

Tugas Kecil 3 IF2211 Strategi Algoritma
Penyelesaian Permainan Word Ladder Menggunakan Algoritma UCS,
Greedy Best First Search, dan A*



Disusun oleh :
Erdianti Wiga Putri Andini (13522053)

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

2024

DAFTAR ISI

DAFTAR ISI.....	1
BAB I	
DESKRIPSI MASALAH.....	4
1.1 Algoritma UCS (Uniform Cost Search).....	4
1.2 Algoritma Greedy Best First Search.....	4
1.3 Algoritma A*.....	4
1.4 Word Ladder.....	5
BAB II	
IMPLEMENTASI ALGORITMA DALAM BAHASA JAVA.....	6
2.1 File AStar.java.....	6
2.2 File AStarNode.java.....	6
2.3 File BFS.java.....	8
2.4 File DictionaryLoader.java.....	9
2.5 File GreedyBFS.java.....	10
2.6 File Main.java.....	11
2.7 File Node.java.....	13
2.8 File SearchResult.java.....	14
2.9 File UCS.java.....	14
2.10 File Utils.java.....	15
2.11 File WordLadderException.java.....	16
2.12 File WordLadderGUI.java.....	17
BAB III	
SOURCE CODE PROGRAM.....	19
3.1 Repository Program.....	19
3.2 Source Code.....	19
3.2.1 AStar.java.....	19
3.2.2 AStarNode.java.....	20
3.2.3 BFS.java.....	21
3.2.4 DictionaryLoader.java.....	23
3.2.5 GreedyBFS.java.....	23
3.2.6 Main.java.....	25
3.2.7 Node.java.....	30

3.2.8 SearchResult.java.....	30
3.2.9 UCS.java.....	31
3.2.10 Utils.java.....	32
3.2.11 WordLadderException.java.....	33
3.2.12 WordLadderGUI.java.....	33
BAB IV	
ANALISIS ALGORITMA.....	41
4.1 Definisi $f(n)$, $g(n)$, dan $h(n)$	41
4.2 Proses Pembuatan Path Word Ladder dengan Algoritma Uniform Cost Search.....	41
4.3 Proses Pembuatan Path Word Ladder dengan Algoritma Greedy Best First Search.....	42
4.4 Proses Pembuatan Path Word Ladder dengan Algoritma A*	43
4.5 Nilai Heuristik pada A*	45
4.6 Perbandingan Algoritma UCS dan BFS pada Word Ladder.....	45
4.7 Perbandingan Algoritma UCS dan A* pada Word Ladder.....	46
4.8 Analisis Solusi Greedy Best First Search.....	46
4.9 Analisis Perbandingan Algoritma UCS, Greedy Best First Search, dan A* dalam Word Ladder.....	47
BAB V	
MASUKAN DAN LUARAN PROGRAM.....	50
5.1 CLI.....	50
5.1.1 Test Case 1 (Tooth – Fairy).....	50
5.1.2 Test Case 2 (Earn – Make).....	52
5.1.3 Test Case 3 (Free – Form).....	54
5.1.4 Test Case 4 (Believer – Evaluate).....	56
5.1.5 Test Case 5 (Sand – Crab).....	58
5.1.6 Test Case 6 (Flower – Sunset).....	60
5.1.7 Test Case 7 (Word not in english).....	62
5.1.8 Test Case 8 (Start or/and end word empty).....	62
5.1.9 Test Case 9 (Word length not equal).....	63
5.2 GUI.....	64
5.2.1 Test Case 1 (Tooth – Fairy).....	64
5.2.2 Test Case 2 (Earn – Make).....	67
5.2.3 Test Case 3 (Free – Form).....	69
5.2.4 Test Case 4 (Believer - Evaluate).....	71

5.2.5 Test Case 5 (Sand – Crab).....	73
5.2.6 Test Case 6 (Flower - Sunset).....	75
5.2.7 Test Case 7 (Word not in english).....	80
5.2.8 Test Case 8 (Start or/and word empty).....	81
5.2.9 Test Case 9 (Word length not equal).....	82
BAB VI	
KESIMPULAN.....	83
6.1 Kesimpulan.....	83
LAMPIRAN.....	84
DAFTAR REFERENSI.....	85

BAB I

DESKRIPSI MASALAH

1.1 Algoritma UCS (*Uniform Cost Search*)

Algoritma dimana pencarian solusi optimal didasarkan pada fungsi evaluasi $f(n)$ untuk setiap simpul, di mana $f(n) = g(n)$, dimana $g(n)$ adalah *cost* dari akar ke simpul n . Pada program ini, *cost* dihitung dengan menambah *cost* kumulatif sebesar 1 untuk setiap langkah (*cost* dibuat seragam). Algoritma ini memanfaatkan *priority queue* yang mengurutkan nilai $g(n)$ dari yang terkecil.

1.2 Algoritma Greedy Best First Search

Algoritma yang menggunakan fungsi evaluasi $f(n)$ untuk setiap simpul, di mana $f(n) = h(n)$, yang merupakan perkiraan *cost* dari simpul n menuju tujuan. Pencarian *greedy best-first* akan memperluas simpul yang tampaknya paling dekat dengan tujuan. Algoritma ini memanfaatkan *priority queue* yang mengurutkan nilai $h(n)$ dari yang terkecil. *Greedy Best First Search* memiliki beberapa permasalahan. Pertama, metode ini tidak lengkap. Kedua, metode ini rentan terjebak dalam optimal lokal minima atau plateau. Ketiga, pendekatan ini tidak dapat dibalik atau diubah (*irrevocable*).

1.3 Algoritma A*

Algoritma yang menghindari perluasan path yang *cost*-nya sudah bernilai tinggi. Fungsi evaluasi $f(n)$ didefinisikan sebagai $g(n) + h(n)$, di mana $g(n)$ adalah *cost* yang telah dikeluarkan untuk mencapai simpul n , dan $h(n)$ adalah perkiraan *cost* dari simpul n ke tujuan. Jadi, $f(n)$ adalah perkiraan total *cost* path melalui simpul n ke tujuan. Algoritma ini memanfaatkan *priority queue* yang mengurutkan nilai $f(n)$ dari yang terkecil. Heuristik yang digunakan pada algoritma A* admissible karena untuk setiap node n , $h(n) \leq h^*(n)$, di mana $h^*(n)$ adalah *cost* sebenarnya untuk mencapai keadaan tujuan dari n . Heuristik yang bersifat *admissible* tidak pernah melebihi-lebihkan *cost* untuk mencapai tujuan, yaitu, heuristik ini bersifat optimis.

1.4 Word Ladder

Word Ladder, juga dikenal dengan nama Doublets, *word-links*, *change-the-word puzzles*, *paragrams*, *laddergrams*, atau *word golf*, adalah permainan kata yang populer di kalangan luas. Permainan ini diciptakan oleh Lewis Carroll, seorang penulis dan matematikawan, pada tahun 1877. Dalam permainan ini, pemain diberikan dua kata yaitu start word (*start word*) dan end word (*end word*). Tujuan permainan adalah menemukan rangkaian kata yang menghubungkan start word ke end word. Setiap kata dalam rangkaian ini harus memiliki jumlah huruf yang sama dan hanya berbeda satu huruf dari kata sebelumnya. Misalnya, jika start word adalah "CAT" dan end word adalah "DOG," pemain harus menemukan kata perantara seperti "COT" dan "COG" yang membentuk *path* menuju end word. Tujuan tugas kecil dalam permainan ini adalah menemukan solusi optimal, yaitu solusi dengan jumlah perubahan kata paling sedikit untuk mencapai end word.

BAB II

IMPLEMENTASI ALGORITMA DALAM BAHASA JAVA

2.1 File AStar.java

File ini berisi program untuk melakukan pencarian *path* dengan menggunakan algoritma A*.

Nama Kelas	Deskripsi
AStar	Kelas yang digunakan untuk melakukan pencarian <i>path</i> dengan menggunakan algoritma A*. Method di dalamnya hanya aStarSearch.

Nama Atribut/Method	Deskripsi
aStarSearch()	Menerapkan algoritma pencarian A* untuk menemukan path transformasi kata terpendek antara kata start dan end, menggunakan kata-kata yang tersedia di dictionary. Memiliki parameter start (start word), end (end word), dictionary (sekumpulan kata untuk digunakan dalam pencarian yang berasal dari dictionary.txt). Fungsi ini mengembalikan objek SearchResult yang berisi <i>solution path</i> dan total jumlah node yang dikunjungi.

2.2 File AStarNode.java

File ini berisi kelas yang berisi atribut dan method untuk node yang digunakan pada algoritma A*.

Nama Kelas	Deskripsi
AStarNode	Kelas yang memiliki atribut dan method untuk node

	A*. Mempunyai atribut word yang berupa String, g dan h yang berupa integer, dan parent yang berupa AStarNode. Mempunyai konstruktor AStarNode(), AStarNode(String word), dan AStarNode(String word, int g, int h, AStarNode parent)
--	---

Nama Atribut/Method	Deskripsi
word	Atribut yang bertipe String dimana menyimpan kata yang diwakili oleh node ini. Kata tersebut digunakan dalam permainan sebagai bagian dari proses pencarian <i>path</i> .
g	Mewakili <i>cost</i> untuk mencapai node ini dari node awal. Dengan kata lain, ini mengukur <i>cost</i> kumulatif perjalanan sepanjang path menuju simpul ini.
h	Mewakili perkiraan heuristik <i>cost</i> dari node ini ke node akhir. Ini digunakan untuk memperkirakan sisa <i>cost</i> untuk mencapai tujuan.
parent	Menyimpan referensi ke node <i>parent</i> di <i>path</i> . Hal ini memungkinkan untuk merekonstruksi <i>path</i> solusi dengan menelusuri mundur dari tujuan ke titik awal.
AStarNode()	Menginisialisasi simpul dengan string kosong sebagai word, nilai nol untuk g dan h, dan parent null.
AStarNode(String word)	Menginisialisasi node dengan kata yang ditentukan. Set nilai g dan h ke nol dan inisialisasi parent ke AStarNode default baru.

AStarNode(String word, int g, int h, AStarNode parent)	Inisialisasi node dengan kata, g, h, dan induk yang diberikan. Konstruktor ini digunakan untuk membuat node dengan atribut lengkap.
getF()	Mengembalikan jumlah g (<i>cost</i> aktual dari simpul awal) dan h (perkiraan <i>cost</i> hingga tujuan), yang menghasilkan total perkiraan cost, yaitu f dalam integer. Ini digunakan untuk menentukan prioritas node dalam algoritma pencarian A*.

2.3 File BFS.java

File ini berisi program untuk melakukan pencarian *path* dengan menggunakan algoritma BFS.

Nama Kelas	Deskripsi
BFS	Kelas yang digunakan untuk melakukan pencarian <i>path</i> dengan menggunakan algoritma Breadth First Search. Mempunyai kelas private di dalamnya yaitu Node. Mempunyai method breadthFirstSearch saja.
Node	Kelas ini digunakan untuk mewakili node dalam pohon pencarian. Mempunyai atribut word, parent, dan depth, serta hanya memiliki konstruktor.

Nama Atribut/Method	Deskripsi
word	Mewakili kata pada node ini.
parent	Merujuk ke node induk dalam pohon pencarian, memungkinkan pelacakan balik untuk merekonstruksi path dari start word.

depth	Menunjukkan tingkat kedalaman node ini, mewakili berapa banyak transformasi dari start word yang dibutuhkan oleh node ini.
Node(String word, Node parent, int depth)	Menginisialisasi objek Node dengan kata yang ditentukan (word), node induk (parent), dan kedalaman (depth).
breadthFirstSearch()	Menerapkan algoritma pencarian luas (BFS) untuk menemukan path transformasi kata terpendek antara kata start dan end, menggunakan kata-kata yang tersedia di dictionary. Memiliki parameter start (start word), end (end word), dictionary (sekumpulan kata untuk digunakan dalam pencarian yang berasal dari dictionary.txt). Fungsi ini mengembalikan objek SearchResult yang berisi <i>solution path</i> dan total jumlah node yang dikunjungi.

2.4 File DictionaryLoader.java

File ini berisi program untuk melakukan load file dictionary.

Nama Kelas	Deskripsi
DictionaryLoader	Kelas ini berfungsi sebagai utilitas untuk memuat daftar kata dari file kamus eksternal ke dalam struktur data yang dapat digunakan oleh program.

Nama Atribut/Method	Deskripsi
loadDictionary()	Method static yang berfungsi untuk membaca file kamus dari file txt dan memuat semua kata yang valid ke dalam Set. Parameternya adalah filename

	yang bertipe String yaitu nama file dari file kamus yang akan dimuat. Method ini mengembalikan sebuah objek Set<String> yang berisi kumpulan semua kata dalam file kamus, semuanya dalam huruf besar untuk konsistensi. Method ini melakukan <i>throw exception</i> IOException jika ada masalah dalam membaca file, seperti tidak ditemukan atau file tidak dapat diakses.
--	---

2.5 File GreedyBFS.java

File ini berisi program untuk melakukan pencarian *path* dengan menggunakan algoritma Greedy Best First Search.

Nama Kelas	Deskripsi
GreedyBFS	Kelas yang digunakan untuk melakukan pencarian <i>path</i> dengan menggunakan algoritma Greedy Best First Search. Mempunyai kelas private di dalamnya yaitu Node. Mempunyai method <code>greedyBestFirstSearch</code> saja.
Node	Kelas ini digunakan untuk mewakili node dalam pohon pencarian. Mempunyai atribut <code>word</code> , <code>parent</code> , <code>depth</code> , dan <code>heuristic</code> serta hanya memiliki konstruktor.

Nama Atribut/Method	Deskripsi
<code>word</code>	Mewakili kata pada node ini.
<code>parent</code>	Merujuk ke node induk dalam pohon pencarian, memungkinkan pelacakan balik untuk

	merekonstruksi path dari start word.
depth	Menunjukkan tingkat kedalaman node ini, mewakili berapa banyak transformasi dari start word yang dibutuhkan oleh node ini.
heuristic	Menyimpan <i>cost</i> estimasi dari kata sekarang ke kata target/tujuan.
Node(String word, Node parent, int depth, int heuristic)	Menginisialisasi objek Node dengan kata yang ditentukan (word), node induk (parent), kedalaman (depth), dan nilai heuristik (heuristic).
greedyBestFirstSearch()	Menerapkan algoritma pencarian Greedy Best First Search untuk menemukan path transformasi kata terpendek antara kata start dan end, menggunakan kata-kata yang tersedia di dictionary. Memiliki parameter start (start word), end (end word), dictionary (sekumpulan kata untuk digunakan dalam pencarian yang berasal dari dictionary.txt). Memanfaatkan <i>priority queue</i> untuk mengurutkan node dengan nilai heuristic terendah. Fungsi ini mengembalikan objek SearchResult yang berisi <i>solution path</i> dan total jumlah node yang dikunjungi.

2.6 File Main.java

File ini berisi program utama Word Ladder berbasis CLI.

Nama Kelas	Deskripsi
Main	Kelas Main adalah kelas yang menyediakan titik masuk program untuk aplikasi Word Ladder Solver. Kelas ini berfungsi untuk mengatur alur kerja utama

	aplikasi, termasuk memuat kamus, menerima input dari pengguna, memilih algoritma pencarian, serta menampilkan hasil pencarian.
--	--

Nama Atribut/Method	Deskripsi
main()	Method utama yang menjadi titik awal eksekusi program. Menampilkan instruksi pada konsol untuk memulai aplikasi. Memuat file kamus (dictionary.txt) dan menyimpannya dalam Set<String> dictionary. Menginisiasi loop utama untuk memungkinkan pengguna memasukkan start word, end word, dan memilih algoritma pencarian.
validateWords()	Metode ini berfungsi untuk memverifikasi start word dan end word agar memenuhi kriteria untuk permainan Word Ladder. Mempunyai parameter startWord, endWord, dan dictionary. Method ini memeriksa apakah salah satu kata kosong, jika iya maka akan throw WordLadderException. Memeriksa apakah kedua kata terdapat di dalam kamus atau tidak, jika tidak throw WordLadderException. Memastikan kedua kata memiliki panjang yang sama, jika tidak throw WordLadderException.

2.7 File Node.java

File ini berisi kelas yang berisi atribut dan method untuk node yang digunakan pada algoritma UCS.

Nama Kelas	Deskripsi
Node	Kelas yang memiliki atribut dan method untuk node UCS. Mempunyai atribut word yang berupa String, parent yang berupa String, dan cost yang berupa integer. Mempunyai konstruktor Node() dan Node(String word, String parent, int cost).

Nama Atribut/Method	Deskripsi
word	Atribut yang bertipe String dimana menyimpan kata yang diwakili oleh node ini. Kata tersebut digunakan dalam permainan sebagai bagian dari proses pencarian <i>path</i> .
parent	Atribut yang bertipe String dimana menyimpan kata yang bertindak sebagai parent pada node ini.
cost	Atribut yang menyimpan nilai cost dari simpul saat ini ke simpul akar.
Node()	Menginisialisasi simpul dengan string kosong sebagai word, parent dengan string kosong, dan cost dengan nilai nol.
Node(String word, String parent, int cost)	Menginisialisasi node dengan kata yang ditentukan. Set parent dengan parent yang ditentukan dan cost dengan nilai cost yang ditentukan.

2.8 File SearchResult.java

File ini berisi kelas yang berfungsi untuk menyimpan hasil pencarian dari suatu algoritma.

Nama Kelas	Deskripsi
SearchResult	Kelas yang berfungsi sebagai wadah hasil pencarian (search result) dari suatu algoritma pencarian. Kelas ini menyimpan informasi tentang path (path) yang ditemukan serta jumlah simpul yang dikunjungi selama pencarian.

Nama Atribut/Method	Deskripsi
path	List yang menyimpan urutan kata dalam path yang ditemukan dari start word ke end word.
nodesVisited	Sebuah angka yang menyimpan jumlah total simpul yang telah dikunjungi selama proses pencarian.
SearchResult(List<String> path, int nodesVisited)	Konstruktor yang menerima dua parameter yaitu path (list yang berisi urutan kata yang membentuk path pencarian) dan nodesVisited (jumlah total simpul yang telah dikunjungi selama pencarian).
getPath()	Method yang mengembalikan objek List<String> yang berisi path dari start word ke end word.
getNodesVisited()	Method yang mengembalikan jumlah total simpul yang dikunjungi selama pencarian.

2.9 File UCS.java

File ini berisi program untuk melakukan pencarian *path* dengan menggunakan algoritma Uniform Cost Search.

Nama Kelas	Deskripsi
UCS	Kelas UCS menyediakan implementasi dari algoritma pencarian dengan <i>cost</i> seragam atau Uniform Cost Search (UCS). Algoritma ini menemukan path optimal dari start word ke end word, menggunakan konsep <i>cost</i> seragam.

Nama Atribut/Method	Deskripsi
uniformCostSearch()	Method ini mempunyai parameter start (start word), end (end word), dan dictionary (sebuah set yang berisi kumpulan kata yang valid selama pencarian yaitu dari dictionary.txt). Menggunakan <i>priority queue</i> untuk menyimpan simpul berdasarkan urutan terkecil <i>cost</i> kumulatif dari simpul awal ke simpul sekarang.

2.10 File Utils.java

File ini berisi method yang sama yang dibutuhkan oleh beberapa algoritma.

Nama Kelas	Deskripsi
Utils	Kelas Utils menyediakan beberapa metode utilitas yang berguna untuk algoritma pencarian kata, termasuk metode untuk menemukan tetangga dari sebuah kata dan metode heuristik untuk menghitung perbedaan antara dua kata.

Nama Atribut/Method	Deskripsi
---------------------	-----------

getNeighbors()	Method yang mempunyai parameter word (kata saat ini yang ingin dicari tetangganya) dan dictionary (kumpulan kata yang valid berupa set kata-kata). Mengembalikan List<String> yang berisi kata-kata yang merupakan tetangga valid dari kata masukan.
calculateHeuristic()	Method yang mempunyai parameter word1 dan word2 yang berupa String. Mengembalikan nilai diff berupa integer, yang menunjukkan jumlah perbedaan karakter antara kedua kata dan dapat digunakan sebagai perkiraan jarak antara kata tersebut.

2.11 File WordLadderException.java

File ini berisi exception dari program word ladder.

Nama Kelas	Deskripsi
WordLadderException	Kelas ini adalah subkelas dari Exception yang dibuat khusus untuk menangani exception dalam permainan Word Ladder.

Nama Atribut/Method	Deskripsi
WordLadderException(String message)	Konstruktor ini memanggil konstruktor superclass Exception untuk menginisialisasi objek pengecualian dengan pesan kesalahan yang diberikan sebagai argumen. Memiliki parameter message berupa String yang menjelaskan informasi lebih lanjut mengenai kesalahan yang terjadi.

2.12 File WordLadderGUI.java

File ini berisi program implementasi dari aplikasi GUI (Graphical User Interface) berbasis Java Swing untuk memecahkan permainan word ladder.

Nama Kelas	Deskripsi
WordLadderGUI	Kelas WordLadderGUI adalah implementasi dari aplikasi GUI (Graphical User Interface) berbasis Java Swing untuk memecahkan permainan word ladder. GUI ini memungkinkan pengguna memasukkan start word dan end word serta memilih algoritma yang akan digunakan untuk menemukan <i>path</i> yang menghubungkan kedua kata tersebut.

Nama Atribut/Method	Deskripsi
startWordField	Field teks untuk start word.
endWordField	Field teks untuk end word.
algorithmBox	Combo box untuk memilih algoritma.
outputPanel	Panel untuk menampilkan hasil path kata yang ditemukan.
outputScrollPane	Panel <i>scroll</i> untuk menampilkan hasil.
dictionary	Set yang menyimpan semua kata dalam kamus yang di-load.
softPink	Konstanta warna dan font untuk memperindah GUI.
verdanaFont	
backgroundColor	

WordLadderGUI	Konstruktor yang berfungsi untuk membuat jendela aplikasi dengan judul "Word Ladder Solver". Mengatur ukuran jendela dan posisi default. Memuat file kamus menggunakan kelas DictionaryLoader. Membuat panel input dengan label dan field teks untuk start word, end word, dan combo box untuk memilih algoritma. Membuat tombol Search! yang memicu pencarian kata ketika diklik. Menggabungkan semua komponen ini dalam panel utama dan menambahkannya ke jendela aplikasi.
createWordGridPanel()	Method untuk membuat panel grid yang berisi label per karakter dari setiap kata dalam path yang ditemukan. Setiap label menampilkan karakter, dan warnanya menyesuaikan dengan karakter yang sama antara kata dalam path dan end word.
findWordLadderPath()	Method untuk mengambil start word dan akhir dari input pengguna dan memastikan bahwa kata-kata tersebut valid. Berdasarkan algoritma yang dipilih, metode ini memanggil fungsi pencarian yang sesuai untuk menemukan path kata. Menampilkan path yang ditemukan dalam format grid beserta statistik (panjang path, total node yang dikunjungi, dan waktu eksekusi) dalam panel hasil.
main(String[] args)	Menginisiasi objek GUI dengan metode SwingUtilities.invokeLater untuk memastikan GUI berjalan dalam thread yang tepat dan menampilkan jendela aplikasi.

BAB III

SOURCE CODE PROGRAM

3.1 Repository Program

Berikut adalah pranala ke repository program:

https://github.com/wigaandini/Tucil3_13522053.git

3.2 Source Code

3.2.1 AStar.java

```
import java.util.*;

public class AStar {
    public static SearchResult aStarSearch(String start, String end,
    Set<String> dictionary) {
        PriorityQueue<AStarNode> openSet = new
    PriorityQueue<>(Comparator.comparingInt(AStarNode::getF));
        Map<String, Integer> gScoreMap = new HashMap<>();
        Set<String> closedSet = new HashSet<>();
        int nodesVisited = 0;

        AStarNode startNode = new AStarNode(start, 0,
    Utils.calculateHeuristic(start, end), null);
        openSet.add(startNode);
        gScoreMap.put(start, 0);

        while (!openSet.isEmpty()) {
            AStarNode current = openSet.poll();
            nodesVisited++;

            if (current.word.equals(end)) {
                List<String> path = new ArrayList<>();
                for (AStarNode node = current; node != null; node =
    node.parent) {
                    path.add(node.word);
                }
            }
        }
    }
}
```

```

        Collections.reverse(path);
        return new SearchResult(path, nodesVisited);
    }

    closedSet.add(current.word);
    for (String neighbor : Utils.getNeighbors(current.word,
dictionary)) {
        if (closedSet.contains(neighbor)) {
            continue;
        }

        int tentativeG = current.g + 1;
        if (tentativeG < gScoreMap.getOrDefault(neighbor,
Integer.MAX_VALUE)) {
            AStarNode neighborNode = new AStarNode(neighbor,
tentativeG, Utils.calculateHeuristic(neighbor, end), current);
            gScoreMap.put(neighbor, tentativeG);
            openSet.add(neighborNode);
        }
    }
}
return new SearchResult(Collections.emptyList(),
nodesVisited);
}
}

```

3.2.2 AStarNode.java

```

class AStarNode {
    String word;
    int g;
    int h;
    AStarNode parent;

    AStarNode() {
        this.word = "";
        this.g = 0;
    }
}

```

```

        this.h = 0;
        this.parent = null ;
    }

    AStarNode(String word) {
        this.word = word;
        this.g = 0;
        this.h = 0;
        this.parent = new AStarNode();
    }

    AStarNode(String word, int g, int h, AStarNode parent) {
        this.word = word;
        this.g = g;
        this.h = h;
        this.parent = parent;
    }

    int getF() {
        return g + h;
    }
}

```

3.2.3 BFS.java

```

import java.util.*;

public class BFS {
    private static class Node {
        String word;
        Node parent;
        int depth;

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

```

```

    }
}

    public static SearchResult breadthFirstSearch(String start,
String end, Set<String> dictionary) {
    Queue<Node> queue = new LinkedList<>();
    Set<String> visited = new HashSet<>();
    queue.add(new Node(start, null, 0));
    visited.add(start);
    int nodesVisited = 0;

    while (!queue.isEmpty()) {
        Node current = queue.poll();
        nodesVisited++;

        if (current.word.equals(end)) {
            List<String> path = new ArrayList<>();
            while (current != null) {
                path.add(current.word);
                current = current.parent;
            }
            Collections.reverse(path);
            return new SearchResult(path, nodesVisited);
        }

        for (String neighbor : Utils.getNeighbors(current.word,
dictionary)) {
            if (!visited.contains(neighbor)) {
                visited.add(neighbor);
                queue.add(new Node(neighbor, current,
current.depth + 1));
            }
        }
    }

    return new SearchResult(Collections.emptyList(),
nodesVisited);
}
}

```

3.2.4 DictionaryLoader.java

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

public class DictionaryLoader {
    public static Set<String> loadDictionary(String filename) throws
IOException {
        Set<String> dictionary = new HashSet<>();
        BufferedReader reader = new BufferedReader(new
FileReader(filename));
        String line;
        while ((line = reader.readLine()) != null) {
            dictionary.add(line.trim().toUpperCase());
        }
        reader.close();
        return dictionary;
    }
}
```

3.2.5 GreedyBFS.java

```
import java.util.*;

public class GreedyBFS {
    private static class Node {
        String word;
        Node parent;
        int depth;
        int heuristic;

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



```

        this.depth = depth;
        this.heuristic = heuristic;
    }
}

    public static SearchResult greedyBestFirstSearch(String start,
String end, Set<String> dictionary) {
        Queue<Node> queue = new
PriorityQueue<>(Comparator.comparingInt(node -> node.heuristic));
        Map<String, Integer> visited = new HashMap<>();
        queue.add(new Node(start, null, 0,
Utils.calculateHeuristic(start, end)));
        int nodesVisited = 0;

        while (!queue.isEmpty()) {
            Node current = queue.poll();
            nodesVisited++;

            if (current.word.equals(end)) {
                List<String> path = new ArrayList<>();
                while (current != null) {
                    path.add(current.word);
                    current = current.parent;
                }
                Collections.reverse(path);
                // System.out.println("\nTotal nodes visited: " +
nodesVisited);
                return new SearchResult(path, nodesVisited);
            }

            for (String neighbor : Utils.getNeighbors(current.word,
dictionary)) {
                if (!visited.containsKey(neighbor) ||
visited.get(neighbor) > current.depth + 1) {
                    visited.put(neighbor, current.depth + 1);
                    queue.add(new Node(neighbor, current,
current.depth + 1, Utils.calculateHeuristic(neighbor, end)));
                }
            }
        }
    }
}

```

```
    }  
    }  
    return new SearchResult(Collections.emptyList(),  
nodesVisited);  
    }  
}
```

3.2.6 Main.java

```
himport java.util.*;
import java.io.IOException;

public class Main {
    public static void main(String[] args) {
        System.out.println("
");
        System.out.println("
");

System.out.println("
");

System.out.println("
");

System.out.println("
");
        System.out.println("
");

        System.out.println("
Welcome to Word Ladder
Solver!
");
        System.out.println("
13522053
");

        System.out.println("
");

        System.out.println("
");
    }
}
```

```

        try {
            Set<String> dictionary =
DictionaryLoader.loadDictionary("dictionary.txt");
            Scanner scanner = new Scanner(System.in);
            boolean continuePlaying = true;

            while (continuePlaying) {
                String startWord, endWord;

                while (true) {
                    System.out.print("Enter start word: ");
                    startWord =
scanner.nextLine().trim().toUpperCase();

                    System.out.print("Enter end word: ");
                    endWord =
scanner.nextLine().trim().toUpperCase();

                    try {
                        validateWords(startWord, endWord,
dictionary);
                        break;
                    } catch (WordLadderException e) {
                        System.out.println(e.getMessage());
                    }
                }

                SearchResult result = null;
                long startTime = 0, endTime = 0;
                long memoryBefore = 0, memoryAfter = 0;
                boolean validAlgorithm = false;

                while (!validAlgorithm) {
                    System.out.println("\nChoose an algorithm: ");
                    System.out.println("1. Uniform Cost Search
(UCS)");
                    System.out.println("2. Greedy Best First Search
(GBFS)");

```

```

        System.out.println("3. A* Search");
        System.out.println("4. Breadth First Search
(BFS)\n");

        System.out.print("Enter the chosen algorithm
(UCS, GBFS, A*, BFS): ");
        String algorithm =
scanner.nextLine().trim().toUpperCase();

        Runtime runtime = Runtime.getRuntime();
        runtime.gc();
        memoryBefore = runtime.totalMemory() -
runtime.freeMemory();

        startTime = System.currentTimeMillis();

        switch (algorithm) {
            case "UCS":
                result =
UCS.uniformCostSearch(startWord, endWord, dictionary);
                validAlgorithm = true;
                break;
            case "GBFS":
                result =
GreedyBFS.greedyBestFirstSearch(startWord, endWord, dictionary);
                validAlgorithm = true;
                break;
            case "A*":
                result = AStar.aStarSearch(startWord,
endWord, dictionary);
                validAlgorithm = true;
                break;
            case "BFS":
                result =
BFS.breadthFirstSearch(startWord, endWord, dictionary);
                validAlgorithm = true;
                break;
            default:

```

```

        System.out.println("Invalid algorithm
selected. Please try again.");
        break;
    }

    endTime = System.currentTimeMillis();
    memoryAfter = runtime.totalMemory() -
runtime.freeMemory();
}

    if (result == null || result.getPath().isEmpty()) {
        System.out.println("\nNo path found.");
    } else {
        System.out.println("Path length: " +
result.getPath().size());
        System.out.println("Path:");
        List<String> path = result.getPath();
        for (int i = 0; i < path.size(); i++) {
            System.out.print(path.get(i));
            if (i < path.size() - 1) {
                System.out.print(" -> ");
            }
        }
        System.out.println("\n\nTotal nodes visited: " +
result.getNodesVisited());
        System.out.println("Execution time: " + (endTime
- startTime) + " ms");
        System.out.println("Memory used: " +
(memoryAfter - memoryBefore) + " bytes");
    }

    System.out.print("\nDo you want to continue? (y/n):
");

    String choice =
scanner.nextLine().trim().toLowerCase();
    System.out.println();
    continuePlaying = choice.equals("y");
    if (!continuePlaying) {

```

```

        System.out.println("Exiting the Word Ladder.
        Goodbye!");
    }
}
} catch (IOException e) {
    System.out.println("Failed to load dictionary: " +
e.getMessage());
}
}

    public static void validateWords(String startWord, String
endWord, Set<String> dictionary) throws WordLadderException {
        if (startWord.length() == 0 || endWord.length() == 0) {
            throw new WordLadderException("Start word and end word
must not be empty!\n");
        }

        if (!dictionary.contains(startWord) &&
!dictionary.contains(endWord)) {
            throw new WordLadderException("Start word and end word
must be words in English!\n");
        }
        if (!dictionary.contains(startWord)) {
            throw new WordLadderException("Start word must be words
in English!\n");
        }
        if (!dictionary.contains(endWord)) {
            throw new WordLadderException("End word must be words in
English!\n");
        }
        if (startWord.length() != endWord.length()) {
            throw new WordLadderException("Start word and end word
must be of equal length.\n");
        }
    }
}
}

```

3.2.7 Node.java

```
class Node {
    String word;
    String parent;
    int cost;

    Node() {
        this.word = "";
        this.parent = "";
        this.cost = 0;
    }

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

3.2.8 SearchResult.java

```
import java.util.List;

public class SearchResult {
    private final List<String> path;
    private final int nodesVisited;

    public SearchResult(List<String> path, int nodesVisited) {
        this.path = path;
        this.nodesVisited = nodesVisited;
    }

    public List<String> getPath() {
        return path;
    }

    public int getNodesVisited() {
```

```

        return nodesVisited;
    }
}

```

3.2.9 UCS.java

```

import java.util.*;

public class UCS {
    public static SearchResult uniformCostSearch(String start,
String end, Set<String> dictionary) {
        PriorityQueue<Node> frontier = new
PriorityQueue<>(Comparator.comparingInt(n -> n.cost));
        Map<String, Integer> costSoFar = new HashMap<>();
        Map<String, String> cameFrom = new HashMap<>();
        int nodesVisited = 0;

        frontier.add(new Node(start, null, 0));
        costSoFar.put(start, 0);
        cameFrom.put(start, null);

        while (!frontier.isEmpty()) {
            Node current = frontier.poll();
            nodesVisited++;

            if (current.word.equals(end)) {
                return new SearchResult(reconstructPath(cameFrom,
current.word), nodesVisited);
            }

            for (String neighbor : Utils.getNeighbors(current.word,
dictionary)) {
                int newCost = current.cost + 1;
                if (!costSoFar.containsKey(neighbor) || newCost <
costSoFar.get(neighbor)) {
                    frontier.add(new Node(neighbor, current.word,
newCost));
                }
            }
        }
    }
}

```



```

        costSoFar.put(neighbor, newCost);
        cameFrom.put(neighbor, current.word);
    }
}
}
return new SearchResult(Collections.emptyList(),
nodesVisited);
}

private static List<String> reconstructPath(Map<String, String>
cameFrom, String current) {
    List<String> path = new ArrayList<>();
    while (current != null) {
        path.add(current);
        current = cameFrom.get(current);
    }
    Collections.reverse(path);
    return path;
}
}

```

3.2.10 Utils.java

```

import java.util.ArrayList;
import java.util.List;
import java.util.Set;

public class Utils {
    public static List<String> getNeighbors(String word, Set<String>
dictionary) {
        List<String> neighbors = new ArrayList<>();
        char[] chars = word.toUpperCase().toCharArray();
        for (int i = 0; i < chars.length; i++) {
            char originalChar = chars[i];
            for (char c = 'A'; c <= 'Z'; c++) {
                if (c != originalChar) {
                    chars[i] = c;

```

```

        String newWord = new String(chars);
        if (dictionary.contains(newWord)) {
            neighbors.add(newWord);
            // System.out.println("Valid neighbor for "
+ word + ": " + newWord);
        }
    }
}
chars[i] = originalChar;
}
return neighbors;
}

public static int calculateHeuristic(String word1, String word2)
{
    int diff = 0;
    for (int i = 0; i < word1.length(); i++) {
        if (word1.charAt(i) != word2.charAt(i)) {
            diff++;
        }
    }
    return diff;
}
}

```

3.2.11 WordLadderException.java

```

class WordLadderException extends Exception {
    public WordLadderException(String message) {
        super(message);
    }
}

```

3.2.12 WordLadderGUI.java

```

import javax.swing.*;

```

```

import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.util.List;
import java.util.Set;

public class WordLadderGUI extends JFrame {
    private JTextField startWordField;
    private JTextField endWordField;
    private JComboBox<String> algorithmBox;
    private JPanel outputPanel;
    private JScrollPane outputScrollPane;
    private Set<String> dictionary;
    private final Color softPink = new Color(194, 125, 150);
    private final Font verdanaFont = new Font("Verdana", Font.BOLD,
14);
    private final Color backgroundColor = new Color(255, 245, 249);

    public WordLadderGUI() {
        setTitle("Word Ladder Solver");
        setSize(700, 600);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);

        try {
            dictionary =
DictionaryLoader.loadDictionary("dictionary.txt");
        } catch (IOException e) {
            JOptionPane.showMessageDialog(this, "Unable to load
dictionary!", "Error", JOptionPane.ERROR_MESSAGE);
            System.exit(1);
        }

        JPanel inputPanel = new JPanel(new GridLayout(3, 2, 10,
10));
        inputPanel.setBorder(BorderFactory.createEmptyBorder(10, 10,
10, 10));

```

```

inputPanel.setOpaque(true);
inputPanel.setBackground(backgroundColor);

JLabel startLabel = new JLabel("Start Word:");
startLabel.setFont(verdanaFont);
startLabel.setForeground(softPink);
startWordField = new JTextField();
startWordField.setFont(verdanaFont);

JLabel endLabel = new JLabel("End Word:");
endLabel.setFont(verdanaFont);
endLabel.setForeground(softPink);
endWordField = new JTextField();
endWordField.setFont(verdanaFont);

JLabel algoLabel = new JLabel("Algorithm:");
algoLabel.setFont(verdanaFont);
algoLabel.setForeground(softPink);
algorithmBox = new JComboBox<>(new String[]{"UCS", "Greedy
BFS", "A*", "BFS"});
algorithmBox.setFont(verdanaFont);

inputPanel.add(startLabel);
inputPanel.add(startWordField);
inputPanel.add(endLabel);
inputPanel.add(endWordField);
inputPanel.add(algoLabel);
inputPanel.add(algorithmBox);

JButton searchButton = new JButton("Search!");
searchButton.setFont(verdanaFont);
searchButton.setForeground(Color.WHITE);
searchButton.setBackground(softPink);
searchButton.setPreferredSize(new Dimension(100, 30));
searchButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        findWordLadderPath();
    }
});

```

```

    }
});

JPanel buttonPanel = new JPanel();
buttonPanel.setOpaque(false);
buttonPanel.setBorder(BorderFactory.createEmptyBorder(20, 0,
20, 0));
buttonPanel.add(searchButton);

outputPanel = new JPanel(new BorderLayout());
outputPanel.setOpaque(true);
outputPanel.setBackground(backgroundColor);

outputScrollPane = new JScrollPane(outputPanel);
outputScrollPane.setOpaque(true);
outputScrollPane.setBackground(backgroundColor);
outputScrollPane.setPreferredSize(new Dimension(680, 300));

JPanel mainPanel = new JPanel(new BorderLayout());
mainPanel.setOpaque(true);
mainPanel.setBackground(backgroundColor);
mainPanel.add(inputPanel, BorderLayout.NORTH);
mainPanel.add(buttonPanel, BorderLayout.CENTER);
mainPanel.add(outputScrollPane, BorderLayout.SOUTH);

add(mainPanel);
}

private JPanel createWordGridPanel(List<String> path, String
endWord) {
    int wordLength = path.get(0).length();
    JPanel gridPanel = new JPanel(new GridLayout(path.size(),
wordLength, 5, 5));
    gridPanel.setOpaque(true);
    gridPanel.setBackground(backgroundColor);

    for (String word : path) {
        for (int i = 0; i < word.length(); i++) {

```

```

        JLabel label = new
JLabel(String.valueOf(word.charAt(i)), SwingConstants.CENTER);
        label.setFont(new Font("Verdana", Font.BOLD, 18));
        label.setOpaque(true);

        if (i < endWord.length() && word.charAt(i) ==
endWord.charAt(i)) {
            label.setBackground(new Color(194, 125, 150));
            label.setForeground(Color.BLACK);
        } else {
            label.setBackground(Color.WHITE);
            label.setForeground(Color.BLACK);
        }

label.setBorder(BorderFactory.createLineBorder(Color.GRAY));
        gridPanel.add(label);
    }
}

return gridPanel;
}

private void findWordLadderPath() {
    String startWord =
startWordField.getText().trim().toUpperCase();
    String endWord =
endWordField.getText().trim().toUpperCase();
    String algorithm = (String) algorithmBox.getSelectedItem();

    try {
        Main.validateWords(startWord, endWord, dictionary);
    } catch (WordLadderException e) {
        JOptionPane.showMessageDialog(this, e.getMessage(),
"Error", JOptionPane.ERROR_MESSAGE);
        return;
    }
}

```

```

        SearchResult result = null;
        long startTime = System.currentTimeMillis();
        switch (algorithm) {
            case "UCS":
                result = UCS.uniformCostSearch(startWord, endWord,
dictionary);
                break;
            case "Greedy BFS":
                result = GreedyBFS.greedyBestFirstSearch(startWord,
endWord, dictionary);
                break;
            case "A*":
                result = AStar.aStarSearch(startWord, endWord,
dictionary);
                break;
            case "BFS":
                result = BFS.breadthFirstSearch(startWord, endWord,
dictionary);
                break;
        }
        long endTime = System.currentTimeMillis();

        outputPanel.removeAll();
        outputPanel.setLayout(new BoxLayout(outputPanel,
BoxLayout.Y_AXIS));

        JPanel wordGridPanel;
        if (result == null || result.getPath().isEmpty()) {
            wordGridPanel = new JPanel();
            wordGridPanel.setOpaque(true);
            wordGridPanel.setBackground(backgroundColor);
            wordGridPanel.add(new JLabel("No path found.));
        } else {
            wordGridPanel = createWordGridPanel(result.getPath(),
endWord);
        }

        wordGridPanel.setAlignmentX(Component.CENTER_ALIGNMENT);

```

```

        outputPanel.add(wordGridPanel);

        StringBuilder stats = new StringBuilder();
        stats.append("<html><body>")
            .append("<p style='font-family:Verdana, sans-serif;'>Path length: ").append(result != null ? result.getPath().size() : 0).append("</p>")
            .append("<p style='font-family:Verdana, sans-serif;'>Total nodes visited: ").append(result != null ? result.getNodesVisited() : 0).append("</p>")
            .append("<p style='font-family:Verdana, sans-serif;'>Execution time: ").append(endTime - startTime).append("ms</p>")
            .append("</body></html>");

        JEditorPane statsPane = new JEditorPane();
        statsPane.setContentType("text/html");
        statsPane.setEditable(false);
        statsPane.setText(stats.toString());
        statsPane.setBackground(backgroundColor);
        statsPane.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

        statsPane.setAlignmentX(Component.CENTER_ALIGNMENT);
        outputPanel.add(statsPane);

        int resultHeight = 80 + 30 * (result != null ? result.getPath().size() : 0);
        outputScrollPane.setPreferredSize(new Dimension(680, Math.min(resultHeight, 500)));

        outputPanel.revalidate();
        outputPanel.repaint();
        this.pack();
    }

    public static void main(String[] args) {

```



```
SwingUtilities.invokeLater(() -> {  
    WordLadderGUI gui = new WordLadderGUI();  
    gui.setVisible(true);  
});  
}  
}
```

BAB IV

ANALISIS ALGORITMA

4.1 Definisi $f(n)$, $g(n)$, dan $h(n)$

Nilai $f(n)$ adalah fungsi evaluasi keseluruhan yang digunakan untuk menentukan prioritas node dalam *priority queue*. Nilai $f(n)$ berbeda-beda tergantung menggunakan algoritma apa. Nilai $g(n)$ adalah *cost* aktual yang sudah dikeluarkan untuk mencapai node n dari node awal. Dalam word ladder, $g(n)$ dihitung sebagai jumlah langkah atau transisi antara kata-kata. Nilai $g(n)$ terus diperbarui saat node baru diproses. Nilai $h(n)$ adalah estimasi *cost* terendah untuk mencapai node tujuan dari node n saat ini. Fungsi heuristik ini harus memperkirakan dengan akurat *cost* tersisa. Dalam word ladder, nilai $h(n)$ dihitung sebagai jumlah huruf berbeda antara kata saat ini dan kata tujuan. Pada algoritma UCS, nilai $f(n) = g(n)$. Pada algoritma *Greedy Best First Search*, $f(n) = h(n)$. Sedangkan pada algoritma A^* , nilai $f(n) = g(n) + h(n)$.

4.2 Proses Pembuatan Path Word Ladder dengan Algoritma Uniform Cost Search

Pembentukan path word ladder dengan algoritma UCS ini menggunakan *priority queue* untuk menyimpan nilai $g(n)$ yaitu *cost* dari akar ke simpul n dengan terurut dari yang terkecil. Prosesnya adalah sebagai berikut:

1. *Priority queue* bernama 'frontier' diinisialisasi untuk mengelola pencarian berdasarkan nilai $g(n)$. *Queue* ini menggunakan komparator untuk memprioritaskan node dengan *cost* terkecil.
2. Map 'costSoFar' dibuat dan digunakan untuk melacak *cost* minimum yang diperlukan untuk mencapai setiap kata.
3. Map 'cameFrom' dibuat dan digunakan untuk menyimpan kata sebelumnya agar dapat merekonstruksi *path* selanjutnya.
4. *Start word* ditambahkan ke 'frontier' dengan *cost* nol dan node awal.
5. Map 'costSoFar' dan 'cameFrom' juga diinisialisasi dengan *start word*.
6. Proses pencarian akan terus berlanjut selama 'frontier' tidak kosong.
7. Node dengan *cost* terendah diambil dari 'frontier'.
8. Jumlah node yang dikunjungi ('nodesVisited') bertambah 1.

9. Jika kata saat ini merupakan *end word*, *path* direkonstruksi menggunakan ``cameFrom`` dan dikembalikan dalam ``SearchResult``, bersama dengan total node yang dikunjungi.
10. Jika kata saat ini bukan *end word*, kata-kata tetangga yang berbeda satu karakter diambil menggunakan ``Utils.getNeighbors``.
11. Untuk setiap tetangga, *cost* kumulatif baru dihitung sebagai *cost* node saat ini ditambah satu (setiap transisi memiliki *cost* 1).
12. Jika tetangga belum dikunjungi atau jika *cost* baru lebih rendah dari *cost* yang sebelumnya diketahui, tetangga tersebut akan ditambahkan ke ``frontier``. *cost* kumulatifnya diperbarui di ``costSoFar``. Serta kata pendahulunya (kata saat ini) diperbarui di ``cameFrom``.
13. Jika tidak ada *path* yang ditemukan antara *start word* dan *end word*, collection empty dikembalikan dalam ``SearchResult``.
14. Metode ``reconstructPath`` melacak dari *end word* ke *start word* menggunakan map ``cameFrom``, alias membangun *path* secara terbalik.
15. *Path* kemudian dibalik sebelum dikembalikan untuk menunjukkan urutan yang benar dari *start word* ke *end word*.

Secara keseluruhan, algoritma UCS memastikan menemukan *path* terpendek (jika ada) dengan selalu menelusuri *path cost* terkecil terlebih dahulu. *Priority queue* memastikan program untuk menjelajahi node sesuai urutan *cost* terkecil dari *start word*.

4.3 Proses Pembuatan Path Word Ladder dengan Algoritma Greedy Best First Search

Pembentukan path word ladder dengan algoritma GBFS ini menggunakan *priority queue* untuk menyimpan nilai heuristik $h(n)$ yaitu perkiraan *cost* dari simpul n ke simpul tujuan dengan terurut dari yang terkecil. Prosesnya adalah sebagai berikut:

1. Membuat kelas Node untuk menyimpan informasi tentang kata saat ini, kata pendahulu, *depth* pohon pencarian, dan nilai heuristik (jumlah perbedaan huruf antara kata saat ini dan end word).
2. *Priority Queue* bernama ``queue`` diinisialisasi menggunakan komparator untuk memprioritaskan node berdasarkan nilai heuristik yang paling rendah.
3. Map ``visited`` diinisialisasi untuk melacak kata yang telah dikunjungi.

4. Node awal dibuat dengan start word, *depth* 0, dan nilai heuristik dihitung dari start word ke end word menggunakan ``Utils.calculateHeuristic``. Node ini dimasukkan ke dalam queue.
5. Proses pencarian akan terus berlanjut selama queue tidak kosong.
6. Node dengan nilai heuristik terendah diambil dari queue.
7. Jumlah node yang dikunjungi (`nodesVisited`) bertambah 1.
8. Jika kata saat ini adalah end word, path direkonstruksi dengan melacak dari end word kembali ke start word melalui referensi kata pendahulu di dalam node.
9. Daftar path (path) kemudian dibalik agar menunjukkan urutan dari start word ke tujuan, lalu dikembalikan dalam ``SearchResult``.
10. Jika kata saat ini bukan end word, kata-kata tetangga yang berbeda satu karakter diambil menggunakan ``Utils.getNeighbors``.
11. Untuk setiap tetangga, jika belum dikunjungi atau *depth*-nya lebih rendah dari kunjungan sebelumnya, tetangga ini ditambahkan ke visited dengan *depth* yang diperbarui.
12. Node tetangga ditambahkan ke queue dengan *depth* baru dan nilai heuristik yang dihitung berdasarkan kata tetangga dan end word.
13. Jika tidak ada path antara start word dan end word, maka daftar kosong dikembalikan dalam ``SearchResult``.

Algoritma Greedy BFS ini berfokus pada nilai heuristik untuk memprioritaskan eksplorasi kata yang terlihat paling dekat dengan end word berdasarkan perbedaan huruf. Walaupun tidak selalu optimal, pendekatan ini sering menemukan solusi dengan cepat jika nilai heuristiknya tepat.

4.4 Proses Pembuatan Path Word Ladder dengan Algoritma A*

Pembentukan path word ladder dengan algoritma A* ini menggunakan *priority queue* untuk menyimpan nilai heuristik $f(n)$ yang bernilai penjumlahan dari $g(n)$ dan $h(n)$ sebagaimana telah dijelaskan bahwa $g(n)$ adalah *cost* dari akar ke simpul n dan $h(n)$ adalah perkiraan *cost* dari simpul n ke simpul tujuan. Nilai $f(n)$ ini terurut dari yang terkecil. Prosesnya adalah sebagai berikut:

1. Membuat *priority queue* ``openSet`` yang menyimpan node berdasarkan nilai f (gabungan g dan h) dari node A*.
2. Membuat map ``gScoreMap`` yang menyimpan nilai g (cost dari titik awal ke node saat ini).
3. Membuat set ``closedSet`` yang berupa kumpulan kata yang sudah diproses (visited).
4. Node awal dibuat menggunakan start word dengan g bernilai 0 dan h (heuristik) dihitung dengan ``Utils.calculateHeuristic`` antara start word dan end word. Node ini ditambahkan ke ``openSet``.
5. Proses pencarian berlanjut selama ``openSet`` tidak kosong.
6. Node dengan nilai f terendah diambil dari ``openSet``.
7. Jumlah node yang dikunjungi (nodesVisited) bertambah 1.
8. Jika kata saat ini cocok dengan end word, path direkonstruksi dengan melacak dari end word kembali ke start word melalui referensi kata pendahulu di dalam node.
9. Daftar path kemudian dibalik agar menunjukkan urutan dari start word ke end word, lalu dikembalikan dalam ``SearchResult``.
10. Kata saat ini ditambahkan ke ``closedSet`` untuk menandai bahwa sudah diproses.
11. Kata-kata tetangga yang berbeda satu karakter diambil menggunakan ``Utils.getNeighbors``.
12. Untuk setiap tetangga, jika tetangga sudah berada di ``closedSet``, lanjutkan ke kata berikutnya.
13. Hitung nilai g sementara sebagai nilai g dari node saat ini ditambah 1.
14. Jika nilai g sementara ini lebih kecil daripada g yang sudah disimpan di ``gScoreMap`` untuk tetangga atau tetangga belum pernah diproses, buat node baru dengan kata tetangga.
15. Node tetangga ditambahkan ke ``openSet`` dengan nilai g sementara dan h yang dihitung dari kata tetangga ke end word.
16. ``gScoreMap`` diperbarui dengan nilai g sementara untuk tetangga tersebut.
17. Jika tidak ada path antara start word dan end word, maka collections empty dikembalikan dalam ``SearchResult``.

Algoritma A* menggabungkan cost yang sudah dikeluarkan (g) dan nilai heuristik (h) untuk memprioritaskan pencarian kata yang memiliki estimasi cost terendah untuk mencapai end word. Ini memastikan pencarian tetap optimal dengan mempertimbangkan jarak aktual yang telah ditempuh serta estimasi jarak yang tersisa.

4.5 Nilai Heuristik pada A*

Heuristik yang digunakan pada algoritma A* admissible karena untuk setiap node n , $h(n) \leq h^*(n)$, di mana $h^*(n)$ adalah cost sebenarnya untuk mencapai keadaan tujuan dari n . Heuristik yang bersifat *admissible* tidak pernah melebihi-lebihkan *cost* untuk mencapai tujuan, yaitu, heuristik ini bersifat optimis.

4.6 Perbandingan Algoritma UCS dan BFS pada Word Ladder

Urutan node yang dibangkitkan oleh algoritma UCS dan BFS berbeda sehingga menyebabkan path yang dihasilkan tidak selalu sama. Hal ini disebabkan oleh beberapa hal. UCS menggunakan *priority queue* untuk memprioritaskan node dengan cost terendah yang akan dicari berikutnya, sedangkan BFS menggunakan *queue* biasa yang bersifat FIFO (*first-in first-out*), memastikan setiap level dicari sepenuhnya sebelum beralih ke level berikutnya. UCS berfokus pada total *cost* kumulatif dari node awal ke node saat ini. Saat dua node memiliki cost yang sama, urutan pengeluaran dari *priority queue* tergantung pada detail implementasi *queue* tersebut. BFS selalu mencari node dalam urutan level, yang berarti node yang ditambahkan lebih awal akan selalu diproses terlebih dahulu. Ketika *cost* antar-node seragam, seperti dalam word ladder, UCS dan BFS dapat menghasilkan path berbeda karena prioritas pengambilan node dari *queue*. UCS akan memilih node dengan *cost* kumulatif terendah (meskipun costnya sama) berdasarkan urutan dalam *priority queue*, sedangkan BFS akan memilih node sesuai urutan dalam *queue* biasa dan mungkin menghasilkan path yang berbeda jika beberapa node berada pada level yang sama tetapi urutan pencarian berbeda. Meskipun UCS dirancang untuk memberikan *path* dengan *cost* total paling rendah, dalam kasus *cost* seragam, path yang lebih pendek belum tentu sama dengan path yang dicari oleh BFS. BFS secara alami memberikan path terpendek berdasarkan jumlah langkah atau level, sementara UCS masih dapat memberikan path yang berbeda karena urutan penarikan node.

4.7 Perbandingan Algoritma UCS dan A* pada Word Ladder

Secara teoritis, algoritma A* cenderung lebih efisien dibandingkan dengan Uniform Cost Search (UCS) pada kasus word ladder karena A* menggunakan informasi heuristik untuk memandu proses pencarian. A* memanfaatkan fungsi heuristik ' $h(n)$ ' untuk memperkirakan cost tersisa menuju tujuan dari node saat ini. Hal ini membantu memperkecil ruang pencarian hanya ke path yang kemungkinan besar mengarah ke solusi secara lebih efisien. Sebaliknya, UCS hanya mempertimbangkan cost kumulatif yang sudah ditempuh yaitu ' $g(n)$ ', tanpa memanfaatkan heuristik, sehingga semua node harus dijelajahi berdasarkan urutan cost aktual.

Algoritma A* menggabungkan ' $g(n)$ ' (cost aktual) dan ' $h(n)$ ' (perkiraan cost tersisa) dalam fungsi evaluasi ' $f(n) = g(n) + h(n)$ ', sehingga memprioritaskan node yang paling menjanjikan berdasarkan cost total gabungan. Hal ini membuat A* lebih fokus dalam mencari solusi daripada UCS. UCS memperlakukan setiap node secara setara tanpa memprioritaskan arah tertentu, sehingga eksplorasi bisa lebih luas dan kurang terarah. Dengan informasi heuristik yang tepat, A* secara efektif memangkas ruang pencarian dan mempercepat proses menemukan path terpendek. Meskipun UCS dapat menemukan solusi optimal, pencariannya bisa memakan waktu lebih lama dibandingkan A* karena tidak memanfaatkan informasi heuristik yang dapat mempercepat pencarian. Pada kasus word ladder, fungsi heuristik yang sering digunakan adalah jumlah perbedaan karakter antara kata saat ini dan kata tujuan, sehingga heuristik ini memberikan estimasi yang baik mengenai seberapa jauh kata tujuan, yang membantu A* mempersempit ruang pencarian.

4.8 Analisis Solusi *Greedy Best First Search*

Algoritma *Greedy Best First Search* (GBFS) tidak dapat menjamin solusi optimal untuk persoalan word ladder. Ini karena algoritma GBFS hanya mengandalkan fungsi heuristik ' $h(n)$ ' untuk memperkirakan jarak antara node saat ini dengan tujuan, memprioritaskan pencarian node yang dianggap paling dekat dengan tujuan berdasarkan perkiraan tersebut. Namun, karena hanya berfokus pada nilai heuristik, GBFS dapat melewati path yang sebenarnya memiliki cost kumulatif lebih rendah, tetapi tidak tampak menjanjikan dari segi perkiraan heuristik.

Tidak seperti algoritma A* yang menggunakan fungsi evaluasi gabungan $f(n) = g(n) + h(n)$, GBFS hanya bergantung pada $h(n)$ dan mengabaikan cost aktual $g(n)$. Akibatnya, GBFS tidak memperhitungkan cost perjalanan yang sudah ditempuh dari titik awal, membuatnya lebih rentan terhadap path yang mungkin lebih panjang atau mahal dalam hal jumlah transformasi.

Karena GBFS hanya berfokus pada nilai heuristik tanpa mempertimbangkan cost aktual, algoritma ini dapat mengejar path yang tampaknya dekat dengan tujuan, tetapi bukan path optimal. Oleh karena itu, meskipun GBFS dapat menemukan solusi lebih cepat daripada algoritma lain, solusi yang dihasilkan tidak selalu optimal dalam hal jumlah langkah alias bisa saja panjang pathnya lebih banyak.

4.9 Analisis Perbandingan Algoritma UCS, *Greedy Best First Search*, dan A* dalam Word Ladder

Berdasarkan Test Case 1 pada poin 5.1.1 dan 5.2.1, didapat bahwa path terpendek dihasilkan dari UCS dan A*, waktu eksekusi terpendek dihasilkan dari *Greedy Best First Search*, memori terkecil dihasilkan dari *Greedy Best First Search*. Berdasarkan Test Case 2 pada poin 5.1.2 dan 5.2.2, didapat bahwa path terpendek dihasilkan dari UCS, *Greedy Best First Search*, dan A*, waktu eksekusi terpendek dihasilkan dari *Greedy Best First Search*, memori terkecil dihasilkan dari A*. Berdasarkan Test Case 3 pada poin 5.1.3 dan 5.2.3, didapat bahwa path terpendek dihasilkan dari UCS, *Greedy Best First Search*, dan A*, waktu eksekusi terpendek dihasilkan dari A*, memori terkecil dihasilkan dari *Greedy Best First Search*. Berdasarkan Test Case 5 pada poin 5.1.5 dan 5.2.5, didapat bahwa path terpendek dihasilkan dari UCS dan A*, waktu eksekusi terpendek dihasilkan dari *Greedy Best First Search*, memori terkecil dihasilkan dari *Greedy Best First Search*. Berdasarkan Test Case 6 pada poin 5.1.6 dan 5.2.6, didapat bahwa path terpendek dihasilkan dari UCS dan A*, waktu eksekusi terpendek dihasilkan dari *Greedy Best First Search*, memori terkecil dihasilkan dari *Greedy Best First Search*.

Berdasarkan beberapa testcase di atas, berikut adalah analisis perbandingan ketiga algoritma tersebut. Dari segi optimalitas, UCS dan A* secara konsisten memberikan jalur terpendek di sebagian besar kasus. Hal ini terjadi karena algoritma UCS dan A* dirancang untuk selalu mencari solusi optimal. Sedangkan algoritma *Greedy Best First Search* (GBFS)

hanya dapat memberikan jalur optimal pada beberapa kasus. Hal ini terjadi karena algoritma GBFS mengutamakan kecepatan menemukan solusi daripada memastikan solusi tersebut optimal.

Dari segi waktu eksekusi, algoritma *Greedy Best First Search* memiliki waktu eksekusi terpendek dalam sebagian besar kasus uji. Hal ini dikarenakan GBFS lebih fokus pada penyelesaian cepat dengan cara mengevaluasi node yang mendekati target berdasarkan heuristik. Algoritma A* umumnya memiliki waktu eksekusi yang lebih cepat dibandingkan UCS karena juga memanfaatkan heuristik untuk membatasi ruang pencarian. Sedangkan algoritma UCS cenderung membutuhkan waktu lebih lama karena algoritma ini mengevaluasi setiap kemungkinan dengan memastikan bahwa semua node diperiksa dalam urutan *cost*.

Dari segi penggunaan memori, *Greedy Best First Search* umumnya memerlukan jumlah memori yang lebih sedikit karena lebih selektif dalam mengeksplorasi node yang paling mungkin menghasilkan solusi cepat. Algoritma A* dapat membutuhkan lebih banyak memori dibandingkan GBFS karena menyimpan informasi *cost* dan heuristik untuk memastikan solusi optimal. Sedangkan UCS memiliki penggunaan memori tertinggi karena mempertahankan daftar lengkap dari semua node yang telah dikunjungi serta yang akan dikunjungi, memastikan bahwa setiap solusi optimal ditemukan.

Algoritma UCS menyimpan semua node yang telah dan akan dieksplorasi dalam frontier, yang mengarah ke kompleksitas ruang $O(b^d)$. Algoritma *Greedy Best First Search* juga memerlukan penyimpanan semua node yang telah dan akan dieksplorasi dengan kompleksitas ruang $O(b^d)$. Algoritma A* memerlukan penyimpanan untuk semua node yang telah dieksplorasi dan yang akan dieksplorasi dengan kompleksitas ruang $O(b^d)$. Ketiga algoritma tersebut memiliki kompleksitas ruang yang mungkin identik dalam hal teoritis yaitu $O(b^d)$ dimana b adalah faktor percabangan rata-rata (jumlah rata-rata tetangga per node), dan d adalah kedalaman solusi. Kompleksitas ruang mereka tergantung pada struktur graf dan parameter seperti faktor percabangan dan kedalaman solusi.

Algoritma UCS bekerja dengan memperluas path dengan total cost terendah terlebih dahulu, tanpa mempertimbangkan jarak heuristic ke tujuan. Ini berarti UCS bisa sangat tidak efisien, terutama jika faktor percabangan tinggi dan tujuan jauh dari titik start, dengan kompleksitas waktu bisa mencapai $O(b^d)$ dimana b adalah faktor percabangan rata-rata

(jumlah rata-rata tetangga per node), dan d adalah kedalaman solusi. Algoritma *Greedy Best First Search* memiliki kompleksitas waktu yang tergantung pada faktor percabangan dan kedalaman solusi. *Greedy Best First Search* secara eksklusif menggunakan heuristik untuk menentukan node mana yang akan dieksplorasi selanjutnya, mengabaikan cost sejauh ini, yang bisa menyebabkan banyak eksplorasi node yang tidak perlu, terutama di graf yang besar atau kompleks. Ini bisa menyebabkan kompleksitas $O(b^d)$, tetapi sering kali eksplorasi lebih banyak node dibanding A* jika heuristik tidak mengarahkan dengan efektif ke tujuan. Algoritma A* secara umum memiliki kompleksitas waktu $O(b^d)$ juga dimana A* mencoba meminimalkan $f(n) = g(n) + h(n)$, dengan $g(n)$ adalah cost dari node awal ke node n , dan $h(n)$ adalah heuristik dari node n ke node tujuan. Heuristik yang baik dan admissible bisa mengurangi jumlah node yang dieksplorasi secara signifikan dibandingkan dengan algoritma UCS. Efisiensi waktu ketiga algoritma ini bisa sangat berbeda berdasarkan heuristik yang digunakan (untuk A* dan *Greedy Best First Search*) dan karakteristik spesifik graf yang dihadapi. A* umumnya dianggap paling efisien jika heuristik yang baik dan admissible digunakan, karena mencoba menyeimbangkan biaya sejauh ini dan estimasi biaya ke tujuan.

Jika tujuan utamanya adalah menemukan jalur terpendek (optimal), maka UCS dan A* menjadi pilihan terbaik. Namun, A* cenderung lebih cepat daripada UCS. Untuk waktu eksekusi tercepat, GBFS adalah pilihan terbaik meskipun dengan kemungkinan bahwa jalur yang ditemukan tidak optimal. Untuk penggunaan memori yang optimal, GBFS dan A* umumnya membutuhkan lebih sedikit memori daripada UCS. Sehingga jika optimalitas merupakan prioritas utama, maka A* lebih efisien daripada UCS. Namun, jika waktu eksekusi yang lebih cepat menjadi prioritas, maka GBFS adalah pilihan terbaik dengan risiko menemukan solusi yang belum tentu optimal.

BAB V

MASUKAN DAN LUARAN PROGRAM

5.1 CLI

5.1.1 Test Case 1 (Tooth – Fairy)

Uniform Cost Search (UCS)
<pre>PS D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053> ./run.bat D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053>cd src D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>javac -d ../bin *.java D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>xcopy dictionary.txt ../bin\ /Y D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>cd .. D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053>cd bin D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\bin>java Main</pre> <div><p>Welcome to Word Ladder Solver! 13522053</p></div> <pre>Enter start word: tooth Enter end word: fairy Choose an algorithm: 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search 4. Breadth First Search (BFS) Enter the chosen algorithm (UCS, GBFS, A*, BFS): ucs Path length: 8 Path: TOOTH -> TOOTS -> TOOLS -> FOOLS -> FOILS -> FAILS -> FAIRS -> FAIRY Total nodes visited: 4478 Execution time: 212 ms Memory used: 17443120 bytes</pre>
Greedy Best First Search (GBFS)

Do you want to continue? (y/n): y

Enter start word: tooth

Enter end word: fairy

Choose an algorithm:

1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
4. Breadth First Search (BFS)

Enter the chosen algorithm (UCS, GBFS, A*, BFS): gbfs

Path length: 15

Path:

TOOTH -> BOOTH -> BOOTY -> FOOTY -> FORTY -> FORKY -> FOLKY -> FOLLY -> FELLY -> FERLY -> FERRY
-> FIRRY -> FIERY -> FAERY -> FAIRY

Total nodes visited: 20

Execution time: 8 ms

Memory used: 465536 bytes

A*

Do you want to continue? (y/n): y

Enter start word: tooth

Enter end word: fairy

Choose an algorithm:

1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
4. Breadth First Search (BFS)

Enter the chosen algorithm (UCS, GBFS, A*, BFS): a*

Path length: 8

Path:


TOOTH -> TOOTS -> TOITS -> TOILS -> TAILS -> FAILS -> FAIRS -> FAIRY

Total nodes visited: 46

Execution time: 19 ms

Memory used: 673944 bytes

5.1.2 Test Case 2 (Earn – Make)

Uniform Cost Search (UCS)
<pre>PS D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053> ./run.bat D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053>cd src D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>javac -d ../bin *.java D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>xcopy dictionary.txt ../bin\ /Y D:dictionary.txt 1 File(s) copied D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>cd .. D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053>cd bin D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\bin>java Main</pre>  <p>Welcome to Word Ladder Solver! 13522053</p> <pre>Enter start word: earn Enter end word: make Choose an algorithm: 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search 4. Breadth First Search (BFS) Enter the chosen algorithm (UCS, GBFS, A*, BFS): ucs Path length: 5 Path: EARN -> CARN -> CARE -> CAKE -> MAKE Total nodes visited: 1277 Execution time: 105 ms Memory used: 7503872 bytes</pre>
Greedy Best First Search (GBFS)

Do you want to continue? (y/n): y

Enter start word: earn

Enter end word: make

Choose an algorithm:

1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
4. Breadth First Search (BFS)

Enter the chosen algorithm (UCS, GBFS, A*, BFS): gbfs

Path length: 5

Path:

EARN -> BARN -> BARE -> MARE -> MAKE

Total nodes visited: 5

Execution time: 8 ms

Memory used: 300552 bytes

A*

Do you want to continue? (y/n): y

Enter start word: earn

Enter end word: make

Choose an algorithm:

1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
4. Breadth First Search (BFS)

Enter the chosen algorithm (UCS, GBFS, A*, BFS): a*

Path length: 5

Path:

EARN -> TARN -> TARE -> TAKE -> MAKE

Total nodes visited: 18


Execution time: 14 ms

Memory used: 250464 bytes

Do you want to continue? (y/n): n

Exiting the Word Ladder. Goodbye!

5.1.3 Test Case 3 (Free – Form)

Uniform Cost Search (UCS)
<pre>PS D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053> ./run.bat D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053>cd src D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>javac -d ../bin *.java D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>xcopy dictionary.txt ../bin\ /Y D:dictionary.txt 1 File(s) copied D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>cd .. D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053>cd bin D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\bin>java Main</pre>  <pre> Welcome to Word Ladder Solver! 13522053 Enter start word: free Enter end word: form Choose an algorithm: 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search 4. Breadth First Search (BFS) Enter the chosen algorithm (UCS, GBFS, A*, BFS): ucs Path length: 7 Path: FREE -> FLEE -> FLEY -> FLAY -> FLAM -> FOAM -> FORM Total nodes visited: 2022 Execution time: 124 ms Memory used: 12088016 bytes</pre>
Greedy Best First Search (GBFS)

Do you want to continue? (y/n): y

Enter start word: free

Enter end word: form

Choose an algorithm:

1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
4. Breadth First Search (BFS)

Enter the chosen algorithm (UCS, GBFS, A*, BFS): gbfs

Path length: 7

Path:

FREE -> FLEE -> FLEY -> FLAY -> FLAM -> FOAM -> FORM

Total nodes visited: 21

Execution time: 17 ms

Memory used: 303064 bytes

A*

Do you want to continue? (y/n): y

Enter start word: free

Enter end word: form

Choose an algorithm:

1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
4. Breadth First Search (BFS)

Enter the chosen algorithm (UCS, GBFS, A*, BFS): a*

Path length: 7

Path:

FREE -> FRAE -> FRAY -> FLAY -> FLAM -> FOAM -> FORM

Total nodes visited: 83

Execution time: 16 ms

Memory used: 621288 bytes

5.1.4 Test Case 4 (Believer – Evaluate)

Uniform Cost Search (UCS)
<pre>PS D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053> ./run.bat D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053>cd src D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>javac -d ../bin *.java D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>xcopy dictionary.txt ../bin\ /Y D:dictionary.txt 1 File(s) copied D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>cd .. D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053>cd bin D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\bin>java Main WORD LADDER Welcome to Word Ladder Solver! 13522053 Enter start word: believer Enter end word: evaluate Choose an algorithm: 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search 4. Breadth First Search (BFS) Enter the chosen algorithm (UCS, GBFS, A*, BFS): ucs No path found.</pre>
Greedy Best First Search (GBFS)
<pre>Do you want to continue? (y/n): y Enter start word: believer Enter end word: evaluate Choose an algorithm: 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search 4. Breadth First Search (BFS) Enter the chosen algorithm (UCS, GBFS, A*, BFS): gbfs No path found.</pre>

A*

Do you want to continue? (y/n): y

Enter start word: believer

Enter end word: evaluate

Choose an algorithm:

1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
4. Breadth First Search (BFS)

Enter the chosen algorithm (UCS, GBFS, A*, BFS): a*

No path found.

Do you want to continue? (y/n): n

Exiting the Word Ladder. Goodbye!

5.1.5 Test Case 5 (Sand – Crab)

Uniform Cost Search (UCS)
<pre>PS D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053> ./run.bat D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053>cd src D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>javac -d ../bin *.java D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>xcopy dictionary.txt ../bin\ /Y D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src> 1 File(s) copied D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>cd .. D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053>cd bin D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\bin>java Main WORD LADDER Welcome to Word Ladder Solver! 13522053 Enter start word: sand Enter end word: crab Choose an algorithm: 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search 4. Breadth First Search (BFS) Enter the chosen algorithm (UCS, GBFS, A*, BFS): ucs Path length: 8 Path: SAND -> BAND -> BEND -> BEAD -> BRAD -> BRAW -> CRAW -> CRAB Total nodes visited: 3723 Execution time: 196 ms Memory used: 7886760 bytes</pre>
Greedy Best First Search (GBFS)

Do you want to continue? (y/n): y

Enter start word: sand

Enter end word: crab

Choose an algorithm:

1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
4. Breadth First Search (BFS)

Enter the chosen algorithm (UCS, GBFS, A*, BFS): gbfs

Path length: 11

Path:

SAND -> SANS -> CANS -> CONS -> COSS -> COST -> COAT -> CHAT -> CHAM -> CRAM -> CRAB

Total nodes visited: 79

Execution time: 19 ms

Memory used: 899744 bytes

A*

Do you want to continue? (y/n): y

Enter start word: sand

Enter end word: crab

Choose an algorithm:

1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
4. Breadth First Search (BFS)

Enter the chosen algorithm (UCS, GBFS, A*, BFS): a*

Path length: 8

Path:

SAND -> SAID -> CAID -> CHID -> CHIS -> CRIS -> CRIB -> CRAB

Total nodes visited: 274

Execution time: 41 ms

Memory used: 1945832 bytes

5.1.6 Test Case 6 (Flower – Sunset)

Uniform Cost Search (UCS)
<pre>PS D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053> ./run.bat D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053>cd src D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>javac -d ../bin *.java D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>xcopy dictionary.txt ../bin\ /Y D:dictionary.txt 1 File(s) copied D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>cd .. D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053>cd bin D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\bin>java Main WORD LADDER Welcome to Word Ladder Solver! 13522053 Enter start word: flower Enter end word: sunset Choose an algorithm: 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search 4. Breadth First Search (BFS) Enter the chosen algorithm (UCS, GBFS, A*, BFS): ucs Path length: 14 Path: FLOWER -> FLOWED -> FLAWED -> FLAKED -> SLAKED -> SOAKED -> SOCKED -> SUCKED -> BUCKED -> BUNKE D -> JUNKED -> JUNKET -> SUNKET -> SUNSET Total nodes visited: 6187 Execution time: 220 ms Memory used: 14907072 bytes</pre>
Greedy Best First Search (GBFS)

Do you want to continue? (y/n): y

Enter start word: flower

Enter end word: sunset

Choose an algorithm:

1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
4. Breadth First Search (BFS)

Enter the chosen algorithm (UCS, GBFS, A*, BFS): gbfs

Path length: 35

Path:

FLOWER -> SLOWER -> SLOWED -> SLOPED -> SLIPED -> SWIPED -> SWIVED -> SWIVES -> SKIVES -> SKIVE
R -> SLIVER -> SLAVER -> SLAVEY -> SLATEY -> SLATES -> STATES -> STALES -> STALED -> SEALED ->
SEELED -> SEEDED -> SENDED -> SANDED -> SANDER -> SANGER -> SINGER -> SINNER -> SINNED -> SUNNE
D -> DUNNED -> DUNKED -> JUNKED -> JUNKET -> SUNKET -> SUNSET

Total nodes visited: 344

Execution time: 32 ms

Memory used: 3006128 bytes

A*

Do you want to continue? (y/n): y

Enter start word: flower

Enter end word: sunset

Choose an algorithm:

1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
4. Breadth First Search (BFS)

Enter the chosen algorithm (UCS, GBFS, A*, BFS): a*

Path length: 14

Path:

FLOWER -> FLOWED -> FLAWED -> BLAWED -> BLADED -> BEADED -> BENDED -> FENDED -> FUNDED -> FUNKE
D -> JUNKED -> JUNKET -> SUNKET -> SUNSET

Total nodes visited: 1837

Execution time: 57 ms

Memory used: 14474400 bytes

Do you want to continue? (y/n): n

Exiting the Word Ladder. Goodbye!

5.1.7 Test Case 7 (Word not in english)

```
PS D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053> ./run.bat

D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053>cd src

D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>javac -d ../bin *.java

D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>xcopy dictionary.txt ../bin\ /Y
D:dictionary.txt
1 File(s) copied

D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>cd ..

D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053>cd bin

D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\bin>java Main

WORD LADDER

Welcome to Word Ladder Solver!
13522053

Enter start word: tangan
Enter end word: sarung
Start word and end word must be words in English!

Enter start word: █
```

5.1.8 Test Case 8 (Start or/and end word empty)

```
PS D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053> ./run.bat

D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053>cd src

D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>javac -d ../bin *.java

D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>xcopy dictionary.txt ../bin\ /Y
D:dictionary.txt
1 File(s) copied

D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>cd ..

D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053>cd bin

D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\bin>java Main

WORD LADDER

Welcome to Word Ladder Solver!
13522053

Enter start word:
Enter end word:
Start word and end word must not be empty!

Enter start word: █
```

5.1.9 Test Case 9 (Word length not equal)

```
PS D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053> ./run.bat

D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053>cd src

D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>javac -d ../bin *.java

D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>xcopy dictionary.txt ../bin/ /Y
D:dictionary.txt
1 File(s) copied

D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\src>cd ..

D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053>cd bin

D:\ITB\Tugas Stima\Tucil\Tucil 3\Tucil3_13522053\bin>java Main

  WORD LADDER

      Welcome to Word Ladder Solver!
      13522053

Enter start word: stranger
Enter end word: love
Start word and end word must be of equal length.

Enter start word: █
```


5.2 GUI

5.2.1 Test Case 1 (Tooth – Fairy)

Uniform Cost Search (UCS)

Word Ladder Solver

Start Word:

tooth

End Word:

fairy

Algorithm:

UCS

Search!

T	O	O	T	H
T	O	O	T	S
T	O	O	L	S
F	O	O	L	S
F	O	I	L	S
F	A	I	L	S
F	A	I	R	S
F	A	I	R	Y

Path length: 8

Total nodes visited: 4478

Execution time: 132 ms

Greedy Best First Search (GBFS)

Word Ladder Solver

Start Word:

tooth

End Word:

fairy

Algorithm:

Greedy BFS

Search!

T	O	O	T	H
B	O	O	T	H
B	O	O	T	Y
F	O	O	T	Y
F	O	R	T	Y
F	O	R	K	Y
F	O	L	K	Y
F	O	L	L	Y
F	E	L	L	Y
F	E	R	L	Y
F	E	R	R	Y
F	I	R	R	Y
F	I	E	R	Y
F	A	E	R	Y
F	A	I	R	Y

Path length: 15

Total nodes visited: 20

Execution time: 9 ms

A*

Word Ladder Solver

Start Word:

tooth

End Word:

fairy

Algorithm:

A*

Search!

T	O	O	T	H
T	O	O	T	S
T	O	I	T	S
T	O	I	L	S
T	A	I	L	S
F	A	I	L	S
F	A	I	R	S
F	A	I	R	Y

Path length: 8

Total nodes visited: 46

Execution time: 16 ms

5.2.2 Test Case 2 (Earn – Make)

Uniform Cost Search (UCS)

Word Ladder Solver

Start Word:

earn

End Word:

make

Algorithm:

UCS

Search!

E	A	R	N
C	A	R	N
C	A	R	E
C	A	K	E
M	A	K	E

Path length: 5

Total nodes visited: 1277

Execution time: 43 ms

Greedy Best First Search (GBFS)

Word Ladder Solver

Start Word:

earn

End Word:

make

Algorithm:

Greedy BFS

Search!

E	A	R	N
B	A	R	N
B	A	R	E
M	A	R	E
M	A	K	E

Path length: 5

Total nodes visited: 5

Execution time: 0 ms

A*

Word Ladder Solver

Start Word:

earn

End Word:

make

Algorithm:

A*

Search!

E	A	R	N
T	A	R	N
T	A	R	E
T	A	K	E
M	A	K	E

Path length: 5

Total nodes visited: 18

Execution time: 0 ms

5.2.3 Test Case 3 (Free – Form)

Uniform Cost Search (UCS)

Word Ladder Solver

Start Word:

free

End Word:

form

Algorithm:

UCS

Search!

F	R	E	E
F	L	E	E
F	L	E	Y
F	L	A	Y
F	L	A	M
F	O	A	M
F	O	R	M

Path length: 7

Total nodes visited: 2022

Execution time: 82 ms

Greedy Best First Search (GBFS)

Word Ladder Solver

Start Word:

free

End Word:

form

Algorithm:

Greedy BFS

Search!

F	R	E	E
F	L	E	E
F	L	E	Y
F	L	A	Y
F	L	A	M
F	O	A	M
F	O	R	M

Path length: 7

Total nodes visited: 21

Execution time: 0 ms

A*

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Search!

F	R	E	E
F	R	A	E
F	R	A	Y
F	L	A	Y
F	L	A	M
F	O	A	M
F	O	R	M

Path length: 7

Total nodes visited: 83

Execution time: 4 ms

5.2.4 Test Case 4 (Believer - Evaluate)

Uniform Cost Search (UCS)

Word Ladder Solver

Start Word:

End Word:

Algorithm:

Search!

No path found.

Path length: 0

Total nodes visited: 7
Execution time: 16 ms

Greedy Best First Search (GBFS)

Word Ladder Solver

Start Word:

End Word:

Algorithm:

No path found.

Path length: 0

Total nodes visited: 8
Execution time: 8 ms

A*

Word Ladder Solver

Start Word:

End Word:

Algorithm:

No path found.

Path length: 0

Total nodes visited: 7
Execution time: 0 ms

5.2.5 Test Case 5 (Sand – Crab)

Uniform Cost Search (UCS)

Word Ladder Solver

Start Word:

sand

End Word:

crab

Algorithm:

UCS

Search!

S	A	N	D
B	A	N	D
B	E	N	D
B	E	A	D
B	R	A	D
B	R	A	W
C	R	A	W
C	R	A	B

Path length: 8

Total nodes visited: 3723

Execution time: 71 ms

Greedy Best First Search (GBFS)

Word Ladder Solver

Start Word:

sand

End Word:

crab

Algorithm:

Greedy BFS

Search!

S	A	N	D
S	A	N	S
C	A	N	S
C	O	N	S
C	O	S	S
C	O	S	T
C	O	A	T
C	H	A	T
C	H	A	M
C	R	A	M
C	R	A	B

Path length: 11

Total nodes visited: 79

Execution time: 1 ms

A*

Word Ladder Solver

Start Word:

sand

End Word:

crab

Algorithm:

A*

Search!

S	A	N	D
S	A	I	D
C	A	I	D
C	H	I	D
C	H	I	S
C	R	I	S
C	R	I	B
C	R	A	B

Path length: 8

Total nodes visited: 274

Execution time: 10 ms

5.2.6 Test Case 6 (Flower - Sunset)

Uniform Cost Search (UCS)

Word Ladder Solver

Start Word:

flower

End Word:

sunset

Algorithm:

UCS

Search!

F	L	O	W	E	R
F	L	O	W	E	D
F	L	A	W	E	D
F	L	A	K	E	D
S	L	A	K	E	D
S	O	A	K	E	D
S	O	C	K	E	D
S	U	C	K	E	D
B	U	C	K	E	D
B	U	N	K	E	D
J	U	N	K	E	D
J	U	N	K	E	T
S	U	N	K	E	T
S	U	N	S	E	T

Path length: 14

Total nodes visited: 6187

Execution time: 73 ms

Greedy Best First Search (GBFS)

Word Ladder Solver

Start Word:

flower

End Word:

sunset

Algorithm:

Greedy BFS

Search!

F	L	O	W	E	R
S	L	O	W	E	R
S	L	O	W	E	D
S	L	O	P	E	D
S	L	I	P	E	D
S	W	I	P	E	D
S	W	I	V	E	D
S	W	I	V	E	S
S	K	I	V	E	S
S	K	I	V	E	R
S	L	I	V	E	R
S	L	A	V	E	R
S	L	A	V	E	Y
S	L	A	T	E	Y
S	L	A	T	E	S
S	T	A	T	E	S
S	T	A	I	E	S

S	T	A	L	E	S
S	T	A	L	E	D
S	E	A	L	E	D
S	E	E	L	E	D
S	E	E	D	E	D
S	E	N	D	E	D
S	A	N	D	E	D
S	A	N	D	E	R
S	A	N	G	E	R
S	I	N	G	E	R
S	I	N	N	E	R
S	I	N	N	E	D
S	U	N	N	E	D
D	U	N	N	E	D
D	U	N	K	E	D
J	U	N	K	E	D
J	U	N	K	E	T
S	U	N	K	E	T
S	U	N	S	E	T
Path length: 35 Total nodes visited: 344 Execution time: 14 ms					
A*					

Word Ladder Solver

Start Word:

flower

End Word:

sunset

Algorithm:

A*

Search!

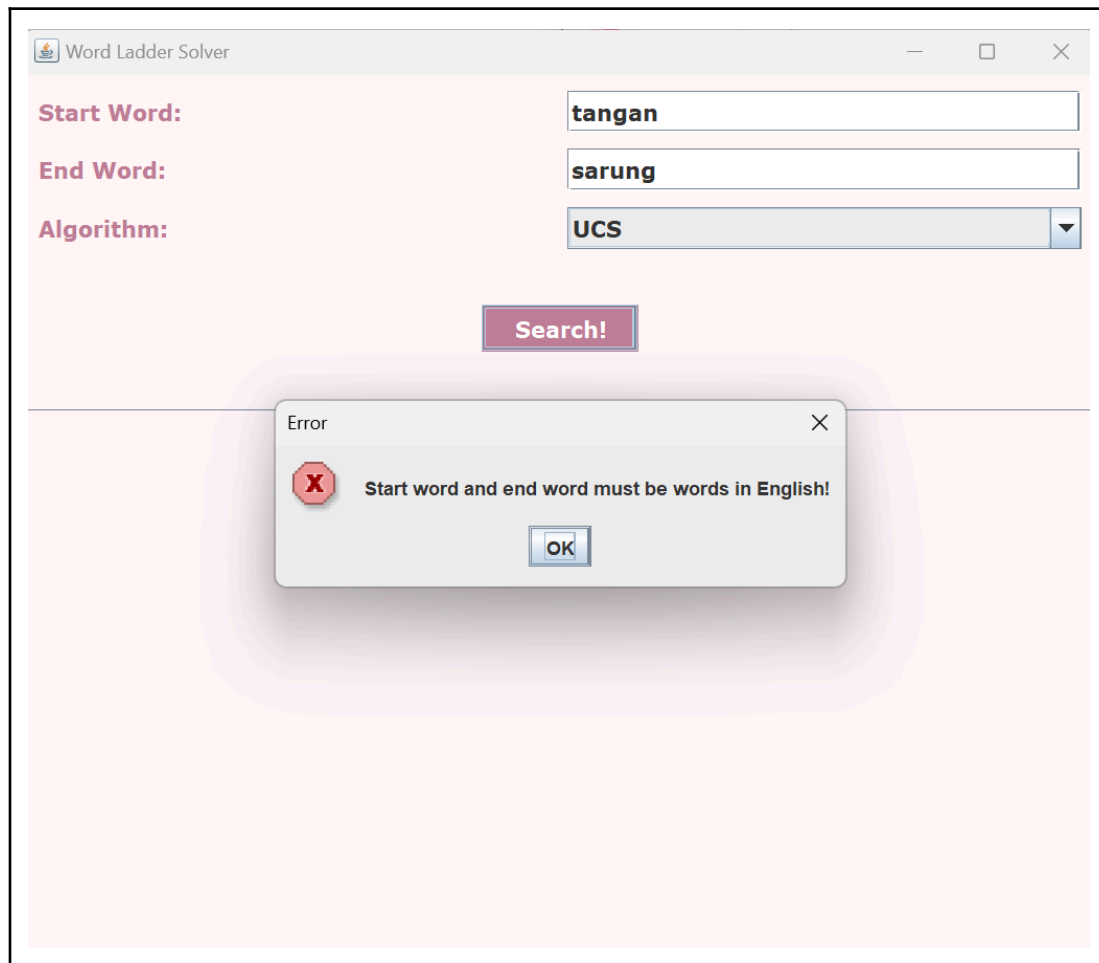
F	L	O	W	E	R
F	L	O	W	E	D
F	L	A	W	E	D
B	L	A	W	E	D
B	L	A	D	E	D
B	E	A	D	E	D
B	E	N	D	E	D
F	E	N	D	E	D
F	U	N	D	E	D
F	U	N	K	E	D
J	U	N	K	E	D
J	U	N	K	E	T
S	U	N	K	E	T
S	U	N	S	E	T

Path length: 14

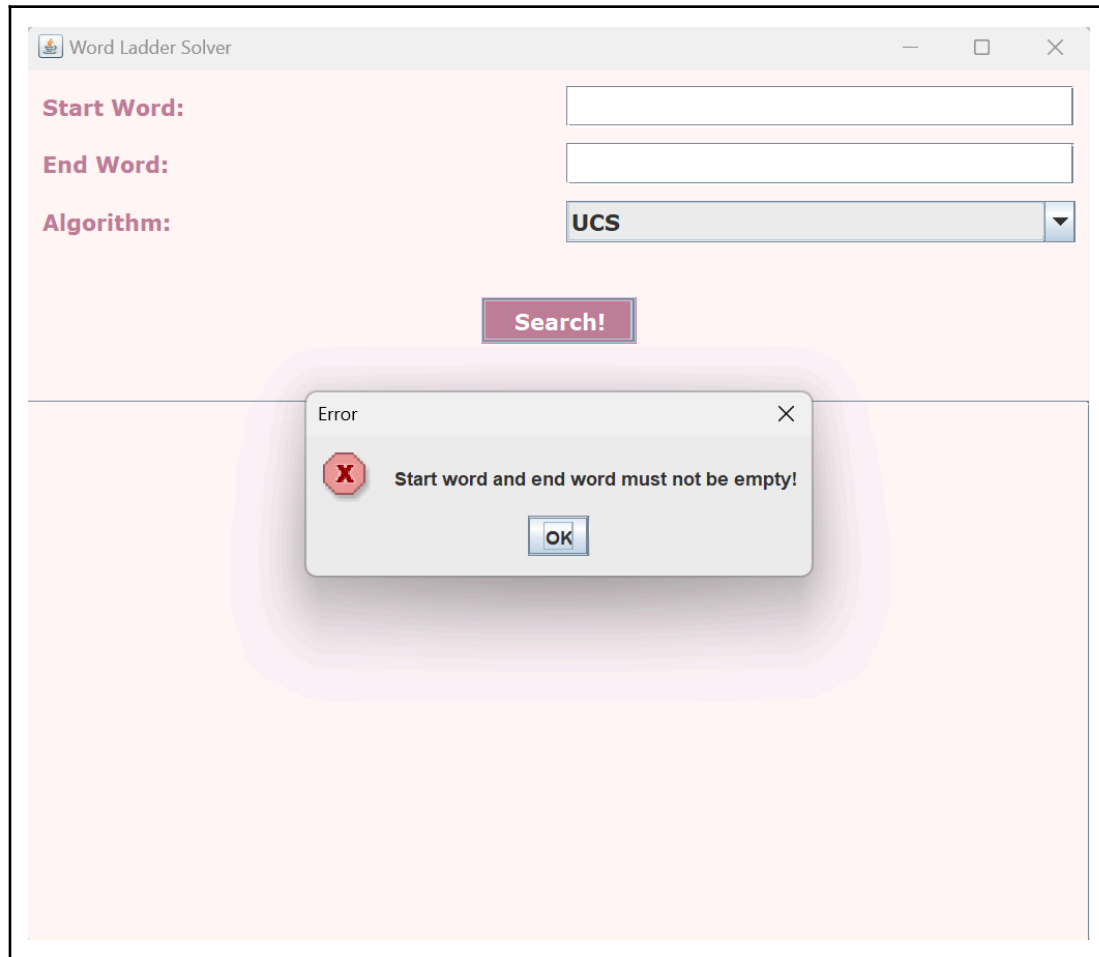
Total nodes visited: 1837

Execution time: 50 ms

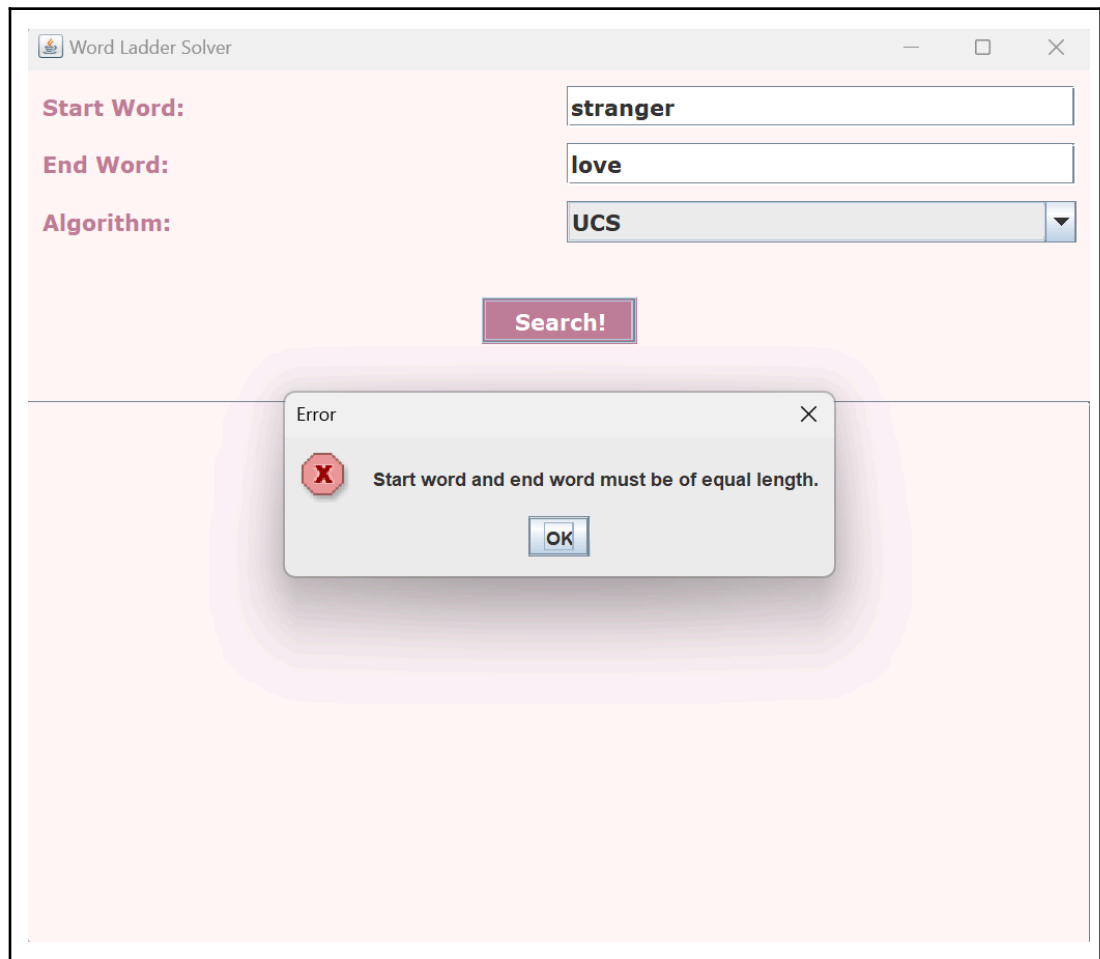
5.2.7 Test Case 7 (Word not in english)



5.2.8 Test Case 8 (Start or/and word empty)



5.2.9 Test Case 9 (Word length not equal)



BAB VI

KESIMPULAN

6.1 Kesimpulan

Melalui Tugas Kecil 3 Strategi Algoritma ini, saya diminta untuk membuat program yang dapat mencari path terpendek (paling optimal) antara dua buah kata berbasis permainan Word Ladder. Saya membuat program ini dengan bahasa Java dan juga membuat GUI dengan menggunakan Java Swing. Melalui pengujian yang telah dilakukan, algoritma A* cenderung paling optimal untuk permainan Word Ladder ini karena dapat menemukan path terpendek dengan waktu yang tidak terlalu lama dan tidak memakan terlalu banyak memori.

LAMPIRAN

No	Poin	Ya	Tidak
1.	Program berhasil dijalankan.	✓	
2.	Program dapat menemukan rangkaian kata dari <i>start word</i> ke <i>end word</i> sesuai aturan permainan dengan algoritma UCS	✓	
3.	Solusi yang diberikan pada algoritma UCS optimal	✓	
4.	Program dapat menemukan rangkaian kata dari <i>start word</i> ke <i>end word</i> sesuai aturan permainan dengan algoritma <i>Greedy Best First Search</i>	✓	
5.	Program dapat menemukan rangkaian kata dari <i>start word</i> ke <i>end word</i> sesuai aturan permainan dengan algoritma A*	✓	
6.	Solusi yang diberikan pada algoritma A* optimal	✓	
7.	[Bonus]: Program memiliki tampilan GUI	✓	

DAFTAR REFERENSI

- [1] <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf>
- [2] <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf>