LAPORAN TUGAS KECIL

PENYELESAIAN PERMAINAN WORD LADDER MENGGUNAKAN ALGORITMA UCS, GREEDY BEST FIRST SEARCH, DAN A*

IF2211 STRATEGI ALGORITMA

Disusun oleh :

Ikhwan Al Hakim (13522147)

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2023/2024

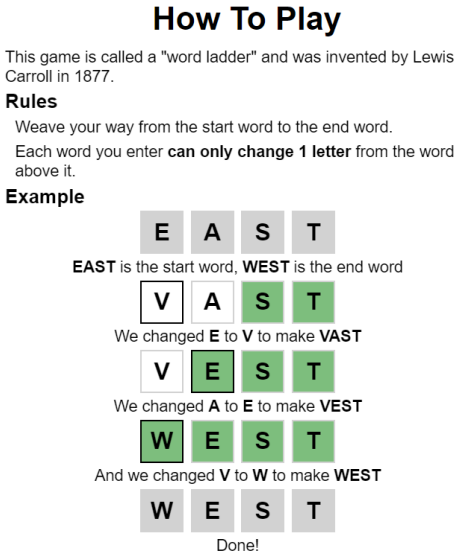
DAFTAR ISI

DAFTAR GAMBAR

BAB I
DESKRIPSI MASALAH

1.1. Tujuan

Tujuan dari pembuatan tugas kecil ini adalah mengimplementasikan algoritma UCS, *Greedy Best First Search,* dan A* untuk menyelesaikan permainan Word Ladder.



Gambar 1.1.1 Ilustrasi dan Cara Bermain Word Ladder

Word ladder adalah permainan teka-teki kata yang dimainkan dengan cara mengubah dari satu kata menjadi kata lainnya dengan mengubah satu huruf per suatu langkah. Setiap langkah harus menghasilkan kata baru yang valid, dan tujuannya adalah untuk mencapai kata target dalam jumlah langkah minimum. Permainan ini biasanya memiliki aturan yang ketat, seperti hanya boleh menggunakan kata-kata yang ada dalam kamus atau mengubah kata pada posisi yang sama dengan posisi huruf yang diubah pada langkah sebelumnya.

1.2. Spesifikasi

Terdapat beberapa spesifikasi yang harus diikuti pada pembuatan tugas kecil ini, diantaranya adalah:

1. Program dibuat dalam bahasa Java berbasis CLI

2. Kata-kata yang dimasukkan ke program harus berbahasa Inggris. Cara untuk melakukan validasi dibebaskan kepada mahasiswa
3. Tugas dikerjakan secara individu

BAB II
DASAR TEORI

2.1. Algoritma Uniform Cost Search

Algoritma Uniform Cost Search (UCS) adalah algoritma pencarian jalur yang menggunakan pendekatan "expand and select" untuk mencari jalur terpendek dari satu simpul ke simpul lain dalam graf berbobot. Proses ini dimulai dengan simpul awal, lalu mengembangkannya secara berurutan berdasarkan bobot lintasan terpendek dari simpul awal ke simpul yang sedang dieksplorasi.

Langkah pertama, "expand", melibatkan pengembangan simpul yang sedang dieksplorasi dengan menambahkan tetangganya yang belum dieksplorasi ke dalam himpunan simpul yang akan dieksplorasi selanjutnya. Langkah kedua, "select", memilih simpul yang akan dieksplorasi selanjutnya berdasarkan bobot lintasan terpendek dari simpul awal. Proses ini terus berlanjut hingga simpul tujuan ditemukan atau tidak ada simpul yang tersisa untuk dieksplorasi.

UCS memastikan bahwa simpul dengan bobot lintasan terendah dieksplorasi terlebih dahulu, sehingga jika jalur terpendek telah ditemukan, algoritma akan segera menghentikan pencarian. Hal ini membuat UCS efisien dalam mencari jalur terpendek dalam graf berbobot.

2.2. Algoritma Greedy Best First Search

Algoritma Greedy Best First Search adalah pendekatan pencarian yang mengutamakan simpul yang paling dekat dengan tujuan saat ini, tanpa mempertimbangkan apakah simpul tersebut merupakan pilihan terbaik secara keseluruhan. Algoritma ini beroperasi dengan cara memilih simpul yang memiliki nilai fungsi evaluasi terendah atau terkecil, yang merupakan estimasi dari jarak dari simpul tersebut ke tujuan. Proses ini terus berlanjut, dengan simpul yang dipilih digunakan sebagai simpul saat ini, dan langkah ini diulang sampai simpul tujuan ditemukan atau tidak ada simpul yang tersisa untuk dieksplorasi.

Algoritma Greedy Best First Search dapat diterapkan pada masalah pencarian seperti pencarian jalan terpendek dalam grafik. Dalam konteks ini, setiap simpul grafik

memiliki nilai fungsi evaluasi yang merupakan estimasi jarak ke simpul tujuan. Algoritma akan memilih simpul dengan nilai evaluasi terendah pada setiap langkah, dengan harapan bahwa simpul tersebut akan mendekatkan pencarian ke tujuan dengan cepat.

Algoritma ini sering digunakan dalam aplikasi yang membutuhkan pencarian jarak terpendek, seperti dalam navigasi GPS atau perencanaan rute dalam robotika. Namun, karena sifatnya yang "serakah" dalam pemilihan simpul, algoritma ini tidak selalu menghasilkan solusi optimal dan dapat terjebak dalam siklus jika tidak diterapkan dengan hati-hati.

2.3. Algoritma A*

Algoritma A* (A-star) adalah algoritma pencarian yang kombinasi antara pendekatan Greedy Best First Search dan UCS Algorithm. Algoritma ini menggabungkan pendekatan "serakah" dari Greedy Best First Search dengan informasi lengkap tentang biaya yang telah ditemukan dari titik awal melalui simpul yang sedang dipertimbangkan dari UCS Algorithm. Ini membuat A* dapat menemukan solusi yang optimal dan efisien dalam beberapa kasus.

Algoritma A* menggunakan dua nilai untuk setiap simpul yang dieksplorasi: nilai g(n), yang merupakan biaya terbaik yang diketahui untuk mencapai simpul tersebut dari titik awal, dan nilai h(n), yang merupakan estimasi biaya yang tersisa untuk mencapai tujuan dari simpul tersebut. Biaya total untuk mencapai simpul tersebut adalah f(n) = g(n) + h(n).

Langkah-langkah algoritma A* adalah pertama dilakukan inisialisasi open list dengan simpul awal dan mengatur nilai g(s) = 0. Setelah itu inisialisasi closed list kosong. Selama open list tidak kosong, maka akan dipilih simpul dengan nilai f(n) terkecil dari open list. Setelah itu simpul tersebut dipindahkan dari open list ke closed list. Jika simpul tersebut adalah simpul tujuan, maka solusi telah ditemukan dan program akan keluar dari algoritma. Untuk setiap simpul tetangga yang belum dieksplorasi, akan dihitung nilai f(n) untuk semua simpul tersebut. Jika simpul belum ada di open list, tambahkan simpul tersebut ke open list dan catat nilai f(n), g(n), dan h(n). Jika simpul sudah ada di open list dan nilai f(n) yang baru lebih kecil daripada yang sebelumnya,

update nilai f(n), g(n), dan h(n) untuk simpul tersebut. Jika open list kosong dan solusi belum ditemukan, maka tidak ada solusi yang mungkin.

Algoritma A* biasanya digunakan dalam berbagai aplikasi yang memerlukan pencarian jalur terpendek, seperti dalam perencanaan rute, navigasi robot, dan kecerdasan buatan. Algoritma ini memiliki keseimbangan yang baik antara optimasi dan efisiensi, membuatnya menjadi pilihan yang populer dalam pemrograman pencarian.

BAB III
PENERAPAN ALGORITMA

3.1. Algoritma Uniform Cost Search

Algoritma UCS menggunakan sebuah graf berbobot, di mana bobot setiap edge menunjukkan biaya untuk berpindah dari satu node ke node lainnya.

Pada implementasi ini, algoritma UCS dimulai dengan membuat node awal (beginNode) yang memiliki kata awal (beginWord) dan biaya 0. Kemudian, sebuah PriorityQueue bernama todo digunakan untuk menyimpan node-node yang akan dieksplorasi berdasarkan biaya terendah. Setiap node yang dieksplorasi akan dihapus dari todo.

Selama algoritma berjalan, setiap node yang dieksplorasi akan dicek apakah kata yang diwakilinya sama dengan endWord. Jika ya, maka jalur (path) dari node awal ke node ini akan dikembalikan sebagai hasil pencarian.

Selain itu, algoritma akan mencoba mengganti setiap karakter dalam kata saat ini dengan setiap karakter dari 'a' sampai 'z' untuk mencari kata-kata yang berbeda satu karakter dengan kata saat ini. Jika kata yang dihasilkan merupakan kata yang valid (ada dalam wordList), maka akan dibuat node baru dengan kata tersebut dan biaya yang diupdate. Node baru ini akan ditambahkan ke todo untuk dieksplorasi lebih lanjut. Jika tidak ada jalur yang ditemukan, maka akan dikembalikan sebuah ArrayList kosong sebagai hasil pencarian.

3.2. Algoritma Greedy Best First Search

Algoritma GBFS menggunakan heuristik untuk menentukan langkah selanjutnya dalam pencarian, tanpa mempertimbangkan biaya sejauh ini. Algoritma ini cenderung memilih jalur yang mengarah ke tujuan sesegera mungkin, meskipun tidak menjamin bahwa jalur tersebut adalah jalur terpendek.

Pada implementasi ini, algoritma GBFS dimulai dengan membuat node awal (beginNode) yang memiliki kata awal (beginWord). Kemudian, sebuah PriorityQueue bernama todo digunakan untuk menyimpan node-node yang akan dieksplorasi

berdasarkan heuristik, yaitu perbedaan antara kata saat ini dengan kata akhir (endWord). Semakin kecil perbedaan ini, semakin baik (lebih dekat) node tersebut.

Selama algoritma berjalan, setiap node yang dieksplorasi akan dicek apakah kata yang diwakilinya sama dengan endWord. Jika ya, maka jalur (path) dari node awal ke node ini akan dikembalikan sebagai hasil pencarian.

Selain itu, algoritma akan mencoba mengganti setiap karakter dalam kata saat ini dengan setiap karakter dari 'a' sampai 'z' untuk mencari kata-kata yang berbeda satu karakter dengan kata saat ini. Jika kata yang dihasilkan merupakan kata yang valid (ada dalam wordList), maka akan dibuat node baru dengan kata tersebut dan jalur yang diupdate. Node baru ini akan ditambahkan ke todo untuk dieksplorasi lebih lanjut. Jika tidak ada jalur yang ditemukan, maka akan dikembalikan sebuah ArrayList kosong sebagai hasil pencarian.

3.3. Algoritma A*

Algoritma A* menggabungkan heuristik dengan biaya sejauh ini untuk menemukan jalur terpendek dari titik awal ke titik akhir dalam sebuah graf berbobot. Heuristik yang digunakan dalam algoritma ini adalah jumlah karakter yang berbeda antara kata saat ini dengan kata akhir (endWord).

Pada implementasi ini, algoritma A* dimulai dengan membuat node awal (beginNode) yang memiliki kata awal (beginWord). Kemudian, sebuah PriorityQueue bernama todo digunakan untuk menyimpan node-node yang akan dieksplorasi berdasarkan nilai f(g) + h(g), di mana f(g) adalah biaya sejauh ini (cost) dan h(g) adalah heuristik dari node tersebut.

Selama algoritma berjalan, setiap node yang dieksplorasi akan dicek apakah kata yang diwakilinya sama dengan endWord. Jika ya, maka jalur (path) dari node awal ke node ini akan dikembalikan sebagai hasil pencarian.

Selain itu, algoritma akan mencoba mengganti setiap karakter dalam kata saat ini dengan setiap karakter dari 'a' sampai 'z' untuk mencari kata-kata yang berbeda satu karakter dengan kata saat ini. Jika kata yang dihasilkan merupakan kata yang valid (ada dalam wordList), maka akan dibuat node baru dengan kata tersebut, biaya yang diupdate, heuristik baru, dan jalur yang diupdate. Node baru ini akan ditambahkan ke todo untuk

dieksplorasi lebih lanjut. Jika tidak ada jalur yang ditemukan, maka akan dikembalikan sebuah ArrayList kosong sebagai hasil pencarian.

BAB IV
ANALISIS DAN IMPLEMENTASI

4.1. Implementasi Algoritma

Berikut adalah implementasi program dalam bahasa Java.

4.2.1. Algoritma Uniform Cost Search

```
import java.util.*;

class NodeUCS {
    String word;
    int cost;
    List<String> path;

    NodeUCS(String word, int cost) {
        this.word = word;
        this.cost = cost;
        this.path = new ArrayList<>();
    }

    NodeUCS(String word, int cost, List<String> path) {
        this.word = word;
        this.cost = cost;
        this.path = path;
    }
}
```

```
import java.util.*;

class UCS {
    static public List<String> algo(String beginWord, String endWord,
Map<String, Boolean> wordList, long[] visited) {
        NodeUCS beginNode = new NodeUCS(beginWord, 0);
        beginNode.path.add(beginWord);
```

```
        PriorityQueue<NodeUCS> todo = new PriorityQueue<>((a, b) ->
Integer.compare(a.cost, b.cost));
        todo.add(beginNode);
        while (!todo.isEmpty()) {
            NodeUCS current = todo.poll();
            visited[0]++;
            if (current.word.equals(endWord)) {
                return current.path;
            }
            wordList.remove(current.word);
            char[] charArray = current.word.toCharArray();
            for (int j = 0; j < charArray.length; j++) {
                char originalChar = charArray[j];
                for (char c = 'a'; c <= 'z'; c++) {
                    charArray[j] = c;
                    String newWord = new String(charArray);
                    if (wordList.containsKey(newWord)) {
                        List<String> tempPath = new
ArrayList<>(current.path);
                        tempPath.add(newWord);
                        int newCost = current.cost + 1;
                        NodeUCS newNode = new NodeUCS(newWord, newCost,
tempPath);
                        todo.add(newNode);
                        wordList.remove(newWord);
                    }
                }
                charArray[j] = originalChar;
            }
        }
        return new ArrayList<>();
    }
}
```

4.2.2. Algoritma Greedy Best First Search

```
import java.util.*;

class NodeGBFS {
    public String word;
    public List<String> path;

    public NodeGBFS(String word) {
        this.word = word;
        this.path = new ArrayList<>();
    }

    public NodeGBFS(String word, List<String> path) {
        this.word = word;
        this.path = path;
    }
}
```

```
import java.util.*;

class GBFS {
    static public List<String> algo(String beginWord, String endWord,
Map<String, Boolean> wordList, long[] visited) {
        NodeGBFS beginNode = new NodeGBFS(beginWord);
        beginNode.path.add(beginWord);

        PriorityQueue<NodeGBFS> todo = new PriorityQueue<>((a, b) -> {
            int diffA = getHeuristic(a.word, endWord);
            int diffB = getHeuristic(b.word, endWord);
            return Integer.compare(diffA, diffB);
        });
        todo.add(beginNode);

        while (!todo.isEmpty()) {
            NodeGBFS current = todo.poll();
            visited[0]++;
```

```
            if (current.word.equals(endWord)) {
                return current.path;
            }
            wordList.remove(current.word);
            char[] charArray = current.word.toCharArray();
            for (int j = 0; j < charArray.length; j++) {
                char originalChar = charArray[j];
                for (char c = 'a'; c <= 'z'; c++) {
                    charArray[j] = c;
                    String newWord = new String(charArray);
                    if (wordList.containsKey(newWord)) {
                        List<String> tempPath = new
ArrayList<>(current.path);
                        tempPath.add(newWord);
                        NodeGBFS newNode = new NodeGBFS(newWord, tempPath);
                        todo.add(newNode);
                        wordList.remove(newWord);
                    }
                }
                charArray[j] = originalChar;
            }
        }
        return new ArrayList<>();
    }

    static private int getHeuristic(String word, String target) {
        int diff = 0;
        for (int i = 0; i < word.length(); i++) {
            if (word.charAt(i) != target.charAt(i)) {
                diff++;
            }
        }
        return diff;
    }
}
```

4.2.3. Algoritma A*

```
import java.util.*;

class NodeAStar {
    String word;
    int cost;
    int heuristic;
    List<String> path;

    NodeAStar(String word) {
        this.word = word;
        this.cost = 0;
        this.heuristic = 0;
        this.path = new ArrayList<>();
    }

    NodeAStar(String word, int cost, int heuristic, List<String> path) {
        this.word = word;
        this.cost = cost;
        this.heuristic = heuristic;
        this.path = path;
    }
}
```

```
import java.util.*;

class AStar {
    static public List<String> algo(String beginWord, String endWord,
Map<String, Boolean> wordList, long[] visited) {
        NodeAStar beginNode = new NodeAStar(beginWord);
        beginNode.path.add(beginWord);

        PriorityQueue<NodeAStar> todo = new PriorityQueue<>(
                (a, b) -> Integer.compare(a.cost + a.heuristic, b.cost +
b.heuristic));
```

```
        todo.add(beginNode);

        while (!todo.isEmpty()) {
            NodeAStar current = todo.poll();
            visited[0]++;
            if (current.word.equals(endWord)) {
                return current.path;
            }
            wordList.remove(current.word);
            char[] charArray = current.word.toCharArray();
            for (int j = 0; j < charArray.length; j++) {
                char originalChar = charArray[j];
                for (char c = 'a'; c <= 'z'; c++) {
                    charArray[j] = c;
                    String newWord = new String(charArray);
                    if (wordList.containsKey(newWord)) {
                        List<String> tempPath = new
ArrayList<>(current.path);
                        tempPath.add(newWord);
                        int heuristic = getHeuristic(newWord, endWord);
                        NodeAStar newNode = new NodeAStar(newWord,
current.cost + 1, heuristic, tempPath);
                        todo.add(newNode);
                        wordList.remove(newWord);
                    }
                }
                charArray[j] = originalChar;
            }
        }
        return new ArrayList<>();
    }

    static private int getHeuristic(String word, String target) {
        int diff = 0;
        for (int i = 0; i < word.length(); i++) {
            if (word.charAt(i) != target.charAt(i)) {
```

```
                diff++;
            }
        }
        return diff;
    }
}
```

4.2. Penjelasan Class dan Method

4.2.1. Algoritma Uniform Cost Search

Terdapat dua Class dalam algoritma ini, yaitu Class NodeUCS dan UCS.

4.2.1.1. Class NodeUCS

Class ini berisi Node yang digunakan oleh Algoritma Uniform Cost Search. Class ini memiliki atribut word yang menyimpan kata dari node tersebut, cost yang menyimpan harga node, dan path yang berisi jalan yang ditempuh dari node awal hingga node sekarang.

4.2.1.2. Class UCS

Class ini berisi implementasi Algoritma Uniform Cost Search untuk menyelesaikan permainan Word Ladder. Terdapat satu method dalam Class ini, yaitu method algo() yang berisi algoritma utama.

4.2.2. Algoritma Greedy Best First Search

Terdapat dua Class dalam algoritma ini, yaitu Class NodeGBFS dan GBFS.

4.2.2.1. Class NodeGBFS

Class ini berisi Node yang digunakan oleh Algoritma Greedy Best First Search. Class ini memiliki atribut word yang menyimpan kata dari node tersebut dan path yang berisi jalan yang ditempuh dari node awal hingga node sekarang.

4.2.2.2. Class GBFS

Class ini berisi implementasi Algoritma Greedy Best First Search untuk menyelesaikan permainan Word Ladder. Terdapat dua method dalam Class ini, yaitu method algo() yang berisi algoritma utama dan method getHeuristic() untuk menghitung nilai heuristik dari setiap node.

4.2.3. Algoritma A*

Terdapat dua Class dalam algoritma ini, yaitu Class NodeAStar dan AStar.

4.2.3.1. Class NodeAStar

Class ini berisi Node yang digunakan oleh Algoritma A*. Class ini memiliki atribut word yang menyimpan kata dari node tersebut, cost yang menyimpan harga node, heuristic yang berisi nilai heuristik dari node, dan path yang berisi jalan yang ditempuh dari node awal hingga node sekarang.

4.2.3.2. Class AStar

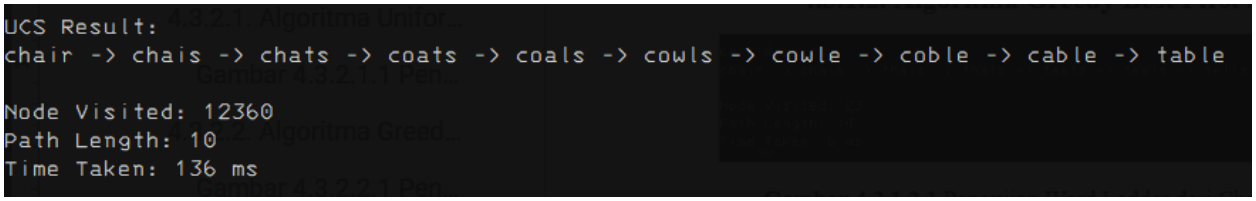
Class ini berisi implementasi Algoritma A* untuk menyelesaikan permainan Word Ladder. Terdapat dua method dalam Class ini, yaitu method algo() yang berisi algoritma utama dan method getHeuristic() untuk menghitung nilai heuristik dari setiap node.

4.3. Pengujian

Berikut adalah pengujian algoritma UCS, GBFS, dan A* terhadap permainan Word Ladder dengan beberapa sampel kata:

4.3.1. Chair ke Table

4.3.1.1. Algoritma Uniform Cost Search

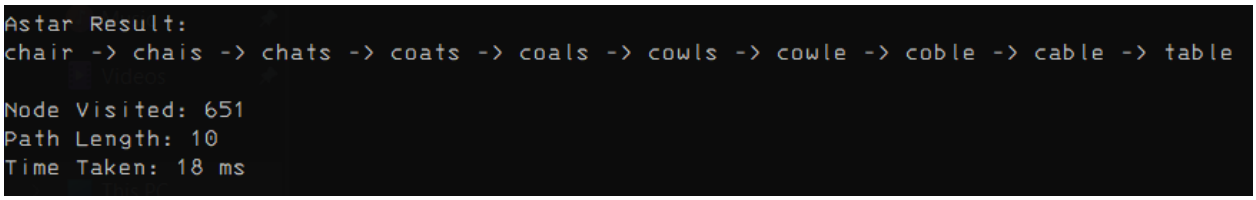


Gambar 4.3.1.1.1 Pengujian Word Ladder dari Chair ke Table dengan Algoritma Uniform Cost Search

4.3.1.2. Algoritma Greedy Best First Search

Gambar 4.3.1.2.1 Pengujian Word Ladder dari Chair ke Table dengan Algoritma Greedy Best First Search

4.3.1.3. Algoritma A*



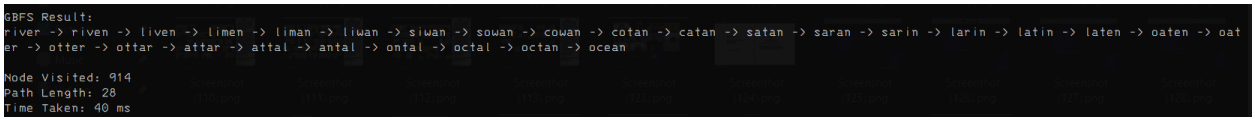
Gambar 4.3.1.3.1 Pengujian Word Ladder dari Chair ke Table dengan Algoritma A*

4.3.2. River ke Ocean

4.3.2.1. Algoritma Uniform Cost Search

Gambar 4.3.2.1.1 Pengujian Word Ladder dari River ke Ocean dengan Algoritma Uniform Cost Search

4.3.2.2. Algoritma Greedy Best First Search



Gambar 4.3.2.2.1 Pengujian Word Ladder dari River ke Ocean dengan Algoritma Greedy Best First Search

4.3.2.3. Algoritma A*

```
Astar Result:
river -> rider -> rides -> aides -> andes -> antes -> antas -> antal -> ontal -> octal -> octan -> ocean

Node Visited: 4803
Path Length: 12
Time Taken: 94 ms
```

Gambar 4.3.2.3.1 Pengujian Word Ladder dari River ke Ocean dengan Algoritma A*

4.3.3. Apple ke Mango

4.3.3.1. Algoritma Uniform Cost Search

```
UCS Result:
apple -> ample -> amole -> anole -> angle -> engle -> eagle -> bagle -> bagge -> bange -> mange -> mango

Node Visited: 7068
Path Length: 12
Time Taken: 106 ms
```

Gambar 4.3.3.1.1 Pengujian Word Ladder dari Apple ke Mango dengan Algoritma Uniform Cost Search

4.3.3.2. Algoritma Greedy Best First Search

```
GBFS Result:
apple -> apply -> appay -> appal -> aptal -> attal -> attar -> atter -> aster -> mster -> mater -> matey -> maney -> mangy -> mango

Node Visited: 31
Path Length: 15
Time Taken: 7 ms
```

Gambar 4.3.3.2.1 Pengujian Word Ladder dari Apple ke Mango dengan Algoritma Greedy Best First Search

4.3.3.3. Algoritma A*

```
Astar Result:
apple -> ample -> amole -> anole -> angle -> engle -> eagle -> bagle -> bagge -> bange -> mange -> mango

Node Visited: 285
Path Length: 12
Time Taken: 11 ms
```

Gambar 4.3.3.3.1 Pengujian Word Ladder dari Apple ke Mango dengan Algoritma A*

4.3.4. Atlases ke Cabaret

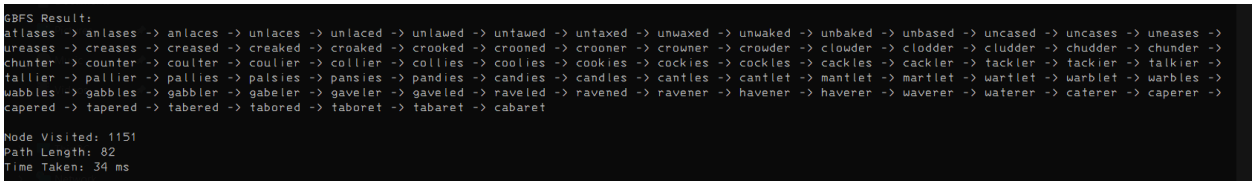
4.3.4.1. Algoritma Uniform Cost Search

```
UCS Result:
atlases -> anlases -> anlaces -> unlaces -> unlaced -> unfaced -> unfaked -> uncaked -> uncakes -> uncases -> uneases -> ureases -> creases -> creased -> creaked ->
croaked -> crocked -> chocked -> shocked -> stocked -> stooked -> stroked -> striked -> strikes -> shrikes -> shrines -> serines -> serenes -> serener -> sevener ->
severer -> leverer -> levered -> lovered -> hovered -> havered -> wavered -> watered -> catered -> capered -> tapered -> tabered -> tabored -> taboret -> tabaret ->
cabaret

Node Visited: 12356
Path Length: 46
Time Taken: 176 ms
```

Gambar 4.3.4.1.1 Pengujian Word Ladder dari Atlases ke Cabaret dengan Algoritma Uniform Cost Search

4.3.4.2. Algoritma Greedy Best First Search



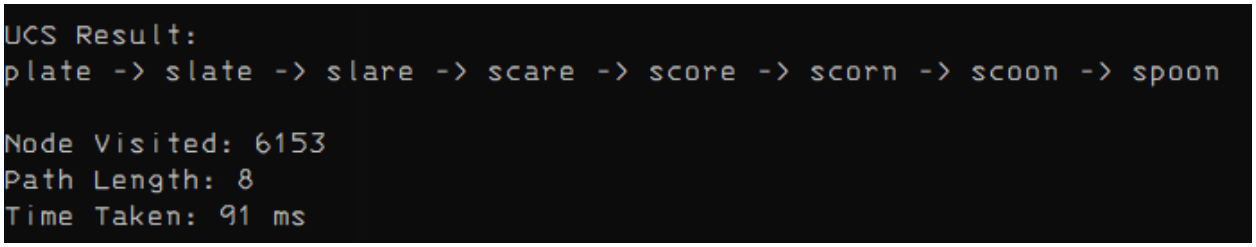
Gambar 4.3.4.2.1 Pengujian Word Ladder dari Atlases ke Cabaret dengan Algoritma Greedy Best First Search

4.3.4.3. Algoritma A*

Gambar 4.3.4.3.1 Pengujian Word Ladder dari Atlases ke Cabaret dengan Algoritma A*

4.3.5. Plate ke Spoon

4.3.5.1. Algoritma Uniform Cost Search



Gambar 4.3.5.1.1 Pengujian Word Ladder dari Plate ke Spoon dengan Algoritma Uniform Cost Search

4.3.5.2. Algoritma Greedy Best First Search

Gambar 4.3.5.2.1 Pengujian Word Ladder dari Plate ke Spoon dengan Algoritma Greedy Best First Search

4.3.5.3. Algoritma A*

```
Astar Result:
plate -> slate -> slote -> shote -> shore -> shorn -> shoon -> spoon

Node Visited: 117
Path Length: 8
Time Taken: 8 ms
```

Gambar 4.3.5.3.1 Pengujian Word Ladder dari Plate ke Spoon dengan Algoritma A*

4.3.6. Queen ke Kings

4.3.6.1. Algoritma Uniform Cost Search

```
UCS Result:
queen -> queet -> quent -> quint -> quins -> quits -> duits -> dunts -> dints -> dings -> kings

Node Visited: 8700
Path Length: 11
Time Taken: 124 ms
```

Gambar 4.3.6.1.1 Pengujian Word Ladder dari Queen ke Kings dengan Algoritma Uniform Cost Search

4.3.6.2. Algoritma Greedy Best First Search

```
GBFS Result:
queen -> queet -> quest -> quist -> quint -> quins -> ruins -> rains -> kains -> karns -> kirns -> kirks -> kinks -> kings

Node Visited: 25
Path Length: 14
Time Taken: 5 ms
```

Gambar 4.3.6.2.1 Pengujian Word Ladder dari Queen ke Kings dengan Algoritma Greedy Best First Search

4.3.6.3. Algoritma A*

```
Astar Result:
queen -> queet -> quent -> quint -> quins -> quits -> duits -> dunts -> dints -> dings -> kings

Node Visited: 165
Path Length: 11
Time Taken: 9 ms
```

Gambar 4.3.6.3.1 Pengujian Word Ladder dari Queen ke Kings dengan Algoritma A*

4.4. Analisis

4.4.1. Algoritma Greedy Best First Search

Algoritma Greedy Best First Search cenderung memberikan hasil yang memakan waktu paling sedikit, tetapi dengan jalur yang lebih panjang dibandingkan dengan dua algoritma

lainnya. Hal ini disebabkan karena Algoritma Greedy Best First Search hanya mempertimbangkan nilai heuristik yang berupa jumlah perbedaan huruf dalam pemilihan node selanjutnya. Ini membuatnya cenderung memilih jalur yang menuju ke tujuan secara langsung tanpa mempertimbangkan biaya yang sudah ditempuh. Namun, hal ini dapat mengakibatkan GBFS terjebak dalam jalur yang tidak optimal jika heuristiknya tidak cukup akurat. Meskipun cenderung cepat dalam menemukan solusi, tetapi tidak menjamin bahwa solusi yang ditemukan memiliki jalur yang terpendek.

Karena GBFS cenderung memilih jalur yang paling dekat ke tujuan, maka GBFS membutuhkan alokasi memori yang paling sedikit dibandingkan dengan dua algoritma lainnya. Hal ini dapat dilihat pada tangkapan layar di bagian pengujian dimana node yang dikunjungi oleh algoritma ini lebih sedikit dibandingkan dengan A* dan UCS sehingga memori yang dibutuhkan untuk menjelajahi node-node tersebut akan lebih sedikit pula.

Berdasarkan penjelasan ini, bisa disimpulkan bahwa Algoritma Greedy Best First Search cukup optimal dalam waktu dan memori, tapi tidak optimal dalam jalur yang dipilih karena jalur yang ditempuh oleh algoritma ini belum tentu jalur terpendek.

4.3.2. Algoritma A*

Algoritma A* mempertimbangkan dua faktor dalam penentuan node selanjutnya, yaitu nilai heuristik berupa jumlah perbedaan huruf dan juga biaya untuk sampai ke node tersebut. Karena mempertimbangkan dua faktor, solusi yang ditemukan cenderung lebih akurat sehingga menghasilkan solusi dengan jalur yang lebih pendek dibandingkan dengan GBFS. Namun, karena faktor yang dipertimbangkan lebih banyak, waktu yang diperlukan untuk mencari solusi dengan algoritma ini juga akan lebih lama dibandingkan dengan GBFS.

Sedangkan untuk memori, jika dilihat dari jumlah node yang dikunjungi, dibutuhkan memori yang lebih besar daripada memori yang dibutuhkan GBFS, tetapi tidak sebesar memori yang dibutuhkan UCS. Hal ini disebabkan Algoritma A* mempertimbangkan nilai heuristik dan biaya dalam pemilihan node sehingga A* melakukan iterasi lebih banyak untuk menemukan node yang sesuai. Karena itu, jalur yang dipilih juga lebih pendek dibandingkan dengan jalur yang ditemukan GBFS.

Karena itulah bisa disimpulkan bahwa optimalitas Algoritma A* berada di level menengah. Algoritma ini cukup optimal dalam waktu dan memori walaupun tidak se-optimal GBFS, tetapi dapat memberikan jalur yang lebih pendek dibandingkan dengan GBFS.

4.3.3. Algoritma Uniform Cost Search

Algoritma Uniform Cost Search memakan waktu paling banyak untuk mencari solusi karena algoritma ini hanya mempertimbangkan biaya untuk sampai ke node tujuan. Algoritma ini sama sekali tidak memperhitungkan nilai heuristik sehingga cenderung menjelajahi semua kemungkinan jalur dengan biaya yang berbeda-beda.

Karena tidak mempertimbangkan nilai heuristik, maka jumlah node yang dikunjungi UCS adalah yang paling banyak jika dibandingkan dengan dua algoritma lainnya. Sebab itu, memori yang dibutuhkan adalah yang paling banyak pula.

Oleh karena itu bisa dikatakan bahwa Algoritma UCS paling tidak optimal dibandingkan dengan dua algoritma lainnya, baik dari segi waktu maupun dari segi memori. Tetapi karena algoritma ini murni hanya memilih node berdasarkan biaya, maka jalur yang dihasilkan dari Algoritma UCS sudah pasti merupakan jalur terpendek.

Walaupun sama-sama melakukan pencarian secara melebar, tetap terdapat perbedaan antara algoritma ini dengan Algoritma BFS. Algoritma BFS melakukan pencarian secara melebar dari simpul awal ke simpul-simpul tetangga tanpa memperhitungkan biaya penjelajahan node. Sedangkan UCS melakukan pencarian sambil memperhitungkan masing-masing bobot dari nodenya. Setelah itu mengembalikan jalur yang memiliki total bobot minimum.

BAB V
KESIMPULAN DAN SARAN

5.1. Kesimpulan

Melalui penyelesaian tugas ini penulis berhasil mengimplementasikan program untuk menyelesaikan masalah Word Ladder dengan Algoritma Uniform Cost Search, Greedy Best First Search, dan A*. Kemudian berdasarkan analisis yang telah dilakukan, dapat ditarik kesimpulan bahwa secara waktu dan memori, GBFS paling unggul, diikuti dengan A*, lalu kemudian UCS.

Tetapi walaupun unggul secara waktu dan memori, GBFS tetap memiliki downside yaitu jalur yang dihasilkan dengan Algoritma GBFS belum tentu jalur yang terpendek. Sedangkan untuk A* dan UCS, walaupun mereka tidak lebih unggul dari GBFS, tetapi jalur yang mereka hasilkan pasti akan lebih pendek daripada jalur yang dihasilkan oleh GBFS.

5.2. Saran

Dalam menyelesaikan tugas ini, terdapat beberapa saran dan peningkatan yang bisa penulis terapkan untuk tugas-tugas selanjutnya. Diantaranya adalah:

1. Lebih mendalami teori mengenai algoritma-algoritma yang akan digunakan dalam tugas ini agar tidak kesulitan diakhir
2. Lebih sering untuk buka QnA agar tidak tertinggal informasi
3. Tidak menunda-nunda pengerjaan Tugas Kecil hanya karena judulnya “Tugas Kecil”

LAMPIRAN

Tautan repository: https://github.com/Nerggg/Tucil3_13522147

DAFTAR PUSTAKA

Munir, Rinaldi. 2024. Penentuan rute (Route/Path Planning) - Bagian 1. *https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf*. Diakses pada 4 Mei 2024.

Munir, Rinaldi. 2024. Penentuan rute (Route/Path Planning) - Bagian 2. *https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf*. Diakses pada 4 Mei 2024.