TUGAS KECIL 3

IF2211 STRATEGI ALGORITMA

SEMESTER II TAHUN 2023/2024

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

OLEH:

Ahmad Hasan Albana 13522041

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2024

KATA PENGANTAR

Puji syukur kita panjatkan ke hadirat Allah SWT, karena atas rahmat dan karunia-Nya, kami dapat menyelesaikan makalah ini dengan judul "Penyelesaian Permainan Word Ladder Menggunakan Algoritma UCS, Greedy Best First Search, dan A*". Makalah ini disusun sebagai salah satu tugas mata kuliah Strategi Algoritma.

Penyusunan makalah ini tidak lepas dari bantuan berbagai pihak. Kami mengucapkan terima kasih kepada dosen mata kuliah Strategi Algoritma yang telah memberikan bimbingan dan panduan selama proses pembuatan makalah ini.

Semoga makalah ini dapat memberikan kontribusi positif dalam pemahaman dan pengembangan suatu aplikasi dari konsep UCS, Greedy Best First Search, dan A*. Akhir kata, kami menyampaikan permohonan maaf atas segala keterbatasan dalam makalah ini, dan kami menerima dengan terbuka segala kritik dan saran yang bersifat membangun.

Bandung, 7 Mei 2024,

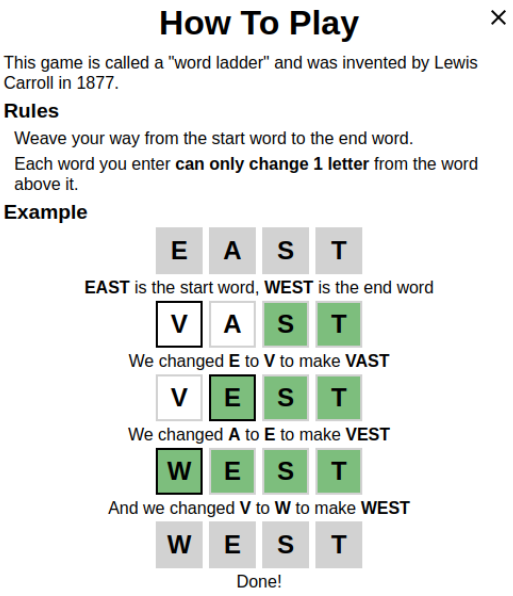
Ahmad Hasan Albana (13522041)

DAFTAR ISI

BAB I
DESKRIPSI MASALAH

1.1. Abstraksi

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 1.1 Ilustrasi dan Peraturan Permainan Word Ladder
(Sumber: https://wordwormdormdork.com/)

Pada Tugas Kecil 3 ini, penulis merancang program untuk menerapkan algoritma UCS, Greedy Best First Search, dan A* yang telah dipelajari di kelas untuk mencari solusi permainan WordLadder..

Program ini dibuat menggunakan bahasa Java dan dijalankan melalui CLI, serta menerima input berupa:

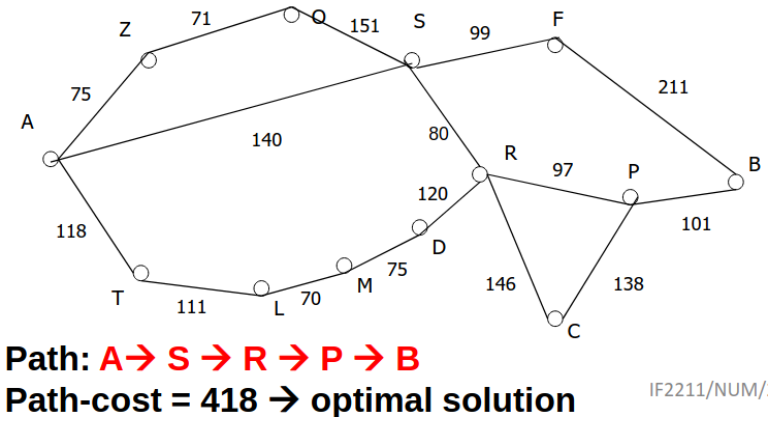
1) File list kata yang dijadikan acuan ‘kata valid’. Secara default akan menggunakan file ‘defaultDict.txt’.
2) Kata awal dan kata akhir pada permainan.
3) Algoritma yang ingin digunakan. (UCS, GBFS, A*)

Setelah menerima input, program akan melakukan pencarian path solusi, lalu menampilkan path solusi tersebut beserta waktu eksekusi dan path yang dikunjungi.

BAB II
IMPLEMENTASI ALGORITMA

2.1. Algoritma *Uniform Cost Search* (UCS)

Algoritma *uniform cost search* adalah salah satu pendekatan algoritma *blind search* untuk mencari sebuah rute dari simpul awal ke simpul akhir dengan *cost* terkecil. Algoritma pencarian rute ini dimulai dengan mengunjungi simpul awal yang memiliki *cost* senilai 0 dan akan dikunjungi juga semua simpul yang bertetangga dengan simpul awal yang memiliki nilai *cost* tertentu. Lalu, akan dipilih simpul hidup dengan *cost* terkecil dan setiap simpul yang bertetangga dengan simpul yang hidup tersebut dengan nilai *cost* sama dengan (*cost* simpul parent + *cost* dari parent ke simpul tersebut), akan dikunjungi dan simpul *parent* mereka akan dimatikan. Hal tersebut akan terus dilakukan hingga telah dikunjungi simpul akhir dan tidak ada simpul dengan *cost* yang lebih kecil dari simpul akhir tersebut atau seluruh simpul telah mati tanpa pernah mengunjungi simpul akhir yang berarti tidak ditemukannya solusi.



Gambar 2.1 Algoritma *Uniform Cost Search*
(Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf)

Algoritma ini memiliki keunggulan pada kepastian optimal dari *path* solusi yang dihasilkan. Hal tersebut dikarenakan algoritma ini menggunakan $g(n)$ atau nilai *cost* sebenarnya dari simpul awal ke simpul sekarang sebagai nilai pertimbangan untuk memilih simpul yang akan dikunjungi.

Pada program ini, nilai *cost* antar simpul yang bertetanggaan adalah 1. Penetapan nilai *cost* tersebut didasarkan pada definisi solusi optimal pada WordLadder adalah solusi dengan banyak simpul yang dilalui paling sedikit, sehingga setiap pengunjungan simpul bernilai sama.

2.2. Algoritma *Greedy Best First Search* (GBFS)

Algoritma *greedy best first search* (GBFS) adalah salah satu pendekatan algoritma *heuristic search* untuk mencari sebuah rute dari simpul awal ke simpul akhir dengan *cost* terkecil. Algoritma pencarian rute ini dimulai dengan mengunjungi simpul awal yang memiliki *cost* tertentu dan akan dikunjungi juga semua simpul yang bertetangga dengan simpul awal yang memiliki nilai *cost* tertentu. Lalu, akan dipilih simpul hidup dengan *cost* terkecil dan setiap simpul yang bertetangga dengan simpul yang hidup tersebut akan dikunjungi dan simpul *parent* mereka akan dimatikan. Hal tersebut akan terus dilakukan hingga telah dikunjungi simpul akhir dan tidak ada simpul dengan *cost* yang lebih kecil dari simpul akhir tersebut atau seluruh simpul telah mati tanpa pernah mengunjungi simpul akhir yang berarti tidak ditemukannya solusi.

Algoritma ini memiliki keunggulan pada kecepatan atau kompleksitas pencarian *path* solusi yang efektif. Hal tersebut dikarenakan algoritma ini menggunakan $h(n)$ atau nilai dari perkiraan *cost* yang diperlukan untuk menuju ke simpul akhir sebagai nilai pertimbangan untuk memilih simpul yang akan dikunjungi. Oleh karena itu, saat algoritma ini telah mengunjungi simpul akhir, maka pencarian pasti akan berhenti karena perkiraan *cost* menuju simpul itu sendiri pasti bernilai 0 yang merupakan nilai paling kecil yang mungkin. Hal tersebut juga akan mengakibatkan tidak terjaminnya bahwa solusi akan selalu optimal.

Pada program ini, nilai *cost* yang digunakan pada setiap simpul adalah banyaknya perbedaan huruf pada kata di simpul tersebut terhadap kata di simpul akhir. Penetapan nilai *cost* tersebut didasarkan pada langkah paling minimum yang dilakukan untuk menuju simpul akhir adalah seminimal mungkin sesuai

banyaknya huruf yang berbeda dengan simpul akhir karena aturan WordLadder hanya memungkinkan mengubah 1 huruf setiap langkahnya.

2.3. Algoritma A*

Algoritma A* adalah salah satu pendekatan algoritma *heuristic search* untuk mencari sebuah rute dari simpul awal ke simpul akhir dengan *cost* terkecil. Algoritma pencarian rute ini dimulai dengan mengunjungi simpul awal yang memiliki *cost* tertentu dan akan dikunjungi juga semua simpul yang bertetangga dengan simpul awal yang memiliki nilai *cost* tertentu. Lalu, akan dipilih simpul hidup dengan *cost* terkecil dan setiap simpul yang bertetangga dengan simpul yang hidup tersebut akan dikunjungi dan simpul *parent* mereka akan dimatikan. Hal tersebut akan terus dilakukan hingga telah dikunjungi simpul akhir dan tidak ada simpul dengan *cost* yang lebih kecil dari simpul akhir tersebut atau seluruh simpul telah mati tanpa pernah mengunjungi simpul akhir yang berarti tidak ditemukannya solusi.

Algoritma ini memiliki keunggulan pada kecepatan atau kompleksitas pencarian *path* solusi yang efektif sekaligus juga solusinya yang optimal. Hal tersebut dikarenakan algoritma ini menggunakan $f(n)$ atau nilai dari perkiraan *cost* yang diperlukan untuk menuju ke simpul akhir ditambah dengan nilai *cost* sebenarnya dari simpul awal ke simpul tersebut sebagai nilai pertimbangan untuk memilih simpul yang akan dikunjungi. Oleh karena itu, algoritma ini adalah campuran dari algoritma UCS dan GBFS sehingga memiliki keoptimalan solusi dari algoritma UCS dan kecepatan pencarian dari algoritma GBFS. Akan tetapi, tentu saja algoritma ini tak sepenuhnya memiliki keuntungan dari kedua algoritma tersebut. Algoritma ini akan tergantung dari dominasi nilai *cost* yang digunakan. Nilai *cost* sebenarnya yang jauh lebih besar dari nilai perkiraan *cost* akan menghasilkan solusi yang optimal tetapi tidak cukup cepat. Begitu pun sebaliknya yang akan mengakibatkan pencarian menjadi cepat tetapi solusi tidak optimal.

Pada program ini, nilai *cost* antar simpul yang bertetanggaan adalah penjumlahan *cost* pada UCS dengan *cost* pada GBFS. Penetapan nilai *cost* tersebut didasarkan pada definisi *cost* pada algoritma A* pada umumnya.

BAB III
IMPLEMENTASI KODE PROGRAM

3.1 Implementasi Kode

```
static class Node implements Comparable<Node> {
    String word;
    int cost;

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

    @Override
    public int compareTo(Node node) {
        if (cost < node.cost) { return -1; }
        else if (cost > node.cost) { return 1; }
        return 0;
    }
}

public static Map<String, List<String>> neighborMaps = new HashMap<>();
public static Map<String, Boolean> isChecked = new HashMap<>();
public static int visitedNode = 0;
```

Pada program ini, kata awal dijadikan sebagai simpul awal dan kata akhir dijadikan sebagai simpul akhir. Ketetanggaan antar simpul menandakan bahwa kedua simpul tersebut merupakan kata yang hanya memiliki beda 1 huruf. Simpul yang diimplementasikan pada program ini menyimpan kata dan juga *cost* dari simpul tersebut.

Variabel neighborMaps adalah variabel yang menyimpan informasi ketetanggaan dari suatu simpul. Variabel isChecked adalah variabel yang digunakan apakah simpul sudah pernah dikunjungi agar tidak terjadi pengunjungan simpul yang berulang-ulang. Variabel visitedNode adalah variabel yang digunakan untuk menghitung jumlah node yang telah dikunjungi.

```java
private static int totalDiff(String word1, String word2) {
    if (word1.length() != word2.length())
        return -1;
    int diffCount = 0;
    for (int i = 0; i < word1.length(); i++) {
        if (word1.charAt(i) != word2.charAt(i))
            diffCount++;
    }
    return diffCount;
}

private static void SaveDict(String pathDict) {
    FileWriter writer;
    try {
        writer = new FileWriter(System.getProperty("user.dir") + "/../test/formatted/" + pathDict);
        for (Map.Entry<String, List<String>> entry : neighborMaps.entrySet()) {
            writer.write(entry.getKey()+"\n");

            List<String> values = entry.getValue();
            for (String value: values) {
                writer.write(value + "\n");
            }

            writer.write(";\n");
        }
        writer.close();
    } catch (IOException e) {
        return;
    }
}

public static void Reset() {
    for (Map.Entry<String, Boolean> entry : isChecked.entrySet()) {
        isChecked.put(entry.getKey(), false);
    }
    visitedNode = 0;
}

private static boolean isWord(String word) {
    return neighborMaps.containsKey(word);
}
```

Program ini memiliki beberapa fungsi bantuan seperti totalDiff yang akan mengembalikan nilai banyaknya huruf yang berbeda antar kedua kata, isWord untuk memeriksa apakah suatu kata valid atau terdapat dalam dictionary, Reset untuk set ulang global variabel yang ada sehingga dapat digunakan unlang untuk mencari solusi kata lain, dan SaveDict untuk menyimpan hasil mapping dictionary pada neighborMap ke dalam file untuk mempercepat proses mapping ulang dictionary.

```java
public static void neighborInitializer(String pathDict) {
    List<String> words;
    boolean isFormatted = false;
    String path = System.getProperty("user.dir") + "/../test/";
    if (Files.exists(Paths.get(path + "formatted/" + pathDict))) {
        pathDict = "formatted/" + pathDict;
        isFormatted = true;
    }

    try {
        words = Files.readAllLines(Paths.get(System.getProperty("user.dir") + "/../test/" + pathDict));
    } catch (IOException e) {
        return;
    }

    String currWord = "";
    if (isFormatted) {
        for (String word : words) {
            isChecked.put(word, false);
            if (currWord.equals("")) {
                currWord = word;
                neighborMaps.put(currWord, new ArrayList<>());
            } else if (word.equals("END")) {
                currWord = "";
            } else {
                neighborMaps.get(currWord).add(word);
            }
        }
    } else {
        for (String word : words) {
            isChecked.put(word, false);
            neighborMaps.put(word, new ArrayList<>());
        }

        int n = words.size();

        for (int i = 0; i < n-1; i++) {
            String word1 = words.get(i);
            System.out.println(word1);
            for (int j = i+1; j < n; j++) {
                String word2 = words.get(j);

                if (totalDiff(word1, word2) == 1) {
                    neighborMaps.get(word1).add(word2);
                    neighborMaps.get(word2).add(word1);
                }
            }
        }
        SaveDict(pathDict);
    }
}
```

Fungsi neighborInitializer adalah fungsi yang secara garis besar adalah memetakan ketetanggaan dari suatu kata dan menyimpan informasi tersebut di neighborMap. Fungsi ini juga akan membuat dictionary baru yang memiliki format kata pada baris paling awal adalah key dari map, dan kata-kata dibawahnya sampai dengan ‘END’ adalah tetangga dari key. Directory baru tersebut akan disimpan pada folder formatted di dalam folder test tempat directory yang asli berada. Hal tersebut akan memudahkan dan mempercepat proses inisialisasi variabel neighborMap karena tidak perlu lagi membandingkan setiap 2 kata pada dictionary yang memiliki kompleksitas $O(n^2)$.

```java
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    String pathFile;

    System.out.print("Insert DictPath: ");
    pathFile = scanner.nextLine();
    if (pathFile.equals("")) { pathFile = "defaultDict.txt"; }
    System.out.println("Initializing dictionary...");
    neighborInitializer(pathFile);

    while (neighborMaps.isEmpty()) {
        System.out.println("Invalid Directory Path! Make sure the file is in 'test' folder.");
        System.out.print("Insert DictPath: ");
        pathFile = scanner.nextLine();
        if (pathFile.equals("")) { pathFile = "defaultDict.txt"; }
        neighborInitializer(pathFile);
    }

    boolean isLoop = true;
    while (isLoop) {
        System.out.print("Insert Root Word: ");
        String start = scanner.nextLine();
        while (!isWord(start)) {
            System.out.println("Invalid English Word!");
            System.out.print("Insert Root Word: ");
            start = scanner.nextLine();
        }
        System.out.print("Insert Target Word: ");
        String end = scanner.nextLine();
        while (!isWord(end)) {
            System.out.println("Invalid English Word!");
            System.out.print("Insert Target Word: ");
            start = scanner.nextLine();
        }

        boolean isValid = false;
        String mode = "";
        while (!isValid) {
            System.out.print("Insert Algorithm to Use (UCS / GBFS / AStar): ");
            mode = scanner.nextLine();
            if (mode.equals("UCS") || mode.equals("GBFS") || mode.equals("AStar")) { isValid = true; }
        }

        long startTime = System.currentTimeMillis();
        List<String> result = Solver(start, end, mode);
        long endTime = System.currentTimeMillis();
        long duration = endTime - startTime;

        System.out.println("Algoritma " + mode);
        System.out.print("Path: ");
        System.out.println(result);
        System.out.println("Waktu eksekusi: " + duration + " ms");
        System.out.println("Banyak node yang dikunjungi: " + visitedNode);

        System.out.print("Wanna solve another word? (y/n): ");
        String input = scanner.nextLine();
        if (input.equals("n")) { isLoop = false; } else { Reset(); }
    }

    scanner.close();
}
```

Pada fungsi Main, dilakukan seluruh pengambilan input dari user, mulai dari directory dictionary, kata awal, kata akhir, dan juga algoritma yang ingin digunakan. Untuk directory dari dictionary, secara default akan menggunakan ‘defaultDict.txt’. Lalu setiap input juga dilakukan validasi seperti directory, input kata, dan juga jenis algoritma yang digunakan.

```java
public static List<String> Solver(String start, String end, String algo) {
    PriorityQueue<Node> priorityQueue = new PriorityQueue<>();
    Map<String, Integer> costMap = new HashMap<>();
    Map<String, String> parentMap = new HashMap<>();

    int newCost = 0;
    if (!algo.equals("UCS")) { newCost += totalDiff(start, end); }
    priorityQueue.offer(new Node(start, newCost));
    costMap.put(start, newCost);
    parentMap.put(start, null);
    isChecked.put(start, true);

    while (!priorityQueue.isEmpty()) {
        visitedNode++;
        Node current = priorityQueue.poll();

        if (current.word.equals(end)) {
            // Reconstruct path
            List<String> path = new ArrayList<>();
            String word = end;
            while (word != null) {
                path.add(word);
                word = parentMap.get(word);
            }
            Collections.reverse(path);
            return path;
        }

        List<String> wordList = neighborMaps.get(current.word);

        for (String neighbor : wordList) {
            if (!isChecked.get(neighbor)) {
                isChecked.put(neighbor, true);
                if (algo.equals("UCS")) { newCost = costMap.get(current.word) + 1; }
                else if (algo.equals("GBFS")) { newCost = totalDiff(neighbor, end); }
                else /*AStar*/ { newCost = totalDiff(neighbor, end) + costMap.get(current.word) + 1; }
                costMap.put(neighbor, newCost);
                priorityQueue.offer(new Node(neighbor, newCost));
                parentMap.put(neighbor, current.word);
            }
        }
    }

    return null; // No path found
}
```

Fungsi solver ini adalah fungsi utama yang menerapkan algoritma UCS, GBFS, dan A*. Fungsi ini menggunakan priorityQueue untuk menyimpan simpul-simpul yang masih hidup dikarenakan pemilihan simpul memprioritaskan simpul dengan *cost* yang paling kecil sehingga cocok menggunakan priorityQueue. Lalu, ada variabel costMap untuk menyimpan nilai *cost* pada setiap simpul dan parentMap untuk menyimpan parent dari suatu simpul pada solusi.

Program dimulai dengan mengiterasi seluruh tetangga dari simpul awal dan menyimpannya ke dalam priorityQueue sebagai simpul hidup. Lalu akan lanjut diiterasi setiap tetangga dari setiap simpul yang hidup dengan prioritas *cost* terkecil. Hingga pada saat simpul akhir telah dikunjungi, maka akan langsung mengembalikan list dari path solusi yang ditemukan. Untuk *cost* yang digunakan pada setiap simpul sesuai dengan penjelasan pada Bab II.

BAB IV
HASIL PENGUJIAN DAN ANALISIS

4.1 Hasil Pengujian

Berikut adalah hasil pengujian dengan menggunakan dictionary yang berada pada folder *test.*

TEST1: CHARGE - COMEDO

```
Insert Root Word: charge
Insert Target Word: comedo
Insert Algorithm to Use (UCS / GBFS / AStar): UCS
Algoritma UCS
Path: [charge, charre, charrs, chirrs, shirrs, shiers, shyers, sayers, payers, pape
rs, papery, popery, popely, pomely, comely, comedy, comedo]
Waktu eksekusi: 18 ms
Banyak node yang dikunjungi: 16351
Wanna solve another word? (y/n): y
Insert Root Word: charge
Insert Target Word: comedo
Insert Algorithm to Use (UCS / GBFS / AStar): GBFS
Algoritma GBFS
Path: [charge, charre, charro, charco, charca, charka, charks, chards, chords, chor
es, chokes, choker, cooker, cooked, cooeed, cooees, cooers, copers, mopers, mopery,
 popery, popely, pomely, comely, comedy, comedo]
Waktu eksekusi: 2 ms
Banyak node yang dikunjungi: 278
Wanna solve another word? (y/n): y
Insert Root Word: charge
Insert Target Word: comedo
Insert Algorithm to Use (UCS / GBFS / AStar): AStar
Algoritma AStar
Path: [charge, charre, charrs, chirrs, shirrs, shiers, shyers, sayers, payers, pape
rs, papery, popery, popely, pomely, comely, comedy, comedo]
Waktu eksekusi: 15 ms
Banyak node yang dikunjungi: 12193
```

TEST2: CAT - DOG

```
Insert Root Word: cat
Insert Target Word: dog
Insert Algorithm to Use (UCS / GBFS / AStar): UCS
Algoritma UCS
Path: [cat, cag, dag, dog]
Waktu eksekusi: 3 ms
Banyak node yang dikunjungi: 1256
Wanna solve another word? (y/n): y
Insert Root Word: cat
Insert Target Word: dog
Insert Algorithm to Use (UCS / GBFS / AStar): GBFS
Algoritma GBFS
Path: [cat, cag, cog, dog]
Waktu eksekusi: 0 ms
Banyak node yang dikunjungi: 4
Wanna solve another word? (y/n): y
Insert Root Word: cat
Insert Target Word: dog
Insert Algorithm to Use (UCS / GBFS / AStar): AStar
Algoritma AStar
Path: [cat, cot, dot, dog]
Waktu eksekusi: 1 ms
Banyak node yang dikunjungi: 109
```

TEST3: ATLASES - CABARET

```
Insert Root Word: atlases
Insert Target Word: cabaret
Insert Algorithm to Use (UCS / GBFS / AStar): UCS
Algoritma UCS
Path: [atlases, anlases, anlaces, unlaces, unlaced, unraced, unraped, uncaped, unca
sed, uncases, uneases, ureases, creases, creased, creaked, croaked, crocked, chocke
d, shocked, stocked, stooked, stroked, striked, strikes, shrikes, shrines, serines,
 serenes, serener, sevener, severer, severed, levered, lovered, hovered, havered, w
avered, watered, catered, capered, tapered, tabered, tabored, taboret, tabaret, cab
aret]
Waktu eksekusi: 26 ms
Banyak node yang dikunjungi: 12422
Wanna solve another word? (y/n): y
Insert Root Word: atlases
Insert Target Word: cabaret
Insert Algorithm to Use (UCS / GBFS / AStar): GBFS
Algoritma GBFS
Path: [atlases, anlases, anlaces, unlaces, unlaced, unraced, unraked, unbaked, unba
sed, uncased, uncases, uneases, ureases, creases, creased, creaked, croaked, cloake
d, clonked, clinked, clinker, chinker, chinner, channer, chanter, chunter, counter,
 coulter, coulier, collier, collies, coolies, coories, corries, carries, parries, p
arties, panties, pandies, candies, candles, cantles, cantlet, mantlet, martlet, war
tlet, warblet, warbles, wabbles, gabbles, gabbler, gabeler, gaveler, gaveled, ravel
ed, ravened, havened, havered, wavered, watered, catered, capered, tapered, tabered
, tabored, taboret, tabaret, cabaret]
Waktu eksekusi: 5 ms
Banyak node yang dikunjungi: 1187
Wanna solve another word? (y/n): y
Insert Root Word: atlases
Insert Target Word: cabaret
Insert Algorithm to Use (UCS / GBFS / AStar): AStar
Algoritma AStar
Path: [atlases, anlases, anlaces, unlaces, unlaced, unraced, unraped, uncaped, unca
sed, uncases, uneases, ureases, creases, creaser, creaker, croaker, crocker, clocke
r, slocker, stocker, stooker, stroker, striker, strikes, shrikes, shrines, serines,
 serenes, serener, sevener, severer, severed, levered, lovered, hovered, havered, w
avered, watered, catered, capered, tapered, tabered, tabored, taboret, tabaret, cab
aret]
Waktu eksekusi: 17 ms
Banyak node yang dikunjungi: 11605
```

TEST4: PLANTS - ZOMBIE

```
Insert Root Word: plants
Insert Target Word: zombie
Insert Algorithm to Use (UCS / GBFS / AStar): UCS
Algoritma UCS
Path: null
Waktu eksekusi: 28 ms
Banyak node yang dikunjungi: 20043
Wanna solve another word? (y/n): y
Insert Root Word: plants
Insert Target Word: zombie
Insert Algorithm to Use (UCS / GBFS / AStar): GBFS
Algoritma GBFS
Path: null
Waktu eksekusi: 24 ms
Banyak node yang dikunjungi: 20043
Wanna solve another word? (y/n): y
Insert Root Word: plants
Insert Target Word: zombie
Insert Algorithm to Use (UCS / GBFS / AStar): AStar
Algoritma AStar
Path: null
Waktu eksekusi: 22 ms
Banyak node yang dikunjungi: 20043
```

TEST5: HORSE - STEAK

```
Insert Root Word: horse
Insert Target Word: steak
Insert Algorithm to Use (UCS / GBFS / AStar): UCS
Algoritma UCS
Path: [horse, corse, coree, soree, stree, strep, steep, steek, steak]
Waktu eksekusi: 20 ms
Banyak node yang dikunjungi: 11012
Wanna solve another word? (y/n): y
Insert Root Word: horse
Insert Target Word: steak
Insert Algorithm to Use (UCS / GBFS / AStar): GBFS
Algoritma GBFS
Path: [horse, horae, horal, soral, seral, seraw, straw, strad, stead, steak]
Waktu eksekusi: 1 ms
Banyak node yang dikunjungi: 29
Wanna solve another word? (y/n): y
Insert Root Word: horse
Insert Target Word: steak
Insert Algorithm to Use (UCS / GBFS / AStar): AStar
Algoritma AStar
Path: [horse, dorse, doree, soree, stree, strae, strad, stead, steak]
Waktu eksekusi: 9 ms
Banyak node yang dikunjungi: 2841
```

TEST6: EARN - MAKE

```
Insert Root Word: earn
Insert Target Word: make
Insert Algorithm to Use (UCS / GBFS / AStar): UCS
Algoritma UCS
Path: [earn, warn, ware, wake, make]
Waktu eksekusi: 11 ms
Banyak node yang dikunjungi: 1578
Wanna solve another word? (y/n): y
Insert Root Word: earn
Insert Target Word: make
Insert Algorithm to Use (UCS / GBFS / AStar): GBFS
Algoritma GBFS
Path: [earn, barn, bare, bake, make]
Waktu eksekusi: 0 ms
Banyak node yang dikunjungi: 5
Wanna solve another word? (y/n): y
Insert Root Word: earn
Insert Target Word: make
Insert Algorithm to Use (UCS / GBFS / AStar): AStar
Algoritma AStar
Path: [earn, carn, care, cake, make]
Waktu eksekusi: 6 ms
Banyak node yang dikunjungi: 232
```

4.2 Pembahasan Hasil Pengujian

Berdasarkan pada hasil pengujian di atas, dapat terlihat bahwa algoritma GBFS selalu lebih cepat daripada kedua algoritma lainnya. Karena berdasarkan

penjelasan pada Bab II, diketahui bahwa algoritma GBFS memang akan langsung menghentikan pencarian saat telah ditemukannya solusi. Oleh karena itu juga, total node yang dikunjungi algoritma GBFS ini juga lebih sedikit. Akan tetapi, saat tidak ditemukannya solusi, mengakibatkan algoritma ini pasti mengecek jumlah node yang sama dengan kedua algoritma lainnya seperti yang terlihat pada TEST4. Berdasarkan hal tersebut juga, dapat disimpulkan penggunaan memori pada algoritma ini adalah paling sedikit karena tidak perlu menyimpan banyak informasi node, sedangkan penggunaan memori paling besar adalah algoritma UCS.

Selain itu, terlihat juga bahwa algoritma GBFS terkadang tidak memiliki solusi paling optimal seperti pada TEST1, TEST3, dan TEST5. Hal ini sesuai dengan penjelasan pada Bab II sebelumnya. Sedangkan algoritma A* pada kasus ini selalu menunjukkan hasil yang optimal bahkan pada test case yang panjang seperti TEST1, walaupun dominasi *cost* dari sisi GBFS lebih besar tetapi tidak signifikan. Sehingga pada kasus ini, algoritma A* lebih efektif dan efisien dibandingkan algoritma UCS yang waktu eksekusinya lebih lama.

Pada ketiga algoritma tersebut juga memliki solusi yang berbeda-beda, bahkan pada test case yang sederhara seperti TEST6. Hal tersebut menunjukkan bahwa algoritma A* tidak sepenuhnya mengikuti algoritma UCS ataupun algoritma GBFS, yang berarti juga kedua algoritma tersebut tidak ada yang terlalu mendominasi.

DAFTAR PUSTAKA

1. Munir, Rinaldi. “Penentuan Rute (Route/Path Planning) Bagian 1: BFS, DFS, UCS, Greedy Best First Search” (online).
(https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf, diakses pada 7 Mei 2024).
2. Munir, Rinaldi. “Penentuan Rute (Route/Path Planning) Bagian 2: Algoritma A*” (online).
(https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf, diakses pada 7 Mei 2024).

LAMPIRAN

Repository

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

https://github.com/Bana-man/Tucil3_13522041.git