

LAPORAN TUGAS KECIL 3
IF2211 STRATEGI ALGORITMA
PENYELESAIAN PERMAINAN *WORD LADDER*
MENGGUNAKAN ALGORITMA UCS, GBFS, DAN A*



Disusun oleh:
Christopher Brian - 13522106

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

DAFTAR ISI

DAFTAR ISI	2
ANALISIS DAN IMPLEMENTASI ALGORITMA UCS	3
ANALISIS DAN IMPLEMENTASI ALGORITMA GBFS	4
ANALISIS DAN IMPLEMENTASI ALGORITMA A*	5
<i>SOURCE CODE</i> PROGRAM	7
<i>TESTING</i>	23
ANALISIS PERBANDINGAN	26
IMPLEMENTASI BONUS	27
LAMPIRAN	28
PRANALA	29

ANALISIS DAN IMPLEMENTASI ALGORITMA UCS

Algoritma *Uniform Cost Search* (UCS) merupakan algoritma pencarian rute yang mencari rute dengan bobot atau harga terkecil. Dalam permainan *Word Ladder*, setiap langkah memiliki harga satu, sehingga algoritma UCS akan mencari rute terpendek dari kata asal ke kata tujuan. Algoritma ini akan mengevaluasi setiap simpul atau kata dengan heuristik $f(n) = g(n)$ di mana $f(n)$ merupakan total harga untuk mencapai kata tujuan dari kata asal dan $g(n)$ merupakan panjang rute dari kata asal ke kata sekarang.

Ketika setiap langkah memiliki harga yang sama, algoritma UCS akan menghasilkan rute yang sama dengan algoritma *Breadth First Search* (BFS). Namun, terdapat perbedaan di urutan simpul yang dibangkitkan. Algoritma UCS akan mengekskansi simpul dengan harga terendah yang belum dikunjungi, sedangkan algoritma BFS akan mengekskansi simpul terdalam yang belum dikunjungi.

Untuk implementasi algoritma UCS, akan digunakan sebuah *priority queue* untuk menyimpan kata berdasarkan panjang rute dari kata asal dan sebuah *set* untuk menyimpan kata yang telah dikunjungi. Kata asal akan menjadi kata yang pertama kali dimasukkan ke *priority queue*. Kemudian, akan dilakukan eksplorasi setiap kata pada *priority queue* sampai *priority queue* kosong. Program akan mengambil kata dengan $g(n)$ terendah, yaitu kata dengan jarak terdekat dari kata asal. Jika kata yang diambil merupakan kata tujuan, maka program akan menghasilkan rute dari kata asal ke kata tersebut. Jika tidak, program akan mengeksplorasi setiap kata tetangga (kata yang berbeda tepat satu huruf) dari kata yang sedang diambil. Jika kata tetangga belum dikunjungi, atur harga kata tersebut menjadi harga kata sekarang ditambah satu dan tambahkan ke *priority queue* sebagai simpul baru. Jika semua simpul telah selesai dieksplorasi dan rute tidak ditemukan, program akan mengembalikan *null*.

ANALISIS DAN IMPLEMENTASI ALGORITMA GBFS

Algoritma *Greedy Best First Search* (GBFS) merupakan algoritma pencarian rute yang selalu memilih solusi optimal lokal pada setiap langkah, yaitu simpul dengan harga terkecil. Dalam permainan *Word Ladder*, harga sebuah simpul tetangga bernilai jumlah huruf yang berbeda dengan simpul tujuan. Algoritma ini akan mengevaluasi setiap simpul atau kata dengan heuristik $f(n) = h(n)$ di mana $f(n)$ merupakan estimasi total harga untuk mencapai kata tujuan dari kata sekarang dan $h(n)$ merupakan estimasi harga dari simpul sekarang ke simpul tujuan, yaitu jumlah huruf yang berbeda antara simpul sekarang ke simpul tujuan. Algoritma ini tidak menjamin solusi optimal global untuk persoalan *Word Ladder*.

Untuk implementasi algoritma GBFS, akan digunakan sebuah *priority queue* untuk menyimpan kata berdasarkan panjang rute dari kata asal dan sebuah *set* untuk menyimpan kata yang telah dikunjungi. Kata asal akan menjadi kata yang pertama kali dimasukkan ke *priority queue*. Kemudian, akan dilakukan eksplorasi setiap kata pada *priority queue* sampai *priority queue* kosong. Program akan mengambil kata dengan $h(n)$ terendah (kata dengan jumlah huruf berbeda terkecil dengan kata tujuan) dari *priority queue*. Jika kata yang diambil merupakan kata tujuan, maka program akan menghasilkan rute dari kata asal ke kata tersebut. Jika tidak, program akan mengeksplorasi setiap kata tetangga (kata yang berbeda tepat satu huruf) dari kata yang sedang diambil. Jika kata tetangga belum dikunjungi, atur harga kata tersebut menjadi jumlah huruf berbeda dengan kata tujuan dan tambahkan ke *priority queue* sebagai simpul baru. Jika semua simpul telah selesai dieksplorasi dan rute tidak ditemukan, program akan mengembalikan *null*.

ANALISIS DAN IMPLEMENTASI ALGORITMA A*

Algoritma *A Star* (A*) merupakan algoritma pencarian rute yang selalu memilih solusi optimal lokal pada setiap langkah, yaitu simpul dengan harga terkecil. Dalam permainan *Word Ladder*, harga sebuah simpul tetangga bernilai panjang rute simpul sekarang dari kata asal ditambah jumlah huruf yang berbeda dengan simpul tujuan. Algoritma ini akan mengevaluasi setiap simpul atau kata dengan heuristik $f(n) = g(n) + h(n)$ di mana $f(n)$ merupakan estimasi total harga untuk mencapai kata tujuan dari kata asal, $g(n)$ merupakan panjang rute dari kata asal ke kata sekarang, dan $h(n)$ merupakan estimasi harga rute simpul sekarang ke simpul tujuan, yaitu jumlah huruf yang berbeda antara simpul sekarang dan simpul tujuan.

Algoritma A* yang digunakan dalam program ini bersifat *admissible*, karena algoritma yang digunakan tidak pernah *overestimate* harga sebenarnya ke simpul tujuan. Hal ini dikarenakan jumlah huruf yang berbeda antara suatu kata dengan kata tujuan tidak akan melebihi jumlah langkah minimal yang mungkin dari kata tersebut ke kata tujuan. Oleh karena itu, secara teoritis algoritma A* lebih efisien dibandingkan dengan algoritma UCS pada kasus *Word Ladder*. Dengan heuristik yang digunakan, algoritma A* dapat “menebak” langkah mana yang mungkin mengarah ke kata tujuan dengan harga terendah sehingga akan menjelajahi lebih sedikit simpul dibandingkan dengan algoritma UCS yang tidak mempertimbangkan estimasi harga ke simpul tujuan.

Untuk implementasi algoritma A*, akan digunakan sebuah *priority queue* untuk menyimpan kata berdasarkan panjang rute dari kata asal dan sebuah *set* untuk menyimpan kata yang telah dikunjungi. Kata asal akan menjadi kata yang pertama kali dimasukkan ke *priority queue*. Kemudian, akan dilakukan eksplorasi setiap kata pada *priority queue* sampai *priority queue* kosong. Program akan mengambil kata dengan harga terendah ($g(n)$ ditambah $h(n)$) dari *priority queue*. Jika kata yang diambil merupakan kata tujuan, maka program akan menghasilkan rute dari kata asal ke kata tersebut. Jika tidak, program akan mengeksplorasi setiap kata tetangga (kata yang

berbeda tepat satu huruf) dari kata yang sedang diambil. Jika kata tetangga belum dikunjungi, atur $g(n)$ kata tersebut menjadi harga kata sekarang ditambah satu dan atur $h(n)$ kata tersebut menjadi jumlah huruf berbeda dengan kata tujuan dan tambahkan ke *priority queue* sebagai simpul baru. Jika semua simpul telah selesai dieksplorasi dan rute tidak ditemukan, program akan mengembalikan *null*.

SOURCE CODE PROGRAM

Kelas SimpulKata

```
// Kelas Simpul
public class SimpulKata {
    // Atribut kelas
    private String kata;
    private int gn;
    private int hn;
    private SimpulKata leluhur;

    // Konstruktor
    public SimpulKata (String kata, int gn, int hn, SimpulKata leluhur) {
        this.kata = kata;
        this.gn = gn;
        this.hn = hn;
        this.leluhur = leluhur;
    }

    // Getter kata
    public String ambilKata() {
        return this.kata;
    }

    // Getter g(n)
    public int ambilgn() {
        return this.gn;
    }

    // Setter h(n)
    public int ambilhn() {
        return this.hn;
    }

    // Setter g(n)
    public void aturgn(int gnBaru) {
        this.gn = gnBaru;
    }
}
```

```

// Setter h(n)
public void aturhn(int hnBaru) {
    this.hn = hnBaru;
}

// Getter leluhur
public SimpulKata getLeluhur() {
    return this.leluhur;
}
}

```

Kelas SimpulKata memiliki empat atribut, yaitu atribut kata yang bertipe String, atribut gn yang bertipe integer, atribut hn yang bertipe integer, dan atribut leluhur yang bertipe SimpulKata. SimpulKata memiliki *user-defined constructor* serta metode *getter* dan *setter* untuk atribut yang diperlukan.

Kelas Penyelesai

```

// Import library
import java.util.*;

// Kelas Penyelesai
public class Penyelesai {
    // Atribut kelas
    private Set<String> kamus;

    // Fungsi untuk membuat rute dari kata asal ke kata tujuan
    // Menerima parameter kata tujuan dan mengembalikan rute yang dibuat
    // dalam bentuk list kata
    private List<String> buatRute (SimpulKata kataTujuan) {
        List<String> listKata = new ArrayList<>();
        SimpulKata kataSekarang = kataTujuan;
        // Buat rute ke kataTujuan dengan memanggil simpul leluhur secara
        // rekursif sampai kata asal
        while (kataSekarang != null) {
            listKata.add(0, kataSekarang.ambilKata());
            kataSekarang = kataSekarang.getLeluhur();
        }
        return listKata;
    }
}

```



```

    }

    // Fungsi untuk mencari semua simpul tetangga (kata yang berbeda satu
huruf)
    // Menerima parameter kata dan mengembalikan semua simpul tetangga
dalam bentuk list kata tetangga
    private List<String> cariTetangga (String kata) {
        List<String> listTetangga = new ArrayList<>();
        char[] arrayKarakterKata = kata.toCharArray();
        // Mencari semua kata lain yang berbeda satu huruf
        for (int i = 0; i < arrayKarakterKata.length; i++) {
            char karakterKata = arrayKarakterKata[i];
            // Mengganti setiap huruf pada kata dengan huruf lain dan
mengecek apakah sebuah kata lain terbentuk
            for (char karakterCek = 'a'; karakterCek <= 'z';
karakterCek++) {
                if (karakterCek != karakterKata) {
                    arrayKarakterKata[i] = karakterCek;
                    String kataTetangga = new String(arrayKarakterKata);
                    // Jika sebuah kata lain terbentuk
                    if (kamus.contains(kataTetangga)) {
                        // Tambahkan kata sebagai kata tetangga
                        listTetangga.add(kataTetangga);
                    }
                }
            }
            arrayKarakterKata[i] = karakterKata;
        }
        return listTetangga;
    }

    // Fungsi untuk menghitung heuristik h(n)
    // Menerima parameter kata dan kata tujuan serta mengembalikan nilai
h(n)
    private int hitunghn (String kata, String kataTujuan) {
        int hn = 0;
        for (int i = 0; i < kata.length(); i++) {
            if (kata.charAt(i) != kataTujuan.charAt(i)) {
                hn++;
            }
        }
    }

```

```

    }
    return hn;
}

// Konstruktor
public Penyelesai (Set<String> kamus) {
    this.kamus = kamus;
}

// Fungsi untuk mencari rute dari kata asal ke kata tujuan menggunakan
// algoritma Uniform Cost Search (UCS)
// Menerima parameter kata asal dan kata tujuan serta mengembalikan
// rute ke kata tujuan yang dibentuk fungsi buatRute dan jumlah kata
// dikunjungi
public Hasil cariRuteUCS(String kataAsal, String kataTujuan) {
    // Buat priority queue berdasarkan harga tiap kata
    Queue<SimpulKata> antrian = new
PriorityQueue<>(Comparator.comparingInt(SimpulKata
->
SimpulKata.ambilgn()));
    // Buat set untuk menyimpan kata yang telah dikunjungi
    Set<String> kataDikunjungi = new HashSet<>();
    // Inisialisasi jumlah kata dikunjungi
    int jumlahKataDikunjungi = 0;
    // Mulai penghitungan waktu eksekusi
    long waktuMulai = System.currentTimeMillis();
    // Tambahkan kata asal ke antrian
    antrian.offer(new SimpulKata(kataAsal, 0, 0, null));
    // Eksplorasi setiap kata pada antrian sampai antrian kosong
    while (!antrian.isEmpty()) {
        // Ambil kata dengan harga terendah dari antrian
        SimpulKata kataSekarang = antrian.poll();
        kataDikunjungi.add(kataSekarang.ambilKata());
        // Jika kata yang diambil merupakan kata tujuan
        if (kataSekarang.ambilKata().equals(kataTujuan)) {
            // Kembalikan rute dari kata asal ke kata sekarang, jumlah
            kata dikunjungi, dan waktu eksekusi
            List<String> rute = buatRute(kataSekarang);
            long waktuSelesai = System.currentTimeMillis();
            long waktuEksekusi = waktuSelesai - waktuMulai;

```

```

        Hasil hasil = new Hasil(rute, jumlahKataDikunjungi,
waktuEksekusi);

        return hasil;
    }

    // Cek setiap kata tetangga dari kata yang sedang diambil
    for (String kataTetangga :
cariTetangga(kataSekarang.ambilKata())) {
        // Jika kata tetangga belum dikunjungi
        if (!kataDikunjungi.contains(kataTetangga)) {
            // Atur harga kata tersebut menjadi harga kata
sekarang ditambah satu dan masukkan ke antrian sebagai simpul baru
            int gn = kataSekarang.ambilgn() + 1;
            antrian.offer(new SimpulKata(kataTetangga, gn, 0,
kataSekarang));

            // Tambah jumlah kata yang dikunjungi
            jumlahKataDikunjungi++;
        }
    }

    }

    // Kembalikan null jika rute tidak ditemukan
    return null;
}

// Fungsi untuk mencari rute dari kata asal ke kata tujuan menggunakan
algoritma Greedy Best First Search (GBFS)

// Menerima parameter kata asal dan kata tujuan serta mengembalikan
rute ke kata tujuan yang dibentuk fungsi buatRute dan jumlah kata
dikunjungi
public Hasil cariRuteGBFS (String kataAsal, String kataTujuan) {
    // Buat priority queue berdasarkan harga tiap kata
    Queue<SimpulKata> antrian = new
PriorityQueue<>(Comparator.comparingInt(SimpulKata
->
SimpulKata.ambilhn()));

    // Buat set untuk menyimpan kata yang telah dikunjungi
    Set<String> kataDikunjungi = new HashSet<>();
    // Inisialisasi jumlah kata dikunjungi
    int jumlahKataDikunjungi = 0;
    // Mulai penghitungan waktu eksekusi
    long waktuMulai = System.currentTimeMillis();
    // Tambahkan kata asal ke antrian

```

```

        antrian.offer(new SimpulKata(kataAsal, 0, hitunghn(kataAsal,
kataTujuan), null));
    // Eksplorasi setiap kata pada antrian sampai antrian kosong
    while (!antrian.isEmpty()) {
        // Ambil kata dengan harga terendah dari antrian
        SimpulKata kataSekarang = antrian.poll();
        kataDikunjungi.add(kataSekarang.ambilKata());
        // Jika kata yang diambil merupakan kata tujuan
        if (kataSekarang.ambilKata().equals(kataTujuan)) {
            // Kembalikan rute dari kata asal ke kata sekarang, jumlah
kata dikunjungi, dan waktu eksekusi
            List<String> rute = buatRute(kataSekarang);
            long waktuSelesai = System.currentTimeMillis();
            long waktuEksekusi = waktuSelesai - waktuMulai;
            Hasil hasil = new Hasil(rute, jumlahKataDikunjungi,
waktuEksekusi);
            return hasil;
        }
        // Cek setiap kata tetangga dari kata yang sedang diambil
        for (String kataTetangga :
cariTetangga(kataSekarang.ambilKata())) {
            // Jika kata tetangga belum dikunjungi
            if (!kataDikunjungi.contains(kataTetangga)) {
                // Hitung heuristik h(n) kata tersebut dan masukkan ke
antrian sebagai simpul baru
                int hn = hitunghn(kataTetangga, kataTujuan);
                antrian.offer(new SimpulKata(kataTetangga, 0, hn,
kataSekarang));

                // Tambah jumlah kata yang dikunjungi
                jumlahKataDikunjungi++;
            }
        }
    }

    // Kembalikan null jika rute tidak ditemukan
    return null;
}

// Fungsi untuk mencari rute dari kata asal ke kata tujuan menggunakan
algoritma A star (A*)

```

```

        // Menerima parameter kata asal dan kata tujuan serta mengembalikan
rute ke kata tujuan yang dibentuk fungsi buatRute dan jumlah kata
dikunjungi

        public Hasil cariRuteAStar (String kataAsal, String kataTujuan) {
            // Buat priority queue berdasarkan harga tiap kata
                Queue<SimpulKata> antrian = new
PriorityQueue<>(Comparator.comparingInt(SimpulKata
->
(SimpulKata.ambilgn() + SimpulKata.ambilhn())));
            // Buat set untuk menyimpan kata yang telah dikunjungi
Set<String> kataDikunjungi = new HashSet<>();
            // Inisialisasi jumlah kata dikunjungi
int jumlahKataDikunjungi = 0;
            // Mulai penghitungan waktu eksekusi
long waktuMulai = System.currentTimeMillis();
            // Tambahkan kata asal ke antrian
                antrian.offer(new SimpulKata(kataAsal, 0, hitunghn(kataAsal,
kataTujuan), null));
            // Eksplorasi setiap kata pada antrian sampai antrian kosong
while (!antrian.isEmpty()) {
                // Ambil kata dengan harga terendah dari antrian
SimpulKata kataSekarang = antrian.poll();
                kataDikunjungi.add(kataSekarang.ambilKata());
                // Jika kata yang diambil merupakan kata tujuan
if (kataSekarang.ambilKata().equals(kataTujuan)) {
                    // Kembalikan rute dari kata asal ke kata sekarang, jumlah
kata dikunjungi, dan waktu eksekusi
List<String> rute = buatRute(kataSekarang);
                    long waktuSelesai = System.currentTimeMillis();
                    long waktuEksekusi = waktuSelesai - waktuMulai;
                    Hasil hasil = new Hasil(rute, jumlahKataDikunjungi,
waktuEksekusi);
                    return hasil;
                }
                // Cek setiap kata tetangga dari kata yang sedang diambil
for (String kataTetangga :
cariTetangga(kataSekarang.ambilKata())) {
                    // Jika kata tetangga belum dikunjungi
if (!kataDikunjungi.contains(kataTetangga)) {

```

```

        // Hitung harga kata tersebut (g(n) yang bernilai g(n)
kata sekarang ditambah satu, ditambah h(n)) dan masukkan ke antrian
sebagai simpul baru

        int gn = kataSekarang.ambilgn() + 1;
        int hn = hitunghn(kataTetangga, kataTujuan);
        antrian.offer(new SimpulKata(kataTetangga, gn, hn,
kataSekarang));

        // Tambah jumlah kata yang dikunjungi
        jumlahKataDikunjungi++;
    }
}

// Kembalikan null jika rute tidak ditemukan
return null;
}
}

```

Kelas Penyelesai memiliki satu atribut, yaitu kamus yang bertipe *set of String*. Kelas ini berisi metode-metode untuk melakukan pencarian rute menggunakan algoritma UCS, GBFS, dan A*. Ketiga algoritma bekerja dengan dua metode pembantu, yaitu metode *buatRute* dan metode *cariTetangga*. Metode *buatRute* menerima parameter simpul kata tujuan dan akan mengembalikan rute dari kata asal ke kata tujuan dengan secara rekursif memanggil leluhur dimulai dari kata tujuan sampai kata asal yang ditandai dengan tidak adanya simpul leluhur. Metode *cariTetangga* akan menerima parameter *String* kata dan mengembalikan semua kata tetangga, yaitu kata yang berbeda tepat satu huruf dengan kata pada parameter. Metode ini akan secara *brute force* mengganti setiap huruf pada kata parameter dengan huruf lainnya dan mengecek apakah terbentuk kata lainnya yang terdapat pada kamus. Khusus untuk algoritma GBFS dan A*, terdapat satu lagi metode pembantu yaitu metode *hitunghn* yang digunakan untuk menghitung heuristik $h(n)$, yaitu jumlah huruf yang berbeda antara suatu kata dengan kata tujuan.

Metode untuk ketiga algoritma memiliki cara kerja yang sama, hanya terdapat perbedaan kecil di penghitungan harga untuk setiap simpul sesuai dengan heuristik yang digunakan masing-masing algoritma. Metode cariRute untuk masing-masing algoritma akan pertama menginisialisasikan *priority queue* berdasarkan harga simpul dan *set of String* untuk menyimpan semua kata yang telah dikunjungi. Kata asal akan menjadi simpul pertama yang dimasukkan ke *priority queue*. Kemudian, akan dilakukan eksplorasi setiap kata pada *priority queue* sampai *priority queue* kosong. Metode akan mengambil kata dengan harga terendah dari *priority queue*. Jika kata yang diambil merupakan kata tujuan, metode akan memanggil metode buatRute untuk menghasilkan rute dari kata asal ke kata sekarang. Jika tidak, metode akan mengeksplorasi setiap kata tetangga dari kata yang sedang diambil. Jika kata tetangga belum dikunjungi, kata tetangga akan ditambah ke *priority queue* sebagai simpul baru dan baik $g(n)$ maupun $h(n)$ kata tersebut diatur menurut heuristik masing-masing algoritma. Jika semua kata telah dikunjungi dan rute tidak ditemukan, metode akan mengembalikan *null*.

Kelas Hasil

```
// Import library
import java.util.*;

// Kelas hasil
public class Hasil {
    // Atribut kelas
    private List<String> rute;
    private int jumlahKataDikunjungi;
    private long waktuEksekusi;

    // Konstruktor
    public Hasil (List<String> rute, int jumlahKataDikunjungi, long waktuEksekusi) {
        this.rute = rute;
        this.jumlahKataDikunjungi = jumlahKataDikunjungi;
        this.waktuEksekusi = waktuEksekusi;
    }

    // Getter rute
    public List<String> ambilRute() {
        return this.rute;
    }

    // Getter jumlah kata dikunjungi
    public int ambilJumlahKataDikunjungi() {
        return this.jumlahKataDikunjungi;
    }

    // Getter waktu eksekusi
    public long ambilWaktuEksekusi() {
        return this.waktuEksekusi;
    }
}
```

Kelas Hasil memiliki tiga atribut, yaitu atribut rute yang bertipe *list of String*, atribut jumlahKataDikunjungi yang bertipe integer, dan atribut waktuEksekusi yang bertipe

long. Hasil memiliki *user-defined constructor* serta metode *getter* dan *setter* untuk atribut yang diperlukan. Kelas Hasil digunakan untuk menyimpan hasil pencarian rute.

Kelas Main

```
// Import library
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.FileReader;
import java.io.IOException;

// Kelas Main
public class Main {
    // Atribut kelas
    private JFrame frame;
    private JTextField kataAsalField;
    private JTextField kataTujuanField;
    private JComboBox<String> algoritmaComboBox;
    private JTextArea infoArea;
    private Set<String> kamus;

    // Konstruktor
    public Main(Set<String> kamus) {
        this.kamus = kamus;
        frame = new JFrame("Word Ladder Solver");
        frame.setSize(400, 700);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new GridLayout(2, 1));
        JPanel mainPanel = new JPanel();
        mainPanel.setLayout(new GridLayout(4, 2));
        JLabel kataAsalLabel = new JLabel("Kata Asal:");
        kataAsalField = new JTextField();
        kataAsalField.setPreferredSize(new Dimension(200, 20));
        JLabel kataTujuanLabel = new JLabel("Kata Tujuan:");
        kataTujuanField = new JTextField();
        kataTujuanField.setPreferredSize(new Dimension(200, 20));
        JLabel algoritmaLabel = new JLabel("Pilih Algoritma:");
        String[] pilihanAlgoritma = {"Uniform Cost Search (UCS)", "Greedy
Best First Search (GBFS)", "A Star (A*)"};
        algoritmaComboBox = new JComboBox<>(pilihanAlgoritma);
        algoritmaComboBox.setPreferredSize(new Dimension(200, 20));
        mainPanel.add(kataAsalLabel);
```

```

        mainPanel.add(kataAsalField);
        mainPanel.add(kataTujuanLabel);
        mainPanel.add(kataTujuanField);
        mainPanel.add(algoritmaLabel);
        mainPanel.add(algoritmaComboBox);
        frame.add(mainPanel);

        JPanel infoPanel = new JPanel();
        infoPanel.setLayout(new BorderLayout());
        JLabel infoLabel = new JLabel("Informasi:");
        infoArea = new JTextArea();
        infoArea.setLineWrap(true);
        JScrollPane scrollPane = new JScrollPane(infoArea);

scrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);

        scrollPane.setPreferredSize(new Dimension(300, 650));
        infoPanel.add(infoLabel, BorderLayout.NORTH);
        infoPanel.add(scrollPane, BorderLayout.CENTER);
        frame.add(infoPanel);

        JButton cariButton = new JButton("Cari Rute");
        cariButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                // Menerima input dari field
                String kataAsal =
kataAsalField.getText().trim().toLowerCase();
                String kataTujuan =
kataTujuanField.getText().trim().toLowerCase();
                String algoritma = (String)
algoritmaComboBox.getSelectedItem();

                // Validasi kata asal dan kata tujuan
                if (!kataValid(kataAsal) || !kataValid(kataTujuan)) {
                    JOptionPane.showMessageDialog(frame, "Kata asal atau
kata tujuan tidak valid!", "Error", JOptionPane.ERROR_MESSAGE);
                    return;
                }

                // Validasi panjang kata asal dan kata tujuan
                if (kataAsal.length() != kataTujuan.length()) {

```

```

        JOptionPane.showMessageDialog(frame, "Panjang kata
asal dan kata tujuan harus sama!", "Error", JOptionPane.ERROR_MESSAGE);
        return;
    }

    if (kataValid(kataAsal) && kataValid(kataTujuan) &&
(kataAsal.length() == kataTujuan.length())) {
        Penyelesai penyelesai = new Penyelesai(kamus);
        // Pilihan algoritma UCS
        if (algoritma.equals("Uniform Cost Search (UCS)")) {
            Hasil hasil = penyelesai.cariRuteUCS(kataAsal,
kataTujuan);

            showInfo(hasil);
        }
        // Pilihan algoritma GBFS
        else if (algoritma.equals("Greedy Best First Search
(GBFS)")) {
            Hasil hasil = penyelesai.cariRuteGBFS(kataAsal,
kataTujuan);

            showInfo(hasil);
        }
        // Pilihan algoritma A*
        else if (algoritma.equals("A Star (A*)")) {
            Hasil hasil = penyelesai.cariRuteAStar(kataAsal,
kataTujuan);

            showInfo(hasil);
        }
    }
}

});
mainPanel.add(cariButton);
frame.add(mainPanel);
frame.setVisible(true);
}

// Fungsi untuk mengeluarkan hasil di GUI
// Menerima parameter hasil
private void showInfo(Hasil hasil) {
    // Jika rute ditemukan
    if (hasil != null) {

```

```

        // Mengeluarkan informasi pada GUI
        java.util.List<String> rute = hasil.ambilRute();
        int jumlahKataDikunjungi = hasil.ambilJumlahKataDikunjungi();
        int panjangRute = rute.size() - 1;
        long waktuEksekusi = hasil.ambilWaktuEksekusi();

        StringBuilder sb = new StringBuilder();
        sb.append("Jumlah kata dikunjungi:
").append(jumlahKataDikunjungi).append("\n");
        sb.append("Panjang rute: ").append(panjangRute).append("\n");
        sb.append("Waktu eksekusi: ").append(waktuEksekusi).append("
ms\n");

        sb.append("Rute:\n");
        for (String kata : rute) {
            sb.append(kata).append("\n");
        }
        infoArea.setText(sb.toString());
    }

    // Jika rute tidak ditemukan
    else {
        infoArea.setText("Rute tidak ditemukan");
    }
}

// Fungsi untuk mengecek validitas kata
// Menerima parameter kata dan mengembalikan boolean validitas kata
private boolean kataValid(String kata) {
    return kamus.contains(kata);
}

// Fungsi main
public static void main(String[] args) {
    Set<String> kamus = new HashSet<>();
    String lokasiFile = "kamus.txt";
    // Olah file kamus.txt menjadi kamus
    try (Scanner scanner = new Scanner(new FileReader(lokasiFile))) {
        while (scanner.hasNextLine()) {
            String kataKamus =
scanner.nextLine().trim().toLowerCase();
            kamus.add(kataKamus);

```

```

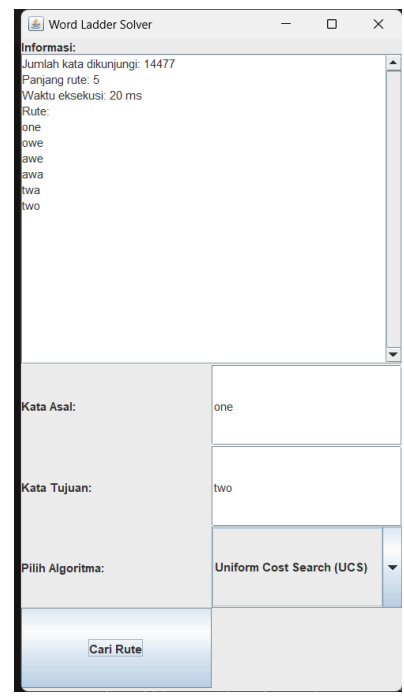
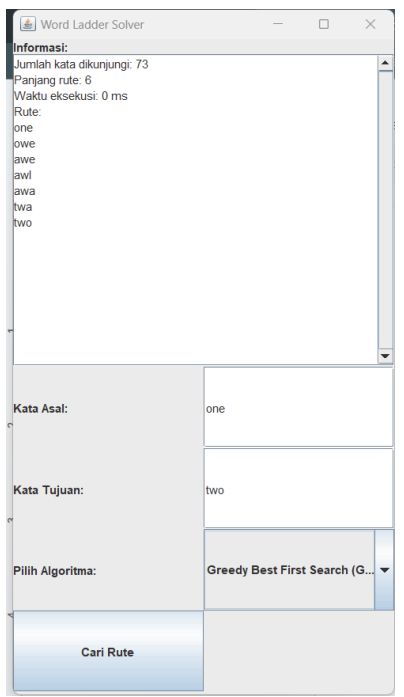
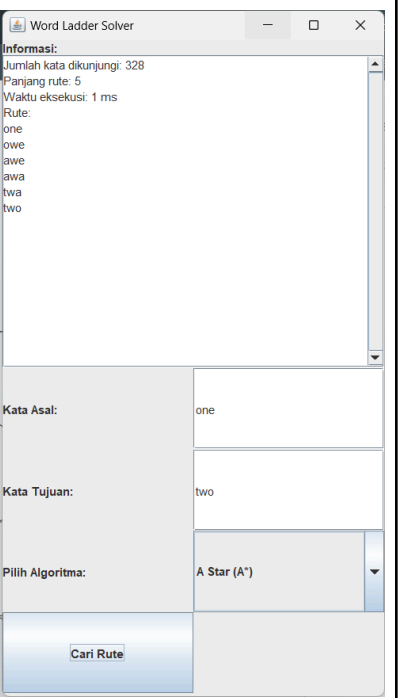
    }
} catch (IOException e) {
    e.printStackTrace();
    System.exit(1);
}
// Menjalankan GUI
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        new Main(kamus);
    }
});
}
}

```

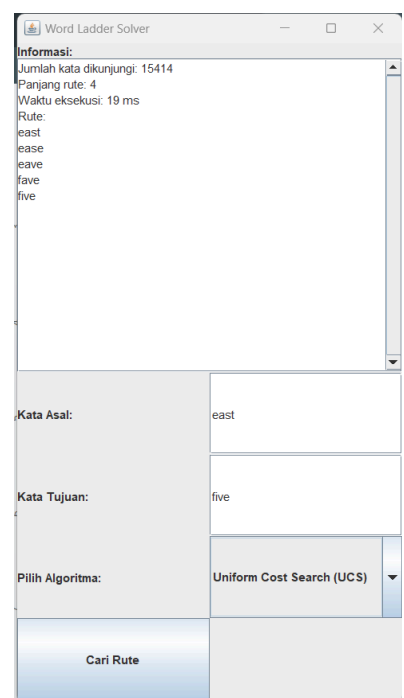
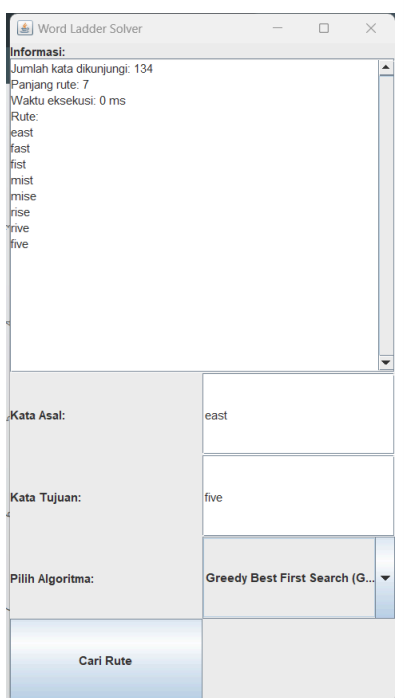
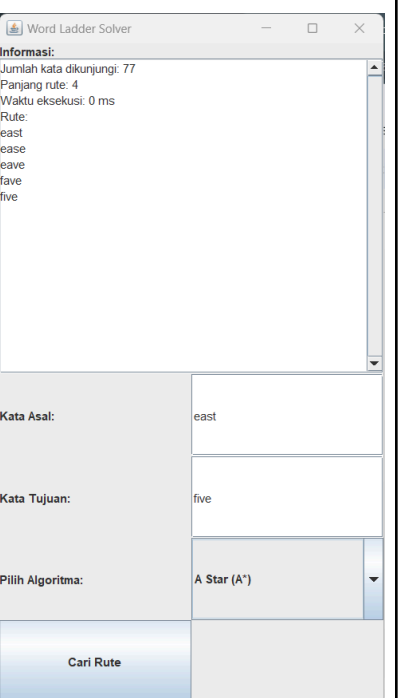
Kelas *Main* merupakan kelas utama yang juga mencakup GUI program dengan serangkaian atribut untuk isi GUI dan atribut kamus bertipe *set of String*. Kelas ini mempunyai satu buah *user-defined constructor*. Kelas ini akan menerima masukan dari pengguna berupa kata asal, kata tujuan, dan pilihan algoritma lalu menampilkan hasil pencarian rute. Kelas ini juga memiliki mekanisme validasi masukan pengguna.

TESTING

“one” ke “two”

		
--	---	--

“east” ke “five”

		
---	--	---

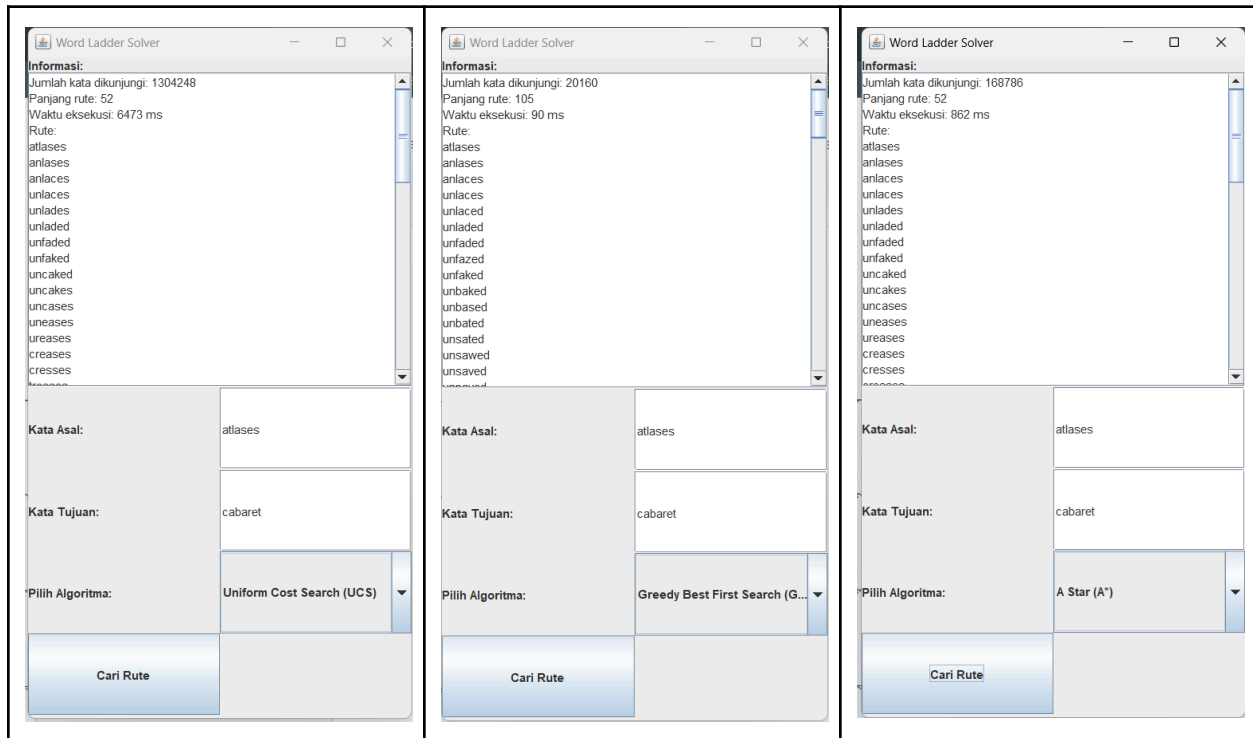
“wrong” ke “right”

Word Ladder Solver	Word Ladder Solver	Word Ladder Solver
Informasi: Jumlah kata dikunjungi: 111254 Panjang rute: 13 Waktu eksekusi: 737 ms Rute: wrong prong prone phone phons pions ilons linns lines sines sinhs sighs sight right	Informasi: Jumlah kata dikunjungi: 4890 Panjang rute: 56 Waktu eksekusi: 12 ms Rute: wrong prong prone prove prose prost prest drest wrest wrist grist grift graft grant grunt	Informasi: Jumlah kata dikunjungi: 27126 Panjang rute: 13 Waktu eksekusi: 58 ms Rute: wrong prong prone phone phons pions ilons linns lines sines sinhs sighs sight right
Kata Asal: wrong	Kata Asal: wrong	Kata Asal: wrong
Kata Tujuan: right	Kata Tujuan: right	Kata Tujuan: right
Pilih Algoritma: Uniform Cost Search (UCS)	Pilih Algoritma: Greedy Best First Search (G...)	Pilih Algoritma: A Star (A*)
Cari Rute	Cari Rute	Cari Rute

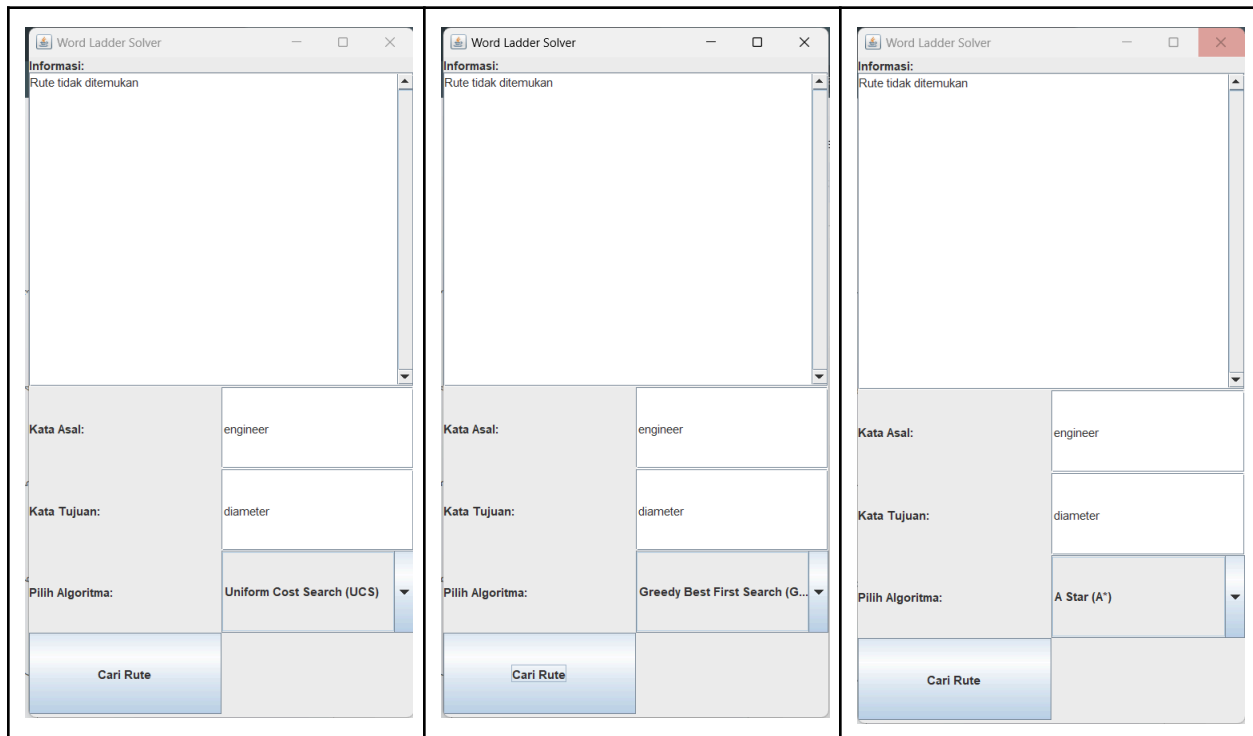
“greedy” ke “normal”

Word Ladder Solver	Word Ladder Solver	Word Ladder Solver
Informasi: Jumlah kata dikunjungi: 137930 Panjang rute: 19 Waktu eksekusi: 1207 ms Rute: greedy greeds breeds bleeds blends blinds blinks blanks flanks flanes flames flamer foamer former forger	Informasi: Jumlah kata dikunjungi: 1011 Panjang rute: 43 Waktu eksekusi: 3 ms Rute: greedy greeds greets greats groats groans groins grains grainy granny cranny cranky cranks cranes craned	Informasi: Jumlah kata dikunjungi: 36428 Panjang rute: 19 Waktu eksekusi: 131 ms Rute: greedy greeds creeds creeks cheeks checks chicks chinks chines chined coined corned corked forked forged
Kata Asal: greedy	Kata Asal: greedy	Kata Asal: greedy
Kata Tujuan: normal	Kata Tujuan: normal	Kata Tujuan: normal
Pilih Algoritma: Uniform Cost Search (UCS)	Pilih Algoritma: Greedy Best First Search (G...)	Pilih Algoritma: A Star (A*)
Cari Rute	Cari Rute	Cari Rute

“atlases” ke “cabaret”



“engineer” ke “diameter”



ANALISIS PERBANDINGAN

Dilihat dari aspek optimalitas, algoritma GBFS dan A* lebih optimal dari algoritma UCS, di mana algoritma GBFS cenderung lebih optimal dari A*. Hal ini dikarenakan algoritma GBFS dan A* memiliki heuristik $h(n)$ yang bisa “menebak” langkah yang akan menghasilkan rute terpendek ke simpul tujuan, sehingga memperkecil jumlah simpul yang harus dikunjungi. Walau demikian, heuristik ini menyebabkan algoritma GBFS dan A* tidak menjamin dapat menghasilkan solusi optimal global. Perbedaan jumlah simpul yang dikunjungi masing-masing algoritma dapat dilihat pada tangkapan layar hasil [testing](#) di bagian sebelumnya.

Dilihat dari aspek waktu eksekusi, algoritma GBFS dan A* cenderung membutuhkan waktu yang jauh lebih kecil dari algoritma UCS, di mana algoritma GBFS cenderung membutuhkan waktu lebih kecil dari A*. Hal ini juga disebabkan oleh perbedaan heuristik yang memungkinkan algoritma GBFS dan A* untuk mengunjungi lebih sedikit simpul dibandingkan dengan algoritma UCS. Perbedaan ini dapat terlihat pada tangkapan layar hasil [testing](#) di bagian sebelumnya.

Dilihat dari aspek memori yang dibutuhkan, algoritma GBFS dan A* cenderung membutuhkan memori yang jauh lebih kecil dibandingkan dengan algoritma UCS, di mana algoritma GBFS cenderung membutuhkan memori lebih kecil dari A*. Hal ini juga dikarenakan oleh perbedaan heuristik yang memungkinkan algoritma GBFS dan A* untuk mengunjungi lebih sedikit simpul dibandingkan dengan algoritma UCS. Perbedaan ini dapat terlihat pada tangkapan layar hasil [testing](#) di bagian sebelumnya.

IMPLEMENTASI BONUS

GUI pada program ini dibuat menggunakan *library* Swing. Pertama, jendela utama aplikasi dibuat menggunakan JFrame dengan ukuran awal 400 * 700 px. Kemudian, digunakan GridLayout dengan 2 baris dan 1 kolom untuk membagi jendela utama. Bagian atas jendela berisi mainPanel dengan GridLayout 4 baris dan 2 kolom. Di panel ini, terdapat *field* dan label untuk masukan kata asal, kata tujuan, serta *dropdown* untuk memilih jenis algoritma. Selain itu, terdapat tombol Cari Rute di bagian bawah jendela. Bagian bawah jendela berisi infoPanel dengan BorderLayout. Di panel ini, terdapat label Informasi di bagian atas dan JTextArea yang digunakan untuk menampilkan informasi hasil pencarian rute. JTextArea dibungkus dengan JScrollPane agar dapat di-*scroll* jika teks isi melebihi ukuran area yang ditampilkan.

Setelah pengguna memberi masukan dan menekan tombol Cari Rute, program akan memeriksa input, melakukan validasi, kemudian menampilkan pesan peringatan jika masukan tidak valid atau memanggil metode showInfo untuk menampilkan informasi hasil pencarian rute di JTextArea. Jika rute berhasil ditemukan, akan ditampilkan jumlah kata yang dikunjungi, panjang rute, waktu eksekusi, dan kata-kata yang membentuk rute tersebut. Jika tidak, akan ditampilkan pesan bahwa rute tidak ditemukan. Semua logika GUI, seperti validasi input dan pencarian rute, dijalankan dalam actionPerformed dari tombol Cari Rute. Pada fungsi *main*, GUI akan dijalankan menggunakan SwingUtilites.invokeLater.

LAMPIRAN

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

PRANALA

https://github.com/ChristopherBrian/Tucil3_13522106