

Tugas Kecil 3 IF2211 Strategi Algoritma
Implementasi Algoritma UCS, A*, dan Greedy Best First Search
dalam Permainan Word Ladder



Disusun oleh:
13522024 - Kristo Anugrah

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG
2023

DAFTAR ISI

DAFTAR ISI	1
BAB I	2
DESKRIPSI MASALAH	2
1.1 Deskripsi Masalah	2
BAB II	3
DASAR TEORI	3
2.1 Algoritma Uniform Cost Search	3
2.2 Algoritma A*	3
2.3 Algoritma Greedy Best First Search	4
BAB III	5
ANALISIS ALGORITMA	5
3.1 Terminologi	5
3.2 Analisis Tahap Preprocessing	5
3.3 Analisis Algoritma Uniform Cost Search	6
3.4 Analisis Algoritma A*	6
3.5 Analisis Algoritma Greedy Best First Search	7
BAB IV	9
SOURCE CODE DAN IMPLEMENTASI	9
4.1 Source Code dan Implementasi	9
BAB V	13
PENGUJIAN	13
5.1 Pengujian	13
BAB VI	44
ANALISIS PENGUJIAN	44
6.1 Tabel Agregasi Pengujian	44
6.2 Analisis Hasil Pengujian berdasarkan Optimalitas	44
6.3 Analisis Hasil Pengujian berdasarkan Memori	45
6.4 Analisis Hasil Pengujian berdasarkan Waktu Eksekusi	45
BAB VII	46
IMPLEMENTASI BONUS	46
7.1 Gambaran Umum Implementasi Bonus	46
7.2 Class Gui	46
7.3 Class InputBox	46
7.4 Class ResultDialog	46
LAMPIRAN	48

BAB I

DESKRIPSI MASALAH

1.1 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.

How To Play

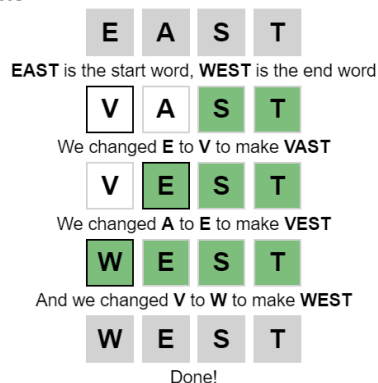
This game is called a "word ladder" and was invented by Lewis Carroll in 1877.

Rules

Weave your way from the start word to the end word.

Each word you enter **can only change 1 letter** from the word above it.

Example



Gambar 1.1.1 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 II

DASAR TEORI

2.1 Algoritma Uniform Cost Search

Algoritma *Uniform Cost Search* (UCS) merupakan algoritma *route planning* yang bersifat *uninformed search* atau *blind search*. Hal ini artinya algoritma ini tidak menggunakan informasi tambahan mengenai simpul yang telah diraih dan simpul hidup yang dipertimbangkan. Algoritma ini hanya mempertimbangkan jumlah bobot dari jalur yang telah dilalui saat ini. Algoritma ini akan menemukan *shortest path* (jalur terpendek) dari simpul awal ke simpul akhir dalam sebuah *weighted graph* (graf berbobot).

Algoritma ini akan menyimpan *list* simpul hidup terurut dari bobot setiap simpul. Algoritma ini juga akan mencatat seluruh simpul yang telah dilalui. Dalam setiap iterasi, algoritma akan memilih sebuah simpul dengan bobot terkecil (jika ada beberapa simpul dengan bobot terkecil, akan dipilih salah satu simpul sembarang). Kemudian, akan dicek apakah simpul ini sudah pernah dilewati sebelumnya. Apabila sudah, maka simpul ini akan diabaikan dan iterasi dilanjutkan. Jika belum, algoritma akan memasukkan seluruh simpul tetangga ke dalam simpul hidup. Bobot simpul tetangga dihitung dari bobot jalur saat ini ditambah dengan bobot sisi ke simpul tetangga. Iterasi dilakukan hingga tidak ada simpul hidup atau simpul tujuan ditemukan.

Jika dimisalkan $f(n)$ sebagai bobot suatu simpul yang dipertimbangkan oleh algoritma UCS, maka $f(n) = g(n)$, dengan $g(n)$: jarak dari simpul awal ke simpul ke- n . Dengan fungsi $g(n)$, algoritma UCS dalam setiap iterasinya akan mempertimbangkan seluruh simpul hidup dan memilih simpul dengan $f(n)$ minimal.

2.2 Algoritma A*

Algoritma A* (dibaca: A-star), merupakan algoritma *route planning* yang bersifat *informed search* atau *heuristic search*. Hal ini artinya algoritma ini akan menggunakan informasi tambahan mengenai simpul yang telah diraih dan simpul hidup yang dipertimbangkan. Algoritma ini tidak hanya mempertimbangkan bobot dari jalur yang telah dilalui saat ini; algoritma ini juga akan menggunakan sebuah fungsi $h(n)$ dalam melakukan pertimbangan. Sama seperti UCS, algoritma ini akan menemukan *shortest path* (jalur terpendek) dari simpul awal ke simpul akhir dalam sebuah *weighted graph* (graf berbobot).

Pada A*, selain bobot jalur digunakan juga fungsi heuristik $h(n)$ dalam menentukan bobot simpul. Fungsi heuristik ini digunakan untuk memberikan informasi tambahan mengenai jarak dari simpul ke simpul akhir.

Sama seperti algoritma UCS, A* juga menyimpan *list* simpul hidup dan mencatat seluruh simpul yang telah dilalui. Dalam setiap iterasi, A* akan memilih simpul dengan bobot terkecil. Simpul tersebut pertama dicek apakah sudah dilewati

atau belum. Jika sudah, maka simpul akan diabaikan dan jika belum simpul tersebut diekspan. Bobot simpul tetangga akan dihitung dengan mempertimbangkan jumlah bobot dari jalur serta nilai fungsi heuristik.

Jika dimisalkan $f(n)$ sebagai bobot suatu simpul yang dipertimbangkan oleh algoritma A^* , maka $f(n) = g(n) + h(n)$, dengan $g(n)$: jarak dari simpul awal ke simpul ke- n dan $h(n)$: fungsi heuristik dari sebuah simpul. Algoritma A^* akan selalu menemukan jalur terpendek dari simpul awal dan simpul akhir jika fungsi $h(n)$ *admissible*. Misalkan $h^*(n)$ merupakan jarak sebenarnya dari sebuah simpul ke simpul akhir, maka sebuah fungsi heuristik $h(n)$ dikatakan *admissible* jika $h(n) \leq h^*(n)$ berlaku untuk seluruh simpul.

2.3 Algoritma Greedy Best First Search

Mirip seperti A^* , algoritma *Greedy Best First Search* (GBFS) merupakan algoritma *route planning* yang bersifat *informed search* atau *heuristic search*. Hal ini artinya algoritma ini akan menggunakan informasi tambahan mengenai simpul yang telah diraih dan simpul hidup yang dipertimbangkan. Perbedaan utama GBFS dengan A^* adalah fungsi $f(n)$ yang digunakan. Dalam A^* , $f(n) = g(n) + h(n)$, namun dalam GBFS $f(n) = h(n)$. Ini artinya GBFS sepenuhnya menggunakan fungsi heuristik dalam mempertimbangkan bobot suatu simpul.

Dalam algoritma GBFS, hanya disimpan simpul yang saat ini dikunjungi dan simpul sebelumnya. Algoritma GBFS tidak memiliki *list* simpul hidup dan catatan dari seluruh simpul yang telah dikunjungi. Hal ini karena GBFS hanya akan mempertimbangkan 1 simpul dalam suatu waktu. Sebelum suatu simpul diekspan, akan dicek apakah simpul tersebut sama dengan simpul sebelumnya. Jika ya, maka kita tahu bahwa kita terjebak dalam sebuah *local minima*, sehingga algoritma akan (dengan keliru) mengasumsikan bahwa tidak ada jalur antara simpul awal dan akhir. Jika simpul tidak sama, maka akan dicari sebuah simpul tetangga dengan nilai $h(n)$ terkecil, dan kemudian eksplorasi dilanjutkan ke simpul tersebut.

Algoritma GBFS tidak selalu menemukan jalur optimal dari simpul awal dan akhir, atau dengan kata lain algoritma ini *non-complete*. Hal ini karena GBFS dapat terjebak dalam *local minima*. Selain itu, GBFS bersifat *irrevocable*, karena proses GBFS tidak dapat diulang/diubah (tidak ada proses *backtracking*).

BAB III

ANALISIS ALGORITMA

3.1 Terminologi

Berikut adalah beberapa terminologi beserta definisinya yang akan digunakan sepanjang makalah ini:

1. Fungsi $f(n)$: fungsi yang mengestimasi jumlah bobot dari jalur yang melewati simpul ke- n menuju simpul akhir.
2. Fungsi $g(n)$: fungsi bernilai sama dengan jumlah bobot dari jalur yang telah dilewati menuju simpul ke- n .
3. Fungsi $h(n)$: fungsi heuristik yang memperkirakan jarak dari simpul ke- n menuju simpul akhir.
4. Fungsi $h^*(n)$: fungsi bernilai sama dengan jarak terdekat sebenarnya dari simpul ke- n menuju simpul akhir.
5. *Admissible*: suatu fungsi heuristik $h(n)$ dikatakan *admissible* jika $h(n) \leq h^*(n)$ untuk seluruh simpul.
6. Pemetaan masalah:
 - a. Simpul: dalam masalah ini, simpul akan dipetakan menjadi sebuah kata dalam kamus.
 - b. Sisi: dalam masalah ini, sisi dari 2 simpul A dan B ada dalam graf jika dan hanya jika kata A dan B hanya berbeda 1 huruf. Perlu diperhatikan bahwa sisi ini *weighted* dengan bobot 1.

3.2 Analisis Tahap Preprocessing

Sebelum menjalankan algoritma UCS, A*, dan GBFS, dilakukan *preprocessing* terlebih dahulu untuk memuat graf ke dalam struktur data program. Graf telah disimpan dalam dua file `"/assets/words_oracle.txt"` dan `"/assets/adjNumOracle.txt"`. File `"/assets/words_oracle.txt"` menyimpan seluruh kata dalam kamus yang akan digunakan, terurut *lexicographically*. Informasi sisi dalam graf disimpan dalam file `"/assets/adjNumOracle.txt"` dalam bentuk sebuah *adjacency list*, dimana data pada baris ke- i menandakan simpul ke- i memiliki sisi ke seluruh simpul yang ditulis. Graf kemudian akan disimpan ke *array static* (untuk informasi sisi dan kata), serta *hashmap* (untuk mendapatkan index dari sebuah kata).

Proses *preprocessing* ini membutuhkan waktu $O(KN)$, dengan N adalah banyak kata dalam kamus yang dimuat dan K adalah banyak sisi maksimum dari simpul. Penulis telah menganalisis bahwa dari [kamus_oracle](#), sebuah simpul paling banyak memiliki 37 sisi, sehingga kompleksitas waktu dari proses ini adalah $O(37N) = O(N)$.

Tahap *preprocessing* dilakukan supaya program tidak perlu lagi mencari seluruh sisi dalam graf, sehingga tahap ini akan mempercepat jalannya program.

3.3 Analisis Algoritma Uniform Cost Search

Langkah-langkah algoritma UCS adalah sebagai berikut:

1. Dari string awal dan akhir yang diberikan, *convert* string ke integer menggunakan *hashmap* yang telah dijelaskan di tahap *preprocessing*.
2. Inisialisasi variabel yang akan digunakan:
 - a. Set *visitedSet* untuk mencatat simpul yang telah dikunjungi.
 - b. *HashMap* *prevHashMap* untuk mencatat simpul sebelumnya yang dikunjungi sebelum mengunjungi sebuah simpul (seperti simpul *parent*).
 - c. *ArrayList* *resultArrayList* untuk menyimpan jalur yang telah ditemukan.
 - d. *Queue* *queue* untuk menyimpan simpul hidup.
3. Masukkan integer awal ke *queue*.
4. Ambil bagian elemen simpul *queue* paling depan, kemudian *pop* elemen tersebut. Jika simpul telah dikunjungi, ulangi langkah 4. Jika belum, lanjutkan ke langkah 5.
5. Jika simpul sama dengan simpul akhir, kembalikan jalur yang telah ditemukan.
6. Masukkan simpul ke *visitedSet* dan *hashmap*.
7. Untuk setiap simpul tetangga, masukkan simpul tetangga ke dalam *queue*.
8. Jika *queue* masih belum kosong, lompat ke langkah 4.

Perhatikan bahwa karena bobot setiap sisi sama dengan 1, kita tidak perlu memakai *PriorityQueue*, karena pada setiap iterasi berlaku $f(n') = f(n) + 1$. Untuk simpul A, simpul B, jika berlaku $f(A) \leq f(B)$, berlaku pula $f(B) \leq f(A')$ untuk setiap simpul tetangga A'.

Pada algoritma UCS, karena seluruh sisi dalam graf berbobot *uniform* (seluruh sisi dalam graf memiliki bobot yang sama), algoritma UCS akan membangkitkan urutan simpul serta jalur yang sama dengan algoritma BFS. Dalam algoritma BFS, nilai fungsi f dari simpul n ke simpul tetangga n' didefinisikan dengan $f(n') = f(n) + 1$. Sedangkan pada UCS, berlaku $f(n') = f(n) + c(n, n')$ dengan $c(n, n')$ merupakan bobot sisi dari simpul n ke simpul n' . Perhatikan bahwa karena dalam masalah ini $c(n, n') = 1$ untuk seluruh sisi pada graf yang menghubungkan simpul n dan n' , didapat fungsi f yang sama untuk algoritma BFS dan UCS. Karena BFS dan UCS juga memiliki jalur algoritma yang sama, disimpulkan UCS dalam masalah ini sama dengan BFS.

Algoritma ini memiliki kompleksitas waktu $O(E + V)$, dengan E banyak sisi yang dilalui hingga simpul tujuan ditemukan dan V banyak simpul yang dilalui hingga simpul tujuan ditemukan. Kompleksitas waktunya adalah $O(V)$, karena hanya disimpan himpunan seluruh simpul.

3.4 Analisis Algoritma A*

Langkah-langkah algoritma A* adalah sebagai berikut:

1. Dari string awal dan akhir yang diberikan, *convert* string ke integer menggunakan *hashmap* yang telah dijelaskan di tahap *preprocessing*.

2. Inisialisasi variabel yang akan digunakan:
 - a. Set `visitedSet` untuk mencatat simpul yang telah dikunjungi.
 - b. `HashMap` `prevHashMap` untuk mencatat simpul sebelumnya yang dikunjungi sebelum mengunjungi sebuah simpul (seperti simpul *parent*).
 - c. `ArrayList` `resultArrayList` untuk menyimpan jalur yang telah ditemukan.
 - d. `PriorityQueue` `pQueue` untuk menyimpan simpul hidup.
3. Masukkan integer awal ke `pQueue`.
4. Ambil bagian elemen simpul `pQueue` paling depan, kemudian pop elemen tersebut. Jika simpul telah dikunjungi, ulangi langkah 4. Jika belum, lanjutkan ke langkah 5.
5. Jika simpul sama dengan simpul akhir, kembalikan jalur yang telah ditemukan.
6. Masukkan simpul ke `visitedSet` dan *hashmap*.
7. Untuk setiap simpul tetangga, masukkan simpul tetangga ke dalam `pQueue`.
8. Jika `pQueue` masih belum kosong, lompat ke langkah 4.

Pada algoritma A^* , digunakan fungsi heuristik $h(n)$ sebagai jarak *hamming distance* antara 2 *string*. *Hamming distance* dari 2 *string* didefinisikan sebagai banyak posisi dimana karakter pada kedua *string* berbeda. Misal untuk *string* "WEST" dan "EAST", *hamming distance* kedua *string* tersebut sama dengan 2, karena *string* berbeda pada posisi pertama dan kedua.

Algoritma ini memiliki kompleksitas waktu $O(E + V \log V)$ dan kompleksitas memori $O(V)$, dengan E banyak sisi dan V banyak simpul. Hal ini dikarenakan proses insersi pada struktur data `PriorityQueue` yang memiliki kompleksitas waktu $O(\log N)$, dengan N banyak elemen yang disimpan dalam struktur data tersebut.

Akan dibuktikan bahwa $h(n)$ *admissible*. Perlu dibuktikan bahwa $h(n) \leq h^*(n)$ berlaku untuk seluruh simpul. Perhatikan bahwa karena untuk setiap penambahan 1 bobot, akan ada 1 karakter yang berubah. Kemungkinan terbaik terjadi ketika setiap kata dari simpul ke- n menuju simpul terakhir ada dalam kamus. Dalam kemungkinan tersebut, mudah dilihat $h(n) = h^*(n)$. Tidak mungkin dicapai $h(n) > h^*(n)$, karena hal tersebut mengimplikasikan bahwa dalam 1 langkah ada lebih dari 1 karakter yang diubah.

Secara teoritis, A^* akan lebih optimal dibanding algoritma UCS dalam artian simpul yang dikunjungi lebih sedikit untuk mencapai simpul akhir. Hal ini karena fungsi heuristik yang digunakan dalam algoritma A^* akan "menuntun" jalannya algoritma lebih cepat ke simpul akhir.

3.5 Analisis Algoritma Greedy Best First Search

Langkah-langkah algoritma *Greedy Best First Search* adalah sebagai berikut:

1. Dari *string* awal dan akhir yang diberikan, *convert* *string* ke integer menggunakan *hashmap* yang telah dijelaskan di tahap *preprocessing*.
2. Inisialisasi variabel yang akan digunakan:
 - a. `ArrayList` `resultArrayList` untuk menyimpan jalur yang telah ditemukan

- b. Inisialisasi integer a sebagai simpul yang sedang diekspan, dan $prev$ sebagai simpul terakhir yang diekspan.
3. Ambil integer a , jika a sama dengan simpul akhir, maka kembalikan jalur yang telah ditemukan.
4. Jika a sama dengan $prev$, maka tidak ada solusi.
5. Untuk setiap simpul tetangga dari a , cari simpul tetangga dengan nilai $h(n)$ paling kecil.
6. Jika simpul yang dipilih sama dengan $prev$, maka tidak ada solusi.
7. Lakukan *assignment* a menjadi simpul tetangga yang dipilih.

Pada algoritma GBFS, digunakan fungsi heuristik $h(n)$ sebagai jarak *hamming distance* antara 2 *string*. *Hamming distance* dari 2 *string* didefinisikan sebagai banyak posisi dimana karakter pada kedua *string* berbeda. Misal untuk *string* "WEST" dan "EAST", *hamming distance* kedua *string* tersebut sama dengan 2, karena *string* berbeda pada posisi pertama dan kedua.

Algoritma GBFS memiliki kompleksitas waktu $O(K)$ dan kompleksitas memori $O(K)$, dengan K adalah jarak simpul awal ke simpul akhir yang ditemukan algoritma. Hal ini karena algoritma hanya akan mempertimbangkan 1 simpul hidup, sehingga banyak simpul yang diekspan jumlahnya sama dengan K . Memori digunakan ketika algoritma membangkitkan kembali jalur yang telah ditemukan.

Secara teoritis, algoritma GBFS tidak akan selalu mendapatkan jalur yang optimal. Hal ini dikarenakan algoritma *Greedy Best First Search* adalah algoritma *incomplete*. Algoritma ini bahkan mungkin terjebak dalam *local optima*. Contohnya adalah simpul awal "CHI" menuju simpul akhir "ATE". Ada jalur yang menghubungkan 2 simpul tersebut, dibuktikan dengan hasil algoritma A^* dan UCS. Namun, algoritma GBFS terjebak dalam sebuah *local optima*, karena GBFS secara keliru mengasumsikan tidak ada solusi.

BAB IV

SOURCE CODE DAN IMPLEMENTASI

4.1 Source Code dan Implementasi

Seluruh *source code* algoritma UCS, A*, dan GBFS terletak dalam *class Algorithm* di file *Algorithm.java*.

4.1.1 Class Variables

Merupakan variabel *class* yang akan digunakan oleh seluruh *method* dalam *class*

```
1  protected int[][] adjacency_list;
2  protected String[] words;
3  final int numWord = 370105;
4  final int maxAdj = 52;
5  public static String fileSeparator = File.separator;
6  protected HashMap<String, Integer> hashMap;
```

Atribut	Deskripsi
<code>adjacency_list</code>	Menyimpan sisi dalam graf dalam bentuk <i>adjacency_list</i>
<code>words</code>	Menyimpan kata dalam kamus dalam struktur data <i>array static</i>
<code>numWord</code>	Variabel konstan untuk jumlah kata yang terdapat dalam kamus
<code>maxAdj</code>	Variabel konstan untuk jumlah sisi maksimum yang dimiliki sebuah simpul
<code>fileSeparator</code>	<i>File separator</i> yang dependen pada sistem operasi
<code>hashMap</code>	<i>Hashmap</i> yang akan digunakan untuk mengubah sebuah string menjadi index

4.1.2 Method UCS

Merupakan *method* untuk menjalankan algoritma UCS

```
1 public Pair<ArrayList<String>, Pair<Integer, Integer>> UCS(String s1, String s2){
2     try {
3         int a = hashMap.get(s1);
4         int b = hashMap.get(s2);
5         final long start = System.nanoTime();
6         Set<Integer> visitedSet = new HashSet<Integer>();
7         HashMap<Integer, Integer> prevHashMap = new HashMap<Integer, Integer>();
8         ArrayList<String> resultArrayList = new ArrayList<String>();
9         Queue<Pair<Integer, Integer>> queue = new LinkedList<>();
10        queue.add(new Pair<Integer, Integer>(a, -1));
11        while(!queue.isEmpty()){
12            Pair<Integer, Integer> topQPair = queue.peek();
13            queue.remove();
14            if(visitedSet.contains(topQPair.first)) continue;
15            visitedSet.add(topQPair.first);
16            prevHashMap.put(topQPair.first, topQPair.second);
17            if(topQPair.first == b) {
18                int back = topQPair.first;
19                while(back != -1) {
20                    resultArrayList.add(words[back]);
21                    back = prevHashMap.get(back);
22                }
23                final long endtime = System.nanoTime();
24                long seconds = TimeUnit.NANOSECONDS.toMillis(endtime - start);
25                return new Pair<ArrayList<String>, Pair<Integer, Integer>>(resultArrayList, new Pair<Integer, Integer>((int)seconds, visitedSet.size()));
26            }
27            for(int i = 0; i < maxAdj && adjacency_list[topQPair.first][i] != -1; i++) {
28                queue.add(new Pair<Integer, Integer>(adjacency_list[topQPair.first][i], topQPair.first));
29            }
30        }
31        return new Pair<ArrayList<String>, Pair<Integer, Integer>>(resultArrayList, new Pair<Integer, Integer>(-1, -1));
32    } catch (Exception e) {
33        return new Pair<ArrayList<String>, Pair<Integer, Integer>>(new ArrayList<String>(), new Pair<Integer, Integer>(-1, -1));
34    }
35 }
```

Atribut	Deskripsi
visitedSet	Set untuk menyimpan seluruh simpul yang telah dikunjungi
prevHashMap	HashMap untuk menyimpan simpul <i>parent</i> dari suatu simpul
resultArrayList	ArrayList untuk menyimpan solusi dari jalur yang ditemukan
queue	Interface queue yang diimplementasikan oleh kelas LinkedList. Digunakan untuk menyimpan seluruh simpul hidup
a, b	Integer yang digunakan dalam proses iterasi. A adalah integer hasil <i>convert</i> dari simpul awal, sedangkan b adalah hasil <i>convert</i> dari simpul akhir
topQPair	Temporary variable untuk mengambil variabel paling atas dari queue
start, endtime, seconds	Variabel untuk menghitung waktu eksekusi algoritma

4.1.3 Method Astar

Merupakan *method* untuk menjalankan algoritma A*

```
1 public Pair<ArrayList<String>, Pair<Integer, Integer>> Astar(String s1, String s2){
2     try {
3         PriorityQueue<Pair<Integer, Pair<Integer, Integer>>> pQueue = new PriorityQueue<Pair<Integer, Pair<Integer, Integer>>>((a, b) -> a.first - b.first);
4         int a = hashMap.get(s1);
5         int b = hashMap.get(s2);
6         final long start = System.nanoTime();
7         Set<Integer> visitedSet = new HashSet<Integer>();
8         HashMap<Integer, Integer> prevHashMap = new HashMap<Integer, Integer>();
9         ArrayList<String> resultArrayList = new ArrayList<String>();
10        pQueue.add(new Pair<Integer, Pair<Integer, Integer>>(hammingDistance(s1, s2), new Pair<Integer, Integer>(a, -1)));
11        while(!pQueue.isEmpty()) {
12            Pair<Integer, Pair<Integer, Integer>> topQPair = pQueue.peek();
13            pQueue.remove();
14            if(visitedSet.contains(topQPair.second.first)) continue;
15            topQPair.first = hammingDistance(words[topQPair.second.first], s2);
16            //System.out.println("Value: " + topQPair.first);
17            visitedSet.add(topQPair.second.first);
18            prevHashMap.put(topQPair.second.first, topQPair.second.second);
19            if(topQPair.second.first == b) {
20                int back = topQPair.second.first;
21                while(back != -1) {
22                    resultArrayList.add(words[back]);
23                    back = prevHashMap.get(back);
24                }
25                final long endTime = System.nanoTime();
26                long seconds = TimeUnit.MILLISECONDS.toMillis(endTime - start);
27                return new Pair<ArrayList<String>, Pair<Integer, Integer>>(resultArrayList, new Pair<Integer, Integer>((int)seconds, visitedSet.size()));
28            }
29            for(int i = 0; i < maxAdj; i++) {
30                pQueue.add(new Pair<Integer, Pair<Integer, Integer>>(topQPair.first + 1 + hammingDistance(words[adjacency_list[topQPair.second.first][i]], s2), new Pair<Integer, Integer>(adjacency_list[topQPair.second.first][i], topQPair.second.first)));
31            }
32        }
33        return new Pair<ArrayList<String>, Pair<Integer, Integer>>(resultArrayList, new Pair<Integer, Integer>(-1, -1));
34    } catch (Exception e) {
35        return new Pair<ArrayList<String>, Pair<Integer, Integer>>(new ArrayList<String>(), new Pair<Integer, Integer>(-1, -1));
36    }
37 }
38 }
```

Atribut	Deskripsi
visitedSet	Set untuk menyimpan seluruh simpul yang telah dikunjungi
prevHashMap	HashMap untuk menyimpan simpul <i>parent</i> dari suatu simpul
resultArrayList	ArrayList untuk menyimpan solusi dari jalur yang ditemukan
pQueue	Kelas <i>PriorityQueue</i> untuk menyimpan simpul hidup
a, b	Integer yang digunakan dalam proses iterasi. A adalah integer hasil <i>convert</i> dari simpul awal, sedangkan b adalah hasil <i>convert</i> dari simpul akhir
topQPair	<i>Temporary variable</i> untuk mengambil variabel paling atas dari queue
start, endTime, seconds	Variabel untuk menghitung waktu eksekusi algoritma

4.1.4 Method GBFS

Merupakan *method* untuk menjalankan algoritma *Greedy Best First Search*

```
1 public Pair<ArrayList<String>, Pair<Integer, Integer>> GBFS(String s1, String s2) {
2     try {
3
4         int a = hashMap.get(s1);
5         int b = hashMap.get(s2);
6         int prev = -1;
7         ArrayList<String> resultArrayList = new ArrayList<String>();
8         long startSeconds = System.nanoTime();
9         while(a != b) {
10             resultArrayList.add(words[a]);
11             //System.out.println(a);
12             int dest = -1;
13             int maxi = Integer.MAX_VALUE;
14             for(int i = 0; i < maxAdj && adjacency_list[a][i] != -1; i++) {
15                 int c = hammingDistance(words[adjacency_list[a][i]], s2);
16                 if(c < maxi) {
17                     dest = adjacency_list[a][i];
18                     maxi = c;
19                 }
20             }
21             if(dest == -1 || dest == prev) {
22                 return new Pair<ArrayList<String>, Pair<Integer,Integer>>(new ArrayList<String>(), new Pair<Integer, Integer>(-1, -1));
23             }
24             prev = a;
25             a = dest;
26         }
27         resultArrayList.add(words[a]);
28         long endSeconds = System.nanoTime();
29         long seconds = TimeUnit.NANOSECONDS.toMillis(endSeconds - startSeconds);
30         Collections.reverse(resultArrayList);
31         return new Pair<ArrayList<String>, Pair<Integer,Integer>>(resultArrayList, new Pair<Integer, Integer>((int)seconds, resultArrayList.size()));
32     } catch (Exception e) {
33         return new Pair<ArrayList<String>, Pair<Integer,Integer>>(new ArrayList<String>(), new Pair<Integer, Integer>(-1, -1));
34     }
35 }
36 }
```

Atribut/Method	Deskripsi
resultArrayList	ArrayList untuk menyimpan solusi dari jalur yang ditemukan
a, b, prev	Integer yang digunakan dalam proses iterasi. A adalah integer hasil <i>convert</i> dari simpul awal, sedangkan b adalah hasil <i>convert</i> dari simpul akhir. Prev digunakan untuk menyimpan simpul sebelumnya
startSeconds, endSeconds, seconds	Variabel untuk menghitung waktu eksekusi algoritma
dest, maxi, c	Variabel sementara untuk mencari simpul tetangga untuk diekspan

BAB V

PENGUJIAN

5.1 Pengujian

1. GO → IF

Pemecah Word Ladder

PANJANG KATA:

2

MULAI:

G

O

AKHIR:

I

F

☒ UCS

☐ A*

☐ GBFS

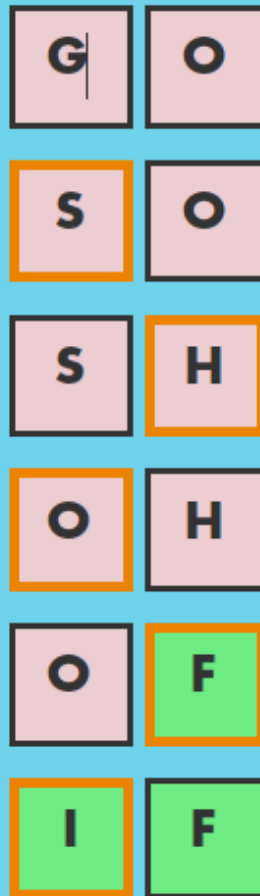
GO

UCS

Solusi:

Panjang sekuens: 6
Waktu berlalu: 2 ms
Simpul dikunjungi: 96

A*

Solusi:

Panjang sekuens: 6
Waktu berlalu: 1 ms
Simpul dikunjungi: 58

GBFS



2. CHI \rightarrow AAS

Pemecah Word Ladder

PANJANG KATA:

3

MULAI:

C

H

I

AKHIR:

A

A

S

☒ UCS ☐ A* ☐ GBFS

GO

UCS

IF2211 Strategi Algoritma

17

Solusi:

C	H	I
K	H	I
K	O	I
K	O	S
K	A	S
A	A	S

Panjang sekuens: 6
Waktu berlalu: 1 ms
Simpul dikunjungi: 372

A*

Solusi:

C	H	I
P	H	I
P	H	T
P	A	T
P	A	S
A	A	S

Panjang sekuens: 6
Waktu berlalu: 0 ms
Simpul dikunjungi: 13

GBFS



3. ATOM → SEES

Pemecah Word Ladder

—□×

PANJANG KATA:

4

MULAI:

A

T

O

M

AKHIR:

S

E

E

S

☒ UCS

☐ A*

☐ GBFS



GO

UCS

Solusi:

A	T	O	M
A	T	O	P
S	T	O	P
S	T	E	P
S	E	E	P
S	E	E	S

Panjang sekuens: 6**Waktu berlalu: 0 ms****Simpul dikunjungi: 151****A***


 Solusi 

Solusi:

A	T	O	M
A	T	O	P
S	T	O	P
S	T	E	P
S	E	E	P
S	E	E	S

Panjang sekuens: 6
Waktu berlalu: 0 ms
Simpul dikunjungi: 6

GBFS

 Solusi ×

Solusi:

A	T	O	M
A	T	O	P
S	T	O	P
S	T	E	P
S	E	E	P
S	E	E	S

Panjang sekuens: 6
Waktu berlalu: 0 ms
Simpul dikunjungi: 6

4. NYLON → MASON

Pemecah Word Ladder

PANJANG KATA:

5

MULAI:

N

Y

L

O

N

AKHIR:

M

A

S

O

N

☒ UCS

☐ A*

☐ GBFS

GO

UCS

Solusi:

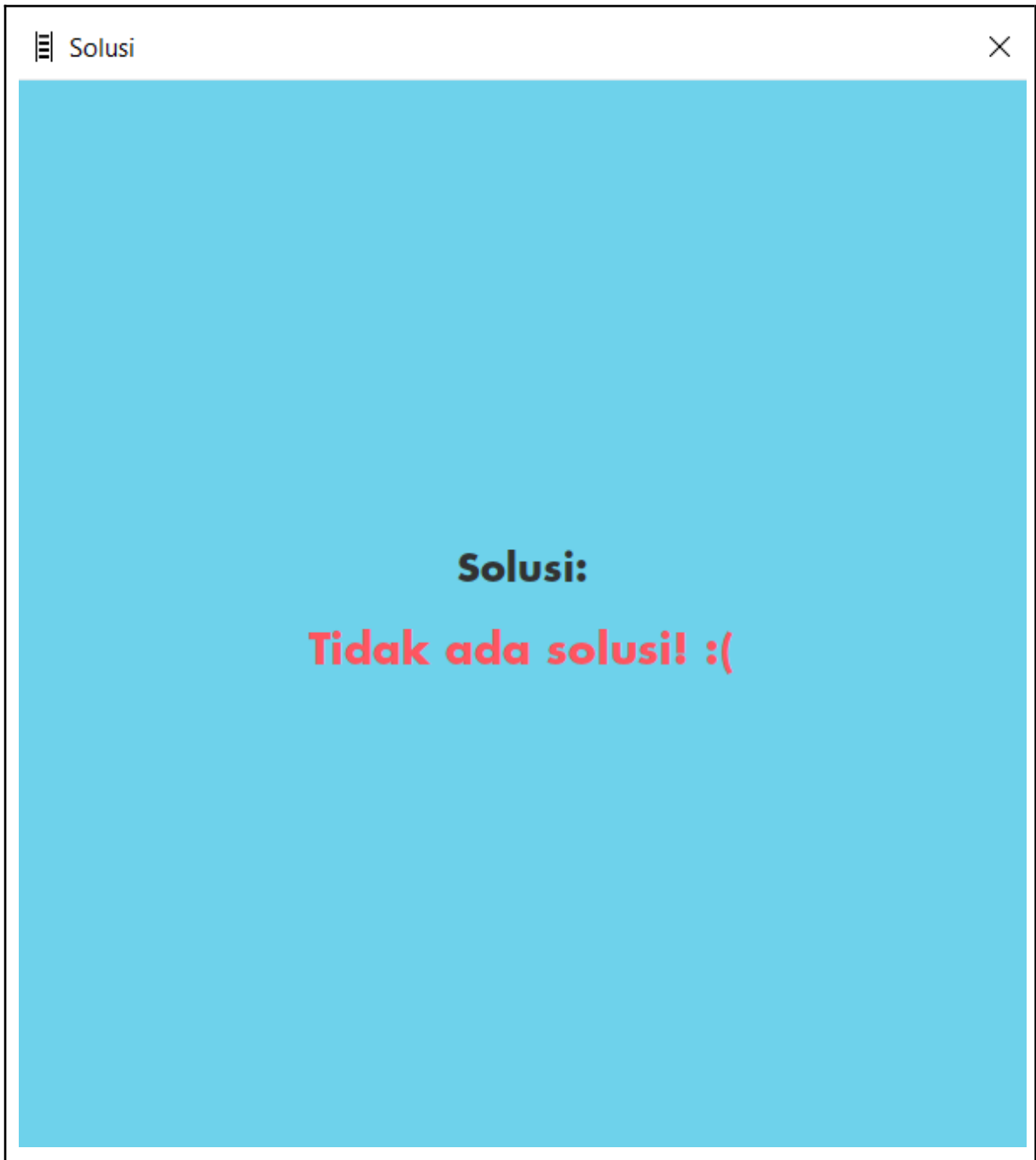
N	Y	L	O	N
P	Y	L	O	N
P	E	L	O	N
M	E	L	O	N
M	E	S	O	N
M	A	S	O	N

Panjang sekuens: 6**Waktu berlalu: 0 ms****Simpul dikunjungi: 7****A***

Solusi:

N	Y	L	O	N
P	Y	L	O	N
P	E	L	O	N
M	E	L	O	N
M	E	S	O	N
M	A	S	O	N

Panjang sekuens: 6**Waktu berlalu: 0 ms****Simpul dikunjungi: 6****GBFS**



5. BOYISH → POLISH

Pemecah Word Ladder

PANJANG KATA:

6

MULAI:

B

O

Y

I

S

H

AKHIR:

P

O

L

I

S

H

☒ UCS

☐ A*

☐ GBFS

GO

UCS

Solusi:

B	O	Y	I	S	H
T	O	Y	I	S	H
T	O	N	I	S	H
M	O	N	I	S	H
M	O	P	I	S	H
P	O	P	I	S	H
P	O	L	I	S	H

Panjang sekuens: 7**Waktu berlalu: 0 ms****Simpul dikunjungi: 19****A***

Solusi:

B	O	Y	I	S	H
T	O	Y	I	S	H
T	O	N	I	S	H
M	O	N	I	S	H
M	O	P	I	S	H
P	O	P	I	S	H
P	O	L	I	S	H

Panjang sekuens: 7
Waktu berlalu: 0 ms
Simpul dikunjungi: 10

GBFS



6. ATLASES → WATERED

Pemecah Word Ladder

PANJANG KATA:

7

MULAI:

A

T

L

A

S

E

S

AKHIR:

W

A

T

E

R

E

D

☒ UCS

☐ A*

☐ GBFS

GO

UCS

IF2211 Strategi Algoritma

33

Solusi:

A	T	L	A	S	E	S
A	N	L	A	S	E	S
A	N	L	A	C	E	S
U	N	L	A	C	E	S
U	N	L	A	D	E	S
U	N	L	A	D	E	D
U	N	F	A	D	E	D
U	N	F	A	K	E	D
U	N	C	A	K	E	D
U	N	C	A	S	E	D
U	N	C	A	S	E	S
U	N	E	A	S	E	S

U	N	E	A	S	E	S
U	R	E	A	S	E	S
C	R	E	A	S	E	S
C	R	E	S	S	E	S
T	R	E	S	S	E	S
T	R	A	S	S	E	S
B	R	A	S	S	E	S
B	R	A	S	H	E	S
B	R	A	S	H	E	R
B	R	A	S	I	E	R
B	R	A	K	I	E	R
B	E	A	K	I	E	R

B	E	A	K	I	E	R
P	E	A	K	I	E	R
P	E	C	K	I	E	R
P	I	C	K	I	E	R
D	I	C	K	I	E	R
D	I	C	K	I	E	S
H	I	C	K	I	E	S
H	A	C	K	I	E	S
H	A	C	K	L	E	S
H	E	C	K	L	E	S
D	E	C	K	L	E	S
D	E	C	I	L	E	S
D	E	C	E	L	E	S

D	E	F	I	L	E	S
R	E	F	I	L	E	S
R	E	V	I	L	E	S
R	E	V	I	L	E	D
R	E	V	E	L	E	D
R	A	V	E	L	E	D
R	A	V	E	N	E	D
H	A	V	E	N	E	D
H	A	V	E	R	E	D
W	A	V	E	R	E	D
W	A	T	E	R	E	D

Panjang sekuens: 45

Waktu berlalu: 3 ms

Simpul dikunjungi: 7458

A*

Solusi:

A	T	L	A	S	E	S
A	N	L	A	S	E	S
A	N	L	A	C	E	S
U	N	L	A	C	E	S
U	N	L	A	C	E	D
U	N	L	A	D	E	D
U	N	F	A	D	E	D
U	N	F	A	K	E	D
U	N	C	A	K	E	D
U	N	C	A	K	E	S
U	N	C	A	S	E	S
U	N	E	A	S	E	S

U	N	E	A	S	E	S
U	R	E	A	S	E	S
C	R	E	A	S	E	S
C	R	E	S	S	E	S
C	R	O	S	S	E	S
C	R	O	S	S	E	R
C	R	O	S	I	E	R
C	R	O	Z	I	E	R
C	R	A	Z	I	E	R
B	R	A	Z	I	E	R
B	R	A	K	I	E	R
B	E	A	K	I	E	R

B	E	A	K	I	E	R
P	E	A	K	I	E	R
P	E	C	K	I	E	R
P	I	C	K	I	E	R
D	I	C	K	I	E	R
D	I	C	K	I	E	S
H	I	C	K	I	E	S
H	A	C	K	I	E	S
H	A	C	K	L	E	S
H	E	C	K	L	E	S
D	E	C	K	L	E	S
D	E	C	I	L	E	S
D	E	E	I	I	E	S

D	E	F	I	L	E	S
D	E	F	I	L	E	D
D	E	V	I	L	E	D
D	E	V	E	L	E	D
R	E	V	E	L	E	D
R	A	V	E	L	E	D
R	A	V	E	N	E	D
H	A	V	E	N	E	D
H	A	V	E	R	E	D
W	A	V	E	R	E	D
W	A	T	E	R	E	D

Panjang sekuens: 45

Waktu berlalu: 17 ms

Simpul dikunjungi: 6675



BAB VI

ANALISIS PENGUJIAN

6.1 Tabel Agregasi Pengujian

Waktu Eksekusi Algoritma

No.	Waktu Eksekusi (ms)		
	UCS	A*	GBFS
1.	2	1	-
2.	1	0	-
3.	0	0	0
4.	0	0	-
5.	0	0	-
6.	3	17	-

Tabel 6.1.1 Waktu eksekusi algoritma

Banyak Simpul yang Dikunjungi Algoritma

No.	Waktu Eksekusi (ms)		
	UCS	A*	GBFS
1.	96	58	-
2.	372	13	-
3.	151	6	6
4.	7	6	-
5.	19	10	-
6.	7458	6675	-

Tabel 6.1.2 Banyak simpul yang dikunjungi algoritma

6.2 Analisis Hasil Pengujian berdasarkan Optimalitas

Dari hasil pengujian pada bab 5.1 mudah dilihat bahwa algoritma *Uniform Cost Search* dan A* selalu menemukan solusi optimal jika terdapat jalur dari simpul awal menuju simpul akhir. Dapat diobservasi bahwa panjang sekuens yang ditemukan algoritma UCS dan A* memiliki panjang yang sama, hal tersebut merupakan reasuransi bahwa kedua algoritma berjalan sesuai desain. Sesuai teori, algoritma

GBFS tidak selalu memberikan algoritma optimal, dan bahkan mengasumsikan bahwa tidak ada jalur dari simpul awal ke simpul akhir.

6.3 Analisis Hasil Pengujian berdasarkan Memori

Dalam analisis ini, dianggap memori yang digunakan berbanding lurus dengan banyak simpul yang dikunjungi. Dari tabel 6.1.2, mudah dilihat bahwa algoritma A* menggunakan lebih sedikit memori dibandingkan algoritma UCS. Hal ini artinya fungsi heuristik yang digunakan membantu algoritma A* untuk menemukan jalur ke simpul akhir lebih efisien. Pada algoritma GBFS, karena banyak simpul yang dikunjungi pasti sama dengan panjang jalur yang ditemukan, besar memori yang digunakan tidak besar.

6.4 Analisis Hasil Pengujian berdasarkan Waktu Eksekusi

Dari tabel 6.1.1 dan bab 5.1, mudah dilihat bahwa waktu eksekusi algoritma A* dan UCS relatif mirip untuk panjang jalur pendek. Namun, pada pengujian ke-6, dengan panjang jalur 45 terlihat ada perbedaan waktu eksekusi yang signifikan antara A* dan UCS. Walaupun banyak simpul yang dikunjungi oleh A* lebih sedikit, A* membutuhkan waktu eksekusi lebih lama. Hal ini karena A* menggunakan struktur data `PriorityQueue` dengan kompleksitas proses insersi sebesar $O(\log N)$, dengan N banyak elemen yang disimpan oleh `PriorityQueue` saat itu. Dibandingkan dengan algoritma UCS yang menggunakan struktur data `Queue` dengan proses insersi dan *removal* $O(1)$, algoritma A* membutuhkan waktu eksekusi lebih lama. Untuk algoritma GBFS, algoritma berjalan cepat karena tidak ada proses pengulangan yang terjadi (sifat *irrevocable* dari algoritma *Greedy Best First Search*).

BAB VII

IMPLEMENTASI BONUS

7.1 Gambaran Umum Implementasi Bonus

Graphical User Interface (GUI) diimplementasikan dengan *library* Swing. Implementasi bonus sepenuhnya dilakukan dalam *file* Gui.java. Dalam *file* ini, kewajiban pembuatan GUI dibagi ke dalam beberapa kelas utama.

7.2 Class Gui

Merupakan kelas utama yang menginisialisasi komponen *graphical user interface*.

Atribut/Method Utama	Deskripsi
<code>algorithm</code>	Merupakan <i>instance</i> dari kelas Algorithm dan digunakan untuk menjalankan algoritma
<code>void start()</code>	Metode utama yang akan menginisialisasi seluruh komponen GUI dan memanggil kelas lain dalam Gui.java

7.3 Class InputBox

Merupakan kelas yang akan menginisialisasi *box input* yang akan digunakan saat *input* dan juga *output*.

Atribut/Method Utama	Deskripsi
<code>InputBox(int n)</code>	Konstruktor kelas yang menginisialisasikan sebuah <i>input box</i> dengan panjang <i>n</i>
<code>InputBox(String, String, int)</code>	Konstruktor kelas yang akan menginisialisasikan sebuah <i>input box</i> yang berisi <i>string input</i> . Digunakan untuk menampilkan hasil jalur yang ditemukan.
<code>void change(int n)</code>	Digunakan untuk mengubah panjang dari suatu <i>input box</i> menjadi <i>n</i> .
<code>String getString()</code>	Metode untuk mendapatkan string yang dituliskan dalam <i>input box</i> .

7.4 Class ResultDialog

Merupakan kelas yang akan menginisialisasi *dialog* dan segala komponen anak untuk memvisualisasikan jalur yang ditemukan.

Atribut/Method Utama	Deskripsi
ResultDialog(JFrame, List<String>, int, int)	Konstruktor kelas untuk menginisialisasi JDialog dan seluruh komponen anak dengan <i>parent</i> JFrame dengan jalur yang tersimpan dalam sebuah <i>list</i> .

LAMPIRAN

[PriorityQueue \(Java Platform SE 8 \)](#)

[UIManager \(Java Platform SE 8 \)](#)

[JComboBox \(Java Platform SE 8 \)](#)

[BorderFactory \(Java Platform SE 8 \)](#)

[JOptionPane \(Java Platform SE 8 \)](#)

[JTextField \(Java Platform SE 8 \)](#)

[Java Swing revalidate\(\) vs repaint\(\) - Stack Overflow](#)

Pranala github:

https://github.com/grst0/Tucil3_13522024

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