LAPORAN TUGAS KECIL 3 IF2211
STRATEGI ALGORITMA

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

Disusun oleh:

Keanu Amadius Gonza Wrahatno - 13522082

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2023/2024

DAFTAR ISI

BAB I
DESKRIPSI TUGAS

Gambar 1. Ilustrasi dan Peraturan Permainan Word Ladder

(Sumber: https://wordwormdormdork.com/)

Word ladder (juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragrams, laddergrams, atau word golf) adalah salah satu permainan kata yang terkenal bagi seluruh kalangan. Word ladder ditemukan oleh Lewis Carroll, seorang penulis dan matematikawan, pada tahun 1877. Pada permainan ini, pemain diberikan dua kata yang disebut sebagai start word dan end word. Untuk memenangkan permainan, pemain harus menemukan rantai kata yang dapat menghubungkan antara start word dan end word. Banyaknya huruf pada start word dan end word selalu sama. Tiap kata yang berdekatan dalam rantai kata tersebut hanya boleh berbeda satu huruf saja. Pada permainan ini, diharapkan solusi optimal, yaitu solusi yang meminimalkan banyaknya kata yang dimasukkan pada rantai kata. Berikut adalah ilustrasi serta aturan permainan.

BAB II
LANDASAN TEORI

2.1 Algoritma Uniform Cost Search (UCS)

Algoritma ini mirip dengan BFS, namun algoritma ini melakukan pencarian dengan ekspansi node berdasarkan cost / biaya dari root. Pada setiap langkah, ekspansi berikutnya ditentukan berdasarkan cost terendah atau disebut sebagai fungsi g(n) dimana g(n) merupakan jumlah biaya edge dari root menuju node n. Node-node tsb disimpan menggunakan priority queue.

Berikut merupakan skema umum dalam algoritma Uniform Cost Search.

```
function UNIFORM-COST-SEARCH (problem) returns a solution, or failure
    node ← a node with STATE = problem. INITIAL-STATE, PATH-COST = 0
    frontier ← a priority queue ordered by PATH-COST, with node as the only element
    explored ← an empty set

    loop do
    if EMPTY? (frontier) then return failure
    node ← POP(frontier)
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem. ACTIONS (node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
            frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
            replace that frontier node with child
```

2.2 Algoritma Greedy Best-First Search (GBFS)

Algoritma Greedy Best-First Search termasuk dalam algoritma Informed Search. Algoritma ini melakukan pencarian dengan mengekspansi node yang memiliki nilai heuristik h(n) paling rendah. Jarak heuristik diperoleh dengan memperkirakan "biaya" atau "jarak" dari simpul saat ini ke tujuan. Algoritma ini tidak dapat backtracking. Pendekatan ini mengasumsikan bahwa hal ini kemungkinan besar akan menghasilkan solusi dengan cepat.

Berikut merupakan skema umum dari algoritma Greedy Best-First Search.

```
function Best-First-Search(Graph g, Node start)
    pq ← a priority queue
```

```
  pq.insert(start)
  loop until (pq is empty)
      u ← PriorityQueue.DeleteMin
      if u is the goal
        Exit
      else
        for each neighbor v of u
          if v "Unvisited"
             Mark v "Visited"
             pq.insert(v)
        Mark u "Examined"
End procedure
```

2.3 Algoritma A* (A Star)

Algoritma A* (A Star) adalah algoritma pencarian yang digunakan untuk menemukan jalur terpendek antara titik awal dan akhir. A* menggunakan konsep algoritma BFS dan menemukan jalur dengan biaya terkecil dari node awal ke node tujuan. Algoritma ini menggunakan fungsi heuristik h(n) jarak ditambah biaya g(n) untuk menentukan urutan di mana search-nya melalui node-node yang ada pada tree. Jarak heuristik h(n) diperoleh dengan memperkirakan "biaya" atau "jarak" dari simpul saat ini ke tujuan. Biaya g(n) merupakan jumlah biaya edge dari root menuju node n.

Berikut merupakan skema umum dari algoritma A*.

```
function UNIFORM-COST-SEARCH (problem) returns a solution, or failure
   node ← a node with STATE = problem. INITIAL-STATE, PATH-COST = 0
   frontier ← a priority queue ordered by PATH-COST, with node as the only element
   explored ← an empty set

   loop do
   if EMPTY? (frontier) then return failure
   node ← POP(frontier)
   if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
   add node.STATE to explored
   for each action in problem. ACTIONS (node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
         frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
         replace that frontier node with child
```

BAB III

ANALISIS PEMECAHAN MASALAH

Pertanyaan pada spesifikasi akan dijawab pada bab ini

3.1 Proses Pemetaan Masalah

Dalam permainan Word Ladder ini, dibutuhkan kamus / data kata dalam bentuk txt. File txt tersebut nantinya akan diubah kedalam sebuah array list of String kata. String dari kata tersebut nantinya akan diubah menjadi node saat simpul tersebut akan dimasukan ke priority queue. Node merupakan representasi dari kata yang di dapat dari file txt dengan struktur berikut ini

1. String *Nama* = merupakan kata pada kamus
2. Integer *Cost* = nilai cost yang akan digunakan untuk parameter PriorityQueue
3. Node *Parent* = mencatat parent untuk membuat path nantinya
4. Integer *timestamp* = untuk menerapkan konsep FIFO apabila cost nya sama

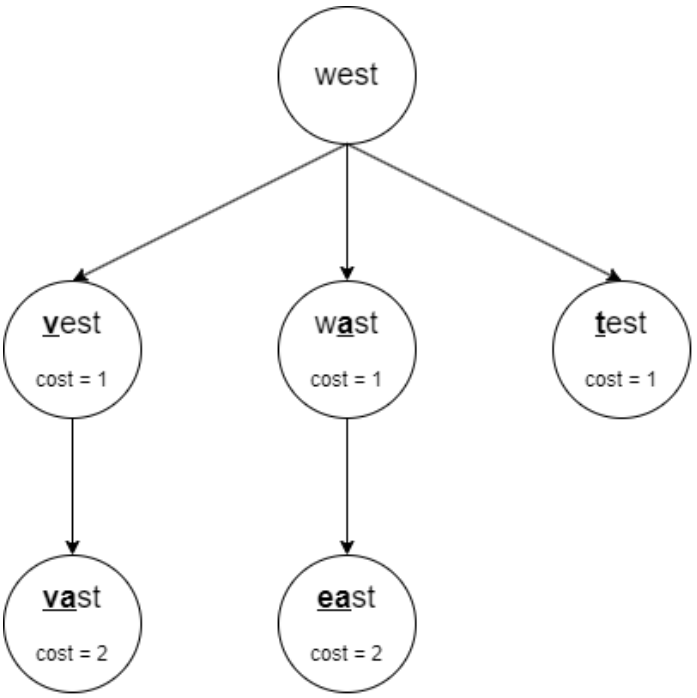
Untuk menampung node & memproses node mana yang akan dilakukan pembangkitan, dibuat Priority Queue. Priority Queue yang penulis buat menerapkan konsep FIFO, dimana apabila ada node yang costnya sama ma ayang didahulukan untuk diproses adalah node yang First In atau yang labih dulu ada di Priority Queue.

Permasalahan ini dapat digambarkan sebagai sebuah graf dinamis atau graf yang terbentuk saat proses pencarian dilakukan. Pada graf dinamis ini, pembangkitan simpul barunya dilakukan dengan fungsi getNeighbors yang akan mencari kata yang memiliki panjang yang sama dengan perbedaan 1 huruf. Contoh node saat ini adalah “kamu”, maka tetangganya adalah “kami”, “tamu”.

Penulis menerapkan optimalisasi terhadap jumlah node yang dicek. Apabila sudah menemukan child yang merupakan node tujuan, maka program akan berhenti dan menghasilkan pathnya. Pada algoritma UCS dan A*, node child akan dimasukan kedalam visited agar tidak terjadi pengecekan berulang kali. Hal ini tidak berpengaruh terhadap hasil panjang path.

3.1.1 Pemetaan ke Algoritma UCS

Algoritma UCS merupakan algoritma Uninformed Search. Algoritma ini akan membangkitkan node dengan cost terendah berdasarkan biaya dari node awal ke node n atau bisa dikatakan g(n). Pada kasus Word Ladder, pengubahan kata hanya boleh 1 huruf yang berarti akan dibentuk child yang memiliki kata dengan perbedaan 1 huruf. Karena child pasti hanya punya perbedaan 1 huruf, maka setiap pembangkitan status costnya akan bertambah 1.



Gambar 3.1.1. Ilustrasi UCS

(Sumber: Milik Pribadi Penulis)

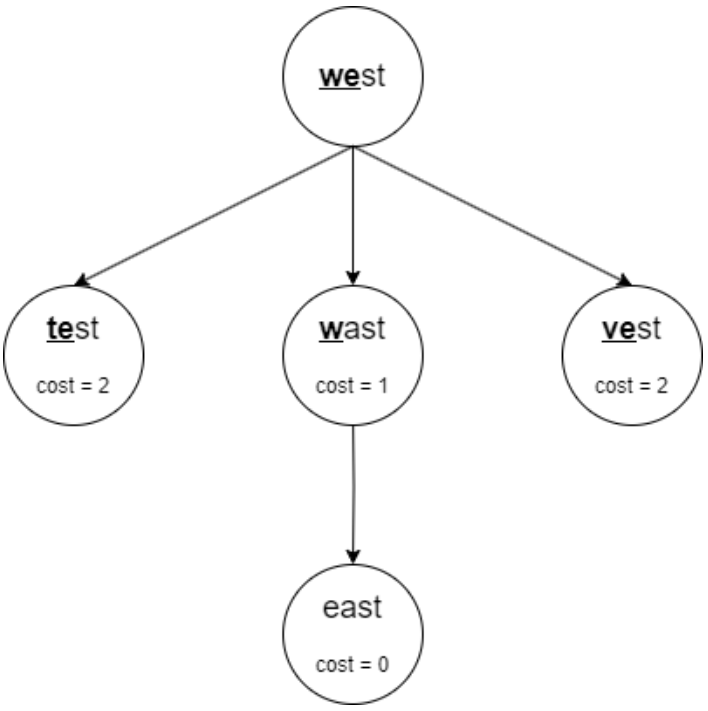
Dapat dilihat bahwa cost g(n) pada UCS merepresentasikan kedalaman pada BFS. UCS akan cenderung membangkitkan node dengan cost terendah sehingga sama dengan konsep BFS yang membangkitkan node pada kedalaman rendah terlebih dahulu. Algoritma UCS memiliki urutan pembangkitan node yang sama seperti BFS dan akan menghasilkan path yang sama dengan BFS.

Node yang dibangkitkan akan disimpan pada Priority Queue FIFO dengan parameter cost. Selama Queue tidak kosong atau Node tujuan belum ditemukan, bangkitkan node dengan pop pada Queue.

Karena algoritma UCS akan membangkitkan node dengan cost terendah terlebih dahulu, maka algoritma ini akan berusaha mencari hasil path dengan jalur terpendek. Berarti hasil path yang dihasilkan algoritma ini adalah optimal.

3.1.2 Pemetaan ke Algoritma GBFS

Algoritma UCS merupakan algoritma Informed Search. Algoritma ini akan membangkitkan node dengan cost terendah berdasarkan jarak heuristiknya / h(n). Jarak heuristik merupakan perkiraan biaya dari node n ke node tujuan. Pada kasus Word Ladder, heuristik dihitung berdasarkan jumlah huruf pada kata saat ini yang berbeda dari kata tujuan. Dengan mengambil cost yang paling kecil, maka GBFS akan mencari jalur menuju kata tujuan.



Gambar 3.1.2. Ilustrasi GBFS

(Sumber: Milik Pribadi Penulis)

Pada kasus di atas, akan ditemukan path west → wast → east. Namun perlu dicatat bahwa algoritma greedy tidak dapat backtracking sehingga ia dapat terjebak dalam minimum lokal. Kalaupun dia tidak terjebak, algoritma ini akan terus membangkitkan node sampai ketemu hasil tanpa memperhatikan panjang atau optimalisasinya. Sehingga hasil yang dihasilkan tidak menjamin solusi yang optimal seperti contoh pada di bawah ini

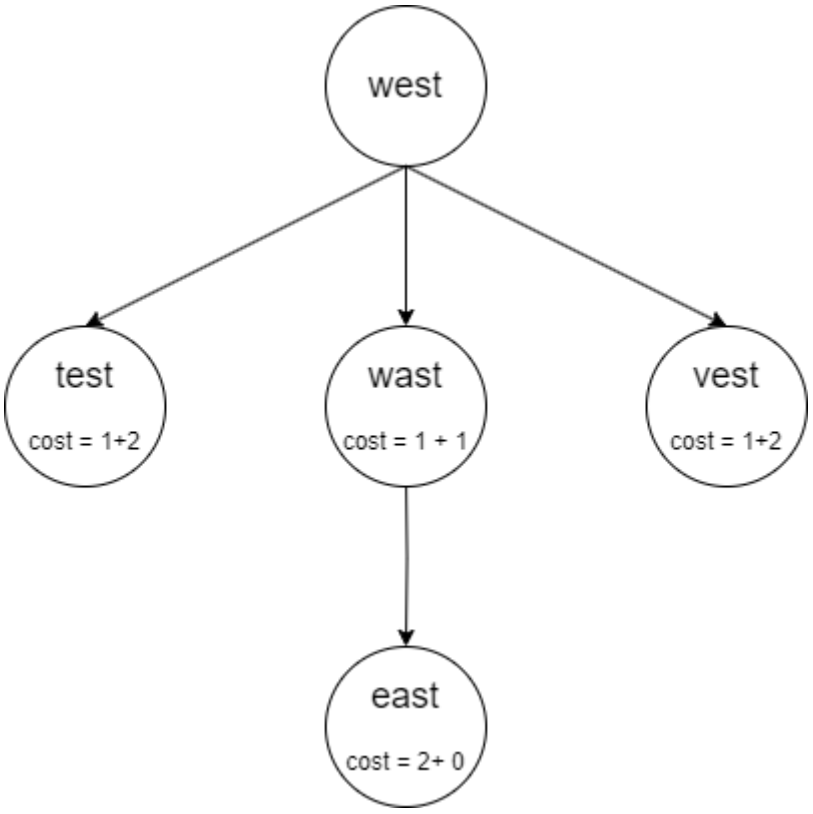
Greedy Best-First Search (GBFS) tidak menjamin solusi optimal karena metode ini cenderung memilih jalur yang "terlihat" paling menjanjikan berdasarkan fungsi heuristik, tanpa memperhitungkan total biaya yang diperlukan untuk mencapai tujuan.

Algoritma ini pertama tama akan mencari child dari west lalu memasukannya ke dalam priority queue. Priority queue disini hanya berfungsi untuk mencari nilai heuristik paling rendah. Lalu akan dibangkitkan node wast karena hanya punya 1 huruf pembeda dari east atau nilai heuristiknya adalah 1. Maka queue dikosongkan lalu diisi dengan child dari wast. Fungsi dari pengosongan queue ini adalah agar algoritma tidak backtracking.

Apabila queue hanya ada 1 elemen dan elemen tersebut sudah tidak memiliki tetangga lagi, maka algoritma ini akan terjebak dalam minimum lokal dan mengeluarkan hasil tidak ada jalur apabila elemen terakhir itu bukanlah kata tujuan. Tidak didapatkan jalur karena queue kosong dan sudah tidak ada lagi yang bisa di bangkitkan

3.1.3 Pemetaan ke Algoritma A*

Algoritma UCS merupakan algoritma Informed Search. Algoritma ini akan membangkitkan node dengan cost terendah berdasarkan biaya g(n) / biaya dari node awal ke node n dan biaya dari heuristiknya / h(n). Jarak heuristik merupakan perkiraan biaya dari node n ke node tujuan. Pada intinya A* merupakan algoritma UCS yang dimodifikasi dengan mempertimbangkan jarak heuristiknya. Dengan ini yang awalnya UCS mencari path optimal tanpa mempertimbangkan waktu, pada A* dilakukan percepatan untuk proses pencarian path yang optimal. Cost didapatkan dengan rumus f(n) = g(n) + h(n).



Gambar 3.1.3. Ilustrasi A*

(Sumber: Milik Pribadi Penulis)

Pada kasus di atas, akan ditemukan path west → wast → east. Perbedaan yang ditemukan pada UCS adalah urutan pembangkitan nodenya. Jika pada UCS urutannya west, test, wast, vest, lalu ke east. Pada A* urutannya langsung dari west, wast, east. Inilah fungsi dari penambahan jarak heuristik yang menyebabkan algoritma akan mengecek east setelah wast.

Node yang dibangkitkan akan dimasukan kedalam Priority Queue FIFO dengan pertimbangan cost nya. Apabila queue tidak kosong, maka node akan dibangkitkan sampai menemukan node tujuan, Apabila queue sudah kosong dan tidak mencapai node tujuan, maka akan dikeluarkan hasil “Jalur tidak ditemukan”.

A* memiliki keuntungan dari UCS dalam hal efisiensi karena heuristiknya dapat membantu mempersempit ruang pencarian. A* dapat mengurangi jumlah node yang harus diperiksa, sehingga lebih cepat daripada UCS.

Jarak heuristik dikatakan admissible apabila tidak melebih-lebihkan biaya dari node n ke tujuan. Dengan kata lain, $h(n) \leq h^*(n)$ untuk semua node n. Hal ini penting dalam A* karena menjamin bahwa solusi yang ditemukan akan optimal. Karena pada kasus Word Ladder heuristik yang digunakan adalah jumlah huruf yang berbeda antara node n ke node tujuan, dan setiap pergantian kata hanya boleh mengubah 1 huruf. Algoritma A* pada Word Ladder adalah admissible. Karena h(n) pada A* admissible, maka algoritma ini menghasilkan hasil yang optimal.

3.1 Langkah langkah implementasi

3.2.1 Langkah-langkah implementasi Algoritma UCS

Berikut ini merupakan langkah langkah dalam mengimplementasikan algoritma UCS ke code Java:

- Langkah pertama yaitu cek apakah start dan end ada dalam kamus.
- Apabila ada dalam kamus cek lagi apakah end = start. Jika iya maka keluarkan hasil dengan path hanya berisi kata start.
- Apabila berbeda maka lakukan pembangkitan node dan masukan node tersebut kedalam Priority Queue FIFO dengan pertimbangan cost terendah. Cost didapatkan dari g(n) = biaya dari node awal ke node n.
- Apabila queue tidak kosong, pop elemen pada queue dan lakukan pembangkitan node tersebut. Child dari node tersebut dimasukan lagi ke queue
- Lakukan sampai queue kosong / ditemukan node tujuan
- Bila Tidak ditemukan node tujuan, keluarkan hasil “Jalur tidak ditemukan”

3.2.2 Langkah-langkah implementasi GBFS

Berikut ini merupakan langkah langkah dalam mengimplementasikan algoritma A*ke code Java:

- Langkah pertama yaitu cek apakah start dan end ada dalam kamus.
- Apabila ada dalam kamus cek lagi apakah end = start. Jika iya maka keluarkan hasil dengan path hanya berisi kata start.
- Apabila berbeda maka lakukan pembangkitan node.
- Kosongkan queue dan masukan child dari node yang dibangkitkan tadi kedalam Priority Queue FIFO dengan pertimbangan cost terendah. Cost didapatkan dari jarak heuristiknya / h(n).
- Apabila queue tidak kosong, pop elemen pada queue dan lakukan pembangkitan node tersebut.
- Kosongkan queue lagi dan masukan child dari node yang dibangkitkan tadi

- Lakukan sampai queue kosong / ditemukan node tujuan
- Bila Tidak ditemukan node tujuan, keluarkan hasil “Jalur tidak ditemukan”

3.2.3 Langkah-langkah implementasi Algoritma A*

Berikut ini merupakan langkah langkah dalam mengimplementasikan algoritma A*ke code Java:

- Langkah pertama yaitu cek apakah start dan end ada dalam kamus.
- Apabila ada dalam kamus cek lagi apakah end = start. Jika iya maka keluarkan hasil dengan path hanya berisi kata start.
- Apabila berbeda maka lakukan pembangkitan node dan masukan node tersebut kedalam Priority Queue FIFO dengan pertimbangan cost terendah. Cost didapatkan dari f(n) = g(n) + h(n).
- Apabila queue tidak kosong, pop elemen pada queue dan lakukan pembangkitan node tersebut. Child dari node tersebut dimasukan lagi ke queue
- Lakukan sampai queue kosong / ditemukan node tujuan
- Bila Tidak ditemukan node tujuan, keluarkan hasil “Jalur tidak ditemukan”

BAB IV

IMPLEMENTASI CODE

Berikut ini merupakan implementasi code untuk algoritma Uniform Cost Search (UCS), Greedy Best-First Search (GBFS), A* (A Star) dalam menyelesaikan kasus Word Ladder.

4.1 UCS.java

```
public static List<String> UCS_Algorithm(String start, String end, List<String>
Database) {
    List<String> noPath = new ArrayList<>();

    //jika start == end maka keluarkan path 1
    if (start.equals(end)) {
        noPath.add(0,"1");
        noPath.add(1,start);
        return noPath;
    }

    //inisialisasi PriorityQueue
    PriorityQueue<Node> queue = new PriorityQueue<>();
    queue.add(new Node(start, 0, null));

    //Inisialisasi visited
    List<String> visited = new ArrayList<>();
    visited.add(start);


    int count = 0;
    while (!queue.isEmpty()) { //loop sampai quue kosong
        Node currentNode = queue.poll(); //pop node pada queue
        String currentWord = currentNode.getWord();
        visited.add(currentWord);

        //jika sudah saampai tujuan, keluarkan hasil
        if (currentWord.equals(end)) {
            List<String> result = makePath(currentNode);
            result.add(0, String.valueOf(count));
            return result;
        }

        //cari tetangga dari currentWord
        List<String> neighbors = Neighbors.getNeighbors(currentWord, Database);
        count++;

        //masukan tetangga ke queue
        for (String neighbor : neighbors) {
            if (!visited.contains(neighbor)) {
                visited.add(neighbor);
                Node newNode = new Node(neighbor, currentNode.getCost() + 1,
currentNode);
```

```
                queue.add(newNode);
                //kalau ada tetangga yang sampai tujuan, keluarkan hasil
                if (neighbor.equals(end)) {
                    List<String> result = makePath(newNode);
                    result.add(0, String.valueOf(count));
                    return result;
                }
            }
        }
    }

    noPath.add(0, String.valueOf(count));
    return noPath;
}

//membuat path
private static List<String> makePath(Node node) {
    List<String> result = new ArrayList<>();
    while (node != null) {
        result.add(node.getWord());
        node = node.getPath();
    }
    Collections.reverse(result);
    return result;
}
```

4.2 GBFS.java

```
public class GBFS{

    public static List<String> GBFS_Algorithm(String start, String end,
List<String> Database) {
        List<String> noPath = new ArrayList<>();

        //jika start == end maka keluarkan path 1
        if (start.equals(end)) {
            noPath.add(0,"1");
            noPath.add(1,start);
            return noPath;
        }

        //inisialisasi PriorityQueue
        PriorityQueue<Node> queue = new PriorityQueue<>();
        queue.add(new Node(start, 0, null));

        //Inisialisasi visited
        List<String> visited = new ArrayList<>();
        visited.add(start);


        int count = 0;
        while (!queue.isEmpty()){ //loop sampai quue kosong
            Node currentNode = queue.poll(); //pop node pada queue
            queue.clear(); //menghapus queue agar tidak backtrack
```

```
            String currentWord = currentNode.getWord();
            visited.add(currentWord);

            //jika sudah saampai tujuan, keluarkan hasil
            if (currentWord.equals(end)) {
                List<String> result = makePath(currentNode);
                result.add(0, String.valueOf(count));
                return result;
            }

            //cari tetangga dari currentWord
            List<String> neighbors = Neighbors.getNeighbors(currentWord,
Database);
            count++;

            //masukan tetangga ke queue
            for (String neighbor : neighbors) {
                if (!visited.contains(neighbor)) {
                    Node newNode = new Node(neighbor, heuristicCost(neighbor,end),
currentNode);
                    queue.add(newNode);
                    if (neighbor.equals(end)) {
                        List<String> result = makePath(newNode);
                        result.add(0, String.valueOf(count));
                        return result;
                    }
                }
            }
        }

        noPath.add(0, String.valueOf(count));
        return noPath;
    }

    //menghitung cost heuristic
    private static int heuristicCost(String current, String target) {
        int count = 0;
        for (int i = 0; i < current.length(); i++) {
            if (current.charAt(i) != target.charAt(i)) {
                count++;
            }
        }
        return count;
    }

    //membuat path
    private static List<String> makePath(Node node) {
        List<String> result = new ArrayList<>();
        while (node != null) {
            result.add(node.getWord());
            node = node.getPath();
        }
        Collections.reverse(result);
        return result;
    }
}
```

4.3 AStar.java

```
public class AStar{

   public static List<String> AStar_Algorithm(String start, String end,
List<String> Database) {
       List<String> noPath = new ArrayList<>();

       //jika start == end maka keluarkan path 1
       if (start.equals(end)) {
           noPath.add(0,"1");
           noPath.add(1,start);
           return noPath;
       }

       //inisialisasi PriorityQueue
       PriorityQueue<Node> queue = new PriorityQueue<>();
       queue.add(new Node(start, 0, null));

       //Inisialisasi visited
       List<String> visited = new ArrayList<>();
       visited.add(start);

       int count = 0;
       while (!queue.isEmpty()){ //loop sampai quue kosong
           Node currentNode = queue.poll(); //pop node pada queue
           String currentWord = currentNode.getWord();
           visited.add(currentWord);

           //jika sudah saampai tujuan, keluarkan hasil
           if (currentWord.equals(end)) {
               List<String> result = makePath(currentNode);
               result.add(0, String.valueOf(count));
               return result;
           }

           //cari tetangga dari currentWord
           List<String> neighbors = Neighbors.getNeighbors(currentWord,
Database);
           count++;

           //masukan tetangga ke queue
           for (String neighbor : neighbors) {
               if (!visited.contains(neighbor)) {
                   visited.add(neighbor);
                   Node newNode = new Node(neighbor, count +
heuristicCost(neighbor,end), currentNode);
                   queue.add(newNode);
                   //kalau ada tetangga yang sampai tujuan, keluarkan hasil
                   if (neighbor.equals(end)) {
                       List<String> result = makePath(newNode);
                       result.add(0, String.valueOf(count));
                       return result;
                   }
               }
           }
//            Print.printPriorityQueue(queue);
```

```
        }

        noPath.add(0, String.valueOf(count));
        return noPath;
    }

    //menghitung cost heuristic
    private static int heuristicCost(String current, String target) {
        int count = 0;
        for (int i = 0; i < current.length(); i++) {
            if (current.charAt(i) != target.charAt(i)) {
                count++;
            }
        }
        return count;
    }

    //membuat path
    private static List<String> makePath(Node node) {
        List<String> result = new ArrayList<>();
        while (node != null) {
            result.add(node.getWord());
            node = node.getPath();
        }
        Collections.reverse(result);
        return result;
    }
}
```

4.4 Node.java

```
public class Node implements Comparable<Node> {
    Node path;
    String word;
    int cost;
    long timestamp;
    private static long Counter = 0;

    public Node(String word, int cost, Node path) {
        this.path = path;
        this.word = word;
        this.cost = cost;
        this.timestamp = Counter++; // Incremental unique sequence
    }

    public Node() {
        this.path = null;
        this.word = null;
        this.cost = 0;
        this.timestamp = Counter++; // Incremental unique sequence
    }

    public String getWord(){
        return word;
```

```
    }

    public int getCost(){
        return cost;
    }

    public Node getPath(){
        return path;
    }

    @Override
    public int compareTo(Node other) {
        int costComparison = Integer.compare(this.cost, other.cost);
        if (costComparison == 0) {
            return Long.compare(this.timestamp, other.timestamp);
        }
        return costComparison;
    }
}
```

4.5 Neighbors.java

```
public class Neighbors {
        public static List<String> getNeighbors(String FindWord, List<String>
Database){
        List<String> neighboors = new ArrayList<>();
        for(String Word : Database){
            int beda = 0;
            for(int i = 0; i < Word.length(); i++){
                if (Word.charAt(i) != FindWord.charAt(i)){
                    beda++;
                }
            }
            if(beda == 1){
                neighboors.add(Word);
            }

        }
        return neighboors;
    }
}
```

4.6 readFile.java

```
public class readFile {
    public static List<String> readWordsFromFile() {
        String filename = "words.txt";
        List<String> result = new ArrayList<>();
```

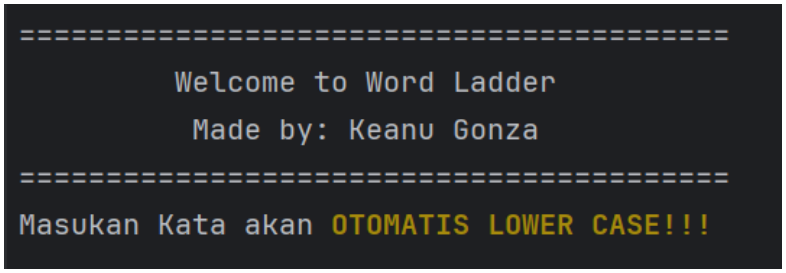
```
        try (BufferedReader br = new BufferedReader(new FileReader(filename))){
            String line;
            while ((line = br.readLine()) != null) {
                String[] lineWords = line.split("\\s+");
                result.addAll(Arrays.asList(lineWords));
            }
        } catch (IOException e){
            System.err.println("Error saat membaca file: " + e.getMessage());
        }
        return result;
    }

    public static boolean isExist(String word, List<String> Database){
        return Database.contains(word);
    }
}
```

BAB V

HASIL UJI DAN ANALISIS

Setelah ketiga algoritma (UCS, GBFS, A*) diimplementasikan, Penulis akan membandingkan hasil dari ketiga algoritma tersebut dan menganalisis kelebihan serata kekurangan untuk masing masing algoritma. Berikut merupakan tampilan awal dan hasil uji program nya

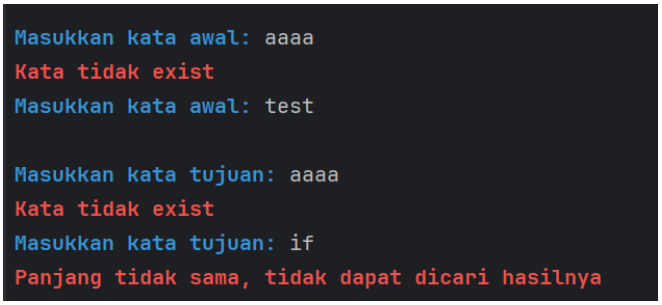


Gambar 5. Tampilan awal CLI

5.1 Hasil Uji Program Wajib

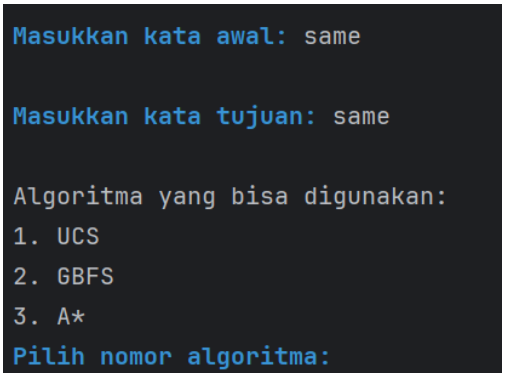
5.1.1 Uji Program 1

Hasil uji program menggunakan CLI dengan input salah / invalid input



5.1.2 Uji Program 2

Hasil uji program apabila kata awal dan tujuan sama



```
Algoritma UCS
======================JAWABAN======================
Jumlah Node dikunjungi: 1
Path: same
Time: 0.0012074 detik
```

```
Algoritma GBFS
======================JAWABAN======================
Jumlah Node dikunjungi: 1
Path: same
Time: 0.0011413 detik
```

```
Algoritma A star
======================JAWABAN======================
Jumlah Node dikunjungi: 1
Path: same
Time: 0.0011596 detik
```

5.1.3 Uji Program 3

Hasil Uji Normal Case

```
Masukkan kata awal: west

Masukkan kata tujuan: east

Algoritma yang bisa digunakan:
1. UCS
2. GBFS
3. A*
Pilih nomor algoritma:
```

```
Algoritma UCS
======================JAWABAN======================
Jumlah Node dikunjungi: 12
Path: west -> wast -> east
Time: 0.009904 detik
```

```
Algoritma GBFS
======================JAWABAN======================
Jumlah Node dikunjungi: 2
Path: west -> wast -> east
Time: 0.0061091 detik
```

```
Algoritma A star
======================JAWABAN======================
Jumlah Node dikunjungi: 2
Path: west -> wast -> east
Time: 0.0053248 detik
```

5.1.4 Uji Program 4

Hasil Uji Kasus dengan path panjang

```
Masukkan kata awal: crawler

Masukkan kata tujuan: cruiser

Algoritma yang bisa digunakan:
1. UCS
2. GBFS
3. A*
Pilih nomor algoritma:
```

```
Algoritma UCS
======================JAWABAN======================
Jumlah Node dikunjungi: 53
Path: crawler -> brawler -> brawled -> brailed -> braised -> bruised -> bruiser -> cruiser
Time: 0.0624118 detik
```

```
Algoritma GBFS
======================JAWABAN======================
Jumlah Node dikunjungi: 10
Path: crawler -> cradler -> cradled -> craaled -> crawled -> brawled -> brailed -> braised -> bruised -> bruiser -> cruiser
Time: 0.029755 detik
```

```
Algoritma A star
======================JAWABAN======================
Jumlah Node dikunjungi: 41
Path: crawler -> brawler -> brawled -> brailed -> braised -> bruised -> bruiser -> cruiser
Time: 0.0368589 detik
```

5.1.5 Uji Program 5

Hasil uji kata yang berbeda jauh

```
Masukkan kata awal: skate

Masukkan kata tujuan: board

Algoritma yang bisa digunakan:
1. UCS
2. GBFS
3. A*
Pilih nomor algoritma:
```

```
Algoritma UCS
======================JAWABAN======================
Jumlah Node dikunjungi: 527
Path: skate -> skats -> scats -> scars -> soars -> boars -> board
Time: 0.1012856 detik
```

```
Algoritma GBFS
======================JAWABAN======================
Jalur tidak ditemukan
Jumlah Node dikunjungi: 3
Time: 0.009109 detik
```

```
Algoritma A star
====================JAWABAN====================
Jumlah Node dikunjungi: 503
Path: skate -> skats -> scats -> scars -> soars -> boars -> board
Time: 0.1002983 detik
```

5.1.6 Uji Program 6

Hasil Uji menunjukan GBFS tidak optimal

```
Masukkan kata awal: bucket

Masukkan kata tujuan: rocket

Algoritma yang bisa digunakan:
1. UCS
2. GBFS
3. A*
Pilih nomor algoritma:
```

```
Algoritma UCS
======================JAWABAN======================
Jumlah Node dikunjungi: 86
Path: bucket -> bucked -> rucked -> rocked -> rocket
Time: 0.0432363 detik
```

```
Algoritma GBFS
======================JAWABAN======================
Jumlah Node dikunjungi: 5
Path: bucket -> becket -> becked -> recked -> rocked -> rocket
Time: 0.01511 detik
```

```
Algoritma A star
======================JAWABAN======================
Jumlah Node dikunjungi: 31
Path: bucket -> tucket -> tacket -> racket -> rocket
Time: 0.0272718 detik
```

5.1.7 Uji Program 7

Hasil Uji dengan kata yang mirip

```
Masukkan kata awal: gadget

Masukkan kata tujuan: widget

Algoritma yang bisa digunakan:
1. UCS
2. GBFS
3. A*
Pilih nomor algoritma:
```

```
Algoritma UCS
======================JAWABAN======================
Jumlah Node dikunjungi: 2330
Path: gadget -> garget -> gorget -> forget -> forged -> fogged -> figged -> fidged -> fidget -> widget
Time: 0.8201498 detik
```

```
Algoritma GBFS
======================JAWABAN======================
Jalur tidak ditemukan
Jumlah Node dikunjungi: 6
Time: 0.0468143 detik
PS C:\Users\Keanu\IdeaProjects\Tucil3_13522082\src>
```

```
Algoritma A star
======================JAWABAN======================
Jumlah Node dikunjungi: 1675
Path: gadget -> garget -> gorget -> forget -> forged -> fogged -> figged -> fidged -> fidget -> widget
Time: 0.5866322 detik
```

5.1.8 Uji Program 8

Hasil uji kasus jalur tidak ditemukan

```
Masukkan kata awal: papers

Masukkan kata tujuan: random

Algoritma yang bisa digunakan:
1. UCS
2. GBFS
3. A*
Pilih nomor algoritma:
```

Hasil:

```
Algoritma UCS
======================JAWABAN======================
Jalur tidak ditemukan
Jumlah Node dikunjungi: 8403
Time: 3.4644991 detik
```

```
Algoritma GBFS
======================JAWABAN=======================
Jalur tidak ditemukan
Jumlah Node dikunjungi: 64
Time: 0.0792465 detik
```

```
Algoritma A star
======================JAWABAN=======================
Jalur tidak ditemukan
Jumlah Node dikunjungi: 8403
Time: 3.3721048 detik
```

5.2 Analisis Algoritma

Berdasarkan implementasi program yang penulis buat, dimana jika child dari suatu node merupakan kata tujuan, maka program akan berhenti untuk optimalisasi. Selain itu juga dilakukan optimalisasi dengan menambahkan child ke visited (khusus untuk algoritma UCS dan A*) sehingga child tidak akan di generate berulang-ulang kali. Dengan demikian, jumlah node yang dikunjungi akan menjadi sedikit.

Pada uji program 2, ketiga algoritma tentunya tidak menghasilkan perbedaan karena hanya menampilkan 1 path (start dan end sama) Untuk uji program yang normal seperti uji program ke 3, A* mengeksekusi dengan waktu tercepat dan UCS memiliki waktu terlama. Sampai sini masih belum ada perbedaan.

Pada uji program ke 5 dihasilkan bahwa algoritma GBFS tidak menemukan jalur. Hal ini dikarenakan algoritma GBFS tidak dapat backtracking sehingga apabila ia sudah melakukan pembangkitan ke suatu node, maka akan terus menelusurinya sampai node tersebut tidak punya tetangga lagi. Ini menyebabkan algoritma GBFS kebanyakan akan menghasilkan “jalur tidak ditemukan”. Pada kasus ini GBFS terjebak pada minimum lokal.

Pengujian ke 4 dan 6 menunjukan bahwa GBFS menghasilkan jalur yang tidak optimal / lebih panjang dari UCS ataupun A*. Namun GBFS memiliki waktu eksekusi tercepat dan pengecekan node paling sedikit yang berarti algoritma ini membutuhkan memory yang kecil dibanding UCS dan A*. Seperti halnya algoritma greedy, Algoritma ini berusaha mencari waktu yang tercepat karena tidak backtracking dan mengekspan node dengan cost heuristic terkecil. Cost heuristik terkecil berarti sebuah kata semakin dekat dengan kata tujuan. GBFS akan melanjutkan jalannya sampai akhir bila sudah ekspansi ke suatu node, sehingga algoritma ini menghasilkan jalur yang panjang atau tidak optimal. Namun pada word ladder, kemungkinan besarnya akan terjebak dalam minimum lokal.

Pengujian 7 terlihat perbedaan yang signifikan antara UCS dan A*. Dimana A* memiliki waktu eksekusi yang lebih cepat. Pada dasarnya, A* dan UCS sama sama untuk mencari solusi yang optimal. Pada word ladder, UCS memiliki kemiripan dengan BFS karena cost tiap mengekspan g(n) pasti selalu bertambah 1, sehingga ini akan merepresentasikan kedalaman di BFS. Algoritma A* akan memodifikasi UCS dengan menambah kan cost heuristik atau biaya node n ke node akhir. Dengan memodifikasi rumus menjadi g(n) + h(n),

algoritma A* akan mempercepat pencarian jalur yang optimal. Heuristiknya dapat membantu mempersempit ruang pencarian. A* dapat mengurangi jumlah node yang harus diperiksa, sehingga lebih cepat daripada UCS. Ini menunjukan juga bahwa memori yang dibutuhkan pada A8 lebih sedikit dibandingkan di UCS.

Pada kasus pengujian ke 8 / pengujian jalur yang tidak ditemukan, didapatkan bahwa GBFS mendapatkan hasil eksekusi yang cepat dengan pengecekan node yang sedikit. Hal ini dikarenakan apabila memang tidak ada jalur maka GBFS pasti terjebak pada minimum lokal yang sudah pasti waktu eksekusinya akan cepat.

BAB V
KESIMPULAN DAN SARAN

6.1 Kesimpulan

Algoritma UCS mirip dengan algoritma BFS dengan cost g(n) untuk ekspan merepresentasikan kedalaman di BFS. dengan ini UCS akan mendapatkan hasil yang optimal, namun tidak memikirkan waktu eksekusi

Algoritma GBFS akan cenderung memikirkan waktu eksekusi dan pengecekan node sedikit dengan membangkitakan node cost terendah berdasarkan h(n) / heuristik nya. Namun sayangnya algoritma ini tidak dapat backtracing sehingga memiliki kemungkinan untuk terjebak dalam minimum lokal. Apabila jalur ditemukan maka jalur tersebut belum tentu jalur yang optimal

Algoritma A* akan mengekspan node dengan cost rendah menurut rumus g(n)+h(n). Algoritma ini memodifikasi UCS agar dapat menghasilkan waktu eksekusi yang lebih cepat namun tetap menghasilkan jalur yang optimal.

Optimalisasi pada visited di algoritma UCS dan A* terbukti sangat berpengaruh dalam mempercepat waktu eksekusi program.

6.2 Saran

Tugas Kecil IF2211 Strategi Algoritma Semester II Tahun 2023/2024 menjadi salah satu tugas menarik bagi penulis. Berdasarkan pengalaman penulis mengerjakan tugas ini, berikut merupakan saran untuk pembaca yang ingin melakukan atau mengerjakan hal yang serupa.

1. Tugas ini sangat bergantung pada kamus yang dipakai, sehingga ada baiknya bila kamus yang dipakai diseragamkan agar proses pengerjaan dan testing dapat berjalan dengan baik.

DAFTAR PUSTAKA

Munir, R. (2022). Penentuan Rute (Route/Path Planning) (1). Retrieved from Homepage Rinaldi Munir: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf

Munir, R. (2022).Penentuan Rute (Route/Path Planning) (1). Retrieved from Homepage Rinaldi Munir: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf

LAMPIRAN

Pranala

Kode program dapat diakses pada link berikut

https://github.com/keanugonza/Tucil3_13522082.git

How to Use

Untuk menjalankan program:

1. cd src
2. javac Main.java
3. java Main

Untuk memulai permainan

1. Masukan kata start apapun, program sudah dapat memvalidasi
2. Masukan kata end apapun, program sudah dapat memvalidasi
3. Pilih kategori dari 1-3
4. Program akan menampilkan hasil
5. Program exit

Tabel Poin

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