

LAPORAN TUGAS KECIL 3 IF2211

STRATEGI ALGORITMA

*PENYELESAIAN PERMAINAN WORD LADDER MENGGUNAKAN ALGORITMA UCS, GREEDY
BEST-FIRST SEARCH, DAN A**



Disusun oleh:

Marzuli Suhada M - 13522070

**PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG**

2023/2024

DAFTAR ISI

BAB I.....	4
BAB II.....	5
2.1 Algoritma Uniform-Cost Search.....	5
2.2 Algoritma Greedy Best-First Search.....	6
2.3 Algoritma A*.....	6
BAB III.....	8
3.1 Analisis dan Implementasi dalam Algoritma Uniform-Cost Search (UCS).....	8
3.2 Analisis dan Implementasi dalam Algoritma Greedy Best-First Search.....	8
3.3 Analisis dan Implementasi dalam Algoritma A*.....	9
3.4 Source Code.....	10
3.4.1 Node.java.....	11
3.4.2 Utility.java.....	12
3.4.3 Result.java.....	14
3.4.4 WordDictionary.java.....	14
3.4.5 UCS.java.....	15
3.4.6 GBFS.java.....	17
3.4.7 AStar.java.....	19
3.4.8 Main.java.....	20
3.4.9 Bonus GUI.java.....	25
BAB IV.....	34
4.1 Hasil Uji Program Wajib.....	34
4.1.1 Uji Program 1.....	34
4.1.2 Uji Program 2.....	35
4.1.3 Uji Program 3.....	36
4.1.4 Uji Program 4.....	38
4.1.5 Uji Program 5.....	39
4.1.6 Uji Program 6.....	41
4.2 Hasil Uji Program Bonus.....	42
4.2.1 Uji Program 1.....	42
4.2.2 Uji Program 2.....	44
4.2.3 Uji Program 3.....	45
4.2.4 Uji Program 4.....	47
4.2.5 Uji Program 5.....	49
4.2.6 Uji Program 6.....	51
4.3 Analisis Perbandingan Solusi UCS, Greedy Best First Search, dan A*.....	52
4.3.1 Optimalitas.....	53
4.3.2 Waktu Eksekusi.....	54
4.3.3 Optimalitas.....	54
4.4 Penjelasan Mengenai Implementasi Bonus.....	55
BAB V.....	59
5.1 Kesimpulan.....	59
5.2 Saran.....	59

DAFTAR PUSTAKA	61
LAMPIRAN	62
Pranala.....	62
How to Use.....	62
Tabel Poin.....	62

BAB I

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.

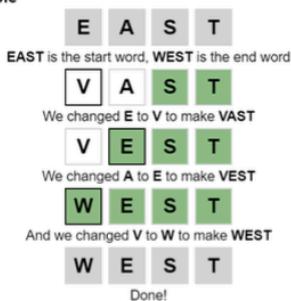
How To Play

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



Gambar 1. Ilustrasi dan Peraturan Permainan *Word Ladder*

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

Permainannya cukup sederhana bukan? Jika belum paham dengan peraturan permainannya, cobalah untuk memainkan permainannya pada link sumber di atas. Dalam pemenuhan tugas ini, kami diminta untuk membuat sebuah solver permainan tersebut dengan harapan kita dapat menemukan solusi paling optimal untuk menyelesaikan permainan *Word Ladder* ini.

BAB II

LANDASAN TEORI

2.1 Algoritma Uniform-Cost Search

Algoritma UCS (Uniform Cost Search) adalah sebuah algoritma pencarian yang digunakan untuk menemukan jalur terpendek dalam graf berarah atau tidak berarah. Algoritma ini mirip dengan algoritma Dijkstra, tetapi difokuskan untuk mencapai satu tujuan tertentu dari titik awal dengan biaya terendah. UCS sering digunakan dalam konteks penjelajahan graf dan juga dalam algoritma AI untuk menemukan solusi optimal. Di dalam algoritma UCS, dapat digunakan sebuah *priority queue* dengan *priority* dari sebuah simpul adalah:

$$g(n) = \text{total biaya dari akar ke simpul } n$$

Berikut merupakan skema umum dalam algoritma Uniform-Cost Search.

```
function UniformCostSearch(graph, start, goal):
    // Buat priority queue yang akan menyimpan simpul-simpul berdasarkan
    // biaya terendah
    priorityQueue = PriorityQueue()

    // Tambahkan simpul awal ke priority queue dengan biaya 0
    priorityQueue.enqueue((0, start, [])) // biaya, simpul, jalur

    // Set untuk menyimpan simpul yang sudah dikunjungi
    visited = Set()

    while not priorityQueue.isEmpty():
        // Ambil simpul dengan biaya terendah dari priority queue
        currentCost, currentNode, currentPath = priorityQueue.dequeue()

        // Jika simpul ini adalah tujuan, kembalikan jalur dan biaya
        if currentNode == goal:
            return (currentPath + [currentNode], currentCost)

        // Jika simpul belum dikunjungi, tandai sebagai dikunjungi
        if currentNode not in visited:
            visited.add(currentNode)

            // Telusuri semua tetangga dari simpul saat ini
            for neighbor, cost in graph.neighbors(currentNode):
                if neighbor not in visited:
                    totalCost = currentCost + cost
                    priorityQueue.enqueue((totalCost, neighbor, currentPath
+ [currentNode]))

    // Jika tidak menemukan solusi
    return None
```

2.2 Algoritma Greedy Best-First Search

Algoritma Greedy Best-First Search (GBFS) adalah algoritma pencarian yang menggunakan heuristik untuk menentukan urutan eksplorasi simpul dalam graf atau pohon. Tujuan dari algoritma ini adalah menemukan jalur menuju simpul tujuan dengan cara "rakus", yaitu selalu memilih simpul yang terlihat paling menjanjikan berdasarkan nilai heuristik. Karena bersifat "rakus", algoritma ini tidak selalu menemukan solusi optimal tapi bisa lebih cepat dibandingkan algoritma pencarian lain. Di dalam algoritma UCS, dapat digunakan sebuah *priority queue* dengan *priority* dari sebuah simpul adalah:

$$h(n) = \text{nilai heuristik simpul } n \text{ ke simpul tujuan}$$

Berikut merupakan skema umum dalam algoritma Greedy Best-First Search.

```
function GreedyBestFirstSearch(graph, start, goal, heuristic):
    // Priority queue untuk menyimpan simpul-simpul berdasarkan nilai
    heuristik terendah
    priorityQueue = PriorityQueue()

    // Tambahkan simpul awal ke priority queue dengan heuristik yang
    dihitung
    priorityQueue.enqueue((heuristic(start, goal), start, [])) // nilai
    heuristik, simpul, jalur

    // Set untuk menyimpan simpul yang sudah dikunjungi
    visited = Set()

    while not priorityQueue.isEmpty():
        // Ambil simpul dengan nilai heuristik terendah
        _, currentNode, currentPath = priorityQueue.dequeue()

        // Jika simpul ini adalah tujuan, kembalikan jalur
        if currentNode == goal:
            return currentPath + [currentNode]

        // Jika simpul belum dikunjungi, tambahkan ke dalam visited
        if currentNode not in visited:
            visited.add(currentNode)

            // Telusuri semua tetangga dari simpul saat ini
            for neighbor in graph.neighbors(currentNode):
                if neighbor not in visited:
                    priorityQueue.enqueue((heuristic(neighbor, goal),
                    neighbor, currentPath + [currentNode]))

    // Jika tidak menemukan solusi
    return None
```

2.3 Algoritma A*

Algoritma A* adalah algoritma pencarian yang menggabungkan elemen-elemen dari Uniform Cost Search dan Greedy Best-First Search. Algoritma ini sering digunakan untuk

menemukan jalur terpendek dalam graf atau pohon, dan sangat populer di berbagai aplikasi, termasuk game, robotika, dan sistem navigasi. A* dikenal karena efisiensi dan kemampuannya untuk menemukan solusi optimal. Di dalam algoritma A*, dapat digunakan sebuah *priority queue* dengan *priority* dari sebuah simpul adalah:

$$g(n) = \text{total biaya dari akar sampai simpul ke } n$$

$$h(n) = \text{nilai heuristik simpul } n \text{ ke simpul tujuan}$$

$$f(n) = g(n) + h(n); f(n) \text{ adalah prioritas simpul } n$$

Berikut merupakan skema umum dalam algoritma A*.

```
function AStar(graph, start, goal, heuristic):
    // Priority queue untuk menyimpan simpul-simpul berdasarkan nilai f
    // terendah
    openSet = PriorityQueue()

    // Tambahkan simpul awal ke openSet dengan g=0 dan f dihitung dari
    // heuristik
    openSet.enqueue((heuristic(start, goal), 0, start, [])) // f, g,
    simpul, jalur

    // Set untuk menyimpan simpul yang sudah dikunjungi
    closedSet = Set()

    while not openSet.isEmpty():
        // Ambil simpul dengan nilai f terendah
        _, currentG, currentNode, currentPath = openSet.dequeue()

        // Jika simpul ini adalah tujuan, kembalikan jalur dan biaya
        // kumulatif
        if currentNode == goal:
            return (currentPath + [currentNode], currentG)

        // Tambahkan simpul ke closedSet untuk mencegah eksplorasi ulang
        closedSet.add(currentNode)

        // Telusuri semua tetangga dari simpul saat ini
        for neighbor, cost in graph.neighbors(currentNode):
            if neighbor not in closedSet:
                tentativeG = currentG + cost
                f = tentativeG + heuristic(neighbor, goal)

                // Jika tetangga belum ada di openSet atau g lebih rendah
                openSet.enqueue((f, tentativeG, neighbor, currentPath +
                [currentNode]))

        // Jika tidak menemukan solusi
        return None
```

BAB III

IMPLEMENTASI

3.1 Analisis dan Implementasi dalam Algoritma Uniform-Cost Search (UCS)

Algoritma Uniform-Cost Search (UCS) menggunakan struktur data *priority queue* untuk memilih simpul dengan biaya terendah dalam pencarian solusi. Biaya ini biasanya terkait dengan panjang jalur atau "cost" kumulatif dari simpul awal ke simpul saat ini. Pada kasus permainan *word ladder*, setiap transisi antara kata-kata memiliki biaya yang sama, sehingga UCS berfungsi seperti Breadth-First Search (BFS), yang menelusuri semua simpul yang memiliki biaya terendah sebelum beralih ke simpul dengan biaya yang lebih tinggi.

A. Langkah-langkah Algoritma UCS:

1. Masukkan simpul awal ke dalam priority queue dengan biaya 0.
2. Setiap iterasi, keluarkan simpul dengan biaya terendah dari priority queue.
3. Jika simpul yang diambil adalah tujuan, maka jalur ditemukan.
4. Jika simpul sudah dikunjungi, lanjutkan ke simpul berikutnya.
5. Jika belum, tandai simpul sebagai dikunjungi dan tambahkan semua tetangganya dengan biaya yang meningkat (berdasarkan hubungan ke simpul saat ini).

B. $f(n)$ dan $g(n)$:

1. $f(n)$: Dalam UCS, $f(n)$ adalah sama dengan $g(n)$, yaitu total biaya untuk mencapai simpul n.
2. $g(n)$: Sama dengan $f(n)$, karena $g(n)$ hanya mempertimbangkan biaya kumulatif. Dalam penyelesaian *word ladder*, $g(n)$ adalah jumlah langkah dari simpul awal ke simpul saat ini.

C. Apakah UCS sama dengan BFS?

Pada kasus *word ladder*, UCS dan BFS akan menghasilkan urutan simpul dan jalur yang sama, karena biaya setiap langkah transisi adalah seragam. Dalam UCS, pemilihan simpul didasarkan pada biaya kumulatif yang identik dengan BFS dalam kasus ini.

3.2 Analisis dan Implementasi dalam Algoritma Greedy Best-First Search

Greedy Best-First Search menggunakan heuristik untuk memprioritaskan simpul dalam pencarian. Fokus utama GBFS adalah pada nilai heuristik, tanpa mempertimbangkan biaya kumulatif dari simpul awal. Ini berarti bahwa GBFS bisa lebih cepat dalam menemukan jalur, tetapi tidak selalu optimal.

A. Langkah-langkah Algoritma Greedy Best-First Search:

1. Masukkan simpul awal ke dalam *priority queue* dengan nilai heuristik yang dihitung dari simpul awal ke simpul tujuan.
2. Setiap iterasi, keluarkan simpul dengan nilai heuristik terendah dari *priority queue*.
3. Jika simpul adalah tujuan, jalur ditemukan.
4. Jika simpul sudah dikunjungi, lanjutkan ke simpul berikutnya.
5. Jika belum, tandai simpul sebagai dikunjungi dan tambahkan semua tetangganya.

B. $f(n)$ dan $h(n)$:

1. $f(n)$: Dalam GBFS, nilai $f(n)$ tidak mempertimbangkan biaya kumulatif dari simpul awal ke simpul saat ini. Algoritma ini hanya fokus pada nilai heuristik untuk memprioritaskan simpul dalam pencarian. Jadi, $f(n)$ untuk GBFS hanyalah nilai heuristik, yaitu $h(n)$.
2. $h(n)$: Dalam GBFS, $h(n)$ adalah estimasi jarak atau biaya antara simpul saat ini dan simpul tujuan. Pada kasus penyelesaian *word ladder*, `Utility.heuristic(n.getWord(), endWord)` menghitung perbedaan antara kata saat ini dan kata tujuan. Estimasi ini berupa jumlah huruf yang berbeda di antara kedua kata tersebut.

C. Apakah GBFS menjamin solusi optimal?

Tidak. Karena GBFS hanya fokus pada nilai heuristik tanpa mempertimbangkan biaya kumulatif, maka algoritma ini mungkin tidak menemukan jalur optimal. Algoritma ini bisa mengarah ke solusi suboptimal jika heuristiknya tidak konsisten atau bisa menyesatkan.

3.3 Analisis dan Implementasi dalam Algoritma A*

A* menggabungkan kelebihan dari algoritma UCS dan GBFS. Algoritma ini menggunakan nilai gabungan antara biaya kumulatif ($g(n)$) dan estimasi heuristik ($h(n)$) untuk memilih simpul berikutnya untuk dieksplorasi. Ini berarti bahwa A* dapat menemukan jalur optimal dengan cara yang lebih efisien daripada UCS, karena menggabungkan biaya kumulatif dengan prediksi heuristik.

A. Langkah-langkah A* :

1. Masukkan simpul awal ke dalam *priority queue* dengan nilai $f(n) = g(n) + h(n)$.
2. Setiap iterasi, keluarkan simpul dengan nilai $f(n)$ terendah dari *priority queue*.
3. Jika simpul adalah tujuan, jalur ditemukan.
4. Jika simpul sudah dikunjungi, lanjutkan ke simpul berikutnya.
5. Jika belum, tandai simpul sebagai dikunjungi dan tambahkan semua tetangganya dengan menghitung nilai $f(n)$ untuk masing-masing.

B. $f(n)$, $g(n)$, dan $h(n)$:

1. $f(n)$: Total estimasi biaya mencapai tujuan melalui simpul n. Dihitung sebagai $g(n) + h(n)$. Dari source code yang terlampir dibawah, $f(n)$ dihitung sebagai `utility.heuristic(n.getWord(), endWord) + n.getCost()`.
2. $g(n)$: Biaya kumulatif dari simpul awal ke simpul saat ini. Dalam kasus *word ladder*, setiap transisi memiliki biaya 1, sehingga $g(n)$ adalah jumlah langkah dari simpul awal ke simpul saat ini.
3. $h(n)$: Sama seperti GBFS, $h(n)$ dalam A* adalah estimasi biaya dari simpul saat ini ke simpul tujuan. Dari source code terlampir di bawah potongan code `utility.heuristic(n.getWord(), endWord)` menghitung jumlah perbedaan huruf antara kata saat ini dan kata tujuan.

C. Apakah heuristik pada A* admissible?

Heuristik dianggap admissible jika tidak melebih-lebihkan biaya aktual untuk mencapai simpul tujuan. Pada penyelesaian *word ladder*, heuristik yang saya gunakan adalah perbedaan jumlah huruf berbeda antara dua kata. Ini dapat dilihat pada potongan code `utility.heuristic(n.getWord(), endWord)`. Karena `utility.heuristic()` menghitung jumlah perbedaan huruf antara kata saat ini dan kata tujuan, maka heuristik ini adalah admissible. Ini karena jumlah perbedaan huruf merupakan estimasi minimal jarak antar kata (karena setiap langkah hanya mengubah satu huruf), dan tidak mungkin melebih-lebihkan jarak sebenarnya.

D. Apakah A* lebih efisien daripada UCS dalam kasus word ladder?

Secara teoritis, A* lebih efisien daripada UCS. A* menggunakan heuristik untuk memandu pencarian, memungkinkan eksplorasi lebih efisien dengan memprioritaskan jalur yang tampak lebih menjanjikan, sehingga mengurangi jumlah simpul yang harus dieksplorasi. Sedangkan UCS hanya mempertimbangkan biaya kumulatif, sehingga mungkin mengeksplorasi lebih banyak simpul sebelum menemukan solusi. Dengan heuristik yang admissible, A* lebih efisien dalam menemukan solusi dengan lebih sedikit simpul yang dieksplorasi.

3.4 Source Code

Struktur folder yang saya buat yaitu:

```
Tucil3_13522070
├── bin
├── doc
│   └── Tucil3_13522070.pdf
├── img
└── src
    └── wordladder
        ├── dictionary
        ├── image
        └── jafafx-sdk-22.0.1
```



3.4.1 Node.java

File Node.java yang berisikan class Node adalah sebuah class yang mewakili sebuah simpul (node) dalam graf. Class ini memiliki beberapa atribut dan method sebagai berikut:

A. Atribut

- String word: Menyimpan kata atau informasi terkait dengan simpul ini.
- int cost: Menyimpan biaya yang terkait dengan simpul ini, mungkin untuk merepresentasikan jarak atau biaya lainnya.
- Node parent: Referensi ke simpul induk (parent) dalam struktur hierarkis atau graf.
- Set<String> visited: Menyimpan kumpulan kata yang telah dikunjungi (misalnya, untuk menghindari siklus).

B. Method

- Node(String word, int cost, Node parent): Konstruktor yang digunakan untuk membuat objek Node. Menginisialisasi word, cost, parent, dan juga membuat visited sebagai HashSet baru.
- String getWord(): Mengembalikan nilai dari atribut word.
- int getCost(): Mengembalikan nilai dari atribut cost.
- Node getParent(): Mengembalikan nilai dari atribut parent.
- Set<String> getVisited(): Mengembalikan objek Set<String> yang berisi kata-kata yang telah dikunjungi.
- void setVisited(Set<String> visited): Mengatur atribut visited dengan set yang diberikan sebagai parameter.

Node.java

```

import java.util.*;

public class Node {
    private String word;
    private int cost;

```

```

private Node parent;
private Set<String> visited;

public Node(String word, int cost, Node parent) {
    this.word = word;
    this.cost = cost;
    this.parent = parent;
    this.visited = new HashSet<>();
}

public String getWord() {
    return word;
}

public int getCost() {
    return cost;
}

public Node getParent() {
    return parent;
}

public Set<String> getVisited() {
    return visited;
}

public void setVisited(Set<String> visited) {
    this.visited = visited;
}

```

3.4.2 Utility.java

Utility adalah sebuah class yang berisi fungsi-fungsi utilitas untuk melakukan pengolahan node ketetanggaan pada *dictionary* yang digunakan dan melakukan perhitungan heuristik. Class ini memiliki dua method statis yang signifikan.

- Method pertama, `getNeighbors(String word, Set<String> dictionary)`, digunakan untuk menemukan daftar kata yang berbeda satu karakter dari suatu *word* dan terdapat di dalam kumpulan kata yang diberikan (*dictionary*). Dalam prosesnya, method ini mengubah kata menjadi array karakter, lalu mengganti setiap karakter dengan setiap huruf dari 'a' hingga 'z', sambil memastikan bahwa hasil penggantian ini bukan karakter yang sama. Jika kata yang dihasilkan dari penggantian ini ada dalam *dictionary*, maka kata tersebut ditambahkan ke daftar

tetangga (neighbors). Method ini mengembalikan daftar kata-kata yang memenuhi kriteria ini.

- Method kedua, heuristic(String word, String endWord), digunakan untuk menghitung jumlah perbedaan karakter antara dua kata dengan membandingkan karakter pada posisi yang sama di kedua kata. Jika ada karakter yang berbeda, penghitung perbedaan (diff) ditambah satu. Nilai akhir dari diff mengindikasikan tingkat perbedaan antara word dan endWord, yang dapat digunakan sebagai heuristik dalam algoritma pencarian atau penilaian.

Utility.java

```
import java.util.*;  
  
public class Utility {  
    public static List<String> getNeighbors(String word, Set<String> dictionary) {  
        List<String> neighbors = new ArrayList<>();  
        char[] wordArr = word.toCharArray();  
  
        for (int i = 0; i < wordArr.length; i++) {  
            char originalChar = wordArr[i];  
            for (char c = 'a'; c <= 'z'; c++) {  
                if (c != originalChar) {  
                    wordArr[i] = c;  
                    String newWord = new String(wordArr);  
                    if (dictionary.contains(newWord)) {  
                        neighbors.add(newWord);  
                    }  
                }  
            }  
            wordArr[i] = originalChar;  
        }  
        return neighbors;  
    }  
  
    public static int heuristic(String word, String endWord) {  
        int diff = 0;  
        for (int i = 0; i < word.length(); i++) {  
            if (word.charAt(i) != endWord.charAt(i)) {  
                diff++;  
            }  
        }  
        return diff;  
    }  
}
```

3.4.3 Result.java

Result adalah class yang mewakili hasil dari operasi pencarian solusi dari *word ladder*. Class ini memiliki beberapa atribut dan method berikut:

A. Atribut

- List<String> path: Menyimpan daftar kata yang mewakili jalur yang ditemukan dalam proses pencarian.
- int nodesVisited: Menyimpan jumlah simpul yang dikunjungi selama proses pencarian.

B. Method

- Result(List<String> path, int nodesVisited): Konstruktor yang menginisialisasi atribut path dan nodesVisited dengan nilai yang diberikan sebagai parameter.
- List<String> getPath(): Method ini mengembalikan daftar path yang mewakili jalur hasil dari proses pencarian.
- int getNodesVisited(): Method ini mengembalikan jumlah simpul yang dikunjungi selama proses pencarian.

Result.java

```
import java.util.List;

public class Result {
    private List<String> path;
    private int nodesVisited;

    public Result(List<String> path, int nodesVisited) {
        this.path = path;
        this.nodesVisited = nodesVisited;
    }

    public List<String> getPath() {
        return path;
    }

    public int getNodesVisited() {
        return nodesVisited;
    }
}
```

3.4.4 WordDictionary.java

WordDictionary adalah class yang bertanggung jawab untuk memuat kamus kata dari sumber eksternal oracle-dictionary.txt. Pada class WordDictionary hanya terdapat 1 method saja yaitu Set<String> loadDictionary(): Method ini menggunakan

BufferedReader untuk membaca file secara bertahap, memastikan setiap baris kata diubah menjadi lowercase dan menambahkannya ke HashSet untuk menghindari duplikasi. Method ini menggunakan struktur try-with-resources untuk memastikan bahwa resource BufferedReader ditutup secara otomatis setelah selesai digunakan. Jika terjadi kesalahan saat membaca file, pesan error akan ditampilkan di konsol. Method ini mengembalikan objek Set<String> yang berisi semua kata yang berhasil dimuat dari file.

```
WordDictionary.java

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashSet;
import java.util.Set;

public class WordDictionary {
    public static Set<String> loadDictionary() {
        String filePath = "dictionary/oracle-dictionary.txt"; // File kamus utama dari oracle

        Set<String> dictionary = new HashSet<>();

        try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            String line;
            while ((line = br.readLine()) != null) {
                dictionary.add(line.trim().toLowerCase()); // Memastikan semua kata menjadi lowercase
            }
        } catch (IOException e) {
            System.err.println("Error loading dictionary: " + e.getMessage());
        }

        return dictionary;
    }
}
```

3.4.5 UCS.java

Class UCS memiliki method `findPath(String startWord, String endWord)` yang melakukan pencarian dengan algoritma Uniform Cost Search, menggunakan priority queue yang diatur berdasarkan biaya (`cost`). Method ini memulai pencarian dengan menambahkan simpul awal (`startWord`) ke antrian dengan biaya 0. Dalam loop, method ini mengeluarkan simpul dengan biaya terendah dari queue dan memeriksa apakah kata dalam simpul tersebut adalah kata tujuan (`endWord`). Jika ya, maka method

membuat jalur dengan mengikuti referensi induk (`parent`) dari simpul tersebut dan mengembalikan objek `Result` dengan jalur dan jumlah simpul yang dikunjungi. Jika tidak, method menambahkan semua tetangga dari kata saat ini ke antrian dengan biaya yang diperbarui, dengan syarat node tetangga tersebut belum dikunjungi sebelumnya. Jika simpul yang dikeluarkan sudah dikunjungi, loop melanjutkan ke iterasi berikutnya. Jika queue habis dan tidak ditemukan jalur, method mengembalikan `Result` dengan jalur kosong dan jumlah simpul yang dikunjungi.

```
UCS.java
```

```
import java.util.*;  
  
public class UCS {  
    private Set<String> dictionary;  
  
    public UCS(Set<String> dictionary) {  
        this.dictionary = dictionary;  
    }  
  
    public Result findPath(String startWord, String endWord) {  
        PriorityQueue<Node> queue = new  
PriorityQueue<>(Comparator.comparingInt(Node::getCost));  
        Set<String> visited = new HashSet<>();  
        queue.add(new Node(startWord, 0, null));  
  
        int nodesVisited = 0;  
  
        while (!queue.isEmpty()) {  
            Node current = queue.poll();  
  
            if (visited.contains(current.getWord())) {  
                continue;  
            }  
  
            visited.add(current.getWord());  
            nodesVisited++;  
            if (current.getWord().equals(endWord)) {  
                List<String> path = new ArrayList<>();  
                Node node = current;  
                while (node != null) {  
                    path.add(0, node.getWord());  
                    node = node.getParent();  
                }  
                return new Result(path, nodesVisited);  
            }  
        }  
    }  
}
```

```

        List<String> neighbors = Utility.getNeighbors(current.getWord(),
dictionary);

        for (String neighbor : neighbors) {
            if (!visited.contains(neighbor)) {
                queue.add(new Node(neighbor, current.getCost() + 1,
current));
            }
        }
    }

    // Jika tidak ada jalur yang ditemukan
    return new Result(Collections.emptyList(), nodesVisited);
}
}

```

3.4.6 GBFS.java

Class `GBFS` memiliki method `findPath(String startWord, String endWord)` yang melakukan pencarian dengan algoritma Greedy Best-First Search, menggunakan `priority queue` yang diatur berdasarkan heuristik (perbedaan karakter antara kata saat ini dan kata tujuan). Method ini dimulai dengan menambahkan simpul awal ke `queue` dan, dalam loop, mengeluarkan simpul dengan heuristik terendah. Jika kata dalam simpul sama dengan `endWord`, jalur dikembalikan dengan mengikuti simpul `parent` hingga mencapai simpul awal, dan objek `Result` dikembalikan dengan jalur dan jumlah simpul yang dikunjungi. Jika tidak, method menambahkan semua node tetangga yang belum dikunjungi ke `queue`, berdasarkan heuristik. `Priority queue` ini diatur ulang setiap kali loop dijalankan agar tidak terjadi *backtracking*. Jika tidak ditemukan jalur setelah antrian kosong, `Result` dengan jalur kosong dan jumlah simpul yang dikunjungi dikembalikan.

GBFS.java

```

import java.util.*;

public class GBFS {
    private Set<String> dictionary;

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

    public Result findPath(String startWord, String endWord) {
        PriorityQueue<Node> queue = new

```

```

PriorityQueue<>(Comparator.comparingInt(n ->
    Utility.heuristic(n.getWord(), endWord)));
}

Set<String> visited = new HashSet<>();
queue.add(new Node(startWord, 0, null));

int nodesVisited = 0;

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

    if (visited.contains(current.getWord())) {
        continue;
    }

    visited.add(current.getWord());
    nodesVisited++;

    if (current.getWord().equals(endWord)) {
        List<String> path = new ArrayList<>();
        Node node = current;
        while (node != null) {
            path.add(0, node.getWord());
            node = node.getParent();
        }
        return new Result(path, nodesVisited);
    }

    queue.clear();
    List<String> neighbors = Utility.getNeighbors(current.getWord(),
dictionary);
    for (String neighbor : neighbors) {
        if (!visited.contains(neighbor)) {
            queue.add(new Node(neighbor, 0, current));
        }
    }
}

// No Solution
return new Result(Collections.emptyList(), nodesVisited);
}
}

```

3.4.7 AStar.java

Class `AStar` menggunakan metode `findPath(String startWord, String endWord)` yang mengimplementasikan algoritma A* Search, menggunakan *priority queue* yang diatur berdasarkan penjumlahan antara *cost* dan heuristik. Method ini sebenarnya adalah gabungan antara UCS dan GBFS. Simpul dengan biaya dan heuristik terendah dikeluarkan, dan jika kata dalam simpul adalah kata tujuan, jalur dikembalikan setelah dibuat dengan mengikuti simpul *parent*. Jika tidak, node tetangga yang belum dikunjungi ditambahkan ke *queue* dengan biaya yang diperbarui. Jika *queue* kosong tanpa menemukan jalur, `Result` dengan jalur kosong dan jumlah simpul yang dikunjungi dikembalikan.

```
AStar.java
```

```
import java.util.*;  
  
public class AStar {  
    private Set<String> dictionary;  
  
    public AStar(Set<String> dictionary) {  
        this.dictionary = dictionary;  
    }  
  
    public Result findPath(String startWord, String endWord) {  
        PriorityQueue<Node> queue = new  
PriorityQueue<>(Comparator.comparingInt(n ->  
            Utility.heuristic(n.getWord(), endWord) + n.getCost()));  
  
        Set<String> visited = new HashSet<>();  
        queue.add(new Node(startWord, 0, null));  
  
        int nodesVisited = 0;  
  
        while (!queue.isEmpty()) {  
            Node current = queue.poll();  
  
            if (visited.contains(current.getWord())) {  
                continue;  
            }  
  
            nodesVisited++;  
  
            if (current.getWord().equals(endWord)) {  
                List<String> path = new ArrayList<>();  
                Node node = current;  
                while (node != null) {
```

```

        path.add(0, node.getWord());
        node = node.getParent();
    }

    return new Result(path, nodesVisited);
}

visited.add(current.getWord());

List<String> neighbors = Utility.getNeighbors(current.getWord(),
dictionary);

for (String neighbor : neighbors) {
    if (!visited.contains(neighbor)) {
        queue.add(new Node(neighbor, current.getCost() + 1,
current));
    }
}
}

return new Result(Collections.emptyList(), nodesVisited);
}
}

```

3.4.8 Main.java

Main.java adalah file utama yang digunakan untuk menjalankan program *word ladder solver* ini. Pertama pengguna akan diminta untuk memasukkan kata awal (`startWord`) dan kata akhir (`endWord`). Jika panjang kedua kata tidak sama atau jika keduanya identik, program akan mengeluarkan validasi dan berhenti. Program kemudian memuat kamus dari `WordDictionary` dan memeriksa apakah kedua kata ada dalam kamus. Jika tidak, program juga akan menghentikan eksekusi dengan pesan kesalahan.

Setelah validasi awal, program meminta pengguna untuk memilih algoritma pencarian. Tiga algoritma yang tersedia adalah Uniform Cost Search (UCS), Greedy Best-First Search (GBFS), dan A* Search. Pengguna memilih algoritma dengan memasukkan angka (1 untuk UCS, 2 untuk GBFS, atau 3 untuk A*). Jika pengguna memasukkan angka yang tidak valid, program menampilkan validasi dan keluar.

Sebelum menjalankan algoritma, program akan mengukur penggunaan memori dan waktu eksekusi. `Runtime` digunakan untuk mendapatkan informasi tentang memori sebelum dan sesudah proses pencarian. Program juga akan mencatat waktu mulai untuk mengukur berapa lama waktu yang diperlukan oleh algoritma yang dipilih. Program kemudian menjalankan algoritma yang sesuai dengan yang dipilih. Masing-masing algoritma mencari jalur dari kata awal ke kata akhir dan mengembalikan objek `Result`, yang berisi jalur yang ditemukan dan jumlah simpul yang dikunjungi selama pencarian.

Setelah algoritma selesai, program mencatat waktu akhir untuk menghitung waktu eksekusi dan mengukur penggunaan memori setelah eksekusi. Program kemudian menampilkan jumlah simpul yang dikunjungi, waktu eksekusi dalam milidetik, dan penggunaan memori dalam megabyte. Jika jalur ditemukan, program menampilkan daftar kata yang membentuk jalur serta jumlah langkah yang diperlukan untuk mencapai kata akhir dari kata awal yaitu `path.size() - 1`. Jika tidak ada jalur yang ditemukan, program mengeluarkan pesan bahwa solusi tidak ditemukan.

Main.java

```
import java.util.Scanner;
import java.util.List;
import java.util.Set;
import java.util.concurrent.TimeUnit;

public class Main {

    // ANSI color codes for terminal output
    private static final String RED = "\u001B[31m";
    private static final String GREEN = "\u001B[32m";
    private static final String YELLOW = "\u001B[33m";
    private static final String BLUE = "\u001B[34m";
    private static final String PURPLE = "\u001B[35m";
    private static final String CYAN = "\u001B[36m";
    private static final String WHITE = "\u001B[37m";
    private static final String RESET = "\u001B[0m";

    private static void printWordAzul() {
        System.out.println(PURPLE + " _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ " + RESET);
        System.out.println(PURPLE + " \\ \\ \\ \\ / / ( ) \\ \\ ( ) + _ ) \\ \\ / ( ) \\ \\ / / _ | _ ) | _ | _ " + RESET);
        System.out.println(PURPLE + " \\ \\ _ / \\ \\ _ / _ / _ / _ / _ / _ / _ / _ / _ | " + RESET);
        System.out.println();
    }

    private static void displayLoading() {
        // ASCII frames to simulate battery charging
        String[] frames = {
            BLUE + "[ _ ]" + RESET,
            BLUE + "[ L _ ]" + RESET,
            BLUE + "[ LO _ ]" + RESET,
            BLUE + "[ LOA _ ]" + RESET,
            BLUE + "[ LOAD _ ]" + RESET,
        };
    }
}
```

```

        BLUE + "[LOADI      ]" + RESET,
        BLUE + "[LOADIN     ]" + RESET,
        BLUE + "[LOADING    ]" + RESET,
        BLUE + "[LOADING.   ]" + RESET,
        BLUE + "[LOADING..  ]" + RESET,
        BLUE + "[LOADING...]"+ RESET,
    } ;

    System.out.print("Loading ");
    for (String frame : frames) {
        System.out.print("\r" + frame);
        try {
            TimeUnit.MILLISECONDS.sleep(200);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
    System.out.println();
}

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

    displayLoading();
    printWordAzul();

    boolean continueProgram = true;

    while (continueProgram) {
        System.out.print(WHITE + "Enter start word: " + RESET);
        String startWord = scanner.next();
        System.out.print(WHITE + "Enter end word: " + RESET);
        String endWord = scanner.next();

        // Validate the input
        while (startWord.length() != endWord.length()) {
            System.out.println(RED + "Start word and end word must have
the same length." + RESET);
            System.out.print(WHITE + "Enter start word: " + RESET);
            startWord = scanner.next();
            System.out.print(WHITE + "Enter end word: " + RESET);
            endWord = scanner.next();
        }
    }
}

```

```

        if (startWord.equalsIgnoreCase(endWord)) {
            System.out.println(RED + "Start word and end word are the
same: " + startWord + RESET);
            continue;
        }

        // Load dictionary
        Set<String> dictionary = WordDictionary.loadDictionary();

        while (!dictionary.contains(startWord.toLowerCase()) ||

!dictionary.contains(endWord.toLowerCase())) {
            if (!dictionary.contains(startWord.toLowerCase())) {
                System.out.println(RED + "Start word " + startWord + " not
in dictionary." + RESET);
            } else if (!dictionary.contains(endWord.toLowerCase())) {
                System.out.println(RED + "End word " + endWord + " not in
dictionary." + RESET);
            } else {
                System.out.println(RED + "Start word and/or end word not
in dictionary." + RESET);
            }
            System.out.print(WHITE + "Enter start word: " + RESET);
            startWord = scanner.next();
            System.out.print(WHITE + "Enter end word: " + RESET);
            endWord = scanner.next();
        }

        // Algorithm choice
        System.out.print(BLUE + "Choose algorithm (1-UCS, 2-GBFS, 3-A*): " +
+ RESET);
        int algoChoice = scanner.nextInt();

        while (algoChoice < 1 || algoChoice > 3) {
            System.out.println(RED + "Invalid choice. Please select 1 for
UCS, 2 for GBFS, or 3 for A*." + RESET);
            System.out.print(BLUE + "Choose algorithm (1-UCS, 2-GBFS,
3-A*): " + RESET);
            algoChoice = scanner.nextInt();
        }

        Runtime runtime = Runtime.getRuntime();
        runtime.gc(); // Garbage collection before measurement
        long memoryBefore = runtime.totalMemory() - runtime.freeMemory();
// Memory used
    }
}

```

```

        List<String> path = null;
        int nodesVisited = 0;
        long startTime = System.currentTimeMillis();

        switch (algoChoice) {
            case 1:
                UCS ucs = new UCS(dictionary);
                Result ucsResult = ucs.findPath(startWord.toLowerCase(),
endWord.toLowerCase());
                path = ucsResult.getPath();
                nodesVisited = ucsResult.getNodesVisited();
                break;
            case 2:
                GBFS gbfs = new GBFS(dictionary);
                Result gbfsResult = gbfs.findPath(startWord.toLowerCase(),
endWord.toLowerCase());
                path = gbfsResult.getPath();
                nodesVisited = gbfsResult.getNodesVisited();
                break;
            case 3:
                AStar astar = new AStar(dictionary);
                Result astarResult =
astar.findPath(startWord.toLowerCase(), endWord.toLowerCase());
                path = astarResult.getPath();
                nodesVisited = astarResult.getNodesVisited();
                break;
        }

        long endTime = System.currentTimeMillis();
        long executionTime = endTime - startTime; // Calculate execution
time

        long memoryAfter = runtime.totalMemory() - runtime.freeMemory();
// Memory used
        long memoryUsed = memoryAfter - memoryBefore; // Calculate memory
usage
        double memoryUsedMB = memoryUsed / (1024.0 * 1024.0); // Convert
to MB
        String memoryUsedFormatted = String.format("%.2f", memoryUsedMB);

        // Output results
        System.out.println(YELLOW + "Nodes visited: " + nodesVisited +
RESET);
    }
}

```

```

        System.out.println(BLUE + "Execution time (ms): " + executionTime
+ RESET);
        System.out.println(CYAN + "Memory used (MB): " +
memoryUsedFormatted + RESET);

        if (path == null || path.isEmpty()) {
            System.out.println(RED + "No path solution found." + RESET);
        } else {
            int steps = path.size() - 1;
            System.out.println(PURPLE + "Steps needed: " + steps + RESET);
            System.out.println(CYAN + "Path:" + RESET);
            for (int i = 0; i < path.size(); i++) {
                String pathElementColor = GREEN;
                System.out.println(pathElementColor + (i + 1) + ". " +
path.get(i) + RESET);
            }
        }

        boolean validResponse = false;
        while (!validResponse) {
            System.out.print(WHITE + "Do you want to run another test?
(y/n): " + RESET);
            String continueInput = scanner.next();
            if (continueInput.trim().equalsIgnoreCase("y")) {
                continueProgram = true;
                validResponse = true;
            } else if (continueInput.trim().equalsIgnoreCase("n")) {
                continueProgram = false;
                validResponse = true;
            } else {
                System.out.println(RED + "Invalid response. Please enter
'y' or 'n'." + RESET);
            }
        }

        System.out.println(BLUE + "Program exited. Goodbye!" + RESET);
    }
}

```

3.4.9 Bonus GUI.java

Penjelasan mengenai program Bonus GUI.java ini akan dijelaskan di Bab 4.4.

GUI.java

```

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Dictionary;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.scene.paint.*;
import javafx.scene.text.*;
import javafx.stage.Stage;
import javafx.geometry.*;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;

public class GUI extends Application {

    @Override
    public void start(Stage primaryStage) {
        VBox mainLayout = new VBox(20);
        mainLayout.setPadding(new Insets(20, 20, 20, 20));
        mainLayout.setAlignment(Pos.TOP_CENTER);

        // Header
        ImageView headerImage = new ImageView(new
Image("images/wordazul.png"));
        headerImage.setFitWidth(400);
        headerImage.setPreserveRatio(true);
        mainLayout.getChildren().add(headerImage);

        Label welcomeLabel = new Label("Selamat Datang di WordAzul!");
        welcomeLabel.setFont(Font.font("Monospace", FontWeight.BOLD, 24));
        mainLayout.getChildren().add(welcomeLabel);

        // Deskripsi WordAzul
        Label descriptionLabel = new Label(
            "WordAzul adalah platform yang dapat memberikan solusi untuk
teka-teki word ladder. Di Word Ladder, kalian ditantang untuk menghubungkan
dua kata dengan mengubah\n"
            + "satu huruf pada satu waktu, menciptakan rangkaian kata yang

```

```

menarik dan logis. Setiap perubahan membawa kalian lebih dekat ke tujuan,
menawarkan pengalaman yang\n"
        + "mendidik dan menghibur. Dengan berbagai tingkat kesulitan,
WordAzul cocok untuk membantu kalian yang mengalami kesulitan. Antarmuka yang
menarik dan ramah pengguna\n"
        + "membuat platform ini menjadi tempat yang sempurna untuk
mengasah keterampilan kosa kata kalian sambil bersenang-senang.\n"
    );
    descriptionLabel.setPrefWidth(1500);
    descriptionLabel.setFont(Font.font("Monospace", FontWeight.SEMI_BOLD,
14));
    mainLayout.getChildren().add(descriptionLabel);

    // How To Use
    Label howToUseLabel = new Label("How to Use");
    howToUseLabel.setFont(Font.font("Monospace", FontWeight.BOLD, 20));
    mainLayout.getChildren().add(howToUseLabel);

    Label stepsLabel = new Label(
        "- Masukkan start word dan end word yang diinginkan.\n"
        + "- Pilih algoritma route planning yang ingin digunakan.\n"
        + "- Klik tombol search, lalu hasilnya akan segera muncul di
bagian bawah."
    );
    mainLayout.getChildren().add(stepsLabel);
    stepsLabel.setFont(Font.font("Monospace", FontWeight.SEMI_BOLD, 14));

    // Main Game Solver
    HBox inputLayout = new HBox(20);
    inputLayout.setAlignment(Pos.CENTER);

    ImageView gameImage = new ImageView(new Image("images/bossteam.png"));
    gameImage.setFitWidth(250);
    gameImage.setPreserveRatio(true);
    mainLayout.getChildren().add(gameImage);

    Label startWordLabel = new Label("Start Word:");
    startWordLabel.setFont(Font.font("Monospace", FontWeight.BOLD, 16));

    TextField startWordField = new TextField();
    startWordField.setPromptText("Enter start word");
    startWordField.setPrefWidth(200);
    startWordField.setFont(Font.font("Monospace", FontWeight.BOLD, 14));

```

```

Label endWordLabel = new Label("End Word:");
endWordLabel.setFont(Font.font("Monospace", FontWeight.BOLD, 16));

TextField endWordField = new TextField();
endWordField.setPromptText("Enter end word");
endWordField.setPrefWidth(200);
endWordField.setFont(Font.font("Monospace", FontWeight.BOLD, 14));

inputLayout.getChildren().addAll(startWordLabel, startWordField,
endWordLabel, endWordField);
mainLayout.getChildren().add(inputLayout);

HBox algorithmLayout = new HBox(20);
algorithmLayout.setAlignment(Pos.CENTER);

Label algorithmLabel = new Label("Choose Algorithm: ");
algorithmLabel.setFont(Font.font("Monospace", FontWeight.BOLD, 16));

ComboBox<String> algorithmDropdown = new ComboBox<>();
algorithmDropdown.getItems().addAll("UCS", "GBFS", "A*");

algorithmLayout.getChildren().addAll(algorithmLabel,
algorithmDropdown);
mainLayout.getChildren().add(algorithmLayout);

Button searchButton = new Button("Search");
searchButton.setFont(Font.font("Monospace", FontWeight.BOLD, 16));
mainLayout.getChildren().add(searchButton);

TextArea resultBox = new TextArea();
resultBox.setEditable(false);
resultBox.setPrefHeight(300);
resultBox.setWrapText(true);
resultBox.setFont(Font.font("Monospace", FontWeight.BOLD, 14));
mainLayout.getChildren().add(resultBox);

searchButton.setOnAction(event -> {
    String startWord = startWordField.getText().trim();
    String endWord = endWordField.getText().trim();
    String selectedAlgorithm = algorithmDropdown.getValue();

    if (selectedAlgorithm == null) {
        resultBox.setText("No algorithm selected.");
        return;
}

```

```

        }

        if (startWord.isEmpty() || endWord.isEmpty()) {
            resultBox.setText("Enter the start word and the end word!");
            return;
        }

        if (startWord.length() != endWord.length()) {
            resultBox.setText("Start word and end word must be of the same
length.");
            return;
        }

        if (startWord.equalsIgnoreCase(endWord)) {
            resultBox.setText("Start word and end word are the same: " +
startWord);
            return;
        }

        Set<String> dictionary = WordDictionary.loadDictionary();

        if (!dictionary.contains(startWord.toLowerCase())) {
            resultBox.setText("Start word " + startWord + " not in
dictionary.");
            return;
        }

        } else if (!dictionary.contains(endWord.toLowerCase())) {
            resultBox.setText("End word " + endWord + " not in
dictionary.");
            return;
        }

        Runtime runtime = Runtime.getRuntime();
        runtime.gc();
        long memoryBefore = runtime.totalMemory() - runtime.freeMemory();

        List<String> path = null;
        int nodesVisited = 0;
        long startTime = System.currentTimeMillis();

        switch (selectedAlgorithm) {
            case "UCS":
                UCS ucs = new UCS(dictionary);
                Result ucsResult = ucs.findPath(startWord.toLowerCase(),

```

```

endWord.toLowerCase());
        path = ucsResult.getPath();
        nodesVisited = ucsResult.getNodesVisited();
        break;
    case "GBFS":
        GBFS gbfs = new GBFS(dictionary);
        Result gbfsResult = gbfs.findPath(startWord.toLowerCase(),
endWord.toLowerCase());
        path = gbfsResult.getPath();
        nodesVisited = gbfsResult.getNodesVisited();
        break;
    case "A*":
        AStar astar = new AStar(dictionary);
        Result astarResult =
astar.findPath(startWord.toLowerCase(), endWord.toLowerCase());
        path = astarResult.getPath();
        nodesVisited = astarResult.getNodesVisited();
        break;
    default:
        resultBox.setText("Invalid Algorithm");
        return;
    }

    long endTime = System.currentTimeMillis();
    long executionTime = endTime - startTime;

    long memoryAfter = runtime.totalMemory() - runtime.freeMemory();
    long memoryUsed = memoryAfter - memoryBefore;
    double memoryUsedMB = memoryUsed / (1024.0 * 1024.0);
    String memoryUsedFormatted = String.format("%.2f", memoryUsedMB);

    StringBuilder result = new StringBuilder();
    result.append("Nodes visited: " + nodesVisited + "\n");
    result.append("Execution time (ms): " + executionTime + "\n");
    result.append("Memory used (MB): " + memoryUsedFormatted + "\n");
    if (path == null || path.isEmpty()) {
        result.append("No Path Found!");
        resultBox.setText(result.toString());
        return;
    }

    int steps = path.size() - 1;
    result.append("Steps needed: " + steps + "\n");
    result.append("Path:\n");

```

```

        for (int i = 0; i < path.size(); i++) {
            result.append((i + 1) + ". " + path.get(i) + "\n");
        }

        resultBox.setText(result.toString());
    });

    // Penjelasan tentang algoritma
    Label algorithmExplanationTitle = new Label("Get to Know the
Algorithm");
    algorithmExplanationTitle.setFont(Font.font("Monospace",
FontWeight.BOLD, 20));
    Label algorithmExplanation = new Label(
        "1. UCS: Uniform Cost Search\n"
        + "Uniform Cost Search (UCS) adalah algoritma yang menggunakan
pendekatan Breadth-First Search (BFS), namun dengan biaya yang seragam.
Algoritma ini mencari jalur \n"
        + "dengan biaya paling rendah antara titik awal dan titik tujuan.
Langkah-Langkah: \n"
        + "- Mulai dari titik awal.\n"
        + "- Telusuri semua tetangga dengan biaya terendah.\n"
        + "- Pilih node dengan biaya terendah untuk dilanjutkan.\n"
        + "- Ulangi hingga mencapai titik tujuan.\n"
        + "Kelebihan: Menemukan jalur dengan biaya terendah.\n"
        + "Kekurangan: Mungkin lambat pada graf yang besar.\n"
        + "\n 2. GBFS: Greedy Best First Search Search\n"
        + "Greedy Best-First Search menggunakan heuristik untuk menentukan
node mana yang akan dijelajahi terlebih dahulu. Algoritma ini tidak
memperhitungkan biaya\n"
        + "keseluruhan, tetapi fokus pada node yang terlihat paling dekat
dengan tujuan. Langkah-Langkah: \n"
        + "- Mulai dari titik awal.\n"
        + "- Gunakan heuristik untuk mengevaluasi tetangga.\n"
        + "- Pilih node dengan nilai heuristik terendah.\n"
        + "- Ulangi hingga mencapai titik tujuan.\n"
        + "Kelebihan: Cepat pada graf yang sederhana.\n"
        + "Kekurangan: Bisa menghasilkan jalur yang tidak optimal.\n"
        + "\n 3. Algoritma A*\n"
        + "A* Search adalah kombinasi UCS dan Greedy Best-First Search.
Algoritma ini menggunakan biaya total dari titik awal hingga tujuan,
memperhitungkan biaya dan\n"
        + "heuristik. Langkah-langkah: \n"
        + "- Mulai dari titik awal.\n"
        + "- Gunakan heuristik untuk mengevaluasi tetangga.\n"
    );
}

```

```

        + "- Pilih node dengan biaya total terendah.\n"
        + "- Ulangi hingga mencapai titik tujuan.\n"
        + "Kelebihan: Biasanya menemukan jalur optimal.\n"
        + "Kekurangan: Bisa lebih lambat dari Greedy Best-First Search.\n"
    );
    algorithmExplanation.setTextAlignment(TextAlignment.LEFT);
    algorithmExplanation.setFont(Font.font("Monospace",
FontWeight.SEMI_BOLD, 14));
    mainLayout.getChildren().add(algorithmExplanationTitle);
    mainLayout.getChildren().add(algorithmExplanation);

    // About Me
    Label aboutMeTitle = new Label("About Me");
    aboutMeTitle.setFont(Font.font("Monospace", FontWeight.BOLD, 20));
    mainLayout.getChildren().add(aboutMeTitle);

    ImageView aboutMeImage = new ImageView(new Image("images/azul.png"));
    aboutMeImage.setFitWidth(200);
    aboutMeImage.setPreserveRatio(true);
    mainLayout.getChildren().add(aboutMeImage);

    Label personalInfo = new Label(
        "Nama : Marzuli Suhada M\n"
        + "NIM : 13522070"
    );
    mainLayout.getChildren().add(personalInfo);
    personalInfo.setFont(Font.font("Monospace", FontWeight.BOLD, 14));

    Label quoteLabel = new Label(
        "\\"Computer Science isn't just about machines and code; it's about
solving problems and shaping the future\\" - Azul"
    );
    quoteLabel.setFont(Font.font("Arial", FontWeight.BOLD,
FontPosture.ITALIC, 14));
    mainLayout.getChildren().add(quoteLabel);

    applyGradientBackground(mainLayout);

    ScrollPane scrollPane = new ScrollPane(mainLayout);
    scrollPane.setFitToWidth(true);

    Scene scene = new Scene(scrollPane, 800, 600);
    primaryStage.setScene(scene);
    primaryStage.show();

```

```
}

private void applyGradientBackground(Pane pane) {
    LinearGradient gradient = new LinearGradient(
        0, 0, 0, 1, true, CycleMethod.NO_CYCLE,
        new Stop(0, Color.WHITE),
        new Stop(1, Color.LIGHTBLUE)
    );
    pane.setBackground(new Background(new BackgroundFill(gradient,
CornerRadii.EMPTY, Insets.EMPTY)));
}

public static void main(String[] args) {
    launch(args);
}
}
```

BAB IV

HASIL UJI DAN ANALISIS

Setelah dibuat algoritma penyelesaian menggunakan Algoritma Uniform-Cost Search, Greedy Best-First Search dan A* dilakukan pengujian dengan hasil terlampir dibawah.

4.1 Hasil Uji Program Wajib

Hasil uji program menggunakan CLI dengan tampilan awal seperti berikut.

```
● azulsuhada@Marzulis-MacBook-Air wordladder % javac AStar.java GBFS.java UCS.java WordDictionary.java Node.java Utility.java  
Result.java Main.java  
○ azulsuhada@Marzulis-MacBook-Air wordladder % java Main  
[LOADING...]  
VVV//(( ))|()|| /(( )' / /| |  
VVV//(( ))|()|| /(( )' / /| |  
Enter start word: ■
```

Gambar 4.1 Tampilan Program Menggunakan CLI

4.1.1 Uji Program 1

ANT → BEE	
Uniform-Cost Search (UCS)	[LOADING...] VVV//(()) () /(()' / / VVV//(()) () /(()' / / Enter start word: ANT Enter end word: BEE Choose algorithm (1-UCS, 2-GBFS, 3-A*): 1 Nodes visited: 445 Execution time (ms): 17 Memory used (MB): 2.92 Steps needed: 4 Path: 1. ant 2. ane 3. aye 4. bye 5. bee Do you want to run another test? (y/n): ■

Greedy
Best-First
Search (GBFS)

```
Enter start word: ANT
Enter end word: BEE
Choose algorithm (1-UCS, 2-GBFS, 3-A*): 2
Nodes visited: 34
Execution time (ms): 3
Memory used (MB): 0.35
Steps needed: 33
Path:
1. ant
2. ane
3. one
4. obe
5. ode
6. oke
7. eke
8. uke
9. use
10. ose
11. ole
12. ale
13. ace
14. ice
15. ire
16. are
17. ere
18. ore
19. ope
20. ape
21. age
22. ate
23. ave
24. eve
25. eme
26. ewe
27. awe
28. owe
29. owl
30. awl
31. aal
32. bal
33. bel
34. bee
```

Do you want to run another test? (y/n): █

A*

```
Enter start word: ANT
Enter end word: BEE
Choose algorithm (1-UCS, 2-GBFS, 3-A*): 3
Nodes visited: 23
Execution time (ms): 3
Memory used (MB): 0.31
Steps needed: 4
Path:
1. ant
2. ane
3. aye
4. bye
5. bee
```

Do you want to run another test? (y/n): █

4.1.2 Uji Program 2

GRAB → SNAG

Uniform-Cost Search (UCS)	<pre> Enter start word: GRAB Enter end word: SNAG Choose algorithm (1-UCS, 2-GBFS, 3-A*): 1 Nodes visited: 809 Execution time (ms): 12 Memory used (MB): 4.60 Steps needed: 5 Path: 1. grab 2. crab 3. crag 4. clag 5. slag 6. snag Do you want to run another test? (y/n): █ </pre>
Greedy Best-First Search (GBFS)	<pre> Enter start word: GRAB Enter end word: SNAG Choose algorithm (1-UCS, 2-GBFS, 3-A*): 2 Nodes visited: 9 Execution time (ms): 0 Memory used (MB): 0.08 Steps needed: 8 Path: 1. grab 2. crab 3. crag 4. brag 5. drag 6. frag 7. flag 8. slag 9. snag Do you want to run another test? (y/n): █ </pre>
A*	<pre> Enter start word: GRAB Enter end word: SNAG Choose algorithm (1-UCS, 2-GBFS, 3-A*): 3 Nodes visited: 29 Execution time (ms): 1 Memory used (MB): 0.25 Steps needed: 5 Path: 1. grab 2. crab 3. crag 4. clag 5. slag 6. snag Do you want to run another test? (y/n): █ </pre>

4.1.3 Uji Program 3

TRICK → TREAT

Uniform-Cost Search (UCS)	<pre> Enter start word: TRICK Enter end word: TREAT Choose algorithm (1-UCS, 2-GBFS, 3-A*): 1 Nodes visited: 1453 Execution time (ms): 17 Memory used (MB): 3.15 Steps needed: 7 Path: 1. trick 2. wrick 3. wreck 4. wreak 5. break 6. bread 7. tread 8. treat Do you want to run another test? (y/n): █ </pre>
Greedy Best-First Search (GBFS)	<pre> Enter start word: TRICK Enter end word: TREAT Choose algorithm (1-UCS, 2-GBFS, 3-A*): 2 Nodes visited: 26 Execution time (ms): 0 Memory used (MB): 0.21 Steps needed: 25 Path: 1. trick 2. track 3. tract 4. trait 5. trapt 6. wrapt 7. wraps 8. traps 9. trips 10. tries 11. trees 12. treks 13. tress 14. trets 15. trews 16. treys 17. greys 18. preys 19. prees 20. brees 21. drees 22. frees 23. grees 24. greet 25. great 26. treat Do you want to run another test? (y/n): █ </pre>

A*

```
Enter start word: TRICK
Enter end word: TREAT
Choose algorithm (1-UCS, 2-GBFS, 3-A*): 3
Nodes visited: 112
Execution time (ms): 2
Memory used (MB): 0.82
Steps needed: 7
Path:
1. trick
2. wrick
3. wreck
4. wreak
5. break
6. bread
7. tread
8. treat
Do you want to run another test? (y/n): █
```

4.1.4 Uji Program 4

Uniform-Cost
Search (UCS)

GIMLETS → TREEING

```
Enter start word: GIMLETS
Enter end word: TREEING
Choose algorithm (1-UCS, 2-GBFS, 3-A*): 1
Nodes visited: 6263
Execution time (ms): 73
Memory used (MB): 9.95
Steps needed: 24
Path:
1. gimlets
2. giglets
3. guglets
4. gullets
5. bullets
6. ballets
7. ballers
8. bailers
9. bailees
10. bailles
11. dailies
12. dallies
13. dollies
14. collies
15. collins
16. codlins
17. codling
18. coaling
19. goaling
20. goading
21. grading
22. graying
23. greying
24. greeing
25. treeing
Do you want to run another test? (y/n): █
```

Greedy Best-First Search (GBFS)	<pre>Enter start word: GIMLETS Enter end word: TREEING Choose algorithm (1-UCS, 2-GBFS, 3-A*): 2 Nodes visited: 19 Execution time (ms): 0 Memory used (MB): 0.50 No path solution found. Do you want to run another test? (y/n): █</pre>
A*	<pre>Enter start word: GIMLETS Enter end word: TREEING Choose algorithm (1-UCS, 2-GBFS, 3-A*): 3 Nodes visited: 4021 Execution time (ms): 50 Memory used (MB): 3.13 Steps needed: 24 Path: 1. gimlets 2. giglets 3. guglets 4. gullets 5. cullets 6. cutlets 7. cutlers 8. cutters 9. putters 10. patters 11. pattens 12. lattens 13. lattins 14. mattins 15. matting 16. masting 17. casting 18. costing 19. coating 20. crating 21. grating 22. graying 23. greying 24. greeing 25. treeing Do you want to run another test? (y/n): █</pre>

4.1.5 Uji Program 5

RATIONS → WINTERY

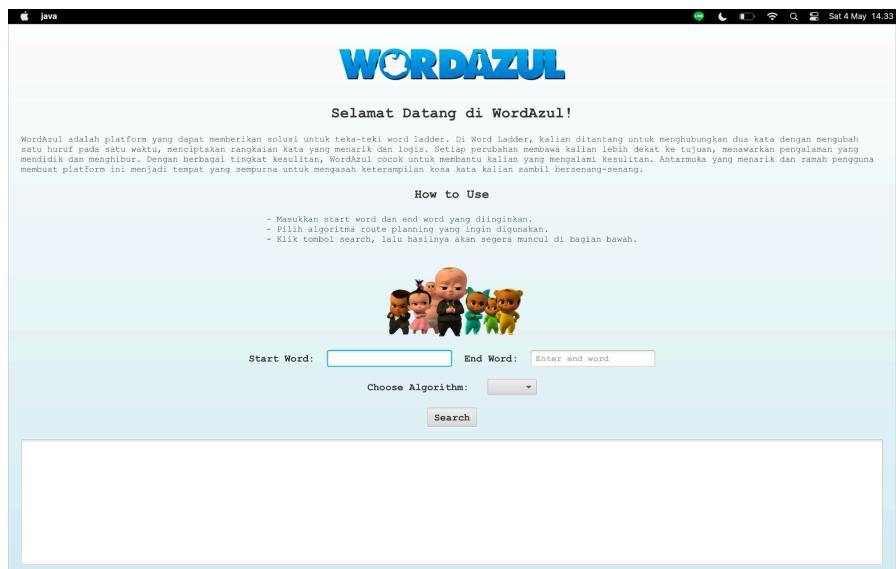
Uniform-Cost Search (UCS)	<pre> Enter start word: RATIONS Enter end word: WINTERY Choose algorithm (1-UCS, 2-GBFS, 3-A*): 1 Nodes visited: 414 Execution time (ms): 6 Memory used (MB): 3.68 Steps needed: 7 Path: 1. rations 2. rattons 3. rattens 4. ratters 5. ranters 6. wanters 7. winters 8. wintery Do you want to run another test? (y/n): █ </pre>
Greedy Best-First Search (GBFS)	<pre> Enter start word: RATIONS Enter end word: WINTERY Choose algorithm (1-UCS, 2-GBFS, 3-A*): 2 Nodes visited: 11 Execution time (ms): 1 Memory used (MB): 0.34 Steps needed: 10 Path: 1. rations 2. rattons 3. rattens 4. ratters 5. ritters 6. bitters 7. fitters 8. hitters 9. hinters 10. winters 11. wintery Do you want to run another test? (y/n): █ </pre>
A*	<pre> Enter start word: RATIONS Enter end word: WINTERY Choose algorithm (1-UCS, 2-GBFS, 3-A*): 3 Nodes visited: 9 Execution time (ms): 0 Memory used (MB): 0.32 Steps needed: 7 Path: 1. rations 2. rattons 3. rattens 4. ratters 5. ranters 6. wanters 7. winters 8. wintery Do you want to run another test? (y/n): █ </pre>

4.1.6 Uji Program 6

BLUBBERY → CLAMPERS	
Uniform-Cost Search (UCS)	<pre> Enter start word: BLUBBERY Enter end word: CLAMPERS Choose algorithm (1-UCS, 2-GBFS, 3-A*): 1 Nodes visited: 30 Execution time (ms): 0 Memory used (MB): 0.32 Steps needed: 5 Path: 1. blubbery 2. blubbers 3. blabbers 4. clabbers 5. clambers 6. clampers Do you want to run another test? (y/n): █ </pre>
Greedy Best-First Search (GBFS)	<pre> Enter start word: BLUBBERY Enter end word: CLAMPERS Choose algorithm (1-UCS, 2-GBFS, 3-A*): 2 Nodes visited: 6 Execution time (ms): 0 Memory used (MB): 0.32 Steps needed: 5 Path: 1. blubbery 2. blubbers 3. clubbers 4. clabbers 5. clambers 6. clampers Do you want to run another test? (y/n): █ </pre>
A*	<pre> Enter start word: BLUBBERY Enter end word: CLAMPERS Choose algorithm (1-UCS, 2-GBFS, 3-A*): 3 Nodes visited: 8 Execution time (ms): 0 Memory used (MB): 0.32 Steps needed: 5 Path: 1. blubbery 2. blubbers 3. blabbers 4. clabbers 5. clambers 6. clampers Do you want to run another test? (y/n): █ </pre>

4.2 Hasil Uji Program Bonus

Hasil Uji pada bagian bonus dilakukan dengan menampilkan GUI yang dibuat dengan javaFX.



Gambar 4.2.1 Tampilan awal GUI

Untuk bagian bonus ini, GUI divisualisasikan seperti gambar diatas. Untuk cara penggunaannya juga sudah tertera saat program dibuka. Saat dijalankan, hasil akan muncul di kolom dibawah tombol search. Hasilnya akan muncul dalam bentuk:

- **Nodes visited:** jumlah kata yang dikunjungi saat dilakukan pencarian jalur.
- **Execution time (ms):** waktu yang dibutuhkan dalam mencari jalur yang paling optimal
- **Memory used (MB):** ruang memori yang dibutuhkan yang sesuai dengan Nodes visited artinya jika Nodes visitednya besar maka ruang memori yang dibutuhkan juga besar
- **Steps needed:** Banyaknya langkah atau perubahan yang harus dilakukan dari kata awal ke kata tujuan. Steps needed dihitung dengan cara mengurangi panjang jalurnya dengan 1 dimana 1 ini adalah kata pertamanya (*start word*).
- **Path:** Jalur yang dihasilkan dari *start word* ke *end word*.

4.2.1 Uji Program 1

ANT → BEE

Uniform-Cost Search (UCS)

The screenshot shows the WordAzul interface with the title "WORDAZUL" and a welcome message "Selamat Datang di WordAzul!". Below it is a "How to Use" section with instructions: "Masukkan start word dan end word yang dinginkan.", "Pilih algoritma route planning yang ingin digunakan.", and "Klik tombol search, lalu hasilnya akan segera muncul di bagian bawah." A cartoon baby icon is displayed above the search form. The search form has "Start Word: ANT" and "End Word: BEE". The "Choose Algorithm:" dropdown is set to "UCS". A "Search" button is present. The results section displays the following information:
Nodes visited: 445
Execution time (ms): 6
Memory used (MB): 3.00
Steps needed: 4
Path:
1. ant
2. ane
3. aye
4. bye
5. bee

Greedy Best-First Search (GBFS)

The screenshot shows the WordAzul interface with the title "WORDAZUL" and a welcome message "Selamat Datang di WordAzul!". Below it is a "How to Use" section with instructions: "Masukkan start word dan end word yang dinginkan.", "Pilih algoritma route planning yang ingin digunakan.", and "Klik tombol search, lalu hasilnya akan segera muncul di bagian bawah." A cartoon baby icon is displayed above the search form. The search form has "Start Word: ANT" and "End Word: BEE". The "Choose Algorithm:" dropdown is set to "GBFS". A "Search" button is present. The results section displays the following information:
Nodes visited: 34
Execution time (ms): 1
Memory used (MB): 0.96
Steps needed: 33
Path:
1. ant
2. ane
3. one
4. obe
5. ode
6. oke
7. eke

A*

java

WORDAZUL

Selamat Datang di WordAzul!

Wordazul adalah platform yang dapat memberikan solusi untuk teka-teki word ladder. Di Word Ladder, kalian ditantang untuk menghubungkan dua kata dengan mengubah satu huruf pada satu waktu, menciptakan rangkulan kata yang menarik dan logis. Setiap perubahan membawa kalian lebih dekat ke tujuan, menawarkan pengalaman yang mendidik dan menghibur. Dengan berbagai tingkat kesulitan, Wordazul cocok untuk membantu kalian yang mengalami kesulitan. Antarmuka yang menarik dan ramah pengguna membuat platform ini menjadi tempat yang sempurna untuk mengasah keterampilan kosa kata kalian sambil bersenang-senang.

How to Use

- Masukkan start word dan end word yang diinginkan.
- Pilih algoritma route planning yang ingin digunakan.
- Klik tombol search, lalu hasilnya akan segera muncul di bagian bawah.



Start Word: ANT End Word: BEE

Choose Algorithm: A*

Search

```
Nodes visited: 23
Execution time (ms): 2
Memory used (MB): 0.96
Steps needed: 4
Path:
1. ant
2. ane
3. eye
4. bye
5. bee
```

4.2.2 Uji Program 2

Uniform-Cost Search (UCS)

GRAB → SNAG

java

WORDAZUL

Selamat Datang di WordAzul!

Wordazul adalah platform yang dapat memberikan solusi untuk teka-teki word ladder. Di Word Ladder, kalian ditantang untuk menghubungkan dua kata dengan mengubah satu huruf pada satu waktu, menciptakan rangkulan kata yang menarik dan logis. Setiap perubahan membawa kalian lebih dekat ke tujuan, menawarkan pengalaman yang mendidik dan menghibur. Dengan berbagai tingkat kesulitan, Wordazul cocok untuk membantu kalian yang mengalami kesulitan. Antarmuka yang menarik dan ramah pengguna membuat platform ini menjadi tempat yang sempurna untuk mengasah keterampilan kosa kata kalian sambil bersenang-senang.

How to Use

- Masukkan start word dan end word yang diinginkan.
- Pilih algoritma route planning yang ingin digunakan.
- Klik tombol search, lalu hasilnya akan segera muncul di bagian bawah.



Start Word: GRAB End Word: SNAG

Choose Algorithm: UCS

Search

```
Nodes visited: 809
Execution time (ms): 6
Memory used (MB): 5.00
Steps needed: 5
Path:
1. grab
2. crab
3. clog
4. clang
5. slag
6. snag
```

Greedy Best-First Search (GBFS)

The screenshot shows the WordAzul application window. At the top, it says "WORDAZUL" and "Selamat Datang di WordAzul!". Below that is a "How to Use" section with instructions: "Masukkan start word dan end word yang diinginkan.", "Pilih algoritma route planning yang ingin digunakan.", and "Klik tombol search, lalu hasilnya akan segera muncul di bagian bawah." A cartoon baby character is displayed above the input fields. The input fields show "Start Word: GRAB" and "End Word: SNAG". The "Choose Algorithm:" dropdown is set to "GBFS". Below these are "Search" and "Clear" buttons. The results section displays the following information:

```

Nodes visited: 9
Execution time (ms): 1
Memory used (MB): 0.50
Steps needed: 8
Path:
1. grab
2. crab
3. crag
4. brag
5. drag
6. frag
7. flag
    
```

A*

The screenshot shows the WordAzul application window. At the top, it says "WORDAZUL" and "Selamat Datang di WordAzul!". Below that is a "How to Use" section with instructions: "Masukkan start word dan end word yang diinginkan.", "Pilih algoritma route planning yang ingin digunakan.", and "Klik tombol search, lalu hasilnya akan segera muncul di bagian bawah." A cartoon baby character is displayed above the input fields. The input fields show "Start Word: GRAB" and "End Word: SNAG". The "Choose Algorithm:" dropdown is set to "A*". Below these are "Search" and "Clear" buttons. The results section displays the following information:

```

Nodes visited: 29
Execution time (ms): 1
Memory used (MB): 0.60
Steps needed: 5
Path:
1. grab
2. crab
3. crag
4. clag
5. slag
6. snag
    
```

4.2.3 Uji Program 3

TRICK → TREAT

Uniform-Cost Search (UCS)

The screenshot shows the WordAzul interface. At the top, it says "Selamat Datang di WordAzul!". Below that is a "How to Use" section with instructions: "Masukkan start word dan end word yang diinginkan.", "Pilih algoritma route planning yang ingin digunakan.", and "Klik tombol search, lalu hasilnya akan segera muncul di bagian bawah." A cartoon baby icon is displayed below the instructions. The search form at the bottom has "Start Word: TRICK" and "End Word: TREAT". The "Choose Algorithm:" dropdown is set to "UCS". A "Search" button is present. The results section shows the following output:

```
Nodes visited: 1453
Execution time (ms): 12
Memory used (MB): 9.50
Steps needed: 7
Path:
1. trick
2. wrick
3. wreck
4. wreak
5. break
6. bread
7. tread
```

Greedy Best-First Search (GBFS)

The screenshot shows the WordAzul interface. At the top, it says "Selamat Datang di WordAzul!". Below that is a "How to Use" section with instructions: "Masukkan start word dan end word yang diinginkan.", "Pilih algoritma route planning yang ingin digunakan.", and "Klik tombol search, lalu hasilnya akan segera muncul di bagian bawah." A cartoon baby icon is displayed below the instructions. The search form at the bottom has "Start Word: TRICK" and "End Word: TREAT". The "Choose Algorithm:" dropdown is set to "GBFS". A "Search" button is present. The results section shows the following output:

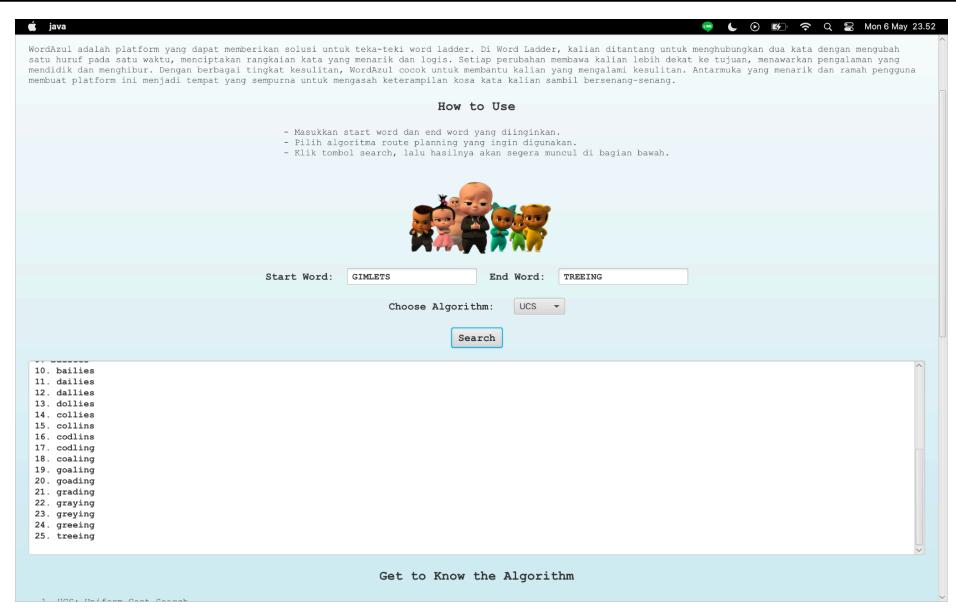
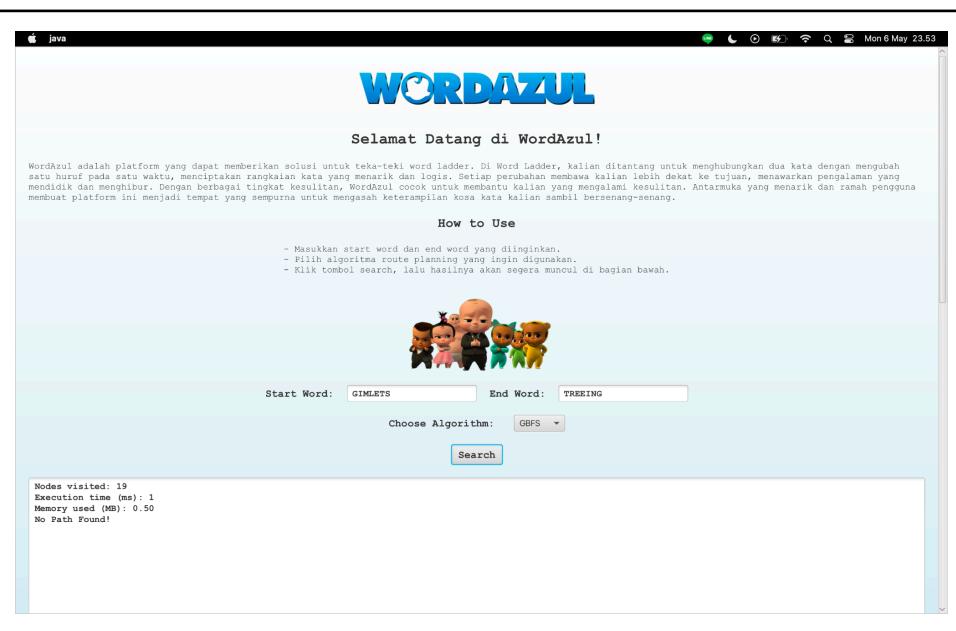
```
Nodes visited: 26
Execution time (ms): 0
Memory used (MB): 0.50
Steps needed: 25
Path:
1. trick
2. trickt
3. trait
4. traitt
5. trapt
6. wrapt
7. wraps
```

A*

The screenshot shows the WordAzul interface with the A* algorithm selected. The start word is TRICK and the end word is TREAT. The algorithm output shows the path: 1. trick, 2. wrick, 3. wreck, 4. wreak, 5. break, 6. bread, 7. tread. The interface includes a 'How to Use' section with instructions and a search button.

4.2.4 Uji Program 4

The screenshot shows the WordAzul interface with the Uniform-Cost Search (UCS) algorithm selected. The start word is GIMLETS and the end word is TREEING. The algorithm output shows the path: 1. gimlets, 2. giglets, 3. guglets, 4. glets, 5. bullets, 6. ballets, 7. ballers. The interface includes a 'How to Use' section with instructions and a search button.

	 <p>Start Word: GIMLETS End Word: TREEING</p> <p>Choose Algorithm: UCS</p> <p>Search</p> <p>1. ballies 11. dailies 12. dallies 13. dollies 14. coaling 15. collines 16. codlins 17. coding 18. coaling 19. goaling 20. coaling 21. grading 22. graying 23. greyling 24. greeling 25. treeing</p> <p>Get to Know the Algorithm</p>
<h3>Greedy Best-First Search (GBFS)</h3>	 <p>Start Word: GIMLETS End Word: TREEING</p> <p>Choose Algorithm: GBFS</p> <p>Search</p> <p>Nodes visited: 19 Execution time (ms): 1 Memory used (MB): 0.50 No Path Found!</p>

A*

WORDAZUL

Selamat Datang di WordAzul!

WordAzul adalah platform yang dapat memberikan solusi untuk teka-teki word ladder. Di Word Ladder, kalian ditantang untuk menghubungkan dua kata dengan mengubah satu huruf pada satu waktu, menciptakan rangkian kata yang menarik dan logis. Setiap perubahan membawa kalian lebih dekat ke tujuan, menawarkan pengalaman yang mendidik dan menghibur. Dengan berbagai tingkat kesulitan, WordAzul cocok untuk membantu kalian yang mengalami kesulitan. Antarmuka yang menarik dan ramah pengguna membuat platform ini menjadi tempat yang sempurna untuk mengasah keterampilan kosa kata kalian sambil bersenang-senang.

How to Use

- Masukkan start word dan end word yang dilengkapi.
- Pilih algoritma route planning yang ingin digunakan.
- Klik tombol search, lalu hasilnya akan segera muncul di bagian bawah.

Start Word: GIMLETS End Word: TREEING

Choose Algorithm: A*

Search

Nodes visited: 4021
Execution time (ms): 43
Memory used (MB): 34.50
Steps needed: 24
Path:
1. gimlets
2. giglets
3. gillets
4. gulletts
5. cullets
6. cutlets
7. cutlers

Nodes visited: 4021
Execution time (ms): 43
Memory used (MB): 34.50
Steps needed: 24
Path:
1. gulletts
2. cullets
3. cutlets
4. cutlers
5. cutters
6. cuttlers
7. cuttling
8. cuttling
9. cuttling
10. cuttling
11. cuttling
12. cuttling
13. cuttling
14. cuttling
15. matting
16. casting
17. casting
18. coating
19. coating
20. coating
21. grating
22. graying
23. greyling
24. greeling
25. treeing

4.2.5 Uji Program 5

RATIONS → WINTERY

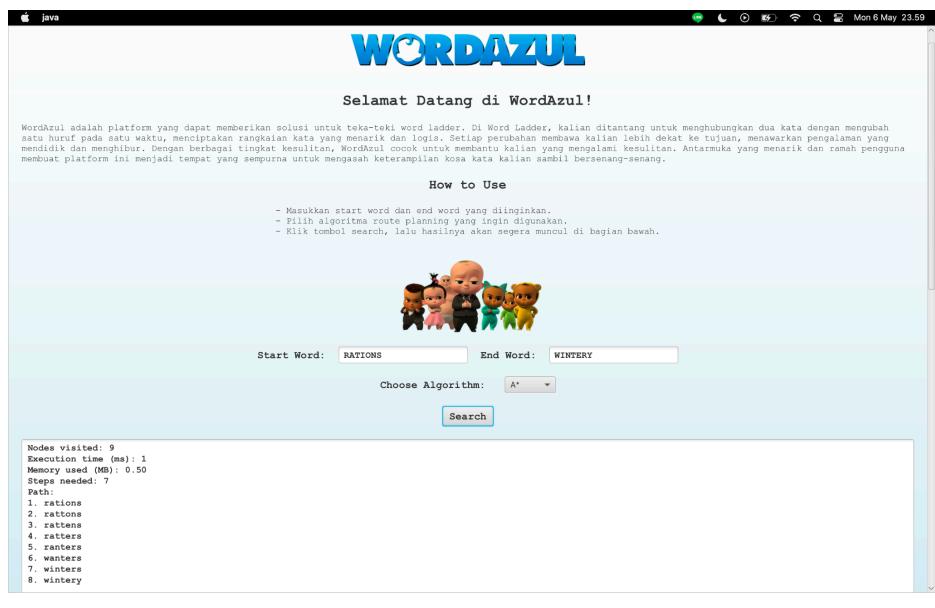
Uniform-Cost Search (UCS)

The screenshot shows the WordAzul application interface. At the top, it says "Selamat Datang di WordAzul!". Below that is a descriptive text about the platform. A "How to Use" section provides instructions: "Masukkan start word dan end word yang diinginkan.", "Pilih algoritma route planning yang ingin digunakan.", and "Klik tombol search, lalu hasilnya akan segera muncul di bagian bawah." There is a small cartoon character icon above the input fields. The search parameters are set to "Start Word: RATIONS" and "End Word: WINTERY". The "Choose Algorithm:" dropdown is set to "UCS". A "Search" button is present. The results section displays the path taken: "Nodes visited: 414", "Execution time (ms): 4", "Memory used (MB): 4.00", and "Steps needed: 7". The path is listed as: 1. rations, 2. rattons, 3. rattens, 4. ritters, 5. ratters, 6. winters, 7. winters, 8. wintery.

Greedy Best-First Search (GBFS)

The screenshot shows the WordAzul application interface. At the top, it says "Selamat Datang di WordAzul!". Below that is a descriptive text about the platform. A "How to Use" section provides instructions: "Masukkan start word dan end word yang diinginkan.", "Pilih algoritma route planning yang ingin digunakan.", and "Klik tombol search, lalu hasilnya akan segera muncul di bagian bawah." There is a small cartoon character icon above the input fields. The search parameters are set to "Start Word: RATIONS" and "End Word: WINTERY". The "Choose Algorithm:" dropdown is set to "GBFS". A "Search" button is present. The results section displays the path taken: "Nodes visited: 11", "Execution time (ms): 0", "Memory used (MB): 0.50", and "Steps needed: 10". The path is listed as: 1. rations, 2. rattons, 3. rattens, 4. ratters, 5. ritters, 6. fitters, 7. fitters, 8. hitters, 9. hitters, 10. winters, 11. wintery.

A*



4.2.6 Uji Program 6

BLUBBERRY → CLAMPERS

Uniform-Cost Search (UCS)	<p>The screenshot shows the WordAzul interface with the title "WORDAZUL" and a banner "Selamat Datang di WordAzul!". Below the banner is a brief description of the platform's purpose: "WordAzul adalah platform yang dapat memberikan solusi untuk teka-teki word ladder. Di Word Ladder, kalian ditantang untuk menghubungkan dua kata dengan mengubah satu huruf pada satu waktu, menciptakan rangkulan kata yang menarik dan logis. Setiap perubahan membawa kalian lebih dekat ke tujuan, menawarkan pengalaman yang mendidik dan menghibur. Dengan berbagai tingkat kesulitan, Wordazul cocok untuk membantu kalian yang mengalami kesulitan. Antarmuka yang menarik dan ramah pengguna membuat platform ini menjadi tempat yang sempurna untuk mengasah keterampilan kosa kata kalian sambil bersenang-senang." The "How to Use" section provides instructions: "Masukkan start word dan end word yang diinginkan, Pilih algoritma route planning yang ingin digunakan, Klik tombol search, lalu hasilnya akan segera muncul di bagian bawah." The search parameters are set to "Start Word: BLUBBERRY" and "End Word: CLAMPERS". The algorithm dropdown is set to "UCS". A cartoon baby icon is displayed above the search input fields. The search results show the path taken: "Nodes visited: 30 Execution time (ms): 3 Memory used (MB): 1.00 Steps needed: 5 Path: 1. blubbery 2. blubbers 3. blabbeers 4. clabbeers 5. clambers 6. clampers".</p>
---------------------------	--

Greedy Best-First Search (GBFS)	<p>WORDAZUL</p> <p>Selamat Datang di WordAzul!</p> <p>Wordazul adalah platform yang dapat memberikan solusi untuk teka-teki word ladder. Di Word Ladder, kalian ditantang untuk menghubungkan dua kata dengan mengubah satu huruf pada satu waktu, menciptakan rangkulan kata yang menarik dan logis. Setiap perubahan membawa kalian lebih dekat ke tujuan, menawarkan pengalaman yang mendidik dan menghibur. Dengan berbagai tingkat kesulitan, Wordazul cocok untuk membantu kalian yang mengalami kesulitan. Antarmuka yang menarik dan ramah pengguna membuat platform ini menjadi tempat yang sempurna untuk mengasah keterampilan kosa kata kalian sambil bersenang-senang.</p> <p>How to Use</p> <ul style="list-style-type: none"> - Masukkan start word dan end word yang diinginkan. - Pilih algoritma route planning yang ingin digunakan. - Klik tombol search, lalu hasilnya akan segera muncul di bagian bawah. <p>Start Word: BLUBBERRY End Word: CLAMPERS</p> <p>Choose Algorithm: GBFS</p> <p>Search</p> <pre> Nodes visited: 6 Execution time (ms): 2 Memory used (MB): 0.92 Steps needed: 5 Path: 1. blubbery 2. blubbers 3. blubbes 4. clabbers 5. clambers 6. clampers </pre>
A*	<p>WORDAZUL</p> <p>Selamat Datang di WordAzul!</p> <p>Wordazul adalah platform yang dapat memberikan solusi untuk teka-teki word ladder. Di Word Ladder, kalian ditantang untuk menghubungkan dua kata dengan mengubah satu huruf pada satu waktu, menciptakan rangkulan kata yang menarik dan logis. Setiap perubahan membawa kalian lebih dekat ke tujuan, menawarkan pengalaman yang mendidik dan menghibur. Dengan berbagai tingkat kesulitan, Wordazul cocok untuk membantu kalian yang mengalami kesulitan. Antarmuka yang menarik dan ramah pengguna membuat platform ini menjadi tempat yang sempurna untuk mengasah keterampilan kosa kata kalian sambil bersenang-senang.</p> <p>How to Use</p> <ul style="list-style-type: none"> - Masukkan start word dan end word yang diinginkan. - Pilih algoritma route planning yang ingin digunakan. - Klik tombol search, lalu hasilnya akan segera muncul di bagian bawah. <p>Start Word: BLUBBERRY End Word: CLAMPERS</p> <p>Choose Algorithm: A*</p> <p>Search</p> <pre> Nodes visited: 8 Execution time (ms): 2 Memory used (MB): 0.81 Steps needed: 5 Path: 1. blubbery 2. blubbers 3. blubbes 4. clabbers 5. clambers 6. clampers </pre>

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

Dari hasil uji 6 *test case* menggunakan GUI diatas, kita dapat hasilnya sebagai berikut.

Tabel 4.3.1 Perbandingan Algoritma UCS, Greedy Best First Search, dan A*

Test Case Ke-	Start Word	End Word	Algoritma	Waktu (ms)	Memori (MB)	Optimalitas
1	ANT	BEE	UCS	6	3	Iya
			GBFS	1	0.96	Tidak
			A*	2	0.96	Iya

2	GRAB	SNAG	UCS	6	5	Iya
			GBFS	1	0.5	Tidak
			A*	1	0.6	Iya
3	TRICK	TREAT	UCS	12	9.5	Iya
			GBFS	0	0.5	Tidak
			A*	1	1	Iya
4	GIMLETS	TREEING	UCS	55	5.45	Iya
			GBFS	1	0.5	Tidak
			A*	43	34.5	Iya
5	RATIONS	WINTERY	UCS	4	4	Iya
			GBFS	0	0.5	Tidak
			A*	1	0.5	Iya
6	BLUBBERRY	CLUMBERS	UCS	3	0.93	Iya
			GBFS	2	0.92	Iya
			A*	2	0.81	Iya

Dari tabel diatas, berikut analisis mengenai optimalitas, waktu eksekusi serta memori yang dibutuhkan dari ketiga Algoritma tersebut.

4.3.1 Optimalitas

Optimalitas menunjukkan apakah solusi yang ditemukan oleh algoritma adalah yang solusi yang terbaik atau dalam permainan *word ladder* ini ditandai dengan menghasilkan jalur yang terpendek.

- Jika kita lihat dari tabel, UCS selalu menemukan solusi yang optimal karena algoritma ini mengevaluasi semua kemungkinan dengan cara yang menjamin solusi terpendek akan ditemukan terlebih dahulu.
- GBFS cenderung tidak optimal karena fokus pada pilihan yang "terbaik" menurut heuristik tanpa mempertimbangkan biaya sebenarnya. Dari keenam test case tersebut, hanya test case 6 saja dimana GBFS menghasilkan panjang jalur yang sama dengan algoritma UCS dan A*, bahkan pada test case 4 algoritma GBFS tidak berhasil menemukan solusi sama sekali. Hal ini disebabkan akibat hanya berfokus pada pilihan menurut heuristik tadi, GBFS berkemungkinan untuk stuck dan tidak bisa melakukan *backtracking*.

- A* dipastikan akan selalu menemukan solusi optimal karena algoritma ini menggabungkan strategi UCS dengan heuristic (greedy) untuk memandu pencarian.

4.3.2 Waktu Eksekusi

Waktu eksekusi mengukur kecepatan algoritma dalam menemukan solusi.

- UCS, yang mengevaluasi semua kemungkinan tadi, cenderung akan berjalan lebih lambat. Dalam beberapa kasus, waktu eksekusi UCS bisa jauh lebih tinggi dibandingkan GBFS atau A* seperti pada test case 3.
- GBFS sering kali menjadi yang tercepat karena algoritma ini hanya mengikuti jalur yang dianggap terbaik oleh heuristik, meskipun solusi yang ditemukan bisa jadi tidak optimal.
- A*, yang menggunakan heuristik untuk membimbing pencarian ke arah yang benar, umumnya lebih cepat daripada UCS dan terkadang juga bisa lebih cepat dari GBFS.

Dari data tabel, terlihat bahwa GBFS memiliki waktu eksekusi paling cepat di setiap kasus, dengan waktu eksekusi sering kali jauh lebih singkat daripada UCS dan A*. UCS cenderung memiliki waktu eksekusi yang paling lama, terutama pada kasus-kasus yang lebih kompleks, seperti pada Test Case 4. Dan algoritma A* menunjukkan waktu eksekusi yang lebih baik dibandingkan UCS dan mendekati GBFS.

4.3.3 Memori

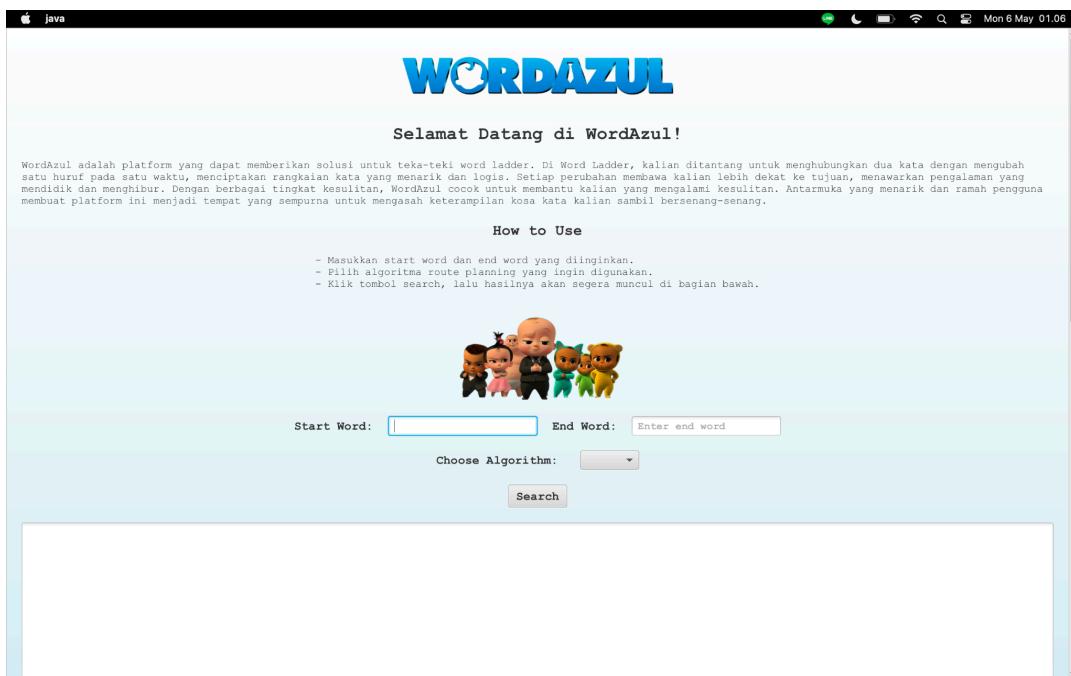
Penggunaan memori mengukur seberapa banyak memori yang diperlukan oleh algoritma. Hal ini bisa dilihat dari jumlah nodes yang dikunjungi juga.

- UCS biasanya menggunakan lebih banyak memori karena harus melacak semua kemungkinan jalur.
- GBFS cenderung menggunakan sedikit memori karena hanya mempertahankan jalur paling menjanjikan berdasarkan heuristik. Jika kita lihat juga nodes yang dikunjungi oleh algoritma GBFS ini selalu sama dengan panjang jalur yang dihasilkan.
- A* menggunakan lebih banyak memori daripada GBFS, tetapi umumnya lebih efisien daripada UCS.

Jadi, dapat diambil kesimpulan bahwa jika optimalitas adalah prioritas, algoritma UCS dan A* adalah pilihan yang tepat, dengan UCS yang lebih terjamin namun dengan waktu yang lebih lambat. Jika kecepatan menjadi prioritas, GBFS menjadi pilihan dengan waktu eksekusi paling cepat, tetapi dengan risiko solusi yang tidak optimal. Sedangkan A* memberikan keseimbangan antara optimalitas dan waktu eksekusi, dengan penggunaan memori yang cukup sehingga A* adalah pilihan algoritma yang terbaik. UCS akan memberikan solusi optimal, namun dengan catatan biaya waktu dan memori yang lebih tinggi.

4.4 Penjelasan Mengenai Implementasi Bonus

Pada Tugas Kecil 3 IF2211 ini saya menggunakan JavaFX untuk membuat antarmuka pengguna grafis (GUI) untuk aplikasi *word ladder solver*. Untuk menggunakan JavaFX dapat dilakukan instalasi terlebih dahulu pada <https://openjfx.io/>. Berikut adalah tampilan awal saat dibuka.



Gambar 4.4.1 Tampilan Awal GUI

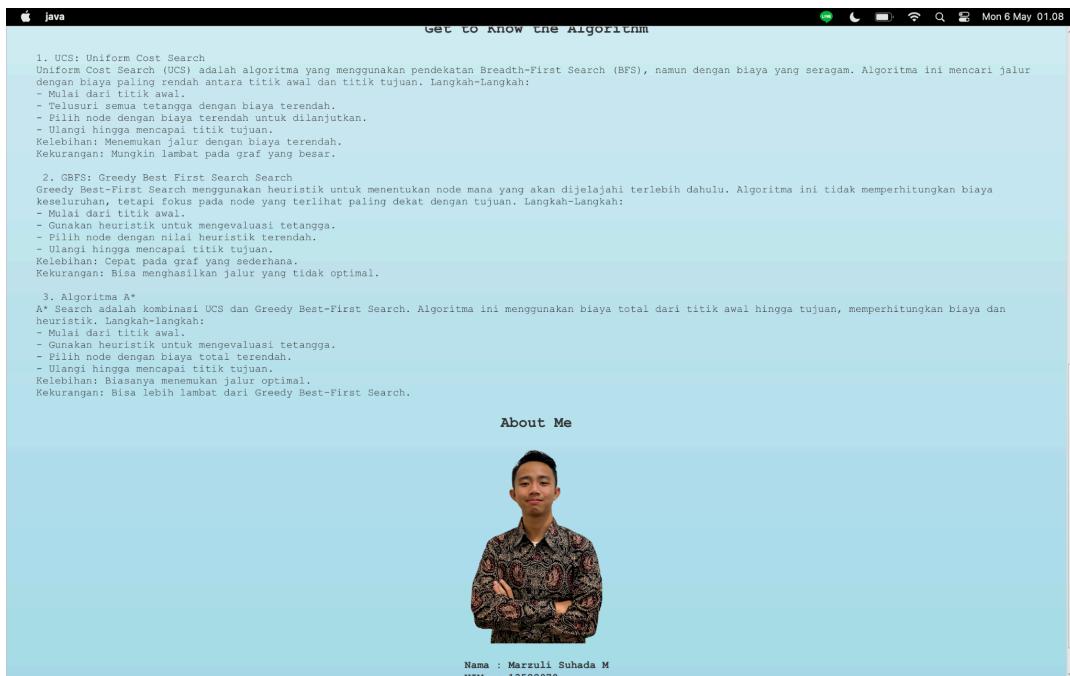
Program ini meng-*extends* class *Application*, sehingga memiliki method `start(Stage primaryStage)` yang menjadi titik awal untuk memulai aplikasi JavaFX. Method ini bertanggung jawab untuk mengatur layout dan elemen-elemen GUI, serta menghubungkan interaksi pengguna dengan logika aplikasi. Pada program ini digunakan sebuah *VBox* sebagai layout utama dengan padding dan jarak antar-komponen yang cukup lebar. Selain itu saya juga menambahkan elemen seperti *ImageView* untuk menampilkan gambar header dan *Label* dengan teks selamat datang dan deskripsi tentang aplikasi untuk memberikan konteks kepada pengguna tentang apa yang bisa mereka gunakan dari aplikasi ini.

Kemudian saya juga menambahkan informasi tentang cara penggunaan aplikasi yang ditandai *label* `howToUseLabel` dan `stepsLabel` yang menjelaskan langkah-langkah yang perlu diikuti oleh pengguna untuk menjalankan aplikasi. Untuk mengimplementasikan bagian input, program menggunakan *HBox* untuk mengatur tata letak label dan *TextField* untuk memasukkan kata awal dan akhir. Selanjutnya, program menggunakan *ComboBox* untuk membuat dropdown pilihan algoritma pencarian (UCS, GBFS, atau A*). Tombol `Search` digunakan untuk memulai pencarian ketika pengguna mengkliknya.

Di bagian logika aplikasi, tombol `Search` dihubungkan dengan *event handler* yang menangani interaksi pengguna. *Event handler* ini akan mengambil input kata awal dan akhir, serta algoritma yang dipilih, dan menjalankan validasi untuk memastikan kata memiliki panjang yang sama, bukan kata yang sama, dan keduanya berada dalam kamus. Jika validasi gagal, pesan kesalahan ditampilkan di dalam `TextArea`.

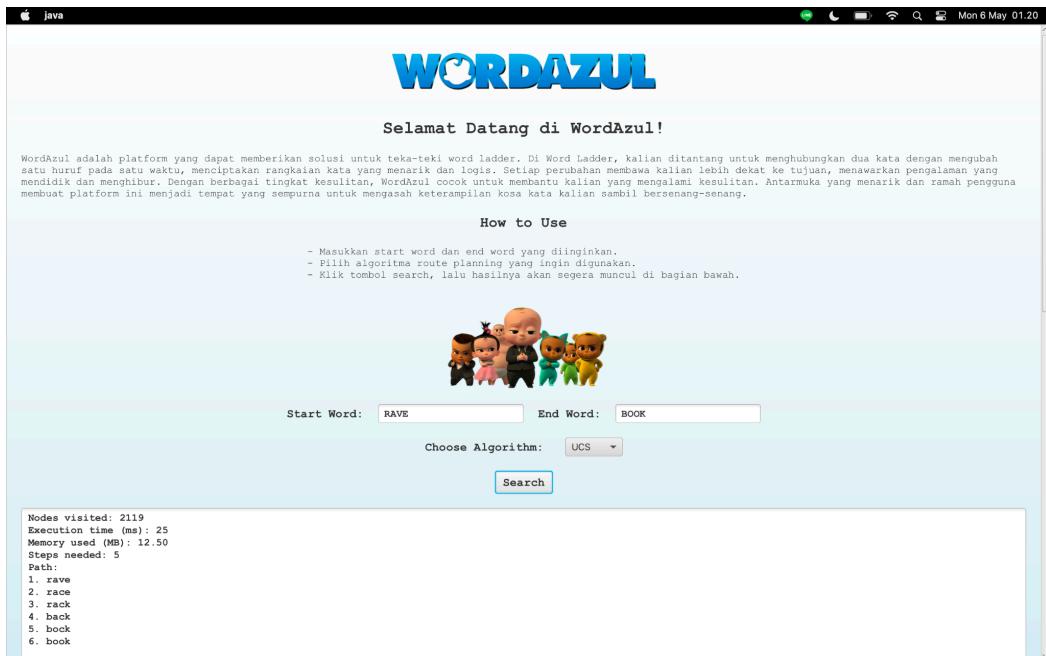
Jika validasi berhasil, program akan mengukur penggunaan memori dan waktu eksekusi sebelum menjalankan algoritma pencarian. Algoritma yang dipilih (UCS, GBFS, atau A*) dijalankan untuk menemukan jalur antara kata awal dan akhir. Setelah algoritma selesai, program mengukur waktu eksekusi dan penggunaan memori, lalu menampilkan hasilnya di dalam `TextArea`. Jika jalur ditemukan, program menampilkan jalur beserta jumlah langkah yang diperlukan. Jika tidak ditemukan, pesan yang sesuai akan ditampilkan.

Selain itu, saya juga menambahkan informasi tentang setiap algoritma pencarian. Label `algorithmExplanation` menjelaskan bagaimana UCS, GBFS, dan A* bekerja, termasuk kelebihan dan kekurangannya masing-masing. Saya juga menambahkan bagian "About Me" yang menampilkan informasi pribadi saya (selaku yang membuat program ini), seperti nama dan NIM, serta kutipan yang saya sukai. Untuk membuat tampilan lebih menarik, saya mengubah latar belakang program dengan gradien warna menggunakan metode `applyGradientBackground`. Terakhir, program menggunakan `ScrollPane` untuk memungkinkan pengguna melakukan *scrolling* antarmuka saat elemen GUI lebih besar daripada tampilan.



Gambar 4.4.2 Tampilan GUI Bagian Tambahan

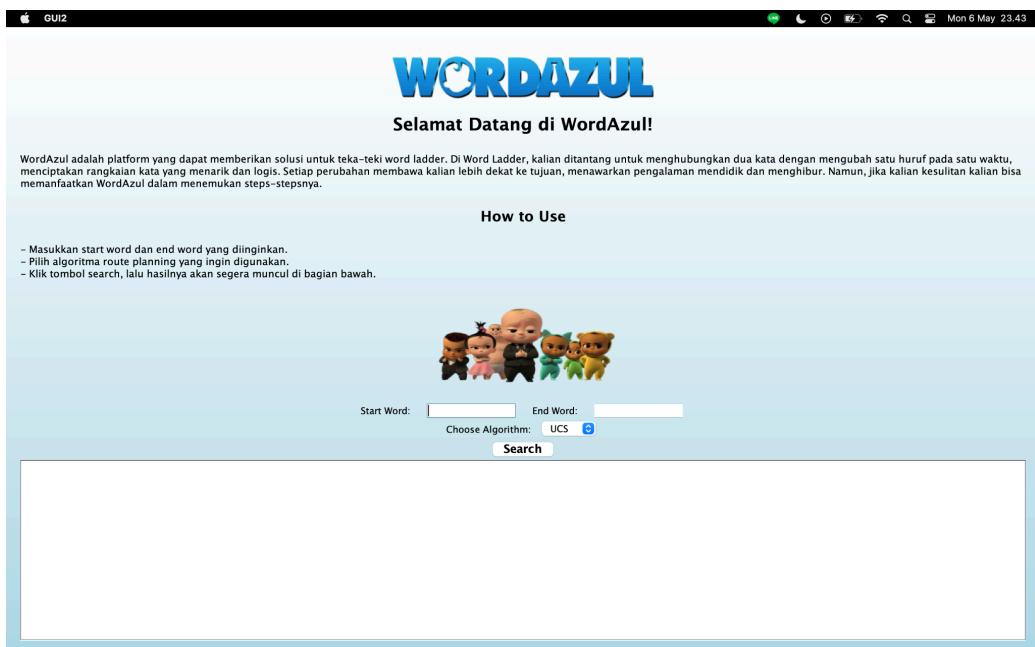
Berikut contoh tampilan hasil saat dilakukan pencarian dari kata RAVE ke kata BOOK menggunakan algoritma UCS.



Gambar 4.4.3 Tampilan Hasil Pencarian

Namun, akibat dari *prerequisite* instalasi JavaFX terlebih dahulu jika ingin menggunakan program ini dan terkadang bisa menyebabkan error jika tidak diatur dengan baik PATH nya, saya membuat cadangan GUI juga yang menggunakan Java Swing. Dalam pembuatan GUI ini, berbagai komponen Java Swing digunakan, seperti JFrame, JTextField, JLabel, JComboBox, JTextArea, JButton, dan JPanel. JFrame menjadi kerangka utama tempat seluruh komponen lain ditempatkan, sementara komponen lainnya bertindak sebagai elemen untuk interaksi pengguna dan tampilan informasi. Misalnya, JTextField digunakan untuk memasukkan kata, JComboBox untuk memilih algoritma, dan JTextArea untuk menampilkan hasil pencarian. Tata letak GUI diatur dengan kombinasi BoxLayout, FlowLayout, dan komponen khusus seperti GradientPanel, yang memberi efek visual gradien pada latar belakang.

Meskipun Java Swing adalah toolkit yang lebih tua, namun sangat cocok menjadi pilihan yang baik untuk membangun GUI yang sederhana dan ringan. Dibandingkan dengan JavaFX, Java Swing mungkin lebih terbatas dalam hal efek grafis dan animasi, namun kelebihannya terletak pada kompatibilitas yang lebih luas dan kemudahan penggunaan. Untuk aplikasi yang memerlukan antarmuka grafis yang lebih kompleks dengan efek visual canggih, JavaFX mungkin menjadi pilihan yang lebih baik, namun Java Swing masih sangat berguna untuk aplikasi yang berfokus pada fungsi dan kesederhanaan.



Gambar 4.4.4 Tampilan Menggunakan GUI Java Swing

BAB V

KESIMPULAN DAN SARAN

5.1 Kesimpulan

Pada tugas kecil III IF2211 Strategi Algoritma ini, telah berhasil diimplementasikan tiga algoritma pencarian *route planning*, yaitu Uniform Cost Search (UCS), Greedy Best-First Search (GBFS), dan A* untuk menyelesaikan permainan *word ladder*. Implementasi algoritma ini melibatkan penggunaan struktur data *priority queue*, kamus kata untuk mencari tetangga yang valid, dan penanganan input dari pengguna.

Algoritma UCS digunakan untuk menemukan jalur dengan biaya terendah, GBFS untuk mencari jalur berdasarkan heuristik, dan A* yang menggabungkan biaya dan heuristik untuk pencarian yang optimal. Hasil dari implementasi ini berhasil direalisasikan dalam file Java untuk penggunaan melalui CLI dan aplikasi berbasis GUI yang menggunakan JavaFX dan Java Swing, di mana pengguna dapat memilih algoritma dan mendapatkan hasil dalam bentuk jalur, jumlah simpul yang dikunjungi, serta waktu dan memori yang digunakan. Berdasarkan hasil implementasi ini, dapat disimpulkan bahwa algoritma UCS, dan A* cocok digunakan untuk menyelesaikan masalah *word ladder* dengan prioritas A* yang terbaik dalam segi waktu eksekusi dan ruang memori yang digunakan, sedangkan algoritma GBFS kurang tepat untuk digunakan karena terkadang tidak memberikan hasil yang optimal atau bahkan tidak memberikan solusi. Tentunya ketiga algoritma ini juga memiliki kelebihan dan kekurangan tersendiri.

5.2 Saran

Tugas Kecil IF2211 Strategi Algoritma Semester III Tahun 2022/2023 menjadi salah satu tugas menarik bagi penulis dan memberikan banyak pelajaran berharga dalam mengimplementasikan algoritma pencarian dan mengembangkan aplikasi GUI. Berikut adalah beberapa saran bagi mereka yang ingin mengerjakan tugas serupa atau mengembangkan aplikasi serupa di masa mendatang:

1. Sebelum mengimplementasikan algoritma seperti UCS, GBFS, atau A*, penting untuk memiliki pemahaman yang kuat tentang konsep algoritma pencarian dan struktur data dasar seperti *priority queue* dan graf. Ini akan mempermudah dalam merancang dan mengimplementasikan solusi.
2. Jika aplikasi membutuhkan library atau dependensi tertentu, pastikan untuk memiliki daftar lengkap dan menginstalnya dengan benar. Mengelola dependensi secara efektif akan mengurangi masalah saat menjalankan aplikasi.
3. Jika proyek yang dibuat melibatkan pengembangan GUI, pemahaman dasar tentang framework seperti JavaFX atau yang lainnya sangat penting. Pengetahuan tentang desain antarmuka pengguna dan interaksi dengan pengguna akan meningkatkan pengalaman pengguna.

4. Sebelum merilis aplikasi, pastikan untuk melakukan pengujian dan evaluasi menyeluruh terhadap algoritma dan fungsionalitas GUI. Ini membantu memastikan bahwa aplikasi berfungsi sebagaimana mestinya dan memberikan hasil yang diharapkan.

DAFTAR PUSTAKA

Munir, R. (2021). Penentuan rute (Route/Path Planning) - Bagian 1 . Retrieved from Homepage Rinaldi Munir:

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf>

Munir, R. (2021). Penentuan rute (Route/Path Planning) - Bagian 2 . Retrieved from Homepage Rinaldi Munir:

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf>

GeeksForGeeks. Greedy Best first search algorithm. Retrieved from:
<https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/>

GeeksForGeeks. Difference between Best-First Search and A* Search? Retrieved from:
<https://www.geeksforgeeks.org/difference-between-best-first-search-and-a-search/>

GeeksForGeeks. Uniform-Cost Search (Dijkstra for large Graphs) Retrieved from:
<https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/>

LAMPIRAN

Pranala

Kode program dapat diakses pada link berikut.

https://github.com/zultopia/Tucil3_13522070.git

How to Use

Pertama, clone dengan cara git clone https://github.com/zultopia/Tucil3_13522070.git lalu navigasi ke `cd ./Tucil3_13522070`

1. Untuk penggunaan program melalui CLI lakukan langkah-langkah berikut.
 - `java -cp bin Main`
2. Untuk penggunaan program melalui GUI JavaFX
 - Pastikan sudah menginstall java setidaknya Java 17 dan sudah menginstall JavaFX pada <https://openjfx.io/> yang sesuai dengan OS yang kalian gunakan
 - Untuk Linux lakukan instalasi dengan mengetikkan `sudo apt install openjdk-17-jdk`
 - Untuk Linux dan MacOS dapat langsung menjalankan `cd src/wordladder` lalu `./run.sh` atau
 - `javac --module-path <path to your javafx sdk path for example /Users/azulsuhada/Downloads/javafx-sdk-22.0.1>/lib --add-modules javafx.controls,javafx.fxml,javafx.graphics AStar.java GBFS.java UCS.java WordDictionary.java Node.java Utility.java Result.java GUI.java`
 - `java --module-path ./javafx-sdk-22.0.1/lib --add-modules javafx.controls,javafx.fxml GUI`
3. Untuk penggunaan program melalui GUI Java Swing
 - `cd src/wordladder`
 - `javac AStar.java GBFS.java UCS.java WordDictionary.java Node.java Utility.java Result.java GUI2.java`
 - `java GUI2`

Tabel Poin

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