Laporan Tugas Kecil 3

Strategi Algoritma

IF 2211

Disusun Oleh :

Rayhan Fadhlan Azka

13522095

Algoritma Program

1. Uniform First Search (UCS)

Uniform First Search atau UCS adalah algoritma searching yang dimana algoritma ini pasti menghasilkan solusi yang paling optimal, dimana optimal disini adalah jarak terpendak dari start ke tujuan. Dalam perhitungan algoritma UCS, digunakan fungsi evaluasi f(n) = g(n) atau cost dari root node ke node sekarang. Pada program word ladder, algoritma UCS bekerja dengan cara menambahkan seluruh node dengan perbedaan huruf pada kedua kata adalah 1 ke priority queue. Pada word ladder ini g(n) dihitung sebagai jarak dari kata awal ke kata yang sekarang, misal dari “car” ke “cop” jaraknya adalah “car”, “cap”, “cop” , maka nilai g(n) untuk cop adalah 2. Langkah-langkah yang digunakan program dalam penyelesaian word ladder menggunakan UCS adalah :

1. Buatlah sebuah priority queue, dimana antrian terdepan adalah node dengan nilai g(n) terkecil.
2. Masukkan kata awal dengan bobot g(n) 0.
3. Cek apakah kata yang diproses pada queue merupakan kata tujuan, jika iya pemrosesan di stop, jika tidak, lanjut ke step 4.
4. Lakukan ekspansi dengan cara untuk setiap kata dengan perbedaan huruf 1 dengan kata awal, cek apakah kata tersebut sudah pernah diproses sebelumnya, jika tidak ,masukkan ke priority queue sebagai node dengan g(n) = g(n) parentnya + 1, dan juga mencatat parent dari kata tersebut.
5. Ulangi step 2 tetapi dengan pemrosesannya bukan kata awal, melainkan node pada priorityqueue dengan bobot g(n) terkecil.
6. Jika sudah menemukan node yang katanya sama dengan kata tujuan, maka akan di generate path dari kata awal ke kata tujuan dengan cara backtracking dari node tujuan sampai node awal (pada setiap node dicatat parent dari node tersebut).

Pada pencarian ini, solusi yang dihasilkan sudah pasti optimal (jarak terpendek) akan tetapi pencarian secara UCS memiliki kekurangan yaitu waktu yang dibutuhkan lebih lama dibandingkan kedua algoritma selanjutnya, hal ini dikarenakan UCS akan menelusuri seluruh node yang diekspan terlebih dahulu sampai menemukan tujuannya. Dalam kasus permainan word ladder ini pembangkitan simpul dan pemrosesan dengan UCS urutannya sama seperti BFS, hal ini diakibatkan oleh perbedaan cost diantara satu simpul dan simpul anaknya sudah pasti 1 sesuai dengan cara kerja word ladder yang dimana hanya bisa mengganti satu huruf di setiap gerakan. Path yang dihasilkan pun akan sama juga dengan BFS, karena alasan yang sama.

2. **Greedy Best First Search (Greedy BFS)**

Greedy Best First Search adalah algoritma searching yang dimana algoritma ini memprioritaskan langkah melewati simpul-simpul yang pada saat itu terlihat lebih optimal sesuai dengan fungsi heuristik yang telah didefinisikan. Fungsi evaluasi pada Greedy Best First Search adalah f(n) = h(n) dimana h(n) adalah estimasi cost dari node saat ini menuju node goal. Fungsi heuristik atau h(n) yang saya gunakan pada pemecahan word ladder adalah jumlah huruf yang berbeda pada kata yang sekarang dengan kata tujuan, dengan jumlah huruf berbeda paling sedikit maka akan diprioritaskan pada priority queue. Cara kerja algoritma ini adalah :

1. Siapkan priority queue dimana antrian terdepan adalah node dengan nilai h(n) terkecil.
2. Masukkan kata awal dengan bobot h(n) 0.
3. Cek apakah kata yang diproses pada queue merupakan kata tujuan, jika iya pemrosesan di stop, jika tidak, lanjut ke step 4.
4. Lakukan ekspansi dengan cara untuk setiap kata dengan perbedaan huruf 1 dengan kata awal, cek apakah kata tersebut sudah pernah diproses sebelumnya, jika tidak ,masukkan ke priority queue sebagai node dengan h(n) = jumlah huruf pada kata tersebut yang berbeda dengan kata tujuan.
5. Ulangi step 2 tetapi dengan pemrosesannya bukan kata awal, melainkan node pada priorityqueue dengan bobot h(n) terkecil.
6. Jika sudah menemukan node yang katanya sama dengan kata tujuan, maka akan di generate path dari kata awal ke kata tujuan dengan cara backtracking dari node tujuan sampai node awal (pada setiap node dicatat parent dari node tersebut).

Greedy Best First Search tidak menjamin solusi yang optimal, hal ini dikarenakan pada algoritma ini, simpul yang diprioritaskan adalah simpul yang secara sekilas terlihat lebih dekat ke simpul tujuan, padahal hal ini belum tentu benar. Keunggulan dari algoritma Greedy BFS ini adalah pemrosesan yang paling cepat diantara kedua algoritma yang lain dikarenakan, pada Greedy BFS tidak mengunjungi simpul secara berurutan, akan tetapi mengunjungi simpul yang terlihat lebih dekat ke tujuan terlebih dahulu.

3. **A***

Pencarian menggunakan algoritma A* adalah pencarian dengan fungsi evaluasi heuristik f(n) = g(n) + h(n), yang dimana f(n) adalah cost dari root node sampai ke node sekarang (g(n)) ditambah dengan estimasi cost dari simpul sekarang ke simpul tujuan (h(n)), atau bisa dibilang algoritma A* ini adalah gabungan dari algoritma Greedy BFS dan algoritma UCS. Secara teoritis algoritma ini lebih efisien dibanding algoritma UCS pada kasus permainan word ladder, hal ini karena pada algoritma A* terdapat fungsi evaluasi tambahan h(n) dalam menentukan simpul mana yang harus dikunjungi sehinga total simpul

yang dikunjungi pada algoritma A* selalu lebih sedikit dibandingkan UCS. Cara kerja algoritma A* adalah :

1. Siapkan priority queue dimana antrian terdepan adalah node dengan nilai f(n) = g(n) + h(n) terkecil.
2. Masukkan kata awal dengan bobot f(n) = 0.
3. Cek apakah kata yang diproses pada queue merupakan kata tujuan, jika iya pemrosesan di stop, jika tidak, lanjut ke step 4.
4. Lakukan ekspansi dengan cara untuk setiap kata dengan perbedaan huruf 1 dengan kata awal, cek apakah kata tersebut sudah pernah diproses sebelumnya, jika tidak ,masukkan ke priority queue sebagai node dengan f(n) = g(n) + h(n) dimana f(n) = (5 + f(n) parentnya) + jumlah kata yang berbeda dengan kata tujuan.
5. Ulangi step 2 tetapi dengan pemrosesannya bukan kata awal, melainkan node pada priorityqueue dengan bobot f(n) terkecil.
6. Jika sudah menemukan node yang katanya sama dengan kata tujuan, maka akan di generate path dari kata awal ke kata tujuan dengan cara backtracking dari node tujuan sampai node awal (pada setiap node dicatat parent dari node tersebut).

Pencarian A* dengan fungsi heuristik f(n) = g(n) + h(n) dimana f(n) = (5 + f(n) parentnya) + jumlah kata yang berbeda dengan kata tujuan admissible, hal ini dikarenakan dalam hal ini fungsi h(n) yang digunakan akan selalu underestimate cost sebenarnya dari simpul itu ke simpul tujuan. h(n) disini akan selalu underestimate karena untuk setiap kedalaman, fungsi g(n) tidak di inkremen dengan 1 seperti UCS, tetapi di inkremen dengan 5. Karena fungsi heuristik admissible, algoritma A* dengan fungsi heuristik ini akan selalu menghasilkan solusi optimal.

Source Code Program

Program ini dibuat dengan bahasa Java serta memanfaatkan konsep pemrograman berorientasi objek.

1. Kelas Dictionary, yaitu kelas yang memparsing dictionary text file menjadi dictionary hashSet pada java.

```java
package util;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashSet;

public class Dictionary {
    public  HashSet<String> dictionary;
    public Dictionary() throws IOException {
        dictionary = new HashSet<>();
        String path  = System.getProperty("user.dir") + "/../src/dictionary.txt";

        BufferedReader reader = new BufferedReader(new FileReader(path));
        String line;

        while ((line = reader.readLine()) != null) {
            String lowerCaseLine = line.toLowerCase();


            dictionary.add(lowerCaseLine);

        }

        reader.close();

    }
    public HashSet<String> getDictionary() {
        return dictionary;
    }
    public boolean contains(String word) {
        return dictionary.contains(word);
    }

}


```

2. Kelas Node, yang berisi string, parent serta cost dari node tersebut

```java
package algo;

public class Node implements Comparable<Node> {

    public String word;
    public Node parent;
    public int cost;

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

```
        return word;
    }
    public Node getParent() {
        return parent;
    }
    public int getCost() {
        return cost;
    }

    public int compareTo(Node node2) {
        return Integer.compare(this.getCost(), node2.getCost());
    }

}
```

3. Kelas Search, yaitu kelas abstrak sebagai parent dari ketiga algoritma searching, UCS, Greedy BFS, dan A*

```
package algo;

import util.Dictionary;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.PriorityQueue;

public abstract class Search {

    public static PriorityQueue<Node> pq;
    public static HashSet<String> visited;
    public static int traversed;

    public Search(){
        pq = new PriorityQueue<>();
        visited = new HashSet<>();
        traversed = 0;
    }

    public ArrayList<String> generatePath(Node endNode){
        ArrayList<String> path = new ArrayList<>();
        Node current = endNode;

        // Backtrack from the end node to the start node
        while (current != null) {
            path.add(current.getWord());
            current = current.getParent();
        }

        // Reverse the path since we started from the end node
        Collections.reverse(path);
        return path;
    }

    public int getTraversed(){
        return traversed;
    }

    public ArrayList<String> search(String start, String end, Dictionary dict){
        pq.clear();
        visited.clear();
        traversed = 0;

        Node first = new Node(start,null,0);
```

```
47.
48.         pq.add(first);
49.         visited.add(start);
50.         while(!pq.isEmpty()){
51.             traversed++;
52.             Node cur = pq.remove();
53.             if(cur.getWord().equals(end)){
54.                 return generatePath(cur);
55.             }
56.
57.
58.
59.             for(int i = 0; i < cur.getWord().length(); i++) {
60.                 for (char c = 'a'; c <= 'z'; c++) {
61.                     if (cur.getWord().charAt(i) == c) {
62.                         continue;
63.                     }
64.                     StringBuilder temp = new StringBuilder(cur.getWord());
65.                     temp.setCharAt(i, c);
66.                     String newWord = temp.toString();
67.                     if(visited.contains(newWord)){
68.                         continue;
69.                     }
70.                     if(dict.contains(newWord) ){
71.                         pq.add(new Node(newWord,cur, calculateCost(cur,end)));
72.                         visited.add(newWord);
73.                     }
74.
75.                 }
76.             }
77.         }
78.         return new ArrayList<>();
79.     }
80.
81.     abstract public int calculateCost(Node node1, String endString);
82.
83. }
84.
```

Pada kelas ini terdapat method search yang nantinya akan digunakan dalam setiap algoritma, pembeda dari setiap algoritma adalah cara masing-masing dalam meng generate cost, UCS dengan g(n), Greedy BFS dengan h(n) dan A* dengan g(n) + h(n)

4. Kelas UCS, kelas ini bertujuan untuk solve word ladder menggunakan UCS

```
 1. package algo;
 2.
 3. import java.util.*;
 4.
 5. import util.Dictionary;
 6.
 7. public class UCS extends Search {
 8.
 9.     public UCS() {
10.         super();
11.     }
12.
13.     public ArrayList<String> searchUCS(String start, String end, Dictionary dict) {
14.         return search(start, end, dict);
15.     }
16.     // self explanatory, the cost of the node is incremented by 1
17.     @Override
18.     public int calculateCost(Node node1, String endString) {
19.         return node1.getCost() + 1;
```

```
    }

}

```

5. Kelas Greedy BFS, kelas ini bertujuan untuk solve word ladder dengan greedy BFS

```
package algo;

import util.Dictionary;

import java.util.ArrayList;

public class GreedyBFS extends Search{

    public GreedyBFS(){
        super();
    }
    public ArrayList<String> searchGreedyBFS(String start, String end, Dictionary dict){
        return search(start,end,dict);
    }

    // Heuristic yang digunakan adalah jumlah karakter yang berbeda antara node sekarang dengan node tujuan
    @Override
    public int calculateCost(Node node1, String endString) {
        int ret = 0;
        String nodeString = node1.getWord();
        for(int i = 0;i<nodeString.length();i++){
            if(nodeString.charAt(i)!=endString.charAt(i)){
                ret++;
            }
        }
        return ret;
    }

}


```

6. Kelas A*, kelas ini bertujuan untuk solve word ladder menggunakan A*

```
package algo;

import util.Dictionary;

import java.util.ArrayList;

public class AStar extends Search {
    public AStar() {
        super();
    }

    public ArrayList<String> searchAstar(String start, String end, Dictionary dict){
        return search(start,end,dict);
    }

    @Override
    public int calculateCost(Node node1, String endString){
        int ret = 0;
```

```
        String nodeString = node1.getWord();
        for(int i = 0;i<nodeString.length();i++){
            if(nodeString.charAt(i)!=endString.charAt(i)){
                ret++;
            }
        }

        // menjadikan heuristic admissible, dengan cara mengunderestimate h(n), dalam hal ini h(n) adalah ret yaitu estimasi jarak node sekaran ke node tujuan
        // dengan cara ini, algoritma A* akan selalu menghasilkan solusi optimal
        // karena nilai h(n) tidak akan pernah melebihi nilai sebenarnya, karena di setiap step, nilai g(n) di increment dengan 5, bukan 1
        return (node1.getCost() + 5) + ret;
    }

}


```

7. Kelas Word Ladder GUI

```
package gui;
import algo.AStar;
import algo.GreedyBFS;
import algo.UCS;
import util.Dictionary;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.IOException;
import java.util.ArrayList;

public class WordLadderGUI extends JFrame implements ActionListener {

    private JTextField startWordField;
    private JTextField endWordField;
    private JTextArea pathArea;
    private JButton searchButton;
    private JButton resetButton;
    private JLabel statusLabel;
    private JComboBox<String> algorithmComboBox;
    private Dictionary dict;
    private UCS ucs;
    private GreedyBFS gbfs;
    private AStar astar;
    public WordLadderGUI() throws IOException {
        super("Word Ladder");
        ucs = new UCS();
        gbfs = new GreedyBFS();
        astar = new AStar();
        dict = new Dictionary();
        // Set the look and feel to Nimbus
        try {
            UIManager.setLookAndFeel("javax.swing.plaf.nimbus.NimbusLookAndFeel");
        } catch (ClassNotFoundException | InstantiationException | IllegalAccessException | UnsupportedLookAndFeelException e) {
            e.printStackTrace();
        }
        pathArea = new JTextArea(20,40);
        pathArea.setEditable(false);

        // Create UI components
        JPanel inputPanel = new JPanel(new GridLayout(2, 2, 10, 10));
        JLabel startWordLabel = new JLabel("Start Word:");
```

```
        startWordField = new JTextField(10);
        JLabel endWordLabel = new JLabel("End Word:");
        endWordField = new JTextField(10);
        inputPanel.add(startWordLabel);
        inputPanel.add(startWordField);
        inputPanel.add(endWordLabel);
        inputPanel.add(endWordField);

        JPanel algorithmPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
        JLabel algorithmLabel = new JLabel("Select Algorithm:");
        String[] algorithms = {"UCS", "A*", "Greedy BFS"};
        algorithmComboBox = new JComboBox<>(algorithms);
        algorithmPanel.add(algorithmLabel);
        algorithmPanel.add(algorithmComboBox);

        JPanel buttonPanel = new JPanel(new FlowLayout(FlowLayout.CENTER, 20, 10));
        searchButton = new JButton("Search");
        searchButton.addActionListener(this);
        resetButton = new JButton("Reset");
        resetButton.addActionListener(this);
        buttonPanel.add(searchButton);
        buttonPanel.add(resetButton);

        JScrollPane scrollPane = new JScrollPane(pathArea);

        statusLabel = new JLabel(" ");
        statusLabel.setHorizontalAlignment(SwingConstants.CENTER);

        // Layout components
        JPanel mainPanel = new JPanel(new BorderLayout(10, 10));
        mainPanel.setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20));
        mainPanel.add(inputPanel, BorderLayout.NORTH);
        mainPanel.add(algorithmPanel, BorderLayout.CENTER);
        mainPanel.add(scrollPane, BorderLayout.SOUTH);
        mainPanel.add(buttonPanel, BorderLayout.EAST);
        mainPanel.add(statusLabel, BorderLayout.WEST);

        // Add main panel to frame and display
        getContentPane().add(mainPanel);
        pack();
        setLocationRelativeTo(null); // Center the window on the screen
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == searchButton) {

            String startWord = startWordField.getText().toLowerCase();
            String endWord = endWordField.getText().toLowerCase();


            if (startWord.length() != endWord.length()) {

                pathArea.setText("Start word and end word must have the same length.");
                statusLabel.setText("");
                return;
            }
            if (!dict.contains(startWord)) {

                pathArea.setText("Start word not found in the dictionary.");
                statusLabel.setText("");
                return;
            }

            if (!dict.contains(endWord)) {

                pathArea.setText("End word not found in the dictionary.");
                statusLabel.setText("");
```

```
114.                 return;
115.             }
116.             String selectedAlgorithm = (String) algorithmComboBox.getSelectedItem();
117.             ArrayList<String> path;
118.             long startTime = 0;
119.             long endTime = 0;
120.             int totalTraversed = 0;
121.             switch (selectedAlgorithm) {
122.                 case "UCS":
123.                     startTime = System.currentTimeMillis();
124.                     path = ucs.searchUCS(startWord, endWord,dict);
125.                     totalTraversed = ucs.getTraversed();
126.                     endTime = System.currentTimeMillis();
127.                     break;
128.                 case "A*":
129.                     startTime = System.currentTimeMillis();
130.                     path = astar.searchAstar(startWord, endWord,dict);
131.                     totalTraversed = ucs.getTraversed();
132.                     endTime = System.currentTimeMillis();
133.                     break;
134.                 case "Greedy BFS":
135.                     startTime = System.currentTimeMillis();
136.                     path = gbfs.searchGreedyBFS(startWord, endWord,dict);
137.                     totalTraversed = ucs.getTraversed();
138.                     endTime = System.currentTimeMillis();
139.                     break;
140.                 default:
141.                     path = new ArrayList<>();
142.                     break;
143.             }
144.
145.
146.             long elapsedTime = endTime - startTime;
147.
148.
149.             updatePathArea(path, elapsedTime, totalTraversed);
150.
151.             statusLabel.setText("");
152.         } else if (e.getSource() == resetButton) {
153.             // Clear all fields
154.             startWordField.setText("");
155.             endWordField.setText("");
156.             pathArea.setText("");
157.             statusLabel.setText("");
158.         }
159.     }
160.
161.     private void updatePathArea(ArrayList<String> path, long elapsedTime, int
totalTraversed) {
162.         if (path.isEmpty()) {
163.             pathArea.setText("No path found");
164.         } else {
165.             StringBuilder pathText = new StringBuilder();
166.             int count = 1;
167.             for (String word : path) {
168.                 pathText.append(count).append(". ").append(word).append("\n");
169.                 count++;
170.             }
171.             pathText.append("\nElapsed Time: ").append(elapsedTime).append(" ms\n");
172.             pathText.append("Total Nodes Traversed: ").append(totalTraversed);
173.             pathArea.setText(pathText.toString());
174.         }
175.     }
176.
177.     public static void main(String[] args) {
178.         SwingUtilities.invokeLater(() -> {
179.             try {
180.                 new WordLadderGUI();
181.             } catch (IOException e) {
182.                 throw new RuntimeException(e);
```

```
183.             }
184.         });
185.     }
186. }
187.
```

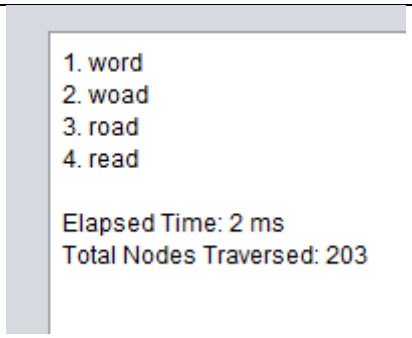
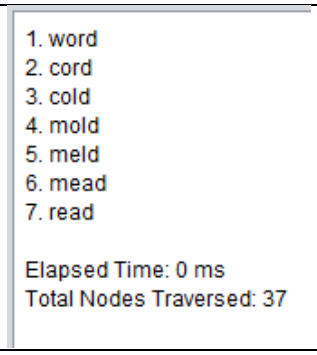
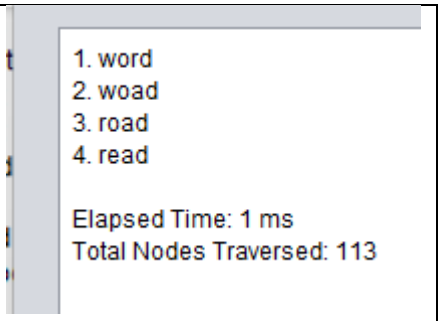
Tangkapan Layar Program

Perbandingan Kasus ketiga algoritma

No	UCS	Greedy BFS	A*
1	1. string 2. sering 3. serins 4. series 5. serges 6. sarges 7. parges 8. parged 9. parted 10. patted 11. pattee 12. pattie 13. cattie 14. cattle 15. battle 16. bottle Elapsed Time: 71 ms Total Nodes Traversed: 7814	1. string 2. sering 3. serins 4. series 5. serves 6. verves 7. vervet 8. verset 9. verser 10. verier 11. varier 12. varies 13. paries 14. parses 15. parser 16. parker 17. parked 18. parted 19. patted 20. batted 21. betted 22. better 23. setter 24. settee 25. settle 26. mettle 27. mottle 28. bottle Elapsed Time: 17 ms Total Nodes Traversed: 479	1. string 2. sering 3. serins 4. series 5. serges 6. sarges 7. parges 8. parged 9. parted 10. patted 11. pattee 12. pattie 13. tattie 14. tattle 15. battle 16. bottle Elapsed Time: 61 ms Total Nodes Traversed: 7167

2.	1. waiting 2. wairing 3. pairing 4. parring 5. parking 6. perking 7. jerking 8. jerkins 9. jerkies 10. jerkier 11. perkier 12. peakier 13. beakier 14. brakier 15. brazier 16. crazier 17. crozier 18. crosier 19. crosser 20. crosses 21. cresses 22. creases 23. ureases 24. uneases 25. uncases 26. uncakes 27. uncaked 28. unbaked Elapsed Time: 138 ms Total Nodes Traversed: 7132	1. waiting 2. baiting 3. bailing 4. bawling 5. yawling 6. yawping 7. yauping 8. jauping 9. jauking 10. jacking 11. yacking 12. yanking 13. tanking 14. tanging 15. tonging 16. ponging 17. ponding 18. podding 19. padding 20. wadding 21. warding 22. warring 23. tarring 24. tarting 96. bloated 97. bleated 98. cleated 99. created 100. creased 101. greased 102. greases 103. ureases 104. uneases 105. uncases 106. uncakes 107. uncaked 108. unbaked Elapsed Time: 15 ms Total Nodes Traversed: 916	1. waiting 2. wairing 3. pairing 4. parring 5. parking 6. perking 7. jerking 8. jerkins 9. jerkies 10. jerkier 11. perkier 12. peakier 13. beakier 14. brakier 15. brazier 16. crazier 17. crozier 18. crosier 19. crosser 20. crosses 21. cresses 22. creases 23. ureases 24. uneases 25. uncases 26. uncakes 27. uncaked 28. unbaked Elapsed Time: 133 ms Total Nodes Traversed: 7036
3.	1. tower 2. toner 3. tuner 4. tunes 5. dunes 6. dunts 7. duits 8. quits 9. quins 10. quint 11. quilt 12. built 13. build Elapsed Time: 52 ms Total Nodes Traversed: 7263	1. tower 2. bower 3. boxer 4. boxed 5. boded 6. bided 7. biked 8. baked 9. based 10. bused 11. buses 12. busks 13. bulks 14. bulls 15. bolls 16. boils 17. noils 18. noily 19. doily 20. drily 21. drill 22. trill 23. thill 24. shill 25. spill 26. spile 27. spite 28. suite 29. quite 30. quire 31. quirt 32. quilt 33. built 34. build Elapsed Time: 13 ms Total Nodes Traversed: 1273	1. tower 2. toner 3. tuner 4. tunes 5. dunes 6. dunts 7. duits 8. quits 9. quins 10. quint 11. quilt 12. built 13. build Elapsed Time: 63 ms Total Nodes Traversed: 7009

4.	1. seven 2. sever 3. siver 4. sizer 5. sizes 6. sines 7. sinhs 8. sighs 9. sight 10. eight Elapsed Time: 36 ms Total Nodes Traversed: 5298	1. seven 2. sever 3. sewer 4. sewed 5. sexed 6. vexed 7. vexes 8. rexes 9. reges 10. rages 11. cages 12. cager 13. eager 14. eater 15. enter 16. ender 17. eider 18. aider 19. aimer 20. dimer 21. diver 22. giver 23. given 24. riven 25. rivet 26. revet 27. reset 28. beset 29. besot 30. begot 31. bigot 32. bight 33. eight Elapsed Time: 9 ms Total Nodes Traversed: 1038	1. seven 2. sever 3. siver 4. sizer 5. sizes 6. sines 7. sinhs 8. sighs 9. sight 10. eight Elapsed Time: 24 ms Total Nodes Traversed: 3485
5.	1. piano 2. pians 3. pions 4. phons 5. phone 6. shone 7. shore 8. chore 9. chord Elapsed Time: 31 ms Total Nodes Traversed: 4450	1. piano 2. pians 3. peans 4. peats 5. peaty 6. peavy 7. leavy 8. leave 9. lease 10. least 11. leapt 12. leaps 13. lears 14. bears 15. boars 16. boors 17. moors 18. mools 19. cools 20. cooly 21. cooky 22. choky 23. choke 24. chore 25. chord Elapsed Time: 2 ms Total Nodes Traversed: 153	1. piano 2. pians 3. pions 4. phons 5. phone 6. shone 7. shore 8. chore 9. chord Elapsed Time: 16 ms Total Nodes Traversed: 2321

6.			

Analisis Perbandingan Solusi

Berdasarkan percobaan, algoritma dengan solusi paling optimal adalah algoritma UCS dan A*, dengan algoritma A* membutuhkan waktu dan traversal nodes yang lebih sedikit dibanding UCS.Algoritma dengan jumlah traversal nodes dan waktu eksekusi paling sedikit adalah algoritma Greedy BFS, namun algoritma Greedy BFS sering tidak optimal dalam mencari jarak terdekat.
Pada algoritma greedy waktu eksekusi yang dibutuhkan paling cepat karena dalam mengekspan simpul, algoritma greedy memprioritaskan simpul yang menurutnya lebih dekat menuju target sehingga algoritma ini tidak memproses seluruh simpul yang ada.

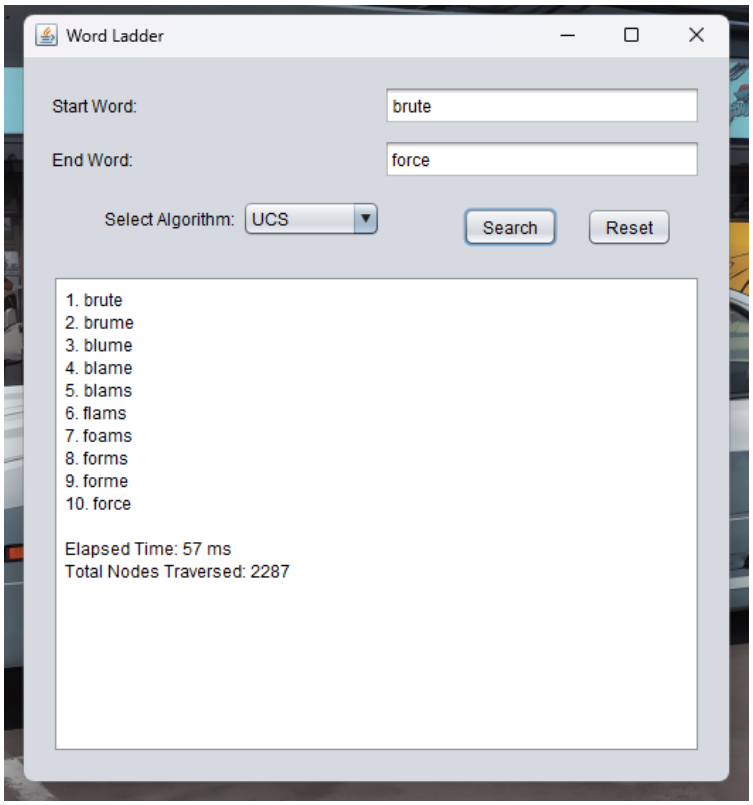
Pada Algoritma A*, solusi yang dihasilkan optimal karena fungsi heuristik yang digunakan admissible seperti yang telah dijelaskan sebelumnya, selain itu algoritma ini juga menghasilkan solusi dengan waktu yang dibutuhkan lebih sedikit jika dibandingkan dengan UCS karena algoritma ini juga mempertimbangka fungsi heuristik yang digunakan oleh algoritma greedy BFS. Algoritma A* juga dapat menghasilkan jumlah node yang di traversed lebih sedikit jika dibandingkan dengan UCS, seperti contoh gambar 5, dimana algoritma A* node traversednya hampir setengah dari algoritma UCS.

Pada algoritma UCS, solusi yang dihasilkan pasti optimal karena pada algoritma ini, cara pencariannya complete, yaitu mencari seluruh simpul secara berurutan sesuai dengan cost dari root sampai simpul saat ini sehingga membutuhkan paling banyak jumlah simpul yang dikunjungi serta memakan paling banyak waktu.

Memori yang dibutuhkan pada algoritma ini berbandiung lurus dengan jumlah node yang dikunjungi, karena setiap node yang dikujungi akan menambahkan simpul-simpul baru pada priority queue, maka algoritma UCS akan membutuhkan lebih banyak memori, kemudian diikuti oleh A star, lalu Greedy BFS.

Implementasi Bonus

Saya mengimplementasikan bonus dengan membuat GUI menggunakan java Swing, GUI menerima input berupa start word, end word dan jenis algoritma serta menampilkan hasilnya. Berikut adalah tampilan dari GUI



Lampiran

Repository github : https://github.com/RayhanFadhlan/Tucil3_13522095