Laporan Tugas Kecil 3
IF2211 Strategi Algoritma

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

Disusun oleh:

Aurelius Justin Philo Fanjaya 13522020

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

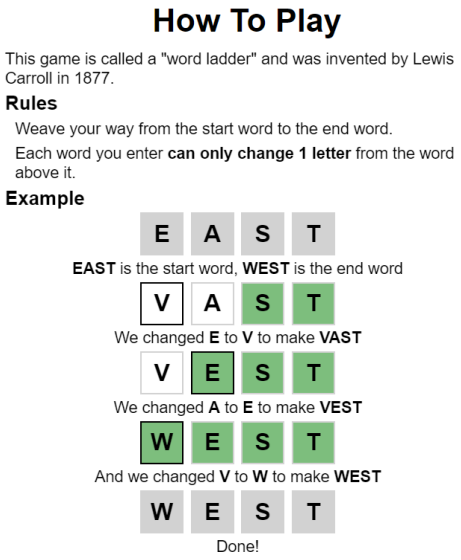
2024

Daftar Isi

BAB 1

Deskripsi Tugas

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. Ilustrasi dan Peraturan Permainan *Word Ladder*
(Sumber: https://wordwormdormdork.com/)

Permainannya cukup sederhana bukan? Jika belum paham dengan peraturan permainannya, cobalah untuk memainkan permainannya pada link sumber di atas. Jika sudah paham dengan permainannya, sekarang adalah waktunya kalian untuk membuat sebuah solver permainan tersebut dengan harapan kita dapat menemukan solusi paling optimal untuk menyelesaikan permainan *Word Ladder* ini.

BAB 2

Analisis dan implementasi dalam algoritma UCS, Greedy Best First Search, dan A*

2.1. Uniform Cost Search (UCS)

Algoritma Uniform Cost Search adalah salah satu algoritma pencarian tanpa tambahan informasi *goal node* (Blind Search atau Uninformed Search). Pencarian dilakukan berdasarkan *cost (g(n))* atau dalam kasus ini adalah jarak *node* dari *root node*. Urutan *expand node* diatur menggunakan struktur data Priority Queue yang terurut dari *cost* terkecil hingga terbesar. Berikut adalah langkah-langkah algoritma UCS yang saya implementasikan untuk kasus Word ladder dalam tugas ini.

Pertama, *start word* atau kata awal dimasukkan ke dalam Priority Queue yang masih kosong. Kemudian, pencarian dilakukan dengan dequeue node yang berada pada head Priority Queue, dan dicek jika *word* di *node* tersebut merupakan *goal word*. Jika node sama dengan *goal word*, maka path ditemukan dan fungsi akan mengembalikan (*return*) *path* dari *root* ke *node* tersebut. Jika tidak sama, maka node akan di-*expand* dan semua *word* yang belum pernah di-*expand* akan di-*enqueue* ke dalam Priority Queue dengan *cost* bernilai *cost* dari *node* yang sedang di-*expand* ditambah 1. Proses *expand* dari Priority Queue dan pengecekan kata akan dilakukan hingga Priority Queue kosong atau ditemukan solusi. Jika Priority Queue kosong, maka tidak ditemukan solusi dan fungsi mengembalikan *null*.

2.2. Greedy Best First Search (GBFS)

Algoritma Greedy Best First Search adalah salah satu algoritma pencarian dengan tambahan informasi *goal node* (Informed Search). Pencarian dilakukan berdasarkan fungsi evaluasi *f(n)* yang bernilai heuristik *h(n)* yaitu jumlah huruf yang berbeda dari kata pada *node* dengan kata pada *goal node*. Urutan *expand node* diatur menggunakan struktur data Priority Queue yang terurut dari *f(n)* terkecil hingga terbesar. Berikut adalah langkah-langkah algoritma GBFS yang saya implementasikan untuk kasus Word ladder dalam tugas ini.

Pertama, *start word* atau kata awal dimasukkan ke dalam Priority Queue yang masih kosong. Kemudian, pencarian dilakukan dengan dequeue node yang berada pada head Priority Queue, dan dicek jika *word* di *node* tersebut merupakan *goal word*. Jika node sama dengan *goal word*, maka path ditemukan dan fungsi akan mengembalikan (*return*) *path* dari *root* ke *node* tersebut. Jika tidak sama, maka node akan di-*expand* dan semua *word* yang belum pernah di-*expand* akan di-*enqueue* ke dalam Priority Queue dengan *f(n)* bernilai heuristik yaitu jumlah perbedaan huruf dari kata pada *node* yang sedang di-*expand* dengan kata pada *goal node*. Proses *expand* dari Priority Queue dan pengecekan kata akan dilakukan hingga Priority Queue kosong atau ditemukan solusi. Jika Priority Queue kosong, maka tidak ditemukan solusi dan fungsi mengembalikan *null*.

2.3. A*

Algoritma A* (dibaca *A star*) adalah salah satu algoritma pencarian dengan tambahan informasi *goal node* (Informed Search). Pencarian dilakukan berdasarkan fungsi evaluasi *f(n)* yang bernilai *cost (g(n))* yaitu jarak dari *root node* ke *node* ditambah nilai heuristik *h(n)* yaitu jumlah huruf yang berbeda dari kata pada *node* dengan kata pada *goal node*. Urutan *expand node* diatur menggunakan struktur data Priority Queue yang terurut dari *f(n)* terkecil hingga terbesar. Berikut adalah langkah-langkah algoritma A* yang saya implementasikan untuk kasus Word ladder dalam tugas ini.

Pertama, *start word* atau kata awal dimasukkan ke dalam Priority Queue yang masih kosong. Kemudian, pencarian dilakukan dengan dequeue node yang berada pada head Priority Queue, dan dicek jika *word* di *node* tersebut merupakan *goal word*. Jika node sama dengan *goal word*, maka path ditemukan dan fungsi akan mengembalikan (*return*) *path* dari *root* ke *node* tersebut. Jika tidak sama, maka node akan di-*expand* dan semua *word* yang belum pernah di-*expand* akan di-*enqueue* ke dalam Priority Queue dengan *f(n)* bernilai cost yaitu jarak *node* ke *root node* ditambah heuristik yaitu jumlah perbedaan huruf dari kata pada *node* yang sedang di-*expand* dengan kata pada *goal node*. Proses *expand* dari Priority Queue dan pengecekan kata akan dilakukan hingga Priority Queue kosong atau ditemukan solusi. Jika Priority Queue kosong, maka tidak ditemukan solusi dan fungsi mengembalikan *null*.

2.3. Analisis Pertanyaan

Urutan prioritas dari struktur data Priority Queue ditentukan berdasarkan nilai *f(n)* yaitu fungsi evaluasi heuristik yang bergantung pada suatu node, goal node, pencarian hingga node tersebut, dan domain dari pencarian. Pada Uniform Cost Search, $f(n) = g(n)$. Pada Greedy Best First Search, $f(n) = h(n)$. Pada Algoritma A*, $f(n) = g(n) + h(n)$. *g(n)* adalah *cost* yang dibutuhkan untuk sampai ke suatu node. Dalam kasus Word Ladder, *g(n)* adalah jarak dari *start word* ke suatu *node*. *h(n)* adalah estimasi jarak/*cost* untuk mencapai *goal node*. Pada kasus Word Ladder, *h(n)* adalah jumlah perbedaan huruf dari kata pada suatu *node* dengan *goal word*.

Heuristik yang *admissible* adalah heuristik yang jika untuk setiap *node* n, $h(n) \leq h^*(n)$, di mana $h^*(n)$ adalah *true cost* untuk mencapai *goal state* dari n. Pada algoritma A*, heuristik yang digunakan adalah jumlah huruf yang berbeda dari kata pada suatu node dengan *goal word*. Heuristik ini bersifat *admissible* karena untuk mencapai suatu kata, pasti dibutuhkan minimal langkah sebanyak jumlah huruf yang berbeda dari kata asal dan kata tujuan untuk mengganti huruf-huruf tersebut menjadi huruf yang sama. Dengan demikian, dijamin bahwa $h(n) \leq h^*(n)$ untuk setiap node dan dapat disimpulkan bahwa heuristik *admissible*.

Pada kasus Word Ladder, algoritma UCS menggunakan Priority Queue yang diurutkan berdasarkan *cost* yaitu langkah dari *root node* yang terkecil. Pemilihan *node expand* akan sama dengan algoritma BFS, karena pada setiap level/depth pencarian pasti memiliki *cost* yang sama sehingga semua *node* pada *level* terkecil akan di-*expand* terlebih dahulu sebelum meng-*expand node* pada level di bawahnya. Dengan demikian, urutan pembangkitan *node* dan *path* yang dihasilkan pasti sama jika urutan pemilihan *node* ketika *cost* sama pada UCS sama dengan urutan pemilihan *node* ketika *level* sama pada BFS.

Secara teoritis, Algoritma A* akan lebih efisien daripada UCS, khususnya pada kasus Word Ladder. Hal ini dikarenakan pada kasus ini algoritma UCS akan meng-*expand* setiap *node* yang ada pada tiap *level* hingga menemukan solusi (seperti BFS). Algoritma A* memiliki *f(n)* yang memperhitungkan juga heuristik dari *node* tersebut menuju *goal word*. Oleh sebab itu, node yang memiliki heuristik yang besar tidak akan di-*expand* oleh algoritma sehingga jumlah *node* yang dicek A* akan menjadi lebih sedikit dengan tetap memastikan optimalitas solusi.

Greedy Best First Search memilih minimum lokal (*child node* yang paling baik pada suatu saat). Oleh sebab itu, jika suatu minimum lokal ternyata bukan *path* terpendek ke *goal word*, maka solusi dapat menjadi tidak optimal karena *node* yang seharusnya menjadi bagian dari *path* solusi optimal tidak pernah dibangkitkan oleh algoritma hingga akhir pencarian.

BAB 3

Source Code

3.1. Main

```
import java.util.*;

class Main {
   public static void main(String[] args) {

       if (args.length != 3) {
           System.out.println("Format salah!");
           System.out.println("Format: java -jar Program.jar <StartWord> <EndWord>
<Astar / UCS / GBFS>");
           System.out.println("Contoh: java -jar Program.jar Trap Hope Astar");
           return;
       }

       if (args[0].length() != args[1].length()) {
           System.out.println("Panjang kata <StartWord> dan <EndWord> berbeda!");
           return;
       }

       String startWord = args[0].toLowerCase();
       String goalWord = args[1].toLowerCase();
       HashSet<String> hs = WordReader.readDictionary("dictionary/dictionary.txt");
       long start, end;
       List<Word> solution;
       Algorithm a;

       // Pilihan Algoritma
       if (args[2].equals("Astar")) { // A*
           System.out.println("\nA* Algorithm\n");
           a = new Astar(startWord, goalWord);
       } else if (args[2].equals("UCS")) { // UCS
           System.out.println("\nUCS Algorithm\n");
           a = new UCS(startWord, goalWord);
       } else if (args[2].equals("GBFS")) { // GBFS
           System.out.println("\nGreedy Best First Search Algorithm\n");
           a = new Greedy(startWord, goalWord);
       } else { // Algoritma tidak valid
```

```
            System.out.println("Algoritma tidak valid!");
            System.out.println("Format: java -jar Program.jar <StartWord> <EndWord>
<Astar / UCS / GBFS>");
            System.out.println("Contoh: java -jar Program.jar Trap Hope Astar");
            return;
        }

        // Search
        double memory = ((double) (Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory()) / 1024 / 1024);
        start = System.nanoTime();
        solution = a.evaluate(hs);
        end = System.nanoTime();
        memory = ((double) (Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory()) / 1024 / 1024) - memory;
        // Print solution
        if (solution != null) {
            for (Word word : solution) {
                word.printWord(goalWord);
            }
            System.out.println("Steps: " + (solution.size()-1));
        } else {
            System.out.println("Tidak ditemukan solusi.");
        }
        System.out.println("Execution Time: " + (end-start)/1000000.0 + " ms");
        System.out.println("Memory Usage: " + memory + " MB");
        System.out.println("Nodes Visited: " + (a.nodesVisited) + "\n");
    }
}
```

3.2. Class Algorithm (Abstract Base Class)

```
import java.util.*;

abstract class Algorithm {
    PriorityQueue<Word> pq;
    HashSet<String> visited;
    String startWord;
    String goalWord;
    Integer nodesVisited;
```

```
    public abstract void addNextMoves(Word currentWord, HashSet<String> hs);

    public List<Word> evaluate(HashSet<String> hs) {
        while (!pq.isEmpty()) {
            nodesVisited++;
            Word currentWord = pq.remove();
            if (currentWord.getWord().equals(goalWord)) {
                return currentWord.getPath();
            } else {
                visited.add(currentWord.getWord());
                this.addNextMoves(currentWord, hs);
            }
        }
        return null;
    }
}
```

3.3. A*

```
import java.util.*;

class Astar extends Algorithm{
    Astar(String startWord, String goalWord) {
        this.pq = new PriorityQueue<>(new WordComparator());
        pq.add(new Word(startWord, 0, null));
        this.visited = new HashSet<>();
        this.startWord = startWord;
        this.goalWord = goalWord;
        this.nodesVisited = 0;
    }

    public void addNextMoves(Word currentWord, HashSet<String> hs) {
        List<String> nextWords = currentWord.getNextWords(hs);
        int nextValue = currentWord.distanceFromRoot() + 1;
        for (String word : nextWords) {
            if (!visited.contains(word)) {
                Word newWord = new Word(word, 0, currentWord);
                newWord.setValue(newWord.distanceToGoal(goalWord) + nextValue);
                pq.add(newWord);
            }
        }
```

```
    }
}
```

3.4. Greedy Best First Search

```
import java.util.*;

class Greedy extends Algorithm{
    Greedy(String startWord, String goalWord) {
        this.pq = new PriorityQueue<>(new WordComparator());
        pq.add(new Word(startWord, 0, null));
        this.visited = new HashSet<>();
        this.startWord = startWord;
        this.goalWord = goalWord;
        this.nodesVisited = 0;
    }

    public void addNextMoves(Word currentWord, HashSet<String> hs) {
        List<String> nextWords = currentWord.getNextWords(hs);
        for (String word : nextWords) {
            if (!visited.contains(word)) {
                Word newWord = new Word(word, 0, currentWord);
                newWord.setValue(newWord.distanceToGoal(goalWord));
                pq.add(newWord);
            }
        }
    }
}
```

3.5. Uniform Cost Search

```
import java.util.*;

class UCS extends Algorithm{
    UCS(String startWord, String goalWord) {
        this.pq = new PriorityQueue<>(new WordComparator());
        pq.add(new Word(startWord, 0, null));
        this.visited = new HashSet<>();
        this.startWord = startWord;
        this.goalWord = goalWord;
        this.nodesVisited = 0;
```

```
    }

    public void addNextMoves(Word currentWord, HashSet<String> hs) {
        List<String> nextWords = currentWord.getNextWords(hs);
        int nextValue = currentWord.distanceFromRoot() + 1;
        for (String word : nextWords) {
            if (!visited.contains(word)) {
                pq.add(new Word(word, nextValue, currentWord));
            }
        }
    }
}
```

3.6. Word

```
import java.util.*;

class Word {
    private String word;
    private Integer value;
    private Word pred;

    // Constructor
    Word(String word, int value, Word pred) {
        this.word = new String(word);
        this.value = value;
        this.pred = pred;
    }

    public String getWord() {
        return word;
    }

    public int getValue() {
        return value;
    }

    public Word getPred() {
        return pred;
    }
```

```
public void setValue(int value) {
    this.value = value;
}

public int distanceFromRoot() {
    if (pred == null) {
        return 0;
    } else {
        return pred.distanceFromRoot() + 1;
    }
}

public int distanceToGoal(String goal) {
    int len = word.length();
    int distance = 0;
    for (int i = 0; i < len; i++) {
        if (this.word.charAt(i) != goal.charAt(i)) {
            distance++;
        }
    }
    return distance;
}

public List<Word> getPath() {
    List<Word> temp;
    if (pred == null) {
        temp = new ArrayList<>();
    } else {
        temp = pred.getPath();
    }
    temp.add(this);
    return temp;
}

public List<String> getNextWords(HashSet<String> hs) {
    List<String> stringList = new ArrayList<>();
    int len = this.word.length();
    for (int i = 0; i < len; i++){
        StringBuilder sb = new StringBuilder(this.word);
        for (Character iterator = 'a'; iterator <= 'z'; iterator++) {
            sb.replace(i, i+1, iterator.toString());
            String currentWord = sb.toString();
```

```
                if (hs.contains(currentWord) && !iterator.equals(word.charAt(i))) {
                    stringList.add(currentWord);
                }
            }
        }
        return stringList;
    }

    public void printWord(String goal) {
        int len = word.length();
        for (int i = 0; i < len; i++) {
            Character c = this.word.charAt(i);
            if ( c == goal.charAt(i)) {
                System.out.print("\u001B[32m" + Character.toUpperCase(c) +
"\u001B[0m");
            } else {
                System.out.print(Character.toUpperCase(c));
            }
        }
        System.out.println();
    }
}

class WordComparator implements Comparator<Word> {
    public int compare(Word w1, Word w2) {
        if (w1.getValue() < w2.getValue()) {
            return -1;
        } else if (w1.getValue() > w2.getValue()){
            return 1;
        } else {
            return w1.getWord().compareTo(w2.getWord());
        }
    }
}
```

3.7. WordReader (reader dictionary)

```
import java.io.File;  // Import the File class
import java.io.FileNotFoundException;  // Import this class to handle errors
import java.util.HashSet;
import java.util.Scanner; // Import the Scanner class to read text files
```

```
class WordReader {
   public static HashSet<String> readDictionary(String path) {
       HashSet<String> hs = new HashSet<>();
       try {
           File myObj = new File(path);
           Scanner myReader = new Scanner(myObj);
           while (myReader.hasNextLine()) {
               String data = myReader.nextLine().stripTrailing();
               hs.add(data);
           }
           myReader.close();
       } catch (FileNotFoundException e) {
           System.out.println("An error occurred while reading the file.");
           e.printStackTrace();
       }
       return hs;
   }
}
```

BAB 4

Test Case

Tabel 4.1. Tabel Test Case

<table>
<tr><th>Test Case</th><th>Algorithms</th></tr>
<tr><td rowspan="6">TC 1

LIME

CARP</td><td>A*</td></tr>
<tr><td>

```
(base) justin@Aureliuss-MacBook-Air Tucil3_13522020 % java -jar WordLadder.jar LIME CARP Astar

A* Algorithm

LIME
LAME
CAME
CAMP
CARP
Steps: 4
Execution Time: 0.7615 ms
Nodes Visited: 6
```

</td></tr>
<tr><td>UCS</td></tr>
<tr><td>

```
(base) justin@Aureliuss-MacBook-Air Tucil3_13522020 % java -jar WordLadder.jar LIME CARP UCS

UCS Algorithm

LIME
LAME
CAME
CAMP
CARP
Steps: 4
Execution Time: 36.503334 ms
Nodes Visited: 2809
```

</td></tr>
<tr><td>Greedy Best First Search</td></tr>
<tr><td>

```
(base) justin@Aureliuss-MacBook-Air Tucil3_13522020 % java -jar WordLadder.jar LIME CARP GBFS

Greedy Best First Search Algorithm

LIME
LAME
CAME
CAMP
CARP
Steps: 4
Execution Time: 0.613458 ms
Nodes Visited: 5
```

</td></tr>
</table>

TC 2 BELL TRIP	**A***
	A* Algorithm BELL TELL TEAL TEAM TRAM TRAP TRIP Steps: 6 Execution Time: 7.903083 ms Nodes Visited: 331
	UCS
	UCS Algorithm BELL TELL TEAL TEAM TRAM TRAP TRIP Steps: 6 Execution Time: 637.077708 ms Nodes Visited: 128136
	Greedy Best First Search
	(base) justin@Aureliuss-MacBook-Air Tucil3_13522020 % java -jar WordLadder.jar BELL TRIP GBFS Greedy Best First Search Algorithm BELL TELL TALL TAIL TAIN THIN THIO TRIO TRIP Steps: 8 Execution Time: 0.890166 ms Nodes Visited: 9

TC 3 PLANE TRAIN	A*
	(base) justin@Aureliuss-MacBook-Air Tucil3_13522020 % java -jar WordLadder.jar PLANE TRAIN Astar A* Algorithm PLANE PLANK PRANK TRANK TRAIK TRAIN Steps: 5 Execution Time: 1.152208 ms Nodes Visited: 15
	UCS
	(base) justin@Aureliuss-MacBook-Air Tucil3_13522020 % java -jar WordLadder.jar PLANE TRAIN UCS UCS Algorithm PLANE PLANK PRANK TRANK TRAIK TRAIN Steps: 5 Execution Time: 65.033333 ms Nodes Visited: 5536
	Greedy Best First Search
	(base) justin@Aureliuss-MacBook-Air Tucil3_13522020 % java -jar WordLadder.jar PLANE TRAIN GBFS Greedy Best First Search Algorithm PLANE ALANE ALAND BLAND BRAND BRAID BRAIN TRAIN Steps: 7 Execution Time: 3.367083 ms Nodes Visited: 65

TC 4 SPEAKER BREAKER	A*
	(base) justin@Aureliuss-MacBook-Air Tucil3_13522020 % java -jar WordLadder.jar SPEAKER BREAKER Astar A* Algorithm SPEAKER SPEARER SPEARED SHEARED SHEAVED SLEAVED CLEAVED CLEATED BLEATED BLEATER BLEAKER BREAKER Steps: 11 Execution Time: 7.377209 ms Nodes Visited: 209
	UCS
	(base) justin@Aureliuss-MacBook-Air Tucil3_13522020 % java -jar WordLadder.jar SPEAKER BREAKER UCS UCS Algorithm SPEAKER SPEARER SHEARER SHEARED SHEAVED SLEAVED CLEAVED CLEATED BLEATED BLEATER BLEAKER BREAKER Steps: 11 Execution Time: 43.532125 ms Nodes Visited: 2558
	Greedy Best First Search

```
(base) justin@Aureliuss-MacBook-Air Tucil3_13522020 % java -jar WordLadder.jar SPEAKER BREAKER GBFS

Greedy Best First Search Algorithm

SPEAKER
SPEARER
SHEARER
SHEERER
CHEERER
CHEEPER
CHEAPER
CHEATER
THEATER
TREATER
GREATER
GREASER
CREASER
CREAMER
CREAMED
CREAKED
WREAKED
WREAKER
BREAKER
Steps: 18
Execution Time: 2.830083 ms
Nodes Visited: 43
```

TC 5	A*
PRINT BRAVE	

```
(base) justin@Aureliuss-MacBook-Air Tucil3_13522020 % java -jar WordLadder.jar PRINT BRAVE Astar

A* Algorithm

PRINT
PRINK
PRANK
CRANK
CRANE
CRAVE
BRAVE
Steps: 6
Execution Time: 1.340959 ms
Nodes Visited: 20
```

	UCS

	(base) justin@Aureliuss-MacBook-Air Tucil3_13522020 % java -jar WordLadder.jar PRINT BRAVE UCS UCS Algorithm PRINT PRINK PRANK CRANK CRANE CRAVE BRAVE Steps: 6 Execution Time: 33.0815 ms Nodes Visited: 2076
	Greedy Best First Search
	(base) justin@Aureliuss-MacBook-Air Tucil3_13522020 % java -jar WordLadder.jar PRINT BRAVE GBFS Greedy Best First Search Algorithm PRINT PRINK BRINK BRANK BRANT BRACT BRACE BRAVE Steps: 7 Execution Time: 1.528167 ms Nodes Visited: 22
TC 6 BASTE LEMON	**A***
	(base) justin@Aureliuss-MacBook-Air Tucil3_13522020 % java -jar WordLadder.jar BASTE LEMON Astar A* Algorithm BASTE BASTS BASKS BACKS BECKS DECKS DECOS DEMOS DEMON LEMON Steps: 9 Execution Time: 29.978625 ms Nodes Visited: 1672
	UCS

```
(base) justin@Aureliuss-MacBook-Air Tucil3_13522020 % java -jar WordLadder.jar BASTE LEMON UCS

UCS Algorithm

BASTE
BASTS
LASTS
LASES
LAMES
LIMES
LIMEN
LIMAN
LEMAN
LEMON
Steps: 9
Execution Time: 1117.863708 ms
Nodes Visited: 225201
```

Greedy Best First Search

```
(base) justin@Aureliuss-MacBook-Air Tucil3_13522020 % java -jar WordLadder.jar BASTE LEMON GBFS

Greedy Best First Search Algorithm

BASTE
BASTS
BESTS
BEATS
BEADS
LEADS
LENDS
LENOS
LINOS
LIMOS
LIMAS
LIMAN
LEMAN
LEMON
Steps: 13
Execution Time: 7.860542 ms
Nodes Visited: 348
```

BAB 5

Analisis Perbandingan Solusi UCS, Greedy Best First Search, dan A*

```
A* Algorithm

BELL
TELL
TEAL
TEAM
TRAM
TRAP
TRIP
Steps: 6
Execution Time: 7.903083 ms
Nodes Visited: 331
```

*Gambar 5.1 Test Case 2 A**

```
UCS Algorithm

BELL
TELL
TEAL
TEAM
TRAM
TRAP
TRIP
Steps: 6
Execution Time: 637.077708 ms
Nodes Visited: 128136
```

Gambar 5.2 Test Case 2 UCS

```
(base) justin@Aureliuss-MacBook-Air Tucil3_13522020 % java -jar WordLadder.jar BELL TRIP GBFS

Greedy Best First Search Algorithm

BELL
TELL
TALL
TAIL
TAIN
THIN
THIO
TRIO
TRIP
Steps: 8
Execution Time: 0.890166 ms
Nodes Visited: 9
```

Gambar 5.1 Test Case 2 Greedy Best First Search

5.1. Perbandingan Optimalitas

Algoritma UCS dan A* pasti menghasilkan solusi yang paling optimal karena kedua algoritma tersebut meng-*expand* seluruh *node* yang masih mungkin menghasilkan solusi optimal. Algoritma Greedy Best First Search memilih *node* yang paling baik pada saat itu (local minima) dan mungkin saja tidak pernah meng-*expand node* yang seharusnya menjadi bagian dari *path* solusi optimal. Oleh sebab itu, jika terdapat pemilihan local minima yang tidak menghasilkan solusi optimal, maka hasil akhir tidak optimal. Oleh sebab itu, Algoritma Greedy Best First Search tidak menjamin solusi optimal. Pada gambar 5.1 dan 5.2, pencarian A* dan UCS dapat menemukan solusi optimal (6 step), sedangkan Greedy Best First Search menemukan solusi dengan 9 step yang bukan merupakan solusi optimal.

5.2. Perbandingan Waktu Eksekusi

Berdasarkan hasil *test case* pada gambar 5.1, 5.2, dan 5.3, serta pada tabel 4.1, dapat dilihat bahwa urutan waktu eksekusi dari yang paling cepat adalah Algoritma Greedy Best First Search, A*, dan UCS. Kecepatan waktu eksekusi dari tiap algoritma ini berbanding lurus dengan jumlah node yang dikunjungi oleh tiap algoritma. Semakin banyak *node* yang dikunjungi pada suatu pencarian, maka waktu eksekusinya menjadi lebih lama. Algoritma Greedy Best First Search memiliki waktu eksekusi paling cepat karena algoritma ini hanya mengecek dan meng-*expand node* dengan heuristik terkecil. Meskipun solusi akhir dapat lebih panjang dari algoritma yang lain (tidak optimal), tetapi algoritma ini tetap lebih cepat karena jumlah node yang dicek lebih sedikit dari algoritma A* dan UCS. Algoritma A* mengunjungi node yang lebih sedikit daripada algoritma UCS karena algoritma A* tidak meng-*expand* node yang tidak mungkin menghasilkan solusi optimal, sehingga waktu eksekusinya menjadi lebih cepat.

5.3. Perbandingan Memory

Memory yang digunakan oleh algoritma dalam suatu pencarian dapat diperkirakan dari jumlah node yang dikunjungi atau di-*expand* pada pencarian tersebut karena ketika meng-*expand* suatu *node*, setiap *node* yang dimasukkan ke dalam *queue* akan menggunakan *memory* untuk menyimpan *node* tersebut. Berdasarkan urutan jumlah *node* yang dikunjungi, didapatkan bahwa urutan penggunaan *memory* algoritma dari yang paling sedikit adalah Greedy Best First Search, A*, dan UCS. Alasan dari urutan ini sama seperti pada perbandingan waktu eksekusi, yaitu bahwa Greedy Best First Search hanya meng-*expand node* dengan heuristik paling kecil, A* hanya meng-*expand node* yang masih mungkin untuk menuju solusi optimal, dan UCS meng-*expand* setiap *node* yang mungkin pada tiap *level* hingga ditemukan solusi.

5.4. Kesimpulan Analisis Perbandingan

Berdasarkan analisis perbandingan di atas, didapatkan bahwa algoritma yang menjamin optimalitas adalah algoritma A* dan UCS, sedangkan Greedy Best First Search tidak menjamin optimalitas. Kemudian, urutan algoritma yang memiliki waktu eksekusi tercepat dan penggunaan memori paling sedikit Greedy Best First Search, A*, dan UCS.

Jadi, algoritma Greedy Best First Search baik digunakan dalam kasus dibutuhkan waktu eksekusi yang cepat, meskipun solusi tidak pasti optimal. Algoritma A* baik digunakan untuk menemukan solusi yang optimal meskipun tidak secepat Greedy Best First Search. Algoritma UCS tidak terlalu baik digunakan dalam kasus ini karena tidak memiliki kelebihan apapun dibandingkan Algoritma A*, tapi dapat digunakan pada kasus lain yang tidak dapat ditentukan heuristik dari persoalan.

BAB 6

Implementasi Bonus

7.1. Code

```
import javax.swing.*;
import javax.swing.table.DefaultTableModel;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.sql.Array;
import java.util.*;

public class GUI extends JFrame {
   private JPanel MainPanel;
   private JTextField AlgorithmField;
   private JTextField StartWordField;
   private JLabel GoalWordLabel;
   private JTable resultTable;
   private JButton searchButton;
   private JLabel startWordLabel;
   private JTextField GoalWordField;
   private JTable InfoTable;

   public GUI() {

       searchButton.addActionListener(new ActionListener() {
           @Override
           public void actionPerformed(ActionEvent e) {
               String Status;
               if (GoalWordField.getText().length() !=
StartWordField.getText().length()) {
                   Status = "Panjang kata <StartWord> dan <EndWord> berbeda!";
                   Object ColumnInfo[] = {"Waktu Eksekusi", "Nodes Visited", "Steps",
"Status"};
                   Object dataInfo[][] = {{"-", "-", "-", Status}};
                   InfoTable.getTableHeader().setVisible(true);
                   InfoTable.setModel(new DefaultTableModel(dataInfo, ColumnInfo));
               } else {
                   String startWord = StartWordField.getText().toLowerCase();
                   String goalWord = GoalWordField.getText().toLowerCase();
```

```
                    HashSet<String> hs =
WordReader.readDictionary("dictionary/dictionary.txt");
                    long start, end;
                    List<Word> solution;
                    Algorithm a;

                    // Pilihan Algoritma
                    if (AlgorithmField.getText().equals("Astar")) { // A*
                        a = new Astar(startWord, goalWord);
                    } else if (AlgorithmField.getText().equals("UCS")) { // UCS
                        a = new UCS(startWord, goalWord);
                    } else if (AlgorithmField.getText().equals("GBFS")) { // GBFS
                        a = new Greedy(startWord, goalWord);
                    } else { // Algoritma tidak valid
                        Status = "Algoritma Tidak Valid! (UCS/Astar/GBFS)";
                        Object ColumnInfo[] = {"Waktu Eksekusi", "Nodes Visited",
"Steps", "Status"};
                        Object dataInfo[][] = {{"-", "-", "-", Status}};
                        InfoTable.getTableHeader().setVisible(true);
                        InfoTable.setModel(new DefaultTableModel(dataInfo,
ColumnInfo));
                        return;
                    }

                    // Search
                    start = System.nanoTime();
                    solution = a.evaluate(hs);
                    end = System.nanoTime();
                    List<char[]> tempList = new ArrayList<>();
                    String exec = "" + (end-start)/1000000.0;

                    // Print solution
                    if (solution != null) {
                        for (Word word : solution) {
                            tempList.add(word.getWord().toCharArray());
                        }

                        Object ColumnName[] = new Object[startWord.length()];
                        for (int i = 0; i < startWord.length(); i++) {
                            ColumnName[i] = 'a';
                        }
```

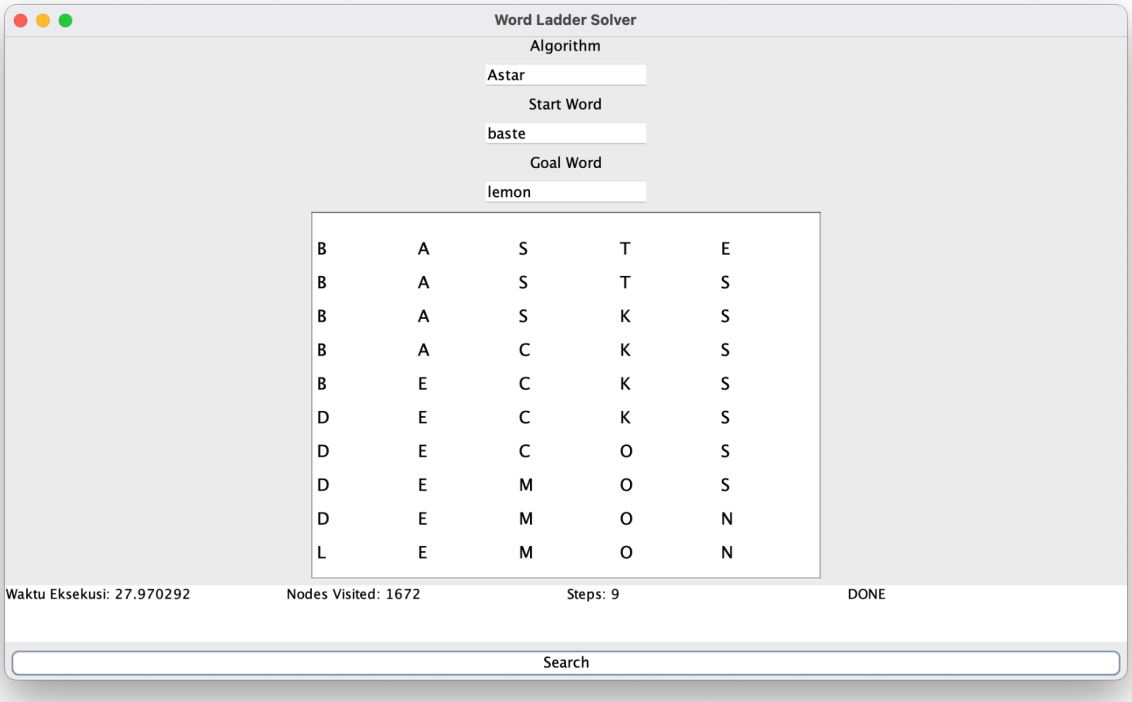
```
                        Object data[][] = new
Object[tempList.size()][startWord.length()];
                        for (int i = 0; i < tempList.size(); i++) {
                            for (int j = 0; j < startWord.length(); j++) {
                                data[i][j] = Character.toUpperCase(tempList.get(i)[j]);
                            }
                        }
                        resultTable.getTableHeader().setVisible(false);
                        resultTable.setRowHeight(30);
                        resultTable.setModel(new DefaultTableModel(data, ColumnName));
                        Object ColumnInfo[] = {"Waktu Eksekusi", "Nodes Visited",
"Steps", "Status"};
                        Object dataInfo[][] = {{"Waktu Eksekusi: " + exec, "Nodes
Visited: " + a.nodesVisited, "Steps: " + (solution.size()-1), "DONE"}};
                        InfoTable.getTableHeader().setVisible(true);
                        InfoTable.setModel(new DefaultTableModel(dataInfo,
ColumnInfo));
                    } else {
                        Status = "Tidak ditemukan solusi";
                        Object ColumnInfo[] = {"Waktu Eksekusi", "Nodes Visited",
"Steps", "Status"};
                        Object dataInfo[][] = {{"Waktu Eksekusi: " + exec, "Nodes
Visited: " + a.nodesVisited, "Steps: -", Status}};
                        InfoTable.getTableHeader().setVisible(true);
                        InfoTable.setModel(new DefaultTableModel(dataInfo,
ColumnInfo));
                    }

                }
            }
        });
    }

    public static void main(String[] args) {
        GUI h = new GUI();
        h.setContentPane(h.MainPanel);
        h.setTitle("Word Ladder Solver");
        h.setSize(1000, 600);
        h.setVisible(true);
        h. setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
```

```
}
```

7.2. Tampilan



7.3. Implementasi

Implementasi GUI menggunakan Java Swing dengan IntelliJ.

Kesimpulan

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	✓	

Link Repository

https://github.com/AureliusJustin/Tucil3_13522020