

# **LAPORAN TUGAS KECIL III**

## **IF2211 STRATEGI ALGORITMA**

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



Disusun oleh:

Ignatius Jhon Hezkiel Chan 13522029

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

**2024**

# Daftar Isi

<b>PENDAHULUAN.....</b>	<b>3</b>
<b>BAB I DASAR TEORI.....</b>	<b>4</b>
A. Uniform Cost Search.....	4
B. Greedy Best First Search (GBFS).....	5
C. A* (A Star) Search.....	5
<b>BAB II ANALISIS DAN IMPLEMENTASI.....</b>	<b>7</b>
A. Uniform Cost Search (UCS).....	7
B. Greedy Best First Search (GBFS).....	9
C. A* Algorithm.....	11
<b>BAB III SOURCE CODE.....</b>	<b>13</b>
1. Class Node.....	13
2. Class Graph.....	14
3. Class UCSGraph.....	15
4. Class GBFSGraph.....	16
5. Class AStarGraph.....	17
<b>BAB IV HASIL PENGUJIAN.....</b>	<b>18</b>
A. Uji Kasus 1 ( BASE -> ROOT).....	18
B. Uji Kasus 2 (CORK -> PLUG).....	20
C. Uji Kasus 3 (PARTY -> TRICK).....	22
D. Uji Kasus 4 (NOMAD -> STAFF).....	24
E. Uji Kasus 5 (STRONG -> MIGHTY).....	26
F. Uji Kasus 6 (WINTER -> SUMMER).....	28
<b>BAB V ANALISIS HASIL PENGUJIAN.....</b>	<b>30</b>
<b>BAB VI PENJELASAN BONUS.....</b>	<b>31</b>
<b>LAMPIRAN.....</b>	<b>32</b>

## PENDAHULUAN

Permainan Word Ladder adalah permainan kata di mana pemain harus mengubah satu kata menjadi kata lain, satu huruf pada satu waktu, dengan setiap perubahan menghasilkan kata yang sah dalam bahasa yang digunakan. Permainan ini biasanya dimainkan di atas kertas, dan setiap langkah harus menghasilkan kata yang valid yang diakui dalam kamus.

### **Cara Bermain Word Ladder:**

1. Pemain memulai dengan sebuah kata awal.
2. Pemain kemudian mengubah satu huruf pada satu waktu untuk membentuk kata baru.
3. Setiap kata peralihan harus merupakan kata yang valid.
4. Tujuannya adalah untuk mencapai kata akhir dalam jumlah langkah sesedikit mungkin.

**Contoh:** Mengubah "HEAD" menjadi "TAIL":

1. HEAD
2. HEAL (mengganti 'D' dengan 'L')
3. TEAL (mengganti 'H' dengan 'T')
4. TELL (mengganti 'A' dengan 'L')
5. TALL (mengganti 'E' dengan 'A')
6. TAIL (mengganti 'L' dengan 'I')

Permainan ini tidak hanya menyenangkan tetapi juga membantu meningkatkan kosakata dan keterampilan pemecahan masalah. Permainan Word Ladder diciptakan oleh Lewis Carroll, penulis "Alice's Adventures in Wonderland," pada tahun 1878.

# BAB I

## DASAR TEORI

### A. Uniform Cost Search

Uniform Cost Search (UCS) merupakan salah satu bentuk Uninformed Search (Blind Search) yang digunakan pada permasalahan pencarian, terutama dalam konteks *graph traversal* dan *pathfinding*. Jika BFS dan IDS menentukan least-cost path berdasarkan jumlah stepnya ( $\text{cost} = \text{step}$ ), UCS menentukan least-cost path ketika  $\text{cost} \neq \text{step}$ . Algoritma ini digunakan untuk mencari least-cost path dari node asal ke node tujuan dalam *weighted graph* dimana *weight*-nya merepresentasikan cost untuk berpindah dari satu node ke lainnya.

Beberapa karakteristik utama dari UCS adalah sebagai berikut.

1. Cost-Based Expansions  
UCS meng-expand nodes berdasarkan jumlah kumulatif cost dari node awal ke node sekarang. Node yang di-expand selalu node dengan jumlah kumulatif cost paling kecil.
2. Priority Queue  
UCS menggunakan priority queue dalam implementasinya, untuk menentukan node mana yang selanjutnya akan di-visit dan di-expand. Priority queue disini mengurutkan node dari yang paling kecil jumlah costnya.
3. Optimality dan Completeness  
UCS memastikan solusi optimal dan komplit. UCS selalu menghasilkan solusi jika ada, dan solusi tersebut dipastikan global optimal karena penelusuran graph / route dilakukan secara menyeluruh.

Berikut langkah-langkah dari algoritma Uniform Cost Search (UCS).

1. Mulai dengan node awal  
Node awal diinisialisasi dengan cost 0. Masukkan node awal pada prio-queue
2. Selagi prio-queue belum kosong, dequeue prio-queue sehingga mendapatkan node dengan cost terendah saat itu.
3. Jika node yang didapatkan sudah pernah dikunjungi, perbarui minimum cost path node tersebut jika hasil dequeue memiliki cost yang lebih kecil. Jika minimum cost path lebih kecil, abaikan node dan lakukan dequeue kembali.
4. Jika node tersebut adalah node tujuan, didapatkan solusi optimal kembalikan route yang didapatkan.
5. Jika bukan, lanjutkan ekspansi dengan memeriksa tetangga-tetangga dari node tersebut.
6. Tambahkan tiap tetangga yang belum dikunjungi pada priority queue dengan cost-nya merupakan cost node sekarang ditambah cost edge antara node sekarang dengan tetangganya.
7. Jika prio-queue sudah kosong dan node tujuan belum dikunjungi, berarti solusi jalur tidak ada.

## B. Greedy Best First Search (GBFS)

Greedy Best First Search (GBFS) merupakan salah satu bentuk Informed Search, yang melakukan penelusuran route / graph dengan menggunakan heuristik tertentu dengan harapan menentukan solusi optimal. Tidak seperti UCS yang berfokus pada total cost dari starting node, GBFS menggunakan teknik heuristik untuk menentukan penelusurannya dengan tujuan untuk meng-expand node yang tampaknya terdekat dengan node tujuan berdasarkan heuristik.

Beberapa karakteristik GBFS adalah sebagai berikut.

1. Heuristic-Based Expansion  
GBFS menggunakan fungsi heuristik  $h(n)$  untuk menentukan estimasi cost dari suatu node ke node tujuan. Node yang di-expand adalah tetangga dari node sekarang yang memiliki cost terendah.
2. Priority Queue  
GBFS menggunakan priority queue dalam implementasinya, untuk menentukan node mana yang selanjutnya akan di-visit dan di-expand. Priority queue disini mengurutkan node dari yang paling kecil jumlah cost heuristiknya.
3. Non-Optimal dan Incomplete  
GBFS tidak dipastikan memberikan solusi optimal dan penelusuran dilakukan secara incomplete. Sifat GBFS yang selalu mencari dan langsung mengambil rute yang dianggapnya “optimal” berdasarkan heuristik tanpa adanya *backtracking* ini yang membuatnya sering mendapatkan solusi non-optimal ataupun tidak menemukan solusi ketika solusi tersebut ada.

Berikut adalah langkah-langkah dari Greedy Best First Search.

1. Mulai dengan node awal  
Node awal diinisialisasi dengan cost 0. Masukkan node awal pada prio-queue
2. Selagi prio-queue belum kosong, dequeue prio-queue sehingga mendapatkan node dengan cost terendah saat itu.
3. Jika node yang didapatkan sudah pernah dikunjungi, skip dan lakukan dequeue kembali
4. Jika node tersebut adalah node tujuan, didapatkan solusi dan kembalikan route yang didapatkan.
5. Jika bukan, lanjutkan ekspansi dengan memeriksa tetangga-tetangga dari node tersebut.
6. Tambahkan tiap tetangga yang belum dikunjungi pada priority queue dengan cost-nya merupakan cost heuristik dari tetangga tersebut.
7. Jika prio-queue sudah kosong dan node tujuan belum dikunjungi, berarti solusi jalur tidak ada.

## C. A\* (A Star) Search

A\* Search merupakan salah satu bentuk Informed Search yang menghindari pencarian jalur yang sudah “mahal” dengan menggunakan fungsi evaluasi berupa gabungan dari fungsi UCS dan GBFS. A\* juga digunakan pada pencarian shortest path pada *weighted graph* ataupun pathfinding.

Beberapa karakteristik GBFS adalah sebagai berikut.

4. Heuristic and Cost based Expansion  
A\* dalam menentukan “cost” dari suatu node menggunakan total dari cost sebenarnya node awal ke node dan nilai heuristik dari node menuju node tujuan
5. Priority Queue  
GBFS menggunakan priority queue dalam implementasinya, untuk menentukan node mana yang selanjutnya akan di-visit dan di-expand. Priority queue disini mengurutkan node dari yang paling kecil jumlah cost heuristiknya.
6. Optimality dan Completeness  
Algoritma A\* dikatakan optimal jika heuristik yang digunakan *admissible* dan consistent. Algoritma ini juga *complete* dan akan menemukan solusi (jika ada) apabila diberikan graf adalah finite dan heuristik *admissible*.

Berikut adalah langkah-langkah dari Greedy Best First Search.

1. Mulai dengan node awal  
Node awal diinisialisasi dengan cost 0. Masukkan node awal pada prio-queue
2. Selagi prio-queue belum kosong, dequeue prio-queue sehingga mendapatkan node dengan cost terendah saat itu.
3. Jika node yang didapatkan sudah pernah dikunjungi, perbarui minimum cost path node tersebut jika hasil dequeue memiliki cost yang lebih kecil. Jika minimum cost path lebih kecil, abaikan node dan lakukan dequeue kembali.
4. Jika node tersebut adalah node tujuan, didapatkan solusi dan kembalikan route yang didapatkan.
5. Jika bukan, lanjutkan ekspansi dengan memeriksa tetangga-tetangga dari node tersebut.
6. Tambahkan tiap tetangga yang belum dikunjungi pada priority queue dengan cost-nya merupakan cost heuristik-nya ditambah cost node sekarang ditambah *weight edge*-nya.
7. Jika prio-queue sudah kosong dan node tujuan belum dikunjungi, berarti solusi jalur tidak ada.

## BAB II

### ANALISIS DAN IMPLEMENTASI

*Word Ladder* merupakan permainan kata yang dapat diselesaikan dengan algoritma route planning seperti UCS, GBFS, dan A\*. Permainan ini mencari solusi route dari kata awal hingga kata tujuan yang dimana sesuai dengan tipe permasalahan pathfinding yang dapat diselesaikan ketiga algoritma ini. Berikut adalah analisis dan implementasi dari algoritma UCS, GBFS, dan juga A\*.

#### A. Uniform Cost Search (UCS)

Implementasi UCS pada permainan *Word Ladder* memiliki kemiripan dengan algoritma umum-nya. Di sini nilai cost dari suatu node ditentukan dari jumlah perubahan huruf yang dilakukan dari node awal hingga node tersebut. Hal ini membuat nilai cost dari tetangga suatu node ketika dilakukan ekspansi adalah cost node tersebut ditambah satu. Penambahan cost sebanyak satu membuat algoritma UCS ini lebih cenderung mirip dengan BFS yang menetapkan cost sebagai nilai step yang memiliki selisih 1 juga antar node terhubungnya.

Berikut adalah langkah-langkah implementasi algoritma UCS pada permainan *Word Ladder*.

1. Membuat priority queue yang mengurutkan node-node di dalam berdasarkan jumlah cost dari yang terkecil hingga terbesar. Dalam hal ini, cost yang dimaksud adalah total perubahan karakter sejauh ini dari kata node awal hingga kata node tersebut.
2. Menambahkan node awal pada priority queue, yang dimana node ini memiliki kata yakni kata awal, jumlah cost yakni 0, dan parent adalah tidak ada.
3. Lakukan dequeue sehingga mendapatkan node dengan nilai cost terkecil saat itu.
4. Jika kata pada node tersebut sudah dikunjungi, skip node tersebut dan lakukan dequeue kembali
5. Tambahkan kata node sekarang pada daftar kata yang telah dikunjungi
6. Jika kata pada node sekarang sama dengan kata tujuan kita, solusi telah ditemukan dan keluar dari while loop
7. Dapatkan daftar suksesor dari kata node sekarang dengan mengganti satu indeks pada kata dengan huruf dari 'a' sampai 'z'. Lakukan untuk tiap indeks satu per satu. Lakukan pengecekan juga apakah kata tersebut valid (terdapat pada kamus).
8. Iterasi tiap suksesor tersebut. Jika kata dari suksesor tersebut belum pernah dikunjungi, tambahkan node baru pada priority queue dengan katanya adalah kata suksesor tersebut, parent nya adalah node sekarang, dan nilai cost nya adalah cost node sekarang ditambah satu.
9. Ulangi langkah 2 sampai 8 hingga priority queue kosong atau solusi telah ditemukan
10. Jika solusi ditemukan (kata terakhir yang di-dequeue adalah kata tujuan), solusi ditemukan dan kembalikan route tersebut
11. Jika tidak, solusi untuk permainan word ladder dengan kata awal dan tujuan tersebut tidak ditemukan

Dapat dilihat pada langkah nomor 4, ketika kita mendapatkan node hasil dequeue yang pernah dikunjungi, kita langsung mengabaikannya tanpa mempertimbangkan jika kata hasil dequeue tersebut lebih optimal. Hal ini disebabkan karena suatu kata yang muncul pertama kali pada queue, dipastikan memiliki cost lebih kecil daripada kata yang sama berikutnya yang

masuk pada queue. Implementasi UCS ini memiliki sifat seperti BFS yang melakukan pencarian layer secara bertahap, sehingga suatu kata yang muncul duluan pada queue pasti berasal dari layer yang lebih kecil daripada kata yang sama berikutnya, yang berarti kata sama yang duluan memiliki cost lebih kecil.

Kompleksitas waktu dan ruang dari implementasi UCS ini adalah  $O(b^d)$  dengan  $b$  adalah *branching factor* dan  $d$  adalah kedalaman dari solusi optimal. Penentuan eksak nilai kompleksitasnya sulit ditentukan karena nilai *branching factor* yang beragam. Keberagaman nilai branching factor disebabkan karena tidak semua hasil pengubahan karakter pada suatu indeks kata menghasilkan kata yang valid.



## B. Greedy Best First Search (GBFS)

Implementasi GBFS pada permainan *Word Ladder* ini menggunakan teknik heuristik yakni penghitungan cost suatu kata/node berdasarkan jumlah karakter yang berbeda dengan kata tujuan. Alur GBFS pada program ini mungkin agak berbeda dengan GBFS umum yang menggunakan priority queue. Implementasi GBFS ini pada tiap ekspansi, akan memilih tetangga dengan cost heuristik terkecil, dan melanjutkan pengulangan dengan node tetangga tersebut. Berikut adalah langkah-langkah implementasi algoritma UCS pada permainan *Word Ladder*.

1. Memulai pencarian dengan menelusuri node awal dahulu. Node awal adalah node dengan kata awal, parent node tidak ada, dan nilai cost adalah nilai heuristik dari kata-nya.
2. Tandai node sekarang telah dikunjungi.
3. Jika node sekarang memiliki kata yang sama dengan kata tujuan, solusi telah ditemukan dan keluar dari *loop*
4. Dapatkan daftar suksesor dari kata node sekarang dengan mengganti satu indeks pada kata dengan huruf dari 'a' sampai 'z'. Lakukan untuk tiap indeks satu per satu. Lakukan pengecekan juga apakah kata tersebut valid (terdapat pada kamus).
5. Inisialisasi variable min dan word, yang digunakan untuk menampung kata suksesor dengan nilai heuristik yang terkecil.
6. Iterasi tiap suksesor dari node sekarang. Jika suksesor belum dikunjungi dan nilai heuristik nya lebih kecil dari nilai minimum sekarang, perbarui nilai minimum dan kata-nya.
7. Jika semua suksesor telah dikunjungi sebelumnya, maka solusi tidak ditemukan dan keluar dari *loop*
8. Lanjut penelusuran dengan node baru, yakni node dengan kata yang memiliki cost terkecil, nilai cost heuristiknya, dan parent-nya adalah node sekarang.
9. Ulangi langkah 2 sampai 8 hingga daftar suksesor node semua telah dikunjungi atau solusi telah ditemukan
10. Jika solusi ditemukan (kata pada current node terakhir sama dengan kata tujuan), solusi ditemukan dan kembalikan route tersebut
11. Jika tidak, solusi untuk permainan word ladder dengan kata awal dan tujuan tersebut tidak ditemukan

Algoritma GBFS ini tidak dipastikan menemukan solusi yang global optimal ataupun menemukan solusi sama sekali. Hal ini karena sifat GBFS yang langsung mengambil tetangga/suksesor dengan cost terkecil dan lanjut bergerak dari situ tanpa melakukan backtracking apabila rute tersebut salah, sehingga rawan salah dan route “buntu”.

Di sini, suatu node dipastikan hanya muncul sekali karena dua atau lebih node dengan kata yang sama akan memiliki cost heuristik yang sama, tetapi jumlah perubahan karakter yang berbeda (“distance” dari kata awal). Sama seperti pada UCS, kata yang muncul duluan menandakan kata tersebut berasal dari ekspansi node dengan layer yang lebih kecil / lebih dekat dengan kata awal sehingga akan menjadi optimal jika langsung membatasi penambahan kata yang sudah pernah ada pada priority queue.

Kompleksitas waktu dan ruang dari implementasi UCS ini adalah  $O(bd)$  dengan  $b$  adalah *branching factor* dan  $d$  adalah kedalaman dari solusi optimal. Kompleksitas ini sangat kecil

dibandingkan algoritma UCS yang eksponensial. Walaupun memiliki kompleksitas yang rendah dari segi waktu dan ruang, *downside*-nya adalah pencarian GBFS tidak complete dan tidak pasti menemukan solusi yang global optimal.

### C.A\* Algorithm

Algoritma A\* merupakan algoritma yang menggabungkan konsep cost pada UCS (actual distance cost) dengan pada GBFS (heuristic cost). Implementasi dari algoritma ini pada permainan *Word Ladder* menggunakan konsep cost yang merupakan total dari perubahan karakter dari kata awal hingga sekarang (cost pada UCS) ditambah dengan nilai perbedaan huruf karakter kata tujuan dengan kata tersebut (cost pada GBFS). Berikut bentuk cost pada A\* dalam bentuk notasi rumus.

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

dengan  $f(n)$  adalah cost A\*,  $g(n)$  adalah cost pada UCS, dan  $h(n)$  adalah cost pada GBFS.

Suatu algoritma A\* dikatakan akan menghasilkan solusi yang optimal global, jika teknik heuristik yang digunakan *admissible*, yakni tidak meng-*overestimate* minimum step yang diperlukan untuk mencapai tujuan. Teknik heuristik  $h(n)$  yang digunakan pada implementasi ini adalah *admissible*, karena tiap perbedaan karakter antara dua kata dapat diubah dengan mengubah satu karakter. Sehingga, jumlah langkah untuk mencapai kata tujuan adalah sebanyak perbedaan karakter tersebut. Karena tiap langkah permainan hanya bisa membawa kita semakin dekat dengan kata tujuan sebanyak satu karakter, maka teknik heuristiknya menyediakan batas bawah pada jumlah langkah permainan yang sebenarnya diperlukan.

Berikut adalah langkah-langkah implementasi A\* pada permainan *Word Ladder*.

1. Membuat priority queue yang mengurutkan node-node di dalam berdasarkan jumlah cost dari yang terkecil hingga terbesar. Dalam hal ini, cost yang dimaksud adalah total perubahan karakter sejauh ini dari kata awal hingga kata tersebut, ditambah dengan jumlah karakter berbeda antara kata tersebut dengan kata tujuan.
2. Menambahkan node awal pada priority queue, yang dimana node ini memiliki kata yakni kata awal, jumlah cost yakni 0, parent tidak ada, dan jarak dari node awal adalah 0.
3. Lakukan dequeue sehingga mendapatkan node dengan nilai cost terkecil saat itu.
4. Jika kata pada node tersebut sudah dikunjungi, skip node tersebut dan lakukan dequeue kembali untuk mendapatkan node baru.
5. Tambahkan kata node sekarang pada daftar kata yang telah dikunjungi
6. Jika kata pada node sekarang sama dengan kata tujuan kita, solusi telah ditemukan dan keluar dari while loop
7. Dapatkan daftar suksesor dari kata node sekarang dengan mengganti satu indeks pada kata dengan huruf dari 'a' sampai 'z'. Lakukan untuk tiap indeks satu per satu. Lakukan pengecekan juga apakah kata tersebut valid (terdapat pada kamus).
8. Iterasi tiap suksesor tersebut. Jika kata dari suksesor tersebut belum pernah dikunjungi, tambahkan node baru pada priority queue dengan katanya adalah kata suksesor tersebut, parent nya adalah node sekarang, dan nilai cost nya adalah jarak sebenarnya node sekarang ditambah nilai heuristik node suksesor.
9. Ulangi langkah 2 sampai 8 hingga priority queue kosong atau solusi telah ditemukan
10. Jika solusi ditemukan (kata terakhir yang di-dequeue adalah kata tujuan), solusi ditemukan dan kembalikan route tersebut
11. Jika tidak, solusi untuk permainan word ladder dengan kata awal dan tujuan tersebut tidak ditemukan

Bentuk dan alur algoritma A\* memiliki kemiripan dengan algoritma UCS sebelumnya. Dapat dilihat, sama seperti UCS, kita melakukan skip ketika hasil dequeue merupakan suatu kata yang sudah pernah dikunjungi. Hal ini sama seperti pada UCS, karena kata yang muncul terlebih dahulu pada queue dipastikan memiliki cost yang lebih kecil dibanding kata sama yang muncul setelahnya, karena nilai heuristiknya sama dan penambahan kata yang pertama muncul berasal dari layer yang lebih dekat dengan node awal (*actual distance* terkecil).

## BAB III


### SOURCE CODE

Sesuai spesifikasi, pembuatan algoritma *pathfinding Word Ladder* ini menggunakan bahasa Java. Untuk pembuatan GUI, saya menggunakan Java Swing dengan Netbeans IDE sebagai designernya. Berikut kode tiap Java class pada logic *pathfinding Word Ladder*.

#### 1. Class Node

```
1  public class Node {
2      private String word;
3      private Node parent;
4      private Integer cost;
5      private Integer distFromStart;
6
7      public Node(String word, Node parent, Integer cost, Integer distFromStart){
8          this.word = word;
9          this.parent = parent;
10         this.cost = cost;
11         this.distFromStart = distFromStart;
12     }
13
14     public Integer getCost(){
15         return cost;
16     }
17
18     public Node getParent(){
19         return parent;
20     }
21
22     public String getWord(){
23         return word;
24     }
25
26     public Integer getDistFromStart(){
27         return distFromStart;
28     }
29
30     public ArrayList<String> expandChild(){
31         ArrayList<String> childs = new ArrayList<>();
32         int lengthWord = word.length();
33         for(int i=0; i<lengthWord; i++){
34             StringBuilder str = new StringBuilder(word);
35             char charTarget = str.charAt(i);
36             for(char c='a'; c<='z'; c++){
37                 if(charTarget==c){
38                     continue;
39                 }
40                 str.setCharAt(i, c);
41                 if(Dict.checkWord(str.toString())){
42                     childs.add(str.toString());
43                 }
44             }
45         }
46         return childs;
47     }
48 }
```

## 2. Class Graph



```
1  abstract public class Graph {
2      private String startingWord;
3      private String destWord;
4      private HashMap<String,Boolean> visited;
5
6      public Graph(String startingWord,String destWord){
7          this.startingWord = startingWord;
8          this.destWord = destWord;
9          visited = new HashMap<>();
10     }
11
12     public String getStartWord(){
13         return startingWord;
14     }
15
16     public String getDestWord(){
17         return destWord;
18     }
19
20     public Boolean checkVisited(String word){
21         return visited.containsKey(word);
22     }
23
24     public void markVisited(String word){
25         visited.put(word, true);
26     }
27
28     abstract public Solution traverseSolution();
29 }
```

### 3. Class UCSGraph

```
1 public class UCSGraph extends Graph{
2     private PriorityQueue<Node> nodeQueue;
3
4     public UCSGraph(String startingWord,String destWord){
5         super(startingWord,destWord);
6         nodeQueue = new PriorityQueue<>(new NodeComparator());
7     }
8
9     @Override
10    public Solution traverseSolution(){
11        long startTime = System.currentTimeMillis();
12
13        nodeQueue.add(new Node(getStartWord(), null, 0,null));
14
15        int nodesVisited = 0;
16        Node curNode = null;
17
18        while (!nodeQueue.isEmpty()) {
19            curNode = nodeQueue.poll();
20            nodesVisited++;
21
22            if(checkVisited(curNode.getWord())) {
23                continue;
24            }
25
26            markVisited(curNode.getWord());
27            if(curNode.getWord().equals(getDestWord())){
28                break;
29            }
30
31            ArrayList<String> childs = curNode.expandChild();
32            for(String child:childs){
33                if(!checkVisited(child)){
34                    nodeQueue.add(new Node(child, curNode, curNode.getCost()+1,null));
35                }
36            }
37        }
38        long endTime = System.currentTimeMillis();
39        double duration = (endTime-startTime);
40
41        if(curNode.getWord().equals(getDestWord())){
42            return new Solution(curNode, duration, nodesVisited);
43        }
44        return new Solution(null, duration, nodesVisited);
45    }
46 }
```

## 4. Class GBFSGraph

```
1 public class GBFSGraph extends Graph{
2
3     public GBFSGraph(String startingWord,String destWord){
4         super(startingWord,destWord);
5     }
6
7     public Integer cost(String word){
8         Integer sum = 0;
9         for(int i=0;i<word.length();i++){
10             if(word.charAt(i)!=getDestWord().charAt(i)){
11                 sum++;
12             }
13         }
14         return sum;
15     }
16
17     @Override
18     public Solution traverseSolution(){
19         long startTime = System.currentTimeMillis();
20
21         Node node = new Node(getStartWord(), null, cost(getStartWord()),null);
22         markVisited(getStartWord());
23
24         int nodesVisited = 0;
25         while(true){
26             nodesVisited++;
27             markVisited(node.getWord());
28             if(node.getWord().equals(getDestWord())){
29                 break;
30             }
31
32             ArrayList<String> children = node.expandChild();
33             Integer min = 9999;
34             String word="";
35             for(String child:children){
36                 if(!checkVisited(child) && cost(child)<min){
37                     min = cost(child);
38                     word = child;
39                 }
40             }
41             if(min==9999) {
42                 break;
43             }
44             node = new Node(word, node, min,null);
45         }
46         long endTime = System.currentTimeMillis();
47         double duration = endTime-startTime;
48
49         if(node.getWord().equals(getDestWord())){
50             return new Solution(node, duration, nodesVisited);
51         }
52         return new Solution(null, duration, nodesVisited);
53     }
54 }
55 }
```



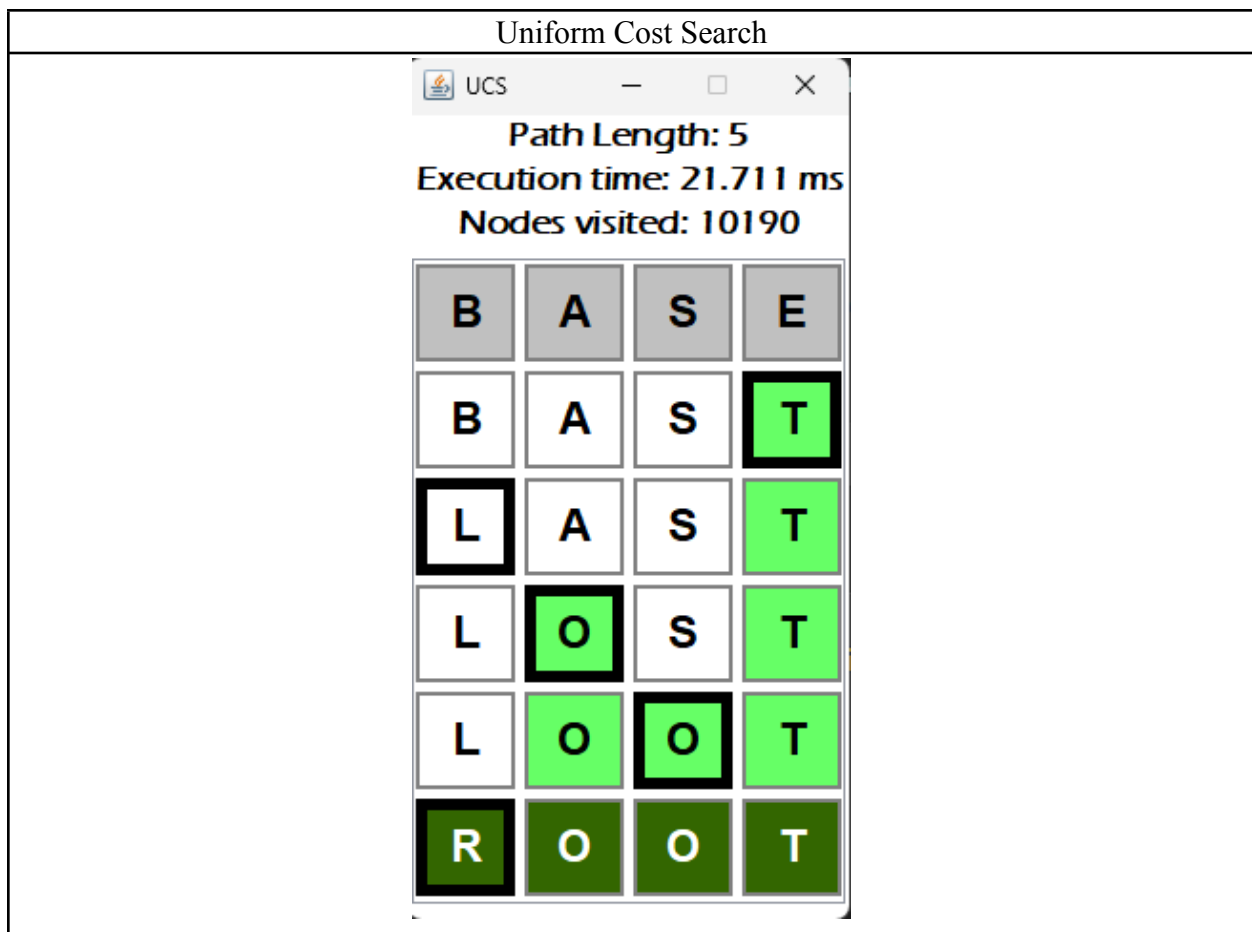
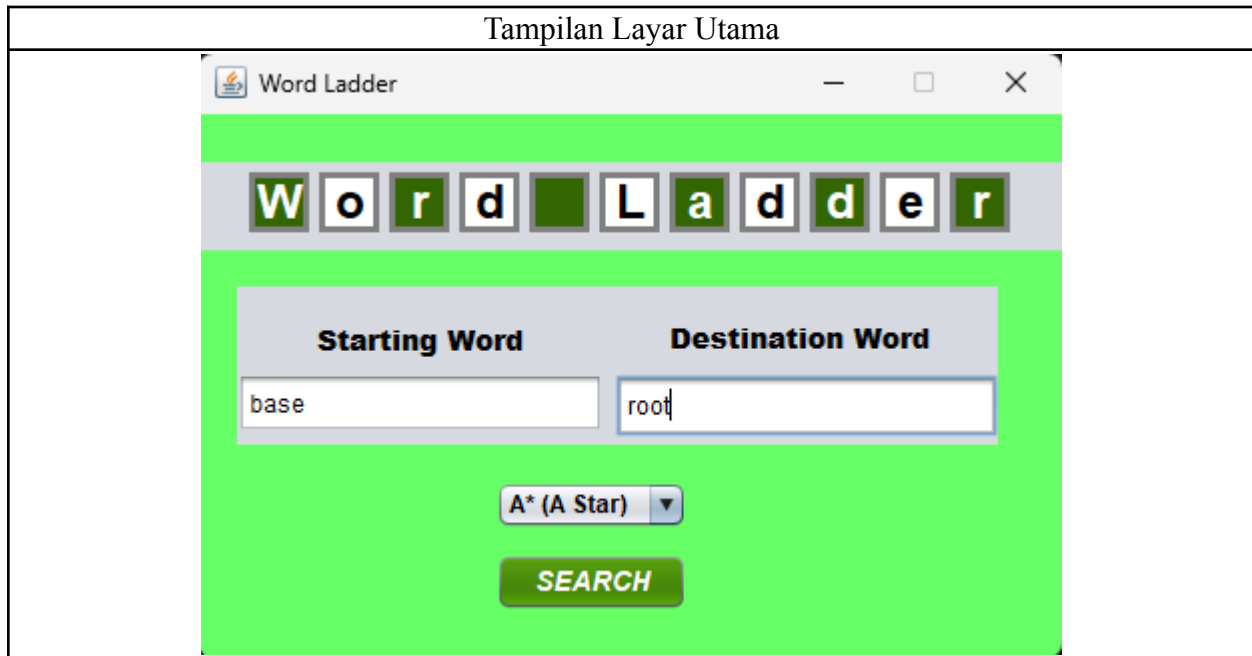
## 5. Class AStarGraph

```
1 public class AStarGraph extends Graph{
2     private PriorityQueue<Node> nodeQueue;
3
4     public AStarGraph(String startingWord,String destWord){
5         super(startingWord,destWord);
6         nodeQueue = new PriorityQueue<>(new NodeComparator());
7     }
8
9     public Integer cost(String word){
10         Integer sum = 0;
11         for(int i=0;i<word.length();i++){
12             if(word.charAt(i)!=getDestWord().charAt(i)){
13                 sum++;
14             }
15         }
16         return sum;
17     }
18
19     @Override
20     public Solution traverseSolution(){
21         long startTime = System.currentTimeMillis();
22
23         nodeQueue.add(new Node(getStartWord(), null, 0,0));
24         Node curNode = null;
25         int nodesVisited = 0;
26         boolean found = false;
27
28         while (!nodeQueue.isEmpty()) {
29             curNode = nodeQueue.poll();
30             nodesVisited++;
31             if(checkVisited(curNode.getWord())){
32                 continue;
33             }
34
35             markVisited(curNode.getWord());
36             if(curNode.getWord().equals(getDestWord())){
37                 found = true;
38                 break;
39             }
40
41             ArrayList<String> childs = curNode.expandChild();
42             for(String child:childs){
43                 if(!checkVisited(child)){
44                     Integer childCost = curNode.getDistFromStart() + 1 + cost(child);
45                     nodeQueue.add(new Node(child, curNode, childCost,curNode.getDistFromStart()+1));
46                 }
47             }
48         }
49
50         long endTime = System.currentTimeMillis();
51         double duration = endTime-startTime;
52         if(found){
53             return new Solution(curNode, duration, nodesVisited);
54         }
55         return new Solution(null, duration, nodesVisited);
56     }
57 }
```

## BAB IV

### HASIL PENGUJIAN

#### A. Uji Kasus 1 ( BASE -> ROOT)



## Greedy Best First Search

GBFS  
Path Length: 10  
Execution time: 0.265 ms  
Nodes visited: 11

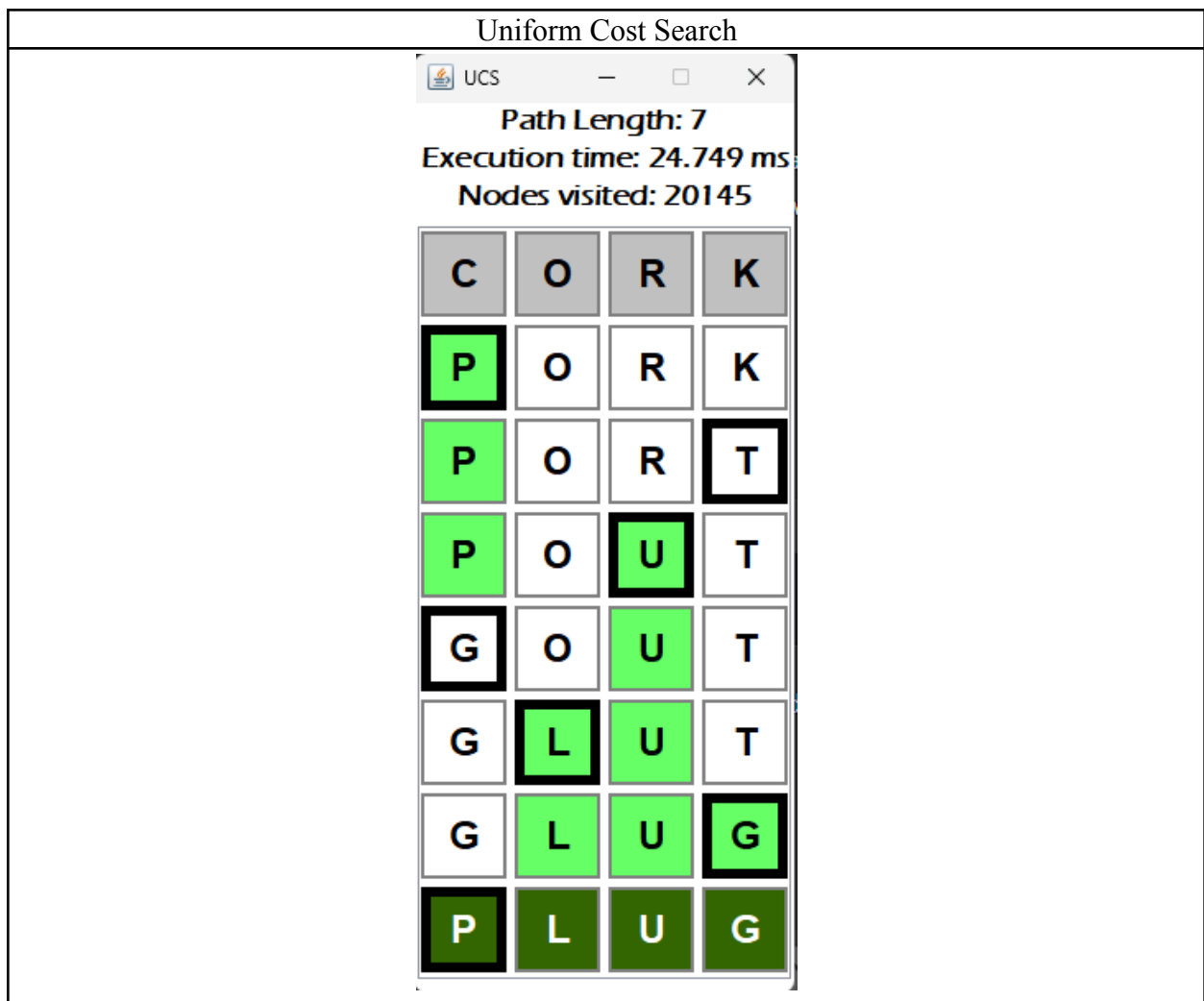
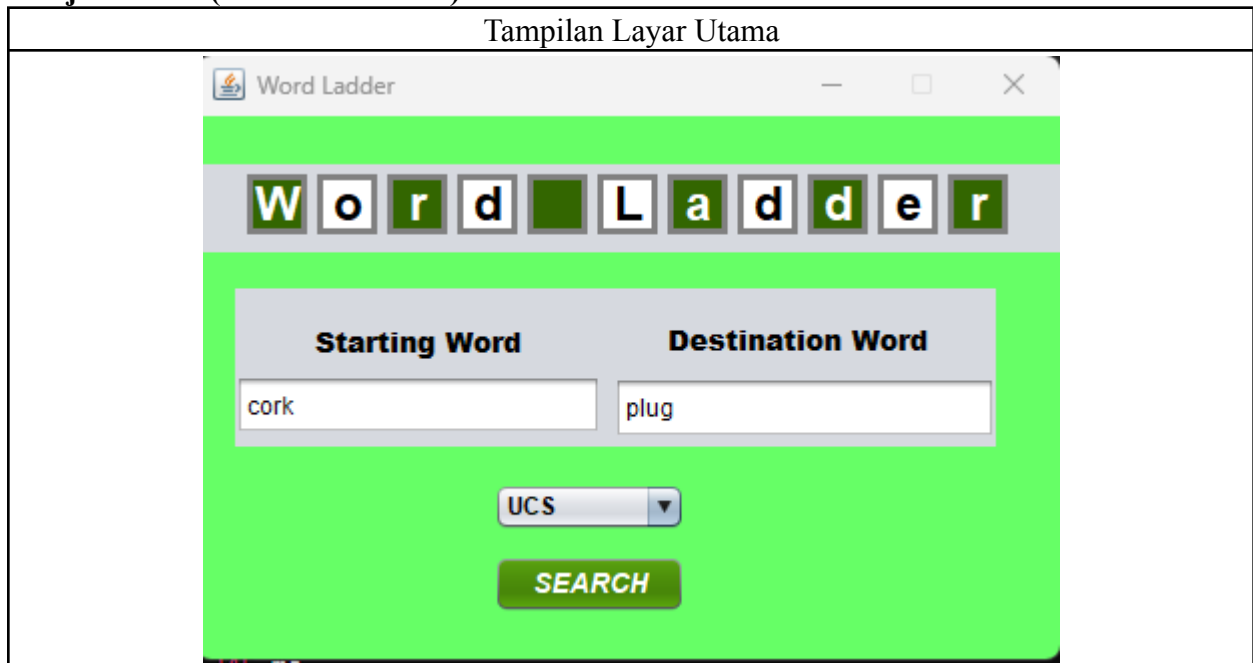
B	A	S	E
R	A	S	E
R	O	S	E
R	O	B	E
R	O	D	E
R	O	L	E
R	O	P	E
R	O	T	E
R	O	U	E
R	O	U	T
R	O	O	T

## A\* Algorithm

A\* (A S...  
Path Length: 5  
Execution time: 0.837 ms  
Nodes visited: 54

B	A	S	E
L	A	S	E
L	O	S	E
L	O	S	T
L	O	O	T
R	O	O	T

## B. Uji Kasus 2 (CORK -> PLUG)



## Greedy Best First Search

The figure displays four sequential screenshots of a Greedy Best First Search (GBFS) algorithm window. Each window shows a 14x4 grid of letters and provides the following statistics:

- Path Length: 54
- Execution time: 0.741 ms
- Nodes visited: 55

The grids show the search progress, with the target word 'COUP' highlighted in green. The search starts from the top-left and moves towards the bottom-right, following a path that visits 55 nodes.

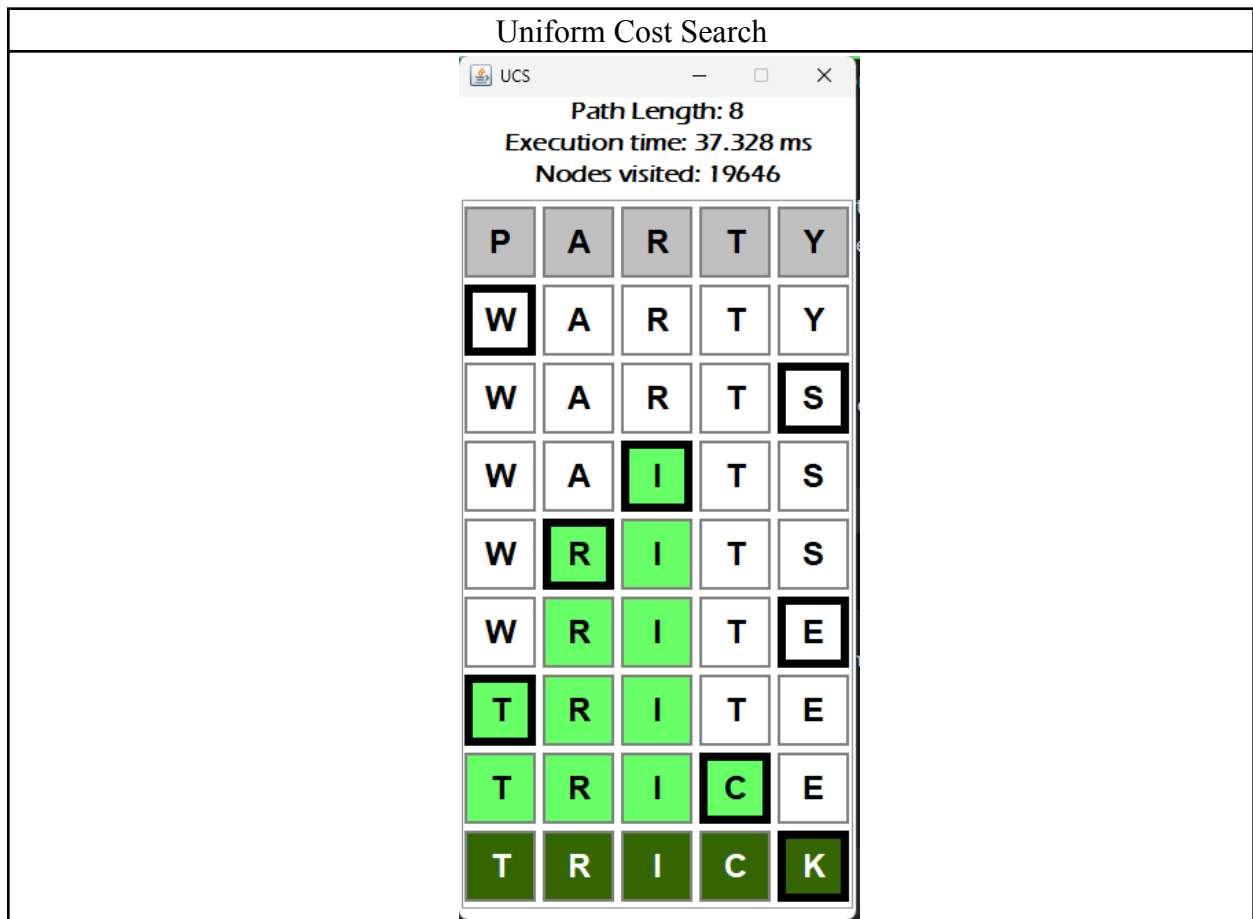
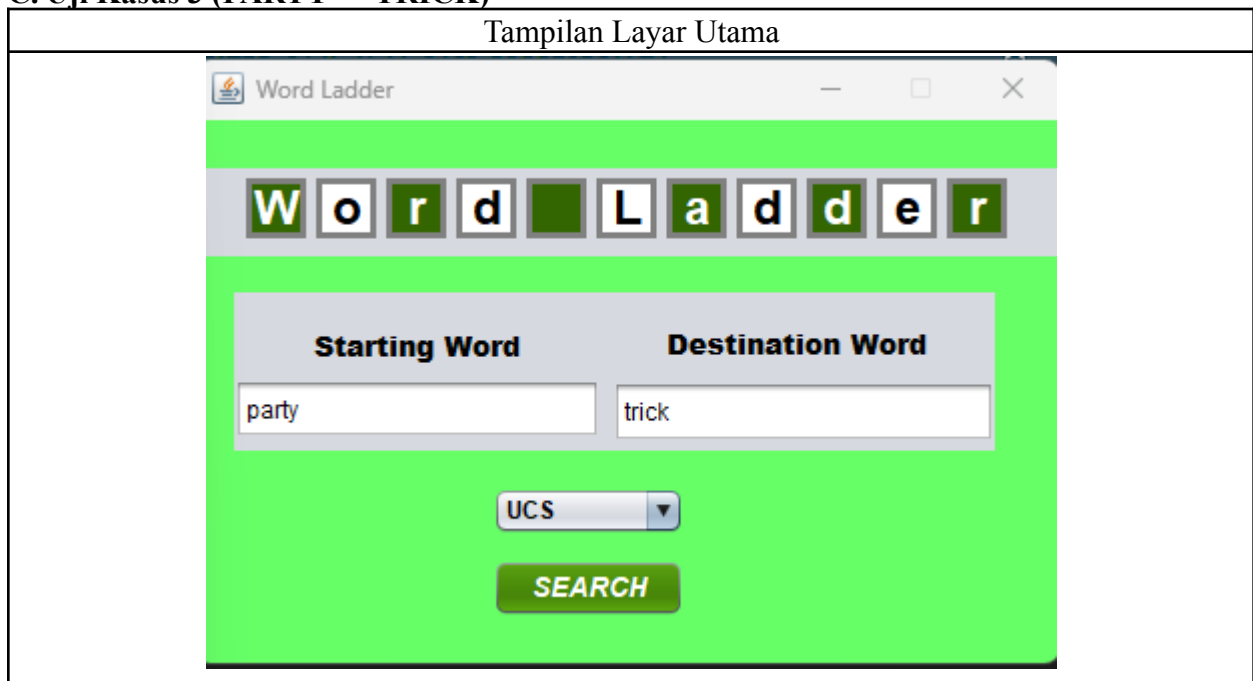
## A\* Algorithm

The figure displays a screenshot of an A\* (A Star) algorithm window. It shows a 14x4 grid of letters and provides the following statistics:

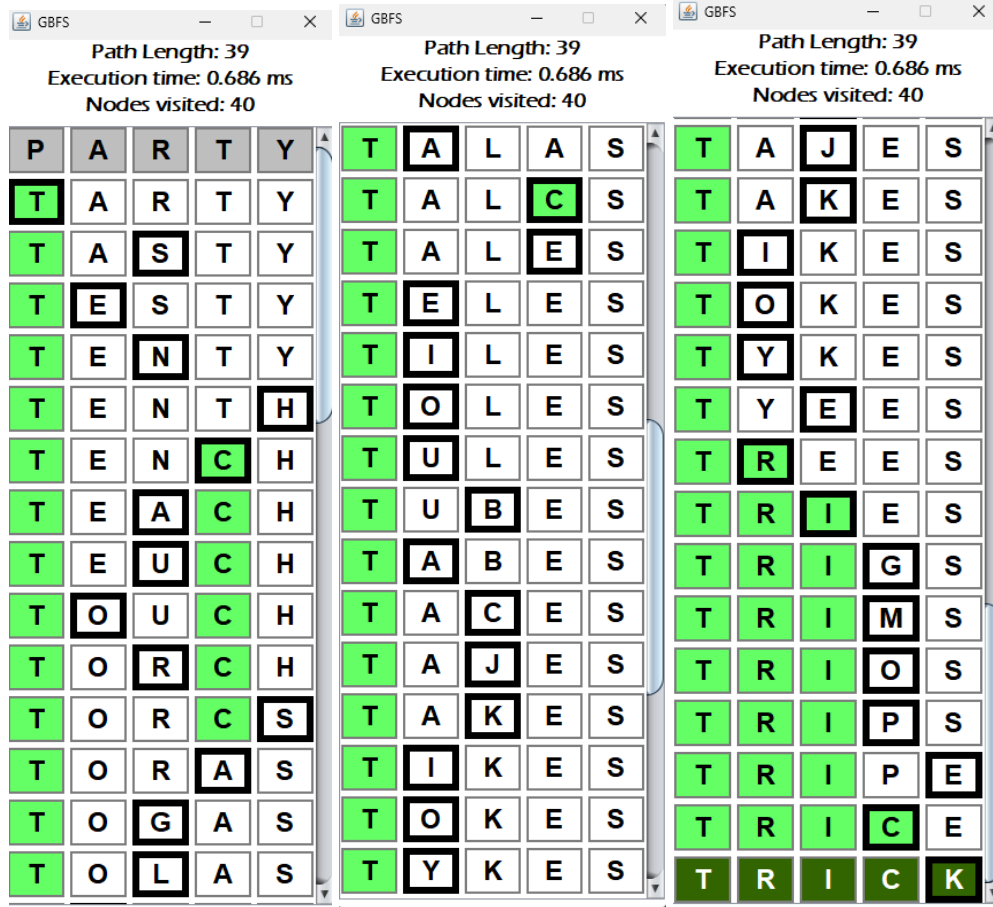
- Path Length: 7
- Execution time: 5.000 ms
- Nodes visited: 823

The grid shows the search progress, with the target word 'COUP' highlighted in green. The search starts from the top-left and moves towards the bottom-right, following a path that visits 823 nodes.

### C. Uji Kasus 3 (PARTY -> TRICK)



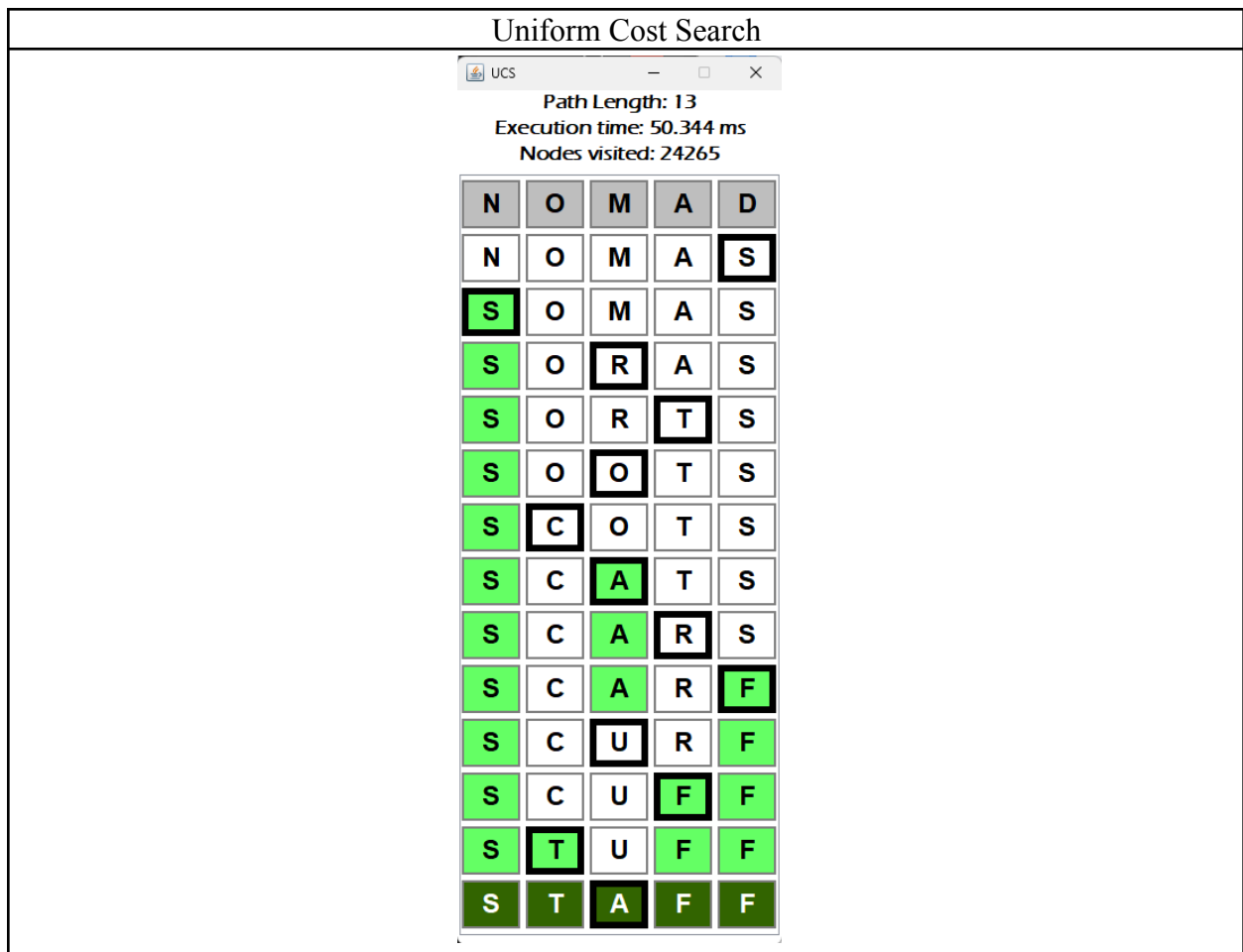
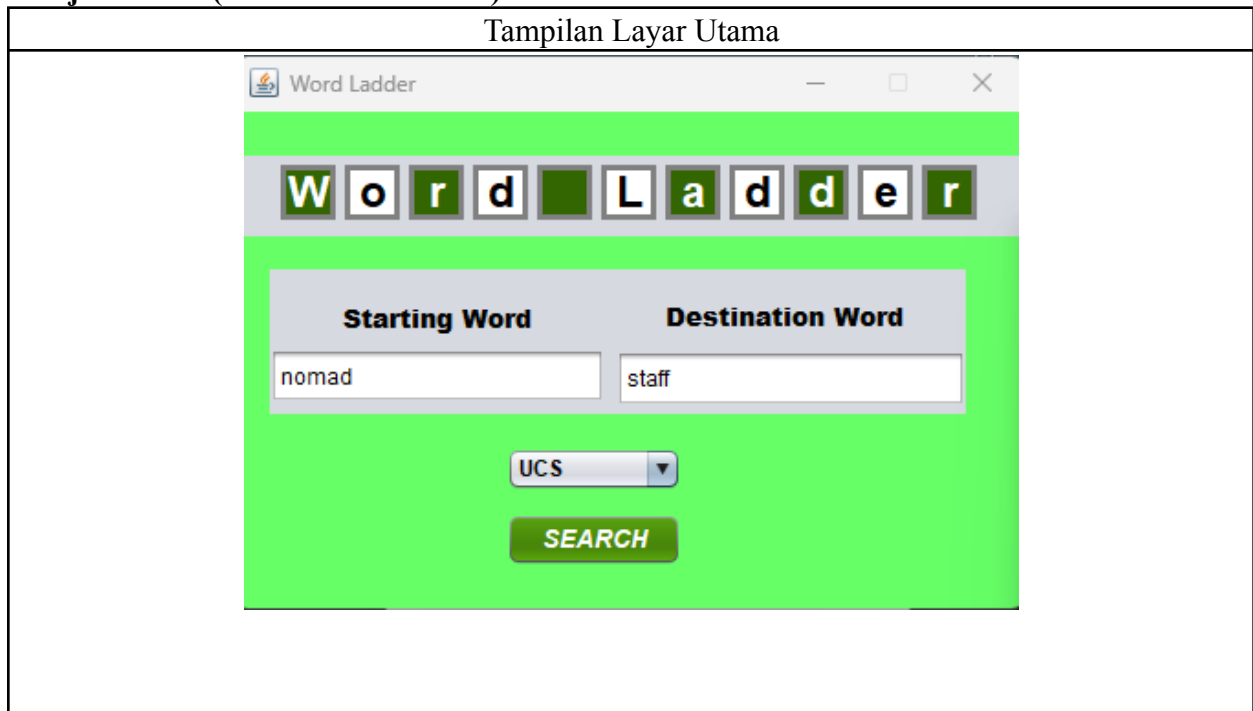
## Greedy Best First Search



## A\* Algorithm

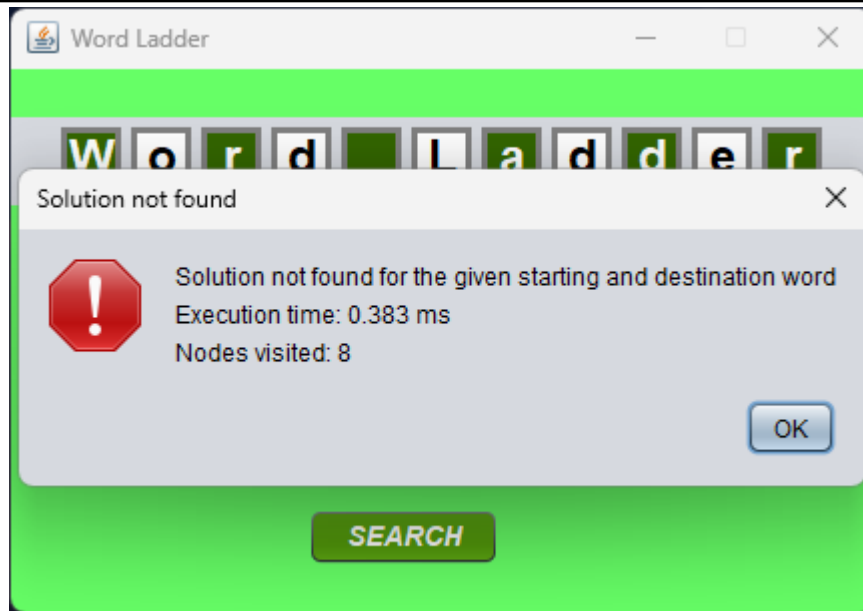


#### D. Uji Kasus 4 (NOMAD -> STAFF)

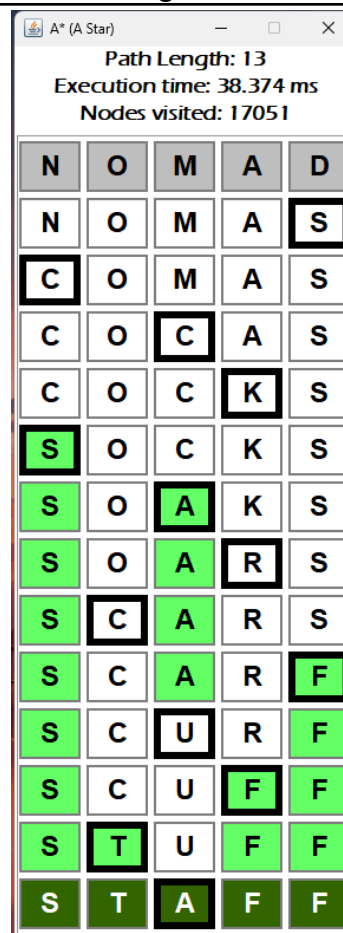




## Greedy Best First Search



## A\* Algorithm



### E. Uji Kasus 5 (STRONG -> MIGHTY)

Tampilan Layar Utama

Word Ladder

Starting Word: strong

Destination Word: mighty

GBFS

SEARCH

Uniform Cost Search

UCS

Path Length: 16  
Execution time: 62.485 ms  
Nodes visited: 20978

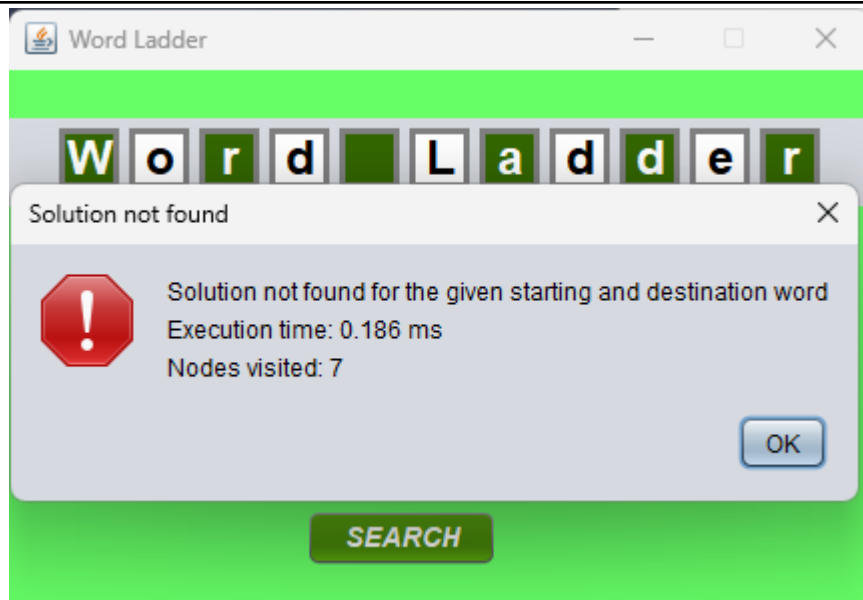
S	T	R	O	N	G
S	T	R	I	N	G
S	O	R	I	N	G
C	O	R	I	N	G
C	O	N	I	N	G
C	O	N	I	N	S
C	O	N	I	E	S
C	O	S	I	E	S
C	O	S	H	E	S
C	A	S	H	E	S
L	A	S	H	E	S
L	A	C	H	E	S
L	I	C	H	E	S
L	I	C	H	T	S
L	I	G	H	T	S

UCS

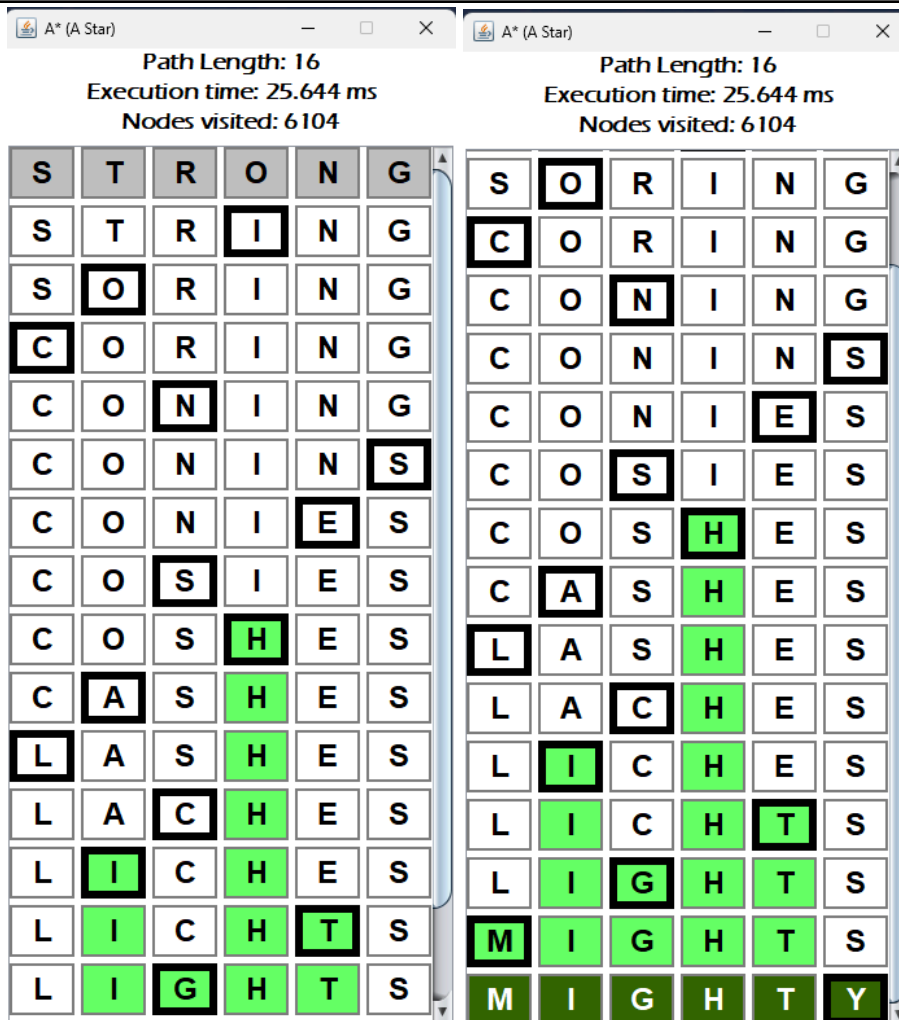
Path Length: 16  
Execution time: 62.485 ms  
Nodes visited: 20978

S	O	R	I	N	G
C	O	R	I	N	G
C	O	N	I	N	G
C	O	N	I	N	S
C	O	N	I	E	S
C	O	S	I	E	S
C	O	S	H	E	S
C	A	S	H	E	S
L	A	S	H	E	S
L	A	C	H	E	S
L	I	C	H	E	S
L	I	C	H	T	S
L	I	G	H	T	S
M	I	G	H	T	S
M	I	G	H	T	Y

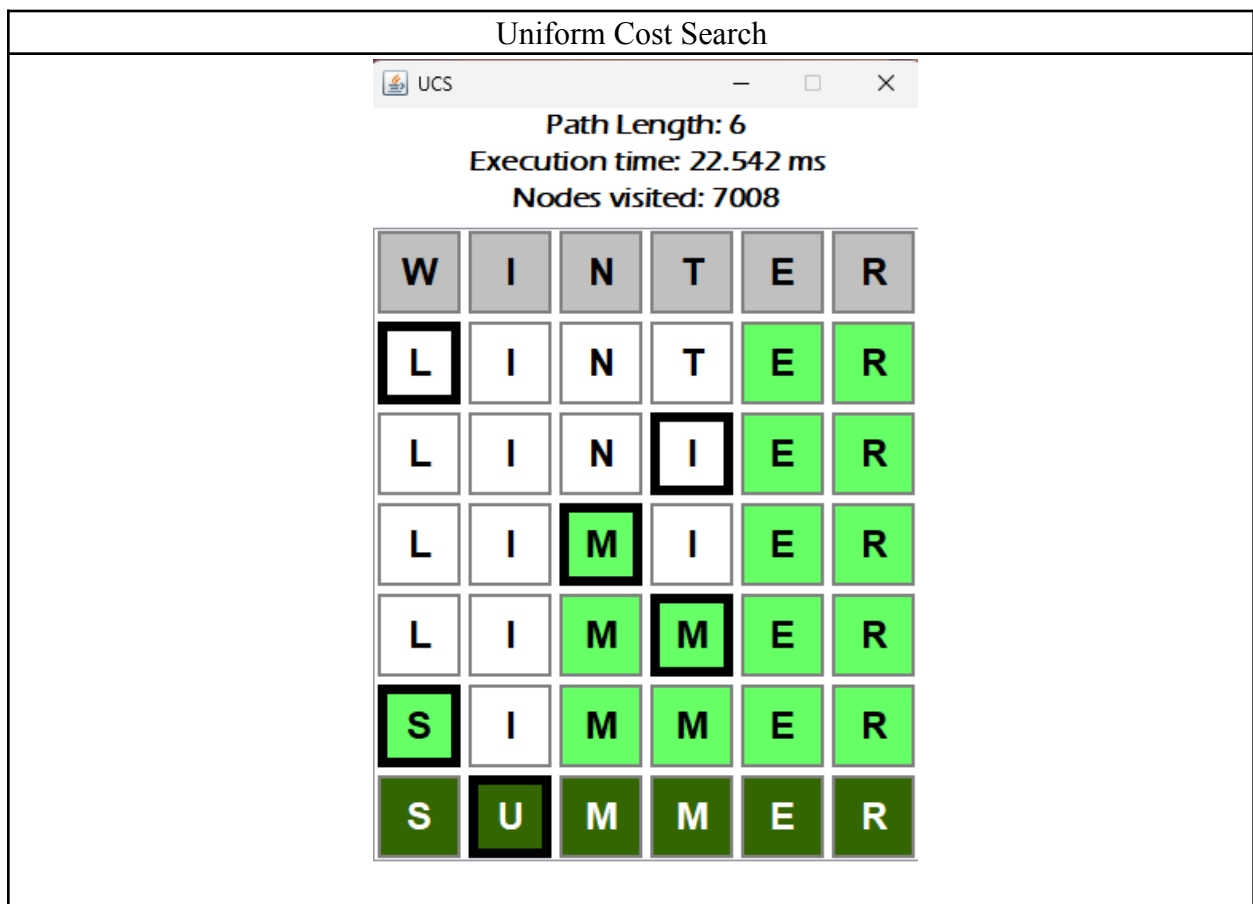
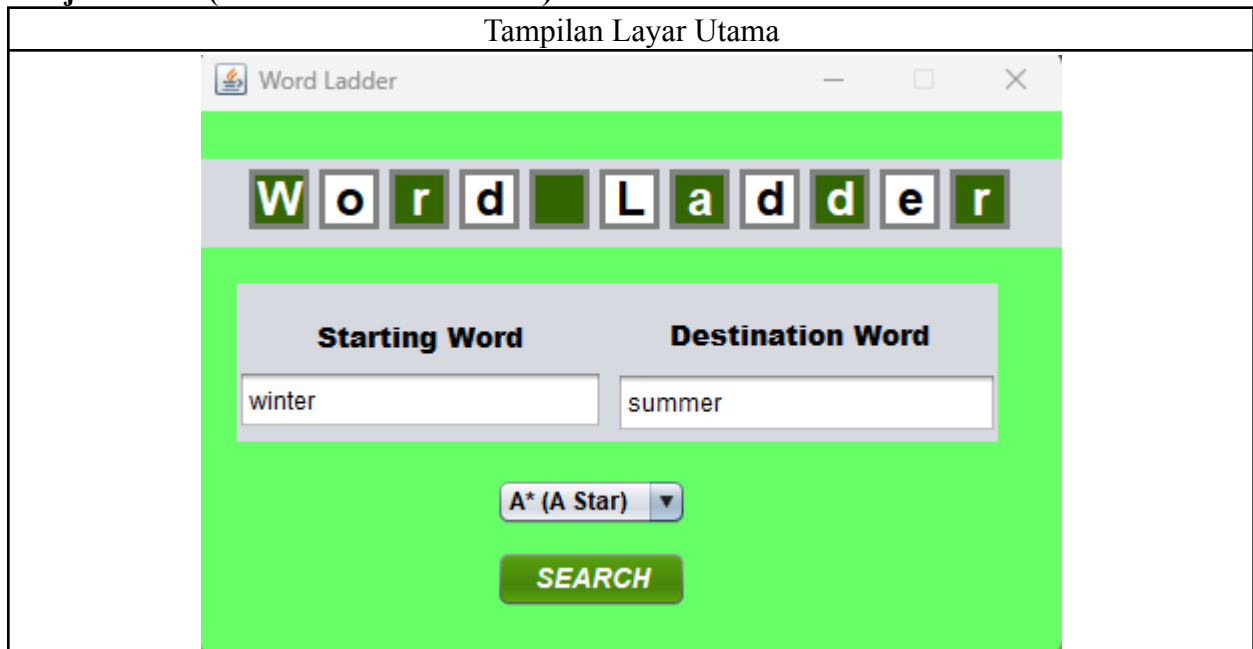
## Greedy Best First Search



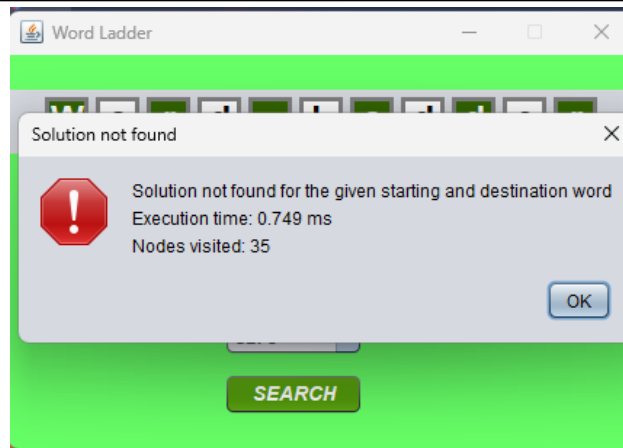
## A\* Algorithm



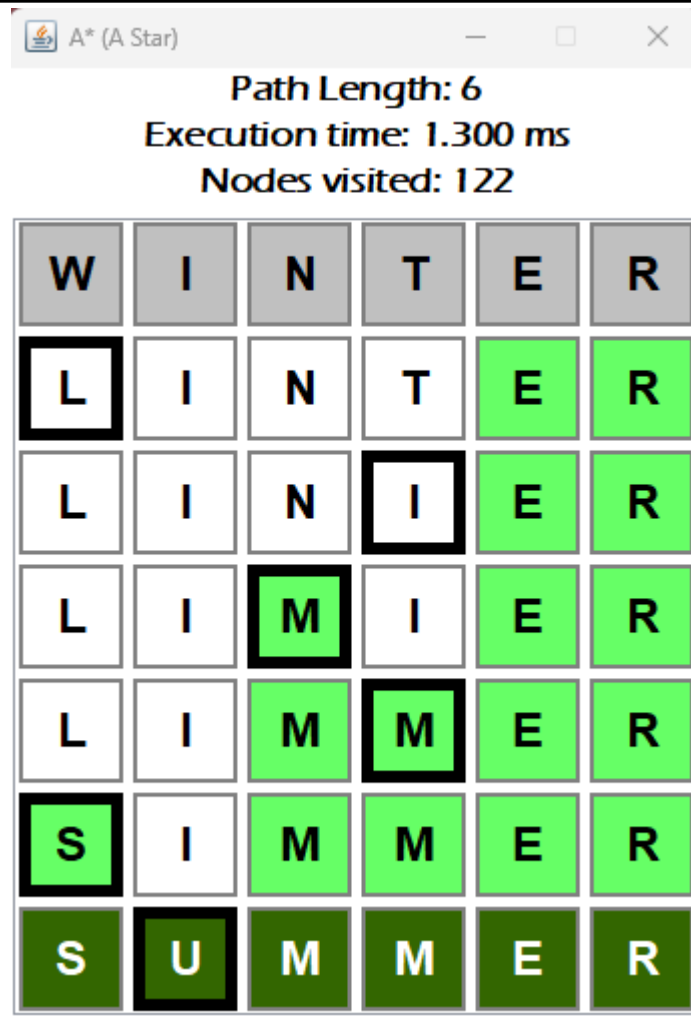
### F. Uji Kasus 6 (WINTER -> SUMMER)



## Greedy Best First Search



## A\* Algorithm



## BAB V

### ANALISIS HASIL PENGUJIAN

No	Algoritma	Waktu (ms)	Visited Nodes	Path Length	Optimal
1	UCS	21.711	10190	5	Iya
	GBFS	0.265	11	10	Tidak
	A*	0.837	54	5	Iya
2	UCS	24.749	20145	7	Iya
	GBFS	0.741	55	54	Tidak
	A*	5.000	823	7	Iya
3	UCS	37.328	19646	8	Iya
	GBFS	0.686	40	39	Tidak
	A*	2.413	242	8	Iya
4	UCS	50.344	24265	13	Iya
	GBFS	0.383	8	-	Tidak
	A*	38.374	17051	13	Iya
5	UCS	62.485	20978	16	Iya
	GBFS	0.186	7	-	Tidak
	A*	25.644	6104	16	Iya
6	UCS	22.542	7008	6	Iya
	GBFS	0.749	35	-	Tidak
	A*	1.300	122	6	Iya

Dapat dilihat pada tabel, walaupun pencarian UCS bersifat complete dan global optimal, UCS memiliki nilai visited nodes dan waktu terbesar. Nilai visited nodes besar ini menandakan penggunaan memori yang besar karena penciptaan objek yang banyak. Waktu dan visited nodes yang besar ini disebabkan karena UCS berjenis Blind Search, yang dimana melakukan pencarian tanpa adanya guide tambahan untuk menuntun pencariannya tersebut.

Untuk GBFS, dapat dilihat bahwa algoritma tersebut tidak tentu akan memberikan solusi optimal global ataupun memberikan solusi sama sekali. Walaupun memiliki nilai waktu dan nodes visited yang paling kecil, tidak menjamin akan memberikan solusi optimum global. Bisa dilihat pada tiga test cases, GBFS tidak mampu untuk memberikan solusi optimal global dan pada tiga test cases terakhir, GBFS tidak mampu untuk memberikan solusi. Hal ini disebabkan karena sifat GBFS yang tidak mengenal backtracking, sehingga rawan untuk *ter-commit* pada jalan yang salah ataupun mendapati jalan “buntu”.

Untuk A\*, dapat dilihat bahwa algoritma tersebut mampu untuk memberikan solusi optimal global karena heuristik-nya yang *admissible*. Bisa dilihat bahwa A\* seperti UCS yang dioptimasi, dimana A\* memiliki waktu dan visited nodes yang cukup kecil. Hal ini disebabkan oleh penambahan teknik heuristik yang membuat penelusuran A\* lebih “*ter-guide*” dibandingkan penelusuran UCS.

## **BAB VI**

### **PENJELASAN BONUS**

Bonus pada Tugas Kecil 3 ini adalah pembuatan GUI (*Graphical User Interface*) untuk permainan Word Ladder. Di sini, saya membuat GUI dengan menggunakan Java Swing dan Netbeans sebagai IDE-nya. Berikut adalah GUI class pada program.

1. App.java

Window home dari permainan Word Ladder, dimana user dapat menginput kata awal dan kata tujuan, serta algoritma penelusuran yang diinginkan

2. SolutionWindow.java

Berisi window yang menampilkan tahap-tahapan solusi word ladder dengan kata awal dan kata tujuan yang telah diinput sebelumnya. Pada window ini, tampilan tahap solusi dibuat mirip dengan website <https://wordwormdormdork.com/>, Huruf yang sudah sesuai dengan kata tujuan akan diwarnai hijau dan perubahan huruf yang dilakukan user akan ditandai dengan border hitam tebal.

## LAMPIRAN

### 1. Check List

Poin	Ya	Tidak
Program berhasil dijalankan	✓	
Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma UCS	✓	
Solusi yang diberikan pada algoritma UCS optimal	✓	
Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma Greedy Best First Search	✓	
Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma A*	✓	
Solusi yang diberikan pada algoritma A* optimal	✓	
<b>[Bonus]</b> Program memiliki tampilan GUI	✓	

### 2. Pranala Repository

[https://github.com/chankiel/Tucil3\\_13522029](https://github.com/chankiel/Tucil3_13522029)