LAPORAN TUGAS KECIL III

IF2211 STRATEGI ALGORITMA

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

Dosen:

Ir. Rila Mandala, M. Eng, Ph. D.

Monterico Adrian, S. T, M. T.

Disusun Oleh:

13522140 Yasmin Farisah Salma

PROGRAM STUDI TEKNIK INFORMATIKA

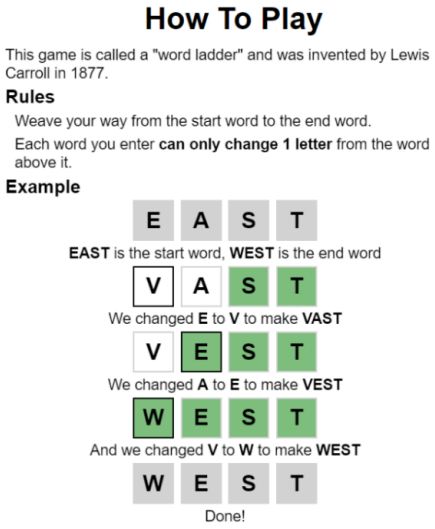
INSTITUT TEKNOLOGI BANDUNG

SEMESTER II TAHUN 2023/2024

DAFTAR ISI

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 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.

BAB II
Analisis dan Implementasi Algoritma

2.1 Analisis dan Implementasi Algoritma Uniform Cost Search

a. Deskripsi Langkah-Langkah Implementasi

Algoritma UCS memulai pencarian dari node awal dan berekspansi ke node tetangga berdasarkan biaya terendah dari jalur yang diambil. Dalam kasus Word Ladder, setiap langkah (perubahan satu huruf) memiliki biaya yang sama, sehingga UCS berfungsi mirip dengan Breadth-First Search (BFS) tetapi dengan prioritas pada path cost yang terakumulasi.

1. Inisialisasi

   Membuat queue prioritas untuk menyimpan node yang akan dieksplorasi berikutnya, dan dictionary untuk menyimpan cost dari node yang telah dikunjungi.
2. Pengecekan Kondisi Awal

   Memastikan panjang kata awal dan akhir sama.
3. Loop utama

   Menarik node dengan cost terendah dari queue dan mengiterasi setiap neighbor yang valid. Jika path cost ke neighbor lebih rendah dari yang telah tercatat, update cost dan masukkan ke queue.

b. Definisi f(n) dan g(n)

1. g(n) = Jumlah langkah dari node start ke node n. Dalam UCS, ini adalah jumlah total perubahan huruf dari start ke n.
2. f(n) = Dalam UCS, f(n) = g(n) karena tidak ada heuristik yang digunakan.

c. Perbandingan dengan BFS

UCS menggunakan priority queue yang diurutkan berdasarkan total cost yang terakumulasi dari path (disebut juga cost g(n)), sedangkan BFS menggunakan queue biasa yang hanya mengurutkan node berdasarkan urutan mereka ditambahkan (mengikuti prinsip first-in-first-out). Ini membuat UCS secara teknis lebih fleksibel karena bisa menyesuaikan dengan kasus di mana biaya per langkah berbeda, meski dalam konteks Word Ladder biaya setiap langkah seragam.

Dalam UCS, node yang memiliki total cost terendah selalu dieksplorasi terlebih dahulu, yang bisa berarti melompati beberapa node yang ditambahkan lebih awal jika node tersebut memiliki cost yang lebih tinggi. Sedangkan BFS akan selalu mengeksplorasi node dalam urutan mereka ditambahkan ke dalam queue, tidak memperhatikan total path cost.

Jika ada beberapa node dengan biaya yang sama, UCS tidak menjamin urutan eksplorasi yang sama dengan BFS. Karena meskipun mereka berdua akan mengeksplorasi node dengan cost paling rendah, urutan node yang memiliki cost sama bisa berbeda tergantung pada bagaimana mereka ditambahkan ke dalam priority queue di UCS.

2.2 Analisis dan Implementasi Algoritma Greedy Best First Search

a. Deskripsi Langkah-Langkah Implementasi

GBFS menggunakan heuristik untuk memprioritaskan node yang diperkirakan paling dekat ke tujuan, tanpa memperhitungkan jumlah langkah yang telah dilalui. Dalam Word Ladder ini, heuristik dihitung berdasarkan jumlah huruf yang berbeda antara kata saat ini dan kata tujuan.

1. Inisialisasi

   Membuat queue prioritas berdasarkan nilai heuristik dari node.

2. Pengecekan Kondisi Awal

   Memastikan panjang kata awal dan akhir sama.

3. Loop utama

   Menarik node dengan heuristic terkecil, mengiterasi setiap neighbor dan menambahkannya ke frontier jika heuristik neighbor lebih baik.

b. Definisi f(n) dan g(n)

1. g(n) = -
2. f(n) = Heuristik dari node n ke tujuan, dihitung berdasarkan jumlah huruf yang berbeda antara n dan end word.

c. Efisiensi dan Optimalitas

GBFS tidak menjamin solusi optimal karena hanya fokus pada node yang tampak paling menjanjikan tanpa mempertimbangkan total path cost yang telah dikeluarkan. Hal ini dapat menyebabkan algoritma mengabaikan jalur yang lebih pendek karena terpikat dengan heuristic yang menarik.

2.3 Analisis dan Implementasi Algoritma A*

a. Deskripsi Langkah-Langkah Implementasi

A* menggabungkan aspek UCS dan GBFS, menggunakan fungsi f(n) = g(n) + h(n) dimana g(n) adalah total biaya dari start ke n dan h(n) adalah heuristik dari n ke tujuan.

1. Inisialisasi

   Membuat queue prioritas berdasarkan f(n).

2. Pengecekan Kondisi Awal

   Memastikan panjang kata awal dan akhir sama.

3. Loop utama

   Mengiterasi melalui node, mengeksplorasi node dengan f(n) terendah, dan menambahkan tetangganya dengan menyesuaikan g(n) dan h(n).

b. Definisi f(n) dan g(n)

1. g(n) = Jumlah langkah dari start ke n.
2. h(n) = Heuristik, dihitung dari perbedaan karakter.
3. f(n) = Total estimasi biaya dari start ke tujuan melalui n.

c. Heuristik Admissible

Heuristik dalam A* admissible jika tidak pernah overestimate biaya aktual ke tujuan. Dalam kasus Word Ladder, menggunakan jumlah huruf yang berbeda sebagai heuristik adalah admissible karena ini menunjukkan jumlah minimum perubahan yang harus dilakukan, yang tidak akan pernah lebih dari jumlah langkah aktual yang dibutuhkan.

D. Efisiensi

A* secara teoritis lebih efisien daripada UCS dalam kasus Word Ladder karena A* menggunakan heuristik untuk mengurangi jumlah node yang dieksplorasi. A* memprioritaskan eksplorasi node yang tidak hanya dekat dengan start (seperti UCS) tetapi juga yang tampak mendekati tujuan, mengurangi path yang tidak perlu yang mungkin dieksplorasi oleh UCS.

BAB III
SOURCE CODE & PENJELASAN ALGORITMA

3.1 *Source Code* & Penjelasan Algoritma *UCS*

```
import java.util.*;

public class UCS {
    private Set<String> dictionary;

    public UCS(Set<String> dictionary) {
        this.dictionary = dictionary;
    }

    public List<String> findPath(String start, String end) {
        if (start.length() != end.length()) {
            System.out.println("Start and end words must be the same
length.");
            return null;
        }

        Queue<Node> frontier = new
PriorityQueue<>(Comparator.comparingInt(n -> n.cost));
        Map<String, Integer> visited = new HashMap<>();
        frontier.add(new Node(start, null, 0));
        int totalVisitedNodes = 0;

        while (!frontier.isEmpty()) {
            Node current = frontier.poll();
            totalVisitedNodes++;

            if (current.word.equals(end)) {
                System.out.println("Total nodes visited: " +
totalVisitedNodes);
                return constructPath(current);
            }

            for (String neighbor : getNeighbors(current.word)) {
                if (!visited.containsKey(neighbor) ||
visited.get(neighbor) > current.cost + 1) {
                    visited.put(neighbor, current.cost + 1);
                    frontier.add(new Node(neighbor, current, current.cost
+ 1));
```

```
                }
            }
        }
        System.out.println("Total nodes visited: " + totalVisitedNodes);
        return null;
    }

    private List<String> constructPath(Node node) {
        LinkedList<String> path = new LinkedList<>();
        while (node != null) {
            path.addFirst(node.word);
            node = node.parent;
        }
        return path;
    }

    private List<String> getNeighbors(String word) {
        List<String> neighbors = new ArrayList<>();
        char[] chars = word.toCharArray();
        for (int i = 0; i < chars.length; i++) {
            char oldChar = chars[i];
            for (char c = 'a'; c <= 'z'; c++) {
                if (c == oldChar) continue;
                chars[i] = c;
                String newWord = new String(chars);
                if (dictionary.contains(newWord)) {
                    neighbors.add(newWord);
                }
            }
            chars[i] = oldChar;
        }
        return neighbors;
    }

    private static class Node {
        String word;
        Node parent;
        int cost;

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

Kelas UCS menggunakan sebuah Set<String> bernama dictionary yang berisi semua kata yang valid, memastikan bahwa setiap kata yang dihasilkan dalam pencarian ada dalam daftar kata yang diperbolehkan. Konstruktor UCS(Set<String> dictionary) menginisialisasi objek UCS dengan kamus kata yang diberikan sebagai parameter, yang digunakan untuk validasi kata selama proses pencarian. Method public List<String> findPath(String start, String end) adalah Method utama dalam kelas ini yang mengatur logika pencarian. Jika kata awal dan akhir tidak memiliki panjang yang sama, Method ini akan mengembalikan null dan menampilkan pesan kesalahan. Dalam pencarian, digunakan sebuah priority queue (frontier) yang menyimpan node dengan urutan berdasarkan biaya terendah. Setiap node merepresentasikan kata dalam langkah pencarian, dengan Map<String, Integer> (visited) yang menyimpan biaya terendah untuk mencapai setiap kata dari kata awal untuk menghindari eksplorasi yang tidak perlu. Selama eksekusi, method ini mengambil node dengan biaya terendah dari frontier, memeriksa apakah itu node tujuan, dan jika ya, membangun jalur kembali ke node awal menggunakan Method private List<String> constructPath(Node node). Jalur ini dibangun dengan mengikuti rantai parent dari node tujuan kembali ke node awal, menjadikannya terurut dari awal ke tujuan. Selain itu, Method private List<String> getNeighbors(String word) digunakan untuk menghasilkan semua tetangga yang valid dari sebuah kata dengan mengubah setiap huruf dan memeriksa keberadaan kata baru dalam kamus.

Kelas Node merupakan struktur data internal yang digunakan untuk merepresentasikan setiap langkah dalam pencarian. Setiap Node menyimpan kata (word), referensi ke node induk (parent) yang digunakan untuk melacak jalur, dan biaya akumulatif (cost) untuk mencapai node tersebut dari kata awal. Konstruktor Node(String word, Node parent, int cost) digunakan untuk menginisialisasi node dengan kata, parent, dan biaya yang ditentukan.

3.2 *Source Code* & Penjelasan Algoritma *Greedy Best First Search*

```
import java.util.*;

public class GBFS {
    private Set<String> dictionary;

    public GBFS(Set<String> dictionary) {
        this.dictionary = dictionary;
    }

    public List<String> findPath(String start, String end) {
        if (start.length() != end.length()) {
            System.out.println("Start and end words must be the same
length.");
            return null;
        }

        Queue<Node> frontier = new
PriorityQueue<>(Comparator.comparingInt(n -> n.heuristic));
        Map<String, Integer> visited = new HashMap<>();
        Set<String> allVisitedNodes = new HashSet<>();

        frontier.add(new Node(start, null, heuristic(start, end)));
        allVisitedNodes.add(start);

        while (!frontier.isEmpty()) {
            Node current = frontier.poll();

            if (current.word.equals(end)) {
                System.out.println("Total visited nodes: " +
allVisitedNodes.size());
                return constructPath(current);
            }

            if (!visited.containsKey(current.word) ||
visited.get(current.word) > current.heuristic) {
                visited.put(current.word, current.heuristic);

                for (String neighbor : getNeighbors(current.word)) {
                    if (!visited.containsKey(neighbor) ||
visited.get(neighbor) > heuristic(neighbor, end)) {
                        frontier.add(new Node(neighbor, current,
heuristic(neighbor, end)));
                        allVisitedNodes.add(neighbor);
                    }
                }
```

```
            }
        }

        System.out.println("Total visited nodes: " +
allVisitedNodes.size());
        return null;
    }

    private List<String> constructPath(Node node) {
        LinkedList<String> path = new LinkedList<>();
        while (node != null) {
            path.addFirst(node.word);
            node = node.parent;
        }
        return path;
    }

    private List<String> getNeighbors(String word) {
        List<String> neighbors = new ArrayList<>();
        char[] chars = word.toCharArray();
        for (int i = 0; i < chars.length; i++) {
            char oldChar = chars[i];
            for (char c = 'a'; c <= 'z'; c++) {
                if (c == oldChar) continue;
                chars[i] = c;
                String newWord = new String(chars);
                if (dictionary.contains(newWord)) {
                    neighbors.add(newWord);
                }
            }
            chars[i] = oldChar;
        }
        return neighbors;
    }

    private int heuristic(String current, String goal) {
        int diff = 0;
        for (int i = 0; i < current.length(); i++) {
            if (current.charAt(i) != goal.charAt(i)) {
                diff++;
            }
        }
        return diff;
    }

    private static class Node {
        String word;
```

```
        Node parent;
        int heuristic;

        Node(String word, Node parent, int heuristic) {
            this.word = word;
            this.parent = parent;
            this.heuristic = heuristic;
        }
    }
}
```

Kelas GBFS (Greedy Best First Search) adalah implementasi dari algoritma pencarian jalur berbasis heuristik untuk permainan Word Ladder. Kelas ini menggunakan sebuah Set<String> yang disebut dictionary yang berisi semua kata valid untuk memverifikasi bahwa setiap kata yang dihasilkan selama proses pencarian adalah kata yang valid. Konstruktor GBFS(Set<String> dictionary) menginisialisasi objek GBFS dengan kamus kata yang disediakan, memastikan bahwa semua operasi pencarian mempertimbangkan hanya kata-kata yang ada dalam kamus ini. Method utama, public List<String> findPath(String start, String end), mencari jalur dari kata awal (start) ke kata tujuan (end) dengan mengoptimalkan berdasarkan heuristik. Heuristik ini dihitung sebagai jumlah huruf yang berbeda antara kata saat ini dan kata tujuan, memberikan estimasi perubahan minimal yang masih diperlukan. Jika panjang kata awal dan akhir tidak sama, Method mengembalikan null dan menampilkan pesan kesalahan. Dalam proses pencarian, GBFS menggunakan PriorityQueue yang dijuluki frontier, di mana node diurutkan berdasarkan nilai heuristik mereka untuk efisiensi dalam memilih node selanjutnya untuk eksplorasi. Map<String, Integer> yang bernama visited digunakan untuk menyimpan nilai heuristik terendah yang diperoleh untuk setiap kata, mencegah pengulangan eksplorasi yang tidak perlu. Selain itu, Set<String> yang bernama allVisitedNodes digunakan untuk melacak semua node yang telah dikunjungi selama pencarian. Method private

List<String> constructPath(Node node) membangun jalur dari node tujuan kembali ke node awal dengan mengikuti rantai parent dari setiap node. Jalur ini dibalik sehingga dimulai dari node awal dan berakhir di node tujuan. private List<String> getNeighbors(String word) menghasilkan semua tetangga yang valid dari kata dengan mengubah setiap huruf secara bergantian dan memeriksa apakah kata baru tersebut terdaftar dalam dictionary.

Kelas Node di dalam GBFS digunakan untuk menyimpan informasi tentang setiap kata selama pencarian jalur. Setiap Node berisi word (kata yang diwakili), parent (node induk dari node saat ini yang digunakan untuk melacak jalur saat membangun solusi), dan heuristic (nilai heuristik untuk node ini yang mengestimasi seberapa dekat node ini ke tujuan). Konstruktor Node(String word, Node parent, int heuristic) menginisialisasi instance dari Node dengan kata, parent, dan nilai heuristik yang ditentukan.

3.3 *Source Code* & Penjelasan Algoritma *A**

```
import java.util.*;

public class AStar {
    private Set<String> dictionary;

    public AStar(Set<String> dictionary) {
        this.dictionary = dictionary;
    }

    public List<String> findPath(String start, String end) {
        if (start.length() != end.length()) {
            System.out.println("Start and end words must be the same
length.");
            return null;
        }

        PriorityQueue<Node> frontier = new
PriorityQueue<>(Comparator.comparingInt(n -> n.fScore));
        Map<String, Integer> gScore = new HashMap<>();
        gScore.put(start, 0);
        Map<String, Node> cameFrom = new HashMap<>();
```

```
        Set<String> visitedNodes = new HashSet<>();  // To track all
visited nodes

        frontier.add(new Node(start, null, 0, heuristic(start, end)));
        visitedNodes.add(start);  // Add the start node to visited nodes

        while (!frontier.isEmpty()) {
            Node current = frontier.poll();

            if (current.word.equals(end)) {
                System.out.println("Total visited nodes: " +
visitedNodes.size());  // Print the number of visited nodes
                return constructPath(current);
            }

            for (String neighbor : getNeighbors(current.word)) {
                if (!visitedNodes.contains(neighbor)) {
                    visitedNodes.add(neighbor);  // Add new nodes to
visited nodes
                }
                int tentativeGScore = gScore.get(current.word) + 1;
                if (!gScore.containsKey(neighbor) || tentativeGScore <
gScore.get(neighbor)) {
                    cameFrom.put(neighbor, current);
                    gScore.put(neighbor, tentativeGScore);
                    int fScore = tentativeGScore + heuristic(neighbor,
end);
                    frontier.add(new Node(neighbor, current,
tentativeGScore, fScore));
                }
            }
        }
        System.out.println("Total visited nodes: " + visitedNodes.size());
// Print the number of visited nodes
        return null;
    }

    private List<String> constructPath(Node node) {
        LinkedList<String> path = new LinkedList<>();
        while (node != null) {
            path.addFirst(node.word);
            node = node.parent;
        }
        return path;
    }

    private List<String> getNeighbors(String word) {
```

```
        List<String> neighbors = new ArrayList<>();
        char[] chars = word.toCharArray();
        for (int i = 0; i < chars.length; i++) {
            char oldChar = chars[i];
            for (char c = 'a'; c <= 'z'; c++) {
                if (c == oldChar) continue;
                chars[i] = c;
                String newWord = new String(chars);
                if (dictionary.contains(newWord)) {
                    neighbors.add(newWord);
                }
            }
            chars[i] = oldChar;
        }
        return neighbors;
    }

    private int heuristic(String current, String goal) {
        int diff = 0;
        for (int i = 0; i < current.length(); i++) {
            if (current.charAt(i) != goal.charAt(i)) {
                diff++;
            }
        }
        return diff;
    }

    private static class Node {
        String word;
        Node parent;
        int gScore;
        int fScore;

        Node(String word, Node parent, int gScore, int fScore) {
            this.word = word;
            this.parent = parent;
            this.gScore = gScore;
            this.fScore = fScore;
        }
    }
}
```

Kelas AStar adalah implementasi dari algoritma pencarian A* yang digunakan untuk menemukan jalur optimal dalam permainan Word Ladder, di mana tujuannya adalah mengubah satu kata menjadi kata lain dengan jumlah langkah minimum. Kelas ini menggunakan sebuah Set<String> bernama dictionary, yang berisi semua kata valid dalam permainan. Set ini digunakan untuk memverifikasi bahwa kata-kata yang dihasilkan selama pencarian ada dalam kamus yang valid. Konstruktor AStar(Set<String> dictionary) menginisialisasi objek AStar dengan kamus kata yang disediakan, menyiapkan algoritma untuk memvalidasi kata-kata selama proses pencarian. Method utama, public List<String> findPath(String start, String end), mengatur proses pencarian dari kata awal (start) ke kata akhir (end). Jika panjang kata awal dan akhir tidak sama, Method ini mengembalikan null dan menampilkan pesan kesalahan. Proses pencarian menggunakan struktur data seperti PriorityQueue yang dijuluki frontier, di mana node disimpan dan diurutkan berdasarkan fScore, kombinasi dari gScore (biaya dari start) dan heuristik (estimasi biaya ke goal). Map<String, Integer> yang bernama gScore menyimpan biaya terendah dari start ke setiap node, dan Map<String, Node> yang bernama cameFrom digunakan untuk melacak hubungan antara node dan pendahulunya, yang penting untuk membangun jalur kembali ke awal setelah mencapai tujuan. Set<String> yang bernama visitedNodes digunakan untuk melacak semua node yang telah dikunjungi selama pencarian. Method private List<String> constructPath(Node node) membangun jalur dari node tujuan kembali ke node awal dengan mengikuti rantai parent dari setiap node, memastikan bahwa jalur tersebut akurat dan lengkap dari awal ke tujuan. Method private List<String> getNeighbors(String word) bertugas menghasilkan semua tetangga yang valid dari kata dengan mengubah setiap huruf secara bergantian dan memeriksa keberadaan kata baru dalam dictionary. Method private int heuristic(String current, String goal)

menghitung heuristik sebagai jumlah huruf yang berbeda antara kata saat ini dan kata tujuan, memberikan estimasi jumlah langkah minimal yang masih diperlukan untuk mencapai tujuan. Ini adalah bagian kritis dari algoritma A* yang membantu memandu pencarian untuk menjadi lebih efisien.

Kelas Node, sebagai struktur data internal, menyimpan informasi penting tentang setiap kata dalam pencarian jalur, termasuk word (kata yang diwakili), parent (node induk yang digunakan untuk melacak jalur), gScore (biaya akumulatif dari start), dan fScore (estimasi biaya total dari start ke tujuan melalui node ini). Konstruktor Node(String word, Node parent, int gScore, int fScore) menginisialisasi node dengan atribut yang diperlukan untuk pencarian A*.

3.4 *Main Program*

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

public class Main {
    private static Set<String> dictionary = new HashSet<>();

    public static void main(String[] args) throws IOException {
        loadDictionary("words.txt");

        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter start word:");
        String start = scanner.nextLine().trim();
        System.out.println("Enter end word:");
        String end = scanner.nextLine().trim();
        System.out.println("Choose algorithm: 1 for UCS, 2 for GBFS, 3 for
A*");
        int choice = scanner.nextInt();

        List<String> path = null;
        long startTime = System.currentTimeMillis();
        switch (choice) {
            case 1:
                UCS ucs = new UCS(dictionary);
                path = ucs.findPath(start, end);
                break;
```

```
            case 2:
                GBFS gbfs = new GBFS(dictionary);
                path = gbfs.findPath(start, end);
                break;
            case 3:
                AStar aStar = new AStar(dictionary);
                path = aStar.findPath(start, end);
                break;
            default:
                System.out.println("Invalid choice.");
                System.exit(1);
        }
        long endTime = System.currentTimeMillis();

        if (path != null) {
            System.out.println("Path: " + path);
            System.out.println("Time taken: " + (endTime - startTime) + "
ms");
        } else {
            System.out.println("No path found.");
        }
    }

    private static void loadDictionary(String fileName) throws IOException
{
        BufferedReader reader = new BufferedReader(new
FileReader(fileName));
        String line;
        while ((line = reader.readLine()) != null) {
            dictionary.add(line.trim().toLowerCase());
        }
        reader.close();
    }
}
```

Kelas Main dalam program ini bertindak sebagai titik masuk utama dan mengelola eksekusi algoritma pencarian jalur untuk permainan Word Ladder. Variabel dictionary, yang merupakan Set<String> statis, berisi semua kata valid yang diambil dari file teks. Fungsi ini memastikan bahwa setiap kata yang digunakan dalam proses pencarian adalah valid dan telah terverifikasi.

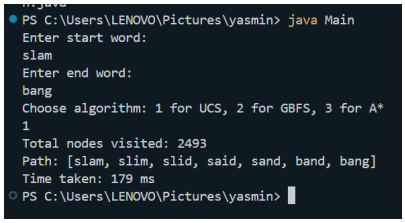
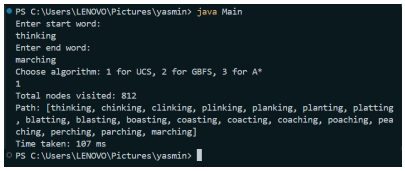
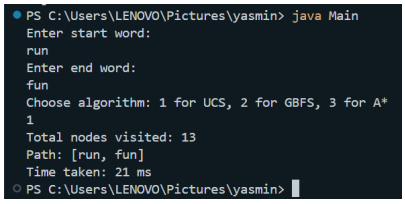
Method public static void main(String[] args) menjalankan program dan bertanggung jawab atas beberapa fungsi utama. Pertama, ia menangani input pengguna melalui Scanner, menerima kata awal dan akhir serta pilihan algoritma yang diinginkan pengguna antara UCS, GBFS, atau AStar. Setelah pengguna memilih algoritma, Method tersebut menginstansiasi kelas algoritma yang sesuai dan memanggil Method findPath untuk memulai pencarian jalur.

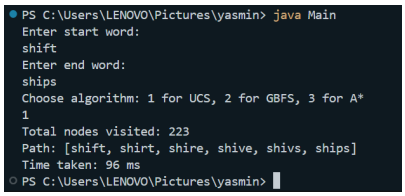
Selama proses pencarian, Method juga menghitung dan menampilkan waktu yang diperlukan untuk menyelesaikan pencarian, memberikan feedback waktu eksekusi yang berguna bagi pengguna. Setelah pencarian selesai, jika sebuah jalur ditemukan, ia akan menampilkan jalur tersebut beserta durasi pencarian. Jika tidak ada jalur yang ditemukan, program akan menampilkan pesan kesalahan yang sesuai.

Method private static void loadDictionary(String fileName) berfungsi untuk memuat kata-kata dari file teks ke dalam dictionary. Method ini menggunakan operasi I/O file untuk membuka dan membaca file yang diberikan, menambahkan setiap kata yang dibaca ke dalam dictionary setelah mengonversi kata-kata tersebut menjadi huruf kecil dan menghilangkan spasi. Hal ini penting untuk mempersiapkan data yang akan digunakan dalam pencarian jalur. Selain itu, Method ini mengandung penanganan eksepsi dengan melempar IOException yang harus ditangani oleh pemanggil, menjamin bahwa kesalahan dalam membaca file ditangani dengan tepat.

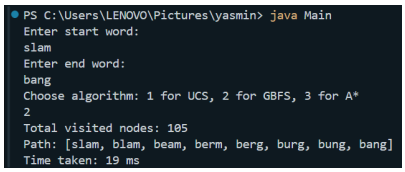
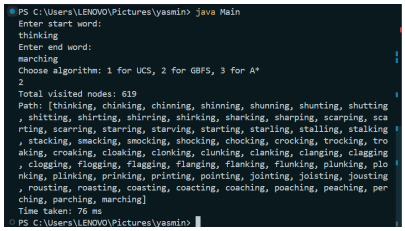
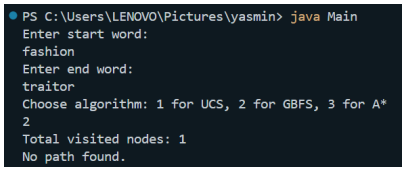
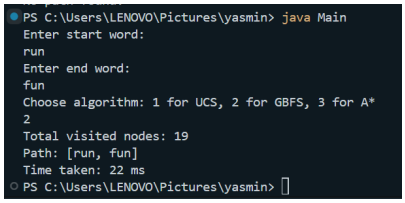
BAB IV
HASIL UJI COBA

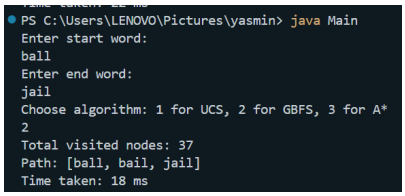
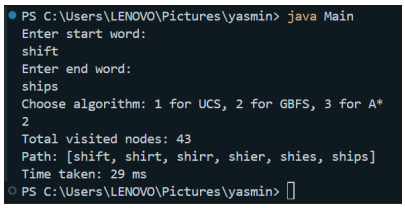
4.1 Hasil Uji Coba Algoritma UCS

Start Word	End Word	Output
slam	bang	
thinking	marching	
fashion	traitor	
run	fun	
ball	jail	

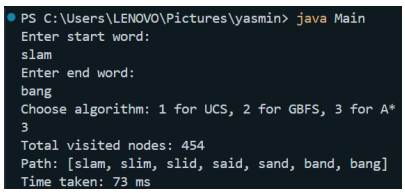
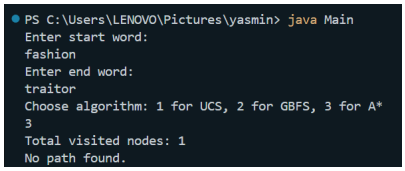
shift	ships	

4.2 Hasil Uji Coba Algoritma Greedy Best First Search

Start Word	End Word	Output
slam	bang	
thinking	marching	
fashion	traitor	
run	fun	

ball	jail	
shift	ships	

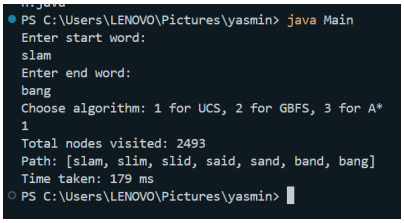
4.3 Hasil Uji Coba Algoritma A*

Start Word	End Word	Output
slam	bang	
thinking	marching	
fashion	traitor	

run	fun	PS C:\Users\LENOVO\Pictures\yasmin> java Main Enter start word: run Enter end word: fun Choose algorithm: 1 for UCS, 2 for GBFS, 3 for A* 3 Total visited nodes: 19 Path: [run, fun] Time taken: 20 ms
ball	jail	PS C:\Users\LENOVO\Pictures\yasmin> java Main Enter start word: ball Enter end word: jail Choose algorithm: 1 for UCS, 2 for GBFS, 3 for A* 3 Total visited nodes: 37 Path: [ball, bail, jail] Time taken: 22 ms
shift	ships	PS C:\Users\LENOVO\Pictures\yasmin> java Main Enter start word: shift Enter end word: ships Choose algorithm: 1 for UCS, 2 for GBFS, 3 for A* 3 Total visited nodes: 90 Path: [shift, shirt, shire, shine, shins, ships] Time taken: 33 ms PS C:\Users\LENOVO\Pictures\yasmin>

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

a. thinking - marching



1. Optimalitas

UCS dan A* menghasilkan jalur yang sama, menunjukkan bahwa keduanya mencapai solusi yang optimal dalam hal jumlah langkah yang diperlukan untuk berpindah dari kata awal ke kata akhir.

GBFS menunjukkan jalur yang lebih panjang dan kompleks. Hal ini mencerminkan karakteristik GBFS yang tidak selalu menghasilkan jalur optimal karena hanya fokus pada heuristik (jarak heuristik ke tujuan) tanpa mempertimbangkan biaya total dari jalur.

2. Waktu Eksekusi

GBFS memiliki waktu eksekusi tercepat di antara ketiganya, yang menunjukkan efisiensinya dalam menemukan solusi lebih cepat. Namun, perlu dicatat bahwa solusi yang ditemukan tidak optimal.

A* memiliki waktu eksekusi yang paling lambat, yang mungkin disebabkan oleh overhead komputasi dalam menggabungkan g(n) (biaya dari awal) dan h(n) (heuristik ke tujuan).

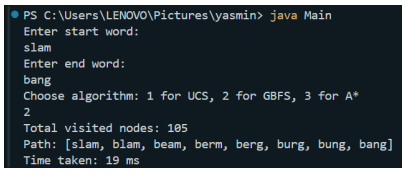
UCS memiliki waktu eksekusi yang moderat dan menemukan solusi yang optimal, menunjukkan keseimbangan antara kecepatan dan kualitas solusi.

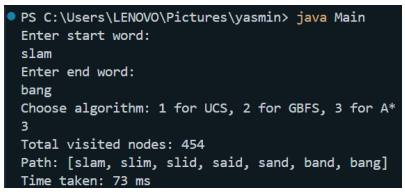
3. Memori (dari total visited nodes)

GBFS mengunjungi node paling sedikit, menunjukkan penggunaan memori yang paling efisien di antara ketiganya. Ini cocok untuk kasus di mana memori lebih dibatasi.

UCS dan A* memiliki jumlah node yang dikunjungi yang lebih tinggi, yang mencerminkan penggunaan memori yang lebih besar. Ini bisa menjadi pertimbangan penting dalam sistem dengan keterbatasan sumber daya.

b. slam - bang





1. Optimalitas dan Efisiensi

   UCS dan A* cenderung menemukan jalur yang optimal (seperti terlihat pada kasus "slam" ke "bang"), mengikuti urutan logis dari perubahan satu huruf dengan total langkah yang minimum. Keduanya memiliki jalur yang sama menunjukkan kecenderungan mencapai solusi optimal.

   GBFS menemukan jalur dengan lebih sedikit node yang dikunjungi, yang menunjukkan penggunaan memori yang lebih efisien dan waktu eksekusi yang jauh lebih cepat. Namun, jalur yang ditemukan bisa jadi tidak selalu optimal (tergantung pada kasus), tetapi

dalam kasus ini, GBFS berhasil menemukan jalur yang valid dan efisien dalam waktu yang singkat.

2. Waktu eksekusi

   GBFS memiliki waktu eksekusi tercepat dalam kasus "slam" ke "bang", yang membuatnya cocok untuk aplikasi yang membutuhkan respons cepat.

   A* memiliki waktu eksekusi yang lebih lambat dibandingkan GBFS tapi lebih cepat dari UCS, dengan jumlah node yang dikunjungi lebih sedikit dibandingkan UCS, menunjukkan keseimbangan yang baik antara kecepatan dan penggunaan sumber daya.

   UCS, meskipun paling lambat, menunjukkan hasil yang konsisten dalam mencari jalur yang optimal.

3. Memori

   Pada UCS, UCS (Uniform Cost Search), jumlah node yang dikunjungi cukup besar, seperti pada kasus "slam" ke "bang" dengan 2493 node. Ini menunjukkan bahwa UCS mungkin membutuhkan lebih banyak memori karena menyimpan banyak path yang mungkin dalam frontier (priority queue), dan setiap path harus menyimpan status lengkap dari path tersebut sampai titik tertentu. Ini bisa meningkatkan penggunaan memori secara signifikan.

   GBFS menunjukkan efisiensi memori yang lebih tinggi dalam kasus "slam" ke "bang", hanya dengan 105 node yang dikunjungi. Dengan fokus pada heuristik tanpa mempertimbangkan cost total, GBFS sering kali menjelajahi lebih sedikit node, yang berarti membutuhkan penyimpanan lebih sedikit dibandingkan dengan UCS.

Ini menjadikan GBFS sebagai pilihan yang lebih baik dalam kasus di mana memori terbatas.

A* pada kasus "slam" ke "bang" mengunjungi 454 node. Ini menunjukkan bahwa A* menggunakan memori lebih efisien daripada UCS tetapi lebih banyak daripada GBFS. A* menyimpan node dalam priority queue yang diurutkan berdasarkan fScore, yang merupakan gabungan dari cost nyata dari awal sampai node saat ini dan heuristik ke tujuan. Oleh karena itu, meskipun lebih memori intensif daripada GBFS, ia sering mencapai keseimbangan yang baik antara efisiensi memori dan pencarian optimal.

Kesimpulan

a. Optimalitas

UCS dan A*Konsisten menemukan jalur yang optimal dengan jumlah langkah minimum yang diperlukan, yang mengindikasikan keandalan mereka dalam menjamin solusi terbaik dalam segi langkah yang diambil.

GBFS Tidak selalu menghasilkan jalur optimal dan cenderung menghasilkan solusi yang lebih panjang atau lebih kompleks, karena algoritma ini lebih mengutamakan kecepatan menemukan tujuan tanpa mempertimbangkan total biaya jalur.

b. Waktu Eksekusi

GBFS Menunjukkan waktu eksekusi yang paling cepat, menjadikannya cocok untuk situasi di mana respons cepat lebih diutamakan daripada ketepatan solusi.

A* Memiliki waktu eksekusi yang lebih lama dari GBFS tapi lebih cepat daripada UCS, menggambarkan efisiensi yang baik antara kecepatan dan akurasi.

UCS Cenderung memiliki waktu eksekusi yang paling lambat tetapi hasil yang diperoleh adalah yang paling konsisten akurat.

c. Penggunaan Memori

GBFS Mengunjungi jumlah node yang paling sedikit, menandakan efisiensi memori yang tinggi. Ini menjadikan GBFS pilihan yang baik untuk situasi dengan keterbatasan memori.

A* Menunjukkan penggunaan memori yang moderat, lebih efisien dari UCS tetapi tidak seefisien GBFS.

UCS Mengunjungi jumlah node yang paling banyak, menunjukkan penggunaan memori yang lebih besar dan bisa menjadi masalah dalam lingkungan dengan keterbatasan sumber daya.

LAMPIRAN

5.1 GitHub Repository

https://github.com/caernations/Tucil3_13522140

5.2 Tabel Spesifikasi

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. **[Bonus]:** Program memiliki tampilan GUI		✓