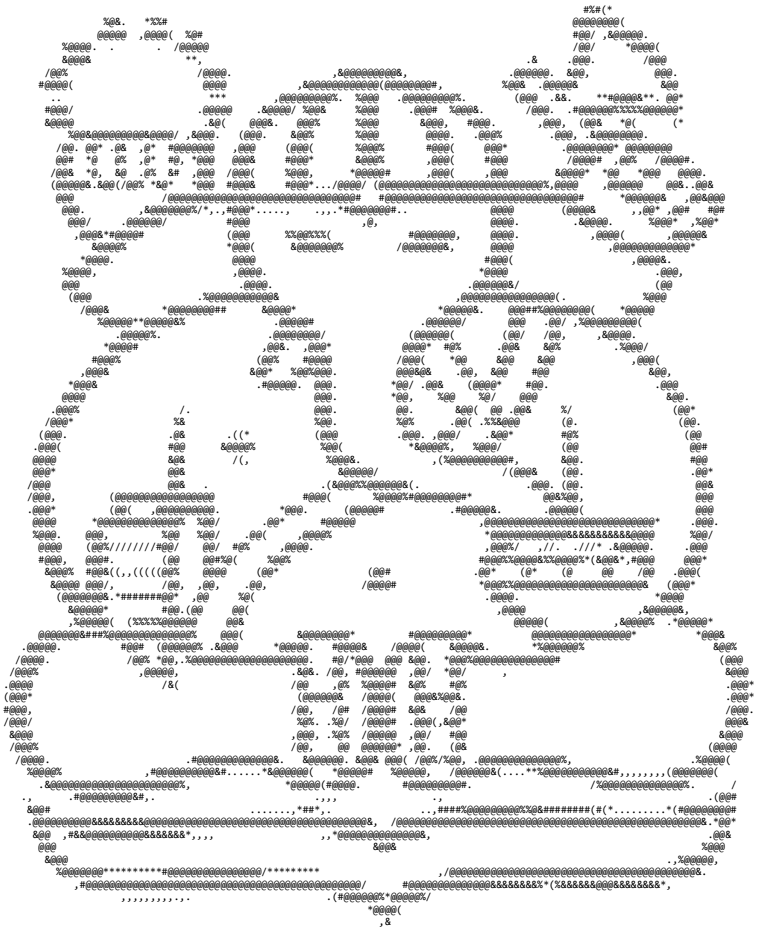
Laporan Tugas Kecil 3

IF2211 Strategi Algoritma

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

Semester II tahun 2023/2024



Disusun Oleh:

Abdul Rafi Radityo Hutomo (13522089)

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2024

Daftar Isi

BAB I DESKRIPSI PERMASALAHAN **4**

BAB II LANDASAN TEORI **5**

2.1. Dasar Teori 5

2.1.1 Algoritma Pathfinding 5

2.1.2 Algoritma Uniform Cost Search 5

2.1.3 Algoritma Greedy Best-First Search 5

2.1.3 Algoritma A Star (A*) 6

BAB III IMPLEMENTASI ALGORITMA **7**

3.1 Uniform Cost Search 7

1. Inisialisasi frontier 7

2. Eksplorasi simpul dan pengecekkan solusi 8

3. Pengulangan hingga ditemukan solusi 9

3.2 Greedy Best First Search Algorithm 10

1. Inisialisasi currentPath 11

2. Eksplorasi simpul dan pengecekkan solusi 11

3. Pengulangan hingga ditemukan solusi 12

3.3 A* Algorithm 12

1. Inisialisasi frontier 13

2. Eksplorasi simpul dan pengecekkan solusi 13

3. Pengulangan hingga ditemukan solusi 14

BAB IV IMPLEMENTASI DAN PENGUJIAN **16**

4.1 Spesifikasi Teknis dan Implementasi Program 16

4.1.1 Spesifikasi Teknis Program 16

4.1.2 Penjelasan Implementasi Algoritma 19

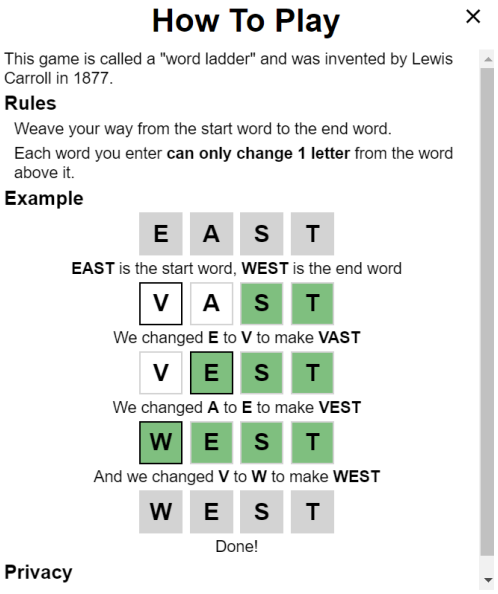
Algoritma Uniform Cost Search 19

Algoritma Greedy Best-First Search 21

Algoritma A* 22
Implementasi *Heuristic* 24
Implementasi GUI (Bonus) 26
4.2 Hasil Pengujian 27
4.3 Analisis Hasil Pengujian 29
Analisis *Heuristic* 29
Perbandingan Ketiga Algoritma 29
BAB V KESIMPULAN DAN SARAN 31
LAMPIRAN 32
DAFTAR PUSTAKA 33

BAB I
DESKRIPSI PERMASALAHAN

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

Pada Tugas Kecil 3 Strategi Algoritma ini, mahasiswa Teknik Informatika 2022 diharapkan untuk membuat sebuah program dalam bahasa Java untuk menyelesaikan permasalahan word ladder dengan menggunakan algoritma path-finding yang dipelajari pada perkuliahan, yaitu algoritma Uniform Cost Search (UCS), Greedy Best First Search, dan A*.

BAB II
LANDASAN TEORI

2.1. Dasar Teori

2.1.1 Algoritma Pathfinding

Algoritma pathfinding dapat adalah algoritma yang bertujuan untuk mencari urutan simpul yang berdekatan dalam sebuah graf terarah, yang menghubungkan dua simpul awal dan akhir. Graf ini merepresentasikan lingkungan yang ingin dilalui, dengan simpul mewakili lokasi dan sisi mewakili koneksi antar lokasi. Algoritma ini bekerja dengan mengevaluasi fungsi biaya untuk setiap sisi, yang mencerminkan "biaya" untuk melewati sisi tersebut. Biaya ini dapat berupa jarak, waktu, konsumsi energi, atau kombinasi dari beberapa faktor.

Proses pencarian jalur dimulai dari simpul awal dan mengeksplorasi sisi-sisi yang terhubung dengannya. Untuk setiap sisi, algoritma menghitung biaya total untuk mencapai simpul tujuan melalui tepian tersebut, dengan menjumlahkan biaya sisi dan biaya yang telah diakumulasikan untuk mencapai simpul saat ini. Simpul-simpul yang dieksplorasi kemudian disimpan dalam struktur data, yang diurutkan berdasarkan biaya total terendah.

Algoritma ini terus mengeksplorasi simpul-simpul baru sampai simpul tujuan ditemukan. Ketika simpul tujuan ditemukan, urutan simpul yang dilalui dari simpul awal ke simpul tujuan merepresentasikan jalur yang optimal.

2.1.2 Algoritma Uniform Cost Search

Algoritma Uniform Cost Search (UCS) adalah salah satu algoritma pathfinding yang dapat digunakan dalam persoalan mencari jalur optimal dalam graf berbobot. Algoritma UCS bekerja dengan cara melakukan eksplorasi simpul secara berurutan, dengan urutan eksplorasi simpul sesuai dengan *cost* atau harga yang dibutuhkan untuk mencapai ke simpul tersebut. Algoritma ini tergolongkan sebagai algoritma *uninformed* sebab tidak mempertimbangkan *heuristic* atau perkiraan kedekatan sebuah simpul dengan simpul target.

2.1.3 Algoritma Greedy Best-First Search

Algoritma Greedy Best-First Search adalah sebuah algoritma greedy yang dapat digunakan untuk menyeselaikan permasalah pathfinding untuk mencari jalur optimal dalam graf berbobot. Berbeda dengan algoritma UCS, algoritma Greedy Best-First search menggunakan sebuah *heuristic*, yaitu perkiraan jarak sebuah simpul dengan simpul target. Algoritma Greedy Best-

First Search bekerja dengan cara mengeksplorasi simpul yang “terlihat” terbaik berdasarkan fungi *heuristic*. Namun, algoritma Greedy Best-First Search memiliki kelemahan, yaitu dapat terjebak dalam minimum lokal sehingga tidak menemukan solusi yang optimal, atau bahkan tidak menemukan solusi sama sekali.

2.1.3 Algoritma A Star (A*)

Algoritma A* adalah algoritma pathfinding yang menggabungkan konsep UCS dan Greedy Best-First Search sehingga diharapkan didapat algoritma yang memiliki kecepatan serupa dengan Greedy Best-First Search tetapi dapat menemukan solusi yang optimal. Algoritma A* bekerja dengan cara mengeksplorasi simpul-simpul dengan memprioritaskan simpul yang memiliki nilai *cost* + nilai *heuristic* paling rendah, dengan fungsi *heuristic* sama dengan pada Greedy Best-First Search, yaitu fungsi untuk memperkirakan jarak antara sebuah simpul dengan simpul target.

BAB III
IMPLEMENTASI ALGORITMA

3.1 Uniform Cost Search

Permasalahan Word Ladder dapat dipetakan ke dalam komponen permasalahan *pathfinding* pada graf berbobot tdengan pemetaan sebagai berikut,

- Simpul : Kata yang valid
- Sisi : Memiliki panjang yang sama dan berbeda pada satu karakter/huruf
- Simpul Akar : Kata awal
- Simpul Daun : Kata Akhir
- Bobot : Bernilai satu untuk setiap sisi, sebab bobot merubah satu kata ke kata lain sama semua

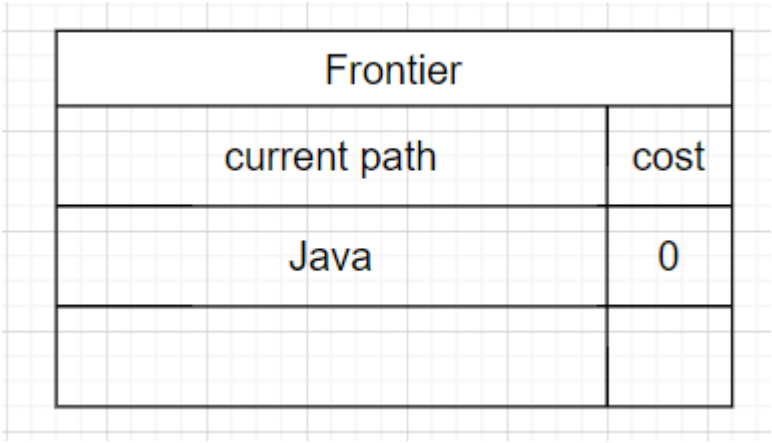
Algoritma Uniform Cost Search dalam perhitungannya membutuhkan sebuah fungsi g(n), yaitu fungsi yang menghitung biaya untuk ke simpul tersebut dari simpul akar. Pada kasus ini fungsi g(n) tidak lain adalah sama dengan kedalaman dari simpul tersebut dari simpul akar yang merepresentasikan berapa kali diperlukan substitusi karakter. Dalam algoritma *pathfinding*, digunakan sebuah struktur data *priority queue* yang berisi path yang masih berpotensi untuk diekspansi, *priority queue* ini sering disebut sebagai sebuah *frontier*.

Dengan pemetaan tersebut didapat algoritma Uniform Cost Search dengan langkah sebagai berikut beserta dengan contoh ilustrasi kasus pencarian dari kata *java* ke *pave* :

1. Inisialisasi frontier

   Pada awal algoritma, frontier diinisialisasi dengan sebuah path berisi hanya simpul akar, yaitu string awal, dengan nilai *cost*, sebesar 0. Kemudian, juga diinisialisasi set visited untuk menyimpan string yang telah dilewati

   Pada ilustrasi kasus, kata *java* dienqueue ke dalam frontier yang dapat digambarkan sebagai berikut.



2. Eksplorasi simpul dan pengecekkan solusi

Path yang memiliki cost paling rendah dalam *frontier*, kemudian di dequeue dan dilakukan eksplorasi kata-kata yang dapat dibuat dengan mengganti satu karakter dan belum pernah dikunjungi atau memiliki *cost* lebih rendah daripada pengunjungan string tersebut sebelumnya. Namun, karena *cost* dari setiap sisi sama, maka pengunjungan pertama dari sebuah kata dijamin memiliki *cost* yang paling rendah. Pada langkah ini, juga diperiksa apakah target yang ingin dicari dapat dicapai dari simpul akhir dari path yang sedang dieksplorasi saat ini, jika solusi bisa dicapai maka program berhenti dan memberikan solusi. Jika solusi belum ditemukan, maka setiap kata-kata yang baru dibangkitkan tersebut dimasukkan ke dalam *frontier* sebagai path yang dibutuhkan untuk mencapai simpul tersebut dan nilai *cost* atau g(n) sama dengan nilai *cost* path sebelumnya + 1.

Pada ilustrasi kasus, dari kata Java diilustrasikan 3 kata yang dapat dicapai, (meskipun ada lebih banyak) yaitu *cava*, *lava*, dan *kava*. Ketiga kemungkinan tersebut dimasukkan ke dalam *frontier* dengan *cost* bernilai 1, yaitu cost dari path *Java* ditambah 1

Frontier	
current path	cost
java, cava	1
java, lava	1
java, kava	1

3. Pengulangan hingga ditemukan solusi

Kemudian, eksplorasi simpul dan pengecekkan solusi dilakukan terus menerus hingga ditemukan solusi atau semua kata telah dieksplorasi yang berarti tidak ada solusi.

Pada ilustrasi kasus, pembangkitan dilanjutkan ke path [java, cava] sebab semua costnya bernilai sama dan didapatkan *frontier* baru sebagai berikut

Frontier	
current path	cost
java, lava	1
java, kava	1
java, cava, cavy	2
java, cava, cave	2
java, cava, caca	2

Dapat diperhatikan bahwa path dengan *cost* 2 akan terletak dibelakang semua *path* yang memiliki *cost* 1. Kemudian, pembangkitan dilanjutkan hingga semua path cost 1 telah di dequeue dan didapat *frontier* sebagai berikut

Frontier	
current path	cost
java, cava, cavy	2
java, cava, cave	2
java, cava, caca	2
java, lava, kafa	2
java, lava, kavi	2
java, kava, yaya	2
java, kava, yaka	2

Kemudian, algoritma dilanjutkan terus menerus hingga saat path [java, cava, cave] di dequeue didapat pave sebagai kata yang dapat dibangkitkan dari cave sehingga pencarian diselesaikan dan solusi dikembalikan.

3.2 Greedy Best First Search Algorithm

Greedy Best-First Search juga dapat digunakan untuk mencoba menyelesaikan permasalahan word ladder dengan pemetaan elemen yang sama dengan UCS. Namun, perbedaan pada algoritma Greedy Best First Search Algorithm terdapat pada pembobotan simpul dan simpul yang dieksplorasi. Pada algoritma Greedy Best First Search digunakan *heuristic* h(n) untuk memperkirakan jarak suatu simpul/string dengan simpul tujuan. Untuk detail *heuristic* h(n)

yang digunakan akan dijelaskan pada subbab 3.4. Berbeda dengan UCS, algoritma Greedy Best-First Search tidak menggunakan Priority Queue atau *frontier*, sebab algoritm Greedy Best-First Search yang diimplementasikan pada Tugas Kecil ini tidak menggunakan backtracking sehingga simpul yang belum dieksplorasi tidak disimpan, melainkan program hanya menyimpan satu path saja yang akan dirujuk dengan istilah currentPath.

Dengan begitu bisa didapat langkah-langkah pada algoritma Greedy Best-First Search dengan ilustrasi kasus yang sama pencarian solusi word ladder dari *pave* ke *vile* dengan *heuristic* hemming distance.

1. Inisialisasi currentPath

   Pada awal algoritma, currentPath diinisialisasi dengan isi path sepanjang satu yaitu simpul akar. Selain itu, set of visited words juga diinisialisasi dengan kata *pave*.

   Pada ilustrasi kasus variabel currentPath diinisialisasi dengan [“pave”]

2. Eksplorasi simpul dan pengecekkan solusi

   Kemudian dari simpul tersebut dieksplorasi kemungkinan kata yang dapat dibuat. Apabila simpul target merupakan salah satu kata yang dapat dibentuk, maka pencarian selesai dan kata akhir ditambahkan ke currentPath sehingga membentuk solusi. Jika tidak, maka dari semua kemungkinan kata, ambil kata dengan jarak terkecil ke target berdasarkan fungsi *heuristic* dan belum pernah dikunjungi. Simpan kata tersebut pada set of visited nodes. Apabila dari semua kemungkinan kata yang ada, semuanya sudah dikunjungi maka algoritma Greedy Best-First Search terjebak dalam minimum lokal dan gagal mendapatkan solusi.

   Pada ilustrasi kasus, dari kata *pave* terdapat kata berikut yang dapat dicapai beserta nilai *hamming distance* terhadap kata target *vile*

Kemungkinan Kata	Hamming Distance (thd. vile)
cave	3
pace	3
dave	3
eave	3
gave	3

page	3
have	3
lave	3
pale	2
nave	3
pane	3
pavo	3
pape	3
rave	3
pare	3
save	3
pase	3
tave	3
pate	3
pavy	3
wave	3

Didapati bahwa kata pale memiliki nilai *heuristic* terendah, yaitu sebesar 2, dibandingkan dengan kata-kata lainnya yang memiliki distance 3. Maka currenPath menjadi sama dengan [pave, pale] dan karena belum ditemukan solusi maka pencarian diteruskan.

3. Pengulangan hingga ditemukan solusi

Kemudian, eksplorai simpul dan pengecekkan solusi diteruskan hingga ditemukan solusi atau sudah terjebak pada minimum lokal. Pada kasus ini, algoritma berhasil menemukan solusi dengan kata pale dilanjutkan dengan kata pile (hamming distance = 1) kemudian vile membentuk solusi [pave, pale, pile, vile].

3.3 A* Algorithm

Terakhir, Algoritma A* mengkombinasikan fungsi g(n) pada UCS dan fungsi h(n) pada Greedy Best-First Search. Algoritma ini juga memiliki kemiripan dengan UCS, yaitu penggunaan Priority Queue yang juga disebut *frontier* yang digunakan untuk pemrosesan simpul/kata secara sekuensial dengan memprioritaskan simpul dengan nilai f(n) = g(n) + h(n) terkecil.

Dengan pemetaan tersebut didapat algoritma A* dengan langkah sebagai berikut beserta dengan contoh ilustrasi kasus pencarian dari kata *vile* ke *omen* dengan heuristic hamming distance :

1. Inisialisasi frontier

    Pada awal algoritma, frontier diinisialisasi dengan sebuah path berisi hanya simpul akar, yaitu string awal, dengan nilai f(n) dihitung sebagai g(n) + h(n) dengan g(n) = 0 karena belum ada gerakan yang dibuat sehingga tidak memerlukan biaya. Kemudian, juga diinisialisasi set visited untuk menyimpan string yang telah dilewati dengan pertama diisi kata awal.

    Pada ilustrasi kasus, path berisi kata *vile* dienqueue ke dalam frontier yang dapat digambarkan sebagai berikut.

Frontier	
Path	f(n)
[vile]	4

2. Eksplorasi simpul dan pengecekkan solusi

    Path yang memiliki nilai f(n) paling rendah dalam *frontier*, kemudian di dequeue dan dilakukan eksplorasi kata-kata yang dapat dibuat dengan mengganti satu karakter dan belum pernah dikunjungi atau memiliki *cost* lebih rendah daripada pengunjungan string tersebut sebelumnya. Namun, karena *cost* dari setiap sisi sama, maka pengunjungan pertama dari sebuah kata dijamin memiliki *cost* yang paling rendah. Pada langkah ini, juga diperiksa apakah target yang ingin dicari dapat dicapai dari simpul akhir dari path yang sedang dieksplorasi saat ini, jika solusi bisa dicapai maka program berhenti dan

memberikan solusi. Jika solusi belum ditemukan, maka setiap kata-kata yang baru dibangkitkan tersebut dimasukkan ke dalam *frontier* sebagai path yang dibutuhkan untuk mencapai simpul tersebut dan nilai *cost* atau g(n) sama dengan nilai *cost* path sebelumnya + 1. Untuk setiap kata tersebut nilai f(n) dihitung dengan menjumlahkan g(n) + h(n).

Pada ilustrasi kasus, dari kata *vile* diilustrasikan 3 kata yang dapat dicapai, (meskipun ada lebih banyak) yaitu *vale*, *bile*, dan *aile*. Ketiga kemungkinan tersebut dimasukkan ke dalam *frontier* dengan *cost* bernilai 1, yaitu cost dari path *Java* ditambah 1 dan f(n) bernilai 5 sebab nilai *hamming distance*-nya semua adalah 4 dari kata *omen*.

Frontier	
Path	f(n)
[vile, vale]	5
[vile, bile]	5
[vile, aile]	5

3. Pengulangan hingga ditemukan solusi

Kemudian, eksplorasi simpul dan pengecekkan solusi dilakukan terus menerus hingga ditemukan solusi atau semua kata telah dieksplorasi yang berarti tidak ada solusi.

Pada ilustrasi kasus, pembangkitan dilanjutkan ke path [vile, vale] sebab semua nilai f(n) bernilai sama dan [vile, vale] adalah path yang dibangkitkan pertama sehingga didapatkan *frontier* baru sebagai berikut

Frontier	
Path	f(n)
[vile, bile]	5
[vile, aile]	5
[vile, vale, dale]	6
[vile, vale, vali]	6
[vile, vale, male]	6

Dapat diperhatikan bahwa path dengan h(n) = 6 akan terletak dibelakang semua *path* yang memiliki h(n) = 5. Kemudian, pembangkitan dilanjutkan hingga semua path dengan h(n) = 5 telah di dequeue

Kemudian, algoritma dilanjutkan terus menerus hingga ditemukan solusi, yaitu [vile, aile, alle, alee, alen, amen, omen]

BAB IV
IMPLEMENTASI DAN PENGUJIAN

4.1 Spesifikasi Teknis dan Implementasi Program

4.1.1 Spesifikasi Teknis Program

Spesifikasi Teknis Word Processing

Class WordSet Implements Iterator : Sebagai kamus yang berisi kata-kata

Atribut	Type	Kegunaan
wordStore	HashSet <String>	Set untuk menyimpan semua kata dalam dictionary
defaultPath	Static string	Default dictionary path

Method	Kegunaan
wordStore(optional String)	Konstruktor, dapat menerima input path ke file txt atau menggunakan dictionary default
loadFile(String)	Melakukan pembacaan sebuah file dan di load ke dalam set, dipanggil pada konstruktor
Boolean wordExist(string)	Mengembalikan true jika argumen string ada dalam dictionary, false jika tidak
Iterator()	Method untuk melakukan iterasi terhadap elemen

Class WordGenerator

Atribut	Type	Kegunaan

Method	Kegunaan
getAdjacentWords(String, WordSet)	Mengembalikan semua kata yang memiliki selisih 1 karakter

Class TextDistanceCalculator : Kalkulator *heuristic* untuk memperkirakan jarak antara 2 kata

Atribut	Type	Kegunaan
table	Map<Character, Map<Character, Double>>	Digunakan untuk menyimpan data metode *Heuristic Frequency Table*

Method	Kegunaan
TextDistanceCalculator()	Konstruktor, membaca dan mengkonstruksi tabel dari file distance.txt
Double calculateDistance(String, String)	Mengembalikan perkiraan jarak antara dua string
generateCSVTable(WordSet)	Membaca dictionary dan membuat *Heuristic Frequency Table* dan disimpan secara persistent ke dalam file txt

Spesifikasi Teknis Algoritma

Class Pair<T, E> Implements Comparable<Pair> : Merepresentasikan elemen queue pada algoritma yang perlu menyimpan path dan juga bobotnya

Atribut	Type	Kegunaan
first	T	Elemen pertama dari pair
second	E	Elemen kedua dari pair

Method	Kegunaan
Getter, setter first dan second	Untuk mendapat dan memberi nilai ke elemen

Class UniformCostSearch : melakukan algoritima UCS

Atribut	Type	Kegunaan
wg	wordGenerator	Melakukan perhitungan kemungkinan kata yang dapat dibangkitkan dari suatu kata lain

dict	wordSet	Dictionary yang digunakan
end	String	Target string yang ingin dicapai
theQueue	PriorityQueue	Menyimpan path-path yang perlu diproses secara terurut berdasarkan bobot *cost*-nya
visited	Map	Menyimpan kata yang telah dieksplor
done	boolean	Status pencarian, bernilai true jika solusi sudah ditemukan
solution	List	Solusi akhir
visitedNode	Integer	Jumlah kata/simpul yang dikunjungi selama pencarian
elapsedTime	Long	Waktu yang dibutuhkan untuk mendapatkan solusi

Method	Kegunaan
UniformCostSearch()	Konstruktor, menginisialisasi variabel-variabel yang dibutuhkan
Solve (String, String)	Menjalankan algoritma untuk membuat wordLadder dari argumen pertama ke argumen kedua
trivialCase()	Handler untuk kasus trivial start = end
evaluateNextNode()	Mengambil elemen pada head queue dan mencari semua kata yang dapat dibangkitkakn dari kata saat itu
resolveBranch(Pair, String)	Menerima sebuah string yang dibangkitkan dari pair dan melakukan pengecekkan solusi serta enqueue ke priority queue jika diperlukan
Getter untuk solution, elapsedTime, dan visitedNode	Digunakan untuk mengakses hasil perhitungan

Class GreedyBFS : melakukan algoritima Greedy Best-First Search

Atribut	Type	Kegunaan
wg	wordGenerator	Melakukan perhitungan kemungkinan kata yang dapat dibangkitkan dari suatu kata lain

dict	wordSet	Dictionary yang digunakan
tdc	TextDistanceCalculator	Objek untuk menghitung *heuristic*
end	String	Target string yang ingin dicapai
currentItem	List of String	Menyimpan path yang saat ini diproses
visited	Map	Menyimpan kata yang telah dieksplor
done	boolean	Status pencarian, bernilai true jika solusi sudah ditemukan
solution	List	Solusi akhir
visitedNode	Integer	Jumlah kata/simpul yang dikunjungi selama pencarian
elapsedTime	Long	Waktu yang dibutuhkan untuk mendapatkan solusi

Method	Kegunaan
GreedyBFS()	Konstruktor, menginisialisasi variabel-variabel yang dibutuhkan
Solve (String, String)	Menjalankan algoritma untuk membuat wordLadder dari argumen pertama ke argumen kedua
trivialCase()	Handler untuk kasus trivial start = end
evaluateNextNode()	Mengeksplor semua kemungkinan kata yang dapat dibangkitkan dari currentItem, kemudian melakukan pengecekkan solusi dan perhitungan simpul dengan bobot minimum untuk ditambahkan ke path currentItem
evaluateWeight(String)	Mengembalikan nilai *heuristic* dari kata pada argumen ke kata target
Getter untuk solution, elapsedTime, dan visitedNode	Digunakan untuk mengakses hasil perhitungan

4.1.2 Penjelasan Implementasi Algoritma

Algoritma Uniform Cost Search

```java
public void solve(String start, String end){  abdulrafirh
    this.end = end;
    this.visitedNode = 1;
    this.elapsedTime = System.nanoTime();

    ArrayList<String> startPath = new ArrayList<>();
    startPath.add(start);
    visited.add(start);
    theQueue.add(new Pair<>(startPath,  _second: 0.0));
    if (start.equals(end)) {trivialCase(); return;}

    while(!done && !theQueue.isEmpty()){
        evaluateNextNode();
    }

    this.elapsedTime = System.nanoTime() - this.elapsedTime;
}
```

Method solve dilakukan dengan cara menginisialisasi Priority Queue dengan path berisi start path kemudian melakukan looping selama belum ditemukan solusi dan masih ada path yang dapat dieksplor. Setiap iterasi looping dievaluasi node yang ada pada head queue.

```java
public void evaluateNextNode(){  1 usage  abdulrafirh
    Pair<List<String>, Double> currentItem = theQueue.remove();

   List<String> possibilities = wg.getAdjacentWords(currentItem.first().getLast(), dict);

   for(int i = 0; i < possibilities.size() && !done; i++){
       if (!visited.contains(possibilities.get(i))){
           resolveBranch(currentItem, possibilities.get(i));
           visited.add(possibilities.get(i));
           this.visitedNode += 1;
       }
   }
}
```

Method evaluateNextNode mengambil node pada head dari queue dan membangkitkan semua kata yang dapat dijangkau dengan satu kali penggantian karakter. Kemudian untuk setiap kata

tersebut, apabila kata tersebut belum dieskplor maka eksplor kata tersebut dan melakukan penandaan bahwa kata tersebut telah dikunjungi.

```
public void resolveBranch(Pair<List<String>, Double> prev, String current){
    if (current.equals(end)){
        done = true;
        solution = new ArrayList<>(prev.first());
        solution.add(current);
    }
    else{
        List<String> newPath = new ArrayList<>(prev.first());
        newPath.add(current);
        theQueue.add(new Pair<>(newPath, _second: prev.second() + 1.0));
    }
}
```

Method resolveBranch melakukan “pengunjungan” sebuah kata, yaitu mengecek apakah kata tersebut merupakan solusi, jika tidak mengenqueue path baru yang sudah termasuk kata tersebut dengan nilai cost diincrement dari path sebelumnya.

Algoritma Greedy Best-First Search

```
public void solve(String start, String end) throws Exception{  abdulrafirh *
    this.end = end;
    this.elapsedTime = System.nanoTime();
    this.visitedNode = 0;

    ArrayList<String> startPath = new ArrayList<>();
    startPath.add(start);
    currentItem = startPath;
    if (start.equals(end)) {trivialCase(); return;}

    while(!done && !visited.contains(currentItem.getLast())){
        visited.add(currentItem.getLast());
        evaluateNextNode();
    }

    this.elapsedTime = System.nanoTime() - this.elapsedTime;
    if (!done){
        currentItem.removeLast();
        solution = currentItem;
        throw new Exception("Greedy BFS : failed to get solution or no solution exists");
    }
}
```

Method solve dilakukan dengan cara menginisialisasi currentItem dengan path berisi start path kemudian melakukan looping selama belum ditemukan solusi dan masih ada kata yang bertetanggaan dari kata terakhir pada path yang belum dieksplor. Setiap iterasi looping dievaluasi simpul currentItem.

```
public void evaluateNextNode(){  1 usage   abdulrafirh
    List<String> possibilities = wg.getAdjacentWords(currentItem.getLast(), dict);
    int nextIdx = 0;
    double minVal = -1;
    for(int i = 0; i < possibilities.size() && !done; i++){
        if (possibilities.get(i).equals(end)){
            done = true;
            solution = currentItem;
            solution.add(end);
            return;
        }
        if (!visited.contains(possibilities.get(i))){
            if (minVal == -1 || evaluateWeight(possibilities.get(i)) < minVal){
                minVal = evaluateWeight(possibilities.get(i));
                nextIdx = i;
            }
            visitedNode += 1;
        }
    }
    currentItem.add(possibilities.get(nextIdx));
}
```

Method evaluateNextNode mengambil membangkitkan semua kata yang dapat dijangkau dengan satu kali penggantian karakter dari kata terakhir yang dikunjungi. Kemudian untuk setiap kata tersebut, apabila kata tersebut belum dieskplor maka dihitung nilai h(n) dari simpul tersebut dan dicari simpul yang belum dieksplor dan memiliki nilai h(n) minimum untuk dicatat sebagai simpul pencarian selanjutnya.

Algoritma A*

```java
public void solve(String start, String end) throws Exception{  abdu
    this.end = end;
    this.visitedNode = 1;
    this.elapsedTime = System.nanoTime();
    ArrayList<String> startPath = new ArrayList<>();
    startPath.add(start);
    visited.add(start);
    theQueue.add(new Pair<>(startPath, evaluateWeight(startPath)));
    if (start.equals(end)) {trivialCase(); return;}

    while(!done && !theQueue.isEmpty()){
        evaluateNextNode();
    }

    this.elapsedTime = System.nanoTime() - this.elapsedTime;
    if (!done){
        solution = new ArrayList<>();
        throw new Exception("A* : No solution exists");
    }
}
```

```java
public void evaluateNextNode(){  1 usage  abdulrafirh
    Pair<List<String>, Double> currentItem = theQueue.remove();

    List<String> possibilities = wg.getAdjacentWords(currentItem.first().getLast(), dict);

    for(int i = 0; i < possibilities.size() && !done; i++){
        if (!visited.contains(possibilities.get(i))){
            resolveBranch(currentItem, possibilities.get(i));
            visited.add(possibilities.get(i));
            this.visitedNode += 1;
        }
    }
}
```

```
public void resolveBranch(Pair<List<String>, Double> prev, String current){  1 usage  abdulrafirh
    if (current.equals(end)){
        done = true;
        solution = new ArrayList<>(prev.first());
        solution.add(current);
    }
    else{
        List<String> newPath = new ArrayList<>(prev.first());
        newPath.add(current);
        theQueue.add(new Pair<>(newPath, evaluateWeight(newPath)));
    }
}

public double evaluateWeight(List<String> current){  2 usages  abdulrafirh
    return current.size() - 1 + tdc.CalculateDistance(current.getLast(), end);
}
```

Penjelasan dipersingkat karena seluruhnya sama persis dengan UCS, hanya memiliki perbedaan saat melakukan enqueue ke dalam Priority Queue dimana bobot yang dimasukkan dihitung sebagai fungsi f(n) = g(n) + h(n), yang dihitung dengan fungsi evaluateWeight, dimana g(n) adalah jarak dari start node, yang dihitung dengan cara menghitung panjang path – 1 ditambah dengan h(n) yang metode perhitungannya akan dibahas pada algoritma *heuristic*

Implementasi *Heuristic*

Terdapat 2 metode *Heuristic* yang dapat menjadi opsi untuk digunakan dalam pencarian ini

a) Hamming Distance

   Hamming distance adalah besaran jarak yang didefinisikan untuk dua string yang memiliki panjang yang sama, dengan nilai hamming distance sama dengan jumlah posisi dimana karakter pada string pertama dan kedua berbeda.

```
public Double CalculateDistance(String text1, String text2, boolean admissible){  1 usage  new *
    double distance = 0.0;
    for(int i = 0; i < text1.length(); i++){
        if (!admissible) {distance += 3*table.get(text1.charAt(i)).get(text2.charAt(i));}
        else {if (text1.charAt(i) != text2.charAt(i)) {distance += 1;}}
    }
    return distance;
}
```

   Implementasi hamming distance adalah for loop yang masuk ke cabang else atau argumen admissible bersifat true

b) Successful Substitution Rate Table (SSR Table)

SSR Table adalah konsep yang saya ciptakan sendiri untuk berusaha memperkirakan jarak antara dua kata. SSR Table merupakan sebuah table 2 dimensi yang diindex dengan 2 buah karakter. Tabel[C1][C2] akan mengembalikan nilai perkiraan jarak antara karakter C1 ke C2, sehingga jarak antara 2 kata adalah jumlah dari jarak setiap karakternya pada posisi yang sama. Nilai jarak antara C1 dan C2 dihitung dengan cara mencoba mengganti setiap kemunculan C1 menjadi C2 dan memeriksa apakah substitusi tersebut menghasilkan kata yang terdapat pada dictionary atau tidak. Besaran yang dijadikan jarak pada tabel dihitung dengan rumus berikut

$$d(c1, c2) = \log_{10}(tryFrequency/10hitFrequency)$$

Dengan c1 adalah karakter pertama dan c2 adalah karakter kedua, tryFrequency adalah jumlah percobaan merubah c1 menjadi c2 dan hitFrequency adalah jumlah dimana mengganti c1 menjadi c2 berhasil memberikan kata baru yang valid dalam dictionary.

```
for (String word : ws) {

    if (progress % (ws.length()/10) == 0){
        System.out.printf("Currently examining %d/%d words\n", progress, ws.length());
    }

    progress++;
    for (int i = 0; i < word.length(); i++){
        char currentChar = word.charAt(i);
        for(char chara = 'a'; chara <= 'z'; chara++){
            if (currentChar != chara) {
                StringBuilder maybeWord = new StringBuilder(word);
                maybeWord.setCharAt(i, chara);

                nTable.get(currentChar).put(chara, nTable.get(currentChar).get(chara) + 1);
                if (ws.wordExist(maybeWord.toString())) {
                    hitTable.get(currentChar).put(chara, hitTable.get(currentChar).get(chara) + 1);
                }
            }
        }
    }
}
```

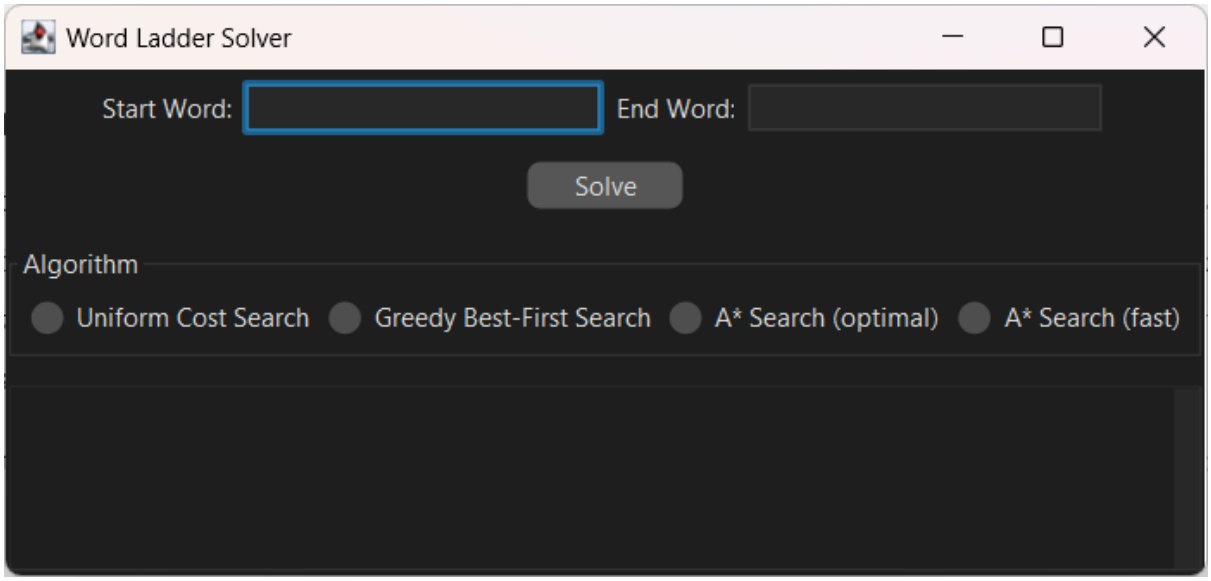
```
FileWriter writer = new FileWriter( fileName: "src/resources/distance.txt");
for (char chara = 'a'; chara <= 'z'; chara++){
    for (char charb = 'a'; charb <= 'z'; charb++){
        if (chara == charb){
            writer.write( str: "0 ");
        }
        else{
            long hitFrequency = hitTable.get(chara).get(charb)*10;
            long tryFrequency = nTable.get(chara).get(charb);
            writer.write(((Double)Math.log10((tryFrequency/(double)hitFrequency))).toString());
            writer.write( str: " ");
        }
    }
    writer.write( str: "\n");
}
writer.close();
```

Implementasi pembuatan SSR Table

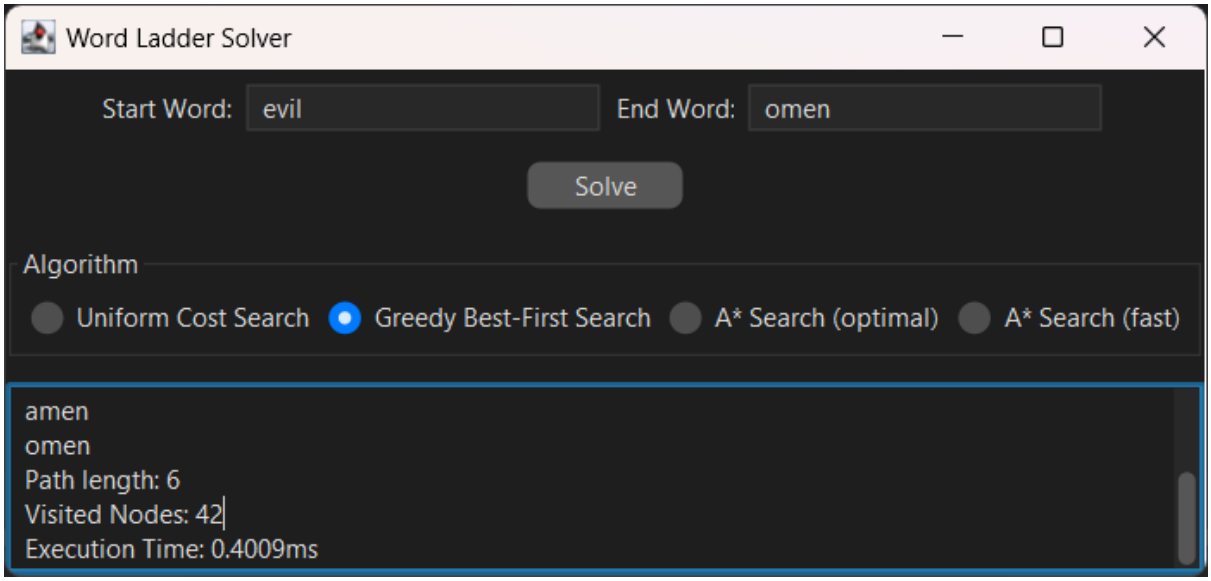
Implementasi GUI (Bonus)

Pada Tugas Kecil ini diimplementasikan interface GUI menggunakan library java swing serta add-on LAF untuk pengaturan display. Implementasi GUI sendiri menggunakan objek-objek bawaan yang ada pada library java swing seperti inputPanel, textArea, Button, RadioButton beserta dengan handlernya

Ketika program pertama kali dimulai, maka akan memunculkan tampilan sebagai berikut :



Kemudian, pengguna dapat memsukkan kata awal dan kata akhir pada kolom start word dan end word kemudian memilih algoritma yang ingin digunakan dan memencet solve untuk mendapatkan hasil.



Kemudian, program akan menunjukkan hasil pada area solusi berupa urutan kata dimulai dari start word hingga end word beserta panjng pathnya, visited node, dan execution time nya.

4.2 Hasil Pengujian

Perbandingan *Heuristic*

Uji GreedyBFS

Kata awal - akhir	Hamming Distance	SSR Table
lava - omen	Path length : 30 Execution time : 1.0396 ms	Path length : 12 Execution time : 0.6096 ms
haunt - ponds	tidak menemukan solusi	Path length : 19 Execution time : 0.6793 ms
rye - epi	Path length : 9 Execution time : 0.2111 ms	Path length : 8 Execution time : 0.2066 ms
love - yang	Path length : 5 0.1503 ms	Path length : 16 Execution time : 0.4072 ms
cookie - finish	tidak menemukan solusi	Tidak menemukan solusi

Uji A*

Pada penggunaan *Heuristic* juga dapat dikalikan faktor, untuk itu dibandingkan juga perlu diuji mana yang paling optimal.

	Hamming Distance	SSR Table

Kata awal akhir	x1	x2	x3	x1	x2	x3
chi - ego	Length 5 1.494 ms	Length 5 0.6174 ms	Length 5 0.09 ms	Length 5 0.3027 ms	Length 5 0.2898 ms	Length 5 0.1059 ms
atom - unau	Length 9 17.870 ms	Length 9 2.6982 ms	Length 9 0.553 ms	Length 9 7.4447 ms	Length 9 1.8439 ms	Length 9 0.574 ms
nylon - iller	Length 12 17.868 ms	Length 12 8.5327 ms	Length 14 5.973 ms	Length 12 4.5007 ms	Length 12 1.3691 ms	Length 15 1.3811 ms
boyish – painch	Length 21 201.47 ms	Length 21 201.38 ms	Length 22 63.703 ms	Length 21 181.10 ms	Length 22 91.944 ms	Length 22 56.52 ms
atlases - cabaret	Length 46 223.03 ms	Length 46 211.82 ms	Length 46 173.20 ms	Length 46 227.33 ms	Length 46 168.11 ms	Length 46 125.04 ms
quirking - wrathing	Length 21 16.966 ms	Length 21 11.456 ms	Length 22 6.9744 ms	Length 21 18.685 ms	Length 21 11.598 ms	Length 21 7.1641 ms

Perbandingan ketiga algoritma

A* admissible adalah menggunakan *heuristic* hamming table bobot 1x

A* fast adalah menggunakan *heuristic* SSR Table bobot 3x

kata awal - akhir	UCS	Greedy BFS	A* (admissible)	A* (fast)
chi - ego	Length 5 26.2501 ms	Length 5 0.2282 ms	Length 5 1.494 ms	Length 5 0.1059 ms
atom - unau	Length 9 47.6468 ms	Length 13 0.3459 ms	Length 9 17.870 ms	Length 9 0.574 ms
nylon - iller	Length 12 145.83 ms	Fail	Length 12 17.868 ms	Length 15 1.3811 ms
boyish – painch	Length 21 303.50 ms	Fail	Length 21 201.47 ms	Length 22 56.52 ms
atlases - cabaret	Length 46 233.84 ms	Fail	Length 46 223.03 ms	Length 46 125.04 ms
quirking - wrathing	Length 21 22.36 ms	Fail	Length 21 16.966 ms	Length 21 7.1641 ms

4.3 Analisis Hasil Pengujian

Analisis *Heuristic*

Heuristic Hamming Distance memiliki kelebihan dibanding SSR Table, yaitu sifatnya yang *admissible* atau nilainya tidak akan pernah *overestimate* biaya sebenarnya yang dibutuhkan untuk berpindah dari satu kata ke kata lain. Sedangkan, SSR table tidak admissible sebab ada beberapa jarak antar karakter yang memiliki nilai lebih dari 1 sehingga ada kemungkinan bahwa jarak yang diperkirakan melebihi jarak sebenarnya.

Analisis Algoritma Uniform Cost Search

Pada Algoritma Uniform Cost Search, pencarian yang dilakukan bisa dibilang sama persis dengan pencarian metode Breadth First Search. Hal ini disebabkan perbedaan dari penggunaan keduanya adalah UCS mempertimbangkan bobot graf dan berusaha mencari jalur dengan bobot terrendah. Namun, pada kasus word ladder bobot untuk bergerak dari satu kata ke kata lainnya adalah sama semua, sehingga sebenarnya permasalahan ini sama dengan permasalahan graf tidak berbobot, maka kedua algoritma berjalan dengan sangat serupa.

Selain itu, *heuristic* SSR Table memiliki *success rate* untuk algoritma GBFS, sehingga dapat disimpulkan bahwa untuk algoritma GBFS *heuristic* SSR Table lebih optimal. Namun, untuk algoritma A* karena *Heuristic* Hamming's Distance bersifat admissible maka apabila dibutuhkan A* yang optimal Hamming's Distance lebih layak untuk digunakan. Meskipun begitu, apabila dibutuhkan algoritma A* yang lebih cepat dapat digunakan *Heuristic* SSR Table.

Analisis Algoritma Greedy Best-First Search

Pada Algoritma Greedy Best-First Search, terlihat salah satu kekurangannya, yaitu algoritma Greedy Best-First-Search tidak dijamin mendapat solusi, atau tidak *complete*. Hal tersebut disebabkan oleh cara kerja algoritma GBFS yang tidak mampu melakukan *backtracking* atau memundurkan kondisi sehingga dapat terjebak dalam minimum lokal. Selain itu, ketika GBFS menemukan solusi juga hasilnya belum tentu optimal.

Perbandingan Ketiga Algoritma

Dari hasil percobaan, didapat bahwa algoritma UCS konsisten memiliki waktu eksekusi yang paling lama dan menggunakan memori yang paling besar dilihat dari jumlah visitedNode nya. Hal ini karena algoritma UCS melakukan pengecekkan yang menyeluruh dengan memperbesar jarak dari kata awal secara berurutan sehingga simpul yang dikunjungi akan lebih banyak daripada GBFS atau pun A*, tetapi hasil dari algoritma UCS selalu optimal karena karakteristik pencariannya yang menyeluruh. Kemudian, algoritma GBFS memiliki waktu eksekusi yang paling cepat, hal ini disebabkan karena algoritma GBFS tidak melakukan pencarian yang menyeluruh melainkan hanya memilih satu simpul pada setiap iterasi untuk dilanjutkan, secara memori juga penggunaannya paling sedikit karena hanya menyimpan satu simpul saja yang ingin diiterasi, tetapi solusinya tidak optimal dan bahkan bisa tidak menemukan solusi. Algoritma A* menggabungkan kelebihan dan kekurangan dari kedua algoritma untuk mendapat *middle ground* dengan solusi yang optimal, tetapi tidak memboroskan waktu melakukan pencarian kepada kata yang terlihat menjauh dari target. Oleh karena itu, Algoritma A* memiliki waktu eksekusi di antara UCS dan GBFS serta penggunaan memori yang juga berada di antaranya. Namun, menariknya algoritma A* dapat dibuat lebih mendekati UCS atau lebih mendekati GBFS dengan melakukan pembobotan terhadap *heuristic* h(n). Apabila, *heuristic* h(n) diberikan pembobotan < 1 (misal 0.5), maka algoritma akan lebih mempertimbangkan *cost* daripada *heuristic* sehingga lebih mendekati UCS yang sama sekali tidak mempertimbangkan *heuristic*. Sebaliknya, jika *heuristic* h(n) diberikan faktor perkalian > 1 seperti 2x atau 3x, maka algoritma A* akan lebih mendekati GBFS yang hanya mempertimbangkan *heuristic* tanpa mempertimbangkan *cost*. Hal ini terlihat pada tabel pengujian heuristic, yang menunjukkan bahwa seiring meningkatnya faktor pengali waktu eksekusi berkurang dan panjang solusi juga cenderung meningkat. Dengan begitu, kita bisa menyesuaikan algoritma A* sesuai kebutuhan kita, dengan mengecilkan faktor pengali h(n) kita mendapat algoritma yang lebih lama, tetapi dengan *heuristic* yang admissible akan selalu memberikan solusi optimal. Sedangkan, dengan memperbesar faktor pengali *heuristic* h(n) kita bisa mendapat algoritma yang lebih cepat, tetapi belum tentu optimal.

BAB V
KESIMPULAN DAN SARAN

Berdasarkan implementasi program dan hasil pengujian, dapat disimpulkan bahwa dalam permasalahan *pathfinding* word ladder, algoritma A* merupakan algoritma yang paling fleksibel jika dibandingkan dengan algoritma UCS dan Greedy Best-First Search. Sebab, dengan melakukan pengaturan bobot *Heuristic* kita dapat mengorbankan kecepatan untuk

LAMPIRAN

Link Repository : https://github.com/abdulrafirh/Tucil3_13522089

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

DAFTAR PUSTAKA

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf