

LAPORAN TUGAS KECIL 3
IF2211 - STRATEGI ALGORITMA

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



Dosen:

Ir. Rila Mandala, M.Eng., Ph.D.
Monterico Adrian, S.T., M.T.

K-03:

Muhammad Fatihul Irhab (13522143)

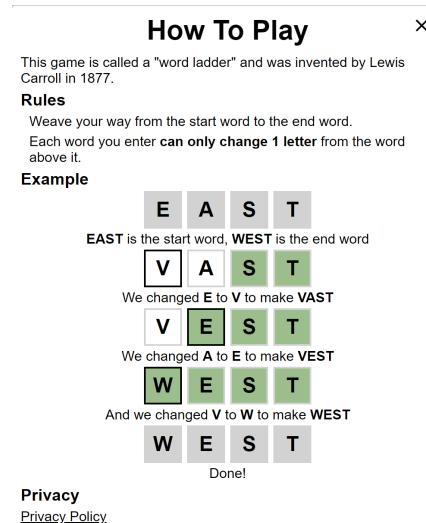
PROGRAM STUDI TEKNIK INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
SEMESTER II TAHUN 2023/2024

BAB I

DESKRIPSI MASALAH DAN ALGORITMA

1.1 Deskripsi Masalah

Word ladder (juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragraphs, 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/>)

Pada tugas kecil kali ini, penulis diminta untuk membuat suatu program yang dapat menghasilkan solusi bagi permainan World Ladder ini, dengan menggunakan algoritma UCS, Greedy BFS, dan A*. Solusi berbentuk path dari kata awal hingga kata akhir.

1.2 Algoritma UCS

Algoritma UCS (*Uniform Cost Search*) adalah sebuah algoritma pencarian jalur terpendek yang bekerja pada graf dengan bobot yang sama untuk setiap langkah atau simpul.

Langkah pertama dalam algoritma ini adalah inisialisasi, dimana sebuah himpunan (set) yang disebut *openList* berisi simpul awal dengan atribut biaya atau bobot yang menunjukkan biaya dari simpul tersebut ke simpul awal. Selain itu, terdapat himpunan kosong *closedList* yang akan menyimpan simpul-simpul yang telah dieksplorasi.

Proses iterasi dilakukan selama *openList* tidak kosong. Pada setiap iterasi, algoritma memilih simpul dengan biaya terkecil dari *openList*. Simpul yang dipilih diekspansi untuk menambahkan simpul-simpul yang dapat dicapai langsung dari simpul tersebut ke *openList*. Biaya baru untuk setiap simpul yang diekspansi disimpan sesuai dengan biaya dari simpul awal ke simpul tersebut ditambah biaya dari simpul tersebut ke simpul sekarang.

Selama proses iterasi, algoritma juga menyimpan jalur terpendek dari simpul awal ke simpul-simpul yang ada dalam *openList*. Setelah mencapai simpul tujuan atau jika *openList* kosong, algoritma berakhir. Hasil akhir dari algoritma UCS adalah jalur terpendek dari simpul awal ke simpul tujuan, yang ditemukan dengan memeriksa *closedList*. Algoritma UCS berguna dalam situasi di mana setiap langkah memiliki biaya yang berbeda-beda, sehingga dapat menemukan jalur terpendek dengan mempertimbangkan bobot setiap langkah.

1.3 Algoritma *Greedy BFS*

Algoritma *Greedy BFS* (*Greedy Best-First Search*) adalah salah satu pendekatan dalam pencarian jalur terpendek yang didasarkan pada prinsip heuristik. Dalam *Greedy BFS*, langkah selanjutnya dipilih berdasarkan nilai heuristik yang menunjukkan seberapa dekat suatu simpul dengan tujuan, tanpa mempertimbangkan biaya atau bobot langkah-langkah tersebut. Proses dimulai dengan inisialisasi *openList* yang berisi simpul awal dan *closedList* yang kosong.

Selama iterasi, *Greedy BFS* memilih simpul dengan nilai heuristik terkecil dari *openList*. Jika simpul yang dipilih adalah simpul tujuan, maka algoritma selesai. Namun, jika bukan, simpul tersebut diekspansi dengan menambahkan simpul-simpul yang dapat dicapai langsung dari simpul tersebut ke *openList*. Selain itu, nilai heuristik untuk setiap simpul yang diekspansi dihitung dan dimasukkan ke dalam *openList*. Setelah itu, simpul yang diekspansi dipindahkan ke *closedList* sebagai tanda bahwa simpul tersebut telah dieksplorasi.

Selama proses iterasi, *Greedy BFS* juga menyimpan jalur terpendek dari simpul awal ke simpul-simpul yang ada dalam *openList*. Ini membantu dalam menemukan jalur terpendek setelah mencapai simpul tujuan. Setelah selesai, *Greedy BFS* menghasilkan jalur terpendek dari simpul awal ke simpul tujuan berdasarkan nilai heuristik terkecil.

1.4 Algoritma A*

Algoritma A* adalah algoritma pencarian yang diterapkan dalam graf berbobot untuk menemukan jalur terpendek atau solusi optimal antara dua titik. Algoritma ini menggabungkan prinsip dari Dijkstra's algorithm dengan penerapan heuristik, sehingga menjadi lebih efisien dalam menemukan jalur terbaik. Langkah-langkahnya dimulai dengan inisialisasi dua himpunan node, yaitu *openList* dan *closedList*. *OpenList* berisi node-node yang masih perlu dievaluasi, sedangkan *closedList* berisi node-node yang sudah dievaluasi. Kemudian, algoritma menentukan node awal dan akhir, dan menghitung fungsi evaluasi $f(n)$ yang merupakan kombinasi dari biaya sejauh ini $g(n)$ dan nilai heuristik yang mengestimasi biaya tersisa hingga mencapai tujuan $h(n)$. Algoritma kemudian memilih node berikutnya dari *openList* berdasarkan nilai $f(n)$ terkecil, mengevaluasi tetangga-tetangga node tersebut, dan mengulangi proses hingga menemukan node akhir. Selanjutnya, algoritma menyusun jalur dari node awal ke node akhir berdasarkan informasi yang telah dievaluasi, dan menghasilkan jalur terpendek sebagai solusi optimal.

Keunggulan utama dari Algoritma A* adalah kemampuannya dalam menghindari eksplorasi yang tidak perlu melalui penerapan nilai heuristik. Heuristik ini memungkinkan algoritma untuk fokus pada node-node yang memiliki potensi membawa ke solusi optimal dengan memperkirakan biaya tersisa menuju tujuan. Dengan demikian, Algoritma A* menjadi pilihan yang efisien dalam menemukan solusi terbaik dalam graf berbobot.

BAB II

IMPLEMENTASI ALGORITMA DAN ANALISIS

Dalam pembuatan program ini, penulis menggunakan bahasa pemrograman Python. Struktur dari program ini terbagi menjadi 4 file, yaitu **Main.java**, **Graph.java**, **Util.java**, **Algoritma.java**.

2.1 File Main.java

File ini merupakan *driver* utama dari program ini, sehingga program ini hanya berisi menu utama untuk meminta masukkan pengguna, memanggil algoritma yang dibutuhkan, dan deklarasi variabel yang digunakan.

Variable Name	Description
string namaFile	<i>Variable</i> untuk menyimpan nama file kamus bahasa inggris yang digunakan
map dataMap	<i>Variable</i> untuk menyimpan semua kata bahasa inggris yang ada dalam data
string start	<i>Variable</i> untuk menyimpan <i>input</i> kata awal
string end	<i>Variable</i> untuk menyimpan <i>input</i> kata akhir
map newDataMap	<i>Variable</i> untuk menyimpan semua kata bahasa inggris yang ada dalam data dengan panjang kata sama dengan panjang kata awal.
int algoritma	<i>Variable</i> untuk menyimpan <i>input</i> nomor algoritma yang digunakan
long startTime	<i>Variable</i> untuk menyimpan waktu awal eksekusi program
long endTime	<i>Variable</i> untuk menyimpan waktu akhir eksekusi program
long excutionTime	<i>Variable</i> untuk menyimpan waktu eksekusi program

2.2 File Graph.java

File ini berisi *class* **Graph** yang dibuat untuk menyimpan node-node yang telah dikunjungi, sehingga kita bisa mendapatkan path dari node yang kita cari.

Variable Name	Description

map adjacencyList	<i>Variable</i> untuk menyimpan node saat ini sebagai <i>key</i> dan untuk menyimpan node tetangga sebagai <i>value</i>
-------------------	---

<i>Methods</i>	<i>Description</i>
int public Graph()	<i>Method</i> untuk meng- <i>instansi graph</i>
public void addVertex(String vertex)	<i>Method</i> untuk menambahkan node sebagai simpul dalam <i>graph</i>
public void addEdge(String source, String destination)	<i>Method</i> untuk menghubungkan kedua node
public List<String> getAdjacentVertices(String vertex)	<i>Method</i> untuk memberikan list node tetangga
public void printPath(String start, String end)	<i>Method</i> untuk mengeprint path dari awal hingga akhir

2.3 File Util.java

File ini berisi kumpulan fungsi yang akan digunakan dalam program ini, fungsi-fungsi yang ada dalam *class* ini akan digunakan baik di algoritma maupun main.

<i>Methods</i>	<i>Description</i>
public static Map<String, Integer> getDataFromTxt(String namaFile)	<i>Method</i> untuk mengambil data bahasa inggris dari file
public static Map<String, Integer> getDataForQueue	<i>Method</i> untuk mengambil data bahasa inggris yang memenuhi syarat antrian berikutnya
public static Map<String, Integer> getDataWithLength(Map<String, Integer> kataMap, String kataCek)	<i>Method</i> untuk mengambil data bahasa inggris yang dengan panjang kata tertentu.

2.4 File Algoritma.java

File ini berisi kumpulan fungsi algoritma yang akan digunakan dalam program ini untuk menyelesaikan persoalan-persoalan tersebut.

<i>Methods</i>	<i>Description</i>
public static void UCS(Map<String, Integer> dataMap, String awal, String akhir)	<i>Method</i> untuk mencari path dari awal ke akhir dengan menggunakan algoritma UCS
public static void GBFS(Map<String, Integer>	<i>Method</i> untuk mencari path dari awal ke akhir dengan menggunakan algoritma <i>Greedy BFS</i>

dataMap, String awal, String akhir)	
public static void ABINTANG(Map<String, Integer> dataMap, String awal, String akhir)	<i>Method</i> untuk mencari path dari awal ke akhir dengan menggunakan algoritma A*

2.5 Analisis Algoritma

Dalam algoritma UCS (*Uniform Cost Search*), urutan prioritas setiap simpul ditentukan oleh fungsi $f(n) = g(n)$, dimana $g(n)$ mengacu pada jumlah langkah dari simpul awal hingga simpul tertentu. Pada implementasi UCS ini, hanya biaya berdasarkan langkah-langkah yang diperhitungkan, tanpa memasukkan pertimbangan tambahan berupa heuristik. Sedangkan dalam algoritma *Greedy BFS*, prioritas ditentukan berdasarkan heuristik yang disebut $h(n)$. Heuristik ini digunakan untuk mengukur seberapa dekat suatu kata saat ini dengan kata tujuan. Dalam implementasi algoritma ini, nilai heuristik dihitung sebagai jumlah karakter yang berbeda antara kata saat ini dengan kata tujuan. Semakin rendah nilai $h(n)$ yang dihasilkan, semakin dekat kata tersebut dengan mencapai tujuan. Sedangkan dalam algoritma A*, urutan prioritas untuk ekstraksi dan ekspansi node ditentukan oleh fungsi biaya $f(n)$, yang merupakan hasil penjumlahan dari $g(n)$ dan $h(n)$. Pada implementasi A*, $g(n)$ mengacu pada biaya langkah-langkah yang sudah dikeluarkan hingga saat ini, sementara $h(n)$ adalah estimasi jarak antara simpul saat ini dan simpul tujuan. Estimasi ini dihitung sebagai jumlah karakter yang berbeda antara dua simpul tersebut. Dengan menggunakan kombinasi $g(n)$ dan $h(n)$, algoritma A* dapat menentukan prioritas dengan memperhitungkan kedua faktor tersebut, memungkinkan untuk ekstraksi node dengan prioritas terendah dan melanjutkan proses ekspansi dengan tepat.

Algoritma A* yang diimplementasikan dapat dikatakan admissible karena dalam implementasi ini, heuristik dihitung sebagai jumlah karakter yang berbeda antara simpul saat ini dan simpul tujuan. Karena estimasi ini merupakan estimasi minimal dari langkah-langkah yang diperlukan untuk mencapai tujuan (setiap karakter yang berbeda harus diubah), heuristik tersebut dapat dianggap admissible. Dengan demikian, heuristik yang digunakan dalam algoritma A* memenuhi syarat admissible, memastikan bahwa algoritma dapat menemukan jalur optimal dengan efisien.

Dalam konteks permasalahan Word Ladder, algoritma UCS dan BFS akan menghasilkan urutan node yang dibangkitkan dan path yang dihasilkan yang sama hal ini terjadi karena langkah atau transisi dari satu kata ke kata lain memiliki biaya yang sama, yaitu biaya langkah-langkah adalah 1, maka UCS dan BFS akan menghasilkan urutan node yang sama karena keduanya akan mencari jalur terpendek dengan biaya yang seragam.

Secara teoritis, algoritma A* memiliki potensi untuk menjadi lebih efisien daripada UCS dalam kasus Word Ladder. Hal ini disebabkan oleh perbedaan pendekatan antara kedua algoritma tersebut. UCS hanya mempertimbangkan biaya langkah yang sudah diambil tanpa memiliki informasi khusus tentang tujuan akhir. Di sisi lain, algoritma A* menggunakan heuristik untuk memperkirakan biaya sisa yang diperlukan untuk mencapai tujuan, yang memungkinkannya mengarahkan pencarian dengan lebih efektif. Heuristik ini membantu A* untuk mengarahkan eksplorasi ke arah yang lebih relevan dan mengurangi jumlah simpul yang harus dieksplorasi untuk mencapai solusi. Dengan demikian, algoritma A* memiliki potensi untuk menjadi lebih efisien dalam menemukan jalur terpendek dalam kasus Word Ladder dibandingkan dengan UCS.

Secara teoritis, algoritma Greedy BFS tidak dapat menjamin solusi yang optimal untuk masalah Word Ladder. Hal ini disebabkan oleh algoritma ini hanya menggunakan heuristik untuk menentukan prioritas dalam pencarian, sehingga ada risiko bahwa jalur yang dipilih bukanlah jalur terpendek. Meskipun Greedy BFS cenderung lebih cepat dalam menemukan jalur karena langsung mengarahkan eksplorasi ke arah tujuan, namun ada kemungkinan bahwa algoritma ini tidak dapat menemukan jalur terpendek jika terdapat jalur alternatif dengan biaya yang lebih rendah.

BAB III

SOURCE CODE

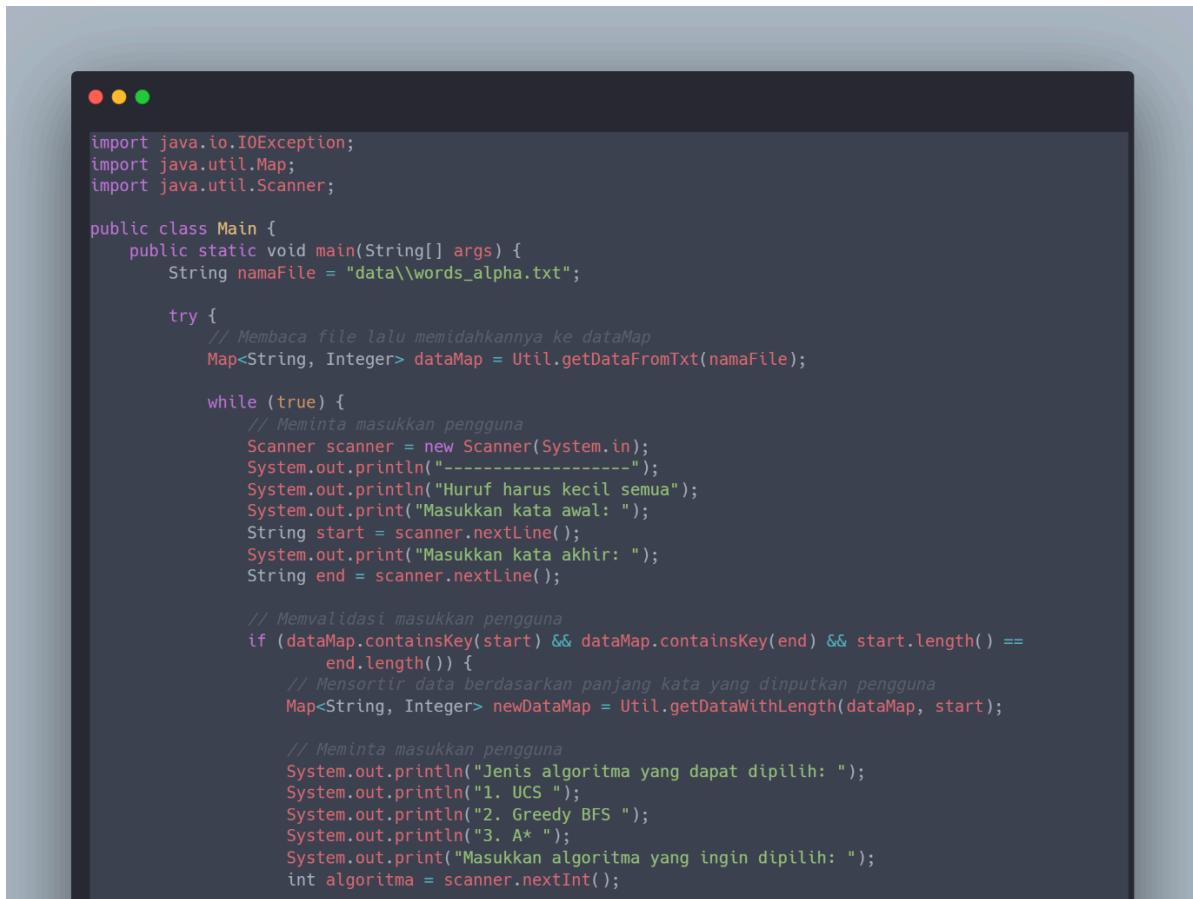
3.1 Repository Program

Repository program dapat diakses melalui tautan GitHub berikut :

https://github.com/muhhul/Tucil3_13522143

3.2 Source Code Program

3.2.1 Main.java



```
import java.io.IOException;
import java.util.Map;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        String namaFile = "data\\words_alpha.txt";

        try {
            // Membaca file lalu memidiakkannya ke dataMap
            Map<String, Integer> dataMap = Util.getDataFromTxt(namaFile);

            while (true) {
                // Meminta masukkan pengguna
                Scanner scanner = new Scanner(System.in);
                System.out.println("-----");
                System.out.println("Huruf harus kecil semua");
                System.out.print("Masukkan kata awal: ");
                String start = scanner.nextLine();
                System.out.print("Masukkan kata akhir: ");
                String end = scanner.nextLine();

                // Memvalidasi masukkan pengguna
                if (dataMap.containsKey(start) && dataMap.containsKey(end) && start.length() == end.length()) {
                    // Mensortir data berdasarkan panjang kata yang dinputkan pengguna
                    Map<String, Integer> newDataMap = Util.getDataWithLength(dataMap, start);

                    // Meminta masukkan pengguna
                    System.out.println("Jenis algoritma yang dapat dipilih: ");
                    System.out.println("1. UCS ");
                    System.out.println("2. Greedy BFS ");
                    System.out.println("3. A* ");
                    System.out.print("Masukkan algoritma yang ingin dipilih: ");
                    int algoritma = scanner.nextInt();
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

// Memvalidasi masukkan pengguna
while (true) {
    if (algoritma > 0 && algoritma < 4) {
        break;
    } else {
        System.out.println("Masukkan anda salah silahkan masukkan ulang ");
        System.out.print("Masukkan algoritma yang ingin dipilih: ");
        algoritma = scanner.nextInt();
    }
}

// Menghitung waktu dan memanggil fungsi algoritma sesuai pilihan pengguna
long startTime = System.currentTimeMillis();
if (algoritma == 1) {
    Algoritma.UCS(newDataMap, start, end);
} else if (algoritma == 2) {
    Algoritma.GBFS(newDataMap, start, end);
} else {
    Algoritma.ABINTANG(newDataMap, start, end);
}
long endTime = System.currentTimeMillis();
long executionTime = endTime - startTime;
System.out.println("Waktu Eksekusi Program: " + executionTime + " ms");

} else {
    System.out.println("Maaf masukkan anda tidak ada dalam kamus atau panjang kata tidak
        sama");
}
}

} catch (IOException e) {
    System.err.println("Gagal membaca file: " + e.getMessage());
}

}
}

```

3.2.2 Graph.java

```

import java.util.*;

class Graph {
    private final Map<String, List<String>> adjacencyList;

    // Konstruktor graph
    public Graph() {
        adjacencyList = new HashMap<>();
    }

    // Menambahkan node ke simpul
    public void addVertex(String vertex) {
        adjacencyList.putIfAbsent(vertex, new ArrayList<>());
    }

    // Menambahkan hubungan antar node
    public void addEdge(String source, String destination) {
        adjacencyList.get(source).add(destination);
    }

    // Mendapatkan tetangga dari suatu node
    public List<String> getAdjacentVertices(String vertex) {
        return adjacencyList.getOrDefault(vertex, Collections.emptyList());
    }
}

```

```

// Mencetak path dari awal ke akhir
public void printPath(String start, String end) {
    List<String> visited = new ArrayList<>();
    visited.add(start);
    System.out.println("Path: ");
    printPathUtil(start, end, visited);
}

// Fungsi rekursif untuk mendapatkan path
private void printPathUtil(String current, String end, List<String> visited)
{
    if (current.equals(end)) {
        for (String path : visited){
            if(path.equals(end)){
                System.out.println(path);
            }else{
                System.out.print(path + " -> ");
            }
        }
        return;
    }

    for (String next : getAdjacentVertices(current)) {
        if (!visited.contains(next)) {
            visited.add(next);
            printPathUtil(next, end, visited);
            visited.remove(visited.size() - 1);
        }
    }
}

```

3.2.3 Util.java

```

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

public class Util {

    // Membaca data dari txt lalu dipindahkan ke dataMap
    public static Map<String, Integer> getDataFromTxt(String namaFile) throws IOException {
        Map<String, Integer> dataMap = new HashMap<>();
        try (BufferedReader reader = new BufferedReader(new FileReader(namaFile))) {
            String line;
            while ((line = reader.readLine()) != null) {
                String isiBaris = line.trim();
                dataMap.put(isiBaris, 0);
            }
        }
        return dataMap;
    }
}

```

```

// Mengembalikan data yang sudah di seleksi berdasarkan syarat
public static Map<String, Integer> getDataForQueue(Map<String, Integer> kataMap, String kataCek,
    String kataFinal, int panjangKataCek, int mode, int depth) {
    Map<String, Integer> kataYangSama = new HashMap<>();

    for (Map.Entry<String, Integer> entry : kataMap.entrySet()) {
        String kata = entry.getKey();
        int panjangKata = entry.getKey().length();

        // Memvalidasi apakah kata sekarang panjangnya sama dengan kata yang akan di cek
        if (panjangKata == panjangKataCek && !kata.equals(kataCek)) {
            int hurufSama = 0;
            for (int i = 0; i < panjangKataCek; i++) {
                if (kata.charAt(i) == kataCek.charAt(i)) {
                    hurufSama++;
                }
            }

            // Memvalidasi untuk algoritma UCS
            if (mode == 1) {
                if (hurufSama == (panjangKata - 1)) {
                    if (kata.equals(kataFinal)) {
                        Map<String, Integer> kataYangSamal = new HashMap<>();
                        kataYangSamal.put(kata, 0);
                        return kataYangSamal;
                    }
                    kataYangSamal.put(kata, depth + 1);
                }
            }
        }

        // Memvalidasi untuk algoritma Greedy BFS
        else if (mode == 2) {
            if (hurufSama == (panjangKata - 1)) {
                if (kata.equals(kataFinal)) {
                    Map<String, Integer> kataYangSamal = new HashMap<>();
                    kataYangSamal.put(kata, 0);
                    return kataYangSamal;
                }
                hurufSama = 0;
                for (int i = 0; i < panjangKataCek; i++) {
                    if (kata.charAt(i) == kataFinal.charAt(i)) {
                        hurufSama++;
                    }
                }
                kataYangSamal.put(kata, panjangKata - hurufSama);
            }
        }

        // Memvalidasi untuk algoritma A*
        else {
            if (hurufSama == (panjangKata - 1)) {
                if (kata.equals(kataFinal)) {
                    Map<String, Integer> kataYangSamal = new HashMap<>();
                    kataYangSamal.put(kata, 0);
                    return kataYangSamal;
                }
                hurufSama = 0;
                for (int i = 0; i < panjangKataCek; i++) {
                    if (kata.charAt(i) == kataFinal.charAt(i)) {
                        hurufSama++;
                    }
                }
                kataYangSamal.put(kata, panjangKata - hurufSama + depth);
            }
        }
    }
    return kataYangSama;
}

```

```
// Mengembalikan data yang sudah di seleksi berdasarkan panjang kata
public static Map<String, Integer> getDataWithLength(Map<String, Integer> kataMap, String kataCek)
{
    Map<String, Integer> kataYangSama = new HashMap<>();

    for (Map.Entry<String, Integer> entry : kataMap.entrySet()) {
        String kata = entry.getKey();
        int panjangKata = entry.getKey().length();

        // Memvalidasi panjang kata
        if (panjangKata == kataCek.length()) {
            kataYangSama.put(kata, 0);
        }
    }
    return kataYangSama;
}
}
```

3.2.4 Algoritma.java

```
● ● ●

import java.util.AbstractMap;
import java.util.HashMap;
import java.util.Map;
import java.util.PriorityQueue;

public class Algoritma {
```

```

// Algoritma mendapatkan path dengan UCS
public static void UCS(Map<String, Integer> dataMap, String awal, String akhir) {
    PriorityQueue<Map.Entry<String, Integer>> mapPriorityQueue = new PriorityQueue<>(
        (a, b) -> a.getValue() - b.getValue());
    Map.Entry<String, Integer> entry = new AbstractMap.SimpleEntry<>(awal, 0);
    mapPriorityQueue.add(entry);
    Map<String, Boolean> visited = new HashMap<>();
    visited.put(awal, true);
    Graph graph = new Graph();
    graph.addVertex(awal);
    int count=0;
    Boolean found=false;

    // Akan melakukan perulangan untuk mengecek node berdasarkan prio queue
    while (!mapPriorityQueue.isEmpty()) {

        // Mengambil seluruh node yang bertetangga dengan current node
        count++;
        Map.Entry<String, Integer> temp = mapPriorityQueue.poll();
        String currentQueue = temp.getKey();
        Map<String, Integer> dataMap2 = Util.getDataForQueue(dataMap, currentQueue,
            akhir, awal.length(), 1, temp.getValue());

        // Memvalidasi apakah ada node yang merupakan kata akhir
        for (Map.Entry<String, Integer> input : dataMap2.entrySet()) {

            // Jika kata akhir ditemukan
            if (input.getKey().equals(akhir)) {
                mapPriorityQueue.clear();
                graph.addVertex(input.getKey());
                graph.addEdge(currentQueue, input.getKey());
                graph.printPath(awal, input.getKey());
                System.out.println("Banyak node yang dikunjungi: "+count);
                found=true;
                break;
            }
            if (!visited.containsKey(input.getKey())) {
                visited.put(input.getKey(), true);
                mapPriorityQueue.add(input);
                graph.addVertex(input.getKey());
                graph.addEdge(currentQueue, input.getKey());
            }
        }
    }

    // Jika kata akhir tidak ditemukan
    if (found==false){
        System.out.println("Tidak ada path yang dapat dilalui");
        System.out.println("Banyak node yang dikunjungi: "+count);
    }
}

```

```

// Algoritma mendapatkan path dengan Greedy BFS
public static void GBFS(Map<String, Integer> dataMap, String awal, String akhir) {
    PriorityQueue<Map.Entry<String, Integer>> mapPriorityQueue = new PriorityQueue<>(
        (a, b) -> a.getValue() - b.getValue());
    Map.Entry<String, Integer> entry = new AbstractMap.SimpleEntry<>(awal, 0);
    mapPriorityQueue.add(entry);
    Map<String, Boolean> visited = new HashMap<>();
    visited.put(awal, true);
    Graph graph = new Graph();
    graph.addVertex(awal);
    int count = 0;
    Boolean found=false;

    // Akan melakukan perulangan untuk mengecek node berdasarkan prio queue
    while (!mapPriorityQueue.isEmpty()) {

        // Mengambil seluruh node yang bertetangga dengan current node
        count++;
        Map.Entry<String, Integer> temp = mapPriorityQueue.poll();
        mapPriorityQueue.clear();
        String currentQueue = temp.getKey();
        Map<String, Integer> dataMap2 = Util.getDataForQueue(dataMap, currentQueue,
            akhir, awal.length(), 2, temp.getValue());

        // Memvalidasi apakah ada node yang merupakan kata akhir
        for (Map.Entry<String, Integer> input : dataMap2.entrySet()) {

            // Jika kata akhir ditemukan
            if (input.getKey().equals(akhir)) {
                mapPriorityQueue.clear();
                graph.addVertex(input.getKey());
                graph.addEdge(currentQueue, input.getKey());
                graph.printPath(awal, input.getKey());
                System.out.println("Banyak node yang dikunjungi: "+count);
                found=true;
                break;
            }
            if (!visited.containsKey(input.getKey())) {
                visited.put(input.getKey(), true);
                mapPriorityQueue.add(input);
                graph.addVertex(input.getKey());
                graph.addEdge(currentQueue, input.getKey());
            }
        }
    }

    // Jika kata akhir tidak ditemukan
    if (found==false){
        System.out.println("Tidak ada path yang dapat dilalui");
        System.out.println("Banyak node yang dikunjungi: "+count);
    }
}

```

```

// Algoritma mendapatkan path dengan A*
public static void ABINTANG(Map<String, Integer> dataMap, String awal, String akhir) {
    PriorityQueue<Map.Entry<String, Integer>> mapPriorityQueue = new PriorityQueue<>(
        (a, b) -> a.getValue() - b.getValue());
    Map.Entry<String, Integer> entry = new AbstractMap.SimpleEntry<>(awal, 0);
    mapPriorityQueue.add(entry);
    Map<String, Integer> visited = new HashMap<>();
    visited.put(awal, 0);
    Graph graph = new Graph();
    graph.addVertex(awal);
    int count = 0;
    Boolean found=false;

    // Akan melakukan perulangan untuk mengecek node berdasarkan prio queue
    while (!mapPriorityQueue.isEmpty()) {
        // Mengambil seluruh node yang bertetangga dengan current node
        count++;
        Map.Entry<String, Integer> temp = mapPriorityQueue.poll();
        String currentQueue = temp.getKey();
        Map<String, Integer> dataMap2 = Util.getDataForQueue(dataMap, currentQueue,
            akhir, awal.length(), 3, visited.get(currentQueue));

        // Memvalidasi apakah ada node yang merupakan kata akhir
        for (Map.Entry<String, Integer> input : dataMap2.entrySet()) {

            // Jika kata akhir ditemukan
            if (input.getKey().equals(akhir)) {
                mapPriorityQueue.clear();
                graph.addVertex(input.getKey());
                graph.addEdge(currentQueue, input.getKey());
                graph.printPath(awal, input.getKey());
                System.out.println("Banyak node yang dikunjungi: "+count);
                found=true;
                break;
            }
            if (!visited.containsKey(input.getKey())) {
                visited.put(input.getKey(), visited.get(currentQueue) + 1);
                mapPriorityQueue.add(input);
                graph.addVertex(input.getKey());
                graph.addEdge(currentQueue, input.getKey());
            }
        }
    }

    // Jika kata akhir tidak ditemukan
    if (found==false){
        System.out.println("Tidak ada path yang dapat dilalui");
        System.out.println("Banyak node yang dikunjungi: "+count);
    }
}
}

```

BAB IV

TESTING PROGRAM DAN ANALISIS HASIL

Pada pengetesan percobaan kali ini, data kamus bahasa inggris yang digunakan merujuk pada kamus ini

<https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/dictionary.txt>

4.1 Percobaan 1

```
Huruf harus kecil semua
Masukkan kata awal: meats
Masukkan kata akhir: lover
Jenis algoritma yang dapat dipilih:
1. UCS
2. Greedy BFS
3. A*
Masukkan algoritma yang ingin dipilih: 1
Path:
meats -> moats -> motts -> motes -> moter -> mover -> lover
Banyak node yang dikunjungi: 2605
Waktu Eksekusi Program: 4575 ms
```

```
Huruf harus kecil semua
Masukkan kata awal: meats
Masukkan kata akhir: lover
Jenis algoritma yang dapat dipilih:
1. UCS
2. Greedy BFS
3. A*
Masukkan algoritma yang ingin dipilih: 2
Path:
meats -> moats -> doats -> dorts -> torts -> tores -> lores -> loves -> lover
Banyak node yang dikunjungi: 8
Waktu Eksekusi Program: 23 ms
```

```
Huruf harus kecil semua
Masukkan kata awal: meats
Masukkan kata akhir: lover
Jenis algoritma yang dapat dipilih:
1. UCS
2. Greedy BFS
3. A*
Masukkan algoritma yang ingin dipilih: 3
Path:
meats -> moats -> molts -> moles -> moves -> loves -> lover
Banyak node yang dikunjungi: 23
Waktu Eksekusi Program: 85 ms
```

4.2 Percobaan 2

```
Huruf harus kecil semua
Masukkan kata awal: hit
Masukkan kata akhir: cap
Jenis algoritma yang dapat dipilih:
1. UCS
2. Greedy BFS
3. A*
Masukkan algoritma yang ingin dipilih: 1
Path:
hit -> hip -> hap -> cap
Banyak node yang dikunjungi: 50
Waktu Eksekusi Program: 8 ms
```

```
Huruf harus kecil semua
Masukkan kata awal: hit
Masukkan kata akhir: cap
Jenis algoritma yang dapat dipilih:
1. UCS
2. Greedy BFS
3. A*
Masukkan algoritma yang ingin dipilih: 2
Path:
hit -> hip -> hap -> cap
Banyak node yang dikunjungi: 3
Waktu Eksekusi Program: 1 ms
```

```
Huruf harus kecil semua
Masukkan kata awal: hit
Masukkan kata akhir: cap
Jenis algoritma yang dapat dipilih:
1. UCS
2. Greedy BFS
3. A*
Masukkan algoritma yang ingin dipilih: 3
Path:
hit -> hip -> hap -> cap
Banyak node yang dikunjungi: 4
Waktu Eksekusi Program: 2 ms
```

4.3 Percobaan 3

```
Huruf harus kecil semua
Masukkan kata awal: long
Masukkan kata akhir: push
Jenis algoritma yang dapat dipilih:
1. UCS
2. Greedy BFS
3. A*
Masukkan algoritma yang ingin dipilih: 1
Path:
long -> lone -> pone -> pose -> posh -> push
Banyak node yang dikunjungi: 1792
Waktu Eksekusi Program: 1196 ms
```

```
Huruf harus kecil semua
Masukkan kata awal: long
Masukkan kata akhir: push
Jenis algoritma yang dapat dipilih:
1. UCS
2. Greedy BFS
3. A*
Masukkan algoritma yang ingin dipilih: 2
Path:
long -> pong -> pung -> puny -> puly -> pule -> pume -> pump -> hump -> hums -> huss -> puss -> push
Banyak node yang dikunjungi: 12
Waktu Eksekusi Program: 12 ms
```

```
Huruf harus kecil semua
Masukkan kata awal: long
Masukkan kata akhir: push
Jenis algoritma yang dapat dipilih:
1. UCS
2. Greedy BFS
3. A*
Masukkan algoritma yang ingin dipilih: 3
Path:
long -> pong -> pung -> puns -> puss -> push
Banyak node yang dikunjungi: 46
Waktu Eksekusi Program: 39 ms
```

4.4 Percobaan 4

```
Huruf harus kecil semua
Masukkan kata awal: table
Masukkan kata akhir: happy
Jenis algoritma yang dapat dipilih:
1. UCS
2. Greedy BFS
3. A*
Masukkan algoritma yang ingin dipilih: 1
Path:
table -> sable -> sably -> saily -> haily -> haply -> happy
Banyak node yang dikunjungi: 736
Waktu Eksekusi Program: 1355 ms
```

```
Huruf harus kecil semua
Masukkan kata awal: table
Masukkan kata akhir: happy
Jenis algoritma yang dapat dipilih:
1. UCS
2. Greedy BFS
3. A*
Masukkan algoritma yang ingin dipilih: 2
Path:
table -> hable -> harle -> harls -> harps -> harpy -> happy
Banyak node yang dikunjungi: 6
Waktu Eksekusi Program: 29 ms
```

```
Huruf harus kecil semua
Masukkan kata awal: table
Masukkan kata akhir: happy
Jenis algoritma yang dapat dipilih:
1. UCS
2. Greedy BFS
3. A*
Masukkan algoritma yang ingin dipilih: 3
Path:
table -> hable -> harle -> harls -> harps -> harpy -> happy
Banyak node yang dikunjungi: 28
Waktu Eksekusi Program: 63 ms
```

4.5 Percobaan 5

```
Huruf harus kecil semua
Masukkan kata awal: make
Masukkan kata akhir: test
Jenis algoritma yang dapat dipilih:
1. UCS
2. Greedy BFS
3. A*
Masukkan algoritma yang ingin dipilih: 1
Path:
make -> mane -> mene -> ment -> tent -> test
Banyak node yang dikunjungi: 1707
Waktu Eksekusi Program: 1116 ms
```

```
Huruf harus kecil semua
Masukkan kata awal: make
Masukkan kata akhir: test
Jenis algoritma yang dapat dipilih:
1. UCS
2. Greedy BFS
3. A*
Masukkan algoritma yang ingin dipilih: 2
Path:
make -> take -> takt -> tact -> fact -> fast -> fest -> test
Banyak node yang dikunjungi: 7
Waktu Eksekusi Program: 7 ms
```

```
Huruf harus kecil semua
Masukkan kata awal: make
Masukkan kata akhir: test
Jenis algoritma yang dapat dipilih:
1. UCS
2. Greedy BFS
3. A*
Masukkan algoritma yang ingin dipilih: 3
Path:
make -> maku -> masu -> mast -> mest -> test
Banyak node yang dikunjungi: 33
Waktu Eksekusi Program: 24 ms
```

4.6 Percobaan 6

```
Huruf harus kecil semua
Masukkan kata awal: leak
Masukkan kata akhir: hope
Jenis algoritma yang dapat dipilih:
1. UCS
2. Greedy BFS
3. A*
Masukkan algoritma yang ingin dipilih: 1
Path:
leak -> leap -> heap -> hemp -> heme -> home -> hope
Banyak node yang dikunjungi: 1305
Waktu Eksekusi Program: 508 ms
```

```
Huruf harus kecil semua
Masukkan kata awal: leak
Masukkan kata akhir: hope
Jenis algoritma yang dapat dipilih:
1. UCS
2. Greedy BFS
3. A*
Masukkan algoritma yang ingin dipilih: 2
Path:
leak -> beak -> beam -> berm -> herm -> here -> hire -> hive -> hove -> hope
Banyak node yang dikunjungi: 9
Waktu Eksekusi Program: 13 ms
```

```
Huruf harus kecil semua
Masukkan kata awal: leak
Masukkan kata akhir: hope
Jenis algoritma yang dapat dipilih:
1. UCS
2. Greedy BFS
3. A*
Masukkan algoritma yang ingin dipilih: 3
Path:
leak -> lean -> loan -> lown -> lowe -> howe -> hope
Banyak node yang dikunjungi: 93
Waktu Eksekusi Program: 33 ms
```

4.7 Analisis Hasil Optimasi

UCS menjamin solusi optimal dengan memprioritaskan node yang memiliki biaya terendah untuk diekspansi, seperti dalam Word Ladder di mana biaya adalah jumlah langkah. Namun, UCS dapat menjadi lambat jika terdapat banyak jalur dengan biaya serupa karena

algoritma ini mempertimbangkan semua kemungkinan jalur. Sedangkan *Greedy* BFS cenderung tidak menjamin solusi optimal karena hanya mengandalkan heuristik dan tidak mempertimbangkan biaya langkah secara menyeluruh, berpotensi memilih jalur suboptimal dalam Word Ladder. Sedangkan A* menawarkan solusi optimal dengan menggabungkan biaya langkah dan heuristik, misalnya, jumlah karakter yang berbeda antara kata saat ini dan kata akhir dalam Word Ladder. A* lebih efisien daripada UCS karena dapat mengarahkan pencarian ke arah tujuan.

4.8 Analisis Hasil Waktu Eksekusi

UCS dapat memiliki waktu eksekusi yang lambat karena algoritma ini secara eksplisit mengeksplorasi semua jalur yang mungkin berdasarkan biaya langkah. Dalam situasi Word Ladder di mana terdapat banyak jalur dengan biaya serupa, UCS mungkin menjadi kurang efisien. Meskipun demikian, UCS menjamin solusi optimal, namun terkadang menjadi lebih lambat dibandingkan dengan algoritma lain yang mungkin memilih jalur dengan eksplorasi yang lebih terbatas. Sedangkan *Greedy* BFS cenderung memiliki waktu eksekusi yang lebih cepat karena hanya mengandalkan heuristik untuk menentukan prioritas. Algoritma ini dapat menemukan jalur dengan cepat karena mengarahkan eksplorasi ke arah yang tampaknya paling dekat dengan tujuan. Namun, *Greedy* BFS berisiko menghasilkan solusi suboptimal, yang berarti mungkin memerlukan lebih banyak waktu untuk menemukan jalur yang sebenarnya lebih pendek jika jalur yang dipilih tidak optimal. Sedangkan A* biasanya lebih cepat daripada UCS karena memanfaatkan heuristik untuk memperkirakan biaya menuju tujuan. Dengan informasi tambahan dari heuristik, A* cenderung mengarahkan pencarian ke jalur yang lebih menjanjikan, mengurangi eksplorasi yang tidak perlu. Waktu eksekusi A* dapat bervariasi tergantung pada kualitas heuristik yang digunakan, namun dalam konteks Word Ladder, A* biasanya lebih efisien dibandingkan UCS.

4.9 Analisis Hasil Memori yang Digunakan

UCS dapat memiliki penggunaan memori yang tinggi karena algoritma ini membutuhkan penyimpanan semua node yang diekspansi dalam struktur data seperti priority queue. Karena UCS mengeksplorasi semua kemungkinan jalur berdasarkan biaya, penggunaan memori dapat meningkat secara signifikan jika terdapat banyak jalur potensial yang harus dipertimbangkan. Sedangkan *Greedy* BFS memiliki kecenderungan untuk menggunakan memori dengan lebih efisien karena hanya mengandalkan heuristik untuk menentukan prioritas. Dengan fokus pada eksplorasi jalur yang tampaknya paling dekat

dengan tujuan, *Greedy* BFS dapat mengurangi jumlah node yang perlu disimpan dan diekspansi, yang berarti mengurangi penggunaan memori secara keseluruhan. Sedangkan A* cenderung lebih efisien dalam penggunaan memori jika dibandingkan dengan UCS karena algoritma ini menggunakan heuristik untuk mengurangi eksplorasi yang tidak perlu. Dengan memperhitungkan heuristik, A* dapat mengurangi jumlah node yang perlu diekspansi, sehingga mengurangi penggunaan memori secara keseluruhan.

BAB V

LAMPIRAN

1.

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		✓