Laporan Tugas Kecil 3 IF2211 Strategi Algoritma

Penyelesaian Permainan Word Ladder dengan Algoritma Uniform Cost Search, Greedy Best First Search, dan A* Search

Dibuat Oleh:

Farel Winalda / 13522047

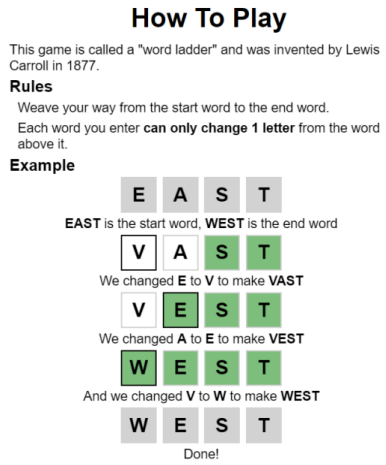
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG 2024

DAFTAR ISI

BAB I
DESKRIPSI MASALAH

Word ladder, juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragrams, atau word golf, adalah salah satu permainan kata yang sangat populer. Permainan ini pertama kali ditemukan oleh Lewis Carroll, seorang penulis terkenal dan matematikawan, pada tahun 1877. Dalam permainan ini, diberikan dua kata yang disebut sebagai *startword* dan *endword*. Tujuan pemain adalah mengubah start word menjadi end word dengan mengganti satu huruf pada setiap langkahnya, sehingga membentuk sebuah rantai kata yang valid. 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. *Permainan WordLadder*

(Sumber: https://wordwormdormdork.com/)

Pada laporan ini, akan dibahas beberapa algoritma yang digunakan untuk menyelesaikan permainan ini. Adapun algoritma yang digunakan, antara lain: algoritma *Uniform Cost Search (UCS)*, algoritma *Greedy Best First Search*, dan algoritma *A** search. Pada laporan ini, akan dibahas juga mengenai implementasi program menggunakan bahasa pemrograman *Java* dan tampilan *GUI* dengan menggunakan Library *JavaFX*.

BAB II
IMPLEMENTASI PROGRAM

Berikut merupakan dari implementasi source code program, beserta penjelasan tiap class dan method yang diimplementasi.

2.1. Main.java

Main.java

```
public class Main extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) throws Exception {
        try {
                                                        FXMLLoader     loader     =     new
FXMLLoader(getClass().getResource("resources/MainScene.fxml"));
            Parent root = loader.load();
            Scene scene = new Scene(root);
            // System.out.println(getClass().getResource("resources/style.css"));
                                                                  String      css      =
this.getClass().getResource("resources/style.css").toExternalForm();
            scene.getStylesheets().add(css);
            primaryStage.setTitle("Word Ladder GUI");
            primaryStage.setScene(scene);
            primaryStage.show();
            primaryStage.setResizable(false);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Di dalam file Main.java, terdapat class Main yang memiliki method main untuk menginisialisasi file *fxml* yang digunakan untuk memuat *GUI JavaFX*, memuat file *CSS (Cascading Style Sheets)* untuk desain *GUI*, dan juga menginisiasi metadata dan konfigurasi dari *GUI* yang digunakan.

2.2. MainController.java

MainController.java

```
package resources;
public class MainController {

    @FXML private HBox startWordBox, endWordBox;
    @FXML private VBox userInputs;
```

```
    @FXML private TextFlow resultTextFlow;
    @FXML private VBox ucsResultsBox, greedyResultsBox, aStarResultsBox;
    @FXML private CheckBox ucsCheckBox;
    @FXML private CheckBox gbfsCheckBox;
    @FXML private CheckBox aStarCheckBox;

    private Generate2Words generator = new Generate2Words();
    private Set<String> dictionary = new HashSet<>();
    private String currentWord;

    private boolean showUCSResults = false;
    private boolean showGBFSResults = false;
    private boolean showAStarResults = false;

    private boolean ucsResultCalculated = false;
    private boolean gbfsResultCalculated = false;
    private boolean aStarResultCalculated = false;

    @FXML
    public void initialize() {
        loadDictionary();
        shuffleWords(null);
    }

    private void loadDictionary() {
        String filePath = "src/words.txt";
        try {
            List<String> lines = Files.readAllLines(Paths.get(filePath));
            generator.organizeWords(lines);
            dictionary.addAll(lines);
        } catch (IOException e) {
              resultTextFlow.getChildren().add(new Text("Failed to read words from file: " +
e.getMessage()));
        }
    }

    private void generateRandomWords() {}

    private void populateWordBoxes(HBox wordBox, String word) {}

    private TextField createSingleCharTextField() {}

    private void resetInputRows(int length) {}

    @FXML
    void shuffleWords(ActionEvent event) {
        generateRandomWords();
        clearResults();
        resultTextFlow.getChildren().clear();

        ucsCheckBox.setSelected(false);
        gbfsCheckBox.setSelected(false);
        aStarCheckBox.setSelected(false);

        ucsResultCalculated = false;
        gbfsResultCalculated = false;
        aStarResultCalculated = false;
    }

    @FXML
    void checkGuess(ActionEvent event) {}

    @FXML
    private void handleAlgorithmCheckbox(ActionEvent event) {}

    private void updateResultsVisibility() {}

    @FXML
    void openCustomWordDialog(ActionEvent event) {}
```

```
    private void processWordGuess(String guessedWord) {}

    private String getEndWord() {}

    private void resetGame() {
      Alert alert = new Alert(Alert.AlertType.CONFIRMATION, "Play again?", ButtonType.YES,
ButtonType.NO);
    alert.showAndWait().ifPresent(response -> {
        if (response == ButtonType.YES) {
            shuffleWords(null);
        }
    });
}

    private boolean isOneLetterChanged(String oldWord, String newWord) {}

    private void addInputRow(int length) {}

    private boolean isValidWord(String word) {
        return dictionary.contains(word);
    }

    @FXML
    void provideHint(ActionEvent event) {
                  if ((showUCSResults && !ucsResultCalculated) || (showGBFSResults &&
!gbfsResultCalculated) || (showAStarResults && !aStarResultCalculated)) {
            if (currentWord != null && !getEndWord().isEmpty()) {
                if (showUCSResults && !ucsResultCalculated) {
                                                    SearchResult ucsResult = new
UniformCostSearch("src/words.txt").search(currentWord, getEndWord());
                    displayResult(ucsResultsBox, ucsResult);
                    ucsResultCalculated = true;
                }

                if (showGBFSResults && !gbfsResultCalculated) {
                                                        SearchResult greedyBFSResult = new
GreedyBestFirstSearch("src/words.txt").search(currentWord, getEndWord());
                    displayResult(greedyResultsBox, greedyBFSResult);
                    gbfsResultCalculated = true;
                }

                if (showAStarResults && !aStarResultCalculated) {
                                                       SearchResult aStarResult = new
AStarSearch("src/words.txt").search(currentWord, getEndWord());
                    displayResult(aStarResultsBox, aStarResult);
                    aStarResultCalculated = true;
                }

                ucsResultsBox.setVisible(showUCSResults);
                greedyResultsBox.setVisible(showGBFSResults);
                aStarResultsBox.setVisible(showAStarResults);
            } else {
                resultTextFlow.getChildren().clear();
                     resultTextFlow.getChildren().add(new Text("Please shuffle words before
requesting hints."));
            }
        }
    }

    private void displayResult(VBox resultsBox, SearchResult searchResult) {
        resultsBox.getChildren().clear();

        // Use the mapping function to get the formatted title based on the VBox ID
        String titleLabelText = getFormattedTitle(resultsBox.getId());
        Label titleLabel = new Label(titleLabelText);
        titleLabel.getStyleClass().add("label-title");
        resultsBox.getChildren().add(titleLabel);

        if (searchResult.getFirstFoundPath().isEmpty()) {
            Label notFoundLabel = new Label("No results found.");
```

```
            notFoundLabel.getStyleClass().add("result-label");
            resultsBox.getChildren().add(notFoundLabel);
        } else {
                            Label timeLabel = new Label(String.format("Time: %.2f ms",
searchResult.getElapsedTime() / 1_000_000.0));
            timeLabel.getStyleClass().add("time-label");
            resultsBox.getChildren().add(timeLabel);


                                Label pathsFoundLabel = new Label("Words Checked: " +
searchResult.getWordCheckCount());
            pathsFoundLabel.getStyleClass().add("result-label");
            resultsBox.getChildren().add(pathsFoundLabel);

            Label resLabel = new Label("Result Path:");
            resLabel.getStyleClass().add("result-label");
            resultsBox.getChildren().add(resLabel);
            searchResult.getFirstFoundPath().forEach(word -> {
                Label resultLabel = new Label(word);
                resultLabel.getStyleClass().add("result-label");
                resultsBox.getChildren().add(resultLabel);
            });

        }
    }

    private String getFormattedTitle(String vboxId) {}

    private void clearResults() {}
}
```

Di Dalam file MainController.java terdapat class MainController yang berisikan beberapa atribut / fungsi untuk mengatur jalannya *GUI* (mulai dari generate start word and end word hingga menyediakan input bagi user untuk bermain hingga mencapai endword), memuat dictionary yang digunakan yang berisi beberapa kosakata valid dalam Bahasa Inggris, dan menginisialisasi hasil dari algoritma UCS, Greedy BFS, dan A*.

2.3. WordLadder.java

WordLadder.java

```
package utils;
public abstract class WordLadder {
    protected Set<String> dictionary;
    private Set<Character> availableCharacters;

    protected abstract SearchResult search(String startWord, String endWord);

    public WordLadder(String filePath) {
        this.dictionary = new HashSet<>();
        this.availableCharacters = new HashSet<>();
        loadWords(filePath);
    }

    private void loadWords(String filePath) {
        try {
            List<String> lines = Files.readAllLines(Paths.get(filePath));
```

```
            for (String line : lines) {
                if (!line.trim().isEmpty()) {
                    dictionary.add(line.trim());
                    for (char c : line.trim().toCharArray()) {
                        availableCharacters.add(c);
                    }
                }
            }
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
        }
    }

    public boolean isLetterAvailable(char letter) {
        return availableCharacters.contains(letter);
    }

    protected List<String> getNextWords(String word) {
        List<String> nextWords = new ArrayList<>();
        char[] chars = word.toCharArray();
        for (int i = 0; i < word.length(); i++) {
            char originalChar = chars[i];
            for (char c = 'a'; c <= 'z'; c++) {
                if (c != originalChar) {
                    chars[i] = c;
                    String nextWord = new String(chars);
                    if (dictionary.contains(nextWord)) {
                        nextWords.add(nextWord);
                    }
                }
            }
            chars[i] = originalChar; // Restore original character
        }
        return nextWords;
    }

    protected List<String> reconstructPath(Node endNode) {
        LinkedList<String> path = new LinkedList<>();
        Node current = endNode;
        while (current != null) {
            path.addFirst(current.word);
            current = current.parent;
        }
        return path;
    }

    protected 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;
        }
    }
}
```

Di Dalam file WordLadder..java terdapat class WordLadder yang memiliki atribut berupa HashSet String dictionary yang berisi semua kata yang ada dalam kamus, lalu Hashset availableCharacters untuk menyimpan karakter yang tersedia, lalu terdapat fungsi untuk memuat

file kamus txt, builder untuk membangun node dari kata awal menuju kata akhir dan struktur data Node yang berisi data berupa *cost*, kata, dan parent dari kata tersebut.

2.4. Generate2Words.java

Generate2Words.java

```
package utils;
public class Generate2Words {
    private Map<Integer, List<String>> wordsByLength = new HashMap<>();
    private Random random = new Random();

    public void organizeWords(List<String> lines) {
        for (String word : lines) {
            int length = word.length();
            if (length >= 2 && length <= 20) {
                wordsByLength.putIfAbsent(length, new ArrayList<>());
                wordsByLength.get(length).add(word);
            }
        }
    }

    public String[] getRandomWords() {
        return getRandomWordsFromMap(null);
    }

    public String[] getRandomWords(int length) {
        return getRandomWordsFromMap(length);
    }

    private String[] getRandomWordsFromMap(Integer specifiedLength) {
        List<Integer> validLengths = new ArrayList<>(wordsByLength.keySet());

         validLengths.removeIf(length -> wordsByLength.get(length).size() < 2 || length < 3
|| length > 15);

        if (specifiedLength != null) {
                    if  (!validLengths.contains(specifiedLength)  ||  specifiedLength  <  3  ||
specifiedLength > 15) {
                return null;
            }
            validLengths.retainAll(Collections.singleton(specifiedLength));
        }

        if (validLengths.isEmpty()) {
            return null;
        }

        int selectedLength = validLengths.get(random.nextInt(validLengths.size()));
        List<String> wordsOfSelectedLength = wordsByLength.get(selectedLength);

        Collections.shuffle(wordsOfSelectedLength);
        String startWord = wordsOfSelectedLength.get(0);
        String endWord = wordsOfSelectedLength.get(1);

        return new String[]{startWord, endWord};
    }
}
```

Di dalam file Generate2Words.java terdapat Class Generate2Words yang berisikan atribut berupa data dari kamus yang telah dikelompokkan berdasarkan panjang dan randomizer untuk

me-random kata yang digunakan dalam permainan, dan terdapat dfungsi yang mengatur kata yang keluar yaitu harus lebih dari 2 kata dan kurang dari 15 kata.

2.5. UniformCostSearch.java

UniformCostSearch.java

```
package utils;
public class UniformCostSearch extends WordLadder {
    public UniformCostSearch(String filePath) {
        super(filePath);
    }

    @Override
    public SearchResult search(String startWord, String endWord) {
        long startTime = System.nanoTime();
            PriorityQueue<Node> frontier = new PriorityQueue<>(Comparator.comparingInt(n ->
n.cost));
        Map<String, Integer> costSoFar = new HashMap<>();
        frontier.add(new Node(startWord, null, 0));
        costSoFar.put(startWord, 0);

        List<String> firstFoundPath = new ArrayList<>();
        boolean found = false;
        int wordCheckCount = 0;

        while (!frontier.isEmpty()) {
            Node current = frontier.poll();
            wordCheckCount++;

            if (current.word.equals(endWord)) {
                if (!found) {
                    firstFoundPath = reconstructPath(current);
                    found = true;
                    break;
                }
            }

            for (String next : getNextWords(current.word)) {
                int newCost = current.cost + 1;
                if (!costSoFar.containsKey(next) || newCost < costSoFar.get(next)) {
                    costSoFar.put(next, newCost);
                    frontier.add(new Node(next, current, newCost));
                }
            }
        }
        long elapsedTime = System.nanoTime() - startTime;
        return new SearchResult(firstFoundPath, wordCheckCount, elapsedTime);
    }
}
```

Di dalam file UniformCostSearch.java terdapat Class UniformCostSearch yang merupakan Inheritance dari Class WordLadder dan memiliki Specialization yaitu fungsi search yang mengoverride fungsi search dari WordLadder dan berisikan implementasi dari Algoritma Uniform Search Cost, mulai dari inisiasi waktu, priorityqueue yang membandingkan *cost* / biaya, HashMap dari biaya sekarang. Selama priorityqueue tidak kosong, node dengan biaya terendah diambil, jika kata awal sama dengan kata akhir, akan direkonstruksi node yang dibentuk dan

pencarian akan berhenti. Lalu, jika kata belum ditemukan, iterasi dilakukan pada setiap kata yang mungkin dengan fungsi getNextWords. Untuk setiap kata selanjutnya biaya akan ditambah 1. Jika kata selanjutnya belum tercatat pada costSoFar, atau biaya baru lebih kecil dari biaya yang telah tercatat, maka kata tersebut akan ditambahkan ke priorityqueue dan costSoFat di update. Fungsi ini akan mengembalikan Class SearchResult yang baru yang diperlukan untuk keperluan GUI

2.6. GreedyBestFirstSearch.java

GreedyBestFirstSearch.java

```
package utils;

public class GreedyBestFirstSearch extends WordLadder {
    public GreedyBestFirstSearch(String filePath) {
        super(filePath);
    }

    @Override
    public SearchResult search(String startWord, String endWord) {
        long startTime = System.nanoTime();
            PriorityQueue<Node> frontier = new PriorityQueue<>(Comparator.comparingInt(n ->
getHeuristic(n.word, endWord)));
        Set<String> visited = new HashSet<>();
        frontier.add(new Node(startWord, null, 0));

        List<String> firstFoundPath = new ArrayList<>();
        boolean found = false;
        int wordCheckCount = 0;

        while (!frontier.isEmpty() && !found) {  // Stop loop once goal is found
            Node current = frontier.poll();
            wordCheckCount++;

            if (current.word.equals(endWord)) {
                firstFoundPath = reconstructPath(current);
                found = true;
                break;  // Exit the loop after finding the goal
            }

            if (!visited.contains(current.word)) {
                visited.add(current.word);
                for (String next : getNextWords(current.word)) {
                    if (!visited.contains(next)) {
                        frontier.add(new Node(next, current, 0));
                    }
                }
            }
        }
        long elapsedTime = System.nanoTime() - startTime;
        return new SearchResult(firstFoundPath, wordCheckCount, elapsedTime);
    }

    private int getHeuristic(String word, String endWord) {
        int mismatch = 0;
        for (int i = 0; i < word.length(); i++) {
            if (word.charAt(i) != endWord.charAt(i)) {
                mismatch++;
            }
        }
        return mismatch;
    }
}
```

Di dalam file GreedyBestFirstSearch.java terdapat Class GreedyBestFirstSearch yang merupakan inheritance dari WordLadder dan secara khusus mengimplementasi algoritma GreedyBestFirstSearch. Pada metode search diawali dengan inisiasi waktu, dan priorityqueue yang diurutkan berdasarkan nilai heuristik yang digunakan untuk menghitung perbedaan karakter antara kata sekarang dengan kata akhir. Digunakan HashSet untuk menyimpan kata yang pernah dikunjungi. Proses pencarian berlanjut selama priorityqueue tidak kosong dan kata akhir belum ditemukan. Node dengan heuristik terendah diambill oleh priorityqueue dan jika ditemukan kata akhir, maka akan direkonstruksi node dan pencarian berhenti dan jika belum kata tersebut akan

ditambahkan ke priorityqueue. Lalu akan dikembalikan Class SearchResult yang baru yang diperlukan untuk GUI.

2.7. AStarSearch.java

AStarSearch.java

```
package utils;
public class AStarSearch extends WordLadder {
    public AStarSearch(String filePath) {
        super(filePath);
    }

    @Override
    public SearchResult search(String startWord, String endWord) {
        long startTime = System.nanoTime();
            PriorityQueue<Node> frontier = new PriorityQueue<>(Comparator.comparingInt(n ->
n.cost + getHeuristic(n.word, endWord)));
        Map<String, Integer> costSoFar = new HashMap<>();
        frontier.add(new Node(startWord, null, 0));
        costSoFar.put(startWord, 0);

        List<String> firstFoundPath = new ArrayList<>();
        boolean found = false;
        int wordCheckCount = 0;

        while (!frontier.isEmpty() && !found) {
            Node current = frontier.poll();
            wordCheckCount++;

            if (current.word.equals(endWord)) {
                firstFoundPath = reconstructPath(current);
                found = true;
                break;  // Exit the loop after finding the goal
            }

            for (String next : getNextWords(current.word)) {
                int newCost = current.cost + 1;
                if (!costSoFar.containsKey(next) || newCost < costSoFar.get(next)) {
                    costSoFar.put(next, newCost);
                    frontier.add(new Node(next, current, newCost));
                }
            }
        }
        long elapsedTime = System.nanoTime() - startTime;
        return new SearchResult(firstFoundPath, wordCheckCount, elapsedTime);
    }

    private int getHeuristic(String word, String endWord) {
        int mismatch = 0;
        for (int i = 0; i < word.length(); i++) {
            if (word.charAt(i) != endWord.charAt(i)) {
                mismatch++;
            }
        }
        return mismatch;
    }
}
```

Di dalam file AStarSearch.java terdapat kelas AStarSearch yang merupakan inheritance dari Class WordLadder dan secara khusus mengimplementasi algoritma A*. Algoritma ini sendiri merupakan gabungan dari Greedy Best First Search dan UCS yang mengoptimalkan

pencarian berdasarkan biaya terendah dan dengan menggunakan heuristik untuk menemukan jalur terpendek. Dalam fungsi search diawali dengan inisiasi waktu, dan pembentukan priorityqueue dan juga menggunakan costSoFar untuk mencatat biaya untuk mencapai setiapk kata dari kata awal. Selama priorityqueue tidak kosong, dan kata akhir belum ditemukan, loop akan terus berjalan. Node dengan biaya terendah diambil dari priorityqueue, jika node merupakan kata akhir akan direkonstruksi dan pencarian akan berhenti. Jika bukan, akan dijalankan fungsi getNextwords untuk menemukan kata berikutnya yang berbeda satu huruf dan untuk setiap kata berikutnya akan dihitung biaya baru. Jika kata tersebut belum ada di costSoFar dan memiliki biaya lebih rendah maka kata akan ditambahkan ke priorityqueue. Akan dikembalikan Class SearchResult yang berisikan data-data yang diperlukan untuk GUI

2.8. SearchResult.java

SearchResult.java

```
package utils;
public class SearchResult {
    private List<String> firstFoundPath;
    private int wordCheckCount;
    private long elapsedTime;

    public SearchResult(List<String> firstFoundPath, int wordCheckCount, long elapsedTime) {
        this.firstFoundPath = firstFoundPath;
        this.wordCheckCount = wordCheckCount;
        this.elapsedTime = elapsedTime;
    }

    public List<String> getFirstFoundPath() {
        return firstFoundPath;
    }

    public int getWordCheckCount() {
        return wordCheckCount;
    }

    public long getElapsedTime() {
        return elapsedTime;
    }
}
```

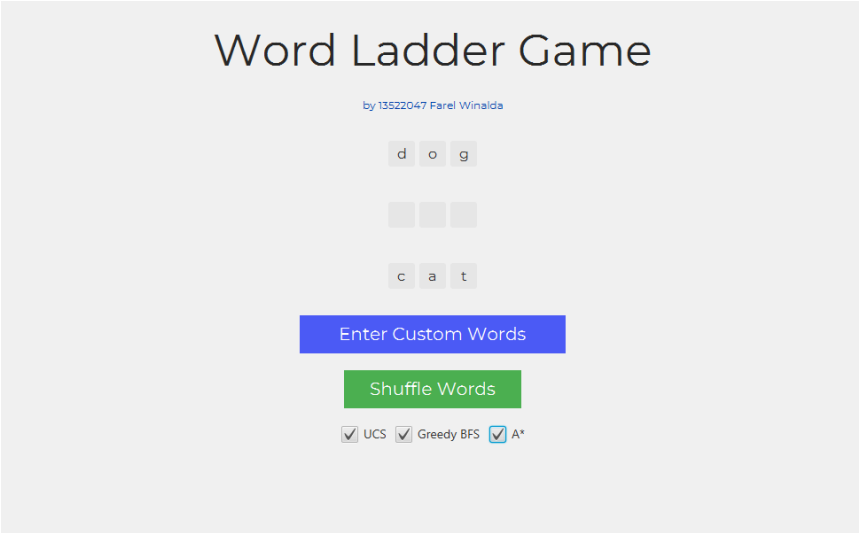
Di dalam file SearchResult.java terdapat kelas SearchResult yang berisikan data-data dari hasil pencarian menggunakan metode tertentu. Kelas ini meiliki atribut berupa jalur yang ditemukan pertama kali, jumlah kata yang telah dikunjungi dan lama eksekusi waktu yang diperlukan untuk ditampilkan oleh GUI.

BAB III

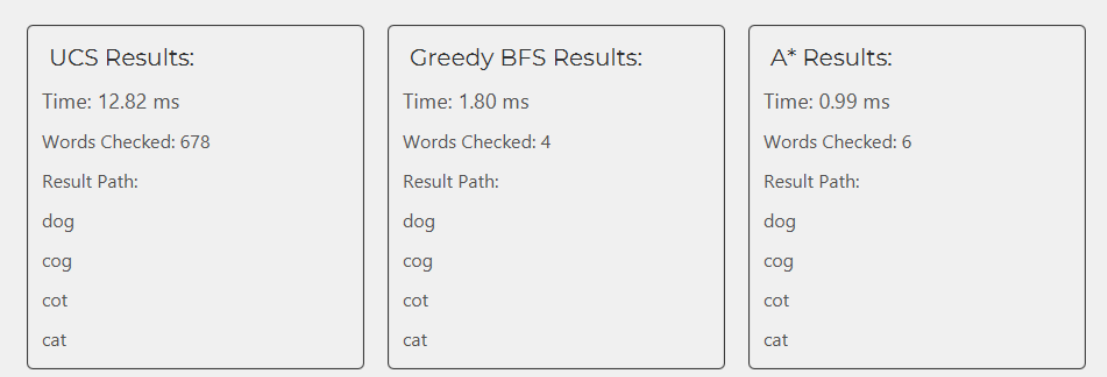
ANALISIS DAN EKSPERIMEN

3.1. Eksperimen

3.1.1. Pengujian 3 Kata dari *dog* ke *cat*

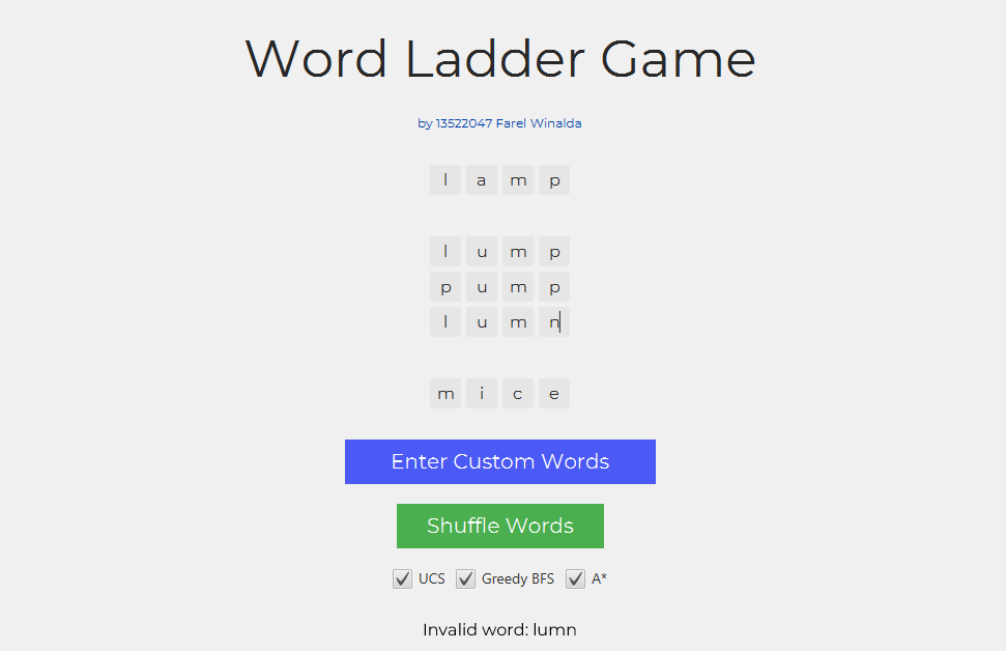


Gambar 3.1.1.1 *Tampilan GUI dari dog ke cat*

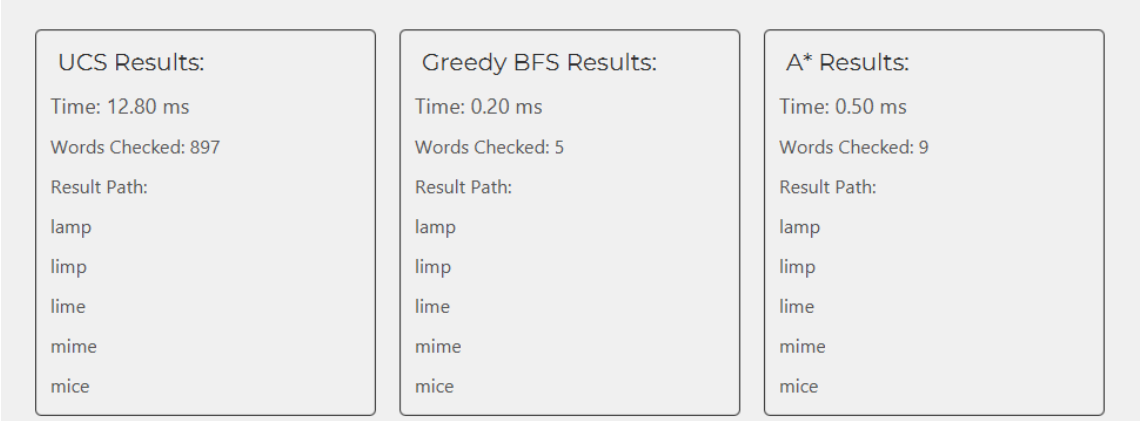


Gambar 3.1.1.2 *Tampilan hasil algoritma dari dog ke cat*

3.1.2. Pengujian 3 Kata dari *lamp* ke *mice*

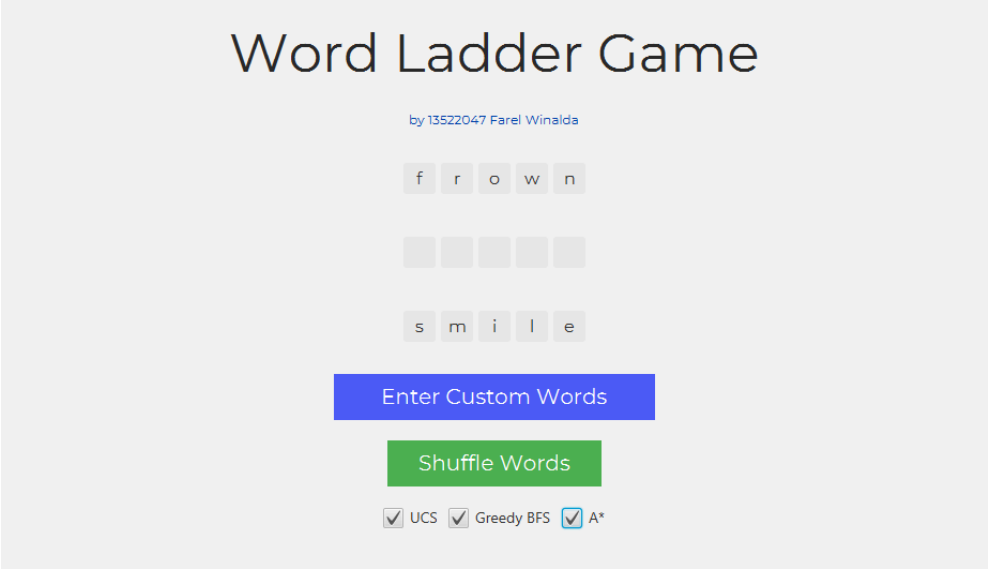


Gambar 3.1.2.1 *Tampilan GUI dari lamp ke mice*



Gambar 3.1.2.2 *Tampilan hasil algoritma dari lamp ke mice*

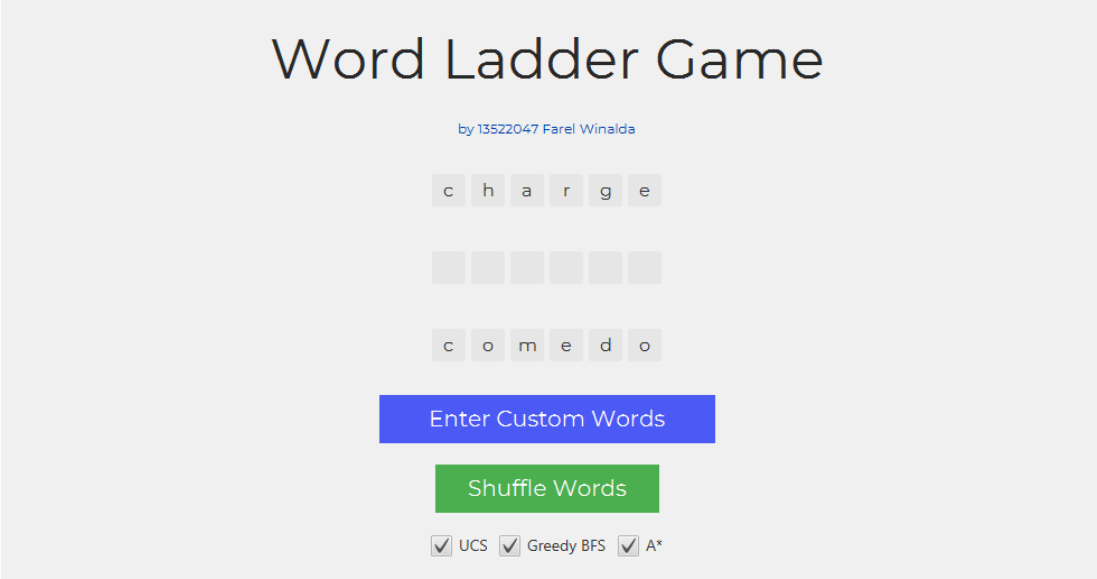
3.1.3. Pengujian 5 Kata dari *frown* ke *smile*



Gambar 3.1.3.1 *Tampilan GUI dari frown ke smile*

Gambar 3.1.3.2 *Tampilan hasil algoritma dari frown ke smile*

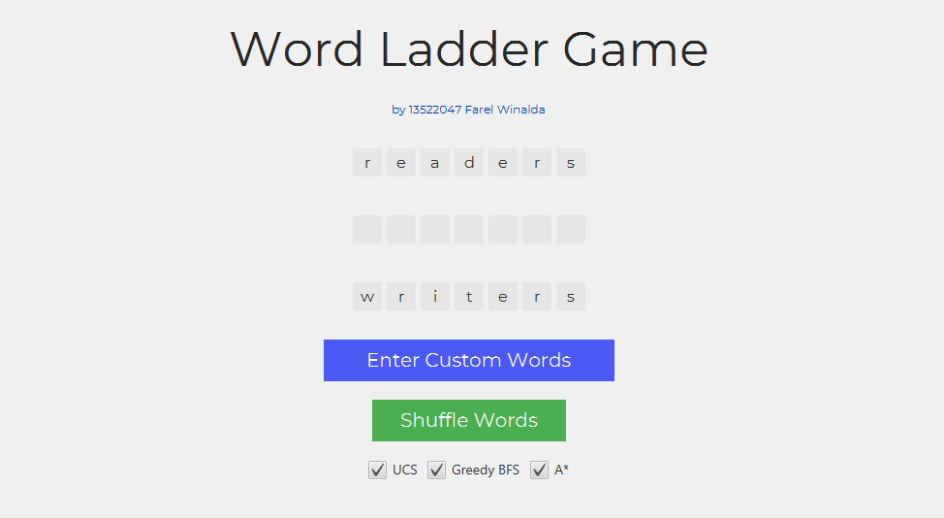
3.1.4. Pengujian 6 Kata dari *charge* ke *comedo*



Gambar 3.1.4.1 *Tampilan GUI dari charge ke comedo*

Gambar 3.1.4.2 *Tampilan hasil algoritma dari charge ke comedo*

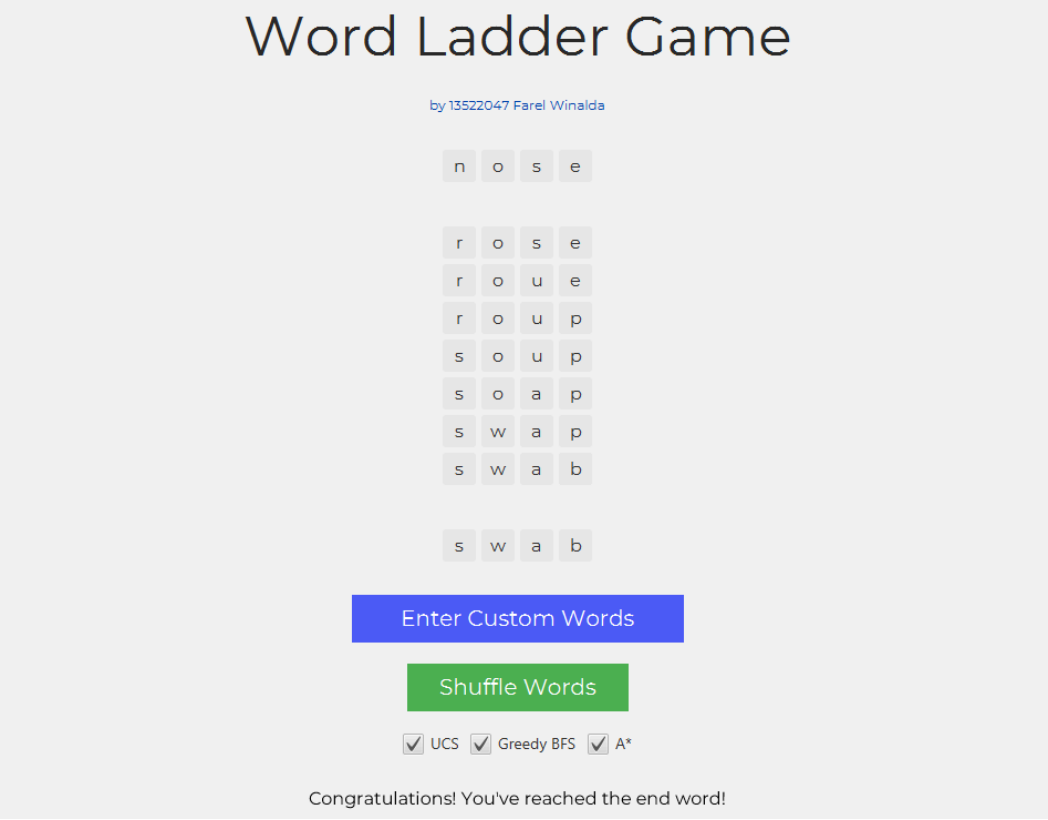
3.1.5. Pengujian 7 Kata dari *readers* ke *writers*



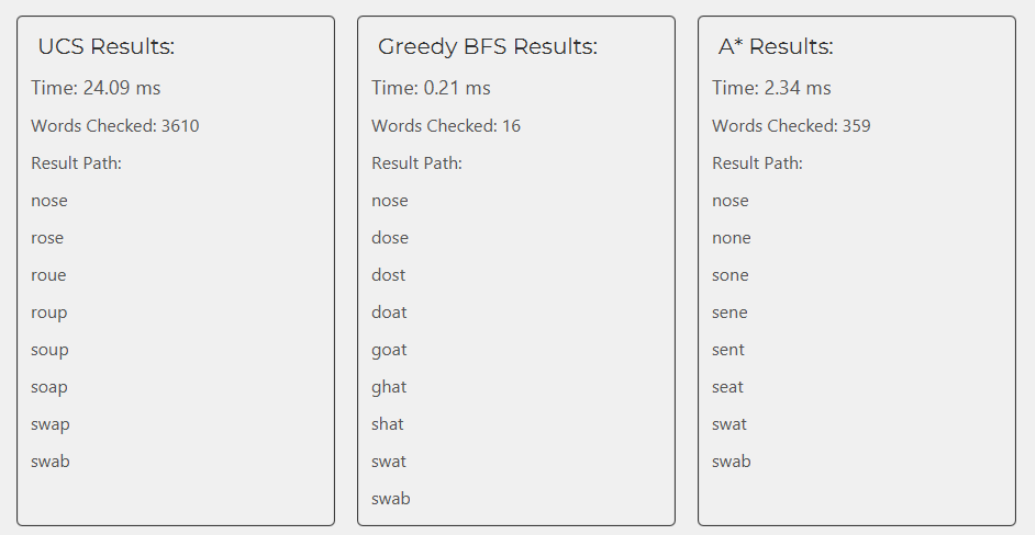
Gambar 3.1.5.1 *Tampilan GUI dari readers ke writers*

Gambar 3.1.5.2 *Tampilan hasil algoritma readers ke writers*

3.1.6. Pengujian Kata dari *nose* ke *swab*

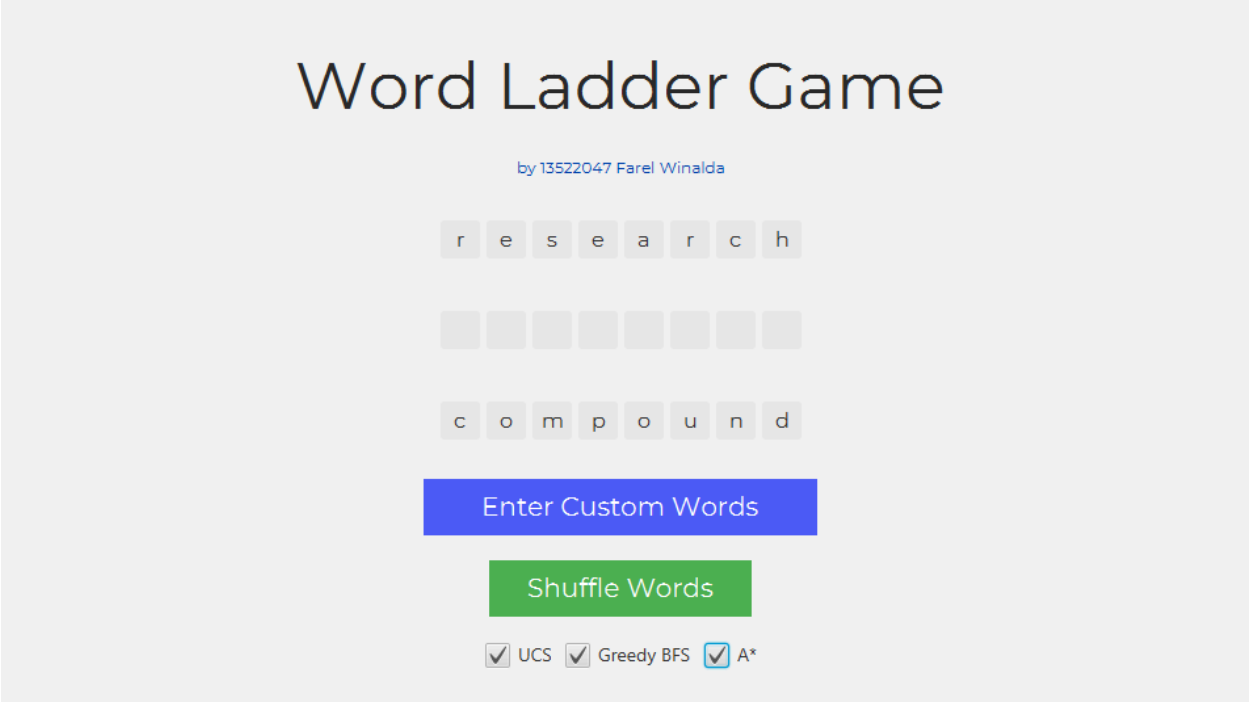


Gambar 3.1.6.1 *Tampilan GUI dari nose ke swab*

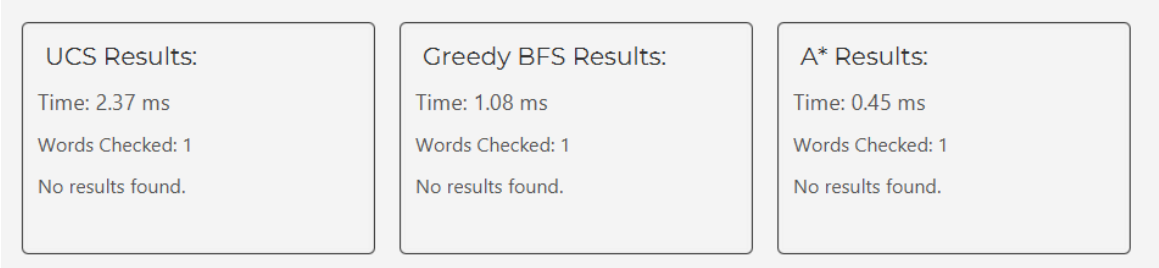


Gambar 3.1.6.2 *Tampilan hasil algoritma nose ke swab*

3.1.7. Pengujian Kata dari research ke compound



Gambar 3.1.7.1 *Tampilan GUI dari research ke compound*



Gambar 3.1.7.2 *Tampilan hasil algoritma research ke compound*

3.2. Analisis

Dari implementasi Algoritma UCS, Greedy Best First Search, dan A*. Berikut langkah-langkah dari masing-masing algoritma.

Langkah-langkah algoritma Uniform Cost Search:

- Dimulai dari Node awal dengan menetapkan biaya awal nol
- Simpul disimpan dalam priorityqueue yang diurutkan berdasarkan biaya path (g(n)) dari node awal, dengan g(n) adalah biaya dari simpul awal ke n
- Node dengan biaya terendah dikeluarkan dari queue dan memeriksa lagi semua node tetangga yang dapat diakses

- Jika node tetangga bisa dicapai dengan biaya lebih rendah, biaya akan diupdate dan enqueue kembali
- Pengulangan proses sampai node tujuan berhasil dietemukan

Langkah-langkah algoritma Greedy Best First Search:

- Dimulai dari Node awal
- Simpul dimasukkan ke dalam priorityqueue yang diurutkan berdasarkan heuristik (h(n)) yang memperkirakan jarak ke suatu tujuan, dengan h(n) adalah heuristik dari n ke tujuan
- Node dengan estimasi heuristik terbaik dikeluarkan dari queue dan memeriksa lagi node tetangga
- Node yang telah dikunjungi tidak dijelajahi lagi
- Pengulangan proses sampai node tujuan berhasil dietemukan

Langkah-langkah algoritma A* Search:

- Dimulai dari Node awal
- Simpul disimpan dalam priorityqueue yang diurutkan berdasarkan f(n) = g(n) + h(n), dengan f(n) adalah estimasi total biaya minimum dari node awal melalui node n ke tujuan
- Node dengan f(n) terkecil dikeluarkan dan memeriksa kembali semua node tetangga yang dapat diakses
- Jika node baru bisa dicapai dengan biaya lebih rendah, biaya akan diupdate dan enqueue kembali
- Pengulangan proses sampai node tujuan berhasil ditemukan

Berdasarkan penjelasan tersebut, g(n) adalah jumlah langkah atau transformasi yang diperlukan untuk mencapai simpul n dari simpul awal. Fungsi ini berfungsi untuk memastikan bahwa jalur yang ditemukan tidak hanya menuju ke tujuan tetapi juga merupakan jalur yang efisien dari segi biaya atau langkah yang dikeluarkan. h(n) adalah heuristik, yaitu perkiraan biaya dari simpul n ke tujuan. Fungsi ini membantu algoritma untuk membuat keputusan yang cerdas tentang simpul mana yang harus dijelajahi selanjutnya dengan memprioritaskan simpul-simpul yang tampaknya lebih dekat ke tujuan. f(n) adalah agregasi dari dua fungsi sebelumnya, yaitu f(n)=g(n)+h(n). Dengan

menggunakan kombinasi dari g(n) dan h(n) untuk menghitung f(n), A* Search berhasil menyeimbangkan antara kebutuhan untuk mendekati tujuan secara cepat (melalui h(n)) dan kebutuhan untuk mencari jalur yang biaya totalnya minimal (melalui g(n)).

Dalam konteks WordLadder, heuristik yang digunakan adalah jumlah huruf yang tidak cocok, sehingga jika kata saat ini adalah "heat" dan kata tujuan adalah "cold", maka heuristik akan menghitung bahwa semua empat huruf berbeda, sehingga menghasilkan skor heuristik 4. Ini memberikan estimasi tentang berapa banyak langkah minimal yang mungkin diperlukan untuk mengubah kata saat ini menjadi kata tujuan, asumsi setiap langkah mengubah satu huruf menjadi huruf yang benar.

Sebuah heuristik dianggap admissible jika memenuhi kriteria khusus: heuristik tersebut tidak pernah memperkirakan biaya untuk mencapai tujuan yang lebih tinggi dari biaya sebenarnya. Dengan kata lain, heuristik harus selalu optimistik, memberikan batas bawah dari biaya aktual untuk mencapai tujuan dari simpul yang diberikan. Dalam WordLadder, heuristik dianggap admissible karena setiap huruf yang berbeda antara kata saat ini dan kata tujuan harus diubah untuk mencapai kata tujuan, yang berarti setiap perbedaan huruf dihitung sebagai setidaknya satu langkah transformasi. Sehingga tidak pernah melebih-lebihkan jumlah langkah yang diperlukan untuk mengubah kata awal menjadi kata tujuan. Setiap langkah dalam Word Ladder melibatkan penggantian satu huruf, dan heuristik hanya menghitung jumlah huruf yang perlu diganti.

Dalam WordLadder, algoritma Uniform Cost Search (UCS) dan Breadth-First Search (BFS) seringkali beroperasi dengan cara yang sangat mirip, namun ada beberapa nuansa penting yang membedakan keduanya dalam hal implementasi dan operasi. Kemiripan antara kedua algoritma ini yaitu, setiap transisi memiliki biaya yang sama, UCS yang mengurutkan node berdasarkan biaya kumulatif dari node awal tidak akan berbeda dalam urutan penjelajahan dibandingkan dengan BFS yang menjelajahi semua node pada kedalaman tertentu sebelum beralih ke kedalaman berikutnya. Kedua algoritma ini akan menghasilkan path yang sama dalam kasus ini, karena keduanya akan menemukan path terpendek ke tujuan. Namun, terdapat perbedaan UCS menggunakan priority queue untuk memilih node berikutnya yang akan dijelajahi, berdasarkan biaya path dari awal. BFS, di sisi lain, menggunakan queue biasa yang beroperasi berdasarkan prinsip first-in-first-out. UCS secara intrinsik dirancang untuk menangani graf dengan

biaya edge yang bervariasi, sedangkan BFS secara asli hanya mempertimbangkan jumlah langkah dari node awal tanpa memperhitungkan biaya transisi.

Secara teoritis, algoritma A* biasanya dianggap lebih efisien daripada Uniform Cost Search (UCS) dikarenakan A* menggabungkan elemen dari UCS (menggunakan biaya total g(n) dari node awal ke node n) dan Greedy Best First Search (menggunakan heuristik h(n) yang memperkirakan biaya dari node n ke tujuan). Jika heuristik yang digunakan adalah admissible maka A* secara teoritis akan mencapai efisiensi yang lebih tinggi karena heuristik membantu mengarahkan pencarian lebih langsung ke tujuan. Dengan menggunakan heuristik, A* dapat mengabaikan jalur yang kurang mungkin mencapai tujuan lebih cepat, yang berarti bahwa jumlah node yang dieksplorasi sebelum menemukan jalur optimal bisa jauh lebih sedikit dibandingkan dengan UCS.

Secara teoritis, algoritma Greedy Best First Search (GBFS) tidak menjamin solusi optimal karena algoritma ini menggunakan pendekatan heuristik yang sangat berfokus pada tujuan tanpa mempertimbangkan biaya yang telah dikeluarkan dari titik awal. Dengan kata lain, Greedy Best First Search dapat mengabaikan jalur paling optimal yaitu jalur terpendek terutama dalam persoalan yang lebih kompleks atau ketika biaya untuk mencapai berbagai simpul bervariasi secara signifikan, namun algoritma ini lebih cenderung menemukan solusi yang cepat daripada solusi yang optimal.

Dari ketiga algoritma tersebut, didapatkan data-data yang membandingkan algoritma mana yang paling optimal dan paling efisien, sehingga dapat dikelompokkan berdasarkan optimalitas (jumlah kata), waktu eksekusi, jumlah kata yang dikunjungi, serta memori yang dibutuhkan. Berikut hasil dari data-data tersebut:

Algoritma	**TC 1**	**TC 2**	**TC 3**	**TC 4**	**TC 5**	**TC 6**
UCS	4	5	10	23	7	8
GBFS	4	5	13	31	10	9
A*	4	5	10	23	7	8

Tabel 3.2.1 *Tabel perbandingan jumlah kata dari tiap algoritma*

Dari hasil ***Tabel 3.2.1*** didapatkan data bahwa UCS dan A* memiliki hasil yang optimal dan selalu memiliki jumlah hasil kata yang sama dibandingkan dengan GBFS karena pada GBFS lebih mengutamakan heuristik tanpa mempertimbangkan biaya / cost. Sedangkan, UCS akan selalu mencari hasil optimal yang memiliki biaya terendah dan A* yang juga demikian namun berdasarkan pertimbangan heuristik.

Algoritma	**TC 1**	**TC 2**	**TC 3**	**TC 4**	**TC 5**	**TC 6**
UCS	12.82	12.80	95.74	86.07	6.55	24.09
GBFS	1.80	0.20	0.55	7.93	0.20	0.21
A*	0.99	0.50	3.52	116.27	0.50	2.34

Tabel 3.2.2 *Tabel perbandingan waktu eksekusi algoritma (dalam milisekon)*

Dari hasil ***Tabel 3.2.2*** didapatkan data bahwa GBFS memiliki waktu eksekusi yang paling cepat disusul dengan A* nomor 2 dan UCS yang paling lambat dikarenakan GBFS selalu memilih node yang tampaknya paling dekat dengan tujuan, berdasarkan heuristik dibandingkan dengan A* yang menghindari eksplorasi beberapa jalur yang jelas tidak efisien, yang tidak dilakukan oleh UCS. Namun, A* masih lebih lambat dibandingkan GBFS karena harus menghitung biaya total juga, bukan hanya mengandalkan heuristik. Sedangkan UCS murni berbasis pada biaya, ini tidak menggunakan informasi heuristik. Ini menyebabkan eksplorasi yang luas dari semua jalur yang mungkin, tidak peduli seberapa jauh atau dekatnya mereka dengan tujuan secara heuristik.

Algoritma	**TC 1**	**TC 2**	**TC 3**	**TC 4**	**TC 5**	**TC 6**
UCS	678	897	4347	8315	693	3610
GBFS	4	5	25	591	14	16
A*	6	9	285	7257	48	359

Tabel 3.2.3 *Tabel perbandingan jumlah kata yang dikunjungi / dicek*

Dari hasil ***Tabel 3.2.3*** didapatkan data bahwa GBFS menjadi algoritma yang hanya sedikit melakukan pengunjungan kata disusul oleh A* nomor dua dan UCS yang melakukan pengunjungan kata terbanyak dikarenakan GBFS hanya mengandalkan heuristik tanpa memperhatikan total biaya, dan berbeda dengan A* dan UCS yang memperhitungkan biaya.

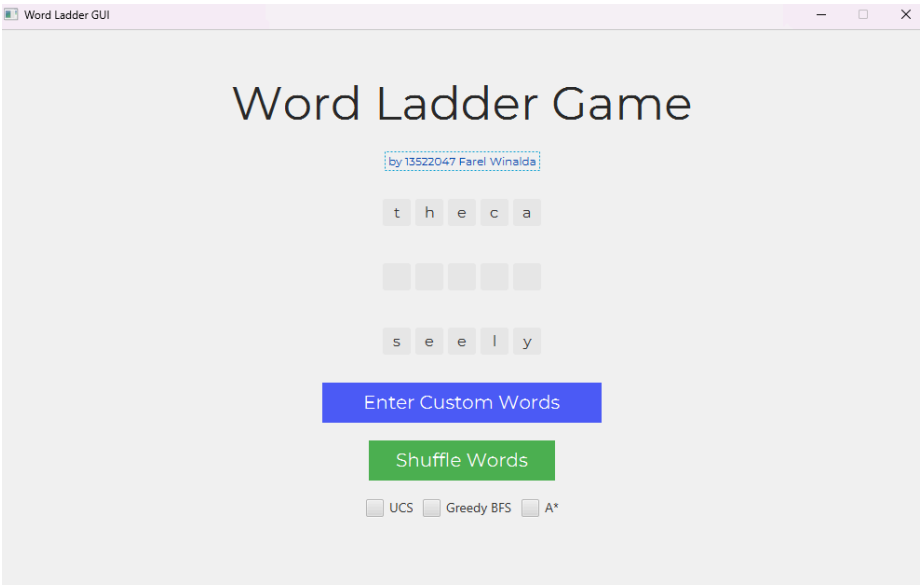
Algoritma	**TC 1**	**TC 2**	**TC 3**	**TC 4**	**TC 5**	**TC 6**
UCS	2823424	4907744	9666368	23101536	5387136	18874368
GBFS	523048	513776	525336	2932464	476312	474280
A*	612112	544488	2097152	11710016	667360	2454912

Tabel 3.2.4 *Tabel perbandingan jumlah memori yang dibutuhkan (dalam bytes)*

Dari hasil ***Tabel 3.2.4*** didapatkan data bahwa GBFS menjadi algoritma yang memakai memori paling sedikit dalam pencariannya disusul dengan A* nomor dua dan UCS yang memakan paling banyak memori dikarenakan GBFS mengutamakan pencarian berdasarkan heuristik yang memandu pencarian ke arah tujuan tanpa memperhatikan biaya sebenarnya dari path yang ditempuh. Ini mengurangi jumlah node yang perlu disimpan dalam memori pada satu waktu. Lalu, A* menggunakan kombinasi dari cost untuk mencapai node saat ini dan heuristik untuk memperkirakan cost ke tujuan (Metode pengambilan data yang diambil yang membuat A* lebih efisien daripada UCS dalam hal jumlah node yang dieksplorasi karena A* bisa lebih fokus pada path yang potensial, namun tetap mempertahankan banyak path kandidat dalam memori untuk memastikan solusi yang optimal. Sedangkan, UCS mengeksplorasi semua kemungkinan jalur hingga menemukan jalur terpendek, sering kali membutuhkan penyimpanan banyak node dalam antrian prioritas. Ini dapat menyebabkan penggunaan memori yang signifikan karena setiap node potensial bersama dengan seluruh jalurnya dari node awal harus disimpan dalam memori hingga solusi ditemukan atau semua kemungkinan telah dijelajahi. dari (Metode pengambilan data yang diambil dari ***Tabel 3.2.4*** tidak berasal dari tampilan GUI, melainkan secara manual menggunakan CLI).

3.3. Bonus

Bonus yang dikerjakan yaitu bonus GUI yang dibuat dengan menggunakan library *JavaFX* dengan menggunakan bahasa *Java* dan juga menerapkan *xml (Extensible Markup Language)* dalam pembuatan komponen-komponen UI, serta menggunakan *CSS (Cascading Style Sheets)* untuk melakukan styling pewarnaan serta posisi UI yang dibuat. Pada GUI yang dibuat ini terdapat sebuah *startword* dan *endword* yang dipilih secara acak oleh sistem yang memiliki panjang kata yang sama. Pengguna akan diberikan input untuk menebak kata yang cocok seperti pada website permainan Word Ladder ***Gambar 1*** (*sumber: https://wordwormdormdork.com/)* dengan aturan permainan yang sama yaitu kata harus valid dan hanya boleh mengubah satu huruf saja. Namun, diberikan opsi untuk mengetahui jawaban / hint menggunakan 3 macam algoritma pencarian UCS, GBFS, dan A*. Diberikan pula opsi untuk memasukkan *startword* dan *endword* secara manual dan diberikan opsi untuk mengacak ulang *startword* dan *endword*.



Gambar 3.3. *Tampilan GUI menggunakan JavaFX*

BAB IV
KESIMPULAN DAN SARAN

4.1. Kesimpulan

Dari hasil data yang didapatkan pada bab 3, dari ketiga algoritma pencarian dalam konteks Word Ladder, yang terdiri dari Uniform Cost Search, Greedy Best First Search, dan A*, dapat ditarik beberapa kesimpulan. Pertama, algoritma Uniform Cost Search, algoritma ini merupakan algoritma yang optimal dalam konteks mencari jalur kata terpendek, namun algoritma ini mempunyai kekurangan yaitu memiliki waktu eksekusi yang lama, pengecekan node yang banyak dan juga menggunakan banyak memori penyimpanan yang disebabkan karena algoritma ini mempertimbangkan dan mengecek semua kemungkinan sehingga algoritma ini mempertahankan banyak node, namun hasil yang dihasilkan tetap optimal karena memperhitungkan biaya yang jika selesai membandingkan semua biaya akan didapatkan solusi yang paling sedikit dan optimal. Lalu, Greedy Best First Search, alforitma ini mempunyai kelebihan yaitu waktu eksekusi yang cepat dan penggunaan memori yang sedikit, namun kekurangan dari algoritma ini terletak pada heuristiknya yang membuat algoritma ini hanya membuat keputusan yang tampak lebih dekat dengan tujuan yang terkadang dapat melewatkan hasil yang optimal dikarenakan algoritma Greedy yang dipakai, sehingga algoritma ini cocok digunakan untuk kecepatan pencarian dan keakuratan tidak terlalu dipentingkan. Lalu, algoritma A*, algoritma ini bisa dikatakan merupakan gabungan dari UCS dan GBFS karena mempertimbangkan cost dan heuristik. Dari segi optimalitas A* bersama dengan UCS merupakan algoritma yang unggul, namun dengan waktu eksekusi, pengecekan kata, maupun pemakaian memori A* jauh mengungguli UCS. Berbeda dengan GBFS yang mengorbankan optimalitas hasil, A* sejauh ini selalu menghasilkan hasil yang optimal dengan waktu eksekusi yang terbilang cukup singkat. Sehingga dalam konteks permainan Word Ladder, A* adalah algoritma yang paling optimal dan efisien.

4.2. Saran

Untuk meningkatkan kecepatan pencarian dan optimalitas hasil dalam konteks Word Ladder, heuristik memegang peran penting dalam penentuan keputusan algoritma baik Greedy Best First Search maupun A*, sehingga pengembangan heuristik diperlukan untuk menghasilkan keakuratan yang lebih. Penggunaan memori yang besar juga sebaiknya perlu dihindari dengan cara menerapkan caching untuk mengurangi pemborosan memori dan juga mempercepat pencarian.

DAFTAR PUSTAKA

Munir, Rinaldi. Penentuan rute (Route/Path Planning) - Bagian 1. Diakses 5 Mei 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf

Munir, Rinaldi. Penentuan rute (Route/Path Planning) - Bagian 2. Diakses 5 Mei 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf

LAMPIRAN

Repository Github:

https://github.com/FarelW/Tucil3_13522047

Tabel Spesifikasi:

Poin	Ya	Tidak
1. Program berhasil dijalankan.	✔	
2. Program dapat menemukan rangkaian kata dari startword 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	✔	