

LAPORAN TUGAS KECIL III
IF2211 STRATEGI ALGORITMA
SEMESTER II 2022-2023

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



Disusun oleh:

Attara Majesta Ayub

13522139

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG 2024

KATA PENGANTAR

Puji syukur kepada Tuhan Yang Maha Esa saya ucapkan atas penuntasan Tugas Kecil II IF2211 Strategi Algoritma yang berjudul “Penyelesaian Permainan Word Ladder Menggunakan Algoritma UCS, Greedy Best First Search, dan A*” ini.

Laporan ini merupakan dokumentasi dan penjelasan atas program dan algoritma yang telah saya implementasikan. Saya berusaha untuk merancang dan mengimplementasikan algoritma Uniform Cost Strategy, Greedy Best First Search, dan A* sedemikian rupa sehingga dapat menemukan solusi yang optimal untuk Word Ladder.

Penyelesaian tugas besar ini tidak lepas dari bimbingan dan arahan yang diberikan oleh dosen pengampu mata kuliah Strategi Algoritma. Saya juga ingin mengucapkan terima kasih kepada asisten mata kuliah Strategi Algoritma yang telah memberikan informasi secara berkala melalui laman *Question and Answer*.

Saya menyadari bahwa tugas besar ini masih jauh dari sempurna. Oleh karena itu, saya sangat terbuka untuk menerima kritik dan saran yang konstruktif demi perbaikan di masa yang akan datang. Akhir kata, semoga tugas besar ini dapat bermanfaat bagi semua pihak yang berkepentingan.

Sumedang, 07 Mei 2024,

13522139

DAFTAR ISI

KATA PENGANTAR.....

DAFTAR ISI.....

BAB I

DESKRIPSI TUGAS

1.1. Deskripsi Tugas

BAB II

LANDASAN TEORI

2.1 Algoritma Uniform Cost Search.....

2.2 Algoritma Greedy Best First Search.....

2.3 Algoritma A*

BAB III

APLIKASI ALGORITMA.....

BAB IV

IMPLEMENTASI DAN PENGUJIAN.....

4.1 Implementasi Uniform Cost Search.....

4.2 Implementasi Greedy Best First Search.....

4.3 Implementasi A*

BAB V

ANALISA ALGORITMA

BAB VI

PENUTUP.....

6.1 Kesimpulan

6.2 Saran

6.3 Refleksi

BAB VII

LAMPIRAN.....

7.1 GitHub Repository (Latest Release)

7.2 Tabel Ketercapaian

BAB I

DESKRIPSI TUGAS

1.1. Deskripsi Tugas

Word ladder, juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragrams, laddergrams, atau word golf, adalah permainan kata yang populer di semua kalangan, ditemukan oleh Lewis Carroll pada tahun 1877. Dalam permainan ini, pemain diberi dua kata awal yang disebut sebagai start word dan end word. Untuk memenangkan permainan, pemain harus menemukan serangkaian kata yang menghubungkan antara start word dan end word tersebut. Jumlah huruf pada start word dan end word selalu sama, dan setiap kata dalam serangkaian kata tersebut hanya boleh berbeda satu huruf dari kata sebelumnya. Dalam permainan ini, pemain ditantang untuk mencari solusi optimal, yaitu solusi yang menggunakan jumlah kata yang paling sedikit dalam serangkaian kata. Berikut adalah ilustrasi dan aturan permainannya.

How To Play

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

Rules

Weave your way from the start word to the end word.

Each word you enter **can only change 1 letter** from the word above it.

Example

E	A	S	T
---	---	---	---

EAST is the start word, WEST is the end word

V	A	S	T
---	---	---	---

We changed E to V to make VAST

V	E	S	T
---	---	---	---

We changed A to E to make VEST

W	E	S	T
---	---	---	---

And we changed V to W to make WEST

W	E	S	T
---	---	---	---

Done!

Gambar 1. Ilustrasi dan Peraturan Permainan Word Ladder

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

Sebagai contoh, jika kita memiliki kata awal "hit" dan kata akhir "cog", serta sebuah daftar kata seperti ["hot","dot","dog","lot","log","cog"], maka urutan transformasi terpendek adalah "hit" -> "hot" -> "dot" -> "dog" -> "cog", yang terdiri dari 5 kata. Namun, jika kata akhir "cog" tidak ada dalam daftar kata, maka tidak ada urutan transformasi yang valid.

Berdasarkan penjelasan di atas, kami diharapkan untuk membuat sebuah program interaktif yakni Word Ladder Solver. Word Ladder Solver diimplementasikan dengan tiga algoritma, Uniform Cost Search, Greedy Best First Search, dan A*, memberikan opsi kepada User untuk mencari solusi yang paling tepat dan optimal. User menginput startWord, kata awal, dan endWord, tujuan transformasi kata. Kemudian, user akan menginput pilihan algoritma yang diinginkan. program diharapkan untuk mengembalikan sekuens kata sebagai transformasi dari startWord ke endWord, jumlah kata dalam sekuens, dan waktu eksekusi. Sebuah transformasi sekuens dari startWord ke endWord menggunakan kamus wordList adalah sebuah sekuens kata startWord -> s1 -> s2 -> sk sedemikian rupa sehingga:

- Setiap pasangan kata yang berdekatan berbeda hanya dalam satu huruf.
- Setiap s_i untuk $1 \leq i \leq k$ ada dalam wordList. Perlu dicatat bahwa startWord perlu ada dalam wordList.
- $s_k == \text{endWord}$

Transformasi sekuens kata memiliki *constraints* berikut:

- $\text{endWord.length} == \text{startWord.length}$
- $\text{wordList}[i].\text{length} == \text{startWord.length}$
- startWord, endWord, and wordList[i] adalah kata dalam bahasa Inggris
- $\text{beginWord} \neq \text{endWord}$
- Semua kata di dalam wordList bersifat unik.

BAB II

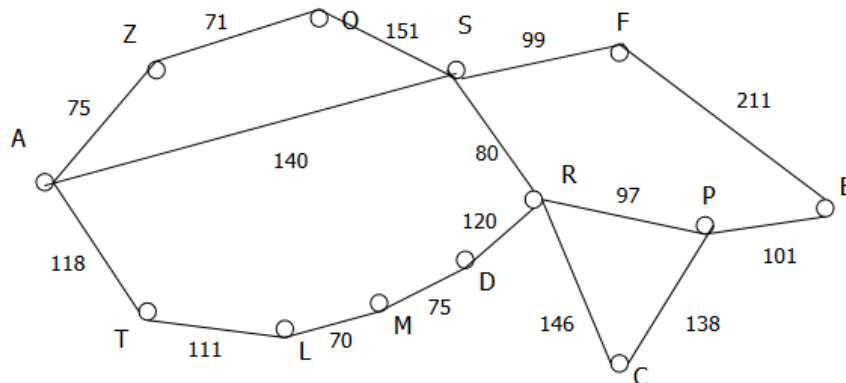
LANDASAN TEORI

2.1 Algoritma Uniform Cost Search (UCS)

Algoritma Uniform Cost Search merupakan salah satu algoritma yang diklasifikasikan sebagai Uninformed Search. Hal ini berarti teknik pencarian dalam algoritma ini tidak memiliki informasi atau pengetahuan tambahan mengenai kondisi luar dari yang telah didefinisikan pada masalah, atau dikenal juga sebagai *blind search*. Uniform Cost Search dapat dijelaskan juga sebagai kasus Breadth First Search (BFS) ketika seluruh *edges* pada *graph* tidak memiliki biaya yang sama.

Pada Uniform Cost Search (UCS), ekspansi node dilakukan berdasarkan biaya dari root. Pada langkah berikutnya, ekspansi ditentukan berdasarkan biaya terendah atau disebut sebagai fungsi $g(n)$. Fungsi $g(n)$ merupakan jumlah biaya edge dari root menuju node n .

Diberikan contoh graf berikut



Gambar 2. Contoh Ilustrasi Graf UCS

(Sumber: <https://~rinaldi.munir/Route-Planning-Bagian1>)

Maka, path yang optimal adalah $A \rightarrow S \rightarrow R \rightarrow B \rightarrow P$ dengan path-cost = 418

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

Gambar 3. Solusi Ilustrasi Graf UCS

(Sumber: <https://~rinaldi.munir/Route-Planning-Bagian1>)

2.2 Algoritma Greedy Best First Search

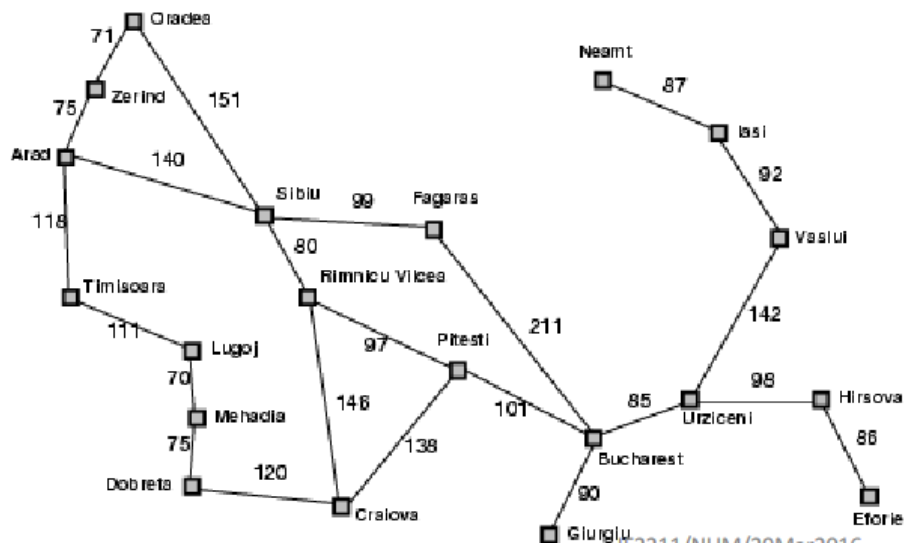
Algoritma Greedy Best First Search merupakan salah satu algoritma yang diklasifikasikan sebagai Informed Search atau Heuristic Search. Hal ini karena pencarian dengan algoritma ini menggunakan *knowledge* yang spesifik kepada permasalahan yang dihadapi di samping dari definisi masalahnya itu sendiri.

Pencarian Heuristik mengestimasi nilai dari sebuah simpul, yang meliputi tingkat keberhasilan suatu simpul, kesulitan dalam memecahkan submasalah, kualitas solusi yang diwakili oleh simpul tersebut, serta jumlah informasi yang diperoleh. Fungsi evaluasi

heuristik $f(n)$ merupakan fungsi evaluasi heuristik yang bergantung pada simpul n , tujuan pencarian, pencarian yang telah dilakukan sejauh ini, dan domain masalah.

Greedy Best-First Search merupakan algoritma pencarian Artificial Intelligence (AI) yang berusaha untuk menemukan jalur yang paling optimal dari root yang diberikan menuju node n . Algoritma ini memberikan prioritas pada *local best moves*, tanpa mempertimbangkan apakah jalur tersebut merupakan jalur terpendek atau tidak. Cara kerja algoritma ini adalah dengan mengevaluasi biaya dari setiap jalur yang mungkin kemudian ekspansi jalur dengan biaya terendah. Proses ini dilakukan secara iteratif hingga tujuan tercapai.

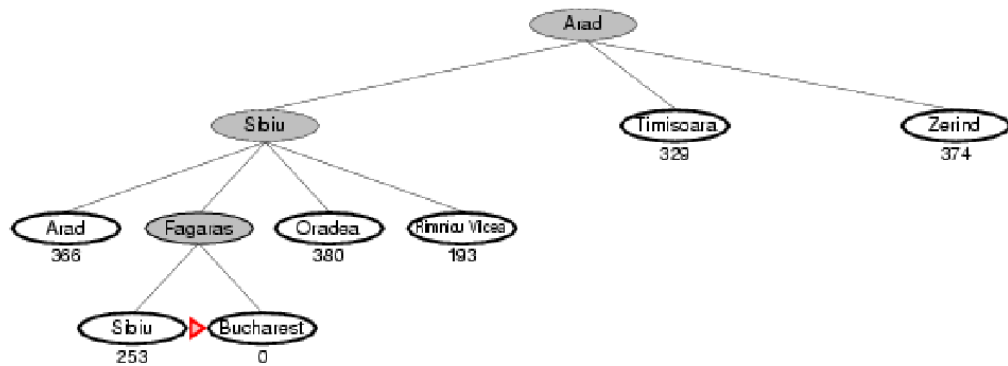
Diberikan contoh graf berikut



Gambar 4. Ilustrasi Graf Greedy Best First Search

(Sumber: <https://~rinaldi.munir/Route-Planning-Bagian1>)

Maka path yang ditemukan adalah $\text{Arad} \rightarrow \text{Sibiu} \rightarrow \text{Fagaras} \rightarrow \text{Bucharest}$, dengan path-cost = 450, solusi yang tidak optimal



Gambar 5. Solusi Ilustrasi Graf Greedy Best First Search

(Sumber: <https://~rinaldi.munir/Route-Planning-Bagian1>)

2.3 Algoritma A*

Algoritma A* merupakan salah satu algoritma yang diklasifikasikan sebagai Informed Search atau Heuristic Search. Hal ini karena pencarian dengan algoritma ini menggunakan *knowledge* yang spesifik kepada permasalahan yang dihadapi di samping dari definisi masalahnya itu sendiri.

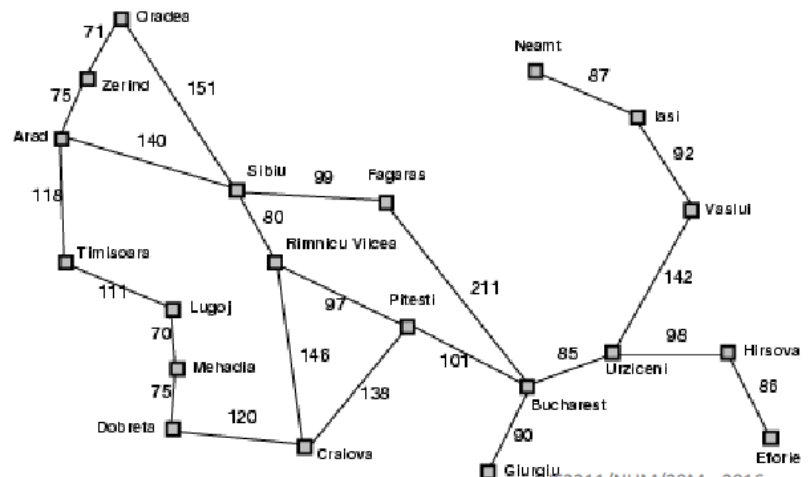
Dalam konteks algoritma A*, heuristik dianggap admissible jika nilai heuristiknya tidak pernah melebihi biaya sebenarnya untuk mencapai tujuan dari simpul yang dievaluasi. Dalam kasus word ladder, heuristik yang digunakan adalah jumlah karakter yang berbeda antara kata saat ini dan kata tujuan. Jika kita mengganti satu karakter pada kata saat ini, itu bisa membawa kita ke kata tujuan. Oleh karena itu, heuristik ini admissible karena estimasi tidak pernah lebih besar dari biaya sebenarnya untuk mencapai tujuan.

Ide dari A* Search adalah menghindari ekspansi jalur-jalur yang sudah mahal biayanya. Fungsi evaluasi $f(n)$ dalam A* Search dihitung sebagai jumlah biaya yang sudah ditempuh ($g(n)$) ditambah dengan estimasi biaya dari simpul n ke tujuan ($h(n)$). Di sini, $g(n)$ adalah biaya yang sudah dikeluarkan untuk mencapai simpul n , sedangkan $h(n)$ adalah perkiraan biaya dari simpul n ke tujuan. Nilai $f(n)$ menggambarkan perkiraan total biaya jalur melalui simpul n ke tujuan. Jika nilai $f(n)$ sama dengan $g(n)$, maka algoritma tersebut menjadi Uniform Cost Search (UCS). Jika nilai $f(n)$ hanya bergantung pada $h(n)$, maka

algoritma tersebut menjadi Greedy Best First Search. Sedangkan jika nilai $f(n)$ merupakan gabungan dari $g(n)$ dan $h(n)$, maka algoritma tersebut menjadi A^* .

Sebuah heuristik $h(n)$ disebut admissible jika untuk setiap simpul n , $h(n) \leq h^*(n)$, di mana $h^*(n)$ adalah biaya sebenarnya untuk mencapai keadaan tujuan dari simpul n . Sebuah heuristik admissible tidak pernah melebihi estimasi biaya untuk mencapai tujuan, artinya heuristik tersebut optimis. Teorema menyatakan bahwa jika $h(n)$ adalah admissible, maka penggunaan A^* menggunakan pencarian pohon (tree-search) akan optimal.

Diberikan contoh graf berikut

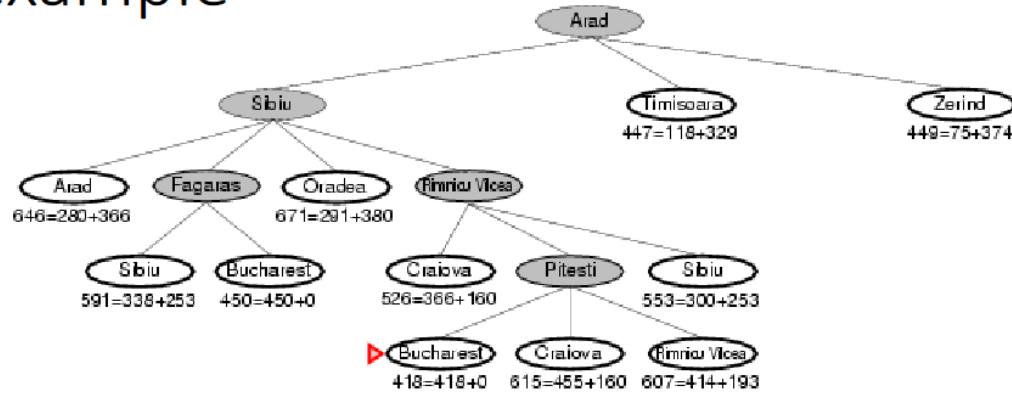


Gambar 6. Ilustrasi Graf A^*

(Sumber: <https://~rinaldi.munir/Route-Planning-Bagian2>)

Maka path yang ditemukan adalah $\text{Arad} \rightarrow \text{Sibiu} \rightarrow \text{Fagaras} \rightarrow \text{Bucharest}$, dengan path-cost = 450, solusi yang tidak optimal

Example



Gambar 7. Solusi Ilustrasi Graf A*

(Sumber: <https://~rinaldi.munir/Route-Planning-Bagian2>)

BAB III

APLIKASI ALGORITMA

5.1 Aplikasi

Uniform Cost Search

Algoritma ini dimulai dengan inisialisasi dua struktur data penting, yaitu cost dan parent. Cost digunakan untuk menyimpan biaya minimum yang diperlukan untuk mencapai suatu kata tertentu dari kata awal (startWord). Sedangkan parent digunakan untuk menyimpan kata sebelumnya yang membawa kita ke kata saat ini dalam pencarian jalur.

Setelah inisialisasi, algoritma memasukkan kata awal ke dalam antrian prioritas queue, dengan biaya awal 0. Kemudian, algoritma memulai iterasi melalui queue. Pada setiap iterasi, algoritma mengambil kata dengan biaya terendah dari queue. Jika kata yang diambil adalah kata tujuan (endWord), maka algoritma akan memanggil fungsi reconstructPath untuk membangun jalur dari kata awal ke kata tujuan.

Jika kata yang diambil bukanlah kata tujuan, algoritma akan memeriksa setiap tetangga dari kata tersebut. Tetangga-tetangga ini didapatkan melalui pemanggilan fungsi getNeighbors. Algoritma akan memperbarui biaya untuk mencapai tetangga baru jika biaya yang baru lebih kecil daripada biaya sebelumnya atau jika tetangga tersebut belum pernah dikunjungi sebelumnya. Setelah itu, tetangga baru akan dimasukkan ke dalam queue untuk dieksplorasi lebih lanjut.

Proses ini akan terus berlanjut hingga queue kosong atau hingga kata tujuan ditemukan. Jika queue kosong dan kata tujuan tidak ditemukan, algoritma akan mengembalikan daftar kosong sebagai jalur.

Dengan demikian, algoritma UCS secara efektif mengeksplorasi graf pencarian dari simpul ke simpul dengan biaya minimum yang semakin bertambah seiring dengan jaraknya dari simpul awal.

Greedy Best First Search

Algoritma Greedy Best-First Search (GBFS) pada kode di atas mengimplementasikan pendekatan heuristik untuk menemukan jalur yang paling menjanjikan dari suatu titik awal ke titik tujuan dalam sebuah graf. Prosesnya dapat dijelaskan sebagai berikut:

Pertama, algoritma melakukan inisialisasi dengan membuat dua struktur data utama, yaitu ``combinations`` dan ``parentMap``. ``Combinations`` digunakan untuk menyimpan pola kata-kata yang memiliki format yang sama dengan menggunakan tanda bintang sebagai pengganti karakter tertentu. Ini memungkinkan algoritma untuk mencocokkan kata-kata dengan pola yang serupa. Sedangkan, ``parentMap`` digunakan untuk menyimpan hubungan antara setiap kata dengan kata sebelumnya dalam jalur.

Setelah inisialisasi, algoritma memulai penelusuran graf menggunakan pendekatan BFS (Breadth-First Search). Ini dilakukan dengan menggunakan sebuah ``queue`` untuk menyimpan kata-kata yang akan dieksplorasi selanjutnya.

Selama penelusuran, algoritma memeriksa setiap kata yang ada dalam ``queue``. Untuk setiap kata, algoritma mencari tetangga-tetangga kata tersebut yang memiliki pola yang sama dalam ``combinations``. Kemudian, algoritma memeriksa apakah tetangga tersebut sudah pernah dikunjungi sebelumnya. Jika belum, tetangga tersebut akan dimasukkan ke dalam ``queue`` untuk dieksplorasi lebih lanjut.

Proses ini akan terus berlanjut hingga `queue` kosong atau hingga kata tujuan (`endWord`) ditemukan. Jika kata tujuan ditemukan, algoritma akan merekonstruksi jalur dari `startWord` ke `endWord` dengan memanfaatkan informasi yang disimpan dalam `parentMap`.

Dengan demikian, algoritma GBFS menggunakan pendekatan heuristik untuk mengeksplorasi graf dan menemukan jalur yang paling menjanjikan dari titik awal ke titik tujuan tanpa mempertimbangkan apakah jalur tersebut merupakan jalur terpendek atau tidak.

A*

Algoritma A* (A-Star) pada kode di atas mengimplementasikan pendekatan pencarian jalur yang menggabungkan informasi tentang biaya sejauh ini (gScore) dan estimasi biaya yang tersisa (fScore) untuk menemukan jalur yang paling optimal dari suatu titik awal ke titik tujuan dalam sebuah graf. Prosesnya dapat dijelaskan sebagai berikut:

Pertama, algoritma melakukan inisialisasi dengan membuat tiga struktur data utama, yaitu `cameFrom`, `gScore`, dan `fScore`. `CameFrom` digunakan untuk menyimpan hubungan antara setiap titik dengan titik sebelumnya dalam jalur terpendek yang telah ditemukan. `gScore` digunakan untuk menyimpan biaya terkecil yang diketahui untuk mencapai setiap titik. Sedangkan, `fScore` digunakan untuk menyimpan estimasi biaya total untuk mencapai titik tujuan dari setiap titik yang sedang dieksplorasi.

Setelah inisialisasi, algoritma memasukkan titik awal ke dalam antrian prioritas `openSet` dengan menggunakan estimasi biaya total dari titik awal ke titik tujuan. Selanjutnya, algoritma memulai iterasi melalui `openSet`. Pada setiap iterasi, algoritma mengambil titik dengan estimasi biaya total terendah dari `openSet`. Jika titik yang diambil adalah titik tujuan, algoritma akan mengembalikan jalur terpendek yang ditemukan menggunakan fungsi `reconstructPath`.

Jika titik yang diambil bukanlah titik tujuan, algoritma akan memeriksa setiap tetangga dari titik tersebut. Untuk setiap tetangga, algoritma menghitung biaya baru untuk mencapai tetangga tersebut dari titik awal melalui titik yang sedang dieksplorasi. Jika biaya baru tersebut lebih kecil dari biaya yang sudah ada, algoritma akan memperbarui informasi yang disimpan di `cameFrom`, `gScore`, dan `fScore` untuk tetangga tersebut. Kemudian, tetangga tersebut akan dimasukkan ke dalam `openSet` untuk dieksplorasi lebih lanjut.

Proses ini akan terus berlanjut hingga `openSet` kosong atau hingga titik tujuan ditemukan. Jika `openSet` kosong dan titik tujuan tidak ditemukan, algoritma akan mengembalikan jalur kosong sebagai jalur terpendek yang ditemukan.

Dengan demikian, algoritma A* menggunakan pendekatan heuristik dan informasi biaya sejauh ini untuk menemukan jalur terpendek dari titik awal ke titik tujuan dalam sebuah graf.

BAB IV

IMPLEMENTASI DAN PENGUJIAN

4.1 Implementasi Algoritma A*

4.1.1. Package dan Import

```
package backend;  
import java.util.*;
```

4.1.2. Methods

public static class SearchResult

```
public static class SearchResult {  
    private List<String> path;  
    private int numberOfPaths;  
  
    public SearchResult(List<String> path, int  
numberOfPaths) {  
        this.path = path;  
        this.numberOfPaths = numberOfPaths;  
    }  
  
    public List<String> getPath() {  
        return path;  
    }  
  
    public int getNumberOfPaths() {  
        return numberOfPaths;  
    }  
}
```

public static SearchResult findLadder(String startWord, String endWord,
Set<String> dictionary)


```

public static SearchResult findLadder(String
startWord, String endWord, Set<String> dictionary) {
    Map<String, String> cameFrom = new
HashMap<>();
    Map<String, Integer> gScore = new HashMap<>();
    Map<String, Integer> fScore = new HashMap<>();
    PriorityQueue<String> openSet = new
PriorityQueue<>(Comparator.comparingInt(fScore::get)
);

    gScore.put(startWord, 0);
    fScore.put(startWord,
heuristicCostEstimate(startWord, endWord));
    openSet.add(startWord);

    while (!openSet.isEmpty()) {
        String current = openSet.poll();
        if (current.equals(endWord)) {
            return new
SearchResult(reconstructPath(cameFrom, current),
gScore.get(endWord));
        }

        for (String neighbor :
getNeighbors(current, dictionary)) {
            int tentativeGScore =
gScore.getDefault(current, Integer.MAX_VALUE) + 1;
            if (tentativeGScore <
gScore.getDefault(neighbor, Integer.MAX_VALUE)) {
                cameFrom.put(neighbor, current);
                gScore.put(neighbor,
tentativeGScore);
                fScore.put(neighbor,
tentativeGScore + heuristicCostEstimate(neighbor,
endWord));

                if (!openSet.contains(neighbor)) {
                    openSet.add(neighbor);
                }
            }
        }
    }
}

```

```

        }

    }

    return new
    SearchResult(Collections.emptyList(), 0);
}

```

private static List<String> reconstructPath(Map<String, String> cameFrom, String
current)

```

private static List<String>
reconstructPath(Map<String, String> cameFrom, String
current) {
    List<String> path = new ArrayList<>();
    while (cameFrom.containsKey(current)) {
        path.add(current);
        current = cameFrom.get(current);
    }
    path.add(current);
    Collections.reverse(path);
    return path;
}

```

private static int heuristicCostEstimate(String current, String goal)

```

private static int heuristicCostEstimate(String
current, String goal) {
    Set<Character> currentChars = new HashSet<>();
    Set<Character> goalChars = new HashSet<>();
    for (char c : current.toCharArray()) {
        currentChars.add(c);
    }
    for (char c : goal.toCharArray()) {
        goalChars.add(c);
    }
    Set<Character> diff = new
HashSet<>(currentChars);
    diff.removeAll(goalChars);
    return diff.size();
}

```

private static List<String> getNeighbors(String word, Set<String> dictionary)

```

private static List<String> getNeighbors(String
word, Set<String> dictionary) {
    List<String> neighbors = new ArrayList<>();
    for (int i = 0; i < word.length(); i++) {
        for (char c = 'a'; c <= 'z'; c++) {
            String neighbor = word.substring(0,
i) + c + word.substring(i + 1);
            if (dictionary.contains(neighbor))
            {
                neighbors.add(neighbor);
            }
        }
    }
    return neighbors;
}

```

4.2 Implementasi Algoritma Greedy Best First Search

4.2.1. Import

```
package backend;  
import java.util.*;
```

4.2.2. Methods

```
public List<String> findLadder(String startWord, String endWord, List<String>  
wordList)
```

```

public List<String> findLadder(String startWord,
String endWord, List<String> wordList) {
    Map<String, List<String>> combinations = new
HashMap<>();
    Map<String, String> parentMap = new
HashMap<>();

    wordList.forEach(word -> {
        for (int i = 0; i < word.length(); i++) {
            String wordRoot = word.substring(0,
i) + '*' + word.substring(i + 1);
            if
(!combinations.containsKey(wordRoot)) {
                combinations.put(wordRoot, new
ArrayList<>());
            }

            combinations.get(wordRoot).add(word);
        }
    });

    Queue<String> queue = new LinkedList<>();
    queue.offer(startWord);
    parentMap.put(startWord, null);
    Set<String> visitedWords = new HashSet<>();
    visitedWords.add(startWord);

    while (!queue.isEmpty()) {
        int size = queue.size();
        for (int k = 0; k < size; k++) {
            String currentWord = queue.poll();
            for (int i = 0; i <
currentWord.length(); i++) {
                String wordRoot =
currentWord.substring(0, i) + '*' +
currentWord.substring(i + 1);
                for (String neighbor :
combinations.getOrDefault(wordRoot, new
ArrayList<>())) {

```

```

        if
        (!visitedWords.contains(neighbor)) {

            visitedWords.add(neighbor);
                                parentMap.put(neighbor,
currentWord);
                                queue.offer(neighbor);
        }
    }
}

    if (visitedWords.contains(endWord)) {
        List<String> path = new
ArrayList<>();

        String node = endWord;
        while (node != null) {
            path.add(node);
            node = parentMap.get(node);
        }
        Collections.reverse(path);
        return path;
    }

    return Collections.emptyList();
}
}

```

4.3 Implementasi Algoritma Uniform Cost Search

4.3.1. Package dan Import

```

package backend;
import java.util.*;

```

4.3.2. Methods

```

public List<String> findLadder(String startWord, String endWord, Set<String>
dictionary)

```

```

public List<String> findLadder(String startWord,
String endWord, Set<String> dictionary) {
    Map<String, Integer> cost = new HashMap<>();
    Map<String, String> parent = new HashMap<>();
    PriorityQueue<String> queue = new
PriorityQueue<>(Comparator.comparingInt(cost::get));

    cost.put(startWord, 0);
    queue.add(startWord);

    while (!queue.isEmpty()) {
        String current = queue.poll();
        if (current.equals(endWord)) {
            return reconstructPath(parent,
current);
        }

        for (String neighbor :
getNeighbors(current, dictionary)) {
            int newCost = cost.get(current) +
1;

            if (!cost.containsKey(neighbor) ||
newCost < cost.get(neighbor)) {
                cost.put(neighbor, newCost);
                parent.put(neighbor, current);
                queue.add(neighbor);
            }
        }
    }

    return Collections.emptyList();
}

```

private List<String> reconstructPath(Map<String, String> parent, String current)

```
private List<String> reconstructPath(Map<String,
String> parent, String current) {
    List<String> path = new ArrayList<>();
    while (current != null) {
        path.add(current);
        current = parent.get(current);
    }
    Collections.reverse(path);
    return path;
}
```

private List<String> getNeighbors(String word, Set<String> dictionary)

```
private List<String> getNeighbors(String word,
Set<String> dictionary) {
    List<String> neighbors = new ArrayList<>();
    for (int i = 0; i < word.length(); i++) {
        for (char c = 'a'; c <= 'z'; c++) {
            String neighbor = word.substring(0,
i) + c + word.substring(i + 1);
            if (dictionary.contains(neighbor))
            {
                neighbors.add(neighbor);
            }
        }
    }
    return neighbors;
}h;
```

4. 4. Pengujian

1. Algoritma Greedy Best First Search

a) Testcase 1

Word Ladder Solver

A CHEAT TO

Word 2024 Ladder

Start Word:

wok

End Word:

who

Algorithm:

GBFS

Generate

Solve

wok
woo
who

Execution Time: 378 milliseconds
Path found (2 words):

b) Testcase 2

Word Ladder Solver

A CHEAT TO

Word 2024 Ladder

Start Word:

atlases

End Word:

cabaret

Algorithm:

GBFS

Generate

Solve

tapered
tabered
tabored
taboret
tabaret
cabaret

Execution Time: 354 milliseconds
Path found (52 words):

c) Testcase 3

Word Ladder Solver

A CHEAT TO

Word 2024 Ladder

Start Word:

apple

End Word:

maple

Algorithm:

GBFS

Generate

Solve

arise
prise
paise
maise
maile
maple

Execution Time: 174 milliseconds
Path found (11 words):

d) Testcase 4

Word Ladder Solver

A CHEAT TO

Word 2024
Ladder



Start Word:

cows

End Word:

hadj

Algorithm:

GBFS

Generate

Solve

cows
caws
cads
cade
hade
hadj

Execution Time: 279 milliseconds

Path found (5 words):

e) **Testcase 5**

Word Ladder Solver

A CHEAT TO

Word 2024
Ladder

Start Word:

dog

End Word:

cat

Algorithm:

GBFS

Generate

Solve

dog

cog

cot

cat

Execution Time: 152 milliseconds
Path found (3 words):

f) Testcase 6

Word Ladder Solver

A CHEAT TO

Word 2024

Ladder

Start Word:

rajas

End Word:

fancy

Algorithm:

GBFS

Generate

Solve

ants

aits

dits

dots

dogs

doge

Execution Time: 237 milliseconds

Path found (6 words):


2. Algoritma UCS

a) Testcase 1

Word Ladder Solver

A CHEAT TO

Word 2024
Ladder



Start Word:

wok

End Word:

who

Algorithm:

UCS

Generate

Solve

wok
woo
who

Execution Time: 26 milliseconds
Path found (2 words):

b) Testcase 2

Word Ladder Solver

A CHEAT TO
**Word 2024
Ladder**

Start Word:

tamer

End Word:

reach

Algorithm:

UCS

Generate

Solve

torcs
torch
porch
poach
roach
reach

Execution Time: 158 milliseconds
Path found (9 words):

c) **Testcase 3**

Word Ladder Solver

A CHEAT TO

Word 2024
Ladder

Start Word:

solid

End Word:

roven

Algorithm:

UCS

Generate

Solve

solid
soled
sowed
rowed
rowen
roven

Execution Time: 54 milliseconds
Path found (5 words):

d) Testcase 4

Word Ladder Solver

A CHEAT TO

Word 2024 Ladder

Start Word:

zit

End Word:

kor

Algorithm:

UCS

Generate

Solve

zit
kit
kir
kor

Execution Time: 1 milliseconds

Path found (3 words):

e) **Testcase 5**

Word Ladder Solver

Word 2024 Ladder

Start Word:

pall

End Word:

wold

Algorithm:

UCS

Generate

Solve

pall
tall
toll
told
wold

Execution Time: 20 milliseconds
Path found (4 words):

f) **Testcase 6**

Word Ladder Solver

A CHEAT TO

Word 2024 Ladder

Start Word:

duro

End Word:

nidi

Algorithm:

UCS

Generate

Solve

duro
dure
dude
nude
nide
nidi

Execution Time: 16 milliseconds
Path found (5 words):

3. Algorithm A*

a) Tetscase 1

Word Ladder Solver

A CHEAT TO

Word 2024
Ladder



Start Word:

tyes

End Word:

abet

Algorithm:

A*

Generate

Solve

tyes
ayes
axes
axed
abed
abet

Execution Time: 17 milliseconds
Path found (5 words):

b) Testcase 2

Word Ladder Solver

Word 2024 Ladder

Start Word:

nab

End Word:

sau

Algorithm:

A*

Generate

Solve

nab
sab
sau

Execution Time: 0 milliseconds
Path found (2 words):

c) Testcase 3

Word Ladder Solver

A CHEAT TO

Word 2024 Ladder

Start Word:

lits

End Word:

seen

Algorithm:

A*

Generate

Solve

lits
lies
lees
sees
seen

Execution Time: 1 milliseconds
Path found (4 words):

d) Testcase 4

Word Ladder Solver

ACHEAT TO

Word 2024 Ladder

Start Word:

lek

End Word:

cad

Algorithm:

A*

Generate

Solve

ek
ed
ad
cad

Execution Time: 1 milliseconds

Path found (3 words):

e) Testcase 5

Word Ladder Solver

A CHEAT TO
**Word 2024
Ladder**

Start Word:
ana

End Word:
oxy

Algorithm:
A*

Generate

Solve

aha
wha
who
oho
oxo
oxy

Execution Time: 7 milliseconds
Path found (6 words):

f) Testcase 6

Word Ladder Solver

A CHEAT TO
**Word 2024
Ladder**

Start Word:
ceps

End Word:
inch

Algorithm:
A*

Generate

Solve

macs
mach
each
etch
itch
inch

Execution Time: 137 milliseconds
Path found (8 words):

BAB V

ANALISIS ALGORITMA

5.1 Analisis Efisiensi dan Efektivitas

Berikut adalah hasil analisis dari pengujian *testcase* dan analisis algoritma dalam *source code*.

5.1.1. Algoritma Uniform Cost Search

Implementasi algoritma UCS pada kasus Word Ladder digunakan untuk menemukan jalur yang paling pendek dari suatu kata awal ke kata akhir dengan mempertimbangkan biaya atau jumlah langkah yang diperlukan. Definisi $f(n)$ dan $g(n)$ dalam algoritma UCS adalah $f(n)$ merupakan biaya total untuk mencapai simpul n , sedangkan $g(n)$ adalah biaya yang telah dikeluarkan untuk mencapai simpul n .

- Time Complexity:

Secara umum, time complexity dari algoritma UCS tergantung pada jumlah kata dalam kamus (dictionary) dan panjang dari kata-kata tersebut. Iterasi utama dilakukan dengan menggunakan sebuah priority queue yang rata-rata memiliki waktu kompleksitas $O(\log n)$ untuk operasi enqueue dan dequeue. Pada setiap iterasi, algoritma memeriksa *neighbors* menggunakan method *getNeighbors* dari kata saat ini. Jumlah *neighbors* yang akan diperiksa tergantung pada panjang kata dan ukuran alfabet, sehingga dalam kasus terburuk, jumlah tetangga yang akan diperiksa adalah $O(k * 26)$, di mana k adalah panjang dari kata tersebut. Total kompleksitas waktu dari algoritma ini dapat dinyatakan sebagai $O((V + E) * \log V)$, di mana V adalah jumlah kata dalam kamus (vertex) dan E adalah jumlah edge yang mungkin antara kata-kata tersebut.

- Efektivitas:

Algoritma Uniform Cost Search cukup efektif dalam menemukan jalur optimal. Algoritma ini menjamin bahwa ketika ia menemukan solusi, solusi tersebut adalah solusi yang memiliki biaya (cost) paling rendah dibandingkan dengan semua solusi yang

mungkin. UCS mencari jalur dengan jumlah langkah (atau biaya) paling sedikit untuk mengubah satu kata menjadi kata lainnya.

- Waktu Eksekusi:

Waktu eksekusi dari algoritma ini akan bervariasi tergantung pada ukuran kamus dan kompleksitas hubungan antara kata-kata dalam kamus. Dibandingkan dengan algoritma Greedy Best-First Search (GBFS), UCS cenderung memiliki waktu eksekusi yang lebih lama karena GBFS hanya mempertimbangkan biaya langkah saat ini tanpa mempertimbangkan biaya langkah sebelumnya. Namun, UCS cenderung memiliki waktu eksekusi yang lebih cepat dibandingkan dengan algoritma A*.

5.1.2. Analisis Algoritma Greedy Best First Search

Algoritma GBFS pada kasus Word Ladder memiliki beberapa kesamaan dengan UCS namun dengan pendekatan yang lebih heuristik. Fungsi biaya $f(n)$ dalam GBFS hanya mencakup nilai heuristik $h(n)$, yang merupakan estimasi jarak langsung dari kata awal ke kata akhir. GBFS menjelajahi kata yang secara heuristik paling dekat dengan tujuan (beda huruf hanya satu). GBFS tidak menjamin solusi optimal karena kecenderungan untuk terjebak dalam lokal maksimum atau minimum.

- Time complexity

Iterasi awal untuk membentuk kombinasi memerlukan waktu $O(N * M)$, di mana N adalah jumlah kata dalam kamus dan M adalah panjang maksimum dari kata-kata tersebut. Selama pencarian, algoritma mengiterasi melalui setiap karakter dari setiap kata dalam kamus, dan kemudian mencari tetangga-tetangga yang sesuai dengan kombinasi wildcard. Jumlah tetangga yang akan diperiksa tergantung pada panjang kata dan ukuran alfabet, sehingga kompleksitas waktu adalah $O(M * 26)$. Total kompleksitas waktu dari algoritma ini adalah $O(N * M + M * 26)$, yang dapat disederhanakan menjadi $O(N * M)$.

- Efektivitas

GBFS tidak menjamin solusi optimal untuk persoalan word ladder. Meskipun GBFS cenderung memilih kata yang secara heuristik dianggap paling dekat dengan tujuan, itu tidak mempertimbangkan biaya aktual. Oleh karena itu, GBFS dapat terjebak dalam mencapai path yang tidak optimal, terutama jika heuristik tidak mencerminkan dengan baik

biaya aktual yang diperlukan untuk mencapai tujuan.. Mengorelasikan dengan UCS, jika biaya untuk setiap langkah sama (misalnya, 1 langkah), maka algoritma UCS akan memiliki hasil yang sama dengan BFS. Keduanya akan menghasilkan jalur terpendek antara dua kata dalam graf yang sama. Namun, UCS secara eksplisit menggunakan biaya aktual untuk mengurutkan simpul, sedangkan BFS hanya menggunakan urutan kedatangan.

- Waktu eksekusi

Waktu eksekusi dari algoritma GBFS akan bervariasi tergantung pada ukuran kamus dan kompleksitas hubungan antara kata-kata dalam kamus. Dibandingkan dengan UCS, GBFS memiliki waktu eksekusi yang paling cepat karena hanya mempertimbangkan estimasi jarak ke target, tanpa mempertimbangkan biaya langkah sebelumnya.

5.1.3 Analisis Algoritma A*

Algoritma A* menggunakan pendekatan yang mirip dengan UCS, dengan tambahan sebuah nilai estimasi (heuristic) yang digunakan untuk memprioritaskan simpul-simpul yang memiliki kemungkinan terbesar untuk mendekati tujuan (endWord). A* menggunakan dua map tambahan: gScore untuk melacak biaya sejauh ini untuk mencapai setiap simpul, dan fScore untuk melacak perkiraan total biaya dari startWord ke endWord melalui simpul saat ini. Algoritma juga menggunakan struktur data PriorityQueue untuk menyimpan simpul-simpul yang akan dieksplorasi selanjutnya, dengan prioritas berdasarkan fScore.

- Time complexity

Time complexity dari algoritma A* serupa dengan UCS, dengan tambahan waktu yang diperlukan untuk menghitung nilai heuristicCostEstimate. Perhitungan nilai heuristik dilakukan pada setiap iterasi untuk setiap simpul, dengan kompleksitas waktu $O(M)$, di mana M adalah panjang maksimum dari kata-kata dalam kamus. Total kompleksitas waktu dari algoritma ini adalah $O((V + E) * \log V)$, di mana V adalah jumlah kata dalam kamus (vertex) dan E adalah jumlah edge yang mungkin antara kata-kata tersebut.

- Efektivitas

Secara teoritis, algoritma A* lebih efisien dibandingkan dengan algoritma UCS dan GBFS dalam kasus word ladder karena A* memanfaatkan heuristik untuk memandu pencarian menuju tujuan, sementara UCS hanya mempertimbangkan biaya aktual. Dengan menggunakan informasi tambahan dari heuristik, A* dapat mengurangi jumlah simpul yang perlu dieksplorasi untuk menemukan jalur terpendek.

- Waktu eksekusi

Waktu eksekusi dari algoritma A* akan bervariasi tergantung pada ukuran kamus, kompleksitas hubungan antara kata-kata dalam kamus, dan kualitas nilai heuristic yang digunakan. Dibandingkan dengan UCS dan GBFS, A* cenderung memiliki waktu eksekusi yang lebih lambat, kemungkinan dipengaruhi nilai heuristic yang tidak akurat.

BAB VI

PENUTUP

Kesimpulan

Ketiga algoritma, yaitu Uniform Cost Search (UCS), Greedy Best-First Search (GBFS), dan A*, adalah pendekatan yang berbeda dalam menyelesaikan masalah Word Ladder. UCS menggunakan strategi penelusuran graf dengan biaya minimum, GBFS memprioritaskan simpul yang paling optimal, sedangkan A* menggabungkan biaya sejauh ini dengan estimasi biaya ke titik tujuan. Waktu eksekusi GBFS cenderung paling cepat, diikuti oleh UCS, dan terakhir A*. Namun, hal ini tetap bergantung kepada kasus yang diberikan.

Saran

Esplorasi terhadap algoritma lain seperti Bidirectional Search (BI-BFS), atau penggunaan heuristik yang lebih canggih dalam A* dapat meningkatkan performa dan efisiensi penyelesaian masalah Word Ladder. Selain itu, melakukan eksperimen dengan struktur data yang berbeda atau optimasi pada metode.

Refleksi terhadap Tugas Besar

Tugas ini memberikan kesempatan untuk memahami dan mengimplementasikan beberapa algoritma penting dalam pemecahan masalah. Pengalaman ini memberi wawasan tentang manajemen waktu, ketelitian, dan bagaimana menghadapi tantangan dalam menyelesaikan masalah pemrograman. Kesadaran akan kegagalan sebagai bagian dari proses pembelajaran juga penting, dan dapat digunakan sebagai pelajaran berharga untuk meningkatkan kualitas solusi di masa depan.

BAB VII







LAMPIRAN

7.1 GitHub Repository

https://github.com/attaramajesta/Tucil3_13522139

7.2 Tabel Ketercapaian Program

Tabel 7.2.1. Tabel Ketercapaian Program

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