Trie vs. Hashtable
Distante de editare

Moscalu Cosmin-Andrei

Universitatea Politehnica din Bucuresti
Facultatea de Automatica si Calculatoare

1 Introducere

1.1 Descrierea problemei

Prelucrarea sirurilor de caractere reprezinta o problema esentiala in lumea calculatoarelor din moment ce folosim cuvinte pentru a reprezenta informatiile intr-un mod pe care il putem intelege. In epoca digitala, producem si prelucram tot mai multe informatii, motiv pentru care avem nevoie tot mai mult de structuri de date si algoritmi eficienti pentru a rezolva problemele cu care ne confruntam. In special in cadrul prelucrarii sirurilor de caractere au aparut multi algoritmi si structuri de date care au o utilizare foarte specifica[11].

Printre aceste structuri de date se numara **trie**-ul, o structura de date sub forma unui arbore de cautare, gandita pentru a stoca si pentru a extrage cuvinte ("re**trie**val"), asociind fiecarui cuvant cate o valoare, la fel ca un hashtable. Din punct de vedere abstract trie-ul seamana mult cu un multiset, multiset-ul putand fi implementat cu hashtable sau cu arbori binari echilibrati (de exemplu, in C++ multiset-ul este implementat cu un arbore rosu-negru[6]). Spre deosebire de arborele rosu-negru, in care timpul de cautare este fie constant, fie depinde de dimensiunea setului de date, in trie timpul depinde doar de dimensiunea cuvantului. Acest lucru poate deveni un avantaj doar atunci cand se lucreaza cu un numar foarte mare de cuvinte.

O alta problema specifica sirurilor de caractere este reprezentata de gasirea distantei minime dintre doua cuvinte date, aici intervine o intreaga categorie de asemenea distante, numite **distante de editare** printre care se numara **distanta Levenshtein** si **distanta Damerau-Levenshtein restrictionata**. Ambele distante sunt asemanatoare, difera doar prin ce modificari permit asupra sirului de caractere, Distanta Levenshtein permite inserarea, stergerea sau schimbarea unui caracter, pe cand distanta Damerau-Levenshtein restrictionata mai permite in plus si permutarea a doua caractere adiacente.

1.2 Exemple de aplicatii practice

Datorita faptului ca trie-ul este un arbore de cautare a carui noduri reprezinta cate un caracter, fiecare nod putand sa faca parte din mai multe siruri, trie-ul poate fi folosit pentru **completare automata**. Daca se cauta toate cuvintele care au un prefix comun atunci se parcurge trie-ul pana cand se ajunge la capatul

cuvantului si apoi se realizeaza o parcurgere in inordine pentru a gasi toate cuvintele din trie care respecta aceasta conditie. In plus, daca nodurile copil ale trie-ului sunt stocate in ordine alfabetica atunci o parcurgere in inordine afiseaza toate cuvintele in ordine lexicografica, astfel, trie-ul se poate folosi pentru **sortarea lexicografica** a unui numar mare de cuvinte foarte rapid. Tinand cont de cele de dinainte, trie-ul se mai poate folosi pentru **gasirea celui mai lung prefix**, o aplicatie foarte utila pentru rutarea pachetelor IP, in care se cauta cel mai lung prefix de retea. Impreuna cu distantele de editare, trie-ul mai poate fi folosit pentru **corectare automata**, un cuvant dat de un utilizator, care contine o greseala va avea un prefix in interiorul trie-ului, cuvantul se cauta in trie cat de mult se poate si din punctul in care literele cuvantului nu mai apar in trie se ofera sugestii in functie de distanta de editare dintre cuvintele gasite in trie, dar si in functie de valoarea pe care trie-ul o asociaza fiecarui cuvant, asta pentru a putea oferi sugestii mai relevante.[9]

Pe langa utilizarea mentionata mai sus, **corectarea automata**, distantele de editare au utilizari foarte variate precum: **potrivirea secventelor genetice** reprezentate sub forma de siruri de caractere, **evaluarea performantelor recunoasterii vocale** sau **motoare de cautare**[1].

1.3 Specificarea solutiilor

Tinand cont de cele spuse mai sus, trie-ul va fi implementat sub forma unui arbore de cautare in care fiecare nod contine un array de pointeri la nodurile care urmeaza, mai exact, fiecare nod va contine 26 de pointeri, adica numarul de litere din alfabet. Aceasta varianta, desi foloseste mai multa memorie, este mai simpla si posibil mai rapida decat alte solutii propuse care reduc utilizarea de memorie[9] datorita faptului ca nu mai este de nevoie de atat de multe comparatii atunci cand se face parcurgerea arborelui.

In cazul distantei Levenshtein se va folosi implementarea de programare dinamica, mai exact algoritmul Wagner-Fischer, iar pentru distanta Damerau-Levenshtein restrictionata se poate folosi o modificare a aceluiasi algoritm. Doar algoritmul pentru distanta Levenshtein va fi comparat cu o varianta mai eficienta din motive explicate mai tarziu.

1.4 Specificarea criteriilor de evaluare pentru solutii

Pentru testare corectitudinii trie-ului se vor folosi o serie de teste care au fost produse cu ajutorul unor programe auxiliare. In continuare, se va verifica timpul de inserare, stergere si cautare in trie in comparatie cu un multiset pe un set de cuvinte alese aleator dintr-un set foarte mare[5]. In cazul testelor lungi, inserarile si cautarile in trie ar trebui sa devina mai rapide pe masura ce trie-ul devine mai completat, in special pe seturi de date care contin cuvinte asemanatoare si care acopera mai multe litere din alfabet.

Algoritmii de calculare a distantelor se vor verifica folosind proprietatile elementare ale distantelor[3]. Pentru teste mai mari corectitudinea se va verifica

fata de o alta implementare care este corecta[4]. Timpul de executie al celor doi algoritmi va fi comparat cu varianta mai eficienta.

2 Prezentarea solutiilor

2.1 Descrierea modului in care functioneaza algoritmii

Trie Asa cum a fost mentionat mai devreme, trie-ul este reprezentat sub forma unui arbore de cautare pentru care fiecare nod are un array de pointeri la nodurile care urmeaza. Astfel, fiecare nod reprezinta un caracter si i se asociaza o valoare pentru a indica daca ramura arborelui pana in acel punct reprezinta un cuvant, dar si pentru a putea fi folosita pentru alte scopuri. In continuare, se vor detalia principalele operatii care se efectueaza asupra unui trie.

Inserare Inserarea va incepe de la copiii radacinei arborelui, radacina este tot un nod doar ca nu i se asociaza nici-o litera. Astfel, operatia de inserare se poate sumariza in felul urmator:

```
1: function INSERT(root, word, value)
2:     node ← root
3:     length ← length of word
4:     for i ← 0, length do
5:         if node→children[word[i]] doesn't exist then
6:             node→children[word[i]] ← new_node()
7:         end if
8:         node ← node→children[word[i]]
9:     end for
10:    node→value ← value
11: end function
```

Observatii: Prin *word[i]* se intelege de fapt *word[i] - 'a'*, pentru a genera un indice valid pentru array, iar prin "→" se intelege membrul unei clase sau a unei structuri.

Practic, algoritmul de mai sus parcurge cuvantul in tandem cu arborele si insereaza cate un nod pentru fiecare litera a cuvantului daca este nevoie, atfel doar parcurge arborele. Odata ce s-a parcurs tot cuvantul, nodul final va contine valoarea care se asociaza cuvantului, semnificand de asemenea ca acea ramura a arborelui reprezinta un cuvant.

Cautare Operatia de cautare este aproape identica cu cea de inserare, doar ca, atunci cand nu se gaseste nodul asociat unei litere din cuvant, inseamna ca acel cuvant nu poate exista in arbore si cautarea se poate incheia mai devreme.

```
1: function SEARCH(root, word)
2:     node ← root
3:     length ← length of word
4:     for i ← 0, length do
5:         if node→children[word[i]] doesn't exist then
6:             return false
7:         end if
8:         node ← node→children[word[i]]
9:     end for
10:    if node→value > 0 then
11:        return node→value
12:    else
13:        return false
14:    end if
15: end function
```

Chiar daca toate literele cuvantului exista in arbore, asta nu inseamna neaparat ca acele litere sunt considerate a fi un cuvant. De exemplu, daca se insereaza in arbore cuvantul *comuna* si apoi se cauta cuvantul *comun* atunci acest cuvant nu va fi gasit, chiar daca literele sale sunt prezente in arbore. Conditia de la final e pusa pentru a verifica acest lucru.

Stergere Dintre aceste trei operatii, se poate considera ca cea de stergere este cea mai complicata, necesitand doua parcurgeri ale arborelui, una de de sus in jos pentru a gasi cuvantul si una de revenire, pentru a elibera memoria ocupata de cuvantul sters, daca se poate. Totusi, are o parte comuna cu operatia de cautare:

```
1: function DELETE(root, word)
2:     node ← root
3:     length ← length of word
4:     for i ← 0, length do
5:         if node→children[word[i]] doesn't exist then
6:             return false
7:         end if
8:         node ← node→children[word[i]]
9:     end for
10:    if node→value ≤ 0 then
11:        return false
12:    end if
13:    for i ← length, 0 do
14:        parent ← node→parent
15:        if is_leaf(node) then
16:            parent→children[word[i]] ← null
17:            free_node(node)
```

```
        else
            return true
        end if
        node ← parent
    end for
    return true
end function
```

Acest algoritm presupune ca fiecare nod are si un pointer la parinte. In cazul in care nodurile nu au pointer la parinte, atunci acest algoritm se poate implementa in mod recursiv: la revenirea din apelurile recursive se verifica daca nodul este frunza si daca este, atunci se elibereaza memoria ocupata de acesta.

Distante de editare Cele doua distante de editare, Levenshtein si Damerau-Levenshtein se pot implementa cu algoritmul Wagner-Fischer de programare dinamica, care presupune calcularea valorilor dintr-o matrice de distante. Pentru a intelege abordarea de programare dinamica, se pleaca de la urmatoare definitie recursiva a distantei Levenshtein dintre doua siruri.

$$lev_{a,b}(i,j) = \begin{cases} max(i,j) & \text{, daca } i = 0 \text{ sau } j = 0 \\ min \begin{cases} lev_{a,b}(i-1,j) + 1 \\ lev_{a,b}(i,j-1) + 1 \\ lev_{a,b}(i-1,j-1) + 1_{a_i \neq b_j} \end{cases} & \text{, } altfel \end{cases}$$

Unde a si b reprezinta cele doua siruri de caractere, iar i si j reprezinta indici indexati de la 1 pentru cele doua siruri, iar prin $1_{a_i \neq b_j}$ se intelege 1 daca $a_i = b_j$, 0 altfel.

Daca din sirul a se incearca a se obtine sirul b, atunci distanta dintre cele doua este egala cu minimul dintre o distanta anterioara plus un cost de inserare, stergere sau schimbare a unui caracter. Prin distanta anterioara se intelege distanta dintre un prefix al lui a si un prefix al lui b, dupa caz.

Totusi, o implementare directa acestui algoritm ar avea o complexitate exponentiala. Din acest motiv, distantele se vor calcula cu ajutorul unei matrici care reprezinta toate distantele posibile dintre toate prefixele sirurilor a si b. Algoritmul de programare dinamica se poate defini in felul urmator:

```
function LEVENSHTEIN(a, b)
    len_a ← length of a
    len_b ← length of b
    matrix[len_a + 1][len_b + 1]                ▷ Se declara matricea
    for i ← 0, len_a + 1 do                     ▷ Se initializeaza prima coloana
        matrix[i][0] ← i
    end for
    for i ← 0, len_b + 1 do                     ▷ Se initializeaza prima linie
        matrix[0][i] ← i
    end for
    for i ← 0, len_a + 1 do
```

12: **for** $j \leftarrow 0, \text{len\_b} + 1$ **do**
13: $matrix[i][j] \leftarrow min(matrix[i-1][j] + 1,$
$matrix[i][j\text{ - }1] + 1,$
$matrix[i-1][j-1] + (a[i-1] \neq b[j-1]))$
14: **end for**
15: **end for**
16: **return** $matrix[len\_a][len\_b]$
17: **end function**

Functia min folosita in algoritm este aceeasi cu cea din definitia recursiva a distantei Levenshtein si semnifica minimul dintre o inserare, o stergere sau o transpozitie. In cazul in care cele doua caractere din sirul a si b sunt egale, atunci costul unei schimbari este 0, practic nu este nevoie de o schimbare si se poate pastra distanta anterioara.

Distanta Damerau-Levenshtein restrictionata mai aduga in plus si operatia de transpozitie a doua caractere alaturate, astfel formula recursiva pentru distanta Damerau-Levenshtein devine:

$$dmlev_{a,b}(i,j) = \begin{cases} max(i,j) & \text{, daca } i = 0 \text{ sau } j = 0 \\ min \begin{cases} dmlev_{a,b}(i-1,j) + 1 \\ dmlev_{a,b}(i,j-1) + 1 \\ dmlev_{a,b}(i-1,j-1) + 1_{a_i \neq b_j} \\ dmlev_{a,b}(i-2,j-2) + 1 \end{cases} & \text{, daca } a_{i-1} = b_j, \\ & \quad a_i = b_{j-1} \text{ si} \\ & \quad i,j > 1 \\ min \begin{cases} dmlev_{a,b}(i-1,j) + 1 \\ dmlev_{a,b}(i,j-1) + 1 \\ dmlev_{a,b}(i-1,j-1) + 1_{a_i \neq b_j} \end{cases} & \text{, altfel} \end{cases}$$

In aceasta formula se mai adauga in calcularea minimului si operatia de interschimbare, mai exact se mai ia si $dmlev_{a,b}(i-2,j-2)+1$ in calcul daca prin interschimbare se produce un rezultat corect, corectitudinea transpozitiei fiind indicata de conditia $a_{i-1} = b_j$, $a_i = b_{j-1}$ si $i,j > 1$.

Calcularea distantei Damerau-Levenshtein restrictionate presupune modificarea algoritmului de programare dinamica pentru distanta Levenshtein, mai exact linia 13 din acel algoritm se poate modifica in felul urmator:

13: $matrix[i][j] \leftarrow min(matrix[i-1][j] + 1,$
$matrix[i][j\text{ - }1] + 1,$
$matrix[i-1][j-1] + (a[i-1] \neq b[j-1]))$
14: **if** $i, j \geq 2$ **and** $a_{i-1} = b_{j-2}$ **and** $a_{i-2} = b_{j-1}$ **then**
15: $matrix[i][j] \leftarrow min(matrix[i][j],$
$matrix[i-2][j-2] + 1)$
16: **end if**

Aceasta modificare presupune doar inca o verificare de minim, operatie care se executa in timp constant.

Pe scurt, calcularea distantei Damerau-Levenshtein restrictionate este aproape identica cu cea a distantei Levenshtein.

2.2 Analiza complexitatii solutiilor

Trie

Inserare Complexitatea operatiei de inserare este data de bucla principala in care se executa operatii cu timp constant, astfel complexitatea inserarii este:

$$O(lungime\_cuvant)$$

Chiar daca complexitatea este data in cea mai mare parte de bucla, operatiile care se executa in bucla pot avea un factor constant destul de mare, alocarea unui nou nod daca este nevoie si parcurgerea nodurilor, indiferent daca se aloca un nod nou sau nu, sunt operatii care pot dura destul de mult in practica.

Cautare La fel ca la inserare, complexitatea operatiei de cautare este data tot de bucla principala in care se executa operatii cu timp constant. Astfel complexitatea cautarii este:

$$O(lungime\_cuvant)$$

Spre deosebire de inserare, cautarea nu trebuie sa aloce noduri noi, deci nu are un factor constant foarte mare in interiorul buclei si nici nu trebuie sa parcurga tot cuvantul pana la capat. Daca nu se gaseste un nod atunci algoritmul se opreste. Asadar cautarea poate fi mai rapida decat inserarea, chiar daca are aceeasi complexitate.

Stergere Stergerea unui cuvant din trie presupune parcurgerea a doua bucle care depind de lungimea cuvantului, dar in care executa operatii cu timp constant. Astel pentru ambele bucle complexitatea va fi $O(lungime\_cuvant)$. Cele doua bucle se executa una dupa alta, deci complexitatea totala va fi:

$$O(lungime\_cuvant)$$

Pentru operatia de cautare exista si cazuri foarte favorabile si cazuri foarte defavorabile, in care complexitatea de mai sus nu spune toata povestea. Pentru prima bucla, exista posibilitatea ca algoritmul sa termine executia inainte sa ajunga la capatul cuvantului, asta daca cuvantul nu exista in trie. Totusi, in cazul cel mai defavorabil in care, de exemplu, se sterge singurul cuvant din trie, atunci cea de-a doua bucla va trebuie sa apeleze functia is_leaf() pentru fiecare nod, mai exact pentru fiecare litera a cuvantului. Astfel, in aceasta situatie complexitatea va fi in realitate mai aproape de:

$$O(lungime\_cuvant + lungime\_alfabet * lungime\_cuvant) = \\ O((1 + lungime\_alfabet) * lungime\_cuvant)$$

Unde *lungime_cuvant* vine din prima bucla si *lungime_alfabet***lungime_cuvant* vine din functia is_leaf() care este rulata pentru fiecare nod si care, in cel mai rau caz, trebuie sa parcurga toti copiii unui nod pentru a verifica daca nodul este frunza.

Observatie: Algoritmul de stergere care va fi folosit in teste nu este identic cu cel mentionat anterior, fiind varianta recursiva a acestuia, dar avand aceeasi complexitate.

Despre Multiset In cadrul testelor, am ales sa compar trie-ul cu un multiset implementat cu hashtable neredimensionabil cu inlantuire directa. In teorie, toate operatiile asupra unui hashtable se executa in timp $O(1)$, dar daca se ia in calcul si operatia de hash atunci complexitatea operatiilor pe un mutliset ar fi tot $O(lungime\_cuvant)$. Totusi, operatia de hash se realizeaza mult mai repede in comparatie cu operatiile care se executa asupra trie-ului, motiv pentru care e de asteptat ca trie-ul sa fie mai lent.

Alta Observatie O alta operatie care se poate defini pe cele doua structuri, dar care nu este o operatie de baza, este afisarea in ordine ale cuvintelor din structura. In cazul trie-ului aceasta operatie se realizeaza printr-o parcurge in inordine, dar in cazul hashtable-ului trebuie concatenate toate listele inlantuite ale acestuia si apoi sortate. Pentru sortarea listelor s-a folosit un algoritm de merge sort recursiv[2]. In cazul trie-ului aceasta operatie se poate extinde usor ca sa intoarca toate cuvintele din trie care au prefix comun, pe cand intr-un hashtable acest lucru nu este posibil. Totusi, aceasta operatie de gasire a cuvintelor cu prefix comun nu a fost testata.

Distante de editare Complexitatea pentru algoritmul Wagner-Fischer este data de cele doua bucle imbricate care depind de lungimile celor doua cuvinte la care se mai adauga si buclele de initializarile. Astfel pentru bucla de pe linia 12 din algoritmul Wagner-Fischer, complexitatea va fi $O(n)$, aceasta bucla este rulata de m ori, si daca se mai adauga si initializarile rezulta urmatoare complexitate:

$$O(m * n + m + n) = O(mn)$$

Unde m este lungimea sirului a si n este lungimea sirului b.

In cazul distantei Damerau-Levenshtein restrictionate, asa cum s-a aratat mai sus, in algoritmul Wagner-Fischer se mai adauga o conditie care se executa in timp constant, asadar complexitatea calcularii distantei Damerau-Levenshtein restrictionate va fi tot $O(mn)$. Aceasta modificare nu se aplica doar pentru algoritmul Wagner-Fischer dar si pentru alti algoritmi mai eficienti pentru calcularea distantei, precum [13][7][10]. In lucrarile citate fie se implica, fi se mentioneaza ca, pentru a face algoritmii imbunatatati sa calculeze distanta Damerau-Levenshtein restrictionata, este nevoie doar de modificari mici care nu maresc complexitatea algoritmului.

Observatie Tinand cont de cele exprimate mai sus, in compararea algoritmilor, se va lua in calcul doar distanta Levenshtein, pentru ca se poate presupune ca algoritmii modificati pentru distanta Damerau-Levenshtein restrictionata vor rula marginal mai lent.

2.3 Avantaje si dezavantaje pentru solutiile luate in considerare

Trie In general, un avantaj al trie-ului il reprezinta aplicabilitatea sa in situatii practice precum corectarea automata, si in algoritmi care ar fi mai greu de

implementat cu alte structuri, de exemplu, algoritmul de compresie Lempel-Ziv care se implementeaza foarte usor cu ajutorul unui trie. Spre deosebire de alte structuri care ar putea fi folosite in situatii similare, complexitatea operatiilor pe trie depinde doar de lungimea cuvantului inserat si, fiind un arbore, acesta poate sa creasca atat cat permite memoria sistemului fara sa fie nevoie de operatii costisitoare de redimensionare de care are nevoie hashtable-ul, de exemplu. Tot in comparatie cu hashtable-ul, trie-ul nu are deloc nevoie de o functie de hash, ceea ce elimina toate dezavantajele care pot veni cu alegerea unei functii de hash nepotrivite. In ceea ce priveste implementarea facuta de mine, aceasta este mai rapida decat altele care nu ocupa la fel de multa memorie, deoarece nodurile copil pot fi accesate direct printr-un indice intr-un array, operatie care se realizeaza foarte rapid.

Totusi, trecand la dezavantaje, implementarea facuta de mine ocupa foarte multa memorie si are factori constanti mari la operatii. Tinand cont de problema memoriei ocupate, exista foarte multe modificari care se pot face asupra unui trie pentru a reduce utilizarea de memorie. Printre aceste modificari se numara: inlocuirea array-ului de noduri copil cu o lista inlantuita, reducerea numarului de noduri frunza prin folosirea unui singur nod pe post de frunza sau, o modificare mai semnificativa, arborele Patricia, un arbore care memoreaza mai multe litere pentru fiecare nod.

Distante de editare Algoritmul Wagner-Fischer este un algoritm foarte simplu si usor de inteles fiind, astfel, un punct bun de plecare in introducerea algoritmilor de programare dinamica. Acest algoritm este si foarte usor de modificat pentru a calcula distanta Damerau-Levenshtein restrictionata. Tot in aceeasi idee, prin utilizarea unei modificari propuse de Ukkonen, acest algoritm poate fi usor modificat pentru a fi mai rapid, modificarea consta in reducerea numarului de elemente din matrice parcurse daca se da din prealabil o distanta maxima dintre cele doua cuvinte.

Chiar si cu modificari, algoritmul Wagner-Fischer este in continuare mai lent decat alti algoritmi aparuti ulteriori care realizeaza acelasi calcul, de exemplu algoritmul paralel pe biti al lui Myers[12]. In ce priveste implementarea facuta de mine, aceasta poate fi modificata pentru a reduce utilizarea de memorie, prin stocarea a doar doua linii sau a doua coloane complexitate spatiala ajunge la $O(min(m, n))$ fara a se schimba complexitatea temporala. Si pentru Wagner-Fischer, dar si pentru alti algoritmi mai eficienti, cum ar fi cel al lui Ukkonen inca se fac destul de multe comparatii pentru fiecare element, astfel rezulta un factor constant mare in interiorul buclelor.

3 Evaluare

In cele ce urmeaza, trie-ul a fost comparat cu un multiset, ambele scrise de mine in limbajul C si implementate in modalitatile descrise anterior. In cazul distantelor de editare, algoritmul Wagner-Fischer a fost scris in C si rulat din

Python, pentru a se putea obtine rezultate comparabile cu implementarea de referinta care este scrisa tot in C si rulata din Python.

Implementarea algoritmului Wagner-Fischer mai contine si imbunatatirea propusa de Ukkonen, in cadrul unor teste se va oferi si o limita pentru distanta si se va testa in comparatie cu implementarea de referinta. De asemenea, implementarea de referinta foloseste o varianta imbunatatita a algoritmului lui Myers[4][10] si are complexitate $O(\lfloor \frac{mn}{w} \rfloor)$, unde m si n au aceleasi semnificatii ca inainte, w este lungimea unui cuvant, asa cum este definit de arhitectura sistemului de calcul, iar $\lfloor x \rfloor$ reprezinta partea intreaga.

In cazul trie-ului si al multiset-ului se vor evalua timpii de rulare pe fiecare operatie: inserare, cautare, stergere si afisare, iar in cazul distantelor se vor verifica timpii pentru intregul test.

3.1 Descrierea modalitatii de construire a setului de teste

Trie Pentru verificarea corectitudinii implementarilor facute de mine, s-au generat cu ajutorul unor script-uri 8 teste care pot fi verificate si manual. Mai departe, pentru verificarea eficientei, s-au generat o serie de alte teste care variaza in dimensiuni si in operatii. Cuvintele au fost extrase dintr-o lista de cuvinte din alfabetul englez[5] si iesirile pentru aceste teste au fost generate cu ajutorul unei implementari de referinta.

Descrierea testelor:

- 9 - 18 • Perechi de cate doua teste in care se insereaza un anumit numar de cuvinte, se cauta toate acele cuvinte si apoi se sterg in alta ordine. Numarul de cuvinte este: 10000, 25000, 50000, 100000, 300000;
- 19, 20 • Doua teste in care se insereaza 100000 de cuvinte apoi se cauta si se sterg toate aceste cuvinte in ordine;
- 21, 22 • Aceste teste sunt asemanatoare cu cele doua de dinainte, doar ca se cauta si se incearca a se sterge cuvinte care nu sunt in trie;
- 23 - 26 • Teste in care se face un numar mare de inserari, urmate apoi de o combinatie aleatoare de inserari, cautari, stergeri sau afisari. Pentru testele 23 si 24 se insereaza 100000 de cuvinte initial, iar pentru 25 si 26 se insereaza 200000 de cuvinte, respectiv 300000. Pentru testul 23 se efectueaza ulterior inserarilor un numar de 100000 de operatii, iar pentru 24, 25 si 26 300000 de operatii. In plus, pentru testele 25 si 26 nu se vor face afisari.

Distante de editare La fel ca la trie, s-au generat 8 teste folosind diverse script-uri cu rolul de a verifica proprietatile de baza ale distantelor pe implementarile facute de mine. Restul testelor sunt facute pentru verificarea eficientei algoritmilor. Cuvintele au fost generate aleator si au diverse lungimi.

Descrierea testelor:

- 9 - 12 • Teste care contin cuvinte cu de la 10 pana la 30 de litere si care au ca dimensiune totala: 50000, 100000, 250000, 500000;

13 - 17 • La fel ca mai inainte, dar distanta dintre cuvinte este de maxim 1, 3, 5, 7, 9, respectiv;

18 - 22 • Identice cu testele 13 - 17, doar ca nu se mai ofera o limita pentru distanta pentru algoritmul Wagner-Fischer;

23, 24, 25 • In aceste teste, cuvintele au de la 5 pana la 10 litere si au o distanta maxima de 1, 3, 5, respectiv. Dimensiunea testelor este de 100000;

26, 27, 28 • Aceleasi teste ca 23, 24, 25, doar ca nu se ofera o limita pentru algoritmi pentru algoritmul cu euristica lui Ukkonen. Dimensiunea testelor este de 100000.

3.2 Specificatiile sistemului de calcul

Toate testele au fost rulate pe un laptop Asus ROG GL753VD cu urmatoarele specificatii:

Procesor	Intel Core i7 - 7700HQ @ 3.80GHz
Memorie	32GB @ 2667MHz
Placa Video	Nvidia GeForce GTX 1050 4GB GDDR5
Stocare	SSD Adata XPG SX8200 Pro 512GB
Sistem de operare	Void Linux kernel 5.15.6_1 64bit

3.3 Ilustrarea rezultatelor evaluarii

Trie

	Trie				Multiset			
Nr. test	Inserare	Cautare	Stergere	Afisare	Inserare	Cautare	Stergere	Afisare
9	12.78	3.75	9.35	6.67	3.5	2.35	2.58	8.77
10	12.82	3.74	8.64	6.58	3.57	2.38	2.66	8.92
11	29.03	11.2	23.4	15.32	9.78	7.97	9.64	23.21
12	29.01	10.78	28.03	15.32	10.18	8.13	9.69	21.87
13	54.37	22.02	45.85	27.48	19.02	17.57	20.32	46.57
14	52.53	20.52	43.89	26.22	18.14	15.67	19.25	44.45
15	97.12	45.15	86.62	48.38	37.55	33.98	41.4	102.23
16	98.55	45.58	96.11	48.72	39.02	34.74	42.66	100.92
17	252.53	143.26	253.14	117.79	117.81	110.73	129.7	347.97
18	246.7	137.08	242.53	113.41	114.37	107.69	123.74	347.57
19	76.67	15.62	61.37	39.58	38.34	23.84	29.51	95.56
20	75.81	15.61	75.79	39.15	37.89	23.55	29.02	89.47
21	91.02	28.56	41.57	46.12	35.24	20.6	20.4	96.62
22	89.52	28.1	40.63	45	34.13	19.97	19.58	89.86
23	120.66	14.92	21.95	4874.29	55.78	10.42	11.42	15286.1
24	169.54	44.56	66.03	16293.58	82.87	31.86	35.69	54298.82
25	204.21	38.17	60.17	0	113.48	28.66	33.19	0
26	264.96	40.94	65.08	0	155.77	30.87	36.39	0

Toti timpii din acest tabel sunt exprimati in milisecunde.

Distante de editare

Nr. test	Limita?	Wagner-Fischer	Referinta
9	N\A	306.06	141.14
10	N\A	636.42	294.67
11	N\A	1544.58	714.5
12	N\A	3116.1	1443.87
13	1	357.83	248
14	3	431.53	246.71
15	5	521.01	258.96
16	7	569.24	253.75
17	9	624.01	259.25
18	N\A	766.62	252.19
19	N\A	765.17	252.32
20	N\A	763.8	253.87
21	N\A	768.97	261.39
22	N\A	778.72	271.31
23	1	224.46	112.51
24	3	266.95	127.59
25	5	278.33	130.8
26	N\A	270.78	112.22
27	N\A	284.7	122.51
28	N\A	280.06	127.89

Toate valorile din acest tabel sunt exprimate in milisecunde.

3.4 Prezentarea valorilor obtinute pe teste

Trie Primele 10 teste din tabel confirma ce s-a observat in cadrul analizei complexitatii: pentru ambele structuri, timpii nu depind de numarul de cuvinte care sunt prezente deja in structura ci doar de dimensiunea setului de intrare, de aceea, daca se dubleaza numarul de cuvinte inserate, se dubleaza si timpul de rulare. Totusi, in trecerea de la testele 15, 16 la testele 17, 18 in care numarul de cuvinte se tripleaza, se observa ca trie-ul nu respecta aceeasi regula, timpii de inserare cresc de aproximativ 2.6 ori, pe cand in hashtable timpul se tripleaza efectiv. Acest lucru se datoreaza faptului ca, pe masura ce trie-ul contine mai multe cuvinte, nu mai trebuie efectuate la fel de multe operatii costisitoare de alocare de memorie si se poate face o parcurgere directa a trie-ului, pe cand in hashtable, hash-ul inca trebuie calculat indiferent de situatie.

Daca se compara timpii dintre cele doua structuri se observa clar ca inserarea si stergerea intr-un trie sunt operatii mult mai lente decat intr-un hashtable. Asta se datoreaza unei observatii facute anterior, aceea ca, desi cele doua structuri au aceleasi complexitati, trie-ul are un factor constant mult mai mare. Diferenta de timp se datoreaza alocarilor si dealocarilor de memorie, iar in cazul stergerilor se mai adauga si faptul ca tot cuvantul este parcurs de doua ori in cea mai rea situtatie.

O alta concluzie care se poate trage din ultimele teste din tabel este ca afisarea trie-ului este mult mai rapida. Acest lucru se datoreaza faptului ca prin folosirea unui trie cuvintele se pot afisa in mod direct in ordine lexicografica, pe cand in hashtable acestea mai trebuie si ordonate intai. Complexitatea afisarii in trie este de $O(n)$ pe cand in hashtable este $O(nlog(n))$, complexitate care reiese din sortare.

Distante de editare Din analiza complexitatii algoritmilor a rezultat ca acestia depind doar de dimensiunea cuvintelor pentru care se calculeaza distanta, fapt care este verificat de testele 9 - 12. Testele 13 - 17 vazute in comparatie cu testele 18 - 22 arata ca euristica lui Ukkonen face ca algoritmul Wagner-Fischer sa fie mai rapid atunci cand este stiuta din prealabil o distanta maxima. Totusi testele 23, 24, 25 in comparatie cu testele 26, 27, 28 mai arata ca euristica se aplica cel mai bine atunci cand limita pentru distanta este mai mica decat lungimile cuvintelor. In testele de la final, cuvintele sunt deja foarte scurte, asa ca euristica nu duce la imbunatatiri semnificative in ceea ce priveste timpul.

Chiar si cu aceste observatii, e evident ca algoritmul Wagner-Fischer, chiar si cu imbunatatiri, este tot mai lent decat alti algoritmi. Se mai observa din testele 13 - 22 ca algoritmul de referinta nu este afectat de limitele pentru distanta.

4 Concluzii

Trie Tinand cont de rezultatele testelor, la fel ca pe hashtable, timpii de cautare pe trie nu depasesc niciodata mai mult de 0.2s, asta si pentru foarte multe operatii de cautare. Daca se poate suporta costul unor inserari initiale, pentru a pregati structura, atunci se poate profita si de celelalte avantaje pe care le ofera trie-ul, precum implementarea usoara a completarii automate.

Desi trie-ul pare mai lent in general, observatiile referitoare la timpi se aplica pentru un hashtable care are toate galetile prealocate. Daca se tine cont de faptul ca in practica hastable-ul pleaca de la o dimensiune mai mica si apoi se redimensioneaza, de exemplu in implementarea de hashtable din java, atunci mai trebuie luat in calcul si timpul pe care hashtable-ul il petrece pentru redimensionari. Astfel, sunt sanse mari ca hashtable-ul sa nu mai aiba un avantaj atat de mare in fata trie-ului in ceea ce priveste inserarile.

Toate concluziile trase anterior se aplica pentru implementarile testate. In practica, se pot folosi variante optimizate ale trie-ului precum arborele ternar de cautare sau arbori Patricia.

Distante de editare O concluzie evidenta care reiese din testarea algoritmului Wagner-Fischer este ca acest algoritm este mult prea lent pentru utilizarea in practica, in special daca se calculeaza distante pentru cuvinte lungi. Totusi, este un algoritm usor de inteles, implementat si de optimizat, deci poate fi bun in scopuri educative. In acelasi timp are si o importanta istorica, fiind punctul de plecare pentru alti algoritmi mai eficienti precum algoritmul lui Ukkonen sau algoritmul Berghel-Roach care este o optimizare pentru algoritmul lui Ukkonen.

Din analizele lui Damerau, a reiesit ca, de cele mai multe ori, utilizatorii fac greseli care se incadreaza in una din cele 4 operatii definite de distanta Damerau-Levenshtein[8]. De aceea, pana in ziua de azi, inca se mai dezvolta algoritmi eficienti pentru calcularea acestei distante[14].

5 Bibliografie

1. Apache lucene, o librarie pentru motoare de cautare. URL https://lucene.apache.org.
2. Merge-sort pe liste inlantuite. 26 Noi 2021. URL https://www.geeksforgeeks.org/merge-sort-for-linked-list.
3. Niste proprietati pe care le au distantele de editare. 3 Mai 2016. URL https://stackoverflow.com/questions/36994342/generate-test-cases-for-levenshtein-distance-implementation-with-quickcheck.
4. O librarie de python care calculeaza distante. 9 Oct 2021. URL https://pypi.org/project/editdistance/.
5. Cuvinte in limba engleza in format text. Octombrie 2020. URL https://github.com/dwyl/english-words/.
6. Implementarea din libstdc++ foloseste arbori rosu-negru. Septembrie 2021. URL https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/bits/stl_multiset.h#L118.
7. Hal Berghel and David Roach. An extension of ukkonen's enhanced dynamic programming asm algorithm. 8 Iul 1996. URL http://www.berghel.net/publications/asm/asm.php.
8. Fred J. Damerau. A technique for computer detection and correction of spelling errors. Mar 1964. URL https://doi.org/10.1145/363958.363994.
9. Daniel Ellard. Tries. 1998. URL https://ellard.org/dan/www/libsq/cb_1998/c06.pdf.
10. Heikki Hyyr. Explaining and extending the bit-parallel approximate string matching algorithm of myers. Oct 2002. URL http://slab.cis.nagasaki-u.ac.jp/~ooya/Hyy01.pdf.
11. Vaidehi Joshi. Trying to understand tries. 31 Iulie 2017. URL https://medium.com/basecs/trying-to-understand-tries-3ec6bede0014.
12. Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. Mai 1999. URL http://www.gersteinlab.org/courses/452/09-spring/pdf/Myers.pdf.
13. Esko Ukkonen. Algorithms for approximate string matching. Ianuarie-Martie 1985. URL https://www.sciencedirect.com/science/article/pii/S0019995885800462.
14. Chunchun Zhao and Sartaj Sahni. String correction using the damerau-levenshtein distance. 6 Iun 2019. URL https://bmcbioinformatics.biomedcentral.com/articles/10.1186/s12859-019-2819-0.