS

Karena berupa jumlah perubahan kata yang dipakai sebagai fungsi evaluasinya dan penelusuran rute dapat dan kemungkinan besar terdiri dari beberapa kata, maka perhitungan nilai evaluasinya dapat dengan mudah dihitung dengan menambah satu saja pada nilai yang simpul yang saat ini mau diekspansi.

2.2 Greedy Best First Search

Algoritma Greedy Best First Search adalah algoritma pencarian graf yang digunakan untuk menemukan jalur terpendek dalam graf berbobot. Algoritma ini mirip dengan algoritma Dijkstra, tetapi alih-alih menggunakan antrian prioritas berdasarkan total biaya untuk mencapai suatu simpul, UCS menggunakan antrian prioritas berdasarkan biaya aktual untuk mencapai setiap simpul dari simpul awal.

Berikut cara kerjanya:

1. Mulai dengan simpul awal dan antrian prioritas kosong.
2. Tambahkan simpul awal ke antrian prioritas dengan biaya 0.
3. Selama antrian prioritas tidak kosong, lakukan hal berikut:
    - Hapus simpul dengan biaya terendah dari antrian prioritas.
    - Jika simpul ini adalah simpul tujuan, maka jalur terpendek telah ditemukan.
    - Jika tidak, perluas simpul (ekspansi) dengan mempertimbangkan semua tetangganya menggunakan fungsi evaluasi h(n).
    - Fungsi evaluasi h(n) ini menghitung biaya dari simpul saat ini kepada simpul akhir *target*.
    - Jika tetangga belum ada di dalam antrian prioritas atau jika biaya yang baru dihitung lebih rendah dari biaya yang ada untuk mencapai tetangga tersebut, perbarui biaya dan tambahkan/perbarui tetangga di dalam antrian prioritas.

4. Jika antrian prioritas menjadi kosong dan simpul tujuan belum tercapai, maka tidak ada jalur dari simpul awal ke simpul tujuan.

Greedy Best First Search memiliki sebuah fungsi evaluasi h(n) yang serupa dengan fungsi evaluasi g(n) pada algoritma UCS yang dilakukan setiap kali sebuah rute diekspansi yang menghitung jarak dari simpul hasil ekspansi sampai simpul akhir, tergantung dari konteksnya, dapat berbagai macam dan umumnya semakin kecil nilainya semakin baik bagi penentuan simpul berikutnya.

2.2.1 Algoritma Greedy Best First Search yang Digunakan

Algoritma, atau lebih tepatnya fungsi h(n) yang digunakan sebagai fungsi evaluasi bagi Algoritma Greedy Best First Search ini adalah **jumlah perbedaan huruf dengan kata *target*** dari simpul hasil ekspansi. Dalam pencarian rute terdekat antara simpul awal dan simpul *target*, serupa dengan algoritma UCS, karena adanya batasan jumlah huruf yang dapat diubah sejumlah satu antar simpulnya, harus diutamakan kata yang memiliki jumlah perbedaan huruf yang seminimal mungkin agar hasil ekspansi dapat terus mengarah pada solusi.

Sekali lagi karena nilai setiap angka (jika dikonversi) dan aturan-aturan bahasa yang mungkin sebenarnya bisa membantu mempermudah pencarian, nilai-nilai ini tidak memberikan nilai yang pasti bahwa kata tersebut mendekati solusi (kata *target*) sehingga tidak dapat digunakan untuk algoritma Greedy Best First Search ini.

Berikut adalah lampiran fungsinya:

```
public int evalFuncH(Tuple wordTuple) {
    int num = 0;
    for (int i = 0; i < targetWord.length(); i++) {
        if (wordTuple.getWord().charAt(i) != targetWord.charAt(i)) {
            num++;
        }
    }
    return num;
}
```

Gambar 2.2.1.1 Algoritma Greedy Best First Search

Karena berupa jumlah perbedaan huruf dengan kata *target* yang dipakai sebagai fungsi evaluasinya dan penelusuran rute dapat dan kemungkinan besar terdiri dari beberapa kata, maka perhitungan memiliki fungsi yang sama semua untuk semua simpul yang diekspansi yaitu menghitung untuk setiap karakter jumlah huruf yang masih berbeda dengan kata *target*.

2.3 A*

Algoritma A* (A-star) adalah sebuah algoritma pencarian jalur yang digunakan dalam masalah pencarian jalur atau perutean dalam graf. Tujuan utamanya adalah untuk menemukan

jalur terpendek antara dua titik dalam graf, sering kali dengan mempertimbangkan biaya yang terkait dengan setiap langkah atau node. Algoritma ini sangat populer dalam bidang kecerdasan buatan, terutama dalam konteks aplikasi seperti game, robotika, dan navigasi.

Berikut cara kerjanya:

1. Mulai dengan simpul awal dan antrian prioritas kosong.
2. Tambahkan simpul awal ke antrian prioritas dengan biaya 0.
3. Selama antrian prioritas tidak kosong, lakukan hal berikut:
    - Hapus simpul dengan biaya terendah dari antrian prioritas.
    - Jika simpul ini adalah simpul tujuan, maka jalur terpendek telah ditemukan.
    - Jika tidak, perluas simpul (ekspansi) dengan mempertimbangkan semua tetangganya menggunakan fungsi evaluasi f(n).
    - Fungsi evaluasi f(n) ini merupakan penjumlahan dari fungsi evaluasi pada UCS (g(n)) dan fungsi evaluasi pada GBFS (h(n)).
    - Jika tetangga belum ada di dalam antrian prioritas atau jika biaya yang baru dihitung lebih rendah dari biaya yang ada untuk mencapai tetangga tersebut, perbarui biaya dan tambahkan/perbarui tetangga di dalam antrian prioritas.
4. Jika antrian prioritas menjadi kosong dan simpul tujuan belum tercapai, maka tidak ada jalur dari simpul awal ke simpul tujuan.

A* menggabungkan dua komponen penting dalam pencarian jalur: biaya sejauh ini dari awal hingga node tertentu (g(n)), dan estimasi biaya dari node tersebut ke titik tujuan (h(n)). Ini menggunakan fungsi evaluasi f(n) = g(n) + h(n) untuk menentukan node mana yang akan dieksplorasi selanjutnya.

Dengan menggunakan pendekatan heuristik, A* dapat menemukan jalur yang optimal dengan keseimbangan antara eksplorasi efisien dan efektivitas dalam menemukan jalur terpendek karena mempertimbangkan dua aspek, yaitu biaya dari simpul awal pada simpul hasil ekspansi dan biaya dari simpul hasil ekspansi pada simpul *target*. Akan tetapi, performanya sendiri masih bergantung pada pendekatan heuristik yang digunakan pada masing-masing UCS dan GBFS.

2.3.1 Algoritma A* yang Digunakan

Algoritma, atau lebih tepatnya fungsi f(n) yang digunakan sebagai fungsi evaluasi bagi Algoritma A* ini adalah **penjumlahan g(n) dan h(n)** dari simpul hasil ekspansi. Dalam pencarian rute terdekat antara simpul awal dan simpul *target*, serupa dengan algoritma kedua algoritma yang lain, karena adanya batasan jumlah huruf yang dapat diubah sejumlah satu antar simpulnya, A* mempertimbangkan kedua aspek penting yaitu jumlah perubahan kata yang sudah dilakukan serta jumlah huruf yang masih berbeda.

Sekali lagi karena nilai setiap angka (jika dikonversi) dan aturan-aturan bahasa yang mungkin sebenarnya bisa membantu mempermudah pencarian dan mengikuti sifat yang dimiliki

oleh algoritma UCS dan GBFS yang digunakan pada program ini, hal-hal tersebut tidak dapat membantu untuk menjadikan sebagai fungsi evaluasi bagi A*.
Berikut adalah lampiran fungsinya:

```
// fungsi G dari UCS
public int evalFuncG(int prevCost) {
    return ucs.evalFuncG(prevCost);
}

// fungsi H dari GBFS
public int evalFuncH(Tuple wordTuple) {
    return gbfs.evalFuncH(wordTuple);
}
```

Gambar 2.3.1.1 Algoritma A*

Karena fungsi evaluasi A* mengikuti fungsi evaluasi dari UCS dan GBFS, pemanggilan fungsi sesuai dengan setiap fungsi dan sifatnya akan berbeda antar kata karena sifat fungsi g(n) yang juga berbeda tergantung pada rute yang sudah dikunjungi yang kemudian menghasilkan simpul hasil ekspansinya.

2.4 Algoritma Utama

Ketiga algoritma yang disajikan di atas masing-masing dibuat dalam sebuah kelas yang merupakan kelas turunan dari Algorithm. Kelas Algorithm memiliki algoritma solve untuk mekanisme ekspansi dan penelusuran dari setiap simpul yang menggunakan *priority queue* karena pada dasarnya ketiga algoritma memiliki proses yang sama, hanya memiliki fungsi evaluasi yang berbeda. Oleh karena itu, kelas Algorithm berupa kelas abstrak dengan metode abstrak EvalFuncG dan EvalFuncH yang harus diimplementasikan oleh setiap kelas. Implementasi fungsi abstraknya beragam yaitu pada UCS diterapkan EvalFuncG saja, pada GBFS hanya EvalFuncH saja, dan pada A* menggunakan EvalFuncG pada UCS dan EvalFuncH pada GBFS. Pemanggilan pada metode utama solve menjumlahkan keduanya dan tetap berjalan dengan baik karena EvalFuncH pada UCS mengembalikan 0 dan EvalFuncG pada GBFS mengembalikan 0. Pada kelas Algorithm ini juga ada beberapa metode tambahan untuk membantu pencarian.

Ada juga kelas tambahan yaitu Tuple yang mencatat kata saat ini, nilai evaluasinya, dan daftar dari semua simpul yang sudah dilalui untuk bisa sampai simpul itu. Kelas ini berfungsi sebagai elemen-elemen pada *priority queue* pada metode solve karena dibandingkan dengan lebih mudah dan tidak mengambil tempat yang begitu besar sehingga lebih baik untuk algoritmanya.

Terakhir, ada kelas Dictionary yang berfungsi untuk membaca kata-kata dari sebuah txt dan mengubahnya menjadi HashSet agar pengaksesan jauh lebih cepat dan terdapat metode tambahan terkait pengolahan kata untuk ekspansi kata, yaitu pencarian setiap kata yang berbeda

satu huruf dengan parameter pada setiap letak huruf dengan semua kemungkinan huruf yang dapat menjadi sebuah kata valid (berada pada *dictionary*).

2.5 Analisis Algoritma

Untuk memperjelas penjelasan di atas, g(n) pada UCS adalah fungsi evaluasi yang sebenarnya merupakan biaya nyata dari simpul awal pada simpul hasil ekspansi, sementara h(n) adalah fungsi evaluasi yang bersifat heuristik yang memberikan perkiraan jarak simpul hasil ekspansi terhadap simpul *target*.

Fungsi dikatakan *admissible* bagi algoritma A* jika nilai dari fungsi heuristik yang dihasilkan yang tidak melebih-lebihkan biaya sebenarnya sehingga menghasilkan nilai kurang dari atau sama dengan nilai hasil fungsinya.

Dalam fungsi yang digunakan untuk algoritma A* ini, fungsi UCS menghitung jumlah perubahan dari simpul awal menuju simpul hasil ekspansi dan fungsi GBFS menghitung jumlah huruf yang masih belum sama dengan simpul tujuan maka jika dijumlahkan akan menghasilkan nilai yang seharusnya sama persis dengan biaya yang dibutuhkan dengan asumsi setiap perubahan mengarah pada solusi.

Pada kasus *Word Ladder*, algoritma UCS dan BFS adalah sama. Ini dikarenakan pada BFS akan dicari semua kata yang memiliki satu huruf berbeda dengan kata yang mau diekspansi, sama dengan UCS yang paling sesuai dengan kasus ini harus menggunakan jumlah perubahan juga. Biaya yang dihasilkan UCS juga menginkrementasi biaya yang sama saja melambangkan penambahan "kedalaman" yaitu perbedaan jumlah huruf dengan simpul awal.

Secara teoritis, algoritma A* akan lebih optimal khususnya untuk kasus dengan perbedaan huruf dengan jumlah yang banyak karena pada jumlah perbedaan huruf yang kecil jumlah simpul yang dibangkitkan tidak berbeda jauh dan bahkan A* akan lebih lambat karena selain menghitung untuk UCS harus juga menghitung untuk GBFS. Akan tetapi, untuk jumlah yang lebih banyak, algoritma UCS yang hanya mempertimbangkan jarak dari simpul awal tanpa mengetahui jarak untuk mencapai simpul tujuan sehingga akan terus mengekspansi sesuai urutan dan kedalaman tanpa mempertimbangkan sama sekali biaya menuju tujuan yang sangat mempengaruhi ekspansi untuk memberikan nilai, bahkan hanya estimasi, terhadap tujuan dengan rute yang lebih singkat.

Secara teoritis, algoritma GBFS tidak menjamin solusi optimal, ini dikarenakan berbeda dengan UCS yang mempertimbangkan jumlah perubahan dari simpul awal, algoritma GBFS hanya mempertimbangkan biaya pada simpul tujuan, yang sepertinya memang terbaik, tetapi simpul yang dibangkitkan tersebut bisa jadi bukan yang terdekat karena mungkin simpul lain dengan nilai yang mungkin lebih rendah pada suatu saat tetapi sebenarnya pada jangka panjangnya lebih mengarah kepada solusi sehingga seharusnya bisa menghasilkan solusi yang lebih optimal.

Bab III
Source Code

3.1 UCS

```java
package Solver;
// jarak dari awal ke saat ini

import Utilities.Tuple;

public class UCS extends Algorithm {
    public UCS(String initialWord, String targetWord) {
        super(initialWord, targetWord);
    }

    // banyaknya perubahan (perubahan seminimal mungkin)
    // lebih kecil -> semakin baik
    public int evalFuncG(int prevCost) {
        return prevCost + 1;
    }

    // tidak ada karena UCS
    public int evalFuncH(Tuple wordTuple) {
        return 0;
    }
}
```

Gambar 3.1.1 Kelas UCS

3.2 GBFS

```java
package Solver;
// jarak dari saat ini ke akhir

import Utilities.Tuple;

public class GBFS extends Algorithm {
    public GBFS(String initialWord, String targetWord) {
        super(initialWord, targetWord);
    }

    // tidak ada karena GBFS
    public int evalFuncG(int prevCost) {
        return 0;
    }

    // jumlah huruf saat ini yang berbeda dengan target
    // lebih kecil -> semakin baik
    public int evalFuncH(Tuple wordTuple) {
        int num = 0;
        for (int i = 0; i < targetWord.length(); i++) {
            if (wordTuple.getWord().charAt(i) != targetWord.charAt(i)) {
                num++;
            }
        }
        return num;
    }
}
```

Gambar 3.2.1 Kelas GBFS

3.3 AStar

```java
package Solver;
// gabungan dari UCS dan GBFS

import Utilities.Tuple;

public class AStar extends Algorithm {
    private UCS ucs;
    private GBFS gbfs;

    public AStar(UCS ucsTemp, GBFS gbfsTemp) {
        // memastikan inisialisasi masih benar
        super(ucsTemp.initialWord, ucsTemp.targetWord);
        this.ucs = ucsTemp;
        this.gbfs = gbfsTemp;
    }

    // fungsi G dari UCS
    public int evalFuncG(int prevCost) {
        return ucs.evalFuncG(prevCost);
    }

    // fungsi H dari GBFS
    public int evalFuncH(Tuple wordTuple) {
        return gbfs.evalFuncH(wordTuple);
    }
}
```

Gambar 3.3.1 Kelas AStar

3.4 Dictionary

```java
package Utilities;

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

public class Dictionary {
    public static Set<String> dictionary = new HashSet<>();

    public static void readDictionary() {
        String filePath = "../src/Utilities/dictionary.txt"; // rute file txt
        try (BufferedReader reader = new BufferedReader(new FileReader(filePath)))
            // baca setiap line dari file
            String word;
            while ((word = reader.readLine()) != null) {
                // pengecekan kata
                dictionary.add(word.toLowerCase());
            }
        } catch (IOException e) {
            // misal error baca
            System.err.println("Error reading the file: " + e.getMessage());
        }
    }

    public static boolean isWordValid(String kata) {
        return dictionary.contains(kata);
    }
}
```

Gambar 3.4.1 Kelas Dictionary

3.5 Tuple

```java
package Utilities;

import java.util.*;

public class Tuple {
    private String word;
    private int evalValue;
    private List<String> prev;

    public Tuple(String word, int evalValue, List<String> prev) {

        this.word = word;
        this.evalValue = evalValue;
        this.prev = new ArrayList<>(prev);
    }

    public Tuple(Tuple temTuple) {

        this.word = temTuple.word;
        this.evalValue = temTuple.evalValue;
        this.prev = new ArrayList<>(temTuple.prev);
    }

    public String getWord() {
        return this.word;
    }

    public int getEvalValue() {
        return this.evalValue;
    }

    public void setEvalValue(int value) {
        this.evalValue = value;
    }

    public List<String> getPrev() {
        return this.prev;
    }
}
```

Gambar 3.5.1 Kelas Tuple

3.6 Algorithm

```java
package Solver;

import Utilities.*;
import Utilities.Dictionary;

import java.util.*;

abstract public class Algorithm {
    protected static List<String> expandedList = new ArrayList<>();

    protected PriorityQueue<Tuple> priorityQueue = new PriorityQueue<>(Comparator.comparing(Tuple::getEvalValue));

    // kata awal
    public String initialWord;
    // kata akhir
    public String targetWord;

    // fungsi g
    abstract public int evalFuncG(int prevCost);

    // fungsi h
    abstract public int evalFuncH(Tuple word);

    public Algorithm(String initialWord, String targetWord) {
        this.initialWord = initialWord;
        this.targetWord = targetWord;
    }

    public List<String> getOneDiffString(String word) {
        List<String> wordArray = new ArrayList<>();
        for (int i = 0; i < word.length(); i++) {
            for (char c = 'a'; c <= 'z'; c++) {
                if (word.charAt(i) == targetWord.charAt(i) || c == word.charAt(i)) {
                    continue;
                }
                String tempWord = new String(word.substring(beginIndex:0, i) + c + word.substring(i + 1));
                if (!expandedList.contains(tempWord) && Dictionary.isWordValid(tempWord)) {
                    wordArray.add(tempWord);
                }
            }
        }
        return wordArray;
    }
```

Gambar 3.6.1 Kelas Algorithm (Part I)

```java
// debug
public void printTuple() {
    for (Tuple t : priorityQueue) {
        System.out.println(t.getWord() + t.getEvalValue());
    }
}

// yang sama hurufnya jadi patokan
public void solve() {
    while (true) { // dilakukan sampai dibreak ketika sama dengan kata yang diekspansi
        // kosong -> tidak bisa expand lagi -> tidak ketemu solusi
        if (priorityQueue.isEmpty() && !expandedList.isEmpty()) {
            System.out.println(x:"Tidak ada solusi!");
            break;
        } else {
            if (priorityQueue.isEmpty()) // belum ada -> inisialisasi
            {
                priorityQueue.add(new Tuple(initialWord, evalValue:0, new ArrayList<String>()));
            }
            // nilai terkecil sekaligus dihapus
            Tuple expandedTuple = priorityQueue.poll();
            // prosedur ekspansi
            // jika elemen terakhir pada path sudah sesuai -> ketemu
            if (expandedTuple.getWord().equals(targetWord)) {
                System.out.println(x:"Path:");
                for (String words : expandedTuple.getPrev()) {
                    System.out.println(words);
                }
                System.out.println(expandedTuple.getWord());
                System.out.println("Banyak node yang dikunjungi: " + expandedTuple.getPrev().size());
                break;
            } else // ekspansi pertama sesuai priority queue
            {
                // katanya belum diekspansi
                if (!expandedList.contains(expandedTuple.getWord())) {
                    List<String> nextWordsList = getOneDiffString(expandedTuple.getWord());
                    List<String> prevList = expandedTuple.getPrev();

                    // masukan semua dalam list
                    for (String word : nextWordsList) {
                        // buat dengan nilai eval dan prev sebelumnya dahulu
                        List<String> newPrevList = new ArrayList<String>(prevList);
                        newPrevList.add(expandedTuple.getWord());
                        // nilai eval untuk inisialisasi tidak penting
                        Tuple newTuple = new Tuple(word, expandedTuple.getEvalValue(), newPrevList);
                        // kasusnya sudah dihandle setiap kelas algoritma
                        int evalValue = evalFuncG(expandedTuple.getEvalValue()) + evalFuncH(newTuple);
                        // ubah value
                        newTuple.setEvalValue(evalValue);
                        // tambah dalam queue
                        priorityQueue.add(newTuple);
                    }
                    // tambah sebagai kata yang sudah diekspansi
                    expandedList.add(expandedTuple.getWord());
                }
            }
        }
    }
}
```

Gambar 3.6.2 Kelas Algorithm (Part II)

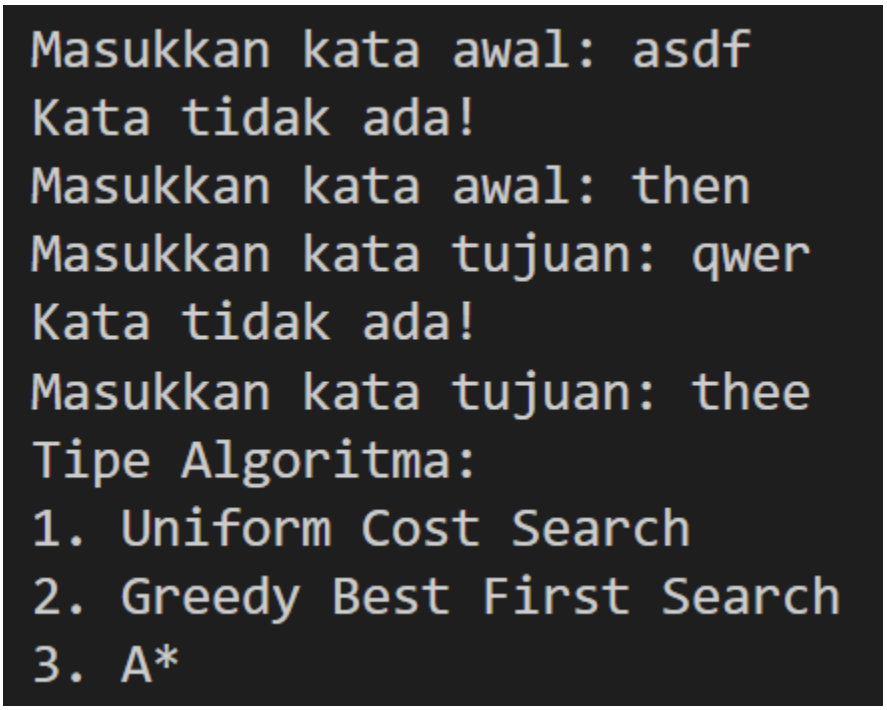
3.7 Main

```java
import java.util.*;

import Solver.*;
import Utilities.Dictionary;

import java.time.Duration;
import java.time.Instant;

public class Main {
    public void start() {
        Dictionary.readDictionary();
        Scanner scanner = new Scanner(System.in);
        System.out.print(s:"Masukkan kata awal: ");
        String kataAwal = scanner.next();
        while (true) {
            if (Dictionary.isWordValid(kataAwal.toLowerCase())) {
                break;
            } else {
                System.out.println(x:"Kata tidak ada!");
                System.out.print(s:"Masukkan kata awal: ");
                kataAwal = scanner.next();
            }
        }
        System.out.print(s:"Masukkan kata tujuan: ");
        String kataAkhir = scanner.next();
        while (true) {
            if (Dictionary.isWordValid(kataAkhir.toLowerCase()) && kataAwal.length() == kataAkhir.length()) {
                break;
            } else if (!Dictionary.isWordValid(kataAkhir.toLowerCase())) {
                System.out.println(x:"Kata tidak ada!");
                System.out.print(s:"Masukkan kata tujuan: ");
                kataAkhir = scanner.next();
            } else {
                System.out.println(x:"Panjang kata tidak sama!");
                System.out.println(x:"Nggak mungkin dong!");
                System.out.print(s:"Masukkan kata tujuan: ");
                kataAkhir = scanner.next();
            }
        }
        System.out.println(x:"Tipe Algoritma:");
        System.out.println(x:"1. Uniform Cost Search");
        System.out.println(x:"2. Greedy Best First Search");
        System.out.println(x:"3. A*");
        System.out.print(s:"Masukkan tipe algoritma (1-3): ");
        int algorithmType = 0;
        algorithmType = scanner.nextInt();
        while (true) {
            if (algorithmType > 0 && algorithmType < 4) {
                break;
            } else {
                System.out.println(x:"Algoritma tidak valid!");
                System.out.print(s:"Masukkan tipe algoritma: ");
                algorithmType = scanner.nextInt();
            }
        }
        scanner.close();
        if (algorithmType == 1) {
            Algorithm tempAlgorithm = new UCS(kataAwal.toLowerCase(), kataAkhir.toLowerCase());
            Instant startTime = Instant.now();
            tempAlgorithm.solve();
            Instant endTime = Instant.now();
            Duration duration = Duration.between(startTime, endTime);
            System.out.println("Waktu eksekusi: " + duration.toMillis() + " ms");
        } else if (algorithmType == 2) {
            Algorithm tempAlgorithm = new GBFS(kataAwal.toLowerCase(), kataAkhir.toLowerCase());
            Instant startTime = Instant.now();
            tempAlgorithm.solve();
            Instant endTime = Instant.now();
            Duration duration = Duration.between(startTime, endTime);
            System.out.println("Waktu eksekusi: " + duration.toMillis() + " ms");
        } else if (algorithmType == 3) {
            UCS tempUCS = new UCS(kataAwal.toLowerCase(), kataAkhir.toLowerCase());
            GBFS tempGBFS = new GBFS(kataAwal.toLowerCase(), kataAkhir.toLowerCase());
            Algorithm tempAlgorithm = new AStar(tempUCS, tempGBFS);
            Instant startTime = Instant.now();
            tempAlgorithm.solve();
            Instant endTime = Instant.now();
            Duration duration = Duration.between(startTime, endTime);
            System.out.println("Waktu eksekusi: " + duration.toMillis() + " ms");
        }
    }

    public void end() {
        System.out.println(x:"Permainan selesai!");
    }

    Run | Debug
    public static void main(String[] args) {
        Main game = new Main();
        game.start();
        game.end();
    }
}
```

Gambar 3.7.1 Kelas Main

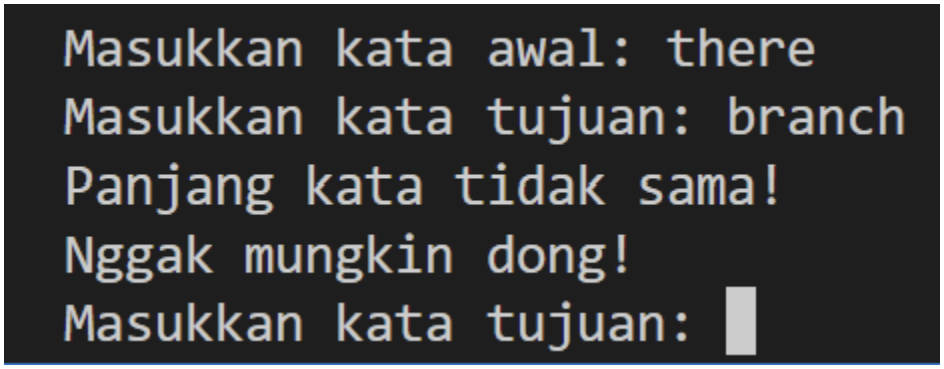
Bab IV
Test Case

4.1 Test Case 1 (Validasi Kata)



Gambar 4.1.1 Test Case Validasi Kata

4.2 Test Case 2 (Validasi Panjang Kata)



Gambar 4.2.1 Test Case Validasi Panjang Kata

4.2 Test Case 3 (Solusi Tidak Ada)

```
Masukkan kata awal: branch
Masukkan kata tujuan: oracle
Tipe Algoritma:
1. Uniform Cost Search
2. Greedy Best First Search
3. A*
Masukkan tipe algoritma (1-3):
Tidak ada solusi!
Waktu eksekusi: 85 ms
Total memori: 4403920 bytes
Permainan selesai!
```

Gambar 4.3.1 Test Case Solusi Tidak Ada

4.4 Test Case 4 (UCS)

```
Masukkan kata awal: branch
Masukkan kata tujuan: chains
Tipe Algoritma:
1. Uniform Cost Search
2. Greedy Best First Search
3. A*
Masukkan tipe algoritma (1-3): 1
Path:
branch
cranch
crance
chance
chanco
charco
charro
charrs
chairs
chains
Banyak node untuk sampai solusi: 9
Banyak node yang dikunjungi: 677
Waktu eksekusi: 95 ms
Total memori: 4823080 bytes
Permainan selesai!
```

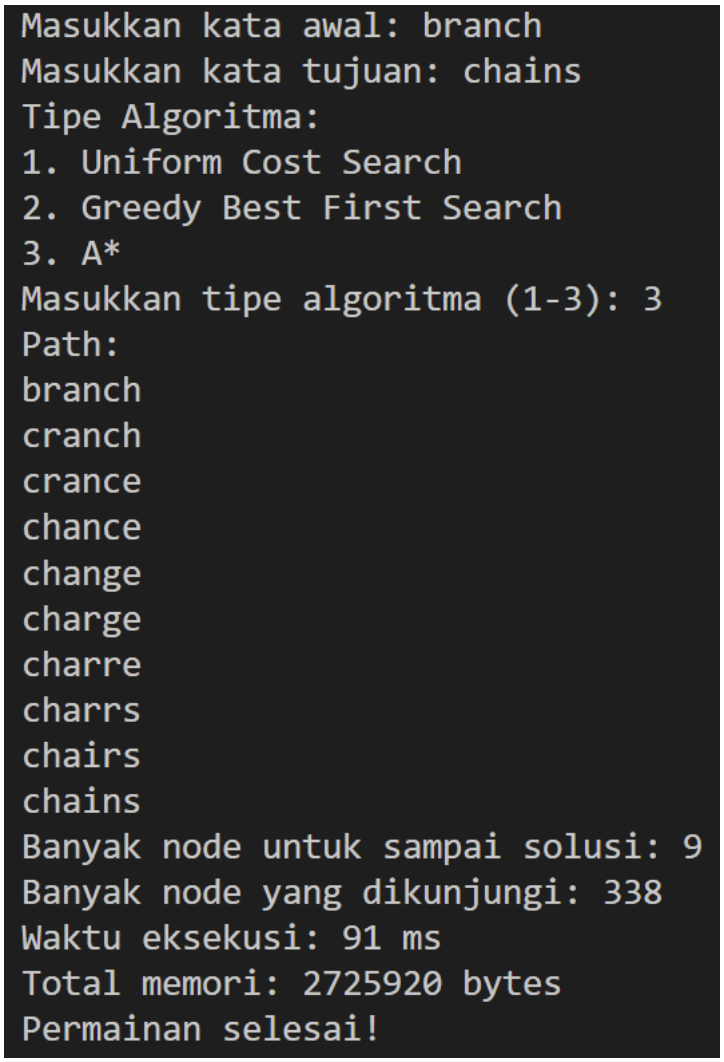
Gambar 4.4.1 Test Case UCS

4.5 Test Case 5 (GBFS)

```
Masukkan kata awal: branch
Masukkan kata tujuan: chains
Tipe Algoritma:
1. Uniform Cost Search
2. Greedy Best First Search
3. A*
Masukkan tipe algoritma (1-3): 2
Path:
branch
cranch
crance
chance
change
changs
chants
charts
charas
charks
charrs
chairs
chains
Banyak node untuk sampai solusi: 12
Banyak node yang dikunjungi: 45
Waktu eksekusi: 24 ms
Total memori: 833736 bytes
Permainan selesai!
```

Gambar 4.5.1 Tes Case GBFS

4.6 Test Case 6 (A*)



Gambar 4.6.1 Test Case A*

Bab V
Analisis Perbandingan

Pada *test case* pertama, dilakukan validasi kata karena tidak semua kata yang dimasukkan pada program ada pada kamus sementara peraturan permainan mengharuskan agar kata yang dimasukkan berupa kata pada kamus bahasa Inggris.

Pada *test case* kedua, dilakukan validasi panjang kata antara kata awal dan kata tujuan yang dimasukkan oleh pengguna karena permainan ini hanya dapat memperbolehkan perubahan huruf, tidak dapat menambah atau mengurangi jumlah huruf sehingga rute tidak dapat ditemukan jika kedua kata yang dimasukkan tidak sama panjangnya.

Pada *test case* ketiga, dapat dilihat pada program bahwa tidak setiap kombinasi kata awal dan kata tujuan dapat menghasilkan jawaban artinya tidak semua kata dapat dihubungkan satu sama lain, bahkan ketika semua rute telah ditelusuri.

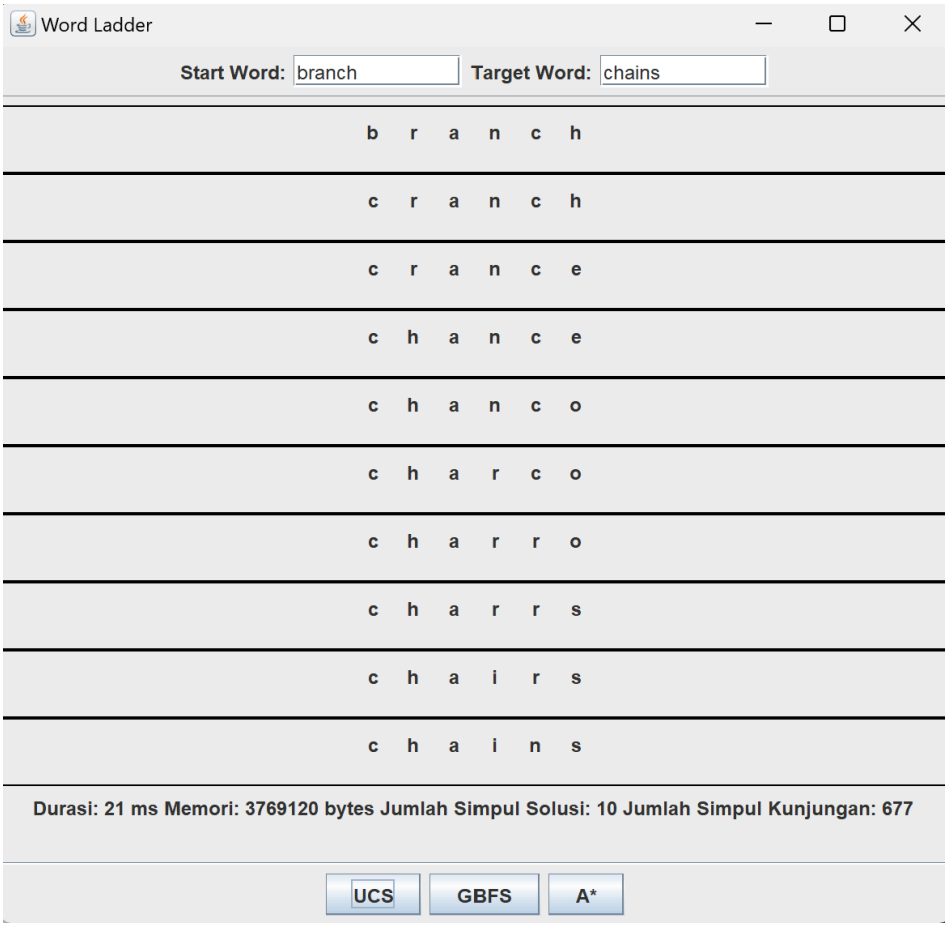
Pada *test case* empat, lima, dan enam dilakukan *test case* yang sama antara kata awal dan kata akhirnya. Dapat dilihat bahwa antara ketiga algoritma tersebut, algoritma UCS dan A* dapat menghasilkan solusi optimal, yaitu panjang minimum rute yang harus dilalui dari kata awal menuju kata tujuan. Dapat dilihat dari setiap parameter yaitu kecepatan, memori yang digunakan, dan jumlah simpul yang dikunjungi yang semuanya sebenarnya saling berkaitan, dapat dilihat bahwa UCS memiliki jumlah paling besar untuk semua, diikuti A*, dan terakhir diikuti GBFS. Ini menunjukkan bahwa dalam kasus ini algoritma UCS kurang efektif dibandingkan A* danA* kurang efektif dibandingkan GBFS dalam pencariannya. Akan tetapi, dapat dilihat bahwa ketiga parameter ini menjadi kurang penting karena solusi yang kurang optimal yang dihasilkan oleh algoritma GBFS. Oleh karena itu, untuk mencapai performa yang baik dan solusi yang optimal, dapat disimpulkan dari ketiga *test case* ini bahwa algoritma A* yang paling baik.

Bab VI
Bonus

Bonus dibuat dengan sebuah GUI sederhana. GUI ini dapat digunakan dengan menjalankan java GUI pada *directory* main. Layar kemudian akan muncul dan dapat dimasukkan kata awal dan kata tujuan, dilanjutkan dengan menekan salah satu tombol algoritma. Setiap tombol algoritma yang ditekan akan langsung menjalankan proses dan memunculkan hasil penelusuran.

Keterbatasan pada fitur bonus ini yaitu tidak dilakukan validasi kata seperti pada program Main menggunakan CLI sehingga akan mencoba mencari jawaban bahkan ketika kata-katanya tidak valid.

Berikut contoh gambar penggunaannya:



Bab VII
Pranala Repository

https://github.com/ZachS17/Tucil3_13522016

Bab VIII
Lampiran

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