Laporan Tugas Kecil 3 IF2211 Strategi Algoritma
Semester II tahun 2023/2024

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

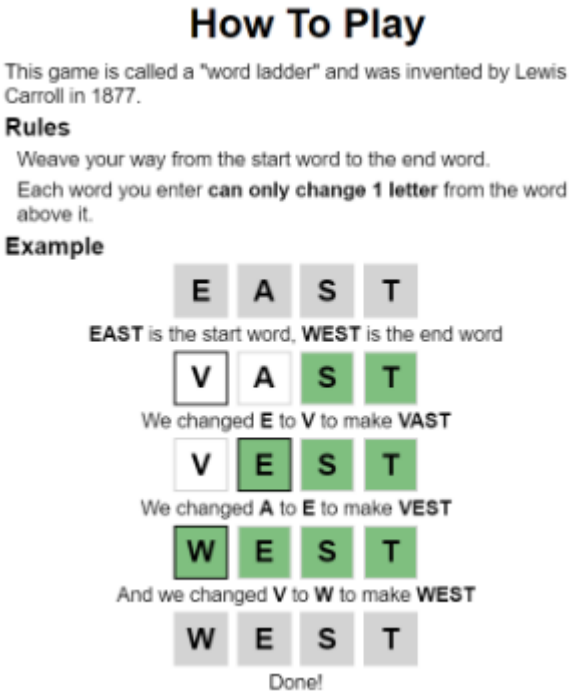
Daniel Mulia Putra Manurung – 13522043

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG 20

DAFTAR ISI

BAB I : DESKRIPSI MASALAH

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 permainan ini, diharapkan solusi optimal, yaitu solusi yang meminimalkan banyaknya kata yang dimasukkan pada rantai kata. Berikut adalah ilustrasi serta aturan permainan.



Gambar 1. Ilustrasi dan Peraturan Permainan Word Ladder
(Sumber: https://wordwormdormdork.com/)

Permainannya cukup sederhana bukan? Jika belum paham dengan peraturan permainannya, cobalah untuk memainkan permainannya pada link sumber di atas. Jika sudah paham dengan permainannya, sekarang adalah waktunya kalian untuk membuat sebuah solver permainan tersebut dengan harapan kita dapat menemukan solusi paling optimal untuk menyelesaikan permainan Word Ladder ini.

BAB 2 : ALGORITMA UCS, GBFS, dan A* Untuk Pencarian Solusi Permainan Word Ladder

1. UCS

Pendekatan pertama menggunakan Algoritma Uniform Cost Search. Dalam kasus Word Ladder, algoritma cost value (**g(n)**) dari algoritma UCS adalah jumlah node yang dilalui dalam sebuah path. Untuk setiap pertambahan node pada sebuah path, cost dari sebuah path akan bertambah **satu**. Dalam kasus ini, saya akan menerapkan path dalam bentuk node, yang memiliki atribut path dan cost. Penambahan nilai g(n) sebanyak satu akan disamakan untuk setiap perubahan kata, hal ini dikarenakan tidak terdapat nilai cost yang pasti dan dapat ditentukan antara sebuah kata dan kata lainnya.

Dengan penambahan cost yang linear dan sama untuk setiap pertambahan node dalam sebuah path, algoritma UCS dalam kasus Word Ladder akan sama dengan algoritma BFS. Hal ini dikarenakan priority queue yang digunakan akan selalu menghasilkan path (dalam algoritma saya direpresentasikan dengan node) baru yang pasti akan memiliki cost lebih tinggi atau sama dengan cost dari path lain dalam priority queue. Sebagai contoh, node pertama akan memiliki 10 kata lain yang akan dimasukkan dalam priority queue, yang mana tiap dari kata tersebut akan memiliki cost 1. Untuk pengulangan selanjutnya, tiap kata dalam queue tersebut akan menghasilkan path lain yang akan memiliki cost 2, dan seterusnya, sehingga algoritma ini akan sama dengan BFS.

Secara keseluruhan cara kerja algoritma UCS saya adalah sebagai berikut:

- Pada awalnya terdapat priority queue yang berisi startword yang memiliki path dirinya sendiri dan cost 0, dan sebuah hashlist untuk melakukan track node yang telah dikunjungi.
- Akan diambil elemen pertama dari priority queue, dan hingga priority queue habis (tak berisi), akan dilakukan pengecekan. Pertama, apakah elemen yang diambil adalah elemen goal, jika ya, akan dikembalikan path dari node tersebut. Kedua, jika word dari node itu pernah dikunjungi, yang mana jika dikunjungi maka telah terdapat path yang lebih singkat, maka akan diabaikan. Jika belum pernah dikunjungi, maka akan dicari seluruh kata yang memiliki perbedaan 1 dari kata sekarang.
- Untuk setiap kata yang didapatkan (kata tetangga dari kata awal), jika kata tersebut belum pernah dikunjungi sebelumnya, maka, akan ditambahkan node tersebut, beserta pathnya dalam priority queue, dan akan diurutkan kembali.
- Hingga didapatkan solusi, atau priority queue habis (tidak ada solusi), perulangan ini akan selalu dilakukan.

Algoritma ini memiliki kompleksitas $O(b^d)$ dengan b branching factor atau jumlah node tetangga, dan d adalah kedalaman end word.

2. GBFS

Pendekatan kedua menggunakan Algoritma Greedy Best First Search. Dalam kasus Word Ladder, algoritma heuristic value (**h(n)**) dari algoritma GBFS adalah jumlah kata yang berbeda dari kata yang didapatkan. Nilai heuristik ini adalah **admissible**. Syarat dari sebuah nilai heuristik yang admissible adalah nilai tersebut tidak boleh overestimates atau melebihi nilai cost lainnya. Dalam kasus Word Ladder, untuk mencapai sebuah endword, diperlukan setidaknya perubahan node sebanyak jumlah huruf yang berbeda (h(n)). Maka, dengan demikian tidak terdapat skenario di mana jumlah perubahan yang dilakukan lebih sedikit dari nilai heuristik. Maka dapat dikatakan bahwa (**g(n)**) >= (**h(n)**). Maka nilai heuristik ini adalah admissible.

Dalam algoritma GBFS ini, backtracking tidak akan diterapkan. Maka untuk itu

solusi yang diberikan oleh algoritma ini sangat memungkinkan untuk **TIDAK OPTIMAL**, atau bahkan tidak menemukan solusi.

Secara keseluruhan cara kerja algoritma GBFS saya adalah sebagai berikut:

- Pada awalnya terdapat priority queue yang berisi startword yang memiliki nilai herusiticnya sendiri, dan sebuah hashlist untuk melakukan track node yang telah dikunjungi. Priority queue ini akan mengurutkan elemennya berdasarkan nilai heuristiknya mulai dari yang terkecil hingga terbesar.
- Akan diambil elemen terdepan dalam priority queue, dan hingga priority queue tak berisi, akan dilakukan pengecekan. Pertama, apakah elemen yang diambil adalah elemen goal, jika ya, akan dikembalikan path dari node tersebut. Jika kata baru ini belum pernah dikunjungi sebelumnya, maka akan dilakukan pengecekan terhadap tetangganya.
- Untuk setiap kata tetangga, jika kata tersebut belum pernah dikunjungi, maka akan ditambahkan dalam priority queue, jika pernah, maka tidak akan ditambahkan.
- Untuk setiap pengambilan elemen dalam priority queue, priority queue akan dibersihkan, hal ini dikarenakan backtracking tidak dapat dilakukan dalam versi algoritma ini.
- Hingga didapatkan solusi, atau priority queue habis (tidak ada solusi), perulangan ini akan terus dilakukan

Algoritma ini memiliki kompleksitas $O(b^d)$ dengan b branching factor atau jumlah node tetangga, dan d adalah kedalaman tertinggi dari search space word pool.

3. A*

Pendekatan ketiga menggunakan Algoritma A*. Dalam kasus Word Ladder algoritma (**f(n)**) adalah gabungan dari cost value algoritma sebelumnya, yaitu jumlah node yang telah dilalui (**g(n)**), dan jumlah perbedaan huruf dalam kata tersebut (**f(n)**).

Secara keseluruhan cara kerja algoritma AStar saya adalah sebagai berikut:

- Pada awalnya terdapat priority queue yang berisi startword yang memiliki nilai heuristicnya sendiri, dan sebuah hashlist untuk melakukan track node yang telah dikunjungi. Priority queue ini akan mengurutkan elemennya berdasarkan nilai cost ditambah nilai heuristiknya mulai dari yang terkecil hingga yang terbesar.
- Akan diambil elemen terdepan dari priority queue, dan hingga priority queue tak berisi, akan dilakukan pengecekan. Pertama, jika kata yang didapatkan adalah endWord, maka path dari node akan dikembalikan sebagai hasil akhir. Kedua, jika kata yang didapatkan belum pernah dikunjungi sebelumnya atau nilai cost **f(n)** lebih besar atau sama dengan cost dari kata yang pernah ditemukan sebelumnya, maka akan dilakukan pencarian tetangga dari node tersebut, yaitu kata yang memiliki perbedaan 1 huruf dari kata sekarang.
- Untuk setiap kata yang didapatkan (kata tetangga dari kata awal), jika kata tersebut belum pernah dikunjungi sebelumnya, maka, akan ditambahkan node tersebut, beserta pathnya dalam priority queue, dan akan diurutkan kembali berdasarkan nilai **f(n)** nya.
- Hingga didapatkan solusi, atau priority queue habis (tidak ada solusi), perulangan ini akan selalu dilakukan.

Algoritma ini memiliki kompleksitas $O(b^d)$ dengan b branching factor atau jumlah node tetangga, dan d adalah kedalaman terendah end word. Tetapi dengan bantuan nilai heuristik, algoritma ini akan mendapatkan solusi yang lebih baik.

BAB 3 : IMPLEMENTASI PROGRAM DENGAN BAHASA JAVA

1. Fungsi Pembantu

Dalam program java saya, terdapat beberapa fungsi pembantu yang akan digunakan di beberapa algoritma. Fungsi ini adalah fungsi yang lebih general dan tidak terikat spesifik pada satu algoritma saja.

```java
public static Set<String> loadWordPool(String filePath) {
    Set<String> wordPool = new HashSet<>();
    try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
        String line;
        while ((line = br.readLine()) != null) {
            wordPool.add(line.trim());
        }
    } catch (IOException e) {
        // e.printStackace();
    }
    return wordPool;
}
```

Fungsi **loadWordPool** adalah fungsi untuk melakukan load dari file txt yang berisi kata kata yang termasuk kata bahasa inggris yang akan digunakan dalam pencarian nantinya. **loadWordPool** digunakan dalam algoritma **UCS, GBFS, dan AStar**.

```java
public static Set<String> getNeighbors(String word, Set<String> wordPool) {
    Set<String> neighbors = new HashSet<>();
    char[] chars = word.toCharArray();

    for (int i = 0; i < chars.length; i++) {
        char ori = chars[i];
        for (char c = 'a'; c <= 'z'; c++) {
            if (c != ori) {
                chars[i] = c;
                String newW = new String(chars);
                if (wordPool.contains(newW)) {
                    neighbors.add(newW);
                }
            }
        }
        chars[i] = ori;
    }

    return neighbors;
}
```

Fungsi **getNeighbors** adalah fungsi yang digunakan untuk menentukan dan mengembalikan

seluruh kata tetangga dari kata dalam parameter, kata tetangga adalah kata yang hanya memiliki 1 perbedaan huruf dibandingkan kata parameter. **getNeighbors** digunakan dalam algoritma **UCS, GBFS, dan AStar.**

2. **UCS**

```
static class Node {
    String word;
    List<String> path;
    int cost;

    Node(String word, List<String> path, int cost) {
        this.word = word;
        this.path = path;
        this.cost = cost;
    }
}
```

Class **Node** dalam algoritma UCS akan memiliki atribut cost, yang adalah nilai g(n), dan atribut path yang digunakan untuk melakukan track dari node-node yang dikunjungi sebelumnya. Dapat dikatakan bahwa node ini berperan sebagai path juga.

```
public static List<String> uniformCostSearch(Set<String> wordPool, String startWord, String endWord) {
    PriorityQueue<Node> searchQueue = new PriorityQueue<>(Comparator.comparingInt(n -> n.cost));
    Set<String> visitedNode = new HashSet<>();

    searchQueue.add(new Node(startWord, new ArrayList<>(List.of(startWord)), cost:0));

    while (!searchQueue.isEmpty()) {
        Node current = searchQueue.poll();

        if (current.word.equals(endWord)) {
            return current.path;
        }

        if (!visitedNode.contains(current.word)) {
            visitedNode.add(current.word);

            for (String neighbor : getNeighbors(current.word, wordPool)) {
                if (!visitedNode.contains(neighbor)) {
                    List<String> newPath = new ArrayList<>(current.path);
                    newPath.add(neighbor);
                    searchQueue.add(new Node(neighbor, newPath, current.cost + 1));
                }
            }
        }
    }

    return null;
}
```

fungsi **uniformCostSearch** adalah fungsi utama UCS. Fungsi ini memiliki beberapa parameter yaitu wordPool yang adalah kumpulan kata - kata yang dapat menjadi node, startWord yang adalah kata awal, dan endWord yang adalah kata tujuan. Fungsi ini akan menjalankan algoritma UCS sesuai dengan penjelasan pada BAB II.

- Pada awalnya terdapat priority queue yang berisi startword yang memiliki path dirinya sendiri dan cost 0, dan sebuah hashlist untuk melakukan track node yang telah dikunjungi.
- Akan diambil elemen pertama dari priority queue, dan hingga priority queue habis (tak berisi), akan dilakukan pengecekan. Pertama, apakah elemen yang diambil adalah elemen goal, jika ya, akan dikembalikan path dari node tersebut. Kedua, jika word dari node itu pernah dikunjungi, yang mana jika dikunjungi maka telah terdapat path yang lebih singkat, maka akan diabaikan. Jika belum pernah dikunjungi, maka akan dicari seluruh kata yang memiliki perbedaan 1 dari kata sekarang.
- Untuk setiap kata yang didapatkan (kata tetangga dari kata awal), jika kata tersebut belum pernah dikunjungi sebelumnya, maka, akan ditambahkan node tersebut, beserta pathnya dalam priority queue, dan akan diurutkan kembali.
- Hingga didapatkan solusi, atau priority queue habis (tidak ada solusi), perulangan ini akan selalu dilakukan.

3. GBFS

```
static class Node {
    String word;
    int heuristic;

    Node(String word, int heuristic) {
        this.word = word;
        this.heuristic = heuristic;
    }
}
```

Class **Node** dalam algoritma GBFS akan memiliki atribut heuristic, yang adalah nilai h(n). Node pada GBFS tidak memerlukan atribut path, pengembalian nilai rute akan menggunakan fungsi whatsPath.

```
public static List<String> whatsPath(Map<String, String> cameFrom, String current) {
    List<String> path = new ArrayList<>();
    while (current != null) {
        path.add(current);
        current = cameFrom.get(current);
    }
    Collections.reverse(path);
    return path;
}
```

Fungsi **whatsPath** adalah fungsi untuk menentukan path dari hasil akhir GBFS. Parameter fungsi ini adalah sebuah Map yang berisi pasangan parent node dan node tetangga, dan sebuah string current yang awalnya adalah nilai endWord. Fungsi ini akan mengembalikan list string berisi seluruh word yang ditemukan, sesuai dengan urutannya. Urutan path ini akan ditentukan menggunakan map cameFrom. Untuk setiap parent node, akan dicari parent nya hingga didapatkan null.

```java
public static int heuristic(String word1, String word2) {
    int diff = 0;
    for (int i = 0; i < word1.length(); i++) {
        if (word1.charAt(i) != word2.charAt(i)) {
            diff++;
        }
    }
    return diff;
}
```

Fungsi **heuristic** adalah fungsi yang mengembalikan nilai perbedaan huruf antar 2 kata. Dikarenakan nilai heuristik yang akan kita gunakan adalah perbedaan huruf 2 kata, diperlukan fungsi heuristic yang akan menghitung perbedaan kedua kata tersebut.

```java
public static List<String> greedyBestFirstSearch(Set<String> wordPool, String startWord, String goalWord) {
    // Queue<Node> nodeQueue = new LinkedList<>();
    PriorityQueue<Node> nodeQueue = new PriorityQueue<>(Comparator.comparingInt(n -> n.heuristic));
    Set<String> visitedNode = new HashSet<>();
    Map<String, String> nodeOrigin = new HashMap<>();

    nodeQueue.add(new Node(startWord, heuristic(startWord, goalWord)));

    while (!nodeQueue.isEmpty()) {
        Node current = nodeQueue.poll();
        nodeQueue.clear();
        if (current.word.equals(goalWord)) {
            return whatsPath(nodeOrigin, current.word);
        }

        if (!visitedNode.contains(current.word)) {
            visitedNode.add(current.word);
            for (String neighbor : getNeighbors(current.word, wordPool)) {
                if (!visitedNode.contains(neighbor)) {
                    System.out.println(neighbor);
                    nodeOrigin.put(neighbor, current.word);
                    nodeQueue.add(new Node(neighbor, heuristic(neighbor, goalWord)));
                }
            }
        }
    }

    return null;
}
```

Fungsi **greedyBestFirstSearch** adalah fungsi utama dari GBFS. Fungsi ini memiliki parameter wordPool, startWord yang adalah kata awal dan endWord yang adalah kata tujuan. Algoritma GBFS ini tidak menerapkan **backtrack**, sehingga memiliki solusi yang tidak cukup optimal. Algoritma ini akan bekerja seperti penjelasan pada BAB II.

- Pada awalnya terdapat priority queue yang berisi startword yang memiliki nilai herusiticnya sendiri, dan sebuah hashlist untuk melakukan track node yang telah dikunjungi. Priority queue ini akan mengurutkan elemennya berdasarkan nilai heuristiknya mulai dari yang terkecil hingga terbesar.
- Akan diambil elemen terdepan dalam priority queue, dan hingga priority queue tak berisi, akan dilakukan pengecekan. Pertama, apakah elemen yang diambil adalah elemen goal, jika ya, akan dikembalikan path dari node tersebut. Jika kata baru ini belum pernah dikunjungi sebelumnya, maka akan dilakukan pengecekan terhadap tetangganya.

- Untuk setiap kata tetangga, jika kata tersebut belum pernah dikunjungi, maka akan ditambahkan dalam priority queue, jika pernah, maka tidak akan ditambahkan.
- Untuk setiap pengambilan elemen dalam priority queue, priority queue akan dibersihkan, hal ini dikarenakan backtracking tidak dapat dilakukan dalam versi algoritma ini.
- Hingga didapatkan solusi, atau priority queue habis (tidak ada solusi), perulangan ini akan terus dilakukan

4. A*

```
static class Node {
    String word;
    List<String> path;
    int g;
    int h;
    int f;

    Node(String word, List<String> path, int g, int h) {
        this.word = word;
        this.path = path;
        this.g = g;
        this.h = h;
        this.f = g + h;
    }
}
```

Class **Node** dalam algoritma A* akan memiliki atribut cost, yang adalah nilai g(n), atribut path yang digunakan untuk melakukan track dari node-node yang dikunjungi sebelumnya, atribut h yang adalah nilai heuristik h(n), dan atribut f yang adalah nilai f(n) = g(n) + h(n). Dapat dilihat bahwa atribut f adalah hasil penjumlahan atribut g dan h.

```java
public static List<String> aStarSearch(Set<String> wordPool, String startWord, String endWord) {
    PriorityQueue<Node> searchQueue = new PriorityQueue<>(Comparator.comparingInt(n -> n.f));
    Map<String, Integer> nodeCosts = new HashMap<>();
    Set<String> visitedNode = new HashSet<>();

    int heuristic = heuristicss(startWord, endWord);
    searchQueue.add(new Node(startWord, new ArrayList<>(List.of(startWord)), g:0, heuristic));

    while (!searchQueue.isEmpty()) {
        Node current = searchQueue.poll();

        if (current.word.equals(endWord)) {
            return current.path;
        }

        int currentCost = current.f;

        if (!visitedNode.contains(current.word)
                || currentCost < nodeCosts.getOrDefault(current.word, defaultValue:999999)) {
            visitedNode.add(current.word);
            nodeCosts.put(current.word, currentCost);

            for (String neighbor : getNeighbors(current.word, wordPool)) {
                int newCost = currentCost + 1;
                int newHeuristic = heuristicss(neighbor, endWord);
                List<String> newPath = new ArrayList<>(current.path);
                newPath.add(neighbor);

                if (!nodeCosts.containsKey(neighbor) || newCost < nodeCosts.get(neighbor)) {
                    searchQueue.add(new Node(neighbor, newPath, newCost, newHeuristic));
                }
            }
        }
    }

    return null;
}
```

Fungsi **aStarSearch** adalah fungsi utama algoritma A*. Sama seperti fungsi algoritma lainnya, fungsi ini memiliki parameter startWord, endWord dan wordPool sebagai seluruh kemungkinan kata yang dapat digunakan dalam pencarian. Cara kerja algoritma ini adalah sama dengan penjelasan pada BAB II.

- Pada awalnya terdapat priority queue yang berisi startword yang memiliki nilai heuristicnya sendiri, dan sebuah hashlist untuk melakukan track node yang telah dikunjungi. Priority queue ini akan mengurutkan elemennya berdasarkan nilai cost ditambah nilai heuristiknya mulai dari yang terkecil hingga yang terbesar.
- Akan diambil elemen terdepan dari priority queue, dan hingga priority queue tak berisi, akan dilakukan pengecekan. Pertama, jika kata yang didapatkan adalah endWord, maka path dari node akan dikembalikan sebagai hasil akhir. Kedua, jika kata yang didapatkan belum pernah dikunjungi sebelumnya atau nilai cost **f(n)** lebih besar atau sama dengan cost dari kata yang pernah ditemukan sebelumnya, maka akan dilakukan pencarian tetangga dari node tersebut, yaitu kata yang memiliki perbedaan 1 huruf dari kata sekarang.
- Untuk setiap kata yang didapatkan (kata tetangga dari kata awal), jika kata tersebut belum pernah dikunjungi sebelumnya, maka, akan ditambahkan node tersebut, beserta pathnya dalam priority queue, dan akan diurutkan kembali berdasarkan nilai **f(n)** nya.

- Hingga didapatkan solusi, atau priority queue habis (tidak ada solusi), perulangan ini akan selalu dilakukan.

BAB 4: EKSPERIMEN

UCS

1. **TC 1**

```
Enter start word: duck
Enter end word: buck
Word ladder found:
duck
buck
Word ladder found with a path of 2 step.
Total nodes visited: 156
Time elapsed: 4 ms
Memory used: 11410 Kilo bytes
```

2. **TC 2**

```
Enter start word: acorn
Enter end word: nomad
Word ladder found:
acorn
scorn
shorn
shore
shote
shots
soots
sorts
soras
somas
nomas
nomad
Word ladder found with a path of 12 step.
Total nodes visited: 21309
Time elapsed: 59 ms
Memory used: 50228 Kilo bytes
```

3. **TC 3**

```
Enter start word: glint
Enter end word: mouse
Word ladder found:
glint
flint
feint
feist
heist
hoist
hoise
house
mouse
Word ladder found with a path of 9 step.
Total nodes visited: 10427
Time elapsed: 17 ms
Memory used: 79059 Kilo bytes
```

4. **TC 4**

```
Enter start word: atlases
Enter end word: cabaret
Word ladder found:
atlases
anlases
anlaces
unlaces
unlaced
unladed
unfaded
unfaked
capered
tapered
tabered
tabored
taboret
tabaret
cabaret
Word ladder found with a path of 53 step.
Total nodes visited: 18108
Time elapsed: 93 ms
Memory used: 121005 Kilo bytes
```

5. **TC 5**

```
Enter start word: nomad
Enter end word: staff
Word ladder found:
nomad
nomas
somas
soras
sorts
soots
scots
scats
scars
scarf
scurf
scuff
stuff
staff
Word ladder found with a path of 14 step.
Total nodes visited: 24448
Time elapsed: 54 ms
Memory used: 67923 Kilo bytes
```

6. **TC 6**

```
Enter start word: ashen
Enter end word: paper
Word ladder found:
ashen
aspen
asper
aster
ester
eater
pater
paper
Word ladder found with a path of 8 step.
Total nodes visited: 7533
Time elapsed: 9 ms
Memory used: 85747 Kilo bytes
```

GBFS

1. **TC1**

```
Enter start word: duck
Enter end word: buck
Word ladder found:
duck
buck
Word ladder found with a path of 2 step.
Total nodes visited: 20
Time elapsed: 5 ms
Memory used: 11410 Kilo bytes
```

2. **TC2**

```
Enter start word: acorn
Enter end word: nomad
No word ladder found.
```

3. **TC3**

```
Enter start word: glint
Enter end word: mouse
No word ladder found.
```

4. **TC4**

```
Enter start word: atlases
Enter end word: cabaret
No word ladder found.
```

5. **TC5**

```
Enter start word: nomad
Enter end word: staff
No word ladder found.
```

6. **TC6**

```
Enter start word: ashen
Enter end word: paper
Word ladder found:
ashen
aspen
asper
aster
ester
eater
pater
paper
Word ladder found with a path of 8 step.
Total nodes visited: 51
Time elapsed: 0 ms
Memory used: 12333 Kilo bytes
```

A*

1. **TC 1**

```
Enter start word: duck
Enter end word: buck
Word ladder found:
duck
buck
Word ladder found with a path of 2 step.
Total nodes visited: 20
Time elapsed: 4 ms
Memory used: 11410 Kilo bytes
```

2. **TC 2**

```
Enter start word: acorn
Enter end word: nomad
Word ladder found:
acorn
scorn
score
scope
slope
slops
clops
coops
comps
comas
nomas
nomad
Word ladder found with a path of 12 step.
Total nodes visited: 2081
Time elapsed: 10 ms
Memory used: 18163 Kilo bytes
```

3. **TC 3**

```
Enter start word: glint
Enter end word: mouse
Word ladder found:
glint
flint
feint
feist
foist
hoist
hoise
house
mouse
Word ladder found with a path of 9 step.
Total nodes visited: 107
Time elapsed: 1 ms
Memory used: 18624 Kilo bytes
```

4. **TC 4**

```
Enter start word: atlases
Enter end word: cabaret
Word ladder found:
atlases
anlases
anlaces
unlaces
unlaced
unladed
catered
capered
tapered
tabered
tabored
taboret
tabaret
cabaret
Word ladder found with a path of 53 step.
Total nodes visited: 18055
Time elapsed: 105 ms
Memory used: 135614 Kilo bytes
```

5. **TC 5**

```
Enter start word: nomad
Enter end word: staff
Word ladder found:
nomad
nomas
somas
soras
sorts
soots
scots
scats
scars
scarf
scurf
scuff
stuff
staff
Word ladder found with a path of 14 step.
Total nodes visited: 20918
Time elapsed: 45 ms
Memory used: 57739 Kilo bytes
```

6. **TC 6**

```
Enter start word: ashen
Enter end word: paper
Word ladder found:
ashen
aspen
asper
aster
ester
eater
pater
paper
Word ladder found with a path of 8 step.
Total nodes visited: 141
Time elapsed: 0 ms
Memory used: 57739 Kilo bytes
```

BAB 5: ANALISIS

Berdasarkan hasil eksperimen yang telah dilakukan, terdapat beberapa perbedaan antara ketiga algoritma pathfinding yang kita gunakan. Terdapat perbedaan signifikan antara algoritma UCS dan A* terhadap algoritma GBFS, selain itu, terdapat juga beberapa perbedaan minor antara algoritma UCS terhadap algoritma A*.

Perbedaan signifikan dapat dilihat bahwa dari ke 6 test case yang digunakan, algoritma GBFS tidak dapat menyelesaikan 4 dari 6 test case yang diberikan, sedangkan algoritma UCS dan A* berhasil mendapatkan solusi. Tidak hanya itu, pada 2 test case yang berhasil diselesaikan oleh GBFS, pada test case 1, GBFS benar mendapatkan solusi optimal, tetapi dapat dilihat node yang didapatkan relatif sedikit dibandingkan UCS. pada test case 6, GBFS hanya melakukan visit terhadap 51 node, dibandingkan UCS yang melakukan visit terhadap 7533 node. Kedua hal ini dikarenakan algoritma GBFS tidak melakukan backtrack. Tidak melakukan backtrack menyebabkan algoritma GBFS akan selalu maju dan mengambil node tetangga dengan nilai heuristik yang paling kecil. Hal ini dapat menyebabkan masuknya path ke kata - kata yang berpotensi tidak memungkinkan mencapai end word, dan setelah memasuki path itu, algoritma tidak dapat melakukan backtrack untuk mendapatkan solusi yang memungkinkan lainnya. Hal ini juga didukung oleh jumlah node yang divisit. Ketidakmampuan algoritma GBFS untuk melakukan backtrack ke node sebelumnya akan sangat membatasi node yang dapat diexplore oleh algoritma ini, sehingga untuk itu algoritma ini menjadi sangat tidak optimal dan tidak efisien dalam kasus rata - rata. Kompleksitas algoritma GBFS.

Perbedaan selanjutnya adalah antara algoritma UCS dan algoritma A*. Hal ini dapat dilihat dalam hampir seluruh test case, waktu yang diperlukan dan node yang dikunjungi oleh algoritma A* relatif lebih singkat dan lebih kecil, padahal kedua algoritma menghasilkan solusi yang sama - sama optimal. Hal ini dikarenakan algoritma A* mendapatkan bantuan dari nilai heuristik, yang memungkinkan algoritma ini untuk lebih mudah mendekat ke solusi akhir. Dengan ini kompleksitas waktu yang diperlukan oleh algoritma A* dapat berpotensi menjadi lebih singkat. Sehingga dapat dikatakan bahwa algoritma A* **lebih efisien** dibandingkan algoritma UCS.

LAMPIRAN

Repository Github

https://github.com/Gryphuss/Tucil3_13522043

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