

IF2240 - Strategi Algoritma

Tugas Kecil 3

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



Hugo Sabam Augusto

13522129

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2024

BAB I

PENDAHULUAN

1. Word Ladder

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/>)

1.1. Algoritma UCS

Pencarian biaya seragam (Uniform Cost Search) adalah algoritma pencarian graf yang digunakan dalam kecerdasan buatan untuk menemukan jalur terpendek dalam sebuah graf berbobot. Berbeda dengan algoritma pencarian graf lainnya yang mempertimbangkan biaya langkahnya saja, Uniform Cost Search mempertimbangkan total biaya dari awal hingga titik tertentu dalam graf. Dengan demikian, algoritma ini efektif untuk pencarian jalur terpendek dalam graf dengan bobot yang berbeda-beda pada setiap edge-nya.

1.2. Algoritma Greedy Best First Search

Greedy Best First Search adalah algoritma pencarian graf yang berfokus pada eksplorasi jalur yang terlihat paling menjanjikan berdasarkan suatu heuristik, tanpa mempertimbangkan biaya total perjalanan. Algoritma ini secara terus-menerus memilih simpul berikutnya yang memiliki nilai heuristik paling rendah, dengan harapan mencapai tujuan lebih cepat. Namun, karena cenderung memilih jalur yang tampaknya paling menguntungkan saat ini tanpa mempertimbangkan konsekuensi jangka panjang, Greedy Best First Search tidak menjamin pencarian jalur terpendek atau optimal.

1.3. Algoritma A* (A Star)

A* adalah algoritma pencarian graf yang menggabungkan konsep dari pencarian biaya seragam (Uniform Cost Search) / $g(n)$ dan Greedy Best First Search / $h(n)$ dengan mempertimbangkan biaya sejauh ini ditambah estimasi biaya yang tersisa hingga tujuan. Dengan menggunakan fungsi heuristik untuk mengestimasi biaya tersisa, A* dapat memilih jalur dengan kombinasi biaya terendah dan kemungkinan terbesar untuk mencapai tujuan. Algoritma ini terkenal karena kemampuannya dalam menemukan jalur terpendek atau optimal dalam graf dengan bobot yang berbeda-beda.

BAB II

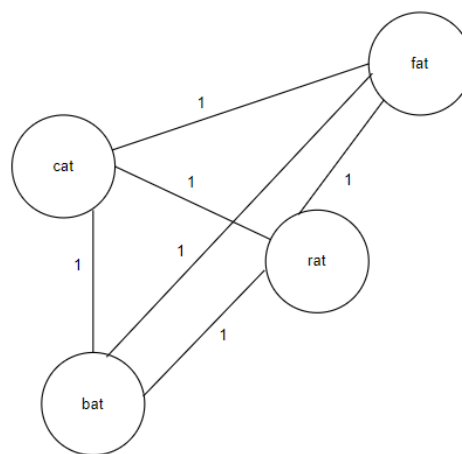
IMPLEMENTASI DAN ANALISIS ALGORITMA

2. Implementasi dan Analisis Algoritma

Untuk implementasi ketiga algoritma ini untuk mencari solusi dari permainan *Wiki Ladder*. Pertama, saya membuat sebuah struktur data *map* (saya sebut outer map). Pembuatan map ini cukup membutuhkan waktu yang lama karena harus *parsing* semua kata dari kamus dan dicari hubungan ketetanggaannya. Karena itu, dalam program saya implementasikan sebuah *MapSaver* yang akan menyimpan data map tersebut ke dalam sebuah file txt dengan panjang kata yang berbeda beda untuk tiap filenya. Jadi, ketika program berjalan dan misal dicari sebuah kata dengan panjang 5, program hanya akan mengambil data map panjang 5. Dengan ini harusnya program bisa berjalan lebih cepat dan efisien.

Key dari *map* tersebut adalah sebuah panjang kata yang sudah saya *parsing* dari kamus data bahasa inggris. Value dari *map* itu sendiri adalah sebuah *map* lagi (saya sebut inner map) dengan key nya berupa tiap *String* kata yang ada di dalam kamus. Value dari inner map adalah sebuah *List of Strings* yang saya gunakan sebagai representasi ‘tetangga’ dari key tersebut. Misal saya mempunyai key “cat” maka tetangganya adalah *String* kata yang **berbeda 1 char dari “cat” tersebut dan terdaftar di dalam kamus data**, seperti “bat”, “dat”, “fat”, “rat”, dan lain lain. Dari struktur data map ini, nanti saya konversi menjadi sebuah graf.

Selanjutnya, Key dari inner map nantinya akan dijadikan simpul dan value(*List of Strings*) dari key tersebut dijadikan tetangga dari simpul tersebut dengan nilai sisi *weight* atau harga bernilai 1. Kenapa 1? Karena hubungan jarak yang saya gunakan adalah perbedaan banyaknya char antar simpul dengan simpul lainnya. Di kasus yang saya jelaskan diatas akan direpresentasikan dalam Graf yang isinya hanya kata kata dengan panjang 3, Jadi untuk tiap panjang kata yang berbeda, akan dibuat dalam graf yang berbeda.



Gambar 2. Representasi Graf dari konversi map
Sumber: Dokumentasi pribadi

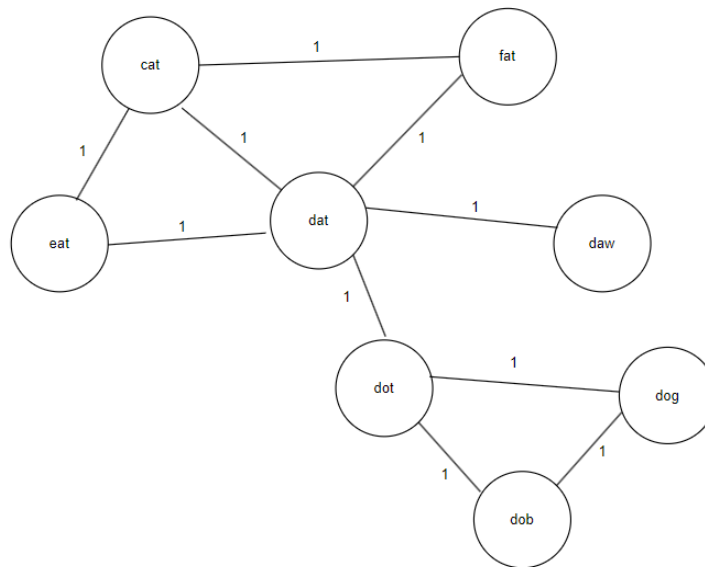
Saya menggunakan struktur data Priority Queue untuk implementasi UCS, Greedy Best First Search dan A*. Elemen dari Priority Queue untuk ketiga algoritma ini adalah sebuah kelas 'Simpul' yang terdiri dari *String* kata, *Integer* totalWeight, *List of String* yang digunakan sebagai jalur, dan *Integer* order(khusus digunakan untuk UCS, dijelaskan lebih lanjut).

2.1. UCS

Untuk implementasi UCS (*Uniform Cost Search*). Prioritas dari priority queue-nya berdasarkan total berat dan order. Apabila total berat lebih ringan maka prioritas lebih tinggi, selanjutnya apabila total berat sama, maka dibandingkan dengan order/urutan.

Langkah algoritmanya adalah:

1. Inisialisasi sebuah 'Simpul' awal yang berisi kata awal dengan total berat 0 dengan order 0(karena urutan pertama di priority). Lalu push simpul ini
2. Setelah itu memasuki while loop, selama priority queue tidak kosong, maka loop terus.
3. Pop 'Simpul' terkiri, cek apabila sudah pernah dilalui maka *continue*, apabila tidak maka akan memasuki pengecekan simpul tetangga
4. Apabila Simpul yang di pop berisi string kata tujuan yang dicari, maka while stop. Jika tidak lanjut
5. Iterasi semua Simpul tetangga dari Simpul yang di pop, lalu cek apakah simpul tetangga merupakan Simpul yang sudah pernah dijelajahi oleh Simpul yang di pop (jalur dijelajahi disimpan dalam *List of String* jalur), apabila iya maka tidak dimasukkan ke priority queue, apabila belum maka di push. Hal ini saya lakukan agar simpul tidak 'bolak-balik' ke simpul yang sebelumnya, jadi ter-*guaranteed* bahwa simpul yang dimasukkan ke priority queue merupakan simpul yang belum pernah dijelajahi.
6. Simpul yang dimasukkan ke priority queue adalah semua simpul tetangga dari simpul yang di pop, dengan *List of Strings* jalur yang sudah di *append* dengan kata, total berat dari simpul yang di pop dan order di *increment*.
7. Ulangi langkah ke 2.
8. Apabila di mendapat kasus priority queue kosong, berarti tidak ada solusi.



Gambar3. Ilustrasi potongan graf kata panjang 3
Sumber: Pribadi

Misal, saya akan menggunakan graf di atas dan saya ingin mencari path dari kata 'cat' ke 'dot', maka Ilustrasi priority queue-nya akan seperti ini:

| | |
|-----------------------|---|
| cat | eat(cat-1), dat(cat-1), fat(cat-1) |
| eat(cat-1) | dat(cat-1),fat(cat-1), dat(cat eat-2) |
| dat(cat-1) | fat(cat-1), dat(cat eat-2) , fat(cat dat - 2), daw(cat dat - 2), eat(cat dat -2), dot(cat dat-2) |
| fat(cat-1) | dat(cat eat-2) , fat(cat dat - 2), daw(cat dat - 2), eat(cat dat -2), dot(cat dat-2), dat(cat fat-2) |
| dat(cat eat-2) | fat(cat dat - 2), daw(cat dat - 2), eat(cat dat -2), dot(cat dat-2), dat(cat fat-2), eat(cat eat dat - 3), fat(cat eat dat - 3),daw(cat eat dat - 3) ,dot(cat eat dat - 3) |
| fat(cat dat-2) | daw(cat dat - 2), eat(cat dat -2), dot(cat dat-2), dat(cat fat-2), eat(cat eat dat - 3), fat(cat eat dat - 3),daw(cat eat dat - 3) ,dot(cat eat dat - 3), dat(cat dat fat-3) |
| daw(cat dat - 2) | eat(cat dat -2), dot(cat dat-2), dat(cat fat-2), eat(cat eat dat - 3), fat(cat eat dat - 3),daw(cat eat dat - 3) ,dot(cat eat dat - 3), dat(cat dat fat-3) |
| eat(cat dat-2) | dot(cat dat-2), dat(cat fat-2), eat(cat eat dat - 3), fat(cat eat dat - 3),daw(cat eat dat - 3) ,dot(cat eat dat - 3), dat(cat dat fat-3) |
| dot(cat dat-2) | Solusi dapat karena Simpul yang di pop sudah sesuai dengan kata tujuan. |

Dilihat dari pola nya, UCS dengan kebetulan bobot berat 1 ini mirip dengan algoritma BFS(Breadth-First Search). Dari segi algoritma, BFS dan UCS itu sangat berbeda karena BFS itu ibaratnya hanyalah sebuah queue tanpa prioritas apapun. Kalau di UCS ada prioritas. Namun karena graf dalam kasus ini semua hubungan antar simpul hanya berbeban 1, terlihat bahwa penyelesaian ini bisa diselesaikan juga dengan BFS. **Jadi Kesimpulan yang bisa saya tarik adalah mau digunakan BFS atau UCS pada masalah *Wiki Ladder* ini, path solusi yang dihasilkan sama.**

2.2. Greedy Best First Search

Untuk implementasi Greedy Best First Search. Prioritas dari priority queue-nya berdasarkan total berat. Apabila total berat lebih ringan maka prioritas lebih tinggi. Heuristik yang digunakan untuk algoritma ini adalah banyaknya perbedaan huruf antar kata, misal kata 'dog' dengan kata 'cat' memiliki nilai heuristik 3 karena tiap huruf berbeda antarkata. Langkah algoritmanya adalah:

1. Inisialisasi sebuah 'Simpul' awal yang berisi kata awal dengan total berat yang bernilai jarak dari kata awal ke kata tujuan. Lalu push simpul ini
2. Setelah itu memasuki while loop, selama priority queue tidak kosong, maka loop terus.
3. Pop 'Simpul' terkiri, cek apabila simpul valid ada di dalam graf dan mempunyai tetangga, apabila tidak memenuhi maka continue.
4. Apabila Simpul yang di pop berisi memiliki nilai heuristik = 0, maka solusi ditemukan, loop dihentikan.
5. Iterasi semua Simpul tetangga dari Simpul yang di pop, lalu cek apakah simpul tetangga merupakan Simpul yang sudah pernah dijelajahi oleh Simpul yang di pop (jalur dijelajahi disimpan dalam *List of String* jalur), apabila iya maka tidak dimasukkan ke priority queue, apabila belum maka di push. Hal ini saya lakukan agar simpul tidak 'bolak-balik' ke simpul yang sebelumnya, jadi ter-*guaranteed* bahwa simpul yang dimasukkan ke priority queue merupakan simpul yang belum pernah dijelajahi.
6. Simpul yang dimasukkan ke priority queue adalah semua simpul tetangga dari simpul yang di pop, dengan *List of Strings* jalur yang sudah di-*append* dengan kata, total berat dengan nilai heuristik antar kata dari simpul tersebut dengan kata tujuan.
7. Ulangi langkah ke 2.
8. Apabila di mendapat kasus priority queue kosong, berarti tidak ada solusi.

Misal, saya akan menggunakan graf di atas (Gambar 3) dan saya ingin mencari path dari kata 'cat' ke 'dog', maka Ilustrasi priority queue-nya akan seperti ini:

| | |
|----------------|---|
| cat3 | dat(cat-2) eat(cat-3) fat(cat-3) |
| dat(cat-2) | dot(cat dat-1) daw(cat dat-2) eat(cat-3) fat(cat-3) |
| dot(cat dat-1) | dog(cat dat dot-0) dob(cat dat dot -1) dot(cat dat-1) daw(cat dat-2) eat(cat-3) fat(cat-3) |

| | |
|-----------------------|---|
| dog(cat dat dot-0) | Selesai karena total berat = 0, solusi ditemukan dengan path cat dat dot dog |
|-----------------------|---|

Terlihat sekilas bahwa algoritma ini sangat efisien dalam mencari solusi karena node yang perlu dieksplor sangat sedikit (hanya 4 kali pop). Namun, secara teoritis algoritma ini tidak menjamin memberikan solusi yang optimal layaknya UCS. Sama halnya dengan kasus saat ini *Word Ladder*, algoritma ini tidak bisa menjamin memberikan solusi yang optimal. Akan terlihat jelas ketika akan dilakukan uji coba antar ketiga algoritma ini.

2.3. A*(A Star)

Untuk implementasi A*, Prioritas dari priority queue-nya berdasarkan total berat. Apabila total berat lebih ringan maka prioritas lebih tinggi. A* sendiri memiliki 2 fungsi untuk menentukan total berat di simpul priority queue-nya yaitu $g(n)$ dan $h(n)$. $g(n)$ untuk implementasi disini adalah jarak dari root ke simpul sekarang yang sedang dieksplor (UCS). $h(n)$ untuk implementasi disini adalah jarak dari simpul sekarang ke tujuan (GBFS). $g(n)$ disini bisa didapatkan dengan menghitung panjang path yang sudah dilalui oleh suatu simpul di priority queue-nya. $h(n)$ bisa didapatkan dengan menghitung nilai heuristik (sama halnya dengan GBFS) antar kata simpul dengan kata tujuan. Berikut langkah algoritmanya..

1. Inisialisasi sebuah 'Simpul' awal yang berisi kata awal dengan total berat yang bernilai 0 (karena jarak dari kata awal ke kata awal = 0) + jarak dari kata awal ke kata tujuan. Lalu push simpul ini
2. Setelah itu memasuki while loop, selama priority queue tidak kosong, maka loop terus.
3. Pop 'Simpul' terkiri, cek apabila simpul valid ada di dalam graf dan mempunyai tetangga, apabila tidak memenuhi maka continue.
4. Apabila Simpul yang di pop berisi memiliki kata yang sama dengan kata tujuan, maka solusi ditemukan dan loop dihentikan.
5. Iterasi semua Simpul tetangga dari Simpul yang di pop, lalu cek apakah simpul tetangga merupakan Simpul yang sudah pernah dijelajahi oleh Simpul yang di pop (jalur dijelajahi disimpan dalam *List of String* jalur), apabila iya maka tidak dimasukkan ke priority queue, apabila belum maka di push. Hal ini saya lakukan agar simpul tidak 'bolak-balik' ke simpul yang sebelumnya, jadi ter-*guaranteed* bahwa simpul yang dimasukkan ke priority queue merupakan simpul yang belum pernah dijelajahi.
6. Simpul yang dimasukkan ke priority queue adalah semua simpul tetangga dari simpul yang di pop, dengan *List of Strings* jalur yang sudah di-*append* dengan kata, total berat dengan nilai jarak kata awal ke simpul sekarang + heuristik antar kata dari simpul tersebut dengan kata tujuan ($g(n) + h(n)$).
7. Ulangi langkah ke 2.
8. Apabila di mendapat kasus priority queue kosong, berarti tidak ada solusi.

Misal, saya akan menggunakan graf di atas (Gambar 3) dan saya ingin mencari path dari kata 'cat' ke 'dog', maka Ilustrasi priority queue-nya akan seperti ini:

| | |
|----------------------|--|
| cat(0+3) | dat(cat - 1+2) eat(cat - 1+3) fat(cat - 1+3) |
| dat(cat- 3) | dot(cat dat - 2+1) daw(cat dat - 2+2) eat(cat - 1+3) fat(cat - 1+3) |
| dot(cat dat -3) | dog(cat dat dot - 3+ 0) dob(cat dat dot - 3 + 1) dot(cat dat - 2+1) daw(cat dat - 2+2) eat(cat - 1+3) fat(cat - 1+3) |
| dog(cat dat dot - 3) | Selesai, dog ditemukan dengan path cat dat dot dog |

Dalam implementasi A* ini, heuristik yang saya pakai ($h(n)$) admissible. Karena $h(n)$ tidak akan pernah melebihi dari jarak sebenarnya. $h(n)$ yang saya gunakan adalah banyaknya huruf yang berbeda antar kata pada letak panjang yang sama. Misal 'moon' ke 'wise', nilai $h(n)$ nya adalah 4. Terbukti jelas bahwa jarak asli dari 'moon' ke 'wise' ini memang 4 sesuai dengan peraturan *Word Ladder* jadi $h(n) \leq 4$. Maka dengan heuristik ini, A* yang saya implementasikan **admissible**.

Secara teoritis, Algoritma A* lebih efisien dibandingkan dengan algoritma UCS karena A* dibantu informasi dari heuristik UCS, meskipun optimal dalam hal biaya, tidak menggunakan heuristik dan hanya mempertimbangkan biaya aktual dari simpul yang dieksplorasi. Ini bisa menjadi masalah dalam kasus Word Ladder di mana setiap langkah memiliki biaya yang sama (yaitu, biaya 1), karena UCS tidak memiliki informasi tambahan untuk memprioritaskan simpul-simpul yang lebih dekat ke solusi. Namun, bisa saja ada kasus dimana A* lebih lambat daripada UCS karena beberapa alasan seperti jumlah simpul yang perlu dieksplorasi dalam A* bisa menjadi lebih besar daripada UCS karena A* cenderung mengeksplorasi jalur-jalur yang lebih beragam untuk mencari solusi yang optimal, sementara UCS hanya mempertimbangkan biaya aktual dari simpul-simpul yang dieksplorasi.

BAB III

ANALISIS DAN UJI COBA

3. Source Code

3.1. Simpul.java

```
package Simpul;

import java.util.List;
import java.util.ArrayList;

public class Simpul implements Comparable<Simpul> {
    private String word;
    private int totalWeight;
    private List<String> path;
    private int order;

    public Simpul(String word, int totalWeight, List<String> path, int
order) {
        this.word = word;
        this.totalWeight = totalWeight;
        this.path = new ArrayList<>(path); // Initialize path as ArrayList
        this.order = order;
    }

    public void addPath(String word) {
        this.path.add(word);
    }

    public String getWord() {
        return this.word;
    }

    public int getTotalWeight() {
        return this.totalWeight;
    }

    public void addOrder(int x) {
        this.order += x;
    }

    public List<String> getPath() {
        return this.path;
    }

    public void addTotalWeight(int weight) {
        this.totalWeight += weight;
    }

    public void printSimpul() {
        System.out.println("Nama Simpul : " + this.word);
        String joined = String.join(", ", this.path);
        System.out.print("Path nya : ");
        System.out.println(joined);
    }
}
```

```

    public int getOrder() {
        return this.order;
    }

    @Override
    public int compareTo(Simpul other) {
        // Prioritize based on total weight, and then order counter
        if (this.totalWeight != other.totalWeight) {
            return Integer.compare(this.totalWeight, other.totalWeight);
        } else {
            return Integer.compare(this.order, other.order);
        }
    }
}

```

3.2. Graf.java

```

package Graf;

import java.util.*;

public class Graf {

    private Map<String, Integer> indexMap; // map untuk memetakan nama node
    ke indeks
    private List<LinkedList<Edge>> adjList; // daftar ketetanggaan untuk
    setiap node

    public class Edge {
        String destination;
        int weight;

        Edge(String destination, int weight) {
            this.destination = destination;
            this.weight = weight;
        }

        public int getWeight() {
            return this.weight;
        }

        public String getDestination() {
            return this.destination;
        }
    }

    public Graf(int V) {
        indexMap = new HashMap<>();
        adjList = new ArrayList<>();
        for (int i = 0; i < V; ++i) {
            adjList.add(new LinkedList<>());
        }
    }
}

```

```

    public List<LinkedList<Edge>> getAdjList() {
        return this.adjList;
    }

    public void addEdge(String source, String destination, int weight) {
        if (!indexMap.containsKey(source)) {
            indexMap.put(source, indexMap.size()); // masuk ke map kalo
belum ada
        }
        int sourceIndex = indexMap.get(source);
        if (!indexMap.containsKey(destination)) {
            indexMap.put(destination, indexMap.size()); // masuk ke map
juga
        }
        int destinationIndex = indexMap.get(destination);

        // ditambah edge ke adjList hanya jika destination valid
        if (destinationIndex < adjList.size()) {
            Edge edge = new Edge(destination, weight);
            adjList.get(sourceIndex).add(edge);
        } else {
            // hapus semua elemen dari adjList jika destination tidak valid
            adjList.get(sourceIndex).clear();
        }
    }

    public void convertMapToGraph(Map<Integer, Map<String, List<String>>>
mapWord) {
        System.out.println("Converting from map to graph...");
        for (Map.Entry<Integer, Map<String, List<String>>> entry :
mapWord.entrySet()) {
            System.out.printf("Sekarang converting map word length %d ke
graph %d \n", entry.getKey(), entry.getKey());
            Map<String, List<String>> lengthMap = entry.getValue();
            for (Map.Entry<String, List<String>> wordEntry :
lengthMap.entrySet()) {
                String word = wordEntry.getKey();
                List<String> neighbors = wordEntry.getValue();
                for (String kata : neighbors) {
                    addEdge(word, kata, 1);
                }
            }
        }
        System.out.println("Converting success!");
    }

    public void printGraph() {
        for (Map.Entry<String, Integer> entry : indexMap.entrySet()) {
            String nodeLabel = entry.getKey();
            int nodeIndex = entry.getValue();
            System.out.print("Node " + nodeLabel + " terhubung ke: ");
            for (Edge edge : adjList.get(nodeIndex)) {
                System.out.print(edge.destination + ",");
            }
            System.out.println();
        }
    }
}

```

```

    public void printAdjNode(String word) {
        int nodeIndex = getNodeIndex(word);
        System.out.println("Node " + word + " Terhubung ke :");
        LinkedList<Edge> edges = adjList.get(nodeIndex);
        for (int i = 0; i < edges.size(); i++) {
            Edge edge = edges.get(i);
            System.out.print(edge.destination);
            if (i < edges.size() - 1) {
                System.out.print(", ");
            }
        }
        System.out.println();
    }
    public String getAdjWord(String word,int i){
        int nodeIndex = getNodeIndex(word);
        LinkedList<Edge> edges = adjList.get(nodeIndex);
        return edges.get(i).destination;
    }

    public int getNodeIndex(String nodeLabel) {
        for (Map.Entry<String, Integer> entry : indexMap.entrySet()) {
            if (entry.getKey().equals(nodeLabel)) {
                return entry.getValue();
            }
        }
        return -1; // idx invalid
    }
    public boolean hasNeighbors(String nodeLabel) {
        int nodeIndex = getNodeIndex(nodeLabel);
        if (nodeIndex != -1) {
            return !adjList.get(nodeIndex).isEmpty();
        }
        return false; // error code tidak nemu
    }
}

```

3.3. UCS.java

```

package UCS;

import java.util.List;
import java.util.PriorityQueue;
import java.util.ArrayList;
import java.util.Comparator;
import Graf.Graf;
import Util.Util;
import Simpul.Simpul;
import java.util.*;

public class UCS {
    private List<String> finalPath;
    private PriorityQueue<Simpul> pq;
    private int orderCounter;
}

```

```

private long time;
private int n_explored;
private boolean isfound;

public UCS() {
    pq = new
PriorityQueue<>(Comparator.comparing(Simpul::getTotalWeight).thenComparing(
Simpul::getOrder));
    finalPath = new ArrayList<>();
    time = 0;
    orderCounter = 0;
    n_explored = 0;
    isfound = false;
}

public List<String> getFinalPath() {
    return this.finalPath;
}

public PriorityQueue<Simpul> getPrioQueue() {
    return this.pq;
}

public void searchUCS(String input, String target, Graf g) {
    long startTime = System.currentTimeMillis();
    List<String> initPath = new ArrayList<>();
    pq.offer(new Simpul(input, 0, initPath, orderCounter++));
    Set<String> visited = new HashSet<>(); // Simpan simpul yang sudah
dieksplorasi

    while (!pq.isEmpty()) {
        Simpul simp = pq.poll();
        if (simp.getWord().equals(target)) {
            long finishTime = System.currentTimeMillis();
            long elapsedTime = finishTime - startTime;
            this.time = elapsedTime;
            this.finalPath = simp.getPath();
            this.finalPath.add(target);
            this.isfound = true;
            return;
        }

        if (visited.contains(simp.getWord())) {
            continue; // Skip simpul yang sudah dieksplorasi
        }

        visited.add(simp.getWord());
        n_explored++;
        int nextId = g.getNodeIndex(simp.getWord());
        for (Graf.Edge edge : g.getAdjList().get(nextId)) {
            if (!simp.getPath().contains(edge.getDestination())) {
                List<String> temp = new ArrayList<>(simp.getPath());
                temp.add(simp.getWord());
                Simpul add = new Simpul(edge.getDestination(),
simp.getTotalWeight() + edge.getWeight(), temp,
orderCounter++);

```



```

        private List<String> finalPath;
        private PriorityQueue<Simpul> pq;
        private long time;
        private int n_explored;
        private boolean isfound;

        public GBFS() {
            pq = new
PriorityQueue<>(Comparator.comparing(Simpul::getTotalWeight));
            finalPath = new ArrayList<>();
            time = 0;
            isfound = false;
        }

        public void searchGBFS(String input, String tujuan, Graf g) {
            long startTime = System.currentTimeMillis();
            List<String> initPath = new ArrayList<>();
            pq.offer(new Simpul(input, Util.countHeuristic(input, tujuan),
initPath, 0));
            while (!pq.isEmpty()) {
                Simpul simp = pq.poll();
                this.n_explored++;
                if (simp.getTotalWeight() == 0) {
                    this.isfound = true;
                    long finishTime = System.currentTimeMillis();
                    long elapsedTime = finishTime - startTime;
                    finalPath = simp.getPath();
                    this.finalPath.add(tujuan);
                    this.time = elapsedTime;
                    return;
                }
                // Node gk valid di graf
                int nextId = g.getNodeIndex(simp.getWord());
                if (nextId == -1) {
                    continue;
                }
                // Simpul gk punya tetangga
                if (g.getAdjWord(simp.getWord(), 0).isEmpty()) {
                    break;
                }
                for (Graf.Edge edge : g.getAdjList().get(nextId)) {
                    if (!simp.getPath().contains(edge.getDestination())) {
                        List<String> temp = new ArrayList<>(simp.getPath());
                        temp.add(simp.getWord());
                        pq.offer(new Simpul(edge.getDestination(),
edge.getDestination()),
temp, 0));
                    }
                }
            }
            // no solusi
            this.isfound = false;
            this.finalPath.clear();
            this.time = 0;
        }
    }

```



```

    }

    public void printPriorityQueue() {
        PriorityQueue<Simpul> tempPQ = new PriorityQueue<>(pq);
        System.out.println("Simpul Hidup : ");
        while (!tempPQ.isEmpty()) {
            Simpul simp = tempPQ.poll();
            System.out.print(simp.getWord() + "(" + simp.getTotalWeight() +
", " + simp.getOrder() + ")" + " ");
        }
        System.out.println("");
    }

    public List<String> getFinalPath() {
        return this.finalPath;
    }

    public void printResult() {
        System.out.println("====SOLUSI GBFS====");
        if(this.isfound){
            String hasil = String.join("-> ", this.finalPath);
            System.out.println(hasil);
        }else{
            System.out.println("Tidak ada solusi");
        }
        System.out.println("Banyak step : " + this.finalPath.size() + "
step");
        System.out.println("Waktu : " + this.time + " ms");
        System.out.println("Waktu : " + (float) this.time / 1000 + "
detik");
        System.out.println("Smpul yang dieksplor: " + this.n_explored);

    }
}

```

3.5. AStar.java

```

package AStar;

import java.util.List;
import java.util.PriorityQueue;
import java.util.ArrayList;
import java.util.Comparator;
import Graf.Graf;
import Util.Util;
import Simpul.Simpul;

public class AStar {
    private List<String> finalPath;
    private PriorityQueue<Simpul> pq;
    private long time;
    private int n_explored;
    private boolean isfound;
}

```

```

    public AStar() {
        pq = new
PriorityQueue<>(Comparator.comparing(Simpul::getTotalWeight));
        finalPath = new ArrayList<>();
        time = 0;
        isfound = false;
    }

    public void searchAStar(String input, String tujuan, Graf g) {
        long startTime = System.currentTimeMillis();
        List<String> initPath = new ArrayList<>();
        pq.offer(new Simpul(input, Util.countHeuristic(input, tujuan),
initPath, 0));
        while (!pq.isEmpty()) {
            // printPriorityQueue();
            Simpul simp = pq.poll();
            this.n_explored++;
            if (simp.getWord().equals(tujuan)) {
                this.isfound = true;
                long finishTime = System.currentTimeMillis();
                long elapsedTime = finishTime - startTime;
                finalPath = simp.getPath();
                this.finalPath.add(tujuan);
                this.time = elapsedTime;
                return;
            }
            //simpul gk valid
            int nextId = g.getNodeIndex(simp.getWord());
            if (nextId == -1) {
                continue;
            }
            // Simpul gk punya tetangga
            if (g.getAdjWord(simp.getWord(), 0).isEmpty()) {
                continue;
            }
            for (Graf.Edge edge : g.getAdjList().get(nextId)) {
                if (!simp.getPath().contains(edge.getDestination())) {
                    List<String> temp = new ArrayList<>(simp.getPath());
                    temp.add(simp.getWord());
                    pq.offer(new Simpul(edge.getDestination(),
                        simp.getPath().size() + edge.getWeight()
                        + Util.countHeuristic(tujuan,
edge.getDestination()),
temp, 0));
                }
            }
            // no solusi
            this.isfound = false;
            this.finalPath.clear();
            this.time = 0; //
        }

        public void printPriorityQueue() {
            PriorityQueue<Simpul> tempPQ = new PriorityQueue<>(pq);
            System.out.println("Simpul Hidup : ");

```

```

        while (!tempPQ.isEmpty()) {
            Simpul simp = tempPQ.poll();
            System.out.print(simp.getWord() + "(" + simp.getTotalWeight() +
", " + simp.getOrder() + ")" + " ");
        }
        System.out.println("");
    }

    public List<String> getFinalPath() {
        return this.finalPath;
    }

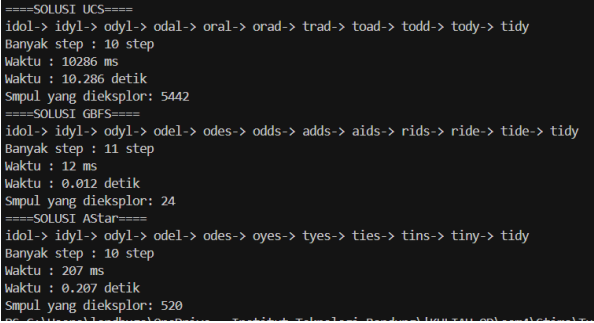
    public void printResult() {
        System.out.println("====SOLUSI AStar====");
        if(this.isfound){
            String hasil = String.join("-> ", this.finalPath);
            System.out.println(hasil);
        }else{
            System.out.println("Tidak ada solusi");
        }
        System.out.println("Banyak step : " + this.finalPath.size() + "
step");
        System.out.println("Waktu : " + this.time + " ms");
        System.out.println("Waktu : " + (float) this.time / 1000 + "
detik");
        System.out.println("Smpul yang dieksplor: " + this.n_explored);
    }
}

```

4. Uji Coba

Untuk uji coba, terdapat 2 kamus data yang sudah di parsing ke dalam map dalam program yaitu 'asisten' dan 'default'. Secara kelengkapan, kamus 'default' memiliki sekitar 370 ribu sedangkan 'asisten' memiliki sekitar 80 ribu kata.

| | | |
|---|---|--|
| 1 | <p>Kata awal : poop Kata akhir : look -default -asisten : Mau pakai kamus folder apa? default</p> <p>Mau Pakai algoritma apa? 1. UCS 2. Greedy-Best First Search 3. AStar 4. Semuanya 4</p> | <pre> =====SOLUSI UCS===== poop-> loop-> look Banyak step : 2 step Waktu : 134 ms Waktu : 0.134 detik Smpul yang dieksplor: 60 =====SOLUSI GBFS===== poop-> loop-> look Banyak step : 2 step Waktu : 6 ms Waktu : 0.006 detik Smpul yang dieksplor: 3 =====SOLUSI AStar===== poop-> loop-> look Banyak step : 2 step Waktu : 4 ms Waktu : 0.004 detik Smpul yang dieksplor: 4 </pre> |
| 2 | <p>Kata awal : recap Kata akhir : timer -default -asisten : Mau pakai kamus folder apa? default</p> <p>Mau Pakai algoritma apa? 1. UCS 2. Greedy-Best First Search 3. AStar 4. Semuanya 4</p> | <pre> =====SOLUSI UCS===== recap-> remap-> reman-> leman-> liman-> limen-> limer-> timer Banyak step : 7 step Waktu : 7457 ms Waktu : 7.457 detik Smpul yang dieksplor: 3699 =====SOLUSI GBFS===== recap-> remap-> reman-> teman-> toman-> tomas-> tomes-> times-> timer Banyak step : 8 step Waktu : 41 ms Waktu : 0.041 detik Smpul yang dieksplor: 14 =====SOLUSI AStar===== recap-> remap-> reman-> leman-> liman-> limen-> limer-> timer Banyak step : 7 step Waktu : 150 ms Waktu : 0.15 detik Smpul yang dieksplor: 64 </pre> |
| 3 | <p>Kata awal : hello Kata akhir : world -default -asisten : Mau pakai kamus folder apa? default</p> <p>Mau Pakai algoritma apa? 1. UCS 2. Greedy-Best First Search 3. AStar 4. Semuanya 4</p> | <pre> =====SOLUSI UCS===== hello-> hollo-> holly-> wolly-> wooly-> woold-> world Banyak step : 6 step Waktu : 8346 ms Waktu : 8.346 detik Smpul yang dieksplor: 4699 =====SOLUSI GBFS===== hello-> hollo-> holly-> wolly-> wooly-> woold-> world Banyak step : 6 step Waktu : 8 ms Waktu : 0.008 detik Smpul yang dieksplor: 8 =====SOLUSI AStar===== hello-> hollo-> holly-> wolly-> wooly-> woold-> world Banyak step : 6 step Waktu : 75 ms Waktu : 0.075 detik Smpul yang dieksplor: 64 </pre> |

| | | |
|---|---|--|
| 4 | <p>Kata awal : idol Kata akhir : tidy -default -asisten : Mau pakai kamus folder apa? default</p> <p>Mau Pakai algoritma apa? 1. UCS 2. Greedy-Best First Search 3. AStar 4. Semuanya 4</p> |  <pre> =====SOLUSI UCS===== idol-> idyl-> ody1-> odal-> oral-> orad-> trad-> toad-> todd-> tody-> tidy Banyak step : 10 step Waktu : 10286 ms Waktu : 10.286 detik Smpul yang dieksplor: 5442 =====SOLUSI GBFS===== idol-> idyl-> ody1-> odel-> odes-> odds-> adds-> aids-> rids-> ride-> tide-> tidy Banyak step : 11 step Waktu : 12 ms Waktu : 0.012 detik Smpul yang dieksplor: 24 =====SOLUSI AStar===== idol-> idyl-> ody1-> odel-> odes-> oyes-> tyes-> ties-> tins-> tiny-> tidy Banyak step : 10 step Waktu : 207 ms Waktu : 0.207 detik Smpul yang dieksplor: 520 </pre> |
| 5 | <p>Kata awal : island Kata akhir : better -default -asisten : Mau pakai kamus folder apa? default</p> <p>Mau Pakai algoritma apa? 1. UCS 2. Greedy-Best First Search 3. AStar 4. Semuanya 4</p> | <p>(Foto nya terlalu besar , untul full image ada di lampiran)</p> <pre> =====SOLUSI UCS===== island-> inland-> unland-> upland-> uplane-> upline-> spline-> saline-> satine-> satins-> matins-> maties-> mattes-> matter-> batter-> better Banyak step : 15 step Waktu : 9199 ms Waktu : 9.199 detik Smpul yang dieksplor: 5209 =====SOLUSI GBFS===== island-> inland-> inlaid-> inlaid-> onlaid-> unlaid-> unland-> unhand-> unhang-> unhung-> unsung-> unsing-> unking-> unkink-> unkind-> unkend-> unkent-> untent-> intent-> intens-> inters-> enters-> esters-> eaters-> caters-> paters-> peters-> meters-> deters-> defers-> refers-> refels-> refell-> refeel-> refeed-> reseed-> rested-> bested-> betted-> better Banyak step : 39 step Waktu : 562 ms Waktu : 0.562 detik Smpul yang dieksplor: 523 =====SOLUSI AStar===== island-> inland-> unland-> upland-> uplane-> upline-> spline-> saline-> satine-> satins-> matins-> maties-> mattes-> matter-> batter-> better Banyak step : 15 step Waktu : 6404 ms Waktu : 6.404 detik Smpul yang dieksplor: 4851 </pre> |

| | | |
|---|--|---|
| 6 | Kata awal : water Kata akhir : phone -default -asisten : Mau pakai kamus folder apa? default Mau Pakai algoritma apa? 1. UCS 2. Greedy-Best First Search 3. AStar 4. Semuanya 4 | <pre> =====SOLUSI UCS===== water-> pater-> poter-> porer-> pores-> porns-> poons-> phons-> phone Banyak step : 8 step Waktu : 23.647 ms Smpul yang dieksplor: 11214 =====SOLUSI GBFS===== water-> pater-> patee-> patte-> paste-> passe-> paise-> paine-> prine-> prone-> phone Banyak step : 10 step Waktu : 71 ms Smpul yang dieksplor: 29 =====SOLUSI AStar===== water-> pater-> pates-> pares-> pores-> porns-> poons-> phons-> phone Banyak step : 8 step Waktu : 18.346 ms Smpul yang dieksplor: 11558 </pre> |
|---|--|---|

5. Analisis

5.1. Optimalitas

| Percobaan | UCS | GBFS | A* |
|-----------|--------|--------------|--------|
| 1 | Unggul | Unggul | Unggul |
| 2 | Unggul | Tidak Unggul | Unggul |
| 3 | Unggul | Unggul | Unggul |
| 4 | Unggul | Tidak Unggul | Unggul |
| 5 | Unggul | Tidak Unggul | Unggul |
| 6 | Unggul | Tidak Unggul | Unggul |

Dari ketiga algoritma tersebut. Terlihat jelas bahwa algoritma yang paling optimal dalam mencari langkah terpendek adalah algoritma UCS dan A*. Dari keenam uji coba yang telah saya lakukan, algoritma UCS dan A* selalu mendapatkan hasil terpendek (step terkecil). Terbukti pada uji coba nomor 2,4,5,6. Dari bukti hasil percobaan diatas juga membuktikan bahwa heuristik pada A* yang saya pakai admissible karena step terkecil yang dihasilkan A* tidak *overestimate* step sebenarnya (step algoritma UCS).

5.2. Waktu Eksekusi

| Percobaan | UCS | GBFS | A* |
|-----------|--------------|--------|--------|
| 1 | Tidak Unggul | | Unggul |
| 2 | Tidak Unggul | Unggul | |
| 3 | Tidak Unggul | Unggul | |
| 4 | Tidak Unggul | Unggul | |

| | | | |
|---|--------------|--------|--|
| 5 | Tidak Unggul | Unggul | |
| 6 | Tidak Unggul | Unggul | |

Sangat terlihat jelas bahwa algoritma GBFS memiliki waktu eksekusi terkecil dari ketiga algoritma. Meskipun GBFS tidak dapat memberikan solusi optimal, GBFS bisa memberikan solusi dengan waktu tercepat. Ada 1 kasus yang menarik untuk saya bahas disini yaitu kasus uji coba nomor 5. GBFS mencari solusi tercepat dibanding yang lain yaitu hanya 0.562 detik, namun solusi yang diberikan sangat tidak optimal bila dibanding kasus uji coba lain yang mungkin hanya selisih 1 atau 2 dari solusi optimalnya. Hal ini bisa disebabkan karena GBFS yang hanya fokus pada unsur heuristik dan tidak digunakan informasi lain layaknya algoritma A*.

5.3. Memori

| Percobaan | UCS | GBFS | A* |
|-----------|--------------|--------|--------------|
| 1 | Tidak Unggul | Unggul | |
| 2 | Tidak Unggul | Unggul | |
| 3 | Tidak Unggul | Unggul | |
| 4 | Tidak Unggul | Unggul | |
| 5 | Tidak Unggul | Unggul | |
| 6 | | Unggul | Tidak Unggul |

Dari segi memori, algoritma UCS yang paling tidak unggul karena simpul yang dieksplorasi menggunakan algoritma UCS mayoritas paling besar (kecuali kasus unik percobaan 6). Hal ini karena UCS mengeksplorasi semua simpul dengan biaya semakin rendah secara bertahap, jadi algoritma ini dapat menghabiskan banyak memori terutama dalam ruang pencarian yang besar. Kembali lagi kepada GBFS yang sangat unggul dalam segi efisiensi memori karena simpul yang dieksplorasi oleh algoritma ini selalu paling sedikit dari keenam percobaan ini. Ada kasus unik ketika saya melakukan uji coba ke-6, halnya cukup mengejutkan karena pada kasus ini A* mengeksplorasi lebih banyak daripada UCS, yang bisa dikatakan UCS itu yang paling boros memori karena eksplorasi semua node. Hal ini terjadi ketika fungsi heuristik tidak memberikan estimasi yang akurat atau tidak membantu dalam memangkas simpul yang mahal, A* dapat mengeksplorasi lebih banyak node daripada (UCS).

BAB IV

KESIMPULAN

Algoritma Uniform Cost Search (UCS) dan A* Search menunjukkan konsistensi dalam mencari solusi optimal dengan jumlah langkah terkecil. Meskipun UCS memiliki kinerja yang lebih lambat dan lebih boros dalam penggunaan memori karena mengeksplorasi semua simpul dengan biaya semakin rendah secara bertahap, A* mampu memberikan solusi dengan keseimbangan yang baik antara optimalitas dan efisiensi waktu eksekusi serta penggunaan memori. Sementara itu, Greedy Best-First Search (GBFS) menunjukkan kinerja yang cepat dalam waktu eksekusi dan penggunaan memori yang sangat efisien, tetapi tidak menjamin solusi optimal dan dapat memiliki keterbatasan. Dalam konteks penggunaan fungsi heuristik pada A* Search, penting untuk memastikan bahwa heuristik tersebut admissible agar dapat memberikan estimasi yang akurat dan menghasilkan solusi optimal.

DAFTAR PUSTAKA

- [1]<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf> (Diakses Senin, 6 Mei 2024)
- [2]<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf> (Diakses Senin, 6 Mei 2024)
- [3]<https://www.geeksforgeeks.org/a-search-algorithm/> (Diakses Minggu, 5 Mei 2024).

LAMPIRAN

- Foto penuh Uji coba nomor 5

```

=====SOLUSI UCS=====
island-> inland-> unland-> upland-> uplane-> upline-> spline-> saline-> satine-> satins-> matins-> maties-> mattes-> matter-> batter-> better
Banyak step : 15 step
waktu : 9199 ms
waktu : 9.199 detik
Smpul yang dieksplor: 5209
=====SOLUSI GBFS=====
island-> inland-> inland-> inland-> onlaid-> unland-> unhand-> unhang-> unhung-> unsung-> unsing-> unking-> unkink-> unkind-> unkend-> unkent-> untent-> intent->
Intens-> inters-> enters-> esters-> eaters-> caters-> paters-> peters-> meters-> deters-> defers-> refers-> refels-> refell-> refeel-> refeed-> reseed-> rested-> bested->
batted-> better
Banyak step : 39 step
waktu : 562 ms
waktu : 0.562 detik
Smpul yang dieksplor: 523
=====SOLUSI AStar=====
island-> inland-> unland-> upland-> uplane-> upline-> spline-> saline-> satine-> satins-> matins-> maties-> mattes-> matter-> batter-> better
Banyak step : 15 step
waktu : 6804 ms
waktu : 6.804 detik
Smpul yang dieksplor: 4851

```

- Tabel Laporan

| Poin | Ya | Tidak |
|---|----|-------|
| Program berhasil dijalankan. | ✓ | |
| Program dapat menemukan rangkaian kata dari <i>start word</i> ke <i>end word</i> sesuai aturan permainan dengan algoritma UCS | ✓ | |
| Solusi yang diberikan pada algoritma UCS optimal | ✓ | |
| Program dapat menemukan rangkaian kata dari <i>start word</i> ke <i>end word</i> sesuai aturan permainan dengan algoritma <i>Greedy Best First Search</i> | ✓ | |
| Program dapat menemukan rangkaian kata dari <i>start word</i> ke <i>end word</i> sesuai aturan permainan dengan algoritma A* | ✓ | |
| Solusi yang diberikan pada algoritma A* optimal | ✓ | |
| [Bonus]: Program memiliki tampilan GUI | | ✓ |

- Pranala GitHub : https://github.com/miannetopokki/Tucil3_13522129

