Verslag - Tom Alard

Ontwerpbeslissingen

Encoder

Bepalen van een optimale partitie

Het eerste probleem dat je tegenkomt is hoe je je ingelezen buffer van karakters gaat opsplitsen in de labels die je dan gaat encoderen. Voor normale adaptive huffman (LEN = 1) bestaat dit probleem natuurlijk niet, je leest 1 karakter in per iteratie en encodeert die direct. Als we labels willen kunnen toevoegen die groter zijn dan 1 byte, zal dit echter niet meer lukken.

Mijn eerste idee was om als volgt te werk te gaan: Lees de eerste LEN bytes in van de buffer b. Bepaal dan of we het label `b[0..LEN]` willen toevoegen of niet. Als het antwoord ja is, encodeer dan dit label en ga verder met `b[LEN+1..2*LEN + 1]`, anders probeer je opnieuw voor `b[0..LEN-1]`, `b[0..LEN-2]` ... In het slechste geval encodeer je `b[0..1]` en doe je verder met `b[2..LEN+2]`. Uiteindelijk zul je een partitie bepaalt hebben voor de hele buffer. De tijdscomplexiteit van dit algoritme is `O(buffer_size * LEN)`, en de geheugencomplexiteit is `O(LEN)`. Op zich is dat redelijk goed, maar het grootste probleem met dit gretig algoritme is dat het geen optimale partitie garandeerd.

Stel bijvoorbeeld een buffer `xabc` en dat we bepaald hebben dat we de labels `xa` en `abc` willen toevoegen. Ons gretig algoritme zal de volgende partitie vinden: `xa|b|c`, terwijl de optimale partitie `x|abc` is. We noemen deze partitie beter omdat het aantal verschillende labels kleiner is. De gevonden partitie kan dus arbitrair slechter zijn dan de beste partitie. We kunnen dit probleem echter oplossen met een goede functie om te bepalen welke labels toe te voegen. Het is belangrijk dat deze functie de volgende eigenschap heeft: als een substring geldig is, is elke suffix van die substring ook geldig. Zo'n functie wordt in het volgende deel voorgesteld. Met deze eigenschap weten we dat het voorbeeld hierboven onmogelijk kan gebeuren. als `abc` geldig is, is `bc` dat ook. Dit gretig algoritme zal voor het voorbeeld dan `xa|bc` geven.

Al heeft deze partitie even veel cuts als `x|abc`, is het nog altijd mogelijk dat `x|abc` een betere partitie is. Als `xa` en/of `bc` bijvoorbeeld nog niet in de boom zitten, moeten we die toevoegen. Het toevoegen van een label is op zich al redelijk duur, en we hebben geen garantie dat dit label hergebruikt zal worden in de buffer door het algoritme, zelfs als het meerdere keren voorkomt. We willen dus een tiebreaker systeem implementeren. Dit systeem beslist, gegeven twee verschillende partities met even veel labels, welke de beste is. Ons gretig algoritme zal dit tiebreaker systeem echter nooit kunnen gebruiken, het bouwt slechts 1 partitie op, we kunnen dus aan het einde van het algoritme partities niet met elkaar vergelijken. We hebben dus een ander algoritme nodig dat alle mogelijke 'beste' partities overloopt, en dan met het tiebreaker systeem de beste kiest. Zo'n algoritme implementeren we hieronder.

Dit probleem is gelukkig heel gelijkaardig aan het volgende LeetCode probleem. In dat probleem wordt een string gesplitst in het minimaal aantal palindromen, maar we kunnen dit abstraheren naar het minimaal aantal geldige substrings. Een belangrijke gelijkenis tussen de twee problemen is dat substrings van lengte 1 altijd geldig zijn. Hierdoor zijn we zeker dat er sowieso een oplossing bestaat voor elk probleem. Dit algoritme garandeerd enkel een optimale partitie voor de buffer, niet voor de hele file. Daarvoor zouden we de hele file in 1 keer moeten inlezen, maar dan zou het algoritme niet meer online zijn.

Het algoritme dat we implementeren is dus sterk gelijkaardig aan het algoritme om het LeetCode probleem op te lossen. Gelukkig is een implementatie daarvan makkelijk te vinden. We starten van de laatste oplossing in het volgende artikel. De basis van dit algoritme blijft behouden, maar in ons probleem hebben we niet alleen het aantal 'cuts' nodig, maar ook de indices waar we moeten cutten. Gelukkig is deze oplossing al kwadratisch in tijd en geheugen (Door het berekenen van de isPalindrome array), dus deze aanpassing verslechterd de theoretische tijdscomplexiteit niet.

We maken nog een laatste aanpassing aan dit algoritme. Als 2 partities vergeleken worden om te bepalen welke de beste is, zal de partitie genomen worden met het minste aantal cuts. Als het aantal cuts echter gelijk is, nemen we de partitie die het minst aantal nieuwe labels toevoegd aan de boom. Hiermee verhogen we de theoretische tijdscomplexiteit van het algoritme naar `O(buffer_size^3 * LEN)`, omdat het bepalen van deze tiebreaker lineaire tijd neemt, maar dit verbeterd wel de compressiefactor.

(Code: `encode/minimum_cuts.c`)

Bepalen wat een geldige substring is

Nu willen we een functie die, gegeven een label en wat extra context, kan bepalen of het acceptabel zou zijn om het label aan onze boom toe te voegen. Dit probleem valt jammer genoeg niet optimaal op te lossen, we weten niet hoe vaak dit label nog in het vervolg zal terugkomen, en als dit label nooit in de boom toegevoegd is geweest, zullen we ook vergeten zijn hoe vaak we hem vroeger al hebben tegengekomen. De enige informatie die we hebben is dus de huidige buffer. Na een paar verschillende strategieën te hebben uitgeprobeerd, was het de volgende simpele heuristiek die de beste resultaten had: Een substring is geldig als hij meer dan 1 keer voorkomt in de buffer. We maken dus lokaal een gretige keuze, in het ergste geval komt de substring slechts 2 keer voor in de buffer en dan nooit meer in de rest van de text. Zelfs in dit geval is het toevoegen van de node niet zo slecht, omdat we onmiddelijk het label al kunnen hergebruiken in dezelfde buffer. Om de voorkomens bij te houden van alle substrings in de buffer hebben we wel een trie nodig. Dit verslechterd de tijdscomplexiteit van de twee algoritmes hierboven met een factor `O(LEN)` voor het opzoeken van de substrings in de trie.

Omdat we deze functie gaan moeten oproepen voor elke mogelijke substring van de buffer is het belangrijk dat hij snel uitgevoerd kan worden. Het zou dus te traag zijn als de functie elke keer dat hij een label krijgt de hele buffer zou moeten overlopen om het aantal voorkomens te tellen. Daarom bouwen we eerst een trie met het aantal voorkomens van elke substring. In dit geval is een trie beter dan bijvoorbeeld een hashmap, omdat de meeste substrings natuurlijk prefixen zullen zijn van andere substrings. We hoeven dus enkel de langste substrings toe te voegen, de kortere substrings zullen hierdoor van zelf in de trie terecht komen. We moeten dus buffer_size substrings toevoegen van lengte LEN, de kost hiervoor is dus `O(buffer_size * LEN)`. Daarna zal het slechts de lengte van een substring kosten voor de functie om te vinden hoe vaak hij voorkomt in de buffer (Dit is niet echt waar, omdat de trie linked lists gebruikt i.p.v. arrays, meer hierover later bij InternalTrie).

(Code: `encode/encode_strategy.c`)

Grootte van de buffer

De grootte van de buffer is zeer belangrijk, omdat de snelheid en kwaliteit van het bepalen van de optimale partitie hier sterk van afhangt. Theoretisch zou je verwachten dat het groter maken van de buffer de compressiefactor ook zou verbeteren. Met mijn implementatie merkte ik dat deze tendens klopte voor een buffergrootte <= 256, maar daarna werd de compressie slechter. Ik heb dus besloten om de maximale buffergrootte vast te leggen op 256. Dit heeft mij ook toegelaten om bij het bepalen van de optimale partitie op vele plaatsen met `unsigned char`'s te werken i.p.v. bijvoorbeeld `int`'s. Hierdoor wordt er meer geheugen over gehouden voor nieuwe nodes.

Tree

Voor de Tree zijn er 2 kritieke operaties die snel moeten kunnen uitgevoerd worden. De eerste hiervan is een functie die een label neemt en de node die bij dit label hoort teruggeeft (of NULL als de label niet in de boom is). De tweede is het vinden van de node met het grootste ordenummer die hetzelfde gewicht heeft als de 'adjust_node'.

Zoek label

Voor het zoeken van de passende node bij een label wordt ook een trie gebruikt. Bij het toevoegen van een nieuwe label wordt deze ook toegevoegd aan de trie. We gebruiken echter geen array, maar linked lists om de kinderen bij te houden. Zie voor meer informatie de documentatie in `tree/internal_trie.c`. Het gevolg hiervan is dat we in het slechste geval alle bladeren zullen moeten overlopen, maar in de praktijk is dit een stuk sneller.

Vind max ordenummer node

Hiervoor houden we een dynamische array bij die de nodes opslaat van grootste naar kleinste ordenummer. Hierdoor kunnen we als volgt te werk gaan om vanuit de originele node `t` de node met het grootste ordenummer (en hetzelfde gewicht) `t'` te vinden: Doordat de lijst gesorteerd is kun je met het ordenummer van `t` de exacte index van `t` in de lijst in constante tijd bepalen. Overloop de lijst dan achterwaards vanaf deze index tot je de laatste node tegenkomt die hetzelfde gewicht heeft als `t`, deze node is natuurlijk `t'`.

Hoe kunnen we deze lijst echter gesorteerd houden zonder bijvoorbeeld `qsort` na elke operatie te moeten uitvoeren? Er zijn 3 operaties die de lijst zullen aanpassen. Als eerste kunnen we een nieuwe label toevoegen. In dit geval kunnen we de nieuwe interne top en het nieuwe blad simpelweg achteraan de lijst toevoegen. De lijst is dan nog altijd gesorteerd, omdat deze twee nodes sowieso het laagste ordenummer hebben van alle nodes in de boom. De tweede operatie is het swappen van twee nodes in de boom. In dit geval swappen we ook deze twee nodes in de lijst, en onze lijst blijft nog altijd gesorteerd. De laatste operatie is het verhogen van het gewicht van een node. In dit geval moeten we helemaal niets doen, aangezien de node waarvan het gewicht verhoogd wordt sowieso de node is met het grootste ordenummer van alle nodes met hetzelfde gewicht (daar zorgt de swap methode namelijk voor). De node wordt dus de node met het kleinste ordenummer van de volgende gewichtsklasse. De lijst zal dus ook hierna op ordenummer (en gewicht) gesorteerd blijven.

De tijdscomplexiteit voor het zoeken van `t'` is `O(n)`. Je moet gemiddeld de helft van alle bladeren overlopen met hetzelfde gewicht als `t`. Als alle bladeren bijvoorbeeld gewicht 1 hebben, en je start van de node met het kleinste ordenummer van gewicht 1, zul je alle bladeren moeten aflopen. In de praktijk zal deze methode natuurlijk veel sneller zijn dan `O(n)`. De twee andere operaties (toevoegen, swappen) gebeuren wel in `O(1)`.

Een voordeel aan deze methode is dat je array altijd even groot is als het aantal nodes in je boom (-1, we voegen nng namelijk niet toe). Je hoeft bijvoorbeeld geen NULL pointers bij te houden voor gewichten die geen enkele node heeft. Als je dit wel zou doen zou je een algoritme kunnen maken dat dit probleem in constante tijd oplost, maar de array zou dan extreem veel geheugen nodig hebben voor, bijvoorbeeld, een file met 1 miljoen keer 'a'. De array heeft ook heel weinig overhead, voor elke node hoeven we slechts 1 pointer op te slaan.

(Code: `tree/node_array_list.c`)

Geheugen

Het meeste geheugen wordt bewaard voor het toevoegen van nodes in de boom. Daarnaast heeft het `minimum_cuts` algoritme `2 * buffer_size^2` `working_memory` nodig om de optimale partitie te bepalen in elke stap. We zorgen er ook voor dat de boom in 1 buffer partitie niet te veel nieuw geheugen zal besteden door een extra overschot aan geheugen vrij te houden. Dit is ongeveer gelijk aan het aantal bytes dat de boom in het 'ergste' geval zal besteden in de volgende partitie van de buffer, in de code noemt deze variabele `misc_memory`. Het totaal aantal bytes dat de boom dan mag gebruiken is `max_bytes - working_memory - misc_memory`.

Als de boom over deze grens gaat stoppen we volledig met nieuwe nodes toevoegen, behalve als dit echt moet (nieuwe 1 byte labels). De partitie van de buffer zal dus minder goed zijn, omdat er minder substrings als geldig beschouwd zullen worden (Enkel substrings van lengte 1 of substrings die al in de boom zitten zijn nog geldig). Hierdoor stoppen we wel echter (bijna) volledig met nieuw geheugen te besteden.

Om er voor te zorgen dat we de working en misc memory variabelen relatief niet te groot moeten nemen t.o.v. het gegeven geheugen, verlagen we de buffer size voor kleinere argumenten aan MEM. Ook dit zal de compressiefactor wat slechter maken.

Resultaten

(Zie `extra/results.md` voor de volledige test resultaten)

We testen de compressie op een paar files uit *The Canterbury Corpus*. Als baseline bekijken we eerst de resultaten voor `LEN = 1`, dit is dus gewone adaptive huffman, waar elke byte apart wordt gecodeerd. We zullen deze resultaten nu vergelijken met ons nieuw algoritme met verschillende waarden voor `MEM`. Omdat ons algoritme enkel substrings encodeerd die minimum 2 keer in de buffer voorkomen, en grotere substrings minder vaak minstens 2 keer voorkomen, zal het algoritme conservatief omgaan met het toevoegen van grotere substrings. Er is minder kans dat we die later nog eens kunnen gebruiken, dus dat is exact wat we willen. Uit experimentele resultaten blijkt dus dat hoe groter het `LEN` argument, hoe beter de compressiefactor. Voor de testen nemen we dus telkens `LEN = 9`.

Uit de resultaten blijkt dat dit algoritme beter presteert voor grotere bestanden. Dat is natuurlijk logisch, hoe groter het bestand, hoe meer kans dat we een label > 1 kunnen hergebruiken. Vooral voor `bible.txt` verkrijgen we heel goede resultaten. Voor `MEM >= 5` krijgen we een compressiefactor van ~3, we gaan dus van 8 bits per byte naar gemiddeld ~2.68 bits per byte. Dat is goed vergelijkbaar met de resultaten van sommige algoritmen voor dezelfde file. (eerste kolom: bits per byte, tweede: compressietijd, derde: decompressietijd). In het bijzonder presteert dit algoritme iets beter op deze file dan het unix commando `compress` qua compressiefactor (maar het comprimeren gaat wel veel trager).

We zien ook interessante resultaten voor `alphabet.txt`. Die file is gewoon alle 26 karakters van het alphabet in volgorde, herhaald tot de file 100 KB groot is. De resultaten voor gewone adaptive huffman zijn natuurlijk niet zo goed, maar voor ons algoritme met kleine `MEM` argumenten ook niet. Voor de kleine `MEM` argumenten nemen we een `buffer_size < 2 * 26`, waardoor ons algoritme ook niet door heeft dat het alphabet herhaald wordt, de resultaten zijn dus bijna even slecht als normaal adaptive huffman. Vanaf `MEM = 3` is de buffer size echter wel groot genoeg om heel het alphabet twee keer te hebben gezien, daardoor worden opeens heel lange substrings gecodeerd en verlaagd de grootte van de gecomprimeerde file opeens van 56 KB naar 8 KB.

Voor kleinere files zien we in het algemeen soms dat meer geheugen kan leiden tot een slechtere compressie. Door de heuristische manier waarop we substrings encoderen kan het zijn dat we 'geen geluk' hebben, en een label encoderen dat we later toch nooit meer gebruiken. Bij grotere files valt dit fenomeen wel weg.

We bekijken nu de (de)compressietijd resultaten voor `bible.txt` met verschillende waardes voor `MEM` en `LEN` (zie bible_compression_time.txt). Zoals verwacht stijgt de compressietijd voor grotere waardes van `MEM` en `LEN`. De decompressietijd ligt ongeveer gelijk in alle gevallen (ongeveer 1 seconde voor uitschrijven naar /dev/null).

Het geheugenverbruik voor de compressie en decompressie is bijna hetzelfde, omdat ze allebei dezelfde boom opbouwen. Het enige verschil is dat de decoder geen

working memory nodig heeft.

Optimale parameters voor

Korte chatberichten

Voor korte texten is de tijd/geheugen kost van grotere `MEM` en `LEN` argumenten de moeite waarschijnlijk niet waard. We nemen dus bijvoorbeeld `LEN = 1` en `MEM = 0`, om te optimaliseren op snelheid en geheugen, aangezien we toch geen goede compressiefactor kunnen krijgen.

Alle tweets van een bekende politicus

Dit zal al een redelijk grote tekst zijn, er is dus een goede kans dat we labels > 1 meerdere keren zullen hergebruiken. Neem dus `LEN = 9` en `MEM = 9` voor een optimale compressiefactor.

De broncode van het Linux-besturingssysteem

Dit zal een nog grotere file zijn, neem dus voor dezelfde reden als in het vorige voorbeeld `LEN = 9` en `MEM = 9`.

Random data

Als deze random data bestaat uit alle mogelijke bytes, encodeer je hem best helemaal niet. De file zal waarschijnlijk groter zijn na compressie. Als enkel bepaalde bytes gebruikt worden (zoals bij `random.txt` in de testen), is het best om het te encoderen met normale adaptive huffman (`LEN = 1` en `MEM = 0` dus). Doordat de data random is, is er namelijk een zeer lage kans dat een label > 1 dat meerdere keren voorkomt in een buffer nog eens zal voorkomen buiten die buffer. Daarom is het niet optimaal om dit algoritme op random data toe te passen.