Laporan Tugas Kecil 3

IF2211 Strategi Algoritma

Penyelesaian Permainan Word Ladder

Disusun oleh:

Muhamad Rafli Rasyiidin 13522088

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2024

Daftar Isi

BAB 1
Algoritma Uniform Cost Search

Uniform Cost Search atau UCS adalah salah satu algoritma yang digunakan untuk mencari jalur terpendek dalam sebuah graf dengan bobot pada setiap lintasannya. Hasil yang didapatkan oleh algoritma ini dijamin optimal. Algoritma ini akan mengutamakan simpul-simpul dengan bobot terendah untuk dikunjungi. Bobot pada simpul tersebut biasa disebut sebagai *g(n)*. Pada kasus permainan Word Ladder, fungsi *g(n)* didefinisikan sebagai banyaknya langkah yang diperlukan dari simpul awal (*start word*) ke simpul n (kata saat ini). Berikut merupakan langkah-langkah pencarian menggunakan algoritma Uniform Cost Search:

1. Lakukan inisiasi Priority Queue untuk menampung kata yang merupakan simpul hidup dengan bobot sebagai penentu urutannya dan inisiasi Hash Map untuk menampung kata yang telah diekspan.
2. Lakukan validasi terhadap *start word* dan *target word*. Pastikan kedua kata tersebut berada pada kamus.
3. Masukkan *start word* ke dalam priority queue dengan bobot 0.
4. Periksa apakah priority queue masih memiliki simpul di dalamnya. Jika priority queue kosong, maka rute dari *start word* ke *target word* tidak ditemukan dan algoritma berhenti.
5. Lakukan *pop* pada priority queue dan periksa apakah kata yang didapat merupakan *target word*. Jika iya, maka algoritma selesai. Jika tidak, maka lakukan ekspansi pada kata tersebut.
6. Periksa kata hasil ekspansi pada langkah 5. Jika kata yang dihasilkan telah berada pada hash map untuk menampung simpul yang telah diekspan, maka kata tersebut tidak perlu ditambahkan ke dalam priority queue. Jika tidak, tambahkan kata baru tersebut ke dalam priority queue dengan bobotnya saat ini.
7. Ulangi langkah yang dilakukan mulai dari langkah 4.

Seperti yang telah dijelaskan sebelumnya, algoritma UCS selalu menghasilkan solusi yang optimal. Namun, waktu yang diperlukan untuk mencari solusi tersebut cukup lama, lebih lama daripada GBFS dan A*. Hal tersebut dikarenakan algoritma UCS harus mengunjungi simpul lebih banyak daripada algoritma lainnya. Pada kasus permainan Word Ladder, jika *target word* berjarak 5 langkah dari *start word*, UCS akan melakukan ekspansi terhadap seluruh kata yang berjarak 1-4 langkah dari *start word* terlebih dahulu, sebelum akhirnya bisa mengekspansi kata yang berjarak 5 langkah.

Pada algoritma UCS, simpul hidup yang dihasilkan oleh simpul ekspan akan selalu memiliki bobot yang lebih besar atau sama dengan bobot simpul yang telah berada di dalam priority queue. Hal tersebut dikarenakan algoritma UCS akan mengiterasi seluruh simpul dengan bobot terendah dan ketika selesai, akan melanjutkan iterasi ke bobot terendah selanjutnya dimana simpul yang memiliki bobot terendah selanjutnya adalah hasil ekspansi dari simpul dengan bobot terendah sebelumnya. Sebagai ilustrasi, misalkan terdapat 5 hasil ekspansi dari suatu kata. Setiap

hasil tersebut pasti memiliki bobot 1. Kemudian, algoritma UCS akan mengiterasi simpul dengan bobot 1 yang menghasilkan simpul-simpul hidup dengan bobot 2. Kemudian, setelah semua simpul dengan bobot 1 selesai diekspansi, seluruh simpul yang tersisa di priority queue akan memiliki bobot 2. Selanjutnya, algoritma UCS akan mulai mengiterasi simpul dengan bobot 2 pada priority queue dan menghasilkan simpul berbobot 3. Hal tersebut akan terus berulang hingga *target word* diekspansi. Dengan penambahan bobot yang seperti itu, pengurutan prioritas tidak terlalu berguna karena simpul hidup yang baru ditambahkan akan selalu ditempatkan di akhir priority queue. Selain itu, hal tersebut juga menyebabkan algoritma UCS berperilaku sama dengan algoritma BFS karena hasil ekspansi kedua algoritma tersebut akan selalu ditempatkan di akhir kontainer.

BAB 2
Algoritma Greedy Best First Search

Algoritma Greedy Best First Search, GBFS, merupakan salah satu algoritma *informed search* untuk menemukan rute terpendek. Algoritma ini memanfaatkan fungsi evaluasi *f(n)* pada setiap simpul untuk mencapai tujuan. Fungsi *f(n)* pada setiap simpul akan memiliki nilai yang sama dengan fungsi heuristik *h(n)*. Fungsi heuristik merupakan perkiraan bobot yang diperlukan dari simpul n ke simpul tujuan. Pada kasus permainan Word Ladder, fungsi heuristik dari setiap kata dapat ditentukan dengan menghitung jumlah huruf yang berbeda dengan *target word*. Algoritma GBFS memiliki waktu proses yang sangat cepat karena tidak perlu mengunjungi setiap simpul dengan bobot yang sama. Namun, solusi yang dihasilkan oleh algoritma ini tidak dijamin optimal. Hal tersebut dikarenakan algoritma GBFS akan terus mencari *target word* pada satu rute dari simpul tertentu dan tidak berpindah ke rute simpul lain. Namun, pada kasus permainan Word Ladder, algoritma GBFS memiliki kemungkinan untuk berpindah ke rute simpul lain jika memang simpul tersebut memiliki bobot terendah. Berikut merupakan langkah-langkah pencarian menggunakan algoritma Greedy Best First Search:

1. Lakukan inisiasi Priority Queue untuk menampung kata yang merupakan simpul hidup dengan bobot sebagai penentu urutannya dan inisiasi Hash Map untuk menampung kata yang telah diekspan.
2. Lakukan validasi terhadap *start word* dan *target word*. Pastikan kedua kata tersebut berada pada kamus.
3. Masukkan *start word* ke dalam priority queue dengan perhitungan bobot menggunakan fungsi heuristik.
4. Periksa apakah priority queue masih memiliki simpul di dalamnya. Jika priority queue kosong, maka rute dari *start word* ke *target word* tidak ditemukan dan algoritma berhenti.
5. Lakukan *pop* pada priority queue dan periksa apakah kata yang didapat merupakan *target word*. Jika iya, maka algoritma selesai. Jika tidak, maka lakukan ekspansi pada kata tersebut.
6. Periksa kata hasil ekspansi pada langkah 5. Jika kata yang dihasilkan telah berada pada hash map untuk menampung simpul yang telah diekspan, maka kata tersebut tidak perlu ditambahkan ke dalam priority queue. Jika tidak, tambahkan kata baru tersebut ke dalam priority queue dengan bobot yang dihitung menggunakan fungsi heuristik.
7. Ulangi langkah yang dilakukan mulai dari langkah 4.

BAB 3
Algoritma A*

Algoritma A* merupakan salah satu algoritma *informed search* yang digunakan untuk mencari rute terpendek. Algoritma ini menggabungkan konsep UCS dan GBFS. Bobot pada algoritma A* ditentukan dari penjumlahan fungsi *g(n)* pada UCS dengan fungsi heuristik *h(n)* pada GBFS. Algoritma A* dijamin mendapatkan solusi yang optimal jika fungsi heuristik yang digunakan *admissible* yaitu nilai heuristik pada setiap simpul n tidak pernah melebihi bobot optimal sebenarnya dari simpul n ke simpul tujuan. Pada kasus permainan Word Ladder, fungsi heuristik didefinisikan sebagai banyaknya huruf yang berbeda antara kata n dengan kata tujuan. Berikut merupakan langkah-langkah pencarian menggunakan algoritma A*:

1. Lakukan inisiasi priority queue untuk menampung kata yang merupakan simpul hidup dengan bobot sebagai penentu urutannya dan inisiasi hash map untuk menampung kata yang telah diekspan.
2. Lakukan validasi terhadap *start word* dan *target word*. Pastikan kedua kata tersebut berada pada kamus.
3. Masukkan *start word* ke dalam priority queue dengan bobot sama dengan fungsi heuristik dari *start word* ke *target word*.
4. Periksa apakah priority queue masih memiliki simpul di dalamnya. Jika priority queue kosong, maka rute dari *start word* ke *target word* tidak ditemukan dan algoritma berhenti.
5. Lakukan *pop* pada priority queue dan periksa apakah kata yang didapat merupakan *target word*. Jika iya, maka algoritma selesai. Jika tidak, maka lakukan ekspansi pada kata tersebut.
6. Periksa kata hasil ekspansi pada langkah 5. Jika kata yang dihasilkan telah berada pada hash map untuk menampung simpul yang telah diekspan, maka kata tersebut tidak perlu ditambahkan ke dalam priority queue. Jika tidak, tambahkan kata baru tersebut ke dalam priority queue dengan bobot ditentukan dari penjumlahan antara jumlah langkah dari *start word* ke kata saat ini (*g(n)*) dengan jumlah karakter yang berbeda dari kata saat ini dengan *target word* (*h(n)*).
7. Ulangi langkah yang dilakukan mulai dari langkah 4.

Seperti yang telah disebutkan sebelumnya, algoritma A* selalu menghasilkan solusi yang optimal dengan waktu yang cepat. Hal tersebut dikarenakan algoritma A* menggunakan fungsi heuristik dengan mempertimbangkan jumlah langkah minimal yang ditempuh dari *start word* ke kata saat ini.

BAB 4
Implementasi dan Analisis

Program ini ditulis dalam bahasa Java dengan memanfaatkan *package* yang terdapat di dalamnya. Terdapat beberapa kelas yang dibuat untuk menjalankan program ini. Pertama, kelas UCS yang berisi algoritma pencarian Uniform Cost Search. Kedua, kelas GBFS yang berisi algoritma pencarian Greedy Best First Search. Ketiga, kelas AStar yang berisi algoritma pencarian A*. Keempat, kelas Node yang digunakan sebagai simpul pada pencarian. Kelima, kelas Controller yang digunakan untuk inisialisasi kamus dan beberapa hal lainnya. Keenam, kelas ReturnElement yang digunakan untuk mengembalikan hasil dari algoritma pencarian ke GUI. Terakhir, kelas GUI yang digunakan untuk menampilkan GUI.

4.1. Implementasi Algoritma UCS

Algoritma UCS memanfaatkan priority queue untuk menampung simpul hidup. Struktur data map digunakan untuk menyimpan simpul yang diekspan dengan tujuan agar pengaksesan dan pengecekan simpul yang telah diekspan lebih cepat. Struktur data map juga digunakan untuk menyimpan bobot setiap simpul dengan alasan untuk pengecekan lebih cepat.

```java
package algorithm;
import java.util.*;

public class UCS extends Controller{
    private PriorityQueue<Node> listPath;
    private boolean found;
    private Map<String, Boolean> expandedNode;
    private Map<Node, Integer> costList;

    public UCS() {
        super();
        this.listPath = new PriorityQueue<Node>(10, new CostComparator());
        this.costList = new HashMap<Node, Integer>();
        this.expandedNode = new HashMap<String, Boolean>();
        this.found = false;
    }

    public ReturnElement ucs(String start, String target) {
        if (start.length() != target.length()) {
            System.out.println("Gabisa bro kalo panjangnya beda");
        }
        else {
            Node startParent = new Node(null, start, 0);
            this.listPath.add(startParent);
            this.costList.put(startParent, 0);
            long begin = System.currentTimeMillis();
            Node targetNode = new Node();
            while (!this.found) {
                Node temp = listPath.poll();
                if (temp.getWord().equals(target)) {
                    found = true;
                    targetNode.copy(temp);
                }
                else {
                    ucsLoop(temp, target);
                    expandedNode.put(temp.getWord(), true);
                }
            }
            long end = System.currentTimeMillis();
            System.out.println(this.found);
            System.out.println(targetNode.getWord());
            List<String> pathList = new ArrayList<String>(targetNode.getPath());
            Collections.reverse(pathList);
            System.out.println(pathList);
            System.out.printf("Time elapsed: %d\n", end-begin);
            return new ReturnElement(pathList, end-begin);
        }
        return null;
    }

    public void ucsLoop(Node el, String end) {
        for (int i = 0; i < el.getWord().length() && !this.found; i++) {
            char charAt = el.getWord().charAt(i);
            for (char c = 'a'; c <= 'z' && !this.found;c++) {
                if (c != charAt) {
                    String word = el.getWord().substring(0, i) + c + el.getWord().substring(i + 1);
                    if (this.listWord.get(word) != null && expandedNode.get(word) == null) {
                        Node current = new Node(el, word, this.costList.get(el));
                        this.costList.put(current, current.getCost()+1);
                        this.listPath.add(current);
                    }
                }
            }
        }
    }
}
```

4.2. Implementasi Algoritma Greedy Best First Search

Algoritma GBFS memanfaatkan priority queue untuk menampung simpul hidup. Struktur data map digunakan untuk menyimpan simpul yang diekspan dengan tujuan agar pengaksesan dan pengecekan simpul yang telah diekspan lebih cepat. Tidak seperti UCS dan A*, algoritma GBFS tidak perlu mengecek bobot dari simpul *parent*-nya untuk menentukan bobotnya saat ini karena bobot saat ini ditentukan oleh perbedaan huruf dari kata saat ini ke kata tujuan sehingga struktur data mpa tidak digunakan untuk menampung *cost*.

```java
package algorithm;
import java.util.*;

public class GBFS extends Controller {
    private PriorityQueue<Node> queue;
    private List<String> expandedNode;
    private boolean found;

    public GBFS() {
        this.queue = new PriorityQueue<Node>(10, new CostComparator());
        this.expandedNode = new ArrayList<String>();
        this.found = false;
    }

    public int getCost(String word, String target) {
        int count = 0;
        for (int i = 0; i < target.length(); i++) {
            if (word.charAt(i) != target.charAt(i)) {
                count++;
            }
        }
        return count;
    }

    public ReturnElement gbfs(String start, String target) {
        if (start.length() != target.length()) {
            System.out.println("Gabisa bro kalo panjangnya beda");
        }
        else {
            Node startParent = new Node(null, start, getCost(start, target));
            this.queue.add(startParent);
            long begin = System.currentTimeMillis();

            Node targetEl = new Node();
            while (!this.found) {
                Node temp = queue.poll();
                if (temp.getWord().equals(target)) {
                    found = true;
                    targetEl.copy(temp);
                }
                else {
                    gbfsLoop(temp, target);
                    expandedNode.add(temp.getWord());
                }
            }
            long end = System.currentTimeMillis();
            System.out.println(this.found);
            System.out.println(targetEl.getWord());
            List<String> pathList = new ArrayList<String>(targetEl.getPath());
            Collections.reverse(pathList);
            System.out.println(pathList);
            System.out.printf("Time elapsed: %d ms\n", end-begin);
            return new ReturnElement(pathList, end-begin);
        }
        return null;
    }

    public void gbfsLoop(Node el, String end) {
        for (int i = 0; i < el.getWord().length() && !this.found; i++) {
            char charAt = el.getWord().charAt(i);
            for (char c = 'a'; c <= 'z' && !this.found;c++) {
                if (c != charAt) {
                    String word = el.getWord().substring(0, i) + c + el.getWord().substring(i + 1);
                    if (this.listWord.get(word) != null && !expandedNode.contains(word)) {
                        Node current = new Node(el, word, getCost(word, end));
                        this.queue.add(current);
                    }
                }
            }
        }
    }
}
```

4.3. Implementasi Algoritma A*

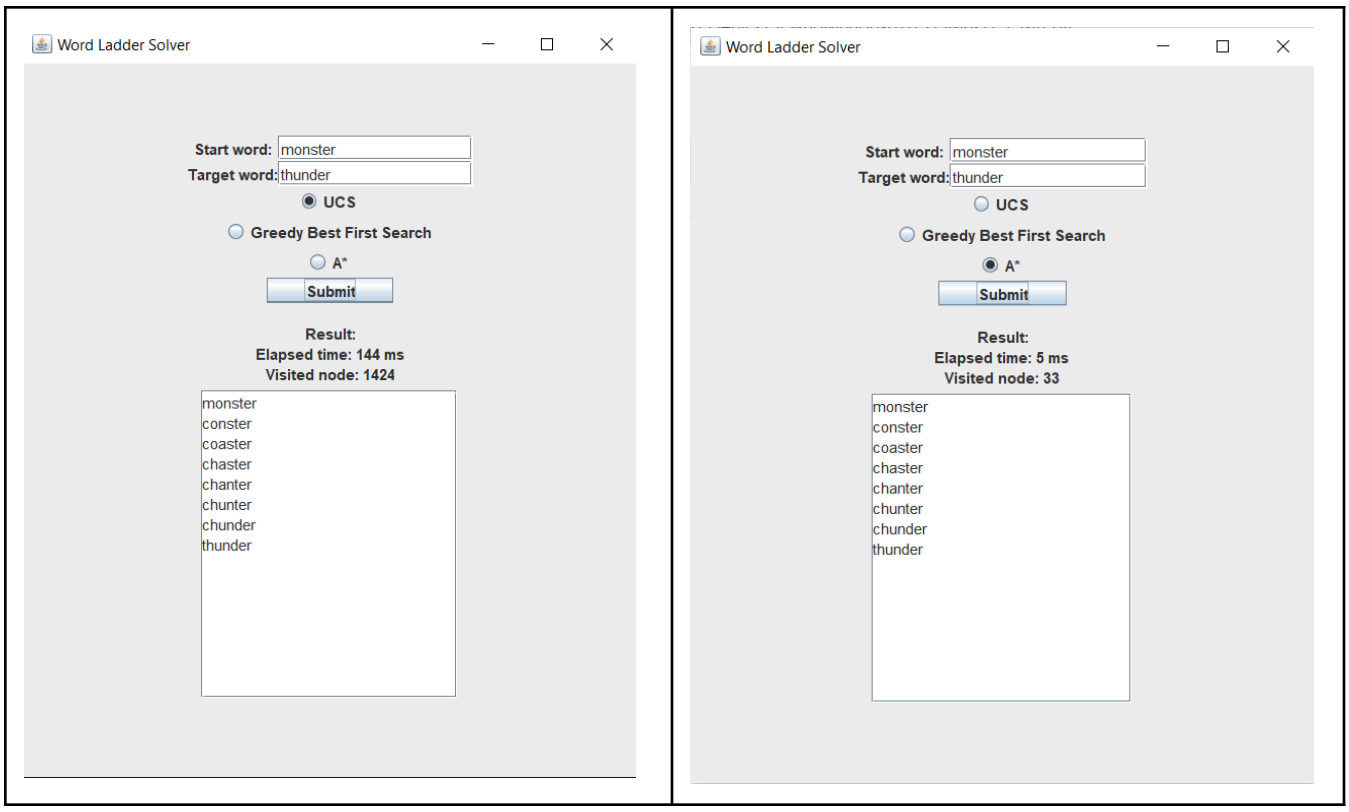
Algoritma A* memanfaatkan priority queue untuk menampung simpul hidup. Struktur data map digunakan untuk menyimpan simpul yang diekspan dengan tujuan agar pengaksesan dan pengecekan simpul yang telah diekspan lebih cepat. Struktur data map juga digunakan untuk menyimpan bobot setiap simpul dengan alasan untuk pengecekan lebih cepat. Secara keseluruhan implementasi algoritma A* sama seperti UCS, yang membedakannya hanyalah penentuan *cost*.

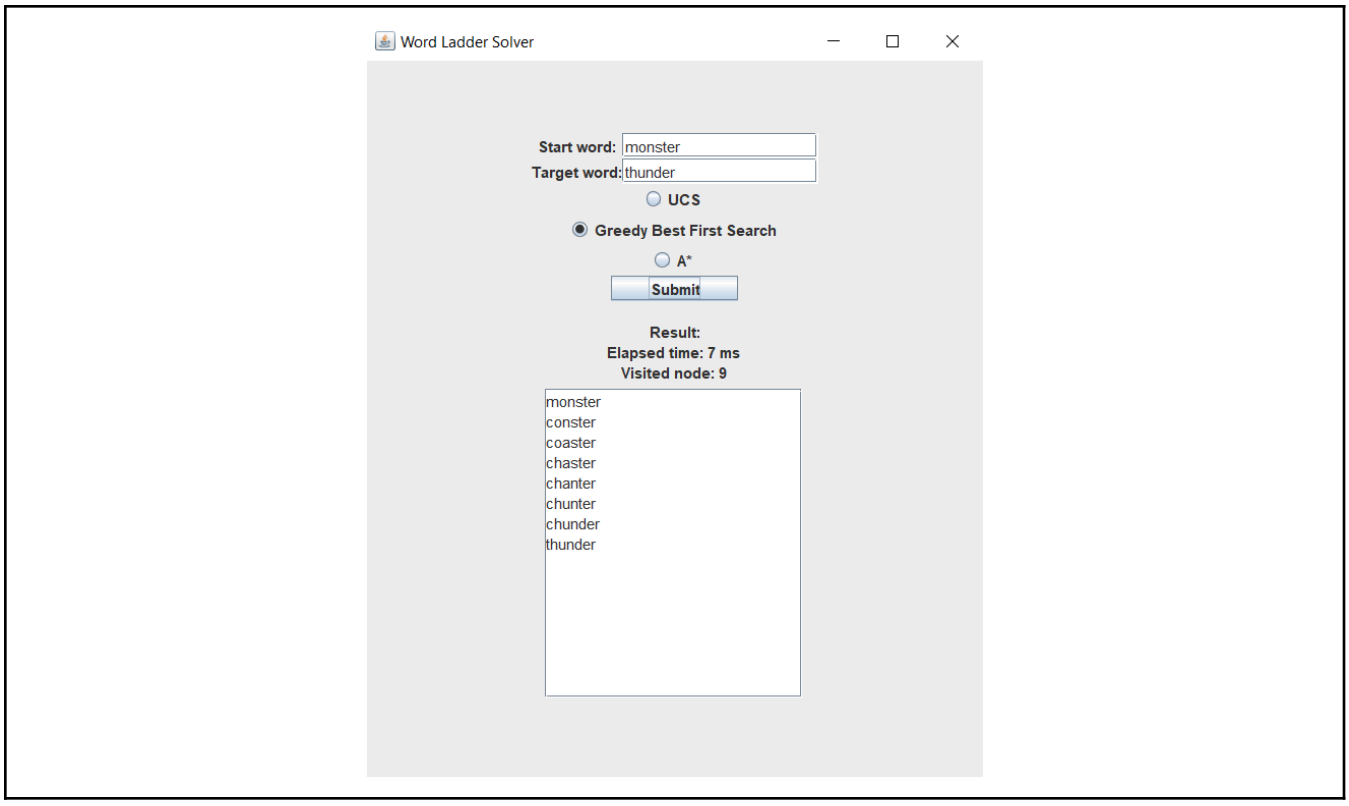
```java
package algorithm;
import java.util.*;

public class AStar extends GBFS {
    private PriorityQueue<Node> queue;
    private Map<String, Boolean> expandedNode;
    private Map<Node, Integer> costList;
    private boolean found;

    public AStar() {
        this.queue = new PriorityQueue<Node>(10, new CostComparator());
        this.costList = new HashMap<Node, Integer>();
        this.expandedNode = new HashMap<String, Boolean>();
        this.found = false;
    }

    public int getCost(String word, String target) {
        int count = 0;
        for (int i = 0; i < target.length(); i++) {
            if (word.charAt(i) != target.charAt(i)) {
                count++;
            }
        }
        return count;
    }

    public ReturnElement aStar(String start, String target) {
        if (start.length() != target.length()) {
            System.out.println("Gabisa bro kalo panjangnya beda");
        }
        else {
            Node startParent = new Node(null, start, 0);
            this.queue.add(startParent);
            this.costList.put(startParent, getCost(start, target));
            long begin = System.currentTimeMillis();
            Node targetNode = new Node();
            while (!this.found) {
                Node temp = queue.poll();
                if (temp.getWord().equals(target)) {
                    found = true;
                    targetNode.copy(temp);
                }
                else {
                    aStarLoop(temp, target);
                    expandedNode.put(temp.getWord(), true);
                }
            }
            long end = System.currentTimeMillis();
            System.out.println(this.found);
            System.out.println(targetNode.getWord());
            List<String> pathList = new ArrayList<String>(targetNode.getPath());
            Collections.reverse(pathList);
            System.out.println(pathList);
            System.out.printf("Time elapsed: %d ms\n", end-begin);
            return new ReturnElement(pathList, end-begin);
        }
        return null;
    }

    public void aStarLoop(Node el, String end) {
        for (int i = 0; i < el.getWord().length() && !this.found; i++) {
            char charAt = el.getWord().charAt(i);
            for (char c = 'a'; c <= 'z' && !this.found;c++) {
                if (c != charAt) {
                    String word = el.getWord().substring(0, i) + c + el.getWord().substring(i + 1);
                    if (this.listWord.get(word) != null && expandedNode.get(word) == null) {
                        int g_n_cost = this.costList.get(el);
                        Node current = new Node(el, word, g_n_cost + getCost(word, end));
                        this.costList.put(current, g_n_cost + 1);
                        this.queue.add(current);
                    }
                }
            }
        }
    }
}
```

4.4. Pengujian

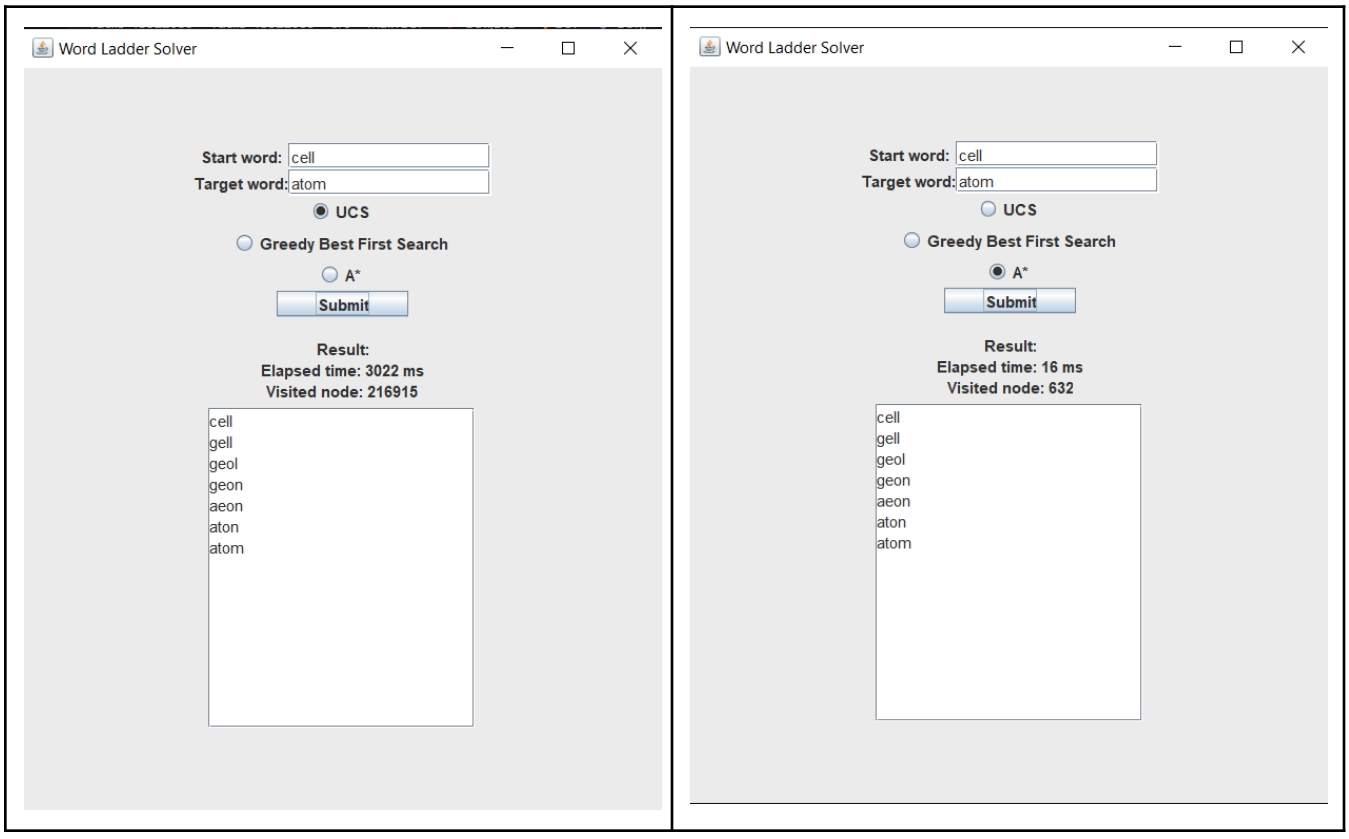
Kamus yang digunakan dalam pengujian berasal dari link berikut**.** Di bawah ini merupakan hasil pengujian dari ketiga algoritma:

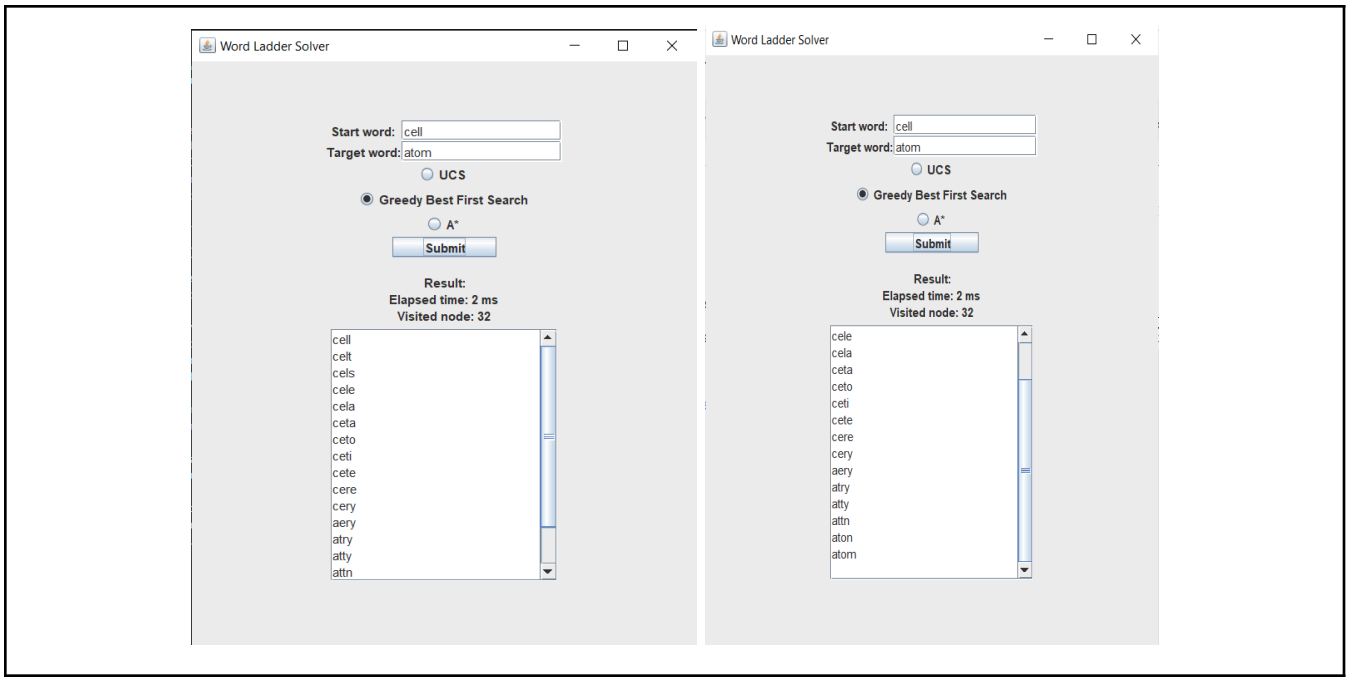
1. monster → thunder



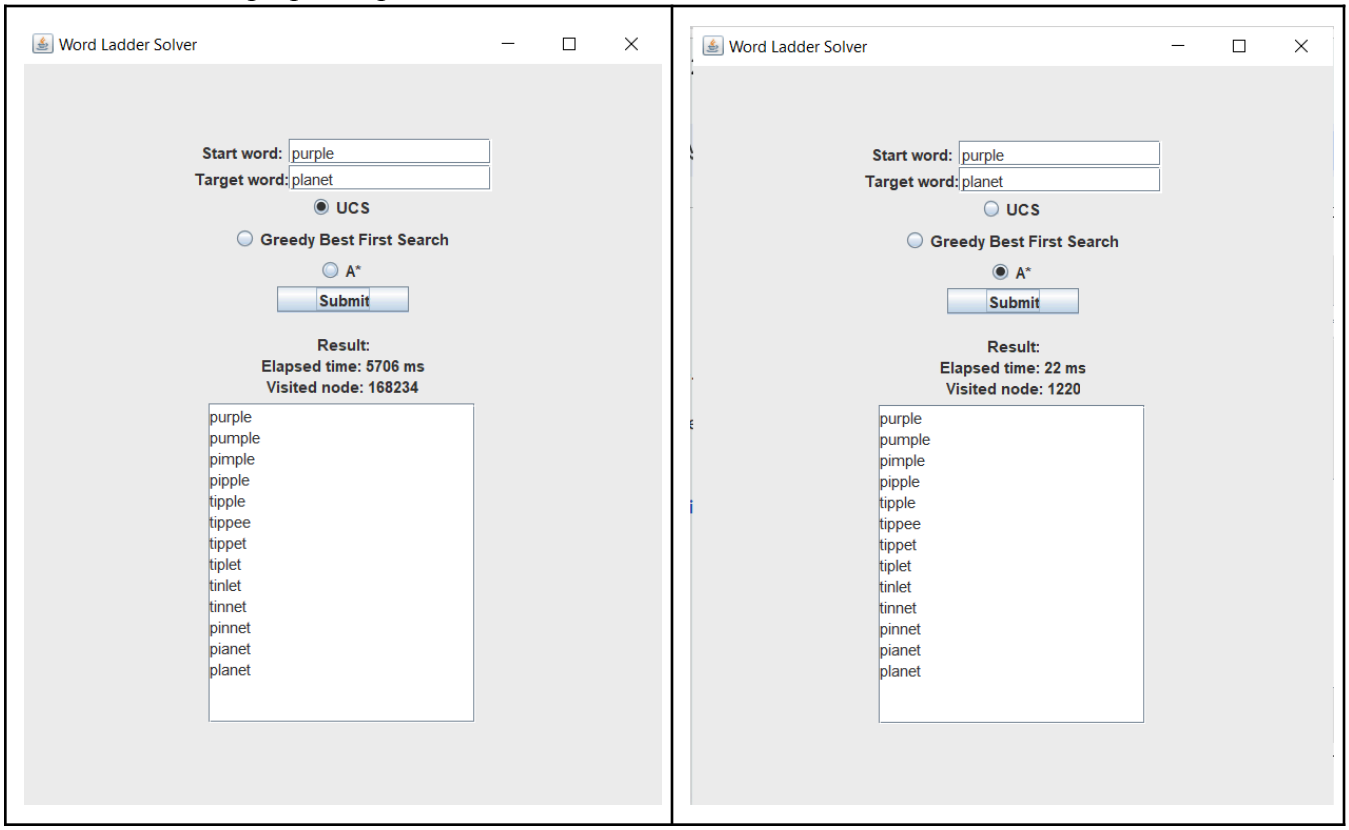

Word Ladder Solver
Start word: monster
Target word: thunder
UCS
Greedy Best First Search
A*
Submit
Result:
Elapsed time: 7 ms
Visited node: 9
monster
conster
coaster
chaster
chanter
chunter
chunder
thunder

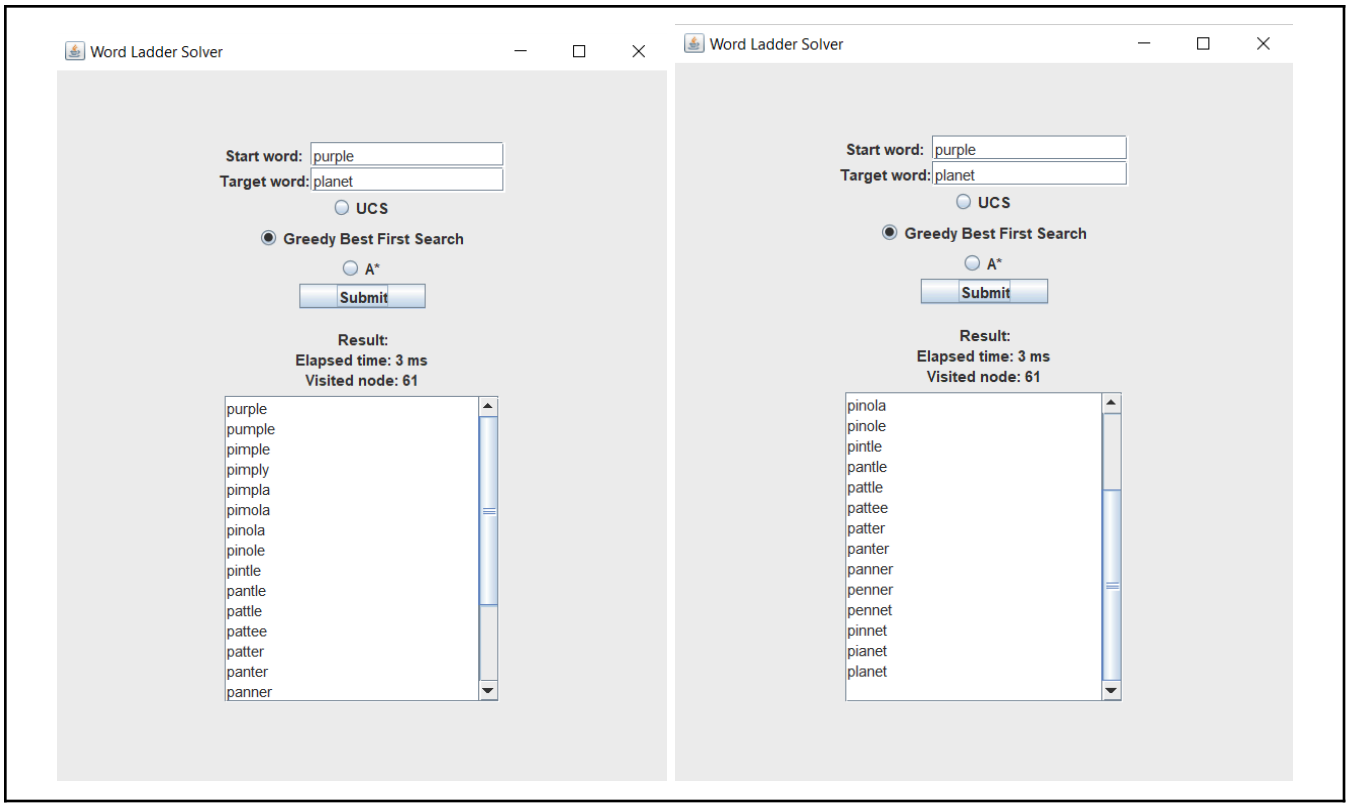
2. cell → atom


Word Ladder Solver
Start word: cell
Target word: atom
UCS
Greedy Best First Search
A*
Submit
Result:
Elapsed time: 3022 ms
Visited node: 216915
cell
gell
geol
geon
aeon
aton
atom
Word Ladder Solver
Start word: cell
Target word: atom
UCS
Greedy Best First Search
A*
Submit
Result:
Elapsed time: 16 ms
Visited node: 632
cell
gell
geol
geon
aeon
aton
atom

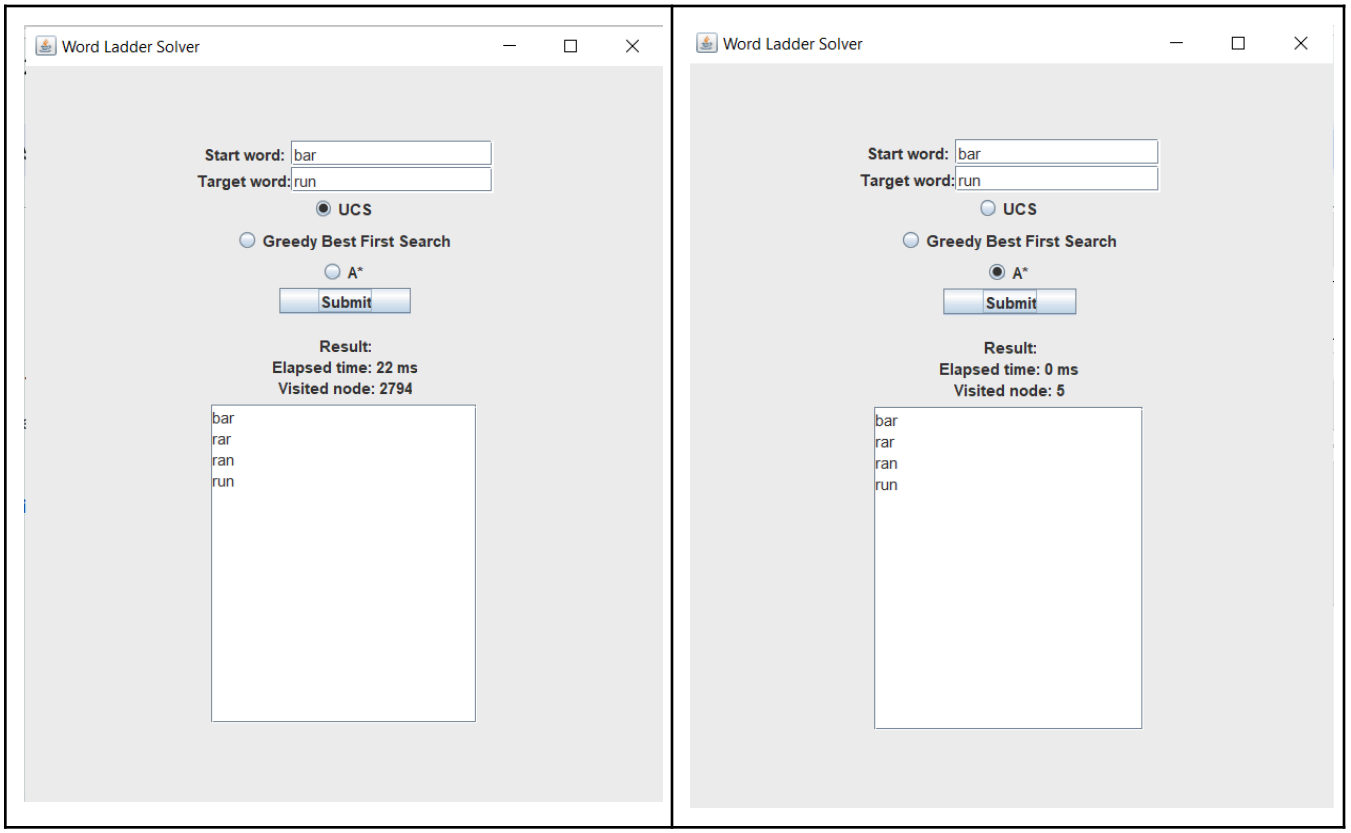


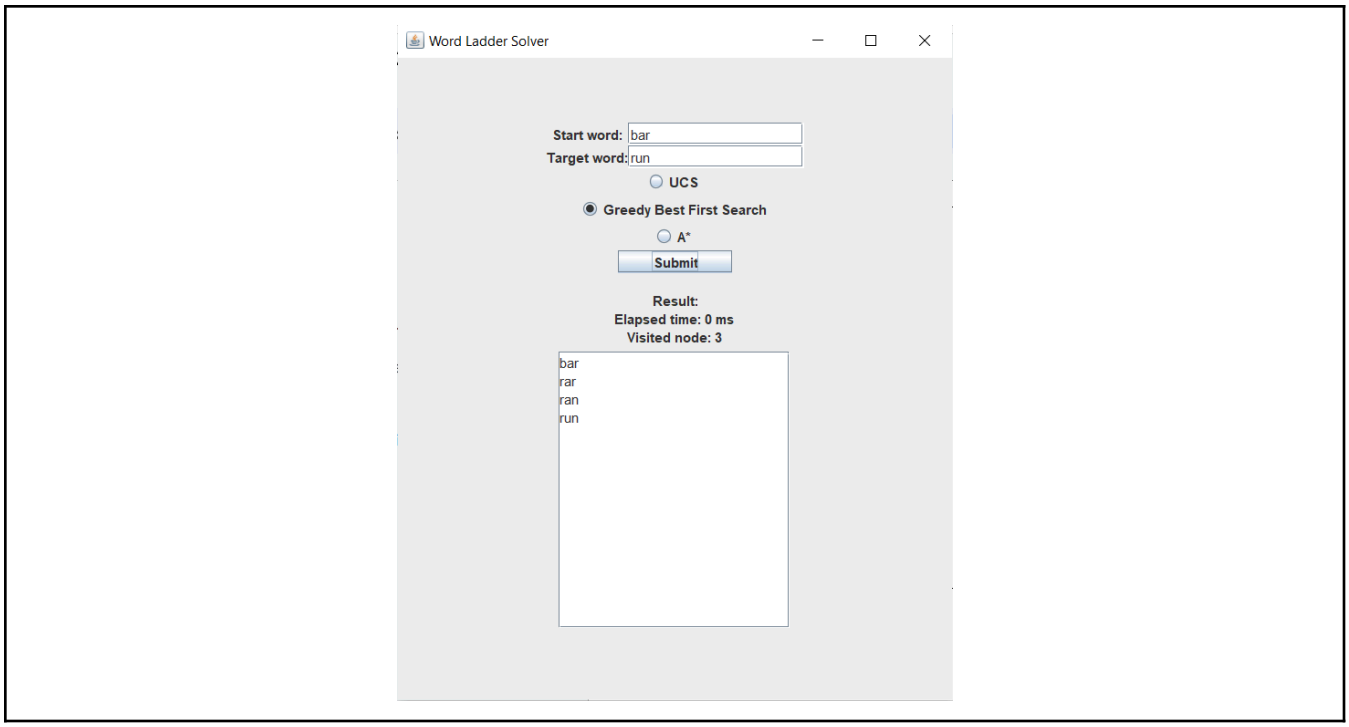
3. purple → planet



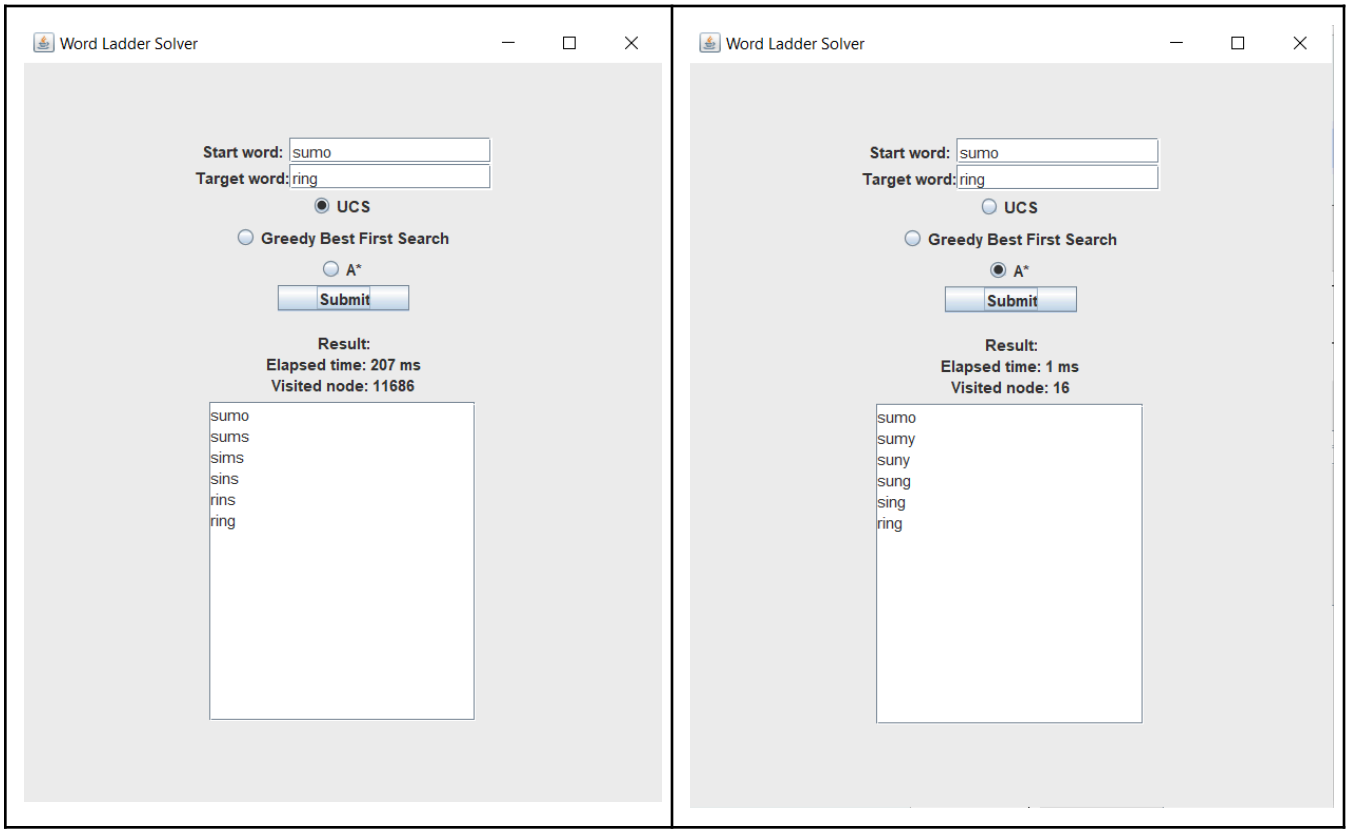

Word Ladder Solver
Start word: purple
Target word: planet
UCS
Greedy Best First Search
A*
Submit
Result:
Elapsed time: 3 ms
Visited node: 61
purple
pumple
pimple
pimply
pimpla
pimola
pinola
pinole
pintle
pantle
pattle
pattee
patter
panter
panner
Word Ladder Solver
Start word: purple
Target word: planet
UCS
Greedy Best First Search
A*
Submit
Result:
Elapsed time: 3 ms
Visited node: 61
pinola
pinole
pintle
pantle
pattle
pattee
patter
panter
panner
penner
pennet
pinnet
pianet
planet

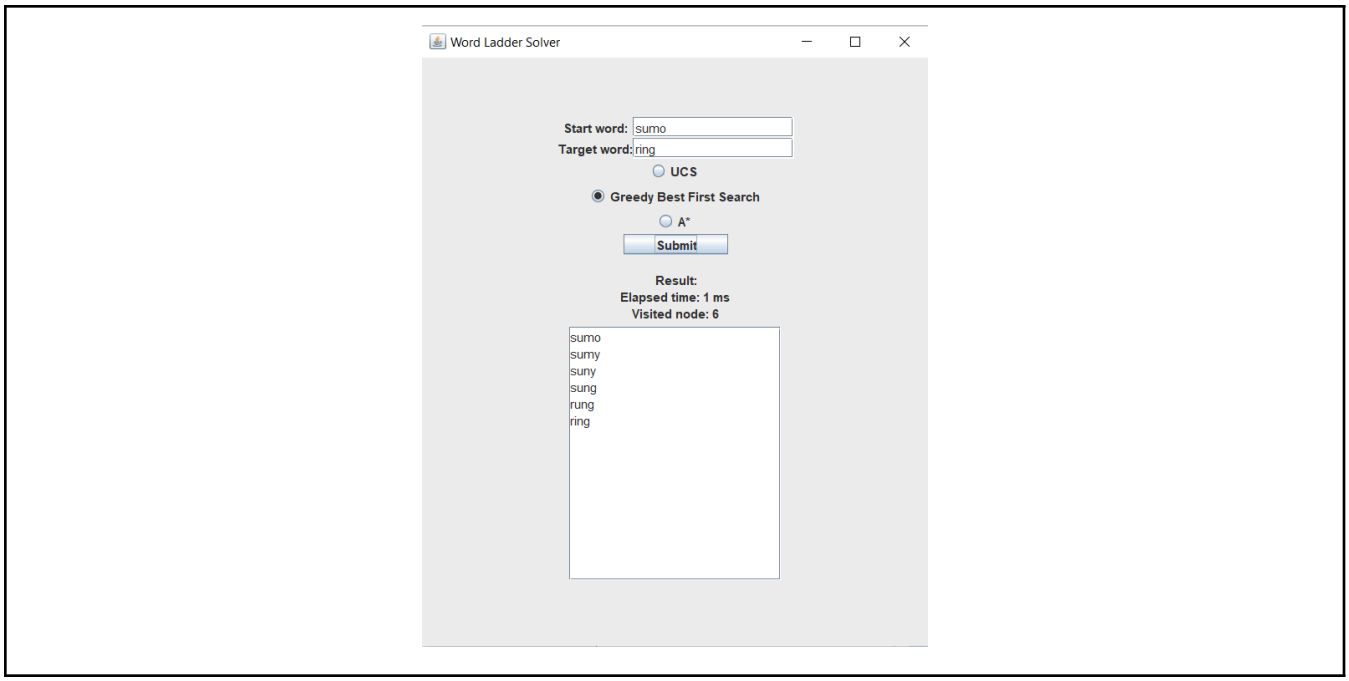
4. bar → run


Word Ladder Solver
Start word: bar
Target word: run
UCS
Greedy Best First Search
A*
Submit
Result:
Elapsed time: 22 ms
Visited node: 2794
bar
rar
ran
run
Word Ladder Solver
Start word: bar
Target word: run
UCS
Greedy Best First Search
A*
Submit
Result:
Elapsed time: 0 ms
Visited node: 5
bar
rar
ran
run

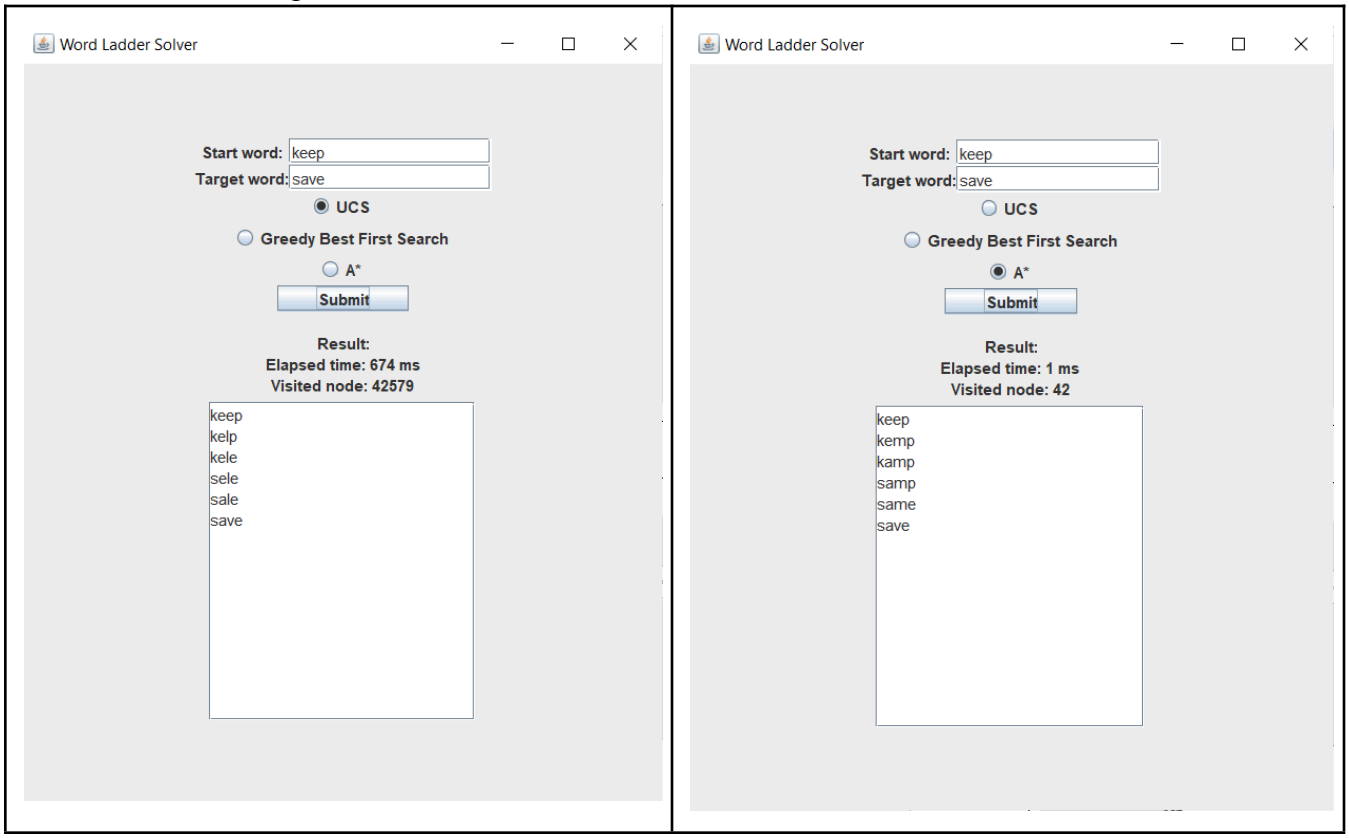

Word Ladder Solver
Start word: bar
Target word: run
UCS
Greedy Best First Search
A*
Submit
Result:
Elapsed time: 0 ms
Visited node: 3
bar
rar
ran
run

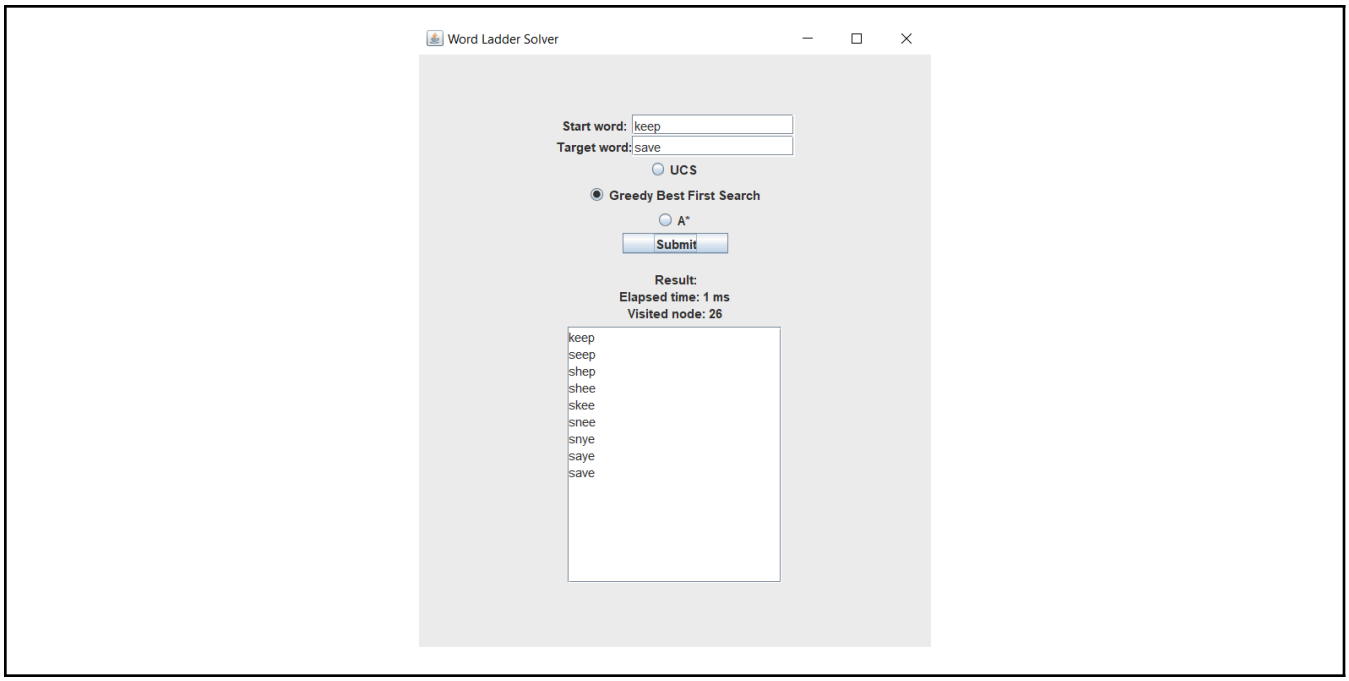
5. sumo → ring


Word Ladder Solver
Start word: sumo
Target word: ring
UCS
Greedy Best First Search
A*
Submit
Result:
Elapsed time: 207 ms
Visited node: 11686
sumo
sums
sims
sins
rins
ring
Word Ladder Solver
Start word: sumo
Target word: ring
UCS
Greedy Best First Search
A*
Submit
Result:
Elapsed time: 1 ms
Visited node: 16
sumo
sumy
suny
sung
sing
ring

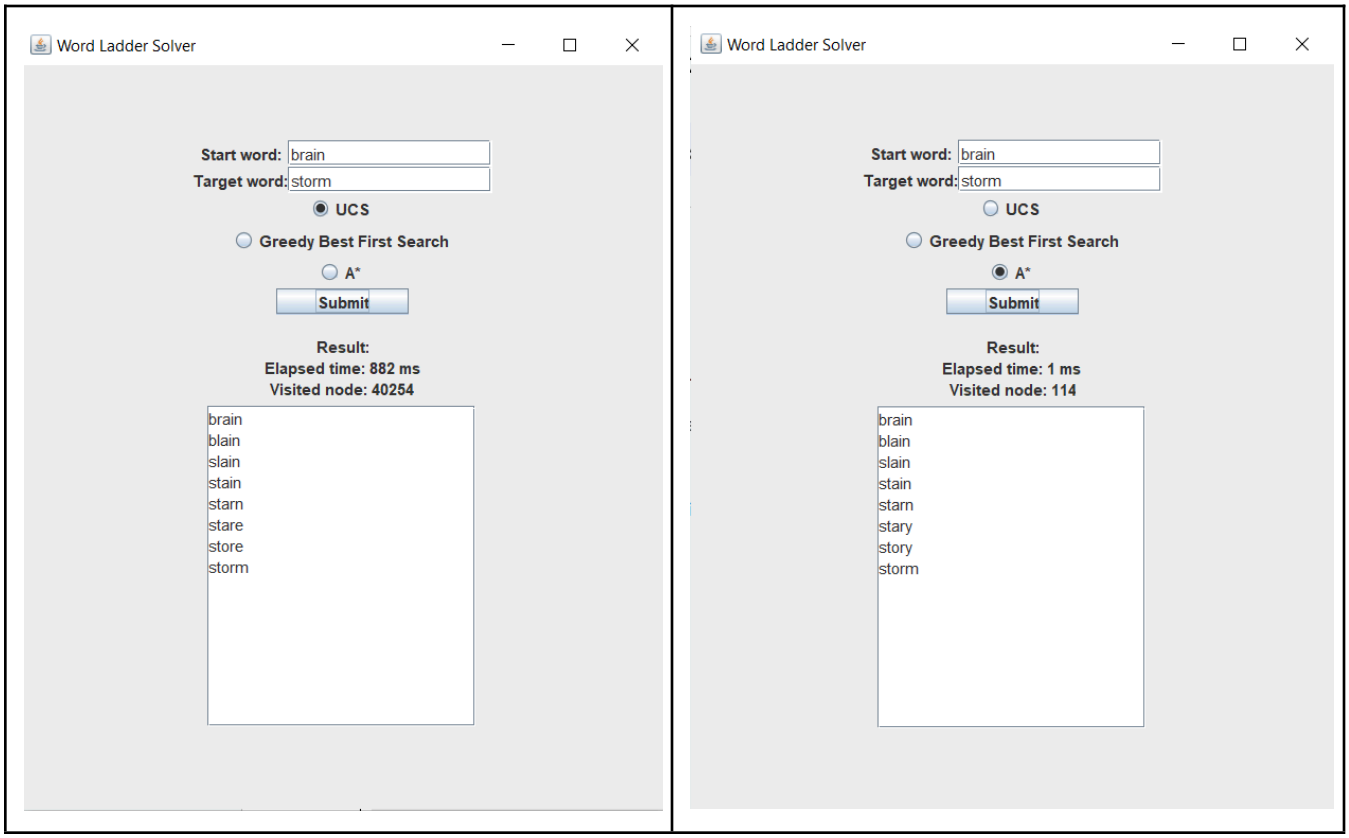

Word Ladder Solver
Start word: sumo
Target word: ring
UCS
Greedy Best First Search
A*
Submit
Result:
Elapsed time: 1 ms
Visited node: 6
sumo
sumy
suny
sung
rung
ring

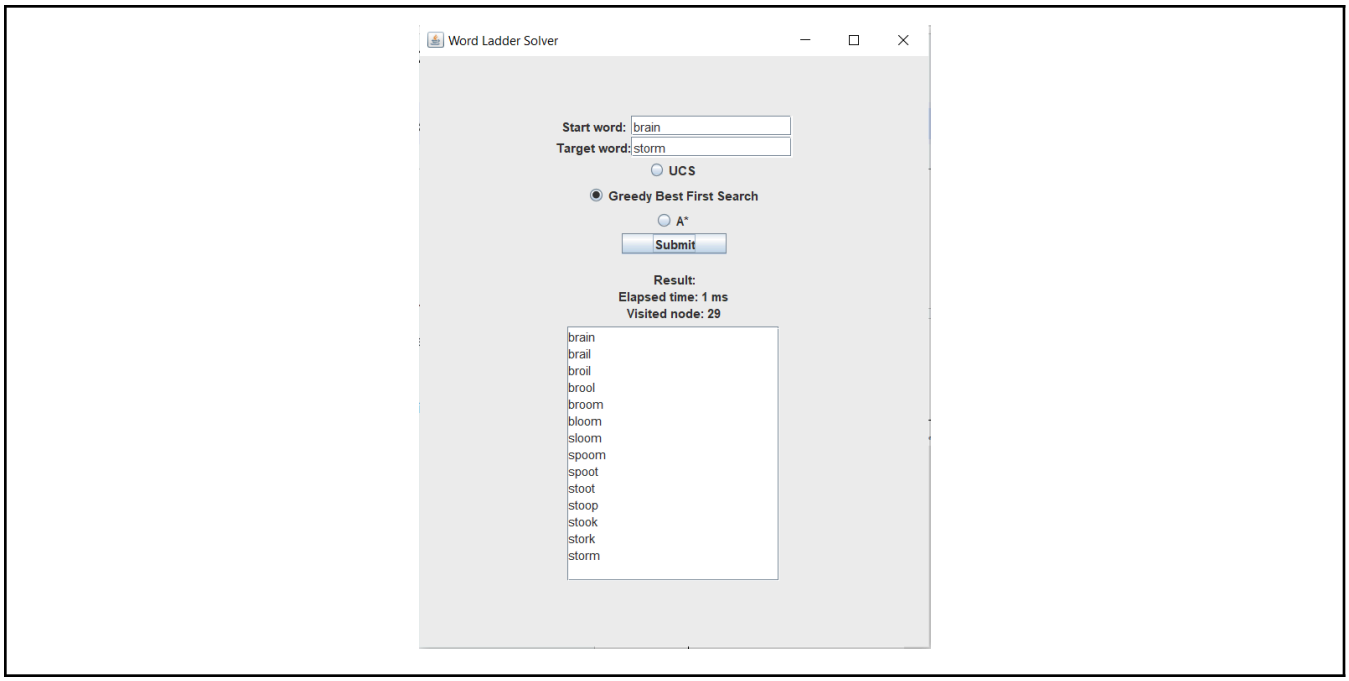
6. keep → save


Word Ladder Solver
Start word: keep
Target word: save
UCS
Greedy Best First Search
A*
Submit
Result:
Elapsed time: 674 ms
Visited node: 42579
keep
kelp
kele
sele
sale
save
Word Ladder Solver
Start word: keep
Target word: save
UCS
Greedy Best First Search
A*
Submit
Result:
Elapsed time: 1 ms
Visited node: 42
keep
kemp
kamp
samp
same
save

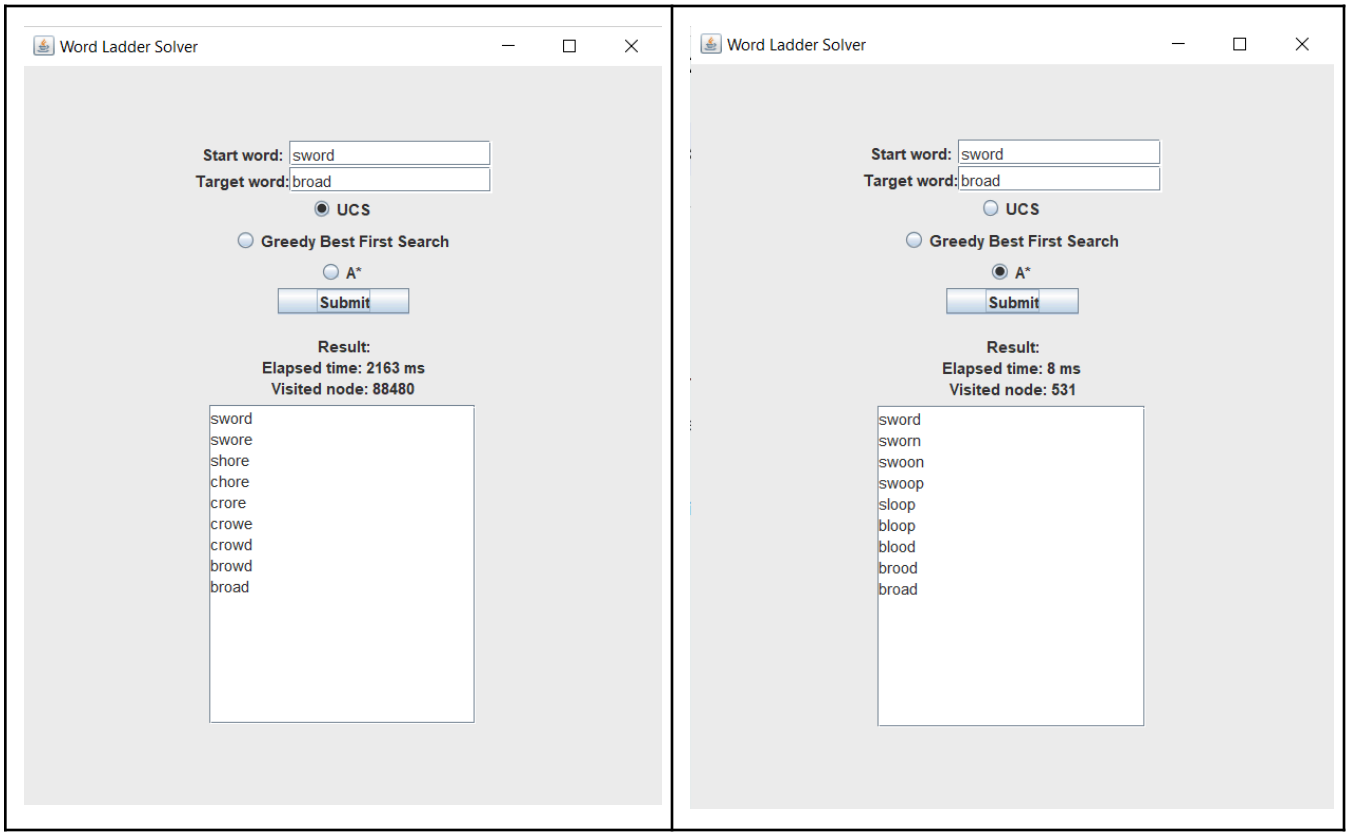


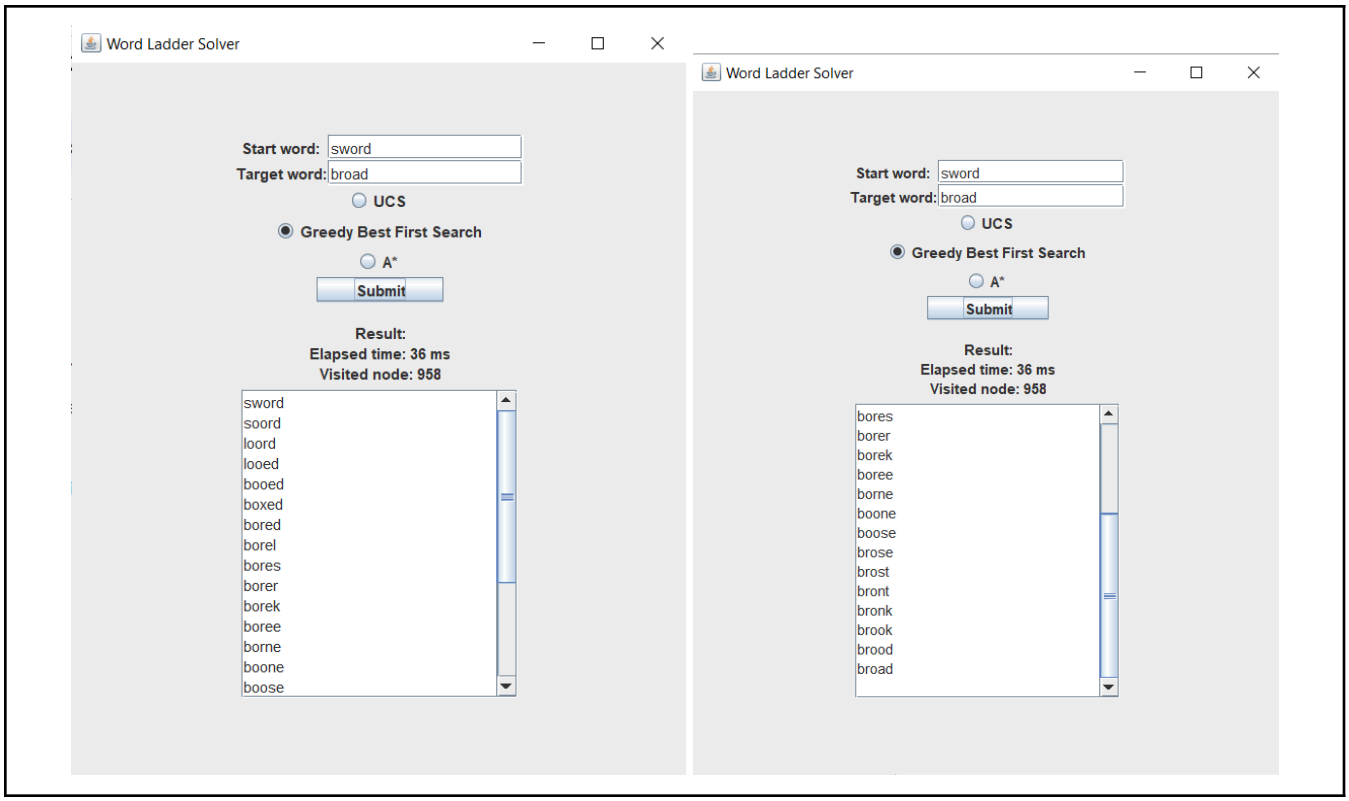
7. brain → storm



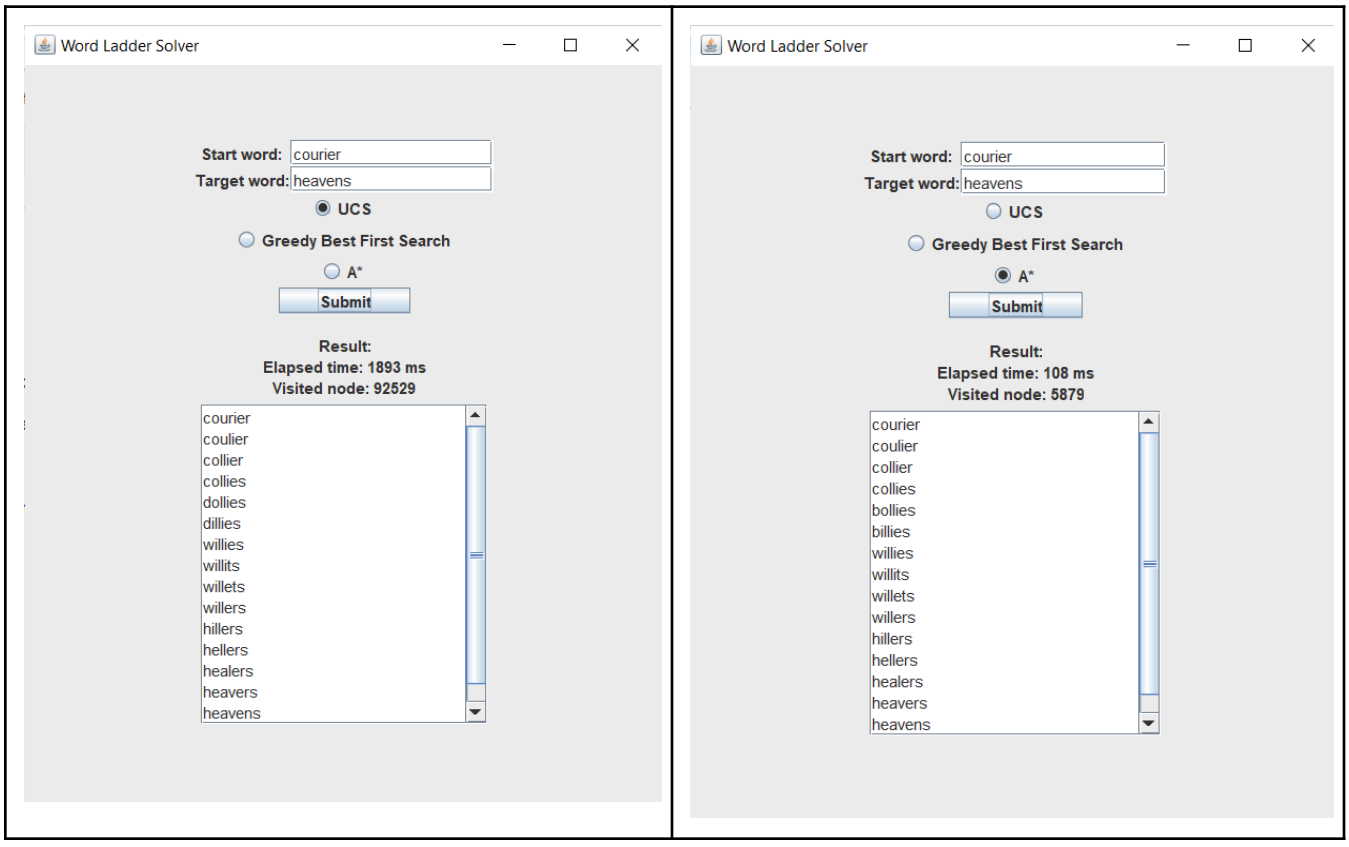


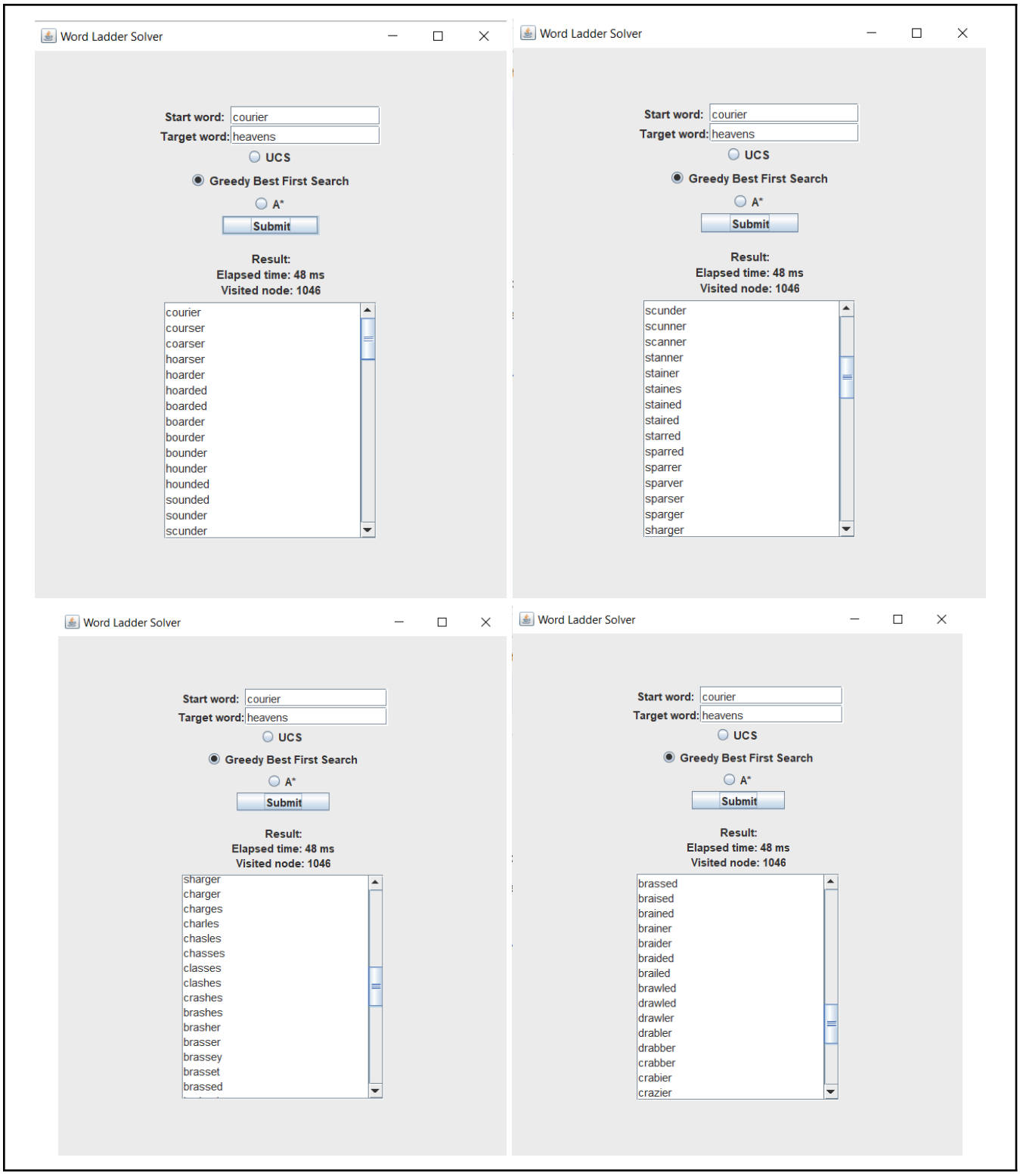
8. sword → broad

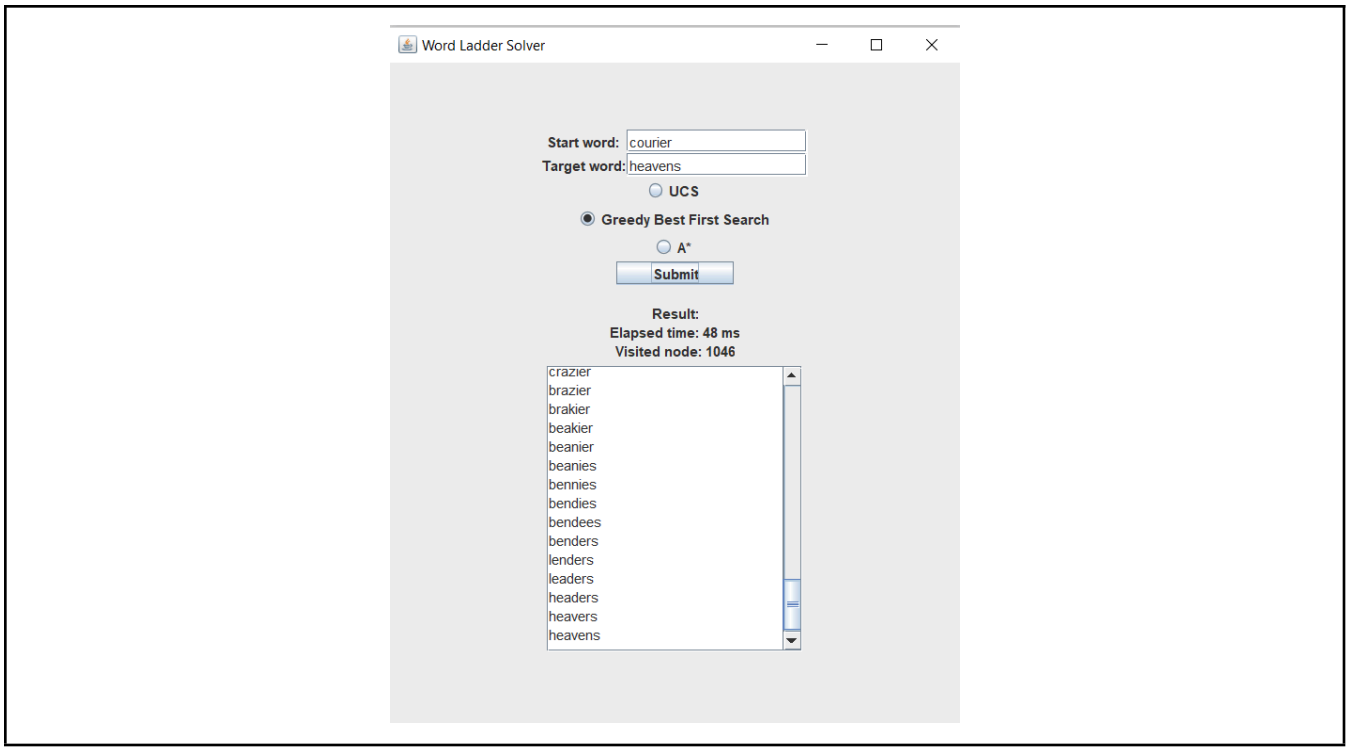




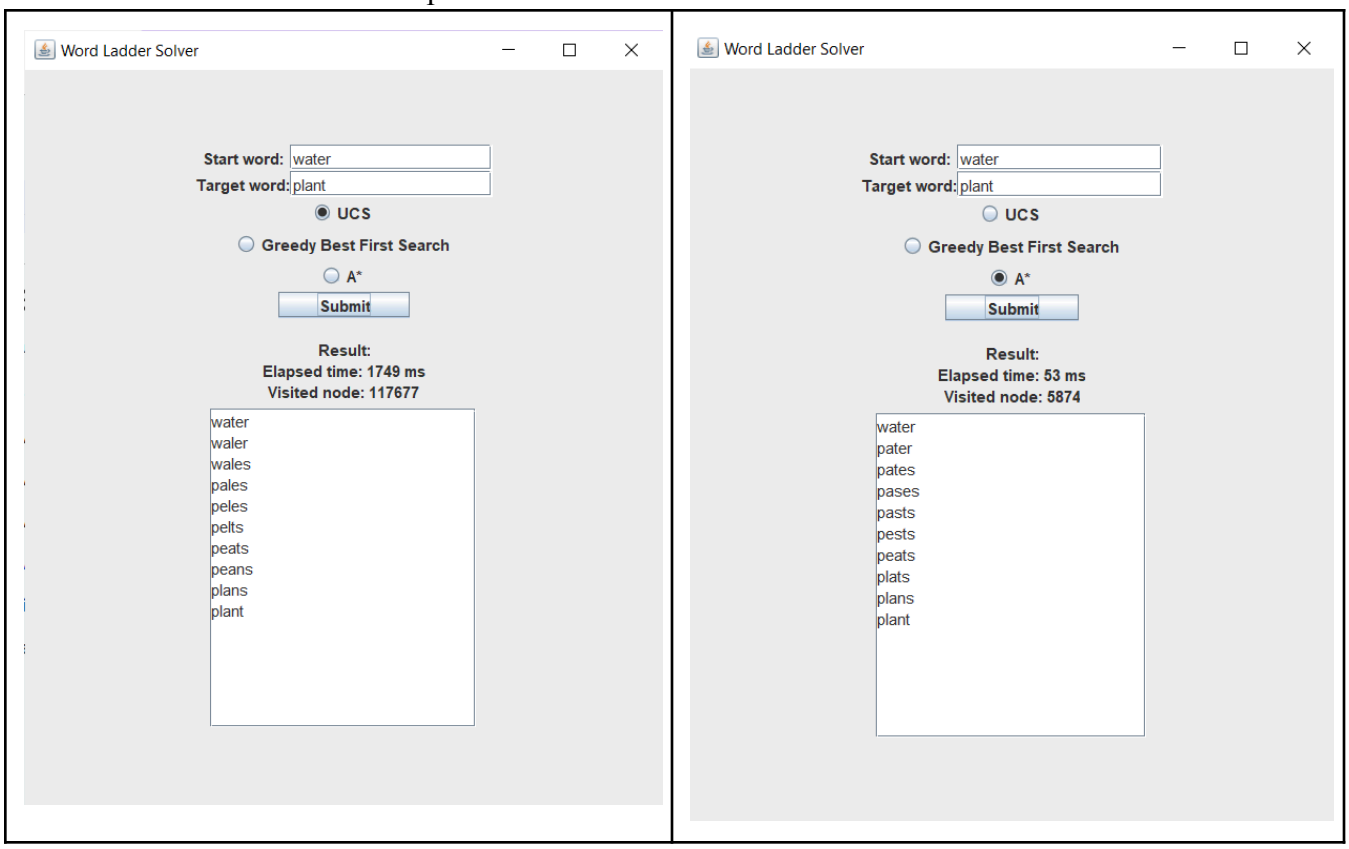
9. courier → heavens

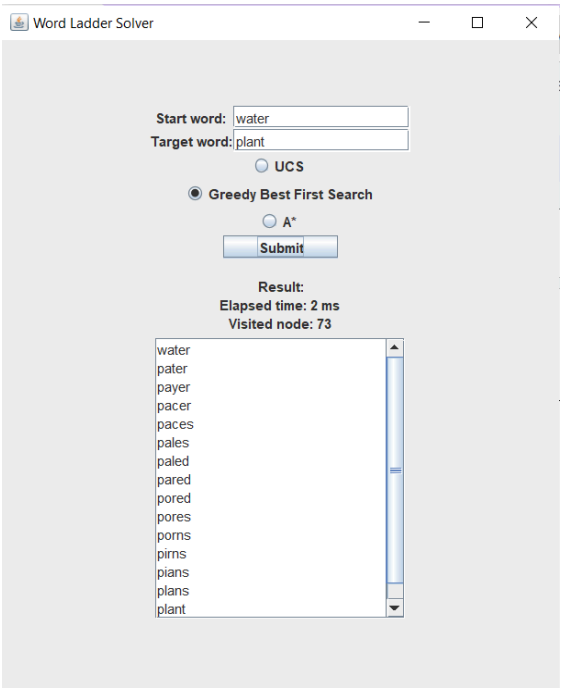



Word Ladder Solver
Start word: courier
Target word: heavens
UCS
Greedy Best First Search
A*
Submit
Result:
Elapsed time: 48 ms
Visited node: 1046
courier
courser
coarser
hoarser
hoarder
hoarded
boarded
boarder
bourder
bounder
hounder
hounded
sounded
sounder
scunder
Word Ladder Solver
Start word: courier
Target word: heavens
UCS
Greedy Best First Search
A*
Submit
Result:
Elapsed time: 48 ms
Visited node: 1046
scunder
scunner
scanner
stanner
stainer
staines
stained
staired
starred
sparred
sparrer
sparver
sparser
sparger
sharger
Word Ladder Solver
Start word: courier
Target word: heavens
UCS
Greedy Best First Search
A*
Submit
Result:
Elapsed time: 48 ms
Visited node: 1046
sharger
charger
charges
charles
chasles
chasses
classes
clashes
crashes
brashes
brasher
brasser
brassey
brasset
brassed
Word Ladder Solver
Start word: courier
Target word: heavens
UCS
Greedy Best First Search
A*
Submit
Result:
Elapsed time: 48 ms
Visited node: 1046
brassed
braised
brained
brainer
braider
braided
brailed
brawled
drawled
drawler
drabler
drabber
crabber
crabier
crazier



10. water → plant





4.5. Analisis

Algoritma	Panjang Langkah	Kata Dikunjungi	Waktu Eksekusi (ms)
1. monster → thunder			
UCS	7	1424	144
Greedy	7	9	7
A*	7	33	5
2. cell → atom			
UCS	6	216915	3022
Greedy	16	32	2
A*	6	632	16
3. purple → planet			
UCS	12	168234	5706
Greedy	19	61	3
A*	12	1220	22

4. bar → run			
UCS	3	2794	22
Greedy	3	3	0
A*	3	5	0
5. sumo → ring			
UCS	5	11686	207
Greedy	5	6	1
A*	5	16	1
6. keep → save			
UCS	5	42579	674
Greedy	8	26	1
A*	5	42	1
7. brain → storm			
UCS	7	40254	882
Greedy	13	29	1
A*	7	114	1
8. sword → broad			
UCS	8	88480	2163
Greedy	21	958	36
A*	8	531	8
9. courier → heavens			
UCS	14	92529	1893
Greedy	70	1046	48
A*	14	5879	108
10. water → plant			

UCS	9	117677	1749
Greedy	14	73	2
A*	9	5874	53

Pada tabel analisis, dapat dilihat bahwa seluruh jumlah langkah yang ditempuh oleh algoritma UCS dan A* sama. Hal tersebut menunjukkan kedua algoritma menghasilkan solusi yang optimal. Fungsi heuristik yang digunakan pun *admissible* karena algoritma A* selalu mendapat solusi yang optimal dan nilai heuristik dari setiap simpul tidak pernah melebihi biaya optimal sesungguhnya (estimasi langkah yang diperlukan dari kata n ke kata tujuan selalu lebih kecil atau sama dengan langkah optimal sebenarnya).

Langkah yang diperlukan oleh algoritma GBFS selalu lebih banyak (tidak optimal) daripada algoritma UCS dan A*. Hal tersebut dikarenakan algoritma UCS dan A* selalu menjamin untuk mendapatkan solusi yang optimal, sedangkan GBFS tidak. Namun, dalam beberapa kasus, algoritma GBFS dapat menghasilkan solusi yang optimal seperti pada pengujian 1. Selain itu, meskipun solusi yang dihasilkan tidak selalu optimal, algoritma GBFS memiliki waktu eksekusi yang sangat cepat dan jumlah kata yang dikunjungi lebih sedikit dari kedua algoritma lainnya.

Dari ketiga algoritma, dapat dilihat bahwa algoritma A* adalah algoritma terbaik di antara semuanya. Algoritma A* selalu mendapatkan solusi yang optimal, tetapi waktu yang diperlukan tidak selama algoritma UCS. Waktu yang diperlukan algoritma A* hanya berbeda sedikit dengan algoritma GBFS. Selain itu, jumlah kata yang dikunjungi algoritma A* cukup sedikit, tidak sebanyak UCS, tetapi juga tidak sesedikit GBFS. Algoritma A* kalah cepat oleh algoritma GBFS karena algoritma A* harus menelusuri simpul lebih banyak untuk memastikan solusi yang diperoleh adalah solusi yang optimal, sedangkan algoritma GBFS hanya berfokus pada satu jalur hingga menemukan kata tujuan (tidak peduli dengan jumlah langkah yang ditempuh). Jika membandingkan algoritma UCS dan A*, algoritma A* selalu lebih unggul. Jumlah kata yang dikunjungi dan waktu eksekusi algoritma A* selalu lebih sedikit daripada algoritma UCS, tetapi solusi yang dihasilkan selalu sama-sama optimal, meskipun rute yang dilaluinya berbeda.

Memori yang digunakan oleh ketiga algoritma, ditentukan dari jumlah kata yang dikunjungi dan banyaknya iterasi dalam mencari solusi. Setiap kata yang diekspansi/dikunjungi akan menghasilkan objek kata baru (Node) sehingga akan menambah penggunaan memori. Dari ketiga algoritma, algoritma GBFS akan menggunakan memori paling sedikit karena jumlah kata yang dikunjungi dan jumlah iterasi yang dilakukan sangat sedikit. Algoritma A* memiliki jumlah iterasi dan jumlah kata yang dikunjungi sedikit lebih banyak daripada GBFS sehingga penggunaan memori akan sedikit lebih banyak juga. Algoritma UCS memiliki jumlah iterasi dan jumlah kata yang dikunjungi paling banyak sehingga memori yang digunakan oleh algoritma UCS akan cukup besar dan paling besar di antara ketiganya. Meskipun begitu, penggunaan

memori ini akan sulit ditampilkan sebagai angka pasti untuk membandingkan ketiganya karena banyaknya faktor lain yang mempengaruhi penggunaan memori tersebut.

Perbedaan kamus yang digunakan juga dapat mempengaruhi hasil dari setiap algoritma. Oleh karena itu, kamus yang digunakan perlu disesuaikan dengan pengujian yang akan dilakukan karena terkadang ada kamus yang kurang lengkap sehingga hasilnya akan berbeda jauh.

BAB 5

Kesimpulan dan Saran

5.1. Kesimpulan

Dari hasil analisis yang telah dilakukan, didapat kesimpulan bahwa algoritma A* adalah algoritma terbaik di antara semuanya dalam kasus permainan Word Ladder. Algoritma A* menghasilkan solusi yang optimal seperti UCS dan juga waktu yang cepat seperti GBFS. Selain itu, algoritma A* juga memiliki penggunaan memori yang cukup sedikit, meskipun tidak sesedikit GBFS.

5.2. Saran

Dalam algoritma UCS, terdapat sebuah cara untuk mengoptimalisasi algoritma. Cara untuk mengoptimalisasinya adalah dengan mengecek apakah *target word* sudah tercapai ketika penentuan simpul hidup. Saat ini, algoritma UCS yang digunakan, melakukan pengecekan ketika simpul diekspan, tetapi jika pengecekan dilakukan saat pembentukan simpul hidup, algoritma tidak perlu menunggu simpul tersebut diekspan dan akan langsung berhenti ketika *target word* ditemukan pertama kali di simpul hidup. Namun, hal tersebut sedikit melanggar konsep UCS. Oleh karena itu, jika tujuannya untuk mendapatkan hasil secepat-cepatnya, pembaca dapat mengubah cara pengecekan seperti yang telah dijelaskan sebelumnya.

Lampiran

Link Repository: https://github.com/MRafliRasyiidin/Tucil3_13522088

Poin	Ya	Tidak
1. Program berhasil dijalankan	✓	
2. Program dapat menemukan rangkaian kata dari *start word* ke *end word* sesuai dengan 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 dengan aturan permainan dengan algoritma A*	✓	
6. Solusi yang diberikan pada algoritma A* optimal	✓	
7. **[Bonus]:** Program memiliki GUI	✓	