Tugas Kecil 3 IF2211 Strategi Algoritma

Semester II tahun 2023/2024

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

Julian Chandra Sutadi 13522080

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2024

DAFTAR ISI

BAB I
DASAR TEORI

1. Permainan Word Ladder

Permainan Word Ladder adalah permainan kata yang ditemukan oleh Lewis Carroll pada tahun 1877. Terdapat dua kata yang menjadi kata mulai dan kata akhir. Untuk memenangkan permainan, pemain harus menyusun rantai kata yang dapat menghubungkan dua kata tersebut. Tiap kata yang berdekatan pada rantai tersebut hanya boleh berbeda satu huruf saja pada posisi yang sama. Pada permainan ini diharapkan solusi yang optimal, yaitu solusi dengan panjang rantai kata yang minimal.

2. Algoritma Uniform Cost Search (UCS)

Algoritma Uniform Cost Search (UCS) adalah salah satu algoritma pencarian rute yang sifatnya uninformed, yaitu tidak memiliki informasi tambahan selain keadaan awal dan akhir serta kemungkinan langkah yang dapat diambil.

Algoritma ini mempertimbangkan adanya optimasi yang ingin dilakukan yang bukan hanya berdasarkan pada jumlah langkah yang diambil, sebagaimana layaknya algoritma BFS dan DFS. Algoritma UCS mendasarkan pilihan langkahnya dengan menghitung g(n), yaitu fungsi biaya yang menghasilkan biaya untuk mencapai simpul n dari akar pohon pencarian. Algoritma UCS akan mengekspan simipul dengan g(n) paling kecil dari semua simpul yang pernah dibangkitkan.

3. Algoritma Greedy Best First Search (GBFS)

Algoritma Greedy Best First Search (GBFS) merupakan algoritma yang secara sifat berbeda dengan UCS. GBFS bersifat informed, yaitu memiliki informasi tambahan terkait estimasi biaya yang diperlukan untuk mencapai tujuan dari simpul saat ini. Estimasi biaya ini biasa disebut sebagai *heuristic* dan diberi notasi h(n). Fungsi estimasi biaya atau *heuristic* ini akan dijadikan dasar untuk membangkitkan simpul selanjutnya. Algoritma GBFS akan mengekspan simpul dengan nilai h(n) paling kecil dan mengunjungi simpul dengan nilai h(n) paling kecil.

Permasalahan dari algoritma GBFS yang disebabkan oleh algoritma tidak melakukan *backtracking* (setidaknya di dalam implementasi program ini) adalah kegagalan mencapai solusi optimal karena terjebak pada nilai minimum lokal. Selain itu, karena tidak mengimplementasikan backtracking, algoritma ini juga dapat menyebabkan kegegalan menemukan solusi sama sekali.

4. Algoritma A Star (A*)

Algoritma A Star (A*) ada karena upaya untuk menggabungkan *heuristic* kepada algoritma GBFS. Keberadaan nilai *heuristic* memberi cerminan akan kesulitan penyelesaian masalah dari simpul yang akan dibangkitkan, kualitas solusi yang melibatkan simpul yang dibangkitkan, serta informasi yang dibangkitkan dengan membangkitkan suatu simpul. Fungsi *heuristic* h(n) bergantung pada simpul n, tujuan, sejarah pencarian sampai simpul ke-n, serta domain permasalahn.

Ide dari algoritma A* adalah menghindari jalur yang sudah terlanjur mahal. Hal ini dihindari dengan meminimasi fungsi g(n) sebagaimana dijelaskan pada algoritma UCS. Untuk melibatkan nilai h(n), fungsi evaluasi f(n) pada algoritma A* akan memiliki bentuk f(n) = g(n) + h(n). Maka, algoritma A* dapat dipandang sebagai penggabungan algoritma UCS dan GBFS.

BAB II
ANALISIS DAN DESAIN ALGORITMA

1. Analisis Komponen dan Desain Algoritma

Komponen-komponen permainan Word Ladder akan dipetakan pada algoritma sebagai berikut:

a. Simpul: kata yang valid, yaitu terdapat di dalam kamus https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/dictionary.txt.
b. Sisi: menandakan kemungkinan untuk mencapai suatu simpul dari simpul asal hanya dengan perubahan satu kata di tempat karakter yang sama.

Berdasarkan kedua komponen graf pencarian di atas, akan dirancang algoritma yang secara umum berlaku untuk UCS, GBFS,d an A*. Struktur algoritma adalah sebagai berikut:

a. Inisiasi list yang akan menyimpan simpul yang dibangkitkan (open list) serta simpul yang pernah diekspan (closed list). Open list disusun secara priority queue berdasarkan nilai f(n) atau fungsi evaluasi.
b. Buat simpul baru berdasarkan kata mulai dan masukkan ke open list.
c. Jika simpul saat ini bukan kata akhir dan open list belum kosong, jadikan elemen pertama open list sebagai simpul saat ini dan masukkan ke dalam closed list.
d. Jika simpul saaat ini adalah kata akhir, tandai bahwa kata akhir sudah ditemukan. Lakukan proses i.
e. Jika simpul saat ini bukan kata akhir, maka bangkitkan simpul-simpul anak dari simpul tersebut, yaitu simpul yang dapat dihubungkan oleh sisi yang valid.
f. Jika algoritma adalah algoritma GBFS, maka hapus seluruh isi open list untuk mencegah backtrack.
g. Untuk setiap simpul anak yang dibangkitkan, masukkan ke dalam open list jika kata yang dikandung simpul belum pernah ada di dalam closed list atau open list dengan nilai cost yang lebih kecil atau sama dengan simpul anak yang baru.
h. Kembali ke c.
i. Jika berhasil sampai ke pada kata akhir, rekonstruksi jalur yang sudah ditemukan.

2. Analisis Nilai g(n) pada Algoritma UCS

Pada implementasi program ini, nilai g(n) adalah kedalaman yang telah ditempuh selama pencarian untuk mencapai simpul kata n. Nilai g(n) dihitung setiap kali simpul

dibangkitkan dengan memberikan nilai g(n) = g(parent(n)) + 1. Artinya, besar biaya g(n) akan sama dengan kedalaman, sehingga algoritma UCS yang diimplementasikan pada dasarnya akan serupa dengan algoritma BFS. Urutan pembangkitan node juga akan sama sebab pencarian simpul yang bertetangga dari kamus dilakukan secara sekuensial sesuai dengan abjad.

3. Nilai h(n) pada Algoritma GBFS dan A* serta Admisabilitasnya

Pada implementasi program ini, nilai h(n) adalah banyaknya kata yang berbeda pada posisi yang sama antara suatu simpul dengan simpul yang lain. Misalnya, kata *brow* dan *snow* memiliki nilai h(n) = 2 karena terdapat dua huruf pada posisi yang sama yang berbeda, yaitu huruf b-s dan r-n.

Suatu fungsi heuristik dikatakan *admisible* jika $\forall n, h(n) \leq h^*(n)$, di mana h(n) menandakan nilai fungsi heuristik dan h*(n) menandakan nilai sebenarnya. Kriteria ini biasa disebut sebagai kriteria optimis, di mana nilai *heuristic* tidak pernah pesimis jika dibandingkan dengan nilai sebenarnya, atau selalu menjadi batas bawah.

Pada kasus program ini, syarat ini ini terpenuhi. Hal ini sebab perubahan antara satu kata menuju kata lain yang berbeda pasti memakan langkah sama atau lebih banyak dibandingkan dengan jumlah huruf yang berbeda. Perubahan huruf pada suatu posisi setidaknya membutuhkan satu langkah dan perubahan huruf ini tidak selalu mungkin dilakukan sebab mungkin saja malah akan membangkitkan simpul yang tidak valid, yaitu dengan kata yang tidak tersedia pada kamus.

4. Perbandingan Efisiensi Algoritma UCS dan A*

Parameter efisiensi algoritma UCS dan A* yang akan digunakan adalah banyaknya simpul yang dibangkitkan. Baik algoritma UCS dan A* akan memberikan solusi yang optimal, atau dengan jumlah simpul pada solusi rantai kata yang minimal. Pemberian solusi yang optimal dijamin oleh UCS karena bekerja layaknya BFS yang memberikan solusi optimal. Sedangkan, solusi A* juga pasti akan optimal jika memiliki fungsi h(n) yang *admissible*.

Namun, karena karakteristika A* yang merupakan *informed search*, yaitu memiliki taksiran akan optimalitas jalur yang dipilih dan kompleksitas subpersoalan yang akan diambil ketika mengunjungi suatu simpul, A* akan membangkitkan simpul yang lebih sedikit. Artinya A* akan lebih efisien bila dibandingkan dengan UCS. Selain itu, pada konteks permasalahan ini UCS tidak melakukan optimisasi apapun sebab ia bekerja layaknya BFS.

5. Analisis Optimalitas Algoritma GBFS

Seperti yang dijelaskan pada BAB I, algoritma GBFS mempunyai kerentanan untuk tidak mencapai titik maksimal global sebab terjebak dengan titik lokal masimal. Selain itu, pada algoritma yang diimplementasikan kali ini, algoritma GBFS tidak melakukan *backtracking*. Pada setiap iterasinya, open list akan dikosongkan kembali sehingga hanya berisi simpul-simpul anak yang baru.

Secara nyata, keterjebakan pada nilai minimum lokal disebabkan oleh algoritma yang memilih kata yang dekat dengan kata akhir, tanpa mempertimbangkan bahwa solusi seharusnya melibatkan kata yang tidak terlalu dekat dengan kata akhir namun perlu untuk pergantian huruf pada suatu posisi yang sulit dicapai.

BAB III
SOURCE CODE DAN PENJELASAN KELAS

1. Struktur Program

Secara umum, struktur program adalah sebagai berikut:

```
.
├── README.md
├── bin
│   └── wordladder
│       ├── Main.class
│       ├── Node.class
│       ├── PrioQueue.class
│       ├── ReturnValues.class
│       ├── Solver$AStarEvaluationFunction.class
│       ├── Solver$EvaluationFunction.class
│       ├── Solver$GreedyBFSEvaluationFunction.class
│       ├── Solver$UCSEvaluationFunction.class
│       └── Solver.class
├── doc
│   └── Tucil3_K2_13522080_Julian Chandra Sutadi.docx
├── run.bat
├── run.sh
├── src
│   ├── dictionary.txt
│   └── wordladder
│       ├── Main.java
│       ├── Node.java
│       ├── PrioQueue.java
│       ├── ReturnValues.java
│       └── Solver.java
└── test
```

Gambar 1. Struktur Program

Pada program terdapat suatu folder src yang mencakup kamus serta semua kelas yang digunakan pada implementasi algoritma. Terdapat *package* wordladder yang di dalamnya terdapat lima kelas, yaitu kelas Node, PrioQueue, ReturnValues, Solver, dan Main. Di dalam kelas Solver sendiri terdapat satu interface dan tiga kelas, yaitu interface EvaluationFunction serta kelas UCSEvaluationFunction, GreedyBFSEvaluationFunction, dan AstarEvaluatioFunction.

Secara umum, program akan berjalan dari Main, yang membaca dictionary untuk mendapat list yang mengandung semua kata kemudian menciptakan objek Solver. Di dalam objek solver akan diciptakan objek EvaluationFunction yang sesuai dengan algoritma pilihan user. Kelas-kelas yang lain akan dihidupkan untuk menjadi objek pembantu bagi objek Solver.

2. Kelas Node

Kelas Node menjadi kelas yang berfungsi untuk menampung informasi penting dari suatu simpul, yaitu informasi word, cost, depth, dan parent. Secara khusus, parent akan menyimpan reference pada simpul yang membangkitkan simpul saat ini agar memori yang dibutuhkan lebih efisien karena tidak banyak memiliki duplikat.

Node.java

```
package wordladder;

import java.util.ArrayList;
import java.util.List;

public class Node {
    private String word;
    private int cost;
    private int depth;
    private Node parent;

    Node() {}

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

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

    public String getWord() {
```

```
        return word;
    }

    public int getCost() {
        return cost;
    }

    public int getDepth() {
        return depth;
    }

    public Node getParent() {
        return parent;
    }

    public void setCost(int cost) {
        this.cost = cost;
    }

    public List<Node> findAdjacent(List<String> listOfWords) {
        int length = listOfWords.size();
        List<Node> adjacentList = new ArrayList<>();
        for (int i = 0; i < length; i++) {
            String tempWord = listOfWords.get(i);
            if (isAdjacent(tempWord)) {
                Node newNode = new Node(tempWord, this.depth + 1,
this);
                adjacentList.add(newNode);
            }
        }
        return adjacentList;
    }

    public List<Node> reconstructPath() {
        Node currentNode = this;
        List<Node> path = new ArrayList<>();
        while (currentNode != null) {
            path.add(currentNode);
            currentNode = currentNode.parent;
        }
```

```
        return path;
    }

    private boolean isAdjacent(String word2) {
        // Check if words have the same length
        if (this.word.length() != word2.length()) {
            return false;
        }

        int diffCount = 0;
        for (int i = 0; i < this.word.length(); i++) {
            if (this.word.charAt(i) != word2.charAt(i)) {
                diffCount++;
                if (diffCount > 1) {
                    return false; // More than one difference found
                }
            }
        }
        return diffCount == 1; // Exactly one difference found
    }
}
```

Method	**Penjelasan**
public List<Node> findAdjacent(List<String> listOfWords)	Mencari semua kata yang adjacent atau hanya berbeda huruf pada satu lokasi.
public List<Node> reconstructPath()	Mengkonstruksi rantai kata dengan cara menelusuri kembali parent dari simpul tujuan.
private boolean isAdjacent(String word2)	Mengembalikan benar jika dua kata adjacent.

3. Kelas PrioQueue

Kelas PrioQueue adalah kelas dengan objek yang dibangkitkan dijadikan placeholder bagi elemen-elemen open list. Kelas in memiliki atribut buffer, yang adalah List of Node. Prioritas pengurutan adalah Node dengan cost yang terurut membesar.

PrioQueue.java

```
package wordladder;

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

```
import java.util.Collections;

/**
* Priority Queue to store the node with the cost in ascending order.
*/
class PrioQueue {
    private List<Node> buffer = new ArrayList<>();

    public List<Node> getBuffer() {
        return buffer;
    }

    public void enqueue(Node q) {
        int idx = Collections.binarySearch(buffer, q, (a, b) ->
a.getCost() - b.getCost());

        // if q is not in buffer, idx = (-(insertion point) - 1).
Therefore insertion point = - (idx + 1)
        if (idx < 0) {
            idx = -(idx + 1);
        }

        buffer.add(idx, q);
    }

    public Node dequeue() {
        Node firstNode = buffer.get(0);
        buffer.remove(0);
        return firstNode;
    }
}
```

Method	**Penjelasan**
`public void enqueue(Node q)`	Memasukkan elemen baru dengan posisi node yang sesuai, menjaga terurut membesar
`public Node dequeue()`	Menghapus elemen pertama pada buffer milik instans PrioQueue dan mengembalikan nilainya.

4. Kelas ReturnValues

Kelas ReturnValues yang diinstansiasikan berperan sebagai placeholder bagi nilai

yang dikembalikan oleh kelas Solver, yaitu rantai kata yang ditemukan dan jumlah simpul yang dikunjungi (atau, jumlah simpul yang terdapat di dalam closed list). Oleh karena itu, kelas ini memeliki atribut path dan numVisited dengan kelas List of String dan tipe data integer.

ReturnValues.java

```
package wordladder;

import java.util.List;

public class ReturnValues {
    private List<String> path;
    private int numVisited;

    ReturnValues(List<String> path, int numVisited) {
        this.path = path;
        this.numVisited = numVisited;
    }

    public List<String> getPath() {
        return path;
    }

    public int getNumVisited() {
        return numVisited;
    }
}
```

5. Kelas Solver

Kelas Solver adalah kelas utama yang memiliki implementasi algoritma utama pencarian rute. Kelas ini menyimpan atribut startWord dan endWord. Di dalam kelas ini juga terdapat interface EvaluationFunction serta implementasi interface tersebut di dalam tiga kelas yang berbeda. Berikut adalah implementasi kelas Solver.

Solver.java

```
package wordladder;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
```

```
import java.util.List;
import java.util.stream.Collectors;

class Solver {
    /**
     * interface EvaluationFunction
     */
    public interface EvaluationFunction {
        public abstract int calculateCost(Node node);
    }

    /**
     * implementation of EvaluationFunction for Uniform Cost Search
     */
    class UCSEvaluationFunction implements EvaluationFunction {
        @Override
        public int calculateCost(Node node) {
            return node.getDepth();
        }
    }

    /**
     * implementation of EvaluationFunction for Greedy Best First
Search involving heuristic
     */
    class GreedyBFSEvaluationFunction implements EvaluationFunction {
        @Override
        public int calculateCost(Node node) {
            String word = node.getWord();

            int numberOfDifferentChars = 0;
            for (int i = 0; i < word.length(); i++) {
                if (word.charAt(i) != endWord.charAt(i)) {
                    numberOfDifferentChars++;
                }
            }
            return numberOfDifferentChars;
        }
    }

    /**
```

```
     * implementation of EvaluationFunction for AStar, combining UCS
evaluation function and Greedy BFS heuristic
     */
    class AStarEvaluationFunction implements EvaluationFunction {
        @Override
        public int calculateCost(Node node) {
            UCSEvaluationFunction ucsEvaluationFunction = new
UCSEvaluationFunction();
            GreedyBFSEvaluationFunction greedyBFSEvaluationFunction =
new GreedyBFSEvaluationFunction();

            return ucsEvaluationFunction.calculateCost(node) +
greedyBFSEvaluationFunction.calculateCost(node);
        }
    }

    private String startWord;
    private String endWord;

    public Solver(String startWord, String endWord) {
        this.startWord = startWord;
        this.endWord = endWord;
    }

    /**
     *
     * @param listOfWords list of words from the dictionary used in
current game
     * @param evaluationFunction evaluation function unique to the
opted algorithm
     * @return resulting path from the route planning algorithm
     */
    public ReturnValues findPath(List<String> listOfWords,
EvaluationFunction evaluationFunction) {
        PrioQueue openList = new PrioQueue();
        List<Node> closedList = new ArrayList<>();

        Node startNode = new Node(this.startWord, 0, 0, null);
        openList.enqueue(startNode);

        Node currentNode = new Node();
```

```
        boolean found = false;
        while (!found && !openList.getBuffer().isEmpty()) {
            currentNode = openList.dequeue();
            closedList.add(currentNode);
            if (currentNode.getWord().equals(endWord)) {
                found = true;
            } else {
                List<Node> childNodes = new ArrayList<>();
                childNodes = currentNode.findAdjacent(listOfWords);
                childNodes.forEach(node -> {

node.setCost(evaluationFunction.calculateCost(node));
                });

                /* if a node with the same position as successor is
in the OPEN list which has a lower f than successor, skip this
successor
                   if a node with the same position as successor is
in the CLOSED list which has a lower f than successor, skip this
successor
                   otherwise, add  the node to the open list
                   ref: https://www.geeksforgeeks.org/a-search-
algorithm/ */

                if (evaluationFunction instanceof
GreedyBFSEvaluationFunction) {
                    openList.getBuffer().clear();
                }
                for (Node childNode: childNodes) {
                    if (!isWordInOpenListWithLowerEvalFunc(openList,
childNode) && !isWordInClosedListWithLowerEvalFunc(closedList,
childNode)) {
                        openList.enqueue(childNode);
                    }
                }
            }
        }

        List<String> result = new ArrayList<>();
        int numVisited = 0;
```

```
        if (found) {
            result =
(currentNode.reconstructPath()).stream().map(Node::getWord).collect(C
ollectors.toList());
            numVisited = closedList.size();
        }
        return new ReturnValues(result, numVisited);
    }

    /**
     *
     * @param closedList closed list of the route planning algorithm
     * @param succesor successor of current node being expanded
     * @return true if there exist a node with the same position as
successor is in the CLOSED list which has a lower f than successor
     */
    private static boolean
isWordInClosedListWithLowerEvalFunc(List<Node> closedList, Node
succesor) {
        if (closedList.isEmpty()) {
            return false;
        }
        for (Node element: closedList) {
            if (element.getWord().equals(succesor.getWord())) {
                if (succesor.getCost() >= element.getCost()) {
                    return true;
                }
            }
        }
        return false;
    }

    /**
     *
     * @param openList open list of the route planning algorithm
     * @param succesor successor of current node being expanded
     * @return true if there exist a node with the same position as
successor is in the OPEN list which has a lower f than successor
     */
    private static boolean
isWordInOpenListWithLowerEvalFunc(PrioQueue openList, Node succesor)
```

```
{
        if (openList.getBuffer().isEmpty()) {
            return false;
        }
        List<Node> buffer = openList.getBuffer();
        for (Node element: buffer) {
            if (element.getWord().equals(succesor.getWord())) {
                if (succesor.getCost() >= element.getCost()) {
                    return true;
                }
            }
        }
        return false;
    }

    /**
     *
     * @param filename the file with list of words (dictionary) to be
used in the game
     * @return list of words that will be used in the game
     */
    public static List<String> loadTxtToList(String filename) {
        List<String> words = new ArrayList<>();
        try (BufferedReader reader = new BufferedReader(new
FileReader(filename))) {
            String line;
            while ((line = reader.readLine()) != null) {
                words.add(line.trim());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return words;
    }
}
```

Method	**Penjelasan**
public int calculateCost(Node node);	Menghitung besar fungsi evaluasi f(n) berdasarkan dengan pilihan algoritma user

public ReturnValues findPath(List<String> listOfWords, EvaluationFunction evaluationFunction)	Implementasi algoritma utama pencarian rute. Method ini menerima fungsi evaluasi sebagai parameternya dan behaviournya akan sedikit berbeda jika evaluationFunction adalah instans dari kelas GreedyBFSEvaluationFunction
private static boolean isWordInClosedListWithLowerEvalFunc(List<Node> closedList, Node succesor)	Mengembalikan benar jika suatu simpul sudah terdapat di dalam closed list dengan cost yang lebih kecil atau sama dengan cost simpul baru.
private static boolean isWordInOpenListWithLowerEvalFunc(PrioQueue openList, Node succesor)	Mengembalikan benar jika suatu simpul sudah terdapat di dalam open list dengan cost yang lebih kecil atau sama dengan cost simpul baru.
public static List<String> loadTxtToList(String filename)	Memuat file .txt yang mengandung daftar kata ke dalam List of String.

6. Kelas Main

Kelas Main adalah driver utama dari program yang menjadi entry point berjalannya program. Implementasinya sederhana dan hanya memiliki satu method static Main. Pada program utama dilakukan validasi masukkan serta pemberian ASCII Art untuk menghias program.

Main.java

```
package wordladder;

import java.util.List;
import java.util.Scanner;

public class Main {
```

```
    public static void main(String args[]) {
        System.out.println( // ASCII ART );
            List<String> listOfWords =
Solver.loadTxtToList("../src/dictionary.txt");
        while (true) {
            Scanner scanner = new Scanner(System.in);



            String opted;
            do {
                System.out.println("\nAlgorithm:\n(1) UCS\n(2) Greedy
BFS\n(3) A*");
                System.out.print("Algorithm: ");
                opted = scanner.nextLine();
                if (!opted.equals("1") && !opted.equals("2") &&
!opted.equals("3")) {
                    System.out.println("Enter a valid option!
(1/2/3)");
                }
            } while (!opted.equals("1") && !opted.equals("2") &&
!opted.equals("3"));

            String startWord;
            String endWord;
            boolean validWord = false;
            do {
                System.out.print("\nEnter the starting word: ");
                startWord = scanner.nextLine();
                if (listOfWords.contains(startWord)) {
                    validWord = true;
                } else {
                    System.out.println("\nPlease enter a valid
english word!");
                }
            } while (!validWord);

            validWord = false;
            do {
                System.out.print("Enter the end word: ");
                endWord = scanner.nextLine();
                if (!listOfWords.contains(endWord)) {
```

```
                    System.out.println("\nPlease enter a valid
english word!");
                } else if (endWord.length() != startWord.length()) {
                    System.out.println("\nPlease enter a word with
the same length as the start word!");
                }
                else {
                    validWord = true;
                }
            } while (!validWord);

            Solver solver = new Solver(startWord, endWord);
            Solver.EvaluationFunction evaluationFunction;
            if (opted.equals("1")) {
                evaluationFunction = solver.new
UCSEvaluationFunction();
            } else if (opted.equals("2")) {
                evaluationFunction = solver.new
GreedyBFSEvaluationFunction();
            } else {
                evaluationFunction = solver.new
AStarEvaluationFunction();
            }

            long startTime = System.nanoTime();
            ReturnValues result = solver.findPath(listOfWords,
evaluationFunction);
            long endTime = System.nanoTime();

            List <String> resultPath = result.getPath();
            int numVisited = result.getNumVisited();

            long executionTime = (endTime - startTime) / 1000000;

            int length = resultPath.size();
            if (length != 0) {
                System.out.println("\nPath:\n");
                for (int i = length - 1; i >= 0; i--) {
                    System.out.println(resultPath.get(i));
                }
            } else {
```

```
                System.out.println("\nNo Solution");
            }

            System.out.println("\nExecution time: " + executionTime +
"ms");
            System.out.println("Number of nodes visited: " +
numVisited);

            String cont;
            do {
                System.out.print("\nDo you want to play again? (Y/N)
");
                cont = scanner.nextLine();
                cont = cont.toLowerCase();
                if (cont.equals("n")) {
                    System.out.println( // ASCII ART );
                    scanner.close();
                    System.exit(0);
                }
            } while (!cont.equals("y") && !cont.equals("n"));
        }
    }
}
```

BAB IV

ANALISIS PERBANDINGAN UJI COBA ALGORITMA

1. Uji Coba Algoritma

BRIGHT – PIRATE

```
Algorithm:
(1) UCS
(2) Greedy BFS
(3) A*
Algorithm: 1

Enter the starting word: bright
Enter the end word: pirate

No Solution

Execution time: 18ms
Number of nodes visited: 13
```

```
Algorithm:
(1) UCS
(2) Greedy BFS
(3) A*
Algorithm: 2

Enter the starting word: bright
Enter the end word: pirate

No Solution

Execution time: 4ms
Number of nodes visited: 6
```

```
Algorithm:
(1) UCS
(2) Greedy BFS
(3) A*
Algorithm: 3

Enter the starting word: bright
Enter the end word: pirate

No Solution

Execution time: 7ms
Number of nodes visited: 13
```

HELLO – WORLD

```
Algorithm:
(1) UCS
(2) Greedy BFS
(3) A*
Algorithm: 1

Enter the starting word: HELLO
Enter the end word: WORLD

Path:

hello
hells
heals
weals
weald
woald
world

Execution time: 488ms
Number of nodes visited: 1726
```

```
Algorithm:
(1) UCS
(2) Greedy BFS
(3) A*
Algorithm: 2

Enter the starting word: HELLO
Enter the end word: WORLD

No Solution

Execution time: 9ms
Number of nodes visited: 21
```

```
Algorithm:
(1) UCS
(2) Greedy BFS
(3) A*
Algorithm: 3

Enter the starting word: HELLO
Enter the end word: WORLD

Path:

hello
hells
wells
weals
weald
woald
world

Execution time: 18ms
Number of nodes visited: 55
```

FLOWER – BRAINS

```
Algorithm:
(1) UCS
(2) Greedy BFS
(3) A*
Algorithm: 1

Enter the starting word: FLOWER
Enter the end word: BRAINS

Path:

flower
flowed
flawed
flamed
flames
flanes
flanks
blanks
branks
brands
braids
brains

Execution time: 1807ms
Number of nodes visited: 4958
```

```
Algorithm:
(1) UCS
(2) Greedy BFS
(3) A*
Algorithm: 2

Enter the starting word: FLOWER
Enter the end word: BRAINS

Path:

flower
blower
blowed
browed
brewed
brewer
brawer
brayer
brayed
braved
braves
bravas
brazas
brazes
brakes
braces
bracts
brants
branks
brands
braids
brains

Execution time: 9ms
Number of nodes visited: 22
```

```
Algorithm:
(1) UCS
(2) Greedy BFS
(3) A*
Algorithm: 3

Enter the starting word: FLOWER
Enter the end word: BRAINS

Path:

flower
flowed
flawed
flamed
flames
flanes
flanks
franks
branks
brands
braids
brains

Execution time: 271ms
Number of nodes visited: 795
```

FRUMPY – POLERS

```
Algorithm:
(1) UCS
(2) Greedy BFS
(3) A*
Algorithm: 1

Enter the starting word: frumpy
Enter the end word: polers

Path:

frumpy
frumps
crumps
crimps
chimps
chirps
chirrs
shirrs
shiers
shyers
sayers
sawers
pawers
powers
polers

Execution time: 2709ms
Number of nodes visited: 6337
```

```
Algorithm:
(1) UCS
(2) Greedy BFS
(3) A*
Algorithm: 2

Enter the starting word: FRUMPY
Enter the end word: POLERS

No Solution

Execution time: 7ms
Number of nodes visited: 12
```

```
Enter the starting word: FRUMPY
Enter the end word: POLERS

Path:

frumpy
frumps
flumps
plumps
plumes
pluses
pauses
parses
parges
pargos
pareos
parers
pawers
powers
polers

Execution time: 541ms
Number of nodes visited: 1514
```

SLATE – BOOKS

```
Algorithm:
(1) UCS
(2) Greedy BFS
(3) A*
Algorithm: 1

Enter the starting word: slate
Enter the end word: books

Path:

slate
slats
slots
blots
boots
books

Execution time: 502ms
Number of nodes visited: 1560
```

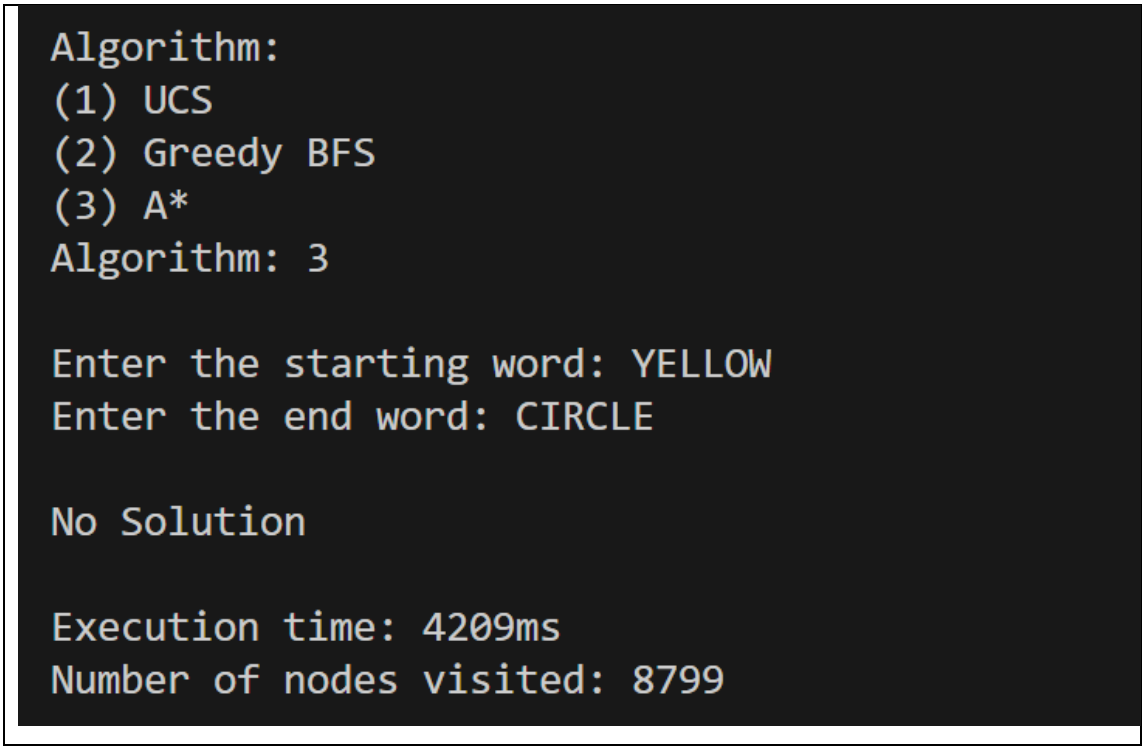
```
Algorithm:
(1) UCS
(2) Greedy BFS
(3) A*
Algorithm: 2

Enter the starting word: slate
Enter the end word: books

Path:

slate
slats
slots
soots
sooks
books

Execution time: 3ms
Number of nodes visited: 6
```

```
Enter the starting word: slate
Enter the end word: books

Path:

slate
slats
blats
boats
boots
books

Execution time: 5ms
Number of nodes visited: 11
```

YELLOW – CIRCLE

```
Algorithm:
(1) UCS
(2) Greedy BFS
(3) A*
Algorithm: 1

Enter the starting word: yellow
Enter the end word: circle

No Solution

Execution time: 3878ms
Number of nodes visited: 8403
```

```
Algorithm:
(1) UCS
(2) Greedy BFS
(3) A*
Algorithm: 2

Enter the starting word: YELLOW
Enter the end word: CIRCLE

No Solution

Execution time: 5ms
Number of nodes visited: 7
```



2. Analisis Hasil Pengujian

Berdasarkan penguijan yang dilakukan, ditemukan bahwa secara umum algoritma GBFS berjalan dengan lebih cepat, baik ketika ditemukan solusi maupun tidak. Pengujian juga selaras dengan teori bahwa hasil dari UCS dan A* pasti akan selalu menemukan optimal bila ada. Namun, dapat dilihat bahwa walaupun ada kasus di mana jumlah simpul yang dikunjungi oleha algoritma A* lebih besar dari algoritma UCS, namun secara umum ketimpangan yang lebih besar terjadi antara UCS dan A* ketika UCS mengunjungi lebih banyak simpul. Hal ini karena pada Word Ladder, UCS yang diimplementasikan akan bekerja seperti BFS yang memiliki kompleksitas memori yang sangat besar.

Artinya, pengujian menunjukkan kebenaran teori bahwa A* dan UCS pasti menghasilkan solusi optimal, sedangkan GBFS tidak. Namun, optimalitas memori UCS tidak sebaik A* dan GBFS.

REFERENSI

N. U.Maulidevi, “Penentuan Rute (Route/Path Planning) Bagian 1: BFS, DFS, UCS, Greedy Best First Search,” *IF2211 Strategi Algoritma*, 2024. [Online]. Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf

N. U.Maulidevi, “Penentuan Rute (Route/Path Planning) Bagian 2: Algoritma A*,” IF2211 *Strategi Algoritma*, 2024. [Online]. Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf

LAMPIRAN

Tabel Hasil

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

Pranala Github

Repository Github dapat diakses melalui pranala berikut: https://github.com/julianchandras/Tucil3_13522080.git