

LAPORAN TUGAS KECIL 3

IF2211 STRATEGI ALGORITMA

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



Disusun oleh:

Agil Fadillah Sabri (13522006)

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2024

DAFTAR ISI

DAFTAR ISI	i
DAFTAR GAMBAR	ii
DAFTAR TABEL	iii
BAB I DESKRIPSI MASALAH	1
BAB II LANGKAH PEMECAHAN MASALAH	2
2.1 Penyelesaian Permainan <i>Word Ladder</i> dengan Algoritma UCS	2
2.2 Penyelesaian Permainan <i>Word Ladder</i> dengan Algoritma <i>Greedy</i> BeFS	3
2.3 Penyelesaian Permainan <i>Word Ladder</i> dengan Algoritma A*	5
BAB III IMPLEMENTASI PROGRAM	7
3.1 Kelas Dictionary	7
3.2 Kelas Node	10
3.3 Kelas SearchAlgorithm	12
3.3.1 Kelas UCS	14
3.3.2 Kelas GreedyBeFS	16
3.3.3 Kelas AStar	18
3.4 Kelas WordLadderGUI	20
3.5 Kelas Main	23
BAB IV UJI COBA	24
4.1 Keterangan Proses Input dan Output	24
4.2 Pengujian Algoritma	25
BAB V ANALISIS SOLUSI	29
5.1 Algoritma UCS	30
5.2 Algoritma <i>Greedy</i> BeFS	31
5.3 Algoritma A*	32
BAB VI IMPLEMENTASI BONUS	33
6.1 Pembuatan GUI	33
DAFTAR PUSTAKA	34
LAMPIRAN	35

DAFTAR GAMBAR

Gambar 1. Ilustrasi dan Peraturan Permainan Word Ladder	1
Gambar 2. <i>Class Dictionary</i> (1)	7
Gambar 3. <i>Class Dictionary</i> (2)	8
Gambar 4. <i>Class Dictionary</i> (3)	8
Gambar 5. <i>Class Node</i> (1)	10
Gambar 6. <i>Class Node</i> (2)	11
Gambar 7. <i>Class SearchAlgorithm</i>	12
Gambar 8. <i>Class UCS</i>	14
Gambar 9. <i>Class GreedyBeFS</i>	16
Gambar 10. <i>Class AStar</i>	18
Gambar 11. <i>Class WordLadderGUI</i> (1)	20
Gambar 12. <i>Class WordLadderGUI</i> (2)	21
Gambar 13. <i>Class WordLadderGUI</i> (3)	22
Gambar 14. <i>Class Main</i>	23
Gambar 15. GUI Memasukkan <i>Input</i>	24
Gambar 16. Hasil Pencarian	25
Gambar 17. Hasil Pencarian TC 1 dengan Algoritma UCS	25
Gambar 18. Hasil Pencarian TC 1 dengan Algoritma GBeFS	25
Gambar 19. Hasil Pencarian TC 1 dengan Algoritma A*	25
Gambar 20. Hasil Pencarian TC 2 dengan Algoritma UCS.....	26
Gambar 21. Hasil Pencarian TC 2 dengan Algoritma GBeFS	26
Gambar 22. Hasil Pencarian TC 2 dengan Algoritma A*	26
Gambar 23. Hasil Pencarian TC 3 dengan Algoritma UCS	26
Gambar 24. Hasil Pencarian TC 3 dengan Algoritma GBeFS	26
Gambar 25. Hasil Pencarian TC 3 dengan Algoritma A*	26
Gambar 26. Hasil Pencarian TC 4 dengan Algoritma UCS	27
Gambar 27. Hasil Pencarian TC 4 dengan Algoritma GBeFS	27
Gambar 28. Hasil Pencarian TC 4 dengan Algoritma A*	27
Gambar 29. Hasil Pencarian TC 5 dengan Algoritma UCS	27
Gambar 30. Hasil Pencarian TC 5 dengan Algoritma GBeFS	27
Gambar 31. Hasil Pencarian TC 5 dengan Algoritma A*	27
Gambar 32. Hasil Pencarian TC 6 dengan Algoritma UCS	28
Gambar 33. Hasil Pencarian TC 6 dengan Algoritma GBeFS	28
Gambar 34. Hasil Pencarian TC 6 dengan Algoritma A*	28
Gambar 35. Hasil Pencarian TC 7 dengan Algoritma UCS	28
Gambar 36. Hasil Pencarian TC 7 dengan Algoritma GBeFS	28
Gambar 37. Hasil Pencarian TC 7 dengan Algoritma A*	28
Gambar 38. Perbandingan Hasil Uji Ketiga Algoritma dengan Parameter Waktu Eksekusi	29
Gambar 39. Perbandingan Hasil Uji Ketiga Algoritma dengan Parameter Total Node yang Dikunjungi	29
Gambar 40. Perbandingan Hasil Uji Ketiga Algoritma dengan Parameter Panjang Total <i>Path</i>	30
Gambar 41. GUI Program <i>Word Ladder</i>	33

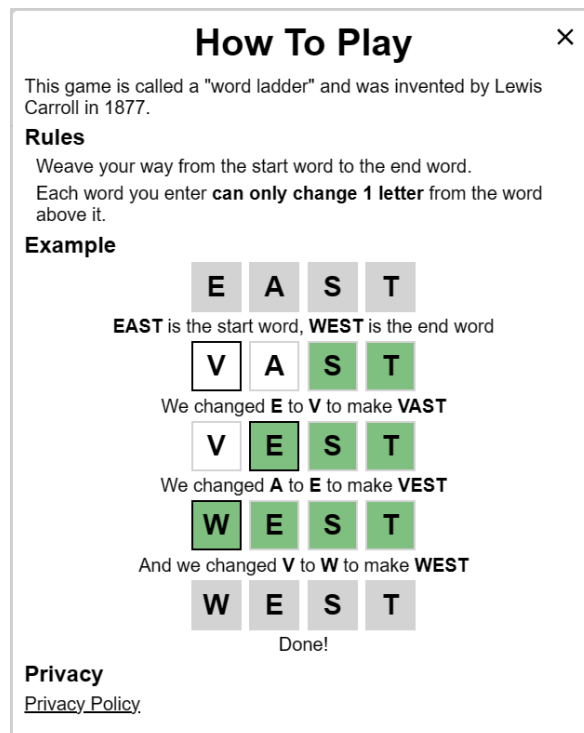
DAFTAR TABEL

Tabel 1. <i>Class Dictionary</i>	9
Tabel 2. <i>Class Node</i>	10
Tabel 3. <i>Class SearchAlgorithm</i>	13
Tabel 4. <i>Class UCS</i>	15
Tabel 5. <i>Class GreedyBeFS</i>	17
Tabel 6. <i>Class AStar</i>	19
Tabel 7. <i>Class WordLadderGUI</i>	22
Tabel 8. <i>Class Main</i>	23
Tabel 9. <i>Test Case 1</i>	25
Tabel 10. <i>Test Case 2</i>	26
Tabel 11. <i>Test Case 3</i>	26
Tabel 12. <i>Test Case 4</i>	27
Tabel 13. <i>Test Case 5</i>	27
Tabel 14. <i>Test Case 6</i>	28
Tabel 15. <i>Test Case 7</i>	28

BAB I

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/>

BAB II

LANGKAH PEMECAHAN MASALAH

2.1 Penyelesaian Permainan *Word Ladder* dengan Algoritma UCS

Uniform Cost Search (UCS) merupakan salah satu algoritma yang dapat digunakan dalam menyelesaikan persoalan optimasi. Dalam algoritma UCS, setiap simpul memiliki biaya yang terkait dengannya, dan algoritma mencoba untuk menemukan jalur dengan biaya total terendah dari simpul awal ke simpul tujuan. Ini dilakukan dengan mengeksplorasi simpul-simpul secara berurutan berdasarkan biaya terendahnya.

Untuk dapat menghitung biaya setiap simpul, digunakan fungsi evaluasi (*evaluation function*) $f(n)$, dengan n menyatakan simpul saat ini/simpul yang akan dievaluasi. Fungsi evaluasi $f(n)$ merupakan suatu fungsi yang digunakan untuk memperkirakan biaya (*cost*) termurah dari simpul n ke simpul solusi (yang mungkin tidak diketahui letaknya). Pada dasarnya $f(n)$ menyatakan batas bawah (*lower bound*) dari biaya (*cost*) pencarian solusi dari simpul n . Pada algoritma UCS, nilai $f(n)$ hanya akan bergantung pada parameter biaya (*cost*) yang telah dikeluarkan dari simpul awal menuju simpul n yang dilambangkan dengan $g(n)$.

Dalam Algoritma UCS, terdapat dua istilah penting untuk menyatakan keadaan suatu simpul. Simpul yang saat ini sedang dievaluasi disebut dengan simpul *expand*. Ketika suatu simpul di-*expand*, pertama-tama akan dicek apakah ia merupakan simpul tujuan atau bukan. Jika simpul *expand* merupakan simpul tujuan, maka pencarian selesai dan jalur dari simpul awal ke simpul tersebut dicatat sebagai solusi beserta biayanya (*cost*). Jika simpul *expand* bukan merupakan simpul tujuan, maka dicari setiap tetangga dari simpul *expand* yang belum pernah dievaluasi sebelumnya dan dimasukkan ke dalam suatu antrian dengan prioritas (*priority queue*). Adapun istilah kedua adalah simpul hidup (*lifenode*). Simpul hidup merupakan setiap simpul yang saat ini berada di dalam *priority queue* yang menunggu untuk di-*expand*.

Algoritma UCS dapat digunakan untuk menyelesaikan permainan *Word Ladder*, karena permainan *Word Ladder* merupakan salah satu bentuk persoalan optimasi, yang menginginkan solusi rantai kata yang seminimal mungkin. Adapun langkah-langkah penyelesaian permainan *Word Ladder* dengan algoritma UCS yaitu:

1. Nyatakan kata awal (*start word*) sebagai simpul pertama tempat dimulainya algoritma UCS. Jadikan sebagai simpul *expand*.
2. Untuk setiap simpul *expand*, cek apakah simpul tersebut merupakan simpul tujuan atau bukan. Jika iya, maka pencarian berhenti dan catat jalur dari kata awal menuju kata tersebut.
3. Jika ternyata simpul *expand* bukan simpul tujuan, cari setiap kata yang ada di dalam kamus yang hanya berbeda satu huruf saja dari kata pada simpul *expand* saat ini. Untuk setiap kata yang diperoleh, hitung $f(n)$ dengan menggunakan $g(n)$ dengan formula sebagai berikut:

$g(n)$ = Banyaknya perubahan yang telah dilakukan dari kata awal ke kata yang dievaluasi saat ini.

Contoh perhitungan: EAST \rightarrow VAST \rightarrow VEST \rightarrow WEST

Kata awal: EAST

Kata tujuan: WEST

EAST $g(n) = 0$

VEST $g(n) = 2$

VAST $g(n) = 1$

WEST $g(n) = 3$

$f(n) = g(n)$

4. Masukkan setiap kata-kata yang diperoleh pada langkah 3 ke dalam *priority queue* dengan nilai prioritas dihitung berdasarkan $f(n)$ dengan $f(n)$ terkecil memiliki nilai prioritas tertinggi.
5. Untuk mengoptimalkan proses pencarian, suatu kata yang sudah pernah menjadi simpul *expand* tidak perlu lagi dimasukkan ke dalam *priority queue* karena pasti akan menghasilkan rantai kata yang lebih panjang.
6. Keluarkan kata dengan prioritas tertinggi dari *priority queue* dan jadikan sebagai simpul *expand* selanjutnya.
7. Ulangi langkah 2-6 hingga kata tujuan tercapai atau hingga tidak ada lagi kata yang berada di dalam *priority queue* kosong. Jika pencarian berhenti karena *priority queue*, maka disimpulkan bahwa kata awal tidak bisa mencapai kata tujuan.

Jika diperhatikan secara seksama, penggunaan nilai $g(n)$ pada formula di atas menyebabkan setiap simpul tetangga dari suatu simpul hanya berbeda satu *cost* saja. Jika misal terdapat simpul A dengan *cost* c , setiap simpul tetangga dari A akan memiliki *cost* $c+1$, dan simpul tetangga dari setiap simpul tetangga dari A akan memiliki *cost* $c+1+1$. Akibatnya, nilai *cost* ini akan sama dengan kedalaman suatu simpul dari simpul awal. Karena simpul dengan *cost* lebih rendah (yang berarti kedalaman lebih kecil) selalu diperiksa terlebih dahulu dari simpul dengan *cost* yang lebih tinggi (yang berarti kedalaman lebih besar), akibatnya urutan pemeriksaan kata-kata pada permainan *Word Ladder* dengan algoritma UCS akan sama persis dengan urutan pemeriksaan kata-kata dengan algoritma BFS (*Breadth First Search*) dan *path* atau rantai kata yang dihasilkan juga akan sama.

2.2 Penyelesaian Permainan *Word Ladder* dengan Algoritma Greedy BeFS

Best First Search (BeFS) merupakan salah satu algoritma yang dapat digunakan dalam menyelesaikan persoalan optimasi. Sama seperti UCS, pada algoritma BeFS juga digunakan suatu nilai evaluasi ($f(n)$) yang akan menentukan urutan pemeriksaan setiap simpul-simpulnya. Salah satu bentuk algoritma BeFS adalah Greedy BeFS yang memasukkan konsep algoritma Greedy ke dalam proses penentuan nilai evaluasi setiap simpul.

Jika UCS menentukan nilai $f(n)$ berdasarkan biaya yang telah dihabiskan dari simpul tujuan menuju simpul n , maka pada algoritma greedy BeFS ini, nilai $f(n)$ akan ditentukan berdasarkan suatu perkiraan biaya (*cost*) yang bersifat *heuristik* yang diperlukan untuk mencapai simpul tujuan dari simpul n . Fungsi *heuristik* yang digunakan ini dilambangkan dengan $h(n)$.

Akibat sifat dari *greedy* BeFS yang hanya memperhitungkan perkiraan yang bersifat *heuristik*, maka pencarian dengan algoritma ini tidak menjamin ditemukannya solusi yang optimal, karena fungsi *heuristik* yang dihasilkan hanya melihat keadaan secara lokal, tetapi tidak memperhatikan keadaan secara global. Akibatnya, bisa terdapat simpul yang mungkin secara lokal menghasilkan *heuristik* yang minimal, namun secara global justru menghasilkan solusi yang tidak optimal karena simpul tersebut justru menyebabkan proses pencarian bergerak menuju simpul yang jauh lebih banyak. Akibatnya, proses pemilihan fungsi *heuristik* yang digunakan harus benar-benar dipilih secara teliti agar solusi yang dihasilkan dapat mendekati ke solusi optimal.

Sama seperti algoritma UCS, algoritma *greedy* BeFS juga dapat digunakan untuk menyelesaikan permainan *Word Ladder*. Adapun langkah-langkah penyelesaian permainan *Word Ladder* dengan algoritma *greedy* BeFS yaitu:

1. Nyatakan kata awal (*start word*) sebagai simpul pertama tempat dimulainya algoritma *greedy* BeFS. Jadikan sebagai simpul *expand*.
2. Untuk setiap simpul *expand*, cek apakah simpul tersebut merupakan simpul tujuan atau bukan. Jika iya, maka pencarian berhenti dan catat jalur dari kata awal menuju kata tersebut.
3. Jika ternyata simpul *expand* bukan simpul tujuan, cari setiap kata yang ada di dalam kamus yang hanya berbeda satu huruf saja dari kata pada simpul *expand* saat ini. Untuk setiap kata yang diperoleh, hitung $f(n)$ dengan menggunakan $h(n)$ dengan formula sebagai berikut:

$h(n)$ = Banyaknya huruf yang berbeda dari kata saat ini dengan kata tujuan.

Contoh perhitungan: EAST \rightarrow VAST \rightarrow VEST \rightarrow WEST

Kata awal: EAST Kata tujuan: WEST

EAST $h(n) = 2$ VEST $h(n) = 1$

VAST $h(n) = 2$ WEST $h(n) = 0$

$f(n) = h(n)$

4. Masukkan setiap kata-kata yang diperoleh pada langkah 3 ke dalam *priority queue* dengan nilai prioritas dihitung berdasarkan $f(n)$ dengan $f(n)$ terkecil memiliki nilai prioritas tertinggi.
5. Untuk mengoptimalkan proses pencarian, suatu kata yang sudah pernah menjadi simpul *expand* tidak perlu lagi dimasukkan ke dalam *priority queue* karena pasti akan menghasilkan rantai kata yang lebih panjang.
6. Keluarkan kata dengan prioritas tertinggi dari *priority queue* dan jadikan sebagai simpul *expand* selanjutnya.
7. Ulangi langkah 2-6 hingga kata tujuan tercapai atau hingga tidak ada lagi kata yang berada di dalam *priority queue*. Jika pencarian berhenti karena *priority queue* kosong, maka disimpulkan bahwa kata awal tidak bisa mencapai kata tujuan.

Seperti penjelasan sebelumnya, algoritma *greedy* BeFS ini tidak menjamin ditemukannya solusi yang optimal pada permainan *Word Ladder* karena belum tentu perubahan minimal yang diperlukan sama dengan nilai fungsi *heuristik*-nya karena tidak semua susunan huruf yang terbentuk ada di dalam kamus.

2.3 Penyelesaian Permainan *Word Ladder* dengan Algoritma A*

A* merupakan salah satu algoritma yang digunakan dalam traversal graf dan pencarian rute yang dapat digunakan dalam menyelesaikan persoalan optimasi. Sama seperti UCS dan greedy BeFS, pada algoritma A* juga digunakan nilai evaluasi ($f(n)$) yang akan menentukan urutan pemeriksaan setiap simpul-simpulnya. Pada dasarnya, algoritma A* merupakan bentuk pengoptimalan kedua algoritma sebelumnya, yang mengambil kelebihan masing-masing dari setiap algoritma. Kelebihan algoritma UCS akan menjamin solusi yang optimal pada algoritma A* dan kelebihan algoritma *greedy* BeFS akan memberikan waktu pencarian algoritma A* yang lebih cepat dari algoritma UCS.

Pada algoritma A*, penentuan nilai $f(n)$ setiap simpul akan menggabungkan $f(n)$ dari algoritma UCS dengan $f(n)$ dari algoritma *greedy* BeFS. Jadi pada A*, selain memperhitungkan biaya yang sudah dihabiskan dari simpul awal menuju simpul n ($g(n)$), perkiraan *heuristik* yang memperkirakan biaya minimal dari simpul n ke simpul tujuan juga akan diperhitungkan ($h(n)$). Jadi pada algoritma A*, $f(n) = g(n) + h(n)$.

Algoritma A* dapat digunakan untuk menyelesaikan permainan *Word Ladder*. Adapun langkah-langkah penyelesaian permainan *Word Ladder* dengan algoritma A* yaitu:

1. Nyatakan kata awal (*start word*) sebagai simpul pertama tempat dimulainya algoritma *greedy* BeFS. Jadikan sebagai simpul *expand*.
2. Untuk setiap simpul *expand*, cek apakah simpul tersebut merupakan simpul tujuan atau bukan. Jika iya, maka pencarian berhenti dan catat jalur dari kata awal menuju kata tersebut.
3. Jika ternyata simpul *expand* bukan simpul tujuan, cari setiap kata yang ada di dalam kamus yang hanya berbeda satu huruf saja dari kata pada simpul *expand* saat ini. Untuk setiap kata yang diperoleh, hitung $f(n)$ dengan menggunakan $h(n)$ dengan formula sebagai berikut:
 $g(n)$ = Banyaknya perubahan yang telah dilakukan dari kata awal ke kata yang dievaluasi saat ini (sama seperti pada UCS).
 $h(n)$ = Banyaknya huruf yang berbeda dari kata saat ini dengan kata tujuan (sama seperti pada *greedy* BeFS).
 $f(n) = g(n) + h(n)$
4. Masukkan setiap kata-kata yang diperoleh pada langkah 3 ke dalam *priority queue* dengan nilai prioritas dihitung berdasarkan $f(n)$ dengan $f(n)$ terkecil memiliki nilai prioritas tertinggi.
5. Untuk mengoptimalkan proses pencarian, suatu kata yang sudah pernah menjadi simpul *expand* tidak perlu lagi dimasukkan ke dalam *priority queue* karena pasti akan menghasilkan rantai kata yang lebih panjang.
6. Keluarkan kata dengan prioritas tertinggi dari *priority queue* dan jadikan sebagai *simpul expand* selanjutnya.
7. Ulangi langkah 2-6 hingga kata tujuan tercapai atau hingga tidak ada lagi kata yang berada di dalam *priority queue*. Jika pencarian berhenti karena *priority queue* kosong, maka disimpulkan bahwa kata awal tidak bisa mencapai kata tujuan.

Fungsi *heuristic* yang digunakan pada algoritma A* haruslah bersifat *admissible* (dapat diterima). Suatu fungsi *heuristic* dikatakan bersifat *admissible* jika perkiraan biaya yang dihasilkannya selalu bersifat *underestimate* (menghasilkan perkiraan nilai yang lebih kecil dari biaya sesungguhnya) untuk persoalan optimasi minimum, dan *overestimate* (menghasilkan perkiraan nilai yang lebih besar dari biaya sesungguhnya) untuk persoalan optimasi maksimum. Jika fungsi *heuristic* menghasilkan nilai yang sebaliknya, maka *heuristic* tersebut dikatakan *not admissible* (tidak dapat diterima).

Pada permainan *Word Ladder*, fungsi *heuristic* yang digunakan dihitung berdasarkan jumlah huruf yang berbeda dari kata tujuan. Fungsi *heuristic* ini pada dasarnya merepresentasikan banyaknya perubahan minimal yang harus dilakukan agar suatu kata dapat menjadi kata tujuan. Jika suatu kata berbeda 3 huruf dari kata tujuan, maka dapat dipastikan bahwa seminimal-minimalnya terjadi 3 kali perubahan agar kata tersebut mencapai kata tujuan, namun perubahan sesungguhnya mungkin saja lebih dari 3 kali jika seandainya kata yang terbentuk tidak ada di dalam kamus sehingga harus menggunakan kata yang lain terlebih dahulu. Akan tetapi, dapat dipastikan bahwa perubahan tidak mungkin dapat dilakukan hanya kurang dari 3 kali, karena setiap kali perubahan hanya memperbolehkan satu huruf saja yang berubah. Dari ilustrasi tersebut, dapat dikatakan bahwa fungsi *heuristic* yang digunakan pada permainan *Word Ladder* ini bersifat *underestimate*. Karena permainan *Word Ladder* merupakan persoalan optimasi minimum, maka fungsi *heuristic* yang digunakan bersifat *admissible*.

Algoritma A* menggunakan pendekatan *heuristic* untuk memandu pencarian, yang dapat membantu mengarahkan pencarian ke arah yang paling menjanjikan untuk mencapai solusi. Fungsi *heuristic* yang digunakan, memungkinkan A* untuk memiliki pemahaman lebih baik tentang "jarak" simpul saat ini menuju simpul tujuan, yang memungkinkan algoritma A* untuk mengurangi jumlah langkah yang diperlukan untuk menemukan solusi optimal. Di sisi lain, UCS tidak menggunakan *heuristic* dan hanya mempertimbangkan biaya aktual dari setiap langkah. Ini berarti UCS mungkin akan mengeksplorasi jalan-jalan yang tidak produktif atau berputar-putar lebih banyak sebelum menemukan solusi optimal. Oleh karena itu, pada kasus *Word Ladder* di mana *heuristic* dapat memberikan panduan yang berguna, A* biasanya lebih efisien daripada UCS dalam menemukan solusi, baik dari segi waktu maupun penggunaan memori.

BAB III

IMPLEMENTASI PROGRAM

Program penyelesaian permainan *Word Ladder* ini diimplementasikan menggunakan bahasa pemrograman python (.py). Adapun untuk pembuatan GUI memanfaatkan *library* javax.swing dan java.awt.

3.1 Kelas Dictionary

Class Dictionary

Kelas ini digunakan untuk menyimpan *dictionary*/kamus yang digunakan pada permainan *Word Ladder*.

```
/**
 * Class Dictionary
 * This class is used to store a dictionary of words and some methods to manipulate the dictionary
 */
public class Dictionary {
    /**
     * Attributes
     */
    private String[] dictionary;
    private int length;

    /**
     * Methods
     */
    /**...
    public Dictionary(String _namaFile) {
        String namaFile = "src/dictionary/" + _namaFile;
        try {
            Scanner scanner = new Scanner(new File(namaFile));
            // Counting the number of lines in the file
            int count = 0;
            while (scanner.hasNextLine()) {
                scanner.nextLine();
                count++;
            }
            this.length = count;
            scanner.close();

            // Reading the file and storing the words in dictionary array
            this.dictionary = new String[this.length];
            scanner = new Scanner(new File(namaFile));
            for (int i = 0; i < count; i++) {
                dictionary[i] = scanner.nextLine();
            }

            scanner.close();
        } catch (FileNotFoundException e) {
            System.out.println("File " + namaFile + " tidak ditemukan.");
        }
    }
}
```

Gambar 2. *Class Dictionary* (1)

```

/** ...
public boolean isWordInDictionary(String word) {
    for (int i = 0; i < length; i++) {
        if (word.equalsIgnoreCase(dictionary[i])) {
            return true;
        }
    }
    return false;
}

/** ...
public int wordDistance(String word1, String word2) {
    if (word1.length() != word2.length()) {
        return -1;
    }
    else {
        int count = 0;
        for (int i = 0; i < word1.length(); i++) {
            if (Character.toLowerCase(word1.charAt(i)) != Character.toLowerCase(word2.charAt(i))) {
                count++;
            }
        }
        return count;
    }
}

/** ...
public String[] getAllWordsWithNLength(int n) {
    String[] result = new String[length];

    // Counting the number of words that are length n
    int count = 0;
    for (int i = 0; i < length; i++) {
        if (dictionary[i].length() == n) {
            result[count] = dictionary[i];
            count++;
        }
    }

    // Trimming the result array
    String[] resultTrimmed = new String[count];
    for (int i = 0; i < count; i++) {
        resultTrimmed[i] = result[i];
    }
    return resultTrimmed;
}

```

Gambar 3. Class Dictionary (2)

```

/** ...
public String[] getAllOneCharDifferenceStrings(String word, String[] listOfWords) {
    String[] result = new String[listOfWords.length];

    // Counting the number of words that are different by one letter with word
    int count = 0;
    for (int i = 0; i < listOfWords.length; i++) {
        if (wordDistance(word, listOfWords[i]) == 1) {
            result[count] = listOfWords[i];
            count++;
        }
    }

    // Trimming the result array
    String[] resultTrimmed = new String[count];
    for (int i = 0; i < count; i++) {
        resultTrimmed[i] = result[i];
    }
    return resultTrimmed;
}

```

Gambar 4. Class Dictionary (3)

Atribut	
- String[] dictionary	Menyimpan daftar kata-kata yang terdapat pada kamus.
- int length	Menyimpan banyaknya kata yang ada pada kamus.
Method	
+ Dictionary(String _namaFile)	Konstruktor pembentuk objek Dictionary dengan parameter sebuah string sebagai nama file tempat kamus disimpan.
+ boolean isWordInDictionary (String word)	Mengembalikan true jika sebuah word terdapat di dalam kamus, false jika tidak.
+ int wordDistance (String word1, String word2)	Mengembalikan jumlah karakter yang berbeda antara word1 dengan word2.
+ String[] getAllWordsWithNLength (int n)	Mengembalikan kumpulan kata (String) yang memiliki banyak karakter sebanyak n.

Tabel 1. *Class Dictionary*

3.2 Kelas Node

Class Node

Kelas ini digunakan untuk merepresentasikan setiap kata sebagai simpul pada graf pencarian. Kelas ini mengimplementasikan interface Comparable yang menjadi dasar perbandingan untuk setiap node ketika hendak ditambahkan ke dalam *priority queue*.

```
/**
 * Class Node
 * This class is used to store a node in the priority queue
 */
public class Node implements Comparable<Node>{
    /**
     * Attributes
     */
    private String word;           // word = n
    private int cost;              // cost = f(n)
    private ArrayList<String> path; // path = [start, ..., parent of n]

    /**
     * Methods
     */

    /** ...
    public Node(String word, int cost) {
        this.word = word;
        this.cost = cost;
        path = new ArrayList<String>();
    }

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

    /** ...
    public int getCost() {
        return this.cost;
    }

    /** ...
    public ArrayList<String> getPath() {
        return this.path;
    }

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

    /** ...
    public void setPath(ArrayList<String> path) {
        this.path = path;
    }
}
```

Gambar 5. Class Node (1)

```

    /** ...
    public void addPath(String word) {
        this.path.add(word);
    }

    /** ...
    @Override
    public int compareTo(Node other) {
        if (Integer.compare(this.cost, other.cost) != 0) {
            return Integer.compare(this.cost, other.cost);
        } else {
            return this.word.compareTo(other.word);
        }
    }
}

```

Gambar 6. Class Node (2)

Atribut	
- String word	Kata saat ini.
- int cost	Nilai evaluasi kata saat ini ($f(n)$).
- ArrayList<String> path;	Jalur dari kata awal menuju kata saat ini.
Method	
+ Node(String word, int cost)	Konstruktor pembentuk objek Node dengan parameter sebuah string sebagai kata saat ini dan cost sebagai nilai $f(n)$.
+ String getWord()	Mengembalikan nilai atribut word.
+ int getCost()	Mengembalikan nilai atribut cost.
+ ArrayList<String> getPath()	Mengembalikan isi atribut path.
+ void setCost(int cost)	Mengubah nilai atribut cost sesuai dengan parameter cost.
+ void setPath (ArrayList<String> path)	Mengubah nilai atribut path sesuai dengan parameter path.
+ void addPath(String word)	Menambahkan word ke dalam atribut path.
+ int compareTo(Node other)	Implementasi dari interface Comparable. Mengembalikan 1 jika objek Node saat ini memiliki cost lebih besar dari Node other, -1 objek Node saat ini memiliki cost lebih kecil dari Node other, dan mengembalikan hasil <code>this.word.compareTo(other.word)</code> jika objek Node saat ini memiliki cost sama dengan Node other.

Tabel 2. Class Node

3.3 Kelas SearchAlgorithm

<i>Class SearchAlgorithm</i>
Merupakan kelas abstrak yang menjadi <i>parent class</i> untuk kelas-kelas algoritma pencarian. Memiliki method abstrak <code>search()</code> , <code>gn()</code> , dan <code>hn()</code> .

```

/**
 * Abstract Class SearchAlgorithm
 * This class is used to implement search algorithms
 */
public abstract class SearchAlgorithm {
    /**
     * Attributes
     */
    protected String start;           // Start word
    protected String goal;           // Goal word
    protected Dictionary dictionary; // Dictionary of words
    protected String[] wordWithSameLengthWithQuery; // Array of words with the same length as the query
    protected Integer VisitedNodes = 0; // Number of visited nodes
    protected Long executionTime; // Execution time
    protected Node result; // Result of the search
    protected PriorityQueue<Node> lifeNode; // Priority queue of nodes
    protected Map<String, Boolean> explored; // Map of explored nodes, key = word, value = true (if the word has been expanded)

    /**
     * Methods
     */

    /**...
    public SearchAlgorithm(String start, String goal, Dictionary dictionary) {
        this.start = start;
        this.goal = goal;
        this.dictionary = dictionary;
        this.lifeNode = new PriorityQueue<Node>();
        this.explored = new HashMap<String, Boolean>();
        this.result = null;
        this.wordWithSameLengthWithQuery = dictionary.getAllWordsWithNLength(start.length());
    }

    /**...
    public Node getResult() {
        return this.result;
    }

    /**...
    public Integer getVisitedNodes() {
        return this.VisitedNodes;
    }

    /**...
    public Long getExecutionTime() {
        return this.executionTime;
    }

    /**...
    public abstract void search();

    /**...
    protected abstract Integer gn(Node parent);
    protected abstract Integer hn(String current);
}

```

Gambar 7. *Class SearchAlgorithm*

Atribut	
# String start	Kata awal.
# String goal	Kata tujuan.
# Dictionary dictionary	Kamus yang digunakan.
# String[] wordWithSameLengthWithQuery	Daftar huruf yang terdapat di dalam dictionary dan memiliki panjang yang sama dengan kata awal dan kata tujuan. Berguna untuk optimalisasi pencarian agar tidak dilakukan traversal berulang-ulang pada dictionary.
# Integer VisitedNodes	Jumlah node yang dikunjungi selama proses pencarian solusi.
# Long executionTime	Waktu eksekusi yang dihabiskan selama proses pencarian solusi.
# Node result	Menyimpan Node solusi.

# PriorityQueue<Node> lifeNode	<i>Priority queue</i> untuk menyimpan urutan node-node yang akan di- <i>expand</i> .
# Map<String,Boolean> explored	Melakukan pelacakan Node-Node yang telah pernah di- <i>expand</i> agar tidak di- <i>expand</i> berkali-kali.
Method	
+ SearchAlgorithm(String start, String goal, Dictionary dictionary)	Konstruktor pembentuk objek SearchAlgorithm dengan parameter sebuah String kata awal, String kata tujuan, dan Dictionary yang digunakan.
+ Node getResult()	Mengembalikan atribut result.
+ Integer getVisitedNodes()	Mengembalikan atribut VisitedNodes.
+ Long getExecutionTime()	Mengembalikan atribut executionTimes.
+ abstract void search()	Method abstrak untuk melakukan proses pencarian solusi.
# abstract Integer gn(Node Parent)	Method abstrak untuk menghitung nilai $g(n)$ suatu Node.
# abstract Integer hn(String current)	Method abstrak untuk menghitung nilai $h(n)$ suatu Node.

Tabel 3. Class SearchAlgorithm

3.3.1 Kelas UCS

Class UCS

Kelas ini merupakan *child class* dari *class* SearchAlgorithm yang digunakan untuk menjalankan proses pencarian solusi permainan *Word Ladder* menggunakan algoritma *Uniform Cost Search* (UCS).

```
/**
 * Class UCS
 * This class is used to implement Uniform Cost Search algorithm
 * UCS is a search algorithm that finds the shortest path between the initial node and the goal node
 * It is an uninformed search algorithm that uses a priority queue to expand the node with the lowest cost
 *  $f(n) = g(n)$ 
 *  $g(n)$  = Cost of the path from the start node to the current node
 */
public class UCS extends SearchAlgorithm {
    /**
     * Method
     */
    /**...
    public UCS(String start, String goal, Dictionary dictionary) {
        super(start, goal, dictionary);
    }
    /**...
    protected Integer gn(Node parent) {
        //  $g(n)$  = Many changes have been made from the start word
        return parent.getPath().size() + 1;
    }
    protected Integer hn(String current) {
        //  $h(n)$  = Heuristic function is calculated based on the number of different letters of the target word
        // In Uniform Cost Search,  $h(n)$  is not used
        return 0;
    }
    /**...
    public void search() {
        Long startTime = System.nanoTime();

        // Create a start node and add it to the priority queue
        Node startNode = new Node(this.start, cost:0);
        this.lifeNode.add(startNode);
        this.explored.put(startNode.getWord(), value:true);

        // Loop until the priority queue is empty or the goal node is found
        while (!this.lifeNode.isEmpty()) {
            // Expand the node with the lowest cost
            Node expandNode = this.lifeNode.poll();
            this.explored.put(expandNode.getWord(), value:true);
            this.VisitedNodes++;

            // Check if the goal node is found
            if (expandNode.getWord().equalsIgnoreCase(this.goal)) {
                this.result = expandNode;
                break;
            }

            // Add the neighbors of the expanded node to the priority queue
            for (String neighbor : this.dictionary.getAllOneCharDifferenceStrings(expandNode.getWord(), this.wordWithSameLengthWithQuery)) {
                if (!this.explored.containsKey(neighbor)) { // Check if the neighbor has been explored
                    //  $f(n) = g(n)$ 
                    Integer fn = gn(expandNode);

                    // Create a new node and add it to the priority queue
                    Node newNode = new Node(neighbor, fn);
                    newNode.setPath(new ArrayList<String>(expandNode.getPath()));
                    newNode.addPath(expandNode.getWord());
                    this.lifeNode.add(newNode);
                }
            }
        }
        Long endTime = System.nanoTime();
        this.executionTime = (endTime - startTime) / 1000000;
    }
}
```

Gambar 8. Class UCS

Atribut	
Method	
+ UCS(String start, String goal, Dictionary dictionary)	Konstruktor pembentuk objek UCS dengan parameter sebuah String kata awal, String kata tujuan, dan Dictionary yang digunakan.
# Integer gn(Node Parent)	Method untuk menghitung nilai $g(n)$ suatu Node.
# Integer hn(String current)	Method untuk menghitung nilai $h(n)$ suatu Node.
+ void search()	Method abstrak untuk melakukan proses pencarian solusi.

Tabel 4. *Class UCS*

3.3.2 Kelas GreedyBeFS

Class GreedyBeFS

Kelas ini merupakan *child class* dari *class* SearchAlgorithm yang digunakan untuk menjalankan proses pencarian solusi permainan *Word Ladder* menggunakan algoritma *Greedy Best First Search* (*Greedy BeFS*).

```
/**
 * Class GreedyBeFS
 * This class is used to implement Greedy Best First Search algorithm
 * GreedyBeFS is a search algorithm that finds the shortest path between the initial node and the goal node
 * It is an informed search algorithm that uses a priority queue to expand the node with the lowest heuristic value
 * f(n) = h(n)
 */
public class GreedyBeFS extends SearchAlgorithm {
    /**
     * Method
     */

    /**...
    public GreedyBeFS(String start, String goal, Dictionary dictionary) { ...

    /**...
    protected Integer gn(Node parent) {
        // g(n) = Many changes have been made from the start word
        // In Greedy Best First Search, g(n) is not used
        return 0;
    }

    protected Integer hn(String current) {
        // h(n) = Heuristic function is calculated based on the number of different letters of the target word
        return this.dictionary.wordDistance(current, this.goal);
    }

    /**...
    public void search() {
        Long startTime = System.nanoTime();

        // Create a start node and add it to the priority queue
        Node startNode = new Node(this.start, this.dictionary.wordDistance(this.start, this.goal));
        this.lifeNode.add(startNode);
        this.explored.put(startNode.getWord(), value:true);

        // Loop until the priority queue is empty or the goal node is found
        while (!this.lifeNode.isEmpty()) {
            // Expand the node with the Lowest heuristic value
            Node expandNode = this.lifeNode.poll();
            this.explored.put(expandNode.getWord(), value:true);
            this.VisitedNodes++;

            // Check if the goal node is found
            if (expandNode.getWord().equalsIgnoreCase(this.goal)) {
                this.result = expandNode;
                break;
            }

            // Add the neighbors of the expanded node to the priority queue
            for (String neighbor : this.dictionary.getAllOneCharDifferenceStrings(expandNode.getWord(), this.wordWithSameLengthWithQuery)) {
                if (!this.explored.containsKey(neighbor)) { // Check if the neighbor has been explored

                    // f(n) = h(n)
                    Integer fn = hn(neighbor);

                    // Create a new node and add it to the priority queue
                    Node newNode = new Node(neighbor, fn);
                    newNode.setPath(new ArrayList<String>(expandNode.getPath()));
                    newNode.addPath(expandNode.getWord());
                    this.lifeNode.add(newNode);
                }
            }
        }
        Long endTime = System.nanoTime();
        this.executionTime = (endTime - startTime) / 1000000;
    }
}
```

Gambar 9. Class GreedyBeFS

Atribut	
Method	
+ GreedyBeFS(String start, String goal, Dictionary dictionary)	Konstruktor pembentuk objek GreedyBeFS dengan parameter sebuah String kata awal, String kata tujuan, dan Dictionary yang digunakan.
# Integer gn(Node Parent)	Method untuk menghitung nilai $g(n)$ suatu Node.
# Integer hn(String current)	Method untuk menghitung nilai $h(n)$ suatu Node.
+ void search()	Method abstrak untuk melakukan proses pencarian solusi.

Tabel 5. *Class GreedyBeFS*

3.3.3 Kelas Astar

Class AStar

Kelas ini merupakan *child class* dari *class SearchAlgorithm* yang digunakan untuk menjalankan proses pencarian solusi permainan *Word Ladder* menggunakan algoritma A*.

```
/**
 * Class AStar
 * This class is used to implement A* algorithm
 * A* is a search algorithm that finds the shortest path between the initial node and the goal node
 * It is an informed search algorithm that uses a heuristic function to estimate the cost of the cheapest path
 *  $f(n) = g(n) + h(n)$ 
 *  $g(n)$  = Many changes have been made from the start word
 *  $h(n)$  = Heuristic function is calculated based on the number of different letters of the target word
 */
public class AStar extends SearchAlgorithm {
    /**
     * Method
     */
    /**...
    public AStar(String start, String goal, Dictionary dictionary) {
        super(start, goal, dictionary);
    }
    /**...
    protected Integer gn(Node parent) {
        //  $g(n)$  = Many changes have been made from the start word
        return parent.getPath().size() + 1;
    }
    protected Integer hn(String current) {
        //  $h(n)$  = Heuristic function is calculated based on the number of different letters of the target word
        return this.dictionary.wordDistance(current, this.goal);
    }
    /**...
    public void search() {
        Long startTime = System.nanoTime();

        // Create a start node and add it to the priority queue
        Node startNode = new Node(this.start, this.dictionary.wordDistance(this.start, this.goal));
        this.lifeNode.add(startNode);
        this.explored.put(startNode.getWord(), value:true);

        // Loop until the priority queue is empty or the goal node is found
        while (!this.lifeNode.isEmpty()) {
            // Expand the node with the Lowest cost
            Node expandNode = this.lifeNode.poll();
            this.explored.put(expandNode.getWord(), value:true);
            this.VisitedNodes++;

            // Check if the goal node is found
            if (expandNode.getWord().equalsIgnoreCase(this.goal)) {
                this.result = expandNode;
                break;
            }

            // Add the neighbors of the expanded node to the priority queue
            for (String neighbor : this.dictionary.getAllOneCharDifferenceStrings(expandNode.getWord(), this.wordWithSameLengthWithQuery)) {
                if (!this.explored.containsKey(neighbor)) { // Check if the neighbor has been explored

                    //  $f(n) = g(n) + h(n)$ 
                    Integer fn = gn(expandNode) + hn(neighbor);

                    // Create a new node and add it to the priority queue
                    Node newNode = new Node(neighbor, fn);
                    newNode.setPath(new ArrayList<String>(expandNode.getPath()));
                    newNode.addPath(expandNode.getWord());
                    this.lifeNode.add(newNode);
                }
            }
        }
        Long endTime = System.nanoTime();
        this.executionTime = (endTime - startTime) / 1000000;
    }
}
```

Gambar 10. *Class AStar*

Atribut	
Method	
+ AStar(String start, String goal, Dictionary dictionary)	Konstruktor pembentuk objek AStar dengan parameter sebuah String kata awal, String kata tujuan, dan Dictionary yang digunakan.
# Integer gn(Node Parent)	Method untuk menghitung nilai $g(n)$ suatu Node.
# Integer hn(String current)	Method untuk menghitung nilai $h(n)$ suatu Node.
+ void search()	Method abstrak untuk melakukan proses pencarian solusi.

Tabel 6. *Class AStar*

3.4 Kelas WordLadderGUI

Class WordLadderGUI

Kelas ini digunakan untuk membuat GUI pada permainan *Word Ladder* yang melakukan *extends* terhadap kelas *JFrame*.

```
/**
 * Class WordLadderGUI
 * This class is used to create the GUI for Word Ladder Solver
 */
public class WordLadderGUI extends JFrame {
    /**
     * Attributes
     */
    private Dictionary dictionary;           // Dictionary of words
    private JTextField startWordField;       // Start word field
    private JTextField goalWordField;        // Goal word field
    private JComboBox<String> algoritmaBox;  // Algorithm choice box
    private JButton searchButton;           // Search button
    private JTextArea outputArea;           // Output
    private JScrollPane scrollPane;         // Scroll pane

    /**...
    public WordLadderGUI(Dictionary dictionary) {
        this.dictionary = dictionary;

        // Create Window
        setTitle(title:"Word Ladder Solver");
        setSize(width:400, height:700);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLocationRelativeTo(c:null);

        // Create Panel
        JPanel panel = new JPanel();
        panel.setLayout(mgr:null);

        // Welcome Label
        JLabel welcomeLabel = new JLabel(text:"Selamat datang di Word Ladder Solver!");
        welcomeLabel.setFont(new Font(name:"Arial", Font.BOLD, size:16));
        welcomeLabel.setBounds(x:20, y:0, width:360, height:25);
        welcomeLabel.setHorizontalAlignment(SwingConstants.CENTER);
        panel.add(welcomeLabel);

        // input start word
        JLabel startWordLabel = new JLabel(text:"Kata Awal");
        startWordLabel.setBounds(x:20, y:30, width:80, height:25);
        panel.add(startWordLabel);

        startWordField = new JTextField(columns:20);
        startWordField.setBounds(x:100, y:30, width:265, height:25);
        panel.add(startWordField);

        // input goal word
        JLabel goalWordLabel = new JLabel(text:"Kata Akhir");
        goalWordLabel.setBounds(x:20, y:60, width:80, height:25);
        panel.add(goalWordLabel);

        goalWordField = new JTextField(columns:20);
        goalWordField.setBounds(x:100, y:60, width:265, height:25);
        panel.add(goalWordField);

        // input algorithm choice
        JLabel algorithmLabel = new JLabel(text:"Algoritma");
        algorithmLabel.setBounds(x:20, y:90, width:80, height:25);
        panel.add(algorithmLabel);

        String[] algoritmaOptions = {"Uniform Cost Search Algorithm", "Greedy Best First Search Algorithm", "A* Algorithm"};
        algoritmaBox = new JComboBox<>(algoritmaOptions);
        algoritmaBox.setBounds(x:100, y:90, width:265, height:25);
        panel.add(algoritmaBox);
    }
}
```

Gambar 11. Class WordLadderGUI (1)


```

// output area
outputArea = new JTextArea();
scrollPane = new JScrollPane(outputArea); // scroll pane
scrollPane.setBounds(x:20, y:150, width:345, height:495);
outputArea.setFont(new Font(name:"Courier New", Font.PLAIN, size:12));
outputArea.setMargin(new Insets(top:10, left:10, bottom:10, right:10));
panel.add(scrollPane);

// search button
searchButton = new JButton(text:"Cari");
searchButton.setBounds(x:160, y:120, width:75, height:25);
searchButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        runAlgorithm();
    }
});
panel.add(searchButton);

// Add panel to frame
add(panel);
setVisible(true);
}

/**...
private void runAlgorithm() {
    String startWord = startWordField.getText().trim();
    String goalWord = goalWordField.getText().trim();
    String algoritma = (String) algoritmaBox.getSelectedItem();

    // input validation
    if (startWord.equals(anObject:"") || goalWord.equals(anObject:"")) {
        JOptionPane.showMessageDialog(this, message:"Kata awal atau kata tujuan tidak boleh kosong.");
        return;
    } else {
        if (startWord.length() != goalWord.length()) {
            JOptionPane.showMessageDialog(this, message:"Kata awal dan kata tujuan harus memiliki panjang yang sama.");
            return;
        } else {
            if (!dictionary.isWordInDictionary(startWord)) {
                JOptionPane.showMessageDialog(this, message:"Kata awal tidak ada di dalam kamus.");
                return;
            } else {
                if (!dictionary.isWordInDictionary(goalWord)) {
                    JOptionPane.showMessageDialog(this, message:"Kata tujuan tidak ada di dalam kamus.");
                    return;
                }
            }
        }
    }
}

// Run the selected algorithm
SearchAlgorithm searchAlgorithm = null;
switch (algoritma) {
    case "Uniform Cost Search Algorithm":
        System.out.println(x:"Running UCS...");
        searchAlgorithm = new UCS(startWord, goalWord, dictionary);
        break;
    case "Greedy Best First Search Algorithm":
        System.out.println(x:"Running Greedy BFS...");
        searchAlgorithm = new GreedyBFS(startWord, goalWord, dictionary);
        break;
    case "A* Algorithm":
        System.out.println(x:"Running A*...");
        searchAlgorithm = new AStar(startWord, goalWord, dictionary);
        break;
}
searchAlgorithm.search();
System.out.println(x:"Finished!");

// Output the result
Node result = searchAlgorithm.getResult();
outputArea.setText(printResult(result, searchAlgorithm.getExecutionTime(), searchAlgorithm.getVisitedNodes()));
}

```

Gambar 12. Class WordLadderGUI (2)

```

/**...
private String printResult(Node node, Long executionTime, int visitedNodes) {
    String output;

    if (node == null) {
        output = "Solusi tidak ditemukan.\n";
    } else {
        output = "Panjang jalur: " + (node.getPath().size()) + "\n";

        output += "Jalur:\n";
        for (int i = 0; i < node.getPath().size(); i++) {
            output += "    " + (i + 1) + ". " + node.getPath().get(i).toLowerCase() + "\n";
        }
        output += "    " + (node.getPath().size() + 1) + ". " + node.getWord().toLowerCase() + "\n";
    }

    output += "Banyak node dikunjungi : " + visitedNodes + " node\n";

    if (executionTime > 1000) {
        output += "Waktu eksekusi          : " + executionTime / 1000 + " s " + executionTime % 1000 + " ms\n";
    } else {
        output += "Waktu eksekusi          : " + executionTime + " ms\n";
    }

    return output;
}
}

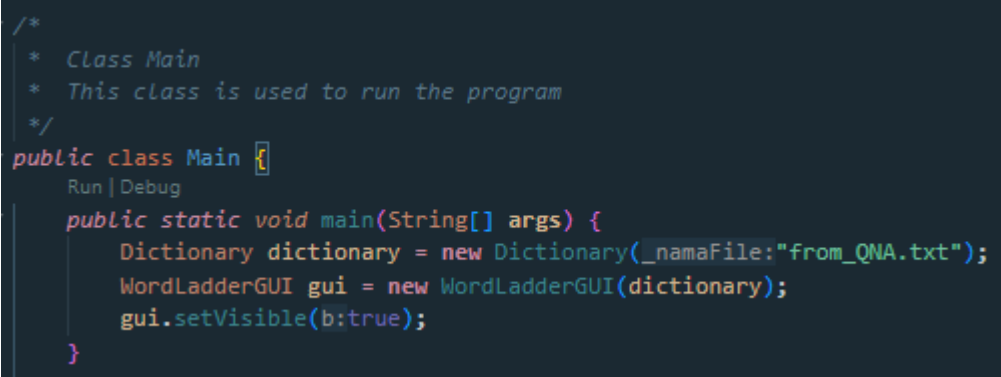
```

Gambar 13. Class WordLadderGUI (3)

Atribut	
- Dictionary dictionary	Dictionari yang digunakan.
- JTextField startWordField	Tempat menerima input kata awal.
- JTextField goalWordField	Tempat menerima input kata tujuan.
- JComboBox<String> algoritmaBox	Tempat memilih algoritma yang akan digunakan.
- JButton searchButton	Tombol <i>search</i> .
- JTextArea outputArea	Tempat hasil keluaran ditampilkan.
- JScrollPane scrollPane	Memberikan kemampuan untuk melakukan <i>scrolling</i> .
Method	
+ WordLadderGUI (Dictionary dictionary)	Konstruktor pembentuk kelas WordLadderGUI dengan parameter dictionary yang digunakan.
+ void runAlgorithm()	Menampilkan GUI, menerima input pengguna, melakukan proses pencarian, dan menampilkan hasil pencarian ke layar.
- String printResult(Node node, long executionTime, int visitedNodes)	Fungsi untuk mengubah hasil pencarian menjadi satu string tunggal untuk ditampilkan di layar.

Tabel 7. Class WordLadderGUI

3.5 Kelas Main

Class Main	
Sebagai kelas pintu masuk program.	
 <pre> /* * Class Main * This class is used to run the program */ public class Main { Run Debug public static void main(String[] args) { Dictionary dictionary = new Dictionary(_namaFile:"from_QNA.txt"); WordLadderGUI gui = new WordLadderGUI(dictionary); gui.setVisible(b:true); } </pre>	
Gambar 14. Class Main	
Atribut	
Method	
public static void main(String[] args)	Entry point untuk program <i>Word Ladder Solver</i> .

Tabel 8. Class Main

BAB IV

UJI COBA

4.1 Keterangan Proses Input dan Output

1. Menjalankan Program

Jalankan program dengan memasukkan perintah berikut ke dalam terminal.

Clone repository

```
> git clone https://github.com/Agil0975/Tucil3_13522006
```

Masuk ke root directory

```
> cd Tucil3_13522006
```

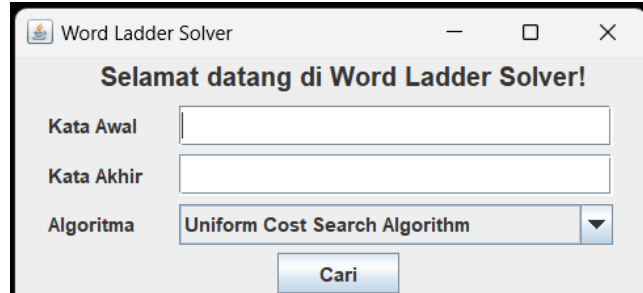
Compile program

```
> javac -d bin src/*.java
```

Jalankan program

```
> java -cp bin src.Main
```

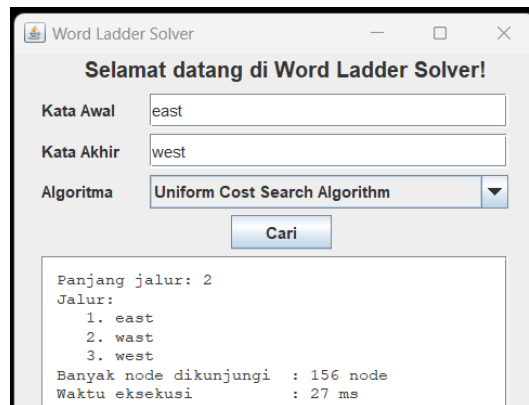
2. Input



Gambar 15. GUI Memasukkan *Input*

Pada kolom kata awal, masukkan kata awal yang ingin dijadikan sebagai kata permulaan pada permainan *Word Ladder*. Pada kolom kata tujuan, masukkan kata tujuan yang ingin dicapai dari kata awal. Pada kolom algoritma, pilih algoritma yang ingin digunakan (UCS, *Greedy* BeFS, dan A*). Setelah memasukkan semua masukan yang diperlukan, tekan tombol “cari” dan tunggu program menampilkan hasilnya.

3. Output



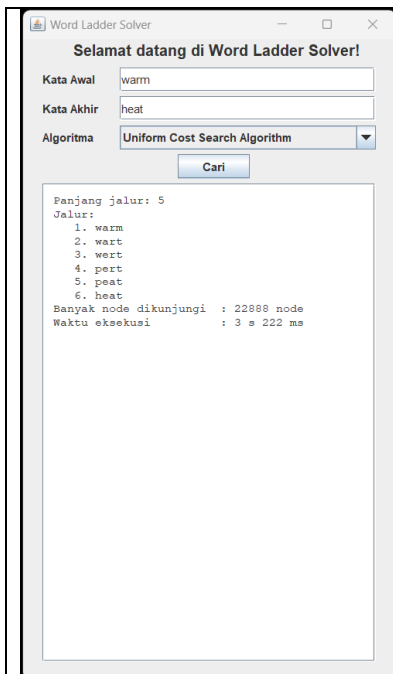
Gambar 16. Hasil Pencarian

Hasil pencarian akan ditampilkan di bagian bawah tombol “cari”. Hasil keluaran akan berupa, panjang *path*, *path* dari kata awal menuju kata tujuan, banyaknya *node* yang dikunjungi selama proses pencarian, dan lama pencarian dalam satuan detik dan milidetik.

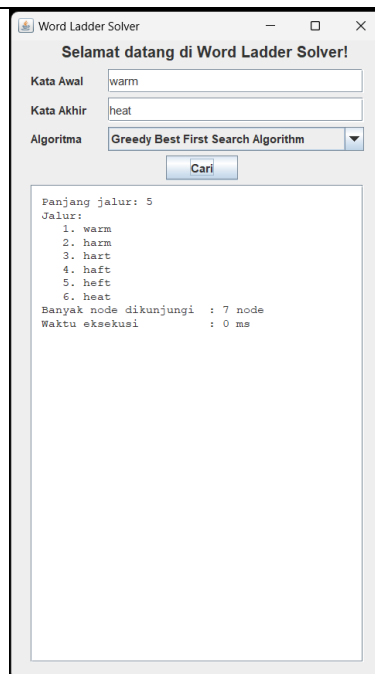
4.2 Pengujian Algoritma

Proses pengujian algoritma dalam program ini menggunakan *dictionary* yang tersedia pada link berikut docs.oracle.com/javase/tutorial/collections/interfaces/examples/dictionary.txt yang diberikan oleh tim asisten pada *sheet* QnA. Total kata yang terdapat pada *dictionary* tersebut adalah sebanyak 80.368 kata dengan variasi panjang kata berkisar pada 2-8 huruf.

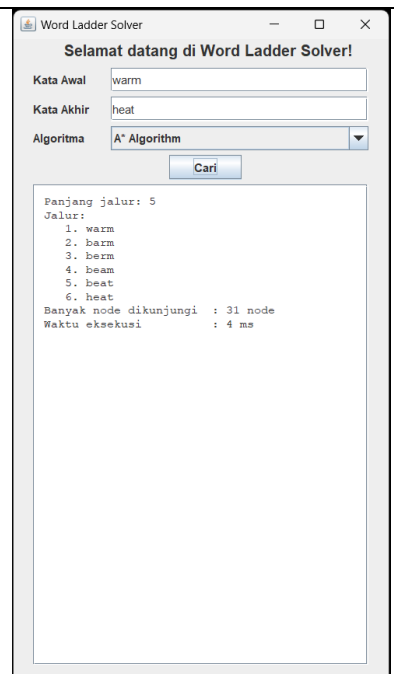
1. Test Case 1



Gambar 17. Hasil Pencarian
TC 1 dengan Algoritma UCS



Gambar 18. Hasil Pencarian
TC 1 dengan Algoritma GBeFS

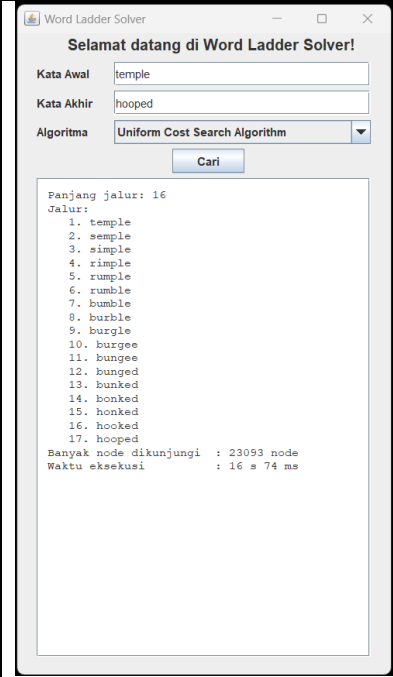
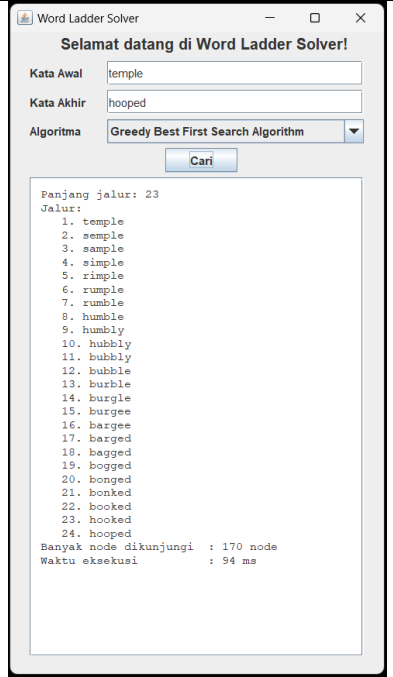
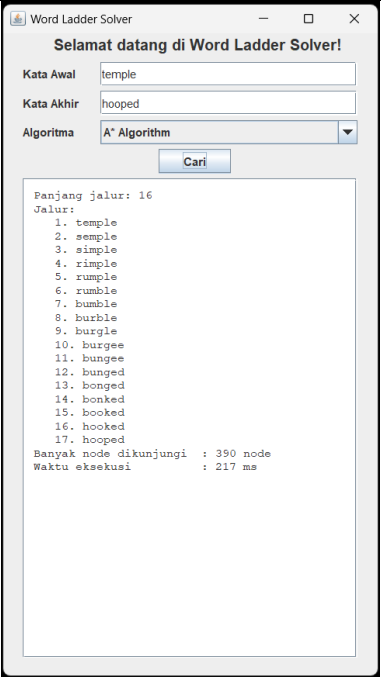


Gambar 19. Hasil Pencarian
TC 1 dengan Algoritma A*

Kata awal : *warm*
Kata tujuan : *heat*

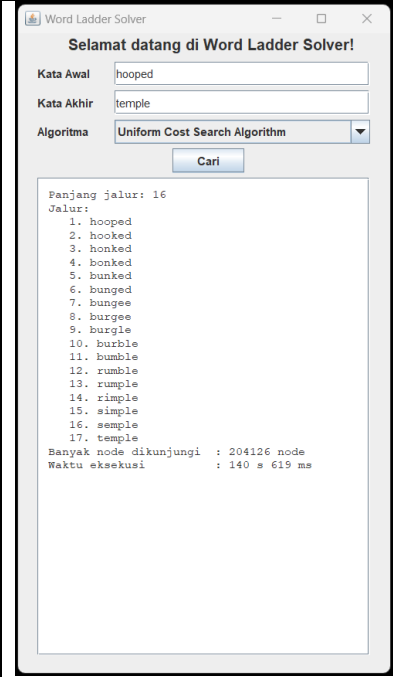
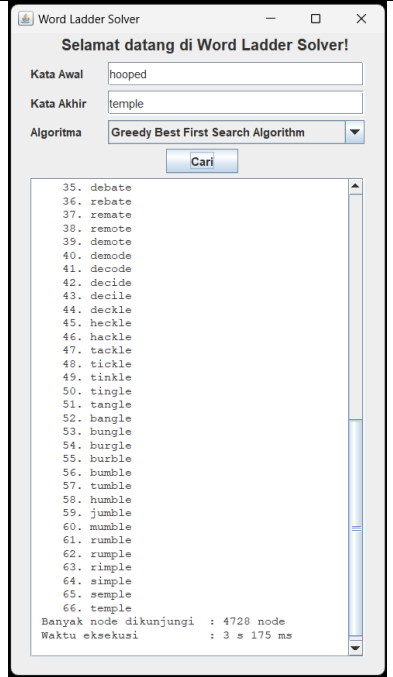
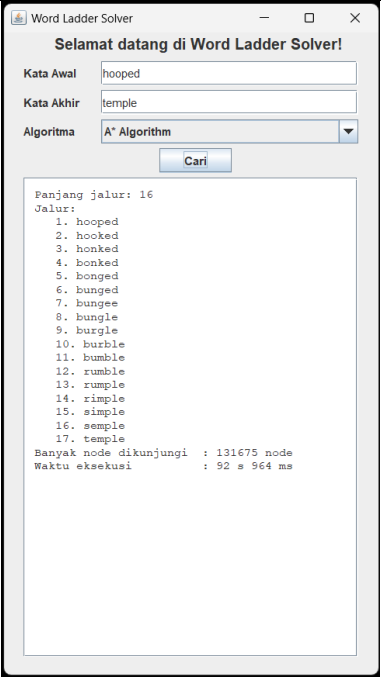
Tabel 10. Test Case 1

2. Test Case 2

		
<p>Gambar 20. Hasil Pencarian TC 2 dengan Algoritma UCS</p>	<p>Gambar 21. Hasil Pencarian TC 2 dengan Algoritma GBeFS</p>	<p>Gambar 22. Hasil Pencarian TC 2 dengan Algoritma A*</p>
<p>Kata awal : <i>temple</i> Kata tujuan : <i>hooped</i></p>		

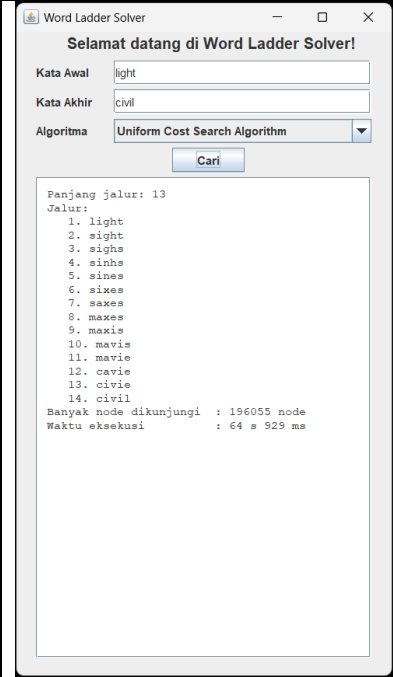
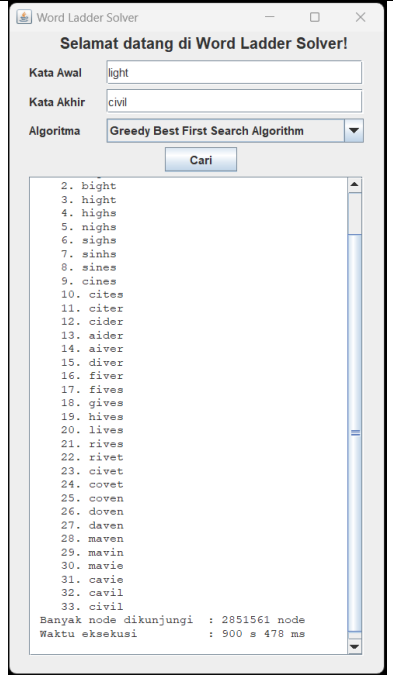
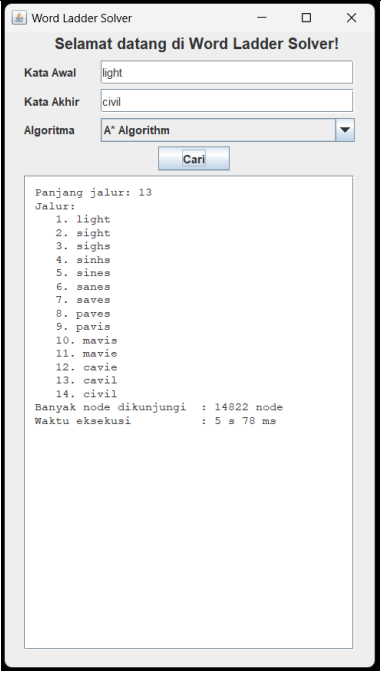
Tabel 11. Test Case 2

3. Test Case 3

		
<p>Gambar 23. Hasil Pencarian TC 3 dengan Algoritma UCS</p>	<p>Gambar 24. Hasil Pencarian TC 3 dengan Algoritma GBeFS</p>	<p>Gambar 25. Hasil Pencarian TC 3 dengan Algoritma A*</p>
<p>Kata awal : <i>hooped</i> Kata tujuan : <i>temple</i></p>		

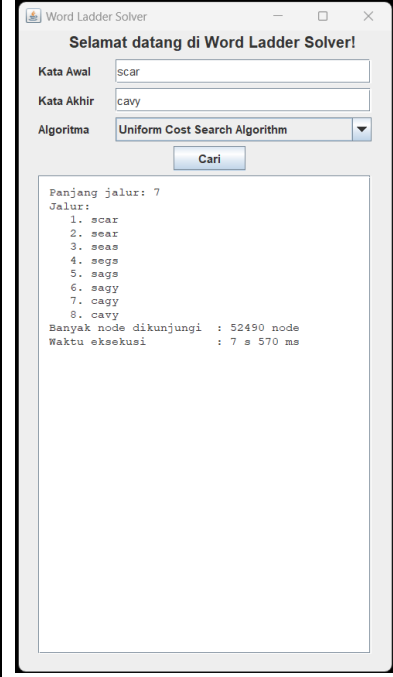
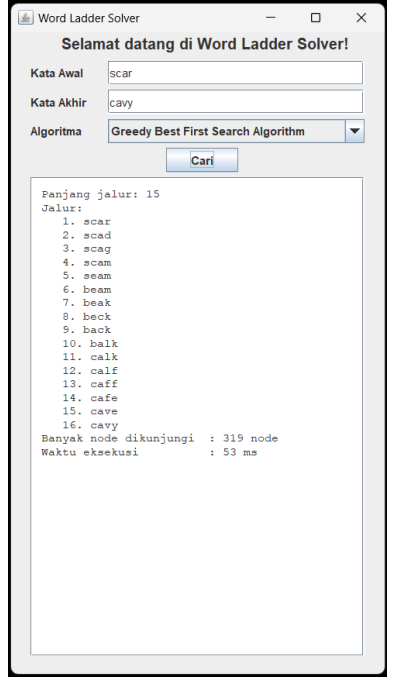
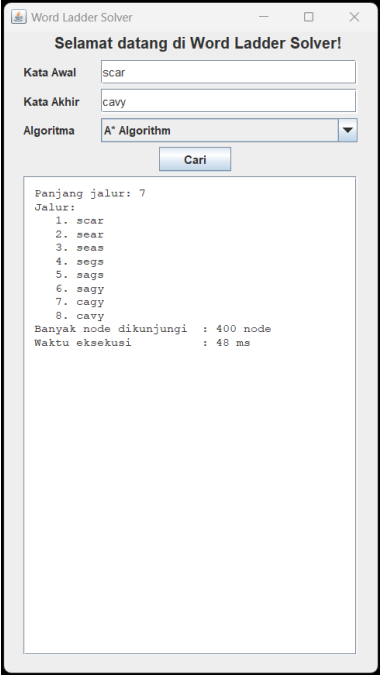
Tabel 12. Test Case 3

4. Test Case 4

		
<p>Gambar 26. Hasil Pencarian TC 4 dengan Algoritma UCS</p>	<p>Gambar 27. Hasil Pencarian TC 4 dengan Algoritma GBFS</p>	<p>Gambar 28. Hasil Pencarian TC 4 dengan Algoritma A*</p>
<p>Kata awal : <i>light</i> Kata tujuan : <i>civil</i></p>		

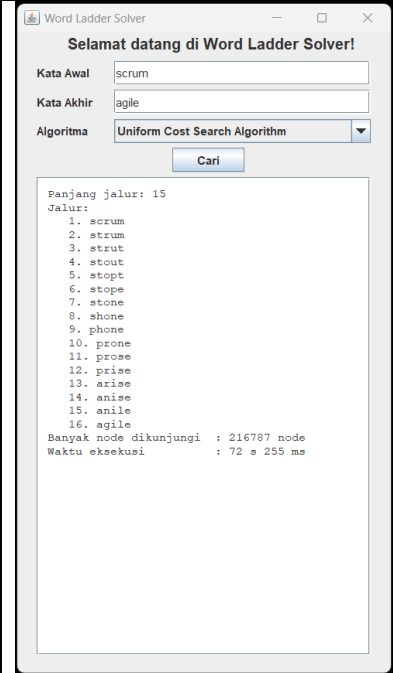
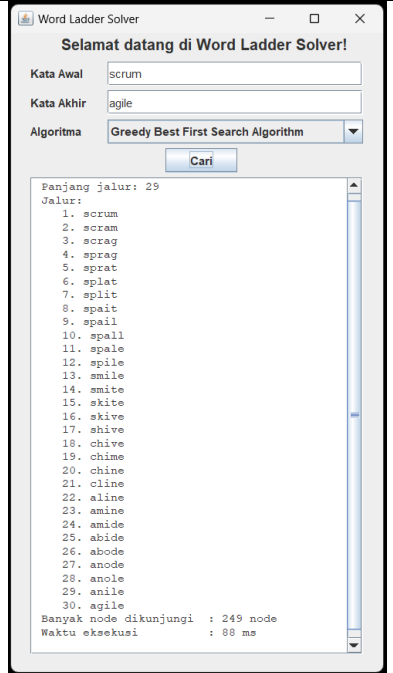
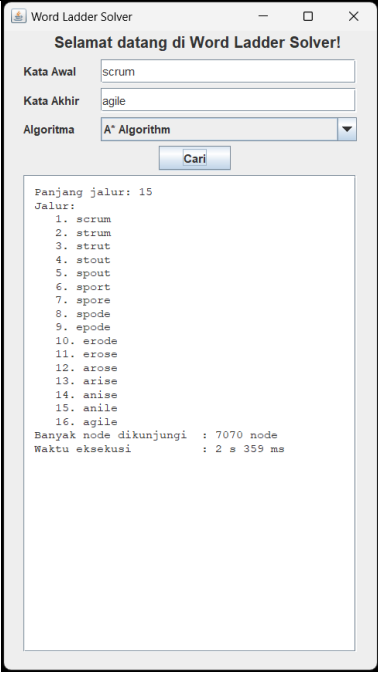
Tabel 13. Test Case 4

5. Test Case 5

		
<p>Gambar 29. Hasil Pencarian TC 5 dengan Algoritma UCS</p>	<p>Gambar 30. Hasil Pencarian TC 5 dengan Algoritma GBFS</p>	<p>Gambar 31. Hasil Pencarian TC 5 dengan Algoritma A*</p>
<p>Kata awal : <i>scar</i> Kata tujuan : <i>cavy</i></p>		

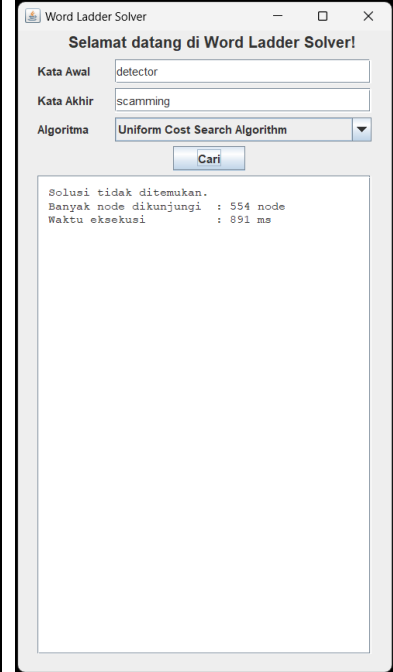
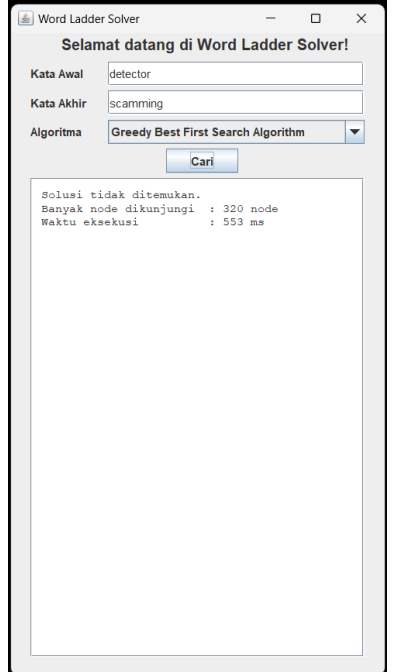
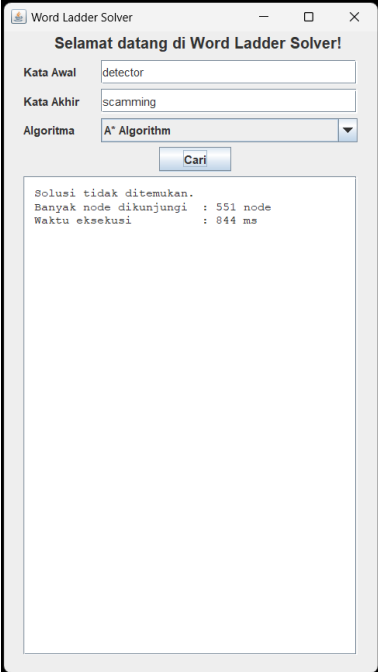
Tabel 14. Test Case 5

6. Test Case 6

		
<p>Gambar 32. Hasil Pencarian TC 6 dengan Algoritma UCS</p>	<p>Gambar 33. Hasil Pencarian TC 6 dengan Algoritma GBFS</p>	<p>Gambar 34. Hasil Pencarian TC 6 dengan Algoritma A*</p>
<p>Kata awal : <i>scrum</i> Kata tujuan : <i>agile</i></p>		

Tabel 15. Test Case 6

7. Test Case 7

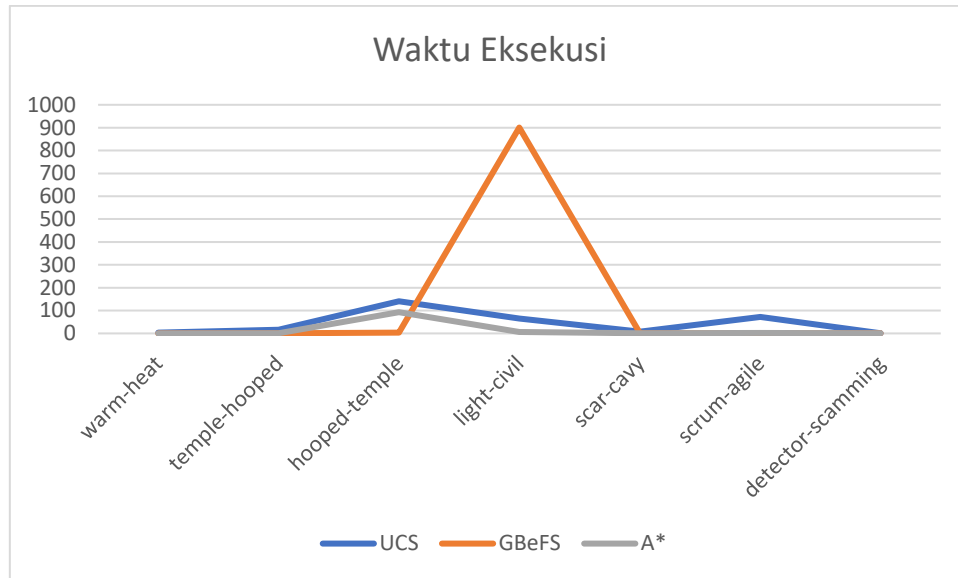
		
<p>Gambar 35. Hasil Pencarian TC 7 dengan Algoritma UCS</p>	<p>Gambar 36. Hasil Pencarian TC 7 dengan Algoritma GBFS</p>	<p>Gambar 37. Hasil Pencarian TC 7 dengan Algoritma A*</p>
<p>Kata awal : <i>detector</i> Kata tujuan : <i>scamming</i></p>		

Tabel 16. Test Case 7

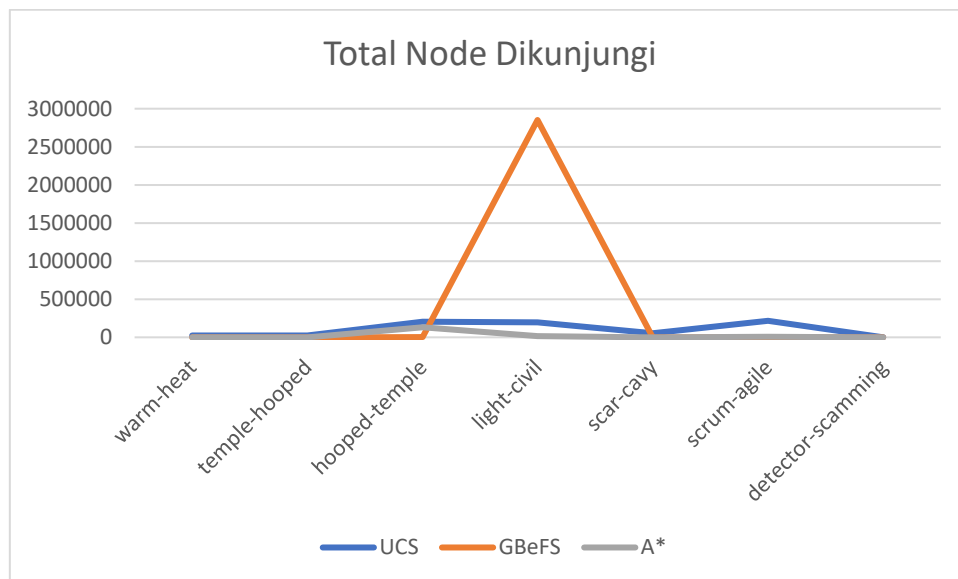
BAB V

ANALISIS SOLUSI

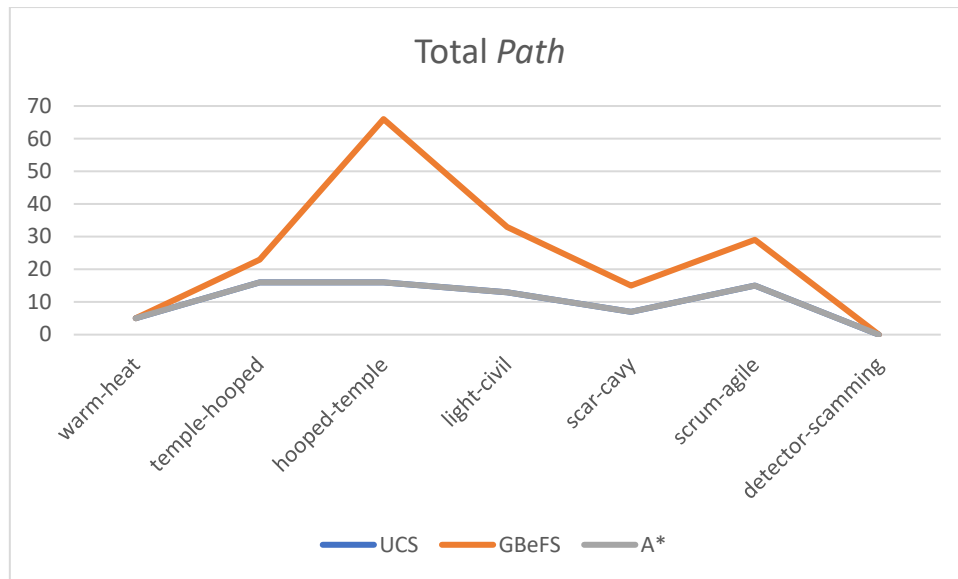
Berikut adalah grafik perbandingan hasil uji coba antara ketiga algoritma berdasarkan uji coba yang dilakukan pada bab sebelumnya.



Gambar 38. Perbandingan Hasil Uji Ketiga Algoritma dengan Parameter Waktu Eksekusi



Gambar 39. Perbandingan Hasil Uji Ketiga Algoritma dengan Parameter Total Node yang Dikunjungi



Gambar 40. Perbandingan Hasil Uji Ketiga Algoritma dengan Parameter Panjang Total *Path*

5.1 Algoritma UCS

Berdasarkan hasil uji coba yang dilakukan pada bab sebelumnya, dapat dilihat bahwa dari segi optimalitas hasil (panjang jalur yang dihasilkan paling minimal), algoritma UCS terbilang optimal, karena selalu berhasil menemukan solusi dengan panjang jalur minimal (jika solusinya ada), seperti yang terlihat pada grafik pada **Gambar 40**. Hal ini karena dalam permainan *Word Ladder*, algoritma UCS memiliki perilaku yang hampir sama dengan algoritma BFS dalam segi pengunjungan *node-node*-nya. Oleh karena itu, algoritma UCS akan selalu menemukan solusi pada kedalaman yang minimum sehingga solusi yang dihasilkan selalu optimal.

Akan tetapi, dari segi waktu eksekusinya, UCS membutuhkan waktu eksekusi yang lebih lama pada banyak kasus, karena UCS lebih banyak menghabiskan waktu pencarian di kedalaman yang lebih rendah sebelum akhirnya berpindah ke kedalaman yang lebih tinggi. Hal ini terlihat dari 7 percobaan yang dilakukan, 6 diantaranya menghasilkan waktu eksekusi yang tertinggi jika dibandingkan dengan dua algoritma lainnya.

Begitu pula dari segi ruang memori yang dibutuhkan. Algoritma UCS membutuhkan ruang memori yang lebih banyak pada banyak kasus. Hal ini dapat dilihat pada grafik pada **Gambar 39**, dimana dari 7 percobaan, 6 diantaranya menghasilkan *node* yang dikunjungi lebih banyak dibandingkan dua algoritma lainnya. Sekali lagi hal ini terjadi karena algoritma UCS akan mengunjungi setiap *node* yang ada pada kedalaman lebih rendah terlebih dahulu sebelum akhirnya berpindah ke kedalaman yang lebih tinggi. Jika seandainya setiap *node* rata-rata memiliki jumlah *node* tetangga sebanyak b , dan solusi berada pada kedalaman l , maka kompleksitas ruang untuk algoritma UCS adalah $O(b^l)$.

5.2 Algoritma Greedy BeFS

Berdasarkan hasil uji coba yang dilakukan, dapat dilihat bahwa dari segi optimalitas hasil (panjang jalur yang dihasilkan paling minimal), algoritma *greedy* BeFS terbilang tidak optimal (atau tidak menjamin solusi yang dihasilkan optimal). Hal ini dapat dilihat pada grafik pada **Gambar 40**, dimana algoritma *greedy* BeFS menghasilkan panjang jalur yang lebih panjang daripada algoritma UCS dan A*, dan bahkan perbedaannya bisa sangat besar daripada algoritma UCS dan A* (pada kasus *hooped* ke *temple* contohnya, dimana algoritma *greedy* BeFS memiliki panjang solusi hingga 65, sedangkan UCS dan A* hanya 16.). Hal ini terjadi karena sifat algoritma *greedy* BeFS yang hanya melihat kandidat solusi berdasarkan nilai *heuristik* $h(n)$ saja dan sepenuhnya mengabaikan nilai $g(n)$ sehingga setiap kali pemilihan *node* dilakukan, mungkin mengarahkan pada nilai *heuristik* yang lebih kecil, namun di saat bersamaan nilai $g(n)$ yang dihasilkan semakin besar tanpa disadari.

Adapun dari segi waktu eksekusinya, *greedy* BeFS membutuhkan waktu eksekusi yang lebih singkat pada banyak kasus, seperti yang terlihat pada grafik pada **Gambar 39**, dimana 6 dari 7 percobaan, algoritma *greedy* BeFS berhasil menemukan solusi dalam waktu yang lebih cepat dibandingkan algoritma UCS dan A*. Hal ini terjadi karena algoritma *greedy* BeFS hanya melihat nilai *heuristik* $h(n)$ saja, yang pada banyak kasus mampu mengarahkan proses pencarian menuju simpul tujuan. Akibatnya juga, sifat algoritma pencarian *greedy* BeFS akan mirip dengan algoritma DFS yang mencoba masuk ke *node* yang lebih dalam terlebih dahulu (yang berarti lebih dekat dengan solusi) sebelum berpindah ke *node* di sebelahnya.

Namun, dari uji coba yang dilakukan, terdapat satu kasus penyimpangan, dimana algoritma *greedy* BeFS membutuhkan waktu eksekusi yang jauh lebih lama dari pada algoritma UCS dan A*, yaitu pada kasus *light* ke *civil*. Ada beberapa hal yang mungkin menyebabkan hal tersebut. Pertama adalah sifat penambahan simpul hidup pada algoritma pencarian, dimana simpul hidup yang belum pernah di-*expand*, akan selalu ditambahkan ke dalam *priority queue*, selama ia belum pernah di-*expand* sebelumnya, sehingga kata yang sama mungkin ditambahkan berkali-kali ke dalam *queue* karena sifat pencarian *greedy* BeFS yang mendalam, bukan melebar. Kemungkinan kedua adalah kata-kata yang menjadi bagian dari solusi mungkin lebih banyak ditambahkan ke bagian tengah *queue*, bukan pada bagian depan (karena kata-kata dengan nilai *heuristik* yang sama, akan diurutkan berdasarkan urutan alfabetis), sehingga simpul tersebut baru di-*expand* setelah waktu yang cukup lama.

Adapun untuk segi penggunaan memori, karena algoritma *greedy* BeFS yang lebih cepat dalam menemukan solusi, maka *node* yang dihasilkan pada banyak kasus juga lebih sedikit dibandingkan algoritma UCS dan A*. Jika sebuah *node* rata-rata memiliki *node* tetangga sebanyak b dan solusi berada pada kedalaman l , maka rata-rata kompleksitas ruang untuk *greedy* BeFS adalah $O(bl)$. Namun, karena sifat penambahan *node* yang berulang-ulang, kompleksitas ruang yang sesungguhnya mungkin sulit untuk didapatkan. Selain itu, juga bisa terjadi anomali, dimana *greedy* BeFS mengunjungi *node* yang jauh lebih banyak daripada UCS dan A*, seperti pada kasus sebelumnya (*light* ke *civil*), dimana *greedy* BeFS mengunjungi hingga 2.851.561 *node*, sedangkan UCS hanya 196.005 *node* dan A* hanya 14.822 *node*.

5.3 Algoritma A*

Berdasarkan hasil percobaan yang dilakukan, dapat dilihat bahwa algoritma A* merupakan algoritma yang memadukan dua kelebihan algoritma sebelumnya, dimana solusi yang dihasilkan optimal seperti algoritma UCS, dan bisa lebih cepat dalam menemukan solusi seperti algoritma *greedy* BeFS.

Dari segi optimalitas hasil (panjang jalur yang dihasilkan paling minimal), algoritma A* selalu berhasil menemukan solusi optimal, seperti algoritma UCS. Hal ini dapat dilihat pada grafik pada **Gambar 40**, dimana grafik algoritma A* berimpit dengan grafik algoritma UCS.

Dari segi waktu eksekusi, algoritma A* selalu lebih baik daripada algoritma UCS. Walau waktu eksekusinya pada kebanyakan kasus lebih lama dari algoritma *greedy* BeFS, tetapi perbedaannya tidaklah terlalu jauh. Pada kebanyakan kasus, solusi selalu dapat ditemukan dalam waktu kurang dari 10 detik, yang tergolong cepat.

Dari segi ruang memori yang dibutuhkan. Algoritma A* selalu membutuhkan ruang memori yang lebih sedikit daripada algoritma UCS, karena A* mampu menemukan solusi lebih cepat. Akan tetapi, dibandingkan algoritma *greedy* BeFS, pada kebanyakan kasus, algoritma A* membutuhkan lebih banyak ruang memori daripada algoritma *greedy* BeFS. Hal ini dapat dilihat lebih jelas pada grafik pada **Gambar 39**, dimana grafik algoritma A* berada di tengah-tengah antara algoritma UCS dengan *greedy* BeFS. Walau begitu, pada kasus terburuk, kompleksitas ruang untuk algoritma A* akan sama dengan algoritma UCS, yaitu $O(b^l)$.

Ketiga hal di atas, dapat dijelaskan dengan menggunakan perhitungan nilai biaya pada algoritma A*, yang memperhitungkan biaya sesungguhnya yang telah dikeluarkan ($g(n)$) serta memperkirakan biaya minimal yang dibutuhkan untuk mencapai solusi ($h(n)$). Penggunaan nilai $g(n)$ menyebabkan algoritma A* mampu mencari solusi dengan biaya seminimal mungkin. Adapun penggunaan nilai $h(n)$ menyebabkan algoritma A* mampu mencari *node* yang semakin mendekatkan dengan *node* tujuan.

BAB VI

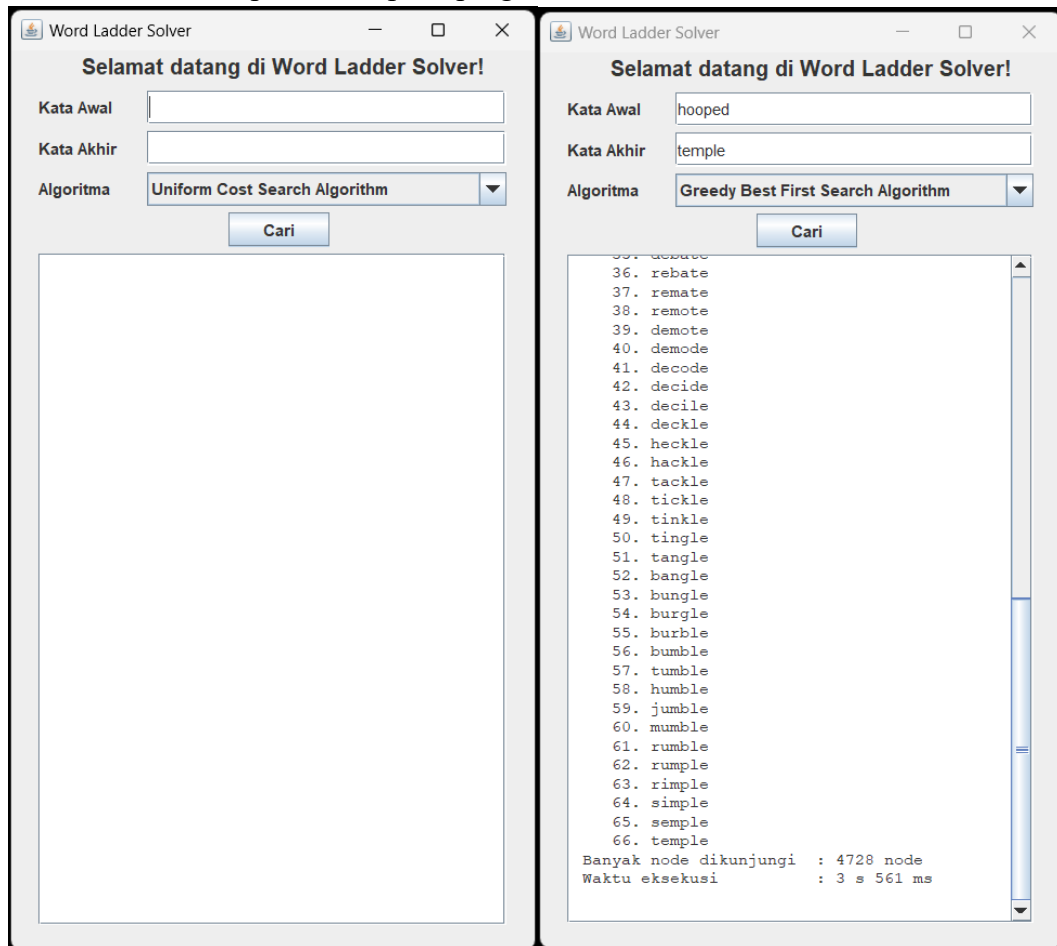
IMPLEMENTASI BONUS

6.1 Pembuatan GUI

Pembuatan GUI (*graphical user interface*) pada program permainan *Word Ladder* ini memanfaatkan *library* javax.swing dan java.awt yang disediakan oleh bahasa pemrograman Java.

Pembuatan GUI dilakukan dengan membuat sebuah kelas bernama WordLadderGUI yang meng-*extends* (menurunkan) kelas JFrame. *Toolkit* yang disediakan oleh javax.swing akan digunakan untuk membuat tampilan antarmuka program, sedangkan java.awt akan melakukan aksi menjalankan algoritma jika seandainya tombol “cari” ditekan.

Berikut adalah tampilan GUI pada program ini.



Gambar 41. GUI Program *Word Ladder*

DAFTAR PUSTAKA

docs.oracle.com/javase/tutorial/collections/interfaces/examples/dictionary.txt (Diakses pada 4 Mei 2024).

<https://github.com/dwyl/english-words.git> (Diakses pada 1 Mei 2024).

[Route-Planning-Bagian1-2021.pdf \(itb.ac.id\)](#) (Diakses pada 1 Mei 2024).

[Route-Planning-Bagian2-2021.pdf \(itb.ac.id\)](#) (Diakses pada 1 Mei 2024).

LAMPIRAN

1. Link Repository

Link : https://github.com/Agil0975/Tucil3_13522006.git

2. Tabel *Checkpoint* Program

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.	✓	