

LAPORAN TUGAS KECIL 3 IF2211
STRATEGI ALGORITMA
SEMESTER II TAHUN 2023/2024

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



Disusun oleh:
Andhita Naura Hariyanto (13522060)

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2023

DAFTAR ISI

DAFTAR ISI.....	1
BAB I.....	2
BAB II.....	3
2.1. Algoritma Uniform Cost Search.....	3
2.2. Algoritma Greedy Best First Search.....	4
2.3. Algoritma A*.....	6
BAB III.....	10
3.1. Astar.java.....	10
3.2. GBFS.java.....	11
3.3. UCS.java.....	13
3.4. Result.java.....	15
3.5. Input.java.....	15
3.6. Node.java.....	16
3.7. DictionaryChecker.java.....	19
3.8. Main.java.....	19
BAB IV.....	21
4.1. Test Case 1.....	21
4.2. Test Case 2.....	22
4.3. Test Case 3.....	23
4.4. Test Case 4.....	24
4.5. Test Case 5.....	27
4.6. Test Case 6.....	28
BAB V.....	30
LAMPIRAN.....	33
DAFTAR PUSTAKA.....	34

BAB I

DESKRIPSI MASALAH

Word ladder (juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragrams, laddergrams, atau word golf) adalah salah satu permainan kata yang terkenal bagi seluruh kalangan. Word ladder ditemukan oleh Lewis Carroll, seorang penulis dan matematikawan, pada tahun 1877. Pada permainan ini, pemain diberikan dua kata yang disebut sebagai start word dan end word. Untuk memenangkan permainan, pemain harus menemukan rantai kata yang dapat menghubungkan antara start word dan end word. Banyaknya huruf pada start word dan end word selalu sama. Tiap kata yang berdekatan dalam rantai kata tersebut hanya boleh berbeda satu huruf saja. Pada permainan ini, diharapkan solusi optimal, yaitu solusi yang meminimalkan banyaknya kata yang dimasukkan pada rantai kata. Berikut adalah ilustrasi serta aturan permainan.



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

BAB II

ALGORITMA UCS, GREEDY BEST FIRST SEARCH, DAN A* DALAM PENYELESAIAN MASALAH

2.1. Algoritma Uniform Cost Search

Algoritma Uniform Cost Search merupakan salah satu algoritma yang memiliki fokus menyelesaikan permasalahan pencarian rute. Algoritma ini melakukan pencarian rute dengan *blind search/uninformed search* atau tanpa adanya informasi kondisi simpul saat ini terhadap simpul target. Algoritma Uniform Cost Search akan memberikan hasil lintasan rute terpendek dengan mempertimbangkan biaya terkecil untuk mencapai target dari posisi/simpul awal. Fungsi $g(n)$ merupakan fungsi evaluasi yang digunakan algoritma Uniform Cost Search untuk memutuskan simpul mana yang akan dilakukan ekspansi. Penentuan fungsi $g(n)$ ini didasarkan pada biaya lintasan untuk mencapai tiap simpul dari simpul awal. Penyelesaian masalah pencarian lintasan kata-kata terpendek dalam menyelesaikan permasalahan *Word Ladder* dengan algoritma Uniform Cost Search adalah sebagai berikut :

1. Tetapkan simpul awal dan simpul tujuan, dalam hal ini berarti konstruksi simpul dari kata awal dan kata yang menjadi target pencarian dari permainan Word Ladder.
2. Konstruksi antrian untuk menyimpan simpul-simpul hasil ekspansi simpul yang akan dikunjungi menggunakan struktur data *priority queue* dengan pengurutan posisi simpul didasarkan biaya berupa jarak dari simpul awal ke simpul yang sedang ditelusuri.
3. Selama *priority queue* tidak kosong, kunjungi simpul yang memiliki lintasan dengan biaya terendah dari seluruh simpul hidup yang ada dalam *priority queue* (melakukan *dequeue* pada *priority queue*).
4. Tinjau nilai (kata) dari simpul yang sedang dikunjungi.
5. Kembalikan lintasan yang berbentuk *List of String* berisikan seluruh lintasan dari simpul awal ke simpul target jika simpul yang sedang dikunjungi bernilai sama dengan simpul target.
6. Jika nilai (kata) dari simpul yang sedang dikunjungi tidak sama dengan nilai simpul tujuan, tinjau kemungkinan tetangga dari simpul

(kata) tersebut dengan melakukan pengecekan penggantian tiap huruf dalam kata dengan seluruh huruf.

7. Susunan kata yang tidak sama dengan kata awal, statusnya belum dikunjungi, dan valid dalam kamus yang digunakan akan dikonstruksi menjadi sebuah node yang berisikan informasi kata dan biaya dari simpul (kata) tersebut. Biaya suatu simpul dihitung dari berapa banyak kata atau langkah yang sudah ditempuh suatu simpul dari simpul awal.
8. Node hasil konstruksi akan dimasukkan ke dalam priority queue yang menandai tempat disimpannya simpul hidup.
9. Ulangi langkah 3-8 hingga priority queue kosong atau target kata tercapai (solusi ditemukan). Jika priority queue kosong, solusi tidak ditemukan dan program akan mengembalikan nilai null untuk lintasan beserta informasi banyaknya simpul yang dikunjungi dan waktu tempuhnya. Jika solusi ditemukan, program akan mengembalikan lintasan simpul yang dilalui program dari simpul awal hingga simpul target beserta informasi banyaknya simpul yang dikunjungi dan waktu tempuhnya.

Algoritma Uniform Cost Search dan Breadth-First Search pada kasus Word Ladder ini akan memiliki urutan pembangkitan simpul dan menghasilkan path yang sama. Hal ini dikarenakan jarak antara suatu simpul dengan seluruh simpul yang dibangkitkannya akan bernilai sama (bernilai konsisten menaik satu/*increment by one* karena fungsi evaluasi $g(n)$ didasarkan pada berapa langkah dari simpul awal menuju simpul yang sedang diekspan). Algoritma akan mengunjungi simpul-simpul sesuai urutan waktu dimasukkannya simpul tersebut ke dalam *priority queue* sesuai dengan algoritma Breadth-First Search yang akan mengunjungi seluruh tetangganya terlebih dahulu. Sehingga, lintasan yang dihasilkan pun akan sama.

2.2. Algoritma Greedy Best First Search

Algoritma Greedy Best First Search merupakan salah satu algoritma yang memiliki fokus menyelesaikan permasalahan pencarian rute. Berbeda dengan algoritma Uniform Cost Search, algoritma Greedy Best First Search menyelesaikan permasalahan pencarian rute dengan *heuristic search/informed search* atau dengan

adanya informasi kondisi simpul saat ini terhadap simpul target. Algoritma Greedy Best First Search akan memberikan hasil lintasan rute terpendek dengan mempertimbangkan biaya perkiraan terkecil bagi tiap simpulnya untuk mencapai simpul tujuan. Fungsi $h(n)$ merupakan fungsi evaluasi yang digunakan algoritma Uniform Cost Search untuk memutuskan simpul mana yang akan dilakukan ekspansi. Penentuan fungsi $h(n)$ ini didasarkan pada biaya perkiraan lintasan untuk mencapai simpul target dari tiap simpulnya. Penyelesaian masalah pencarian lintasan kata-kata terpendek dalam menyelesaikan permasalahan *Word Ladder* dengan algoritma Greedy Best First Search adalah sebagai berikut :

1. Tetapkan simpul awal dan simpul tujuan, dalam hal ini berarti konstruksi simpul dari kata awal dan kata yang menjadi target pencarian dari permainan Word Ladder.
2. Konstruksi antrian untuk menyimpan simpul-simpul hasil ekspansi simpul yang akan dikunjungi menggunakan struktur data *priority queue* dengan pengurutan posisi simpul didasarkan biaya perkiraan berupa jarak dari simpul awal ke simpul yang sedang ditelusuri.
3. Selama *priority queue* tidak kosong, kunjungi simpul yang memiliki lintasan dengan biaya terendah dari seluruh simpul hidup yang ada dalam *priority queue* (melakukan *dequeue* pada *priority queue*).
4. Tinjau nilai (kata) dari simpul yang sedang dikunjungi.
5. Kembalikan lintasan yang berbentuk *List of String* berisikan seluruh lintasan dari simpul awal ke simpul target jika simpul yang sedang dikunjungi bernilai sama dengan simpul target.
6. Jika nilai (kata) dari simpul yang sedang dikunjungi tidak sama dengan nilai simpul tujuan, tinjau kemungkinan tetangga dari simpul (kata) tersebut dengan melakukan pengecekan penggantian tiap huruf dalam kata tinjauan dengan seluruh huruf.
7. Susunan kata yang tidak sama dengan kata awal, statusnya belum dikunjungi, dan valid dalam kamus yang digunakan akan dikonstruksi menjadi sebuah node yang berisikan informasi kata dan biaya dari simpul (kata) tersebut. Biaya suatu simpul dihitung dari berapa banyak perbedaan huruf antara tiap kata dengan kata tujuan. Semakin sedikit

perbedaan hurufnya, semakin kecil nilai fungsi evaluasi ($h(n)$) dari kata tersebut (mengindikasikan semakin dekat jaraknya ke target).

8. Node hasil konstruksi akan dimasukkan ke dalam *priority queue* yang menandai tempat disimpannya simpul hidup.
9. Ulangi langkah 3-8 hingga *priority queue* kosong atau target kata tercapai (solusi ditemukan). Jika *priority queue* kosong, solusi tidak ditemukan dan program akan mengembalikan nilai null untuk lintasan beserta informasi banyaknya simpul yang dikunjungi dan waktu tempuhnya. Jika solusi ditemukan, program akan mengembalikan lintasan simpul yang dilalui program dari simpul awal hingga simpul target beserta informasi banyaknya simpul yang dikunjungi dan waktu tempuhnya.

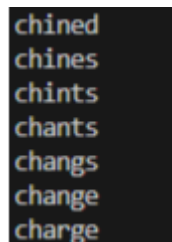
Secara teoritis, algoritma Greedy Best First Search tidak menjamin didapatkannya sebuah solusi optimal global. Algoritma akan memiliki sifat *greedy by heuristic* yang berarti setiap langkah yang diambil oleh algoritma hanya mempertimbangkan seberapa dekat jarak setiap simpul menuju simpul target tanpa mempertimbangkan sudah seberapa jauh penelusuran dilakukan karena tidak mempertimbangkan jarak dari simpul yang sedang dikunjungi dari simpul awal. Sehingga, algoritma Greedy Best First Search hanya menjamin sebuah solusi yang bersifat optimum lokal dari setiap langkah yang diambil dan tidak menjamin terhasilkannya sebuah solusi yang bersifat optimum global.

2.3. Algoritma A*

Algoritma A* merupakan bentuk algoritma yang memiliki fokus menyelesaikan permasalahan pencarian rute dengan mempertimbangkan nilai dari suatu node. Nilai yang algoritma A* gunakan untuk mencari lintasan terpendek antara dua simpul merupakan gabungan dari nilai yang algoritma Uniform Cost Search dan algoritma Greedy Best First Search terapkan untuk suatu simpul dalam mencari lintasan terpendek antara dua simpul.

Algoritma A* menyelesaikan permasalahan pencarian rute dengan heuristic search/informed search atau dengan adanya informasi kondisi simpul saat ini terhadap simpul target. Algoritma Greedy Best First Search akan memberikan hasil

lintasan rute terpendek dengan mempertimbangkan biaya perkiraan terkecil bagi tiap simpulnya untuk mencapai simpul tujuan. Fungsi $f(n)$ merupakan fungsi evaluasi yang digunakan algoritma A* untuk memutuskan simpul mana yang akan dilakukan ekspansi. Penentuan fungsi $f(n)$ ini didasarkan pada penjumlahan biaya lintasan untuk mencapai tiap simpul dari simpul awal ($g(n)$) dan biaya perkiraan lintasan untuk mencapai simpul target dari tiap simpulnya ($h(n)$). Heuristik yang digunakan oleh algoritma A* ini, yaitu biaya perkiraan lintasan untuk mencapai simpul target dari tiap simpulnya ($h(n)$), bersifat *admissible* karena memiliki nilai perkiraan jarak menuju simpul target yang pasti akan selalu lebih kecil daripada nilai jarak menuju simpul target yang sebenarnya. Misalnya dari sebuah kata *chined* menuju *charge*, dalam pencarian dengan heuristik, jarak yaitu banyak perbedaan huruf antara kedua kata bernilai 4, sedangkan pada kondisi pencarian yang sebenarnya, *chines* harus melewati 6 kata terlebih dahulu sebelum mencapai kata *charge* seperti yang digambarkan pada tangkapan layar hasil pencarian dua kata dengan algoritma A* berikut.



A screenshot of a word ladder search interface. It shows a vertical list of words: 'chined', 'chines', 'chints', 'chants', 'changs', 'change', and 'charge'. The words are displayed in a light blue font on a dark background. The word 'chined' is at the top, and 'charge' is at the bottom, indicating the path of the search.

Penyelesaian masalah pencarian lintasan kata-kata terpendek dalam menyelesaikan permasalahan *Word Ladder* dengan algoritma A* adalah sebagai berikut :

1. Tetapkan simpul awal dan simpul tujuan, dalam hal ini berarti konstruksi simpul dari kata awal dan kata yang menjadi target pencarian dari permainan Word Ladder.
2. Konstruksi antrian untuk menyimpan simpul-simpul hasil ekspansi simpul yang akan dikunjungi menggunakan struktur data *priority queue* dengan pengurutan posisi simpul didasarkan biaya berupa jarak dari simpul awal ke simpul yang sedang ditelusuri.

3. Selama *priority queue* tidak kosong, kunjungi simpul yang memiliki lintasan dengan biaya terendah dari seluruh simpul hidup yang ada dalam *priority queue* (melakukan *dequeue* pada *priority queue*).
4. Tinjau nilai (kata) dari simpul yang sedang dikunjungi.
5. Kembalikan lintasan yang berbentuk *List of String* berisikan seluruh lintasan dari simpul awal ke simpul target jika simpul yang sedang dikunjungi bernilai sama dengan simpul target.
6. Jika nilai (kata) dari simpul yang sedang dikunjungi tidak sama dengan nilai simpul tujuan, tinjau kemungkinan tetangga dari simpul (kata) tersebut dengan melakukan pengecekan penggantian tiap huruf dalam kata dengan seluruh huruf.
7. Susunan kata yang tidak sama dengan kata awal, statusnya belum dikunjungi, dan valid dalam kamus yang digunakan akan konstruksi menjadi sebuah node yang berisikan informasi kata dan biaya dari simpul (kata) tersebut. Biaya suatu simpul ($f(n)$) dihitung dari penjumlahan banyak kata atau langkah yang sudah ditempuh suatu simpul dari simpul awal ($g(n)$) dan banyak perbedaan huruf antara tiap kata dengan kata tujuan ($h(n)$).
8. Node hasil konstruksi akan dimasukkan ke dalam *priority queue* yang menandai tempat disimpannya simpul hidup.
9. Ulangi langkah 3-8 hingga *priority queue* kosong atau target kata tercapai (solusi ditemukan). Jika *priority queue* kosong, solusi tidak ditemukan dan program akan mengembalikan nilai null untuk lintasan beserta informasi banyaknya simpul yang dikunjungi dan waktu tempuhnya. Jika solusi ditemukan, program akan mengembalikan lintasan simpul yang dilalui program dari simpul awal hingga simpul target beserta informasi banyaknya simpul yang dikunjungi dan waktu tempuhnya.

Secara teoritis, algoritma A* akan menghasilkan solusi dengan lebih efisien dibandingkan dengan algoritma Uniform Cost Search. Selain mempertimbangkan jarak dari simpul awal ke simpul yang sedang dikunjungi, algoritma A* juga mempertimbangkan penambahan nilai heuristik yaitu jarak dari tiap simpul ke simpul

target sebagai nilai dari fungsi evaluasinya. Pada simpul yang berada dalam level sama, nilai cost yang dihasilkan akan bernilai sama, namun fungsi evaluasi akhir akan berbeda karena fungsi evaluasi menambahkan nilai heuristik. Simpul yang berada dalam level sama namun memiliki jarak yang lebih dekat ke simpul target atau nilai heuristik yang lebih kecil tentunya akan berada pada posisi lebih dekat untuk dikunjungi pada *priority queue*. Hal ini mendukung peningkatan efisiensi pencarian solusi karena algoritma A* akan langsung mengunjungi simpul yang selain memang relatif dekat dengan simpul awal, tentunya juga relatif dekat dengan simpul target.

BAB III

IMPLEMENTASI PROGRAM

Algoritma untuk menyelesaikan pencarian lintasan antara kedua simpul atau dua masukan kata dalam permasalahan permainan Word Ladder yang sudah dijelaskan pada Bab II akan diimplementasikan dengan menggunakan bahasa Java. Program terdapat pada folder “src” yang terdiri dari struktur folder “Algoritma” berisikan file Astar.java, GBFS.java, dan UCS.java, struktur folder “Util” berisikan file Result.java, Input.java, Node.java, dan DictionaryChecker.java, serta program utama dalam file Main.java.

3.1. Astar.java

File Astar.java berisikan *class* Astar yang memiliki *method* berupa fungsi bernama “resulting”. Fungsi menerima input dua node sebagai parameter awal, yaitu *startWord* (hasil konstruksi node dari kata awal) dan *endWord* (hasil konstruksi node dari kata target). Fungsi “resulting” mengimplementasikan algoritma A* untuk mencari jalur terpendek dari *startWord* ke *endWord*. Fungsi menggunakan *priority queue* bawaan dari Java untuk menyimpan setiap simpul hidup hasil ekspansi simpul ekspansi dengan prioritas pengurutan didasarkan pada simpul masukan yang memiliki fungsi evaluasi terkecil. *Priority queue* diurutkan berdasarkan penjumlahan nilai dari atribut *weight* dari setiap simpul masukan yang menandakan fungsi evaluasi berupa penjumlahan banyak kata atau langkah yang sudah ditempuh suatu simpul dari simpul awal dan banyak perbedaan huruf antara tiap kata dengan kata tujuan. *Return value* dari fungsi berupa objek *Result* yang memuat informasi durasi pencarian, jumlah node dikunjungi, dan lintasan hasil.

```

1 package Algorithm;
2 import java.util.ArrayList;
3 import java.util.Collections;
4 import java.util.Comparator;
5 import java.util.HashMap;
6 import java.util.Map;
7 import java.util.PriorityQueue;
8
9 import Util.DictionaryChecker;
10 import Util.Node;
11 import Util.Result;
12
13 import java.util.List;
14
15 public class Astar {
16     public static Result resulting(Node startWord, Node endWord) {
17         Map<Node, Node> parent = new HashMap<>();
18         Map<String, Boolean> visited = new HashMap<>();
19
20         String candidateWord = null;
21         int nodeCount = 0;
22         int cost = 0;
23
24         PriorityQueue<Node> pq = new PriorityQueue<Node>(Comparator.comparingInt(Node::getWeight));
25
26         pq.add(startWord);
27
28         long startTime = System.nanoTime();
29         Node currentWord = null;
30         while (!pq.isEmpty()) {
31             currentWord = pq.poll();
32             visited.put(currentWord.getWord(), true);
33             nodeCount++;
34
35             if (currentWord.getWord().equals(endWord.getWord())) {
36                 break;
37             }
38
39             if (currentWord.getCost() == 0){
40                 cost++;
41             } else {
42                 cost = currentWord.getCost() + 1;
43             }
44
45             for (int j = 0; j < startWord.getWord().length(); j++) {
46                 for (char c = 'a'; c <= 'z'; c++) {
47                     if (j == 0) {
48                         candidateWord = c + currentWord.getWord().substring(1, startWord.getWord().length());
49                     } else if (j == currentWord.getWord().length() - 1) {
50                         candidateWord = currentWord.getWord().substring(0, j) + c;
51                     } else {
52                         candidateWord = currentWord.getWord().substring(0, j) + c + currentWord.getWord().substring(j + 1, startWord.getWord().length());
53                     }
54
55                     if (candidateWord != currentWord.getWord() && !visited.containsKey(candidateWord)) {
56                         if (DictionaryChecker.isExist(candidateWord)) {
57                             int heuristic = DictionaryChecker.differentChar(candidateWord, endWord.getWord());
58                             int weight = heuristic + cost;
59                             Node child = new Node(candidateWord, weight, heuristic, cost);
60                             parent.put(child, currentWord);
61                             pq.add(child);
62                         }
63                     }
64                 }
65             }
66         }
67
68         long endTime = System.nanoTime();
69         double runtime = (double)(endTime - startTime)/1000000;
70
71         if (pq.size() != 0) {
72             List<String> path = new ArrayList<>();
73             while (currentWord != null) {
74                 path.add(currentWord.getWord());
75                 currentWord = parent.get(currentWord);
76             }
77             Collections.reverse(path);
78             return new Result(nodeCount, runtime, path);
79         } else {
80             return new Result(nodeCount, runtime, null);
81         }
82     }
83 }
84
85

```

3.2. GBFS.java

File GBFS.java berisikan *class* GBFS yang memiliki *method* berupa fungsi bernama “resulting”. Fungsi menerima input dua node sebagai parameter awal, yaitu *startWord* (hasil konstruksi node dari kata awal) dan *endWord* (hasil

konstruksi node dari kata target). Fungsi “resulting” mengimplementasikan algoritma Greedy Best First Search untuk mencari jalur terpendek dari *startWord* ke *endWord*. Fungsi menggunakan *priority queue* bawaan dari Java untuk menyimpan setiap simpul hidup hasil ekspansi simpul ekspan dengan prioritas pengurutan didasarkan pada simpul masukan yang memiliki fungsi evaluasi terkecil. *Priority queue* diurutkan berdasarkan nilai dari atribut heuristic dari setiap simpul masukan yang menandakan fungsi evaluasi banyak perbedaan huruf antara tiap kata dengan kata tujuan. *Return value* dari fungsi berupa objek *Result* yang memuat informasi durasi pencarian, jumlah node dikunjungi, dan lintasan hasil.

```

1 package Algorithm;
2 import java.util.ArrayList;
3 import java.util.Collections;
4 import java.util.Comparator;
5 import java.util.HashMap;
6 import java.util.Map;
7 import java.util.PriorityQueue;
8
9 import Util.DictionaryChecker;
10 import Util.Node;
11 import Util.Result;
12
13 import java.util.List;
14
15 public class GBFS {
16     public static Result resulting(Node startWord, Node endWord) {
17         Map<Node, Node> parent = new HashMap<>();
18         Map<String, Boolean> visited = new HashMap<>();
19
20         String candidateWord = null;
21         int nodeCount = 0;
22
23         PriorityQueue<Node> pq = new PriorityQueue<Node>(Comparator.comparingInt(Node::getHeuristic));
24
25         pq.add(startWord);
26
27         long startTime = System.nanoTime(); // Record start time
28         Node currentWord = null;
29         while (!pq.isEmpty()) {
30             currentWord = pq.poll();
31             visited.put(currentWord.getWord(), true);
32             nodeCount++;
33
34             if (currentWord.getWord().equals(endWord.getWord())) {
35                 break;
36             }
37
38             for (int j = 0; j < startWord.getWord().length(); j++) {
39                 for (char c = 'a'; c <= 'z'; c++) {
40                     if (j == 0) {
41                         candidateWord = c + currentWord.getWord().substring(1, startWord.getWord().length());
42                     } else if (j == currentWord.getWord().length() - 1) {
43                         candidateWord = currentWord.getWord().substring(0, j) + c;
44                     } else {
45                         candidateWord = currentWord.getWord().substring(0, j) + c + currentWord.getWord().substring(j + 1, startWord.getWord().length());
46                     }
47
48                     if (candidateWord != currentWord.getWord() && !visited.containsKey(candidateWord)) {
49                         if (DictionaryChecker.isExist(candidateWord)) {
50                             int heuristic = DictionaryChecker.differentChar(candidateWord, endWord.getWord());
51                             Node child = new Node(candidateWord, 0, heuristic, 0);
52                             parent.put(child, currentWord);
53                             pq.add(child);
54                         }
55                     }
56                 }
57             }
58         }
59
60         long endTime = System.nanoTime(); // Record end time
61         double runtime = (double)(endTime - startTime)/1000000; // Calculate runtime in nanoseconds
62
63         if (pq.size() != 0) {
64             List<String> path = new ArrayList<>();
65             while (currentWord != null) {
66                 path.add(currentWord.getWord());
67                 currentWord = parent.get(currentWord);
68             }
69             Collections.reverse(path);
70             return new Result(nodeCount, runtime, path);
71         } else {
72             return new Result(nodeCount, runtime, null);
73         }
74     }
75 }
76
77

```

3.3. UCS.java

File UCS.java berisikan *class* UCS yang memiliki *method* berupa fungsi bernama “resulting”. Fungsi menerima input dua node sebagai parameter awal, yaitu *startWord* (hasil konstruksi node dari kata awal) dan *endWord* (hasil konstruksi node dari kata target). Fungsi “resulting” mengimplementasikan algoritma Uniform Cost Search untuk mencari jalur terpendek dari *startWord* ke *endWord*. Fungsi menggunakan *priority queue* bawaan dari Java untuk menyimpan

setiap simpul hidup hasil ekspansi simpul ekspansi dengan prioritas pengurutan didasarkan pada simpul masukan yang memiliki fungsi evaluasi terkecil. *Priority queue* diurutkan berdasarkan nilai dari atribut cost dari setiap simpul masukan yang menandakan fungsi evaluasi berupa banyak kata atau langkah yang sudah ditempuh tiap simpulnya dari simpul awal. *Return value* dari fungsi berupa objek *Result* yang memuat informasi durasi pencarian, jumlah node dikunjungi, dan lintasan hasil.

```

1 package Algorithm;
2 import java.util.ArrayList;
3 import java.util.Collections;
4 import java.util.Comparator;
5 import java.util.HashMap;
6 import java.util.Map;
7 import java.util.PriorityQueue;
8
9 import Util.DictionaryChecker;
10 import Util.Node;
11 import Util.Result;
12
13 import java.util.List;
14
15 public class UCS {
16     public static Result resulting(Node startWord, Node endWord) {
17         Map<Node, Node> parent = new HashMap<>();
18         Map<String, Boolean> visited = new HashMap<>();
19
20         String candidateWord = null;
21         int cost = 0;
22         int nodeCount = 0;
23
24         PriorityQueue<Node> pq = new PriorityQueue<Node>(Comparator.comparingInt(Node::getCost));
25
26         pq.add(startWord);
27
28         long startTime = System.nanoTime(); // Record start time
29         Node currentWord = null;
30         while (!pq.isEmpty()) {
31             currentWord = pq.poll();
32             visited.put(currentWord.getWord(), true);
33             nodeCount++;
34
35             if (currentWord.getWord().equals(endWord.getWord())) {
36                 break;
37             }
38
39             if (currentWord.getCost() == 0) {
40                 cost++;
41             } else {
42                 cost = currentWord.getCost() + 1;
43             }
44
45             for (int j = 0; j < currentWord.getWord().length(); j++) {
46                 for (char c = 'a'; c <= 'z'; c++) {
47                     if (j == 0) {
48                         candidateWord = c + currentWord.getWord().substring(1, currentWord.getWord().length());
49                     } else if (j == currentWord.getWord().length() - 1) {
50                         candidateWord = currentWord.getWord().substring(0, j) + c;
51                     } else {
52                         candidateWord = currentWord.getWord().substring(0, j) + c + currentWord.getWord().substring(j + 1, currentWord.getWord().length());
53                     }
54
55                     if (candidateWord != currentWord.getWord() && !visited.containsKey(candidateWord)) {
56                         if (DictionaryChecker.isExist(candidateWord)) {
57                             Node child = new Node(candidateWord, 0, 0, cost);
58                             parent.put(child, currentWord);
59                             pq.add(child);
60                         }
61                     }
62                 }
63             }
64         }
65
66         long endTime = System.nanoTime(); // Record end time
67         double runtime = (double)(endTime - startTime)/1000000; // Calculate runtime in nanoseconds
68
69         if (pq.size() != 0) {
70             List<String> path = new ArrayList<>();
71             while (currentWord != null) {
72                 path.add(currentWord.getWord());
73                 currentWord = parent.get(currentWord);
74             }
75             Collections.reverse(path);
76             return new Result(nodeCount, runtime, path);
77         } else {
78             return new Result(nodeCount, runtime, null);
79         }
80     }
81 }
82

```

3.4. Result.java

File Result.java berisikan *class* Result untuk mengonstruksikan seluruh hasil dari pencarian menggunakan tiap algoritmanya. Atribut pada *class* ini :

- Visited : Menyimpan jumlah simpul yang dikunjungi oleh program
- Duration : Menyimpan durasi pencarian lintasan
- Path : Menyimpan lintasan hasil pencarian

Class Result juga dilengkapi getter setiap atribut sesuai kebutuhan.

```
1  package Util;
2  import java.util.List;
3
4  public class Result {
5      private int visited;
6      private double duration;
7      private List<String> path;
8
9      public Result(int visited, double duration, List<String> path) {
10         this.visited = visited;
11         this.duration = duration;
12         this.path = path;
13     }
14
15     public int getVisited() {
16         return this.visited;
17     }
18
19     public List<String> getPath() {
20         return this.path;
21     }
22
23     public double getDuration() {
24         return this.duration;
25     }
26 }
27
```

3.5. Input.java

File Input.java berisikan *class* Input untuk melakukan penerimaan masukan dari pengguna. Atribut pada *class* ini :

- inputNode : Konstruksi simpul dari masukan kata awal
 - targetNode : Konstruksi simpul dari masukan kata target
 - algorithmChoice : Pilihan algoritma dalam integer
- Class Input* juga dilengkapi getter setiap atribut sesuai kebutuhan.

```

1 package Util;
2 import java.util.Scanner;
3
4 public class Input {
5     private Node inputNode;
6     private Node targetNode;
7     private int algorithmChoice;
8
9     public Input() {
10         Scanner sc = new Scanner(System.in);
11         String str1, str2;
12         int algoChoice;
13         System.out.print("Enter start word: ");
14         str1 = sc.nextLine(); // Read start word
15         do {
16             System.out.print("Enter end word: ");
17             str2 = sc.nextLine(); // Read end word
18
19             // Check if str2 has the same length as str1
20             if (str2.length() != str1.length()) {
21                 System.out.println("End word must have the same length as the start word.");
22             }
23         } while (str2.length() != str1.length()); // Repeat if lengths are different
24
25         do {
26             System.out.print("1. A* Algorithm\n2. Greedy Best First Search Algorithm\n3. Uniform Cost Search Algorithm\nEnter algorithm choice (1/2/3): ");
27             algoChoice = sc.nextInt(); // Read integer input
28         } while (algoChoice < 1 || algoChoice > 3);
29
30         sc.close();
31
32         this.inputNode = new Node(str1, 0, 0, 0);
33         this.targetNode = new Node(str2, 0, 0, 0);
34         this.algorithmChoice = algoChoice;
35     }
36
37     public Node getInputNode() {
38         return this.inputNode;
39     }
40
41     public Node getTargetNode() {
42         return this.targetNode;
43     }
44
45     public int getAlgorithmChoice() {
46         return this.algorithmChoice;
47     }
48 }
49

```

3.6. Node.java

File Node.java berisikan *class* Node untuk melakukan penerimaan masukan dari pengguna. Atribut pada *class* ini :

- word: Kata yang disimpan
- cost: Banyak kata atau langkah yang sudah ditempuh tiap simpulnya dari simpul awal (pada penggunaan node di algoritma Greedy Best First Search, cost selalu bernilai 0)
- heuristic: Banyak perbedaan huruf antara tiap kata dengan kata tujuan (pada penggunaan node di algoritma Uniform Cost Search, cost selalu bernilai 0)
- weight: Penjumlahan heuristic dan cost

Class Node juga dilengkapi getter dan setter setiap atribut sesuai kebutuhan.

```
1 package Util;
2 public class Node {
3     private int weight;
4     private int cost;
5     private int heuristic;
6     private String word;
7
8     public Node(String word, int weight, int heuristic, int cost) {
9         this.word = word;
10        this.weight = weight;
11        this.heuristic = heuristic;
12        this.cost = cost;
13    }
14
15    // public void setChildNode (Node node) {
16    //     this.children.add(node);
17    // }
18
19    // public List<Node> getChildren() {
20    //     return this.children;
21    // }
22
23    public String getWord() {
24        return this.word;
25    }
26
27    public int getWeight() {
28        return this.weight;
29    }
30
31    public int getCost() {
32        return this.cost;
33    }
34
35    public int getHeuristic() {
36        return this.heuristic;
37    }
38
39    public void setWord(String word) {
40        this.word = word;
41    }
42
43    public void setWeight(int weight) {
44        this.weight = weight;
45    }
46
47    public void setCost(int cost) {
48        this.cost = cost;
49    }
50
51    public void setHeuristic(int heuristic) {
52        this.heuristic = heuristic;
53    }
54 }
```

3.7. DictionaryChecker.java

File DictionaryChecker.java berisikan *class* DictionaryChecker yang memiliki method berupa fungsi dictionary untuk mendapatkan semua kata dalam kamus, isExist yang mengembalikan boolean true jika suatu kata masukan ada dalam kamus dan false jika tidak ada, serta differentChar yang mengembalikan banyaknya perbedaan huruf dari dua kata.

```
1 package Util;
2 import java.io.BufferedReader;
3 import java.io.FileReader;
4 import java.io.IOException;
5 import java.util.HashSet;
6
7 public class DictionaryChecker {
8     private static final HashSet<String> dictionary = getDictionary("../lib\\Dictionary.txt");
9
10    private static HashSet<String> getDictionary(String fileName) {
11        HashSet<String> dictionary = new HashSet<>();
12
13        try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
14            String line;
15            while ((line = br.readLine()) != null) {
16                dictionary.add(line.trim().toLowerCase()); // Trim to remove leading/trailing whitespace and convert to lowercase
17            }
18        } catch (IOException e) {
19            e.printStackTrace();
20        }
21
22        return dictionary;
23    }
24
25    public static boolean isExist(String word) {
26        return dictionary.contains(word.toLowerCase());
27    }
28
29    public static int differentChar(String word1, String word2) {
30        int count = 0;
31        for (int i = 0; i < Math.min(word1.length(), word2.length()); i++) {
32            if (word1.charAt(i) != word2.charAt(i)) {
33                count++;
34            }
35        }
36        return count;
37    }
38 }
39
```

3.8. Main.java

File Main.java berisikan program utama yang memanfaatkan fungsi-fungsi pada modul-modul di atas.

```

1  import Algorithm.Astar;
2  import Algorithm.GBFS;
3  import Algorithm.UCS;
4  import Util.Result;
5  import Util.Input;
6
7  public class Main {
8      public static void main(String[] args) {
9          Input input = new Input();
10
11         Result hasil = new Result(0, 0, null);
12         if (input.getAlgorithmChoice() == 1) {
13             hasil = Astar.resulting(input.getInputNode(), input.getTargetNode());
14         } else if (input.getAlgorithmChoice() == 2) {
15             hasil = GBFS.resulting(input.getInputNode(), input.getTargetNode());
16         } else {
17             hasil = UCS.resulting(input.getInputNode(), input.getTargetNode());
18         }
19
20         if (hasil.getPath() != null) {
21             System.out.println("Hasil : ");
22             for(int i = 0; i < hasil.getPath().size(); i++) {
23                 System.out.println(hasil.getPath().get(i));
24             }
25         } else {
26             System.out.println("Path tidak ditemukan.");
27         }
28         System.out.println("Banyak node dikunjungi : " + hasil.getVisited());
29         System.out.println("Durasi : " + hasil.getDuration() + " ms.");
30     }
31 }

```

BAB IV

HASIL PENCARIAN

4.1. Test Case 1

Kata masukan : bag

Kata target : sit

- **Algoritma A***

```
Enter start word: bag
Enter end word: sit
1. A* Algorithm
2. Greedy Best First Search Algorithm
3. Uniform Cost Search Algorithm
Enter algorithm choice (1/2/3): 1
Hasil :
bag
sag
sat
sit
Banyak node dikunjungi : 6
Durasi : 118.3672 ms.
```

- **Algoritma Uniform Cost Search**

```
Enter start word: bag
Enter end word: sit
1. A* Algorithm
2. Greedy Best First Search Algorithm
3. Uniform Cost Search Algorithm
Enter algorithm choice (1/2/3): 3
Hasil :
bag
bat
sat
sit
Banyak node dikunjungi : 705
Durasi : 190.4865 ms.
```

- **Algoritma Greedy Best First Search**

```

Enter start word: bag
Enter end word: sit
1. A* Algorithm
2. Greedy Best First Search Algorithm
3. Uniform Cost Search Algorithm
Enter algorithm choice (1/2/3): 2
Hasil :
bag
sag
sat
sit
Banyak node dikunjungi : 4
Durasi : 114.8202 ms.

```

4.2. Test Case 2

Kata masukan : pear

Kata target : miss

- **Algoritma A***

```

Enter start word: pear
Enter end word: miss
1. A* Algorithm
2. Greedy Best First Search Algorithm
3. Uniform Cost Search Algorithm
Enter algorithm choice (1/2/3): 1
Hasil :
pear
peas
pias
piss
miss
Banyak node dikunjungi : 5
Durasi : 109.5177 ms.

```

- **Algoritma Uniform Cost Search**

```

Enter start word: pear
Enter end word: miss
1. A* Algorithm
2. Greedy Best First Search Algorithm
3. Uniform Cost Search Algorithm
Enter algorithm choice (1/2/3): 3
Hasil :
pear
peas
pias
piss
miss
Banyak node dikunjungi : 4359
Durasi : 354.7083 ms.

```

- **Algoritma Greedy Best First Search**

```
Enter start word: pear
Enter end word: miss
1. A* Algorithm
2. Greedy Best First Search Algorithm
3. Uniform Cost Search Algorithm
Enter algorithm choice (1/2/3): 2
Hasil :
pear
peas
pias
piss
miss
Banyak node dikunjungi : 5
Durasi : 111.4737 ms.
```

4.3. Test Case 3

Kata masukan : ready

Kata target : great

- **Algoritma A***

```
Enter start word: ready
Enter end word: great
1. A* Algorithm
2. Greedy Best First Search Algorithm
3. Uniform Cost Search Algorithm
Enter algorithm choice (1/2/3): 1
Hasil :
ready
beady
beads
brads
braes
brees
grees
greet
great
Banyak node dikunjungi : 310
Durasi : 148.8575 ms.
```

- **Algoritma Uniform Cost Search**


```

Enter start word: ready
Enter end word: great
1. A* Algorithm
2. Greedy Best First Search Algorithm
3. Uniform Cost Search Algorithm
Enter algorithm choice (1/2/3): 3
Hasil :
ready
reads
beads
brads
braes
brees
grees
greet
great
Banyak node dikunjungi : 81393
Durasi : 1751.7033 ms.

```

○ **Algoritma Greedy Best First Search**

```

Enter start word: ready
Enter end word: great
1. A* Algorithm
2. Greedy Best First Search Algorithm
3. Uniform Cost Search Algorithm
Enter algorithm choice (1/2/3): 2
Hasil :
ready
reedy
seedy
seeds
seers
seeps
seepy
seely
seels
seems
seeks
weeks
weeps
weepy
weeny
weens
wrens
brems
brees
grees
greet
great
Banyak node dikunjungi : 43
Durasi : 126.0362 ms.

```

4.4. Test Case 4

Kata masukan : comedo

Kata target : charge

- **Algoritma A***

```
Enter start word: comedo
Enter end word: charge
1. A* Algorithm
2. Greedy Best First Search Algorithm
3. Uniform Cost Search Algorithm
Enter algorithm choice (1/2/3): 1
Hasil :
comedo
comedy
comely
homely
homily
hominy
homing
coming
coning
conins
conies
conges
conger
conner
conned
coined
chined
chines
chints
chants
changs
change
charge
Banyak node dikunjungi : 17288
Durasi : 822.1162 ms.
```

- **Algoritma Uniform Cost Search**

```
Enter start word: comedo
Enter end word: charge
1. A* Algorithm
2. Greedy Best First Search Algorithm
3. Uniform Cost Search Algorithm
Enter algorithm choice (1/2/3): 3
Hasil :
comedo
comedy
comely
homely
homily
hominy
homing
honing
coning
conins
conies
conges
conger
conner
coiner
coined
chined
chines
chints
chants
changes
change
charge
Banyak node dikunjungi : 94850
Durasi : 2262.7871 ms.
```

- **Algoritma Greedy Best First Search**

```

Enter start word: comedo
Enter end word: charge
1. A* Algorithm
2. Greedy Best First Search Algorithm
3. Uniform Cost Search Algorithm
Enter algorithm choice (1/2/3): 2
Hasil :
comedo
comedy
comely
homely
homily
hominy
homing
coming
coding
coying
coving
covins
covens
codens
coders
cowers
comers
comets
compts
coapts
coasts
clasts
clasps
clamps
champs
champy
chammy
chasmy
chasms
charms
charts
chants
changs
change
charge
Banyak node dikunjungi : 314
Durasi : 159.9126 ms.

```

4.5. Test Case 5

Kata masukan : Genuine

Kata target : Foreign

- Algoritma A*

```
Enter start word: genuine
Enter end word: foreign
1. A* Algorithm
2. Greedy Best First Search Algorithm
3. Uniform Cost Search Algorithm
Enter algorithm choice (1/2/3): 1
Path tidak ditemukan.
Banyak node dikunjungi : 1
Durasi : 121.7353 ms.
```

- **Algoritma Uniform Cost Search**

```
Enter start word: genuine
Enter end word: foreign
1. A* Algorithm
2. Greedy Best First Search Algorithm
3. Uniform Cost Search Algorithm
Enter algorithm choice (1/2/3): 3
Path tidak ditemukan.
Banyak node dikunjungi : 1
Durasi : 107.4828 ms.
```

- **Algoritma Greedy Best First Search**

```
Enter start word: genuine
Enter end word: foreign
1. A* Algorithm
2. Greedy Best First Search Algorithm
3. Uniform Cost Search Algorithm
Enter algorithm choice (1/2/3): 2
Path tidak ditemukan.
Banyak node dikunjungi : 1
Durasi : 114.3316 ms.
```

4.6. Test Case 6

Kata masukan : accident

Kata target : accepted

- **Algoritma A***

```
Enter start word: accident
Enter end word: accepted
1. A* Algorithm
2. Greedy Best First Search Algorithm
3. Uniform Cost Search Algorithm
Enter algorithm choice (1/2/3): 1
Path tidak ditemukan.
Banyak node dikunjungi : 2
Durasi : 106.004 ms.
```

- **Algoritma Uniform Cost Search**

```
Enter start word: accident
Enter end word: accepted
1. A* Algorithm
2. Greedy Best First Search Algorithm
3. Uniform Cost Search Algorithm
Enter algorithm choice (1/2/3): 3
Path tidak ditemukan.
Banyak node dikunjungi : 2
Durasi : 125.9516 ms.
```

- **Algoritma Greedy Best First Search**

```
Enter start word: accident
Enter end word: accepted
1. A* Algorithm
2. Greedy Best First Search Algorithm
3. Uniform Cost Search Algorithm
Enter algorithm choice (1/2/3): 2
Path tidak ditemukan.
Banyak node dikunjungi : 2
Durasi : 105.4106 ms.
```

BAB V

ANALISIS PERBANDINGAN SOLUSI

Dari hasil percobaan pencarian rute dengan berbagai variasi panjang kata, didapatkan hasil lintasan, panjang lintasan, waktu pencarian, dan jumlah node terkunjungi yang bervariasi pada kasus panjang kata yang berbeda-beda. Pada kasus didapatkannya solusi lintasan dari kata awal menuju kata target, algoritma Uniform Cost Search selalu menghasilkan lintasan yang memang optimal secara panjang lintasan namun selalu memiliki durasi pencarian terbesar dan jumlah node terkunjungi terbanyak. Hal ini dikarenakan algoritma Uniform Cost Search akan melakukan pengunjungan terhadap seluruh node yang berada pada simpul hidup akibat biaya yang sama pada setiap levelnya. Algoritma Uniform Cost Search dijamin menghasilkan solusi optimal namun tidak efisien dalam waktu eksekusi dan penggunaan memori.

Algoritma UCS ini memiliki kompleksitas waktu $O(b^d)$ dan kompleksitas ruang $O(b^d)$ dengan b merupakan jumlah tetangga yang dimiliki oleh sebuah simpul dan d berupa kedalaman dari pohon ruang status.

```

Enter start word: bag
Enter end word: sit
1. A* Algorithm
2. Greedy Best First Search Algorithm
3. Uniform Cost Search Algorithm
Enter algorithm choice (1/2/3): 3
Hasil :
bag
boat
sat
sit
Banyak node dikunjungi : 705
Durasi : 190.4865 ms.

Enter start word: ready
Enter end word: great
1. A* Algorithm
2. Greedy Best First Search Algorithm
3. Uniform Cost Search Algorithm
Enter algorithm choice (1/2/3): 3
Hasil :
ready
reads
reads
braes
braes
braes
grees
grees
greet
great
Banyak node dikunjungi : 81393
Durasi : 1751.7033 ms.

```

Algoritma Greedy Best First Search tidak selalu memberikan solusi optimal dari hasil pencariannya. Pada kasus pencarian lintasan kata dengan panjang lima huruf (ready menuju start), didapatkan lintasan solusi dengan panjang 22 kata yang melebihi solusi optimal dari hasil pencarian dengan algoritma A* dan UCS. Pada kasus pencarian lintasan kata dengan panjang enam huruf (comedo menuju charge) didapatkan lintasan solusi dengan panjang 34 kata yang melebihi solusi optimal dari hasil pencarian dengan algoritma A* dan UCS. Hal ini karena algoritma Greedy Best First Search akan selalu mengambil langkah yang nilainya bersifat optimum lokal, yaitu mengambil langkah

yang nilai heuristik atau biaya perkiraan menuju simpul target paling kecil tanpa memikirkan sudah sejauh mana pergerakan simpul pada program dilakukan. Sehingga penelusuran akan terus maju dilakukan hingga mendapat solusi akhir tanpa mempertimbangkan jarak penelusuran dari simpul awal. Pengambilan nilai optimum lokal ini tidak menjamin terhasilkannya nilai optimum global, namun dari hasil pengamatan didapatkan algoritma ini beberapa kali memiliki waktu eksekusi yang lebih cepat dan penggunaan memori yang lebih kecil karena sudah ‘merasa puas’ dengan solusi yang didapat akibat selalu melihat nilai optimum lokal tanpa mempertimbangkan kembali adanya kemungkinan susunan solusi lain yang memiliki jarak lebih kecil.

[illegible]

Algoritma Greedy Best First Search tidak selalu memberikan solusi optimal dari hasil pencariannya. Pada kasus pencarian lintasan kata dengan panjang lima huruf (ready menuju start), didapatkan lintasan solusi dengan panjang 22 kata yang melebihi solusi optimal dari hasil pencarian dengan algoritma A* dan UCS. Pada kasus pencarian lintasan kata dengan panjang enam huruf (comedo menuju charge) didapatkan lintasan solusi dengan panjang 34 kata yang melebihi solusi optimal dari hasil pencarian dengan algoritma A* dan UCS. Hal ini karena algoritma Greedy Best First Search akan selalu

mengambil langkah yang nilainya bersifat optimum lokal, yaitu mengambil langkah yang nilai heuristik atau biaya perkiraan menuju simpul target paling kecil tanpa memikirkan sudah sejauh mana pergerakan simpul pada program dilakukan. Sehingga penelusuran akan terus maju dilakukan hingga mendapat solusi akhir tanpa mempertimbangkan jarak penelusuran dari simpul awal. Pengambilan nilai optimum lokal ini tidak menjamin terhasilkannya nilai optimum global, namun dari hasil pengamatan didapatkan algoritma ini beberapa kali memiliki waktu eksekusi yang lebih cepat dan penggunaan memori yang lebih kecil karena sudah ‘merasa puas’ dengan solusi yang didapat akibat selalu melihat nilai optimum lokal tanpa mempertimbangkan kembali adanya kemungkinan susunan solusi lain yang memiliki jarak lebih kecil.

Algoritma A* pada pencarian solusi selalu menghasilkan lintasan yang optimal karena tentunya memiliki pertimbangan yang lebih menyeluruh untuk menentukan urutan pengunjungan simpul-simpul hidupnya hingga didapatkan solusi yang optimal. Algoritma A* mempertimbangkan jarak tiap simpul ke simpul target sehingga selalu mencari kemungkinan jarak tempuh terpendek menuju solusi optimal sekaligus mempertimbangkan jarak simpul awal ke setiap simpul sehingga dimungkinkan terjadi pengunjungan simpul pada cost yang lebih rendah namun memiliki jarak yang lebih optimal menuju simpul target.

Algoritma A* ini memiliki kompleksitas waktu $O(b^d)$ dan kompleksitas ruang $O(b^d)$ dengan b merupakan jumlah tetangga yang dimiliki oleh sebuah simpul dan d berupa kedalaman dari pohon ruang status. Namun, penghasilan solusi bisa lebih optimal dari algoritma UCS karena selain mempertimbangkan jarak simpul awal ke tiap simpulnya, algoritma A* juga mempertimbangkan nilai heuristik berupa jarak simpul ke simpul target untuk pengurutan pengunjungan simpul hidup.

Maka, dapat disimpulkan algoritma A* dan UCS akan selalu menghasilkan solusi optimal global. Namun, algoritma A* lebih efisien secara ruang dan waktu dalam proses pencariannya. Sedangkan algoritma Greedy Best First Search tidak selalu menjamin dihasilkannya solusi optimal global karena pengambilan setiap langkahnya didasarkan pada nilai optimum lokal.

LAMPIRAN

Link repository :

https://github.com/andhitanh/Tucil3_13522060.git

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		✓

DAFTAR PUSTAKA

Munir, Rinaldi. 2024. “Penentuan Rute(Route/Path Planning)(Bagian 1)”(online). (<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf>, diakses 4 Mei 2024).

Munir, Rinaldi. 2024. “Penentuan Rute(Route/Path Planning)(Bagian 2)”(online). (<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf>, diakses 4 Mei 2024).