PROJECT ALGORITMEN & DATASTRUCTUREN 3: VERSLAG

1) Initiële implementatie

Er zijn een aantal implementaties die al vanaf het begin zo optimaal mogelijk geïmplementeerd zijn, of die belangrijk zijn voor de latere optimalisaties.

1.1 Databank inlezen

De databank wordt op een simpele manier ingelezen: elke plaats wordt opgeslaan als een element in een linked list. Een linked list houdt het eerste en laatste element bij, en de lengte van de lijst.

1.2 Query opsplitsen

De queries worden gesplitst volgens een bitpatroon. We weten dat de query uit n woorden bestaat, en dus n-1 spaties. Elk bitpatroon stelt voor welke spaties een "separator" teken moet worden. Zo kunnen we alle opsplitsingen van 0 tot 2^{n-1}-1 overlopen (in totaal dus 2^{n-1}).

1.3 Shift-AND algoritme

Het shift-AND (met fouten) algoritme heeft een paar lichte aanpassingen gekregen, zodat het zo goed mogelijk gebruikt kan worden voor bounding (zie later in het verslag).

- De originele shift-AND met fouten (dat we in de les hebben gezien) kijkt enkel naar vervanging van tekens. Hier zijn insertion en deletion ook als toegestane fouten geïmplementeerd.
- Stel je hebt een zoekstring z, en een tekst t. Als de lengte van z groter is dan die van t, dan controleren we nu of t in z zit (dus het omgekeerde). De volgorde maakt dus niet zo veel meer uit.

1.4 Editeer afstand

De editeer afstand die we in de les hebben gezien, voldoet niet aan de editeer afstand van het project. Het algoritme moet dus aangepast worden om het mogelijk te maken om 2 lettertekens te verwisselen met een kost van 1.

1.5 Overlopen van queries

Stel dat een query op k manieren kan worden opgesplitst, dan wordt elk van deze opsplitsingen appart behandeld. Dit zorgt voor minder complexe code, en minder informatie die tussentijds moet worden opgeslaan.

1.6 Opslaan van de beste totale matches

De linked list krijgt een extra methode, "sorted insert". Deze methode voegt een element toe op basis van vergelijkingen met de andere elementen in de lijst. Het kan ook zijn dat het element helemaal niet wordt toegevoegd, aangezien de methode meekrijgt hoeveel elementen er maximum in de lijst mogen zijn. Hier bevat die lijst dus maximaal 5 elementen, en is de lijst op elk moment gesorteerd.

1.7 Combineren & score berekening van totale matches

Wanneer we alle matches hebben gevonden van de verschillende delen van een query (dus de M_i uit de opgave) combineren we deze met een niet-recursief algoritme. Het algoritme houdt een lijst van indices bij voor elke verzameling M_i, en deze indices zo update dat alle mogelijk combinaties worden overlopen, en indien nodig toegevoegd worden aan de top 5 matches.

2) Optimalisatie

De implementatie waarvan we in dit deel beginnen, is zoals hierboven beschreven. Merk wel op dat de editeer afstand hier nog maar deels met UTF8 is geïmplementeerd, en het shift-AND algoritme nog helemaal niet.

We doen tijdsmetingen voor de volgende queries:

bosleentje lovendegem
vrijheidstraat bruhe
De Sterre Gent
hoge bosstraat eeklo

Hier is “hoge bosstraat eeklo” een interessante zoekopdracht, aangezien deze heel veel matches vindt bij de opsplitsing “hoge | bosstraat | eeklo”.

2.1 Bounding: shift-AND

Het shift-AND algoritme kan gebruikt worden als een bounding stap vóór het effectief berekenen van de editeer afstand. Zo hoeven we niet voor elke plaats in de databank de editeer afstand te berekenen, wat in de praktijk een trager algoritme is dan shift-AND.

Shift-AND is geïmplementeerd met insertion, deletion en substitution, maar niet met het wisselen van 2 tekens. We zullen daarom 2 maal de maximale editeer afstand aan fouten toelaten. Een verwisseling kan namelijk ook in minimaal 2 stappen door de 2 lettertekens die je zou wisselen, te vervangen, wat kost 2 is.

Als een (deel van een) query niet in de plaatsnaam te vinden is - of omgekeerd - met maximaal dit aantal fouten, zal de editeer afstand zeker een te hoge kost hebben. Naar die plaatsen moeten we dus niet meer kijken.

	Zonder filteren	Filteren met shift-AND
bosleentje lovendegem	2.620s	0.991s
vrijheidstraat bruhe	2.433s	1.227s
De Sterre Gent	3.376s	2.746s
hoge bosstraat eeklo	16.714s	15.507s

We zien dat er inderdaad een goede impact is bij het filteren met shift-AND. Bij de eerste 2 queries krijgen we zelfs een halvering van de tijd.

2.2 Optimalisatie: editeer afstand

De editeer afstand kan beter geïmplementeerd worden, we kunnen namelijk zien dat niet op elk moment heel de matrix wordt gebruikt. Het volstaat om gewoon een deel van de matrix bij te houden, namelijk 3 rijen, aangezien we op elk moment maar de 2 vorige rijen nodig hebben, en 1 rij voor de huidige berekeningen.

	Editeer afstand met matrix	Editeer afstand met 3 rijen
bosleentje lovendegem	0.991s	0.714s
vrijheidstraat bruhe	1.227s	0.741s
De Sterre Gent	2.746s	1.608s
hoge bosstraat eeklo	15.507s	12.430s

We zien dat de tijdsmetingen hierdoor terug zijn verminderd. Toch valt het op dat we voor een query met veel matches wat traag te zijn.

2.3 Bounding: matches alloceren

Momenteel wordt elke totale match gealloceerd in het geheugen, ookal weten we dat die niet in de lijst zal komen door te vergelijken met het laatste element van de lijst. Om dit te vermijden alloceren we nu pas geheugen als we weten dat het element effectief in de lijst zal komen.

	Alle matches alloceren	Alloceren wanneer nodig
bosleentje lovendegem	0.714s	0.722s
vrijheidstraat bruhe	0.741s	0.792s
De Sterre Gent	1.608s	1.576s
hoge bosstraat eeklo	12.430s	12.429s

Dit geeft niet veel verschil, dus dit was geen groot probleem. Het lijkt echter betere “practice” om deze code te houden, en enkel te alloceren wanneer nodig.

2.4 Bounding: string lengtes
We kunnen de string lengtes van de 2 te vergelijken strings als een goede ondergrens gebruiken voor de editeer afstand. Als string lengtes meer dan 2 van elkaar verschillen, dan is de editeer afstand minstens 3, omdat er al 3 of meer insertions/deletions nodig zouden zijn. Dit heeft een kost van oneindig volgens de opgave, en zal dus zeker niet in de top matches komen.

Tot nu toe filteren we eerst op shift-AND, maar deze shift-AND berekenen is eigenlijk enkel nodig als de lengtes minder dan 3 van elkaar verschillen. We kunnen nu op voorhand het verschil tussen de lengtes berekenen.

	Zonder filteren op lengte	Met filteren op lengte
bosleentje lovendegem	0.722s	0.288s
vrijheidstraat bruhe	0.792s	0.283s
De Sterre Gent	1.576s	0.468s
hoge bosstraat eeklo	12.429s	8.744s

Dit is ondertussen een van de optimalisaties met de beste resultaten, we zien bijna overal meer dan een halvering van de tijd, behalve bij de laatste zoekopdracht. Er is daar dus nog steeds een “bottle neck” die moet opgelost worden.

2.4 Preprocessing: databank
Een nieuw idee om de databank te overlopen, is om de plaatsen in de databank op te slaan in lijsten met plaatsen van dezelfde lengte. Zo krijgen we dus een array met op index i, alle plaatsen waarvan de plaatsnaam lengte i+1 heeft.

We kunnen bij het overlopen van de databank enkel de lijsten overlopen die de plaatsnamen bevatten waarvan de lengte minder dan 3 verschilt met wat we zoeken.

	Databank als lijst	Databank als array van lijsten
bosleentje lovendegem	0.288s	0.321s
vrijheidstraat bruhe	0.283s	0.291s
De Sterre Gent	0.468s	0.472s
hoge bosstraat eeklo	8.744s	8.430s

Dit geeft geen groot verschil, en duurt zelfs in sommige gevallen langer aangezien de databank inlezen nu iets langer zal duren. We gaan hier niet verder op werken, en laten deze implementatie voor wat het is. Ookal zou het interessant zijn om dit beter uit te werken, we hechten momenteel meer belang aan het probleem bij de laatste query.

2.5 Optimalisatie: combineren van matches (deel 1)
Als we meer specifiek de tijd meten, vinden we dat voor de opsplitsingen van “hoge | bosstraat | eeklo”, het combineren en berekenen van de totale matches 8.211 seconden inneemt.

Voor elke totale match doen we (bij queries van meerdere delen) meerdere ‘get’ oproepen op een linked list. Deze get overloopt de lijst tot hij het juiste element vindt. Als er heel veel matches zijn per deel van de query, zullen we grote lijsten krijgen, die veel overlopen moeten worden, wat de reden is voor de bottleneck.

De niet-recursieve methode is in deze situatie dus amper bruikbaar. We opteren hier beter voor een recursieve methode, waarbij we de lijsten overlopen en ondertussen de totale matches maken.

	Niet-recursief combineren	Recursief combineren
bosleentje lovendegem	0.288s	0.164s
vrijheidstraat bruhe	0.283s	0.162s
De Sterre Gent	0.468s	0.269s
hoge bosstraat eeklo	8.744s	2.557s

De tijd van het grote aantal combinaties van de laatste query is nu slechts 2.316 seconden meer, een grote verbetering. Ook de tijd bij de andere queries is bijna gehalveerd. Dit is dus een heel nuttige optimalisatie.

2.6 UTF8 ondersteuning
Nu we al veel hebben geoptimaliseerd, kunnen we starten met de editeer afstand en shift-AND algoritmen UTF8 te laten ondersteunen.

Bij Shift-AND is het geen goed idee om alle karakterestieke vectoren op te slaan voor het UTF8 alfabet, dat zijn er veel te veel. Wat we wel doen, is de vectoren maken voor het ASCII alfabet (dus 128 vectoren) en dan eventueel extra vectoren voor de UTF8 tekens in de zoekstring. We indexeren die dan als *128 + eerste positie in de zoekstring*. Tekens met accenten of hoofdletters worden eerst omgezet naar een kleine letter zonder accent, en vallen dan onder het ASCII alfabet of dus onder de eerste 128 karakteristieke vectoren.

De andere implementaties, voor bv. het opsplitsen van zoekopdrachten of tellen van woorden, moeten we niet aanpassen. De reden hiervoor is dat de 1-byte UTF8 tekens dezelfde encodering hebben als de ASCII tekens. Een spatie waarop we willen splitsen wordt dus nog steeds voorgesteld op dezelfde manier (nog steeds met 1 byte die begint met 0). De string kan dus nog steeds byte per byte (of dus char per char) overlopen worden, in plaats van per UTF8 teken.

	Minimale UTF8 ondersteuning	Volledige UTF8 ondersteuning
bosleentje lovendegem	0.164s	0.213s
vrijheidstraat bruhe	0.162s	0.202s
De Sterre Gent	0.269s	0.349s
hoge bosstraat eeklo	2.557s	2.890s

We zien dat dit het programma iets trager maakt, wat wel te verwachten was, aangezien we met meer moeten rekening houden (accenten, ...).

2.7 Optimalisatie: combineren van matches (deel 2)
Nu de laatste query bijna 3 seconden duurt, kan er misschien nog extra geoptimaliseerd worden in het recursief algoritme. We zouden namelijk tussentijdse resultaten kunnen berekenen, in plaats van dit pas te doen als we de volledige combinatie van de totale match hebben.

	Recursief combineren	Geoptimaliseerd recursief combineren
bosleentje lovendegem	0.213s	0.212s
vrijheidstraat bruhe	0.202s	0.201s
De Sterre Gent	0.349s	0.314s
hoge bosstraat eeklo	2.890s	1.293s

Zo zien we inderdaad nog een grote versnelling bij het combineren, maar op de kleinere zoekopdrachten heeft dit niet meer zo veel invloed.

3.0 Tijdscomplexiteit

We gebruiken volgende gegevens:

n = aantal woorden in de query
z = lengte van de query (aantal tekens)
m = aantal plaatsen in de databank
l = lengte van de langste databank entry (dus de langste lijn)
u = lengte van de langste plaatsnaam in de databank
k = aantal toegestane fouten voor shift AND met O(k) = 2*3 (3 = maximale editeer afstand, 2 komt uit de redenering in deel 2.1)
r = grootste aantal matches voor een deel van een query = O(m)

Voor de tijdscomplexititeit te berekenen, gaan we nu over de belangrijkste delen van de implementatie.

3.1 Inlezen van de databank

- Elke databank entry is *O(l)* lang.
- Overloop alle m lijnen en lees ze in, dus met kost *O(m*l)*.
- Sla elke plaats op in de linked list die de databank voorstelt. Dit kan in constante tijd aangezien de linked list het laatste element in de lijst bijhoudt.

De kost is hier *O(m*l)*.

3.2 Overlopen van de opsplitsingen van de query

- Filteren van de query overloopt heel de string, dus heeft kost z.
- Tel het aantal woorden in de query, dit kan ook met een kost van z. Het resultaat is n = aantal woorden.
- Overloop de 2^{n-1} mogelijke opsplitsingen, het genereren van 1 opsplitsing kost z, in totaal dus $2^{n-1}*z$

De kost is hier $O(2*z + 2^{n-1}*(z + \text{kost voor 1 opsplitsing}))$
$= O(2*z + 2^{n-1}*z + 2^{n-1}* \text{kost voor 1 opsplitsing})$

3.3 Opgesplitste query behandelen

- We gaan er hier van uit dat n een bovengrens is voor het aantal delen in een query, dus elke opgesplitste query heeft *O(n)* delen.
- Itereer over de databank (m elementen) en itereer per plaats in de databank, over de *O(n)* velden van de opgesplitste query.
- Ga nu uit van het slechtste geval: elke plaats in de databank zal een match zijn voor elk deel van de query.
- Voor deze *O(m*n)* opties berekenen we dus de lengtes, shift-AND, de editeer afstand en voegen we ze toe aan een lijst. Dit geeft een totatele kost van:

*O(m*n*(kost van lengtes + shift-AND + editeer afstand + toevoegen))*

- Kost van lengtes berekenen is *O(z+u)*.
- Shift-AND heeft een kost van *O(t*k*s)* met t = |tekst| en s = |zoekstring|. Aangezien *O(k) = 6*, is dit een constante en kunnen we dat laten vallen. Nu vervangen we t door u, en s door z, aangezien zij bovengrenzen geven voor de langste strings: *O(u*z)*,
- Editeer afstand kan ook in *O(u*z)*.
- Het toevoegen aan een linked list kan in constante tijd.

De totatele tijd om 1 opsplitsing van een query te behandelen is

*O(m*n*(z + u + 2*u*z) + kost matches combineren)*

Dit vereenvoudigen geeft *O(m*n*u*z + kost matches combineren).*

3.4 Alle mogelijke matches combineren voor 1 opsplitsing van een query

- Stel we hebben *O(n)* lijsten, elke lijst stelt de matches voor van een bepaald deel van de query en bevat *O(r)* matches. Het aantal mogelijke combinaties of totale matches is $O(r^n)$.
- Voor elke van deze totatele matches doen we berekeningen in constante tijd.
- De match toevoegen aan de lijst van beste matches, aan de hand van sorted insert, kan in constante tijd, aangezien er max 5 elementen in de lijst zitten. (Als we deze 5 niet als een constante zien, dan kost dit lineaire tijd.)

  De kost is hier $O(r^n)$

  Aangezien $r = O(m)$ in het slechtste geval waar we op elk element in de databank een match vinden, is de kost eigenlijk $O(m^n)$.

3.5 Besluit

Na al deze bekomen tijdscomplexiteiten te vereenvoudigen, en de belangrijkste delen te behouden, bekomen we:

$O(m*l) + O(2*z + 2^{n-1}*z + 2^{n-1}*(m*n*u*z + m^n))$

$= O(m*l + 2^{n-1}*(m*n*u*z + m^n))$

We zien dat het combineren van de mogelijke matches een probleem kan vormen voor de totale tijdscomplexiteit van het programma. We krijgen daar exponentiële tijd in functie van m en n, en dit voor elke opsplitsing. Dit is natuurlijk wel in het allerslechtste geval, in de praktijk zal het helemaal niet zo zijn dat heel de databank voor elk deel van de query matcht.

Dat het combineren van matches veel tijd kan nemen hebben we ook gezien bij het uitvoeren van “hoge bosstraat eeklo” en hoe dit kon zorgen voor een bottleneck, zeker als er geen efficiënte code gebruikt werd.