

LAPORAN TUGAS KECIL III

IF2211 Strategi Algoritma

Penyelesaian Permainan Word Ladder Menggunakan Algoritma

UCS, Greedy Best First Search, dan A*



Disusun oleh:

Dhidit Abdi Aziz (13522040)

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

Jl. Ganesha 10, Bandung 40132

Daftar Isi

Daftar Isi.....	2
1 Deskripsi Permasalahan.....	3
2 Dasar Teori.....	4
2.1 Dasar Teori.....	4
2.2 Algoritma UCS.....	4
2.2 Algoritma Greedy BFS.....	5
2.3 Algoritma A*.....	6
3 Analisis Pemecahan Masalah.....	7
3.1 Representasi Graf pada Pemecahan Masalah.....	7
3.2 Pemetaan Masalah dengan Algoritma UCS.....	7
3.3 Pemetaan Masalah dengan Algoritma Greedy BFS.....	8
3.4 Pemetaan Masalah dengan Algoritma A*.....	9
4 Implementasi dan Pengujian.....	11
4.1 Kelas Astar.....	11
4.2 Kelas Greedy BFS.....	12
4.3 Kelas Kamus.....	13
4.4 Kelas Result.....	15
4.5 Kelas Simpul.....	15
4.6 Kelas SimpulStar.....	17
4.7 Kelas Solver.....	18
4.8 Kelas UCS.....	20
4.2 Tata Cara Penggunaan Program.....	21
4.3 Pengujian.....	22
4.4 Analisis Hasil Pengujian.....	25
5 Kesimpulan.....	27
6 Lampiran.....	28
5.1 Tautan Repository.....	28
5.2 Checklist Program.....	28
5.3 Penjelasan Implementasi Bonus.....	28
7 Daftar Pustaka.....	29

1 Deskripsi Permasalahan

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

How To Play

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

Rules

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

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

Example

E	A	S	T
---	---	---	---

EAST is the start word, WEST is the end word

V	A	S	T
---	---	---	---

We changed **E** to **V** to make **VAST**

V	E	S	T
---	---	---	---

We changed **A** to **E** to make **VEST**

W	E	S	T
---	---	---	---

And we changed **V** to **W** to make **WEST**

W	E	S	T
---	---	---	---

Done!

Privacy

Gambar 1. Ilustrasi dan Peraturan Permainan Word Ladder

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

2 Dasar Teori

2.1 Dasar Teori

Konsep yang mendasari program ini adalah penjelajahan atau traversal graf. Artinya, simpul-simpul dengan cara yang sistematis yaitu dengan algoritma. Algoritma yang digunakan pada pencarian rute dalam permainan Word Ladder

adalah algoritma pencarian solusi berbasis graf tanpa informasi (uninformed, blind search), yaitu Uniform Cost Search (UCS). Sebagai pembandingan, juga digunakan algoritma pencarian solusi berbasis informasi, yaitu Greedy Breadth First Search, dan A*

2.2 Algoritma UCS

Uniform-Cost Search merupakan algoritma pencarian tanpa informasi (uninformed search) yang menggunakan biaya kumulatif terendah untuk menemukan jalur dari node sumber ke node tujuan. Algoritma ini beroperasi di sekitar ruang pencarian berbobot terarah untuk berpindah dari node awal ke salah satu node akhir dengan biaya akumulasi minimum. Algoritma Uniform-Cost Search masuk dalam algoritma pencarian uninformed search atau blind search karena bekerja dengan cara brute force, yaitu tidak mempertimbangkan keadaan node atau ruang pencarian. Biasanya algoritma Uniform-Cost Search diimplementasikan dengan menggunakan priority queue di mana prioritasnya adalah menurunkan biaya kumulatif. Perhitungan biaya kumulatif tersebut menggunakan fungsi:

$$f(h) = g(h)$$

Dimana $g(h)$ adalah jarak dari simpul h ke akar

Berikut adalah cara kerja algoritma uniform-cost search:

1. Masukkan node root ke dalam priority queue
2. Ulangi langkah berikut saat antrian (queue) tidak kosong:
 - Hapus elemen dengan prioritas tertinggi

- Jika node yang dihapus adalah node tujuan, cetak total biaya (cost) dan hentikan algoritma
- Jika tidak, enqueue semua child dari node saat ini ke priority queue, dengan biaya kumulatifnya dari root sebagai prioritas

Di sini node root adalah node awal untuk jalur pencarian, dan priority queue tetap untuk mempertahankan jalur dengan biaya paling rendah untuk dipilih pada traversal berikutnya. Jika 2 jalur memiliki biaya traversal yang sama, node diurutkan berdasarkan abjad. (Trivusi, 2022)

2.2 Algoritma Greedy BFS

Greedy BFS merupakan jenis algoritma BFS yang paling sederhana (Mahmud et al., 2012). Sama seperti algoritma sebelumnya, greedy BFS juga menggunakan struktur data queue tetapi yang dijadikan prioritas bukanlah jarak yang ditempuh. Sebagai gantinya, kita perlu membuat heuristic untuk menentukan apakah suatu node lebih baik dibanding yang lain untuk mencapai node tujuan. Heuristic sendiri ialah metode pemecahan masalah yang menggunakan jalan pintas untuk menghasilkan solusi yang cukup baik. Fungsi heuristic sebenarnya dapat sangat bervariasi tergantung bagaimana sebuah graf diatur (Lawrence & Bulitko, 2012). Namun dalam kasus ini, graf dibuat untuk melakukan pathfinding pada grid dua dimensi sehingga fungsi heuristic akan digunakan untuk mengukur jarak di antara suatu node (bukan start node) dengan node tujuan sehingga greedy BFS akan mempertimbangkan node yang lebih dekat dengan tujuan menjadi kandidat yang lebih baik untuk dieksplor. (Sugianti et al., 2020)

Algoritma greedy BFS menggunakan fungsi evaluasi yang meniadakan perkiraan biaya G . G di sini adalah jarak node awal ke suatu node. Sehingga fungsi evaluasi yang digunakan adalah sebagai berikut:

$$f(n) = h(n)$$

Dimana $h(n)$ adalah perkiraan jarak (vektor) dari simpul n ke lokasi tujuan

2.3 Algoritma A*

Algoritma A* adalah salah satu metode pathfinding paling populer yang banyak digunakan dan telah terjamin optimalisasinya (Hart et al., 1968). Namun, algoritma ini dapat dikatakan boros sumber daya (Yiu et al., 2018). A* sendiri merupakan perpaduan dari algoritma BFS dan Dijkstra (Gonçalves et al., 2019). Algoritma ini menggunakan pencarian BFS untuk menemukan jalur dengan biaya terendah dari suatu node ke node awal. Sedangkan fungsi heuristic yang mirip dengan Dijkstra digunakan untuk mencari jarak paling kecil dari suatu node ke node tujuan. Oleh karena itu, A* menggunakan fungsi evaluasi yang terdiri dari dua bagian, yaitu heuristic $H(n)$ dan perkiraan biaya $G(n)$, di mana

$$F(n) = G(n) + H(n) \quad (3)$$

$$fCost = gCost + hCost \quad (4)$$

gCost di sini berarti jarak dari node awal sedangkan hCost ialah fungsi heuristic yang berarti jarak dari node tujuan. Dalam percobaan ini fungsi heuristic yang digunakan ialah manhattan distance. Perlu diingat bahwa fungsi heuristic hanya memberi nilai estimasi karena fungsi ini tidak mempertimbangkan blocked node. Jarak sebenarnya bisa saja lebih besar dari heuristic yang telah dihitung. Sehingga dapat disimpulkan bahwa fCost merupakan jarak pasti dari node awal ditambah dengan perkiraan jarak menuju node tujuan. Hasilnya akan digunakan untuk memilih node berikutnya yang masuk ke dalam queue. (Sugianti et al., 2020)

3 Analisis Pemecahan Masalah

3.1 Representasi Graf pada Pemecahan Masalah

Sebab aturan dalam permainan Word Ladder yang hanya memperbolehkan perubahan satu huruf di setiap iterasi terhadap susunan sebelumnya hingga tercapai kata tujuan, akan terdapat berbagai kemungkinan susunan kata yang sesuai dengan masukan. Dengan demikian, proses penelusuran kata tersebut direpresentasikan sebagai suatu graf. Simpul pada graf mewakili setiap kata yang dikunjungi pada langkah permainan. Sedangkan sisi pada graf mewakili langkah yang diambil pada setiap iterasi untuk mengubah satu huruf dari simpul sebelumnya untuk membentuk simpul selanjutnya.

3.2 Pemetaan Masalah dengan Algoritma UCS

Fungsi evaluasi *cost* dari suatu sisi ditentukan dengan fungsi

$$f(n) = g(n)$$

Dimana $g(n)$ adalah berapa banyak huruf yang berubah dari satu simpul ke simpul lainnya.

Berikut ini langkah-langkah pencarian solusi dengan algoritma UCS

1. Inisialisasi sebuah Priority Queue yang akan menyimpan simpul-simpul yang akan dieksplorasi
2. Inisialisasi sebuah HashSet untuk menyimpan kata kata yang telah dieksplorasi sehingga tidak dikunjungi lebih dari sekali
3. Inisialisasi sebuah HashMap untuk menyimpan kata kata pada kamus
4. Sebagai awalan, kata start dijadikan simpul dan dimasukkan ke Priority Queue
5. Iterasi akan dilakukan selama Priority Queue tidak kosong
6. Pada setiap iterasi, ambil simpul dengan dengan cost terendah dari Priority Queue
7. Apabila kata pada simpul yang diambil sudah sama dengan kata end, maka pencarian selesai

8. Apabila kata belum sama dengan kata end, maka akan dicari child dari kata tersebut, dengan mencoba berbagai macam kombinasi perubahan satu huruf yang mungkin pada kata tersebut.
9. Setiap kombinasi kata yang valid (terdapat pada HashMap) dan belum pernah dikunjungi (tidak ada pada HashSet), maka akan dihitung costnya dan dimasukkan ke dalam Priority Queue
10. Pada setiap simpul yang telah dibuat, akan dimasukkan juga ke dalam HashSet agar tidak perlu dikunjungi kembali

3.3 Pemetaan Masalah dengan Algoritma Greedy BFS

Fungsi evaluasi *cost* dari suatu sisi ditentukan dengan fungsi

$$f(n) = h(n)$$

Dimana $h(n)$ adalah berapa banyak huruf yang diperlukan untuk berubah agar dapat mencapai kata end.

Berikut ini langkah-langkah pencarian solusi dengan algoritma Greedy BFS

1. Inisialisasi sebuah HashSet untuk menyimpan kata kata yang telah dieksplorasi sehingga tidak dikunjungi lebih dari sekali
2. Inisialisasi sebuah HashMap untuk menyimpan kata kata pada kamus
3. Sebagai awalan, kata start dijadikan simpul
4. Iterasi akan dilakukan sampai terjadi kasus yang memanggil break ataupun return
5. Pada setiap iterasi, ambil simpul yang terakhir dibuat dan cek kata yang dikandungnya
6. Apabila kata pada simpul yang diambil sudah sama dengan kata end, maka pencarian selesai
7. Apabila kata belum sama dengan kata end, maka akan dicari child dari kata tersebut, dengan mencoba berbagai macam kombinasi perubahan satu huruf yang mungkin pada kata tersebut.
8. Pada percabangan ini, juga dilakukan inisialisasi sebuah String bernama Greedy String dengan null untuk mengecek apakah pencarian masih dapat dilanjutkan

9. Setiap kombinasi kata yang valid (terdapat pada HashMap) dan belum pernah dikunjungi (tidak ada pada HashSet), maka akan dihitung costnya dan disimpan sementara di Greedy String
10. Akan dicari child yang memiliki cost paling minimum (dan abjad ter-awal)
11. Apabila Greedy Word masih berisi null, artinya tidak ditemukan child yang belum pernah dikunjungi atau tidak ada lagi kombinasi kata yang valid di kamus. Artinya, pencarian tidak dapat dilanjutkan sehingga dilakukan break
12. Apabila Greedy Word sudah bukan berisi null, artinya proses pencarian child dengan cost terminimum berhasil, maka kata tersebut dijadikan simpul baru untuk dasar perhitungan perulangan selanjutnya

3.4 Pemetaan Masalah dengan Algoritma A*

Fungsi evaluasi *cost* dari suatu sisi ditentukan dengan fungsi

$$f(n) = g(n) + h(n)$$

Dimana $g(n)$ adalah berapa banyak huruf yang telah berubah sepanjang perjalanan dari kata start ke suatu kata. Sedangkan, $h(n)$ adalah berapa banyak huruf yang diperlukan untuk berubah agar dapat mencapai kata end.

Berikut ini langkah-langkah pencarian solusi dengan algoritma A*

1. Inisialisasi sebuah Priority Queue untuk menyimpan simpul simpul kata yang akan diekspan
2. Inisialisasi sebuah HashSet untuk menyimpan kata kata yang telah dieksplorasi sehingga tidak dikunjungi lebih dari sekali
3. Inisialisasi sebuah HashMap untuk menyimpan kata kata pada kamus
4. Sebagai awalan, kata start dijadikan simpul
5. Iterasi akan dilakukan selama Priority Queue tidak kosong
6. Pada setiap iterasi, ambil simpul dengan dengan cost terendah dari Priority Queue
7. Apabila kata pada simpul yang diambil sudah sama dengan kata end, maka pencarian selesai

8. Apabila kata belum sama dengan kata end, maka akan dicari child dari kata tersebut, dengan mencoba berbagai macam kombinasi perubahan satu huruf yang mungkin pada kata tersebut.
9. Setiap kombinasi kata yang valid (terdapat pada HashMap) dan belum pernah dikunjungi (tidak ada pada HashSet), maka akan dihitung costnya dan dimasukkan ke dalam Priority Queue
10. Pada setiap simpul yang telah dibuat, akan dimasukkan juga ke dalam HashSet agar tidak perlu dikunjungi kembali

4 Implementasi dan Pengujian

4.1 Kelas Astar

Bertanggung jawab untuk melakukan pencarian berdasarkan algoritma A*

```
import java.util.HashSet;
import java.util.PriorityQueue;

public class Astar {
    public static Result solve(String startword, String endword, Kamus
    kamus) {
        PriorityQueue<SimpulStar> q = new PriorityQueue<>(); //Penyimpanan
        simpul yang akan diekspan
        HashSet<String> visited = new HashSet<>(); //Agar node yang sama
        tidak dikunjungi lagi
        q.add(new SimpulStar(startword));
        int count = 0; //Menghitung node yang dikunjungi

        while(!q.isEmpty()){
            SimpulStar currSimpul = q.remove(); //Ambil simpul dengan cost
            terendah

            String currWord = currSimpul.getWord();

            count++;
            if(currWord.equals(endword)){
                //Apabila sudah sama dengan endword, maka iterasi
                dihentikan

                return new Result(currSimpul.findPath(), count);
            }else{
                for(int i = 0; i<currWord.length(); i++){
                    for(char c = 'A'; c<= 'Z'; c++){ //Dicoba berbagai
                    kombinasi huruf untuk mencari child

                        String tetangga = currWord.substring(0, i) + c +
                        currWord.substring(i+1);
```

```

        if(kamus.isValid(tetangga) &&
!visited.contains(tetangga)){ //Apabila terdapat dikamus dan belum pernah
dikunjungi, maka child dibuat menjadi simpul
            visited.add(tetangga);
            q.add(new SimpulStar(tetangga,
currSimpul.getG() + 1, Simpul.Distance(tetangga, endword), currSimpul));
        }
    }
}
}

return new Result(null, count);

}
}

```

4.2 Kelas Greedy BFS

Bertanggung jawab untuk melakukan pencarian berdasarkan algoritma Greedy BFS

```

import java.util.HashSet;

public class GreedyBFS {
    public static Result solve(String startword, String endword, Kamus
kamus){
        HashSet<String> visited = new HashSet<>(); //Agar simpul yang sama
tidak dikunjungi dua kali
        Simpul currSimpul = new Simpul(startword);
        int count = 0;
        boolean found = false;

        while(!found){
            String greedString;
            int minCost;
            String currWord = currSimpul.getWord();
            count++;
            if(currWord.equals(endword)){ //Apabila sudah sama dengan end
word, maka iterasi selesai

```

```

        System.out.println("Reach result!");
        return new Result(currSimpul.findPath(), count);
    }else{
        greedString = null;
        minCost = 9999;
        for(int i = 0; i<currWord.length(); i++){//Pencarian child
dengan mencoba berbagai kombinasi kata
            for(char c = 'A'; c<= 'Z'; c++){
                String tetangga = currWord.substring(0, i) + c +
currWord.substring(i+1);
                if(kamus.isValid(tetangga) &&
!visited.contains(tetangga)){
                    //Apabila terdapat di kamus dan belum pernah
dikunjungi, maka dapat menjadi alternatif solusi
                    visited.add(tetangga);
                    int cost = Simpul.Distance(tetangga, endword);
                    if(cost<minCost){
                        //Child yang dipilih hanyalah yang
memiliki cost terendah
                        minCost = cost;
                        greedString = tetangga;
                    }
                }
            }
        }

        if (greedString == null){
            //Apabila tidak ada child yang mampu menggantikan
inisialisasi variabel, artinya pencarian tidak dapat dilanjutkan
            break;
        }
        else{
            currSimpul = new Simpul(greedString, minCost,
currSimpul);
        }
    }
    return new Result(null, count);
}
}

```

4.3 Kelas Kamus

Menyimpan kata-kata dengan panjang yang seragam dengan bentuk HashSet

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.HashSet;

public class Kamus {
    private HashSet<String> dictionary = new HashSet<>();
    FileReader f;

    public Kamus(String filepath){
        try{
            f = new FileReader(filepath);
        } catch (FileNotFoundException e){
            System.out.println("File not found: " + filepath);
        }

        BufferedReader b = new BufferedReader(f);
        String line = null;
        try{
            line = b.readLine();
        } catch (Exception e){
            System.out.println("Error while reading file");
        }

        while(line != null){
            this.dictionary.add(line);

            try{
                line = b.readLine();
            } catch (Exception e){
                System.out.println("Error while reading file");
            }
        }

        try{
            b.close();
        }
```

```

        } catch (Exception e) {
            System.out.println("Error when closing the file reader");
        }

    }

    public boolean isValid(String word) {
        return this.dictionary.contains(word);
    }
}

```

4.4 Kelas Result

Menyimpan hasil pencarian algoritma. Merupakan struct yang terdiri atas path hasil pencarian dan jumlah node yang dikunjungi

```

import java.util.ArrayList;

public class Result {
    private ArrayList<String> path;
    private int count;

    public Result(ArrayList<String> path, int count) {
        this.path = path;
        this.count = count;
    }

    public ArrayList<String> getPath() {
        return this.path;
    }

    public int getCount() {
        return this.count;
    }
}

```

4.5 Kelas Simpul

Merupakan struct untuk menyimpan kata beserta costnya

```

import java.util.ArrayList;

public class Simpul implements Comparable<Simpul>{
    private int cost;
    private String word;
    private Simpul parent;

    public Simpul(String word){
        this.word = word;
        this.cost = 0;
        this.parent = null;
    }

    public Simpul(String word, int cost, Simpul parent){
        this.word = word;
        this.cost = cost;
        this.parent = parent;
    }

    public ArrayList<String> findPath(){
        ArrayList<String> p = new ArrayList<>();
        Simpul s = this.parent;
        p.addFirst(this.word);
        while(s!=null){
            p.addFirst(s.word);
            s = s.getParent();
        }

        return p;
    }

    public int getCost(){
        return this.cost;
    }

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

```



```

public Simpul getParent() {
    return parent;
}

@Override
public int compareTo(Simpul other) {
    return Integer.compare(this.getCost(), other.getCost());
}

public static int Distance(String p , String q) {
    int d = 0;
    for(int i = 0; i<p.length(); i++){
        if(p.charAt(i) != q.charAt(i)){
            d++;
        }
    }
    return d;
}
}

```

4.6 Kelas SimpulStar

Merupakan kelas turunan dari kelas simpul untuk menyimpan kata pada algoritma A*. Perbedaannya dengan kelas parentnya, kelas Simpul Star memisahkan antara $g(n)$ dengan $h(n)$ untuk kemudahan perhitungan cost.

```

public class SimpulStar extends Simpul {
    private int g;
    private int h;

    public SimpulStar(String word) {
        super(word);
        this.g = 0;
        this.h = 0;
    }

    public SimpulStar(String word, int g, int h, Simpul parent) {
        super(word, g+h, parent);
        this.g = g;
        this.h = h;
    }
}

```

```

    public int getG(){
        return this.g;
    }

    public int getH(){
        return this.h;
    }
}

```

4.7 Kelas Solver

Merupakan main dari keseluruhan fungsi, bertanggung jawab untuk menghandle input dan mengeluarkan output

```

import java.util.Scanner;
public class Solver{
    public static void main(String[] args){
        //Inisialisasi variabel
        Scanner scanner = new Scanner(System.in);
        Boolean inputvalid = false;
        Boolean stopvalid = false;
        Boolean algorithmValid = false;
        Boolean stopGame = false;
        Integer algorithm = null;
        String startword = null;
        String endword = null;
        String filepath = null;
        Integer play = null;
        Kamus kamus = null;
        Result res = null;
        long startTime = 0;
        long endTime = 0;

        //Menerima inputan kata
        while(!inputvalid){
            System.out.print("Start word: ");
            startword = scanner.nextLine();
            System.out.print("End word: ");

```

```

        endword = scanner.nextLine();

        if(startword.length() != endword.length()){
            //Apabila panjang dua kata berbeda, maka tidak valid
            System.out.println("Length of startword and endword is
different!");
        }else{
            //Pembuatan dictionary berdasarkan panjang kata
            filepath = "resources\\" + startword.length() +
"-letter-words.txt";
            kamus = new Kamus(filepath);
            startword = startword.toUpperCase();
            endword = endword.toUpperCase();

            if(!kamus.isValid(startword) || !kamus.isValid(endword)){
                //Apabila setidaknya salah satu dari kata inputan
tidak terdapat di kamus, maka tidak valid
                System.out.println("Your words isn't valid! Try again
please!");
            }
            else{
                //Stop meminta input
                inputvalid = true;
            }
        }
    }

    //Menerima jenis algoritma
    while(!algorithmValid){
        System.out.print("Choose an algorithm:\n1. UCS\n2. Greedy
BFS\n3. A*\nYour choice (1/2/3): ");
        algorithm = scanner.nextInt();

        if(algorithm == 1){
            startTime = System.nanoTime();
            res = UCS.solve(startword, endword, kamus);
            endTime = System.nanoTime();
            algorithmValid = true;
        }
        else if(algorithm == 2){

```

```

        startTime = System.nanoTime();
        res = GreedyBFS.solve(startword, endword, kamus);
        endTime = System.nanoTime();
        algorithmValid = true;
    }
    else if(algorithm == 3){
        startTime = System.nanoTime();
        res = Astar.solve(startword, endword, kamus);
        endTime = System.nanoTime();
        algorithmValid = true;
    }
    else{
        System.out.println("Your input is invalid! Try again
please!");
    }
}

System.out.println("Node visited: " + res.getCount());
double miliseconds = (double) (endTime - startTime)/1000000;
System.out.println("Time: " + miliseconds + "ms");
if(res.getPath() == null){
    System.out.println("Path: Not Found");
}else{
    System.out.println("Path:");
    for(int i = 0; i<res.getPath().size(); i++){
        System.out.println((i+1) + ". " + res.getPath().get(i));
    }
}

scanner.close();
}
}

```

4.8 Kelas UCS

Melakukan pencarian solusi dengan algoritma UCS

```

import java.util.HashSet;
import java.util.PriorityQueue;

```

```

public class UCS {
    public static Result solve(String startword, String endword, Kamus
kamus){
        PriorityQueue<Simpul> q = new PriorityQueue<>();
        HashSet<String> visited = new HashSet<>();
        q.add(new Simpul(startword));
        int count = 0;

        while(!q.isEmpty()){
            Simpul currSimpul = q.remove();
            String currWord = currSimpul.getWord();

            count++;
            if(currWord.equals(endword)){
                System.out.println("Reach result!");
                return new Result(currSimpul.findPath(), count);
            }else{
                for(int i = 0; i<currWord.length(); i++){
                    for(char c = 'A'; c<= 'Z'; c++){
                        String tetangga = currWord.substring(0, i) + c +
currWord.substring(i+1);
                        /*
                        try{
                            Thread.sleep(300);
                        }catch(Exception e){
                            System.out.println("Hehe");
                        }
                        */
                        if(kamus.isValid(tetangga) &&
!visited.contains(tetangga)){
                            //System.out.print(tetangga);
                            visited.add(tetangga);
                            q.add(new Simpul(tetangga,
currSimpul.getCost() + 1, currSimpul));
                        }
                    }
                }
            }
        }
    }
}

```

```

        return new Result(null, count);
    }
}

```

4.2 Tata Cara Penggunaan Program

- Lakukan clone pada repository github program ini
- Akses folder src
- Pada terminal, jalankan: `javac Solver.java`
- Selanjutnya, jalankan: `java Solver`
- Input kata start dan kata end pada terminal
- Input pilihan algoritma pada terminal
- Output hasil akan dikeluarkan secara otomatis

4.3 Pengujian

Testing ke-1: ARC ke CAR		
<p>UCS</p> <pre> Start word: arc End word: car Choose an algorithm: 1. UCS 2. Greedy BFS 3. A* Your choice (1/2/3): 1 Reach result! Node visited: 484 Time: 501.5817ms Path: 1. ARC 2. ARY 3. CRY 4. CAY 5. CAR </pre>	<p>Greedy BFS</p> <pre> Start word: ARC End word: CAR Choose an algorithm: 1. UCS 2. Greedy BFS 3. A* Your choice (1/2/3): 2 Node visited: 2 Time: 27.1989ms Path: Not Found </pre>	<p>A*</p> <pre> Start word: ARC End word: CAR Choose an algorithm: 1. UCS 2. Greedy BFS 3. A* Your choice (1/2/3): 3 Node visited: 16 Time: 36.0414ms Path: 1. ARC 2. ARY 3. CRY 4. CAY 5. CAR </pre>

Testing ke-2: EAST ke WEST		
<p>UCS</p> <pre> Start word: EAST End word: WEST Choose an algorithm: 1. UCS 2. Greedy BFS 3. A* Your choice (1/2/3): 1 Reach result! Node visited: 81 Time: 54.4398ms Path: 1. EAST 2. WAST 3. WEST </pre>	<p>Greedy BFS</p> <pre> Start word: EAST End word: WEST Choose an algorithm: 1. UCS 2. Greedy BFS 3. A* Your choice (1/2/3): 2 Reach result! Node visited: 3 Time: 31.5664ms Path: 1. EAST 2. WAST 3. WEST </pre>	<p>A*</p> <pre> Start word: EAST End word: WEST Choose an algorithm: 1. UCS 2. Greedy BFS 3. A* Your choice (1/2/3): 3 Node visited: 3 Time: 19.3197ms Path: 1. EAST 2. WAST 3. WEST </pre>
Testing ke-3: MOUTH ke GLASS		
<p>UCS</p> <pre> Start word: MOUTH End word: GLASS Choose an algorithm: 1. UCS 2. Greedy BFS 3. A* Your choice (1/2/3): 1 Reach result! Node visited: 6108 Time: 378.7762ms Path: 1. MOUTH 2. SOUTH 3. SOUTS 4. SLUTS 5. SLATS 6. CLATS 7. CLASS 8. GLASS </pre>	<p>Greedy BFS</p> <pre> Start word: MOUTH End word: GLASS Choose an algorithm: 1. UCS 2. Greedy BFS 3. A* Your choice (1/2/3): 2 Node visited: 9 Time: 29.6724ms Path: Not Found </pre>	<p>A*</p> <pre> Start word: MOUTH End word: GLASS Choose an algorithm: 1. UCS 2. Greedy BFS 3. A* Your choice (1/2/3): 3 Node visited: 72 Time: 41.3728ms Path: 1. MOUTH 2. ROUTH 3. ROUTS 4. GOUTS 5. GOATS 6. GOADS 7. GLADS 8. GLASS </pre>
Testing ke-4: BLACK ke GREEN		

<p>UCS</p> <pre> Start word: BLACK End word: GREEN Choose an algorithm: 1. UCS 2. Greedy BFS 3. A* Your choice (1/2/3): 1 Reach result! Node visited: 2716 Time: 283.8814ms Path: 1. BLACK 2. CLACK 3. CLECK 4. CLEEK 5. CREEK 6. GREEK 7. GREEN </pre>	<p>Greedy BFS</p> <pre> Start word: BLACK End word: GREEN Choose an algorithm: 1. UCS 2. Greedy BFS 3. A* Your choice (1/2/3): 2 Node visited: 6 Time: 21.8525ms Path: Not Found </pre>	<p>A*</p> <pre> Start word: BLACK End word: GREEN Choose an algorithm: 1. UCS 2. Greedy BFS 3. A* Your choice (1/2/3): 3 Node visited: 32 Time: 76.4751ms Path: 1. BLACK 2. FLACK 3. FLECK 4. FLEEK 5. GLEEK 6. GREEK 7. GREEN </pre>
Testing ke-5: GREEN ke BLACK		
<p>UCS</p> <pre> Start word: GREEN End word: BLACK Choose an algorithm: 1. UCS 2. Greedy BFS 3. A* Your choice (1/2/3): 1 Reach result! Node visited: 2339 Time: 221.3871ms Path: 1. GREEN 2. GREEK 3. GLEEK 4. FLEEK 5. FLECK 6. FLACK 7. BLACK </pre>	<p>Greedy BFS</p> <pre> Start word: GREEN End word: BLACK Choose an algorithm: 1. UCS 2. Greedy BFS 3. A* Your choice (1/2/3): 2 Reach result! Node visited: 7 Time: 26.6995ms Path: 1. GREEN 2. GREEK 3. GLEEK 4. CLEEK 5. CLECK 6. CLACK 7. BLACK </pre>	<p>A*</p> <pre> Start word: GREEN End word: BLACK Choose an algorithm: 1. UCS 2. Greedy BFS 3. A* Your choice (1/2/3): 3 Node visited: 16 Time: 44.5928ms Path: 1. GREEN 2. GREEK 3. GLEEK 4. CLEEK 5. CLECK 6. CLACK 7. BLACK </pre>

Testing ke-6: WAITING ke UNBAKED

UCS

```

Start word: WAITING
End word: UNBAKED
Choose an algorithm:
1. UCS
2. Greedy BFS
3. A*
Your choice (1/2/3): 1
Reach result!
Node visited: 12201
Time: 903.1167ms
Path:
1. WAITING
2. WASTING
3. PASTING
4. POSTING
5. POSTINS
6. POSTIES
7. POSSIES
8. MOSSIES
9. MOUSTIES
10. MOUSSES
11. POUSSSES
12. PLUSSES
13. PLISSES
14. PRISSES
15. PRESSES
16. PREASES
17. UREASES
18. UNEASES
19. UNCASES
20. UNCASED
21. UNBASED
22. UNBAKED

```

Greedy BF

```

Start word: WAITING
End word: UNBAKED
Choose an algorithm:
1. UCS
2. Greedy BFS
3. A*
Your choice (1/2/3): 2
Node visited: 28
Time: 35.4633ms
Path: Not Found

```

S

A*

```

Start word: WAITING
End word: UNBAKED
Choose an algorithm:
1. UCS
2. Greedy BFS
3. A*
Your choice (1/2/3): 3
Node visited: 10017
Time: 606.1918ms
Path:
1. WAITING
2. WASTING
3. PASTING
4. POSTING
5. POSTINS
6. POSTIES
7. POSSIES
8. MOSSIES
9. MOUSTIES
10. MOUSSES
11. POUSSSES
12. PLUSSES
13. PLISSES
14. PRISSES
15. PRISSED
16. PRESSED
17. PREASED
18. PREASES
19. UREASES
20. UNEASES
21. UNCASES
22. UNCAKES
23. UNCAKED
24. UNBAKED

```

4.4 Analisis Hasil Pengujian

Pada beberapa kasus, algoritma Greedy BFS mampu menghasilkan solusi dengan efisiensi dan efektivitas yang tinggi. Hal ini dapat terlihat dari jumlah node yang dikunjungi paling sedikit, serta waktu eksekusi yang paling cepat. Sayangnya, pada beberapa kasus lainnya, terjadi edge case yaitu kata yang terlanjur dipilih pada pencarian dengan Greedy BFS tidak dapat menuju solusi sama sekali sehingga tidak dihasilkan path apapun.

Algoritma A* memiliki efektivitas di bawah Greedy BFS. Kelebihannya, algoritma ini minim kemungkinan akan terjadi dead end seperti pada Greedy BFS. Sayangnya, terdapat satu kasus yaitu pada testing ke-6 dimana hasil yang dikeluarkan oleh A* bukanlah yang optimal. Dari sini dapat diambil kesimpulan bahwa heuristik yang digunakan oleh penulis masih kurang admissible sehingga hasil yang dikeluarkan kurang akurat

Sementara itu, UCS memerlukan waktu eksekusi yang paling lama dan mengunjungi simpul paling banyak apabila dibandingkan dengan dua algoritma lainnya. Hal ini disebabkan oleh penentuan cost yang kurang merepresentasikan proses keseluruhan. UCS hanya menggunakan $g(n)$ tanpa mempertimbangkan jarak antara simpul ke end word. Akibatnya, penjelajahan yang dilakukan cenderung kurang efektif sehingga waktu yang dibutuhkan lebih lama.

Secara kompleksitas ruangan, GBFS adalah yang terbaik sebab tidak memerlukan penyimpanan berupa Priority Queue pada algoritma lainnya. Pada kasus terbaik, GBFS juga memiliki kompleksitas waktu yang terbaik sebab simpul simpul yang dipilih merupakan yang paling optimal.

5 Kesimpulan

Berdasarkan pengujian, dapat disimpulkan bahwa algoritma A* adalah algoritma paling efektif dan efisien jika dibandingkan dengan UCS selama digunakan heuristik yang admissible. Pada beberapa kasus, Greedy BFS mampu menghasilkan output dengan lebih cepat dibanding A* tetapi beresiko tinggi untuk digunakan sebab bisa saja terjebak dalam suatu dead end.

6 Lampiran

5.1 Tautan Repository

https://github.com/dhiabziz/Tucil3_13522040.git

5.2 Checklist Program

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

7 Daftar Pustaka

- Sugianti, N., Mardhiyah, A., & Fadhilah, N. R. (2020, November 3). *Komparasi Kinerja Algoritma BFS, Dijkstra, Greedy BFS, dan A* dalam Melakukan Pathfinding*. e-journal UIN Suka. Retrieved May 7, 2024, from <https://ejournal.uin-suka.ac.id/saintek/JISKA/article/download/53-07/1786/6251>
- Trivusia. (2022, October 17). *Apa itu Uniform-Cost Search? Pengertian dan Cara Kerjanya*. Trivusi. Retrieved May 7, 2024, from <https://www.trivusi.web.id/2022/10/apa-itu-algoritma-uniform-cost-search.html>