

Tugas Kecil 3 IF2211 Strategi Algoritma
Penyelesaian Permainan *Word Ladder* Menggunakan Algoritma
UCS, *Greedy Best First Search*, dan A*



Disusun Oleh:
13522108 - Muhammad Neo Cicero Koda

INSTITUT TEKNOLOGI BANDUNG
2024

BAB 1: ANALISIS DAN IMPLEMENTASI	3
BAB 2: SOURCE CODE PROGRAM	5
2.1. File dictionary.txt	5
2.2. File DictSet.java	6
2.3. File WordNode.java	7
2.4. File PathFinder.java	8
2.5. File GUI.java	10
2.6. File Main.java	12
BAB 3: TANGKAPAN LAYAR	13
3.1. Test Case 1 (Four - Crop)	13
3.1.1. UCS	13
3.1.2. Greedy Best First Search	13
3.1.3. A*	13
3.2. Test Case 2 (River - Month)	14
3.2.1. UCS	14
3.2.2. Greedy Best First Search	14
3.2.3. A*	14
3.3. Test Case 3 (Tablet - Rhythm)	15
3.3.1. UCS	15
3.3.2. Greedy Best First Search	15
3.3.3. A*	15
3.4. Test Case 4 (Pet - Ham)	16
3.4.1. UCS	16
3.4.2. Greedy Best First Search	16
3.4.3. A*	16
3.5. Test Case 5 (Grease - Remote)	17
3.5.1. UCS	17
3.5.2. Greedy Best First Search	17
3.5.3. A*	17
3.6. Test Case 6 (Black - Store)	18
3.6.1. UCS	18
3.6.2. Greedy Best First Search	18
3.6.3. A*	18
BAB 4: ANALISIS HASIL PERBANDINGAN SOLUSI	19
BAB 5: PENJELASAN IMPLEMENTASI BONUS	20
BAB 6: PRANALA REPOSITORY	21
LAMPIRAN	22

BAB 1: ANALISIS DAN IMPLEMENTASI

Algoritma yang diimplementasikan adalah algoritma UCS, Greedy Best First Search, dan A*. Ketiga algoritma tersebut memiliki implementasi yang hampir sama, namun berbeda dalam penentuan *cost*. Untuk menentukan jalur antara kata awal dan kata akhir, dibutuhkan beberapa struktur data penyimpan informasi terlebih dahulu, yaitu penyimpan daftar kata yang sudah pernah dikunjungi (*visited*) dan antrian untuk menentukan kata yang tetangganya akan dikunjungi selanjutnya (*wordQueue*). Antrian tersebut merupakan *priority queue* yang mengurutkan antrian berdasarkan prioritas berupa *cost* kata yang terkecil terlebih dahulu.

Pada awal penentuan jalur, ketiga algoritma akan mencatat kata awal dalam *visited* dan memasukkan kata tersebut ke dalam antrian. Nilai *cost* dari kata tersebut adalah nol. Selama antrian tersebut tidak kosong, ketiga algoritma akan mengambil kata pada antrian paling depan dan menentukan tetangga dari kata tersebut, jalur yang dilalui untuk mencapai tetangga, serta *cost* dari masing-masing tetangga. Cara penentuan nilai *cost* tersebut berbeda untuk ketiga algoritma. Untuk setiap tetangga, jika tetangga tersebut merupakan kata tujuan, pencarian akan berhenti dan akan didapatkan jalur yang dilalui dari kata awal hingga kata akhir serta jumlah kata yang dikunjungi. Jika kata tidak merupakan kata tujuan, tetapi kata belum dikunjungi, kata tersebut akan dicatat ke dalam *visited* dan dimasukkan ke dalam antrian. Hal tersebut akan terus berlaku hingga antrian kosong.

Nilai *cost* untuk setiap algoritma ditentukan oleh fungsi $f(n)$. Fungsi $f(n)$ merupakan fungsi evaluasi yang merepresentasikan *cost* total yang dimiliki oleh simpul n . Fungsi $g(n)$ merepresentasikan *cost* untuk mencapai simpul n dari simpul awal. Fungsi $h(n)$ merepresentasikan *cost* (yang diestimasikan dengan heuristik) untuk mencapai simpul tujuan dari simpul n . Pada kasus Word Ladder, fungsi $g(n)$ yang digunakan adalah jarak simpul n dari simpul awal. Setiap perpindahan dari satu kata ke kata yang lain memiliki jarak sebesar satu satuan. Fungsi $h(n)$ yang digunakan adalah jumlah karakter yang berbeda antara kata pada simpul n dan kata akhir. Untuk algoritma UCS, $f(n) = g(n)$. Untuk algoritma Greedy Best First Search, $f(n) = h(n)$. Untuk algoritma A*, $f(n) = g(n) + h(n)$.

Heuristik yang digunakan pada algoritma A*, yaitu $h(n)$, bersifat *admissible*. Suatu heuristik dapat dikatakan *admissible* jika bobot yang dihasilkan oleh heuristik selalu lebih kecil atau sama dengan bobot yang sebenarnya. Dalam kata lain, heuristik yang *admissible* tidak akan melakukan *overestimation* terhadap bobot sebenarnya untuk mencapai tujuan. Pada kasus Word Ladder, heuristik yang dipakai (jumlah karakter yang berbeda antara kata saat ini dan kata akhir) bersifat *admissible* karena langkah yang diperlukan untuk mencapai kata akhir dari kata saat ini setidaknya akan memerlukan $h(n)$ langkah. Kasus tersebut terjadi ketika selalu terdapat kata yang memiliki jumlah perbedaan karakter dengan kata akhir yang semakin kecil pada tiap pencarian tetangga.

Pada kasus Word Ladder, algoritma UCS sama dengan BFS. Pada dasarnya, mekanisme pembangkitan simpul pada UCS mirip dengan BFS. Perbedaannya adalah BFS memakai struktur data *queue*, sedangkan UCS memakai struktur data *priority queue* yang menempatkan simpul dengan bobot terkecil pada antrian terdepan. Pada kasus Word Ladder, jarak antara suatu simpul

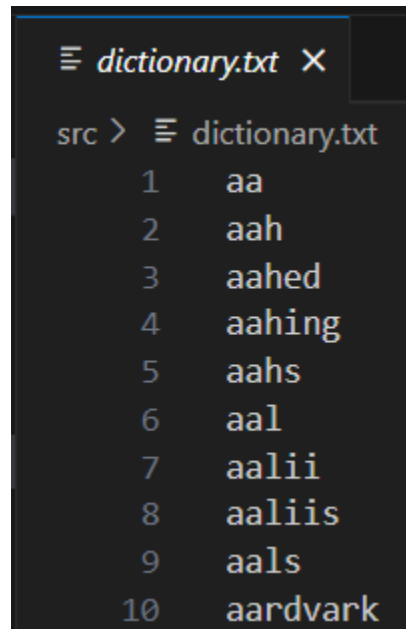
dengan simpul tetangganya selalu bernilai satu satuan. Hal tersebut menyebabkan simpul yang terdekat dengan simpul awal ditelusuri terlebih dahulu sehingga algoritma UCS akan menelusuri kata-kata dengan urutan yang sama seperti BFS.

Secara teoritis, algoritma A* akan lebih efisien dibandingkan dengan algoritma UCS pada kasus *word ladder*. Algoritma A* menggunakan heuristik untuk mengestimasi simpul-simpul yang meyakinkan dan menelusuri simpul-simpul tersebut terlebih dahulu menggunakan fungsi $f(n) = g(n) + h(n)$. Dengan menelusuri simpul-simpul yang lebih meyakinkan terlebih dahulu dan mencegah ekspansi jalur yang diestimasi mahal, algoritma A* akan mampu mencapai kata akhir dengan lebih cepat.

Secara teoritis, algoritma Greedy Best First Search tidak menjamin solusi optimal untuk persoalan *word ladder*. Algoritma Greedy Best First Search hanya menentukan simpul yang terbaik untuk ditelusuri berdasarkan heuristik dan tanpa melihat bobot untuk mencapai simpul tersebut dari simpul awal. Oleh karena itu, meskipun algoritma Greedy Best First Search sudah mendapatkan bobot yang besar untuk mencapai suatu simpul, algoritma tersebut tetap akan menelusuri anak dari simpul tersebut jika nilai $h(n)$ dari anak tersebut baik.

BAB 2: SOURCE CODE PROGRAM

2.1. File dictionary.txt



A screenshot of a code editor window titled 'dictionary.txt'. The editor shows a list of 10 words, each preceded by a line number from 1 to 10. The words are: aa, aah, aahed, aahing, aahs, aal, aalii, aaliis, aals, and aardvark. The editor has a dark background and a light-colored text.

```
src > dictionary.txt
1 aa
2 aah
3 aahed
4 aahing
5 aahs
6 aal
7 aalii
8 aaliis
9 aals
10 aardvark
```

Gambar 2.1.1. Cuplikan dictionary.txt

File dictionary.txt berisi daftar kata yang dianggap sebagai kata berbahasa Inggris yang diakui oleh program.

2.2. File DictSet.java

```
class DictSet {
    private HashSet<String> dict;

    public DictSet() {
        dict = new HashSet<>();
    }

    public HashSet<String> getDict() {
        return dict;
    }

    public void loadDictFromFile(String filename) {
        try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = br.readLine()) != null) {
                dict.add(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Gambar 2.2.1. Isi file DictSet.java

File DictSet.java berisi kelas DictSet, yaitu kelas yang bertanggung jawab untuk memuat daftar kata dari dictionary.txt ke program.

Daftar atribut yang terdapat dalam kelas tersebut adalah:

- dict: Struktur data HashSet yang digunakan untuk menyimpan daftar kata.

Daftar *method* yang terdapat dalam kelas tersebut adalah:

- DictSet(): Konstruktor publik DictSet.
- getDict(): Mendapatkan isi kamus pada DictSet.
- loadDictFromFile(): Memuat isi kamus dari sebuah file.

2.3. File WordNode.java

```
public class WordNode implements Comparable<WordNode> {
    private String word;
    private ArrayList<String> path;
    private int cost;

    public WordNode(String word, ArrayList<String> path, int cost) {
        this.word = word;
        this.path = path;
        this.cost = cost;
    }

    public String getWord() {
        return word;
    }

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

    public int getCost() {
        return cost;
    }

    public int compareTo(WordNode otherWord) {
        return Integer.compare(this.cost, otherWord.cost);
    }
}
```

Gambar 2.3.1. Isi file WordNode.java

File WordNode.java berisi kelas WordNode, yaitu struktur data untuk merepresentasikan simpul pada pencarian rute. Kelas ini mengimplementasikan *interface* Comparable agar dapat diurutkan berdasarkan bobot pada *priority queue*.

Daftar atribut yang terdapat dalam kelas tersebut adalah:

- word: Kata yang terdapat pada WordNode tersebut.
- path: Jalur yang dilalui untuk mencapai kata tersebut.
- cost: Bobot yang dimiliki simpul.

Daftar *method* yang terdapat dalam kelas tersebut adalah:

- WordNode(): Konstruktor publik WordNode.
- getWord(), getPath(), getCost(): Masing-masing merupakan *getter* untuk atribut word, path, dan cost.
- compareTo(): Implementasi *interface* Comparable yang membandingkan dua WordNode berdasarkan atribut cost.

2.4. File Pathfinder.java

```
class Pathfinder {
    private HashSet<String> visited;
    private PriorityQueue<WordNode> wordQueue;
    private ArrayList<String> resultPath;

    public Pathfinder() {
        visited = new HashSet<>();
        wordQueue = new PriorityQueue<>();
        resultPath = new ArrayList<>();
    }

    public void resetPathFinder() {
        visited.clear();
        wordQueue.clear();
        resultPath.clear();
    }

    public ArrayList<String> getResultPath() {
        return resultPath;
    }

    public int getNumVisited() {
        return visited.size();
    }

    public int countCharacterDiff(String word1, String word2) {
        if (word1.length() != word2.length()) {
            return -1;
        }

        int count = 0;
        for (int i = 0; i < word1.length(); i++) {
            if (word1.charAt(i) != word2.charAt(i)) {
                count++;
            }
        }

        return count;
    }

    public boolean isChildWord(String parent, String child) {
        if (parent.length() != child.length()) {
            return false;
        }

        int count = 0;
        for (int i = 0; i < parent.length(); i++) {
            if (parent.charAt(i) != child.charAt(i)) {
                count++;
            }
        }
        if (count > 1) {
            return false;
        }
    }
}
```

Gambar 2.4.1. Isi file Pathfinder.java (1)


```

public ArrayList<WordNode> findChildNodes(WordNode parent, String endWord, DictSet dictset, String method) {
    ArrayList<WordNode> childNodes = new ArrayList<WordNode>();

    for (String childWord : dictset.getDict()) {
        if (isChildWord(parent.getWord(), childWord)) {
            WordNode childNode;
            ArrayList<String> newPath = new ArrayList<>(parent.getPath());
            newPath.add(parent.getWord());

            switch (method) {
                case "UCS":
                    childNode = new WordNode(childWord, newPath, newPath.size());
                    childNodes.add(childNode);
                    break;
                case "GBFS":
                    childNode = new WordNode(childWord, newPath, countCharacterDiff(childWord, endWord));
                    childNodes.add(childNode);
                    break;
                case "A*":
                    childNode = new WordNode(childWord, newPath, newPath.size() + countCharacterDiff(childWord, endWord));
                    childNodes.add(childNode);
                    break;
            }
        }
    }

    return childNodes;
}

public void findPath(String startWord, String endWord, DictSet dictset, String method) {
    resetPathFinder();

    if (startWord.equals(endWord)) {
        resultPath.add(startWord);
        return;
    }

    ArrayList<String> startPath = new ArrayList<>();
    WordNode startNode = new WordNode(startWord, startPath, cost:0);

    wordQueue.add(startNode);
    visited.add(startNode.getWord());

    while (!wordQueue.isEmpty()) {
        WordNode wordNode = wordQueue.poll();

        ArrayList<WordNode> childNodes = findChildNodes(wordNode, endWord, dictset, method);
        for (WordNode childNode : childNodes) {
            if (childNode.getWord().equals(endWord)) {
                resultPath.addAll(childNode.getPath());
                resultPath.add(childNode.getWord());
                return;
            }
            if (!visited.contains(childNode.getWord())) {
                wordQueue.add(childNode);
                visited.add(childNode.getWord());
            }
        }
        childNodes.clear();
    }
}

```

Gambar 2.4.2. Isi file PathFinder.java (2)

Kelas PathFinder berisi kode untuk menentukan jalur antara kata awal dan kata akhir.

Daftar atribut yang terdapat pada kelas tersebut adalah:

- visited: HashSet yang berisi daftar kata yang sudah dikunjungi.

- wordQueue: PriorityQueue yang menyimpan antrian WordNode yang akan diekspan selanjutnya.
- resultPath: Jalur hasil antara kata awal dan kata akhir.

Daftar *method* yang terdapat pada kelas tersebut adalah:

- Pathfinder(): Konstruktor publik Pathfinder.
- resetPathFinder(): Menghapus isi atribut Pathfinder.
- getResultPath(): Mengembalikan jalur hasil antara kata awal dan kata akhir.
- getNumVisited(): Mengembalikan jumlah kata yang telah dikunjungi Pathfinder dalam satu kali pencarian.
- countCharacterDiff(): Menghitung jumlah karakter yang berbeda antara dua String.
- isChildWord(): Mengembalikan *true* jika jumlah perbedaan karakter antara kata induk dan kata anak sebesar satu, *false* jika tidak.
- findChildNodes(): Menghasilkan ArrayList yang berisi WordNode yang merupakan anak dari kata induk. Setiap WordNode akan berisi kata anak, jalur yang dilalui untuk mencapai kata anak, dan bobot untuk mencapai kata anak. Bobot tersebut dihitung berdasarkan metode pencarian yang dilakukan.
- findPath(): Menghasilkan jalur dari kata awal hingga kata akhir berdasarkan metode yang dipilih.

2.5. File GUI.java

```
public class GUI extends JFrame implements ActionListener {
    private JTextField startWordField, endWordField, modeField;
    private JTextArea outputArea;

    public GUI() {
        setTitle(title:"Path Finder");
        setSize(width:800, height:600);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new BorderLayout());

        JPanel inputPanel = new JPanel(new GridLayout(rows:4, cols:2));
        inputPanel.add(new JLabel(text:"Start Word:"));
        startWordField = new JTextField();
        inputPanel.add(startWordField);

        inputPanel.add(new JLabel(text:"End Word:"));
        endWordField = new JTextField();
        inputPanel.add(endWordField);

        inputPanel.add(new JLabel(text:"Mode (UCS, GBFS, A*):"));
        modeField = new JTextField();
        inputPanel.add(modeField);

        JButton findButton = new JButton(text:"Find Path");
        findButton.addActionListener(this);
        inputPanel.add(findButton);

        add(inputPanel, BorderLayout.NORTH);

        outputArea = new JTextArea();
        add(new JScrollPane(outputArea), BorderLayout.CENTER);
    }
}
```

Gambar 2.5.1. File GUI.java (1)

```

public void actionPerformed(ActionEvent e) {
    String startWord = startWordField.getText().toLowerCase();
    String endWord = endWordField.getText().toLowerCase();
    String mode = modeField.getText().toUpperCase();

    boolean showResults = true;

    DictSet dictset = new DictSet();
    dictset.loadDictFromFile(filename:"src/dictionary.txt");

    if (!dictset.getDict().contains(startWord) || !dictset.getDict().contains(endWord)) {
        outputArea.setText(t:"Invalid input. Words are not in dictionary.");
        showResults = false;
    }
    if (startWord.length() != endWord.length()) {
        outputArea.setText(t:"Invalid input. Words are of different length.");
        showResults = false;
    }
    if (!(mode.equals(anObject:"UCS") || mode.equals(anObject:"GBFS") || mode.equals(anObject:"A*"))) {
        outputArea.setText(t:"Invalid mode. Please enter UCS, GBFS, or A*.");
        showResults = false;
    }

    Pathfinder pf = new Pathfinder();

    if (showResults) {
        long startTime = System.currentTimeMillis();
        pf.findPath(startWord, endWord, dictset, mode);
        long endTime = System.currentTimeMillis();
        long executionTime = endTime - startTime;

        if (pf.getResultPath().size() == 0) {
            outputArea.setText(t:"No path was found between start word and end word.\n");
        } else {
            outputArea.setText("Path: " + pf.getResultPath() + "\n");
        }
        outputArea.append("Nodes visited: " + pf.getNumVisited() + "\n");
        outputArea.append("Execution time: " + executionTime + " ms");
    }
}

```

Gambar 2.5.2. File GUI.java (2)

Kelas GUI berfungsi untuk menyediakan GUI pada program. Pengguna berinteraksi dengan program melalui GUI tersebut.

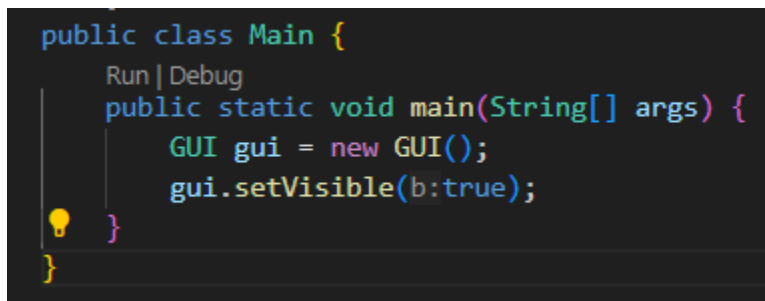
Daftar atribut yang terdapat dalam kelas tersebut adalah:

- startWordField, endWordField, modeField: JTextField untuk menerima masukan dari pengguna.
- outputArea: JTextArea untuk menampilkan hasil keluaran program.

Daftar metode yang terdapat dalam kelas tersebut adalah:

- GUI(): Konstruktor publik GUI yang berfungsi untuk menampilkan tampilan awal kepada pengguna.
- actionPerformed(): Berfungsi untuk menampilkan hasil keluaran kepada pengguna ketika tombol untuk menampilkan hasil ditekan.

2.6. File Main.java



```
public class Main {  
    Run | Debug  
    public static void main(String[] args) {  
        GUI gui = new GUI();  
        gui.setVisible(b:true);  
    }  
}
```

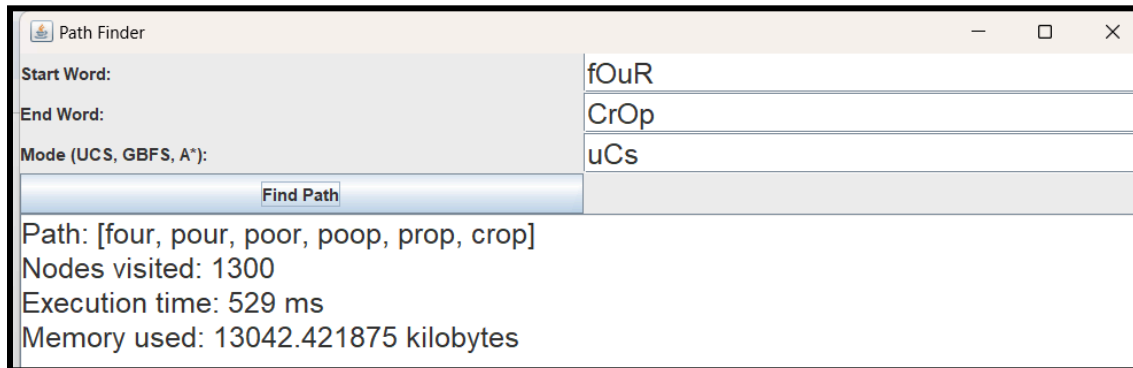
Gambar 2.6.1. Isi Main.java

File Main.java berfungsi sebagai *entrypoint* program yang menginstansiasikan GUI dan menampilkan GUI kepada pengguna.

BAB 3: TANGKAPAN LAYAR

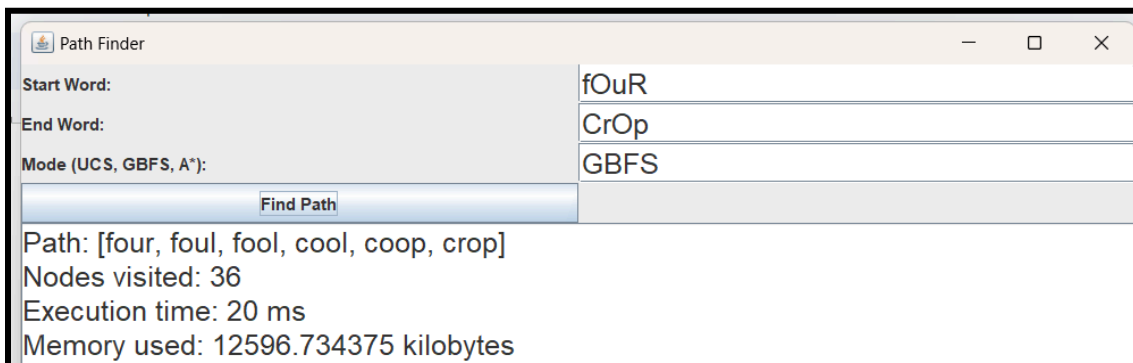
3.1. Test Case 1 (Four - Crop)

3.1.1. UCS



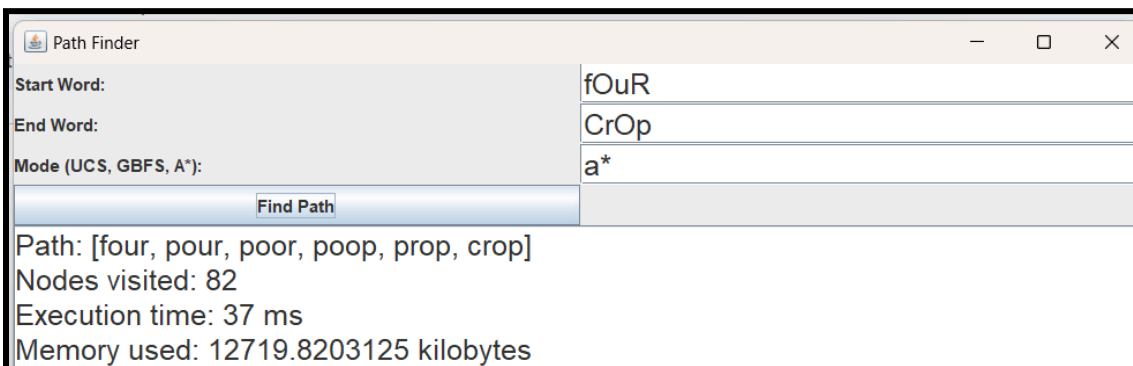
Gambar 3.1.1.1. Kasus pengujian *four - crop* dengan UCS

3.1.2. Greedy Best First Search



Gambar 3.1.2.1. Kasus pengujian *four - crop* dengan Greedy Best First Search

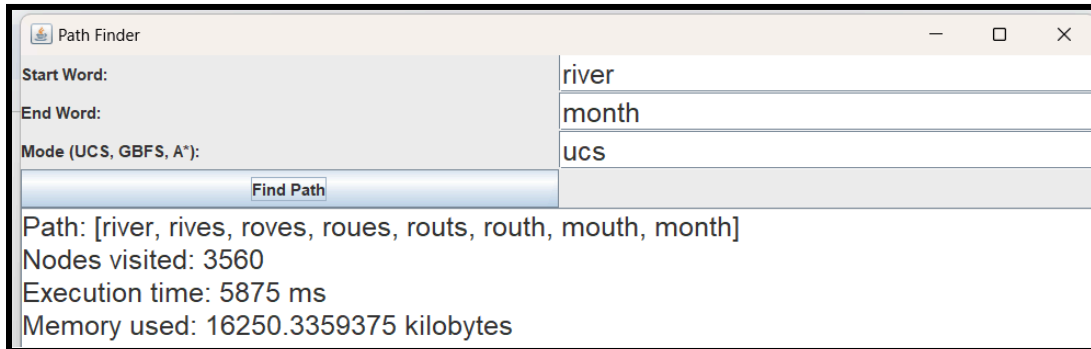
3.1.3. A*



Gambar 3.1.3.1. Kasus pengujian *four - crop* dengan A*

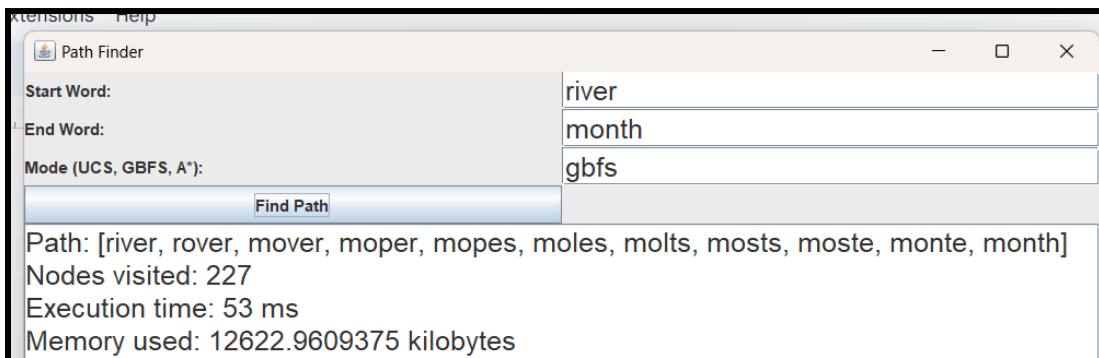
3.2. Test Case 2 (River - Month)

3.2.1. UCS



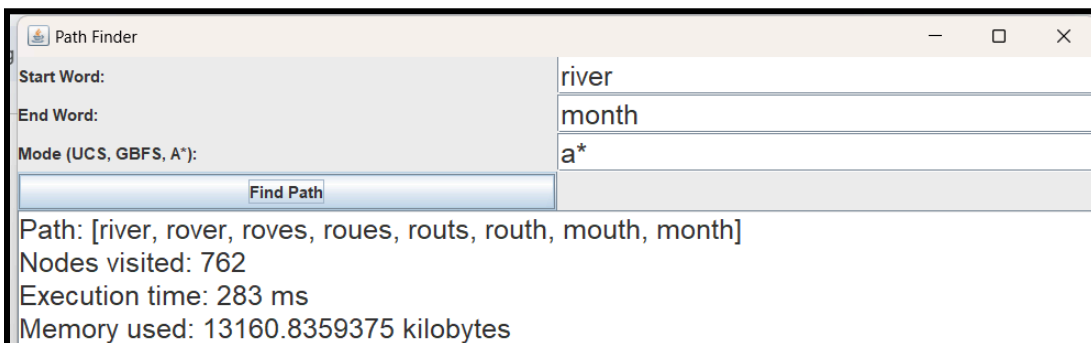
Gambar 3.2.1.1. Kasus pengujian *river - month* dengan UCS

3.2.2. Greedy Best First Search



Gambar 3.2.2.1. Kasus pengujian *river - month* dengan Greedy Best First Search

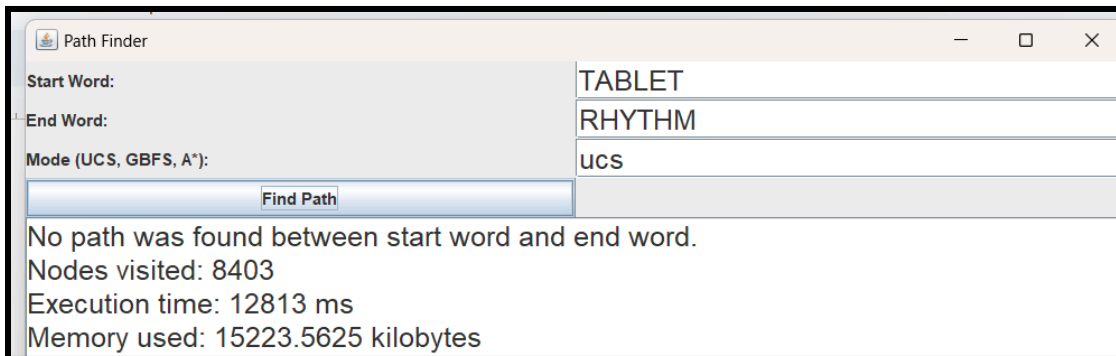
3.2.3. A*



Gambar 3.2.3.1. Kasus pengujian *river - month* dengan A*

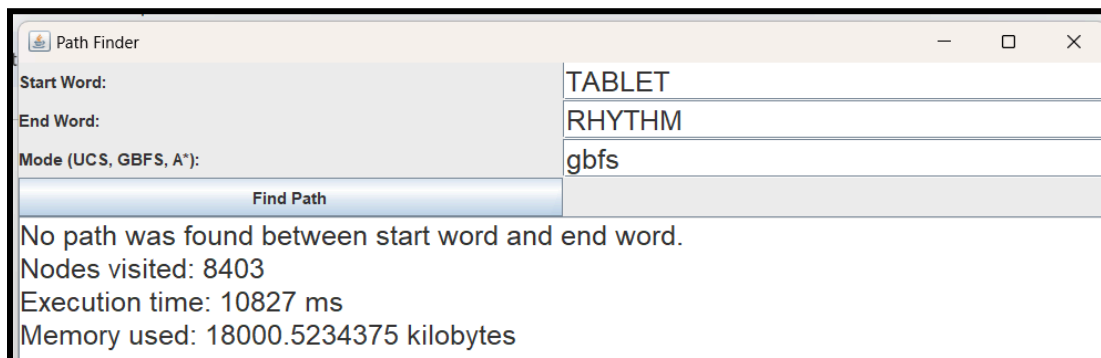
3.3. Test Case 3 (Tablet - Rhythm)

3.3.1. UCS



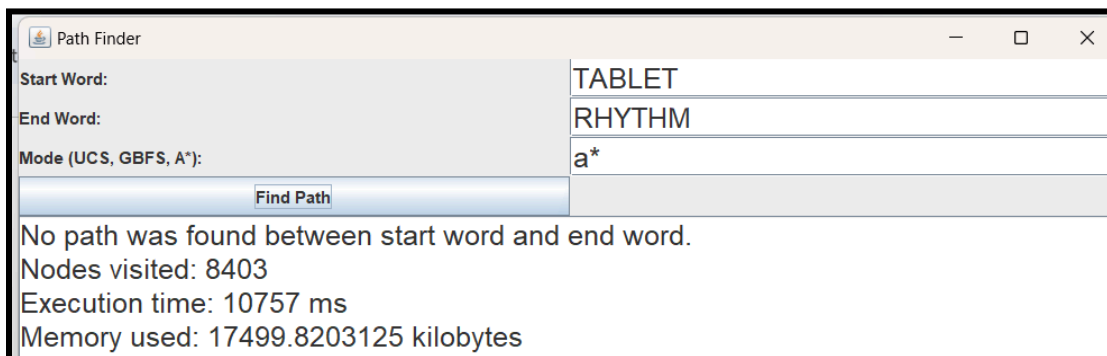
Gambar 3.3.1.1. Kasus pengujian *tablet - rhythm* dengan UCS

3.3.2. Greedy Best First Search



Gambar 3.3.2.1. Kasus pengujian *tablet - rhythm* dengan Greedy Best First Search

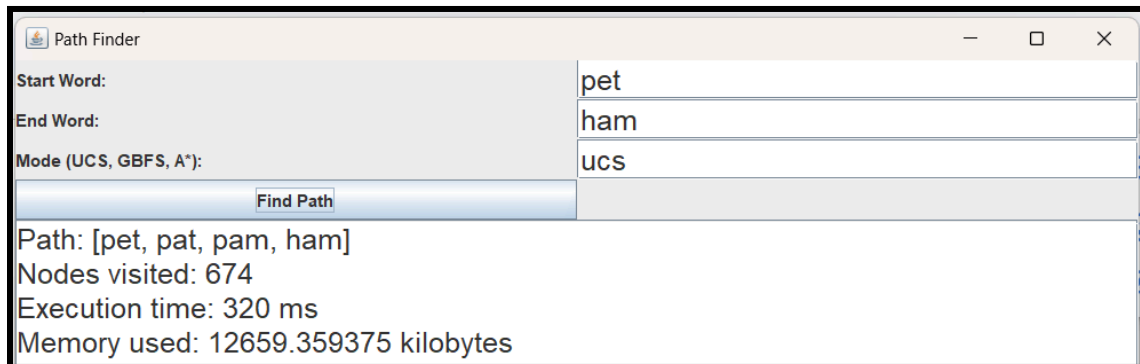
3.3.3. A*



Gambar 3.3.3.1. Kasus pengujian *tablet - rhythm* dengan A*

3.4. Test Case 4 (Pet - Ham)

3.4.1. UCS

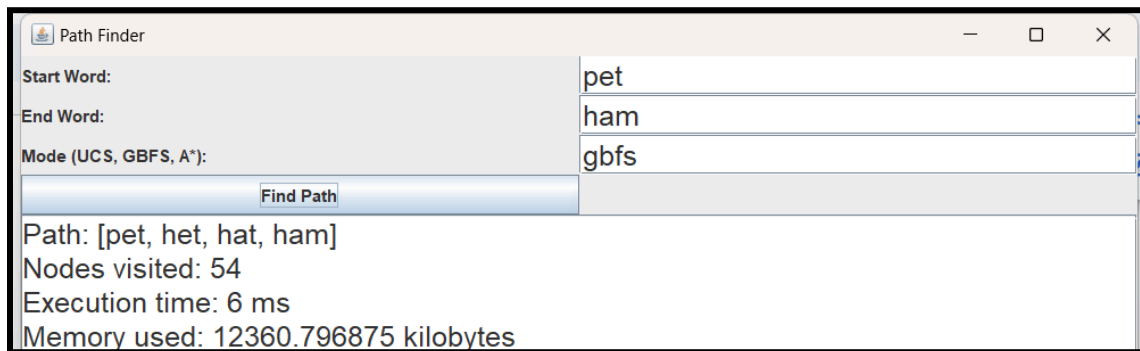


The screenshot shows the 'Path Finder' application window. The 'Start Word' field contains 'pet', the 'End Word' field contains 'ham', and the 'Mode (UCS, GBFS, A*)' dropdown is set to 'ucs'. A 'Find Path' button is visible. Below the button, the results are displayed: 'Path: [pet, pat, pam, ham]', 'Nodes visited: 674', 'Execution time: 320 ms', and 'Memory used: 12659.359375 kilobytes'.

Start Word:	pet
End Word:	ham
Mode (UCS, GBFS, A*):	ucs
<input type="button" value="Find Path"/>	
Path: [pet, pat, pam, ham]	
Nodes visited: 674	
Execution time: 320 ms	
Memory used: 12659.359375 kilobytes	

Gambar 3.4.1.1. Kasus pengujian *pet - ham* dengan UCS

3.4.2. Greedy Best First Search

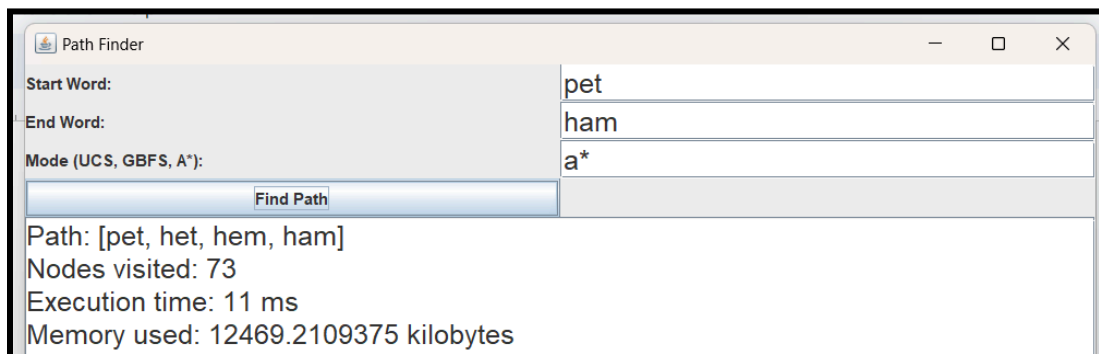


The screenshot shows the 'Path Finder' application window. The 'Start Word' field contains 'pet', the 'End Word' field contains 'ham', and the 'Mode (UCS, GBFS, A*)' dropdown is set to 'gbfs'. A 'Find Path' button is visible. Below the button, the results are displayed: 'Path: [pet, het, hat, ham]', 'Nodes visited: 54', 'Execution time: 6 ms', and 'Memory used: 12360.796875 kilobytes'.

Start Word:	pet
End Word:	ham
Mode (UCS, GBFS, A*):	gbfs
<input type="button" value="Find Path"/>	
Path: [pet, het, hat, ham]	
Nodes visited: 54	
Execution time: 6 ms	
Memory used: 12360.796875 kilobytes	

Gambar 3.4.2.1. Kasus pengujian *pet - ham* dengan Greedy Best First Search

3.4.3. A*



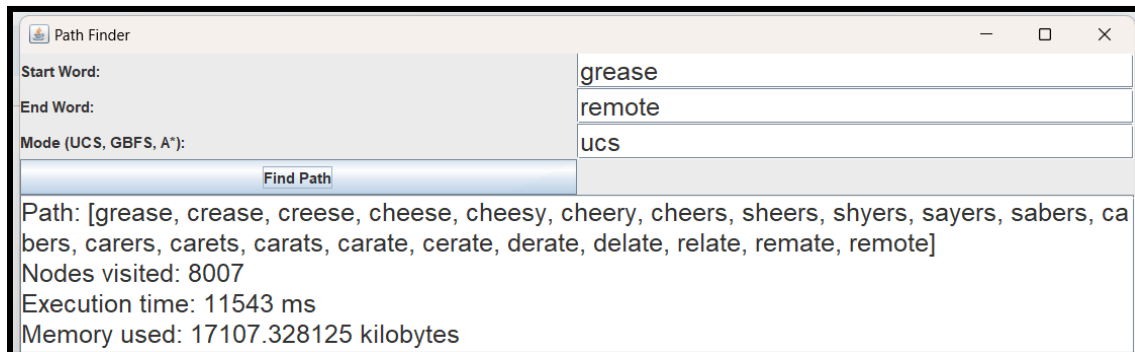
The screenshot shows the 'Path Finder' application window. The 'Start Word' field contains 'pet', the 'End Word' field contains 'ham', and the 'Mode (UCS, GBFS, A*)' dropdown is set to 'a*'. A 'Find Path' button is visible. Below the button, the results are displayed: 'Path: [pet, het, hem, ham]', 'Nodes visited: 73', 'Execution time: 11 ms', and 'Memory used: 12469.2109375 kilobytes'.

Start Word:	pet
End Word:	ham
Mode (UCS, GBFS, A*):	a*
<input type="button" value="Find Path"/>	
Path: [pet, het, hem, ham]	
Nodes visited: 73	
Execution time: 11 ms	
Memory used: 12469.2109375 kilobytes	

Gambar 3.4.3.1. Kasus pengujian *pet - ham* dengan A*

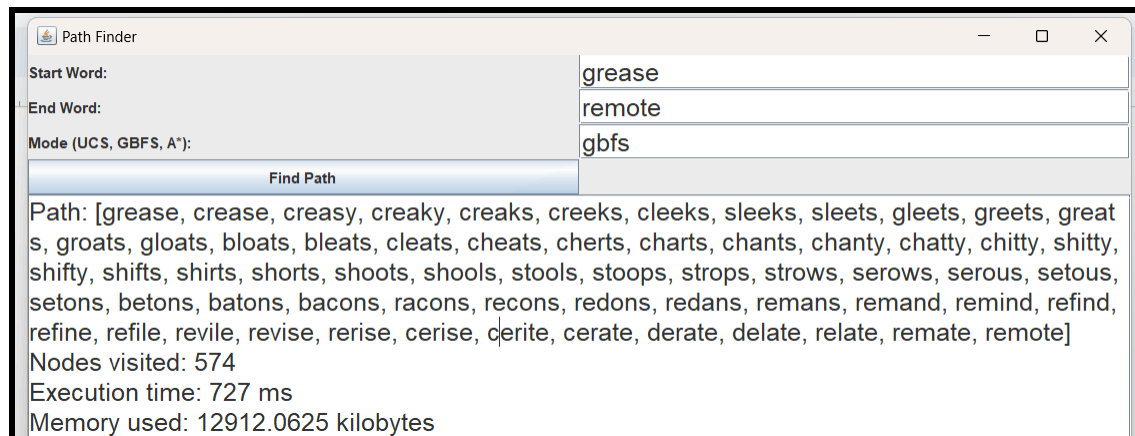
3.5. Test Case 5 (Grease - Remote)

3.5.1. UCS



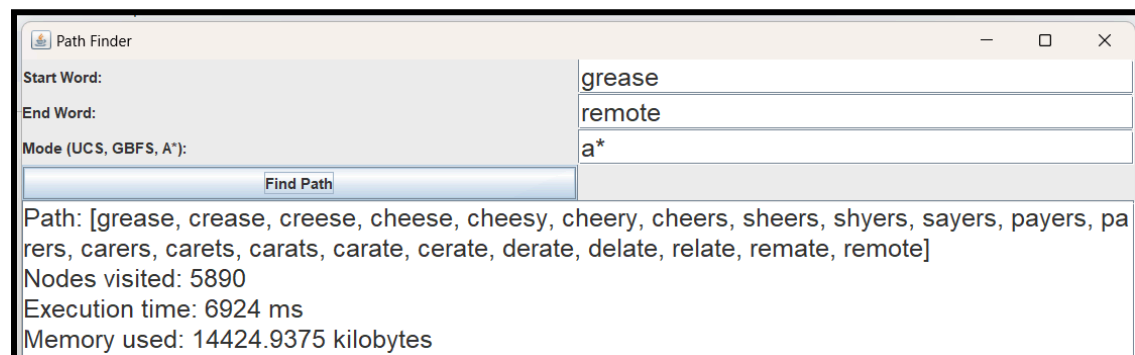
Gambar 3.5.1.1. Kasus pengujian *grease - remote* dengan UCS

3.5.2. Greedy Best First Search



Gambar 3.5.2.1. Kasus pengujian *grease - remote* dengan Greedy Best First Search

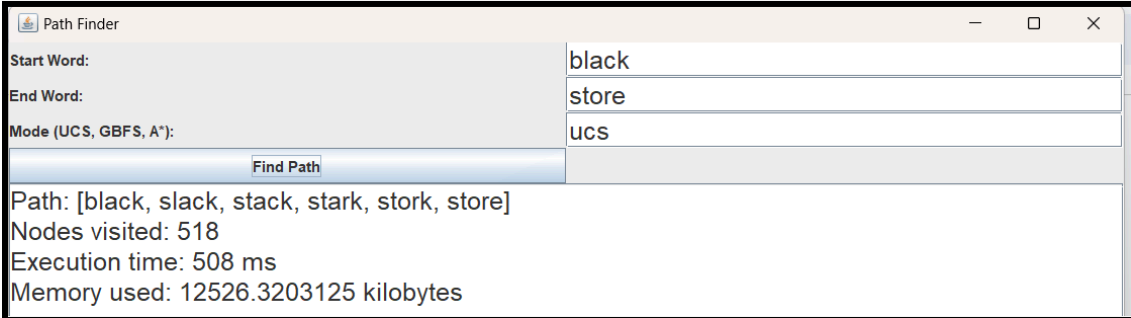
3.5.3. A*



Gambar 3.5.3.1. Kasus pengujian *grease - remote* dengan A*

3.6. Test Case 6 (Black - Store)

3.6.1. UCS

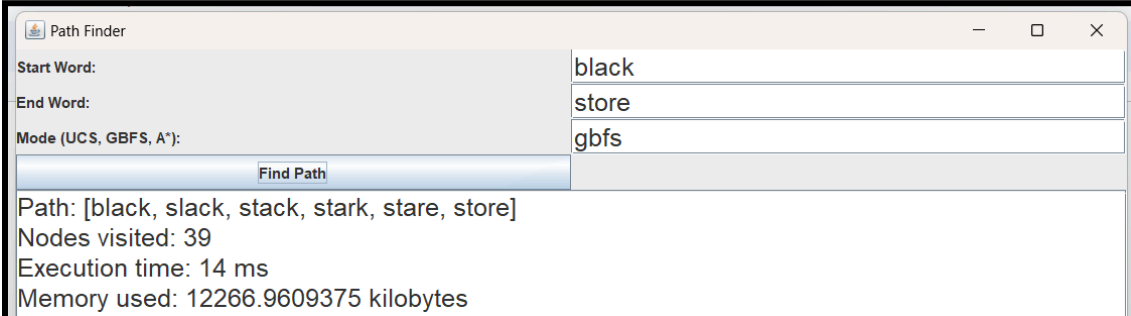


The screenshot shows the 'Path Finder' application window. The 'Start Word' field contains 'black', the 'End Word' field contains 'store', and the 'Mode (UCS, GBFS, A*)' dropdown is set to 'ucs'. A 'Find Path' button is visible. Below the button, the results are displayed: 'Path: [black, slack, stack, stark, stork, store]', 'Nodes visited: 518', 'Execution time: 508 ms', and 'Memory used: 12526.3203125 kilobytes'.

Start Word:	black
End Word:	store
Mode (UCS, GBFS, A*):	ucs
Find Path	
Path: [black, slack, stack, stark, stork, store]	
Nodes visited: 518	
Execution time: 508 ms	
Memory used: 12526.3203125 kilobytes	

Gambar 3.6.1.1. Kasus pengujian *black - store* dengan UCS

3.6.2. Greedy Best First Search

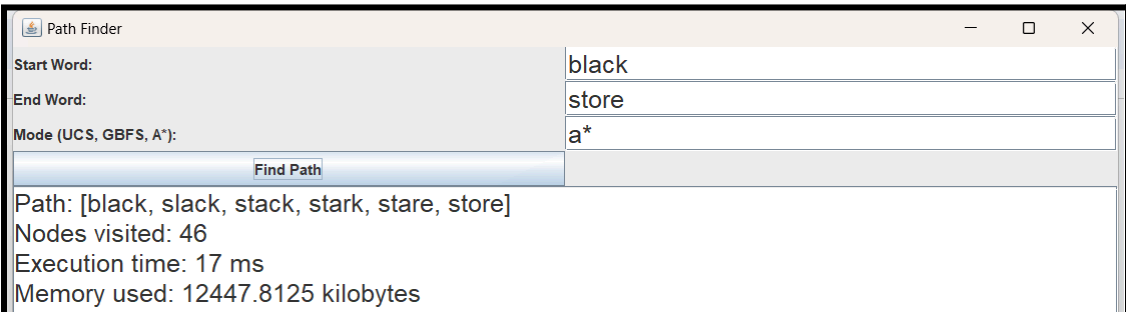


The screenshot shows the 'Path Finder' application window. The 'Start Word' field contains 'black', the 'End Word' field contains 'store', and the 'Mode (UCS, GBFS, A*)' dropdown is set to 'gbfs'. A 'Find Path' button is visible. Below the button, the results are displayed: 'Path: [black, slack, stack, stark, stare, store]', 'Nodes visited: 39', 'Execution time: 14 ms', and 'Memory used: 12266.9609375 kilobytes'.

Start Word:	black
End Word:	store
Mode (UCS, GBFS, A*):	gbfs
Find Path	
Path: [black, slack, stack, stark, stare, store]	
Nodes visited: 39	
Execution time: 14 ms	
Memory used: 12266.9609375 kilobytes	

Gambar 3.6.2.1. Kasus pengujian *black - store* dengan Greedy Best First Search

3.6.3. A*



The screenshot shows the 'Path Finder' application window. The 'Start Word' field contains 'black', the 'End Word' field contains 'store', and the 'Mode (UCS, GBFS, A*)' dropdown is set to 'a*'. A 'Find Path' button is visible. Below the button, the results are displayed: 'Path: [black, slack, stack, stark, stare, store]', 'Nodes visited: 46', 'Execution time: 17 ms', and 'Memory used: 12447.8125 kilobytes'.

Start Word:	black
End Word:	store
Mode (UCS, GBFS, A*):	a*
Find Path	
Path: [black, slack, stack, stark, stare, store]	
Nodes visited: 46	
Execution time: 17 ms	
Memory used: 12447.8125 kilobytes	

Gambar 3.6.3.1. Kasus pengujian *black - store* dengan A*

BAB 4: ANALISIS HASIL PERBANDINGAN SOLUSI

Dari hasil *test case* yang diberikan, terdapat bahwa algoritma UCS dan A* dapat memberikan jalur terpendek antara kata awal dan kata akhir, sedangkan algoritma Greedy Best First Search terkadang tidak memberi jalur terpendek. Jika dilihat dari segi jalur terpendek, algoritma UCS dan A* sudah bersifat optimal, sedangkan algoritma Greedy Best First Search belum optimal. Kasus ketika algoritma Greedy Best First Search tidak memberikan solusi yang optimal dapat dilihat pada bagian 3.2 dan 3.5, khususnya pada Gambar 3.2.1.1, Gambar 3.2.2.1, Gambar 3.2.3.1, Gambar 3.5.1.1, Gambar 3.5.2.1, dan Gambar 3.5.3.1. Kasus tersebut terjadi karena algoritma Greedy Best First Search hanya meninjau optimum lokal dan melihat estimasi bobot antara kata saat ini dan kata akhir tanpa melihat jarak yang sudah terakumulasi dari kata awal hingga kata saat ini.

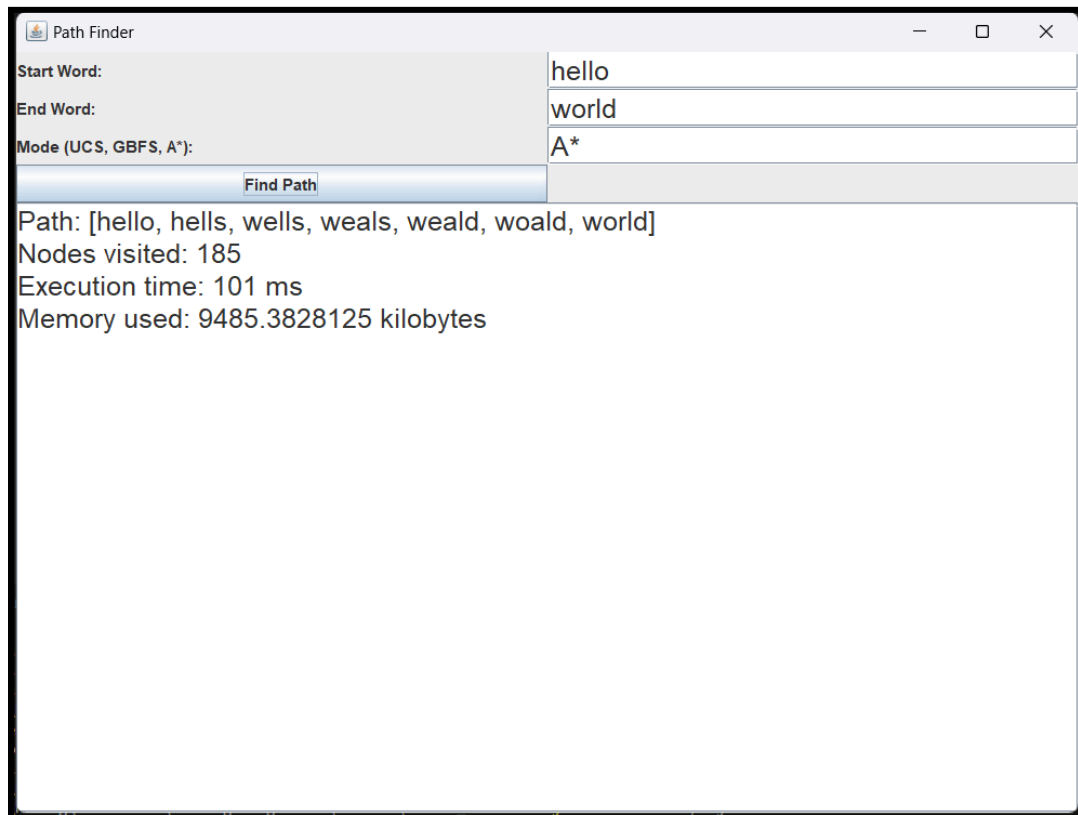
Dari segi waktu eksekusi, terlihat bahwa algoritma yang tercepat adalah algoritma Greedy Best First Search. Algoritma tersebut merupakan algoritma tercepat karena ruang pencariannya memprioritaskan heuristik dan algoritma tersebut tidak melihat jarak yang telah ditempuh untuk mencapai suatu kata. Algoritma yang terlambat adalah UCS karena UCS mencoba semua kemungkinan kata anak tanpa memprioritaskan kata-kata yang menjanjikan (kata yang memiliki perbedaan huruf dengan kata akhir yang sedikit). Waktu eksekusi algoritma A* berada di antara kedua algoritma sebelumnya karena algoritma A* memprioritaskan heuristik dalam pencariannya, namun juga mempertimbangkan jarak antara kata awal hingga kata yang sedang ditelusuri. Terdapat *outlier* dalam segi waktu eksekusi pada *test case* 3. Algoritma UCS memiliki waktu eksekusi yang lebih cepat daripada algoritma A* dan algoritma Greedy Best First Search. Hal tersebut dapat terjadi karena terdapat sedikit variansi dalam waktu eksekusi setiap algoritma yang bersifat *non deterministic*.

Dari segi memori yang dibutuhkan, terlihat pada hasil-hasil *test case* bahwa urutan algoritma dimulai dari yang memerlukan memori tersedikit adalah Greedy Best First Search, A*, dan UCS. Hal tersebut terjadi karena algoritma yang memiliki jumlah kata yang dikunjungi terkecil adalah algoritma Greedy Best First Search dan algoritma yang memiliki jumlah kata yang dikunjungi terbanyak adalah algoritma UCS. Jumlah kata yang dikunjungi oleh algoritma Greedy Best First Search sedikit karena menggunakan heuristik tanpa mementingkan jarak yang sudah ditempuh. Jumlah kata yang dikunjungi oleh algoritma UCS banyak karena tidak memanfaatkan heuristik. Jumlah kata yang dikunjungi oleh algoritma A* berada di antara kedua algoritma sebelumnya karena memanfaatkan heuristik, namun tetap melihat jarak yang sudah ditempuh.

Berdasarkan hasil analisis, dapat disimpulkan bahwa algoritma Greedy Best First Search memiliki waktu eksekusi serta kebutuhan memori terkecil. Akan tetapi, algoritma tersebut tidak menjamin rute terpendek. Algoritma UCS menjamin rute terpendek, namun memiliki waktu eksekusi serta kebutuhan memori terbanyak. Algoritma A* menjamin rute terpendek dengan waktu eksekusi serta kebutuhan memori yang lebih sedikit daripada UCS dengan memanfaatkan heuristik.

BAB 5: PENJELASAN IMPLEMENTASI BONUS

Bonus yang diimplementasikan adalah Graphical User Interface (GUI). GUI dibuat menggunakan package *javax.swing* dan *javax.awt*. Kelas GUI merupakan turunan dari *JFrame* yang merepresentasikan sebuah *window* dan mengimplementasikan *interface ActionListener* agar mampu menanggapi *event* seperti tombol yang ditekan. Pengguna mengisi masukan melalui variabel *startWordField*, *endWordField*, dan *modeField* dan keluaran ditampilkan melalui variabel *outputArea*. Kelas tersebut memiliki konstruktor yang menginisialisasikan beberapa aspek dari GUI, seperti judul dan ukuran *window*. Konstruktor juga membuat sebuah *JButton* dengan label “Find Path”. Kelas GUI akan menjadi *ActionListener* untuk *JButton* tersebut. Kelas tersebut juga memiliki metode *actionPerformed* yang akan menghasilkan rute dari kata awal dan kata akhir yang diberikan pengguna setiap kali tombol “Find Path” ditekan.



Gambar 5.1. Contoh tampilan GUI beserta masukan dan keluaran

BAB 6: PRANALA REPOSITORY

Pranala repository: https://github.com/neokoda/Tucil3_13522108

LAMPIRAN

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