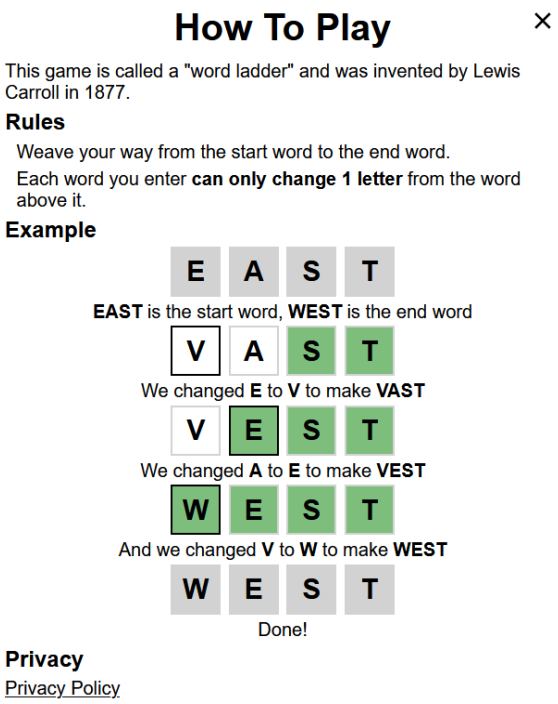
Laporan Tugas Kecil 3
IF2211 Strategi Algoritma
Penyelesaian Permainan Word Ladder Menggunakan Algoritma UCS, Greedy Best First Search, dan A*
Semester II Tahun 2023/2024

Disusun Oleh:
Randy Verdian 13522067

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2024

Bab 1 Deskripsi Masalah

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.



Pada tugas kecil ini akan dibuat penyelesaian permainan Word Ladder dengan algoritma UCS, *Greedy Best First Search*, dan A*.

Bab 2 Implementasi Algoritma

2.1. Uniform Cost Search

Algoritma UCS (Uniform Cost Search) merupakan salah satu algoritma pencarian jalur dalam graf yang menggunakan pendekatan pencarian terarah berbobot. UCS berusaha menemukan jalur yang memiliki biaya total terendah dari simpul awal ke simpul tujuan. Dalam kasus word ladder ini, jarak antar node adalah sama, sehingga algoritma UCS pada word ladder ini cenderung mirip BFS.

Berikut adalah langkah-langkah dari algoritma UCS yang dilakukan:

1. Inisialisasi:
    - Membuat struktur data wordQueue, distanceQueue, dan pathsQueue yang merupakan LinkedList untuk menyimpan kata-kata, biaya, dan jalur dari simpul awal ke simpul saat ini.
    - Inisialisasi variabel solution sebagai LinkedList untuk menyimpan jalur solusi.
    - Inisialisasi variabel nodeVisited untuk menghitung jumlah simpul yang telah dikunjungi.
2. Tambahkan simpul awal ke queue:
    - Tambahkan simpul start ke dalam wordQueue.
    - Tambahkan jarak awal 1 ke dalam distanceQueue.
    - Hapus simpul awal dari dict (kamus kata-kata yang mungkin).
3. Lakukan iterasi sampai wordQueue tidak kosong:
    - Ambil simpul saat ini dari wordQueue.
    - Ambil jarak saat ini dari distanceQueue.
    - Ambil jalur saat ini dari pathsQueue.
    - Tambahkan 1 ke nodeVisited untuk menandai kunjungan ke simpul saat ini.
4. Periksa apakah simpul saat ini adalah tujuan:
    - Jika currWord sama dengan end, maka:
        - Jika currDistance lebih kecil dari result (biaya solusi terbaik yang ditemukan sejauh ini), update result dengan currDistance.
        - Perbarui solution dengan currentPathQueue sebagai jalur solusi terbaik yang ditemukan sejauh ini.
5. Cari tetangga dari simpul saat ini:
    - Dapatkan daftar tetangga dari simpul saat ini menggunakan fungsi getNeighbors dari kelas Neighbors.
    - Untuk setiap tetangga:
        - Tambahkan tetangga ke wordQueue.

        - Tambahkan jarak dari simpul saat ini ke tetangga ditambah 1 ke distanceQueue.
        - Hapus tetangga dari dict.
        - Buat jalur baru dengan menyalin jalur saat ini, tambahkan tetangga ke jalur baru, dan tambahkan jalur baru ke pathsQueue.
6. Setelah iterasi selesai:
    - Simpan jalur solusi terbaik dalam variabel solution.
    - Simpan jumlah simpul yang dikunjungi dalam variabel nodeVisited.

Algoritma ini akan berhenti ketika simpul tujuan pertama kali ditemukan dan akan memberikan jalur terpendek yang ditemukan dari simpul awal ke simpul tujuan berdasarkan biaya terkecil yang dihitung.

2.2. Greedy Best First Search

Algoritma GBFS (Greedy Best-First Search), yang merupakan varian dari algoritma pencarian jalur dalam graf yang menggunakan pendekatan pencarian terarah tanpa informasi arah yang berfokus pada ekspansi simpul dengan heuristik terbaik.

Berikut adalah langkah-langkah dari algoritma GBFS:

1. Inisialisasi:
    - Membuat struktur data wordQueue berupa PriorityQueue yang diurutkan berdasarkan nilai heuristik dari fungsi heuristic terhadap setiap kata terhadap kata tujuan yang mana heuristic berdasarkan kemiripan kata yang paling mirip.
    - Membuat cameFrom yang merupakan map untuk menyimpan informasi dari mana setiap kata berasal.
    - Membuat visited yang merupakan set untuk menyimpan kata-kata yang sudah dikunjungi.
    - Memasukkan kata awal ke dalam wordQueue dan menandainya sebagai telah dikunjungi dengan memasukkannya ke dalam visited.
2. Iterasi sampai wordQueue tidak kosong:
    - Ambil kata teratas dari wordQueue (yang memiliki nilai heuristik terkecil) sebagai currWord.
    - Jika currWord sama dengan kata tujuan, maka algoritma berhenti.
3. Untuk setiap tetangga dari currWord:
    - Dapatkan daftar tetangga dari currWord menggunakan fungsi getNeighbors dari kelas Neighbors.
    - Untuk setiap tetangga:
    - Jika tetangga belum dikunjungi:

- Tambahkan tetangga ke wordQueue.
- Tandai tetangga sebagai telah dikunjungi dengan memasukkannya ke dalam visited.
- Simpan informasi bahwa tetangga berasal dari currWord dalam cameFrom.

4. Setelah iterasi selesai:
    - Bangun jalur solusi dari cameFrom yang mengandung informasi tentang jalur mundur dari kata tujuan ke kata awal dengan memanggil fungsi buildPath.
    - Simpan jalur solusi dalam variabel solution.
    - Simpan jumlah simpul yang dikunjungi dalam variabel nodeVisited.
5. Fungsi buildPath:
    - Fungsi ini mengambil cameFrom dan kata tujuan sebagai argumen.
    - Memulai dari kata tujuan dan secara mundur mengikuti jalur yang ditandai dalam cameFrom, menambahkan setiap kata ke awal jalur (path).
    - Mengembalikan jalur solusi.

Algoritma GBFS cenderung mengeksplorasi jalur yang menunjukkan kemajuan menuju tujuan berdasarkan heuristik, meskipun tidak menjamin solusi optimal. Ini memprioritaskan ekspansi ke simpul yang memiliki nilai heuristik yang lebih rendah, dengan harapan untuk mendekati solusi dengan cepat.

2.3. A*

Algoritma A* (A-star) yang merupakan metode pencarian jalur dalam graf yang menggabungkan pendekatan pencarian terarah berbobot dengan algoritma pencarian terarah tanpa informasi arah. A* menggunakan fungsi evaluasi heuristik untuk memperkirakan biaya total dari awal ke tujuan melalui simpul yang sedang dieksplorasi.

Berikut adalah langkah-langkah dari algoritma A* yang disajikan di dalam kelas Astar:

1. Inisialisasi:
    - Membuat struktur data openSet berupa PriorityQueue yang diurutkan berdasarkan nilai fCost dari setiap simpul.
    - Membuat allNodes yang merupakan map untuk menyimpan semua simpul yang telah dibuat beserta informasi terkait.
    - Membuat simpul awal (startNode) dengan nilai gCost 0, hCost yang dihitung menggunakan fungsi heuristik dari start ke end, dan menambahkannya ke openSet dan allNodes.
2. Iterasi sampai openSet tidak kosong:
    - Ambil simpul teratas (current) dari openSet.

    - Tambahkan 1 ke nodeVisited untuk menghitung jumlah simpul yang telah dikunjungi.
    - Jika simpul saat ini adalah tujuan, maka jalur solusi telah ditemukan. Bangun jalur solusi menggunakan fungsi buildPath dan hentikan iterasi.
3. Untuk setiap tetangga dari current:
    - Dapatkan daftar tetangga (neighbors) dari kata saat ini menggunakan fungsi getNeighbors dari kelas Neighbors.
    - Untuk setiap tetangga:
        - Jika tetangga belum ada dalam allNodes atau biaya aktual (gCost) dari awal ke tetangga melalui current lebih rendah dari biaya sebelumnya:
        - Buat simpul tetangga baru atau perbarui simpul tetangga yang ada dalam allNodes.
        - Tentukan tetangga ini berasal dari current dan hitung nilai gCost, hCost, dan fCost-nya.
        - Tambahkan atau perbarui simpul tetangga dalam openSet.
4. Setelah iterasi selesai:
    - Simpan jumlah simpul yang dikunjungi dalam variabel nodeVisited.
5. Fungsi buildPath:
    - Fungsi ini mengambil simpul tujuan sebagai argumen.
    - Mulai dari simpul tujuan dan secara mundur mengikuti jalur yang ditandai dalam cameFrom, menambahkan setiap kata ke awal jalur (path).
    - Mengembalikan jalur solusi.

Algoritma A* menggunakan pendekatan yang mirip dengan GBFS tetapi memperhitungkan biaya aktual dari awal ke setiap simpul saat mengeksplorasi jalur. Ini memungkinkan A* untuk menemukan solusi optimal jika fungsi heuristik memenuhi sifat-sifat tertentu seperti admissibility.

Bab 3 Analisis Algoritma

3.1. Uniform Cost Search

Algoritma UCS berfungsi dengan cara memperluas pencarian dari simpul dengan biaya terendah terlebih dahulu. Ini dilakukan dengan mempertimbangkan biaya setiap langkah yang diambil untuk mencapai setiap simpul.

Pada intinya, algoritma ini mirip dengan algoritma Breadth-First Search (BFS) dalam hal memeriksa semua kemungkinan langkah pada setiap level pencarian. Namun, perbedaannya terletak pada pendekatan dalam menentukan urutan ekspansi simpul. Di BFS, simpul diekspansi dalam urutan yang sama seperti kedalaman pencarian, sementara di UCS, simpul diekspansi berdasarkan biaya yang terkait dengan mencapai simpul tersebut.

Dalam konteks Word Ladder, UCS secara berulang mencoba semua kemungkinan kata yang dapat dijangkau dari kata saat ini, dengan mempertimbangkan biaya (jumlah langkah) yang diperlukan untuk mencapainya. Namun pada kasus Word Ladder ini, setiap dua node memiliki *cost* yang sama yaitu 1, sehingga algoritma mencoba semua kemungkinan dari *neighbors* nya terlebih dahulu sebelum beralih ke *depth* selanjutnya. Jadi, meskipun secara konseptual mirip dengan BFS, UCS secara efektif memperhitungkan biaya perpindahan antar-kata, yang memungkinkan pencarian jalur terpendek. Ini menjadikannya lebih efisien dalam menemukan solusi optimal dalam kasus-kasus di mana bobot atau biaya langkah-langkah berbeda.

3.2. Greedy Best First Search

Secara teoritis, algoritma Greedy Best First Search (GBFS) tidak menjamin solusi optimal untuk persoalan Word Ladder. Algoritma GBFS melakukan ekspansi simpul berdasarkan heuristik yang dipilih, yang dalam konteks Word Ladder sering kali berupa estimasi jarak dari simpul saat ini ke tujuan. Dengan demikian, GBFS cenderung memilih simpul yang menjanjikan terdekat dengan tujuan tanpa mempertimbangkan total biaya atau jarak sejauh ini.

Meskipun pendekatan ini dapat mengarah pada solusi yang cepat dalam beberapa kasus, itu tidak menjamin solusi optimal karena GBFS tidak mempertimbangkan secara lengkap semua kemungkinan jalur. Dalam Word Ladder, di mana setiap langkah memiliki biaya yang sama, pendekatan GBFS dapat melewatkan jalur yang lebih panjang tetapi lebih optimal secara keseluruhan. Ini karena GBFS cenderung terjebak dalam jalur yang terlihat lebih dekat dengan tujuan tetapi pada akhirnya dapat menghasilkan solusi yang lebih panjang atau kurang optimal secara keseluruhan.

Dengan demikian, secara teoritis, GBFS tidak menjamin solusi optimal untuk Word Ladder karena algoritma tersebut cenderung lebih fokus pada heuristik lokal daripada evaluasi global dari semua kemungkinan jalur.

3.3. A*

Dalam kasus word ladder, fungsi-fungsi f(n), g(n), dan h(n) memiliki peran penting dalam algoritma A* untuk menemukan jalur terpendek dari satu kata ke kata lain dengan mengubah satu huruf pada satu waktu. g(n) adalah biaya aktual dari node awal ke node n dalam word ladder. Dalam konteks word ladder, ini adalah jumlah langkah atau perubahan kata yang telah dilakukan untuk mencapai node n dari kata awal. h(n) adalah heuristik atau perkiraan biaya dari node n ke node tujuan. Dalam word ladder, heuristik ini berupa jumlah huruf yang berbeda antara kata pada node n dan kata tujuan. Heuristik harus dirancang sedemikian rupa sehingga tidak pernah melebihi estimasi biaya sebenarnya, yang menjadikannya admissible. f(n) adalah fungsi evaluasi yang digunakan untuk memprioritaskan node mana yang harus dieksplorasi selanjutnya dalam open set. f(n) dihitung dengan menambahkan g(n) dan h(n) bersama-sama, yaitu f(n)=g(n)+h(n). Ini memberikan estimasi total biaya terendah dari node awal ke tujuan melalui node n.

Heuristik yang digunakan pada algoritma A* adalah admissible jika estimasi biaya dari simpul saat ini ke simpul tujuan tidak pernah melebihi biaya sebenarnya yang diperlukan untuk mencapai tujuan. Dalam konteks Word Ladder, heuristik yang digunakan adalah estimasi jarak yaitu perbedaan setiap huruf pada simpul sebuah kata dengan simpul kata target. Jika estimasi jarak tersebut tidak pernah melebihi jarak sebenarnya (jumlah langkah yang diperlukan), maka heuristik tersebut admissible. Oleh karena itu, heuristik yang digunakan pada algoritma A* dalam kasus word ladder adalah admissible.

Secara teoritis, algoritma A* dapat lebih efisien daripada algoritma Uniform Cost Search (UCS) dalam kasus word ladder karena A* menggunakan heuristik untuk memandu pencarian. Ini memungkinkan A* untuk mengabaikan jalur yang tidak mungkin mengarah ke solusi yang optimal, sedangkan UCS akan menjelajahi semua jalur dengan biaya yang sama tanpa mempertimbangkan tujuan akhir. Oleh karena itu, A* sering kali mencapai solusi lebih cepat daripada UCS, terutama dalam kasus dengan ruang pencarian yang besar.

Bab 4 Source Code

4.1. Package Algorithm

4.1.1. Ucs.java

```
package algorithm;
import java.util.*;
import utils.*;

public class Ucs {
    private Integer nodeVisited;
    private List<String> solution;

    public List<String> getSolution(){
        return solution;
    }

    public Integer getNodeVisited(){
        return nodeVisited;
    }

     public void ladderPathUcs(String start, String end, HashSet<String> dict)
{

        LinkedList<String> wordQueue = new LinkedList<>();
        LinkedList<Integer> distanceQueue = new LinkedList<>();
        LinkedList<LinkedList<String>> pathsQueue = new LinkedList<>();
        LinkedList<String> solution = new LinkedList<>();

        wordQueue.add(start);
        distanceQueue.add(1);
        dict.remove(start);

        LinkedList<String> path = new LinkedList<>();
        path.add(start);
        pathsQueue.add(path);
```

```
        int result = Integer.MAX_VALUE;
        int nodeVisited = 0;

        while (!wordQueue.isEmpty()) {
            String currWord = wordQueue.pop();
            Integer currDistance = distanceQueue.pop();
            LinkedList<String> currentPathQueue = pathsQueue.pop();
            nodeVisited ++;

            if (currWord.equals(end)) {
                if (currDistance < result) {
                    result = currDistance;
                    solution = currentPathQueue;
                }
            }

            List<String> neighbors = Neighbors.getNeighbors(currWord, dict);
            for (String neighbor : neighbors) {
                wordQueue.add(neighbor);
                distanceQueue.add(currDistance + 1);
                dict.remove(neighbor);

                                        LinkedList<String>  newPath  =  new
LinkedList<>(currentPathQueue);
                newPath.add(neighbor);
                pathsQueue.add(newPath);
            }
        }

        this.solution = solution;
        this.nodeVisited = nodeVisited;

    }
}
```

Penjelasan kelas dan method:

Kelas Ucs mengimplementasikan algoritma UCS (Uniform Cost Search) untuk mencari jalur terpendek antara dua kata dalam kamus berdasarkan biaya langkah terkecil. Method ladderPathUcs digunakan untuk mencari jalur tersebut, sementara getNodeVisited mengembalikan jumlah node yang telah dikunjungi selama pencarian, dan getSolution mengembalikan jalur terpendek yang ditemukan.

4.1.2. Gbfs.java

```
package algorithm;
import java.util.*;
import utils.*;

public class Gbfs {
    private Integer nodeVisited;
    private List<String> solution;

    public List<String> getSolution(){
        return solution;
    }

    public Integer getNodeVisited(){
        return nodeVisited;
    }

    public void ladderPathGbfs(String start, String end, HashSet<String> dict)
{

        boolean isFound = false;
                                PriorityQueue<String>    wordQueue    =    new
PriorityQueue<>(Comparator.comparingInt(s -> Heuristic.heuristic(s, end)));
        Map<String, String> cameFrom = new HashMap<>();
        Set<String> visited = new HashSet<>();
        wordQueue.add(start);
        visited.add(start);

        while (!wordQueue.isEmpty()) {
            String currWord = wordQueue.poll();
            if (currWord.equals(end)) {
                this.solution = buildPath(cameFrom, end);
```

```
                isFound = true;
                break;
            }

            for (String neighbor : Neighbors.getNeighbors(currWord, dict)) {
                if (!visited.contains(neighbor)) {
                    wordQueue.add(neighbor);
                    visited.add(neighbor);
                    cameFrom.put(neighbor, currWord);
                }
            }
        }

        if(!isFound){
            this.solution = new ArrayList<>();
        }

        this.nodeVisited = visited.size();
    }

    private static List<String> buildPath(Map<String, String> cameFrom, String
end) {
        LinkedList<String> path = new LinkedList<>();
        String current = end;
        while (current != null) {
            path.addFirst(current);
            current = cameFrom.get(current);
        }
        return path;
    }

}
```

Penjelasan kelas dan method:

Kelas Gbfs mengimplementasikan algoritma GBFS (Greedy Best-First Search) untuk mencari jalur terpendek antara dua kata dalam kamus berdasarkan estimasi heuristik terbaik. Method ladderPathGbfs digunakan untuk mencari jalur tersebut, getNodeVisited

mengembalikan jumlah node yang telah dikunjungi selama pencarian, dan getSolution mengembalikan jalur terpendek yang ditemukan.

4.1.3. Astar.java

```
package algorithm;
import java.util.*;
import utils.*;

public class Astar {
    private Integer nodeVisited;
    private List<String> solution;

    public List<String> getSolution(){
        return solution;
    }

    public Integer getNodeVisited(){
        return nodeVisited;
    }

      public void ladderPathAstar(String start, String end, HashSet<String>
dict) {

        boolean isFound = false;

        int nodeVisited = 0;
                                    PriorityQueue<Node>     openSet     =     new
PriorityQueue<>(Comparator.comparingInt(n -> n.fCost));
        Map<String, Node> allNodes = new HashMap<>();

         Node startNode = new Node(start, null, 0, Heuristic.heuristic(start,
end));
        allNodes.put(start, startNode);
        openSet.add(startNode);

        while (!openSet.isEmpty()) {
            Node current = openSet.poll();
            nodeVisited ++;
```

```
            if (current.word.equals(end)) {
                this.solution = buildPath(current);
                isFound = true;
                break;
            }

            for (String neighbor : Neighbors.getNeighbors(current.word, dict))
{
                    if (!allNodes.containsKey(neighbor) || current.gCost + 1 <
allNodes.get(neighbor).gCost) {
                        Node neighborNode = allNodes.computeIfAbsent(neighbor, n
-> new Node(n, current, dict.size(), Heuristic.heuristic(n, end)));
                    neighborNode.cameFrom = current;
                    neighborNode.gCost = current.gCost + 1;
                                    neighborNode.fCost  =  neighborNode.gCost  +
neighborNode.hCost;
                    openSet.add(neighborNode);
                }
            }
        }

        if(!isFound){
            this.solution = new ArrayList<>();
        }
        this.nodeVisited = nodeVisited;
    }

    private static List<String> buildPath(Node endNode) {
        LinkedList<String> path = new LinkedList<>();
        Node current = endNode;
        while (current != null) {
            path.addFirst(current.word);
            current = current.cameFrom;
        }
        return path;
    }

    private static class Node {
```

```
        String word;
        Node cameFrom;
        int gCost;
        int hCost;
        int fCost;

        Node(String word, Node cameFrom, int gCost, int hCost) {
            this.word = word;
            this.cameFrom = cameFrom;
            this.gCost = gCost;
            this.hCost = hCost;
            this.fCost = gCost + hCost;
        }
    }
}
```

Penjelasan kelas dan method:

Kelas Astar mengimplementasikan algoritma A* untuk mencari jalur terpendek antara dua kata dalam kamus dengan mempertimbangkan biaya langkah dan estimasi heuristik. Method ladderPathAstar digunakan untuk mencari jalur tersebut, getNodeVisited mengembalikan jumlah node yang telah dikunjungi selama pencarian, dan getSolution mengembalikan jalur terpendek yang ditemukan.

4.2. Package Utils

4.2.1. Heuristic.java

```
package utils;

public class Heuristic {
    public static int heuristic(String word, String target) {
        int diffCount = 0;
        for (int i = 0; i < word.length(); i++) {
            if (word.charAt(i) != target.charAt(i)) {
                diffCount++;
            }
        }
        return diffCount;
```

```
    }
}
```

Penjelasan kelas dan method:

Kelas Heuristic menyediakan sebuah method static bernama heuristic yang menghitung estimasi heuristik antara dua kata dengan menghitung jumlah karakter yang berbeda pada posisi yang sama antara kata pertama dan kata target. Semakin sedikit perbedaan, semakin dekat kedua kata secara heuristik.

4.2.2. Neighbors.java

```java
package utils;

import java.util.*;

public class Neighbors {
    public static List<String> getNeighbors(String word, Set<String> dict) {
        List<String> neighbors = new ArrayList<>();
        for (int i = 0; i < word.length(); i++) {
            char[] currCharArr = word.toCharArray();
            for (char c = 'a'; c <= 'z'; c++) {
                currCharArr[i] = c;
                String newWord = new String(currCharArr);
                if (dict.contains(newWord)) {
                    neighbors.add(newWord);
                }
            }
        }
        return neighbors;
    }
}
```

Penjelasan kelas dan method:

Kelas Neighbors menyediakan sebuah method static bernama getNeighbors yang mengembalikan daftar kata tetangga dari sebuah kata tertentu berdasarkan kamus yang diberikan. Method ini mengganti setiap karakter dalam kata dengan setiap huruf

dari 'a' hingga 'z' satu per satu, dan jika kata baru yang dihasilkan terdapat dalam kamus, kata tersebut ditambahkan ke dalam daftar tetangga.

4.3. Main.java

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.*;

import algorithm.*;

public class Main {
    public static void main(String[] args) {
          System.out.println("----------------------------- WordLadder Solver
-----------------------------");
        Scanner input = new Scanner(System.in);
        HashSet<String> dictionary = readDictionaryFromFile("words.txt");

        do {
            String startWord, endWord;
            do {
                System.out.print("Start word    : ");
                startWord = input.nextLine().toLowerCase();
                System.out.print("End word      : ");
                endWord = input.nextLine().toLowerCase();

                                    if  (!dictionary.contains(startWord)  ||
!dictionary.contains(endWord)) {
                        System.out.println("Start word or end word is not in
dictionary, please input again.");
                    continue;
                }

                if (startWord.length() != endWord.length()) {
                    System.out.println("Start word and end word must have same
length, please input again.");
                    continue;
                }
```

```
                break;
            } while (true);

            long startTimeUcs = System.currentTimeMillis();
            Ucs SolveUcs = new Ucs();
                                SolveUcs.ladderPathUcs(startWord,   endWord,   new
HashSet<>(dictionary));
            long endTimeUcs = System.currentTimeMillis();
            List<String> pathUcs = SolveUcs.getSolution();

            long startTimeGbfs = System.currentTimeMillis();
            Gbfs SolveGbfs = new Gbfs();
                               SolveGbfs.ladderPathGbfs(startWord,   endWord,   new
HashSet<>(dictionary));
            long endTimeGbfs = System.currentTimeMillis();
            List<String> pathGbfs = SolveGbfs.getSolution();

            long startTimeAstar = System.currentTimeMillis();
            Astar SolveAstar = new Astar();
                              SolveAstar.ladderPathAstar(startWord,   endWord,   new
HashSet<>(dictionary));
            long endTimeAstar = System.currentTimeMillis();
            List<String> pathAstar = SolveAstar.getSolution();

                System.out.println("---------------------------------- RESULT
----------------------------------");
            System.out.println("Uniform Cost Search");
            printPath(pathUcs, startTimeUcs, endTimeUcs);
            System.out.print("Node visited count: ");
            System.out.println(SolveUcs.getNodeVisited());

System.out.println("------------------------------------------------------------
------------------");
            System.out.println("Greedy Best First Search");
            printPath(pathGbfs, startTimeGbfs, endTimeGbfs);
            System.out.print("Node visited count: ");
            System.out.println(SolveGbfs.getNodeVisited());
```

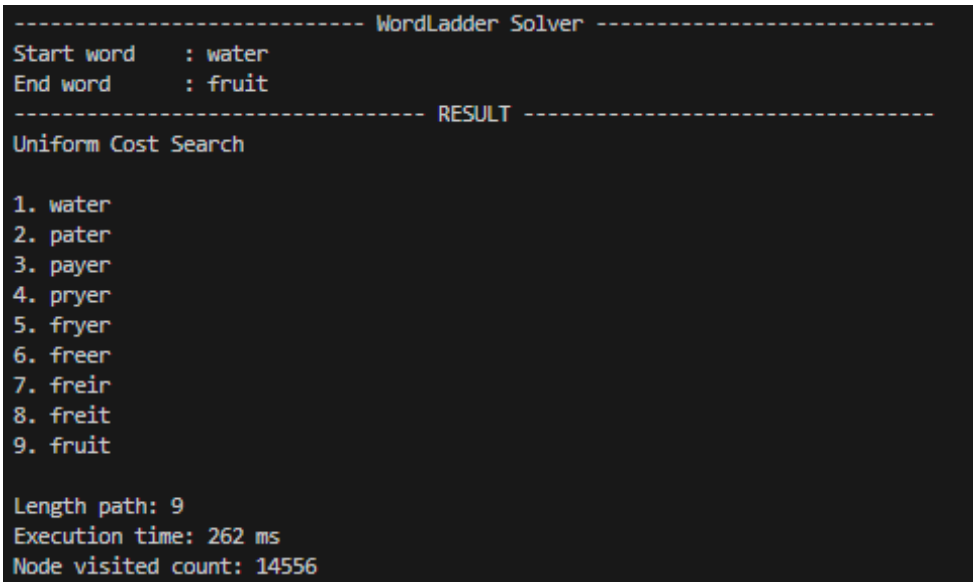
```
System.out.println("------------------------------------------------------------
------------------");
            System.out.println("A star");
            printPath(pathAstar, startTimeAstar, endTimeAstar);
            System.out.print("Node visited count: ");
            System.out.println(SolveAstar.getNodeVisited());

System.out.println("------------------------------------------------------------
------------------");

            System.out.print("Do you want to run the program again? (y/n): ");
            String choice = input.nextLine().toLowerCase();
            if (!choice.equals("y")) {
                break;
            }
        } while (true);

        input.close();
    }



    private static HashSet<String> readDictionaryFromFile(String filename) {
        HashSet<String> dictionary = new HashSet<>();
        try {
            File file = new File(filename);
            Scanner scanner = new Scanner(file);
            while (scanner.hasNextLine()) {
                String word = scanner.nextLine().trim();
                dictionary.add(word);
            }
            scanner.close();
        } catch (FileNotFoundException e) {
            System.out.println("File tidak ditemukan.");
            e.printStackTrace();
        }
        return dictionary;
```

```
    }

      private static void printPath(List<String> path, long startTime, long
endTime) {
        if (!path.isEmpty()) {
            System.out.println();
            for (int i = 0; i < path.size(); i++) {
                System.out.println((i + 1) + ". " + path.get(i));
            }
            System.out.println();
            System.out.println("Length path: " + path.size());
        } else {
            System.out.println("No solution found.");
        }
           System.out.println("Execution time: " + (endTime - startTime) + "
ms");
    }

}
```

Penjelasan kelas dan method:

Kelas Main adalah kelas utama yang menjalankan program WordLadder Solver. Program ini memungkinkan pengguna untuk memasukkan kata awal dan kata akhir, kemudian menjalankan tiga algoritma berbeda (UCS, GBFS, A*) untuk mencari jalur terpendek antara dua kata tersebut. Program membaca kamus kata-kata dari file "words.txt", kemudian menampilkan solusi dari setiap algoritma beserta jumlah node yang dikunjungi dan waktu eksekusi.

Bab 5 Uji Coba Program

No	Result
1.	

```
--------------------------------------------------------------------------------
Greedy Best First Search

1. water
2. bater
3. bates
4. fates
5. fated
6. feted
7. feued
8. flued
9. fluer
10. flurr
11. flurt
12. blurt
13. bluet
14. bruet
15. bruit
16. fruit

Length path: 16
Execution time: 33 ms
Node visited count: 250
--------------------------------------------------------------------------------
A star

1. water
2. pater
3. payer
4. pryer
5. fryer
6. freer
7. freir
8. freit
9. fruit

Length path: 9
Execution time: 48 ms
Node visited count: 788
--------------------------------------------------------------------------------
```

2.

```
Start word    : paper
End word      : place
--------------------------------- RESULT ---------------------------------
Uniform Cost Search

1. paper
2. paler
3. pales
4. peles
5. pelts
6. peats
7. plats
8. plate
9. place

Length path: 9
Execution time: 618 ms
Node visited count: 14556
--------------------------------------------------------------------------
Greedy Best First Search

1. paper
2. paler
3. palew
4. pelew
5. peles
6. pelts
7. peats
8. plats
9. plate
10. place

Length path: 10
Execution time: 25 ms
Node visited count: 135
--------------------------------------------------------------------------
```

```
A star

1. paper
2. paler
3. pales
4. peles
5. pelts
6. peats
7. plats
8. plate
9. place

Length path: 9
Execution time: 98 ms
Node visited count: 706
```

3.

```
Start word    : apple
End word      : quiet
---------------------------------- RESULT ----------------------------------
Uniform Cost Search

1. apple
2. ample
3. amole
4. atole
5. stole
6. stile
7. stine
8. suine
9. suint
10. quint
11. quiet

Length path: 11
Execution time: 197 ms
Node visited count: 14556
----------------------------------------------------------------------------
```

```
Greedy Best First Search

1. apple
2. apply
3. appay
4. appal
5. aptal
6. attal
7. attar
8. atter
9. otter
10. outer
11. muter
12. muser
13. muset
14. mulet
15. culet
16. curet
17. buret
18. burst
19. buist
20. quist
21. quiet

Length path: 21
Execution time: 21 ms
Node visited count: 542
--------------------------------------------------------------------------------
```

```
A star

1. apple
2. ample
3. amole
4. atole
5. stole
6. stile
7. stilt
8. stint
9. suint
10. quint
11. quiet

Length path: 11
Execution time: 39 ms
Node visited count: 153
--------------------------------------------------------------------------------
```

4.

```
Start word    : ant
End word      : car
---------------------------------- RESULT ----------------------------------
Uniform Cost Search

1. ant
2. jnt
3. jat
4. cat
5. car

Length path: 5
Execution time: 35 ms
Node visited count: 2128
--------------------------------------------------------------------------------
Greedy Best First Search

1. ant
2. any
3. ary
4. cry
5. cay
6. car

Length path: 6
Execution time: 47 ms
Node visited count: 107
--------------------------------------------------------------------------------
A star

1. ant
2. jnt
3. jat
4. cat
5. car

Length path: 5
Execution time: 20 ms
Node visited count: 37
--------------------------------------------------------------------------------
```

5.

```
Start word    : door
End word      : book
---------------------------------- RESULT ----------------------------------
Uniform Cost Search

1. door
2. boor
3. book

Length path: 3
Execution time: 99 ms
Node visited count: 7081
--------------------------------------------------------------------------------
Greedy Best First Search

1. door
2. boor
3. book

Length path: 3
Execution time: 16 ms
Node visited count: 25
--------------------------------------------------------------------------------
A star

1. door
2. boor
3. book

Length path: 3
Execution time: 25 ms
Node visited count: 4
--------------------------------------------------------------------------------
```

6.

```
Start word    : drive
End word      : cycle
---------------------------------- RESULT ----------------------------------
Uniform Cost Search

1. drive
2. wrive
3. waive
4. warve
5. carve
6. carle
7. cable
8. coble
9. cocle
10. cycle

Length path: 10
Execution time: 184 ms
Node visited count: 14556
--------------------------------------------------------------------------------
Greedy Best First Search

1. drive
2. drave
3. crave
4. chave
5. cheve
6. chese
7. chuse
8. cause
9. carse
10. carle
11. cable
12. coble
13. cocle
14. cycle

Length path: 14
Execution time: 23 ms
Node visited count: 252
--------------------------------------------------------------------------------
```

```
A star

1. drive
2. wrive
3. waive
4. warve
5. carve
6. carle
7. cable
8. coble
9. cocle
10. cycle

Length path: 10
Execution time: 43 ms
Node visited count: 647
--------------------------------------------------------------------------------
```

Bab 6 Hasil Analisis Perbandingan Algoritma

Words (start -> end)	Length path			Execution time (ms)			Node visited count		
	UCS	GBFS	A*	UCS	GBFS	A*	UCS	GBFS	A*
water -> fruit	9	16	9	262	33	48	14556	250	788
paper -> place	9	10	9	618	25	98	14556	135	706
apple -> quiet	11	21	11	197	21	39	14556	542	153
ant -> car	5	6	5	35	47	20	2128	107	37
door -> book	3	3	3	99	16	25	7081	25	4
drive -> cycle	10	14	10	184	23	43	14556	252	647

Dapat dilihat dari tabel di atas, length path dari UCS dan A* sama, sedangkan GBFS kebanyakan lebih besar, hal ini menandakan bahwa UCS dan A* sama sama optimal, namun GBFS tidak optimal. UCS dan A* dikatakan optimal karena keduanya dapat menemukan solusi optimal, yaitu solusi dengan biaya terendah. UCS secara khusus menjamin solusi optimal karena selalu memilih jalur dengan biaya terendah pada setiap langkah, sedangkan A* menggunakan fungsi heuristik untuk memprioritaskan langkah-langkah yang lebih mungkin mengarah ke solusi optimal. GBFS, di sisi lain, tidak dijamin optimal. GBFS hanya mengikuti heuristik yang mengarah ke simpul yang paling menjanjikan menurut estimasi jarak ke tujuan, tanpa mempertimbangkan biaya sejauh ini. Ini dapat menyebabkan GBFS terjebak dalam loop atau memilih jalur yang lebih mahal.

Kemudian dari segi waktu eksekusi, GBFS paling cepat atau dengan kata lain jika diurutkan dari yang terkecil yaitu GBFS, A*, dan UCS. GBFS dan A* cenderung cepat dikarenakan mereka melakukan pencarian path berdasarkan fungsi heuristik sehingga meminimalkan pencarian node. Kemudian jika memori yang dibutuhkan UCS paling banyak jika dilihat dari jumlah node yang dikunjungi karena UCS cenderung mempertimbangkan setiap simpul secara merata, sedangkan GBFS dan A* cenderung menggunakan memori dengan lebih efisien karena mereka dapat menggunakan informasi heuristik untuk membatasi pencarian path.

Bab 7 Lampiran

7.1. Checklist Poin

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		✔

7.2. Link Repository

https://github.com/randyver/Tucil3_13522067