

LAPORAN TUGAS KECIL 03
IF2211 STRATEGI ALGORITMA

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



Disusun oleh:

Farrel Natha Saskoro K-03 13522145

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

DAFTAR ISI

| | |
|--|-----------|
| DAFTAR ISI | 2 |
| BAB 1 | 3 |
| BAB 2 | 4 |
| 2.1 Algoritma Uniform Cost Search | 4 |
| 2.2 Algoritma Greedy Best First Search | 5 |
| 2.3 Algoritma A | 6 |
| BAB 3 | 7 |
| 3.1 Definisi dari $f(n)$ dan $g(n)$ | 7 |
| 3.1.1 Uniform Cost Search (UCS) | 7 |
| 3.1.2 Greedy Best First Search (GBFS) | 7 |
| 3.1.3 A* | 7 |
| 3.2 Heuristik pada algoritma A* | 8 |
| 3.3 Algoritma UCS dan BFS (dalam kasus word ladder) | 8 |
| 3.4 Algoritma A* dan Algoritma UCS (dalam kasus word ladder) | 8 |
| 3.5 Apakah algoritma Greedy Best First Search menjamin solusi optimal untuk persoalan word ladder? | 9 |
| 3.6 Implementasi Website | 10 |
| 3.6 Source Code Program | 11 |
| 3.6.1 UCS.java | 11 |
| 3.6.2 GreedyBestFirst.java | 14 |
| 3.6.3 AStar.java | 17 |
| 3.6.4 Controller.java | 20 |
| 3.6.5 GlobalExceptionHandler.java | 21 |
| 3.6.6 Result.java | 22 |
| 3.6.7 Front End | 24 |
| BAB 4 | 26 |
| 4.1 Tampilan Website | 26 |
| 4.2 UCS | 26 |
| 4.3 Greedy Best First | 29 |
| 4.4 A* | 31 |
| 4.5 HASIL ANALISIS | 33 |
| BAB 5 | 34 |
| 5.1 Kesimpulan | 34 |
| DAFTAR PUSTAKA | 35 |
| LAMPIRAN | 36 |

BAB 1

DESKRIPSI MASALAH

Word ladder (juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragrams, laddergrams, atau word golf) adalah salah satu permainan kata yang terkenal bagi seluruh kalangan. Word ladder ditemukan oleh Lewis Carroll, seorang penulis dan matematikawan, pada tahun 1877. Pada permainan ini, pemain diberikan dua kata yang disebut sebagai start word dan end word. Untuk memenangkan permainan, pemain harus menemukan rantai kata yang dapat menghubungkan antara start word dan end word. Banyaknya huruf pada start word dan end word selalu sama. Tiap kata yang berdekatan dalam rantai kata tersebut hanya boleh berbeda satu huruf saja. Pada permainan ini, diharapkan solusi optimal, yaitu solusi yang meminimalkan banyaknya kata yang dimasukkan pada rantai kata. Berikut adalah ilustrasi serta aturan permainan. Gambar 1. Ilustrasi dan Peraturan Permainan Word Ladder

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

E A S T

EAST is the start word, **WEST** is the end word

V A S T

We changed **E** to **V** to make **VAST**

V E S T

We changed **A** to **E** to make **VEST**

W E S T

And we changed **V** to **W** to make **WEST**

W E S T

Done!

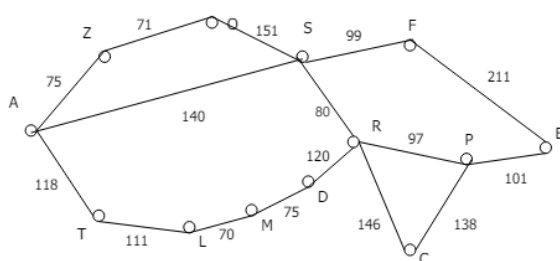
BAB 2

TEORI SINGKAT

2.1 Algoritma Uniform Cost Search

Uniform Cost Search (UCS)

- BFS & IDS find path with fewest steps (A-S-F-B)
- If steps \neq cost, this is not relevant (to optimal solution)
- How can we find the shortest path (measured by sum of distances along path)?
- $g(n)$ = path cost from root to n



Path: A → S → R → P → B
Path-cost = 418 → optimal solution

IF2211/NUM/29Mar2016

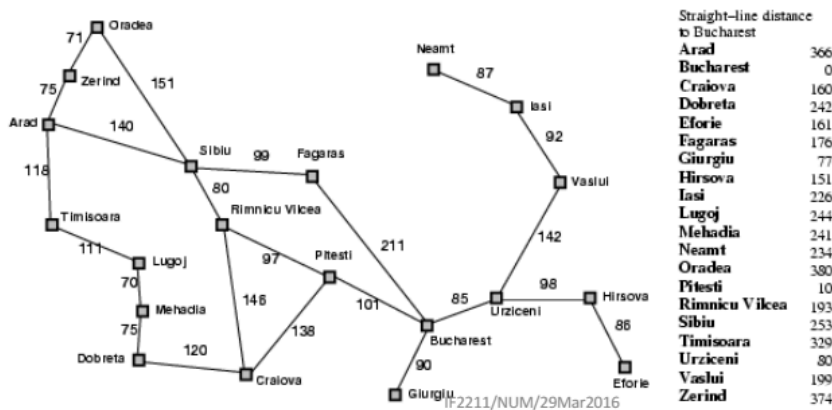
| Simpul-E | Simpul Hidup |
|-----------------------|--|
| A | Z _{A-75} , T _{A-118} , S _{A-140} |
| Z _{A-75} | T _{A-118} , S _{A-140} , O _{AZ-146} |
| T _{A-118} | S _{A-140} , O _{AZ-146} , L _{AT-229} |
| S _{A-140} | O _{AZ-146} , R _{AS-220} , L _{AT-229} , F _{AS-239} , O _{AS-291} |
| O _{AZ-146} | R _{AS-220} , L _{AT-229} , F _{AS-239} , O _{AS-291} |
| R _{AS-220} | L _{AT-229} , F _{AS-239} , O _{AS-291} , P _{ASR-317} , D _{ASR-340} , C _{ASR-366} |
| L _{AT-229} | F _{AS-239} , O _{AS-291} , M _{ATL-299} , P _{ASR-317} , D _{ASR-340} , C _{ASR-366} |
| F _{AS-239} | O _{AS-291} , M _{ATL-299} , P _{ASR-317} , D _{ASR-340} , C _{ASR-366} , B _{ASF-450} |
| O _{AS-291} | M _{ATL-299} , P _{ASR-317} , D _{ASR-340} , C _{ASR-366} , B _{ASF-450} |
| M _{ATL-299} | P _{ASR-317} , D _{ASR-340} , D _{ATLM-364} , C _{ASR-366} , B _{ASF-450} |
| P _{ASR-317} | D _{ASR-340} , D _{ATLM-364} , C _{ASR-366} , B _{ASRP-418} , C _{ASRP-455} , B _{ASF-450} |
| D _{ASR-340} | D _{ATLM-364} , C _{ASR-366} , B _{ASRP-418} , C _{ASRP-455} , B _{ASF-450} |
| D _{ATLM-364} | C _{ASR-366} , B _{ASRP-418} , C _{ASRP-455} , B _{ASF-450} |
| C _{ASR-366} | B _{ASRP-418} , C _{ASRP-455} , B _{ASF-450} |
| B _{ASF-450} | Solusi ketemu |

Algoritma UCS (Uniform Cost Search) adalah sebuah algoritma pencarian graf yang sangat berguna dalam menemukan jalur terpendek di dalam graf berbobot. Langkah pertama dalam algoritma ini adalah melakukan inisialisasi dengan memulai dari simpul awal dan menetapkan biaya awalnya sebagai 0, sementara biaya untuk semua simpul lainnya diatur sebagai tak terhingga. Kemudian, algoritma akan mulai mengekskansi simpul dengan biaya terendah terlebih dahulu. Setiap kali sebuah simpul diekspansi, biaya total untuk mencapai simpul tersebut dihitung sebagai jumlah biaya dari simpul awal ke simpul tersebut ditambah dengan biaya dari simpul tersebut ke simpul yang sedang diekspansi. Algoritma kemudian memeriksa apakah simpul yang diekspansi adalah simpul tujuan. Jika belum, biaya untuk setiap simpul terhubung diperbarui dengan membandingkan biaya baru dengan biaya sebelumnya. Jika biaya baru lebih rendah, biaya tersebut diganti dengan biaya baru. Langkah-langkah ini diulangi sampai simpul tujuan ditemukan atau tidak ada lagi simpul yang tersisa untuk diekspansi. Meskipun UCS memiliki kelebihan dalam menemukan jalur terpendek dalam graf berbobot non-negatif, namun kinerjanya dapat menjadi lambat jika graf memiliki banyak simpul atau jika biaya dalam graf tidak terbatas

2.2 Algoritma Greedy Best First Search

Greedy Best-First Search

- Idea: use an **evaluation function** $f(n)$ for each node
 - $f(n) = h(n) \rightarrow$ estimates of cost from n to goal
 - e.g., $h_{SLD}(n)$ = straight-line distance from n to Bucharest
- Greedy best-first search expands the node that **appears** to be closest to goal



Romania with step costs in km:

12

Algoritma Greedy Best-First Search adalah metode pencarian yang mengandalkan heuristik lokal untuk menentukan langkah selanjutnya. Pada setiap langkah, algoritma ini memilih simpul berdasarkan nilai heuristiknya yang dianggap paling menjanjikan menuju ke simpul tujuan, tanpa mempertimbangkan secara menyeluruh jalur keseluruhan. Mulai dari simpul awal, algoritma memilih simpul dengan nilai heuristik terbaik sebagai simpul berikutnya, cenderung memilih simpul yang paling dekat dengan tujuan. Selanjutnya, algoritma memeriksa apakah simpul yang diekspansi adalah simpul tujuan. Jika belum, nilai heuristik dari simpul-simpul terhubung diperbarui berdasarkan kriteria tertentu, yang sering menggambarkan estimasi jarak atau biaya yang tersisa menuju simpul tujuan. Langkah-langkah ini diulangi sampai simpul tujuan ditemukan atau tidak ada simpul lain yang tersisa untuk diekspansi. Keuntungan dari algoritma ini adalah kemampuannya untuk menemukan solusi dengan cepat, terutama dalam graf yang besar. Namun, kelemahannya adalah algoritma ini tidak menjamin solusi optimal karena hanya mempertimbangkan informasi lokal pada setiap langkahnya.

A* Search

➤ Idea: avoid expanding paths that are already expensive

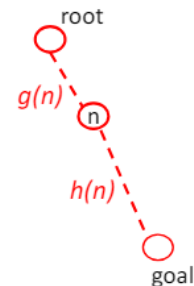
➤ Evaluation function $f(n) = g(n) + h(n)$

- $g(n)$ = cost so far to reach n
- $h(n)$ = estimated cost from n to goal
- $f(n)$ = estimated total cost of path through n to goal

➤ if $f(n) = g(n) \rightarrow$ Uniform Cost Search (UCS)

if $f(n) = h(n) \rightarrow$ Greedy Best First Search

if $f(n) = g(n) + h(n) \rightarrow A^*$



IF2211/NUM/RIN/19Mar2020

6

Algoritma A* (A-star) menggabungkan pendekatan Greedy Best-First dengan pencarian berbasis biaya untuk mencari jalur terpendek dari simpul awal ke simpul tujuan dalam sebuah graf. Dengan meminimalkan jumlah biaya total yang ditempuh dan memanfaatkan informasi heuristik, A* memilih simpul dengan nilai $f(n)$ terendah untuk diekspansi, di mana $f(n)$ merupakan jumlah dari biaya aktual ($g(n)$) dari simpul awal ke simpul saat ini, ditambah dengan nilai heuristik ($h(n)$) yang mengestimasi biaya dari simpul saat ini ke simpul tujuan. Langkah-langkah ini diulangi hingga simpul tujuan ditemukan atau tidak ada simpul lain yang tersisa untuk diekspansi. Algoritma A* mampu memberikan solusi optimal jika fungsi heuristik memenuhi persyaratan konsisten dan monotonik, namun kinerjanya terpengaruh oleh kualitas fungsi heuristik dan kompleksitas graf.

BAB 3

ANALISIS & IMPLEMENTASI PROGRAM

3.1 Definisi dari $f(n)$ dan $g(n)$

3.1.1 Uniform Cost Search (UCS)

- $f(n)$: Dalam algoritma UCS, tidak terdapat konsep $f(n)$ seperti pada algoritma A*. Algoritma UCS fokus pada pencarian jalur dengan biaya terendah dari simpul awal ke simpul tujuan tanpa mempertimbangkan nilai heuristik. Oleh karena itu, $f(n)$ tidak diterapkan dalam UCS.
- $g(n)$: Dalam UCS, $g(n)$ didefinisikan sebagai biaya aktual yang diperlukan untuk mencapai simpul saat ini dari simpul awal. Secara formal, $g(n)$ merupakan biaya jalur terpendek yang telah ditemukan dari simpul awal ke simpul saat ini. $g(n)$ digunakan untuk membandingkan biaya antara jalur-jalur yang berbeda dan menentukan urutan ekspansi simpul selanjutnya. Dengan kata lain, $g(n)$ menggambarkan biaya aktual atau sejauh mana kita telah mencapai simpul saat ini dari simpul awal.

3.1.2 Greedy Best First Search (GBFS)

- $f(n)$: Pada algoritma Greedy Best-First Search, $f(n)$ tidak secara eksplisit ditentukan. Algoritma ini hanya mempertimbangkan nilai heuristik $h(n)$, yang mengestimasi jarak atau biaya yang tersisa dari simpul saat ini ke simpul tujuan.
- $g(n)$: Tidak diterapkan dalam Greedy Best-First Search, karena algoritma ini hanya mempertimbangkan informasi heuristik lokal untuk memilih simpul berikutnya. Oleh karena itu, tidak ada konsep biaya aktual ($g(n)$) seperti pada algoritma lain.

3.1.3 A*

- $f(n)$: Dalam algoritma A*, $f(n)$ didefinisikan sebagai jumlah dari biaya aktual ($g(n)$) dari simpul awal ke simpul saat ini, ditambah dengan nilai heuristik ($h(n)$) yang mengestimasi biaya dari simpul saat ini ke simpul tujuan. Secara formal, $f(n) = g(n) + h(n)$.
- $g(n)$: Merupakan biaya aktual dari simpul awal ke simpul saat ini. Digunakan dalam perhitungan nilai $f(n)$ untuk menentukan urutan ekspansi simpul selanjutnya.

3.2 Heuristik pada algoritma A*

Heuristik pada algoritma A* adalah sebuah fungsi yang memberikan estimasi tentang biaya yang tersisa dari simpul saat ini ke simpul tujuan. Fungsi heuristik ini, sering dilambangkan sebagai $h(n)$, berperan penting dalam memberikan arahan kepada algoritma A* untuk memilih simpul berikutnya yang paling menjanjikan dalam pencarian jalur terpendek. Untuk memastikan efektivitas algoritma, fungsi heuristik harus memenuhi dua syarat utama: pertama, fungsi heuristik harus admissible, artinya estimasi biaya yang diberikan tidak boleh melebihi biaya sebenarnya dari simpul saat ini ke simpul tujuan, sehingga algoritma dapat menjamin menemukan solusi optimal. Kedua, fungsi heuristik juga harus monotonic, yang berarti estimasi biaya untuk mencapai simpul tujuan tidak boleh pernah berkurang saat bergerak ke simpul berikutnya. Dengan demikian, penggunaan informasi dari fungsi heuristik memungkinkan algoritma A* untuk mengarahkan pencarian dengan lebih efisien, memprioritaskan ekspansi simpul yang paling menjanjikan, dan dalam beberapa kasus, bahkan menemukan solusi optima

3.3 Algoritma UCS dan BFS (dalam kasus word ladder)

Algoritma UCS (Uniform Cost Search) dan BFS (Breadth-First Search) memiliki pendekatan yang berbeda dalam menemukan jalur terpendek dari titik awal ke titik tujuan. Meskipun keduanya memiliki tujuan yang serupa, yaitu menemukan jalur terpendek, cara kerja keduanya memiliki perbedaan mendasar. BFS melakukan eksplorasi pada semua simpul pada tingkat yang sama sebelum beralih ke tingkat berikutnya, sehingga dapat menemukan jalur terpendek pertama kali dan menghasilkan jalur optimal dalam hal jumlah langkah. Namun, BFS membutuhkan penyimpanan memori yang signifikan karena menyimpan semua simpul pada setiap tingkat. Di sisi lain, UCS mempertimbangkan biaya aktual dari titik awal ke titik saat ini dan memprioritaskan simpul dengan biaya yang lebih rendah terlebih dahulu. Dalam konteks word ladder, UCS akan menelusuri simpul dengan biaya yang semakin rendah tanpa memperhatikan tingkat atau jarak dari titik awal. Meskipun UCS dapat lebih efisien dalam penggunaan memori karena hanya menyimpan simpul yang sedang dieksplorasi dalam antrian prioritas, implementasi antrian prioritas yang tidak efisien atau tidak benar dapat meningkatkan penggunaan memori. Oleh karena itu, dalam memilih antara kedua algoritma ini, perlu mempertimbangkan trade-off antara penggunaan memori dan kinerja, tergantung pada struktur data yang digunakan dan ruang pencarian yang dihadapi.

3.4 Algoritma A* dan Algoritma UCS (dalam kasus word ladder)

Secara prinsip, perbedaan kunci antara algoritma A* dan UCS terletak pada penggunaan nilai heuristik. Di A*, nilai heuristik digunakan untuk memberikan estimasi tentang biaya atau jarak yang tersisa untuk mencapai tujuan dari setiap simpul, sementara UCS hanya memperhitungkan biaya aktual dari simpul awal ke simpul saat ini. Dengan memanfaatkan nilai heuristik, A* dapat membuat perkiraan yang lebih tepat tentang jalur terpendek menuju tujuan, karena mengkombinasikan biaya yang sudah ditempuh dengan estimasi biaya yang tersisa. Ini memungkinkan A* untuk menghindari pengeksplorasian jalur yang tidak produktif, mempercepat pencarian, dan mengurangi jumlah simpul yang harus dianalisis secara keseluruhan.

Dalam konteks word ladder, di mana tujuannya adalah menemukan jalur terpendek antara dua kata dengan mengubah satu huruf pada setiap langkah, A* akan memilih simpul dengan nilai $f(n)$ (total biaya dari simpul awal ke simpul saat ini ditambah nilai heuristik) yang lebih rendah. Ini menandakan bahwa simpul tersebut menunjukkan jalur yang lebih menjanjikan untuk mencapai tujuan dalam jumlah langkah yang lebih sedikit. Namun, efisiensi relatif A* dibandingkan UCS juga bergantung pada kualitas nilai heuristiknya. Jika nilai heuristik sangat akurat dan admissible (tidak melebihi biaya aktual), maka A* kemungkinan akan lebih efisien karena dapat melakukan analisis lebih lanjut dalam pencarian solusi terpendek.

Sementara itu, mengenai penggunaan memori, A* biasanya membutuhkan penyimpanan tambahan untuk nilai heuristik dari setiap simpul yang dieksplorasi. Ini dapat meningkatkan kebutuhan memori dibandingkan dengan UCS, yang hanya mempertimbangkan biaya aktual. Namun, dalam beberapa kasus, penggunaan memori tambahan ini dapat dianggap sebagai trade-off yang dapat diterima untuk meningkatkan kinerja pencarian secara keseluruhan.

3.5 Apakah algoritma Greedy Best First Search menjamin solusi optimal untuk persoalan word ladder?

Tentu, algoritma Greedy Best-First Search tidak menjamin solusi optimal untuk persoalan word ladder. Algoritma ini cenderung memilih simpul yang dianggap paling menjanjikan berdasarkan nilai heuristik lokal tanpa mempertimbangkan keseluruhan jalur. Dalam kasus word ladder, di mana tujuan adalah menemukan jalur terpendek antara dua kata dengan mengubah satu huruf pada setiap langkah, algoritma Greedy Best-First Search mungkin tidak selalu menghasilkan jalur yang optimal karena keputusan di setiap langkah hanya didasarkan pada informasi heuristik lokal. Oleh karena itu, meskipun Greedy Best-First Search bisa menjadi pilihan yang cepat, terutama dalam situasi di mana waktu eksekusi menjadi faktor kritis, namun tidak dapat menjamin solusi terpendek atau optimal dalam semua kasus.

Mengenai total memori yang digunakan, Greedy Best-First Search biasanya membutuhkan lebih sedikit memori dibandingkan dengan algoritma yang menyimpan seluruh jalur seperti Breadth-First Search atau Uniform Cost Search. Namun, penggunaan memori masih dapat bervariasi tergantung pada kompleksitas struktur data yang digunakan untuk menyimpan simpul yang dieksplorasi serta kompleksitas nilai heuristik yang digunakan. Dalam beberapa kasus, nilai heuristik yang buruk atau tidak tepat dapat menyebabkan algoritma ini melakukan banyak eksplorasi yang tidak produktif, yang pada gilirannya dapat memerlukan lebih banyak memori. Oleh karena itu, meskipun Greedy Best-First Search cenderung lebih hemat memori daripada beberapa algoritma pencarian lainnya, efisiensi totalnya masih tergantung pada sejumlah faktor, termasuk kompleksitas masalah yang dihadapi.

3.6 Implementasi Website

Frontend dari aplikasi ini dibuat menggunakan React dengan TypeScript dan Vite. Titik awal aplikasi terletak pada file `main.tsx`, yang bertugas merender komponen App ke dalam elemen dengan id 'root'. Komponen utama dari aplikasi ini adalah App, yang diimplementasikan dalam file `App.tsx`. Komponen ini menggunakan berbagai komponen antarmuka pengguna (UI) seperti Input, Button, dan Dialog. Skrip pembangunan untuk frontend ini diatur dalam `package.json`. Skrip ini pertama-tama menjalankan kompilator TypeScript (`tsc`) dan kemudian membangun aplikasi menggunakan Vite. Pengaturan linting untuk proyek ini menggunakan ESLint yang membantu memeriksa kode pada kesalahan umum dan gaya penulisan yang tidak konsisten. Selain itu, pengaturan untuk komponen UI ditentukan dalam file `components.json`, termasuk pengaturan untuk Tailwind CSS dan beberapa alias path. Dependensi untuk aplikasi ini, termasuk React dan pustaka lainnya, ditentukan dalam `package.json`.

Backend dari aplikasi ini menggunakan Spring Boot, sebuah framework Java yang memfasilitasi pembuatan aplikasi web. Anotasi `@SpringBootApplication` dalam file `BackendApplication.java` menandakan penggunaan Spring Boot, dengan metode `main` bertanggung jawab untuk memulai aplikasi melalui pemanggilan `SpringApplication.run(BackendApplication.class, args);`. File `BackendApplicationTests.java` digunakan untuk menulis tes unit dengan JUnit. Konfigurasi aplikasi ini, seperti nama aplikasi Spring Boot, ditentukan dalam file `application.properties`. Proyek backend juga menggunakan Maven sebagai sistem manajemen proyek, yang membantu dalam manajemen dependensi, build, dan pengujian proyek ini melalui file `pom.xml` dan `mvnw`.

3.6 Source Code Program

3.6.1 UCS.java

```
1 package fnathas.stima.tucil3.UCS;
2
3 import fnathas.stima.tucil3.Result.Result;
4 import org.springframework.stereotype.Service;
5
6 import java.io.IOException;
7 import java.nio.file.Files;
8 import java.nio.file.Paths;
9 import java.util.*;
10 import java.util.stream.Collectors;
11
12 class Node {
13     String word;
14     int cost;
15     Node parent;
16
17     Node(String word, int cost, Node parent) {
18         this.word = word;
19         this.cost = cost;
20         this.parent = parent;
21     }
22 }
23
24 @Service
25 public class UCS {
26     public Result ucs(String start, String goal) {
27
28         Set<String> dictionary = new HashSet<>();
29         try {
30             dictionary = Files.lines(Paths.get("C:/Informatika/sem4/stima/Tucil3_13522145/src/backend/dictionary.txt")).collect(Collectors.toSet());
31         } catch (IOException e) {
32             e.printStackTrace();
33         }
34
35         if (!dictionary.contains(start) || !dictionary.contains(goal)) {
36             throw new IllegalArgumentException("Start or end word is not in the dictionary");
37         }
38
39         if (start.length() != goal.length()) {
40             throw new IllegalArgumentException("Start and end word must have the same length");
41         }
42
43         PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(node -> node.cost));
44         Set<String> visited = new HashSet<>();
45         queue.add(new Node(start, 0, null));
46
47         while (!queue.isEmpty()) {
48             Node current = queue.poll();
49             visited.add(current.word);
50
51             if (current.word.equals(goal)) {
52                 return new Result(nodeToList(current), visited.size());
53             }
54
55             for (int i = 0; i < current.word.length(); i++) {
56                 char[] chars = current.word.toCharArray();
57                 for (char c = 'a'; c <= 'z'; c++) {
58                     chars[i] = c;
59                     String newWord = new String(chars);
60                     if (dictionary.contains(newWord) && !visited.contains(newWord)) {
61                         queue.add(new Node(newWord, current.cost + 1, current));
62                     }
63                 }
64             }
65         }
66
67         return new Result(null, visited.size());
68     }
69
70     private List<String> nodeToList(Node node) {
71         List<String> list = new ArrayList<>();
72         while (node != null) {
73             list.add(0, node.word);
74             node = node.parent;
75         }
76         return list;
77     }
78 }
```

- **Kelas UCS:**

Kelas ini digunakan untuk melakukan pencarian dengan algoritma Uniform Cost Search (UCS). Kelas ini memiliki dua metode `ucs` dan `nodeToList`.

- **Kelas Node:**

Kelas ini digunakan untuk merepresentasikan sebuah node dalam pencarian. Setiap node memiliki tiga atribut:

- `word`: Sebuah String yang merepresentasikan kata yang diwakili oleh node ini.
- `cost`: Sebuah int yang merepresentasikan biaya untuk mencapai node ini dari node awal.
- `parent`: Sebuah Node yang merepresentasikan node sebelumnya dalam jalur pencarian.

Konstruktor `Node(String word, int cost, Node parent)` digunakan untuk membuat instance baru dari kelas `Node`.

- **Metode `ucs`:**

Metode ini melakukan pencarian UCS dari kata awal (`start`) ke kata tujuan (`goal`). Metode ini mengembalikan objek `Result` yang berisi jalur dari kata awal ke kata tujuan dan jumlah kata yang dikunjungi. Dalam metode ini, kamus kata dibaca dari file teks dan disimpan dalam sebuah set bernama `dictionary`. Selanjutnya, dilakukan pengecekan keberadaan kata awal (`start`) dan kata akhir (`goal`) dalam kamus. Jika salah satu kata tidak ditemukan, sebuah `IllegalArgumentException` dilemparkan. Kemudian, dilakukan pengecekan terhadap panjang kata awal dan akhir untuk memastikan kesamaannya; jika tidak sama, sebuah `IllegalArgumentException` lagi dilemparkan. Selanjutnya, sebuah antrian prioritas `queue` dibuat untuk menyimpan node, di mana node dengan biaya terendah memiliki prioritas tertinggi. Set `visited` dibuat untuk menyimpan kata-kata yang telah dikunjungi. Proses dimulai dengan menambahkan node awal ke dalam `queue`. Selama `queue` tidak kosong, node dengan biaya terendah diambil dari `queue`, ditambahkan ke `visited`, dan dilakukan pengecekan apakah kata dalam node saat ini sama dengan kata akhir. Jika iya, jalur dari node awal ke node saat ini dan jumlah kata yang dikunjungi dikembalikan. Jika tidak, dilakukan eksplorasi untuk menghasilkan semua kata yang mungkin dengan mengubah satu huruf pada suatu waktu, dan kata-kata baru yang memenuhi syarat ditambahkan ke `queue` jika mereka ada dalam kamus dan belum dikunjungi. Jika `queue` kosong dan kata akhir belum ditemukan, `null` dikembalikan untuk jalur dan jumlah kata yang dikunjungi. Dengan demikian, algoritma UCS diimplementasikan secara sistematis untuk menemukan jalur terpendek dari kata awal ke kata akhir dengan mempertimbangkan biaya aktual dari setiap langkah.

- **Metode nodeToList:**

Metode ini mengubah jalur dari node tujuan ke node awal menjadi sebuah List<String>. Metode ini digunakan oleh metode ucs untuk menghasilkan jalur solusi.

3.6.2 GreedyBestFirst.java

```
1 package fnathas.stima.tucil3.GreedyBestFirst;
2
3 import fnathas.stima.tucil3.Result.Result;
4 import org.springframework.stereotype.Service;
5
6 import java.io.IOException;
7 import java.nio.file.Files;
8 import java.nio.file.Paths;
9 import java.util.*;
10 import java.util.stream.Collectors;
11
12 class Node {
13     String word;
14     int cost;
15     Node parent;
16
17     Node(String word, int cost, Node parent) {
18         this.word = word;
19         this.cost = cost;
20         this.parent = parent;
21     }
22 }
23
24 @Service
25 public class GreedyBestFirst {
26     public Result greedyBestFirst(String start, String goal) {
27
28         Set<String> dictionary = new HashSet<>();
29         try {
30             dictionary = Files.lines(Paths.get("C:/Informatika/sem4/stima/Tucil3_13522145/src/backend/dictionary.txt")).collect(Collectors.toSet());
31         } catch (IOException e) {
32             e.printStackTrace();
33         }
34
35         if (!dictionary.contains(start) || !dictionary.contains(goal)) {
36             throw new IllegalArgumentException("Start or end word is not in the dictionary");
37         }
38
39         if (start.length() != goal.length()) {
40             throw new IllegalArgumentException("Start and end word must have the same length");
41         }
42
43         PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(node -> node.cost));
44         Set<String> visited = new HashSet<>();
45         queue.add(new Node(start, heuristic(start, goal), null));
46
47         while (!queue.isEmpty()) {
48             Node current = queue.poll();
49             visited.add(current.word);
50
51             if (current.word.equals(goal)) {
52                 return new Result(nodeToList(current), visited.size());
53             }
54
55             for (int i = 0; i < current.word.length(); i++) {
56                 char[] chars = current.word.toCharArray();
57                 for (char c = 'a'; c <= 'z'; c++) {
58                     chars[i] = c;
59                     String newWord = new String(chars);
60                     if (dictionary.contains(newWord) && !visited.contains(newWord)) {
61                         queue.add(new Node(newWord, heuristic(newWord, goal), current));
62                     }
63                 }
64             }
65         }
66
67         return new Result(null, visited.size());
68     }
69
70     private List<String> nodeToList(Node node) {
71         List<String> list = new ArrayList<>();
72         while (node != null) {
73             list.add(0, node.word);
74             node = node.parent;
75         }
76         return list;
77     }
78
79     private int heuristic(String word, String goal) {
80         int cost = 0;
81         for (int i = 0; i < word.length(); i++) {
82             if (word.charAt(i) != goal.charAt(i)) {
83                 cost++;
84             }
85         }
86         return cost;
87     }
88 }
89
90
```

- **Kelas GreedyBestFirst:**

Kelas ini digunakan untuk melakukan pencarian dengan algoritma Greedy Best-First Search. Kelas ini memiliki tiga metode `greedyBestFirst(String start, String goal)`, `nodeToList(Node node)`, dan `heuristic(String word, String goal)`.

- **Kelas Node:**

Kelas ini digunakan untuk merepresentasikan sebuah node dalam pencarian. Setiap node memiliki tiga atribut:

- `word`: Sebuah String yang merepresentasikan kata yang diwakili oleh node ini.
- `cost`: Sebuah int yang merepresentasikan biaya untuk mencapai node ini dari node awal.
- `parent`: Sebuah Node yang merepresentasikan node sebelumnya dalam jalur pencarian.

Konstruktor `Node(String word, int cost, Node parent)` digunakan untuk membuat instance baru dari kelas `Node`.

- **Metode greedyBestFirst:**

Metode ini melakukan pencarian Greedy Best-First dari kata awal (`start`) ke kata tujuan (`goal`). Metode ini mengembalikan objek `Result` yang berisi jalur dari kata awal ke kata tujuan dan jumlah kata yang dikunjungi. Algoritma Greedy Best-First Search diimplementasikan dalam metode ini. Pertama, kamus kata dibaca dari file teks dan disimpan dalam sebuah set bernama `dictionary`. Selanjutnya, dilakukan pengecekan keberadaan kata awal (`start`) dan kata akhir (`goal`) dalam kamus, serta pengecekan panjang kata awal dan akhir untuk memastikan kesamaannya. Kemudian, sebuah antrian prioritas `queue` dibuat untuk menyimpan node, di mana node dengan biaya heuristik terendah memiliki prioritas tertinggi. Set `visited` digunakan untuk menyimpan kata-kata yang telah dikunjungi. Proses dimulai dengan menambahkan node awal ke dalam `queue` dengan biaya heuristik dari kata awal ke kata akhir. Selama `queue` tidak kosong, node dengan biaya heuristik terendah diambil dari `queue`, ditambahkan ke `visited`, dan dilakukan pengecekan apakah kata dalam node saat ini sama dengan kata akhir. Jika iya, jalur dari node awal ke node saat ini dan jumlah kata yang dikunjungi dikembalikan. Jika tidak, dilakukan eksplorasi untuk menghasilkan semua kata yang mungkin dengan mengubah satu huruf pada suatu waktu. Kata-kata baru yang memenuhi syarat ditambahkan ke `queue` dengan biaya heuristik dari kata baru ke kata akhir. Jika `queue` kosong dan kata akhir belum ditemukan, `null` dikembalikan untuk jalur dan jumlah kata yang dikunjungi. Fungsi `nodeToList` digunakan untuk mengubah jalur dari node awal ke node saat ini menjadi daftar kata, sedangkan fungsi `heuristic` digunakan untuk menghitung biaya heuristik dari kata saat ini ke kata akhir, yaitu jumlah huruf yang berbeda antara dua kata. Dengan demikian, algoritma GBFS diimplementasikan secara sistematis untuk mencari jalur terpendek dari kata awal ke kata akhir dengan mempertimbangkan biaya heuristik dari setiap langkah.

- **Metode nodeToList:**

Metode ini mengubah jalur dari node tujuan ke node awal menjadi sebuah List<String>. Metode ini digunakan oleh metode greedyBestFirst untuk menghasilkan jalur solusi.

- **Metode heuristic:**

Metode ini menghitung dan mengembalikan biaya heuristik dari kata word ke kata goal. Biaya heuristik dihitung sebagai jumlah karakter yang berbeda antara kata word dan kata goal.

3.6.3 AStar.java

```
1 package fnathas.stima.tucil3.AStar;
2
3 import fnathas.stima.tucil3.Result.Result;
4 import org.springframework.stereotype.Service;
5
6 import java.io.IOException;
7 import java.nio.file.Files;
8 import java.nio.file.Paths;
9 import java.util.*;
10 import java.util.stream.Collectors;
11
12
13 class Node {
14     String word;
15     int cost;
16     Node parent;
17
18     Node(String word, int cost, Node parent) {
19         this.word = word;
20         this.cost = cost;
21         this.parent = parent;
22     }
23 }
24
25 @Service
26 public class AStar {
27     public Result aStar(String start, String goal) {
28
29         Set<String> dictionary = new HashSet<>();
30         try {
31             dictionary = Files.lines(Paths.get("C:/Informatika/sem4/stima/Tucil3_13522145/src/backend/dictionary.txt")).collect(Collectors.toSet());
32         } catch (IOException e) {
33             e.printStackTrace();
34         }
35
36         if (!dictionary.contains(start) || !dictionary.contains(goal)) {
37             throw new IllegalArgumentException("Start or end word is not in the dictionary");
38         }
39
40         if (start.length() != goal.length()) {
41             throw new IllegalArgumentException("Start and end word must have the same length");
42         }
43
44         PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(node -> node.cost));
45         Set<String> visited = new HashSet<>();
46         queue.add(new Node(start, heuristic(start, goal), null));
47
48         while (!queue.isEmpty()) {
49             Node current = queue.poll();
50             visited.add(current.word);
51
52             if (current.word.equals(goal)) {
53                 return new Result(nodeToList(current), visited.size());
54             }
55
56             for (int i = 0; i < current.word.length(); i++) {
57                 char[] chars = current.word.toCharArray();
58                 for (char c = 'a'; c <= 'z'; c++) {
59                     chars[i] = c;
60                     String newWord = new String(chars);
61                     if (dictionary.contains(newWord) && !visited.contains(newWord)) {
62                         queue.add(new Node(newWord, current.cost + 1 + heuristic(newWord, goal), current));
63                     }
64                 }
65             }
66         }
67
68         return new Result(null, visited.size());
69     }
70
71     private List<String> nodeToList(Node node) {
72         List<String> list = new ArrayList<>();
73         while (node != null) {
74             list.add(0, node.word);
75             node = node.parent;
76         }
77         return list;
78     }
79
80     private int heuristic(String word, String goal) {
81         int cost = 0;
82         for (int i = 0; i < word.length(); i++) {
83             if (word.charAt(i) != goal.charAt(i)) {
84                 cost++;
85             }
86         }
87         return cost;
88     }
89 }
90
91
92
```

- **Kelas AStar:**

Kelas ini digunakan untuk melakukan pencarian dengan algoritma A*. Kelas ini memiliki tiga metode `aStar(String start, String goal)`, `nodeToList(Node node)`, dan `heuristic(String word, String goal)`.

- **Kelas Node:**

Kelas ini digunakan untuk merepresentasikan sebuah node dalam pencarian. Setiap node memiliki tiga atribut:

- `word`: Sebuah String yang merepresentasikan kata yang diwakili oleh node ini.
- `cost`: Sebuah int yang merepresentasikan biaya untuk mencapai node ini dari node awal.
- `parent`: Sebuah Node yang merepresentasikan node sebelumnya dalam jalur pencarian.

Konstruktor `Node(String word, int cost, Node parent)` digunakan untuk membuat instance baru dari kelas Node.

- **Metode aStar:**

Metode ini melakukan pencarian A* dari kata awal (start) ke kata tujuan (goal). Metode ini mengembalikan objek `Result` yang berisi jalur dari kata awal ke kata tujuan dan jumlah kata yang dikunjungi. Algoritma A* diimplementasikan dalam metode ini. Pertama, kamus kata dibaca dari file teks dan disimpan dalam sebuah set bernama `dictionary`. Selanjutnya, dilakukan pengecekan keberadaan kata awal (start) dan kata akhir (goal) dalam kamus, serta pengecekan panjang kata awal dan akhir untuk memastikan kesamaannya. Kemudian, sebuah antrian prioritas `queue` dibuat untuk menyimpan node, di mana node dengan biaya total terendah (biaya aktual ditambah biaya heuristik) memiliki prioritas tertinggi. Set `visited` digunakan untuk menyimpan kata-kata yang telah dikunjungi. Proses dimulai dengan menambahkan node awal ke dalam `queue` dengan biaya heuristik dari kata awal ke kata akhir. Selama `queue` tidak kosong, node dengan biaya total terendah diambil dari `queue`, ditambahkan ke `visited`, dan dilakukan pengecekan apakah kata dalam node saat ini sama dengan kata akhir. Jika iya, jalur dari node awal ke node saat ini dan jumlah kata yang dikunjungi dikembalikan. Jika tidak, dilakukan eksplorasi untuk menghasilkan semua kata yang mungkin dengan mengubah satu huruf pada suatu waktu. Kata-kata baru yang memenuhi syarat ditambahkan ke `queue` dengan biaya total dari kata baru ke kata akhir (biaya aktual ditambah biaya heuristik). Jika `queue` kosong dan kata akhir belum ditemukan, `null` dikembalikan untuk jalur dan jumlah kata yang dikunjungi. Fungsi `nodeToList` digunakan untuk mengubah jalur dari node awal ke node saat ini menjadi daftar kata, sedangkan fungsi `heuristic` digunakan untuk menghitung biaya heuristik dari kata saat ini ke kata akhir, yaitu jumlah huruf yang berbeda antara dua kata. Dengan demikian, algoritma A* diimplementasikan secara sistematis untuk mencari jalur terpendek dari kata awal ke kata akhir dengan mempertimbangkan biaya total dari setiap langkah.

- **Metode nodeToList:**

Metode ini mengubah jalur dari node tujuan ke node awal menjadi sebuah List<String>. Metode ini digunakan oleh metode aStar untuk menghasilkan jalur solusi.

- **Metode heuristic**

Metode ini menghitung dan mengembalikan biaya heuristik dari kata word ke kata goal. Biaya heuristik dihitung sebagai jumlah karakter yang berbeda antara kata word dan kata goal.

3.6.4 Controller.java

```
1 package fnathas.stima.tucil3.Controller;
2
3 import fnathas.stima.tucil3.UCS.UCS;
4 import fnathas.stima.tucil3.AStar.AStar;
5 import fnathas.stima.tucil3.GreedyBestFirst.GreedyBestFirst;
6 import fnathas.stima.tucil3.Result.Result;
7 import org.springframework.web.bind.annotation.CrossOrigin;
8 import org.springframework.web.bind.annotation.GetMapping;
9 import org.springframework.web.bind.annotation.RequestParam;
10 import org.springframework.web.bind.annotation.RestController;
11
12 import java.util.HashMap;
13 import java.util.Map;
14
15 @RestController
16 @CrossOrigin(origins = "*")
17 public class Controller {
18
19     private final UCS ucs;
20     private final AStar aStar;
21     private final GreedyBestFirst greedyBestFirst;
22
23     public Controller(UCS ucs, AStar aStar, GreedyBestFirst greedyBestFirst) {
24         this.ucs = ucs;
25         this.aStar = aStar;
26         this.greedyBestFirst = greedyBestFirst;
27     }
28
29     @GetMapping(value = "/run", produces = "application/json")
30     public Map<String, Object> runAlgorithm(@RequestParam String start, @RequestParam String goal, @RequestParam String algorithm) {
31         long startTime = System.currentTimeMillis();
32         long usedMemoryBefore = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
33         Object result = switch (algorithm.toLowerCase()) {
34             case "ucs" -> ucs.ucs(start, goal);
35             case "astar" -> aStar.aStar(start, goal);
36             case "gbfs" -> greedyBestFirst.greedyBestFirst(start, goal);
37             default -> throw new IllegalArgumentException("Invalid algorithm: " + algorithm);
38         };
39         long usedMemoryAfter = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
40         long executionTime = System.currentTimeMillis() - startTime;
41         long memoryUsed = (usedMemoryAfter - usedMemoryBefore) / 1024;
42
43         Map<String, Object> map = new HashMap<>();
44         if (result instanceof Result) {
45             Result res = (Result) result;
46             map.put("path", res.getPath());
47             map.put("memoryUsed", memoryUsed);
48         } else {
49             map.put("path", null);
50         }
51         map.put("executionTime", executionTime);
52         map.put("totalNodesVisited", ((Result) result).getTotalNodesVisited());
53         return map;
54     }
55 }
```

- **Kelas Controller:**

Kelas ini digunakan sebagai controller dalam aplikasi Spring Boot. Kelas ini memiliki tiga atribut:

- ucs: Sebuah objek dari kelas UCS yang digunakan untuk melakukan pencarian dengan algoritma Uniform Cost Search.

- aStar: Sebuah objek dari kelas AStar yang digunakan untuk melakukan pencarian dengan algoritma A*.
- greedyBestFirst: Sebuah objek dari kelas GreedyBestFirst yang digunakan untuk melakukan pencarian dengan algoritma Greedy Best-First Search.

Konstruktor `Controller(UCS ucs, AStar aStar, GreedyBestFirst greedyBestFirst)` digunakan untuk membuat instance baru dari kelas `Controller`.

- **Metode `runAlgorithm`**

Metode ini digunakan untuk menjalankan algoritma pencarian berdasarkan parameter `algorithm`. Parameter `start` dan `goal` digunakan sebagai kata awal dan kata tujuan dalam pencarian. Metode ini mengembalikan `Map<String, Object>` yang berisi jalur solusi, waktu eksekusi, jumlah node yang dikunjungi, dan total memori yang dipakai. Jika algoritma yang diberikan tidak valid, metode ini akan melempar `IllegalArgumentException`.

3.6.5 `GlobalExceptionHandler.java`

```

1  package fnathas.stima.tucil3.Exception;
2
3  import org.springframework.http.HttpStatus;
4  import org.springframework.http.ResponseEntity;
5  import org.springframework.web.bind.annotation.ControllerAdvice;
6  import org.springframework.web.bind.annotation.ExceptionHandler;
7
8  @ControllerAdvice
9  public class GlobalExceptionHandler {
10
11      @ExceptionHandler(IllegalArgumentException.class)
12      public ResponseEntity<String> handleIllegalArgumentException(IllegalArgumentException e) {
13          return new ResponseEntity<>(e.getMessage(), HttpStatus.BAD_REQUEST);
14      }
15  }
16

```

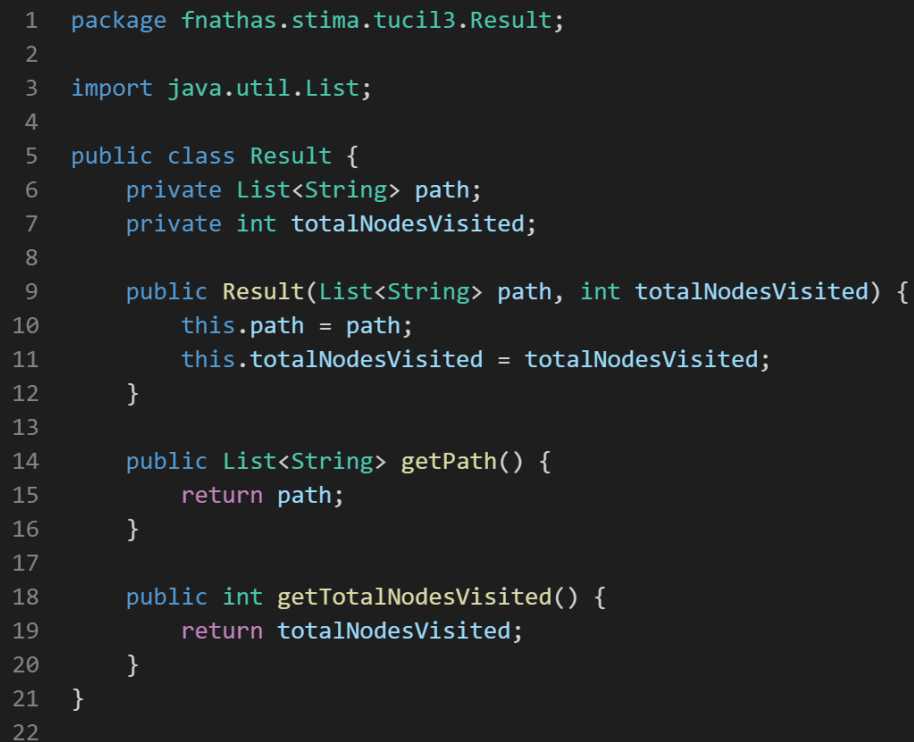
- **Kelas `GlobalExceptionHandler`**

Kelas ini berfungsi sebagai penanganan eksepsi global dalam aplikasi Spring Boot. Kelas ini didekorasi dengan anotasi `@ControllerAdvice` yang membuatnya menjadi komponen khusus yang digunakan untuk mendefinisikan `@ExceptionHandler`, `@InitBinder`, dan `@ModelAttribute` metode yang berlaku untuk semua `@RequestMapping` metode.

- **Metode `handleIllegalArgumentException`**

Metode ini ditandai dengan `@ExceptionHandler(IllegalArgumentException.class)`, yang berarti metode ini akan dipanggil saat ada `IllegalArgumentException` yang tidak ditangani oleh metode lain dalam aplikasi. Metode ini mengembalikan `ResponseEntity<String>` yang berisi pesan dari eksepsi dan status HTTP `BAD_REQUEST`.

3.6.6 Result.java



```
1 package fnathas.stima.tucil3.Result;
2
3 import java.util.List;
4
5 public class Result {
6     private List<String> path;
7     private int totalNodesVisited;
8
9     public Result(List<String> path, int totalNodesVisited) {
10         this.path = path;
11         this.totalNodesVisited = totalNodesVisited;
12     }
13
14     public List<String> getPath() {
15         return path;
16     }
17
18     public int getTotalNodesVisited() {
19         return totalNodesVisited;
20     }
21 }
22
```

- **Kelas Result**

kelas yang mewakili hasil dari algoritma pencarian. Kelas ini memiliki dua bidang: `path` yang merupakan daftar string dan `totalNodesVisited` yang merupakan bilangan bulat. `Result(List<String> path, int totalNodesVisited)` adalah konstruktor dari kelas `Result`. Konstruktor ini menerima dua parameter: `path` yang merupakan daftar string dan

totalNodesVisited yang merupakan bilangan bulat. Parameter ini disimpan dalam bidang yang sesuai.

- **Metode getPath**

Metode getter untuk bidang path. Metode ini mengembalikan daftar string yang mewakili jalur yang ditemukan oleh algoritma.

- **Metode getTotalNodesVisite**

Metode getter untuk bidang totalNodesVisited. Metode ini mengembalikan bilangan bulat yang mewakili total jumlah node yang dikunjungi oleh algoritma.

3.6.7 Front End

```
1 import { useState } from 'react'
2 import { Input } from '@components/ui/input'
3 import { Button } from '@components/ui/button'
4 import { useEffect } from 'react'
5 import {
6   Dialog,
7   DialogContent,
8   DialogDescription,
9   DialogHeader,
10  DialogTitle,
11  DialogTrigger
12 } from '@components/ui/dialog'
13
14 import './App.css'
15
16 type data = {
17   executionTime?: number
18   path: string[]
19   totalNodesVisited?: number
20 }
21
22 function App() {
23   const [startWord, setStartWord] = useState('')
24   const [endWord, setEndWord] = useState('')
25   const [algorithm, setAlgorithm] = useState('')
26   const [data, setData] = useState<data | null>(null)
27   const [isLoading, setIsLoading] = useState(false)
28
29   useEffect(() => {
30     let intervalId: NodeJS.Timeout | null = null
31     if (isLoading) {
32       // Call the API here
33       fetch('http://localhost:8080/run?start=${startWord}&goal=${endWord}&algorithm=${algorithm}')
34         .then(response => {
35           if (!response.ok) {
36             if (response.status === 400) {
37               // If the status is 400, parse the response body to get the error message
38               return response.text().then(errorMessage => {
39                 throw new Error(errorMessage);
40               });
41             } else {
42               throw new Error('HTTP error! status: ${response.status}');
43             }
44           }
45           return response.json();
46         })
47         .then(data => {
48           console.log(data)
49           setData(data)
50         })
51         .catch(error => {
52           console.error('There was an error!', error)
53           if (error.message) {
54             alert(error.message)
55           }
56         })
57         .finally(() => {
58           setIsLoading(false)
59           if (intervalId) {
60             clearInterval(intervalId)
61           }
62         })
63     }
64     return () => {
65       if (intervalId) {
66         clearInterval(intervalId)
67       }
68     }
69   }, [isLoading])
70
71   const handleSubmit = () => {
72     if (startWord === '' || endWord === '' || algorithm === '') {
73       return
74     }
75     setData(null)
76     setIsLoading(true)
77   }
78 }
79
```



```

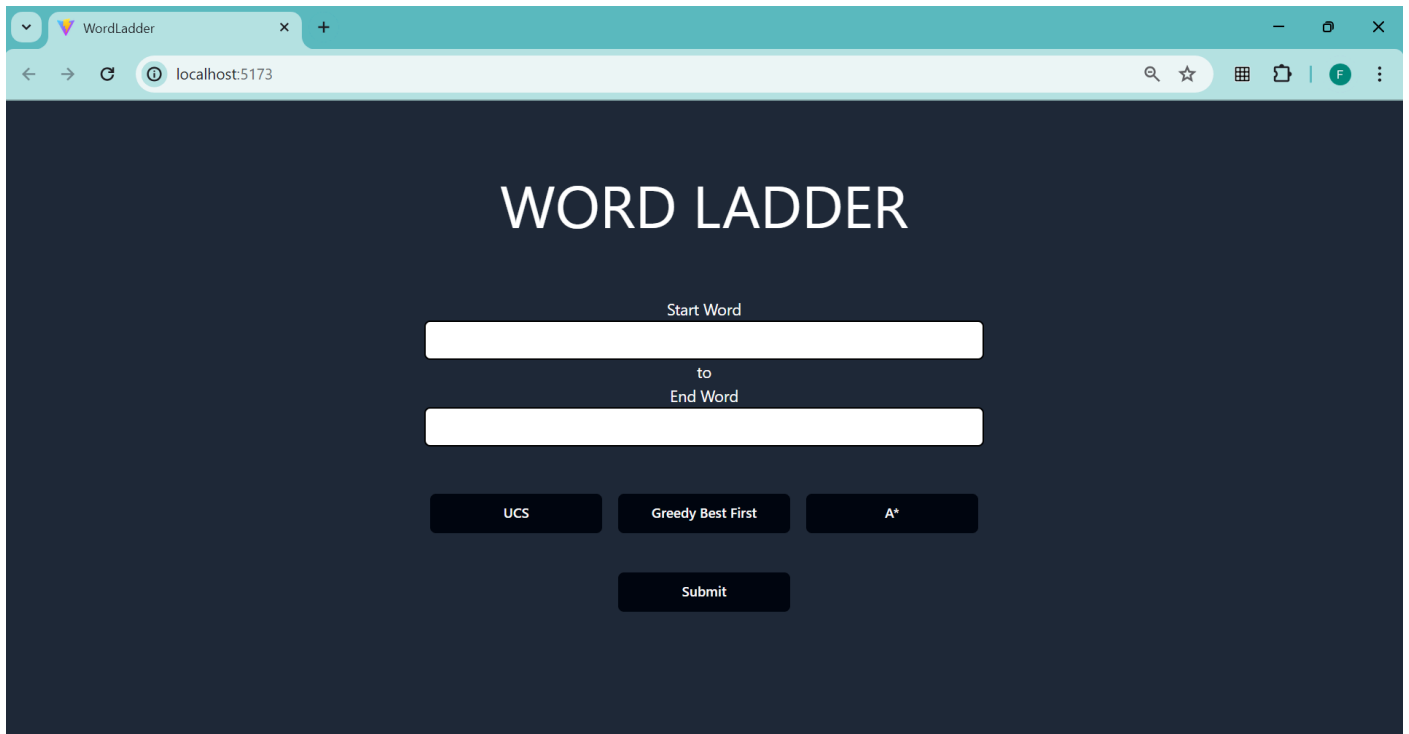
65     return () => {
66       if (intervalId) {
67         clearInterval(intervalId)
68       }
69     }
70   }, [isLoading])
71
72   const handleSubmit = () => {
73     if (startWord === '' || endWord === '' || algorithm === '') {
74       return
75     }
76     setData(null)
77     setIsLoading(true)
78   }
79
80   return (
81     <>
82     <div className="bg-gray-800 h-screen">
83       <div className="h-[200px]">
84         <h1 className="text-6xl text-white text-center p-[75px]">WORD LADDER</h1>
85       </div>
86       <div>
87         <div className="px-[30%] h-[200px]">
88           <label className="text-white flex justify-center">Start Word</label>
89           <input className="text-center text-[20px] border-black border-2"
90             type="text"
91             value={startWord}
92             onChange={(e) => setStartWord(e.target.value)}
93           />
94           <p className="text-white text-center">to />
95           <label className="text-white flex justify-center">End Word</label>
96           <input className="text-center text-[20px] border-black border-2"
97             type="text"
98             value={endWord}
99             onChange={(e) => setEndWord(e.target.value)}
100           />
101         </div>
102         <div className="flex justify-center gap-4 h-[75px]">
103           <button
104             className="cursor-pointer w [175px] bg-gray-950 hover:bg-gray-500 active:bg-gray-400 focus:outline-none focus:ring focus:ring-gray-400"
105             onClick={() => setAlgorithm('ucs')}>UCS</button>
106           <button
107             className="cursor-pointer w [175px] bg-gray-950 hover:bg-gray-500 active:bg-gray-400 focus:outline-none focus:ring focus:ring-gray-400"
108             onClick={() => setAlgorithm('gbfs')}>Greedy Best First</button>
109           <button
110             className="cursor-pointer w [175px] bg-gray-950 hover:bg-gray-500 active:bg-gray-400 focus:outline-none focus:ring focus:ring-gray-400"
111             onClick={() => setAlgorithm('astar')}>A*</button>
112         </div>
113         <div className="flex justify-center h-[50px]">
114           <Dialog>
115             <DialogTrigger>
116               <button className="cursor-pointer w-[175px] bg-gray-950 hover:bg-gray-500 active:bg-gray-400" onClick={handleSubmit}>Submit</button>
117             </DialogTrigger>
118             <isloading && {
119               <DialogContent>
120                 <DialogHeader>
121                   <DialogTitle>Loading...</DialogTitle>
122                 </DialogHeader>
123                 <DialogDescription>
124                   <p>Please wait for the result.</p>
125                 </DialogDescription>
126               </DialogContent>
127             )}
128             <data && { // Only render the DialogContent when there's no error and showDialog is true
129               <DialogContent>
130                 <DialogHeader>
131                   <DialogTitle>Result</DialogTitle>
132                 </DialogHeader>
133                 <DialogDescription>
134                   <data.path !== null ? {
135                     <div>
136                       <p>Path: {data.path.join(' -> ')}</p>
137                       <p>Execution Time: {data.executionTime} ms</p>
138                       <p>Node Visited: {data.totalNodesVisited}</p>
139                     </div>
140                   ) : {
141                     <div>
142                       <p>No path found</p>
143                       <p>Execution Time: {data.executionTime} ms</p>
144                       <p>Node Visited: {data.totalNodesVisited}</p>
145                     </div>
146                   }
147                 </DialogDescription>
148               </DialogContent>
149             </Dialog>
150           </div>
151         </div>
152       </div>
153     </div>
154   )
155 }
156 }
157
158 export default App

```

BAB 4

EKSPERIMEN DAN ANALISIS

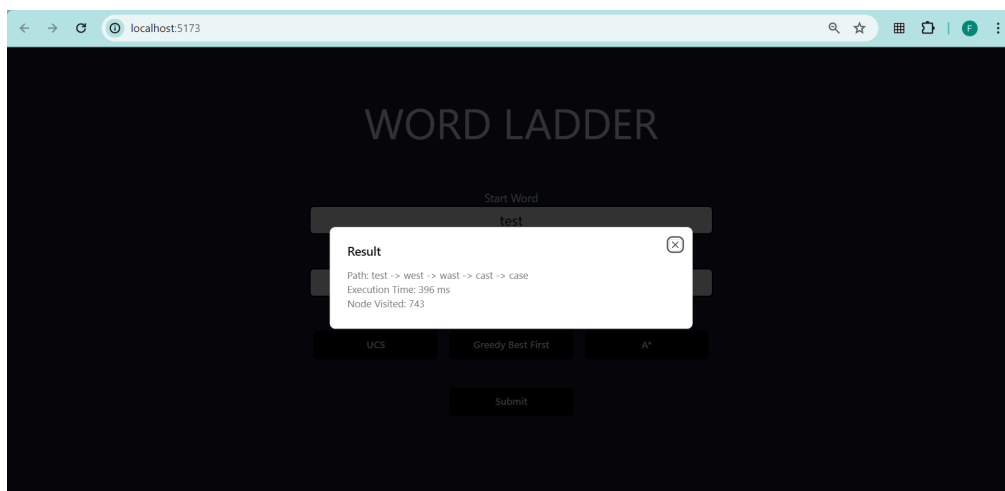
4.1 Tampilan Website



4.2 UCS

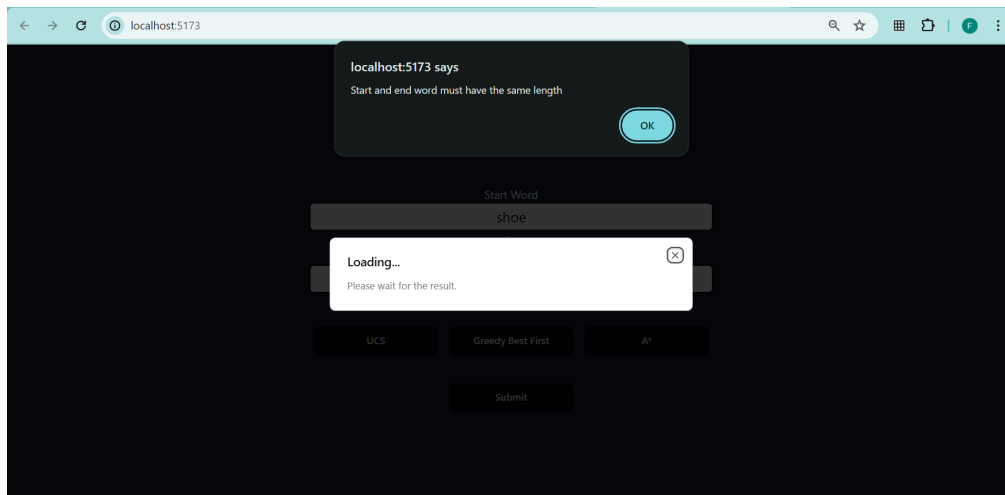
Start Word : test

End Word : case



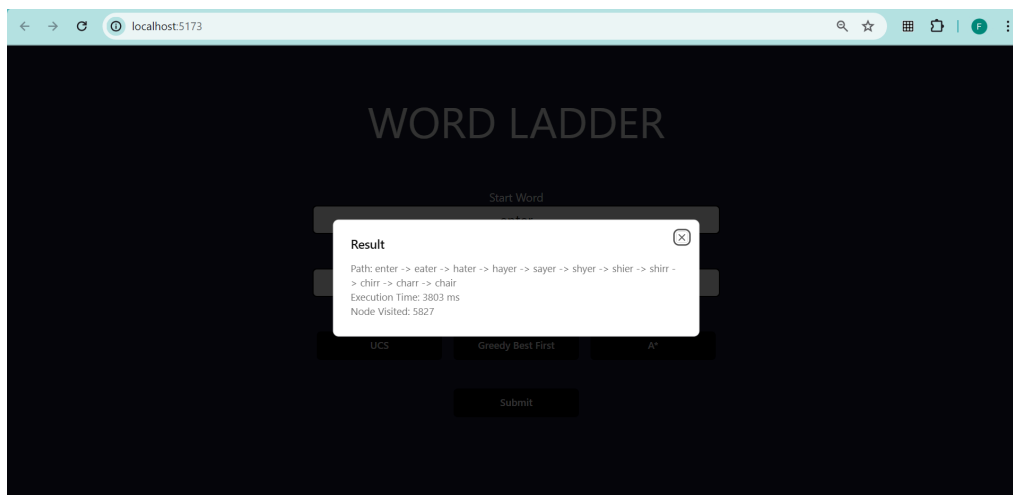
Start Word : shoe

End Word : car



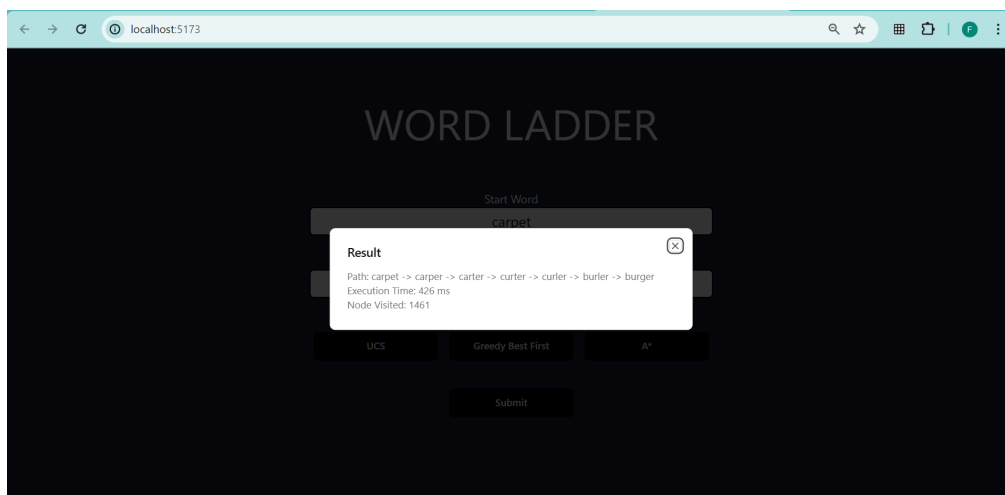
Start Word : enter

End Word : chair



Start Word : carpet

End Word : burger



Start Word : car

End Word : xyz

localhost:5173 says
Start or end word is not in the dictionary

OK

Start Word
car

to
End Word
xyz

UCS Greedy Best First A*

Submit

Start Word : board

End Word : human

WORD LADDER

Start Word
board

Result

No path found
Execution Time: 3915 ms
Node Visited: 7396

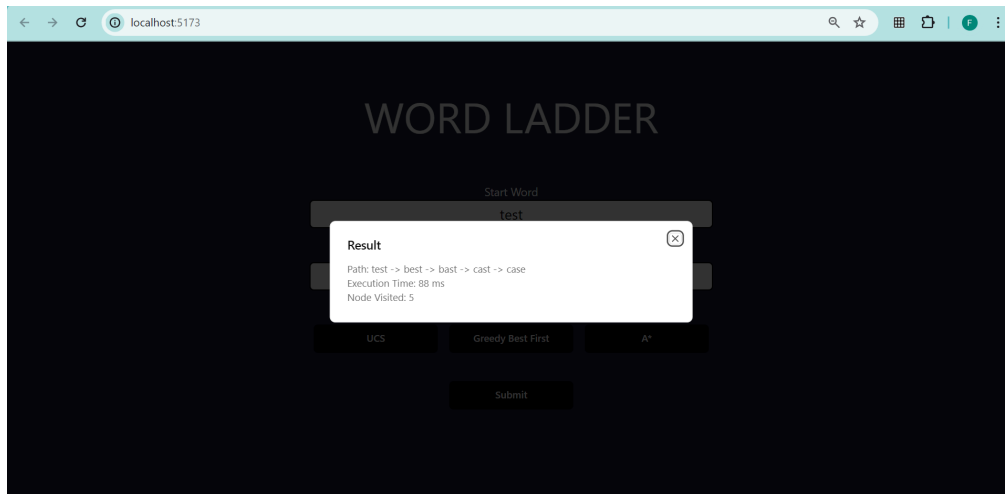
UCS Greedy Best First A*

Submit

4.3 Greedy Best First

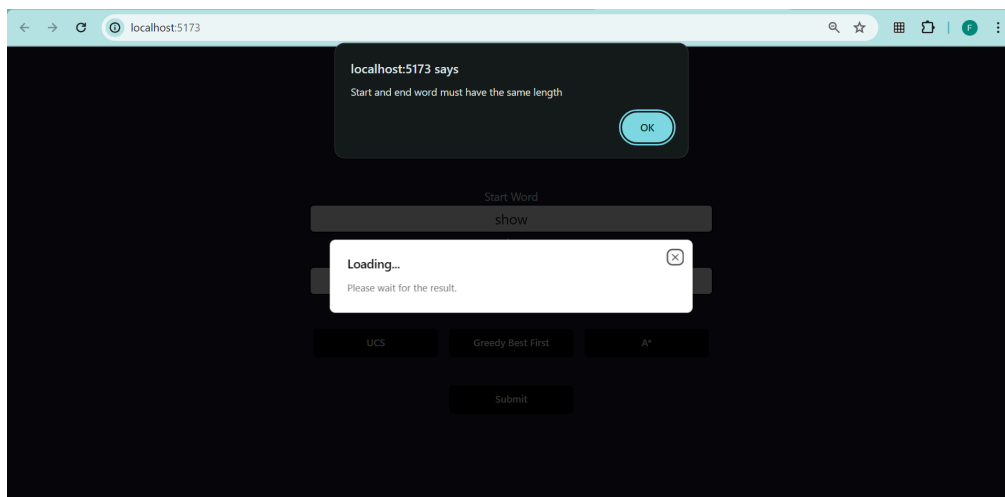
Start Word : test

End Word : case



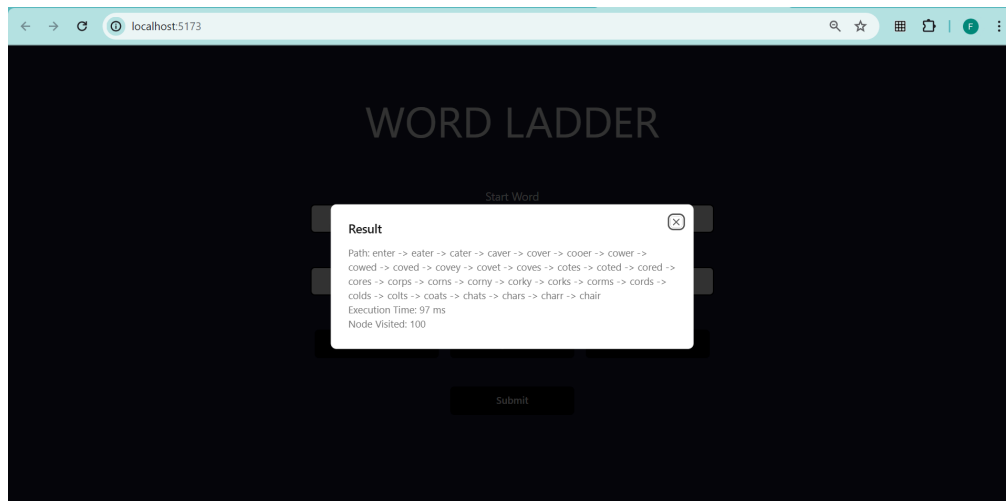
Start Word : shoe

End Word : car



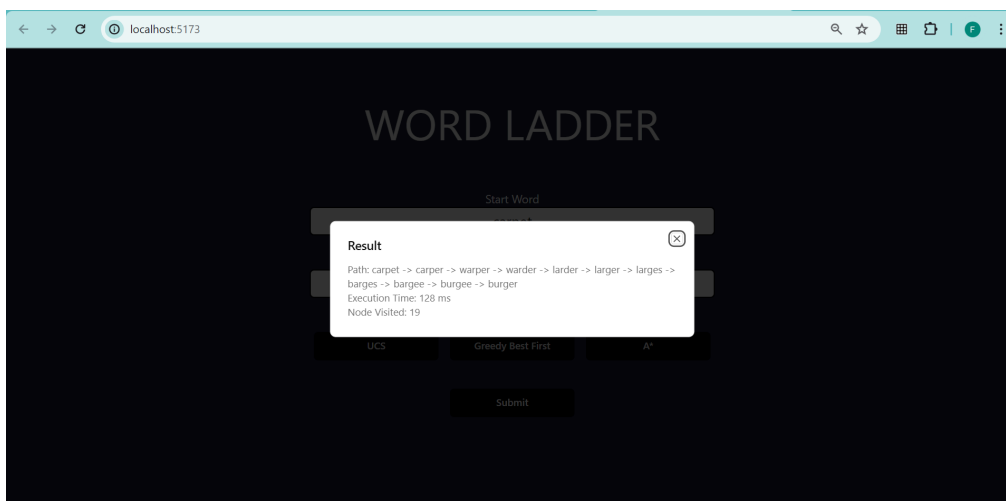
Start Word : enter

End Word : chair



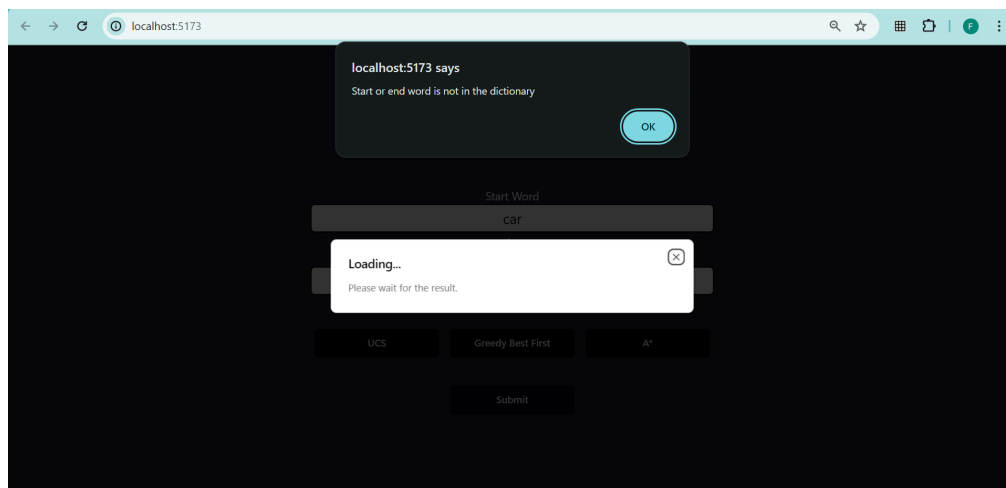
Start Word : carpet

End Word : burger



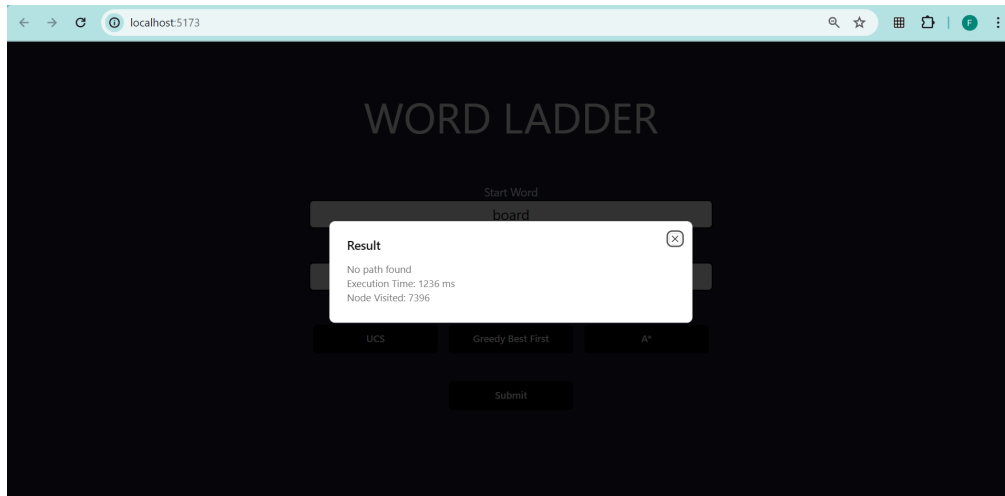
Start Word : car

End Word : xyz



Start Word : board

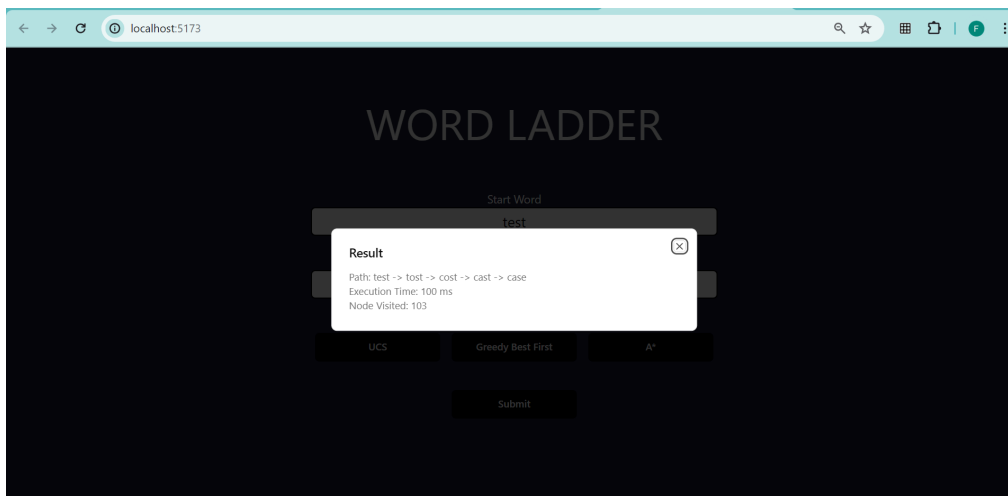
End Word : human



4.4 A*

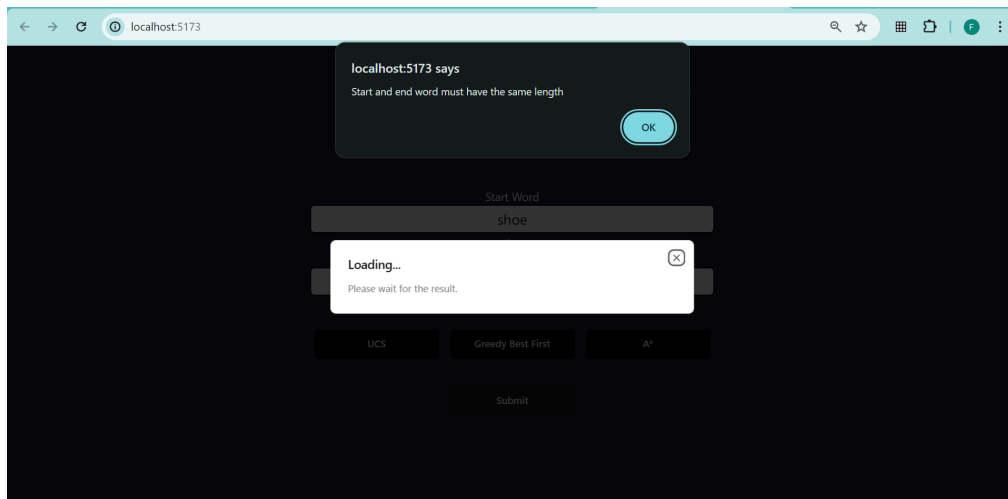
Start Word : test

End Word : case



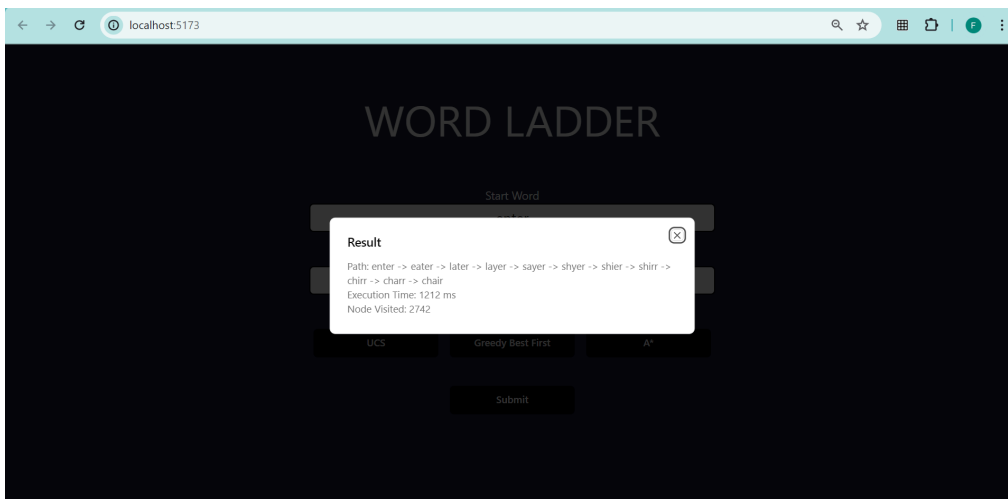
Start Word : shoe

End Word : car



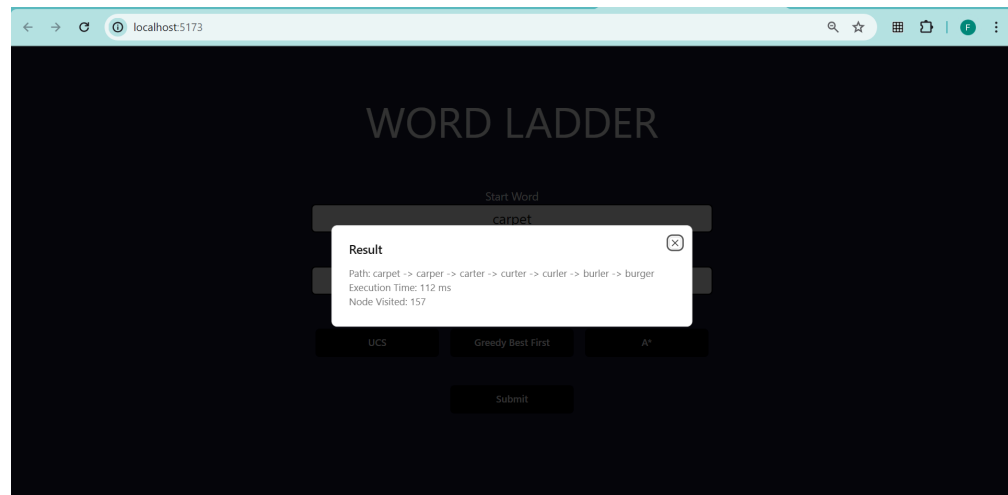
Start Word : enter

End Word : chair



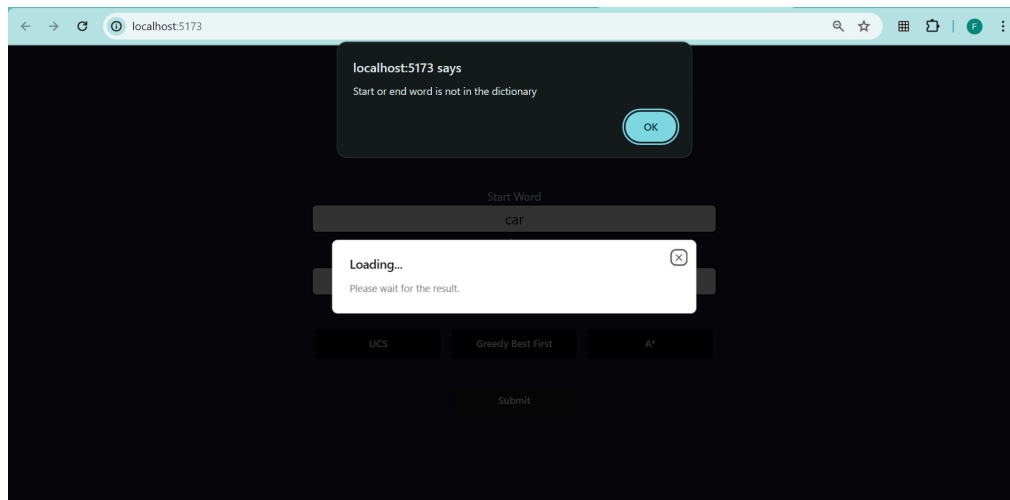
Start Word : carpet

End Word : burger



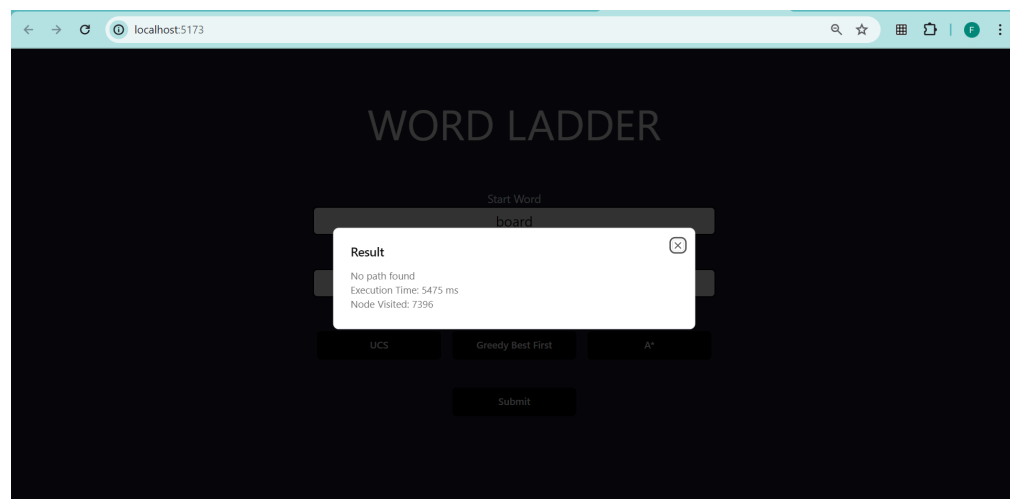
Start Word : car

End Word : xyz



Start Word : board

End Word : human



4.5 HASIL ANALISIS

Dari eksperimen di atas, dapat dilihat bahwa algoritma UCS memiliki hasil yang optimal namun dengan waktu eksekusi yang lambat karena tidak menggunakan nilai heuristik, memori yang digunakan pun lebih banyak yang mungkin dikarenakan oleh lambatnya waktu eksekusi tersebut. Algoritma GBFS tidak memiliki hasil yang optimal namun memiliki waktu eksekusi yang paling cepat dan penggunaan memorinya lebih kecil dibanding yang lain yang mungkin dikarenakan hanya mempertimbangkan informasi heuristik lokal pada setiap langkah. Algoritma A* memiliki hasil yang optimal, waktu eksekusi yang lebih cepat dan penggunaan memori yang lebih sedikit dibanding UCS serta waktu eksekusi yang sedikit lebih lambat dan penggunaan memori yang sedikit lebih banyak dibanding GBFS.

BAB 5

PENUTUP

5.1 Kesimpulan

Dari Tugas Kecil 3 ini, dapat disimpulkan bahwa berbagai algoritma pencarian seperti UCS, A*, dan Greedy Best-First Search memiliki pendekatan yang berbeda dalam menemukan jalur terpendek dalam persoalan word ladder. UCS menawarkan pendekatan yang lebih sistematis dengan mempertimbangkan biaya aktual atau menelusuri secara horizontal, sementara A* memanfaatkan nilai heuristik untuk estimasi jalur terpendek. Meskipun A* memiliki potensi untuk memberikan solusi yang lebih efisien jika nilai heuristiknya tepat, Greedy Best-First Search tidak menjamin solusi optimal karena hanya mempertimbangkan informasi heuristik lokal. Sementara itu, dalam hal penggunaan memori, Greedy Best-First Search biasanya lebih hemat daripada algoritma lainnya, tetapi efisiensi totalnya tergantung pada kualitas nilai heuristik dan kompleksitas masalah yang dihadapi. Oleh karena itu, dalam memilih algoritma pencarian untuk word ladder, perlu mempertimbangkan trade-off antara optimalitas solusi, kebutuhan memori, dan kinerja algoritma sesuai dengan kebutuhan dan karakteristik spesifik dari masalah yang dihadapi.

DAFTAR PUSTAKA

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

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

LAMPIRAN

LINK REPOSITORY

Link repository GitHub: https://github.com/fnathas/Tucil3_13522145

| 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 | ✓ | |