LAPORAN TUGAS KECIL
STRATEGI ALGORITMA
IF2211

Disusun Oleh:

Raden Rafly Hanggaraksa Budiarto

13522014

I. DAFTAR ISI

II. 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.

Gambar 1 Cara bermain word ladder

III. IMPLEMENTASI ALGORITMA

A. UNIFORM COST SEARCH

Uniform Cost Search merupakan teknik penelusuran graf dengan memperhatikan harga dimulai dari simpul awal. Setiap hasil ekspansi akan dimasukkan ke dalam *priority queue* sesuai dengan harga / jarak kumulatif dari simpul tersebut. Harga pada *priority queue* diurutkan dari terendah hingga terbesar sehingga menyebabkan penelusuran secara Uniform Cost Search selalu memberikan hasil yang optimal. Pada problem ini, harga ditentukan dari jarak tempuh yang dilalui kata awal menuju kata yang sedang diekspansi. Penelusuran ini memiliki kompleksitas waktu $O(b^d)$ dengan b adalah simpul ekspansi maksimum dari suatu simpul dan d merupakan kedalaman simpul tujuan terendah serta kompleksitas ruang dari penelusuran ini serupa dengan kompleksitas waktunya.

Implementasi algoritma Uniform Cost Search pada program kami (ditulis dalam bahasa pemrograman java) dilakukan dengan langkah – langkah sebagai berikut.

1. Kata pertama dimasukkan, di cek apakah kata tersebut merupakan kata tujuan atau tidak. Jika tidak, lanjut.
2. Kata akan diekspansi dengan seluruh kata yang ada pada kamus / *dictionary* yang hanya memiliki satu perbedaan huruf saja. Harga simpul dari ekspansi ini merupakan banyak kata yang dilalui untuk menempuh setiap kata pada hasil ekspansi. Hasil ekspansi akan dimasukkan ke dalam *priority queue* dengan nilai harga terurut menaik.
3. Elemen pertama dari *priority queue* akan dilepas dan dicek apakah merupakan kata tujuan. Jika bukan, lakukan ekspansi kembali.
4. Ekspansi terus dilakukan hingga ditemukan solusi atau elemen dari Priority Queue sudah habis. Apabila elemen habis, maka tidak ditemukan solusinya.

B. GREEDY BEST FIRST SEARCH

Greedy Best First Search merupakan teknik penelusuran graf dengan memilih simpul yang merupakan pilihan terbaik berdasarkan suatu fungsi heuristik. Teknik penelusuran ini memiliki tujuan untuk memperoleh solusi secara cepat namun belum tentu memiliki jalur yang optimal. Kompleksitas waktu dari penelusuran ini adalah $O(b^m)$ dengan m merupakan kedalaman maksimum serta kompleksitas ruang dari penelusuran ini memiliki nilai yang sama dengan kompleksitas waktunya.

Fungsi evaluasi heuristik, f(n), pada Greedy Best First Search merupakan estimasi harga dari suatu kata ke kata tujuan, ditandakan dengan notasi h(n). Pada problem ini, h(n) pada program ini adalah banyak kata yang berbeda dari suatu kata ke kata tujuan.

Dalam menentukan simpul selanjutnya yang akan diekspansi, Greedy Best First Search memilih simpul dengan harga terkecil berdasarkan nilai yang diberikan oleh fungsi heuristik. Asumsikan fungsi heuristik yang diberikan selalu memberikan nilai yang sebenarnya kepada suatu simpul. Greedy Best First Search akan bertindak seperti penelusuran Greedy biasa sehingga ia hanya memperhatikan nilai optimum lokal dengan harapan akan memperoleh optimum global. Nilai yang dijadikan referensi untuk dijadikan pemilihan ekspansi tidak bersifat holistik sehingga dapat mengecohkan penelusuran. Kasus ini serupa dengan kasus pertukaran koin melalui metode greedy hanya saja seluruh elemen greedy kita peroleh melalui struktur data graf. Hal ini menyebabkan solusi yang diberikan oleh penelusuran ini tidak terjamin optimalisasinya.

Implementasi algoritma Uniform Cost Search pada program ini (ditulis dalam bahasa pemrograman java) dilakukan dengan langkah yang serupa dengan Uniform Cost Search. Namun, pembeda dari kedua algoritma ini adalah cara menentukan cost dari setiap simpul ekspansi.

C. A* SEARCH (A STAR SEARCH)

A* Search merupakan teknik penelusuran yang menggabungkan kelebihan dari Greedy Best First Search dan Uniform Cost Search. Kelebihan yang dimaksud adalah dengan mempertimbangkan harga untuk mencapai suatu simpul g(n) dan estimasi cost untuk suatu simpul ke tujuan h(n). Kompleksitas waktu dan ruang dari penelusuran ini serupa dengan penelusuran Uniform Cost Search.

Fungsi evaluasi dari A* Search memiliki notasi f(n) = g(n) + h(n). g(n) merupakan jarak dari awal simpul ke simpul n. Sementara h(n) merupakan estimasi cost dari n ke goal. Pada problem ini, g(n) merupakan kedalaman yang dibutuhkan untuk mencapai n dan h(n) banyaknya perbedaan huruf pada kata n hingga kata tujuan. Heuristik yang digunakan pada problem ini bersifat admissible dikarenakan harga yang diberikan selalu memiliki nilai yang lebih kecil dibandingkan harga sesungguhnya.

Implementasi algoritma A* Search pada program ini (ditulis dalam bahasa pemrograman java) dilakukan dengan langkah yang serupa dengan Uniform Cost Search. Namun, pembeda dari kedua algoritma ini adalah cara menentukan cost dari setiap simpul ekspansi.

D. PERBANDINGAN ALGORITMA

Perbandingan	UCS	GDBFS	A*
Solusi Optimal	Selalu Optimal	Belum tentu optimal	Bisa optimal asalkan memiliki fungsi heuristik yang admissible
Waktu Eksekusi	Paling lambat	Paling cepat	Diantara keduanya
Jumlah Simpul Dikunjungi	Paling banyak	Paling sedikit	Diantara keduanya

Tabel 1 Perbandingan Algoritma

Penelusuran secara Uniform Cost Search dan Breadth First Search memiliki perbedaan terhadap cara untuk menentukan harga setiap simpul ekspansinya. Pada Uniform Cost Search, harga dari suatu simpul ke simpul lainnya merupakan "jarak" secara harfiah terhadap keduanya. Dalam kasus Word Ladder, jarak dari setiap kata yang diekspansi selalu bernilai satu dengan hasil ekspansinya. Ekspansi pada UCS akan memilih jarak kumulatif yang paling optimum lokal. Hal ini menyebabkan seluruh hasil kalimat yang diekspansi akan selalu menempati posisi terakhir dalam *priority queue*. Kejadian ini serupa dengan penelusuran secara Breadth First Search yang selalu menempatkan hasil ekspansinya ke dalam *queue*. Sehingga, dengan kata lain, pada problem Word Ladder, penelusuran Uniform Cost Search memiliki pemilihan simpul yang sama dengan Breadth First Search.

Perbandingan dari penelusuran dengan A* dan Uniform Cost Search dapat dilihat dari banyaknya simpul yang diekspansi. Pada penelusuran melalui A*, suatu simpul hanya akan diekspansi apabila simpul tersebut terlihat "menjanjikan", dalam hal ini menjanjikan merupakan nilai fungsi evaluasi merupakan optimum lokal. Sementara Uniform Cost Search melakukan ekspansi apabila simpul memiliki jarak sesungguhnya yang optimal. Pada kasus Word Ladder, Jarak setiap simpul memiliki nilai yang sama, kita asumsikan bernilai satu, sehingga seluruh simpul akan diekspansi sampai ditemukan solusinya. Hal ini menyebabkan pencarian melalui Uniform Cost Search menelusuri simpul lebih banyak dan memperlama durasi pencarian sementara penelusuran dengan A* menelusuri simpul lebih sedikit sehingga mempersingkat durasi pencarian. Namun, apabila fungi heuristik dari penelusuran A* tidak admissible, solusi yang diperoleh belum tentu optimal. Oleh karena itu, dapat disimpulkan

bahwa, untuk kasus Word Ladder, algoritma A* lebih efisien dibandingkan Uniform Cost Search.

IV. SOURCE CODE

Pada program ini, paradigma yang digunakan ialah berorientasi objek sehingga penggunaan prinsip polymorophism dan inheritance sangat terlihat dalam kode projek ini. Dikarenakan cara penelusuran dari ketiga teknik tersebut memiliki cara yang sama, program ini memiliki suatu super class yang bernama “Solver” yang berperan untuk menyelesaikan tugas tersebut.

```
public abstract class Solver {
    Integer node_amount;
    String goal_word;
    String start_word;
    PriorityQueue<WordNode> queue;
    final Map<String,Boolean> visited_node;

    /**
     * Constructor
     * */
    public Solver(String start_word,String goal_word) throws Exception {
        if(start_word.length() != goal_word.length()){
            throw new Exception("Start word length does not match Goal Word length");
        }
        Dictionary dictionary = new Dictionary(goal_word.length());

        visited_node =  dictionary.getWords();
        if(!visited_node.containsKey(start_word.toLowerCase())){
            throw new Exception("Your start word doesn't exist");
        }
        if(!visited_node.containsKey(goal_word.toLowerCase())){
            throw new Exception("Your goal words doesn't exist");
        }

        this.goal_word = goal_word.toLowerCase();
        this.start_word = start_word.toLowerCase();
        this.node_amount = 0;

        // Inisialisasi Queue
        queue = new PriorityQueue<>(new WordNodeComparator());
        queue.add(new WordNode(this.start_word,0,null));
    }

    public Solver(String start_word,String goal_word,String filePath) throws Exception {
        if(start_word.length() != goal_word.length()){
            throw new Exception("Start word length does not match Goal Word length");
        }
        Dictionary dictionary = new Dictionary(goal_word.length(),filePath);

        visited_node =  dictionary.getWords();
        if(!visited_node.containsKey(start_word.toLowerCase())){
            throw new Exception("Your start word doesn't exist");
        }
        if(!visited_node.containsKey(goal_word.toLowerCase())){
            throw new Exception("Your goal words doesn't exist");
        }

        this.goal_word = goal_word.toLowerCase();
        this.start_word = start_word.toLowerCase();
        this.node_amount = 0;

        // Inisialisasi Queue
        queue = new PriorityQueue<>(new WordNodeComparator());
        queue.add(new WordNode(this.start_word,0,null));
    }

    /**
```

```
     * Method untuk Ekspansi,
     * Ekspansi bersifat asbtrak agar setiap algoritma dapat menyesuaikan
     * */
    public abstract void getAdjacentWords(WordNode current_node);


    /**
     * Fungsi untuk mengembalikan jumlah simpul yang dikunjungi
     * */
    public Integer getNodeAmount() {
        return this.node_amount;
    }

    /**
     * Fungsi untuk melakukan implementasi algoritma
     * */
    public WordNode solve() throws Exception {
        System.out.println("Solving " + this.goal_word + " from " + this.start_word +"....");
        while(!queue.isEmpty()){
            this.node_amount++;
            WordNode current_node = queue.remove();
            if(current_node.getWord().equalsIgnoreCase(goal_word)){
                return current_node;
            } else {
                visited_node.put(current_node.getWord().toLowerCase(),true);
                getAdjacentWords(current_node);
            }
        }
        throw new SolutionNotFoundException();
    }
}
```

Struktur data yang berperan menjadi “Simpul” dalam penelusuran graf diberi nama kelas “WordNode”. Kelas ini menyimpan data tentang kata yang menempati, harga yang dimiliki, dan linked list menuju WordNode sebelumnya.

```
public class WordNode {
    String word;
    Integer totalCost;
    WordNode previous;

    /**
     * Constructor
     * */
    public WordNode(String word, Integer totalCost, WordNode prev){
        this.word = word;
        this.totalCost = totalCost;
        this.previous = prev;
    }

    public String getWord() {
        return word;
    }

    public Integer getTotalCost() {
        return totalCost;
    }

    public WordNode getPrevious() {
        return previous;
    }
```

```
    /**
     * Fungsi untuk mendapatkan banyak path
     * @return jumlah path
     * */
    public static Integer countPath(WordNode othernode){
        Integer count = 0;
        WordNode current = othernode;
        while(current != null){
            count++;
            current = current.getPrevious();
        }

        return count;
    }

    /**
     * Fungsi untuk mendapatkan alur path
     * @return alur path
     * */
    public static List<String> getPaths(WordNode othernode) {
        ArrayList<String> paths = new ArrayList<>();
        WordNode current = othernode;

        while(current != null){
            paths.add(current.getWord());
            current = current.getPrevious();
        }

        Collections.reverse(paths);
        return paths;
    }
}
```

Kelas untuk penelusuran menggunakan A*, UCS, GDBFS memiliki bentuk yang serupa. Kelas ini sama – sama memiliki super class yang sama, yaitu Solver.H al yang membedakan dari ketiga kelas tersebut adalah metode “getCost” yang memiliki sifat – sifat unik untuk setiap kelasnya.

```
public class GreedyBest extends Solver {
    /**
     * Constructor
     * */
    public GreedyBest(String start_word, String end_word) throws Exception {
        super(start_word,end_word);
    }

    public GreedyBest(String start_word, String end_word, String filePath) throws Exception {
        super(start_word,end_word,filePath);
    }

    /**
     * Fungsi untuk mendapatkan biaya untuk suatu simpul
     * */
    public Integer getCost(String word){
        return Dictionary.getDistance(word,this.goal_word);
    }

    public void getAdjacentWords(WordNode current_node){
        for(String word : visited_node.keySet()){
            if(!visited_node.get(word) && (Dictionary.getDistance(current_node.getWord(),word)
== 1)){
                WordNode temp_node = new WordNode(word,getCost(word),current_node);
                this.queue.add(temp_node);
            }
        }
```

```
    }
}
```

```
public class AStar extends Solver {
    /**
     * Constructor
     * */
    public AStar(String start_word, String end_word) throws Exception {
        super(start_word,end_word);
    }

    public AStar(String start_word, String end_word, String filePath) throws Exception {
        super(start_word,end_word,filePath);
    }

    /**
     * Fungsi untuk mendapatkan biaya untuk suatu simpul
     * */
    public Integer getCost(WordNode otherNode, String otherWord) {
        return Dictionary.getDistance(otherWord,goal_word) + WordNode.countPath(otherNode) + 1;
    }

    public void getAdjacentWords(WordNode current_node){
        for(String word : visited_node.keySet()){
            if(!visited_node.get(word) && (Dictionary.getDistance(current_node.getWord(),word)
== 1)){
                WordNode temp_node = new WordNode(word,getCost(current_node,word),current_node);
                this.queue.add(temp_node);
            }
        }
    }
}
```

```
public class UniformSearch extends Solver {
    /**
     * Constructor
     * */
    public UniformSearch(String start_word,String goal_word) throws Exception {
        super(start_word, goal_word);
    }

    public UniformSearch(String start_word,String goal_word, String filePath) throws Exception {
        super(start_word, goal_word,filePath);
    }
    /**
     * Fungsi untuk mendapatkan biaya untuk suatu simpul
     * */
    public Integer getCost(WordNode otherNode) {
        return otherNode.getTotalCost() + 1;
    }

    public void getAdjacentWords(WordNode current_node){

        for(String word : visited_node.keySet()){
            if(!visited_node.get(word) && (Dictionary.getDistance(current_node.getWord(),word)
== 1)){
                WordNode temp_node = new WordNode(word, getCost(current_node), current_node);
                queue.add(temp_node);
            }
        }
    }
}
```

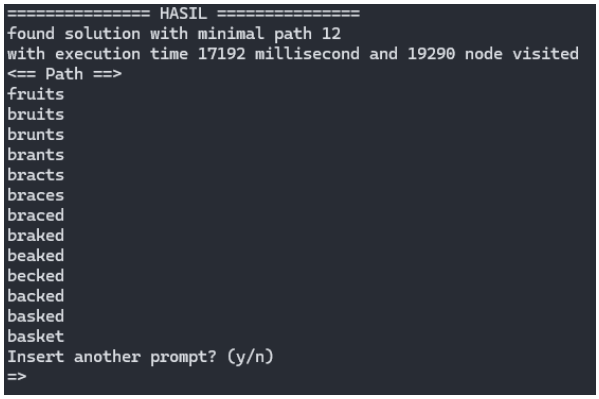
Penggunaan dari implementasi kelas – kelas tersebut adalah melalui kelas Program. Kelas ini berperan sebagai entry point setelah kelas Main untuk melakukan implementasi.

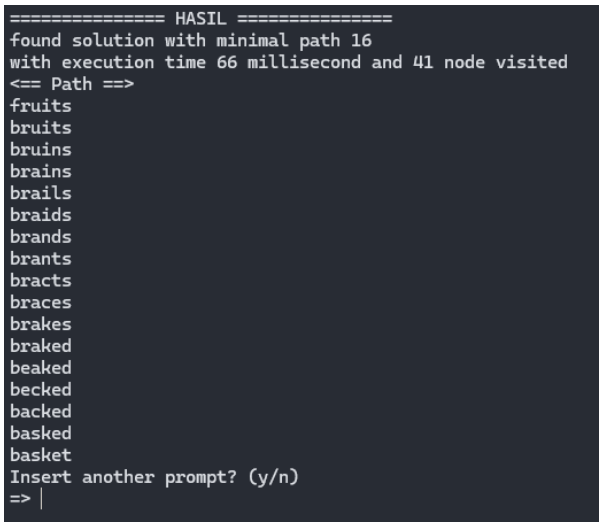
```
public class Program {
    private final String initial_string;
    private final String final_string;

    public Program() {
        Commands.clearConsole();
        System.out.println("========= Word Ladder Solver =========");
        System.out.print("Input your initial word\n=> ");
        this.initial_string = Commands.stringInput();
        System.out.print("Input your goal Word\n=> ");
        this.final_string = Commands.stringInput();
    }

    public void run() {
        try {
            System.out.println("Choose your algorithm below:");
            System.out.println("1. Uniform Cost Search");
            System.out.println("2. Greedy Best Search");
            System.out.print("3. A* Algorithm\n=> ");
            int choice = Commands.intInput();
            String customDictionary;

            Solver sv;
            while(true){
                try {
                    customDictionary = Commands.fileInput();
                    switch (choice) {
                        case 1 : sv = new UniformSearch(initial_string,
final_string,customDictionary); break;
                        case 2 : sv = new GreedyBest(initial_string,
final_string,customDictionary); break;
                        case 3 : sv = new AStar(initial_string, final_string,customDictionary);
break;
                        default: throw new Exception();
                    }
                    break;
                } catch (FileNotFoundException e) {
                    System.out.println(e.getMessage());
                } catch (Exception e) {
                    System.out.println(e.getMessage());
                    switch (choice) {
                        case 1 : sv = new UniformSearch(initial_string,final_string); break;
                        case 2 : sv = new GreedyBest(initial_string,final_string); break;
                        case 3 : sv = new AStar(initial_string,final_string); break;
                        default: throw new Exception();
                    }
                    break;
                }
            }

            Instant start = Instant.now();
            WordNode result;
            try{
                result = sv.solve();
            } catch (SolutionNotFoundException err){
                Instant end = Instant.now();
                Commands.printResult(Duration.between(start,end),sv.getNodeAmount());
                return;
            }
            Instant end = Instant.now();
            Commands.printResult(result, Duration.between(start,end),sv.getNodeAmount());
        } catch(Exception e) {
            System.out.println("====== Error ======");
            System.out.println(e.getMessage());
        }
```

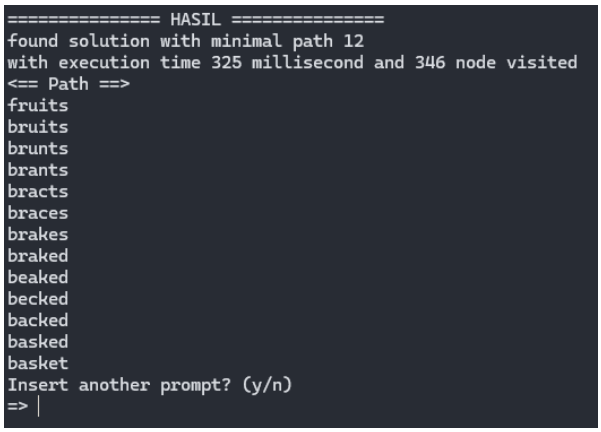
```
    }
}
```

V. HASIL PENGUJIAN

Pengujian pada tabel berikut menggunakan dictionary yang diberikan pada QnA.

<table>
<tr><th>No</th><th>Data</th><th>Hasil</th></tr>
<tr><td>1.</td><td>Start: Fruits
Finish: Basket</td><td>UCS:
 

GDBFS:
 

A*:
 
</td></tr>
<tr><td>2.</td><td>Starts: Twelve</td><td>UCS:</td></tr>
</table>

	Finish: Thirty	`============== HASIL ==============`
		`NO SOLUTION FOUND!!!`
		`Traversed path with execution time 14 millisecond and 1 node visited`
		`Insert another prompt? (y/n)`
		`=>`
		
		GDBFS:
		
		`============== HASIL ==============`
		`NO SOLUTION FOUND!!!`
		`Traversed path with execution time 1 millisecond and 1 node visited`
		`Insert another prompt? (y/n)`
		`=>`
		
		A*:
		
		`============== HASIL ==============`
		`NO SOLUTION FOUND!!!`
		`Traversed path with execution time 1 millisecond and 1 node visited`
		`Insert another prompt? (y/n)`
		`=>`

3.	Starts: Piano
	Finish: Chord
		UCS:
		
		`=============== HASIL ===============`
		`found solution with minimal path 8`
		`with execution time 12137 millisecond and 44197 node visited`
		`<== Path ==>`
		`piano`
		`pians`
		`plans`
		`plays`
		`clays`
		`chays`
		`chars`
		`chard`
		`chord`
		`Insert another prompt? (y/n)`
		`=>`
		
		GDBFS:
		
		`=============== HASIL ===============`
		`found solution with minimal path 12`
		`with execution time 8 millisecond and 23 node visited`
		`<== Path ==>`
		`piano`
		`pians`
		`pions`
		`cions`
		`clons`
		`clops`
		`chops`
		`chows`
		`chews`
		`chaws`
		`chars`
		`chard`
		`chord`
		`Insert another prompt? (y/n)`
		`=>`
		
		A*:

		`=============== HASIL ===============`
		`found solution with minimal path 8`
		`with execution time 36 millisecond and 90 node visited`
		`<== Path ==>`
		`piano`
		`pians`
		`plans`
		`plays`
		`clays`
		`chays`
		`chars`
		`chard`
		`chord`
		`Insert another prompt? (y/n)`
		`=>`

4.	Starts: Seven
	Finish: Eight
		UCS:
		`=============== HASIL ===============`
		`found solution with minimal path 9`
		`with execution time 16420 millisecond and 57146 node visited`
		`<== Path ==>`
		`seven`
		`sever`
		`serer`
		`seres`
		`sires`
		`sines`
		`sinhs`
		`sighs`
		`sight`
		`eight`
		`Insert another prompt? (y/n)`
		`=>`
		GDBFS:
		`=============== HASIL ===============`
		`found solution with minimal path 21`
		`with execution time 31 millisecond and 92 node visited`
		`<== Path ==>`
		`seven`
		`sever`
		`siver`
		`fiver`
		`river`
		`riven`
		`rives`
		`jives`
		`dives`
		`diver`
		`dived`
		`diced`
		`riced`
		`riled`
		`riped`
		`siped`
		`sipes`
		`sines`
		`sinhs`
		`sighs`
		`sight`
		`eight`
		`Insert another prompt? (y/n)`
		`=>`
		A*:

		`=============== HASIL ===============`
		`found solution with minimal path 9`
		`with execution time 315 millisecond and 987 node visited`
		`<== Path ==>`
		`seven`
		`semen`
		`semes`
		`sexes`
		`sixes`
		`sines`
		`sinhs`
		`sighs`
		`sight`
		`eight`
		`Insert another prompt? (y/n)`
		`=>`

5	Start: Frown
	Finish: Smile
		UCS:
		`=============== HASIL ===============`
		`found solution with minimal path 9`
		`with execution time 18730 millisecond and 38455 node visited`
		`<== Path ==>`
		`frown`
		`frows`
		`crows`
		`chows`
		`shows`
		`shots`
		`shote`
		`smote`
		`smite`
		`smile`
		`Insert another prompt? (y/n)`
		`=>`
		GDBFS:
		`=============== HASIL ===============`
		`found solution with minimal path 22`
		`with execution time 71 millisecond and 62 node visited`
		`<== Path ==>`
		`frown`
		`drown`
		`drawn`
		`drawl`
		`draws`
		`drams`
		`drama`
		`grama`
		`grams`
		`grabs`
		`drabs`
		`dribs`
		`dries`
		`tries`
		`trips`
		`tripe`
		`trice`
		`twice`
		`twine`
		`swine`
		`spine`
		`spile`
		`smile`
		`Insert another prompt? (y/n)`
		`=>`
		A*:

		`============== HASIL ==============`
		`found solution with minimal path 9`
		`with execution time 293 millisecond and 646 node visited`
		`<== Path ==>`
		`frown`
		`frows`
		`flows`
		`flops`
		`flips`
		`slips`
		`slipe`
		`stipe`
		`stile`
		`smile`
		`Insert another prompt? (y/n)`
		`=>`

6	Start: Bank
	Finish: Loan
		UCS:
		`============== HASIL ==============`
		`found solution with minimal path 5`
		`with execution time 4010 millisecond and 31733 node visited`
		`<== Path ==>`
		`bank`
		`bonk`
		`book`
		`boon`
		`loon`
		`loan`
		`Insert another prompt? (y/n)`
		`=>`
		GBDFS:
		`============== HASIL ==============`
		`found solution with minimal path 6`
		`with execution time 2 millisecond and 10 node visited`
		`<== Path ==>`
		`bank`
		`lank`
		`link`
		`linn`
		`lion`
		`loon`
		`loan`
		`Insert another prompt? (y/n)`
		`=>`
		A*:
		`============== HASIL ==============`
		`found solution with minimal path 5`
		`with execution time 9 millisecond and 50 node visited`
		`<== Path ==>`
		`bank`
		`bonk`
		`book`
		`boon`
		`loon`
		`loan`
		`Insert another prompt? (y/n)`
		`=>`

Tabel 2 Hasil Pengujian

VI. ANALISIS KASUS

Test Case	Uniform Cost Search		Greedy Best First Search		A*	
	Ruang (Jumlah Node)	Waktu (ms)	Ruang (Jumlah Node)	Waktu (ms)	Ruang (Jumlah Node)	Waktu (ms)
1.	19290	17192	41	66	346	325
2.	-	14	-	1	-	1
3.	44197	12137	23	8	90	36
4.	57146	16420	92	31	987	315
5.	38455	18730	62	71	646	293
6.	31733	4010	10	2	50	9

Tabel 3 Perbandingan waktu dan ruang penelusuran

Pada enam percobaan yang telah dilakukan, rujuk **Tabel 3** , terlihat perbedaan yang signifikan terhadap waktu eksekusi dan memori yang ditunjukkan oleh setiap penelusuran. Pada penelusuran menggunakan Uniform Cost Search, jumlah simpul yang dikunjungi serta waktu eksekusi yang dibutuhkan memiliki nilai yang melampaui jauh Greedy Best First Search dan A*. Hal ini menunjukkan bahwa Uniform Cost Search menjelajahi simpul lebih banyak yang mengakibatkan waktu eksekusinya menjadi lama. Namun, Uniform Cost Search selalu menghasilkan solusi yang optimal dikarenakan ekspansi pada Uniform Cost Search akan memilih jarak kumulatif yang paling optimum lokal .

Pada penelusuran Greedy Best First Search, jumlah simpul yang dikunjunginya beserta waktu eksekusinya memiliki nilai terkecil di antara kedua penelusuran lainnya. Greedy Best First Search hanya akan memilih simpul yang diharapkan akan memiliki jarak yang paling dekat menuju solusi sehingga memiliki kemungkinan besar untuk mencapai simpul solusi namun belum menjamin bahwa jalur yang diambil optimal.

Penelusuran secara A* menggabungkan keunggulan dari kedua penelusuran lainnya. A* memiliki waktu eksekusi dan jumlah simpul yang tidak lebih besar maupun lebih kecil di antara keduanya. Data tersebut mendukung teori dari algoritma A* lebih efisien dibandingkan UCS dikarenakan waktu eksekusinya yang lebih singkat. Dikarenakan solusi yang diberikan oleh A* selalu optimal, referensi dengan penelusuran Uniform Cost Search, dapat disimpulkan bahwa fungsi heuristik yang digunakan bersifat admissible.

VII. KESIMPULAN

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*.

Penelusuran kata yang digunakan untuk menyelesaikan masalah menggunakan Uniform Cost Search, Greedy Best First Search, dan A* Search memiliki perbedaan yang signifikan. Uniform Cost Search memiliki kompleksitas waktu dan ruang yang lebih besar diantara keduanya, namun memberikan hasil yang selalu optimal. Greedy Best First Search memiliki kompleksitas yang lebih rendah dibandingkan dengan dua algoritma pencarian lainnya. Hal ini mengakibatkan waktu yang dibutuhkan untuk menyelesaikan pencarian menjadi sangat sedikit, dan jumlah simpul yang dieksplorasi juga minim. Selain itu, penggunaan fungsi heuristik yang eksklusif pada Greedy Best First Search dapat mengurangi kemungkinan menemukan solusi yang optimal. Penelusuran dengan menggunakan A* memadukan kelebihan dari kedua metode penelusuran lainnya. Algoritma ini memiliki tingkat kompleksitas yang berada di antara Uniform Cost Search dan Greedy Best First Search. Jika fungsi heuristik yang digunakan adalah admissible (tidak melebihi nilai sebenarnya), maka dapat dipastikan bahwa penelusuran dengan A* akan menghasilkan solusi yang optimal.

VIII. LAMPIRAN

Tautan Github: https://github.com/raflyhangga/Tucil3_13522014

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