LAPORAN TUGAS KECIL 3

Semester II tahun 2023/2024

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

NIM 13522112

Nama : Dimas Bagoes Hendrianto

Kelas : K02

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

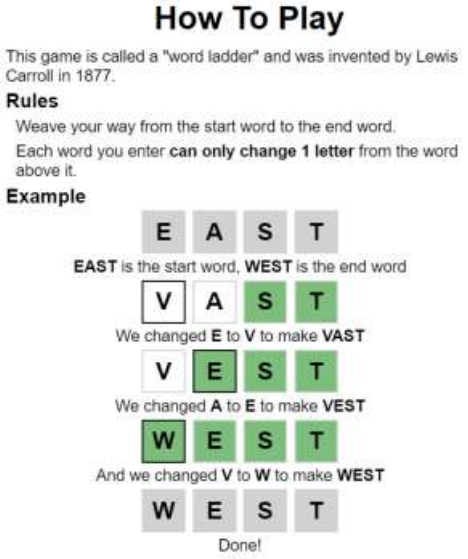
2024

DAFTAR ISI

BAB I

Dekripsi 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 1.1. Ilustrasi dan Peraturan Permainan Word Ladder

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

Permainannya cukup sederhana bukan? Jika belum paham dengan peraturan permainannya, cobalah untuk memainkan permainannya pada link sumber di atas. Jika sudah paham dengan permainannya, sekarang adalah waktunya kalian untuk membuat sebuah solver permainan tersebut dengan harapan kita dapat menemukan solusi paling optimal untuk menyelesaikan permainan Word Ladder ini.

BAB II

Teori Singkat

Uniform-Cost Search merupakan algoritma pencarian tanpa informasi (uninformed search) yang menggunakan biaya kumulatif terendah untuk menemukan jalur dari node sumber ke node tujuan. Algoritma ini beroperasi di sekitar ruang pencarian berbobot terarah untuk berpindah dari node awal ke salah satu node akhir dengan biaya akumulasi minimum. Algoritma Uniform-Cost Search masuk dalam algoritma pencarian uninformed search atau blind search karena bekerja dengan cara brute force, yaitu tidak mempertimbangkan keadaan node atau ruang pencarian.Algoritma ini umumnya digunakan untuk menemukan jalur dengan biaya kumulatif terendah dalam graph berbobot di mana node diperluas sesuai dengan biaya traversalnya dari node root. Biasanya algoritma Uniform-Cost Search diimplementasikan dengan menggunakan priority queue di mana prioritasnya adalah menurunkan biaya operasi.

Uniform-Cost Search juga dapat disebut sebagai varian dari algoritma Dijikstra. Hal ini karena pada uniform cost search, alih-alih memasukkan semua simpul ke dalam antrian prioritas (priority queue), kita hanya menyisipkan node sumber, lalu memasukkan satu per satu bila diperlukan. Di setiap iterasi, kita memeriksa apakah item sudah dalam antrian prioritas (menggunakan array yang dikunjungi). Jika ya, kita melakukan kunci penurunan, jika tidak, kita memasukkan item tersebut ke dalam antrian.

Berikut adalah cara kerja algoritma uniform-cost search:

1. Masukkan node root ke dalam priority queue
2. Ulangi langkah berikut saat antrian (queue) tidak kosong:
3. Hapus elemen dengan prioritas tertinggi
4. Jika node yang dihapus adalah node tujuan, cetak total biaya (cost) dan hentikan algoritma
5. Jika tidak, enqueue semua child dari node saat ini ke priority queue, dengan biaya kumulatifnya dari root sebagai prioritas

Di sini node root adalah node awal untuk jalur pencarian, dan priority queue tetap untuk mempertahankan jalur dengan biaya paling rendah untuk dipilih pada traversal berikutnya. Jika 2 jalur memiliki biaya traversal yang sama, node diurutkan berdasarkan abjad.

Time complexity pada algoritma uniform cost search dapat dirumuskan:

$$O(b(1 + C / \varepsilon))$$

Dimana:

b - branching factor

C - biaya optimal

ε - biaya setiap langkah

Greedy Best First Search termasuk dalam kategori algoritma informed search (pencarian heuristik). Prinsip Greedy adalah mengambil keputusan terbaik pada saat terjadi masalah yang diharapkan keputusan tersebut menjadi solusi terbaik. Oleh karena itu, keputusan yang dibuat harus keputusan terbaik, karena keputusan yang telah diambil tidak dapat diubah lagi. Greedy Best First Search sama seperti algoritma Best First Search yang memiliki sebuah fungsi evaluasi f(n). Nilai fungsi evaluasi pada Greedy Best First Search bergantung pada nilai fungsi heuristik h(n) itu sendiri. Fungsi heuristik h(n) akan memberikan estimasi arah yang benar, sehingga pencarian jalur terpendek dapat sangat cepat. Secara matematis fungsi evaluasi pada Greedy Best First Search dapat ditulis : f(n)=h(n) Dengan : f(n) = fungsi evaluasi h(n) = estimasi biaya dari simpul n ke simpul tujuan

Algoritma A* (A Star) adalah algoritma pencarian yang digunakan untuk menemukan jalur terpendek antara titik awal dan akhir. Algoritma ini sering digunakan untuk penjelajahan peta guna menemukan jalur terpendek yang akan diambil. A* awalnya dirancang sebagai masalah penjelajahan graph (graph traversal), untuk membantu robot agar dapat menemukan arahnya sendiri. A* saat ini masih tetap menjadi algoritma yang sangat populer untuk graph traversal. Algoritma A* mencari jalur yang lebih pendek terlebih dahulu, sehingga menjadikannya algoritma yang optimal dan lengkap. Algoritma yang optimal akan menemukan hasil yang paling murah dalam hal biaya untuk suatu masalah, sedangkan algoritma yang lengkap menemukan semua hasil yang mungkin dari suatu masalah. Aspek lain yang membuat A* begitu powerful adalah penggunaan graph berbobot dalam penerapannya. Graph berbobot menggunakan angka untuk mewakili biaya pengambilan setiap jalur atau tindakan. Ini berarti bahwa algoritma dapat mengambil jalur dengan biaya paling sedikit, dan menemukan rute terbaik dari segi jarak dan waktu. Adapun kelemahan utama dari algoritma ini adalah kompleksitas ruang dan waktunya. Algoritma A* membutuhkan banyak ruang untuk menyimpan semua kemungkinan jalur dan banyak waktu untuk menemukannya.

Notasi yang dipakai oleh algoritma A* adalah sebagai berikut:

$$f(n) = g(n) + h(n)$$

Dimana

f(n) = biaya estimasi terendah

g(n) = biaya dari node awal ke node n

h(n) = perkiraan biaya dari node n ke node akhir

BAB III

Implementasi

Algoritma UCS yang saya gunakan :

```
public String[] findUCS(String startWord, String endWord, Set<String> wordList) {
        Queue<Node> priorityQueue = new PriorityQueue<>(Comparator.comparingInt(Node::getCost));
        Set<String> exploredSet = new HashSet<>();
        Map<String, Integer> costMap = new HashMap<>();
        Map<String, String> parentMap = new HashMap<>();

        Node startNode = new Node(startWord);
        priorityQueue.add(startNode);
        costMap.put(startWord, 0);
        parentMap.put(startWord, null);
        this.total = 2;

        while (!priorityQueue.isEmpty()) {
            Node currentNode = priorityQueue.poll();
            String currentWord = currentNode.getWord();
            int currentCost = costMap.get(currentWord);

            if (currentWord.equals(endWord)) {
                List<String> ladder = new ArrayList<>();
                while (currentWord != null) {
                    ladder.add(0, currentWord);
                    currentWord = parentMap.get(currentWord);
                }

                return ladder.toArray(new String[0]);
            }

            exploredSet.add(currentWord);

            for (String neighbor : wordList) {
                if (!exploredSet.contains(neighbor) && isOneLetterDifference(currentWord, neighbor)) {
                    int newCost = currentCost + 1;

                    if (!costMap.containsKey(neighbor) || newCost < costMap.get(neighbor)) {
                        costMap.put(neighbor, newCost);
                        parentMap.put(neighbor, currentWord);
                        Node neighborNode = new Node(neighbor);
                        neighborNode.setCost(newCost);
                        priorityQueue.add(neighborNode);
                        this.total++;
                    }
                }
            }
        }

        return new String[]{"No ladder found"};
    }
```

1. Inisialisasi Variabel dan Struktur Data:
    - priorityQueue: digunakan untuk mengatur urutan ekspansi node berdasarkan biaya saat ini. Node dengan biaya terendah ditempatkan di depan antrian.
    - exploredSet: menyimpan kata-kata yang sudah dieksplorasi sehingga kita tidak

mengulangi ekspansi node yang sama.

- costMap: menyimpan biaya saat ini untuk setiap kata yang dieksplorasi.
- parentMap: menyimpan kata parent untuk setiap kata yang dieksplorasi.
- total: menghitung total node yang dieksplorasi.

2. Inisialisasi Node Awal:
    - Node awal diinisialisasi dengan kata awal (startWord).
    - Node awal dimasukkan ke dalam priorityQueue.
    - Biaya untuk kata awal diatur menjadi 0 dalam costMap.
    - Kata awal ditetapkan sebagai parent dengan nilai null dalam parentMap.
3. Ekspansi Node:
    - Selama priorityQueue tidak kosong, loop akan terus berjalan.
    - Node dengan biaya terendah diekstrak dari priorityQueue.
    - Kata saat ini, biaya saat ini, dan parent saat ini diambil dari node yang diekstrak.
4. Tujuan Ditemukan:
    - Jika kata saat ini sama dengan kata akhir (endWord), itu berarti kita telah menemukan jalur dari kata awal ke kata akhir.
    - Dalam hal ini, kita membangun jalur dari kata akhir ke kata awal dengan menggunakan parentMap dan menyimpannya dalam ladder.
5. Ekspansi Node Tetangga:
    - Setiap kata dalam wordList yang bukan merupakan bagian dari exploredSet dan memiliki perbedaan satu huruf dengan kata saat ini dieksplorasi.
    - Jika biaya baru (biaya saat ini + 1) lebih kecil dari biaya saat ini yang disimpan dalam costMap untuk tetangga tersebut, kita memperbarui biaya dan parent dalam costMap dan parentMap masing-masing.
    - Node tetangga dibuat dan dimasukkan ke dalam priorityQueue.
6. Pencarian Selesai:
    - Jika pencarian selesai dan jalur ditemukan, array string yang berisi jalur dari kata awal ke kata akhir dikembalikan.
    - Jika tidak ada jalur yang ditemukan, array string dengan satu elemen yang berisi pesan "No ladder found" dikembalikan.

Algoritma Greedy Best First Search yang saya gunakan :

```
public String[] findGBFS(String startWord, String endWord, Set<String> wordList) {
    Queue<Node> queue = new LinkedList<>();
    Map<String, String> parentMap = new HashMap<>();
    Set<String> visited = new HashSet<>();

    Node startNode = new Node(startWord);
    queue.add(startNode);
    visited.add(startWord);
    parentMap.put(startWord, null);
    this.total = 2;

    while (!queue.isEmpty()) {
        Node currentNode = queue.poll();
        String currentWord = currentNode.getWord();

        if (currentWord.equals(endWord)) {
            List<String> ladder = reconstructPath(parentMap, endWord);
            return ladder.toArray(new String[0]);
        }

        for (String neighbor : wordList) {
            if (!visited.contains(neighbor) && isOneLetterDifference(currentWord, neighbor)) {
                visited.add(neighbor);
                Node neighborNode = new Node(neighbor);
                queue.add(neighborNode);
                parentMap.put(neighbor, currentWord);
                this.total++;
            }
        }
    }

    return new String[]{"No ladder found"};
}
```

1. Inisialisasi Variabel dan Struktur Data:
    - queue: menyimpan node yang akan diekspansi selanjutnya.
    - parentMap: menyimpan kata parent untuk setiap kata yang dieksplorasi.
    - visited:menyimpan kata-kata yang sudah dieksplorasi untuk menghindari pengulangan.
2. Inisialisasi Node Awal:
    - Node awal diinisialisasi dengan kata awal (startWord).
    - Node awal dimasukkan ke dalam queue.
    - Kata awal ditandai sebagai sudah dieksplorasi dengan menambahkannya ke dalam visited.
    - Kata awal ditetapkan sebagai parent dengan nilai null dalam parentMap.
3. Ekspansi Node:
    - Selama queue tidak kosong, loop akan terus berjalan.
    - Node dengan biaya terendah (dalam konteks GBFS, biaya adalah estimasi jarak ke target) diekstrak dari queue.

- Kata saat ini diambil dari node yang diekstrak.

4. Tujuan Ditemukan:
    - Jika kata saat ini sama dengan kata akhir (endWord), itu berarti kita telah menemukan jalur dari kata awal ke kata akhir.
    - Dalam hal ini, kita membangun jalur dari kata akhir ke kata awal dengan menggunakan parentMap dan fungsi reconstructPath.
5. Ekspansi Node Tetangga:
    - Setiap kata dalam wordList yang belum dieksplorasi dan memiliki perbedaan satu huruf dengan kata saat ini dieksplorasi.
    - Node tetangga dibuat dan dimasukkan ke dalam queue.
    - Kata tetangga ditandai sebagai sudah dieksplorasi dengan menambahkannya ke dalam visited.
    - Parent dari kata tetangga diatur ke kata saat ini dalam parentMap.
6. Pencarian Selesai:
    - Jika pencarian selesai dan jalur ditemukan, array string yang berisi jalur dari kata awal ke kata akhir dikembalikan.
    - Jika tidak ada jalur yang ditemukan, array string dengan satu elemen yang berisi pesan "No ladder found" dikembalikan.

Algoritma A* yang saya gunakan :

```
public String[] findAstar(String startWord, String endWord, Set<String> wordList) {
    PriorityQueue<NodeH> openSet = new PriorityQueue<>(Comparator.comparingInt(NodeH::getTotalCost));
    Set<String> closedSet = new HashSet<>();
    Map<String, Integer> costMap = new HashMap<>();
    Map<String, String> parentMap = new HashMap<>();

    NodeH startNode = new NodeH(startWord, 0, heuristic(startWord, endWord));
    openSet.add(startNode);
    costMap.put(startWord, 0);
    parentMap.put(startWord, null);
    this.total = 2;

    while (!openSet.isEmpty()) {
        NodeH currentNode = openSet.poll();
        String currentWord = currentNode.getWord();

        if (currentWord.equals(endWord)) {
            List<String> ladder = reconstructPath(parentMap, endWord);
            return ladder.toArray(new String[0]);
        }
        closedSet.add(currentWord);

        for (String neighbor : wordList) {
            if (!closedSet.contains(neighbor) && isOneLetterDifference(currentWord, neighbor)) {
                int newCost = costMap.get(currentWord) + 1;
                if (!costMap.containsKey(neighbor) || newCost < costMap.get(neighbor)) {
                    costMap.put(neighbor, newCost);
                    parentMap.put(neighbor, currentWord);
```

```
                    NodeH neighborNode = new NodeH(neighbor, newCost, heuristic(neighbor, endWord));
                    openSet.add(neighborNode);
                    this.total++;
                }
            }
        }
    }

    return new String[]{"No ladder found"};
}
```

1. Inisialisasi Variabel dan Struktur Data:
    - openSet: menyimpan node yang akan diekspansi berdasarkan perkiraan biaya total (biaya sejauh ini ditambah estimasi biaya yang tersisa).
    - closedSet: menyimpan kata-kata yang sudah dieksplorasi untuk menghindari pengulangan.
    - costMap: menyimpan biaya sejauh ini untuk setiap kata yang dieksplorasi.
    - parentMap: menyimpan kata parent untuk setiap kata yang dieksplorasi.
2. Inisialisasi Node Awal:
    - Node awal diinisialisasi dengan kata awal (startWord), biaya sejauh ini 0, dan perkiraan biaya tersisa dihitung menggunakan fungsi heuristik.
    - Node awal dimasukkan ke dalam openSet.
    - Biaya sejauh ini untuk kata awal diatur menjadi 0 dalam costMap.
    - Kata awal ditetapkan sebagai parent dengan nilai null dalam parentMap.
3. Ekspansi Node:
    - Selama openSet tidak kosong, loop akan terus berjalan.
    - Node dengan biaya total terendah (dalam konteks A*, biaya total adalah biaya sejauh ini ditambah estimasi biaya tersisa) diekstrak dari openSet.
    - Kata saat ini diambil dari node yang diekstrak.
4. Tujuan Ditemukan:
    - Jika kata saat ini sama dengan kata akhir (endWord), itu berarti kita telah menemukan jalur dari kata awal ke kata akhir.
    - Dalam hal ini, kita membangun jalur dari kata akhir ke kata awal dengan menggunakan parentMap dan fungsi reconstructPath.
5. Ekspansi Node Tetangga:
    - Setiap kata dalam wordList yang belum dieksplorasi dan memiliki perbedaan satu huruf dengan kata saat ini dieksplorasi.
    - Jika kata tetangga belum ada di closedSet (belum dieksplorasi) atau biaya baru (biaya sejauh ini + 1) lebih kecil dari biaya yang disimpan dalam costMap untuk

tetangga tersebut:

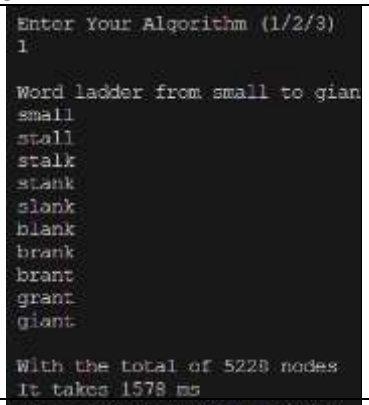
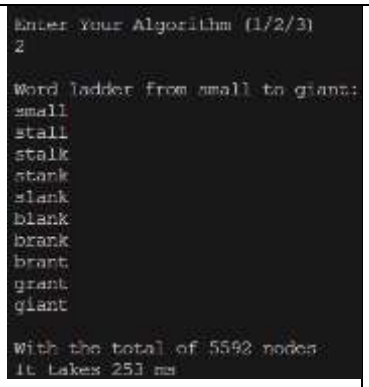
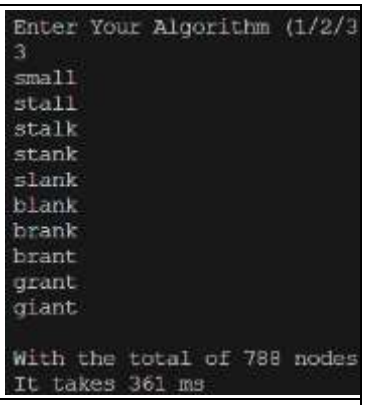
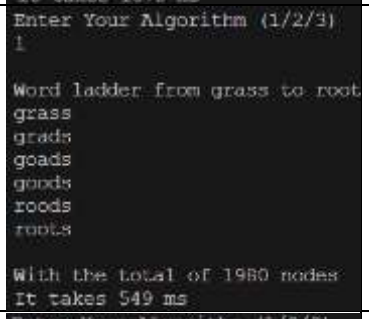
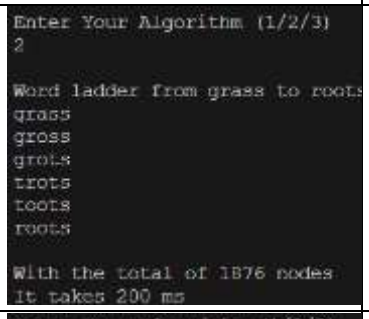
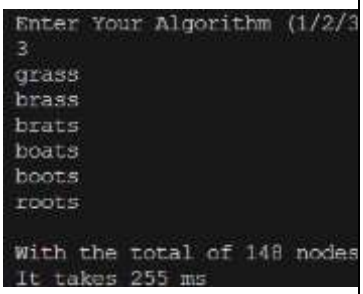
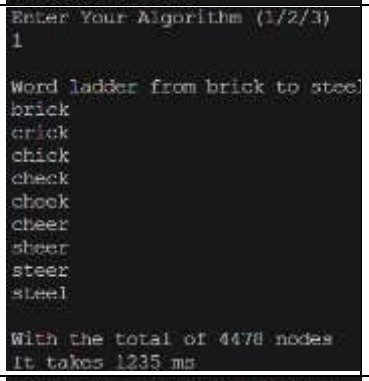
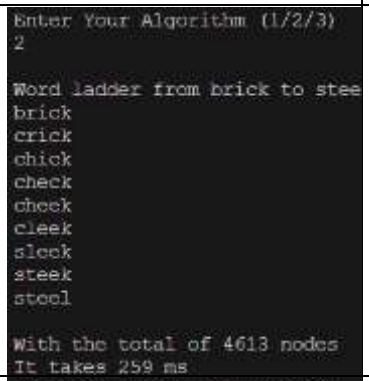
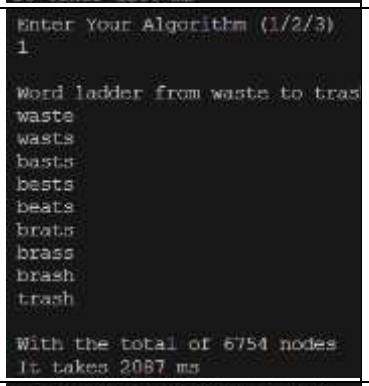
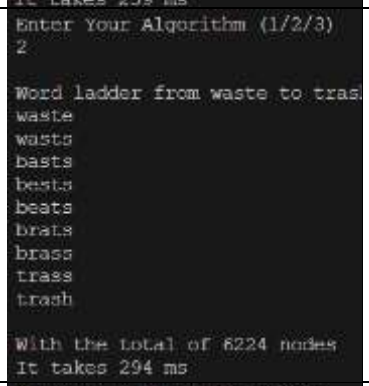
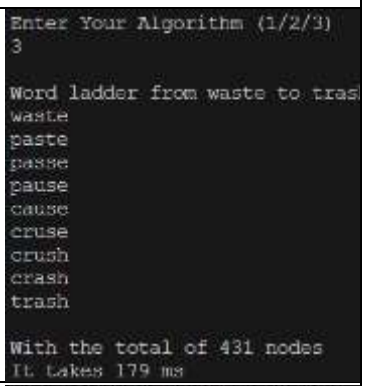
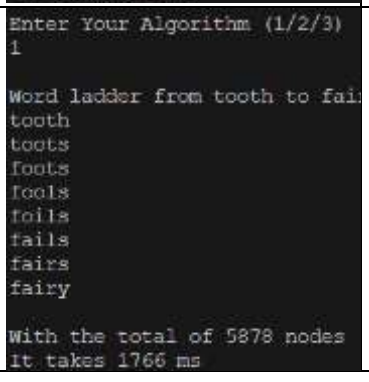
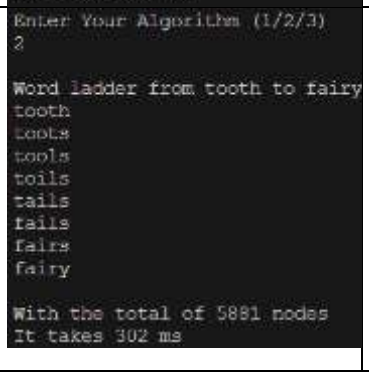
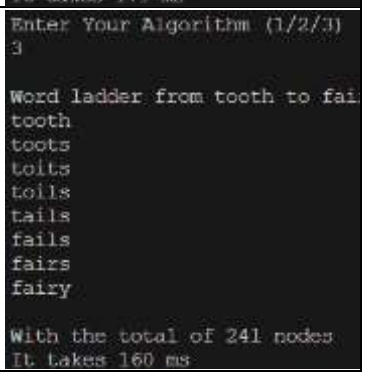
- Biaya sejauh ini untuk tetangga diperbarui dalam costMap.
- Parent dari tetangga diatur ke kata saat ini dalam parentMap.
- Node tetangga dibuat dengan biaya total yang dihitung berdasarkan biaya sejauh ini dan estimasi biaya tersisa menggunakan fungsi heuristik.
- Node tetangga dimasukkan ke dalam openSet.

6. Pencarian Selesai:
    - Jika pencarian selesai dan jalur ditemukan, array string yang berisi jalur dari kata awal ke kata akhir dikembalikan.
    - Jika tidak ada jalur yang ditemukan, array string dengan satu elemen yang berisi pesan "No ladder found" dikembalikan.

BAB IV

Eksperimen

Ladder	UCS	GBFS	A*
frown ↓ smile	Enter Your Algorithm (1/2/3) 1 Word ladder from frown to smile frown frows flows slows slots slits skits skite smite smile With the total of 5752 nodes It takes 1721 ms	Enter Your Algorithm (1/2/3) 2 Word ladder from frown to smile frown flown flows slows slows stoas stoae stole stile smile With the total of 5594 nodes It takes 246 ms	Enter Your Algorithm (1/2/3) 3 frown frows flows slows slops slope stope stole stile smile With the total of 546 nodes It takes 318 ms
knack ↓ skill	Enter Your Algorithm (1/2/3) 1 Word ladder from knack to skill knack snack stack stalk stall still skill With the total of 1140 nodes It takes 424 ms	Enter Your Algorithm (1/2/3) 2 Word ladder from knack to skill knack snack stack stalk stall still skill With the total of 1027 nodes It takes 164 ms	Enter Your Algorithm (1/2/3) 3 knack snack stack stalk stall still skill With the total of 68 nodes It takes 282 ms
mitts ↓ scarf	Enter Your Algorithm (1/2/3) 1 Word ladder from mitts to scarf mitts motts botts boats boars soars scars scarf With the total of 5826 nodes It takes 1903 ms	Enter Your Algorithm (1/2/3) 2 Word ladder from mitts to scarf mitts motts moats meats seats scats scars scarf With the total of 6145 nodes It takes 293 ms	Enter Your Algorithm (1/2/3) 3 mitts motts moots soots scots scats scars scarf With the total of 638 nodes It takes 285 ms
quiet ↓ place	Enter Your Algorithm (1/2/3) 1 Word ladder from quiet to place quiet quilt guilt guile guide glide glade glace place With the total of 4157 nodes It takes 1145 ms	Enter Your Algorithm (1/2/3) 2 Word ladder from quite to place quite suite skite skate slate plate place With the total of 1720 nodes It takes 165 ms	Enter Your Algorithm (1/2/3) 3 quiet quilt guilt guile guide glide glade glace place With the total of 100 nodes It takes 253 ms

small ↓ giant			
grass ↓ roots			
brick ↓ steel			
waste ↓ trash			
tooth ↓ fairy			

grain ↓ wheat	Enter Your Algorithm (1/2/3) 1 Word ladder from grain to whea grain groin groan groat gloat bloat bleat cleat cheat wheat With the total of 5286 nodes It takes 1720 ms	Enter Your Algorithm (1/2/3) 2 Word ladder from grain to whea grain groin groan groat gloat bloat bleat cleat cheat wheat With the total of 3885 nodes It takes 257 ms	Enter Your Algorithm (1/2/3) 3 Word ladder from grain to wheat grain groin groan groat gloat bloat bleat cleat cheat wheat With the total of 301 nodes It takes 203 ms
sore ↓ head	Enter Your Algorithm (1/2/3) 1 Word ladder from sore to hea sore sere here herd head With the total of 3150 nodes It takes 438 ms	Enter Your Algorithm (1/2/3) 2 Word ladder from sore to head: sore sere here herd head With the total of 3090 nodes It takes 207 ms	Enter Your Algorithm (1/2/3) 3 Word ladder from sore to hea sore sere here herd head With the total of 61 nodes It takes 127 ms
skim ↓ milk	Enter Your Algorithm (1/2/3) 1 Word ladder from skim to mil skim skis seis sels mels mils milk With the total of 3556 nodes It takes 590 ms	Enter Your Algorithm (1/2/3) 2 Word ladder from skim to mil skim skis seis sels mels mils milk With the total of 3588 nodes It takes 259 ms	Enter Your Algorithm (1/2/3) 3 Word ladder from skim to mil skim skis seis sels mels mils milk With the total of 213 nodes It takes 135 ms

Kesimpulan

Dari pengujian yang telah dilakukan bisa dilihat kalau waktu yang diperlukan untuk memperoleh ladder dengan algoritma UCS dan GBFS cenderung sama dengan jumlah node yang tidak berpaut jauh. Dengan menggunakan algoritma UCS terdapat perbedaan yang cukup signifikan dimana lebih lama dari segi waktu yang juga beriringan dengan tidak lebih efisien dalam hal penggunaan nodes. Penggunaan memori untuuk algoritma A* tentunya lebih banyak apabila dibandingkan dengan dua algoritma lainnya hal ini terjadi karena A* akan dilakukan secara heuristik yang memerlukan alokasi penggunaan memori yang lebih. Dari tabel pengujian dapat dilihat kalau jalur yang dihasilkan hisa berbeda antar satu algoritma dengan yang lainnya hal ini terjadi karena pembangkitan node untuk setiap algoritma memang berbeda beda, akan tetapi keseluruhan algoritma tetap membangkitkan node tetangganya dengan hanya satu huruf berbeda untuk setiap katanya.

Lampiran

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		✓

Link

https://github.com/dimasb1954/Tucil3_13522112.git