LAPORAN TUGAS KECIL III

IF2211 - STRATEGI ALGORITMA

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

Disusun oleh :

RAFFAEL BOYMIAN SIAHAAN

13522046

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

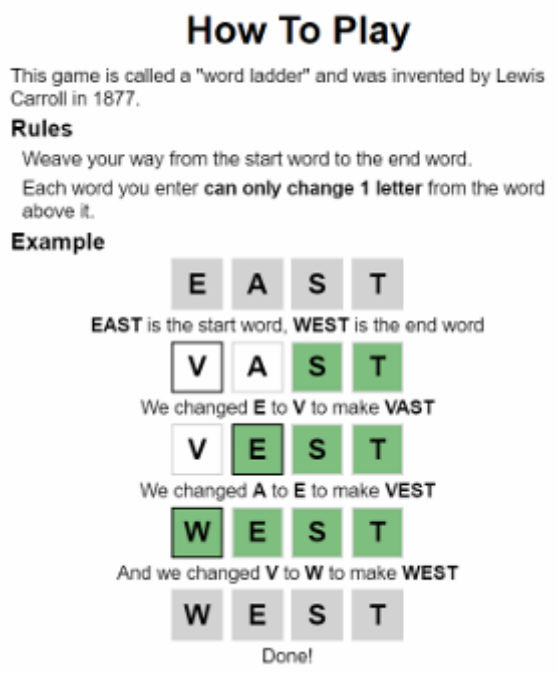
INSTITUT TEKNOLOGI BANDUNG

2024

DAFTAR ISI

BAB 1
DESKRIPSI MASALAH

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



Tugas kecil III ini bertujuan untuk membuat sebuah solver permainan tersebut dengan harapan kita dapat menemukan solusi paling optimal untuk menyelesaikan permainan *Word Ladder* ini. Program dibuat dalam bahasa Java berbasis CLI (Command Line Interface) atau menggunakan GUI apabila mengerjakan bonus. Program harus menemukan solusi permainan word ladder menggunakan algoritma UCS, Greedy Best First Search, dan A*.

BAB 2
TEORI SINGKAT

2.1 Algoritma *Uniform Cost Search* (UCS)

Algoritma Uniform Cost Search (UCS), juga dikenal sebagai algoritma Dijkstra yang dimodifikasi, adalah algoritma pencarian graf yang digunakan untuk menemukan jalur dengan biaya terendah dari simpul awal ke simpul tujuan. Algoritma ini terutama digunakan dalam graf berbobot di mana setiap tepi memiliki biaya atau bobot tertentu, dan tujuannya adalah untuk menemukan jalur total dengan biaya minimum.

UCS berbeda dengan Breadth-First Search (BFS) dan Iterative Deepening Search (IDS) yang umumnya menemukan jalur berdasarkan jumlah langkah terkecil. Dalam konteks BFS dan IDS, jumlah langkah adalah kriteria utama, yang cocok jika semua langkah memiliki biaya yang sama. Namun, jika biaya langkah berbeda, strategi ini tidak akan menghasilkan solusi optimal berdasarkan biaya total.

2.2 Algoritma *Greedy Best First Search* (G-BFS)

Algoritma Greedy Best First Search (GBFS) adalah algoritma pencarian yang menggunakan pendekatan serakah untuk bergerak melalui ruang pencarian. Algoritma ini memprioritaskan node yang tampak paling menjanjikan atau paling dekat dengan tujuan, berdasarkan fungsi heuristik tertentu. Dalam GBFS, setiap langkah memilih node yang memiliki nilai heuristik terkecil (atau terbesar, tergantung definisi masalah), di mana heuristik diharapkan mengindikasikan kedekatan ke tujuan. Heuristik harus merumuskan perkiraan biaya dari node saat ini ke tujuan tanpa memperhitungkan biaya yang telah dikeluarkan untuk mencapai node tersebut.

GBFS dapat sangat cepat dalam menemukan solusi karena langsung menuju tujuan yang tampak terbaik dan efisien dalam penggunaan memori karena hanya mempertahankan frontier pencarian yang minimal.

Namun, algoritma ini tidak selalu menemukan solusi optimal karena hanya fokus pada node yang tampak paling menjanjikan tanpa mempertimbangkan biaya keseluruhan dan dapat terjebak dalam minima lokal atau plateaus jika heuristik tidak sempurna.

2.3 Algoritma A*

Algoritma A* adalah algoritma pencarian yang memperhitungkan biaya terakumulasi dari node awal ke node saat ini ditambah dengan perkiraan biaya dari node saat ini ke tujuan (heuristik). Algoritma ini menggunakan fungsi $f(n) = g(n) + h(n)$, di mana:

- g(n) adalah biaya terakumulasi untuk mencapai node n dari node awal.
- h(n) adalah heuristik berupa perkiraan biaya dari node n ke tujuan.

A* dijamin akan menemukan solusi optimal jika heuristik yang digunakan adalah admissible (tidak pernah melebih-lebihkan biaya sebenarnya ke tujuan) serta sangat fleksibel dengan pemilihan heuristik, memungkinkan penyesuaian terhadap banyak jenis masalah.

Namun, algoritma ini dapat memerlukan banyak memori, terutama pada ruang pencarian yang luas, karena harus menyimpan semua node yang telah diekspan dan yang akan diekspan. Selain itu, performanya sangat bergantung pada kualitas heuristik. Heuristik yang buruk dapat menyebabkan pencarian yang tidak efisien.

BAB 3
ANALISIS & IMPLEMENTASI ALGORITMA

3.1 Implementasi Algoritma

Kode yang saya buat menggunakan pendekatan 3 algoritma, yaitu *Uniform Cost Search* (UCS)*, Greedy Best First Search,* dan A* untuk menemukan solusi paling optimal untuk menyelesaikan permainan *Word Ladder* (dalam konteks Tucil ini adalah mencari jalur terpendek berdasarkan pilihan salah satu algoritma).

Secara ekspansi kemungkinan jalur, ketiga algoritma tersebut memiliki cara kerja yang sama, tetapi hal yang membedakan ketiga algoritma tersebut adalah sebagai berikut.

1. Nilai cost dan heuristic untuk menentukan urutan ekspansi node, dimana UCS menggunakan nilai cost (g(n)), *Greedy Best First Search* menggunakan nilai heuristik (h(n)), serta A* menggunakan total dari cost dan heuristik (f(n) = g(n) + h(n)).
2. Pengaturan prioritas pada queue simpul ekspan, dimana dimana UCS memprioritaskan nilai cost terkecil, *Greedy Best First Search* memprioritaskan nilai heuristik terkecil, serta A* memprioritaskan total dari nilai cost dan heuristik.

Langkah-langkah utama dalam ketiga algoritma tersebut adalah sebagai berikut.

1. Inisialisasi node dan struktur data

- Node, disimpan dalam sebuah class dengan atribut string *currentWord* (kata di node saat ini), string *parent* (orang tua dari *currentWord*, berguna untuk merekonstruksi jalur pencarian serta lebih efisien alokasi memori), dan variabel tipe data int (UCS memiliki variabel cost bertipe integer, G-BFS memiliki variabel heuristic bertipe integer, dan A* memiliki variabel cost, heuristic, dan total bertipe integer) dan sebuah konstruktor Node.
- Sebuah priority queue, yang merupakan kumpulan simpul ekspan untuk menyimpan node yang ‘kelak’ akan dikunjungi. Prioritas urutan queue tergantung algoritma yang digunakan (dijelaskan secara spesifik di bawah).
- Sebuah set, yang merupakan kumpulan simpul hidup untuk menyimpan node yang sudah dikunjungi, sehingga algoritma tidak akan menjelajahi node yang sama.

2. Menambahkan node awal pada priority queue serta menambahkan kata awal (start word) ke dalam set of simpul hidup.
3. Selanjutnya, dilakukan proses pencarian menggunakan while-loop yang akan berjalan selama priority queue tidak kosong. Dalam loop tersebut, node dengan nilai terendah (nilai pada UCS adalah cost, G-BFS adalah heuristic, dan A* adalah cost + heuristic) akan di-pop dari queue.
4. Setelah node di-pop dari queue, algoritma akan memeriksa apakah kata pada node tersebut sama dengan kata tujuan (*end word*). Jika iya, maka program akan me-return jalur yang akan dibangun, banyaknya node yang dikunjungi, dan waktu eksekusi program.
5. Jika node saat ini bukan node tujuan, algoritma akan melakukan pencarian tetangga dari kata pada node saat ini yang berada dalam dictionary melalui sebuah fungsi (getNeighbors()). Setiap tetangga yang belum dikunjungi akan ditambahkan ke priority queue sebagai node baru dengan biaya yang diperbarui (UCS => node + 1, G-BFS => heuristik dari simpul n ke *end word*, A* => total dari *cost* dan nilai heuristik terbaru).
6. Jika priority queue kosong dan algoritma belum menemukan kata tujuan, algoritma akan mengembalikan null karena tidak ada jalur yang tersedia.

Asumsi yang ditambahkan pada langkah-langkah di atas adalah :

1. Dictionary dalam bentuk .txt beserta class untuk pemrosesan file tersebut sudah pasti terdefinisi.
2. *Start word* dan *end word* memiliki panjang yang sama (sudah dilakukan validasi sebelum masuk ke dalam fungsi algoritma).

3.2 Analisis Algoritma

Berdasarkan spesifikasi tugas besar, terdapat beberapa pertanyaan yang harus dijawab seputar algoritma yang digunakan. Pertanyaan-pertanyaan tersebut akan dijawab sebagai berikut.

1. Definisi dari f(n) dan g(n), sesuai dengan salindia kuliah

Menurut salindia kuliah, - **f(n)** adalah fungsi evaluasi atau nilai yang diberikan kepada simpul n. nilai

tersebut berupa estimasi total biaya dari simpul awal ke simpul tujuan melewati simpul n, yang merupakan penjumlahan dari **g(n)** dan **h(n)**,

- **g(n)** adalah biaya jalur (path cost) yang sudah diketahui dari simpul awal ke simpul n.
- **h(n)** adalah nilai heuristik yang digunakan untuk memprediksi biaya yang tersisa dalam perjalanan ke tujuan dari simpul n.

2. Apakah heuristik yang digunakan pada algoritma A* *admissible*? Jelaskan sesuai definisi *admissible* dari salindia kuliah.

Suatu fungsi heuristik dikatakan admissible apabila tidak melebih-lebihkan (*overestimate*) estimasi biaya untuk mencapai goal state pada state saat ini.
Fungsi heuristik yang digunakan dalam algoritma ini adalah *hamming distance*, dimana *hamming distance* dalam konteks *word ladder* mengukur jumlah posisi huruf yang berbeda antara dua kata. Misalnya, dari kata "cold" ke "warm", Hamming distance adalah 4 karena semua huruf berbeda.
Dalam hal ini, *Hamming distance* tidak akan pernah mengestimasi lebih dari jumlah langkah minimal yang diperlukan untuk merubah satu kata menjadi kata lain, karena setiap perubahan satu huruf merupakan satu langkah yang valid dalam *word ladder.* Oleh karena itu, Hamming distance adalah heuristik yang *admissible* untuk *word ladder* solver.

3. Pada kasus *word ladder*, apakah algoritma UCS sama dengan BFS? (dalam artian urutan node yang dibangkitkan dan path yang dihasilkan sama)

Dengan mengasumsikan bahwa setiap langkah dari satu kata ke kata berikutnya dianggap memiliki cost (biaya) yang sama, yaitu 1, maka UCS akan memiliki operasi yang sangat mirip, bahkan sama dengan BFS.
Jika biaya untuk setiap langkah adalah sama, kedua algoritma akan menghasilkan node dalam urutan yang sama, karena kedua algoritma akan memperluas node terdekat dengan sumber terlebih dahulu.
Kedua algoritma akan menghasilkan path yang sama dalam kasus biaya konstan per langkah, karena kedua algoritma akan menemukan jalur terpendek pertama yang

mungkin, asalkan tidak ada varian tambahan dalam biaya perubahan kata.
Oleh karena itu, dalam konteks *word ladder* dimana setiap transisi memiliki biaya yang sama, UCS dan BFS bisa dikatakan sama dalam hal urutan node yang dibangkitkan dan path yang dihasilkan, lain konteks apabila cost berbeda untuk setiap perpindahan

4. Secara teoritis, apakah algoritma A* lebih efisien dibandingkan dengan algoritma UCS pada kasus word *ladder*?

Secara teoritis, algoritma A* diharapkan lebih efisien dibandingkan UCS, terutama jika menggunakan heuristik yang admissible.
Heuristik ini membantu A* dalam memprioritaskan node-node yang lebih mungkin mendekatkan ke tujuan, berbeda dengan UCS yang hanya mempertimbangkan biaya yang telah terakumulasi dari node awal ke node saat ini tanpa memperhatikan jaraknya ke tujuan.
Karena A* mempertimbangkan kedua aspek biaya sejauh ini (g(n)) dan perkiraan biaya ke tujuan (h(n)), algoritma ini mampu mengurangi ruang pencarian dengan tidak mengexplore node yang tidak efisien menuju tujuan, yang mana UCS tidak memiliki informasi tentang "arah" menuju tujuan dan oleh karena itu bisa menghabiskan waktu lebih banyak untuk menjelajahi area yang tidak relevan dengan solusi.
Sehingga dapat disimpulkan bahwa dengan asumsi bahwa setiap langkah memiliki biaya yang sama dan heuristik yang digunakan admissible (seperti *hamming distance*), A* tidak hanya akan menemukan solusi yang efisien tetapi juga melakukannya dengan lebih cepat dan lebih hemat memori dibandingkan dengan UCS, yang tidak menggunakan informasi heuristik untuk mengarahkan pencariannya.

5. Secara teoritis, apakah algoritma *Greedy Best First Search* menjamin solusi optimal untuk persoalan *word ladder*?

Tidak. Algoritma tersebut tidak menjamin solusi optimal untuk persoalan tersebut karena alasan berikut.

1. **Terjebak pada minimal lokal atau plateau.**

   Dalam konteks word ladder, minima lokal dapat terjadi ketika sebuah kata tampak dekat dengan kata tujuan berdasarkan heuristik yang digunakan (misalnya, Hamming distance), tetapi tidak ada langkah selanjutnya yang mendekatkan lebih lagi atau bahkan mungkin menjauhkan dari tujuan. Plateau adalah situasi di mana semua operasi yang mungkin dari suatu titik memberikan nilai heuristik yang sama, sehingga tidak ada kemajuan menuju tujuan.

2. **Irreversibilitas (tidak dapat dibalikkan atau diubah)**

   Setelah node dipilih untuk dieksplorasi, keputusan tersebut tidak dapat dibatalkan atau diubah meskipun mungkin saja keputusan tersebut tidak mengarah pada solusi yang optimal. Hal tersebut dapat membatasi kemampuan algoritma untuk menavigasi dengan efektif di ruang pencarian yang kompleks atau ketika heuristik yang digunakan tidak memberikan informasi yang cukup untuk membuat keputusan yang tepat.

BAB 4
SOURCE CODE PROGRAM

Pseudocode sederhana versi saya untuk potongan program yang berfokus pada algoritma *brute force* adalah sebagai berikut:

UCS.java

```
package backend;

import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Set;

public class UCS {

    public static class Node {
        String currentWord;
        Node parent;
        int cost;

        Node(String currentWord, Node parent, int cost) {
            this.currentWord = currentWord;
            this.parent = parent;
            this.cost = cost;
        }
    }

    public static Utility.Result ucs(String start, String goal,
Set<String> dict) {
        PriorityQueue<Node> queue = new
PriorityQueue<>(Comparator.comparingInt(node -> node.cost));
        Set<String> visited = new HashSet<>();
        queue.add(new Node(start, null, 0));
        visited.add(start);
        int count = 0;

        long startTime = System.nanoTime();

        while (!queue.isEmpty()) {
            Node head = queue.poll();
            count++;

            // Jika tujuan telah tercapai
            if (head.currentWord.equals(goal)) {
                long endTime = System.nanoTime();
                List<String> path = Utility.makeUCSPath(head);
                return new Utility.Result(path, count, (endTime -
startTime) / 1000000);
            }
```

```
            for (String neighbor :
Utility.getNeighbors(head.currentWord, dict)) {
                if (!visited.contains(neighbor)) {
                    queue.add(new Node(neighbor, head, head.cost +
1));
                    visited.add(neighbor);
                }
            }
        }
        return new Utility.Result(null, count, (System.nanoTime() -
startTime) / 1000000);
    }
}
```

Class UCS

Method

Nama	Tipe	Parameter	Deskripsi
`ucs`	`Utility.Result`	`String start, String goal, Set<String> dict`	Menjalankan Uniform Cost Search dari node start ke node goal menggunakan kamus dict sebagai daftar kata yang bisa dijelajahi. Mengembalikan Utility.Result yang mengandung path yang ditemukan, jumlah node yang dikunjungi, dan waktu eksekusi dalam milidetik.

Sub Class Node

Attribute

Nama	Tipe	Deskripsi
`currentWord`	`String`	Kata saat ini yang direpresentasikan oleh node.
`Parent`	`Node`	Node induk dalam jalur pencarian UCS.
`Cost`	`Int`	Biaya dari start node ke

		node saat ini.

Constructor

Nama	Tipe	Parameter	Deskripsi
`Node`	`None`	`String currentWord, Node parent, int cost`	Membuat instance baru dari kelas Node dengan kata tertentu, node induk, dan biaya untuk mencapai node ini.

GBFS.java

```
package backend;

import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Set;

public class GBFS {

    public static class Node {
        String currentWord;
        Node parent;
        int heuristic;

        Node(String currentWord, Node parent, int heuristic) {
            this.currentWord = currentWord;
            this.parent = parent;
            this.heuristic = heuristic;
        }
    }

    public static Utility.Result gbfs(String start, String goal,
Set<String> dict) {
        // Prioritas berdasarkan nilai heuristik terendah
        PriorityQueue<Node> queue = new
PriorityQueue<>(Comparator.comparingInt(node -> node.heuristic));
        Set<String> visited = new HashSet<>();
        queue.add(new Node(start, null,
Heuristic.hammingDistance(start, goal)));
        visited.add(start);
        int count = 0;

        long startTime = System.nanoTime();

        while (!queue.isEmpty()) {
            Node head = queue.poll();
            count++;
```

```
            System.out.println(head.currentWord);

            // Jika tujuan ditemukan
            if (head.currentWord.equals(goal)) {
                long endTime = System.nanoTime();
                List<String> path = Utility.makeGBFSPath(head);
                return new Utility.Result(path, count, (endTime -
startTime) / 1000000);
            }

            for (String neighbor :
Utility.getNeighbors(head.currentWord, dict)) {
                int heuristic = Heuristic.hammingDistance(neighbor,
goal);

                if (!visited.contains(neighbor)) {
                    queue.add(new Node(neighbor, head, heuristic));
                    visited.add(neighbor);
                }
            }
        }

        return new Utility.Result(null, count, (System.nanoTime() -
startTime) / 1000000);
    }
}
```

Class GBFS

Method

Nama	Tipe	Parameter	Deskripsi
`gbfs`	`Utility.Result`	`String start, String goal, Set<String> dict`	Menjalankan *Greedy Best First Search* dari node start ke node goal menggunakan kamus dict sebagai daftar kata yang bisa dijelajahi. Mengembalikan Utility.Result yang mengandung path yang ditemukan, jumlah node yang dikunjungi, dan waktu eksekusi dalam milidetik.

Sub Class Node

Attribute

Nama	Tipe	Deskripsi
`currentWord`	`String`	Kata saat ini yang direpresentasikan oleh node.
`Parent`	`Node`	Node induk dalam jalur pencarian UCS.
`Heuristic`	`Int`	Nilai heuristik dari node saat ini ke node tujuan.

Constructor

Nama	Tipe	Parameter	Deskripsi
`Node`	`None`	`String currentWord, Node parent, int heuristic`	Membuat instance baru dari kelas Node dengan kata tertentu, node induk, dan nilai heuristik.

Astar.java

```
package backend;

import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Set;

public class AStar {

    public static class Node {
        String currentWord;
        Node parent;
        int cost;
        int heuristic;
        int total;

        Node(String currentWord, Node parent, int cost, int
heuristic) {
            this.currentWord = currentWord;
            this.parent = parent;
            this.cost = cost;
            this.heuristic = heuristic;
            this.total = cost + heuristic;
        }
    }

    public static Utility.Result a_star(String start, String goal,
```

```
Set<String> dict) {
        PriorityQueue<Node> queue = new
PriorityQueue<>(Comparator.comparingInt(node -> node.total));
        Set<String> visited = new HashSet<>();
        queue.add(new Node(start, null, 0,
Heuristic.hammingDistance(start, goal)));
        visited.add(start);
        int count = 0;

        long startTime = System.nanoTime();

        while (!queue.isEmpty()) {
            Node head = queue.poll();
            count++;

            if (head.currentWord.equals(goal)) {
                long endTime = System.nanoTime();
                List<String> path = Utility.makeASPath(head);
                return new Utility.Result(path, count, (endTime -
startTime) / 1000000);
            }

            for (String neighbor :
Utility.getNeighbors(head.currentWord, dict)) {
                int cost = head.cost + 1;
                int heuristic = Heuristic.hammingDistance(neighbor,
goal);

                if (!visited.contains(neighbor)) {
                    queue.add(new Node(neighbor, head, cost,
heuristic));
                    visited.add(neighbor);
                }
            }
        }

        return new Utility.Result(null, count, (System.nanoTime() -
startTime) / 1000000);
    }
}
```

Class UCS

Method

Nama	Tipe	Parameter	Deskripsi
`getNeighbors`	`List<String>`	`String word, set<String> dict`	Mendapatkan tetangga dari currentWord yang ada dalam kamus dict.
`makeUCSPath`	`List<String>`	`UCS.Node`	Membangun jalur dari node untuk

			algoritma UCS.
`makeGBFSPath`	`List<String>`	`GBFS.Node`	Membangun jalur dari node untuk algoritma GBFS.
`makeASPath`	`List<String>`	`AStar.Node`	Membangun jalur dari node untuk algoritma A*.

Sub Class Node

Attribute

Nama	Tipe	Deskripsi
`currentWord`	`String`	Kata saat ini yang direpresentasikan oleh node.
`Parent`	`Node`	Node induk dalam jalur pencarian UCS.
`Cost`	`Int`	Biaya dari start node ke node saat ini.
`Heuristic`	`int`	Nilai heuristik dari node saat ini ke node tujuan.
`Total`	`int`	Total dari cost dan nilai heuristik.

Constructor

Nama	Tipe	Parameter	Deskripsi
`Node`	`None`	`String currentWord, Node parent, int cost, int heuristic`	Membuat instance baru dari kelas Node dengan kata tertentu, node induk, biaya untuk mencapai node ini, nilai heuristik, serta menjumlahkan kedua attribute tersebut ke attribute total.

Utility.java

```
package backend;

import java.util.ArrayList;
import java.util.List;
import java.util.Set;

public class Utility {

    public static List<String> getNeighbors(String word, Set<String>
dict) {
        List<String> neighbors = new ArrayList<>();
        char[] chars = word.toCharArray();

        for (int i = 0; i < chars.length; i++) {
            char oldChar = chars[i];
            for (char c = 'A'; c <= 'Z'; c++) {
                if (c != oldChar) {
                    chars[i] = c;
                    String newWord = new String(chars);
                    if (dict.contains(newWord)) {
                        neighbors.add(newWord);
                    }
                }
            }
            chars[i] = oldChar;
        }
        return neighbors;
    }

    public static List<String> makeUCSPath(UCS.Node node) {
        List<String> path = new ArrayList<>();
        while (node != null) {
            path.add(0, node.currentWord);
            node = node.parent;
        }
        return path;
    }

    public static List<String> makeGBFSPath(GBFS.Node node) {
        List<String> path = new ArrayList<>();
        while (node != null) {
            path.add(0, node.currentWord);
            node = node.parent;
        }
        return path;
    }

    public static List<String> makeASPath(AStar.Node node) {
        List<String> path = new ArrayList<>();
        while (node != null) {
            path.add(0, node.currentWord);
            node = node.parent;
        }
        return path;
```

```
    }

    public static class Result {
        private List<String> path;
        private int nodesVisited;
        private long timeTakenNano;

        public Result(List<String> path, int nodesVisited, long
timeTakenNano) {
            this.path = path;
            this.nodesVisited = nodesVisited;
            this.timeTakenNano = timeTakenNano;
        }

        public int getNodesVisited() {
            return this.nodesVisited;
        }

        public List<String> getPath() {
            return this.path;
        }

        public long getTime() {
            return this.timeTakenNano;
        }
    }
}
```

Class Utility

Method

Nama	Tipe	Parameter	Deskripsi
`getPath`	`List<String>`		Mengembalikan path (jalur).
`getNodesVisited`	`int`		Mengembalikan jumlah node yang dikunjungi.
`getTime`	`long`		Mengembalikan waktu dalam nanodetik (yang kemudian akan dibagi 10^6)

Sub Class Result

Attribute

Nama	Tipe	Deskripsi
`path`	`List<String>`	Jalur yang ditemukan..
`nodesVisited`	`int`	Jumlah node yang dikunjungi
`timeTakenNano`	`long`	Waktu yang dibutuhkan dalam nanodetik.

Constructor

Nama	Tipe	Parameter	Deskripsi
`Result`	`None`	`List<String> path, int nodesVisited, long timeTakenNano`	Membuat instance baru dari kelas Result.

Method

Nama	Tipe	Parameter	Deskripsi
`getPath`	`List<String>`		Mengembalikan path (jalur).
`getNodesVisited`	`int`		Mengembalikan jumlah node yang dikunjungi.
`getTime`	`long`		Mengembalikan waktu dalam nanodetik (yang kemudian akan dibagi 10^6)

Heuristic.java

```
package backend;

public class Heuristic {
    public static int hammingDistance(String s1, String s2) {
        int count = 0;

        for (int i = 0; i < s1.length(); i++) {
            if (s1.charAt(i) != s2.charAt(i)) {
                count++;
```

```
            }
        }

        return count;
    }
}
```

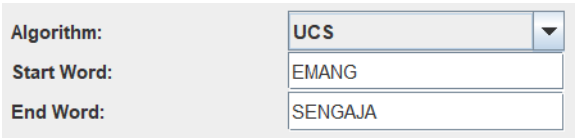
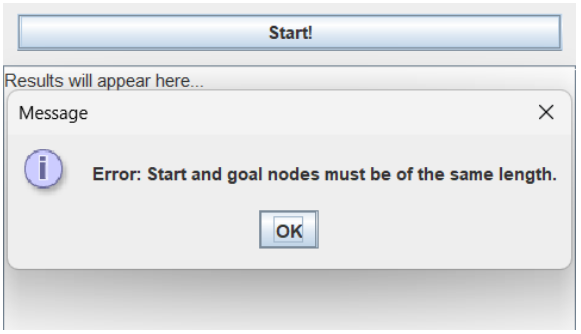
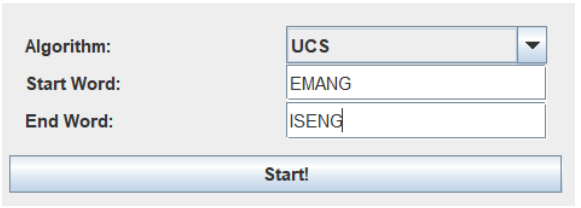
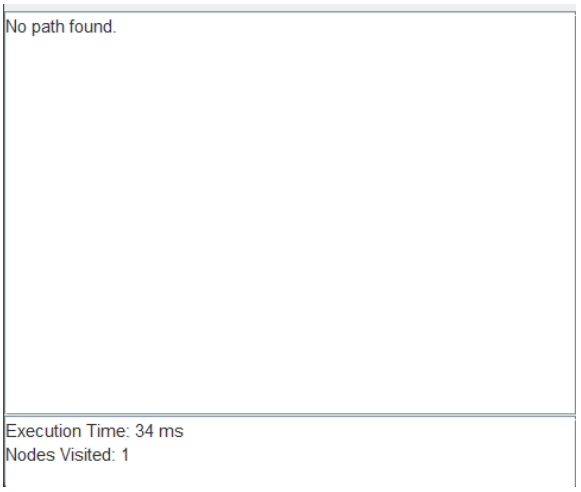
class Heuristic

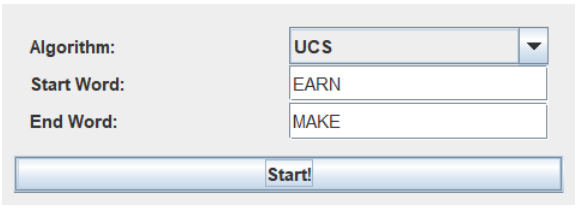
Method

Nama	Tipe	Parameter	Deskripsi
`hammingDistance`	`int`	`String s1, String s2`	mengembalikan jumlah huruf yang berbeda dari dua string. Method ini digunakan untuk menghitung nilai heuristik.

BAB 5
TESTING PROGRAM

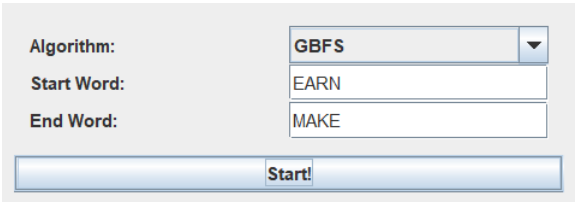
5.1 Test Case

Test Case	Result
	
	



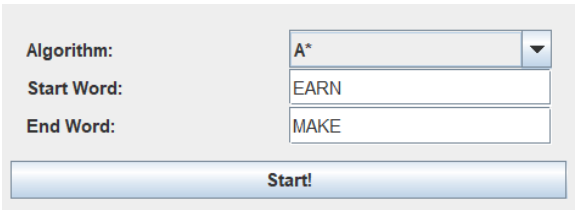
E	A	R	N
C	A	R	N
C	A	R	E
C	A	K	E
M	A	K	E

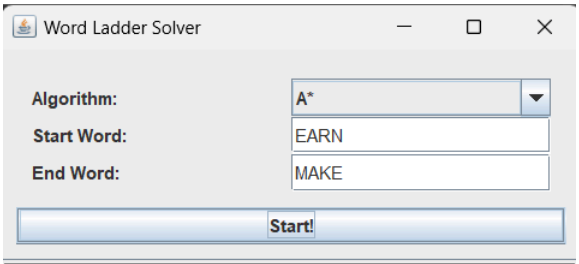
Path Length: 5
Execution Time: 16 ms
Nodes Visited: 1394



E	A	R	N
B	A	R	N
B	A	R	E
M	A	R	E
M	A	K	E

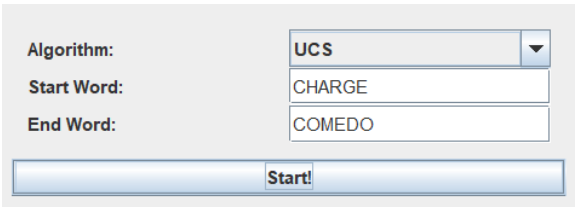
Path Length: 5
Execution Time: 2 ms
Nodes Visited: 5





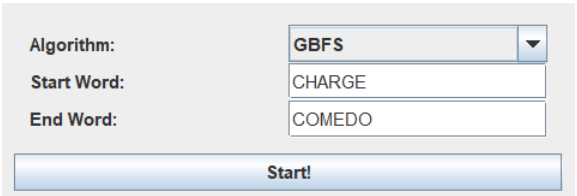
E	A	R	N
C	A	R	N
C	A	R	E
M	A	R	E
M	A	K	E

Path Length: 5
Execution Time: 0 ms
Nodes Visited: 16



C	H	A	R	G	E
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	I	N	E	R
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

Path Length: 23
Execution Time: 136 ms
Nodes Visited: 8712



C	H	A	R	G	E
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	I	L	E	D
C	O	O	L	E	D

	C O O E E D C O O E E S C O O E R S C O D E R S C O D E N S C O D O N S C O L O N 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 Path Length: 28 Execution Time: 34 ms Nodes Visited: 250
Algorithm: A* Start Word: CHARGE End Word: COMEDO Start!	C H A R G E 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 Path Length: 23 Execution Time: 84 ms Nodes Visited: 7439

5.2 Hasil analisis perbandingan solusi UCS, Greedy Best First Search, dan A*

Test case yang dibuat mencakup 4 kondisi, yaitu.

1. panjang start word dan end word berbeda.
2. start word dan end word tidak terdefinisi pada kamus atau jalur tidak ditemukan.
3. start word dan end word dengan panjang kata yang sedikit dan jalur yang tidak begitu panjang.
4. start word dan end word dengan panjang kata yang cukup panjang dan jalur yang cukup panjang.

Untuk mempermudah analisis, analisis akan dibagi menjadi beberapa sub-analisis, yaitu.

1. Optimalitas

   UCS

E	A	R	N
C	A	R	N
C	A	R	E
C	A	K	E
M	A	K	E

   panjang jalur : 5

   GBFS

E	A	R	N
B	A	R	N
B	A	R	E
M	A	R	E
M	A	K	E

panjang jalur : 5

A*

E	A	R	N
C	A	R	N
C	A	R	E
M	A	R	E
M	A	K	E

panjang jalur : 5

Dalam kasus ini : Path Length UCS = Path Length GBFS = Path Length A*

Namun, dalam kasus "CHARGE" ke "KOMEDO" (*screenshot tidak dilampirkan karena akan sangat panjang*), *Path Length* GBFS (28) > *Path Length* UCS (23) = *Path Length* A* (23)

Berdasarkan kasus tersebut, dapat disimpulkan bahwa,

- UCS menjamin untuk menemukan solusi optimal dalam semua kasus. Algoritma ini akan menjelajahi semua jalur secara merata berdasarkan biaya (*cost*) dari node awal hingga node saat ini tanpa mempertimbangkan estimasi ke tujuan.
- GBFS mungkin bisa menghasilkan solusi yang optimal dalam ruang pencarian berskala kecil. Namun, algoritma ini tidak menjamin solusi optimal karena hanya fokus pada node yang memiliki heuristik terbaik menuju tujuan. Hal ini dapat menyebabkan algoritma mengabaikan jalur yang lebih pendek atau lebih murah karena tidak mempertimbangkan *cost* sama sekali. Buktinya, dapat

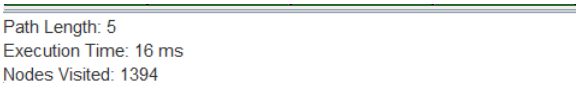
dilihat pada ruang pencarian yang besar, path length dari algoritma ini cukup jauh melebihi dua algoritma lainnya (UCS an A*)

- A* dijamin menemukan solusi optimal asalkan heuristik yang digunakan *admissible*. Dengan menggabungkan total biaya perjalanan dengan estimasi biaya ke tujuan, A* secara efektif mengatur pencarian untuk mengoptimalkan kedua aspek tersebut.

2. Waktu Eksekusi

   Untuk kasus "EARN" ke "MAKE", berikut adalah waktu eksekusi dari kasus tersebut.

   UCS

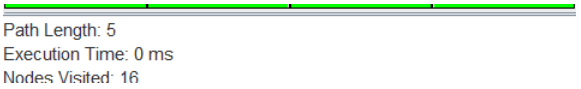
   waktu eksekusi : 16 ms

   GBFS

   waktu eksekusi : 2 ms

   A*

   waktu eksekusi : 0 < t < 1 ms

   Dalam kasus ini : *Execution Time* UCS > *Execution Time* GBFS > *Execution Time* A*

   Namun, kasus "CHARGE" ke "COMEDO" menghasilkan kesimpulan yang berbeda, yaitu Execution Time UCS (136 ms) > Execution Time A* (84 ms) > Execution Time GBFS (34 ms). Bukti *screenshot* dapat dilihat di bawah.

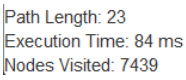
   UCS

   GBFS

   A*



Berdasarkan kasus tersebut, dapat disimpulkan bahwa,

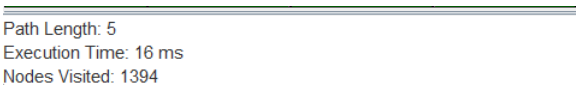
- UCS membutuhkan waktu eksekusi yang lama, terutama dalam ruang pencarian yang besar, karena harus menjelajahi semua node berdasarkan biaya terakumulasi tanpa arahan tentang mana yang lebih mungkin menuju tujuan.
- GBFS memiliki waktu eksekusi yang cenderung cepat karena langsung mengarah ke tujuan berdasarkan heuristik, sangat terlihat ketika ruang pencariannya cukup besar.
- A* cenderung memiliki waktu eksekusi yang lebih efisien daripada UCS jika heuristik yang digunakan baik dan relevan dengan masalahnya (dalam konteks ini adalah *hamming distance*).

3. Memori yang dibutuhkan

   Penggunaan memori pada kasus *word ladder solver* dapat dilihat berdasarkan node yang dikunjungi/diekspan.

   Untuk kasus “EARN” ke “MAKE”, berikut adalah jumlah node yang dikunjungi dari kasus tersebut.

   UCS

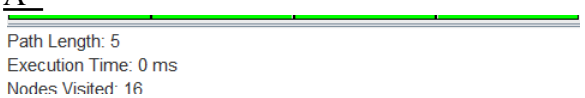


   Node yang dikunjungi : 1394

   GBFS

   Node yang dikunjungi : 5

   A*



   Node yang dikunjungi : 16

   Dalam kasus ini : Nodes Visited UCS > Nodes Visited A* > Nodes Visited GBFS dengan gap antara UCS dan A* yang cukup jauh (cenderung mendekati GBFS).

   Pada kasus “CHARGE” ke “COMEDO”,

UCS

Node yang dikunjungi : 8712

GBFS

Node yang dikunjungi : 250

A*

Node yang dikunjungi : 7439

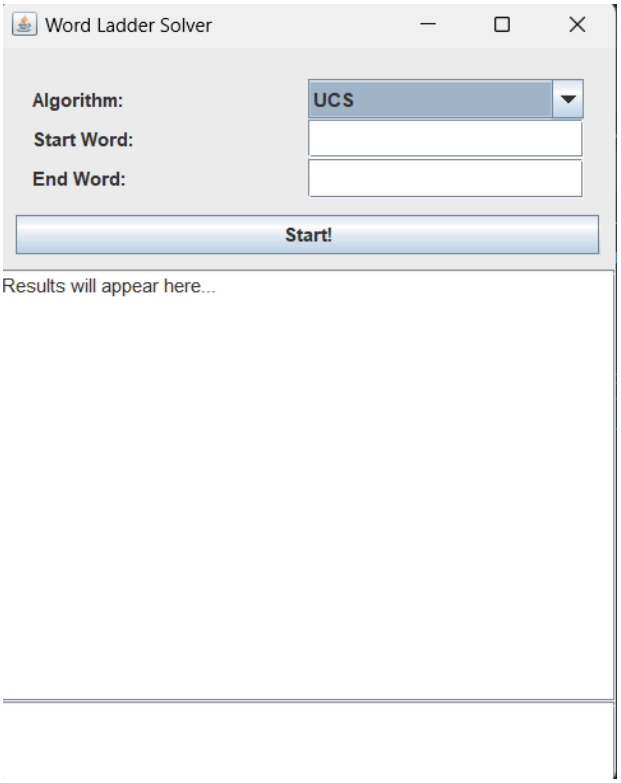
Dalam kasus ini : Nodes Visited UCS > Nodes Visited A* > Nodes Visited GBFS dengan gap antara UCS dan A* yang cukup dekat.

Berdasarkan kasus tersebut, dapat disimpulkan bahwa,

- UCS dapat mengonsumsi jumlah memori yang signifikan (cenderung boros dan tidak efisien), karena menyimpan semua node yang diperluas dalam *frontier* untuk memastikan pencarian yang lengkap dan optimal.
- GBFS memiliki keuntungan dalam penggunaan memori yang lebih rendah karena hanya perlu mempertahankan set node yang "paling menjanjikan" pada setiap langkah. namun, ini bisa bermasalah jika banyak node memiliki heuristik yang sama atau sangat serupa.
- A* umumnya membutuhkan lebih banyak memori, namun tidak sebanyak UCS karena menyimpan semua node dalam frontier untuk memastikan bahwa jalur optimal dapat diidentifikasi. Penggunaan memori A* dapat menjadi masalah dalam ruang pencarian yang sangat besar atau kompleks.

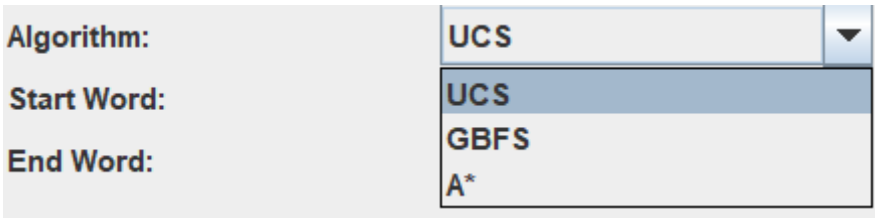
5.3 Implementasi Bonus

Implementasi GUI sederhana disimpan dalam SearchAlgorithmGUI.java dan WordGrid.java. Berikut adalah tampilan dari GUI yang dibuat.

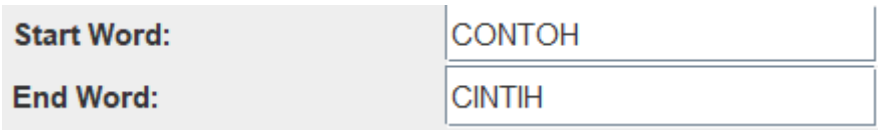


Fitur-fitur yang terdapat pada GUI adalah sebagai berikut.

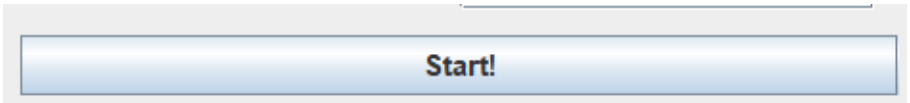
1. pilihan jenis algoritma (UCS, GBFS, atau A*)



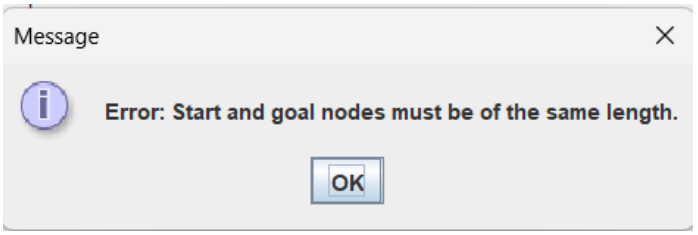
2. Start Word dan End Word (input harus berupa huruf kapital)



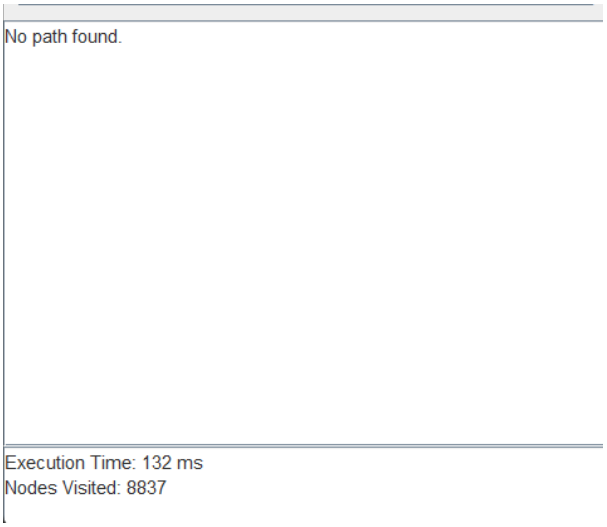
3. Tombol start



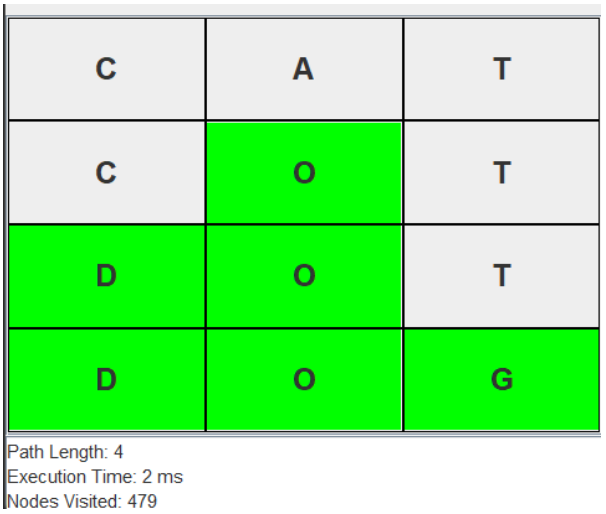
4. Hasil berupa jalur, waktu eksekusi, dan node yang dikunjungi.
    - Apabila panjang kata dari start word dan end word berbeda, akan me-return pop-up message seperti gambar di bawah.



- Apabila jalur tidak ditemukan, akan me-return seperti pada gambar di bawah.



- Apabila jalur ditemukan, akan me-return seperti pada gambar di bawah.



BAB 6
KESIMPULAN

5.1 Kesimpulan

Berdasarkan analisis yang telah dilakukan, dapat disimpulkan bahwa A* adalah algoritma yang paling efisien jika heuristik yang *admissible* (dalam konteks ini adalah *hamming distance*), karena menawarkan keseimbangan yang baik antara kecepatan waktu eksekusi, efisiensi memori, dan terutama, optimalitas solusi. GBFS bisa sangat cepat dan hemat memori tetapi berisiko tidak menemukan solusi yang optimal atau terjebak pada kondisi yang tidak menguntungkan (misal minima lokal atau plateaus). UCS, meskipun optimal, bisa menjadi lambat dan tidak praktis dalam ruang pencarian yang besar karena sifatnya yang melakukan ekspansi besar-besaran (dalam konteks ini layaknya BFS).

5.2 Saran

Saya merasa program yang diselesaikan masih jauh dari kata sempurna. Untuk kedepannya, masih banyak pengembangan yang dapat dilakukan untuk memaksimalkan program ini. Dengan memperhatikan fungsi, memanfaatkan kegunaan untuk hal lain, memperbaiki apa yang masih kurang, dan eksekusi yang lebih baik dalam tugas berikutnya.

5.3 Komentar

No komen sih, tapi saya mengucapkan terima kasih kepada asisten-asisten yang bertanggung jawab terhadap pelaksanaan tugas kecil ini. Ditunggu tugas besar III nya hehe

5.4 Refleksi

Tugas kecil terakhir ini telah memberikan pengalaman berharga bagi saya atas situasi dan tantangan yang muncul. Salah satu kendala yang saya hadapi adalah alokasi waktu pengerjaan yang kurang efisien dan efektif karena bertabrakan dengan tugas besar/*milestone* matkul lain dan kuis-kuis yang ada. Selain itu, target yang ingin dicapai juga menjadi tantangan tersendiri, dan akhirnya saya berhasil mengimplementasikan bonus GUI! HORAYYYY! Selain itu Setelah mengerjakan tugas kecil perdana ini, saya semakin *aware* dengan *time management* dan *planning* yang cukup matang selama mengerjakan tugas-tugas kedepannya yang kini semakin banyak dan semakin gila. Saya percaya bahwa segala bentuk

kerja keras dan *struggle* selama mengerjakan tugas ini dapat memotivasi diri saya untuk jauh lebih maksimal di tugas-tugas berikutnya di Teknik Informatika ITB.

REFERENSI

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

LAMPIRAN

Link Repository GitHub :
https://github.com/slntkllr01/Tucil3_13522046

Progress Tracking

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