

**Laporan Tugas Kecil 3 IF2211 Strategi Algoritma
Semester II Tahun Akademik 2023/2024**

**Penciptaan Solusi dari Permainan *World Ladder*
Menggunakan Algoritma A*, Greedy Best First Search,
dan Uniform Cost Search**



Disusun oleh Zaki Yudhistira Candra / 13522031

K-01

**Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung**

2024

A. Analisis Kasus dan Implementasi

Pada permainan *word ladder*, pemain akan diperlukan untuk mencari *path* kata yang dapat menghubungkan kata awal dan kata akhir. Akan digunakan algoritma A*, *Greedy Best First Search*, dan *Uniform Cost Search* untuk menemukan solusi jawaban yang paling optimal.

Akan dimuat sebuah kamus yang akan menjadi acuan untuk validasi kebenaran dari kata.

Untuk pemetaan masalah, setiap kata pada kamus akan menjadi sebuah *node*, dan jarak antar *node* berupa jumlah perbedaan huruf antar kata. Misalkan kata 'ball' dan 'fall' memiliki jarak antar *node* sebesar 1 karena terdapat satu huruf yaitu 'f' dan 'b' yang berbeda.

Kata awal atau *starting word* akan menjadi *root node* dan kata akhir atau *end word* akan menjadi *target node*. *Root node* akan dibangkitkan terlebih dahulu, pembangkitan di sini dalam artian mencari seluruh *adjacent nodes* yang belum pernah dikunjungi.

Berikut akan dibahas rancangan implementasi dan hipotesis teoritis untuk masing-masing algoritma.

I. Algoritma *Uniform Cost Search*

Uniform Cost Search atau UCS merupakan algoritma yang mencari rute terpendek berdasarkan *node* terdekat, *node* terdekat dihitung dari jarak ke *node* yang sudah dieksplorasi sebelumnya. Algoritma akan memilih *node* terdekat untuk dibangkitkan. Berikut adalah langkah implementasi algoritma UCS.

1. Mulai dari *root node* terlebih dahulu
2. Bangkitkan *root node*
3. Tempatkan hasil pembangkitan *root node* pada suatu antrean prioritas dan disortir berdasarkan jarak terpendek $g(n)$ ($g(n)$ yaitu perbedaan huruf sebesar satu)
4. Setiap antrean akan diproses dari depan dan dibangkitkan, hasil pembangkitan setiap *node* akan diperlakukan seperti *root node*
5. *Node* yang sudah diproses akan menjadi '*root node*' baru, dalam artian sudah dikunjungi, dan *node* terpendek dari '*root node*' akan diproses terlebih dahulu

6. Ulangi setiap langkah hingga *target node* tercapai

Dalam konteks implementasi ini, UCS sama dengan BFS karena setiap *node* memiliki jarak yang sama antar *adjacent node* dan dari definisi, BFS adalah UCS dengan jarak *adjacent node* yang seragam.

II. Algoritma *Greedy Best First Search*

Greedy Best First Search atau disingkat *Greedy BFS* merupakan algoritma *pathfinding* yang menentukan rute berdasarkan jarak *node* ke *node* akhir. Algoritma merupakan algoritma *informed search*, yaitu ketika sebuah algoritma akan diberikan sebuah informasi terkait *end goal* yang dapat membantu pencarian rute. Berikut adalah langkah penyelesaiannya :

1. Mulai dari *root node* terlebih dahulu
2. Cari *node* lain yang bersebelahan
3. Gunakan $h(n)$ untuk menentukan *node* mana yang ingin dijelajahi selanjutnya
4. Dalam kasus ini, $h(n)$ merupakan jarak *node* ke hasil akhir, yaitu jumlah perbedaan karakternya. Yang paling sedikit akan dipilih
5. Pilih *node* dengan $h(n)$ yang paling kecil
6. Ulangi hingga ditemukan hasil, atau tidak ada *node* yang bisa dieksplorasi

Algoritma ini cepat dan hemat memori, tetapi tidak menjamin akan ditemukan hasil yang optimal, bahkan bisa jadi tidak ditemukan hasil sama sekali. Hal ini terjadi karena algoritma tidak melakukan *backtrack*.

III. Algoritma A^*

A^* atau dikenal sebagai AStar merupakan algoritma *pathfinding* yang menggunakan fungsi evaluasi gabungan antara Greedy BFS dan UCS. Hal ini dilakukan untuk menghindari pembangkitan *node* yang mahal. Fungsi evaluasi $f(n)$ merupakan jumlahan antara $h(n)$ dan $g(n)$, dinotasikan sebagai berikut $f(n) = h(n) + g(n)$. Akan dipilih $f(n)$ yang paling kecil sebagai *node* selanjutnya untuk dibangkitkan. Berikut adalah langkah eksekusi algoritma A^* :

1. Mulai dari *root node* terlebih dahulu
2. Bangkitkan *root node*

3. Masukkan hasil pembangkitan dalam suatu antrean yang diurutkan berdasarkan $f(n)$ yang dijelaskan sebelumnya
4. Proses siap *node* di antrean dari yang terdepan
5. Ulangi hingga ditemukan *target node*

Secara teoritis, A^* lebih sangkil ketimbang UCS karena *node* yang dibangkitkan tidak akan sebanyak UCS berhubung A^* tidak akan membangkitkan *node* yang ‘mahal’ atau jauh dari hasil dan jauh dari lokasi *root node*.

Untuk *admissibility* dari $h(n)$, $h(n)$ *admissible* karena jarak yang dihasilkan oleh $h(n)$ selalu sama dengan jarak *node* sebenarnya dengan *target node* atau $h^*(n)$.

B. Source Code Program

Program ini diimplementasikan menggunakan bahasa Java versi 16.0.2. Program ini terdiri dari beberapa paket dan kelas utama :

Daftar paket:

1. Algorithm

Paket yang berisi implementasi algoritma, terdiri dari kelas:

- a. AStar
- b. AStarQueue
- c. Compute
- d. GreedyBfs
- e. NodeProcessing
- f. NoPossiblePath
- g. UCS
- h. UCSQueue

2. DataStructs

Paket yang berisi struktur data yang digunakan, terdiri dari kelas:

- a. ArrayRet
- b. Path
- c. Return

3. GUI

Paket yang berisi tampilan, terdiri dari kelas:

- a. Frame
- b. ResultFrame

4. Util, terdiri dari kelas:

- a. Loader
- b. Run


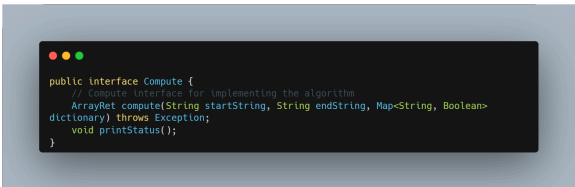
Dan terdapat kelas Main yang bukan merupakan bagian dari paket manapun

Daftar tipe data *library* yang digunakan:

1. Map<String, Boolean>, digunakan untuk menyimpan kamus
2. ArrayList<String>, digunakan untuk *array* hasil

Paket yang berisi alat yang membantu untuk menjalankan program

Berikut adalah implementasi dari kelas dan penjelasan lebih lanjut:

Implementasi	Nama Kelas - Paket - Penjelasan
 <pre>public class NodeProcessing { // Node processing library protected ArrayList<Path> getAdjacentNode(Path path, Map<String, Boolean> dictionary){ // Returns unvisited adjacent nodes ArrayList<Path> ret = new ArrayList<>(); for(Map.Entry<String, Boolean> key : dictionary.entrySet()){ // Add a visited node if(key.getKey().length() == path.path.length()){ if(!key.getValue()){ if(path.getDistanceFrom(key.getKey()) == 1){ key.setValue(true); ret.add(new Path(key.getKey(), path)); } } } } return ret; } protected ArrayList<String> reverseArrayList(ArrayList<String> alist) { // Array reverse algorithm ArrayList<String> revArrayList = new ArrayList<String>(); for (int i = alist.size() - 1; i >= 0; i--) { revArrayList.add(alist.get(i)); } return revArrayList; } protected ArrayList<String> getPath(String target, Path start) { // Return a path for result ArrayList<String> ret = new ArrayList<>(); while(true){ if(start.predecessor.path.equals(target)){ ret.add(start.path); break; } else { ret.add(start.path); start = start.predecessor; } } ret = reverseArrayList(ret); ret.add(0, target); return ret; } }</pre>	NodeProcessing - Algorithm - Kelas untuk melakukan pembangkitan <i>node</i>
 <pre>public interface Compute { // Compute interface for implementing the algorithm ArrayRet compute(String startString, String endString, Map<String, Boolean> dictionary) throws Exception; void printStatus(); } }</pre>	Compute - Algorithm - Kelas yang merupakan <i>interface</i> untuk menjalankan algoritma.

```

public class AStar extends NodeProcessing implements Compute {
    public ArrayRet compute(String startString, String endString, Map<String, Boolean>
dictionary) throws Exception {
        int count = 0;

        Path first = new Path(startString, null);

        AStarQueue queue = new AStarQueue(startString, endString);
        // Priority queue

        queue.addAll(getAdjacentNode(first, dictionary));
        // First node initiation

        while(!queue.isEmpty()){
            // Node processing algorithm
            count++;
            Path process = queue.poll();
            if(process.path.equals(endString)){
                return new ArrayRet(count, getPath(startString, process));
            } else {
                queue.addAll(getAdjacentNode(process, dictionary));
            }
        }

        throw new NoPossiblePath();
    }

    public void printStatus(){
        System.out.println("Running the A* Algorithm...");
    }
}

```

AStar - Algorithm - Kelas implementasi algoritma A*.

```

class AStarQueue extends ArrayList<Path>{
    public static String root;
    public static String goal;

    AStarQueue(String begin, String end){
        // Queue constructor
        super();
        root = begin;
        goal = end;
    }

    @Override
    public boolean add(Path e) {
        for(int i = 0 ; i < super.size() ; i++){
            if(sumCost(e, root, goal) < sumCost(super.get(i), root, goal)){
                super.add(i, e);
                return true;
            }
        }
        return super.add(e);
    }

    public Path poll(){
        Path temp = super.get(0);
        super.remove(0);
        return temp;
    }

    public int sumCost(Path path, String start, String target){
        // Minimum sum-cost path
        return path.getDistanceFrom(start) + path.getDistanceFrom(target);
    }
}

```

AStarQueue - Algorithm - Kelas untuk implementasi fungsi evaluasi A* dan antrian A*

```

public class GreedyBFS extends NodeProcessing implements Compute {
    public ArrayRet compute(String firstWord, String secondWord, Map<String, Boolean> dictionary) throws Exception {
        Path first_path = new Path(firstWord, null);
        ArrayList<Path> res = new ArrayList<>();
        ArrayRet ret = handleGreedyBFS(firstWord, first_path, secondWord, dictionary, res, 0);
        return ret;
    }

    public ArrayRet handleGreedyBFS(String start_word, Path path_to_be_processed, String goal_path, Map<String, Boolean> dictionary,
ArrayList<Path> result, int count) throws Exception{
        // Recursive Greedy BFS
        ArrayList<Path> next_nodes = getAdjacentNode(path_to_be_processed, dictionary); // Retrieve the next nodes

        if(next_nodes.isEmpty()){
            throw new NoPossiblePath(); // No path is available, throw no possible path exception
        }

        Path next_node = nextNode(goal_path, next_nodes); // Retrieve the next node to visit
        result.add(path_to_be_processed);

        count++;
        if(next_node.path.equals(goal_path)){ // If a match is found, get the result array and return
            result.add(next_node);
            ArrayList<String> res = new ArrayList<>();
            for(Path node : result){
                ret.add(node.path);
            }
            return new ArrayRet(count, ret);
        }
        return handleGreedyBFS(start_word, next_node, goal_path, dictionary, result, count); // If not, the algorithm continues
    }

    private Path nextNode(String target, ArrayList<Path> node_list){ // Function to decide the next node, or first
        Path shortest = node_list.get(0);
        for(Path node : node_list){
            if(node.getDistanceFrom(target) < shortest.getDistanceFrom(target)){ // Calculates the next nearest node to the result
                shortest = node;
            }
        }
        return shortest;
    }

    public void printStatus(){
        System.out.println("Running the Greedy Breadth First Search Algorithm...");
    }
}

```

GreedyBFS - Algorithm - Kelas untuk implementasi algoritma GreedyBFS

 <pre> public class UCS extends NodeProcessing implements Compute { public ArrayRet compute(String first_word, String second_word, Map<String, Boolean> dictionary) throws Exception { int count = 0; Path first = new Path(first_word, null); // Initializes the first path UCSQueue queue = new UCSQueue(second_word); // Initialize the priority queue based on the distance to target string queue.addAll(getAdjacentNode(first, dictionary)); // First node initiation while(!queue.isEmpty()){ // Node processing algorithm Path process = queue.poll(); count++; if(process.path.equals(second_word)){ return new ArrayRet(count, getPath(first_word, process)); } else { queue.addAll(getAdjacentNode(process, dictionary)); } } throw new NoPossiblePath(); // Throw exception if there are no possible path } public void printStatus(){ System.out.println("Running the Uniform Cost Search Algorithm..."); } } </pre>	<p>UCS - Algorithm - Kelas untuk implementasi algoritma UCS</p>
 <pre> class UCSQueue extends ArrayList<Path> { // Priority queue implementation public static String root; UCSQueue(String begin){ super(); root = begin; } @Override public boolean add(Path e) { for(int i = 0; i < super.size(); i++){ if(e.getDistanceFrom(root) < super.get(i).getDistanceFrom(root)){ super.add(i, e); return true; } } return super.add(e); } public Path poll(){ Path temp = super.get(0); super.remove(0); return temp; } } </pre>	<p>UCSQueue- Algorithm - Kelas untuk implementasi fungsi evaluasi UCS dan antrian UCS</p>
 <pre> class NoPossiblePath extends Exception { // No possible path exception NoPossiblePath(){ super("Target String unreachable"); } } </pre>	<p>NoPossiblePath - Algorithm - Kelas Exception bila tidak ada <i>path</i> yang ditemukan</p>
 <pre> public class ArrayRet { public int count; public ArrayList<String> array; public ArrayRet(int c, ArrayList<String> ar){ count = c; array = ar; } } </pre>	<p>ArrayRet - DataStructs - Kelas struktur data ketika <i>return</i> dari algoritma, berisi <i>path</i> dan jumlah <i>node</i> yang diproses</p>

 <pre> public class Path implements Comparable<Path>{ // Linked list implementation for path result public String path; public Path predecessor; public Path(String path, Path pred){ this.path = path; this.predecessor = pred; } public Integer getDistanceFrom(String root){ // Retrieve distance between string int count = 0; for(int i = 0 ; i < root.length() ; i++){ if(path.charAt(i) != root.charAt(i)){ count++; } } return count; } public int compareTo(Path other) { return path.compareTo(other.path); } } </pre>	<p>Path - DataStructs - Kelas struktur data berupa <i>linked list</i> untuk memperoleh <i>path</i> hasil</p>
 <pre> public class Return { // Return class, to be displayed to the user public ArrayList<String> path; public long exec_time; public long mem_usage; public long words_processed; public Return(ArrayRet res, long time, long usage){ path = res.array; exec_time = time; mem_usage = usage; words_processed = res.count; } public ArrayList<String> getPath(){return path;} public long getExecTime(){return exec_time;} } </pre>	<p>Return - DataStructs - Kelas struktur data yang dikirim ke GUI untuk ditampilkan</p>
<p>Tangkap layar kode terlalu besar</p>	<p>Frame - GUI - Kelas tampilan GUI utama</p>
<p>Tangkap layar kode terlalu besar</p>	<p>ResultFrame - GUI - Kelas tampilan GUI hasil</p>
 <pre> public class Loader { // Utility class for loading dictionary public static Map<String, Boolean> loadDictionary(String dict_path) throws FileNotFoundException{ Map<String, Boolean> map = new HashMap<>(); File file = new File(dict_path); Scanner scan = new Scanner(file); while(scan.hasNextLine()){ String data = scan.nextLine().toUpperCase(); map.put(data, false); } scan.close(); return map; } } </pre>	<p>Loader - Util - Kelas untuk memuat kamus</p>


```

public class Run {
    Map<String, Boolean> main_dict;
    public String first;
    public String second;

    public Run(final Map<String, Boolean> dict, String first_word, String second_word){
        int word_length = first_word.length();
        main_dict = new HashMap<String, Boolean>();
        for(Map.Entry<String, Boolean> key : dict.entrySet()){ // Add a visited node
            if(key.getKey().length() == word_length){
                main_dict.put(key.getKey(), false);
            }
        }
        first = first_word;
        second = second_word;
    }

    public Return runDFS() throws Exception{
        long start_mem = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
        long start = System.currentTimeMillis();
        GreedyIt b = new GreedyIt();
        ArrayMet res = b.compute(first, second, main_dict);
        long end_memory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
        long runtime = System.currentTimeMillis() - start;
        return new Return(res, runtime, end_memory - start_mem);
    }

    public Return runStar() throws Exception{
        long start_mem = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
        long start = System.currentTimeMillis();
        AStar b = new AStar();
        ArrayMet res = b.compute(first, second, main_dict);
        long end_memory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
        long runtime = System.currentTimeMillis() - start;
        return new Return(res, runtime, end_memory - start_mem);
    }

    public Return runUCS() throws Exception{
        long start_mem = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
        long start = System.currentTimeMillis();
        UCS b = new UCS();
        ArrayMet res = b.compute(first, second, main_dict);
        long end_memory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
        long runtime = System.currentTimeMillis() - start;
        return new Return(res, runtime, end_memory - start_mem);
    }
}

```

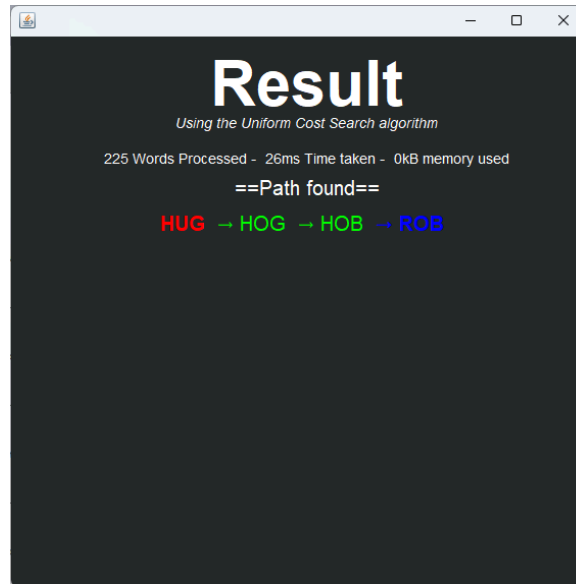
Run - Util - Kelas untuk menjalankan algoritma

C. Pengujian

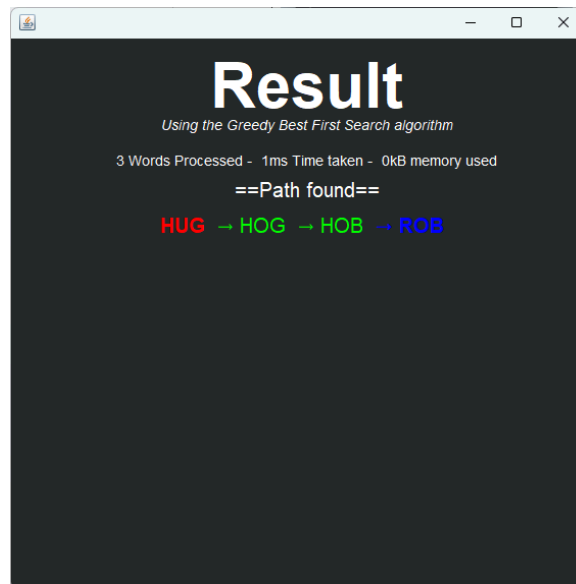
Akan dilakukan pengujian terhadap program, pengujian berupa hasil tampilan hasil dari kasus yang diberikan.

1. Kasus 1, Hug → Rob

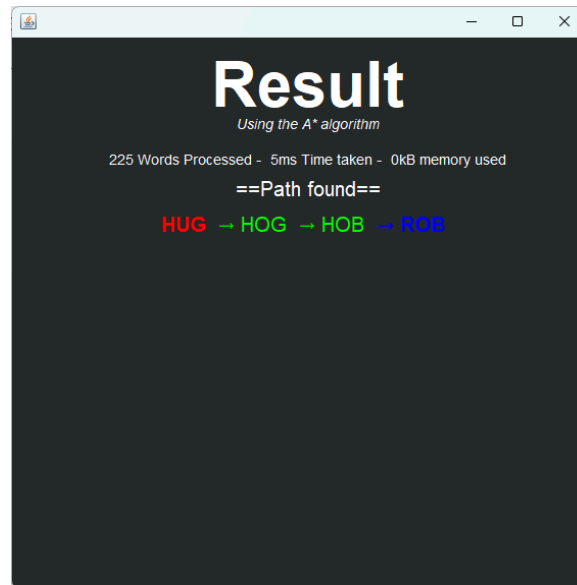
a. Menggunakan UCS



b. Menggunakan Greedy BFS

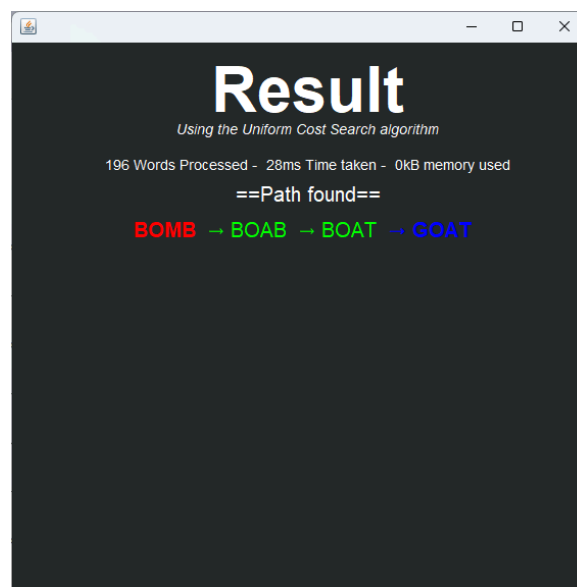


c. Menggunakan A*

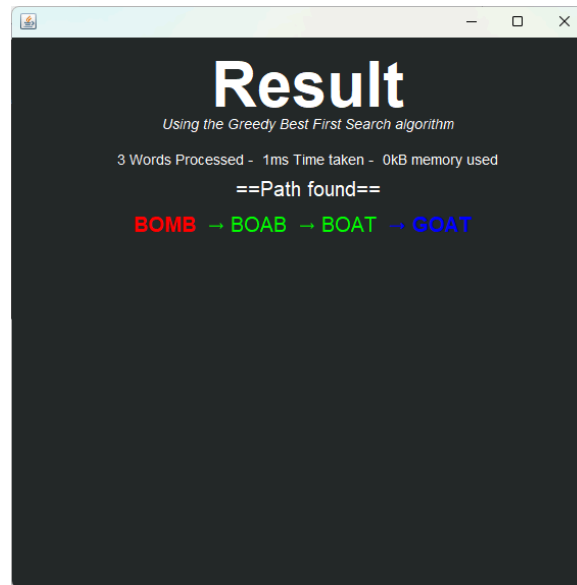


2. Kasus 2, Bomb → Goat

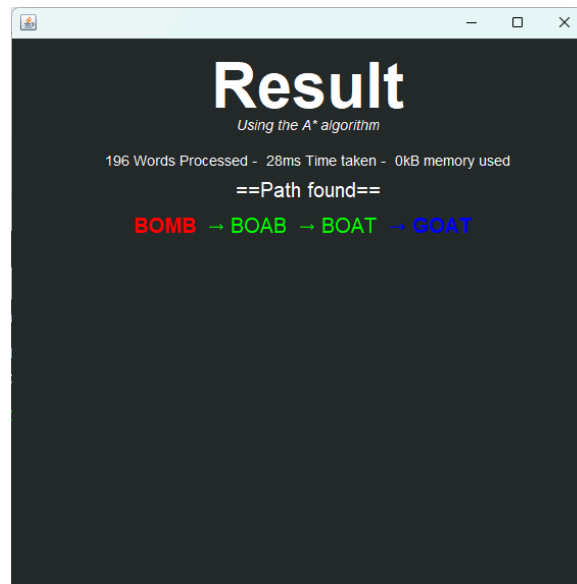
a. Menggunakan UCS



b. Menggunakan Greedy BFS

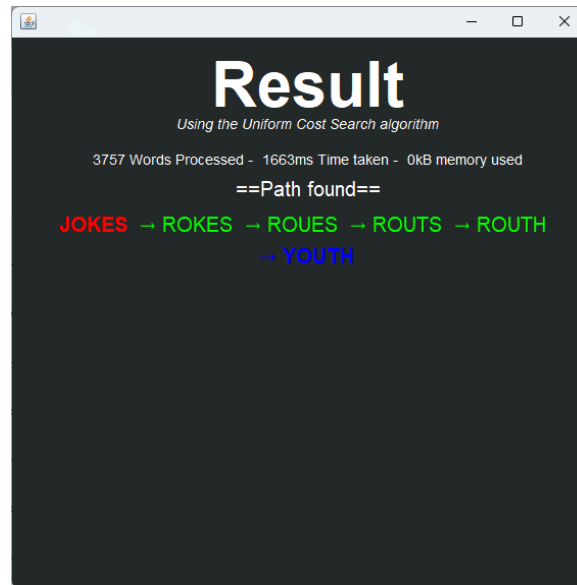


c. Menggunakan A*



3. Kasus 3, Jokes, Youth

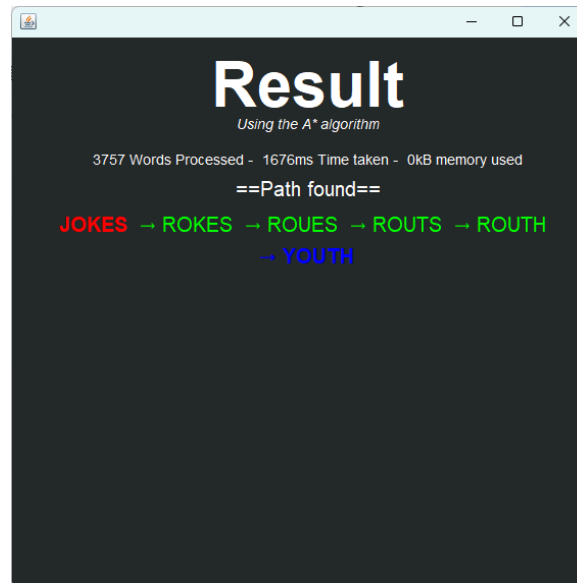
a. Menggunakan UCS



b. Menggunakan Greedy BFS

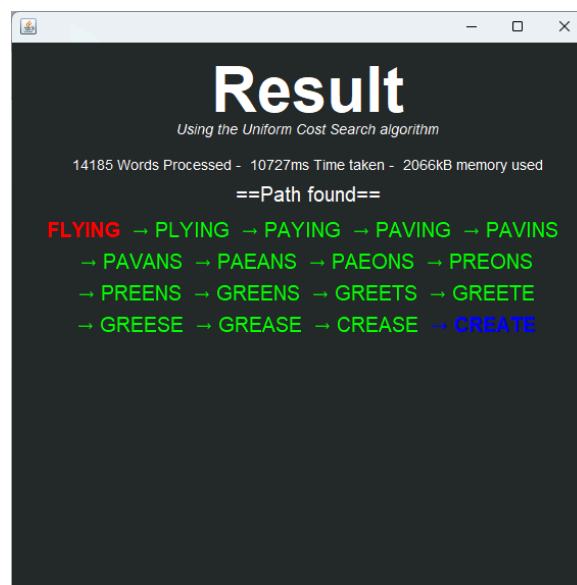


c. Menggunakan A*

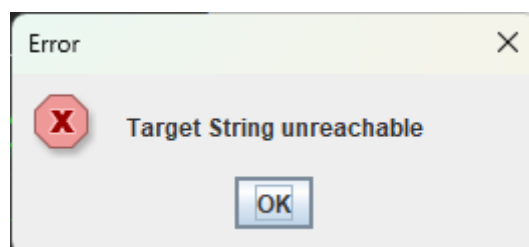


4. Kasus 4, Flying → Create

a. Menggunakan UCS

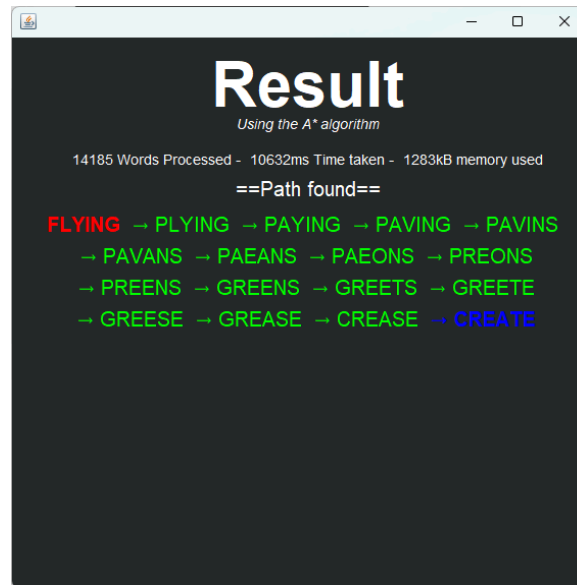


b. Menggunakan Greedy BFS



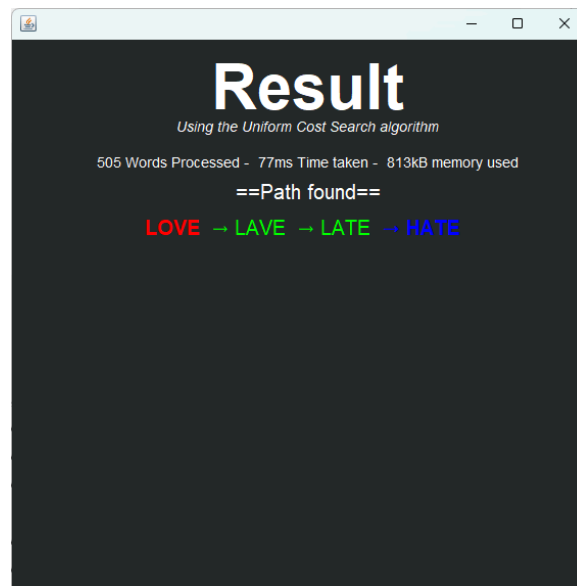
Tidak ditemukan *path* bila menggunakan Greedy BFS

c. Menggunakan A*

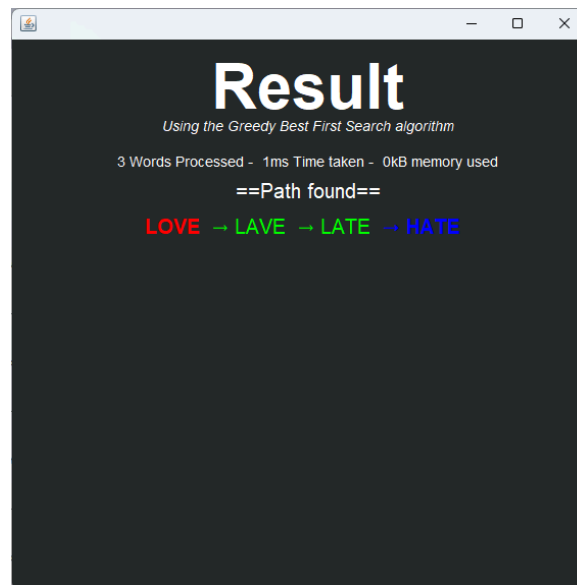


5. Kasus 5, Love → Hate

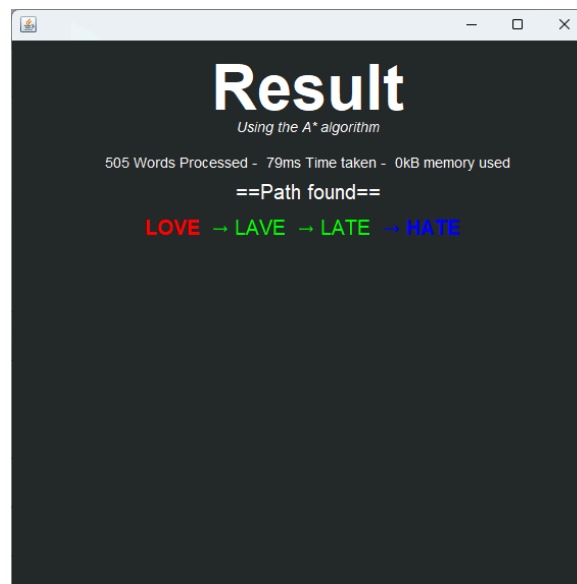
a. Menggunakan UCS



b. Menggunakan Greedy BFS

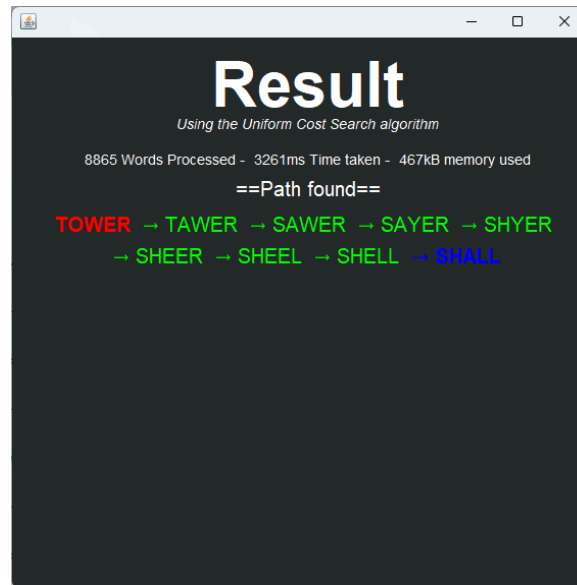


c. Menggunakan A*



6. Kasus 6, Tower, Shall

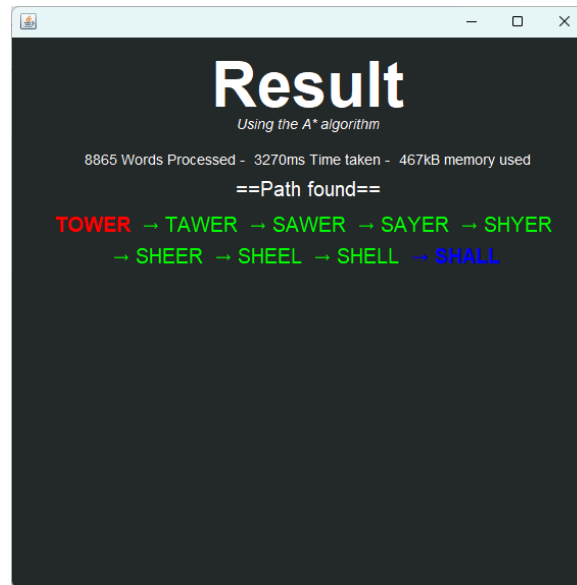
a. Menggunakan UCS



b. Menggunakan Greedy BFS



c. Menggunakan A*



D. Analisis Perbandingan Algoritma

Dari kasus - kasus pada bab sebelumnya, dilihat bahwa GreedyBFS selalu memiliki *runtime* yang lebih cepat dari ketiganya, tetapi tidak selalu memberikan solusi yang optimal, bahkan pada satu kasus tidak berhasil dalam memberikan solusi.

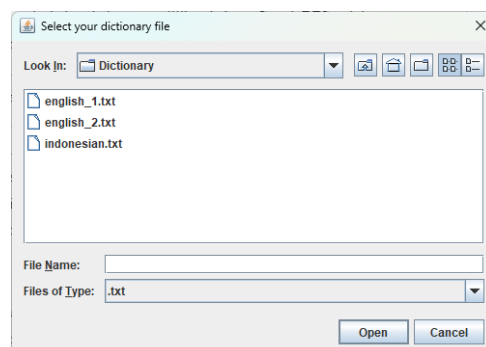
Juga dapat dilihat bahwa algoritma UCS dan A* pada konteks ini tidak begitu memiliki perbedaan yang signifikan dari segi *runtime* dan *memory usage*, mungkin pada satu kasus A* memiliki *memory usage* yang lebih rendah, tetapi hal ini juga mungkin berhubungan dengan *garbage collector* yang digunakan oleh Java, berhubung kita tidak memiliki *memory freedom* seperti pada bahasa C dan C++.

Dapat disimpulkan bahwa hasil tidak begitu jauh dari hipotesis yang diberikan pada awal pembahasan.

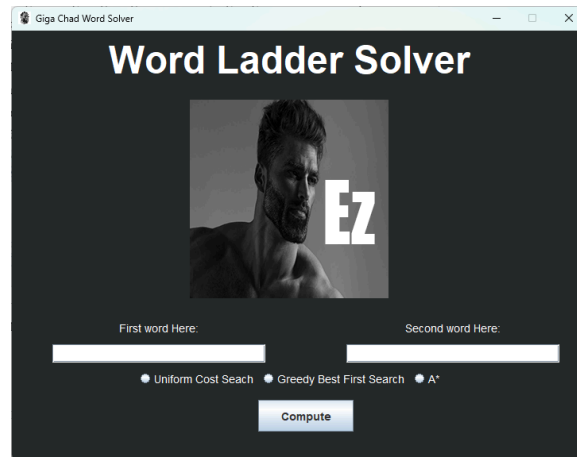
E. Penjelasan Bonus

Dikerjakan bonus yaitu GUI yang memiliki fitur sebagai berikut

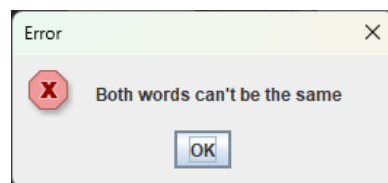
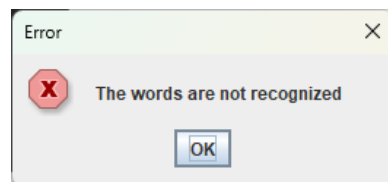
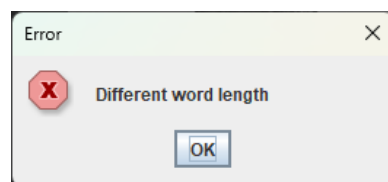
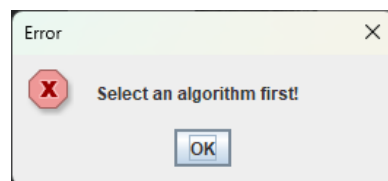
1. Memilih *file dictionary* yang akan digunakan



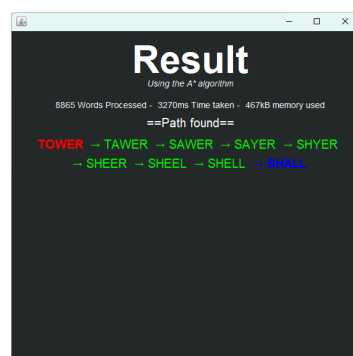
2. Tampilan utama



3. Memasukkan kata yang ingin dipakai beserta *error handling*



4. Menampilkan hasil



F. Lampiran

Keterangan tambahan:

1. Dilakukan optimalisasi pada pemrosesan kamus, ketika pengguna memasukkan huruf, akan dibuat kamus baru yang hanya terdiri dari kata dengan jumlah huruf yang sama
2. GreedyBFS dapat mencari *path* ke titik akhir, tetapi tidak pada semua kasus

Tautan *repository* : https://github.com/ZakiYudhistira/Tucil3_13522031

Poin	Ya	Tidak
Program berhasil dijalankan.	V	
Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma UCS	V	
Solusi yang diberikan pada algoritma UCS optimal	V	
Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma Greedy Best First Search	V	V
Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma A*	V	
Solusi yang diberikan pada algoritma A* optimal	V	