LAPORAN TUGAS KECIL III

IF2211 STRATEGI ALGORITMA

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

Disusun oleh:

Suthasoma Mahardhika Munthe 13522098

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2024

Daftar Isi

BAGIAN I ANALISIS DAN IMPLEMENTASI 3
BAGIAN II SOURCE CODE PROGRAM 5
BAGIAN III SCREENSHOT HASIL TEST 15
BAGIAN IV ANALISIS HASIL 20
BAGIAN V BONUS 22
LINK REPOSITORY 22
CHECKLIST 22

BAGIAN I
ANALISIS DAN IMPLEMENTASI

Program yang dibuat mengimplementasikan algoritma UCS (*Uniform Cost Search*), *Greedy* BFS, dan A*. Pada algoritma UCS dan GBFS, simpul yang telah dikunjungi tidak akan ditambahkan ke dalam *queue* (antrian). Secara sederhana, jika ada simpul yang sama diciptakan (dilakukan *expand*) pada simpul yang sedang dikunjungi sekarang, maka simpul ini tidak akan ditambahkan karena kedalaman yang disimpan sama atau lebih dalam. Hal ini menyebabkan pemanfaatan set yang menyimpan simpul yang telah disimpan mencegah redudansi. Implementasi ketiga algoritma ini juga akan menggunakan *Priority Queue* untuk menyimpan antrian simpul yang akan diperiksa.

Implementasi *f(n), g(n),* dan *h(n)* akan dijelaskan pada bagian di bawah ini.

1. Fungsi *f(n)* atau *cost* yang diterapkan disesuaikan dengan tipe algoritma yang akan digunakan. Pada algoritma UCS, *f(n) = g(n)*. Pada algoritma GBFS, *f(n) = h(n)*. Sementara pada algoritma A*, *f(n) = g(n) + h(n)*.
2. Fungsi *g(n)* adalah fungsi *cost* dari simpul akar hingga simpul saat ini. Artinya, setiap terjadi *expand*, simpul yang diciptakan akan bertambah nilainya sebanyak satu.
3. Fungsi *h(n)* adalah fungsi heuristik yang digunakan untuk menghitung jumlah karakter yang berbeda dari simpul saat ini dengan simpul tujuan. Contohnya, jika simpul saat ini adalah "TOOL" dan simpul tujuan adalah "ROLL", maka nilai heuristiknya adalah 2. Jika simpul tujuan sama dengan simpul saat ini, tentu saja *cost*-nya adalah nol.

Secara teori, *h(n)* yang digunakan adalah *admissible*. Jumlah kata berbeda secara harfiah menunjukkan bahwa nilai heuristik sama dengan nilai sesungguhnya yang akan diperlukan untuk mencapai tujuan. Jika jumlah karakter yang berbeda adalah tiga, maka langkah pengubahan huruf menuju tujuan optimalnya adalah sebanyak 3 kali. Sehingga *h(n) = h*(n)*. Berdasarkan referensi dari salindia kuliah alasan ini menunjukkan bahwa algoritma A* yang akan direalisasikan akan menjamin hasil yang optimal.

Langkah-langkah yang dilakukan pada algoritma UCS dapat dilihat pada penjelasan di bawah ini.

1. Bangkitkan simpul akar, yaitu simpul *start* dan masukkan ke dalam antrian.
2. Jika antrian sudah kosong lompat ke proses 7.
3. Keluarkan satu simpul dari antrian dengan *cost* terkecil.
4. Jika simpul ini merupakan tujuan maka lompat ke proses 7 dan setel simpul ini sebagai solusi.
5. Jika simpul ini bukan tujuan maka lakukan *expand*. Bangkitkan setiap simpul baru (semua kata baru yang mungkin dicapai dengan mengubah satu huruf dan valid atau sesuai dengan kamus). Kemudian tambahkan semua simpul ini ke dalam antrian dengan mempertimbangkan nilai *cost*-nya (diurutkan secara menaik) jika sebelumnya belum pernah dikunjungi. *Cost* yang digunakan algoritma ini adalah *g(n)*.
6. Kembali ke proses 2.
7. Terminasi program.

Langkah-langkah yang dilakukan pada algoritma GBFS dapat dilihat pada penjelasan di bawah ini.

1. Bangkitkan simpul akar, yaitu simpul *start* dan jadikan simpul ini sebagai simpul *expand*.
2. Jika simpul *expand* saat ini adalah NULL, lompat ke proses 6.
3. Jika simpul *expand* saat ini merupakan tujuan maka lompat ke proses 6 dan setel simpul ini sebagai solusi.
4. Jika simpul ini bukan tujuan maka lakukan *expand*. Bangkitkan setiap simpul baru (semua kata baru yang mungkin dicapai dengan mengubah satu huruf yang valid atau sesuai dengan kamus). Kemudian dengan melakukan *sorting* pada semua simpul anak yang valid, pilih simpul anak dengan nilai *cost* yang paling kecil. *Cost* yang digunakan algoritma ini adalah *h(n)*.
5. Kembali ke proses 2.
6. Terminasi program.

Langkah-langkah yang dilakukan pada algoritma A* dapat dilihat pada penjelasan di bawah ini.

1. Bangkitkan simpul akar, yaitu simpul *start* dan masukkan ke dalam antrian.
2. Jika antrian sudah kosong lompat ke proses 7.
3. Keluarkan satu simpul dari antrian dengan *cost* terkecil.
4. Jika simpul ini merupakan tujuan maka lompat ke proses 7 dan setel simpul ini sebagai solusi.
5. Jika simpul ini bukan tujuan maka lakukan *expand*. Bangkitkan setiap simpul baru (semua kata baru yang mungkin dicapai dengan mengubah satu huruf yang valid atau sesuai dengan kamus). Kemudian tambahkan semua simpul ini ke dalam antrian dengan mempertimbangkan nilai *cost*-nya (diurutkan secara menaik). *Cost* yang digunakan algoritma ini adalah *g(n) + h(n)*. Pada algoritma ini tidak ada dilakukan pengecekan terhadap simpul yang pernah dikunjungi.
6. Kembali ke proses 2.
7. Terminasi program.

Ada sedikit modifikasi pada algoritma UCS dan A*. Pada algoritma UCS setiap penambahan simpul baru ke dalam antrian, akan dicek terlebih dahulu apa sebelumnya pernah dikunjungi atau belum. Jika belum pernah dikunjungi maka akan ditambahkan ke dalam antrian sekaligus menyetel simpul ini menjadi sudah pernah dikunjungi. Tujuannya adalah supaya tidak terjadi redundansi data simpul. Beberapa simpul hidup yang diciptakan dari simpul *expand* pada kedalaman yang sama tidak perlu ditambahkan kedalam antrian karena satu simpul induk saja sudah valid untuk menghasilkan solusi yang optimal jika solusi memang ada.

Sementara itu, pada algoritma A* ditambahkan pula variabel yang akan menyimpan riwayat pemeriksaan simpul untuk mencegah terjadinya pemakaian memori yang tidak terkontrol. Tujuan lain dari proses ini adalah untuk memangkas semua simpul pada state yang sama agar tidak terjadi pengulangan yang tidak diperlukan dalam pembentukan pohon pencarian.

Urutan simpul yang akan dibangkitkan pada algoritma UCS dan GBFS tidaklah sama. Hal ini terjadi karena perbedaan nilai *cost* yang digunakan. Oleh karena itu, simpul yang dibangkitkan tidak akan membentuk pohon pencarian yang sama.

Secara teoritis, pada kasus *word ladder* UCS lebih efisien daripada A* karena ada pada algoritma UCS simpul dengan nilai *g(n)* terkecil diperiksa terlebih dahulu. Sementara pada algoritma A*, yang diperiksa adalah nilai *g(n) + h(n)* yang mana dapat terjadi *backtracking* ke simpul dengan nilai *g(n)* yang lebih kecil. Selain itu, pada prosesnya A* masih lakukan pencarian meski solusi ditemukan untuk memastikan solusi yang dicapai adalah yang paling optimal.

Algoritma GBFS tidak memberikan solusi yang optimal karena penggunaan nilai *cost* yang ditetapkan tidak menjamin simpul yang valid dan lebih dekat ke tujuan ada. Hal ini juga disebabkan oleh ketidakmampuan algoritma ini dapat berhenti pada solusi optimal lokal dan tidak dapat melakukan *backtracking* ke simpul yang sebelumnya di-*expand*.

BAGIAN II
SOURCE CODE PROGRAM

Proyek ini diimplementasikan menggunakan Bahasa Java dengan compiler versi 21+. Kelas fungsionalitas utama yang dibuat terbagi menjadi beberapa kelas di bawah ini.

1. Search, UCSearch, AStarSearch, dan BestFirstSearch
   Kelas Search adalah kelas abstrak yang akan diturunkan pada ketiga kelas di atas. Ada satu fungsi yang harus diimplementasikan kelas turunannya. Kelas UCSearch adalah kelas yang akan melakukan proses pencarian dengan algoritma UCS. Begitu pula dengan dua kelas lain yang disebutkan di atas. Berikut ini adalah source code kelas-kelas tersebut.

Search.java

```java
package util;

import java.util.Comparator;
import java.util.HashSet;
import java.util.PriorityQueue;

public abstract class Search {

    protected String start;
    protected String end;
    protected int counterNode;
    protected boolean found;
    protected Node solution;
    protected PriorityQueue<Node> queue;
    protected HashSet<String> visited;

    public Search(String start, String end, Comparator<Node> comparator) {
        this.start = start;
        this.end = end;
```

```
        this.counterNode = 0;
        this.solution = null;
        this.found = false;
        this.queue = new PriorityQueue<>(comparator);
        visited = new HashSet<>();
    }

    public int getCounterNode() {
        return this.counterNode;
    }

    public Node getSolution() {
        return this.solution;
    }

    public abstract void search();
}
```

UCSearch.java

```
package util;

import java.util.ArrayList;

public class UCSearch extends Search{

    public UCSearch(String start, String end) {
        super(start, end, new UCSComparator());
    }

    @Override
    public void search() {
        Node startNode = new Node(start);
        queue.add(startNode);
        visited.add(startNode.getValue());

        while (!queue.isEmpty() && !found) {
            counterNode++;
            Node currNode = queue.poll();

            if (currNode.getValue().equals(end)) {
                found = true;
                solution = currNode;
                continue;
            }

            ArrayList<Node> childs = currNode.generateChild();
```

```
            for (Node child : childs) {
                if (!visited.contains(child.getValue())){
                    queue.add(child);
                }
                visited.add(child.getValue());
            }
        }
    }

}
```

BestFirstSearch.java

```
package util;

public class BestFirstSearch extends Search {

    public BestFirstSearch(String start, String end) {
        super(start, end, null);
    }

    public void search() {
        Node startNode = new Node(start);
        Node currNode = startNode;

        while (currNode != null && !found) {
            counterNode++;

            if (currNode.getValue().equals(end)) {
                found = true;
                solution = currNode;
                continue;
            }

            currNode = currNode.generateChild(end);
        }
    }

}
```

AStarSearch.java

```
package util;

import java.util.ArrayList;

public class AStartSearch extends Search{
```

```
    public AStartSearch(String start, String end) {
        super(start, end, new AStarComparator());
    }

    public void search() {
        Node startNode = new Node(start);
        queue.add(startNode);

        while (!queue.isEmpty()) {
            Node currNode = queue.poll();

            visited.add(currNode.getValue());
            counterNode++;

            if (currNode.getValue().equals(end)) {
                if (found) {
                    if (solution.length() > currNode.length()){
                        solution = currNode;
                    }
                } else {
                    solution = currNode;
                    found = true;
                }

                continue;

            } else if (found) {
                if (currNode.getPriorityValue() > solution.length()) {
                    continue;
                }
            }
            ArrayList<Node> childs = currNode.generateChildHeuristic(end);

            for (Node child : childs) {
                if (!visited.contains(child.getValue())) {
                    queue.add(child);
                }
            }
        }
    }
}
```

2. Kelas Node, AStarComparator, dan UCSComparator
   Kelas Node adalah kelas yang akan merepresentasikan simpul pencarian. Kelas ini dibuat mirip dengan *linked list* untuk mengurangi penggunaan memori berlebihan. Kelas Node akan menerapkan method *generator child* yang saat simpul dari Node ini sedang di-*expand*. Ketiga kelas lainnya digunakan dalam melakukan komparasi saat dimasukkan kedalam *Priority Queue* milik Java. *Comparator* ini disesuaikan dengan nilai *cost* yang diterapkan sesuai dengan algoritma pencarian yang digunakan. Berikut ini adalah *source code* kelas-kelas tersebut.

Node.java

```
package util;

import java.util.ArrayList;
import java.util.HashSet;

public class Node {

    /**
     * Menyimpan dictionary yang akan digunakan untuk validasi Node baru
yang akan diciptakan
     */
    public static HashSet<String> dictionary;

    public static char letters[] = new char[]{
        'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
        'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'
    };

    public static void initDictionary(String path) {
        try {
            dictionary = Dictionary.getDictionary(path);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // Mengandung node dari parent-nya
    private Node before;

    // Nilai yang tersimpan
    private String value;
    // Number dari node g(n)
    private int length;
    // The heuristic value h(n)
    private int heuristic;
```

```
    public Node() {
        value = "";
        before = null;
        length = 0;
        heuristic = 0;
    }

    public Node(String value) {
        this.value = value;
        length = 1;
        before = null;
        heuristic = 0;
    }

    public Node(String value, int heuristic) {
        this(value);
        this.heuristic = heuristic;
    }

    public int length() {
        return length;
    }

    public void setLength(int length) {
        this.length = length;
    }

    public String getValue() {
        return value;
    }

    public void setParent(Node parent) {
        before = parent;
    }

    public Node getParent() {
        return before;
    }

    public int getHeuristic() {
        return heuristic;
    }

    private static int countHeuristic(String current, String target) {
        int len = target.length();
```

```
        int counter = 0;

        for (int i = 0; i < len; i++) {
            if (current.charAt(i) != target.charAt(i)) {
                counter++;
            }
        }

        return counter;
    }

    public int getPriorityValue() {
        return length + heuristic;
    }

    private boolean isValueVisited(String val) {
        Node current = this;
        while (current != null) {
            if (current.value.equals(val)) {
                return true;
            }
            current = current.before;
        }
        return false;
    }

    public ArrayList<Node> generateChild() {
        StringBuilder builder = new StringBuilder(value);

        int len = value.length();
        ArrayList<Node> list = new ArrayList<>();

        for (int i = 0; i < len; i++) {

            if (i > 0){
                builder.setCharAt(i-1, value.charAt(i-1));
            }

            for (int j = 0; j < 26; j++) {

                builder.setCharAt(i, letters[j]);
                String newValue = new String(builder);

                if (dictionary.contains(newValue)) {
```

```
                    if (!isValueVisited(newValue)) {

                        Node newNode = new Node(newValue);
                        newNode.setParent(this);
                        newNode.setLength(this.length+1);
                        list.add(newNode);

                    }
                }
            }
        }

        return list;
    }

    public ArrayList<Node> generateChildHeuristic(String target) {
        StringBuilder builder = new StringBuilder(value);

        int len = value.length();
        ArrayList<Node> list = new ArrayList<>();

        for (int i = 0; i < len; i++) {

            if (i > 0){
                builder.setCharAt(i-1, value.charAt(i-1));
            }

            for (int j = 0; j < 26; j++) {

                builder.setCharAt(i, letters[j]);
                String newValue = new String(builder);

                if (dictionary.contains(newValue)) {
                    if (!isValueVisited(newValue)) {

                        Node newNode = new Node(newValue,
countHeuristic(newValue, target));
                        newNode.setParent(this);
                        newNode.setLength(this.length+1);
                        list.add(newNode);

                    }
                }
            }
        }
```

```
        return list;
    }

    public Node generateChild(String target) {
        StringBuilder builder = new StringBuilder(value);

        int len = value.length();

        int minHeuristic = len;
        String minString = null;

        for (int i = 0; i < len; i++) {

            if (i > 0){
                builder.setCharAt(i-1, value.charAt(i-1));
            }

            for (int j = 0; j < 26; j++) {

                builder.setCharAt(i, letters[j]);
                String newValue = new String(builder);

                if (dictionary.contains(newValue)) {
                    if (!isValueVisited(newValue)) {

                        int currentHeuristic = countHeuristic(newValue,
target);

                        if (currentHeuristic < minHeuristic) {
                            minHeuristic = currentHeuristic;
                            minString = newValue;
                        }
                    }
                }
            }
        }

        if (minString == null) {
            return null;
        }

        Node out = new Node(minString, minHeuristic);
        out.setParent(this);
        out.setLength(this.length + 1);
```

```
        return out;
    }

}
```

AStarComparator.java

```
package util;

import java.util.Comparator;

public class AStarComparator implements Comparator<Node>{
    @Override
    public int compare(Node obj1, Node obj2){
        return Integer.compare(obj1.getPriorityValue(),
obj2.getPriorityValue());
    }
}
```

UCSComparator.java

```
package util;

import java.util.Comparator;

public class UCSComparator implements Comparator<Node>{
    @Override
    public int compare(Node obj1, Node obj2){
        return Integer.compare(obj1.getPriorityValue(),
obj2.getPriorityValue());
    }
}
```

3. Kelas Dictionary

Kelas ini adalah kelas yang akan membaca kamus di mana kata-kata yang valid akan disimpan. Fungsi statik yang diimplementasikan akan membaca file eksternal dan akan mengembalikan data dalam bentuk *Hash Set* untuk mempercepat proses pembacaan dan pengecekan kamus.

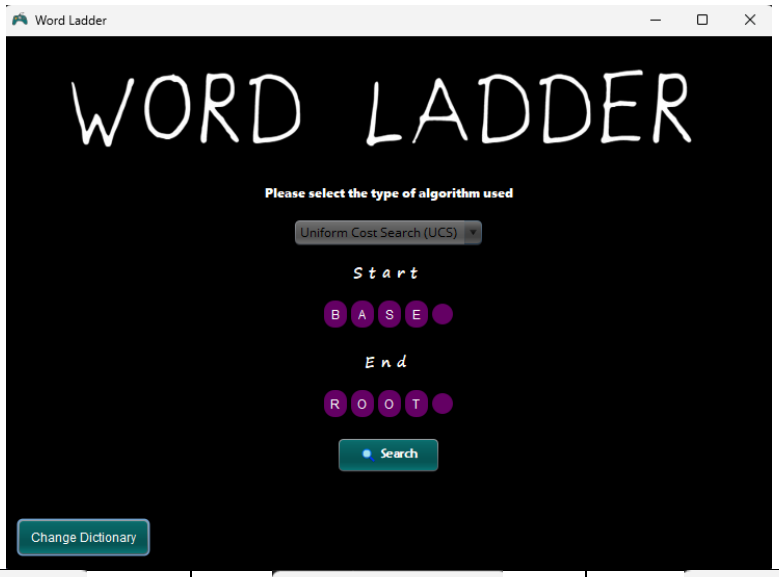
```
      package util;

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

public class Dictionary {
    public static HashSet<String> getDictionary() throws Exception {

        HashSet<String> dictionary = new HashSet<>();
```

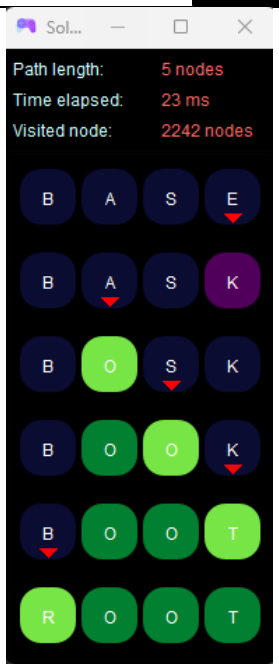
```
        String inputFile = "src/util/dictionary.txt";

        try (BufferedReader br = new BufferedReader(new
FileReader(inputFile))) {
            String line;
            while ((line = br.readLine()) != null) {
                dictionary.add(line);
            }
            System.out.println("Berhasil memuat kamus data.");
            return dictionary;

        } catch (Exception e) {
            throw e;
        }
    }
}
```

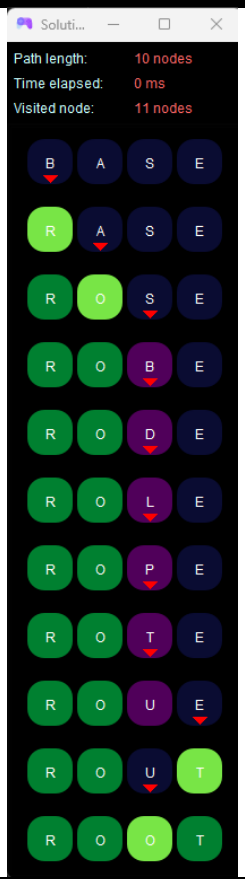
BAGIAN III
SCREENSHOT HASIL TEST

UCS	GBFS	A*
Tes pertama Start: ROOT		

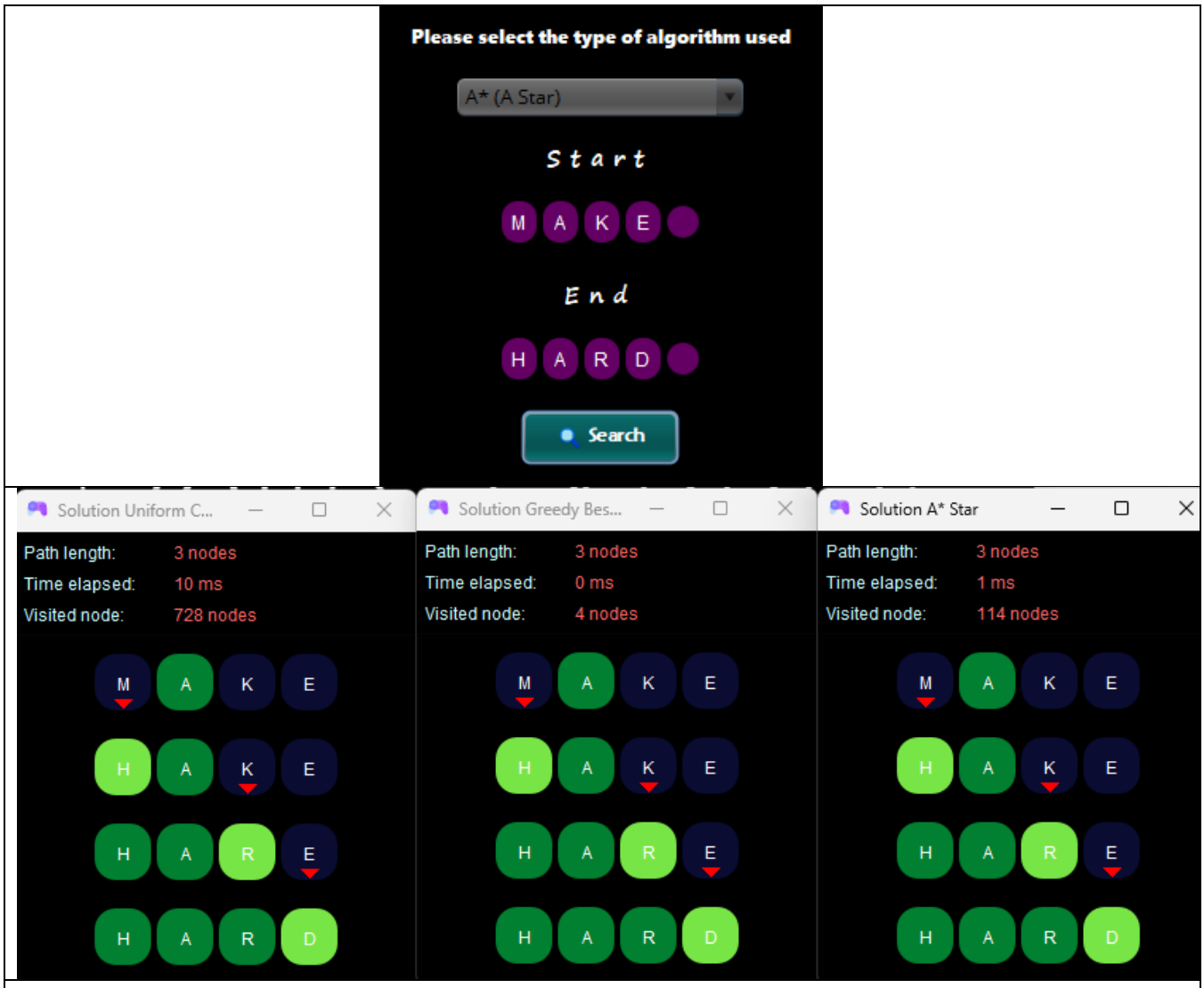
End: BASE



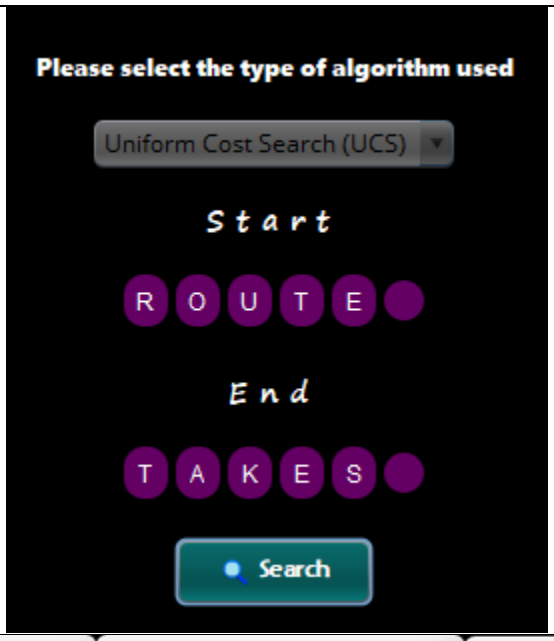




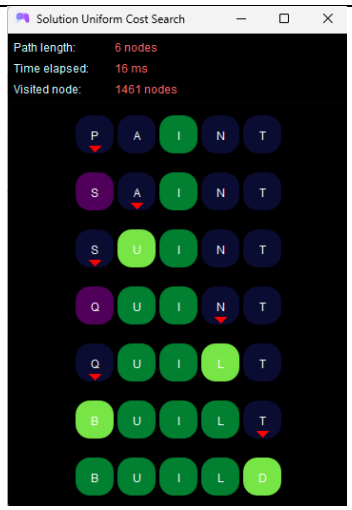
Tes Kedua
Start: MAKE
End: HARD

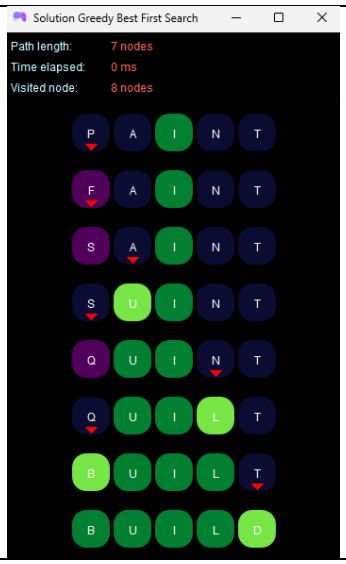


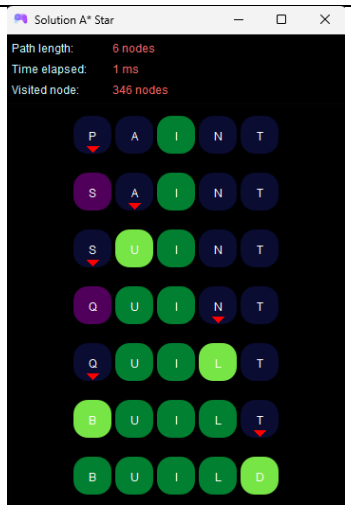
Tes Ketiga
Start: ROUTE
End: TAKES



Tes Keempat
Start: PAINT
End: BUILD

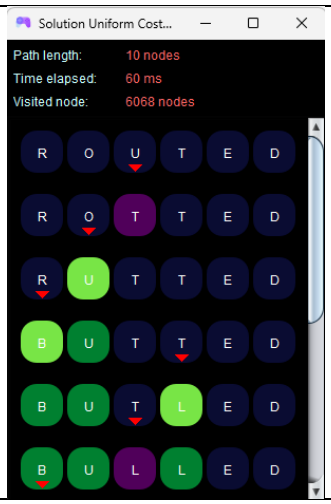
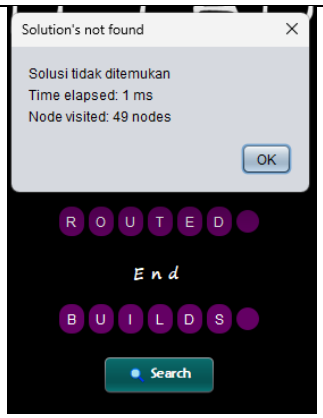
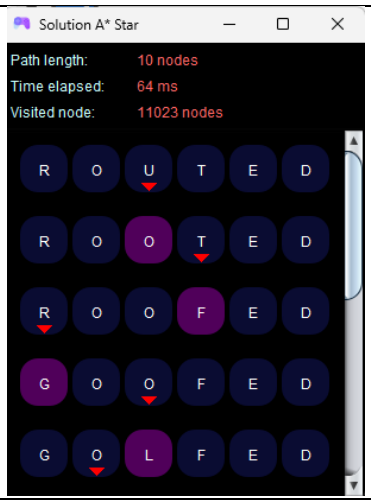
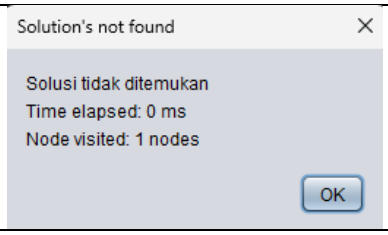
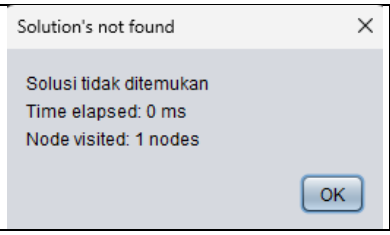
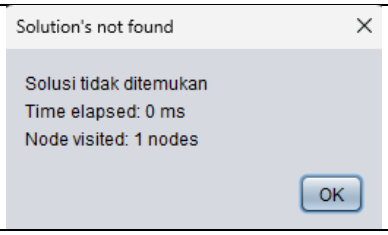






Tes Kelima
Start: CATTY
End: DUCKY

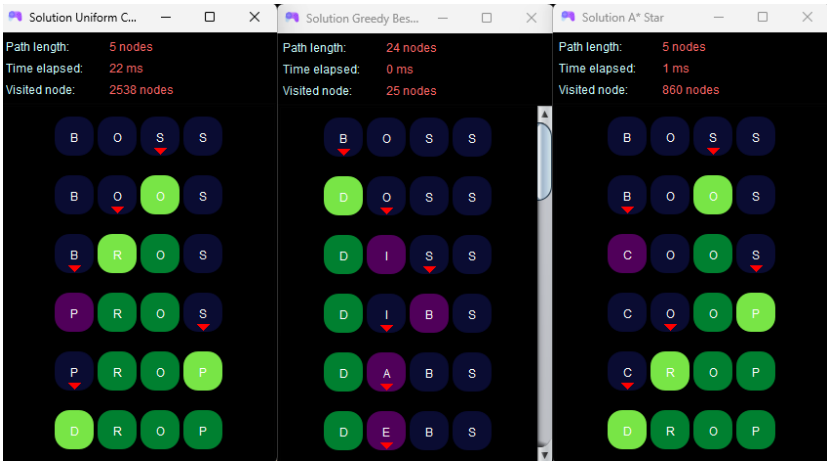
Tes Keenam
Start: ROUTED
End: BUILDS

<table>
<tr>
<td> 
</td>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td colspan="3">Tes Ketujuh
Start: CONTROL
End: GRANTED</td>
</tr>
<tr>
<td> 
</td>
<td> 
</td>
<td> 
</td>
</tr>
</table>

BAGIAN IV
ANALISIS HASIL

Berdasarkan beberapa contoh pengujian pada Bagian III dan beberapa contoh pengujian lain yang tidak diikutsertakan, UCS dan A* memberikan solusi yang optimal. Sementara algoritma GBFS tidak memberikan solusi yang optimal dan bahkan tidak menunjukkan solusi walaupun sebenarnya ada solusi. Hal ini terjadi karena GBFS tidak dapat melakukan *backtrack* sehingga simpul hidup hasil *expand* dari simpul leluhur sebelumnya tidak akan di cek lagi. GBFS juga tidak memberikan solusi yang optimal. Hal ini dapat dilihat dari pengujian pertama pada bagian ini. Pada contoh ini dapat dilihat jika panjang lintasan yang didapat pada solusi GBFS adalah 10, tetapi UCS dan A* hanya 5.

Waktu eksekusi yang diperlukan setiap algoritma berbeda-beda. UCS tergolong lebih stabil perubahan waktu eksekusinya dari pada A*. Sementara algoritma GBFS sendiri memang jauh lebih cepat dari kedua algoritma lain karena langsung mematikan simpul hidup hasil *expand* yang tidak terpilih. Hal ini yang kemudian menyebabkan jumlah simpul yang diperiksa menjadi lebih sedikit daripada kedua algoritma lainnya. Beberapa kasus di bawah dapat dilihat untuk menjelaskan perbedaan waktu eksekusi setiap algoritma yang digunakan.



Gambar 4.1 Hasil eksekusi (kasus boss ke drop)

Algoritma A* sedikit dimodifikasi pada bagian ini sesuai dengan apa yang dijelaskan pada bagian ini. Modifikasi ini tetap menjaga optimalitas dari algoritma A* sendiri. Algoritma GBFS, seperti yang sudah dijelaskan sebelumnya, tidak optimal. Dapat dilihat pada Gambar 4.1 di atas, algoritma GBFS menunjukkan panjang lintasan 24 simpul, jauh lebih besar dari solusi optimal, yaitu 5 simpul saja.

Memori yang diperlukan untuk menyelesaikan permainan pada algoritma A* adalah yang paling besar karena proses pemeriksaan untuk menangani simpul yang sudah diperiksa sebelumnya tidak seoptimal UCS. GBFS sendiri tidak membutuhkan banyak memori karena untuk setiap iterasi simpul yang diperiksa tepat hanya satu sehingga kompleksitas memori GBFS adalah *O(n)* dengan *n* adalah kedalaman pencarian. UCS memiliki kompleksitas memori O(b^n) dengan *b* adalah maksimal cabang ke simpul anak dari simpul induk yang mungkin dan *n* adalah kedalaman pencarian. Nilai *b* adalah panjang *word* yang diperiksa dikali dengan jumlah huruf yang valid, yaitu 26 huruf. Misalnya *b* = 104 untuk kata ‘drop’. A* juga memiliki kompleksitas memori yang sama seperti UCS. Namun, filterisasi simpul anak tidak sebaik UCS sehingga pada praktiknya memori yang diperlukan A* lebih besar. Perbedaan ukuran memori yang digunakan untuk ketiga algoritma yang digunakan dapat dilihat pada uji coba kasus ‘routed’ ke ‘builds’ di bawah ini.

```
run:
Berhasil memuat kamus data.
src/util/dictionary.txt
Total consumed memory: 30207 KB
```

a

```
run:
Berhasil memuat kamus data.
src/util/dictionary.txt
Total consumed memory: 2048 KB
```

b

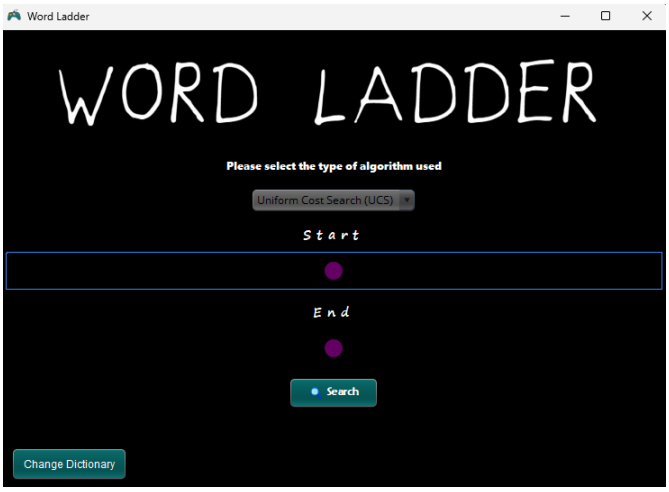
```
run:
Berhasil memuat kamus data.
src/util/dictionary.txt
Total consumed memory: 13842 KB
```

c

Gambar 4.2 Memori yang terpakai a) UCS b) GBFS c) A*

BAGIAN V
BONUS

Bonus yang dikerjakan adalah pembuatan GUI. Pada bagian bonus ini, penulis harus mengubah struktur folder yang ditentukan karena harus mengikuti struktur bawaan IDE Apache Netbeans. Folder *src* masih sesuai, namun folder *bin* akan digantikan dengan folder *build* dan *dist*. Folder *build* akan berisi file hasil kompilasi program. Folder *dist* akan berisi file *Driver* yang berekstensikan *jar*.



Gambar 5.1 Dashboard

Penjelasan penggunaan sudah disertakan pada README file, silahkan dibaca terlebih dahulu. Untuk memasukkan *start word* dan *end word* fokus harus ada pada komponen tersebut. Seperti contoh pada Gambar 5.1 di atas, fokus sudah ada pada komponen *start input text* sehingga pengguna dapat mengetikkan kata yang akan dimasukkan.

Ada fitur tambahan digunakan untuk memudahkan pengguna dalam mengganti kamus bahasa yang digunakan. Silahkan digunakan untuk mengganti kamus jika diperlukan.

LINK REPOSITORY

https://github.com/sotul04/Tucil3_13522098

CHECKLIST

Poin	Ya	Tidak
1. Program berhasil dijalankan.	✓	
2. Program dapat menemukan rangkaian kata dari start word ke end word sesuai denga 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.	✓	