NAMA : Renaldy Arief Susanto
NIM : 13522022

LAPORAN TUGAS KECIL 3 IF2211 STRATEGI ALGORITMA

A. Deskripsi Permasalahan

"Word Ladder" adalah sebuah permainan teka-teki pemain tunggal yang menantang pemain untuk mengubah sebuah kata menjadi sebuah kata lain menggunakan langkah-langkah. Pada permainan ini, sebuah langkah didefinisikan sebagai mengubah satu huruf pada kata yang sekarang. Setiap langkah perantara harus menghasilkan kata yang valid dan tujuannya adalah mencapai kata target dalam jumlah langkah yang sekecil-kecilnya. Kata awal dan kata target harus mempunyai panjang yang sama. Permainan ini menguji kemampuan kosakata dan dapat menjadi hiburan sekaligus tantangan yang menarik.

Contoh ilustrasi kasus adalah sebagai berikut. Diberikan dua buah kata (dalam Bahasa Inggris), yaitu *wait* dan *done*. Pemain memulai permainan dari kata *wait* dan berusaha melakukan langkah-langkah hingga mencapai kata *done*. Contoh urutan langkah yang dapat dimainkan pemain adalah seperti ini.

1. Langkah pertama, pemain mengubah *wait* menjadi *want*
2. Langkah kedua, pemain mengubah *want* menjadi *wont*
3. Langkah ketiga, pemain mengubah *wont* menjadi *dont*
4. Langkah keempat, pemain mengubah *dont* menjadi *done*. *Done* adalah kata target sehingga pemain sudah memenangkan permainan dalam empat langkah.

Mahasiswa ditugaskan untuk membuat program sederhana dalam bahasa Java yang mengimplementasikan Algoritme Uniform Cost Search (UCS), Greedy Best First Search (GBFS), dan A* untuk mencari solusi paling optimal permainan *word ladder*.

B. Implementasi Algoritme UCS

Algoritme Uniform Cost Search adalah algoritme yang dapat digunakan untuk mencari jalur terpendek dalam sebuah graf berbobot. Algoritme ini intinya adalah BFS yang dimodifikasi sehingga dapat menangani graf berbobot. Dalam graf yang tidak berbobot, BFS dapat mencari jalur terpendek karena semua jarak antarsimpul dianggap sama, yaitu satu. Untuk graf berbobot, BFS tidak bisa seenaknya digunakan (dalam konteks solusi yang diinginkan harus optimal) karena sekarang, yang menjadi pertimbangan jarak antarsimpul adalah bobot atau *cost* jalur yang bervariasi. Oleh karena itu, jalur penulusuran perlu disesuaikan agar penjelajahan simpul tetap terjadi secara terurut berdasarkan dekatnya dengan simpul sumber terlebih dahulu. Inilah logika utama dari Uniform Cost Search.

Dari deskripsi tersebut, disimpulkan bahwa algoritme UCS sangat kuat karena jika digunakan *priority queue*, solusi yang optimal dapat dihasilkan dalam waktu yang *loglinear* terhadap jumlah simpul dan tepi untuk graf berbobot. Namun, rupanya, UCS terlalu *overkill* untuk menyelesaikan permainan *word ladder*. Dalam *word ladder*, setiap kata yang valid direpresentasikan sebagai sebuah simpul. Dua kata dihubungkan oleh sebuah *edge* apabila huruf-hurufnya berbeda di tepat satu posisi. Ini artinya pemain bisa menggunakan sebuah langkah untuk bergerak antara dua kata tersebut. Tidak ada perbedaan banyaknya langkah untuk dua kata apa pun selama syarat di atas terpenuhi. Lantas, permasalahan ini dapat dimodelkan dengan **graf tidak berbobot**. Oleh karena itu, penggunaan UCS untuk pencarian jarak terdekat antara dua kata dapat digantikan dengan BFS karena secara efektif, **kedua algoritme tersebut akan melakukan hal yang sama persis**.

Berikut adalah rincian proses algoritme BFS sekaligus UCS untuk permasalahan ini.

1. Kata awal dijadikan simpul hidup dan dimasukkan ke dalam antrean. Catat jaraknya sebagai 0. Inisialisasi juga sebuah *map* yang akan memetakan sebuah simpul ke simpul induknya (parent node).
2. Ambil kata terdepan dalam antrean beserta jaraknya. Jika kata ini adalah kata target, algoritme selesai dan jalur kata dapat ditelusuri balik hingga kata awal (langkah 5). Jika bukan, algoritme berlanjut.
3. Masukkan semua kata valid yang bisa dicapai dari kata ini ke dalam antrean, kecuali sudah pernah ditelusuri. Catat jarak dari simpul-simpul baru sebagai

jarak dari simpul ini ditambah satu. Tandai pula kata ini sebagai simpul induk dari kata-kata baru yang telah dimasukkan ke *queue*.

4. Ulangi langkah 2-3 hingga kata terdepan adalah kata target.
5. Ketika sudah menemukan kata target, traversal *map* dari simpul akhir ke simpul awal untuk mengonstruksi ulang jalur solusi.

C. Implementasi Algoritme Greedy Best First Search

Telah dijelaskan pada bagian B bahwa UCS adalah sebuah algoritme yang menelusuri simpul graf secara terurut berdasarkan jaraknya dari simpul asal. Algoritme Greedy Best First Search atau GBFS sangat mirip dengan UCS. Perbedaan hanya terletak pada pengurutan simpul hidup. Untuk GBFS, didefinisikan sebuah fungsi heuristik ***f(n)***, yaitu fungsi untuk menghitung estimasi jarak terdekat dari sebuah simpul ke simpul tujuan. Kemudian, antrean simpul hidup akan terurut berdasarkan nilai fungsi tersebut. Langkah-langkah GBFS pada permasalahan ini sama persis dengan langkah-langkah UCS, hanya saja simpul diekspan berdasarkan urutan nilai ***f(n)*** nya.

Fungsi heuristik yang saya pilih adalah ***f(n)*** = **banyaknya huruf yang berbeda antara kata *n* dengan kata tujuan.** Tentunya nilai f(n) ini dapat dihitung dengan traversal string. Jelas bahwa algoritme ini tidak akan menghasilkan solusi yang optimal akibat fungsi heuristik yang tidak menjamin hasilnya akan sama dengan jarak yang sebenarnya.

Perhatikan bahwa, dalam implementasi aslinya, GBFS pada dasarnya tidak senantiasa menyimpan simpul hidup, tetapi hanya memilih satu dari semua simpul hidup yang terbaik pada saat itu, lalu mengekspan simpul itu saja. Artinya, semestinya bisa saja GBFS tidak menemukan solusi karena tidak semua jalur dicoba. Namun, dalam implementasi saya, saya tetap menyimpan simpul hidup sehingga GBFS akan selalu menghasilkan solusi. Saya melakukan ini agar bisa menunjukkan bahwa pilihan simpul pada GBFS sangat mungkin bukan bagian dari solusi optimal. Di samping itu, saya telah mencoba skema GBFS yang sesungguhnya dan banyak kasus yang tidak menghasilkan solusi. Hal tersebut tidak menarik sehingga saya memilih untuk merubahnya agar selalu menghasilkan solusi.

D. Implementasi Algoritme A* (A-Star)

Algorime A* adalah penggabungan antara algoritme UCS dan GBFS. Fungsi heuristik pada GBFS digabungkan dengan jarak simpul ke simpul awal pada UCS untuk menghasilkan fungsi evaluasi $f(n) = g(n) + h(n)$ yang digunakan sebagai penentu nilai sebuah simpul dalam pengurutannya.

- $g(n) =$ fungsi jarak yang memetakan jarak kata awal ke kata n
- $h(n) =$ fungsi heuristik, dalam implementasi ini banyaknya huruf yang berbeda antara kata n dengan kata tujuan
- $f(n) =$ estimasi jarak total dari kata awal ke kata tujuan dengan melalui n

Algoritme A* bersifat heuristik dan solusi yang dihasilkan akan bersifat optimal atau tidak, bergantung pada fungsi heuristik yang dipilih. Lebih spesifiknya, jika fungsi heuristik bersifat *admissible* atau selalu meremehkan biaya sebenarnya ($h(n)$ adalah lebih kecil dari $h^*(n)$), maka A* dijamin menemukan solusi optimal. Pada implementasi ini, fungsi heuristik yang saya pilih dipastikan meremehkan biaya sebenarnya. Sebuah kata dengan panjang s yang mempunyai sejumlah m huruf yang tidak sama dengan kata tujuan sudah pasti memerlukan setidaknya $s - m$ langkah untuk sampai pada kata tujuan. Oleh karena itu, funsgi A* yang saya implementasikan ***admissible*** **dan pasti menghasilkan solusi yang optimal.**

Untuk langkah-langkah spesifik algoritme A* adalah sama dengan UCS dengan modifikasi fungsi evaluasi sehingga tidak akan saya jelaskan lagi di sini.

E. Implementasi Program

Program dibuat dalam bahasa Java dengan antarmuka berupa CLI (*Command Line Input*). Program dimulai dengan memuat dataset kata dari sebuah file txt. Jika sudah selesai, pengguna dapat memasukkan (secara berurutan) kata awal, kata akhir, dan pilihan algoritme.

Logika program dibagi menjadi tiga: *Interface* Solver yang akan diimplementasi masing-masing kelas algoritme untuk melakukan pencarian solusi, kelas WordLoader

yang bertanggung jawab memuat isi file, serta Main.java yang berisi *control flow* program. Berikut beberapa detail implementasi yang sekiranya perlu saya jelaskan.

1. Kata-kata *dictionary* dimuat dalam bentuk sebuah *trie* atau *prefix tree*. Struktur data ini mendukung pencarian kata dalam sebuah *dictionary* dalam waktu O(n) dengan n adalah panjang kata. Struktur data ini dipilih karena waktu pengecekan yang sangat cepat dan tidak ada *collision* seperti pada *hash set.*

```
class TrieNode {
    public char val;
    public boolean isWord;
    public TrieNode[] children = new TrieNode[26];
    public TrieNode() {}
    TrieNode(char c){ ···
}

public class Trie {

    private TrieNode root;
    public Trie() { ···

    // (this O(word.length())
    public void insert(String word) { ···

    // (this is also O(word.length()))
    public boolean exists(String word) { ···

}
```

2. Saya membuat sebuah *interface* bernama *Solver* yang mempunyai metode *solve*. Kemudian, saya membuat sebuah kelas untuk masing-masing algoritme yang akan mengimplementasi metode *solve* tersebut.

```
public interface Solver {
    ArrayList<String> solve(String startword, String endword) throws Exception,
    OutOfMemoryError;
    public void setTrie(Trie trie_);
}
```

```java
public class UCSSolver implements Solver {

    private Trie trie;

    public UCSSolver() {}
    public void setTrie(Trie trie_) {
        trie = trie_;
    }

    public ArrayList<String> solve(String startword, String endword) throws Exception,
    OutOfMemoryError {···


}
```

```java
public class GBFSSolver implements Solver {

    private Trie trie;
    public void setTrie(Trie trie_) {
        trie = trie_;
    }

    public ArrayList<String> solve(String startword, String endword) throws Exception,
    OutOfMemoryError {···
}
```

```java
public class AStarSolver implements Solver {

    private Trie trie;
    public void setTrie(Trie trie_) {
        trie = trie_;
    }

    public ArrayList<String> solve(String startword, String endword) throws Exception,
    OutOfMemoryError {···
}
```

3. Algoritme UCS memanfaatkan *queue* biasa untuk menyimpan simpul hidup, sementara GBFS dan A* memanfaatkan *priority queue* dengan fungsi comparator yang merupakan fungsi evaluasi masing-masing algoritme.

NAMA : Renaldy Arief Susanto
NIM : 13522022

```java
public ArrayList<String> solve(String startword, String endword) throws Exception,
OutOfMemoryError {

    Queue<String> queue = new LinkedList<String>();
```

```java
public ArrayList<String> solve(String startword, String endword) throws Exception,
OutOfMemoryError {

    GBFSComp comp = new GBFSComp(endword);
    PriorityQueue<String> pq = new PriorityQueue<String>(comp);
```

```java
public ArrayList<String> solve(String startword, String endword) throws Exception,
OutOfMemoryError {

    AStarComp comp = new AStarComp(endword);
    PriorityQueue<StringDepth> pq = new PriorityQueue<StringDepth>(comp);
```

```java
class GBFSComp implements Comparator<String> {
    private String endstr;
    public GBFSComp(String str){
        endstr = str;
    }

    // implementasi fungsi komparator, jika f(n) adalah fungsi evaluasi,
    // maka compare akan mengembalikan nilai f(str1) - f(str2)
    public int compare(String str1, String str2) {
        int ans1 = 0;
        int ans2 = 0;
        int l = endstr.length();
        for (int i = 0; i < l; i++) {
            if (endstr.charAt(i) != str1.charAt(i)) ans1++;
            if (endstr.charAt(i) != str2.charAt(i)) ans2++;
        }
        return ans1 - ans2;
    }
}
```

NAMA : Renaldy Arief Susanto
NIM : 13522022

```java
class AStarComp implements Comparator<StringDepth> {

    private String endstr;
    public AStarComp(String str) {
        endstr = str;
    }

    public int compare(StringDepth obj1, StringDepth obj2) {

        int depth1 = obj1.getDepth();
        int depth2 = obj2.getDepth();
        int ans1 = depth1;
        int ans2 = depth2;
        String str1 = obj1.getStr();
        String str2 = obj2.getStr();

        int l = endstr.length();
        for (int i = 0; i < l; i++) {
            if (endstr.charAt(i) != str1.charAt(i)) ans1++;
            if (endstr.charAt(i) != str2.charAt(i)) ans2++;
        }
        if (ans1 == ans2) return depth1 - depth2;
        return ans1  - ans2;

    }
}
```

F. Uji Kasus

1. Uji Kasus 1, *Hello* ke *World*. (Input algoritme, output, serta waktu eksekusi dan total simpul yang dijelajahi tertera pada gambar).

```
>> Hello World UCS

----------
1. hello
2. hollo
3. holly
4. wolly
5. wooly
6. woold
7. world
----------
Done! Execution time: 213 ms
Total nodes visited: 4699
```

```
>> Hello World GBFS

----------
1. hello
2. hollo
3. holly
4. wolly
5. wooly
6. woold
7. world
----------
Done! Execution time: 8 ms
Total nodes visited: 9
```

```
>> Hello World AStar

----------
1. hello
2. hollo
3. holly
4. wolly
5. wooly
6. woold
7. world
----------
Done! Execution time: 11 ms
Total nodes visited: 93
```

Pada uji kasus ini, baik algoritme UCS, GBFS, maupun A* menghasilkan solusi optimal.

2. Uji Kasus 2 – *Llama* ke *Horse*

```
>> Llama Horse UCS

----------
1. llama
2. ilama
3. ilima
4. clima
5. clime
6. crime
7. prime
8. prise
9. poise
10. hoise
11. horse
----------
Done! Execution time: 69 ms
Total nodes visited: 4520
>> Llama Horse AStar
```

```
>> Llama Horse GBFS

----------
1. llama
2. ilama
3. ilima
4. clima
5. clime
6. clive
7. clove
8. close
9. crose
10. brose
11. boose
12. hoose
13. horse
----------
Done! Execution time: 1 ms
Total nodes visited: 27
```

```
>> Llama Horse AStar

----------
1. llama
2. ilama
3. ilima
4. clima
5. clime
6. crime
7. prime
8. prise
9. poise
10. hoise
11. horse
----------
Done! Execution time: 4 ms
Total nodes visited: 102
```

Pada uji kasus ini, dapat dilihat kekurangan dari algoritme GBFS, yaitu belum tentu menghasilkan solusi yang optimal. Dapat dilihat juga bahwa Algoritme A* jauh lebih cepat daripada algoritme UCS.

3. Uji Kasus 3 – *Orange* ke *Melody*

```
>> Orange Melody UCS
Solution doesn't exist
Total nodes visited: 20043
Exited after: 611 ms
```

```
>> Orange Melody GBFS
Solution doesn't exist
Total nodes visited: 20043
Exited after: 491 ms
```

```
>> Orange Melody AStar
Solution doesn't exist
Total nodes visited: 20043
Exited after: 756 ms
```

4. Uji Kasus 4 – *Unbaked* ke *Cookies*

```
>> Unbaked Cookies UCS

----------
1. unbaked
2. uncaked
3. uncased
4. uncases
5. uneases
6. ureases
7. creases
8. cresses
9. tresses
10. trasses
11. brasses
12. brasser
13. brasier
14. brakier
15. beakier
16. peakier
17. peckier
18. pockier
19. cockier
20. cockies
21. cookies
----------
Done! Execution time: 65 ms
Total nodes visited: 3752
```

```
>> Unbaked Cookies GBFS

----------
1. unbaked
2. uncaked
3. uncakes
4. uncases
5. uneases
6. ureases
7. creases
8. cresses
9. crosses
10. drosses
11. drowses
12. browses
13. browsed
14. browned
15. crowned
16. crooned
17. crooked
18. croaked
19. cloaked
20. clonked
21. clunked
22. clunker
23. clunter
24. counter
25. courter
26. courier
27. coulier
28. collier
29. collies
30. coolies
31. cookies
----------
Done! Execution time: 4 ms
Total nodes visited: 195
```

```
>> Unbaked Cookies AStar

----------
1. unbaked
2. uncaked
3. uncakes
4. uncases
5. uneases
6. ureases
7. creases
8. cresses
9. tresses
10. trasses
11. brasses
12. brasser
13. brasier
14. brakier
15. beakier
16. peakier
17. peckier
18. pockier
19. cockier
20. cockies
21. cookies
----------
Done! Execution time: 46 ms
Total nodes visited: 1691
```

5. Uji Kasus 5 – *Let* ke *Him*

```
>> Let Him UCS

----------
1. let
2. het
3. hit
4. him
----------
Done! Execution time: 24 ms
Total nodes visited: 861
```

```
>> Let Him GBFS

----------
1. let
2. het
3. hit
4. him
----------
Done! Execution time: 0 ms
Total nodes visited: 3
```

```
>> Let Him AStar

----------
1. let
2. het
3. hit
4. him
----------
Done! Execution time: 0 ms
Total nodes visited: 7
```

6. Uji Kasus 6 – *Coffee* ke *Farmer*

```
>> Coffee Farmer UCS

----------
1. coffee
2. coffer
3. confer
4. conker
5. corker
6. forker
7. former
8. farmer
----------
Done! Execution time: 46 ms
Total nodes visited: 2908
```

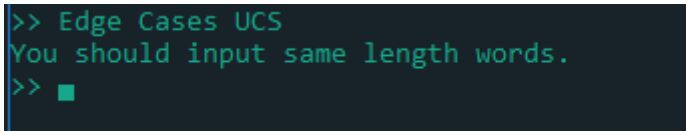
```
>> Coffee Farmer AStar

----------
1. coffee
2. coffer
3. confer
4. conker
5. corker
6. forker
7. former
8. farmer
----------
Done! Execution time: 1 ms
Total nodes visited: 47
```

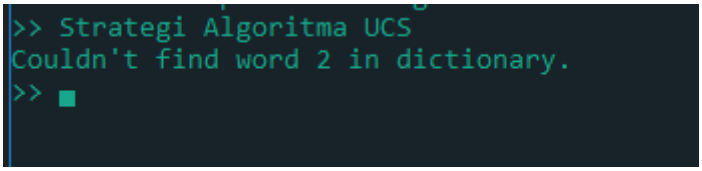
```
>> Coffee Farmer GBFS

----------
1. coffee
2. coffer
3. goffer
4. gaffer
5. gaufer
6. gauger
7. mauger
8. mauler
9. marler
10. marker
11. darker
12. darner
13. warner
14. warmer
15. farmer
----------
Done! Execution time: 2 ms
Total nodes visited: 33
```

7. Uji Kasus 7 – *Edge* ke *Cases*



8. Uji Kasus 8 – *Strategi* ke *Algoritma*



G. Analisis Hasil Uji

Setelah dilakukan pengujian, telah dibuktikan bahwa algoritme UCS dan A* pada implementasi ini berhasil menemukan solusi optimal untuk semua solusi (dilihat dari semua uji kasus). Selain itu, algoritme A* dan GBFS menghasilkan solusi dalam waktu yang lebih cepat daripada UCS sebagaimana ditunjukkan pada uji kasus dua dan enam. Namun, GBFS belum tentu menghasilkan solusi yang optimal seperti pada uji kasus 2, 4, dan 6. Di samping itu, ada beberapa *insight* yang bisa didapat dari pengujian yang dilakukan:

1. Fungsi heuristik GBFS cukup baik sehingga bisa memberikan solusi optimal untuk sebagian kasus, terutama apabila panjang kata kecil seperti pada uji kasus 1 dan 5.
2. Pada uji kasus 3 (tidak ada solusi), Dapat dilihat properti *exhaustive* dari ketiga algoritme. Bahwa sebenarnya, ketiga algoritma tetap mencoba seluruh jalur (dilihat dari *total nodes visited*-nya), hanya berbeda di urutan pengecekannya. Waktu eksekusi pun rata-rata sama.
3. Uji kasus 5 mengilustrasikan bagaimana UCS tidak memprioritaskan simpul apa pun sehingga jumlah simpul penelusurannya sangat banyak dibanding GBFS dan A*.
4. Jika diurutkan berdasarkan jumlah node yang dijelajahi, GBFS < A* < UCS. Oleh karena jumlah simpul ekspan berkontribusi kepada jumlah simpul hidup yang harus disimpan dalam *queue*, jumlah memori yang digunakan ketiga algoritme ini pun mempunyai urutan yang sama.

H. Tautan Repository GitHub

https://github.com/AldyDPP/Tucil3_13522022