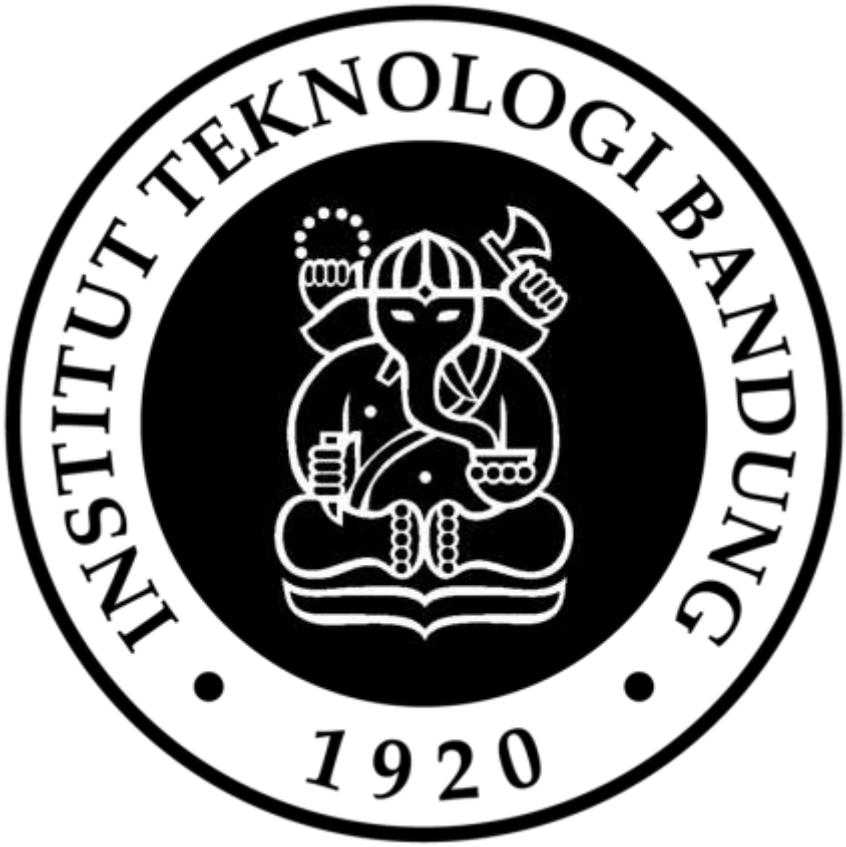
Laporan Tugas Kecil 3
IF2211 Strategi Algoritma

Aplikasi UCS, A*, dan Greedy Best First Search untuk Mencari Solusi Terpendek dalam Permainan Word Ladder



Disusun oleh:

Akbar Al Fattah 13522036

PROGRAM STUDI TEKNIK INFORMATIKA

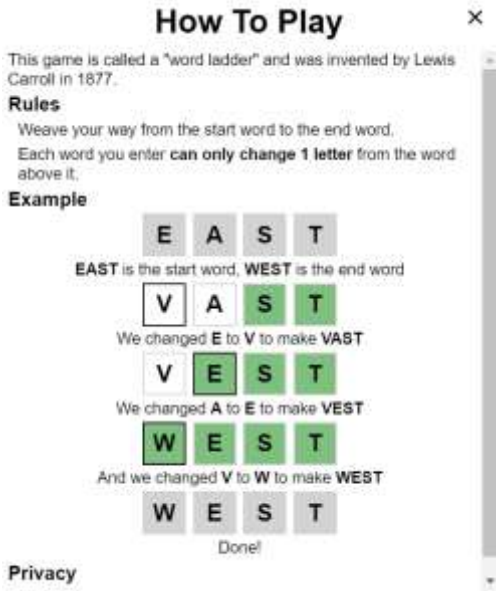
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

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/)

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 2: Dasar Teori

A. Algoritma A*

Algoritma A* adalah algortima pencarian di graf yang menggunakan jarak antar edge sesungguhnya (g(n)) dan nilai heuristik/estimasi cost (h(n)). Pada algoritma A*, nilai f(n) adalah f(n) = g(n) + h(n). Nilai heuristik h(n) yang digunakan harus admissible.
Nilai heuristik yang admissible adalah nilai heuristik yang tidak melebihi cost sungguhan g(n). Fungsi untuk menghitung nilai heuristik dibebaskan asal admissible dan dapat diaplikasikan sesuai dengan permasalahan yang ingin diselesaikan.
Berikut adalah langkah-langkah algoritma A*

1. Buat Priority Queue yang mengutamakan nilai f(n) yang paling kecil.
2. Hitung nilai heuristik h(n) untuk semua node dan g(n) untuk semua edge dan cost g(n) untuk node awal (g(n) = 0 karena g(n) adalah jarak dari node ke titik awal)
3. Hitung nilai f(n) = g(n) + h(n) dari node awal.
4. Masukkan node awal ke priority queue dengan nilai prioritasnya adalah f(n).
5. Ulangi langkah 5 dan seterusnya sampai priority queue kosong atau path sudah ditemukan:
6. Dapatkan node paling depan di priority queue
7. Jika node tersebut sudah pernah dikunjungi, ulangi dari langkah 5, jika tidak, lanjut ke langkah 6.
8. Jika node tersebut tidak sama dengan node tujuan, periksa semua tetangga dari node tersebut dengan langkah di bawah ini:
9. Hitung nilai g(n) yang baru dengan cara mendapatkan nilai g node sekarang ditambah dengan cost dari edge node sekarang ke node neighbor yang sedang diperiksa.
10. Jika nilai g(n) yang baru lebih kecil dari nilai g(n) tetangga yang sedang diperiksa lakukan langkah 10-14. Jika tidak, lewati langkah 10-14,periksa tetangga berikutnya, dan ulangi dari langkah 9.
11. Perbarui nilai g(n) tetangga yang sedang diperiksa dengan g(n) baru yang baru saja dihitung di langkah 8
12. Hitung nilai f(n) = g(n) + h(n) dari node tetangga yang sedang diperiksa.
13. Buat parent dari node tetangga yang sedang diperiksa menjadi node yang sekarang sedang dikunjungi.
14. Masukkan node tetangga tersebut ke priority queue.
15. Jika node yang dikunjungi pada langkah 7 adalah node tujuan, susun path yang ditemukan dan hentikan semua langkah algoritma ini.

B. Algoritma Greedy Best First Search (GBFS)

Algoritma Greedy Best First Search adalah algoritma A* yang memiliki syarat khusus **semua nilai g(n) tidak digunakan (dianggap tidak ada)** dan **h(n) digunakan.**
Akibatnya, nilai f(n) untuk Greedy Best First Search adalah **f(n) = h(n).**
Langkah-langkah algoritma GBFS kurang lebih mirip dengan A* dengan beberapa perbedaan. Berikut adalah langkah-langkah algoritma GBFS.

1. Buat Priority Queue yang mengutamakan nilai f(n) yang paling kecil.

2. Hitung nilai heuristik h(n) untuk semua node
3. Hitung nilai f(n) = h(n) dari node awal.
4. Masukkan node awal ke priority queue dengan nilai prioritasnya adalah f(n).
5. Ulangi langkah 5 dan seterusnya sampai priority queue kosong atau path sudah ditemukan:
6. Dapatkan node paling depan di priority queue
7. Jika node tersebut sudah pernah dikunjungi, ulangi dari langkah 5, jika tidak, lanjut ke langkah 6.
8. Jika node tersebut tidak sama dengan node tujuan, periksa semua tetangga dari node tersebut dengan langkah di bawah ini:
9. Hitung nilai f(n) = h(n) dari node tetangga yang sedang diperiksa.
10. Buat parent dari node tetangga yang sedang diperiksa menjadi node yang sekarang dikunjungi.
11. Masukkan node tetangga tersebut ke priority queue.
12. Jika node yang dikunjungi pada langkah 7 adalah node tujuan, susun path yang ditemukan dan hentikan semua langkah algoritma ini.

C. Algoritma Uniform Cost Search (UCS)

Algoritma UCS adalah algoritma A* yang memiliki syarat khusus **semua nilai h(n) tidak digunakan** dan **g(n) digunakan**. Artinya, UCS tidak menggunakan nilai heuristik h(n) dan tidak perlu mendefinisikan fungsi h(n). Akibatnya, nilai f(n) untuk UCS adalah **f(n) = g(n).**

Langkah-langkah algoritma UCS kurang lebih mirip dengan A* dengan beberapa perbedaan. Berikut adalah langkah-langkah algoritma UCS.

1. Buat Priority Queue yang mengutamakan nilai f(n) yang paling kecil.
2. Hitung nilai g(n) untuk semua edge dengan g(n) node awal adalah 0
3. Hitung nilai f(n) = g(n) dari node awal.
4. Masukkan node awal ke priority queue dengan nilai prioritasnya adalah f(n).
5. Ulangi langkah 5 dan seterusnya sampai priority queue kosong atau path sudah ditemukan:
6. Dapatkan node paling depan di priority queue
7. Jika node tersebut sudah pernah dikunjungi, ulangi dari langkah 5, jika tidak, lanjut ke langkah 6.
8. Jika node tersebut tidak sama dengan node tujuan, periksa semua tetangga dari node tersebut dengan langkah di bawah ini:
9. Hitung nilai g(n) yang baru dengan cara mendapatkan nilai g node sekarang ditambah dengan cost dari edge node sekarang ke node neighbor yang sedang diperiksa.
10. Jika nilai g(n) yang baru lebih kecil dari nilai g(n) tetangga yang sedang diperiksa lakukan langkah 10-14. Jika tidak, lewati langkah 10-14,periksa tetangga berikutnya, dan ulangi dari langkah 9.
11. Perbarui nilai g(n) tetangga yang sedang diperiksa dengan g(n) baru yang baru saja dihitung di langkah 8
12. Hitung nilai f(n) = g(n) dari node tetangga yang sedang diperiksa.

13. Buat parent dari node tetangga yang sedang diperiksa menjadi node yang sekarang sedang dikunjungi.
14. Masukkan node tetangga tersebut ke priority queue.
15. Jika node yang dikunjungi pada langkah 7 adalah node tujuan, susun path yang ditemukan dan hentikan semua langkah algoritma ini.

D. Hubungan Antara Algoritma A*, UCS, Greedy Best First Search, dan Breadth First Search

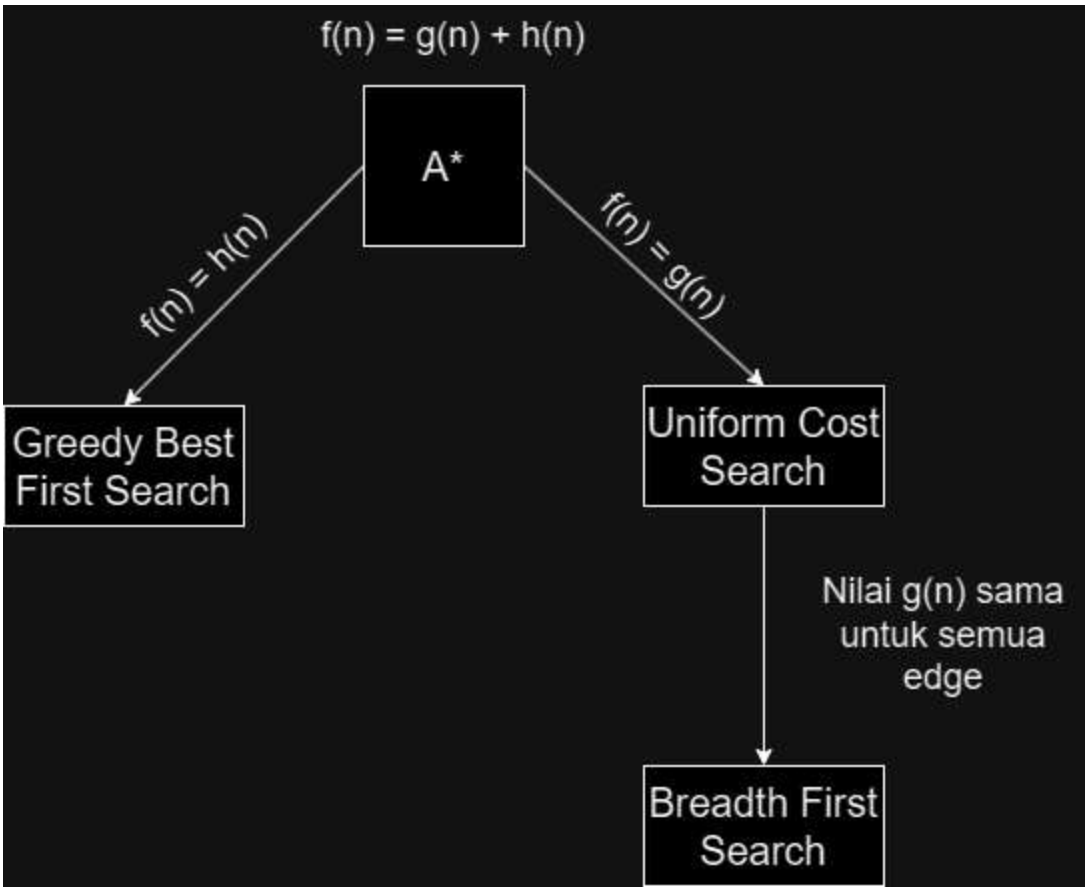
Sebelumnya telah kita ketahui bahwa algoritma A* adalah bentuk umum dari kedua algoritma lain yang digunakan (UCS dan GBFS). Namun, ada lagi algoritma yang berhubungan dengan ketiga algoritma di atas, yaitu breadth first search (BFS).

Breadth First Search adalah **kasus khusus dari UCS yang memiliki nilai g(n) yang sama semua untuk setiap edge**.

Berikut ini adalah pembuktian bahwa BFS adalah UCS yang memiliki nilai g(n) yang sama semua untuk semua edge.

1. Pada UCS, yang dijadikan nilai prioritas untuk pencarian node yang akan dikunjungi terlebih dahulu adalah nilai **f(n) = g(n).** Nilai g(n) (jarak antar kedua node) bisa sama untuk beberapa atau semua edge dan bisa berbeda dengan edge yang lain. Node yang memiliki nilai g(n) yang sama yang akan dikunjungi terlebih dahulu bergantung pada definisi yang ditetapkan oleh programmer, apakah urutan dimasukkan ke priority queue atau berdasarkan ketentuan lain.
2. Akan tetapi, jika **g(n) bernilai sama semua untuk semua edge**, maka **nilai f(n) tidak ada artinya karena tidak ada nilai f(n) yang berbeda.** Akibatnya, nilai prioritas node akan sama semua dan **penggunaan priority queue menjadi tidak berguna di sini** karena **tidak ada node yang diprioritaskan untuk dikunjungi terlebih** dahulu di queue. Selain itu, nilai prioritas node yang sama semua akan menyebabkan node yang dimasukkan ke priority queue adalah **semua node yang merupakan tetangga dari node yang sedang dikunjungi.**
3. **Perhatikan bahwa pernyataan di atas adalah definisi dari Breadth First Search (BFS).**
4. Dari pernyataan nomor 2 dan 3, dapat disimpulkan bahwa **Breadth First Search adalah kasus khusus UCS yang memiliki nilai g(n) yang sama untuk semua edge.**

Berikut adalah gambar hubungan antara A*, GBFS, UCS, dan BFS.



Gambar 2. Hubungan A*, GBFS, UCS, dan BFS
(Sumber: Diagram yang dibuat sendiri oleh penulis di https://app.diagrams.net/)

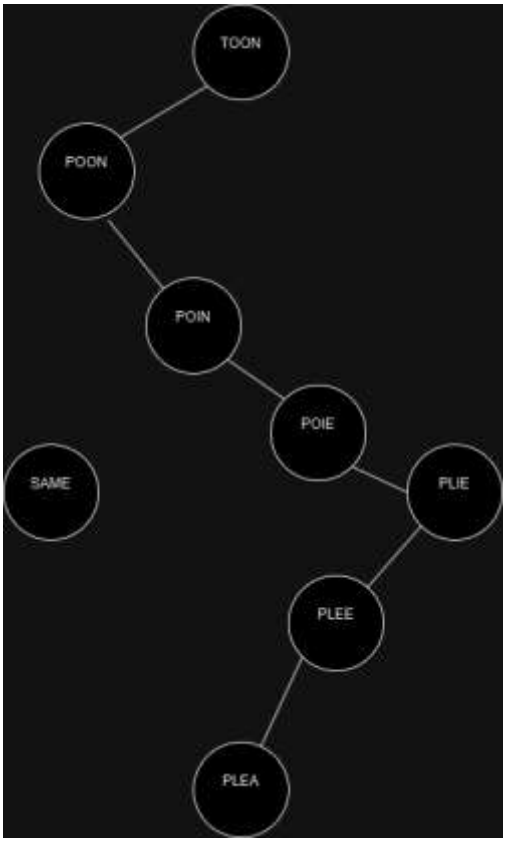
Bab 3: Pemecahan Masalah

A. Konstruksi Graf

Pada Word Ladder, definisi tetangga dari suatu node adalah **semua kata yang hanya berbeda satu huruf dari node tersebut.**

Contoh ilustrasi: Misalkan kita punya dictionary {TOON, POON, PLEE, SAME, POIE, PLEA, PLIE, POIN}, titik awalnya dimulai dari TOON dan tujuannya adalah PLEA.

Maka, graf yang dikonstruksi untuk memecahkan masalah ini terdapat pada gambar di bawah ini.



Gambar 3. Hasil graf yang dibangun dari dictionary
(Sumber: Diagram yang dibuat sendiri oleh penulis di https://app.diagrams.net/)

Hasilnya, didapatkan sebuah graf **unweighted/berbobot sama semua dan undirected/tidak berarah.**

B. Definisi Nilai h(n) (nilai heuristik graf)

Nilai heuristik tiap node yang digunakan pada graf ini adalah **banyak huruf yang berbeda dari kata node tersebut dengan tujuan akhir.**

Nilai heuristik ini **admissible** karena estimasi nilai h(n) tidak melebihi dari jarak/cost/g(n) sebenarnya dari setiap node. Nilai **h(n) minimum adalah 0** (semua huruf kata tersebut sama dengan tujuan akhir) dan **nilai h(n) maksimum adalah panjang dari kata tersebut** (semua huruf kata tersebut berbeda dengan tujuan akhir), sedangkan **nilai g(n) minimum adalah 0** dan **nilai g(n) maksimum adalah tak hingga.**

C. Aplikasi ke tiap algoritma

1. UCS

Perhatikan kembali bahwa graf Word Ladder adalah **graf unweighted/berbobot sama semua.** Akibatnya, ini adalah kasus khusus UCS, yaitu Breadth First Search (BFS). Oleh karena itu, algoritma UCS yang digunakan di graf Word Ladder **ekuivalen dengan algoritma Breadth First Search.** Karena UCS yang digunakan pada adalah BFS, maka UCS **menjamin path yang ditemukan adalah path terpendek.**

2. **GBFS**
   Berbeda dengan UCS dan A*, GBFS **tidak menjamin path yang ditemukan adalah path terpendek** karena GBFS hanya memikirkan nilai heuristik h(n) terkecil dari suatu node tanpa memikirkan jarak node dari titik asal.
3. **A***
   Graf Word Ladder adalah graf unweighted, kita anggap nilai g(n) untuk semua edge adalah 1. Karena h(n) adalah nilai heuristik yang admissible, maka A* **menjamin path yang ditemukan adalah path terpendek**. Berbeda dengan UCS/BFS, A* jauh lebih efisien dan lebih cepat karena algoritma A* **menghindari beberapa node yang dianggap sudah memiliki g(n) yang besar,** sedangkan UCS/BFS mengecek semua tetangga**.**

Bab 4: Source Code Algortima Pencarian Path

Screenshot source code

Semua source code ini dapat dilihat di Graph.java

- Algoritma UCS
  (Catatan: Implementasi algoritma UCS di problem ini ekuivalen dengan Breadth First Search (BFS) karena graf pada permasalahan ini adalah graf yang tidak memiliki bobot *(unweighted graph),* sudah dijelaskan alasannya sebelumnya)

```java
public ArrayList<String> UCS(String src, String target){
    //NOTE: Graph ini memiliki cost yang sama di semua edge (graph word ladder adalah graph UNWEIGHTED)
    //karena semua edge memiliki cost yang sama, maka UCS bekerja sama persis dengan Breadth first search
    //BFS adalah kasus khusus dari UCS di mana semua edge memiliki cost yang sama

    //BFS
    Queue<String> nodeQueue = new LinkedList<>();
    ArrayList<String> visitedNodes = new ArrayList<>();
    HashMap<String,String> parentMap = new HashMap<>(); //map yang menunjukkan relasi (A dan B, B adalah parent dari A)
    ArrayList<String> path = new ArrayList<>();
    try(BufferedWriter BW = new BufferedWriter(new FileWriter(fileName:"./src/cache/visitedNodes.dat"))){ //catat ke visitedNodes.dat di folder cache
        visitedNodes.add(src); //tandai sudah dikunjungi
        nodeQueue.add(src); //masukkan ke queue
        parentMap.put(src, value:null); //node awal tidak punya parent
        String top = new String();
        while(!nodeQueue.isEmpty()){//selama queue tidak kosong
            top = nodeQueue.poll(); //dapatkan node yang paling depan di queue
            BW.write(top); //tulis ke visitedNodes.dat
            BW.newLine(); //tulis newline ke visitedNodes.dat
            if(top.equals(target)){ //hentikan pencarian jika elemen terdepan queue = kata akhir
                break;
            }
            TreeSet<String> neighbors = graph.get(top); //dapatkan semua neighbor dari node terdepan queue
            for (String neighbor : neighbors) {
                if (!visitedNodes.contains(neighbor)) { //jika neighbor yang diperiksa belum dikunjungi
                    visitedNodes.add(neighbor); //tandai sudah dikunjungi
                    nodeQueue.add(neighbor);//masukkan ke queue
                    parentMap.put(neighbor,top); //node terdepan saat ini adalah parent dari neighbor
                }
            }
        }

        if(top.equals(target)){ //jika path ditemukan
            //rekonstruksi path yang dicari dari belakang (target)
            String cur = target;
            while (cur != null) {
                path.add(index:0, cur); //tambahkan node yang ditandai sekarang ke path dari depan
                cur = parentMap.get(cur); //pindahkan tanda ke parentnya
            }
        }
    } catch (Exception e){
        e.getStackTrace();
    }
    return path; //kembalikan pathnya
}
```

Tangkapan Layar Source Code Algoritma UCS (yang ekuivalen dengan BFS)

- Algoritma Greedy Best First Search
  (Catatan: nilai heuristik yang digunakan adalah banyaknya huruf pada suatu kata yang **berbeda** dengan kata **tujuan. Namun, GBFS tidak menjamin bahwa path yang dicari adalah path terpendek.**)

```
public ArrayList<String> GBFS(String src, String target){
    //Nilai F dari GBFS hanyalah nilai heuristik node yang bersangkutan dan G pasti sama dengan 0
    //Nilai heuristik adalah banyak huruf yang berbeda dari kata suatu node dengan node tujuan
    PriorityQueue<HeuristicNode> pq = new PriorityQueue<>((n1,n2) -> n1.getF() - n2.getF()); //prioritasnya adalah node dengan nilai f yang terkecil di depan
    HashSet<String> visited = new HashSet<>();


    try(BufferedWriter BW = new BufferedWriter(new FileWriter(fileName:"./src/cache/visitednodes.dat"))){//catat semua node yang dikunjungi
        pq.add(new HeuristicNode(src, g:0, mismatchCounter(src, target)));//tambahkan node baru ke priority queue
        while(!pq.isEmpty()){
            HeuristicNode cur = pq.poll();// dapatkan node paling depan
            if (visited.contains(cur.getWord())) { //lewati node paling depan sekarang jika sudah pernah dikunjungi
                continue;
            }
            visited.add(cur.getWord()); //tandai node saat ini sudah dikunjungi
            BW.write(cur.getWord()); //tulis node saat ini ke visitednodes.dat
            BW.newLine();
            if(cur.getWord().equals(target)){ //sudah ketemu
                //rekonstruksi path dari source
                ArrayList<String> path = new ArrayList<>();
                HeuristicNode curNode = cur;

                while(curNode != null){
                    path.add(index:0,curNode.getWord()); //tambahkan node dari depan
                    curNode = curNode.getParent();  //lanjut ke parent dari node saat ini
                }

                return path; //kembalikan path saat ini
            }

            for(String neighbor : graph.get(cur.getWord())){ //cek semua tetangga dari node saat ini
                HeuristicNode neighborNode = new HeuristicNode(neighbor, g:0, mismatchCounter(neighbor, target));
                //buat node baru dengan g = 0 (karena tidak digunakan di GBF)
                neighborNode.setParent(cur);
                //parent dari tetangga saat ini adalah node saat ini
                pq.add(neighborNode); //masukkan node tetangga ke priority queue
            }
        }
    } catch (Exception e){
        e.printStackTrace();
    }
    return new ArrayList<>();
}
```

Tangkapan Layar Source Code Algoritma Greedy Best First Search

- Algoritma A*

```
        } catch (Exception e){
            e.printStackTrace();
        }

        return new ArrayList<>(); //jawabannya tidak ada karena pathnya tidak ditemukan
    }
```

*Tangkapan Layar Source Code Algoritma A**

Bab 5: Uji Coba Program

Keterangan: dictionary yang digunakan program ini dapat dilihat di situs di bawah ini
https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/dictionary.txt

1. EAST -> WEST
   A. UCS

   B. Greedy Best First Search

C. A*

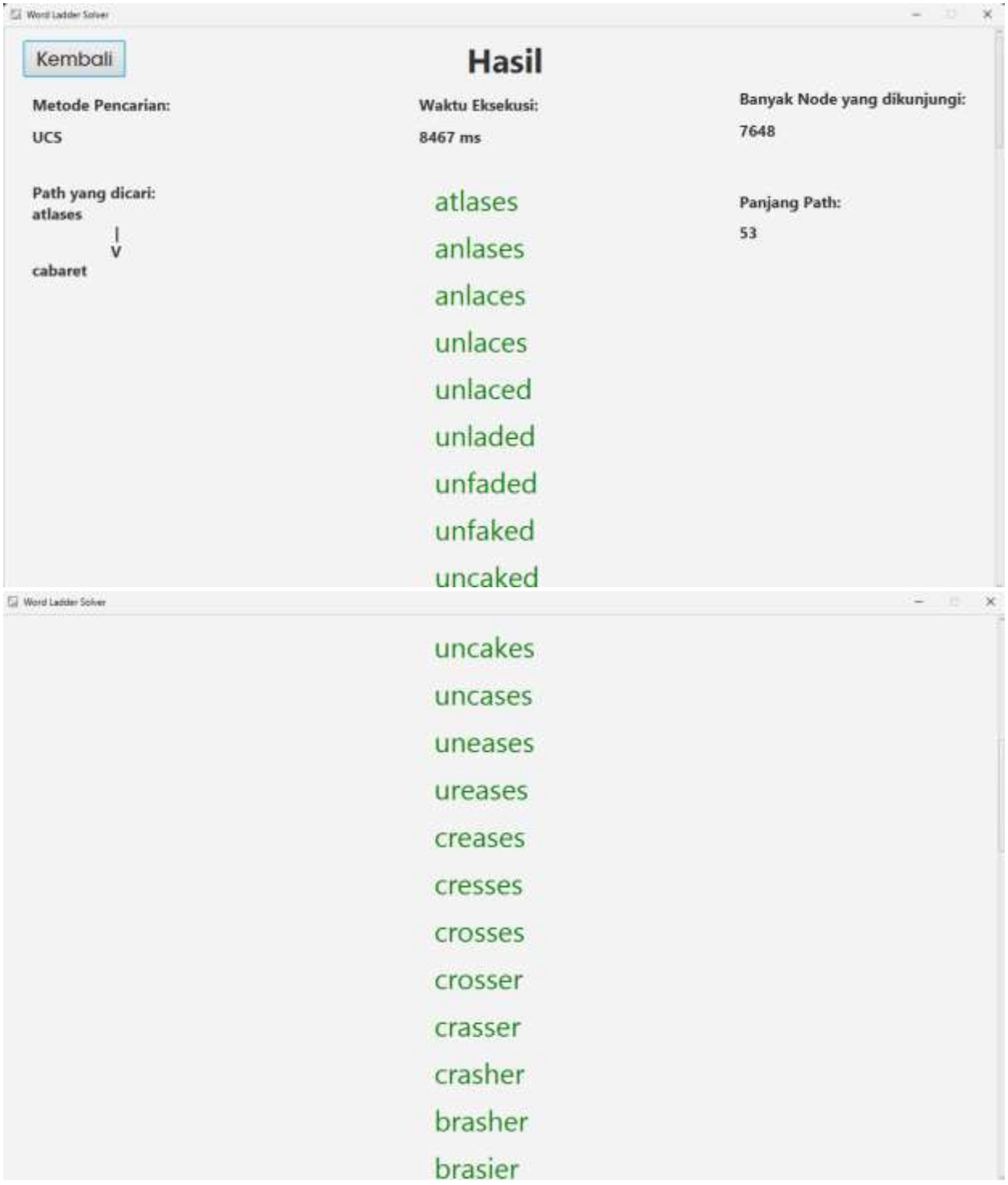
2. GUITAR -> PHONES

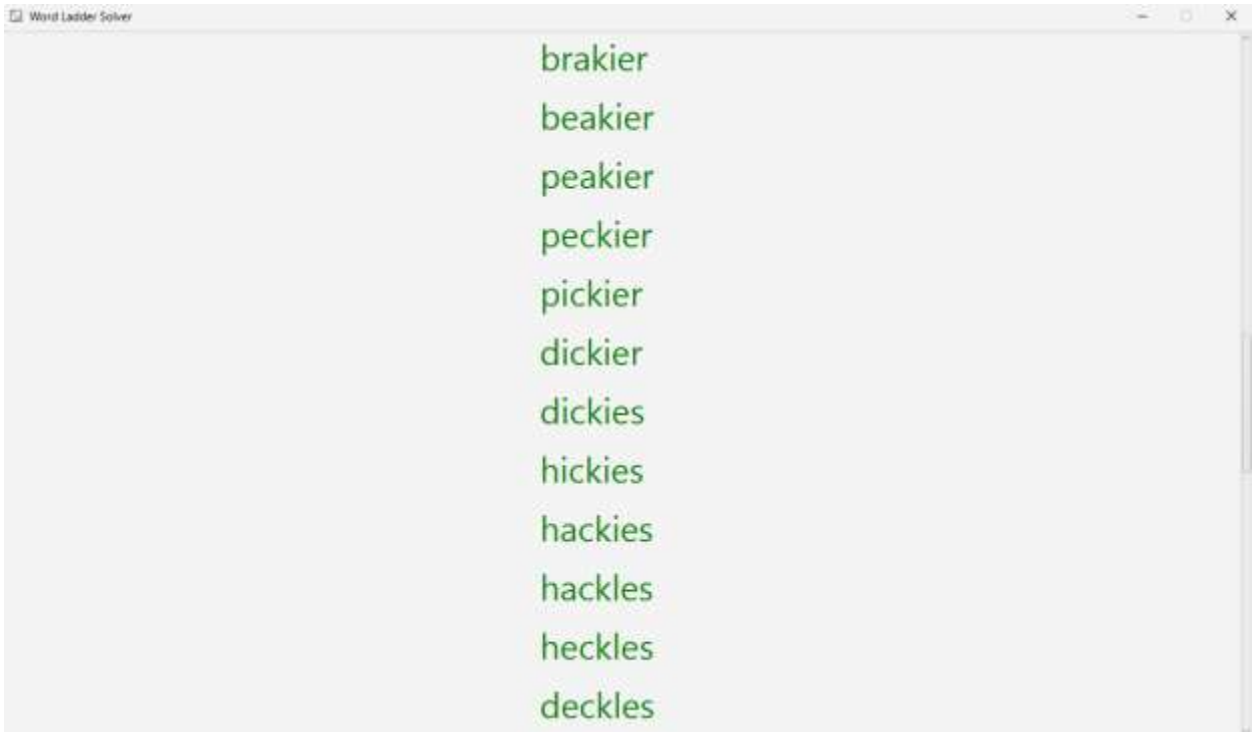
A. UCS

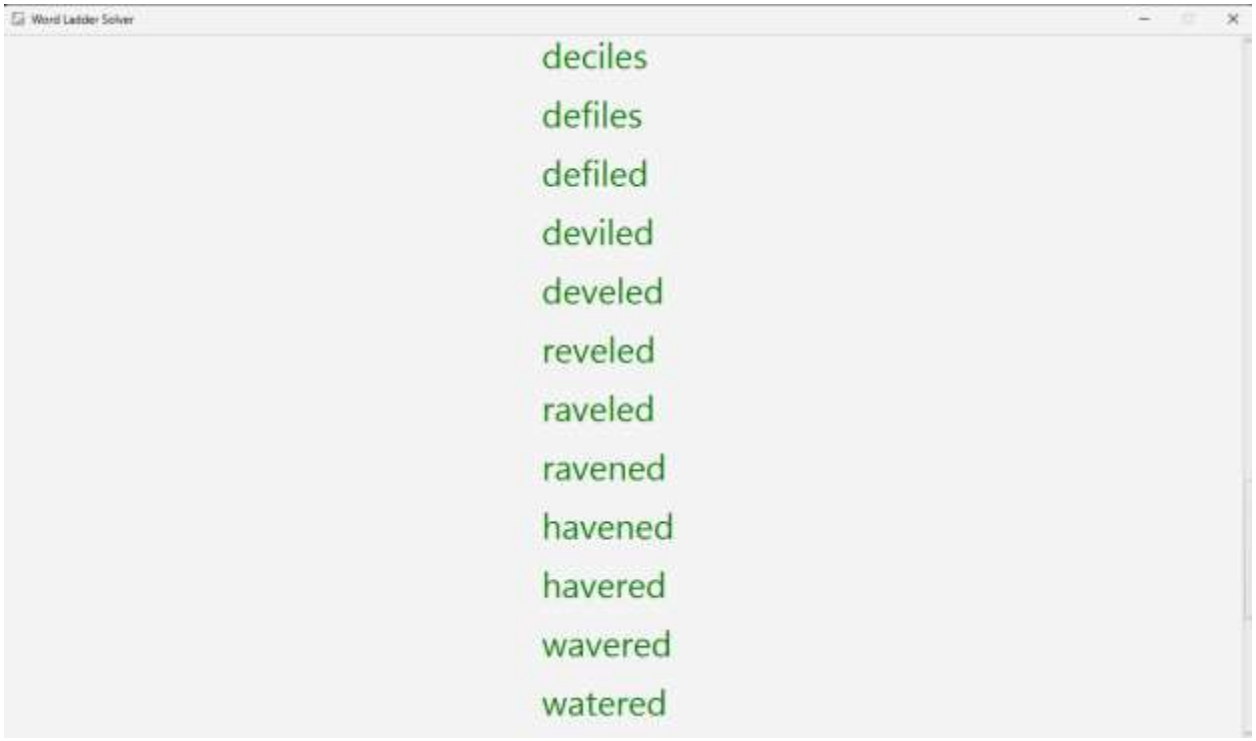
B. Greedy Best First Search

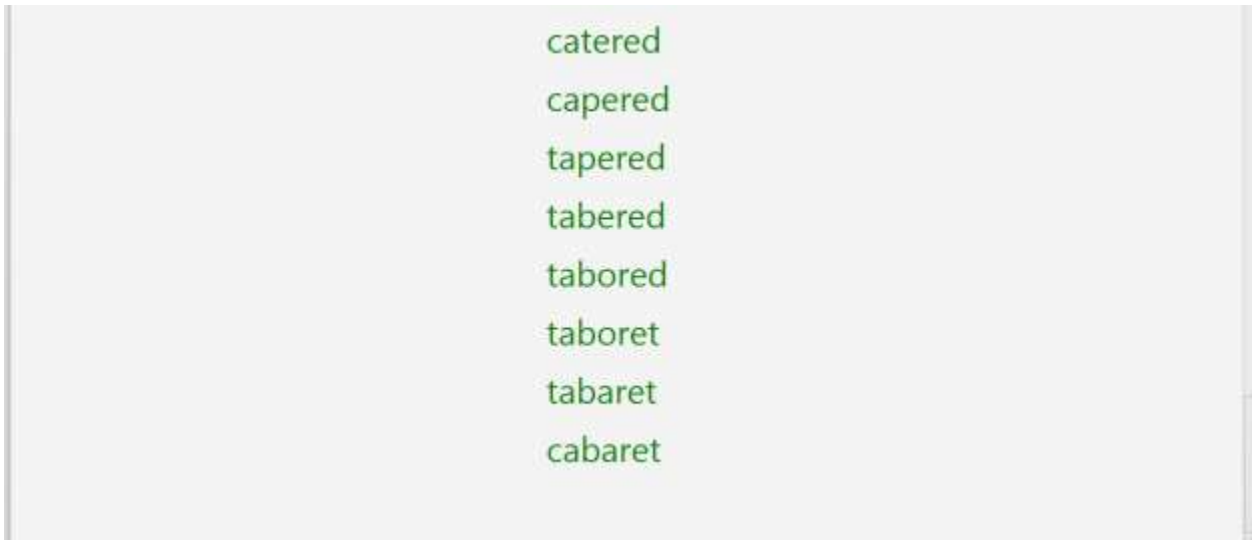
C. A*

3. ATLASES -> CABARET
   A. UCS



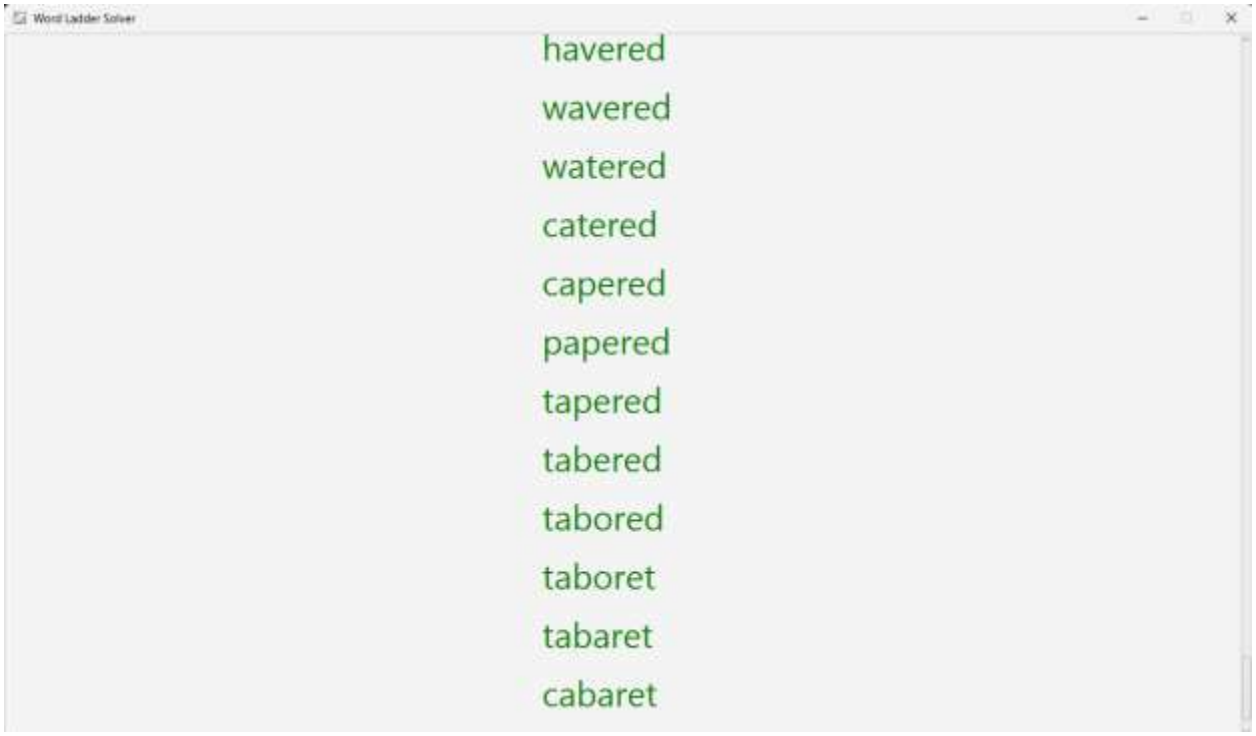

Word Ladder Solver
brakier
beakier
peakier
peckier
pickier
dickier
dickies
hickies
hackies
hackles
heckles
deckles


Word Ladder Solver
deciles
defiles
defiled
deviled
develed
reveled
raveled
ravened
havened
havered
wavered
watered

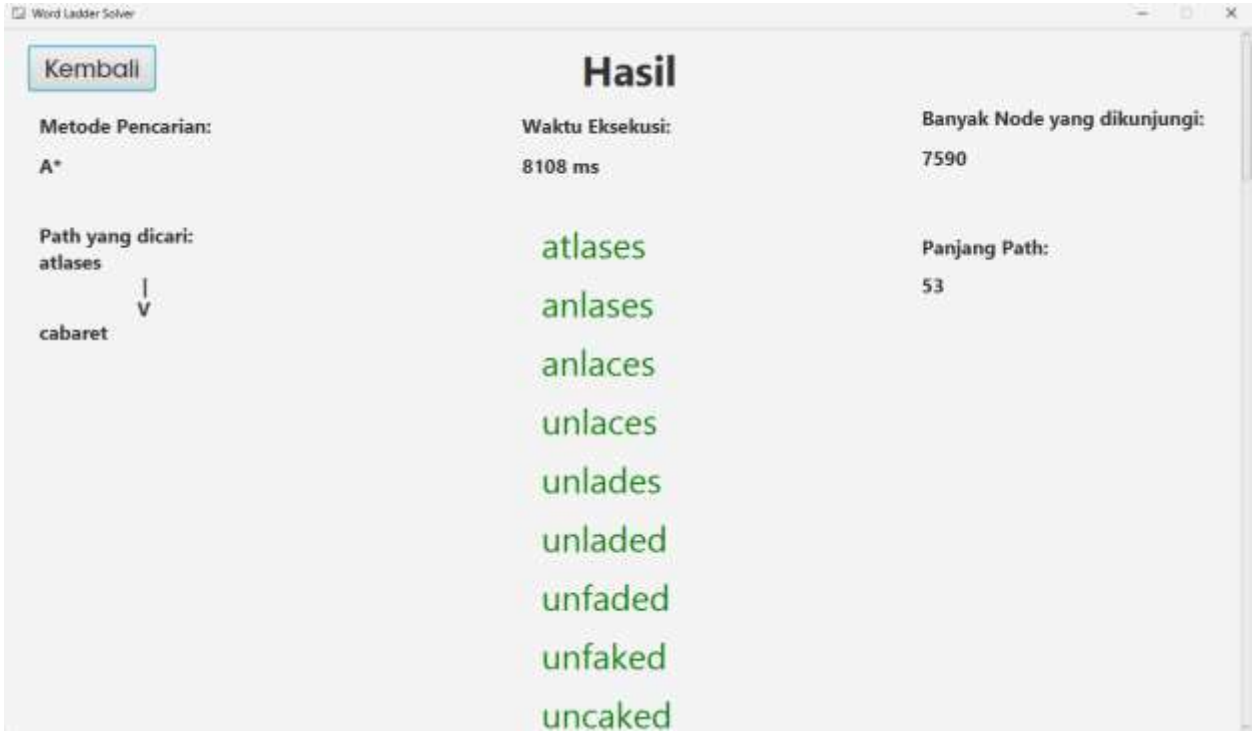


B. Greedy Best First Search

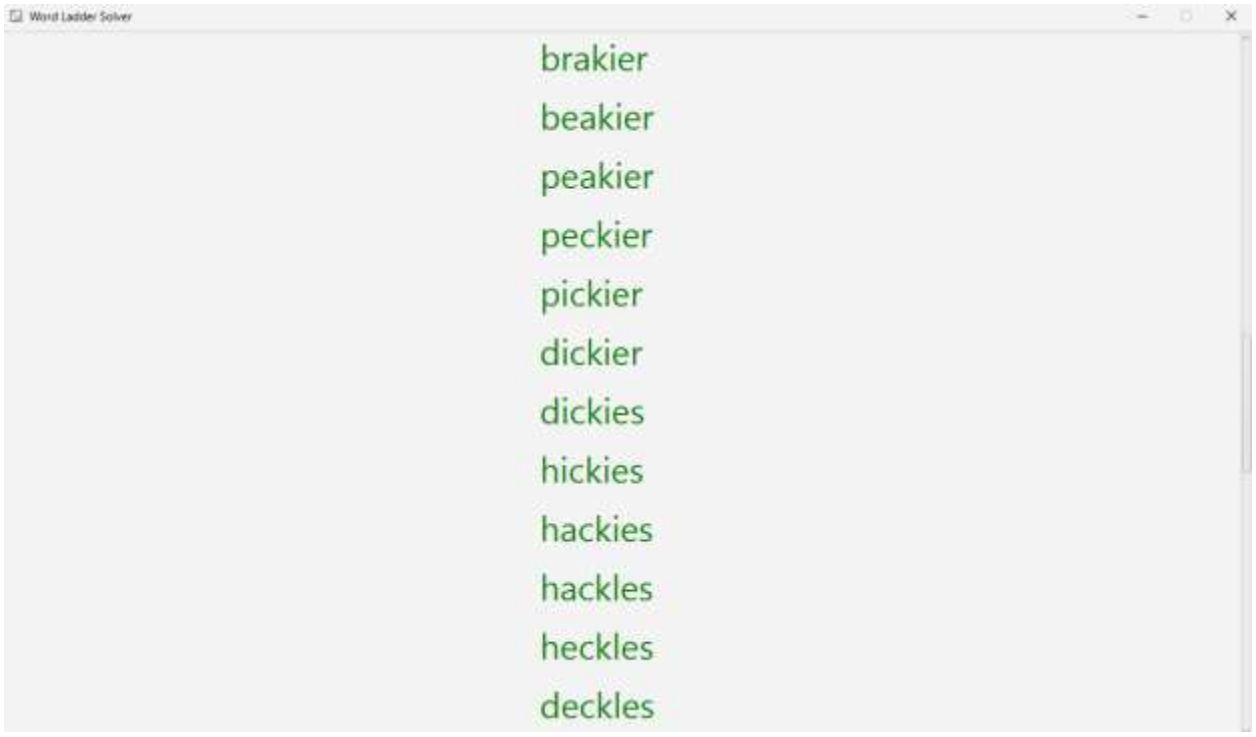
(path di bagian tengah dilewati karena path terlalu panjang)

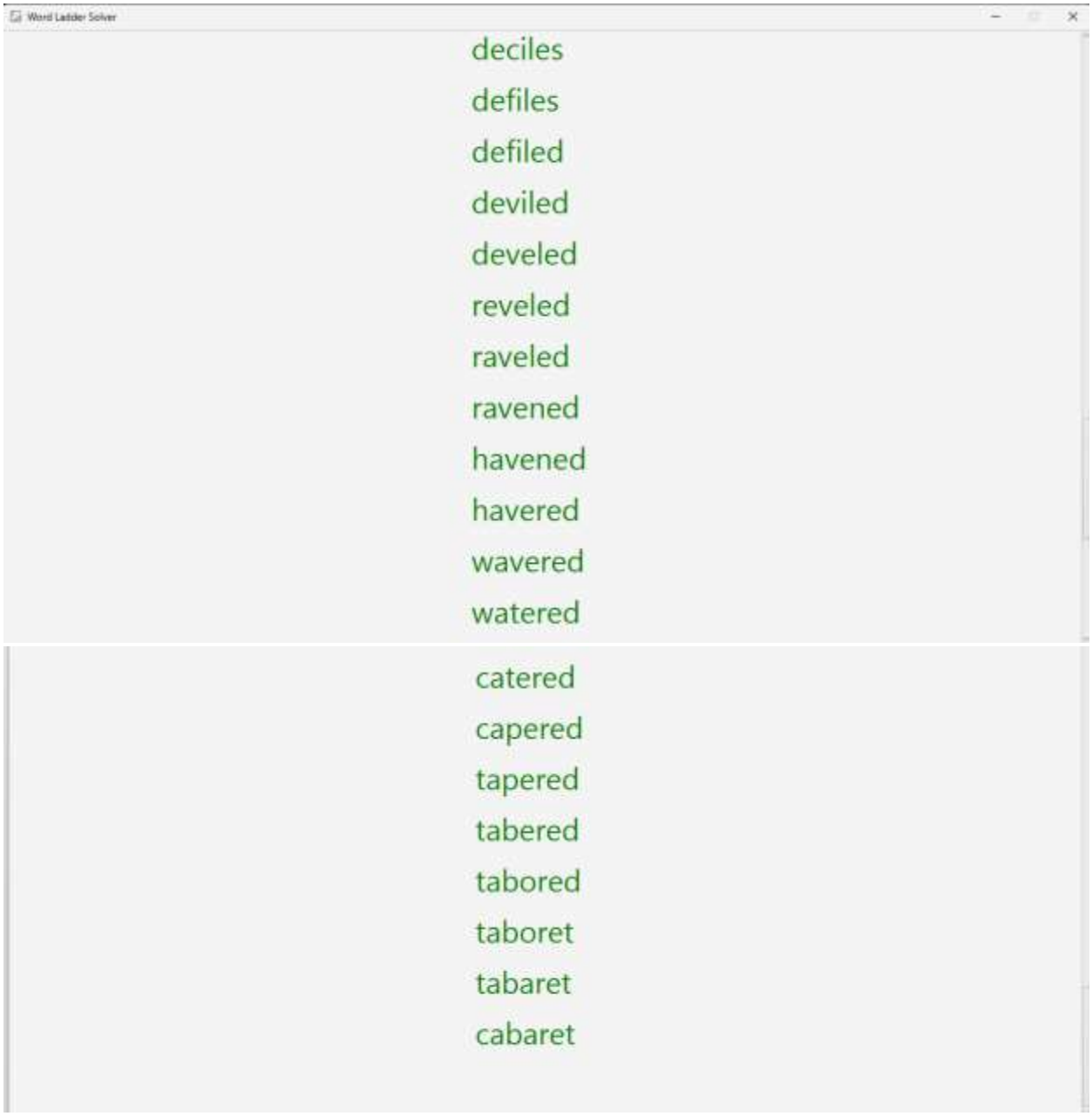

Word Ladder Solver
havered
wavered
watered
catered
capered
papered
tapered
tabered
tabored
taboret
tabaret
cabaret

C. A*


Word Ladder Solver
Kembali
Hasil
Metode Pencarian:
A*
Waktu Eksekusi:
8108 ms
Banyak Node yang dikunjungi:
7590
Path yang dicari:
atlases
|
V
cabaret
atlases
anlases
anlaces
unlaces
unlades
unladed
unfaded
unfaked
uncaked
Panjang Path:
53


Word Ladder Solver
uncakes
uncases
uneases
ureases
creases
cresses
crosses
crosser
crasser
crasher
brasher
brasier


Word Ladder Solver
brakier
beakier
peakier
peckier
pickier
dickier
dickies
hickies
hackies
hackles
heckles
deckles

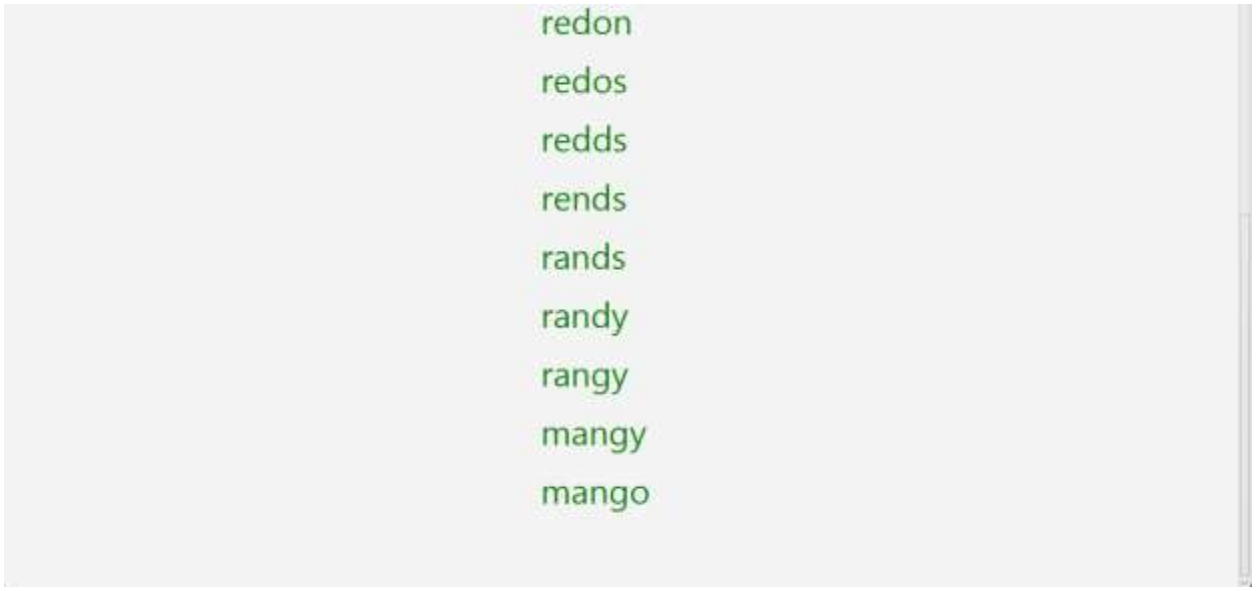

Word Ladder Solver
deciles
defiles
defiled
deviled
develed
reveled
raveled
ravened
havened
havered
wavered
watered
catered
capered
tapered
tabered
tabored
taboret
tabaret
cabaret

4. PONY -> SOCK
   A. UCS

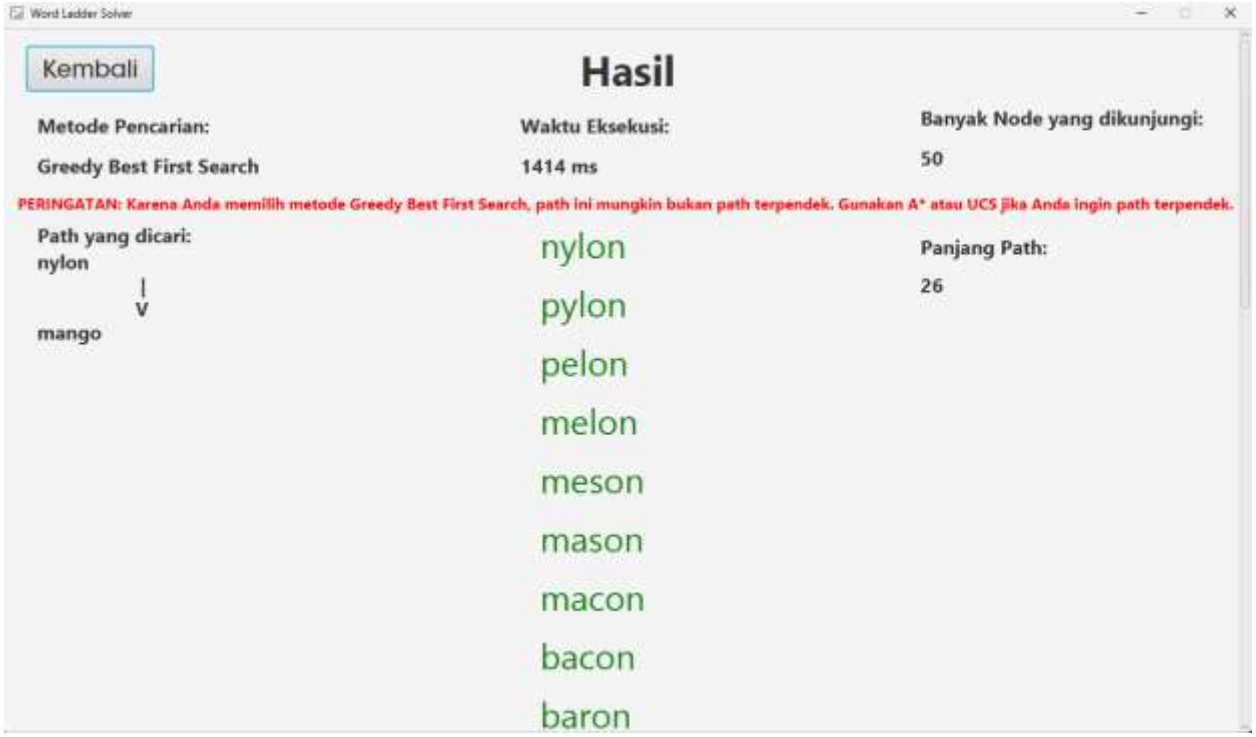
   B. Greedy Best First Search

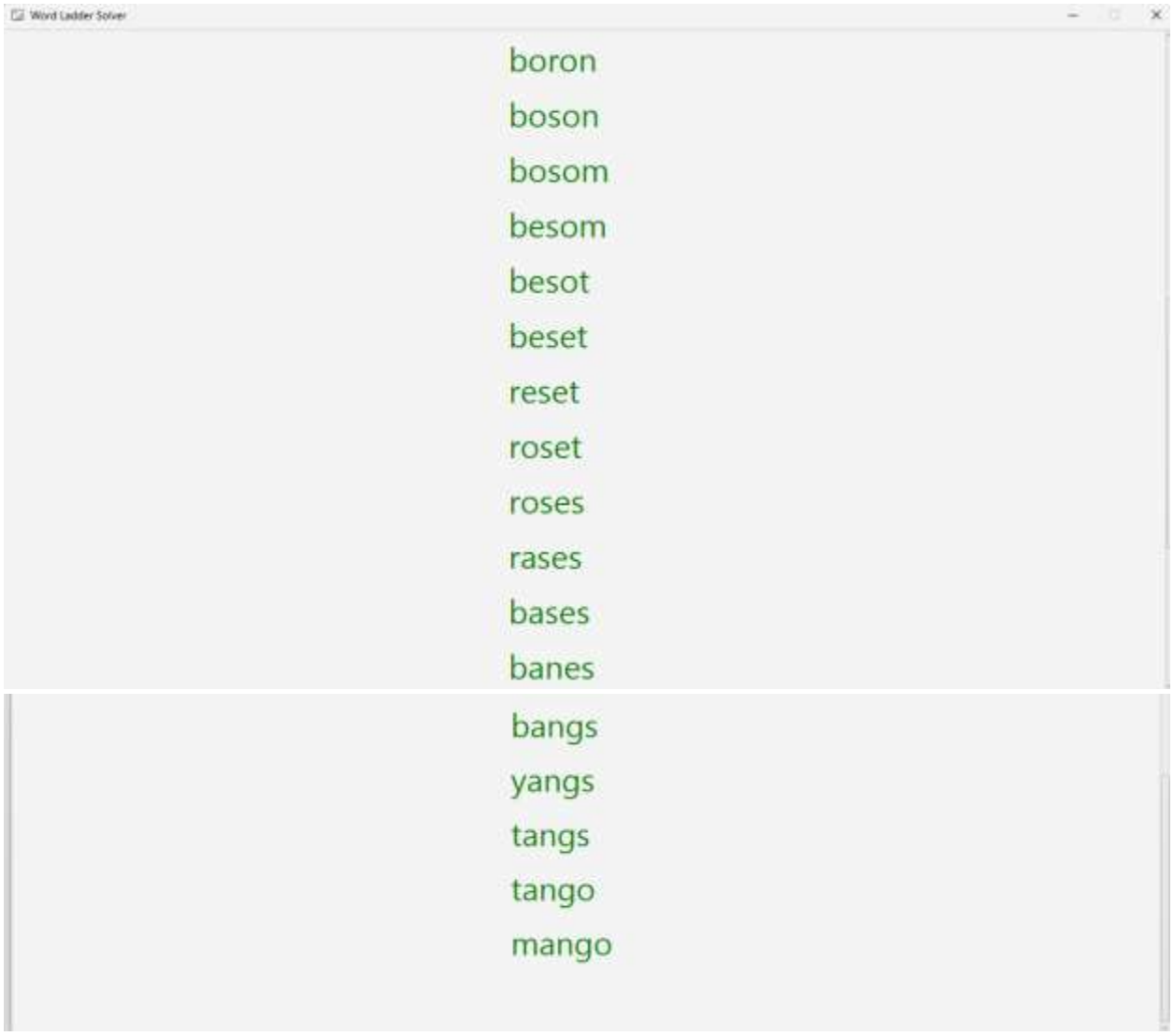
C. A*

5. NYLON -> MANGO
   A. UCS

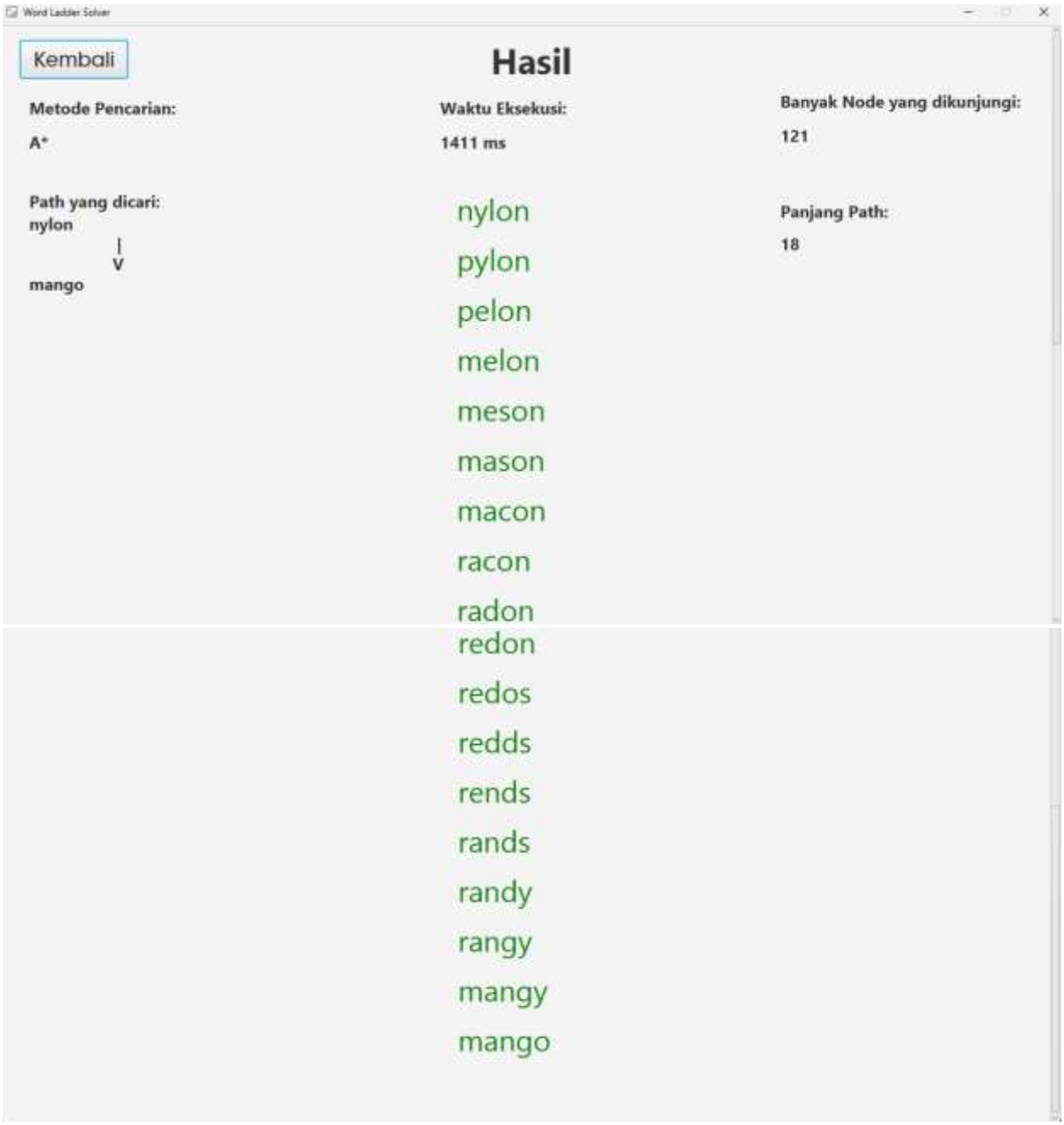


B. Greedy Best First Search




Word Ladder Solver
boron
boson
bosom
besom
besot
beset
reset
roset
roses
rases
bases
banes
bangs
yangs
tangs
tango
mango

C. A*


Word Ladder Solver
Kembali
Hasil
Metode Pencarian:
A*
Waktu Eksekusi:
1411 ms
Banyak Node yang dikunjungi:
121
Path yang dicari:
nylon
|
v
mango
Panjang Path:
18
nylon
pylon
pelon
melon
meson
mason
macon
racon
radon
redon
redos
redds
rends
rands
randy
rangy
mangy
mango

6. KELP -> KELP
   A. UCS

   B. Greedy Best First Search

C. A*


Word Ladder Solver
Kembali
Hasil
Metode Pencarian:
A*
Waktu Eksekusi:
147 ms
Banyak Node yang dikunjungi:
1
Path yang dicari:
kelp
|
V
kelp
kelp
Panjang Path:
1

Bab 6: Analisis Perbandingan Algoritma

Berdasarkan hasil testing pada program, ketiga algoritma tersebut dapat kita analisis sebagai berikut:

A. UCS (ekuivalen dengan BFS) dan A* berhasil menemukan path paling optimal, sedangkan GBFS belum tentu berhasil menemukan path paling optimal
B. GBFS memiliki efisiensi yang paling tinggi karena memiliki waktu eksekusi relatif paling rendah dibandingkan dengan algoritma lain, namun hasil yang diberikan oleh GBFS tidak selalu optimal.
C. A* lebih efisien daripada UCS selain karena keduanya berhasil menemukan path paling optimal, A* mengunjungi banyak node yang jauh lebih sedikit dibandingkan dengan UCS dan waktu yang dibutuhkan tidak jauh berbeda (atau terkadang lebih cepat sedikit dibandingkan dengan UCS).
D. Perbedaan waktu eksekusi ketiga algoritma di atas relatif tidak jauh berbeda.

Oleh karena itu, algoritma yang paling cocok untuk mencari solusi Word Ladder adalah A* karena algoritma tersebut berhasil menemukan path terpendek dari Word Ladder dan merupakan algoritma yang paling efisien dibanding algoritma yang lain

Bab 7: Implementasi Bonus

Bonus yang dikerjakan adalah GUI. GUI program ini menggunakan JavaFX versi 22.0.1
Berikut adalah susunan layar GUI yang dibuat untuk tugas ini.

a. Layar Landing

Keterangan: Tombol “Mulai!” berfungsi untuk berpindah ke layar input utama (dan melakukan pemisahan kata dari dictionary berdasarkan panjang hurufnya)

b. Layar Input Utama

Keterangan:

a. Tombol “Kembali” berfungsi untuk kembali ke layar landing
b. Input teks kata awal dan kata akhir bersifat **case insensitive**.
c. Metode pencarian adalah sebuah dropdown yang berisi pilihan algoritma pencarian pada tugas kecil ini, yaitu UCS, A*, dan Greedy Best First Search.
d. Tombol “Cari” berfungsi untuk melakukan pencarian path menggunakan algoritma yang telah dipilih.

c. Layar Hasil

Keterangan:

a. Tombol “Kembali” berfungsi untuk kembali ke layar input

b. Di layar ini terdapat *scrollbar* untuk melakukan scrolling solusi ke bawah jika path yang diberikan panjang.
c. Terdapat keterangan hasil pencarian yang tertera di dalam layar ini.

Lampiran

Link Repository: https://github.com/DeltDev/Tucil3_13522036

Check list program:

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