LAPORAN TUGAS KECIL 03

IF2211 STRATEGI ALGORITMA

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

Disusun oleh:

Mohammad Andhika Fadillah K-03 13522128

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

DAFTAR ISI

BAB 1

DESKRIPSI MASALAH

Word ladder (juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragrams, laddergrams, atau word golf) adalah salah satu permainan kata yang terkenal bagi seluruh kalangan. Word ladder ditemukan oleh Lewis Carroll, seorang penulis dan matematikawan, pada tahun 1877. Pada permainan ini, pemain diberikan dua kata yang disebut sebagai start word dan end word. Untuk memenangkan permainan, pemain harus menemukan rantai kata yang dapat menghubungkan antara start word dan end word. Banyaknya huruf pada start word dan end word selalu sama. Tiap kata yang berdekatan dalam rantai kata tersebut hanya boleh berbeda satu huruf saja. Pada permainan ini, diharapkan solusi optimal, yaitu solusi yang meminimalkan banyaknya kata yang dimasukkan pada rantai kata. Berikut adalah ilustrasi serta aturan permainan. Gambar 1. Ilustrasi dan Peraturan Permainan Word Ladder

BAB 2
TEORI SINGKAT

2.1 Algoritma Uniform Cost Search

UCS menggunakan struktur data graf. Graf adalah representasi visual dari objek dan relasi di antara mereka. Graf terdiri dari simpul (*node*) yang mewakili objek, dan sisi (*edge*) yang mewakili relasi antara objek tersebut. Setiap sisi pada graf memiliki bobot (*weight*) yang menunjukkan biaya atau jarak antara dua simpul yang terhubung. Bobot bisa berupa angka real atau integer, dan bisa mewakili berbagai jenis informasi seperti jarak fisik, waktu tempuh, atau biaya. UCS memerlukan node awal (*start node*) dan node tujuan (*goal node*) untuk memulai pencarian jalur terpendek. Node awal adalah simpul dari mana pencarian dimulai, sedangkan node tujuan adalah simpul yang ingin dicapai.

UCS menggunakan *explored set* (set yang telah dieksplorasi) untuk melacak simpul-simpul yang akan dieksplorasi dan yang telah dieksplorasi.Dalam prosesnya, UCS juga menghitung biaya akumulatif (cost-to-come) dari jalur yang telah dieksplorasi. Biaya akumulatif ini digunakan untuk membandingkan jalur-jalur yang berbeda dan memilih jalur dengan biaya terendah. Biaya ini lalu dimasukkan ke dalam *priority queue* untuk mengurutkan simpul-simpul berdasarkan biaya akumulatif jalur yang telah dieksplorasi. Simpul dengan biaya akumulatif yang lebih rendah diberikan prioritas lebih tinggi untuk dieksplorasi lebih dahulu.

2.2 Algoritma Greedy Best First Search

Algoritma Greedy Best-First Search (GBFS) merupakan metode pencarian yang berfokus pada pengambilan keputusan simpul berikutnya berdasarkan nilai heuristik terbaik, yaitu perkiraan jarak simpul tersebut ke tujuan. Dalam setiap langkah pencariannya, GBFS memprioritaskan simpul dengan nilai heuristik paling rendah atau yang paling mendekati tujuan, tanpa mempertimbangkan kemungkinan solusi di masa depan. Proses ini berlangsung secara iteratif, dimulai dari simpul awal, dan mengarah ke simpul-simpul tetangga yang memiliki estimasi jarak terkecil ke tujuan. Meskipun GBFS efisien dalam hal kinerja komputasi, namun kelemahannya terletak pada potensi terjebaknya dalam solusi lokal optimum jika nilai heuristik tidak memadai. GBFS umumnya digunakan dalam aplikasi yang membutuhkan pencarian solusi optimal di ruang pencarian yang besar, seperti navigasi robot, perencanaan jadwal, dan pemrosesan bahasa alami, di mana ketersediaan informasi yang cepat dan efisien menjadi kunci utama.

2.3 Algoritma *A**

Algoritma A* (A-star) adalah algoritma pencarian informasi yang digunakan untuk mencari jalur terpendek dalam graf berbobot. A* memproses struktur data graf, yang terdiri dari simpul (*node*) yang mewakili objek, dan sisi (*edge*) yang mewakili relasi antara objek tersebut. Setiap sisi pada graf memiliki bobot (*weight*) yang menunjukkan biaya atau jarak antara dua simpul yang terhubung. A* memerlukan node awal (*start node*) dan node tujuan *(goal node*) untuk memulai pencarian jalur terpendek

A* menggunakan fungsi heuristik untuk memperkirakan biaya tersisa atau jarak yang tersisa (*heuristic cost*) dari simpul saat ini ke node tujuan. Fungsi heuristik ini membantu A* dalam memilih simpul yang paling berpotensi mengarah ke jalur terpendek, sehingga bisa mempercepat pencarian. A* menghitung biaya akumulatif (cost-to-come) dari jalur yang telah dieksplorasi, ditambah dengan nilai heuristik, untuk membandingkan jalur-jalur yang berbeda dan memilih jalur dengan biaya terendah dan heuristik terbaik. Sama

seperti UCS, A* juga menggunakan *explored set* (set yang telah dieksplorasi) dan *priority queue* untuk melacak simpul-simpul yang akan dieksplorasi dan yang telah dieksplorasi, serta untuk mengurutkan simpul-simpul pada graf berdasarkan biaya akumulatif dan nilai heuristik. Simpul dengan nilai heuristik dan biaya akumulatif yang lebih rendah diberikan prioritas lebih tinggi untuk dieksplorasi lebih dahulu.

2.4 Java Swing

Java Swing merupakan toolkit pengembangan antarmuka grafis (GUI) untuk aplikasi Java yang memudahkan pembuatan komponen-komponen GUI seperti tombol, kotak teks, tabel, dan sebagainya. Swing menggunakan pola desain Model-View-Controller (MVC) yang memisahkan tata letak, logika bisnis, dan tampilan sehingga memudahkan pengembangan dan perawatan aplikasi. Dengan pendekatan event-driven, Swing menangani interaksi pengguna seperti klik tombol atau input teks dengan event handling, memungkinkan respons dinamis terhadap tindakan pengguna. Selain itu, Swing juga menyediakan layout managers untuk mengatur tata letak komponen GUI dengan fleksibilitas, serta mendukung tema yang dapat disesuaikan untuk memberikan tampilan yang konsisten dan menarik. Keunggulan lainnya adalah portabilitas tinggi, artinya aplikasi yang dibuat dengan Swing dapat dijalankan di berbagai platform dengan JVM yang sesuai.

BAB 3

ANALISIS & IMPLEMENTASI PROGRAM

3.1 Definisi dari f(n) dan g(n)

3.1.1 Uniform Cost Search (UCS)

- f(n) adalah biaya total dari node mulai hingga node n.
- g(n) adalah biaya dari root hingga node ke n.

Dalam UCS, kita mempertimbangkan biaya dari node mulai hingga node saat ini (dinyatakan sebagai g(n)) untuk menentukan urutan ekspansi node-node berikutnya.

3.1.2 Greedy Best First Search (GBFS)

- f(n) adalah biaya dari node n sampai ke tujuan atau nilai heuristik dari node n.
- g(n) tidak digunakan dalam GBFS.

GBFS hanya mempertimbangkan nilai heuristik dari node untuk memilih node yang akan diekspansi selanjutnya. Ini cenderung memilih node yang mendekati tujuan secara heuristik, tanpa memperhitungkan biaya sebenarnya dari node mulai hingga node tersebut.

3.1.3 A*

- f(n) adalah biaya total dari node mulai hingga node n ditambah nilai heuristik dari node n sampai ke tujuan.
- g(n) adalah biaya dari node mulai hingga node n.

Dalam A*, kita mempertimbangkan biaya sebenarnya dari node mulai hingga node saat ini (dinyatakan sebagai g(n)), serta nilai heuristik dari node tersebut (dinyatakan sebagai h(n)) untuk menentukan urutan ekspansi node-node berikutnya.

3.2 Heuristik pada algoritma A*

Heuristik yang digunakan pada A* dikatakan admissible jika nilai heuristik tidak pernah melebihi biaya sebenarnya dari node saat ini hingga tujuan. Dalam kasus ini, heuristik yang digunakan adalah jumlah karakter yang berbeda antara node saat ini dan tujuan. Heuristik ini admissible karena tidak pernah lebih besar dari biaya sebenarnya yang dibutuhkan untuk mencapai tujuan.

3.3 Algoritma UCS dan BFS (dalam kasus word ladder)

Algoritma UCS (Uniform Cost Search) dan BFS (Breadth-First Search) dapat menghasilkan path yang sama dalam kasus word ladder jika diimplementasikan dengan benar. Namun, terdapat perbedaan fundamental dalam cara kedua algoritma ini melakukan pencarian.

BFS bekerja dengan cara mengeksplorasi semua node pada level yang sama sebelum bergerak ke level selanjutnya. Dalam kasus word ladder, ini berarti BFS akan mulai dengan kata awal, lalu mengeksplorasi semua kata yang memiliki jarak satu langkah dari kata awal, kemudian kata-kata yang memiliki jarak dua

langkah, dan seterusnya, hingga menemukan kata tujuan. Ini menghasilkan path yang optimal dalam hal jumlah langkah, karena BFS menjamin menemukan solusi terdekat terlebih dahulu.

Di sisi lain, UCS bekerja dengan cara mempertimbangkan biaya aktual dari node awal hingga node saat ini. UCS memprioritaskan node yang memiliki biaya lebih rendah terlebih dahulu. Dalam permasalahan word ladder, ini berarti UCS akan mengeksplorasi node-node dengan biaya yang semakin rendah, tanpa memperhatikan level atau jarak dari node awal. Jika implementasi antrian prioritas pada UCS mempertimbangkan biaya secara benar, maka algoritma ini juga dapat menghasilkan path yang optimal.

3.4 Algoritma A* dan Algoritma UCS (dalam kasus word ladder)

Secara teoritis, perbedaan utama antara algoritma A* dan UCS terletak pada penggunaan nilai heuristik. Algoritma A* menggunakan nilai heuristik yang menggambarkan perkiraan jarak atau biaya yang tersisa untuk mencapai tujuan dari setiap node, sementara UCS hanya mempertimbangkan biaya sebenarnya dari node mulai hingga node saat ini. Dengan memanfaatkan nilai heuristik, A* dapat membuat estimasi yang lebih baik tentang jalur terpendek menuju tujuan, karena ia menggabungkan biaya sebenarnya yang telah ditempuh dengan perkiraan biaya yang tersisa. Hal ini memungkinkan A* untuk menghindari mengeksplorasi jalur yang tidak produktif, mempercepat pencarian, dan mengurangi jumlah node yang perlu dieksplorasi secara keseluruhan.

Dalam kasus word ladder, di mana kita mencari jalur terpendek antara dua kata dengan mengubah satu huruf pada setiap langkah, A* dapat memilih node yang memiliki nilai f(n) (biaya total dari node mulai hingga node saat ini ditambah nilai heuristik) yang lebih kecil, yang menunjukkan jalur yang lebih potensial untuk mencapai tujuan dalam jumlah langkah yang lebih sedikit. Namun, efisiensi relatif dari A* dibandingkan dengan UCS juga tergantung pada kualitas nilai heuristik yang digunakan. Jika nilai heuristik cukup akurat dan admissible (tidak pernah melebihi biaya sebenarnya), A* cenderung lebih efisien karena mampu "melihat" lebih jauh ke depan dalam pencarian solusi terpendek.

3.5 Apakah algoritma Greedy Best First Search menjamin solusi optimal untuk persoalan word ladder?

Dalam konteks persoalan word ladder, di mana kita mencari jalur terpendek antara dua kata dengan mengubah satu huruf pada setiap langkah, algoritma GBFS mungkin tidak menghasilkan solusi optimal. Hal ini karena GBFS hanya mempertimbangkan nilai heuristik dari node tanpa memperhitungkan biaya sebenarnya (jumlah langkah yang dibutuhkan) untuk mencapai tujuan. Meskipun GBFS cenderung memilih node yang terlihat mendekati tujuan berdasarkan nilai heuristiknya, itu tidak menjamin bahwa node tersebut merupakan pilihan terbaik dalam mencapai solusi terpendek.

3.6 Implementasi Bonus

Program dapat berjalan dengan GUI (Graphical User Interface) menggunakan *toolkit Java Swing*

GUI memiliki komponen-komponen sebagai berikut:

- JTextField startWordField dan endWordField: Untuk memasukkan kata awal dan kata akhir dari Word Ladder.
- JComboBox algorithmComboBox: Untuk memilih algoritma yang digunakan, yaitu Uniform Cost Search, Greedy Best First Search, atau A*.
- JButton solveButton: Untuk memulai proses pencarian solusi Word Ladder berdasarkan input yang diberikan.

- JTextArea resultArea: Untuk menampilkan hasil pencarian, termasuk jalur yang ditemukan, jumlah node yang dikunjungi, dan waktu eksekusi.

GUI ini juga menggunakan sebuah custom JPanel (BackgroundPanel) dengan background image. Panel ini digunakan untuk menampilkan komponen-komponen GUI dengan latar belakang gambar. Proses pencarian jalur Word Ladder dilakukan saat tombol "Solve" ditekan. Algoritma pencarian yang digunakan (UCS, GBFS, atau A*) tergantung pada pilihan yang dibuat di algorithmComboBox.

3.7 Source Code Program

3.7.1 UCS.java

```
import java.util.*;

public class UCS {

   private static class Node {
      String word;
      Node parent;
      int cost;

      Node(String word, Node parent, int cost) {
         this.word = word;
         this.parent = parent;
         this.cost = cost;
      }
   }

   public static List<String> findPath(String start, String end, Set<String> words) {
      Queue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(node -> node.cost));
      Set<String> visited = new HashSet<>();
      queue.add(new Node(start, null, 0));
      visited.add(start);

      int nodesVisited = 0; // Add a counter for nodes visited

      while (!queue.isEmpty()) {
         Node current = queue.poll();
         String currentWord = current.word;

         nodesVisited++; // Increment the counter each time a node is visited

         if (currentWord.equals(end)) {
            List<String> path = new ArrayList<>();
            while (current != null) {
               path.add(current.word);
               current = current.parent;
            }
            Collections.reverse(path);

            path.add("Nodes visited: " + nodesVisited); // Add the total nodes visited to the path

            return path;
```

```
            }

            StringBuilder wordBuilder = new StringBuilder(currentWord);
            for (int i = 0; i < wordBuilder.length(); i++) {
                char originalChar = wordBuilder.charAt(i);
                for (char c = 'a'; c <= 'z'; c++) {
                    if (c != originalChar) {
                        wordBuilder.setCharAt(i, c);
                        String newWord = wordBuilder.toString();
                        if (words.contains(newWord) && !visited.contains(newWord)) {
                            int cost = current.cost + 1;
                            queue.add(new Node(newWord, current, cost));
                            visited.add(newWord);
                        }
                    }
                }
                wordBuilder.setCharAt(i, originalChar);
            }
        }

        // If no path found, return the total nodes visited
        List<String> result = new ArrayList<>();
        result.add("No path found.");
        result.add("Nodes visited: " + nodesVisited);
        return result;
    }
}
```

- **Kelas UCS:**

  Kelas ini berisi implementasi dari algoritma Uniform Cost Search dalam pencarian solusi permainan *word ladder*.

- **Kelas Node:**

  Kelas ini digunakan untuk merepresentasikan simpul dalam algoritma pencarian UCS. Kelas ini memiliki tiga atribut yaitu word (kata yang direpresentasikan oleh simpul), parent (simpul induk dari simpul ini), dan cost (biaya atau jarak dari simpul awal ke simpul ini).

- **Metode findPath:**

  Metode ini merupakan pengimplementasian algoritma ucs dalam mencari penyelesaian game word ladder. Metode ini menerima tiga parameter yaitu start (kata awal), end (kata tujuan), dan words (himpunan kata-kata yang mungkin digunakan dalam pencarian).Metode ini menggunakan struktur data Queue (antrian) dan Set (himpunan) untuk melacak simpul-simpul yang dikunjungi dan yang akan dikunjungi. Pada setiap iterasi,

metode ini mengambil simpul dengan biaya terendah dari antrian, kemudian memeriksa kata-kata yang mungkin dihasilkan dengan mengubah satu karakter pada kata saat ini. Jika kata baru ditemukan dan belum dikunjungi sebelumnya, kata tersebut ditambahkan ke antrian untuk diproses lebih lanjut. Proses ini berlanjut hingga ditemukan kata tujuan atau tidak ada jalur yang memungkinkan. Jika jalur ditemukan, metode ini mengembalikan daftar kata-kata yang membentuk jalur dari kata awal ke kata tujuan. Jika tidak, metode ini mengembalikan nilai null untuk menandakan bahwa tidak ada jalur yang ditemukan.

3.7.1 GBFS.java

```
import java.util.*;

public class GBFS {

   private static class Node {
      String word;
      Node parent;
      int heuristic; // Heuristic value for GBFS

      Node(String word, Node parent, int heuristic) {
         this.word = word;
         this.parent = parent;
         this.heuristic = heuristic;
      }
   }

   public static List<String> findPath(String start, String end, Set<String> words) {
      PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(node ->
node.heuristic));
      Set<String> visited = new HashSet<>();
      queue.add(new Node(start, null, calculateHeuristic(start, end)));
      visited.add(start);

      int nodesVisited = 0; // Add a counter for nodes visited

      while (!queue.isEmpty()) {
         Node current = queue.poll();
         String currentWord = current.word;

         nodesVisited++; // Increment the counter each time a node is visited

         if (currentWord.equals(end)) {
            List<String> path = new ArrayList<>();
            while (current != null) {
               path.add(current.word);
               current = current.parent;
            }
            Collections.reverse(path);

            path.add("Nodes visited: " + nodesVisited); // Add the total nodes visited to the path
```

```
            return path;
        }

        StringBuilder wordBuilder = new StringBuilder(currentWord);
        for (int i = 0; i < wordBuilder.length(); i++) {
            char originalChar = wordBuilder.charAt(i);
            for (char c = 'a'; c <= 'z'; c++) {
                if (c != originalChar) {
                    wordBuilder.setCharAt(i, c);
                    String newWord = wordBuilder.toString();
                    if (words.contains(newWord) && !visited.contains(newWord)) {
                        int heuristic = calculateHeuristic(newWord, end);
                        queue.add(new Node(newWord, current, heuristic));
                        visited.add(newWord);
                    }
                }
            }
            wordBuilder.setCharAt(i, originalChar);
        }
    }

    // If no path found, return the total nodes visited
    List<String> result = new ArrayList<>();
    result.add("No path found.");
    result.add("Nodes visited: " + nodesVisited);
    return result;
  }

  private static int calculateHeuristic(String current, String end) {
    int heuristic = 0;
    for (int i = 0; i < current.length(); i++) {
        if (current.charAt(i) != end.charAt(i)) {
            heuristic++;
        }
    }
    return heuristic;
  }
}
```

- **Kelas GBFS:**

  Kelas ini berisi implementasi dari algoritma Greedy Best-First Search dalam pencarian solusi permainan *word ladder.*

- **Kelas Node:**

  Kelas ini digunakan untuk merepresentasikan simpul dalam algoritma pencarian GBFS. Kelas ini memiliki tiga atribut yaitu word (kata yang direpresentasikan oleh simpul), parent (simpul induk dari simpul ini), dan heuristic (nilai heuristik untuk simpul ini).

- **Metode findPath:**

  Metode ini merupakan pengimplementasian algoritma gbfs dalam mencari penyelesaian game word ladder. Metode ini menerima tiga parameter: start (kata awal), end (kata tujuan), dan words (himpunan kata-kata yang mungkin digunakan dalam pencarian). Metode ini menggunakan struktur data PriorityQueue untuk mengurutkan simpul berdasarkan nilai heuristik terbaik. Pada setiap iterasi, metode ini mengambil simpul dengan nilai heuristik terendah dari antrian, kemudian memeriksa kata-kata yang mungkin dihasilkan dengan mengubah satu karakter pada kata saat ini. Jika kata baru ditemukan dan belum dikunjungi sebelumnya, kata tersebut ditambahkan ke antrian untuk diproses lebih lanjut dengan nilai heuristik yang dihitung. Proses ini berlanjut hingga ditemukan kata tujuan atau tidak ada jalur yang memungkinkan. Jika jalur ditemukan, metode ini mengembalikan daftar kata-kata yang membentuk jalur dari kata awal ke kata tujuan. Jika tidak, metode ini mengembalikan nilai null untuk menandakan bahwa tidak ada jalur yang ditemukan.

- **Metode calculateHeuristic**

  Metode ini digunakan untuk menghitung nilai heuristik antara dua kata. Metode ini menerima dua parameter yaitu current (kata saat ini) dan end (kata tujuan). Metode ini menghitung nilai heuristik sebagai jumlah karakter yang berbeda antara current dan end.

3.7.3 AStar.java

```
import java.util.*;

public class AStar {

    private static class Node {
        String word;
        Node parent;
        int cost;
        int heuristic;

        Node(String word, Node parent, int cost, int heuristic) {
            this.word = word;
            this.parent = parent;
            this.cost = cost;
            this.heuristic = heuristic;
        }

        int getFValue() {
            return cost + heuristic;
        }
    }
```

```
    public static List<String> findPath(String start, String end, Set<String> words) {
        PriorityQueue<Node> queue = new
PriorityQueue<>(Comparator.comparingInt(Node::getFValue));
        Map<String, Integer> costs = new HashMap<>();
        queue.add(new Node(start, null, 0, calculateHeuristic(start, end)));
        costs.put(start, 0);

        int nodesVisited = 0; // Add a counter for nodes visited

        while (!queue.isEmpty()) {
            Node current = queue.poll();
            String currentWord = current.word;

            nodesVisited++; // Increment the counter each time a node is visited

            if (currentWord.equals(end)) {
                List<String> path = new ArrayList<>();
                while (current != null) {
                    path.add(current.word);
                    current = current.parent;
                }
                Collections.reverse(path);

                path.add("Nodes visited: " + nodesVisited); // Add the total nodes visited to the
path

                return path;
            }

            StringBuilder wordBuilder = new StringBuilder(currentWord);
            for (int i = 0; i < wordBuilder.length(); i++) {
                char originalChar = wordBuilder.charAt(i);
                for (char c = 'a'; c <= 'z'; c++) {
                    if (c != originalChar) {
                        wordBuilder.setCharAt(i, c);
                        String newWord = wordBuilder.toString();
                        if (words.contains(newWord)) {
                            int newCost = costs.getOrDefault(currentWord, Integer.MAX_VALUE)
+ 1;
                            if (newCost < costs.getOrDefault(newWord, Integer.MAX_VALUE)) {
                                costs.put(newWord, newCost);
                                int heuristic = calculateHeuristic(newWord, end);
                                queue.add(new Node(newWord, current, newCost, heuristic));
                            }
                        }
                    }
                }
                wordBuilder.setCharAt(i, originalChar); // revert back for next iteration
            }
        }

        // If no path found, return the total nodes visited
```

```
        List<String> result = new ArrayList<>();
        result.add("No path found.");
        result.add("Nodes visited: " + nodesVisited);
        return result;
    }


    private static int calculateHeuristic(String current, String end) {
        int heuristic = 0;
        for (int i = 0; i < current.length(); i++) {
            if (current.charAt(i) != end.charAt(i)) {
                heuristic++;
            }
        }
        return heuristic;
    }
}
```

- **Kelas AStar:**

  Kelas ini berisi implementasi dari algoritma A* dalam pencarian solusi permainan *word ladder.*

- **Kelas Node:**

  Kelas ini digunakan untuk merepresentasikan simpul dalam algoritma pencarian A*. Kelas ini memiliki empat atribut yaitu word (kata yang direpresentasikan oleh simpul), parent (simpul induk dari simpul ini), cost (biaya atau jarak dari simpul awal ke simpul ini), dan heuristic (nilai heuristik untuk simpul ini)..

- **Metode getFValue:**

  Metode ini terdapat dalam kelas Node. Metode ini dibuat untuk menghitung nilai fungsi evaluasi (F-value) dari simpul berdasarkan biaya dan nilai heuristik

- **Metode findPath:**

  Metode ini merupakan pengimplementasian algoritma a* dalam mencari penyelesaian game word ladder. Metode ini menerima tiga parameter yaitu start (kata awal), end (kata tujuan), dan words (himpunan kata-kata yang mungkin digunakan dalam pencarian). Metode ini menggunakan struktur data PriorityQueue untuk mengurutkan simpul berdasarkan nilai fungsi evaluasi terbaik. Metode ini menggunakan Map untuk melacak biaya terbaik yang telah ditemukan untuk setiap kata. Pada setiap iterasi, metode ini mengambil simpul dengan

nilai F-value terendah dari antrian, kemudian memeriksa kata-kata yang mungkin dihasilkan dengan mengubah satu karakter pada kata saat ini. Jika kata baru ditemukan dan memiliki biaya yang lebih rendah, kata tersebut ditambahkan ke antrian untuk diproses lebih lanjut dengan nilai biaya dan nilai heuristik yang dihitung. Proses ini berlanjut hingga ditemukan kata tujuan atau tidak ada jalur yang memungkinkan. Jika jalur ditemukan, metode ini mengembalikan daftar kata-kata yang membentuk jalur dari kata awal ke kata tujuan. Jika tidak, metode ini mengembalikan nilai null untuk menandakan bahwa tidak ada jalur yang ditemukan.

- **Metode calculateHeuristic**

  Metode ini digunakan untuk menghitung nilai heuristik antara dua kata. Metode ini menerima dua parameter yaitu current (kata saat ini) dan end (kata tujuan). Metode ini menghitung nilai heuristik sebagai jumlah karakter yang berbeda antara current dan end.

3.7.4 WordLadderSolverGUI.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.List;
import java.util.Set;

// Custom JPanel with background image
class BackgroundPanel extends JPanel {
    private Image backgroundImage;

    public BackgroundPanel(Image backgroundImage) {
        this.backgroundImage = backgroundImage;
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawImage(backgroundImage, 0, 0, getWidth(), getHeight(), this);
    }
}

public class WordLadderSolverGUI extends JFrame {
    private JTextField startWordField, endWordField;
    private JComboBox<String> algorithmComboBox;
    private JTextArea resultArea;

    public WordLadderSolverGUI(Set<String> englishWords) {
        super("Word Ladder Solver ");
```

```
        // Load background image
        ImageIcon backgroundImageIcon = new ImageIcon("../doc/background.jpg");
        Image backgroundImage = backgroundImageIcon.getImage();

        // Initialize components
        JLabel titleLabel = new JLabel("Word Ladder Solver", SwingConstants.CENTER);
        titleLabel.setFont(new Font("Arial", Font.BOLD, 18));
        titleLabel.setForeground(Color.WHITE);

        // Assuming `container` is the parent container of `titleLabel`
        titleLabel.setLayout(new BoxLayout(titleLabel, BoxLayout.Y_AXIS));
        titleLabel.setAlignmentX(Component.CENTER_ALIGNMENT);

        JLabel startWordLabel = new JLabel() {
            @Override
            protected void paintComponent(Graphics g) {
                // Draw the image
                g.drawImage(backgroundImage, 0, 0, getWidth(), getHeight(), this);

                // Set the text color
                g.setColor(Color.WHITE);

                // Set the font
                Font font = new Font("Arial", Font.BOLD, 15);
                g.setFont(font);

                // Calculate the width and height of the text
                FontMetrics metrics = g.getFontMetrics(font);
                int textWidth = metrics.stringWidth("Start Word:");
                int textHeight = metrics.getHeight();

                // Calculate the x and y coordinates where the text should be drawn
                int x = (getWidth() - textWidth) / 2;
                int y = (getHeight() - textHeight) / 2 + metrics.getAscent(); // The ascent is the distance
from the baseline to the top of the text

                // Draw the text
                g.drawString("Start Word:", x, y);
            }
        };

        startWordLabel.setHorizontalAlignment(SwingConstants.RIGHT);
        startWordField = new JTextField(15);

        JLabel endWordLabel = new JLabel() {
            @Override
            protected void paintComponent(Graphics g) {
                // Draw the image
                g.drawImage(backgroundImage, 0, 0, getWidth(), getHeight(), this);

                // Set the text color
                g.setColor(Color.WHITE);
```

```
            // Set the font
            Font font = new Font("Arial", Font.BOLD, 15);
            g.setFont(font);

            // Calculate the width and height of the text
            FontMetrics metrics = g.getFontMetrics(font);
            int textWidth = metrics.stringWidth("End Word:");
            int textHeight = metrics.getHeight();

            // Calculate the x and y coordinates where the text should be drawn
            int x = (getWidth() - textWidth) / 2;
            int y = (getHeight() - textHeight) / 2 + metrics.getAscent(); // The ascent is the distance
from the baseline to the top of the text

            // Draw the text
            g.drawString("End Word:", x, y);
        }
    };
    endWordLabel.setHorizontalAlignment(SwingConstants.RIGHT);
    endWordField = new JTextField(15);

    JLabel algorithmLabel = new JLabel() {
        @Override
        protected void paintComponent(Graphics g) {
            // Draw the image
            g.drawImage(backgroundImage, 0, 0, getWidth(), getHeight(), this);

            // Set the text color
            g.setColor(Color.WHITE);

            // Set the font
            Font font = new Font("Arial", Font.BOLD, 15);
            g.setFont(font);

            // Calculate the width and height of the text
            FontMetrics metrics = g.getFontMetrics(font);
            int textWidth = metrics.stringWidth("Algorithm:");
            int textHeight = metrics.getHeight();

            // Calculate the x and y coordinates where the text should be drawn
            int x = (getWidth() - textWidth) / 2;
            int y = (getHeight() - textHeight) / 2 + metrics.getAscent(); // The ascent is the distance
from the baseline to the top of the text

            // Draw the text
            g.drawString("Algorithm:", x, y);
        }
    };

    algorithmLabel.setHorizontalAlignment(SwingConstants.RIGHT);
    String[] algorithms = {"Uniform Cost Search", "Greedy Best First Search", "A*"};
    algorithmComboBox = new JComboBox<>(algorithms);
    algorithmComboBox.setBackground(Color.LIGHT_GRAY);
```

```
algorithmComboBox.setForeground(Color.BLACK);

JButton solveButton = new JButton("Solve");
solveButton.setBackground(Color.BLACK);
solveButton.setForeground(Color.WHITE);
solveButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        String startWord = startWordField.getText().toLowerCase();
        String endWord = endWordField.getText().toLowerCase();
        String selectedAlgorithm = (String) algorithmComboBox.getSelectedItem();
        List<String> path = null;
        long startTime = System.currentTimeMillis();

        // Check if startWord and endWord have the same length
        if (startWord.length() != endWord.length()) {
            resultArea.setText("Error! \nStart and end words must have the same length.");

            return; // Exit the method early
        }

        switch (selectedAlgorithm) {
            case "Uniform Cost Search":
                path = UCS.findPath(startWord, endWord, englishWords);
                break;
            case "Greedy Best First Search":
                path = GBFS.findPath(startWord, endWord, englishWords);
                break;
            case "A*":
                path = AStar.findPath(startWord, endWord, englishWords);
                break;
            default:
                break;
        }

        long endTime = System.currentTimeMillis();

        // If you want to know the memory used, use this code
        // if (path != null) {
        //     // Calculate memory usage
        //     Runtime runtime = Runtime.getRuntime();
        //     long memoryUsedBytes = runtime.totalMemory() - runtime.freeMemory();
        //     String memoryUsedKB = String.format("%.2f", (double) memoryUsedBytes / 1024);

        //     resultArea.setText("Path found:\n" + String.join(" -> ", path) +
        //             "\nNumber of nodes visited: " + path.size() +
        //             "\nExecution time: " + (endTime - startTime) + " milliseconds" +
        //             "\nMemory used: " + memoryUsedKB + " kilobytes");
        // } else {
        //     resultArea.setText("No path found.");
        // }
```

```
            if (path != null && !path.isEmpty()) {
                List<String> pathWithoutLastElement = path.subList(0, path.size() - 1);
                resultArea.setText("Path found:\n" + String.join(" -> ", pathWithoutLastElement) +
                        "\n" + path.get(path.size() - 1) +
                        "\nExecution time: " + (endTime - startTime) + " milliseconds");
            } else {
                resultArea.setText("No path found.");
            }
        }
    });

    resultArea = new JTextArea(10, 30);
    resultArea.setEditable(false);
    JScrollPane resultScrollPane = new JScrollPane(resultArea);

    // Layout setup
    JPanel inputPanel = new JPanel(new GridLayout(3, 2, 10, 10));
    inputPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
    inputPanel.add(startWordLabel);
    inputPanel.add(startWordField);
    inputPanel.add(endWordLabel);
    inputPanel.add(endWordField);
    inputPanel.add(algorithmLabel);
    inputPanel.add(algorithmComboBox);

    JPanel buttonPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
    buttonPanel.add(solveButton);

    JPanel mainPanel = new BackgroundPanel(backgroundImage); // Use custom panel with
background
    mainPanel.setLayout(new BoxLayout(mainPanel, BoxLayout.Y_AXIS));
    mainPanel.add(titleLabel);
    mainPanel.add(Box.createVerticalStrut(10));
    mainPanel.add(inputPanel);
    mainPanel.add(Box.createVerticalStrut(10));
    mainPanel.add(buttonPanel);
    mainPanel.add(Box.createVerticalStrut(10));
    mainPanel.add(resultScrollPane);

    // Add main panel to frame
    add(mainPanel);

    // Frame settings
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    pack(); // Auto-size frame based on components
    setLocationRelativeTo(null);
    setVisible(true);
  }

  public static void loadWords(String filename, Set<String> words) {
    try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
        String line;
        while ((line = reader.readLine()) != null) {
```

```
                words.add(line.trim().toLowerCase());
            }
        } catch (IOException e) {
            System.err.println("Error reading file: " + e.getMessage());
        }
    }
}
```

- **Kelas BackgroundPanel:**

  Kelas ini merupakan JPanel custom yang memiliki latar belakang gambar. Kelas ini memiliki atribut backgroundImage untuk menyimpan gambar latar belakang. Kelas ini memiliki metode paintComponent(Graphics g) untuk menggambar gambar latar belakang di panel.

- **Metode BackgroundPanel:**

  Metode ini terdapat dalam kelas BackgroundPanel dan berfungsi sebagai constructor kelas tersebut.

- **Metode paintComponent:**

  Metode ini terdapat dalam kelas BackgroundPanel. Metode ini merupakan sebuah metode yang di-override dari kelas JPanel yang digunakan untuk menggambar komponen panel, khususnya untuk menggambar gambar latar belakang di panel dengan menggunakan objek Graphics.

- **Kelas WordLadderSolverGUI**

  Kelas ini merupakan JFrame yang berfungsi sebagai antarmuka pengguna (GUI) untuk menyelesaikan Word Ladder. Kelas ini memiliki beberapa komponen GUI seperti JTextField untuk kata awal dan akhir, JComboBox untuk memilih algoritma pencarian, JButton untuk menyelesaikan permasalahan, dan JTextArea untuk menampilkan hasil pencarian.

- **Metode WordLadderSolverGUI**

  Metode ini terdapat didalam kelas WordLadderSolverGUI dan berfungsi sebagai Konstruktor untuk inisialisasi dan menyiapkan antarmuka pengguna.

- **Metode loadWords**

  Metode ini digunakan untuk memuat kata-kata dari file teks ke dalam sebuah himpunan kata.

3.8 Main.java

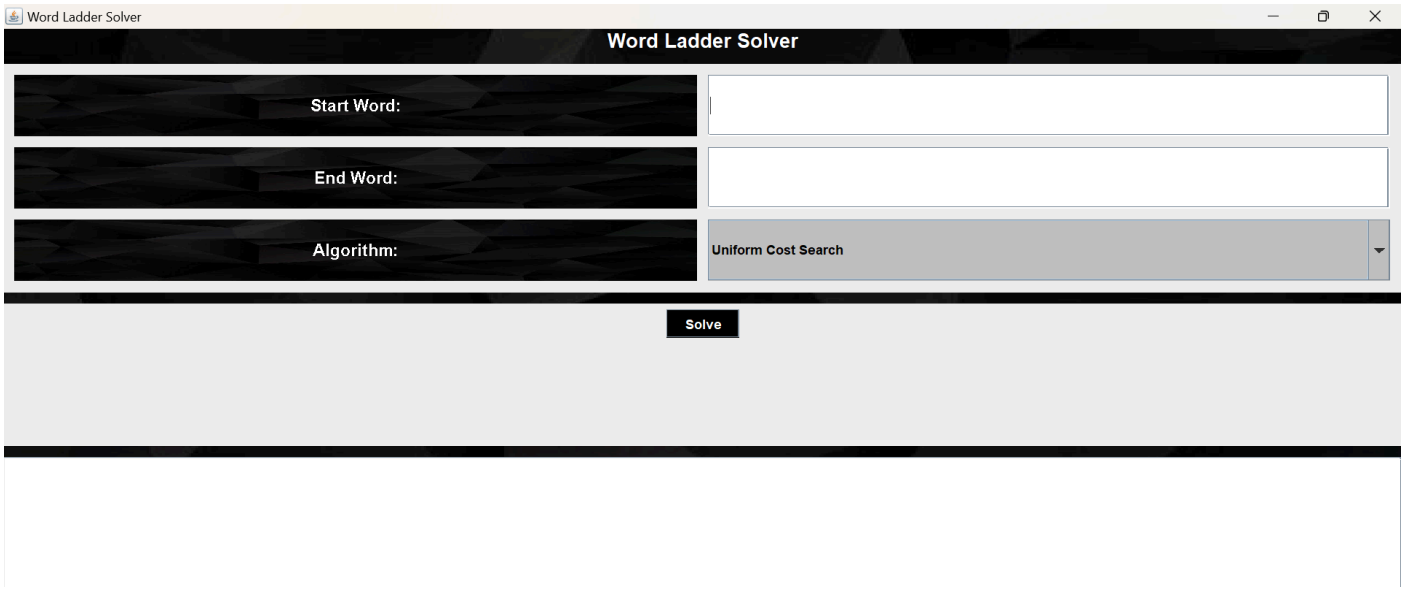
Program ini digunakan untuk menjalankan seluruh code yang ada secara keseluruhan

```
import java.util.HashSet;
import java.util.Set;

import javax.swing.SwingUtilities;

public class Main {
   public static void main(String[] args) {
      Set<String> englishWords = new HashSet<>();
      WordLadderSolverGUI.loadWords("words_alpha.txt", englishWords); // Load words from file

      SwingUtilities.invokeLater(new Runnable() {
         public void run() {
            new WordLadderSolverGUI(englishWords);
         }
      });
   }
}
```

BAB 4

EKSPERIMEN

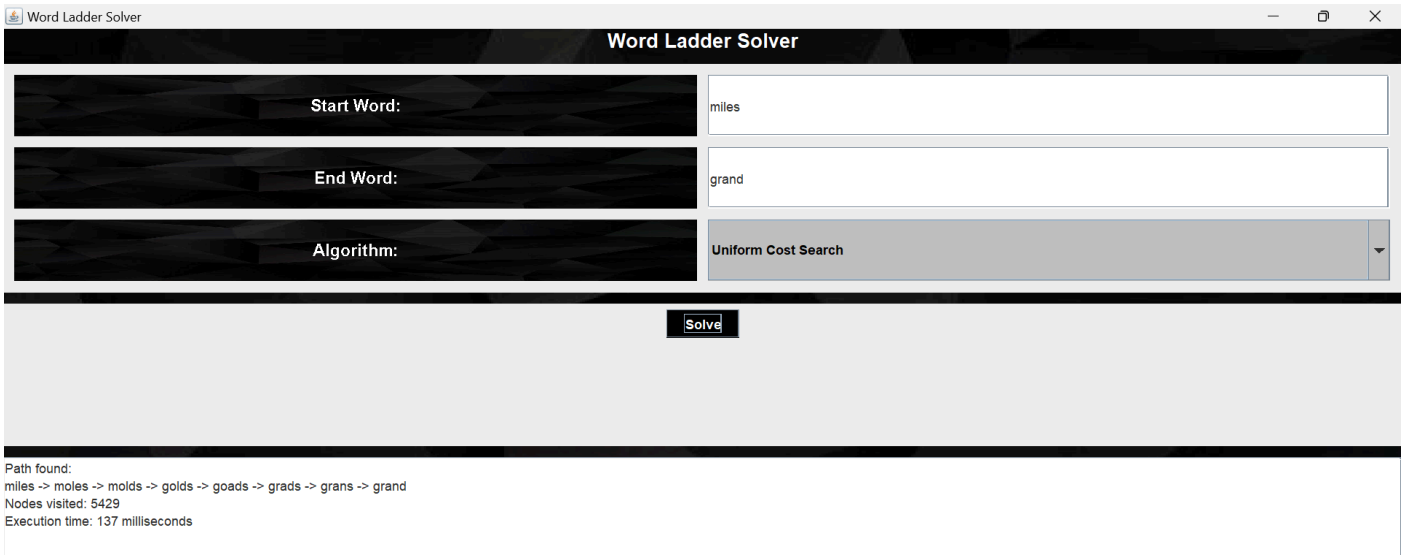
4.1 Tampilan GUI



4.2 Uniform Cost Search

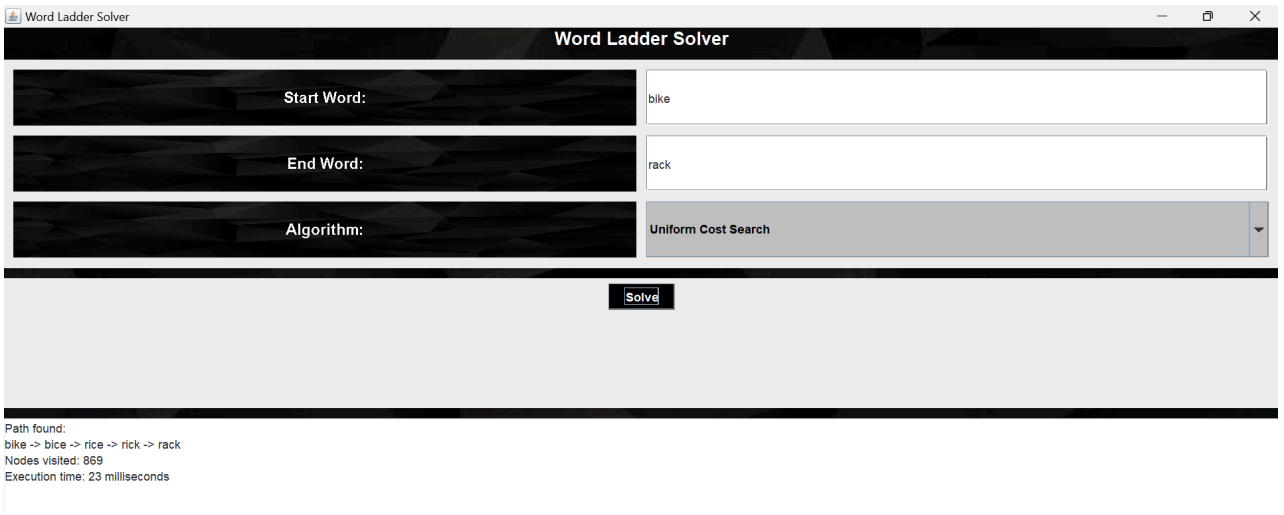
Start Word : miles

End Word : grand



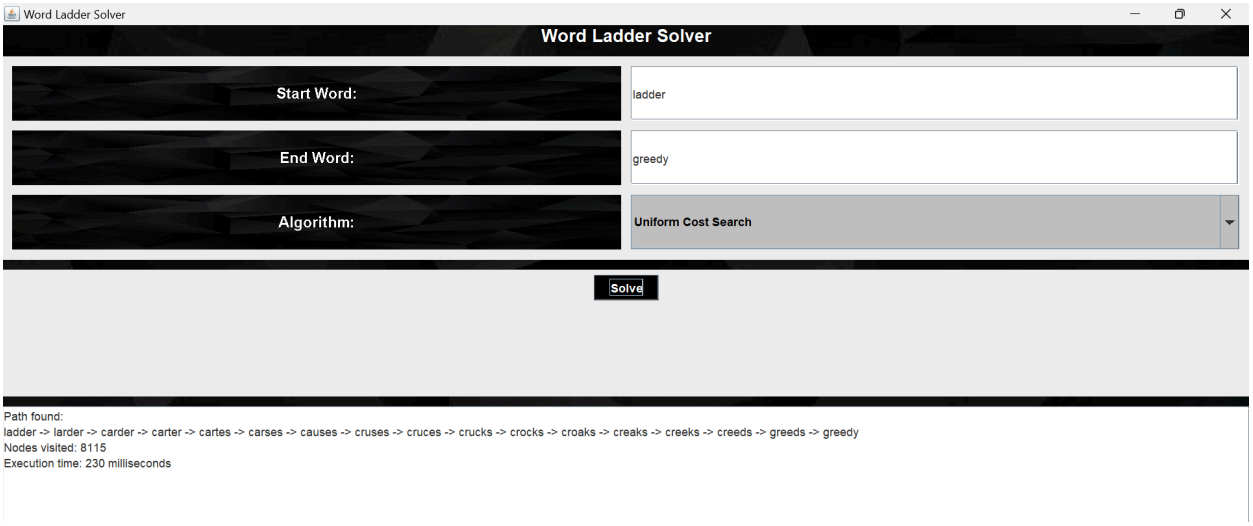
Start Word : bike

End Word : rack



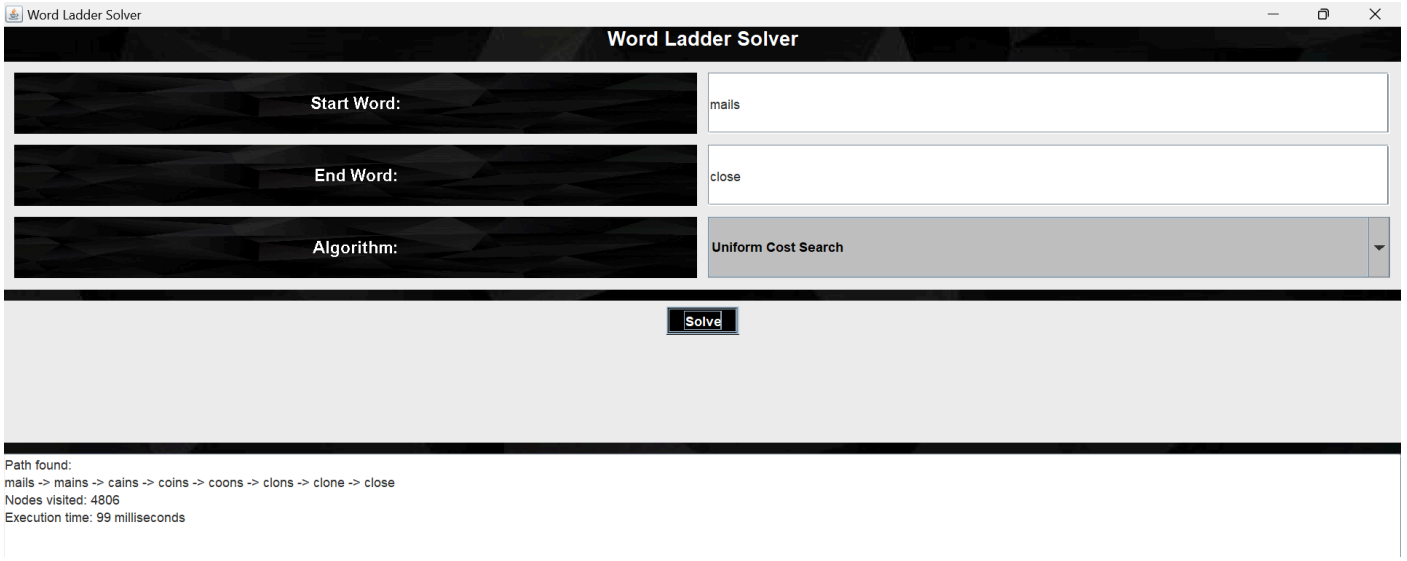
Start Word : ladder

End Word : greedy



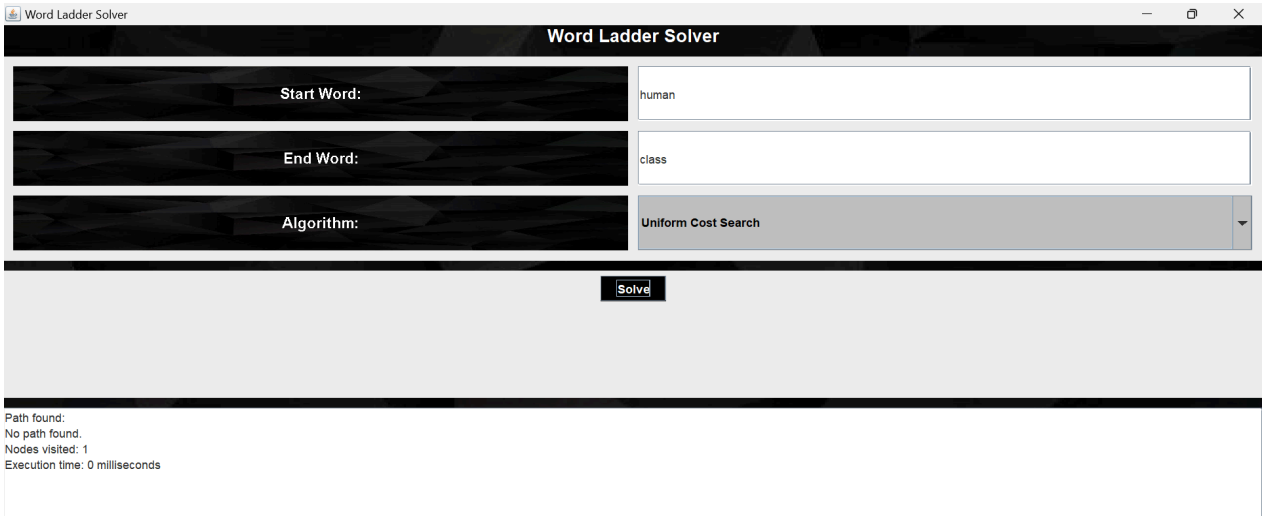
Start Word : mails

End Word : close



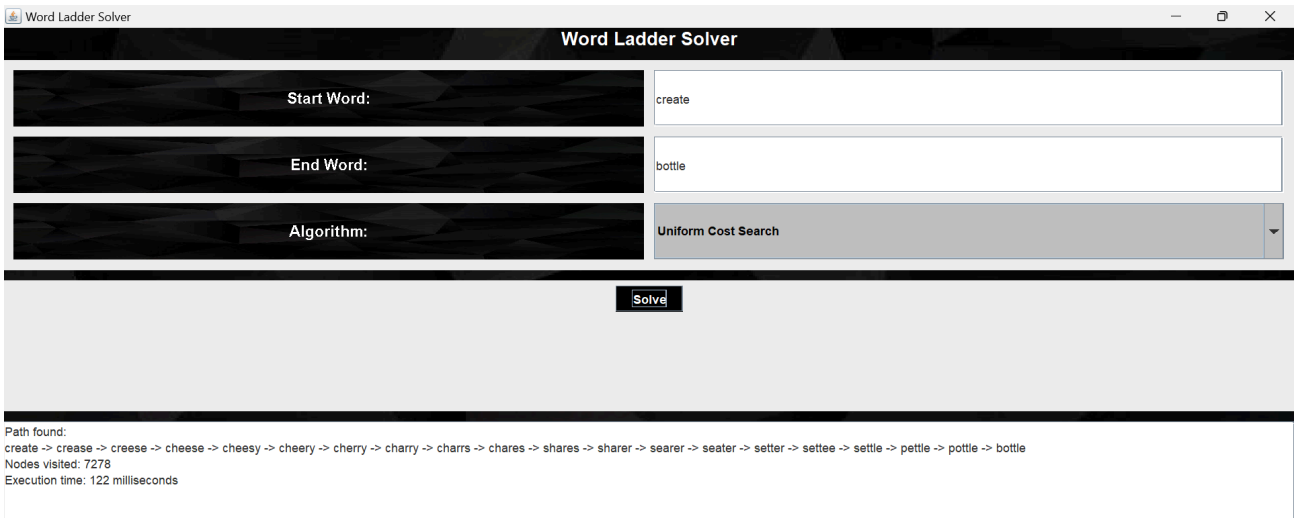
Start Word : human

End Word : class



Start Word : create

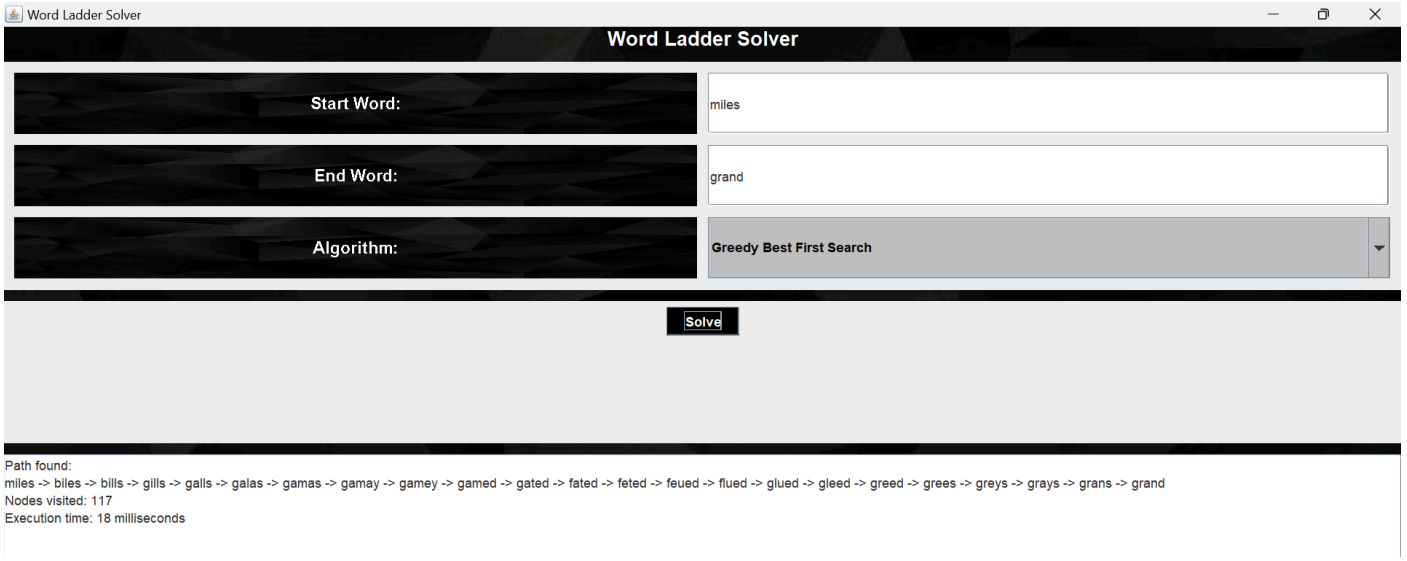
End Word : bottle



4.3 Greedy Best First Search

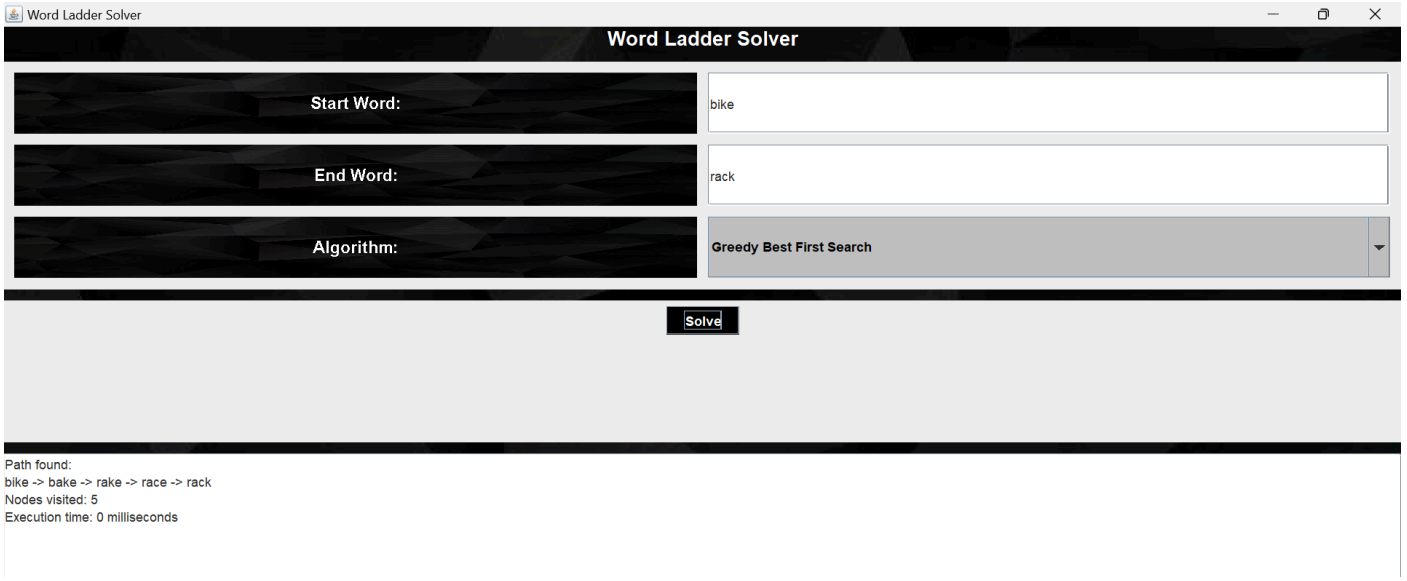
Start Word : miles

End Word : grand



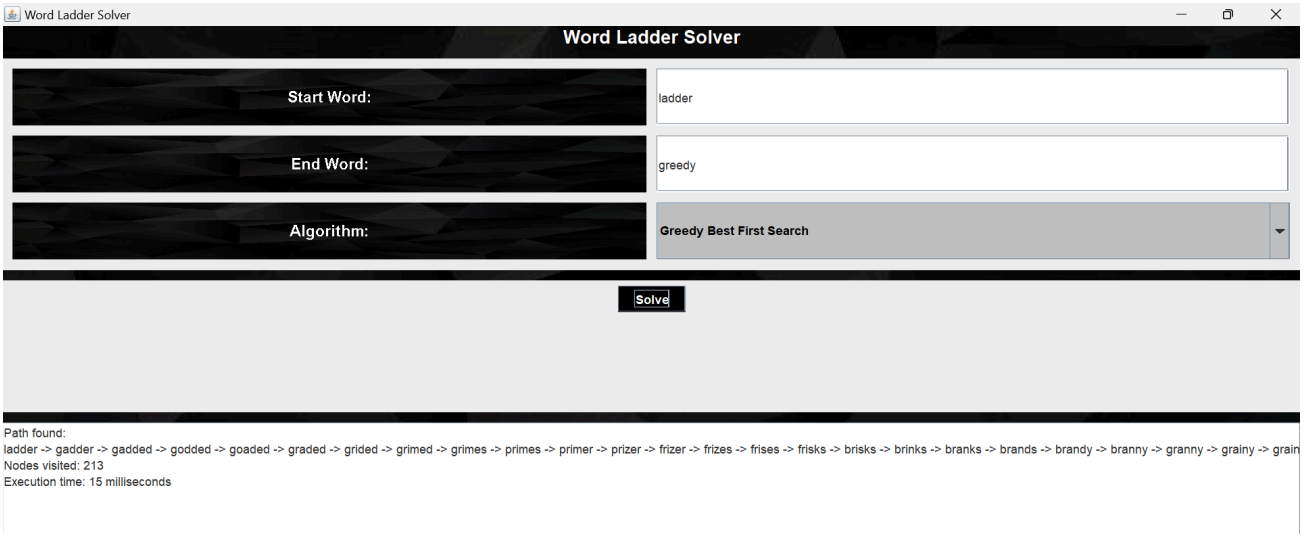
Start Word : bike

End Word : rack



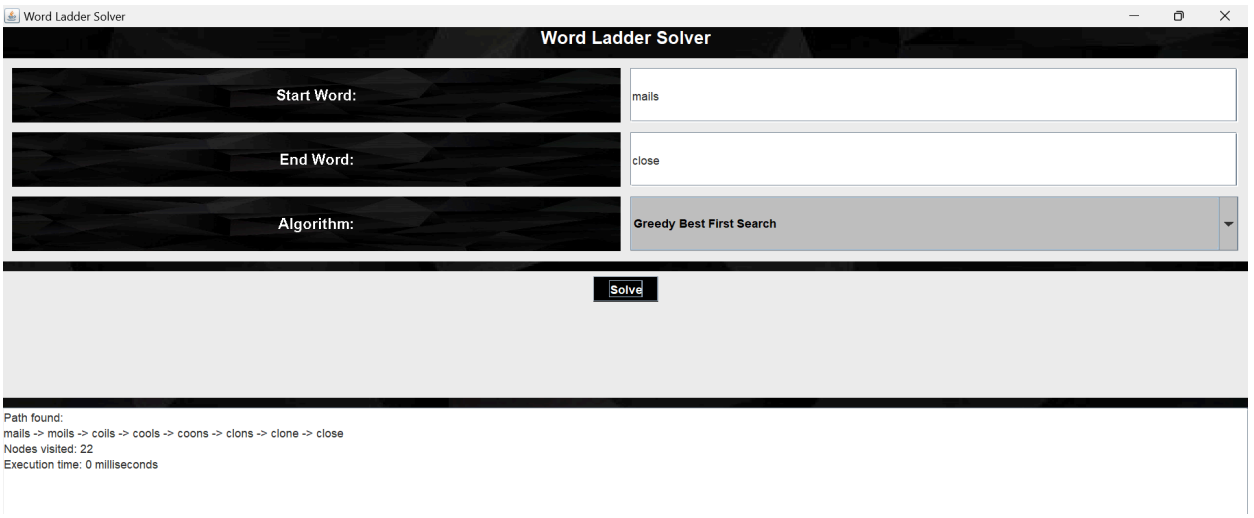
Start Word : ladder

End Word : greedy



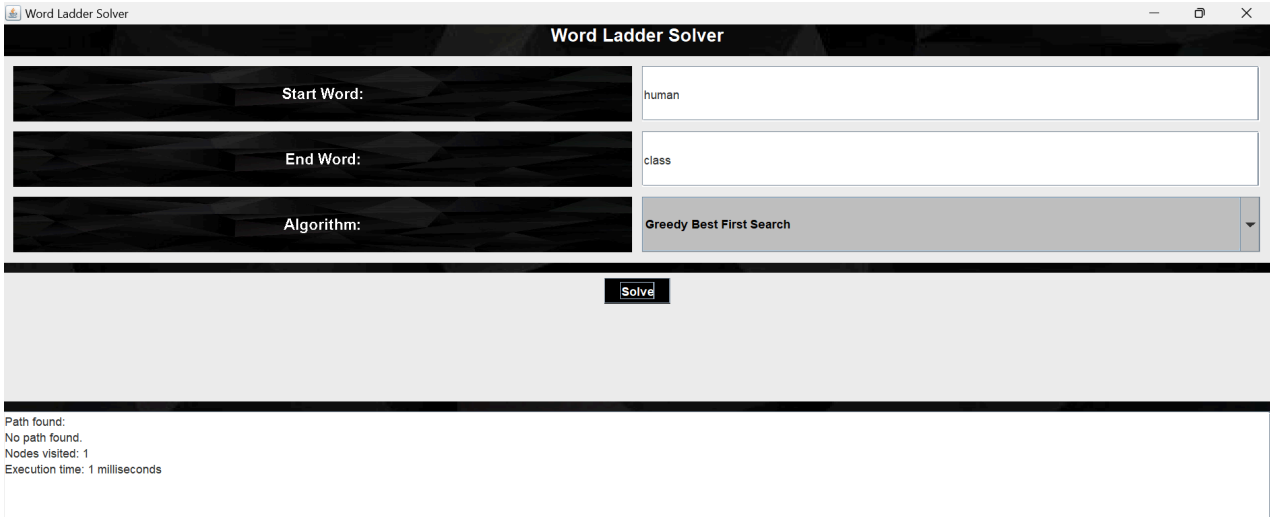
Start Word : mails

End Word : close



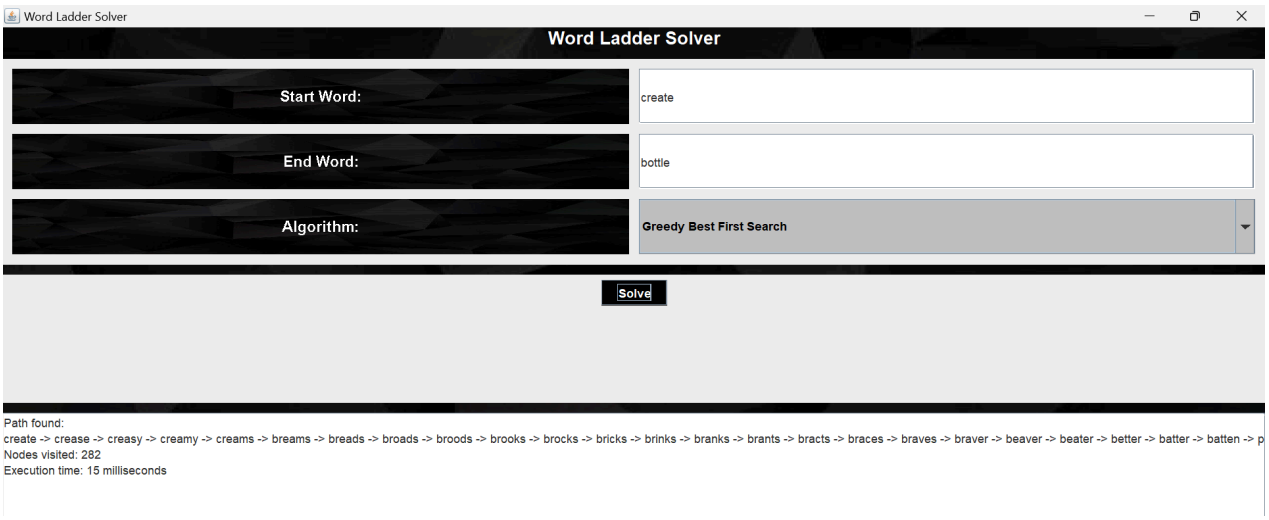
Start Word : human

End Word : class



Start Word : create

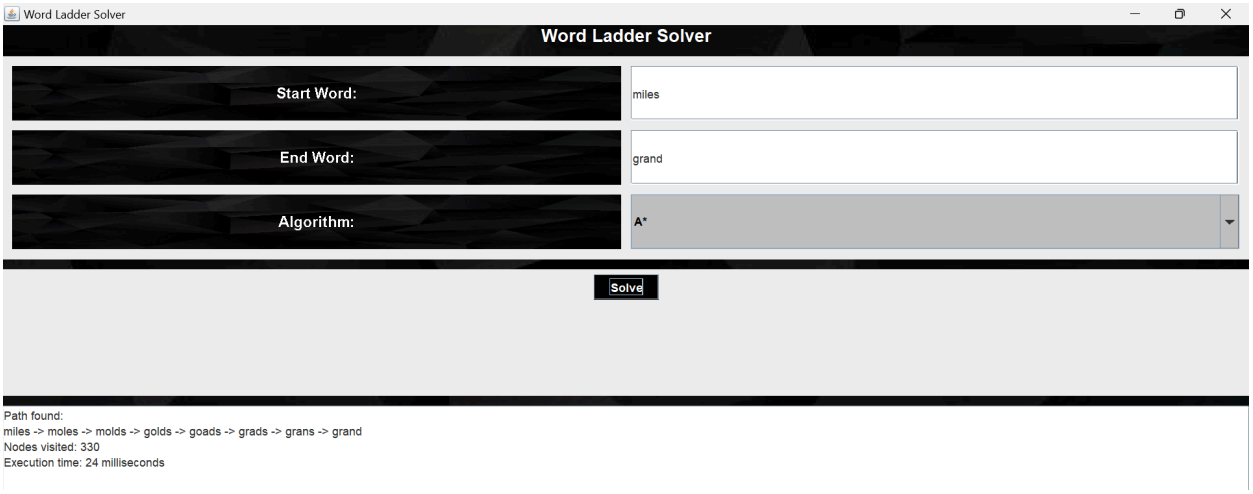
End Word : bottle



4.4 A*

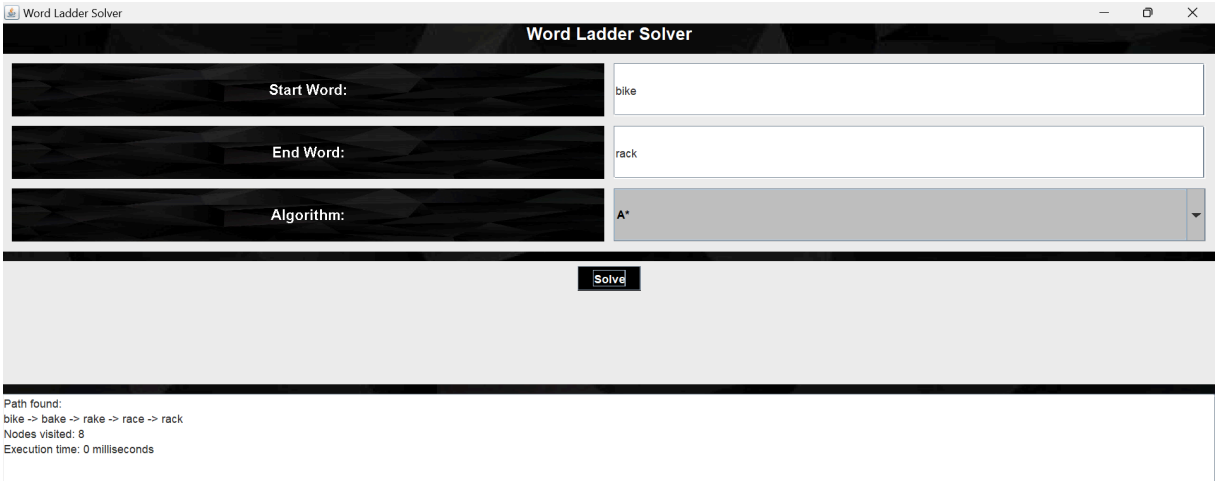
Start Word : miles

End Word : grand



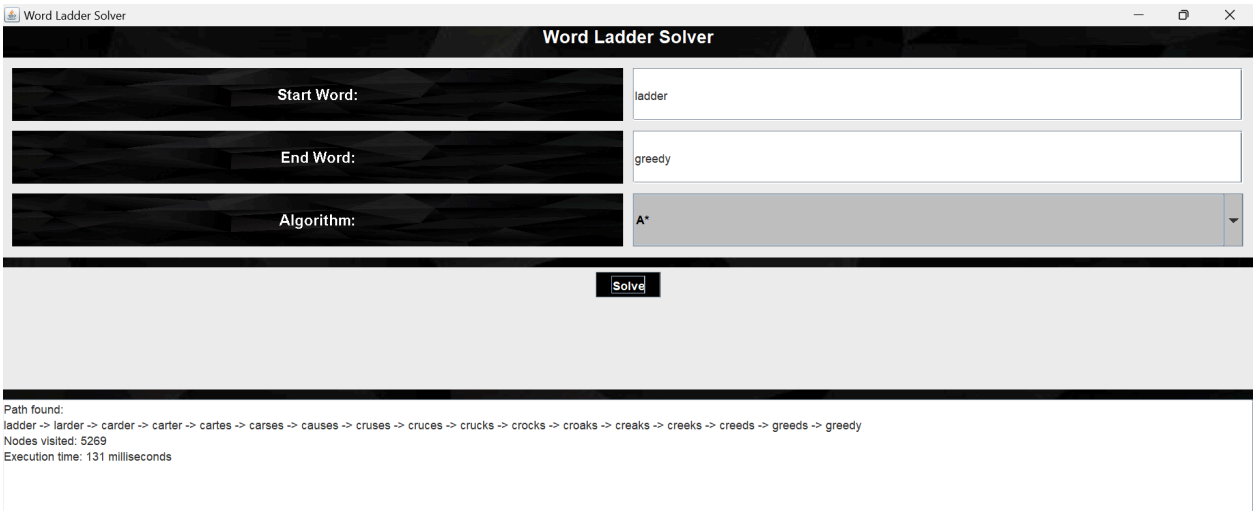
Start Word : bike

End Word : rack



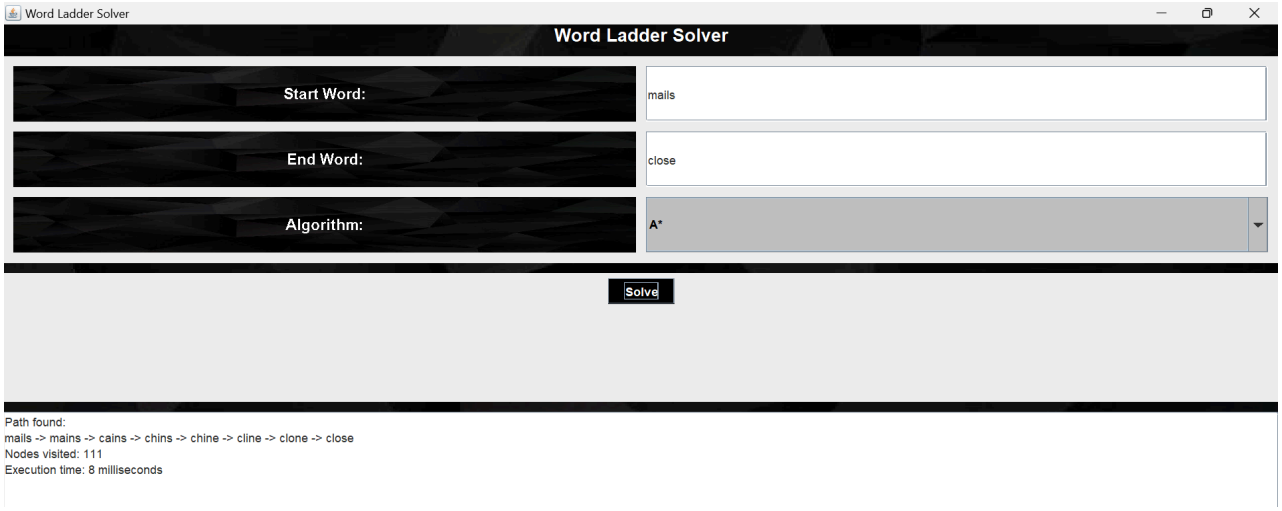
Start Word : ladder

End Word : greedy



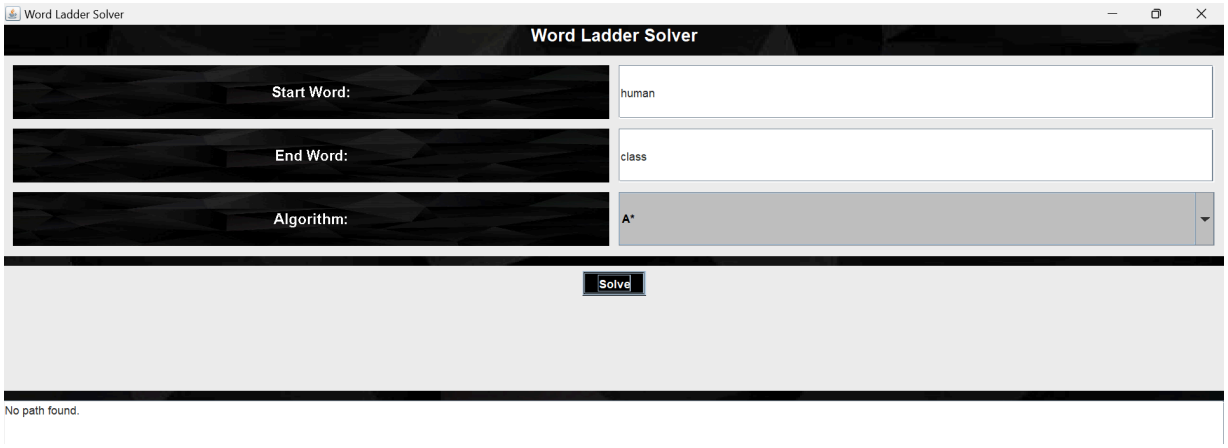
Start Word : mails

End Word : close



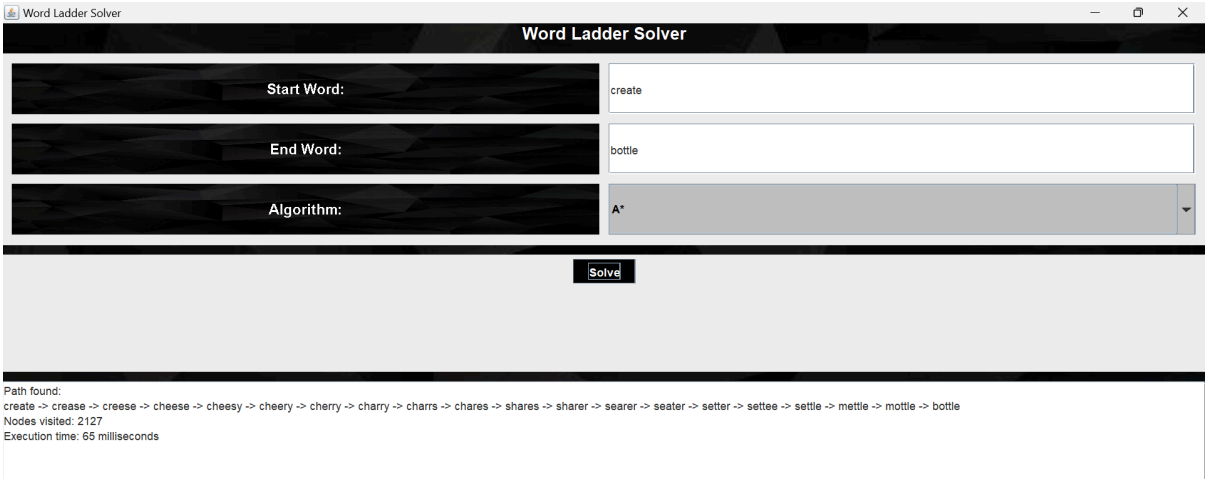
Start Word : human

End Word : class



Start Word : create

End Word : bottle



4.5 Hasil Analisis

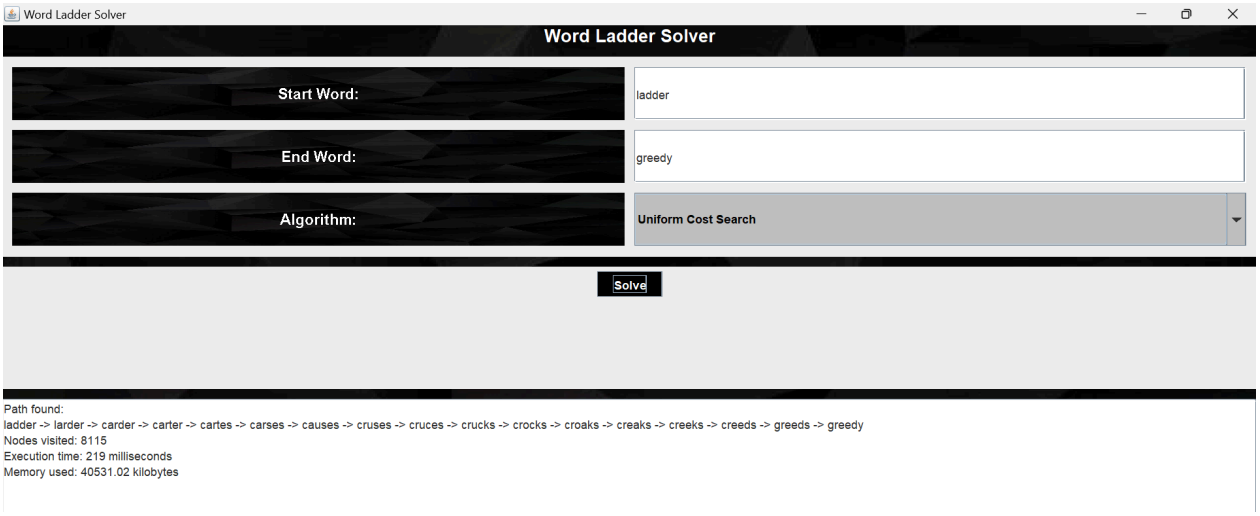
Berikut adalah hasil analisis dan bukti pendukung terkait analisis yang dilakukan menggunakan kata awal yaitu “ladder” dan kata akhir yaitu “greedy”

Untuk mengetahui memory yang digunakan oleh suatu algoritma, perlu ditambahkan kode berikut di dalam file WordLadderGUI.java :

```
if (path != null) {
    // Calculate memory usage
    Runtime runtime = Runtime.getRuntime();
    long memoryUsedBytes = runtime.totalMemory() - runtime.freeMemory();
    String memoryUsedKB = String.format("%.2f", (double) memoryUsedBytes / 1024);
```

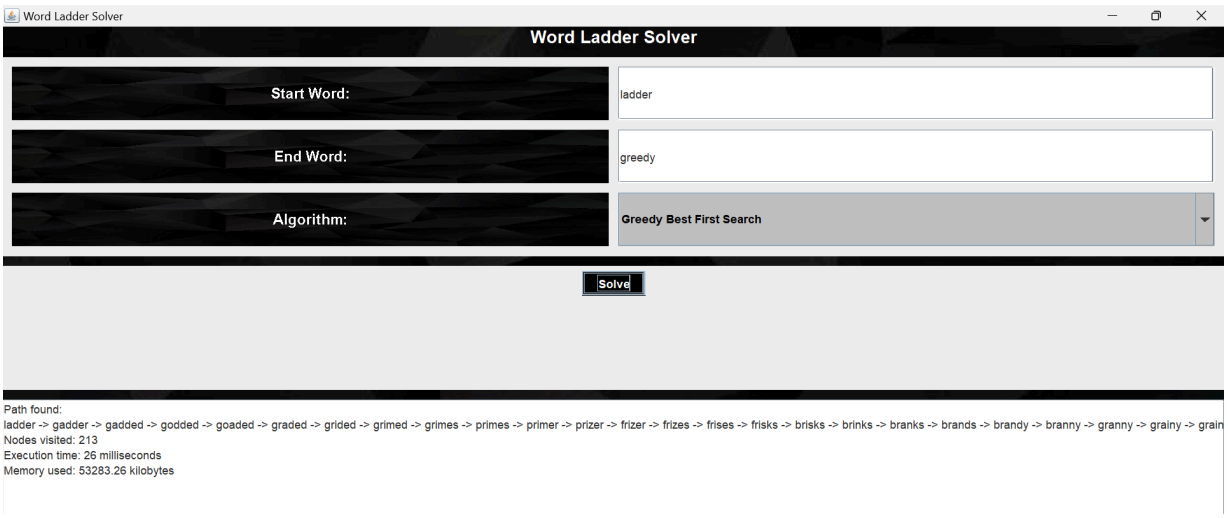
```
            List<String> pathWithoutLastElement = path.subList(0, path.size() - 1);
            resultArea.setText("Path found:\n" + String.join(" -> ", pathWithoutLastElement) +
                    "\n" + path.get(path.size() - 1) +
                    "\nExecution time: " + (endTime - startTime) + " milliseconds" +
                    "\nMemory used: " + memoryUsedKB + " kilobytes");
        } else {
            resultArea.setText("No path found.");
        }
```

- **Uniform Cost Search (UCS)**



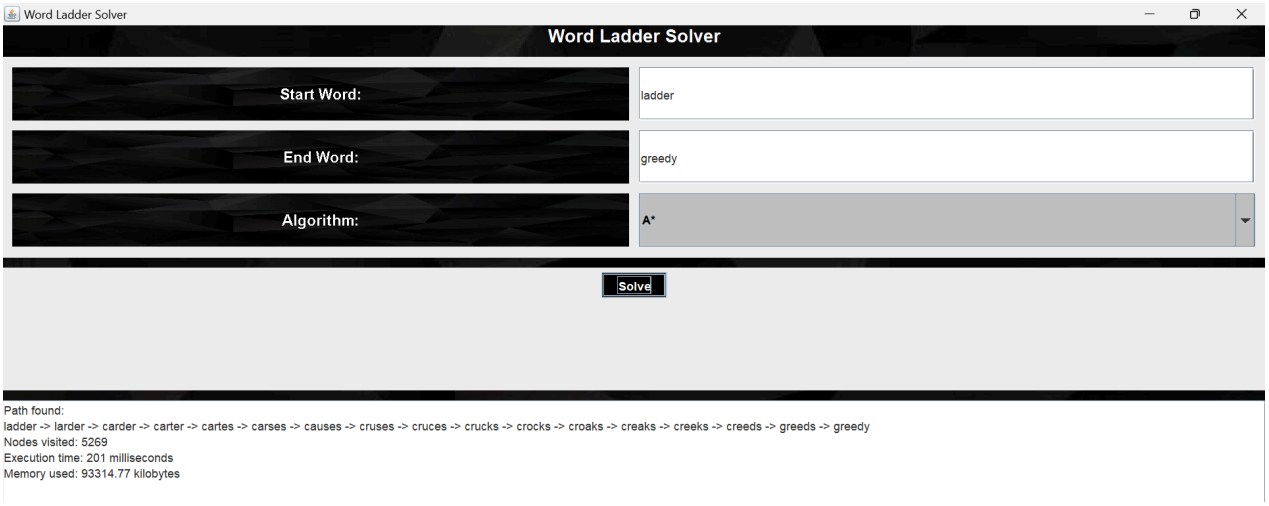
1. Optimalitas: Algoritma UCS menjamin optimalitas karena mengunjungi simpul dengan biaya minimum terlebih dahulu.
2. Waktu Eksekusi: Waktu eksekusi UCS tergantung pada jumlah simpul yang dieksplorasi. Karena UCS mengeksplorasi simpul berdasarkan biaya, ia mungkin membutuhkan waktu yang lebih lama daripada algoritma lain jika ada banyak simpul dengan biaya rendah. Hal tersebut bisa dibuktikan pada gambar yang telah dicantumkan.
3. Memori: UCS membutuhkan memori untuk menyimpan simpul yang belum dieksplorasi dan biaya setiap simpul. Namun, karena hanya menyimpan informasi biaya, memori yang dibutuhkan tidak terlalu besar dibanding memori yang dibutuhkan oleh algoritma gbfs dan a*

- **Greedy Best First Search (GBFS)**



1. Optimalitas: Algoritma GBFS tidak menjamin optimalitas karena hanya mempertimbangkan heuristik lokal (jarak ke "*End Word*") tanpa memperhatikan biaya total. Ini dapat mengarah ke penemuan jalur yang lebih pendek secara lokal tetapi tidak optimal secara global.
2. Waktu Eksekusi: Algoritma GBFS dapat menemukan jalur dengan cepat jika heuristiknya mendekati solusi secara akurat. Namun, jika heuristiknya tidak akurat, GBFS dapat menghabiskan waktu untuk menjelajahi jalur yang tidak optimal.
3. Memori: Algoritma GBFS membutuhkan memori untuk menyimpan simpul yang belum dieksplorasi dan nilai heuristik setiap simpul. Jumlah memori yang dibutuhkan tergantung pada presisi heuristik yang digunakan.

- **A***



1. Optimalitas:Algoritma A* menjamin optimalitas jika fungsi heuristiknya admissible (tidak melebih-lebihkan biaya sebenarnya) dan konsisten.
2. Waktu Eksekusi: Algoritma A* biasanya efisien karena kombinasi antara biaya aktual dan heuristik. Namun, waktu eksekusi dapat bervariasi tergantung pada heuristik yang digunakan dan struktur graf yang dieksplorasi.
3. Memori: Algoritma A* membutuhkan memori untuk menyimpan simpul yang belum dieksplorasi, biaya setiap simpul, nilai heuristik, serta biaya total (biaya aktual + heuristik).

Hal itu membuat algoritma A* membutuhkan kapasitas memori yang lebih besar dibanding ucs dan gbfs

BAB 5
PENUTUP

5.1 Kesimpulan

Dalam mencari solusi optimal permainan *word ladder* terdapat beberapa kesimpulan yaitu Algoritma UCS menjamin optimalitas dengan memprioritaskan pengunjungan simpul berbiaya minimum, namun waktu eksekusinya mungkin lebih lama terutama jika terdapat banyak simpul dengan biaya rendah. Algoritma GBFS tidak menjamin optimalitas karena hanya mempertimbangkan heuristik lokal, sehingga dapat menemukan solusi cepat tetapi tidak optimal secara global. Algoritma A* menjamin optimalitas dengan waktu eksekusi yang relatif efisien, tetapi membutuhkan lebih banyak memori karena menyimpan informasi lebih lengkap daripada UCS dan GBFS.

5.2 Saran

Beberapa saran yang dapat kami berikan dari pengerjaan tugas kecil 3 ini adalah sebagai berikut,

1. Perlu dilakukan pendalaman dan pemahaman terlebih dahulu terhadap bahasa java dan tool kit GUI yang digunakan.
2. Perlu dilakukan pendalaman dan pemahaman terlebih dahulu terhadap algoritma yang dipakai (ucs, gbfs, a*)

DAFTAR PUSTAKA

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

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

https://www.javatpoint.com/java-swing

LAMPIRAN

LINK REPOSITORY

Link repository GitHub : https://github.com/Andhikafdh/Tucil3_13522128

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	✓	