LAPORAN TUGAS KECIL 3

IF2211 STRATEGI ALGORITMA

SEMESTER II TAHUN 2023/2024

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

Disusun oleh:

Ivan Hendrawan Tan

13522111

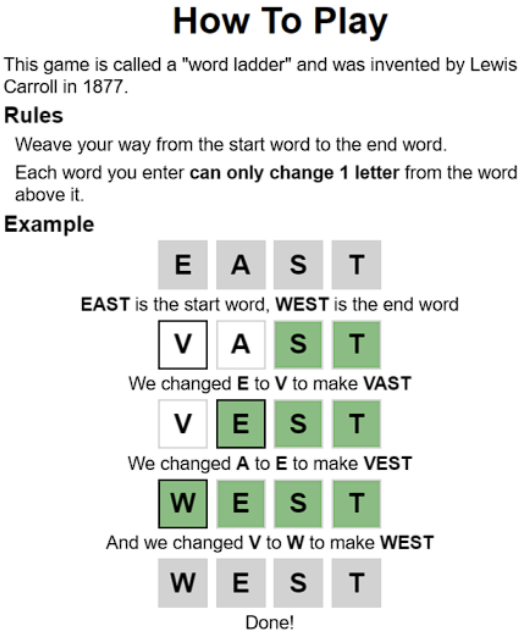
PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA (STEI)

INSTITUT TEKNOLOGI BANDUNG

DAFTAR ISI

BAB I
DESKRIPSI MASALAH



Word ladder (juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragrams, laddergrams, atau word golf) adalah salah satu permainan kata yang terkenal bagi seluruh kalangan. Word ladder ditemukan oleh Lewis Carroll, seorang penulis dan matematikawan, pada tahun 1877. Pada permainan ini, pemain diberikan dua kata yang disebut sebagai start word dan end word. Untuk memenangkan permainan, pemain harus menemukan rantai kata yang dapat menghubungkan antara start word dan end word. Banyaknya huruf pada start word dan end word selalu sama. Tiap kata yang berdekatan dalam rantai kata tersebut hanya boleh berbeda satu huruf saja. Pada permainan ini, diharapkan solusi optimal, yaitu solusi yang meminimalkan banyaknya kata yang dimasukkan pada rantai kata. Berikut adalah ilustrasi serta aturan permainan.

BAB II
TEORI DASAR

2.1. Algoritma *Uniform Cost Search* (UCS)

Uniform Cost Search adalah sebuah algoritma pencarian tanpa informasi tambahan yang digunakan untuk mencari jalur terpendek dari simpul awal ke simpul tujuan dalam graf. Algoritma ini mengutamakan prinsip pencarian jalur dengan biaya terkecil. *Uniform Cost Search* menggunakan *priority queue* sebagai tempat penyimpanan simpul, dengan prioritas tertinggi adalah simpul dengan biaya jalur total terkecil.

Eksplorasi dengan UCS dilakukan berdasarkan biaya total terkecil terlebih dahulu, bukan yang pertama kali muncul untuk meminimalkan biaya total. Dalam UCS, biaya dari simpul ke simpul dinyatakan dengan g(n).

2.2. Algoritma *Greedy Best First Search* (GBFS)

Greedy Best First Search adalah algoritma pencarian dengan informasi tambahan yang menggunakan heuristik sebagai bantuan untuk mencari simpul tujuan. *Greedy Best First Search* melakukan pencarian berbasis heuristik dengan fokus pada perluasan simpul yang tampaknya paling dekat dengan tujuan berdasarkan perkiraan biaya dari simpul saat ini ke simpul tujuan. Dalam *word ladder solver*, biaya heuristik ini dapat berupa jumlah persamaan atau perbedaan dari dua kata yang diberikan.

GBFS juga menggunakan *priority queue* sebagai basis penyimpanan simpul. Prioritas yang diterapkan dapat berbeda tergantung penciptanya. Jika berfokus pada jumlah perbedaan kata, maka prioritas diterapkan pada nilai heuristik terkecil. Sedangkan bila berfokus pada jumlah kesamaan kata, maka prioritas diterapkan berdasarkan nilai heuristik terbesar. Dalam GBFS, nilai heuristik dinyatakan dalam h(n).

2.3. Algoritma A* (*A-Star*)

A* adalah sebuah algoritma pencarian yang banyak digunakan untuk menemukan jalur terpendek dari simpul awal menuju simpul akhir. Algoritma ini

merupakan algoritma gabungan dari *Uniform Cost Search* yang berfokus pada biaya jalur terendah dan *Greedy Best First Search* yang menerapkan heurisitk dalam melakukan pencarian.

Heuristik dalam algoritma A* harus bersifat *admissible*, yaitu nilai heuristik tidak boleh melebihi nilai sebenarnya untuk menuju tujuan agar algoritma A* dapat menghasilkan solusi yang paling optimal. Parameter penilaian dalam algoritma A* adalah f(n) = g(n) + h(n) yang merupakan gabungan dari fungsi UCS dan GBFS. Jenis struktur data yang digunakan sama seperti UCS dan GBFS yaitu *priority queue*, namun dengan prioritas tertinggi berupa f(n) terkecil.

BAB III
ANALISIS PEMECAHAN PERMASALAHAN

3.1. *Uniform Cost Search* (UCS)

Penerapan UCS dalam algoritma ini dimulai dari pembacaan kata awal, kata tujuan, dan kata-kata Bahasa Inggris yang disimpan dalam file txt. Pencarian langkah-langkah diawali dengan mengecek terlebih dahulu apakah kamus menyimpan kata awal dan juga kata tujuan, jika salah satu kata tidak ada maka algoritma akan mengembalikan list kosong karena tidak ada jalan untuk mencapai solusi yang diinginkan.

Setelah itu, sebuah *priority queue* dan *hash map* dibuat masing-masing untuk menyimpan node dengan prioritas biaya terkecil dan menyimpan apakah sebuah kata telah dikunjungi sebelumnya. Penambahan node pertama dilakukan dengan kata awal, *null parent*, dan biaya sebesar 0.

Selanjutnya, lakukan iterasi hingga *priority queue* kosong. Didalam iterasi ini, ambil node dengan prioritas terbesar dan catat penambahan node yang telah dikunjungi sebesar satu. Jika node memiliki kata yang sama dengan kata tujuan, pencarian diakhiri dengan melakukan *print* terhadap jumlah node yang telah dikunjungi dan mengembalikan sebuah *list* yang berisi string yang telah diproses berdasarkan urutannya dari kata awal hingga kata tujuan.

Jika solusi belum ditemukan, lanjut pengecekan kata dari node tadi dengan fungsi *getNeighbors*. Fungsi *getNeighbors* akan melakukan iterasi terhadap setiap huruf dari kata tersebut dengan mengubahnya dari alfabet “a” hingga “z” dan mengecek apakah kata yang diubah hurufnya terdapat di dalam kamus. Jika terdapat di dalam kamus, maka kata baru tersebut akan dimasukkan ke dalam *list of string*. Untuk setiap kata baru tersebut, biaya baru dimasukkan sebesar biaya node sebelumnya ditambah dengan satu. Setelah itu, lakukan pengecekan terhadap kata baru tersebut apakah belum dikunjungi sebelumnya atau jika pernah dikunjungi, periksa apakah biaya baru lebih kecil daripada yang ada di dalam *visited*.

Jika solusi tidak ditemukan, sistem tetap akan menulis jumlah node yang pernah dikunjungi dan mengembalikan sebuah list kosong.

3.2. *Greedy Best First Search* (GBFS)

Penerapan prinsip GBFS tidak jauh berbeda dibandingkan dengan UCS, perbedaan yang ada terdapat di biaya yang diganti dengan sebuah nilai heuristik yang didapat dengan fungsi tambahan. Pencarian solusi dilakukan dengan mengecek terlebih dahulu apakah kata yang diberikan terdapat di dalam kamus atau tidak, jika tidak maka pencarian akan langsung dihentikan dan fungsi akan mengembalikan sebuah list kosong.

Selanjutnya, buat sebuah *priority queue* dan *hash set* yang masing-masing akan menyimpan node dengan prioritas berupa nilai heuristik terkecil dan *hash set* akan menyimpan apakah sebuah kata telah dikunjungi sebelumnya. Masukkan kkata awal dalam bentuk node ke *priority* queue. Selama *priority queue* tadi belum kosong, lakukan iterasi, keluarkan node dari *priority queue*, dan catat jumlah node yang telah dikunjungi. Jika *priority queue* telah kosong namun belum ditemukan solusi, maka GBFS akan mengeluarkan jumlah node yang telah dikunjungi dan sebuah list kosong.

Jika kata dari node yang diambil sebelumnya sama dengan kata tujuan, pencarian diakhiri dengan mengeluarkan jumlah node yang telah dikunjungi dan sebuah *list of string* yang berisi proses perubahan kata awal menuju kata tujuan. Jika belum ada solusi, kata tadi akan dicoba untuk ditambahkan ke dalam *hash set*. Jika belum ada di dalam *hash set*, maka kata tersebut akan dimasukkan ke dalam *hash set* dan lakukan pengecekan kata dengan fungsi *getNeighbors*.

Fungsi *getNeighbors* akan melakukan iterasi terhadap setiap huruf dari kata tersebut dengan mengubahnya dari alfabet “a” hingga “z” dan mengecek apakah kata yang diubah hurufnya terdapat di dalam kamus. Jika terdapat di dalam kamus, maka kata baru tersebut akan dimasukkan ke dalam *list of string*. Untuk setiap kata baru tersebut, lakukan pengecekan terhadap kata baru tersebut apakah telah dikunjungi sebelumnya, jika belum maka nilai heuristik baru dihitung dengan fungsi *getHeuristic* dan masukkan node baru dengan kata baru tersebut ke dalam *priority queue*.

Jika dibandingkan dengan algoritma UCS, ada beberapa perbedaan dengan algoritma UCS dalam hal urutan node yang dibangkitkan dan juga path yang dihasilkan. Dalam UCS, algoritma akan menelusuri seluruh kemungkinan huruf yang memiliki *cost* terkecil tanpa memedulikan kemiripan dari huruf tersebut sehingga akan memerlukan waktu lebih dalam melakukan penelusuran dibandingkan dengan GBFS. Berbeda

dengan GBFS yang berfokus terhadap mengurangi jumlah perbedaan kata yang ada. Maka dari itu, urutan node yang dibangkitkan dan path yang dihasilkan akan berbeda.

Secara teoritis, algoritma *Greedy Best First Search* tidak selalu menjamin solusi yang paling optimal dalam permasalahan *word ladder* karena nilai heuristik yang diterapkan pada GBFS berfokus pada mengurangi jumlah huruf yang berbeda, berbeda dengan UCS dan A* yang berfokus pada jalur tercepat untuk mencapai solusi.

3.3. *A-Star* (A*)

Algoritma A* menggabungkan prinsip UCS dan GBFS untuk mengeliminasi kelemahan dari kedua algoritma tersebut. Proses pencarian solusi dalam A* juga memiliki kemiripan dengan UCS dan GBFS. Pencarian diawali dengan mengecek terlebih dahulu apakah kata awal dan tujuan terdapat di dalam kamus, jika tidak ada maka pencarian akan langsung dihentikan dan fungsi akan mengembalikan sebuah list kosong.

Selanjutnya, buat sebuah *priority queue* dan *hash map* yang masing-masing akan menyimpan node dengan prioritas berupa nilai heuristik terkecil dan *hash map* akan menyimpan apakah sebuah kata telah dikunjungi sebelumnya. Perbedaan yang ada yaitu atribut yang dimiliko oleh node, jika node pada UCS memiliki atribut kata, node *Parent*, dan biaya, sedangkan node pada GBFS mengganti biaya dengan nilai heuristik, maka node pada A* akan menggunakan kedua atribut tersebut sehingga atribut node pada A* adalah kata, node *Parent*, biaya, dan nilai heuristik.

Setelah itu, Masukkan kata awal dalam bentuk node ke *priority* queue. Selama *priority queue* tadi belum kosong, lakukan iterasi, keluarkan node dari *priority queue*, dan catat jumlah node yang telah dikunjungi. Jika *priority queue* telah kosong namun belum ditemukan solusi, maka A* akan mengeluarkan jumlah node yang telah dikunjungi dan sebuah list kosong.

Jika kata dari node yang diambil sebelumnya sama dengan kata tujuan, pencarian diakhiri dengan mengeluarkan jumlah node yang telah dikunjungi dan sebuah *list of string* yang berisi proses perubahan kata awal menuju kata tujuan. Jika belum ada solusi, kata tadi akan dicoba untuk ditambahkan ke dalam *hash map*. Jika belum ada di

dalam *hash map*, maka kata tersebut akan dimasukkan ke dalam *hash* map dan lakukan pengecekan kata dengan fungsi *getNeighbors*.

Fungsi *getNeighbors* akan melakukan iterasi terhadap setiap huruf dari kata tersebut dengan mengubahnya dari alfabet "a" hingga "z" dan mengecek apakah kata yang diubah hurufnya terdapat di dalam kamus. Jika terdapat di dalam kamus, maka kata baru tersebut akan dimasukkan ke dalam *list of strings*. Selanjutnya, dapatkan biaya baru dengan mengecek biaya kata awal ditambah satu. Untuk setiap kata baru dalam *list of strings*, lakukan pengecekan apakah kata baru itu belum ada di dalam *hash map* atau apakah biaya baru yang didapat sebelumnya lebih kecil dibandingkan dengan biaya dari kata baru tersebut.

Jika memenuhi salah satu syarat, masukkan kata baru tersebut sebagai key dan biaya barunya sebagai value ke dalam *hash map*. Lalu hitung nilai heuristik dari kata baru itu dan masukkan ke dalam node baru yang berisi kata baru, node yang diambil di awal iterasi, biaya barunya, dan nilai heuristik tadi.

Jika dibandingkan dengan algoritma UCS atau GBFS, algoritma A* memberikan solusi yang optimal dengan tingkat efisiensi yang lebih baik daripada algoritma UCS namun sedikit lebih buruk daripada algoritma GBFS. Akan tetapi, algoritma GBFS terkadang memberikan solusi yang kurang optimal sehingga algoritma A* dapat dijadikan sebagai jalan tengahnya dengan hasil yang optimal dan penggunaan komputasi yang sedikit lebih banyak dibandingkan dengan GBFS.

BAB IV
IMPLEMENTASI DAN PENGUJIAN

4.1. Source Code

Utility.java

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

public final class Utility {
    public static HashSet<String> loadWordsFromFile(String filePath) throws FileNotFoundException {
        HashSet<String> words = new HashSet<>();
        try (Scanner scanner = new Scanner(new File(filePath))) {
            while (scanner.hasNextLine()) {
                String word = scanner.nextLine().trim();
                if (!word.isEmpty()) {
                    words.add(word);
                }
            }
        }
        return words;
    }

    // utility function to get all valid one-letter modifications of a word
    public static List<String> getNeighbors(String word, Set<String> dictionary) {
        List<String> neighbors = new ArrayList<>();
        char[] tempChars = word.toCharArray();

        for (int i = 0; i < tempChars.length; i++) {
            char oldChar = tempChars[i];
            for (char c = 'a'; c <= 'z'; c++) {
                if (c != oldChar) {
                    tempChars[i] = c;
                    String newWord = new String(tempChars);
                    if (dictionary.contains(newWord)) {
                        neighbors.add(newWord);
                    }
                }
            }
            tempChars[i] = oldChar;
        }
        return neighbors;
    }

    // heuristic function for Greedy Best First Search and A* algorithm
    public static int getHeuristic(String current, String goal) {
        int differences = 0;
        for (int i = 0; i < current.length(); i++) {
            if (current.charAt(i) != goal.charAt(i)) {
                differences++;
            }
        }
        return differences;
    }
}
```

Utility.java dibuat untuk menyimpan method yang digunakan secara bersamaan. Di dalamnya terdapat tiga buah fungsi seperti loadWordsFromFile, getNeighbors, getHeurisitc. Fungsi loadWordsFromFile berguna untuk menyimpan setiap kata yang ada di dalam kamus ke dalam bentuk *hash set* untuk memastikan setiap kata hanya berjumlah satu.

Selain itu, fungsi getNeighbors digunakan untuk melakukan perubahan terhadap satu huruf dari setiap huruf yang ada di kata yang diinput dan mengecek apakah kata yang telah diubah tadi valid dalam kamus. Fungsi yang terakhir yaitu getHeuristic untuk menghitung jumlah perbedaan huruf dari kata yang sedang dicek dengan kata tujuan.

UCS.java

```java
import java.util.*;

public class UCS {
    private HashSet<String> dictionary;
    private String startWord;
    private String endWord;

    // node class to hold each step of the word ladder
    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;
        }
    }

    public UCS(HashSet<String>wordSet, String startWord, String endWWord) {
        this.startWord = startWord;
        this.endWord = endWWord;
        this.dictionary = new HashSet<>(wordSet);
    }

    public List<String> findLadder() {
        if (!dictionary.contains(startWord) || !dictionary.contains(endWord)) {
            return Collections.emptyList();
        }

        PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(node -> node.cost));
        Set<String> visited = new HashSet<>();
        queue.add(new Node(startWord, null, 0));

        int nodesVisited = 0;
        while (!queue.isEmpty()) {
            Node current = queue.poll();
            nodesVisited++;
            //System.out.println("Current: " + current.word + ", Cost: " + current.cost);

            if (current.word.equals(endWord)) {
                System.out.println("\nNodes visited: " + nodesVisited);
                return buildPath(current);
            }

            for (String neighbor : Utility.getNeighbors(current.word, dictionary)) {
                int newCost = current.cost + 1;
                if (!visited.contains(neighbor)) {
                    visited.add(neighbor);
                    queue.add(new Node(neighbor, current, newCost));
                }
            }
        }
        System.out.println("\nNodes visited: " + nodesVisited);
        return Collections.emptyList();
    }

    // reconstruct path from endWWord to startWord
    private List<String> buildPath(Node endNode) {
        LinkedList<String> path = new LinkedList<>();
        for (Node node = endNode; node != null; node = node.parent) {
            path.addFirst(node.word);
        }
        return path;
    }
}
```

Di dalam kelas UCS, terdapat sebuah kelas lagi yang bernama Node untuk menyimpan sebuah kata, Node *parent* dari kata tersebut, beserta biaya dari kata itu. Method yang ada dalam kelas Node hanyalah konstruktornya untuk menciptakan objeknya.

Untuk method dari kelas UCS itu sendiri, ada method konstruktor, method findLadder untuk mencari solusi dari kata awalan dan kata tujuan, method buildPath untuk membuat sebuah *list of string* yang berisi urutan perubahan kata dari awal hingga akhir.

GreedyBFS.java

```java
import java.util.*;

public class GreedyBFS {
    private HashSet<String> dictionary;
    private String startWord;
    private String endWord;

    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;
        }
    }

    public GreedyBFS(HashSet<String>wordSet, String startWord, String endWord) {
        this.dictionary = new HashSet<>(wordSet);
        this.startWord = startWord;
        this.endWord = endWord;
    }

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

    public List<String> findLadder() {
        if (!dictionary.contains(startWord) || !dictionary.contains(endWord)) {
            return Collections.emptyList();
        }

        PriorityQueue<Node> frontier = new PriorityQueue<>(Comparator.comparingInt(node -> node.heuristic));
        Set<String> visited = new HashSet<>();
        frontier.offer(new Node(startWord, null, Utility.getHeuristic(startWord, endWord)));

        int nodesVisited = 0;

        while (!frontier.isEmpty()) {
            Node current = frontier.poll();
            nodesVisited++;
            //System.out.println("Current: " + current.word + ", Heuristic: " + current.heuristic);

            if (current.word.equals(endWord)) {
                System.out.println("\nNodes visited: " + nodesVisited);
                return buildPath(current);
            }
            if (visited.add(current.word)) {
                for (String neighbor : Utility.getNeighbors(current.word, dictionary)) {
                    if (!visited.contains(neighbor)) {
                        int heuristic = Utility.getHeuristic(neighbor, endWord);
                        frontier.offer(new Node(neighbor, current, heuristic));
                    }
                }
            }
        }
        System.out.println("\nNodes visited: " + nodesVisited);
        return Collections.emptyList();
    }
}
```

Mirip seperti UCS, kelas GreedyBFS juga memiliki kelas Node namun ada sedikit perbedaan yaitu biaya dari kata tersebut diganti dengan nilai heuristik dari kata tersebut. Selain dari itu, terdapat konstruktor untuk kelas ini, method buildPath yang sama seperti pada UCS, dan method findLadder yang juga digunakan untuk mencari solusi dari kata awal ke kata tujuan dengan sedikit penyesuaian berdasarkan perubahan atribut biaya dalam Node menjadi nilai heuristik.

AStar.java

```java
import java.util.*;

public class AStar {
    private HashSet<String> dictionary;
    private String startWord;
    private String endWord;

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

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

    public AStar(HashSet<String>wordSet, String startWord, String end) {
        this.startWord = startWord;
        this.endWord = end;
        this.dictionary = new HashSet<>(wordSet);
    }

    private List<String> buildPath(Node endNode) {
        LinkedList<String> path = new LinkedList<>();
        for (Node at = endNode; at != null; at = at.parent) {
            path.addFirst(at.word);
        }
        return path;
    }

    public List<String> findLadder() {
        if (!dictionary.contains(startWord) || !dictionary.contains(endWord)) {
            return Collections.emptyList();
        }

        PriorityQueue<Node> frontier = new PriorityQueue<>(Comparator.comparingInt(n -> n.cost + n.heuristic));
        Map<String, Integer> visitedWithCost = new HashMap<>();

        frontier.offer(new Node(startWord, null, 0, Utility.getHeuristic(startWord, endWord)));
        visitedWithCost.put(startWord, 0);
        int nodesVisitedWithCost = 0;

        while (!frontier.isEmpty()) {
            Node current = frontier.poll();
            if (current.cost > visitedWithCost.get(current.word)) {
                continue;
            }
            nodesVisitedWithCost++;

            if (current.word.equals(endWord)) {
                System.out.println("\nNodes Visited: " + nodesVisitedWithCost);
                return buildPath(current);
            }

            for (String neighbor : Utility.getNeighbors(current.word, dictionary)) {
                int newCost = visitedWithCost.get(current.word) + 1;
                if (!visitedWithCost.containsKey(neighbor)) {
                    visitedWithCost.put(neighbor, newCost);
                    int heuristic = Utility.getHeuristic(neighbor, endWord);
                    frontier.offer(new Node(neighbor, current, newCost, heuristic));
                }
            }
        }
        System.out.println("\nNodes visited: " + nodesVisitedWithCost);
        return Collections.emptyList();
    }
}
```

Untuk kelas AStar, terdapat pula perbedaan dalam kelas Node yaitu penggunaan atribut biaya beserta nilai heuristik yang bersamaan sebagai bahan evaluasi dalam method findLadder. Method lainnya dalam kelas ini adalah sebuah konstruktor, buildPath untuk mengurutkan proses perubahan kata dari awal hingga akhir, dan findLadder untuk mencari solusi yang mungkin didapat.

Main.java

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

public class Main {
    public static void main(String[] args) {
        String filePath, startWord, endWord;
        filePath = "";
        try {
            Scanner input = new Scanner(System.in);
            System.out.print("Input dictionary's file name and its extension: ");
            filePath = input.nextLine();
            int pilihan;
            HashSet<String> dict = Utility.loadWordsFromFile(filePath);
            if (dict.isEmpty()) {
                System.exit(0);
            }
            Collections.unmodifiableSet(dict);
            while (true) {
                System.out.print("Input start word: ");
                startWord = input.nextLine();
                System.out.print("Input end word: ");
                endWord = input.nextLine();
                startWord = startWord.toLowerCase();
                endWord = endWord.toLowerCase();
                if (startWord.length() == endWord.length()) {
                    break;
                }
                System.out.println("THe length of the startWord and endWord must be the same\n");
            }
            System.out.println("\n1. Uniform Cost Search");
            System.out.println("2. Greedy Best First Search");
            System.out.println("3. A*");
            System.out.println("4. All algorithm");
            while (true) {
                System.out.print("Input the number: ");
                pilihan = input.nextInt();
                if (pilihan >= 1 && pilihan <= 4) {
                    break;
                }
                System.out.println("Input valid from 1 until 4");
            }
            long startTime, endTime;
            Runtime run = Runtime.getRuntime();
            long startMemory, endMemory;

            if (pilihan == 1) {
                startTime = System.nanoTime();
                startMemory = run.totalMemory() - run.freeMemory();
                UCS UCSsolver = new UCS(dict, startWord, endWord);
                List<String> result = UCSsolver.findLadder();
                endMemory = run.totalMemory() - run.freeMemory();
                endTime = System.nanoTime();
                System.out.println("Memory used by UCS: " + (endMemory - startMemory)/1024 + " kb");
                System.out.println("Shortest path using UCS: " + result);
                System.out.println(((endTime - startTime)/1000000) + " ms");
            } else if (pilihan == 2) {
                startTime = System.nanoTime();
```

```
            startMemory = run.totalMemory() - run.freeMemory();
            GreedyBFS GBFSsolver = new GreedyBFS(dict, startWord, endWord);
            List<String> result = GBFSsolver.findLadder();
            endMemory = run.totalMemory() - run.freeMemory();
            endTime = System.nanoTime();
            System.out.println("Memory used by GBFS: " + (endMemory - startMemory)/1024 + " kb");
            System.out.println("Shortest path using Greedy BFS: " + result);
            System.out.println(((endTime - startTime)/1000000) + " ms");
        } else if (pilihan == 3) {
            startTime = System.nanoTime();
            startMemory = run.totalMemory() - run.freeMemory();
            AStar aStarSolver = new AStar(dict, startWord, endWord);
            List<String> result = aStarSolver.findLadder();
            endMemory = run.totalMemory() - run.freeMemory();
            endTime = System.nanoTime();
            System.out.println("Memory used by A*: " + (endMemory - startMemory)/1024 + " kb");
            System.out.println("Shortest path using A*: " + result);
            System.out.println(((endTime - startTime)/1000000) + " ms");
        } else if (pilihan == 4) {
            startTime = System.nanoTime();
            startMemory = run.totalMemory() - run.freeMemory();
            UCS UCSsolver = new UCS(dict, startWord, endWord);
            List<String> result = UCSsolver.findLadder();
            endMemory = run.totalMemory() - run.freeMemory();
            endTime = System.nanoTime();
            System.out.println("Memory used by UCS: " + (endMemory - startMemory)/1024 + " kb");
            System.out.println("Shortest path using UCS: " + result);
            System.out.println(((endTime - startTime)/1000000) + " ms");

            startTime = System.nanoTime();
            startMemory = run.totalMemory() - run.freeMemory();
            GreedyBFS GBFSsolver = new GreedyBFS(dict, startWord, endWord);
            List<String> result2 = GBFSsolver.findLadder();
            endMemory = run.totalMemory() - run.freeMemory();
            endTime = System.nanoTime();
            System.out.println("Memory used by GBFS: " + (endMemory - startMemory)/1024 + " kb");
            System.out.println("Shortest path using Greedy BFS: " + result2);
            System.out.println(((endTime - startTime)/1000000) + " ms");

            startTime = System.nanoTime();
            startMemory = run.totalMemory() - run.freeMemory();
            AStar aStarSolver = new AStar(dict, startWord, endWord);
            List<String> result3 = aStarSolver.findLadder();
            endMemory = run.totalMemory() - run.freeMemory();
            endTime = System.nanoTime();
            System.out.println("Memory used by A*: " + (endMemory - startMemory)/1024 + " kb");
            System.out.println("Shortest path using A*: " + result3);
            System.out.println(((endTime - startTime)/1000000) + " ms");
        }
        input.close();
    }
    catch (FileNotFoundException e) {
        System.out.println("File not found: " + filePath);
    }
  }
}
```

Kelas Main dibuat untuk menampilkan melakukan input terhadap kata awal, kata tujuan, dan juga kamus yang digunakan. Kelas ini tidak memiliki atribut atau method apapun karena hanya bertujuan untuk memanggil main.

4.2. Hasil Pengujian

1. Test Case 1

```
Input dictionary's file name and its extension: words.txt
Input start word: dry
Input end word: bee

1. Uniform Cost Search
2. Greedy Best First Search
3. A*
4. All algorithm
Input the number: 4

Nodes visited: 182
Memory used by UCS: 4569 kb
Shortest path using UCS: [dry, dey, dee, bee]
24 ms

Nodes visited: 4
Memory used by GBFS: 2949 kb
Shortest path using Greedy BFS: [dry, dey, bey, bee]
13 ms

Nodes Visited: 5
Memory used by A*: 2947 kb
Shortest path using A*: [dry, dey, bey, bee]
10 ms
```

2. Test Case 2

```
Input dictionary's file name and its extension: words.txt
Input start word: back
Input end word: pray

1. Uniform Cost Search
2. Greedy Best First Search
3. A*
4. All algorithm
Input the number: 4

Nodes visited: 3325
Memory used by UCS: 19574 kb
Shortest path using UCS: [back, pack, peck, peak, peat, prat, pray]
57 ms

Nodes visited: 11
Memory used by GBFS: 2952 kb
Shortest path using Greedy BFS: [back, pack, peck, peak, peag, peat, prat, pray]
11 ms

Nodes Visited: 115
Memory used by A*: 4095 kb
Shortest path using A*: [back, pack, peck, peak, peat, prat, pray]
16 ms
```

3. Test Case 3

```
Input dictionary's file name and its extension: words.txt
Input start word: drag
Input end word: beds

1. Uniform Cost Search
2. Greedy Best First Search
3. A*
4. All algorithm
Input the number: 4

Nodes visited: 740
Memory used by UCS: 8194 kb
Shortest path using UCS: [drag, brag, bras, baas, bads, beds]
57 ms

Nodes visited: 6
Memory used by GBFS: 2752 kb
Shortest path using Greedy BFS: [drag, brag, bras, baas, bads, beds]
21 ms

Nodes Visited: 20
Memory used by A*: 3389 kb
Shortest path using A*: [drag, brag, bras, baas, bads, beds]
17 ms
```

4. Test Case 4

```
Input dictionary's file name and its extension: words.txt
Input start word: watch
Input end word: plank

1. Uniform Cost Search
2. Greedy Best First Search
3. A*
4. All algorithm
Input the number: 4

Nodes visited: 3117
Memory used by UCS: 22917 kb
Shortest path using UCS: [watch, ratch, retch, reach, peach, peace, place, plane, plank]
66 ms

Nodes visited: 12
Memory used by GBFS: 3706 kb
Shortest path using Greedy BFS: [watch, patch, parch, perch, peach, peace, place, plane, plank]
13 ms

Nodes Visited: 71
Memory used by A*: 3359 kb
Shortest path using A*: [watch, patch, parch, perch, peach, peace, place, plack, plank]
11 ms
```

5. Test Case 5

```
Input dictionary's file name and its extension: words.txt
Input start word: parka
Input end word: scarf

1. Uniform Cost Search
2. Greedy Best First Search
3. A*
4. All algorithm
Input the number: 4

Nodes visited: 3955
Memory used by UCS: 27961 kb
Shortest path using UCS: [parka, parks, perks, peaks, pears, sears, scars, scarf]
73 ms

Nodes visited: 44
Memory used by GBFS: 3145 kb
Shortest path using Greedy BFS: [parka, parks, parrs, pairs, fairs, wairs, hairs, haars, hears, sears, scars, scarf]
12 ms

Nodes Visited: 35
Memory used by A*: 3384 kb
Shortest path using A*: [parka, parks, perks, peaks, pears, sears, scars, scarf]
16 ms
```

6. Test Case 6

```
Input dictionary's file name and its extension: words.txt
Input start word: cheddar
Input end word: panther

1. Uniform Cost Search
2. Greedy Best First Search
3. A*
4. All algorithm
Input the number: 4

Nodes visited: 7649
Memory used by UCS: 69217 kb
Shortest path using UCS: []
137 ms

Nodes visited: 18108
Memory used by GBFS: -64277 kb
Shortest path using Greedy BFS: []
112 ms

Nodes visited: 7648
Memory used by A*: 67584 kb
Shortest path using A*: []
74 ms
```

4.3. Hasil Analisis

Berdasarkan data dari hasil pengujian yang dilakukan sebelumnya, berikut adalah penyajiannya dalam bentuk tabel.

Nomor Percobaan	Jenis Percobaan	Jumlah Path	Waktu Eksekusi (ms)	Banyak node yang dikunjungi	Penggunaan memori (KB)
1	UCS	4	24	182	4569
	GBFS	4	13	4	2949
	A*	4	10	5	2947
2	UCS	7	57	3325	19574
	GBFS	8	11	11	2952
	A*	7	16	115	4095
3	UCS	6	57	740	8194
	GBFS	6	21	6	2752
	A*	6	17	20	3389

4	UCS	9	66	3117	22917
	GBFS	9	13	12	3706
	A*	9	11	71	3359
5	UCS	8	73	3955	27961
	GBFS	12	12	44	3145
	A*	8	16	35	3384
6	UCS	0	137	7649	69217
	GBFS	0	112	18108	-64277
	A*	0	74	7648	67584

Dari tabel data di atas, terdapat perbedaan signifikan dalam performa dari ketiga jenis algoritma yang digunakan dalam memecahkan permasalahan dalam permainan *Word ladder*, yaitu *Uniform Cost Search* (UCS), *Greedy Best First Search* (GBFS), dan *A-Star* (A*).

Uniform Cost Search menunjukkan bahwa solusi yang dihasilkan akan optimal bila dihitung dari jumlah path untuk menuju kata tujuan lebih sedikit dibandingkan dengan algoritma *Greedy Best First Search*. Akan tetapi, UCS memakan waktu yang paling lama dan juga memakan memori terbesar dibandingkan dengan kedua pesaingnya. Hal ini terlihat dari perbedaan jumlah node yang dikunjungi yang dapat mencapai 2 kali lipat hingga ratusan kali.

Di sisi lain, *Greedy Best First Search* menunjukkan persaingannya dalam hal waktu eksekusi dan jumlah node yang dikunjungi. Akan tetapi, muncul sebuah kelemahan baru yaitu solusi yang diberikan bisa jadi tidak optimal. Ketidakoptimal ini terjadi karena GBFS yang cenderung memilih jalur yang nampaknya mengarah ke tujuan, tanpa mempertimbangkan keseluruhan jalur dan node lain yang dapat memberikan solusi yang lebih optimal sehingga memicu ketidakoptimalan ini.

Algoritma A* tampak seperti pemenang berdasarkan data di atas, dilihat dari solusi yang dihasilkan selalu sama dengan UCS yang berarti bahwa solusi yang diberikan oleh A* selalu optimal. Penggunaan memorinya juga lebih baik dibandingkan dengan UCS namun masih sedikit lebih buruk dari GBFS. Jumlah node yang dikunjungi juga lebih baik dibandingkan dengan UCS, tetapi juga sedikit lebih buruk daripada GBFS. algoritma A* Dengan mempertimbangkan hasil yang diberikan beserta dengan performanya, algoritma A* dapat menjadi pilihan terbaik untuk memecahkan masalah dalam permainan *Word ladder*.

Pada test case 6, penggunaan memori pada algoritma GBFS bernilai negatif. Hal ini disebabkan karena terdapat *glitch* dari JVM dan bukan karena terdapat kesalahan dalam alokasi memori oleh program.

DAFTAR REFERENSI

Munir, Rinaldi. Penentuan rute (Route/Path Planning) - Bagian 1. Diakses 4 Mei 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf

Munir, Rinaldi. Penentuan rute (Route/Path Planning) - Bagian 2. Diakses 4 Mei 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf

LAMPIRAN

Link repository GitHub: https://github.com/Bodleh/Tucil3-13522111.git

Tabel kelayakan

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		✓