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

Disusun oleh:
Marvel Pangondian (13522075)

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung

2024

Daftar Isi

Bab I
Deskripsi Masalah

1.1 Deskripsi Masalah

Word ladder adalah permainan kata yang populer di berbagai kalangan. Dalam *Word Ladder*, pemain diberikan dua kata: kata awal (*start word*) dan kata akhir (*end word*). Tujuan dari permainan ini adalah untuk menghubungkan kedua kata tersebut melalui serangkaian kata perantara. Setiap kata dalam rangkaian ini harus memiliki jumlah huruf yang sama dengan kata awal dan akhir, dan hanya satu huruf yang boleh berbeda antara dua kata berturut-turut dalam rangkaian tersebut. Tujuan dari permainan ini adalah untuk menemukan solusi yang paling efisien, yaitu dengan menggunakan jumlah kata perantara yang paling sedikit.

Dalam Tugas Kecil ini, penulis diinginkan untuk membuat program yang dapat mencari solusi dalam permainan *Word Ladder* menggunakan algoritma UCS, *Greedy Best First Search,* serta A*. Penulis juga harus menganalisis ketiga algoritma tersebut berdasarkan optimalitas, waktu eksekusi, serta memori yang dibutuhkan.

1.2 Spesifikasi Program

Terdapat beberapa spesifikasi yang harus dipenuhi oleh program, antara lain sebagai berikut :

1. Program dalam bahasa Java berbasis CLI (Command Line Interface) – bonus jika menggunakan GUI
2. Program dapat menemukan solusi permainan *Word Ladder* menggunakan algoritma UCS, *Greedy Best First Search*, dan A*.
3. Kata-kata yang dapat dimasukkan harus berbahasa Inggris.
4. Program harus bisa menangani berbagai panjang kata.
5. Program harus bisa menampilkan satu *Path* antara *Start Word* dan *End Word*.
6. Program harus bisa menampilkan banyaknya node yang dikunjungi.
7. Program harus bisa menampilkan waktu eksekusi program.

Bab II

Algoritma UCS, *Greedy Best First Search*, dan A*

2.1 Analisis Algoritma UCS dalam Penyelesaian Permainan Word Ladder

Algoritma UCS (*Uniform Cost Search*) adalah algoritma mencari rute terpendek antara simpul awal (*Start Node*) dengan simpul akhir (*End Node*). Algoritma ini merupakan salah satu pendekatan pencarian tanpa informasi yang fokus pada pengeksplorasian jalur berdasarkan biaya kumulatif terendah hingga mencapai tujuan, tanpa mempertimbangkan estimasi jarak yang tersisa ke tujuan (*uninformed Search*).

Dalam permainan *Word Ladder*, UCS menggunakan g(n) yakni "Jarak dari sebuah simpul (dalam hal ini sebuah kata) dengan kata awal (*Start Node*)". "Jarak" dalam hal ini adalah berapa lompatan dari kata awal ke kata saat itu. Contoh : *cold -> wold -> word.* Dalam contoh tersebut, *cold* memiliki nilai g(n) = 0 sebab kata tersebut merupakan kata awal, *wold* memiliki nilai g(n) = 1 sebab dia berjarak 1 lompatan dari *cold*, dan akhirnya *word* memiliki nilai g(n) = 2. Algoritma ini menggunakan struktur data *Priority Queue* untuk memasukan simpul - simpul, dalam hal ini kata - kata, berdasarkan g(n). Algoritma UCS akan melakukan pencarian dari simpul awal (*Start Word*) kemudian mengeksplorasi simpul-simpul yang bertetangga dengan simpul awal tersebut. Untuk setiap simpul yang dijelajahi, algoritma ini akan menghitung biaya jalur menggunakan g(n), setiap anak simpul tetangga yang dibangkitkan akan dihitung nilai g(n) kemudian dimasukkan kedalam *Priority Queue*, Lalu algoritma akan mengambil simpul berikutnya dengan g(n) terkecil pada *Priority Queue,* proses ini dilanjutkan sampai ditemukan simpul akhir.

Dikarenakan definisi g(n) yang sama dengan definisi *depth*/kedalaman dalam *Breadth First Search* (BFS) maka dapat dikatakan bahwa dalam penyelesaian permainan *Word Ladder,* algoritma UCS dan BFS sama, yakni urutan node yang dibangkitkan dan path yang dihasilkan sama. Walaupun dalam BFS tidak digunakannya *Priority Queue*, yang digunakan hanyalah *Queue* dimana simpul - simpul tetangga dimasukkan ke akhiran *Queue* dengan sifat *First In First Out* (FIFO*)*, tetapi dikarenakan definisi g(n), maka simpul - simpul tetangga dalam algoritma UCS selalu diletakan di belakang

Priority Queue sehingga *Priority Queue* dalam UCS bersifat sama dengan *Queue* dalam BFS.

2.2 Analisis Algoritma *Greedy Best First Search* dalam Penyelesaian Permainan Word Ladder

Algoritma *Greedy Best First Search* adalah algoritma pencarian rute terpendek antara simpul awal (*Start Node*) dengan simpul akhir (*End Node*) dengan memilih simpul terbaik/paling menjanjikan pada setiap langkah. Algoritma ini termasuk dalam kategori *informed search* dan heuristik sebab sudah tersedia informasi mengenai *goal state* dan cara mencapainya.Informasi tersebut akan digunakan untuk memilih simpul terbaik setiap langkahnya. Sifat heuristik tersebut dapat memungkinkan pencarian rute dengan cepat, tetapi terkadang tidak optimal. Algoritma ini juga bersifat *Irrevocable* yang berarti tidak dapat diubah/*reversed*, sifat tersebut memungkinkan situasi dimana saat proses pencarian terjebak pada *local minima/plateau*.

Dalam permainan *Word Ladder*, *Greedy Best First Search* menggunakan h(n) (*heuristic function*) untuk menentukan simpul terbaik yang dipilih setiap langkah, simpul itu sendiri adalah kata - kata, dan h(n) adalah "Banyaknya huruf-huruf yang berbeda antara kata saat itu dengan kata tujuan". Algoritma dimulai dari simpul awal yakni *Start Word*, dari simpul tersebut akan diekspansi simpul - simpul yang bertetangga dengan simpul saat itu, dari antara simpul - simpul yang diekspansi, dipilihnya simpul dengan nilai h(n) terendah sebagai simpul yang diekspansi selanjutnya, proses tersebut diulang sampai menemukan simpul akhir (*End Word*).

Secara Teoritis, dikarenakan sifatnya yang *pure heuristic* serta ketidakmampuan untuk *backtrack* (*Irrevocable*), maka algoritma *Greedy Best First Search* tidak selalu memberikan solusi yang optimal. Dalam penyelesaian permainan *Word Ladder*, terdapat banyak situasi di mana algoritma memilih kata yang, meskipun pada saat itu tampak sebagai kata yang paling dekat dengan kata tujuan, tetapi pilihan tersebut akhirnya mengarah pada solusi yang kurang optimal dalam rangkaian kata.

2.3 Analisis Algoritma A* dalam Penyelesaian Permainan Word Ladder

Algoritma A* adalah algoritma pencarian rute terpendek antara simpul awal (Start Node) dengan simpul akhir (End Node). Algoritma ini termasuk dalam kategori *informed search* yang menggunakan f(n) untuk menentukan biaya sebuah node. Dalam algoritma ini, f(n) didefinisikan sebagai fungsi yang gabungan antara g(n) dan h(n) dengan rumus f(n) = g(n) + h(n). Algoritma ini mengkombinasikan sifat heuristik *Greedy Best First Search* dengan pencarian rute optimal UCS untuk mengembangkan algoritma yang lebih cepat dan efisien

Dalam penyelesaian permainan *Word Ladder*, A* menggunakan f(n) yang didefinisikan sebagai "Jarak sebuah simpul (kata) dengan simpul awal ditambah dengan banyaknya huruf yang berbeda antara simpul dengan simpul akhir" (f(n) = g(n) + h(n)). Sama seperti UCS, algoritma ini menggunakan struktur data *Priority Queue* untuk menyimpan simpul - simpul yang sudah dibangkitkan. Algoritma ini dimulai dengan simpul awal (*Start Word*) yang kemudian dibangkitkan simpul - simpul tetangganya, setiap simpul tersebut dihitung nilai f(n) dan dimasukkan ke dalam *Priority Queue.* Kemudian simpul berikutnya dengan nilai f(n) terkecil diambil dari *Priority Queue* dan kemudian diproses sampai menemukan simpul akhir (*End Word*). Ketika menemukan simpul akhir, dicek terlebih dahulu apakah terdapat simpul - simpul yang memiliki nilai f(n) lebih kecil dibandingkan dengan simpul akhir tersebut, jika iya maka proses dilanjutkan sampai tidak ada simpul yang memiliki nilai f(n) lebih kecil dibandingkan simpul akhir. Jika terdapat simpul akhir baru yang memiliki nilai f(n) lebih kecil dibandingkan dengan simpul akhir lama, maka simpul akhir lama akan diganti dengan simpul akhir baru.

Heuristik (h(n)) yang digunakan oleh algoritma ini selalu *underestimate* jarak sebenarnya dari sebuah kata ke kata akhir (h*(n)) dikarenakan heuristik selalu memberikan jarak yang bersifat *best case scenario,* akibatnya dalam implementasi algoritma A* ini selalu berlakunya $h(n) \leq h^*(n)$ sehingga heuristic ini *admissible*. Karena heuristic *admissible,* algoritma A* sama dengan UCS, selalu memberikan rute optimal dari kata awal (*Start Word*) ke kata akhir (*End Word*).

Secara Teoritis, algoritma A* dengan heuristik yang efektif dan *admissible* merupakan algoritma yang lebih efisien dibandingkan dengan algoritma UCS, hal ini dikarenakan fungsi heuristik yang dapat membantu algoritma A* mencapai *End Word*

lebih cepat dibandingkan dengan UCS. Heuristik tersebut juga membantu algoritma A* untuk mengurangi simpul - simpul yang ditelusuri untuk mencapai simpul akhir sehingga mengurangi memori yang dipakai dan waktu eksekusi sehingga lebih efisien dibandingkan dengan UCS.

2.4 Implementasi Algoritma

2.4.1. Implementasi Struktur Data

Struktur data yang akan digunakan oleh setiap algoritma adalah struktur data Node (merepresentasikan kata) dan *Set* bernama *Visited* (mencatat kata - kata yang sebelumnya sudah dilewati). Algoritma UCS dan A* juga menggunakan data struktur tambahan yakni *Priority Queue* sebagai *container* simpul - simpul (Node) yang sudah dibangkitkan.

2.4.2. Implementasi Algoritma UCS

Algoritma UCS menggunakan data struktur Node sebagai representasi setiap kata yang dibangkitkan, *Priorty Queue* sebagai container simpul - simpul, serta *Set Visited* untuk mencatat kata - kata yang sudah pernah ditelusuri sebelumnya. Berikut adalah langkah - langkah penyelesaian permainan *Word Ladder* menggunakan algoritma UCS :

1. Pertama, dilakukannya validasi *Start Word* dan *End Word* berupa validasi keberadaannya dalam kamus Inggris serta validasi ukuran kedua kata tersebut.
2. Setelah proses validasi, akan dibentuknya struktur data *Priority Queue* dengan Node awal (*Start Word*) sebagai elemen pertama dan struktur data *Set Visited* untuk mencatat node - node yang sudah dikunjungi sebelumnya.
3. Setelah itu, algoritma dimulai dengan membangkitkan elemen pertama pada *Priority Queue* (dequeue).
4. Node yang dibangkitkan dicek apakah sudah pernah ditelusuri sebelumnya, jika sudah maka diabaikan, jika belum maka proses berlanjut.

5. Node yang dibangkitkan tersebut dicek apakah merupakan node tujuan (*End Word*) atau bukan, jika iya maka proses diberhentikan dan segera membentuk rangkaian kata, jika tidak maka proses dilanjutkan.
6. Proses berikutnya adalah mengekspansi Node tersebut dengan mengumpulkan node - node tetangga. Node - node tetangga adalah node dengan jumlah huruf yang sama dengan node tersebut dan memiliki satu huruf yang berbeda.
7. Sebelum dimasukkan ke dalam *Priority Queue,* dihitungnya biaya g(n) yakni jaraknya node tersebut dengan node awal (*Start Word*). Pemasukan node ke dalam *Priority Queue* akan berdasarkan nilai g(n) masing - masing node (diurutkan dari terkecil hingga terbesar).
8. Setelah selesai ekspansi, node yang dibangkitkan kemudian di tandai sudah ditelusuri menggunakan *Set Visited.*
9. Proses 3 - 8 dilakukan berulang sampai ditemukan node akhir (*End Word*) atau *Priority Queue* sudah kosong.

2.4.3. Implementasi Algoritma *Greedy Best First Search*

Algoritma *Greedy Best First Search* menggunakan data struktur Node sebagai representasi setiap kata yang dibangkitkan, serta *Set Visited* untuk mencatat kata - kata yang sudah pernah ditelusuri sebelumnya. Berikut adalah langkah - langkah penyelesaian permainan *Word Ladder* menggunakan algoritma *Greedy Best First Search* :

1. Pertama, dilakukannya validasi *Start Word* dan *End Word* berupa validasi keberadaannya dalam kamus Inggris serta validasi ukuran kedua kata tersebut.
2. Setelah itu, algoritma dimulai dengan node awal (*Start Word*) sebagai *currNode* (node pada sebuah langkah tertentu).
3. Pertama, *currNode* akan diperiksa apakah merupakan node akhir (*End Word*), jika iya maka proses akan diberhentikan, jika tidak maka proses akan dilanjutkan.
4. Proses berikutnya adalah mengekspansi *currNode* dengan mengumpulkan node - node tetangga. Node - node tetangga adalah

node dengan jumlah huruf yang sama dengan *currNode* dan memiliki satu huruf yang berbeda.

5. Setelah itu, dilakukannya iterasi pada seluruh node - node tetangga. Sebelum dihitung nilai heuristicnya, tiap node tetangga akan dicek terlebih dahulu menggunakan *Set Visited* untuk menghindari node yang pernah ditelusuri sebelumnya. Tiap node tetangga akan dihitung nilai h(n) yakni jumlah huruf yang berbeda dengan kata akhir. Lalu *CurrNode* akan diubah menjadi node tetangga yang sudah dipilih. Node tetangga yang dipilih adalah node dengan nilai h(n) terendah.
6. Proses 2 - 5 akan diulangi sampai ditemukan node terakhir atau sampai terjebak di *local minima/ plateau.*

2.4.4. Implementasi Algoritma A*

Algoritma A* menggunakan data struktur Node sebagai representasi setiap kata yang dibangkitkan, *Priorty Queue* sebagai container simpul - simpul, serta Set Visited untuk mencatat kata - kata yang sudah pernah ditelusuri sebelumnya. Berikut adalah langkah - langkah penyelesaian permainan *Word Ladder* menggunakan algoritma A* :

1. Pertama, dilakukannya validasi *Start Word* dan *End Word* berupa validasi keberadaannya dalam kamus Inggris serta validasi ukuran kedua kata tersebut.
2. Setelah proses validasi, akan dibentuknya struktur data *Priority Queue* dengan Node awal (*Start Word*) sebagai elemen pertama dan struktur data *Set Visited* untuk mencatat node - node yang sudah dikunjungi sebelumnya. A* juga memiliki node yakni *lastNode* untuk mencatat solusi node terakhir, node ini diinisialisasi sebagai node kosong.
3. Setelah itu, algoritma dimulai dengan membangkitkan elemen pertama pada *Priority Queue* (dequeue).
4. Node yang dibangkitkan dicek apakah sudah pernah ditelusuri sebelumnya, jika sudah maka diabaikan, jika belum maka proses berlanjut.
5. Node yang dibangkitkan tersebut dicek apakah merupakan node tujuan (*End Word*) atau bukan, jika iya maka node akan dimasukkan ke dalam

variabel *lastNode* dan lanjut memproses node berikutnya di *Priority Queue*, jika tidak maka lanjut proses berikutnya.

6. Proses berikutnya adalah mengekspansi Node tersebut dengan mengumpulkan node - node tetangga. Node - node tetangga adalah node dengan jumlah huruf yang sama dengan node tersebut dan memiliki satu huruf yang berbeda.
7. Sebelum dimasukkan ke dalam *Priority Queue,* dihitungnya biaya f(n) yakni jaraknya node tersebut dengan node awal (*Start Word*) ditambah dengan jumlah huruf yang berbeda dengan kata akhir. Pemasukan node ke dalam *Priority Queue* akan berdasarkan nilai f(n) masing - masing node (diurutkan dari terkecil hingga terbesar).
8. Proses tersebut dilanjutkan sampai node yang dibangkitkan memiliki nilai f(n) yang lebih besar dengan nilai f(n) *lastNode*, atau sampai *Priority Queue* kosong.

Bab III

Source Code

3.1 *Class Main* (Main.java)

Kelas Main adalah kelas yang pertama dijalankan oleh program. Pada kelas ini terdapat metode main(String args[]) yang akan dijalankan pertama. Pada metode tersebut, program akan meminta *dictionary*/kamus yang akan dipakai, kamus harus dalam bentuk .txt file pada *folder test*, pastikan setiap kata terdapat pada satu line berbeda. Program ini juga akan meminta mode *display*, terdapat dua mode yakni mode CLI dan GUI. Untuk mode CLI, akan dibentuk objek CLI yang akan menjalakan program dalam CLI, untuk mode GUI akan dibentuk objek AppGUI yang akan menjalankan program dalam bentuk GUI.

```
import java.io.IOException;
import java.util.Scanner;
import dictionary.*;
import util.StringUtil;
import cli.*;
import gui.*;
import prompt.*;

public class Main {
    public static void main(String args[]){

        Scanner scanner = new Scanner(System.in);
        Prompt prompt =  new Prompt(scanner);

        System.out.println("      _.--._  _.--._\r\n" + //
                        ",-=.-\":;:;:;\\':;:;:;\"-._\r\n" + //
                        "\\\\\\\:;:;:;:;:;\\:;:;:;:;:;\\\r\n" + //
                        " \\\\\\:;:;:;:;:;\\:;:;:;:;:;\\\r\n" + //
                        "  \\\\\\:;:;:;:;:;\\:;:;:;:;:;\\\r\n" + //
                        "   \\\\\\:;:;:;:;:;\\:;:;:;:;:\\\r\n" + //
                        "    \\\\\\;:;:;:;:;\\:;:;:;:;:\\\r\n" + //
                        "     \\\\\\;:;:_:--:\\:_:--:_;:;\\ \r\n" + //
                        "      \\\\\\_.-\"      :      \"-._\\\r\n" + //
                        "       \\`_..--\"\"--.;.--\"\"--.._=>\r\n" + //
                        "        \"\");

        System.out.println("Word Ladder Program");
        System.out.println("By Marvel Pangondian - 13522075");
        System.out.println("========================================================================");
        System.out.println("Picking dictionary");
        System.out.println("make sure the dictionary is in the test folder !");

        Dictionary dictionary;
        try {
            String dictionaryFileName = prompt.promptString("Enter Dictionary name (with .txt): ");
```

```
            dictionary = new Dictionary("test/" + dictionaryFileName);

        } catch (IOException e){
            System.out.println(StringUtil.getWordInRed("Something Went Wrong"));
            System.out.println(e.getMessage());
            scanner.close();
            return;
        }
        System.out.println("================================================================");
        System.out.println("Please select a display mode:");
        System.out.println("1. GUI (Graphical User Interface)");
        System.out.println("2. CLI (Command Line Interface)");
        int displayChoice = prompt.promptInt("Input: ");
        while (displayChoice <= 0 || displayChoice > 2) {
            System.out.println("Invalid Input! ");
            displayChoice = prompt.promptInt("Input: ");

        }
        switch (displayChoice) {
            case 1:
                AppGUI gui =  new AppGUI();
                gui.showGUI(dictionary);
                break;
            case 2:
                CLI cli = new CLI(dictionary,prompt);
                cli.showCLI();
                break;

            default:
                break;
        }
        scanner.close();

    }

}
```

3.2 *Package datastructure*

Package datastructure adalah *package* yang berisi kelas - kelas yang akan digunakan oleh algoritma UCS, *Best First Search*, serta A* sebagai data struktur.

3.2.1. *Class Node* (Node.java)

Kelas *Node* adalah kelas yang merepresentasikan kata - kata pada dictionary. Sebuah objek *Node* memiliki atribut *word (*kata pada dictionary*), value* (nilai node tersebut, dapat berupa f(n), h(n), atau g(n)), dan *path* yang menampung rangkaian kata.

```
package datastructure;
import customexception.CustomException;
import customexception.InvalidLengthWord;
import dictionary.Dictionary;
import util.*;
```

```java
public class Node {
    private String word;
    private int value; // value can be g(n), h(n), or f(n) = g(n) + f(n)
    private String path[];
    private static int nodesTraverse;
    private static int nodesGenerated;
    private static String startWord;
    private static String endWord;

    // Concstructor
    public Node(){
        this.word = "";
        this.value = -1;
        this.path = new String[]{};
    }
    // Concstructor
    // with node word and value
    public Node(String word, int value){
        this.word = word;
        this.value = value;
        this.path = new String[]{this.word};
    }

    // Constructor
    // constructor with parent path, extend parent path with current word
    public Node(String word, int value, String[] parentPath){
        this.word = word;
        this.value = value;
        int tempSize = parentPath.length;
        this.path = new String[tempSize + 1];
        System.arraycopy(parentPath, 0, this.path, 0, tempSize);
        this.path[tempSize] = word;
    }

    // Constructor
    // constructor for greedy best first search
    public Node(String word, String target ,String[] parentPath) throws
CustomException{
        this.word = word;
        this.value = this.getWordDifference(target); // heuristic function
        int tempSize = parentPath.length;
        this.path = new String[tempSize + 1];
        System.arraycopy(parentPath, 0, this.path, 0, tempSize);
        this.path[tempSize] = word;
    }

    // getters
    public String getWord(){
        return this.word;
    }

    public int getValue(){
        return this.value;
    }

    public String[] getPath(){
        return this.path;
    }

    public static String getStartWord(){
```

```
        return Node.startWord;
    }

    public static String getEndWord(){
        return Node.endWord;
    }

    public static int getNodesTraverse(){
        return Node.nodesTraverse;
    }

    public static int getNodesGenerated(){
        return Node.nodesGenerated;
    }


    // setters
    public void setPath(String[] path){
        this.path = new String[path.length];
        System.arraycopy(path, 0, this.path, 0, path.length);
    }

    public static void setStartWord(String startWord){
        Node.startWord = startWord;
    }

    public static void setEndWord(String endWord){
        Node.endWord = endWord;
    }

    public void setWord(String word){
        this.word = word;
    }

    public void setValue(int value){
        this.value = value;
    }


    // reset static variables
    public static void resetNodesTraverse(){
        Node.nodesTraverse = 0;
    }

    public static void resetNodesGenerated(){
        Node.nodesGenerated = 0;
    }

    public static void resetNodeClass(){
        Node.resetNodesGenerated();;
        Node.resetNodesTraverse();
        Node.startWord = "";
        Node.endWord = "";
    }


    // increment static variables
    public static void incrementNodesTraverse(){
        Node.nodesTraverse++;
    }
```

```
    public static void incrementNodesGenerated(){
        Node.nodesGenerated++;
    }

    // toString for debugging
    public String toString(){
        String temp = "[";
        for (int i = 0; i < this.path.length; i++) {
            temp = temp + this.path[i];

            if (i != this.path.length - 1) {
                temp += ", ";
            } else {

            }
        }
        temp += "]";
        return temp;
    }

    // printPath to print result of a node
    public void printPath(float timeInMs, int algorithmType, String
endWord,Dictionary dictionary){
        final String RED = "\u001B[31m";
        final String RESET = "\u001B[0m";
        boolean found = true;
        if (this.getWord().compareTo(endWord) != 0){
            found = false;
            System.out.println(RED + "No path can be found !" + RESET);
        }
        System.out.println("Word path : ");
        int count = 1;
        for (String word : this.path){
            System.out.print(String.format("-> %d. ", count));
            count++;
            System.out.print(StringUtil.printNodeInColor(word, endWord));
            if (!found){
                System.out.print(", child nodes :
");StringUtil.printArrNodeInColor(dictionary.getNeighbors(word).toArray(new
String[0]), endWord);
            }
            System.out.println();
        }
        if (this.getWord().compareTo(endWord) != 0){
            System.out.println(RED + "STUCK/DEAD END" + RESET);
        }
        if (algorithmType == 1){
            System.out.print("g(n) = ");
        } else if (algorithmType == 2){
            System.out.print("h(n) = ");
        } else if (algorithmType == 3){
            System.out.print("g(n) + h(n) = ");
        } else {
            return;
        }
        System.out.println(this.value);
        System.out.println("Nodes traversed : " + Node.getNodesTraverse());
        System.out.println("Nodes generated : " + Node.getNodesGenerated());
        System.out.printf("Time : %f ms\n",timeInMs);
```

```
    }

    // Heuristic function
    // is used by Node Heuristic Constructor
    public int getWordDifference(String endWord) throws CustomException{
        if (this.getWord().length() != endWord.length()){
            throw new InvalidLengthWord(this.getWord(), endWord);
        }
        int count = 0;
        char[] arr1 = this.getWord().toCharArray();
        char[] arr2 = endWord.toCharArray();
        // asumsikan kedua word memiliki leng
        for (int i = 0 ; i < arr1.length ; i++){
            if (arr1[i] != arr2[i]){
                count++;
            }
        }
        return count;
    }

}
```

3.2.2. *Class PriorityQueueNode* (PriorityQueue.java)

Kelas *PriorityQueueNode* adalah kelas yang digunakan untuk membuat objek yang digunakan sebagai penampung node. Objek *PriorityQueueNode* mematuhi aturan *Priority Queue* dimana node yang dimasukan berurut sesuai dengan *value* node tersebut.

```
package datastructure;

import java.util.Comparator;
import java.util.PriorityQueue;

public class PriorityQueueNode {
    private PriorityQueue<Node> priorityQueue; // Priority Queue Object

    // Constructor
    public PriorityQueueNode(){
        this.priorityQueue = new
PriorityQueue<>(Comparator.comparingInt(Node::getValue)); // initialize priority
queue
    }

    // addNode to insert node
    public void addNode(Node n){
        this.priorityQueue.add(n);
    }

    // dequeueNode to dequeue the first element of queue
    public Node dequeueNode(){
        return this.priorityQueue.remove();
    }

    // isEmpty to determine if queue is empty or not
```

```
    public boolean isEmpty(){
        return this.priorityQueue.isEmpty();
    }
}
```

3.3 *Package algorithm*

Package algorithm menampung implementasi - implementasi algoritma yang akan digunakan untuk menyelesaikan permainan *Word Ladder.*

3.3.1. *Interface SearchAlgorithm* (SearchAlgorithm.java)

Interace yang harus diimplementasi oleh seluruh algoritma

```
package algorithm;
import datastructure.*;
import dictionary.Dictionary;
import customexception.*;

public interface SearchAlgorithm {
    Node search(String start, String end, Dictionary dict) throws CustomException;
}
```

3.3.2. *Class UCS* (UCS.java)

Kelas UCS adalah kelas yang digunakan untuk membuat objek yang dapat menyelesaikan masalah permainan *Word Ladder* menggunakan algoritma UCS.

```
package algorithm;

import java.util.*;
import datastructure.*;
import dictionary.Dictionary;
import customexception.*;


public class UCS implements SearchAlgorithm {
    private Set<String> visited;
    public UCS(){
        visited = new HashSet<>(); // initialize set object
    }

     public Node search(String startWord, String endWord, Dictionary dictionary )
throws CustomException{
        visited.clear(); // clear visited set
        startWord = startWord.toLowerCase();
        endWord = endWord.toLowerCase();
        Node.resetNodeClass(); // reset Node static variables

        // Validation
        if (!dictionary.isWord(startWord)){
            throw new InvalidStartWordException();
```

```
        }
        if (!dictionary.isWord(endWord)){
            throw new InvalidEndWordException();
        }

        if (startWord.length() != endWord.length()){
            throw new InvalidLengthWord(startWord, endWord);
        }

        // Set start word and end word static variables
        Node.setStartWord(startWord);
        Node.setEndWord(endWord);

         // Initialize essential nodes
           Node start = new Node(startWord,0); // distance of start node with root
(itself) is 0
        Node record = new Node(); // will hold the last processed node

        // Initialize queue
        PriorityQueueNode queue = new PriorityQueueNode();
        queue.addNode(start);// first element of queue
        // visited.add(startWord);

        while (!queue.isEmpty()){

            // dequeue first element in queue
            Node currNode = queue.dequeueNode();
            Node.incrementNodesTraverse(); // increment nodes traversed

            // skip if already been visited before
            if (visited.contains(currNode.getWord())){
                continue;
            }
            record = currNode;

            //check if currNode is the end word
            if (currNode.getWord().equals(endWord) ){
                return currNode;
            }

            // Get all neighbour nodes/words
            List<String> newNodes = dictionary.getNeighbors(currNode.getWord());

            // iterate every neighbour
            for(String nodeString : newNodes){
                if (!visited.contains(nodeString)){

                        Node.incrementNodesGenerated(); // increment nodesGenerated
static variable
                          Node node = new Node(nodeString,currNode.getValue() + 1,
currNode.getPath()); // create node, with g(n) = distance of node with the start
node

                    if (node.getWord().equals(endWord) ){ // if curr node is the end
word, no need to process anymore
                        return node;
                    }

                    // add node to queue
                    queue.addNode(node);
```

```
                }
                else { // skip visited nodes
                    continue;
                }
            }
            visited.add(currNode.getWord()); // update visited after expansion
        }
        // situation where there is no solution
        return record;

    }

}
```

3.3.3. *Class GreedyBestFirst* (GreedyBestFirst.java)

Kelas *GreedyBestFirst* adalah kelas yang digunakan untuk membuat objek yang dapat menyelesaikan masalah permainan *Word Ladder* menggunakan algoritma Greedy Best First Search.

```
package algorithm;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import customexception.*;
import datastructure.*;
import dictionary.Dictionary;

public class GreedyBestFirst implements SearchAlgorithm{
    private Set<String> visited;

    public GreedyBestFirst(){
        visited = new HashSet<>(); // initialize set object
    }

    public Node search(String startWord, String endWord, Dictionary dictionary)
throws CustomException{
        visited.clear(); // clear visited set
        startWord = startWord.toLowerCase();
        endWord = endWord.toLowerCase();
        Node.resetNodeClass(); // reset Node static variabless

        // Validation
        if (!dictionary.isWord(startWord)){
            throw new InvalidStartWordException();
        }
        if (!dictionary.isWord(endWord)){
            throw new InvalidEndWordException();
        }
        if (startWord.length() != endWord.length()){
            throw new InvalidLengthWord(startWord,endWord);
        }

        // Set start word and end word static variables
        Node.setStartWord(startWord);
        Node.setEndWord(endWord);
```

```
        // Initialize essential nodes
        Node lastNode = new Node(); // will hold final answer
        Node targetNode = new Node(endWord, endWord, new String[]{}); // hold end
word, for heuristic comparison
        Node currNode = new Node(startWord,endWord,new String[]{}); // will hold
current node
        Node record = new Node(); // will hold the last node processed
        visited.add(startWord);

        while (currNode != null){
            Node.incrementNodesTraverse(); // increment nodes traversed
            record = currNode; // store last node processed

            //check if currNode is the end word
            if (currNode.getWord().equals(endWord)){
                lastNode = currNode;
                currNode = null;
                continue;
            }

            // Get all neighbour nodes/words
            List<String> allNode = dictionary.getNeighbors(currNode.getWord());

            String[] path = currNode.getPath();
            currNode = null;
            int count = 0;

            // iterate every neighbour
            for (String temp : allNode){
                if (visited.contains(temp)){ // skip visited nodes
                    continue;
                }
                Node.incrementNodesGenerated(); // increment nodesGenerated static
variable

                // picking best node for currNode
                if (count == 0){
                    currNode = new Node(temp,endWord,path);
                    count++; // count is for the first word only
                }
                else {

                    // heuristic comparison
                    // compare which node is better for currNode
                    if (targetNode.getWordDifference(temp) < currNode.getValue()){
                        currNode = new Node(temp,endWord,path);
                    }
                }
            }

            // update visited
            if (currNode != null){
                visited.add(currNode.getWord());
            }

        }

        // situation where there is no path
        if (lastNode.getValue() == -1){
            lastNode = record;
```

```
        }
        return lastNode;
    }

}
```

3.3.4. *Class AStar* (AStar.java)

Kelas *AStar* adalah kelas yang digunakan untuk membuat objek yang dapat menyelesaikan masalah permainan *Word Ladder* menggunakan algoritma A*.

```
package algorithm;

import java.util.HashSet;
import java.util.List;
import java.util.Set;

import datastructure.*;
import customexception.*;
import dictionary.Dictionary;

public class AStar implements SearchAlgorithm{
    private Set<String> visited;

    public AStar(){
        visited = new HashSet<>(); // initialize set object
    }

    public Node search(String startWord, String endWord, Dictionary dictionary )
throws CustomException{
        visited.clear(); // clear visited set
        startWord = startWord.toLowerCase();
        endWord = endWord.toLowerCase();
        Node.resetNodeClass(); // reset Node static variables

        // Validation
        if (!dictionary.isWord(startWord)){
            throw new InvalidStartWordException();
        }
        if (!dictionary.isWord(endWord)){
            throw new InvalidEndWordException();
        }

        if (startWord.length() != endWord.length()){
            throw new InvalidLengthWord(startWord, endWord);
        }

        // Set start word and end word static variables
        Node.setStartWord(startWord);
        Node.setEndWord(endWord);

        // Initialize essential nodes
        Node start = new Node(startWord,endWord,new String[]{}); // distance of
start node with root (itself) is 0
        Node lastNode = new Node(); // will hold final answer
        Node record = new Node(); // will hold the last processed node

        // Initialize queue
```

```
        PriorityQueueNode queue = new PriorityQueueNode();
        queue.addNode(start); // first element of queue
        boolean done = false;

        while (!queue.isEmpty() && !done){

            // dequeue first element in queue
            Node currNode = queue.dequeueNode();
            Node.incrementNodesTraverse(); // increment nodes traversed

            // skip visited nodes
            if (visited.contains(currNode.getWord())){
                continue;
            }

            record = currNode; // store last node processed

            // Check if the process should be stopped or not
            if (lastNode.getValue() != -1){
                if (currNode.getValue() > lastNode.getValue()){
                    done = true;
                    continue;
                }
            }

            //check if currNode is the end word
            if (currNode.getWord().equals(endWord) ){
                lastNode = currNode;
            }

            // Get all neighbour nodes/words
            List<String> newNodes = dictionary.getNeighbors(currNode.getWord());

            // iterate every neighbour
            // expansion
            for(String nodeString : newNodes){
                if (!visited.contains(nodeString)){
                    Node.incrementNodesGenerated(); // increment nodesGenerated
static variable
                    Node node = new Node(nodeString, currNode.getPath().length ,
currNode.getPath()); // create node, with f(n) = distance of node with the start
node (g(n))
                    node.setValue(node.getValue() +
node.getWordDifference(endWord)); // update value to use heuristic h(n), final value
is f(n) = g(n) + h(n)
                    queue.addNode(node); // add node to queue
                }
                else { // skip visited nodes
                    continue;
                }
            }
            visited.add(currNode.getWord()); // update visited after expansion
        }

        // situation where there is no solution
        if (lastNode.getValue() == -1){
            lastNode = record;
        }
        return lastNode;
    }
```

```
}
```

3.4 *Package dictionary*

Package dictionary berisi kelas yang akan digunakan untuk menampung kata - kata pada *dictionary/*kamus.

3.4.1. *Class Dictionary* (Dictionary.java)

Kelas *Dictionary* adalah kelas yang membuat objek untuk menampung kata - kata pada *dictionary.* Sebuab objek *Dictionary* akan membaca file *filename* yang terletak pada *folder test.*

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

public class Dictionary {
    private Set<String> words = new HashSet<>();
    private Map<String, List<String>> neighbors = new HashMap<>();

    // Constructor
    public Dictionary(String fileName) throws IOException {
        System.out.println("Loading Dictionary...");
        this.loadWords(fileName);
        this.generateNeighbors();
        System.out.println(StringUtil.getWordInGreen("DONE"));
    }


    // Load words in txt file
    private void loadWords(String fileName) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader(fileName));
        String line;
        while ((line = reader.readLine()) != null) {
            words.add(line.trim().toLowerCase());
        }
        reader.close();
    }

    // Generate neighbouring words for every word in the dictionary
    private void generateNeighbors() {
        for (String word : words) {
            List<String> adjWords = new ArrayList<String>();
            char[] wordArray = word.toCharArray();
            for (int i = 0; i < wordArray.length; i++) {
                char originalChar = wordArray[i];
                for (char c = 'a'; c <= 'z'; c++) {
                    if (c == originalChar) {
                        continue;
                    }
```

```
                    wordArray[i] = c;
                    String newWord = new String(wordArray);
                    if (words.contains(newWord)) {
                        adjWords.add(newWord);
                    }
                }
                wordArray[i] = originalChar;
            }
            neighbors.put(word, adjWords);
        }
    }

    // Determine if a word is in dictionary or not
    public boolean isWord(String word) {
        return this.words.contains(word);
    }

    // get all neighbouring words
    public List<String> getNeighbors(String word) {
        return this.neighbors.getOrDefault(word, new ArrayList<>());
    }
}
```

3.5 *Package customexception*

Package customexception adalah *package* yang terdiri atas kelas - kelas *exception* yang digunakan oleh program

3.5.1. *Class CustomException* (CustomException.java)

Kelas *CustomException* adalah kelas yang menghasilkan objek yang dapat dilempar ketika terdapat kesalahan.

```
package customexception;

public class CustomException extends Exception {
    public CustomException (){
        super("something went wrong");
    }
    public CustomException (String message) {
        super(message);
    }
}
```

3.5.2. *Class EmptyWordException* (EmptyWordException.java)

Kelas *EmptyWordException* adalah kelas yang menghasilkan objek yang dapat dilempar ketika masukan kata pengguna kosong.

```
package customexception;

public class EmptyWordException extends CustomException{
    public EmptyWordException(){
```

```
        super("Start or end word cannot be empty.");
    }
}
```

3.5.3. *Class InvalidEndWordException* (InvalidEndWordException.java)

Kelas *InvalidEndWordException* adalah kelas yang menghasilkan objek yang dapat dilempar ketika masukan kata akhir pengguna tidak valid atau tidak ada di *dictionary*.

```
package customexception;

public class InvalidEndWordException extends CustomException{
    public InvalidEndWordException(){
        super("end word doesn't exist");
    }
}
```

3.5.4. *Class InvalidStartWordException* (InvalidStartWordException.java)

Kelas *InvalidStartWordException* adalah kelas yang menghasilkan objek yang dapat dilempar ketika masukan kata pertama pengguna tidak valid atau tidak ada di *dictionary*.

```
package customexception;

public class InvalidStartWordException extends CustomException{
    public InvalidStartWordException(){
        super("start word doesn't exist");
    }
}
```

3.5.5. *Class InvalidLengthWord* (InvalidLengthWord.java)

Kelas *InvalidLengthWord* adalah kelas yang menghasilkan objek yang dapat dilempar ketika masukan kata pertama dan akhir pengguna memiliki panjang yang berbeda.

```
package customexception;

public class InvalidLengthWord extends CustomException{
    public InvalidLengthWord(String word1, String word2){
        super(word1 + " and " + word2 + " have Unequal lengths");
    }
}
```

3.6 *Package prompt*

Package prompt adalah *package* yang berisi kelas untuk menangani masukan pengguna.

3.6.1. *Class Prompt* (Prompt.java)

Kelas utama pada *package prompt,* untuk menangani masukan pengguna.

```
package prompt;

import java.util.Scanner;

public class Prompt {
    private Scanner scanner;

    public Prompt(Scanner scanner){
        this.scanner = scanner;
    }

    public String promptString(String message) {
        System.out.print(message);
        return scanner.nextLine().toLowerCase();
    }

    public int promptInt(String message) {
        System.out.print(message);
        while (!scanner.hasNextInt()) {
            scanner.next(); // Consume the invalid input
            System.out.println("Error: Input was not an integer. Please try
again.");
            System.out.print(message);
        }
        int temp = scanner.nextInt();
        scanner.nextLine();
        return temp ;
    }
}
```

3.7 *Package cli*

Package cli adalah *package* yang menangani penampilan *Command Line Interface.*

3.7.1. *Class CLI* (CLI.java)

Kelas *CLI* adalah kelas yang menangani penampilan *Command Line Interface.*

```
package cli;
import dictionary.*;
import customexception.*;
import algorithm.*;
import datastructure.*;
import prompt.*;
```

```java
public class CLI {
    private Dictionary dictionary; // Hold Dictionary object
    private Prompt prompt; // hold prompt for input

    // Constructor
    public CLI(Dictionary dictionary, Prompt prompt) {
        this.dictionary = dictionary;
        this.prompt = prompt;
    }

    // Method to show CLI
    public void showCLI(){
        int choice = 0;
        while (choice != -1) {
            try {

System.out.println("==========================================================================
=");
                System.out.println("Main Menu");
                String startWord = this.prompt.promptString("Enter Start Word: ");
                String endWord = this.prompt.promptString("Enter End Word: ");
                System.out.println("\nPick algorithm to use ");
                System.out.println("1. Uniform Cost Search Algorithm ");
                System.out.println("2. Greedy Best-First Search Algorithm ");
                System.out.println("3. A-Star Search Algorithm ");
                choice = this.prompt.promptInt("Input: ");
                System.out.println();

                SearchAlgorithm algorithm = null;
                switch (choice) {
                    case 1: algorithm = new UCS(); break;
                    case 2: algorithm = new GreedyBestFirst(); break;
                    case 3: algorithm = new AStar(); break;
                    default: System.out.println("Input Choice is invalid !");
continue;
                }

                long startTime = System.nanoTime();
                Node result = algorithm.search(startWord, endWord, dictionary);
                long endTime = System.nanoTime();
                float executionTime = (endTime - startTime) / 1_000_000.0f;
                result.printPath(executionTime, choice, endWord, dictionary);

            } catch (CustomException e) {
                System.out.println(e.getMessage());
            } catch (Exception e) {
                System.out.println("An error occurred: " + e.getMessage());
            } finally {
                choice = this.prompt.promptInt("Do you want to continue ? (-1 for
no, any number for yes): ");

            }
        }

    }
}
```

3.8 *Package gui*

Package gui adalah *package* yang menangani penampilan *Graphical User Interface. Package* ini dibuat untuk memenuhi *bonus* tugas kecil.

3.8.1. *Class AppGUI* (AppGUI.java)

Kelas *AppGUI* adalah kelas yang menangani penampilan *Graphical User Interface.*

```
package gui;
import javax.swing.*;
import algorithm.AStar;
import algorithm.GreedyBestFirst;
import algorithm.SearchAlgorithm;
import algorithm.UCS;
import customexception.CustomException;
import customexception.EmptyWordException;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import dictionary.Dictionary;
import datastructure.*;

public class AppGUI extends JFrame {
        private JTextField startWordField;
        private JTextField endWordField;
        private JComboBox<String> algorithmDropdown;
        private JButton searchButton;

        // Constructor
        public AppGUI() {
            super("Word Ladder");
        }

        // Method to initialize GUI
        private void initializeGUI(Dictionary dictionary) {
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            setSize(400, 300);
            setLayout(new GridLayout(5, 2, 10, 10));

            startWordField = new JTextField();
            endWordField = new JTextField();

              String[] algorithms = {"Uniform Cost Search", "Best First Search", "A
Star"};
            algorithmDropdown = new JComboBox<>(algorithms);

            searchButton = new JButton("Search");

            add(new JLabel("Start Word:"));
            add(startWordField);
            add(new JLabel("End Word:"));
            add(endWordField);
            add(new JLabel("Algorithm:"));
            add(algorithmDropdown);
            add(searchButton);
```

```
            searchButton.addActionListener(new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                    performSearch(dictionary);
                }
            });

            setVisible(true);
        }

        // method to peform search
        private void performSearch(Dictionary dictionary) {
            try {
                String startWord = startWordField.getText().toLowerCase();
                String endWord = endWordField.getText().toLowerCase();
                                            String   selectedAlgorithm   =   (String)
algorithmDropdown.getSelectedItem();

                if (startWord.isEmpty() || endWord.isEmpty()) {
                    throw new EmptyWordException();
                }
                SearchAlgorithm algorithm = null;
                switch (selectedAlgorithm) {
                    case "Uniform Cost Search" : algorithm = new UCS(); break;
                       case "Best First Search": algorithm = new GreedyBestFirst();
break;
                    case "A Star": algorithm = new AStar(); break;
                    default: throw new CustomException("Invalid Algorithm");
                }
                long startTime = System.nanoTime();
                Node result = algorithm.search(startWord, endWord, dictionary);
                long endTime = System.nanoTime();
                float executionTime = (endTime - startTime) / 1_000_000.0f;
                if (result.getWord().compareTo(endWord) != 0){
                            throw new CustomException(String.format("No path can be
found!\nUsing %s \nTime take: %f ms\nNodes Traversed: %d \nNodes Generated: %d",
selectedAlgorithm,executionTime,Node.getNodesTraverse(),Node.getNodesGenerated()));
                }
                                            SearchResultWindow   resultWindow   =   new
SearchResultWindow(result,executionTime,selectedAlgorithm);
                resultWindow.setVisible(true);

                  System.out.println("Starting search from: " + startWord + " to " +
endWord + " using " + selectedAlgorithm);

            }catch (CustomException e){
                    JOptionPane.showMessageDialog(this, "Error during search: " +
e.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);

            }catch (Exception e) {
                    JOptionPane.showMessageDialog(this, "Error during search: " +
e.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);

            }
        }

        public void showGUI(Dictionary dictionary) {
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    initializeGUI(dictionary);
```

```
                }
            });
        }
}
// class for search result
class SearchResultWindow extends JFrame {
        private JList<String> resultList;
        private DefaultListModel<String> listModel;

              public  SearchResultWindow(Node  result,  float  executionTime,  String
algorithmName) {
            super("Search Results");
            setLocationRelativeTo(null);

            String[] searchResults = result.getPath();

            listModel = new DefaultListModel<>();
            resultList = new JList<>(listModel);
            resultList.setFont(new Font("Arial", Font.PLAIN, 14));

            JScrollPane listScrollPane = new JScrollPane(resultList);
                listScrollPane.setBorder(BorderFactory.createEmptyBorder(10, 10, 10,
10));

            JPanel headerPanel = new JPanel(new GridLayout(6, 1));
            JLabel resultHeader = new JLabel("Result", JLabel.CENTER);
            resultHeader.setFont(new Font("Arial", Font.BOLD, 24));
            headerPanel.add(resultHeader);
            headerPanel.add(new JLabel("Using " + algorithmName, JLabel.CENTER));
                headerPanel.add(new  JLabel("Time  taken:  " + executionTime + " ms",
JLabel.CENTER));
                                headerPanel.add(new   JLabel("Nodes   Traversed:   "   +
Node.getNodesTraverse(), JLabel.CENTER));
                                headerPanel.add(new   JLabel("Nodes   Generated:   "   +
Node.getNodesGenerated(), JLabel.CENTER));
            headerPanel.add(new JLabel("Length of Path: " + result.getPath().length,
JLabel.CENTER));

            setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

            JPanel gridPanel = new JPanel(new GridLayout(searchResults.length, 1, 0,
10));
            JScrollPane gridScrollPane = new JScrollPane(gridPanel);

gridScrollPane.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR
_AS_NEEDED);

gridScrollPane.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_
NEEDED);

            String endWord = Node.getEndWord();
            for (String word : searchResults) {
                  JPanel wordPanel = new JPanel(new GridLayout(1, word.length(), 10,
0));
                wordPanel.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));
                for (int i = 0; i < word.length(); i++) {
                     JLabel letterLabel = new JLabel(String.valueOf(word.charAt(i)),
SwingConstants.CENTER);

letterLabel.setBorder(BorderFactory.createLineBorder(Color.BLACK));
```

```
                    letterLabel.setFont(new Font("Arial", Font.BOLD, 18));
                    if (endWord != null && word.charAt(i) == endWord.charAt(i)) {
                        letterLabel.setBackground(Color.GREEN);
                        letterLabel.setOpaque(true);
                    }
                    wordPanel.add(letterLabel);
                }
                gridPanel.add(wordPanel);
            }

            getContentPane().add(headerPanel, BorderLayout.NORTH);
            getContentPane().add(gridScrollPane, BorderLayout.CENTER);


            if (searchResults.length > 10) {
                setSize(600, 800);
            } else {
                setSize(600, 400);
            }

            setVisible(true);
        }
}
```

3.9 *Package util*

Package util adalah *package* yang berisi *utilities* yang digunakan oleh program.

3.9.1. *Class StringUtil* (StringUtil.java)

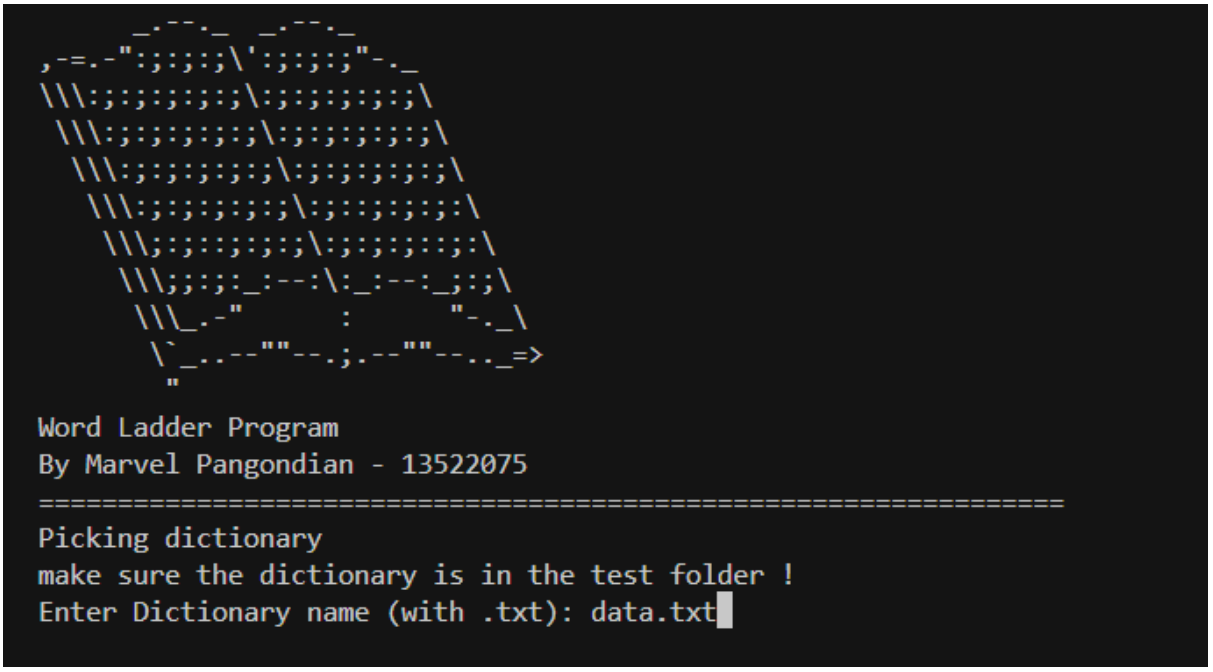
Kelas *StringUtil* adalah kelas yang menangani utilities berhubungan dengan objek *String*.

```
package util;

public class StringUtil {

    // Static method to print a word in color, green letters for letters that are
the same as endWord
    public static String printNodeInColor(String startWord, String endWord){
        final String ANSI_GREEN = "\u001B[32m";
        final String ANSI_RESET = "\u001B[0m";
        StringBuilder coloredOutput = new StringBuilder();

        for (int i = 0; i < startWord.length(); i++) {
            if (i < endWord.length() && startWord.charAt(i) == endWord.charAt(i)) {

coloredOutput.append(ANSI_GREEN).append(startWord.charAt(i)).append(ANSI_RESET);
            } else {
                coloredOutput.append(startWord.charAt(i));
            }
        }
    return coloredOutput.toString();
    }

    // Static method to get string in red color
    public static String getWordInRed(String word){
```

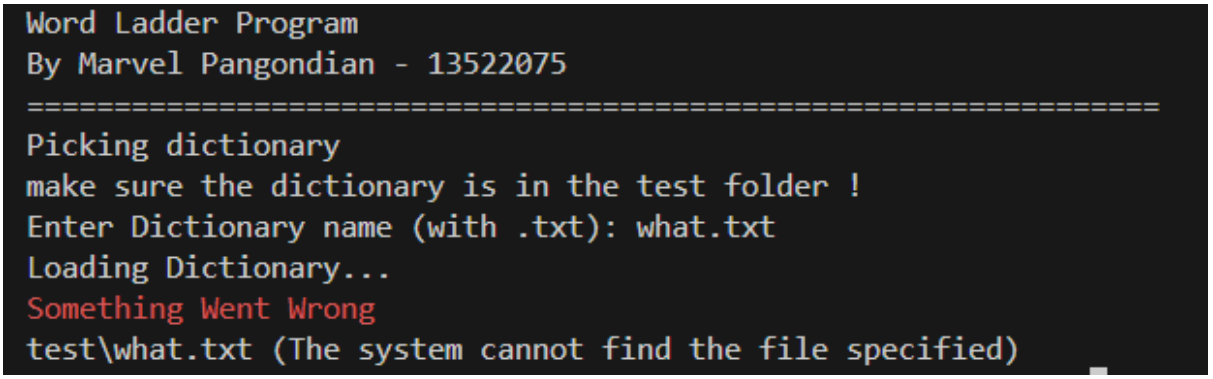
```
        final String ANSI_RESET = "\u001B[0m";
        final String RED = "\u001B[31m";
        return RED + word + ANSI_RESET;
    }

    // Static method to get string in green color
    public static String getWordInGreen(String word){
        final String ANSI_GREEN = "\u001B[32m";
        final String ANSI_RESET = "\u001B[0m";
        return ANSI_GREEN + word + ANSI_RESET;
    }

    // Static method to print array of words/nodes in color
    public static void printArrNodeInColor(String[] nodes, String endWord){
        System.out.print("[");
        for (int i = 0 ; i < nodes.length ; i++){
            System.out.print(StringUtil.printNodeInColor(nodes[i], endWord));
            if (i != nodes.length - 1){
                System.out.print(", ");
            }
        }
        System.out.print("]");
    }

    // Static method to highlight characters, will be used in GUI
    public static String highlightCertainCharacters(String word, String target) {
        StringBuilder highlighted = new StringBuilder();
        for (int i = 0; i < word.length(); i++) {
            if (i < target.length() && word.charAt(i) == target.charAt(i)) {
                highlighted.append("<span
style='color:green'>").append(word.charAt(i)).append("</span>");
            } else {
                highlighted.append(word.charAt(i));
            }
        }
        return highlighted.toString();
    }
}
```

Bab IV
TEST DAN ANALISIS

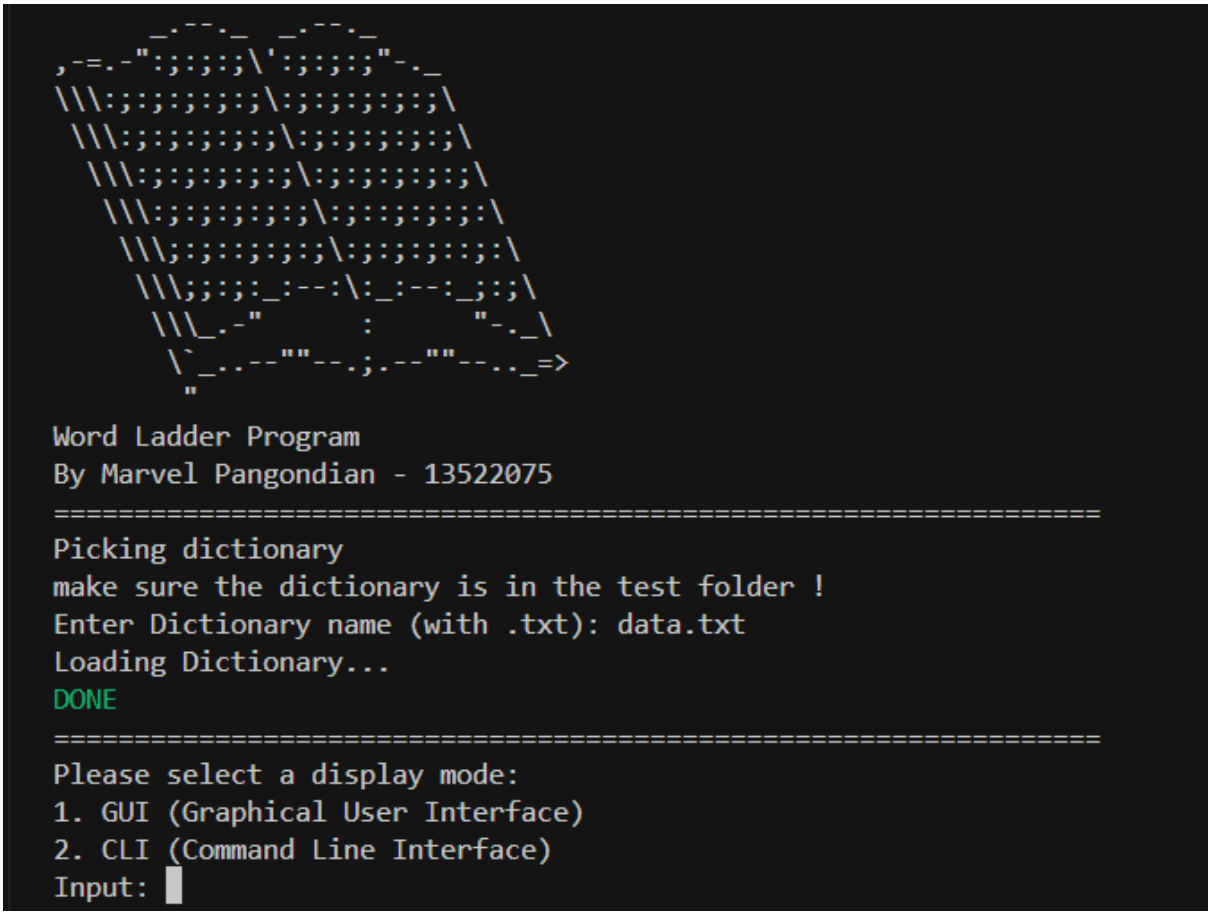
4.1 Tampilan Utama



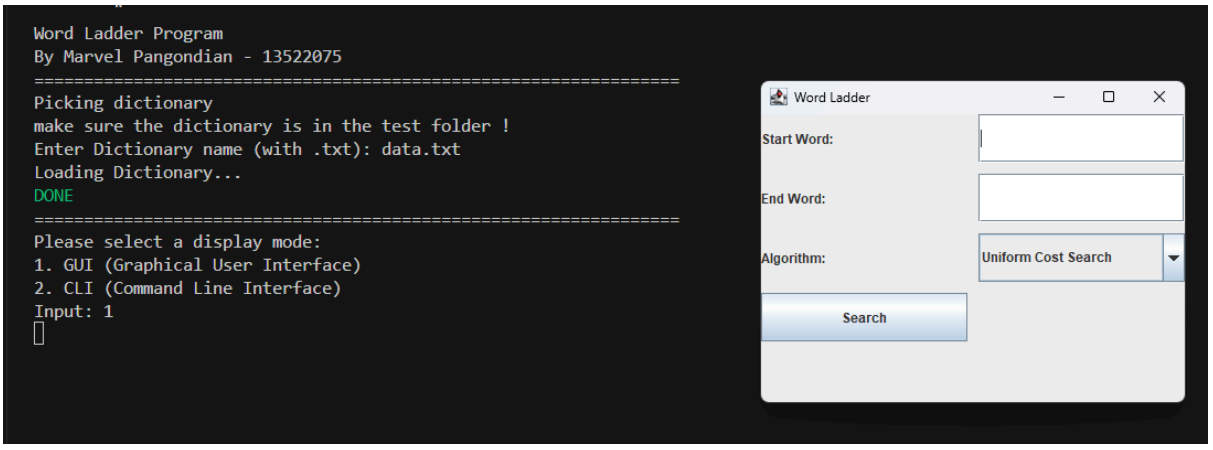
Gambar 4.1.1 Tampilan Utama Program.



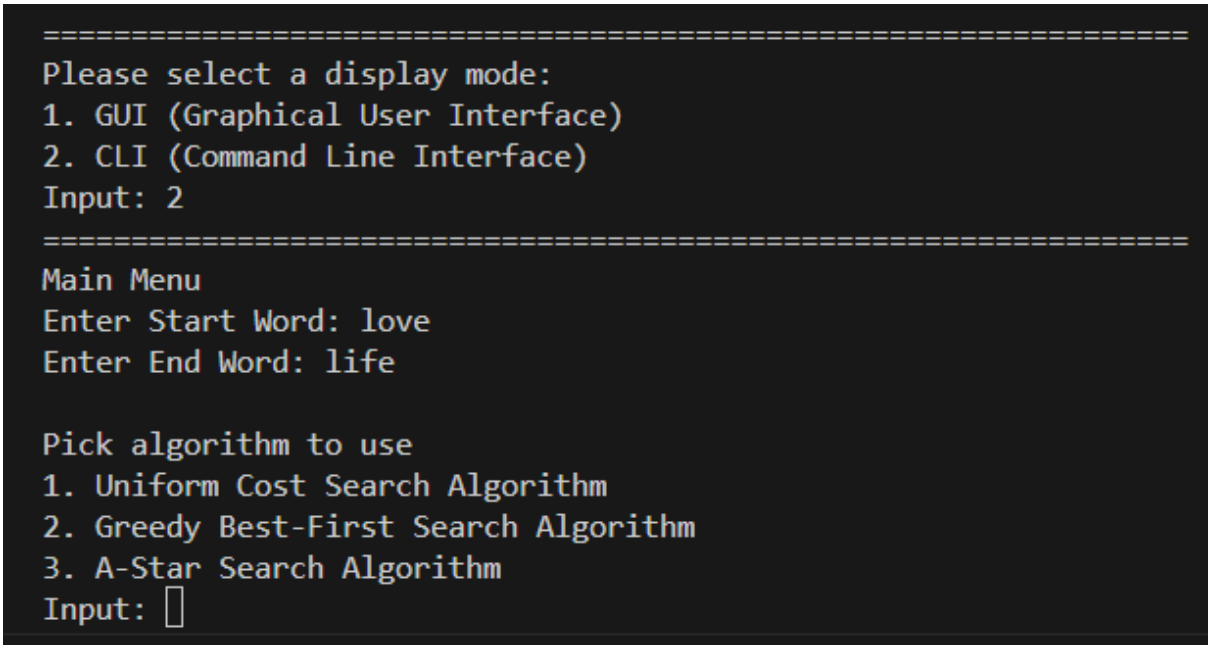
Gambar 4.1.2 Tampilan Utama Program ketika masukan file kamus tidak ada pada folder test.



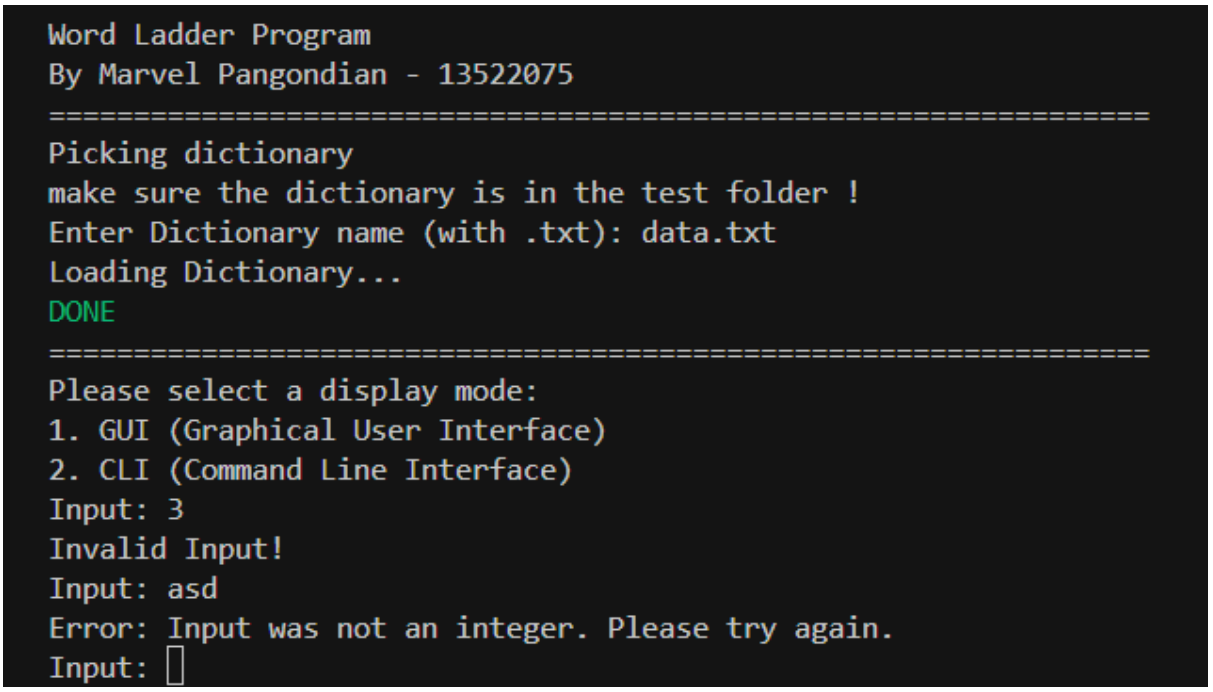
Gambar 4.1.3 Tampilan Utama Program untuk memilih mode tampilan.



Gambar 4.1.4 Tampilan Utama Program ketika memilih GUI.

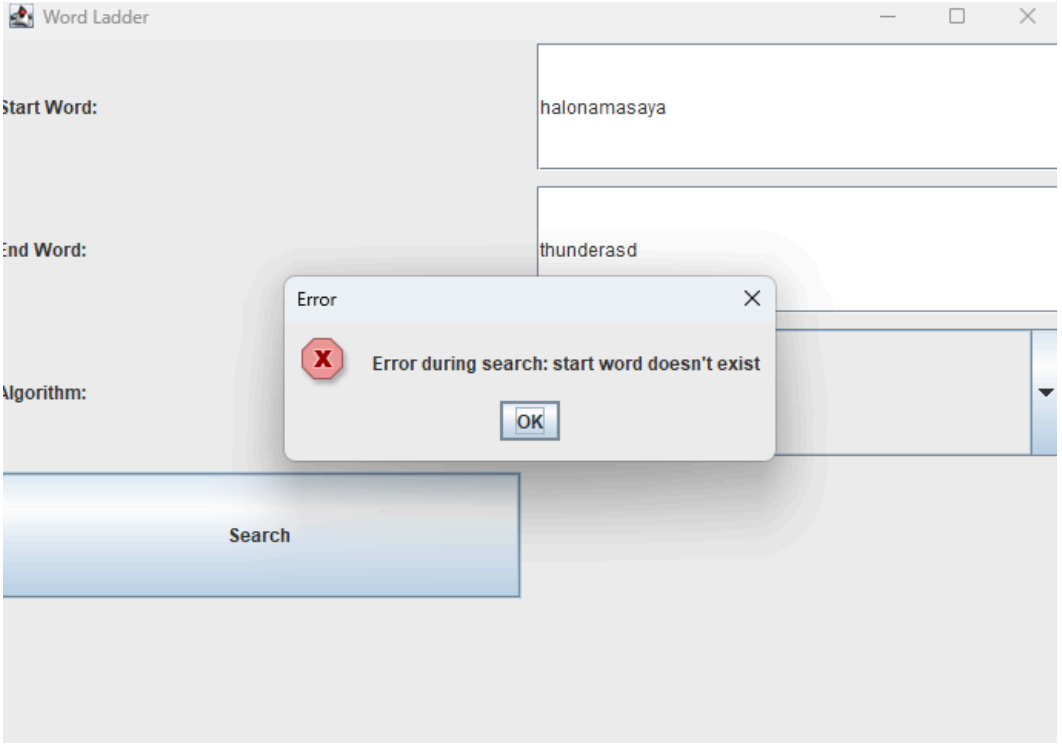


Gambar 4.1.5 Tampilan Utama Program ketika memilih CLI.

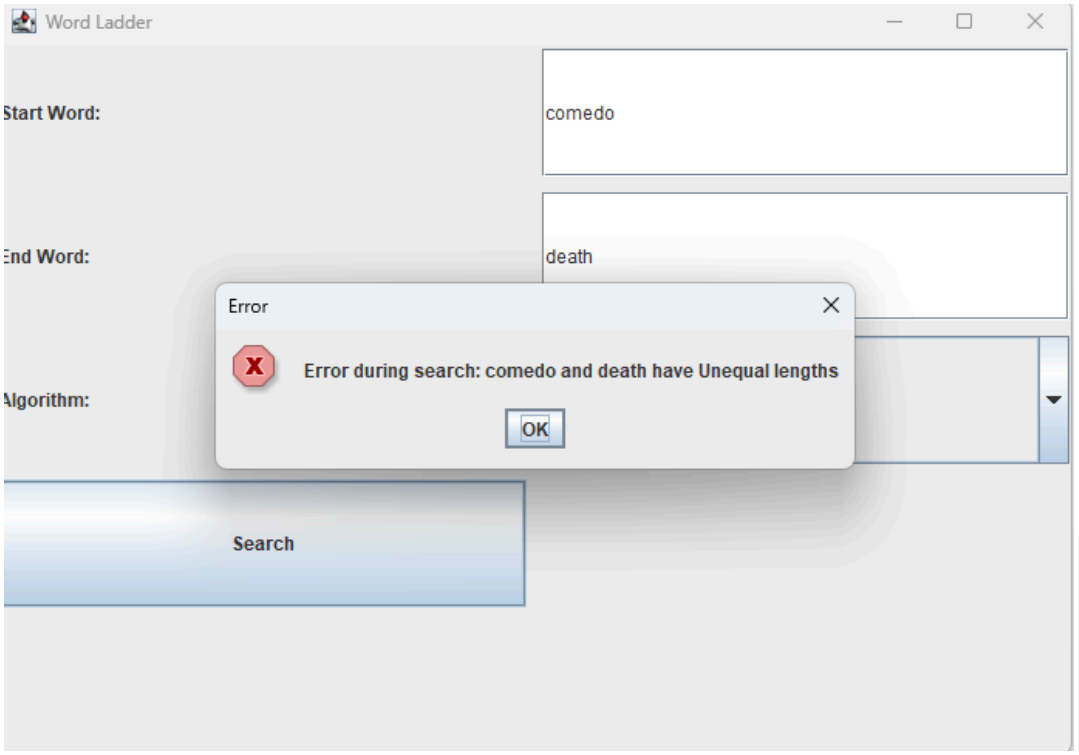


Gambar 4.1.6 Tampilan Utama ketika masukan tidak valid.

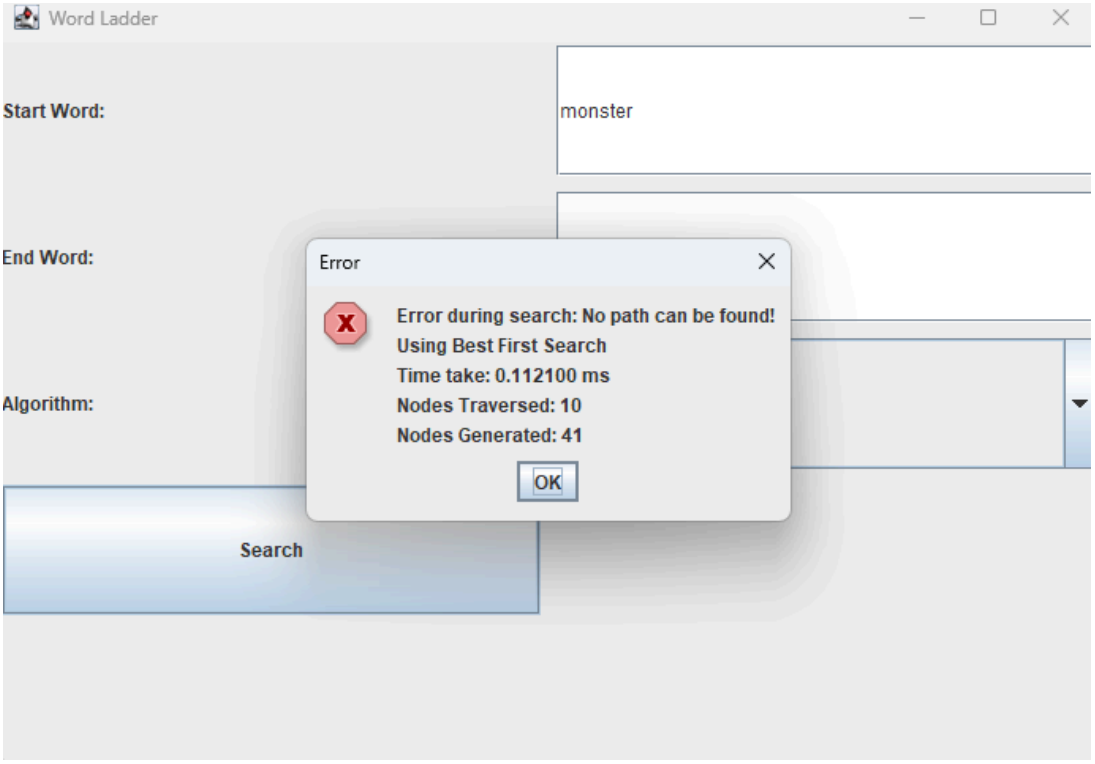
4.2 Tampilan GUI dan Skema Validasi



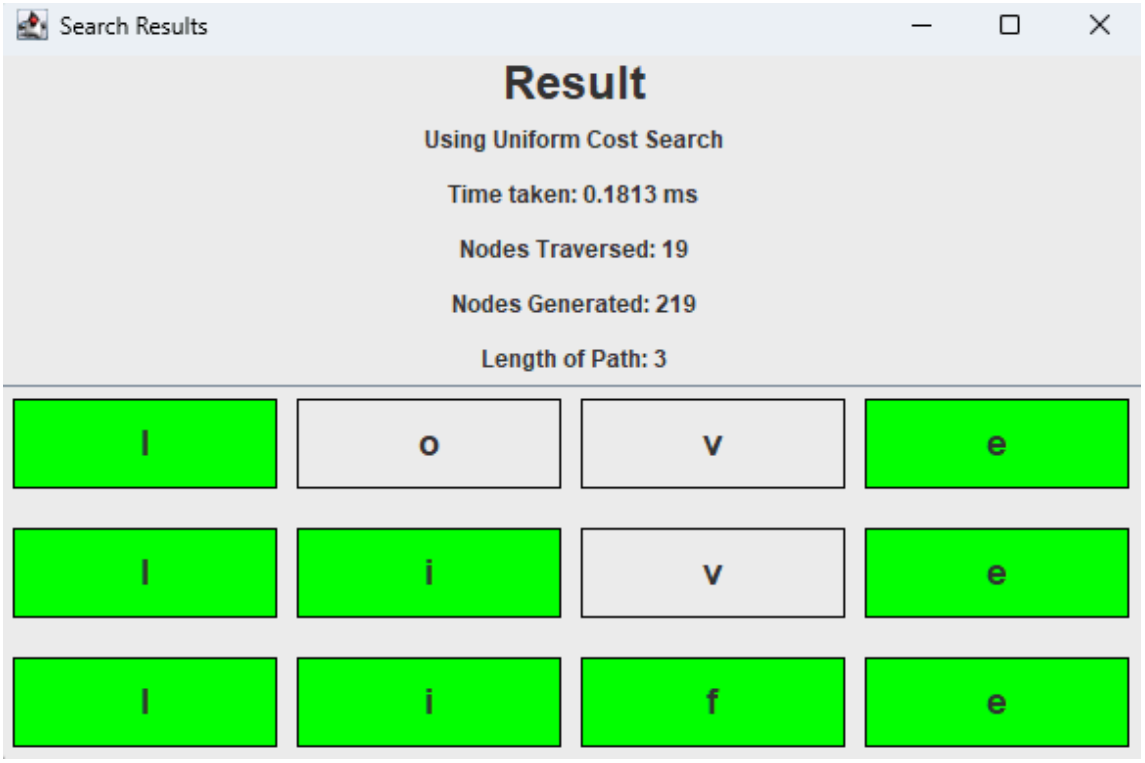
Gambar 4.2.1 Tampilan GUI ketika memasukan kata yang tidak ada dalam *dictionary*



Gambar 4.2.2 Tampilan GUI ketika kata awal dan kata akhir memiliki panjang yang berbeda.

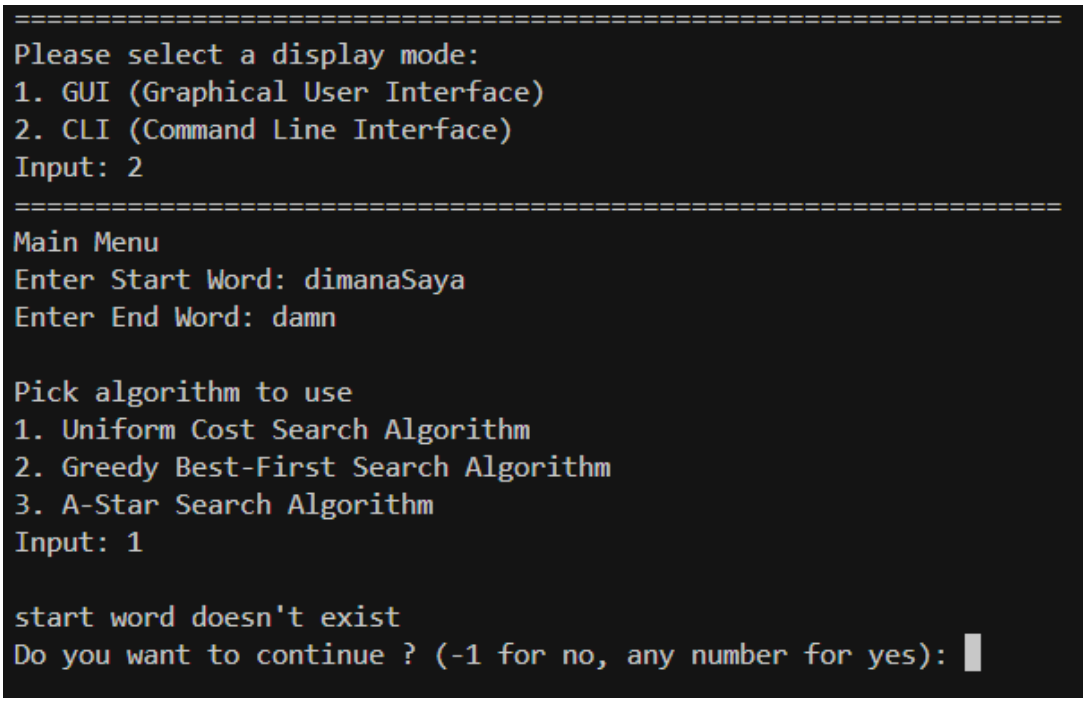


Gambar 4.2.3 Tampilan GUI ketika tidak ditemukan solusi rangkaian kata.

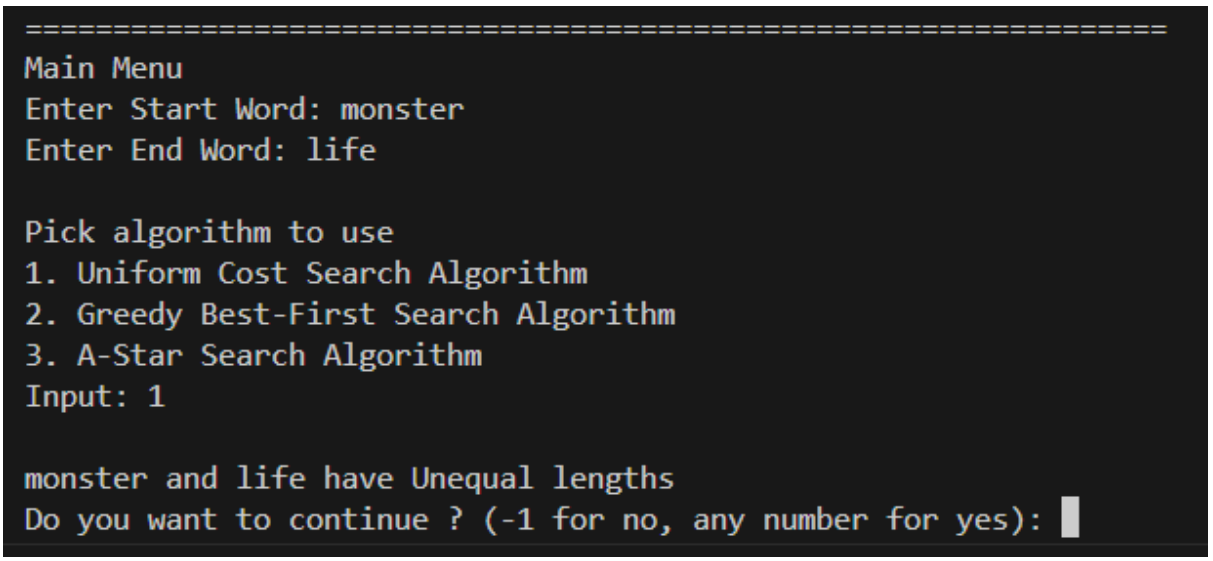


Gambar 4.2.4 Tampilan GUI ketika ditemukan solusi rangkaian kata.

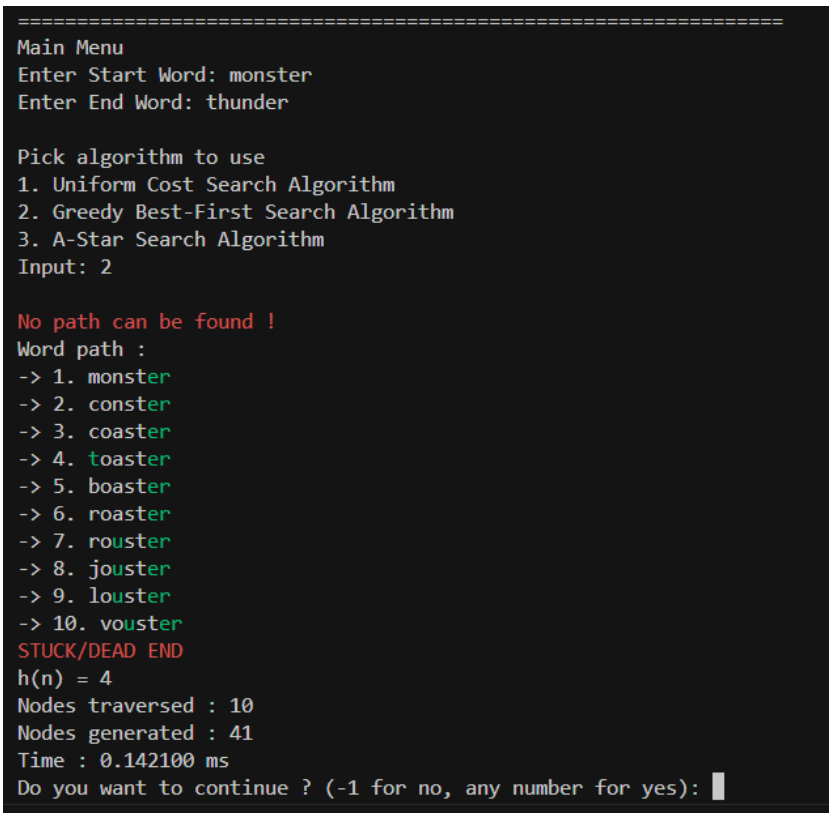
4.3 Tampilan CLI dan Skema Validasi



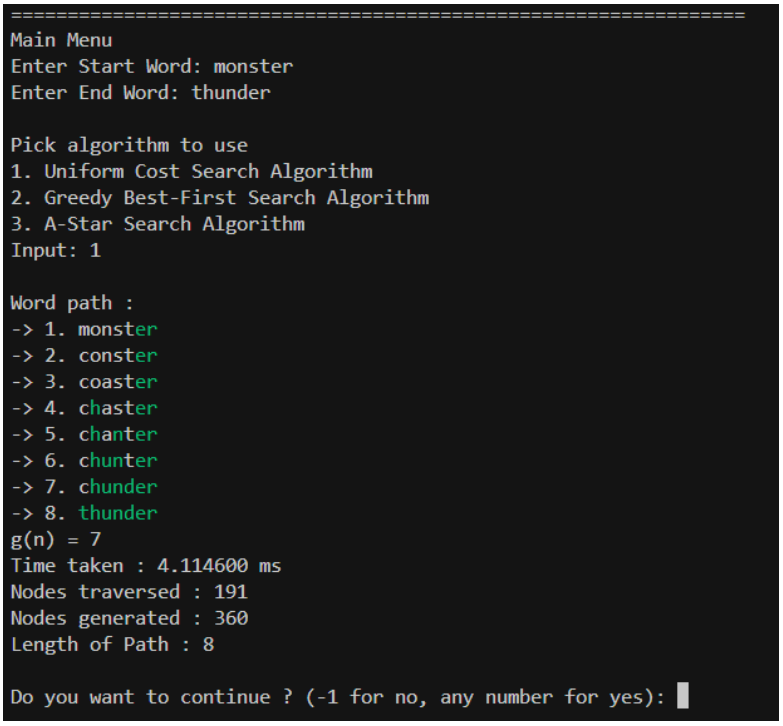
Gambar 4.3.1 Tampilan CLI ketika memasukan kata yang tidak ada dalam *dictionary*



Gambar 4.3.2 Tampilan CLI ketika kata awal dan kata akhir memiliki panjang yang berbeda.



Gambar 4.3.3 Tampilan CLI ketika tidak ditemukan solusi rangkaian kata.



Gambar 4.3.4 Tampilan CLI ketika ditemukan solusi rangkaian kata.

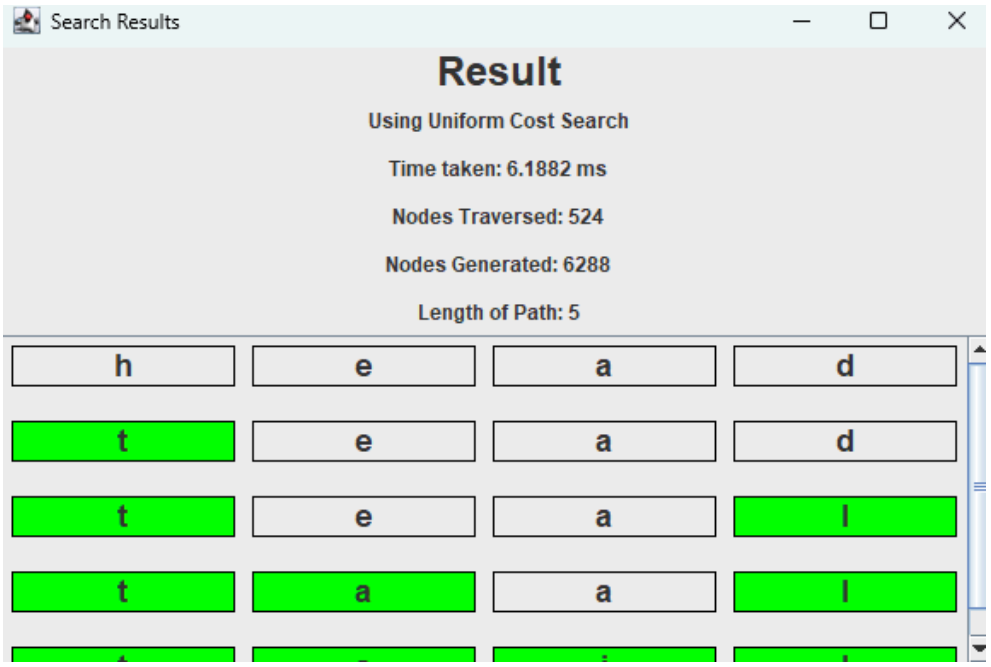
4.4 Test Case dengan *data.txt*

Test case akan dilakukan dengan menggunakan *data.txt* yang merupakan *dictionary* berisi kata - kata bahasa inggris. Penulis mendapatkan *dictionary* tersebut dari https://github.com/dwyl/english-words yakni file *words_alpha.txt*. Test case juga akan ditampilkan dalam CLI dan GUI, untuk 2 test case pertama akan dalam GUI, sedangkan 8 terakhir dalam CLI untuk mempermudah analisis. Test case yang digunakan adalah 10 kata yakni :

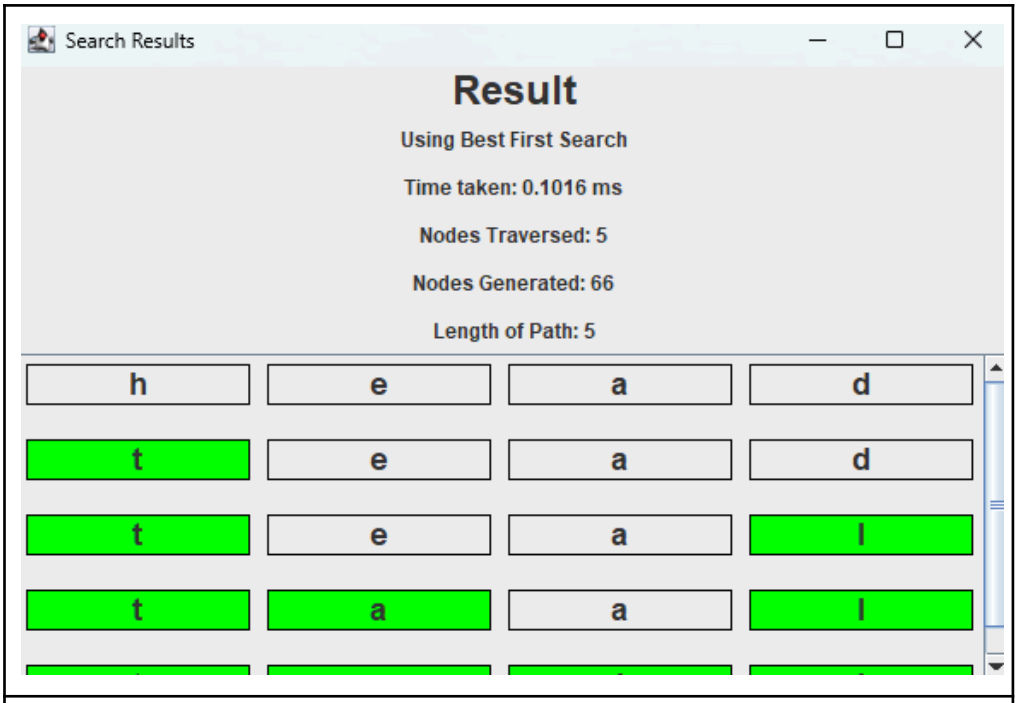
Berikut adalah hasil test case :

1. Head -> Tail

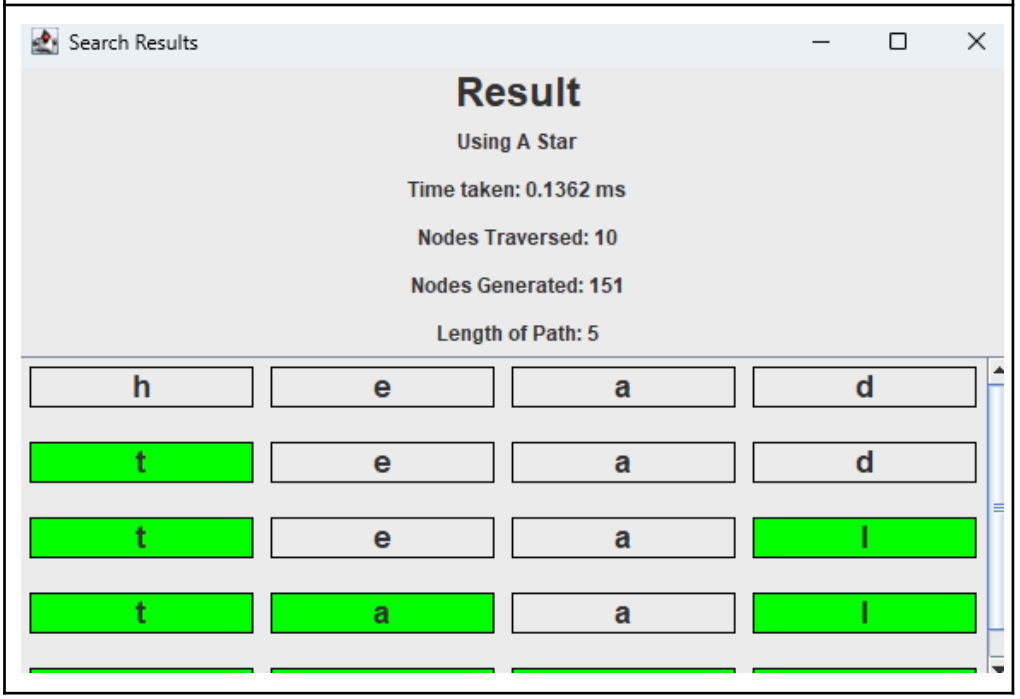
UCS



Greedy Best First Search

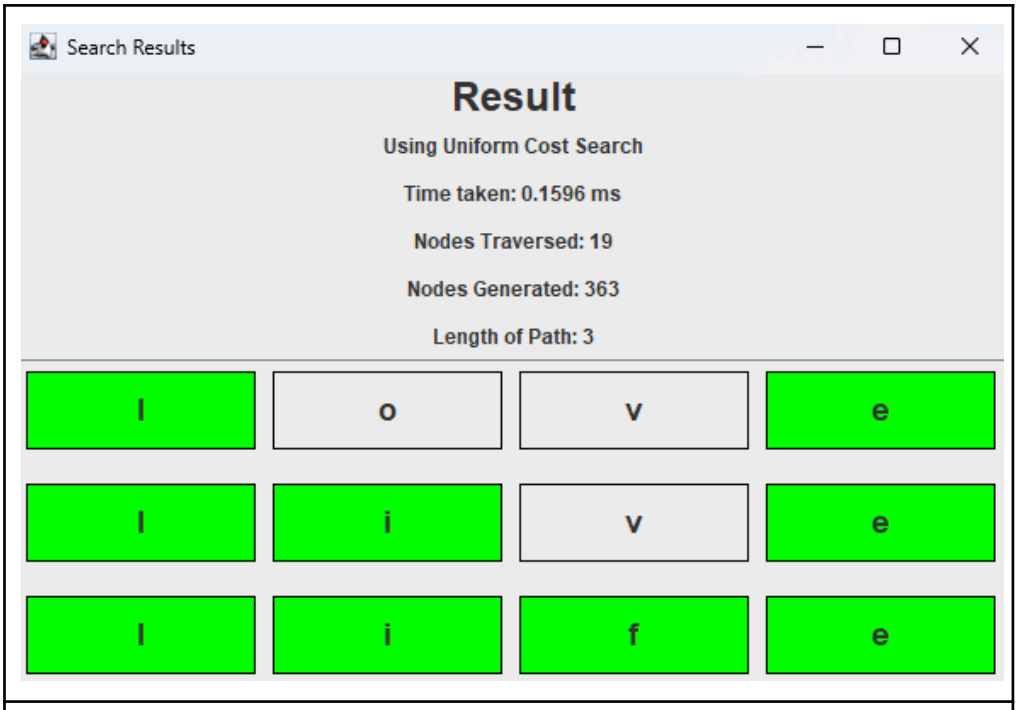


A*



2. Love -> life

UCS



Greedy Best First Search

A*

3. Grass -> Green

UCS
Main Menu Enter Start Word: grass Enter End Word: green Pick algorithm to use 1. Uniform Cost Search Algorithm 2. Greedy Best-First Search Algorithm 3. A-Star Search Algorithm Input: `1 Error: Input was not an integer. Please try again. Input: 1 Word path : -> 1. grass -> 2. grays -> 3. greys -> 4. grees -> 5. green g(n) = 4 Time taken : 0.642300 ms Nodes traversed : 234 Nodes generated : 1058 Length of Path : 5
Greedy Best First Search

```
Pick algorithm to use
1. Uniform Cost Search Algorithm
2. Greedy Best-First Search Algorithm
3. A-Star Search Algorithm
Input: 2

Word path :
-> 1. grass
-> 2. gross
-> 3. gruss
-> 4. grues
-> 5. grees
-> 6. green
h(n) = 0
Time taken : 0.044600 ms
Nodes traversed : 6
Nodes generated : 48
Length of Path : 6
```

A*

```
Pick algorithm to use
1. Uniform Cost Search Algorithm
2. Greedy Best-First Search Algorithm
3. A-Star Search Algorithm
Input: 3

Word path :
-> 1. grass
-> 2. grays
-> 3. greys
-> 4. grees
-> 5. green
g(n) + h(n) = 4
Time taken : 0.250900 ms
Nodes traversed : 13
Nodes generated : 93
Length of Path : 5
```

4. Plant -> bloom

UCS

```
Main Menu
Enter Start Word: plant
Enter End Word: bloom

Pick algorithm to use
1. Uniform Cost Search Algorithm
2. Greedy Best-First Search Algorithm
3. A-Star Search Algorithm
Input: 1

Word path :
-> 1. plant
-> 2. alant
-> 3. aland
-> 4. bland
-> 5. blond
-> 6. blood
-> 7. bloom
g(n) = 6
Time taken : 5.072900 ms
Nodes traversed : 5208
Nodes generated : 11576
Length of Path : 7
```

Greedy Best First Search

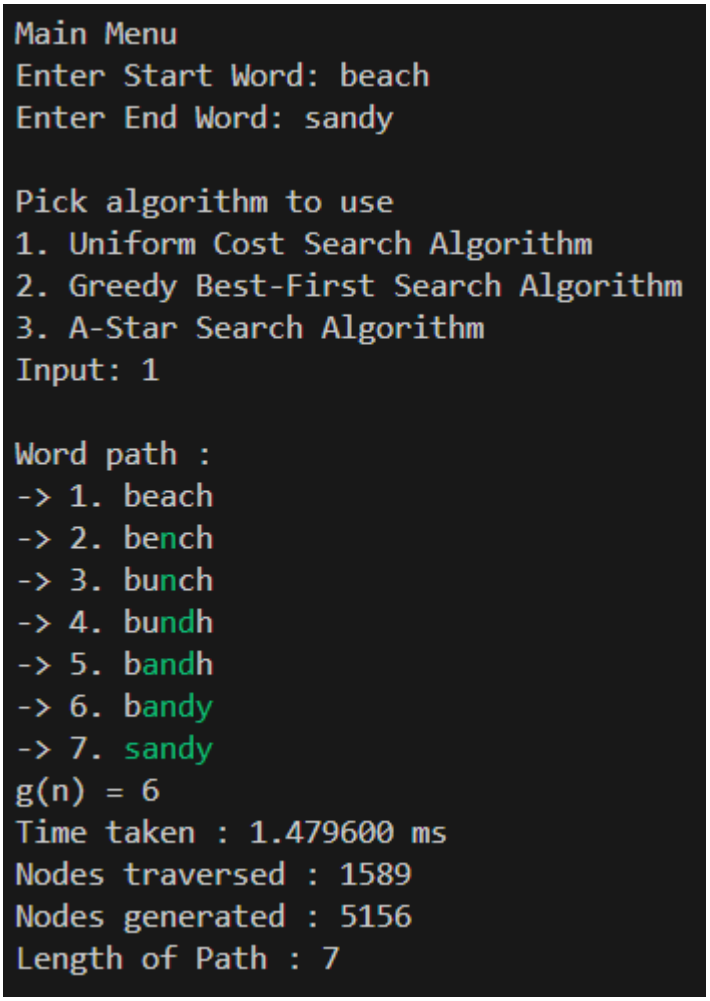
```
Pick algorithm to use
1. Uniform Cost Search Algorithm
2. Greedy Best-First Search Algorithm
3. A-Star Search Algorithm
Input: 2

Word path :
-> 1. plant
-> 2. alant
-> 3. slant
-> 4. slent
-> 5. blent
-> 6. blunt
-> 7. bluet
-> 8. blurt
-> 9. blart
-> 10. blirt
-> 11. blist
-> 12. blast
-> 13. blest
-> 14. bleat
-> 15. bloat
-> 16. blout
-> 17. alout
-> 18. clout
-> 19. cloot
-> 20. sloot
-> 21. sloom
-> 22. bloom
h(n) = 0
Time taken : 0.153200 ms
Nodes traversed : 22
Nodes generated : 142
Length of Path : 22
```

A*

```
Main Menu
Enter Start Word: plant
Enter End Word: bloom

Pick algorithm to use
1. Uniform Cost Search Algorithm
2. Greedy Best-First Search Algorithm
3. A-Star Search Algorithm
Input: 3

Word path :
-> 1. plant
-> 2. plank
-> 3. blank
-> 4. bland
-> 5. blond
-> 6. blood
-> 7. bloom
g(n) + h(n) = 6
Time taken : 0.218200 ms
Nodes traversed : 115
Nodes generated : 583
Length of Path : 7
```

5. Beach -> Sandy

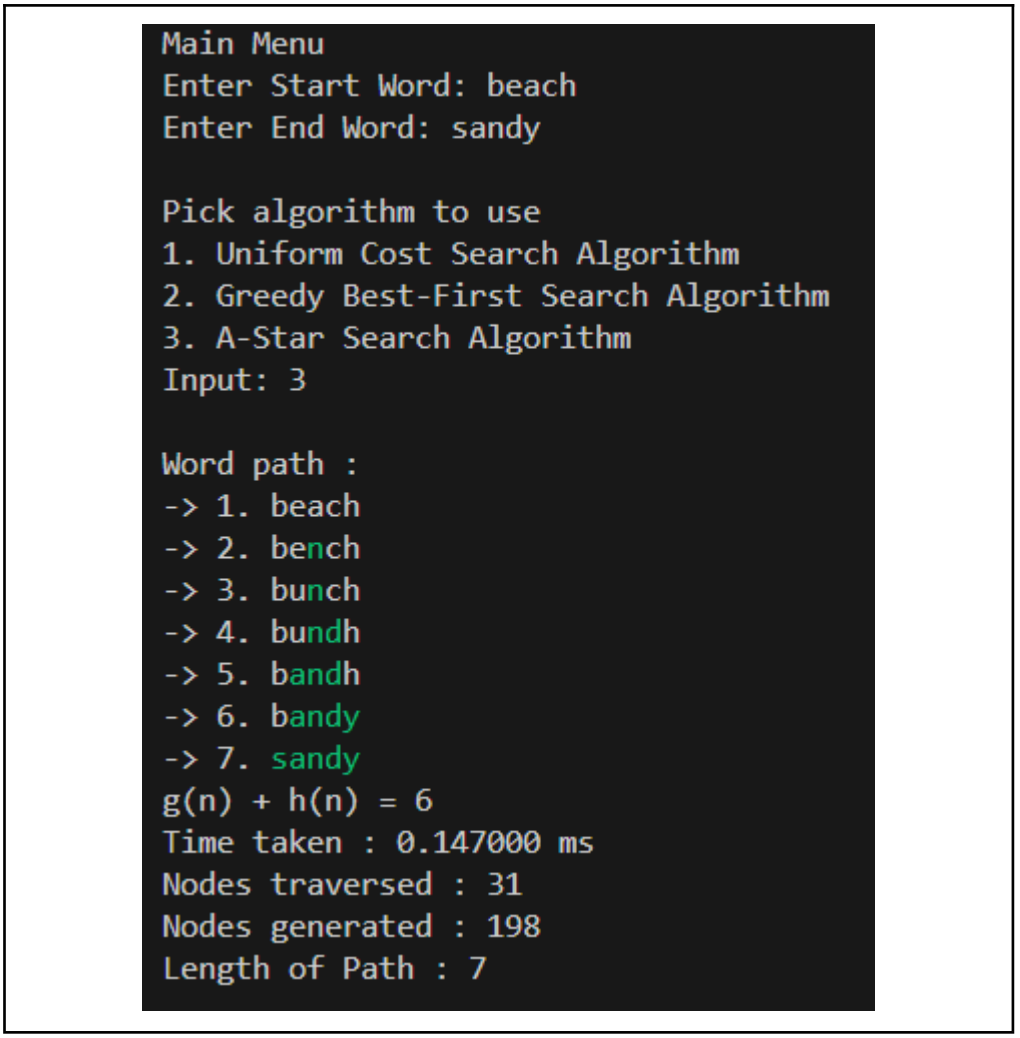
UCS



Greedy Best First Search

```
Main Menu
Enter Start Word: beach
Enter End Word: sandy

Pick algorithm to use
1. Uniform Cost Search Algorithm
2. Greedy Best-First Search Algorithm
3. A-Star Search Algorithm
Input: 2

Word path :
-> 1. beach
-> 2. bench
-> 3. kench
-> 4. lench
-> 5. lanch
-> 6. canch
-> 7. ganch
-> 8. hanch
-> 9. ranch
-> 10. rance
-> 11. dance
-> 12. dancy
-> 13. sancy
-> 14. sandy
h(n) = 0
Time taken : 0.101200 ms
Nodes traversed : 14
Nodes generated : 101
Length of Path : 14
```

A*



6. Right -> Wrong

UCS

```
Pick algorithm to use
1. Uniform Cost Search Algorithm
2. Greedy Best-First Search Algorithm
3. A-Star Search Algorithm
Input: 1

Word path :
-> 1. right
-> 2. bight
-> 3. bigot
-> 4. begot
-> 5. begut
-> 6. beaut
-> 7. beant
-> 8. brant
-> 9. orant
-> 10. orang
-> 11. wrang
-> 12. wrong
g(n) = 11
Time taken : 16.828199 ms
Nodes traversed : 32070
Nodes generated : 40797
Length of Path : 12
```

Greedy Best First Search

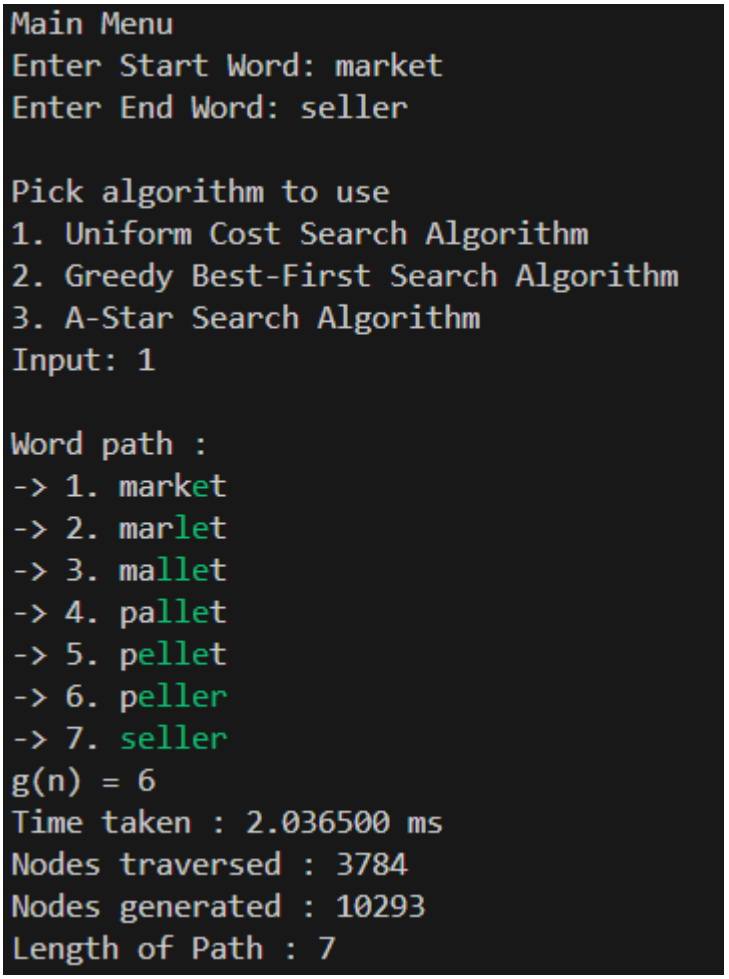
```
Main Menu
Enter Start Word: right
Enter End Word: wrong

Pick algorithm to use
1. Uniform Cost Search Algorithm
2. Greedy Best-First Search Algorithm
3. A-Star Search Algorithm
Input: 2

Word path :
-> 1. right
-> 2. wight
-> 3. wicht
-> 4. wecht
-> 5. hecht
-> 6. pecht
-> 7. pacht
-> 8. hacht
-> 9. jacht
-> 10. yacht
-> 11. yasht
-> 12. dasht
-> 13. dasnt
-> 14. wasnt
-> 15. warnt
-> 16. warns
-> 17. wains
-> 18. whins
-> 19. whing
-> 20. wring
-> 21. wrong
h(n) = 0
Time taken : 0.090500 ms
Nodes traversed : 21
Nodes generated : 125
Length of Path : 21
```

A*

```
Pick algorithm to use
1. Uniform Cost Search Algorithm
2. Greedy Best-First Search Algorithm
3. A-Star Search Algorithm
Input: 3

Word path :
-> 1. right
-> 2. hight
-> 3. hicht
-> 4. hacht
-> 5. hasht
-> 6. hasnt
-> 7. haunt
-> 8. daunt
-> 9. drunt
-> 10. drung
-> 11. wrung
-> 12. wrong
g(n) + h(n) = 11
Time taken : 2.559700 ms
Nodes traversed : 2132
Nodes generated : 7753
Length of Path : 12
```

7. Market -> Seller

UCS



Greedy Best First Search

```
Main Menu
Enter Start Word: market
Enter End Word: seller

Pick algorithm to use
1. Uniform Cost Search Algorithm
2. Greedy Best-First Search Algorithm
3. A-Star Search Algorithm
Input: 2

Word path :
-> 1. market
-> 2. marlet
-> 3. mallet
-> 4. sallet
-> 5. sallee
-> 6. mallee
-> 7. malled
-> 8. melled
-> 9. meller
-> 10. seller
h(n) = 0
Time taken : 0.059000 ms
Nodes traversed : 10
Nodes generated : 84
Length of Path : 10
```

A*

```
Main Menu
Enter Start Word: market
Enter End Word: seller

Pick algorithm to use
1. Uniform Cost Search Algorithm
2. Greedy Best-First Search Algorithm
3. A-Star Search Algorithm
Input: 3

Word path :
-> 1. market
-> 2. marlet
-> 3. mallet
-> 4. mullet
-> 5. muller
-> 6. meller
-> 7. seller
g(n) + h(n) = 6
Time taken : 0.169800 ms
Nodes traversed : 65
Nodes generated : 497
Length of Path : 7
```

8. Branch -> Leaves

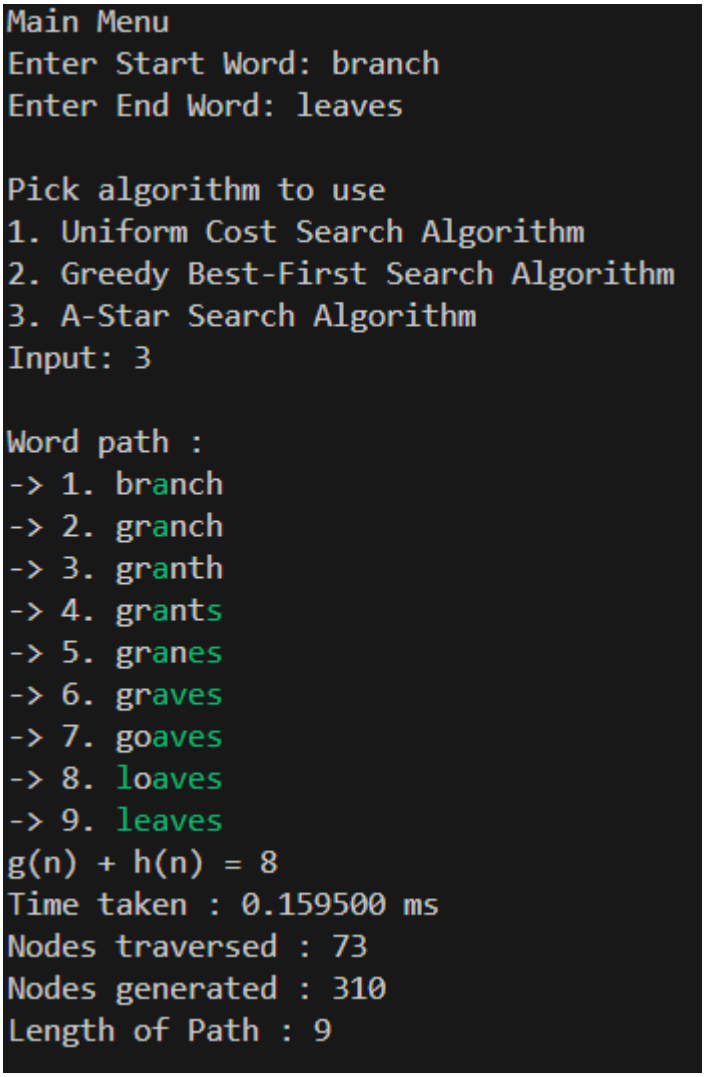
UCS

```
Main Menu
Enter Start Word: branch
Enter End Word: leaves

Pick algorithm to use
1. Uniform Cost Search Algorithm
2. Greedy Best-First Search Algorithm
3. A-Star Search Algorithm
Input: 1

Word path :
-> 1. branch
-> 2. granch
-> 3. granth
-> 4. grants
-> 5. granes
-> 6. graves
-> 7. goaves
-> 8. loaves
-> 9. leaves
g(n) = 8
Time taken : 0.709900 ms
Nodes traversed : 880
Nodes generated : 1771
Length of Path : 9
```

Greedy Best First Search

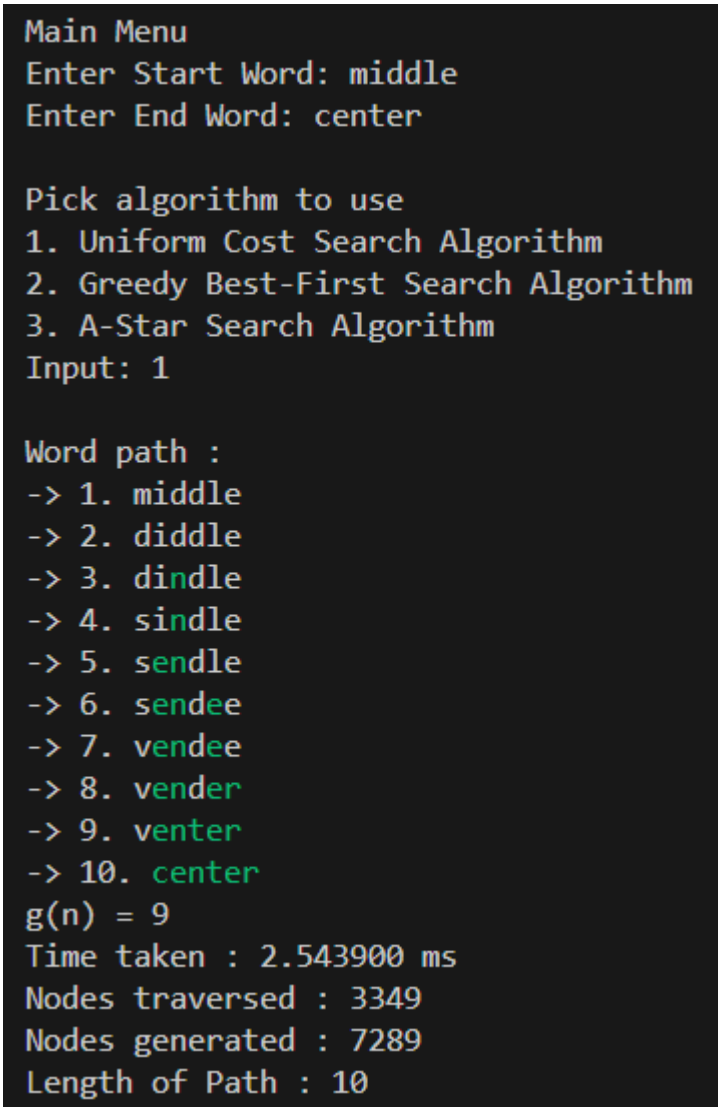
```
Main Menu
Enter Start Word: branch
Enter End Word: leaves

Pick algorithm to use
1. Uniform Cost Search Algorithm
2. Greedy Best-First Search Algorithm
3. A-Star Search Algorithm
Input: 2

Word path :
-> 1. branch
-> 2. cranch
-> 3. granch
-> 4. granth
-> 5. grants
-> 6. granes
-> 7. graves
-> 8. braves
-> 9. craves
-> 10. traves
-> 11. troves
-> 12. droves
-> 13. groves
-> 14. proves
-> 15. probes
-> 16. proles
-> 17. proses
-> 18. prases
-> 19. peases
-> 20. leases
-> 21. leaves
h(n) = 0
Time taken : 0.089100 ms
Nodes traversed : 21
Nodes generated : 145
Length of Path : 21
```

A*



9. Middle -> Center

UCS



Greedy Best First Search

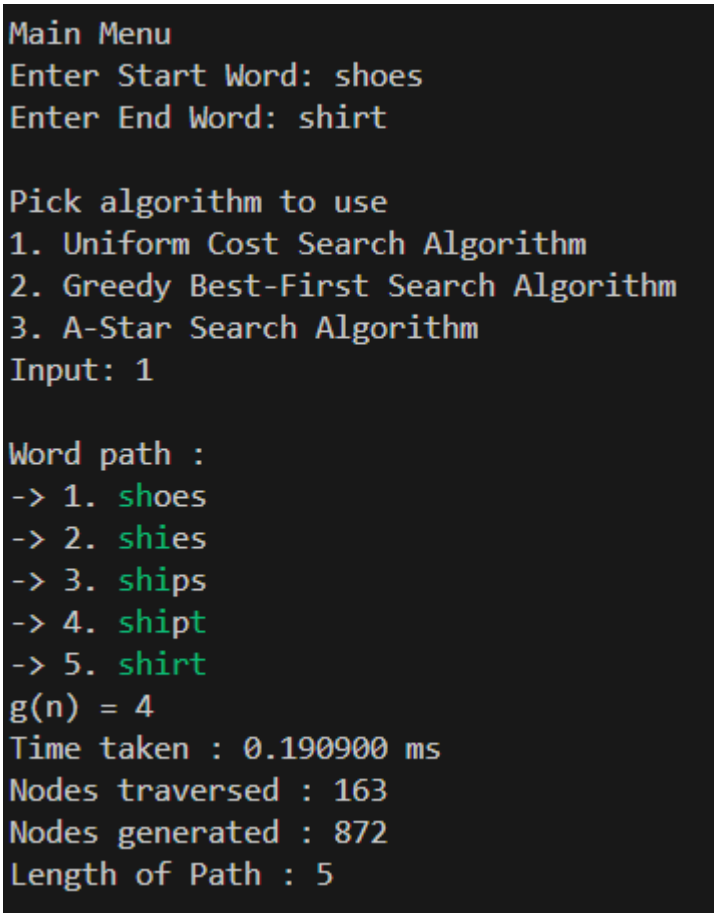
```
Main Menu
Enter Start Word: middle
Enter End Word: center

Pick algorithm to use
1. Uniform Cost Search Algorithm
2. Greedy Best-First Search Algorithm
3. A-Star Search Algorithm
Input: 2

Word path :
-> 1. middle
-> 2. meddle
-> 3. heddle
-> 4. peddle
-> 5. pendle
-> 6. sendle
-> 7. sendee
-> 8. sender
-> 9. bender
-> 10. fender
-> 11. fenter
-> 12. center
h(n) = 0
Time taken : 0.044100 ms
Nodes traversed : 12
Nodes generated : 102
Length of Path : 12
```

A*

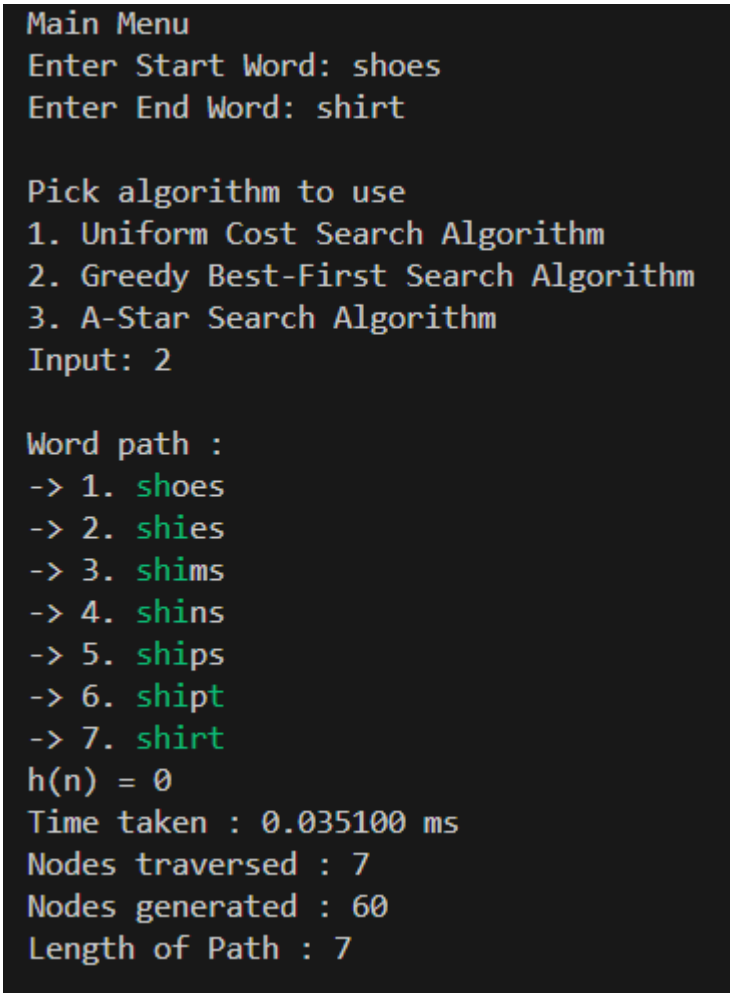
```
Main Menu
Enter Start Word: middle
Enter End Word: center

Pick algorithm to use
1. Uniform Cost Search Algorithm
2. Greedy Best-First Search Algorithm
3. A-Star Search Algorithm
Input: 3

Word path :
-> 1. middle
-> 2. diddle
-> 3. dindle
-> 4. sindle
-> 5. sendle
-> 6. sendee
-> 7. vendee
-> 8. vender
-> 9. venter
-> 10. center
g(n) + h(n) = 9
Time taken : 0.605400 ms
Nodes traversed : 537
Nodes generated : 1676
Length of Path : 10
```

10. Shoes -> Shirt

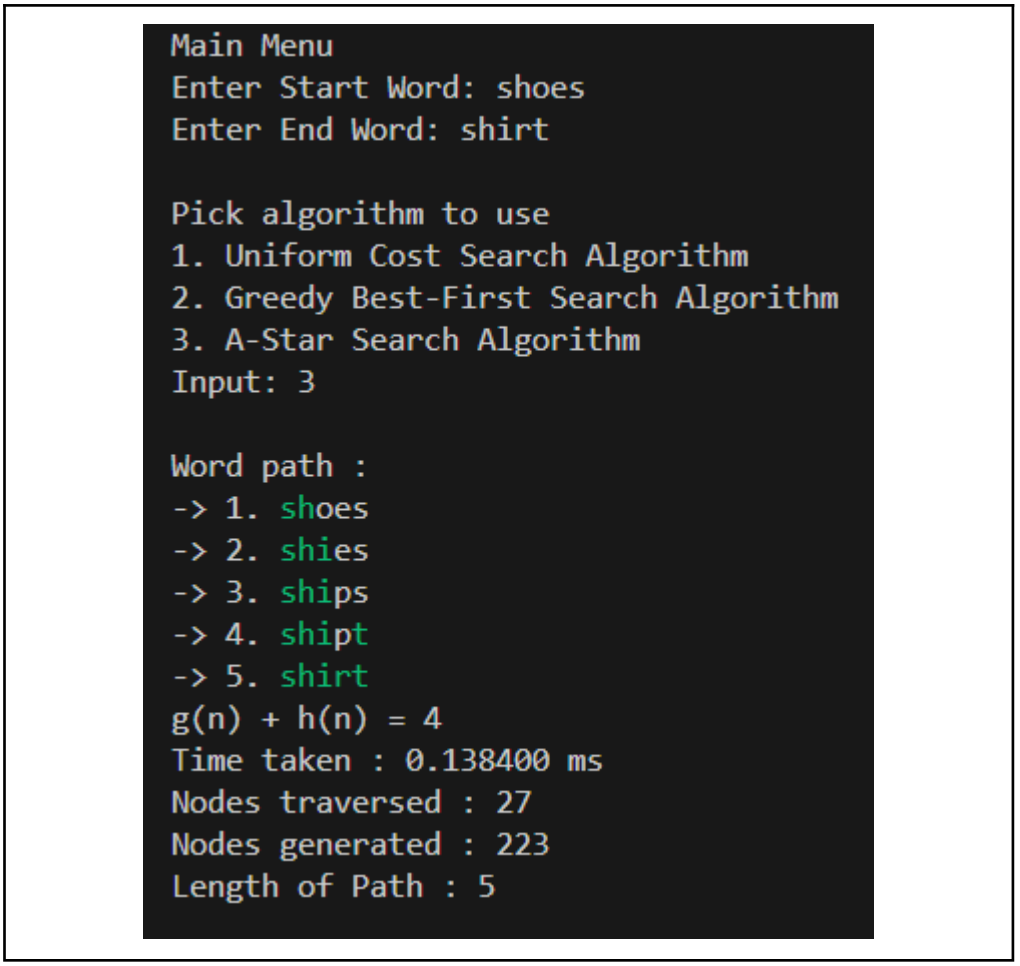
UCS



Greedy Best First Search


Main Menu
Enter Start Word: shoes
Enter End Word: shirt
Pick algorithm to use
1. Uniform Cost Search Algorithm
2. Greedy Best-First Search Algorithm
3. A-Star Search Algorithm
Input: 2
Word path :
-> 1. shoes
-> 2. shies
-> 3. shims
-> 4. shins
-> 5. ships
-> 6. shipt
-> 7. shirt
h(n) = 0
Time taken : 0.035100 ms
Nodes traversed : 7
Nodes generated : 60
Length of Path : 7

A*



4.5 Analisis Test Case dengan *data.txt*

Berikut adalah ringkasan mengenai hasil test case :

Tabel 4.5.1 Tabel hasil *test case*.

No	UCS				*Greedy Best First Search*				A*			
	Waktu eksekusi (Time Taken in ms)	Banyaknya node yang dikunjungi (Nodes Traversed)	Memori (Nodes Generated)	optimalitas (Length of path)	Waktu eksekusi (Time Taken in ms)	Banyaknya node yang dikunjungi (Nodes Traversed)	Memori (Nodes Generated)	optimalitas (Length of path)	Waktu eksekusi (Time Taken in ms)	Banyaknya node yang dikunjungi (Nodes Traversed)	Memori (Nodes Generated)	optimalitas (Length of path)
1.	6.1882	524	6288	5	0.1016	5	66	5	0.248	10	151	5
2.	0.1596	19	363	3	0.0422	3	44	3	0.1049	4	61	3
3.	0.6423	234	1058	5	0.0446	6	48	6	0.2509	13	93	5
4.	5.0729	5208	11576	7	0.1532	22	142	22	0.2182	115	583	7

5.	1.4796	1589	5156	7	0.1012	14	101	14	0.1470	31	198	7
6.	16.8281	32070	40797	12	0.0905	21	125	21	2.5597	2132	7753	12
7.	2.0365	3784	10293	7	0.0590	10	84	10	0.1698	65	497	7
8.	0.7099	880	1771	9	0.0891	21	145	21	0.1595	73	310	9
9.	2.54390	3349	7289	10	0.0441	12	102	12	0.6054	537	1676	10
10.	0.1909	163	872	5	0.0351	7	60	7	0.1384	27	223	5
Avg:	3.58519	4782	8546.3	7	0.07606	12.1	91.7	12.1	0.46018	300.7	1154.5	7

Berdasarkan hasil *test case,* didapat bahwa algoritma *Greedy Best First Search* merupakan algoritma terbaik untuk mencari solusi dengan cepat dan penggunaan memori minimal, hanya saja solusi yang dihasilkan mungkin **bukan solusi optimal** disebabkan oleh sifatnya yang heuristik. Untuk mencari solusi yang optimal (rangkaian kata yang terpendek), algoritma *Uniform Cost Search* (UCS) merupakan algoritma yang tentu akan selalu mendapatkan solusi optimal, hanya saja tidak efisien dengan banyaknya node yang dikunjungi, pemakaian memori, serta waktu eksekusi yang sangat tinggi. Algoritma A* dari *test case* terlihat sebagai algoritma terbaik di antara ketiga algoritma, algoritma A* selalu akan mendapatkan solusi yang optimal sebab heuristik yang *admissible* serta banyaknya node yang dikunjungi, pemakaian memori, serta waktu eksekusi yang rendah dibandingkan dengan algoritma UCS.

Bab V
KESIMPULAN DAN SARAN

5.1 Kesimpulan

Program dibuat penulis untuk memenuhi tugas kecil 3, mata kuliah IF2211, Strategi Algoritma. Program ini bertujuan menemukan solusi permainan *Word Ladder* dengan menggunakan algoritma *Uniform Cost Search* (UCS), *Greedy Best First Search,* dan A*. Melalui percobaan pribadi penulis, didapat bahwa algoritma A* merupakan algoritma yang paling optimal dan efisien untuk menemukan rangkaian kata terpendek dalam permainan *Word Ladder*. Algoritma UCS, walau menghasilkan solusi yang optimal, memakan memori yang banyak serta waktu eksekusi yang lambat. Algoritma *Greedy Best First Search* walau cepat dan memakan sedikit memori, tetapi solusi yang dihasilkan kadang tidak optimal. Akhirnya didapat bahwa algoritma A* merupakan algoritma yang paling tepat dari antara ketiga algoritma tersebut.

5.2 Saran

Tugas Kecil 3 IF2211 Strategi Algoritma Semester II Tahun 2023/2024 memberikan pengalaman baru yang berharga bagi penulis. Berikut beberapa saran yang dapat diberikan kepada pembaca yang ingin menyelesaikan atau mengikuti tugas serupa:

1. Melakukan penelitian yang matang terlebih dahulu sebelum mengembangkan kode untuk mencegah kesalahan saat pengembangan program.
2. Manajemen waktu yang baik dan benar, terutama jika sebagian besar waktu digunakan untuk tugas - tugas yang lain.

Bab VI
LAMPIRAN

6.1 *Link Repository*

https://github.com/MarvelPangondian/Tucil3_13522075

6.2 Tabel *Checklist*

Status : *Completed*

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	✓	