LAPORAN TUGAS KECIL 3

IF2211 STRATEGI ALGORITMA

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

Disusun oleh:

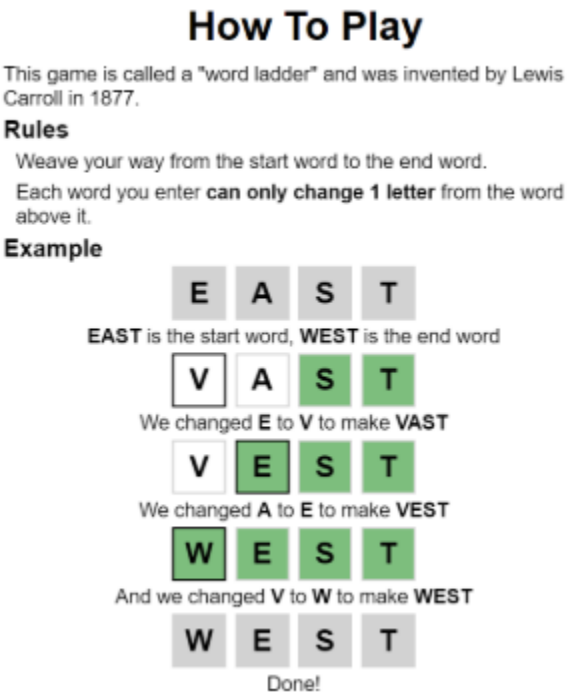
Rafif Ardhinto Ichwantoro (13522159)

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

BANDUNG
2024

DAFTAR ISI

BAB I
DESKRIPSI MASALAH



Gambar 1. Ilustrasi dan Peraturan Permainan Word Ladder
(Sumber: Weaver - A daily word ladder game)

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.

BAB II
TEORI SINGKAT

2.1 Algoritma Greedy Best First Search

Greedy Best-First Search (GBFS) adalah algoritma pencarian yang mirip dengan algoritma Best-First Search, tetapi dengan tambahan sifat serakah (greedy). Algoritma ini memprioritaskan simpul yang memiliki nilai fungsi evaluasi terendah dari simpul yang dapat dijangkau saat ini. Dengan kata lain, GBFS cenderung memilih simpul yang secara heuristik tampak paling dekat dengan solusi, tanpa mempertimbangkan seluruh gambaran ruang pencarian.

2.2 Algoritma Uniform Cost Search

Uniform Cost Search adalah algoritma pencarian untuk mencari jalur yang paling optimal berdasarkan bobot setiap node yang dilalui degan kata lain dengan total bobot terendah. Kelebihan UCS adalah kemampuannya menemukan jalur terpendek dengan biaya minimal. Namun, algoritma ini bisa menjadi lambat jika biaya antar simpul tidak seragam atau jika ada banyak simpul yang harus dieksplorasi.

2.3 Algoritma A*

Algoritma A* adalah algoritma pencarian yang digunakan untuk menemukan jalur terpendek atau solusi terbaik dalam ruang keadaan yang kompleks. Algoritma ini menggabungkan pendekatan Greedy Best-First Search (GBFS) dengan konsep biaya aktual dan estimasi biaya (heuristik) untuk mencapai tujuan. Algoritma A* secara umum memberikan solusi yang optimal karena menggabungkan strategi Greedy Best-First

Search untuk mempercepat pencarian dengan estimasi biaya yang membantu menghindari jalur-jalur yang tidak mengarah ke tujuan secara efisien. Namun, keefektifan A* sangat tergantung pada kualitas heuristik yang digunakan; heuristik yang buruk dapat menyebabkan pencarian jalur yang suboptimal atau bahkan tidak menemukan solusi.

BAB III
ANALISIS DAN IMPLEMENTASI ALGORITMA

3.1 Algoritma Greedy Best-First Search

1. Program menerima inputan berupa string end dan start dan Hastset<string> yang menrupakan dictionary word yang bisa dipakai dalam penyelesaian.
2. Program membuat List<string> sebagai path penyelesaian permainan. Lalu memasukan stirng start sebagai elemen pertama.
3. Program mencari setiap stirng yang memilki perbedaan 1 huruf dengan startNode yang memiliki nilai terkecil sesuai dengan fungsi f(n)
4. Jika sudah ditemukan program akan memasukan string tersebut kedalam List<stirng>, dan menjadikan menjadikan string yang baru saja dimasukan menjadi startNode.
5. Proses 3 dan 4 diulang terus sampai current string = goal , jika sudah tidak terdapat kata baru yang dimasukna ke path, maka path diaangap tidak dapat dicari.

3.2 Algoritma UCS

1. Inisialisasi PriorityQueue (queue) yang akan menyimpan node-node yang akan dieksplorasi berdasarkan costnya, HashSet visited untuk menyimpan node yang sudah dieksplorasi, serta HashMap cameFrom dan costSoFar untuk melacak jalur dan cost dari start node ke node lainnya.
2. Tambahkan node start ke dalam queue dengan cost awal 0, dan tandai sebagai sudah dikunjungi di visited.
3. Selama queue tidak kosong, lakukan langkah-langkah berikut:
   a. Ambil node dengan cost terkecil dari queue sebagai node saat ini (current).

b. Jika current sama dengan node tujuan (end), maka rekonstruksi jalur dari start ke end menggunakan informasi yang disimpan di cameFrom dan costSoFar, lalu kembalikan jalur tersebut.

c. Jika current bukan node tujuan, maka lakukan langkah-langkah berikut untuk setiap kata (word) yang berbeda satu karakter dengan current:

i. Cek apakah kata tersebut belum pernah dikunjungi (!visited.contains(word)).

ii. Hitung cost baru (newCost) dengan menambahkan cost dari current ke word dengan fungsi calculateCost(current.getWord(), word).

iii. Jika cost baru lebih kecil dari cost yang sudah ada atau node tersebut belum pernah dikunjungi sebelumnya, update costSoFar, tambahkan node ke queue, dan tandai node tersebut telah dikunjungi di visited. Simpan informasi jalur di cameFrom.

4. Jika queue kosong dan belum ditemukan jalur, kembalikan null.

3.3 Algoritma A*

1. Inisialisasi PriorityQueue (queue) yang akan menyimpan node-node yang akan dieksplorasi berdasarkan nilai prioritasnya, HashSet visited untuk menyimpan node yang sudah dieksplorasi, serta HashMap cameFrom dan costSoFar untuk melacak jalur dan cost dari start node ke node lainnya.
2. Tambahkan node start ke dalam queue dengan nilai prioritas awal 0, dan tandai sebagai sudah dikunjungi di visited.
3. Selama queue tidak kosong, lakukan langkah-langkah berikut:

   a. Ambil node dengan nilai prioritas terkecil dari queue sebagai node saat ini (current).

   b. Jika current sama dengan node tujuan (end), maka rekonstruksi jalur dari start ke end menggunakan informasi yang disimpan di cameFrom dan costSoFar, lalu kembalikan jalur tersebut.

   c. Jika current bukan node tujuan, maka lakukan langkah-langkah berikut untuk setiap kata (word) yang berbeda satu karakter dengan current dan belum pernah dikunjungi:

   i. Hitung cost baru (newCost) dengan menambahkan cost dari start ke current dengan cost dari current ke word menggunakan fungsi UCS.calculateCost(current.getWord(), word).

ii. Jika cost baru lebih kecil dari cost yang sudah ada atau node tersebut belum pernah dikunjungi sebelumnya, update costSoFar, hitung nilai prioritas baru (priority) dengan menambahkan cost baru dengan estimasi jarak ke tujuan menggunakan fungsi GBFS.levenshteinDistance(word, end), tambahkan node ke queue, dan tandai node tersebut telah dikunjungi di visited. Simpan informasi jalur di cameFrom.

4. Jika queue kosong dan belum ditemukan jalur, kembalikan null.

3.4 Definisi f(n) dan g(n)

Fungsi f(n) digunakan dalam program Gbfs dan A*. fungsi f (n) merupakan fungsi lavenstheinDistance. fungsi Ini adalah metode yang digunakan untuk mengukur seberapa berbedanya dua string berdasarkan jumlah operasi minimum (insert, delete, replace) yang diperlukan untuk mengubah satu string menjadi string yang lain. Fungsi ini memanfaatkan tabel dinamis (dynamic programming) untuk menghitung jarak Levenshtein antara dua string.

Fungsi g(n) digunakna dalam program UCS dan A*. dalam program fungsi g(n) bernama calculateCost. Fungsi ini berguna untuk menghitung perbedaan huruf antara currentword dan endword.

3.5 Analisi Heuristik pada Program

Heuristik dikatakan admissable jika memenuhi kreteria konsiten dan optimistik. Levenshtein distance memenuhi sifat konsisten, Nilai Levenshtein distance antara dua string adalah jumlah minimum operasi (insert, delete, replace) yang diperlukan untuk mengubah satu string menjadi string yang lain. Dengan demikian, nilai Levenshtein distance antara dua string tidak akan lebih besar dari cost untuk mencapai salah satu string dari yang lain. Levenshtein distance memberikan estimasi jarak yang cukup baik dalam konteks pencarian jalur. Meskipun tidak selalu optimal, Levenshtein distance umumnya memberikan perkiraan yang cukup dekat dengan jarak sebenarnya, terutama dalam kasus-kasus di mana perbedaan antara dua string tidak terlalu besar. Dengan ini heurisstik yang digunakan dikatakan admissable.

3.6 Analisi Hubungan UCS dengan BFS

Secara pencarian UCS dengan BFS memiliki sedikit kemiripan yaitu melakukan backtracking dalam pencarianya. Namun dalam UCS dilakukan backtracking dengan melakukan perhitungan dari cost yang telah didapatkan dari setiap perjalanan searching menuju ke node tujuan. Sedangkan pada BFS hanya dilakukan pencarian secara melebar sampai ke node tujuan. Secara hasil sudah jelas dalam beberapa kasus akan menghasilkan path pencarian yang berbeda.

3.7 Analisis Perbandingan Keefesienan Algoritam UCS dan A*

Perbedaan keefisienan Algoritma UCS dan A* dapat dilihat dari dua aspek yaitu kecepatan pencarian solusi dan Optimalitas solusi. Dalam kasus ini UCS memiliki solusi yang paling optimal dibanding semua algoritma yang telah kita buat karena dalam prosesnya algoritma ini mencari path dengan akumulasi cost paling rendah. Namun, karena mencoba semua kemungkinan tentu saja prosesnya memerlukan waktu yang lama. Algoritma A* menawarkan keefesienan yang lebih seimbang antara kecepatan dan optimalitas solusi. Dalam prosesnya algoritma ini menggunakna 2 pendekatan yaitu denga menghitung cost ke tujuan (seperti UCS) dan menggunakan heuristik Levenshtein distance (sepeti GBFS) secara garis bersar algoritma A* menggabungkan 2 konsep. Jadi kesimpulanya tergantung dari peraturan permaninan, jika memperhitungkan waktu maka penggunaan A* lebih tepat, jika hanya mementinkan solusi dengan path terpendek maka gunakna UCS.

3.8 Analisi Algoritma GBFS terhadap Solusi

Dari solusi yang didapatkan menggunakna Algoritma GBFS didapatkan waktu yang sangat cepat untuk menemukan solusi. Namun, dalam beberapa kasus pencarian program tidak dapat menemukan solusi karena Algoritma pencarian tidak bisa melakukan backtracking dan terjebak dipilihan kata yang sudah dilalui.

BAB IV
PENJELASAN PROGRAM

4.1 Class *GBFS*

```java
public static List<String> Gbfs(String start, String end, HashSet<String> words) {
        List<String> path = new ArrayList<>();
        path.add(start);
        boolean found = false;
        boolean stuck = false;
        Node startNode = new Node(start,cost:0);
        Integer min = 999999999;
        Node newStartNode = new Node(word:"",cost:0);
        while(found == false && stuck == false){
            for (String word : words){
                if (isOneLetterDifferentSamePosition(startNode.getWord(), word) && !path.contains(word)){
                    Integer temp = levenshteinDistance(word, end);
                    if (temp < min){
                        min = temp;
                        newStartNode = new Node(word,cost:0);
                    }
                }
            }
            if (newStartNode.getWord().equals(anObject:"")){
                stuck = true;
                return new ArrayList<>();
            }else if(newStartNode.getWord().equals(startNode.getWord())) {
                stuck = true;
                return new ArrayList<>();
            } else {
                startNode = newStartNode;
                path.add(startNode.getWord());
                if (startNode.getWord().equals(end)){
                    found = true;
                }
            }
        }
        return path;
    }
```

Gambar 2. Snippet file GBFS.java

1. Fungsi lavenstheinDistance

```java
public static int levenshteinDistance(String word1, String word2) {
    int[][] dp = new int[word1.length() + 1][word2.length() + 1];

    for (int i = 0; i <= word1.length(); i++) {
        dp[i][0] = i;
    }
    for (int j = 0; j <= word2.length(); j++) {
        dp[0][j] = j;
    }

    for (int i = 1; i <= word1.length(); i++) {
        for (int j = 1; j <= word2.length(); j++) {
            int cost = word1.charAt(i - 1) == word2.charAt(j - 1) ? 0 : 1;
            dp[i][j] = Math.min(dp[i - 1][j - 1] + cost, Math.min(dp[i - 1][j] + 1, dp[i][j - 1] + 1));
        }
    }

    return dp[word1.length()][word2.length()];
}
```

Gambar 3. Snippet file GBFS.java

2. Fungsi isOneLetterDifferentSamePosition

```java
public static boolean isOneLetterDifferentSamePosition(String word1, String word2) {
    if (word1.length() != word2.length()) {
        return false; // Kata-kata harus memiliki panjang yang sama
    }

    int diffCount = 0;
    for (int i = 0; i < word1.length(); i++) {
        if (word1.charAt(i) != word2.charAt(i)) {
            diffCount++;
            if (diffCount > 1) {
                return false; // Kata-kata berbeda lebih dari satu huruf
            }
        }
    }

    return diffCount == 1; // Kata-kata berbeda tepat satu huruf di posisi yang sama
}
```

Gambar 4. Snippet file GBFS.java

Dalam class GBFS berisi method-method yang diperlukan dalam penyelsaian Algoritma GBFS.

Method	Deskripsi
public static List<String> Gbfs(String start, String end, HashSet<String> words)	Method ini berfungsi untuk menjalankan algoritma GBFS seperti pada subab 3.1
public static boolean isOneLetterDifferentSamePosition(String word1, String word2)	Method ini berfungsi untuk memfilter agar hanya didapatkan string yang memiliki perbedaan huruf satu

public static int levenshteinDistance(String word1, String word2)	Method ini merupakan heuristik yang digunakan untuk proses GBFS.

4.2 Class *UCS*

```java
public static List<String> Ucs(String start, String end, HashSet<String> words) {
    PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(Node::getCost));
    queue.add(new Node(start, cost:0));

    HashSet<String> visited = new HashSet<>();
    visited.add(start);

    HashMap<String, String> cameFrom = new HashMap<>();
    HashMap<String, Integer> costSoFar = new HashMap<>();
    costSoFar.put(start, value:0);

    while (!queue.isEmpty()) {
        Node current = queue.poll(); // mengembalikan head queue

        if (current.getWord().equals(end)) {
            // Reconstruct path
            List<String> path = new ArrayList<>();
            while (current != null && current.getWord() != null) {
                path.add(current.getWord());
                if (cameFrom.get(current.getWord()) != null) {
                    current = new Node(cameFrom.get(current.getWord()), costSoFar.get(cameFrom.get(current.getWord())));
                } else {
                    current = null;
                }
            }
            Collections.reverse(path);
            return path;
        }

        for (String word : words) {
            if (GBFS.isOneLetterDifferentSamePosition(current.getWord(), word) && !visited.contains(word)) {
                if (costSoFar.containsKey(current.getWord())) {
                    int newCost = costSoFar.get(current.getWord()) + calculateCost(current.getWord(), word);

                    if (!costSoFar.containsKey(word) || newCost < costSoFar.get(word)) {
                        costSoFar.put(word, newCost);
                        queue.add(new Node(word, newCost));
                        cameFrom.put(word, current.getWord());
                        visited.add(word);
                    }
                }
            }
        }
    }

    return null; // Return null if no path is found
}
```

Gambar 5. Snippet file UCS.java

1. Fungsi calculateCost

```java
public static int calculateCost(String startWord, String endWord) {
    int cost = 0;
    for (int i = 0; i < startWord.length(); i++) {
        if (startWord.charAt(i) != endWord.charAt(i)) {
            cost++; // Tambahkan cost jika karakter tidak sama
        }
    }

    return cost;
}
```

Gambar 6. Snippet file UCS.java

Class UCS berisi method-method untuk keperluan Algoritma UCS.

Method	Deskripsi
public static List<String> Ucs(String start, String end, HashSet<String> words)	Method ini berfungsi untuk menjalankan algoritma UCS seperti pada subab 3.2
public static int calculateCost(String startWord, String endWord)	Method ini berfungsi untuk menghitung cost node sebagai perhitungan pemiiliihan rute pada algoritma UCS.

4.3 Class A*

```java
public static List<String> AStar(String start, String end, HashSet<String> words) {
    PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(Node::getCost));
    queue.add(new Node(start, cost:0));

    HashSet<String> visited = new HashSet<>();
    visited.add(start);

    HashMap<String, String> cameFrom = new HashMap<>();
    HashMap<String, Integer> costSoFar = new HashMap<>();
    costSoFar.put(start, value:0);

    while (!queue.isEmpty()) {
        Node current = queue.poll();

        if (current.getWord().equals(end)) {
            // Reconstruct path
            List<String> path = new ArrayList<>();
            while (current != null && current.getWord() != null) {
                path.add(current.getWord());
                if (cameFrom.get(current.getWord()) != null) {
                    current = new Node(cameFrom.get(current.getWord()), costSoFar.get(cameFrom.get(current.getWord())));
                } else {
                    current = null;
                }
            }
            Collections.reverse(path);
            return path;
        }

        for (String word : words) {
            if (GBFS.isOneLetterDifferentSamePosition(current.getWord(), word)) {
                int newCost = costSoFar.get(current.getWord()) + UCS.calculateCost(current.getWord(), word);
                if (!costSoFar.containsKey(word) || newCost < costSoFar.get(word)) {
                    costSoFar.put(word, newCost);
                    int priority = newCost + GBFS.levenshteinDistance(word, end);
                    queue.add(new Node(word, priority));
                    cameFrom.put(word, current.getWord());
                    visited.add(word);
                }
            }
        }
    }

    return null; // Return null if no path is found
```

Gambar 7. Snippet file Astar.java

Class Astar berisi method-method untuk keperluan algoritma Astar

Method	Deskripsi
public static List<String> AStar(String start, String end, HashSet<String> words)	Method ini berfungsi untuk menjalankan algoritma Astar seperti pada subab 3.3

4.4 Class Node

```java
public class Node {
    private String word;
    private int cost;

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

    public String getWord() {
        return word;
    }

    public int getCost() {
        return cost;
    }

    public void setCost(int cost) {
        this.cost = cost;
    }

}
```

Gambar 8. Snippet file Node.java

Class node berisi untuk menyimpan nilai string dan cost suatu node.

Method	Deskripsi
public Node(String word, int cost)	Constructor node
public String getWord()	Mendapatkan nilai word pada suatu node
public int getCost()	Mendapatkan cost suatu node
public void setCost(int cost)	Mengatur cost suatu node

BAB V
EKSPERIMEN

5.1 Algoritma GBFS

```
Enter start word: cat
Enter end word: dog
1. UCS
2. GBFS
3. A*
Pilih Algoritma: 2
Path: [cat, cot, cog, dog]
Cost: 3
Time: 37 ms
```

```
Enter start word: bird
Enter end word: song
1. UCS
2. GBFS
3. A*
Pilih Algoritma: 2
Path: [bird, bind, bond, bong, song]
Cost: 4
Time: 43 ms
```

```
Enter start word: wheat
Enter end word: bread
1. UCS
2. GBFS
3. A*
Pilih Algoritma: 2
No path found
Time: 23 ms
```

```
Enter start word: crawl
Enter end word: fight
1. UCS
2. GBFS
3. A*
Pilih Algoritma: 2
No path found
Time: 21 ms
```

```
Enter start word: break
Enter end word: point
1. UCS
2. GBFS
3. A*
Pilih Algoritma: 2
No path found
Time: 22 ms
```

```
Enter start word: zagging
Enter end word: wrathed
1. UCS
2. GBFS
3. A*
Pilih Algoritma: 2
No path found
Time: 21 ms
```

5.2 Algoritma UCS

```
Enter start word: cat
Enter end word: dog
1. UCS
2. GBFS
3. A*
Pilih Algoritma: 1
Path: [cat, cot, cog, dog]
Cost: 3
Time: 812 ms
```

```
Enter start word: bird
Enter end word: song
1. UCS
2. GBFS
3. A*
Pilih Algoritma: 1
Path: [bird, bind, bond, bong, song]
Cost: 4
Time: 1483 ms
```

```
Enter start word: wheat
Enter end word: bread
1. UCS
2. GBFS
3. A*
Pilih Algoritma: 1
Path: [wheat, cheat, cleat, bleat, bleak, break, bread]
Cost: 6
Time: 1012 ms
```

```
Enter start word: crawl
Enter end word: fight
1. UCS
2. GBFS
3. A*
Pilih Algoritma: 1
Path: [crawl, craws, crews, trews, trees, tyees, tynes, tines, sines, sinhs, sighs, sight, fight]
Cost: 12
Time: 19675 ms
```

```
Enter start word: break
Enter end word: point
1. UCS
2. GBFS
3. A*
Pilih Algoritma: 1
Path: [break, wreak, wreck, wrick, prick, prink, print, point]
Cost: 7
Time: 1671 ms
```

```
Enter start word: zagging
Enter end word: wrathed
1. UCS
2. GBFS
3. A*
Pilih Algoritma: 1
Path: [zagging, bagging, barging, parging, parking, perking, jerking, jerkins, jerkies, jerkier, perkier, peakier, peatier, p
latier, platter, clatter, chatter, chitter, whitter, whither, writher, writhed, wrathed]
Cost: 22
Time: 20688 ms
```

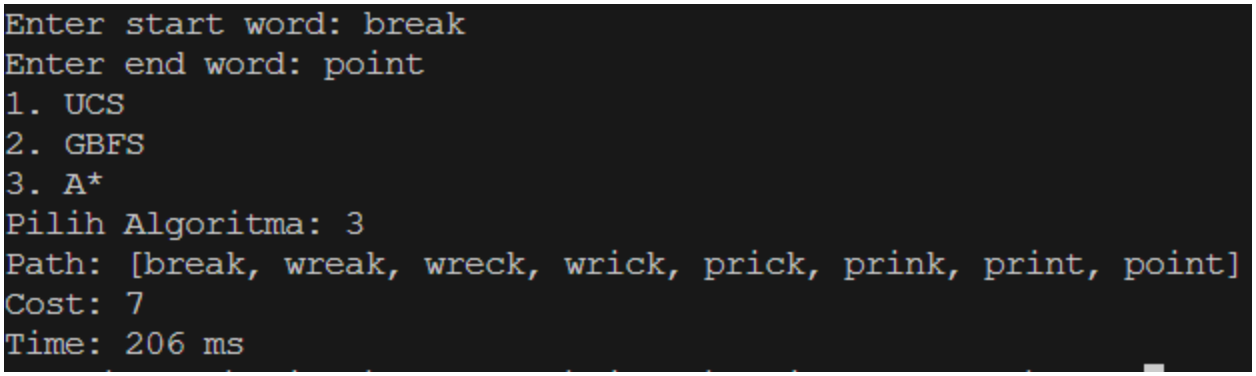
5.4 Algoritma A*

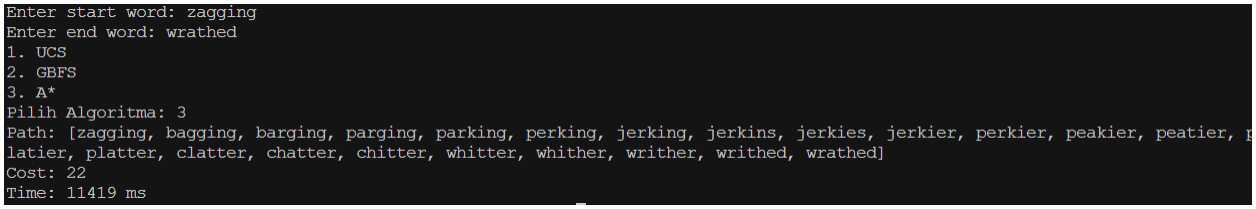
```
Enter start word: cat
Enter end word: dog
1. UCS
2. GBFS
3. A*
Pilih Algoritma: 3
Path: [cat, cot, cog, dog]
Cost: 3
Time: 48 ms
```

```
Enter start word: bird
Enter end word: song
1. UCS
2. GBFS
3. A*
Pilih Algoritma: 3
Path: [bird, bind, bond, bong, song]
Cost: 4
Time: 44 ms
```

```
Enter start word: weath
Enter end word: bread
1. UCS
2. GBFS
3. A*
Pilih Algoritma: 3
Path: [weath, wrath, wroth, troth, trots, trets, trees, treed, tread, bread]
Cost: 9
Time: 661 ms
```

```
Enter start word: crawl
Enter end word: fight
1. UCS
2. GBFS
3. A*
Pilih Algoritma: 3
Path: [crawl, brawl, braws, brads, beads, bends, binds, bines, sines, sinhs, sighs, sight, fight]
Cost: 12
Time: 4746 ms
```


Enter start word: break
Enter end word: point
1. UCS
2. GBFS
3. A*
Pilih Algoritma: 3
Path: [break, wreak, wreck, wrick, prick, prink, print, point]
Cost: 7
Time: 206 ms


Enter start word: zagging
Enter end word: wrathed
1. UCS
2. GBFS
3. A*
Pilih Algoritma: 3
Path: [zagging, bagging, barging, parging, parking, perking, jerking, jerkins, jerkies, jerkier, perkier, peakier, peatier,
latier, platter, clatter, chatter, chitter, whitter, whither, writher, writhed, wrathed]
Cost: 22
Time: 11419 ms

BAB VII
KESIMPULAN

Kesimpulan dari perbandingan solusi yang didapatkan dari ketiga Algoritma yang diujikan. Terdapat tiga penelaian yaitu optimalitas, waktu eksekusi ,dan memori yang diperlukan. Dalam penilaian optimalitas algoritma UCS merupakan penghasil sulusi yang optimal. Dapat dilihat hasil dari algoritma UCS selalu menghasilkan path dengan cost terendah. Contohnya pada testcase 3 wheat dan bread pada algoritma UCS didapatkan path dengan cost 6, sedangkan pada A* didapatkan cost 9, dan di GBFS tidak mendapatkan solusi. Dalam kecepatan sudah jelas GBFS merupakan Algoritma yang tercepat. Sedangkan yang terlambat merupakan algoritma UCS. Dalam penggunaan memori Algoritma UCS dan A* hampir sama, sedangkan GBFS paling kecil penggunaan memorynya. GBFS memiliki keunggulan di kecepatan dan penggunaan memori, tetapi Algoritma ini tidak bisa dikatakan yang paling efisisen karena dari testcase diatas sendiri Algorima ini hanya berhasil mendapatkan 2 dari 6 testcase. Jadi kesimpulannya algoritma yang paling efisien adalah A* karena memiliki keseimbangan di segala aspek.

LAMPIRAN

1. Github Link: https://github.com/ozarabal/Tucil3_13522159
2.

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

3.