

## **Laporan Tugas Kecil 3**

### **IF2211 Strategi Algoritma**

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



Disusun oleh :

M. Hanief Fatkhan Nashrullah      13522100

**PROGRAM STUDI TEKNIK INFORMATIKA  
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA  
INSTITUT TEKNOLOGI BANDUNG  
2024**

## Daftar Isi

Daftar Isi	2
Deskripsi Persoalan	3
Analisis dan Implementasi Algoritma	4
Implementasi Program	6
Pengujian dan Analisis Pengujian Algoritma	15
Lampiran	28

## Deskripsi Persoalan

*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 persoalan ini, diminta untuk membuat sebuah program yang dapat memberikan solusi dari permainan *word ladder*. Program perlu mengimplementasikan tiga algoritma pencarian, yaitu *Uniform Cost Search*, *Greedy Best-First Search*, dan *A\**. Bahasa yang digunakan untuk memeriksa kebenaran atau validasi dari kata dalam persoalan ini adalah bahasa Inggris.

## Analisis dan Implementasi Algoritma

Algoritma *Uniform Cost Search*, *Greedy Best-First Search*, dan A\* merupakan algoritma yang digunakan untuk melakukan suatu pencarian rute. Ketiga algoritma ini bergantung pada suatu fungsi untuk mengambil keputusan. Algoritma *Uniform Cost Search* bergantung pada fungsi  $f(n)$  dengan nilai  $g(n)$  yang menentukan beban atau harga dari satu *node* menuju *node* lain. Algoritma *Greedy Best-First Search* bergantung pada fungsi *heuristic*  $f(n)$  dengan nilai  $h(n)$  yang memperkirakan jarak langsung dari satu *node* menuju *node* tujuan. Berbeda dengan *Uniform Cost Search* dan *Greedy Best-First Search*, algoritma A\* memiliki fungsi  $f(n)$  yang bergantung pada dua fungsi lain, yaitu  $g(n)$  dari *Uniform Cost Search* dijumlahkan dengan  $h(n)$  dari *Greedy Best-First Search*.

Dalam persoalan permainan *word ladder*,  $g(n)$  didefinisikan sebagai jumlah langkah yang diambil dari *start word* menuju *node* yang akan dituju. Meskipun jarak antara setiap *node* sebenarnya adalah satu karena hanya dapat mengubah satu huruf, menggunakan jumlah langkah yang diambil dari *start word* akan mempermudah implementasi pada code. Pendefinisian  $h(n)$  pada persoalan ini adalah jumlah karakter yang berbeda dari *word* yang sedang dikunjungi dengan *end word*. Pendekatan ini dipilih karena banyaknya karakter yang berbeda menunjukkan seberapa dekat dengan solusi akhir. Jika jumlah karakter yang berbeda bernilai nol, maka *end word* sudah tercapai karena seluruh karakter dari kata adalah sama.

Berikut adalah langkah-langkah implementasi penyelesaian dari setiap algoritma:

### 1. Algoritma *Greedy Best-First Search*

- a. Masukkan *start word* ke *node* yang akan diexpand
- b. Cari seluruh kombinasi huruf (*expand*) dengan perubahan satu huruf yang valid
- c. Hitung jumlah perbedaan huruf dari kombinasi yang valid menuju *end word*
- d. Ambil kata dengan perbedaan huruf paling sedikit dan atur *parent node* menjadi *node* yang diexpand sebelumnya.
- e. Ulangi langkah (a) dengan menggunakan kata yang diambil pada langkah (d)
- f. Langkah diulangi sampai *end word* diexpand seperti langkah (b)

### 2. Algoritma *Uniform Cost Search*

- a. Masukkan *start word* ke *node* yang akan diexpand
- b. Cari seluruh kombinasi huruf (*expand*) dengan perubahan satu huruf yang valid
- c. Setiap *node* akan memiliki cost yang memiliki nilai satu dari *start word* dan bertambah satu untuk setiap langkah yang diambil
- d. Ambil kata manapun yang berada pada level yang sama jika level yang sama masih ada, jika tidak ambil satu level lebih dalam
- e. Ulangi langkah (a) dengan menggunakan kata yang diambil pada langkah (d)
- f. Langkah diulangi sampai *end word* diexpand seperti langkah (b)

### 3. Algoritma A\*

- a. Masukkan *start word* ke node yang akan diexpand
- b. Cari seluruh kombinasi huruf (*expand*) dengan perubahan satu huruf yang valid
- c. Setiap *node* akan memiliki cost yang memiliki nilai satu dari *start word* dan bertambah satu untuk setiap langkah yang diambil ditambah dengan jumlah perbedaan karakter dengan *end word*.
- d. Masukkan setiap node ke dalam priority queue
- e. Ambil kata yang memiliki cost paling rendah
- f. Ulangi langkah (a) dengan menggunakan kata yang diambil pada langkah (e)
- g. Langkah diulangi sampai *end word* diexpand seperti langkah (b)

Heuristik yang digunakan oleh algoritma A\* pada persoalan ini merupakan heuristik yang *admissible*. Misalkan terdapat sebuah *start word* yang memiliki perbedaan jumlah karakter dengan *end word* berjumlah  $m$ . Maka setidaknya terdapat  $m$  langkah untuk menuju *end word*. Jika harga  $g(n)$  untuk antar node adalah  $n$  dengan  $n$  jumlah langkah yang diambil dari *start word*, maka tidak mungkin bahwa jarak sebenarnya akan lebih kecil dibandingkan dengan  $f(n)$ .

Pada kasus ini, *Uniform Cost Search* (UCS) berbeda dengan *Greedy Best-First Search* (GBFS) dalam pembangkitan nodenya. Pada UCS, node yang dibangkitkanurut pada satu level yang sama seperti *Breadth First Search*. Untuk GBFS, node yang dibangkitkan adalah node yang memiliki nilai *heuristic* yang terkecil.

Secara teoritis, algoritma A\* memiliki efisiensi yang lebih baik dibandingkan dengan UCS pada persoalan *word ladder*. Algoritma A\* akan membangkitkan *node* yang lebih sedikit karena algoritma ini akan memprioritaskan *node* yang memiliki perbedaan jumlah karakter yang lebih sedikit, sedangkan algoritma UCS akan memeriksa seluruh *node* yang valid untuk dibangkitkan. Dengan demikian, *node* yang diperiksa akan lebih sedikit.

Untuk solusi yang dihasilkan oleh GBFS tidak menjamin solusi yang optimal. Hal ini dikarenakan perilaku dari algoritma pencarian ini sangat mirip dengan greedy. GBFS akan berhenti ketika *node* telah ditemukan sedangkan terdapat kemungkinan *node* tujuan masih dapat dicapai dari jalur lain yang lebih dekat.

# Implementasi Program

## 1. Class WordNode

```
1 public class WordNode{
2
3     // Attribute
4     private String word;
5     private Integer weight;
6     private Integer charDiff;
7     private WordNode parent;
8
9     // Constructor
10    public WordNode(String word, Integer weight, Integer charDiff, WordNode parent){
11        this.word = word;
12        this.weight = weight;
13        this.charDiff = charDiff;
14        this.parent = parent;
15    }
16
17    public WordNode(String word, Integer weight, Integer charDiff){
18        this(word, weight, charDiff, null);
19    }
20
21    // Setter
22    public void setWord(String word) {
23        this.word = word;
24    }
25
26    public void setWeight(Integer weight) {
27        this.weight = weight;
28    }
29
30    public void setcharDiff(Integer charDiff) {
31        this.charDiff = charDiff;
32    }
33
34    public void setParent(WordNode parent) {
35        this.parent = parent;
36    }
37
38    //Getter
39    public String getWord() {
40        return word;
41    }
42
43    public Integer getWeight() {
44        return weight;
45    }
46
47    public Integer getcharDiff() {
48        return charDiff;
49    }
50}
```

```
51     public WordNode getParent() {
52         return parent;
53     }
54
55     public Integer getTotalCost(){
56         return weight+charDiff;
57     }
58
59     //Operations
60     public Integer getDifference(String target){
61         if(this.word.length()!=target.length()){
62             return -1;
63         }
64         else{
65             Integer diff = 0;
66             for(int i=0;i<this.word.length();i++){
67                 if(this.word.charAt(i)!=target.charAt(i)){
68                     diff++;
69                 }
70             }
71             return diff;
72         }
73     }
74
75     public Integer getDifference(WordNode target){
76         return this.getDifference(target.getWord());
77     }
78
79     public Integer getWordLength(){
80         return this.word.length();
81     }
82
83     public ArrayList<String> generatePossibleMove(){
84         StringBuilder editor;
85         ArrayList<String> result = new ArrayList<>();
86         for(int i=0;i<this.getWordLength();i++){
87             for(int j=97;j<123;j++){
88                 editor = new StringBuilder(this.getWord());
89                 editor.setCharAt(i, (char)j);
90                 if(!editor.toString().equals(this.word)){
91                     result.add(editor.toString());
92                 }
93             }
94         }
95         return result;
96     }
97 }
98 }
```

Class `WordNode` merupakan sebuah representasi dari *node* untuk setiap kata. Class ini memiliki atribut `word` (`String`) sebagai kata yang disimpan, `weight` (`Integer`) sebagai harga untuk berpindah dari satu *node* ke *node* lain, `charDiff` (`Integer`) sebagai nilai *heuristik* yang merepresentasikan jumlah karakter yang berbeda dari *node* tersebut dengan *node* tujuan dan `parent` (`WordNode`) sebagai representasi *parent node* dari suatu *node*. Method yang dimiliki oleh class ini kebanyakan adalah getter dan setter. Method yang unik dari class ini adalah `getDifference` dan `generatePossibleMove`. `getDifference` menghitung banyak karakter yang berbeda untuk nilai *heuristic*. `generatePossibleMove` untuk membuat seluruh kombinasi kata yang dapat dibuat (belum divalidasi oleh kamus bahasa Inggris).

## 2. Class `BaseSolver`

```
1 public abstract class BaseSolver{
2     protected PriorityQueue<WordNode> openQueue;
3     protected WordNode targetNode;
4     protected HashSet<String> visitedNode;
5     protected ArrayList<String> pathSolution;
6     protected long runtime;
7
8     public BaseSolver(String startNode, String targetNode){
9         this.targetNode = new WordNode(targetNode, null, null);
10        this.visitedNode = new HashSet<>();
11    }
12
13    public WordNode getTargetNode() {
14        return targetNode;
15    }
16
17    public ArrayList<String> getPathSolution(){
18        return this.pathSolution;
19    }
20    abstract public void solve(GameDictionary gameDictionary);
21    abstract public Integer getVisitCount();
22
23
24 }
```

Class `BaseSolver` merupakan class abstrak yang akan memiliki spesialisasi di setiap algoritmanya. Class ini memiliki atribut `openQueue` (`PriorityQueue`) untuk menampung *node* yang akan diproses, `targetNode` untuk menyimpan *node* tujuan akhir, `visitedNode` untuk menyimpan kata yang sudah dikunjungi, `pathSolution` untuk menyimpan jalur solusi, dan `runtime` untuk menyimpan lama pencarian solusi. Class ini



memiliki konstruktor yang menginisialisasi `targetNode` dan `visitedNode`. Method selain *setter* dan *getter* dari class ini adalah `solve`. Method ini akan menjalankan algoritma pencarian solusi sesuai dengan class yang telah terspesialisasi.

### 3. Class GreedyBFS

```
1 public class GreedyBFS extends BaseSolver {
2     private HashSet<String> multipleVisit;
3
4     public Integer getVisitCount(){
5         return this.visitedNode.size()+this.multipleVisit.size();
6     }
7
8     public GreedyBFS(String startNode, String targetNode) {
9         super(startNode, targetNode);
10        this.openQueue = new PriorityQueue<>(new GBFSComparator());
11        this.openQueue.add(new WordNode(startNode, 0, this.targetNode.getDifference(startNode)));
12        this.multipleVisit = new HashSet<>();
13    }
14
15    public void solve(GameDictionary gameDictionary){
16        long startTime = System.currentTimeMillis();
17        ArrayList<String> possibleMove;
18        WordNode currentNode = openQueue.peek();
19        if(this.getTargetNode().getWord().length()==currentNode.getWord().length() &&
20        gameDictionary.getData().contains(currentNode.getWord()) &&
21        gameDictionary.getData().contains(this.getTargetNode().getWord())){
22            while(!this.openQueue.isEmpty() && !currentNode.getWord().equals(this.targetNode.getWord())){
23                currentNode = this.openQueue.poll();
24                possibleMove = currentNode.generatePossibleMove();
25                for(String item : possibleMove){
26                    if(gameDictionary.getData().contains(item)){
27                        openQueue.add(new WordNode(item, currentNode.getWeight()+1,
28                        targetNode.getDifference(item), currentNode));
29                    }
30                }
31                System.out.println("Expanded " + currentNode.getWord());
32                if(this.multipleVisit.contains(currentNode.getWord())){
33                    WordNode noSolution = new WordNode("No Solution", null, null);
34                    this.targetNode = noSolution;
35                    pathSolution = new ArrayList<>();
36                    pathSolution.add(0, noSolution.getWord());
37                    break;
38                }
39            }
40        }
41    }
42 }
```

```

36         if(this.visitedNode.contains(currentNode.getWord())){
37             multipleVisit.add(currentNode.getWord());
38         }
39         else{
40             this.visitedNode.add(currentNode.getWord());
41         }
42     }
43     if(currentNode.getWord().equals(this.targetNode.getWord())){
44         this.targetNode = currentNode;
45         pathSolution = new ArrayList<>();
46         while (currentNode!=null) {
47             pathSolution.add(0,currentNode.getWord());
48             currentNode = currentNode.getParent();
49         }
50     }
51     else{
52         WordNode noSolution = new WordNode("No Solution", null, null);
53         this.targetNode = noSolution;
54         pathSolution = new ArrayList<>();
55         pathSolution.add(0, noSolution.getWord());
56     }
57 }
58 else{
59     WordNode noSolution = new WordNode("No Solution", null, null);
60     this.targetNode = noSolution;
61     pathSolution = new ArrayList<>();
62     pathSolution.add(0, noSolution.getWord());
63 }
64 long endTime = System.currentTimeMillis();
65 this.runtime = endTime-startTime;
66 }
67 }

```

Class GreedyBFS merupakan class turunan dari class abstrak BaseSolver. Class ini mengimplementasikan abstract method dari class BaseSolver. Untuk konstruktor dari kelas ini menggunakan konstruktor dari class super dengan perbedaan pada comparator untuk PriorityQueue. Method solver pada GreedyBFS membutuhkan satu atribut tambahan yaitu multipleVisit. Atribut ini sebenarnya sama seperti atribut visitedNode sehingga atribut tambahan ini digunakan untuk pengecekan apakah suatu node sudah dikunjungi lebih dari dua kali. Ketika sudah dikunjungi lebih dari dua kali, solver akan memasukkan “No Solution” ke atribut pathSolution untuk mencegah terjadinya *infinite loop* pada solver.

#### 4. Class GBFSComparator

```
1 class GBFSComparator implements Comparator<WordNode>{
2     public int compare(WordNode n1, WordNode n2) {
3         if(n1.getcharDiff(>)n2.getcharDiff()){
4             return 1;
5         }
6         else if(n1.getcharDiff(<)n2.getcharDiff()){
7             return -1;
8         }
9         return 0;
10    }
11 }
```

Class GBFSComparator merupakan comparator untuk menentukan urutan dalam PriorityQueue.

#### 5. Class UniformCostSearch

```
1 public class UniformCostSearch extends BaseSolver {
2
3     public Integer getVisitCount(){
4         return this.visitedNode.size();
5     }
6
7     public UniformCostSearch(String startNode, String targetNode) {
8         super(startNode, targetNode);
9         this.openQueue = new PriorityQueue<>(new UCSCComparator());
10        this.openQueue.add(new WordNode(startNode, 0,this.targetNode.getDiference(startNode)));
11    }
12
13    public void solve(GameDictionary gameDictionary){
14        long startTime = System.currentTimeMillis();
15        ArrayList<String> possibleMove;
16        WordNode currentNode = openQueue.peek();
17        if(this.getTargetNode().getWord().length()==currentNode.getWord().length() &&
18        gameDictionary.getData().contains(currentNode.getWord()) &&
19        gameDictionary.getData().contains(this.getTargetNode().getWord())){
20            while(!this.openQueue.isEmpty() && !currentNode.getWord().equals(this.targetNode.getWord())){
21                currentNode = this.openQueue.poll();
22                possibleMove = currentNode.generatePossibleMove();
23                for(String item : possibleMove){
24                    WordNode inserter = new WordNode(item,currentNode.getWeight()+1,
25                    targetNode.getDiference(item),currentNode);
26                    if(gameDictionary.getData().contains(item) && !visitedNode.contains(inserter.getWord())){
27                        openQueue.add(inserter);
28                    }
29                }
30            }
31        }
32    }
33 }
```

```

27         System.out.println("Expanded "+currentNode.getWord());
28         this.visitedNode.add(currentNode.getWord());
29     }
30
31     if(currentNode.getWord().equals(this.targetNode.getWord())){
32         this.targetNode = currentNode;
33         pathSolution = new ArrayList<>();
34         while (currentNode!=null) {
35             pathSolution.add(0,currentNode.getWord());
36             currentNode = currentNode.getParent();
37         }
38     }
39     else{
40         WordNode noSolution = new WordNode("No Solution", null, null);
41         this.targetNode = noSolution;
42         pathSolution = new ArrayList<>();
43         pathSolution.add(0, noSolution.getWord());
44     }
45 }
46 else{
47     WordNode noSolution = new WordNode("No Solution", null, null);
48     this.targetNode = noSolution;
49     pathSolution = new ArrayList<>();
50     pathSolution.add(0, noSolution.getWord());
51 }
52 long endTime = System.currentTimeMillis();
53 this.runtime = endTime-startTime;
54 }
55 }

```

Class UniformCostSearch merupakan class turunan dari class abstrak BaseSolver. Class ini mengimplementasikan abstract method dari class BaseSolver. Untuk konstruktor dari kelas ini menggunakan konstruktor dari class super dengan perbedaan pada comparator untuk PriorityQueue.

## 6. Class UCSComparator

```
1 class UCSComparator implements Comparator<WordNode>{
2     public int compare(WordNode n1, WordNode n2) {
3         if(n1.getWeight()>n2.getWeight()){
4             return 1;
5         }
6         else if(n1.getWeight()<n2.getWeight()){
7             return -1;
8         }
9         return 0;
10    }
11 }
```

Class UCSComparator merupakan comparator untuk menentukan urutan dalam PriorityQueue.

## 7. Class AStar

```
1 public class AStar extends UniformCostSearch {
2     public AStar(String startNode, String targetNode) {
3         super(startNode, targetNode);
4         this.openQueue = new PriorityQueue<>(new AStarComparator());
5         this.openQueue.add(new WordNode(startNode, 0,this.targetNode.getDifference(startNode)));
6     }
7 }
```

Class AStar merupakan class turunan dari UniformCostSearch. Secara algoritma, kelas AStar memiliki algoritma yang sama dan hanya memiliki perbedaan pada comparatornya saja.

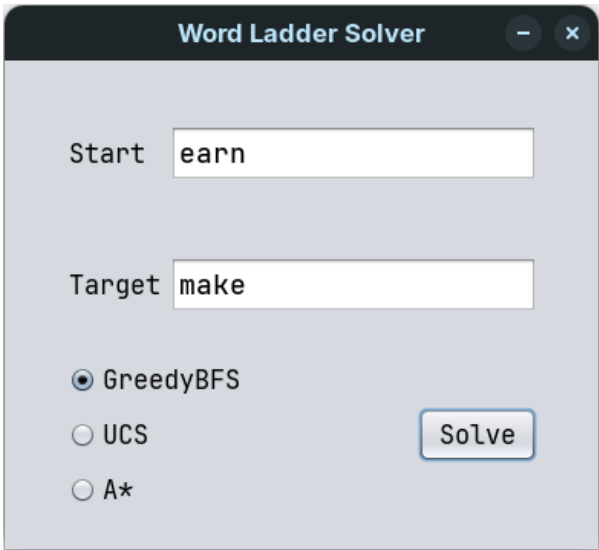
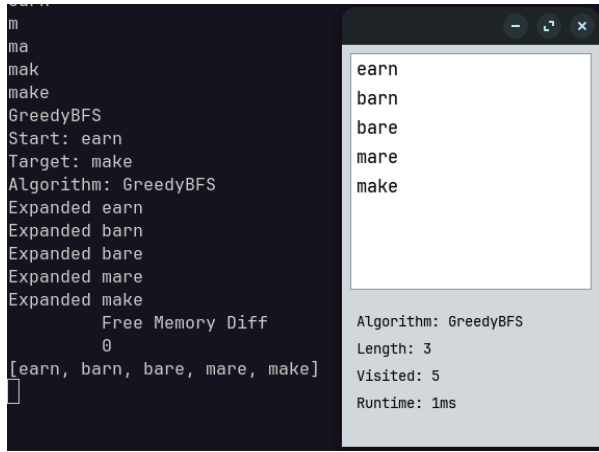
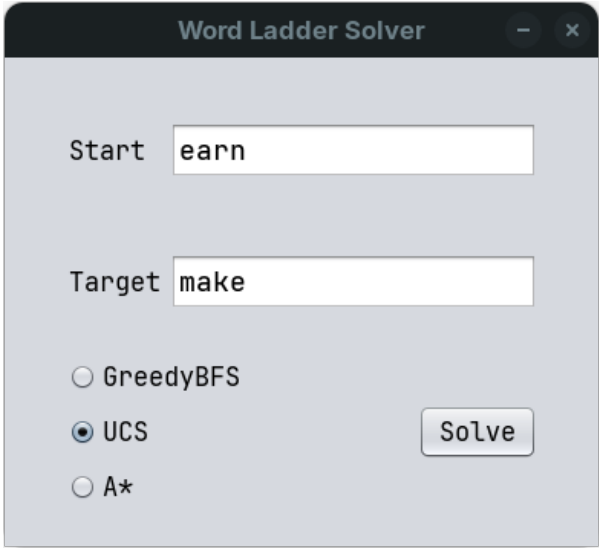
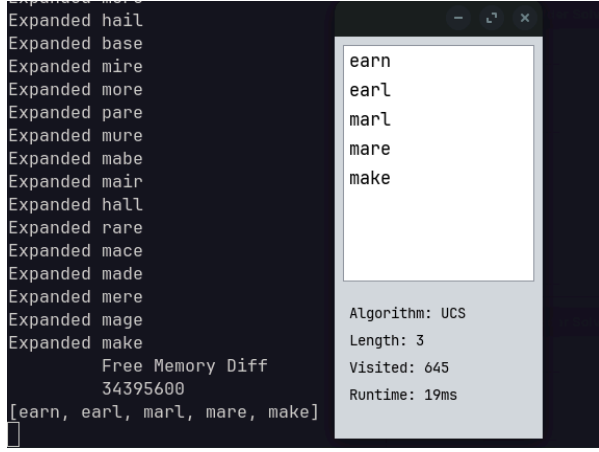
## 8. Class AStarComparator

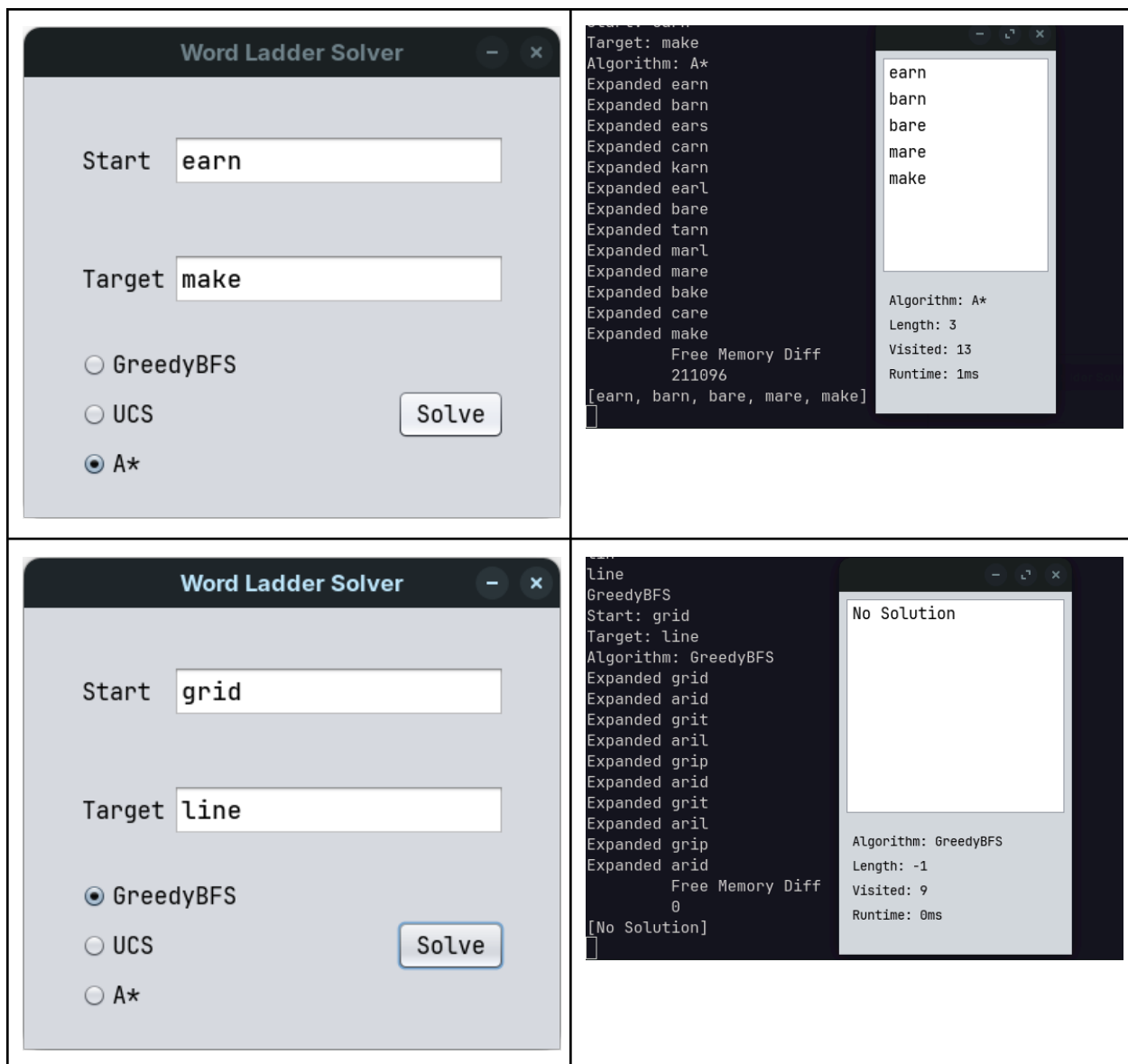
```
1 class AStarComparator implements Comparator<WordNode>{
2     public int compare(WordNode n1, WordNode n2) {
3         if(n1.getTotalCost(>)>n2.getTotalCost()){
4             return 1;
5         }
6         else if(n1.getTotalCost(<)<n2.getTotalCost()){
7             return -1;
8         }
9         return 0;
10    }
11 }
```

Class AStarComparator merupakan comparator untuk menentukan urutan dalam PriorityQueue.

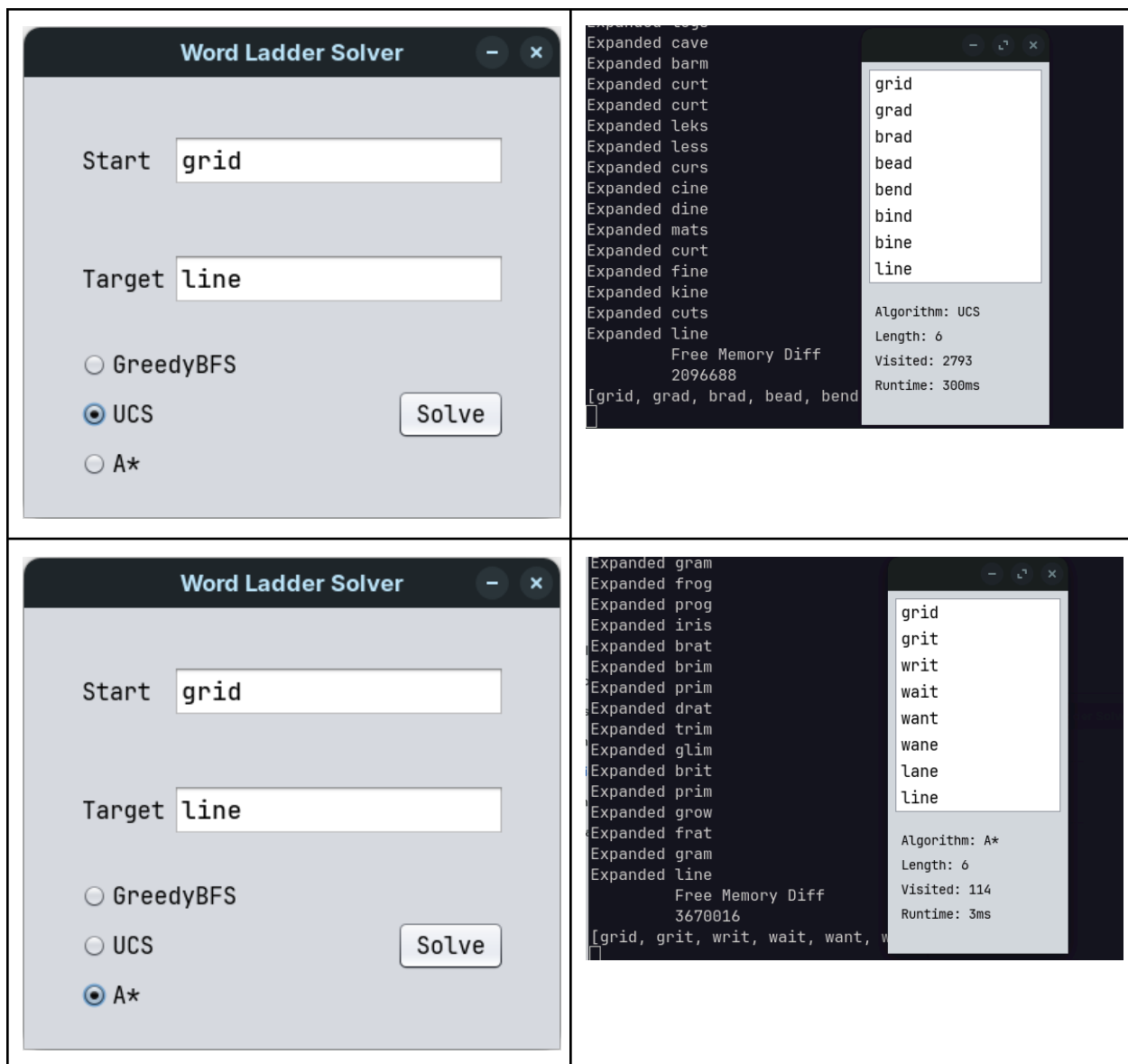
## Pengujian dan Analisis Pengujian Algoritma

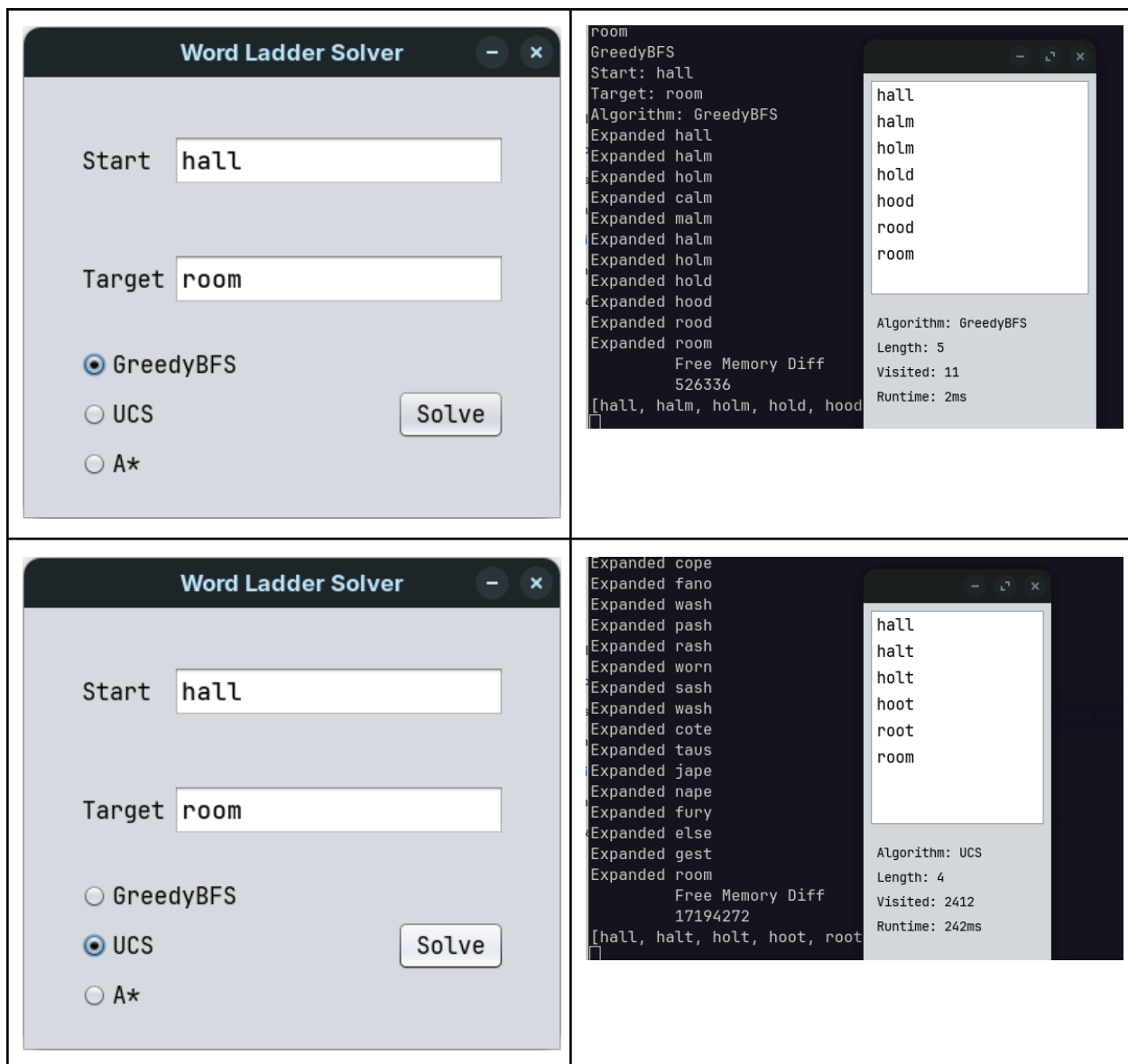
Berikut adalah hasil pengujian algoritma dari program yang telah dibuat

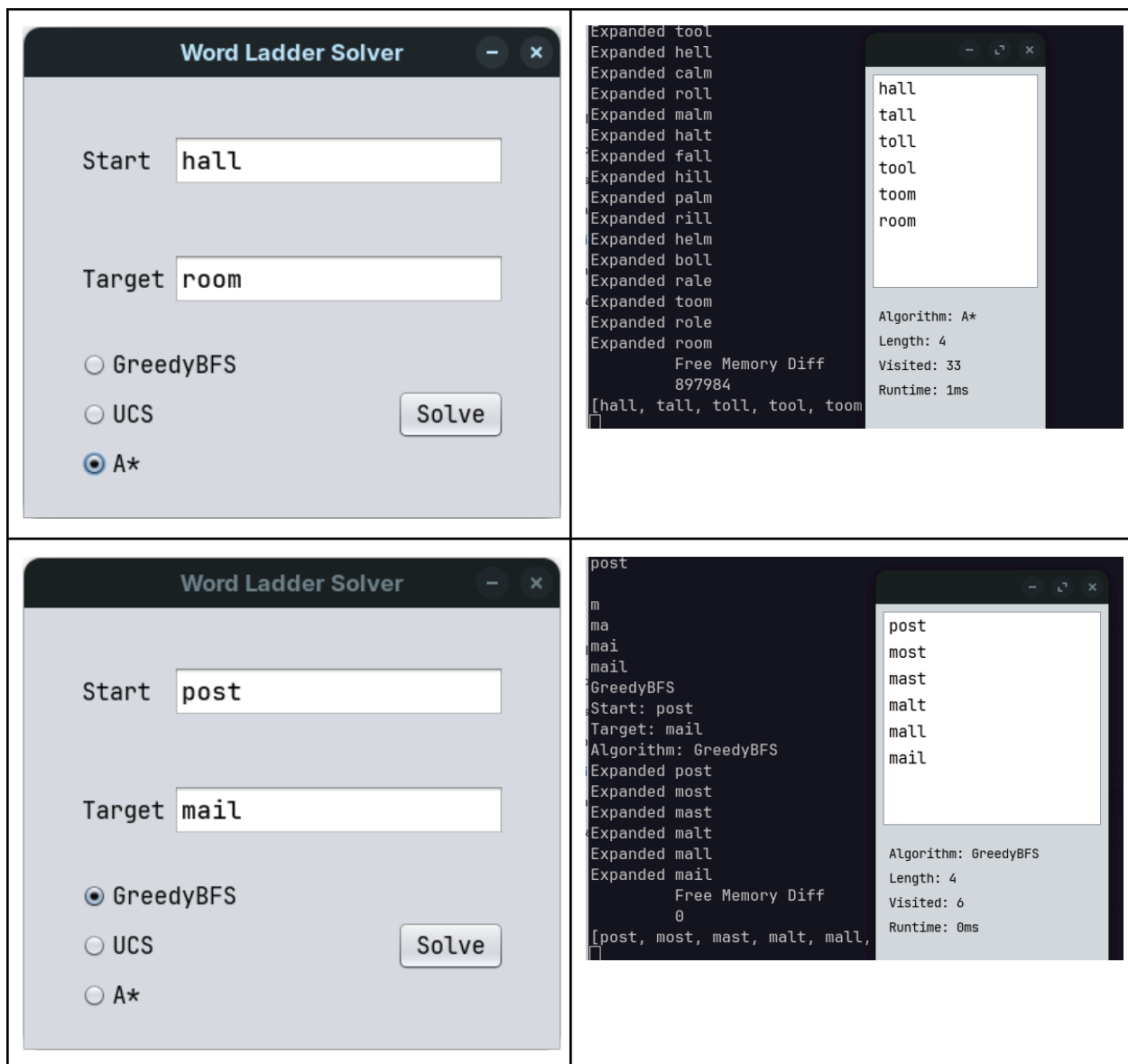
Input	Output
	
	

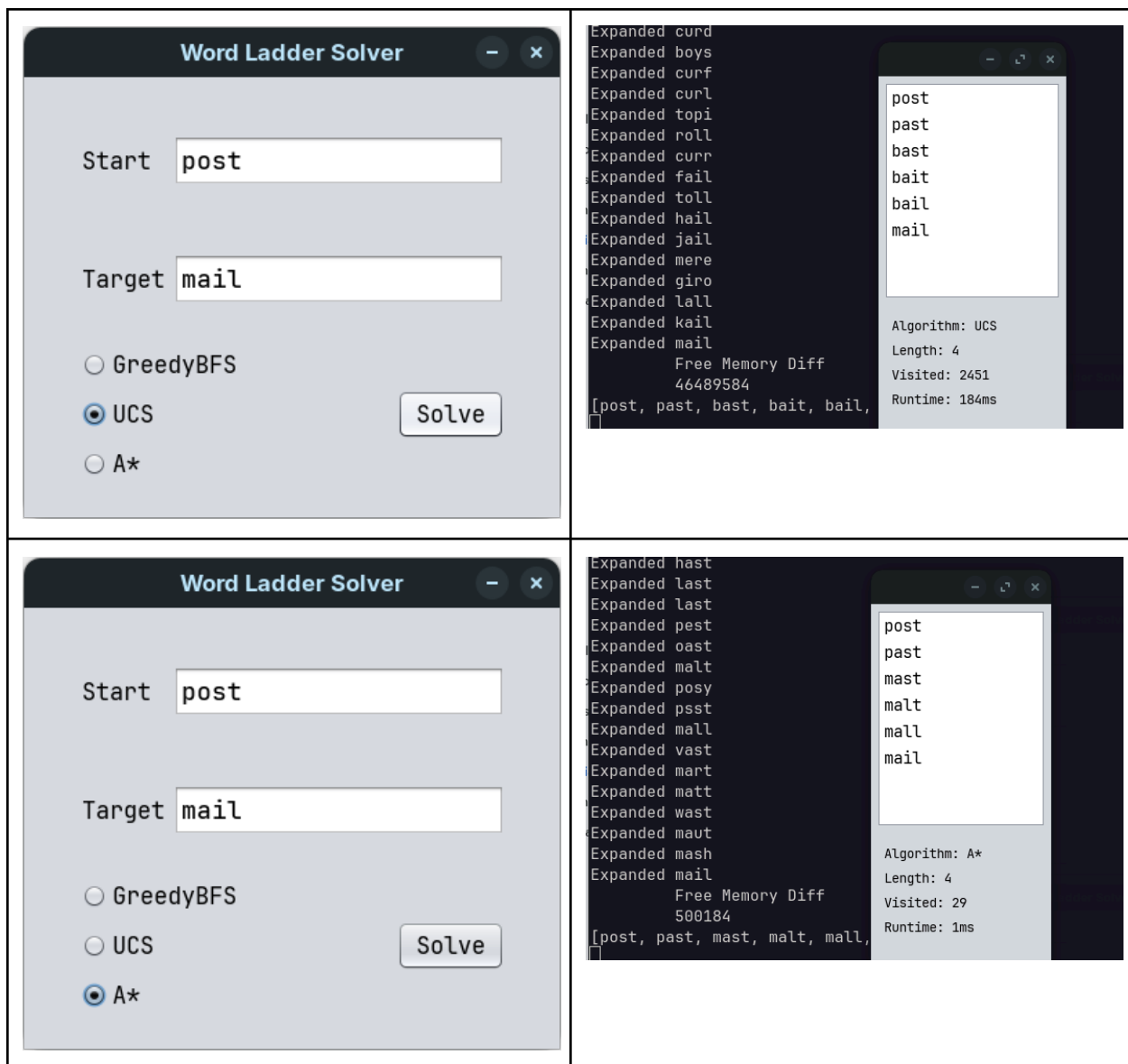


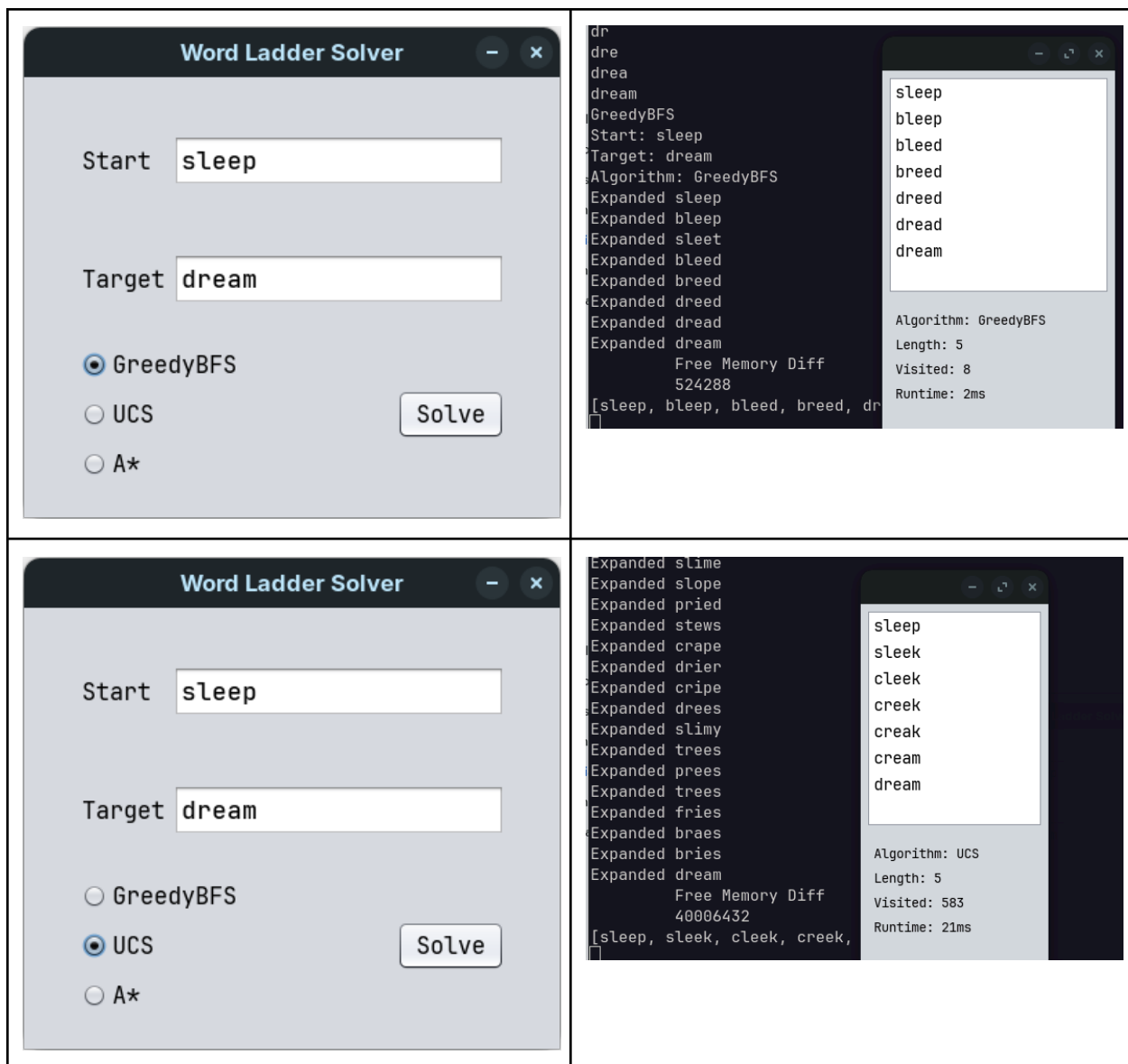












Word Ladder Solver

Start

sleep

Target

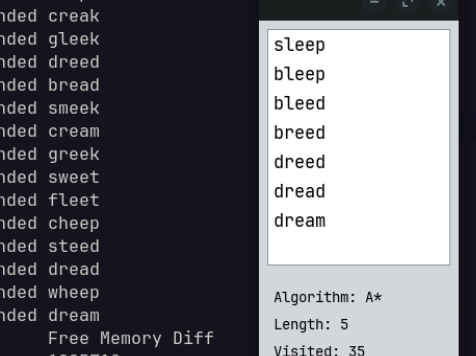
dream

☐ GreedyBFS

☐ UCS

☒ A\*

Solve



The screenshot shows a terminal window with a word ladder. The words are listed vertically, with the first letter of each word highlighted in yellow. A white box on the right side of the terminal contains the words from 'sleep' to 'dream'. Below the word ladder, the text 'Free Memory Diff' and '1295712' are displayed. At the bottom, a prompt character '[' is followed by the words 'sleep, bleep, bleed, breed, dream'.

```
Expanded breed
Expanded slept
Expanded creak
Expanded gleeek
Expanded dreed
Expanded bread
Expanded smeek
Expanded cream
Expanded greek
Expanded sweet
Expanded fleet
Expanded cheep
Expanded steed
Expanded dread
Expanded wheep
Expanded dream
Expanded dream

Free Memory Diff
1295712

[sleep, bleep, bleed, breed, dream]
```

sleep  
leep  
bleed  
breed  
dreed  
dread  
dream

Algorithm: A\*  
Length: 5  
Visited: 35  
Runtime: 1ms

Word Ladder Solver

Start

paint

Target

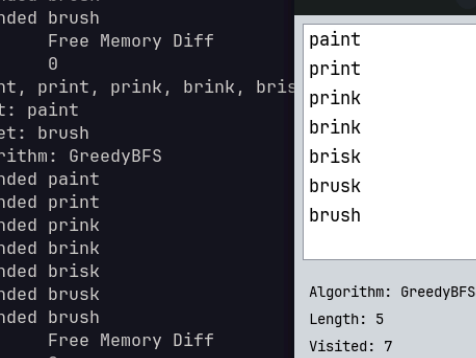
brush

☒ GreedyBFS

☐ UCS

☐ A\*

Solve



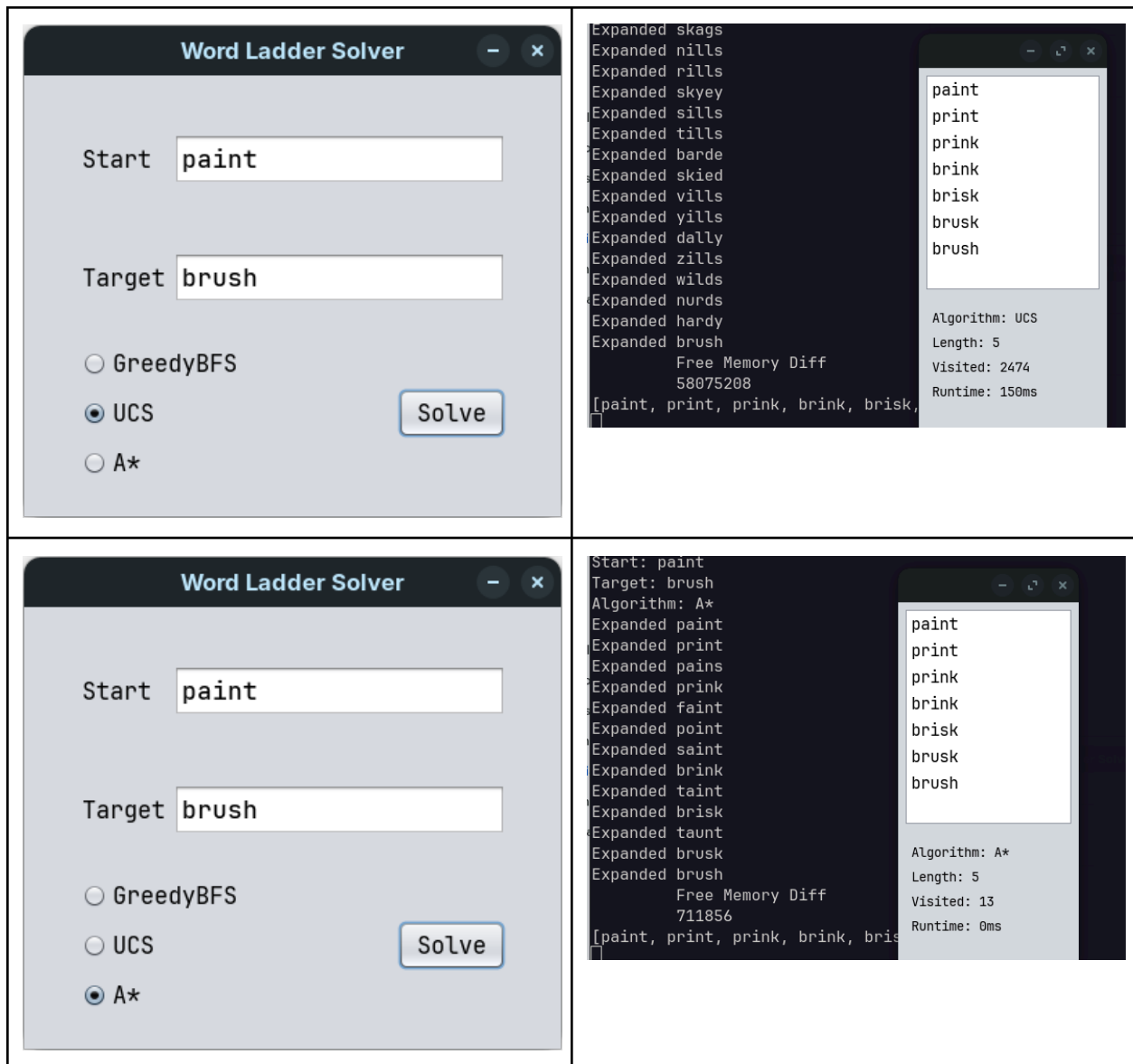
```

Expanded brusk
Expanded brusk
Expanded brush
    Free Memory Diff
    0
[paint, print, print, brink, brisk, brusk, brush]
Start: paint
Target: brush
Algorithm: GreedyBFS
Expanded paint
Expanded print
Expanded print
Expanded brink
Expanded brink
Expanded brink
Expanded brusk
Expanded brusk
Expanded brusk
Expanded brush
    Free Memory Diff
    0
[paint, print, print, brink, brisk, brusk, brush]

```

paint  
print  
prink  
brink  
brisk  
brusk  
brush

Algorithm: GreedyBFS  
Length: 5  
Visited: 7  
Runtime: 1ms



Terdapat tiga aspek yang dapat diamati pada hasil percobaan tersebut, yaitu penggunaan *memory*, *runtime*, dan optimalitas dari solusi yang dihasilkan. Pada penggunaan *memory*, algoritma UCS konsisten memiliki penggunaan *memory* yang paling besar. Hal ini disebabkan karena banyaknya *node* yang dikunjungi melebihi kedua algoritma lainnya. Begitu juga dengan *runtime*, UCS memiliki *runtime* paling lama di antara ketiga algoritma. Berkebalikan dengan UCS, algoritma GBFS konsisten memiliki penggunaan *memory* yang sangat rendah dan memiliki *runtime* yang sangat singkat. Meskipun demikian, hasil yang diberikan tidak selalu optimal. Terdapat kasus di mana algoritma ini mengalami *circular visit* seperti pada pengujian ke-4 sehingga program harus berhenti untuk mencegah *resource* dari komputer digunakan terlalu banyak. Selain itu, terdapat juga kasus seperti pada kata *hall* menuju *room* di mana hasil yang diberikan oleh algoritma GBFS lebih panjang satu langkah dibanding kedua algoritma lainnya. Algoritma A\* selalu konsisten dengan hasil yang optimal, penggunaan *memory* rendah dan

*runtime* yang singkat. Hal ini disebabkan pencarian tidak dilakukan secara menyeluruh seperti UCS, tetapi memeriksa dengan lebih lengkap dan memeriksa alternatif lain jika dibandingkan dengan GBFS.



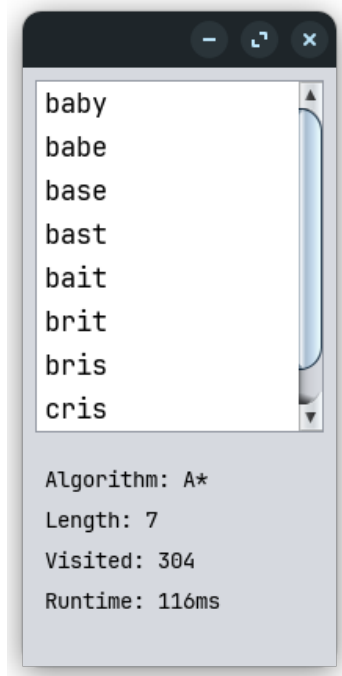
## Implementasi GUI

Pada GUI dari program, terdapat empat bagian utama, yaitu *start word textfield*, *target word textfield*, *algorithm radio button*, dan *solve button*. Sesuai dengan namanya, bagian *textfield* merupakan tempat untuk menentukan kata awal dan kata tujuan dari permainan *word ladder*. *Algorithm radio button* digunakan untuk memilih algoritma yang akan digunakan dalam menyelesaikan *word ladder*. Terakhir, terdapat *solve button* yang digunakan untuk memicu agar penyelesaian dapat dilakukan.



*GUI dari Word Ladder Solver yang telah dibuat*

Setelah *solve button* ditekan, maka akan muncul *pop-up* baru yang menampilkan solusi dari persoalan *word ladder* sesuai dengan kata yang dipilih. Tidak semua persoalan memiliki solusi. Terdapat kasus di mana persoalan tidak memiliki solusi, seperti ketika GBFS masuk dalam circular loop, panjang dari kata berbeda, ataupun memang tidak ada kombinasi karakter yang membentuk kata untuk menghubungkan kata awal dengan kata tujuan. Berikut adalah tampilan *pop-up* dari program ketika menampilkan solusi dan ketika tidak memiliki solusi.

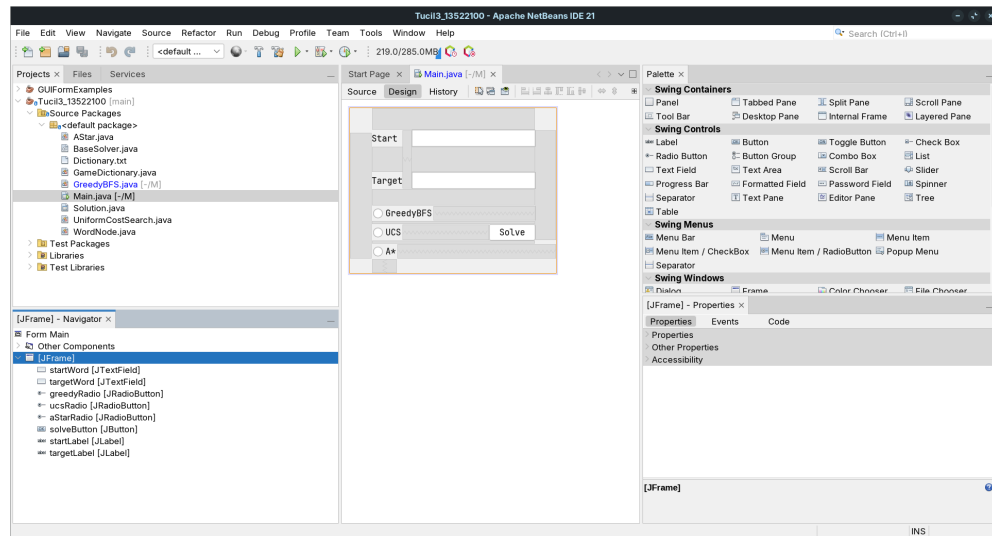


*Pop-up ketika program berhasil menemukan solusi*



*Pop-up ketika program gagal menemukan solusi*

GUI dari program ini diimplementasikan dengan menggunakan Apache Netbeans. Apache Netbeans menyediakan editor untuk mempermudah dalam pembuatan GUI. Ant juga digunakan dalam program ini untuk mempermudah proses build dan kompilasi dari program.



*Tampilan GUI editor*

## **Lampiran**

Link GitHub Repository : [https://github.com/hannoobz/Tucil3\\_13522100](https://github.com/hannoobz/Tucil3_13522100)