

#Tucil3LetsGo!

LAPORAN TUGAS KECIL 3
IF2211 STRATEGI ALGORITMA
Penyelesaian Permainan *Word Ladder* Menggunakan Algoritma
UCS, Greedy Best First Search, dan A*



Disusun Oleh:

13522069 Nabila Shikoofa Muida

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

2024

DAFTAR ISI

DAFTAR ISI.....	1
BAB I	
DESKRIPSI MASALAH.....	3
1.1. Deskripsi Persoalan.....	3
1.2. Spesifikasi Tugas.....	4
1.2.1. Input.....	4
1.2.2. Output.....	4
1.2.3. Bonus.....	4
BAB II	
IMPLEMENTASI PROGRAM.....	5
2.1. Word Ladder Solver dengan Memanfaatkan Algoritma Uniform Cost Search (UCS).....	5
2.2. Word Ladder Solver dengan Memanfaatkan Algoritma Greedy Best First Search.....	6
2.3. Word Ladder Solver dengan Memanfaatkan Algoritma A*.....	8
2.4. Analisis Implementasi Word Ladder Solver.....	9
BAB III	
SOURCE CODE PROGRAM.....	12
3.1. File dictionary.txt.....	12
3.2. AStar.java.....	13
3.3. GBFS.java.....	14
3.4. Main.java.....	15
3.5. Node.java.....	16
3.6. PathFinder.java.....	18
3.7. SearchResult.java.....	19
3.8. UCS.java.....	20
3.9. WordLadderSearch.java.....	21
3.10. WordLadderSolver.java.....	22
3.11. WordValidator.java.....	26
BAB IV	
PENGUJIAN DAN ANALISIS.....	28
4.1. Test Case 1 (home-work).....	28
4.2. Test Case 2 (knit - yarn).....	29
4.3. Test Case 3 (flown - spits).....	30
4.4. Test Case 4 (paint - brush).....	31
4.5. Test Case 5 (mitts - scarf).....	32
4.6. Test Case 6 (ladder - laders).....	33

4.7. Analisis Perbandingan Solusi.....	34
4.7.1. Optimalitas.....	34
4.7.2. Waktu Eksekusi.....	35
4.7.3. Memori yang Dibutuhkan.....	36
4.8. Analisis Keunggulan dan Kelemahan.....	37
4.8.1. Uniform Cost Search (UCS).....	37
4.8.2. Greedy Best First Search (GBFS).....	38
4.8.3. A*.....	38
BAB V	
PENJELASAN IMPLEMENTASI BONUS.....	39
DAFTAR PUSTAKA.....	41
LAMPIRAN.....	42
Lampiran 1 Link Repository.....	42
Lampiran 2 Tabel Checklist Poin.....	42

BAB I

DESKRIPSI MASALAH

1.1. Deskripsi Persoalan

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

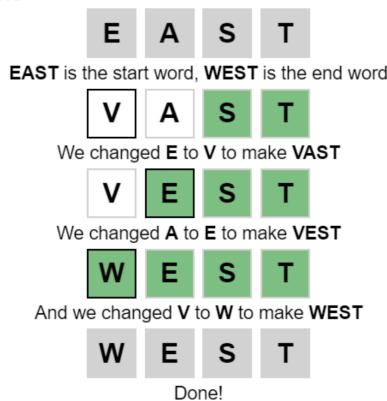
This game is called a "word ladder" and was invented by Lewis Carroll in 1877.

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. Ilustrasi dan Peraturan Permainan Word Ladder

1.2. Spesifikasi Tugas

Program *word ladder solver* yang dibangun memanfaatkan algoritma **UCS**, **Greedy Best First Search**, dan **A*** dalam bahasa Java. Berikut adalah detail dari spesifikasi program yang harus dibuat.

1.2.1. *Input*

Pengguna perlu memasukkan:

- **Start word** dan **end word** yang dapat menangani berbagai panjang kata
- **Pilihan algoritma** yang digunakan (**UCS**, **Greedy Best First Search**, dan **A***)

1.2.2. *Output*

Program menampilkan:

- Path yang dihasilkan dari start word ke end word (cukup 1 path saja)
- Banyaknya node yang dikunjungi
- Waktu eksekusi program

1.2.3. Bonus

Berikut adalah spesifikasi bonus dari program yang dibangun.

- Program dapat berjalan dengan GUI (Graphical User Interface)

BAB II

IMPLEMENTASI PROGRAM

2.1. Word Ladder Solver dengan Memanfaatkan Algoritma *Uniform Cost Search (UCS)*

Salah satu cara untuk menyelesaikan permainan Word Ladder adalah dengan menggunakan algoritma *Uniform Cost Search (UCS)*. Pada algoritma ini, fungsi menerima dua masukan yaitu *start word* dan *end word* dengan pilihan pencarian *Uniform Cost Search*. Algoritma ini menggunakan fungsi evaluasi pencarian. Pada algoritma ini, digunakan fungsi evaluasi $f(n) = g(n)$, dimana nilai $g(n)$ suatu simpul adalah jumlah seluruh cost pada simpul yang dilalui untuk mencapai simpul n dari simpul awal.

Langkah-Langkah Penyelesaian Permainan Word Ladder Menggunakan Algoritma *Uniform Cost Search*:

1. Inisialisasi struktur data *map*, *parentMap* untuk menyimpan relasi antara kata-kata dalam lintasan dan *costMap* untuk menyimpan *cost* lintasan terpendek yang diketahui saat ini dari titik awal ke setiap kata. Kemudian, gunakan priority queue untuk menyimpan simpul yang akan dieksplorasi selanjutnya berdasarkan *cost* lintasan terpendek yang diketahui saat ini.
2. Tambahkan titik awal ke dalam *parentMap* dengan nilai null (karena tidak ada titik sebelumnya), dan masukkan ke dalam *costMap* dengan *cost* lintasan awal 0. Kemudian, tambahkan titik awal ke dalam priority queue dengan *cost* 0.
3. Selama priority queue tidak kosong, lakukan langkah-langkah berikut:
 - a. Ambil simpul dengan *cost* lintasan terpendek dari priority queue.
 - b. Periksa apakah simpul saat ini adalah titik akhir yang diinginkan. Jika ya, kembalikan lintasan yang ditemukan.
 - c. Untuk setiap tetangga dari simpul saat ini, hitung *cost* pergerakan ke tetangga tersebut dan total *cost* untuk mencapainya dari titik awal melalui simpul saat ini.
 - d. Jika total *cost* baru yang ditemukan lebih rendah dari *cost* yang saat ini diketahui untuk tetangga tersebut, perbarui *costMap* dan *parentMap*, dan tambahkan tetangga tersebut ke dalam priority queue dengan *cost* baru.

4. Jika *priority queue* kosong dan tidak ada solusi yang ditemukan, algoritma mengembalikan informasi bahwa tidak ada lintasan yang ditemukan.

Waktu yang dibutuhkan oleh algoritma UCS tergantung pada jumlah simpul dan jumlah tepi dalam graf pencarian. Pada setiap iterasi, algoritma harus memeriksa setiap tepi keluar dari simpul yang sedang dieksplorasi. Oleh karena itu, kompleksitas waktu UCS dapat dinyatakan sebagai $O(E \log V)$, di mana E adalah jumlah tepi dan V adalah jumlah simpul dalam graf pencarian. Selain itu, karena UCS menggunakan priority queue untuk menyimpan simpul-simpul yang akan dieksplorasi, waktu yang dibutuhkan untuk operasi penambahan dan penghapusan dari priority queue juga perlu dipertimbangkan. Kompleksitas waktu penambahan dan penghapusan dari priority queue biasanya adalah $O(\log V)$, di mana V adalah jumlah simpul dalam graf pencarian. Secara keseluruhan, kompleksitas waktu dari algoritma UCS adalah $O((E + V) \log V)$.

Ruang yang dibutuhkan oleh algoritma UCS juga tergantung pada jumlah simpul dan jumlah tepi dalam graf pencarian. Algoritma membutuhkan ruang untuk menyimpan informasi seperti *parentMap*, *costMap*, dan *priority queue*. Jumlah ruang yang dibutuhkan untuk menyimpan informasi ini tergantung pada ukuran graf pencarian. Kompleksitas ruang UCS dapat dinyatakan sebagai $O(V)$, di mana V adalah jumlah simpul dalam graf pencarian. Namun, perlu diingat bahwa dalam kasus terburuk, jika semua simpul dan tepi dalam graf harus disimpan dalam memori, kompleksitas ruang bisa menjadi $O(E + V)$, di mana E adalah jumlah tepi dan V adalah jumlah simpul dalam graf pencarian.

2.2. Word Ladder Solver dengan Memanfaatkan Algoritma *Greedy Best First Search*

Penyelesaian Word Ladder juga dapat dilakukan dengan menggunakan algoritma *Greedy Best First Search*. Algoritma Greedy Best First Search adalah salah satu algoritma yang digunakan untuk menyelesaikan permasalahan Word Ladder. Algoritma ini termasuk ke dalam kategori algoritma Informed Search, yang berarti algoritma ini menggunakan fungsi evaluasi untuk mengevaluasi setiap simpul. Pada algoritma Greedy Best First Search, fungsi evaluasi ini direpresentasikan sebagai $f(n)$, di mana $f(n)$ merupakan estimasi *cost* dari simpul n ke tujuan akhir.

Algoritma ini memilih simpul yang terlihat paling dekat dengan tujuan akhir pada setiap langkah pencarian. Dengan kata lain, Greedy Best First Search cenderung memilih jalur yang dianggap paling baik pada saat itu berdasarkan estimasi *cost* ke tujuan. Dalam implementasi kode yang Anda berikan, estimasi *cost* ini diwakili oleh nilai heuristik dari setiap kata.

Dalam penyelesaian word ladder ini, fungsi heuristik yang digunakan adalah *hamming distance* atau jumlah karakter yang berbeda antara kata saat ini dan kata tujuan. Misalnya, jika kata saat ini adalah "cat" dan kata tujuan adalah "dog", maka nilai heuristiknya adalah 3 karena terdapat 3 karakter yang berbeda antara kedua kata tersebut.

Langkah-Langkah Penyelesaian Permainan Word Ladder Menggunakan Algoritma *Greedy Best First Search*:

1. Inisialisasi struktur data *map*, *parentMap* untuk menyimpan relasi antara kata-kata dalam lintasan dan *heuristicMap* untuk menyimpan nilai heuristik dari setiap kata. Kemudian, gunakan priority queue untuk menyimpan simpul yang akan dieksplorasi selanjutnya berdasarkan nilai heuristik yang diperkirakan.
- 2.Tambahkan titik awal ke dalam *parentMap* dengan nilai null (karena tidak ada titik sebelumnya), dan masukkan ke dalam *heuristicMap* dengan nilai heuristik yang dihitung menggunakan fungsi heuristik. Kemudian, tambahkan titik awal ke dalam priority queue dengan nilai heuristik yang diperkirakan.
3. Selama *priority queue* tidak kosong, lakukan langkah-langkah berikut:
 - a. Ambil simpul dengan nilai heuristik terendah dari priority queue.
 - b. Periksa apakah simpul saat ini adalah titik akhir yang diinginkan. Jika ya, kembalikan lintasan yang ditemukan.
 - c. Untuk setiap tetangga dari simpul saat ini, periksa apakah tetangga tersebut belum dieksplorasi sebelumnya. Jika belum, tambahkan tetangga tersebut ke dalam *parentMap* dan *heuristicMap*, dan tambahkan ke dalam priority queue dengan nilai heuristik yang diperkirakan.
4. Jika *priority queue* kosong dan tidak ada solusi yang ditemukan, algoritma mengembalikan informasi bahwa tidak ada lintasan yang ditemukan..

Kompleksitas waktu dan ruang dari Greedy Best First Search bergantung pada faktor percabangan (branching factor) dan panjang maksimum jalur (maximum path length). Dalam kasus terburuk, kompleksitas waktu dan ruang dari algoritma ini adalah $O(b^m)$, di mana b adalah faktor percabangan dan m adalah panjang maksimum jalur.

2.3. Word Ladder Solver dengan Memanfaatkan Algoritma A*

Algoritma A* merupakan pendekatan yang menggabungkan keunggulan dari dua algoritma sebelumnya dalam menyelesaikan permasalahan Word Ladder. Dalam konteks Word Ladder Solver, algoritma A* menggunakan fungsi evaluasi yang mencakup dua faktor utama: biaya aktual ($g(n)$) dan estimasi biaya sisa ($h(n)$) dari simpul saat ini ke simpul tujuan. Fungsi evaluasi ini dinyatakan sebagai $f(n) = g(n) + h(n)$.

Estimasi biaya sisa ($h(n)$), juga dikenal sebagai fungsi heuristik, sangat mempengaruhi kinerja dan keefektifan algoritma A*. Fungsi heuristik yang baik dapat memberikan perkiraan yang akurat tentang biaya sisa ke tujuan, membantu algoritma dalam menemukan jalur yang optimal. Namun, fungsi heuristik yang buruk atau tidak akurat dapat mengarah pada solusi yang suboptimal atau bahkan kegagalan dalam menemukan solusi.

Langkah-Langkah Penyelesaian Permainan Word Ladder Menggunakan Algoritma A*:

1. Langkah pertama adalah menginisialisasi struktur data yang diperlukan, termasuk map untuk menyimpan relasi antara kata-kata dalam lintasan (*parentMap*), map untuk menyimpan biaya aktual (*gScoreMap*), map untuk menyimpan nilai fungsi evaluasi (*fScoreMap*), dan priority queue (*openSet*) untuk menyimpan simpul-simpul yang akan dieksplorasi berdasarkan nilai fungsi evaluasi.
2. Titik awal (*start*) dimasukkan ke dalam *parentMap* dengan nilai null (karena tidak ada titik sebelumnya). Biaya aktual (*gScoreMap*) untuk titik awal diatur menjadi 0. Nilai fungsi evaluasi (*fScoreMap*) untuk titik awal dihitung menggunakan fungsi heuristik (*calculateFScore*) dari titik awal ke titik akhir (*end*). Titik awal juga dimasukkan ke dalam *openSet* dengan nilai fungsi evaluasi yang telah dihitung.
3. Selama *openSet priority queue* tidak kosong, algoritma melakukan langkah-langkah berikut:

- a. Ambil simpul dengan nilai fungsi evaluasi terkecil dari openSet.
 - b. Periksa apakah simpul saat ini adalah titik akhir yang diinginkan. Jika ya, kembalikan lintasan yang ditemukan.
 - c. Untuk setiap tetangga dari simpul saat ini, hitung biaya aktual (*tentativeGScore*) dengan menambahkan 1 ke biaya aktual simpul saat ini.
 - d. Jika tetangga belum dieksplorasi sebelumnya atau memiliki biaya aktual yang lebih rendah daripada biaya aktual yang sebelumnya diketahui, update parentMap, gScoreMap, dan fScoreMap dengan nilai yang baru, dan tambahkan tetangga ke dalam *priority queue* dengan nilai fungsi evaluasi yang telah dihitung.
4. Jika *priority queue* kosong dan tidak ada solusi yang ditemukan, algoritma mengembalikan informasi bahwa tidak ada lintasan yang ditemukan.

2.4. Analisis Implementasi Word Ladder Solver

Implementasi Word Ladder Solver menggunakan tiga algoritma (Uniform Cost Search, Greedy Best First Search, dan A*) memiliki struktur yang mirip, memanfaatkan struktur data seperti map dan priority queue. Mereka berbeda dalam penentuan cost, di mana Uniform Cost Search menggunakan biaya aktual, Greedy Best First Search menggunakan estimasi biaya sisa, dan A* menggabungkan keduanya. Selain itu, implementasi menggunakan daftar kata yang telah dikunjungi untuk mencegah siklus dan memastikan pencarian berjalan dengan benar. Penggunaan priority queue memungkinkan algoritma untuk mengatur antrian berdasarkan prioritas tertentu, seperti cost terkecil atau estimasi biaya terkecil, yang membantu memilih simpul yang akan dieksplorasi selanjutnya dengan efisien.

Pada setiap algoritma, nilai cost ditentukan oleh fungsi $f(n)$. Fungsi evaluasi $f(n)$ merepresentasikan total cost yang dimiliki oleh simpul n . Fungsi $g(n)$ menunjukkan cost untuk mencapai simpul n dari simpul awal, sedangkan fungsi $h(n)$ mewakili cost yang diestimasi (dengan menggunakan heuristik) untuk mencapai simpul tujuan dari simpul n . Dalam kasus Word Ladder, fungsi $g(n)$ adalah jarak simpul n dari simpul awal, dengan setiap perpindahan antar kata memiliki jarak satu satuan. Fungsi $h(n)$ adalah jumlah karakter yang berbeda antara kata pada simpul n dan kata akhir. Algoritma UCS memiliki $f(n) = g(n)$, Greedy Best First Search memiliki $f(n) = h(n)$, dan A* memiliki $f(n) = g(n) + h(n)$.

Pada algoritma A*, heuristik yang digunakan dikatakan admissible jika bobot yang dihasilkannya selalu lebih kecil atau sama dengan bobot yang sebenarnya untuk mencapai tujuan. Dalam kasus Word Ladder, heuristik yang digunakan adalah jumlah karakter yang berbeda antara kata saat ini dan kata akhir, juga dikenal sebagai Hamming distance. Heuristik ini bersifat admissible karena langkah yang diperlukan untuk mencapai kata akhir dari kata saat ini setidaknya akan memerlukan jumlah langkah sebesar nilai heuristik ($h(n)$). Ketika menggunakan Hamming distance sebagai heuristik, selalu ada kata yang memiliki jumlah perbedaan karakter dengan kata akhir yang semakin kecil. Sebagai contoh, jika sebuah kata hanya memiliki satu karakter yang berbeda dengan kata akhir, itu berarti hanya perlu mengubah satu karakter untuk mencapai kata akhir. Oleh karena itu, nilai heuristik akan selalu lebih kecil atau sama dengan nilai sebenarnya ($h(n) \leq h^*(n)$), yang menyatakan jumlah langkah yang diperlukan dalam solusi optimal. Akibatnya, heuristik hamming distance pada algoritma A* dapat dianggap admissible, yang menjamin bahwa solusi yang ditemukan oleh algoritma A* pasti optimal.

Meskipun algoritma UCS dan BFS memiliki perbedaan dalam struktur data yang digunakan (*queue* untuk BFS dan *priority queue* untuk UCS), keduanya akan menghasilkan urutan node yang dibangkitkan yang sama. Dalam Word Ladder, setiap kata memiliki jarak satu langkah dengan tetangganya, yang mengakibatkan simpul yang paling dekat dengan simpul awal akan dijelajahi terlebih dahulu oleh algoritma UCS. Hal ini menyebabkan algoritma UCS menghasilkan urutan penelusuran kata yang serupa dengan BFS, dimulai dari simpul awal dan bergerak ke simpul-simpul tetangga secara berurutan. Dengan demikian, pada kasus Word Ladder, algoritma UCS mirip dengan BFS dalam hal urutan node yang dibangkitkan dan path yang dihasilkan.

Secara teoritis, algoritma A* akan lebih efisien dibandingkan dengan algoritma UCS pada kasus word ladder karena algoritma A* menggunakan heuristik untuk mengestimasi simpul-simpul yang lebih meyakinkan dan menelusuri simpul-simpul tersebut terlebih dahulu menggunakan fungsi evaluasi $f(n) = g(n) + h(n)$. Dengan menggunakan heuristik, A* dapat membuat perkiraan yang lebih baik tentang biaya total yang diperlukan untuk mencapai tujuan, sehingga memungkinkan untuk mengeksplorasi jalur-jalur yang lebih menjanjikan terlebih dahulu. Hal ini membantu algoritma A* untuk mencapai kata akhir dengan lebih

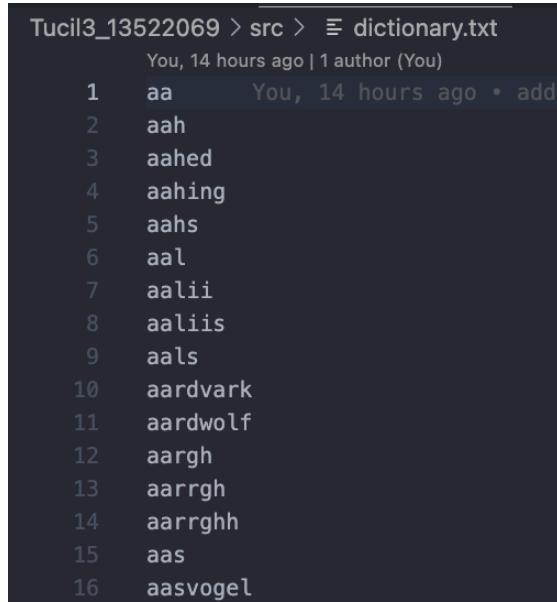
cepat karena mencegah ekspansi jalur-jalur yang diestimasi memiliki biaya yang mahal. Sebagai hasilnya, A* memiliki potensi untuk menemukan solusi dengan jumlah langkah yang lebih sedikit dibandingkan dengan UCS.

Secara teoritis, algoritma Greedy Best First Search tidak menjamin solusi optimal untuk persoalan Word Ladder karena algoritma ini hanya memilih simpul yang terbaik untuk ditelusuri berdasarkan heuristik tanpa mempertimbangkan bobot untuk mencapai simpul tersebut dari simpul awal. Karena sifatnya yang serakah, Greedy Best First Search dapat memulai penelusuran ke jalur tak terbatas dan tidak kembali untuk mencoba kemungkinan lain, sehingga algoritma ini bersifat tidak lengkap. Kegemarannya yang serakah dalam memilih jalur dapat mengarah pada jalan buntu, dan meskipun Greedy Best First Search akan melakukan pencarian ulang ke simpul terdalam yang belum dieksplorasi, namun hal ini tidak menjamin solusi optimal.

BAB III

SOURCE CODE PROGRAM

3.1. File dictionary.txt



```
Tucil3_13522069 > src > dictionary.txt
You, 14 hours ago | 1 author (You)
1 aa      You, 14 hours ago • add
2 aah
3 aahed
4 aahing
5 aahs
6 aal
7 aalii
8 aaliis
9 aals
10 aardvark
11 aardwolf
12 aargh
13 aarrgh
14 aarrghh
15 aas
16 aasvogel
```

Gambar 3.1.1. Isi File “dictionary.txt.”



```
Tucil3_13522069 > src > dictionary.txt
80354 zyme
80355 zymes
80356 zymogen
80357 zymogene
80358 zymogens
80359 zymogram
80360 zymology
80361 zymosan
80362 zymosans
80363 zymoses
80364 zymosis
80365 zymotic
80366 zymurgy
80367 zyzyva
80368 zyzyvas
```

Gambar 3.1.2. Isi File “dictionary.txt.”

File dictionary.txt berisi 80368 kata dalam bahasa Inggris yang dianggap valid untuk menyelesaikan kasus Word Ladder.

3.2. AStar.java



```
1 import java.util.*;
2
3 public class AStar implements WordLadderSearch {
4     @Override
5     public SearchResult findWordLadder(String start, String end, List<String> dictionary) {
6         Map<String, String> parentMap = new HashMap<>();
7         Map<String, Integer> gScoreMap = new HashMap<>();
8         Map<String, Integer> fScoreMap = new HashMap<>();
9         PriorityQueue<Node> openSet = new PriorityQueue<>(Comparator.comparingInt(Node::getFScore));
10
11        parentMap.put(start, null);
12        gScoreMap.put(start, 0);
13        fScoreMap.put(start, calculateFScore(start, end));
14
15        openSet.add(new Node(start, 0, fScoreMap.get(start)));
16
17        int nodeCount = 0;
18
19        while (!openSet.isEmpty()) {
20            nodeCount++;
21            Node current = openSet.poll();
22            String currentWord = current.getWord();
23
24            if (currentWord.equals(end)) {
25                return new SearchResult(PathFinder.constructPath(parentMap, end), nodeCount);
26            }
27
28            for (String neighbor : PathFinder.getNeighbors(currentWord, dictionary)) {
29                int tentativeGScore = gScoreMap.get(currentWord) + 1;
30
31                if (!gScoreMap.containsKey(neighbor) || tentativeGScore < gScoreMap.get(neighbor)) {
32                    parentMap.put(neighbor, currentWord);
33                    gScoreMap.put(neighbor, tentativeGScore);
34                    fScoreMap.put(neighbor, tentativeGScore + calculateFScore(neighbor, end));
35                    openSet.add(new Node(neighbor, tentativeGScore, fScoreMap.get(neighbor)));
36                }
37            }
38        }
39
40        return new SearchResult(Collections.emptyList(), nodeCount); // No ladder found
41    }
42
43    private static int calculateFScore(String word, String target) {
44        int heuristic = 0;
45        for (int i = 0; i < word.length(); i++) {
46            if (word.charAt(i) != target.charAt(i)) {
47                heuristic++;
48            }
49        }
50        return heuristic;
51    }
52 }
```

Gambar 3.2. Source Code “AStar.java”

Kelas AStar bertanggung jawab untuk menemukan jalur kata-kata dari kata awal ke kata akhir menggunakan algoritma A*.

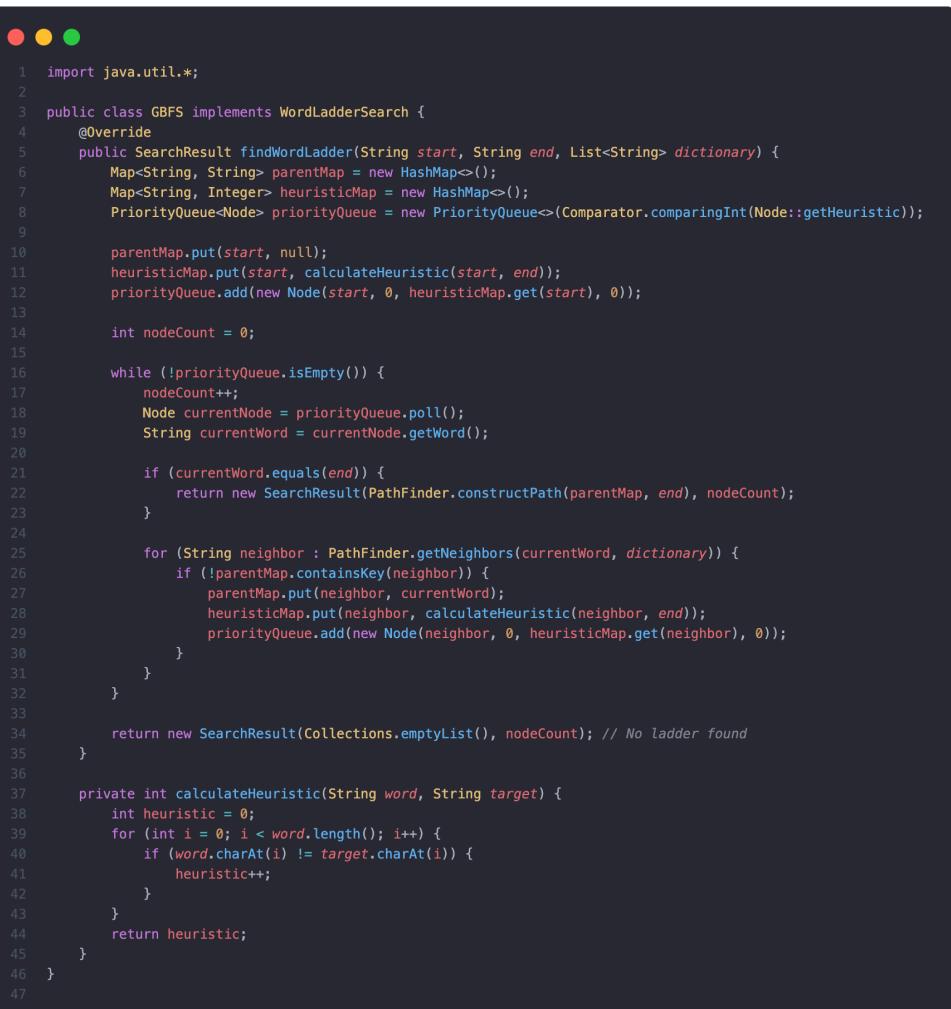
3.2.1. Method

- ❖ `findWordLadder`: Method utama yang mencari jalur dari kata awal ke kata akhir. Method ini menggunakan algoritma A* untuk mengeksplorasi simpul-simpul

dalam ruang kata. Melakukan iterasi hingga ditemukan jalur yang valid atau tidak ada jalur yang ditemukan.

- ❖ `calculateFScore`: Method untuk menghitung nilai fungsi evaluasi $f(n)$ untuk suatu kata. Method ini menghitung heuristik berupa jumlah karakter yang berbeda antara kata saat ini dan kata akhir.

3.3. GBFS.java



```
1 import java.util.*;
2
3 public class GBFS implements WordLadderSearch {
4     @Override
5     public SearchResult findWordLadder(String start, String end, List<String> dictionary) {
6         Map<String, String> parentMap = new HashMap<>();
7         Map<String, Integer> heuristicMap = new HashMap<>();
8         PriorityQueue<Node> priorityQueue = new PriorityQueue<>(Comparator.comparingInt(Node::getHeuristic));
9
10        parentMap.put(start, null);
11        heuristicMap.put(start, calculateHeuristic(start, end));
12        priorityQueue.add(new Node(start, 0, heuristicMap.get(start), 0));
13
14        int nodeCount = 0;
15
16        while (!priorityQueue.isEmpty()) {
17            nodeCount++;
18            Node currentNode = priorityQueue.poll();
19            String currentWord = currentNode.getWord();
20
21            if (currentWord.equals(end)) {
22                return new SearchResult(PathFinder.constructPath(parentMap, end), nodeCount);
23            }
24
25            for (String neighbor : PathFinder.getNeighbors(currentWord, dictionary)) {
26                if (!parentMap.containsKey(neighbor)) {
27                    parentMap.put(neighbor, currentWord);
28                    heuristicMap.put(neighbor, calculateHeuristic(neighbor, end));
29                    priorityQueue.add(new Node(neighbor, 0, heuristicMap.get(neighbor), 0));
30                }
31            }
32        }
33
34        return new SearchResult(Collections.emptyList(), nodeCount); // No ladder found
35    }
36
37    private int calculateHeuristic(String word, String target) {
38        int heuristic = 0;
39        for (int i = 0; i < word.length(); i++) {
40            if (word.charAt(i) != target.charAt(i)) {
41                heuristic++;
42            }
43        }
44        return heuristic;
45    }
46 }
47
```

Gambar 3.3. Source Code “GBFS.java”

Kelas GBFS adalah implementasi dari interface WordLadderSearch yang bertugas menemukan jalur kata dari kata awal ke kata akhir menggunakan algoritma Greedy Best First Search.

3.3.1. Method

- ❖ `findWordLadder`: Method utama yang mencari jalur dari kata awal ke kata akhir. Menggunakan algoritma Greedy Best First Search untuk mengeksplorasi simpul-simpul dalam ruang kata. Iterasi berlanjut hingga jalur ditemukan atau tidak ada jalur yang ditemukan.
- ❖ `calculateHeuristic`: Method untuk menghitung nilai heuristik untuk suatu kata. Menghitung jumlah karakter yang berbeda antara kata saat ini dan kata akhir sebagai heuristik.

3.4. Main.java



```
1 import javax.swing.*;
2
3 public class Main {
4     public static void main(String[] args) {
5         String[] options = {"GUI", "CLI"};
6         int choice = JOptionPane.showOptionDialog(null, "Choose user interface",
7             "Interface Selection", JOptionPane.DEFAULT_OPTION, JOptionPane.QUESTION_MESSAGE, null, options, options[0]);
8
9         if (choice == 0) {
10             SwingUtilities.invokeLater(new Runnable() { // GUI
11                 @Override
12                 public void run() {
13                     new GUI().setVisible(true);
14                 }
15             });
16         } else {
17             WordLadderSolver.runWordLadderSolver(); // CLI
18         }
19     }
20 }
21
```

Gambar 3.4. Source Code “Main.java”

Kelas Main berfungsi sebagai entry point dari program, pengguna dapat memilih apakah akan melakukan penyelesaian permainan word ladder dengan menggunakan Command Line Interface (CLI) atau Graphical User Interface (GUI).

3.5. Node.java

```
 1  public class Node {  
 2      private final String word;  
 3      private final int cost;  
 4      private final int heuristic;  
 5      private final int gScore;  
 6      private final int fScore;  
 7  
 8      public Node(String word, int cost){  
 9          this.word = word;  
10          this.cost = cost;  
11          this.heuristic = 0;  
12          this.fScore = 0;  
13          this.gScore = 0;  
14      }  
15  
16      public Node(String word, int gScore, int fScore) {  
17          this.word = word;  
18          this.gScore = gScore;  
19          this.fScore = fScore;  
20          this.cost = 0;  
21          this.heuristic = 0;  
22      }  
23  
24      public Node(String word, int cost, int heuristic, int fScore) {  
25          this.word = word;  
26          this.cost = cost;  
27          this.heuristic = heuristic;  
28          this.fScore = fScore;  
29          this.gScore = 0;  
30      }  
31  
32      public String getWord() {  
33          return word;  
34      }  
35  
36      public int getCost() {  
37          return cost;  
38      }  
39  
40      public int getHeuristic() {  
41          return heuristic;  
42      }  
43  
44      public int getFScore() {  
45          return fScore;  
46      }  
47  
48      public int getGScore() {  
49          return gScore;  
50      }  
51  }
```

Gambar 3.5. Source Code “Node.java”

Kelas Node digunakan untuk merepresentasikan simpul dalam algoritma pencarian.

3.5.1. Atribut

Setiap simpul memiliki atribut sebagai berikut:

- ❖ `word`: Merupakan kata yang direpresentasikan oleh simpul.
- ❖ `cost`: Merupakan biaya atau cost untuk mencapai simpul tersebut dari simpul sebelumnya.
- ❖ `heuristic`: Merupakan estimasi biaya dari simpul tersebut ke tujuan akhir.
- ❖ `gScore`: Merupakan biaya aktual dari simpul awal ke simpul saat ini.
- ❖ `fScore`: Merupakan nilai evaluasi dari simpul tersebut, dihitung sebagai jumlah dari `gScore` dan `heuristic`.

3.5.2. Method

- ❖ `Node(String word, int cost)`: Konstruktor untuk membuat objek `Node` dengan inisialisasi `word` dan `cost`.
- ❖ `Node(String word, int gScore, int fScore)`: Konstruktor untuk membuat objek `Node` dengan inisialisasi `word`, `gScore`, dan `fScore`.
- ❖ `Node(String word, int cost, int heuristic, int fScore)`: Konstruktor untuk membuat objek `Node` dengan inisialisasi semua atribut.
- ❖ Metode akses `getWord()`, `getCost()`, `getHeuristic()`, `getFScore()`, dan `getGScore()`: Digunakan untuk mendapatkan nilai dari masing-masing atribut.

3.6. PathFinder.java



```
1 import java.util.ArrayList;
2 import java.util.Collections;
3 import java.util.List;
4 import java.util.Map;
5
6 public class PathFinder {
7     public static List<String> constructPath(Map<String, String> parentMap, String end) {
8         List<String> path = new ArrayList<>();
9         String current = end;
10        while (current != null) {
11            path.add(current);
12            current = parentMap.get(current);
13        }
14        Collections.reverse(path);
15        return path;
16    }
17
18    public static List<String> getNeighbors(String word, List<String> dictionary) {
19        List<String> neighbors = new ArrayList<>();
20        for (String dictWord : dictionary) {
21            if (isNeighbor(word, dictWord)) {
22                neighbors.add(dictWord);
23            }
24        }
25        return neighbors;
26    }
27
28    private static boolean isNeighbor(String word1, String word2) {
29        if (word1.length() != word2.length()) {
30            return false;
31        }
32        int diffCount = 0;
33        for (int i = 0; i < word1.length(); i++) {
34            if (word1.charAt(i) != word2.charAt(i)) {
35                diffCount++;
36            }
37        }
38        return diffCount == 1;
39    }
40 }
41
```

Gambar 3.6. Source Code “PathFinder.java”

Kelas PathFinder digunakan untuk membantu dalam konstruksi jalur dan sebagai representasi hasil dari pencarian jalur dalam penyelesaian Word Ladder.

3.6.1. Method

- ❖ `constructPath(Map<String, String> parentMap, String end)`: Metode ini digunakan untuk membangun jalur dari simpul awal hingga simpul tujuan.
- ❖ `getNeighbors(String word, List<String> dictionary)`: Metode ini digunakan untuk mendapatkan tetangga-tetangga dari suatu kata dalam kamus (dictionary).
- ❖ `isNeighbor(String word1, String word2)`: Metode ini digunakan untuk memeriksa apakah dua kata merupakan tetangga satu sama lain.

3.7. SearchResult.java



```
1 import java.util.List;
2
3 public class SearchResult {
4     private final List<String> path;
5     private final int nodeCount;
6
7     public SearchResult(List<String> path, int nodeCount) {
8         this.path = path;
9         this.nodeCount = nodeCount;
10    }
11
12    public List<String> getPath() {
13        return path;
14    }
15
16    public int getNodeCount() {
17        return nodeCount;
18    }
19 }
20
```

Gambar 3.7. Source Code “SearchResult.java”

Kelas SearchResult digunakan untuk mengembalikan jalur yang ditemukan dari kata awal ke kata akhir beserta jumlah sumpil yang dievaluasi selama pencarian jalur.

3.7.1. Atribut

- ❖ path: Menyimpan jalur yang ditemukan dari kata awal ke kata akhir.
- ❖ nodeCount: Menyimpan jumlah simpul yang dievaluasi selama pencarian jalur.

3.7.2. Method

- ❖ getPath(): Mengembalikan jalur yang ditemukan dari kata awal ke kata akhir.
- ❖ getNodeCount(): Mengembalikan jumlah simpul yang dievaluasi selama pencarian jalur.

3.8. UCS.java

```
1 import java.util.*;
2
3 public class UCS implements WordLadderSearch {
4     @Override
5     public SearchResult findWordLadder(String start, String end, List<String> dictionary) {
6         Map<String, String> parentMap = new HashMap<>();
7         Map<String, Integer> costMap = new HashMap<>();
8         PriorityQueue<Node> priorityQueue = new PriorityQueue<>(Comparator.comparingInt(Node::getCost));
9
10        parentMap.put(start, null);
11        costMap.put(start, 0);
12        priorityQueue.add(new Node(start, 0));
13
14        int nodeCount = 0;
15
16        while (!priorityQueue.isEmpty()) {
17            nodeCount++;
18            Node currentNode = priorityQueue.poll();
19            String currentWord = currentNode.getWord();
20
21            if (currentWord.equals(end)) {
22                return new SearchResult(PathFinder.constructPath(parentMap, end), nodeCount);
23            }
24
25            for (String neighbor : PathFinder.getNeighbors(currentWord, dictionary)) {
26                int edgeCost = calculateEdgeCost(currentWord, neighbor);
27                int newCost = costMap.get(currentWord) + edgeCost;
28
29                if (costMap.computeIfAbsent(neighbor, k -> Integer.MAX_VALUE) > newCost) {
30                    costMap.put(neighbor, newCost);
31                    parentMap.put(neighbor, currentWord);
32                    priorityQueue.add(new Node(neighbor, newCost));
33                }
34            }
35        }
36
37        return new SearchResult(Collections.emptyList(), nodeCount); // No ladder found
38    }
39
40    private static int calculateEdgeCost(String word1, String word2) {
41        int cost = 0;
42        for (int i = 0; i < word1.length(); i++) {
43            if (word1.charAt(i) != word2.charAt(i)) {
44                cost++;
45            }
46        }
47        return cost;
48    }
49 }
```

Gambar 3.8. Source Code “UCS.java”

Kelas UCS adalah implementasi dari algoritma Uniform Cost Search (UCS) untuk menyelesaikan Word Ladder.

3.8.1. Method

- ❖ `findWordLadder(String start, String end, List<String> dictionary)`: Method ini digunakan untuk mencari jalur dari kata awal ke kata akhir. Implementasi ini menggunakan algoritma UCS.
- ❖ `calculateEdgeCost(String word1, String word2)`: Method ini digunakan untuk menghitung biaya perpindahan dari satu kata ke kata lain. Biaya perpindahan dihitung sebagai jumlah karakter yang berbeda antara dua kata.

3.9. WordLadderSearch.java



```
1 import java.util.List;
2
3 public interface WordLadderSearch {
4     SearchResult findWordLadder(String start, String end, List<String> dictionary);
5 }
6
```

Gambar 3.9. Source Code “WordLadderSearch.java”

Interface `WordLadderSearch` digunakan untuk menentukan kontrak bagi kelas-kelas yang mengimplementasikannya. Setiap kelas yang mengimplementasikan interface `WordLadderSearch` harus menyediakan implementasi untuk metode `findWordLadder`.

3.9.1. Method

- ❖ `findWordLadder(String start, String end, List<String> dictionary)`: Metode ini bertugas untuk mencari jalur dari kata awal ke kata akhir dalam permainan Word Ladder. Metode ini menerima tiga parameter: `start` (kata awal), `end` (kata akhir), dan `dictionary` (daftar kata-kata yang valid). Metode ini mengembalikan objek `SearchResult` yang berisi jalur yang ditemukan dan jumlah simpul yang dieksplorasi selama pencarian.

3.10. WordLadderSolver.java

```

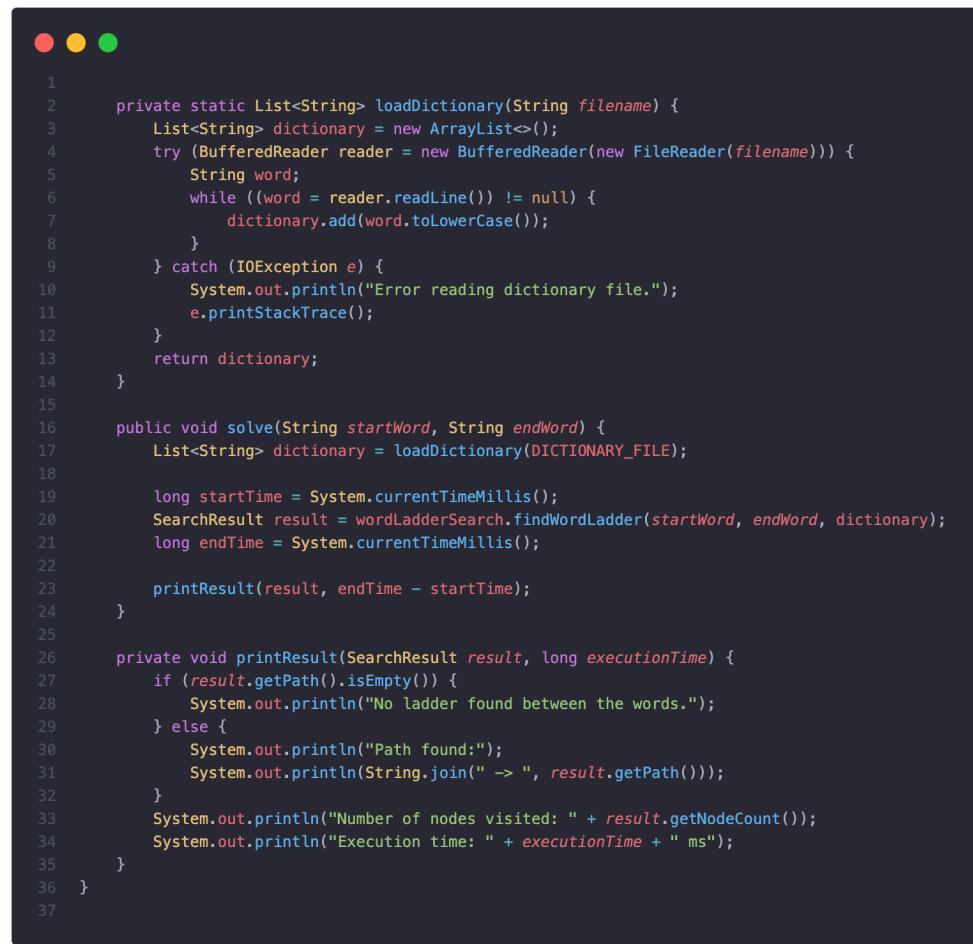
1 import java.io.BufferedReader;
2 import java.io.FileReader;
3 import java.io.IOException;
4 import java.util.ArrayList;
5 import java.util.List;
6 import java.util.Scanner;
7
8 public class WordLadderSolver {
9     private final WordLadderSearch wordLadderSearch;
10    private static final String DICTIONARY_FILE = "dictionary.txt";
11
12    public WordLadderSolver(WordLadderSearch wordLadderSearch) {
13        this.wordLadderSearch = wordLadderSearch;
14    }
15
16    public static void runWordLadderSolver() {
17        try (Scanner scanner = new Scanner(System.in)) {
18            boolean exit = false;
19
20            do {
21                List<String> dictionary = loadDictionary(DICTIONARY_FILE);
22                WordValidator wordValidator = new WordValidator(dictionary, scanner);
23
24                String startWord = wordValidator.getValidStartWord();
25                String endWord = wordValidator.getValidEndWord(startWord);
26
27                int choice = selectAlgorithm(scanner);
28
29                if (choice == 4) {
30                    compareAllAlgorithms(startWord, endWord);
31                } else {
32                    WordLadderSearch wordLadderSearch = createAlgorithm(choice);
33                    WordLadderSolver solver = new WordLadderSolver(wordLadderSearch);
34                    solver.solve(startWord, endWord);
35                }
36
37                System.out.print("Do you want to find another word ladder? (yes/no): ");
38                String input = scanner.nextLine().toLowerCase();
39                exit = !input.equals("yes");
40            } while (!exit);
41        } catch (Exception e) {
42            System.out.println("An unexpected error occurred: " + e.getMessage());
43            e.printStackTrace();
44        }
45    }
46
47    private static int selectAlgorithm(Scanner scanner) {
48        System.out.println("Choose an algorithm to use:");
49        System.out.println("1. Uniform Cost Search (UCS)");
50        System.out.println("2. Greedy Best-First Search (GBFS)");
51        System.out.println("3. A* Search");
52        System.out.println("4. Compare All Algorithms");
53
54        int choice = getValidChoice(scanner, 1, 4);
55        return choice;
56    }
57}

```

Gambar 3.10.1. Source Code “WordLadderSolver.java”

```
1  private static int getValidChoice(Scanner scanner, int min, int max) {
2      int choice;
3      while (true) {
4          System.out.print("Enter your choice: ");
5          try {
6              choice = Integer.parseInt(scanner.nextLine());
7              if (choice >= min && choice <= max) {
8                  break;
9              } else {
10                  System.out.println("Invalid choice.");
11                  System.out.println("Please enter a number between " + min + " and " + max + ".");
12              }
13          } catch (NumberFormatException e) {
14              System.out.println("Invalid choice. Please enter a number.");
15          }
16      }
17      return choice;
18  }
19
20
21
22  private static WordLadderSearch createAlgorithm(int choice) {
23      switch (choice) {
24          case 1:
25              return new UCS();
26          case 2:
27              return new GBFS();
28          case 3:
29              return new AStar();
30          case 4:
31              return null;
32          default:
33              throw new IllegalArgumentException("Invalid choice.");
34      }
35  }
36
37  private static void compareAllAlgorithms(String startWord, String endWord) {
38      System.out.println("Comparing all algorithms...");
39
40      WordLadderSearch[] algorithms = {new UCS(), new GBFS(), new AStar()};
41
42      for (WordLadderSearch algorithm : algorithms) {
43          WordLadderSolver solver = new WordLadderSolver(algorithm);
44          System.out.println("===== "
45          + algorithm.getClass().getSimpleName() + " =====");
46          solver.solve(startWord, endWord);
47      }
48  }
```

Gambar 3.10.2. Source Code “WordLadderSolver.java”



```

1  private static List<String> loadDictionary(String filename) {
2      List<String> dictionary = new ArrayList<>();
3      try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
4          String word;
5          while ((word = reader.readLine()) != null) {
6              dictionary.add(word.toLowerCase());
7          }
8      } catch (IOException e) {
9          System.out.println("Error reading dictionary file.");
10         e.printStackTrace();
11     }
12     return dictionary;
13 }
14 }
15
16 public void solve(String startWord, String endWord) {
17     List<String> dictionary = loadDictionary(DICTIONARY_FILE);
18
19     long startTime = System.currentTimeMillis();
20     SearchResult result = wordLadderSearch.findWordLadder(startWord, endWord, dictionary);
21     long endTime = System.currentTimeMillis();
22
23     printResult(result, endTime - startTime);
24 }
25
26 private void printResult(SearchResult result, long executionTime) {
27     if (result.getPath().isEmpty()) {
28         System.out.println("No ladder found between the words.");
29     } else {
30         System.out.println("Path found:");
31         System.out.println(String.join(" -> ", result.getPath()));
32     }
33     System.out.println("Number of nodes visited: " + result.getNodeCount());
34     System.out.println("Execution time: " + executionTime + " ms");
35 }
36 }
37

```

Gambar 3.10.3. *Source Code “WordLadderSolver.java”*

Kelas WordLadderSolver adalah kelas utama yang bertanggung jawab untuk menyelesaikan permainan Word Ladder. Berikut adalah penjelasan atribut dan metode yang dimilikinya:

3.10.1. Atribut

- ❖ wordLadderSearch: Sebuah objek dari kelas yang mengimplementasikan interface WordLadderSearch. Objek ini digunakan untuk melakukan pencarian jalur menggunakan algoritma tertentu.

3.10.2. Method

- ❖ WordLadderSolver(WordLadderSearch wordLadderSearch): Konstruktor untuk membuat objek WordLadderSolver dengan menginisialisasi atribut wordLadderSearch sesuai dengan algoritma pencarian yang dipilih.
- ❖ runWordLadderSolver(): Metode statis yang menjalankan permainan Word Ladder. Pengguna diminta untuk memilih algoritma pencarian dan kata awal serta kata akhir. Metode ini kemudian memanggil metode solve untuk menyelesaikan permainan.
- ❖ selectAlgorithm(Scanner scanner): Meminta pengguna untuk memilih algoritma pencarian.
- ❖ getValidChoice(Scanner scanner, int min, int max): Memvalidasi pilihan yang dimasukkan oleh pengguna.
- ❖ createAlgorithm(int choice): Membuat objek algoritma pencarian berdasarkan pilihan pengguna.
- ❖ compareAllAlgorithms(String startWord, String endWord): Membandingkan semua algoritma pencarian yang tersedia.
- ❖ loadDictionary(String filename): Membaca kata-kata dari file kamus dan menyimpannya dalam daftar.
- ❖ solve(String startWord, String endWord): Menyelesaikan permainan Word Ladder dengan menggunakan algoritma pencarian yang dipilih.
- ❖ printResult(SearchResult result, long executionTime): Mencetak hasil pencarian, termasuk jalur yang ditemukan, jumlah node yang dikunjungi, dan waktu eksekusi.

3.11. WordValidator.java



```

1 import java.util.List;
2 import java.util.Scanner;
3
4 public class WordValidator {
5     private final List<String> dictionary;
6     private final Scanner scanner;
7
8     public WordValidator(List<String> dictionary, Scanner scanner) {
9         this.dictionary = dictionary;
10        this.scanner = scanner;
11    }
12
13    public String getValidStartWord() {
14        String startWord;
15        do {
16            System.out.print("Enter the start word: ");
17            startWord = scanner.nextLine().toLowerCase();
18            if (!isValidWord(startWord)) {
19                System.out.println("Word '" + startWord + "' is not found in the dictionary. Please enter a valid word.");
20            }
21        } while (!isValidWord(startWord));
22        return startWord;
23    }
24
25    public String getValidEndWord(String startWord) {
26        String endWord;
27        do {
28            System.out.print("Enter the end word: ");
29            endWord = scanner.nextLine().toLowerCase();
30            if (!isValidWord(endWord)) {
31                System.out.println("Word '" + endWord + "' is not found in the dictionary. Please enter a valid word.");
32            } else if (endWord.length() != startWord.length()) {
33                System.out.println("The end word must have the same length as the start word. Please enter a valid word.");
34            } else if (endWord.equals(startWord)) {
35                System.out.println("The end word cannot be the same as the start word. Please enter a different word.");
36            }
37        } while (!isValidWord(endWord) || endWord.length() != startWord.length());
38        return endWord;
39    }
40
41    private boolean isValidWord(String word) {
42        return dictionary.contains(word);
43    }
44}

```

Gambar 3.11. Source Code “WordValidator.java”

Kelas WordValidator bertanggung jawab untuk memvalidasi kata-kata yang dimasukkan oleh pengguna sebagai kata awal dan kata akhir permainan Word Ladder. Berikut adalah penjelasan atribut, konstruktor, dan metode yang dimiliki oleh kelas ini:

3.11.1. Atribut

- ❖ **dictionary:** Sebuah daftar kata-kata dari kamus yang digunakan untuk memvalidasi kata-kata yang dimasukkan oleh pengguna.

- ❖ `scanner`: Objek `Scanner` yang digunakan untuk menerima masukan dari pengguna.

3.11.2. Method

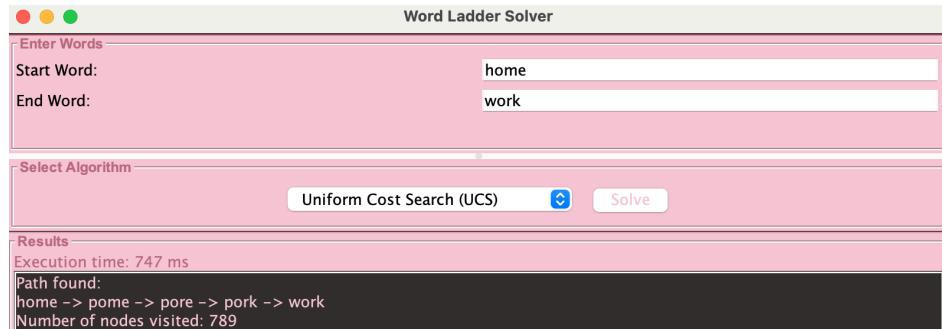
- ❖ `WordValidator(List<String> dictionary, Scanner scanner)`: Konstruktor untuk membuat objek `WordValidator` dengan menginisialisasi atribut `dictionary` dan `scanner` sesuai dengan nilai yang diberikan.
- ❖ `getValidStartWord()`: Meminta pengguna untuk memasukkan kata awal dan memvalidasi masukan tersebut. Metode ini akan terus meminta masukan baru hingga kata yang dimasukkan valid.
- ❖ `getValidEndWord(String startWord)`: Meminta pengguna untuk memasukkan kata akhir dan memvalidasi masukan tersebut. Metode ini akan memastikan kata yang dimasukkan valid, memiliki panjang yang sama dengan kata awal, dan berbeda dari kata awal.
- ❖ `isValidWord(String word)`: Memeriksa apakah kata yang diberikan terdapat dalam kamus atau tidak.

BAB IV

PENGUJIAN DAN ANALISIS

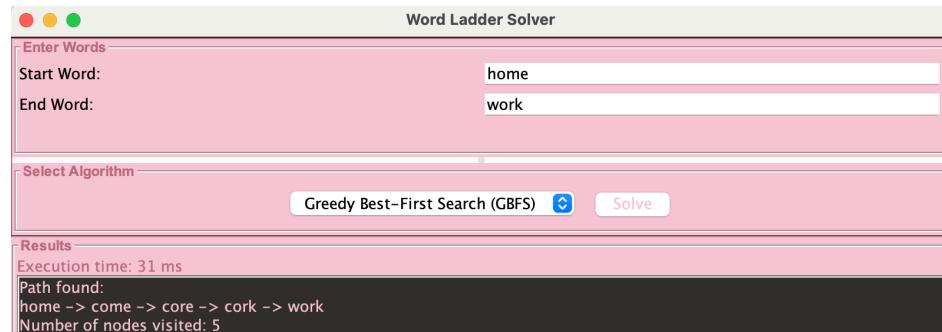
4.1. Test Case 1 (home-work)

4.1.1. UCS



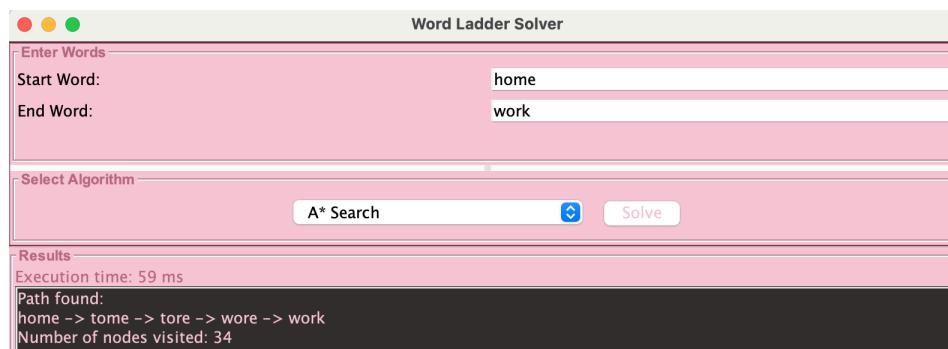
Gambar 4.1. Kasus Pengujian “home-work” dengan UCS

4.1.2. Greedy Best First Search



Gambar 4.2. Kasus Pengujian “home-work” dengan Greedy Best First Search

4.1.3. A*



Gambar 4.3. Kasus Pengujian “home-work” dengan A*

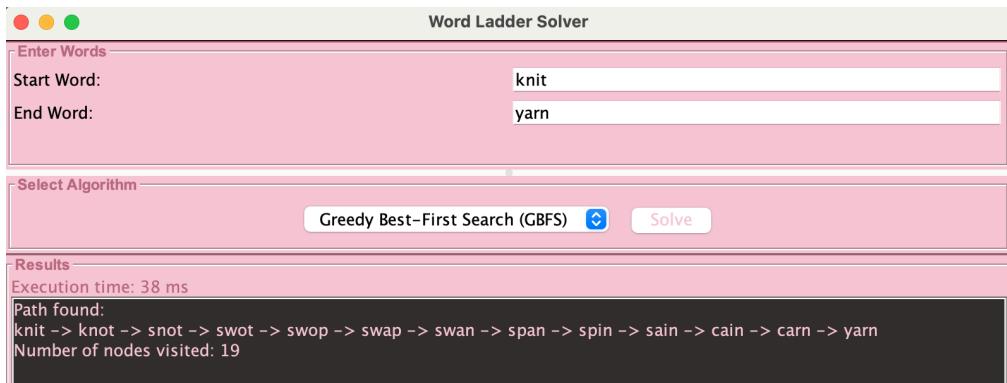
4.2. Test Case 2 (*knit - yarn*)

4.2.1. UCS



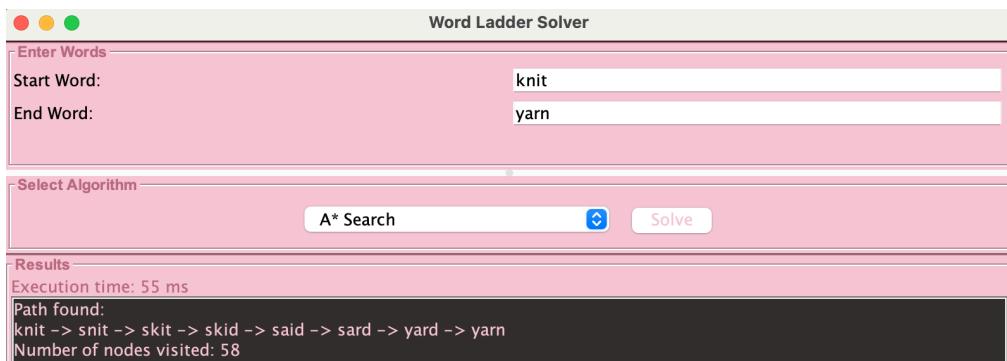
Gambar 4.4. Kasus Pengujian “*knit-yarn*” dengan UCS

4.2.2. Greedy Best First Search



Gambar 4.5. Kasus Pengujian “*knit-yarn*” dengan Greedy Best First Search

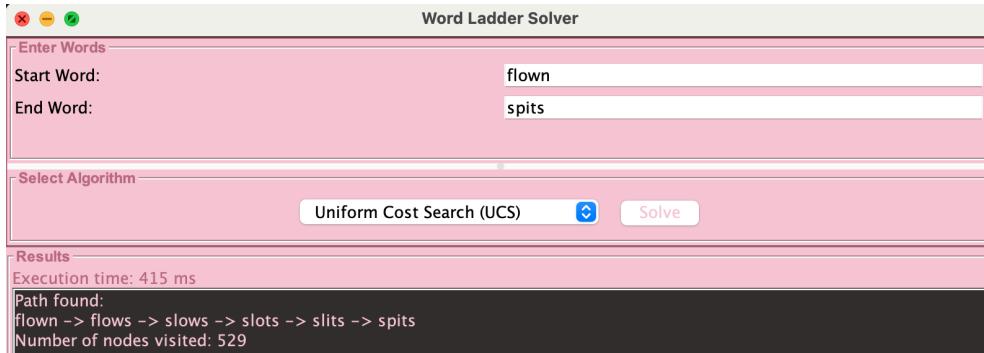
4.2.3. A*



Gambar 4.6. Kasus Pengujian “*knit-yarn*” dengan A*

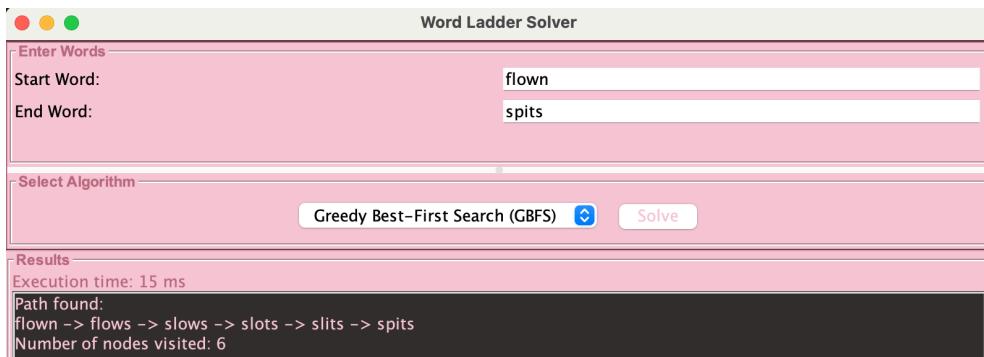
4.3. Test Case 3 (*flown - spits*)

4.3.1. UCS



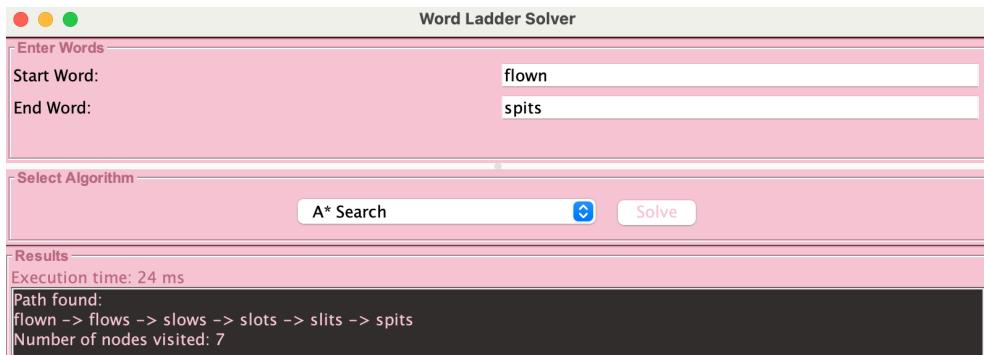
Gambar 4.7. Kasus Pengujian “*flown-spits*” dengan UCS

4.3.2. Greedy Best First Search



Gambar 4.8. Kasus Pengujian “*flown-spits*” dengan Greedy Best First Search

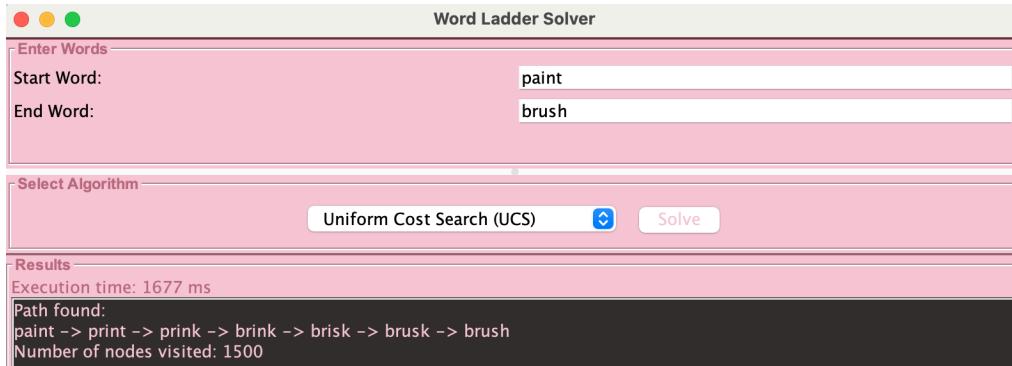
4.3.3. A*



Gambar 4.9. Kasus Pengujian “*flown-spits*” dengan A*

4.4. Test Case 4 (*paint - brush*)

4.4.1. UCS



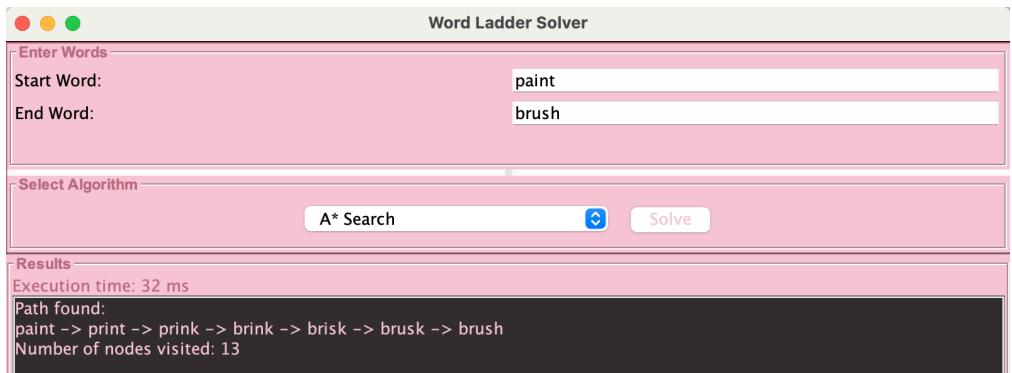
Gambar 4.10. Kasus Pengujian “*paint-brush*” dengan UCS

4.4.2. Greedy Best First Search



Gambar 4.11. Kasus Pengujian “*paint-brush*” dengan Greedy Best First Search

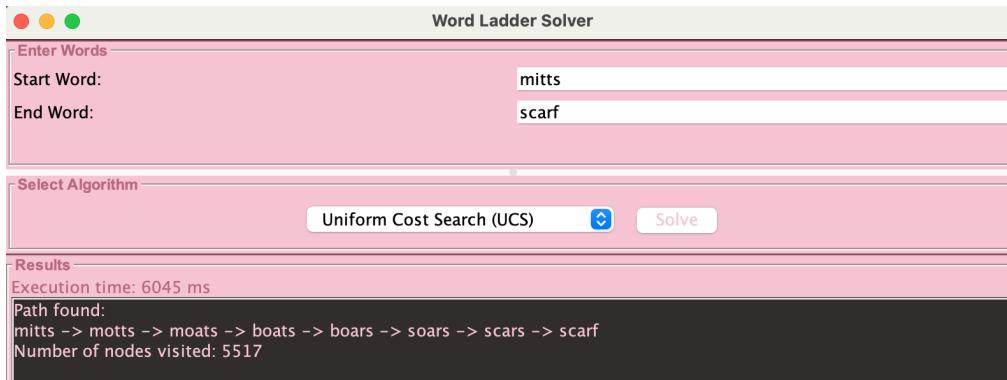
4.4.3. A*



Gambar 4.12. Kasus Pengujian “*paint-brush*” dengan A*

4.5. Test Case 5 (*mitts - scarf*)

4.5.1. UCS



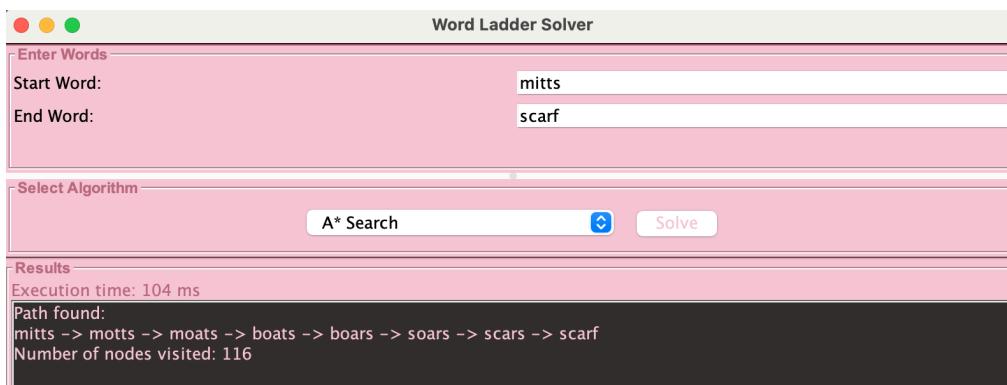
Gambar 4.1. Kasus Pengujian “mitts-scarf” dengan UCS

4.5.2. Greedy Best First Search



Gambar 4.2. Kasus Pengujian “mitts-scarf” dengan Greedy Best First Search

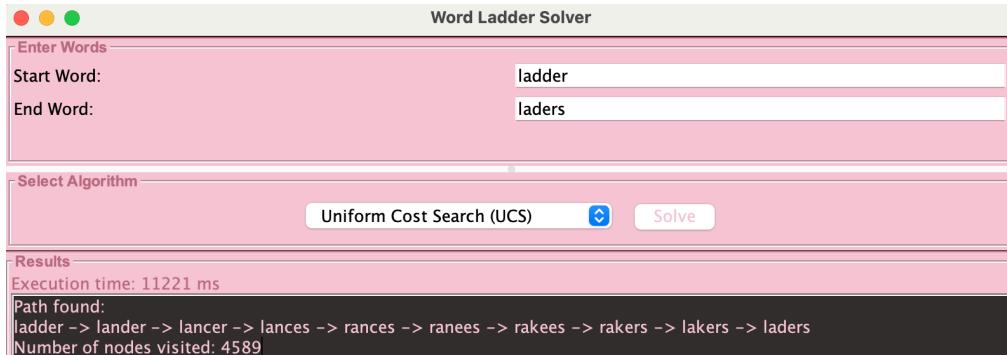
4.5.3. A*



Gambar 4.3. Kasus Pengujian “mitts-scarf” dengan A*

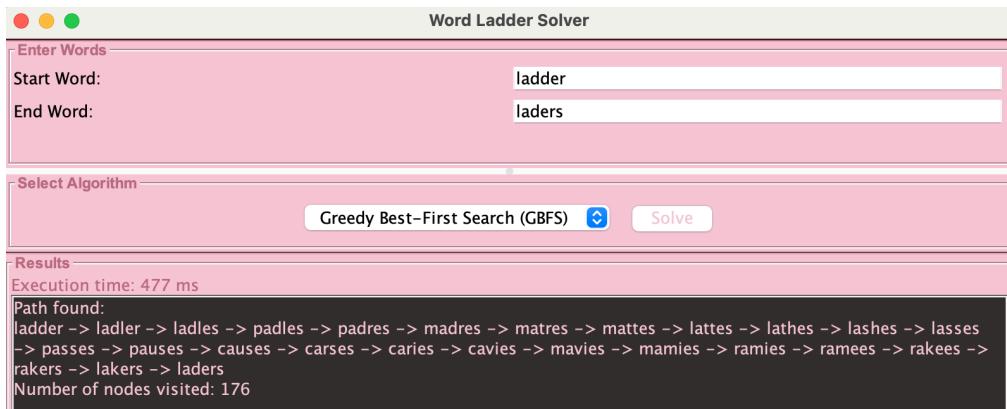
4.6. Test Case 6 (*ladder - laders*)

4.6.1. UCS



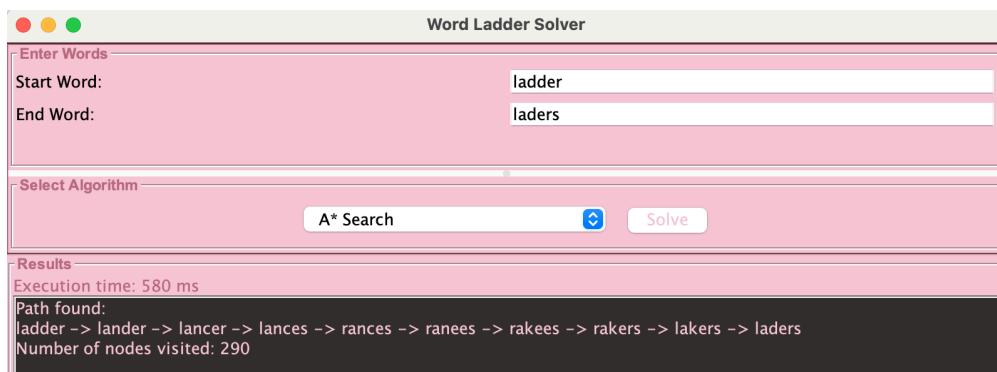
Gambar 4.1. Kasus Pengujian “*ladder-laders*” dengan UCS

4.6.2. Greedy Best First Search



Gambar 4.2. Kasus Pengujian “*ladder-laders*” dengan Greedy Best First Search

4.6.3. A*



Gambar 4.3. Kasus Pengujian “*ladder-laders*” dengan A*

4.7. Analisis Perbandingan Solusi

4.7.1. Optimalitas

Tabel 4.1. Tabel Perbandingan “Optimalitas”

No	Test Case	Panjang Solusi		
		UCS	GBFS	A*
1	home - work	5	5	5
2	knit - yarn	8	13	8
3	flown - spits	6	6	6
4	paint - brush	7	7	7
5	mitts - scarf	8	18	8
6	ladder - laders	10	26	10

Berdasarkan optimalitasnya, algoritma UCS dan A* memberikan solusi dengan jalur terpendek antara kata awal dan kata akhir, sedangkan algoritma Greedy Best First Search terkadang tidak memberikan solusi dengan jalur terpendek. UCS mempertimbangkan biaya setiap langkah dengan merencanakan rute terpendek, sedangkan A* menggabungkan biaya langkah yang sebenarnya dengan estimasi biaya tersisa (heuristik) untuk memilih jalur terpendek. Greedy Best First Search tidak dianggap optimal karena dapat memberikan solusi dengan jalur yang lebih panjang. GBFS hanya fokus pada simpul yang paling dekat dengan tujuan berdasarkan heuristik, tanpa mempertimbangkan biaya total perjalanan.

Pada kasus no 2, 5, dan 6 diatas GBFS tidak menghasilkan solusi yang optimal karena ia hanya memeriksa optimum lokal dan melihat estimasi antara kata saat ini dan kata akhir tanpa mempertimbangkan biaya total perjalanan yang sudah terakumulasi dari kata awal hingga kata saat ini. Sehingga, GBFS mungkin memilih jalur yang terlihat lebih dekat ke tujuan tetapi sebenarnya memerlukan lebih banyak langkah.

4.7.2. Waktu Eksekusi

Tabel 4.2. Tabel Perbandingan “Waktu Eksekusi”

No	Test Case	Waktu Eksekusi (ms)		
		UCS	GBFS	A*
1	home - work	747	31	59
2	knit - yarn	1626	38	55
3	flown - spits	415	15	24
4	paint - brush	6045	39	104
5	mitts - scarf	1677	62	32
6	ladder - laders	11221	477	580

Berdasarkan waktu eksekusinya, terlihat bahwa algoritma yang tercepat adalah algoritma Greedy Best First Search. Algoritma tersebut merupakan algoritma tercepat karena hanya fokus pada simpul yang paling dekat dengan tujuan. GBFS memiliki ruang pencarian yang memprioritaskan heuristik dan algoritma ini tidak melihat jarak yang telah ditempuh untuk mencapai suatu kata.

Algoritma yang terlambat adalah UCS karena UCS mencoba semua kemungkinan kata anak tanpa memprioritaskan kata-kata yang menjanjikan. Sehingga dapat mengakibatkan waktu eksekusi yang lebih lama terutama jika terdapat banyak kemungkinan langkah dan biaya yang berbeda-beda. Waktu eksekusi UCS cenderung meningkat secara linear seiring dengan meningkatnya kompleksitas masalah.

Waktu eksekusi A* berada diantara UCS dan GBFS karena heuristiknya memandu pencarian menuju tujuan secara lebih efisien. Namun, kualitas heuristik dapat memengaruhi kinerja A*. Heuristik yang buruk dapat menghasilkan estimasi biaya yang tidak akurat, sehingga dapat memperpanjang waktu eksekusi.

Terdapat outlier dalam segi waktu eksekusi pada test case 5. Algoritma Greedy Best First Search memiliki waktu eksekusi yang lebih cepat daripada algoritma A*. Hal tersebut dapat terjadi karena algoritma A* saat itu mungkin tidak memberikan estimasi biaya yang akurat.

4.7.3. Memori yang Dibutuhkan

Tabel 4.3. Tabel Perbandingan “Memori yang Dibutuhkan”

No	Test Case	Memori yang Dibutuhkan		
		UCS	GBFS	A*
1	home - work	789	5	34
2	knit - yarn	2257	19	58
3	flown - spits	529	6	7
4	paint - brush	5517	41	116
5	mitts - scarf	1500	7	13
6	ladder - laders	4589	176	290

Berdasarkan memori yang dibutuhkan, Greedy Best First Search membutuhkan lebih sedikit memori dibandingkan dengan UCS dan A* pada sebagian besar kasus uji. Algoritma GBFS umumnya membutuhkan sedikit memori karena hanya mempertimbangkan simpul yang paling dekat dengan tujuan. GBFS tidak menyimpan informasi tentang biaya sepanjang jalur atau jalur itu sendiri, sehingga membutuhkan lebih sedikit ruang penyimpanan.

Algoritma A* sedikit lebih banyak dibandingkan GBFS karena A* perlu menyimpan informasi tambahan seperti biaya sepanjang jalur dan estimasi biaya tersisa hingga tujuan.

Sedangkan algoritma UCS cenderung membutuhkan lebih banyak memori dibandingkan GBFS dan A*. UCS perlu menyimpan informasi tentang semua simpul yang dieksplorasi, termasuk biaya yang diperlukan untuk mencapai setiap simpul dan jalur yang telah diambil. Sehingga, semakin banyak simpul yang dieksplorasi, semakin banyak memori yang diperlukan.



Gambar 4.19. Kasus Khusus Heuristic

Terdapat kasus dimana GBFS memiliki simpul yang dilewati sama dengan A*. Dalam contoh *hat - hap*, GBFS dan A* mungkin memilih jalur yang sama karena keduanya menggunakan heuristic yang serupa atau memiliki perbedaan yang kecil dalam estimasi biaya yang tersisa hingga tujuan. Meskipun A* secara teoritis lebih optimal karena menggunakan biaya sepanjang jalur yang telah ditempuh ditambah estimasi biaya tersisa hingga tujuan (heuristik), namun jika heuristic yang digunakan tidak cukup membedakan antara simpul yang berdekatan, maka hasilnya mungkin serupa dengan GBFS.

4.8. Analisis Keunggulan dan Kelemahan

Berdasarkan kajian teori dan uji coba yang telah dilakukan, terdapat beberapa keunggulan dan kelemahan masing-masing algoritma antara lain

4.8.1. Uniform Cost Search (UCS)

- ❖ Keuntungan: Menjamin rute terpendek karena mengeksplorasi semua kemungkinan jalur dengan *cost* yang meningkat secara bertahap.
- ❖ Kekurangan: Waktu eksekusi dan kebutuhan memori yang lebih tinggi karena harus mengeksplorasi semua kemungkinan jalur.

4.8.2. Greedy Best First Search (GBFS)

- ❖ Keuntungan: Waktu eksekusi dan kebutuhan memori terkecil.
- ❖ Kekurangan: Tidak menjamin rute terpendek karena hanya memprioritaskan simpul yang paling dekat dengan tujuan, tanpa memperhatikan biaya sepanjang jalur yang telah ditempuh.

4.8.3. A*

- ❖ Keuntungan: Menjamin rute terpendek dengan memanfaatkan heuristik untuk mengarahkan pencarian menuju tujuan secara lebih efisien daripada UCS. Waktu eksekusi dan kebutuhan memori lebih sedikit daripada UCS karena menggabungkan keunggulan UCS dengan penggunaan heuristik.
- ❖ Kekurangan: Bergantung pada kualitas heuristik yang digunakan, karena heuristik yang buruk dapat menghasilkan estimasi biaya yang tidak akurat.

BAB V

PENJELASAN IMPLEMENTASI BONUS

Bonus yang diimplementasikan berupa graphical user interface (GUI) untuk aplikasi Word Ladder Solver. GUI dibuat dengan melakukan import dari package `javax.swing` dan `java.awt`. Java AWT adalah kumpulan kelas untuk membuat dan mengelola antarmuka pengguna grafis (GUI) dalam aplikasi Java. Sedangkan Java Swing adalah kerangka kerja GUI yang lebih modern dan fleksibel dibandingkan dengan AWT. Salah satu keuntungan utama Swing adalah portabilitas yang lebih besar; tampilan antarmuka pengguna seragam di berbagai platform. Swing menggunakan pemetaan tampilan sendiri, sehingga tampilan komponen sama di semua platform.



Gambar 5.1. Antarmuka Word Ladder Solver

5.1. Komponen GUI

- ❖ JTextField: Digunakan untuk memasukkan kata awal dan akhir.
- ❖ JTextArea: Digunakan untuk menampilkan hasil pencarian.
- ❖ JLabel: Digunakan untuk menampilkan waktu eksekusi.
- ❖ JPanel: Digunakan untuk mengelompokkan elemen-elemen GUI.
- ❖ JComboBox: Digunakan untuk memilih algoritma yang akan digunakan.
- ❖ JButton: Digunakan untuk memulai pencarian kata.
- ❖ JSplitPane: Digunakan untuk mengatur tata letak panel masukan dan panel algoritma secara vertikal.
- ❖ JScrollPane: Digunakan untuk memberikan scroll ke area hasil.

5.2. Cara Penggunaan

- 1) Pastikan berada di direktori *bin* dari proyek tersebut dalam terminal atau command prompt
- 2) Jalankan perintah untuk menjalankan program Java GUI dengan mengetikkan perintah berikut: `run GUI`
- 3) Setelah menjalankan perintah tersebut, jendela GUI akan muncul di layar.
- 4) Masukkan kata awal dan akhir yang ingin dicari jalur antar-kata (Word Ladder).
- 5) Pilih algoritma yang akan digunakan dari menu drop-down.
- 6) Klik tombol "Solve" untuk memulai pencarian.
- 7) Hasil pencarian akan ditampilkan di area hasil bersama dengan jumlah node yang dikunjungi beserta waktu eksekusi.

DAFTAR PUSTAKA

- [1] Munir, R. (2021). *informatika.stei.itb.ac.id*, “Strategi Algoritma: Penentuan Rute (Route/Path Planning) Bagian I”. Diakses Sabtu, 4 Mei 2024. Dilansir dari Laman Rinaldi Munir: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf>
- [2] Munir, R. (2021). *informatika.stei.itb.ac.id*, “Strategi Algoritma: Penentuan Rute (Route/Path Planning) Bagian II”. Diakses Sabtu, 4 Mei 2024. Dilansir dari Laman Rinaldi Munir: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf>

LAMPIRAN

Lampiran 1 Link Repository

Repository untuk source code dan hasil kompilasi program ini dapat diakses melalui tautan berikut: https://github.com/nabilashikoafa/Tucil3_13522069

Lampiran 2 Tabel *Checklist Poin*

No.	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	✓	