OpenSeekMap

Project Algoritmen en Datastructuren 3

Ieben Smessaert
UGent
9 december 2020

Opbouw

Als eerste werd er gekeken om de datasets te kunnen inladen en dit aan de hand van een performante manier, optimalisaties om dit te verwezenlijken zijn neergeschreven in de sectie onder *Optimalisaties*. Met een regex wordt elke database entry getoetst of het aan het vereiste formaat van een *Openstreetmap* element voldoet. Indien dit niet het geval is wordt een waarschuwing uitgeprint en naar het volgende element gegaan. We preprocessen de database door zowel de originele naam als een genormaliseerde versie op te slaan. Deze laatste wordt doorheen de algoritmes gebruikt om eenvoudig geen rekening te moeten houden met de operaties van kost 0 en na de ondersteuning van UTF-8 ook als een reeks van UTF-8 compatibele tekens op te slaan, d.i. als een reeks van *uint32_t*. Entries die een langere of gelijke lengte hebben dan 3 meer dan de langste query worden eruit gefilterd tijdens het inladen omdat deze nooit gematched kunnen worden, om deze te kunnen matchen zouden er namelijk minstens 3 toevoegingen nodig zijn aan de zoekstring wat niet toegestaan is. Wat de impact hiervan is op de uitvoeringstijd valt hieronder te lezen onder *Optimalisaties*.

Een volgende stap in het verloop van het programma is het verwerken van de ingevoerde queries door de gebruiker door deze eerst te verwerken. Uit een query berekenen we namelijk eerst alle mogelijke onderverdelingen. Dit doen we recursief, we splitsen op de spaties in de query en voor elk nieuw deeltje voegen we dit eenmaal toe als nieuwe opsplitsing en eenmaal als extra woord aan de laatste opsplitsing van alle reeds bestaande combinaties. Voor elk nieuw woord in de query lopen we dus over de reeds bestaande onderverdelingen, eenmaal breiden we die uit met een nieuwe onderverdeling, eenmaal dupliceren we deze maar passen we de laatste onderverdeling aan.

Nu komt het erop aan om alle onderverdelingen te toetsen aan onze ingeladen database met een van de algoritmes. Het toepassen van het algoritme op de query en een database entry is enkel zinvol als het tot een match kan leiden, d.i. de lengte tussen de twee verschilt met maximaal 2, anders zal de kost oneindig zijn en hoeven we het algoritme niet toe te passen. Extra uitleg bij deze bounding en welke impact deze heeft vindt u als tweede optimalisatie onder *Optimalisaties*.

Voor de algoritmes werd er in de eerste stap vertrokken vanaf het shiftAND algoritme die werd uitgebreid met fouten en vervolgens aangepast om een

volledige string te matchen met een zoekstring. Een eerste implementatie bestond eruit om alles in datastructuren op te slaan, de matrices voor 0, 1, ... fouten worden als een lijst van kolommen opgeslagen die op hun beurt worden voorgesteld door een lijst van getallen, die elk een waarde 0 of 1 kunnen zijn. Later werd er een tweede algoritme - editeerafstand - geïmplementeerd die zoals in de sectie onder *Optimalisaties* te zien is, een aanzienlijk snellere uitvoeringstijd in aantal instructies heeft. Uit de verontwaardiging van het grote verschil tussen beide algoritmes werd het eerste algoritme herschreven om minder gebruik te maken van eigen gedefinieerde datastructuren en het gebruik van mallocs in het mate van mogelijke te beperken, dit vertaalt zich zoals verwacht in een sneller algoritme. Verdere toelichting over de verschillende algoritmes worden hieronder besproken.

Als laatste stap worden de beste 5 matches bijgehouden door op basis van de meegeleverde en geïmplementeerde formules te kijken of de waarde van de gevonden match hoger is dan de reeds bestaande 5 best matches. Wordt een huidige beste match vervangen door een nieuwe, dan wordt de oude waarde nog vergeleken met de andere huidige waarden of deze misschien zijn plaats kan innemen voordat deze voor goed wordt uitgesloten als een van de 5 beste matches.

Algoritmes

shiftAND legacy (voor rewrite)

Als eerste implementatie om de kost te berekenen tussen de zoekstring (de query) en de tekst (de database entry) werd vertrokken vanaf het shiftAND algoritme. Die werd zodanig aangepast dat het met errors werkt en om een matching van de volledige string gaat. Hiertoe worden achtereenvolgend de matrices M_0, M_1, M_2 en M_3 berekend om na te gaan of een match met dat bepaald aantal fouten gevonden kan worden. Om toe te staan dat er een teken wordt toegevoegd, of een teken wordt verwijderd wordt er bij het bestaande algoritme nog extra bitwise operators aan toe gevoegd, meer bepaald een *OR shift(M^{l-1}[][j])* voor het toevoegen en *M^{l-1}[][j-1]* voor het verwijderen van een teken. Om toe te staan twee tekens te verwisselen van plaats wordt op dit geheel nog eens een bitwise *OR* toegepast met *((C[t[j-1]] & shift((M[][j-1])) & shift(C[t[j]] & M[][j-1]))*. Dit controleert of

zowel de karakteristieke vector van het vorige teken overeenkomt met het huidige en dat vorige teken ook al een waarde 1 had, zodat zonder meerkost ook dit teken op 1 gezet kan worden, en dat ook de karakteristieke vector van het huidige teken matcht met het vorige teken.

Met m de lengte van de zoekstring, n de lengte van de tekst (de naam in de database), w het aantal woorden in de zoekstring en e het aantal elementen in de databank kunnen we nu de tijdscomplexiteit bepalen. Hieruit halen we nu de complexiteit van het berekenen van de karakteristieke vectoren: $O(m^2)$ - voor elk karakter in de zoekstring wordt namelijk gekeken of er al een karakteristieke vector is aangemaakt die kan worden aangepast of als er een nieuwe moet worden aangemaakt. Als de zoekstring in het slechtste geval uit allemaal verschillende tekens bestaat zal in elke stap dus een nieuwe karakteristieke vector worden aangemaakt en zal voor het laatste teken n eerst de n-1 voorgaande karakteristieke vectoren moeten worden gecontroleerd. Het berekenen van de matrix M heeft een tijdscomplexiteit van O(n*m). Voor elk karakter in de tekst wordt namelijk de kolom berekend, het berekenen van die kolom omvat een constant aantal bewerkingen van elk O(m). Uit voorstaande twee volgt nu de tijdscomplexiteit van het volledige algoritme: $O(m^2 + n*m)$.

Om de tijdscomplexiteit van het volledige programma te berekenen, kijken we nu hoevaak dit algoritme wordt opgeroepen. Voor elke onderverdeling wordt er voor elk deeltje gekeken of er een match is met een element uit de database. Er zijn $(n+2)*2^{n-1}$ zo'n deeltjes en e elementen in de database waarvoor het algoritme moet worden opgeroepen. Zoals in het deel onder *Optimalisaties* besproken is dit gelukkig slechts een slechtste geval bovengrens van $O((n+2)*2^{n-1} * e * (m^2 + n*m))$, we hoeven namelijk niet altijd voor alle gevallen het algoritme toe te passen.

Editeerafstand

Een tweede weg om de kost te berekenen bestaat eruit het editeerafstand-algoritme te gebruiken. Ook dit algoritme werd aangepast om het wisselen van twee opeenvolgende karakters met kost 1 te ondersteunen, hiertoe werd een extra check toegevoegd die de wisselkost op 0 zet als het vorige karakter uit de zoekstring gelijk is aan het huidige karakter van de tekst én het huidige karakter in de zoekstring gelijk is aan het vorige karakter in de tekst.

Voor dit algoritme kunnen we de tijdscomplexiteit bepalen als O(m*n) omdat er een mxn matrix wordt opgebouwd waar overal slechts éénmaal wordt geïtereerd. Er worden vooraf geen karakteristieke vectoren berekend maar er wordt rechtstreeks gekeken of de 2 karakters gelijk zijn. Om de karakteristieke vectoren te berekenen wordt er namelijk ook herhaaldelijk gecontroleerd of de lettertekens gelijk zijn. Als we die vooraf berekenen moeten we tijdens het algoritme nog steeds de bits (0/1) vergelijken aangezien er geen bitwise operatoren mogelijk zijn daar de kost rechtstreeks wordt opgeslagen in de matrix. De tijdscomplexiteit van het totale programma wordt dan $O((n+2)*2^{n-1} * e * m*n))$.

shiftAND (na rewrite)

Uit de verbazing dat het algoritme dat meteen de editeerafstand berekent zoveel sneller gaat dan het shiftAND algoritme werd onderzocht waarom dit algoritme zoveel trager ging. Naast de optimalisaties onder *Optimalisaties* beschreven, werd het algoritme ook herschreven met oog voor het zoveel mogelijk reduceren van mallocs en eigen datastructuren en om de geheugentoegangen eenvoudiger te bereiken. Hiertoe wordt nu een tweedimensionale tabel gebruikt van integers om de matrix voor te stellen. Omdat de structuur van het algoritme gelijk is gebleven komen we voor de tijdscomplexiteit ook nog steeds op $O(m^2 + n*m)$ uit, met dezelfde verklaring als in het eerste shiftAND algoritme. Hoewel de complexiteit dezelfde is, wordt de uitvoeringstijd in #instructies toch gehalveerd. We zien dus dat het werk per stap ook zeer belangrijk is en niet enkel het aantal iteraties.

Optimalisaties

Eerst werd een gelinkte lijst gebruikt om de dataset in het geheugen te laden waarbij enkel een pointer naar het eerste element werd bijgehouden in de variabele head. Het programma had toen 29m30s nodig om alles in te laden en uit te printen (zonder dat reeds een algoritme voor het verwerken van de queries was geïmplementeerd). Dit werd geoptimaliseerd door ook steeds een pointer naar het laatste element in de gelinkte lijst (tail) bij te houden, zodat het toevoegen van een nieuwe entry sneller kon gebeuren. Dit had een enorm effect op de uitvoeringstijd, het inlezen en terug uitprinten naar stdout van alle 169982 entries kostte hierna maar 23s meer.

Na het volledig implementeren van het algoritme en linken tot een eerste werkend programma kost het zoeken naar de **queries "De Sterre Gent" en "Vrijheidstraat Brugge"** maar liefst 78,488,694,867 instructies. Voor enkel de de **query "De Sterre Gent"** kost het ons 46,740,157,598 instructies.

Om dit te meten wordt gebruik gemaakt van de *callgrind* tool binnen *valgrind*, we kijken hiervoor naar het *I refs* resultaat. De queries worden uitgevoerd op de *belgie-ascii* dataset en er worden geen coordinaten als argument meegegeven.

Een eerste verder optimalisatie bestaat erin om enkel de database entries in het geheugen in te laden die een kans maken om voor onze queries gematcht te worden, dit doen we door eerst de queries in te lezen en te kijken wat de lengte is van de langste query. Vervolgens worden enkel queries opgeslagen die kleiner zijn dan de max lengte + 3, dit zijn de 3 toegestane fouten om 3 karakters aan de query toe te voegen en zo te matchen met een entry. Als we nu het programma met deze optimalisatie opnieuw runnen, bekomen we voor de 2 queries een totaal instructies van 68,603,820,177 en voor enkel de eerste query 30,393,713,883 instructies. Dit is dus al meteen een derde van het aantal instructies of dus de uitvoeringstijd die we kunnen reduceren bij de enkele query. De reden dat dit zo'n impact heeft, is het feit dat we voor elk deeltje van elke query-onderverdeling over minder database entries moeten itereren om te controleren op een mogelijke match.

Een tweede optimalisatie volgt ietwat uit de eerste, namelijk door te controleren of het berekenen van de kost met het aangepaste shiftAND algoritme wel zin heeft. Hiervoor kijken we vooraf of we op basis van de lengte van de query en van de database entry al kunnen uitsluiten dat een match niet mogelijk is. Meer

concreet toetsen we vooraf of volgende voldaan is: $|entry| \leq |query| + 3$ *and* $|entry| \geq |query| - 3$ en $|entry| \leq |query| + (1+|query|/3)$ *and* $|entry| \geq |query| - (1+|query|/3)$. Na het toepassen van deze optimalisatie krijgen we voor het oproepen van het programma met beide queries een totaal van 18,218,054,294 instructies, een drastische vermindering dus. Ook bij het oproepen met de enkele query zien we een drastische vermindering in het aantal instructies, namelijk nog maar een totaal van 9,279,466,073.

Een volgende optimalisatie zou kunnen zijn om bij het berekenen van de query opsplitsingen te kijken of een bepaald deeltje uit de zoekstring wel tot een mogelijke match kan leiden in vergelijking met de kleinste database entry. Tijdens het inladen van de dataset houden we het kleinste element bij en bij het berekenen van de mogelijke opsplitsingen toetsen we de lengte van beiden aan de maximale toegestane fout. Na het runnen van de 2 queries met ons aangepast programma krijgen we 18,228,188,377 instructies uitvoeringstijd. We zien dus dat de instructies terug gestegen zijn en dit geen goede optimalisatie was. Als we naar de reden hiervoor zoeken kunnen we de verklaring vinden in het feit dat er entries in de database van maar 1 karakter lang zitten, waardoor er dus nooit gefilterd gaat kunnen worden tijdens de berekening van de query-opsplitsingen. De extra instructie-kost zit hem in het bijhouden en controleren voor deze kost. We maken deze 'optimalisatie' dus terug ongedaan.

Door voor onze datastructuren M een extra pointer bij te houden naar de laatste kolom in plaats van steeds vanaf *head* te moeten itereren naar de laatste kolom, kunnen we nog meer tijd besparen. Het verschil in uitvoeringstijd is echter klein, maar wel beter. We kunnen de optimalisatie dus behouden. Voor de dubbele query gaan we van 23,787,619,231[1] instructies naar 23,723,302,702 en voor de enkele query van 11,876,786,829 naar 11,845,453,298.

[1] We starten van een hoger aantal instructies dan dat we bij de vorige optimalisatie zijn geëindigd door het feit dat in tussentijd UTF-8 ondersteuning en lettertekens wisselen met kost 1 werd geïmplementeerd. Vanaf dit punt wordt het programma opgeroepen op de *belgie* dataset (UTF-8).

	Enkele query	Dubbele query
shiftAND voor rewrite	13,839,229,596	22,239,385,209
edit_distance	1,945,977,638	2,324,559,594
shiftAND na rewrite	6,084,204,024	10,791,168,242

Tabel 1: Overzicht van de uitvoeringstijden in #instructies per algoritme

Ook het edit_distance-algoritme kunnen we verder optimaliseren. Een manier die werd uitgeprobeerd, en nuttig bleek is door het berekenen van de afstand vroegtijdig te stoppen wanneer de kleinste waarde voor een volledige rij al groter bleek dan de maximaal toegestane kost, want dan kunnen we nog onmogelijk aan een lagere afstand komen op het einde. Dit brengt ons tot 2,335,438,426 instructies voor de dubbele query en 1,995,903,850 voor de enkele, wat toch weer een stuk minder is dan ons vorige eindresultaat.

In bovenstaande tabel (Tabel 1) wordt de uitvoeringstijd in instructies van de verschillende algoritmes op een rij gezet om een duidelijk overzicht te krijgen in de verschillen tussen de drie. Dit is bij het toepassen van alle hierboven beschreven optimalisaties. We zien dus dat het *edit_distance* algoritme duidelijker sneller is dan de andere twee. Bij het oproepen van het programma kan een algoritme gekozen worden met behulp van de *--algorithm* optie. Een uitvoeringstijd in seconden kan verkregen worden via *--time*. In onderstaande tabel (Tabel 2) staat ter volledigheid de uitvoeringstijd gemeten in aantal seconden. Omdat dit afhankelijk is van de omgeving waarin het programma wordt uitgevoerd wordt deze slechts eenmalig toegevoegd om een verder idee te geven van de uitvoeringstijd.

	Enkele query	Dubbele query
shiftAND voor rewrite	2.835465	4.666419
edit_distance	0.432851	0.559455
shiftAND na rewrite	1.555803	2.018884

Tabel 2: Overzicht van de uitvoeringstijden in seconden per algoritme