

**TUGAS KECIL 3 IF2211 STRATEGI ALGORITMA**  
**Penyelesaian Permainan Word Ladder Menggunakan Algoritma**  
**UCS, Greedy Best First Search, dan A\***



Oleh:  
**Ahmad Mudabbir Arif**  
**13522072**

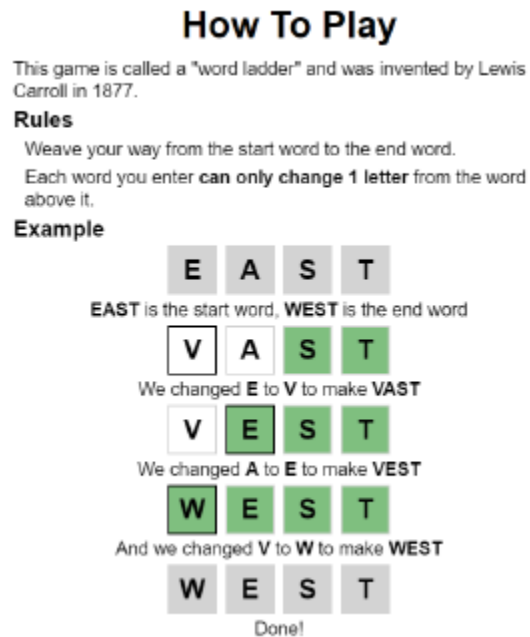
**Sekolah Teknik Elektro dan Informatika**  
**Institut Teknologi Bandung**  
**2024**

## BAB I

### PENDAHULUAN

#### A. 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/>)

#### B. Algoritma Pencarian

##### 1. UCS (Uniform Cost Search)

Uniform Cost Search adalah algoritma pencarian graf yang digunakan untuk mencari jalur terpendek dari satu node ke node lainnya. Algoritma ini mirip dengan algoritma Dijkstra, namun UCS tidak memperhitungkan bobot dari edge. Algoritma ini memerlukan struktur data priority queue untuk menyimpan node yang akan dieksplorasi selanjutnya. Algoritma UCS akan memilih node dengan cost terkecil dari priority queue

untuk dieksplorasi selanjutnya. Algoritma ini akan terus berjalan hingga menemukan node target atau frontier kosong.

## 2. Greedy Best First Search

Greedy Best First Search adalah algoritma pencarian graf yang digunakan untuk mencari jalur terpendek dari satu node ke node lainnya. Algoritma ini memilih node yang memiliki nilai heuristik terkecil dari priority queue untuk dieksplorasi selanjutnya. Algoritma ini tidak memperhitungkan cost dari node, sehingga algoritma ini tidak selalu menghasilkan jalur terpendek. Algoritma ini akan terus berjalan hingga menemukan node target atau frontier kosong.

## 3. A\* (A Star)

A\* adalah algoritma pencarian graf yang digunakan untuk mencari jalur terpendek dari satu node ke node lainnya. Algoritma ini menggabungkan UCS dan Greedy Best First Search dengan mempertimbangkan cost dari node dan nilai heuristik. Algoritma ini memilih node dengan nilai  $f = g + h$  terkecil dari priority queue untuk dieksplorasi selanjutnya. Algoritma ini akan terus berjalan hingga menemukan node target atau frontier kosong.

## 4. Penyelesaian Word Ladder

Algoritma UCS, Greedy Best First Search, dan A\* digunakan untuk menemukan jalur dari kata awal ke kata akhir dengan biaya yang sama untuk setiap langkah. Algoritma ini akan menghasilkan jalur terpendek dari kata awal ke kata akhir dengan mempertimbangkan cost dari node dan nilai heuristik. Algoritma ini akan terus berjalan hingga menemukan node target atau frontier kosong.

## BAB II

### ANALISIS DAN IMPLEMENTASI

#### A. UCS (*Uniform Cost Search*)

Algoritma UCS (*Uniform Cost Search*) digunakan untuk menemukan jalur dari kata awal ke kata akhir dengan biaya yang sama untuk setiap langkah. Pada implementasinya, nilai  $f(n) = g(n)$ , dengan  $g(n)$  adalah banyaknya langkah yang dihitung dari node yang pertama di explore. Algoritma ini menginisialisasi frontier sebagai *PriorityQueue* yang dalam hal ini untuk menyimpan simpul hidup dengan comparator yang membandingkan cost dari node. Kemudian, algoritma ini menambahkan start node ke frontier sebagai node pertama yang diperiksa dan selama frontier tidak kosong, algoritma ini mengambil node dari frontier yang memiliki cost terkecil. Algoritma ini menambahkan 1 ke *nodesExplored* untuk menghitung jumlah node yang dikunjungi dan jika node saat ini adalah target node, maka algoritma ini akan mengembalikan path dari node tersebut. Algoritma ini menambahkan node saat ini ke *explored* dan generate neighbors dari node saat ini. Untuk setiap neighbor, algoritma ini membuat node baru dengan cost yang sama dengan  $g(\text{node saat ini})$  dan menambahkan node saat ini sebagai parent dari neighbor. Algoritma ini menambahkan neighbor ke frontier jika belum pernah dieksplorasi dan tidak ada di frontier. Jika neighbor sudah ada di frontier, algoritma ini membandingkan costnya dengan cost node di frontier. Jika cost neighbor lebih kecil, algoritma ini akan menghapus node di frontier dan menambahkan neighbor ke frontier. Jika frontier kosong, algoritma ini akan mengembalikan null.

Algoritma UCS tidak sama dengan BFS pada kasus word ladder. Algoritma UCS akan memilih node yang memiliki nilai  $g(n)$  terkecil untuk diekspansi selanjutnya, sedangkan Greedy BFS akan memilih node yang berada pada level yang lebih dalam menurut nilai heuristik terlebih dahulu untuk diekspansi selanjutnya. Sehingga, urutan node yang dibangkitkan dan path yang dihasilkan oleh algoritma UCS dan BFS pada kasus word ladder tidak akan sama.

#### B. GBFS (*Greedy Best-First Search*)

Algoritma Greedy Best-First Search adalah algoritma yang mirip dengan UCS, namun algoritma ini hanya mempertimbangkan nilai heuristik dari node. Algoritma ini akan memilih node yang memiliki nilai heuristik terkecil dari *priority queue* untuk dieksplorasi selanjutnya. Pada implementasinya, nilai  $f(n) = h(n)$ , dengan  $h(n)$  adalah jumlah karakter atau huruf yang berbeda antara *current word* dan *end word*. Implementasi algoritma ini dimulai dengan inisialisasi frontier sebagai *PriorityQueue* dengan comparator yang membandingkan cost dari node. Selanjutnya, inisialisasi *explored* sebagai Set untuk menyimpan node yang sudah dieksplorasi. Kemudian,

tambahkan node start ke frontier. Selama frontier tidak kosong, lakukan langkah-langkah berikut: ambil node dari frontier yang memiliki nilai heuristik terkecil, jika node tersebut merupakan node target, maka kembalikan jalur dari node start ke node target, tambahkan node tersebut ke explored, generate neighbors dari node tersebut, untuk setiap tetangga, jika tetangga belum dieksplorasi, maka tambahkan tetangga tersebut ke frontier dengan nilai heuristik yang sesuai, ambil node selanjutnya dari frontier dan hapus semua node dari frontier kecuali node selanjutnya. Jika frontier kosong, kembalikan null.

Pada *Greedy Best-First Search* ini penentuan node yang akan di explore berikutnya akan menurut nilai dari heuristik node neighbor yang dibangun dan pencarian dilakukan secara greedy yaitu mengambil langkah terbaik saat itu atau minimum local. Sehingga node neighbor tidak perlu di simpan karena tidak adanya *backtracking*.

Algoritma Greedy Best First Search tidak selalu menemukan solusi pada persoalan word ladder. Algoritma Greedy Best First Search hanya memilih node yang memiliki nilai  $h(n)$  terkecil untuk diekspansi selanjutnya. Sehingga, algoritma Greedy Best First Search dapat terjebak pada local minimum atau local maximum yang tidak menghasilkan solusi optimal. Dengan demikian, algoritma Greedy Best First Search tidak selalu menemukan solusi pada persoalan word ladder.

### C. A\* Search

A\* adalah algoritma yang menggabungkan UCS dan Greedy Best-First Search. Algoritma ini mempertimbangkan nilai  $g(n)$  dan  $h(n)$  dari node untuk menentukan prioritas node yang akan dieksplorasi selanjutnya. Algoritma ini menginisialisasi frontier sebagai PriorityQueue dengan comparator yang membandingkan cost dari node. Kemudian, algoritma ini menambahkan start node ke frontier dan selama frontier tidak kosong, algoritma ini mengambil node dari frontier yang memiliki nilai  $f(n)$  terkecil. Algoritma ini menambahkan 1 ke nodesExplored untuk menghitung jumlah node yang dikunjungi dan jika node saat ini adalah target node, maka algoritma ini akan mengembalikan path dari node tersebut Algoritma ini menambahkan node saat ini ke explored dan generate neighbors dari node saat ini. Untuk setiap neighbor, algoritma ini membuat node baru dengan cost  $f(n)$  dan menambahkan node saat ini sebagai parent dari neighbor. Algoritma ini menambahkan neighbor ke frontier jika belum pernah dieksplorasi dan tidak ada di frontier. Jika neighbor sudah ada di frontier, algoritma ini membandingkan costnya dengan cost node di frontier. Jika cost neighbor lebih kecil, algoritma ini akan menghapus node di frontier dan menambahkan neighbor ke frontier. Jika frontier kosong, algoritma ini akan mengembalikan null.

Heuristik yang digunakan pada algoritma A\* adalah admissible karena nilai  $h(n)$  tidak pernah melebihi nilai sebenarnya dari node n ke goal node. Dengan kata lain, nilai

$h(n)$  selalu lebih kecil atau sama dengan nilai sebenarnya dari node  $n$  ke goal node. Sehingga, algoritma A\* akan menemukan solusi optimal jika heuristik yang digunakan adalah admissible.

Secara teoritis, algoritma A\* lebih efisien dibandingkan dengan algoritma UCS pada kasus word ladder. Algoritma A\* menggunakan heuristik  $h(n)$  yang dapat membantu algoritma untuk memilih node yang memiliki jarak terpendek dari root node ke goal node untuk diekspansi selanjutnya. Sehingga, algoritma A\* akan menemukan solusi optimal dengan jumlah node yang lebih sedikit dibandingkan dengan algoritma UCS.

#### D. Perbandingan Ketiga Algoritma

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

##### 1. Optimalitas:

- UCS: Optimal karena mengeksplorasi semua node yang ada dan memilih jalur dengan biaya terendah.
- Greedy Best First Search: Tidak optimal karena hanya mempertimbangkan biaya dari node saat ini ke node target.
- A\*: Optimal karena mempertimbangkan biaya dari node saat ini ke node target dan biaya dari node awal ke node saat ini.

##### 2. Waktu eksekusi:

- UCS: Waktu eksekusi paling lama karena mengeksplorasi semua node yang ada.
- Greedy Best First Search: Waktu eksekusi lebih cepat dari UCS karena hanya mempertimbangkan biaya dari node saat ini ke node target.
- A\*: Waktu eksekusi lebih cepat dari UCS karena mempertimbangkan biaya dari node saat ini ke node target dan biaya dari node awal ke node saat ini.

##### 3. Memori yang dibutuhkan:

- UCS: Membutuhkan memori yang lebih besar karena mengeksplorasi semua node yang ada.
- Greedy Best First Search: Membutuhkan memori yang lebih kecil dari UCS karena hanya mempertimbangkan biaya dari node saat ini ke node target.
- A\*: Membutuhkan memori yang lebih kecil dari UCS karena mempertimbangkan biaya dari node saat ini ke node target dan biaya dari node awal ke node saat ini.

Berdasarkan hasil analisis di atas, solusi A\* merupakan solusi terbaik karena optimal, waktu eksekusi lebih cepat dari UCS, dan membutuhkan memori yang lebih kecil dari UCS. Solusi Greedy Best First Search tidak optimal, waktu eksekusi lebih cepat dari UCS, dan membutuhkan memori yang lebih kecil dari UCS. Solusi UCS optimal, waktu eksekusi paling lama, dan membutuhkan memori yang lebih besar dari

Greedy Best First Search dan A\*. Sehingga, solusi A\* merupakan solusi terbaik untuk menyelesaikan permasalahan word ladder solver.

## BAB III

### SOURCE CODE

#### A. Algorithms

##### 1. Algorithm.java

```
1 public abstract class Algorithm implements FunctionValue, GValue, HValue {
2     private HashSet<String> dictionary;
3     public static int nodesExplored = 0;
4
5     public Algorithm(String filename) {
6         dictionary = new Dictionary(filename).getDictionary();
7     }
8
9     public HashSet<String> getDictionary() {
10         return dictionary;
11     }
12
13     protected List<String> generateNeighbors(Node node, Dictionary dictionary) {
14         List<String> neighbors = new ArrayList<>();
15         String word = node.getWord();
16         char[] chars = word.toCharArray();
17         for (int i = 0; i < word.length(); i++) {
18             char originalChar = chars[i];
19             for (char c = 'A'; c <= 'Z'; c++) {
20                 chars[i] = c;
21                 String neighborWord = new String(chars);
22                 if (!neighborWord.equals(word) && dictionary.isContains(neighborWord)) {
23                     neighbors.add(neighborWord);
24                 }
25             }
26             chars[i] = originalChar;
27         }
28         return neighbors;
29     }
30
31     public List<String> constructPath(Node node) {
32         List<String> ladder = new ArrayList<>();
33         while (node != null) {
34             ladder.add(node.getWord());
35             node = node.getParent();
36         }
37         Collections.reverse(ladder);
38         return ladder;
39     }
40
41     public int g(Node node) {
42         int gValue = 0;
43         Node currentNode = node;
44         while (currentNode != null) {
45             gValue++;
46             currentNode = currentNode.getParent();
47         }
48         return gValue;
49     }
50
51     public int h(String node, String targetWord) {
52         int hValue = 0;
53         for (int i = 0; i < node.length(); i++) {
54             if (node.charAt(i) != targetWord.charAt(i)) {
55                 hValue++;
56             }
57         }
58         return hValue;
59     }
60
61     public int f(Node node, String targetWord) {
62         return g(node) + h(node.getWord(), targetWord);
63     }
64
65     public abstract List<String> findPath(Node startWord, Node targetWord, Dictionary dictionary);
66 }
```



Kelas Algorithm merupakan kelas abstrak yang mengimplementasikan interface nilai fungsi yang memiliki beberapa method yang digunakan untuk mencari jalur dari suatu kata ke kata lainnya. Kelas ini memiliki beberapa method yang digunakan untuk menghitung nilai fungsi, nilai g, nilai h, serta method untuk menghasilkan tetangga dari suatu kata. Kelas ini memiliki tiga kelas turunan yaitu GreedyBFS, UCS, dan AStar. Ketiga kelas tersebut memiliki method findPath yang digunakan untuk mencari jalur dari suatu kata ke kata lainnya. Ketiga kelas tersebut menggunakan method-method yang ada di kelas Algorithm untuk mencari jalur dari suatu kata ke kata lainnya.

Atribut:

- Dictionary, untuk menyimpan dictionary yang akan dijadikan referensi pada algoritma pencarian.
- nodeExplored, untuk menyimpan jumlah node yang dijelajahi.

Method:

- Algorithm(String filename), sebagai konstruktor.
- getDictionary(), getter atribut dictionary.
- generateNeighbors(Node node, Dictionary dictionary), untuk men-generate simpul tetangga dari node dengan referensi kata dari dictionary.
- constructPath(Node node), untuk melakukan konstruksi jalur yang sudah dibangun dengan menelusuri parent dari node hingga parentnya null.
- f(Node node, String targetWord), untuk mendapatkan nilai fungsi heuristik  $g(n)$  dan  $h(n)$ .
- g(Node node), untuk mendapatkan nilai  $g(n)$  dengan menghitung jumlah langkah dari node awal ke node saat ini.
- h(String node, String targetWord), untuk mendapatkan nilai heuristik  $h(n)$  dengan menghitung jumlah karakter/huruf yang berbeda antara kata pada node dengan target katanya atau goal word.
- findPath(Node startWord, Node targetWord, Dictionary dictionary), merupakan method abstrak untuk mencari path berdasarkan algoritma turunannya.

## 2. UCS.java

```
1 public class UCS extends Algorithm {
2     public UCS(String filename) {
3         super(filename);
4     }
5
6     @Override
7     public List<String> findPath(Node start, Node target, Dictionary dictionary) {
8         PriorityQueue<Node> frontier = new PriorityQueue<>(Comparator.comparing(Node::getCost));
9         Set<String> explored = new HashSet<>();
10        frontier.add(start);
11
12        while (!frontier.isEmpty()) {
13            Node currentNode = frontier.poll();
14            nodesExplored++;
15            if (currentNode.getWord().equals(target.getWord())) {
16                return constructPath(currentNode);
17            }
18            explored.add(currentNode.getWord());
19            List<Node> neighborsNode = new ArrayList<>();
20            List<String> neighbors = generateNeighbors(currentNode, dictionary);
21            for (String neighborWord : neighbors) {
22                Node neighborNode = new Node(neighborWord, g(currentNode) + 1);
23                currentNode.addChild(neighborNode);
24                neighborNode.setParent(currentNode);
25                neighborsNode.add(neighborNode);
26            }
27            for (Node neighbor : neighborsNode) {
28                if (!explored.contains(neighbor.getWord()) && !frontier.contains(neighbor)) {
29                    frontier.add(neighbor);
30                    explored.add(neighbor.getWord());
31                } else if (frontier.contains(neighbor)) {
32                    for (Node node : frontier) {
33                        if (node.getWord().equals(neighbor.getWord()) && node.getCost() > neighbor.getCost()) {
34                            frontier.remove(node);
35                            frontier.add(neighbor);
36                        }
37                    }
38                }
39            }
40        }
41
42        return null;
43    }
44 }
```

Method:

- UCS(String filename), sebagai konstruktor.
- findPath(Node startWord, Node targetWord, Dictionary dictionary), merupakan method turunan kelas Algorithm untuk mencari path dari start word ke target word dengan referensi kata dari dictionary sesuai algoritma UCS.

### 3. GreedyBFS.java

```
1 public class GreedyBFS extends Algorithm {
2     public GreedyBFS(String filename) {
3         super(filename);
4     }
5
6     @Override
7     public List<String> findPath(Node start, Node target, Dictionary dictionary) {
8         PriorityQueue<Node> frontier = new PriorityQueue<>(Comparator.comparing(Node::getCost));
9         Set<String> explored = new HashSet<>();
10        frontier.add(start);
11
12        while (!frontier.isEmpty()) {
13            Node currentNode = frontier.poll();
14            nodesExplored++;
15            if (currentNode.getWord().equals(target.getWord())) {
16                return constructPath(currentNode);
17            }
18            explored.add(currentNode.getWord());
19            List<String> neighbors = generateNeighbors(currentNode, dictionary);
20            for (String neighborWord : neighbors) {
21                if (!explored.contains(neighborWord)) {
22                    Node neighborNode = new Node(neighborWord, h(neighborWord, target.getWord()));
23                    neighborNode.setParent(currentNode);
24                    frontier.add(neighborNode);
25                }
26            }
27            Node nextNode = frontier.poll();
28            if (nextNode != null) {
29                frontier.clear();
30                frontier.add(nextNode);
31            }
32        }
33
34        return null;
35    }
36
37 }
```

Method:

- GreedyBFS(String filename), sebagai konstruktor.
- findPath(Node startWord, Node targetWord, Dictionary dictionary), merupakan method turunan kelas Algorithm untuk mencari path dari start word ke target word dengan referensi kata dari dictionary sesuai algoritma GreedyBFS.

#### 4. AStar.java

```
1 public class AStar extends Algorithm {
2     public AStar(String filename) {
3         super(filename);
4     }
5
6     @Override
7     public List<String> findPath(Node start, Node target, Dictionary dictionary) {
8         PriorityQueue<Node> frontier = new PriorityQueue<>(Comparator.comparing(Node::getCost));
9         Set<String> explored = new HashSet<>();
10        frontier.add(start);
11
12        while (!frontier.isEmpty()) {
13            Node currentNode = frontier.poll();
14            nodesExplored++;
15            if (currentNode.getWord().equals(target.getWord())) {
16                return constructPath(currentNode);
17            }
18            explored.add(currentNode.getWord());
19            List<Node> neighborsNode = new ArrayList<>();
20            List<String> neighbors = generateNeighbors(currentNode, dictionary);
21            for (String neighborWord : neighbors) {
22                Node neighborNode = new Node(neighborWord, f(currentNode, target.getWord()) + 1);
23                currentNode.addChild(neighborNode);
24                neighborNode.setParent(currentNode);
25                neighborsNode.add(neighborNode);
26            }
27            for (Node neighbor : neighborsNode) {
28                if (!explored.contains(neighbor.getWord())) {
29                    frontier.add(neighbor);
30                    explored.add(neighbor.getWord());
31                } else if (frontier.contains(neighbor)) {
32                    for (Node node : frontier) {
33                        if (node.getWord().equals(neighbor.getWord()) && node.getCost() > neighbor.getCost()) {
34                            frontier.remove(node);
35                            frontier.add(neighbor);
36                        }
37                    }
38                }
39            }
40        }
41
42        return null;
43    }
44 }
```

Method:

- AStar(String filename), sebagai konstruktor.
- findPath(Node startWord, Node targetWord, Dictionary dictionary), merupakan method turunan kelas Algorithm untuk mencari path dari start word ke target word dengan referensi kata dari dictionary sesuai algoritma A\*.

## B. Nodes

### 1. Nodes.java

```
1 public class Nodes {
2     private Node root;
3
4     public Nodes(String word) {
5         this.root = new Node(word, 0);
6     }
7
8     public Node getRoot() {
9         return root;
10    }
11
12    public static class Node {
13        private String word;
14        private int cost;
15        private Node parent;
16        private List<Node> children;
17
18        public Node(String word, int cost) {
19            this.word = word;
20            this.cost = cost;
21            this.children = new ArrayList<>();
22        }
23
24        public String getWord() {
25            return word;
26        }
27
28        public int getCost() {
29            return cost;
30        }
31
32        public Node getParent() {
33            return parent;
34        }
35
36        public void setParent(Node parent) {
37            this.parent = parent;
38        }
39
40        public List<Node> getChildren() {
41            return children;
42        }
43
44        public void addChild(Node child) {
45            children.add(child);
46        }
47
48        public void printTree(Node root, String prefix, boolean isTail) {
49            System.out.println(prefix + (isTail ? "└─ " : "├─ ") + root.getWord() + " (" + root.getCost() + ")");
50            List<Node> children = root.getChildren();
51            for (int i = 0; i < children.size() - 1; i++) {
52                printTree(children.get(i), prefix + (isTail ? "    " : "|  "), false);
53            }
54            if (children.size() > 0) {
55                printTree(children.get(children.size() - 1), prefix + (isTail ? "    " : "|  "), true);
56            }
57        }
58    }
59 }
```

Kelas ini merupakan kelas yang berfungsi untuk membuat node yang akan digunakan dalam algoritma.

Kelas ini memiliki atribut:

- word yang merupakan kata yang akan dijadikan node
- cost yang merupakan biaya dari node tersebut

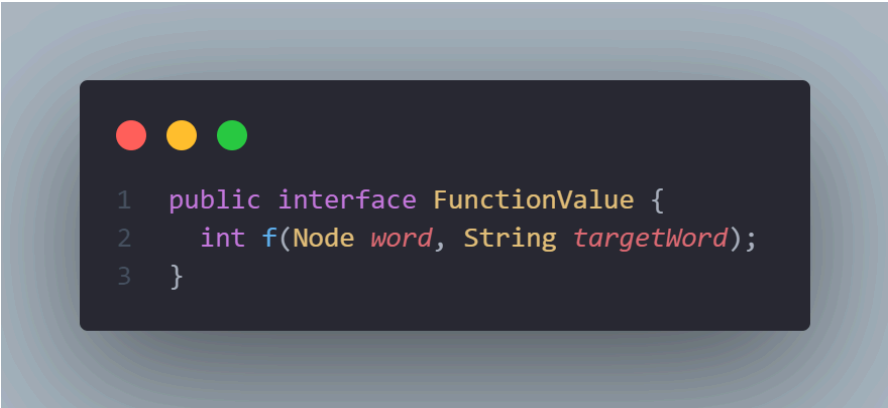
- parent yang merupakan node parent dari node tersebut
- children yang merupakan list dari node-node yang merupakan tetangga dari node tersebut.

Kelas ini memiliki method:

- `getWord()` yang digunakan untuk mendapatkan kata dari node tersebut
- `getCost()` yang digunakan untuk mendapatkan biaya dari node tersebut
- `getParent()` yang digunakan untuk mendapatkan node parent dari node tersebut
- `setParent()` yang digunakan untuk mengubah node parent dari node tersebut
- `getChildren()` yang digunakan untuk mendapatkan list dari node-node yang merupakan tetangga dari node tersebut
- `addChild()` yang digunakan untuk menambahkan node tetangga dari node tersebut
- `printTree()` yang digunakan untuk mencetak node tersebut beserta node-node tetangganya dalam bentuk pohon (untuk memudahkan debugging).

## C. Interfaces

### 1. FunctionValue.java




```

1  public interface FunctionValue {
2      int f(Node word, String targetWord);
3  }

```

Interface ini merupakan interface yang digunakan untuk menghitung nilai  $f(n)$   $= g(n) + h(n)$  pada algoritma.


## 2. GValue.java



```
1 public interface GValue {  
2     int g(Node word);  
3 }
```

Interface ini merupakan interface yang digunakan untuk menghitung nilai  $f(n)$  =  $g(n)$  pada algoritma.

## 3. HValue.java



```
1 public interface HValue {  
2     int h(String word, String targetWord);  
3 }
```

Interface ini merupakan interface yang digunakan untuk menghitung nilai  $f(n)$  =  $h(n)$  pada algoritma.

## D. Dictionary

### 1. Dictionary.java

```
1  public class Dictionary {
2      private static HashSet<String> dictionary;
3
4      public Dictionary(String filename) {
5          dictionary = new HashSet<>();
6          loadDictionary(filename);
7      }
8
9      public Dictionary(String filename, String Word) {
10         dictionary = new HashSet<>();
11         loadWordDictionary(filename, Word);
12     }
13
14     private void loadDictionary(String filename) {
15         try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
16             String word;
17             while ((word = br.readLine()) != null) {
18                 dictionary.add(word.toUpperCase());
19             }
20         } catch (IOException e) {
21             e.printStackTrace();
22         }
23     }
24
25     private void loadWordDictionary(String filename, String Word) {
26         try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
27             String word;
28             while ((word = br.readLine()) != null){
29                 if (word.length() == Word.length()) {
30                     dictionary.add(word.toUpperCase());
31                 }
32             }
33         } catch (IOException e) {
34             e.printStackTrace();
35         }
36     }
37
38     public boolean isContains(String word) {
39         return dictionary.contains(word.toUpperCase());
40     }
41
42     public HashSet<String> getDictionary() {
43         return dictionary;
44     }
45 }
```

Kelas ini merupakan kelas yang berfungsi untuk membaca file dictionary dan menyimpannya dalam atribut static dictionary berbentuk HashSet. Kelas ini




memiliki dua konstruktor, yaitu Dictionary(String filename) dan Dictionary(String filename, String Word).

Method:

- loadDictionary(String filename) digunakan untuk membaca file dictionary dan menyimpannya dalam bentuk HashSet.
- loadWordDictionary(String filename, String Word) digunakan untuk membaca file dictionary dan menyimpan kata-kata yang memiliki panjang yang sama dengan kata yang diberikan dalam bentuk HashSet.
- getDictionary() digunakan untuk mengembalikan dictionary yang telah dibaca dari file.
- isContains(String word) untuk mengecek apakah kata yang diberikan terdapat dalam dictionary.

## E. Error

### 1. Error.java



```
1 public class Error {
2     public static void showErrorPopup(String message) {
3         JOptionPane.showMessageDialog(null, message, "Error", JOptionPane.ERROR_MESSAGE);
4     }
5 }
```

Kelas ini merupakan kelas yang digunakan untuk menampilkan popup error pada GUI. Kelas ini memiliki satu method yaitu showErrorPopup yang digunakan untuk menampilkan popup error dengan pesan yang diberikan. Kelas ini tidak memiliki hubungan dengan kelas lainnya.

## F. Main Program

### 1. Driver.java

Kelas ini merupakan kelas yang berfungsi untuk menjalankan algoritma UCS, GreedyBFS, dan AStar. Kelas ini memiliki tiga method yaitu DriverUCS, DriverGreedyBFS, dan DriverAStar. Method-method ini menerima input berupa start word, target word, dan directory path. Method-method ini akan menjalankan algoritma UCS, GreedyBFS, dan AStar untuk mencari path dari start word ke target word. Method-method ini akan menampilkan hasil path yang ditemukan, jumlah node yang diexplore, dan waktu yang dibutuhkan untuk menjalankan algoritma. Kelas ini memiliki hubungan dengan kelas-kelas Algorithm, Dictionary, dan Nodes.

Kelas Driver menggunakan kelas Algorithm, Dictionary, dan Nodes untuk menjalankan algoritma UCS, GreedyBFS, dan AStar.

## 2. Main.java

Kelas ini merupakan kelas utama yang berfungsi untuk menjalankan program Word Ladder Solver dalam bentuk CLI. Kelas ini memiliki method main yang berfungsi untuk menerima input dari pengguna berupa kata awal dan kata akhir, serta memilih algoritma yang akan digunakan untuk menyelesaikan permasalahan Word Ladder. Kelas ini juga memiliki beberapa atribut konstanta yang berisi ANSI escape codes untuk memberikan warna pada output program. Kelas ini juga menggunakan kelas Driver untuk menjalankan algoritma UCS, Greedy Best First Search, dan A\*.

```

Welcome to Word Ladder Solver!
Please select directory path for dictionary file (e.g. dictionary.txt)
Directory path: dictionary.txt
Please enter the start word: crisp
Please enter the end word: count
Pilih algoritma:
1. UCS
2. Greedy Best First Search
3. A*
Pilihan: 3
=====
Start word: CRISP
Target word: COUNT
216 nodes explored
Path found: 7 steps
CRISP - CRISE - CRUSE - CRUDE - COUDE - COURE - COURT - COUNT
Time taken: 10ms

```

### 3. WordLadderApp.java

Kelas ini merupakan kelas utama yang berfungsi sebagai GUI untuk menampilkan aplikasi Word Ladder dengan menggunakan java swing.

# WORD LADDER

Start Word

base

End Word

root

Algorithm

A\* Search

Dictionary

dictionary.txt

Solve

Result

BASE

BAST

RAST

ROST

ROOT

Paths Found: 4

Time: 91ms

Nodes Explored: 46

## BAB III

### HASIL PENGUJIAN

#### 1. Test 1

**WORD LADDER**

Start Word  
base

End Word  
root

Algorithm: Uniform Cost Search | Dictionary: dictionary2.bt

**Solve**

**Result**

- BASE
- BASK
- BOSK
- BOOK
- BOOT
- ROOT

Paths Found: 5 | Time: 44ms | Nodes Explored: 2242

**WORD LADDER**

Start Word  
base

End Word  
root

Algorithm: Greedy Best-First Se... | Dictionary: dictionary2.bt

**Solve**

**Result**

- BASE
- RASE
- ROSE
- ROBE
- RODE
- ROLE
- ROPE
- ROTE
- ROUE
- ROUT
- ROOT

Paths Found: 10 | Time: 22ms | Nodes Explored: 11

**WORD LADDER**

Start Word  
base

End Word  
root

Algorithm: A\* Search | Dictionary: dictionary2.bt

**Solve**

**Result**

- BASE
- BASK
- BOSK
- BOOK
- BOOT
- ROOT

Paths Found: 5 | Time: 21ms | Nodes Explored: 411

## 2. Test 2

# WORD LADDER

Start Word

End Word

Algorithm

Dictionary

**Solve**

### Result

CRISP  
CRIMP  
CRIMS  
CREMS  
CREES  
TREES  
TYEES  
TYKES  
TAKES  
TAKEN

Paths Found: 9      Time: 85ms      Nodes Explored: 3828

# WORD LADDER

Start Word

End Word

Algorithm

Dictionary

**Solve**

### Result

No solution found or word not in dictionary.

Path found: 0 steps      Time taken: N/A      Nodes Explored: 13

# WORD LADDER

Start Word

End Word

Algorithm

Dictionary

**Solve**

### Result

CRISP  
CRIMP  
CRIMS  
CRIES  
TRIES  
TREES  
TYEES  
TYKES  
TAKES  
TAKEN

Paths Found: 9      Time: 35ms      Nodes Explored: 669

### 3. Test 3

# WORD LADDER

Start Word

End Word

Algorithm

Dictionary

**Solve**

## Result

SCRUB  
SHRUB  
SHRUG  
SPRUG  
SPRUE  
SPREE  
SAREE  
LAREE  
LARES  
LAKES  
LAKER  
FAKER

Paths Found: 11      Time: 119ms      Nodes Explored: 3998

# WORD LADDER

Start Word

End Word

Algorithm

Dictionary

**Solve**

## Result

No solution found or word not in dictionary.

Path found: 0 steps      Time taken: N/A      Nodes Explored: 3

# WORD LADDER

Start Word

End Word

Algorithm

Dictionary

**Solve**

## Result

SCRUB  
SHRUB  
SHRUG  
SPRUG  
SPRUE  
SPREE  
SAREE  
LAREE  
LARES  
LAKES  
LAKER  
FAKER

Paths Found: 11      Time: 38ms      Nodes Explored: 145

#### 4. Test 4

# WORD LADDER

Start Word  
ATLASES

End Word  
COBBERS

Algorithm  
A\* Search

Dictionary  
dictionary.txt

Solve

## Result

- PEATIER
- PETTIER
- PUTTIER
- PUTTIES
- PUTTEES
- PUTTERS
- CUTTERS
- CUTLERS
- CURLERS
- CURBERS
- CUMBERS
- COMBERS
- COBBERS

Paths Found: 36      Time: 97ms      Nodes Explored: 3073

# WORD LADDER

Start Word  
ATLASES

End Word  
COBBERS

Algorithm  
Greedy Best-First Se...

Dictionary  
dictionary.txt

Solve

## Result

No solution found or word not in dictionary.

Path found: 0 steps      Time taken: N/A      Nodes Explored: 10

# WORD LADDER

Start Word  
ATLASES

End Word  
COBBERS

Algorithm  
A\* Search

Dictionary  
dictionary.txt

Solve

## Result

- PEATIER
- PEATIER
- PETTIER
- PUTTIER
- PUTTIES
- PUTTEES
- PUTTERS
- CUTTERS
- CUTLERS
- CURLERS
- CURBERS
- CUMBERS
- COMBERS
- COBBERS

Paths Found: 36      Time: 107ms      Nodes Explored: 3073

## 5. Test 5

# WORD LADDER

Start Word  
RAVE

End Word  
book

Algorithm  
Uniform Cost Search

Dictionary  
dictionary.bt

Solve

### Result

RAVE  
RACE  
RACK  
BACK  
BOCK  
BOOK

Paths Found: 5      Time: 80ms      Nodes Explored: 2426

# WORD LADDER

Start Word  
RAVE

End Word  
book

Algorithm  
Greedy Best-First Se...

Dictionary  
dictionary.bt

Solve

### Result

WAVE  
WOKE  
COKE  
HOKE  
JOKE  
MOKE  
POKE  
SOKE  
TOKE  
YOKE  
YORE  
BORE  
BORK  
BOOK

Paths Found: 20      Time: 48ms      Nodes Explored: 21

# WORD LADDER

Start Word  
RAVE

End Word  
book

Algorithm  
A\* Search

Dictionary  
dictionary.bt

Solve

### Result

RAVE  
RACE  
RACK  
ROCK  
ROOK  
BOOK

Paths Found: 5      Time: 62ms      Nodes Explored: 138



## 6. Test 6

# WORD LADDER

Start Word  
confirmed

End Word  
CLASSROOM

Algorithm  
Uniform Cost Search

Dictionary  
dictionary.txt

Solve

### Result

No solution found or word not in dictionary.

Path found: 0 steps      Time taken: N/A      Nodes Explored: 4

# WORD LADDER

Start Word  
confirmed

End Word  
CLASSROOM

Algorithm  
Greedy Best-First Se...

Dictionary  
dictionary.txt

Solve

### Result

No solution found or word not in dictionary.

Path found: 0 steps      Time taken: N/A      Nodes Explored: 4

# WORD LADDER

Start Word  
confirmed

End Word  
CLASSROOM

Algorithm  
A\* Search

Dictionary  
dictionary.txt

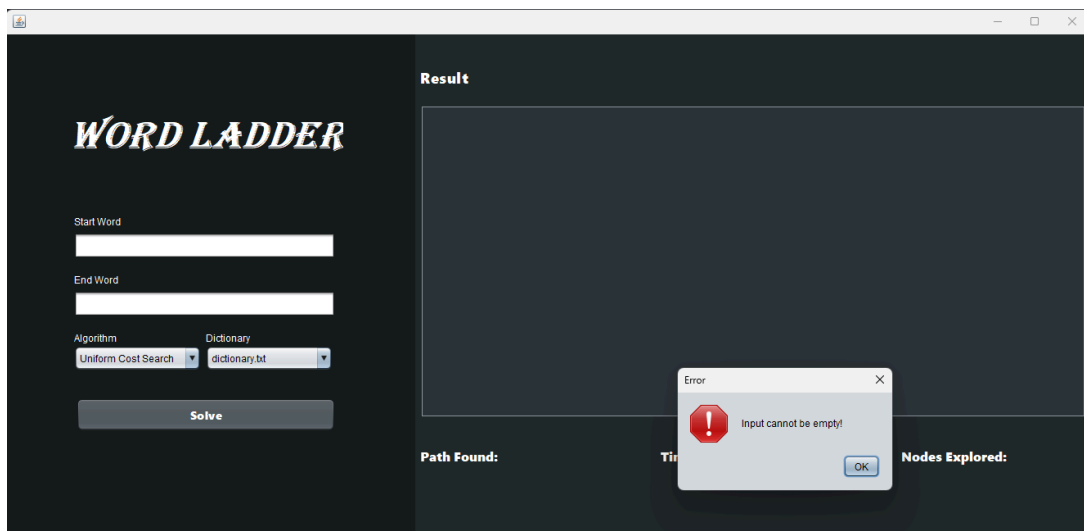
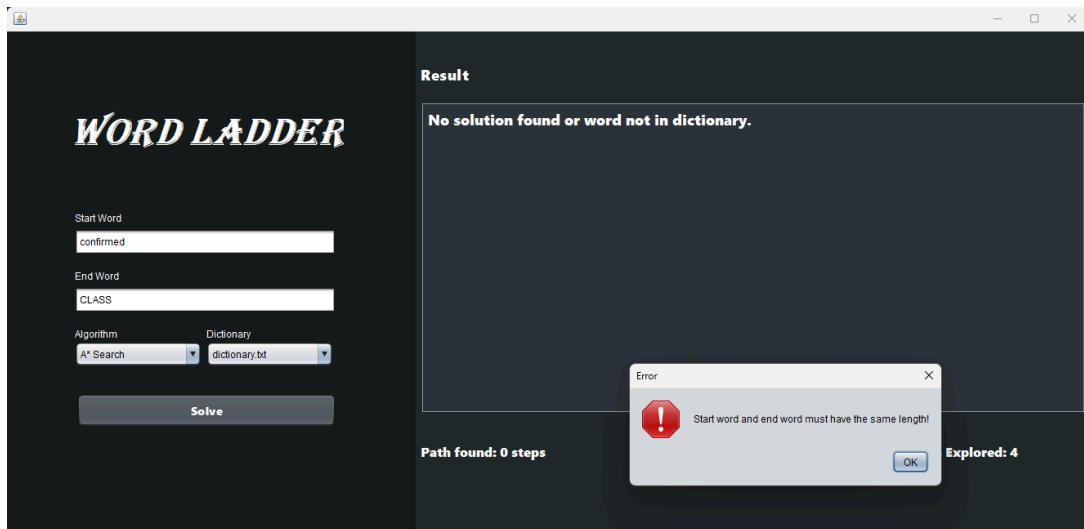
Solve

### Result

No solution found or word not in dictionary.

Path found: 0 steps      Time taken: N/A      Nodes Explored: 4

## 7. Error









## BAB IV

### LAMPIRAN

Link Repository:

[https://github.com/Dabbir/Tucil3\\_13522072.git](https://github.com/Dabbir/Tucil3_13522072.git)

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	