# LAPORAN TUGAS KECIL 3 IF2211 STRATEGI ALGORITMA

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

**Oleh:**
**Chelvadinda**
**13522154**

**PROGRAM STUDI TEKNIK INFORMATIKA**
**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA**
**INSTITUT TEKNOLOGI BANDUNG**
**2023/2024**

**DAFTAR ISI**

## BAB I
## 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.

Spesifikasi Tugas Kecil 3

- Buatlah program dalam bahasa Java berbasis CLI (Command Line Interface) – bonus jika menggunakan GUI – yang dapat menemukan solusi permainan word ladder menggunakan algoritma UCS, Greedy Best First Search, dan A*.
- Kata-kata yang dapat dimasukkan harus berbahasa Inggris. Cara kalian melakukan validasi sebuah kata dibebaskan, selama kata-kata tersebut benar terdapat pada dictionary dan proses validasi tersebut tidak memakan waktu yang terlalu lama.
- Tugas wajib dikerjakan secara individu.
- Input : Format masukan dibebaskan, dengan catatan dijelaskan pada README dan laporan. Komponen yang perlu menjadi masukan yaitu. 1. Start word dan end word. Program harus bisa menangani berbagai panjang kata (tidak hanya kata dengan 4 huruf saja seperti Gambar 1) 2. Pilihan algoritma yang digunakan (UCS, Greedy Best First Search, atau A*)
- Output : Berikut adalah luaran dari program yang diekspektasikan. 1. Path yang dihasilkan dari start word ke end word (cukup 1 path saja) 2. Banyaknya node yang dikunjungi 3. Waktu eksekusi program

# BAB II
# ANALISIS DAN IMPLEMENTASI

## 2.1 UCS (Uniform Cost Search)

Uniform Cost Search (UCS) adalah algoritma pencarian yang digunakan untuk menemukan jalur terpendek dalam graf berbobot. Algoritma ini beroperasi seperti algoritma Breadth-First Search (BFS), namun dengan mempertimbangkan bobot setiap langkah yang diambil dalam penelusuran. UCS secara terus-menerus memilih simpul dengan biaya terendah yang belum dieksplorasi, sehingga menjamin menemukan jalur terpendek dengan bobot terendah ke setiap simpul dalam graf. Pada setiap langkah, UCS mempertimbangkan biaya jalur terpendek dari simpul awal ke simpul saat ini. Jika ada jalur baru dengan biaya yang lebih rendah ke simpul tertentu, UCS memperbarui jalur tersebut. Hal ini dilakukan sampai simpul tujuan ditemukan atau semua simpul telah dieksplorasi. Kelebihan UCS adalah kemampuannya untuk menemukan jalur terpendek dengan bobot terendah. Namun, kelemahannya adalah pada graf dengan bobot yang tinggi atau tidak terbatas, UCS dapat menjadi tidak efisien karena mungkin harus mengeksplorasi banyak simpul sebelum menemukan jalur optimal.

Dalam Uniform Cost Search (UCS), biaya dari simpul n ke simpul m direpresentasikan sebagai c(n,m). Biaya total untuk mencapai simpul tertentu dalam graf direpresentasikan sebagai g(n), di mana n adalah simpul tersebut.Rumus UCS untuk menghitung biaya total g(n) dari simpul awal ke simpul saat ini adalah:

**g(n)=biaya total terendah dari simpul awal ke n**

## 2.2 Greedy Best First Search

Greedy Best First Search adalah algoritma pencarian yang memilih langkah berikutnya berdasarkan perkiraan biaya (heuristik) dari simpul tersebut menuju tujuan. Algoritma ini tidak mempertimbangkan keseluruhan biaya jalur, melainkan hanya memilih simpul yang paling dekat dengan tujuan berdasarkan estimasi heuristik. Greedy Best First Search cenderung memilih jalur yang secara lokal optimal, namun tidak menjamin menemukan jalur yang optimal secara global. Algoritma ini sangat cepat dan efisien dalam situasi di mana heuristik yang digunakan cukup akurat dalam memperkirakan jarak dari simpul saat ini ke tujuan. Namun, kekurangannya adalah dapat menemukan jalur yang tidak optimal jika estimasi heuristik tidak akurat.

Dalam Greedy Best First Search, kita menggunakan fungsi heuristik h(n) yang memberikan perkiraan biaya dari simpul saat ini n ke simpul tujuan. Biaya total untuk mencapai simpul tertentu dalam graf direpresentasikan sebagai f(n), di

mana n adalah simpul tersebut. Rumus Greedy Best First Search untuk menghitung biaya total f(n) dari simpul awal ke simpul saat ini adalah:

$$f(n)=h(n)$$

Algoritma ini hanya mempertimbangkan estimasi biaya langsung dari simpul saat ini ke tujuan (heuristik), tanpa memperhatikan total biaya sejauh ini (cost).

## 2.3 A*

A* adalah algoritma pencarian yang menggabungkan keunggulan dari UCS dan Greedy Best First Search. Algoritma ini menggunakan fungsi heuristik untuk memperkirakan biaya sisa dari simpul saat ini ke tujuan, dan mempertimbangkan biaya sejauh ini untuk mencapai simpul tersebut. Dengan memadukan informasi biaya sejauh ini dan estimasi biaya sisa, A* dapat menemukan jalur optimal dengan efisiensi yang tinggi. A* biasanya dianggap sebagai salah satu algoritma pencarian jalur paling efisien dan efektif. Dengan memanfaatkan informasi heuristik, A* dapat menjelajahi ruang pencarian dengan lebih cerdas, menghindari pengulangan yang tidak perlu, dan menuju jalur optimal dengan cepat. Dalam A*, kita juga menggunakan fungsi heuristik h(n) yang memberikan perkiraan biaya dari simpul saat ini n ke simpul tujuan. Biaya total untuk mencapai simpul tertentu dalam graf direpresentasikan sebagai (n), di mana n adalah simpul tersebut. Rumus A* untuk menghitung biaya total f(n) dari simpul awal ke simpul saat ini adalah:

**f(n)=g(n)+h(n)**

Di sini, g(n) adalah biaya sejauh ini (*cost*) untuk mencapai simpul n, sementara h(n) adalah estimasi biaya (nilai heuristik) dari simpul n ke tujuan. Heuristik yang digunakan pada algoritma A* adalah admissible. Sebuah heuristik dikatakan admissible jika nilainya tidak pernah melebih biaya sebenarnya untuk mencapai tujuan. Dalam konteks algoritma A*, heuristik digunakan untuk memperkirakan biaya yang tersisa dari simpul saat ini ke simpul tujuan. Dalam algoritma A*, nilai heuristik dari setiap simpul harus selalu kurang dari atau sama dengan biaya sebenarnya untuk mencapai tujuan dari simpul tersebut. Jika nilai heuristik untuk setiap simpul memenuhi kondisi ini, maka algoritma A* dapat menemukan solusi optimal.

## 2.4 Implementasi Algoritma

Ketiga algoritma, yaitu UCS, GBFS, dan A* memiliki implementasi yang hampir serupa. Perbedaan ketiga algoritma ini terletak pada penentuan *cost* dimana UCS menggunakan fungsi g(n), GBFS menggunakan fungsi f(n) = h(n), dan A* menggunakan f(n) = g(n) + h(n).

**UCS (Uniform Cost Search)**

1. Inisialisasi set untuk menyimpan simpul yang sudah dieksplorasi dan priority queue untuk menyimpan simpul yang akan dieksplorasi berdasarkan biaya sejauh ini.
2. Ekspansi simpul dengan *cost* terendah dari priority queue.
3. Simpan jalur yang ditemukan jika simpul tujuan telah ditemukan.
4. Output jalur yang ditemukan, jumlah node yang dikunjungi, dan waktu eksekusi program.

**GBFS (Greedy Best First Search)**

1. Inisialisasi set untuk menyimpan simpul yang sudah dieksplorasi dan priority queue untuk menyimpan simpul yang akan dieksplorasi berdasarkan nilai heuristiknya.
2. Ekspansi simpul dengan nilai heuristik terendah dari priority queue.
3. Simpan jalur yang ditemukan jika simpul tujuan telah ditemukan.
4. Output jalur yang ditemukan, jumlah node yang dikunjungi, dan waktu eksekusi program.

**A***

1. Inisialisasi set untuk menyimpan simpul yang sudah dieksplorasi dan priority queue untuk menyimpan simpul yang akan dieksplorasi berdasarkan nilai total cost (biaya sejauh ini + heuristik).
2. Ekspansi simpul dengan nilai total cost terendah dari priority queue.
3. Simpan jalur yang ditemukan jika simpul tujuan telah ditemukan.
4. Output jalur yang ditemukan, jumlah node yang dikunjungi, dan waktu eksekusi program.

# BAB III
# *SOURCE CODE* PROGRAM

## 3.1 main.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashSet;
import java.util.Scanner;
import java.util.Set;

public class main {

   // Kamus bahasa Inggris
   private static final String DICTIONARY_FILE_PATH = "Tucil3_13522154/src/kamus.txt";
   private static Set<String> dictionary = new HashSet<>(); // Set untuk menyimpan kata-kata dalam kamus

   // Method untuk memuat kamus dari file teks ke dalam Set
   private static void loadDictionary() {
      try (BufferedReader reader = new BufferedReader(new FileReader(DICTIONARY_FILE_PATH))) {
         String word;
         while ((word = reader.readLine()) != null) {
            dictionary.add(word.toLowerCase());
         }
      } catch (IOException e) {
         e.printStackTrace();
      }
   }

   // Method untuk memeriksa apakah kata ada dalam kamus
   private static boolean isValidEnglishWord(String word) {
      return dictionary.contains(word.toLowerCase());
   }

   public static void main(String[] args) {
      // Memuat kamus saat program dimulai
      loadDictionary();

      // Mendapatkan input dari pengguna
      try (Scanner scanner = new Scanner(System.in)) {
         System.out.print("Masukkan kata awal: ");
         String startWord = scanner.nextLine().toLowerCase(); // Konversi input ke huruf kecil
         System.out.print("Masukkan kata akhir: ");
         String endWord = scanner.nextLine().toLowerCase(); // Konversi input ke huruf kecil

         // Memvalidasi panjang kata
         if (startWord.length() != endWord.length()) {
            System.out.println("Panjang kata awal dan kata akhir harus sama.");
            return;
```

```
        }

        // Memvalidasi kata dalam bahasa Inggris
        if (!isValidEnglishWord(startWord) || !isValidEnglishWord(endWord)) {
            System.out.println("Kata yang dimasukkan harus dalam bahasa Inggris.");
            return;
        }

        System.out.print("Pilih algoritma (1 = UCS, 2 = Greedy Best First Search, 3 = A*): ");
        int algorithmChoice = scanner.nextInt(); // Membaca pilihan algoritma dari pengguna

        // Memilih algoritma yang sesuai
        switch (algorithmChoice) {
            case 1:
                UCS ucsSolver = new UCS();
                ucsSolver.solve(startWord, endWord); // Menyelesaikan masalah dengan algoritma UCS
                break;
            case 2:
                GreedyBestFirstSearch gbfsSolver = new GreedyBestFirstSearch();
                gbfsSolver.solve(startWord, endWord); // Menyelesaikan masalah dengan algoritma Greedy Best
First
                                                    // Search
                break;
            case 3:
                AStar aStarSolver = new AStar(); // Membuat objek solver A*
                aStarSolver.solve(startWord, endWord); // Menyelesaikan masalah dengan algoritma A*
                break;
            default:
                System.out.println("Pilihan algoritma tidak valid."); // Menampilkan pesan kesalahan jika pilihan
                                                    // algoritma tidak valid
        }
    }
  }
}
```

***Class* main:** berfungsi sebagai titik masuk untuk program.

***Method***

**loadDictionary()**: Metode untuk memuat kamus bahasa Inggris dari file teks ke dalam set dictionary. Metode ini membaca setiap kata dari file, mengonversinya menjadi huruf kecil untuk pencarian yang tidak sensitif terhadap kasus, dan menambahkannya ke dalam set.

**isValidEnglishWord(String word)**: Metode untuk memeriksa apakah sebuah kata tertentu ada dalam kamus bahasa Inggris. Metode ini mengembalikan true jika kata tersebut ditemukan dalam set dictionary (pencarian tidak sensitif terhadap kasus), jika tidak ditemukan, maka mengembalikan false.

**main(String[] args)**: Metode utama dari program. Metode ini memuat kamus, mengambil input dari pengguna untuk kata awal dan akhir, memvalidasi input, meminta pengguna untuk memilih algoritma (UCS, Greedy Best First Search, atau A*), dan kemudian menyelesaikan masalah tangga kata menggunakan algoritma yang dipilih.

## 3.2 UCS.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.*;

public class UCS {

   public void solve(String startWord, String endWord) {
      long startTime = System.nanoTime(); // Waktu awal eksekusi

      // Inisialisasi set untuk menyimpan state yang sudah dieksplorasi
      Set<String> explored = new HashSet<>();

      // Inisialisasi priority queue untuk menyimpan state yang akan dieksplorasi
      PriorityQueue<Node> priorityQueue = new PriorityQueue<>(Comparator.comparingInt(Node::getCost));
      priorityQueue.add(new Node(startWord, 0, null));

      // Inisialisasi variabel untuk menyimpan jalur yang ditemukan
      List<String> path = null;

      // Inisialisasi variabel untuk menyimpan banyaknya node yang dikunjungi
      int visitedNodes = 0;

      // Memulai ekplorasi
      while (!priorityQueue.isEmpty()) {
         // Ambil state dengan cost terendah dari priority queue
         Node currentNode = priorityQueue.poll();

         // Tambahkan state saat ini ke dalam set explored
         explored.add(currentNode.getWord());

         // Increment jumlah node yang dikunjungi
         visitedNodes++;

         // Periksa jika state saat ini adalah end word
         if (currentNode.getWord().equals(endWord)) {
            // Rekam jalur yang ditemukan
            path = new ArrayList<>();
            Node traceBackNode = currentNode;
            while (traceBackNode != null) {
```

```
            path.add(traceBackNode.getWord());
            traceBackNode = traceBackNode.getParent();
          }
          Collections.reverse(path);
          break;
        }

        // Dapatkan tetangga dari state saat ini
        List<String> neighbors = getNeighbors(currentNode.getWord(), explored);

        // Eksplorasi tetangga
        for (String neighbor : neighbors) {
          int newCost = currentNode.getCost() + 1; // cost dari transisi ke tetangga adalah 1
          Node neighborNode = new Node(neighbor, newCost, currentNode);
          priorityQueue.add(neighborNode);
        }
      }

      long endTime = System.nanoTime(); // Waktu akhir eksekusi
      double executionTime = (endTime - startTime) / 1e6; // Konversi waktu ke milidetik

      // Cetak output
      if (path != null) {
        System.out.println("Path: " + String.join(" -> ", path));
        System.out.println("Banyaknya node yang dikunjungi: " + visitedNodes);
        System.out.println("Waktu eksekusi program: " + executionTime + " milidetik");
      } else {
        System.out.println("Tidak ditemukan jalur yang menghubungkan " + startWord + " dan " + endWord);
        System.out.println("Waktu eksekusi program: " + executionTime + " milidetik");
      }
    }

    private List<String> getNeighbors(String word, Set<String> explored) {
      // Metode untuk mendapatkan tetangga (kata yang berbeda satu huruf) dari sebuah kata
      List<String> neighbors = new ArrayList<>();
      for (int i = 0; i < word.length(); i++) {
        char[] charArray = word.toCharArray();
        for (char c = 'a'; c <= 'z'; c++) {
          charArray[i] = c;
          String newWord = new String(charArray);
          if (!newWord.equals(word) && isValidEnglishDictionaryLocal(newWord) &&
!explored.contains(newWord)) {
            neighbors.add(newWord);
          }
        }
      }
      return neighbors;
    }

    // Kamus bahasa Inggris
    private static final String DICTIONARY_FILE_PATH = "Tucil3_13522154/src/kamus.txt";
    private static Set<String> dictionary = new HashSet<>();
```

```
  // Method untuk memuat kamus dari file teks ke dalam Set
  private static void loadDictionary() {
    try (BufferedReader reader = new BufferedReader(new FileReader(DICTIONARY_FILE_PATH))) {
      String word;
      while ((word = reader.readLine()) != null) {
        dictionary.add(word.toLowerCase());
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  // Method untuk memeriksa apakah kata ada dalam kamus
  private static boolean isValidEnglishDictionaryLocal(String word) {
    return dictionary.contains(word.toLowerCase());
  }

  static {
    // Memuat kamus saat program dimulai
    loadDictionary();
  }

  static class Node {
    private String word;
    private int cost;
    private Node parent;

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

    public String getWord() {
      return word;
    }

    public int getCost() {
      return cost;
    }

    public Node getParent() {
      return parent;
    }
  }
}
```

***Class* UCS**: mengimplementasikan algoritma UCS untuk menemukan solusi Word ladder.

***Method***

Metode:
**solve(String startWord, String endWord)**: Metode untuk menyelesaikan Word ladder dengan menggunakan algoritma UCS.

**getNeighbors(String word, Set<String> explored)**: Metode untuk mendapatkan tetangga (kata yang berbeda satu huruf) dari sebuah kata.

**loadDictionary()**: Metode untuk memuat kamus bahasa Inggris dari file teks ke dalam set dictionary.

**isValidEnglishDictionaryLocal(String word)**: Metode untuk memeriksa apakah sebuah kata ada dalam kamus bahasa Inggris.

***Class* Node**: Kelas ini merepresentasikan simpul.

***Method***:

**getWord()**: Mengembalikan kata yang direpresentasikan oleh simpul.

**getCost()**: Mengembalikan biaya yang telah dikeluarkan untuk mencapai simpul.

**getParent()**: Mengembalikan simpul induk dari simpul saat ini dalam jalur pencarian.

## 3.3 GreedyBestFirstSearch.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.*;

public class GreedyBestFirstSearch {

   public void solve(String startWord, String endWord) {
      long startTime = System.nanoTime(); // Waktu awal eksekusi

      // Inisialisasi set untuk menyimpan state yang sudah dieksplorasi
      Set<String> explored = new HashSet<>();

      // Inisialisasi priority queue untuk menyimpan state yang akan dieksplorasi
      PriorityQueue<Node> priorityQueue = new
PriorityQueue<>(Comparator.comparingInt(Node::getHeuristic));
      priorityQueue.add(new Node(startWord, heuristic(startWord, endWord), null));
```

```
// Inisialisasi variabel untuk menyimpan jalur yang ditemukan
List<String> path = null;

// Inisialisasi variabel untuk menyimpan banyaknya node yang dikunjungi
int visitedNodes = 0;

// Memulai ekplorasi
while (!priorityQueue.isEmpty()) {
   // Ambil state dengan heuristic terendah dari priority queue
   Node currentNode = priorityQueue.poll();

   // Tambahkan state saat ini ke dalam set explored
   explored.add(currentNode.getWord());

   // Increment jumlah node yang dikunjungi
   visitedNodes++;

   // Periksa jika state saat ini adalah end word
   if (currentNode.getWord().equals(endWord)) {
      // Rekam jalur yang ditemukan
      path = new ArrayList<>();
      Node traceBackNode = currentNode;
      while (traceBackNode != null) {
         path.add(traceBackNode.getWord());
         traceBackNode = traceBackNode.getParent();
      }
      Collections.reverse(path);
      break;
   }

   // Dapatkan tetangga dari state saat ini
   List<String> neighbors = getNeighbors(currentNode.getWord(), explored);

   // Eksplorasi tetangga
   for (String neighbor : neighbors) {
      Node neighborNode = new Node(neighbor, heuristic(neighbor, endWord), currentNode);
      priorityQueue.add(neighborNode);
   }
}

long endTime = System.nanoTime(); // Waktu akhir eksekusi
double executionTime = (endTime - startTime) / 1e6; // Konversi waktu ke milidetik

// Cetak output
if (path != null) {
   System.out.println("Path: " + String.join(" -> ", path));
   System.out.println("Banyaknya node yang dikunjungi: " + visitedNodes);
   System.out.println("Waktu eksekusi program: " + executionTime + " milidetik");
} else {
   System.out.println("Tidak ditemukan jalur yang menghubungkan " + startWord + " dan " + endWord);
   System.out.println("Waktu eksekusi program: " + executionTime + " milidetik");
}
```

```
    }

    private int heuristic(String word, String endWord) {
        // Menghitung heuristik sebagai jumlah karakter yang berbeda antara word dan endWord
        int count = 0;
        for (int i = 0; i < word.length(); i++) {
            if (word.charAt(i) != endWord.charAt(i)) {
                count++;
            }
        }
        return count;
    }

    private List<String> getNeighbors(String word, Set<String> explored) {
        // Metode untuk mendapatkan tetangga (kata yang berbeda satu huruf) dari sebuah kata
        List<String> neighbors = new ArrayList<>();
        for (int i = 0; i < word.length(); i++) {
            char[] charArray = word.toCharArray();
            for (char c = 'a'; c <= 'z'; c++) {
                charArray[i] = c;
                String newWord = new String(charArray);
                if (!newWord.equals(word) && isValidEnglishDictionaryLocal(newWord) &&
!explored.contains(newWord)) {
                    neighbors.add(newWord);
                }
            }
        }
        return neighbors;
    }

    // Kamus bahasa Inggris
    private static final String DICTIONARY_FILE_PATH = "Tucil3_13522154/src/kamus.txt";
    private static Set<String> dictionary = new HashSet<>(); // Set untuk menyimpan kata-kata dalam kamus

    // Method untuk memuat kamus dari file teks ke dalam Set
    private static void loadDictionary() {
        try (BufferedReader reader = new BufferedReader(new FileReader(DICTIONARY_FILE_PATH))) {
            String word;
            while ((word = reader.readLine()) != null) {
                dictionary.add(word.toLowerCase());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    // Method untuk memeriksa apakah kata ada dalam kamus
    private static boolean isValidEnglishDictionaryLocal(String word) {
        return dictionary.contains(word.toLowerCase());
    }

    static {
```

```
        // Memuat kamus saat program dimulai
        loadDictionary();
    }

    static class Node {
        private String word;
        private int heuristic;
        private Node parent;

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

        public String getWord() {
            return word;
        }

        public int getHeuristic() {
            return heuristic;
        }

        public Node getParent() {
            return parent;
        }
    }
}
```

***Class*** **GreedyBestFirstSearch :** Kelas utama yang mengimplementasikan algoritma Greedy Best First Search untuk mencari solusi word ladder.

***Method*** :

**solve(String startWord, String endWord)**: Metode untuk menyelesaikan masalah word ladder dengan menggunakan algoritma Greedy Best First Search.

**heuristic(String word, String endWord)**: Metode untuk menghitung heuristic antara sebuah kata dan kata akhir. Metode ini menghitung jumlah karakter yang berbeda antara kedua kata tersebut.

**getNeighbors(String word, Set<String> explored)**: Metode untuk mendapatkan tetangga (kata yang berbeda satu huruf) dari sebuah kata.

**loadDictionary()**: Metode untuk memuat kamus bahasa Inggris dari file teks ke dalam set dictionary.

**isValidEnglishDictionaryLocal(String word)**: Metode untuk memeriksa apakah sebuah kata ada dalam kamus bahasa Inggris.

***Class Node* :** Kelas ini merepresentasikan simpul dalam pencarian.

***Method* :**

**getWord()**: Mengembalikan kata yang direpresentasikan oleh simpul.

**getHeuristic()**: Mengembalikan nilai heuristic dari simpul.

**getParent()**: Mengembalikan simpul induk dari simpul saat ini dalam jalur pencarian.

## 3.4 AStar.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.*;

public class AStar {

   public void solve(String startWord, String endWord) {
      long startTime = System.nanoTime(); // Waktu awal eksekusi

      // Inisialisasi set untuk menyimpan state yang sudah dieksplorasi
      Set<String> explored = new HashSet<>();

      // Inisialisasi priority queue untuk menyimpan state yang akan dieksplorasi
      PriorityQueue<Node> priorityQueue = new
PriorityQueue<>(Comparator.comparingInt(Node::getTotalCost));
      priorityQueue.add(new Node(startWord, heuristic(startWord, endWord), 0, null));

      // Inisialisasi variabel untuk menyimpan jalur yang ditemukan
      List<String> path = null;

      // Inisialisasi variabel untuk menyimpan banyaknya node yang dikunjungi
      int visitedNodes = 0;

      // Memulai ekplorasi
      while (!priorityQueue.isEmpty()) {
         // Ambil state dengan total cost terendah dari priority queue
         Node currentNode = priorityQueue.poll();

         // Tambahkan state saat ini ke dalam set explored
```

```
        explored.add(currentNode.getWord());

        // Increment jumlah node yang dikunjungi
        visitedNodes++;

        // Periksa jika state saat ini adalah end word
        if (currentNode.getWord().equals(endWord)) {
            // Rekam jalur yang ditemukan
            path = new ArrayList<>();
            Node traceBackNode = currentNode;
            while (traceBackNode != null) {
                path.add(traceBackNode.getWord());
                traceBackNode = traceBackNode.getParent();
            }
            Collections.reverse(path);
            break;
        }

        // Dapatkan tetangga dari state saat ini
        List<String> neighbors = getNeighbors(currentNode.getWord(), explored);

        // Eksplorasi tetangga
        for (String neighbor : neighbors) {
            int newCost = currentNode.getCost() + 1; // cost dari transisi ke tetangga adalah 1
            int heuristicCost = heuristic(neighbor, endWord);
            Node neighborNode = new Node(neighbor, heuristicCost, newCost, currentNode);
            priorityQueue.add(neighborNode);
        }
    }

    long endTime = System.nanoTime(); // Waktu akhir eksekusi
    double executionTime = (endTime - startTime) / 1e6; // Konversi waktu ke milidetik

    // Cetak output
    if (path != null) {
        System.out.println("Path: " + String.join(" -> ", path));
        System.out.println("Banyaknya node yang dikunjungi: " + visitedNodes);
        System.out.println("Waktu eksekusi program: " + executionTime + " milidetik");
    } else {
        System.out.println("Tidak ditemukan jalur yang menghubungkan " + startWord + " dan " + endWord);
        System.out.println("Waktu eksekusi program: " + executionTime + " milidetik");
    }
}

private int heuristic(String word, String endWord) {
    // Menghitung heuristik sebagai jumlah karakter yang berbeda antara word dan endWord
    int count = 0;
    for (int i = 0; i < word.length(); i++) {
        if (word.charAt(i) != endWord.charAt(i)) {
            count++;
        }
    }
```

```
        return count;
    }

    private List<String> getNeighbors(String word, Set<String> explored) {
        // Metode untuk mendapatkan tetangga (kata yang berbeda satu huruf) dari sebuah
        // kata
        List<String> neighbors = new ArrayList<>();
        for (int i = 0; i < word.length(); i++) {
            char[] charArray = word.toCharArray();
            for (char c = 'a'; c <= 'z'; c++) {
                charArray[i] = c;
                String newWord = new String(charArray);
                if (!newWord.equals(word) && isValidEnglishDictionaryLocal(newWord) &&
!explored.contains(newWord)) {
                    neighbors.add(newWord);
                }
            }
        }
        return neighbors;
    }

    // Kamus bahasa Inggris
    private static final String DICTIONARY_FILE_PATH = "Tucil3_13522154/src/kamus.txt";
    private static Set<String> dictionary = new HashSet<>(); // Set untuk menyimpan kata-kata dalam kamus

    // Method untuk memuat kamus dari file teks ke dalam Set
    private static void loadDictionary() {
        try (BufferedReader reader = new BufferedReader(new FileReader(DICTIONARY_FILE_PATH))) {
            String word;
            while ((word = reader.readLine()) != null) {
                dictionary.add(word.toLowerCase());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    // Method untuk memeriksa apakah kata ada dalam kamus
    private static boolean isValidEnglishDictionaryLocal(String word) {
        return dictionary.contains(word.toLowerCase());
    }

    static {
        // Memuat kamus saat program dimulai
        loadDictionary();
    }

    static class Node {
        private String word;
        private int heuristic;
        private int cost;
        private Node parent;
```

```
    public Node(String word, int heuristic, int cost, Node parent) {
      this.word = word;
      this.heuristic = heuristic;
      this.cost = cost;
      this.parent = parent;
    }

    public String getWord() {
      return word;
    }

    public int getHeuristic() {
      return heuristic;
    }

    public int getCost() {
      return cost;
    }

    public Node getParent() {
      return parent;
    }

    public int getTotalCost() {
      return cost + heuristic;
    }
  }
}
```

***Class* Astar** : Kelas utama yang mengimplementasikan algoritma A* untuk mencari solusi word ladder.

***Method* :**

**solve(String startWord, String endWord)** : Metode untuk menyelesaikan masalah word ladder dengan menggunakan algoritma A*.

**heuristic(String word, String endWord)** : Metode untuk menghitung heuristic antara sebuah kata dan kata akhir. Metode ini menghitung jumlah karakter yang berbeda antara kedua kata tersebut.

**getNeighbors(String word, Set<String> explored)** : Metode untuk mendapatkan tetangga (kata yang berbeda satu huruf) dari sebuah kata.

**loadDictionary()** : Metode untuk memuat kamus bahasa Inggris dari file teks ke dalam set `dictionary`.

**isValidEnglishDictionaryLocal(String word)** : Metode untuk memeriksa apakah sebuah kata ada dalam kamus bahasa Inggris.

***Class* Node** : Kelas ini merepresentasikan simpul dalam pencarian.

***Method* :**

**getWord()** : Mengembalikan kata yang direpresentasikan oleh simpul.

**getHeuristic()** : Mengembalikan nilai heuristic dari simpul.

**getCost()** : Mengembalikan biaya dari simpul saat ini.

**getParent()** : Mengembalikan simpul induk dari simpul saat ini dalam jalur pencarian.

**getTotalCost()** : Mengembalikan total biaya (biaya saat ini ditambah nilai heuristic) dari simpul.

# BAB IV
# *TEST CASE*

## 4.1 Test Case 1

*Start word* : CAT
*End word* : DOG

| | |
|---|---|
| UCS | `Masukkan kata awal: CAT`<br>`Masukkan kata akhir: DOG`<br>`Pilih algoritma (1 = UCS, 2 = Greedy Best First Search, 3 = A*): 1`<br>`Path: cat -> cot -> dot -> dog`<br>`Banyaknya node yang dikunjungi: 2316`<br>`Waktu eksekusi program: 82.5615 milidetik` |
| GreedyBestFirstSearch | `Masukkan kata awal: CAT`<br>`Masukkan kata akhir: DOG`<br>`Pilih algoritma (1 = UCS, 2 = Greedy Best First Search, 3 = A*): 2`<br>`Path: cat -> cot -> dot -> dog`<br>`Banyaknya node yang dikunjungi: 4`<br>`Waktu eksekusi program: 20.9137 milidetik` |
| A* | `Masukkan kata awal: CAT`<br>`Masukkan kata akhir: DOG`<br>`Pilih algoritma (1 = UCS, 2 = Greedy Best First Search, 3 = A*): 3`<br>`Path: cat -> cot -> dot -> dog`<br>`Banyaknya node yang dikunjungi: 5`<br>`Waktu eksekusi program: 6.5598 milidetik` |

## 4.2 Test Case 2

*Start word* : LOVE
*End word* : HATE

| | |
|---|---|
| UCS | `Masukkan kata awal: LOVE`<br>`Masukkan kata akhir: HATE`<br>`Pilih algoritma (1 = UCS, 2 = Greedy Best First Search, 3 = A*): 1`<br>`Path: love -> lave -> late -> hate`<br>`Banyaknya node yang dikunjungi: 428`<br>`Waktu eksekusi program: 29.3209 milidetik` |
| GreedyBestFirstSearch | `Masukkan kata awal: LOVE`<br>`Masukkan kata akhir: HATE`<br>`Pilih algoritma (1 = UCS, 2 = Greedy Best First Search, 3 = A*): 2`<br>`Path: love -> hove -> have -> hate`<br>`Banyaknya node yang dikunjungi: 4`<br>`Waktu eksekusi program: 8.4126 milidetik` |

| | |
|---|---|
| A* | Masukkan kata awal: LOVE<br>Masukkan kata akhir: HATE<br>Pilih algoritma (1 = UCS, 2 = Greedy Best First Search, 3 = A*): 3<br>Path: love -> hove -> have -> hate<br>Banyaknya node yang dikunjungi: 7<br>Waktu eksekusi program: 6.4583 milidetik |

4.3 Test Case 3

*Start word* : EAST

*End word* : WEST

| | |
|---|---|
| UCS | Masukkan kata awal: EAST<br>Masukkan kata akhir: WEST<br>Pilih algoritma (1 = UCS, 2 = Greedy Best First Search, 3 = A*): 1<br>Path: east -> wast -> west<br>Banyaknya node yang dikunjungi: 120<br>Waktu eksekusi program: 13.2364 milidetik |
| GreedyBestFirstSearch | Masukkan kata awal: EAST<br>Masukkan kata akhir: WEST<br>Pilih algoritma (1 = UCS, 2 = Greedy Best First Search, 3 = A*): 2<br>Path: east -> wast -> west<br>Banyaknya node yang dikunjungi: 3<br>Waktu eksekusi program: 6.6311 milidetik |
| A* | Masukkan kata awal: EAST<br>Masukkan kata akhir: WEST<br>Pilih algoritma (1 = UCS, 2 = Greedy Best First Search, 3 = A*): 3<br>Path: east -> wast -> west<br>Banyaknya node yang dikunjungi: 3<br>Waktu eksekusi program: 5.6693 milidetik |

4.4 Test Case 4

*Start word* : HAPPY

*End word* : BRAVE

| | |
|---|---|
| UCS | Masukkan kata awal: HAPPY<br>Masukkan kata akhir: BRAVE<br>Pilih algoritma (1 = UCS, 2 = Greedy Best First Search, 3 = A*): 1<br>Path: happy -> harpy -> harry -> parry -> perry -> perky -> peaky -> beaky -> braky -> brake -> brave<br>Banyaknya node yang dikunjungi: 80757<br>Waktu eksekusi program: 906.5503 milidetik |
| GreedyBestFirstSearch | Masukkan kata awal: HAPPY<br>Masukkan kata akhir: BRAVE<br>Pilih algoritma (1 = UCS, 2 = Greedy Best First Search, 3 = A*): 2<br>Path: happy -> gappy -> guppy -> gulpy -> gulps -> gulls -> bulls -> balls -> bills -> billy -> bilgy -> bilge -> bulge -> budge -> badge -> barge -> parge -> parve -> carve -> calve -> halve -> helve -> heave -> deave -> drave -> brave<br>Banyaknya node yang dikunjungi: 90<br>Waktu eksekusi program: 16.908 milidetik |

| | |
|---|---|
| A* | Masukkan kata awal: HAPPY<br>Masukkan kata akhir: BRAVE<br>Pilih algoritma (1 = UCS, 2 = Greedy Best First Search, 3 = A*): 3<br>Path: happy -> harpy -> hardy -> handy -> bandy -> bendy -> beady -> beaky -> braky -> brake -> brave<br>Banyaknya node yang dikunjungi: 1926<br>Waktu eksekusi program: 94.1715 milidetik |

4.5 Test Case 5

*Start word* : EAT

*End word* : SIT

| | |
|---|---|
| UCS | Masukkan kata awal: EAT<br>Masukkan kata akhir: SIT<br>Pilih algoritma (1 = UCS, 2 = Greedy Best First Search, 3 = A*): 1<br>Path: eat -> sat -> sit<br>Banyaknya node yang dikunjungi: 80<br>Waktu eksekusi program: 9.224 milidetik |
| GreedyBestFirstSearch | Masukkan kata awal: EAT<br>Masukkan kata akhir: SIT<br>Pilih algoritma (1 = UCS, 2 = Greedy Best First Search, 3 = A*): 2<br>Path: eat -> sat -> sit<br>Banyaknya node yang dikunjungi: 3<br>Waktu eksekusi program: 5.8124 milidetik |
| A* | Masukkan kata awal: EAT<br>Masukkan kata akhir: SIT<br>Pilih algoritma (1 = UCS, 2 = Greedy Best First Search, 3 = A*): 3<br>Path: eat -> sat -> sit<br>Banyaknya node yang dikunjungi: 3<br>Waktu eksekusi program: 5.8454 milidetik |

4.6 Test Case 6

*Start word* : SMILE

*End word* : ANGRY

| | |
|---|---|
| UCS | Masukkan kata awal: SMILE<br>Masukkan kata akhir: ANGRY<br>Pilih algoritma (1 = UCS, 2 = Greedy Best First Search, 3 = A*): 1<br>Tidak ditemukan jalur yang menghubungkan smile dan angry<br>Waktu eksekusi program: 1493.6404 milidetik |
| GreedyBestFirstSearch | Masukkan kata awal: SMILE<br>Masukkan kata akhir: ANGRY<br>Pilih algoritma (1 = UCS, 2 = Greedy Best First Search, 3 = A*): 2<br>Tidak ditemukan jalur yang menghubungkan smile dan angry<br>Waktu eksekusi program: 417.3397 milidetik |

| | |
|---|---|
| A* | Masukkan kata awal: SMILE<br>Masukkan kata akhir: ANGRY<br>Pilih algoritma (1 = UCS, 2 = Greedy Best First Search, 3 = A*): 3<br>Tidak ditemukan jalur yang menghubungkan smile dan angry<br>Waktu eksekusi program: 1413.2366 milidetik |

4.7 Test Case Input Error (tidak berbahasa Inggris)
*Start word* : AYAM
*End word* : KAKI

```
Masukkan kata awal: AYAM
Masukkan kata akhir: KAKI
Kata yang dimasukkan harus dalam bahasa Inggris.
```

4.8 Test Case Input Error (panjang kata tidak sama)
*Start word* : SAD
*End word* : FIGHTING

```
Masukkan kata awal: SAD
Masukkan kata akhir: FIGHTING
Panjang kata awal dan kata akhir harus sama.
```

# BAB V
# HASIL ANALISIS

Berdasarkan hasil uji coba, algoritma UCS dapat menemukan jalur dengan biaya terendah, karena secara sistematis memeriksa semua kemungkinan jalur dari simpul awal hingga simpul tujuan. Algoritma Greedy Best First Search tidak menjamin solusi optimal karena hanya mempertimbangkan nilai heuristik dari simpul tanpa memperhatikan biaya. Sedangkan algoritma A* *admissible* sehingga solusi yang diberikan pasti optimal.

Pada ketiga algoritma, waktu eksekusi UCS bisa lebih lama daripada algoritma heuristik seperti Greedy Best First Search dan A*, karena ada banyak simpul atau jalur yang harus diperiksa. Waktu eksekusi Greedy Best First Search biasanya lebih cepat daripada UCS karena algoritma ini hanya mempertimbangkan nilai heuristik dan tidak perlu memeriksa semua kemungkinan jalur. Sedangkan waktu eksekusi A* lebih cepat daripada UCS dan secara umum lebih lambat daripada Greedy Best First Search.

Selanjutnya, UCS dapat menggunakan banyak memori karena harus menyimpan semua simpul yang sudah diperiksa dalam memori. Penggunaan memori Greedy Best First Search bisa lebih rendah daripada UCS karena tidak perlu menyimpan semua simpul yang sudah dieksplorasi. Sedangkan penggunaan memori A* bisa cukup tinggi karena harus menyimpan semua simpul yang sudah diperiksa dalam memori, meskipun lebih efisien daripada UCS.

## LAMPIRAN

Repository : https://github.com/chelvadinda/Tucil3_13522154

| Poin | Ya | Tidak |
|---|---|---|
| 1. Program berhasil dijalankan. | v | |
| 2. Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma UCS | v | |
| 3. Solusi yang diberikan pada algoritma UCS optimal | v | |
| 4. Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma Greedy Best First Search | v | |
| 5. Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma A* | v | |
| 6. Solusi yang diberikan pada algoritma A* optimal | v | |
| 7. [Bonus]: Program memiliki tampilan GUI | | v |