

IF2211 Strategi Algoritma
**Implementasi Permainan Word Ladder Menggunakan Algoritma
*Uniform-Cost Search, A**, dan *Greedy Best-First Search***

Laporan Tugas Kecil 3

Disusun untuk memenuhi tugas mata kuliah Strategi Algoritma
pada Semester 2 (dua) Tahun Akademik 2023/2024



Disusun oleh:

Rizqika Mulia Pratama (13522126)

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

**BANDUNG
2024**

DAFTAR ISI

DAFTAR ISI	2
BAB I	3
1.1. Abstrak	3
1.2. Pendahuluan	3
1.3. Landasan Teori	4
1.3.1. Word Ladder	4
1.3.2. Algoritma Uniform-Cost Search	5
1.3.3. Algoritma A*	5
1.3.4. Algoritma Greedy Best-First Search	5
1.3.5. Bahasa Pemrograman Java	5
BAB II	6
2.1. Spesifikasi Tugas	6
BAB III	7
3.1. Implementasi Algoritma Uniform-Cost Search	7
3.2. Implementasi Algoritma A*	8
3.3. Implementasi Algoritma Greedy Best-First Search	9
BAB IV	11
4.1. Source Code	11
4.2. Pengetesan Kode	20
BAB V	25
5.1. Analisis Perbandingan Algoritma UCS, A*, dan Greedy BFS dalam Pencarian Jalur dalam Game Word Ladder	25
5.1.1. Optimalitas	25
5.1.2. Waktu Eksekusi	26
5.1.3. Penggunaan Memori	27
BAB VI	28
6.1. Kesimpulan	28
6.2. Saran	28
BAB VII	29
7.1. Sumber Pustaka	29
7.2. Lampiran	29

BAB I

PENDAHULUAN

1.1. Abstrak

Makalah ini membahas implementasi permainan Word Ladder menggunakan tiga pendekatan algoritma pencarian: Uniform-Cost Search, A*, dan Greedy Best-First Search. Word Ladder adalah permainan kata-kata di mana pemain mengubah satu kata menjadi kata lain dengan mengganti satu huruf pada satu waktu, dengan setiap langkah menghasilkan kata yang valid. Tujuan dari penelitian ini adalah untuk menganalisis dan membandingkan kinerja ketiga algoritma tersebut dalam menyelesaikan permainan Word Ladder. Algoritma Uniform-Cost Search memberikan jaminan solusi optimal, tetapi mungkin memerlukan waktu yang lebih lama. Algoritma A* menggabungkan pendekatan Uniform-Cost Search dengan heuristic untuk mempercepat pencarian, sementara Greedy Best-First Search lebih fokus pada heuristic tanpa jaminan solusi optimal. Dalam makalah ini, dilakukan evaluasi waktu eksekusi, jumlah langkah yang diperlukan, dan kompleksitas komputasi dari setiap algoritma. Hasil penelitian ini memberikan wawasan mengenai keunggulan dan keterbatasan masing-masing algoritma dalam konteks permainan Word Ladder, yang dapat membantu dalam memilih metode pencarian yang paling efektif sesuai kebutuhan aplikasi.

1.2. Pendahuluan



Gambar 1 Lewis Carroll

(Sumber: <https://poemanalysis.com/lewis-carroll/biography/>)

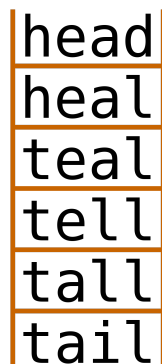
Permainan Word Ladder, yang diciptakan oleh Lewis Carroll pada tahun 1879, merupakan permainan kata yang menantang dan menghibur. Permainan ini mengharuskan pemain mengubah satu kata menjadi kata lain dengan mengganti satu huruf pada satu waktu, dengan setiap perubahan menghasilkan kata yang valid. Meskipun sederhana dalam konsep, permainan ini dapat menjadi sangat kompleks dan menuntut strategi yang baik untuk mencapai hasil yang optimal.

Di era kecerdasan buatan dan algoritma pencarian, permainan Word Ladder menjadi studi kasus menarik yang melibatkan masalah pencarian jalur dalam sebuah graf. Masing-masing kata dalam permainan dapat dianggap sebagai simpul dalam graf, dan transisi dari satu kata ke kata lain melalui perubahan huruf mewakili sisi graf. Tujuan permainan adalah menemukan jalur terpendek dari kata awal ke kata akhir, dengan sejumlah minimal langkah dan validitas kata-kata yang digunakan.

Algoritma pencarian seperti Uniform-Cost Search, A*, dan Greedy Best-First Search adalah beberapa pendekatan populer untuk menyelesaikan masalah pencarian jalur. Algoritma-algoritma ini memiliki kelebihan dan kekurangan masing-masing, dan pilihan yang tepat tergantung pada konteks dan kriteria pencarian. Uniform-Cost Search memastikan jalur optimal tetapi seringkali memakan waktu lebih lama. A* menggabungkan pendekatan optimalitas dengan penggunaan heuristic untuk mempercepat pencarian. Sementara itu, Greedy Best-First Search menggunakan heuristic secara lebih agresif, yang dapat meningkatkan kecepatan tetapi mengorbankan jaminan optimalitas.

1.3. Landasan Teori

1.3.1. Word Ladder



head
heal
teal
tell
tall
tail

Gambar 2 Word Ladder

(Sumber: https://en.wikipedia.org/wiki/Word_ladder)

Word Ladder adalah permainan kata yang pertama kali diperkenalkan oleh Lewis Carroll, penulis "Alice's Adventures in Wonderland", pada tahun 1879. Konsep dasar permainan ini adalah mengubah satu kata menjadi kata lain dengan mengganti satu huruf pada satu waktu, dengan setiap perubahan menghasilkan kata yang valid dalam bahasa tertentu. Dalam permainan ini, pemain mencoba mencapai kata akhir dalam jumlah langkah sesedikit mungkin, yang menjadikannya tantangan untuk menemukan jalur terpendek dalam sebuah graf yang mewakili kata-kata dan transformasinya.

1.3.2. Algoritma Uniform-Cost Search

Algoritma Uniform-Cost Search (UCS) adalah salah satu algoritma pencarian yang digunakan untuk menemukan jalur terpendek dalam graf berbobot. UCS bekerja dengan prinsip mengunjungi simpul yang memiliki biaya terendah terlebih dahulu, tanpa memperhatikan kedalaman simpul dalam pohon pencarian. Algoritma ini umumnya digunakan dalam kasus di mana bobot sisi (edge) bervariasi dan tidak diketahui sebelumnya.

1.3.3. Algoritma A*

Algoritma A* (dibaca "A-star") adalah algoritma pencarian jalur yang menggabungkan pendekatan eksplorasi berdasarkan biaya (seperti Uniform-Cost Search) dengan penggunaan fungsi heuristik untuk memperkirakan biaya dari simpul tertentu ke simpul tujuan. Pendekatan ini membuat A* lebih efisien dalam menemukan jalur terpendek karena dapat "mengarahkan" pencarian ke arah yang lebih menjanjikan.

Penggunaan heuristik adalah yang membedakan A* dari algoritma pencarian lainnya. Heuristik berfungsi sebagai "panduan" yang membantu algoritma memfokuskan pencarian pada area yang lebih mungkin mengarah ke jalur optimal. Agar A* tetap optimal dan konsisten, heuristik yang digunakan harus bersifat "admissible" (tidak melebihi-lebihkan biaya sebenarnya) dan "consistent" (mengikuti prinsip segitiga).i.

1.3.4. Algoritma Greedy Best-First Search

Algoritma Greedy Best-First Search adalah algoritma pencarian yang menggunakan strategi heuristik untuk memilih simpul berikutnya yang akan dieksplorasi. Greedy Best-First Search bersifat "serakah" karena selalu memilih simpul yang dianggap paling menjanjikan berdasarkan heuristik tanpa mempertimbangkan biaya kumulatif dari simpul awal. Hal ini membuat algoritma ini cepat dalam mengeksplorasi graf, tetapi tanpa jaminan menemukan jalur terpendek.

Greedy Best-First Search mengandalkan heuristik untuk menentukan simpul yang akan dieksplorasi berikutnya. Berbeda dengan A*, algoritma ini tidak mempertimbangkan biaya kumulatif, melainkan hanya fokus pada nilai heuristik. Heuristik yang digunakan harus dapat memperkirakan seberapa dekat simpul saat ini dengan simpul tujuan.

1.3.5. Bahasa Pemrograman Java

Java adalah salah satu bahasa pemrograman yang paling populer dan digunakan secara luas di dunia teknologi. Dikenal karena kemampuannya untuk dijalankan di berbagai platform dan dukungan komunitas yang kuat, Java memainkan peran penting dalam pengembangan aplikasi, baik di lingkungan perusahaan maupun konsumen.

BAB II

SPESIFIKASI

2.1. Spesifikasi Tugas

- Buatlah program dalam bahasa Java berbasis CLI (Command Line Interface) - bonus jika menggunakan GUI - yang dapat menemukan solusi permainan word ladder menggunakan algoritma UCS < Greedy Best First Search, dan A*
- Kata-kata yang dapat dimasukkan harus berbahasa Inggris. Cara kalian melakukan validasi sebuah kata dibebaskan, selama kata-kata tersebut benar terdapat pada dictionary dan proses validasi tersebut tidak memakan waktu yang terlalu lama.
- Tugas wajib dikerjakan secara individu
- **Input :**
Format masukan **dibebaskan**, dengan catatan dijelaskan pada README dan laporan.
Komponen yang perlu menjadi masukan yaitu:
 1. Start word dan end word. Program harus bisa menangani berbagai panjang kata (tidak hanya kata dengan 4 huruf saja seperti contoh).
 2. Pilihan algoritma yang digunakan (UCS, Greedy Best First Search, atau A*)
- **Output:**
Berikut adalah luaran dari program yang diekspektasikan:
 1. Path yang dihasilkan dari start word ke end word (cukup 1 path saja)
 2. Banyak node yang dikunjungi.
 3. Waktu eksekusi program
- **Bonus:**
Pastikan sudah mengerjakan spesifikasi wajib sebelum mengerjakan bonus.
 1. Program dapat berjalan dengan GUI (Graphical User Interface) - silakan berkreasi dalam membuat tampilan GUI untuk tucil ini. Untuk kakas GUI dibebaskan asalkan program algoritma UCS < Greedy Best First Search, dan A* dalam bahasa Java

BAB III

ANALISIS DAN IMPLEMENTASI ALGORITMA

3.1. Implementasi Algoritma *Uniform-Cost Search*

Uniform-Cost Search (UCS) adalah algoritma pencarian yang bertujuan menemukan jalur terpendek dengan mengutamakan node yang memiliki biaya terendah. Biaya ini bisa berarti jumlah langkah atau besaran lain yang dapat diukur. Implementasi UCS dalam konteks Word Ladder berusaha menemukan urutan transformasi kata yang paling efisien dari kata awal hingga kata akhir.

Pada awalnya, program membaca file teks yang berisi daftar kata. Setiap kata berada pada baris yang berbeda dalam file teks tersebut. File ini diambil dan setiap kata dimasukkan ke dalam Set, yang memungkinkan pencarian cepat ketika memeriksa apakah suatu kata ada dalam kamus. Set ini adalah inti dari kamus kata-kata yang dapat digunakan untuk menemukan jalur dalam Word Ladder.

Setelah kamus disiapkan, program meminta input dari pengguna untuk kata awal dan kata akhir. Program memvalidasi bahwa kedua kata ini ada dalam kamus dan memiliki panjang yang sama. Jika salah satu dari kata tidak valid, pengguna diminta untuk memasukkan ulang hingga kata-kata yang valid diberikan.

Dalam implementasi UCS, algoritma dimulai dengan menambahkan kata awal ke dalam priority queue yang memiliki urutan prioritas berdasarkan jumlah langkah, atau yang disebut steps. Priority queue ini dirancang sedemikian rupa sehingga setiap kali node diambil dari antrian, node dengan biaya terendah yang selalu diambil terlebih dahulu. Dalam kasus UCS, biaya ini hanya didasarkan pada jumlah langkah yang diperlukan untuk mencapai node tersebut.

Saat UCS berjalan, algoritma mengekstrak node dari priority queue, yang selalu akan memiliki biaya terendah. Node ini kemudian diekspansi untuk menemukan tetangga, yaitu kata-kata yang berbeda hanya satu karakter dari kata tersebut. Ekspansi dilakukan dengan mengubah setiap karakter pada kata dengan alfabet dan memeriksa apakah kata yang dihasilkan ada di dalam kamus. Jika ya, kata ini dianggap sebagai tetangga dan ditambahkan ke priority queue dengan biaya langkah yang meningkat. Setelah node diekspansi, kata tersebut dihapus dari kamus (Set) untuk menghindari eksplorasi ulang, yang membantu menjaga efisiensi dan menghindari siklus.

Selama proses ini, jika node yang diekstraksi adalah kata akhir, algoritma berhenti dan mulai merekonstruksi jalur dari kata akhir ke kata awal. Rekonstruksi ini dilakukan dengan

mengikuti referensi 'prev' dalam setiap node. Dengan cara ini, jalur yang ditemukan bisa dibangun kembali dari akhir ke awal, dan kemudian dapat ditampilkan kepada pengguna.

Dalam UCS, prioritas setiap node ditentukan oleh $f(n) = g(n)$, di mana $g(n)$ adalah jumlah langkah dari awal hingga node tersebut. Dalam implementasi ini, UCS hanya mempertimbangkan biaya berbasis langkah tanpa memperhitungkan heuristik tambahan.

Dengan pendekatan ini, implementasi UCS berhasil menemukan jalur terpendek dari kata awal hingga kata akhir dalam konteks Word Ladder, memastikan bahwa jalur yang ditemukan adalah yang paling efisien berdasarkan jumlah langkah. Meskipun UCS hanya menggunakan biaya berbasis langkah, hasilnya masih optimal untuk konteks ini karena memastikan node dengan biaya terendah selalu diekstraksi dan diekspansi terlebih dahulu.

3.2. Implementasi Algoritma A*

Algoritma A* adalah metode pencarian jalur yang menggunakan heuristik untuk memperkirakan biaya sisa menuju tujuan, digabungkan dengan biaya langkah yang sudah dikeluarkan. Implementasi A* dalam konteks Word Ladder mengutamakan jalur yang paling efisien dengan mempertimbangkan jarak antara kata saat ini dan kata akhir.

Program dimulai dengan membaca file teks yang berisi daftar kata. Daftar ini diubah menjadi Set, yang menjadi kamus untuk memeriksa validitas kata dan mencari tetangga (kata yang berbeda satu karakter). Setelah kamus siap, pengguna diminta untuk memasukkan kata awal dan kata akhir. Program memastikan kedua kata ini valid dan memiliki panjang yang sama.

Dalam algoritma A*, prioritas ditentukan oleh fungsi $f(n)$, yaitu jumlah dari $g(n)$ dan $h(n)$. Dalam implementasi ini, $g(n)$ adalah biaya langkah yang telah dikeluarkan, sementara $h(n)$ adalah estimasi jarak antara kata saat ini dan kata akhir, dihitung sebagai jumlah karakter yang berbeda. Dengan kombinasi ini, prioritas ditentukan oleh gabungan langkah dan heuristik, memungkinkan algoritma untuk mengekstraksi node dengan prioritas terendah dan melakukan ekspansi.

Algoritma A* mengekstrak node dari priority queue, dengan prioritas terendah berdasarkan $f(n)$. Ekspansi dilakukan dengan mengganti setiap karakter dalam kata saat ini dengan alfabet, mencari kata baru yang hanya berbeda satu karakter. Jika kata baru ini ada dalam kamus, itu dianggap sebagai tetangga dan ditambahkan ke priority queue dengan biaya langkah yang meningkat. Untuk menghindari eksplorasi ulang, kata-kata yang diekspansi dihapus dari kamus. Jika kata akhir ditemukan, algoritma berhenti dan merekonstruksi jalur dari akhir ke awal dengan mengikuti referensi 'prev'.

Salah satu kriteria penting dalam algoritma A* adalah bahwa heuristik harus admissible. Heuristik disebut admissible jika tidak melebihi-lebihkan biaya aktual menuju tujuan, sehingga memastikan bahwa jalur yang ditemukan oleh A* adalah optimal. Dalam implementasi ini, heuristik dihitung sebagai jumlah karakter yang berbeda antara kata saat ini dan kata akhir. Karena ini adalah estimasi minimal dari langkah-langkah yang diperlukan untuk mencapai tujuan (setiap karakter yang berbeda harus diubah), heuristik ini dapat dianggap admissible.

Secara teoritis, algoritma A* bisa lebih efisien dibandingkan UCS pada kasus Word Ladder. UCS hanya mempertimbangkan biaya langkah yang telah diambil, tanpa mempertimbangkan informasi apa pun tentang tujuan. A* menggunakan heuristik untuk memperkirakan biaya sisa menuju tujuan, sehingga dapat mengarahkan pencarian lebih efektif. Dalam konteks Word Ladder, di mana setiap langkah melibatkan perubahan satu karakter, heuristik yang menghitung jumlah karakter yang berbeda antara kata saat ini dan kata akhir bisa mempercepat pencarian dengan mengarahkan eksplorasi ke arah yang lebih relevan.

Secara keseluruhan, implementasi algoritma A* ini menggabungkan biaya langkah dengan estimasi menuju tujuan untuk menentukan prioritas ekspansi, yang memungkinkan pencarian jalur yang lebih efisien. Dengan heuristik yang admissible, algoritma A* dapat menemukan jalur yang optimal sambil menghemat waktu dan sumber daya dengan fokus pada jalur yang paling menjanjikan.

3.3. Implementasi Algoritma *Greedy Best-First Search*

Algoritma Greedy Best-First Search (Greedy BFS) adalah metode pencarian yang mengarahkan eksplorasi berdasarkan estimasi jarak menuju tujuan, yang dikenal sebagai heuristik. Dalam kasus Word Ladder, heuristik ini dapat dihitung sebagai jumlah karakter yang berbeda antara kata saat ini dan kata tujuan. Greedy BFS berbeda dengan algoritma lain seperti A* atau Uniform-Cost Search (UCS), karena algoritma ini hanya mempertimbangkan heuristik untuk menentukan prioritas ekspansi tanpa memperhitungkan biaya langkah yang telah ditempuh.

Implementasi Greedy BFS dimulai dengan membaca file teks yang berisi daftar kata-kata. Setiap kata dalam file ini berada pada baris yang berbeda. Setelah membaca file, daftar kata ini diubah menjadi Set, yang menjadi kamus untuk memeriksa validitas kata dan mencari tetangga. Set digunakan untuk memastikan bahwa pencarian kata cepat dan efisien.

Selanjutnya, pengguna diminta untuk memasukkan kata awal dan kata akhir. Program memvalidasi apakah kedua kata ini ada dalam kamus dan memiliki panjang yang sama. Validasi

ini penting untuk memastikan bahwa kata-kata yang akan diekspansi selama pencarian adalah kata-kata yang valid.

Greedy BFS menggunakan priority queue untuk menentukan prioritas node yang akan diekstraksi dan diekspansi. Dalam algoritma ini, prioritas didasarkan pada heuristik, yang disebut $h(n)$. Heuristik ini mengukur seberapa dekat kata saat ini dengan kata tujuan. Dalam implementasi dengan algoritma ini, heuristik dihitung sebagai jumlah karakter yang berbeda antara kata saat ini dan kata akhir. Semakin rendah nilai $h(n)$, semakin dekat kata tersebut dengan tujuan.

Node awal, yaitu kata awal, ditambahkan ke priority queue, dan algoritma mulai mengekstrak node dengan prioritas terendah berdasarkan heuristik. Saat node diekstraksi, algoritma melakukan ekspansi dengan mengubah setiap karakter dalam kata saat ini dengan alfabet, mencari kata baru yang berbeda satu karakter. Jika kata ini ada dalam kamus, ia ditambahkan ke priority queue, dan kamus diperbarui untuk menghindari eksplorasi ulang. Jika node yang diekstraksi adalah kata akhir, algoritma berhenti dan mulai merekonstruksi jalur dari akhir ke awal dengan mengikuti referensi 'prev'. Jalur ini kemudian dapat ditampilkan kepada pengguna untuk menunjukkan urutan kata yang menghubungkan kata awal dan kata akhir.

Dalam konteks Greedy BFS, $f(n)$, yang menentukan prioritas, sama dengan $h(n)$. Artinya, prioritas ditentukan hanya berdasarkan estimasi jarak menuju tujuan, tanpa memperhitungkan biaya langkah yang telah ditempuh. Dengan demikian, Greedy BFS cenderung mengeksplorasi node yang tampaknya paling dekat dengan tujuan, tetapi ini bisa berarti bahwa algoritma melewati jalur yang mungkin lebih efisien karena tidak memperhitungkan biaya total.

Secara teoritis, algoritma Greedy BFS tidak menjamin solusi optimal untuk persoalan Word Ladder. Karena algoritma ini hanya menggunakan heuristik untuk menentukan prioritas, ada risiko bahwa jalur yang dipilih bukanlah jalur terpendek. Greedy BFS mungkin lebih cepat dalam menemukan jalur karena langsung mengarahkan eksplorasi ke arah tujuan, tetapi bisa gagal menemukan jalur terpendek jika ada jalur alternatif dengan biaya yang lebih rendah.

Dengan demikian, meskipun Greedy BFS dapat menghasilkan solusi yang cepat, ini bukan jaminan bahwa jalur tersebut adalah yang terpendek atau paling efisien. Pada akhirnya, algoritma ini mengorbankan optimalitas demi kecepatan, yang bisa mengakibatkan hasil yang kurang optimal dalam beberapa kasus, termasuk dalam persoalan Word Ladder.

BAB IV

SOURCE CODE DAN PENGETESAN KODE

4.1. Source Code

Main.java

```
1  import java.nio.file.Files;
2  import java.nio.file.Paths;
3  import java.io.IOException;
4  import java.util.*;
5  import java.util.stream.Stream;
6
7  public class Main {
8      public static void main(String[] args) {
9          try {
10             String path = "words_dictionary.txt";
11             System.out.println("Welcome to WordLadder Well!!\n");
12
13             Set<String> dictSet = new HashSet<>();
14
15             // add every line of the txt file to Hashset for a faster look-up.
16             try (Stream<String> stream = Files.lines(Paths.get(path))) {
17                 stream.forEach(dictSet::add);
18             }
19
20             boolean passWord = false;
21             Scanner beginScanner = null;
22             Scanner endScanner = null;
23             Scanner algoScanner = null;
24             String beginWord = "";
25             String endWord = "";
26
27             // enter and validate the start word and end word.
28             while (!passWord){
29                 beginScanner = new Scanner(System.in);
30                 System.out.print("Enter the start word: ");
31                 beginWord = beginScanner.nextLine().toLowerCase();
32                 final String finalBeginWord = beginWord;
33
34                 endScanner = new Scanner(System.in);
35                 System.out.print("Enter the end word: ");
36                 endWord = endScanner.nextLine().toLowerCase();
37                 final String finalEndWord = endWord;
38
39                 if (!dictSet.contains(beginWord) && !dictSet.contains(endWord)){
40                     System.out.println("I'm pretty sure those start word and end word are not English words. Please input English words only.\n");
41                 }
42                 else if(!dictSet.contains(beginWord) && dictSet.contains(endWord)){
43                     System.out.println("I'm pretty sure that start word is not an English word. Please input English words only.\n");
44                 }
45                 else if(dictSet.contains(beginWord) && !dictSet.contains(endWord)){
46                     System.out.println("I'm pretty sure that end word is not an English word. Please input English words only.\n");
47                 }
48                 else if(finalBeginWord.length() != finalEndWord.length()){
49                     System.out.println("You have to enter words with the same length.\n");
50                 }
51                 else{
52                     passWord = true;
53                     dictSet.removeIf(word -> word.length() != finalBeginWord.length());
54                 }
55             }
56         }
```



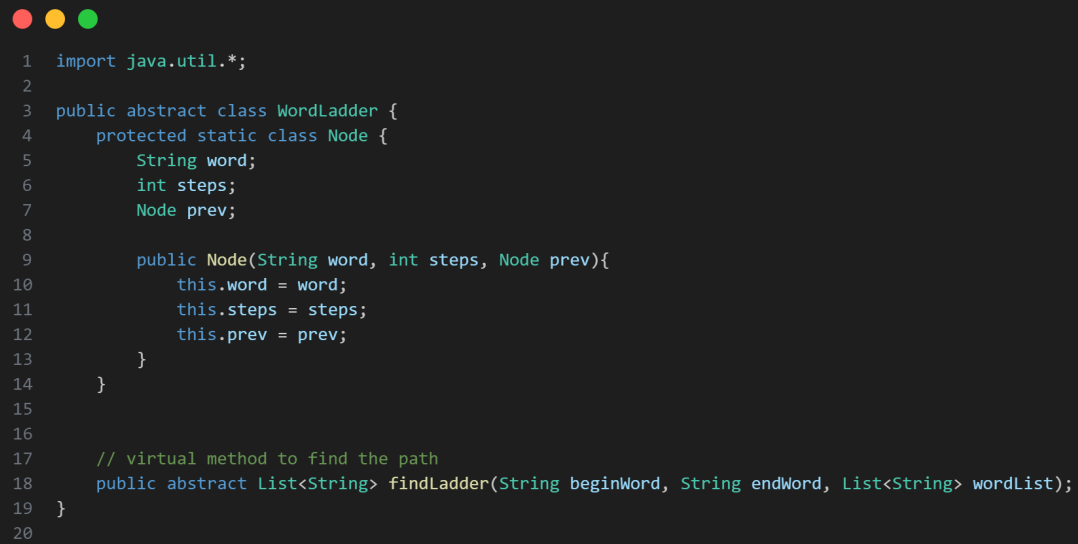
```
1 // convert the previous HashSet into list to be read (the component is definitely unique)
2 List<String> dictList = new ArrayList<>(dictSet);
3
4 boolean passAlgo = false;
5
6 long startTime = 0; // start counting the elapsed time
7
8 // algorithm validation
9 while (!passAlgo){
10     System.out.println("\nChoose the algorithm: ");
11     System.out.println("1. Uniform-Cost Search (UCS)");
12     System.out.println("2. A* Algorithm");
13     System.out.println("3. Greedy Best-First-Search");
14     System.out.print("\nEnter chosen algorithm: ");
15     algoScanner = new Scanner(System.in);
16
17     try {
18         int algorithm = algoScanner.nextInt();
19         System.out.println();
20
21         // if user select 1, then use Uniform-Cost-Search algorithm
22         if (algorithm == 1){
23             startTime = System.currentTimeMillis();
24             passAlgo = true;
25             UCS ucs = new UCS();
26             List<String> ladder = ucs.findLadder(beginWord, endWord, dictList);
27
28             // if the algorithm found nothing than print message to terminal
29             if (ladder.size() == 0){
30                 System.out.println("No path found from " + beginWord + " to " + endWord);
31             }
32
33             // if the algorithm found the path, then print the path to terminal
34             else{
35                 System.out.println("Path: " + String.join(" -> ", ladder));
36             }
37         }
38
39         // if user select 2, then use A* algorithm
40         else if (algorithm == 2){
41             startTime = System.currentTimeMillis();
42             passAlgo = true;
43             Astar astar = new Astar();
44             List<String> ladder = astar.findLadder(beginWord, endWord, dictList);
45
46             // if the algorithm found nothing than print message to terminal
47             if (ladder.size() == 0){
48                 System.out.println("No path found from " + beginWord + " to " + endWord);
49             }
50
51             // if the algorithm found the path, then print the path to terminal
52             else{
53                 System.out.println("Path: " + String.join(" -> ", ladder));
54             }
55         }
56     }
57 }
```

```

1  // if user select 3, then use Greedy Best-First-Search algorithm
2      else if (algorithm == 3){
3          startTime = System.currentTimeMillis();
4          passAlgo = true;
5          GreedyBFS gbfs = new GreedyBFS();
6          List<String> ladder = gbfs.findLadder(beginWord, endWord, dictList);
7
8          // if the algorithm found nothing than print message to terminal
9          if (ladder.size() == 0){
10             System.out.println("No path found from " + beginWord + " to " + endWord);
11         }
12
13         // if the algorithm found the path, then print the path to terminal
14         else{
15             System.out.println("Path: " + String.join(" -> ", ladder));
16         }
17     }
18     }
19     else{
20         System.out.println("There's no choice with that number.");
21     }
22     } catch (InputMismatchException e) {
23         System.out.println("Invalid input. Please enter an integer.");
24         algoScanner.next();
25     }
26 }
27
28 long endTime = System.currentTimeMillis();
29 long executionTime = endTime - startTime; // calculate the elapsed time
30 System.out.println("Execution time: " + executionTime + " ms"); // print the elapsed time
31
32 beginScanner.close();
33 endScanner.close();
34 algoScanner.close();
35 } catch (IOException e) {
36     System.out.println("An error occurred while reading the file.");
37     e.printStackTrace();
38 }
39 }
40 }
41

```

WordLadder.java



```
1  import java.util.*;
2
3  public abstract class WordLadder {
4      protected static class Node {
5          String word;
6          int steps;
7          Node prev;
8
9          public Node(String word, int steps, Node prev){
10             this.word = word;
11             this.steps = steps;
12             this.prev = prev;
13         }
14     }
15
16
17     // virtual method to find the path
18     public abstract List<String> findLadder(String beginWord, String endWord, List<String> wordList);
19 }
20
```

UCS.java

```

1  import java.util.*;
2
3  public class UCS extends WordLadder {
4
5      @Override
6      public List<String> findLadder(String beginWord, String endWord, List<String> wordList){
7
8          // if the end word is not available in the dictionary, then return empty list
9          if (!wordList.contains(endWord)){
10             return Collections.emptyList();
11         }
12
13         Set<String> dict = new HashSet<>(wordList); // convert wordList into a HashSet for faster look-up times
14
15         // initializes a priority queue that orders nodes by the number of steps taken, characteristic of UCS
16         PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(node -> node.steps));
17         queue.add(new Node(beginWord, 1, null));
18
19         // create a list to store the checked nodes
20         List<String> checkedNodes = new ArrayList<>();
21
22         while (!queue.isEmpty()){
23             Node node = queue.remove(); // removes the node with the lowest number of steps from the queue
24             String word = node.word; // stores the current word from the node
25
26             checkedNodes.add(word);
27
28             // checks if the current word is the endWord
29             if (word.equals(endWord)){
30                 List<String> ladder = new ArrayList<>();
31                 while (node != null){
32                     ladder.add(0, node.word);
33                     node = node.prev;
34                 }
35                 System.out.println("Number of Checked Nodes: " + checkedNodes.size());
36                 return ladder;
37             }
38
39             // converts the word to a character array to make checking easier
40             char[] arr = word.toCharArray();
41             for (int i = 0; i < arr.length; i++){ // loops over each character in the array
42                 for (char c = 'a'; c < 'z'; c++){
43                     char temp = arr[i]; // stores the original character for restoration later
44                     if (arr[i] != c){ // change the character if the character is different
45                         arr[i] = c;
46                     }
47
48                     String newWord = new String(arr); // creates a new string from the modified character
49
50                     // if the new word is in the dictionary, add a new node for the new word
51                     if (dict.contains(newWord)){
52                         queue.add(new Node(newWord, node.steps + 1, node));
53                         dict.remove(newWord); // remove the new word from the dictionary to avoid revisiting
54                     }
55
56                     arr[i] = temp; // restores the original character
57                 }
58             }
59         }
60         return Collections.emptyList(); // return empty list if no path found
61     }
62 }
63

```

GreedyBFS.java


```

1  import java.util.*;
2
3  public class GreedyBFS extends WordLadder {
4
5      // method for counting the heuristic feature (the sum of the different character)
6      private int heuristic(String word, String endWord){
7          int diff = 0;
8          for (int i = 0; i < word.length(); i++){
9              if (word.charAt(i) != endWord.charAt(i)){
10                 diff++;
11             }
12         }
13         return diff;
14     }
15
16     @Override
17     public List<String> findLadder(String beginWord, String endWord, List<String> wordList){
18
19         // if the end word is not available in the dictionary, then return empty list
20         if (!wordList.contains(endWord)){
21             return Collections.emptyList();
22         }
23
24         Set<String> dict = new HashSet<>(wordList); // convert wordList into a HashSet for faster look-up times
25
26         // initializes a priority queue that orders nodes solely based on the heuristic value, characteristic of Greedy BFS
27         PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(node -> heuristic(node.word, endWord)));
28         queue.add(new Node(beginWord, 1, null));
29
30         // create a list to store the checked nodes
31         List<String> checkedNodes = new ArrayList<>();
32
33         while (!queue.isEmpty()){
34             Node node = queue.remove(); // removes the node with the lowest number of steps from the queue
35             String word = node.word; // stores the current word from the node
36
37             checkedNodes.add(word);
38
39             // checks if the current word is the endWord
40             if (word.equals(endWord)){
41                 List<String> ladder = new ArrayList<>();
42                 while (node != null){
43                     ladder.add(0, node.word);
44                     node = node.prev;
45                 }
46                 System.out.println("Number of Checked Nodes: " + checkedNodes.size());
47                 return ladder;
48             }
49
50             // converts the word to a character array to make checking easier
51             char[] arr = word.toCharArray();
52             for (int i = 0; i < arr.length; i++){ // loops over each character in the array
53                 for (char c = 'a'; c < 'z'; c++){
54                     char temp = arr[i]; // stores the original character for restoration later
55                     if (arr[i] != c){ // change the character if the character is different
56                         arr[i] = c;
57                     }
58
59                     String newWord = new String(arr); // creates a new string from the modified character
60
61                     // if the new word is in the dictionary, add a new node for the new word
62                     if (dict.contains(newWord)){
63                         queue.add(new Node(newWord, node.steps + 1, node));
64                         dict.remove(newWord); // remove the new word from the dictionary to avoid revisiting
65                     }
66
67                     arr[i] = temp; // restores the original character
68                 }
69             }
70         }
71         return Collections.emptyList(); // return empty list if no path found
72     }
73 }
74

```

Astar.java

```

1  import java.util.*;
2
3  public class Astar extends WordLadder {
4
5      // method for counting the heuristic feature (the sum of the different character)
6      private int heuristic(String word, String endWord){
7          int diff = 0;
8          for (int i = 0; i < word.length(); i++){
9              if (word.charAt(i) != endWord.charAt(i)){
10                 diff++;
11             }
12         }
13         return diff;
14     }
15
16     @Override
17     public List<String> findLadder(String beginWord, String endWord, List<String> wordList){
18
19         // if the end word is not available in the dictionary, then return empty list
20         if (!wordList.contains(endWord)){
21             return Collections.emptyList();
22         }
23
24         Set<String> dict = new HashSet<>(wordList); // convert wordList into a HashSet for faster look-up times
25
26         // initializes a priority queue that orders nodes by the sum of the current steps and the heuristic value
27         PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(node -> node.steps + heuristic(node.word, endWord)));
28         queue.add(new Node(beginWord, 1, null));
29
30         // create a list to store the checked nodes
31         List<String> checkedNodes = new ArrayList<>();
32
33         while (!queue.isEmpty()){
34             Node node = queue.remove(); // removes the node with the lowest number of steps from the queue
35             String word = node.word; // stores the current word from the node
36
37             checkedNodes.add(word);
38
39             // checks if the current word is the endWord
40             if (word.equals(endWord)){
41                 List<String> ladder = new ArrayList<>();
42                 while (node != null){
43                     ladder.add(0, node.word);
44                     node = node.prev;
45                 }
46                 System.out.println("Number of Checked Nodes: " + checkedNodes.size());
47                 return ladder;
48             }
49
50             // converts the word to a character array to make checking easier
51             char[] arr = word.toCharArray();
52             for (int i = 0; i < arr.length; i++){ // loops over each character in the array
53                 for (char c = 'a'; c < 'z'; c++){
54                     char temp = arr[i]; // stores the original character for restoration later
55                     if (arr[i] != c){ // change the character if the character is different
56                         arr[i] = c;
57                     }
58
59                     String newWord = new String(arr); // creates a new string from the modified character
60
61                     // if the new word is in the dictionary, add a new node for the new word
62                     if (dict.contains(newWord)){
63                         queue.add(new Node(newWord, node.steps + 1, node));
64                         dict.remove(newWord); // remove the new word from the dictionary to avoid revisiting
65                     }
66
67                     arr[i] = temp; // restores the original character
68                 }
69             }
70         }
71         return Collections.emptyList(); // return empty list if no path found
72     }
73 }
74

```

4.2. Pengetesan Kode

Input	Output
Start Word: Hit End Word: Log Algoritma: Uniform-Cost Search	<pre>Enter the start word: hit Enter the end word: log Choose the algorithm: 1. Uniform-Cost Search (UCS) 2. A* Algorithm 3. Greedy Best-First-Search Enter choosen algorithm: 1 Number of Checked Nodes: 2264 Path: hit -> lit -> lot -> log Execution time: 32 ms</pre>
Start Word: Make End Word: Pure Algoritma: Uniform-Cost Search	<pre>Enter the start word: make Enter the end word: pure Choose the algorithm: 1. Uniform-Cost Search (UCS) 2. A* Algorithm 3. Greedy Best-First-Search Enter choosen algorithm: 1 Number of Checked Nodes: 935 Path: make -> mare -> pare -> pure Execution time: 28 ms</pre>
Start Word: Pan End Word: And Algoritma: Uniform-Cost Search	<pre>Enter the start word: pan Enter the end word: and Choose the algorithm: 1. Uniform-Cost Search (UCS) 2. A* Algorithm 3. Greedy Best-First-Search Enter choosen algorithm: 1 Number of Checked Nodes: 4183 Path: pan -> prn -> arn -> ard -> and Execution time: 47 ms</pre>

<p>Start Word: Zeus End Word: Damn Algoritma: Uniform-Cost Search</p>	<pre> Enter the start word: zeus Enter the end word: damn Choose the algorithm: 1. Uniform-Cost Search (UCS) 2. A* Algorithm 3. Greedy Best-First-Search Enter choosen algorithm: 1 Number of Checked Nodes: 6385 Path: zeus -> deus -> debs -> dabs -> dams -> damn Execution time: 64 ms </pre>
<p>Start Word: Meats End Word: Lover Algoritma: Uniform-Cost Search</p>	<pre> Enter the start word: meats Enter the end word: lover Choose the algorithm: 1. Uniform-Cost Search (UCS) 2. A* Algorithm 3. Greedy Best-First-Search Enter choosen algorithm: 1 Number of Checked Nodes: 10194 Path: meats -> moats -> mosts -> moses -> moser -> loser -> lover Execution time: 98 ms </pre>
<p>Start Word: Ballon End Word: Change Algoritma: Uniform-Cost Search</p>	<pre> Enter the start word: ballon Enter the end word: change Choose the algorithm: 1. Uniform-Cost Search (UCS) 2. A* Algorithm 3. Greedy Best-First-Search Enter choosen algorithm: 1 Number of Checked Nodes: 20486 Path: ballon -> ballot -> ballet -> callet -> calles -> c arles -> carnes -> caines -> clines -> clings -> clangs - > changs -> change Execution time: 306 ms </pre>
<p>Start Word: Hit End Word: Log Algoritma: A*</p>	<pre> Enter the start word: hit Enter the end word: log Choose the algorithm: 1. Uniform-Cost Search (UCS) 2. A* Algorithm 3. Greedy Best-First-Search Enter choosen algorithm: 2 Number of Checked Nodes: 7 Path: hit -> lit -> lot -> log Execution time: 14 ms </pre>

<p>Start Word: Make End Word: Pure Algoritma: A*</p>	<pre> Enter the start word: make Enter the end word: pure Choose the algorithm: 1. Uniform-Cost Search (UCS) 2. A* Algorithm 3. Greedy Best-First-Search Enter choosen algorithm: 2 Number of Checked Nodes: 5 Path: make -> mare -> pare -> pure Execution time: 14 ms </pre>
<p>Start Word: Pan End Word: And Algoritma: A*</p>	<pre> Enter the start word: pan Enter the end word: and Choose the algorithm: 1. Uniform-Cost Search (UCS) 2. A* Algorithm 3. Greedy Best-First-Search Enter choosen algorithm: 2 Number of Checked Nodes: 81 Path: pan -> pad -> pbd -> abd -> and Execution time: 15 ms </pre>
<p>Start Word: Zeus End Word: Damn Algoritma: A*</p>	<pre> Enter the start word: zeus Enter the end word: damn Choose the algorithm: 1. Uniform-Cost Search (UCS) 2. A* Algorithm 3. Greedy Best-First-Search Enter choosen algorithm: 2 Number of Checked Nodes: 28 Path: zeus -> deus -> deut -> daut -> daun -> damn Execution time: 26 ms </pre>
<p>Start Word: Meats End Word: Lover Algoritma: A*</p>	<pre> Enter the start word: meats Enter the end word: lover Choose the algorithm: 1. Uniform-Cost Search (UCS) 2. A* Algorithm 3. Greedy Best-First-Search Enter choosen algorithm: 2 Number of Checked Nodes: 52 Path: meats -> moats -> molts -> moles -> moves -> loves -> lover Execution time: 35 ms </pre>

<p>Start Word: Ballon End Word: Change Algoritma: A*</p>	<pre> Enter the start word: ballon Enter the end word: change Choose the algorithm: 1. Uniform-Cost Search (UCS) 2. A* Algorithm 3. Greedy Best-First-Search Enter choosen algorithm: 2 Number of Checked Nodes: 4735 Path: ballon -> ballot -> callot -> carlot -> carnot -> c arnet -> carnes -> caines -> clines -> clings -> clangs - > changs -> change Execution time: 77 ms </pre>
<p>Start Word: Hit End Word: Log Algoritma: Greedy Best First Search</p>	<pre> Enter the start word: hit Enter the end word: log Choose the algorithm: 1. Uniform-Cost Search (UCS) 2. A* Algorithm 3. Greedy Best-First-Search Enter choosen algorithm: 3 Number of Checked Nodes: 4 Path: hit -> lit -> lot -> log Execution time: 13 ms </pre>
<p>Start Word: Make End Word: Pure Algoritma: Greedy Best First Search</p>	<pre> Enter the start word: make Enter the end word: pure Choose the algorithm: 1. Uniform-Cost Search (UCS) 2. A* Algorithm 3. Greedy Best-First-Search Enter choosen algorithm: 3 Number of Checked Nodes: 4 Path: make -> mare -> pare -> pure Execution time: 33 ms </pre>

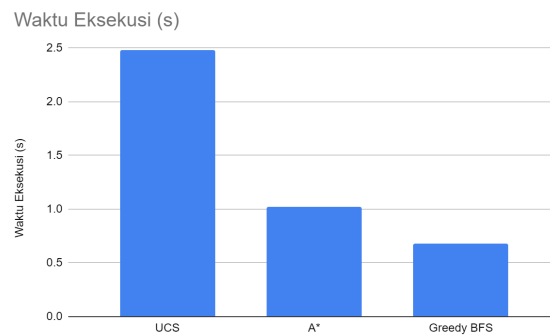
<p>Start Word: Pan End Word: And Algoritma: Greedy Best First Search</p>	<pre> Enter the start word: pan Enter the end word: and Choose the algorithm: 1. Uniform-Cost Search (UCS) 2. A* Algorithm 3. Greedy Best-First-Search Enter choosen algorithm: 3 Number of Checked Nodes: 9 Path: pan -> pad -> pbd -> abd -> and Execution time: 13 ms </pre>
<p>Start Word: Zeus End Word: Damn Algoritma: Greedy Best First Search</p>	<pre> Enter the start word: zeus Enter the end word: damn Choose the algorithm: 1. Uniform-Cost Search (UCS) 2. A* Algorithm 3. Greedy Best-First-Search Enter choosen algorithm: 3 Number of Checked Nodes: 17 Path: zeus -> deus -> deas -> dean -> denn -> dann -> dam n Execution time: 15 ms </pre>
<p>Start Word: Meats End Word: Lover Algoritma: Greedy Best First Search</p>	<pre> Enter the start word: meats Enter the end word: lover Choose the algorithm: 1. Uniform-Cost Search (UCS) 2. A* Algorithm 3. Greedy Best-First-Search Enter choosen algorithm: 3 Number of Checked Nodes: 19 Path: meats -> moats -> loats -> louts -> louty -> louey -> lovey -> lover Execution time: 32 ms </pre>
<p>Start Word: Ballon End Word: Change Algoritma: Greedy Best First Search</p>	<pre> Enter the start word: ballon Enter the end word: change Choose the algorithm: 1. Uniform-Cost Search (UCS) 2. A* Algorithm 3. Greedy Best-First-Search Enter choosen algorithm: 3 Number of Checked Nodes: 117 Path: ballon -> ballow -> callow -> carlow -> carlot -> c arnot -> cannot -> cannat -> cannae -> cannie -> lannie - > linnie -> winnie -> wienie -> weenie -> weanie -> jeani e -> jeanne -> deanne -> dianne -> dianna -> sianna -> sh anna -> channa -> changa -> change Execution time: 23 ms </pre>

BAB V

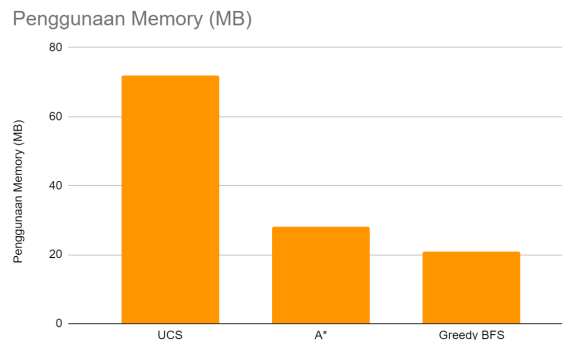
HASIL ANALISIS ALGORITMA

5.1. Analisis Perbandingan Algoritma UCS, A*, dan Greedy BFS dalam Pencarian Jalur dalam Game Word Ladder

Uniform-Cost Search (UCS), A*, dan Greedy Best-First Search (Greedy BFS) adalah tiga algoritma pencarian jalur yang memiliki karakteristik dan penggunaan yang berbeda. Masing-masing algoritma memiliki kelebihan dan kekurangan dalam hal optimalitas, waktu eksekusi, dan penggunaan memori. Berikut ini adalah analisis perbandingan ketiga algoritma tersebut.



Gambar 3 Chart Perbandingan Waktu Eksekusi



Gambar 4 Chart Perbandingan Penggunaan Memory

5.1.1. Optimalitas

1. Uniform-Cost Search (UCS)

UCS adalah algoritma pencarian yang menjamin solusi optimal karena selalu memilih node dengan biaya terendah untuk ekspansi. Dalam kasus Word Ladder, biaya adalah jumlah langkah yang diambil untuk mencapai suatu kata. UCS akan selalu menemukan jalur terpendek

karena mengutamakan node dengan jumlah langkah terendah. Namun, karena UCS mengeksplorasi semua kemungkinan jalur berdasarkan biaya, algoritma ini bisa menjadi lambat jika ada banyak jalur potensial dengan biaya serupa.

2. A*

Algoritma A* juga menjamin solusi optimal, dengan asumsi heuristik yang digunakan adalah admissible (tidak melebihi-lebihkan biaya aktual menuju tujuan). Dalam konteks Word Ladder, jika heuristik adalah jumlah karakter yang berbeda antara kata saat ini dan kata akhir, heuristik tersebut dianggap admissible, memastikan bahwa A* akan menemukan jalur optimal. Karena A* menggabungkan biaya langkah ($g(n)$) dan heuristik ($h(n)$), algoritma ini cenderung lebih efisien dibandingkan UCS karena dapat mengarahkan pencarian ke arah tujuan.

3. Greedy Best-First Search (Greedy BFS)

Greedy BFS tidak menjamin solusi optimal karena hanya mengandalkan heuristik untuk menentukan prioritas. Tanpa mempertimbangkan biaya langkah, algoritma ini mungkin memilih jalur yang tampaknya paling dekat dengan tujuan, tetapi bisa saja jalur tersebut bukanlah jalur terpendek. Greedy BFS berisiko menemukan solusi suboptimal karena tidak selalu mempertimbangkan total biaya menuju tujuan.

5.1.2. Waktu Eksekusi

1. Uniform-Cost Search (UCS)

Waktu eksekusi UCS bisa lambat karena algoritma ini mengeksplorasi semua jalur potensial berdasarkan biaya langkah. Dalam kasus Word Ladder, jika ada banyak jalur dengan biaya serupa, UCS bisa menjadi kurang efisien. Namun, karena UCS menjamin solusi optimal, algoritma ini terkadang lebih lambat dibandingkan algoritma lain yang mungkin memilih jalur dengan lebih sedikit eksplorasi.

2. A*

A* biasanya lebih cepat daripada UCS karena menggunakan heuristik untuk memperkirakan biaya menuju tujuan. Dengan informasi tambahan dari heuristik, A* cenderung mengarahkan pencarian ke jalur yang lebih menjanjikan, mengurangi eksplorasi yang tidak perlu. Waktu eksekusi A* dapat bervariasi tergantung pada kualitas heuristik, tetapi dalam konteks Word Ladder, A* biasanya lebih efisien daripada UCS.

3. Greedy Best-First Search (Greedy BFS)

Greedy BFS cenderung memiliki waktu eksekusi yang lebih cepat karena hanya mengandalkan heuristik untuk menentukan prioritas. Algoritma ini dapat menemukan jalur dengan cepat karena mengarahkan eksplorasi ke arah yang tampaknya paling dekat dengan tujuan. Namun, Greedy BFS bisa menghasilkan solusi suboptimal, yang berarti mungkin

memerlukan lebih banyak waktu untuk menemukan jalur yang sebenarnya lebih pendek jika jalur yang dipilih tidak optimal.

5.1.3. Penggunaan Memori

1. Uniform-Cost Search (UCS)

Penggunaan memori UCS bisa tinggi karena algoritma ini menyimpan semua node yang diekspansi dalam struktur data seperti priority queue. Karena UCS mengeksplorasi semua kemungkinan jalur berdasarkan biaya, penggunaan memori bisa meningkat jika ada banyak jalur potensial.

2. A*

A* cenderung lebih efisien dalam penggunaan memori dibandingkan UCS karena algoritma ini menggunakan heuristik untuk mengurangi eksplorasi yang tidak perlu. Dengan lebih sedikit node yang perlu diekspansi, A* dapat mengurangi penggunaan memori.

3. Greedy Best-First Search (Greedy BFS)

Greedy BFS biasanya memiliki penggunaan memori yang lebih rendah karena hanya menggunakan heuristik untuk menentukan prioritas. Karena algoritma ini lebih fokus pada eksplorasi jalur yang tampaknya paling dekat dengan tujuan, Greedy BFS dapat mengurangi jumlah node yang perlu disimpan dan diekspansi, sehingga mengurangi penggunaan memori.

BAB VI

KESIMPULAN DAN SARAN

6.1. Kesimpulan

Dalam analisis perbandingan antara algoritma Uniform-Cost Search (UCS), A*, dan Greedy Best-First Search (Greedy BFS) dalam konteks Word Ladder, ditemukan bahwa algoritma A* menunjukkan kinerja yang paling optimal dalam hal waktu eksekusi dan penggunaan memori, sambil tetap menjamin solusi optimal. Meskipun secara teoritis algoritma UCS juga menjamin solusi optimal, dalam praktiknya, A* mampu memberikan hasil lebih cepat berkat penggunaan heuristik untuk mengarahkan pencarian menuju tujuan. Greedy BFS, sementara itu, menawarkan waktu eksekusi yang lebih cepat dengan penggunaan memori yang lebih rendah, tetapi tanpa jaminan solusi optimal.

Hasil yang lebih baik yang ditunjukkan oleh A* kemungkinan disebabkan oleh kemampuan algoritma untuk menggabungkan biaya langkah dengan estimasi jarak menuju tujuan, memungkinkan pengurangan eksplorasi yang tidak perlu. UCS cenderung mengeksplorasi semua jalur potensial berdasarkan biaya langkah, yang dapat meningkatkan waktu eksekusi dan penggunaan memori. Greedy BFS, di sisi lain, fokus pada eksplorasi jalur yang tampaknya paling dekat dengan tujuan, yang dapat menghasilkan solusi yang cepat tetapi tidak selalu optimal.

Perbedaan dalam kinerja mungkin juga dipengaruhi oleh implementasi spesifik dari algoritma, ukuran dan kompleksitas dataset, serta faktor-faktor lingkungan seperti kapasitas memori dan daya komputasi. Kesimpulannya, algoritma A* umumnya menawarkan keseimbangan terbaik antara kecepatan dan optimalitas dalam konteks Word Ladder, sementara UCS dan Greedy BFS memiliki keunggulan dan kekurangan masing-masing tergantung pada kebutuhan optimalitas dan efisiensi.

6.2. Saran

Untuk pencarian jalur di Word Ladder, algoritma A* bisa dikombinasikan dengan pendekatan lain untuk hasil yang lebih optimal, seperti penggunaan heuristik yang lebih canggih atau kombinasi dengan algoritma lain yang dapat mempercepat pencarian. Misalnya, kombinasi A* dengan teknik pruning atau pengindeksan khusus dapat meningkatkan efisiensi eksplorasi. Selain itu, dapat dicoba mengintegrasikan metode cache atau memoization untuk mengurangi eksplorasi ulang, sehingga menghemat waktu dan memori. Dengan demikian, pendekatan ini bisa meningkatkan kinerja dan hasil pencarian jalur di berbagai kasus, termasuk dalam konteks Word Ladder.

BAB VII

DAFTAR PUSTAKA

7.1. Sumber Pustaka

- [1]R. Munir, “Penentuan Rute (Route/Path Planning).” Available: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf>
- [2]dwyl, “dwyl/english-words,” *GitHub*, Jun. 03, 2019. <https://github.com/dwyl/english-words>
- [3]“Greedy Best first search algorithm,” *GeeksforGeeks*, Dec. 15, 2022. <https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/>
- [4]R. Belwariar, “A* Search Algorithm - GeeksforGeeks,” *GeeksforGeeks*, Sep. 07, 2018. <https://www.geeksforgeeks.org/a-search-algorithm>
- [5]“Uniform-Cost Search (Dijkstra for large Graphs),” *GeeksforGeeks*, Mar. 25, 2019. <https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/>
- [6]“Word ladder,” *Wikipedia*, Mar. 19, 2024. https://en.wikipedia.org/wiki/Word_ladder (accessed May 04, 2024).

7.2. Lampiran

1. Github: https://github.com/rizqikapratamaa/Tucil3_13522126
- 2.

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		✓

