

**Laporan Tugas Kecil 3  
IF2211 Strategi Algoritma**

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



**Disusun oleh:**

Maximilian Sulistiyo 13522061

**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA**

**INSTITUT TEKNOLOGI BANDUNG**

**2024**

## Daftar Isi

Daftar Isi.....	2
BAB 1	
Deskripsi Masalah.....	3
BAB 2	
Landasan Teori.....	4
2.1 Penjelasan Algoritma Uniform Cost Search.....	4
2.2 Penjelasan Algoritma Greedy Best First Search.....	4
2.2 Penjelasan Algoritma A*.....	5
BAB 3	
Analisis dan Implementasi Masalah.....	6
3.1 Implementasi Node.....	6
3.2 Definisi $g(n)$ dan $h(n)$ Pada Word Ladder.....	6
3.4 Analisis dan Implementasi dengan Algoritma Greedy Best First Search.....	8
3.5 Analisis dan Implementasi dengan Algoritma A*.....	8
BAB 4	
Source Code.....	10
4.1 Kelas Node.....	10
4.2 Kelas NodePrioQueue.....	11
4.3 Kelas Dictionary.....	11
4.4 Kelas Solver.....	13
4.5 Kelas CLI.....	16
BAB 5	
Percobaan.....	21
5.1 Test Case 1 Lost -> Soul.....	21
5.2 Test Case 2 Rain -> Snow.....	24
5.3 Test Case 3 Mind->Soul.....	27
5.4 Test Case 4 Kiln->Bake.....	30
5.5 Test Case 5 Coal->Fire.....	33
5.6 Test Case 6 Seek -> Find.....	36
BAB 6	
Analisis Solusi UCS, GBFS, dan A*.....	39
6.1 Optimalitas.....	39
6.2 Waktu Eksekusi.....	40
6.3 Memori yang Dibutuhkan.....	41
BAB 7	
Penjelasan Implementasi Bonus.....	42
7.1 Kakas yang Digunakan.....	42
7.2 Menghubungkan Frontend dan Backend.....	43
7.3 Cara Penggunaan.....	43

Daftar Pustaka.....	44
Lampiran.....	45

## BAB 1

### Deskripsi Masalah

Word Ladder merupakan permainan kata yang ditemukan oleh Lewis Carrol pada tahun 1877. Cara kerja permainan ini adalah diberikan kata awal dan kata akhir dimana untuk memenangkan permainan ini kita harus mencari rantai kata yang menghubungkan kata awal dan kata akhir. Batasan dari permainan ini adalah panjang kata awal dan kata akhir sama dan dalam rantai kata tersebut hanya berbeda satu huruf saja (contoh *cat* -> *cut*). Dari permainan ini pemain diharapkan menggunakan kata sedikit dikitnya untuk mendapatkan solusi yang optimal, artinya kita mencari rantai kata dengan panjang terpendek.

Tentunya sebagai manusia normal yang tidak mengetahui seluruh dictionary bahasa inggris permainan ini sulit. Oleh karena itu kami ditugaskan oleh asisten menggunakan algoritma searching untuk mencari rantai kata dari kata awal ke kata akhir. Di sini kami menggunakan algoritma *Uniform Cost Search*, *Greedy Best First Search*, dan *A\**.

## BAB 2

### Landasan Teori

#### 2.1 Penjelasan Algoritma *Uniform Cost Search*

Algoritma *Uniform Cost Search* merupakan salah satu algoritma yang digunakan untuk mencari *node* dan mencari *path* pada sebuah graph. Algoritma ini biasa digunakan pada sebuah graf yang terdiri dari *cost* sisi non-negatif semua.

Jika ditelusuri lebih mendalam, *UCS* dapat dikatakan sebagai tipe algoritma *Breadth First Search* yang membangkitkan node dengan *cost* terendah terlebih dahulu. Algoritma *searching* ini pun termasuk ke dalam kategori *blind search* dimana kita tidak mendapatkan informasi apapun dalam pencarian.

Algoritma ini dipastikan akan mendapatkan solusi yang optimal, hal ini karena pembangkitan node pada *UCS* berdasarkan nilai *cost* terendah maka semua path dengan *cost* terendah akan dicari terlebih dahulu sebelum mempertimbangkan path dengan *cost* lebih tinggi. Oleh karena itu jika *UCS* menemukan sebuah node pada pertama kali maka path yang diambil merupakan path dengan *cost* terendah maka jika goal node ditemukan maka path diambil dipastikan memiliki *cost* terendah. Algoritma *GBFS* juga dipastikan akan mendapatkan solusi karena pada dasarnya dapat dikatakan sebagai *exhaustive search* dengan priority asalkan jumlah node tidak terbatas.

Langkah-langkah *general* pada algoritma ini adalah

1. Buat *priority queue* dengan isi awalnya node awal dengan *cost* sisi 0
2. Ekspansi node dengan *cost* terendah pada *priority queue*. Untuk semua node yang dibangkitkan cari jarak total dari node ke node awal untuk dijadikan nilai dalam perbandingan *priority queue*
3. Periksa apakah node yang sedang diekspansi merupakan solusi, jika iya maka kembalikan path yang diambil, jika tidak maka periksalah node lain pada *priority queue*.
4. Lakukan ini hingga solusi ditemukan atau *priority queue* kosong.

#### 2.2 Penjelasan Algoritma *Greedy Best First Search*

Algoritma *Greedy Best First Search* merupakan salah satu algoritma yang digunakan untuk mencari *node* dan mencari *path* pada sebuah graph. Berbeda dengan *UCS* yang merupakan *blind search*, algoritma termasuk salah satu algoritma *heuristic search* dimana kita mengambil nilai *heuristic* dari sebuah node dengan harapan mencapai hasil optimal. Oleh karena itu *GBFS* membangkitkan node dengan nilai *heuristic* terendah.

Algoritma ini tidak dipastikan akan mendapatkan solusi yang optimal karena hanya berfokus pada pencarian goal secepat mungkin berdasarkan nilai *heuristic*. Kemudian algoritma ini pun tidak *complete* dimana algoritma ini dapat menuju kepada jalan buntu ataupun ke dalam sebuah path cyclic (contoh comb -> cobb -> comb -> cobb -> ....) dimana menurut nilai *heuristic* node yang dibangkitkan adalah node yang membangkitkannya (sehingga terjadi cycle)

Langkah-langkah *general* pada algoritma ini adalah

1. Buat *priority queue* dengan isi awalnya node awal dengan *heuristic value*
2. Ekspansi node dengan *heuristic value* terendah pada *priority queue*. Untuk semua node yang dibangkitkan cari *heuristic value* untuk dijadikan nilai dalam perbandingan *priority queue*
3. Periksa apakah node yang sedang diekspansi merupakan solusi, jika iya maka kembalikan path yang diambil, jika tidak maka periksalah node lain pada *priority queue*.
4. Lakukan ini hingga solusi ditemukan atau *priority queue* kosong.

## 2.2 Penjelasan Algoritma $A^*$

Algoritma  $A^*$  merupakan salah satu algoritma yang digunakan untuk mencari *node* dan mencari *path* pada sebuah graph. Sama dengan *GBFS* algoritma ini mengandalkan nilai *heuristic* untuk mencari solusi namun berbeda dengan itu algoritma ini juga mempertimbangkan *cost* dari sisi-sisi node sama seperti *UCS*. Dapat dikatakan bahwa algoritma ini merupakan perluasan dan penyempurnaan dari kedua algoritma sebelumnya.

Karena kedua hal tersebut algoritma ini *complete* dan juga *optimal* dimana akan dijamin solusi dan solusi dipastikan optimal pada pencarian asalkan memiliki nilai *heuristic* yang *admissible* yang berarti nilai ini tidak *overestimate* dari *cost* sebenarnya.

Langkah-langkah *general* pada algoritma ini adalah

1. Buat *priority queue* dengan isi awalnya node awal dengan  $f(n)$  yang merupakan gabungan dari jarak dari node awal ke node sekarang ditambah nilai *heuristic* ( $f(n) = g(n) + h(n)$ )
2. Ekspansi node dengan  $f(n)$  terendah pada *priority queue*. Untuk semua node yang dibangkitkan cari  $f(n)$  untuk dijadikan nilai dalam perbandingan *priority queue*
3. Periksa apakah node yang sedang diekspansi merupakan solusi, jika iya maka kembalikan path yang diambil, jika tidak maka periksalah node lain pada *priority queue*.
4. Lakukan ini hingga solusi ditemukan atau *priority queue* kosong.

## BAB 3

### Analisis dan Implementasi Masalah

#### 3.1 Implementasi Node

Dalam penyelesaian masalah ini pertama kami membutuhkan struktur data node yang akan digunakan. Pada dasarnya node memiliki atribut :

1. *currentWord* : String
2. *value*: Integer
3. *paths* : List Of Strings

Penulis memutuskan menggunakan atribut karena beberapa hal. Pertama dibutuhkan *currentWord* untuk memeriksa apakah node yang sedang diperiksa merupakan solusi atau bukan dan juga dibutuhkan untuk menentukan nilai *heuristic* dan. Kemudian *value* digunakan untuk menyimpan *cost* atau *heuristic value*, juga dijadikan pembanding pada *priority queue* agar tetap terurut sesuai algoritma yang dipakai. Kemudian *paths* digunakan untuk mengetahui path yang dilalui untuk menuju node tersebut.

#### 3.2 Definisi $g(n)$ dan $h(n)$ Pada Word Ladder

Pada penggunaan algoritma *UCS*, *GBFS*, dan *A\** maka kita perlu mendefinisikan  $g(n)$  yang merupakan *cost* yang dibutuhkan untuk menuju node dan juga perlu didefinisikan  $h(n)$  yang merupakan nilai *heuristic* yang digunakan.

Seperti yang sudah dijelaskan  $g(n)$  merupakan fungsi untuk menghitung *cost* dari node awal ke node sekarang. Maka dengan menggunakan struktur data node yang telah dibuat. Dengan menghitung *length* dari *paths* node yang membangkitkannya, maka kita dapat mendapatkan *cost* yang dibutuhkan untuk mencapai node tersebut, misal dari cat ke dot maka cat->cot->dot, jika diambil path untuk ke cot [cat, cot] memiliki panjang 2 dan benar kita membutuhkan jarak 2 untuk mencapai dot.

Kemudian untuk  $h(n)$  kita harus mencari nilai *heuristic* yang admissible. Sesuai dengan salindia kuliah maka kita harus mencari estimasi total *cost* dari node ke goal. Maka dapat digunakan *difference* atau perbedaan dari kata untuk menghitung nilai tersebut dimana semakin sedikit perbedaan antar kata maka *cost* yang dibutuhkan semakin rendah. Penggunaan nilai *heuristic* ini admissible karena untuk mengubah satu kata menjadi kata lain dibutuhkan  $n$  perubahan sesuai dengan perbedaan kata. Misal dari cat ke dog, dibutuhkan 3 kali perubahan kata misalkan cat->dat->dot->dog. Melihat bahwa terdapat batasan merupakan kata diantaranya

harus berada dalam kamus inggris maka dapat terdapat kata penjemabatan dalam rantai kata. Seperti pada contoh “dat” tidak termasuk pada bahasa inggris maka kita harus melalui kata lain. Oleh karena itu dapat dipastikan bahwa nilai *heuristic* ini *admissible* karena lebih kecil atau sama dengan *cost* aslinya dan tidak akan melebihi *cost* sesungguhnya.

### 3.2 Pembangkitan Node

Untuk mempercepat dan mempermudah pembangkitan node, maka dibuat map terlebih dahulu untuk mendapatkan kata yang mirip. Dapat dilihat bahwa setiap kata memiliki *pattern* untuk kata yang berbeda satu huruf, contohnya pada cat yang memiliki *pattern* berbeda satu huruf pada “\*at”, “c\*t” dan “ca\*t” maka kita juga dapat mencari kata lain yang memiliki pattern yang sama, sebagai contoh lagi pattern “c\*t” memiliki kata seperti cat, cut, cot. Dengan ini kita dapat mendapatkan seluruh kata yang berbeda 1 huruf dengan cat dengan cara mencari semua kata yang memiliki *pattern* yang berbeda 1 huruf dengan cat. Dengan membuat map pun pembangkitan node menjadi lebih cepat dimana kita tidak usah melakukan linear search untuk mencari semua kata yang berbeda satu huruf.

### 3.3 Analisis dan Implementasi dengan Algoritma *Uniform Cost Search*

Seperti yang telah dijelaskan *UCS* mencari solusi dengan cara membangkitkan node dengan *cost* terendah terlebih dahulu, pencarian dilakukan hingga ditemukan solusi atau *priority queue* kosong.

Langkah algoritma ini adalah:

1. Inisialisasi *priority queue* dengan kata awal, diberikan juga *value* 0 (karena *cost* dari awal ke awal adalah 0), diinisialisasi juga map *isVisited* dengan kata awal.
2. Ambil node pada atas *priority queue* yang memiliki nilai *value/cost* terendah dan bangkitkan kata-kata yang berbeda satu huruf dengan memanggil semua *pattern* dari kata. Diperiksa juga apakah kata yang dibangkitkan ada di *isVisited* map, jika sudah ada maka node tersebut tidak dibangkitkan. Masukkan *value/cost* dengan menghitung panjang *paths* dari node pemanggilnya. Kemudian masukkan semua kata-kata tersebut ke dalam *priority queue*
3. Lakukan hal tersebut hingga solusi ditemukan atau *priority queue* menjadi kosong menandakan tidak adanya solusi

Jika ditelusuri lebih mendalam, terdapat batasan dalam permainan ini dimana dari satu kata ke kata lain hanya diperbolehkan perbedaan satu kata. Dengan ini proses pencarian pun sama dengan proses pencarian pada algoritma *BFS*. Hal ini dapat dijelaskan dengan melihat bahwa pada kasus ini *cost* sama



dengan step yang diambil. Dalam urutan node pun *priority queue* disini menjadi sama dengan *queue* biasa yang dipakai oleh *BFS* dimana node yang lebih dekat dengan kata awal maka akan ditelusuri terlebih dahulu dilanjutkan dengan anak-anak dari node tersebut yang memiliki jarak 1 lebih jauh, terlihat hal tersebut merupakan pencarian melebar yang sama pada *BFS*.

### 3.4 Analisis dan Implementasi dengan Algoritma *Greedy Best First Search*

Seperti dijelaskan sebelumnya algoritma ini membangkitkan node dengan nilai *heuristic* terendah terlebih dahulu. Pencarian dilakukan hingga ditemukan solusi, *priority queue* kosong, atau terdeteksi loop. Maka langkah algoritma ini adalah :

1. Inisialisasi *priority queue* dengan kata awal, diberikan juga *value* nilai *heuristic* sesuai dengan  $f(n)$ , diinisialisasi juga map *isVisited* dengan kata awal.
2. Ambil node pada atas *priority queue* yang memiliki nilai *value/heuristic* terendah. Periksa apakah kata tersebut sudah pernah dibangkitkan, jika iya maka tidak ada solusi. Hal ini untuk mencegah loop dimana jika terdapat kata A yang membangkitkan B dengan *heuristic* terendah, kemudian jika B membangkitkan A dengan *heuristic* terendah maka A pun akan membangkitkan B kembali dan seterusnya menyebabkan cyclic loop.
3. Jika tidak ada pada *isVisited* maka bangkitkan kata-kata yang berbeda satu huruf dengan memanggil semua pattern dari kata. Masukkan *value/heuristic* dengan menggunakan  $f(n)$  yang sudah didefinisikan. Kemudian masukkan semua kata-kata tersebut ke dalam *priority queue*.
4. Lakukan hal tersebut hingga *priority queue* kosong atau ditemukannya solusi

Salah satu kontra dari *GBFS* adalah tidak dijaminkannya solusi yang optimal. Hal ini karena sifatnya yang *heuristic* dan juga tidak mempertimbangkan *cost* asli dari setiap penelusuran. Hal tersebut dapat menyebabkan masalah dimana menurut  $f(n)$  kita harus melalui suatu node namun pada akhirnya menghasilkan path yang tidak optimal. Algoritma ini hanya mementingkan cepatnya menuju node dengan harapan path tersebut merupakan solusi optimal

### 3.5 Analisis dan Implementasi dengan Algoritma *A\**

Seperti dijelaskan sebelumnya algoritma ini mempertimbangkan *cost* dan juga nilai *heuristic* dimana kedua nilai tersebut ditambahkan. Pembangkitan node pun dari total *value* terendah pada *priority queue*. Pencarian dilakukan hingga ditemukan solusi atau *priority queue* kosong. Langkah algoritma ini adalah:

1. Inisialisasi *priority queue* dengan kata awal, diberikan juga *value* yaitu *cost* untuk menuju node ini ditambah dengan nilai *heuristic* sesuai dengan  $f(n)$ , diinisialisasi juga map *isVisited* dengan kata awal.
2. Ambil node pada atas *priority queue* yang memiliki nilai total  $cost + heuristic$  terendah dan bangkitkan kata-kata yang berbeda satu huruf dengan memanggil semua pattern dari kata. Diperiksa juga apakah kata yang dibangkitkan ada di *isVisited* map, jika sudah ada maka node tersebut tidak dibangkitkan. Masukkan  $cost + heuristic$  dengan menghitung panjang paths dari node pemanggilnya ditambah dengan nilai *heuristic* dari kata ke kata akhir. Kemudian masukkan semua kata-kata tersebut ke dalam *priority queue*
3. Lakukan hal tersebut hingga solusi ditemukan atau *priority queue* menjadi kosong menandakan tidak adanya solusi

Algoritma ini dijamin menemukan solusi dan solusi tersebut optimal, sama seperti dalam penggunaan algoritma *UCS*. Namun algoritma ini memiliki keuntungan dibandingkan dengan *UCS* dalam kasus penggunaan ini. Seperti sudah dijelaskan sebelumnya, *UCS* disini menjadi sama halnya dengan *BFS* sehingga menjadi *exhaustive search*. Namun dalam algoritma ini terdapat nilai *heuristic* yang membuat pengurutan pemrosesan node pun menjadi lebih cepat karena lebih berfokus pada kata akhir. Contohnya pada kata cold ke warm dimana pada *UCS* mungkin terjadi *queue* seperti ini {bold, cowl, cord} yang semuanya berbeda satu huruf saja, maka akan terjadi pencarian mendalam. Namun dengan nilai *heuristic* maka terlihat bahwa cord merupakan kata paling dekat sehingga akan ditelusuri terlebih dahulu. Dengan ini pun pembangkitan node akan menjadi lebih efisien dan pencarian pun menjadi lebih cepat.

## BAB 4

### Source Code

#### 4.1 Kelas Node

Kelas Node merupakan implementasi dari struktur data node yang telah dijelaskan diatas

##### Node.java

```
import java.util.*;

public class Node implements Comparable<Node>{
    private String currentWord;
    private int value;
    private List<String> paths;

    public Node(String currentWord, int value,List<String> paths){
        this.currentWord = currentWord;
        this.value = value;
        this.paths= new ArrayList<>(paths);
        this.paths.add(currentWord);
    }

    public int compareTo(Node other) {
        return Integer.compare(this.value, other.value);
    }

    public String getCurrentWord() {
        return this.currentWord;
    }

    public int getValue() {
        return this.value;
    }

    public List<String> getPaths() {
        return this.paths;
    }
}
```

Method yang berada di kelas ini hanyalah getter dan juga override dari compareTo yang digunakan pada *priority queue*

## 4.2 Kelas NodePrioQueue

Kelas ini digunakan sebagai implementasi dari *priority queue* yang sudah dijelaskan sebelumnya.

### NodePrioQueue.java

```
import java.util.*;

public class NodePrioQueue {
    private PriorityQueue<Node> queue;

    public NodePrioQueue() {
        this.queue = new PriorityQueue<Node>();
    }

    public void addNode(Node node) {
        queue.add(node);
    }

    public Node remove() {
        return queue.poll();
    }

    public boolean isEmpty() {
        return queue.isEmpty();
    }
}
```

Method yang ada disini digunakan untuk menambahkan (addNode()), menghapus elemen teratas (remove()) dan memeriksa apakah queue kosong.

## 4.3 Kelas Dictionary

Kelas ini menangani semua hal mengenai dictionary inggris. Dia membuat list kata kata inggris dari sebuah file.txt dan juga membuat patternMap yang sudah dijelaskan sebelumnya

### Dictionary.java

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

public class Dictionary {

    private List<String> words;
    private Map<String, List<String>>> patternMap;

    public Dictionary() {
```

```

        this.words = createWordList();
        this.patternMap = createPatternMap();
    }

    public List<String> createWordList() {
        ArrayList<String> words = new ArrayList<String>();
        File dictionaryFile = new File("src/data.txt");
        try {
            Scanner scan = new Scanner(dictionaryFile);
            while (scan.hasNextLine()) {
                String word = scan.nextLine().trim();
                words.add(word);
            }
            scan.close();
        } catch (FileNotFoundException e) {
            System.out.println("File tidak ditemukan!!");
        }

        return words;
    }

    public Map<String, List<String>> createPatternMap() {
        Map<String, List<String>> wordPatternMap = new HashMap<>();
        List<String> words = getWords();
        for (String word : words) {
            for (int i = 0; i < word.length(); i++) {
                String pattern = word.substring(0, i) + "*" + word.substring(i + 1);
                wordPatternMap.computeIfAbsent(pattern, k -> new ArrayList<>()).add(word);
            }
        }
        return wordPatternMap;
    }

    public List<String> getWords() {
        return this.words;
    }

    public Map<String, List<String>> getPatternMap() {
        return this.patternMap;
    }

    public List<String> getSimilarWords(String originWord) {
        List<String> similarWords = new ArrayList<>();
        for (int i = 0; i < originWord.length(); i++) {
            String pattern = originWord.substring(0, i) + "*" + originWord.substring(i + 1);
            List<String> words = this.patternMap.getOrDefault(pattern, new ArrayList<>());
            for (String word : words) {
                if (!word.equals(originWord)) {
                    similarWords.add(word);
                }
            }
        }
    }

```

```

    }
    }
    return similarWords;
}

public boolean isInDictionary(String word){
    return this.words.contains(word);
}
}

```

Method `createWordList()` membaca `file.txt` dan mengambil seluruh kata dan mengembalikan sebuah list. Method `createPatternMap` memanggil list kata yang telah dibuat dan membuat pattern map untuk semua kata pada list tersebut. Method `getSimilarWords()` mengambil `patternMap` sebelumnya dan menerima suatu kata untuk mengembalikan semua kata yang berbeda satu kata dengan kata yang diinput. Kemudian method `isInDictionary()` memeriksa apakah kata terdefinisi pada `.txt` yang dipakai. Selebihnya terdapat method-method getter

#### 4.4 Kelas Solver

Kelas ini bertanggung jawab pada pencarian solusi maka pada kelas ini terdapat ketiga algoritma pencarian

##### **Solver.java**

```

public class Solver {
    private Dictionary dictionary;
    private int nodeTraversed;

    public Solver() {
        this.dictionary = new Dictionary();
        this.nodeTraversed = 0;
    }

    public Dictionary getDictionary() {
        return dictionary;
    }

    public int getNodeTraversed() {
        return nodeTraversed;
    }

    public int getLetterDifference(String word1, String word2){
        int difference = 0;
        for(int i = 0; i < word1.length(); i++){

```

```

        if(word1.charAt(i) != word2.charAt(i)){
            difference++;
        }
    }
    return difference;
}

public int gN(Node currentNode){
    return currentNode.getPaths().size();
}

public int hN(String currentWord, String endWord){
    return getLetterDifference(currentWord, endWord);
}

public int fN(Node currentNode, String currentWord, String endWord){
    return gN(currentNode) + hN(currentWord, endWord);
}

public List<String> solveUCS(String startWord, String endWord) {
    this.nodeTraversed = 0;
    NodePrioQueue queue = new NodePrioQueue();
    List<String> initPath = new ArrayList<>();
    Set<String> isVisited = new HashSet<>();
    Node startNode = new Node(startWord, 0, initPath);
    queue.addNode(startNode);
    isVisited.add(startWord);

    while(!queue.isEmpty()){
        Node currentNode = queue.remove();
        this.nodeTraversed++;

        if (currentNode.getCurrentWord().equals(endWord)) {
            return currentNode.getPaths();
        }

        List<String> children = dictionary.getSimilarWords(currentNode.getCurrentWord());
        for (String child : children) {
            if (!isVisited.contains(child)) {
                Node newNode = new Node(child, gN(currentNode), currentNode.getPaths());
                queue.addNode(newNode);
                isVisited.add(child);
            }
        }
    }
    return initPath;
}

public List<String> solveGBFS(String startWord, String endWord) {
    this.nodeTraversed = 0;

```

```

NodePrioQueue queue = new NodePrioQueue();
List<String> initPath = new ArrayList<>();
Set<String>isVisited = new HashSet<>();
Node startNode = new Node(startWord, hN(startWord, endWord), initPath);
queue.addNode(startNode);

while(!queue.isEmpty()){
    Node currentNode = queue.remove();
    this.nodeTraversed++;

    if (currentNode.getCurrentWord().equals(endWord)) {
        return currentNode.getPaths();
    }

    if(isVisited.contains(currentNode.getCurrentWord() )){
        return initPath;
    }

    List<String> children = dictionary.getSimilarWords(currentNode.getCurrentWord());
    for (String child : children) {
        Node newNode = new Node(child, hN(child, endWord), currentNode.getPaths());
        queue.addNode(newNode);
    }
    isVisited.add(currentNode.getCurrentWord());
}
return initPath;
}

public List<String> solveAStar(String startWord, String endWord) {
    this.nodeTraversed = 0;
    NodePrioQueue queue = new NodePrioQueue();
    List<String> initPath = new ArrayList<>();
    Node startNode = new Node(startWord, hN(startWord, endWord), initPath);
    queue.addNode(startNode);
    Set<String>isVisited = new HashSet<>();
    isVisited.add(startWord);

    while(!queue.isEmpty()){
        Node currentNode = queue.remove();
        this.nodeTraversed++;

        if (currentNode.getCurrentWord().equals(endWord)) {
            return currentNode.getPaths();
        }

        List<String> children = dictionary.getSimilarWords(currentNode.getCurrentWord());
        for (String child : children) {
            if (!isVisited.contains(child)) {
                Node newNode = new Node(child, fN(currentNode, child, endWord),

```



```
currentNode.getPaths();
    queue.addNode(newNode);
    isVisited.add(child);
}
}
}
return initPath;
}
}
```

Untuk method *UCS*, *GBFS* dan *A\** hanya mengimplementasikan penjelasan algoritma yang telah dijelaskan diatas. Terdapat juga implementasi dari  $g(n)$  pada  $gN$  yang menghitung panjang dari path, kemudian implementasi  $h(n)$  pada  $hN$  yang menghitung perbedaan kata dari kata sekarang ke kata akhir, kemudian  $f(n)$  pada  $fN$  yang menambahkan kedua nilai tersebut. Kemudian pada kelas ini terdapat atribut `nodeTraversed` yang dibutuhkan pada output dan terdapat getternya. Method `getLetterDifference` menghitung banyak nya perbedaan dari `word1` dengan `word2`. Kelas ini juga memiliki atribut dictionary yang digunakan pada ketiga metode pencarian.

## 4.5 Kelas CLI

Kelas ini adalah kelas yang handle input CLI dari pengguna

# CLI.java

[illegible]



```

        System.out.print(word.charAt(i));
    }
}
System.out.println();
}

public String readWord(String type, Dictionary dictionary, Scanner scanner){
    String word = "";
    boolean valid = false;
    while(!valid){
        System.out.print("Enter " + type + " word: ");
        word = scanner.nextLine().trim().toLowerCase();
        if(dictionary.isInDictionary(word)){
            valid = true;
        } else {
            System.out.println("Your word is not in the dictionary, please try again !");
        }
    }
    return word;
}

public void run(){
    Solver solver = new Solver();
    boolean run = true;
    Scanner scanner = new Scanner(System.in);
    while(run){
        writeTitle();
        int choice = 0;
        long start = 0;
        long end = 0;
        try{
            String startWord = "";
            String endWord = "";
            startWord = readWord("start", solver.getDictionary(), scanner);
            endWord = readWord("end", solver.getDictionary(), scanner);
            if(!(startWord.length() == endWord.length())) {
                System.out.println("Your words do not have the same length, there is no solution");
                System.out.print("Do you want to try again? (Default Y) (y/n): ");
                String tryAgain = scanner.nextLine().trim().toLowerCase();
                if(tryAgain.equals("n")){
                    run = false;
                }
                continue;
            }
        }
        boolean valid = false;
        while(!valid){
            System.out.println("Choose the Algorithm");
            System.out.println("1. UCS");
            System.out.println("2. G-BFS");
            System.out.println("3. A*");

```

```

    try {
        System.out.print("Choice: ");
        choice = scanner.nextInt();
        if (choice >= 1 && choice <= 3) {
            valid = true;
        } else {
            System.out.println("Please enter a valid choice between 1-3 - _-");
            continue;
        }
    } catch (InputMismatchException e) {
        System.out.println("Please enter a number - _-");
    }
    scanner.nextLine();
}
List<String> result = new ArrayList<>();
Runtime runtime = Runtime.getRuntime();
long beforeUsedMem = runtime.totalMemory() - runtime.freeMemory();
start = System.currentTimeMillis();
result = switch (choice) {
    case 1 -> solver.solveUCS(startWord, endWord);
    case 2 -> solver.solveGBFS(startWord, endWord);
    case 3 -> solver.solveAStar(startWord, endWord);
    default -> new ArrayList<>();
};
end = System.currentTimeMillis();
long afterUsedMem = runtime.totalMemory() - runtime.freeMemory();
long actualMemUsed = afterUsedMem - beforeUsedMem;

if(!result.isEmpty()){
    System.out.println("Path found:");
    for (String word : result) {
        writeSameLetter(word, endWord);
    }
    System.out.println("Node traversed: " + solver.getNodeTraversed());
    System.out.println("Runtime: " + (end-start) + "ms");
    System.out.println("Memory used: " + actualMemUsed);
}else{
    System.out.println("No path found");
}

} catch (NoSuchElementException e) {
    System.out.println("No input");
} catch (IllegalStateException e) {
    System.out.println("Error occured, scanner closed.");
}
System.out.print("Do you want to exit? (Default n) (y/n): ");
String exit = scanner.nextLine().trim().toLowerCase();
if(exit.equals("y")){
    writeEnding();
    scanner.close();
}

```

```
        run = false;
    }
}
}
```

Method writeSameLetter digunakan untuk mewarnakan kata yang sama menjadi warna hijau untuk memperbagus visualisasi. Method readWord digunakan untuk menerima input kata dari pengguna. Terakhir method run digunakan untuk menjalankan CLI.

## **BAB 5**

### **Percobaan**

#### **5.1 Test Case 1 Lost -> Soul**

##### **UCS Lost -> Soul CLI**



```

      .---.      .---.
    .---|__|      .-  |~~~|
  .--|===|--|_   |__|   |~~~|--.
| |===| |'\   .---!~| .--| |--| | | |
|%%| | |'\   |===| |--|%%| | |
|%%| | |\.'\ | | |__| | | |
| | | | \ \ |===| |==| | | |
| | |__| \.'\ | |__| |~~~|__|
| |===|--| \.'\|===|~|--|%%|~~~|--|
A--A--'---A   '-1'---A--A--A--'---A

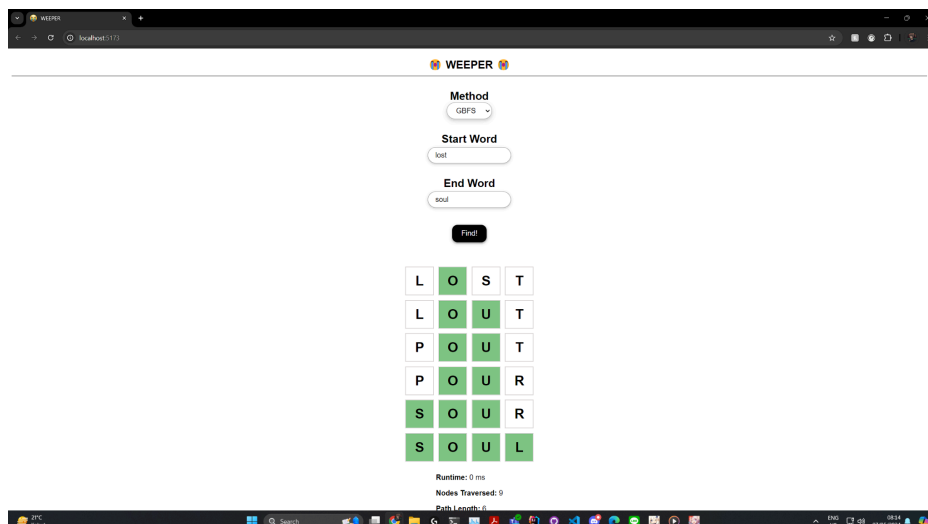
--      --      --      --      --
\ \   / /__  _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
\ \ / / / _ \ ' _ / _ ' _ _ _ _ _ _ _ _ _ _ _
\ V V / ( _ ) | | | ( _ | | | ( _ | | | ( _ |
  \_/_/ \_/_/_/_/ \_/_/_/ \_/_/_/ \_/_/_/ \_/_/_/

Word Ladder autocomplete search using UCS, G-BFS, or A*
By: Maximilian Sulistiyo 13522061

Enter start word: lost
Enter end word: soul
Choose the Algorithm
1. UCS
2. G-BFS
3. A*
Choice: 2
Path found:
lost
lout
pout
pour
sour
soul
Node traversed: 9
Runtime: 0ms
Path Length: 6
Do you want to exit? (Default n) (y/n):

```

## GBFS Lost -> Soul GUI



## A\* Lost -> Soul CLI

```

      .---.
    .---|_|
    .---|===|--|_
    | |===| |'\
    |%%| | |.\
    |%%| | |\.\
    | | | |\ \
    | | | _| \
    | | | _| \.\
    | |===|--| \.\
    ^---^---^---^

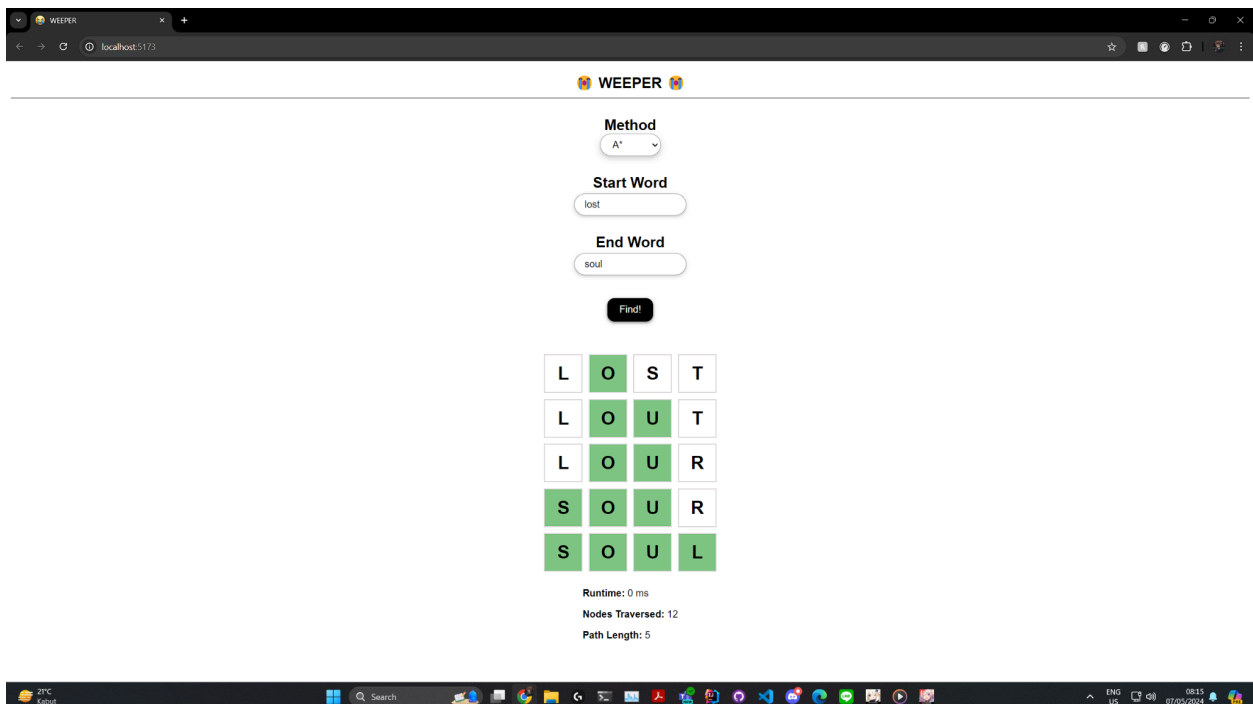
-- -- -- -- --
\ \ / / _ _ _ _ _
\ \ / / _ _ _ _ _
\ \ / / ( ) ( ) ( ) ( )
\ \ / / _ _ _ _ _

Word Ladder autocomplete search using UCS, G-BFS, or A*
By: Maximilian Sulistiyo 13522061

Enter start word: lost
Enter end word: soul
Choose the Algorithm
1. UCS
2. G-BFS
3. A*
Choice: 3
Path found:
lost
lout
lour
sour
soul
Node traversed: 12
Runtime: 1ms
Path Length: 5
Do you want to exit? (Default n) (y/n): |

```

## A\* Lost -> Soul GUI





## 5.2 Test Case 2 Rain -> Snow

## UCS Rain -> Snow CLI

[illegible]

## UCS Rain -> Snow GUI

WEEPER

Method  
UCS

Start Word  
rain

End Word  
snow

Find!

R	A	I	N
S	A	I	N
S	H	I	N
S	H	I	P
S	H	O	P
S	H	O	W
S	N	O	W

Runtime: 0 ms  
Nodes Traversed: 85  
Path Length: 7

## GBFS Rain -> Snow CLI

```

      .--.      .---.
    .---|__|      .-.  |~~~|
  .--|===|--|_  |__|  |~~~|--.
|  |===|  |'\    .---!~|  .--|  |--| | | |
|%%|  |  |.\    |===|  |--|%%|  |  |
|%%|  |  |.\.'\  |  |  |__|  |  |  |
|  |  |  | \ \  |===|  |==|  |  |  |
|  |  |__|  \.'\  |__|__|  |~~~|__|
|  |===|--|  \.'\|===|~|--|%%|~~~|--|
^--^--^'--^  ^-'--^--^--^--^'--^


--      --      -      -      -
\ \      / /__ _ __ __| | | |  __ _ __| |__| | __ _ __
\ \ /\ / / _ \ | '___/ _' | | | / _' | / _' | / _' | \ '___|
 \ v v / ( ) | | | ( _ | | |__| ( _ | ( _ | ( _ | __/ |
  \/\_/ \___/|_| \___|_| |____\___| \___|_| \___|_| \___|_|

Word Ladder autocomplete search using UCS, G-BFS, or A*
By: Maximilian Sulistiyo 13522061

Enter start word: rain
Enter end word: snow
Choose the Algorithm
1. UCS
2. G-BFS
3. A*
Choice: 2
No solution found
Node traversed: 5
Runtime: 2ms

```

## GBFS Rain -> Snow GUI



---

Method

GBFS

Start Word

rain

End Word

snow

Find

R	A	I	N
?	?	?	?
S	N	O	W

Runtime: 0 ms

Nodes Traversed: 5

No Solution!

### A\* Rain -> Snow CLI

[illegible]

## A\* Rain -> Snow GUI

🗣️ **WEEPER** 🗣️

## Method

**Start Word**

rain

### End Word

**SNOW**

Find!

R	A	I	N
S	A	I	N
S	H	I	N
S	H	I	P
S	H	O	P
S	H	O	W
S	N	O	W

Runtime: 0 ms

**Nodes Traversed: 85**

Path Length: 7

### 5.3 Test Case 3 Mind->Soul

## UCS Mind->Soul CLI

[illegible]

## UCS Mind->Soul GUI

 **WEEPER** 

**Method**

UCS

**Start Word**  
mind

**End Word**

soul

Find!

M	I	N	D
M	I	L	D
M	I	L	L
M	A	L	L
M	A	U	L
S	A	U	L
S	O	U	L

Runtime: 8 ms  
Nodes Traversed: 3361  
Path Length: 7

## GBFS Mind->Soul CLI

```

      .---.      .---.
    .---|_ _|      .---|_ _ _|
  .---|===|--|_ _      |_ _|      | _ _ _|---.
| |===| |'\      .---|~|      .---| |---| | | |
|%%| | |'\      |===| |--|%%| | |
|%%| | |\.'\      | | _ _| | | |
| | | | |\.'\      |===| |==| | | |
| | _ _| |\.'\      |_ _ _| | _ _ _|
| |===|--| |\.'\|===|~|--|%%| _ _ _|---|
A _ _ _ _ _ _ A      ~ _ _ _ _ _ A _ _ _ _ _ _

--      --      --      --      --      --
\ \      / / _ _ _ _ _| | | | _ _ _ _ _| | _ _ _ _ _
\ \ \ / / / _ _| ' _ _ / ' | | | / _ _ / / _ _ / _ _ \ ' _ _|
 \ \ \ / / ( ) | | | C _ | | | _ _| C _ | | C _ | | C _ | | _ _ / |
  \ \ \ / \ _ _ / | | | \ _ _ _ _| \ _ _ _ _| \ _ _ _ _| \ _ _ _ _|
Word Ladder autocomplete search using UCS, G-BFS, or A*
By: Maximilian Sulistiyo 13522061

Enter start word: mind
Enter end word: soul
Choose the Algorithm
1. UCS
2. G-BFS
3. A*
Choice: 2
Path found:
mind
bind
bond
bold
sold
sola
sols
sous
soul
Node traversed: 13
Runtime: 0ms
Path Length: 9
```

## GBFS Mind->Soul GUI

WEEPER

Method  
GBFS

Start Word  
mind

End Word  
soul

Find

M	I	N	D
B	I	N	D
B	O	N	D
B	O	L	D
S	O	L	D
S	O	L	A
S	O	L	S
S	O	U	S
S	O	U	L

Runtime: 0 ms

Nodes Traversed: 13

Path Length: 9

## A\* Mind->Soul CLI

```

      .--.      .--.      .--.
    .---|___|    .-|    |~|~|~|
  .--|===|--|_   |__|    |~|~|~|--.
| |===| |'\    .---!~|   .--| |--| | | | |
|%%| | |.\    |===| |--|%%| | |
|%%| | |.\.\  | |__| | | |
| | | | |.\ \ |===| |==| | | |
| | |__| \.\ \ |__|__| |~|~|__|
| |===|--|   \.\|===|~|--|%%|~|~|--|
A--A--'---A   '---'---A--A--A--'---'

--
\ \      / /__  _ _ _ | | |  _ _ _ | _ _ _ | _ _ _ | _ _ _ |
\ \ \ / / _ \ | ' _ / _ | | |  / _ \ / _ \ / _ \ / _ \ ' _ |
 \ \ \ / / ( ) | | | ( _ | | | ( _ | ( _ | ( _ | ( _ | _ / |
  \ \ \ / \ _ _ / | _ | \ _ _ _ \ _ _ _ \ _ _ _ \ _ _ _ \ _ _ _ |


Word Ladder autocomplete search using UCS, G-BFS, or A*
By: Maximilian Sulistiyo 13522061

Enter start word: mind
Enter end word: soul
Choose the Algorithm
1. UCS
2. G-BFS
3. A*
Choice: 3
Path found:
mind
mild
sild
sill
sall
saul
soul
Node traversed: 144
Runtime: 1ms
Path Length: 7

```

## A\* Mind->Soul GUI

---



---

Method  
A\*

Start Word  
mind

End Word  
soul

Find!

M	I	N	D
M	I	L	D
S	I	L	D
S	I	L	L
S	A	L	L
S	A	U	L
S	O	U	L

Runtime: 1 ms

Nodes Traversed: 144

Path Length: 7

## 5.4 Test Case 4 Kiln->Bake

## UCS Kiln->Bake CLI

[illegible]

## UCS Kiln->Bake GUI

WEEPER

Method

UCS

Start Word

kiln

End Word

bake

Find!

K	I	L	N
K	I	R	N
K	A	R	N
B	A	R	N
B	A	R	E
B	A	K	E

Runtime: 5 ms

Nodes Traversed: 2072

Path Length: 6

## GBFS Kiln->Bake CLI

```

      .---.
    .---|_|      .-      |~~~|
  .--|===|--|_|      |_|      |~~~|--.
|_||===|_|'\      .---!~| .---|_|--| | | |
|%%|_|_|'\      |===|_|--|%%|_|_|
|%%|_|_|'\'\      |_|_|_|_|_|_|
|_|_|_|'\      |===|_|==|_|_|_|
|_|_|_|'\'\      |_|_|_|_|_|_|
|_|_|_|_|'\      |_|_|_|_|_|_|
|_|===|--|_|'\      |~|~|%%|~~~|--|
^---^---'^---^'^---^---^---^---'^---^
--      --      -      -      -
\ \      / /      _ _      | |      _ _      _ _      _ _      _ _
\ \ / /      \ \      _ _      | |      / /      \ \      \ \      \ \
 \ \ \ /      \ \      | |      | |      | |      | |      | |      | |
  \ \ \ /      \ \      | |      | |      | |      | |      | |      | |
   \ \ \ /      \ \      | |      | |      | |      | |      | |      | |
Word Ladder autocomplete search using UCS, G-BFS, or A*
By: Maximilian Sulistiyono 13522061

Enter start word: kiln
Enter end word: bake
Choose the Algorithm
1. UCS
2. G-BFS
3. A*
Choice: 2
Path found:
kiln
kirn
kern
barn
bare
bake

Node traversed: 6
Runtime: 0ms
Path Length: 6

```

## GBFS Kiln->Bake GUI

**WEEPER**

## Method

### Start Word

kilr

## End Words

bal

Find

K	I	L	N
K	I	R	N
K	A	R	N
B	A	R	N
B	A	R	E
B	A	K	E

Runtime: 0 m

**Nodes Traversed:**

**Path Length:**



## A\* Kiln->Bake CLI

[illegible]

## A\* Kiln->Bake GUI

WEEPER

Method

A\*

Start Word

kiln

End Word

bake

Find!

K	I	L	N
K	I	L	L
B	I	L	L
B	A	L	L
B	A	L	E
B	A	K	E

Runtime: 0 ms

Nodes Traversed: 12

Path Length: 6

### 5.5 Test Case 5 Coal->Fire

## UCS Coal-> Fire CLI

```

      .---.
      .---|___|      .-.-      |~~~~|
      .--|===|--|_      |_      |~~~~|--.
      | |===|  |'\      .---!~|  .--|  |--| | | |
      |%|  |  |.'\      |===|  |--|%|  |  |
      |%|  |  |'\.'\      |  |  |__|  |  |  |
      |  |  |  |'\ \      |===|  |==|  |  |  |
      |  |  |__|  \.'\  |  |__|  |~~~~|__|
      |  |===|--|  \.'\|===|~|--|%|~~~~|--|
      A--A--'---A      '---'---A--A--A--'---'

--      --      -      -      -      -
\ \      / /  _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
\ \ / / / _ \ ' _ / _ \ / _ \ / _ \ / _ \ ' _ \
 \ \ v / ( _ ) | | | ( _ ) | | | ( _ ) | | | _ /
  \/_/_/ _ _ _/_/_ \_ _ _/_ _ _ _/_ _ _/_ _ _/_/_
Word Ladder autocomplete search using UCS, G-BFS, or A*
By: Maximilian Sulistiyono 13522061

Enter start word: coal
Enter end word: fire
Choose the Algorithm
1. UCS
2. G-BFS
3. A*
Choice: 1
Path found:
coal
foal
foam
form
firm
fire
Node traversed: 1637
Runtime: 3ms
Path Length: 6

```

## UCS Coal-> Fire GUI

WEEPER

Method

UCS

Start Word

coal

End Word

fire

Find!

C	O	A	L
F	O	A	L
F	O	A	M
F	O	R	M
F	I	R	M
F	I	R	E

Runtime: 3 ms

Nodes Traversed: 1037

Path Length: 6

## GBFS Coal-> Fire CLI

```

      .--.      .---.
      .---|__|      .-  |~~~|
      .--|===|--|_      |__| |~~~|--.
      | |===| |'\      .---!~| .--| |--| | | |
      |%%| | |.\      |===| |--|%%| | |
      |%%| | |.\.'\      | | |__| | | |
      | | | | \ \      |===| |==| | | |
      | | |__| \.'\ |      |__|__| |~~~|__|
      | |===|--| \.'\|===|~|--|%%|~~~|--|
      ^--^---'---^ \.'\---^--^--^---'---^

--      --      -      -
\ \      / /__ _ _ _ _| | | | _ _ _ _| |__| | _ _ _ _
\ \ / \ / / _ \ | ' _ / _` | | | / _` | / _` | / _` | \ ' _ |
  \ V V / ( ) | | | ( | | | |__| ( | | ( | | ( | | _ / |
    \_/ \ / \ _ _ / | | \ _ _ | | _ _ _ \ _ _ | \ _ _ | \ _ _ | _ |
Word Ladder autocomplete search using UCS, G-BFS, or A*
By: Maximilian Sulistiyo 13522061

Enter start word: coal
Enter end word: fire
Choose the Algorithm
1. UCS
2. G-BFS
3. A*
Choice: 2
No solution found
Node traversed: 9
Runtime: 0ms
```

## GBFS Coal-> Fire GUI

---

WEEPER

---

Method

GBFS

Start Word

coal

End Word

fire

Find!

C	O	A	L
?	?	?	?
F	I	R	E

Runtime: 0 ms

Nodes Traversed: 9

No Solution!

## A\* Coal-> Fire CLI

[illegible]

## A\* Coal-> Fire GUI

WEEPER

Method

A\*

Start Word

coal

End Word

fire

Find!

C	O	A	L
F	O	A	L
F	O	A	M
F	O	R	M
F	I	R	M
F	I	R	E

Runtime: 0 ms

Nodes Traversed: 17

Path Length: 6

## 5.6 Test Case 6 Seek -> Find

### UCS Seek -> Find CLI

```
Word Ladder autocomplete search using UCS, G-BFS, or A*
By: Maximilian Sulistiyo 13522061

Enter start word: seek
Enter end word: find
Choose the Algorithm
1. UCS
2. G-BFS
3. A*
Choice: 1
Path found:
seek
seed
send
fend
find
Node traversed: 1329
Runtime: 4ms
Path Length: 5
```

### UCS Seek -> Find GUI

WEEPER

Method  
UCS

Start Word  
seek

End Word  
find

Find!

S	E	E	K
S	E	E	D
S	E	N	D
F	E	N	D
F	I	N	D

Runtime: 0 ms  
Nodes Traversed: 1329  
Path Length: 5

## GBFS Seek -> Find CLI

## GBFS Seek -> Find GUI

 **WEEPER** 

## Method

GBFS 

### Start Word

seek

## End Word

find

Find!

S	E	E	K
S	E	E	D
F	E	E	D
F	E	N	D
F	I	N	D

Runtime: 0 ms

Nodes Traversed: 5

**Path Length: 5**

**A\* Seek -> Find CLI**

```

      .---.      .---.
      .---|__|      .-.      |~~~|
---|===|--|_      |_|      |~~~|--.
| |===| |'\      .---!~|  ---|  |--| | | |
|%%|  | |.\      |===| |--|%%|  | |
|%%|  | |.\'\      | |  |__|  | | |
| |  | | \ \ \      |===| ==|  | | |
| |  |__| \.\' |  |__|  |~~~|__|
| |===|--|  \.\'===|~|--|%%|~~~|--|
_ _ _ _ _ ' _ _ _ _ _ ' _ _ _ _ _ ' _ _ _ _ _ ' _ _ _ _ _ ' _ _ _ _ _ ' _ _ _ _ _ '
--      --      -      -      -      -
\\      // _ _ _ _ _ | | | | _ _ _ _ _ | _ _ _ _ _
 \ \ / / / \ | ' _ / _ ' | | | / _ ' / _ ' / _ ' \ _ _ |
  \ v v / ( ) | | | ( | | | | ( | ( | ( | ( | _ _ / |
   \ / \ / \ _ _ / | | \ _ _ / | _ _ _ \ _ _ / \ _ _ / \ _ _ /
Word Ladder autocomplete search using UCS, G-BFS, or A*
By: Maximilian Sulistiyo 13522061

Enter start word: seek
Enter end word: find
Choose the Algorithm
1. UCS
2. G-BFS
3. A*
Choice: 3
Path found:
seek
seed
feed
fend
find
Node traversed: 6
Runtime: 1ms
Path Length: 5

```

## A\* Seek -> Find GUI

**WEEPER**

Method

**Start Word**

seek

## End Word

find

Find!

S	E	E	K
S	E	E	D
F	E	E	D
F	E	N	D
F	I	N	D

Runtime: 0 ms

**Nodes Traversed: 6**

**Path Length: 5**

## BAB 6

### Analisis Solusi *UCS*, *GBFS*, dan *A\**

#### 6.1 Optimalitas

Dari percobaan dapat ditarik kesimpulan dari optimalitas masing masing algoritma. Terlihat dari kasus-kasus bahwa untuk algoritma *UCS* dan *A\** kompak dalam menghasilkan jawaban yang sama dan optimal. Namun untuk algoritma *GBFS* walaupun terdapat kasus dimana dia menghasilkan panjang path yang sama dengan *UCS* dan *A\**, dia juga memiliki kasus dimana tidak optimal seperti pada TC1 dan TC3. Kemudian dalam segi *completeness* juga terlihat bahwa terdapat kasus dimana *GBFS* tidak dapat menemukan solusi sementara dua lainnya dapat. Berdasarkan percobaan lanjutan yang dilakukan penulis, banyak kasus dimana hal ini terjadi.

WEPPER

Method  
UCS

Start Word  
lost

End Word  
soul

Find!

L	O	S	T
L	O	U	T
L	O	U	R
S	O	U	R
S	O	U	L

Runtime: 2 ms  
Nodes Traversed: 771  
Path Length: 5

WEPPER

Method  
GBFS

Start Word  
lost

End Word  
soul

Find!

L	O	S	T
L	O	U	T
P	O	U	T
P	O	U	R
S	O	U	R
S	O	U	L

Runtime: 0 ms  
Nodes Traversed: 9  
Path Length: 6

WEPPER

Method  
A\*

Start Word  
lost

End Word  
soul

Find!

L	O	S	T
L	O	U	T
L	O	U	R
S	O	U	R
S	O	U	L

Runtime: 0 ms  
Nodes Traversed: 12  
Path Length: 5



## 6.2 Waktu Eksekusi

Terlihat juga dari percobaan bahwa algoritma *UCS* memiliki waktu eksekusi paling buruk dari ketiganya dimana runtime dari *GBFS* dan *UCS* sangat cepat hingga waktu dalam satuan ms pun tidak dapat menangkap waktu eksekusi, penulis seharusnya menggunakan satuan waktu yang lebih akurat. Hal ini diakibatkan dengan algoritma *UCS* yang menjadi *BFS* pada kasus ini sehingga melakukan *exhaustive search* dimana pencarian menjadi memeriksa setiap node yang dibangkitkan hingga menemukan solusi. Beda halnya dengan algoritma *GBFS* dan *A\** yang menggunakan nilai *heuristic* sehingga pencarian pun menjadi “terarah” sehingga pencarian pun tidak melalui banyak node.

🧩 WEEPER 🧩

Method  
UCS

Start Word  
mind

End Word  
soul

Find!

M	I	N	D
M	I	L	D
M	I	L	L
M	A	L	L
M	A	U	L
S	A	U	L
S	O	U	L

Runtime: 10 ms  
Nodes Traversed: 3361  
Path Length: 7

🧩 WEEPER 🧩

Method  
A\*

Start Word  
mind

End Word  
soul

Find!

M	I	N	D
M	I	L	D
S	I	L	D
S	I	L	L
S	A	L	L
S	A	U	L
S	O	U	L

Runtime: 1 ms  
Nodes Traversed: 144  
Path Length: 7

### 6.3 Memori yang Dibutuhkan

Disini penulis mendefinisikan memori sebagai banyaknya node yang dilalui atau *node traversed*. Dalam hal ini juga terlihat bahwa algoritma *UCS* membutuhkan memori terbanyak dimana di rata-rata kasus algoritma ini dapat menggunakan ratusan hingga ribuan nodes. Berbeda halnya dengan algoritma *GBFS* dan *A\**, dengan menggunakan nilai *heuristic* maka pencarian pun bisa “terarah” sehingga tidak memeriksa semua node yang dapat dibangkitkan.

WEEPER

Method  
UCS

Start Word  
kiln

End Word  
bake

Find!

K	I	L	N
K	I	R	N
K	A	R	N
B	A	R	N
B	A	R	E
B	A	K	E

Runtime: 8 ms  
Nodes Traversed: 2072  
Path Length: 6

WEEPER

Method  
A\*

Start Word  
kiln

End Word  
bake

Find!

K	I	L	N
K	I	L	L
B	I	L	L
B	A	L	L
B	A	L	E
B	A	K	E

Runtime: 1 ms  
Nodes Traversed: 12  
Path Length: 6

## BAB 7

### Penjelasan Implementasi Bonus

#### 7.1 Kakas yang Digunakan

Penulis memutuskan untuk ~~menyiksa diri~~ menggunakan kakas web sebagai GUI, hal ini dikarenakan pengalaman penulis (Tubes 2 Stima) dalam menggunakan GUI ini. Framework yang digunakan disini adalah *React* dan CSS biasa untuk styling. Design yang digunakan terinspirasi dari website word weaver yang diberikan sebagai referensi game.

---

🎮 WEEPER 🎮

Method  
UCS

Start Word  
Enter word here

End Word  
Enter word here

Find!

---

🎮 WEEPER 🎮

Method  
UCS

Start Word  
cat

End Word  
dog

Find!

C	A	T
C	O	T
D	O	T
D	O	G

Runtime: 4 ms  
Nodes Traversed: 462  
Path Length: 4

## 7.2 Menghubungkan Frontend dan Backend

Untuk melakukan hal tersebut maka perlu dibuatkannya sebuah API. Dengan memanfaatkan library *springboot*, penulis pun berhasil dalam membuat API sederhana. Di dalam API ini terdapat kelas *SolverController* yang bertanggung jawab untuk memanggil *SolverService*. *SolverService* sendiri bertanggung jawab untuk memberikan service menggunakan kelas solver dimana pada solver terdapat algoritma penyelesaian. Kemudian FE dan BE berinteraksi pada `localhost/8080/api` jadi jika ingin digunakan, pastikan port tersebut sedang tidak digunakan proses lain.

## 7.3 Cara Penggunaan

Untuk penggunaan GUI ini maka penulis membuat 2 repository baru, satu untuk backend dan satu untuk frontend

Repository Frontend : [https://github.com/riyorax/Tucil3\\_13522061FE](https://github.com/riyorax/Tucil3_13522061FE)

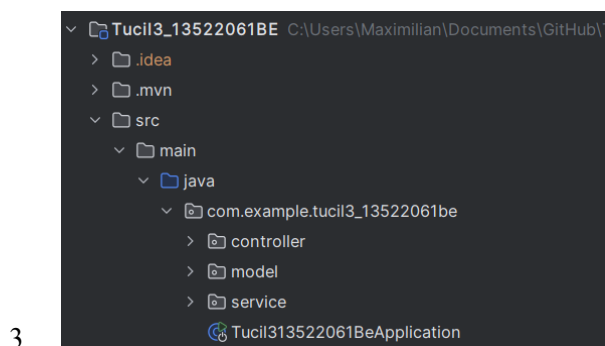
Untuk penggunaanya (terdapat juga pada README)

1. Pastikan npm sudah terdownload di PC anda
2. Clone repository dan ketik `npm install` untuk mengunduh dependencies
3. Ketik `npm run dev`

Repository Backend : [https://github.com/riyorax/Tucil3\\_13522061BE](https://github.com/riyorax/Tucil3_13522061BE)

Untuk penggunaannya

1. Download IDE IntelliJ pada pc anda, jika ingin mengambil versi ultimate, daftar lah dengan akun std
2. Buka IDE tersebut dan pilih new project dari repository, masukkan link repo backend.



Buka `Tucil313522061BeApplication`, kemudian IDE akan menginstall dependencies yang dibutuhkan. Jika tidak bisa coba click run pada kanan atas untuk memulai proses tersebut

4. Pastikan port 8080 tidak digunakan oleh proses lain

## Daftar Pustaka

1. <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf>
2. <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf>
3. [https://www.youtube.com/watch?v=h9iTnkgv05E&t=845s&ab\\_channel=NeetCode](https://www.youtube.com/watch?v=h9iTnkgv05E&t=845s&ab_channel=NeetCode)

## Lampiran

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

Link Repository : [https://github.com/riyorax/Tucil3\\_13522061](https://github.com/riyorax/Tucil3_13522061)