LAPORAN TUGAS KECIL III

IF2211 - STRATEGI ALGORITMA

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

Disusun oleh :

Bryan Cornelius Lauwrence 13522033

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

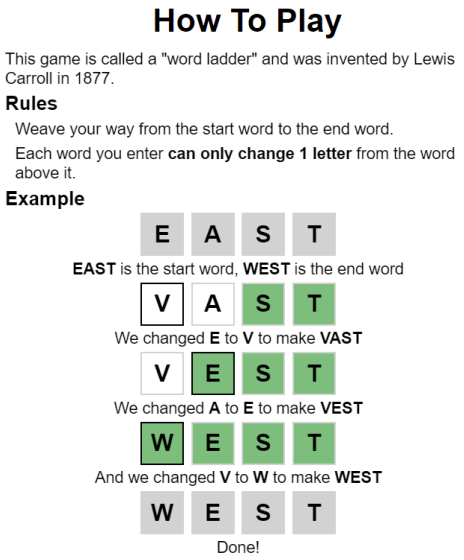
INSTITUT TEKNOLOGI BANDUNG

2024

DAFTAR ISI

BAB I
DESKRIPSI TUGAS

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



Gambar 1.1 Ilustrasi dan Peraturan Permainan Word Ladder
(Sumber: https://wordwormdormdork.com/)

Tugas kecil ini bertujuan untuk mengimplementasikan program untuk mencari solusi dari *word ladder* dengan algoritma *path planning*, yaitu *uniform cost search* (*UCS*), *greedy best first search* (*GBFS*) dan A*. Program akan mencari solusi optimal berdasarkan masing-masing algoritma. Program dibuat menggunakan bahasa pemrograman Java.

BAB II
LANDASAN TEORI

2.1 Algoritma *Path Planning*

Algoritma *path planning* adalah algoritma untuk menentukan rute, jika bisa rute terpendek, dari suatu simpul ke simpul lainnya pada graf tertentu. Algoritma *path planning* terdiri dari dua jenis, yaitu *uninformed search* dan *informed search*. *Uninformed search* adalah pencarian tanpa memedulikan lokasi simpul tujuan sehingga seluruh simpul akan dibangkitkan secara merata sampai ditemukan simpul tujuan. *UCS* merupakan salah satu algoritma yang termasuk *uninformed search*. *Informed search* adalah pencarian yang memedulikan lokasi simpul tujuan sehingga hanya simpul yang mengarah ke simpul tujuanlah yang akan dibangkitkan. *GBFS* dan A* termasuk *Informed search*.

Ketiga algoritma yang telah disebutkan memiliki sebuah fungsi evaluasi *f(n)* pada setiap simpul yang telah dibangkitkan. Fungsi evaluasi diibaratkan sebagai harga yang diperlukan untuk mencapai simpul tersebut. Fungsinya untuk mengoptimalkan rute yang dicari dengan hanya mengunjungi simpul-simpul dengan harga minimum. Fungsi evaluasi menggunakan perhitungan *greedy g(n)* dan perhitungan *heuristic h(n)* sesuai masalah yang akan diselesaikan. Fungsi evaluasi dapat menggunakan keduanya atau salah satu saja tergantung algoritma yang dipakai.

2.2 Algoritma *Uniform Cost Search*

Algoritma *uniform cost search* merupakan algoritma yang *f(n)*-nya hanya menggunakan fungsi *g(n)*. Algoritma ini akan membangkitkan anak-anak dari suatu simpul, kemudian menghitung biayanya, misalnya pada pencarian rute biayanya adalah jarak tempuh, dari simpul awal ke simpul yang sedang dievaluasi. Algoritma ini kemudian akan memasukkan simpul-simpul baru ke dalam suatu antrean yang mengutamakan simpul dengan biaya terkecil. Simpul ini kemungkinan besar dapat menemukan rute terpendek, tetapi alokasi memori yang digunakan sangat besar karena rute akan dibangkitkan berdasarkan biayanya saat ini sehingga rute yang menjauh dari simpul tujuan pun akan terus dibangkitkan.

2.3 Algoritma *Greedy Best First Search*

Algoritma *greedy best first search* merupakan algoritma yang *f(n)*-nya hanya menggunakan fungsi *h(n)*. Algoritma ini juga menghitung biaya dari anak-anak yang

dibangkitkan oleh suatu simpul. Misalnya pada pencarian jalur, biayanya adalah jarak simpul saat ini ke simpul tujuan pada garis lurus. Algoritma juga menggunakan sebuah antrean yang diurutkan berdasarkan biaya terkecil. Algoritma ini akan memiliki alokasi memori yang lebih sedikit dibandingkan *UCS* karena algoritma ini hanya membangkitkan simpul yang mengarah ke simpul tujuan. Namun, algoritma ini sulit menemukan rute yang optimal karena dapat terjebak pada simpul dengan nilai optimal pertama dan tidak bisa mencari simpul lainnya.

2.4 Algoritma A*

Algoritma A* merupakan algoritma hasil perpaduan dari *UCS* dan *GBFS*. *F(n)* yang digunakan oleh A* adalah total nilai dari *g(n)* dan *h(n)*. Sama seperti kedua algoritma sebelumnya, A* juga memiliki antrean dan membangkitkan simpul yang biayanya minimal. Algoritma A* memiliki kelebihan dalam penggunaan memori karena hanya membangkitkan simpul yang mengarah ke simpul tujuan dan pasti dapat memberikan solusi optimal dengan syarat *h(n) admissible*. *Admissible* adalah kondisi ketika nilai *h(n)* selalu lebih kecil atau sama dengan biaya sesungguhnya. Contohnya pada kasus pencarian jalur, nilai *h(n)* adalah jarak garis lurus suatu lokasi ke lokasi lainnya. Namun, *h(n)* pasti lebih kecil atau sama dengan jarak sesungguhnya karena jalan dari suatu lokasi ke lokasi lainnya bisa saja berbelok-belok. Jadi, pada kasus pencarian rute, penggunaan algoritma A* akan menghasilkan rute optimal.

BAB III
IMPLEMENTASI

Ketiga algoritma yang dibuat menggunakan struktur data yang sama, yaitu *Node* sebagai simpul dari kata-kata, *Priority Queue Node* sebagai penyimpan simpul-simpul yang telah dibangkitkan, *Set* yang berisi kata-kata dari kamus, dan *map* yang kuncinya adalah kata yang sudah ditinjau dan nilainya adalah kedalaman kata tersebut ditemukan. Terdapat sebuah kelas *Path* untuk membentuk jalur dari kata awal ke kata tujuan jika kata tujuan ditemukan. Fungsi *g(n)* dan *h(n)* pada kasus *word ladder* memiliki pengertian sebagai berikut:

1. *g(n)* adalah jumlah perubahan kata yang dilakukan pada n dari kata awal
2. *h(n)* adalah jumlah huruf pada n yang posisinya sama, tetapi hurufnya berbeda dengan huruf pada kata tujuan.

3.1 Implementasi Algoritma *Uniform Cost Search*

Algoritma *UCS* memiliki fungsi evaluasi dari *g(n)* saja. Implementasinya sebagai berikut:

1. Masukkan kata awal ke dalam *queue*.
2. Ambil simpul pada *head* dari *queue*.
3. Jika kata adalah kata tujuan, loncat ke langkah 8.
4. Cari kata-kata yang bisa dibangkitkan dari simpul tersebut dengan mengganti setiap huruf dari kata tersebut dengan A sampai Z satu per satu.
5. Jika kata ditemukan pada kamus, cari di map kata tersebut sudah dicek pada kedalaman sebelumnya atau belum.
6. Jika belum, tambahkan map dan bangkitkan sebuah simpul, hitung *g(n)* kemudian masukkan ke dalam *queue* dengan memprioritaskan simpul yang fungsi evaluasinya lebih kecil.
7. Lakukan langkah 2 sampai 5, sampai simpul yang diambil adalah kata tujuan.
8. Pastikan jalur yang ditemukan memiliki fungsi evaluasi yang minimal.
9. Jika belum minimal, kembali ke langkah 2.
10. Jika minimal, program selesai.

3.3 Implementasi Algoritma *Greedy Best First Search*

Algoritma *GBFS* memiliki fungsi evaluasi dari *h(n)* saja. Implementasinya sebagai berikut:

1. Masukkan kata awal ke dalam *queue*.
2. Ambil simpul pada *head* dari *queue*.
3. Jika kata adalah kata tujuan, loncat ke langkah 8.
4. Cari kata-kata yang bisa dibangkitkan dari simpul tersebut dengan mengganti setiap huruf dari kata tersebut dengan A sampai Z satu per satu.
5. Jika kata ditemukan pada kamus, cari di map kata tersebut sudah dicek pada kedalaman sebelumnya atau belum.
6. Jika belum, tambahkan map dan bangkitkan sebuah simpul, hitung *h(n)* kemudian masukkan ke dalam *queue* dengan memprioritaskan simpul yang fungsi evaluasinya lebih kecil.
7. Lakukan langkah 2 sampai 5, sampai simpul yang diambil adalah kata tujuan.
8. Pastikan jalur yang ditemukan memiliki fungsi evaluasi yang minimal.
9. Jika belum minimal, kembali ke langkah 2.
10. Jika minimal, program selesai.

3.4 Implementasi Algoritma A*

Algoritma A* memiliki fungsi evaluasi berupa jumlah *g(n)* dengan *h(n)*. Implementasinya sebagai berikut:

1. Masukkan kata awal ke dalam *queue*.
2. Ambil simpul pada *head* dari *queue*.
3. Jika kata adalah kata tujuan, loncat ke langkah 8.
4. Cari kata-kata yang bisa dibangkitkan dari simpul tersebut dengan mengganti setiap huruf dari kata tersebut dengan A sampai Z satu per satu.
5. Jika kata ditemukan pada kamus, cari di map kata tersebut sudah dicek pada kedalaman sebelumnya atau belum.
6. Jika belum, tambahkan map dan bangkitkan sebuah simpul, hitung total *h(n)* dan *g(n)* kemudian masukkan ke dalam *queue* dengan memprioritaskan simpul yang fungsi evaluasinya lebih kecil.
7. Lakukan langkah 2 sampai 5, sampai simpul yang diambil adalah kata tujuan.
8. Pastikan jalur yang ditemukan memiliki fungsi evaluasi yang minimal.
9. Jika belum minimal, kembali ke langkah 2.
10. Jika minimal, program selesai.

3.5 Analisis Implementasi Algoritma

Pada kasus permainan *word ladder*, fungsi *heuristic h(n)* menganggap jumlah huruf yang berbeda adalah jumlah langkah yang diperlukan. Misalnya dari kata “BABY” ke kata “CRIB” akan dianggap memerlukan empat kali perubahan oleh *h(n)*. Padahal, dari BABY ke CRIB memerlukan setidaknya sembilan kali perubahan. Contoh lainnya dari kata “BAR” ke “RUN” menurut *h(n)* memerlukan tiga kali perubahan dan nyatanya dari BAR ke RUN memerlukan setidaknya tiga kali perubahan. Jadi, nilai *h(n)* pasti kurang dari atau sama dengan jumlah perubahan sesungguhnya. Oleh karena itu, A* pada kasus ini pasti bersifat *admissible*.

Algoritma *GBFS* belum tentu menghasilkan jalur yang sama dengan algoritma *UCS*. Hal ini dikarenakan fungsi *g(n)* yang digunakan oleh *UCS* menghitung berapa kali perubahan dilakukan sehingga seluruh perubahan dari kata awal ke kata tujuan akan ditinjau secara bersamaan terus menerus. Sedangkan fungsi *h(n)* yang digunakan oleh *GBFS* awalnya memang melakukan hal yang sama seperti *UCS*, yaitu meninjau semua kata yang dibangkitkan, tetapi setelah menemukan satu kata saja dengan huruf yang sama pada posisi yang sama dengan kata tujuan, kata tersebut dan anak-anaknya akan terus dibangkitkan sampai tidak lagi ditemukan kata atau mencapai kata tujuan. Jadi, urutan simpul yang dibangkitkan *UCS* dan *GBFS* bisa jadi berbeda dan rute hasilnya juga kemungkinan besar berbeda.

Perbedaan algoritma A* dengan *UCS* adalah digunakannya fungsi *h(n)* oleh A* dalam fungsi evaluasinya. Namun, keduanya sama-sama menggunakan *g(n)* dalam fungsi evaluasinya. Kedua algoritma tersebut pasti akan memberikan solusi optimal pada permainan ini. Alasan A* pasti memberikan solusi optimal sudah dijelaskan pada paragraf pertama, sedangkan *UCS* pasti memberikan solusi optimal karena *UCS* menghitung banyak perubahan yang dilakukan sehingga seluruh simpul ditinjau secara merata sampai ditemukan kata akhir. Namun, algoritma A*, secara teori, lebih efisien dibandingkan *UCS* karena *UCS* membangkitkan semua simpul secara merata, sedangkan A* hanya membangkitkan simpul yang nilai *h(n)*-nya lebih rendah. Jadi, algoritma A* lebih efisien daripada algoritma *UCS*.

Algoritma *GBFS* akan memiliki kecepatan yang tinggi karena sekalinya menemukan *h(n)* yang lebih rendah, *GBFS* akan meninjau kata tersebut saja tanpa memedulikan simpul lainnya. Namun, solusi yang ditemukan *GBFS* belum tentu optimal karena bisa saja terdapat kata lain yang *h(n)*-nya lebih besar, tetapi jalurnya secara nyata lebih sedikit daripada kata lainnya. Jadi, *GBFS* tidak menjamin solusi optimal.

BAB IV
SOURCE CODE PROGRAM

4.1 *Class Node*

Berikut merupakan implementasi kelas *node* pada bahasa pemrograman Java:

```
public class Node {
    // Attributes
    private String word;
    private Node parent;
    private int depth;
    // Greedy value
    private int g;
    // Heuristic value
    private int h;

    // Methods
    // Constructors
    /**
     * Make a new Node
     * @param word the word in the node
     */
    public Node(String word) {
        this.word = word;
        this.parent = null;
        depth = 0;
        g = 0;
        h = 0;
    }

    /**
     * Make a new Node
     * @param word the word in the node
     * @param parent the Node it generated from
     */
    public Node(String word, Node parent) {
        this.word = word;
        this.parent = parent;
        if (parent == null) {
            depth = 0;
        } else {
```

```
            depth = parent.depth+1;
        }
        g = 0;
        h = 0;
    }

    /**
     * Make a new Node
     * @param word the word in the node
     * @param parent the Node it generated from
     * @param g the greedy value
     */
    public Node(String word, Node parent, int g) {
        this.word = word;
        this.parent = parent;
        this.depth = parent.depth + 1;
        this.g = g;
        h = 0;
    }

    /**
     * Make a new Node
     * @param n a Node that will be copied
     */
    public Node(Node n) {
        this.word = n.word;
        this.parent = n.parent;
        this.depth = n.depth;
        this.g = n.g;
        this.h = n.h;
    }

    // Getter
    /**
     * Getter for the word
     * @return word in the nod
     */
    public String getWord() { return word; }
    /**
     * Getter for parent Node
     * @return reference to the parent
     */
```

```
    public Node getParent() { return parent; }
    /**
     * Getter for depth
     * @return the depth of the node
     */
    public int getDepth() { return depth; }
    /**
     * Getter for g
     * @return greedy value of the node
     */
    public int getG() { return g; }
    /**
     * Getter for total g and h
     * @return the cost of the node
     */
    public int getCost() { return g+h; }

    /**
     * Finding the heuristic value of the word in this node
     * The heuristic value is the amount of different alphabet with
other
     * @param other destination string
     */
    public void changeH(String other) {
        int temp = 0;
        for (int i = 0; i < other.length(); i++) {
            if (word.charAt(i) != other.charAt(i)) {
                temp++;
            }
        }
        h = temp;
    }
}
```

Kelas *Node* berfungsi sebagai penyimpan kata dalam bentuk suatu simpul. Simpul tersebut akan menyimpan *reference* ke simpul asalnya. Simpul pertama yang dibangkitkan akan memiliki *attribute* parent *null*. Depth berfungsi untuk mencatat kedalaman simpul, g sebagai nilai *g(n)* dan h sebagai nilai *h(n)*. Kelas ini memiliki tiga *constructor* dan sebuah *copy constructor*. Kelas ini juga memiliki *getter* untuk atribut-atributnya. Terdapat sebuah atribut tambahan, yaitu changeH() untuk menghitung nilai *h(n)* berdasarkan kata yang diberikan.

4.2 *Class PrioQueueNode*

Berikut merupakan implementasi kelas *PrioQueueNode* pada bahasa pemrograman Java:

```
import java.util.*;

public class PrioQueueNode {
    // Attributes
    private List<Node> queueOfWords;
    private Map<String, Integer> foundWord; /* key = word, value =
depth found */
    private int generateAmount;

    //Methods
    /**
     * Make a new Priority Queue
     * @param begin the starting word
     */
    public PrioQueueNode(String begin) {
        queueOfWords = new ArrayList<Node>();
        foundWord = new HashMap<String, Integer>();
        foundWord.put(begin, 0);
        Node n = new Node(begin, null);
        enqueue(n);
    }

    /**
     * Getter for the queue
     * @return the queue it has as List<Node>
     */
    public List<Node> getQueueOfWords() { return queueOfWords; }
    /**
     * Getter for generateAmount
     * @return the amount of node that the process has made
     */
    public int getGeneratedNode() { return generateAmount; }

    /**
     * Insert a node to a queue. The lower thes cost of the node, the
higher the priority
     * @param n the new node to be inserted
     */
    private void enqueue(Node n) {
```

```
        int cost = n.getCost();
        int idx = 0;

        while (idx < queueOfWords.size() && cost >=
queueOfWords.get(idx).getCost()) {
            idx++;
        }
        queueOfWords.add(idx, n);
        foundWord.put(n.getWord(), n.getDepth());
    }

    /**
     * Erasing the head of the queue
     * @return The first element in the queue
     */
    public Node dequeue() {
        Node ret = queueOfWords.get(0);
        queueOfWords.remove(0);
        return ret;
    }

    /**
     * @return true if the queue is empty
     */
    public boolean isEmptyQueue() {
        return queueOfWords.isEmpty();
    }

    /**
     * Check the path has the most minimum cost
     * @param p The path to be checked
     * @return true if cost of queue's head >= path's cost
     */
    public boolean isPathMinimal(Path p) {
        if (isEmptyQueue()) { return true; }
        return (p.pathCost()) <= (queueOfWords.get(0).getCost());
    }

    /**
     * Check whether the word is already in the map with lower value
     * @param s the word to be checked
     * @param depth the depth of the word
```

```
     * @return true if word doesn't exist in the map or exists with
value <= depth
     */
    private boolean isWordChecked(String s, int depth) {
        if (foundWord.get(s) != null) {
            if (foundWord.get(s) >= depth) {
                foundWord.remove(s);
                return false;
            }
            return true;
        }
        return false;
    }

    /**
     * List all children of the node and enqueue them to the queue
     * The word will be enqueued if word is not found in parent,
     * isWordChecked returns false, dan word is in dictionary
     * @param now current node
     * @param n the destination word
     * @param greedy true if greedy function is needed
     * @param heuristic true if heuristic function is needed
     * @param d the dictionary
     */
    public void listChild(Node now, String n, boolean greedy, boolean
heuristic, Dictionary d) {
        generateAmount++;
        String x = now.getWord();
        int currentDepth = now.getDepth() + 1;
        int g = 0;
        if (greedy) { g = now.getG()+1; }
        for (int i = 0; i < x.length(); i++) {
            for (int j = 65; j <= 90; j++) {
                char c = (char)j;
                String temp = (x.substring(0, i) + c);
                if (i != x.length()-1) {
                    temp = temp + x.substring(i+1);
                }
                if ((d.isWordValid(temp)) && !(isWordChecked(temp,
currentDepth))) {
                    Node tail = new Node(temp, now, g);
                    if (heuristic) { tail.changeH(n); }
```

```
                    enqueue(tail);
                }
            }
        }
    }
}
```

Kelas ini memiliki atribut list sebagai *queue*-nya, *map* untuk menyimpan kata yang sudah dikunjungi, dan generateAmount untuk menghitung jumlah simpul yang dikunjungi. *Map* menyimpan kata beserta kedalaman kata tersebut ditemukan. *Constructor* memiliki atribut kata awal. Terdapat metode *getter* untuk *queue* dan generateAmount. Enqueue akan memasukan sebuah simpul baru ke *queue* diurutkan berdasarkan *cost* terkecil, dequeue akan mengambil elemen pertama dari *queue* dan mengembalikannya, isEmptyQueue untuk mengecek kosong tidaknya *queue* saat ini, isPathMinimal untuk memastikan bahwa biaya jalur yang ditemukan memang minimal, isWordChecked untuk memastikan apakah kata sudah dibangkitkan pada kedalaman sebelumnya, dan listChild untuk membangkitkan anak-anak dari suatu simpul dan menyimpannya ke dalam *queue*.

4.3 *Class Path*

Berikut merupakan implementasi kelas *Path* pada bahasa pemrograman Java:

```
import java.util.*;

public class Path {
    // Attributes
    private int cost;
    private List<String> result;

    // Methods
    /**
     * Make a new path
     */
    public Path() {
        cost = 0;
        result = new ArrayList<>();
    }

    /**
     * Getter for the cost of the path
     * @return path's cost
```

```
     */
    public int pathCost() {
        return cost;
    }
    /**
     * Getter for the path from the start word to destination word
     * @return path result as List<String>
     */
    public List<String> getResult() {
        return result;
    }

    /**
     * @return true if the path is empty (no path found)
     */
    public boolean isEmptyPath() {
        return result.isEmpty();
    }

    /**
     * Making path from the node to its root
     * @param n the first node (leaf node)
     */
    public void makePath(Node n) {
        Node temp = new Node(n);
        while (temp.getParent() != null) {
            result.add(temp.getWord());
            temp = new Node(temp.getParent());
        }
        result.add(temp.getWord());
        Collections.reverse(result);
    }

    /**
     * Erase the path found
     */
    public void cleanPath() {
        result.clear();
    }
}
```

Kelas ini merupakan kelas untuk menyimpan jalur di dalam list result dan biayanya pada atribut cost. Kedua atribut memiliki metode *getter* dan terdapat sebuah *constructor*. Metode

isEmptyPath berfungsi untuk mengecek sudah ditemukan jalur atau belum, makePath untuk membuat jalur dari suatu simpul, dan cleanPath untuk mengosongkan isi list.

4.4 *Class Process*

Berikut merupakan implementasi kelas *Process* pada bahasa pemrograman Java:

```
public class Process {
    // Atrributes
    private String end;
    private PrioQueueNode q;
    private Path p;
    private Dictionary d;
    private boolean greedy, heuristic;

    /**
     * Make a new process
     * @param begin starting word
     * @param end destination word
     * @param greedy true if the greedy function is used (UCS and A*)
     * @param heuristic true if the heuristic fucntion is used (G-BFS
and A*)
     * @param d the dictionary for the words
     */
    public Process(String begin, String end, boolean greedy, boolean
heuristic, Dictionary d) {
        this.end = end;
        q = new PrioQueueNode(begin);
        p = new Path();
        this.d = d;
        this.greedy = greedy;
        this.heuristic = heuristic;
    }

    /**
     * Getter for the path (result list)
     * @return the path in the process
     */
    public Path getPath() { return p; }
    /**
     * Getter for
     * @return the queue of nodes
     */
```

```
    public PrioQueueNode getQueue() { return q; }

    /**
     * Check the word in the node is the destination word
     * @param n the node to be checked
     * @return true if word in node is destination, false otherwise
     */
    private boolean found(Node n) {
        return end.equals(n.getWord());
    }

    /**
     * Check the path has the most minimun cost
     * @return true if the path's cost is minimum form every node in
q
     */
    public boolean isMostMinimum() { return q.isPathMinimal(p); }

    /**
     * Generating child of a node in the head of the queue
     */
    public void generateChild() {
        Node head = q.dequeue();
        if (found(head)) {
            if (!p.isEmptyPath()) {
                p.cleanPath();
            }
            p.makePath(head);
        } else {
            q.listChild(head, end, greedy, heuristic, d);
        }
    }
}
```

Kelas ini merupakan kelas untuk menjalankan fungsi-fungsi dan algoritma *path planning*. Kelas memiliki atribut kata tujuan, *queue*, *path*, kamus, dan *boolean* untuk menentukan fungsi yang akan digunakan selama proses berjalan. *greedy* akan bernilai *true* jika algoritma yang digunakan adalah *UCS* atau A* dan *heuristic* akan bernilai benar jika algoritma yang digunakan adalah *GBFS* atau A*. *Constructor* akan membentuk objek dari kelas-kelas lain. Terdapat metode *getter* untuk *queue* dan *path* pada proses. Metode found untuk mengecek apakah kata yang dicek merupakan kata tujuan, isMostMinimum untuk memastikan bahwa

jalur yang ditemukan memang bernilai minimal, generateChild sebagai proses untuk membangkitkan simpul-simpul baru.

4.5 *Class Dictionary*

Berikut merupakan implementasi kelas *Dictionary* pada bahasa pemrograman Java:

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

public class Dictionary {
    // Atribute
    private Set<String> kamus;

    // Methods
    /**
     * Make a new dictionary from the txt file
     */
    public Dictionary() {
        kamus = new HashSet<String>();
        try {
            BufferedReader f = new BufferedReader(new
FileReader("bin/dictionary.txt"));
            String line;
            while ((line = f.readLine()) != null) {
                line = line.trim().toUpperCase();
                kamus.add(line);
            }
            f.close();
        } catch (IOException e) {
            System.out.println("Terjadi kesalahan. Mohon jalankan
ulang");
        }
    }

    /**
     * Searching a word in the dictionary
     * @param s the word to be looked for
     * @return true if s exists in the dictionary, false otherwise
     */
    public boolean isWordValid(String s) {
        return kamus.contains(s);
    }
```

```
}
```

Kelas ini berfungsi sebagai kamus untuk mengecek kata yang akan dibangkitkan. Kata-kata akan disimpan dalam sebuah *set*. Kamus dibentuk dari sebuah file teks yang berisi kata-kata dalam bahasa Inggris. Metode isWordValid berfungsi untuk mengecek apakah kata yang ditinjau ada di dalam kamus.

4.6 *Class Main*

Berikut merupakan implementasi kelas *Main* pada bahasa pemrograman Java:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Main {
    public static void main(String[] args) {
        // Making dictionary
        Dictionary dict = new Dictionary();

        // Main frame
        JFrame f = new JFrame("Word Ladder Solver");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(430, 470);
        f.setLayout(new FlowLayout());

        // Title
        JPanel title = new JPanel();
        JLabel header = new JLabel("WORD LADDER SOLVER");
        title.add(header);
        title.setLayout(new FlowLayout(FlowLayout.CENTER, 0, 10));

        // Input field (statring word and destination word)
        JPanel panel1 = new JPanel();
        panel1.setLayout(new FlowLayout(FlowLayout.CENTER, 0, 10));
        JLabel start = new JLabel("Starting word: ");
        JTextField input1 = new JTextField(20);
        panel1.add(start);
        panel1.add(input1);

        JPanel panel2 = new JPanel();
        panel2.setLayout(new FlowLayout(FlowLayout.CENTER, 0, 10));
        JLabel end = new JLabel("Destination word: \n");
```

```
        JTextField input2 = new JTextField(18);
        panel2.add(end);
        panel2.add(input2);

        // Button for running the program
        JPanel buttonPanel = new JPanel();
        JButton submitButton = new JButton("FIND");
        buttonPanel.add(submitButton);
        buttonPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5,
10));

        // Radio buttons
        JRadioButton radioButton1 = new JRadioButton("UCS");
        JRadioButton radioButton2 = new JRadioButton("G-BFS");
        JRadioButton radioButton3 = new JRadioButton("A*");

        // Initially set "UCS" radio button as selected
        radioButton1.setSelected(true);

        // Button group
        JLabel tekspilihan = new JLabel("CHOOSE ALGORITHM");
        JPanel teksSementara = new JPanel();
        teksSementara.add(tekspilihan);
        teksSementara.setLayout(new FlowLayout(FlowLayout.CENTER, 5,
10));
        ButtonGroup buttonGroup = new ButtonGroup();
        buttonGroup.add(radioButton1);
        buttonGroup.add(radioButton2);
        buttonGroup.add(radioButton3);
        JPanel algoritma = new JPanel();
        algoritma.setLayout(new FlowLayout(FlowLayout.CENTER, 5,
10));
        algoritma.add(radioButton1);
        algoritma.add(radioButton2);
        algoritma.add(radioButton3);

        // Output field
        JPanel resPanel = new JPanel();
        JLabel res = new JLabel();
        resPanel.add(res);
        resPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5, 10));
        JPanel nodePanel = new JPanel();
```

```
        JLabel nodeLabel = new JLabel();
        nodePanel.add(nodeLabel);
        nodePanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5,
10));
        JPanel timePanel = new JPanel();
        JLabel timeLabel = new JLabel();
        timePanel.add(timeLabel);
        timePanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5,
10));
        JPanel resultPanel = new JPanel();
        JLabel resultLabel = new JLabel();
        resultPanel.add(resultLabel);
        resultPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5,
10));
        JPanel memoryPanel = new JPanel();
        JLabel memoryLabel = new JLabel();
        memoryPanel.add(memoryLabel);
        memoryPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5,
10));

        // Main content
        JPanel contentPane = new JPanel();
        contentPane.setLayout(new BoxLayout(contentPane,
BoxLayout.Y_AXIS));
        contentPane.add(title);
        contentPane.add(panel1);
        contentPane.add(panel2);
        contentPane.add(Box.createVerticalStrut(10)); // Add some
vertical space
        contentPane.add(teksSementara);
        contentPane.add(algoritma);
        contentPane.add(Box.createVerticalStrut(10)); // Add some
vertical space
        contentPane.add(buttonPanel);
        contentPane.add(Box.createVerticalStrut(10)); // Add some
vertical space
        contentPane.add(resPanel);
        contentPane.add(resultPanel);
        contentPane.add(nodePanel);
        contentPane.add(timePanel);
        contentPane.add(memoryPanel);
```

```
        // Adding scroll pane
        JScrollPane scrollPane = new JScrollPane(contentPane,
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);

        // Action listener to the submit button
        submitButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Runtime r = Runtime.getRuntime();
                Process pr;

                // Get the values of input1 and input2
                String startingWord = input1.getText().toUpperCase();
                String destinationWord =
input2.getText().toUpperCase();

                // Check whether the words valid or not
                if (startingWord.length() !=
destinationWord.length()) {
                    JOptionPane.showMessageDialog(f, "Input words
must have the same length.");
                    return;
                }
                if (!dict.isWordValid(startingWord) ||
!dict.isWordValid(destinationWord)) {
                    JOptionPane.showMessageDialog(f, "Input words
must be valid.");
                    return;
                }

                // Starting process
                long startTime = System.currentTimeMillis();
                r.gc();
                long beforeMem = r.totalMemory() - r.freeMemory();
                pr = new Process(startingWord, destinationWord,
false, false, dict);
                // Get the selected algorithm
                if (radioButton1.isSelected()) {
                    pr = new Process(startingWord, destinationWord,
true, false, dict);
                } else if (radioButton2.isSelected()) {
                    pr = new Process(startingWord, destinationWord,
```

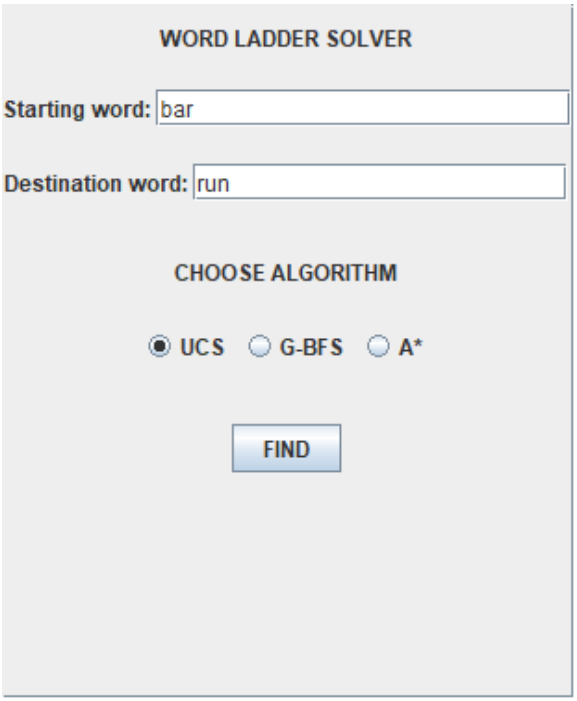
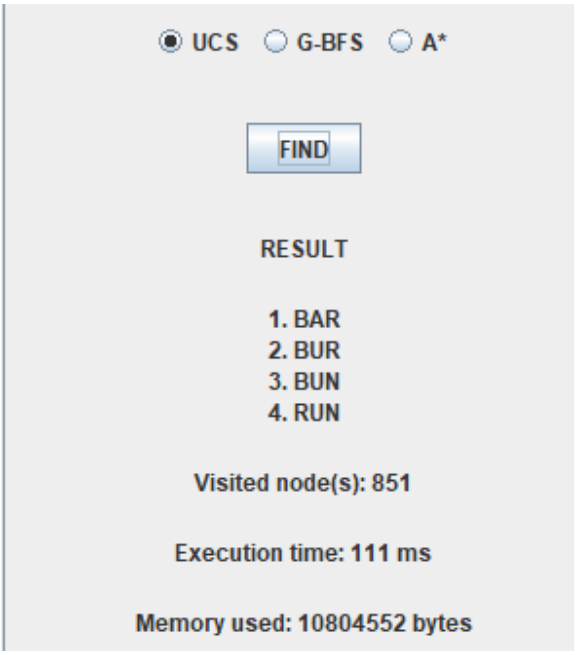
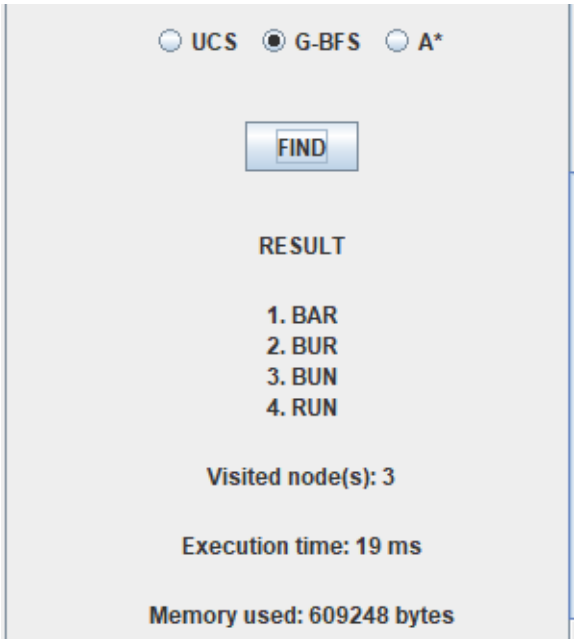
```
false, true, dict);
                } else if (radioButton3.isSelected()) {
                    pr = new Process(startingWord, destinationWord,
true, true, dict);
                }

                // Making child nodes untill found a path or queue is
empty
                while (pr.getPath().isEmptyPath() &&
!(pr.getQueue().isEmptyQueue())) {
                    pr.generateChild();
                }

                // Continue if the path found is not minimum
                while (!pr.isMostMinimum() &&
!(pr.getQueue().isEmptyQueue())) {
                    pr.generateChild();
                }
                long afterMem = r.totalMemory() - r.freeMemory();
                long stopTime = System.currentTimeMillis();

                // Displaying Result
                res.setText("RESULT");
                if (pr.getPath().isEmptyPath()) {
                    resultLabel.setText("No path found");
                } else {
                    String sb = "";
                    for (int i = 0; i <
pr.getPath().getResult().size(); i++) {
                        sb = sb + ((i+1) + ". " +
pr.getPath().getResult().get(i) + "<br>");
                    }
                    resultLabel.setText("<html>" + sb + "</html>");
                }
                nodeLabel.setText("Visited node(s): " +
pr.getQueue().getGeneratedNode());
                timeLabel.setText("Execution time: " +
(stopTime-startTime) + " ms");
                memoryLabel.setText("Memory used: " + (afterMem -
beforeMem) + " bytes");
            }
        });
```

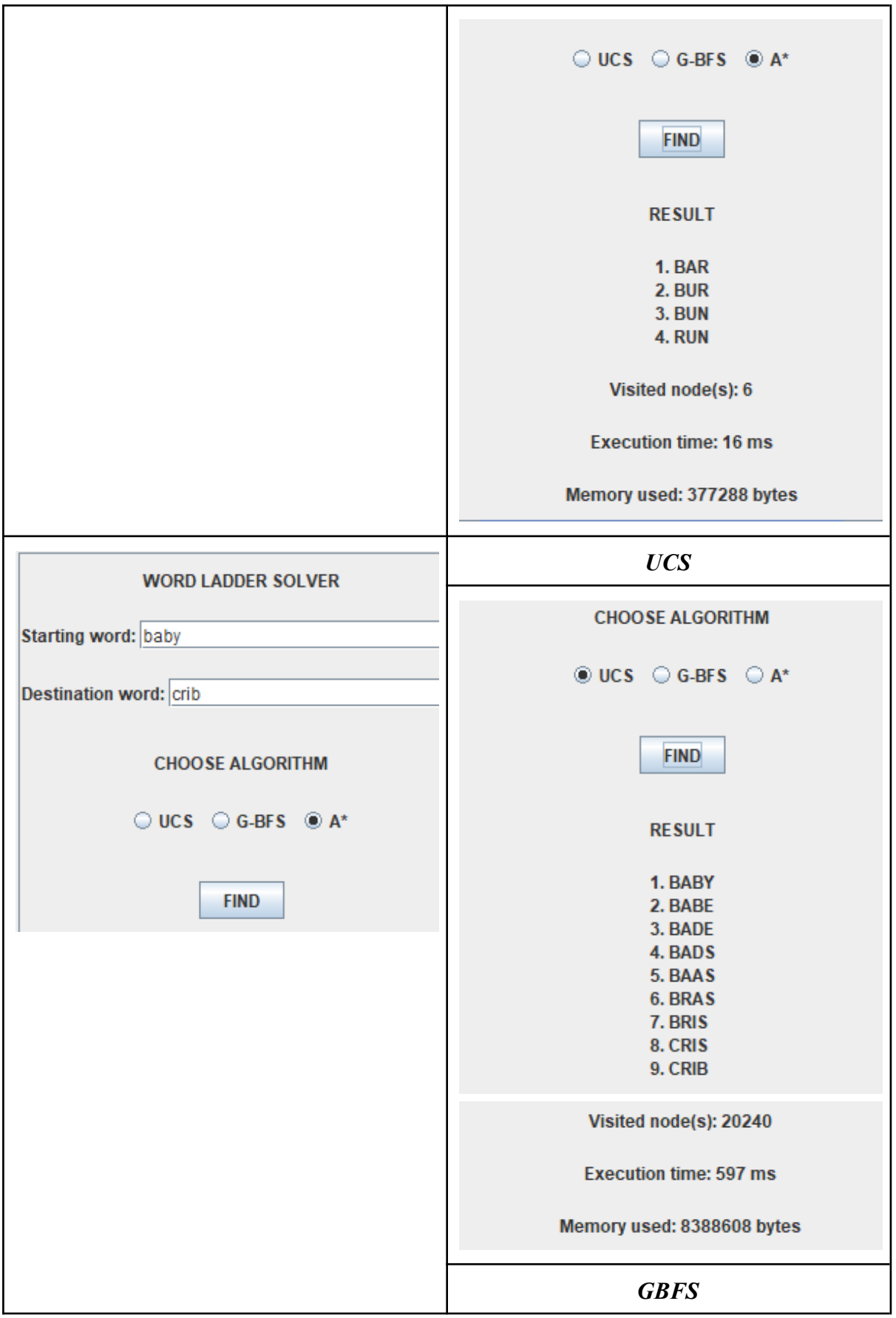
```
        // Adding scroller
        scrollPane.setViewportView(contentPane);
        f.add(scrollPane);

        // Center the frame on the screen
        f.setLocationRelativeTo(null);

        f.setVisible(true);
    }
}
```

Kelas *Main* merupakan kelas utama yang akan menjalankan program. Pada awal, kelas ini akan membuat sebuah kamus yang akan digunakan selama program berjalan. Kemudian, kelas akan membangun label-label untuk keperluan antarmuka. Program menunggu input dari pengguna, yaitu kata awal, kata tujuan, dan algoritma yang dipilih. Ketika tombol 'FIND' ditekan, program akan membuat sebuah proses yang akan mencari kata tujuan. Jika ditemukan kata tujuan dengan jalur yang optimal berdasarkan algoritma, program akan berhenti.

BAB V
PENGUJIAN PROGRAM DAN ANALISIS PERBANDINGAN

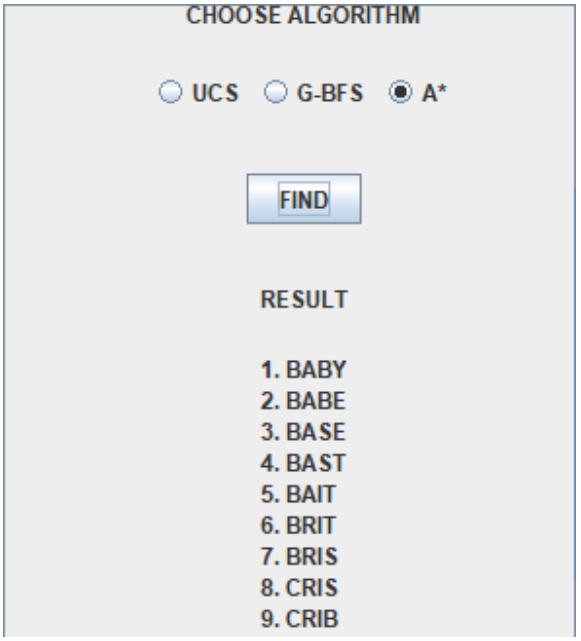
5.1 Pengujian Program

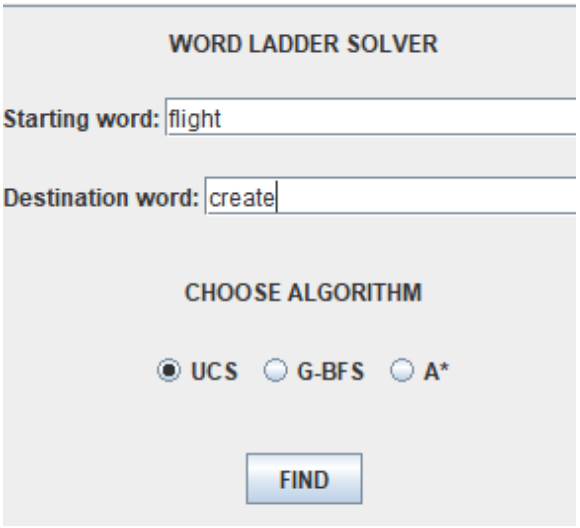
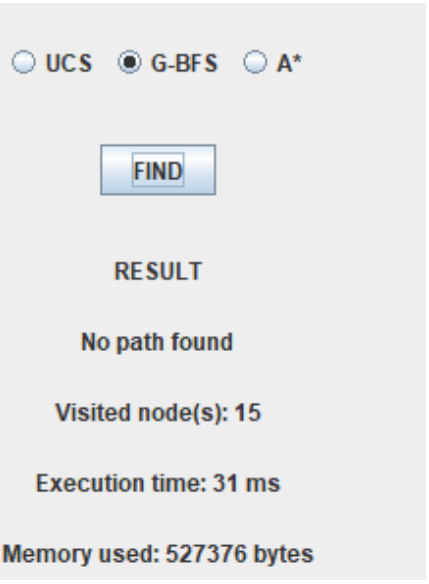
<table>
<tr><th>Test Case</th><th>Result</th></tr>
<tr><td rowspan="6"> 
</td><td>UCS</td></tr>
<tr><td> 
</td></tr>
<tr><td>GBFS</td></tr>
<tr><td> 
</td></tr>
<tr><td>A*</td></tr>
</table>

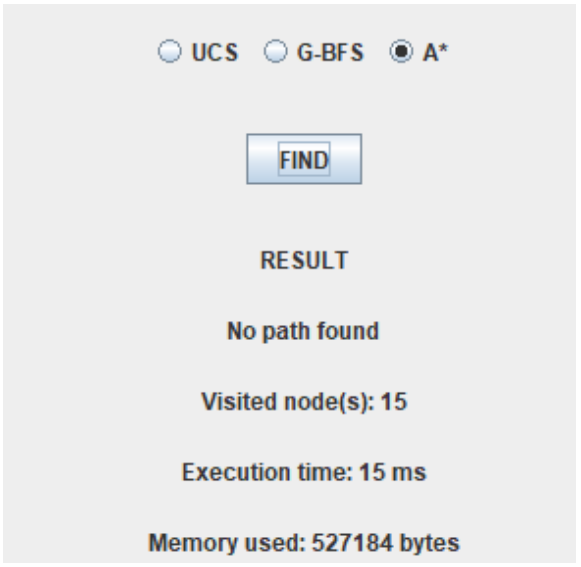

UCS G-BFS A*
FIND
RESULT
1. BAR
2. BUR
3. BUN
4. RUN
Visited node(s): 6
Execution time: 16 ms
Memory used: 377288 bytes
WORD LADDER SOLVER
Starting word: baby
Destination word: crib
CHOOSE ALGORITHM
UCS G-BFS A*
FIND
UCS
CHOOSE ALGORITHM
UCS G-BFS A*
FIND
RESULT
1. BABY
2. BABE
3. BADE
4. BADS
5. BAAS
6. BRAS
7. BRIS
8. CRIS
9. CRIB
Visited node(s): 20240
Execution time: 597 ms
Memory used: 8388608 bytes
GBFS

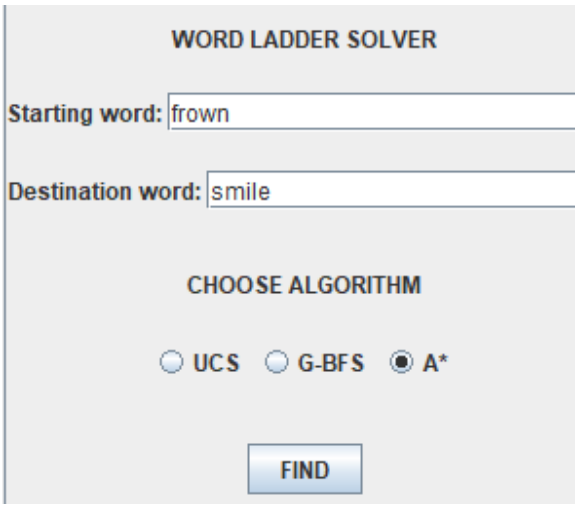

CHOOSE ALGORITHM
UCS
G-BFS
A*
FIND
RESULT
1. BABY
2. GABY
3. GAMY
4. GAMB
5. GARB
6. CARB
7. CARD
8. CAID
9. CHID
10. CHIS
11. CRIS
12. CRIB
Visited node(s): 27
Execution time: 16 ms
Memory used: 608256 bytes

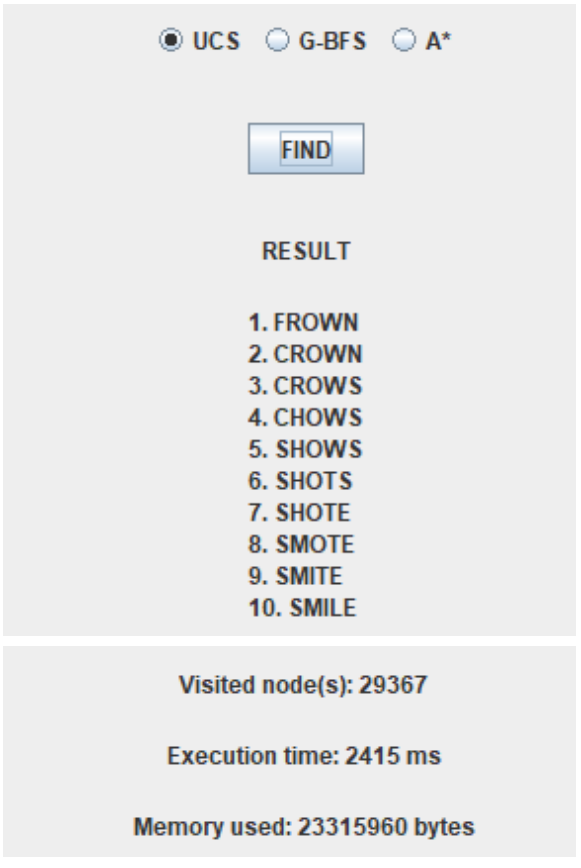
A*


CHOOSE ALGORITHM
UCS
G-BFS
A*
FIND
RESULT
1. BABY
2. BABE
3. BASE
4. BAST
5. BAIT
6. BRIT
7. BRIS
8. CRIS
9. CRIB

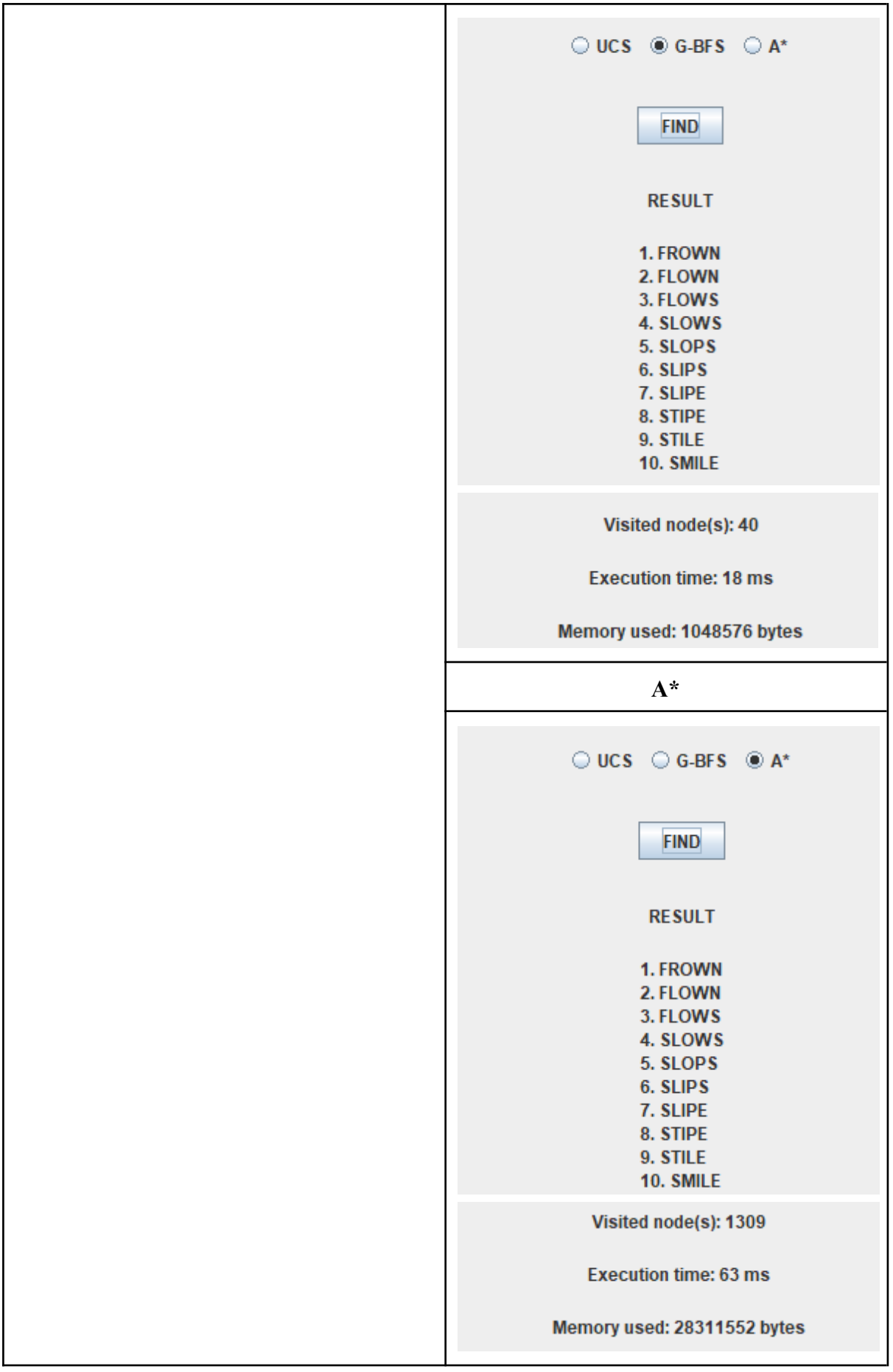
<table>
<tr><td></td><td>Visited node(s): 2098
Execution time: 188 ms
Memory used: 17902176 bytes</td></tr>
<tr><td rowspan="4"> 
</td><td>UCS</td></tr>
<tr><td>RESULT
No path found
Visited node(s): 15
Execution time: 15 ms
Memory used: 856072 bytes</td></tr>
<tr><td>GBFS</td></tr>
<tr><td> 
</td></tr>
<tr><td>A*</td></tr>
</table>

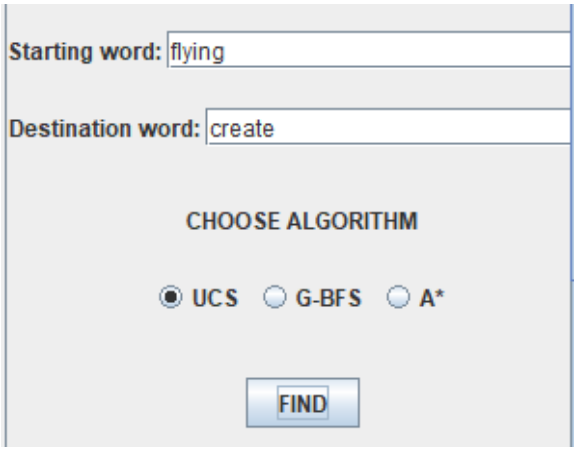
<table>
<tr>
<td rowspan="2"></td>
<td>


</td>
</tr>
<tr>
<td>


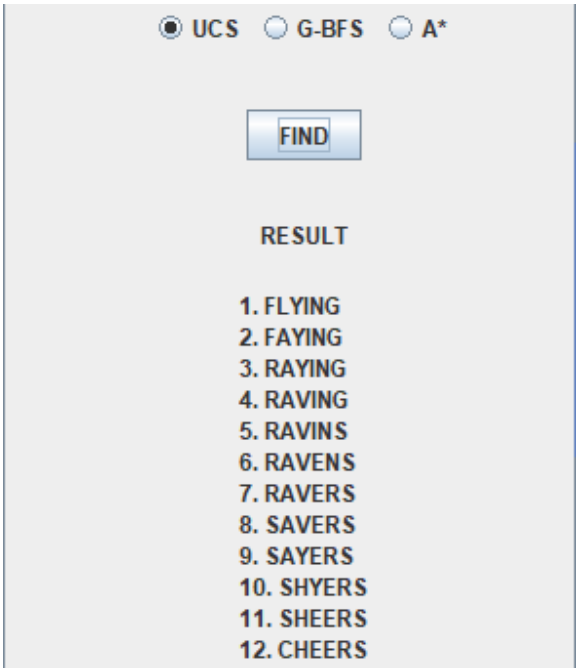
</td>
</tr>
<tr>
<td></td>
<td>UCS</td>
</tr>
<tr>
<td rowspan="2"></td>
<td>


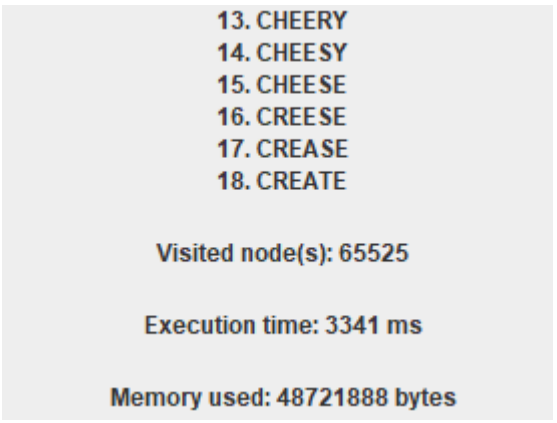
</td>
</tr>
<tr>
<td></td>
<td>GBFS</td>
</tr>
</table>


UCS
G-BFS
A*
FIND
RESULT
1. FROWN
2. FLOWN
3. FLOWS
4. SLOWS
5. SLOPS
6. SLIPS
7. SLIPE
8. STIPE
9. STILE
10. SMILE
Visited node(s): 40
Execution time: 18 ms
Memory used: 1048576 bytes
A*
UCS
G-BFS
A*
FIND
RESULT
1. FROWN
2. FLOWN
3. FLOWS
4. SLOWS
5. SLOPS
6. SLIPS
7. SLIPE
8. STIPE
9. STILE
10. SMILE
Visited node(s): 1309
Execution time: 63 ms
Memory used: 28311552 bytes

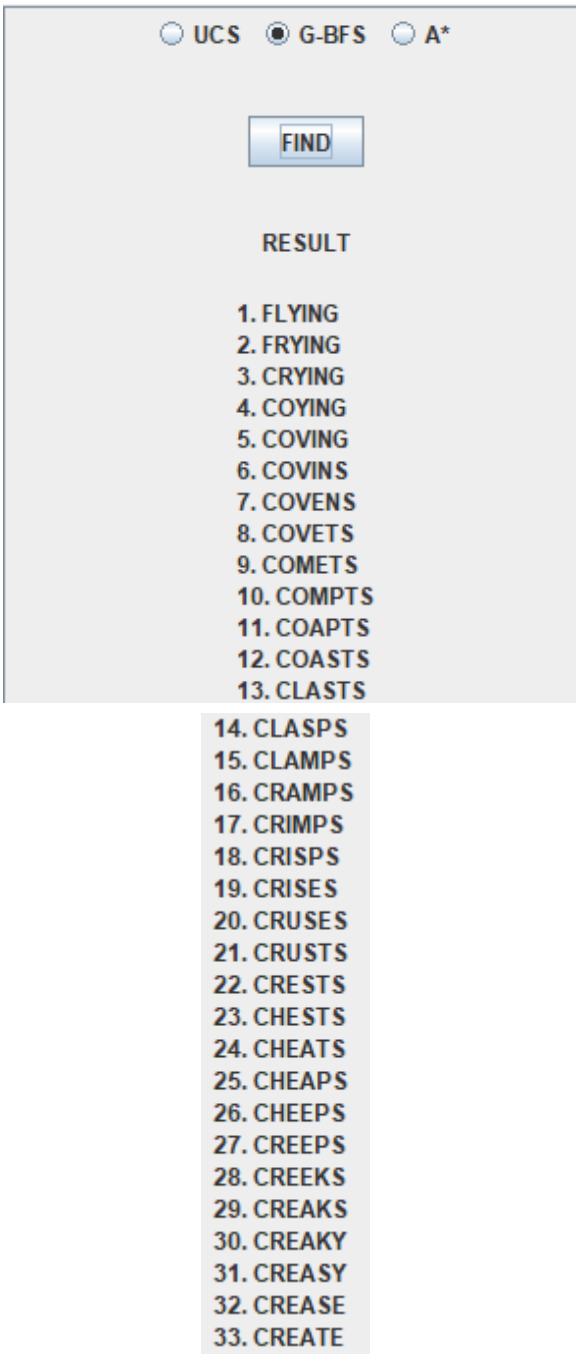

Starting word: flying
Destination word: create
CHOOSE ALGORITHM
UCS G-BFS A*
FIND

UCS


UCS G-BFS A*
FIND
RESULT
1. FLYING
2. FAYING
3. RAYING
4. RAVING
5. RAVINS
6. RAVENS
7. RAVERS
8. SAVERS
9. SAYERS
10. SHYERS
11. SHEERS
12. CHEERS

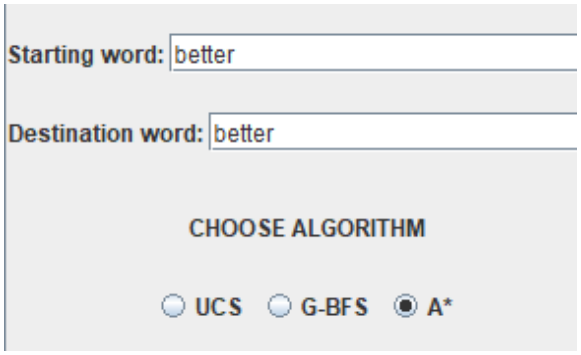

13. CHEERY
14. CHEESY
15. CHEESE
16. CREESE
17. CREASE
18. CREATE
Visited node(s): 65525
Execution time: 3341 ms
Memory used: 48721888 bytes

GBFS

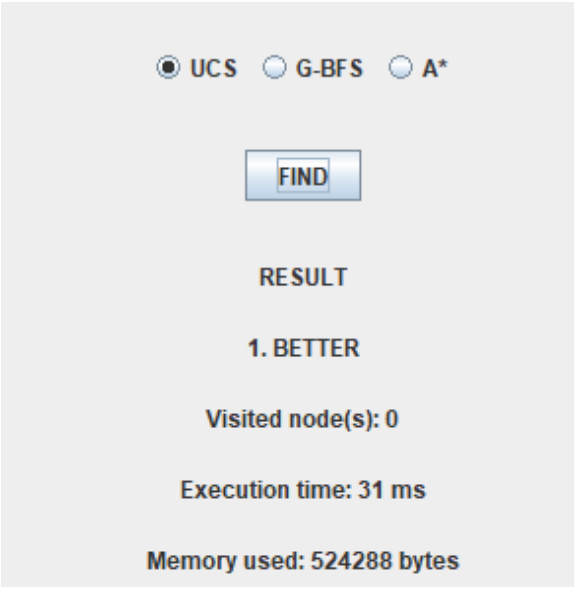

UCS
G-BFS
A*
FIND
RESULT
1. FLYING
2. FRYING
3. CRYING
4. COYING
5. COVING
6. COVINS
7. COVENS
8. COVETS
9. COMETS
10. COMPTS
11. COAPTS
12. COASTS
13. CLASTS
14. CLASPS
15. CLAMPS
16. CRAMPS
17. CRIMPS
18. CRISPS
19. CRISES
20. CRUSES
21. CRUSTS
22. CRESTS
23. CHESTS
24. CHEATS
25. CHEAPS
26. CHEEPS
27. CREEPS
28. CREEKS
29. CREAKS
30. CREAKY
31. CREASY
32. CREASE
33. CREATE
Visited node(s): 301
Execution time: 31 ms
Memory used: 8388608 bytes

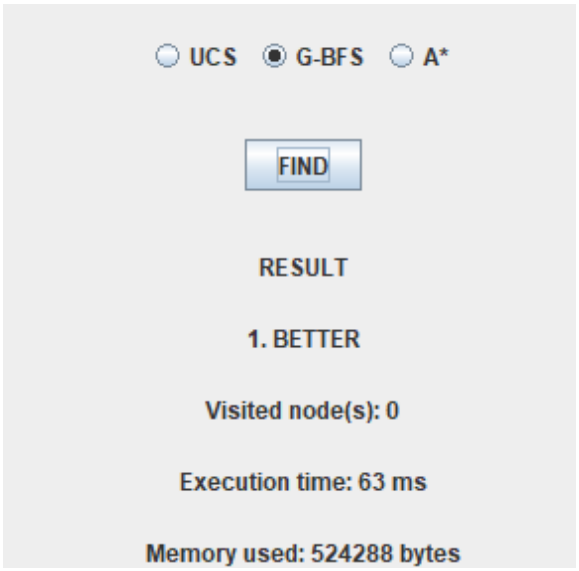
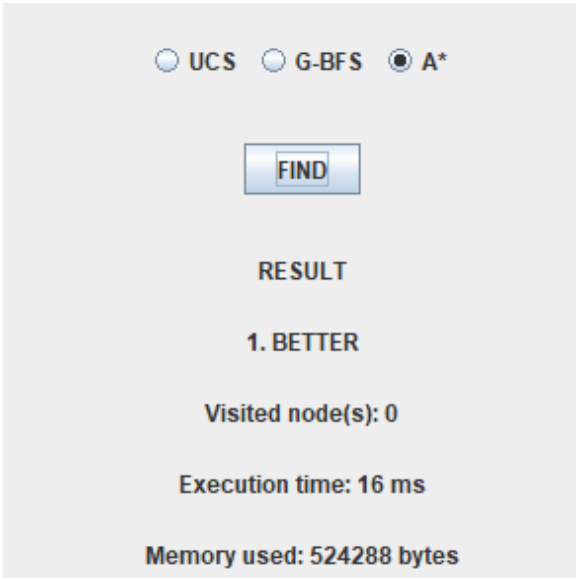
A*


UCS
G-BFS
A*
FIND
RESULT
1. FLYING
2. FAYING
3. RAYING
4. RAVING
5. RAVINS
6. RAVENS
7. RAVERS
8. SAVERS
9. SAYERS
10. SHYERS
11. SHEERS
12. CHEERS
13. CHEERY
14. CHEESY
15. CHEESE
16. CREESE
17. CREASE
18. CREATE
Visited node(s): 12584
Execution time: 612 ms
Memory used: 26948080 bytes


Starting word: better
Destination word: better
CHOOSE ALGORITHM
UCS
G-BFS
A*

UCS


UCS
G-BFS
A*
FIND
RESULT
1. BETTER
Visited node(s): 0
Execution time: 31 ms
Memory used: 524288 bytes

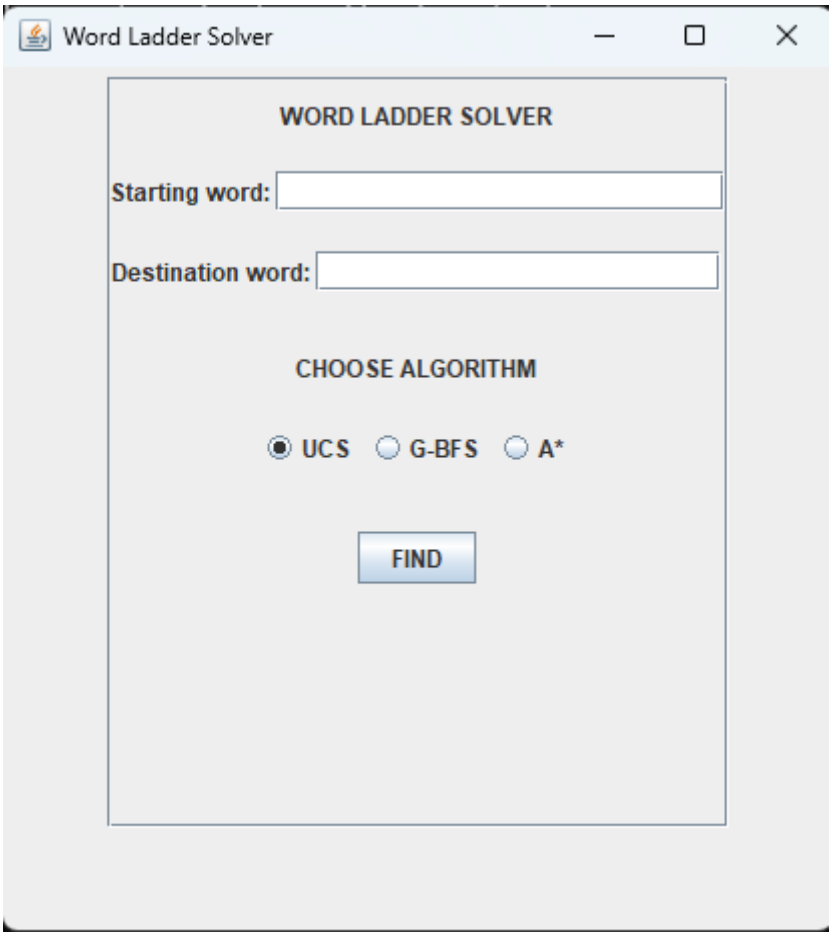
	GBFS
	
	A*
	

5.2 Analisis Perbandingan

Berdasarkan pengujian yang telah dilakukan, eksekusi waktu *GBFS* selalu lebih cepat daripada kedua algoritma lainnya. Waktu eksekusi A* lebih cepat dibandingkan *UCS*, hal ini sesuai dengan teori yang telah dipaparkan sebelumnya. Dilihat dari hasil solusi, *GBFS* tidak selalu memberikan solusi optimal, sedangkan algoritma A* dan *UCS* selalu berhasil memberikan jalur yang paling optimal. Dilihat pada penggunaan memori, A* cenderung menggunakan memori lebih sedikit dibandingkan *UCS* dan *GBFS*. *GBFS* menggunakan memori yang lebih besar dibandingkan *UCS*.

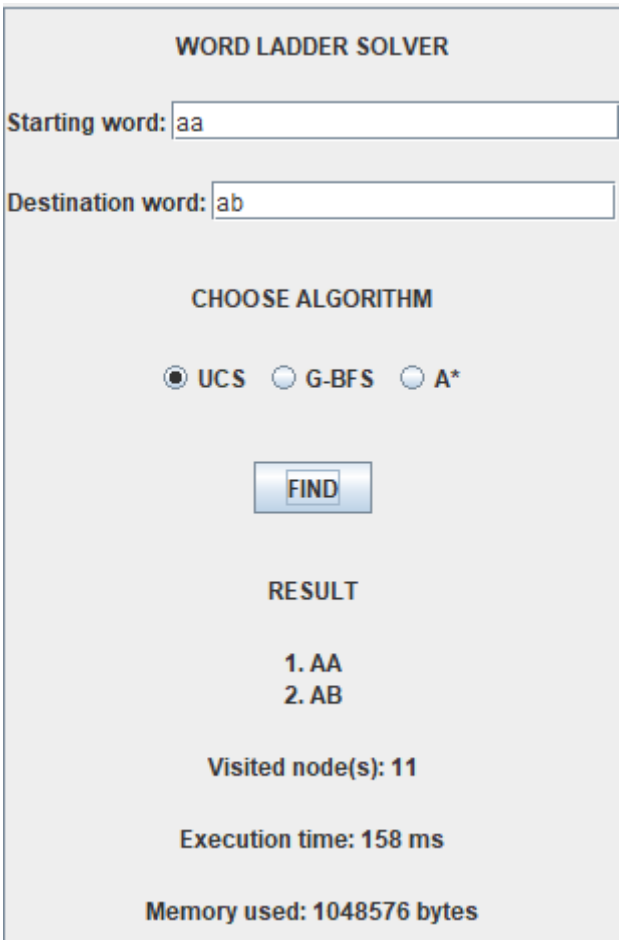
5.3 Implementasi Bonus

Bonus dalam tugas kecil ini adalah membuat *GUI* (*graphical user interface*). Untuk menerapkan bonus ini, digunakan Java Swing. *GUI* dirancang dengan membungkus *input form* untuk memasukkan kata-kata dan *radio button* untuk memilih algoritma yang akan digunakan. Program akan mengecek terlebih dahulu apakah kata yang diberikan valid atau tidak, kemudian melakukan pencarian. Hasil akan ditampilkan berurutan setelah pengguna menjalankan program. Program akan menampilkan *frame* berikut:



Gambar 5.3.1 Tampilan Awal Program

Setelah diberikan masukkan, program akan menampilkan hasil sebagai berikut:



Gambar 5.3.2 Tampilan Program Setelah Dijalankan

BAB VI
KESIMPULAN DAN SARAN

6.1 Kesimpulan

Berdasarkan analisis dan pengujian yang telah dilakukan sebelumnya, urutan algoritma dari yang paling cepat adalah *GBFS*, A*, dan *UCS*. Urutan algoritma dari yang menghasilkan solusi optimal adalah A* dengan *UCS* dan *GBFS*. Sedangkan, urutan berdasarkan penggunaan memori terendah adalah A*, *GBFS*, dan *UCS*.

Ketiganya memiliki kelebihan dan kelemahannya sendiri. A* memiliki hasil yang cukup memuaskan karena seimbang antara penggunaan memori dan kecepatannya, *UCS* dapat menghasilkan solusi yang pasti optimal, dan *GBFS* mampu memberikan hasil yang jauh lebih cepat dibandingkan *UCS* ataupun A*. Namun, untuk pencarian solusi optimal, lebih baik digunakan A* karena A* selalu memberikan hasil yang optimal seperti *UCS*, dan memiliki waktu eksekusi yang cukup cepat, serta menggunakan memori yang lebih rendah.

6.2 Saran

Ketiga algoritma mampu memberikan solusi jika solusi memang ada. Oleh karena itu, algoritma yang digunakan harus menyesuaikan tujuan dan sumber daya yang ada. Apabila ingin meminimalkan waktu eksekusi tanpa memedulikan optimasi, dapat digunakan *GBFS*. Jika ingin mengoptimalkan hasil, sebaiknya menggunakan algoritma A*.

LAMPIRAN

Tautan Repository GitHub :

https://github.com/BryanLauw/Tucil3_13522033

Progress Tracking

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	✔	

DAFTAR PUSTAKA

Munir, Rinaldi. *Penentuan rute (Route/Path Planning) - Bagian 1: BFS, DFS, UCS, Greedy Best First Search*. Diakses 3 Mei 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf

Munir, Rinaldi. *Penentuan rute (Route/Path Planning) - Bagian 2: Algoritma A**. Diakses 3 Mei 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf