LAPORAN TUGAS KECIL 3 IF2211
STRATEGI ALGORITMA
SEMESTER II TAHUN 2023/2024

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

Disusun oleh:

Salsabiila (13522062)

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

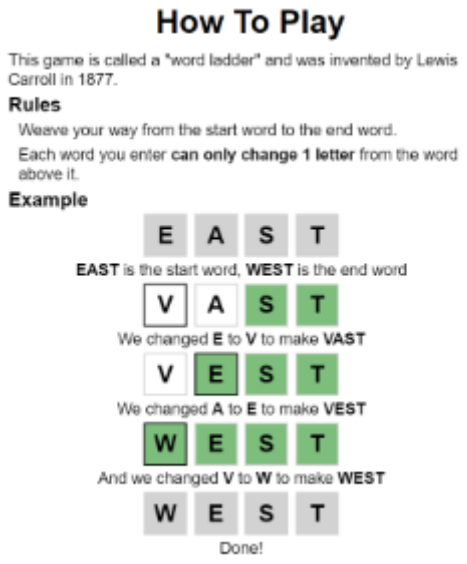
2023

DAFTAR ISI

DAFTAR ISI **1**
BAB I **2**
BAB II **3**
2.1. Algoritma Uniform Cost Search 3
2.2. Algoritma Greedy Best First Search 4
2.3. Algoritma A* Search 6
BAB III **8**
3.1. Words.java 8
3.2. Node.java 9
3.3. Result.java 11
3.4. UCS.java 11
3.5. GreedySearch.java 13
3.6. Astar.java 14
BAB IV **17**
4.1. Test Case 1 17
4.2. Test Case 2 19
4.3. Test Case 3 22
4.4. Test Case 4 27
4.5. Test Case 5 28
4.6. Test Case 6 30
4.7. Error Handling 31
BAB V **36**
BAB VI
IMPLEMENTASI BONUS **37**
LAMPIRAN **44**
DAFTAR PUSTAKA **45**

BAB I
DESKRIPSI MASALAH

Word ladder (juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragrams, laddergrams, atau word golf) adalah salah satu permainan kata yang terkenal bagi seluruh kalangan. Word ladder ditemukan oleh Lewis Carroll, seorang penulis dan matematikawan, pada tahun 1877. Pada permainan ini, pemain diberikan dua kata yang disebut sebagai start word dan end word. Untuk memenangkan permainan, pemain harus menemukan rantai kata yang dapat menghubungkan antara start word dan end word. Banyaknya huruf pada start word dan end word selalu sama. Tiap kata yang berdekatan dalam rantai kata tersebut hanya boleh berbeda satu huruf saja. Pada permainan ini, diharapkan solusi optimal, yaitu solusi yang meminimalkan banyaknya kata yang dimasukkan pada rantai kata. Berikut adalah ilustrasi serta aturan permainan.

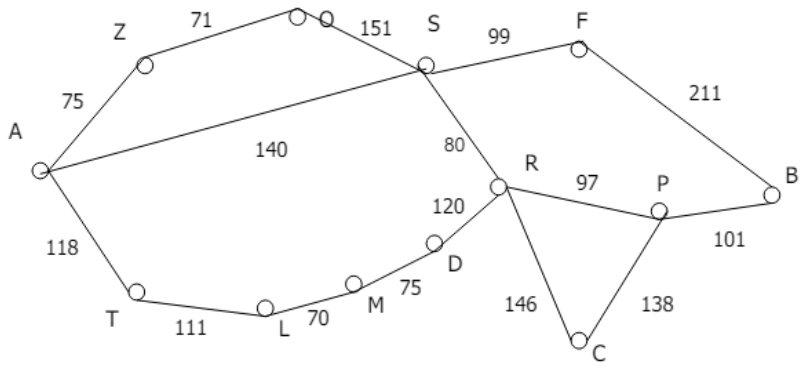


Gambar 1. Ilustrasi dan Peraturan Permainan Word Ladder (Sumber: https://wordwormdormdork.com/)

BAB II
ANALISIS ALGORITMA

2.1. Algoritma Uniform Cost Search

Algoritma UCS (Uniform Cost Search) adalah algoritma pencarian yang digunakan untuk menemukan jalur terpendek antara simpul awal dan simpul tujuan dalam sebuah graf berbobot. Pada kasus umu, algoritma ini mirip dengan algoritma pencarian breadth-first search (BFS), namun berbeda dalam cara menentukan urutan ekspansi simpul.



Gambar 2.1.1. Ilustrasi Graf Berbobot (Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/stima23-24.htm)

Pada pencarian rute terpendek dari simpul A ke simpul B pada graf gambar 2.1, akan digunakan *priority queue* sebagai struktur data simpul hidup yang mengurutkan elemennya berdasarkan fungsi $g(n)$ yang merupakan jarak atau *path cost* dari satu simpul ke simpul tetangganya. Pemilihan simpul ekspan dilakukan dengan memilih simpul yang memiliki *total cost* terendah. Tabel berikut merupakan proses iterasi dari penggunaan algoritma UCS untuk mencapai simpul target B dari simpul awal A.

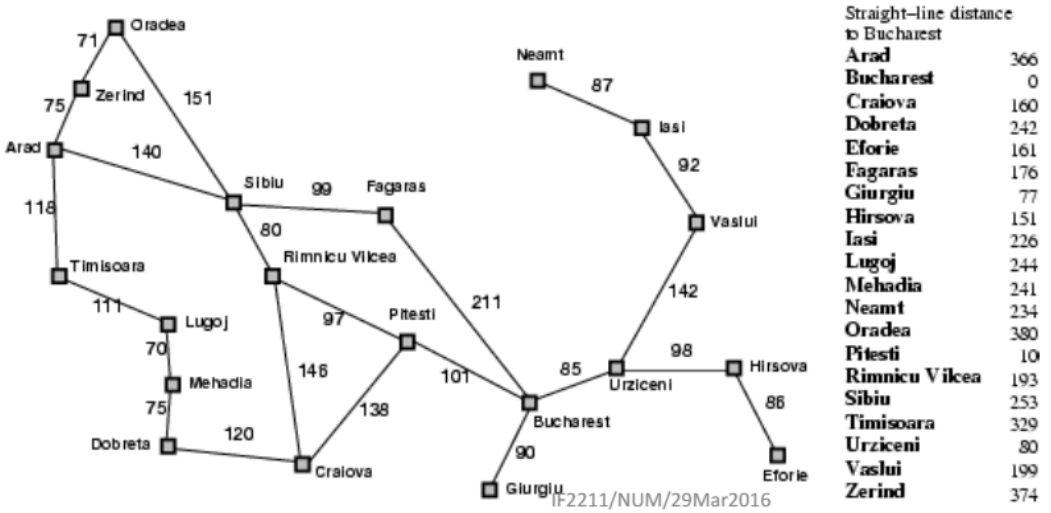
Simpul-E	Simpul Hidup
A	Z_{A-75}, T_{A-118}, S_{A-140}
Z_{A-75}	T_{A-118}, S_{A-140}, O_{AZ-146}
T_{A-118}	S_{A-140}, O_{AZ-146}, L_{AT-229}
S_{A-140}	O_{AZ-146}, R_{AS-220}, L_{AT-229}, F_{AS-239}, O_{AS-291}
O_{AZ-146}	R_{AS-220}, L_{AT-229}, F_{AS-239}, O_{AS-291}
R_{AS-220}	L_{AT-229}, F_{AS-239}, O_{AS-291}, $P_{ASR-317}$, $D_{ASR-340}$, $C_{ASR-366}$
L_{AT-229}	F_{AS-239}, O_{AS-291}, $M_{ATL-299}$, $P_{ASR-317}$, $D_{ASR-340}$, $C_{ASR-366}$
F_{AS-239}	O_{AS-291}, $M_{ATL-299}$, $P_{ASR-317}$, $D_{ASR-340}$, $C_{ASR-366}$, $B_{ASF-450}$
O_{AS-291}	$M_{ATL-299}$, $P_{ASR-317}$, $D_{ASR-340}$, $C_{ASR-366}$, $B_{ASF-450}$
$M_{ATL-299}$	$P_{ASR-317}$, $D_{ASR-340}$, $D_{ATLM-364}$, $C_{ASR-366}$, $B_{ASF-450}$
$P_{ASR-317}$	$D_{ASR-340}$, $D_{ATLM-364}$, $C_{ASR-366}$, $B_{ASRP-418}$, $C_{ASRP-455}$, $B_{ASF-450}$
$D_{ASR-340}$	$D_{ATLM-364}$, $C_{ASR-366}$, $B_{ASRP-418}$, $C_{ASRP-455}$, $B_{ASF-450}$
$D_{ATLM-364}$	$C_{ASR-366}$, $B_{ASRP-418}$, $C_{ASRP-455}$, $B_{ASF-450}$
$C_{ASR-366}$	$B_{ASRP-418}$, $C_{ASRP-455}$, $B_{ASF-450}$
$B_{ASRP-418}$	Solusi ketemu

Gambar 2.1.2. Tabel Iterasi Pencarian dengan Algoritma UCS (Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/stima23-24.htm)

Pada konteks penyelesaian permainan Word Ladder, algoritma UCS dan algoritma BFS mirip tetapi tidak sepenuhnya sama. Pada BFS, semua *path cost* dari satu node ke node lain tidak diperhitungkan dan dianggap sama sehingga tidak dibutuhkan *priority queue* karena akan mengunjungi setiap simpul di tiap kedalamannya. Perbedaan BFS dan UCS pada konteks ini hanya pada pengecekan *cost*-nya. Meskipun pada permainan Word Ladder *path cost* dari satu simpul ke simpul lain bernilai sama, tetap dilakukan pengecekan *total cost* untuk memastikan mendapat jalur terpendek menuju simpul tujuan

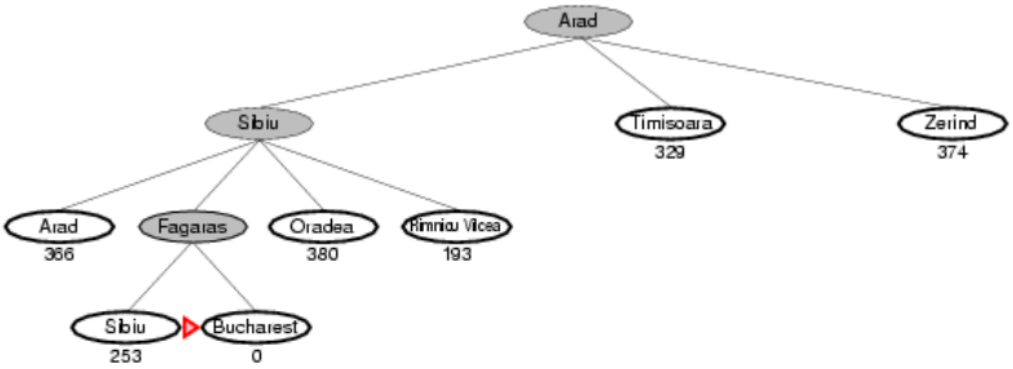
2.2. Algoritma Greedy Best First Search

Algoritma Greedy Best First Search adalah algoritma pencarian yang mengutamakan ekspansi simpul berdasarkan fungsi heuristik $h(n)$ yang merupakan estimasi *cost* dari simpul ke simpul akhir. Pada setiap permasalahan, fungsi heuristik $h(n)$ dapat memiliki definisi yang berbeda-beda. Berikut merupakan salah satu contoh kasus penggunaan algoritma ini.



Gambar 2.2.1 Ilustrasi Graf Berbobot Antar Kota (Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/stima23-24.htm)

Pada pencarian rute terpendek dari Kota Arad ke Kota Bucharest seperti ilustrasi pada gambar 2.2.1, fungsi heuristik $h(n)$ didefinisikan sebagai jarak kota simpul yang ditarik lurus ke Kota Bucharest. Berikut merupakan iterasi pencarian dengan menggunakan algoritma ini.

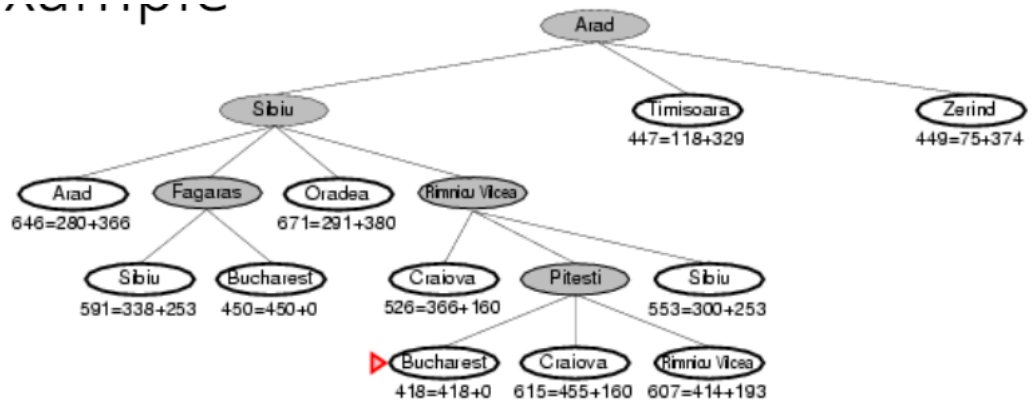


Gambar 2.2.2 Iterasi Pencarian dengan Algoritma Greedy Best First Search (Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/stima23-24.htm)

Meskipun algoritma ini memiliki waktu *run time* yang relatif cepat karena pemilihan simpul yang instan, hasil pencarian yang menggunakan algoritma Greedy Best First Search tidak menjamin untuk menghasilkan solusi optimal karena pemilihan simpul ekspan yang hanya berdasarkan nilai estimasi heuristik ke simpul tujuan. Algoritma greedy ini akan selalu memilih nilai optimum lokal dengan harapan nilai optimum lokal yang dipilih merupakan nilai optimum global dan oleh sebab itulah mengapa algoritma ini tidak menjamin dapat menghasilkan solusi optimal.

2.3. Algoritma A* Search

Algoritma A* adalah algoritma pencarian yang menggabungkan pendekatan Greedy Best First Search dengan informasi tambahan dari biaya sejauh ini, menciptakan kombinasi optimalisasi dan efisiensi. Pendekatan algoritma ini dapat dikatakan sebagai gabungan dari pendekatan algoritma UCS dan algoritma Greedy Best First Search. Dalam A*, setiap simpul dinilai berdasarkan biaya total yang diperlukan untuk simpul tujuan dari simpul awal, ditambah estimasi jarak tersisa ke simpul tujuan menggunakan fungsi heuristik. Algoritma ini memilih simpul dengan biaya terendah secara keseluruhan dengan menggunakan fungsi evaluasi $f(n) = g(n) + h(n)$ di mana $g(n)$ adalah *total cost* hingga sampai ke simpul ekspan dari simpul awal dan $h(n)$ adalah *estimated cost* atau nilai heuristik hingga mencapai simpul tujuan dari simpul ekspan saat ini. Dengan menggunakan perhitungan ini memungkinkan A* untuk menemukan jalur optimal ke simpul tujuan, selama fungsi heuristik memenuhi kondisi *admissibility* dan *consistency*.



Gambar 2.3 Iterasi Pencarian dengan Algoritma A* Berdasarkan Persoalan Gambar 2.2.1
(Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/stima23-24.htm)

Pada konteks penyelesaian permainan Word Ladder, penggunaan algoritma A* secara konsisten dapat memberikan solusi optimal dengan waktu tercepat. Hal ini bisa tercapai salah satunya karena fungsi heuristik yang digunakan bersifat *admissable* yang artinya nilai heuristik tidak akan pernah meng-*overestimate* langkah yang dibutuhkan untuk mencapai simpul tujuan. Untuk mencapai suatu kata dari kata awal, jumlah langkah yang diambil tidak mungkin lebih dari jumlah perbedaan karakternya. Oleh sebab itulah fungsi heuristik pada *Word Ladder Solver* memenuhi *admissibility* dan dapat memberikan solusi yang optimum dengan efisien.

BAB III
SOURCE CODE

3.1. Words.java

Kelas Words bertanggung jawab dalam meng-*handle* pekerjaan yang berhubungan dengn kata dan pembuatan *dictionary*.

Method	Keterangan
createWordMap(String filePath)	Membuat sebuah map dengan key panjang kata dan value list dari kata-kata tersebut
countCharDifference(String word1, String word2)	Menghitung jumlah perbedaan karakter antara word1 dengan word2
isExists(String word, List<String> wordList, int type)	Melakukan pengecekan apakah kata terdapat dalam dictionary
isSameLength(String startWord, String targetWord)	Melakukan pengecekan apakah startWord dan targetWord memiliki panjang yang sama
isInputValid(String word)	Melakukan pengecekan apakah input kata dari pengguna adalah valid.

```
package util;
import java.io.*;
import java.util.*;

public class Words {
    public static Map<Integer, List<String>> createWordMap(String
filePath) {
        Map<Integer, List<String>> map = new HashMap<>();

        try (BufferedReader reader = new BufferedReader(new
FileReader(filePath))) {
            String line;
            while ((line = reader.readLine()) != null) {
                line = line.trim();
                int wordLength = line.length();

                map.putIfAbsent(wordLength, new ArrayList<>());
                map.get(wordLength).add(line);
            }

        } catch (IOException e) {
            e.printStackTrace();
        }

        return map;
```

```
    }

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

    public static boolean isExists(String word, List<String>
wordList, int type) {
        String var = type == 0 ? "Starting" : "Target";
        if (!wordList.contains(word)) {
            throw new IllegalArgumentException(var +" word does
not exist in the word list!");
        }
        return true;
    }

    public static boolean isSameLength(String startWord, String
targetWord) {
        if (startWord.length() != targetWord.length()) {
            throw new IllegalArgumentException("Starting and
target words must be of the same length!");
        }
        return true;
    }

    public static boolean isInputValid(String word) {
        if (word == null || word.isEmpty()) {
            throw new IllegalArgumentException("Input cannot be
empty!");
        } else if (word.length() < 2) {
            throw new IllegalArgumentException("Input must be at
least 2 characters long!");
        } else if (word.length() > 8) {
            throw new IllegalArgumentException("Input must be at
most 8 characters long!");
        }
        return true;
    }
}
```

3.2. Node.java

Kelas Node berperan sebagai simpul yang menyimpan nilai *cost* hingga sampai di simpul tersebut dan nilai heuristic.

Method	Keterangan
getWord()	Mengembalikan nilai word dari node

getCost()	Mengembalikan nilai cost dari node
getTotalCost()	Mengembalikan nilai heuristic dari node
getHeuristic()	Mengembalikan nilai word dari node
createChildren(String word, List<String> wordList)	Membangkitkan simpul ekspan

```
package util;
import java.util.*;

public class Node{
    private String word;
    private int cost;
    private int heuristic;

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

    public String getWord() {
        return word;
    }

    public int getCost() {
        return cost;
    }

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

    public int getHeuristic() {
        return heuristic;
    }

    public static List<String> createChildren(String word,
List<String> wordList) {
        List<String> children = new ArrayList<>();

        for (int i = 0; i < word.length(); i++) {
            char[] chars = word.toCharArray();

            for (char c = 'a'; c <= 'z'; c++) {
                chars[i] = c;
                String child = new String(chars);

                if (!child.equals(word) &&
wordList.contains(child)) {
                    children.add(child);
                }
            }
```

```
        }

        return children;
    }
}
```

3.3. Result.java

Kelas Result merupakan return type untuk kelas-kelas algoritma karena pada java tidak bisa melakukan return multi-value.

Method	Keterangan
getPaths()	Mengembalikan nilai paths dari Result
getVisited()	Mengembalikan nilai visited dari Result

```
package util;

import java.util.List;

public class Result {
    private List<List<String>> paths;
    private int visited;

    public Result(List<List<String>> paths, int visited) {
        this.paths = paths;
        this.visited = visited;
    }

    public List<List<String>> getPaths() {
        return paths;
    }

    public int getVisited() {
        return visited;
    }
}
```

3.4. UCS.java

Kelas USC bertanggung jawab dalam melakukan pencarian dengan algoritma USC.

Method	Keterangan
searchUCS(String startWord, String targetWord, List<String> wordList)	Melakukan pencarian dengan algoritma UCS dan mengembalikan path dan jumlah visited node dalam bentuk kelas Result

```
package algorithms;
import java.util.*;
import util.*;

public class UCS {
    public static Result searchUCS(String startWord, String
targetWord, List<String> wordList) {
        if (startWord.equals(targetWord)) {
            return new
Result(Collections.singletonList(Collections.singletonList(startW
ord)), 1);
        } else {
            Map<String, List<String>> parentMap = new
HashMap<>();
            Map<String, Integer> distances = new HashMap<>();
            PriorityQueue<Node> pq = new
PriorityQueue<>(Comparator.comparingInt(Node::getCost).thenCompar
ing(Node::getWord));

            distances.put(startWord, 0);
            pq.offer(new Node(startWord, 0,
Words.countCharDifference(startWord, targetWord)));

            int visited = 0;
            while (!pq.isEmpty()) {
                Node currentNode = pq.poll();
                String currentWord = currentNode.getWord();
                int currentCost = currentNode.getCost();
                visited++;

                if (currentWord.equals(targetWord)) {
                    break;
                }

                List<String> children =
Node.createChildren(currentWord, wordList);

                for (String child : children) {
                    int newCost = currentCost + 1;
                    if (!distances.containsKey(child) || newCost
< distances.get(child)) {
                        distances.put(child, newCost);
                        parentMap.put(child, new ArrayList<>());
                        pq.offer(new Node(child, newCost,
Words.countCharDifference(child, targetWord)));
                    }

                    if (newCost == distances.get(child)) {
                        parentMap.get(child).add(currentWord);
                    }
                }
            }

            List<List<String>> paths = new ArrayList<>();
            List<String> path = new ArrayList<>();
            path.add(targetWord);
```

```
            AStar.backtrack(targetWord, startWord, parentMap,
path, paths);
            return new Result(paths, visited);
        }
    }
}
```

3.5. GreedySearch.java

Kelas GreedySearch bertanggung jawab dalam melakukan pencarian dengan algoritma Greedy Best First Search.

Method	Keterangan
searchGreedySearch(String startWord, String targetWord, List<String> wordList)	Melakukan pencarian dengan algoritma Greedy Best First Search dan mengembalikan path dan jumlah visited node dalam bentuk kelas Result

```
package algorithms;
import java.util.*;
import util.*;

public class GreedySearch {
    public static Result searchGreedySearch(String startWord,
String targetWord, List<String> wordList) {
        if (startWord.equals(targetWord)) {
            return new
Result(Collections.singletonList(Collections.singletonList(startW
ord)), 1);
        } else {
            Map<String, List<String>> parentMap = new
HashMap<>();
            Map<String, Integer> distances = new HashMap<>();
            PriorityQueue<Node> pq = new
PriorityQueue<>(Comparator.comparingInt(Node::getHeuristic).thenC
omparing(Node::getWord));

            distances.put(startWord, 0);
            pq.offer(new Node(startWord, 0,
Words.countCharDifference(startWord, targetWord)));

            int visited = 0;
            while (!pq.isEmpty()) {
                Node currentNode = pq.poll();
                String currentWord = currentNode.getWord();
                int currentCost = currentNode.getCost();
                visited++;

                if (currentWord.equals(targetWord)) {
                    break;
                }
```

```
                List<String> children =
Node.createChildren(currentWord, wordList);

                for (String child : children) {
                    int newCost = currentCost + 1;
                    if (!distances.containsKey(child) || newCost
< distances.get(child)) {
                        distances.put(child, newCost);
                        parentMap.put(child, new ArrayList<>());
                        pq.offer(new Node(child, newCost,
Words.countCharDifference(child, targetWord)));
                    }

                    if (newCost == distances.get(child)) {
                        parentMap.get(child).add(currentWord);
                    }
                }
            }

            List<List<String>> paths = new ArrayList<>();
            List<String> path = new ArrayList<>();
            path.add(targetWord);
            AStar.backtrack(targetWord, startWord, parentMap,
path, paths);
            return new Result(paths, visited);
        }
    }

}
```

3.6. Astar.java

Kelas AStar bertanggung jawab dalam melakukan pencarian dengan algoritma A*.

Method	Keterangan
searchAStar(String startWord, String targetWord, List<String> wordList)	Melakukan pencarian dengan algoritma A* dan mengembalikan path dan jumlah visited node dalam bentuk kelas Result
backtrack(String targetWord, String startWord, Map<String, List<String>> parentMap, List<String> path, List<List<String>> paths)	Melakukan path building untuk menghasilkan path yang paling optimal

```
package algorithms;
import java.util.*;
import util.*;
```

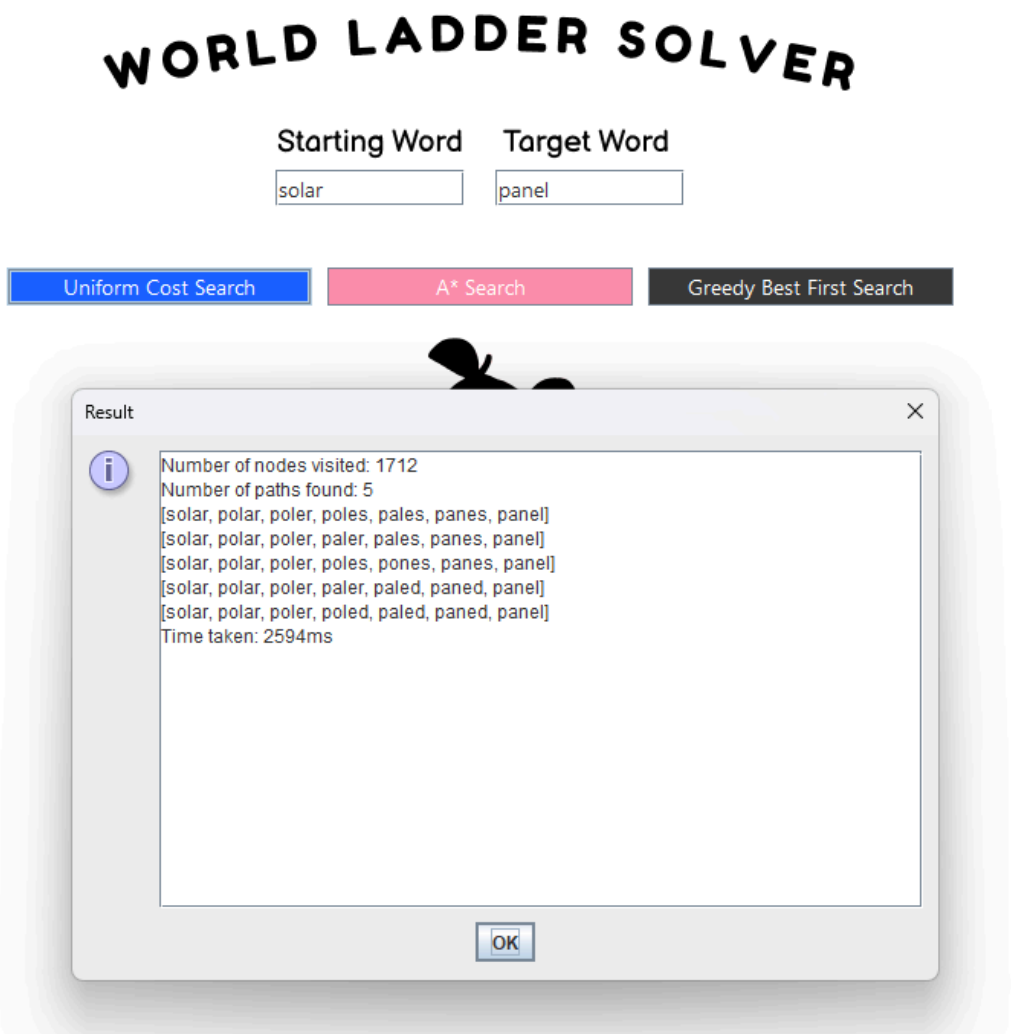
```
public class AStar {
    public static Result searchAStar(String startWord, String
targetWord, List<String> wordList) {
        if (startWord.equals(targetWord)) {
            return new
Result(Collections.singletonList(Collections.singletonList(startW
ord)), 1);
        } else {
            Map<String, List<String>> parentMap = new
HashMap<>();
            Map<String, Integer> distances = new HashMap<>();
            PriorityQueue<Node> pq = new
PriorityQueue<>(Comparator.comparingInt(Node::getTotalCost).thenC
omparing(Node::getWord));

            distances.put(startWord, 0);
            pq.offer(new Node(startWord, 0,
Words.countCharDifference(startWord, targetWord)));

            int visited = 0;
            while (!pq.isEmpty()) {
                Node currentNode = pq.poll();
                String currentWord = currentNode.getWord();
                int currentCost = currentNode.getCost();
                visited++;

                if (currentWord.equals(targetWord)) {
                    break;
                }

                List<String> children =
Node.createChildren(currentWord, wordList);

                for (String child : children) {
                    int newCost = currentCost + 1;
                    if (!distances.containsKey(child) || newCost
< distances.get(child)) {
                        distances.put(child, newCost);
                        parentMap.put(child, new ArrayList<>());
                        pq.offer(new Node(child, newCost,
Words.countCharDifference(child, targetWord)));
                    }

                    if (newCost == distances.get(child)) {
                        parentMap.get(child).add(currentWord);
                    }
                }
            }

            List<List<String>> paths = new ArrayList<>();
            List<String> path = new ArrayList<>();
            path.add(targetWord);
            backtrack(targetWord, startWord, parentMap, path,
paths);
            return new Result(paths, visited);
        }
    }
```

```
    public static void backtrack(String targetWord, String
startWord, Map<String, List<String>> parentMap, List<String>
path, List<List<String>> paths) {
        if (targetWord.equals(startWord)) {
            paths.add(new ArrayList<>(path));
            Collections.reverse(paths.get(paths.size() - 1));
            return;
        }

        if (!parentMap.containsKey(targetWord)) {
            return;
        }

        for (String parent : parentMap.get(targetWord)) {
            path.add(parent);
            backtrack(parent, startWord, parentMap, path, paths);
            path.remove(path.size() - 1);
        }
    }
}
```

BAB IV
TESTING

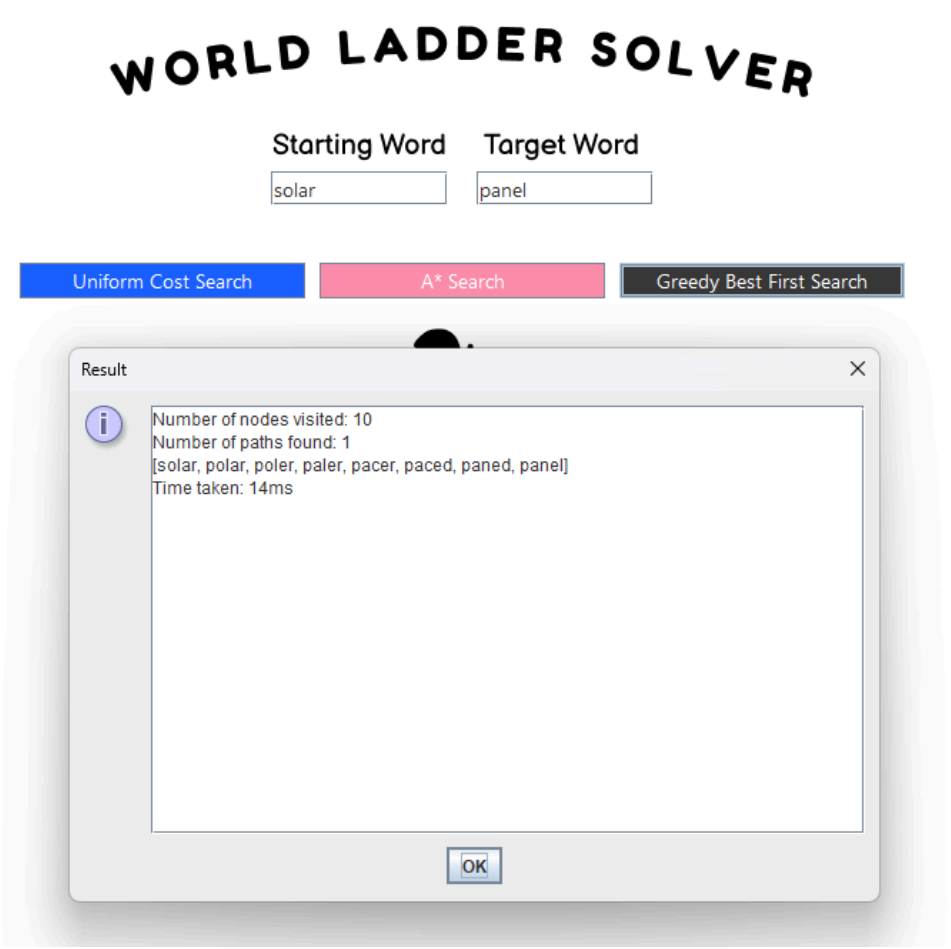
4.1. Test Case 1

1. Uniform Cost Search



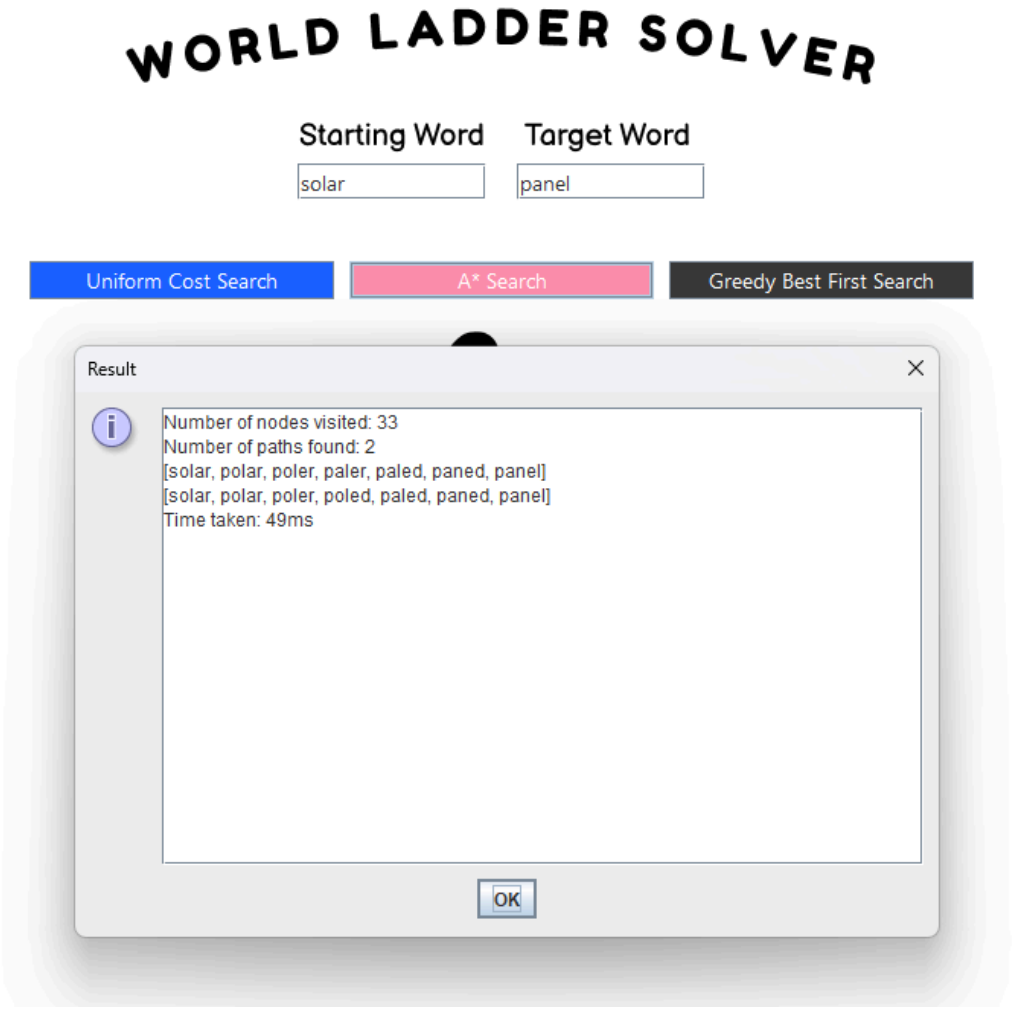
Gambar 4.1.1 Hasil Test Case Solar-Panel dengan UCS

2. Greedy Best First Search



Gambar 4.1.2 Hasil Test Case Solar-Panel dengan GBFS

3. A*



Gambar 4.1.3 Hasil Test Case Solar-Panel dengan A*

4.2. Test Case 2

1. Uniform Cost Search

Gambar 4.2.1 Hasil Test Case Pulse-Trust dengan UCS

2. Greedy Best First Search

Gambar 4.2.2 Hasil Test Case Pulse-Trust dengan GBFS

3. A*

Gambar 4.2.3 Hasil Test Case Pulse-Trust dengan A*

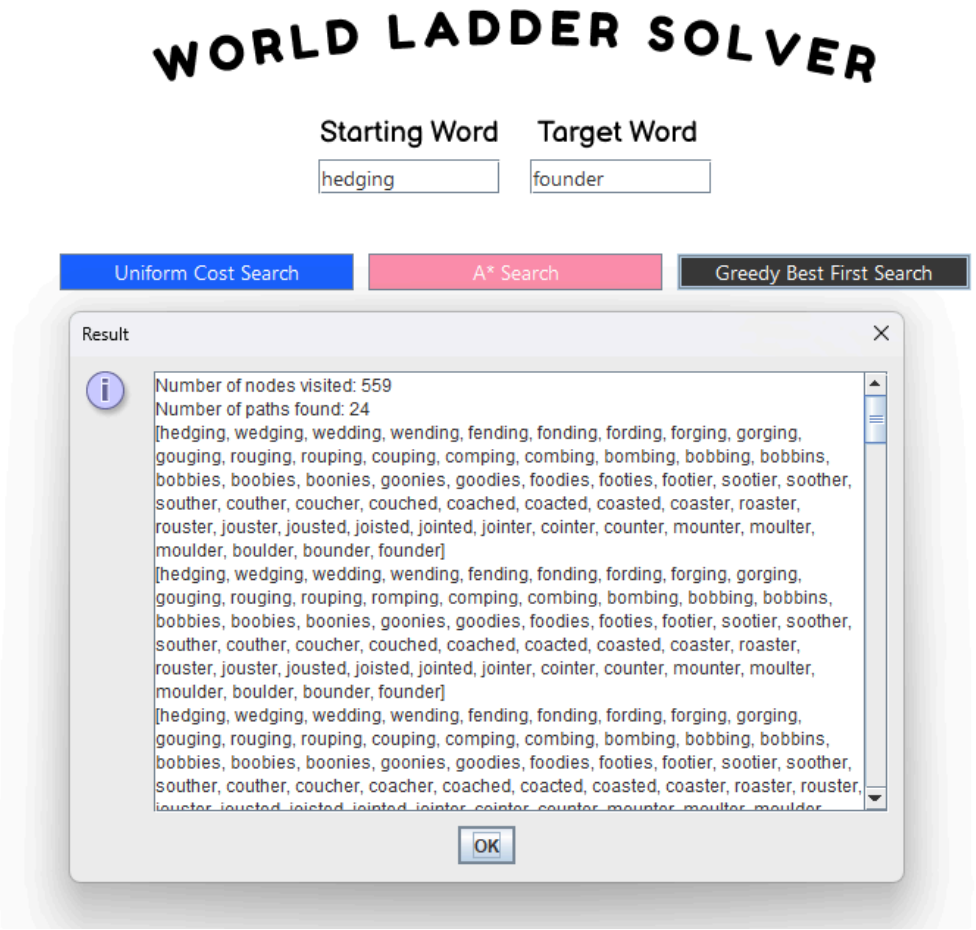
4.3. Test Case 3

1. Uniform Cost Search

Gambar 4.3.1 Hasil Test Case Hedging-Founder dengan UCS

Gambar 4.3.1.2 Hasil Test Case Hedging-Founder dengan UCS (Cont.)

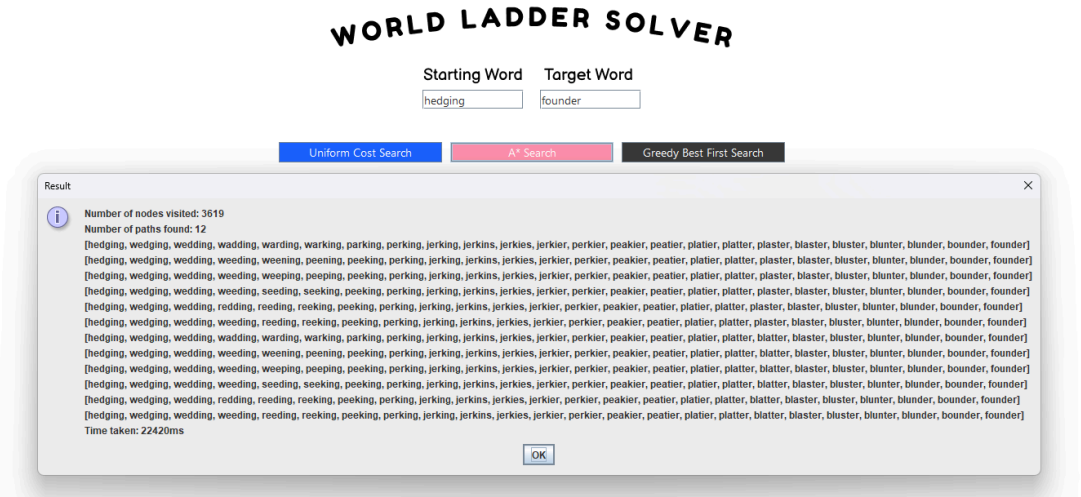
2. Greedy Best First Search



Gambar 4.3.1.1 Hasil Test Case Hedging-Founder dengan GBFS

Gambar 4.3.1.2 Hasil Test Case Hedging-Founder dengan GBFS (Cont.)

3. A*



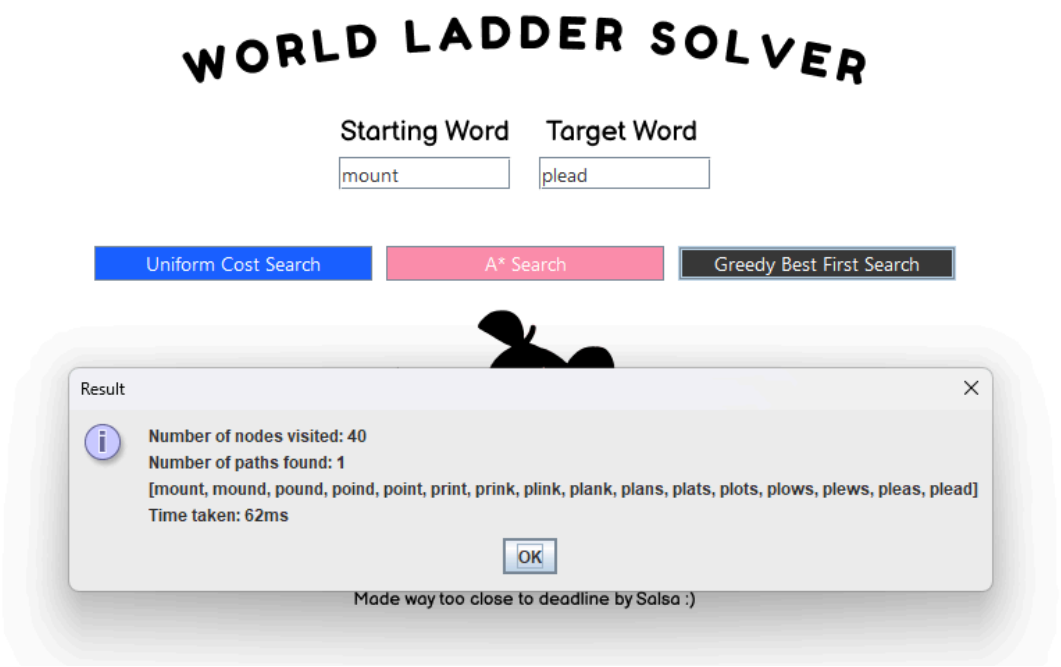
Gambar 4.3.3 Hasil Test Case Hedging-Founder dengan A*

4.4. Test Case 4

1. Uniform Cost Search

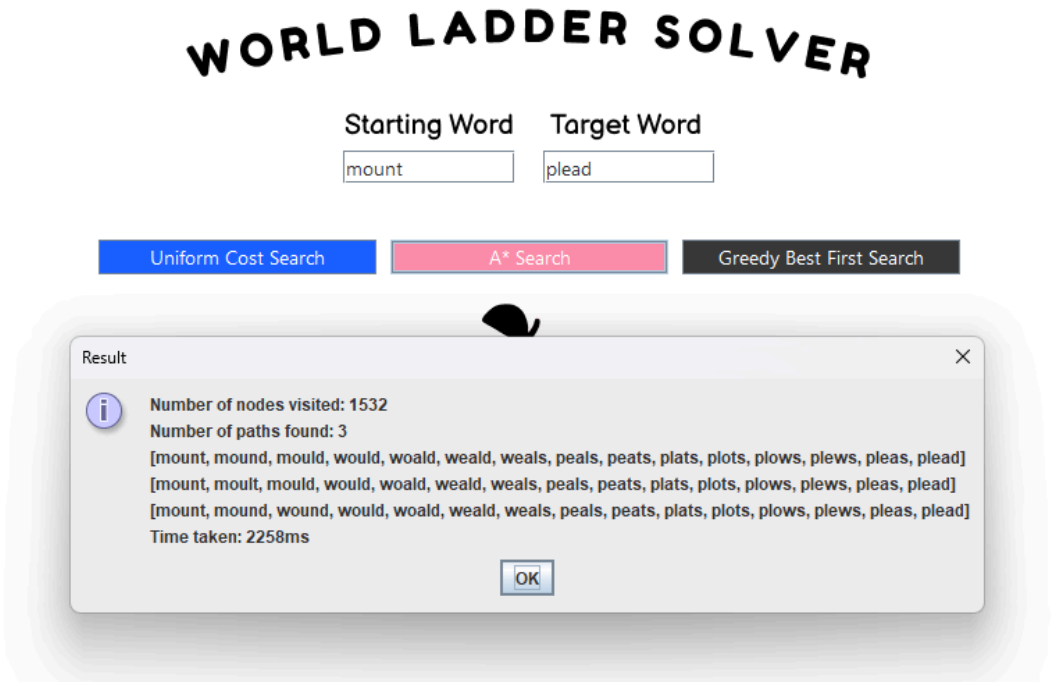
Gambar 4.4.1 Hasil Test Case Mount-Plead dengan UCS

2. Greedy Best First Search



Gambar 4.4.2 Hasil Test Case Mount-Plead dengan GBFS

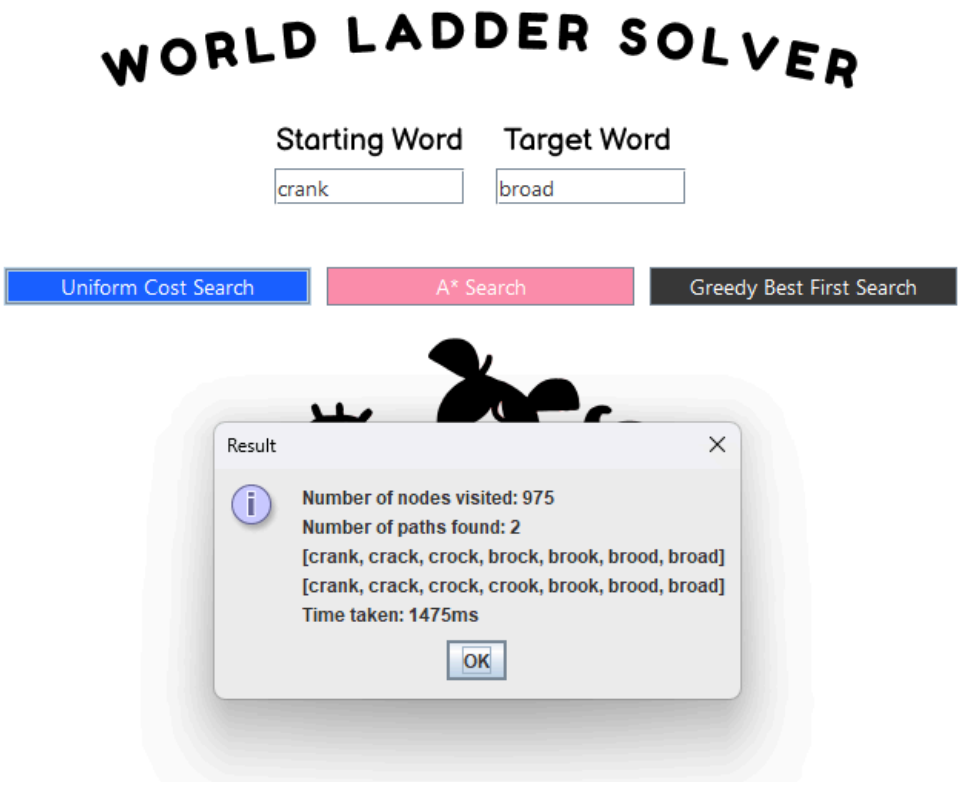
3. A*



Gambar 4.4.1 Hasil Test Case Mount-Plead dengan A*

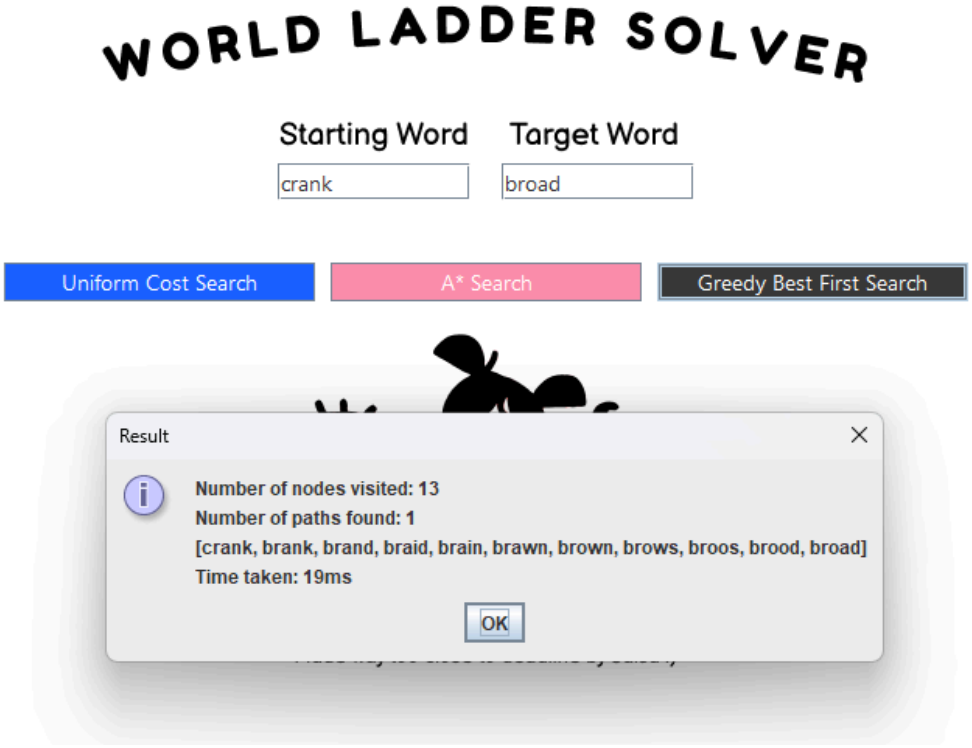
4.5. Test Case 5

1. Uniform Cost Search



Gambar 4.5.1 Hasil Test Case Crank-Broad dengan UCS

2. Greedy Best First Search



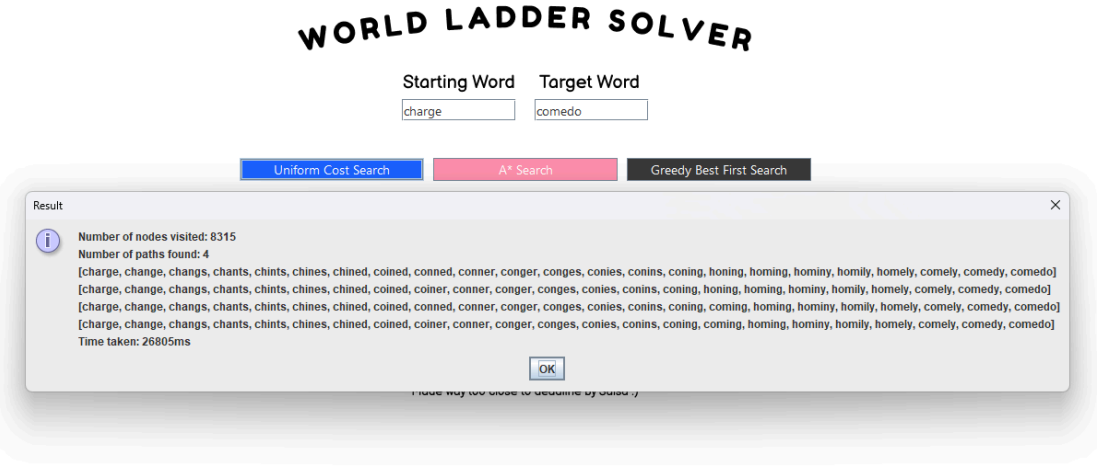
Gambar 4.5.2 Hasil Test Case Crank-Broad dengan GBFS

3. A*

Gambar 4.5.1 Hasil Test Case Crank-Broad dengan A*

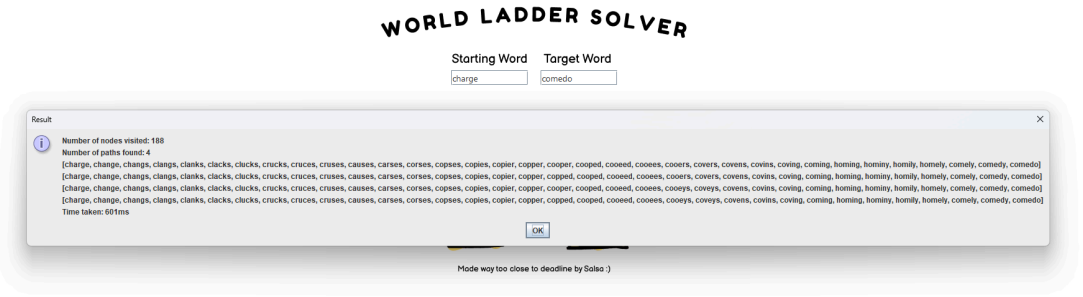
4.6. Test Case 6

1. Uniform Cost Search



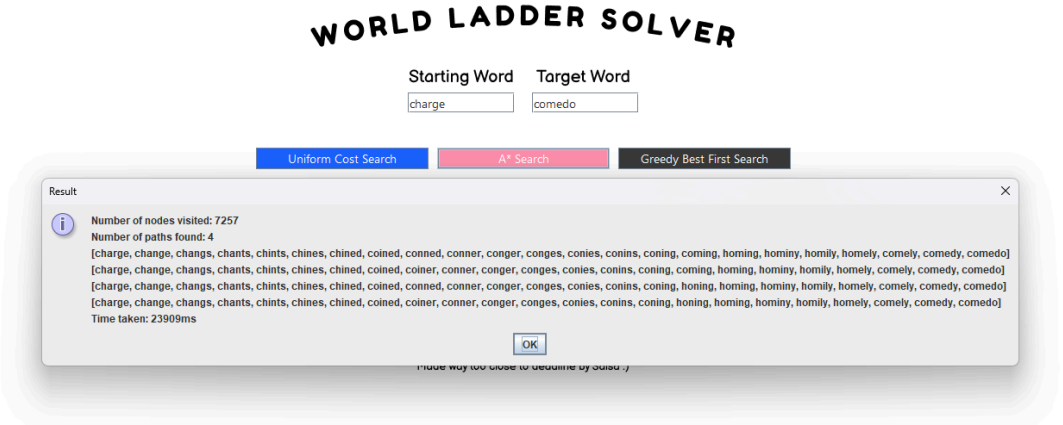
Gambar 4.6.1 Hasil Test Case Charge-Comedo dengan UCS

2. Greedy Best First Search



Gambar 4.6.1 Hasil Test Case Charge-Comedo dengan GBFS

3. A*



Gambar 4.6.1 Hasil Test Case Charge-Comedo dengan A*

4.7. Error Handling

1. Tidak Input

Gambar 4.7.1 Hasil Test Case Tidak Ada Input

2. Kata Tidak Terdapat dalam Word List

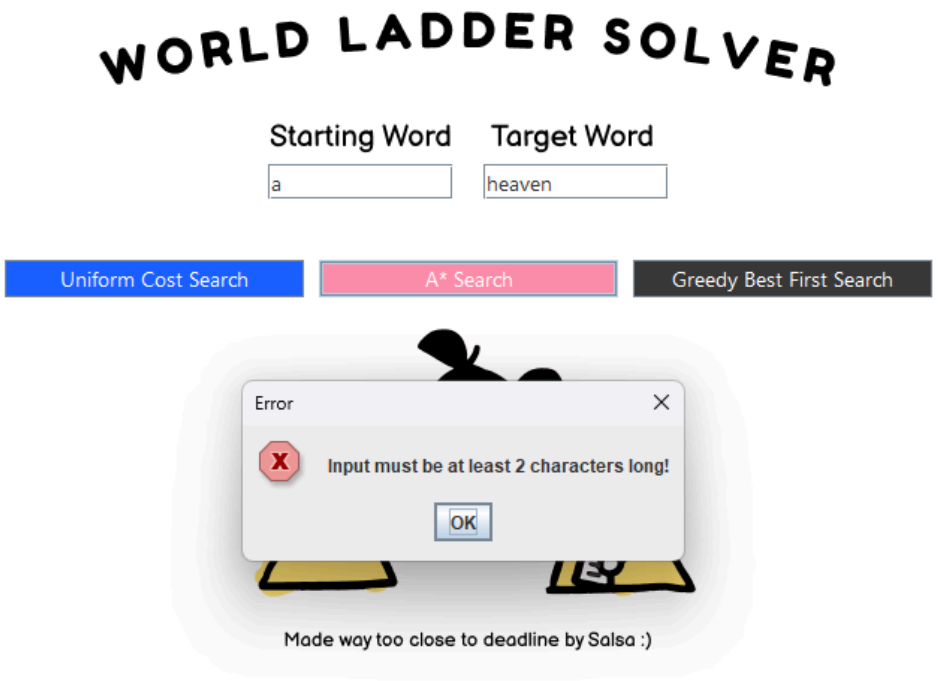
Gambar 4.7.2.1 Hasil Test Case Kata Awal Tidak Terdapat pada Dictionary

Gambar 4.7.2.2 Hasil Test Case Kata Akhir Tidak Terdapat pada Dictionary

3. Panjang Kata Tidak Sama

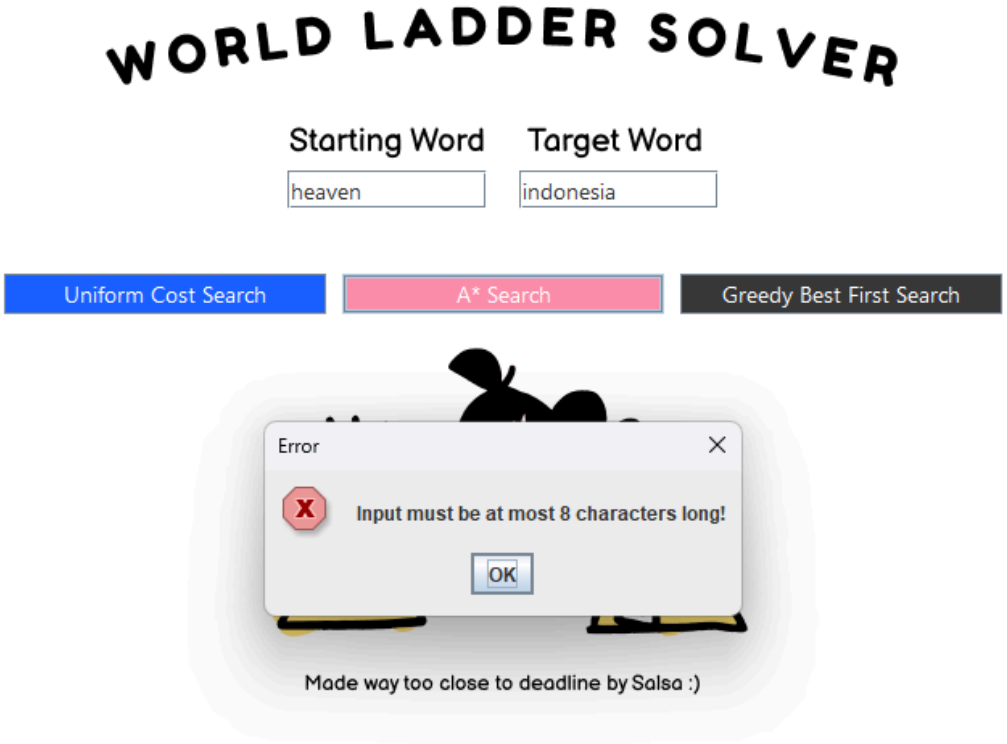
Gambar 4.7.3 Hasil Test Case Panjang Kata Berbeda

4. Panjang Kata Kurang dari Dua



Gambar 4.7.4 Hasil Test Case Kata Memiliki Panjang Kurang dari Dua

5. Panjang Kata Lebih dari Delapan



Gambar 4.7.5 Hasil Test Case Kata Memiliki Panjang Lebih dari Delapan

BAB V
ANALISIS PERBANDINGAN SOLUSI

Dari hasil uji coba yang telah dilakukan dan terlihat pada Bab III laporan ini, algoritma A* merupakan algoritma yang selalu menghasilkan solusi optimum dari penyelesaian permainan Word Ladder ini karena memperhitungkan *cost* yang dari simpul awal hingga simpul ekspan serta nilai heuristik dari simpul ekspan saat ini hingga simpul tujuan. Selain itu, fungsi heuristik yang digunakan, yaitu perbedaan jumlah karakter simpul dengan simpul tujuan, merupakan fungsi yang *admissible* karena tidak mungkin meng-*overestimate* langkah yang diambil di mana $h(n) \leq h^*(n)$ di mana $h^*(n)$ adalah jumlah langkah yang sebenarnya. Hal ini terbukti dengan langkah minimum yang harus diambil adalah nilai heuristik dari simpul awal ke simpul akhir.

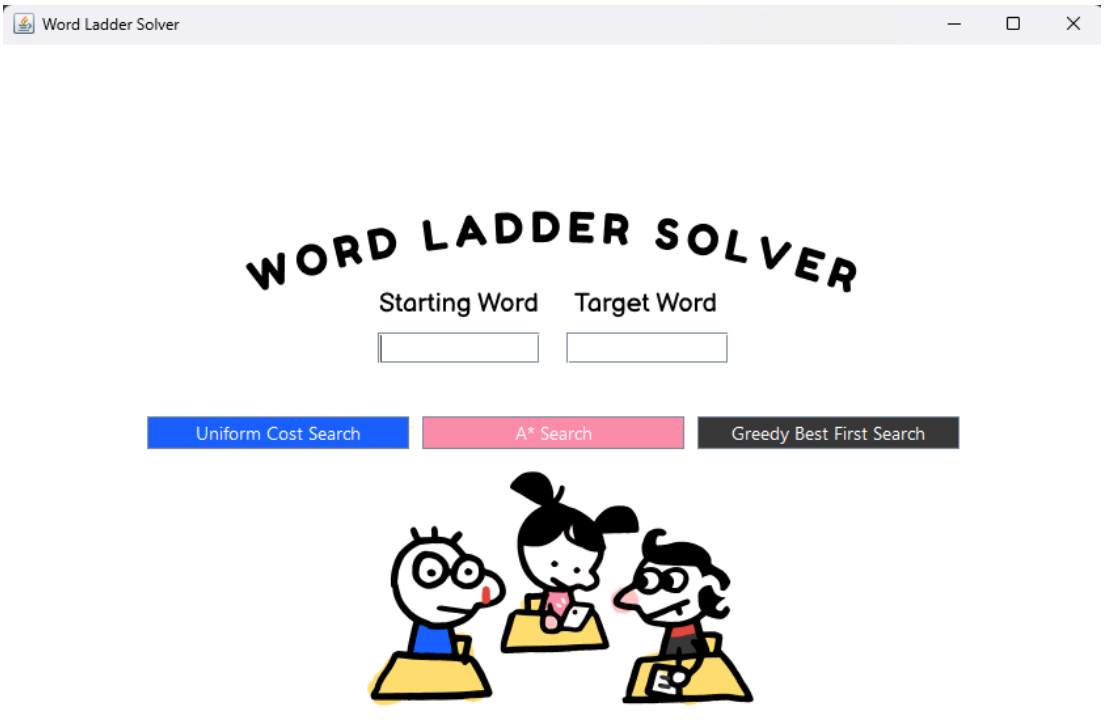
Selain dari sisi optimalitas dan *run time*, jika dilakukan perbandingan efisiensi memori, algoritma dengan memori yang paling efisien adalah algoritma Greedy Best First Search karena tidak harus menginstansiasi node untuk banyak simpul, diikuti oleh algoritma A*, dan pada posisi terakhir algoritma Uniform Cost Search.

Berdasarkan pertimbangan ketiga aspek tersebut, dapat disimpulkan bahwa algoritma terbaik untuk menyelesaikan permainan Word Ladder adalah dengan menggunakan algoritma searching A* yang merupakan penggabungan konsep Uniform Cost Search dan Greedy Best First Search dengan fungsi evaluasi $f(n) = g(n) + h(n)$ di mana $g(n)$ adalah *total cost* hingga sampai ke simpul ekspan dari simpul awal dan $h(n)$ adalah *estimated cost* atau nilai heuristik hingga mencapai simpul tujuan dari simpul ekspan saat ini.

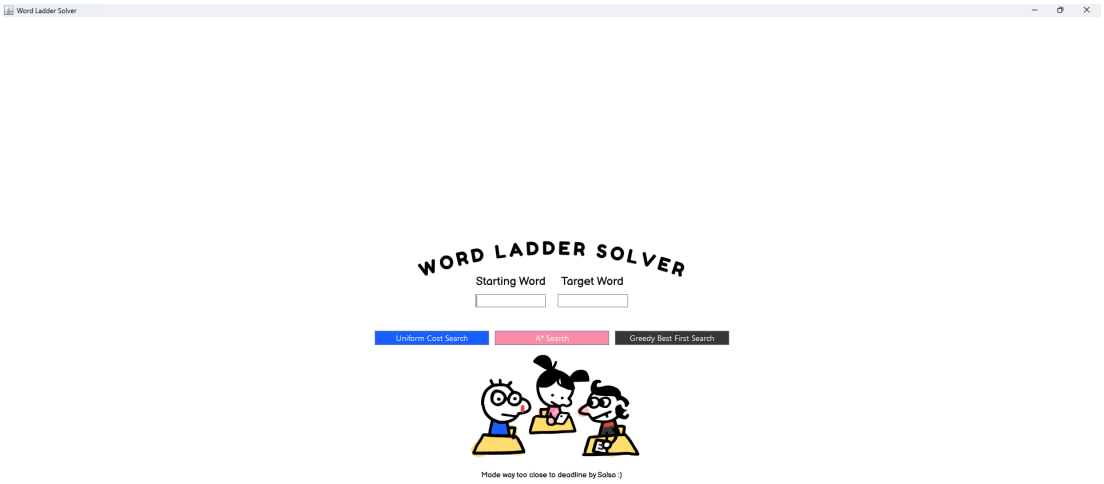
BAB VI
IMPLEMENTASI BONUS

Pembuatan GUI atau Graphical User Interface berbasis java untuk tugas kecil ini utamanya menggunakan Javax Swing dan Java AWT untuk menghandle gambar yang dijadikan *background* dari GUI. Javax Swing adalah kerangka kerja pengembangan antarmuka pengguna (UI) untuk aplikasi Java. Swing memberikan seperangkat komponen UI seperti tombol, kotak teks, daftar, dan tabel yang dapat digunakan untuk membuat aplikasi desktop yang interaktif.

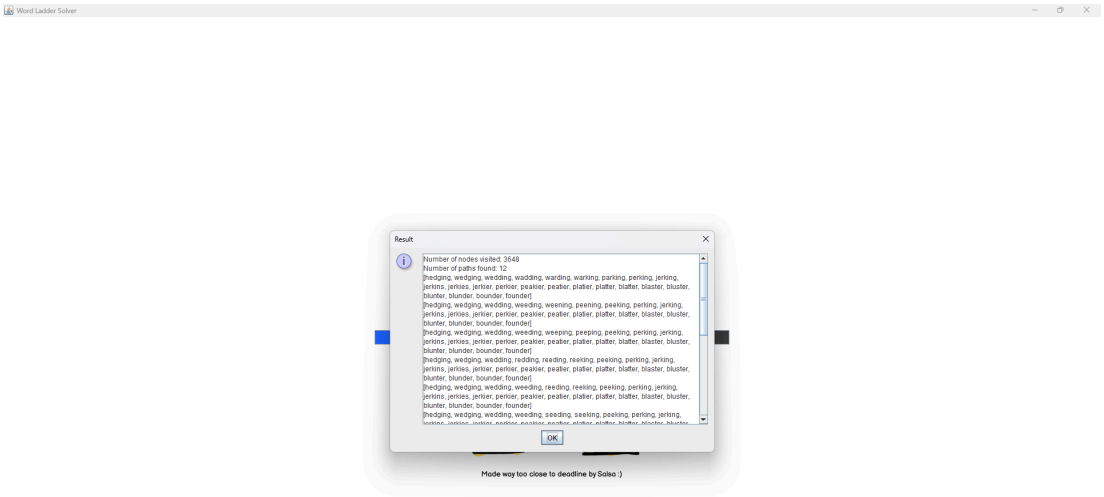
Pemilihan penggunaan Swing dibandingkan JavaFX dikarenakan penggunaanya yang tidak melakukan instalasi apapun sehingga semua orang yang telah melakukan instalasi java dapat menjalankannya. Dalam implementasinya, komponen yang digunakan adalah textField, button, dan panel untuk memastikan semua komponen tetap *centered*. Output dari hasil pencarian ditampilkan dengan menggunakan optionPane serta untuk menangani output text yang panjang, digunakan textArea sehingga teks output lebih mudah untuk dibaca.



Gambar 6.1 Tampilan pada Mode Windowed



Gambar 6.2 Tampilan Mode Full Screen



Gambar 6.3 Tampilan Result

Berikut merupakan kode pemrograman untuk GUI berbasis java pada tugas kecil ini.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.time.*;
import java.util.List;
```

```
import javax.imageio.ImageIO;
import java.io.IOException;
import java.awt.image.BufferedImage;

import util.*;
import algorithms.*;

public class MainGUI {
    private static Map<Integer, List<String>> wordMap;
    private static JTextField startWordField, targetWordField;
    private static JFrame frame;
    private static final Dimension BUTTON_SIZE = new Dimension(200,
25);
    private static BufferedImage backgroundImage;

    public static void main(String[] args) {
        wordMap = Words.createWordMap("util/words.txt");

        try {
            backgroundImage =
ImageIO.read(MainGUI.class.getResourceAsStream("Frame 2.png")); //
Load the image
        } catch (IOException ex) {
            ex.printStackTrace();
        }

        SwingUtilities.invokeLater(() -> {
            createAndShowGUI();
        });
    }

    private static void createAndShowGUI() {
        frame = new JFrame("Word Ladder Solver");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel mainPanel = new JPanel(new BorderLayout()) {
            @Override
            protected void paintComponent(Graphics g) {
                super.paintComponent(g);
                if (backgroundImage != null) {
                    // Draw the background image, centered in the
panel
                    int x = (getWidth() -
backgroundImage.getWidth()) / 2;
                    int y = (getHeight() -
backgroundImage.getHeight()) / 2;
                    g.drawImage(backgroundImage, x, y, this);
                }
            }
        };
        frame.add(mainPanel);

        // Add margin insets to the main panel
        mainPanel.setBorder(BorderFactory.createEmptyBorder(200,
100, 200, 100));

        JPanel centerPanel = new JPanel(new GridBagLayout());
        centerPanel.setOpaque(false); // Make center panel
```

```
transparent
        GridBagConstraints gbc = new GridBagConstraints();
        gbc.gridx = 0;
        gbc.gridy = 0;
        gbc.insets = new Insets(10, 0, 10, 0); // Adjust insets here
to decrease distance

        JPanel inputPanel = new JPanel();
        inputPanel.setOpaque(false); // Make input panel transparent
        inputPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 0,
10));

        startWordField = new JTextField(10);
        startWordField.setFont(new Font("Gadugi", Font.PLAIN, 14));
        inputPanel.add(startWordField);

        inputPanel.add(Box.createHorizontalStrut(20));
        targetWordField = new JTextField(10);
        targetWordField.setFont(new Font("Gadugi", Font.PLAIN, 14));
        inputPanel.add(targetWordField);

        centerPanel.add(inputPanel, gbc);

        JPanel buttonPanel = new JPanel();
        buttonPanel.setOpaque(false); // Make button panel
transparent
        buttonPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 10,
10));

        JButton ucsButton = new JButton("Uniform Cost Search");
        configureButton(ucsButton);
        ucsButton.setBackground(new Color(29, 99, 255));
        ucsButton.setForeground(Color.WHITE);
        ucsButton.setFont(new Font("Gadugi", Font.PLAIN, 14));
        ucsButton.addActionListener(new UCSListener());
        buttonPanel.add(ucsButton);

        JButton aStarButton = new JButton("A* Search");
        configureButton(aStarButton);
        aStarButton.setBackground(new Color(254, 143, 172));
        aStarButton.setForeground(Color.WHITE);
        aStarButton.setFont(new Font("Gadugi", Font.PLAIN, 14));
        aStarButton.addActionListener(new AStarListener());
        buttonPanel.add(aStarButton);

        JButton gbfsButton = new JButton("Greedy Best First
Search");
        configureButton(gbfsButton);
        gbfsButton.setBackground(new Color(58, 58, 58));
        gbfsButton.setForeground(Color.WHITE);
        gbfsButton.setFont(new Font("Gadugi", Font.PLAIN, 14));
        gbfsButton.addActionListener(new GreedyListener());
        buttonPanel.add(gbfsButton);

        gbc.gridy = 1;
        centerPanel.add(buttonPanel, gbc);

        mainPanel.add(centerPanel, BorderLayout.CENTER);
```

```
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }

    private static void configureButton(JButton button) {
        button.setPreferredSize(BUTTON_SIZE);
        button.setMaximumSize(BUTTON_SIZE);
        button.setMinimumSize(BUTTON_SIZE);
    }

    private static class UCSListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String startWord =
startWordField.getText().toLowerCase();
            String targetWord =
targetWordField.getText().toLowerCase();

            try {
                Words.isInputValid(startWord);
                Words.isInputValid(targetWord);
                Words.isExists(startWord,
wordMap.get(startWord.length()), 0);
                Words.isExists(targetWord,
wordMap.get(targetWord.length()), 1);
                Words.isSameLength(startWord, targetWord);
                long startTime = System.nanoTime();
                Result res = UCS.searchUCS(startWord, targetWord,
wordMap.get(startWord.length()));
                long endTime = System.nanoTime();
                displayResult(res, startTime, endTime);
            } catch (IllegalArgumentException ex) {
                JOptionPane.showMessageDialog(frame,
ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
            }
        }
    }

    private static class GreedyListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String startWord =
startWordField.getText().toLowerCase();
            String targetWord =
targetWordField.getText().toLowerCase();

            try {
                Words.isInputValid(startWord);
                Words.isInputValid(targetWord);
                Words.isExists(startWord,
wordMap.get(startWord.length()), 0);
                Words.isExists(targetWord,
wordMap.get(targetWord.length()), 1);
                Words.isSameLength(startWord, targetWord);
                long startTime = System.nanoTime();
                Result res =
GreedySearch.searchGreedySearch(startWord, targetWord,
wordMap.get(startWord.length()));
```

```
                long endTime = System.nanoTime();
                displayResult(res, startTime, endTime);
            } catch (IllegalArgumentException ex) {
                JOptionPane.showMessageDialog(frame,
ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
            }
        }
    }

    private static class AStarListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String startWord =
startWordField.getText().toLowerCase();
            String targetWord =
targetWordField.getText().toLowerCase();

            try {
                Words.isInputValid(startWord);
                Words.isInputValid(targetWord);
                Words.isExists(startWord,
wordMap.get(startWord.length()), 0);
                Words.isExists(targetWord,
wordMap.get(targetWord.length()), 1);
                Words.isSameLength(startWord, targetWord);
                long startTime = System.nanoTime();
                Result res = AStar.searchAStar(startWord,
targetWord, wordMap.get(startWord.length()));
                long endTime = System.nanoTime();
                displayResult(res, startTime, endTime);
            } catch (IllegalArgumentException ex) {
                JOptionPane.showMessageDialog(frame,
ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
            }
        }
    }

    private static void displayResult(Result res, long startTime,
long endTime) {
        List<List<String>> paths = res.getPaths();
        if (paths.isEmpty()) {
            JOptionPane.showMessageDialog(frame, "No path found.",
"Result", JOptionPane.INFORMATION_MESSAGE);
        } else {
            StringBuilder result = new StringBuilder();
            result.append("Number of nodes visited:
").append(res.getVisited()).append("\n");
            result.append("Number of paths found:
").append(paths.size()).append("\n");
            for (List<String> path : paths) {
                result.append(path).append("\n");
            }
            result.append("Time taken:
").append(Duration.ofNanos(endTime -
startTime).toMillis()).append("ms");

            // Create a JTextArea to hold the result
            JTextArea textArea = new JTextArea(result.toString());
            textArea.setEditable(false);
```

```
            textArea.setLineWrap(true);
            textArea.setWrapStyleWord(true);

            // Wrap the JTextArea in a JScrollPane to enable
scrolling
            JScrollPane scrollPane = new JScrollPane(textArea);
            scrollPane.setPreferredSize(new Dimension(500, 300)); //
Set the maximum size here

            // Show the JScrollPane in the option pane
            JOptionPane.showMessageDialog(frame, scrollPane,
"Result", JOptionPane.INFORMATION_MESSAGE);
        }
    }
}
```

LAMPIRAN

Link repository : https://github.com/salsbiila/Tucil3_13522062

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

DAFTAR PUSTAKA

Munir, Rinaldi. 2024. “Penentuan rute (Route/Path Planning) - Bagian 1”(online). (https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf, diakses 4 Mei 2024).

Munir, Rinaldi. 2024. “Penentuan rute (Route/Path Planning) - Bagian 2”(online). (https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf, diakses 4 Mei 2024).