Tugas Kecil 3 IF2211 Strategi Algoritma

Semester II tahun 2022/2023

Pengaplikasian Algoritma UCS, Greedy Best First dan A* dalam Menyelesaikan Persoalan Word Ladder

Oleh:

Samy Muhammad Haikal (13522151)

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2023

DAFTAR ISI

BAB I
DESKRIPSI TUGAS

1.1 Deskripsi Masalah

Word Ladder adalah permainan kata atau teka-teki yang tujuannya adalah mengubah satu kata menjadi kata lain, biasanya memiliki panjang yang sama, dengan mengubah huruf satu per satu. Setiap langkah dalam prosesnya harus merupakan kata yang valid dalam kamus, dan tujuannya adalah melakukan hal ini dalam langkah sesedikit mungkin.

Misalnya, untuk mengubah kata "FOOL" menjadi "SAGE", kita dapat melakukan rangkaian langkah berikut, di mana setiap kata hanya berbeda satu huruf:

- FOOL
- POOL
- POLL
- POLE
- PALE
- SALE
- SAGE

Algoritma UCS, Greedy Best First Search, dan Algoritma A* adalah beberapa algoritma pencarian rute yang bisa digunakan untuk menentukan rute terpendek dari sebuah graf. Pada tugas kali ini penulis diminta untuk membuat implementasi dari ketiga algoritma tersebut pada penyelesaian teka teki word ladder.

1.2 Algoritma Uniform Cost Search

Uniform Cost Search atau UCS merupakan salah satu algoritma shortest path (pencarian jalur terpendek) pada weighted graph (graf berbobot). Algoritma ini mempertimbangkan bobot setiap simpul pada graf dan mencari jalur dengan total bobot yang paling minimum dari titik awal ke titik tujuan.

UCS menggunakan pendekatan pencarian graf terurut dengan mengurutkan simpul berdasarkan biaya saat ini dari jalur yang mengarah ke simpul tersebut. Algoritma ini akan mencatat daftar simpul yang dikunjungi dan daftar simpul yang belum dikunjungi.

Pada setiap iterasi, UCS memilih simpul dengan biaya terendah dari daftar simpul yang belum dikunjungi dan menambahkan simpul tersebut ke daftar simpul yang telah dikunjungi. Kemudian, algoritma mengevaluasi simpul tersebut dan memeriksa apakah simpul tersebut adalah simpul tujuan. Jika ya, algoritma mengembalikan jalur dengan biaya minimum dari simpul awal ke simpul tujuan. Jika tidak, algoritma akan memperluas simpul tersebut dan menambahkan anak-anak dari simpul yang sekarang diperiksa ke daftar simpul yang belum Dikunjungi.

Pada algoritma UCS, kompleksitas waktu yang dimiliki lebih tinggi dibandingkan algoritma Breadth First Search (BFS) meskipun memiliki metode yang sama. Hal ini dikarenakan algoritma ini mempertimbangkan bobot setiap simpul untuk memastikan menemukan jalur terpendek dari titik awal ke titik tujuan.

1.3 Algoritma Greedy Best First Search

Algoritma Greedy Best-First Search (GBFS) adalah salah satu algoritma pencarian yang digunakan untuk menemukan jalur terpendek dari titik awal ke titik akhir dalam ruang pencarian. Algoritma ini menggunakan pendekatan "greedy" atau rakus dengan memilih simpul yang dianggap paling dekat dengan tujuan pada setiap langkahnya. Berikut adalah langkah-langkah penyelesaian masalah dengan algoritma greedy best first search:

1. **Inisialisasi**: Mulai dari simpul awal sebagai simpul saat ini.
2. **Pilih simpul yang akan dieksplorasi**: Pilih simpul yang belum dieksplorasi dan dianggap paling dekat dengan tujuan. Jarak ini dihitung menggunakan fungsi heuristik, yang merupakan perkiraan jarak dari simpul saat ini ke simpul tujuan.
3. **Eksplorasi simpul**: Periksa apakah simpul yang dipilih adalah simpul tujuan. Jika ya, algoritma selesai. Jika tidak, tambahkan simpul tersebut ke daftar simpul yang telah dieksplorasi dan lanjutkan ke langkah berikutnya.
4. **Perbarui simpul saat ini**: Pilih simpul yang paling dekat dengan tujuan dari daftar simpul yang telah dieksplorasi sebagai simpul saat ini.
5. **Ulangi langkah 3 dan 4:** Ulangi langkah-langkah tersebut sampai simpul tujuan ditemukan atau tidak ada lagi simpul yang dapat dieksplorasi.

Algoritma GBFS memiliki keunggulan dalam kecepatan pencarian karena hanya fokus pada simpul-simpul yang dianggap paling dekat dengan tujuan. Namun, kelemahannya adalah kemungkinan terjebak dalam jalan buntu (local optimum) jika simpul yang dipilih tidak mengarahkan ke solusi terbaik secara keseluruhan.

1.4 Algoritma A*

Algoritma A* menggunakan kombinasi dari algoritma UCS untuk pencarian jalur terpendek dan Greedy Best-First Search untuk mengurangi jumlah simpul yang dieksplorasi.

Berikut adalah langkah-langkah algoritma A*:

- **Inisialisasi**: Tentukan titik awal sebagai simpul saat ini. Tetapkan nilai g(n) untuk simpul awal (biaya sejauh ini untuk mencapai simpul saat ini) dan h(n) untuk simpul awal (perkiraan biaya tersisa untuk mencapai simpul tujuan).
- **Pilih simpul yang akan dieksplorasi**: Pilih simpul yang belum dieksplorasi dengan nilai f(n) = g(n) + h(n) terkecil. Nilai f(n) menggabungkan biaya sejauh ini (g(n)) dengan perkiraan biaya tersisa ke simpul tujuan (h(n)).
- **Eksplorasi simpul**: Periksa apakah simpul yang dipilih adalah simpul tujuan. Jika ya, algoritma selesai. Jika tidak, tambahkan simpul tersebut ke daftar simpul yang telah dieksplorasi dan lanjutkan ke langkah berikutnya.
- **Perbarui nilai g(n):** Jika nilai g(n) dari simpul yang dipilih lebih kecil dari nilai g(n) sebelumnya untuk simpul tersebut, perbarui nilai g(n) dan perbarui jalur terbaik yang ditemukan ke simpul tersebut.
- **Perbarui simpul saat ini**: Pilih simpul yang memiliki nilai f(n) terkecil dari daftar simpul yang telah dieksplorasi sebagai simpul saat ini.
- **Ulangi langkah 3-5**: Ulangi langkah-langkah tersebut sampai simpul tujuan ditemukan atau tidak ada lagi simpul yang dapat dieksplorasi.

BAB II

ANALISIS PEMECAHAN MASALAH

3.1 Langkah-langkah pemecahan masalah

Pertama-tama program akan menerima input berupa dua buah string yaitu kata awal dan kata tujuan. Untuk mempersingkat waktu pemrosesan penulis hanya akan memasukkan kata dengan panjang yang sama ke dalam wordSet. Setelah itu program akan menerima masukan yaitu pilihan algoritma, lalu program akan menyelesaikan kasus tersebut sesuai dengan algoritma yang dipilih

3.1.1 UCS

Dalam Algoritma UCS kita memrlukan sebuah fungsi f(n) untuk menentukan cost dari suatu node ke node lain, dalam konteks word ladder f(n) adalah perbedaan karakter dari satu kata dengan kata lainnya. karena pada permainan word ladder kita hanya bisa mengubah satu karakter pada setiap langkah maka bisa disimpulkan bahwa cost dari semua node ke node tetangganya bernilai 1.

Pada algoritma ini, akan digunakan tipe data priority queue, untuk menyimpan rute-rute yang akan diperiksa. Pada priority queue ini, yang akan menjadi prioritas adalah total cost dari rute tersebut, semakin rendah cost nya, maka urutannya akan semakin di depan untuk diperiksa. Priority queue akan diinisialisasi dengan simpul awal dengan cost 0. Lalu, akan dilakukan loop dengan kondisi selama priority queue tersebut tidak Kosong. Dalam setiap loop tersebut, pertama akan dilakukan dequeue dari priority queue tersebut untuk mendapatkan elemen pertama. Kedua, akan dilakukan pengecekan apakah elemen terakhir dari rute tersebut merupakan simpul tujuan. Jika ya, maka akan dikembalikan rute yang mengandung simpul tujuan tersebut.

Jika tidak mendapat simpul tujuan pada elemen terakhir rute yang sedang diperiksa, maka akan dilakukan penambahan rute, dengan menambahkan semua simpul tetangga dari simpul terakhir pada rute yang sedang dicek. Simpul tetangga yang ditambahkan juga dipastikan untuk tidak terdapat pada rute yang sedang dilalui. Loop tersebut akan dilakukan selama queue tidak kosong, jika queue kosong maka tidak ditemukan rute dari simpul awal ke tujuan.

Karena cost dari semua node ke node tetangganya bernilai sama maka maka UCS pada dasarnya akan menjadi serupa dengan BFS dalam konteks pencarian jalur dalam word ladder.

3.2.2 Greedy Best First Search

Dalam Algoritma Greedy Best First Search kita memrlukan sebuah fungsi evaluasi g(n) untuk menentukan cost dari suatu node ke node tujuan, dalam konteks word ladder g(n) adalah perbedaan karakter dari kata saat ini dengan kata tujuan.

Pada Algoritma ini tidak diperlukan priority queue untuk menyimpan simpul, karena implementasinya sendiri tidak diperbolehkan melakukan backtracking. Pada setiap

langkah akan dicek apakah ada simpul tetangga yang jarak ke tujuannya lebih kecil dari jarak ke tujuan saat ini. Langkah ini diulang hingga mencapai simpul tujuan atau tidak ada lagi kata yang dapat dieksplorasi.

Algoritma GBFS cenderung menemukan jalur yang lebih singkat daripada algoritma BFS dalam word ladder karena algoritma ini menggunakan heuristik yang mengarahkan pencarian ke arah yang benar secara umum. Namun, algoritma ini tidak menjamin menemukan jalur terpendek karena sifatnya yang "rakus" (greedy) dalam memilih langkah berikutnya. Kekurangan lainnya dari algoritma ini adalah tidak komplit karena simpul yang dievaluasi hanya simpul dengan lokal minimum cost saja, bisa saja ada solusi yang tidak melewati lokal minimum cost yang tidak akan ditemukan ketika menggunakan algoritma ini.

3.2.2 A*

Algoritma A* bisa dibilang adalah bentuk penggabungan dari algoritma UCS dan GBFS, Algoritma ini menggunakan fungsi evaluasi f(n) = g(n) + h(n) dengan g(n) adalah fungsi cost saat ini dan h(n) adalah jarak dari simpul saat ini ke simpul tujuan.

Heuristik yang digunakan adalah perbedaan karakter kata saat ini dengan kata tujuan, dengan menghitung jumlah karakter yang berbeda antara kata saat ini dan kata tujuan, kita memberikan perkiraan dari jumlah langkah yang diperlukan untuk mencapai kata tujuan. Jika kita mengevaluasi setiap langkah dengan mengganti satu karakter, jumlah karakter yang berbeda antara kata saat ini dan kata tujuan memberikan batas atas dari jumlah langkah yang diperlukan. Dengan demikian, heuristik ini memenuhi kriteria admissible dan dapat digunakan secara efektif dalam algoritma A* untuk word ladder.

Algoritma A* memiliki keunggulan dalam menemukan jalur terpendek dengan memperkirakan biaya sisa ke simpul tujuan (heuristik), sehingga dapat menyesuaikan rute pencarian berdasarkan informasi ini. Algoritma A* lebih efisien dibandingkan algoritma UCS karena pada A* kita akan mengembangkan simpul yang memiliki jarak ke tujuan lebih kecil dahulu sebelum mengembangkan simpul simpul lainnya.

BAB III

IMPLEMENTASI DAN PENGUJIAN

4.1 Implementasi program

Implementasi program terdiri dari Main.java, UniformCostSearch.java, GreedyBestFirst.java, A_Star.java, WordNode.java, dan WordSet.java.

Repository program:
https://github.com/rendangmunir/Tucil3_13522151

a. Main.java

File ini merupakan driver utama dari program ini, sehingga tidak terdapat fungsi di dalamnya, hanya berisi menu utama dari program ini serta deklarasi variabel yang akan digunakan.

Source Code:

```
package src;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import java.util.Set;
public class Main {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    System.out.print("Start Word: ");
    String startWord = scanner.nextLine();
    System.out.print("End Word: ");
    String endWord = scanner.nextLine();
    Set<String> wordSet =
WordSet.readWordSetFromFile("../data/dictionary.txt",
startWord.length());
    if(startWord.length()!=endWord.length()){

    }
    System.out.println("Pilih Algoritma!");
    System.out.println("1. UCS");
    System.out.println("2. Greedy Best First");
    System.out.println("3. A*");
    System.out.print("Pilihan(1/2/3): ");
```

```
    int Pilihan = scanner.nextInt();
    long startTime = System.currentTimeMillis();
    List<String> shortestPath = new ArrayList<>();
    if(Pilihan==1){
      shortestPath = UniformCostSearch.findShortestPath(startWord,
endWord, wordSet);
    }else if (Pilihan==2) {
      shortestPath = GreedyBestFirst.findShortestPath(startWord,
endWord, wordSet);
    }else{
      shortestPath = A_Star.findShortestPath(startWord, endWord,
wordSet);
    }
    long endTime = System.currentTimeMillis();
    long duration = endTime-startTime;
    System.out.println("Processing time\t: " + duration + "
miliseconds");

    if (shortestPath != null) {
        System.out.println("Path length\t: " + shortestPath.size());
        System.out.println("Shortest Path from " + startWord + " to " +
endWord + ": " + shortestPath);
    } else {
        System.out.println("No path found from " + startWord + " to " +
endWord);
    }
    scanner.close();

  }
}
```

b. WordSet.java

File ini berisi fungsi untuk membaca word set dari kamus yang sudah disiapkan

Method	Deskripsi
`public static Set<String>`	Fungsi untuk membuat set dari kata yang ada pada kamus, fungsi ini memfilter kata dengan panjang

`readWordSetFromFile(String filename, int length)`	tertentu karena panjang kata pada permainan word ladder selalu sama pada setiap langkah

Source Code:

```
package src;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.HashSet;
import java.util.Scanner;
import java.util.Set;

public class WordSet {

    public static Set<String> readWordSetFromFile(String filename, int
length) {
        Set<String> wordSet = new HashSet<>();
        try {
            Scanner scanner = new Scanner(new File(filename));
            while (scanner.hasNextLine()) {
                String word = scanner.nextLine().trim();
                if(word.length()==length){
                    wordSet.add(word);
                }
            }
            scanner.close();
        } catch (FileNotFoundException e) {
            System.err.println("File not found: " + filename);
            e.printStackTrace();
        }
        return wordSet;
    }
}
```

c. WordNode.java

File ini berisi Implementasi dari kelas simpul.

Method/Attributes	Deskripsi

`String word;`	Kata saat ini
`int cost;`	Cost yang diperlukan untuk mencapai kata saat ini

Source Code:

```
package src;
public class WordNode {
  String word;
  int cost;

  WordNode(String word, int cost) {
      this.word = word;
      this.cost = cost;
  }
}
```

d. UniformCostSearch.java

File ini berisi Implementasi dari algoritma UCS.

Method/Attributes	Deskripsi
`public static List<String> findShortestPath(String start, String goal, Set<String> wordSet)`	Fungsi utama untuk mencari path terpendek dari start ke goal dengan UCS
`private static List<String> getNeighbors(String word, Set<String> wordSet)`	Fungsi untuk mendapatkan list dari simpul tetangga
`private static List<String> reconstructPath(Map<String, String> cameFrom, String current)`	Fungsi untuk mengkonstruksi path setelah path sudah sampai ke tujuan

Source Code:

```
package src;
import java.util.*;
```

```java
public class UniformCostSearch {

    public static List<String> findShortestPath(String start, String
goal, Set<String> wordSet) {
        if(start.length()!=goal.length() ||  !wordSet.contains(start) ||
!wordSet.contains(goal)){
            System.out.println("Node visited\t: 0");
            return null;
        }
        PriorityQueue<WordNode> frontier = new
PriorityQueue<>(Comparator.comparingInt(node -> node.cost));
        Map<String, String> cameFrom = new HashMap<>();
        Map<String, Integer> costSoFar = new HashMap<>();

        frontier.add(new WordNode(start, 0));
        cameFrom.put(start, null);
        costSoFar.put(start, 0);
        Integer nodeCount = 0;
        while (!frontier.isEmpty()) {
            WordNode current = frontier.poll();

            if (current.word.equals(goal)) {
                System.out.println("Node visited\t: " + nodeCount);
                return reconstructPath(cameFrom, current.word);
            }

            for (String next : getNeighbors(current.word, wordSet)) {
                int newCost = costSoFar.get(current.word) + 1;
                if (!costSoFar.containsKey(next) || newCost <
costSoFar.get(next)) {
                    costSoFar.put(next, newCost);
                    frontier.add(new WordNode(next, newCost));
                    cameFrom.put(next, current.word);
                }
                nodeCount++;
            }
        }

        return null; // No path found
```

```
    }

    private static List<String> getNeighbors(String word, Set<String>
wordSet) {
        List<String> neighbors = new ArrayList<>();
        char[] chars = word.toCharArray();
        for (int i = 0; i < chars.length; i++) {
            char originalChar = chars[i];
            for (char c = 'z'; c >= 'a'; c--) {
                if (c != originalChar) {
                    chars[i] = c;
                    String neighbor = new String(chars);
                    if (wordSet.contains(neighbor)) {
                        neighbors.add(neighbor);
                    }
                }
            }
            chars[i] = originalChar;
        }
        return neighbors;
    }

    private static List<String> reconstructPath(Map<String, String>
cameFrom, String current) {
        List<String> path = new ArrayList<>();
        while (current != null) {
            path.add(0, current);
            current = cameFrom.get(current);
        }
        return path;
    }
}
```

e. GreedyBestFirst.java

File ini berisi Implementasi dari algoritma greedy best first search..

Method/Attributes	Deskripsi
`public static List<String> findShortestPath(String start,`	Fungsi utama untuk mencari path terpendek dari start ke goal dengan GBFS

`String goal, Set<String> wordSet)`	
`private static String getNeighbors(String word, String goal, List<String> visited, Set<String> wordSet)`	Fungsi untuk mendapatkan simpul tetangga dengan jarak terdekat ke tujuan
`private static int getDiff(String word, String goal)`	Fungsi untuk mencari perbedaan karakter dari kata saat ini ke tujuan

Source Code:

```
package src;
import java.util.ArrayList;
import java.util.List;
import java.util.Set;

public class GreedyBestFirst {
  public static List<String> findShortestPath(String start, String goal,
Set<String> wordSet) {
    if(start.length()!=goal.length() || !wordSet.contains(start) ||
!wordSet.contains(goal)){
      System.out.println("Node visited\t: 0");
      return null;
    }
    List<String> visited = new ArrayList<>();
    visited.add(start);
    String next = new String();
    int nodeCount=0;
    next = getNeighbors(start, goal, visited, wordSet);
    if(next.isBlank() || visited.contains(next)){
      System.out.println("Node visited\t: "+ nodeCount);
      return null;
    }
    visited.add(next);
    nodeCount++;

    while (!next.equals(goal)) {
```

```
      getNeighbors(next, goal, visited, wordSet);
      next = getNeighbors(next, goal, visited, wordSet);
      if(visited.contains(next)){
        System.out.println("Node visited: \t" + nodeCount);
        return null;
      }
      visited.add(next);
      nodeCount++;
    }
    System.out.println("Node visited\t: " + nodeCount);
    return visited;

  }

  private static String getNeighbors(String word, String goal,
List<String> visited, Set<String> wordSet) {
    String next;
    next = word;
    Integer diff = getDiff(word, goal);
    char[] chars = word.toCharArray();
    for (int i = 0; i < chars.length; i++) {
        char originalChar = chars[i];
        for (char c = 'z'; c >= 'a'; c--) {
            if (c != originalChar) {
                chars[i] = c;
                String neighbor = new String(chars);

                if (wordSet.contains(neighbor) &&
!visited.contains(neighbor) && getDiff(neighbor, goal)<= diff) {
                    next = neighbor;
                    diff = getDiff(neighbor, goal);
                }
            }
        }
        chars[i] = originalChar;
    }
    return next;
  }
```

```
  private static int getDiff(String word, String goal){
    char[] startWord = word.toCharArray();
    char[] goalWord = goal.toCharArray();
    int diff = 0;
    for(int i=0; i<startWord.length; i++){
      if(startWord[i]!=goalWord[i]){
        diff++;
      }
    }
    return diff;
  }

}
```

f. A_Star.java

File ini berisi Implementasi dari Algoritma A*.

Method/Attributes	Deskripsi
`public static List<String> findShortestPath(String start, String goal, Set<String> wordSet)`	Fungsi utama untuk mencari path terpendek dari start ke goal dengan A*
`private static List<String> reconstructPath(Map<String, String> cameFrom, String current)`	Fungsi untuk mengkonstruksi path setelah path sudah sampai ke tujuan
`private static List<String> getNeighbors(String word, Set<String> wordSet)`	Fungsi untuk mendapatkan list dari simpul tetangga
`private static int getDiff(String word, String goal)`	Fungsi untuk mencari perbedaan karakter dari kata saat ini ke tujuan

Source Code:

```
package src;
import java.util.ArrayList;
```

```java
import java.util.Comparator;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Set;

public class A_Star {
  public static List<String> findShortestPath(String start, String goal,
Set<String> wordSet) {
    if(start.length()!=goal.length() || !wordSet.contains(start) ||
!wordSet.contains(goal)){
      System.out.println("Node visited\t: 0");
      return null;
    }
    PriorityQueue<WordNode> frontier = new
PriorityQueue<>(Comparator.comparingInt(node -> node.cost));
    Map<String, String> cameFrom = new HashMap<>();
    Map<String, Integer> costSoFar = new HashMap<>();

    frontier.add(new WordNode(start, 0));
    cameFrom.put(start, null);
    costSoFar.put(start, 0);
    Integer nodeCount = 0;
    while (!frontier.isEmpty()) {
        WordNode current = frontier.poll();

        if (current.word.equals(goal)) {
            System.out.println("Node visited\t: " + nodeCount);
            return reconstructPath(cameFrom, current.word);
        }

        for (String next : getNeighbors(current.word, wordSet)) {
            int newCost = costSoFar.get(current.word) + 1 +
getDiff(current.word, goal);
            if (!costSoFar.containsKey(next) || newCost <
costSoFar.get(next)) {
              costSoFar.put(next, newCost);
              frontier.add(new WordNode(next, newCost));
```

```
                cameFrom.put(next, current.word);
            }
            nodeCount++;
        }
    }

    return null; // No path found
  }

    private static List<String> reconstructPath(Map<String, String>
cameFrom, String current) {
        List<String> path = new ArrayList<>();
        while (current != null) {
            path.add(0, current);
            current = cameFrom.get(current);
        }
        return path;
    }

    private static List<String> getNeighbors(String word, Set<String>
wordSet) {
        List<String> neighbors = new ArrayList<>();
        char[] chars = word.toCharArray();
        for (int i = 0; i < chars.length; i++) {
            char originalChar = chars[i];
            for (char c = 'z'; c >= 'a'; c--) {
                if (c != originalChar) {
                    chars[i] = c;
                    String neighbor = new String(chars);
                    if (wordSet.contains(neighbor)) {
                        neighbors.add(neighbor);
                    }
                }
            }
            chars[i] = originalChar;
        }
        return neighbors;
    }
```

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

}
```

4.2 Tata cara penggunaan program

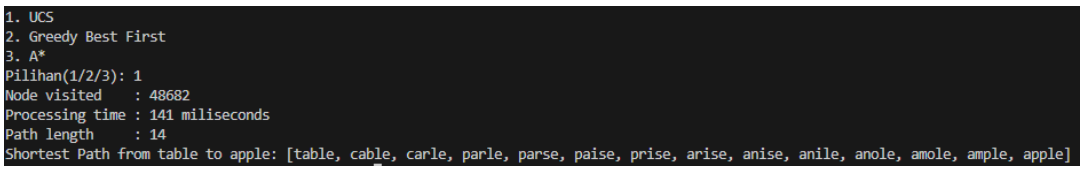
Tata cara menjalankan program:

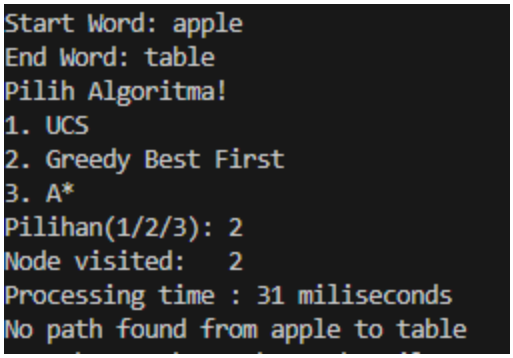
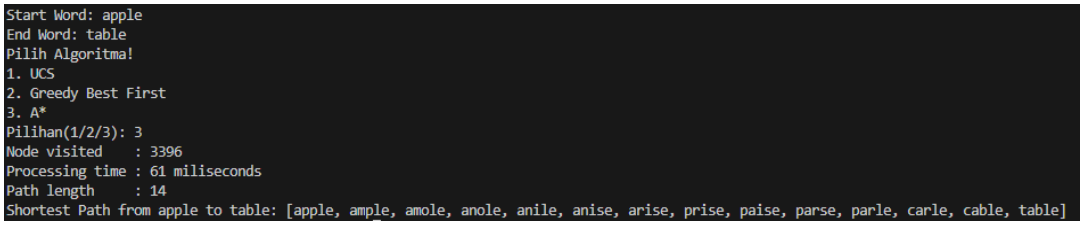
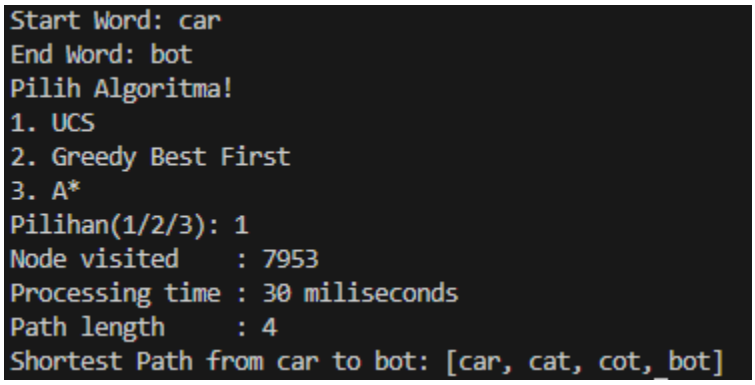
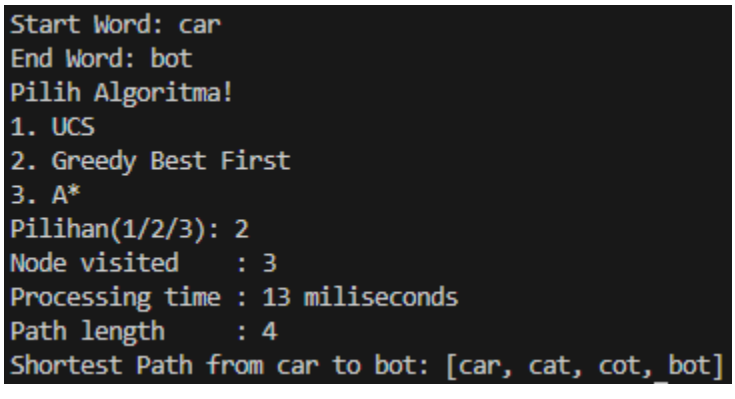
1. Masuk ke folder src
2. Compile “javac -d ..\bin Main.java A_Star.java UniformCostSearch.java GreedyBestFirst.java WordNode.java WordSet.java”
3. Run : “java -cp ../bin src/Main”

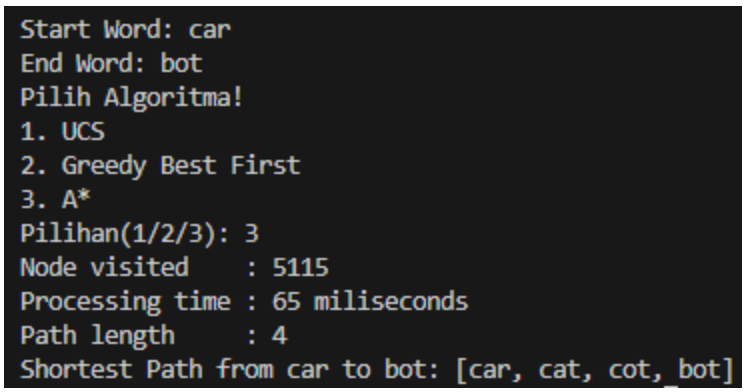
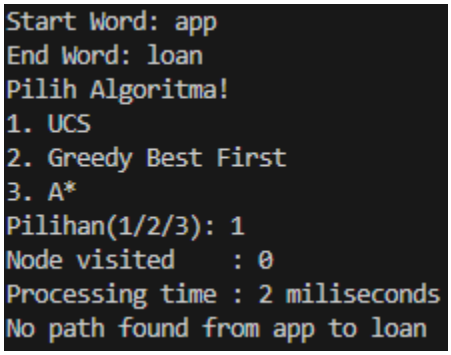
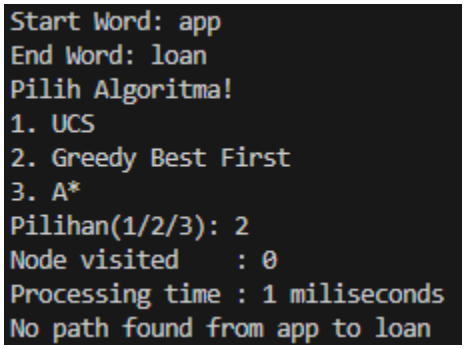
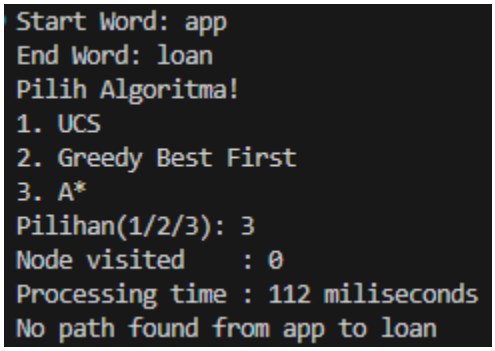
Tata cara menggunakan :

1. Masukkan kata awal
2. Masukkan kata tujuan
3. Pilih algoritma UCS, GBFS atau A*

4.3 Hasil Pengujian

Test Case 1	
UCS	

GBFS	
A*	
Test Case 2	
UCS	
GBFS	

A*	
Test Case 3	
UCS	
GBFS	
A*	
Test Case 4	

UCS	Start Word: earn End Word: make Pilih Algoritma! 1. UCS 2. Greedy Best First 3. A* Pilihan(1/2/3): 1 Node visited : 19970 Processing time : 266 miliseconds Path length : 5 Shortest Path from earn to make: [earn, tarn, tare, take, make]
GBFS	Start Word: earn End Word: make Pilih Algoritma! 1. UCS 2. Greedy Best First 3. A* Pilihan(1/2/3): 2 Node visited : 4 Processing time : 8 miliseconds Path length : 5 Shortest Path from earn to make: [earn, earl, marl, mare, make]
A*	Start Word: earn End Word: make Pilih Algoritma! 1. UCS 2. Greedy Best First 3. A* Pilihan(1/2/3): 3 Node visited : 7448 Processing time : 76 miliseconds Path length : 5 Shortest Path from earn to make: [earn, tarn, tare, take, make]
Test Case 5	

UCS	Start Word: sock End Word: goat Pilih Algoritma! 1. UCS 2. Greedy Best First 3. A* Pilihan(1/2/3): 1 Node visited : 29104 Processing time : 93 miliseconds Path length : 6 Shortest Path from sock to goat: [sock, sook, gook, good, goad, goat]
GBFS	Start Word: sock End Word: goat Pilih Algoritma! 1. UCS 2. Greedy Best First 3. A* Pilihan(1/2/3): 2 Node visited : 6 Processing time : 38 miliseconds Path length : 7 Shortest Path from sock to goat: [sock, soak, soap, soar, boar, boat, goat]
A*	Start Word: sock End Word: goat Pilih Algoritma! 1. UCS 2. Greedy Best First 3. A* Pilihan(1/2/3): 3 Node visited : 8347 Processing time : 41 miliseconds Path length : 6 Shortest Path from sock to goat: [sock, soak, soar, boar, boat, goat]
Test Case 6	
UCS	Start Word: battle End Word: couple Pilih Algoritma! 1. UCS 2. Greedy Best First 3. A* Pilihan(1/2/3): 1 Processing time : 124 miliseconds No path found from battle to couple

GBFS	Start Word: battle End Word: couple Pilih Algoritma! 1. UCS 2. Greedy Best First 3. A* Pilihan(1/2/3): 2 Node visited: 5 Processing time : 13 miliseconds No path found from battle to couple
A*	Start Word: battle End Word: couple Pilih Algoritma! 1. UCS 2. Greedy Best First 3. A* Pilihan(1/2/3): 3 Processing time : 146 miliseconds No path found from battle to couple

4.3 Analisis Algoritma

Metode pencarian UCS menghasilkan hasil yang lengkap namun kurang efisien baik dalam waktu maupun memori karena pengecekan simpul tidak mempertimbangkan heuristik. Metode pencarian GBFS adalah metode pencarian paling cepat dan hemat memori, tetapi tidak semua solusi bisa ditemukan dengan pencarian ini. Sementara itu pencarian A* lebih efisien dibandingkan dengan UCS dan menghasilkan hasil yang lengkap pula.

BAB IV

Kesimpulan dan Saran

Kesimpulan

Dari hasil test case dapat disimpulkan bahwa ketiga algoritma memiliki kelebihan dan kekurangan masing-masing. UCS menghasilkan solusi lengkap tetapi mengorbankan efisiensi, GBFS efisien tetapi tidak menghasilkan solusi yang lengkap, A* adalah penggabungan keduanya yang menghasilkan solusi lengkap tetapi tidak terlalu mengorbankan efisiensi. Pada akhirnya pemilihan algoritma bisa disesuaikan dengan kebutuhan pada setiap kasus tetapi jika harus memilih salah satu penulis akan memilih algoritma A* sebagai algoritma yang terbaik.

Saran

1. Bisa dibuat makefile agar mempermudah compile

LAMPIRAN

Link Repository Github: https://github.com/rendangmunir/Tucil3_13522151

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		✅

DAFTAR PUSTAKA

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf