

Laporan Tugas Kecil 3 IF2211 Strategi Algoritma
Pemanfaatan Algoritma *UCS*, *Greedy-BFS*, dan *A-Star* dalam
Penyelesaian Permainan Word Ladder

Semester II Tahun 2023/2024



Disusun oleh:

Albert Ghazaly - 13522150

INSTITUT TEKNOLOGI BANDUNG

2023

BAB I

Deskripsi Masalah

Word Ladder atau permainan tangga kata (juga dikenal sebagai Doublets, tautan kata, teka-teki ubah-kata, paragrams, tangga kata, atau golf kata) adalah permainan kata yang diciptakan oleh Lewis Carroll. Teka-teki tangga kata dimulai dengan dua kata, dan untuk memecahkan teka-teki tersebut seseorang harus menemukan rangkaian kata lain untuk menghubungkan kedua kata tersebut, di mana dua kata berturut-turut (itu adalah, kata-kata dalam langkah-langkah yang berhasil) berbeda satu huruf.

Pada Tugas Kecil 3 Strategi Algoritma ini, mahasiswa diminta untuk mengimplementasikan algoritma *UCS*, *Greedy-BFS*, dan *A** untuk menyelesaikan permainan *word ladder* dan memberikan hasil berupa tahap penyelesaian, total langkah yang dibutuhkan, dan durasi penyelesaian.

BAB II

Landasan Teori

2.1 Algoritma UCS

Algoritma UCS (Uniform Cost Search) adalah algoritma pencarian graf yang digunakan untuk menemukan jalur terpendek dalam graf berbobot, di mana biaya setiap langkahnya berbeda. UCS memastikan bahwa jalur yang ditemukan memiliki biaya total minimum. Algoritma ini bekerja dengan cara memeriksa simpul secara berurutan berdasarkan biaya jalur terpendek yang telah ditemukan untuk mencapai simpul tersebut dari simpul awal.

2.2 Algoritma Greedy-BFS

Greedy-BFS adalah sebuah pendekatan dalam pencarian graf, menggabungkan prinsip serakah (greedy) dengan metode Breadth-First Search (BFS). Algoritma ini memilih simpul berikutnya untuk dieksplorasi berdasarkan pada kriteria yang menjanjikan hasil terbaik secara lokal. Kriteria yang dimaksud adalah kriteria dengan fungsi yang memiliki parameter berhubungan dengan efisiensi dalam mencari target atau tujuan. Jadi, daripada secara sistematis mengeksplorasi semua kemungkinan langkah, seperti yang dilakukan dalam BFS, algoritma ini memilih jalur yang tampaknya paling menjanjikan pada saat itu. Ini bisa berarti memilih simpul yang tampaknya paling dekat dengan tujuan atau memiliki sifat tertentu yang diharapkan membawa lebih cepat ke solusi.

2.3 Algoritma A*

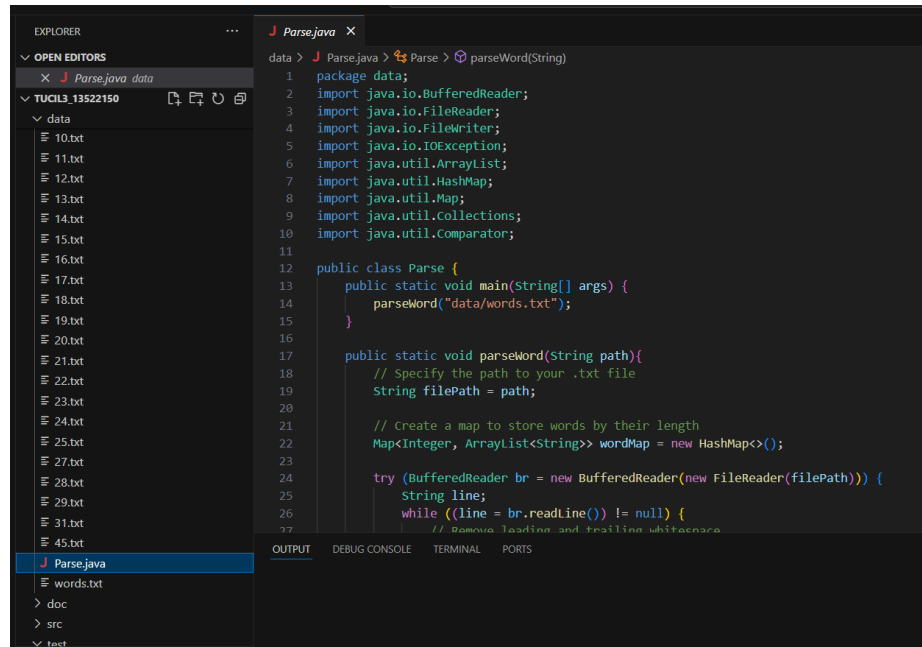
Algoritma A* muncul sebagai solusi cerdas dalam pencarian jalur di kecerdasan buatan dan pemodelan masalah pemetaan rute. Fokus utamanya adalah memetakan jalur terpendek atau jalur optimal di antara dua titik dalam sebuah graf. Algoritma ini menggabungkan informasi biaya aktual untuk mencapai simpul tertentu dengan perkiraan biaya yang tersisa untuk mencapai tujuan, yang dihitung dengan menggunakan fungsi heuristik. Dengan mempertimbangkan kedua informasi tersebut, A* memilih simpul

berikutnya untuk dieksplorasi. Tujuannya sederhana, yakni menemukan jalur yang paling menjanjikan menuju solusi dengan biaya total terkecil.

BAB III

Aplikasi Algoritma

3.1 Aplikasi Pemetaan Kamus Kata



Gambar 3.1.1 Parse.java

Kamus diunduh dari internet dan dimasukkan ke dalam *project directory* tepatnya di dalam folder data dengan *extension* .txt yang berisi kata setiap *line*. Selanjutnya, file tersebut akan di-*parsing* menggunakan program berbasis java bernama *Parse.java* yang memiliki fungsionalitas yang dideskripsikan sebagai berikut

1. Membaca file .txt yang telah disebutkan dan diiterasi setiap line dengan memasukkannya dalam sebuah variabel *string*
2. Mengecek apakah string tersebut berisi angka atau kode non alfabet, jika variabel tersebut terindikasi memiliki setidaknya sebuah angka atau kode non alfabet, maka string akan dibuang dan dilanjutkan ke kata pada line berikutnya.
3. Jika tidak ada angka atau kode non alfabet dalam kata tersebut maka kata akan dikapitalkan agar menyamai format dan dimasukkan ke dalam map yang memiliki key berupa panjang kata dan value berupa list of string (berisi kata-kata yang telah divalidasi)

4. Iterasi hingga semua kata dalam file .txt telah diiterasi
5. Buat file baru dengan format “[panjang kata].txt” dengan menulis semua kata yang berada pada map dengan *key* panjang kata tersebut sampai semua kata dalam map telah dimasukkan ke file.

3.2 Implementasi struktur data Tree

```
public class Tree{
    private int depth;
    private Tree parent;
    private String word;
    private int fn;
    private List<Tree> children;

    public Tree(){
        this.depth = 0;
        this.parent = null;
        this.word = "";
        this.fn = 0;
        this.children = new ArrayList<Tree>();
    }
    public Tree(Tree parent,String word,int depth,int fn){
        this.depth = depth;
        this.parent = parent;
        this.word = word;
        this.fn = fn;
        this.children = new ArrayList<Tree>();
    }
    public String getWord(){
        return this.word;
    }
}
```

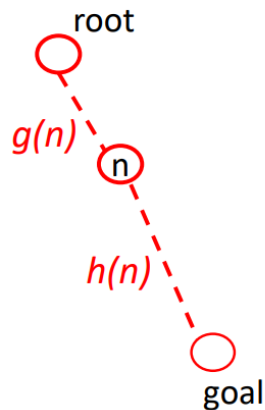
Gambar 3.2.1 Struktur data Tree

Pada tugas ini, digunakan struktur data berupa *tree* dalam menyimpan kata dan cost dari sebuah kata. Selain merepresentasikan sebuah *node* dari kata tersebut, *tree* yang digunakan juga menyimpan parent node dan *children node* yang merupakan kata mana saja yang memungkinkan untuk dikunjungi (hanya berbeda 1 huruf dari kata saat ini). Selain itu, struktur data ini juga memiliki atribut *fn* atau $f(n)$, yakni total cost function dan *depth* atau kedalaman yang nantinya akan dijelaskan pada 3.3.

Sebelum itu, akan dijelaskan mengenai mengapa struktur data yang dipilih adalah *tree* dan bukan graf. Singkatnya, *tree* digunakan dalam tugas ini karena *tree* dapat menghindari duplikat yang memudahkan program agar tidak perlu memvalidasi apakah sebuah node telah dikunjungi. Dengan begitu, sebuah node hanya boleh dimiliki satu *parent node* tidak bisa lebih selayaknya graf. Hal itu berguna karena dalam permainan

word ladder, tidak mungkin bagi “solusi paling optimal” untuk kembali ke kata yang sudah pernah ditulis sebelumnya.

3.3 Gambaran Umum Algoritma Pencarian



Gambar 3.3.1 Ilustrasi umum algoritma pencarian

Pada program ini, disediakan tiga jenis algoritma pencarian (*Searching Algorithm*), yakni *Uniform Cost Search* (UCS), *Greedy Breadth First Search* (Greedy-BFS), dan *A-Star* (A^*). Ketiga algoritma yang telah disebutkan menggunakan sebuah parameter yang dinamakan *cost solution* $F(n)$ yang merepresentasikan sebagai harga keseluruhan yang dibutuhkan dari node awal hingga mencapai node goal. Sedangkan di antaranya juga menggunakan parameter $g(n)$ yang merepresentasikan harga dari node awal menuju suatu node tertentu yang bukan node goal. Terakhir, terdapat parameter $h(n)$ yang merepresentasikan harga yang dibutuhkan dari suatu node, n_1 , tertentu menuju node goal, n_g , $n_1 \neq n_g$. Meskipun memiliki konsep yang berbeda, ketiga algoritma ini memiliki kesamaan, yakni mereka mencari solusi dengan menggapai node yang memiliki harga total ($f(n)$) terkecil.

3.4 Aplikasi Algoritma UCS

```
public void addChildren(String target, String method){
    if (!MainProgram.listVocabs.isEmpty()){
        int n = MainProgram.listVocabs.size();
        int j = 0;
        for (int i=0;i<n;i++){
            if (isChild(MainProgram.listVocabs.get(j))){
                if (method.compareTo(anotherString:"UCS")==0){
                    this.addChild(new Tree(this,MainProgram.listVocabs.get(j),this.depth+1,this.fn+1));
                }else if (method.compareTo(anotherString:"Greed")==0){
                    this.addChild(new Tree(this,MainProgram.listVocabs.get(j),this.depth+1,difLetter(MainProgram.listVocabs.get(j),target)));
                }else if (method.compareTo(anotherString:"A*")==0){
                    this.addChild(new Tree(this,MainProgram.listVocabs.get(j),this.depth+1,difLetter(MainProgram.listVocabs.get(j), target)+(this.depth+1)));
                }
                MainProgram.listVocabs.remove(j);
                j --1;
            }
            j+=1;
        }
    }
}

public static void addQueueTree(List <Tree> arr,Tree node){
    int i = 0;
    while (i<arr.size() && node.fn >= arr.get(i).fn) {
        i++;
    }
    arr.add(i,node);
}
```

Gambar 3.5.1 Implementasi $f(n)$ pada UCS

Algoritma UCS diimplementasikan dengan menjadikan total cost ($f(n)$) sama dengan cost dari node awal menuju node sekarang ($g(n)$) atau bisa ditulis sebagai $f(n) = g(n)$. Algoritma UCS yang dibuat berpacu pada $g(n)$ sebagai jumlah langkah yang dibutuhkan *word* awal menuju *word* tertentu. Perlu diperhatikan bahwa jumlah langkah yang dibutuhkan dari suatu kata menuju kata tertentu tidak sama dengan jumlah huruf yang berbeda karena terdapat kondisi di mana sebuah kata yang memiliki perbedaan huruf n mengunjungi node yang memiliki perbedaan huruf yang sama (n) dengan kata awal. Yang artinya tidak logis, jika children node memiliki perbedaan harga yang sama, delta cost = 0, dibanding parent node nya.

Oleh karena itu, $g(n)$ yang diambil dalam algoritma UCS adalah jumlah langkah atau bisa dikaitkan dengan kedalaman node tersebut relatif terhadap node kata awal sebagai *root node*. Hal tersebut terasa familiar karena algoritma UCS dalam kasus word ladder tak lain adalah algoritma *Breadth First Search* (BFS) disebabkan perbedaan harga yang diturunkan dari *parent node* menuju *children node* sama dengan satu, delta cost = 1. Oleh sebab itu, $f(n) = \text{depth}$ merupakan total cost dalam algoritma UCS yang digunakan dalam program. Adapun tahapan algoritma UCS dalam program

1. Memasukkan root node yang berisi kata awal dan $depth = 0$ ke dalam queue
2. Mencari children node dari root node dengan perbedaan huruf $=1$ dan $depth = parent_node + 1$ dan cek apakah children node tersebut terdapat target node
3. Jika terdapat target node, maka return node tersebut. Jika tidak, maka keluarkan root node dari queue dan masukkan children node ke dalam queue
4. Ulangi langkah 1 dengan node awal pada queue, $queue[0]$, layaknya BFS hingga target node ditemukan dan di-*return* node tersebut

3.5 Aplikasi Algoritma Greedy-BFS

```

public void addChildren(String target, String method){
    if (!MainProgram.listVocabs.isEmpty()){
        int n = MainProgram.listVocabs.size();
        int j = 0;
        for (int i=0;i<n;i++){
            if (isChild(MainProgram.listVocabs.get(j))){
                if (method.compareTo(anotherString:"UCS")==0){
                    this.addChild(new Tree(this,MainProgram.listVocabs.get(j),this.depth+1,this.fn+1));
                }else if (method.compareTo(anotherString:"Greedy")==0){
                    this.addChild(new Tree(this,MainProgram.listVocabs.get(j),this.depth+1,difLetter(MainProgram.listVocabs.get(j),target)));
                }else if (method.compareTo(anotherString:"A*")==0){
                    this.addChild(new Tree(this,MainProgram.listVocabs.get(j),this.depth+1,difLetter(MainProgram.listVocabs.get(j), target)+(this.depth+1)));
                }
                MainProgram.listVocabs.remove(j);
                j --1;
            }
            j+=1;
        }
    }
}

public static void addQueueTree(List <Tree> arr,Tree node){
    int i = 0;
    while (i<arr.size() && node.fn >= arr.get(i).fn) {
        i++;
    }
    arr.add(i,node);
}

```

Gambar 3.5.1 Implementasi $f(n)$ pada Greedy-BFS

Aplikasi algoritma Greedy-BFS yang digunakan dalam program ini berpacu pada total cost $f(n)$ sama dengan harga yang dibutuhkan *current node* menuju *goal node*, $h(n)$, atau $f(n)=h(n)$. Algoritma Greedy-BFS yang digunakan merepresentasikan cost dari node tertentu menuju *target node* ($h(n)$) sebagai perbedaan huruf antara *target node* dan node tertentu. Dengan kata lain, algoritma Greedy-BFS dalam program word ladder solver ini akan mencari node yang memiliki kemiripan terdekat dengan *target node* meskipun belum tentu terdapat sebuah kata yang menyambungkan kedua *node* tersebut. Adapun tahapan algoritma Greedy-BFS dalam program

1. Masukkan root node yang berisi kata awal ke dalam queue, queue ini adalah queue adalah priority queue di mana akan menjadikan node dengan $f(n)$ terkecil sebagai prioritas
2. Mencari children dari root node dengan perbedaan huruf = 1 dan $f(n)$ sama dengan $\text{different_letter}(\text{child_node}, \text{target_node})$.
3. Cek apakah children node terdapat target node diantaranya, jika iya maka return node tersebut
4. Jika tidak, maka keluarkan root node dari priority queue dan masukkan children node ke dalam priority queue berdasarkan $f(n)$ terendah
5. Ulangi langkah di atas dengan node yang memiliki $f(n)$ terendah atau $\text{queue}[0]$ hingga target node ditemukan dan di-*return*.

3.6 Aplikasi Algoritma A*

Aplikasi algoritma A-Star (A*) yang digunakan dalam program *word ladder solver* ini berdasar pada total cost ($f(n)$) sama dengan harga yang dibutuhkan untuk mencapai node selanjutnya ($g(n)$) ditambah dengan harga yang dibutuhkan dari node selanjutnya tersebut menuju *goal node* ($h(n)$), atau $f(n) = g(n) + h(n)$. Dengan kata lain, algoritma A-Star ini mengimplementasikan gabungan kedua cost yang digunakan dalam UCS, yakni $g(n) = \text{depth}$ dan Greedy-BFS, yakni $h(n) = \text{different_letter}(\text{curr_node}, \text{goal_node})$. Adapun tahapan algoritma A-Star (A*) sebagai berikut

1. Masukkan root node yang berisi kata awal ke dalam queue, queue ini adalah queue adalah priority queue di mana akan menjadikan node dengan $f(n)$ terkecil sebagai prioritas
2. Mencari children dari root node dengan perbedaan huruf = 1, $\text{depth} = \text{parent_depth} + 1$, dan $f(n)$ sama dengan $\text{depth} + \text{different_letter}(\text{child_node}, \text{target_node})$.
3. Cek apakah children node terdapat target node diantaranya, jika iya maka return node tersebut
4. Jika tidak, maka keluarkan root node dari priority queue dan masukkan children node ke dalam priority queue berdasarkan $f(n)$ terendah

5. Ulangi langkah di atas dengan node yang memiliki $f(n)$ terendah atau `queue[0]` hingga target node ditemukan dan di-*return*.

BAB IV

Implementasi dan Pengujian

4.1 Implementasi Struktur Data (Tree.java)

```
import java.util.ArrayList;
import java.util.List;
public class Tree{
    private int depth;
    private Tree parent;
    private String word;
    private int fn;
    private List<Tree> children;

    public Tree(){
        this.depth = 0;
        this.parent = null;
        this.word = "";
        this.fn = 0;
        this.children = new ArrayList<Tree>();
    }
    public Tree(Tree parent,String word,int depth,int fn){
        this.depth = depth;
        this.parent = parent;
        this.word =word;
        this.fn = fn;
        this.children = new ArrayList<Tree>();
    }
    public String getWord(){
        return this.word;
    }
    public int getFn(){
        return this.fn;
    }
    public List<Tree> getChildren(){
        return this.children;
    }
    public int getDepth(){
        return this.depth;
    }
    public void addChild(Tree newNode){
        this.children.add(this.children.size(),newNode);
    }
    public Tree getParent(){
        return this.parent;
    }
    public static int sameLetter(String curr,String target){
        int nSame = 0;
        for (int i=0;i<curr.length();i++){
            if (target.charAt(i)==curr.charAt(i)){
                nSame ++;
            }
        }
        return nSame;
    }
    public static int difLetter(String curr,String target){
        return target.length()-sameLetter(curr, target);
    }
    public boolean isChild(String childWord){
        int difference = 0;
        for (int i=0;i<this.word.length();i++){
            if (childWord.charAt(i)!=this.word.charAt(i)){
                difference ++;
            }
        }
        return difference==1;
    }
    public void addChildren(String target, String method){
        if (!MainProgram.listVocabs.isEmpty()){
            int n = MainProgram.listVocabs.size();
            int j = 0;
            for (int i=0;i<n;i++){
                if (isChild(MainProgram.listVocabs.get(j))){
                    if (method.compareTo("UCS")==0){
                        this.addChild(new
Tree(this,MainProgram.listVocabs.get(j),this.depth+1,this.fn+1));
                    }else if (method.compareTo("Greed")==0){
                        this.addChild(new
Tree(this,MainProgram.listVocabs.get(j),this.depth+1,difLetter(MainProgram.listVocabs.get(j),target));
                    }else if (method.compareTo("A*")==0){
                        this.addChild(new
Tree(this,MainProgram.listVocabs.get(j),this.depth+1,difLetter(MainProgram.listVocabs.get(j), target)+
(this.depth+1)));
                    }
                    MainProgram.listVocabs.remove(j);
                    j -=1;
                }
                j+=1;
            }
        }
    }
    public static void addQueueTree(List <Tree> arr,Tree node){
        int i = 0;
        while (i<arr.size() && node.fn >= arr.get(i).fn) {
            i++;
        }
        arr.add(i,node);
    }
}
```

4.2 Implementasi Algoritma UCS (UCS.java)

```
import java.util.ArrayList;
import java.util.List;

public class UCS {
    public static int visited_node = 0;
    public static void findUcsSolution( String wordAwal, String wordAkhir){
        UCS.visited_node = 0;
        long startTime = System.nanoTime();
        Tree root = new Tree(null,wordAwal,0,0);
        int i = 0;
        for (;i<MainProgram.listVocabs.size() &&
MainProgram.listVocabs.get(i).compareTo(wordAwal)!=0;i++){
        }
        MainProgram.listVocabs.remove(i);
        Tree result = new Tree();

        List<Tree> pq = new ArrayList<>();
        Tree.addQueueTree(pq,root);
        boolean found = false;
        while (!found && !pq.isEmpty()){
            Tree node = pq.remove(0);
            if (node.getWord().compareTo(wordAkhir)==0){
                found = true;
                result = node;
            }
            UCS.visited_node += 1;
            node.addChildren(wordAkhir, "UCS");
            for (int n=0;n<node.getChildren().size() && !found;n++){
                Tree.addQueueTree(pq,node.getChildren().get(n));
            }
        }
        long endTime = System.nanoTime();
        long elapsedTimeInMillis = (endTime - startTime) / 1_000_000;
        if (found){
            List<String> resultList = new ArrayList<String>();
            i = 0;
            while (result!=null){
                resultList.add(0,result.getWord());
                result = result.getParent();
                i++;
            }
            System.out.print("Step: ");
            System.out.println(i-1);
            System.out.print("Hasil: ");
            for (i=0;i<resultList.size();i++){
                System.out.print(i+1);
                System.out.print(". ");
                System.out.println(resultList.get(i));
            }
            System.out.println("Time execution: " + elapsedTimeInMillis + " ms");
            System.out.print("Visited Node: ");
            System.out.println(UCS.visited_node);
        }else{
            System.out.println("Gak Nemu !");
            System.out.println("Time execution: " + elapsedTimeInMillis + " ms");
            System.out.print("Visited Node: ");
            System.out.println(UCS.visited_node);
        }
    }
}
```

4.3 Implementasi Algoritma Greedy-BFS (Greed.java)

```
import java.util.ArrayList;
import java.util.List;

public class Greed {
    public static int visited_node = 0;

    public static void findGreedSolution(String wordAwal, String wordAkhir){
        Greed.visited_node = 0;
        long startTime = System.nanoTime();
        Tree root = new Tree(null, wordAwal, 0, 0);
        int i = 0;
        for (; i < MainProgram.listVocabs.size() &&
MainProgram.listVocabs.get(i).compareTo(wordAwal) != 0; i++){
        }
        MainProgram.listVocabs.remove(i);
        Tree result = new Tree();
        List<Tree> pq = new ArrayList<>();
        Tree.addQueueTree(pq, root);
        boolean found = false;
        if (wordAwal.compareTo(wordAkhir) == 0){
            found = true;
            result = root;
        }
        while (!found && !pq.isEmpty()){
            Tree node = pq.remove(0);
            if (node.getWord().compareTo(wordAkhir) == 0){
                found = true;
                result = node;
            }
            Greed.visited_node += 1;
            node.addChildren(wordAkhir, "Greed");
            for (int n = 0; n < node.getChildren().size() && !found; n++){
                Tree.addQueueTree(pq, node.getChildren().get(n));
            }
        }
        long endTime = System.nanoTime();
        long elapsedTimeInMillis = (endTime - startTime) / 1_000_000;
        if (found){
            List<String> resultList = new ArrayList<String>();
            i = 0;
            while (result != null){
                resultList.add(0, result.getWord());
                result = result.getParent();
                i++;
            }
            System.out.print("Step: ");
            System.out.println(i-1);
            System.out.println("Hasil: ");
            for (i = 0; i < resultList.size(); i++){
                System.out.print(i+1);
                System.out.print(" ");
                System.out.println(resultList.get(i));
            }
            System.out.println("Time execution: " + elapsedTimeInMillis + " ms");
            System.out.print("Visited Node: ");
            System.out.println(Greed.visited_node);
        } else {
            System.out.println("Gak Nemu !");
            System.out.println("Time execution: " + elapsedTimeInMillis + " ms");
            System.out.print("Visited Node: ");
            System.out.println(Greed.visited_node);
        }
    }
}
```

4.4 Implementasi Algoritma A* (AStar.java)

```
import java.util.ArrayList;
import java.util.List;

public class AStar {
    public static int visited_node = 0;

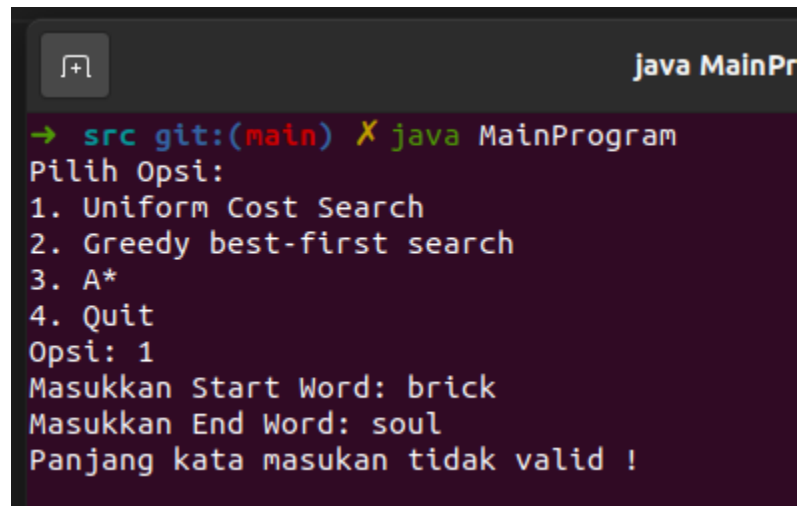
    public static void findAStarSolution(String wordAwal, String wordAkhir){
        AStar.visited_node = 0;
        long startTime = System.nanoTime();
        Tree root = new Tree(null, wordAwal, 0, 0);
        int i = 0;
        for (; i < MainProgram.listVocabs.size() &&
MainProgram.listVocabs.get(i).compareTo(wordAwal) != 0; i++){
        }
        MainProgram.listVocabs.remove(i);
        Tree result = new Tree();
        List<Tree> pq = new ArrayList<>();
        Tree.addQueueTree(pq, root);
        boolean found = false;
        while (!found && !pq.isEmpty()){
            Tree node = pq.remove(0);
            if (node.getWord().compareTo(wordAkhir) == 0){
                found = true;
                result = node;
            }
            AStar.visited_node += 1;
            node.addChildren(wordAkhir, "A*");
            for (int n = 0; n < node.getChildren().size() && !found; n++){
                Tree.addQueueTree(pq, node.getChildren().get(n));
            }
        }
        long endTime = System.nanoTime();
        long elapsedTimeInMillis = (endTime - startTime) / 1_000_000;
        if (found){
            List<String> resultList = new ArrayList<String>();
            i = 0;
            while (result != null){
                resultList.add(0, result.getWord());
                result = result.getParent();
                i++;
            }
            System.out.print("Step: ");
            System.out.println(i-1);
            System.out.println("Hasil: ");
            for (i = 0; i < resultList.size(); i++){
                System.out.print(i+1);
                System.out.print(" ");
                System.out.println(resultList.get(i));
            }
            System.out.println("Time execution: " + elapsedTimeInMillis + " ms");
            System.out.print("Visited Node: ");
            System.out.println(AStar.visited_node);
        } else {
            System.out.println("Gak Nemu !");
            System.out.println("Time execution: " + elapsedTimeInMillis + " ms");
            System.out.print("Visited Node: ");
            System.out.println(AStar.visited_node);
        }
    }
}
```

4.5 Pengujian

1. Test Case 1

Kata awal: brick

Kata akhir: soul



```
java MainPr  
→ src git:(main) X java MainProgram  
Pilih Opsi:  
1. Uniform Cost Search  
2. Greedy best-first search  
3. A*  
4. Quit  
Opsi: 1  
Masukkan Start Word: brick  
Masukkan End Word: soul  
Panjang kata masukan tidak valid !
```

2. Test Case 2

Kata awal: qusg

Kata Akhir: bskl


```
java MainProgram
→ src git:(main) X java MainProgram
Pilih Opsi:
1. Uniform Cost Search
2. Greedy best-first search
3. A*
4. Quit
Opsi: 1
Masukkan Start Word: brick
Masukkan End Word: soul
Panjang kata masukan tidak valid !

Masukkan Start Word: qusg
Masukkan End Word: bskl
Kata awal tidak valid !
```

3. Test Case 3

Kata awal: get

Kata Akhir : god

UCS

```
Pilih Opsi:
1. Uniform Cost Search
2. Greedy best-first search
3. A*
4. Quit
Opsi: 1
Masukkan Start Word: get
Masukkan End Word: god
Step: 2
Hasil:
1. GET
2. GED
3. GOD
Time execution: 12 ms
Visited Node: 292
```

Greedy-BFS

```
Pilih Opsi:
1. Uniform Cost Search
2. Greedy best-first search
3. A*
4. Quit
Opsi: 2
Masukkan Start Word: get
Masukkan End Word: god
Step: 2
Hasil:
1. GET
2. GED
3. GOD
Time execution: 2 ms
Visited Node: 3
```

A*

```
Pilih Opsi:
1. Uniform Cost Search
2. Greedy best-first search
3. A*
4. Quit
Opsi: 3
Masukkan Start Word: get
Masukkan End Word: god
Step: 2
Hasil:
1. GET
2. GED
3. GOD
Time execution: 3 ms
Visited Node: 3
```

4. Test Case 4

Kata awal: goal

Kata Akhir : girl

UCS

```
Pilih Opsi:  
1. Uniform Cost Search  
2. Greedy best-first search  
3. A*  
4. Quit  
Opsi: 1  
Masukkan Start Word: goal  
Masukkan End Word: girl  
Step: 3  
Hasil:  
1. GOAL  
2. GOLL  
3. GILL  
4. GIRL  
Time execution: 142 ms  
Visited Node: 1339  
Pilih Opsi:
```

Greedy-BFS

```
Pilih Opsi:  
1. Uniform Cost Search  
2. Greedy best-first search  
3. A*  
4. Quit  
Opsi: 2  
Masukkan Start Word: goal  
Masukkan End Word: girl  
Step: 3  
Hasil:  
1. GOAL  
2. GOLL  
3. GILL  
4. GIRL  
Time execution: 2 ms  
Visited Node: 5  
Pilih Opsi:
```

A*

```
=====
Pilih Opsi:
1. Uniform Cost Search
2. Greedy best-first search
3. A*
4. Quit
Opsi: 3
Masukkan Start Word: goal
Masukkan End Word: girl
Step: 3
Hasil:
1. GOAL
2. GOLL
3. GILL
4. GIRL
Time execution: 2 ms
Visited Node: 10
```

5. Test Case 5

Kata awal: mount

Kata Akhir : moist

UCS

```
Pilih Opsi:  
1. Uniform Cost Search  
2. Greedy best-first search  
3. A*  
4. Quit  
Opsi: 1  
Masukkan Start Word: mount  
Masukkan End Word: moist  
Step: 5  
Hasil:  
1. MOUNT  
2. MOULT  
3. MOULE  
4. MOUSE  
5. MOISE  
6. MOIST  
Time execution: 167 ms  
Visited Node: 814
```

Greedy-BFS

```
Pilih Opsi:  
1. Uniform Cost Search  
2. Greedy best-first search  
3. A*  
4. Quit  
Opsi: 2  
Masukkan Start Word: mount  
Masukkan End Word: moist  
Step: 5  
Hasil:  
1. MOUNT  
2. MOULT  
3. MOULE  
4. MOUSE  
5. MOISE  
6. MOIST  
Time execution: 3 ms  
Visited Node: 10
```

A*

```
Pilih Opsi:  
1. Uniform Cost Search  
2. Greedy best-first search  
3. A*  
4. Quit  
Opsi: 3  
Masukkan Start Word: mount  
Masukkan End Word: moist  
Step: 5  
Hasil:  
1. MOUNT  
2. MOULT  
3. MOULE  
4. MOUSE  
5. MOISE  
6. MOIST  
Time execution: 5 ms  
Visited Node: 17
```

6. Test Case 6

Kata awal: playing

Kata Akhir : tremble

UCS

```
Pilih Opst:
1. Uniform Cost Search
2. Greedy best-first search
3. A*
4. Quit
Opst: 1
Masukkan Start Word: playing
Masukkan End Word: tremble
Step: 27
Hasil:
1. PLAYING
2. PLATING
3. PRATING
4. GRATING
5. GRATINE
6. GRATIAE
7. GRATIAN
8. GRATTAN
9. GRATTEN
10. GRITTEN
11. GRITTER
12. CRITTER
13. CHITTER
14. CHATTER
15. CHARTER
16. CHARGER
17. CHARGE
18. CHARLEE
19. CHARLIE
20. CHARPTE
21. CHAPPIE
22. CRAPPIE
23. CRAPPLE
24. CRIPPLE
25. CRIMPLE
26. CRIMBLE
27. TRIMBLE
28. TREMBLE
Time execution: 7954 ms
Visited Node: 15627
```

Greedy-BFS

```
Pilih Opsi:
1. Uniform Cost Search
2. Greedy best-first search
3. A*
4. Quit
Opsi: 2
Masukkan Start Word: playing
Masukkan End Word: tremble
Step: 56
Hasil:
1. PLAYING
2. PRAYING
3. PREYING
4. PREEING
5. TREEING
6. TRUEING
7. TRUCING
8. TRACING
9. TRADING
10. GRADING
11. GRADINE
```

```
48. CHARLIE
49. CHARPIE
50. CHAPPIE
51. CRAPPIE
52. CRAPPLE
53. CRIPPLE
54. CRIMPLE
55. CRIMBLE
56. TRIMBLE
57. TREMBLE
Time execution: 4768 ms
Visited Node: 8281
```

A*


```
Pilih Opsi:
1. Uniform Cost Search
2. Greedy best-first search
3. A*
4. Quit
Ops: 3
Masukkan Start Word: playing
Masukkan End Word: tremble
Step: 27
Hasil:
1. PLAYING
2. PRAYING
3. GRAYING
4. GRATING
5. GRATINE
6. GRATIAE
7. GRATIAN
8. GRATTAN
9. GRATTEN
10. GRITTEN
11. GRITTER
12. CRITTER
13. CHITTER
14. CHATTER
15. CHARTER
16. CHARGER
17. CHARGE
18. CHARLEE
19. CHARLIE
20. CHARPIE
21. CHAPPIE
22. CRAPPIE
23. CRAPPLE
24. CRIPPLE
25. CRIMPLE
26. CRIMBLE
27. TRIMBLE
28. TREMBLE
Time execution: 6308 ms
Visited Node: 13165
```

7. Test Case 7

Kata awal: ADZHARISTAN

Kata Akhir : HAMMERHEADS

```

Pilih Opsi:
1. Uniform Cost Search
2. Greedy best-first search
3. A*
4. Quit
Opsi: 1
Masukkan Start Word: ADZHARISTAN
Masukkan End Word: HAMMERHEADS
Gak Nemu !
Time execution: 8 ms
Visited Node: 1
Pilih Opsi:

```

8. Test Case 8

Kata Awal: PNEUMONULTRAMICROSCOPICSILICOVOLCANOCONIOSIS

Kata Akhir : PNEUMONULTRAMICROSCOPICSILICOVOLCANOCONIOSIS

```

Masukkan Start Word: PNEUMONULTRAMICROSCOPICSILICOVOLCANOCONIOSIS
Masukkan End Word: PNEUMONULTRAMICROSCOPICSILICOVOLCANOCONIOSIS
Step: 0
Hasil:
1. PNEUMONULTRAMICROSCOPICSILICOVOLCANOCONIOSIS
Time execution: 0 ms
Visited Node: 1

```

4.6 Analisis

1. Solusi Optimal

Pada program yang dibuat, dari ketiga algoritma (UCS, Greedy-BFS, dan A*), tidak semua algoritma menemukan solusi optimal. Pada Algoritma UCS, karena algoritma tersebut sudah pasti mengecek semua node yang ada dimulai dari depth terendah seperti halnya BFS, pada kasus *word ladder*, algoritma UCS merupakan algoritma yang pasti menemukan solusi optimal. Dapat dilihat dari test case yang diberikan di atas, jumlah path hasil UCS pasti terkecil.

Sama halnya pada algoritma A*, algoritma A* pada kasus *word ladder* memiliki heuristik yang *admissible* karena $h(n)$ pada algoritma A* ini, merepresentasikan perbedaan huruf dari node tertentu menuju target node di mana hasil tersebut pasti lebih kecil sama dengan perubahan yang dibutuhkan dari node tersebut menuju target node disebabkan tidak pada realitanya tidak semua kata berhubung langsung tanpa menggunakan kata perantara yakni $f(n) \text{ parent} = f(n) \text{ child}$. Karena hal tersebut, $h(n) \leq h'(n)$ yang menandakan bahwa heuristik pada algoritma A* dalam konteks *word ladder* bersifat *admissible*. Oleh karena itu, sesuai teori dan hasil pengujian bahwa algoritma A* memiliki jumlah path hasil terkecil.

Di sisi lain, algoritma Greedy-BFS dalam konteks *word ladder* memilih $f(n) = h(n)$ sebagai cost dari node tertentu menuju target node. Dengan kata lain, tidak peduli seberapa jauh node tersebut dari node awal, yang hanya dipedulikan adalah kedekatan antara node tersebut dengan target node. Sekilas terdengar optimal, akan tetapi dalam konteks *word ladder* dekatnya node tertentu ke sebuah node tidak menggambarkan cost yang dibutuhkan antara kedua node tersebut karena bisa jadi diperlukan node tambahan sebagai “penghubung” bagi kedua node tersebut. Oleh karena itu, meskipun cepat karena ruang lingkup node yang diperiksa dalam Greedy-BFS dekat dengan node target, tetapi solusi yang dihasilkan Greedy-BFS dapat menjadi solusi non-optimal dan terdapat kemungkinan bahwa solusi tidak ditemukan meskipun dalam dictionary memungkinkan.

2. Kompleksitas Waktu dan Memori

Pada dasarnya, kompleksitas waktu yang dibutuhkan dengan acuan Big-O bagi ketiga algoritma sama dengan $O(b^m)$, yakni eksponensial. Akan tetapi, $T(b,m)$ yang dibutuhkan dalam Greedy-BFS lebih cepat dibanding UCS dan A*. Hal ini karena Greedy-BFS berpacu pada $h(n)$ tanpa memperdulikan node lainnya yang menyebabkan ruang lingkup pencarian metode Greedy-BFS lebih kecil dan terpusat kepada target node. Selain itu, hal tersebut juga membuat kompleksitas memori $T(b,m)$ lebih kecil dari UCS dan A* karena memungkinkan terdapat node yang tidak diperiksa dan disimpan ke dalam queue. Hal tersebut juga dibuktikan dengan hasil pengujian di mana jumlah node dan *time execution* yang diperiksa Greedy-BFS paling rendah dibanding UCS dan A*.

Di sisi lain, algoritma UCS merupakan algoritma pencarian dalam konteks *word ladder* dengan menjamin solusi yang optimal. Akan tetapi, disebabkan $f(n) = g(n)$ di mana $g(n)$ merupakan kedalaman (*depth*) dari node tersebut, layaknya BFS, algoritma UCS dalam konteks ini juga memiliki kompleksitas waktu yang paling lama dibanding Greedy-BFS dan A* karena harus memeriksa semua node pada depth tertentu meskipun sama-sama $O(b^m)$. Hal yang sama juga berlaku kepada kompleksitas memori UCS karena harus menyimpan semua node pada setiap depth, kompleksitas memori UCS juga merupakan terberat dibandingkan Greedy-BFS dan A*. Hal tersebut juga dibuktikan pada hasil pengujian yang memberikan informasi bahwa jumlah node yang diperiksa dan *time execution* UCS merupakan yang paling besar.

Terakhir, Algoritma A* sebagai kombinasi dari UCS dan Greedy-BFS. Algoritma ini memiliki kompleksitas waktu dan memori di antara UCS dan Greedy-BFS, yakni lebih besar dari Greedy-BFS dan lebih kecil dari UCS. Meskipun begitu, algoritma A* dalam konteks *word ladder* memberikan solusi yang optimal karena *heuristic* pada algoritma ini bersifat *admissible* yang telah dibahas pada 4.4. Pada pengujian juga telah dibuktikan, algoritma A* memiliki jumlah node yang dikunjungi dan *time execution* lebih besar dari Greedy-BFS dan lebih kecil dari

UCS. Meskipun begitu, solusi yang dihasilkan algoritma ini paling optimal dengan jumlah path sama dengan solusi UCS meskipun lebih efisien.

3. Solusi Algoritma Paling Sesuai

Seperti yang sudah dibahas pada subbab di atas, kompleksitas algoritma Greedy-BFS memang yang paling rendah dan efisien. Akan tetapi, solusi yang dihasilkan tidak menjamin optimal yang membuat algoritma ini tidak sesuai untuk menjadi algoritma *word ladder solver*. Di sisi lain, algoritma UCS merupakan algoritma yang menjanjikan solusi optimal disebabkan pencarian berpusat pada node paling dekat dengan *root node* meskipun kekurangannya terdapat pada kompleksitas algoritma yang berat. Dengan adanya algoritma A* yang memiliki kompleksitas lebih ringan dari UCS dan menjamin solusi optimal tidak seperti Greedy-BFS, dapat disimpulkan bahwa algoritma A-Star (A*) merupakan algoritma paling sesuai dari ketiga algoritma yang digunakan dalam program dengan konteks permainan *word ladder*.

Bab V

Kesimpulan

5.1 Kesimpulan

Pada esensinya, algoritma A* merupakan algoritma yang memiliki kompleksitas algoritma lebih berat dari algoritma Greedy-BFS dan lebih ringan dari algoritma UCS. Meskipun begitu, algoritma A* dalam konteks *word ladder* menjamin solusi optimal seperti algoritma UCS dan tidak seperti algoritma Greedy-BFS. Oleh karena itu, algoritma A-Star(A*) merupakan algoritma yang paling sesuai untuk digunakan dalam mencari solusi paling optimal dalam permainan *word ladder*.

5.2 Saran

Dalam pengecekan kata, disarankan kedepannya untuk mencari dan menggunakan kamus yang lebih lengkap karena jumlah data yang tersedia juga mempengaruhi hasil dari pencarian. Semakin lengkap kamus, semakin besar kemungkinan terdapat solusi dari algoritma yang dijalankan. Analisis yang ditulis berasumsi bahwa terdapat setidaknya satu jalur untuk menuju target node dari node awal. Akan tetapi, terdapat kemungkinan bahwa memang tidak ada jalan dari node awal ke node akhir. Jika hal tersebut terjadi, bahkan algoritma UCS dan A* pun tidak dapat mencari path karena memang tidak ada.

Lampiran

Link repository GitHub:

https://github.com/AlbertGhazaly/Tucil3_13522150

CheckPoint

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	•	Note:dengan solusi belum tentu optimal dan tidak terjamin ketemu
5. Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma A*	•	
6. Solusi yang diberikan pada algoritma A* optimal	•	
7. [Bonus]: Program memiliki tampilan GUI		•

Daftar Pustaka

- Munir, Rinaldi. 2024. Penentuan rute (Route/Path Planning) - Bagian 1” di <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian-1-2021.pdf> (diakses 6 Mei 2024).
- Munir, Rinaldi. 2024. Penentuan rute (Route/Path Planning) - Bagian 2” di <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian-2-2021.pdf> (diakses 6 Mei 2024).