IF2211- Strategi Algoritma

Tugas Kecil 3

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

Disusun Oleh:

Moh Fairuz Alauddin Yahya 13522057

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

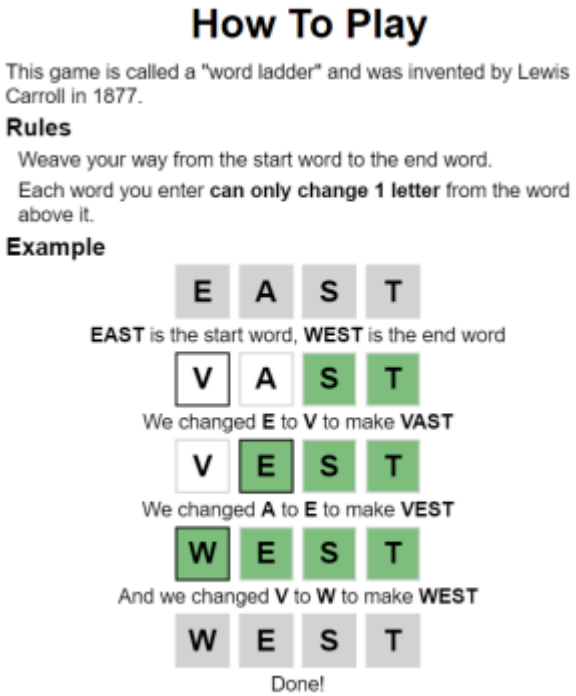
INSTITUT TEKNOLOGI BANDUNG

2024

DAFTAR ISI

BAB I
DESKRIPSI TUGAS

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



Gambar 1 Ilustrasi dan Peraturan Permainan Word Ladder

(Sumber: https://wordwormdormdork.com/)

BAB II
LANDASAN TEORI

2.1. Dasar Teori

Graf adalah struktur data non-linear yang terdiri dari vertex (simpul) dan edge (tepi). Vertex terkadang disebut juga simpul, sedangkan edge berupa garis atau lengkung yang menghubungkan dua simpul manapun dalam graf. Secara lebih formal, graf tersusun dari kumpulan vertex (V) dan kumpulan edge (E). Graf dinyatakan dengan G(V, E).

Struktur data graf merupakan alat yang ampuh untuk merepresentasikan dan menganalisis hubungan kompleks antara objek atau entitas. Graf sangat berguna dalam bidang-bidang seperti analisis jaringan sosial, sistem rekomendasi, dan jaringan komputer. Dalam bidang ilmu data olahraga, struktur data graf dapat digunakan untuk menganalisis dan memahami dinamika performa tim dan interaksi antar pemain di lapangan.

2.2. Proses Pemetaan Persoalan

Aturan dalam permainan ini adalah mencari susunan kata yang tepat dengan hanya mengubah satu huruf di tiap iterasi dari susunan sebelumnya hingga mencapai kata tujuan yang sesuai dengan input, sehingga terdapat berbagai kemungkinan susunan kata yang bisa dibentuk untuk mencapai kata tujuan.

Proses pencarian solusi dalam permainan ini dapat direpresentasikan dalam bentuk graf, di mana setiap simpulnya memiliki definisi sebagai berikut:

- Simpul: Mewakili kata yang dikunjungi pada setiap langkah permainan.
- Sisi: Mewakili langkah yang diambil di setiap iterasi permainan untuk mengubah satu huruf pada kata sebelumnya dan membentuk kata baru.

Dalam penyelesaian masalah ini, tujuan dari program adalah menemukan sisi yang harus dipilih dari simpul awal hingga mencapai kata tujuan yang telah diberikan melalui input.

Untuk menyelesaikan tugas ini, terdapat 3 algoritma yang akan dibahas dan diimplementasikan sebagai berikut.

2.1.1. Uniform Cost Search (UCS)

Uniform Cost Search (UCS) adalah algoritma pencarian jalur yang digunakan untuk menemukan jalur terpendek dengan *cost* terendah dari satu simpul ke simpul lain dalam sebuah graf berbobot. Algoritma ini mempertimbangkan *cost* lintasan dari simpul awal ke simpul-simpul lain dan secara bertahap mengembangkan jalur dengan memilih simpul berdasarkan *cost* terendah yang telah ditempuh. Kelebihan utama UCS adalah kemampuannya menghasilkan solusi optimal, yang berarti jalur yang ditemukan adalah jalur terpendek atau *cost* terendah. UCS memprioritaskan simpul dengan cost terendah untuk dieksplorasi lebih lanjut, namun dapat mengalami kompleksitas waktu yang tinggi jika terdapat banyak simpul dengan cost yang sama. Hal ini dapat mempengaruhi penggunaan memori terutama pada graf yang besar karena perlu menyimpan informasi cost lintasan terendah dari simpul awal ke simpul lain.

Dalam konteks Word Ladder, di mana setiap langkah hanya melibatkan perubahan satu karakter pada kata sebelumnya untuk mencapai kata tujuan, UCS memanfaatkan fungsi $f(n)=g(n)$ untuk mengevaluasi *cost*, di mana definisi $g(n)$ adalah kedalaman dari simpul awal hingga simpul ke-n.

Langkah-langkah pencarian solusi dengan algoritma UCS pada kasus word ladder solver sebagai berikut:

1. Inisialisasi sebuah PriorityQueue yang akan menyimpan simpul-simpul yang akan dieksplorasi, dengan prioritas berdasarkan *cost* yang ditempuh untuk mencapai simpul tersebut terendah dan sebuah HashSet "visited" digunakan untuk menyimpan kata-kata yang telah dieksplorasi sehingga tidak dieksplorasi lagi pada langkah berikutnya.
2. Tambahkan kata start sebagai simpul awal ke dalam PriorityQueue dengan *cost* 0.
3. Lakukan iterasi pemrosesan selama PriorityQueue tidak kosong.
4. Pada setiap iterasi, ambil simpul dengan *cost* terendah dari PriorityQueue.
5. Periksa apakah kata dari simpul yang diambil telah dieksplorasi sebelumnya. Jika belum, maka lakukan *register* kata tersebut.
6. Pada tiap iterasi, lakukan perbandingan kata dari simpul yang diambil dengan kata tujuan, jika sama maka program akan menghentikan eksekusi

dan mengembalikan susunan kata melalui pembangunan *path* dari parent tiap simpul.

7. Jika belum ditemukan kata tujuan, algoritma mencari *neighbor* dari kata saat ini dengan mengubah satu karakter pada setiap langkahnya dan memeriksa apakah kata *neighbor* tersebut adalah kata yang valid dalam kamus.
8. Jika kata *neighbor* tersebut adalah kata yang valid, maka tetangga tersebut ditambahkan ke dalam PriorityQueue dengan cost yang dihitung berdasarkan *cost* saat ini ditambah 1.
9. Langkah-langkah ini terus berulang sampai PriorityQueue kosong atau kata tujuan ditemukan.
10. Jika PriorityQueue kosong tanpa menemukan kata tujuan, algoritma mengembalikan hasil bahwa solusi tidak ditemukan.

Pada kasus Word Ladder, urutan ekspansi node yang dihasilkan oleh algoritma ini mengikuti pola yang mirip dengan Breadth-First Search (BFS) dengan asumsi bahwa PriorityQueue tidak mengacak urutan simpul yang lebih panjang dari antrian inputnya, sehinggaa akan selaras menurut BFS di mana pencarian dimulai dari kedalaman terendah, lalu mengeksplorasi simpul dengan kedalaman yang sama hingga habis dan lanjut pada kedalaman yang lebih tinggi.

2.1.2. Greedy Best First Search

Greedy Best-First Search (GBFS) adalah algoritma pencarian jalur yang mengutamakan ekspansi ke simpul yang memiliki *cost* heuristik terendah, tanpa mempertimbangkan *cost* sebenarnya dari simpul awal ke simpul tersebut. Algoritma ini cenderung memilih jalur dengan *cost* heuristik terendah pada setiap langkahnya, sehingga bisa saja tidak menghasilkan solusi optimal.

Kelebihan dari GBFS adalah kemampuannya mengekspansi jalur dengan *cost* heuristik terendah, yang sering kali mengarahkan pencarian ke arah solusir. Algoritma ini juga efisien dalam menghemat waktu maupun memori karena hanya mempertimbangkan *cost* heuristik.Namun, GBFS memiliki beberapa kelemahan, dimana algoritma ini tidak menjamin solusi optimal karena hanya mengutamakan *cost* heuristik tanpa mempertimbangkan *cost* sebenarnya dari simpul awal ke simpul tujuan. Kedua, GBFS bisa terjebak dalam loop atau jalur yang tidak

menghasilkan solusi jika *cost* heuristik tidak mencerminkan estimasi yang akurat dari *cost* sebenarnya.

Dalam konteks Word Ladder, GBFS digunakan untuk mencari jalur solusi dengan memanfaatkan fungsi *f(n)=h(n),* dimana heuristik *h(n)* adalah jumlah karakter berbeda antara suatu kata dengan kata tujuan. Namun permasalahannya bahwa algoritma ini bisa terjebak pada local minima, dimana algoritma ini hanya memperbolehkan melalui hanya 1 simpul di tiap iterasi yang memungkinkan solusi tidak bisa didapatkan. Namun dari segi penggunaan *resource* dari algoritma ini tentu adalah yang terbaik.

Langkah-langkah pencarian solusi dengan algoritma Greedy Best First Search pada kasus word ladder solver sebagai berikut:

1. Inisialisasi HashSet "visited" digunakan untuk menyimpan kata-kata yang telah dieksplorasi sehingga tidak dieksplorasi lagi pada langkah berikutnya.
2. Tambahkan simpul awal sebagai simpul currentNode dengan nilai heuristik adalah jumlah karakter berbeda terhadap kata tujuan.
3. Lakukan iterasi pemrosesan selama currentNode memiliki nilai dan tidak kosong.
4. Pada setiap iterasi, ambil kata dari simpul currentNode untuk dieksplorasi lebih lanjut.
5. Lakukan perbandingan kata yang diambi dengan kata tujuan. Jika sama, algoritma mengembalikan hasil berupa jalur yang telah ditemukan.
6. Jika belum ditemukan kata tujuan, algoritma mencari *neighbor* dengan cost minimum (greedy) berdasarkan heuristik *h(n)*.
7. Cari *neighbor* dengan mengubah satu karakter pada setiap langkahnya dan memeriksa apakah tetangga tersebut adalah kata yang valid dalam kamus dan belum pernah dieksplorasi sebelumnya.
8. Jika *neighbor* tersebut memenuhi syarat, tambahkan *neighbor* tersebut ke dalam visited dan hitung nilai heuristiknya.
9. Pilih *neighbor* dengan heuristik minimum sebagai simpul currentNode untuk langkah berikutnya.
10. Langkah-langkah ini terus berulang sampai currentNode kosong atau kata tujuan ditemukan.

11. Jika currentNode kosong tanpa menemukan kata tujuan, algoritma mengembalikan hasil bahwa solusi tidak ditemukan.

2.1.3. A*

Algoritma A* adalah algoritma pencarian jalur yang menggabungkan *cost* sebenarnya dari lintasan yang telah ditempuh dengan heuristik untuk menentukan langkah-langkah selanjutnya. Algoritma ini menggabungkan pendekatan Greedy Best-First Search (GBFS) dengan Uniform Cost Search (UCS), di mana ia mempertimbangkan *cost* sebenarnya dari lintasan sejauh ini *g(n)* dan perkiraan cost yang tersisa menuju tujuan *h(n)* untuk menentukan prioritas ekspansi simpul. Sehingga *cost* yang dipertimbangkan pada algoritma ini dinyatakan dalam *f(n)=g(n)+h(n),* dimana *g(n)* adalah kedalaman suatu node ke-*n* dan *h(n)* adalah jumlah karakter beda antar kata node dengan kata tujuan. Untuk memastikan bahwa heuristik *f(n)* bersifat admissible, maka perlu dilakukan *weighting*, dimana nilai *f(n)=f(n-1)*+panjang kata tujuan. Hal ini dilakukan agar *cost* yang sebenarnya dari *g(n)* tidak di-*underestimate* oleh *h(n)*.

Kelebihan dari algoritma A* adalah kemampuannya menghasilkan solusi optimal (jika heuristiknya admissible) serta efisiensinya dalam mengekspansi jalur-jalur yang memiliki potensi untuk menjadi solusi optimal. Namun, A* juga memiliki beberapa kelemahan, yaitu keakuratan heuristik yang digunakan sangat penting untuk menjamin solusi optimal. Jika heuristik tidak admissible atau tidak akurat, algoritma ini dapat menghasilkan solusi yang suboptimal. Algoritma A* dapat memerlukan lebih banyak sumber daya dibandingkan dengan algoritma UCS, terutama jika heuristik yang digunakan kompleks.

Dalam konteks Word Ladder, di mana setiap langkah hanya melibatkan perubahan satu karakter pada kata sebelumnya, heuristik yang digunakan ini memberikan estimasi *cost* yang sesuai dengan kriteria *admissible*. Karena Word Ladder membatasi langkah-langkah hanya pada perubahan satu karakter, heuristik ini sesuai dengan definisi admissible yang mengharuskan estimasi *cost* tidak melebihi *cost* sebenarnya dari simpul saat ini ke simpul tujuan.

Algoritma A* menggunakan pendekatan yang lebih cerdas dengan menggabungkan *cost,* sehingga A* dapat memprioritaskan ekspansi simpul-simpul

yang memiliki estimasi *cost* terendah menuju tujuan, sehingga mengurangi jumlah simpul yang dieksplorasi dan meningkatkan efisiensi pencarian. Maka A* lebih baik dari UCS dala kasus Word Ladder.

Langkah-langkah pencarian solusi dengan algoritma A* pada kasus word ladder solver sebagai berikut:

1. Inisialisasi sebuah PriorityQueue yang akan menyimpan simpul-simpul yang akan dieksplorasi, dengan prioritas berdasarkan *cost* yang ditempuh untuk mencapai simpul tersebut ditambah nilai heuristik *h(n)* yang merupakan jumlah karakter beda dengan kata tujuan terendah dan sebuah HashSet "visited" digunakan untuk menyimpan kata-kata yang telah dieksplorasi sehingga tidak dieksplorasi lagi pada langkah berikutnya.
2. Tambahkan kata start sebagai simpul awal ke dalam PriorityQueue dengan *cost* adalah nilai heuristic *h(n)*.
3. Lakukan iterasi pemrosesan selama PriorityQueue tidak kosong.
4. Pada setiap iterasi, ambil simpul dengan *cost* terendah dari PriorityQueue.
5. Periksa apakah kata dari simpul yang diambil telah dieksplorasi sebelumnya. Jika belum, maka lakukan *register* kata tersebut.
6. Pada tiap iterasi, lakukan perbandingan kata dari simpul yang diambil dengan kata tujuan, jika sama maka program akan menghentikan eksekusi dan mengembalikan susunan kata melalui pembangunan *path* dari parent tiap simpul.
7. Jika belum ditemukan kata tujuan, algoritma mencari *neighbor* dari kata saat ini dengan mengubah satu karakter pada setiap langkahnya dan memeriksa apakah kata *neighbor* tersebut adalah kata yang valid dalam kamus.
8. Jika kata *neighbor* tersebut adalah kata yang valid, maka tetangga tersebut ditambahkan ke dalam PriorityQueue dengan cost yang dihitung berdasarkan *cost* saat ini ditambah 1 dan nilai heuristic *h(n)*.
9. Langkah-langkah ini terus berulang sampai PriorityQueue kosong atau kata tujuan ditemukan.
10. Jika PriorityQueue kosong tanpa menemukan kata tujuan, algoritma mengembalikan hasil bahwa solusi tidak ditemukan.

BAB III
IMPLEMENTASI DAN PENGUJIAN

3.1. Implementasi

Pengimplementasian algoritma pada bagian backend menggunakan bahasa Java dengan bantuan framework Spring Boot. Pengelolaan backend dilakukan dengan menghasilkan REST API sebagai media komunikasi dengan frontend.

1. Kelas Dictionary

Kelas ini bertanggung jawab untuk melakukan load file txt yang akan digunakan sebagai kaus dalam pencocokan string yang valid

```
public class Dictionary {
    private HashSet<String> dictionary = new
HashSet<>();

    public Dictionary(String filepath) throws
Exception {
        FileReader fr= new FileReader(filepath);
        BufferedReader reader = new
BufferedReader(fr);

        String word = reader.readLine();
        while (word != null) {
            this.dictionary.add(word);
            word = reader.readLine();
        }

        fr.close();
    }

    public boolean isValidWord(String word) {
        return this.dictionary.contains(word);
    }

}
```

2. **Kelas Node**

Kelas node merepresentasikan node dari graph yang menyimpan informasi kata, *cost*, dan parent dari suatu node. Kelas ini juga bertanggung jawab untuk merekonstruksi path hasil solver.

```
public class Node implements Comparable<Node> {
    private int cost;
    private String word;
    private Node parent;

    public Node(String word, int cost) {
        this.word = word;
        this.cost = cost;
    }

    public Node(String word, int cost, Node parent)
{
        this.word = word;
        this.cost = cost;
        this.parent = parent;
    }

    public Node(String word) {
        this.word = word;
        this.cost = 0;
    }

    public int getCost() {
        return cost;
    }

    public String getWord() {
        return word;
    }

    public Node getParent() {
        return parent;
    }

    @Override
```

```
    public int compareTo(Node other) {
        return Integer.compare(this.getCost(),
other.getCost());
    }

    // Build the path from the start node to the
current node
    public static ArrayList<String> buildPath(Node
node) {
        ArrayList<String> path = new ArrayList<>();
        Node current = node;
        while (current != null) {
            path.addFirst(current.getWord());
            current = current.getParent();
        }
        return path;
    }
}
```

3. Kelas Heuristic

Kelas ini bertanggung jawab untuk menentukan value heuristic dari *h(n)* yang merupakan jumlah karakter beda dari suatu kata dengan kata tujuan.

```
public class Heuristic {
    public static int getDistance(String startWord,
String targetWord) {
        // Make sure the length of the two words are
the same
        if (startWord.length() !=
targetWord.length())
            return Integer.MAX_VALUE;

        // Calculate the same character count at the
same position
        int diffCount = 0;
        for (int i = 0; i < startWord.length(); ++i)
{
            if (startWord.charAt(i) !=
targetWord.charAt(i))
```

```
                diffCount++;
            }

            return diffCount;
        }
}
```

4. **Kelas UCS**

Kelas ini bertanggung jawab untuk solver dengan algoritma UCS, yag mengimplementasikan interface PathFindingAlgorithm.

```
public class UCS implements PathFindingAlgorithm {
    @Override
    public PathFindingResult findPath(String
startWord, String endWord, Dictionary dictionary) {
        PriorityQueue<Node> queue = new
PriorityQueue<>();
        HashSet<String> visited = new HashSet<>();
        queue.add(new Node(startWord));
        int counter = 0;

        while (!queue.isEmpty()) {
            Node currentNode = queue.remove();
            String currentWord =
currentNode.getWord();

            counter++;

            // goalNode found
            if
(currentWord.equalsIgnoreCase(endWord)) {
                return new
PathFindingResult(Node.buildPath(currentNode),
counter);
            }

            // Find neighbors of the currentNode
word with different character at one
            // position
```

```
            for (int i = 0; i <
currentWord.length(); i++) {
                // Try to change the character to
all possible characters
                for (char c = 'a'; c <= 'z'; c++) {
                    if (c != currentWord.charAt(i))
{
                        String neighbor =
currentWord.substring(0, i) + c +
currentWord.substring(i + 1);
                        // Append the neighbor to
the ArrayList if it is a valid word
                        if
(dictionary.isValidWord(neighbor) &&
!visited.contains(neighbor)) {
                            visited.add(neighbor);
                            queue.add(new
Node(neighbor, currentNode.getCost() + 1,
currentNode));
                        }
                    }
                }
            }

        }
        return new PathFindingResult(null, counter);
// No solution found
    }
}
```

5. **Kelas GreedyBFS**

Kelas ini bertanggungjawab sebagi solver dengan menggunakan algoritma Greedy Best First Search.

```
public class GreedyBFS implements
PathFindingAlgorithm {
    @Override
    public PathFindingResult findPath(String
startWord, String endWord, Dictionary dictionary) {
```

```
        Node currentNode = new Node(startWord,
Heuristic.getDistance(startWord, endWord));
        HashSet<String> visited = new HashSet<>();
        int counter = 0;

        // Search until the end word is found
        while (currentNode != null) {
            counter++;
            String currentWord =
currentNode.getWord();

            // If the current word is the end word,
return the path
            if
(currentWord.equalsIgnoreCase(endWord)) {
                return new
PathFindingResult(Node.buildPath(currentNode),
counter);
            }

            int minimumCost = Integer.MAX_VALUE;
            String greedyWord = null;

            // Find the neighbor with the minimum
cost
            for (int i = 0; i <
currentWord.length(); i++) {
                for (char c = 'a'; c <= 'z'; c++) {
                    if (currentWord.charAt(i) != c)
{
                        // Replace the character at
index i with c
                        String newWord =
currentWord.substring(0, i) + c +
currentWord.substring(i + 1);
                        // Check if the new word is
a valid English word and has not been visited
                        if
(dictionary.isValidWord(newWord) &&
!visited.contains(newWord)) {
```

```
                                visited.add(newWord);
                                // Calculate the cost of
the new word
                                int cost =
Heuristic.getDistance(newWord, endWord);
                                if (cost <= minimumCost)
{
                                    minimumCost = cost;
                                    greedyWord =
newWord;
                                }
                            }
                        }
                    }
                }

                if (greedyWord != null) {
                    Node neighborNode = new
Node(greedyWord, minimumCost, currentNode);
                    currentNode = neighborNode;
                } else {
                    currentNode = null;
                    break;
                }
            }

            return new PathFindingResult(null, counter);

        }
}
```

6. **Kelas AStar**

Kelas ini bertanggungjawab dalam solver pemrosesan menggunakan algoritma A*.

```
public class AStar implements PathFindingAlgorithm {
    @Override
    public PathFindingResult findPath(String
startWord, String endWord, Dictionary dictionary) {
```

```
        HashSet<String> visited = new HashSet<>();
        PriorityQueue<Node> queue = new
PriorityQueue<>();
        queue.add(new Node(startWord,
Heuristic.getDistance(startWord, endWord)));

        int counter = 0;

        while (!queue.isEmpty()) {
            Node currentNode = queue.remove();
            String currentWord =
currentNode.getWord();

            counter++;
            visited.add(currentWord);

            // Solution found
            if
(currentWord.equalsIgnoreCase(endWord)) {
                return new
PathFindingResult(Node.buildPath(currentNode),
counter);
            }

            // Search for neighbors of the
currentNode word with different character at one
            // position
            for (int i = 0; i <
currentWord.length(); i++) {
                // Try to change the chdaracter to
all possible characters
                for (char c = 'a'; c <= 'z'; c++) {
                    if (c != currentWord.charAt(i))
{
                        String neighbor =
currentWord.substring(0, i) + c +
currentWord.substring(i + 1);
                        // Append the neighbor to
the list if it is a valid word and has not been
                        if
```

```
(dictionary.isValidWord(neighbor) &&
!visited.contains(neighbor)) {
                            visited.add(neighbor);
                            int cost =
currentNode.getCost() + endWord.length() +
Heuristic.getDistance(neighbor, endWord);
                            queue.add(new
Node(neighbor, cost, currentNode));
                        }
                    }
                }
            }

        }

        return new PathFindingResult(null, counter);
// No solution found
    }
}
```

7. **Kelas PathFindingService**

Kelas ini bertanggungjawab untuk menentukan algoritma apa yang akan digunakan berdasarkan query dari user menggunakan spring boot java.

```
@Service
public class PathFindingService {
    public PathFindingAlgorithm getAlgorithm(String
method) {
        switch (method) {
            case "ucs":
                return new UCS();
            case "greedy":
                return new GreedyBFS();
            case "astar":
                return new AStar();
            default:
                throw new
IllegalArgumentException("Invalid algorithm method:
" + method);
```

```
        }
    }
}
```

8. **Kelas AlgorithmController**

Kelas ini bertanggung jawab dalam pengelolaan semua kelas, sekaligus menjadi REST API yang menghunbungkan logic BE dengan FE.

```
@RestController
public class AlgorithmController {

    private final PathFindingService
pathFindingService;
    private final Dictionary dictionary;

    @Autowired
    public AlgorithmController(PathFindingService
pathFindingService) {
        this.pathFindingService =
pathFindingService;
        try {
            this.dictionary = new
Dictionary("src/main/resources/dictionary.txt");
        } catch (Exception e) {
            throw new RuntimeException("Failed to
load dictionary");
        }
    }

    @PostMapping("/algorithm")
    public ResponseEntity<PathResponse> findPath(
            @RequestParam(value = "method",
defaultValue = "ucs") String method,
            @RequestBody PathRequest request) {
        try {
            String startWord =
request.getStartWord().toLowerCase();
            String endWord =
request.getEndWord().toLowerCase();
```

```
            if (startWord.length() !=
endWord.length()) {
                return
ResponseEntity.badRequest().body(new
PathResponse(null, 0, 0, "Start and end words must
have the same length"));
            }

            if (!dictionary.isValidWord(startWord))
{
                return
ResponseEntity.badRequest().body(new
PathResponse(null, 0, 0, "Start word is not
valid"));
            }

            if (!dictionary.isValidWord(endWord)) {
                return
ResponseEntity.badRequest().body(new
PathResponse(null, 0, 0, "End word is not valid"));
            }

            PathFindingAlgorithm algorithm =
pathFindingService.getAlgorithm(method);
            long startTime = System.nanoTime();
            PathFindingResult sol =
algorithm.findPath(startWord, endWord,
this.dictionary);
            long endTime = System.nanoTime();
            ArrayList<String> path = sol.getPath();
            int counter = sol.getCounter();
            double msruntime = (endTime - startTime)
/ 1e6;
            if (path == null) {
                return ResponseEntity.ok(new
PathResponse(null, msruntime, counter, "No path
found"));
            }
            return ResponseEntity.ok(new
```

```
PathResponse(path, msruntime, counter, "Path
found"));
        } catch (IllegalArgumentException e) {
            return ResponseEntity.badRequest()
                    .body(new PathResponse(null, 0,
0, "Invalid algorithm method: " + method));
        }
    }
}
```

3.2. Pengujian

1. eat-let

```
==========RESULT==========
Start Word: eat
End Word: let
Algorithm: Uniform Cost Search
Runtime: 1.2377ms
Visited Nodes: 107
Path:
1. eat
2. lat
3. let
```

```
==========RESULT==========
Start Word: eat
End Word: let
Algorithm: Greedy Best First Search
Runtime: 0.0851ms
Visited Nodes: 3
Path:
1. eat
2. lat
3. let
```

```
==========RESULT==========
Start Word: eat
End Word: let
Algorithm: A*
Runtime: 0.577ms
Visited Nodes: 21
Path:
1. eat
2. lat
3. let
```

2. **east-west**

```
==========RESULT==========
Start Word: east
End Word: west
Algorithm: Uniform Cost Search
Runtime: 1.1045ms
Visited Nodes: 83
Path:
1. east
2. wast
3. west
```

```
==========RESULT==========
Start Word: east
End Word: west
Algorithm: Greedy Best First Search
Runtime: 0.0961ms
Visited Nodes: 3
Path:
1. east
2. wast
3. west
```

```
==========RESULT==========
Start Word: east
End Word: west
Algorithm: A*
Runtime: 0.2574ms
Visited Nodes: 16
Path:
1. east
2. wast
3. west
```

3. **green-black**

```
==========RESULT==========
Start Word: green
End Word: black
Algorithm: Uniform Cost Search
Runtime: 40.6391ms
Visited Nodes: 2252
Path:
1. green
2. grees
3. brees
```

```
4. braes
5. brans
6. brank
7. blank
8. black
```

```
 ==========RESULT==========
Start Word: green
End Word: black
Algorithm: Greedy Best First Search
Runtime: 0.5736ms
Visited Nodes: 13
Path:
Not found
```

```
==========RESULT==========
Start Word: green
End Word: black
Algorithm: A*
Runtime: 11.0021ms
Visited Nodes: 632
Path:
1. green
2. grees
3. brees
4. braes
5. brans
6. brank
7. blank
8. black
```

4. **waiting-unbaked**

```
==========RESULT==========
Start Word: waiting
End Word: unbaked
Algorithm: Uniform Cost Search
Runtime: 177.0891ms
Visited Nodes: 7254
Path:
1. waiting
2. wairing
3. warring
4. parring
5. parking
6. perking
7. jerking
```

8. jerkins
9. jerkies
10. jerkier
11. perkier
12. peakier
13. beakier
14. brakier
15. brasier
16. brasher
17. brashes
18. brasses
19. trasses
20. tresses
21. cresses
22. creases
23. ureases
24. uneases
25. uncases
26. uncased
27. uncaked
28. unbaked

==========RESULT==========
Start Word: waiting
End Word: unbaked
Algorithm: Greedy Best First Search
Runtime: 0.2279ms
Visited Nodes: 3
Path:
Not found

==========RESULT==========
Start Word: waiting
End Word: unbaked
Algorithm: A*
Runtime: 171.6336ms
Visited Nodes: 6997
Path:
1. waiting
2. wairing
3. pairing
4. parring
5. parking
6. perking
7. jerking
8. jerkins
9. jerkies
10. jerkier

11. perkier
12. peakier
13. beakier
14. brakier
15. brasier
16. brasher
17. brashes
18. brasses
19. trasses
20. tresses
21. cresses
22. creases
23. ureases
24. uneases
25. uncases
26. uncased
27. uncaked
28. unbaked

5. **four-nine**

==========RESULT==========
Start Word: four
End Word: nine
Algorithm: Uniform Cost Search
Runtime: 44.7925ms
Visited Nodes: 3262
Path:
1. four
2. dour
3. dorr
4. dore
5. dire
6. dine
7. nine

==========RESULT==========
Start Word: four
End Word: nine
Algorithm: Greedy Best First Search
Runtime: 0.3691ms
Visited Nodes: 3
Path:
Not found

==========RESULT==========
Start Word: four

```
End Word: nine
Algorithm: A*
Runtime: 15.0747ms
Visited Nodes: 982
Path:
1. four
2. tour
3. torr
4. tore
5. tone
6. none
7. nine
```

6. **chord-piano**

```
==========RESULT==========
Start Word: chord
End Word: piano
Algorithm: Uniform Cost Search
Runtime: 46.8712ms
Visited Nodes: 2640
Path:
1. chord
2. chard
3. chars
4. chaws
5. claws
6. clans
7. plans
8. pians
9. piano
```

```
==========RESULT==========
Start Word: chord
End Word: piano
Algorithm: Greedy Best First Search
Runtime: 0.1153ms
Visited Nodes: 5
Path:
Not found
```

```
==========RESULT==========
Start Word: chord
End Word: piano
Algorithm: A*
Runtime: 25.0267ms
Visited Nodes: 1331
```

Path:
1. chord
2. chard
3. chars
4. chaws
5. claws
6. clans
7. plans
8. pians
9. piano

3.3. Analisis Hasil

Dari hasil analisis, terlihat bahwa algoritma Greedy Best First Search (GBFS) memiliki kinerja yang paling baik dalam hal waktu eksekusi dan jumlah simpul yang dikunjungi. GBFS memerlukan waktu eksekusi yang singkat dan hanya mengunjungi sedikit simpul untuk menemukan solusi, hal ini dikarenakan algortitma BFS hanya mengaganti suatu currentNode dengan hasil ekspansi greedy tanpa perlu mengekspansi setiap kemungkinan node.

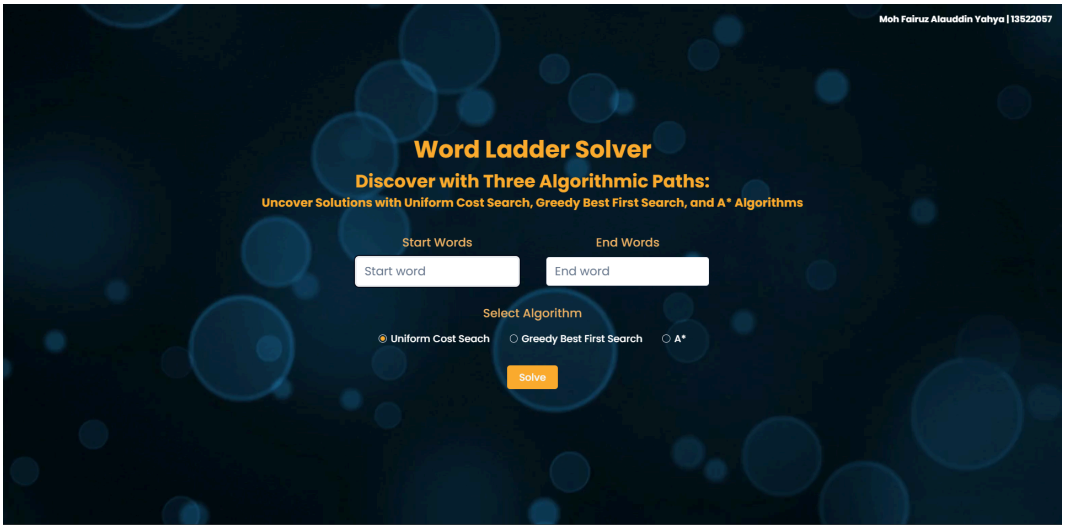
Algoritma A* juga menunjukkan hasil yang cukup baik, walaupun membutuhkan sedikit lebih banyak waktu eksekusi dibandingkan dengan GBFS dalam beberapa kasus. Namun, A* cenderung lebih stabil dalam menemukan solusi optimal. Dapat dilihat juga, bahwa hasil A* selalu sama dengan UCS yang membuktikan bahwa heuristik nilai yang digunakan benar, sehingga memberikan hasil yang optimal, namun dengan eksekusi yang lebih cepat dari pada UCS.

Sementara itu, algoritma Uniform Cost Search (UCS) memerlukan waktu eksekusi yang lebih lama dan mengunjungi lebih banyak simpul dibandingkan dengan GBFS dan A*. Hal ini disebabkan karena UCS tidak mempertimbangkan heuristik dalam pengambilan keputusan, sehingga cenderung menjelajahi lebih banyak simpul untuk mencapai tujuan, namun dapat dipastikan menghasilkan solusi yang optimal.

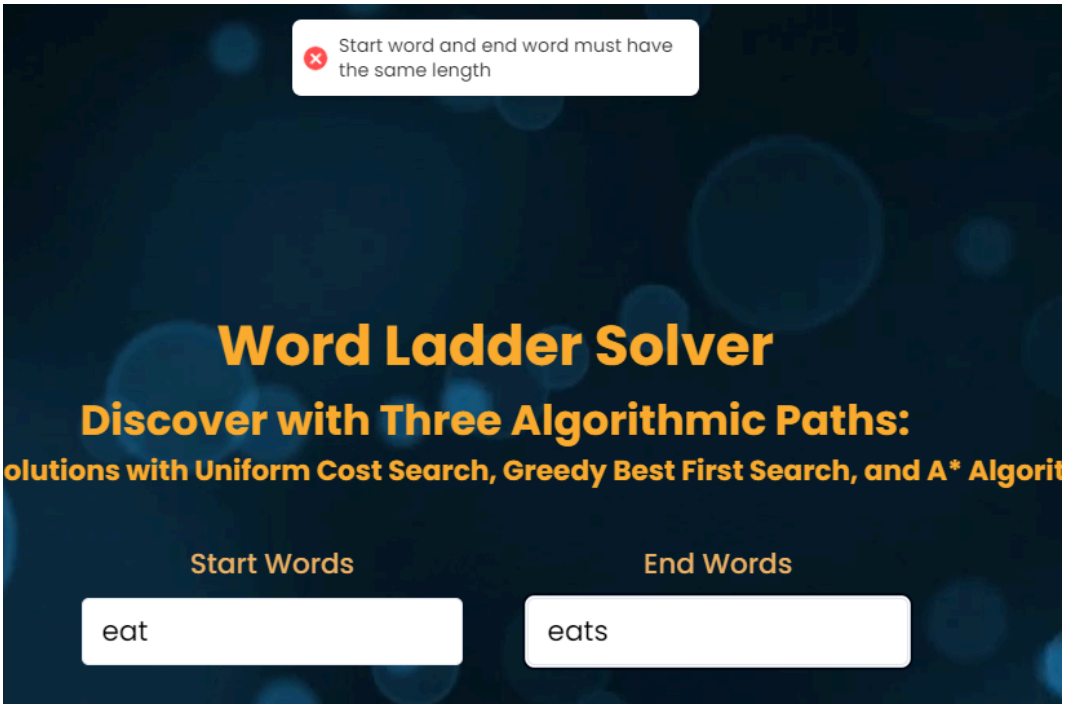
Dalam kategori *space complexity*, tentu GBFS adalah yang terbaik karena tidak perlu menyimpan PriorityQueue seperti 2 algoritma yang lain.

3.4. Implementasi Bonus

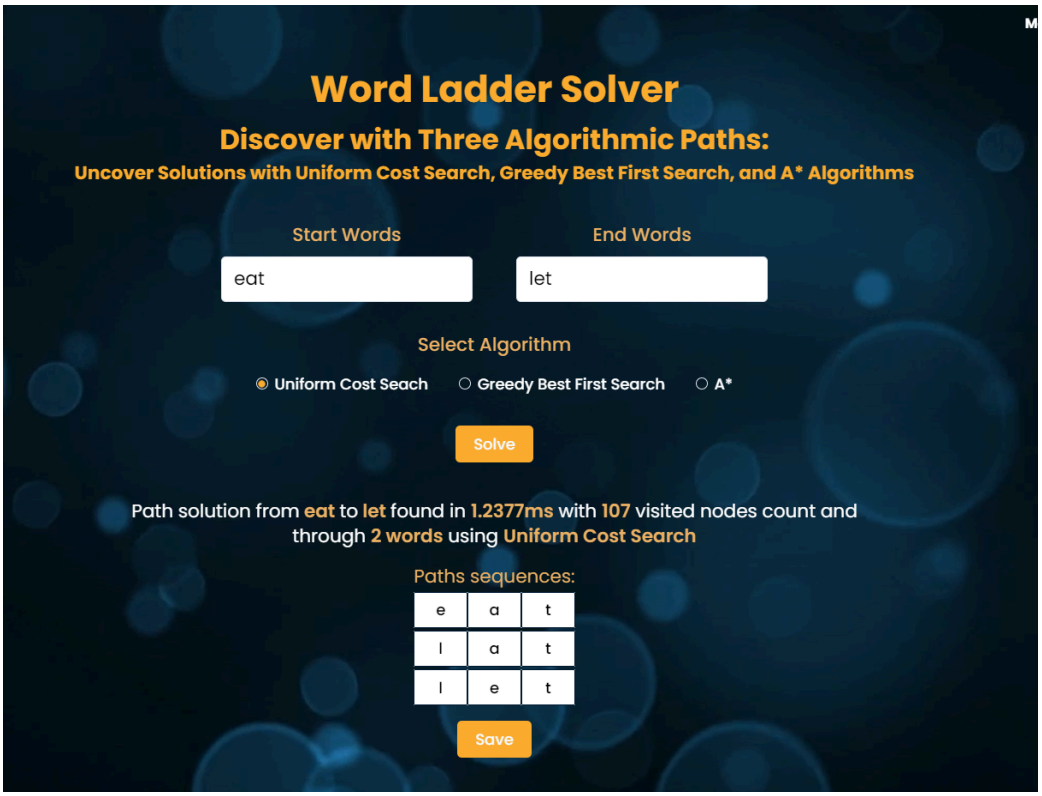
Saya mengimplementasikan bonus berupa antarmuka grafis (GUI) menggunakan Next.js dan Tailwind sebagai frontend GUI. GUI ini mencakup fitur input teks untuk kata awal dan akhir, validasi toast, tombol radio, dan komponen tombol lainnya. Selain itu, GUI juga menampilkan kemungkinan solusi dan informasi terkait, seperti waktu eksekusi, jumlah node yang dikunjungi, dan jumlah jalur yang tersedia, bersama dengan algoritma yang digunakan. Saya juga menambahkan fitur agar pengguna dapat menyimpan hasil solver sebagai file teks (txt) yang meminta masukan dari pengguna untuk nama file.



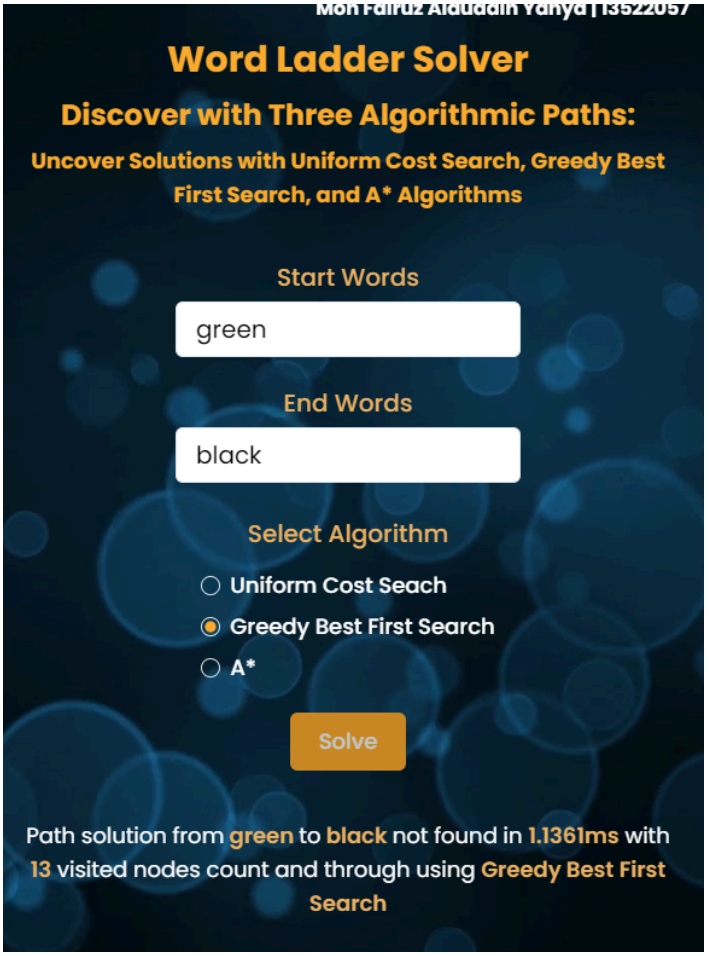
Gambar 3.4.1 Tampilan Awal GUI



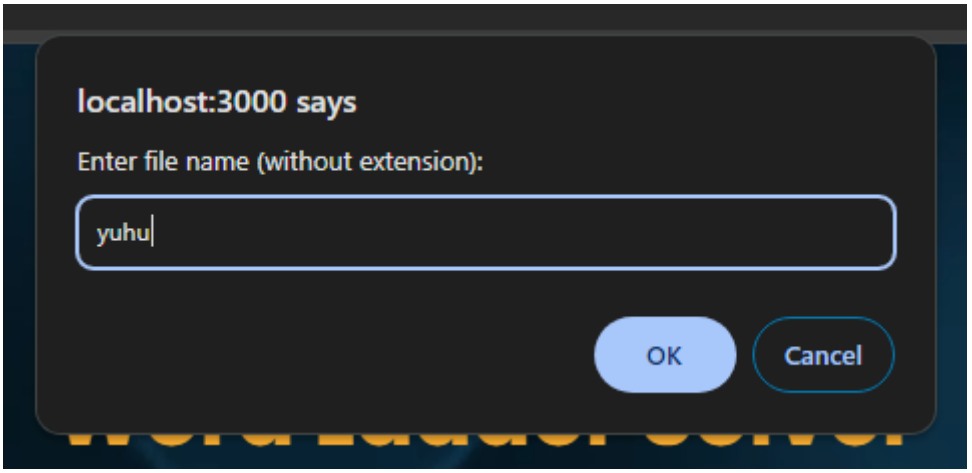
Gambar 3.4.2 Contoh Incorrect Test Case



Gambar 3.4.3 Contoh Test Case Bersolusi



Gambar 3.4.4 Contoh Responsive No Solution TestCase



Gambar 3.4.5 Saving Feature

BAB IV
KESIMPULAN

Kesimpulan dari hasil percobaan dan analisis membuktikan bahwa algoritma A* adalah algortima yang lebih baik daripada UCS dan Greedy Best First Search. Hal ini dipertimbangkan dari hasil yang didapatkan dan waktu eksekusi terbaik.

DAFTAR PUSTAKA

1. Munir, Rinaldi. “Penentuan Rute (Route/Path Planning) Bahan Kuliah IF2211 Strategi Algoritma Bagian 1: BFS, DFS, UCS, Greedy Best First Search” (online). (https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf, diakses pada 5 Meil 2024).
2. Munir, Rinaldi. “Penentuan Rute (Route/Path Planning) Bahan Kuliah IF2211 Strategi Algoritma Bagian 2: Algoritma A* 2” (online). (https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf, diakses pada 5 Meil 2024).

LAMPIRAN

Repository

Link Repository dari Tugas Besar 03 IF2211 Strategi Algoritma adalah sebagai berikut.

https://github.com/fairuzald/Tucil3_13522057

Checklist

No.	Poin	Ya	Tidak
1.	Program berhasil dijalankan.	✔	
2.	Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma UCS.	✔	
3.	Solusi yang diberikan pada algoritma UCS optimal.	✔	
4.	Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma Greedy Best First Search. (Parsial, karena algoritma tersebut non-complete)	✔	
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.	✔	