Laporan Tugas Kecil 3

IF2211 Strategi Algoritma

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

Rafiki Prawhira Harianto 13522065

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

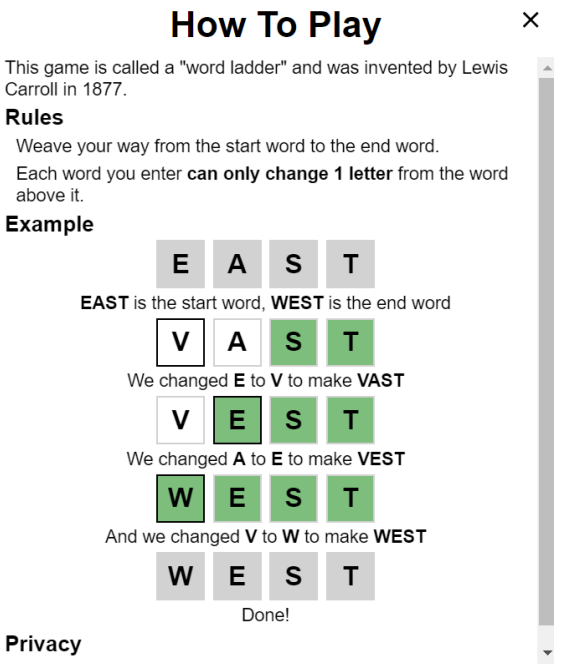
2024

Daftar Isi

Daftar Isi **2**
Deskripsi Persoalan **3**
Algoritma Route Planning **4**
Implementasi Algoritma Route Planning **5**
Hasil Program dan Analisis **6**
Lampiran **7**

Deskripsi Persoalan

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.



Gambar 1. Ilustrasi dan Peraturan Permainan Word Ladder

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

Algoritma Route Planning

Permainan word ladder dapat diseleasikan menggunakan pendekatan route planning. Terdapat dua kategori algoritma route planning. *Uninformed* search, yaitu pencarian tanpa informasi tambahan graf, serta *Informed* search, pencarian dengan informasi tambahan graf. Algoritma yang dipakai dalam tucil ini adalah UCS (Uniform Cost Search) yang termasuk Uninformed search, serta GBFS (Greedy Best First Search) dan A* Search yang termasuk Informed search.

UCS, GBFS, dan A* Search memiliki algoritma umum yang serupa. Pembeda ketiga algoritma tersebut berupa nilai *cost* atau f(n). Terdapat 2 variabel utama dalam cost, yaitu g(n) dan h(n). g(n) ditentukan berdasarkan jumlah perubahan dari awal hingga node, sedangkan h(n) berupa fungsi heuristik yang admissible. Dalam tucil ini, kedua variabel tersebut berupa:

$g(n)$: Jumlah perubahan dari word awal hingga word n

$h(n)$: Banyaknya huruf berbeda dari word n dengan word akhir

Hal yang perlu diperhatikan dalam menentukan h(n) berupa syaratnya, yaitu fungsi heuristik h(n) harus *admissible*. Dalam kasus ini, h(n) termasuk *admissible* karena h(n) selalu *overestimate* jarak asli word n dengan word akhir, yaitu h(n) selalu memiliki nilai yang lebih kecil (dalam konteks mencari minimum cost) dari jarak aslinya. Hal ini dapat dilihat dari perubahan huruf langsung ke huruf target tidak menjamin adanya kata tersebut dalam kamus bahasa, mengakibatkan word tersebut perlu melewati huruf lain, sehingga jarak asli pasti lebih besar dari h(n). Sesuai teorema, karena h(n) *admissible*, pencarian algoritma A* terjamin optimal.

Node diproses berdasakan prioritas *cost* yang paling kecil. Nilai *cost* yang dari masing-masing algoritma adalah sebagai berikut.

UCS : $f(n) = g(n)$

GBFS : $f(n) = g(n)$

A* : $f(n) = g(n) + h(n)$

Pada kasus word ladder, algoritma UCS sama dengan BFS (Breadth First Search) karena g(n) setiap node baru pasti hanya berbeda 1 g(n) dari node awal. Hal ini sama dengan membangkitkan semua node dengan *depth* sama terlebih dahulu dalam algoritma BFS, sehingga urutan node yang dibangkitkan dan path UCS sama dengan BFS.

Secara teoritis, algoritma A* lebih efisien daripada UCS dalam word ladder. Karena h(n) *admissible*, A* menjamin keoptimalan solusi dan karena ada *cost* heuristik h(n) membuat pencarian A* lebih terarah menuju solusi word akhir. Namun, GBFS tidak menjamin keoptimalan solusi. GBFS tidak ada *cost* g(n), membuat lingkup pencarian sangat terfokus terhadap word akhir dan membuat pencarian yang cepat, tetapi tidak adanya *cost* g(n) membuat GBFS pencarian yang tidak sistematis dan mudah terperangkap dalam minimum lokal.

Implementasi Algoritma Route Planning

Implementasi algoritma route planning dalam tucil ini menggunakan berbagai kelas buatan, yaitu DictReader, Node, WordQueue, dan Algorithm. DictReader berupa kelas yang membaca file dictionary.txt ke dalam HashSet. Node merupakan kelas serupa struktur data untuk menyimpan word, g(n) dan h(n), dan daftar parentnya. WordQueue merupakan kelas berisi String start dan end, HashMap graf yang menyimpan semua node berisi word dalam kondisi awal, PriorityQueue buffer untuk menyimpan pemrosesan queue utama dan diinisialisasi dengan word start dan algoritma route planning terkait, dan HashSet dict untuk semua word dalam dict yang telah difilter sesuai jumlah huruf start.

Dalam Main hanya memanggil fungsi-fungsi yang berada dalam WordQueue dan DictReader, serta format input dan berberapa output. Selain itu, terdapat kelas abstak Algorithm yang dipakai dalam PriorityQueue buffer, yang dipakai oleh kelas UCS, GBFS, dan A*. UCS berupa kelas Algorithm dengan f(n) = n.g, GBFS dengan f(n) = n.h, dan A* dengan f(n) = n.g + n.h, sehingga PriorityQueue dapat diatur sesuai algoritma yang diperlukan.

Secara garis besar, langkah-langkah program adalah sebagai berikut:

1. Ambil kamus dari file menggunakan DictReader, lalu terima input start word, end word, dan jenis algoritma
2. Jika start word dan end word sama, filter hasil DictReader sesuai jumlah hurufnya.
3. Simpan hasilnya di dalam konstruktor WordQueue, lalu iterasi setiap kata dalam dict untuk inisalisasi graf dan buffer berisi start word.
4. Hingga buffer WordQueue kosong atau buffer[0] berisi end word, lakukan pemrosesan WordQueue.
5. Jika buffer kosong, solusi tidak ditemukan. Jika buffer[0] berisi end word, solusi ditemukan.
6. Lakukan print keluaran sesuai solusi yang dihasilkan

Langkah dalam pemrosesan WordQueue sebagai berikut:

1. Ambil Node buffer[0] sebagai curr
2. Cari word-word berikutnya dalam dict berdasarkan curr.word, simpan dalam nextWords
3. Tambahkan curr.word dalam list visited
4. Untuk setiap word dalam nextWords, jika word tidak ada di dalam visited:
    a) Cari node word dalam graf,
    b) Ubah node.g dengan curr.g +1,
    c) Ambil curr.thread + curr.word dan simpan dalam node.thread,
    d) Tambahkan buffer dengan n sesuai PriorityQueue,
    e) Tambahkan visited dengan word.

Hasil Program dan Analisis

No TC	UCS	GBFS	A*
1. word -> ladder	Start word count is not equal to end word	Start word count is not equal to end word	Start word count is not equal to end word
2. ionospherically -> prelocalization	Solution cannot be reached from start word Visited count: 1 Time taken: 61ms	Solution cannot be reached from start word Visited count: 1 Time taken: 58ms	Solution cannot be reached from start word Visited count: 1 Time taken: 61ms
3. great -> break	Visited count: 469 Steps count: 3 Time taken: 74ms	Visited count: 21 Steps count: 3 Time taken: 66ms	Visited count: 30 Steps count: 3 Time taken: 57ms
4. atlases -> cabaret	Visited count: 12444 Steps count: 45 Time taken: 547ms	Visited count: 1948 Steps count: 81 Time taken: 163ms	Visited count: 11573 Steps count: 45 Time taken: 463ms
5. winter -> summer	Visited count: 6917 Steps count: 6 Time taken: 257ms	Visited count: 228 Steps count: 10 Time taken: 87ms	Visited count: 690 Steps count: 6 Time taken: 76ms
6. passing -> surgery	Visited count: 5616 Steps count: 13 Time taken: 289ms	Visited count: 1039 Steps count: 27 Time taken: 116ms	Visited count: 1937 Steps count: 13 Time taken: 158ms

Berdasarkan hasil yang didapatkan, terlihat berberapa pola terkait ketiga algoritma route planning. Untuk visited count, yaitu memory yang dipakai, paling sedikit berada pada GBFS, lalu A*, dan yang paling buruk UCS. Selain itu, steps count atau jumlah lompatan dari word awal ke akhir optimal dimiliki UCS dan A*. Untuk time taken berbanding lurus dengan visited count, jika visited count cukup besar.

Hasil yang didapatkan memperlihatkan karakteristik algoritma route planing. UCS merupakan algoritma optimal, tetapi lambat dan memakan memori. GBFS berupa algoritma cepat dengan memori kecil, tetapi tidak optimal. A* dengan heuristik *admissible* menggabungkan keunggulan keduanya dengan pencarian solusi yang optimal, sementara memiliki kecepatan dan penggunaan memori yang lebih baik dengan UCS.

Lampiran

A. Tabel checklist

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

B. Link Repository: https://github.com/Intermaze/Tucil3_13522065

C. Source program java

Main.java

```java
import java.util.Comparator;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Scanner;

public class Main{
    private static HashSet<String> filterByLength(HashSet<String> dict, int length){
        HashSet<String> filter = new HashSet<String>();
        Iterator<String> wordItr = dict.iterator();
        while(wordItr.hasNext()){
            String nextWord = wordItr.next();
            if (nextWord.length() == length) filter.add(nextWord);
        }
        return filter;
    }

    public static void main(String[] args) {
        System.out.println("Loading dictionary.txt...");
```

```
        DictReader dict = new DictReader("dictionary.txt");
        Scanner in = new Scanner(System.in);
        String start, end;
        int algorithm;

        System.out.print("\033[H\033[2J");
        System.out.flush();

        System.out.println("Welcome to word Ladder! ");

        System.out.print("Start word: ");
        start = in.nextLine();

        System.out.print("End word: ");
        end = in.nextLine();

        System.out.println("====== Algorithm List ======");
        System.out.println("1: UCS (Uniform Cost Search)");
        System.out.println("2: GBFS (Greedy Best First Search)");
        System.out.println("3: A* Search");
        System.out.println("============================");
        System.out.print("Algorithm to use: ");
        algorithm = in.nextInt();


        if (start.length() != end.length()){
            System.err.println("Start word letter count is not equal to end word");
        }
        else if (!dict.getDict().contains(start)){
            System.err.println("Start word is not found inside the dictionary");
        }
        else if (!dict.getDict().contains(end)){
            System.err.println("End word is not found inside the dictionary");
        }
        else{
            long startTime = System.currentTimeMillis();

            HashSet<String> filteredDict = filterByLength(dict.getDict(),
start.length());
            WordQueue wq;

            if (algorithm == 1){
                wq = new WordQueue(filteredDict, new UCS(), start, end);
                System.out.println("Using UCS...");
            }
            else if (algorithm == 2){
                wq = new WordQueue(filteredDict, new GBFS(), start, end);
```

```
                System.out.println("Using GBFS...");
            }
            else{
                //Kalau input selain 1 dan 2, automatis dipilih algoritma A*
                wq = new WordQueue(filteredDict, new Astar(), start, end);
                System.out.println("Using A* Search...");
            }

            while (!wq.isDone()){
                wq.processNext();
            }

            if (wq.bufferIsEmpty()){
                System.out.println("Solution cannot be reached from start word.");
                wq.printVisitedOnly();
            }
            else{
                wq.printSolution();
            }
            long endTime = System.currentTimeMillis();
            System.out.println("Time taken:  " + (endTime - startTime) + "ms");
        }
        in.close();
    }
}

abstract class Algorithm implements Comparator<Node>{
    abstract int fn(Node n);

    public int compare(Node n1, Node n2){
        if (fn(n1) > fn(n2)) return 1;
        else if (fn(n1) < fn(n2)) return -1;
        else return 0;
    }
}

class UCS extends Algorithm{
    int fn(Node n){
        return n.g;
    }
}

class GBFS extends Algorithm{
    int fn(Node n){
        return n.h;
    }
}
```

```
class Astar extends Algorithm{
    int fn(Node n){
        return n.g + n.h;
    }
}
```

Node.java

```
import java.util.LinkedList;
import java.util.Queue;

public class Node {
    public String word;
    public int g, h;
    public Queue<String> thread;

    public Node(String word, int g, int h){
        this.word = word;
        this.g = g;
        this.h = h;
        this.thread = new LinkedList<String>();
    }

    public void pushThread(String w){
        thread.add(w);
    }

    public void copyThread(Queue<String> newThread){
        this.thread = new LinkedList<String>(newThread);
    }

    public String popThread(){
        return thread.remove();
    }

}
```

WordQueue.java

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.HashMap;
```

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

public class WordQueue {
    private HashMap<String, Node> graph;
    private PriorityQueue<Node> buffer;
    private Set<String> visited;
    private HashSet<String> dict;
    private String end;

    public WordQueue(HashSet<String> dict, Comparator<Node> alg, String start, String
end){
        this.graph = new HashMap<String, Node>();
        this.buffer = new PriorityQueue<Node>(alg);
        this.visited = new HashSet<String>();
        this.dict = dict;
        this.end = end;

        int g,h;

        Iterator<String> wordItr = this.dict.iterator();

        while (wordItr.hasNext()){
            String nextWord = wordItr.next();
            g = 0;
            h = diffLetters(nextWord, end);

            Node node = new Node(nextWord, g, h);
            this.graph.put(node.word, node);

            if (node.word.equals(start)){
                this.buffer.add(node);
            }
        }

    }

    public void processNext(){
        Node curr = this.buffer.poll();
        ArrayList<String> nextWords = findNext(curr.word);
        visited.add(curr.word);
        for (String w : nextWords){
            if (!visited.contains(w)){
                Node n = getNodeInGraph(w);
                n.g = curr.g + 1;
```

```
                n.copyThread(curr.thread);
                n.pushThread(curr.word);
                buffer.add(n);
                visited.add(w);
            }
        }
    }

    public boolean isDone(){
        if (bufferIsEmpty()) return true;
        else{
            //Kalau tidak kosong, cek dulu kalau ketemu solusi
            if (buffer.peek().word.equals(end)){
                return true;
            }
            //Kalau belum ketemu, lanjut dengan return false
            return false;
        }
    }

    //Prekondisi: wq.isDone()
    public void printSolution(){
        Node solution = this.buffer.poll();
        int steps = solution.thread.size();

        while (!solution.thread.isEmpty()){
            System.out.print(solution.popThread() + " -> ");
        }
        System.out.println(solution.word);
        System.out.println("Visited count: " + visited.size());
        System.out.println("Steps count: " + steps);
    }

    public void printVisitedOnly(){
        System.out.println("Visited count: " + visited.size());
    }

    public boolean bufferIsEmpty(){
        return buffer.isEmpty();
    }

    //Prekondisi: panjang string a dan b sama
    private int diffLetters(String a, String b){
        int count = 0;
        for (int i=0; i<a.length(); i++){
            if (a.charAt(i) != b.charAt(i)){
                count++;
```

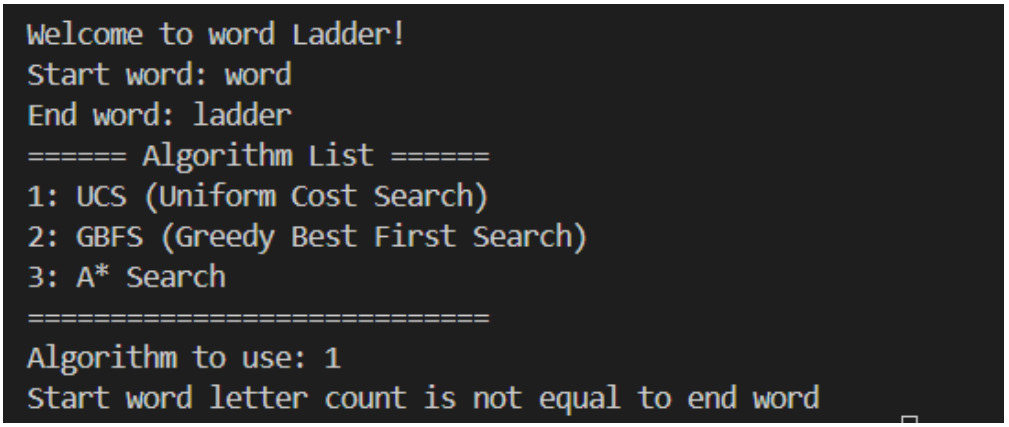
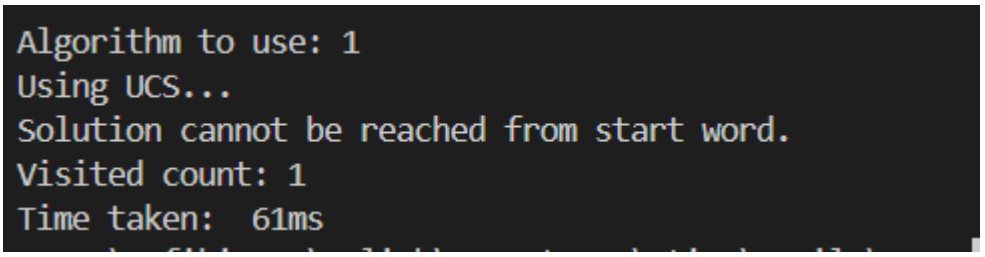
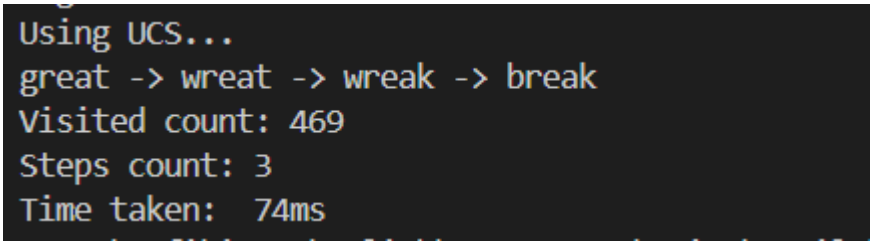
```
            }
        }
        return count;
    }

    private ArrayList<String> findNext(String word){
        ArrayList<String> next = new ArrayList<String>();
        String temp;
        for(int i=0; i<word.length(); i++){
            for (char aToz : "abcdefghijklmnopqrstuvwxyz".toCharArray()){
                temp = word.substring(0, i) + aToz + word.substring(i+1);
                if (this.dict.contains(temp) && !temp.equals(word)){
                    next.add(temp);
                }
            }
        }
        return next;
    }

    // Prekondisi: word pasti ada di dalam graph
    private Node getNodeInGraph(String word){
        return graph.get(word);
    }

}
```

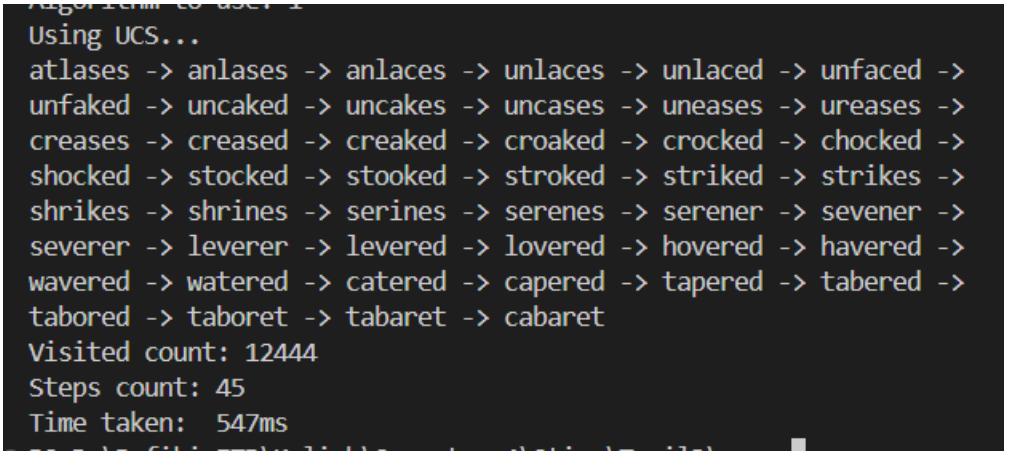
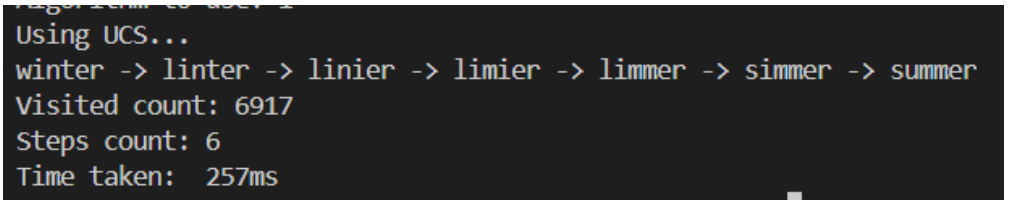
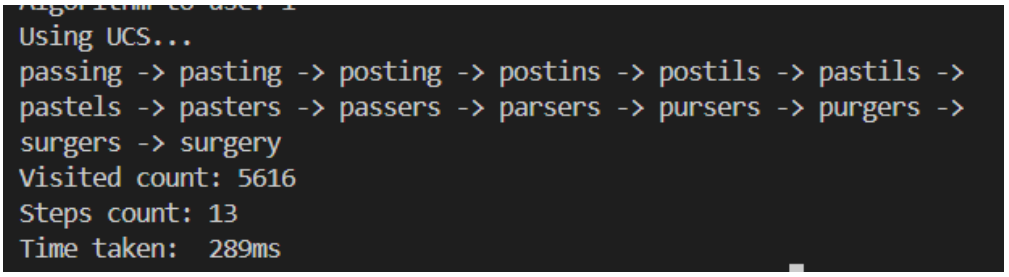
DictReader.java

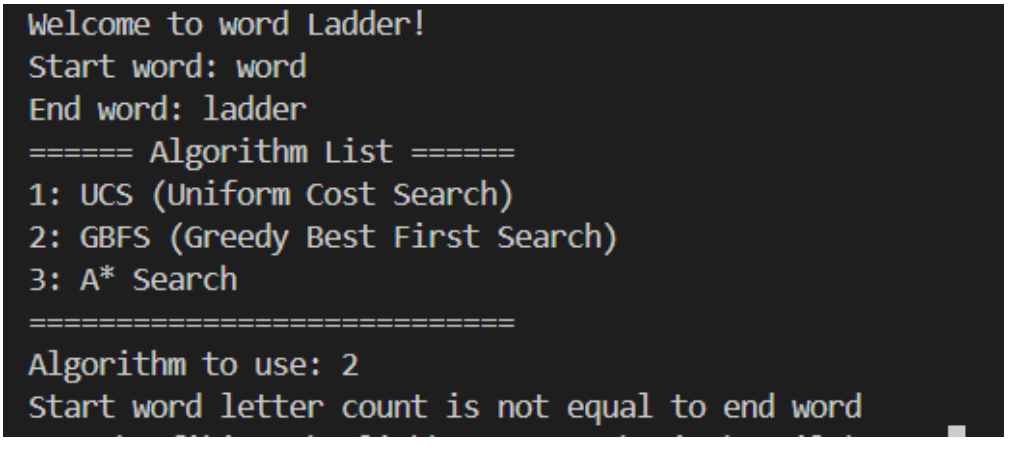
```
import java.io.File;
import java.util.HashSet;
import java.util.Scanner;

public class DictReader {
    private HashSet<String> dict;

    public DictReader(String filename){
        try{
            Scanner s = new Scanner(new File(filename));
            this.dict = new HashSet<String>();
            while (s.hasNext()){
                this.dict.add(s.next());
            }
            s.close();
        }
        catch(Exception e){
```

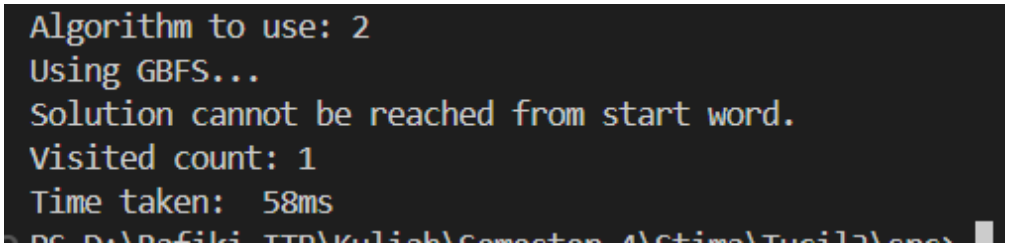
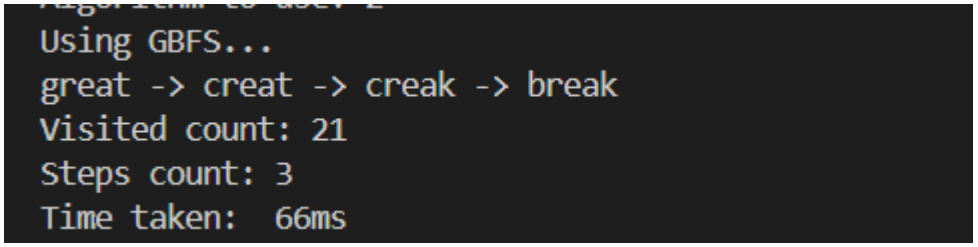
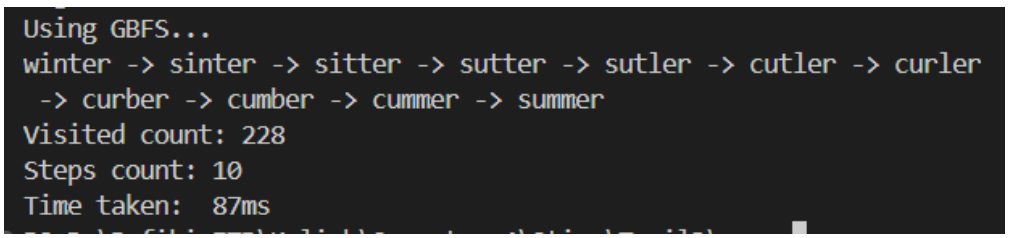
```
            System.out.println("File dictionary tidak ditemukan.");
            System.out.println(e.getMessage());
            System.exit(0);
        }
    }

    public HashSet<String> getDict(){
        return this.dict;
    }

}
```

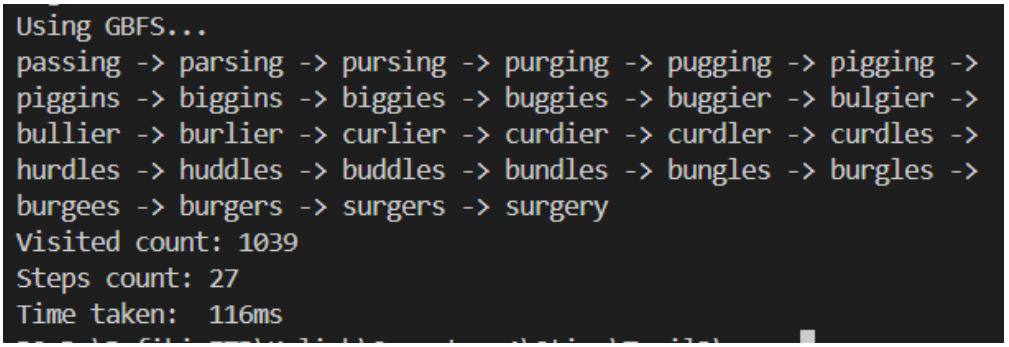
D. Test Case

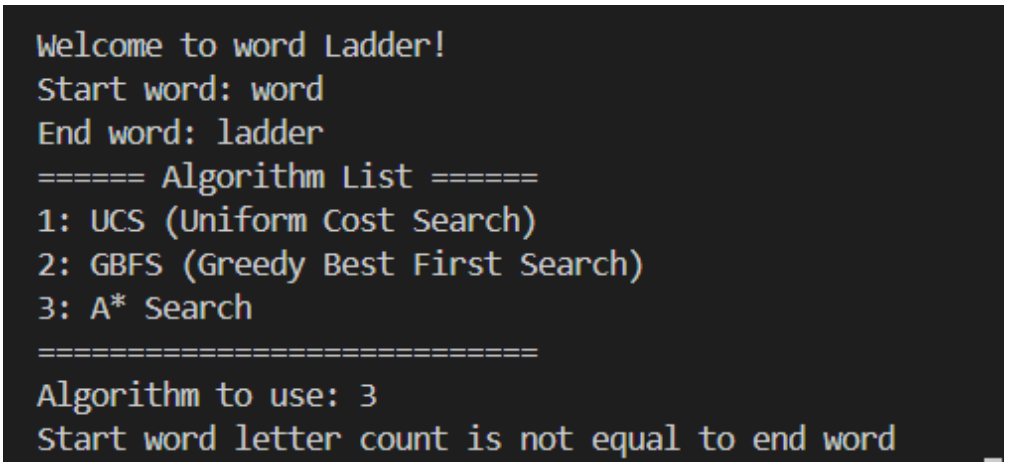
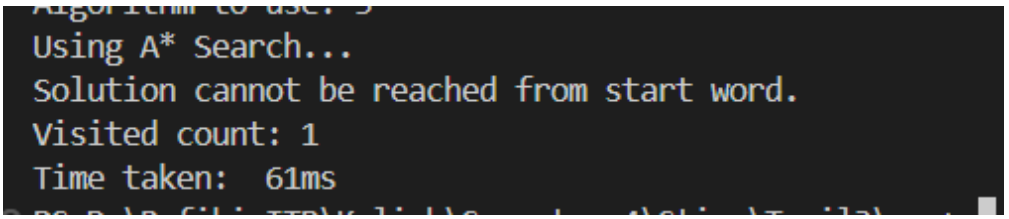
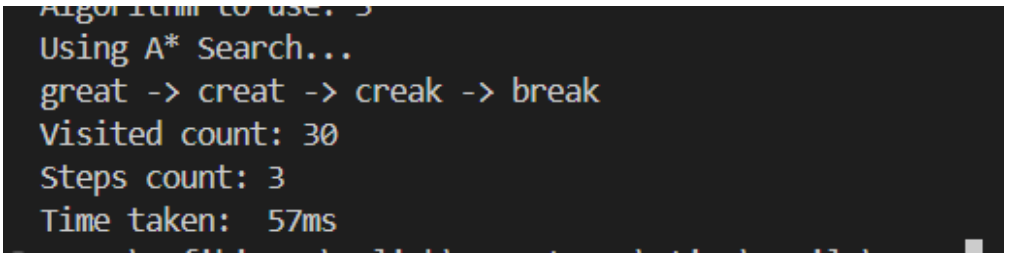
No TC	UCS
1. word -> ladder	
2. ionospheri cally -> prelocaliza tion	
3. great -> break	

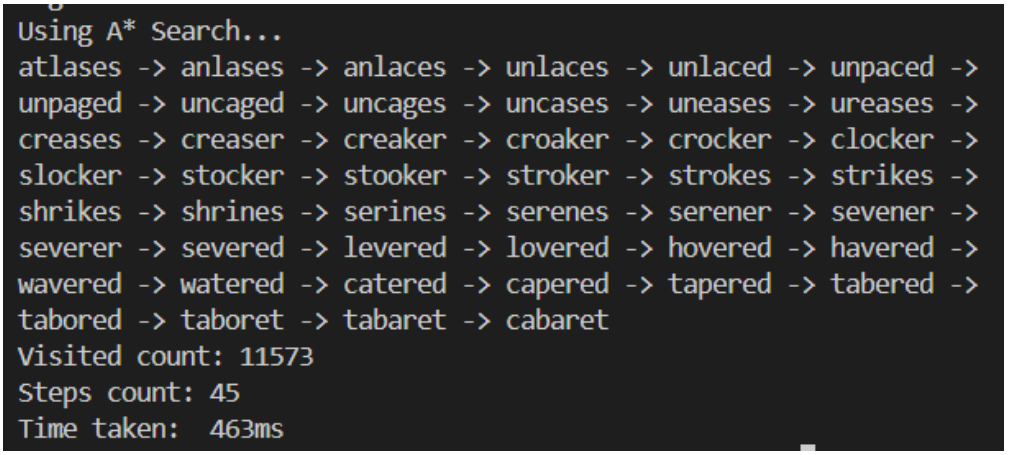
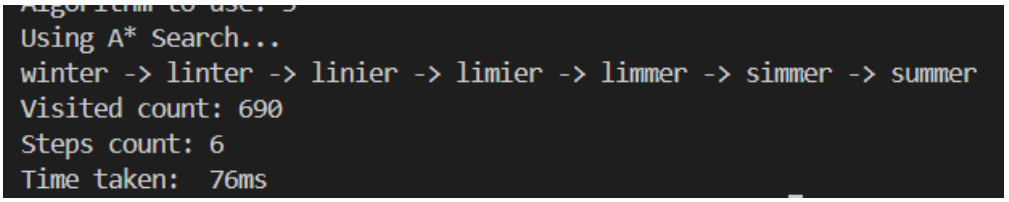
4. atlases -> cabaret	
5. winter -> summer	
6. passing -> surgery	

No TC	GBFS
1. word -> ladder	

2. ionospheri cally -> prelocaliza tion	
3. great -> break	
4. atlases -> cabaret	
5. winter -> summer	

6. passing -> surgery	

No TC	A*
1. word -> ladder	
2. ionospheri cally -> prelocaliza tion	
3. great -> break	

4. atlases -> cabaret	
5. winter -> summer	
6. passing -> surgery