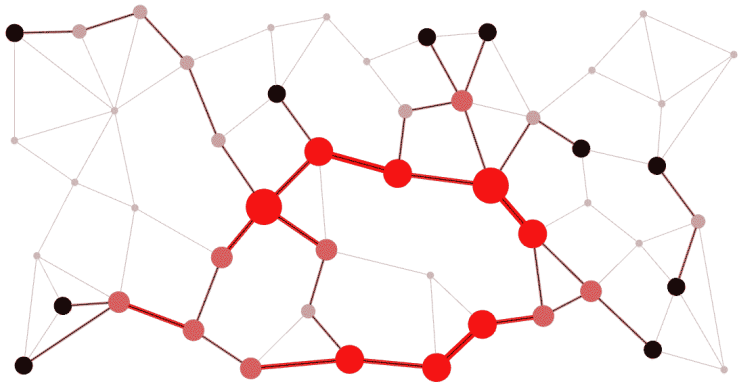
Tugas Kecil
IF2211 Strategi Algoritma
Penyelesaian Permainan Word Ladder Menggunakan Algoritma UCS, Greedy Best First Search, dan A*



Sumber: https://medium.com/generative-design/routing-with-graphs-5fb564b02a74

Oleh:

Muhammad Rasheed Qais Tandjung 13522158

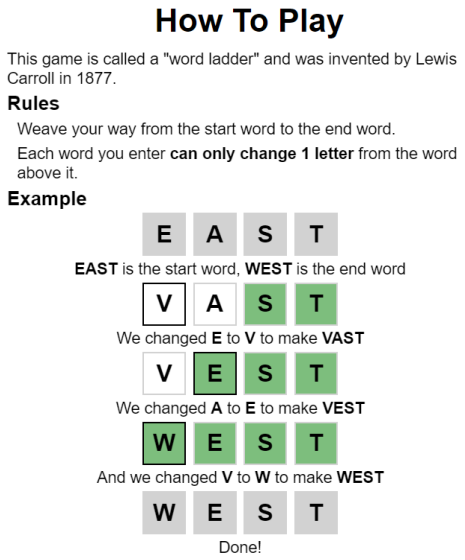
PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG

2024

Bab I

Deskripsi Tugas

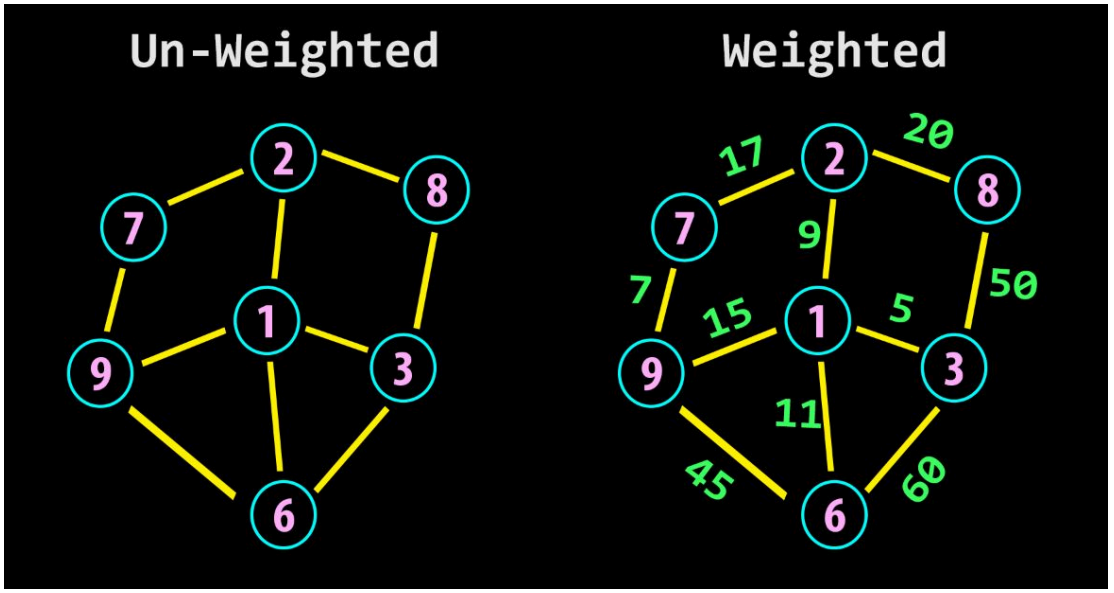
Word ladder (juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragrams, laddergrams, atau word golf) adalah salah satu permainan kata yang terkenal bagi seluruh kalangan. Word ladder ditemukan oleh Lewis Carroll, seorang penulis dan matematikawan, pada tahun 1877. Pada permainan ini,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



Gambar 1. Ilustrasi dan Peraturan Permainan Word Ladder
(Sumber: https://wordwormdormdork.com/

Bab II
Pencarian Solusi dengan Algoritma *UCS, GBFS, dan A**

2.1 Formulasi Masalah



Gambar 2. Ilustrasi struktur data graf
Sumber: https://simplesnippets.tech/what-is-a-graph-data-structure-important-graph-terms-properties/

Permasalahan mencari rute pendek antara dua kata pada permainan *Word Ladder* dapat diformulasikan sebagai masalah pencarian rute pada graf. Simpul pada graf ini adalah sebuah kata, dan tetangga dari simpul tersebut merupakan seluruh kata yang hanya berbeda satu huruf dengan kata tersebut pada kamus yang tersedia. Oleh karena itu, menyelesaikan masalah ini dapat menggunakan algoritma-algoritma yang umumnya digunakan untuk melakukan pencarian rute terpendek pada graf, seperti algoritma *Uniform-Cost Search (UCS), Greedy Best-First Search (GBFS),* ataupun *A**.

2.2 Pendekatan dengan Algoritma *Uniform-Cost Search* (UCS)

Algoritma UCS umumnya digunakan untuk pencarian rute terpendek pada graf berbobot. Namun pada permasalahan graf *Word Ladder* yang sudah disebutkan tidak terdapat sebuah bobot tertentu antara dua simpul, sehingga jumlah lompatan dari simpul akar ke simpul tertentu akan digunakan sebagai bobot total dari simpul tersebut. Menggunakan pendekatan tersebut, maka bobot antara dua simpul yang bertetangga akan selalu bernilai 1.

Konsep dari algoritma UCS adalah menyimpan seluruh simpul yang sudah ditemukan, dan pada setiap langkah algoritma akan memilih simpul yang memiliki bobot terkecil. Sehingga implementasi algoritma ini dapat memanfaatkan struktur data *priority*

queue untuk menyimpan seluruh simpul yang ingin dikunjungi. Seluruh simpul yang dimasukkan ke *queue* akan ditempatkan pada posisi yang tepat sesuai dengan bobotnya, sehingga setiap aksi *dequeue* pada *queue* akan selalu menghasilkan simpul dengan bobot terkecil.

Untuk sebuah kata *start* dan sebuah kata *end*, algoritma UCS akan mencari rute terpendek antara dua kata tersebut melalui prosedur berikut:

1. Inisialisasi *priority queue* untuk menyimpan simpul-simpul yang ingin dikunjungi, serta sebuah *hash map minCost* untuk menyimpan simpul yang sudah pernah dikunjungi dan bobot terkecilnya.
2. Lakukan pengecekan *start* dan *end* terlebih dahulu. Jika kedua kata sama, maka algoritma selesai. Jika kedua kata berbeda, masukkan *start* ke *priority queue*. Simpul *start* akan menjadi simpul pertama yang dilakukan ekspansi.
3. Lakukan *dequeue* pada *queue* sehingga didapatkan sebuah simpul *current*.
4. Untuk setiap simpul *neighbor* yang bertetangga dengan *current*, cek simpul tersebut.
    a. Jika *neighbor* = *end*, maka rute terpendek ditemukan dan algoritma selesai.
    b. Jika *neighbor* sudah pernah ditemukan dengan *cost* yang lebih kecil, lewati simpul tersebut.
    c. Jika *neighbor* sudah pernah ditemukan dengan *cost* yang lebih besar, masukkan *neighbor* ke *queue* dan modifikasi bobot minimum *neighbor* pada *hash map minCost*.
    d. Selain itu, masukkan *neighbor* ke *queue* dan tuliskan bobotnya pada *hash map minCost*.
5. Kembali ke langkah 3 sampai ditemukan *neighbor* = *end* atau sampai *queue* kosong.
    a. Jika ditemukan *neighbor* = *end*, maka ditemukan rute dari *start* ke *end* dengan jumlah lompatan minimum.
    b. Jika *queue* kosong, maka tidak ditemukan rute dari *start* ke *end*, sehingga tidak terdapat solusi untuk rute tersebut.

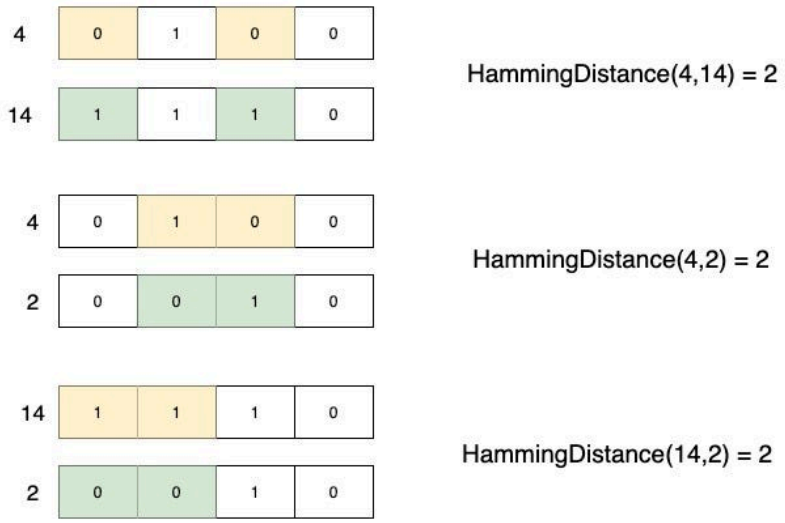
Hal yang perlu dicatat terkait penerapan algoritma UCS pada permasalahan ini adalah bahwa untuk setiap simpul, bobot dari simpul tersebut adalah jumlah lompatan dari akar ke simpul tersebut. Simpul yang dikunjungi selanjutnya akan selalu berupa simpul yang memiliki kedalaman terkecil pada pohon pencarian dari simpul akar, dan algoritma akan selalu mengecek seluruh simpul pada kedalaman k sebelum melanjutkan pencarian ke kedalaman $k + 1$. Dengan kata lain, algoritma UCS pada permasalahan ini sama persis dengan algoritma BFS biasa.

2.3 Pendekatan dengan Algoritma *Greedy Best-First Search* (GBFS)

Algoritma GBFS akan mencari rute antara dua simpul pada graf dengan memanfaatkan sebuah fungsi heuristik. Pada setiap langkah, algoritma GBFS akan mengunjungi simpul dengan nilai fungsi heuristik paling optimal. Perbedaan utama terletak

pada batasan bahwa algoritma GBFS tidak dapat melakukan *backtracking*, sehingga setelah sebuah simpul sudah dipilih, algoritma tidak dapat mundur ke simpul sebelumnya. Oleh karena itu, tidak diperlukan struktur data *queue* pada algoritma ini, karena tidak perlu dilakukan penyimpanan atas simpul-simpul alternatif yang ingin dikunjungi. Namun tetap diperlukan struktur data *hash map* yang menyimpan simpul-simpul yang sudah pernah dikunjungi, agar algoritma tidak masuk ke dalam *infinite loop*.

Fungsi heuristik h(n) yang digunakan untuk menentukan nilai sebuah simpul merupakan komponen yang penting, sebab pemilihan fungsi yang tepat akan berpengaruh besar pada optimalitas dan kelengkapan solusi algoritma. Pada permasalahan ini, akan digunakan *hamming distance* dari kata A ke kata tujuan sebagai fungsi heuristik. *Hamming distance* dari dua buah *string* A dan B dengan panjang yang sama didefinisikan sebagai jumlah posisi yang memiliki nilai yang berbeda pada A dan B.



Gambar 3. Ilustrasi fungsi hamming distance yang akan digunakan sebagai fungsi heuristik
Sumber: https://medium.com/geekculture/total-hamming-distance-problem-1b74decd71c9

Untuk sebuah kata *start* dan sebuah kata *end*, algoritma GBFS akan mencari rute terpendek antara dua kata tersebut melalui prosedur berikut:

1. Inisialisasi *hash map seen* yang akan menyimpan seluruh simpul yang sudah pernah dikunjungi.
2. Lakukan pengecekan *start* dan *end* terlebih dahulu. Jika kedua kata sama, maka algoritma selesai. Jika kedua kata berbeda, *start* akan diperlakukan sebagai simpul *current* pertama yang dikunjungi dan diekspansi.
3. Untuk setiap simpul *neighbor* yang bertetangga dengan *current*, cari tetangga *minimumNeighbor,* yaitu tetangga yang menghasilkan nilai fungsi heuristik terkecil.
4. Berdasarkan simpul *minimumNeighbor* yang dipilih, akan terdapat beberapa kasus:

   a. Jika *minimumNeighbor* sudah pernah dikunjungi, maka algoritma dihentikan agar tidak masuk ke *infinite loop*. Keluarkan hasil bahwa rute solusi gagal ditemukan.
   b. Jika *minimumNeighbor* = *end*, maka rute solusi ditemukan, sehingga algoritma selesai.
   c. Selain itu, nyatakan *minimumNeighbor* sebagai simpul selanjutnya yang ingin dikunjungi dan diekspansi, lalu kembali ke langkah 3.
5. Lakukan langkah 3 dan 4 sampai salah satu kasus dari langkah 4 terjadi.

Yang perlu digaris bawahi dari implementasi ini adalah penentuan batasan bahwa algoritma GBFS tidak melakukan *backtracking*. Terdapat jenis implementasi GBFS lain yaitu *open-list GBFS* yang dapat melakukan *backtracking* dan menyimpan simpul-simpul yang tidak dikunjungi dalam sebuah *queue*, seperti pada algoritma UCS dan A*. Terdapat implementasi *open-list GBFS* pada tugas kecil ini yang dapat digunakan sebagai perbandingan, namun algoritma GBFS utama yang digunakan adalah algoritma GBFS tanpa *backtracking* yang sudah disebutkan di atas, sesuai dengan algoritma yang diajarkan pada mata kuliah IF2211.

2.4 Pendekatan dengan Algoritma A*

Algoritma A* dapat dianggap sebagai gabungan dari algoritma UCS dan GBFS yang sudah disebutkan. Pada algoritma ini, setiap simpul yang ingin dikunjungi akan disimpan pada sebuah *queue*, dan simpul selanjutnya yang akan dikunjungi dan diekspansi merupakan simpul yang menghasilkan nilai fungsi f(n) terkecil. Fungsi f(n) didefinisikan sebagai

$$f(n) = g(n) + h(n),$$

dengan g(n) merupakan total bobot untuk mencapai simpul n dari simpul akar seperti pada algoritma UCS, dan h(n) merupakan fungsi heuristik untuk simpul n seperti pada algoritma GBFS. Sehingga pada setiap langkah, algoritma A* akan memilih simpul yang menghasilkan nilai heuristik paling optimal, namun tidak terlalu jauh dari simpul akar.

Untuk sebuah kata *start* dan sebuah kata *end*, algoritma GBFS akan mencari rute terpendek antara dua kata tersebut melalui prosedur berikut:

1. Inisialisasi *priority queue* untuk menyimpan simpul-simpul yang ingin dikunjungi, serta sebuah *hash map minCost* untuk menyimpan simpul yang sudah pernah dikunjungi dan bobot terkecilnya.
2. Lakukan pengecekan *start* dan *end* terlebih dahulu. Jika kedua kata sama, maka algoritma selesai. Jika kedua kata berbeda, masukkan *start* ke *priority queue*. Simpul *start* akan menjadi simpul pertama yang dilakukan ekspansi.
3. Lakukan *dequeue* pada *queue* sehingga didapatkan sebuah simpul *current*.
   a. Jika solusi sementara *tempSolution* sudah pernah ditemukan sebelumnya, cek apakah nilai f(n) pada *current* sudah melewati nilai f(n) *tempSolution*. Jika sudah, maka program akan selesai dan menyatakan *tempSolution* sebagai solusi akhir.

4. Untuk setiap simpul *neighbor* yang bertetangga dengan *current*, cek simpul tersebut.
    a. Jika *neighbor* = *end*, maka lakukan pengecekan tersebut:
        i. Jika *tempSolution* sudah pernah didapatkan sebelumnya, cek apakah *neighbor* memiliki nilai f(n) yang lebih kecil. Jika iya, maka nyatakan *neighbor* sebagai *tempSolution* terbaru. Jika tidak, maka abaikan *neighbor*.
    b. Jika *neighbor* sudah pernah ditemukan dengan *cost* yang lebih kecil, abaikan *neighbor*.
    c. Jika *neighbor* sudah pernah ditemukan dengan *cost* yang lebih besar, masukkan *neighbor* ke *queue* dan modifikasi bobot minimum *neighbor* pada *hash map minCost*.
    d. Selain itu, masukkan *neighbor* ke *queue* dan tuliskan bobotnya pada *hash map minCost*.
5. Kembali ke langkah 3 sampai program berhenti di langkah 4.a.1, atau sampai *queue* kosong.
    a. Jika program berhenti di langkah 4.a.1, maka ditemukan rute solusi dengan bobot minimum.
    b. Jika *queue* kosong, maka tidak ditemukan rute dari *start* ke *end*, sehingga tidak terdapat solusi untuk rute tersebut.

2.5 Analisis Kasus

1. Definisi *f(n)* dan *g(n)*
   Pada setiap algoritma yang sudah disebutkan, pemilihan simpul selanjutnya yang akan dikunjungi akan menggunakan fungsi f(n), dengan definisi

   $$f(n) = g(n) + h(n),$$

   dengan g(n) merupakan total bobot untuk mencapai simpul n dari simpul akar dan h(n) merupakan fungsi heuristik untuk simpul n.

   - Pada algoritma UCS, f(n) = g(n), karena pemilihan simpul selanjutnya berdasarkan simpul dengan bobot terkecil.
   - Pada algoritma GBFS, f(n) = h(n), karena pemilihan simpul selanjutnya berdasarkan simpul dengan nilai heuristik terkecil.
   - Pada algoritma A*, f(n) = g(n) + h(n), sehingga A* dapat dianggap sebagai gabungan dari algoritma UCS dan GBFS.

2. Apakah heuristik yang digunakan pada algoritma A* *admissible*?
   Definisi fungsi heuristik yang *admissible* pada algoritma A* adalah fungsi yang nilai h(n) dapat dipastikan selalu lebih kecil atau sama dengan nilai bobot sesungguhnya h*(n) dari simpul n ke simpul tujuan. Atau dalam kata lain, $h(n) \leq h^*(n)$ untuk semua simpul.

Pada algoritma A* yang diimplementasikan, fungsi heuristik yang digunakan merupakan *hamming distance* dari simpul A ke simpul B, seperti yang sudah dijelaskan pada penjelasan algoritma GBFS di Bagian 3.3. Fungsi h(n) ini akan selalu bernilai lebih kecil dari h*(n), karena setiap lompatan hanya akan merubah satu huruf. Rute dari simpul A ke simpul B dengan *hamming distance* N akan membutuhkan paling sedikit N penggantian huruf, sehingga jumlah lompatan dari A ke B akan selalu bernilai N atau lebih besar, dan fungsi heuristik ini dapat dianggap sebagai fungsi yang *admissible*.

3. Pada kasus word ladder, apakah algoritma UCS sama dengan BFS? (dalam artian urutan node yang dibangkitkan dan path yang dihasilkan sama)
   Seperti yang sudah dijelaskan pada Bagian 3.2, graf yang dihasilkan pada permasalahan *word ladder* adalah graf yang tidak berbobot, sehingga bobot dari simpul A ke simpul B dapat dianggap sebagai jumlah lompatan dari A ke B.

   Jumlah lompatan ini sama dengan kedalaman simpul B pada pohon pencarian yang berakar di simpul A. Algoritma UCS akan selalu memilih untuk mengunjungi simpul dengan kedalaman yang lebih rendah terlebih dahulu (karena bobot sama dengan kedalaman), sehingga pencarian yang dilakukan akan sama persis seperti pencarian menggunakan algoritma BFS.

4. Secara teoritis, apakah algoritma A* lebih efisien dibandingkan dengan algoritma UCS pada kasus word ladder?
   Algoritma A* secara statistik dapat diekspektasikan untuk lebih sering mendapatkan solusi secara lebih efisien karena memanfaatkan fungsi heuristik untuk menebak simpul-simpul yang lebih meyakinkan. Namun secara teori, algoritma A* tidak dipastikan untuk selalu lebih efisien dari UCS, karena algoritma A* tidak dapat langsung berhenti ketika menemukan tujuan karena adanya kemungkinan didapatkan jalur yang lebih pendek, sedangkan ketika UCS menemukan tujuan dapat langsung berhenti.

5. Secara teoritis, apakah algoritma Greedy Best First Search menjamin solusi optimal untuk persoalan word ladder?
   Algoritma GBFS tidak dapat menjamin solusi optimal untuk permasalahan ini, karena algoritma ini akan mengorbankan *completeness* dalam mencari solusi optimal demi minimalisasi waktu dan memori yang terpakai. Algoritma GBFS akan mendapatkan solusi dengan waktu yang jauh lebih cepat dan memori lebih dikit, namun dengan harga solusi yang tidak optimal.

Bab III
Source Code Algoritma dalam Bahasa *Java*

A. class Node

```
public class Node implements Comparable<Node> {
    public String word;
    public String path;
    public int cost;
    public int heuristic;
    public boolean useCost;

    public int GetHammingDistance(String target) {
        int distance = 0;
        for (int i = 0; i < target.length(); i++) {
            if (word.charAt(i) != target.charAt(i)) {
                distance++;
            }
        }
        return distance;
    }

    public Node(String word, String path, int cost, String GoalWord, String
algorithm) {
        this.word = word;
        this.path = path;

        if (algorithm.equals("UCS")) {
            this.cost = cost;
            this.heuristic = 0;
            this.useCost = true;
        } else if (algorithm.equals("GBFS") ||
algorithm.equals("GBFSWithBacktrack")) {
            this.cost = cost;
            this.heuristic = GetHammingDistance(GoalWord);
            this.useCost = false;
        } else if (algorithm.equals("A*")) {
            this.cost = cost;
            this.heuristic = GetHammingDistance(GoalWord);
            this.useCost = true;
```

```
        }
    }

    public int GetPriority() {
        return this.cost + this.heuristic;
    }

    @Override
    public int compareTo(Node other) {
        int ThisPriority, OtherPriority;
        if (this.useCost) {
            ThisPriority = this.cost + this.heuristic;
            OtherPriority = other.cost + other.heuristic;
        } else {
            ThisPriority = this.heuristic;
            OtherPriority = other.heuristic;
        }

        return Integer.compare(ThisPriority, OtherPriority);
    }
}
```

Kelas ini merupakan kelas *wrapper* yang berperan sebagai simpul untuk sebuah kata dalam graf. Kelas ini akan menyimpan kata, serta nilai prioritas yaitu *cost* dan *heuristic* yang akan digunakan untuk pengurutan dalam *priority queue*.

B. class Result

```
public class Result {
    public boolean found;
    public long time;
    public long memory;
    public int count;
    public Node node;

    public Result(boolean found, long time, long memory, int count, Node
node) {
        this.found = found;
        this.time = time;
        this.memory = memory;
```

```
        this.count = count;
        this.node = node;
    }
}
```

Kelas ini merupakan kelas yang akan dikembalikan oleh algoritma UCS, GBFS, dan A*. Kelas ini menyimpan waktu, jumlah simpul yang ditelusuri, dan simpul tujuan yang ditemukan di graf.

C. class UCS

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;

public class UCS {
    public static Result SearchUCS(String start, String end, Map<String,
List<String>> wordMap) {
        // Return true if the start and end words are the same
        if (start.equals(end)) {
            return new Result(true, 0, 0, 0, new Node(start, start, 0, end,
"UCS"));
        }

        // Track starting time
        long startTime = System.nanoTime();

        // Create a priority queue
        PriorityQueue<Node> queue = new PriorityQueue<>();

        // Create a minimum cost map
        Map<String, Integer> minCost = new HashMap<>();

        // Create node for starting word
        Node startNode = new Node(start, start, 0, end, "UCS");

        // Add the start node to the queue
        queue.add(startNode);
```

```
        // Loop until queue is empty
        int count = 0;
        while (!queue.isEmpty()) {
            // Get the node with the lowest cost
            Node current = queue.poll();

            // Get the neighbors of the current node
            List<String> neighbors = wordMap.get(current.word);

            // For each neighbor
            if (neighbors == null) {
                continue;
            }

            for (String neighbor : neighbors) {
                // Skip if the neighbor has already been visited with a
lower cost
                if (minCost.containsKey(neighbor)) {
                    if (current.cost >= minCost.get(neighbor)) {
                        continue;
                    }
                }

                // Increment the count
                count++;

                // If the neighbor is the end word
                if (neighbor.equals(end)) {
                    // Track the ending time
                    long endTime = System.nanoTime();

                    // Calculate the time taken
                    long time = endTime - startTime;

                    // Create a new node
                    Node newNode = new Node(neighbor, current.path + " -> "
+ neighbor, current.cost + 1, end, "UCS");

                    // Return the result
                    return new Result(true, time, 0, count, newNode);
                }
```

```
                // Track the minimum cost
                minCost.put(neighbor, current.cost);

                // Create a new node
                Node newNode = new Node(neighbor, current.path + " -> " +
neighbor, current.cost + 1, end, "UCS");

                // Add the new node to the queue
                queue.add(newNode);
            }
        }

        // Calculate the time taken
        long time = System.nanoTime() - startTime;
        return new Result(false, time, 0, count, null);
    }
}
```

Kelas ini merupakan kelas yang hanya memiliki satu metode statik, yaitu metode yang akan menjalankan pencarian rute dengan algoritma UCS. Penjelasan terkait algoritma UCS terdapat di Bagian 2.2.

D. class GBFS

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;

public class GBFS {
    public static Result SearchGBFS(String start, String end, Map<String,
List<String>> wordMap) {
        // Return true if the start and end words are the same
        if (start.equals(end)) {
            return new Result(true, 0, 0, 0, new Node(start, start, 0, end,
"GBFS"));
        }

        // Track starting time
```

```
        long startTime = System.nanoTime();

        // Create a seen map
        Map<String, Boolean> seen = new HashMap<>();

        // Loop until end goal is found
        int count = 0;
        Node node = new Node(start, start, 0, end, "GBFS");

        while (true) {
            // Get the neighbors of the current node
            List<String> neighbors = wordMap.get(node.word);

            // If there are no neighbors
            if (neighbors == null) {
                return new Result(false, System.nanoTime() - startTime, 0,
count, null);
            }

            // Find the neighbor with the lowest heuristic
            int minHeuristic = Integer.MAX_VALUE;
            Node nextNode = null;
            for (String neighbor : neighbors) {
                Node neighborNode = new Node(neighbor, node.path + " -> " +
neighbor, node.cost + 1, end, "GBFS");

                if (neighborNode.heuristic < minHeuristic) {
                    minHeuristic = neighborNode.heuristic;
                    nextNode = neighborNode;
                }
            }

            // If next node has already been seen
            if (seen.containsKey(nextNode.word)) {
                return new Result(false, System.nanoTime() - startTime, 0,
count, null);
            }

            // Increment the count
            count++;
```

```
            // If the neighbor is the end word
            if (nextNode.word.equals(end)) {
                // Track the ending time
                long endTime = System.nanoTime();

                // Calculate the time taken
                long time = endTime - startTime;

                // Return the result
                return new Result(true, time, 0, count, nextNode);
            }

            // Mark the current node as seen
            seen.put(nextNode.word, true);

            // Move to the next node
            node = nextNode;
        }
    }
}
```

Kelas ini merupakan kelas yang hanya memiliki satu metode statik, yaitu metode yang akan menjalankan pencarian rute dengan algoritma GBFS. Penjelasan terkait algoritma GBFS terdapat di Bagian 2.2.

E. Class AStar

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;

public class AStar {
    public static Result SearchAStar(String start, String end,
Map<String,List<String>> wordMap) {
        // Return true if the start and end words are the same
        if (start.equals(end)) {
            return new Result(true, 0, 0, 0, new Node(start, start, 0, end,
"A*"));
        }
```

```
        // Track starting time
        long startTime = System.nanoTime();

        // Create a priority queue
        PriorityQueue<Node> queue = new PriorityQueue<>();

        // Create a minimum cost map
        Map<String, Integer> minCost = new HashMap<>();

        // Create node for starting word
        Node startNode = new Node(start, start, 0, end, "A*");

        // Add the start node to the queue
        queue.add(startNode);

        // Loop until queue is empty
        int count = 0;
        boolean found = false;
        Node result = null;
        while (!queue.isEmpty()) {
            // Get the node with the lowest cost
            Node current = queue.poll();

            // If found and current node cost is higher than result node
cost
            if (found && (current.cost > result.cost)) {
                return new Result(true, System.nanoTime() - startTime, 0,
count, result);
            }

            // Get the neighbors of the current node
            List<String> neighbors = wordMap.get(current.word);

            // For each neighbor
            if (neighbors == null) {
                continue;
            }

            for (String neighbor : neighbors) {
                // Skip if the neighbor has already been visited with a
```

```
lower cost
                if (minCost.containsKey(neighbor)) {
                    if (current.cost >= minCost.get(neighbor)) {
                        continue;
                    }
                }

                // Increment the count
                count++;

                // If the neighbor is the end word
                if (neighbor.equals(end)) {
                    // // If not found or found with lower cost
                    if (!found || (found && result.cost < current.cost)) {
                        // Create a new node
                        Node newNode = new Node(neighbor, current.path + "
-> " + neighbor, current.cost + 1, end, "A*");

                        // Set result node to new node
                        result = newNode;
                    }

                    // Set found to true
                    found = true;



                    // Track the ending time
                    // long endTime = System.nanoTime();

                    // // Calculate the time taken
                    // long time = endTime - startTime;

                    // // Calculate the memory used
                    // // int size = queue.size();
                    // // long memoryUsage = size * Node.BYTES;

                    // // Create a new node
                    // Node newNode = new Node(neighbor, current.path + " ->
" + neighbor, current.cost + 1, end, "A*");

                    // // Return the result
```

```
                    // return new Result(true, time, 0, count, newNode);
                }

                // Track the minimum cost
                minCost.put(neighbor, current.cost);

                // Create a new node
                Node newNode = new Node(neighbor, current.path + " -> " +
neighbor, current.cost + 1, end, "A*");

                // Add the new node to the queue
                queue.add(newNode);
            }
        }

        // Calculate the time taken
        long time = System.nanoTime() - startTime;
        return new Result(found, time, 0, count, result);
    }
}
```

Kelas ini merupakan kelas yang hanya memiliki satu metode statik, yaitu metode yang akan menjalankan pencarian rute dengan algoritma A*. Penjelasan terkait algoritma A* terdapat di Bagian 2.2.

F. class GBFSWithBacktrack

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;

public class GBFSWithBacktrack {
    public static Result SearchGBFSWithBacktrack(String start, String end,
Map<String, List<String>> wordMap) {
        // Return true if the start and end words are the same
        if (start.equals(end)) {
            return new Result(true, 0, 0, 0, new Node(start, start, 0, end,
"GBFSWithBacktrack"));
```

```
        }

        // Track starting time
        long startTime = System.nanoTime();

        // Create a priority queue
        PriorityQueue<Node> queue = new PriorityQueue<>();

        // Create a seen map
        Map<String, Boolean> seen = new HashMap<>();

        // Create node for starting word
        Node startNode = new Node(start, start, 0, end,
"GBFSWithBacktrack");

        // Add the start node to the queue
        queue.add(startNode);

        // Loop until queue is empty
        int count = 0;
        while (!queue.isEmpty()) {
            // Get the node with the lowest cost
            Node current = queue.poll();

            // If current node is seen, return with not found to prevent an
infinite loop
            // if (seen.containsKey(current.word)) {
            //     return new Result(false, System.nanoTime() - startTime,
0, count, null);
            // }

            // Get the neighbors of the current node
            List<String> neighbors = wordMap.get(current.word);

            // For each neighbor
            if (neighbors == null) {
                continue;
            }

            for (String neighbor : neighbors) {
                // Skip if node has already been seen
```

```
                if (seen.containsKey(neighbor)) {
                    continue;
                }

                // Increment the count
                count++;

                // Create a new node for the neighbor
                Node neighborNode = new Node(neighbor, current.path + " -> "
+ neighbor, current.cost + 1, end, "GBFSWithBacktrack");

                // If the neighbor is the end word
                if (neighbor.equals(end)) {
                    // Track the ending time
                    long endTime = System.nanoTime();

                    // Calculate the time taken
                    long time = endTime - startTime;

                    // Return the result
                    return new Result(true, time, 0, count, neighborNode);
                }

                // Mark the current node as seen
                seen.put(current.word, true);

                // Add the neighbor node to the queue
                queue.add(neighborNode);
            }
        }

        // Return false if the path is not found
        return new Result(false, System.nanoTime() - startTime, 0, count,
null);
    }
}
```

Kelas ini merupakan kelas yang hanya memiliki satu metode statik, yaitu metode yang akan menjalankan pencarian rute dengan algoritma GBFS versi *open-list*. Algoritma ini bersifat hampir sama persis seperti UCS dan A*, kecuali tidak memperhitungkan *cost* saja dan prioritas hanya berdasarkan fungsi heuristik.

G. class Main

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Scanner;

import java.util.LinkedList;
import java.util.Queue;

public class Main {
   public static void main(String[] args) {
        // Variables
        String filename = "oracle.txt";
        String start;
        String end;
        // String algorithm;

        // Get user input
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter starting word: ");
        start = scanner.nextLine().toLowerCase();

        System.out.print("Enter ending word: ");
        end = scanner.nextLine().toLowerCase();

        boolean correct = false;
        String algorithm = "";
        do {
            System.out.println();
            System.out.println("Enter which algorithm to use: ");
            System.out.println("1. UCS");
            System.out.println("2. GBFS");
            System.out.println("3. A*");
            System.out.println("4. GBFSWithBacktrack (Unoptimal)");

            System.out.println();
```

```
            System.out.print("Choice: ");
            String choice = scanner.nextLine();

            switch (choice) {
                case "1":
                    System.out.println("Using UCS algorithm.");
                    algorithm = "UCS";
                    correct = true;
                    break;
                case "2":
                    System.out.println("Using GBFS algorithm.");
                    algorithm = "GBFS";
                    correct = true;
                    break;
                case "3":
                    System.out.println("Using A* algorithm.");
                    algorithm = "A*";
                    correct = true;
                    break;
                case "4":
                    System.out.println("Using GBFSWithBacktrack
algorithm.");
                    algorithm = "GBFSWithBacktrack";
                    correct = true;
                    break;
                default:
                    System.out.println("Invalid choice. Please try again.");
                    break;
            }
        } while (!correct);
        System.out.println();
        scanner.close();

        // Read file and store data to map
        Map<String, List<String>> wordMap = new HashMap<>();

        // Get map of word graph
        try (BufferedReader br = new BufferedReader(new
FileReader(filename))) {
            String line;
            while ((line = br.readLine()) != null) {
```

```
                // Split the line by spaces
                String[] parts = line.split(" ");

                // The first part is the key, and the rest are values
                String key = parts[0];
                List<String> values = new ArrayList<>();
                for (int i = 1; i < parts.length; i++) {
                    values.add(parts[i]);
                }

                // Put key-value pair into the map
                wordMap.put(key, values);
            }
        } catch (IOException e) {
            // Handle IOException
            e.printStackTrace();
        }

        Result res;

        if (algorithm.equals("UCS")) {
            res = UCS.SearchUCS(start, end, wordMap);
        } else if (algorithm.equals("GBFS")) {
            res = GBFS.SearchGBFS(start, end, wordMap);
        } else if (algorithm.equals("GBFSWithBacktrack")) {
            res = GBFSWithBacktrack.SearchGBFSWithBacktrack(start, end,
wordMap);
        } else  {
            res = AStar.SearchAStar(start, end, wordMap);
        }

        // Print the result
        System.out.println("Algorithm: " + algorithm);
        long time_ms = res.time / 1000000;
        if (res.found) {
            System.out.println("Time: " + time_ms + " ms");
            System.out.println("Path: " + res.node.path);
            System.out.println("Degrees: " + res.node.cost);
            System.out.println("Nodes visited: " + res.count);
        } else {
            System.out.println("Path not found.");
```

```
            System.out.println("Time: " + time_ms + " ms");
            System.out.println("Nodes visited: " + res.count);
        }
        System.out.println();
    }
}
```

Method ini merupakan *entrypoint* dari program. Kelas ini pada awal akan melakukan *loading* terkait *database* kata yang sudah di-*convert* sebelumnya menjadi bentuk struktur data graf. Program akan meminta input kata awal, kata tujuan, dan algoritma, dan menjalankan metode pencarian sesuai dengan algoritma yang dipilih pengguna.

Bab IV
Hasil Pengujian

4.1. Algoritma UCS

4.1.1 Hello, World

Input :

```
Enter starting word: Hello
Enter ending word: World

Enter which algorithm to use:
1. UCS
2. GBFS
3. A*
4. GBFSWithBacktrack (Unoptimal)

Choice: 1
Using UCS algorithm.
```

Output :

```
Choice: 1
Using UCS algorithm.

Algorithm: UCS
Time: 16 ms
Path: hello -> hells -> wells -> weals -> weald -> woald -> world
Degrees: 6
Nodes visited: 2216
```

4.1.2 Slow → Loop

Input :

```
Enter starting word: slow
Enter ending word: loop

Enter which algorithm to use:
1. UCS
2. GBFS
3. A*
4. GBFSWithBacktrack (Unoptimal)

Choice: 1
Using UCS algorithm.
```

Output:

```
Algorithm: UCS
Time: 15 ms
Path: slow -> slot -> soot -> loot -> loop
Degrees: 4
Nodes visited: 387
```

4.1.3 Cry → Die

Input:

```
Enter starting word: cry
Enter ending word: die
```

Output:

```
Algorithm: UCS
Time: 12 ms
Path: cry -> dry -> dey -> dee -> die
Degrees: 4
Nodes visited: 703
```

4.1.4 Small → World

Input:

```
Enter starting word: small
Enter ending word: world
```

Output:

```
Algorithm: UCS
Time: 32 ms
Path: small -> shall -> shale -> shame -> shams -> seams -> seals -> weals
-> weald -> woald -> world
Degrees: 10
Nodes visited: 5355
```

4.1.5 Spot → Ball

Input:

```
$ java -cp bin Main
Enter starting word: spot
Enter ending word: ball
```

Output:

```
Choice: 1
Using UCS algorithm.

Algorithm: UCS
Time: 15 ms
Path: spot -> soot -> boot -> bolt -> boll -> ball
Degrees: 5
Nodes visited: 1692
```

4.1.6 Swing → Black

Input:

```
Enter starting word: swing
Enter ending word: black
```

Output:

```
Algorithm: UCS
Time: 12 ms
Path: swing -> swink -> slink -> slick -> slack -> black
Degrees: 5
Nodes visited: 535
```

4.2 Algoritma GBFS

4.2.1 Ball → Blue

Input:

```
$ java -cp bin Main
Enter starting word: ball
Enter ending word: blue
```

Output:

```
Algorithm: GBFS
Path not found.
Time: 9 ms
Nodes visited: 3
```

4.2.2 Make → Mall

Input:

```
Enter starting word: make
Enter ending word: mall
```

Output:

```
Algorithm: GBFS
Time: 8 ms
Path: make -> male -> mall
Degrees: 2
Nodes visited: 2
```

4.2.3 Greed → Glass

Input:

```
Enter starting word: greed
Enter ending word: glass
```

Output:

```
Algorithm: GBFS
Time: 7 ms
Path: greed -> gleed -> glees -> gleds -> glads -> glass
Degrees: 5
Nodes visited: 5
```

4.2.4 Block → World

Input:

```
Enter starting word: block
Enter ending word: world
```

Output:

```
Algorithm: GBFS
Path not found.
Time: 9 ms
Nodes visited: 5
```

4.2.5 Small → Swing

Input :

```
Enter starting word: small
Enter ending word: swing
```

Output :

```
Algorithm: GBFS
Time: 8 ms
Path: small -> scall -> scald -> scale -> swale -> swage -> swags -> swans
-> swang -> swing
Degrees: 9
Nodes visited: 9
```

4.2.6 Plane → Blade

Input :

```
Enter starting word: plane
Enter ending word: blade
```

Output :

```
Algorithm: GBFS
Time: 9 ms
Path: plane -> alane -> alate -> blate -> blade
Degrees: 4
Nodes visited: 4
```

4.3 Algoritma A*

4.3.1 Atlases → Cabaret

Input :

```
Enter starting word: atlases
Enter ending word: cabaret
```

Output :

```
Algorithm: A*
Time: 42 ms
Path: atlases -> anlases -> anlaces -> unlaces -> unlades -> unladed -> unf
aded -> unfaked -> uncaked -> uncased -> uncases -> uneases -> ureases -> c
reases -> cresses -> crosses -> crosser -> crasser -> crasher -> brasher ->
 brasier -> brakier -> beakier -> peakier -> peckier -> pickier -> dickier
-> dickies -> hickies -> hackies -> hackles -> heckles -> deckles -> decile
s -> defiles -> defiled -> deviled -> develed -> reveled -> raveled -> rave
ned -> havened -> havered -> wavered -> watered -> catered -> capered -> ta
pered -> tabered -> tabored -> taboret -> tabaret -> cabaret
Degrees: 52
Nodes visited: 7857
```

4.3.2 Swing → Blues

Input :

```
Enter starting word: swing
Enter ending word: blues
```

Output :

```
Algorithm: A*
Time: 14 ms
Path: swing -> swang -> swans -> swabs -> slabs -> blabs -> blubs -> blues
Degrees: 7
Nodes visited: 833
```

4.3.3 Clue → Ball

Input :

```
Enter starting word: clue
Enter ending word: ball
```

Output :

```
Algorithm: A*
Time: 23 ms
Path: clue -> flue -> flux -> faux -> falx -> fall -> ball
Degrees: 6
Nodes visited: 1267
```

4.3.4 Flush → Spore

Input :

```
Enter starting word: flush
Enter ending word: spore
```

Output :

```
Algorithm: A*
Time: 14 ms
Path: flush -> flash -> flask -> flank -> slank -> spank -> spark -> spare
-> spore
Degrees: 8
Nodes visited: 967
```

4.3.5 Blade → Gecko

Input :

```
Enter starting word: blade
Enter ending word: gecko
```

Output :

```
Algorithm: A*
Time: 23 ms
Path: blade -> blame -> blams -> beams -> beaks -> becks -> gecks -> gecko
Degrees: 7
Nodes visited: 2009
```

4.3.6 Cats → Dogs

Input:

```
Enter starting word: cats
Enter ending word: dogs
```

Output:

```
Algorithm: A*
Time: 20 ms
Path: cats -> cots -> cogs -> dogs
Degrees: 3
Nodes visited: 634
```

Bab V
Analisis Hasil Pengujian

5.1 Optimalitas

Dari segi optimalitas, algoritma GBFS tidak optimal sehingga tidak akan selalu mendapatkan rute yang paling pendek dari simpul awal ke simpul akhir. Algoritma GBFS juga tidak komplit dan akan sering tidak menemukan solusi, karena tidak mengecek seluruh simpul. Hal ini terbukti dari hasil pengujian yang didapatkan, karena beberapa kali algoritma GBFS tidak mendapatkan solusi, sedangkan algoritma lainnya tidak terdapat uji kasus yang tidak mendapatkan solusi.

Algoritma UCS dan A* dua-duanya merupakan algoritma komplit sehingga akan selalu menemukan solusi jika terdapat solusi tersebut. Secara optimalitas, algoritma A* akan lebih optimal karena memanfaatkan fungsi heuristik untuk mengecek simpul-simpul yang lebih meyakinkan.

5.2 Waktu Eksekusi

Dari segi waktu eksekusi, didapatkan bahwa algoritma GBFS akan lebih sering mendapatkan solusi dalam waktu yang lebih cepat. Ini karena algoritma GBFS tidak melakukan *backtracking* sama sekali dan pada setiap langkah akan membuang simpul yang tidak paling optimal menurut fungsi heuristik, sehingga akan menghasilkan solusi yang sangat cepat.

Algoritma UCS dan A* tidak jauh beda dalam segi waktu eksekusi, karena kedua algoritma bersifat komplit dan mengecek seluruh simpul sampai *queue* kosong. Namun pada umumnya algoritma A* akan mendapatkan solusi lebih cepat, karena memanfaatkan fungsi heuristik pada GBFS.

5.3 Memori yang Dibutuhkan

Dari segi memori yang dipakai, terlihat algoritma GBFS menelusuri simpul jauh lebih sedikit, sehingga pemakaian memori jauh lebih sedikit dibanding kedua algoritma lainnya. Algoritma GBFS juga tidak membutuhkan struktur data *queue* sama sekali, sehingga menyebabkan pemakaian memori yang sangat sedikit.

Algoritma UCS dan A* menelusuri simpul dengan jumlah yang tidak jauh berbeda, karena keduanya merupakan algoritma *backtracking* yang komplit dan membutuhkan *queue* untuk menyimpan data-data simpul. Namun pada umumnya A* akan menggunakan lebih sedikit memori karena akan mendapatkan solusi lebih cepat sehingga simpul yang ditelusuri lebih sedikit.

Bab VI
Kode Program

Repository program dapat dilihat pada tautan berikut:
https://github.com/trimonuter/Tucil3_13522158

Bab VII
Lampiran

Poin	Ya	Tidak
1. Program berhasil dijalankan.	✓	
2. Program dapat menemukan rangkaian kata dari *start word* ke *end word* sesuai aturan permainan dengan algoritma UCS	✓	
3. Solusi yang diberikan pada algoritma UCS optimal	✓	
4. Program dapat menemukan rangkaian kata dari *start word* ke *end word* sesuai aturan permainan dengan algoritma *Greedy Best First Search*	✓	
5. Program dapat menemukan rangkaian kata dari *start word* ke *end word* sesuai aturan permainan dengan algoritma A*	✓	
6. Solusi yang diberikan pada algoritma A* optimal	✓	
7. [Bonus]: Program memiliki tampilan GUI		✓