

LAPORAN TUGAS KECIL 3
Penyelesaian Permainan Word Ladder Menggunakan
Algoritma UCS, Greedy Best First Search, dan A*



Disusun oleh:
13522050 - Kayla Namira Mariadi

Dosen Pengampu : Dr. Nur Ulfa Maulidevi, S.T, M.Sc.
IF2211 - Strategi Algoritma

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN
INFORMATIKA INSTITUT TEKNOLOGI BANDUNG
2024

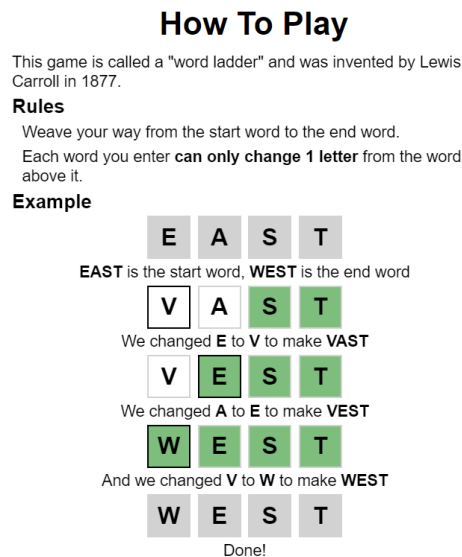
DAFTAR ISI

BAB 1 DESKRIPSI MASALAH	4
BAB 2 TEORI DASAR	6
BAB 3	8
ANALISIS DAN IMPLEMENTASI	8
3.1. Implementasi Greedy Best First Search	8
3.2. Implementasi UCS	8
3.3. Implementasi A*	9
3.4. Implementasi Program	11
3.3.1. Class Dictionary	11
3.3.2. Class Word	12
3.3.3. Class Solver	13
3.3.4. Class GBFS	15
3.3.5. Class Ucs	16
3.3.6. Class AStar	17
3.3.7. Class Game dan Board	18
3.5. Penjelasan Terkait Implementasi Bonus	20
Bab 4	21
EKSPERIMEN	21
4.1 Eksperimen Algoritma	21
BAB 5 PENUTUP	32
5.1. Kesimpulan	32
5.2. Saran	32
5.3. Komentar dan Refleksi	32
DAFTAR PUSTAKA	34
LAMPIRAN	35

BAB 1

DESKRIPSI MASALAH

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/>)

Tugas kecil berikut merupakan solver permainan tersebut dengan harapan dapat menemukan solusi paling optimal untuk menyelesaikan permainan *Word Ladder* ini.

Batasan dari implementasi adalah sebagai berikut:

- Implementasi menggunakan bahasa Java
- Solusi menggunakan algoritma UCS, A*, dan *Greedy Best First Search*.
- Masukan program adalah 2 kata dengan panjang sama dan valid terdapat pada kamus bahasa Inggris, beserta pilihan algoritma pencarian (UCS, Greedy Best First Search, atau A*)
- Luaran program adalah rute yang dihasilkan dari *start word* ke *end word* yaitu sekumpulan kata yang menggambarkan transformasi dari kata pertama (*start word*) hingga kata kedua (*end word*), tiap transformasi hanya berbeda 1 huruf. Harapannya jumlah transformasi yang diperlukan minimal. Program juga mengeluarkan jumlah

simpul/kata yang dikunjungi, beserta waktu eksekusi program

- Program juga boleh memberikan visualisasi permainan pada GUI
- Kamus diambil dari <https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/dictionary.txt>

BAB 2 TEORI DASAR

2.1. Algoritma Greedy Best First Search

Greedy Best First Search merupakan algoritma pencarian yang sesuai dengan namanya yang berarti rakus, tama. Algoritma ini mengambil keputusan berdasarkan simpul terbaik saat itu (terbesar untuk kasus maksimasi, atau terkecil untuk kasus minimasi) tanpa mempertimbangkan konsekuensi ke depan. Algoritma ini menggunakan fungsi heuristik untuk memperkirakan biaya tersisa atau jarak yang tersisa (heuristic cost) dari simpul saat ini ke node tujuan. Keputusan yang diambil saat ini diharapkan dapat mengantarkan kepada solusi terbaik di akhir. Keputusan yang telah diambil saat ini tidak dapat dibatalkan(tak ada *backtrack*). Hal ini mungkin menyebabkan algoritma ini tidak dapat memberi solusi yang optimal.

2.2. Algoritma A*

Algoritma A* (A-star) adalah algoritma pencarian informasi yang digunakan untuk mencari jalur terpendek dalam graf berbobot. A* memproses struktur data graf, yang terdiri dari simpul (node) yang mewakili objek, dan sisi (edge) yang mewakili relasi antara objek tersebut. Setiap sisi pada graf memiliki bobot (weight) yang menunjukkan biaya atau jarak antara dua simpul yang terhubung. A* memerlukan node awal (start node) dan node tujuan (goal node) untuk memulai pencarian jalur terpendek.

A* menggunakan fungsi heuristik untuk memperkirakan biaya tersisa atau jarak yang tersisa (heuristic cost) dari simpul saat ini ke node tujuan. Fungsi heuristik ini membantu A* dalam memilih simpul yang paling berpotensi mengarah ke jalur terpendek, sehingga bisa mempercepat pencarian. A* menghitung biaya akumulatif dari jalur yang telah dieksplorasi, ditambah dengan nilai heuristik, untuk membandingkan jalur-jalur yang berbeda dan memilih jalur dengan biaya terendah dan heuristik terbaik. A* menggunakan visited set (set yang telah dieksplorasi) dan priority queue untuk melacak simpul-simpul yang akan dieksplorasi dan yang telah dieksplorasi, serta untuk mengurutkan simpul-simpul pada graf berdasarkan biaya akumulatif dan nilai heuristik. Simpul dengan nilai heuristik dan biaya akumulatif yang lebih rendah diberikan prioritas lebih tinggi untuk dieksplorasi lebih dahulu.

2.3. Algoritma Uniform Cost Search (UCS)

UCS menggunakan struktur data graf. Graf adalah representasi visual dari objek dan relasi di antara mereka. Graf terdiri dari simpul (node) yang mewakili objek, dan sisi (edge) yang mewakili relasi antara objek tersebut. Setiap sisi pada graf memiliki bobot (weight) yang menunjukkan biaya atau jarak antara dua simpul yang

terhubung. Bobot bisa berupa angka real atau integer, dan bisa mewakili berbagai jenis informasi seperti jarak fisik, waktu tempuh, atau biaya. UCS memerlukan node awal (start node) dan node tujuan (goal node) untuk memulai pencarian jalur terpendek. Node awal adalah simpul dari mana pencarian dimulai, sedangkan node tujuan adalah simpul yang ingin dicapai.

Sama seperti A^* , UCS menggunakan visited set (set yang telah dieksplorasi) untuk melacak simpul-simpul yang akan dieksplorasi dan yang telah dieksplorasi. Dalam prosesnya, UCS juga menghitung biaya akumulatif (cost-to-come) dari jalur yang telah dieksplorasi. Biaya akumulatif ini digunakan untuk membandingkan jalur-jalur yang berbeda dan memilih jalur dengan biaya terendah. Biaya ini lalu dimasukkan ke dalam priority queue untuk mengurutkan simpul-simpul berdasarkan biaya akumulatif jalur yang telah dieksplorasi. Simpul dengan biaya akumulatif yang lebih rendah diberikan prioritas lebih tinggi untuk dieksplorasi lebih dahulu.

BAB 3

ANALISIS DAN IMPLEMENTASI

3.1. *Implementasi Greedy Best First Search*

Solusi dengan pendekatan Greedy Best First Search adalah dengan mengevaluasi nilai heuristik dari suatu simpul/kata. Kata atau simpul yang terpilih merupakan simpul dengan nilai heuristik terendah. Pada algoritma ini, simpul yang diekspan merupakan simpul terbaik saat itu (tidak ada *backtracking* untuk mencari jalur yang lebih optimal). Akibatnya pencarian mungkin terjebak dalam optimal lokal sehingga **tidak menjamin menemukan solusi yang optimal**. Pada algoritma ini, didefinisikan fungsi evaluasi $f(n)$ simpul sebagai nilai heuristik $h(n)$, dimana **nilai heuristik suatu simpul adalah banyaknya perbedaan huruf antara kata tersebut dengan kata tujuan**. Fungsi dapat dirumuskan sebagai berikut:

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

Berikut langkah-langkah implementasi algoritma GBFS.

1. Inisialisasi queue yang akan menyimpan simpul-simpul kandidat, HashSet *visited* untuk menyimpan simpul yang sudah dikunjungi, serta List *result* kosong. Pada awalnya, queue hanya berisi kata pertama (start word)
2. Selama queue belum kosong, suatu simpul dengan nilai heuristik terkecil akan diekspan dari queue (simpul ini dinamakan *currentWord*) serta ditambahkan ke dalam Set *visited*
3. Jika kata yang sedang diekspan (*currentWord*) merupakan kata tujuan, List *result* diset berisi sekumpulan simpul yang merepresentasikan jalur dari simpul awal ke tujuan (detailnya akan dijelaskan pada bagian implementasi program)
4. Tentukan simpul-simpul tetangga dengan mengganti satu per satu huruf pada *currentWord* dengan huruf "a"- "z", jika kata yang dihasilkan valid (ada pada *dictionary*) dan belum dikunjungi sebelumnya, kata tersebut dimasukkan ke dalam queue
5. Ulangi proses 2-4 hingga queue kosong atau rute ditemukan. Jika memang tidak ada jalur dari kata awal hingga tujuan, *result* akan tetap kosong.

Penulis memanfaatkan set *visited* untuk menghindari suatu simpul dikunjungi lebih dari sekali (Jika simpul dilalui dua kali, belum tentu hasilnya optimal). Hal ini dilakukan sebagai optimisasi pencarian (walaupun optimal tetap lokal).

3.2. *Implementasi UCS*

Pada pendekatan *UCS*, diimplementasikan fungsi evaluasi simpul yang mempertimbangkan biaya sesungguhnya simpul. Pada algoritma ini, didefinisikan fungsi evaluasi $f(n)$ nilai biaya $g(n)$, dimana **nilai biaya adalah jumlah perubahan kata yang sudah dilakukan dari simpul/kata awal**. Tentunya, perbedaan biaya antara simpul yang anak dan induknya adalah 1. Khusus kata pertama, biayanya 0.

Fungsi dapat dirumuskan sebagai berikut:

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

$$g(child) = 1 + g(parent)$$

Simpul terpilih merupakan simpul dengan biaya/jumlah transformasi dari kata awal terendah. Langkah-langkah implementasi algoritma UCS adalah sebagai berikut.

1. Inisialisasi Priority Queue yang akan menyimpan simpul-simpul kandidat, dimana nilai prioritas ditentukan sebagai nilai $f(n)$ yang telah didefinisikan sebelumnya di atas. HashSet *visited* untuk menyimpan simpul yang sudah dikunjungi, serta List *result* kosong. Pada awalnya, queue hanya berisi kata pertama (start word) dengan nilai *cost* atau $g(n) = 0$.
2. Selama queue belum kosong, suatu simpul dengan nilai $f(n)$ terkecil akan di-*dequeue* dari queue (simpul ini dinamakan *currentWord*) serta ditambahkan ke dalam Set *visited*
3. Jika simpul yang sedang di-*dequeue* (*currentWord*) merupakan kata tujuan, List *result* diset berisi sekumpulan simpul yang merepresentasikan jalur dari simpul awal ke tujuan
4. Tentukan simpul-simpul tetangga dengan mengganti satu per satu huruf pada *currentWord* dengan huruf "a"-z, jika kata yang dihasilkan valid (ada pada *dictionary*) dan belum dikunjungi sebelumnya, kata tersebut dimasukkan ke dalam priority queue
5. Ulangi langkah 2-4 hingga queue kosong atau rute ditemukan. Jika memang tidak ada jalur dari kata awal hingga tujuan, *result* akan tetap kosong.

Algoritma UCS menjamin ditemukannya solusi optimal (jika solusi memang ada) karena dilakukan *backtracking*. Dapat diobservasi nantinya bahwa iterasi yang dilakukan pada algoritma ini lebih banyak dibandingkan kedua algoritma lainnya. Karena biaya antar simpul anak dan induk selalu 1 dan setiap anak dari satu induk yang sama memiliki biaya/*cost* yang identik, **UCS pada konteks ini merupakan BFS**. Kedua algoritma sama-sama akan mengeksplorasi semua kata yang berbeda 1 huruf pada suatu level sebelum berpindah ke level selanjutnya.

Secara teoritis, kompleksitas waktu dan ruang untuk algoritma ini adalah $O(b^d)$ dengan b adalah branching factor dan d kedalaman solusi yang optimal. Algoritma ini memang memiliki kompleksitas waktu yang eksponensial sehingga jika dibandingkan dengan GBFS, lebih lama.

3.3. Implementasi A*

Pada pendekatan A*, diimplementasikan fungsi evaluasi simpul yang mempertimbangkan nilai heuristik dari simpul tersebut beserta biaya sesungguhnya dari simpul. Mirip dengan UCS dan GBFS, pada algoritma ini, didefinisikan fungsi evaluasi **$f(n)$ sebagai nilai heuristik $h(n)$ ditambah dengan nilai biaya $g(n)$** , dimana nilai heuristik suatu simpul adalah banyaknya perbedaan huruf antara kata

tersebut dengan kata tujuan, dan nilai biaya adalah jumlah perubahan kata yang sudah dilakukan dari simpul/kata awal. Fungsi dapat dirumuskan sebagai berikut:

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

Simpul terpilih merupakan simpul dengan nilai heuristik dan biaya terendah. Persamaan algoritma ini dengan GBFS yaitu sama-sama mempertimbangkan nilai heuristik dalam pembangkitan dari queue. Langkah-langkah implementasi algoritma A* adalah sebagai berikut.

1. Inisialisasi Priority Queue yang akan menyimpan simpul-simpul kandidat, dimana nilai prioritas ditentukan sebagai nilai $f(n)$ yang telah didefinisikan sebelumnya di atas. HashSet *visited* untuk menyimpan simpul yang sudah dikunjungi, serta List *result* kosong. Pada awalnya, queue hanya berisi kata pertama (start word) dengan nilai *cost* atau $g(n) = 0$.
2. Selama queue belum kosong, suatu simpul dengan nilai $f(n)$ terkecil akan di-*dequeue* dari queue (simpul ini dinamakan *currentWord*) serta ditambahkan ke dalam Set *visited*
3. Jika kata yang sedang di-*dequeue* (*currentWord*) merupakan kata tujuan, List *result* diset berisi sekumpulan simpul yang merepresentasikan jalur dari simpul awal ke tujuan
4. Tentukan simpul-simpul tetangga dengan mengganti satu per satu huruf pada *currentWord* dengan huruf "a"-*z*", jika kata yang dihasilkan valid (ada pada *dictionary*) dan belum dikunjungi sebelumnya, kata tersebut dimasukkan ke dalam priority queue
5. Ulangi langkah 2-4 hingga queue kosong atau rute ditemukan. Jika memang tidak ada jalur dari kata awal hingga tujuan, *result* akan tetap kosong.

Untuk algoritma ini, dijamin akan ditemukan solusi optimal jika heuristik *admissible* (dan solusi memang benar ada). Heuristik dari suatu simpul n , atau $h(n)$, dikatakan *admissible* jika untuk setiap simpul, $h(n) \leq h^*n$, dimana h^*n adalah biaya sesungguhnya untuk mencapai simpul tujuan. Pada implementasi *word ladder* ini, **dapat dijamin heuristik *admissible* karena banyaknya perbedaan huruf antara kata dengan kata tujuan pasti lebih sedikit atau sama dengan banyaknya transformasi sesungguhnya yang dibutuhkan untuk mencapai kata tujuan.** Misal, dari kata *charge* menuju *comedo* memiliki perbedaan 6 huruf sehingga heuristik bernilai 6, tapi pada kenyataannya mengganti huruf sebanyak 6 kali tidak cukup untuk mendapatkan kata *comedo*, karena kata yang diganti tidak ada pada *dictionary*.

Walaupun sebenarnya algoritma A* tidak mempertimbangkan apakah suatu simpul sudah dikunjungi sebelumnya dan hanya bergantung pada nilai $f(n)$, penulis tetap tidak menambahkan simpul yang sudah di-*dequeue* ke simpul kandidat untuk menghemat waktu pencarian dan ruang. Hal ini dikarenakan pada konteks permainan *word ladder*, suatu kata/simpul yang sama pasti memiliki nilai heuristik yang sama juga. Jika sudah ditemukan suatu kata, dan kemudian kata tersebut

ditemukan kembali melalui jalur berbeda, kata tersebut pasti memiliki biaya yang lebih besar (hal ini disebabkan tiap iterasi bertambah, kata selalu memiliki perbedaan 1 huruf dengan simpul sebelumnya, sehingga biaya bertambah 1).

Sebenarnya, tak jauh berbeda dengan UCS, kompleksitas waktu dan ruang untuk algoritma ini adalah $O(b^d)$ dengan b adalah branching factor dan d kedalaman solusi yang optimal.

3.4. Implementasi Program

Dalam implementasi algoritma, digunakan bahasa Java dan untuk visualisasi menggunakan Java Swing. Source code file algoritma dan gui terdapat pada direktori src. Pada implementasi program, terdapat 3 kelas utama, yaitu kelas Dictionary (merepresentasikan dictionary/kamus), kelas Word (merepresentasikan simpul), dan kelas Solver (sebagai base class dari ketiga algoritma). Selain itu juga terdapat kelas Game dan Board untuk keperluan GUI.

3.3.1. Class Dictionary

Class ini adalah kelas yang merepresentasikan kata-kata dengan panjang yang sama pada kamus. Kelas ini memiliki constructor yang menerima parameter length, kemudian membaca file dictionary dan memasukkannya kata-kata dengan panjang sesuai input ke Hashset. Tujuan kata-kata yang di-load adalah kata dengan panjang yang sama dengan kata input untuk mengurangi traversal yang terlalu banyak pada dictionary dalam proses pencarian nantinya. Berikut adalah atribut pada kelas:

- words : sekumpulan kata dari dictionary yang disimpan pada Hashset
- wordLength : panjang sebuah kata dari dictionary
- wordCount : banyak kata pada dictionary (ukuran HashSet words)

```
package src;

import java.util.*;
import java.io.*;

public class Dictionary {
    protected HashSet<String> words;
    protected int wordLength;
    protected int wordCount = 0;

    public Dictionary(int length) throws FileNotFoundException {
        this.wordLength = length;
        Scanner s = new Scanner(new File("src/dictionary.txt"));
        words = new HashSet<>();
        while (s.hasNext()) {
            String nextWord = s.next();
            if (nextWord.length() == length) {
                words.add(nextWord);
            }
        }
    }
}
```

```
        wordCount++;
    }
}
s.close();
}

public boolean contains(String word){
    return this.words.contains(word.toLowerCase());
}

//Debugging purpose
public void printDictionary(){
    for (String word : words){
        System.out.println(word);
    }
    System.out.printf("\nWord in dictionary: %d\n", wordCount);
}
}
```

3.3.2. Class Word

Class ini adalah kelas yang merepresentasikan simpul berupa kata. Nilai cost pada atribut merepresentasikan nilai $f(n)$ yang telah didefinisikan sebelumnya pada bagian Implementasi.

```
package src;

public class Word {
    protected String word;        // word name
    protected int cost;           // cost value
    protected Word parent;        // previous Word

    /*Constructor for startWord */
    public Word(String word){
        this.parent = null;
        this.word = word;
        this.cost = 0;
    }

    /*Constructor for word */
    public Word(String word, Word parentWord, int cost){
        this.word = word;
        this.cost = cost;
        this.parent = parentWord;
    }

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

    public int getCost(){
```

```
        return this.cost;
    }

    public Word getparent(){
        return this.parent;
    }

    // Debugging purpose
    public void printPath(){
        System.out.printf("f(n): %d,", this.cost);
        System.out.printf("Path to %s: ", this.word);
        Word ancestor = parent;
        String path = this.word;
        while (ancestor != null){
            path = ancestor.getWord() + "->" + path;
            ancestor = ancestor.getparent();
        }
        System.out.println(path);
        System.out.println();
    }
}
```

3.3.3. *Class Solver*

Class ini merupakan base class bagi ketiga algoritma. Class ini memiliki abstract method solve yang implementasinya berbeda tergantung algoritma. Selain itu juga terdapat method calculateCost untuk menghitung $g(n)$ serta calculateHeuristic untuk menghitung $f(n)$, fungsi ini nantinya akan dibutuhkan oleh class anak dalam mengimplementasikan method solve. Fungsi solve yang diimplementasikan nantinya berbeda dalam prioritas queue. Terdapat pula method helper seperti generatePath untuk mendapatkan path dengan menelusuri dari simpul akhir ke simpul awal serta printResult untuk mencetak hasil pada terminal.

```
package src;

import java.util.*;

public abstract class Solver {
    public Dictionary dictionary;
    public String startWord;
    public String endWord;
    public int visitedWords;
    public PriorityQueue<Word> queue;
    public List<Word> result;
    public HashSet<String> visited;

    // Constructor
    public Solver(String startWord, String endWord, Dictionary
dictionary) {
        this.dictionary = dictionary;
    }
}
```

```
        this.startWord = startWord.toLowerCase();
        this.endWord = endWord.toLowerCase();
        this.visitedWords = 0;
        this.queue = new PriorityQueue<>(new Comparator<Word>() {
            public int compare(Word word1, Word word2){
                return Integer.compare(word1.getCost(),
word2.getCost());
            }
        });
        this.result = new ArrayList<>();
        this.visited = new HashSet<>();
    }

    public abstract void solve() ;

    public void printResult(){
        int pathLength = result.size();
        if (pathLength > 0){
            System.out.printf("\nVisited words: %d\n",
this.visitedWords);
            System.out.printf("Path length: %d\n", pathLength);
            System.out.println("Path: ");
            for (int i = 0; i < pathLength-1 ;i++){
                System.out.printf("%s -> ",
result.get(i).getWord());
            }
            System.out.println(result.get(pathLength-1).getWord());
        }
        else{
            System.out.println("\nNo result found!");
        }
    }

    // generate path from start word to end word
    public void generateEndPath(Word w){
        Word ancestor = w.getparent();
        this.result.add(w);
        while (ancestor != null){
            this.result.add(0, ancestor);
            ancestor = ancestor.getparent();
        }
    }

    /*calculate heuristic by finding offset to endword */
    public int calculateHeuristic(String word, String endword){
        int offset = 0;
        int len = word.length();
        for (int i = 0; i < len; i++){
            if (word.charAt(i) != endword.charAt(i)){
                offset ++;
            }
        }
        return offset;
    }
}
```

```
    }

    /*calculate cost for children */
    public int calculateCost(Word parentWord){
        return parentWord.getCost() + 1;
    }
}
```

3.3.4. Class GBFS

Class ini merupakan kelas yang mengimplementasikan algoritma *Greedy BFS*.

```
package src;

public class GBFS extends Solver{
    public GBFS(String startWord, String endWord, Dictionary
dictionary) {
        super(startWord, endWord, dictionary);
    }

    public void solve(){
        // nNo word in dict
        if (!dictionary.contains(startWord) ||
!dictionary.contains(endWord)){
            return;
        }
        // same input for start word and endword
        if (startWord.equals(endWord)){
            return;
        }

        queue.add(new Word(startWord));

        while (!queue.isEmpty()) {

            Word currentWord = queue.poll(); //dequeue the first
Word in queue
            visited.add(currentWord.getWord()); // label current
word as visited
            visitedWords++;

            // path is found
            if (currentWord.getWord().equals(endWord)){
                generateEndPath(currentWord);
                return;
            }

            // generate neighbors (words in dict with 1 offset from
current word)
            char[] characters = currentWord.getWord().toCharArray();
```

```

        for (int j = 0; j < characters.length; ++j) {
            char originalChar = characters[j];
            for (char ch = 'a'; ch <= 'z'; ++ch) {
                characters[j] = ch;
                String newWord = new String(characters);
                // Skip if new word is not in the word set or
                // already visited
                if (!dictionary.contains(newWord) ||
                    visited.contains(newWord)) {
                    continue;
                }
                // Add new word to the queue
                queue.add(new Word(newWord, currentWord,
                    calculateHeuristic(newWord, endWord)));
            }
            characters[j] = originalChar;
        }
    }
}

```

3.3.5. Class Ucs

Class ini merupakan kelas yang mengimplementasikan algoritma UCS.

```

package src;

public class Ucs extends Solver {

    public Ucs(String startWord, String endWord, Dictionary
dictionary) {
        super(startWord, endWord, dictionary);
    }

    public void solve(){
        // nNo word in dict
        if (!dictionary.contains(startWord) ||
            !dictionary.contains(endWord)) {
            return;
        }
        // same input for start word and endword
        if (startWord.equals(endWord)) {
            return;
        }

        queue.add(new Word(startWord));

        while (!queue.isEmpty()) {

```

```

        Word currentWord = queue.poll(); //dequeue the first
Word in queue
        visited.add(currentWord.getWord()); // label current
word as visited
        visitedWords++;

        // path is found
        if (currentWord.getWord().equals(endWord)) {
            generateEndPath(currentWord);
            return;
        }

        // generate neighbors (words in dict with 1 offset from
current word)
        char[] characters = currentWord.getWord().toCharArray();
        for (int j = 0; j < characters.length; ++j) {
            char originalChar = characters[j];
            for (char ch = 'a'; ch <= 'z'; ++ch) {
                characters[j] = ch;
                String newWord = new String(characters);
                // Skip if new word is not in the word set or
visited
                if (!dictionary.contains(newWord) ||
visited.contains(newWord)) {
                    continue;
                }
                // Add new word to the queue
                queue.add(new Word(newWord, currentWord,
calculateCost(currentWord)));
            }
            characters[j] = originalChar;
        }
    }
}

```

3.3.6. Class AStar

Class ini merupakan kelas yang mengimplementasikan algoritma A*.

```

package src;

public class AStar extends Solver {

    public AStar(String startWord, String endWord, Dictionary
dictionary) {
        super(startWord, endWord, dictionary);
    }

    public void solve(){

```



```

        // No word in dict
        if (!dictionary.contains(startWord) ||
!dictionary.contains(endWord)) {
            return;
        }
        // same input for start word and endword
        if (startWord.equals(endWord)) {
            return;
        }

        queue.add(new Word(startWord));

        while (!queue.isEmpty()) {

            Word currentWord = queue.poll(); //dequeue the first
Word in queue
            visited.add(currentWord.getWord()); // label current
word as visited
            visitedWords++;

            // path is found
            if (currentWord.getWord().equals(endWord)) {
                generateEndPath(currentWord);
                return;
            }

            // generate neighbors (words in dict with 1 offset from
current word)
            char[] characters = currentWord.getWord().toCharArray();
            for (int j = 0; j < characters.length; ++j) {
                char originalChar = characters[j];
                for (char ch = 'a'; ch <= 'z'; ++ch) {
                    characters[j] = ch;
                    String newWord = new String(characters);
                    // Skip if new word is not in the word set or
visited
                    if (!dictionary.contains(newWord) ||
visited.contains(newWord)) {
                        continue;
                    }
                    // Add new word to the queue
                    int cost = calculateHeuristic(newWord, endWord)
+ calculateCost(currentWord);
                    queue.add(new Word(newWord, currentWord, cost));
                }
                characters[j] = originalChar;
            }
        }
    }
}

```

3.3.7. Class Game dan Board

Kelas berikut merupakan kelas yang menginisiasi GUI dan mengandung method main. Keseluruhan kode tidak dimasukkan ke laporan untuk mempersingkat dan dapat dilihat melalui repository. Berikut dilampirkan cuplikan kode initAlgorithm pada Game.java yang akan membuat objek Solver dan menjalankan method solve untuk mencari solusi. Sebelum itu, juga dihandle berbagai kasus seperti panjang kata tak sama, atau kata tak terdapat di dictionary. Lalu, file *dictionary* akan dibaca.

```
public void initAlgorithm(String startWord, String endWord, int
algoType) {
    clearResult("",0,0);
    try {
        int len = startWord.length();
        if (len != endWord.length()){
            String errormsg = "Startword and endword are not in
the same length!\n";
            clearResult(errormsg,0,0);
            System.out.println(errorMessage);
        }
        else if (startWord.equals(endWord)){
            String errormsg = "Start word is equal to end
word!\n";
            clearResult(errormsg,0,0);
            System.out.println(errorMessage);
        }
        else{
            Dictionary d = new Dictionary(len);
            if (!d.contains(startWord) || !d.contains(endWord)){
                String errormsg = "Your word is not on
dictionary!\n";
                clearResult(errormsg,0,0);
                System.out.println(errorMessage);
            }

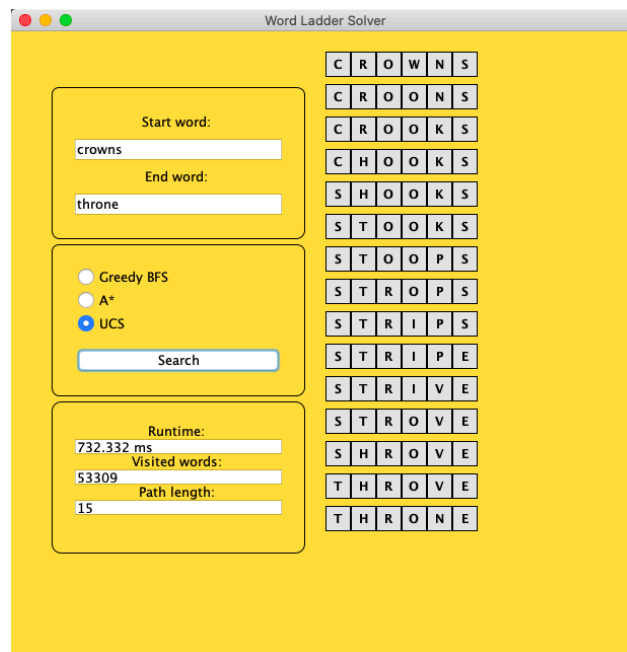
            long startTime = System.nanoTime();
            if (algoType == 1){
                this.solver = new GBFS(startWord, endWord, d);
            }
            else if (algoType == 2){
                this.solver = new AStar(startWord, endWord, d);
            }
            else{
                this.solver = new Ucs(startWord, endWord, d);
            }
            this.solver.solve();
            long endTime = System.nanoTime();
            long duration = (endTime - startTime);
            this.solver.printResult();

            String formatedSeconds = String.format("%d ms",
duration/1000000);
            System.out.println("total runtime = "+
```

```
formattedSeconds);
    this.runtime = formattedSeconds;
    this.visitedLength = solver.visitedWords;
    if (this.solver.result.size() > 0) {
        this.setBoard(this.solver);
        this.errorMsg = "";
        this.pathLength = solver.result.size();
    }
    else{
        this.errorMsg = "No path found!\n";
        clearResult(errorMsg, 0, visitedLength);
    }
    // return this.solver.result;
}
} catch (FileNotFoundException e) {
    String errormsg = "No dictionary file found\n";
    clearResult(errormsg, 0, 0);
    System.out.println(errorMsg);
    // return null;
}
}
```

3.5. *Penjelasan Terkait Implementasi Bonus*

Untuk melihat visualisasi, ikuti langkah setup pada README.



contoh GUI

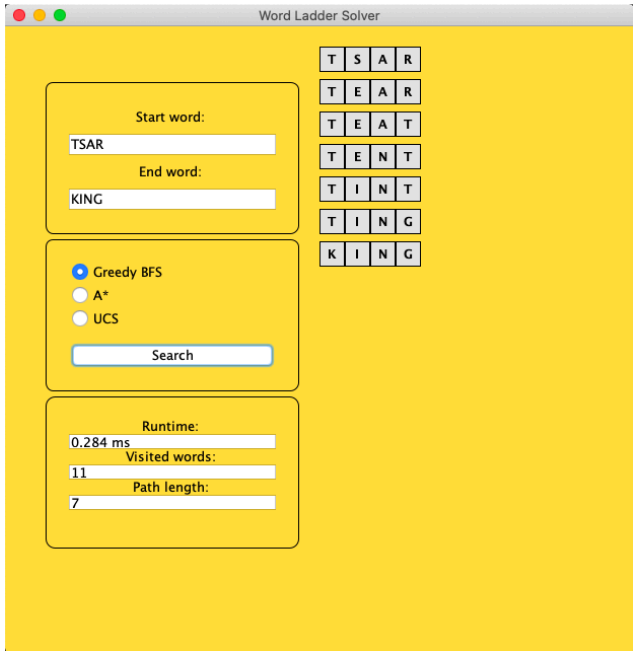
Berikut alur pemakaian GUI:

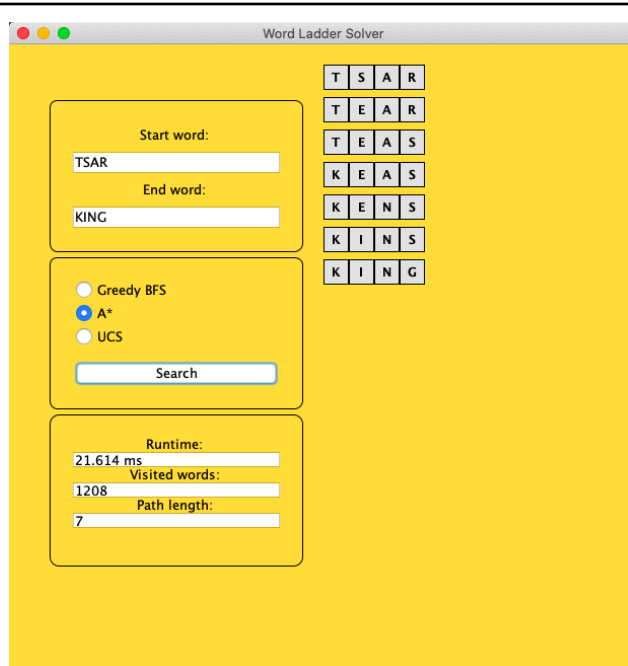
1. Masukkan start word dan end word, kata harus dengan panjang yang sama. Jika

- memiliki panjang berbeda atau tak ada pada kamus, akan dikeluarkan pesan error.
2. Masukkan pilihan algoritma yang tersedia
3. GUI akan menampilkan rute kata-kata beserta runtime, panjang rute, dan jumlah simpul yang dikunjungi. Saat menjalankan GUI, jalannya program juga dapat dipantau dari terminal.

Bab 4 EKSPERIMEN

4.1 Eksperimen Algoritma

Testcase	Perbandingan Hasil Tiap Algoritma	Detail
<p>Testcase 1: TSAR -> KING</p>		<p>Algoritma GBFS Runtime : 0.284 ms Node yang dikunjungi: 11 Panjang rute:7</p>



Word Ladder Solver

Start word: TSAR

End word: KING

☐ Greedy BFS
☒ A*
☐ UCS

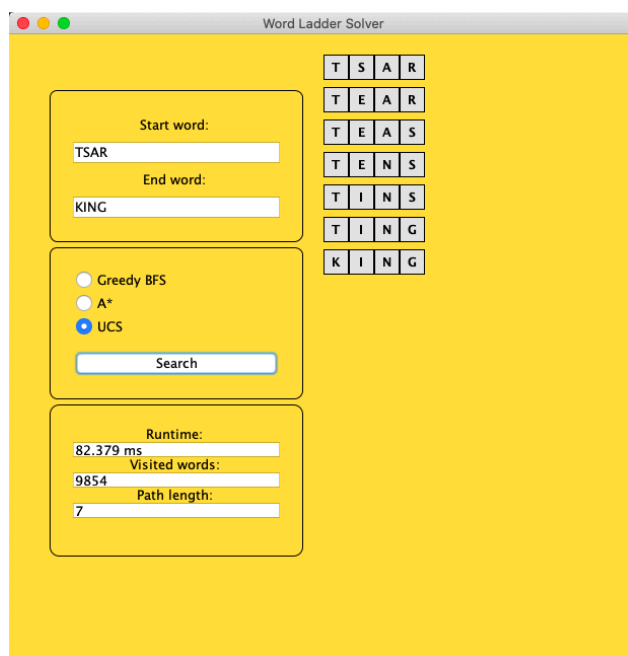
Search

Runtime: 21.614 ms
Visited words: 1208
Path length: 7

Path visualization:

T	S	A	R
T	E	A	R
T	E	A	S
K	E	A	S
K	E	N	S
K	I	N	S
K	I	N	G

Algoritma: A*
Runtime : 21.614 ms
Node yang dikunjungi: 1208
Panjang rute:7



Word Ladder Solver

Start word: TSAR

End word: KING

☐ Greedy BFS
☐ A*
☒ UCS

Search

Runtime: 82.379 ms
Visited words: 9854
Path length: 7

Path visualization:

T	S	A	R
T	E	A	R
T	E	A	S
T	E	N	S
T	I	N	S
T	I	N	G
K	I	N	G

Algoritma: UCS
Runtime : 82.379 ms
Node yang dikunjungi: 9854
Panjang rute:7

Testcase
2:
CROWN
S
->
THRONE

Word Ladder Solver

Start word:
crowns

End word:
throne

☒ Greedy BFS
☐ A*
☐ UCS

Search

Runtime:
20.797 ms

Visited words:
1391

Path length:
69

C	R	O	W	N	S
B	R	O	W	N	S
B	R	O	W	N	Y
B	R	A	W	N	Y
B	R	A	W	N	S
B	R	A	I	N	S
T	R	A	I	N	S
T	R	A	I	T	S
T	R	A	I	L	S
T	R	A	I	K	S
T	R	A	N	K	S
T	H	A	N	K	S
T	H	A	C	K	S
T	H	I	C	K	S
C	H	I	C	K	S
C	H	I	R	K	S
C	H	I	R	R	S
C	H	U	R	R	S

Algoritma: GBFS
Runtime : 20.797 ms
Node yang dikunjungi: 1391
Panjang rute:69

Word Ladder Solver

Start word:
crowns

End word:
throne

☐ Greedy BFS
☒ A*
☐ UCS

Search

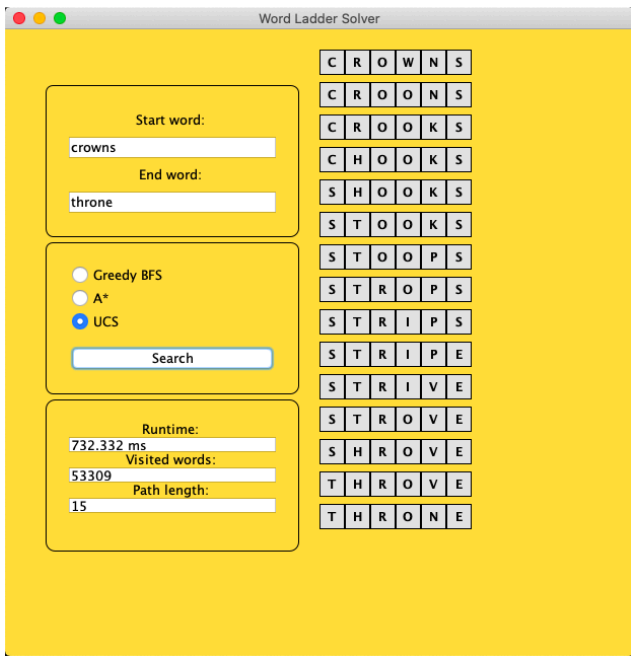
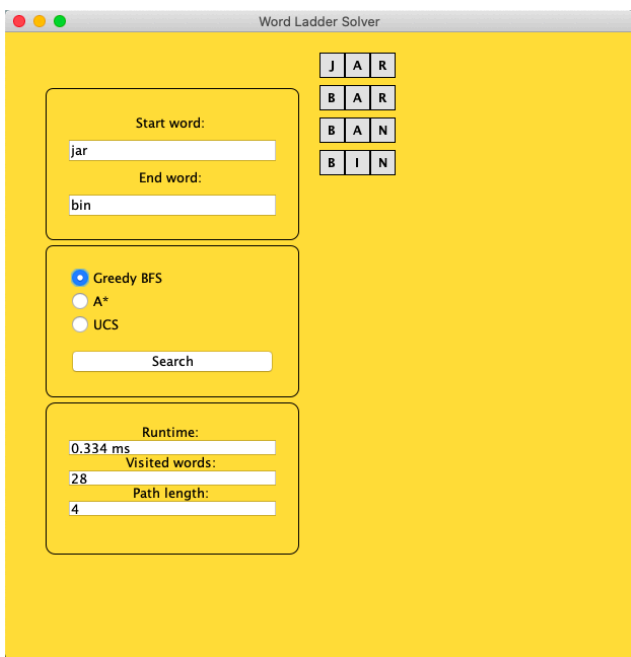
Runtime:
350.869 ms

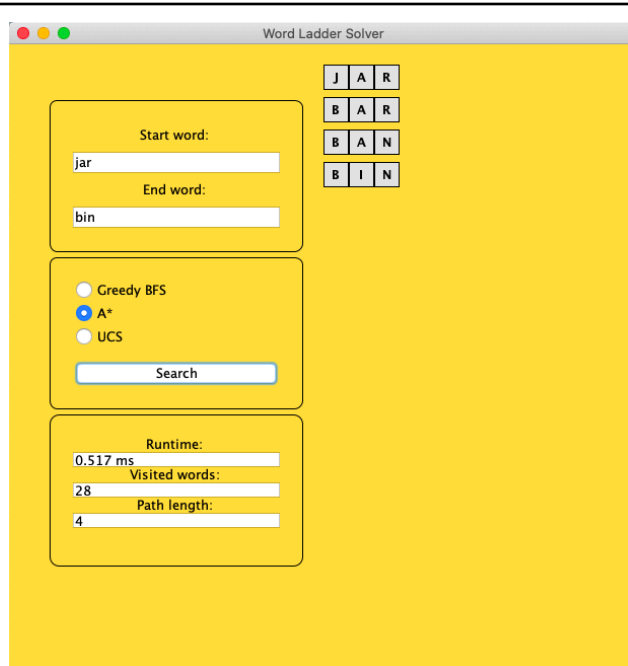
Visited words:
11660

Path length:
15

C	R	O	W	N	S
C	R	O	O	N	S
C	R	O	O	K	S
C	H	O	O	K	S
S	H	O	O	K	S
S	T	O	O	K	S
S	T	O	O	P	S
S	T	R	O	P	S
S	T	R	I	P	S
S	T	R	I	P	E
S	T	R	I	V	E
S	H	R	I	V	E
S	H	R	O	V	E
T	H	R	O	V	E
T	H	R	O	N	E

Algoritma: A*
Runtime : 350.869 ms
Node yang dikunjungi: 11660
Panjang rute:15

		<p>Algoritma: UCS Runtime : 732.332 ms Node yang dikunjungi: 53309 Panjang rute:15</p>
<p>Testcase 3: JAR -> BIN</p>		<p>Algoritma: GBFS Runtime : 0.334 ms Node yang dikunjungi: 28 Panjang rute: 4</p>



Word Ladder Solver

Start word: jar

End word: bin

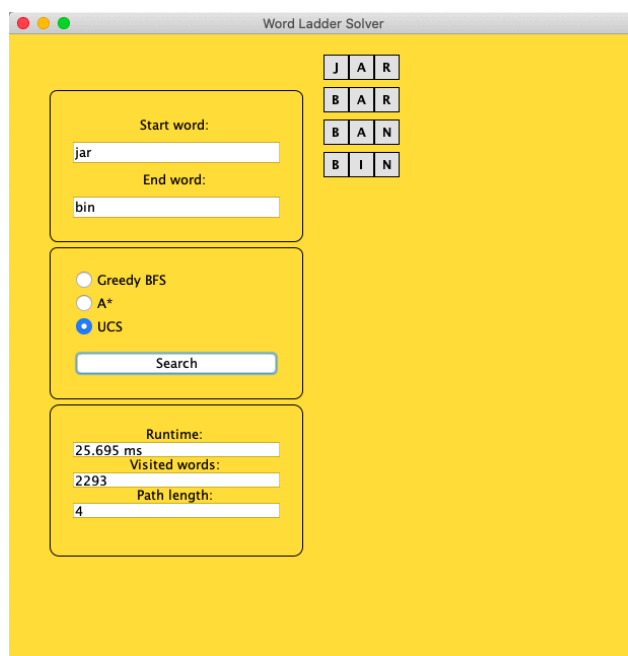
☐ Greedy BFS
☒ A*
☐ UCS

Search

Runtime: 0.517 ms
Visited words: 28
Path length: 4

J A R
B A R
B A N
B I N

Algoritma: A*
Runtime : 0.517 ms
Node yang dikunjungi: 28
Panjang rute:4



Word Ladder Solver

Start word: jar

End word: bin

☐ Greedy BFS
☐ A*
☒ UCS

Search

Runtime: 25.695 ms
Visited words: 2293
Path length: 4

J A R
B A R
B A N
B I N

Algoritma: UCS
Runtime : 25.695 ms
Node yang dikunjungi: 2293
Panjang rute:4

Testcase
4:
RAIN
->
PLAY

Word Ladder Solver

Start word: rain

End word: play

☒ Greedy BFS
☐ A*
☐ UCS

Search

Runtime: 0.437 ms
 Visited words: 6
 Path length: 6

Word ladder visualization:

R	A	I	N
P	A	I	N
P	E	I	N
P	E	A	N
P	L	A	N
P	L	A	Y

Algoritma: GBFS
Runtime : 0.437 ms
Node yang dikunjungi: 6
Panjang rute:6

Word Ladder Solver

Start word: rain

End word: play

☐ Greedy BFS
☒ A*
☐ UCS

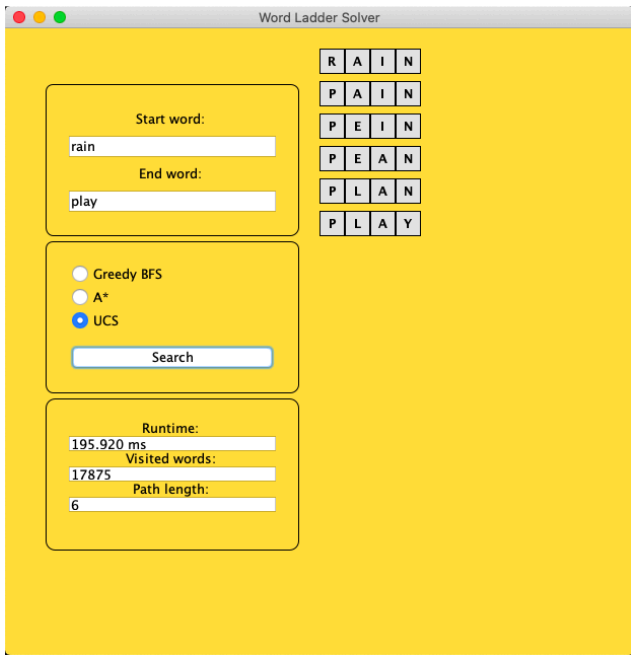
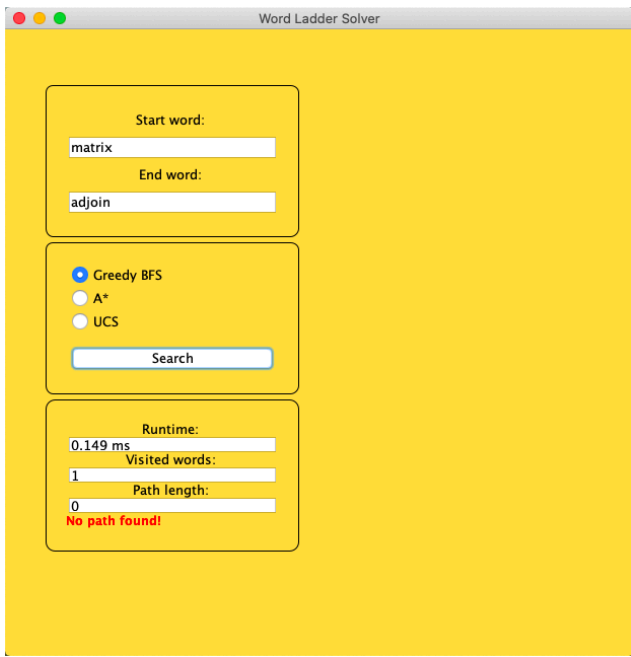
Search

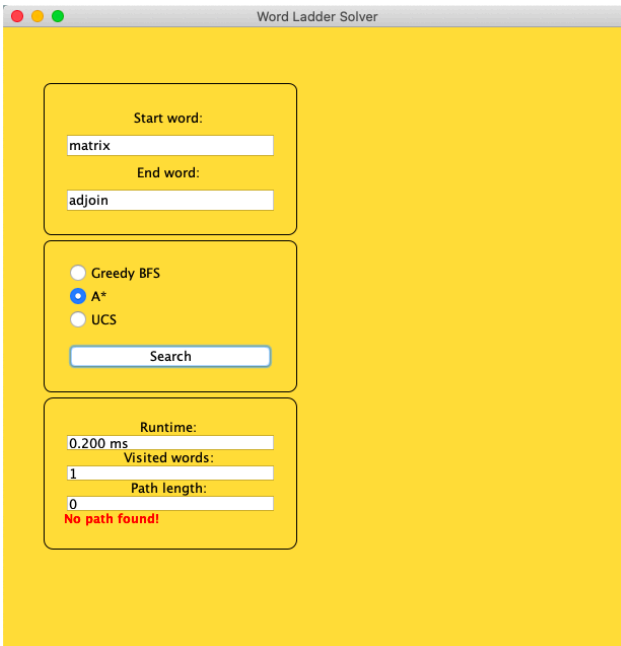
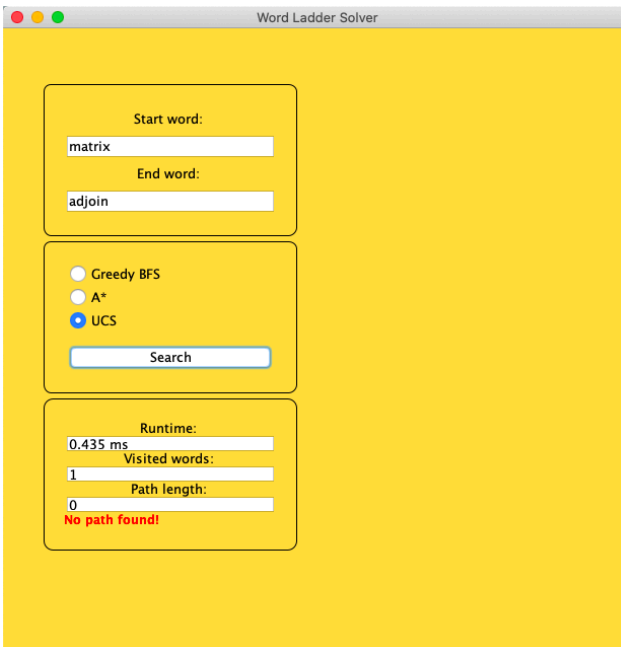
Runtime: 7.394 ms
 Visited words: 238
 Path length: 6

Word ladder visualization:

R	A	I	N
P	A	I	N
P	E	I	N
P	E	A	N
P	L	A	N
P	L	A	Y

Algoritma: A*
Runtime : 7.394 ms
Node yang dikunjungi: 238
Panjang rute:6

		<p>Algoritma: UCS Runtime : 195.920 ms Node yang dikunjungi: 17875 Panjang rute:6</p>
<p>Testcase 5: MATRIX -> ADJOIN</p>		<p>Algoritma: GBFS Runtime : 0.149 ms Node yang dikunjungi: 1 Panjang rute:0</p>

	 <p>The screenshot shows the 'Word Ladder Solver' application window. The 'Start word' field contains 'matrix' and the 'End word' field contains 'adjoin'. Under the algorithm selection, 'A*' is selected with a blue radio button, while 'Greedy BFS' and 'UCS' are unselected. A 'Search' button is visible. The results section shows 'Runtime: 0.200 ms', 'Visited words: 1', and 'Path length: 0'. A red message 'No path found!' is displayed at the bottom of the results section.</p>	<p>Algoritma: A* Runtime : 0.2 ms Node yang dikunjungi: 1 Panjang rute: 0</p>
	 <p>The screenshot shows the 'Word Ladder Solver' application window. The 'Start word' field contains 'matrix' and the 'End word' field contains 'adjoin'. Under the algorithm selection, 'UCS' is selected with a blue radio button, while 'Greedy BFS' and 'A*' are unselected. A 'Search' button is visible. The results section shows 'Runtime: 0.435 ms', 'Visited words: 1', and 'Path length: 0'. A red message 'No path found!' is displayed at the bottom of the results section.</p>	<p>Algoritma: UCS Runtime : 0.435 ms Node yang dikunjungi: 1 Panjang rute:0</p>

Testcase
6:
PERCH
->
ROOST

Word Ladder Solver

Start word: perch

End word: roost

☒ Greedy BFS
☐ A*
☐ UCS

Search

Runtime: 0.936 ms
 Visited words: 31
 Path length: 11

P	E	R	C	H
P	O	R	C	H
P	O	O	C	H
C	O	O	C	H
C	O	U	C	H
C	O	U	T	H
R	O	U	T	H
R	O	U	T	E
R	O	U	S	E
R	O	O	S	E
R	O	O	S	T

Algoritma: GBFS
 Runtime : 0.936 ms
 Node yang dikunjungi: 31
 Panjang rute:11

Word Ladder Solver

Start word: perch

End word: roost

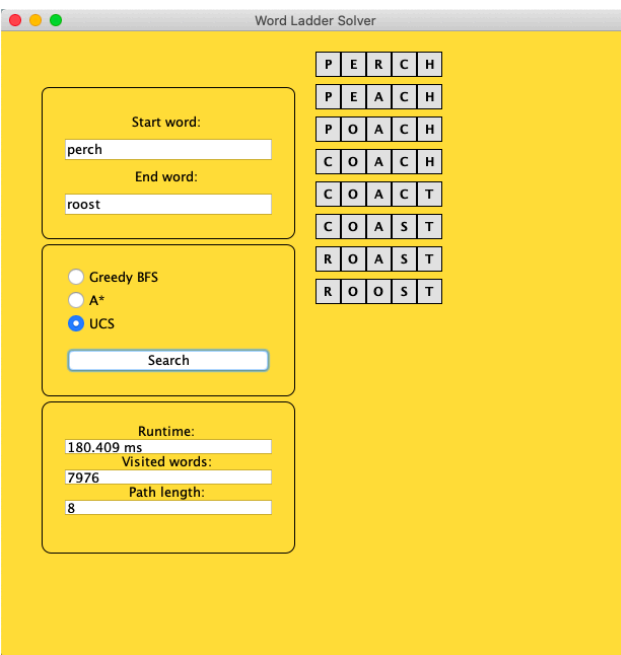
☐ Greedy BFS
☒ A*
☐ UCS

Search

Runtime: 31.451 ms
 Visited words: 505
 Path length: 8

P	E	R	C	H
P	O	R	C	H
P	O	A	C	H
C	O	A	C	H
C	O	A	C	T
C	O	A	S	T
R	O	A	S	T
R	O	O	S	T

Algoritma: A*
 Runtime : 31.451 ms
 Node yang dikunjungi: 505
 Panjang rute:8

		<p>Algoritma: UCS Runtime : 180.409 ms Node yang dikunjungi: 7976 Panjang rute:8</p>
--	--	---

4.3. Analisis Kompleksitas Waktu dan Ruang Algoritma

Ketiga algoritma dipengaruhi oleh struktur kelas Word dan Dictionary yang sama.

1. Kelas Word

Word memiliki kompleksitas waktu konstan

2. Kelas Dictionary

Dictionary memanfaatkan struktur data HashSet untuk menyimpan kumpulan kata, sehingga kompleksitas waktu untuk mengecek apakah suatu kata ada di dictionary memiliki kompleksitas waktu $O(1)$ dan kompleksitas ruang $O(m)$ dimana m adalah jumlah kata unik dengan panjang yang sama pada dictionary.txt. Untuk membaca file txt sendiri butuh kompleksitas waktu $O(n)$, dengan n banyak kata di file.

Ketiga implementasi algoritma memiliki kompleksitas yang mirip. A* dan GBFS, melakukan kalkulasi untuk mendapatkan nilai heuristik dengan kompleksitas waktu $O(n)$ dengan n adalah panjang kata. Untuk menambahkan dan menghapus elemen Word pada queue, dibutuhkan kompleksitas waktu $O(\log n)$ dengan n banyak Word pada queue. Kemudian, dilakukan iterasi selama queue belum kosong atau solusi belum ditemukan. Pada kasus terburuk, iterasi dilakukan sebanyak elemen pada queue. Di setiap iterasi, ditentukan simpul tetangga. Pembentukan kata baru dilakukan sebanyak panjang kata dikali 26, juga dilakukan pengecekan kata pada Hashset visited dan dictionary dengan kompleksitas $O(1)$ serta kalkulasi heuristik dan cost dengan kompleksitas $O(n)$. Oleh karena itu, di setiap iterasi memiliki kompleksitas waktu $T(n) = n * k * 26$. Dalam notasi big O, kompleksitas waktu algoritma A* keseluruhan adalah $O(n * k * 26 * \log n)$ dengan k

adalah panjang kata, n adalah banyak elemen pada queue. Untuk kompleksitas ruang pada queue, visited set result list, adalah $O(m)$, dengan m tergantung pada jumlah elemen pada masing masing struktur data.

Pada algoritma UCS, sebenarnya kompleksitas waktu dan ruangnya tidak jauh berbeda dengan A^* . Namun, karena A^* mempertimbangkan nilai heuristik, solusi lebih cepat ditemukan.

4.4. Analisis Hasil Eksperimen

Dapat dilihat pada tabel, bahwa untuk setiap *test case*, didapatkan runtime algoritma Greedy Best First Search (GBFS) lebih kecil dibandingkan runtime algoritma A^* dan UCS. Hal ini dikarenakan GBFS fokus untuk memilih simpul yang optimal saat itu, sehingga lebih cepat. Jumlah simpul yang dikunjungi juga lebih sedikit dibandingkan UCS dan A^* . *Trade off* nya adalah hasil GBFS tidak selalu optimal karena simpul tersebut hanya optimal secara lokal dan belum tentu optimal hingga akhir. (Bahkan pada test case 2, GBFS menemukan rute dengan panjang 69, sedangkan UCS dan A^* dapat menemukan rute dengan panjang 15)

Sementara itu, algoritma A^* dan UCS dapat memberikan hasil yang optimal. Akan tetapi A^* lebih cepat karena mempertimbangkan 2 faktor, heuristik dan cost. Heuristik ini membuat jumlah simpul yang dieksplorasi lebih sedikit dan lebih menjanjikan karena simpul tersebut semakin mendekati simpul tujuan. Sementara itu, karena UCS hanya mempertimbangkan cost, (dan pada kasus Word Ladder cost kata-kata yang berasal dari satu induk yang sama akan memiliki cost yang sama pula), jumlah simpul yang dimasukkan ke queue semakin banyak. Hal ini dapat dilihat dari jumlah simpul yang dikunjungi lebih signifikan banyaknya dibandingkan kedua algoritma.

Berdasarkan analisis di atas, dalam penyelesaian permainan Word Ladder, algoritma A^* lebih efisien dibandingkan UCS. Walaupun sama-sama mendapatkan hasil yang optimal, runtime A^* tidak selama algoritma UCS. Dalam segi runtime tercepat, memang GBFS jauh lebih cepat, namun solusi belum tentu optimal.

BAB 5 PENUTUP

5.1. Kesimpulan

Dari pengerjaan tugas ini, penulis mendapatkan beberapa kesimpulan, yaitu

1. Algoritma GBFS belum tentu menemukan rute optimal/terpendek dari suatu persoalan, namun lebih cepat karena murni memprioritaskan rute dengan prediksi cost/heuristik terendah.
2. Algoritma UCS dapat menemukan rute optimal/terpendek dari suatu persoalan dengan memprioritaskan rute dengan total bobot terendah, namun lebih lama karena simpul-simpul pada kedalaman samal memiliki bobot yang sama
3. Algoritma A* dapat menemukan rute optimal/terpendek dari suatu persoalan dengan memprioritaskan rute dengan prediksi cost terendah serta cost sesungguhnya. Dalam kasus ini, A* memiliki efisiensi paling baik.

5.2. Saran

Implementasi pencarian rute yang dibuat masih dapat dikembangkan lebih lanjut. Dari segi performa, implementasi algoritma juga masih dapat ditingkatkan.

5.3. Komentar dan Refleksi

Penulis senang karena pelajaran yang diberikan dosen di kelas dapat diimplementasikan dengan baik pada tugas kali ini. Penulis berterima kasih kepada Dosen Pengampu mata kuliah Strategi Algoritma yang telah memberikan materi dengan jelas dan mendorong penulis untuk rajin mengeksplorasi dalam materi Stima ataupun GUI *development*.

5.4. Tabel Checkpoint

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

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Tucil3-2024.pdf>

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf>

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf>

LAMPIRAN

LINK REPOSITORY

Link repository GitHub

: https://github.com/kaylanamira/Tucil3_13522050