

Tugas Kecil 3 IF2211 Strategi Algoritma
Semester II tahun 2023/2024
Penyelesaian Permainan Word Ladder Menggunakan Algoritma UCS,
Greedy Best First
Search, dan A*



Oleh :
Muhammad Zaki 13522136
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA INSTITUT
TEKNOLOGI BANDUNG TAHUN AJARAN 2023/2024

BAB 1

Landasan Teori

1.1. Algoritma UCS

Algoritma UCS atau uniform cost search adalah salah satu algoritma pencarian graf yang digunakan untuk mencari jalur dengan biaya terkecil atau jarak terpendek dari satu simpul ke simpul lainnya. Uniform-Cost Search mirip dengan algoritma Dijkstra.

Dalam algoritma ini, dari keadaan awal, kita akan mengunjungi state yang berdekatan dan akan memilih state yang paling murah biayanya, kemudian kita akan memilih keadaan berikutnya yang paling murah biayanya dari semua keadaan yang belum dikunjungi dan berdekatan dengan keadaan yang sudah dikunjungi. Dengan cara ini, kita akan mencoba mencapai keadaan tujuan (perhatikan bahwa kita tidak akan melanjutkan jalur melalui keadaan tujuan), bahkan jika kita sudah mencapai keadaan tujuan, kita akan terus mencari jalur-jalur lain yang mungkin (jika ada beberapa tujuan). Kita akan menyimpan antrian prioritas yang akan memberikan keadaan berikutnya yang paling murah biayanya dari semua keadaan yang berdekatan dengan keadaan yang sudah dikunjungi.

1.2. Algoritma Greedy Best First Search

Greedy Best-First Search adalah algoritma pencarian kecerdasan buatan yang berusaha untuk menemukan jalur paling menjanjikan dari suatu titik awal yang diberikan ke tujuan. Algoritma ini memberikan prioritas pada jalur-jalur yang tampaknya paling menjanjikan, terlepas dari apakah jalur tersebut benar-benar jalur terpendek atau tidak. Algoritma ini bekerja dengan mengevaluasi biaya dari setiap jalur yang mungkin kemudian memperluas jalur dengan biaya terendah. Proses ini diulang sampai tujuan tercapai.

Algoritma ini bekerja dengan menggunakan fungsi heuristik untuk menentukan jalur mana yang paling menjanjikan. Fungsi heuristik mempertimbangkan biaya dari jalur saat ini dan biaya perkiraan dari jalur-jalur yang tersisa. Jika biaya dari jalur saat ini lebih rendah dari perkiraan biaya dari jalur-jalur yang tersisa, maka jalur saat ini dipilih. Proses ini diulang sampai tujuan tercapai.

1.3. Algoritma A*

A* adalah algoritma traversal graf dan pencarian jalur, yang digunakan dalam banyak bidang ilmu komputer karena kelengkapan, optimalitas, dan efisiensi optimalnya. Diberikan sebuah graf berbobot, sebuah node sumber, dan sebuah node tujuan, algoritma ini menemukan jalur terpendek (dengan memperhatikan bobot yang diberikan) dari sumber ke tujuan.

Salah satu kekurangan praktis utamanya adalah kompleksitas ruangnya $O(b^d)$, di mana d adalah kedalaman solusi (jalur terpendek) dan b adalah faktor cabang (rata-rata jumlah pewaris per keadaan), karena menyimpan semua node yang dihasilkan di dalam memori. Oleh karena itu, dalam sistem rute perjalanan praktis, algoritma ini umumnya kalah dari algoritma yang dapat memproses graf sebelumnya untuk mencapai kinerja yang lebih baik, serta oleh pendekatan yang membatasi penggunaan memori; namun, A^* masih merupakan solusi terbaik dalam banyak kasus.

BAB 2

ANALISIS DAN IMPLEMENTASI

2.1 Algoritma UCS

```
public static SearchResult findPath(Map<String, List<String>> graph, String startWord, String
endWord) {
    PriorityQueue<WordNode> queue = new
PriorityQueue<>(Comparator.comparingInt(WordNode::getCost));
    Map<String, Integer> distances = new HashMap<>();
    Map<String, String> parents = new HashMap<>();
    Set<String> visited = new HashSet<>();
    int nodesExplored = 0;
    long startTime = System.nanoTime();

    queue.offer(new WordNode(startWord, 0));
    distances.put(startWord, 0);

    while (!queue.isEmpty()) {
        WordNode current = queue.poll();
        String currentWord = current.getWord();
        nodesExplored++;

        if (currentWord.equals(endWord)) {
            // Path found, reconstruct and return it
            List<String> path = new ArrayList<>();
            while (!currentWord.equals(startWord)) {
                path.add(0, currentWord);
                currentWord = parents.get(currentWord);
            }
            path.add(0, startWord);
            long endTime = System.nanoTime();
            long runtime = endTime - startTime;
            long memoryUsed = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
            return new SearchResult(path, nodesExplored, memoryUsed, runtime);
        }

        visited.add(currentWord);

        for (String neighbor : graph.getOrDefault(currentWord, Collections.emptyList())) {
            if (!visited.contains(neighbor)) {
                int newDistance = distances.get(currentWord) + 1;
                if (!distances.containsKey(neighbor) || newDistance < distances.get(neighbor)) {
                    distances.put(neighbor, newDistance);
                    parents.put(neighbor, currentWord);
                    queue.offer(new WordNode(neighbor, newDistance));
                }
            }
        }
    }

    return new SearchResult(Collections.emptyList(), nodesExplored, 0, 0); // No path found
}
```

Program diatas adalah implementasi dari algoritma UCS, algoritma UCS ini memiliki $g(n)$ yaitu cost dari root ke n . Pada awalnya, algoritma memasukkan titik awal ke dalam antrian prioritas dengan biaya awal 0. Selanjutnya, algoritma mengulangi langkah-langkah pencarian: mengeluarkan node dengan biaya terendah dari antrian prioritas, memeriksa apakah node tersebut adalah tujuan pencarian, memperbarui jarak terpendek ke tetangga-tetangga yang belum dieksplorasi, dan memasukkan

tetangga-tetangga yang telah diperbarui ke dalam antrian prioritas. Proses ini berlanjut hingga jalur terpendek dari titik awal ke titik tujuan ditemukan atau antrian prioritas kosong, menandakan bahwa tidak ada jalur yang tersedia antara kedua titik tersebut. Dengan menggunakan strategi ini, algoritma UCS secara efisien menemukan jalur terpendek dalam graf dengan mempertimbangkan biaya dari setiap langkah yang diambil.

Untuk permainan word ladder algoritma Uniform-Cost Search (UCS) dan Breadth-First Search (BFS) menghasilkan jalur yang sama, namun urutan node yang dihasilkan mungkin berbeda. Hal ini karena BFS memeriksa node pada tingkat yang sama secara berurutan, sedangkan UCS memprioritaskan node dengan biaya terendah.

2.2 Algoritma Greedy Best First Search

```
public static SearchResult findPath(Map<String, List<String>> graph, String startWord, String endWord)
{
    PriorityQueue<WordNode> queue = new
    PriorityQueue<>(Comparator.comparingInt(WordNode::getCost));
    Map<String, String> parents = new HashMap<>();
    Set<String> visited = new HashSet<>();
    int nodesExplored = 0;
    long startTime = System.nanoTime();

    queue.offer(new WordNode(startWord, Utility.calculateHeuristic(startWord, endWord)));

    while (!queue.isEmpty()) {
        WordNode current = queue.poll();
        String currentWord = current.getWord();
        nodesExplored++;

        if (currentWord.equals(endWord)) {
            // Path found, reconstruct and return it
            List<String> path = new ArrayList<>();
            while (!currentWord.equals(startWord)) {
                path.add(0, currentWord);
                currentWord = parents.get(currentWord);
            }
            path.add(0, startWord);
            long endTime = System.nanoTime();
            long runtime = endTime - startTime;
            long memoryUsed = Runtime.getRuntime().totalMemory() -
            Runtime.getRuntime().freeMemory();
            return new SearchResult(path, nodesExplored, memoryUsed, runtime);
        }

        visited.add(currentWord);

        for (String neighbor : graph.getOrDefault(currentWord, Collections.emptyList())) {
            if (!visited.contains(neighbor)) {
                int heuristic = Utility.calculateHeuristic(neighbor, endWord);
                queue.offer(new WordNode(neighbor, heuristic));
                parents.put(neighbor, currentWord);
            }
        }
    }

    return new SearchResult(Collections.emptyList(), nodesExplored, 0, 0); // No path found
}
```

Algoritma Greedy Best-First Search (GBFS) mengadopsi pendekatan yang menggunakan fungsi evaluasi $f(n)$ untuk setiap node dalam pencarian. Fungsi evaluasi ini, $f(n)$, biasanya merupakan estimasi biaya dari node n ke tujuan akhir. Contohnya adalah $h(n)$, yang dapat merepresentasikan estimasi biaya dari node n ke titik tujuan. GBFS kemudian memperluas node yang tampaknya paling dekat dengan tujuan, yaitu node yang memiliki nilai evaluasi $f(n)$ yang paling rendah. Dengan pendekatan ini, GBFS berusaha untuk memprioritaskan node-node yang secara heuristik paling dekat dengan tujuan akhir, meskipun demikian, tidak ada jaminan bahwa jalur yang dihasilkan akan optimal. Fungsi $f(n)$ yang digunakan atau heuristic yang digunakan adalah dengan menghitung perbedaan antara satu kata dengan yang lain

Proses pencarian dimulai dengan memasukkan node awal (startWord) ke dalam antrian prioritas dengan biaya heuristik dari startWord ke endWord. Selama antrian prioritas tidak kosong, algoritma mengulangi langkah-langkah pencarian berikut:

1. Mengeluarkan node dengan biaya terendah dari antrian prioritas.
2. Jika node yang diekstrak adalah tujuan (endWord), jalur dari endWord ke startWord direkonstruksi menggunakan informasi yang tersimpan dalam parents.
3. Node yang diekstrak ditandai sebagai sudah dieksplorasi dengan menambahkannya ke dalam visited.
4. Tetangga-tetangga dari node saat ini yang belum dieksplorasi ditambahkan ke dalam antrian prioritas dengan biaya heuristik masing-masing. Informasi hubungan antara tetangga-tetangga baru ini dan node saat ini disimpan dalam parents.

Proses ini diulangi hingga jalur dari startWord ke endWord ditemukan atau antrian kosong. Jika tidak ada jalur yang ditemukan, hasil pencarian dengan jalur kosong dan informasi nol dikembalikan. Dengan demikian, algoritma GBFS mencoba untuk menemukan jalur terpendek dengan mempertimbangkan biaya heuristik dari setiap langkah dalam pencarian.

Secara teori GBFS tidak menjamin mendapatkan solusi optimal karena GBFS hanya mempertimbangkan langkah terbaik pada setiap iterasi tanpa mempertimbangkan keseluruhan gambaran masalah, ia bisa jatuh ke dalam infinite loop local optimum. Artinya, meskipun GBFS dapat menemukan solusi yang memuaskan atau cukup baik, tidak ada jaminan bahwa solusi tersebut adalah yang optimal.

2.3 Algoritma A*

```
public static SearchResult findPath(Map<String, List<String>> graph, String startWord, String endWord)
{
    PriorityQueue<WordNode> queue = new
    PriorityQueue<>(Comparator.comparingInt(WordNode::getCost));
    Map<String, Integer> distances = new HashMap<>();
    Map<String, String> parents = new HashMap<>();
    Set<String> visited = new HashSet<>();
    int nodesExplored = 0;
    long startTime = System.nanoTime();

    queue.offer(new WordNode(startWord, 0));
    distances.put(startWord, 0);

    while (!queue.isEmpty()) {
        WordNode current = queue.poll();
        String currentWord = current.getWord();
        nodesExplored++;

        if (currentWord.equals(endWord)) {
            // Path ketemu
            List<String> path = new ArrayList<>();
            while (!currentWord.equals(startWord)) {
                path.add(0, currentWord);
                currentWord = parents.get(currentWord);
            }
            path.add(0, startWord);
            long endTime = System.nanoTime();
            long runtime = endTime - startTime;
            long memoryUsed = Runtime.getRuntime().totalMemory() -
            Runtime.getRuntime().freeMemory();
            return new SearchResult(path, nodesExplored, memoryUsed, runtime);
        }

        visited.add(currentWord);

        for (String neighbor : graph.getOrDefault(currentWord, Collections.emptyList())) {
            if (!visited.contains(neighbor)) {
                int newDistance = distances.get(currentWord) + 1;
                if (!distances.containsKey(neighbor) || newDistance < distances.get(neighbor)) {
                    distances.put(neighbor, newDistance);
                    parents.put(neighbor, currentWord);
                    // Calculate heuristic for A*
                    int heuristic = Utility.calculateHeuristic(neighbor, endWord);
                    int totalCost = newDistance + heuristic;
                    // System.out.println("Word: " + neighbor + ", Cost: " + totalCost); // Debug
                    // Add heuristic to cost for A*
                    queue.offer(new WordNode(neighbor, totalCost));
                }
            }
        }
    }

    return new SearchResult(Collections.emptyList(), nodesExplored, 0, 0); //Tidak ada path
}
```

Algoritma A* ini merupakan gabungan dari UCS dan juga GBFS atau memiliki fungsi $f(n) = g(n) + h(n)$, dimana $g(n)$ merupakan UCS dan juga $h(n)$ merupakan GBFS, algoritma A* menggunakan heuristik untuk memperkirakan biaya yang diperlukan untuk mencapai tujuan. Di dalam iterasi utamanya, algoritma memeriksa setiap node secara berurutan berdasarkan total biaya dari node awal ke node tersebut dan perkiraan biaya dari node tersebut ke node tujuan. Algoritma terus mengeksansi node-node tersebut sampai menemukan node tujuan atau sampai tidak ada node yang tersisa untuk dieksplorasi.

Setiap kali sebuah node dieksplorasi, algoritma memperbarui jarak terpendek yang telah ditemukan hingga saat ini untuk mencapai setiap node, serta memperbarui informasi tentang node-nodenya. Algoritma menggunakan sebuah PriorityQueue untuk mengelola node-node yang akan dieksplorasi, dengan prioritas diberikan berdasarkan perkiraan biaya total dari node tersebut ke tujuan.

Saat sebuah node dieksplorasi, algoritma memperhitungkan semua tetangga yang belum dieksplorasi. Algoritma kemudian menghitung biaya baru untuk mencapai setiap tetangga tersebut, dan jika biaya baru tersebut lebih kecil dari biaya sebelumnya, algoritma memperbarui informasi tentang tetangga tersebut dan memasukkannya ke dalam PriorityQueue dengan mempertimbangkan heuristiknya. Algoritma berakhir ketika node tujuan ditemukan atau ketika tidak ada node lagi yang tersisa untuk dieksplorasi.

Sebuah heuristik dikatakan admissible jika tidak pernah memperkirakan biaya yang lebih besar dari biaya sebenarnya untuk mencapai tujuan. Dengan kata lain, untuk setiap simpul (node) dalam ruang pencarian, nilai heuristik ($h(n)$) harus kurang dari atau sama dengan nilai heuristik sebenarnya ($h^*(n)$), di mana $h^*(n)$ adalah biaya yang sebenarnya atau optimal untuk mencapai keadaan tujuan dari simpul n .

Fungsi heuristik yang saya gunakan adalah admissible karena menghitung jumlah karakter yang berbeda antara kata saat ini dan kata tujuan. Dalam setiap langkah, heuristik hanya memperkirakan jumlah langkah minimum yang diperlukan untuk mengubah kata saat ini menjadi kata tujuan. Karena kita hanya menghitung jumlah karakter yang berbeda, heuristik ini tidak pernah melebihi jumlah langkah yang sebenarnya diperlukan untuk mencapai tujuan. Dengan demikian, nilai yang dikembalikan oleh heuristik ini selalu kurang dari atau sama dengan biaya sebenarnya untuk mencapai solusi, memenuhi syarat admissible.

Secara teoritis, algoritma A* cenderung lebih efisien daripada algoritma UCS dalam kasus Word Ladder karena A* menggunakan heuristik untuk mengarahkan pencarian ke arah solusi potensial, sementara UCS mengeksplorasi secara sistematis setiap kemungkinan langkah tanpa pengetahuan tentang arah yang benar menuju tujuan.

BAB 3

KODE PROGRAM

3.1 Main.java

```
import java.util.*;

public class Main {
    public static void main(String[] args) {

        Map<String, List<String>> loadedWordGraph = Utility.loadWordGraphFromFile("word_graph.txt");

        Scanner scanner = new Scanner(System.in);
        System.out.print("Masukkan kata awal: ");
        String startWord = scanner.nextLine().trim().toLowerCase();

        System.out.print("Masukkan kata akhir: ");
        String endWord = scanner.nextLine().trim().toLowerCase();
        scanner.close();

        if (!loadedWordGraph.containsKey(startWord) || !loadedWordGraph.containsKey(endWord)) {
            System.out.println("Kata awal atau kata akhir tidak ditemukan dalam graf kata.");
            return;
        }

        UCS.SearchResult ucsResult = UCS.findPath(loadedWordGraph, startWord, endWord);
        List<String> shortestPathUCS = ucsResult.getPath();
        int nodesExploredUCS = ucsResult.getNodesExplored();
        long memoryUsedUCS = ucsResult.getMemoryUsed();
        long runtimeUCS = ucsResult.getRuntime();

        AStar.SearchResult aStarResult = AStar.findPath(loadedWordGraph, startWord, endWord);
        List<String> shortestPathAStar = aStarResult.getPath();
        int nodesExploredAStar = aStarResult.getNodesExplored();
        long memoryUsedAStar = aStarResult.getMemoryUsed();
        long runtimeAStar = aStarResult.getRuntime();

        GreedyBestFirstSearch.SearchResult greedyResult =
GreedyBestFirstSearch.findPath(loadedWordGraph, startWord, endWord);
        List<String> shortestPathGreedy = greedyResult.getPath();
        int nodesExploredGreedy = greedyResult.getNodesExplored();
        long memoryUsedGreedy = greedyResult.getMemoryUsed();
        long runtimeGreedy = greedyResult.getRuntime();

        System.out.print("Shortest path from '" + startWord + "' to '" + endWord + "' " + "UCS" + ":
");
        Utility.PathPrinter.printResults(shortestPathUCS, nodesExploredUCS, memoryUsedUCS, runtimeUCS);
        System.out.print("Shortest path from '" + startWord + "' to '" + endWord + "' " + "AStar" + ":
");
        Utility.PathPrinter.printResults(shortestPathAStar, nodesExploredAStar, memoryUsedAStar,
runtimeAStar);
        System.out.print("Shortest path from '" + startWord + "' to '" + endWord + "' " + "GBFS" + ":
");
        Utility.PathPrinter.printResults(shortestPathGreedy, nodesExploredGreedy, memoryUsedGreedy,
runtimeGreedy);

    }

}
```

3.2 Utility.java

```
private static boolean isOneLetterApart(String word1, String word2) {
    if (word1.length() != word2.length()) {
        return false;
    }

    int diffCount = 0;
    for (int i = 0; i < word1.length(); i++) {
        if (word1.charAt(i) != word2.charAt(i)) {
            diffCount++;
            if (diffCount > 1) {
                return false;
            }
        }
    }

    return diffCount == 1;
}

public static int calculateHeuristic(String currentWord, String endWord) {
    int diffCount = 0;

    for (int i = 0; i < currentWord.length(); i++) {
        if (currentWord.charAt(i) != endWord.charAt(i)) {
            diffCount++;
        }
    }

    return diffCount;
}

private static void printPath(List<String> path) {
    for (int i = 0; i < path.size(); i++) {
        System.out.print(path.get(i));
        if (i != path.size() - 1) {
            System.out.print(" -> ");
        }
    }
    System.out.println();
}

public class PathPrinter {
    public static void printResults( List<String> shortestPath, int nodesExplored, long memoryUsed,
    long runtime) {

        printPath(shortestPath);
        System.out.println("Jumlah Path : " + shortestPath.size());
        System.out.println("Number of nodes explored: " + nodesExplored);
        System.out.println("Memory used: " + memoryUsed + " bytes");
        System.out.println("Runtime: " + runtime / 1e6 + " milliseconds\n");
    }

    private static void printPath(List<String> path) {
        for (int i = 0; i < path.size(); i++) {
            System.out.print(path.get(i));
            if (i != path.size() - 1) {
                System.out.print(" -> ");
            }
        }
        System.out.println();
    }
}
```



```

public static void saveWordGraphToFile(Map<String, List<String>> wordGraph, String filename) {
    try (FileWriter writer = new FileWriter(filename)) {
        for (Map.Entry<String, List<String>> entry : wordGraph.entrySet()) {
            String word = entry.getKey();
            List<String> connectedWords = entry.getValue();
            StringBuilder sb = new StringBuilder();
            sb.append(word).append(": ");
            for (String connectedWord : connectedWords) {
                sb.append(connectedWord).append(", ");
            }
            sb.delete(sb.length() - 2, sb.length()); // Menghapus koma dan spasi terakhir
            sb.append("\n");
            writer.write(sb.toString());
        }
        System.out.println("Graf telah disimpan ke file: " + filename);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static Map<String, List<String>> loadWordGraphFromFile(String filename) {
    Map<String, List<String>> loadedWordGraph = new HashMap<>();

    try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
        String line;
        while ((line = reader.readLine()) != null) {
            String[] parts = line.split(":");
            String word = parts[0].trim();
            List<String> connectedWords = new ArrayList<>();
            if (parts.length > 1) {
                String[] connectedWordArray = parts[1].split(",");
                for (String connectedWord : connectedWordArray) {
                    connectedWords.add(connectedWord.trim());
                }
            }
            loadedWordGraph.put(word, connectedWords);
        }
        System.out.println("Graf telah dimuat dari file: " + filename);
    } catch (IOException e) {
        e.printStackTrace();
    }

    return loadedWordGraph;
}

// Fungsi untuk membaca kata-kata dari file dan menyimpannya dalam Map
private static Map<Integer, List<String>> readWordsFromFile(String filename) {
    Map<Integer, List<String>> wordMap = new HashMap<>();

    try {
        FileReader fileReader = new FileReader(filename);
        BufferedReader bufferedReader = new BufferedReader(fileReader);

        String line;
        while ((line = bufferedReader.readLine()) != null) {
            int length = line.length();
            wordMap.computeIfAbsent(length, k -> new ArrayList<>()).add(line);
        }

        bufferedReader.close();
    } catch (IOException e) {
        e.printStackTrace();
    }

    return wordMap;
}

// Fungsi untuk membuat graf word ladder menggunakan multithreading
private static Map<String, List<String>> buildWordGraphMultiThread(Map<Integer, List<String>>
wordMap) {
    Map<String, List<String>> wordGraph = new HashMap<>();
    ExecutorService executor =
        Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

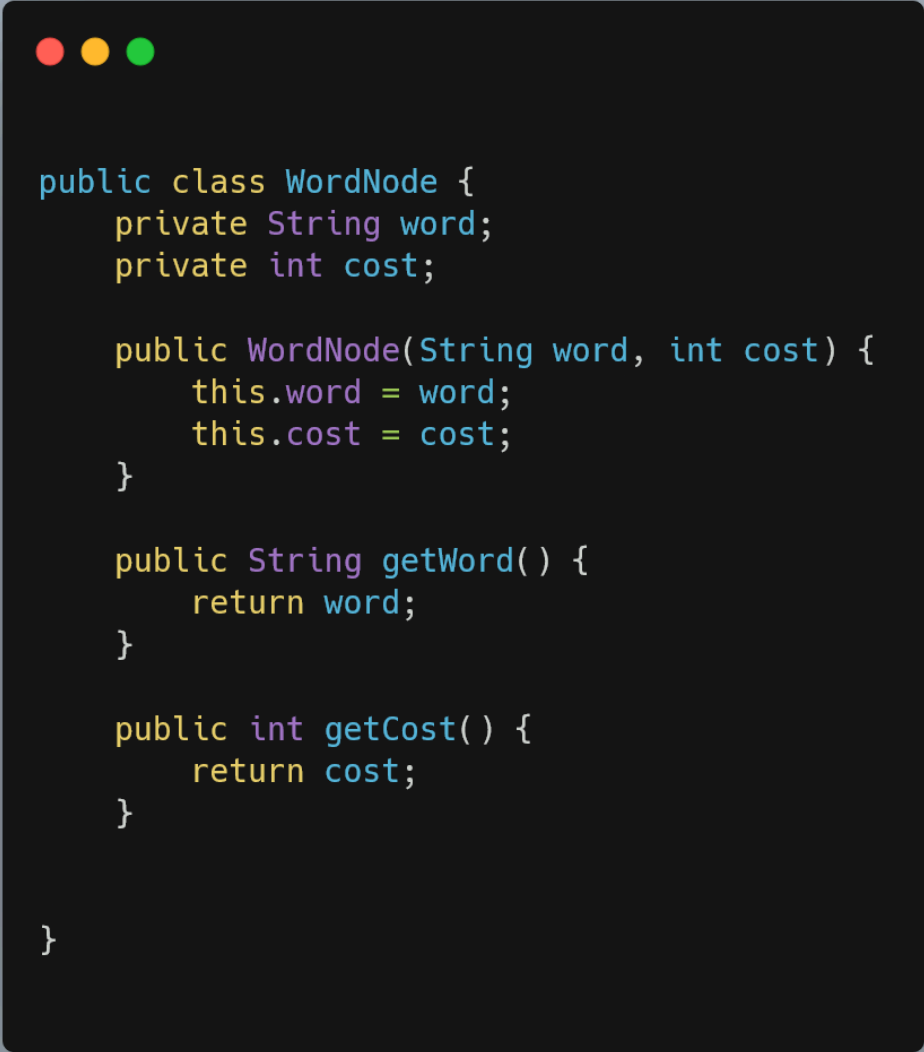
    try {
        for (List<String> words : wordMap.values()) {
            for (String word : words) {
                executor.submit(() -> {
                    // Buat koneksi dengan kata-kata yang memiliki perbedaan satu huruf
                    List<String> connectedWords = new ArrayList<>();
                    for (String otherWord : words) {
                        if (!word.equals(otherWord) && isOneLetterApart(word, otherWord)) {
                            connectedWords.add(otherWord);
                        }
                    }
                    synchronized (wordGraph) {
                        wordGraph.put(word, connectedWords);
                    }
                });
            }
        }
        // Debug: Tampilkan kata dan kata-kata terhubungnya
        System.out.println("Kata: " + word + ", Terhubung: " + connectedWords);
    } finally {
        executor.shutdown();
        try {
            executor.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    return wordGraph;
}

```

Method	Fungsi
isOneLetterApart()	digunakan untuk mencari kata mana saja yang memiliki perbedaan hanya satu kata
calculateHeuristic()	Digunakan untuk menghitung heuristic
PrintPath()	Digunakan untuk print path yang dihasilkan
PrintResult()	Digunakan untuk print result seperti jumlah node yang dikunjungi, waktu, memory yang digunakan
saveWordGraphToFile()	Menyimpan graph yang dihasilkan ke file. txt
readWordsFromFile()	membaca word dari file
buildWordGraphMultiThread()	Membuat graph
loadWordGraphFromFile()	membaca graph dari file

3.3 WordNode.java



```
public class WordNode {  
    private String word;  
    private int cost;  
  
    public WordNode(String word, int cost) {  
        this.word = word;  
        this.cost = cost;  
    }  
  
    public String getWord() {  
        return word;  
    }  
  
    public int getCost() {  
        return cost;  
    }  
  
}
```

BAB 4

TESTING

4.1 work -> tech

```
Masukkan kata awal: work
Masukkan kata akhir: tech
Shortest path from 'work' to 'tech' UCS: work -> pork -> perk -> peck -> teck -> tech
Jumlah Path : 6
Number of nodes explored: 3995
Memory used: 100475304 bytes
Runtime: 26.8375 milliseconds

Shortest path from 'work' to 'tech' AStar: work -> fork -> ferk -> feck -> teck -> tech
Jumlah Path : 6
Number of nodes explored: 37
Memory used: 100475304 bytes
Runtime: 1.4181 milliseconds

Shortest path from 'work' to 'tech' GBFS: work -> cork -> cock -> coch -> toch -> tech
Jumlah Path : 6
Number of nodes explored: 6
Memory used: 100475304 bytes
Runtime: 0.8294 milliseconds
```

4.2 better -> singer

```
Masukkan kata awal: better
Masukkan kata akhir: singer
Shortest path from 'better' to 'singer' UCS: better -> setter -> sitter -> sinter -> singer
Jumlah Path : 5
Number of nodes explored: 1271
Memory used: 100591048 bytes
Runtime: 9.21 milliseconds

Shortest path from 'better' to 'singer' AStar: better -> bitter -> sitter -> sinter -> singer
Jumlah Path : 5
Number of nodes explored: 6
Memory used: 100591048 bytes
Runtime: 0.3004 milliseconds

Shortest path from 'better' to 'singer' GBFS: better -> bitter -> sitter -> sinter -> singer
Jumlah Path : 5
Number of nodes explored: 5
Memory used: 100591048 bytes
Runtime: 0.272 milliseconds
```

4.3 flow -> gold

```

Shortest path from 'flow' to 'gold' UCS: flow -> glow -> glod -> good -> gold
Jumlah Path : 5
Number of nodes explored: 975
Memory used: 100942952 bytes
Runtime: 8.72 milliseconds

Shortest path from 'flow' to 'gold' AStar: flow -> glow -> glod -> good -> gold
Jumlah Path : 5
Number of nodes explored: 5
Memory used: 100942952 bytes
Runtime: 0.1955 milliseconds

Shortest path from 'flow' to 'gold' GBFS: flow -> glow -> glod -> good -> gold
Jumlah Path : 5
Number of nodes explored: 5
Memory used: 100942952 bytes
Runtime: 0.4224 milliseconds

```

4.4 dummy -> shark

```

Masukkan kata awal: dummy
Masukkan kata akhir: shark
Shortest path from 'dummy' to 'shark' UCS: dummy -> gummy -> gumma -> guama -> guana -> ghana -> thana -> th
nk -> shank -> shark
Jumlah Path : 10
Number of nodes explored: 10962
Memory used: 101608720 bytes
Runtime: 43.3324 milliseconds

Shortest path from 'dummy' to 'shark' AStar: dummy -> gummy -> gumma -> guama -> guana -> ghana -> thana -> t
hank -> shank -> shark
Jumlah Path : 10
Number of nodes explored: 559
Memory used: 101711872 bytes
Runtime: 5.7554 milliseconds

Shortest path from 'dummy' to 'shark' GBFS: dummy -> tummy -> tommy -> tammy -> sammy -> saimy -> sairy -> sp
iry -> spary -> spark -> shark
Jumlah Path : 11
Number of nodes explored: 16
Memory used: 101711872 bytes
Runtime: 0.9099 milliseconds

```

4.5 floc -> best

```

Masukkan kata awal: floc
Masukkan kata akhir: best
Shortest path from 'floc' to 'best' UCS: floc -> bloc -> blot -> blat -> beat -> best
Jumlah Path : 6
Number of nodes explored: 1404
Memory used: 101989512 bytes
Runtime: 10.4413 milliseconds

Shortest path from 'floc' to 'best' AStar: floc -> bloc -> blot -> blat -> beat -> best
Jumlah Path : 6
Number of nodes explored: 29
Memory used: 101989512 bytes
Runtime: 0.5586 milliseconds

Shortest path from 'floc' to 'best' GBFS: floc -> bloc -> blot -> blat -> beat -> best
Jumlah Path : 6
Number of nodes explored: 6
Memory used: 101989512 bytes
Runtime: 0.1642 milliseconds

```


4.6 hacker -> racism

```
Graf telah dimuat dari file: word_graph.txt
Masukkan kata awal: hacker
Masukkan kata akhir: racism
Shortest path from 'hacker' to 'racism' UCS: hacker -> hacked -> harked -> marked -> market -> mariet -> marist -> maoist -> taoist -> tapist -> rapist -> racist -> racism
Jumlah Path : 13
Number of nodes explored: 15857
Memory used: 105113168 bytes
Runtime: 48.4538 milliseconds

Shortest path from 'hacker' to 'racism' AStar: hacker -> hacked -> harked -> marked -> market -> mariet -> marist -> maoist -> taoist -> tapist -> rapist -> racist -> racism
Jumlah Path : 13
Number of nodes explored: 4518
Memory used: 106295296 bytes
Runtime: 22.8998 milliseconds

Shortest path from 'hacker' to 'racism' GBFS: hacker -> racker -> racked -> racket -> rachet -> rachel -> raphes -> raphes -> rashes -> rasher -> rasper -> raster -> rafter -> ranter -> rancer -> rancel -> rances -> ranees -> ragees -> rakees -> ramees -> ramies -> mamies -> maves -> maries -> mariet -> marist -> malist -> malism -> magism -> manism -> maoism -> taoism -> tapism -> papism -> papist -> rapist -> racist -> racism
Jumlah Path : 39
Number of nodes explored: 723
Memory used: 107210352 bytes
Runtime: 6.1664 milliseconds
```

BAB 5

PEMBAHASAN

Dari 6 percobaan yang sudah dilakukan dapat dilihat bahwa algoritma A* mempunyai optimalisasi yang lebih baik diantara kedua algoritma lainnya, hal ini bisa dibuktikan dari jumlah path akhir yang lebih sedikit dibandingkan GBFS dan juga time execution yang lebih rendah dibandingkan UCS. Lalu jumlah path yang dilalui oleh A* juga lebih sedikit dibandingkan algoritma UCS yang mempunyai jumlah path akhir yang sama dengan A*. Dari segi penggunaan memori ke-3 algoritma mempunyai selisih yang sangat sedikit antara satu dengan yang lain, sehingga tidak bisa dijadikan acuan apakah dari segi memori A* lebih efisien.

Algoritma yang dihasilkan A* juga sudah efisien, selain dibuktikan pada percobaan diatas penulis juga mencoba memainkan pada website [wordwormdormdork](http://wordwormdormdork.com) dan mendapat poin yang optimal

LAMPIRAN

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

Link repository : https://github.com/mzaki9/Tucil3_135522136