LAPORAN TUGAS KECIL 3

IF2211 - STRATEGI ALGORITMA

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

Dosen:

Ir. Rila Mandala, M.Eng., Ph.D.

Monterico Adrian, S.T., M.T.

Kelompok 74:

Muhammad Zaidan Sa’dun Robbani (13522146)

PROGRAM STUDI TEKNIK INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

SEMESTER II TAHUN 2023/2024

DAFTAR ISI

BAB I

Definisi Algoritma UCS, Greedy Best First Search, dan A*

1. Uniform Cost Search (UCS)

UCS, singkatan dari Uniform Cost Search, adalah metode pencarian graf yang berguna untuk menemukan jalur terpendek antara dua titik dalam sebuah graf yang memiliki bobot pada setiap edge-nya. Prinsip utama yang menjadi landasan bagi UCS adalah mempertimbangkan biaya aktual dari setiap simpul yang sedang dieksplorasi. Saat menjelajahi graf, UCS berupaya untuk memilih jalur dengan biaya total terendah untuk mencapai tujuan yang ditentukan. Dengan kata lain, UCS secara sistematis menelusuri jalur-jalur dengan biaya terendah terlebih dahulu, memungkinkan identifikasi jalur optimal dengan biaya minimum.

2. Greedy Best First Search (GBFS)

GBFS, singkatan dari Greedy Best-First Search, adalah sebuah algoritma pencarian graf yang mengandalkan strategi heuristik untuk menentukan simpul mana yang akan dieksplorasi selanjutnya. Algoritma ini memperhitungkan nilai heuristik dari setiap simpul dan memilih simpul dengan nilai heuristik terendah, tanpa memperhatikan biaya aktual jalur yang telah ditempuh. Dengan pendekatan ini, GBFS cenderung memprioritaskan pencarian jalur yang memiliki estimasi paling dekat dengan tujuan. Meskipun demikian, penting untuk dicatat bahwa GBFS tidak menjamin solusi optimal karena fokus utamanya hanya pada pencarian simpul dengan nilai heuristik terendah.

3. A*

A* (dibaca A Star) merupakan salah satu algoritma penjelajahan graf berbobot dengan informasi. Algoritma ini dapat digunakan untuk menentukan jalur dengan harga terendah dari simpul akar ke simpul target (solusi optimal). A* bekerja dengan cara mempertimbangkan semua jalur yang mungkin dari simpul akar ke simpul target dengan memilih jalur dengan harga terendah pada setiap iterasinya dengan memanfaatkan struktur data PriorityQueue. Algoritma ini akan terus melakukan iterasi selama masih ada jalur dengan harga yang lebih rendah walau simpul target telah ditemukan atau sampai PriorityQueue kosong jika simpul target belum juga ditemukan. Harga yang dimiliki simpul n pada graf menurut algoritma ini adalah f(n) yaitu bobot total dari setiap simpul yang dilewati dari akar sampai ke simpul n yaitu g(n) ditambah dengan estimasi harga dari simpul n ke simpul target yang dihitung menggunakan fungsi heuristik yaitu h(n).

BAB II

Analisis dan implementasi dalam Algoritma UCS, Greedy Best First Search, dan A*

1. Definisi dari f(n) dan g(n)

f(n) Merupakan nilai total dari sebuah simpul dalam ruang pencarian. Nilai ini mencakup biaya aktual yang telah dikeluarkan untuk mencapai simpul tersebut (g(n)), ditambah dengan nilai heuristik yang mengestimasi biaya yang tersisa untuk mencapai tujuan dari simpul tersebut (h(n)). Secara umum, f(n) = g(n) + h(n). Dalam hal ini, f(n) merupakan nilai total yang perlu dievaluasi untuk menentukan urutan simpul yang akan dieksplorasi selanjutnya

g(n) Merupakan biaya aktual atau jarak sejauh ini yang ditempuh untuk mencapai simpul tertentu dalam ruang pencarian dari simpul awal. Ini adalah biaya aktual yang telah dikeluarkan untuk mencapai simpul tersebut. Dalam beberapa konteks, g(n) juga dapat dianggap sebagai jarak terpendek atau biaya minimum yang diperlukan untuk mencapai simpul tersebut.

Dalam implementasi algoritma UCS, nilai g(n) (*cost*) direpresentasikan oleh biaya aktual yang telah dikeluarkan untuk mencapai setiap simpul, sedangkan dalam algoritma A* dan GBFS, f(n) digunakan untuk menentukan urutan simpul yang akan dieksplorasi selanjutnya, dengan menggunakan nilai total biaya dari simpul tersebut ke tujuan akhir.

2. Apakah heuristik yang digunakan pada algoritma A* *admissible*?

Sebuah heuristik dikatakan *admissible* jika estimasi biaya dari simpul ke tujuan tidak melebihi biaya sebenarnya untuk mencapai tujuan dari simpul tersebut. Maka, sebuah heuristik adalah *admissible* jika tidak pernah *overestimate* biaya yang diperlukan untuk mencapai solusi. Dalam algoritma A* penting untuk diperhatikan apakah heuristiknya *admissible* atau tidak karena dapat memengaruhi hasilnya optimal atau tidak. Suatu algoritma A* bisa dikatakan selalu mendapatkan hasil yang optimal jika heuristiknya *admissible*.

Dalam konteks permainan *Word Ladder*, heuristik yang digunakan adalah estimasi biaya terendah untuk mencapai kata tujuan dari setiap kata saat ini dalam pencarian jalur. Dalam algoritma A*, heuristik ini diterapkan sebagai estimasi

terendah dari biaya sisa (h(n)), yaitu jumlah perubahan huruf yang diperlukan untuk mencapai kata tujuan dari kata saat ini.

Sebagai contoh, jika kata saat ini adalah "play" dan kata tujuan adalah "game", maka heuristik akan memberikan estimasi jumlah perubahan huruf yang diperlukan untuk mencapai "play" dari "game". Dalam hal ini, heuristik akan memberikan nilai yang sama atau kurang dari jumlah perubahan huruf yang sebenarnya yang diperlukan untuk mencapai kata tujuan, karena itu memenuhi definisi admissible.

Dengan heuristik yang admissible, algoritma A* dalam permainan Word Ladder dijamin akan menemukan jalur yang optimal, yaitu jalur dengan jumlah perubahan huruf minimum, karena fungsinya akan menghindari simpul yang memiliki estimasi biaya yang lebih tinggi dari biaya sebenarnya. Hal ini sesuai dengan tujuan pencarian jalur terpendek dalam permainan Word Ladder. Maka algoritma A* pada permainan *World Ladder* akan selalu mendapatkan hasil yang optimal / rute terpendeknya.

3. Pada kasus word ladder, apakah algoritma UCS sama dengan BFS?

Algoritma UCS (Uniform Cost Search) dan BFS (Breadth-First Search) memiliki perbedaan dalam menentukan urutan node yang dibangkitkan dan path yang dihasilkan, terutama dalam penanganan biaya atau cost.

Algoritma BFS bekerja dengan cara mengeksplorasi semua simpul pada kedalaman tertentu sebelum melanjutkan ke simpul-simpul yang lebih dalam. Sementara itu, Algoritma UCS bekerja dengan mempertimbangkan biaya atau cost dari setiap jalur yang dibangkitkan. Ini berarti UCS akan lebih memilih untuk mengeksplorasi jalur-jalur yang memiliki biaya lebih rendah terlebih dahulu, sehingga dapat menemukan jalur dengan biaya total minimum.

Dalam permainan *Word Ladder* ini, algoritma UCS akan mempertimbangkan biaya atau cost dari setiap perubahan huruf antar kata, sedangkan BFS hanya akan mempertimbangkan kedalaman pencarian. Maka dari itu, urutan node yang dibangkitkan dan path yang dihasilkan oleh UCS dan BFS kemungkinan akan berbeda, terutama jika ada perbedaan dalam biaya perubahan huruf antar kata.

4. Secara teoritis, apakah algoritma A* lebih efisien dibandingkan dengan algoritma UCS pada kasus word ladder?

Secara teoritis, algoritma A* dapat lebih efisien daripada algoritma UCS dalam kasus permainan Word Ladder. Hal Ini disebabkan oleh penggunaan heuristik yang lebih baik dalam algoritma A* untuk memandu pencarian.

Pada permainan Word Ladder, algoritma A* menggunakan heuristik untuk memberikan perkiraan biaya yang tersisa untuk mencapai kata tujuan dari setiap kata saat ini. Heuristik ini memungkinkan algoritma A* untuk fokus pada penjelajahan jalur-jalur yang memiliki potensi untuk menjadi jalur terpendek, mengarah pada eksplorasi yang lebih efisien.

Di sisi lain, algoritma UCS hanya mempertimbangkan biaya aktual dari setiap jalur yang dibangkitkan tanpa menggunakan informasi tambahan tentang jarak yang tersisa. Ini dapat menyebabkan algoritma UCS untuk mengeksplorasi jalur-jalur yang tidak efisien, terutama jika terdapat banyak kemungkinan jalur dengan biaya yang sama.

Maka secara teoritis jika heuristik yang digunakan dalam algoritma A* adalah *admissible* dan berdasarkan analisis pada no 2 dapat diperkirakan bahwa algoritma A* akan jauh lebih efisien daripada algoritma UCS.

5. Secara teoritis, apakah algoritma Greedy Best First Search menjamin solusi optimal untuk persoalan word ladder?

Secara teoritis, algoritma Greedy Best First Search (GBFS) tidak menjamin solusi optimal untuk persoalan Word Ladder. Hal ini karena GBFS hanya berfokus pada memilih simpul yang memiliki nilai heuristik terendah pada setiap langkahnya tanpa mempertimbangkan total biaya yang dikeluarkan untuk mencapai simpul tersebut.

Dalam kasus *Word Ladder*, GBFS akan memilih simpul yang memiliki perkiraan biaya paling rendah untuk mencapai kata tujuan, tanpa memperhatikan biaya sebenarnya yang telah dikeluarkan untuk mencapai simpul tersebut. Karena GBFS hanya mempertimbangkan nilai heuristik pada setiap langkahnya, hal itu dapat mengakibatkan terperangkapnya dalam jalur yang terlihat optimal berdasarkan heuristik tetapi tidak optimal secara keseluruhan. Oleh karena itu, meskipun GBFS dapat menghasilkan solusi dengan cepat, itu tidak menjamin solusi optimal untuk persoalan *Word Ladder*.

BAB III

Source Code

1. UCS.java

```
import java.util.*;

class NodeUCS {
    String word;
    List<String> path;
    int cost;

    public NodeUCS(String word, int cost) {
        this.word = word;
        this.cost = cost;
        this.path = new ArrayList<>();
    }

    public NodeUCS(String word, int cost, List<String> path) {
        this.word = word;
        this.cost = cost;
        this.path = new ArrayList<>(path);
    }
}

public class UCS {
    private static int nodesVisited = 0;
    static public List<String> algo(String beginWord, String endWord, Map<String,
Boolean> wordList) {
        NodeUCS beginNode = new NodeUCS(beginWord, 0);
        beginNode.path.add(beginWord);

        PriorityQueue<NodeUCS> todo = new PriorityQueue<>((a, b) ->
Integer.compare(a.cost, b.cost));
        todo.add(beginNode);

        while (!todo.isEmpty()) {
            NodeUCS current = todo.poll();
            nodesVisited++;
            if (current.word.equals(endWord)) {
                return current.path;
            }
            wordList.remove(current.word);
            char[] charArray = current.word.toCharArray();
            for (int j = 0; j < charArray.length; j++) {
```

```
            char originalChar = charArray[j];
            for (char c = 'a'; c <= 'z'; c++) {
                charArray[j] = c;
                String newWord = new String(charArray);
                if (wordList.containsKey(newWord)) {
                    List<String> tempPath = new ArrayList<>(current.path);
                    tempPath.add(newWord);
                    int newCost = current.cost + 1; // Biaya untuk UCS adalah 1
                    NodeUCS newNode = new NodeUCS(newWord, newCost,
tempPath);
                    todo.add(newNode);
                    wordList.remove(newWord);
                }
            }
            charArray[j] = originalChar;
        }
    }
    return new ArrayList<>();
  }
  static public int getNodesVisited() {
    return nodesVisited;
  }
}
```

Penjelasan:

Class NodeUCS:

- Ini adalah kelas yang merepresentasikan simpul dalam algoritma UCS.
- Properti-propertinya meliputi:

  word: String yang menyimpan kata pada simpul tersebut.

  path: List<String> yang berisi jalur dari simpul awal ke simpul ini.

  cost: Integer yang menyimpan biaya untuk mencapai simpul ini dari simpul awal.

- Terdapat dua konstruktor:

  Konstruktor pertama menerima kata dan biaya sebagai argumen, digunakan saat membuat simpul awal.

Konstruktor kedua menerima kata, biaya, dan jalur sebagai argumen, digunakan saat membuat simpul baru selama pencarian.

Class UCS:

- Ini adalah kelas yang berisi metode untuk algoritma UCS.
- Terdapat properti statis nodesVisited untuk melacak jumlah simpul yang dikunjungi selama pencarian.
- Method algo adalah method utama untuk algoritma UCS. Ini mengambil kata awal, kata akhir, dan daftar kata-kata yang mungkin sebagai input.

  Dimulai dengan membuat simpul awal dan menambahkannya ke dalam priority queue todo.

  Selama priority queue todo tidak kosong, algoritma mengekstrak simpul dengan biaya terendah dari queue.

  Untuk setiap karakter dalam kata tersebut, algoritma mengganti karakter tersebut dengan setiap huruf dalam alfabet dan memeriksa apakah kata yang baru terbentuk ada dalam daftar kata-kata yang mungkin.

  Jika ada, maka akan dibuat simpul baru dengan kata tersebut dan biaya yang diperbarui, dan ditambahkan ke dalam priority queue.

  Algoritma berlanjut sampai menemukan kata akhir atau hingga semua kemungkinan simpul telah dieksplorasi.

- Method getNodesVisited mengembalikan jumlah simpul yang telah dikunjungi selama pencarian.

2. GBFS.java

```
import java.util.*;

public class GBFS {
    private static int nodesVisited = 0;

    static class Node implements Comparable<Node> {
        String word;
        Node parent;
```

```
        int cost;

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

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

    public static List<String> algo(String startWord, String endWord, Set<String>
validWords) {
        nodesVisited = 0; // Reset the counter before running the algorithm
        PriorityQueue<Node> queue = new PriorityQueue<>();
        Set<String> visited = new HashSet<>();
        queue.add(new Node(startWord, null, 0));
        visited.add(startWord);

        while (!queue.isEmpty()) {
            Node currentNode = queue.poll();
            nodesVisited++; // Increment the counter for each visited node
            String word = currentNode.word;

            if (word.equals(endWord)) {
                List<String> ladder = new ArrayList<>();
                Node node = currentNode;
                while (node != null) {
                    ladder.add(0, node.word);
                    node = node.parent;
                }
                return ladder;
            }

            for (int i = 0; i < word.length(); i++) {
                char[] wordArray = word.toCharArray();
                for (char c = 'a'; c <= 'z'; c++) {
                    wordArray[i] = c;
                    String nextWord = new String(wordArray);

                    if (validWords.contains(nextWord) && visited.add(nextWord)) {
                        int cost = heuristic(nextWord, endWord); // Greedy approach, cost is
heuristic
                        queue.add(new Node(nextWord, currentNode, cost));

                    }
```

```
            }
        }
    }
    return Collections.emptyList();
  }

  public static int getNodesVisited() {
    return nodesVisited;
  }

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

Penjelasan:

Class GBFS:

- Kelas ini merepresentasikan algoritma Greedy Best First Search (GBFS).
- Properti statis nodesVisited digunakan untuk melacak jumlah simpul yang dikunjungi selama pencarian.
- Method algo:

  Ini adalah method utama untuk algoritma GBFS. Ini mengambil kata awal, kata akhir, dan kumpulan kata yang valid sebagai input.

  Dimulai dengan membuat simpul awal dan menambahkannya ke dalam priority queue queue.

  Selama priority queue queue tidak kosong, algoritma mengambil simpul dengan biaya terendah dari queue.

  Untuk setiap karakter dalam kata tersebut, algoritma mengganti karakter tersebut dengan setiap huruf dalam alfabet dan memeriksa apakah kata yang baru terbentuk ada dalam kumpulan kata yang valid.

Jika kata tersebut valid dan belum dikunjungi sebelumnya, maka akan dihitung nilai heuristiknya menggunakan metode heuristic dan ditambahkan ke dalam priority queue.

Algoritma berlanjut sampai menemukan kata akhir atau hingga tidak ada kemungkinan jalur yang tersisa.

- Method getNodesVisited:

  Method ini sederhana dan hanya mengembalikan jumlah simpul yang telah dikunjungi selama pencarian.

  Method heuristic:

  Method ini digunakan untuk menghitung nilai heuristik antara kata saat ini dan kata target.

  Nilai heuristik dihitung sebagai jumlah karakter yang berbeda antara kedua kata tersebut.

Class Node:

- Ini adalah kelas internal yang merepresentasikan simpul dalam algoritma GBFS.
- Properti-propertinya meliputi:

  word: String yang menyimpan kata pada simpul tersebut.

  parent: Node yang merupakan simpul "induk" dari simpul saat ini dalam jalur.

  cost: Integer yang menyimpan biaya atau nilai heuristik dari simpul saat ini.

  Implementasi dari Comparable<Node> digunakan untuk membandingkan simpul berdasarkan biaya.

3. AStar.java

```
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;

class NodeAStar {
    String word;
    List<String> path;
    int cost;
    int heuristic;

    public NodeAStar(String word, int cost, int heuristic, List<String> path) {
        this.word = word;
        this.cost = cost;
        this.heuristic = heuristic;
        this.path = path;
    }
}

public class AStar {
    private static int nodesVisited = 0;

    static public List<String> algo(String beginWord, String endWord, Map<String,
Boolean> wordList) {
        nodesVisited = 0; // Reset the counter before running the algorithm
        NodeAStar beginNode = new NodeAStar(beginWord, 0,
getHeuristic(beginWord, endWord), new ArrayList<>());
        beginNode.path.add(beginWord);

        PriorityQueue<NodeAStar> todo = new PriorityQueue<>(
                (a, b) -> Integer.compare(a.cost + a.heuristic, b.cost + b.heuristic));
        todo.add(beginNode);

        while (!todo.isEmpty()) {
            NodeAStar current = todo.poll();
            nodesVisited++; // Increment the counter for each visited node
            if (current.word.equals(endWord)) {
                return current.path;
            }
            wordList.remove(current.word);
            char[] charArray = current.word.toCharArray();
            for (int j = 0; j < charArray.length; j++) {
                char originalChar = charArray[j];
                for (char c = 'a'; c <= 'z'; c++) {
                    charArray[j] = c;
                    String newWord = new String(charArray);
                    if (wordList.containsKey(newWord)) {
                        List<String> tempPath = new ArrayList<>(current.path);
                        tempPath.add(newWord);
```

```
                    int heuristic = getHeuristic(newWord, endWord);
                    NodeAStar newNode = new NodeAStar(newWord, current.cost + 1,
heuristic, tempPath);
                    todo.add(newNode);
                    wordList.remove(newWord);
                }
            }
            charArray[j] = originalChar;
        }
    }
    return new ArrayList<>();
  }

  static public int getNodesVisited() {
    return nodesVisited;
  }

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

Penjelasan:

Class NodeAStar:

- Ini adalah kelas yang merepresentasikan simpul dalam algoritma A*.
- Properti-propertinya meliputi:

  word: String yang menyimpan kata pada simpul tersebut.

  path: List<String> yang berisi jalur dari simpul awal ke simpul ini.

  cost: Integer yang menyimpan biaya untuk mencapai simpul ini dari simpul awal.

  heuristic: Integer yang menyimpan nilai heuristik dari simpul ini ke simpul tujuan.

- Terdapat satu konstruktor yang digunakan untuk membuat objek NodeAStar dengan memberikan kata awal, biaya, nilai heuristik, dan jalur yang ditempuh.

Class AStar:

- Kelas ini berisi metode untuk algoritma A*.
- Properti statis nodesVisited digunakan untuk melacak jumlah simpul yang dikunjungi selama pencarian.
- Method algo:

  Ini adalah method utama untuk algoritma A*. Ini mengambil kata awal, kata akhir, dan sebuah map yang berisi kata-kata yang mungkin sebagai input.

  Dimulai dengan membuat simpul awal dan menambahkannya ke dalam priority queue todo.

  Selama priority queue todo tidak kosong, algoritma mengambil simpul dengan nilai biaya terendah ditambah dengan nilai heuristiknya dari queue.

  Untuk setiap karakter dalam kata tersebut, algoritma mengganti karakter tersebut dengan setiap huruf dalam alfabet dan memeriksa apakah kata yang baru terbentuk ada dalam map kata-kata yang mungkin.

  Jika kata tersebut valid, maka akan dihitung nilai heuristiknya menggunakan metode getHeuristic, dan ditambahkan ke dalam priority queue.

  Algoritma berlanjut sampai menemukan kata akhir atau hingga tidak ada kemungkinan jalur yang tersisa.

- Method getNodesVisited:

  Method ini mengembalikan jumlah simpul yang telah dikunjungi selama pencarian.

- Method getHeuristic:

  Method ini digunakan untuk menghitung nilai heuristik antara kata saat ini dan kata tujuan.

Nilai heuristik dihitung sebagai jumlah karakter yang berbeda antara kedua kata tersebut.

4. Main.java

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.*;
import java.util.function.Function;
import java.util.stream.Collectors;

public class Main {
    public static void main(String[] args) {
        // Baca dictionary dari file.txt
        Set<String> dictionary = readDictionaryFromFile("dictionary.txt");

        // Convert Set<String> to Map<String, Boolean>
        Map<String, Boolean> wordList = dictionary.stream()
            .collect(Collectors.toMap(Function.identity(), word -> Boolean.TRUE));

        // Input start word, end word, dan pilihan algoritma
        Scanner scanner = new Scanner(System.in);
        System.out.print("Masukkan start word: ");
        String startWord = scanner.nextLine().trim().toLowerCase();
        System.out.print("Masukkan end word: ");
        String endWord = scanner.nextLine().trim().toLowerCase();
        System.out.print("Pilih algoritma:\n1. UCS (Uniform Cost Search)\n2. A* (A
Star Search)\n3. GBFS\nMasukkan nomor algoritma : ");
        String algorithm = scanner.nextLine().trim().toUpperCase();
        scanner.close();

        // Jalankan algoritma sesuai pilihan
        List<String> ladder = new ArrayList<>();
        int nodesVisited = 0;
        long startTime = System.currentTimeMillis();
        if (algorithm.equals("1")) {
            ladder = UCS.algo(startWord, endWord, wordList);
            nodesVisited = UCS.getNodesVisited();
        } else if (algorithm.equals("2")) {
            ladder = AStar.algo(startWord, endWord, wordList);
            nodesVisited = AStar.getNodesVisited();
        } else if(algorithm.equals("3")) {
            Set<String> wordSet = wordList.keySet(); // Convert Map<String,
Boolean> to Set<String>
            ladder = GBFS.algo(startWord, endWord, wordSet);
            nodesVisited = GBFS.getNodesVisited();
```

```
        } else {
            System.out.println("Pilihan algoritma tidak valid.");
            return;
        }
        long endTime = System.currentTimeMillis();

        // Cetak hasil
        if (!ladder.isEmpty()) {
            System.out.println("Path:");
            for (String word : ladder) {
                System.out.println(word);
            }
        } else {
            System.out.println("Tidak ditemukan path yang menghubungkan start word
dan end word.");
        }
        System.out.println("Banyaknya node yang dikunjungi: " + nodesVisited);
        System.out.println("Waktu eksekusi program: " + (endTime - startTime) + "
milidetik");
    }

    private static Set<String> readDictionaryFromFile(String fileName) {
        Set<String> dictionary = new HashSet<>();
        try {
            File file = new File(fileName);
            Scanner scanner = new Scanner(file);
            while (scanner.hasNextLine()) {
                String word = scanner.nextLine().trim().toLowerCase();
                dictionary.add(word);
            }
            scanner.close();
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + fileName);
        }
        return dictionary;
    }
}
```

Penjelasan:

Method main:

- Method ini adalah method utama yang akan dieksekusi ketika program dijalankan.
- Pertama-tama, program membaca kamus kata dari sebuah file teks menggunakan method readDictionaryFromFile.
- Setelah itu, program meminta input dari pengguna berupa kata awal (startWord), kata akhir (endWord), dan pilihan algoritma pencarian.

- Kemudian, program menjalankan algoritma pencarian sesuai dengan pilihan pengguna dan mengukur waktu eksekusi serta jumlah simpul yang dikunjungi.
- Setelah selesai, program mencetak hasil pencarian, termasuk jalur yang ditemukan (jika ada), jumlah simpul yang dikunjungi, dan waktu eksekusi.

Method readDictionaryFromFile:

- Method ini digunakan untuk membaca kamus kata dari sebuah file teks.
- Kamus kata tersebut disimpan dalam sebuah Set<String> dan kemudian dikembalikan.

Variabel dan Objek:

- Terdapat beberapa variabel dan objek yang digunakan dalam method main, seperti dictionary, wordList, scanner, startWord, endWord, algorithm, ladder, nodesVisited, startTime, dan endTime. Variabel-variabel ini digunakan untuk menyimpan informasi yang diperlukan selama proses eksekusi program.

BAB IV

Input dan *Output*

UCS

INPUT 1 **OUTPUT 1**

startWord : frown

endWord : smile

```
Path:
frown
frows
flows
slows
slots
spots
spits
spite
smite
smile
Banyaknya node yang dikunjungi: 4347
Waktu eksekusi program: 39 milidetik
```

INPUT 2 **OUTPUT 2**

startWord : root

endWord : base

```
Path:
root
boot
boos
boss
bass
base
Banyaknya node yang dikunjungi: 2381
Waktu eksekusi program: 29 milidetik
```

INPUT 3 **OUTPUT 3**

startWord : earn

endWord : make

```
Path:
earn
carn
care
cake
make
Banyaknya node yang dikunjungi: 1277
Waktu eksekusi program: 22 milidetik
```

INPUT 4 **OUTPUT 4**

startWord : parka

endWord : scarf

```
Path:
parka
parks
perks
peaks
pears
sears
scars
scarf
Banyaknya node yang dikunjungi: 4477
Waktu eksekusi program: 49 milidetik
```

INPUT 5 **OUTPUT 5**

startWord : story

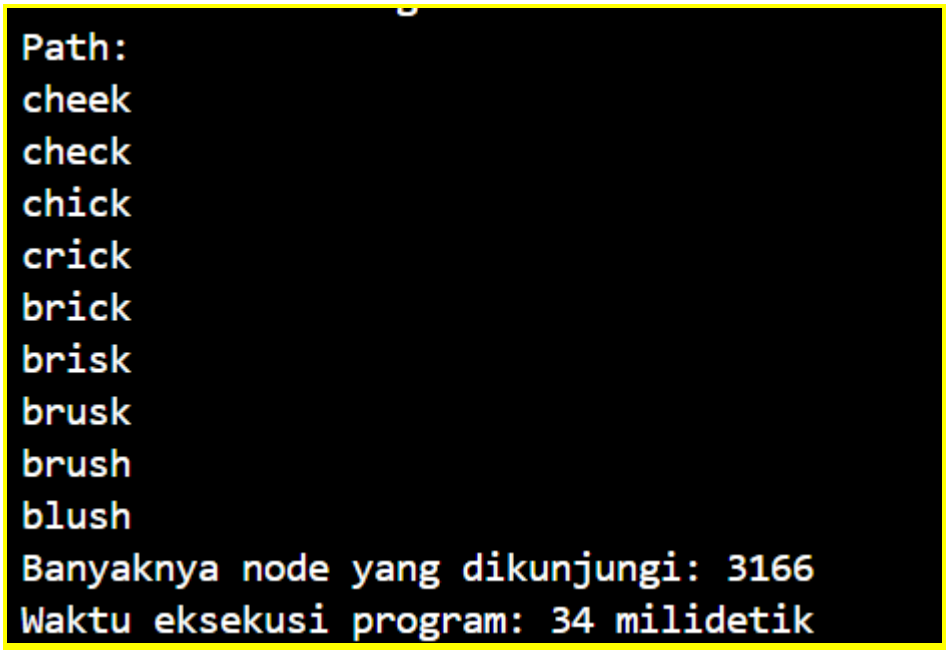
endWord : board

```
Path:
story
stork
stark
stars
soars
boars
board
Banyaknya node yang dikunjungi: 855
Waktu eksekusi program: 20 milidetik
```

INPUT 6 **OUTPUT 6**

startWord : cheek

endWord : blush



GBFS

INPUT 1 OUTPUT 1

startWord : frown

endWord : smile

```
Path:
frown
brown
brows
broos
brios
trios
trips
tripe
trine
thine
shine
spine
spile
smile
Banyaknya node yang dikunjungi: 24
Waktu eksekusi program: 2 milidetik
```

INPUT 2 **OUTPUT 2**

startWord : root

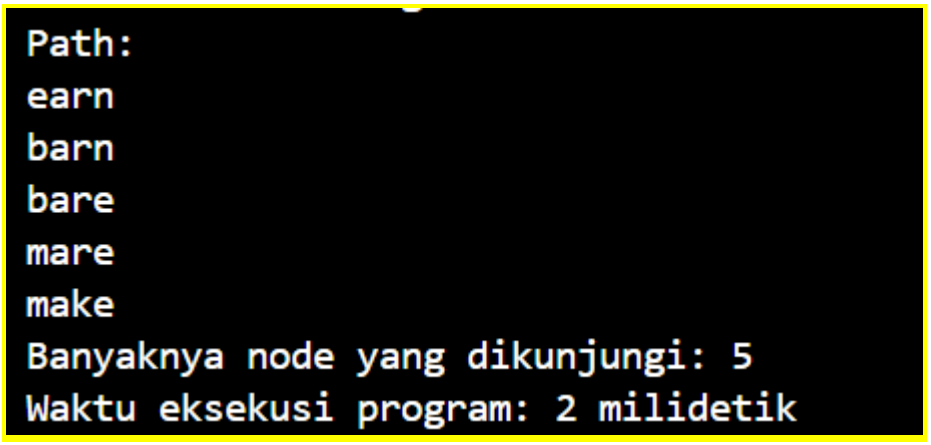
endWord : base

```
Path:
root
boot
bolt
bole
bale
base
Banyaknya node yang dikunjungi: 8
Waktu eksekusi program: 2 milidetik
```

INPUT 3 **OUTPUT 3**

startWord : earn

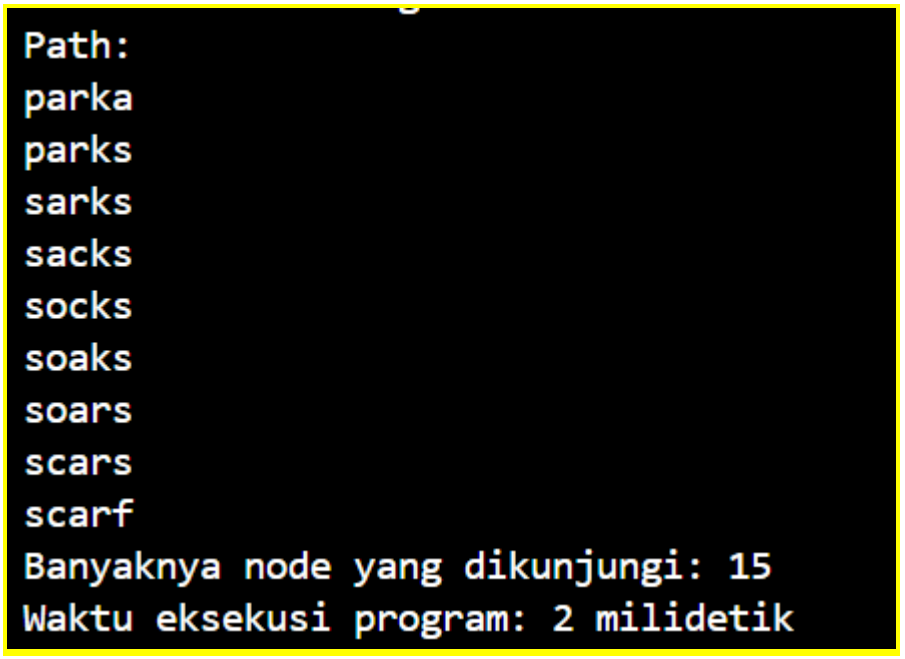
endWord : make



INPUT 4 **OUTPUT 4**

startWord : parka

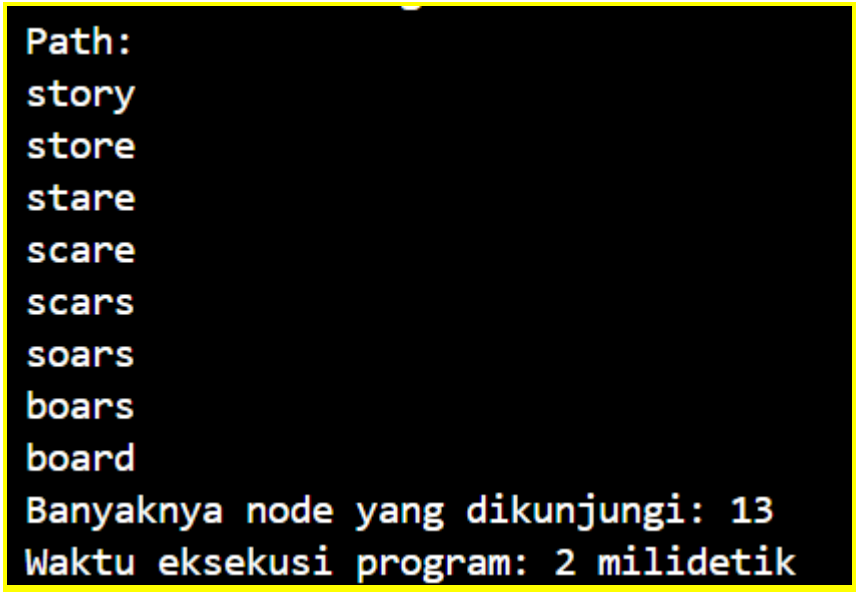
endWord : scarf



INPUT 5 **OUTPUT 5**

startWord : story

endWord : board



INPUT 6 **OUTPUT 6**

startWord : cheek

endWord : blush

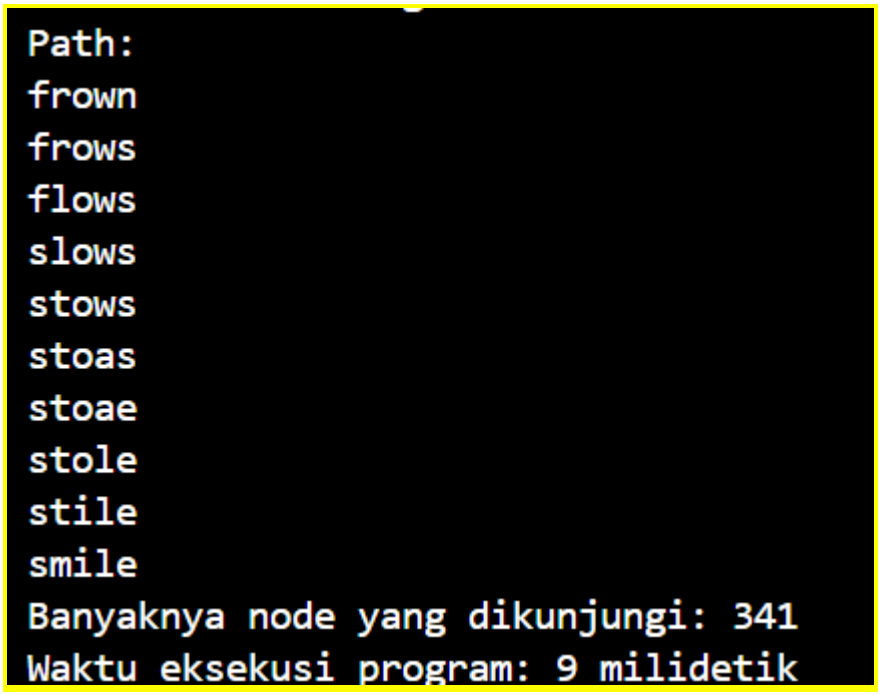
```
Path:
cheek
cleek
gleek
gleed
bleed
blued
bluet
blunt
blent
blest
blast
clast
clash
flash
flush
blush
Banyaknya node yang dikunjungi: 70
Waktu eksekusi program: 4 milidetik
```

AStar

INPUT 1 OUTPUT 1

startWord : frown

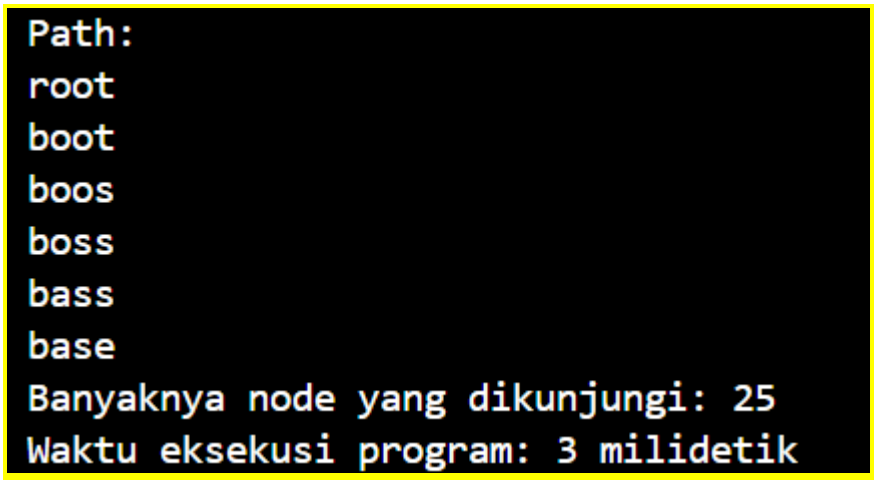
endWord : smile



INPUT 2 **OUTPUT 2**

startWord : root

endWord : base



INPUT 3 **OUTPUT 3**

startWord : earn

endWord : make

```
Path:
earn
carn
care
mare
make
Banyaknya node yang dikunjungi: 16
Waktu eksekusi program: 4 milidetik
```

INPUT 4 OUTPUT 4

startWord : parka

endWord : scarf

```
Path:
parka
parks
perks
peaks
pears
sears
scars
scarf
Banyaknya node yang dikunjungi: 35
Waktu eksekusi program: 3 milidetik
```

INPUT 5 OUTPUT 5

startWord : story

endWord : board

```
Path:
story
store
stare
stars
soars
boars
board
Banyaknya node yang dikunjungi: 32
Waktu eksekusi program: 2 milidetik
```

INPUT 6 OUTPUT 6

startWord : cheek

endWord : blush

```
Path:
cheek
creek
creel
cruel
cruet
crust
crush
brush
blush
Banyaknya node yang dikunjungi: 122
Waktu eksekusi program: 6 milidetik
```

BAB V

Hasil Analisis Perbandingan UCS, Greedy Best First Search, dan A*

Dari eksperimen yang dilakukan sebagaimana yang tercatat dalam BAB V, terlihat bahwa solusi yang dihasilkan oleh algoritma Uniform Cost Search selalu memiliki panjang yang sama dengan solusi dari algoritma A*. Namun, solusi yang dihasilkan oleh algoritma Greedy Best First Search kadang memiliki panjang yang lebih besar. Ini konsisten dengan konsep bahwa algoritma Uniform Cost Search dan A* mampu memberikan solusi optimal, sementara algoritma Greedy Best First Search tidak menjamin solusi optimal.

Dilihat dari jumlah simpul yang dikunjungi, algoritma Greedy Best First Search selalu unggul dengan jumlah paling sedikit, diikuti oleh A*, dan terakhir Uniform Cost Search. Ini juga berlaku untuk penggunaan memori dan waktu eksekusi. Algoritma Greedy Best First Search cenderung memilih simpul berikutnya berdasarkan opsi lokal, tanpa memperhatikan konsekuensi jangka panjangnya, sehingga menjelajahi simpul lebih sedikit. Sebaliknya, Uniform Cost Search dan A* mempertimbangkan seluruh jalur dengan fungsi heuristiknya masing-masing. Di antara keduanya, A* lebih unggul karena memiliki fungsi heuristik yang lebih efisien dalam menemukan solusi optimal dengan cepat.

Dalam konteks permainan Word Ladder, kesimpulan yang dapat diambil adalah algoritma A* adalah pilihan terbaik untuk mendapatkan solusi optimal. Namun, jika kecepatan solusi lebih diutamakan daripada keoptimalan, algoritma Greedy Best First Search merupakan pilihan yang lebih baik.

Pranala Repository Github

Link Github :

zaidanav/Tucil3_13522146 (github.com)

https://github.com/zaidanav/Tucil3_13522146

Link dictionary:

https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/dictionary.txt

Lampiran

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