LAPORAN TUGAS KECIL 3
Penyelesaian Permainan Word Ladder Menggunakan Algoritma UCS, Greedy Best First Search, dan A*

Disusun oleh:
13522152 - Muhammad Roihan

PROGRAM STUDI TEKNIK INFORMATIKA

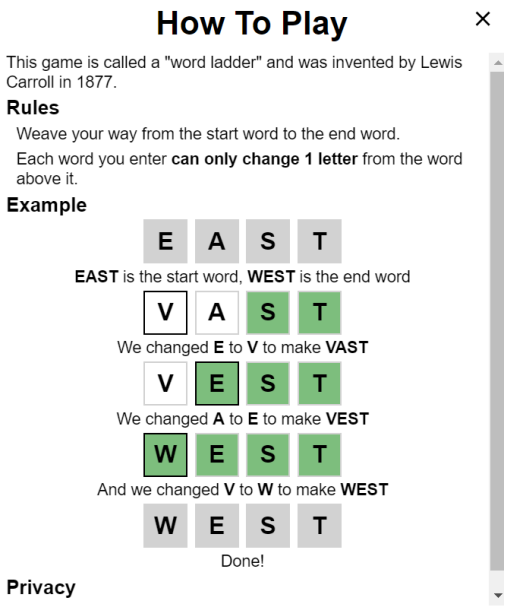
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2023/2024

BAB 1
DESKRIPSI MASALAH

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.



Gambar 1. Ilustrasi dan Peraturan Permainan Word Ladder
(Sumber: https://wordwormdormdork.com/)

BAB 2
ANALISIS DAN IMPLEMENTASI

2.1. Analisis *Uniform Cost Search* (*UCS*)

Algoritma Uniform Cost Search (UCS) merupakan pendekatan yang kuat dalam pencarian jalur terpendek dalam graf berbobot. Dengan mempertimbangkan biaya atau bobot yang terkait dengan setiap langkah, UCS mampu menemukan solusi optimal yang meminimalkan total biaya dari simpul awal ke simpul tujuan. Kelebihan utama dari UCS adalah optimalitasnya yang menjamin solusi terbaik dan kemampuannya menangani graf dengan bobot yang beragam. Namun, UCS memiliki beberapa kelemahan, seperti kompleksitas waktu yang tinggi terutama pada graf besar dan kebutuhan ruang yang besar untuk penyimpanan simpul yang dieksplorasi. Analisis kompleksitas waktu dan ruang UCS membantu dalam memahami kinerjanya dalam berbagai situasi. Oleh karena itu, UCS direkomendasikan ketika solusi optimal diperlukan dan bobot langkah perlu dipertimbangkan, terutama pada graf yang tidak terlalu besar. Namun, untuk masalah dengan graf yang sangat besar atau biaya langkah yang seragam, pendekatan lain seperti GBFS atau ASTAR mungkin lebih sesuai.

Dalam kasus permainan word ladder, dimana setiap langkah memiliki biaya yang sama, UCS akan berperilaku sama dengan BFS. Ini karena UCS akan mempertimbangkan biaya aktual dari simpul awal ke simpul yang sedang dieksplorasi, dan karena biaya setiap langkah dalam permainan word ladder adalah sama, UCS akan memilih simpul berikutnya berdasarkan jarak minimum dari simpul awal.

Pada program yang dibuat, algoritma UCS bekerja dengan mengeksplorasi simpul-simpul secara berurutan, dimulai dari simpul awal, dan mempertimbangkan biaya langkah saat memilih simpul berikutnya untuk dieksplorasi. Pada program ini, sebuah PriorityQueue digunakan untuk menyimpan simpul-simpul yang akan dieksplorasi selanjutnya, dengan simpul yang memiliki biaya terendah memiliki prioritas tertinggi dalam antrian. Selama proses pencarian, simpul-simpul dieksplorasi secara berulang dengan memperbarui biaya dan jalur terpendek ke setiap simpul. Program ini juga mencatat simpul-simpul yang telah dieksplorasi dalam daftar visited, sehingga mencegah pengulangan simpul yang sama. Jika simpul tujuan ditemukan, program akan menghasilkan jalur terpendek yang ditemukan. Namun, jika jalur tidak ditemukan, program akan mencetak pesan "Path Not Found". Dengan demikian, program tersebut memberikan solusi untuk mencari jalur terpendek dalam permainan *word ladder* menggunakan algoritma UCS.

2.2. Analisis *Greedy Best First Search* (*GBFS*)

Algoritma Greedy Best First Search (GBFS) adalah pendekatan pencarian yang mengutamakan heuristik lokal terbaik saat memilih simpul berikutnya untuk dieksplorasi. Dengan mempertimbangkan nilai heuristik terendah atau paling dekat dengan simpul tujuan pada setiap langkah, GBFS cenderung menuju simpul yang secara kasar dianggap sebagai solusi terbaik. Kelebihan utama GBFS adalah efisiensinya dalam masalah heuristik, khususnya dalam

mencari jarak terpendek dalam graf besar, serta kompleksitas ruang yang rendah karena hanya menyimpan simpul yang telah dieksplorasi. Namun, GBFS tidak menjamin solusi optimal dan rentan terjebak dalam minimum lokal. Kualitas solusi GBFS sangat tergantung pada kualitas fungsi heuristik, dan algoritma ini dapat mengalami masalah siklus dalam graf yang tidak diarahkan. Analisis kompleksitas waktu dan ruang GBFS dapat membantu dalam memahami kinerjanya, namun keputusan penggunaan algoritma ini tergantung pada kualitas heuristik, efisiensi ruang, dan kebutuhan akan solusi optimal dalam masalah yang dihadapi.

Secara teoritis, algoritma Greedy Best First Search tidak menjamin solusi optimal untuk persoalan word ladder karena algoritma ini cenderung membuat keputusan berdasarkan heuristik yang mengarahkan pencarian ke simpul yang paling dekat dengan tujuan, tanpa mempertimbangkan biaya total yang sudah ditempuh. Oleh karena itu, meskipun Greedy Best First Search dapat menghasilkan solusi dengan cepat, solusi yang diberikan tidak selalu optimal.

Pada program yang dibuat, sebuah Queue digunakan untuk menyimpan simpul-simpul yang akan dieksplorasi selanjutnya. Selama proses pencarian, simpul-simpul dieksplorasi secara berurutan dari antrian. Ketika simpul dieksplorasi, simpul tersebut ditandai sebagai sudah dikunjungi dan dicatat dalam daftar visited. Jika simpul tujuan ditemukan, program akan menghasilkan jalur terpendek yang ditemukan. Namun, jika jalur tidak ditemukan, program akan mencetak pesan "Path Not Found". Algoritma ini memiliki keunggulan dalam kecepatan pencarian karena hanya mempertimbangkan informasi lokal terbaik, namun, kelemahannya adalah tidak menjamin solusi optimal dan rentan terjebak dalam minimum lokal. Dengan demikian, program tersebut memberikan solusi untuk mencari jalur terpendek dalam permainan *word ladder* menggunakan algoritma GBFS.

2.3. Analisis A*

Algoritma A* merupakan pendekatan pencarian jalur terpendek yang memanfaatkan informasi heuristik untuk mengestimasi biaya tersisa dari simpul saat ini ke simpul tujuan. Kelebihan utama A* adalah kemampuannya untuk menemukan solusi optimal dengan meminimalkan biaya total, asalkan fungsi heuristiknya konsisten. Algoritma ini juga lebih efisien dibandingkan dengan pencarian blind seperti BFS atau DFS karena menggunakan informasi heuristik untuk memandu pencarian. Namun, kualitas solusi A* sangat bergantung pada kualitas fungsi heuristik yang digunakan, sehingga fungsi heuristik yang buruk dapat menghasilkan solusi yang tidak optimal atau kinerja yang buruk. Kompleksitas waktu A* juga dapat menjadi tinggi terutama pada graf yang besar atau ketika fungsi heuristik memiliki kompleksitas tinggi. Meskipun demikian, A* cocok digunakan ketika solusi optimal diperlukan dan informasi heuristik tersedia atau dapat diperkirakan dengan baik, terutama dalam menemukan jalur terpendek dalam graf yang besar. Namun, jika fungsi heuristik yang baik tidak tersedia atau kompleksitas waktu menjadi kendala, algoritma lain seperti GBFS atau UCS mungkin lebih cocok untuk digunakan. Dengan mempertimbangkan kelebihan, kekurangan, serta kompleksitas waktu dan ruang A*, kita dapat menentukan apakah algoritma ini sesuai untuk masalah

pencarian yang dihadapi, dengan memperhatikan kualitas solusi yang diinginkan dan ketersediaan informasi heuristik.

Heuristik yang digunakan dalam algoritma A* untuk permainan word ladder haruslah admissible, yaitu memberikan estimasi yang tidak melebihi biaya sebenarnya untuk mencapai simpul tujuan. Dalam konteks permainan word ladder, heuristik umumnya adalah jumlah perbedaan karakter antara kata saat ini dengan kata tujuan. Heuristik ini memperkirakan jumlah langkah minimum yang dibutuhkan untuk mencapai kata tujuan dari kata saat ini dengan cara menghitung berapa banyak karakter yang harus diubah. Karena setiap langkah dalam permainan word ladder melibatkan perubahan satu karakter, nilai heuristik ini tidak akan pernah melebihi biaya sebenarnya untuk mencapai kata tujuan. Selain itu, heuristik ini juga konsisten, karena setiap langkah yang diambil akan selalu mengurangi jumlah perbedaan karakter antara kata saat ini dan kata tujuan. Oleh karena itu, heuristik jumlah perbedaan karakter antara kata saat ini dengan kata tujuan adalah admissible dalam algoritma A* untuk permainan word ladder, memenuhi kriteria keadmissibility dan konsistensi yang diperlukan dalam penggunaan heuristik pada algoritma A*.

Secara teoritis, dalam kasus word ladder, algoritma A* cenderung lebih efisien dibandingkan dengan algoritma Uniform Cost Search (UCS). Algoritma A* menggunakan heuristik untuk memperkirakan biaya yang tersisa atau jarak yang tersisa untuk mencapai tujuan dari setiap simpul yang sedang dieksplorasi. Dengan menggunakan heuristik yang baik, A* bisa mengarahkan pencarian ke arah yang benar lebih cepat daripada UCS, karena A* cenderung mengutamakan simpul yang memiliki estimasi biaya yang lebih rendah. Sementara itu, algoritma UCS (Uniform Cost Search) tidak menggunakan heuristik dan hanya mempertimbangkan biaya aktual dari simpul saat ini ke simpul-simpul tetangganya. Ini berarti UCS akan memeriksa semua kemungkinan langkah tanpa memiliki wawasan tentang arah yang benar menuju tujuan.

Pada program yang dibuat, sebuah PriorityQueue digunakan untuk menyimpan simpul-simpul yang akan dieksplorasi selanjutnya, dengan simpul yang memiliki biaya terendah memiliki prioritas tertinggi dalam antrian. Selama proses pencarian, simpul-simpul dieksplorasi secara berulang dari antrian. Ketika simpul dieksplorasi, simpul tersebut ditandai sebagai sudah dikunjungi dan dicatat dalam daftar visited. Jika simpul tujuan ditemukan, program akan menghasilkan jalur terpendek yang ditemukan. Namun, jika jalur tidak ditemukan, program akan mencetak pesan "Path Not Found". Algoritma ini memiliki keunggulan dalam menemukan solusi optimal jika fungsi heuristiknya konsisten, namun, kompleksitas waktu dan ruangnya dapat menjadi tinggi terutama pada graf yang besar. Dengan demikian, program tersebut memberikan solusi untuk mencari jalur terpendek dalam permainan *word ladder* menggunakan algoritma A*.

2.4. Implementasi Algoritma

Untuk mempermudah implementasi ketiga algoritma pada kasus ini, pertama - tama akan dibuat sebuah file bernama graph.txt. File ini digunakan untuk mendefinisikan graf yang terbentuk pada word.txt.

Secara garis besar ketiga algoritma ini memiliki implementasi yang hampir serupa**,**

Perbedaan paling mencolok dari ketiga algoritma ini adalah perbedaannya adalah perhitungan cost suatu simpul, Algoritma UCS menggunakan fungsi g(n), algoritma GBFS menggunakan fungsi f(n) = h(n), dan algoritma A* menggunakan fungsi f(n) = g(n) + h(n) = g(n) + h*(n).

Berikut merupakan langkah implementasi algoritma dalam kasus ini :

1. Pada awalnya terdefinisi sebuah *queue* untuk GBFS dan *priority queue* untuk UCS dan A*. *Priority queue* tersebut menggunakan komparator yang membuat simpul yang memiliki jarak terdekat dari simpul awal memiliki prioritas tertinggi untuk UCS dan simpul yang memiliki nilai terkecil dari penjumlahan jarak ke simpul awal dan jarak ke simpul tujuan memiliki prioritas tertinggi untuk A*.
2. Masukkan kata awal yang dinyatakan dalam sebuah simpul ke dalam *queue.*
3. Lakukan *dequeue* pada *queue*.
4. Periksa apakah kata yang di *dequeue* tersebut merupakan kata tujuan atau bukan. Jika kata tersebut merupakan kata tujuan tandai path telah ditemukan.
5. Untuk algoritma GBFS periksa seluruh child, untuk mencari *child* yang terdekat dengan tujuan. Untuk algoritma UCS dan A* periksa seluruh child.
6. Untuk algoritma GBFS periksa apakah *child* tersebut sudah dikunjungi atau belum. Jika belum tambahkan ke *queue*. Untuk algoritma UCS dan A* periksa *child* - *child* tersebut, jika belum dikunjungi dan tidak ada di *queue* tambahkan ke *queue*. Jika ada di *queue* namun simpul yang ada di queue memiliki biaya yang lebih mahal dibanding yang diperiksa, hapus simpul yang di *queue* dan tambahkan simpul yang diperiksa ke *queue*.
7. Ulangi langkah 2 - 5 selama *queue* tidak kosong dan *path* belum ditemukan.

BAB 3
SOURCE CODE

3.1 Algo.java

```
public class Algo {
    protected int nodeCnt = 0; // Buat nyimpen jumlah node yang dikunjungi

    public boolean cekVisited(List<Node> visited, Node nodeCek) { // Fungsi
buat ngecek node udah dikunjungi atau belum
        for (Node n : visited) {
            if (n.getValue() == nodeCek.getValue()) {
                return true;
            }
        }
        return false;

    }

    public boolean inQ(PriorityQueue<Node> pq, Node nodeCek) { // Fungsi buat
ngecek node ada didalam queue atau nggak
        PriorityQueue<Node> copyPq = new PriorityQueue<>(pq);
        while (!copyPq.isEmpty()) {
            Node node = copyPq.poll();
            if (node.getValue() == nodeCek.getValue()) {
                return true;
            }
        }
        return false;
    }

    public void delFromQ(PriorityQueue<Node> pq, Node nodeCek) { // Fungsi buat
ngehapus node dari queue
        while (!pq.isEmpty()) {
            Node node = pq.poll();
            if (node.getValue() == nodeCek.getValue()) {
                pq.remove(node);
            }
        }
    }

    public Integer costFromGoal(String word1, String word2) { // Fungsi buat
ngehitung perbedaan huruf
        int diff = 0;
        word1 = word1.toLowerCase();
```

```
        word2 = word2.toLowerCase();

        for (int i = 0; i < word1.length(); i++) {
            if (word1.charAt(i) != word2.charAt(i)) {
                diff++;
            }
        }
        return diff;
    }

    public Node findShortest(List<Node> childs,Node Goal) { // Fungsi buat cari
node paling deket dari goal
        Node shortest = new Node("kosong");
        for (Node n : childs) {
            if (shortest.getValue().equals("kosong")) {
                shortest = n;
            } else if (costFromGoal(n.getValue(), Goal.getValue()) <
costFromGoal(shortest.getValue(),
                    Goal.getValue())) {
                shortest = n;
            }
        }
        return shortest;
    }

    public void printPath(Node target) { // Fungsi buat cetak path
        List<Node> path = new ArrayList<Node>();
        for (Node node = target; node != null; node = node.getParent()) {
            path.add(node);
            // System.out.println(node.getValue());
        }

        Collections.reverse(path);
        System.out.println("Path :");
        int cost = -1;
        for (Node n : path) {
            System.out.print(n.getValue() + " ");
            cost++;
        }
        System.out.println();
        System.out.println("Cost :" + cost);
        System.out.println("Jumlah node dikunjungi :"+nodeCnt);

    }
}
```

2. UCS.java

```
public class UCS extends Algo {
    public void solveUCS(Node start, Node end, Map<String, List<String>>
wordMap) {
        // int pathCost = 0;
        PriorityQueue<Node> pq = new PriorityQueue<Node>(
                new Comparator<Node>() {
                    public int compare(Node i, Node j) {
                        if (i.getCost() > j.getCost()) {
                            return 1;                           // Comperator yg
ngebuat node yang memiliki cost terendah memiliki prioritas tertinggi
                        } else if (i.getCost() < j.getCost()) {
                            return -1;
                        } else {
                            return 0;
                        }
                    }
                });

        List<Node> visited = new ArrayList<Node>();
        boolean found = false;

        pq.add(start);

        while (!pq.isEmpty() && (found == false)) {
            nodeCnt++;
            // printPq(pq);
            Node curr = pq.poll();
            visited.add(curr);

            if (curr.getValue().equals(end.getValue())) {
                end = curr;
                found = true;
            }

            for (Node n : curr.getChilds()) {

                // Kalo node belum dilihat dan gak ada di queue, tambahin node
ke queue
                if (!cekVisited(visited, n) && !inQ(pq, n)) {
                    // System.out.println("currCost : " + curr.getCost() + "
word " + n.getValue());
                    n.setCost(curr.getCost() + 1);
```

```
                    n.setParent(curr);
                    List<String> child = wordMap.get(n.getValue());
                    n.setChild(child);
                    pq.add(n);

                }

                // Kalo node ada di queue tapi cost node yang ada di queue
lebih besar dibanding yang dicek sekarang, tambahin node ke queue
                else if (inQ(pq, n) && ((curr.getCost() + 1) < n.getCost())) {
                    System.out.println("currCost : " + curr.getCost() + "
previousCost " + n.getCost());
                    n.setCost(curr.getCost() + 1);
                    n.setParent(curr);
                    List<String> child = wordMap.get(n.getValue());
                    n.setChild(child);
                    delFromQ(pq, n);
                    pq.add(n);
                }
            }
        }
        if (found) {
            printPath(end);
        }
        else {
            System.out.println("Path Not Found");
        }
    }

}
```

3. GBFS.java

```
public class GBFS extends Algo{
    public void solveGBFS(Node start, Node end, Map<String, List<String>>
wordMap) {
        Queue<Node> q = new LinkedList<Node>();
        List<Node> visited = new ArrayList<Node>();
        boolean found = false;

        q.add(start);

        while (!q.isEmpty() && (found == false)) {
            nodeCnt++;
```

```
            Node curr = q.poll();
            visited.add(curr);

            if (curr.getValue().equals(end.getValue())) {
                end = curr;
                found = true;
            }

            Node nearest = findShortest(curr.getChilds(), end);

            // Kalo node belum dilihat dan gak ada di queue, tambahin node ke
queue
            if (!cekVisited(visited, nearest)) {
                nearest.setParent(curr);
                List<String> child = wordMap.get(nearest.getValue());
                nearest.setChild(child);
                q.add(nearest);
            }
        }
        if (found) {
            printPath(end);
        }
        else {
            System.out.println("Path Not Found");
        }
    }
}
```

4. AS.java

```
public class AS extends Algo{
    public void solveAS(Node start, Node end, Map<String, List<String>>
wordMap) {
        // int pathCost = 0;
        PriorityQueue<Node> pq = new PriorityQueue<Node>(
                new Comparator<Node>() {
                    public int compare(Node i, Node j) {
                        if (i.getCost() > j.getCost()) {
                            return 1;                           // Comperator
yg ngebuat node yang memiliki cost terendah memiliki prioritas tertinggi
                        } else if (i.getCost() < j.getCost()) {
                            return -1;
```

```
                        } else {
                            return 0;
                        }
                    }
                });

        List<Node> visited = new ArrayList<Node>();
        boolean found = false;

        pq.add(start);

        while (!pq.isEmpty() && (found == false)) {
            // printPq(pq);
            nodeCnt++;
            Node curr = pq.poll();
            visited.add(curr);

            if (curr.getValue().equals(end.getValue())) {
                end = curr;
                found = true;
            }

            for (Node n : curr.getChilds()) {

                // Kalo node belum dilihat dan gak ada di queue, tambahin node
ke queue
                if (!cekVisited(visited, n) && !inQ(pq, n)) {
                    // System.out.println("currCost : " + curr.getCost() + "
word " + n.getValue());
                    n.setCost(curr.getCost() + 1 + costFromGoal(n.getValue(),
end.getValue()));
                    n.setParent(curr);
                    List<String> child = wordMap.get(n.getValue());
                    n.setChild(child);
                    pq.add(n);

                }

                // Kalo node ada di queue tapi cost node yang ada di queue
lebih besar dibanding yang dicek sekarang, tambahin node ke queue
                else if (inQ(pq, n) && ((curr.getCost() + 1 +
costFromGoal(n.getValue(), end.getValue())) < n.getCost())) {
                    // System.out.println("currCost : " + curr.getCost() + "
previousCost " + n.getCost());
                    n.setCost(curr.getCost() + 1 + costFromGoal(n.getValue(),
```

```
end.getValue()));
                    n.setParent(curr);
                    List<String> child = wordMap.get(n.getValue());
                    n.setChild(child);
                    delFromQ(pq, n);
                    pq.add(n);
                }
            }
        }
        if (found) {
            printPath(end);
        }
        else {
            System.out.println("Path Not Found");
        }
    }
```

BAB 4
UJI COBA

4.1 Test Case 1

```
Masukkan Startword:earn
Start Word: earn
Masukkan Endword:Make
End Word: Make
Load graph....
Pilih Algoritma:
1. UCS
2. Greedy Best First Search
3. A*
1
Finding Shortest Path....
Path :
earn barn bare bake make
Cost :4
Jumlah node dikunjungi :1873
Execution time: 3928 ms
```

```
Masukkan Startword:EarN
Start Word: EarN
Masukkan Endword:make
End Word: make
Load graph....
Pilih Algoritma:
1. UCS
2. Greedy Best First Search
3. A*
2
Finding Shortest Path....
Path :
earn barn bare bake make
Cost :4
Jumlah node dikunjungi :5
Execution time: 9 ms
```

```
Masukkan Startword:earn
Start Word: earn
Masukkan Endword:make
End Word: make
Load graph....
Pilih Algoritma:
1. UCS
2. Greedy Best First Search
3. A*
3
Finding Shortest Path....
Path :
earn carn care cake make
Cost :4
Jumlah node dikunjungi :155
Execution time: 119 ms
```

4.2 Test Case 2

```
Masukkan Startword:rose
Start Word: rose
Masukkan Endword:gold
End Word: gold
Load graph....
Pilih Algoritma:
1. UCS
2. Greedy Best First Search
3. A*
1
Finding Shortest Path....
Path :
rose hose hole hold gold
Cost :4
Jumlah node dikunjungi :4583
Execution time: 25714 ms
```

```
Masukkan Startword:rose
Start Word: rose
Masukkan Endword:gold
End Word: gold
Load graph....
Pilih Algoritma:
1. UCS
2. Greedy Best First Search
3. A*
2
Finding Shortest Path....
Path :
rose role bole bold gold
Cost :4
Jumlah node dikunjungi :5
Execution time: 13 ms
```

```
Masukkan Startword:rose
Start Word: rose
Masukkan Endword:gold
End Word: gold
Load graph....
Pilih Algoritma:
1. UCS
2. Greedy Best First Search
3. A*
3
Finding Shortest Path....
Path :
rose role cole cold gold
Cost :4
Jumlah node dikunjungi :205
Execution time: 305 ms
```

4.3 Test Case 3

```
Masukkan Startword:tram
Start Word: tram
Masukkan Endword:ride
End Word: ride
Load graph....
Pilih Algoritma:
1. UCS
2. Greedy Best First Search
3. A*
1
Finding Shortest Path....
Path :
tram team ream read redd rede ride
Cost :6
Jumlah node dikunjungi :14280
Execution time: 113894 ms
```

```
Masukkan Startword:tram
Start Word: tram
Masukkan Endword:ride
End Word: ride
Load graph....
Pilih Algoritma:
1. UCS
2. Greedy Best First Search
3. A*
2
Finding Shortest Path....
Path Not Found
Execution time: 1 ms
```

```
Masukkan Startword:tram
Start Word: tram
Masukkan Endword:ride
End Word: ride
Load graph....
Pilih Algoritma:
1. UCS
2. Greedy Best First Search
3. A*
3
Finding Shortest Path....
Path :
tram team ream read redd rede ride
Cost :6
Jumlah node dikunjungi :702
Execution time: 520 ms
```

4.4 Test Case 4

```
Masukkan Startword:flown
Start Word: flown
Masukkan Endword:smile
End Word: smile
Load graph....
Pilih Algoritma:
1. UCS
2. Greedy Best First Search
3. A*
1
Finding Shortest Path....
Path :
flown flows slows slots spots spits spite spile smile
Cost :8
Jumlah node dikunjungi :11568
Execution time: 24023 ms
```

```
Masukkan Startword:flown
Start Word: flown
Masukkan Endword:smile
End Word: smile
Load graph....
Pilih Algoritma:
1. UCS
2. Greedy Best First Search
3. A*
2
Finding Shortest Path....
Path Not Found
Execution time: 0 ms
```

```
Masukkan Startword:flown
Start Word: flown
Masukkan Endword:smile
End Word: smile
Load graph....
Pilih Algoritma:
1. UCS
2. Greedy Best First Search
3. A*
3
Finding Shortest Path....
Path :
flown flows slows slops slips slipe stipe stile smile
Cost :8
Jumlah node dikunjungi :1411
Execution time: 1002 ms
```

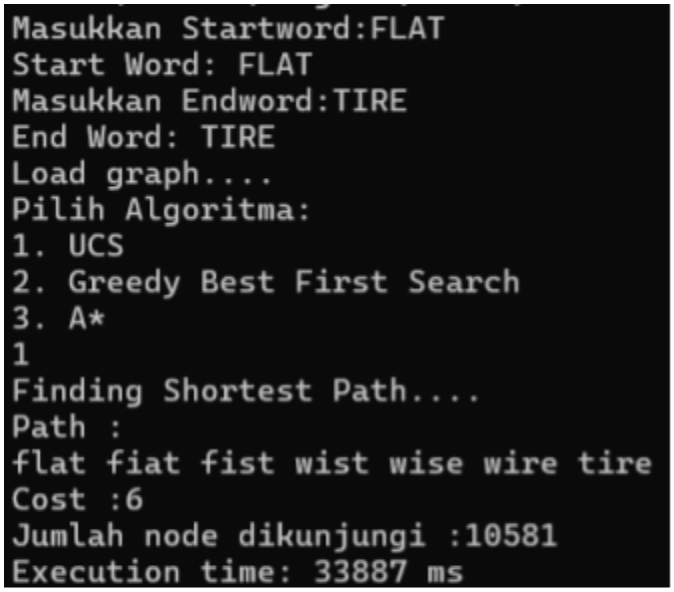
4.5 Test Case 5

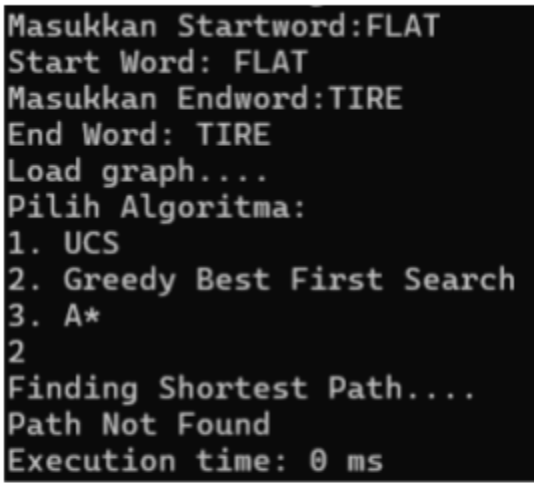
```
Masukkan Startword:hat
Start Word: hat
Masukkan Endword:bat
End Word: bat
Load graph....
Pilih Algoritma:
1. UCS
2. Greedy Best First Search
3. A*
1
Finding Shortest Path....
Path :
hat bat
Cost :1
Jumlah node dikunjungi :2
Execution time: 19 ms
```

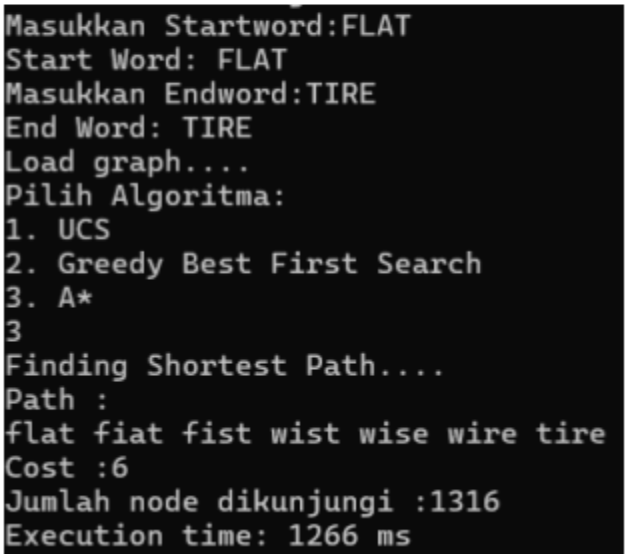
```
Masukkan Startword:hat
Start Word: hat
Masukkan Endword:bat
End Word: bat
Load graph....
Pilih Algoritma:
1. UCS
2. Greedy Best First Search
3. A*
2
Finding Shortest Path....
Path :
hat bat
Cost :1
Jumlah node dikunjungi :2
Execution time: 9 ms
```

```
Masukkan Startword:hat
Start Word: hat
Masukkan Endword:bat
End Word: bat
Load graph....
Pilih Algoritma:
1. UCS
2. Greedy Best First Search
3. A*
3
Finding Shortest Path....
Path :
hat bat
Cost :1
Jumlah node dikunjungi :2
Execution time: 10 ms
```

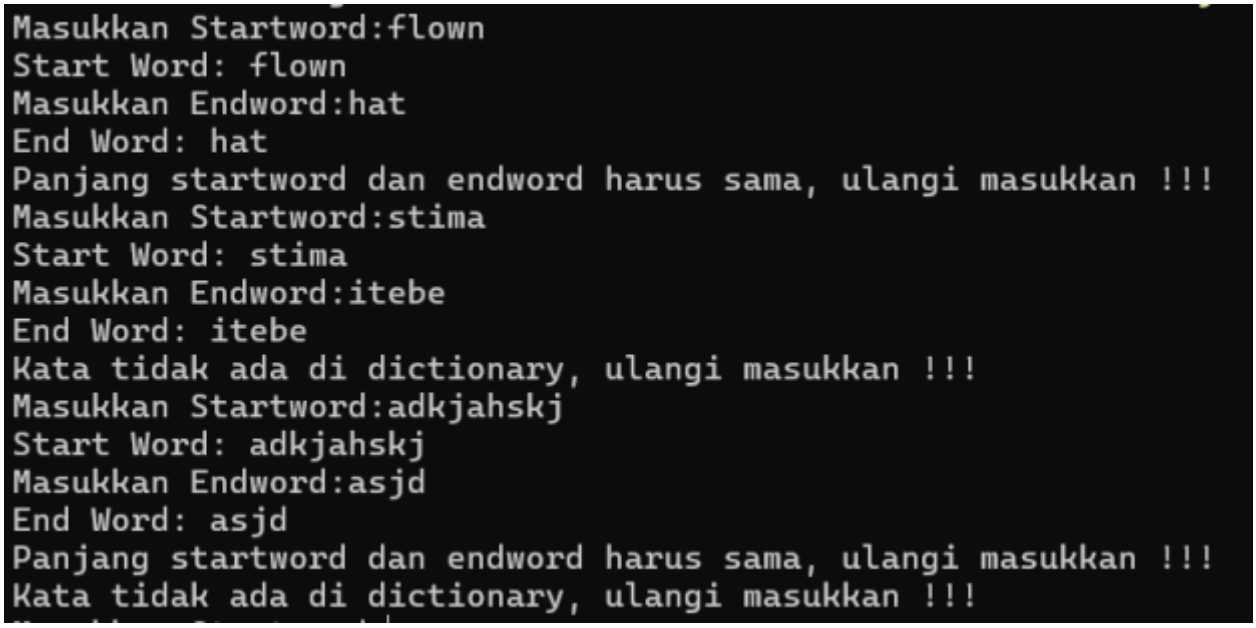
4.6 Test Case 6







4.7 Test Case Input Error



BAB 5
ANALISIS HASIL

5.1 Analisis Hasil Uji Coba

Berdasarkan hasil uji coba, algoritma UCS dan A* selalu memberikan solusi yang optimal, sementara algoritma GBFS terkadang gagal menemukan solusi dalam beberapa kasus uji. Hal ini dapat dijelaskan oleh prinsip kerja masing-masing algoritma. UCS, yang mirip dengan BFS, bekerja dengan mempertimbangkan biaya langkah antar simpul yang seragam, sehingga selalu menghasilkan solusi yang optimal dalam kasus graf dengan bobot yang seragam. Sementara itu, algoritma A* sudah dijelaskan pada bab 2 bahwa algoritma tersebut admissible sehingga hasil yang diberikan pasti optimal.

Dalam kasus uji di mana algoritma GBFS gagal, kemungkinan besar disebabkan oleh sifat algoritma tersebut yang rentan terhadap minimum lokal. GBFS memilih simpul berikutnya berdasarkan nilai heuristiknya tanpa mempertimbangkan biaya aktual, sehingga bisa saja terjebak dalam jalur yang tidak optimal. Hal ini menjadi masalah terutama jika heuristik yang digunakan tidak menggambarkan jarak yang sebenarnya ke simpul tujuan. Oleh karena itu, walaupun GBFS dapat bekerja dengan cepat dalam beberapa kasus, ia tidak menjamin solusi optimal dan rentan terhadap kesalahan.

GBFS memiliki waktu eksekusi yang lebih cepat dibandingkan dengan A* dan UCS dalam beberapa kasus tertentu karena sifat algoritmanya yang hanya mempertimbangkan informasi lokal terbaik tanpa memperhitungkan biaya aktual. A* memiliki waktu eksekusi yang lebih lambat dibandingkan dengan GBFS karena algoritma ini memadukan informasi heuristik dengan biaya aktual saat memilih simpul berikutnya untuk dieksplorasi. A* mempertimbangkan biaya aktual dari simpul awal ke simpul yang sedang dieksplorasi, sehingga memerlukan lebih banyak waktu komputasi untuk mengevaluasi dan membandingkan biaya aktual dari setiap simpul dalam antrian prioritas. UCS memiliki waktu eksekusi yang paling lambat di antara ketiga algoritma ini karena sifatnya yang mirip dengan BFS, di mana algoritma ini mempertimbangkan biaya aktual dari setiap langkah dalam pencarian. UCS harus mengevaluasi semua kemungkinan langkah dengan mempertimbangkan biaya aktualnya, sehingga kompleksitas waktu UCS cenderung lebih tinggi daripada GBFS dan A*.

Untuk memori yang dibutuhkan, dapat dilihat pada bab sebelumnya UCS selalu mengunjungi node terbanyak dibandingkan A* dan GBFS. Hal ini menyebabkan UCS membutuhkan memori paling banyak, diikuti A* dan GBFS yang menghabiskan memori paling sedikit.

BAB 6
KESIMPULAN

6.1 Kesimpulan

Dalam konteks uji coba yang dilakukan, UCS dan A* terbukti lebih andal dalam memberikan solusi optimal, Walaupun UCS memiliki waktu eksekusi yang lebih lambat dibanding A*. Sebaliknya, GBFS mungkin memberikan solusi dengan cepat dalam beberapa kasus, tetapi tidak dapat diandalkan untuk memberikan solusi optimal dalam semua kasus. Oleh karena itu, pemilihan algoritma tergantung pada kebutuhan spesifik masalah dan ketersediaan informasi heuristik.

LAMPIRAN

REPO : https://github.com/mroihn/Tucil3_13522152

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		✅