**Tugas Kecil 2 IF2211 Strategi Algoritma**

**Semester II tahun 2023/2024**

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

Disusun oleh:

Rayhan Ridhar Rahman (13522160)

**PROGRAM STUDI TEKNIK INFORMATIKA SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA INSTITUT TEKNOLOGI BANDUNG 2024**

# DAFTAR ISI

# BAB 1 DESKRIPSI TUGAS

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.

A screenshot of a game

Description automatically generated

**Gambar 1.** Ilustrasi dan Peraturan Permainan Word Ladder

(Sumber: <https://wordwormdormdork.com/)>

# BAB 2 TEORI SINGKAT

## 2.1 Algoritma Uniform-Cost Search

BFS dan IDS dapat digunakan untuk mencari solusi dengan jumlah node yang terlewati paling minimum, bukan jarak terpendek. Algoritma uniform-cost search ini akan menelusuri graf dari akar sampai tujuan dengan mengamati jarak yang telah ditempuh sampai node tersebut. Pada awalnya algoritma ini akan terus mencari pada graf node selanjutnya melalui node aktif dengan jarak terpendek, kemudian ketika mendapatkan solusi, solusi tersebut disimpan sementara. Lalu akan dilakukan eksplorasi lagi terhadap node-node aktif yang belum ditelusuri. Jika jarak yang ditempuh melewati jarak solusi sementara, maka node tersebut menjadi dead end. Jika menemukan solusi lain dengan jarak lebih rendah, solusi sementara diganti dengan solusi ersebut. Hal ini dilakukan sampai tidak terdapat lagi node yang dapat ditelusuri. Sehingga bisa dipastikan algoritma ini dapat mencari solusi dengan jarak terendah. Tetapi karena hal tersebut juga, algoritma ini memiliki kompleksitas waktu dan kompleksitas ruang dengan b sebagai banyak cabang dari suatu akar dan d adalah kedalaman solusi.

## 2.2 Algortima Greedy Best First Search

Algoritma ini termasuk jenis algoritma heuristik yang mengetahui nilai node berdasarkan jaraknya terhadap tujuan. Algoritma ini pada awalnya akan menelusuri node akar. Kemudian dari node akar tersebut, akan dilanjutkan dengan node akar aktif baru dengan jarak ke tujuan yang paling rendah. Jarak ke tujuan ini merupakan hasil dari perhitungan tertentu, pada contohnya dalam peta adalah jarak garis lurus ke tujuan. Kemudian akan menelusuri lagi node dengan nilai terendah. Dilakukan sampai menemukan tujuannya. Algoritma ini memiliki kompleksitas waktu dan kompleksitas ruang dengan b sebagai banyak cabang dari suatu akar, d adalah kedalaman terendah. Dalam kasus rata-rata, algoritma ini bisa lebih cepat daripada UCS apalagi dengan fungsi heuristic yang baik. Tetapi algoritma ini mungkin tidak mengembalikan solusi paling optimal dan dapat terjebak dalam minimal local.

## 2.3 Algoritma A\*

Algoritma ini merupakan gabungan dari kedua lagoritma sebelumnya. Setiap node memiliki nilai jarak yang ditempuh dari awal ditambah dengan jarak untuk pergi ke tujuan. Kemudian sama dengan kedua algoritma sebelumnya ini, akan memprioritaskan untuk menelusui node aktif dengan nilai terendah. Ketika menemukan tujuannya, maka akan langsung berhenti menelusuri graf. Dalam teori, algortima ini komplet dengan solusi yang paling optimal. Memiliki kompleksitas waktu dan kompleksitas ruang dengan b sebagai banyak cabang dari suatu akar, m adalah kedalaman terendah. Algortima ini seharusnya lebih cepat dibandingkan UCS karena akan mencoba untuk langsung menelusuri node terdekat dengan tujuan.

# BAB 3 ANALISIS DAN IMPLEMENTASI ALGORITMA

## 3.1 Analisis Algoritma Uniform-Cost Search

## 3.2 Analisis Algoritma Greedy Best First Search

## 3.3 Analisis Algoritma A\*

## 3.4 Source Code

### 3.4.1 englishWords.java

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.FileNotFoundException;

import java.io.IOException;

import java.util.\*;

public class englishWords {

    static Map<Integer, Set<String>> english\_words = new **HashMap**<>();

    static {

**load\_sowpods**();

    }

    public static void **load\_sowpods**() {

        String filename = "sowpods.txt";

        String filedir = "english\_words";

        try  {

            BufferedReader english\_file = new **BufferedReader**(new **FileReader**("./" + filedir + "/" + filename));

            String word;

            while ((word = english\_file.**readLine**()) != null) {

                // System.out.println(word);

                english\_words.**computeIfAbsent**(word.**length**(), k -> new **HashSet**<String>()).**add**(word);

            }

            english\_file.**close**();

        } catch(FileNotFoundException FNFe){

            FNFe.**printStackTrace**();

        } catch(IOException IOe) {

            IOe.**printStackTrace**();

        }

    }

    public static Set<String> **getDictionary**(int word\_length) {

        return english\_words.**get**(word\_length);

    }

    public static void **printDictionary**() {

        english\_words.**forEach**((key, values) -> {

            values.**forEach**(value -> {System.out.**println**("Key: " + key); System.out.**println**("Value: " + value);});

        });

    }

    // Calculate the difference between 2 word (For heuristic function)

    public static int **charDiff**(String word\_1, String word\_2) {

        if (word\_1.**length**() != word\_2.**length**()) {

            System.out.**println**("\n The strings are not of the same length");

            return -1;

        }

        int diff\_counter = 0;

        for (int i = 0; i < word\_1.**length**(); i++) {

            if (word\_1.**charAt**(i) != word\_2.**charAt**(i)) {

                diff\_counter++;

            }

        }

        return diff\_counter;

    }

    public static boolean **validWords**(String start, String end) {

        if (start.**length**() != end.**length**() || start.**length**() > 15 || start.**length**() < 1 || end.**length**() > 15) {

            return false;

        }

        return (englishWords.**getDictionary**(start.**length**()).**contains**(start) && englishWords.**getDictionary**(end.**length**()).**contains**(end));

    }

}

Kelas ini digunakan sebagai tempat membaca seluruh kata-kata yang tersedia dalam sowpods.txt. Terdapat map English\_words dengan panjang kata sebagai key ke set yang berisi kata-kata dengan Panjang tersebut, supaya saat mencari solusi, tidak harus menelusuri seluruh kata.

### 3.4.2 treeNode.java

public class treeNode {

    String word;

    int cost;

**treeNode**(String word, int cost) {

        this.word = word;

        this.cost = cost;

    }

}

Kelas ini digunakan untuk menyimpan node yang berisi kata dan cost yang merupakan nilai dari f(n) masing-masing algoritma.

### 3.4.3 solver.java

import java.util.LinkedList;

import java.util.List;

import java.util.Map;

import java.util.Set;

abstract class solver {

    protected String start;

    protected String end;

    protected Set<String> used\_dictionary;

    protected List<String> path\_result;

    public void **findSolution**() {}

    public void **printResult**() {

        if (path\_result == null) {

            System.out.**println**(" There are no path found from " + start + " to " + end);

            return;

        }

        System.out.**println**("Here is the path:");

        int i = 0;

        for (; i < path\_result.**size**() - 1; i++) {

            System.out.**printf**("%s -> ", path\_result.**get**(i));

        }

        System.out.**println**(path\_result.**get**(i));

    }

    protected void **constructPath** (Map<String, String> parentMap) {

        List<String> path = new **LinkedList**<>();

        String currentWord = this.end;

        while (currentWord != null) {

            path.**add**(0, currentWord);

            currentWord = parentMap.**get**(currentWord);

        }

        this.path\_result = path;

    }

}

Sebagai abstrak kelas algoritma. Memiliki atribut kata awal, kata tujuan, kamus kata yang digunakan, dan hasil dari pencarian solusi. Seluruh kelas algoritma akan menjadi subclass dari kelas ini dan melakukan override terhadap fungsi findSolution. Memiliki juga printResult untuk menunjukkan hasil dan constructPath untuk membangun ulang solusi.

### 3.4.4 UCS.java

import java.util.\*;

public class UCS extends solver{

    public **UCS**(String startingWord, String goalWord) {

        this.start = startingWord;

        this.end = goalWord;

        this.used\_dictionary = englishWords.**getDictionary**(this.start.**length**());

    }

    public void **findSolution**() {

        PriorityQueue<treeNode> active\_nodes = new **PriorityQueue**<>(Comparator.**comparingInt**(node -> node.cost));

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

        Map<String, String> parentMap = new **HashMap**<>();

        active\_nodes.**add**(new **treeNode**(start, 0));

        while (!active\_nodes.**isEmpty**()) {

            treeNode currentNode = active\_nodes.**poll**();

            String currentWord = currentNode.word;

            int currentCost = currentNode.cost;

            if (currentWord.**equals**(end)) {

**constructPath**(parentMap);

                System.out.**println**("Node yang dikunjungi : " + visited.**size**());

                return;

            }

            if (visited.**contains**(currentWord)) {

                continue;

            }

            visited.**add**(currentWord);

            List<String> adjacent = new **ArrayList**<>();

            for (String candidate : this.used\_dictionary) {

                if (englishWords.**charDiff**(currentWord, candidate) == 1) {

                    adjacent.**add**(candidate);

                }

            }

            for (String nextWord : adjacent) {

                int newCost = currentCost + 1;

                if (!visited.**contains**(nextWord) && !parentMap.**containsKey**(nextWord)) {

                    active\_nodes.**add**(new **treeNode**(nextWord, newCost));

                    parentMap.**put**(nextWord, currentWord);

                }

            }

        }

        System.out.**println**(" No solution found...");

    }

}

Kelas untuk algoritma uniform-cost search. Melakukan extends terhadap kelas abstrak solver. Node diisi dengan cost berupa jarak node tersebut dari start. Dalam implementasinya, karena jarak seluruh word dan hubungannya hanya satu, maka jarak sama saja dengan kedalamannya. Algoritma ini mirip dengan algoritma BFS, karena pada dasarnya UCS akan selalu menelusuri semua node pada kedalaman tertentu dahulu, sehingga solusi pertama yang ditemukan, pasti juga solusi paling optimal.

### 3.4.5 GBFS.java

import java.util.\*;

public class GBFS extends solver{

    public **GBFS**(String startingWord, String goalWord) {

        this.start = startingWord;

        this.end = goalWord;

        this.used\_dictionary = englishWords.**getDictionary**(this.start.**length**());

    }

    @Override

    public void **findSolution**(){

        PriorityQueue<treeNode> active\_nodes = new **PriorityQueue**<>(Comparator.**comparingInt**(node -> node.cost));

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

        Map<String, String> parentMap = new **HashMap**<>();

        active\_nodes.**add**(new **treeNode**(start, englishWords.**charDiff**(start, end)));

        while (!active\_nodes.**isEmpty**()){

            treeNode currentNode = active\_nodes.**poll**();

            String currentWord = currentNode.word;

            if (currentWord.**equals**(end)) {

**constructPath**(parentMap);

                System.out.**println**("Node yang dikunjungi : " + visited.**size**());

                return;

            }

            if (visited.**contains**(currentWord)) {

                continue;

            }

            visited.**add**(currentWord);

            List<String> adjacent = new **ArrayList**<>();

            for (String candidate : this.used\_dictionary) {

                if (englishWords.**charDiff**(currentWord, candidate) == 1) {

                    adjacent.**add**(candidate);

                }

            }

            for (String nextWord : adjacent) {

                if (!visited.**contains**(nextWord) && !parentMap.**containsKey**(nextWord)) {

                    active\_nodes.**add**(new **treeNode**(nextWord, englishWords.**charDiff**(nextWord, end)));

                    parentMap.**put**(nextWord, currentWord);

                }

            }

        }

        System.out.**println**(" No solution found...");

    }

}

Kelas untuk algoritma greedy best first search. Melakukan extends terhadap kelas abstrak solver dan melakukan override terhadap fungsi findSolution. Node diisi dengan cost berupa estimasi jarak node ke tujuannya berdasarkan *hamming distance* yang berupa banyaknya perbedaan karakter di antara kata tersebut. Perlu diberi peringatan, dalam implementasinya, akan melakukan ekspansi dengan cost terendah sehingga backtrack mungkin terjadi.

### 3.4.6 Source Code A\*

import java.util.\*;

class ASTAR extends solver {

    public **ASTAR**(String startingWord, String goalWord) {

        this.start = startingWord;

        this.end = goalWord;

        this.used\_dictionary = englishWords.**getDictionary**(this.start.**length**());

    }

    @Override

    public void **findSolution**() {

        PriorityQueue<treeNode> active\_nodes = new **PriorityQueue**<>(Comparator.**comparingInt**(node -> node.cost));

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

        Map<String, String> parentMap = new **HashMap**<>();

        active\_nodes.**add**(new **treeNode**(start, englishWords.**charDiff**(start, end)));

        while (!active\_nodes.**isEmpty**()){

            treeNode currentNode = active\_nodes.**poll**();

            String currentWord = currentNode.word;

            int distance\_from\_start = currentNode.cost - englishWords.**charDiff**(currentWord, end);

            if (currentWord.**equals**(end)) {

**constructPath**(parentMap);

                System.out.**println**("Node yang dilewati : " + visited.**size**());

                return;

            }

            if (visited.**contains**(currentWord)) {

                continue;

            }

            visited.**add**(currentWord);

            List<String> adjacent = new **ArrayList**<>();

            for (String candidate : this.used\_dictionary) {

                if (englishWords.**charDiff**(currentWord, candidate) == 1) {

                    adjacent.**add**(candidate);

                }

            }

            for (String nextWord : adjacent) {

                int new\_cost = distance\_from\_start + 1 + englishWords.**charDiff**(nextWord, end);

                if (!visited.**contains**(nextWord) && !parentMap.**containsKey**(nextWord)) {

                    active\_nodes.**add**(new **treeNode**(nextWord, new\_cost));

                    parentMap.**put**(nextWord, currentWord);

                }

            }

        }

        System.out.**println**(" No solution found...");

    }

}

Kelas untuk algoritma A\* Melakukan extends terhadap kelas abstrak solver dan melakukan override terhadap fungsi findSolution. Node diisi dengan cost berupa estimasi jarak node ke tujuannya berdasarkan banyaknya karakter yang berbeda.

### 3.4.7 wordladder.java

import java.util.InputMismatchException;

import java.util.Scanner;

public class wordladder {

    static Scanner sc = new **Scanner**(System.in);

    public static void **main**(String args[]) {

**printWelcome**();

        System.out.**print**(" Start word: ");

        String start\_string = sc.**nextLine**().**trim**().**toUpperCase**();

        System.out.**print**(" Target word: ");

        String end\_string = sc.**nextLine**().**trim**().**toUpperCase**();

        long startTime = System.**currentTimeMillis**();

        if (englishWords.**validWords**(start\_string, end\_string)) {

            int method\_select = **methodSelector**();

            startTime = System.**currentTimeMillis**();

            solver wordLadderSolver;

            switch (method\_select) {

                case 1:

                    System.out.**println**("+----------------------------------------------------------+");

                    System.out.**println**("| Solving using Uniform-Cost Search                        |");

                    System.out.**println**("+----------------------------------------------------------+");

                    wordLadderSolver = new **UCS**(start\_string, end\_string);

                    break;

                case 2:

                    System.out.**println**("+----------------------------------------------------------+");

                    System.out.**println**("| Solving using Greedy Best First Search                   |");

                    System.out.**println**("+----------------------------------------------------------+");

                    wordLadderSolver = new **GBFS**(start\_string, end\_string);

                    break;

                default:

                    System.out.**println**("+----------------------------------------------------------+");

                    System.out.**println**("| Solving using A\*                                         |");

                    System.out.**println**("+----------------------------------------------------------+");

                    wordLadderSolver = new **ASTAR**(start\_string, end\_string);

                    break;

            }

            wordLadderSolver.**findSolution**();

            wordLadderSolver.**printResult**();

        } else if (start\_string.**equals**("CATHY") || end\_string.**equals**("CATHY")) {

            System.out.**println**(" Who is \*\*\*\*\*?");

        } else if (start\_string.**equals**("CATHERINE") || end\_string.**equals**("CATHERINE")) {

            System.out.**println**(" Who is \*\*\*\*\*\*\*\*\*?");

        }

        else {

            System.out.**println**(" The strings you have chosen are not eligible");

        }

        long elapsed\_time = System.**currentTimeMillis**() - startTime;

        System.out.**println**("\n</> The program took " + elapsed\_time + " ms");

        sc.**close**();

    }

    private static void **printWelcome**() {

        System.out.**println**("+----------------------------------------------------------+");

        System.out.**println**("| Welcome to word ladder solver (Using SOWPODS dictionary) |");

        System.out.**println**("+----------------------------------------------------------+");

    }

    private static void **printSelection**() {

        System.out.**println**("+--------------------- SELECT METHODS ---------------------+");

        System.out.**println**("| 1. Uniform-Cost Search algorithm                         |");

        System.out.**println**("| 2. Greedy Best First Search algorithm                    |");

        System.out.**println**("| 3. A\* algorithm                                          |");

        System.out.**println**("+----------------------------------------------------------+");

    }

    private static int **methodSelector**() {

        int input = 0;

**printSelection**();

        while (input < 1 || input > 3) {

            System.out.**print**(" Please select the method's number: ");

            try {

                input = sc.**nextInt**();

                if (input < 1 || input > 3) {

                    System.out.**println**("\n There's no such option, please try again!\n");

                }

            } catch (InputMismatchException e) {

                System.out.**println**("\n Please input an integer!\n");

                sc.**next**();

            }

        }

    return input;

    }

}

Kelas utama untuk menjalankan program. Menerima input dan sebagainya, walaupun tidak menyimpan data apa-apa. Pada fungsi main terdapat switch yang membuat kelas dari salah satu jenis algoritma. Terdapat juga fungsi untuk memperindah CLI.

# BAB 4 UJI COBA

## 4.1 Tata Cara Penggunaan Program

## 4.2 Hasil Uji Coba

## 4.3 Analisis Uji Coba

# BAB 5 PENUTUP

## 5.1 Kesimpulan

## 5.2 Saran

# DAFTAR PUSTAKA

Munir, Rinaldi. 2020. “Penentuan rute (Route/Path Planning) - Bagian 1”. Diakses pada 5 Mei 2024. <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf>

Munir, Rinaldi. 2020. “Penentuan rute (Route/Path Planning) - Bagian 1”. Diakses pada 5 Mei 2024. <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf>

# LAMPIRAN

**Pranala repository GitHub**:

<https://github.com/Rinnearu/Tucil3_13522160>

**Checklist:**

|  |  |  |
| --- | --- | --- |
| **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 |  | **✓** |