



Institut Teknologi Bandung

Sekolah Teknik Elektro dan Informatika

Program Studi Informatika Semester II 2023/2024

Laporan Tugas Kecil III

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

IF2211 Strategi Algoritma

Zahira Dina Amalia K01 - 13522085

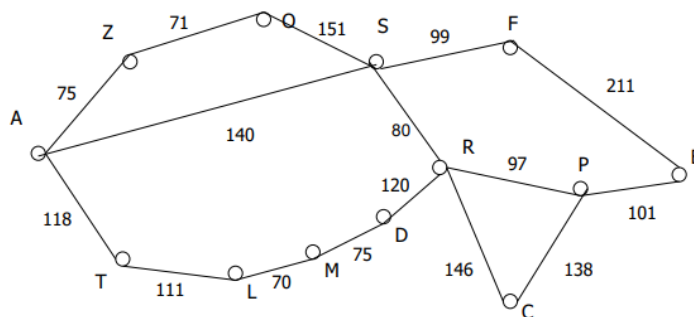
Daftar Isi

Daftar Isi.....	2
Analisis Implementasi dalam Algoritma UCS, Greedy Best First Search, dan A*.....	3
Source Code Program.....	9
Tangkapan Layar Input dan Output.....	23
1. Test case 1: frown → smile.....	23
2. Test case 1: jean → gene.....	24
3. Test case 3: veep → exec.....	26
4. Test case 4: strong → dreams.....	27
5. Test case 5: grass → roots.....	30
6. Test case 6: pray → amen.....	32
7. Test case 7: edge cases.....	34
Hasil Analisis Perbandingan Solusi UCS, Greedy Best First Search, dan A*.....	37
Implementasi Bonus.....	39
Lampiran.....	48

Analisis Implementasi dalam Algoritma UCS, Greedy Best First Search, dan A*

Dalam analisis dan implementasi algoritma *Uniform Cost Search* (UCS), *Greedy Best First Search*, dan A* dalam penyelesaian persoalan rute, seperti yang dijelaskan dalam materi kuliah IF2211 Strategi Algoritma, diperlukan pemahaman mengenai fungsi $f(n)$, $g(n)$ dan $h(n)$. Pada setiap jenis algoritma diperlukan $f(n)$ yang merupakan fungsi evaluasi sebagai estimasi biaya untuk mencapai simpul tujuan melalui simpul n . Definisinya pada setiap macam algoritma akan menjadi pembeda dari masing-masing algoritma.

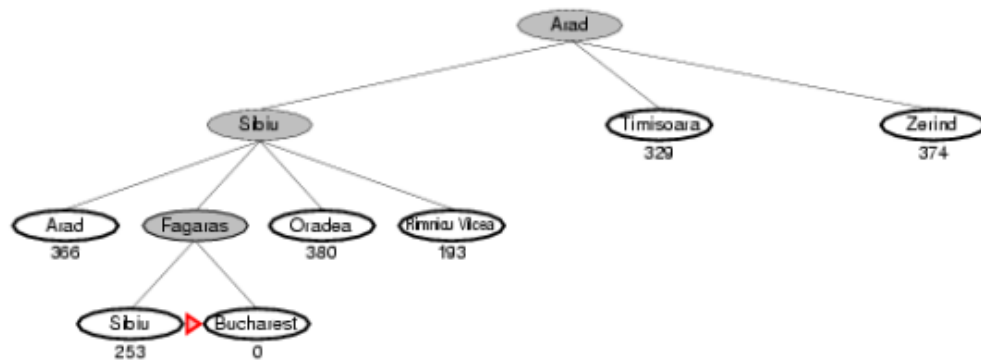
Dalam UCS, $f(n)$ identik dengan $g(n)$, yakni mewakili biaya yang sudah dikeluarkan untuk mencapai simpul n . UCS tidak menggunakan fungsi heuristik eksternal untuk estimasi biaya tambahan, $f(n)$ dalam konteks UCS tidak digunakan secara eksplisit karena pada dasarnya algoritma UCS (*Uniform Cost Search*) memprioritaskan ekspansi simpul-simpul dengan biaya terendah yang diperlukan untuk mencapai mereka. $g(n)$ dalam UCS secara khusus menggambarkan biaya aktual yang diperlukan untuk mencapai simpul n dari simpul awal yakni nilai yang diakumulasikan selama pencarian saat algoritma membangun jalur dari simpul awal ke simpul tujuan. Dalam penjelasan teoretis, kadang-kadang $f(n)$ digunakan untuk menunjukkan fungsi heuristik, sementara $g(n)$ digunakan untuk menunjukkan biaya aktual. Tetapi dalam konteks penggunaan praktis UCS, $f(n)$ tidak digunakan secara eksplisit karena UCS hanya mempertimbangkan biaya aktual $g(n)$ dari simpul awal ke simpul yang sedang dieksplorasi. Berikut contoh aplikasi algoritma UCS berdasarkan graf yang disediakan.



Simpul-E	Simpul Hidup
A	$Z_{A-75}, T_{A-118}, S_{A-140}$
Z_{A-75}	$T_{A-118}, S_{A-140}, O_{AZ-146}$
T_{A-118}	$S_{A-140}, O_{AZ-146}, L_{AT-229}$
S_{A-140}	$O_{AZ-146}, R_{AS-220}, L_{AT-229}, F_{AS-239}, O_{AS-291}$
O_{AZ-146}	$R_{AS-220}, L_{AT-229}, F_{AS-239}, O_{AS-291}$
R_{AS-220}	$L_{AT-229}, F_{AS-239}, O_{AS-291}, P_{ASR-317}, D_{ASR-340}, C_{ASR-366}$
L_{AT-229}	$F_{AS-239}, O_{AS-291}, M_{ATL-299}, P_{ASR-317}, D_{ASR-340}, C_{ASR-366}$
F_{AS-239}	$O_{AS-291}, M_{ATL-299}, P_{ASR-317}, D_{ASR-340}, C_{ASR-366}, B_{ASF-450}$
O_{AS-291}	$M_{ATL-299}, P_{ASR-317}, D_{ASR-340}, C_{ASR-366}, B_{ASF-450}$
$M_{ATL-299}$	$P_{ASR-317}, D_{ASR-340}, D_{ATLM-364}, C_{ASR-366}, B_{ASF-450}$
$P_{ASR-317}$	$D_{ASR-340}, D_{ATLM-364}, C_{ASR-366}, B_{ASRP-418}, C_{ASRP-455}, B_{ASF-450}$
$D_{ASR-340}$	$D_{ATLM-364}, C_{ASR-366}, B_{ASRP-418}, C_{ASRP-455}, B_{ASF-450}$
$D_{ATLM-364}$	$C_{ASR-366}, B_{ASRP-418}, C_{ASRP-455}, B_{ASF-450}$
$C_{ASR-366}$	$B_{ASRP-418}, C_{ASRP-455}, B_{ASF-450}$
$B_{ASRP-418}$	Solusi ketemu

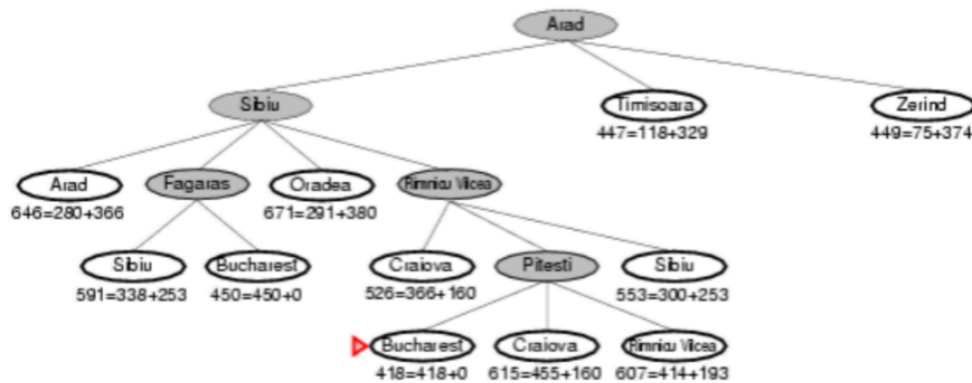
Maka dengan algoritma ini akan didapatkan jalur $A \rightarrow S \rightarrow R \rightarrow P \rightarrow B$ dengan biaya 418 yang merupakan solusi yang paling optimal.

Adapun pada Greedy Best First Search, evaluasi fungsi berupa $f(n) = h(n)$, di mana $h(n)$ adalah estimasi biaya dari simpul n ke tujuan. Algoritma ini memilih simpul yang dianggap paling dekat dengan tujuan untuk dieksplorasi berikutnya. Dengan pendekatan ini, algoritma dapat memberikan hasil pencarian yang cepat. Akan tetapi dalam beberapa kasus, algoritma ini cenderung terjebak dalam lokal minimum. Hal ini dikarenakan ia memilih simpul-simpul yang tampaknya dekat dengan tujuan berdasarkan estimasi heuristik, tetapi sebenarnya tidak mengarah ke solusi optimal secara keseluruhan. Kelemahan ini disebabkan oleh sifat serakah GBFS yang hanya mempertimbangkan estimasi biaya lokal saat memilih langkah selanjutnya, tanpa mempertimbangkan gambaran keseluruhan ruang pencarian. Pada algoritma GBFS, keputusan yang sudah diambil tidak dapat diperbaiki. Artinya, jika algoritma telah memilih jalur yang terbukti tidak mengarah ke solusi optimal, ia tidak dapat mengoreksi langkah-langkahnya. Hal ini membuat GBFS kurang cocok untuk masalah di mana perubahan jalur selama pencarian diperlukan untuk mencapai solusi optimal. Berikut contoh algoritma Greedy Best First Search,



terlihat bahwa algoritma ini akan memberikan jalur $A \rightarrow S \rightarrow F \rightarrow B$ dengan biaya 450 yang bukan merupakan solusi paling optimal.

Sementara itu, algoritma A* menggunakan evaluasi fungsi $f(n) = g(n) + h(n)$ yang memadukan biaya yang sudah dikeluarkan $g(n)$ dengan estimasi biaya ke tujuan $h(n)$. Dalam konteks algoritma A*, $h(n)$ mewakili heuristik, sering kali dihitung sebagai estimasi jarak lurus (straight-line distance) dari simpul n ke tujuan. Sementara itu, $g(n)$ seperti yang disebutkan sebelumnya, adalah biaya aktual yang telah dikeluarkan dari simpul awal hingga simpul n , yang digunakan untuk memperhitungkan biaya aktual yang telah ditempuh untuk mencapai simpul n . Dengan menggunakan nilai $f(n)$, $g(n)$, dan $h(n)$ algoritma A* dapat mempertimbangkan kedua biaya aktual dan estimasi biaya ke tujuan saat memilih simpul berikutnya untuk dieksplorasi. Dengan demikian, A* berusaha untuk mencari jalur yang optimal secara keseluruhan, dengan mempertimbangkan kedua faktor ini. Ini membuat A* menjadi salah satu algoritma pencarian yang paling efisien dan sering digunakan dalam berbagai aplikasi yang melibatkan pencarian jalur, seperti navigasi rute, perencanaan logistik, dan permainan video. Berikut contoh penggunaan algoritma ini pada contoh yang sama dengan algoritma sebelumnya,



Algoritma A* akan menemukan solusi yang paling optimal jika heuristik yang digunakan *admissible*. Sebuah heuristik $h(n)$ dikatakan *admissible* jika untuk setiap simpul n dalam ruang pencarian, nilainya tidak pernah lebih besar dari nilai sebenarnya $h^*(n)$, di mana $h^*(n)$ adalah biaya sebenarnya untuk mencapai keadaan tujuan dari simpul n . Contoh, dalam pencarian rute, jika heuristik mengestimasi jarak langsung dari simpul saat ini ke tujuan, heuristik tersebut *admissible* karena tidak mungkin jarak sebenarnya lebih besar dari estimasi tersebut. Namun, jika heuristik memperkirakan jarak yang lebih pendek dari yang sebenarnya, maka itu tidak *admissible*. Dengan kata lain, heuristik *admissible* dapat memberikan estimasi yang konservatif tetapi tidak terlalu pesimis terhadap biaya sebenarnya untuk mencapai tujuan. Dengan kata lain, heuristik ini tidak pernah memperkirakan biaya mencapai tujuan dengan terlalu tinggi. yang membuatnya memiliki sifat optimis.

Dengan menggunakan heuristik yang *admissible*, algoritma A* akan selalu menemukan solusi optimal jika solusi tersebut ada dikarenakan A* menggunakan perkiraan biaya (heuristik) untuk memutuskan urutan pencarian dan algoritma tidak akan terjebak dalam mengejar solusi yang terlalu berlebihan atau terperangkap dalam pencarian yang tidak produktif. Dalam konteks ini, jika sebuah heuristik $h(n)$ dapat diterima, maka penggunaan algoritma A* dengan metode pencarian pohon (TREE-SEARCH) akan menghasilkan solusi optimal. A* menjadi algoritma pencarian yang sangat efisien karena mampu menggabungkan informasi heuristik dengan pencarian sistematis untuk mencapai solusi optimal dalam ruang pencarian yang besar. Oleh karena itu, penting untuk memilih heuristik yang sesuai dan *admissible* dalam implementasi algoritma pencarian untuk memastikan kinerja optimal dalam menemukan solusi.

Pada kasus permainan *Word Ladder*, algoritma UCS memiliki cara yang berbeda dalam mengeksplorasi ruang pencarian dengan algoritma BFS. Walau pada dasarnya, algoritma Uniform Cost Search (UCS) dan Breadth-First Search (BFS) memiliki prinsip yang sama dalam menjelajahi graf, yaitu mereka mempertimbangkan simpul secara berurutan berdasarkan jaraknya dari simpul awal. Namun, terdapat perbedaan mendasar antara keduanya yang dapat mempengaruhi urutan node yang dibangkitkan dan jalur yang dihasilkan pada kasus seperti *Word Ladder*. Pada BFS, graf dijelajahi secara melebar, yakni semua simpul pada kedalaman tertentu dieksplorasi terlebih dahulu sebelum melanjutkan ke kedalaman berikutnya. Hal ini membuat BFS akan menemukan jalur terpendek dari simpul awal ke simpul tujuan jika jalur tersebut ada, karena BFS memeriksa semua jalur pada level yang sama sebelum beralih ke level berikutnya. Sementara itu, UCS adalah algoritma pencarian yang mempertimbangkan biaya dari simpul ke simpul lainnya. UCS akan mencari jalur dengan biaya terkecil, tanpa memperhatikan struktur level dalam graf. Oleh karena itu, UCS mungkin akan menghasilkan urutan node yang berbeda dan kadang-kadang jalur yang lebih panjang dari BFS, terutama jika ada simpul yang memiliki biaya yang rendah tetapi tidak langsung berdekatan dengan simpul awal. Jadi, meskipun kedua algoritma menggunakan prinsip yang mirip dalam menjelajahi graf, UCS dan BFS bisa menghasilkan urutan node yang berbeda dan jalur yang berbeda, terutama pada kasus seperti *Word Ladder* di mana biaya perpindahan antar simpul berbeda-beda.

Secara teoritis, implementasi algoritma A* lebih efisien dibandingkan algoritma UCS dalam kasus permainan *Word Ladder* atau masalah pencarian serupa karena A* memanfaatkan informasi tambahan melalui heuristik untuk mengarahkan pencarian ke arah yang paling menjanjikan. Dalam *Word Ladder*, heuristik umumnya adalah jarak Levenshtein antara kata saat ini dan kata tujuan, yang merupakan jumlah minimal operasi untuk mengubah satu kata menjadi kata lainnya. Dengan heuristik ini, A* dapat memilih untuk mengeksplorasi jalur yang paling menjanjikan terlebih dahulu, memperkirakan biaya terkecil untuk mencapai tujuan. Namun, keefektifan A* sangat tergantung pada pemilihan heuristik yang tepat; heuristik yang buruk dapat mengurangi efisiensinya bahkan hingga menyamai kinerja UCS. Dengan heuristik yang baik, A* dapat secara signifikan memangkas jumlah simpul yang harus dieksplorasi untuk mencapai solusi optimal, menjadikannya secara teoritis lebih efisien dari UCS.

Adapun dengan implementasi algoritma Greedy Best First Search (GBFS) dalam persoalan *Word Ladder* atau masalah pencarian serupa tidak menjamin solusi optimal. GBFS menggunakan heuristik untuk memilih simpul yang paling "dekat" dengan tujuan tanpa memperhatikan biaya aktual untuk mencapai simpul tersebut. Dalam *Word Ladder*, GBFS cenderung memilih simpul dengan estimasi heuristik terendah, yaitu simpul dengan sedikit kata yang berbeda dengan kata tujuan. Namun, ini tidak menjamin jalur yang ditemukan memiliki jumlah langkah minimum. GBFS bisa terjebak dalam "lokasi lokal" di mana simpul terdekat dengan tujuan tidak selalu membawa ke jalur terpendek secara keseluruhan. Algoritma ini hanya mempertimbangkan heuristik untuk memilih simpul berikutnya, tanpa memperhitungkan biaya total dari awal hingga simpul saat ini, sehingga tidak menjamin solusi optimal.

Source Code Program

→ Pair.java

```
import java.util.Objects;

// Kelas untuk merepresentasikan pasangan objek dengan tipe data
// berbeda, didefinisikan oleh parameter generic `<K, V>`. Setiap objek
// `Pair` memiliki `key` dan `value` untuk menyimpan informasi kunci dan
// nilainya
public class Pair<K, V> {
    private final K key;
    private final V value;

    // Konstruktor
    public Pair(K key, V val) {
        this.key = key;
        this.value = val;
    }

    public K getKey() { return key; } // method untuk akses key

    public V getValue() { return value; } // method untuk akses value

    // Fungsi `equals` untuk memeriksa kesamaan antara dua objek `Pair`,
    // membandingkan kunci dan nilai
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Pair<?, ?> pair = (Pair<?, ?>) o;
        return Objects.equals(key, pair.key) && Objects.equals(value,
pair.value);
    }
}
```

→ Astar.java

Pada file ini dilakukan Implementasi algoritma A* untuk mencari jalur optimal dari sebuah kata awal ke sebuah kata akhir dalam sebuah kumpulan kata. Di dalamnya terdapat:

- Kelas Node: Mewakili simpul dalam pencarian dengan kata, biaya sampai ke simpul tersebut, dan prioritasnya dalam antrian prioritas.
- Fungsi ``h_function``: Fungsi heuristik untuk menghitung estimasi jarak antara dua kata berdasarkan jumlah karakter yang berbeda.
- Fungsi ``nextWords``: Menghasilkan daftar kata-kata yang mungkin dijangkau dari sebuah kata dengan satu perubahan karakter.
- Metode ``optimumSolution``: Menemukan jalur optimal menggunakan algoritma A* dengan memanfaatkan antrian prioritas untuk mengeksplorasi langkah-langkah selanjutnya.

```
import java.util.*;

// Kelas Astar yang mengimplementasikan algoritma A* untuk mencari
// solusi alias jalur dari start word menuju end word
public class Astar {
    // Kelas Node merepresentasikan simpul dengan kata, biaya saat itu,
    // dan prioritasnya
    static class Node implements Comparable<Node> {
        String word; // Kata
        int cost; // Biaya total sampai node
        int priority; // Prioritas node dalam priority queue

        // Konstruktor Node
        Node(String word, int cost, int priority) {
            this.word = word;
            this.cost = cost;
            this.priority = priority;
        }

        // Metode membandingkan prioritas node (digunakan untuk priority
```

```

queue)

@Override
public int compareTo(Node other) {
    return Integer.compare(this.priority, other.priority);
}

}

// Fungsi heuristik yang menghitung estimasi jarak antara dua kata
// berdasarkan jumlah karakter yang berbeda
private static int h_function(String now, String end) {
    int count = 0;
    for (int i = 0; i < now.length(); i++) {
        if (now.charAt(i) != end.charAt(i)) {
            count++;
        }
    }
    return count;
}

// Fungsi menghasilkan calon daftar kata yang mungkin menjadi solusi
// yakni yang memiliki perbedaan satu huruf
private static List<String> nextWords(String word, Set<String>
wordSet) {
    List<String> nextWordsList = new ArrayList<>();
    char[] chars = word.toCharArray();
    for (int i = 0; i < word.length(); i++) {
        char charNow = chars[i];
        for (char c = 'a'; c <= 'z'; c++) {
            if (c != charNow) {
                chars[i] = c;
                String newWord = new String(chars);
                if (wordSet.contains(newWord)) {
                    nextWordsList.add(newWord);
                }
            }
        }
        chars[i] = charNow;
    }
}

```

```

        return nextWordsList;
    }

    // Fungsi untuk menemukan jalur optimal dari kata awal ke kata akhir
    menggunakan algoritma A*.
    public static Pair<List<String>, Integer> optimumSolution(String
start, String end, Set<String> wordSet) {
        Map<String, String> wordBefore = new HashMap<>();
        PriorityQueue<Pair<String, Integer>> priorityQueue = new
PriorityQueue<>(Comparator.comparingInt(pair -> pair.getValue() +
h_function(pair.getKey(), end)));
        Set<String> visited = new HashSet<>();
        int visitedNodes = 0;

        priorityQueue.add(new Pair<>(start, 0));
        while (!priorityQueue.isEmpty()) {
            Pair<String, Integer> wordPair = priorityQueue.poll();
            String wordNow = wordPair.getKey();
            int currentCost = wordPair.getValue();
            visitedNodes++;

            if (wordNow.equals(end)) {
                List<String> path = new ArrayList<>();
                String word = end;
                while (word != null) {
                    path.add(word);
                    word = wordBefore.get(word);
                }
                Collections.reverse(path);
                return new Pair<>(path, visitedNodes);
            }

            visited.add(wordNow);
            for (String nextWord : nextWords(wordNow, wordSet)) {
                int newCost = currentCost + h_function(wordNow,
nextWord);
                if (!visited.contains(nextWord) &&
(!wordBefore.containsKey(nextWord) || newCost < currentCost)) {

```

```

        priorityQueue.add(new Pair<>(nextWord, newCost));
        wordBefore.put(nextWord, wordNow);
    }
}
}
return new Pair<>(null, visitedNodes);
}
}

```

→ GBFS.java

Kelas GBFS adalah implementasi algoritma *Greedy Best-First Search* (GBFS) yang digunakan untuk mencari solusi dari sebuah kata awal ke sebuah kata akhir dalam sebuah kumpulan kata. Berikut penjelasan singkat dari kelas dan metodenya:

- Fungsi ``h_function``: Fungsi heuristik untuk menghitung estimasi jarak antara dua kata berdasarkan jumlah karakter yang berbeda. Fungsi ini membantu algoritma GBFS untuk memilih simpul berikutnya yang paling menjanjikan.
- Fungsi ``nextWords``: Fungsi untuk menghasilkan daftar kata-kata yang mungkin menjadi lanjutan solusi dari sebuah kata dengan perbedaan satu huruf alias untuk menemukan kemungkinan langkah selanjutnya dalam pencarian.
- Metode ``solution``: Metode utama untuk menemukan solusi menggunakan algoritma GBFS. Metode ini menggunakan *priority queue* untuk mengeksplorasi langkah-langkah selanjutnya berdasarkan heuristik, dengan tujuan mencapai kata akhir.

```

import java.util.*;

public class GBFS {
    private static int h_fuction(String current, String end) {
        int count = 0;
        for (int i = 0; i < current.length(); i++) {

```

```

        if (current.charAt(i) != end.charAt(i)) {
            count++;
        }
    }
    return count;
}

private static List<String> nextWords(String word, Set<String>
wordSet) {
    List<String> nextWordsList = new ArrayList<>();
    char[] chars = word.toCharArray();
    for (int i = 0; i < word.length(); i++) {
        char charNow = chars[i];
        for (char c = 'a'; c <= 'z'; c++) {
            if (c != charNow) {
                chars[i] = c;
                String newWord = new String(chars);
                if (wordSet.contains(newWord)) {
                    nextWordsList.add(newWord);
                }
            }
        }
        chars[i] = charNow;
    }
    return nextWordsList;
}

public static Pair<List<String>, Integer> solution(String start,
String end, Set<String> wordSet) {
    Map<String, String> wordBefore = new HashMap<>();
    PriorityQueue<String> priorityQueue = new
PriorityQueue<>(Comparator.comparingInt(word -> h_fuction(word, end)));
    Set<String> visited = new HashSet<>();
    int visitedNodes = 0;
    priorityQueue.add(start);

    while (!priorityQueue.isEmpty()) {
        String wordNow = priorityQueue.poll();
    }
}

```

```

        visitedNodes++;

        if (wordNow.equals(end)) {
            List<String> path = new ArrayList<>();
            String word = end;
            while (word != null) {
                path.add(word);
                word = wordBefore.get(word);
            }
            Collections.reverse(path);
            return new Pair<>(path, visitedNodes);
        }

        visited.add(wordNow);
        for (String nextWord : nextWords(wordNow, wordSet)) {
            if (!visited.contains(nextWord)) {
                if (!wordBefore.containsKey(nextWord)) {
                    priorityQueue.add(nextWord);
                    wordBefore.put(nextWord, wordNow);
                }
            }
        }
    }
    return new Pair<>(null, visitedNodes);
}
}

```

→ UCS.java

Kelas UCS merupakan implementasi algoritma Uniform Cost Search (UCS) untuk mencari solusi dari sebuah kata awal ke sebuah kata akhir dalam sebuah kumpulan kata. Berikut penjelasan singkat dari kelas dan metodenya:

- Kelas Node: Representasi simpul dalam pencarian, menyimpan kata dan biaya yang diperlukan untuk mencapai simpul tersebut.
- Fungsi `nextWords`: Menghasilkan daftar kata-kata yang mungkin dijangkau dari sebuah kata dengan satu perubahan karakter.

- Metode `optimumSolution`: Metode utama yang menggunakan antrian prioritas untuk mengeksplorasi langkah-langkah selanjutnya berdasarkan biaya terendah, dengan tujuan mencapai kata akhir.

```
import java.util.*;

public class UCS {
    static class Node implements Comparable<Node> {
        String word;
        int cost;

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

        @Override
        public int compareTo(Node other) {return
Integer.compare(this.cost, other.cost);}
    }

    private static List<String> nextWords(String word, Set<String>
wordSet) {
        List<String> nextWordsList = new ArrayList<>();
        char[] wordChars = word.toCharArray();
        for (int i = 0; i < wordChars.length; i++) {
            char charNow = wordChars[i];
            for (char c = 'a'; c <= 'z'; c++) {
                if (c != charNow) {
                    wordChars[i] = c;
                    String newWord = new String(wordChars);
                    if (wordSet.contains(newWord)) {
                        nextWordsList.add(newWord);
                    }
                }
            }
            wordChars[i] = charNow;
        }
        return nextWordsList;
    }
}
```



```

    }

    public static Pair<List<String>, Integer> optimumSolution(String
start, String end, Set<String> wordSet) {
        Map<String, String> wordBefore = new HashMap<>();
        PriorityQueue<Node> priorityQueue = new PriorityQueue<>();
        Set<String> visited = new HashSet<>();
        priorityQueue.add(new Node(start, 0));
        int visitedNodes = 0;

        while (!priorityQueue.isEmpty()) {
            Node lowestCost = priorityQueue.poll();
            String wordNow = lowestCost.word;
            int currentCost = lowestCost.cost;
            visitedNodes++;

            if (wordNow.equals(end)) {
                List<String> path = new ArrayList<>();
                String word = end;
                while (word != null) {
                    path.add(word);
                    word = wordBefore.get(word);
                }
                Collections.reverse(path);
                return new Pair<>(path, visitedNodes);
            }

            visited.add(wordNow);
            for (String newWord : nextWords(wordNow, wordSet)) {
                if (!visited.contains(newWord)) {
                    int newCost = currentCost + 1;
                    if (!wordBefore.containsKey(newWord) || newCost <
lowestCost.cost) {
                        priorityQueue.add(new Node(newWord, newCost));
                        wordBefore.put(newWord, wordNow);
                    }
                }
            }
        }
    }

```

```

    }

    return new Pair<>(null, visitedNodes);
}
}

```

→ WordLadderGame.java

Kelas WordLadderGame adalah sebuah program sederhana untuk memainkan permainan Word Ladder, yang bertujuan untuk menemukan jalur perubahan satu huruf dari sebuah kata awal ke sebuah kata akhir, dengan memanfaatkan berbagai algoritma pencarian seperti UCS, GBFS, dan A*. Penjelasan singkat dari kelas dan metodenya:

- Metode readWordsFromFile: Membaca kata-kata dari sebuah file dan menyimpannya ke dalam sebuah set, dengan menghapus spasi dan mengubah huruf menjadi lowercase.
- Metode isValidWord: Memeriksa apakah sebuah kata valid, yaitu hanya terdiri dari huruf-huruf alfabet.
- Metode main: Metode utama yang menjalankan permainan Word Ladder.
 - Meminta input dari pengguna untuk kata awal dan kata akhir beserta memilih algoritma pencarian.
 - Menjalankan algoritma yang dipilih untuk menemukan jalur Word Ladder dan menampilkan hasil jalur yang ditemukan beserta jumlah simpul yang dikunjungi dan waktu eksekusi.
- Metode printPath: Mencetak jalur yang ditemukan dalam permainan Word Ladder.
- Metode printExecutionTimeAndPathLength: Mencetak panjang jalur dan waktu eksekusi yang diperlukan untuk menemukan jalur tersebut.

```

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

```

```

import java.util.*;

public class WordLadderGame {
    public static Set<String> readWordsFromFile(String fileName) throws
IOException {
        Set<String> words = new HashSet<>();
        BufferedReader reader = new BufferedReader(new
FileReader(fileName));
        String word;
        while ((word = reader.readLine()) != null) {
            words.add(word.trim().toLowerCase());
        }
        reader.close();
        return words;
    }

    public static boolean isValidWord(String word) {
        return word.matches("[a-zA-Z]+");
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try {
            Set<String> wordSet = readWordsFromFile("words.txt");
            System.out.print("WELCOME TO THE WORD LADDER GAME!\n");

            String start, end;
            int choice;

            do {
                System.out.print("Enter start word: ");
                start = scanner.nextLine().trim().toLowerCase();
                if (!isValidWord(start)) {
                    System.out.println("Warning: Start word should
contain only alphabets.");
                    continue;
                }
            }

```

```

        if (!wordSet.contains(start)) {
            System.out.println("Warning: Start word does not
exist in the word list.");
            continue;
        }
        break;
    } while (true);

    do {
        System.out.print("Enter end word: ");
        end = scanner.nextLine().trim().toLowerCase();
        if (!isValidWord(end)) {
            System.out.println("Warning: End word should contain
only alphabets.");
            continue;
        }
        if (!wordSet.contains(end)) {
            System.out.println("Warning: End word does not exist
in the word list.");
            continue;
        }
        break;
    } while (true);

    do {
        System.out.println("Choose Algorithm:");
        System.out.println("1. UCS");
        System.out.println("2. GBFS");
        System.out.println("3. A*");
        System.out.print("Enter your choice: ");
        choice = scanner.nextInt();
        scanner.nextLine();

        if (choice < 1 || choice > 3) {
            System.out.println("Invalid choice. Please enter a
number between 1 and 3.");
            continue;
        }
    }

```

```

        break;
    } while (true);

    long startTime, endTime;
    Pair<List<String>, Integer> result;

    switch (choice) {
        case 1:
            System.out.println("UCS:");
            startTime = System.currentTimeMillis();
            result = UCS.optimumSolution(start, end, wordSet);
            endTime = System.currentTimeMillis();
            System.out.println("Visited nodes count: " +
result.getValue());
            printPath(result.getKey());
            printExecutionTimeAndPathLength(startTime, endTime,
result.getKey());
            break;
        case 2:
            System.out.println("Greedy Best First Search:");
            startTime = System.currentTimeMillis();
            result = GBFS.solution(start, end, wordSet);
            endTime = System.currentTimeMillis();
            System.out.println("Visited nodes count: " +
result.getValue());
            printPath(result.getKey());
            printExecutionTimeAndPathLength(startTime, endTime,
result.getKey());
            break;
        case 3:
            System.out.println("A*");
            startTime = System.currentTimeMillis();
            result = Astar.optimumSolution(start, end, wordSet);
            endTime = System.currentTimeMillis();
            System.out.println("Visited nodes count: " +
result.getValue());
            printPath(result.getKey());
            printExecutionTimeAndPathLength(startTime, endTime,

```

```

result.getKey());
            break;
        default:
            System.out.println("Invalid choice.");
        }

    } catch (IOException e) {
        System.out.println("Error reading words from file: " +
e.getMessage());
    } finally {
        scanner.close();
    }
}

private static void printPath(List<String> path) {
    if (path != null) {
        for (String word : path) {
            System.out.print(word + " ");
        }
        System.out.println();
    } else {
        System.out.println("No path found");
    }
    System.out.println();
}

private static void printExecutionTimeAndPathLength(long startTime,
long endTime, List<String> path) {
    long execution = endTime - startTime;
    if (path != null) {
        System.out.println("Path length: " + path.size());
    } else {
        System.out.println("No path found");
    }
    System.out.println("Execution time: " + execution + " ms\n");
}
}

```

Tangkapan Layar *Input* dan *Output*

1. Test case 1: frown → smile

Algoritma Greedy Best First Search (GBFS)

Word Ladder Game

Start Word: frownEnd Word: smile

UCSGBFSA*

Algorithm: GBFSPath Length: 14

Visited Nodes: 24Execution Time: 0 ms

F	R	O	W	N
B	R	O	W	N
B	R	O	W	S
B	R	O	O	S
B	R	I	O	S
T	R	I	O	S
T	R	I	P	S
T	R	I	P	E
T	R	I	N	E
T	H	I	N	E
S	H	I	N	E
S	P	I	N	E
S	P	I	L	E
S	M	I	L	E

Algoritma UCS

Word Ladder Game

Start Word:

frown

End Word:

smile

UCS

GBFS

A*

Algorithm: UCS

Path Length: 10

Visited Nodes: 4347

Execution Time: 49 ms

F	R	O	W	N
F	R	O	W	S
F	L	O	W	S
S	L	O	W	S
S	L	O	T	S
S	P	O	T	S
S	P	I	T	S
S	P	I	T	E
S	M	I	T	E
S	M	I	L	E

Algoritma A*

Word Ladder Game

Start Word:

jean

End Word:

gene

UCS

GBFS

A*

Algorithm: A*

Path Length: 6

Visited Nodes: 32

Execution Time: 0 ms

J	E	A	N
P	E	A	N
P	E	A	T
P	E	N	T
G	E	N	T
G	E	N	E

2. Test case 1: jean → gene

Algoritma Greedy Best First Search (GBFS)

Word Ladder Game

Start Word:

jean

End Word:

gene

UCS

GBFS

A*

Algorithm : GBFS

Path Length: 8

Visited Nodes: 11

Execution Time: 0 ms

J	E	A	N
B	E	A	N
B	E	A	U
B	E	D	U
B	E	D	S
G	E	D	S
G	E	N	S
G	E	N	E

Algoritma UCS

Word Ladder Game

Start Word:

jean

End Word:

gene

UCS

GBFS

A*

Algorithm : UCS

Path Length: 6

Visited Nodes: 2080

Execution Time: 28 ms

J	E	A	N
B	E	A	N
B	E	A	T
B	E	N	T
B	E	N	E
G	E	N	E

Algoritma A*

Word Ladder Game

Start Word:

jean

End Word:

gene

UCS

GBFS

A*

Algorithm: A*

Path Length: 6

Visited Nodes: 32

Execution Time: 1 ms

J	E	A	N
P	E	A	N
P	E	A	T
P	E	N	T
G	E	N	T
G	E	N	E

3. Test case 3: veep → exec

Algoritma Greedy Best First Search (GBFS)

Word Ladder Game

Start Word:

veep

End Word:

exec

UCS

GBFS

A*

Algorithm: GBFS

Path Length: 7

Visited Nodes: 49

Execution Time: 0 ms

V	E	E	P
V	E	E	S
L	E	E	S
L	Y	E	S
E	Y	E	S
E	X	E	S
E	X	E	C

Algoritma UCS

Word Ladder Game

Start Word:

veep

End Word:

exec

UCS

GBFS

A*

Algorithm : UCS

Path Length: 7

Visited Nodes: 2651

Execution Time: 30 ms

V	E	E	P
V	E	E	S
P	E	E	S
P	Y	E	S
E	Y	E	S
E	X	E	S
E	X	E	C

Algoritma A*

Word Ladder Game

Start Word:

veep

End Word:

exec

UCS

GBFS

A*

Algorithm : A*

Path Length: 7

Visited Nodes: 166

Execution Time: 2 ms

V	E	E	P
V	E	E	S
R	E	E	S
R	Y	E	S
E	Y	E	S
E	X	E	S
E	X	E	C

4. Test case 4: strong → dreams

Algoritma Greedy Best First Search (GBFS)

Word Ladder Game

Start Word:

strong

End Word:

dreams

UCS

GBFS

A*

Algorithm: GBFS

Path Length: 41

Visited Nodes: 373

Execution Time: 7 ms

S	T	R	O	N	G
S	A	R	O	N	G
B	A	R	O	N	G
B	A	R	O	N	S
B	A	C	O	N	S
R	A	C	O	N	S
R	E	C	O	N	S
R	E	D	O	N	S
R	E	D	A	N	S
D	E	D	A	N	S
D	E	W	A	N	S
S	E	W	A	N	S
S	O	W	A	N	S
R	O	W	A	N	S
R	O	W	E	N	S
R	O	W	E	R	S
D	O	W	E	R	S
D	O	T	E	R	S
D	A	T	E	R	S
T	A	T	E	R	S
T	A	T	A	R	S
T	A	L	A	R	S

Word Ladder Game

Start Word:

strong

End Word:

dreams

UCS

GBFS

A*

Algorithm: GBFS

Path Length: 41

Visited Nodes: 373

Execution Time: 7 ms

T	A	T	E	R	S
T	A	T	A	R	S
T	A	L	A	R	S
M	A	L	A	R	S
M	O	L	A	R	S
P	O	L	A	R	S
P	O	L	E	R	S
P	O	K	E	R	S
T	O	K	E	R	S
T	O	Y	E	R	S
T	W	Y	E	R	S
T	W	I	E	R	S
T	R	I	E	R	S
T	R	I	E	N	S
T	R	E	E	N	S
G	R	E	E	N	S
G	R	E	E	T	S
G	R	E	A	T	S
T	R	E	A	T	S
T	R	E	A	D	S
D	R	E	A	D	S
D	R	E	A	M	S

Algoritma UCS

Word Ladder Game

Start Word:

strong

End Word:

dreams

UCS

GBFS

A*

Algorithm : UCS

Path Length: 17

Visited Nodes: 7555

Execution Time: 84 ms

S	T	R	O	N	G
S	T	R	I	N	G
S	E	R	I	N	G
S	E	R	I	N	S
S	E	R	I	E	S
S	C	R	I	E	S
S	C	R	I	P	S
S	T	R	I	P	S
S	T	R	E	P	S
S	T	E	E	P	S
S	L	E	E	P	S
S	L	E	E	K	S
C	L	E	E	K	S
C	R	E	E	K	S
C	R	E	A	K	S
C	R	E	A	M	S
D	R	E	A	M	S

Algoritma A*

Word Ladder Game

Start Word:

strong

End Word:

dreams

UCS

GBFS

A*

Algorithm: A*

Path Length: 17

Visited Nodes: 2327

Execution Time: 31 ms

S	T	R	O	N	G
S	T	R	I	N	G
S	E	R	I	N	G
S	E	R	I	N	S
S	E	R	I	E	S
S	C	R	I	E	S
S	C	R	I	P	S
S	T	R	I	P	S
S	T	R	E	P	S
S	T	E	E	P	S
S	L	E	E	P	S
S	L	E	E	K	S
C	L	E	E	K	S
C	R	E	E	K	S
C	R	E	A	K	S
C	R	E	A	M	S
D	R	E	A	M	S

5. Test case 5: grass → roots

Algoritma Greedy Best First Search (GBFS)

Word Ladder Game

Start Word:

grass

End Word:

roots

UCS

GBFS

A*

Algorithm : GBFS

Path Length: 6

Visited Nodes: 6

Execution Time: 1 ms

G	R	A	S	S
G	R	O	S	S
G	R	O	T	S
T	R	O	T	S
T	O	O	T	S
R	O	O	T	S

Algoritma UCS

Word Ladder Game

Start Word:

grass

End Word:

roots

UCS

GBFS

A*

Algorithm : UCS

Path Length: 6

Visited Nodes: 604

Execution Time: 8 ms

G	R	A	S	S
G	R	A	D	S
G	O	A	D	S
R	O	A	D	S
R	O	O	D	S
R	O	O	T	S

Algoritma A*

Word Ladder Game

Start Word:

grass

End Word:

roots

UCS

GBFS

A*

Algorithm: A*

Path Length: 6

Visited Nodes: 25

Execution Time: 1 ms

G	R	A	S	S
B	R	A	S	S
B	R	A	T	S
B	O	A	T	S
B	O	O	T	S
R	O	O	T	S

6. Test case 6: pray → amen

Algoritma Greedy Best First Search (GBFS)

Word Ladder Game

Start Word:

pray

End Word:

amen

UCS

GBFS

A*

Algorithm: GBFS

Path Length: 13

Visited Nodes: 49

Execution Time: 6 ms

P	R	A	Y
P	R	E	Y
G	R	E	Y
G	L	E	Y
G	L	E	E
A	L	E	E
A	G	E	E
A	G	E	S
A	X	E	S
A	X	I	S
A	M	I	S
A	M	I	N
A	M	E	N

Algoritma UCS

Word Ladder Game

Start Word:

pray

End Word:

amen

UCS

GBFS

A*

Algorithm: UCS

Path Length: 8

Visited Nodes: 3610

Execution Time: 46 ms

P	R	A	Y
G	R	A	Y
G	R	A	D
G	R	I	D
A	R	I	D
A	M	I	D
A	M	I	N
A	M	E	N

Algoritma A*

Word Ladder Game

Start Word:

pray

End Word:

amen

UCS

GBFS

A*

Algorithm: A*

Path Length: 9

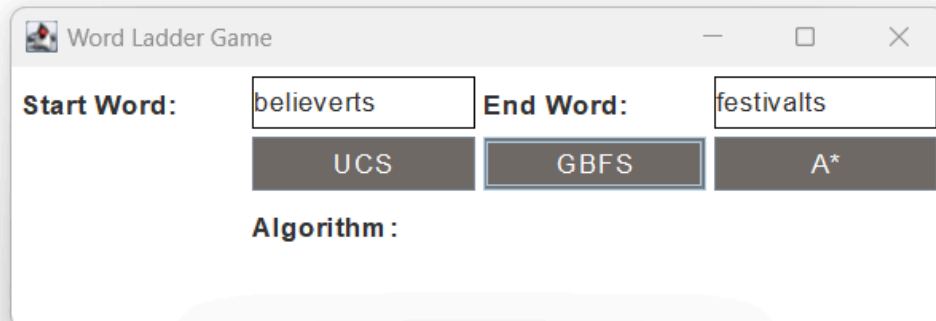
Visited Nodes: 442

Execution Time: 6 ms

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

7. Test case 7: edge cases

word doesn't exist

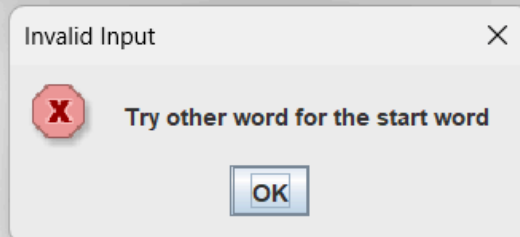


Word Ladder Game


Start Word: End Word:

UCS GBFS A*

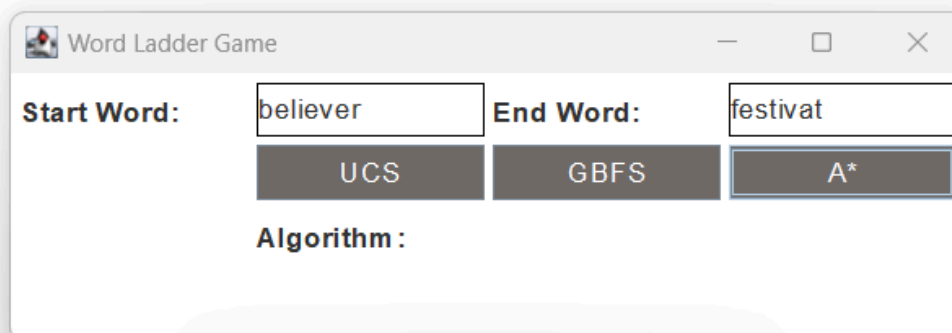
Algorithm :



Invalid Input

 Try other word for the start word

OK

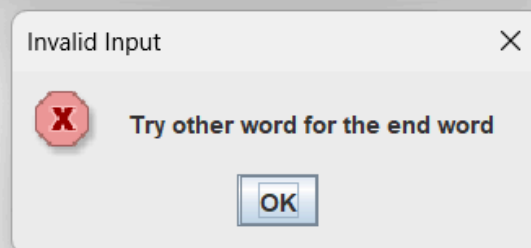


Word Ladder Game


Start Word: End Word:

UCS GBFS A*

Algorithm :



Invalid Input

 Try other word for the end word

OK


different length

Word Ladder Game

Start Word: End Word:

Algorithm :

Invalid Input

 Make sure the start and end words have the same length


word contain other than alphabet

Word Ladder Game

Start Word: End Word:

Algorithm :

Invalid Input

 Please input alphabetic words only


Path not found (believes → festival)

Word Ladder Game

Start Word: End Word:

Algorithm:

Not Found

 No path found

Hasil Analisis Perbandingan Solusi UCS, Greedy Best First Search, dan A*

Perbandingan Solusi dari menggunakan algoritma UCS, Greedy Best First Search, dan A* dapat dilihat dari waktu eksekusi, panjang jalur, dan jumlah node yang dikunjungi. Dengan bukti tertera pada tangkapan layar hasil uji coba pada bab sebelumnya.

Kompleksitas waktu dari algoritma *Uniform Cost Search* (UCS) tergantung pada struktur graf yang dihadapi. Secara umum, kompleksitasnya adalah $O(n^{(d+1)})$, di mana 'n' adalah faktor branching (jumlah maksimum pilihan setiap simpul) dan 'd' adalah kedalaman solusi terpendek dari simpul awal. *Greedy Best First Search* memiliki kompleksitas waktu yang tergantung pada kualitas heuristik yang digunakan. Dengan heuristik yang sesuai, kompleksitasnya bisa menjadi sangat baik, sekitar $O(n^d)$ di mana 'd' adalah kedalaman solusi terpendek. Sementara itu, A* memiliki kompleksitas waktu $O(n^m)$ dengan asumsi fungsi heuristiknya konsisten, yang memprediksi biaya sisa menuju tujuan. Dalam hal ini, 'm' adalah kedalaman solusi terpendek. Dengan demikian, meskipun UCS memiliki kompleksitas yang tinggi, A* sering kali menjadi pilihan yang efisien karena menggabungkan keunggulan UCS dan Greedy Best First Search. Oleh karena itu dapat dilihat bahwa hasil waktu eksekusi dari algoritma A* selalu lebih cepat daripada algoritma UCS. Apabila algoritma A* dibandingkan dengan GBFS, dari segi waktu GBFS seringkali lebih cepat dibandingkan A* karena dia tidak mempertimbangkan jalur-jalur lain.

Kompleksitas ruang dari algoritma *Uniform Cost Search* (UCS) sbergantung pada jumlah simpul yang dibangkitkan dan disimpan dalam memori. Dalam kasus terburuk, kompleksitas ruang UCS adalah $O(n^d)$, di mana 'n' adalah faktor branching (jumlah maksimum pilihan setiap simpul) dan 'd' adalah kedalaman solusi terpendek dari simpul awal. Greedy Best First Search juga memiliki kompleksitas ruang tergantung pada faktor pengurangan simpul, dengan kasus terburuk yang juga mencapai $O(n^d)$. Sementara itu, A* memiliki kompleksitas ruang terburuk yang sama dengan $O(n^d)$, tetapi dalam praktiknya, seringkali lebih efisien karena penyesuaian heuristik yang dilakukan dan penggunaan memori yang lebih cerdas. Dengan demikian, meskipun ketiganya memiliki kompleksitas ruang yang sama dalam kasus terburuk, A* dapat

menjadi pilihan yang lebih baik dalam aplikasi praktis karena peningkatan efisiensi yang diberikan oleh penggunaan heuristik dan manajemen memori yang lebih bijaksana.

Dari segi optimalitas terdapat perbedaan dalam pendekatan mereka terhadap penentuan solusi yang optimal. UCS, dengan pendekatannya yang sistematis, menjamin solusi optimal dengan memprioritaskan jalur dengan biaya terendah. Dalam hal ini, UCS adalah pilihan yang dapat diandalkan untuk masalah di mana mencari solusi dengan biaya minimal adalah prioritas utama. Dapat dilihat pada test case ke-6 yaitu dari kata *pray* ke *amen*. Di sisi lain, Greedy Best First Search, dengan pendekatannya yang bersifat serakah, tidak menjamin optimalitas karena hanya mempertimbangkan biaya langkah berikutnya tanpa memperhatikan biaya total jalur. Ini bisa mengakibatkan solusi yang ditemukan tidak mencapai biaya minimal. Hal ini bisa dilihat dari setiap dilakukan uji coba algoritma ini memberikan jalur terpanjang dibandingkan ketiganya. Sedangkan A*, algoritma yang menggabungkan elemen dari UCS dan Greedy BFS, menawarkan optimalitas saat fungsi heuristiknya konsisten. Dengan memanfaatkan heuristik untuk memprediksi biaya sisa menuju tujuan, A* dapat menemukan solusi optimal dengan efisien, menjadikannya pilihan yang sangat baik untuk berbagai jenis masalah, tetapi pada program ini terdapat kasus di mana algoritma ini memberikan jalur yang lebih panjang tetapi tidak jauh dari algoritma UCS.

Dari pemaparan dan hasil uji coba di atas, dapat disimpulkan apabila dilihat dari segi optimal, algoritma UCS selalu memberikan hasil yang paling optimal dibandingkan algoritma yang lain. Adapun untuk segi waktu eksekusi GBFS selalu lebih cepat dari yang lainnya. Adapun untuk yang paling efisien adalah algoritma A*.

Implementasi Bonus

Bonus pada tugas kecil kali ini ialah membuat tampilan grafis yakni GUI untuk permainan *Word Ladder*. Dengan mempertimbangkan waktu pengerjaan tugas dan berbagai hal lainnya, pada tugas kecil ini GUI menggunakan JavaSwing dikarenakan kemudahan dan simplisitas yang dimilikinya. GUI ini terletak pada file WordLadderGUI.java yang akan dijelaskan sebagai berikut.

→ WordLadderGUI.java

Kelas WordLadderGUI adalah antarmuka pengguna grafis (GUI) untuk permainan Word Ladder. Ini memungkinkan pengguna untuk memasukkan kata awal dan akhir, memilih algoritma pencarian, dan menampilkan jalur yang dihasilkan bersama dengan statistik terkait. Penjelasan singkat dari kelas dan metodenya:

- **Konstruktor:** Membuat antarmuka pengguna dengan menginisialisasi elemen-elemen seperti teks field, tombol, dan panel, serta menambahkan aksi pemantau ke tombol.
- **Metode wordExists:** Memeriksa apakah sebuah kata ada dalam file teks dengan membandingkan setiap baris dalam file dengan kata yang diberikan.
- **Metode createTextField:** Membuat dan mengkonfigurasi teks field untuk input.
- **Metode actionPerformed:** Menangani aksi yang dilakukan oleh pengguna seperti klik tombol. Memvalidasi input, menjalankan algoritma pencarian yang dipilih, dan menampilkan hasilnya.
- **Metode displayPath:** Menampilkan jalur yang dihasilkan, algoritma yang digunakan, jumlah simpul yang dikunjungi, dan waktu eksekusi.
- **Metode createWordPanel:** Membuat panel untuk menampilkan kata dalam jalur, dengan menyesuaikan warna dan tampilan sesuai dengan kata akhir.
- **Metode createLetterPanel:** Membuat panel untuk menampilkan setiap huruf dalam kata, dengan menyesuaikan warna dan tampilan jika huruf tersebut merupakan bagian dari kata akhir.
- **Metode main:** Memulai aplikasi GUI dengan membaca kata-kata dari file teks dan membuat objek WordLadderGUI.

```
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.event.*;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.List;
import java.util.Set;

public class WordLadderGUI extends JFrame implements ActionListener {
    private JTextField startField, endField;
    private JButton ucsButton, gbfsButton, aStarButton;
    private JPanel resultPanel;
    private JLabel algorithmLabel;
    private JLabel visitedNodesLabel;
    private JLabel pathLabel;
    private JLabel timeLabel;
    private Set<String> wordSet;

    public WordLadderGUI(Set<String> wordSet) {
        super("Word Ladder Game");
        this.wordSet = wordSet;

        startField = createTextField();
        endField = createTextField();
        ucsButton = new JButton("UCS");
        gbfsButton = new JButton("GBFS");
        aStarButton = new JButton("A*");
        resultPanel = new JPanel(new GridLayout(0, 1));

        Font font = new Font("Arial", Font.PLAIN, 14);
        startField.setFont(font);
        endField.setFont(font);
        ucsButton.setFont(font);
        gbfsButton.setFont(font);
        aStarButton.setFont(font);
    }
}
```



```
ucsButton.setBackground(new Color(110, 105, 103));
gbfsButton.setBackground(new Color(110, 105, 103));
aStarButton.setBackground(new Color(110, 105, 103));
ucsButton.setForeground(Color.WHITE);
gbfsButton.setForeground(Color.WHITE);
aStarButton.setForeground(Color.WHITE);

Font labelFont = new Font("Monospace", Font.BOLD, 14);

JLabel startLabel = new JLabel("Start Word: ");
startLabel.setFont(labelFont);

JLabel endLabel = new JLabel("End Word: ");
endLabel.setFont(labelFont);

algorithmLabel = new JLabel("Algorithm: ");
algorithmLabel.setFont(labelFont);
algorithmLabel.setMaximumSize(new Dimension(10, 5));

pathLabel = new JLabel();
pathLabel.setFont(labelFont);
pathLabel.setMaximumSize(new Dimension(10, 5));

visitedNodesLabel = new JLabel();
visitedNodesLabel.setFont(labelFont);
visitedNodesLabel.setMaximumSize(new Dimension(10, 5));

timeLabel = new JLabel();
timeLabel.setFont(labelFont);
timeLabel.setMaximumSize(new Dimension(10, 5));

JPanel inputPanel = new JPanel(new GridLayout(4, 4, 4, 4));
inputPanel.setBorder(new EmptyBorder(5, 5, 5, 5));
inputPanel.setBackground(Color.WHITE);
inputPanel.add(startLabel);
inputPanel.add(startField);
inputPanel.add(endLabel);
inputPanel.add(endField);
```

```

        inputPanel.add(new JLabel());
        inputPanel.add(ucsButton);
        inputPanel.add(gbfsButton);
        inputPanel.add(aStarButton);

        ucsButton.addActionListener(this);
        gbfsButton.addActionListener(this);
        aStarButton.addActionListener(this);

        ucsButton.setAlignmentX(Component.CENTER_ALIGNMENT);
        gbfsButton.setAlignmentX(Component.CENTER_ALIGNMENT);
        aStarButton.setAlignmentX(Component.CENTER_ALIGNMENT);

        inputPanel.add(new JLabel());
        inputPanel.add(algorithmLabel);
        inputPanel.add(new JLabel());
        inputPanel.add(pathLabel);
        inputPanel.add(new JLabel());
        inputPanel.add(visitedNodesLabel);
        inputPanel.add(new JLabel());
        inputPanel.add(timeLabel);

        getContentPane().setLayout(new BorderLayout());
        getContentPane().add(inputPanel, BorderLayout.NORTH);
        getContentPane().add(new JScrollPane(resultPanel),
BorderLayout.CENTER);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }

    public static boolean wordExists(String filePath, String word) {
        try (BufferedReader br = new BufferedReader(new
FileReader(filePath))) {
            String line;
            while ((line = br.readLine()) != null) {

```

```

        if (line.trim().equalsIgnoreCase(word)) {
            return true;
        }
    }
} catch (IOException e) {
    e.printStackTrace();
}
return false;
}

private JTextField createTextField() {
    JTextField textField = new JTextField(10);
    textField.setBorder(new LineBorder(Color.BLACK));
    return textField;
}

@Override
public void actionPerformed(ActionEvent e) {
    String start = startField.getText().trim().toLowerCase();
    String end = endField.getText().trim().toLowerCase();

    if (start.length() != end.length()) {
        JOptionPane.showMessageDialog(this, "Make sure the start and
end words have the same length", "Invalid Input",
JOptionPane.ERROR_MESSAGE);
        return;
    }

    if (!start.matches("[a-zA-Z]+") || !end.matches("[a-zA-Z]+")) {
        JOptionPane.showMessageDialog(this, "Please input alphabetic
words only", "Invalid Input", JOptionPane.ERROR_MESSAGE);
        return;
    }

    if (!wordExists("words.txt", start)) {
        JOptionPane.showMessageDialog(this, "Try other word for the
start word", "Invalid Input", JOptionPane.ERROR_MESSAGE);
        return;
    }
}

```

```

        if (!wordExists("words.txt", end)) {
            JOptionPane.showMessageDialog(this, "Try other word for the
end word", "Invalid Input", JOptionPane.ERROR_MESSAGE);
            return;
        }

        Pair<List<String>, Integer> resultPair;
        String endWord = "";
        String algorithm = "";
        int visitedNodes = 0;
        long startTime = System.currentTimeMillis();

        try {
            if (e.getSource() == ucsButton) {
                resultPair = UCS.optimumSolution(start, end, wordSet);
                algorithm = "UCS";
            } else if (e.getSource() == gbfsButton) {
                resultPair = GBFS.solution(start, end, wordSet);
                algorithm = "GBFS";
            } else {
                resultPair = Astar.optimumSolution(start, end, wordSet);
                algorithm = "A*";
            }

            long endTime = System.currentTimeMillis();

            List<String> path = resultPair.getKey();
            visitedNodes = resultPair.getValue();

            if (path == null || path.isEmpty()) {
                JOptionPane.showMessageDialog(this, "No path found",
"Not Found", JOptionPane.INFORMATION_MESSAGE);
                return;
            }

            endWord = endField.getText().trim().toLowerCase();

```

```

        displayPath(path, endWord, algorithm, visitedNodes, endTime
- startTime);
    } catch (Exception ex) {
        JOptionPane.showMessageDialog(this, "Error: " +
ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
    }
}

private void displayPath(List<String> path, String endWord, String
algorithm, int visitedNodes, long elapsedTime) {
    resultPanel.removeAll();
    if (path != null) {
        for (String word : path) {
            JPanel wordPanel = createWordPanel(word.toUpperCase(),
endWord.toUpperCase());
            resultPanel.add(wordPanel);
        }
        algorithmLabel.setText("Algorithm: " + algorithm);
        timeLabel.setText("Execution Time: " + elapsedTime + " ms");
        pathLabel.setText("Path Length: " + path.size());
        visitedNodesLabel.setText("Visited Nodes: " + visitedNodes);
    } else {
        resultPanel.add(new JLabel("No path found"));
        timeLabel.setText("Execution Time: " + elapsedTime + " ms");
        visitedNodesLabel.setText("Visited Nodes: " + visitedNodes);
        algorithmLabel.setText("Algorithm: N/A");
    }
    resultPanel.revalidate();
    resultPanel.repaint();
    pack();
}

private JPanel createWordPanel(String word, String endWord) {
    word = word.toUpperCase();
    JPanel wordPanel = new JPanel(new GridLayout(1, word.length(),
5, 5));
    wordPanel.setBorder(new LineBorder(Color.WHITE));
    wordPanel.setBackground(Color.WHITE);
}

```

```

        wordPanel.setPreferredSize(new Dimension(30 * word.length(),
30));

        for (int i = 0; i < word.length(); i++) {
            JPanel letterPanel = createLetterPanel(word.charAt(i), i,
endWord.charAt(i), endWord);
            wordPanel.add(letterPanel);
        }
        return wordPanel;
    }

    private JPanel createLetterPanel(char letter, int position, char
endLetter, String endWord) {
        JPanel panel = new JPanel();
        int panelSize = 30;
        panel.setPreferredSize(new Dimension(panelSize, panelSize));
        panel.setMaximumSize(new Dimension(panelSize, panelSize));

        panel.setBackground(new Color(191, 181, 178));

        JLabel label = new JLabel(String.valueOf(letter));
        label.setHorizontalAlignment(SwingConstants.CENTER);
        label.setVerticalAlignment(SwingConstants.CENTER);

        if (position < endWord.length() && letter == endLetter) {
            panel.setBackground(Color.BLACK);
            label.setForeground(Color.WHITE);
        }

        panel.add(label);
        return panel;
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            try {
                Set<String> wordSet =
WordLadderGame.readWordsFromFile("words.txt");

```

```
        new WordLadderGUI(wordSet);
    } catch (IOException e) {
        JOptionPane.showMessageDialog(null, "Error reading words
from file: " + e.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
    }
    });
}
}
```

Lampiran

A. Pranala Repository:

https://github.com/hiirs/Tucil3_13522085

B. Checklist:

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	✓	