

LAPORAN TUGAS KECIL III

IF2211 STRATEGI ALGORITMA



Disusun oleh:

Raden Francisco Trianto Bratadiningrat

13522091

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2024

DAFTAR ISI

DAFTAR ISI.....	2
DAFTAR GAMBAR.....	3
DAFTAR TABEL.....	4
BAB I DESKRIPSI TUGAS.....	5
BAB II LANDASAN TEORI.....	7
2.1 Algoritma Uniform Cost Search.....	7
2.2 Algoritma Greedy Best First Search.....	7
2.3 Algoritma A*.....	8
BAB III ANALISIS DAN IMPLEMENTASI.....	10
3.1 Algoritma Uniform Cost Search.....	10
3.1.1 Implementasi.....	10
3.1.2 Analisis.....	11
3.2 Algoritma Greedy Best First Search.....	11
3.2.1 Implementasi.....	11
3.2.2 Analisis.....	12
3.3 Algoritma A*.....	13
3.3.1 Implementasi.....	13
3.3.2 Analisis.....	14
3.4 Graphical User Interface.....	14
BAB IV SOURCE CODE.....	16
4.1 Repository Program.....	16
4.2 Class dan Method.....	16
4.2.1 Class Pair.....	16
4.2.2 Class Dictionary.....	16
4.2.3 Class Graph.....	17
4.2.4 Class AStarSearch.....	17
4.2.5 Class GreedyBestFirstSearch.....	18
4.2.6 Class UniformCostSearch.....	18
4.3 Source Code Program.....	19
4.3.1 Main.java.....	19
4.3.2 Dictionary.java.....	28
4.3.3 Graph.java.....	30
4.3.4 Pair.java.....	31
4.3.5 AStarSearch.java.....	32
4.3.6 GreedyBestFirstSearch.java.....	34

4.3.7 UniformCostSearch.java.....	36
BAB V ANALISIS DAN PENGUJIAN.....	39
5.1 Pengujian.....	39
5.1.1 Tests Case 1.....	39
5.1.2 Test Case 2.....	40
5.1.3 Test Case 3.....	42
5.1.4 Test Case 4.....	44
5.1.5 Test Case 5.....	46
5.1.6 Test Case 6.....	49
5.2 Analisis Pengujian.....	51
5.2.1 Optimalitas.....	51
5.2.2 Waktu Eksekusi.....	52
5.2.3 Memori.....	53
BAB VI KESIMPULAN DAN SARAN.....	55
6.1 Kesimpulan.....	55
6.2 Saran.....	55
6.3 Refleksi.....	55
LAMPIRAN.....	56
DAFTAR PUSTAKA.....	57

DAFTAR GAMBAR

Gambar 1.1 Ilustrasi dan Peraturan Permainan Word Ladder

Gambar 2.1.1 Graf Uniform Cost Search

Gambar 2.2.1 Graf Greedy Best First Search

Gambar 2.3.1 Graf A*

Gambar 2.3.2 Pencarian A*

Gambar 3.4.1 Menu Program

Gambar 3.4.2 Contoh Hasil Pencarian Program

DAFTAR TABEL

Tabel 4.2.1.1 Atribut Kelas Pair

Tabel 4.2.1.2 Method Kelas Pair

Tabel 4.2.2.1 Atribut Kelas Dictionary

Tabel 4.2.2.2 Method Kelas Dictionary

Tabel 4.2.3.1 Atribut Kelas Graph

Tabel 4.2.3.2 Method Kelas Graph

Tabel 4.2.4.1 Method Kelas AStarSearch

Tabel 4.2.5.1 Method Kelas GreedyBestFirstSearch

Tabel 4.2.6.1 Method Kelas UniformCostSearch

Tabel 5.2.1 Data panjang rute hasil pengujian

Tabel 5.2.1 Data panjang rute hasil pengujian

Tabel 5.2.3.1 Data aproksimasi memori hasil pengujian

BAB I

DESKRIPSI TUGAS

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

How To Play

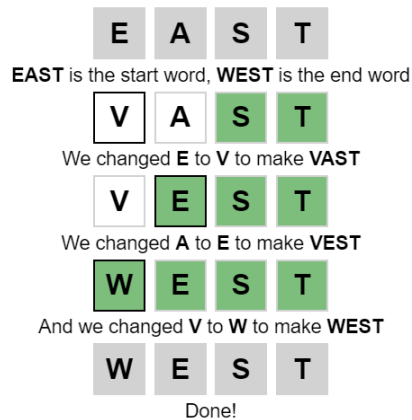
This game is called a "word ladder" and was invented by Lewis Carroll in 1877.

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



Gambar 1.1 Ilustrasi dan Peraturan Permainan Word Ladder

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

Permainannya cukup sederhana bukan? Jika belum paham dengan peraturan permainannya, cobalah untuk memainkan permainannya pada link sumber di atas. Jika sudah

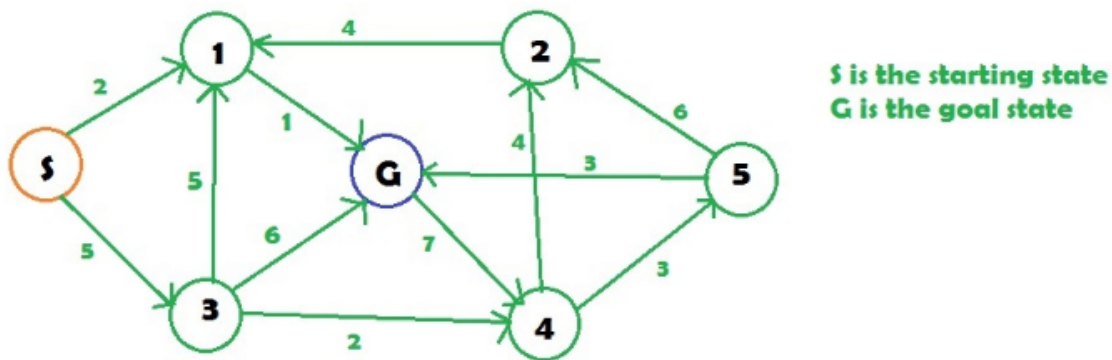
paham dengan permainannya, sekarang adalah waktunya kalian untuk membuat sebuah solver permainan tersebut dengan harapan kita dapat menemukan solusi paling optimal untuk menyelesaikan permainan Word Ladder ini.

BAB II

LANDASAN TEORI

2.1 Algoritma Uniform Cost Search

Uniform Cost Search atau UCS merupakan salah satu algoritma pencarian rute yang merupakan variasi dari algoritma Dijkstra. UCS melakukan pencarian dengan menghitung cost untuk menuju semua simpul yang mungkin dicapai dan memilih cost yang paling baik untuk mendapatkan hasil yang paling optimal.



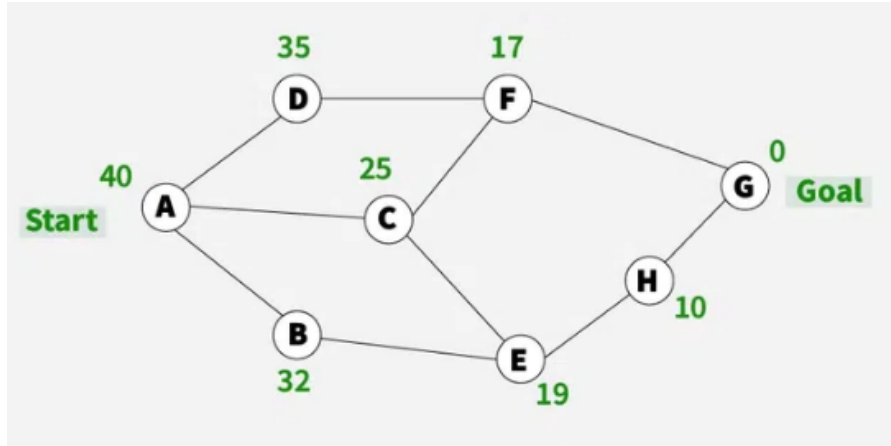
Gambar 2.1.1 Graf Uniform Cost Search

(Sumber: <https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/>)

Algoritma UCS dimulai dengan menambahkan simpul akar ke dalam priority queue. Kemudian simpul akar akan dikunjungi dan dilakukan pengecekan apakah target sudah dicapai, karena belum maka UCS akan menambahkan simpul-simpul yang bertetangga dari simpul akar ke dalam priority queue dengan nilai cost yang dihitung. Priority Queue pada UCS disortir berdasarkan simpul yang memiliki cost yang paling kecil. UCS kemudian mengunjungi simpul dengan cost paling kecil dan mengulangi langkah-langkah seperti yang dilakukan pada simpul akar hingga target ditemukan atau seluruh graph sudah dikunjungi.

2.2 Algoritma Greedy Best First Search

Greedy Best First Search atau GBFS merupakan algoritma pencarian rute yang mencari solusi berdasarkan strategi Greedy yaitu menggunakan solusi lokal yang paling baik walaupun belum tentu solusi global yang dihasilkan merupakan solusi yang paling optimal.



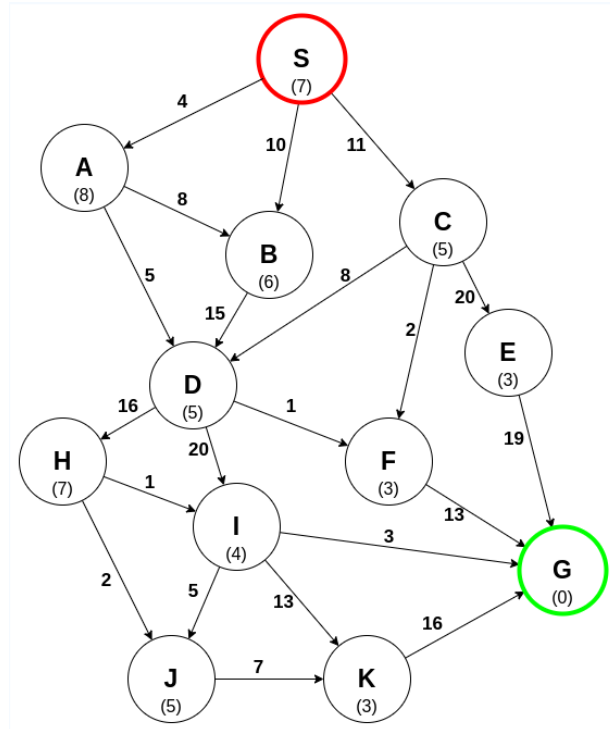
Gambar 2.2.1 Graf Uniform Cost Search

(Sumber: <https://www.geeksforgeeks.org/greedy-best-first-search-algorithm>)

GBFS menentukan solusi lokal terbaik berdasarkan fungsi heuristik untuk menentukan seberapa baik suatu simpul untuk mencapai suatu target. Pada gambar 2.2.1, hasil pencarian rute menggunakan algoritma GBFS adalah A - C - F - G dengan total nilai heuristik adalah $40 + 25 + 17 + 0 = 82$. Namun rute tersebut belum tentu adalah solusi optimal global karena sifat GBFS yang memilih rute berdasarkan solusi optimal lokal.

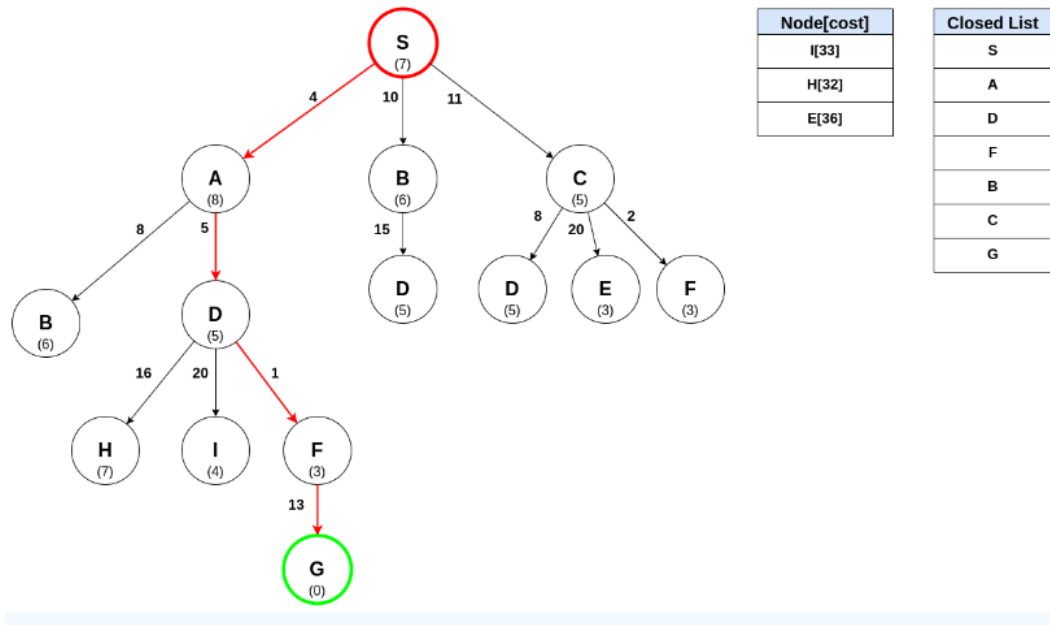
2.3 Algoritma A*

A* (A Star) adalah salah satu algoritma pencarian terbaik dalam pencarian rute dan traversal graf. Algoritma A* memilih rute terbaik berdasarkan nilai f yang dihitung berdasarkan ketentuan tertentu. Nilai f setara dengan nilai heuristik pada algoritma GBFS, namun nilai f berasal dari dua parameter, yaitu g (cost untuk mencapai suatu simpul dari simpul akar) dan h (cost untuk mencapai simpul target). Mirip dengan algoritma Dijkstra, A* menggunakan dua list berbeda, satu untuk menyimpan path saat ini, dan satu untuk menyimpan nilai f untuk setiap simpul yang dapat dikunjungi.



Gambar 2.3.1 Graf A*

(Sumber: <https://www.codecademy.com/resources/docs/ai/search-algorithms/a-star-search>)



Gambar 2.3.2 Pencarian A*

(Sumber: <https://www.codecademy.com/resources/docs/ai/search-algorithms/a-star-search>)

BAB III

ANALISIS DAN IMPLEMENTASI

3.1 Algoritma Uniform Cost Search

3.1.1 Implementasi

Dalam penggunaan algoritma Uniform Cost Search untuk mencari rute terbaik dari suatu kata asal menuju kata target, penulis menggunakan cost yang merupakan jarak dari kata asal menuju suatu kata tujuan. Berikut adalah langkah-langkah algoritma Uniform Cost Search yang diimplementasikan.

1. Sebelumnya program sudah membuat graf yang terbuat dari simpul berupa kata-kata yang memiliki panjang kata sesuai dengan panjang kata masukkan (kata asal dan target), dan sisi berupa hubungan dua kata yang memiliki perbedaan salah satu karakternya.
2. Program membuat priority queue yang diurutkan berdasarkan cost dari suatu simpul.
3. Program membuat map visited untuk mencatat simpul-simpul yang sudah pernah dikunjungi dan cost saat dikunjungi.
4. Program kemudian memasukan simpul akar yang dibuat dari kata asal dan cost yang bernilai 0.
5. Selama priority queue tidak kosong, simpul dengan cost paling rendah akan diambil dari priority queue.
6. Simpul tersebut dibandingkan dengan kata yang dicari atau kata target.
7. Jika simpul sudah sesuai dengan kata target, maka program akan berhenti dan mengembalikan rute hasil pencarian.
8. Jika simpul belum sesuai maka simpul tersebut akan dimasukkan kedalam map visited.
9. Program kemudian akan menambahkan semua simpul anak ke dalam priority queue dengan syarat simpul tersebut belum pernah dikunjungi.

10. Jika simpul anak sudah pernah dikunjungi maka program akan melakukan pengecekan jika cost simpul anak saat pengunjungan ini lebih rendah daripada cost pada pengunjungan sebelumnya. Jika iya maka map visited untuk simpul tersebut akan diubah costnya.
11. Program akan mengulangi langkah 5 hingga 9 sampai priority queue kosong, atau sebuah solusi ditemukan.

3.1.2 Analisis

Dalam persoalan Word Ladder, algoritma UCS menggunakan cost yang berupa jarak langkah dari kata asal menuju kata tujuan. Hal tersebut dikarenakan Word Ladder tidak memiliki opsi cost yang lainnya karena permainan hanya memperbolehkan perubahan satu karakter hingga semua karakter berubah menjadi kata target. Hal ini menyebabkan implementasi algoritma UCS menjadi mirip dengan algoritma BFS. BFS melakukan pencarian secara luas atau melakukan pengecekan semua simpul anak dari simpul sebelum mengunjungi anak dari simpul anak.

Karena BFS sebenarnya dapat disamakan dengan UCS yang memiliki cost satu untuk setiap langkahnya, maka dapat dibilang BFS dan UCS dalam persoalan Word Ladder adalah sama. Namun perlu diperhatikan implementasi UCS berbeda dengan BFS karena penggunaan priority queue dan cost sedangkan BFS melakukan pencarian tanpa menyimpan cost. Urutan pengunjungan simpul antara BFS dan UCS sama, karena cost untuk UCS yang bernilai satu per langkah. Perhitungan cost tersebut tidak baik dalam benar-benar menggunakan algoritma UCS dengan se penuh efektivitasnya.

3.2 Algoritma Greedy Best First Search

3.2.1 Implementasi

Dalam penggunaan algoritma Greedy Best First Search untuk mencari rute terbaik dari suatu kata asal menuju kata target, penulis menggunakan nilai $f(n)$ sebagai nilai heuristik yang dihitung dari perbedaan antara kata suatu simpul dengan kata target. Semakin banyak perbedaan karakter antar kata, maka semakin besar nilai heuristiknya. Berikut adalah langkah-langkah algoritma Greedy Best First Search yang diimplementasikan.

1. Sebelumnya program sudah membuat graf yang terbuat dari simpul berupa kata-kata yang memiliki panjang kata sesuai dengan panjang kata masukkan (kata asal dan target), dan sisi berupa hubungan dua kata yang memiliki perbedaan salah satu karakternya.
2. Program membuat priority queue yang diurutkan berdasarkan nilai heuristik suatu simpul yang merupakan jumlah perbedaan karakter antara simpul dengan kata target.
3. Program membuat set “visited” untuk mencatat simpul-simpul yang sudah pernah dikunjungi.
4. Program membuat list result untuk mencatat rute yang diambil
5. Program kemudian memasukan simpul akar yang dibuat dari kata asal dan nilai heuristiknya.
6. Selama priority queue tidak kosong, simpul dengan nilai heuristik paling rendah akan diambil dari priority queue dan setelah diambil priority queue dikosongkan.
7. Kata tersebut dimasukan ke dalam set visited dan list result.
8. Simpul tersebut kemudian dibandingkan dengan kata yang dicari atau kata target.
9. Jika simpul sudah sesuai dengan kata target, maka program akan berhenti dan mengembalikan rute hasil pencarian.
10. Jika belum, program kemudian akan menambahkan semua simpul anak yang belum pernah dikunjungi ke dalam priority queue dengan nilai heuristik masing-masing simpul.
11. Program akan mengulangi langkah 6 hingga 10 sampai priority queue kosong, atau sebuah solusi ditemukan.

3.2.2 Analisis

Dalam persoalan Word Ladder, algoritma GBFS menggunakan nilai $f(n)$ yaitu nilai heuristik berupa perbedaan kata pada simpul terhadap kata target dimana semakin banyak perbedaan maka nilai heuristik akan semakin besar. Penggunaan algoritma GBFS tidak menjadi solusi yang optimal atau bahkan menemukan solusi sama sekali. Hal tersebut dikarenakan sifat GBFS yang memilih rute berdasarkan nilai heuristik dari semua anak simpul. Walaupun sebuah simpul adalah optimum lokal karena memiliki nilai heuristik terbaik, tidak menjamin bahwa

solusi yang dibuat adalah solusi optimum global. Tidak hanya itu GBFS juga dapat memiliki masalah seperti terjebak pada local minima, dimana dia memilih optimum lokal namun optimum lokal dari simpul menunjuk pada simpul yang sudah dikunjungi sebelumnya.

3.3 Algoritma A*

3.3.1 Implementasi

Dalam penggunaan algoritma A* untuk mencari rute terbaik dari suatu kata asal menuju kata target, penulis menggunakan nilai $f(n)$ yang merupakan penjumlahan 2 komponen yaitu nilai $g(n)$ dan nilai $h(n)$. Nilai $h(n)$ adalah nilai heuristik yang semakin banyak perbedaan karakter antar kata, maka semakin besar nilai heuristiknya. Nilai $g(n)$ adalah banyak langkah yang diperlukan dari simpul awal menuju simpul tersebut. Berikut adalah langkah-langkah algoritma A* yang diimplementasikan.

1. Sebelumnya program sudah membuat graf yang terbuat dari simpul berupa kata-kata yang memiliki panjang kata sesuai dengan panjang kata masukkan (kata asal dan target), dan sisi berupa hubungan dua kata yang memiliki perbedaan salah satu karakternya.
2. Program membuat priority queue yang diurutkan berdasarkan nilai $f(n)$ suatu simpul yang merupakan hasil penjumlahan nilai $g(n)$ dan nilai $h(n)$.
3. Program membuat map gScore yang menyimpan nilai $g(n)$ suatu simpul.
4. Program kemudian memasukan simpul akar yang dibuat dari kata asal dan nilai $f(n)$ yang dimiliki..
5. Selama priority queue tidak kosong, simpul dengan nilai $f(n)$ paling rendah akan diambil dari priority queue.
6. Simpul tersebut dibandingkan dengan kata yang dicari atau kata target. Jika sudah sesuai dengan target maka program akan berhenti dan mengembalikan hasil pencarian.
7. Jika belum, program kemudian akan menambahkan nilai $g(n)$ ke dalam map gScore dan menambahkan semua simpul anak ke dalam priority queue dengan nilai $f(n)$ masing-masing simpul.

8. Program akan mengulangi langkah 5 hingga 7 sampai priority queue kosong, atau sebuah solusi ditemukan.

3.3.2 Analisis

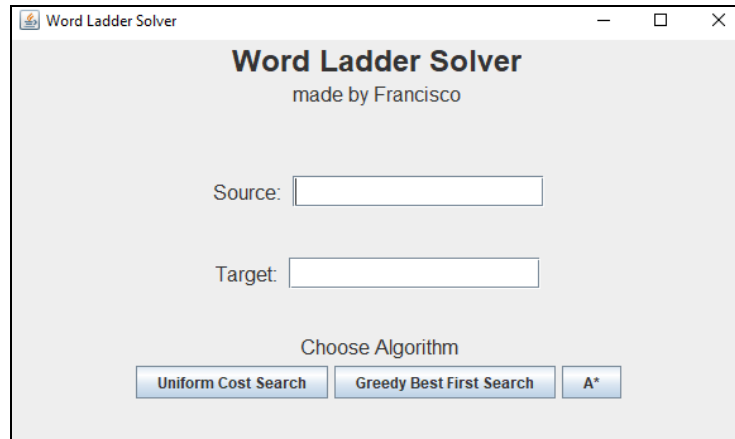
Dalam persoalan Word Ladder, algoritma A* menggunakan nilai $f(n)$ yang merupakan penjumlahan 2 komponen yaitu nilai $g(n)$ dan nilai $h(n)$. Nilai $h(n)$ adalah nilai heuristik yang semakin banyak perbedaan karakter antar kata, maka semakin besar nilai heuristiknya. Nilai $g(n)$ adalah banyak langkah yang diperlukan dari simpul awal menuju simpul tersebut.

Algoritma A* menggunakan fungsi $h(n)$ atau heuristik yang admissible. $h(n)$ adalah jumlah perbedaan karakter antara simpul dengan kata target. Nilai $h(n)$ tidak pernah bernilai lebih dari $h^*(n)$ atau cost asli untuk mencapai goal state. $h(n)$ yang digunakan menggunakan perbandingan antara kata dengan kata tujuan sehingga tidak mungkin nilai $h(n)$ melebihi $h^*(n)$ atau $h(n) \leq h^*(n)$. Sehingga fungsi $h(n)$ yang digunakan selalu mendapatkan rute yang optimal atau algoritma A* hasil implementasi adalah admissible.

Algoritma A* secara teoritis seharusnya lebih efisien dibandingkan dengan UCS. Hal tersebut dikarenakan A* menggunakan nilai $f(n) = g(n) + h(n)$ dan UCS menggunakan cost sebagai penentu dalam pemilihan rute. $g(n)$ dan $f(n)$ pada UCS adalah sama, karena pada UCS $f(n) = g(n)$, dengan $g(n)$ adalah cost untuk mencapai suatu simpul dari simpul akar. Karena $f(n)$ pada A* lebih kompleks dibandingkan dengan UCS, maka kecepatan pencarian target juga semakin cepat. Hal ini dikarenakan nilai $f(n)$ dengan range yang lebih besar akan mempercepat penentuan rute yang optimal sehingga lebih sedikitnya diperlukan runut-balik (backtrack). Sehingga UCS lebih banyak melakukan runut-balik dibandingkan dengan A* sehingga A* lebih efisien dalam pencarian rute.

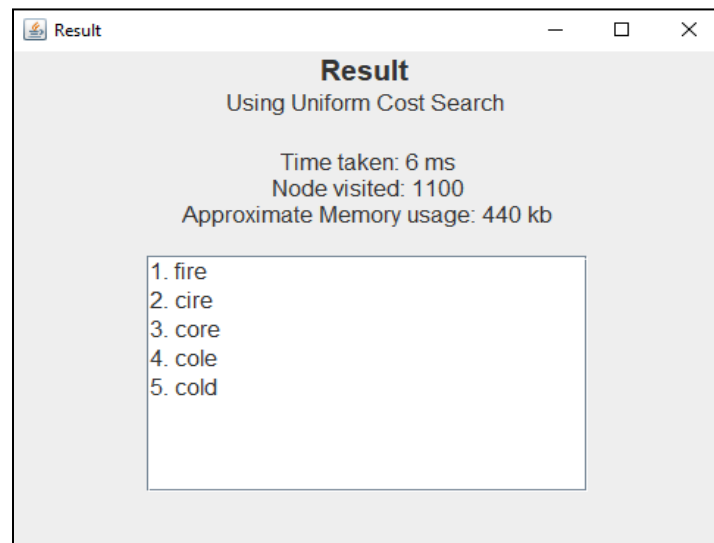
3.4 Graphical User Interface

Program menggunakan library Java Swing untuk menghasilkan sebuah Graphical User Interface yang sederhana. Program dapat menerima masukan berupa kata asal (*Source*) dan kata target (*Target*) yang ingin ditemukan. Berikut adalah gambar dari hasil implementasi GUI yang sederhana.



Gambar 3.4.1 Menu Program

Program dapat melakukan pencarian dengan menekan salah satu dari tiga tombol pada bagian bawah GUI. Tombol “Uniform Cost Search” akan melakukan pencarian rute dengan menggunakan algoritma UCS. Tombol “Greedy Best First Search” akan menggunakan algoritma GBFS, dan tombol “A*” akan menggunakan algoritma A*.



Gambar 3.4.2 Contoh Hasil Pencarian Program

Hasil dari pencarian berupa langkah-langkah dari kata asal menuju kata target atau apakah hasil tidak ditemukan. GUI juga menunjukkan waktu yang dibutuhkan untuk melakukan pencarian, jumlah simpul yang dikunjungi, dan aproksimasi memori yang digunakan.

BAB IV

SOURCE CODE

4.1 Repository Program

Repository program dapat diakses melalui tautan Github berikut:

https://github.com/NoHaitch/Tucil3_13522091

4.2 Class dan Method

4.2.1 Class Pair

Class Pair adalah kelas Generik sebagai struktur data Pair, yaitu pasangan dua nilai.

Tabel 4.2.1.1 Atribut Kelas Pair

Atribut	Keterangan
first	Nilai pertama, Generik F
second	Nilai kedua, Generik S

Tabel 4.2.1.2 Method Kelas Pair

Method	Keterangan
getFirst()	Mengembalikan nilai pertama pada pair
getSecond()	Mengembalikan nilai kedua pada pair

4.2.2 Class Dictionary

Class Dictionary (berbeda dengan `java.util.Dictionary`) merupakan kelas yang merepresentasikan dictionary inggris yang menyimpan kumpulan kata-kata dalam bahasa inggris.

Tabel 4.2.2.1 Atribut Kelas Dictionary

Atribut	Keterangan
words	Hash Set yang menyimpan kumpulan kata dalam bahasa inggris

Tabel 4.2.2.2 Method Kelas Dictionary

Method	Keterangan
getWords	Mengembalikan Hash Set words

loadWords(filename)	Membentuk dictionary dengan membaca file txt yang berisi daftar kata dalam bahasa inggris
isValidWord(word)	Mengecek apakah kata yang diberikan ada pada dictionary
limitWordLength(length)	Melakukan filtrasi sehingga dictionary hanya memiliki kata-kata dengan panjang karakter length.

4.2.3 Class Graph

Class Graph merupakan implementasi dari struktur data Graph yang digunakan sebagai dasar dalam pencarian rute.

Tabel 4.2.3.1 Atribut Kelas Graph

Atribut	Keterangan
adjacencyList	Hashmap dengan key adalah kata, dan value adalah list of kata yang merepresentasikan hubungan dalam graph

Tabel 4.2.3.2 Method Kelas Graph

Method	Keterangan
getAdjacencyList()	Mengembalikan adjacencyList
buildGraph(words)	Membentuk adjacencyList Words adalah hash set yang berisi kata-kata dalam dictionary inggris.

4.2.4 Class AStarSearch

Class AStarSearch digunakan untuk melakukan pencarian rute menggunakan algoritma A*. Di dalam class ini juga dibuat class Node yang menyimpan 3 variabel, word yaitu kata, fscore yaitu nilai f, dan cost yaitu cost untuk mencapai suatu simpul dari simpul akar.

Tabel 4.2.4.1 Method Kelas AStarSearch

Method	Keterangan
heuristic(word, target)	Menghitung nilai heuristik suatu simpul. Semakin banyak perbedaan karakter antara word dan target maka heuristik semakin besar

findShortestPath(graph, source, target)	Melakukan pencarian dengan algoritma A* Mengembalikan Pasangan nilai list of kata dari source menuju target, serta jumlah simpul yang dikunjungi
---	---

4.2.5 Class GreedyBestFirstSearch

Class AStarSearch digunakan untuk melakukan pencarian rute menggunakan algoritma Greedy Best First Search. Di dalam class ini juga dibuat class Node yang menyimpan 2 variabel, word yaitu kata dan heuristic value yaitu nilai heuristik yang dihitung berdasarkan jarak dari simpul dasar..

Tabel 4.2.5.1 Method Kelas GreedyBestFirstSearch

Method	Keterangan
heuristic(word, target)	Menghitung nilai heuristik suatu simpul. Semakin banyak perbedaan karakter antara word dan target maka heuristik semakin besar
findShortestPath(graph, source, target)	Melakukan pencarian dengan algoritma GBFS Mengembalikan Pasangan nilai list of kata dari source menuju target, serta jumlah simpul yang dikunjungi

4.2.6 Class UniformCostSearch

Class AStarSearch digunakan untuk melakukan pencarian rute menggunakan algoritma Uniform Cost Search. Di dalam class ini juga dibuat class Node yang menyimpan 2 variabel, word yaitu kata dan cost.

Tabel 4.2.6.1 Method Kelas UniformCostSearch

Method	Keterangan
findShortestPath(graph, source, target)	Melakukan pencarian dengan algoritma UCS Mengembalikan Pasangan nilai list of kata dari source menuju target, serta jumlah simpul yang dikunjungi

4.3 Source Code Program

4.3.1 Main.java

```
import java.util.List;
import javax.swing.*;
import java.awt.*;

import dictionary.*;
import graph.*;
import search.*;
import pair.*;

public class Main {
    private static String dictionaryPath = "../dictionary/dictionary.txt";
    private static Dictionary fullDictionary = new Dictionary(dictionaryPath);
    private static int FRAME_HEIGHT_MENU = 360;
    private static int FRAME_WIDTH_MENU = 600;
    private static int FRAME_HEIGHT_RESULT = 400;
    private static int FRAME_WIDTH_RESULT = 500;

    public static void main(String[] args) {
        // Test.testGBFS();
        // Test.testUCS();
        // Test.testAstar();

        System.out.println("===== Program Started
        =====");

        if (fullDictionary.getWords().isEmpty()) {
            System.err.println("Dictionary not found");
            return;
        }

        // Create and show the Swing UI
        SwingUtilities.invokeLater(() -> createAndShowGUI());

    }

    /**
     * Perform the Search
     */
    private static void performSearch(String algorithm, String source, String target)
    {
        // Check if source or target are empty
        if (source.trim().isEmpty() || target.trim().isEmpty()) {
            JOptionPane.showMessageDialog(null, "Source and Target words cannot be
            empty.", "Error",
            JOptionPane.ERROR_MESSAGE);
            return;
        }
    }
}
```

```

        // Check if source or target contain spaces
        if (source.contains(" ") || target.contains(" ")) {
            JOptionPane.showMessageDialog(null, "Word can not contain space (\\" \")",
"Error",
                JOptionPane.ERROR_MESSAGE);
            return;
        }

        source = source.toLowerCase();
        target = target.toLowerCase();

        // Check if source and target are valid
        if (source.length() != target.length()) {
            JOptionPane.showMessageDialog(null, "Both Words must be the same
length.", "Error",
                JOptionPane.ERROR_MESSAGE);
            return;
        }

        if (!fullDictionary.isValidWord(source)) {
            JOptionPane.showMessageDialog(null, source + " is not in the
dictionary.", "Error",
                JOptionPane.ERROR_MESSAGE);
            return;
        }

        if (!fullDictionary.isValidWord(target)) {
            JOptionPane.showMessageDialog(null, target + " is not in the
dictionary.", "Error",
                JOptionPane.ERROR_MESSAGE);
            return;
        }

        if(source.equals(target)){
            JOptionPane.showMessageDialog(null, "Word must be different.", "Error",
                JOptionPane.ERROR_MESSAGE);
            return;
        }

        // Create a local dictionary
        Dictionary dictionary = new Dictionary(fullDictionary);

        // Only get words with source length
        dictionary.limitWordLength(source.length());

        // Create a graph from the dictionary
        Graph graph = new Graph(dictionary);

        long startTime, endTime;
        long memoryBefore, memoryAfter;
        Pair<List<String>, Integer> pairOutput;

```

```

        if (algorithm.equals("UCS")) {
            // Uniform Cost Search
            Runtime runtime = Runtime.getRuntime();
            memoryBefore = runtime.freeMemory();
            startTime = System.currentTimeMillis();

            pairOutput = UniformCostSearch.findShortestPath(graph, source, target);

            endTime = System.currentTimeMillis();
            memoryAfter = runtime.freeMemory();

        } else if (algorithm.equals("GBFS")) {
            // Greedy Best First Search
            Runtime runtime = Runtime.getRuntime();
            memoryBefore = runtime.freeMemory();
            startTime = System.currentTimeMillis();

            pairOutput = GreedyBestFirstSearch.findShortestPath(graph, source,
target);

            endTime = System.currentTimeMillis();
            memoryAfter = runtime.freeMemory();

        } else { // AS
            // A* Search
            Runtime runtime = Runtime.getRuntime();
            memoryBefore = runtime.freeMemory();
            startTime = System.currentTimeMillis();

            pairOutput = AStarSearch.findShortestPath(graph, source, target);

            endTime = System.currentTimeMillis();
            memoryAfter = runtime.freeMemory();

        }

        List<String> result = pairOutput.getFirst();
        int nodeVisited = pairOutput.getSecond();

        // Calculate time taken
        long timeTaken = endTime - startTime;
        long memoryUsed = memoryBefore - memoryAfter;
        if(result.size() != 0){
            displayResult(result, timeTaken, nodeVisited, memoryUsed, algorithm);
        } else{
            displayResult(timeTaken, nodeVisited, memoryUsed, algorithm);
        }
    }

    /**
     * Create Main Menu Window

```

```

*/
private static void createAndShowGUI() {
    // Create window
    JFrame frame = new JFrame("Word Ladder Solver");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(FRAME_WIDTH_MENU, FRAME_HEIGHT_MENU); // Set frame size

    // Center the frame on the screen
    Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
    int x = (screenSize.width - frame.getWidth()) / 2;
    int y = (screenSize.height - frame.getHeight()) / 2;
    frame.setLocation(x, y);

    // Create a panel
    JPanel panel = new JPanel();
    panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));

    // Create header panel
    JPanel headerPanel = new JPanel();
    headerPanel.setLayout(new BoxLayout(headerPanel, BoxLayout.Y_AXIS));

    // Create a title label
    JLabel titleLabel = new JLabel("Word Ladder Solver");
    Font titleFont = new Font("FigTree", Font.BOLD, 24);
    titleLabel.setFont(titleFont);
    titleLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
    headerPanel.add(titleLabel);

    // Create a subtitle label
    JLabel subtitleLabel = new JLabel("made by Francisco");
    Font subtitleFont = new Font("FigTree", Font.PLAIN, 16);
    subtitleLabel.setFont(subtitleFont);
    subtitleLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
    headerPanel.add(subtitleLabel);

    // Add some vertical space
    headerPanel.add(Box.createVerticalStrut(50));

    // Add header panel to the main panel
    panel.add(headerPanel);

    // Create a content panel
    JPanel contentPanel = new JPanel();
    contentPanel.setLayout(new BoxLayout(contentPanel, BoxLayout.Y_AXIS)); // Set
    BoxLayout with Y_AXIS alignment

    // Create a panel for source content
    JPanel sourcePanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
    JLabel sourceLabel = new JLabel("Source: ");
    sourceLabel.setFont(subtitleFont);
    JTextField sourceTextField = new JTextField(15);
    sourceTextField.setFont(subtitleFont);

```

```

sourcePanel.add(sourceLabel);
sourcePanel.add(sourceTextField);

// Create a panel for target content
JPanel targetPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
JLabel targetLabel = new JLabel("Target: ");
targetLabel.setFont(subtitleFont);
JTextField targetTextField = new JTextField(15);
targetTextField.setFont(subtitleFont);
targetPanel.add(targetLabel);
targetPanel.add(targetTextField);

// Add both text box to the main panel
contentPanel.add(sourcePanel);
contentPanel.add(targetPanel);
panel.add(contentPanel);

// Create a panel for Choosing an algorithm
JPanel algorithmPanel = new JPanel();
algorithmPanel.setLayout(new BoxLayout(algorithmPanel, BoxLayout.Y_AXIS));

JLabel algorithmLabel = new JLabel("Choose Algorithm");
Font algorithmFont = new Font("FigTree", Font.PLAIN, 16);
algorithmLabel.setFont(algorithmFont);
algorithmLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
algorithmPanel.add(algorithmLabel);

// Create a panel for buttons
JPanel buttonPanel = new JPanel();

// UCS Button
JButton buttonUCS = new JButton("Uniform Cost Search");
buttonUCS.setAlignmentX(Component.CENTER_ALIGNMENT);
buttonPanel.add(buttonUCS);

// Greedy Best First Search Button
JButton buttonGBFS = new JButton("Greedy Best First Search");
buttonGBFS.setAlignmentX(Component.CENTER_ALIGNMENT);
buttonPanel.add(buttonGBFS);

// A*
JButton buttonAS = new JButton("A*");
buttonAS.setAlignmentX(Component.CENTER_ALIGNMENT);
buttonPanel.add(buttonAS);

// Add action listeners to the buttons
buttonUCS.addActionListener(e -> performSearch("UCS",
sourceTextField.getText(), targetTextField.getText()));
buttonGBFS.addActionListener(e -> performSearch("GBFS",
sourceTextField.getText(), targetTextField.getText()));
buttonAS.addActionListener(e -> performSearch("AS",
sourceTextField.getText(), targetTextField.getText()));

```



```

        // Add algorithm to main panel
        algorithmPanel.add(buttonPanel);
        panel.add(algorithmPanel);

        // Add panel to the frame
        frame.add(panel);

        // Display the window
        frame.setVisible(true);
    }

    /**
     * Display Result Window
     */
    private static void displayResult(List<String>result, long timeTaken, int
nodeVisited, long memoryUsed, String algorithm) {
        // Create and configure the result window
        JFrame resultFrame = new JFrame("Result");
        resultFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        resultFrame.setSize(FRAME_WIDTH_RESULT, FRAME_HEIGHT_RESULT);

        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
        int x = (screenSize.width - resultFrame.getWidth()) / 2;
        int y = (screenSize.height - resultFrame.getHeight()) / 2;
        resultFrame.setLocation(x, y);

        // Create a panel
        JPanel panel = new JPanel();
        panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));

        // Create header panel
        JPanel headerPanel = new JPanel();
        headerPanel.setLayout(new BoxLayout(headerPanel, BoxLayout.Y_AXIS));

        // Create a title Label
        JLabel titleLabel = new JLabel("Result");
        Font titleFont = new Font("FigTree", Font.BOLD, 20);
        titleLabel.setFont(titleFont);
        titleLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
        headerPanel.add(titleLabel);

        // Create a subtitle Label
        JLabel subtitleLabel;
        if (algorithm.equals("UCS")) {
            subtitleLabel = new JLabel("Using Uniform Cost Search");
        } else if (algorithm.equals("GBFS")) {
            subtitleLabel = new JLabel("Using Greedy Best First Search");
        } else {
            subtitleLabel = new JLabel("Using A* algorithm");
        }
        Font subtitleFont = new Font("FigTree", Font.PLAIN, 16);

```

```

        subtitleLabel.setFont(subtitleLabel.getFont());
        subtitleLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
        headerPanel.add(subtitleLabel);

        // Add some vertical space
        headerPanel.add(Box.createVerticalStrut(24));

        // Add header panel to the main panel
        panel.add(headerPanel);

        // Create a panel for main content
        JPanel contentPanel = new JPanel();
        contentPanel.setLayout(new BoxLayout(contentPanel, BoxLayout.Y_AXIS));

        JLabel timeLabel = new JLabel("Time taken: " + timeTaken + " ms");
        timeLabel.setFont(new Font("Arial", Font.PLAIN, 16));
        timeLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
        contentPanel.add(timeLabel);

        JLabel visitedLabel = new JLabel("Node visited: " + nodeVisited);
        visitedLabel.setFont(new Font("Arial", Font.PLAIN, 16));
        visitedLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
        contentPanel.add(visitedLabel);

        JLabel memoryLabel = new JLabel("Approximate Memory usage: " +
(memoryUsed/1000) + " kb");
        memoryLabel.setFont(new Font("Arial", Font.PLAIN, 16));
        memoryLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
        contentPanel.add(memoryLabel);

        contentPanel.add(Box.createVerticalStrut(15));

        // Create Panel for list or string result
        JPanel resultTextPanel = new JPanel(new BorderLayout());

        // Create a List model to hold the result
        DefaultListModel<String> listModel = new DefaultListModel<>();
        for (int i = 0; i < result.size(); i++) {
            listModel.addElement((i + 1) + ". " + result.get(i));
        }

        // Create JList with the list model
        JList<String> resultList = new JList<>(listModel);
        resultList.setFont(new Font("Arial", Font.PLAIN, 16));

        // Wrap the resultList in a JScrollPane
        JScrollPane scrollPane = new JScrollPane(resultList);
        scrollPane.setPreferredSize(new Dimension(300,
scrollPane.getPreferredSize().height)); // Set preferred width

        // Wrap the scrollPane in another panel with FlowLayout
        JPanel resultListPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));

```

```

        resultListPanel.add(scrollPane); // Add the scroll pane to the panel

        // Add the resultListPanel to the resultTextPanel
        resultTextPanel.add(resultListPanel, BorderLayout.CENTER);

        // Add the resultTextPanel to the contentPanel
        contentPanel.add(resultTextPanel);

        panel.add(contentPanel);

        // Add result panel to the frame and display
        resultFrame.add(panel);
        resultFrame.setVisible(true);
    }

    /**
     * Display Result Window for solution not found
     */
    private static void displayResult(long timeTaken, int nodeVisited, long
memoryUsed, String algorithm) {
        // Create and configure the result window
        JFrame resultFrame = new JFrame("Result");
        resultFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        resultFrame.setSize(FRAME_WIDTH_RESULT, FRAME_HEIGHT_RESULT);

        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
        int x = (screenSize.width - resultFrame.getWidth()) / 2;
        int y = (screenSize.height - resultFrame.getHeight()) / 2;
        resultFrame.setLocation(x, y);

        // Create a panel
        JPanel panel = new JPanel();
        panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));

        // Create header panel
        JPanel headerPanel = new JPanel();
        headerPanel.setLayout(new BoxLayout(headerPanel, BoxLayout.Y_AXIS));

        // Create a title Label
        JLabel titleLabel = new JLabel("Result");
        Font titleFont = new Font("FigTree", Font.BOLD, 20);
        titleLabel.setFont(titleFont);
        titleLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
        headerPanel.add(titleLabel);

        // Create a subtitle Label
        JLabel subtitleLabel;
        if (algorithm.equals("UCS")) {
            subtitleLabel = new JLabel("Using Uniform Cost Search");
        } else if (algorithm.equals("GBFS")) {
            subtitleLabel = new JLabel("Using Greedy Best First Search");
        } else {

```

```

        subtitleLabel = new JLabel("Using A* algorithm");
    }
    Font subtitleFont = new Font("FigTree", Font.PLAIN, 16);
    subtitleLabel.setFont(subtitleFont);
    subtitleLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
    headerPanel.add(subtitleLabel);

    // Add some vertical space
    headerPanel.add(Box.createVerticalStrut(24));

    // Add header panel to the main panel
    panel.add(headerPanel);

    // Create a panel for main content
    JPanel contentPanel = new JPanel();
    contentPanel.setLayout(new BoxLayout(contentPanel, BoxLayout.Y_AXIS));

    JLabel timeLabel = new JLabel("Time taken: " + timeTaken + " ms");
    timeLabel.setFont(new Font("Arial", Font.PLAIN, 16));
    timeLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
    contentPanel.add(timeLabel);

    JLabel visitedLabel = new JLabel("Node visited: " + nodeVisited);
    visitedLabel.setFont(new Font("Arial", Font.PLAIN, 16));
    visitedLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
    contentPanel.add(visitedLabel);

    JLabel memoryLabel = new JLabel("Approximate Memory usage: " +
(memoryUsed/1000) + " kb");
    memoryLabel.setFont(new Font("Arial", Font.PLAIN, 16));
    memoryLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
    contentPanel.add(memoryLabel);

    contentPanel.add(Box.createVerticalStrut(15));

    // Create Panel for list or string result
    JLabel noSolutionLabel = new JLabel("No Solution Found");
    noSolutionLabel.setFont(new Font("Arial", Font.PLAIN, 16));
    noSolutionLabel.setAlignmentX(Component.CENTER_ALIGNMENT);

    // Add the resultTextPanel to the contentPanel
    contentPanel.add(noSolutionLabel);

    panel.add(contentPanel);

    // Add result panel to the frame and display
    resultFrame.add(panel);
    resultFrame.setVisible(true);
}
}

```

4.3.2 Dictionary.java

```
package dictionary;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashSet;
import java.util.Iterator;

/**
 * Dictionary filled with English Words.
 * <p>
 * Word are made only of alphabets without any special char
 */
public class Dictionary {

    /**
     * Hash Set containing all the english words
     */
    private HashSet<String> words;

    /**
     * CTOR
     *
     * @param filename
     */
    public Dictionary(String filename) {
        words = new HashSet<>();
        loadWords(filename);
    }

    /**
     * CCTOR
     *
     * @param other
     */
    public Dictionary(Dictionary other) {
        this.words = new HashSet<>(other.words);
    }

    public HashSet<String> getWords(){
        return words;
    }

    /**
     * Load the english dictionary from text file
     *
     * @param filename name of file containing english dictionary
     */
    private void loadWords(String filename) {
```

```

        try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = br.readLine()) != null) {
                words.add(line.trim().toLowerCase());
            }
        } catch (IOException e) {
            //error
        }
    }

    /**
     * check if the word is in the english dictionary
     *
     * @param word
     * @return true if word is an english word
     */
    public boolean isValidWord(String word) {
        return words.contains(word.toLowerCase());
    }

    /**
     * Limit the word in the dictionary to a certain length
     * <p>
     * All other words is removed
     *
     * @param Length
     */
    public void limitWordLength(int length) {
        Iterator<String> iterator = words.iterator();
        while (iterator.hasNext()) {
            String word = iterator.next();
            if (word.length() != length) {
                iterator.remove();
            }
        }
    }

    /**
     * Debugging
     * <p>
     * Print words in dictionary using foreach until amount is printed
     *
     * @param amount of words
     */
    public void printWord(int amount) {
        int count = 0;
        for (String x : words) {
            System.out.println(x);
            count++;
            if (count > amount) {
                break;
            }
        }
    }

```

```

    }
}
}

```

4.3.3 Graph.java

```

package graph;

import dictionary.*;
import java.util.Map;
import java.util.HashMap;
import java.util.HashSet;
import java.util.ArrayList;

/**
 * Make a Graph
 * <p>
 * Vertices: words
 * <p>
 * Edge: words that has a 1 letter difference
 */
public class Graph {

    private HashMap<String, ArrayList<String>> adjacencyList;

    public Graph(Dictionary dictionary) {
        this.adjacencyList = new HashMap<>();
        buildGraph(dictionary.getWords());
    }

    /**
     * Get AdjacencyList
     */
    public HashMap<String, ArrayList<String>> getAdjacencyList() {
        return adjacencyList;
    }

    /**
     * Create a Graph from a pool of words with the same length
     */
    private void buildGraph(HashSet<String> words) {
        for (String word : words) {
            adjacencyList.put(word, new ArrayList<>());
        }

        for (String word : words) {
            // convert word into array of letters

```

```

        char[] chars = word.toCharArray();
        for (int i = 0; i < word.length(); i++) {
            char originalChar = chars[i];

            // try changing the char into all alphabet
            for (char c = 'a'; c <= 'z'; c++) {
                // if char changed is not the original char
                if (c != originalChar) {
                    chars[i] = c;
                    String newWord = String.valueOf(chars);

                    // if the newly built word is a valid word in the dictionary
                    if (words.contains(newWord)) {
                        adjacencyList.get(word).add(newWord);
                    }
                }
            }
            // Reset the character
            chars[i] = originalChar;
        }
    }
}

/**
 * Debugging
 * <p>
 * Print every node and its connections
 */
public void printGraph() {
    for (Map.Entry<String, ArrayList<String>> entry : adjacencyList.entrySet()) {
        String word = entry.getKey();
        ArrayList<String> neighbors = entry.getValue();
        System.out.print(word + ": ");
        for (String neighbor : neighbors) {
            System.out.print(neighbor + " ");
        }
        System.out.println();
    }
}
}

```

4.3.4 Pair.java

```

package pair;

public class Pair<F, S> {
    private final F first;

```



```

private final S second;

public Pair(F first, S second) {
    this.first = first;
    this.second = second;
}

public F getFirst() {
    return first;
}

public S getSecond() {
    return second;
}
}

```

4.3.5 AStarSearch.java

```

package search;

import java.util.*;

import graph.*;
import pair.*;

public class AStarSearch {
    /**
     * A* Nodes
     * <p>
     * Store word, fscore, and cost
     */
    private static class Node {
        private String word;
        private int fScore;
        private int cost;

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

        public String getWord() {
            return word;
        }

        public int getFScore() {
            return fScore;
        }
    }
}

```

```

    }

    public int getCost() {
        return cost;
    }
}

/**
 * Calculate Heuristic Value
 */
private static int heuristic(String word, String target) {
    int difference = 0;
    for (int i = 0; i < word.length(); i++) {
        if (word.charAt(i) != target.charAt(i)) {
            difference++;
        }
    }
    return difference;
}

/**
 * A* algorithm
 */
public static Pair<List<String>, Integer> findShortestPath(Graph graph, String
source, String target) {
    // Queue of nodes sorted by their fscore
    PriorityQueue<Node> pq = new
PriorityQueue<>(Comparator.comparingInt(Node::getFScore));

    // Map to track visited nodes and their gScores
    Map<String, Integer> gScores = new HashMap<>();

    // Map to keep track of the parent node for each node in the shortest path
    Map<String, String> parent = new HashMap<>();

    // Initialize g-scores for all nodes
    for (String word : graph.getAdjacencyList().keySet()) {
        gScores.put(word, Integer.MAX_VALUE);
    }

    // Add the source node to the priority queue
    pq.offer(new Node(source, heuristic(source, target), 0));
    gScores.put(source, 0);

    // Node visited counter
    int nodeVisited = 0;

    // Perform A* Search
    while (!pq.isEmpty()) {
        // Dequeue the node with the lowest f-score
        Node currentNode = pq.poll();

```

```

        nodeVisited++;
        String currentWord = currentNode.getWord();
        int currentCost = currentNode.getCost();

        // Target found
        if (currentWord.equals(target)) {
            List<String> shortestPath = new ArrayList<>();

            // Reconstruct the result
            String word = target;
            while (word != null) {
                shortestPath.add(0, word);
                word = parent.get(word);
            }
            return new Pair<>(shortestPath, nodeVisited);
        }

        // Add neighboring node
        for (String neighbor : graph.getAdjacencyList().getOrDefault(currentWord,
            new ArrayList<>())) {
            // Calculate the tentative g-score for the neighbor
            int tentativeGScore = currentCost + 1;

            // If the tentative g-score is lower than the current g-score
            if (tentativeGScore < gScores.get(neighbor)) {
                // Update the parent node for the neighbor
                parent.put(neighbor, currentWord);

                // Update the g-score for the neighbor
                gScores.put(neighbor, tentativeGScore);

                // Calculate the f-score for the neighbor
                int fScore = tentativeGScore + heuristic(neighbor, target);

                // Add the neighbor to the priority queue
                pq.offer(new Node(neighbor, fScore, tentativeGScore));
            }
        }
    }

    // If the target not found
    return new Pair<>(Collections.emptyList(), nodeVisited);
}

```

4.3.6 GreedyBestFirstSearch.java

```
package search;
```

```

import java.util.*;

import graph.*;
import pair.*;

public class GreedyBestFirstSearch {

    /**
     * Greedy Best First Search Nodes
     * <p>
     * Store word and heuristic value
     */
    private static class Node {
        private String word;
        private int heuristicValue;

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

        public String getWord() {
            return word;
        }

        public int getHeuristicValue() {
            return heuristicValue;
        }
    }

    /**
     * Calculate Heuristic Value
     */
    private static int heuristic(String word, String target) {
        int difference = 0;
        for (int i = 0; i < word.length(); i++) {
            if (word.charAt(i) != target.charAt(i)) {
                difference++;
            }
        }
        return difference;
    }

    /**
     * Greedy Best First Search
     */
    public static Pair<List<String>, Integer> findShortestPath(Graph graph, String
source, String target) {
        // Queue of nodes sorted by heuristic
        PriorityQueue<Node> pq = new
PriorityQueue<>(Comparator.comparingInt(Node::getHeuristicValue));

```

```

// Map to track visited nodes
Set<String> visited = new HashSet<>();

// Map to keep track of the parent node for each node in the shortest path
List<String> result = new ArrayList<>();

// Add the source node to the priority queue
pq.offer(new Node(source, heuristic(source, target)));

// Node visited counter
int nodeVisited = 0;

// Greedy Best First Search
while (!pq.isEmpty()) {
    // Get the node with the smallest heuristic
    Node currentNode = pq.poll();
    pq.clear();

    nodeVisited++;
    String currentWord = currentNode.getWord();
    visited.add(currentWord);
    result.add(currentWord);

    // Target found
    if (currentWord.equals(target)) {
        return new Pair<>(result, nodeVisited);
    }

    // add neighboring node
    for (String neighbor :
graph.getAdjacencyList().getOrDefault(currentWord, new ArrayList<>())) {
        // If neighbor have not been visited
        if(!visited.contains(neighbor)){

            // Add the neighbor to the priority queue
            pq.offer(new Node(neighbor, heuristic(neighbor, target)));
        }
    }
}

// If the target not found
return new Pair<>(Collections.emptyList(), nodeVisited);
}
}

```

4.3.7 UniformCostSearch.java

```
package search;
```

```

import java.util.*;

import graph.*;
import pair.*;

public class UniformCostSearch {
    /**
     * Uniform Cost Search Nodes
     * <p>
     * Store word and cost
     */
    private static 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;
        }
    }

    /**
     * Uniform Cost Search
     */
    public static Pair<List<String>, Integer> findShortestPath(Graph graph, String
source, String target) {
        // Queue of nodes sorted by their cost
        PriorityQueue<Node> pq = new
PriorityQueue<>(Comparator.comparingInt(Node::getCost));

        // Map to track visited nodes and the smallest cost
        Map<String, Integer> visited = new HashMap<>();

        // Map to keep track of the parent node for each node in the shortest path
        Map<String, String> parent = new HashMap<>();

        // Add the source node to the priority queue with cost 0
        pq.offer(new Node(source, 0));

        // Node visited counter
        int nodeVisited = 0;

        // Uniform Cost Search

```

```

while (!pq.isEmpty()) {
    // Get the node with the lowest cost
    Node currentNode = pq.poll();

    nodeVisited++;
    String currentWord = currentNode.getWord();
    int currentCost = currentNode.getCost();

    // Target found
    if (currentWord.equals(target)) {
        List<String> shortestPath = new ArrayList<>();

        // Reconstruct the result
        String word = target;
        while (word != null) {
            shortestPath.add(0, word);
            word = parent.get(word);
        }
        return new Pair<>(shortestPath, nodeVisited);
    }

    // Add node to visited
    visited.put(currentWord, currentCost);

    // Add neighboring node
    for (String neighbor :
graph.getAdjacencyList().getOrDefault(currentWord, new ArrayList<>())) {
        int neighborCost = currentCost + 1;

        if (!visited.containsKey(neighbor) || neighborCost <
visited.get(neighbor)) {
            // Update the cost of the neighbor
            visited.put(neighbor, neighborCost);

            // Update the parent node for the neighbor
            parent.put(neighbor, currentWord);

            // Add the neighbor to the priority queue
            pq.offer(new Node(neighbor, neighborCost));
        }
    }
}

// If target is not found
return new Pair<>(Collections.emptyList(), nodeVisited);
}
}

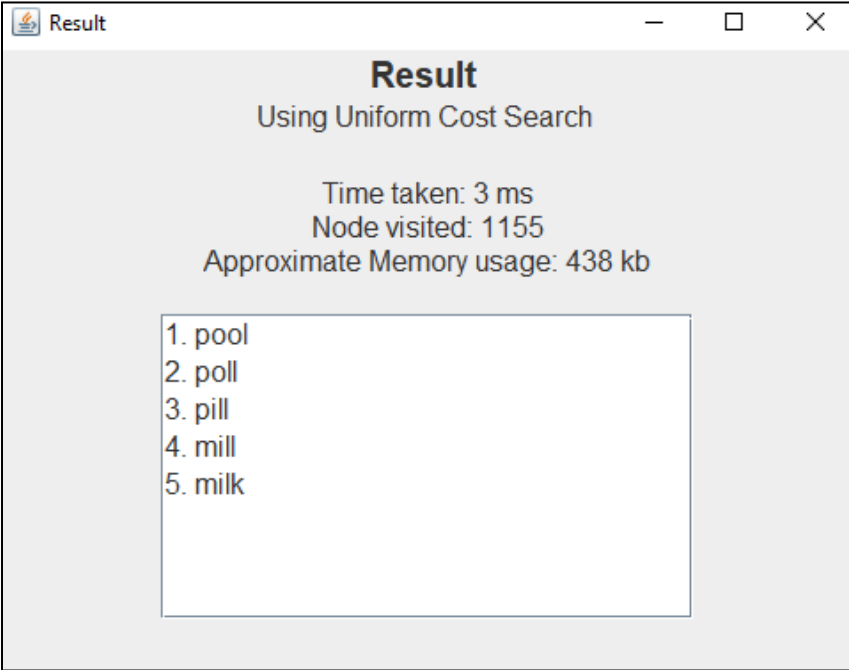
```

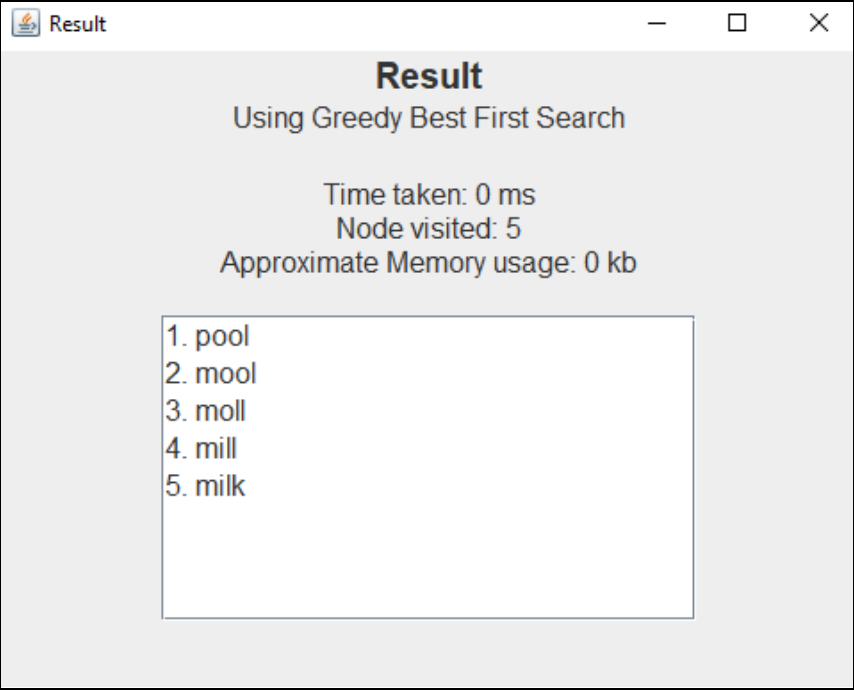
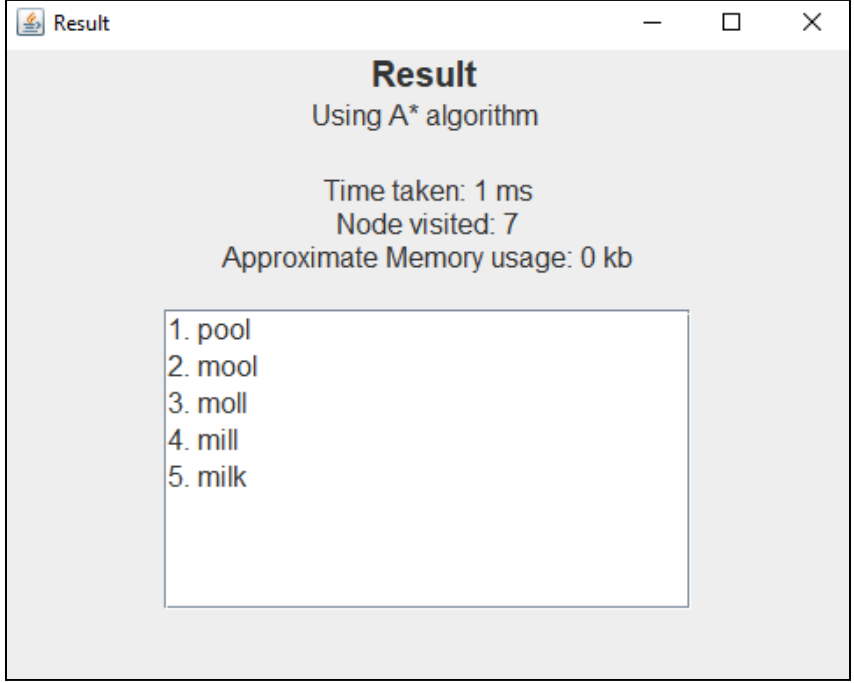
BAB V

ANALISIS DAN PENGUJIAN

5.1 Pengujian

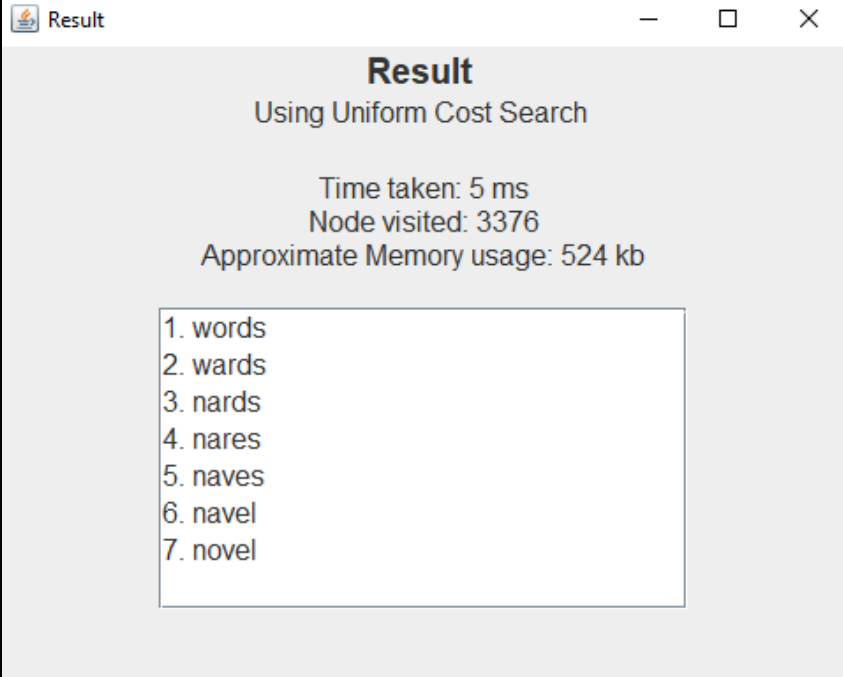
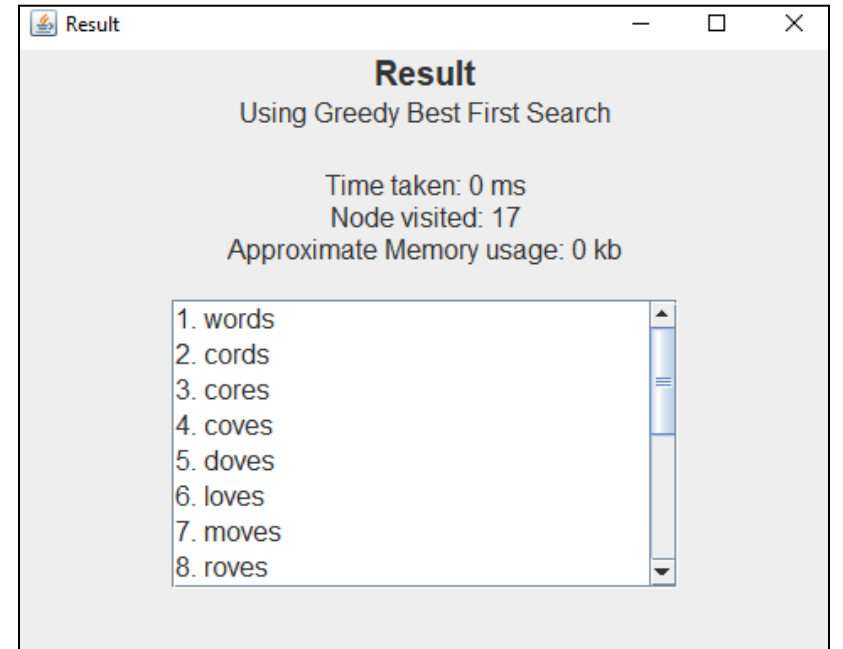
5.1.1 Tests Case 1

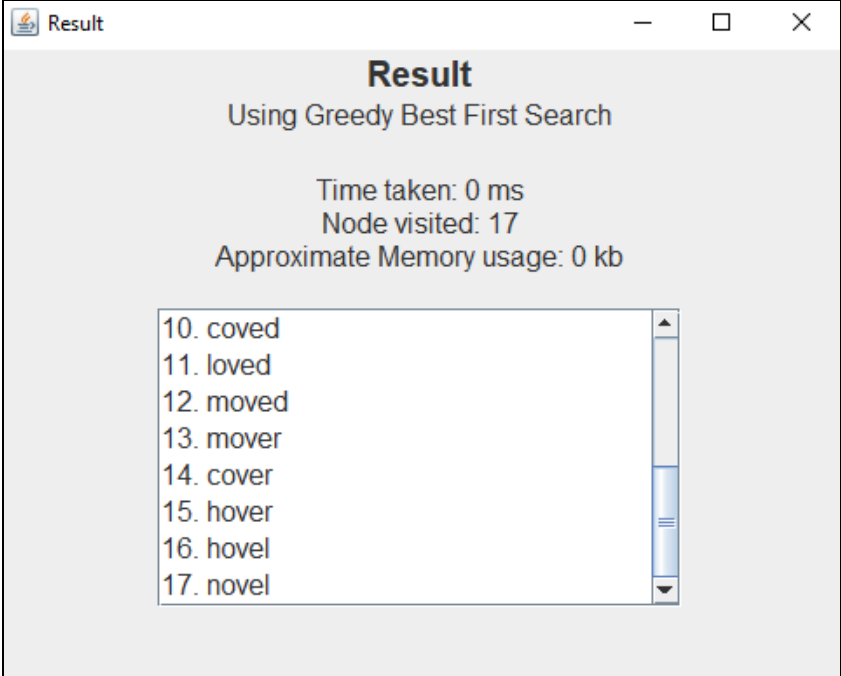
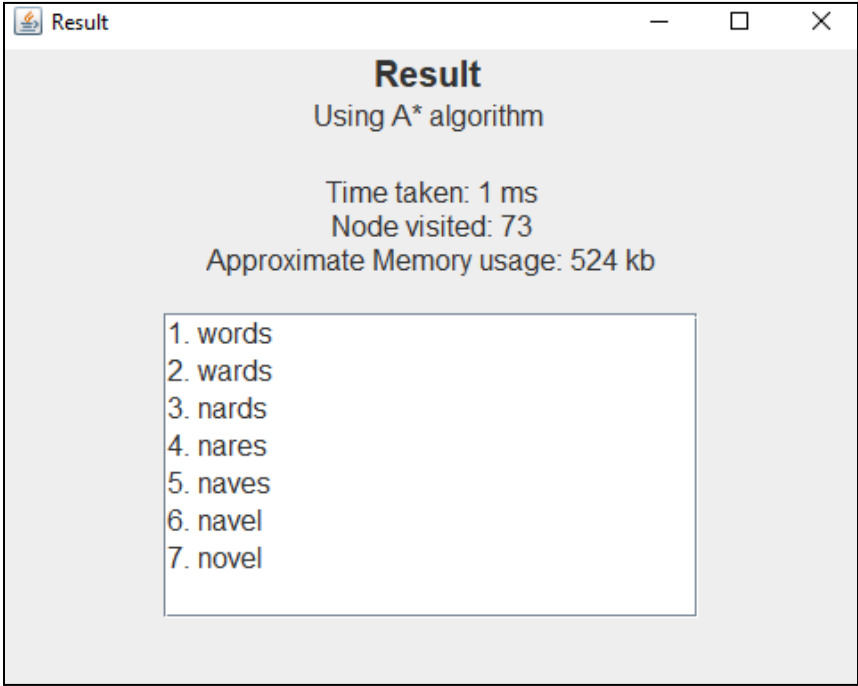
Input: <ul style="list-style-type: none">- Source: pool- Target: milk	Hasil Pengujian
Uniform Cost Search	

Greedy Best First Search	
A*	

5.1.2 Test Case 2

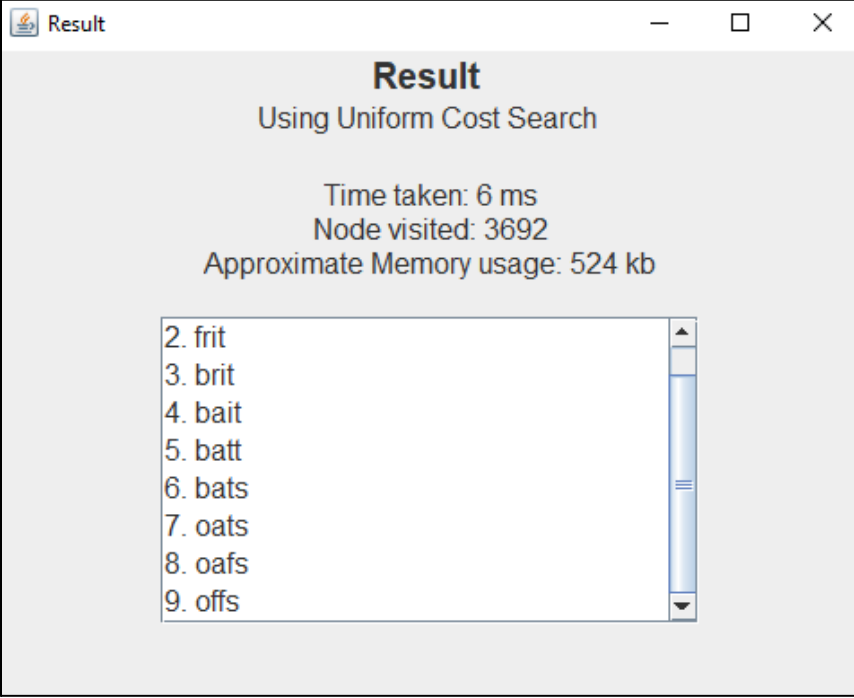
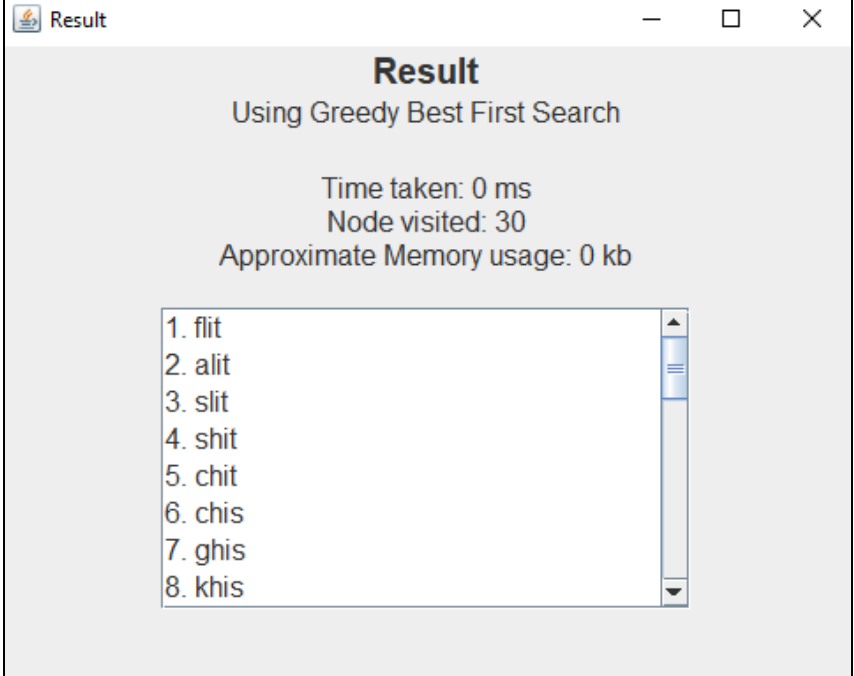
Input: - Source: words	Hasil Pengujian
----------------------------------	------------------------

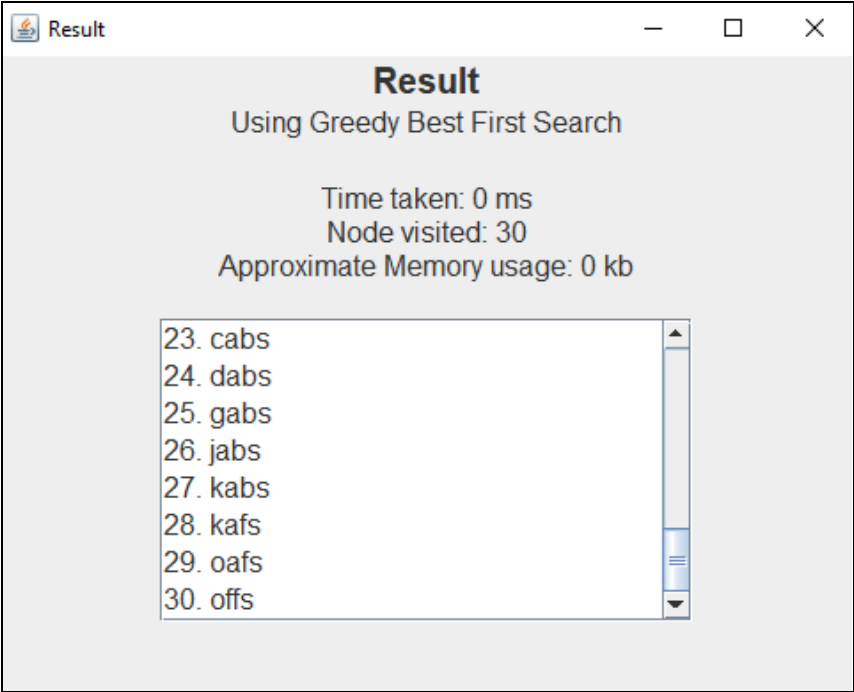
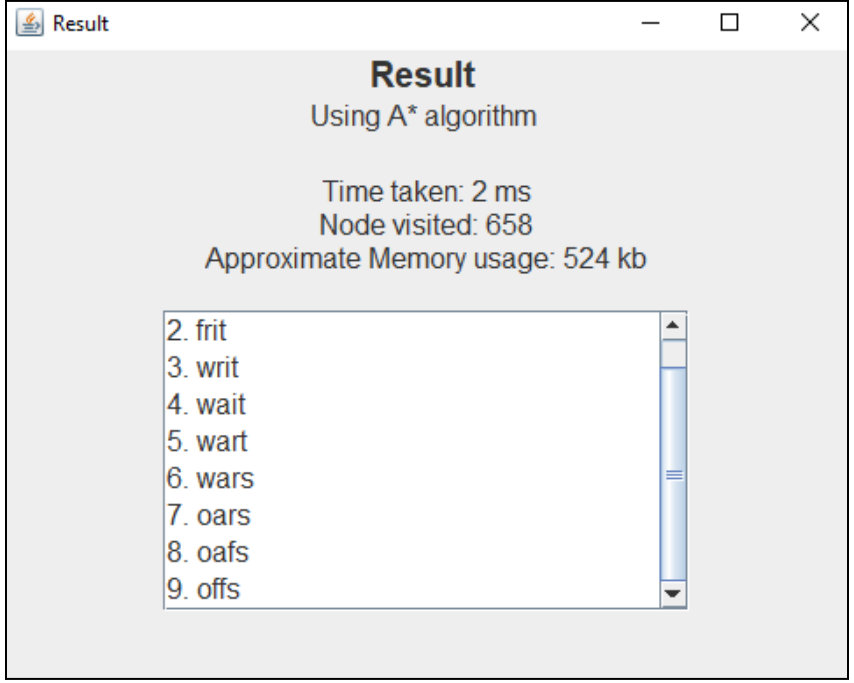
<p>- Target: novel</p>	
<p>Uniform Cost Search</p>	 <p>Result Using Uniform Cost Search</p> <p>Time taken: 5 ms Node visited: 3376 Approximate Memory usage: 524 kb</p> <ol style="list-style-type: none"> 1. words 2. wards 3. nards 4. nares 5. naves 6. navel 7. novel
<p>Greedy Best First Search</p>	 <p>Result Using Greedy Best First Search</p> <p>Time taken: 0 ms Node visited: 17 Approximate Memory usage: 0 kb</p> <ol style="list-style-type: none"> 1. words 2. cords 3. cores 4. coves 5. doves 6. loves 7. moves 8. roves

	
A*	

5.1.3 Test Case 3

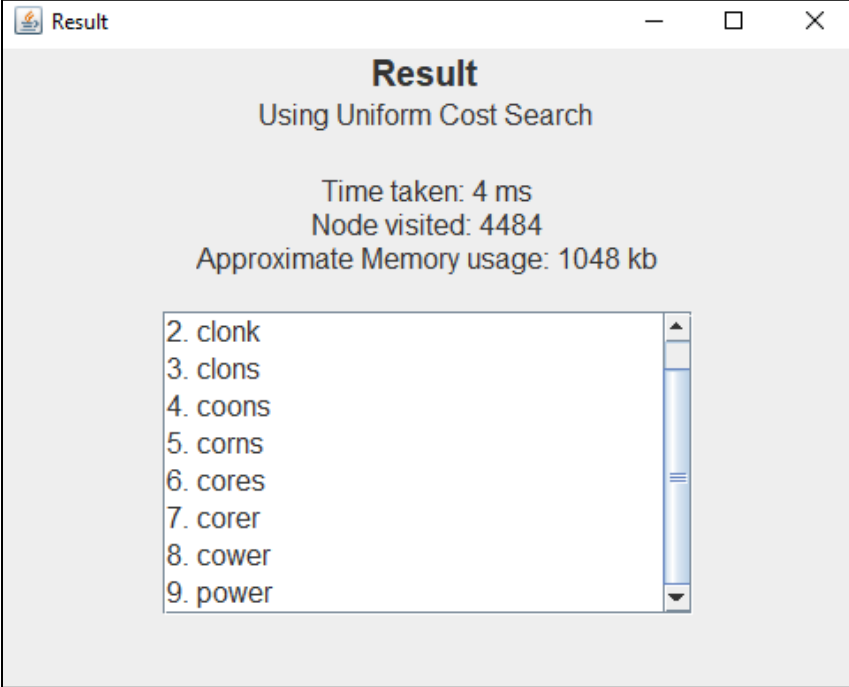
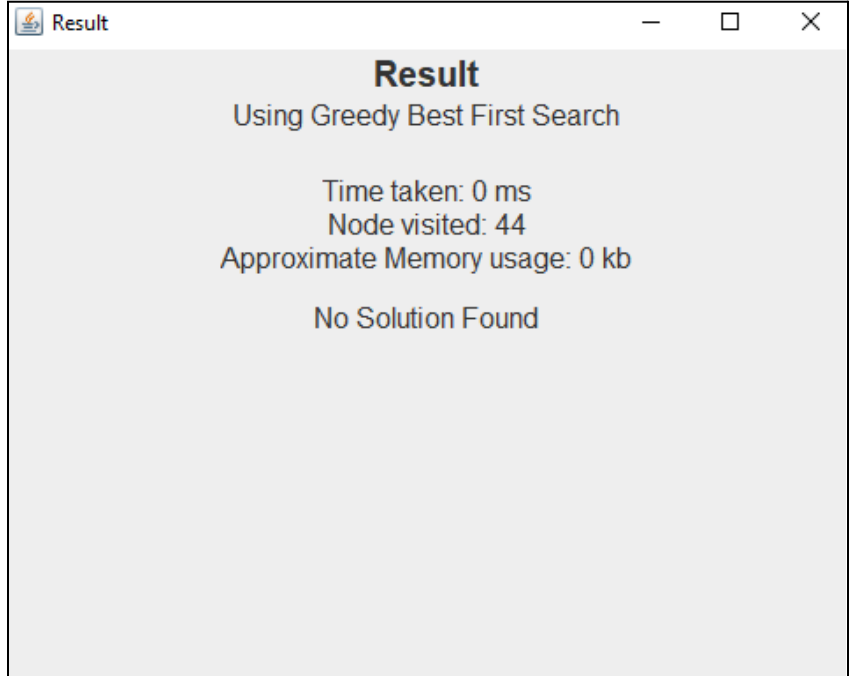
Input: - Source: flit	Hasil Pengujian
---------------------------------	------------------------

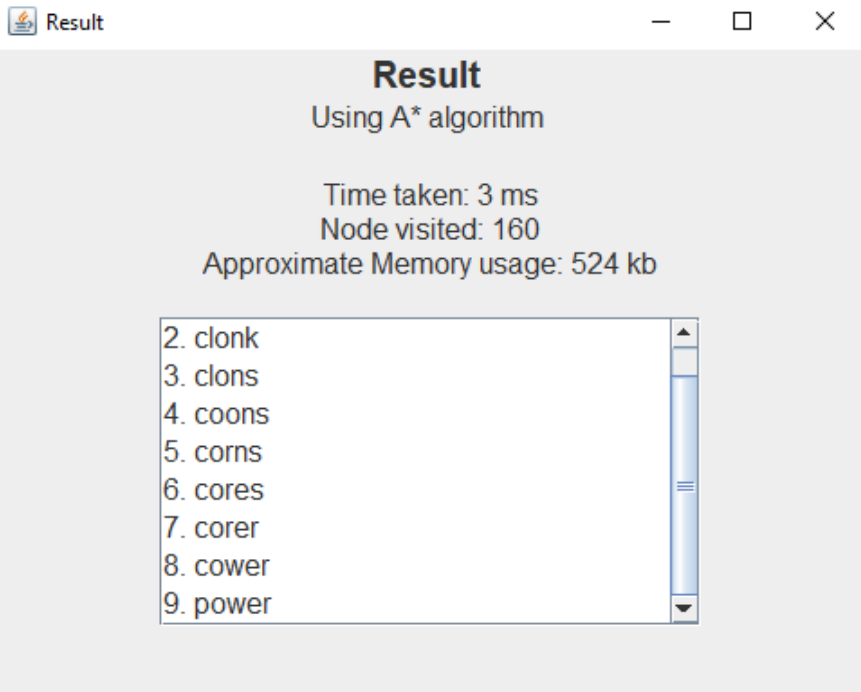
<p>- Target: offs</p>	
<p>Uniform Cost Search</p>	 <p>Result</p> <p>Using Uniform Cost Search</p> <p>Time taken: 6 ms Node visited: 3692 Approximate Memory usage: 524 kb</p> <ul style="list-style-type: none"> 2. frit 3. brit 4. bait 5. batt 6. bats 7. oats 8. oafs 9. offs
<p>Greedy Best First Search</p>	 <p>Result</p> <p>Using Greedy Best First Search</p> <p>Time taken: 0 ms Node visited: 30 Approximate Memory usage: 0 kb</p> <ul style="list-style-type: none"> 1. flit 2. alit 3. slit 4. shit 5. chit 6. chis 7. ghis 8. khis

	
A*	

5.1.4 Test Case 4

Input: - Source: clock	Hasil Pengujian
----------------------------------	------------------------

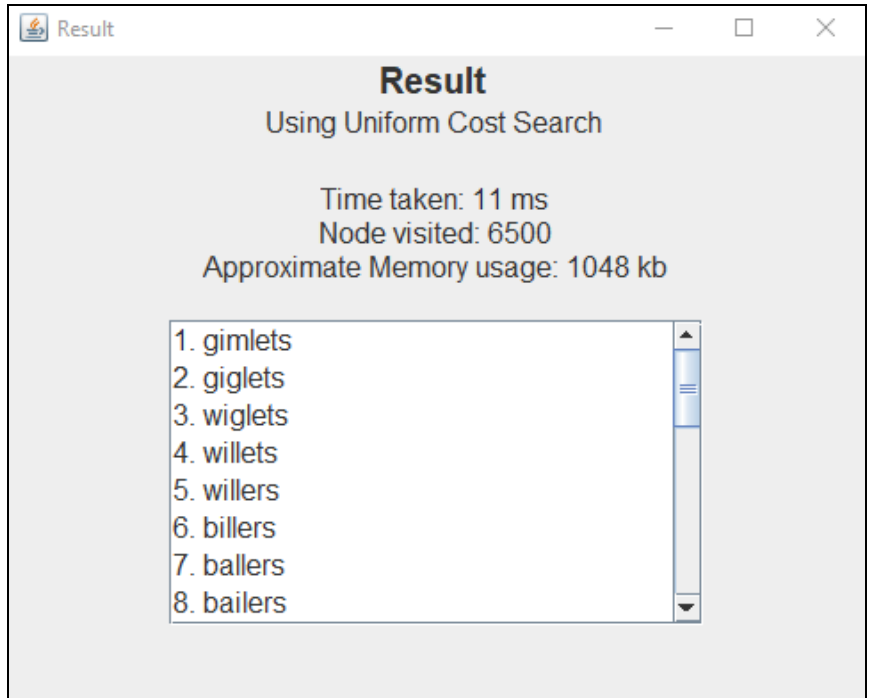
<p>- Target: power</p>	
<p>Uniform Cost Search</p>	 <p>Result Using Uniform Cost Search</p> <p>Time taken: 4 ms Node visited: 4484 Approximate Memory usage: 1048 kb</p> <ul style="list-style-type: none"> 2. clonk 3. clons 4. coons 5. corns 6. cores 7. corer 8. cower 9. power
<p>Greedy Best First Search</p>	 <p>Result Using Greedy Best First Search</p> <p>Time taken: 0 ms Node visited: 44 Approximate Memory usage: 0 kb</p> <p>No Solution Found</p>

A*	 <p>Result Using A* algorithm</p> <p>Time taken: 3 ms Node visited: 160 Approximate Memory usage: 524 kb</p> <ul style="list-style-type: none"> 2. clonk 3. clons 4. coons 5. corns 6. cores 7. corer 8. cower 9. power
----	--

5.1.5 Test Case 5

Input: <ul style="list-style-type: none"> - Source: gimlets - Target: treeing 	Hasil Pengujian
--	------------------------

Uniform Cost Search

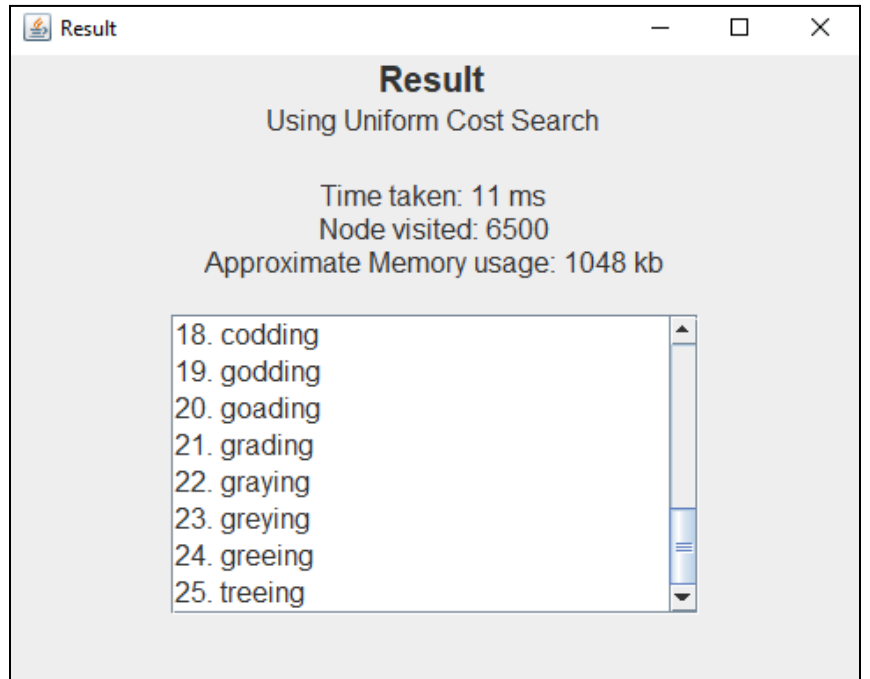


Result

Using Uniform Cost Search

Time taken: 11 ms
Node visited: 6500
Approximate Memory usage: 1048 kb

1. gimlets
2. giglets
3. wiglets
4. willets
5. willers
6. billers
7. ballers
8. bailers



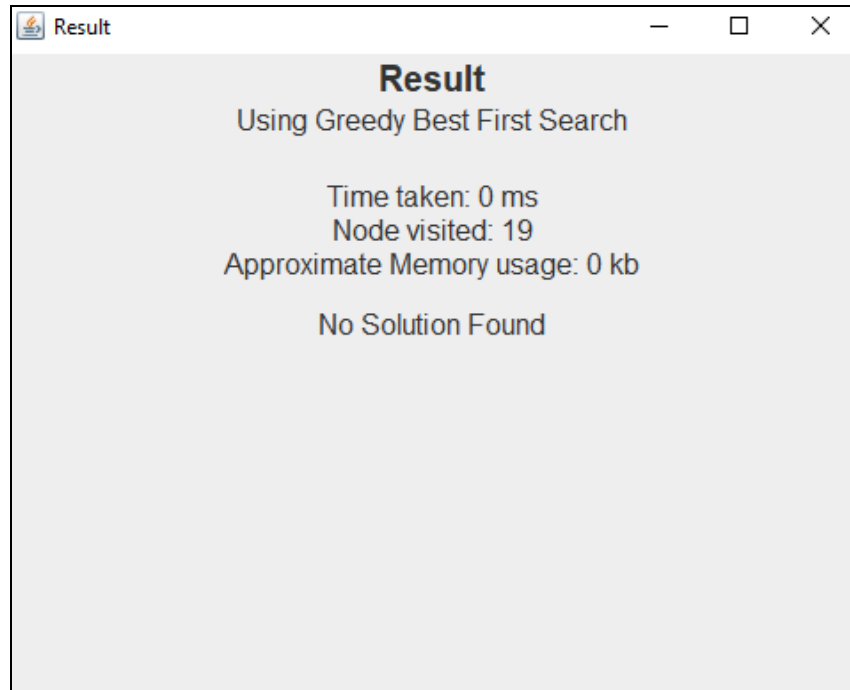
Result

Using Uniform Cost Search

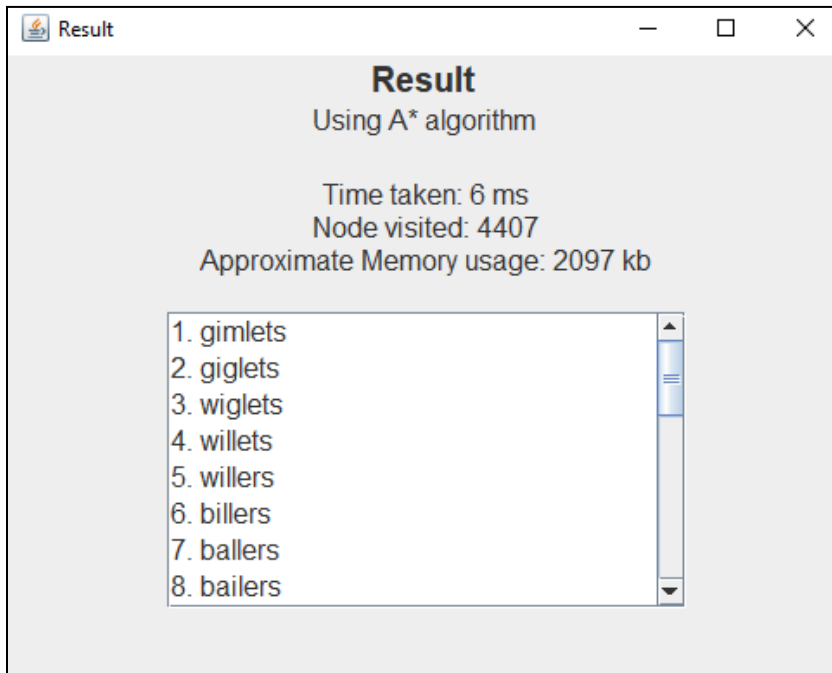
Time taken: 11 ms
Node visited: 6500
Approximate Memory usage: 1048 kb

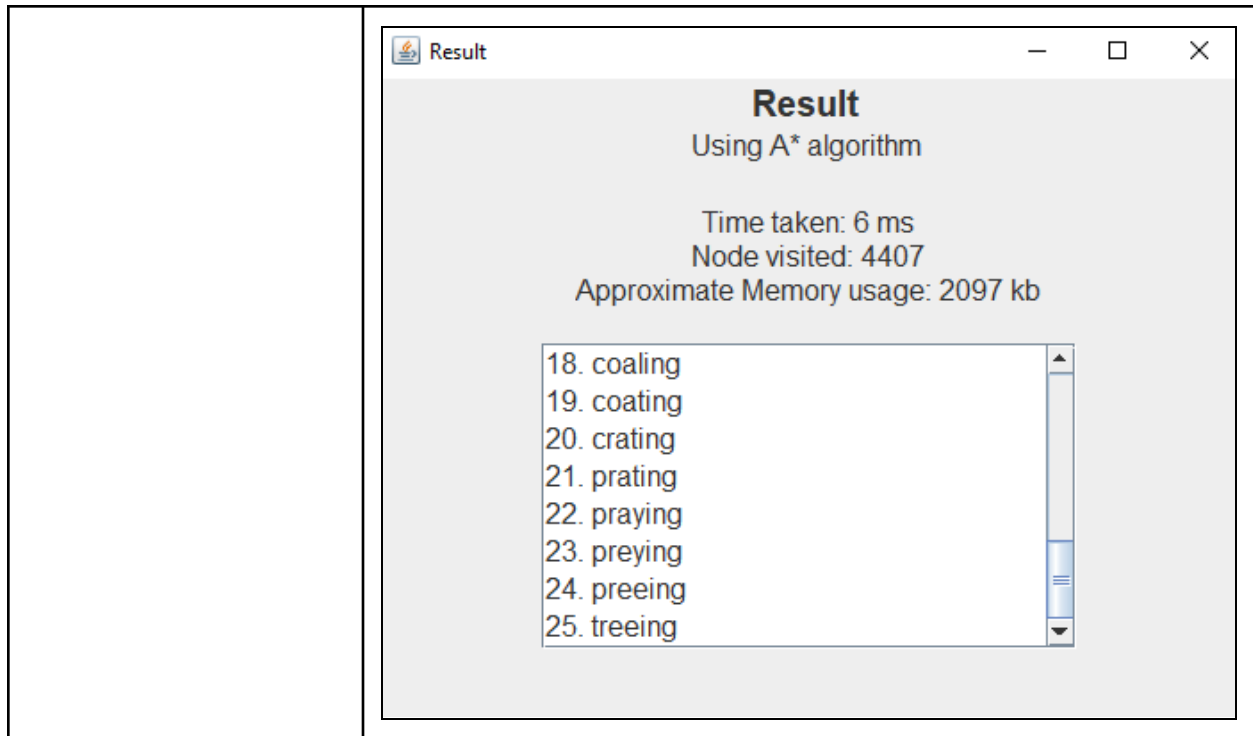
18. coddling
19. godding
20. goading
21. grading
22. graying
23. greying
24. greeing
25. treeing

Greedy Best First Search



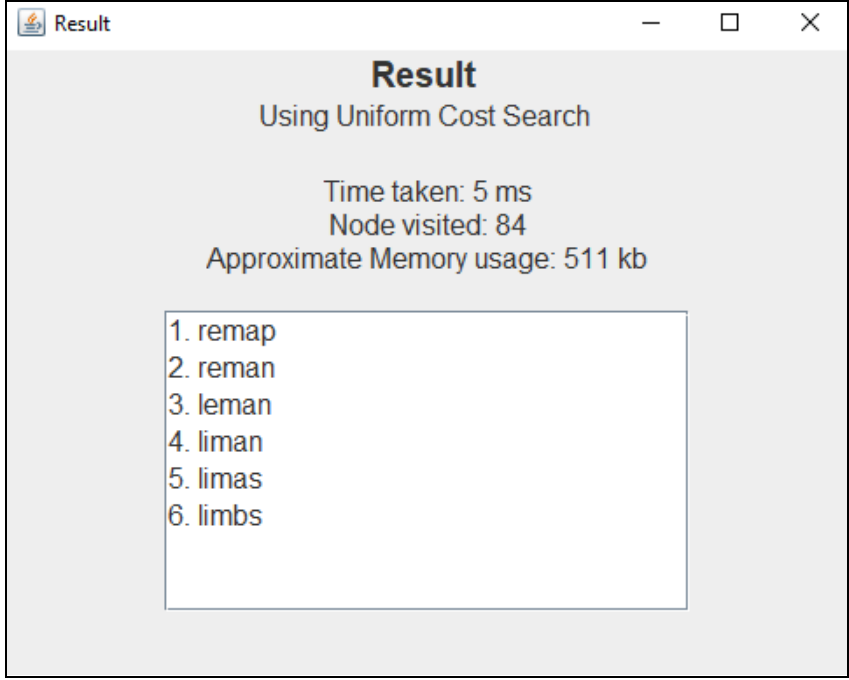
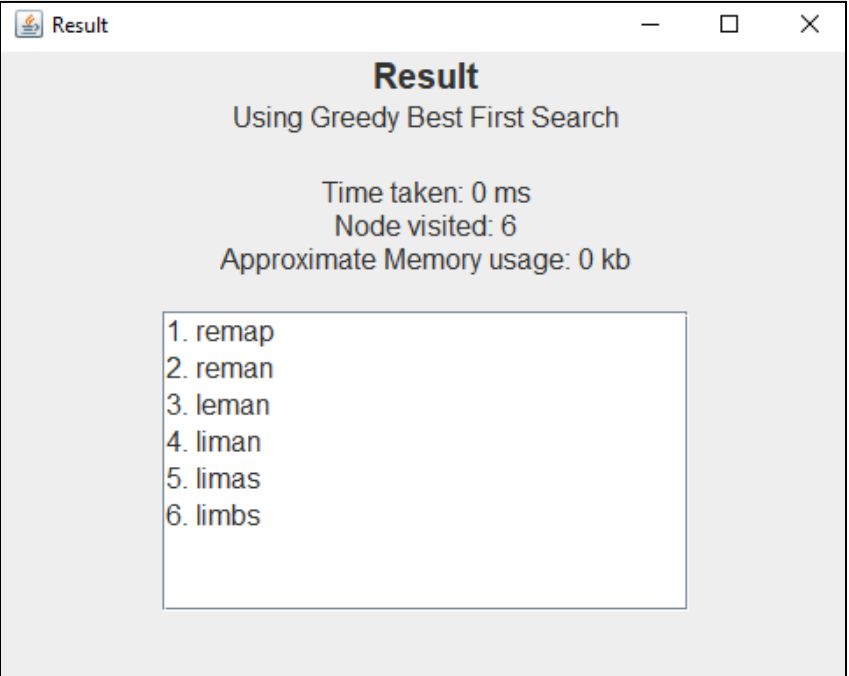
A*

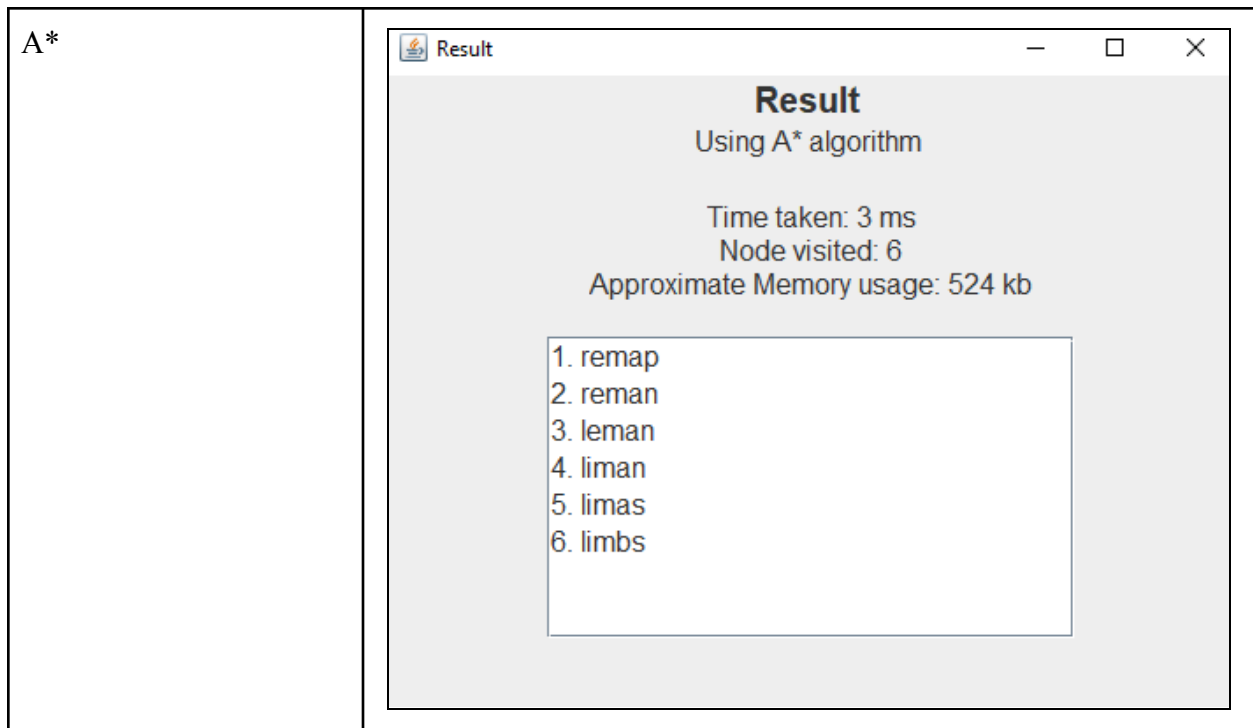




5.1.6 Test Case 6

Input: <ul style="list-style-type: none">- Source: remap- Target: limbs	Hasil Pengujian
---	------------------------

Uniform Cost Search	 <p>Result</p> <p>Using Uniform Cost Search</p> <p>Time taken: 5 ms Node visited: 84 Approximate Memory usage: 511 kb</p> <ol style="list-style-type: none"> 1. remap 2. reman 3. leman 4. liman 5. limas 6. limbs
Greedy Best First Search	 <p>Result</p> <p>Using Greedy Best First Search</p> <p>Time taken: 0 ms Node visited: 6 Approximate Memory usage: 0 kb</p> <ol style="list-style-type: none"> 1. remap 2. reman 3. leman 4. liman 5. limas 6. limbs



5.2 Analisis Pengujian

Hasil rute dari permasalahan Word Ladder dengan tiga algoritma berbeda, yaitu Uniform Cost Search, Greedy Best First Search, dan A* memiliki perbedaan dalam optimalitas, waktu eksekusi dan efektivitas memori.

5.2.1 Optimalitas

Untuk permasalahan optimalitas, dapat melihat data berupa panjang rute yang dihasilkan masing-masing algoritma untuk setiap test case.

Tabel 5.2.1 Data panjang rute hasil pengujian

Nomor Test Case	Panjang Rute		
	UCS	GBFS	A*
1.	5	5	5
2.	7	17	7
3.	9	30	9
4.	9	0	9

5.	25	0	25
6.	6	6	6

Dari data tersebut dapat dilihat bahwa algoritma UCS dan A* sudah optimal, hal tersebut dapat dilihat bahwa hasil dari semua test case memberikan panjang rute yang optimal walaupun rute yang diberikan dapat memiliki urutan yang berbeda. Misal untuk test case 3, rute hasil pencarian menggunakan UCS adalah flit > frit > brit > bait > batt > bats > oats > oafs > offs, sedangkan menggunakan algoritma A* adalah flit > frit > writ > wait > wart > wars > oars > oafs > offs. Kedua hasil rute sama-sama optimal karena memiliki langkah yang paling minimum, urutan atau kata-kata yang digunakan berbeda.

Untuk GBFS, masih belum optimal, karena GBFS panjang rute yang dihasilkan jauh dari hasil algoritma UCS atau A*. Hal tersebut menyebabkan seringnya algoritma GBFS terjebak pada lokal minimum, sehingga dapat tidak menemukan hasil atau hasil yang ditemukan tidak optimal.

5.2.2 Waktu Eksekusi

Untuk permasalahan waktu eksekusi, dapat melihat data berupa waktu eksekusi yang diperlukan untuk masing-masing algoritma dalam menyelesaikan test case.

Tabel 5.2.1 Data panjang rute hasil pengujian

Nomor Test Case	Waktu Eksekusi dalam milisecond		
	UCS	GBFS	A*
1.	3	0	1
2.	5	0	1
3.	6	0	2
4.	4	0	3
5.	11	0	6
6.	5	0	3
Rata-rata	5.67	0	2.67

Dari data tersebut dapat dilihat bahwa algoritma A* lebih cepat dibandingkan dengan algoritma UCS walaupun kedua algoritma memberikan solusi yang optimal. Hal tersebut sesuai teoritis pada analisis implementasi algoritma A*. Untuk algoritma GBFS, waktu eksekusi adalah yang paling cepat. Hal tersebut dikarenakan GBFS yang memilih solusi optimal lokal sehingga sangat cepat. Namun karena perlu diperhatikan GBFS kurang tepat digunakan untuk mencari solusi optimal global.

5.2.3 Memori

Untuk permasalahan memori, dapat melihat data berupa aproksimasi memori yang diperlukan untuk masing-masing algoritma dalam menyelesaikan test case.

Tabel 5.2.3.1 Data aproksimasi memori hasil pengujian

Nomor Test Case	Aproksimasi Penggunaan Memori dalam kilobyte (kb)		
	UCS	GBFS	A*
1.	438	0	0
2.	524	0	524
3.	524	0	524
4.	1048	0	524
5.	1048	0	2097
6.	511	0	524

Perlu diperhatikan bahwa data adalah aproksimasi dengan melihat perubahan memori dari Java Virtual Machine (JVM) khususnya menggunakan kelas Runtime. Karena Java tidak memiliki cara untuk mendapatkan jumlah memori yang digunakan dengan persis, maka penulis hanya dapat memberi aproksimasi menggunakan JVM. Data memori tersebut tidak akurat namun dapat memberi gambaran perbandingan memori masing-masing algoritma. Java juga memiliki garbage collector, sehingga pengecekan memori sangat konsisten dan tidak akurat.

Penulis juga tidak dapat menggunakan *process manager* untuk mendapatkan jumlah memori. Hal tersebut dikarenakan penggunaan GUI yang memakan memori sehingga perubahan memori untuk pencarian rute tidak terlihat.

Dari data tersebut dapat dilihat bahwa algoritma GBFS menggunakan memori yang sangat kecil. Hal tersebut dikarenakan GBFS tidak menyimpan banyak data dibandingkan dengan algoritma UCS dan A*. UCS dan A* memiliki priority queue berisi simpul-simpul yang isinya dapat menumpuk sehingga memori yang digunakan lebih besar. Namun dari source code sendiri dapat dilihat bahwa tidak ada proses yang menggunakan jumlah memori yang sangat signifikan.

BAB VI

KESIMPULAN DAN SARAN

6.1 Kesimpulan

Pada tugas kecil 3 IF2211 Strategi Algoritma Semester 2 Tahun Ajaran 2023/2024 ini, penulis diminta untuk membuat program yang dapat menyelesaikan persoalan Word Ladder dengan menggunakan algoritma Uniform Cost Search, Greedy Best First Search, dan A* algorithm. Penulis berhasil membuat program yang dapat menyelesaikan Word Ladder dengan kecepatan yang cukup baik. Penulis menggunakan bahasa pemrograman Java dalam membuat sebuah program dengan GUI yang dapat menyelesaikan Word Ladder

6.2 Saran

Saran untuk penulis diantaranya :

1. Memulai pengerjaan dari lebih awal
2. Mendalami pembuatan GUI yang lebih baik

6.3 Refleksi

Refleksi yang didapatkan penulis dari tugas ini antara lain adalah untuk meningkatkan kinerja pengerjaan tugas dengan cara pengerjaan tugas yang lebih awal sehingga tugas dapat dikerjakan dengan lebih baik dan lebih teratur. Penulis juga mendapatkan pengalaman penting dalam pengembangan program berbasis Java dalam penyelesaian masalah. Penulis juga mendapat pengalaman dalam pengembangan algoritma UCS, Greedy Best First Search, dan A* yang teroptimasi.

LAMPIRAN

Link Repository: https://github.com/NoHaitch/Tucil3_13522091

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

DAFTAR PUSTAKA

Rinaldi Munir. Penentuan Rute (Bagian 1). Diakses pada 3 Mei 2024 dari <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf>

Rinaldi Munir. Penentuan Rute (Bagian 2). Diakses pada 3 Mei 2024 dari <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf>

Geeksforgeeks. Uniform-Cost Search (Dijkstra for large Graphs). Diakses pada 3 Mei 2024 dari <https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/>

Geeksforgeeks. Greedy Best First Search Algorithm. Diakses pada 4 Mei 2024 dari <https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/>

Geeksforgeeks. A* Search Algorithm. Diakses pada 4 Mei 2024 dari <https://www.geeksforgeeks.org/a-search-algorithm/>

Geeksforgeeks. Admissibility of A* Algorithm. Diakses pada 4 Mei 2024 dari <https://www.geeksforgeeks.org/a-is-admissible/>.