

Nama : Vincent Hasiholan

NIM : 13518108

1.  $F(n)$  dan  $G(n)$  adalah nilai yang didapatkan dalam penggunaan algoritma UCS, Greedy Best First Search, dan  $A^*$ .  $F(n)$  adalah nilai total yang didapatkan dari penjumlahan nilai  $G(n)$  yang diperlukan dari awal pencarian hingga titik terakhir pencarian.  $G(n)$  adalah nilai yang didapatkan dari titik ke tujuan. Perlu diingatkan bahwa nilai  $G(n)$  ini diakumulasi sebagai penentuan dari  $F(n)$  mana yang memiliki terendah. Ada juga kondisi dimana  $F(n)$  merupakan nilai gabungan dari  $G(n)$  dan  $H(n)$ .  $H(n)$  adalah nilai heuristik yang ditentukan oleh pembuat algoritma untuk menentukan estimasi terdekat dari semua titik ke titik tujuan. Algoritma yang menggunakan  $G(n)$  saja adalah UCS,  $H(n)$  saja adalah Greedy Best First Search, dan gabungan antara  $G(n)$  dengan  $H(n)$  adalah  $A^*$ . Greedy Best First Search tidak menjamin untuk memberikan solusi yang optimal dikarenakan pengguna harus menemukan estimasi yang sesuai untuk seluruh kemungkinan untuk mendapatkan solusi yang optimal.  $H(n)$  yang digunakan admissible dikarenakan setelah program dijalankan, hasil yang diberikan merupakan nilai terendah.
- 2.

No	Algoritma
1	<pre>public List&lt;String&gt; ucs(String startWord, String endWord) {     PriorityQueue&lt;Node&gt; queue = new PriorityQueue&lt;&gt;();     Set&lt;String&gt; visited = new HashSet&lt;&gt;();     Map&lt;String, Integer&gt; pathCost = new HashMap&lt;&gt;();     Map&lt;String, String&gt; parent = new HashMap&lt;&gt;();      queue.offer(new Node(startWord, 0));     pathCost.put(startWord, 0);      while (!queue.isEmpty()) {         Node currentNode = queue.poll();         String currentWord = currentNode.word;         int currentCost = currentNode.cost;          if (currentWord.equals(endWord)) {</pre>

```

        return reconstructPath(startWord,
endWord, parent);
    }

    visited.add(currentWord);

    for (String neighbor :
graph.get(currentWord)) {
        int newCost = currentCost + 1;
        if (!visited.contains(neighbor) &&
(!pathCost.containsKey(neighbor) || newCost <
pathCost.get(neighbor))) {
            pathCost.put(neighbor, newCost);
            parent.put(neighbor, currentWord);
            queue.offer(new Node(neighbor,
newCost));
        }
    }
}

return null;
}

```

2

```

public List<String> aStar(String startWord, String
endWord) {
    PriorityQueue<Node> queue = new
PriorityQueue<>();
    Set<String> visited = new HashSet<>();
    Map<String, Integer> pathCost = new HashMap<>();
    Map<String, String> parent = new HashMap<>();

    queue.offer(new Node(startWord, 0 +
heuristic(startWord, endWord)));
    pathCost.put(startWord, 0);

    while (!queue.isEmpty()) {
        Node currentNode = queue.poll();
        String currentWord = currentNode.word;
        int currentCost = currentNode.cost;

        if (currentWord.equals(endWord)) {

```

```

        return reconstructPath(startWord,
endWord, parent);
    }

    visited.add(currentWord);

    for (String neighbor :
graph.get(currentWord)) {
        int newCost = currentCost + 1;
        if (!visited.contains(neighbor) &&
(!pathCost.containsKey(neighbor) || newCost <
pathCost.get(neighbor))) {
            pathCost.put(neighbor, newCost);
            parent.put(neighbor, currentWord);
            queue.offer(new Node(neighbor,
newCost + heuristic(neighbor, endWord)));
        }
    }

    return null;
}

```

3

```

public List<String> greedyBestFirstSearch(String
startWord, String endWord) {
    PriorityQueue<Node> queue = new
PriorityQueue<>(Comparator.comparingInt(n ->
heuristic(n.word, endWord)));
    Set<String> visited = new HashSet<>();
    Map<String, String> parent = new HashMap<>();

    queue.offer(new Node(startWord, 0));

    while (!queue.isEmpty()) {
        Node currentNode = queue.poll();
        String currentWord = currentNode.word;

        if (currentWord.equals(endWord)) {
            return reconstructPath(startWord,
endWord, parent);

```

```

    }

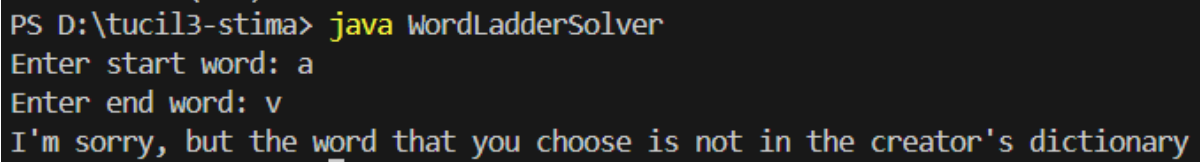
    visited.add(currentWord);

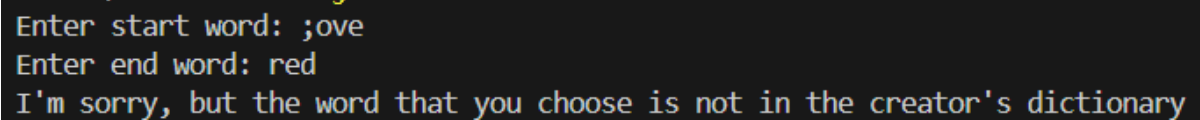
    for (String neighbor :
graph.get(currentWord)) {
        if (!visited.contains(neighbor)) {
            parent.put(neighbor, currentWord);
            queue.offer(new Node(neighbor,
heuristic(neighbor, endWord)));
        }
    }
}

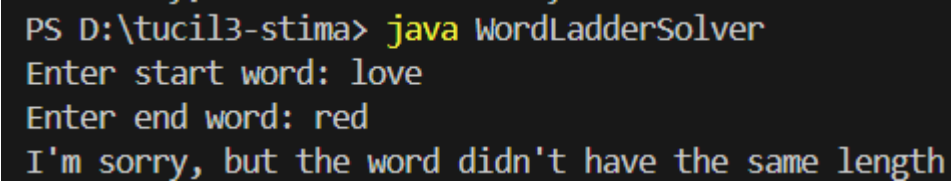
return null;
}

```

### 3. Screenshot

a. 

b. 

c. 

```

PS D:\tucil3-stima> java WordLadderSolver
Enter start word: life
Enter end word: live
UCS:
Shortest path: [LIFE, LIVE]
Time taken (UCS): 825
Greedy Best First Search:
Shortest path: [LIFE, LIVE]
Time taken (GBFS): 6131
A*:
Shortest path: [LIFE, LIVE]
Time taken (A*): 88

```

d.

```

Enter start word: power
Enter end word: money
UCS:
No path found.
Time taken (UCS): 802
Greedy Best First Search:
No path found.
Time taken (GBFS): 5448
A*:
No path found.
Time taken (A*): 61

```

e.

```

PS D:\tucil3-stima> java WordLadderSolver
Enter start word: absolute
Enter end word: obsolete
UCS:
No path found.
Time taken (UCS): 798
Greedy Best First Search:
No path found.
Time taken (GBFS): 3722
A*:
No path found.
Time taken (A*): 85

```

f.

#### 4. Hasil

Dari hasil ss yang diberikan, A\* memberikan hasil yang sangat baik walau tidak ada path yang terbentuk. Hal ini disebabkan karena dictionary yang dibuat masih tidak

memiliki word yang dapat diubah dari start word ke next word hingga pada akhirnya sampai ke end word. Program hanya menampilkan berapa lama proses dari algoritma yang dibangun

5. Github : <https://github.com/Neptune3X/tucil3-stima>