Tugas Kecil 3 IF2211

Strategi Algoritma Semester II tahun 2023/2024

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

Disusun oleh:

Fedrianz Dharma 13522090

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

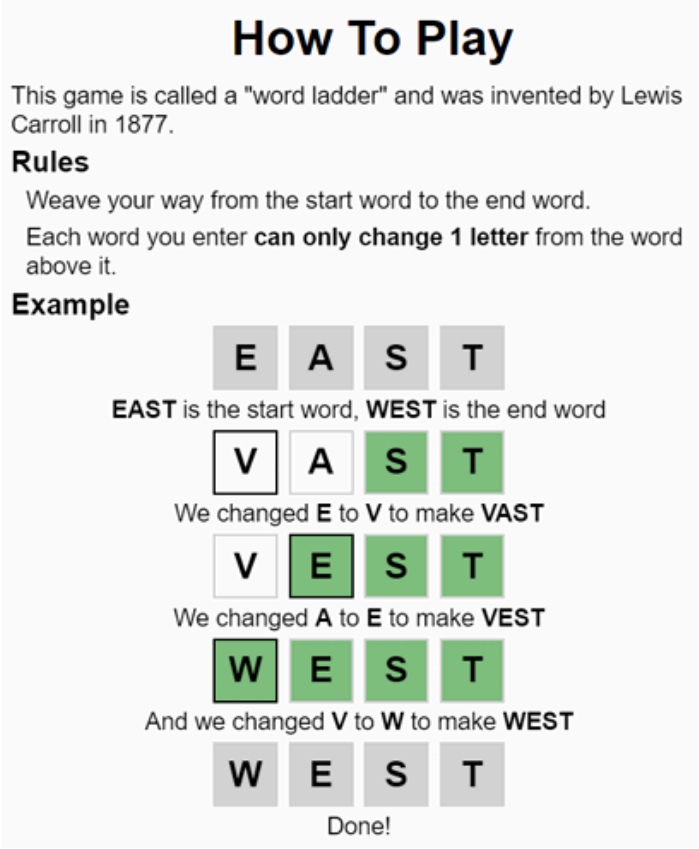
Institut Teknologi Bandung

2024

Daftar Isi

Bab I
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.



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

Bab II
Analisis dan Implementasi dalam Algoritma UCS, Greedy Best First Search, dan A*

2.1 Struktur Data Node

Implementasi simpul pada algoritma akan menggunakan *class Node* yang memiliki atribut *pred* yang menyimpan Node *parent*-nya. Oleh karena itu, rute yang dilalui dari simpul awal hingga suatu simpul bisa didapatkan dengan melakukan *traversal* menggunakan atribut *pred*.

2.2 Analisis Algoritma Uniform Cost Search

Uniform Cost Search (UCS) adalah algoritma *uninformed search* yang bertujuan untuk mendapatkan rute atau lintasan dari simpul awal hingga mencapai simpul tujuan pada graf dengan *cost* atau ongkos yang terendah. ***Cost* yang dibutuhkan dari simpul awal hingga suatu simpul n dinyatakan dengan notasi *g(n).*** Sehingga pada algoritma UCS, fungsi evaluasi f(n) = g(n).

Dengan menggunakan notasi g(n) sebagai prioritas dalam mengunjungi suatu simpul, algoritma UCS menjamin solusi yang ditemukan adalah **solusi yang optimal**. Oleh karena itu, struktur data yang cocok digunakan mengimplementasikan urutan simpul yang akan dikunjungi berdasarkan prioritas adalah *priority queue*.

Untuk menyelesaikan permasalahan *Word Ladder*, langkah-langkah algoritma UCS adalah sebagai berikut:

1. Tentukan *start word* dan *end word* yang akan dijadikan sebagai simpul awal dan simpul tujuan pada graf.
2. Buat *priority queue* terurut menaik untuk menampung simpul-simpul yang akan dikunjungi berdasarkan nilai f(n) simpul tersebut. **Dengan nilai f(n) = g(n). *Cost* antar simpul akan bernilai 1 sehingga g(n) akan bertambah 1 setiap berpindah simpul.**
3. Jika *priority queue* tidak kosong, lakukan *dequeue* dan simpan simpul tersebut.
4. Jika simpul tersebut merupakan *end word* atau simpul tujuan maka kembalikan rute yang didapatkan dari simpul awal hingga simpul tersebut.
5. Jika simpul tersebut bukan simpul tujuan, ekspan simpul tersebut dan masukkan simpul-simpul hasil ekspan yang belum pernah di ekspan ke dalam *priority queue*.
6. Ulangi langkah 3-5 hingga ditemukan *priority queue* kosong atau ditemukan solusi. Jika *priority queue* kosong, artinya tidak ada rute dari simpul awal hingga simpul tujuan yang ditemukan.

Pada kasus *word ladder*, algoritma UCS akan membangkitkan *node* dengan urutan yang sama dengan algoritma BFS. Karena pada kasus *Word Ladder cost* dari suatu simpul ke simpul lainnya adalah 1 atau ***steps = cost***. Urutan pada *priority queue* juga akan terurut sesuai dengan level atau *depth* dari masing-masing simpul sama seperti algoritma BFS. Sehingga algoritma UCS akan mengunjungi semua simpul pada suatu *depth* sebelum mengunjungi simpul yang ada pada *depth* berikutnya.

Hal ini sama dengan algoritma BFS yang akan akan mengunjungi semua simpul pada suatu *depth* sebelum mengunjungi simpul yang ada pada *depth* berikutnya. Karena urutan simpul yang dikunjungi sama, maka algoritma UCS juga akan menghasilkan *path* yang sama dengan algoritma BFS. Oleh karena itu, algoritma UCS pada kasus *Word Ladder* sama dengan algoritma BFS.

2.3 Analisis Algoritma Greedy Best First Search

Greedy Best First Search merupakan algoritma *informed search* yang bertujuan untuk mendapatkan rute dari simpul awal hingga mencapai simpul akhir pada graf. Pada algoritma ini, **estimasi *cost* antara suatu simpul n ke simpul akhir dinyatakan dalam notasi *h(n)* sudah diketahui sebelum pencarian dimulai. Estimasi *cost* yang dibutuhkan ini disebut sebagai *heuristic*. Algoritma Greedy Best First Search tidak menjamin ditemukan solusi yang optimal karena tidak *complete* dan bisa terjebak pada lokal minima.**

Sama seperti algoritma UCS, algoritma ini juga menggunakan skema prioritas. Hanya saja, pada algoritma Greedy Best First Search, nilai prioritas yang digunakan adalah nilai *h(n)*. Oleh karena itu, algoritma ini juga akan menggunakan struktur data *priority queue.*

Untuk menyelesaikan permasalahan *Word Ladder*, langkah-langkah algoritma Greedy Best First Search adalah sebagai berikut:

1. Tentukan *start word* dan *end word* yang akan dijadikan sebagai simpul awal dan simpul tujuan pada graf.
2. Buat *priority queue* terurut menaik untuk menampung simpul-simpul yang akan dikunjungi berdasarkan nilai f(n) simpul tersebut. **Dengan nilai f(n) = h(n). h(n) yang digunakan akan jumlah perbedaan karakter pada simpul n dengan *end word* pada urutan yang sama.**
3. Jika *priority queue* tidak kosong, lakukan *dequeue* dan simpan simpul tersebut.
4. Jika simpul tersebut merupakan *end word* atau simpul tujuan maka kembalikan rute yang didapatkan dari simpul awal hingga simpul tersebut.

5. Jika simpul tersebut bukan simpul tujuan, ekspan simpul tersebut dan masukkan simpul-simpul hasil ekspan yang belum pernah di ekspan ke dalam *priority queue*.
6. Ulangi langkah 3-5 hingga ditemukan *priority queue* kosong atau ditemukan solusi. Jika *priority queue* kosong, artinya tidak ada rute dari simpul awal hingga simpul tujuan yang ditemukan.

2.4 Analisis Algoritma A*

Algoritma A* adalah algoritma *informed search* yang bertujuan untuk mendapatkan rute dari simpul awal hingga mencapai simpul akhir pada graf. Algoritma A* akan menggunakan informasi ***cost* sejauh ini dari *root* ke suatu simpul n (*g(n)*)** dan **estimasi *cost* dari simpul n ke *goal* atau *end word* (*h(n)*)**. Kedua informasi tersebut akan digabungkan untuk membentuk fungsi evaluasi ***f(n) = g(n) + h(n)***. ***f(n)* merupakan estimasi *total cost* menuju simpul n hingga *goal*.**

Untuk menyelesaikan permasalahan *Word Ladder*, langkah-langkah algoritma A* adalah sebagai berikut:

1. Tentukan *start word* dan *end word* yang akan dijadikan sebagai simpul awal dan simpul tujuan pada graf.
2. Buat *priority queue* terurut menaik untuk menampung simpul-simpul yang akan dikunjungi berdasarkan nilai f(n) simpul tersebut. **Dengan nilai f(n) = g(n) + h(n). *Cost* antar simpul akan bernilai 1 sehingga g(n) akan bertambah 1 setiap berpindah simpul. Nilai h(n) yang digunakan akan jumlah perbedaan karakter pada simpul n dengan *end word* pada urutan yang sama.**
3. Jika *priority queue* tidak kosong, lakukan *dequeue* dan simpan simpul tersebut.
4. Jika simpul tersebut merupakan *end word* atau simpul tujuan maka kembalikan rute yang didapatkan dari simpul awal hingga simpul tersebut.
5. Jika simpul tersebut bukan simpul tujuan, ekspan simpul tersebut dan masukkan simpul-simpul hasil ekspan yang belum pernah di ekspan ke dalam *priority queue*.
6. Ulangi langkah 3-5 hingga ditemukan *priority queue* kosong atau ditemukan solusi. Jika *priority queue* kosong, artinya tidak ada rute dari simpul awal hingga simpul tujuan yang ditemukan.

Heuristik yang *admissible* adalah jika untuk setiap simpul n, nilai heuristiknya lebih kecil atau sama dengan *cost* sebenarnya yang dibutuhkan untuk mencapai *goal* dari simpul n. Heuristik yang digunakan pada algoritma A* *admissible* karena jumlah perbedaan karakter pada simpul n dengan *end word* pada tiap slotnya tidak akan pernah melebihi jumlah *cost* yang dibutuhkan untuk

mengubah *word* pada simpul n menjadi *end word*. Dengan menggunakan heuristik yang *admissible*, algoritma A* akan dijamin selalu menemukan solusi yang optimal.

Secara teoritis, pada kasus *Word Ladder* walaupun algoritma UCS dan A* sama-sama menjamin solusi yang optimal, namun algoritma A* lebih efisien. Karena algoritma A* menggunakan f(n) = g(n) + h(n) untuk mengetahui estimasi total *cost* menuju simpul n hingga *goal*. Sehingga rute dengan estimasi total *cost* yang mahal tidak akan diekspan, sedangkan algoritma UCS hanya membandingkan *cost* yang dibutuhkan dari simpul awal hingga simpul n f(n) = g(n). Dengan begitu, algoritma A* akan mengekspan lebih sedikit simpul dibanding algoritma UCS sehingga lebih efisien.

Bab III
Implementasi Algoritma dalam Bahasa Java

3.1 Class Node

```
class Node{
    private String w;
    private int fn;
    private int gn;
    private int hn;
    public Node pred;
    public static int nodeCreated = 0;
    public static int nodeVisited = 0;

    Node(String s, int cost, int hn, Node pred){
        this.w = s;
        this.pred = pred;
        this.hn = hn;
        // root node
        if(pred == null){
            this.gn = 0;
            this.fn = 0;
        }else{ // child node
            this.gn = this.pred.gn + cost;
            this.fn = this.gn + this.hn;
        }
        nodeCreated++;
    }
```

Class Node digunakan untuk merepresentasikan simpul pada pohon pencarian. Atribut String w digunakan untuk menyimpan kata. Atribut fn digunakan untuk menyimpan fungsi evaluasi. Atribut gn digunakan untuk menyimpan *cost* yang dibutuhkan untuk mencapai *Node* dari *root*. Atribut hn digunakan untuk menyimpan nilai fungsi heuristik yang digunakan. Atribut pred digunakan untuk mendapatkan Node parent-nya. Atribut nodeCreated digunakan untuk mengetahui total *Node* yang diciptakan. Atribut nodeVisited digunakan untuk menyimpan berapa kali *Node* diekspan.

Constructor Node untuk UCS pada bagian int hn akan diisi dengan 0, sedangkan untuk GBFS pada bagian int *cost* akan diisi dengan 0. Untuk A* int *cost* dan int hn akan diisi sesuai dengan kebutuhan.

```
    public static void resetNode(){
            nodeCreated = 0;
            nodeVisited = 0;
        }
        public String getString(){
            return w;
        }
        public List<Node> pathToRoot(){
            List<Node> l = new ArrayList<>();
            Node currNode = this;
            while(currNode != null){
                l.add(currNode);
                currNode = currNode.pred;
            }
            Collections.reverse(l);
            return l;
        }
        public int getFn(){
            return fn;
        }
public List<String> getExpandNode(Map<String, Boolean> m){
        String alphabet = "abcdefghijklmnopqrstuvwxyz";
        List<String> l = new ArrayList<>();
        for(int i = 0; i < w.length(); i++){
            for(int j = 0; j < alphabet.length(); j++){
                if(w.charAt(i) == alphabet.charAt(j)){
                    continue;
                }
                StringBuilder wtmp = new StringBuilder(w);
                wtmp.setCharAt(i, alphabet.charAt(j));
                String newWord = wtmp.toString();
```

```
                // jika ada di dalam kamus, tambahkan ke dalam
list
                if(m.containsKey(newWord)){
                    l.add(newWord);
                }
            }
        }
        return l;
    }
```

- Method resetNode digunakan untuk mereset atribut static dari kelas Node
- Method getString digunakan untuk mendapatkan atribut String w pada objek Node
- Method pathToRoot digunakan untuk mendapatkan rute dari suatu objek Node hingga *root*-nya
- Method getFn digunakan untuk mendapatkan fn dari suatu objek Node
- Method getExpandNode digunakan untuk mendapat list yang berisi hasil ekspan dari sebuah Node

3.2 Class NodeComparator

```
class NodeComparator implements Comparator<Node>{
    public int compare(Node n1, Node n2) {
        if (n1.getFn() < n2.getFn()){
            return -1;
        } else if (n1.getFn() > n2.getFn()){
            return 1;
        } else{
            return 0;
        }
    }
}
```

Kelas NodeComparator digunakan untuk mengurutkan objek Node berdasarkan nilai atribut fn-nya pada *Priority Queue*.

3.3 Class UtilityFunc

Kelas UtilityFunc berisi method-method yang digunakan untuk membantu proses pemrograman.

```
public static int getDifference(String w1, String w2){
```

```
        int diff = 0;
        for(int i = 0; i < w1.length(); i++){
            if(w1.charAt(i) != w2.charAt(i)){
                diff++;
            }
        }
        return diff;
    }
```

Method getDifference akan mengembalikan jumlah karakter yang berbeda antara kedua String pada urutan yang sama. Method ini digunakan untuk menghitung nilai dari fungsi heuristik.

```
public static void printString(String s1, String s2){
        for(int i = 0; i < s1.length(); i++){
            if(s1.charAt(i) == s2.charAt(i)){
                System.out.print(ANSI_GREEN + s1.charAt(i)
+ ANSI_RESET);
            }else{
                System.out.print(s1.charAt(i));
            }
        }
        System.out.println();
    }
```

Method printString digunakan untuk *print* sebuah String s1 dengan tiap karakter yang sama dengan String s2 pada urutan yang sama dengan warna hijau.

```
 public static void printList(List<Node> l){
        String endword = l.get(l.size()-1).getString();
        l.forEach(s -> printString(s.getString(),
endword));
    }
```

Method printList digunakan untuk print semua String dari Node yang ada pada List dengan menggunakan bantuan method printString.

```
public static List<String> readInputStartEnd(){
        String startword;
        String endword;
        try{
            BufferedReader bufferedReader = new
BufferedReader(new InputStreamReader(System.in));
            System.out.print("Enter start word: ");
            startword =
bufferedReader.readLine().trim().toLowerCase();

while((!startword.chars().allMatch(Character::isLetter)) ||
startword.length() == 0 ){
                System.out.print("Enter start word: ");
                startword =
bufferedReader.readLine().trim().toLowerCase();
            }
            System.out.print("Enter end word: ");
            endword =
bufferedReader.readLine().trim().toLowerCase();

while((!endword.chars().allMatch(Character::isLetter)) ||
endword.length() == 0 || endword.length() !=
startword.length()){
                System.out.print("Enter end word: ");
                endword =
bufferedReader.readLine().trim().toLowerCase();
            }
            List<String> l = new ArrayList<>();
            l.add(startword);
            l.add(endword);
            return l;
        }
        catch (Exception e) {
            System.out.println("Terjadi kesalahan");
```

```
            System.out.println(e.getMessage());
            return null;
        }
    }
```

Method readInputStartEnd digunakan untuk mendapatkan *start word* dan *end word* dari masukan pengguna.

```
    public static int readAlgo(){
        try{
            BufferedReader bufferedReader = new
BufferedReader(new InputStreamReader(System.in));
            System.out.println("Masukkan algoritma yang
diinginkan: ");
            System.out.println("1. Uniform Cost Search
Algorithm");
            System.out.println("2. Greedy Best First Search
Algorithm");
            System.out.println("3. A* Algorithm\n");
            System.out.print("Masukkan pilihan algoritma:
");
            int choice = 0;
            try{
                choice =
Integer.parseInt(bufferedReader.readLine().trim());
            }catch(Exception e){
                System.out.println("\nInput harus berupa
angka");
            }
            while(choice < 1 || choice > 3){
                System.out.println("\nPilihan salah");
                System.out.print("Masukkan pilihan
algoritma: ");
                try{
```

```
                    choice =
Integer.parseInt(bufferedReader.readLine().trim());
                }catch(Exception e){
                    System.out.println("\nInput harus
berupa angka");
                }
            }
            return choice;
        }catch (Exception e) {
            System.out.println("Terjadi kesalahan");
            System.out.println(e.getMessage());
            return 0;
        }

    }
```

Method readAlgo digunakan untuk mendapatkan pilihan algoritma dari masukan pengguna.

3.4 Class DictReader

Kelas DictReader berisi method-method yang berhubungan dengan pembacaan file dictionary dan pemrosesannya.

```
public class DictReader {
    public static Map<String,Boolean> englishDictionary;
    public static Map<Integer, List<String>>
findLengthWord(){
        Map<Integer, List<String>> mp= new HashMap<>();
        File file = new File("./src/dict/dict2.txt");
        try(BufferedReader br = new BufferedReader(new
FileReader(file))){
            String st;
            while ((st = br.readLine()) != null){
                List<String> exist =
mp.putIfAbsent(st.length(), new
ArrayList<>(Arrays.asList(st)));
```

```
                if(exist != null){
                    exist.add(st);
                }
            }
        } catch(IOException e){
            System.out.println(e.getMessage());
        }
        return mp;
    }
    public static void divideDict(){
        Map<Integer, List<String>> varLength =
findLengthWord();
        for (Integer i : varLength.keySet()){
            File file = new File("./src/dict/" +
Integer.toString(i) + ".txt");
            try (BufferedWriter bw = new BufferedWriter(new
FileWriter(file))){
                for(String word : varLength.get(i)){

if(word.equals(varLength.get(i).get(varLength.get(i).size()
-1))){
                        // System.out.println(word + " " +
varLength.get(i).get(varLength.get(i).size()-1));
                        bw.write(word);
                    }else{
                        bw.write(word + '\n');
                    }
                }
            } catch(IOException e){
                System.out.println(e.getMessage());
            }
        }
```

```
    }
    public static Map<String,Boolean> fileToMap(String
filepath){
        Map<String,Boolean> dict = new HashMap<>();
        File file = new File(filepath);
        try(BufferedReader br = new BufferedReader(new
FileReader(file))){
            String st;
            while ((st = br.readLine()) != null){
                dict.put(st, true);
            }
        } catch(IOException e){
            System.out.println(e.getMessage());
            dict = null;
        }
        englishDictionary = dict;
        return dict;
    }
}
```

- Method fileLengthWord mendapatkan mapping jumlah karakter dengan list kata dengan jumlah kata tersebut.
- Method divideDict digunakan untuk membagi file dictionary menjadi beberapa file berdasarkan jumlah karakternya sehingga dapat kamus yang perlu dibaca jumlahnya lebih sedikit.
- Method fileToMap digunakan untuk mendapatkan Map yang berisi semua kata dalam file.

3.5 Class UCS

```
public class UCS {
    public static final int UCS_HN = 0;
    public static List<Node> UCSsolve(String start, String
end){
        Node.resetNode();
        Map<String, Boolean> visited = new HashMap<>();

        // panjang tidak sama
```

```
        if(start.length() != end.length()){
            return null;
        }

        // end word tidak ada di dictionary
        if(!DictReader.englishDictionary.containsKey(end)){
            return null;
        }

        // Buat prioQueue
        PriorityQueue<Node> pq = new PriorityQueue<>(new
NodeComparator());
        pq.add(new Node(start,0, UCS_HN, null));

        while(!pq.isEmpty()){
            Node currNode = pq.poll();
            List<String> possibleString;
            Node.nodeVisited++;
            if(currNode.getString().equals(end)){
                return currNode.pathToRoot();
            }
            possibleString =
currNode.getExpandNode(DictReader.englishDictionary);
            visited.put(currNode.getString(), true);
            for(String s : possibleString){
                if(!visited.containsKey(s)){
                    Node newNode = new Node(s, 1, UCS_HN,
currNode);
                    pq.add(newNode);
                }
            }
        }
        return null;
    }
```

```
}
```

Method UCSsolve digunakan untuk mendapatkan solusi menggunakan algoritma UCS. Method ini akan mengembalikan list yang berisi *path* dari *root* hingga Node tujuan.

3.6 Class GBFS

```
public class GBFS {
    public static final int GBFS_GN = 0;
    public static List<Node> GBFSsolve(String start, String
end){
        Node.resetNode();
        Map<String, Boolean> visited = new HashMap<>();

        // panjang tidak sama
        if(start.length() != end.length()){
            return null;
        }

        // end word tidak ada di dictionary
        if(!DictReader.englishDictionary.containsKey(end)){
            return null;
        }

        // Buat prioQueue
        PriorityQueue<Node> pq = new PriorityQueue<>(new
NodeComparator());
        pq.add(new Node(start,GBFS_GN,
UtilityFunc.getDifference(start, end), null));

        while(!pq.isEmpty()){
            Node currNode = pq.poll();
            List<String> possibleString;
            Node.nodeVisited++;
            if(currNode.getString().equals(end)){
                return currNode.pathToRoot();
```

```
            }
            possibleString =
currNode.getExpandNode(DictReader.englishDictionary);
            visited.put(currNode.getString(), true);
            for(String s : possibleString){
                if(!visited.containsKey(s)){
                    Node newNode = new Node(s, GBFS_GN,
UtilityFunc.getDifference(s, end), currNode);
                    pq.add(newNode);
                }
            }
         }
        return null;
    }
}
```

Method GBFSsolve digunakan untuk mendapatkan solusi menggunakan algoritma Greedy Best First Search. Method ini akan mengembalikan list yang berisi *path* dari *root* hingga Node tujuan.

3.7 Class A_Star

```
public class A_Star {
    public static List<Node> A_Starsolve(String start,
String end){
        Node.resetNode();
        Map<String, Boolean> visited = new HashMap<>();

        // panjang tidak sama
        if(start.length() != end.length()){
            return null;
        }

        // end word tidak ada di dictionary
        if(!DictReader.englishDictionary.containsKey(end)){
            return null;
        }
```

```
        // Buat prioQueue
        PriorityQueue<Node> pq = new PriorityQueue<>(new
NodeComparator());
        pq.add(new Node(start,0, 0, null));

        while(!pq.isEmpty()){
            Node currNode = pq.poll();
            List<String> possibleString;
            Node.nodeVisited++;
            if(currNode.getString().equals(end)){
                return currNode.pathToRoot();
            }
            possibleString =
currNode.getExpandNode(DictReader.englishDictionary);
            visited.put(currNode.getString(), true);
            for(String s : possibleString){
                if(!visited.containsKey(s)){
                    Node newNode = new Node(s, 1,
UtilityFunc.getDifference(s, end), currNode);
                    pq.add(newNode);
                }
            }
        }
        return null;
    }
}
```

Method A_Starsolve digunakan untuk mendapatkan solusi menggunakan algoritma A*. Method ini akan mengembalikan list yang berisi *path* dari *root* hingga Node tujuan.

3.8 Class Main

Kelas Main berisi main method sebagai *entry point* pada program.

```java
public static void main(String[] args) {
    long startTime = 0;
    long endTime = 0;
    long before = 0;
    long after = 0;
    List<String> l = UtilityFunc.readInputStartEnd();
    while (l == null){
        l = UtilityFunc.readInputStartEnd();
    }
    String startword = l.get(0);
    String endword = l.get(1);
    int wordLength = startword.length();
    DictReader.fileToMap( filepath: "./src/dict/" + wordLength + ".txt");
//    DictReader.fileToMap("./src/dict/dict3.txt");

    while(DictReader.englishDictionary == null){
        l = UtilityFunc.readInputStartEnd();
        while (l == null){
            l = UtilityFunc.readInputStartEnd();
        }
        startword = l.get(0);
        endword = l.get(1);
        wordLength = startword.length();
        DictReader.fileToMap( filepath: "./src/dict/" + wordLength + ".txt");
    }

    int choice = UtilityFunc.readAlgo();
```

Pada bagian ini program akan meminta input dan melakukan validasi dari pengguna berupa *start word, end word*, dan algoritma yang diinginkan.

```java
DictReader.fileToMap( filepath: "./src/dict/" + wordLength + ".txt");
DictReader.fileToMap("./src/dict/dict3.txt");
```

Pada bagian ini program akan membaca file *txt* yang berisi kata-kata dalam bahasa Inggris. File membutuhkan nama dengan format “<panjang kata>.txt” yang berada di dalam folder src/dict. Jika tidak ditemukan file dengan panjang dari kata yang dimasukkan, program akan meminta ulang input masukan kata.

```
switch (choice) {
    case 1 -> {
        before = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
        startTime = System.nanoTime();
        sol = UCS.UCSsolve(startword, endword);
        endTime = System.nanoTime();
        after = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
    }
    case 2 -> {
        before = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
        startTime = System.nanoTime();
        sol = GBFS.GBFSsolve(startword, endword);
        endTime = System.nanoTime();
        after = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
    }
    case 3 -> {
        before = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
        startTime = System.nanoTime();
        sol = A_Star.A_Starsolve(startword, endword);
        endTime = System.nanoTime();
        after = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
    }
    default -> System.out.println("Something gone wrong");
}
```

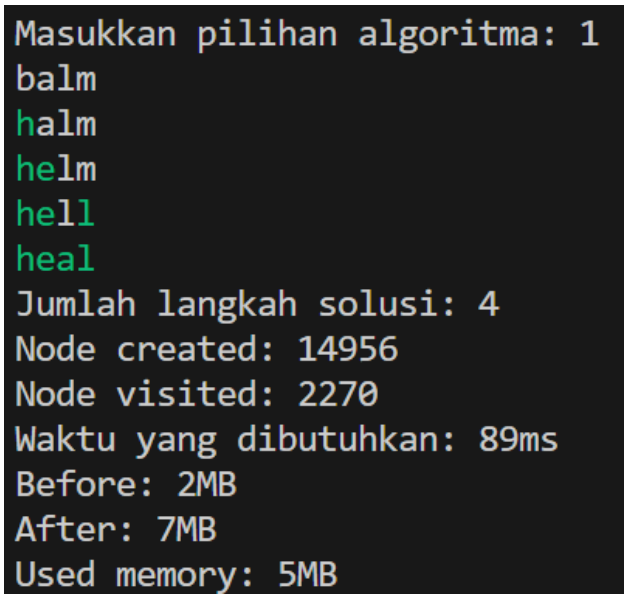
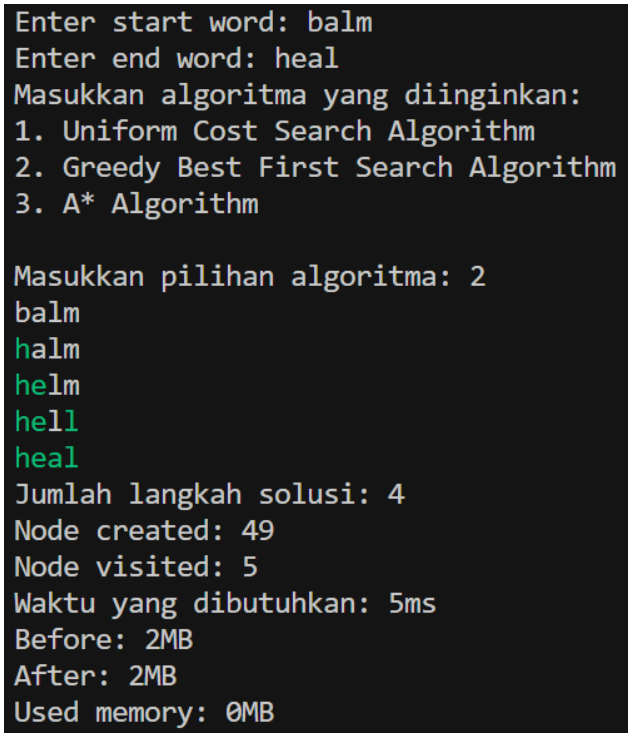
Pada bagian ini berisi *switch case* berdasarkan pemilihan algoritma. Setelah ada algoritma yang terpilih, program akan memulai perhitungan waktu eksekusi dan jumlah memory yang dibutuhkan.

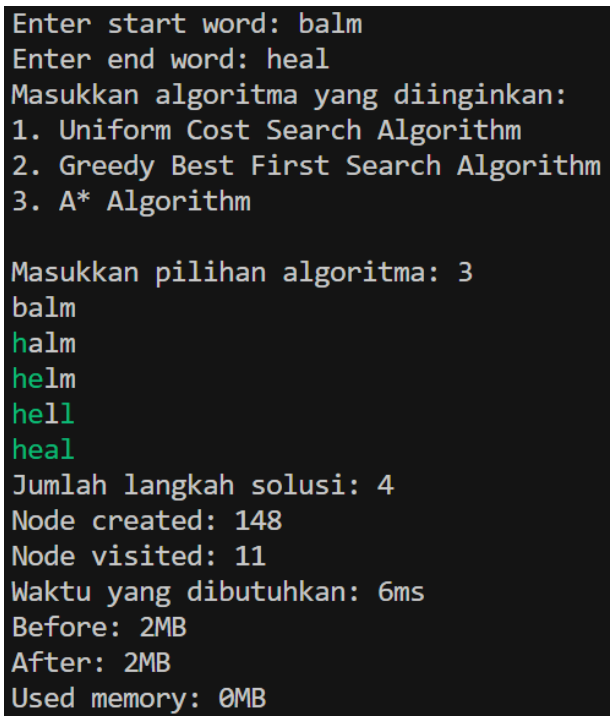
```
if(sol == null){
    System.out.println(x:"No solution found");
}else{
    UtilityFunc.printList(sol);
    System.out.println("Jumlah langkah solusi: " + (sol.size()-1));
}
System.out.println("Node created: " + Node.nodeCreated);
System.out.println("Node expanded: " + Node.nodeExpanded);
long duration = (endTime - startTime);
System.out.println("Waktu yang dibutuhkan: " + duration/1000000 + "ms");
System.out.println("Before: " + before/(1024*1024) +  "MB");
System.out.println("After: " + after/(1024*1024) +  "MB");
long used = after/(1024*1024) - before/(1024*1024);
System.out.println("Used memory: " + used + "MB") ;
```

Pada bagian ini, program akan mengeluarkan hasil solusi serta informasi waktu eksekusi, jumlah memory yang dibutuhkan, *Node* yang diciptakan dan diekspan.

Bab IV
Hasil Percobaan

4.1 Balm ke Heal

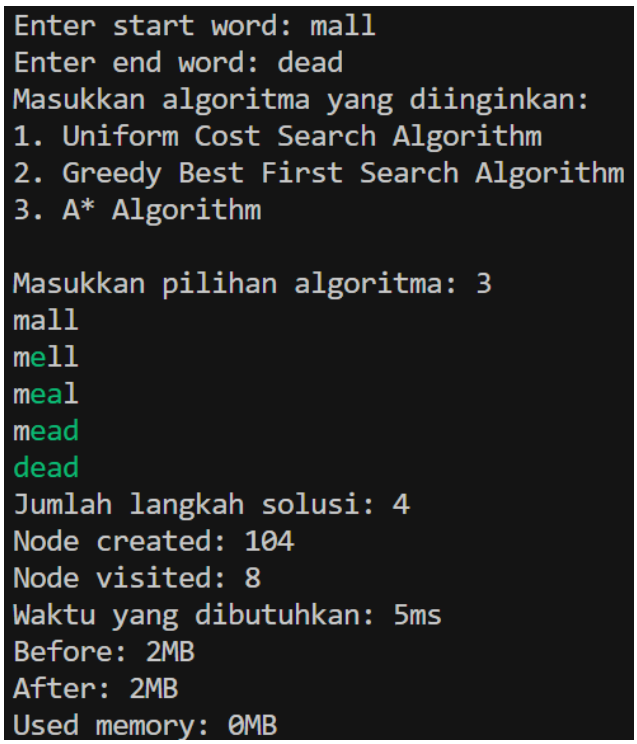
Algorithm	Picture
UCS	
GBFS	

A*	

4.2 Mall ke Dead

Algorithm	Picture

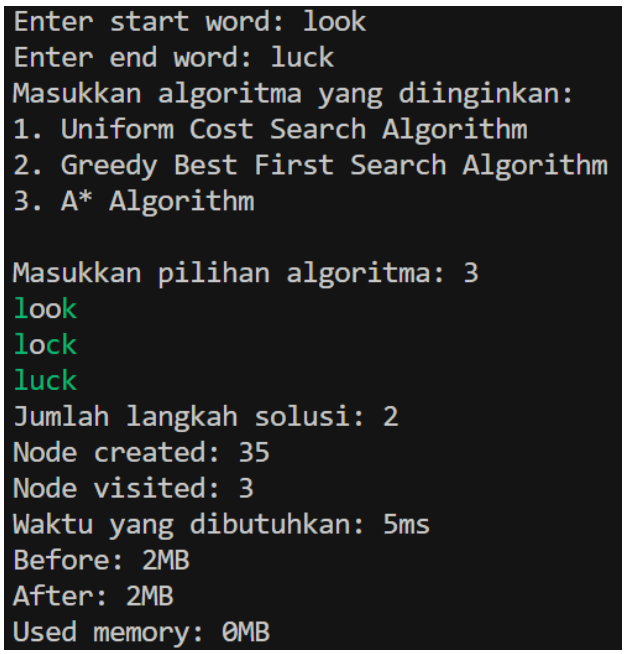
UCS	Enter start word: mall Enter end word: dead Masukkan algoritma yang diinginkan: 1. Uniform Cost Search Algorithm 2. Greedy Best First Search Algorithm 3. A* Algorithm Masukkan pilihan algoritma: 1 mall mell dell deal dead Jumlah langkah solusi: 4 Node created: 21312 Node visited: 3936 Waktu yang dibutuhkan: 101ms Before: 2MB After: 25MB Used memory: 23MB
GBFS	Enter start word: mall Enter end word: dead Masukkan algoritma yang diinginkan: 1. Uniform Cost Search Algorithm 2. Greedy Best First Search Algorithm 3. A* Algorithm Masukkan pilihan algoritma: 2 mall mell dell deal dead Jumlah langkah solusi: 4 Node created: 77 Node visited: 5 Waktu yang dibutuhkan: 6ms Before: 2MB After: 2MB Used memory: 0MB

A*	

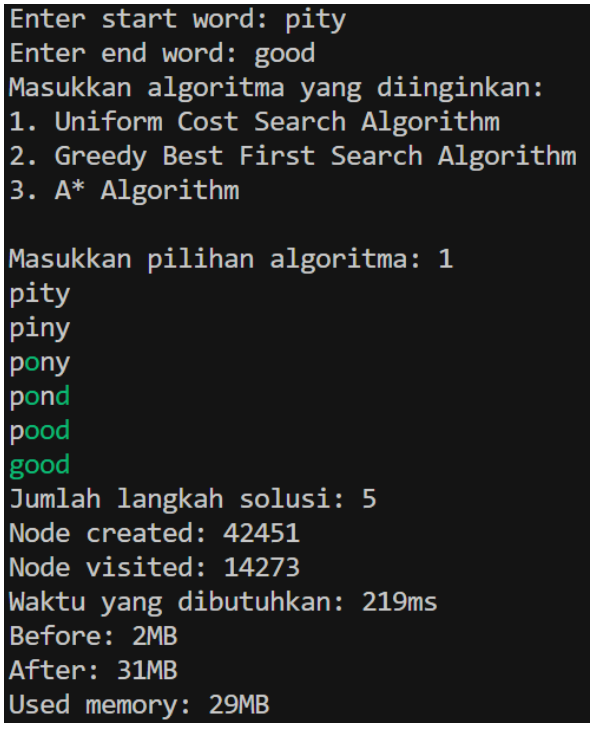
4.3 Look ke Luck

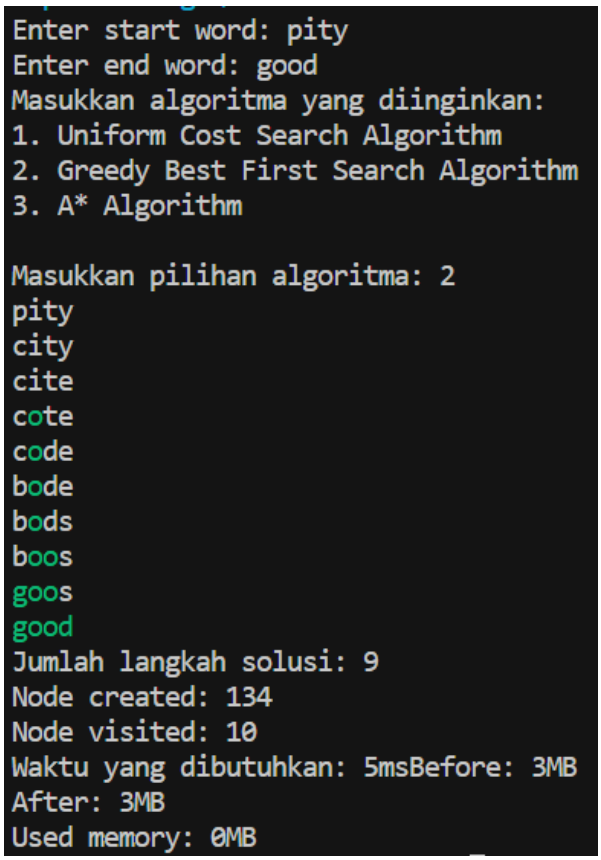
Algorithm	Picture

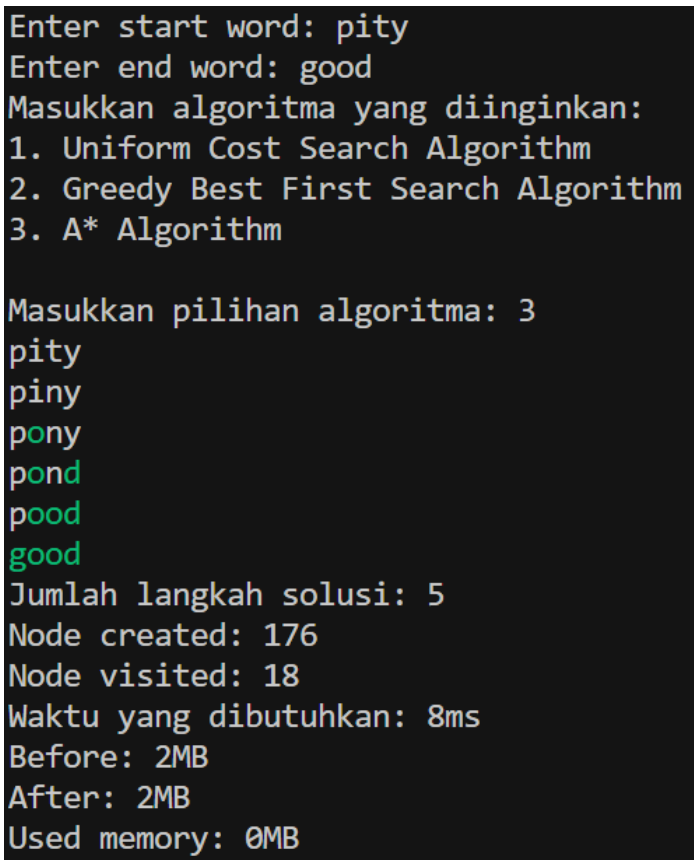
UCS	Enter start word: look Enter end word: luck Masukkan algoritma yang diinginkan: 1. Uniform Cost Search Algorithm 2. Greedy Best First Search Algorithm 3. A* Algorithm Masukkan pilihan algoritma: 1 look lock luck Jumlah langkah solusi: 2 Node created: 1116 Node visited: 152 Waktu yang dibutuhkan: 23ms Before: 2MB After: 4MB Used memory: 2MB
GBFS	Enter start word: look Enter end word: luck Masukkan algoritma yang diinginkan: 1. Uniform Cost Search Algorithm 2. Greedy Best First Search Algorithm 3. A* Algorithm Masukkan pilihan algoritma: 2 look lock luck Jumlah langkah solusi: 2 Node created: 35 Node visited: 3 Waktu yang dibutuhkan: 5ms Before: 2MB After: 2MB Used memory: 0MB

A*	

4.4 Pity ke Good

Algorithm	Picture
UCS	

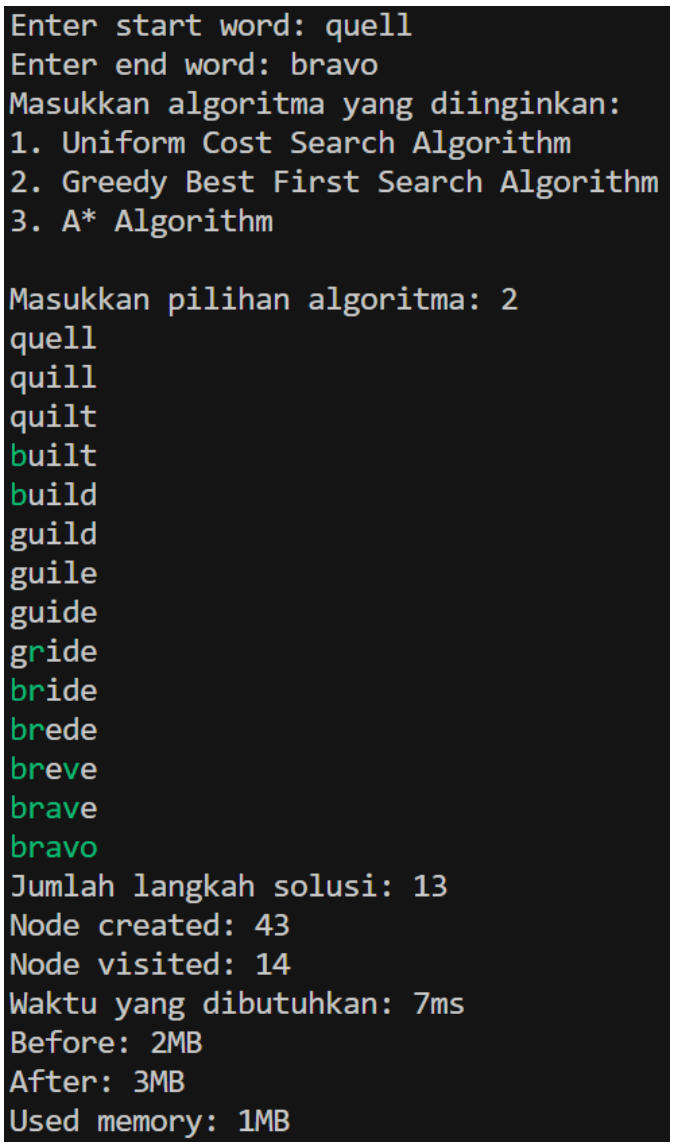
GBFS	 Enter start word: pity Enter end word: good Masukkan algoritma yang diinginkan: 1. Uniform Cost Search Algorithm 2. Greedy Best First Search Algorithm 3. A* Algorithm Masukkan pilihan algoritma: 2 pity city cite cote code bode bods boos goos good Jumlah langkah solusi: 9 Node created: 134 Node visited: 10 Waktu yang dibutuhkan: 5msBefore: 3MB After: 3MB Used memory: 0MB

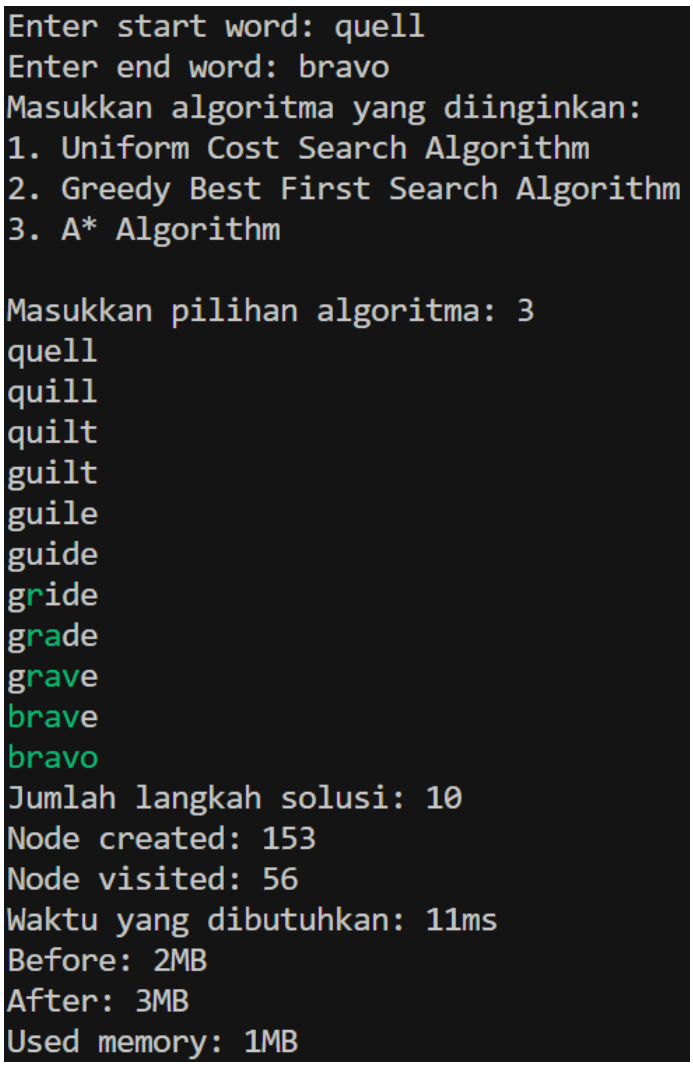
A*	

4.5 Quell ke Bravo

Algorithm	Picture

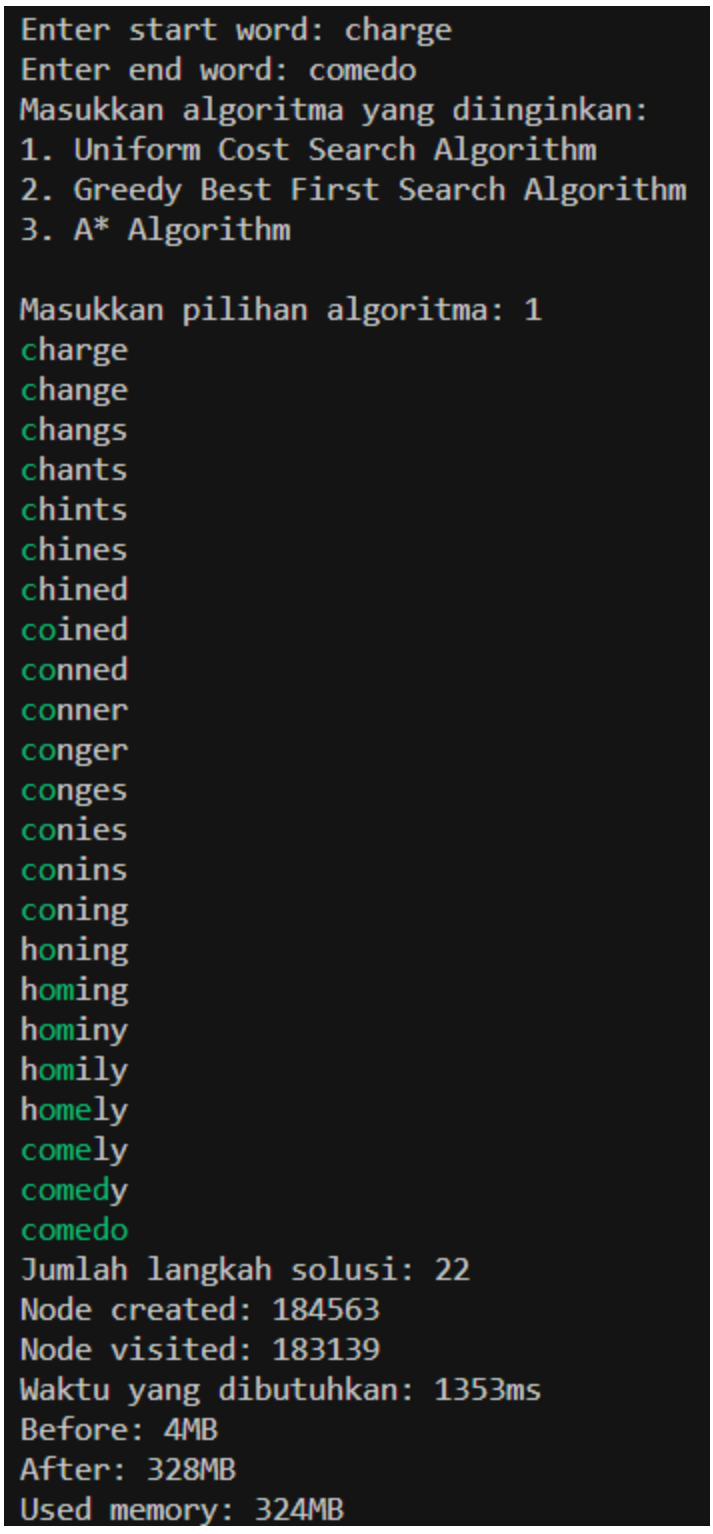
UCS	Enter start word: quell Enter end word: bravo Masukkan algoritma yang diinginkan: 1. Uniform Cost Search Algorithm 2. Greedy Best First Search Algorithm 3. A* Algorithm Masukkan pilihan algoritma: 1 quell quill quilt guilt guile guide gride grade grave brave bravo Jumlah langkah solusi: 10 Node created: 16785 Node visited: 4610 Waktu yang dibutuhkan: 129ms Before: 2MB After: 44MB Used memory: 42MB

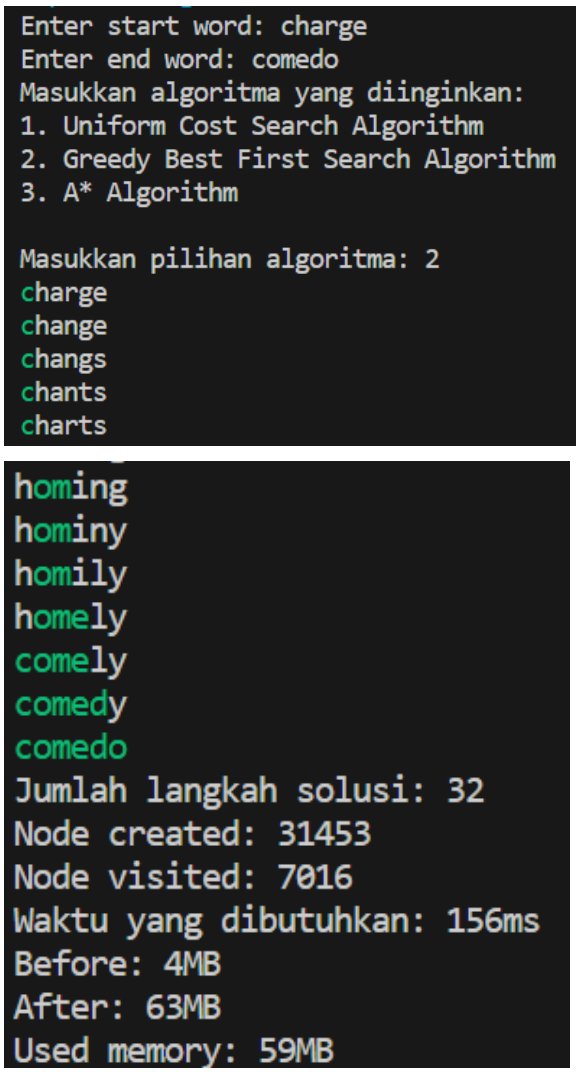
GBFS	 Enter start word: quell Enter end word: bravo Masukkan algoritma yang diinginkan: 1. Uniform Cost Search Algorithm 2. Greedy Best First Search Algorithm 3. A* Algorithm Masukkan pilihan algoritma: 2 quell quill quilt built build guild guile guide gride bride brede breve brave bravo Jumlah langkah solusi: 13 Node created: 43 Node visited: 14 Waktu yang dibutuhkan: 7ms Before: 2MB After: 3MB Used memory: 1MB

A*	

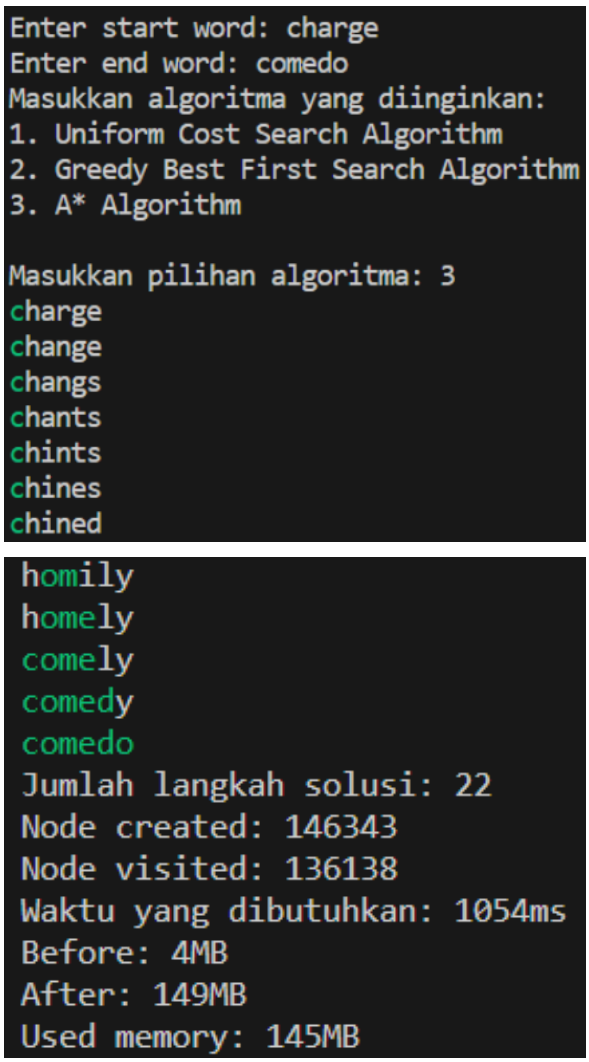
4.6 Charge ke Comedo

Algorithm	Picture

UCS	 Enter start word: charge Enter end word: comedo Masukkan algoritma yang diinginkan: 1. Uniform Cost Search Algorithm 2. Greedy Best First Search Algorithm 3. A* Algorithm Masukkan pilihan algoritma: 1 charge change changs chants chints chines chined coined conned conner conger conges conies conins coning honing homing hominy homily homely comely comedy comedo Jumlah langkah solusi: 22 Node created: 184563 Node visited: 183139 Waktu yang dibutuhkan: 1353ms Before: 4MB After: 328MB Used memory: 324MB

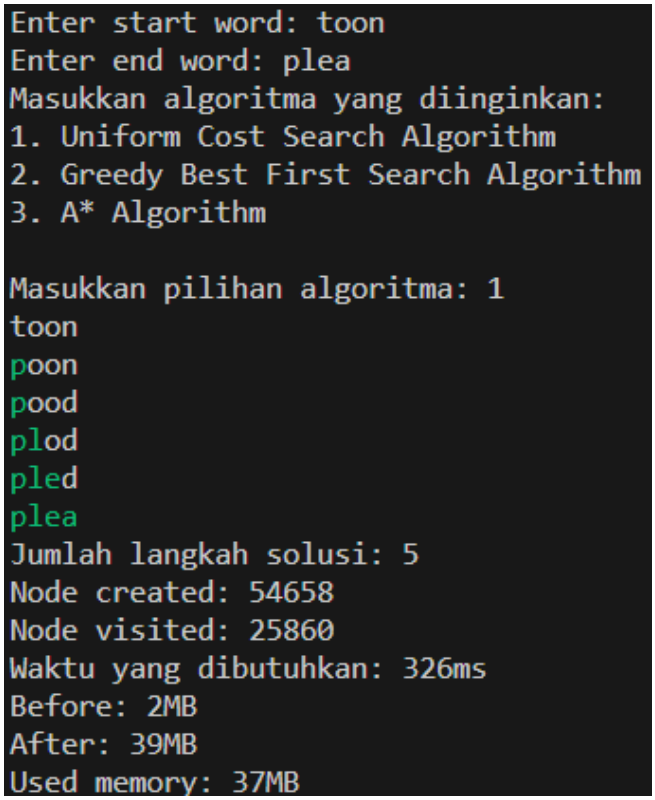
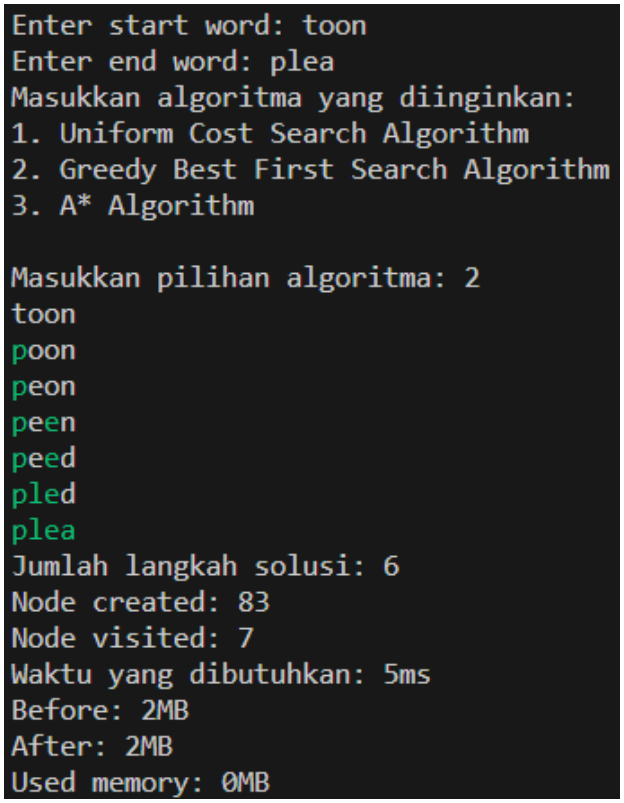
GBFS	

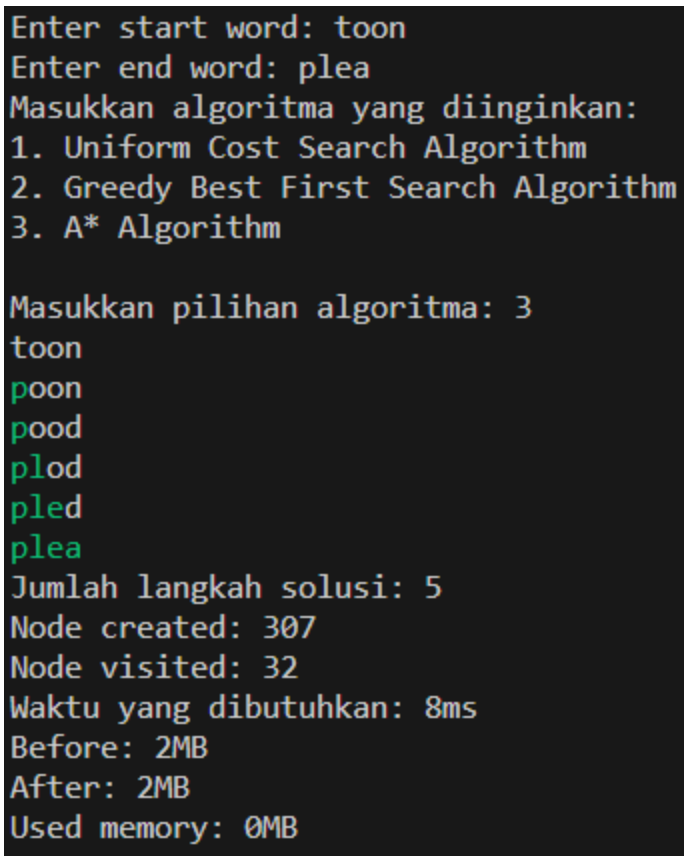
Enter start word: charge
Enter end word: comedo
Masukkan algoritma yang diinginkan:
1. Uniform Cost Search Algorithm
2. Greedy Best First Search Algorithm
3. A* Algorithm
Masukkan pilihan algoritma: 2
charge
change
changs
chants
charts
homing
hominy
homily
homely
comely
comedy
comedo
Jumlah langkah solusi: 32
Node created: 31453
Node visited: 7016
Waktu yang dibutuhkan: 156ms
Before: 4MB
After: 63MB
Used memory: 59MB

A*	

4.7 Toon ke Plea

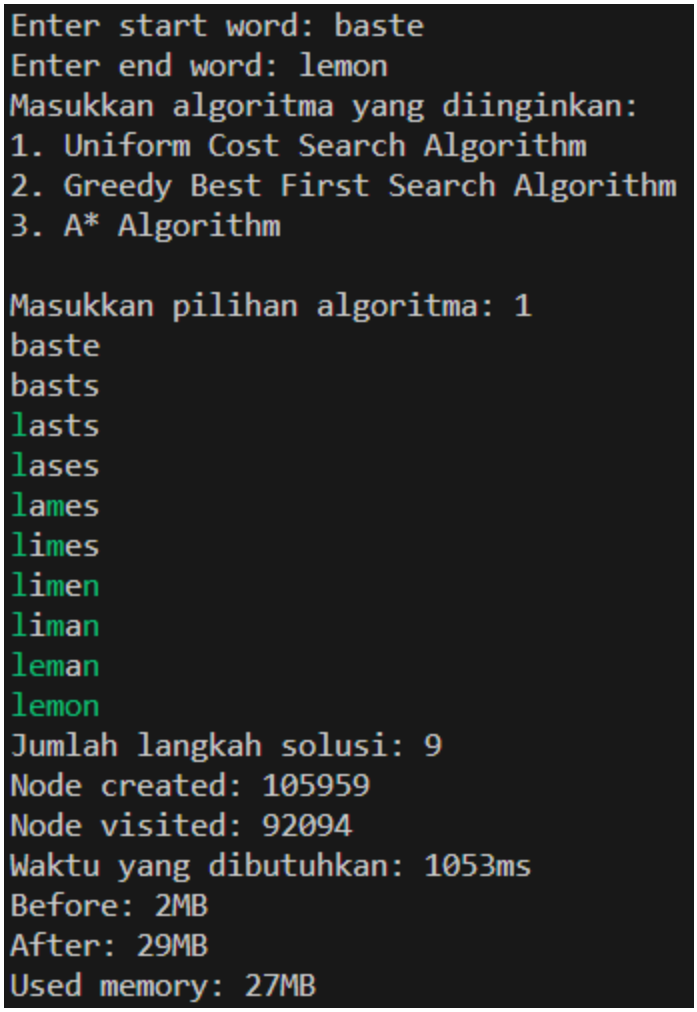
Algorithm	Picture

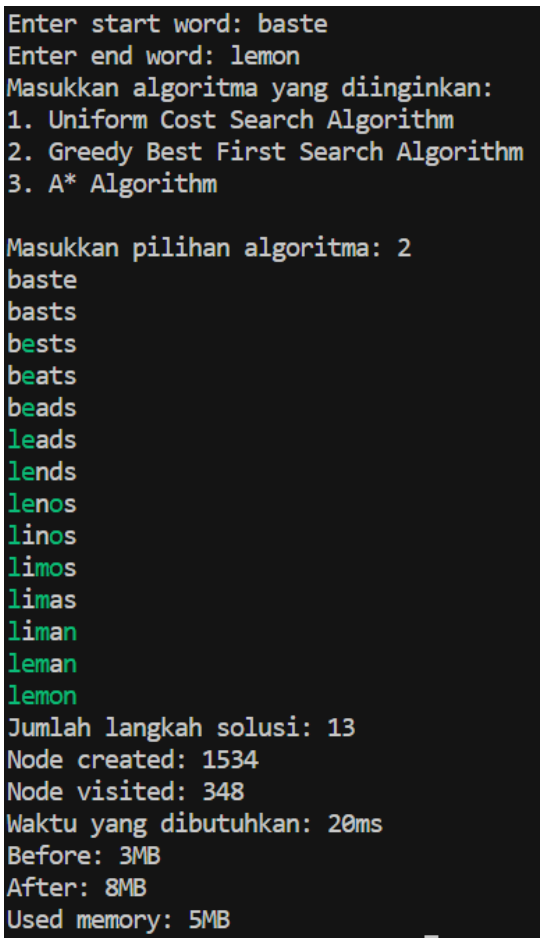
UCS	 Enter start word: toon Enter end word: plea Masukkan algoritma yang diinginkan: 1. Uniform Cost Search Algorithm 2. Greedy Best First Search Algorithm 3. A* Algorithm Masukkan pilihan algoritma: 1 toon poon pood plod pled plea Jumlah langkah solusi: 5 Node created: 54658 Node visited: 25860 Waktu yang dibutuhkan: 326ms Before: 2MB After: 39MB Used memory: 37MB
GBFS	 Enter start word: toon Enter end word: plea Masukkan algoritma yang diinginkan: 1. Uniform Cost Search Algorithm 2. Greedy Best First Search Algorithm 3. A* Algorithm Masukkan pilihan algoritma: 2 toon poon peon peen peed pled plea Jumlah langkah solusi: 6 Node created: 83 Node visited: 7 Waktu yang dibutuhkan: 5ms Before: 2MB After: 2MB Used memory: 0MB

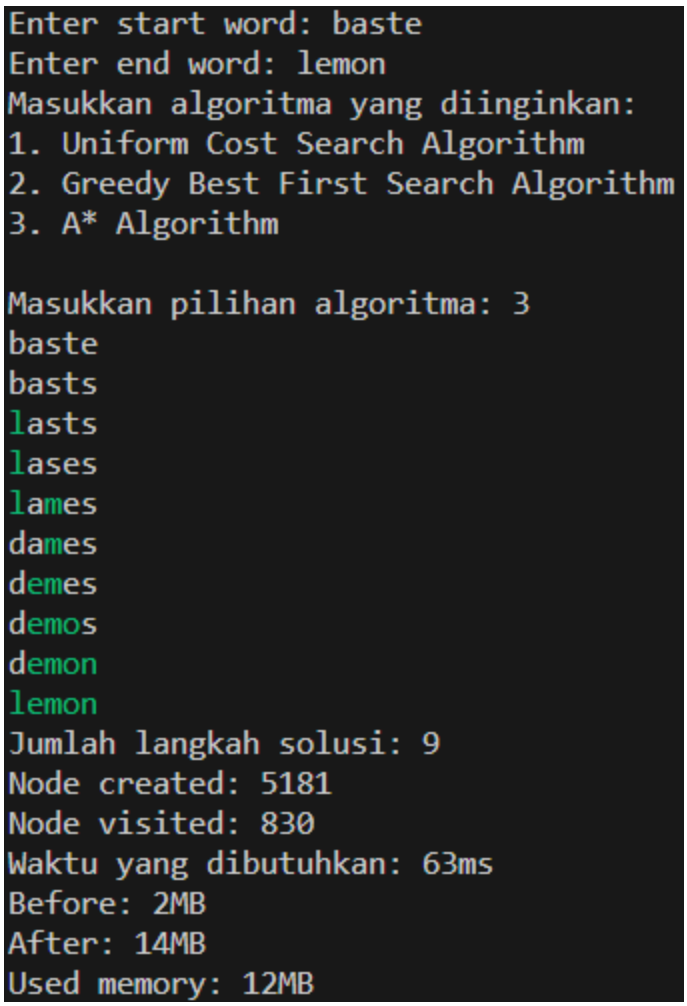
A*	

4.8 Baste ke Lemon

Algorithm	Picture

UCS	 Enter start word: baste Enter end word: lemon Masukkan algoritma yang diinginkan: 1. Uniform Cost Search Algorithm 2. Greedy Best First Search Algorithm 3. A* Algorithm Masukkan pilihan algoritma: 1 baste basts lasts lases lames limes limen liman leman lemon Jumlah langkah solusi: 9 Node created: 105959 Node visited: 92094 Waktu yang dibutuhkan: 1053ms Before: 2MB After: 29MB Used memory: 27MB
GBFS	

	 Enter start word: baste Enter end word: lemon Masukkan algoritma yang diinginkan: 1. Uniform Cost Search Algorithm 2. Greedy Best First Search Algorithm 3. A* Algorithm Masukkan pilihan algoritma: 2 baste basts bests beats beads leads lends lenos linos limos limas liman leman lemon Jumlah langkah solusi: 13 Node created: 1534 Node visited: 348 Waktu yang dibutuhkan: 20ms Before: 3MB After: 8MB Used memory: 5MB

A*	 Enter start word: baste Enter end word: lemon Masukkan algoritma yang diinginkan: 1. Uniform Cost Search Algorithm 2. Greedy Best First Search Algorithm 3. A* Algorithm Masukkan pilihan algoritma: 3 baste basts lasts lases lames dames demes demos demon lemon Jumlah langkah solusi: 9 Node created: 5181 Node visited: 830 Waktu yang dibutuhkan: 63ms Before: 2MB After: 14MB Used memory: 12MB

Bab V

Analisis Perbandingan Solusi UCS, Greedy Best First Search, dan A*

```
homily
homely
comely
comedy
comedo
Jumlah langkah solusi: 22
Node created: 184563
Node visited: 183139
Waktu yang dibutuhkan: 1353ms
Before: 4MB
After: 328MB
Used memory: 324MB
```

Gambar 5.1. Algoritma UCS

```
homing
hominy
homily
homely
comely
comedy
comedo
Jumlah langkah solusi: 32
Node created: 31453
Node visited: 7016
Waktu yang dibutuhkan: 156ms
Before: 4MB
After: 63MB
Used memory: 59MB
```

Gambar 5.2 Algoritma GBFS

```
comely
comedy
comedo
Jumlah langkah solusi: 22
Node created: 146343
Node visited: 136138
Waktu yang dibutuhkan: 1054ms
Before: 4MB
After: 149MB
Used memory: 145MB
```

Gambar 5.3 Algoritma A*

Secara teoritis, algoritma UCS dan A* dengan heuristik yang *admissible* akan selalu mendapatkan solusi yang optimal, sedangkan algoritma Greedy Best First Search tidak menjamin solusi yang optimal. Hal ini dapat terlihat pada hasil percobaan 4.6 dari Charge ke Comedo. Algoritma UCS dan A* mendapatkan solusi dengan 22 langkah (Gambar 5.1 dan Gambar 5.3), sedangkan algoritma Greedy Best First Search mendapatkan solusi dengan 32 langkah (Gambar 5.2).

```
Enter start word: pity
Enter end word: good
Masukkan algoritma yang diinginkan:
1. Uniform Cost Search Algorithm
2. Greedy Best First Search Algorithm
3. A* Algorithm

Masukkan pilihan algoritma: 1
pity
piny
pony
pond
pood
good
Jumlah langkah solusi: 5
Node created: 42451
Node visited: 14273
Waktu yang dibutuhkan: 219ms
Before: 2MB
After: 31MB
Used memory: 29MB
```

Gambar 5.4. Algoritma UCS

```
Enter start word: pity
Enter end word: good
Masukkan algoritma yang diinginkan:
1. Uniform Cost Search Algorithm
2. Greedy Best First Search Algorithm
3. A* Algorithm

Masukkan pilihan algoritma: 2
pity
city
cite
cote
code
bode
bods
boos
goos
good
Jumlah langkah solusi: 9
Node created: 134
Node visited: 10
Waktu yang dibutuhkan: 5msBefore: 3MB
After: 3MB
Used memory: 0MB
```

Gambar 5.5 Algoritma GBFS

```
Enter start word: pity
Enter end word: good
Masukkan algoritma yang diinginkan:
1. Uniform Cost Search Algorithm
2. Greedy Best First Search Algorithm
3. A* Algorithm

Masukkan pilihan algoritma: 3
pity
piny
pony
pond
pood
good
Jumlah langkah solusi: 5
Node created: 176
Node visited: 18
Waktu yang dibutuhkan: 8ms
Before: 2MB
After: 2MB
Used memory: 0MB
```

Gambar 5.6 Algoritma A*

Secara teoritis, algoritma A* lebih efisien dibandingkan dengan algoritma UCS. Karena algoritma A* menggunakan f(n) = g(n) + h(n) untuk mengetahui estimasi total *cost* menuju simpul n hingga *goal*. Sehingga rute dengan estimasi total *cost* yang mahal tidak akan diekspan, sedangkan algoritma UCS hanya membandingkan *cost* yang dibutuhkan dari simpul awal hingga simpul n. Dengan begitu, algoritma A* akan mengekspan lebih sedikit simpul dibanding algoritma UCS.

Hal ini terbukti pada hasil percobaan 4.4 dari Pity ke Good. Walaupun keduanya menghasilkan solusi yang optimal, namun algoritma A* membutuhkan waktu dan memory yang lebih sedikit dibandingkan dengan algoritma UCS. Terlihat juga perbandingan Node yang diciptakan dan di-expand/dikunjungi pada algoritma UCS jauh lebih besar dibandingkan pada algoritma A* (Gambar 5.4 dan 5.6). Oleh karena itu, terbukti bahwa algoritma A* lebih efisien dibandingkan dengan algoritma UCS.

```
Masukkan pilihan algoritma: 1
balm
halm
helm
hell
heal
Jumlah langkah solusi: 4
Node created: 14956
Node visited: 2270
Waktu yang dibutuhkan: 89ms
Before: 2MB
After: 7MB
Used memory: 5MB
```

Gambar 5.7. Algoritma UCS

```
Enter start word: balm
Enter end word: heal
Masukkan algoritma yang diinginkan:
1. Uniform Cost Search Algorithm
2. Greedy Best First Search Algorithm
3. A* Algorithm

Masukkan pilihan algoritma: 2
balm
halm
helm
hell
heal
Jumlah langkah solusi: 4
Node created: 49
Node visited: 5
Waktu yang dibutuhkan: 5ms
Before: 2MB
After: 2MB
Used memory: 0MB
```

Gambar 5.8 Algoritma GBFS

```
Enter start word: balm
Enter end word: heal
Masukkan algoritma yang diinginkan:
1. Uniform Cost Search Algorithm
2. Greedy Best First Search Algorithm
3. A* Algorithm

Masukkan pilihan algoritma: 3
balm
halm
helm
hell
heal
Jumlah langkah solusi: 4
Node created: 148
Node visited: 11
Waktu yang dibutuhkan: 6ms
Before: 2MB
After: 2MB
Used memory: 0MB
```

Gambar 5.9 Algoritma A*

```
Enter start word: toon
Enter end word: plea
Masukkan algoritma yang diinginkan:
1. Uniform Cost Search Algorithm
2. Greedy Best First Search Algorithm
3. A* Algorithm

Masukkan pilihan algoritma: 1
toon
poon
pood
plod
pled
plea
Jumlah langkah solusi: 5
Node created: 54658
Node visited: 25860
Waktu yang dibutuhkan: 326ms
Before: 2MB
After: 39MB
Used memory: 37MB
```

Gambar 5.10. Algoritma UCS

```
Enter start word: toon
Enter end word: plea
Masukkan algoritma yang diinginkan:
1. Uniform Cost Search Algorithm
2. Greedy Best First Search Algorithm
3. A* Algorithm

Masukkan pilihan algoritma: 2
toon
poon
peon
peen
peed
pled
plea
Jumlah langkah solusi: 6
Node created: 83
Node visited: 7
Waktu yang dibutuhkan: 5ms
Before: 2MB
After: 2MB
Used memory: 0MB
```

```
Enter start word: toon
Enter end word: plea
Masukkan algoritma yang diinginkan:
1. Uniform Cost Search Algorithm
2. Greedy Best First Search Algorithm
3. A* Algorithm

Masukkan pilihan algoritma: 3
toon
poon
pood
plod
pled
plea
Jumlah langkah solusi: 5
Node created: 307
Node visited: 32
Waktu yang dibutuhkan: 8ms
Before: 2MB
After: 2MB
Used memory: 0MB
```

Gambar 5.12 Algoritma A*

Gambar 5.11 Algoritma GBFS

```
Enter start word: baste
Enter end word: lemon
Masukkan algoritma yang diinginkan:
1. Uniform Cost Search Algorithm
2. Greedy Best First Search Algorithm
3. A* Algorithm

Masukkan pilihan algoritma: 1
baste
basts
lasts
lases
lames
limes
limen
liman
leman
lemon
Jumlah langkah solusi: 9
Node created: 105959
Node visited: 92094
Waktu yang dibutuhkan: 1053ms
Before: 2MB
After: 29MB
Used memory: 27MB
```

Gambar 5.13. Algoritma UCS

```
Enter start word: baste
Enter end word: lemon
Masukkan algoritma yang diinginkan:
1. Uniform Cost Search Algorithm
2. Greedy Best First Search Algorithm
3. A* Algorithm

Masukkan pilihan algoritma: 2
baste
basts
bests
beats
beads
leads
lends
lenos
linos
limos
limas
liman
leman
lemon
Jumlah langkah solusi: 13
Node created: 1534
Node visited: 348
Waktu yang dibutuhkan: 20ms
Before: 3MB
After: 8MB
Used memory: 5MB
```

Gambar 5.14 Algoritma GBFS

```
Enter start word: baste
Enter end word: lemon
Masukkan algoritma yang diinginkan:
1. Uniform Cost Search Algorithm
2. Greedy Best First Search Algorithm
3. A* Algorithm

Masukkan pilihan algoritma: 3
baste
basts
lasts
lases
lames
dames
demes
demos
demon
lemon
Jumlah langkah solusi: 9
Node created: 5181
Node visited: 830
Waktu yang dibutuhkan: 63ms
Before: 2MB
After: 14MB
Used memory: 12MB
```

Gambar 5.15 Algoritma A*

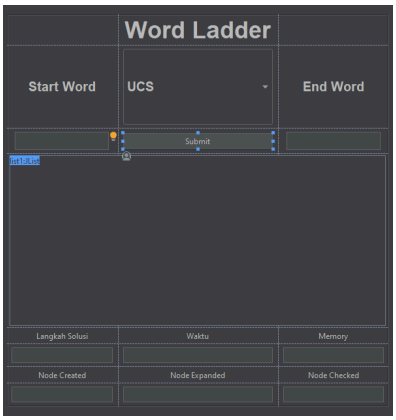
Jika dilihat dari perbandingan waktu eksekusi dan memory yang dibutuhkan, algoritma Greedy Best First Search merupakan algoritma yang paling cepat dan membutuhkan memory paling sedikit. Algoritma A* berada di urutan kedua dan Algoritma UCS berada di urutan terakhir. Jika dilihat dari optimalitasnya, solusi yang diberikan algoritma GBFS tidak optimal (langkah solusinya lebih panjang) seperti yang terlihat pada **Gambar 5.2, 5.5, 5.11, 5.14**. Sedangkan algoritma UCS dan A* selalu memberikan solusi yang optimal.

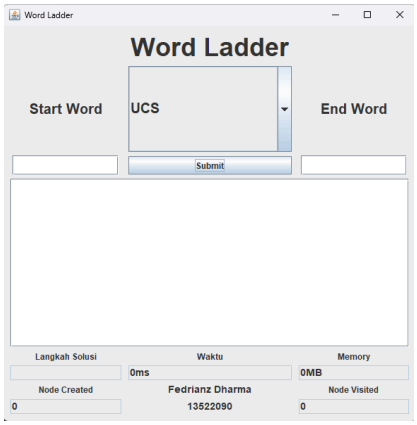
Oleh karena itu, jika dilihat berdasarkan optimalitas, waktu eksekusi, dan memory yang dibutuhkan, algoritma A* merupakan algoritma yang paling baik untuk menyelesaikan persoalan *Word Ladder*. Algoritma A* akan memberikan solusi yang optimal dengan waktu eksekusi yang cepat dan memory yang kecil. Bahkan pada beberapa kasus, perbedaan waktu eksekusi dan memory yang dibutuhkan antara algoritma A* dan GBFS tidak jauh berbeda, namun algoritma A* tetap memberikan solusi yang optimal.

Kesimpulannya adalah algoritma Greedy Best First Search baik digunakan jika menginginkan solusi dengan waktu eksekusi yang cepat, meskipun hasilnya kurang optimal. Algoritma A* baik digunakan jika menginginkan hasil solusi yang optimal dengan waktu yang cepat. Algoritma UCS kurang baik digunakan dalam kasus *Word*

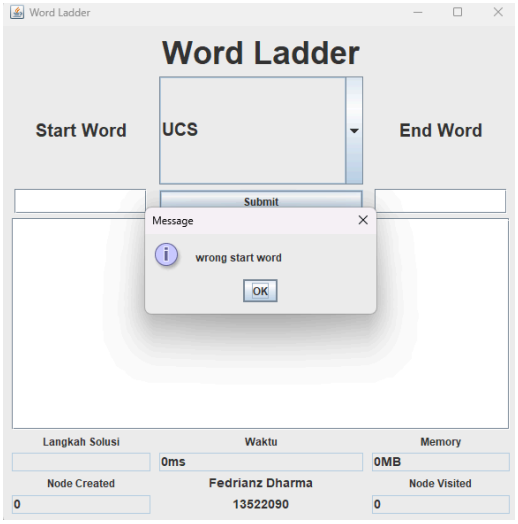
Ladder karena waktu eksekusinya lama, meskipun menghasilkan solusi yang optimal, lebih baik menggunakan algoritma A*.

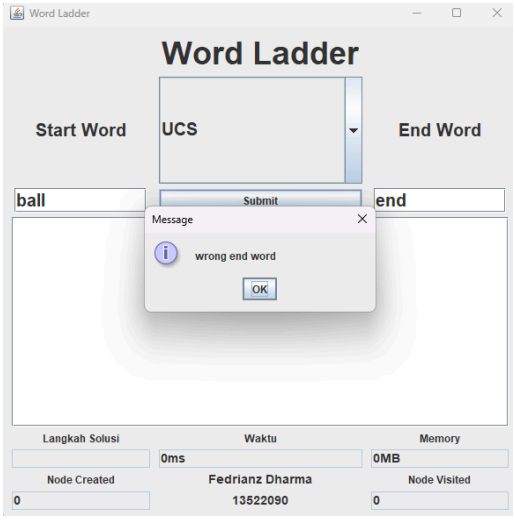
Bab VI
Penjelasan GUI



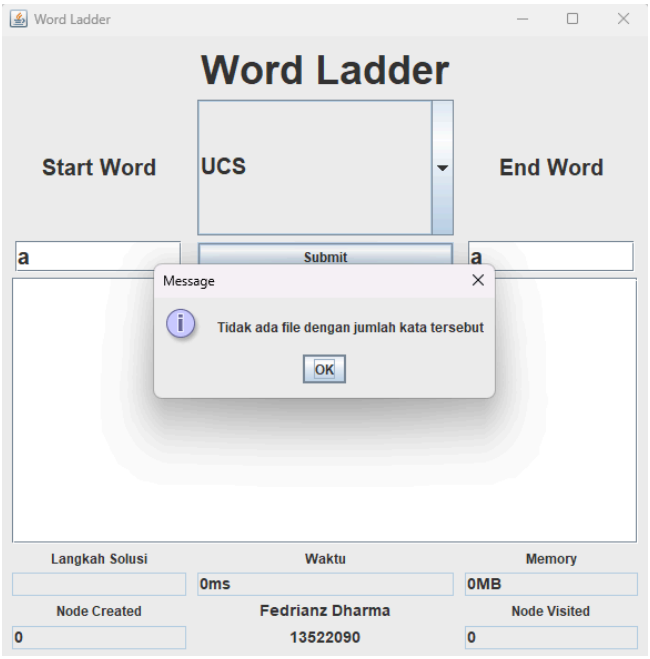


GUI dibangun dengan menggunakan bantuan Intelij Idea GUI form. Pengguna dapat memasukkan *start word* yang diinginkan pada *text box* sebelah kiri dan *end word* pada sebelah kanan. Setelah itu, pengguna bisa memilih algoritma yang diinginkan pada *dropdown menu* ditengah dan meng-klik tombol *submit*. Hasil rute rangkaian kata yang dibutuhkan untuk mencapai *end word* dari *start word* muncul di bagian tengah. Jumlah langkah solusi, waktu eksekusi, memory yang dibutuhkan, jumlah *Node* yang diciptakan dan jumlah *Node* yang diekspand akan ada di bagian bawah.

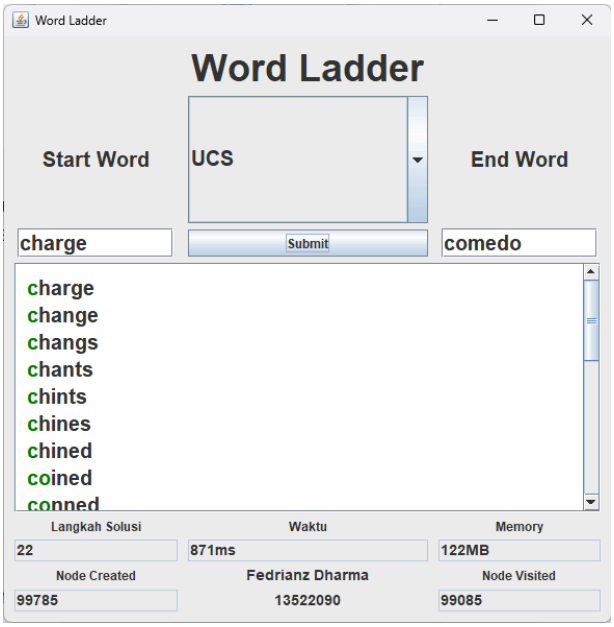




Terdapat *message*, jika memasukkan *start word* kosong atau jumlah karakter pada *start word* dan *end word* berbeda.



Selain itu juga, terdapat pemberitahuan jika jumlah karakter pada kata yang dimasukkan tidak ada dalam kamus yang digunakan.



Jika semuanya sudah berhasil, ini adalah contoh hasil keluaran dari Charge ke Comedo.

Kesimpulan

Dengan menggunakan algoritma UCS, Greedy Best First Search, dan A*, kita dapat menemukan rute rangkaian kata dari *start word* ke *end word* sesuai dengan aturan permainan *Word Ladder*. Algoritma Greedy Best First Search memiliki waktu eksekusi paling cepat dan memory yang dibutuhkannya paling sedikit, namun tidak menjamin solusi yang diberikan optimal. Algoritma UCS dan A* (dengan heuristik *admissible*) dijamin selalu memberikan solusi yang optimal. Namun, algoritma A* memiliki waktu eksekusi yang lebih cepat dan membutuhkan memory yang lebih kecil dibandingkan algoritma UCS. Waktu eksekusi algoritma A* juga tidak terlalu berbeda dengan algoritma GBFS, namun memberikan solusi optimal. Oleh karena itu, secara *overall* algoritma A* merupakan algoritma yang paling bagus untuk menyelesaikan persoalan *Word Ladder*.

Lampiran

Github repository:

https://github.com/FedrianzD/Tucil3_13522090

Referensi:

- **https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf**
- **https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf**

Tabel:

No	Poin	Yes	No
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	✔️	