

TUGAS KECIL 3

IMPLEMENTASI ALGORITMA UCS, GREEDY BEST SEARCH, DAN A* UNTUK PENYELESAIAN PERMAINAN WORD LADDER

IF2211 - STRATEGI ALGORITMA



Disusun Oleh:

Debrina Veisha Rashika W 13522025

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

DAFTAR ISI

DAFTAR ISI	1
BAB I DESKRIPSI PERSOALAN	2
BAB II ANALISIS ALGORITMA	3
2.1 Fungsi Cost yang Digunakan	3
2.2. Deskripsi Algoritma UCS	3
2.3. Penerapan Algoritma UCS	3
2.4. Deskripsi Algoritma Greedy Best First Search	4
2.5. Penerapan Algoritma Greedy Best First Search	4
2.6. Deskripsi Algoritma A*	4
2.7. Admissibility Heuristik pada Algoritma A*	5
2.8. Penerapan Algoritma A*	5
BAB III IMPLEMENTASI	6
3.1. Kelas dan Method	6
3.2. Rancangan GUI	22
3.2.1 Tampilan Awal	23
3.2.2. Tampilan Input Tidak Valid	23
3.2.3. Tampilan Masukan Input	24
3.2.4. Tampilan Output	24
BAB IV HASIL PENGUJIAN DAN PEMBAHASAN	25
4.1 Hasil Uji 1	25
4.2 Hasil Uji 2	26
4.3 Hasil Uji 3	28
4.4 Hasil Uji 4	29
4.5 Hasil Uji 5	31
4.6 Hasil Uji 6	32
4.7 Hasil Analisis	34
BAB V KESIMPULAN	36
5.1. Kesimpulan	36
LAMPIRAN	37
A. Repository Github	37
B. Checklist	37

BAB I DESKRIPSI PERSOALAN

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.

How To Play

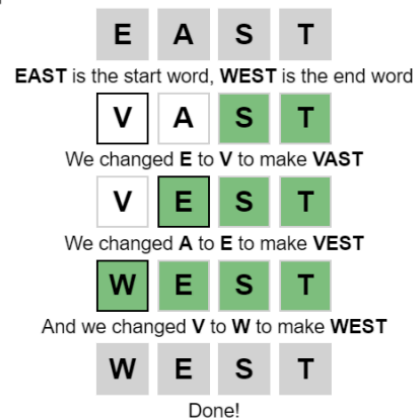
This game is called a "word ladder" and was invented by Lewis Carroll in 1877.

Rules

Weave your way from the start word to the end word.

Each word you enter **can only change 1 letter** from the word above it.

Example



Gambar 1. Ilustrasi dan Peraturan Permainan Word Ladder

Pada Tugas Kecil kali ini, diberikan tugas untuk menemukan solusi paling optimal dalam menyelesaikan permainan word ladder menggunakan algoritma UCS, *Greedy Best First Search*, dan *A** dengan menggunakan bahasa pemrograman Java berbasis CLI (*Command Line Interface*).

BAB II ANALISIS ALGORITMA

2.1 Fungsi Cost yang Digunakan

Berikut penjelasan lebih lanjut terkait nilai cost yang akan digunakan dalam menyelesaikan kasus *word ladder*:

- Fungsi $g(n)$ adalah biaya sejauh ini atau banyak perubahan yang sudah dilalui dari node awal hingga node saat ini (current node). Cost ini digunakan dalam melakukan pencarian secara UCS.
- Fungsi $h(n)$ adalah nilai heuristik yang merupakan estimasi biaya dari node saat ini ke node tujuan (goal node). Nilai heuristik didapatkan dari perbedaan huruf node saat ini dengan node tujuan. Cost ini digunakan untuk melakukan pencarian secara G-BFS.
- Fungsi $f(n)$ merupakan fungsi evaluasi yang merupakan jumlah biaya sejauh ini ($g(n)$), ditambah dengan estimasi biaya tersisa ke tujuan ($h(n)$). Secara matematis, $f(n) = g(n) + h(n)$. Cost ini digunakan untuk melakukan pencarian menggunakan algoritma A^* .

2.2. Deskripsi Algoritma UCS

Uniform-Cost Search adalah algoritma pencarian tanpa informasi (*uninformed*) yang menggunakan biaya terendah secara kumulatif untuk menemukan jalur dari satu node sumber ke node tujuan. Algoritma ini beroperasi di ruang pencarian berbobot terarah untuk berpindah dari node awal ke salah satu node akhir dengan biaya akumulasi minimum. Algoritma ini termasuk dalam kategori pencarian tanpa informasi karena menggunakan pendekatan brute force, tanpa mempertimbangkan informasi spesifik tentang node atau ruang pencarian. Tujuan utamanya adalah mencari jalur dengan biaya total terendah dalam graf berbobot, di mana setiap ekspansi node didasarkan pada biaya traversal dari node akar.

Algoritma Uniform-Cost Search sering diimplementasikan dengan menggunakan priority queue, di mana prioritasnya adalah untuk mengeksekusi operasi dengan biaya terendah terlebih dahulu. Dengan menyisipkan simpul sumber terlebih dahulu dan kemudian memasukkan simpul lainnya satu per satu sesuai kebutuhan. Setiap iterasi dilakukan untuk memeriksa apakah suatu item sudah ada dalam antrian prioritas. Jika belum, maka item tersebut dimasukkan ke dalam antrian prioritas.

2.3. Penerapan Algoritma UCS

1. Inisiasi Priority Queue berdasarkan cost $g(n)$, yakni jumlah pergantian huruf yang telah dilakukan.
2. Letakkan simpul awal pada Priority Queue.
3. Dequeue Priority Queue.
4. Cek apakah currentNode hasil Dequeue merupakan target.
5. Jika currentNode adalah target, pencarian dihentikan dan rute menuju target diperoleh.
6. Jika tidak. Enqueue Priority Queue dengan rute menuju setiap child node yang belum pernah dikunjungi.
7. Ulangi langkah 3 - 6 hingga ditemukan node target.

2.4. Deskripsi Algoritma *Greedy Best First Search*

Algoritma Greedy Best-First Search adalah metode pencarian yang memilih keputusan berdasarkan informasi terbaik yang tersedia pada saat itu, tanpa mempertimbangkan konsekuensi di masa depan. Dalam konteks maksimisasi, ini berarti memilih opsi terbesar, sedangkan dalam konteks minimisasi, ini berarti memilih opsi terkecil. Tujuannya adalah mencapai solusi terbaik dengan mengambil keputusan saat ini tanpa mempertimbangkan perubahan di masa mendatang. Keputusan yang diambil tidak dapat dibatalkan.

Dalam Greedy BFS, keputusan didasarkan pada fungsi evaluasi $f(n)$ yang menggunakan nilai heuristik $h(n)$, tanpa memperhatikan nilai sebenarnya $g(n)$. Greedy Search memilih simpul berdasarkan nilai heuristik $h(n)$ dengan aturan tertentu untuk memilih simpul yang dianggap terbaik pada setiap langkah pencarian. Dalam menyelesaikan kasus ini, digunakan greedy by different letter sebagai nilai dari $h(n)$ untuk menentukan langkah selanjutnya dalam proses pencarian.

Perbedaan dari UCS adalah pada BFS node dibangkitkan berdasarkan nilai cost $h(n)$ yang paling optimal sedangkan pada UCS node dibangkitkan berdasarkan nilai cost $g(n)$. UCS cenderung menelusuri hampir semua kemungkinan jalur sehingga merupakan algoritma yang *complete* sedangkan BFS hanya menelusuri jalur dengan cost $h(n)$ yang paling optimal sehingga jalur yang dihasilkan bisa berbeda antara UCS dengan BFS dan jalur yang dihasilkan oleh BFS belum tentu merupakan jalur yang paling optimal karena proses pencariannya tidak menyeluruh (*incomplete*).

2.5. Penerapan Algoritma *Greedy Best First Search*

1. Inisiasi Priority Queue berdasarkan cost $h(n)$, yakni cost heuristic yang didapat dari perbedaan huruf (greedy by different letter) yang digunakan untuk menyimpan simpul aktif.
2. Letakkan simpul awal pada Priority Queue.
3. Dequeue Priority Queue.
4. Cek apakah currentNode hasil Dequeue merupakan target.
5. Jika currentNode adalah target, pencarian dihentikan dan rute menuju target diperoleh.
6. Jika tidak, expand node dan simpan hasil node yang belum pernah dikunjungi pada priority queue.
7. Ulangi langkah 3 - 6 hingga ditemukan node target.

2.6. Deskripsi Algoritma A*

Algoritma A* (A Star) adalah algoritma pencarian yang digunakan untuk mencari jalur terpendek antara dua titik, yang sering diterapkan pada pemetaan untuk menemukan jalur terpendek. Algoritma ini bekerja dengan mencari jalur yang paling efisien terlebih dahulu, sehingga dianggap dapat mencari jalur secara optimal dan lengkap. Algoritma optimal menghasilkan solusi paling ekonomis dalam hal biaya, sementara algoritma lengkap akan menemukan semua solusi yang memungkinkan.

Salah satu kekuatan utama A* adalah penggunaan grafik berbobot dan nilai heuristik. Grafik berbobot menggambarkan biaya setiap jalur atau tindakan, memungkinkan algoritma memilih rute dengan biaya minimal untuk mencapai tujuan dalam hal jarak atau waktu. Penggunaan nilai heuristik, sebagai bagian dari pencarian berinformasi (*informed*), membantu mengarahkan pencarian ke arah yang lebih potensial menuju tujuan dengan mengurangi jumlah node yang perlu dikunjungi.

Secara teoritis, Algoritma A* dapat lebih efisien dibandingkan dengan UCS dalam penyelesaian Word Ladder jika heuristik yang digunakan ($h(n)$) adalah konsisten (admissible) dan memberikan estimasi biaya yang akurat ke tujuan. Heuristik yang baik dapat mengarahkan A* untuk mengeksplorasi jalur-jalur yang lebih menjanjikan terlebih dahulu, mengurangi jumlah node yang perlu dikunjungi.

2.7. Admissibility Heuristik pada Algoritma A*

Sebuah heuristik dikatakan admissible jika estimasi biaya yang diberikan tidak pernah melebihi biaya sebenarnya untuk mencapai tujuan. Dengan kata lain, $h(n)$ adalah admissible jika $h(n) \leq$ biaya sebenarnya (actual cost) untuk mencapai tujuan dari node saat ini. Berikut beberapa contoh perbandingan cost:

Simpul	Simpul Tujuan	$h(n)$	$h^*(n)$	Admissible?
earn	deny	4	5	yes
boss	bowl	2	2	yes
boars	bored	3	4	yes

2.8. Penerapan Algoritma A*

1. Inisiasi Priority Queue berdasarkan cost $f(n)$.
 $g(n) = \text{total penggantian huruf dari akar ke simpul goal}$
 $h(n) = \text{beda huruf currentNode dengan simpul goal}$
 $f(n) = g(n) + h(n) : f(n) \text{ adalah prioritas simpul } n$
2. Letakkan simpul awal pada Priority Queue.
3. Dequeue Priority Queue.
4. Cek apakah currentNode hasil Dequeue merupakan target.
5. Jika currentNode adalah target, pencarian dihentikan dan rute menuju target diperoleh.
6. Jika tidak, Enqueue Priority Queue dengan rute menuju setiap child node yang belum pernah dikunjungi.
7. Ulangi langkah 3 - 6 hingga ditemukan node target.

BAB III IMPLEMENTASI

3.1. Kelas dan Method

1. Class UCS

UCS.java	
atribut	<ul style="list-style-type: none"> static Set<String> visited {menyimpan node yang telah dilalui}
Method	<ul style="list-style-type: none"> public static ArrayList<String> solveUCS() { method utama untuk mencari solusi menggunakan algoritma UCS dan mengembalikan path yang ditemukan dalam bentuk Array of string }
Implementasi	
<pre> public class UCS { static Set<String> visited = new HashSet<>(); // override kelas comparator untuk mengurutkan array berdasarkan cost path public static class PathsComparator implements Comparator<Paths> { @Override public int compare(Paths p1, Paths p2) { return Integer.compare(p1.getCost(), p2.getCost()); } } // method untuk mencari path menggunakan algoritma UCS public static ArrayList<String> solveUCS() { // hapus list visited setiap memulai baru visited.clear(); // Executor agar program bisa berjalan secara paralel ExecutorService executor = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors()); // menyiapkan priority queue dan map untuk menyimpan path PriorityQueue<Paths> pq = new PriorityQueue<>(new PathsComparator()); CompletionService<Void> completionService = new ExecutorCompletionService<>(executor); Map<String, Integer> minCostToNode = new HashMap<>(); // menyimpan start path ke priority queue Node startNode = new Node(Input.startinput); </pre>	

```
Paths startPath = new Paths(startNode, new ArrayList<>(), 0);
pq.add(startPath);
minCostToNode.put(startNode.getWord(), 0);

// melakukan iterasi selama prioqueue belum kosong
while (!pq.isEmpty()) {
    // dequeue prioqueue dan jadikan currentpath
    Paths currentPath = pq.poll();
    Node currentNode = currentPath.getCurrNode();

    // jika node saat ini adalah goal maka return hasil
    if (currentNode.getWord().equals(Input.targetinput)) {
        System.out.println("UCS FOUND!!!");
        System.out.println("End Node: " + currentNode.getWord());
        System.out.println("Total Cost: " + currentPath.getCost());
        System.out.println("length: " + currentPath.getpath().size());
        System.out.println("Path: " + currentPath.getpath());
        currentPath.getpath().add(currentNode.getWord());
        executor.shutdown();
        return currentPath.getpath();
    }

    // jika bukan, expand node yang belum dikunjungi
    if (!visited.contains(currentNode.getWord())) {
        visited.add(currentNode.getWord());

        currentNode.expandNode();
        List<Callable<Void>> tasks = new ArrayList<>();
        // tambahkan node tetangga yang belum pernah dikunjungi pada prioqueue
        for (Node neighbor : currentNode.getNeighbour()) {
            tasks.add(() -> {
                // hitung cost yang ada pada masing masing node
                int newCost = currentPath.getCost() + neighbor.getcostUCS();
                synchronized (minCostToNode) {
                    if (!minCostToNode.containsKey(neighbor.getWord()) ||
newCost < minCostToNode.get(neighbor.getWord())) {
                        minCostToNode.put(neighbor.getWord(), newCost);
                        ArrayList<String> newRoute = new
ArrayList<>(currentPath.getpath());
                        newRoute.add(currentNode.getWord());
                        Paths newPath = new Paths(neighbor, newRoute, newCost);
                        pq.add(newPath); // tambah path ke prioqueue
                    }
                }
            });
        }
        return null;
    }
}

for (Callable<Void> task : tasks) {
```



```

        completionService.submit(task);
    }

    // tunggu semua task selesai sebelum lanjut ke iterasi selanjutnya
    try {
        for (int i = 0; i < tasks.size(); i++) {
            completionService.take().get();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

}
executor.shutdown();
return new ArrayList<String>();
}
}

```

2. Class BFS

BFS.java	
atribut	<ul style="list-style-type: none"> static Set<String> visited {menyimpan node yang telah dilalui}
Method	<ul style="list-style-type: none"> public static ArrayList<String> solveBFS() { method utama untuk mencari solusi menggunakan algoritma BFS dan mengembalikan path yang ditemukan dalam bentuk Array of string }
Implementasi	
<pre> public class BFS { static Set<String> visited = new HashSet<>(); // override kelas comparator untuk mengurutkan array berdasarkan cost path public static class PathsComparator implements Comparator<Paths> { @Override public int compare(Paths p1, Paths p2) { return Integer.compare(p1.getCost(), p2.getCost()); } } // method untuk mencari path menggunakan algoritma BFS public static ArrayList<String> solveBFS() { </pre>	

```
// hapus list visited setiap memulai baru
visited.clear();

// Executor agar program bisa berjalan secara paralel
ExecutorService executor =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

// menyiapkan priority queue untuk menyimpan node yang sudah diexpand
PriorityQueue<Paths> pq = new PriorityQueue<>(new PathsComparator());
CompletionService<Void> completionService = new
ExecutorCompletionService<>(executor);
Map<String, Integer> minCostToNode = new HashMap<>();

// menyimpan start path ke priority queue
Node startNode = new Node(Input.startinput);
Paths startPath = new Paths(startNode, new ArrayList<>(), 0);
pq.add(startPath);
minCostToNode.put(startNode.getWord(), 0);

// melakukan iterasi selama prioqueue belum kosong
while (!pq.isEmpty()) {

    // dequeue prioqueue dan jadikan currentpath
    Paths currentPath = pq.poll();
    Node currentNode = currentPath.getCurrNode();

    // jika node saat ini adalah goal maka return hasil
    if (currentNode.getWord().equals(Input.targetinput)) {
        System.out.println("Astar FOUND!!!");
        System.out.println("End Node: " + currentNode.getWord());
        System.out.println("Total Cost: " + currentPath.getCost());
        System.out.println("length: " + currentPath.getpath().size());
        System.out.println("Path: " + currentPath.getpath());
        currentPath.getpath().add(currentNode.getWord());
        executor.shutdown();
        return currentPath.getpath();
    }

    // jika bukan, expand node yang belum dikunjungi
    if (!visited.contains(currentNode.getWord())) {
        visited.add(currentNode.getWord());

        currentNode.expandNode();
        List<Callable<Void>> tasks = new ArrayList<>();
        // tambahkan node tetangga yang belum pernah dikunjungi pada prioqueue
        for (Node neighbor : currentNode.getNeighbour()) {
            tasks.add(() -> {
                // hitung heuristic cost yang ada pada masing masing node
```

```

        int newCost = neighbor.getcostBFS();
        synchronized (minCostToNode) {
            if (!minCostToNode.containsKey(neighbor.getWord()) ||
newCost < minCostToNode.get(neighbor.getWord())) {
                minCostToNode.put(neighbor.getWord(), newCost);
                ArrayList<String> newRoute = new
ArrayList<>(currentPath.getpath());
                newRoute.add(currentNode.getWord());
                Paths newPath = new Paths(neighbor, newRoute, newCost);
                pq.add(newPath); // tambah path ke prioqueue
            }
        }
        return null;
    });
}

for (Callable<Void> task : tasks) {
    completionService.submit(task);
}

// tunggu semua task selesai sebelum lanjut ke iterasi selanjutnya
try {
    for (int i = 0; i < tasks.size(); i++) {
        completionService.take().get();
    }
} catch (Exception e) {
    e.printStackTrace();
}
}

}

executor.shutdown();
return new ArrayList<String>();
}
}

```

3. Class Astar

Astar.java	
atribut	<ul style="list-style-type: none"> static Set<String> visited {menyimpan node yang telah dilalui}
Method	<ul style="list-style-type: none"> public static ArrayList<String> solveAstar() { method utama untuk mencari solusi menggunakan algoritma A* dan mengembalikan path yang ditemukan dalam bentuk Array of string }

Implementasi

```
public class Astar {

    static Set<String> visited = new HashSet<>();

    // override kelas comparator untuk mengurutkan array berdasarkan cost path
    public static class PathsComparator implements Comparator<Paths> {
        @Override
        public int compare(Paths p1, Paths p2) {
            return Integer.compare(p1.getCost(), p2.getCost());
        }
    }

    // method untuk mencari path menggunakan algoritma A*
    public static ArrayList<String> solveAstar() {

        // hapus list visited setiap memulai baru
        visited.clear();

        // Executor agar program bisa berjalan secara paralel
        ExecutorService executor =
        Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

        // menyiapkan priority queue dan map untuk menyimpan path
        PriorityQueue<Paths> pq = new PriorityQueue<>(new PathsComparator());
        CompletionService<Void> completionService = new
        ExecutorCompletionService<>(executor);
        Map<String, Integer> minCostToNode = new HashMap<>();

        // menyimpan start path ke priority queue
        Node startNode = new Node(Input.startinput);
        Paths startPath = new Paths(startNode, new ArrayList<>(), 0);
        pq.add(startPath);
        minCostToNode.put(startNode.getWord(), 0);

        // melakukan iterasi selama prioqueue belum kosong
        while (!pq.isEmpty()) {

            // dequeue prioqueue dan jadikan currentpath
            Paths currentPath = pq.poll();
            Node currentNode = currentPath.getCurrNode();

            // jika node saat ini adalah goal maka return hasil
            if (currentNode.getWord().equals(Input.targetinput)) {
                System.out.println("Astar FOUND!!!");
                System.out.println("End Node: " + currentNode.getWord());
                System.out.println("Total Cost: " + currentPath.getCost());
                System.out.println("length: " + currentPath.getpath().size());
            }
        }
    }
}
```

```
        System.out.println("Path: " + currentPath.getpath());
        currentPath.getpath().add(currentNode.getWord());
        executor.shutdown();
        return currentPath.getpath();
    }

    // jika bukan, expand node yang belum dikunjungi
    if (!visited.contains(currentNode.getWord())) {
        visited.add(currentNode.getWord());

        currentNode.expandNode();
        List<Callable<Void>> tasks = new ArrayList<>();
        // tambahkan node tetangga yang belum pernah dikunjungi pada prioqueue
        for (Node neighbor : currentNode.getNeighbour()) {
            tasks.add(() -> {
                // hitung cost yang ada pada masing masing node ( $f(n) = g(n) + h(n)$ )

                int newCost = neighbor.getcostBFS() + currentPath.getpath().size()+1;
                synchronized (minCostToNode) {
                    if (!minCostToNode.containsKey(neighbor.getWord()) || newCost <
minCostToNode.get(neighbor.getWord())) {
                        minCostToNode.put(neighbor.getWord(), newCost);
                        ArrayList<String> newRoute = new
ArrayList<>(currentPath.getpath());
                        newRoute.add(currentNode.getWord());
                        Paths newPath = new Paths(neighbor, newRoute, newCost);
                        pq.add(newPath); // tambah path ke prioqueue
                    }
                }
            });
        }

        for (Callable<Void> task : tasks) {
            completionService.submit(task);
        }

        // tunggu semua task selesai sebelum lanjut ke iterasi selanjutnya
        try {
            for (int i = 0; i < tasks.size(); i++) {
                completionService.take().get();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    executor.shutdown();
```

```
return new ArrayList<String>();
}
}
```

4. Class Word

Word.java	
atribut	<ul style="list-style-type: none"> private static final Set<String> englishWords { menyimpan kata-kata bahasa inggris dari dictionary }
Method	<ul style="list-style-type: none"> public static boolean isEnglishWord(String word) { method untuk memeriksa apakah sebuah kata merupakan bahasa inggris atau terdapat pada dictionary } public static void searchNeighbour(Node node) { method untuk mencari kata-kata tetangga yang berbahasa inggris dari sebuah node } public static int diffletter(String currWord) { method untuk mendapat heuristic cost dengan menghitung jumlah kata yang berbeda }
Implementasi	
<pre>public class Word { private static final Set<String> englishWords = new HashSet<>(); static { // Load dictionary try (BufferedReader reader = new BufferedReader(new FileReader("dictionary.txt"))) { String line; while ((line = reader.readLine()) != null) { // tambahkan kata pada string englishWords.add(line.trim().toLowerCase()); } } catch (IOException e) { System.err.println("Error loading word list: " + e.getMessage()); e.printStackTrace(); } } // method untuk memeriksa apakah sebuah kata merupakan bahasa inggris yang valid</pre>	

```

public static boolean isEnglishWord(String word) {
    return englishWords.contains(word.toLowerCase());
}

// mencari kata-kata tetangga dari sebuah node
public static void searchNeighbour(Node node) {
    ArrayList<Character> alphabet = new ArrayList<>(Arrays.asList(
        'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
        'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
        'u', 'v', 'w', 'x', 'y', 'z'));

    // Iterasi melalui setiap karakter dalam kata
    for (int i = 0; i < node.getWord().length(); i++) {
        for (int j = 0; j < alphabet.size(); j++) {
            if (node.getWord().charAt(i) == Input.targetInput.charAt(i)) {
                continue;
            }
            StringBuilder sb = new StringBuilder(node.getWord());
            sb.setCharAt(i, alphabet.get(j));
            String newWord = sb.toString();
            // jika kata merupakan bahasa inggris yang valid
            if (isEnglishWord(newWord) && !newWord.equals(node.getWord())) {
                Node newNode = new Node(newWord);
                node.addNode(newNode);
            }
        }
    }
}

// method untuk mendapat heuristic cost dengan menghitung jumlah kata yang berbeda
public static int diffLetter(String currWord) {
    int diff = 0;
    for (int i = 0; i < currWord.length(); i++) {
        if (Input.targetInput.charAt(i) != currWord.charAt(i)) {
            diff += 1;
        }
    }
    return diff;
}
}

```

5. Class Node

Node.java	
atribut	<ul style="list-style-type: none"> private String word; {kata yang ada pada node}

	<ul style="list-style-type: none"> ○ private ArrayList<Node> neighbour { node yang bertetangga } ○ private int costUCS { nilai cost g(n)} ○ private int costBFS {nilai cost h(n)} ○ private boolean visited; {status pengunjungan}
Method	<ul style="list-style-type: none"> ○ public Node(String word) {Konstruktor node} ○ public String getWord() {method untuk mendapat nilai word} ○ public ArrayList<Node> getNeighbour() {method untuk mendapat list tetangga node} ○ public int getcostUCS() {method untuk mendapat cost ucs} ○ public int getcostBFS() {method untuk mendapat cost bfs} ○ public Node getOptimalNeighbour() {method untuk mencari node tetangga dengan cost optimal} ○ public void setCostBFS() {method untuk mengeset nilai cost heuristic bfs} ○ public void setVisited() {method untuk mengeset node telah dikunjungi} ○ public boolean isVisited() {method untuk mengetahui apakah node telah dikunjungi} ○ public void printNeighbour() {method untuk mencetak semua tetangga node} ○ public void expandNode() {method untuk mengexpand node} ○ public void addNode(Node node) { method untuk menambah node tetangga }

Implementasi

```
public class Node {
    private String word;
    private ArrayList<Node> neighbour;
    private int costUCS;
    private int costBFS;
    private boolean visited;

    // konstruktor Node
    public Node(String word){
        this.word = word;
        this.costBFS = 0;
        this.costUCS = 1;
        this.visited = false;
        this.neighbour = new ArrayList<>();
    }

    // method untuk mendapat nilai word
    public String getWord(){
        return this.word;
    }

    // method untuk mendapat list tetangga node
    public ArrayList<Node> getNeighbour(){
        return this.neighbour;
    }

    // method untuk mendapat cost ucs
    public int getcostUCS(){
        return this.costUCS;
    }

    // method untuk mendapat cost bfs
    public int getcostBFS(){
        setCostBFS();
        return this.costBFS;
    }

    // method untuk mencari node tetangga dengan cost optimal
    public Node getOptimalNeighbour(){
        Node opt = null;
        int min = 100;
        for (Node i : neighbour){
            if(i.getcostBFS()<min){
                opt = i;
            }
        }
        return opt;
    }
}
```

```
}

// method untuk mengeset nilai cost heuristic bfs
public void setCostBFS() {
    int cost = Word.diffletter(this.getWord());
    this.costBFS = cost;
}

// method untuk mengeset node telah dikunjungi
public void setVisited() {
    this.visited = true;
}

// method untuk mengetahui apakah node telah dikunjungi
public boolean isVisited() {
    return visited;
}

// method untuk mencetak semua tetangga node
public void printNeighbour() {
    System.out.println("neighbour!!");
    for (Node i : neighbour) {
        System.out.println(i.getWord());
    }
}

// method untuk mengexpand node
public void expandNode() {
    Word.searchNeighbour(this);
}

// method untuk menambah node tetangga
public void addNode(Node node) {
    neighbour.add(node);
    // Sort secara alphabet
    Collections.sort(this.neighbour, new Comparator<Node>() {
        @Override
        public int compare(Node node1, Node node2) {
            return node1.getWord().compareTo(node2.getWord());
        }
    });
}
}
```

6. Class Paths

Paths.java

atribut	<ul style="list-style-type: none"> ○ private Node currNode; { current node dari path } ○ private ArrayList<String> path; { path yang sudah dilalui dari start ke node } ○ private int totalcost; { total cost dari node yang sudah dilalui }
Method	<ul style="list-style-type: none"> ○ public Paths(Node node, ArrayList<String> path, int totalcost) {Konstruktor path} ○ public Node getCurrNode() {method untuk mendapat current node} ○ public ArrayList<String> getpath() {method untuk mendapat list path yang dilalui} ○ public int getCost() {method untuk mendapat total cost}

Implementasi

```
public class Paths {

    private Node currNode;
    private ArrayList<String> path;
    private int totalcost;

    // konstruktor
    public Paths(Node node, ArrayList<String> path, int totalcost){
        this.currNode = node;
        this.path = path;
        this.totalcost = totalcost;
    }

    // method untuk mendapat current node
    public Node getCurrNode(){
        return currNode;
    }

    // method untuk mendapat path
    public ArrayList<String> getpath(){
        return path;
    }

    // method untuk mendapat cost
```

```

    public int getCost() {
        return totalcost;
    }
}

```

7. Class GUI

GUI.java	
atribut	<ul style="list-style-type: none"> ○ private JTextField startWordField, endWordField; ○ private JComboBox<String> algorithmComboBox; ○ private JButton findPathButton; ○ private JTextArea resultArea;
Method	<ul style="list-style-type: none"> ○ public GUI() {Konstruktor GUI} ○ private void findPath() { method untuk mencari path berdasarkan masukan user }
Implementasi	
<pre> public class GUI extends JFrame { private JTextField startWordField, endWordField; private JComboBox<String> algorithmComboBox; private JButton findPathButton; private JTextArea resultArea; public GUI() { setTitle("Word Ladder Solver"); setSize(500, 450); setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); setResizable(false); setLayout(new BorderLayout()); getContentPane().setBackground(new Color(255, 220, 220)); // Title Panel JPanel titlePanel = new JPanel(); titlePanel.setBackground(new Color(255, 105, 180)); JLabel titleLabel = new JLabel("Word Ladder Solver"); titleLabel.setFont(new Font("Comic Sans MS", Font.BOLD, 28)); titleLabel.setForeground(Color.WHITE); titlePanel.add(titleLabel); add(titlePanel, BorderLayout.NORTH); } } </pre>	

```
// Input Panel
JPanel inputPanel = new JPanel();
inputPanel.setLayout(new GridLayout(4, 1, 30, 1));
inputPanel.setBorder(new EmptyBorder(20, 80, 10, 80));
inputPanel.setBackground(new Color(255, 240, 240));

JLabel startLabel = new JLabel("Start Word:");
startLabel.setFont(new Font("Comic Sans MS", Font.BOLD, 16));
inputPanel.add(startLabel);

startWordField = new JTextField();
startWordField.setPreferredSize(new Dimension(500, 100));
inputPanel.add(startWordField);

JLabel endLabel = new JLabel("End Word:");
endLabel.setFont(new Font("Comic Sans MS", Font.BOLD, 16));
inputPanel.add(endLabel);

endWordField = new JTextField();
endWordField.setPreferredSize(new Dimension(500, 100));
inputPanel.add(endWordField);

JLabel algo = new JLabel("Algorithm :");
algo.setFont(new Font("Comic Sans MS", Font.BOLD, 16));
inputPanel.add(algo);

String[] algorithms = {"UCS", "Greedy Best First Search", "A*"};
algorithmComboBox = new JComboBox<>(algorithms);
algorithmComboBox.setFont(new Font("Comic Sans MS", Font.PLAIN, 14));
algorithmComboBox.setBackground(new Color(255, 182, 193));
algorithmComboBox.setPreferredSize(new Dimension(500, 30));
inputPanel.add(algorithmComboBox);

add(inputPanel, BorderLayout.CENTER);

// Result Panel
JPanel resultPanel = new JPanel();
resultPanel.setLayout(new BorderLayout());
resultPanel.setBorder(new EmptyBorder(10, 50, 20, 40));
resultPanel.setBackground(new Color(255, 240, 240));

resultArea = new JTextArea();
resultArea.setEditable(false);
resultArea.setFont(new Font("Comic Sans MS", Font.PLAIN, 14));
resultArea.setBackground(new Color(255, 255, 255));
resultArea.setWrapStyleWord(true);
resultArea.setLineWrap(true);
```

```
JScrollPane scrollPane = new JScrollPane(resultArea);
scrollPane.setPreferredSize(new Dimension(140, 140));

resultPanel.add(scrollPane, BorderLayout.CENTER);

// Find Path Button
findPathButton = new JButton("Find Path");
findPathButton.setFont(new Font("Comic Sans MS", Font.BOLD, 18));
findPathButton.setForeground(Color.WHITE);
findPathButton.setBackground(new Color(255, 105, 180));
findPathButton.setFocusPainted(false);
findPathButton.setBorderPainted(false);
findPathButton.setOpaque(true);
findPathButton.setPreferredSize(new Dimension(200, 50));
findPathButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        findPath();
    }
});

// Add hover effect on button
findPathButton.addMouseListener(new java.awt.event.MouseAdapter() {
    public void mouseEntered(java.awt.event.MouseEvent evt) {
        findPathButton.setBackground(new Color(255, 182, 193));
    }

    public void mouseExited(java.awt.event.MouseEvent evt) {
        findPathButton.setBackground(new Color(255, 105, 180));
    }
});

resultPanel.add(findPathButton, BorderLayout.SOUTH);

add(resultPanel, BorderLayout.SOUTH);

setLocationRelativeTo(null);
setVisible(true);
}

// method untuk mencari path berdasarkan masukan user
private void findPath() {
    String startWord = startWordField.getText().trim();
    String endWord = endWordField.getText().trim();
    String algorithm = (String) algorithmComboBox.getSelectedItem();

    if (!Word.isEnglishWord(startWord) || !Word.isEnglishWord(endWord)) {
        JOptionPane.showMessageDialog(this, "Please enter valid English words.");
        return;
    }
}
```

```
    }

    if (startWord.length() != endWord.length()) {
        JOptionPane.showMessageDialog(this, "Start and end words must have the same
length.");
        return;
    }

    Input.startinput = startWord.toLowerCase();
    Input.targetinput = endWord.toLowerCase();

    ArrayList<String> path = new ArrayList<String>();
    int visited = 0;
    long startTime = System.currentTimeMillis();

    if(algorithm=="UCS"){
        path = UCS.solveUCS();
        visited = UCS.visited.size();
    }else if(algorithm=="Greedy Best First Search"){
        path = BFS.solveBFS();
        visited = BFS.visited.size();
    }else{
        path = Astar.solveAstar();
        visited = Astar.visited.size();
    }

    long endTime = System.currentTimeMillis();

    if(path.size()>0){
        resultArea.setText("Path: " + String.join(" -> ", path) + "\n" +
        "Visited Nodes: " + visited + "\n"+
        "Execution Time (ms): " + (endTime - startTime));
    } else{
        resultArea.setText("No Solution\n" +
        "Visited Nodes: " + visited + "\n"+
        "Execution Time (ms): " + (endTime - startTime));
    }

}
```

3.2. Rancangan GUI

Pada tugas ini, saya membuat antarmuka grafis pengguna (GUI) menggunakan Java Swing untuk memudahkan dan mempercantik proses pencarian solusi. Java Swing adalah sebuah toolkit untuk pengembangan aplikasi desktop Java yang memungkinkan pembuatan

antarmuka grafis yang interaktif dan menarik. Program untuk antarmuka terdapat pada file GUI.java. Berikut adalah beberapa hasil GUI yang sudah dibuat.

3.2.1 Tampilan Awal



The screenshot shows the 'Word Ladder Solver' application window. It has a pink header with the title 'Word Ladder Solver'. Below the header, there are three input fields: 'Start Word:', 'End Word:', and 'Algorithm :'. The 'Algorithm' field is a dropdown menu currently showing 'UCS'. Below these fields is a large empty rectangular box. At the bottom of the window is a pink button labeled 'Find Path'.

3.2.1 Tampilan Awal Program

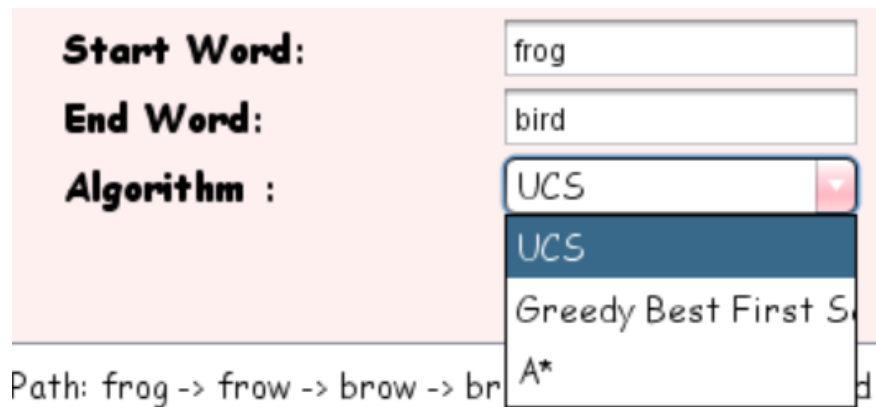
3.2.2. Tampilan Input Tidak Valid



The screenshot shows the 'Word Ladder Solver' application window with an error message dialog box overlaid. The dialog box has a title bar that says 'Message' and a close button (X). It contains an information icon (i) and the text 'Please enter valid English words.' with an 'OK' button. The background window is partially obscured by the dialog box, but the 'Find Path' button is still visible at the bottom.

3.2.2. Tampilan Input Tidak Valid

3.2.3. Tampilan Masukan Input



The screenshot shows the input interface of the Word Ladder Solver. It features three input fields: 'Start Word' with the value 'frog', 'End Word' with the value 'bird', and 'Algorithm' with a dropdown menu. The dropdown menu is open, showing options: 'UCS' (selected), 'UCS', 'Greedy Best First S', and 'A*'. Below the input fields, the text 'Path: frog -> frow -> brow -> br' is partially visible.

3.2.3. Tampilan Masukan Input

3.2.4. Tampilan Output



The screenshot shows the output interface of the Word Ladder Solver. It features a pink header with the title 'Word Ladder Solver'. Below the header, there are input fields for 'Start Word' (frog), 'End Word' (bird), and 'Algorithm' (UCS). A large text box displays the solved path: 'Path: frog -> frow -> brow -> bras -> bias -> bins -> bind -> bird'. Below the path, it shows 'Visited Nodes: 2476' and 'Execution Time (ms): 236'. At the bottom, there is a pink button labeled 'Find Path'.

3.2.4 Tampilan Output

BAB IV HASIL PENGUJIAN DAN PEMBAHASAN

4.1 Hasil Uji 1

Uniform Cost Search	<div><h3>Word Ladder Solver</h3><p>Start Word: <input type="text" value="earn"/></p><p>End Word: <input type="text" value="deny"/></p><p>Algorithm : <input type="text" value="UCS"/></p><div>Path: earn -> darn -> dare -> dere -> dene -> deny Visited Nodes: 2042 Execution Time (ms): 141</div><div>Find Path</div></div>
Greedy Best First Search	<div><h3>Word Ladder Solver</h3><p>Start Word: <input type="text" value="earn"/></p><p>End Word: <input type="text" value="deny"/></p><p>Algorithm : <input type="text" value="Greedy Best Fi..."/></p><div>Path: earn -> darn -> dark -> dank -> dang -> dong -> dons -> dens -> deny Visited Nodes: 12 Execution Time (ms): 16</div><div>Find Path</div></div>

A*	<div><h3>Word Ladder Solver</h3><p>Start Word: <input type="text" value="earn"/></p><p>End Word: <input type="text" value="deny"/></p><p>Algorithm : <input type="text" value="A*"/></p><div><p>Path: earn -> darn -> dare -> dere -> dene -> deny</p><p>Visited Nodes: 20</p><p>Execution Time (ms): 7</p></div><p>Find Path</p></div>
-----------	--

4.2 Hasil Uji 2

Uniform Cost Search	<div><h3>Word Ladder Solver</h3><p>Start Word: <input type="text" value="vase"/></p><p>End Word: <input type="text" value="bowl"/></p><p>Algorithm : <input type="text" value="UCS"/></p><div><p>Path: vase -> vale -> vole -> bole -> boll -> bowl</p><p>Visited Nodes: 2133</p><p>Execution Time (ms): 79</p></div><p>Find Path</p></div>
----------------------------	--

Greedy Best First Search

Word Ladder Solver

Start Word:

vase

End Word:

bowl

Algorithm :

Greedy Best Fi... ▼

Path: vase -> base -> bade -> bode -> bods -> bows -> bowl
Visited Nodes: 6
Execution Time (ms): 0

Find Path

A*

Word Ladder Solver

Start Word:

vase

End Word:

bowl

Algorithm :

A* ▼

Path: vase -> base -> bass -> boss -> bows -> bowl
Visited Nodes: 23
Execution Time (ms): 8

Find Path

4.3 Hasil Uji 3

Uniform Cost Search	<div><h2>Word Ladder Solver</h2><p>Start Word: <input type="text" value="smile"/></p><p>End Word: <input type="text" value="bored"/></p><p>Algorithm : <input type="text" value="UCS"/></p><div>Path: smile -> smite -> spite -> spits -> spots -> soots -> boots -> borts -> bores -> bored Visited Nodes: 4194 Execution Time (ms): 189</div><div>Find Path</div></div>
Greedy Best First Search	<div><h2>Word Ladder Solver</h2><p>Start Word: <input type="text" value="smile"/></p><p>End Word: <input type="text" value="bored"/></p><p>Algorithm : <input type="text" value="Greedy Best Fi..."/></p><div>Path: smile -> spile -> spite -> suite -> suits -> suets -> suers -> seers -> beers -> bears -> boars -> boats -> borts -> bores -> bored Visited Nodes: 26 Execution Time (ms): 10</div><div>Find Path</div></div>

A*

Word Ladder Solver

Start Word:

End Word:

Algorithm :

Path: smile -> smite -> spite -> spits -> spots -> soots -> boots -> borts -> bores -> bored

Visited Nodes: 234

Execution Time (ms): 16

Find Path

4.4 Hasil Uji 4

Uniform Cost Search

Word Ladder Solver

Start Word:

End Word:

Algorithm :

Path: lonely -> lanely -> lankly -> dankly -> darkly -> darkey -> darker -> marker -> market

Visited Nodes: 358

Execution Time (ms): 15

Find Path

Greedy Best First Search	<div><h3>Word Ladder Solver</h3><p>Start Word: <input type="text" value="lonely"/></p><p>End Word: <input type="text" value="market"/></p><p>Algorithm : <input type="text" value="Greedy Best Fi..."/></p><div><p>Path: lonely -> lanely -> lankly -> dankly -> darkly -> darkey -> darked -> marked -> market</p><p>Visited Nodes: 8</p><p>Execution Time (ms): 0</p></div><div>Find Path</div></div>
A*	<div><h3>Word Ladder Solver</h3><p>Start Word: <input type="text" value="lonely"/></p><p>End Word: <input type="text" value="market"/></p><p>Algorithm : <input type="text" value="A*"/></p><div><p>Path: lonely -> lanely -> lankly -> dankly -> darkly -> darkey -> darker -> marker -> market</p><p>Visited Nodes: 18</p><p>Execution Time (ms): 0</p></div><div>Find Path</div></div>

4.5 Hasil Uji 5

Uniform Cost Search	<div><h3>Word Ladder Solver</h3><p>Start Word: <input type="text" value="small"/></p><p>End Word: <input type="text" value="large"/></p><p>Algorithm : <input type="text" value="UCS"/></p><div>Path: small -> shall -> shill -> shiel -> shied -> shred -> sired -> siree -> saree -> laree -> large Visited Nodes: 4000 Execution Time (ms): 204</div><div>Find Path</div></div>
Greedy Best First Search	<div><h3>Word Ladder Solver</h3><p>Start Word: <input type="text" value="small"/></p><p>End Word: <input type="text" value="large"/></p><p>Algorithm : <input type="text" value="Greedy Best Fi..."/></p><div>Path: small -> shall -> shale -> scale -> scare -> stare -> state -> slate -> alate -> alane -> plane -> place -> peace -> pence -> ponce -> nonce -> nance -> rance -> range -> mange -> marge -> large Visited Nodes: 113 Execution Time (ms): 30</div><div>Find Path</div></div>

A*

Word Ladder Solver

Start Word:

End Word:

Algorithm :

Path: small -> shall -> shill -> shiel -> shied -> shred -> sired
-> siree -> saree -> sarge -> large

Visited Nodes: 489

Execution Time (ms): 47

Find Path

4.6 Hasil Uji 6

Uniform Cost Search

Word Ladder Solver

Start Word:

End Word:

Algorithm :

Path: brick -> crick -> chick -> check -> cheek -> cheep ->
sheep -> steep -> steel

Visited Nodes: 2643

Execution Time (ms): 204

Find Path

Greedy Best First Search

Word Ladder Solver

Start Word:

brick

End Word:

steel

Algorithm :

Greedy Best Fi... ▼

Path: brick -> brock -> trock -> truck -> trunk -> trank ->
trans -> trays -> treys -> trees -> treed -> tweed -> tweet
-> sweet -> sweep -> steep -> steel

Visited Nodes: 33

Execution Time (ms): 11

Find Path

A*

Word Ladder Solver

Start Word:

brick

End Word:

steel

Algorithm :

A* ▼

Path: brick -> crick -> chick -> check -> cheek -> cheep ->
sheep -> steep -> steel

Visited Nodes: 99

Execution Time (ms): 16

Find Path

4.7 Hasil Analisis

TC	UCS			G-BFS			A*		
	Panjang	Visited Node	Waktu	Panjang	Visited Node	Waktu	Panjang	Visited Node	Waktu
1	5	2042	141	8	12	16	5	20	7
2	5	2133	79	6	6	0	5	23	8
3	9	4194	189	14	26	10	9	234	16
4	8	358	15	8	8	0	8	18	0
5	10	4000	204	21	113	30	10	489	47
6	8	2643	204	16	33	11	8	99	16

Dari hasil analisis yang dilakukan, dapat disimpulkan perbandingan kinerja tiga algoritma pencarian, Greedy Best First Search (GBFS), Uniform Cost Search (UCS), dan A* Search dalam konteks penyelesaian Word Ladder.

Greedy Best First Search (GBFS) terbukti paling cepat dalam memperoleh solusi karena hanya mempertimbangkan nilai heuristik untuk memilih simpul berikutnya. Algoritma ini cenderung mengeksplorasi jalur yang memiliki nilai heuristik terbaik. Namun, kelemahan utama GBFS adalah ketidakmampuannya untuk menjamin solusi optimal karena tidak memeriksa semua kemungkinan (*not complete*). Pendekatannya yang hanya memperhatikan nilai heuristik dapat menghasilkan jalur yang tidak optimal secara keseluruhan.

Di sisi lain, Uniform Cost Search (UCS) adalah algoritma yang lengkap (*complete*) dan optimal. UCS secara sistematis mengeksplorasi semua kemungkinan jalur dengan mempertimbangkan biaya sejauh ini. Algoritma ini dapat menemukan solusi yang optimal dengan mengeksplorasi jalur yang memiliki biaya terendah, meskipun memerlukan waktu dan memori lebih banyak terutama jika terdapat banyak simpul atau jalur yang perlu dieksplorasi.

Kemudian, A* Search merupakan pilihan optimal dalam penyelesaian Word Ladder. A* menggabungkan pendekatan heuristik dengan biaya sejauh ini ($g(n)$) dan cost heuristic ke tujuan ($h(n)$). Dengan mempertimbangkan kedua faktor ini dalam fungsi evaluasi ($f(n) = g(n) + h(n)$), A* dapat menemukan jalur dengan biaya total terendah menuju solusi. Algoritma ini dijamin lengkap (*complete*) dan optimal jika fungsi heuristiknya konsisten (*admissible*) dan monoton (*consistent*).

UCS	G-BFS	A*
Total Memory: 15 MB Free Memory: 4 MB Used Memory: 10 MB	Total Memory: 15 MB Free Memory: 8 MB Used Memory: 6 MB	Total Memory: 15 MB Free Memory: 8 MB Used Memory: 7 MB

Dari hasil analisis memori yang digunakan dalam melakukan pencarian dari kata love menuju duck dapat dilihat bahwa UCS menggunakan memori paling banyak diikuti dengan A* dan G-BFS hal ini sejalan dengan simpul-simpul yang diperiksa. Pada UCS semua kemungkinan rute cenderung disimpan dan diperiksa sehingga diperlukan memori yang banyak. Pada A* juga menyimpan banyak kemungkinan rute namun hanya rute yang memiliki cost gabungan yang terbaik. Sedangkan pada G-BFS hanya menyimpan rute dengan cost heuristic terbaik.

Dalam konteks efisiensi dan keoptimalan penyelesaian Word Ladder, meskipun GBFS mungkin lebih cepat, solusi yang dihasilkannya tidak selalu optimal. UCS, sementara lengkap dan optimal, mungkin memerlukan lebih banyak waktu dan memori. Oleh karena itu, A* menjadi pilihan terbaik karena kemampuannya untuk menemukan solusi optimal dengan efisien, mempertimbangkan biaya sejauh ini dan heuristic cost ke tujuan.

BAB V

KESIMPULAN

5.1. Kesimpulan

Dari perbandingan antara Greedy Best First Search (GBFS), Uniform Cost Search (UCS), dan A* Search untuk penyelesaian Word Ladder, kita dapat menarik kesimpulan bahwa pemilihan algoritma pencarian dapat disesuaikan dengan prioritas yang diinginkan. GBFS lebih unggul dalam kecepatan karena hanya mempertimbangkan nilai heuristik, namun tidak menjamin solusi optimal. UCS, meskipun memerlukan waktu dan memori lebih banyak, namun dapat diandalkan untuk memberikan solusi optimal dalam konteks biaya minimal. Sedangkan A* Search, dengan pendekatan heuristik yang menggabungkan ($g(n)$) dan ($h(n)$), dapat menjadi pilihan terbaik untuk mencapai solusi optimal dengan efisien, asalkan fungsi heuristiknya konsisten (admissible) dan monoton (consistent).

Dalam memilih algoritma pencarian, penting untuk mempertimbangkan karakteristik masalah dan prioritas eksekusi. Jika kecepatan menjadi faktor utama dan solusi optimal tidak diperlukan, GBFS bisa menjadi pilihan yang layak. Namun, untuk mencapai solusi optimal dalam konteks biaya minimal, UCS atau A* Search menjadi pilihan yang lebih tepat, tergantung pada kompleksitas masalah dan ketersediaan memori. Oleh karena itu, pemilihan algoritma pencarian dapat disesuaikan dengan kebutuhan spesifik masalah yang dihadapi, dengan mempertimbangkan antara kecepatan, keoptimalan solusi, dan penggunaan memori.

LAMPIRAN

A. Repository Github : https://github.com/debrinashika/Tucil3_13522025

B. Checklist

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	