

Laporan Tugas Kecil 3 IF2211 Strategi Algoritma

Semester II Tahun Akademik 2023/2024

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



Disusun oleh:

Muhammad Gilang Ramadhan

13520137

K01

**Program Studi S1 Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2024**

ANALISIS DAN IMPLEMENTASI MASING-MASING ALGORITMA

1. Algoritma Uniform Cost Search

Adapun sesuai dengan definisinya, yaitu dengan melakukan pencarian dengan priority queue dengan menyimpan cost pada simpul n sesuai dengan persamaan:

$$f(n) = g(n)$$

Adapun pada kasus word ladder ini, setiap perubahan kata hanya boleh berbeda satu huruf. Dengan kata lain, kita dapat mendeduksikan $\text{cost} = \text{step}$ untuk setiap penelusurannya. Hal tersebut dapat dibuktikan melalui observasi pada penambahan cost yang dihasilkan untuk setiap child berikutnya bertambah sebanyak 1, maka untuk satu level yang sama nilai cost yang dihasilkan juga pasti sama, demikian seterusnya pertambahan cost yang dihasilkan untuk child selanjutnya mengikuti dengan jumlah kedalaman level yang ditambahkan.

Akibatnya, permasalahan pencarian dengan **algoritma UCS sama dengan algoritma BFS**, tetapi ditambahkan optimasi dengan priority queue untuk mendapatkan node dengan cost paling minimum yang dihitung kumulatif dari root.

Adapun untuk optimasi pembangkitan path yang dilakukan supaya pasti tercapai shortest path, karena cukup diambil satu path saja yang menuju solusi (jika ada), maka mengakibatkan untuk setiap depth i, j dengan $j > i$, mengakibatkan $\text{cost}_j > \text{depth}_i$.

Akibatnya karena ingin mendapatkan nilai cost yang minimum, maka haruslah dihindari untuk mengunjungi node tersebut.

2. Algoritma Best First Search

Adapun sesuai dengan definisinya, yaitu dengan melakukan pencarian dengan priority queue dengan menyimpan cost pada simpul n sesuai dengan persamaan:

$$f(n) = h(n)$$

Adapun pada kasus word ladder ini, nilai heuristik $h(n)$ dapat dihitung melalui banyaknya huruf yang berbeda antara kata tersebut dengan kata tujuan. Namun, pada kondisi ini tidak mempertimbangkan actual cost yang diperoleh pada saat penelusuran yang melibatkan cost yang dihasilkan sebelumnya, yang bisa saja mengakibatkan nilai costnya terjebak di local optima, yang sebenarnya belum tentu menghasilkan optimum global.

Sehingga GBFS tidak menjamin untuk selalu mencapai solusi optimum pada permasalahan word ladder.

3. Algoritma A*

Adapun sesuai dengan definisinya, yaitu dengan melakukan pencarian dengan priority queue dengan menyimpan cost pada simpul n sesuai dengan persamaan:

$$f(n) = g(n) + h(n)$$

Adapun pada kasus word ladder ini, nilai heuristik $h(n)$ dapat dihitung melalui banyaknya huruf yang berbeda antara kata tersebut dengan kata tujuan. Akibatnya, banyaknya transformasi yang perlu dilakukan dengan kata tujuan \geq banyaknya huruf yang berbeda. Sehingga nilai heuristik tersebut dapat mengakibatkan **algoritma A* menjadi Admissible yang menjamin tercapainya solusi optimal.**

Perhatikan juga bahwa, karena terdapat heuristik yang dilakukan maka algoritma A* akan membuat pembangkitannya menjadi lebih sedikit daripada algoritma UCS, dan jika dibandingkan dengan GBFS yang mungkin saja terjebak pada local optima yang mengakibatkan penelusuran nodenya menjadi lebih banyak. **Akibatnya secara umum A* lebih efisien daripada UCS dan GBFS.**

Dari ketiga algoritma tersebut, dapat dideduksikan langkah-langkah sebagai berikut untuk menyelesaikan masalah tersebut.

- Inisialisasi kontainer (bisa apa saja, misal queue, priority queue, stack, dan lain-lain).
- Bangkitkan Simpul Akar, yaitu simpul yang merepresentasikan start word dan masukkan ke dalam kontainer.
- Buang simpul pada antrian paling depan dari kontainer sebagai simpul yang akan diekspansi (current node)
- Jika current node sama dengan goal word, maka kembalikan rute dari akar menuju simpul root.
- Jika tidak, lanjutkan penelusuran pada seluruh kata yang bisa dibentuk dari setiap simpul ekspansi dengan melakukan perubahan pada satu huruf saja.

- Lakukan iterasi terhadap seluruh kata yang telah dibentuk, kemudian untuk setiap current node tersebut jika sudah dikunjungi (bisa dicek dengan melakukan HashMap) maka dapat diskip, dan jika belum maka buatlah simpul baru sebagai next child.
- Untuk setiap prioritas pengurutan node pada antrian disesuaikan dengan masing-masing algoritma yang digunakan:
 1. UCS: $g(n)$
 2. GBFS: $h(n)$
 3. A*: $g(n) + h(n)$
- Lakukan iterasi pada langkah 3 sampai 6 hingga kontainer kosong.

SOURCE CODE PROGRAM DALAM BAHASA PEMROGRAMAN JAVA

Struktur Data yang dipakai

1. Node
 - word: string
 - cost: integer
2. HashMap<string, string> untuk menyimpan parent
3. HashMap<string, integer> untuk menyimpan distance
4. Priority Queue
5. Queue
6. LinkedList (sebagai queue yang kontigu pada UCS)

Class AStar.java

```
import java.util.PriorityQueue;
import java.util.*;

public class AStar extends Algorithm {

    public AStar(Map<String, Boolean> dictionary) {
        setDictionary(dictionary);
        setPath(new ArrayList<>());
        setSolveStatus(false);
        setAlphabets(Constant.ALPHABETS);
    }

    private int getHeuristicCostToGoal(String currWord) {
        int ret = 0;
        String goalWord = getGoal();
        int sz = currWord.length();
        for (int i = 0; i < sz; i++) {
            if (currWord.charAt(i) != goalWord.charAt(i)) {
```

```

        ret++;
    }
}

return ret;
}

private void traverseAndReversePath(Map<String, String> parent, String
startWord, String goalWord) {
    String currWord = goalWord;
    while (!Objects.equals(currWord, startWord)) {
        addWord(currWord);
        currWord = parent.get(currWord);
    }
    addWord(startWord);
    reversePath();
}

private void setToDefault() {
    setTotalVisitedNodes(0);
    setSolveStatus(false);
    clearPath();
}

private long getTimeNow() {
    return System.nanoTime();
}

public void solve(String startWord, String goalWord) throws Exception {
    setToDefault();
    setStart(startWord);
    setGoal(goalWord);
    long currTime = getTimeNow();
    if (startWord.length() != goalWord.length()) {

```

```

        throw new Exception("Invalid Input (Lengths are not the same)");
    }

    PriorityQueue<Node> queue = new PriorityQueue<>();
    queue.add(new Node(startWord, getHeuristicCostToGoal(startWord)));

    Map<String, Integer> dist = new HashMap<>();
    dist.put(startWord, 0);
    Map<String, String> parent = new HashMap<>();
    while (!queue.isEmpty()) {
        Node currNode = queue.poll();
        incTotalVisNodes();

        if (currNode.getWord().equals(goalWord)) {
            setSolveStatus(true);
            traverseAndReversePath(parent, startWord, goalWord);
            break;
        }

        String curr = currNode.getWord();
        int currLen = curr.length();
        for (int i = 0; i < currLen; i++) {
            for (char alphabet : getAlphabets()) {
                if (curr.charAt(i) == alphabet) continue;
                String next = curr.substring(0, i) + alphabet +
curr.substring(i + 1);
                if (isContainKey(next)) {
                    int heuristic = getHeuristicCostToGoal(next);
                    int prevCost = dist.getOrDefault(next, 0) + heuristic;
                    int currCost = dist.getOrDefault(curr, 0) + 1 +
heuristic;

                    if (prevCost ≤ currCost && parent.containsKey(next)) {
                        continue;

```

```

        }
        parent.put(next, curr);
        dist.put(next, dist.getOrDefault(curr, 0) + 1);
        queue.add(new Node(next, dist.get(next) + heuristic));
// f(n) = g(n) + h(n)
    }
}
}
}

double calTime = (getNowTime() - currTime) / 1000000.0;
setTimeExec(calTime);
}
}

```

Class GreedyBestFirstSearch.java

```

import java.util.*;
import java.util.PriorityQueue;

public class GreedyBestFirstSearch extends Algorithm {

    public GreedyBestFirstSearch(Map<String, Boolean> dictionary) {
        setDictionary(dictionary);
        setPath(new ArrayList<>());
        setSolveStatus(false);
        setAlphabets(Constant.ALPHABETS);
    }

    private int getHeuristicCostToGoal(String currWord) {
        int ret = 0;
        String goalWord = getGoal();
        int sz = currWord.length();
    }
}

```



```

        for (int i = 0; i < sz; i++) {
            if (currWord.charAt(i) != goalWord.charAt(i)) {
                ret++;
            }
        }
        return ret;
    }

```

```

    private void traverseAndReversePath(Map<String, String> parent, String
startWord, String goalWord) {
        String currWord = goalWord;
        while (!Objects.equals(currWord, startWord)) {
            addWord(currWord);
            currWord = parent.get(currWord);
        }
        addWord(startWord);
        reversePath();
    }

```

```

    private void setToDefault() {
        setTotalVisitedNodes(0);
        setSolveStatus(false);
        clearPath();
    }

```

```

    private long getTimeNow() {
        return System.nanoTime();
    }

```

```

    public void solve(String startWord, String goalWord) throws Exception {
        setToDefault();
        setStart(startWord);
        setGoal(goalWord);
    }

```

```

    long currTime = getTimeNow();
    if (startWord.length() != goalWord.length()) {
        throw new Exception("Invalid Input (Lengths are not the same)");
    }

    PriorityQueue<Node> queue = new PriorityQueue<>();
    queue.add(new Node(startWord, getHeuristicCostToGoal(startWord)));

    Map<String, String> parent = new HashMap<>();
    while (!queue.isEmpty()) {
        Node currNode = queue.poll();
        incTotalVisNodes();

        if (currNode.getWord().equals(goalWord)) {
            setSolveStatus(true);
            traverseAndReversePath(parent, startWord, goalWord);
            break;
        }

        String curr = currNode.getWord();
        int currLen = curr.length();
        for (int i = 0; i < currLen; i++) {
            for (char alphabet : getAlphabets()) {
                String next = curr.substring(0, i) + alphabet +
curr.substring(i + 1);
                if (isContainKey(next) && !parent.containsKey(next)) {
                    parent.put(next, curr);
                    int heuristic = getHeuristicCostToGoal(next);
                    queue.add(new Node(next, heuristic));
                }
            }
        }
    }
}

```

```

        double calTime = (getNowTime() - currTime) / 1000000.0;
        setTimeExec(calTime);
    }
}

```

Class UniformCostSearch.java

```

import java.util.*;
import java.util.PriorityQueue;

public class UniformCostSearch extends Algorithm{
    public UniformCostSearch(Map<String, Boolean> dictionary){
        setDictionary(dictionary);
        setPath(new ArrayList<>());
        setSolveStatus(false);
        setAlphabets(Constant.ALPHABETS);
    }

    private void traverseAndReversePath(Map<String, String> parent, String
startWord, String goalWord) {
        String currWord = goalWord;
        while (!Objects.equals(currWord, startWord)) {
            addWord(currWord);
            currWord = parent.get(currWord);
        }
        addWord(startWord);
        reversePath();
    }

    private void setToDefault(){
        setTotalVisitedNodes(0);
        setSolveStatus(false);
    }
}

```

```
clearPath();  
}
```

```
private long getTimeNow(){  
    return System.nanoTime();  
}
```

// In this case, it happened with cost equal to step, so it most like similar as BFS

```
public void solve(String startWord, String goalWord) throws Exception {  
    this.setToDefault();  
    this.setStart(startWord);  
    this.setGoal(goalWord);  
    long currTime = getTimeNow();  
    if(startWord.length() != goalWord.length()){  
        throw new Exception("Invalid Input (Length are not same)");  
    }  
}
```

```
Map<String, String> parent = new HashMap<>();  
Queue<String> queue = new LinkedList<>();  
queue.add(startWord);
```

```
while(!queue.isEmpty()){  
    int sz = queue.size();  
    for(int i = 0; i<sz; i++){  
        String curr = queue.poll();  
        this.incTotalVisNodes();  
  
        if(Objects.equals(curr, goalWord)){  
            this.setSolveStatus(true);  
            this.traverseAndReversePath(parent, startWord, goalWord);  
            break;  
        }  
    }  
}
```

```

    }

    assert curr != null;
    int subNodeSz = curr.length();
    for(int j = 0; j<subNodeSz; j++){
        for (char alphabet : getAlphabets()) {
            if(curr.charAt(j) == alphabet) continue;

            String next = curr.substring(0, j) + alphabet +
                curr.substring(j + 1);

            if (isContainKey(next)
                && !parent.containsKey(next)) {
                parent.put(next, curr);
                queue.add(next);
            }
        }
    }
}

if(this.getIsSolvable()){
    break;
}

}

Double calTime = (getTimeNow()-currTime)/1000000.0;
setTimeExec(calTime);
}
}

```

Pada setiap class AStar.java, GreedyBestFirstSearch.java, dan UniformCostSearch.java tersebut, dilakukan inheritance terhadap interface Solver.java dan implementasi class Algorithm.java.

Pada class AStart dan GreedyBestFirstSearch tersebut, terdapat beberapa method yang sama dan fungsinya juga sama, yaitu pada fungsi perhitungan heuristic. Sedangkan untuk UniformCostSearch, sebenarnya implementasinya mirip dengan algoritma BFS karena cost = step, tetapi dilakukan optimasi dengan menggunakan priority queue pada penyimpanan setiap node-nya. Berikut disajikan interface Solver yang menjadi dasar class yang dipakai.

Interface Solver

```
import java.util.List;
import java.util.Map;

public interface Solver {
    public void setSolveStatus(boolean status);
    public void setPath(List<String> path);
    public void setDictionary(Map<String, Boolean> dictionary);
    public void setStart(String start);
    public void setGoal(String goal);
    public void setTotalVisitedNodes(int total);
    public void setTimeExec(Double time);
    public void reversePath();
    public void addWord(String word);
    public void clearPath();
    public boolean getIsSolvable();
    public List<String> getPath();
    public String getStart();
    public String getGoal();
    public Integer getTotalVisitedNodes();
    public Integer getSolutionLength();
    public Double getTimeExec();
    public boolean isContainKey(String nextWord);
    public void solve(String startWord, String goalWord) throws Exception;
    public void setAlphabets(char[] alphabets);
    public char[] getAlphabets();
}
```

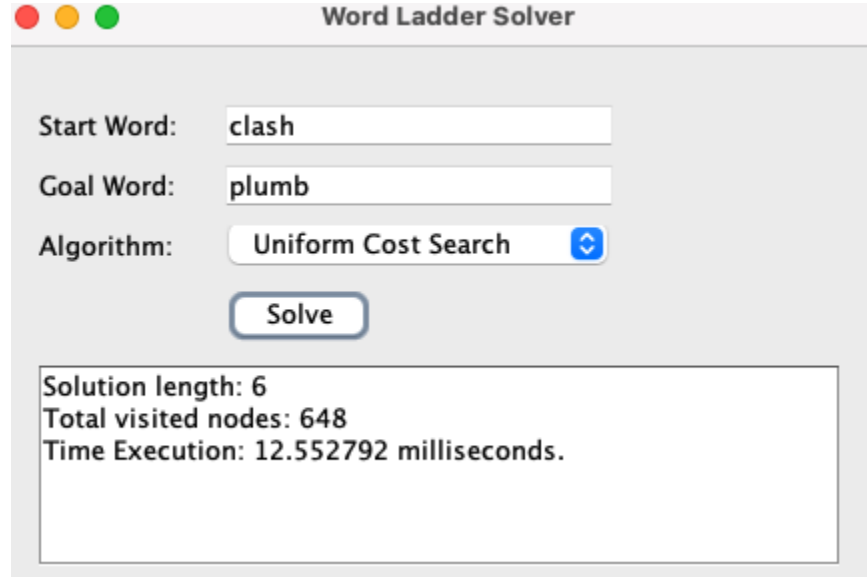
```
}
```

Adapun design pattern yang digunakan pada pembuatan word ladder solver ini ialah Strategy Pattern, karena akan cenderung menghemat penggunaan method yang berulang dan memudahkan maintenance method yang krusial pada masing-masing class inheritance yang diimplementasikan mandiri. Yang mana untuk setiap method bantu yang tidak terlalu berhubungan dengan logic algoritma A*, GBFS, ataupun UCS diimplementasikan pada class Algorithm yang mengimplementasikan interface Solver.

HASIL PERBANDINGAN SOLUSI DAN ANALISIS

1. Test Case 1

- Uniform Cost Search



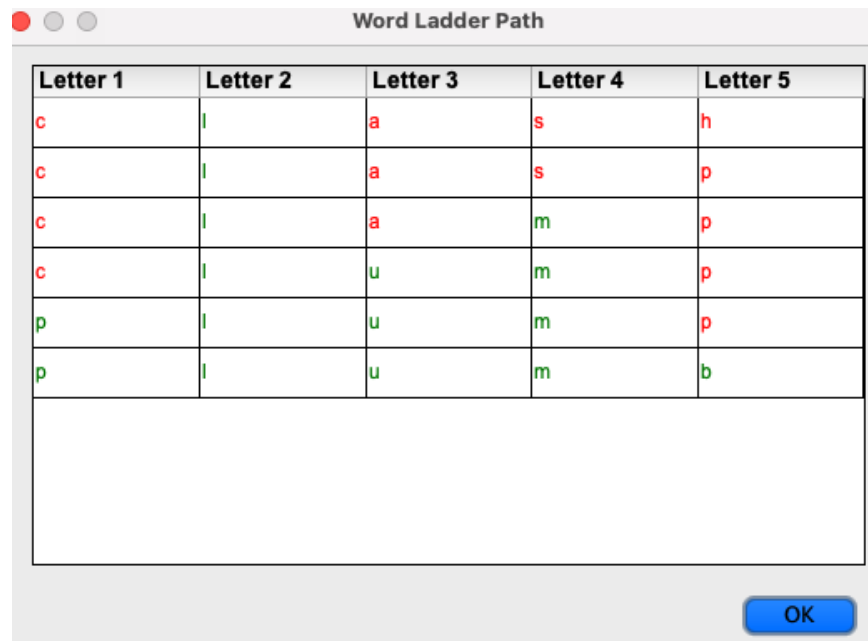
Word Ladder Solver

Start Word:

Goal Word:

Algorithm:

Solution length: 6
Total visited nodes: 648
Time Execution: 12.552792 milliseconds.



Word Ladder Path

Letter 1	Letter 2	Letter 3	Letter 4	Letter 5
c	l	a	s	h
c	l	a	s	p
c	l	a	m	p
c	l	u	m	p
p	l	u	m	p
p	l	u	m	b

- Greedy Best First Search

Word Ladder Solver

Start Word:

clash

Goal Word:

plumb

Algorithm:

Greedy Best-First Sea...

Solve

Solution length: 11

Total visited nodes: 73

Time Execution: 6.019667 milliseconds.

Word Ladder Path

Letter 1	Letter 2	Letter 3	Letter 4	Letter 5
c	l	a	c	k
p	l	a	c	k
p	l	a	c	e
p	l	a	t	e
b	l	a	t	e
b	l	a	m	e
b	l	u	m	e
p	l	u	m	e
p	l	u	m	b


OK

- A*

Word Ladder Solver

Start Word:

Goal Word:

Algorithm: 

Solution length: 6
 Total visited nodes: 15
 Time Execution: 1.750417 milliseconds.

Word Ladder Path

Letter 1	Letter 2	Letter 3	Letter 4	Letter 5
c	l	a	s	h
c	l	a	s	p
c	l	a	m	p
c	l	u	m	p
p	l	u	m	p
p	l	u	m	b

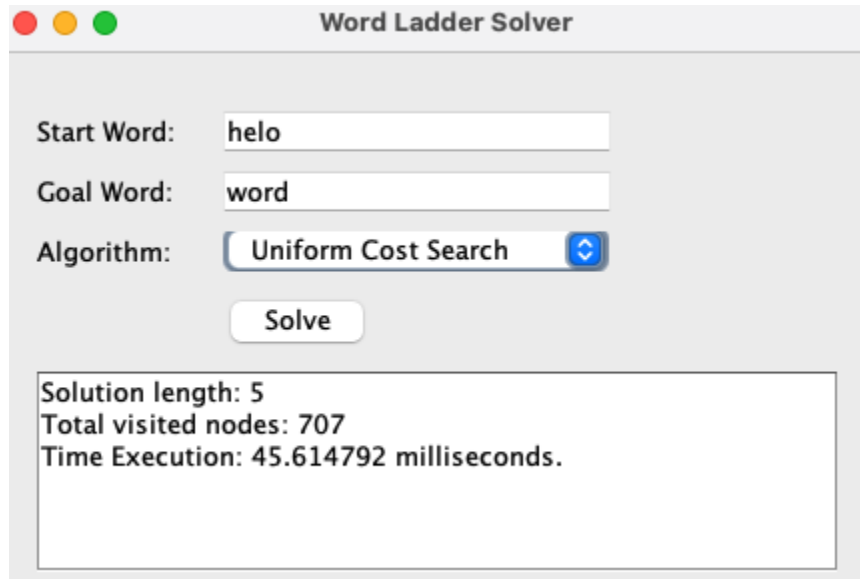
Dari test case tersebut terlihat bahwa algoritma A* dan UCS lebih optimal daripada algoritma GBFS dikarenakan pada kasus ini terdapat jalur node yang terjebak di local optima, dan untuk pembangkitan node yang dilakukan pada UCS cenderung lebih signifikan lebih banyak daripada dengan algoritma A* karena untuk level yang lebih dalam pembangkitan child pada graf yang dihasilkan lebih banyak daripada node yang berada pada level sebelumnya, hal tersebut dapat dilihat dari banyaknya node yang dikunjungi oleh setiap algoritma tersebut. Sehingga diperoleh:

- Dari sisi optimalitas, jelas $UCS = A^* > GBFS$.
- Dari sisi waktu eksekusi $UCS > GBFS > A^*$.

- Dari sisi penggunaan memori UCS > GBFS > A*.

2. Test Case 2

- Uniform Cost Search



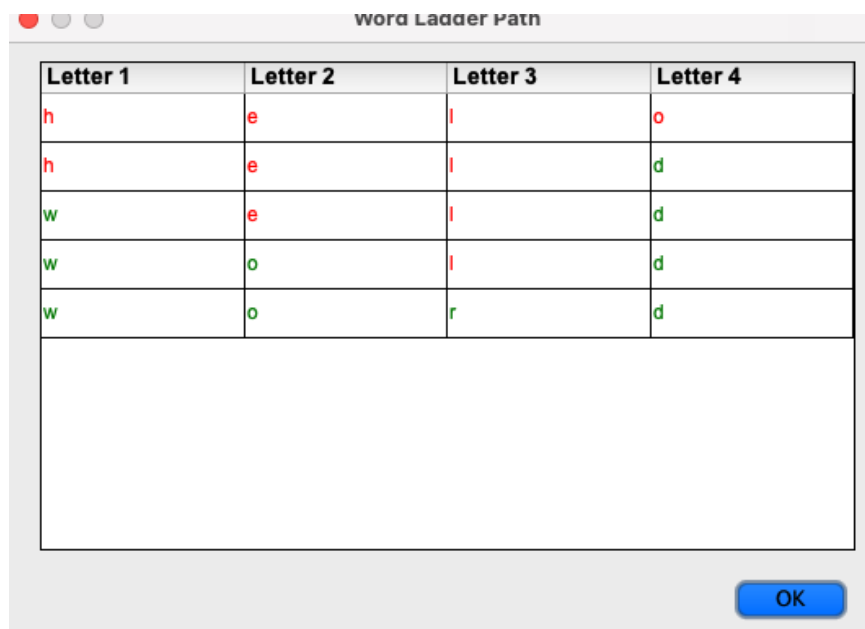
Word Ladder Solver

Start Word:

Goal Word:

Algorithm:

Solution length: 5
 Total visited nodes: 707
 Time Execution: 45.614792 milliseconds.



Word Ladder Path


Letter 1	Letter 2	Letter 3	Letter 4
h	e	l	o
h	e	l	d
w	e	l	d
w	o	l	d
w	o	r	d

- Greedy Best First Search

Word Ladder Solver

Start Word:

Goal Word:

Algorithm: 

Solution length: 6

Total visited nodes: 6

Time Execution: 1.27575 milliseconds.

Word Ladder Path

Letter 1	Letter 2	Letter 3	Letter 4
h	e	l	o
h	e	l	d
h	e	r	d
h	a	r	d
w	a	r	d
w	o	r	d

- A*

Word Ladder Solver

Start Word: helo

Goal Word: word

Algorithm: A*

Solve

Solution length: 5

Total visited nodes: 8

Time Execution: 1.820042 milliseconds.

Word Ladder Path

Letter 1	Letter 2	Letter 3	Letter 4
h	e	l	o
h	e	l	d
h	o	l	d
w	o	l	d
w	o	r	d

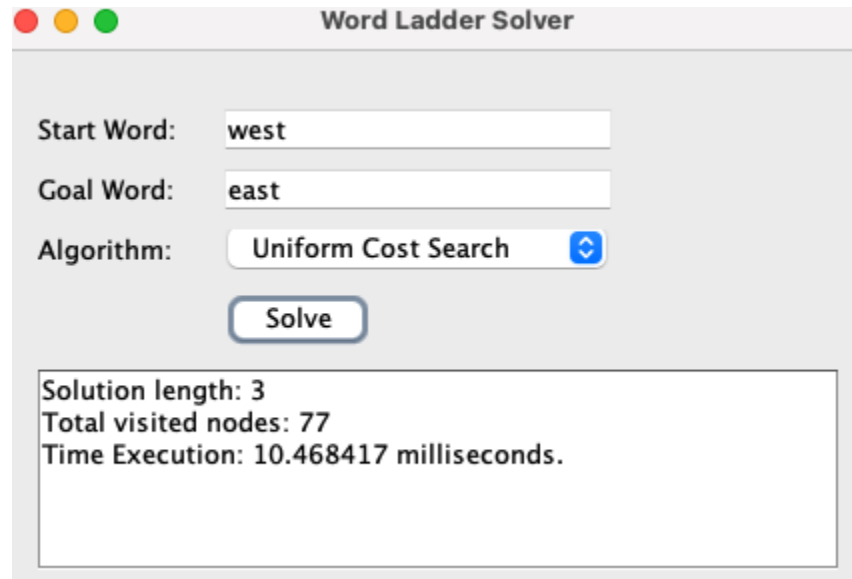
OK

Dari test case tersebut terlihat bahwa algoritma A* dan UCS lebih optimal daripada algoritma GBFS dikarenakan pada kasus ini terdapat jalur node yang terjebak di local optima, dan untuk pembangkitan node yang dilakukan pada UCS cenderung lebih signifikan lebih banyak daripada dengan algoritma A* karena untuk level yang lebih dalam pembangkitan child pada graf yang dihasilkan lebih banyak daripada node yang berada pada level sebelumnya, hal tersebut dapat dilihat dari banyaknya node yang dikunjungi oleh setiap algoritma tersebut. Kemudian untuk kasus level yang melebar ini penelusuran A* menjadi lebih banyak dari pada GBFS, Sehingga diperoleh:

- Dari sisi optimalitas, jelas $UCS = A^* > GBFS$.
- Dari sisi waktu eksekusi $GBFS > UCS > A^*$.
- Dari sisi penggunaan memori $GBFS > UCS > A^*$.

3. Test Case 3

- Uniform Cost Search



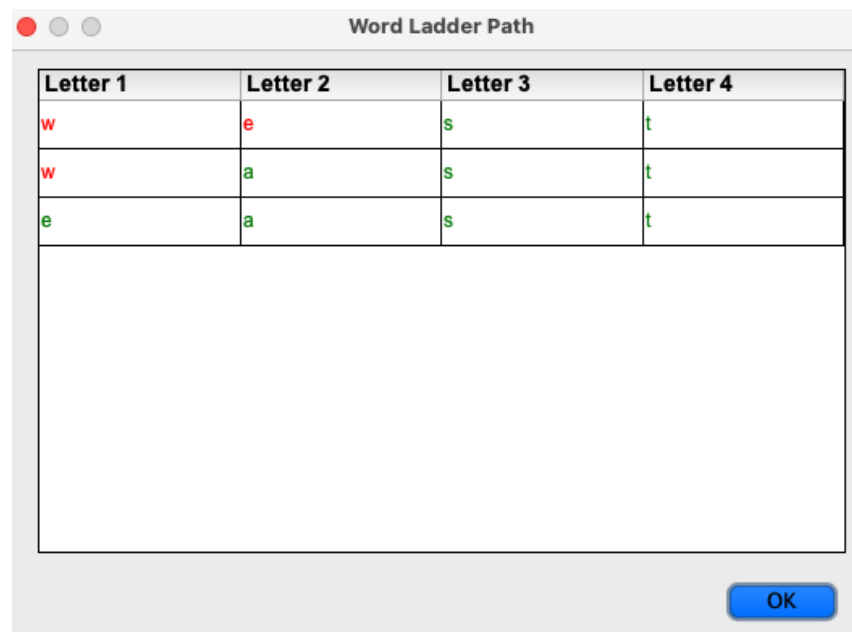
Word Ladder Solver

Start Word:

Goal Word:

Algorithm:

Solution length: 3
Total visited nodes: 77
Time Execution: 10.468417 milliseconds.



Word Ladder Path

Letter 1	Letter 2	Letter 3	Letter 4
w	e	s	t
w	a	s	t
e	a	s	t

- Greedy Best First Search

Word Ladder Solver

Start Word:

west

Goal Word:

east

Algorithm:

Greedy Best-First Sea...

Solve

Solution length: 3

Total visited nodes: 3

Time Execution: 1.489708 milliseconds.

Word Ladder Path

Letter 1	Letter 2	Letter 3	Letter 4
w	e	s	t
w	a	s	t
e	a	s	t

OK

- A*

Word Ladder Solver

Start Word:

Goal Word:

Algorithm:

Solution length: 3
Total visited nodes: 3
Time Execution: 0.977625 milliseconds.

Word Ladder Path

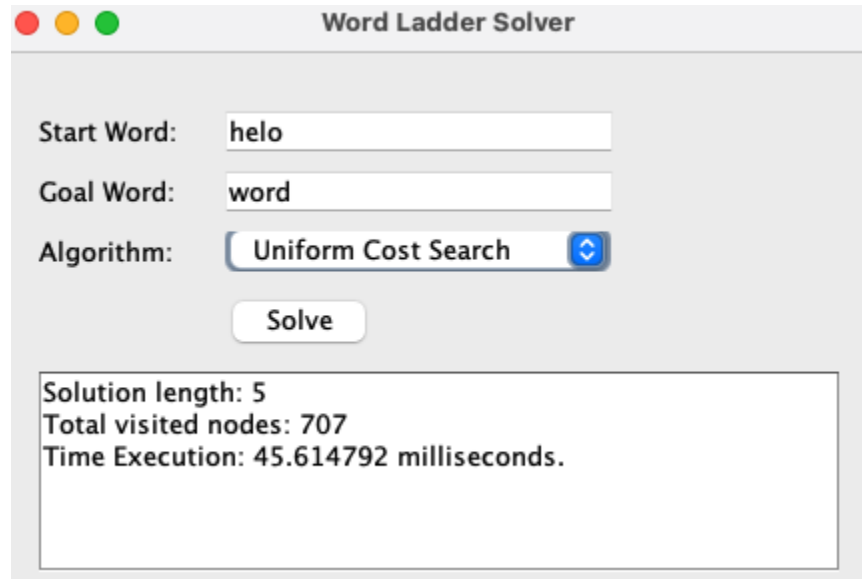
Letter 1	Letter 2	Letter 3	Letter 4
w	e	s	t
w	a	s	t
e	a	s	t

Dari test case tersebut terlihat bahwa algoritma A* dan UCS lebih optimal daripada algoritma GBFS dikarenakan pada kasus ini terdapat jalur node yang terjebak di local optima, dan untuk pembangkitan node yang dilakukan pada UCS cenderung lebih signifikan lebih banyak daripada dengan algoritma A* karena untuk level yang lebih dalam pembangkitan child pada graf yang dihasilkan lebih banyak daripada node yang berada pada level sebelumnya, hal tersebut dapat dilihat dari banyaknya node yang dikunjungi oleh setiap algoritma tersebut. Sehingga diperoleh:

- Dari sisi optimalitas, jelas $UCS = A^* = GBFS$.
- Dari sisi waktu eksekusi $UCS > GBFS > A^*$.
- Dari sisi penggunaan memori $UCS > GBFS = A^*$.

4. Test Case 4

- Uniform Cost Search



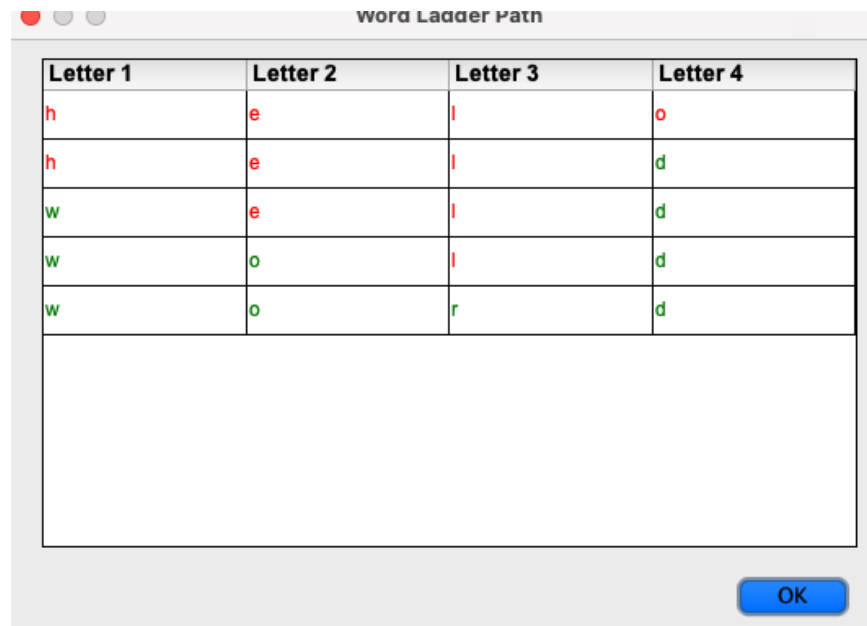
Word Ladder Solver

Start Word:

Goal Word:

Algorithm:

Solution length: 5
 Total visited nodes: 707
 Time Execution: 45.614792 milliseconds.



Word Ladder Path


Letter 1	Letter 2	Letter 3	Letter 4
h	e	l	o
h	e	l	d
w	e	l	d
w	o	l	d
w	o	r	d

- Greedy Best First Search

Word Ladder Solver

Start Word:

Goal Word:

Algorithm: 

Solution length: 6

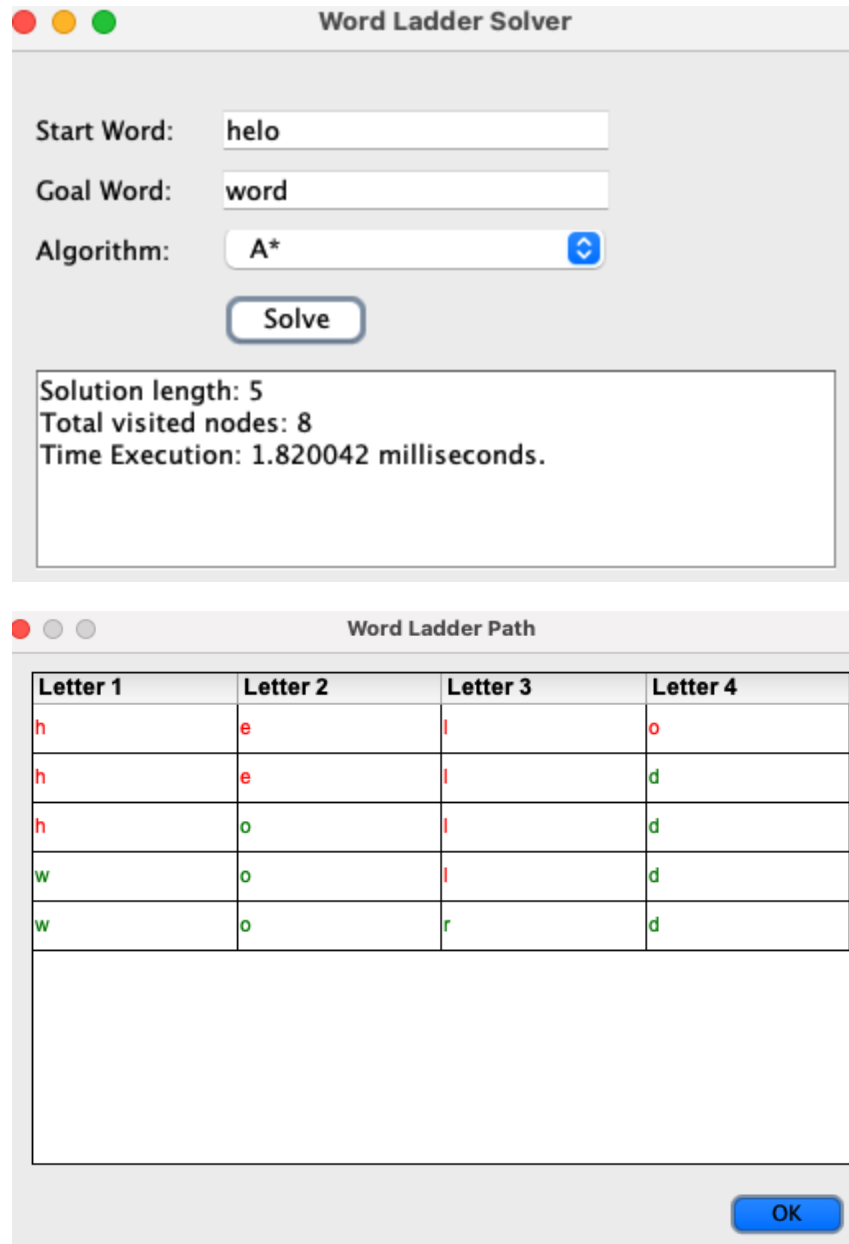
Total visited nodes: 6

Time Execution: 1.27575 milliseconds.

Word Ladder Path

Letter 1	Letter 2	Letter 3	Letter 4
h	e	l	o
h	e	l	d
h	e	r	d
h	a	r	d
w	a	r	d
w	o	r	d

- A*



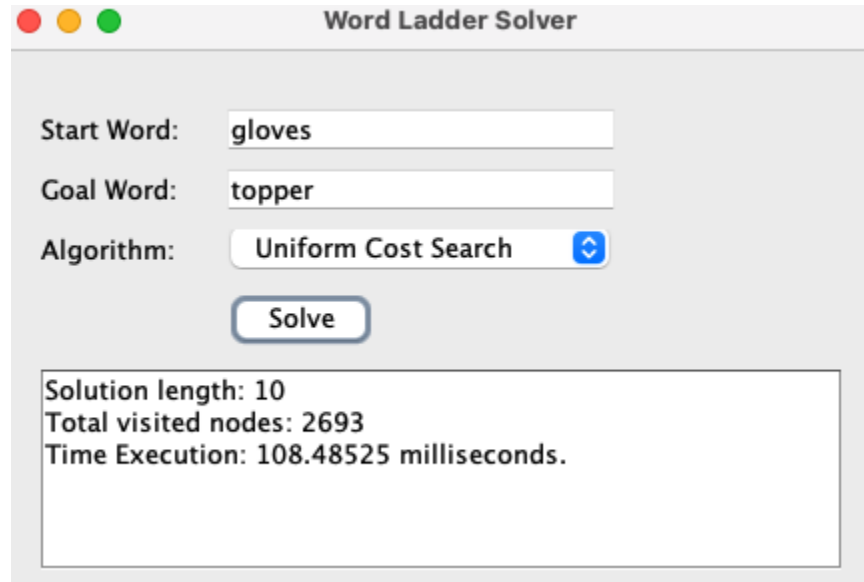
Dari test case tersebut terlihat bahwa algoritma A* dan UCS lebih optimal daripada algoritma GBFS dikarenakan pada kasus ini terdapat jalur node yang terjebak di local optima, dan untuk pembangkitan node yang dilakukan pada UCS cenderung lebih signifikan lebih banyak daripada dengan algoritma A* karena untuk level yang lebih dalam pembangkitan child pada graf yang dihasilkan lebih banyak daripada node yang berada pada level sebelumnya, hal tersebut dapat dilihat dari banyaknya node yang dikunjungi oleh setiap algoritma tersebut. Sehingga diperoleh:

- Dari sisi optimalitas, jelas $UCS = A^* > GBFS$.

- Dari sisi waktu eksekusi $UCS > GBFS > A^*$.
- Dari sisi penggunaan memori $UCS > GBFS > A^*$.

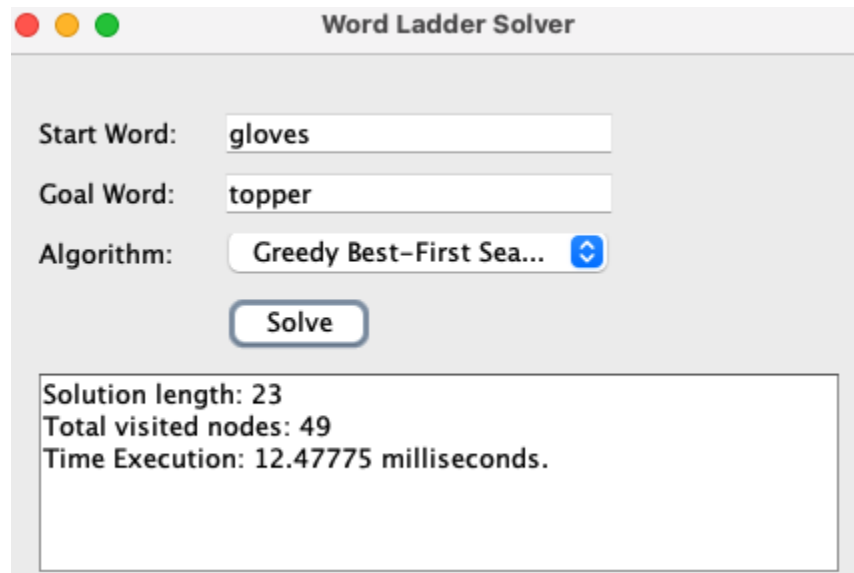
5. Test Case 5

- Uniform Cost Search



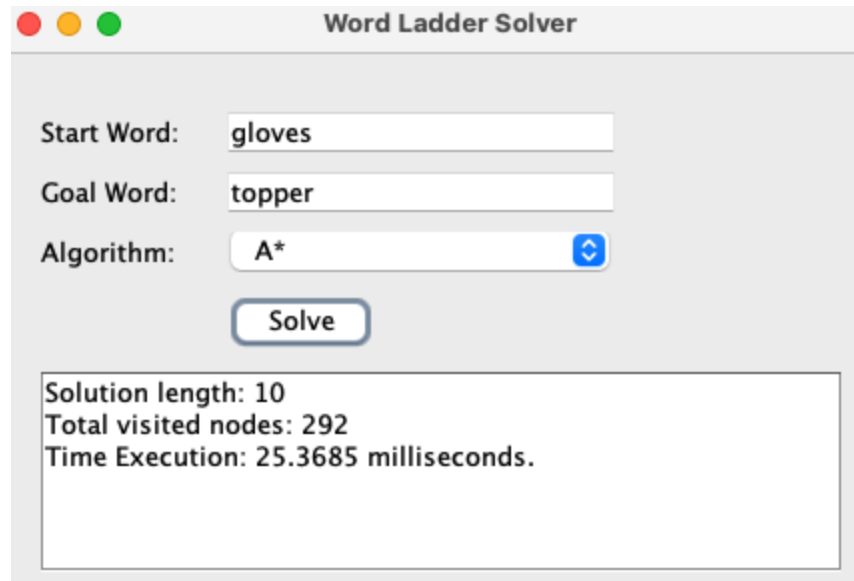
The screenshot shows a window titled "Word Ladder Solver". It contains three input fields: "Start Word:" with the value "gloves", "Goal Word:" with the value "topper", and "Algorithm:" with a dropdown menu set to "Uniform Cost Search". Below these fields is a "Solve" button. At the bottom of the window, a text box displays the results: "Solution length: 10", "Total visited nodes: 2693", and "Time Execution: 108.48525 milliseconds."

- Greedy Best First Search



The screenshot shows a window titled "Word Ladder Solver". It contains three input fields: "Start Word:" with the value "gloves", "Goal Word:" with the value "topper", and "Algorithm:" with a dropdown menu set to "Greedy Best-First Sea...". Below these fields is a "Solve" button. At the bottom of the window, a text box displays the results: "Solution length: 23", "Total visited nodes: 49", and "Time Execution: 12.47775 milliseconds."

- A^*



Dari test case tersebut terlihat bahwa algoritma A* dan UCS lebih optimal daripada algoritma GBFS dikarenakan pada kasus ini terdapat jalur node yang terjebak di local optima, dan untuk pembangkitan node yang dilakukan pada UCS cenderung lebih signifikan lebih banyak daripada dengan algoritma A* karena untuk level yang lebih dalam pembangkitan child pada graf yang dihasilkan lebih banyak daripada node yang berada pada level sebelumnya, hal tersebut dapat dilihat dari banyaknya node yang dikunjungi oleh setiap algoritma tersebut. Sehingga diperoleh:

- Dari sisi optimalitas, jelas $UCS = A^* > GBFS$.
- Dari sisi waktu eksekusi $UCS > GBFS > A^*$.
- Dari sisi penggunaan memori $UCS > GBFS > A^*$.


6. Test Case 6

- Uniform Cost Search

Word Ladder Solver

Start Word:

Goal Word:

Algorithm: 


No Solution Found!
Total visited nodes: 1
Time Execution: 0.253042 milliseconds.

- Greedy Best First Search

Word Ladder Solver

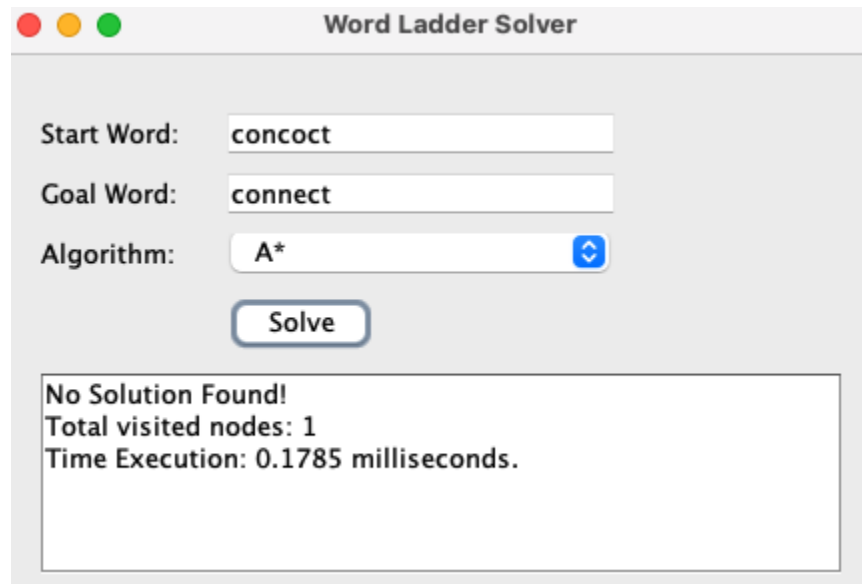
Start Word:

Goal Word:

Algorithm: 

No Solution Found!
Total visited nodes: 2
Time Execution: 0.209 milliseconds.

- A*



Word Ladder Solver

Start Word: concoct

Goal Word: connect

Algorithm: A*

Solve

No Solution Found!
Total visited nodes: 1
Time Execution: 0.1785 milliseconds.

Dari test case tersebut terlihat bahwa ketiganya tidak memiliki solusi karena perubahan kata dari start ke goal tidak memenuhi prinsip heuristik yang admissible terlalu sedikit dan juga dipengaruhi oleh adanya kata yang tidak ada di kamus.

- Dari sisi optimalitas, jelas $UCS = A^* = GBFS$.
- Dari sisi waktu eksekusi $UCS > GBFS > A^*$.
- Dari sisi penggunaan memori $UCS = GBFS = A^*$.

PENJELASAN IMPLEMENTASI BONUS (GUI)

Untuk Implementasi GUI dengan menggunakan Javax sebagai library untuk untuk melakukan kreasi terhadap pembuatan setiap komponen yang ada di GUI desktop yang terdiri dari komponen input form, popup, banner, text area, dan lainnya yang pada awalnya dilakukan inisialisasi pada method initialize() pada class Gui di file Gui.java. Kemudian, dilakukan pengisian terhadap setiap komponen output seperti text area, pop up, dan lainnya setelah masing-masing algoritma diproses selesai dan mendapatkan solusi berupa komponen path, time execution, dan number of node visited. Kemudian akan dilakukan show dengan method show yang dipanggil pada class Main setelah proses algoritma tersebut selesai dieksekusi pada method show. Adapun berikut adalah potongan kode pada class yang mengimplementasikan GUI tersebut.

```
import javax.swing.*;
import javax.swing.table.DefaultTableModel;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class Gui {
    private JFrame frame;
    private JTextField startWordField;
    private JTextField goalWordField;
    private JComboBox<String> algorithmComboBox;
    private JButton solveButton;
    private JTextArea resultArea;
```



```
public Gui() {  
    initialize();  
}  
  
private void initialize() {  
    // Logic  
}  
  
private void solveWordLadder() {  
    // Logic  
}  
  
private void visualizePath(List<String> path) {  
    // Logic  
}  
  
public void show() {  
    frame.setVisible(true);  
}  
}
```

Melalui implementasi GUI tersebutlah, user menjadi lebih interaktif dalam menggunakan aplikasi desktop Word Ladder Solver yang telah penulis buat.

Lampiran

Lampiran 1

Checklist Penilaian:

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

Lampiran 2

Link Repository Github: https://github.com/gilangr301102/Tucil3_13520137