LAPORAN TUGAS KECIL 03
IF2211 STRATEGI ALGORITMA

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

Disusun oleh:
Mesach Harmasendro 13522117

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA (STEI)
INSTITUT TEKNOLOGI BANDUNG

2024

DAFTAR ISI

ANALISIS
ALGORITMA

A. Algortima UCS

Uniform Cost Search (UCS) adalah sebuah metode dalam ilmu komputer untuk menemukan jalur terpendek dari titik awal ke titik tujuan pada graf berbobot. UCS beroperasi dengan prinsip yang sederhana yaitu selalu memperluas node dengan "biaya kumulatif terendah" dari titik awal hingga saat itu. Dalam algoritma ini, setiap cabang atau edge memiliki bobot tertentu, dan tujuan dari UCS adalah untuk meminimalkan total biaya jalur. UCS menggunakan struktur data antrian prioritas untuk menyimpan semua node yang belum dikunjungi berdasarkan biaya jalur mereka secara ascending, dan memproses node dengan biaya terendah terlebih dahulu. Hal ini membuat UCS ideal untuk pencarian pada graf yang bobot edgenya tidak negatif dan memastikan bahwa jalur yang ditemukan adalah yang optimal. Selain itu, UCS tidak memerlukan informasi heuristik yang sering kali diperlukan oleh algoritma pencarian jalur lain seperti A*.

Dalam solver permainan word ladder ini setiap kata akan mempunyai beberapa kata-kata lain yang terhubung dengannya. Kata-kata tersebut adalah kata-kata lain yang bermakna yang bisa didapatkan dari mensubstitusikan satu huruf pada sebuah kata. Secara sederhana bisa dikatakan bahwa kata-kata yang saling terhubung adalah kata-kata bermakna yang hanya mempunyai satu perbedaan huruf.

Dalam pembuatan solver permainan word ladder ini, dengan menggunakan algortima UCS akan digunakan struktur data priority queue yang digunakan untuk mempermudah proses penentuan node atau kata dalam graf yang akan dieksplorasi selanjutnya (bobot nilai terkecil). Berikut adalah langkah-lahkah penerapan algortima UCS dalam penyelesaian permaianna word ladder:

1. Siapkan beberapa struktur data yang diperlukan seperti priority queue, hash set yang berperan sebagai kamus bahasa, dan map yang berperan sebagai pencatat status apakah sebuah node sudah dikunjungi atau tidak dan sebagai pencatat nilai (cost) ketika suatu node atau kata tersebut dikunjungi. Dibuat juga sebuah struktur data baru yaitu Node yang berisi word, parent, dan cost. Priority queue

yang dibuat akan menampung Node dan akan mengurutkan Node berdasakran nilai cost terkecil.

2. Setelah semuanya siap masukan Node kata pertama ke dalam queue. Cost untuk kata pertama adalah 0. Update map visited untuk kata pertama menjadi sudah dikunjungi dan simpan juga cost saat ini pada map tersebut yaitu 0.
3. Jika queue tidak kosong maka ambil Node yang berada paling depan pada queue. Cek apakah Node tersebut merupakan kata tujuan atau tidak jika benar maka jawaban ditemukan dan proses UCS akan langsung berakhir. Jika tidak maka Node tersebut akan di ekspansi dan akan diperoleh semua anaknya (kata-kata bermakna yang mempunyai perbedaan satu huruf denganya). Proses ekspansi kata atau pencarian anak akan memanfaatkan hash set kamus untuk mengecek apakah kata tersebut memang ada dan bermakna atau tidak.
4. Data semua anak tersebut kemudian akan diiterasi satu per satu pada setiap iterasi akan dicek apakah kata tersebut sudah pernah dikunjungi atau belum jika belum maka kata tersebut akan dimasukkan kedalam queue dengan cost ditambah satu dari cost parentnya dan dengan data parentnya juga disimpan pada Node. Kemudian status visited nya beserta costnya akan dicatat juga pada map dikunjungi. Jika ternyata kata tersebut sudah pernah dikunjungi dan nilai cost yang disimpan pada map visted lebih kecil dari cost terbaru (cost dari parent saat ini ditambah satu).maka kata tersebut akan diabaikan. Namun jika ternyata nilai cost yang disimpan pada map visited lebih besar dibandingkan nilai cost terbaru maka data pada map visited akan diupdate menjadi nilai cost terbaru dan kata tersebut akan dimasukkan kembali ke dalam queue dengan nilai cost terbaru dan data parent saat ini disimpan juga pada Node.
5. Ulangi langkah 3 dan 4 hingga jawaban ditemukan atau hingga queue kosong. Jika jawaban belum ditemukan hingga queue kosong maka berarti solusi dari permainam word ladder tersebut memang tidak ada. Sebelum proses UCS ini dilakukan juga dilakukan pengecekan terlebih dahulu pada kata mulai dan kata tujuan yang diberikan. Jika panjang kata berbeda dan kedua kata tersebut ternyata tidak bermakna (tidak ada dalam kamus) maka proses UCS akan langsung digagalkan.

Dalam kasus word ladder bobot nilai antara dua kata yang saling terhubung dalam graf akan selalu bernilai satu yang menandakan berapa banyak huruf yang berbeda di antara dua buah kata. Bobot akan selalu bernilai satu karena memang dalam proses ekspansi kata hanya kata dengan satu huruf berbeda saja yang akan dipilih untuk menjadi anak dari suatu kata asal. Proses ekspansi dilakukan seperti itu (memilih kata dengan perbedaan satu huruf saja) untuk menyesuaikan dengan aturan dari permainan word ladder dimana pada setiap langkahnya hanya diperbolehkan mengubah satu huruf saja.

Dalam setiap algoritma UCS pasti akan terdapat fungsi evaluasi yang biasa disebut g(n). Fungsi g(n) akan menunjukkan total cost dari node asal ke suatu node n. Pada kasus word ladder ini g(n) akan sama dengan jumlah substitusi huruf yang perlu dilakukan untuk mengubah kata asal menjadi kata tujuan. Dalam word ladder satu langkah akan sama dengan satu substitusi huruf yang berarti bahwa fungsi g(n) juga bisa digunakan untuk menandakan berapa banyak langkah yang diperlukan untuk memperoleh solusi dari kata asal ke kata tujuan. Tujuan dari permainan word ladder adalah untuk mengubah suatu kata menjadi kata lain dengan jumlah langkah seminimal mungkin.Dikarenakan algoritma UCS selalu mencari solusi dengan cost paling minimal dan dalam kasus ini jumlah cost akan sama dengan jumlah langkah maka bisa dipastikan bahwa algortima UCS disini pasti akan menemukan solusi yang paling optimal (jumlah langkah paling sedikit) untuk permainan word ladder.

Seperti yang sudah dijelaskan di atas besar bobot antara dua kata dalam graf akan selalu bernilai satu, maka bisa dikatakan bahwa graf yang dibentuk pada algortima UCS ini bukanlah graf berbobot melainkan hanya graf tidak berbobot biasa. Dikarenakan graf yang digunakan adalah graf tidak berbobot maka sebenarnya algoritma UCS yang digunakan disini tidak ada bedanya dengan algortima BFS biasa dimana cost yang digunakan pada algoritma UCS ini sama dengan istilah kedalaman yang digunakan dalam BFS. Algortima UCS akan selalu memproses Node dengan cost paling sedikit terlebih dahulu sedangkan algortima BFS akan selalu memproses Node dengan kedalaman terkecil terlebih dahulu. Dikarenakan pada kasus ini cost sama dengan kedalaman maka bisa dikatakan bahwa proses pembangunan Node dan proses mengunjungi Node yang dilakukan pada algoritma UCS secara umum akan sama dengan proses yang terjadi di

algoritma BFS biasa. Solusi yang ditemukan diantara algoritma UCS dan BFS juga mempunyai kemungkinan tinggi untuk sama.

B. Algortima Greedy Best First Search

Greedy Best-First Search (GBFS) adalah algoritma pencarian yang menggunakan heuristik untuk memandu proses pencariannya menuju target dengan cara yang efisien. Berbeda dengan algoritma pencarian yang memeriksa setiap kemungkinan jalur secara menyeluruh, GBFS memilih untuk mengembangkan node atau jalur yang paling menjanjikan pada setiap langkah berdasarkan heuristik tertentu, biasanya jarak terdekat atau biaya terendah yang diperkirakan menuju tujuan. Heuristik ini memberikan estimasi tentang seberapa dekat setiap node berikutnya ke tujuan akhir, sehingga algoritma cenderung mengikuti jalur yang tampak paling menjanjikan tanpa mempertimbangkan keseluruhan biaya. Karena sifatnya yang 'serakah' atau 'greedy', algoritma ini bisa sangat cepat, tetapi tidak selalu menjamin penemuan jalur terpendek yang optimal, terutama di ruang pencarian yang kompleks atau ketika heuristiknya tidak admissible (tidak pernah overestimate true cost).

Dalam Greedy BFS akan dikenal istilah fungsi evaluasi atau fungsi heuristik yang biasa dituliskan sebagai h(n). Fungsi h(n) akan menunjukkan nilai atau cost dari suatu Node n ke Node tujuan. Semakin kecil nilai h(n) maka bisa diartikan bahwa suatu Node n tersebut lebih dekat dengan Node tujuan. Seperti namanya, nilai yang didapatkan dari fungsi h(n) buukanlah nilai pasti tetapi hanya suatu nilai prediksi atau perkiraan yang digunakan untuk membatasi proses pencarian.

Dalam kasus word ladder ini fungsi heuristik yang digunakan adalah fungsi yang menghitung hamming distance antara dua buah kata atau Node. Hamming distance bisa diartikan sebagai banyak huruf berbeda yang terdapat di antara dua buah kata dengan panjang yang sama. Dalam Greedy BFS juga akan digunakan priority queue dimana Node di dalam queue akan diurutkan secara ascending berdasarkan nilai hamming distance dari suatu Node ke Node tujuan.

Berikut adalah langkah-langkah dari algoritma Greddy BFS yang digunakan:

1. Siapkan beberapa struktur data yang diperlukan seperti priority queue, hash set yang berperan sebagai kamus bahasa, dan hash set kedua yang berperan sebagai pencatat status apakah sebuah node sudah dikunjungi atau tidak. Dibuat juga sebuah struktur data baru yaitu Node yang berisi word, parent, dan cost. Priority queue yang dibuat akan menampung Node dan akan mengurutkan Node berdasarkan nilai cost terkecil.
2. Setelah semuanya siap masukan Node kata pertama ke dalam queue. Hitung cost untuk kata pertama dengan menggunakan fungsi hamming distance yang sudah dibuat sebelumnya. Cost dan parent dari kata pertama (parentnya NULL) akan disimpan juga dalam Node. Masukan kata pertama ke dalam hash set dikunjungi untuk menandaka bahwa kata pertama sudah pernah dikunjungi.
3. Jika queue tidak kosong maka ambil Node yang berada paling depan pada queue. Cek apakah Node tersebut merupakan kata tujuan atau tidak jika benar maka jawaban ditemukan dan proses Greedy BFS akan langsung berakhir. Jika tidak maka Node tersebut akan di ekspansi dan akan diperoleh semua anaknya (kata-kata bermakna yang mempunyai perbedaan satu huruf denganya). Proses ekspansi kata atau pencarian anak akan memanfaatkan hash set kamus untuk mengecek apakah kata tersebut memang ada dan bermakna atau tidak.
4. Data semua anak tersebut kemudian akan diiterasi satu per satu pada setiap iterasi akan dicek apakah kata tersebut sudah pernah dikunjungi atau belum jika belum maka kata tersebut akan dimasukkan kedalam queue dengan cost merupakan hamming distance antar kata itu dengan kata tujuan dan dengan data parentnya juga disimpan pada Node. Kemudian, kata tersebut juga akan dimasukkan kedalam hash set visited untuk menandakan bahwa kata tersebut sudah pernah dikunjungi.
5. Ulangi langkah 3 dan 4 hingga jawaban ditemukan atau hingga queue kosong. Jika jawaban belum ditemukan hingga queue kosong maka berarti solusi dari permainam word ladder tersebut memang tidak ada. Sebelum proses Greedy BFS ini dilakukan juga dilakukan pengecekan terlebih dahulu pada kata mulai dan kata tujuan yang diberikan. Jika panjang kata berbeda dan kedua kata tersebut ternyata tidak bermakna (tidak ada dalam kamus) maka proses UCS akan langsung digagalkan.

Dalam konteks permainan word ladder, Greedy BFS yang menggunakan Hamming Distance sebagai heuristik biasanya tidak menjamin solusi yang optimal. Ini karena Greedy BFS berfokus pada node yang memiliki nilai heuristik terendah pada setiap langkahnya tanpa mempertimbangkan biaya total yang telah dikeluarkan untuk mencapai node tersebut. Hamming Distance mengukur jumlah perbedaan huruf antara kata saat ini dan kata tujuan, yang merupakan cara yang baik untuk memperkirakan seberapa dekat suatu kata dengan kata tujuan. Namun, pendekatan ini mungkin mengarahkan pencarian ke jalur yang tampak menjanjikan awalnya tetapi berakhir lebih panjang atau lebih rumit dibandingkan dengan solusi optimal.

Greedy BFS cenderung efisien dalam menemukan solusi dengan cepat karena fokusnya yang kuat pada pencapaian tujuan tanpa mempertimbangkan biaya yang ditempuh. Namun, ini juga menjadi kelemahannya karena bisa terjebak dalam kondisi dimana ia mengikuti jalur yang tampak menarik tetapi pada akhirnya tidak efisien. Dalam banyak kasus, Greedy BFS akan menemukan solusi, tetapi tidak selalu yang terpendek atau yang paling efisien dari segi jumlah langkah.

C. Algoritma A*

Algoritma A* adalah algoritma pencarian jalur yang canggih dan efisien, sering digunakan dalam pemrograman komputer untuk menemukan jalur terpendek antara titik awal dan tujuan dalam graf. Algoritma ini menggunakan fungsi penilaian, f(n) = g(n) + h(n), di mana g(n) adalah biaya dari titik awal ke node n, dan h(n) adalah heuristik yang memperkirakan biaya terendah dari node n ke tujuan. Heuristik harus admissible, yaitu tidak pernah melebih-lebihkan biaya sebenarnya untuk mencapai tujuan, agar A* dapat menjamin solusi optimal. Algoritma A* secara efektif menyeimbangkan antara pencarian greedy yang berusaha segera mencapai tujuan dan metode yang lebih hati-hati yang mempertimbangkan biaya total perjalanan.

Dalam kasus word ladder algoritma A* yang dibuat akan menggunakan fungsi evaluasi yang merupakan gabungan dari fungsi-fungsi evaluasi yang digunakan pada algoritma UCS dan Greedy BFS. Fungsi g(n) yang digunakan akan

sama dengan fungsi g(n) pada algoritma UCS yang menandakan jumlah langkah yang diperlukan untuk mengubah kata asal menjadi suatu kata pada Node n. Fungsi h(n) yang digunakan juga akan sama dengan fungsi h(n) pada algoritma Greddy BFS yang menghitung hamming distance antar suatu kata pada Node n dengan kata tujuan. Priority queue yang digunakan pada algoritma A* ini akan mengurutkan Node berdasarkan nilai total cost (g(n) + h(n)) secara ascending.

Berikut adalah langkah-langkah dari algortima A* yang digunakan:

1. Siapkan beberapa struktur data yang diperlukan seperti priority queue, hash set yang berperan sebagai kamus bahasa, dan map yang berperan sebagai pencatat status apakah sebuah node sudah dikunjungi atau tidak dan sebagai pencatat nilai g(n) dari suatu kata ketika suatu node atau kata tersebut dikunjungi. Dibuat juga sebuah struktur data baru yaitu Node yang berisi word, parent, dan cost. Priority queue yang dibuat akan menampung Node dan akan mengurutkan Node berdasakran nilai cost terkecil.
2. Setelah semuanya siap masukan Node kata pertama ke dalam queue. Nilai g(n) untuk kata pertama adalah 0 dan akan disimpan pada map visited. Nilai cost yang disimpan pada Node adalah nilai g(n) ditambah dengan nilai dari hamming distance (h(n)) yang diperoleh.
3. Jika queue tidak kosong maka ambil Node yang berada paling depan pada queue. Cek apakah Node tersebut merupakan kata tujuan atau tidak jika benar maka jawaban ditemukan dan proses A* akan langsung berakhir. Jika tidak maka Node tersebut akan di ekspansi dan akan diperoleh semua anaknya (kata-kata bermakna yang mempunyai perbedaan satu huruf denganya). Proses ekspansi kata atau pencarian anak akan memanfaatkan hash set kamus untuk mengecek apakah kata tersebut memang ada dan bermakna atau tidak.
4. Data semua anak tersebut kemudian akan diiterasi satu per satu pada setiap iterasi akan dicek apakah kata tersebut sudah pernah dikunjungi atau belum jika belum maka kata tersebut akan dimasukkan kedalam queue dengan total cost adalah cost g(n) parentnya yang tersimpan pada map visited ditambah satu dan ditambah lagi dengan hamming distance antara kata tersebut dengan kata tujuan. Nilai cost g(n) terbaru kemudian akan dicatat juga pada map dikunjungi. Jika ternyata kata tersebut sudah pernah dikunjungi dan nilai cost g(n) yang disimpan

pada map visited lebih kecil dari cost g(n) terbaru, maka kata tersebut akan diabaikan. Namun jika ternyata nilai cost g(n) yang disimpan pada map visited lebih besar dibandingkan nilai cost g(n) terbaru maka data pada map visited akan diupdate menjadi nilai cost g(n) terbaru dan kata tersebut akan dimasukkan kembali ke dalam queue dengan total cost adalah nilai cost g(n) terbaru ditambah dengan hamming distance antara kata tersebut dengan kata tujuan.

5. Ulangi langkah 3 dan 4 hingga jawaban ditemukan atau hingga queue kosong. Jika jawaban belum ditemukan hingga queue kosong maka berarti solusi dari permainam word ladder tersebut memang tidak ada. Sebelum proses UCS ini dilakukan juga dilakukan pengecekan terlebih dahulu pada kata mulai dan kata tujuan yang diberikan. Jika panjang kata berbeda dan kedua kata tersebut ternyata tidak bermakna (tidak ada dalam kamus) maka proses A* akan langsung digagalkan.

Dalam kasus solver word ladder ini penggunaan fungsi heuristic yang menghitung hamming distance antara dua buah kata sudah admissible karena nilai hamming distance yang diperoleh tidak pernah melebih-lebihkan nilai dari h(n) yang sebenarnya. Nilai dari hamming distance hanya menunjukkan jumlah langkah atau substitusi minimum yang diperlukan untuk mengubah suatu kata menjadi kata tujuan. Pada kenyataannya jumlah langkah tidak selalu sama dengan jumlah substitusi minimum dikarenakan terdapat beberapa kata yang tidak bermakna dan tidak valid yang mengakibatkan jumlah langkah yang diperlukan menjadi lebih banyak. Dikarenakan algortima A* sudah dipastikan admissible dan juga penggunaan fungsi g(n) yang sama dengan UCS maka bisa dipastikan bahwa algortima A* in pasti bisa memberikan solusi yang optimal (langkah paling sedikit).

Algortima UCS dan algortima A* sama-sama bisa memberikan solusi yang optimal namun diantara kedua algortima tersebut algortima A* lah yang paling efisien baik dari segi waktu maupun dari segi memori. Dalam kasus word ladder algortima UCS bertindak seperti algortima BFS biasa yang berarti bahwa semua Node akan dikunjungi dan akan diekspansi hingga menemukan solusi yang tentu saja akan memakan banyak waktu dan memori. Dalam kasus algoritma A* dengan adanya fungsi heuristik jumlah Node yang perlu diproses akan dibatasi dan hanya

Node-node yang terlihat menjanjikan saja yang akan diproses. Dengan semakin sedikitnya jumlah Node yang perlu diproses tentu saja itu akan membuat algoritma A* menjadi lebih efisien baik dari segi waktu dan juga memori.

SOURCE CODE

A. Screnshot Source Code

GUI

```java
package src.GUI;

import javax.swing.*;

import src.Solver.AStarSolver;
import src.Solver.GBFSSolver;
import src.Solver.ReadDict;
import src.Solver.UCSSolver;
import src.Solver.WordLadderSolver;

import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.util.HashSet;
import java.util.List;
import java.util.Vector;

You, 4 minutes ago | 1 author (You)
public class WordLadderGUI extends JFrame {
    private JTextField startWordField;
    private JTextField endWordField;
    private JList<String> resultDisplay;
    private WordLadderSolver ucsSolver;
    private WordLadderSolver greedySolver;
    private WordLadderSolver aStarSolver;
    private JLabel timeExecLabel;
    private JLabel countVisitedLabel;

    public WordLadderGUI() {
        super(title:"Word Ladder Solver");
        initializeSolvers();
        initializeUI();
    }

    private void initializeSolvers() {
        try {

            HashSet<String> dictionary = ReadDict.readDict(filename:"./dictionary/words.txt");
            // Assuming constructors that take a dictionary
            ucsSolver = new UCSSolver(dictionary);
            greedySolver = new GBFSSolver(dictionary);
            aStarSolver = new AStarSolver(dictionary);
        }

        catch (IOException e) {
            // System.out.println(e.getMessage());
            JOptionPane.showMessageDialog(this,
                    "Error loading dictionary: " + e.getMessage(),
                    title:"Error",
                    JOptionPane.ERROR_MESSAGE);
            System.exit(status:1);
        }
    }
```

```
    private void initializeUI() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(width:600, height:600);
        setLocationRelativeTo(c:null);
        setLayout(new BorderLayout());

        JPanel inputPanel = setupInputPanel();
        add(inputPanel, BorderLayout.NORTH);

        JPanel statsPanel = new JPanel(new FlowLayout(FlowLayout.LEFT));
        timeExecLabel = new JLabel(text:"Time Exec: 0 ms");
        countVisitedLabel = new JLabel(text:"Nodes Visited: 0");
        statsPanel.add(timeExecLabel);
        statsPanel.add(countVisitedLabel);
        add(statsPanel, BorderLayout.SOUTH);

        resultDisplay = new JList<>();
        resultDisplay.setFont(new Font(name:"Monospaced", Font.BOLD, size:16));
        JScrollPane scrollPane = new JScrollPane(resultDisplay);
        scrollPane.setBorder(BorderFactory.createTitledBorder(title:"Path"));
        add(scrollPane, BorderLayout.CENTER);

        setVisible(b:true);
    }

    private JPanel setupInputPanel() {
        JPanel inputPanel = new JPanel(new GridBagLayout());
        GridBagConstraints c = new GridBagConstraints();
        c.insets = new Insets(top:10, left:10, bottom:10, right:10);

        addFieldAndLabel(labelText:"Start Word:", startWordField = new JTextField(columns:10), inputPanel, c, row:0);
        addFieldAndLabel(labelText:"End Word:", endWordField = new JTextField(columns:10), inputPanel, c, row:1);
        addButton(text:"Solve with UCS", this::solveWithUCS, inputPanel, c, row:2);
        addButton(text:"Solve with Greedy BFS", this::solveWithGreedyBFS, inputPanel, c, row:3);
        addButton(text:"Solve with A*", this::solveWithAStar, inputPanel, c, row:4);

        return inputPanel;
    }

    private void addFieldAndLabel(String labelText, JTextField field, JPanel panel, GridBagConstraints constraints,
            int row) {
        constraints.gridx = 0;
        constraints.gridy = row;
        panel.add(new JLabel(labelText), constraints);

        constraints.gridx = 1;
        panel.add(field, constraints);
    }

    private void addButton(String text, ActionListener listener, JPanel panel, GridBagConstraints constraints,
            int row) {
        JButton button = new JButton(text);
        button.addActionListener(listener);
        constraints.gridx = 0;
        constraints.gridy = row;
        constraints.gridwidth = 2;
        panel.add(button, constraints);
    }
```

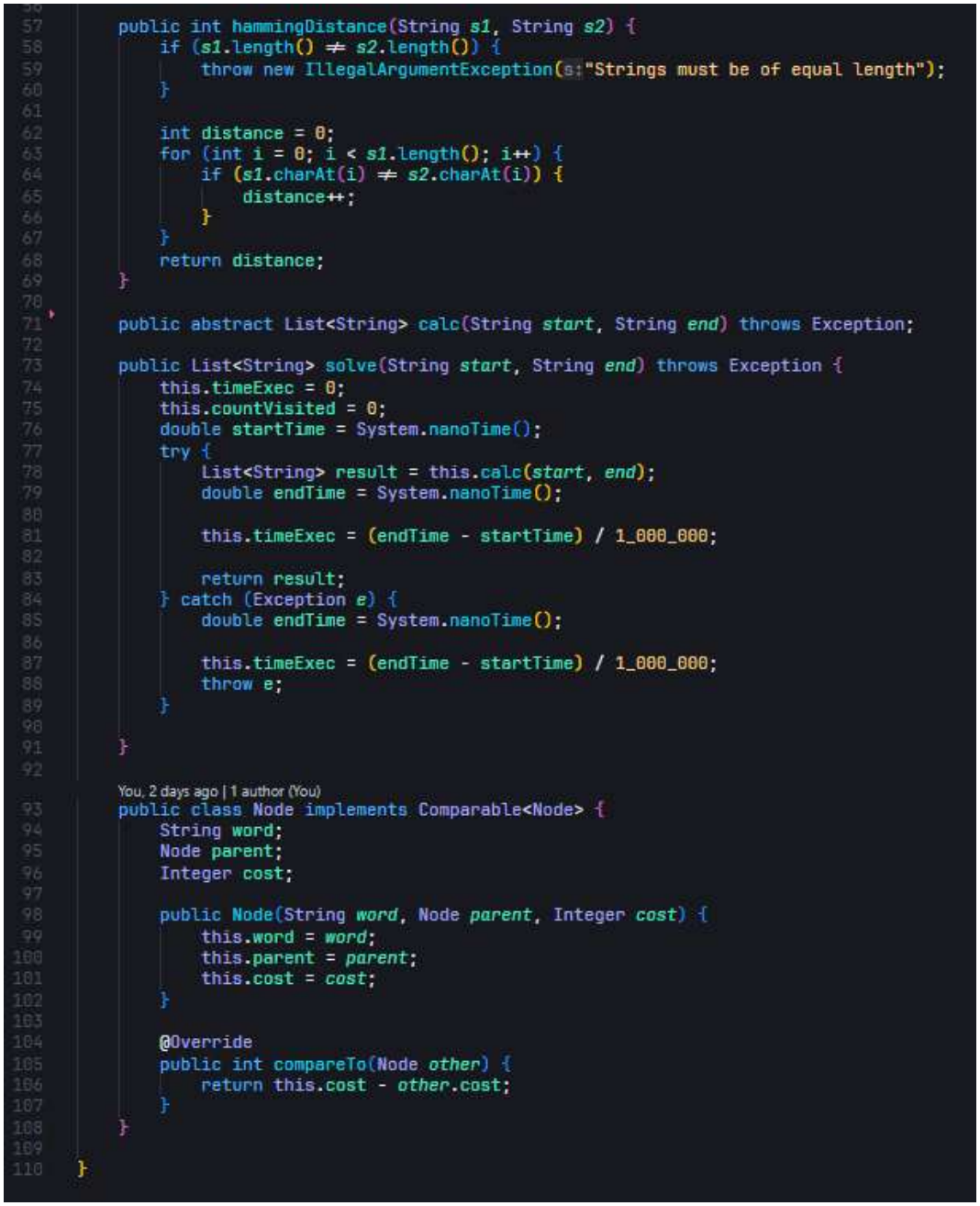
```
    private void solve(WordLadderSolver solver) {
        String start = startWordField.getText().toLowerCase().trim();
        String end = endWordField.getText().toLowerCase().trim();
        try {
            List<String> result = solver.solve(start, end);

            // Formatting results with numbering
            Vector<String> formattedResult = new Vector<>();
            if (result.size() > 0) {
                for (int i = 0; i < result.size(); i++) {
                    formattedResult.add((i + 1) + ". " + result.get(i));
                }
            } else {
                formattedResult.add(e:"No solution found");
            }

            resultDisplay.setListData(formattedResult);
            timeExecLabel.setText("Time Exec: " + solver.getTimeExec() + " ms");
            countVisitedLabel.setText("Nodes Visited: " + solver.getCountVisited());
        } catch (Exception e) {
            JOptionPane.showMessageDialog(this, "Error: " + e.getMessage(), title:"Error", JOptionPane.ERROR_MESSAGE);
        }
    }

    private void solveWithUCS(ActionEvent event) {
        solve(ucsSolver);
    }

    private void solveWithGreedyBFS(ActionEvent event) {
        solve(greedySolver);
    }

    private void solveWithAStar(ActionEvent event) {
        solve(aStarSolver);
    }

}
```

ReadDict

```
You, 2 days ago | 1 author (You)
package src.Solver;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashSet;

You, 2 days ago | 1 author (You)
public class ReadDict {
    public static HashSet<String> readDict(String filename) throws IOException {
        HashSet<String> dict = new HashSet<>();
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = reader.readLine()) != null) {
                dict.add(line.trim());
            }
        }

        return dict;
    }
}
```

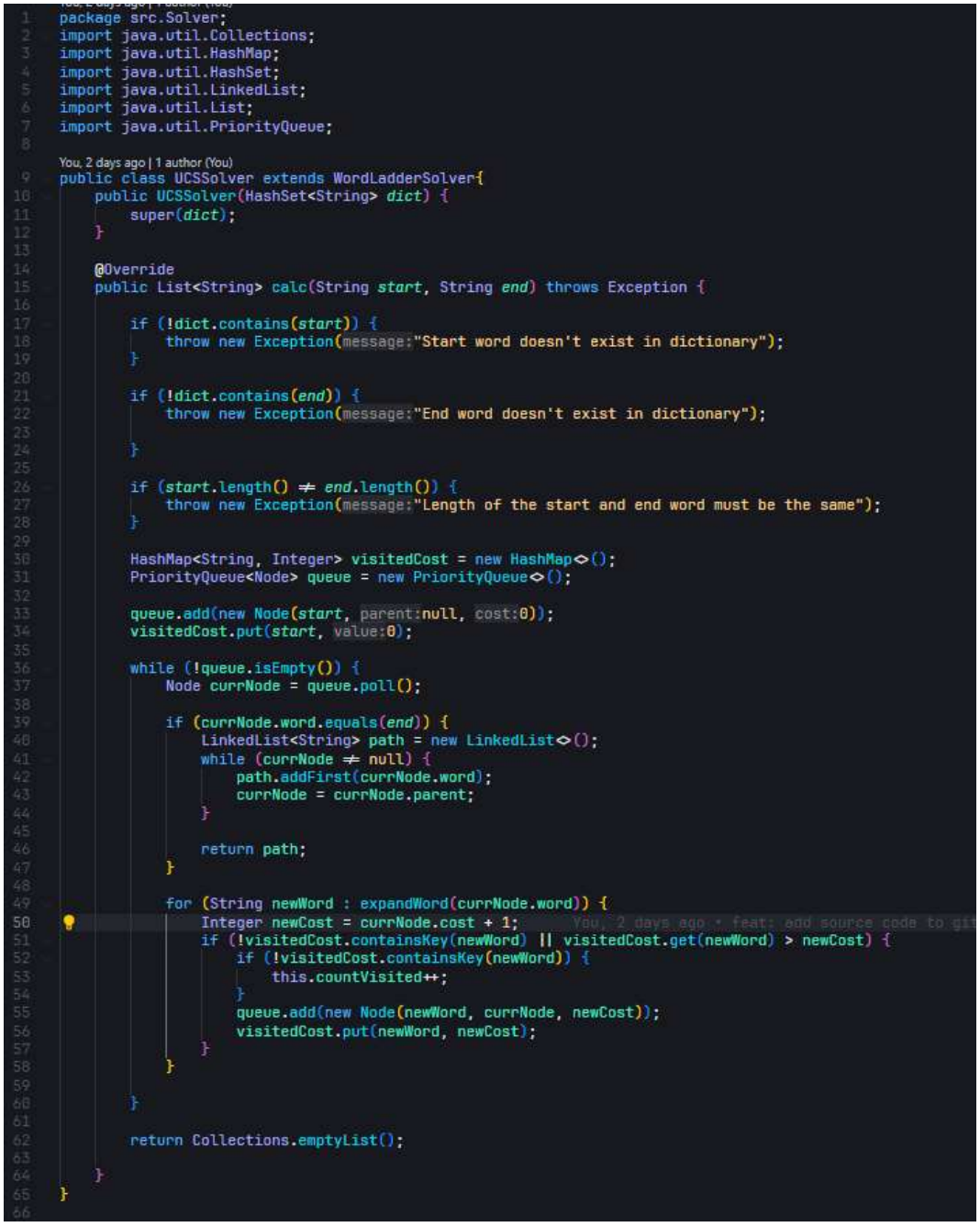
WordLadderSolver

```
You, 2 days ago | 1 author (You)
package src.Solver;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;

You, 2 days ago | 1 author (You)
public abstract class WordLadderSolver {

    protected HashSet<String> dict = new HashSet<>();
    protected double timeExec = 0;
    protected int countVisited = 0;

    public WordLadderSolver(HashSet<String> dict) {
        this.dict = dict;
    }

    public double getTimeExec() {
        return this.timeExec;
    }

    public int getCountVisited() {
        return this.countVisited;      You, 2 days ago • feat: a
    }

    public List<String> expandWord(String word) {
        List<String> newWords = new ArrayList<>();
        char[] chars = word.toLowerCase().toCharArray();

        for (int i = 0; i < chars.length; i++) {
            char oldChar = chars[i];
            for (char c = 'a'; c <= 'z'; c++) {
                if (c != oldChar) {
                    chars[i] = c;
                    String newWord = new String(chars);
                    if (dict.contains(newWord)) {
                        newWords.add(newWord);
                    }
                }
            }
            chars[i] = oldChar;
        }

        return newWords;
    }

    public List<String> buildPath(Node node) {
        LinkedList<String> path = new LinkedList<>();
        while (node != null) {
            path.addFirst(node.word);
            node = node.parent;
        }

        return path;
    }
```



```
    public int hammingDistance(String s1, String s2) {
        if (s1.length() != s2.length()) {
            throw new IllegalArgumentException(s:"Strings must be of equal length");
        }

        int distance = 0;
        for (int i = 0; i < s1.length(); i++) {
            if (s1.charAt(i) != s2.charAt(i)) {
                distance++;
            }
        }
        return distance;
    }

    public abstract List<String> calc(String start, String end) throws Exception;

    public List<String> solve(String start, String end) throws Exception {
        this.timeExec = 0;
        this.countVisited = 0;
        double startTime = System.nanoTime();
        try {
            List<String> result = this.calc(start, end);
            double endTime = System.nanoTime();

            this.timeExec = (endTime - startTime) / 1_000_000;

            return result;
        } catch (Exception e) {
            double endTime = System.nanoTime();

            this.timeExec = (endTime - startTime) / 1_000_000;
            throw e;
        }

    }

    You, 2 days ago | 1 author (You)
    public class Node implements Comparable<Node> {
        String word;
        Node parent;
        Integer cost;

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

        @Override
        public int compareTo(Node other) {
            return this.cost - other.cost;
        }
    }

}
```

UCSSolver

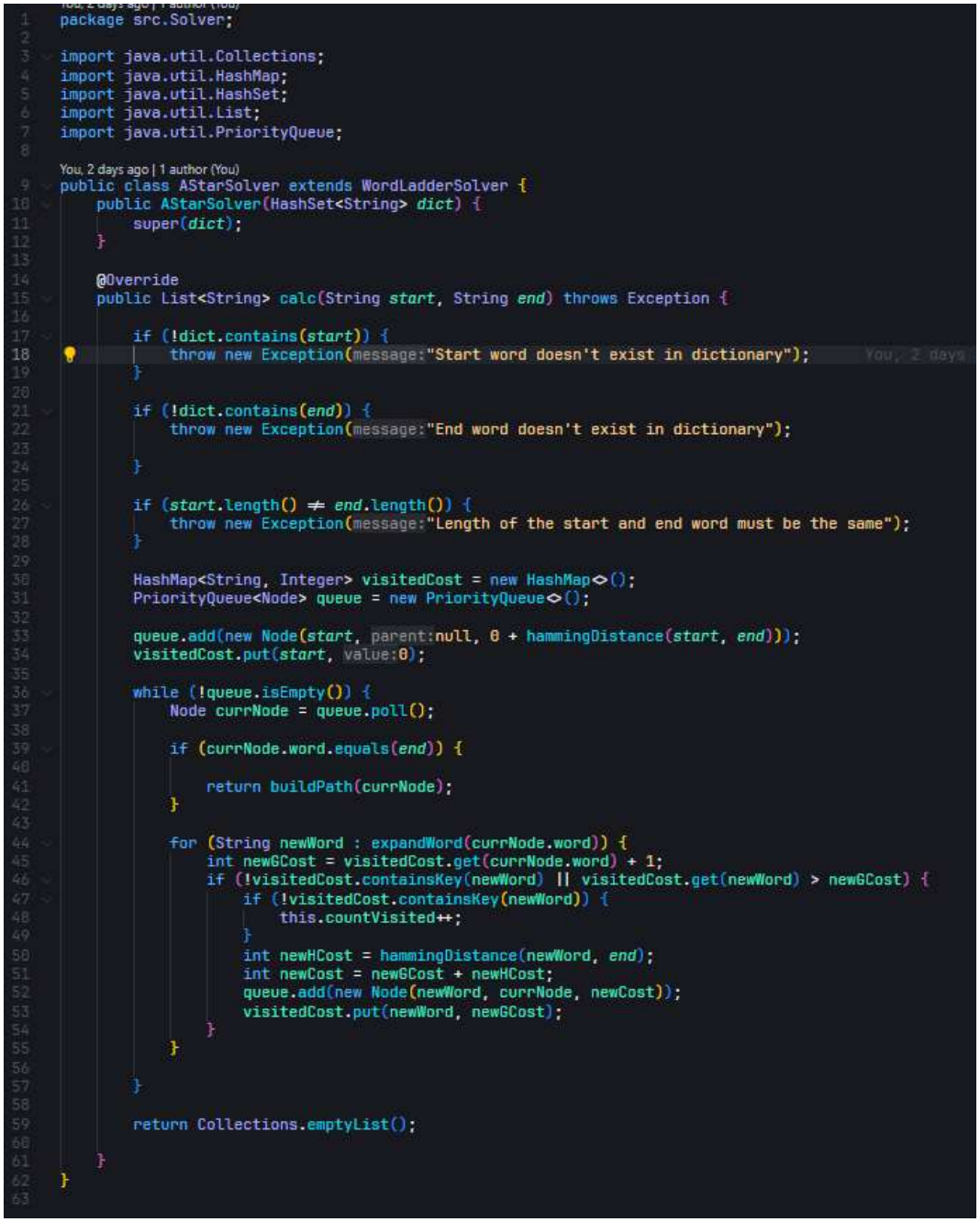


```java
package src.Solver;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.PriorityQueue;

You, 2 days ago | 1 author (You)
public class UCSSolver extends WordLadderSolver{
    public UCSSolver(HashSet<String> dict) {
        super(dict);
    }

    @Override
    public List<String> calc(String start, String end) throws Exception {

        if (!dict.contains(start)) {
            throw new Exception(message:"Start word doesn't exist in dictionary");
        }

        if (!dict.contains(end)) {
            throw new Exception(message:"End word doesn't exist in dictionary");

        }

        if (start.length() != end.length()) {
            throw new Exception(message:"Length of the start and end word must be the same");
        }

        HashMap<String, Integer> visitedCost = new HashMap<>();
        PriorityQueue<Node> queue = new PriorityQueue<>();

        queue.add(new Node(start, parent:null, cost:0));
        visitedCost.put(start, value:0);

        while (!queue.isEmpty()) {
            Node currNode = queue.poll();

            if (currNode.word.equals(end)) {
                LinkedList<String> path = new LinkedList<>();
                while (currNode != null) {
                    path.addFirst(currNode.word);
                    currNode = currNode.parent;
                }

                return path;
            }

            for (String newWord : expandWord(currNode.word)) {
                Integer newCost = currNode.cost + 1;      You, 2 days ago • feat: add source code to gi
                if (!visitedCost.containsKey(newWord) || visitedCost.get(newWord) > newCost) {
                    if (!visitedCost.containsKey(newWord)) {
                        this.countVisited++;
                    }
                    queue.add(new Node(newWord, currNode, newCost));
                    visitedCost.put(newWord, newCost);
                }
            }

        }

        return Collections.emptyList();

    }
}
```

GBFSSolver

```java
package src.Solver;
import java.util.Collections;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;

You, 2 days ago | 1 author (You)
public class GBFSSolver extends WordLadderSolver {
    public GBFSSolver(HashSet<String> dict) {
        super(dict);
    }

    @Override
    public List<String> calc(String start, String end) throws Exception {

        if (!dict.contains(start)) {
            throw new Exception(message:"Start word doen't exist in dictionary");
        }

        if (!dict.contains(end)) {
            throw new Exception(message:"End word doen't exist in dictionary");

        }

        if (start.length() != end.length()) {
            throw new Exception(message:"Length of the start and end word must be the same");
        }

        HashSet<String> visitedCost = new HashSet<>();
        PriorityQueue<Node> queue = new PriorityQueue<>();

        queue.add(new Node(start, parent:null, hammingDistance(start, end)));
        visitedCost.add(start);

        while (!queue.isEmpty()) {
            Node currNode = queue.poll();

            // queue.clear();

            if (currNode.word.equals(end)) {

                return buildPath(currNode);
            }

            for (String newWord : expandWord(currNode.word)) {
                if (!visitedCost.contains(newWord)) {
                    this.countVisited++;
                    // count++;
                    Integer newCost = hammingDistance(newWord, end);
                    queue.add(new Node(newWord, currNode, newCost));
                    visitedCost.add(newWord);
                }
            }      You, 2 days ago • feat: add source code to github

        }

        return Collections.emptyList();

    }
}
```

AstarSolver



```java
package src.Solver;

import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;

You, 2 days ago | 1 author (You)
public class AStarSolver extends WordLadderSolver {
    public AStarSolver(HashSet<String> dict) {
        super(dict);
    }

    @Override
    public List<String> calc(String start, String end) throws Exception {

        if (!dict.contains(start)) {
            throw new Exception(message:"Start word doesn't exist in dictionary");      You, 2 days
        }

        if (!dict.contains(end)) {
            throw new Exception(message:"End word doesn't exist in dictionary");

        }

        if (start.length() != end.length()) {
            throw new Exception(message:"Length of the start and end word must be the same");
        }

        HashMap<String, Integer> visitedCost = new HashMap<>();
        PriorityQueue<Node> queue = new PriorityQueue<>();

        queue.add(new Node(start, parent:null, 0 + hammingDistance(start, end)));
        visitedCost.put(start, value:0);

        while (!queue.isEmpty()) {
            Node currNode = queue.poll();

            if (currNode.word.equals(end)) {

                return buildPath(currNode);
            }

            for (String newWord : expandWord(currNode.word)) {
                int newGCost = visitedCost.get(currNode.word) + 1;
                if (!visitedCost.containsKey(newWord) || visitedCost.get(newWord) > newGCost) {
                    if (!visitedCost.containsKey(newWord)) {
                        this.countVisited++;
                    }
                    int newHCost = hammingDistance(newWord, end);
                    int newCost = newGCost + newHCost;
                    queue.add(new Node(newWord, currNode, newCost));
                    visitedCost.put(newWord, newGCost);
                }
            }

        }

        return Collections.emptyList();

    }
}
```

B. Penjelasan Source Code

Semua kode program dalam tugas ini terletak pada folder src. Di dalam folder src akan terdapat dua folder yaitu GUI dan Solver serta terdapat satu file Main.java yang akan menjadi starting point dari program yang dibuat. Folder GUI berisi file atau class WordLadderGUI yang didalamnya berisi semua kode-kode yang berkaitan dengan tampilan dan funsgionalitas GUI yang dibuat.

Dalam folder Solver berisi kode-kode serta algoritma-algoritma utama yang digunakan untuk menyelesaikan permainan word ladder. Secara umum hanya

terdapat 2 class utama dalam folder ini yaitu ReadDict dan WordLadderSolver. Class ReadDict hanya berisi satu fungsi static readDict yang digunakan untuk membaca file kamus dalam bentuk txt dan mengubahnya ke dalam bentuk hash set. Class WordLadderSolver akan menjadi dasar dari penerapan algoritma-algortima yang akn digunakan. Class WordLadderSolver memiliki 3 buah class anak yaitu AstartSolver, GBFSSolver, dan UCS Solver. Masing-masing dari class anak tersebut memiliki fungsi calc yang berisi penerapan dari masing-masing algortima baik UCS, GBFS, ataupun A*.

Pada class induk WordLadderSolver terdapat beberapa fungsi yang mendukung proses berjalannya masing-masing algoritma. Fungsi pendukung yang pertama adalah fungsi expandWord yang digunakan untuk mencari anak (kata bermakna yang mempunyai perbedaan satu huruf dari kata asal) dari sebuah kata. Ada juga fungsi buildPath yang digunakan untuk membuat jalur solusi dari kata asal hingga ke kata tujuan ketika solusi ditemukan. Fungsi untuk menghitung nilai heuristik juga terdapat pada class ini yang dinamakan dengan fungsi hammingDistance. Dalam class ini juga terdapat class solve yang menjadi starting point untuk menjalankan masing-masing algoritma. Pada fungsi solve ini akan dipanggil fungsi calc untuk menjalankan algoritma pencarian untuk mencari solusi. Dalam fungsi solve ini juga akan diletakkan timer untuk menghitung waktu eksekusi dari algoritma yang digunakan. Selain fungsi-fungsi pendukung juga terdapat fungsi-fungsii getter untuk mendapatkan data dari class WordLadderSolver ini. Hanya ada 2 fungsi getter dalam class ini yaitu untuk fungsi getter untuk mendapatkan waktu eksekusi dan fungsi getter untuk mendapatkan jumlah node yang dikunjungi.

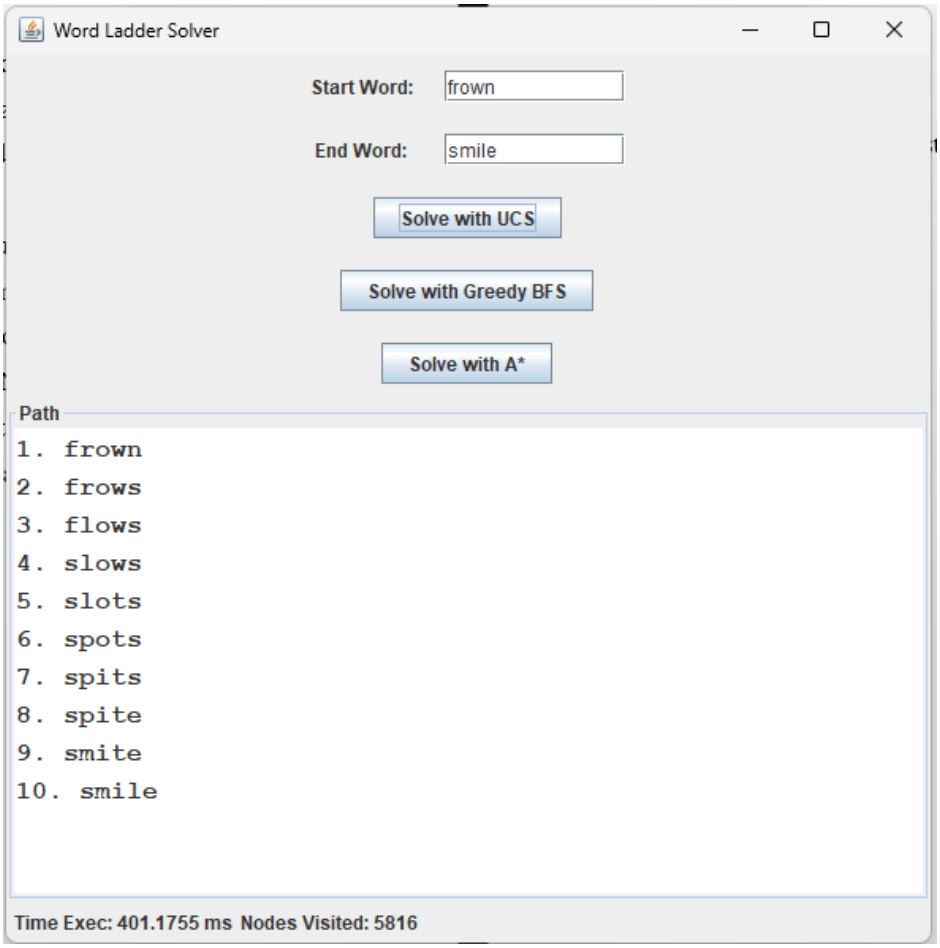
Selain fungsi dalam class WordLadderSolver juga terdapat helper class yang bernama Node yang merepresentasikan node dalam graf. Class Node hanya bertugas sebagai penyimpan data dimana dalam class Node terdapat tiga data yang disimpan yaitu word, parent node, dan juga cost. Class Node ini juga merupakan turunan dari interface Comparable yang mengimplementasikan fungsi compareTo untuk mempermudah priority queue melakukan pengurutan nantinya.

TEST CASE

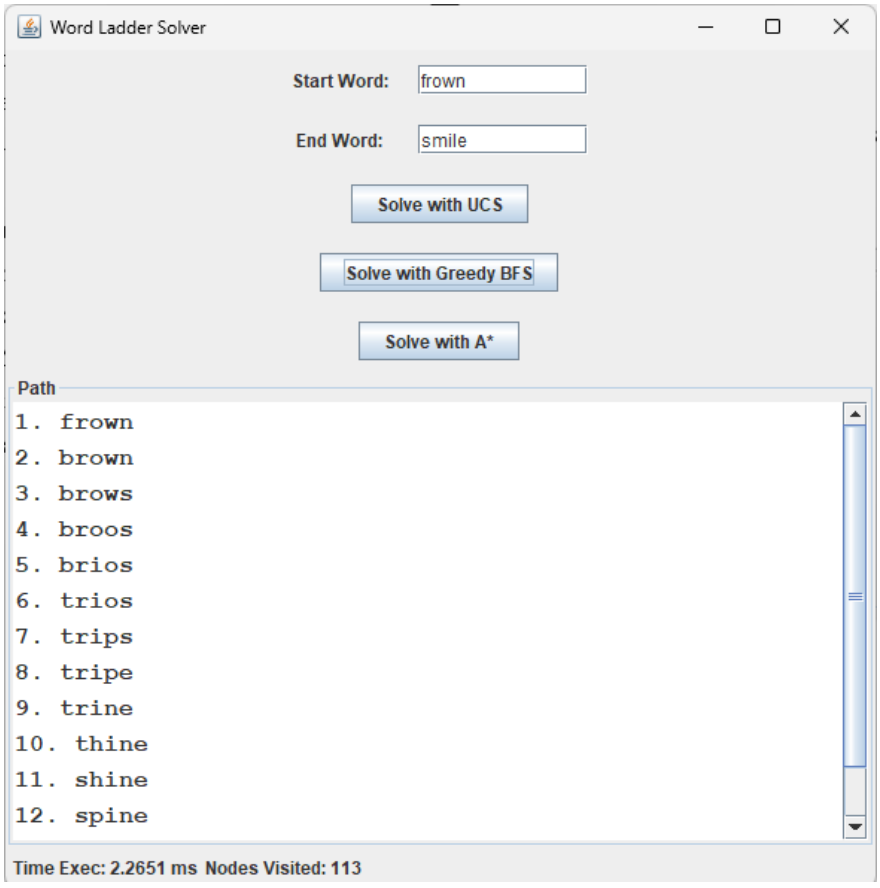
A. Screenshot Test Case

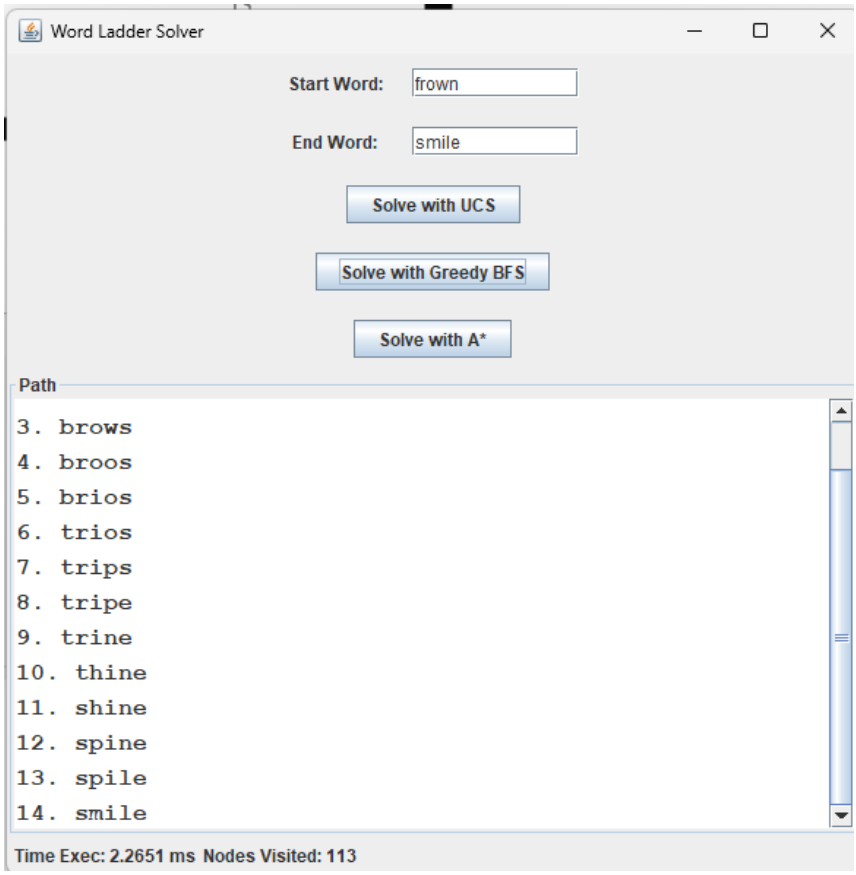
1. Test Case 1

UCS

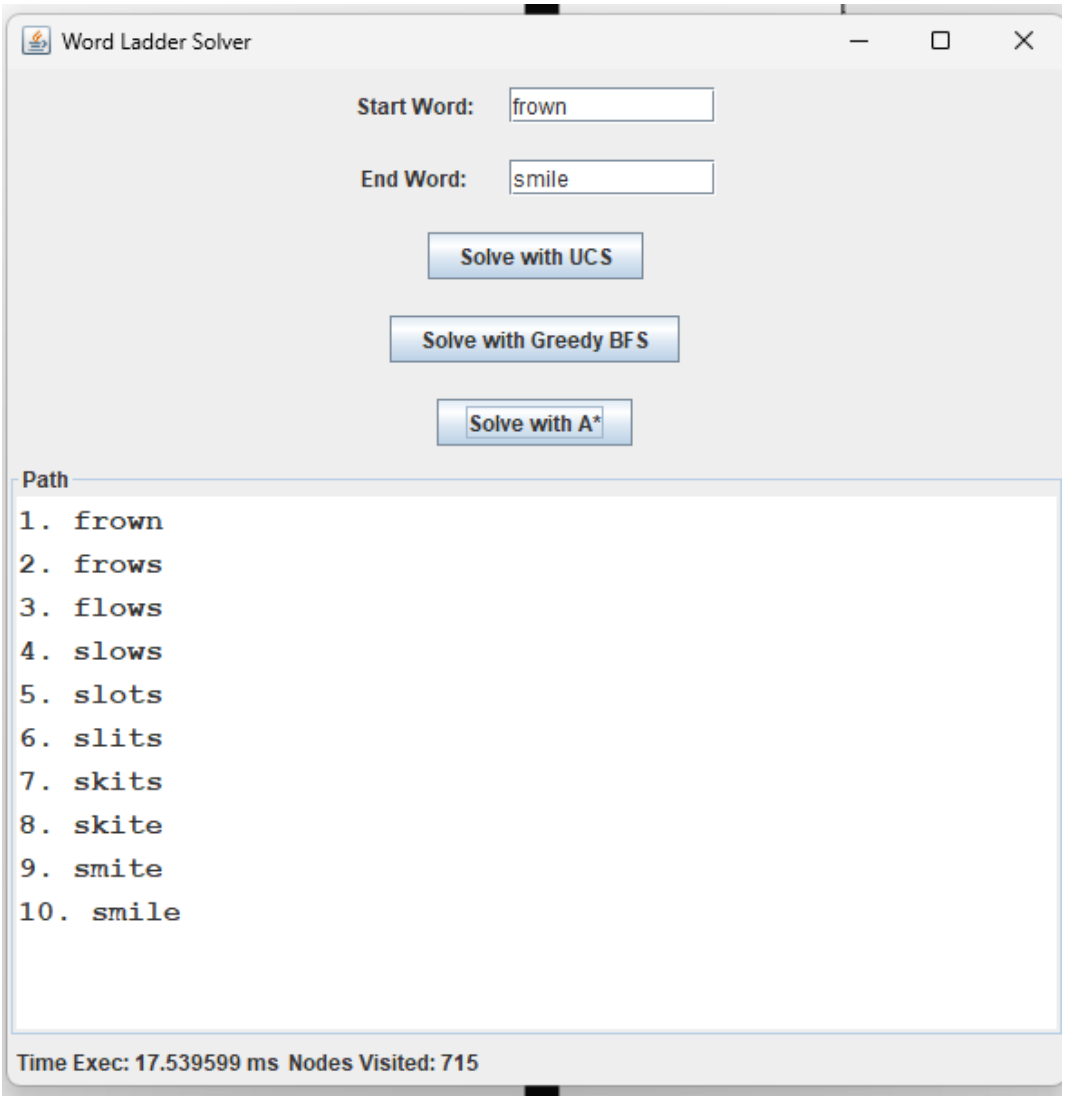


GBFS


Word Ladder Solver
Start Word:
frown
End Word:
smile
Solve with UCS
Solve with Greedy BFS
Solve with A*
Path
1. frown
2. brown
3. brows
4. broos
5. brios
6. trios
7. trips
8. tripe
9. trine
10. thine
11. shine
12. spine
Time Exec: 2.2651 ms Nodes Visited: 113

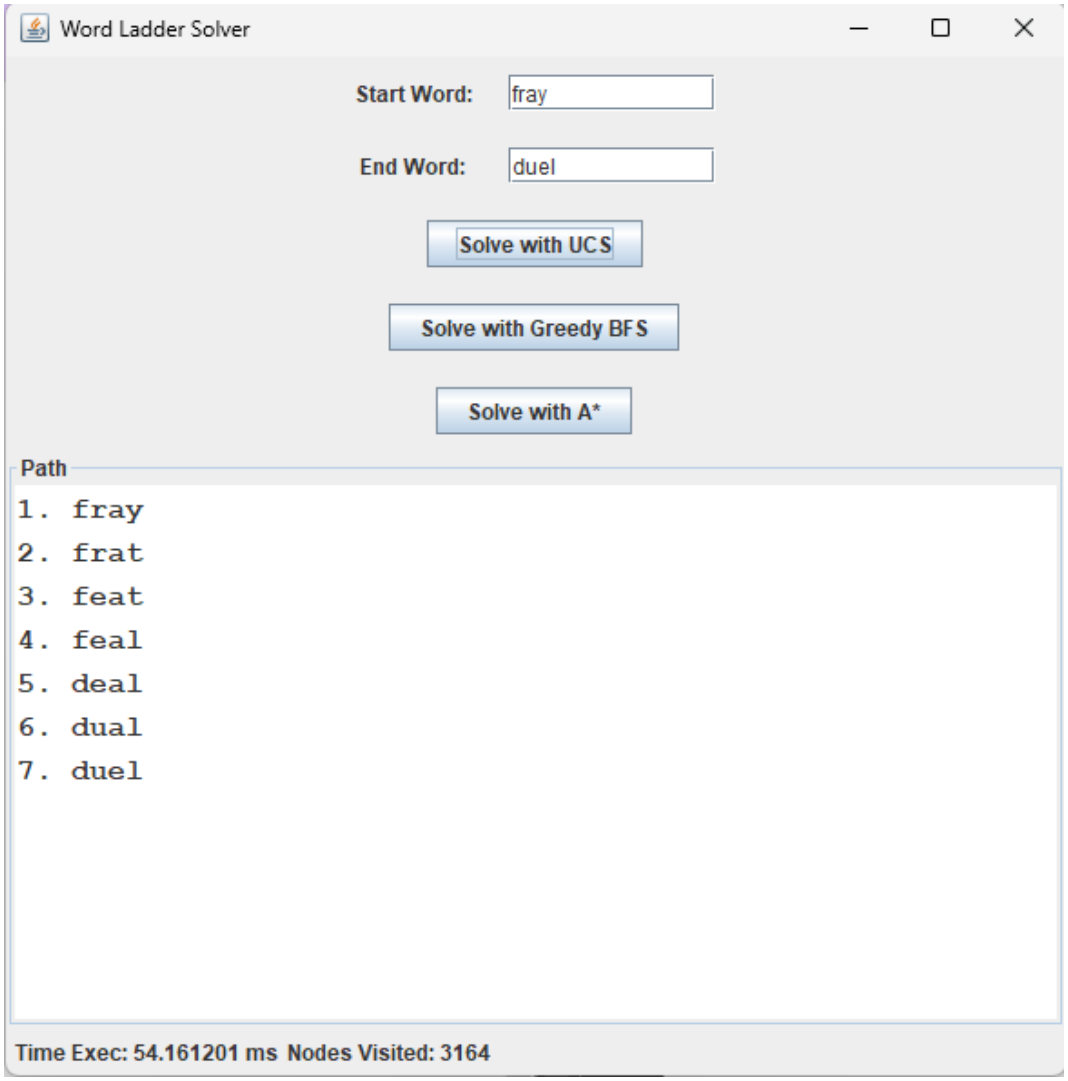

Word Ladder Solver
Start Word:
frown
End Word:
smile
Solve with UCS
Solve with Greedy BFS
Solve with A*
Path
3. brows
4. broos
5. brios
6. trios
7. trips
8. tripe
9. trine
10. thine
11. shine
12. spine
13. spile
14. smile
Time Exec: 2.2651 ms Nodes Visited: 113

A*

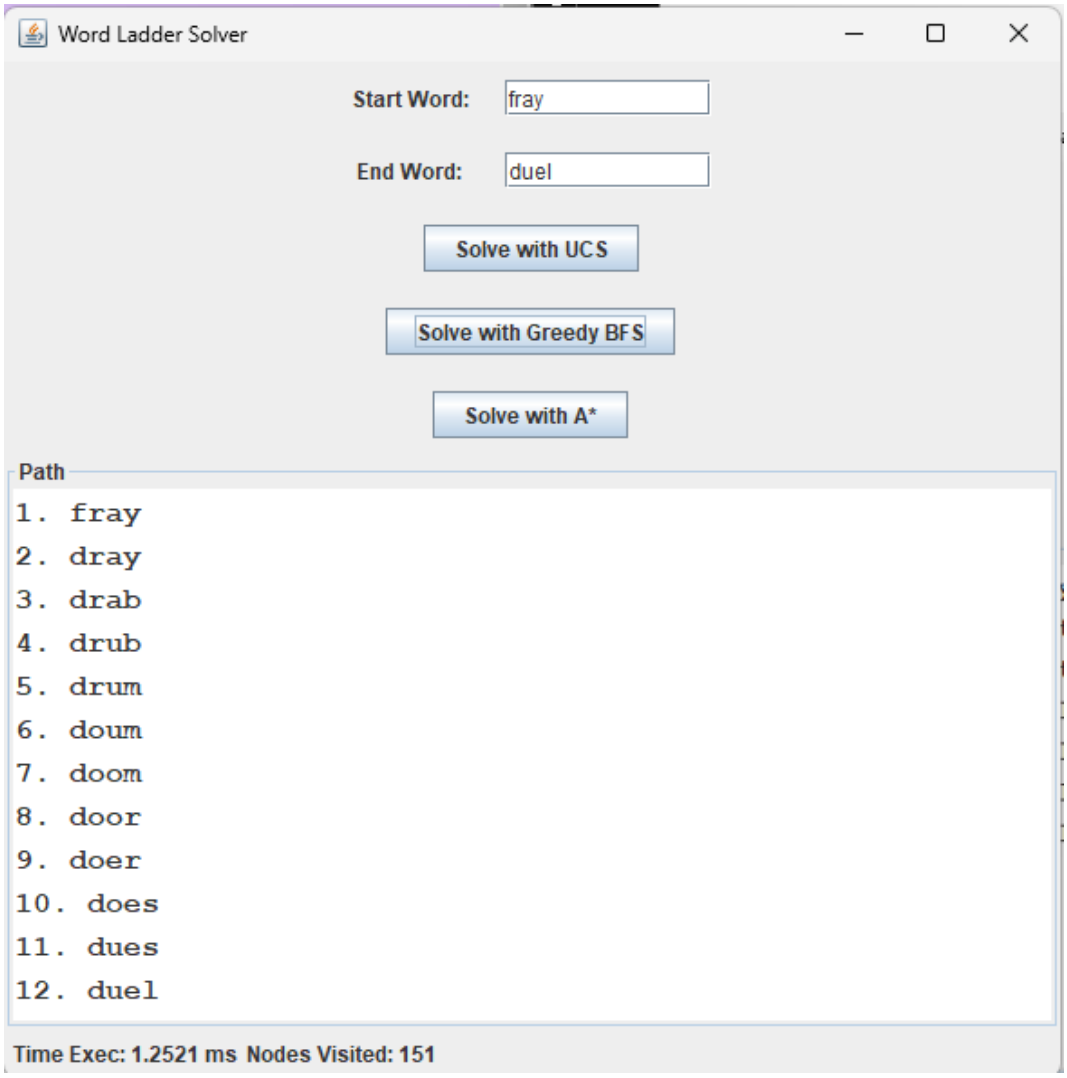


2. Test Case 2

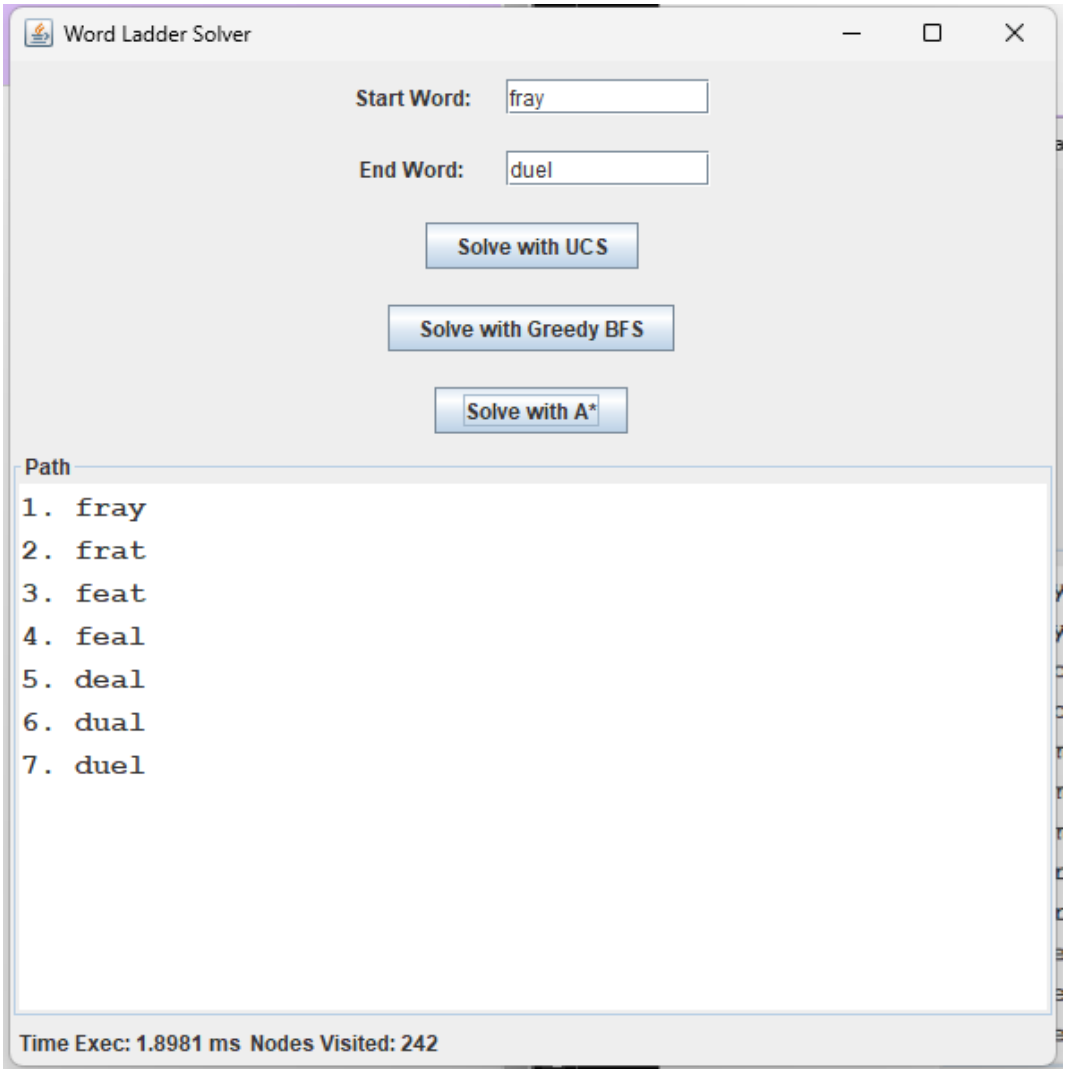
UCS


Word Ladder Solver
Start Word:
fray
End Word:
duel
Solve with UCS
Solve with Greedy BFS
Solve with A*
Path
1. fray
2. frat
3. feat
4. feal
5. deal
6. dual
7. duel
Time Exec: 54.161201 ms Nodes Visited: 3164

GBFS

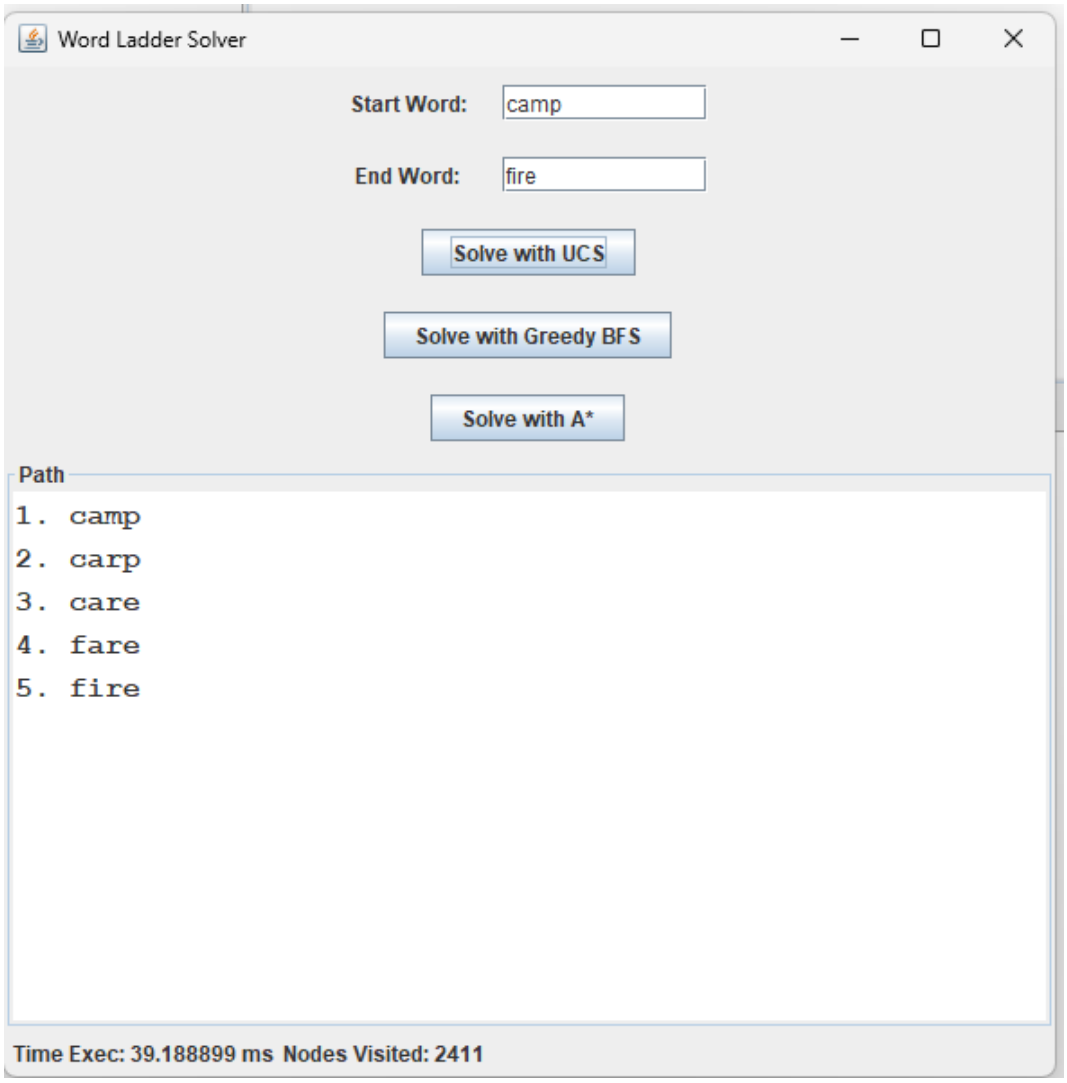

Word Ladder Solver
Start Word:
fray
End Word:
duel
Solve with UCS
Solve with Greedy BFS
Solve with A*
Path
1. fray
2. dray
3. drab
4. drub
5. drum
6. doum
7. doom
8. door
9. doer
10. does
11. dues
12. duel
Time Exec: 1.2521 ms Nodes Visited: 151

A*

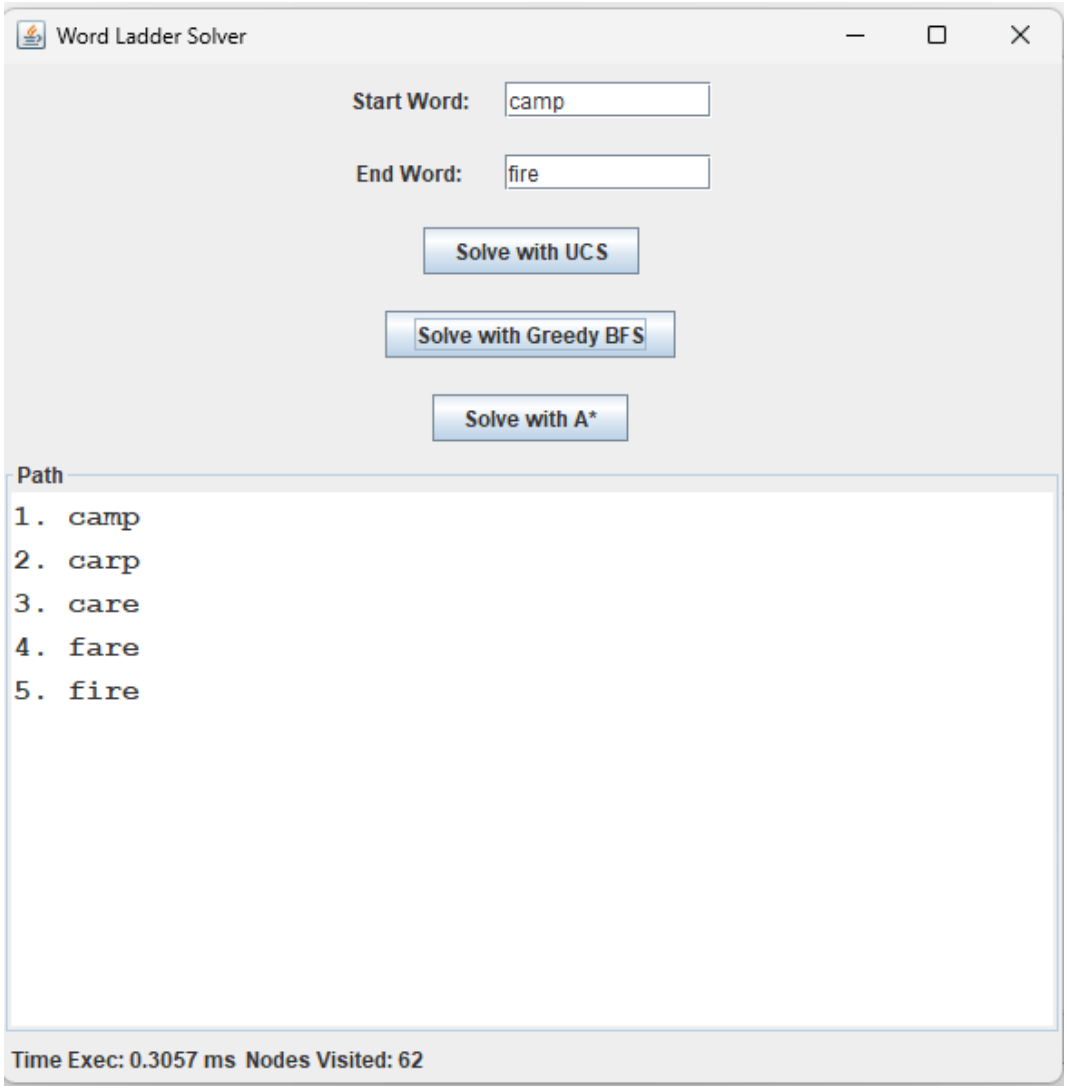


3. Test Case 3

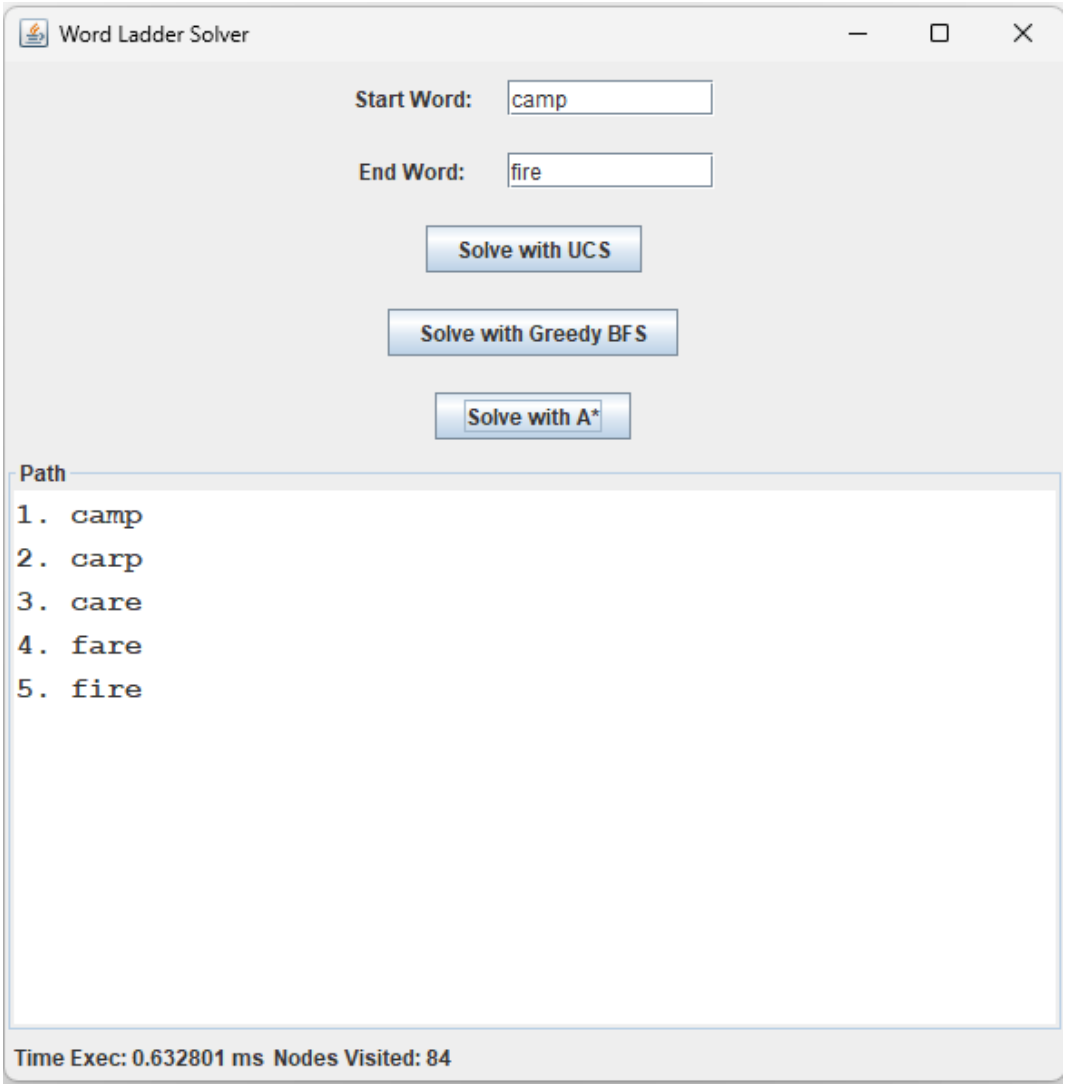
UCS


Word Ladder Solver
Start Word:
camp
End Word:
fire
Solve with UCS
Solve with Greedy BFS
Solve with A*
Path
1. camp
2. carp
3. care
4. fare
5. fire
Time Exec: 39.188899 ms Nodes Visited: 2411

GBFS

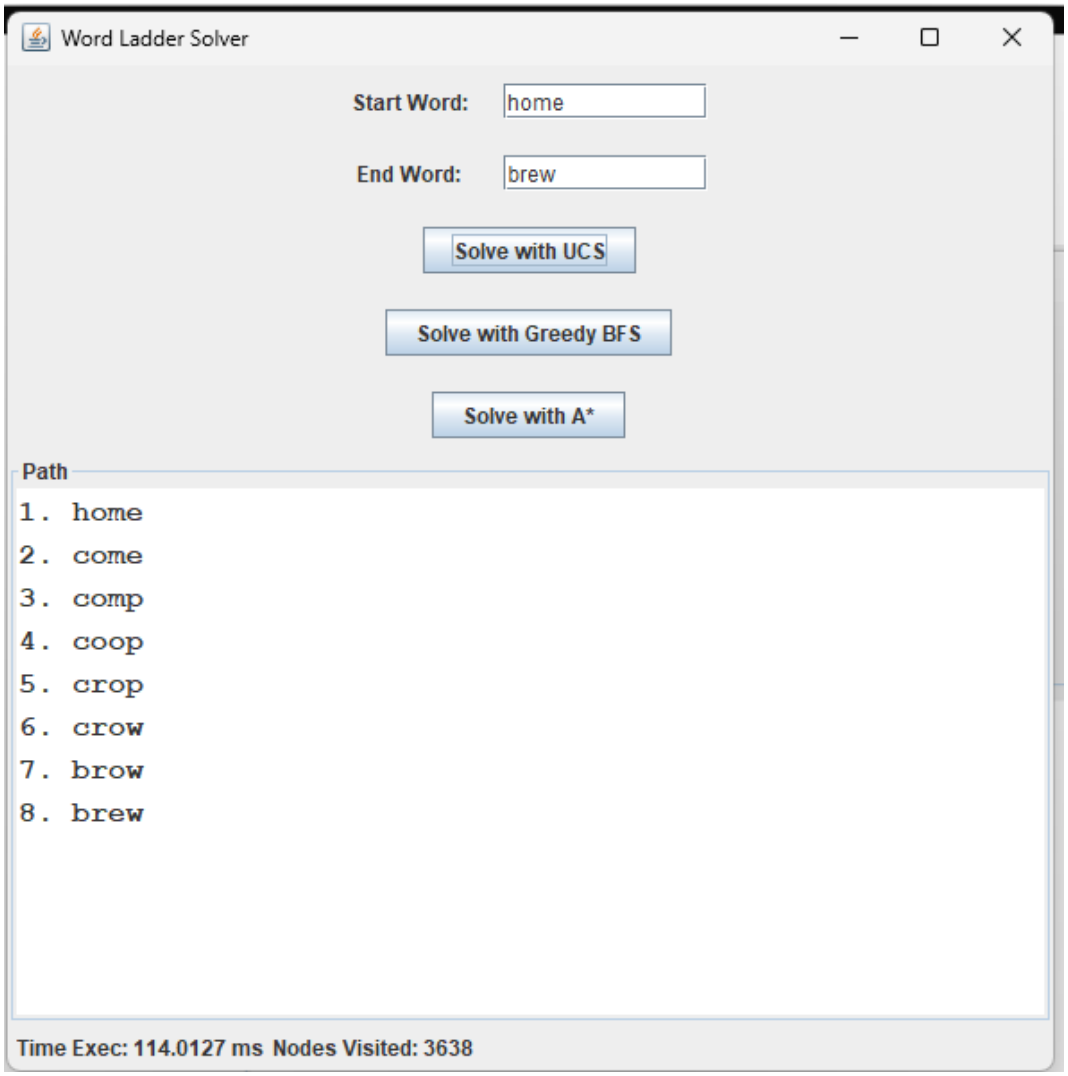

Word Ladder Solver
Start Word:
camp
End Word:
fire
Solve with UCS
Solve with Greedy BFS
Solve with A*
Path
1. camp
2. carp
3. care
4. fare
5. fire
Time Exec: 0.3057 ms Nodes Visited: 62

A*

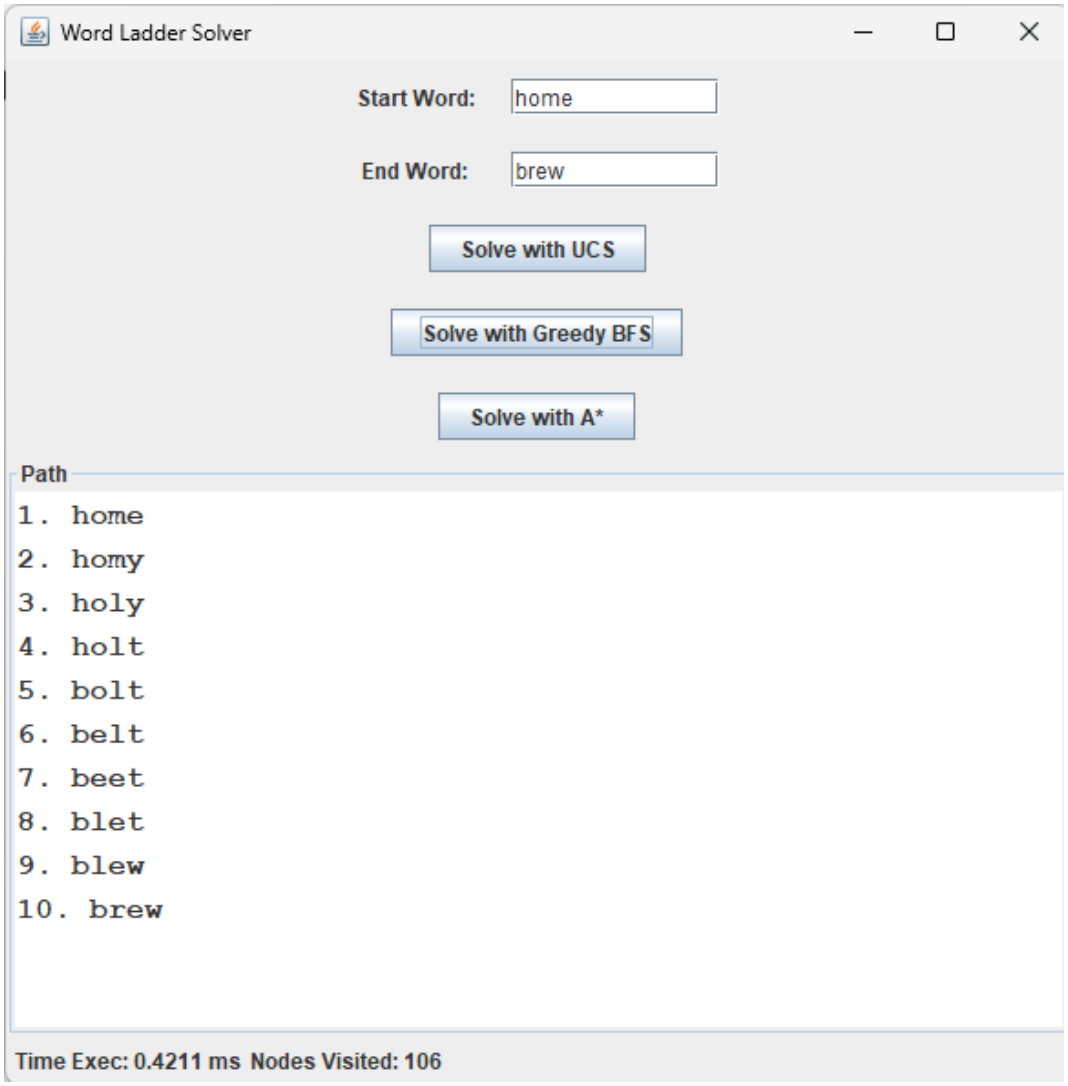


4. Test Case 4

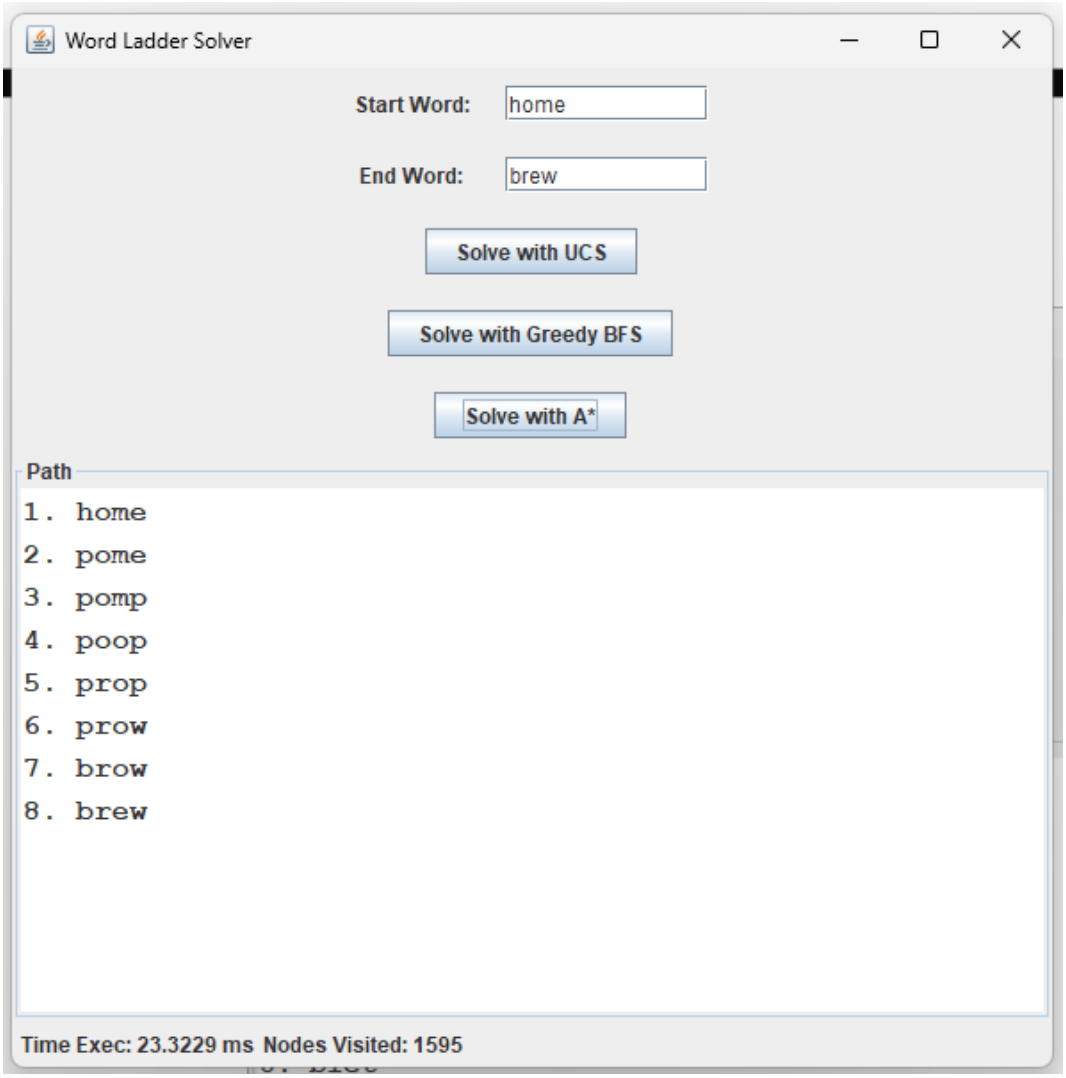
UCS


Word Ladder Solver
Start Word:
home
End Word:
brew
Solve with UCS
Solve with Greedy BFS
Solve with A*
Path
1. home
2. come
3. comp
4. coop
5. crop
6. crow
7. brow
8. brew
Time Exec: 114.0127 ms Nodes Visited: 3638

GBFS

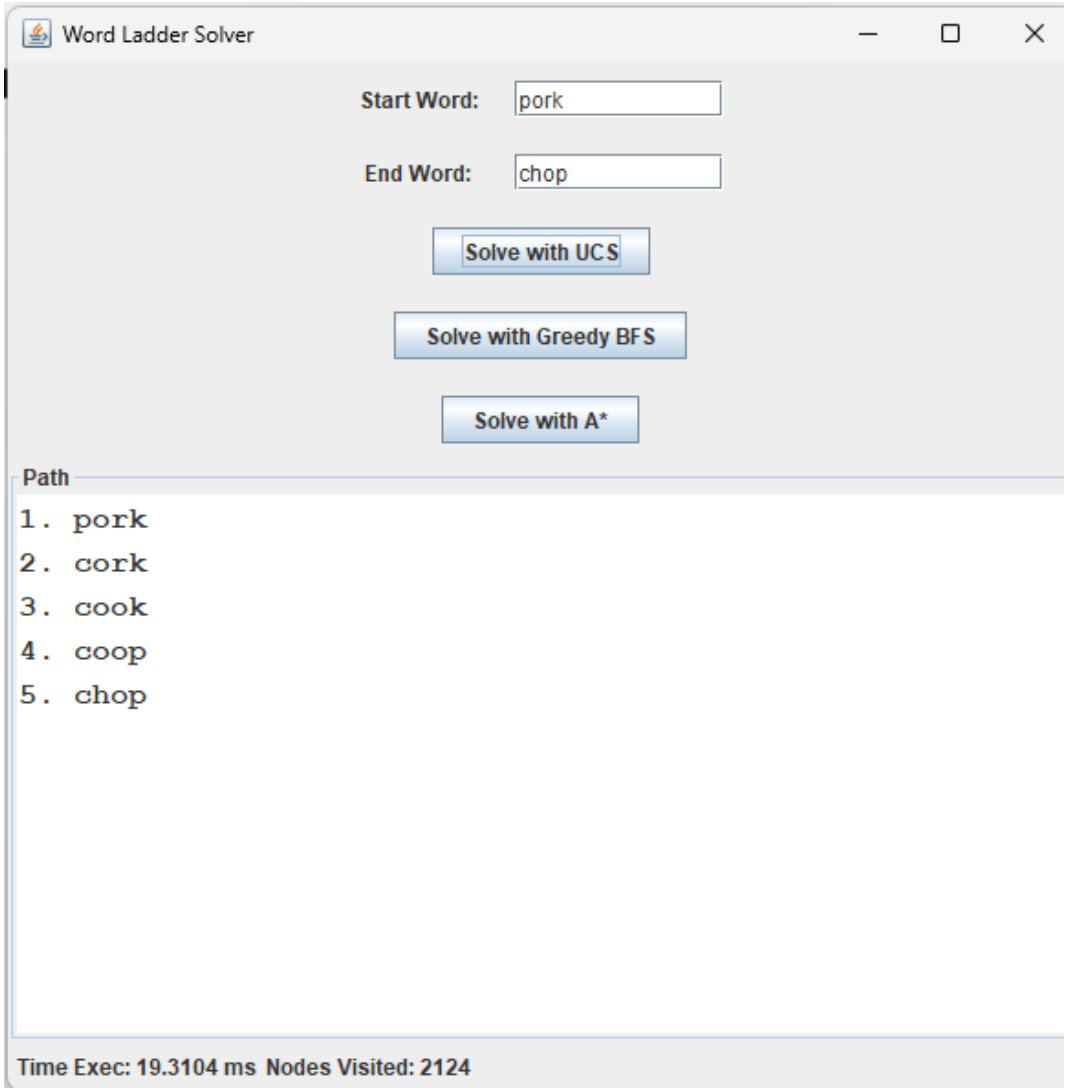

Word Ladder Solver
Start Word:
home
End Word:
brew
Solve with UCS
Solve with Greedy BFS
Solve with A*
Path
1. home
2. homy
3. holy
4. holt
5. bolt
6. belt
7. beet
8. blet
9. blew
10. brew
Time Exec: 0.4211 ms Nodes Visited: 106

A*

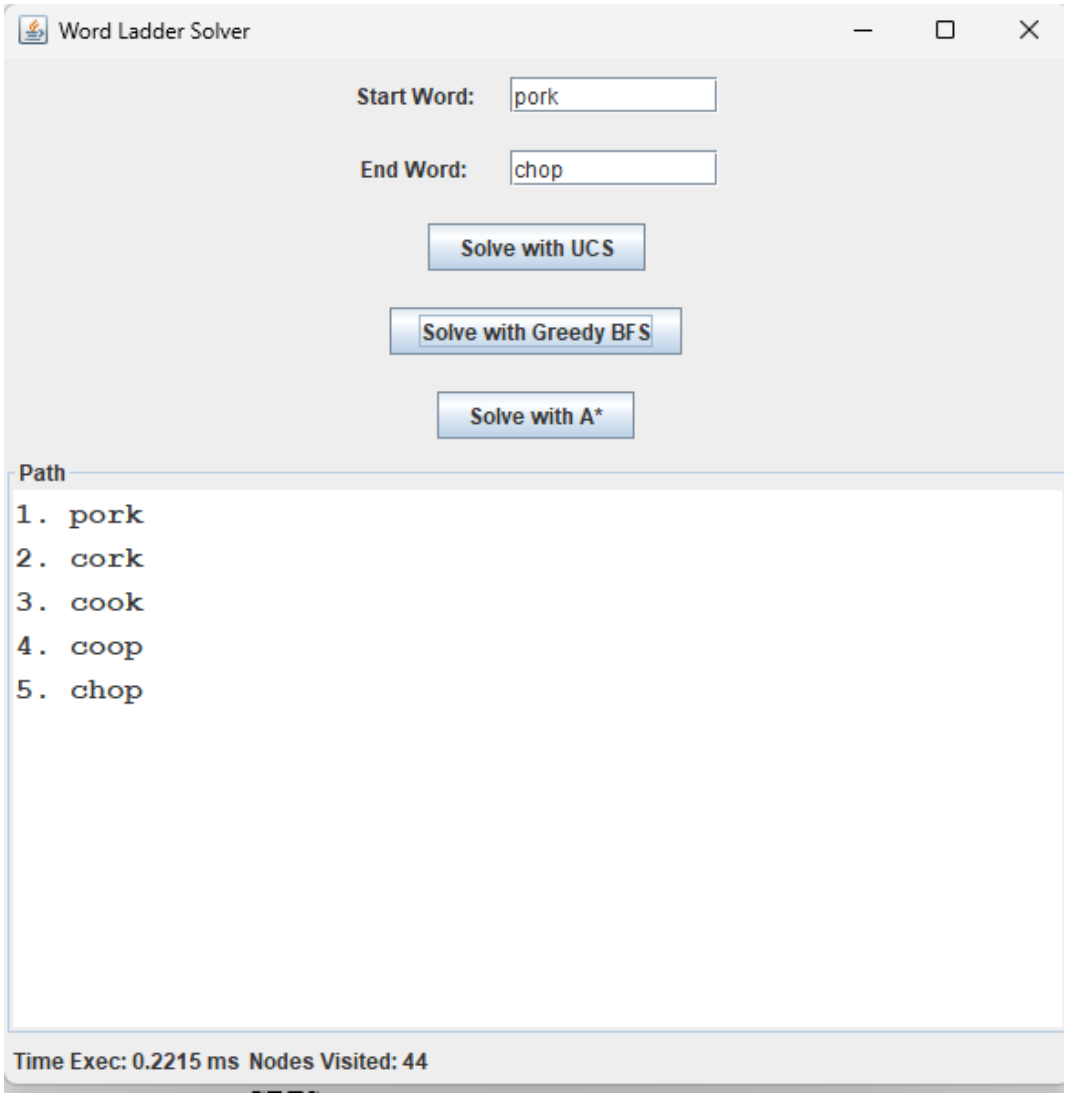


5. Test Case 5

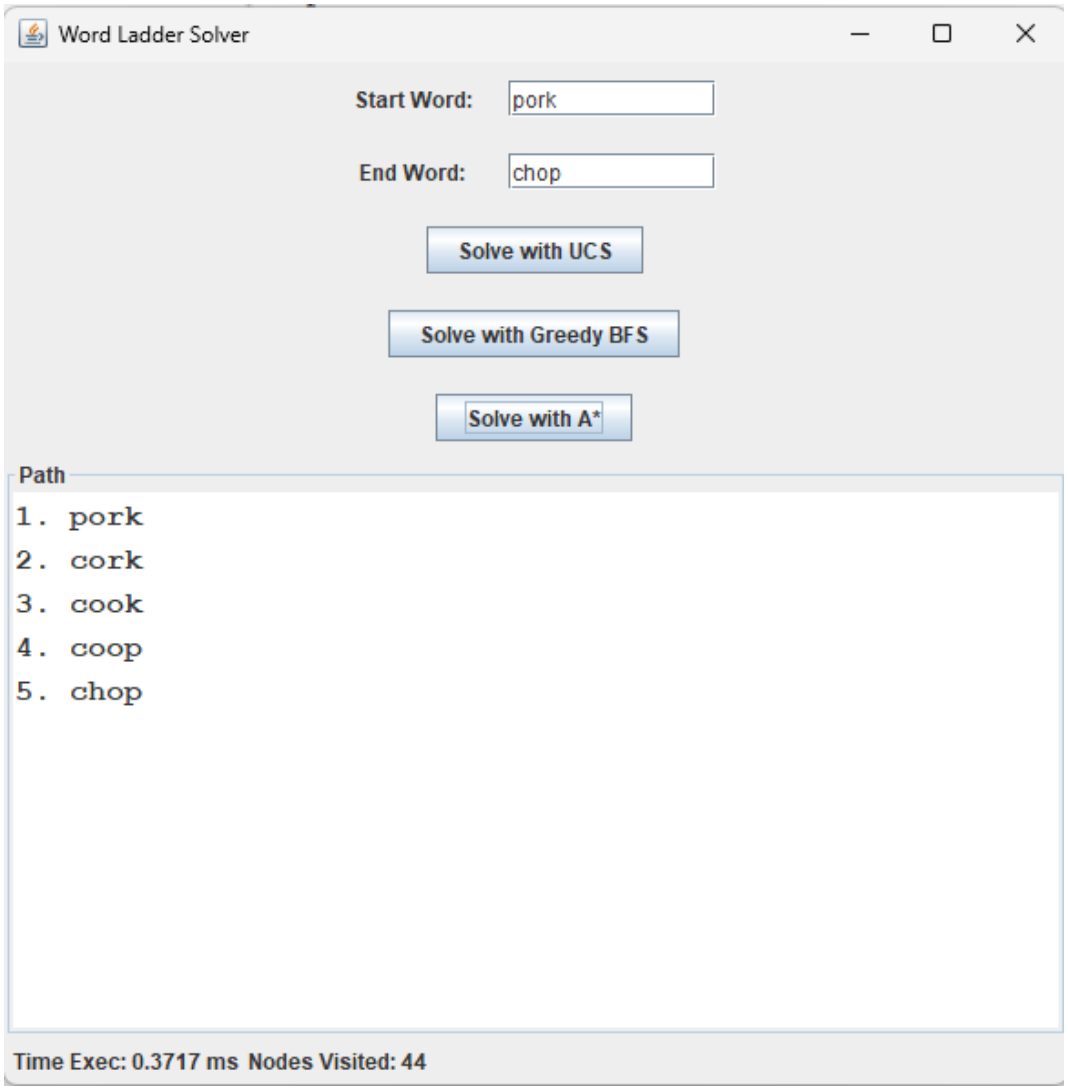
UCS


Word Ladder Solver
Start Word:
pork
End Word:
chop
Solve with UCS
Solve with Greedy BFS
Solve with A*
Path
1. pork
2. cork
3. cook
4. coop
5. chop
Time Exec: 19.3104 ms Nodes Visited: 2124

GBFS

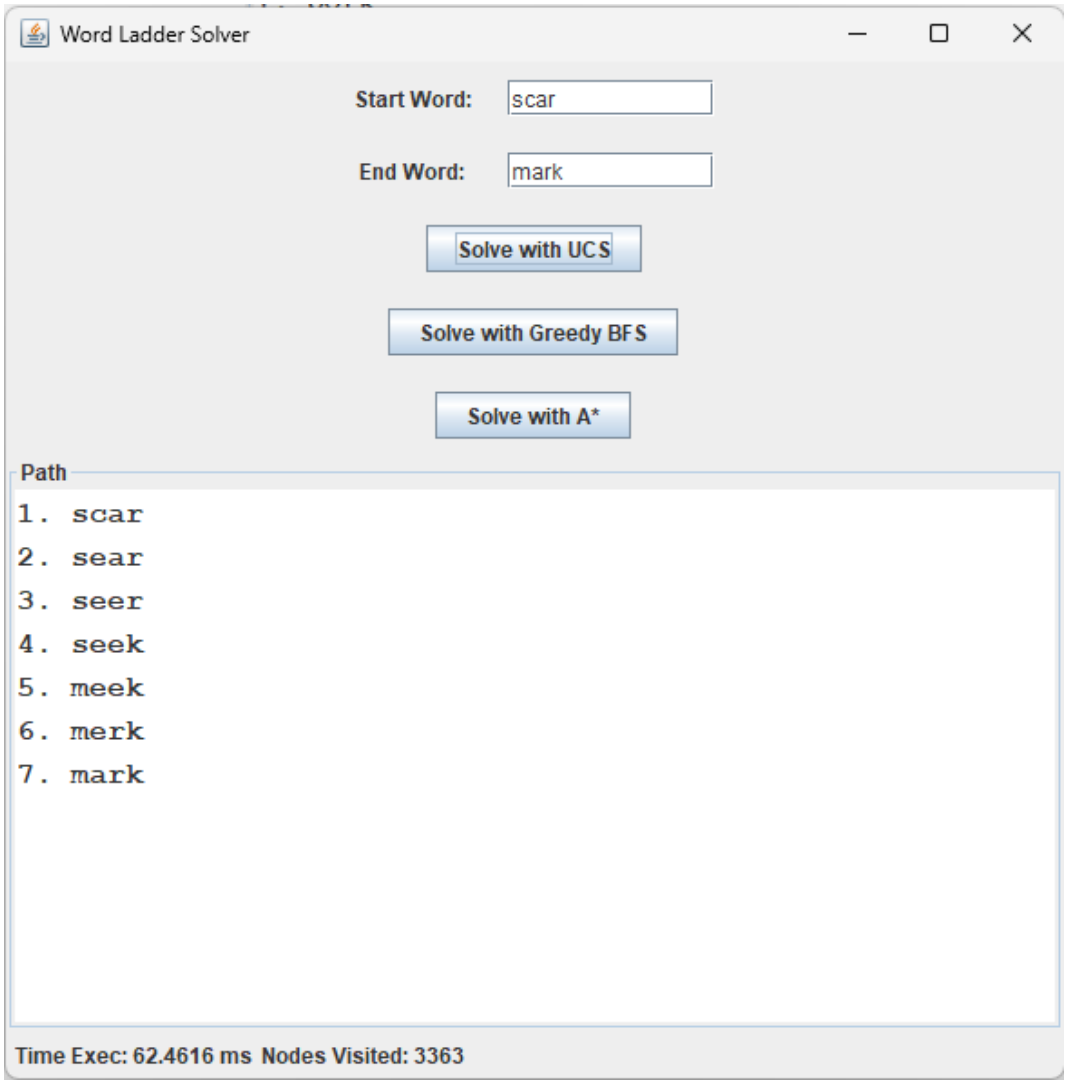

Word Ladder Solver
Start Word:
pork
End Word:
chop
Solve with UCS
Solve with Greedy BFS
Solve with A*
Path
1. pork
2. cork
3. cook
4. coop
5. chop
Time Exec: 0.2215 ms Nodes Visited: 44

A*

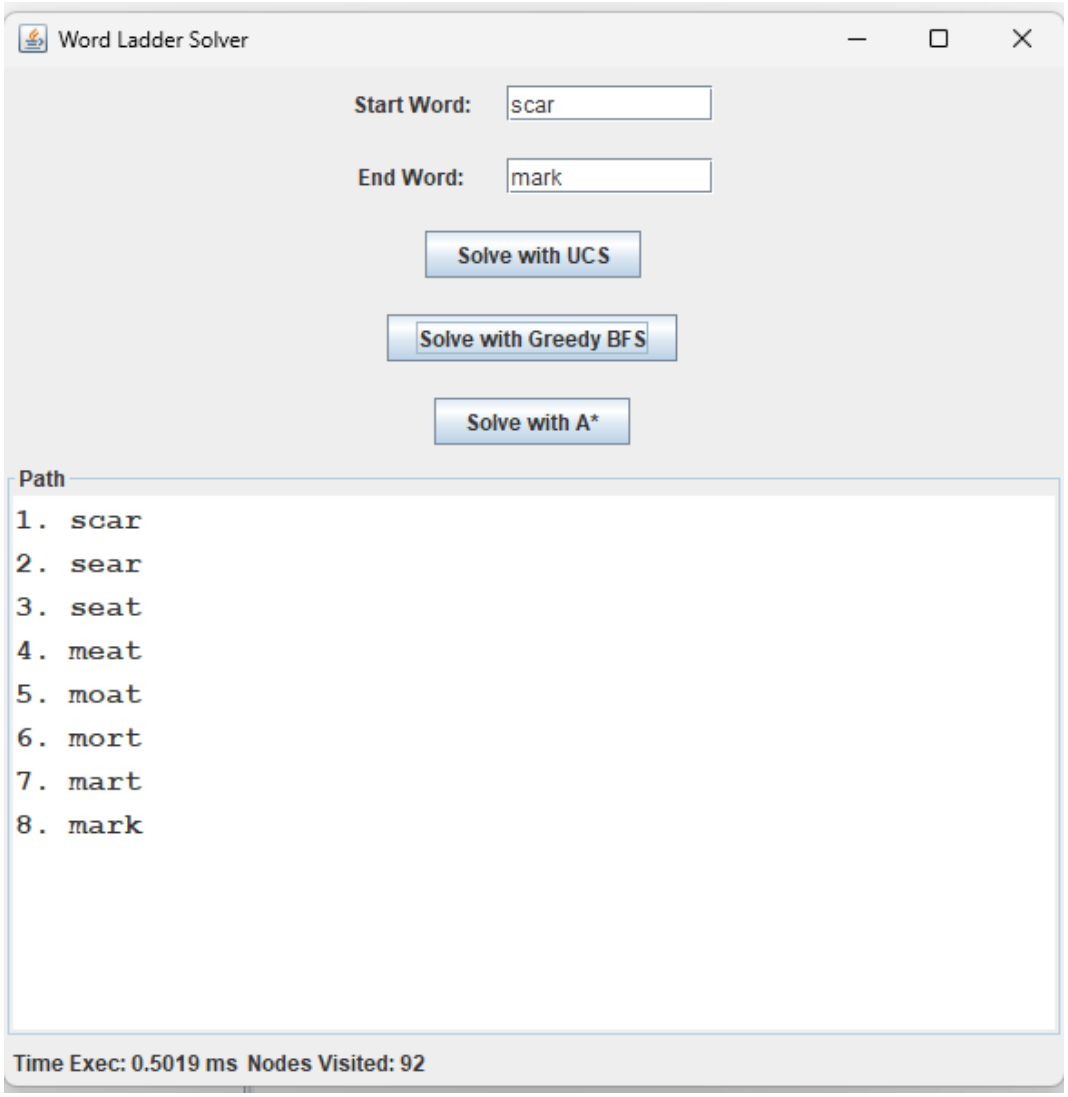


6. Test Case 6

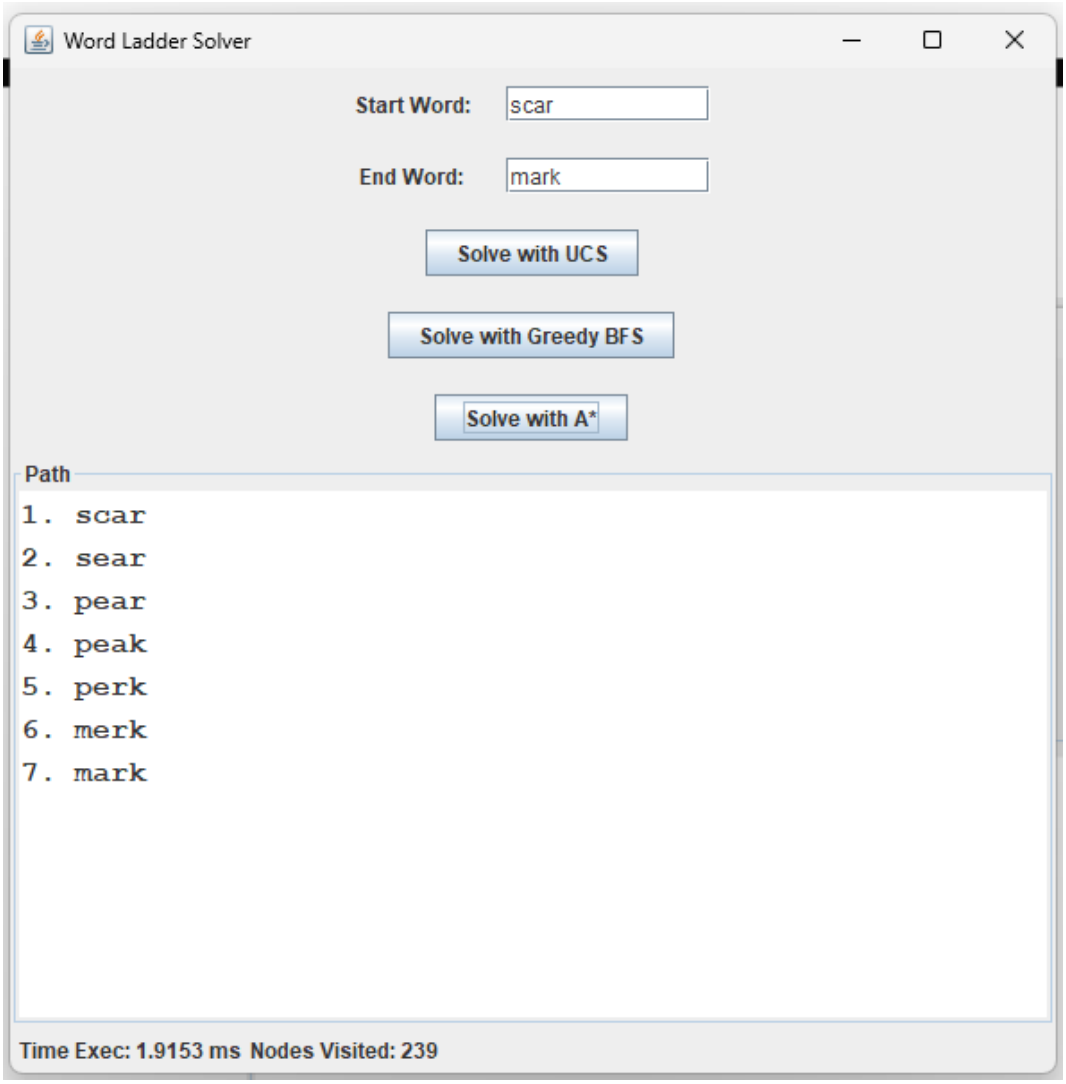
UCS


Word Ladder Solver
Start Word:
scar
End Word:
mark
Solve with UCS
Solve with Greedy BFS
Solve with A*
Path
1. scar
2. sear
3. seer
4. seek
5. meek
6. merk
7. mark
Time Exec: 62.4616 ms Nodes Visited: 3363

GBFS


Word Ladder Solver
Start Word:
scar
End Word:
mark
Solve with UCS
Solve with Greedy BFS
Solve with A*
Path
1. scar
2. sear
3. seat
4. meat
5. moat
6. mort
7. mart
8. mark
Time Exec: 0.5019 ms Nodes Visited: 92

A*



B. Pembahasan

Berdasarkan test case yang telah dilakukan di atas bisa diketahui bahwa algoritma UCS dan juga A* selalu memberikan solusi yang optimal sedangkan algoritma Greedy BFS tidak selalu memberikan solusi yang optimal. Di beberapa kondisi terkadang algoritma Greedy BFS bisa menemukan solusi dengan langkah yang lebih banyak dibandingkan solusi yang ditemukan oleh algoritma UCS dan algoritma A*. Meskipun algoritma Greedy BFS tidak optimal namun diantara ketiga algoritma yang dibuat algoritma Greedy BFS merupakan algoritma yang paling efisien dimana seperti yang bisa dilihat pada test case diatas algoritma Greedy BFS mempunyai rata-rata waktu eksekusi dan banyak node yang

dikunjungi yang paling rendah. Algoritma Greedy dengan fungsi heuristiknya membatasi jumlah node yang akan diproses dengan cukup ketat yang berakibat pada penggunaan memori dan waktu yang sangat efisien.

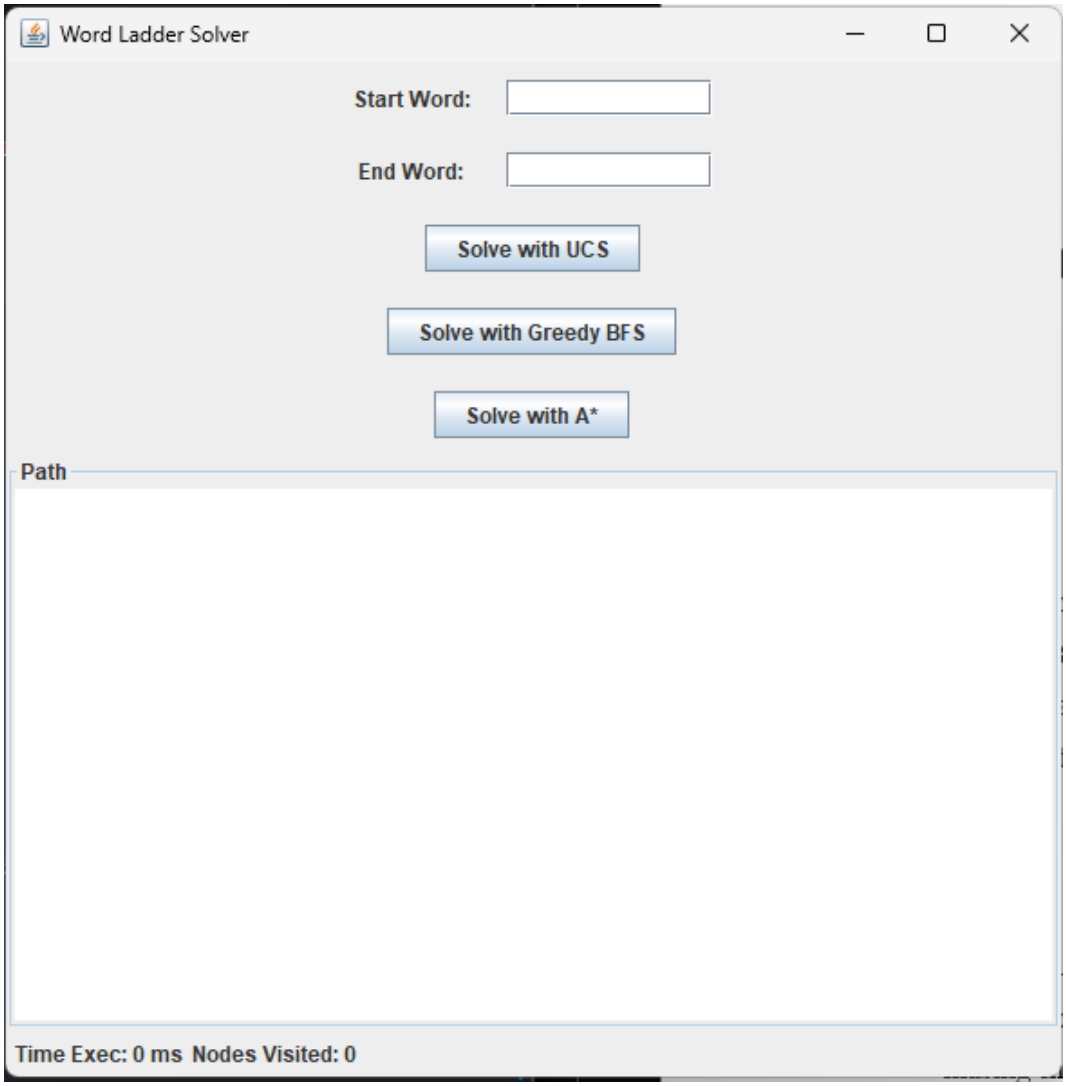
Algoritma A* juga sebenarnya cukup efisien walaupun tidak se-efisien algortma Greedy BFS. Rata-rata waktu eksekusi dam jumlah node yang dikunjungi pada algoeritma A* tidak terlalu jauh berbeda dengan algortima Greedy BFS. Sama seperti algoritma Greedy BFS algoritma A* juga menggunakan fungsi heuristik yang membatasi jumlah node yang perlu diproses sehingga dapat meningkatkan efisiensi namun karena algoritma A* masih mempertimbangkan nilai cost secara keseluruhan maka terdapat beberapa proses tambahan yang diperlukan yang membuatnya menjadi tidak se-efisien algoritma Greedy BFS.

Algoritma UCS adalah algoritma yang paling tidak efisien diantara ketiga algoritma yang dibuat. Pada kasus word ladder ini algoritma UCS tidak jauh berbeda jika dibandingkan dengan algoritma BFS biasa. Pada algoritma ini semua node yang ada dalam graf akan diproses semua yang membuat kebutuhannya akan memori menjadi sangat besar dan membuat waktu eksekusinya menjadi lebih lama dibandingkan algoritma yang lain. Jika dilihat pada test case diatas algoritma ini mempunyai rata-rata waktu eksekusi dan jumlah node dikunjungi yang paling besar. Selisih waktu eksekusi dan jumlah node yang dikunjungi antara algortima ini dengan dua algortima lainnya juga cukup besar dan akan semakin terlihat besar seiring berkembangnya ukuran lingkup pencarian.

Secara keseluruhan untuk kasus word ladder ini, algoritma A* adalah pilihan algoritma yang terbaik karena mampu menyeimbangkan antara optimalitas dan efisiensi dengan cukup baik. Greedy BFS mungkin cocok untuk kasus di mana waktu eksekusi adalah prioritas utama dan solusi suboptimal dapat diterima. UCS dapat diandalkan untuk menemukan solusi yang paling optimal tetapi mungkin tidak praktis dalam situasi dengan ruang pencarian yang sangat besar atau keterbatasan memori yang ketat.

BONUS

Bonus dalam tugas ini adalah pembuatan GUI untuk aplikasi word ladder solver. Bonus GUI ini dibuat dengan menggunakan bahasa Java dengan menggunakan package GUI bawaaan Java yaitu Java Swing.Implementasi kode GUI ini terletak pada folder src/GUI. Berikut adalah tampilan dari GUI yang dibuat.



Pada GUI terdapat dua buah text box yang digunakan untuk memasukkan kata asal dan juga kata tujuan. Kemudian terdapat juga tiga tombol yang mewakili masing-masing algoritma. Ketika tombol ditekan maka solusi dari word ladder akan dicari dengan algortima pencarian yang dipilih. Hasil dari proses pencarian akan ditampilkan pada box di bawah beserta dengan waktu eksekusi dan jumlah node yang dikunjungi.

LAMPIRAN

Link GitHub: https://github.com/Otzzu/Tucil3_13522117

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