Verslag Algoritmen en Datastructuren 3

Arnoud De Jonge

Academiejaar 2020-2021

1 Onderzoek

1.1 Algoritmes voor het zoeken

Mijn eerste poging was om te starten met ShiftAnd en op de gevonden matches de editeerafstand te berekenen om zo te controleren of het een geldige match was. Het was onmiddellijk duidelijk dat dit niet zal werken voor "Brughe" als men "Brugge" bedoelde.

Het volgende idee was om enkel de editeerafstand te gebruiken, dit werkt maar het was minstens twee maal zo traag in vergelijking met het eerste idee. We willen dus zo weinig mogelijk het algoritme uitvoeren. We zullen dit doen door het niet uit te voeren indien het zeker is dat er geen match zal zijn. Aangezien een drie of meerdere letters toevoegen of verwijderen equivalent is met een ongeldige match kunnen we eerst controleren op de lengte voordat we het algoritme uitvoeren. Dus indien het verschil in lengte tussen de twee input strings groter of gelijk is aan 3, zullen we overgaan naar de volgende mogelijke match. Dit idee wordt toegepast door in het algoritme de lengtes te vergelijken en 4 terug te geven indien de lengte te veel verschilt. Het is nu lichtjes trager dan het eerste optie, diegene met eerst ShiftAnd gebruikt en dan pas op editeerafstand zal controleren, maar het werkt nu wel voor "Brughe". Merk op dat je dit deel later overbodig werd door de woorden in de databank te sorteren op lengte.

Merk op dat er in de opgave $<= min(3, ...)$ staat. Ik probeer ook enkel kleiner dan te gebruiken. Dit zal betere matches geven voor vrij specifieke, correcte matches aangezien we enkel de editeerafstand gebruiken. Indien we met kleiner dan werken krijgen we voor "De Sterre Gent" 5 keer "De Sterre" telkens in de buurt van Gent. Indien we met kleiner of gelijk aan werken krijgen we nog steeds als de beste matches "De Sterre" in Gent, maar daarna komen er wel enkele matches in Antwerpen zoals "Siberië". Maar voor matches met meer tikfouten toch iets beter te ondersteunen zullen we toch kleiner dan of gelijk aan gebruiken. Dit geeft als nadeel wel dat het veel trager is voor langere zoek strings, bijvoorbeeld 4 of meer woorden. Dit komt omdat we veel meer matches vinden en dus ook meer totale matches moeten beoordelen. Een effect op de kwaliteit van de beste match is er alleen indien de editeerafstand van de bedoelde match exact 3 was. Anders krijg je dus nog steeds de plaats die je verwachte. Het probleem van efficiëntie met een input met veel woorden zullen we proberen op te lossen door te zoeken op $<= min(3, ...)$ als er 3 of minder woorden zijn en we gebruiken $< min(3, ...)$ als er 4 of meer woorden in de input zijn. Dit zal het probleem deels oplossen, maar als je veel meer dan 4 woorden ingeeft zal het nog steeds heel traag gaan.

1.2 Preprocessing

De poging was om de data te sorteren op eerste letter zodat de ShiftAnd sneller werkte, maar dit had natuurlijk geen effect omdat het het ook een substring

kan zijn en het woord "vislaan" ook in "de vislaan" kan staan. Dit is totaal nutteloos.

Wat wel kan als preprocessing is op lengte sorteren. Dat moeten we enkel degene doorzoeken die een lengte tussen $zoeklengte - 3$ en $zoeklengte + 3$ hebben. Dit kunnen we implementeren in de datastructuur van de databank of in de editeerafstand zelf. Eerst had ik gekozen om dit in editeerafstand zelf te doen (zie sectie 1.1). Hierdoor moeten we de databank niet sorteren op lengte en hoeven we geen indexen vanaf waar tot waar we moeten zoeken voor lengte. Een klein nadeel is dat we dan wel voor elke entry van de data moeten een functieoproep doen naar editeerafstand en de lengte vergelijken. Maar dit zal uiteindelijk sneller zijn dan de volledige databank te sorteren en indexeren.

Maar na het toevoegen van UTF-8 support moeten we elke zoekstring en mogelijke match (momenteel dus elke lijn in de databank) eerst sanitizen door de accenten te verwijderen zodat we de juiste lengte hebben. Hierdoor is het niet 1 enkele 'if' per keer dat je editeerafstand te veel oproept, maar een volledig woord overlopen en accenten & hoofdletters verwijderen. Hierdoor heb ik dan toch besloten om de data te sorteren op sanitized lengte. Hierdoor duurt het inlezen veel langer door het sorteren van de databank. Maar het opzoeken gaat veel sneller, zeker voor heel lange of juist heel korte zoekstrings. Dit komt omdat er voor deze woorden niet veel entries in de database zijn die tussen de $zoeklengte + 3$ en $zoeklengte - 3$ van de zoekstring liggen.

1.3 UTF-8

Voor accenten en hoofdletters te verwijderen gebruiken we de sanitize functie.

Aangezien we geen ShiftAnd meer gebruiken is dit niet meer relevant. Maar het idee was om een soort mapping te maken voor ShiftAnd. (zie afbeelding)

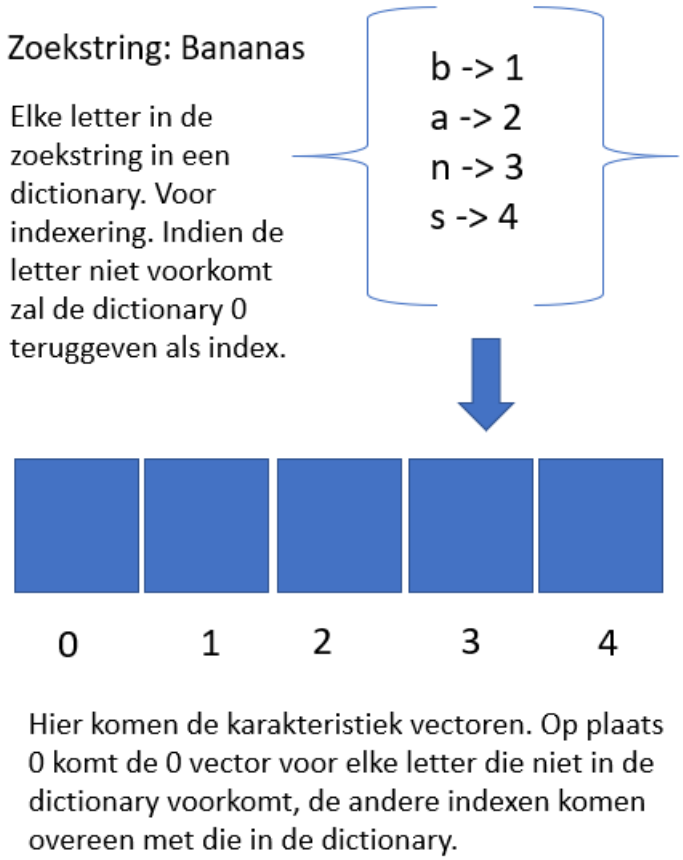

Figure 1: Example Mapping

Het voordeel hiervan is dat het altijd veel geheugen gaat besparen. Tenzij dat men een overdreven lange zoekstring ingeeft die bijna alle karakters van UTF-8 bevat, maar dit zal normaal gezien niet voorkomen. De snelheid is iets trager zijn door het opzoeken maar zo goed als verwaarloosbaar. En het zeker verwaarloosbaar als je kijkt naar de winst in geheugen.

2 Implementatiekeuzes

Vaak moeten we een keuze maken die ofwel het programma sneller maakt maar meer geheugen gebruikt ofwel minder geheugen gebruikt maar daardoor het programma trager werkt.
Voor de volgende keuzes maken we telkens gebruik van het voorbeeld "De Sterre Gent".

2.1 Opnieuw opzoeken VS Opslaan van matches

Voor het opzoeken zullen we alles maar 1 keer opzoeken en opslaan, dit zal meer geheugen gebruiken maar zal sneller werken. Indien we per opsplitsing, zoals ["De", "Sterre", "Gent"], zouden zoeken en het geheugen daarna weer vrijgeven zullen we bij de volgende opsplitsing: ["De", "Sterre Gent"] nogmaals "De" moeten opzoeken. Daarom zullen we alle mogelijkheden (hier: "De", "Sterre", "Gent", "De Sterre", "Sterre Gent" en "De Sterre Gent") eenmaal opzoeken en opslaan. Dit kan in een triangulaire matrix, in de code: 'match_matrix'.
In dit voorbeeld zullen we het dubbel opzoeken van "De" en "Gent" vermijden, maar voor grotere voorbeelden zal dit nog meer doorwegen en zullen we dingen die 5 keer werden opgezocht ook reduceren tot 1 keer.

2.2 Mogelijke opsplitsingen

Als we voor alle mogelijke combinaties de string nog eens op te slaan zou het geheugen snel vol lopen. Daarom zullen we getallen bijhouden in plaats van de woorden zelf. Wat nog beter is is in plaats van voor elke combinatie een array van getallen bij te houden gewoon 1 array bijhouden en een functie die de volgende berekent. Dit zal nog meer geheugen besparen. Aangezien bij weinig input woorden (hier 3: De Sterre Gent) zal het niet zo veel effect hebben. Maar bij langere zoekstrings (met meer woorden) kan je het effect wel duidelijk zien.

Om de 2^{n-1} opsplitsingen, met n het aantal woorden, te maken is het idee als volgt: We kunnen tussen elke woord, dus op $n-1$ plaatsen, mogelijk splitsen. We moeten dus voor elk van deze $n-1$ plaatsen kiezen of we daar splitsen of niet, zo komen we aan 2^{n-1} opsplitsingen. We kunnen dan een for loop van een getal 0 tot 2^{n-1} en dan naar de bits kijken waar de opsplitsingen zitten. Merk op dat indien $n > 64$ dit niet zal werken maar zo een groot aantal woorden is een onrealistische input voor ons programma.

2.3 Kleine details

Als we totale matches aan het berekenen zijn moeten voor bijvoorbeeld de opsplitsing ["De", "Sterre", "Gent"] alle combinaties zoeken tussen de matches van "De", "Sterre" en "Gent". Indien we al bij "De" geen matches zouden vinden kunnen we onmiddellijk stoppen want er zullen dan ook geen totale matches zijn. Hierdoor kunnen we onmiddellijk door naar de volgende opsplitsing.

2.4 Dynamisch alloceren van geheugen

Voor geheugen te alloceren is er altijd een vaste grootte (niet al te groot). Indien het er niet inpast zullen we de het gealloceerde geheugen verdubbelen. Zoals we gezien hebben in AD2 is dit een efficiënte manier. Indien na het alloceren het geheugen niet onmiddellijk vrijgegeven wordt, bijvoorbeeld de entries uit de databank, zullen we ook realloceren om het ongebruikte gealloceerde geheugen terug vrij te geven.

2.5 Bijhouden van totale matches

In het begin voor het bereken van de scores van de totale matches berekenen we alle scores van alle totale matches en sorteren we die dan. We houden dan alle totale matches bij wat niet nodig is. We kunnen telkens de beste 5 bijhouden wat heel wat geheugen zal uitsparen. We zullen ook niet gesorteerd toevoegen en pas op het einde 1 keer de top vijf totale matches sorteren.

3 snelheids analyse

3.1 Preprocessing van de databank

3.1.1 Databank inlezen

Lees de databank één maal = lengte databank = d

3.1.2 Databank sorteren op lengte

Sorteer de databank, vervolgens overloop je de data om de indexen te bepalen = d*log(d) + d = $O(d * log(d))$

3.2 Opzoeking voor 1 lijn input

Voor één lijn input moeten we de beste totale matches bepalen.

3.2.1 Opsplitsingen bepalen

Eerst bepalen we alle mogelijke opsplitsingen en vervolgens zoeken we deze op in de databank = $2^{n-1} * d' * k$
Met n het aantal woorden, d' een deel van de databank dat een match kan zijn op basis van de lengte en k de complexiteit van de editeerafstand tussen de twee woorden = $O(s1 * s2)$ met s1 de lengte van woord 1 en s2 is dan de lengte van woord 2 (zie lemma 21 in de cursus). Hierdoor moeten we niet de volledige databank d overlopen maar slechts een significant kleiner deel, daarom gebruiken we d' (zie sectie 1.2). Merk op dat 2^{n-1} niet altijd zo groot is aangezien sommige matches dubbel zijn en we die niet nogmaals opzoeken maar zullen hergebruiken (zie sectie 2.1).

3.2.2 Voor één zo een opsplitsing uit de vorige sectie

We berekenen nu alle totale matches en de score hiervoor en controleren vervolgens of deze in de top 5 valt. Dit is gelijk aan het aantal totale matches t = product van matches voor $a_1...a_j$ met j het aantal woorden in de opsplitsing en a_i het aantal matches voor deel i in de opsplitsing.
Om de scores te berekenen zullen we telkens loopen over $w_1...w_j$ met j het aantal woorden in de opsplitsing en w_i het woord/de woorden zelf op positie i in de opsplitsing.
voor één totale match = 2 * j (1x voor synergie en 1x voor correctheid)

3.2.3 Top 5 van alle totale matches bepalen en printen

Nu moeten we de beste 5 totale matches sorteren en printen, maar dit is gelijk aan 5log(5) = constante (want er kunnen maximum 5 totale matches zijn die we willen sorteren).

3.3 Conclusie complexiteit

O(databank inlezen + aantal lijnen input * (alle woorden opzoeken + voor elke totale match de score bereken)).

Met gebruikte afkortingen zoals hierboven en het aantal input lijnen = i:
0(databank preprocessing + elke input lijn opzoeken en beste matches printen)
$= 0(d * log(d) + i * (2^{n-1} * d' * k + 2^{n-1} * t * j))$
$= 0(d * log(d) + i * (2^{n-1} * (d' * k + t * j))$

De snelheid van het programma hangt dus sterk af van de lengte van de databank voor het preprocessen en voor het opzoeken hangt het vooral af van het aantal woorden in een input lijn (en natuurlijk ook van hoeveelheid input lijnen).

4 Testen

Het testen bestaat uit twee delen. Het eerste deel zijn basis testen die één bepaalde functie testen, het tweede deel zijn docker testen. Deze testen kunnen uitgevoerd worden aan de hand van het shellscript.

4.1 Basis testen

Deze testen zijn kleine testjes die één functie van de code testen, de volgende functies werden getest.

- Sanitizer: correcte werking UTF-8, hoofdletters & accenten.
- Combinaties: correcte combinaties opstellen.
- Editeerafstand: correcte berekeningen van de editeerafstand.

4.2 Docker testen

Deze testen kijken enkel naar de beste match en vergelijken deze met de verwachte locatie. De volgende aspecten van het programma werden getest:

- Basis: gewone input zonder fouten.
- Met fouten: input met editeerafstand kleiner of gelijk aan 3.
- Hoofdletters & accenten: Input met hoofdletters en accenten.
- Meerdere input lijnen: Zorgen dat meerdere lijnen verwerkt kunnen worden.
- Met coördinaten: Zorgen dat we de juiste plaats krijgen rond een bepaalde locatie.