FACULTEIT
WETENSCHAPPEN

Algoritmen & Datastructuren 3

Lineaire compressie met LZ77 en deelstring bomen

Nathaniel Joos

Academiejaar: 2018-2019

UNIVERSITEIT
GENT

Inhoudsopgave

1 Inleiding

Het project van Algoritmen & Datastructuren 3 gaat over het implementeren van een lineair algoritme voor LZ77 door gebruik te maken van het algoritme van Ukkonen voor deelstring bomen. Het project is opgedeeld in twee delen. Enerzijds is er de implementatie van het algoritme van Ukkonen en anderzijds is er de implementatie van LZ77 dat gebruik zal maken van het eerste deel. Aangezien beide algoritmen uitgebreid uitgelegd werden in de cursus, wordt er meer aandacht besteed aan de uniciteit van mijn versie.

Opmerking: het bestand *time.ipynb* (een jupyter notebook bestand dat gebruik maakt van SageMath) in de folder *extra/* bevat alle tijdsmetingen alsook de code voor het maken van de grafieken.

2 Ukkonen

2.1 Implementatiekeuzes

Alle code omtrent Ukkonen is gebaseerd op een *stackoverflow* post [1], alle gebruikte namen voor de variabelen zijn dus sterk gebaseerd op de uitleg hiervan.

2.1.1 De 4 regels

In de code is er sprake van vier soorten regels, deze zijn gebaseerd op de drie regels die besproken werden in de post. Deze zullen hier besproken worden.

Regel 1: Indien er een boog gesplitst wordt vanuit de wortel, dan

- `active_node` blijft de wortel
- `active_edge` wordt gekoppeld aan de volgende suffix die toegevoegd moet worden
- `active_length` wordt gedeïncrementeerd

Regel 2: Indien er een boog gesplitst wordt en een nieuwe top aangemaakt wordt en dat is niet de eerste top die aangemaakt is tijdens de huidige iteratie, dan koppelen we de vorige aangemaakte top met de huidige aangemaakte top via een *suffix link*.

Regel 3: Indien er een boog gesplitst wordt vanuit een top dat niet de wortel is dan volgen we zijn suffix link, m.a.w. zetten we `active_node` gelijk aan de suffix link. Heeft deze top geen suffix link dan passen we regel vier toe.

Regel 4: Indien er een boog gesplitst wordt vanuit een top dat niet de wortel is en deze heeft geen suffix link, dan

- `active_node` wordt de wortel
- `active_edge` wordt gekoppeld aan de volgende suffix die toegevoegd moet worden
- `active_length` wordt gelijk gesteld aan het aantal toe te voegen suffixen - 1

2.1.2 De functie `traverse_down`

De functie `traverse_down` is een functie die zal kijken of de `active_length` groter of gelijk is dan de lengte van de huidige boog. Indien dit het geval is zal deze naar het einde van de boog gaan en bijgevolg alle waardes corrigeren. Dit wordt herhaald tot de `active_length` kleiner is dan de huidige boog. Deze functie wordt elke keer opgeroepen indien de `active_length` niet gelijk is aan nul omdat nergens in de code het einde van een boog gecontroleerd wordt.

2.1.3 Begin van een iteratie

Mijn versie van het algoritme van Ukkonen gaat ervan uit dat als een iteratie begint en de `active_length` gelijk is aan nul, dit equivalent is met het toevoegen van een karakter vanuit de wortel. Nu is dit niet altijd waar want dit kan bijvoorbeeld ook zijn indien we aan het einde van een boog zitten. Dit kan echter nooit gebeuren in mijn implementatie door de `traverse_down` functie die elke keer opgeroepen wordt in het *else-blok*. Indien na het oproepen de `active_length` nul is, dan wordt dit apart bekeken vanwege **Regel 4.**

2.2 Testen

Allereerst moet vermeld worden dat alle testen ervan uitgaan dat elke tekst eindigt op een uniek teken, anders gelden niet noodzakelijk alle komende eigenschappen. Anderzijds zijn alle testen geschreven in pure C code.

2.2.1 Primaire testen

De test is een executable dat bestanden verwacht als parameters. Deze zal al deze bestanden een voor een testen op volgende eigenschappen:

1. Of het aantal uitgaande bogen van de wortel gelijk is aan het aantal unieke karakters in de tekst
2. Of het aantal bladeren in de boom gelijk is aan de lengte van de tekst
3. Of alle mogelijk suffixen in de boom zitten

2.2.2 Tijdsmetingen en lineariteitstest

Om te testen op lineariteit heb ik 9 bestanden aangemaakt, beginnend bij 1 miljoen tot 9 miljoen karakters per bestand. Vervolgens gebruik ik het UNIX commando `time` om te meten hoe lang het algoritme duurt op een tekstbestand. Dit commando voer ik drie keer uit op elk bestand en neem dan het gemiddelde van de drie waardes.

```
$ time ./ukkonen < random_X.txt > random_X.out
```

Vervolgens de waargenomen tijdsmetingen:

- 1.000.000 karakters: 3.563s
- 2.000.000 karakters: 6.897s
- 3.000.000 karakters: 10.269s
- 4.000.000 karakters: 14.135s
- 5.000.000 karakters: 18.088s
- 6.000.000 karakters: 22.204s
- 7.000.000 karakters: 27.683s
- 8.000.000 karakters: 33.603s
- 9.000.000 karakters: 38.106s

Figuur 1 toont de lineariteit van het algoritme aan.

3 LZ77

3.1 Implementatiekeuzes

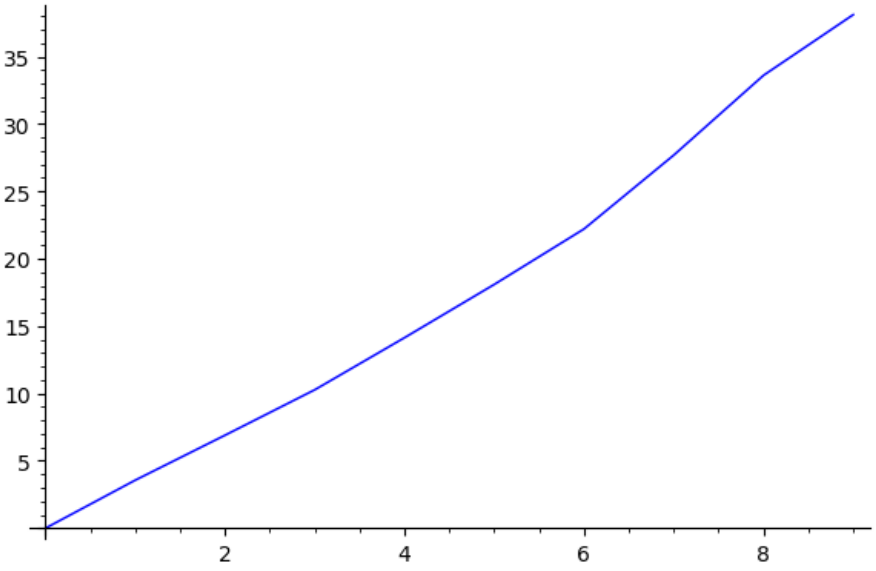
3.1.1 Berekenen van $beg(t)$

De code van het eerste deel is een klein beetje aangepast om de $beg(t)$ waardes te berekenen per top. De berekening hiervan was redelijk simpel in functie van $remainder$ en de huidige index. Voor het splitsen van een boog moet de nieuwe top dezelfde $beg(t)$ waarde krijgen als de vorige top.

3.1.2 Match state

De struct `match_state` wordt gebruikt om de langste match te vinden. In het begin staat het `state` veldje altijd op 0, dit wil zeggen dat er zogezegd een match is. Wanneer er een match is zal het algoritme geen output veroorzaken maar blijven verder werken tot er geen match gevonden is. Dus in het begin zal er nooit een match gevonden worden met het eerste karakters omdat de deelstring boom nog leeg is. In dit geval hebben we geen match, dus we printen de drie opgebouwde waardes uit naar stdout. Deze struct houdt ook bij waar we op dit moment zitten in de deelstring boom om te weten wat de volgende karakter zal zijn. In het geval we een match vinden, dus het geval waar het volgende karakter in de boom gelijk is aan het huidige karakter dat we wensen te comprimeren, zullen de nodige waardes aangepast worden. Het `state` veldje zal wel nog altijd op 0 staan.

Indien we aan het einde van het bestand zitten dan maakt het niet uit of we een match hebben of niet, we voegen het *null*-karakter toe zodat we altijd een match hebben en printen die dan uit.

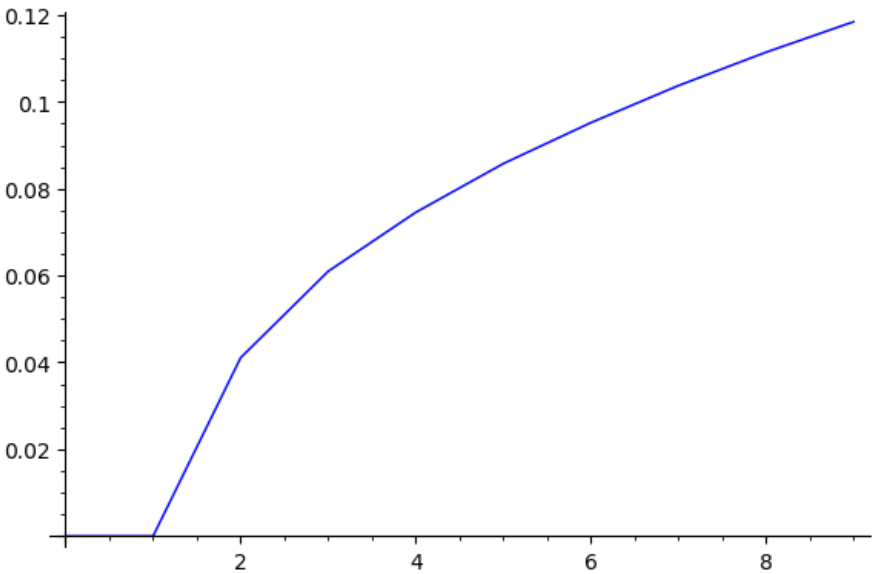


Figuur 1: Uitvoeringstijd in functie van aantal karakters

3.1.3 Geheugen optimalisaties

Om het geheugengebruik te optimaliseren heb ik ervoor gekozen om een nieuwe deelstring boom te starten na 10.000.000 karakters. Indien een bestand meegegeven wordt dat meer dan 10.000.000 karakters bevat zonder de −o vlag, zal er een bericht verzonden worden naar stderr dat de gebruiker verplicht om het programma te gebruiken met de −o vlag.

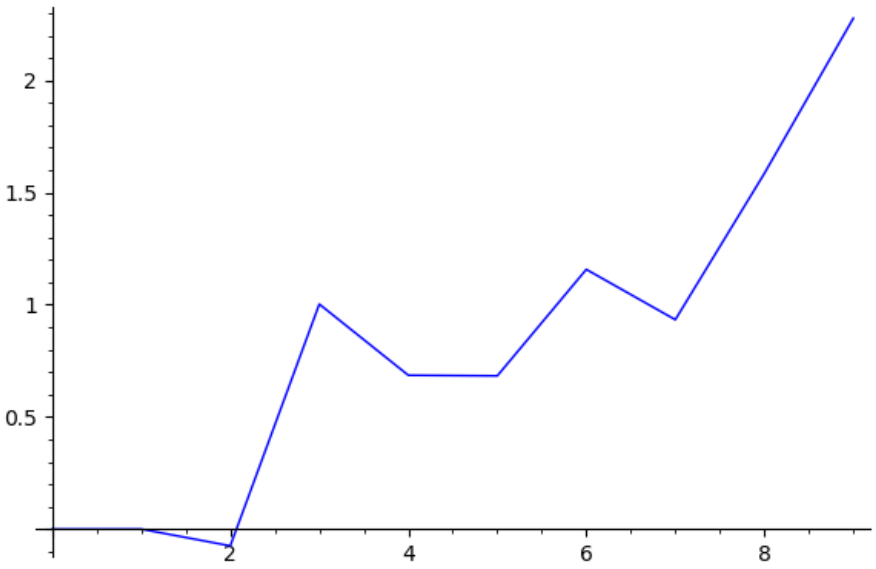
Het effect van het starten van een nieuwe boom na x aantal karakters is natuurlijk dat de compressie niet optimaal is. Echter blijkt het verlies niet zo heel erg te zijn als men x goed kiest. Figuur 2 is het resultaat van het verlies tussen het comprimeren met nieuwe bomen en een grote boom waarbij we om de 1.000.000 karakters een nieuwe boom starten. Het valt direct op dat het verlies logaritmisch stijgt in functie van het aantal karakters in de volledige tekst. Een ander opvallend effect is dat het gecomprimeerd bestand van 2.000.000 karakters maar 4% keer groter is met optimalisaties dan zonder. Een goede keuze voor x is dus eigenlijk de helft van de lengte van de tekst. Hierbij gebruik je maar half zoveel werkgeheugen ten koste van 4% meer bestandsgrootte.



Figuur 2: Percentage verlies in functie van aantal karakters (in miljoen)

Een andere opmerking is een verbetering in uitvoeringstijd bij het optimaliseren. Op figuur 3 zie je de tijdswinst in functie van het aantal karakters. Dit is heel simpel te verklaren doordat je steeds werkt met een nieuwe boom. Het vrij maken van het gealloceerde geheugen is trager bij grotere bomen dan bij kleinere bomen, dit komt doordat je

voor elke interne top 128 bogen moet vrijmaken. Een kleinere boom heeft bijgevolg minder interne toppen en zal dus een betere snelheid hebben om vrij te maken. Uitdrukkelijk wil dit zeggen dat het vrijmaken van een aantal kleinere bomen sneller is dan het vrijmaken van een grote boom.



Figuur 3: Aantal seconden winst in uitvoeringstijd in functie van het aantal karakters

3.1.4 Het bland karakter

Het bland karakter wordt simpelweg gebruikt om aan te tonen bij het decomprimeren dat een nieuwe boom werd gebruikt op dit punt. Het decomprimeren staat dus volledig los van de gekozen max_size

3.2 Testen

3.2.1 Primaire test

De test is net als bij Ukkonen een executable dat bestanden verwacht als parameters. Het testen is daarentegen heel simpel hier. We voeren compressie toe op een bestand en achteraf decompressie en vergelijken beide bestanden met het UNIX commando `diff`. Hetzelfde wordt herhaald voor de geoptimaliseerde versie.

3.2.2 Tijdsmetingen en lineariteitstest

Gelijkaardig als bij Ukkonen zal ik ook hier werken met de 9 bestanden met willekeurige tekens.

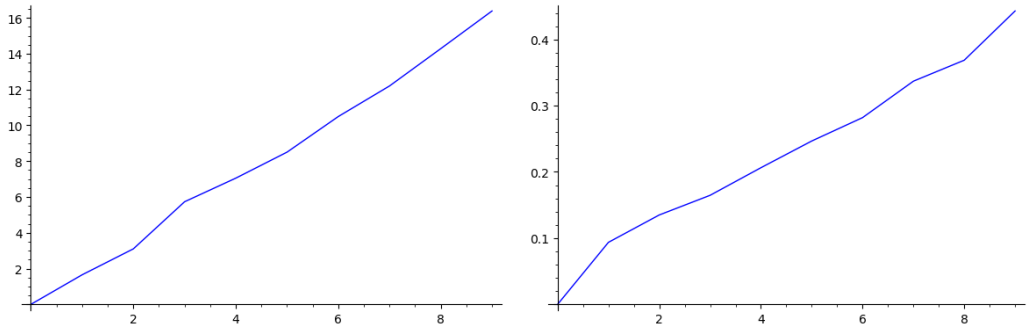
```
$ time ./lz77 -c < random_X.txt > random_X.bin
$ time ./lz77 -d < random_X.bin > random_X2.txt
```

Vervolgens de waargenomen tijdsmetingen voor het comprimeren respectievelijk decomprimeren:

- 1.000.000 karakters: 1.652s
- 2.000.000 karakters: 3.101s
- 3.000.000 karakters: 5.732s
- 4.000.000 karakters: 7.051s
- 5.000.000 karakters: 8.504s
- 6.000.000 karakters: 10.492s
- 7.000.000 karakters: 12.200s
- 8.000.000 karakters: 14.285s
- 9.000.000 karakters: 16.388s

- 1.000.000 karakters: 0.094s
- 2.000.000 karakters: 0.135s
- 3.000.000 karakters: 0.164s
- 4.000.000 karakters: 0.206s
- 5.000.000 karakters: 0.247s
- 6.000.000 karakters: 0.282s
- 7.000.000 karakters: 0.337s
- 8.000.000 karakters: 0.369s
- 9.000.000 karakters: 0.443s

Figuur 4 toont de lineariteit van zowel het comprimeren als decomprimeren aan.



Figuur 4: Uitvoeringstijd van compressie en decompressie in functie van aantal karakters

3.3 Opmerking

Alle gecomprimeerde bestanden zijn meestal niet zo veel kleiner dan hun originele, soms zijn ze zelfs veel groter dan hun originele. Dit heeft te maken met kleine herhalingen, te hoge willekeurigheid en ook met het aantal bits een triple in beslag neemt. Als we kijken naar het grootste getal je kan voorstellen met 32 bits ($2^{32} = 4294967296$) dan moet je al een heel grote herhaling hebben om hier winst uit te krijgen. Als we echter maar 16 bits nemen, dan heb je nog altijd een groot bereik voor de lengte van een match ($2^{16} = 65536$). De kans dat je in een normale tekst een herhaling tegenkomt dat groter is dan 65536 is niet bepaald groot. Het effect is hierbij dat je veel minder bits gebruikt om een triple te beschrijven en bijgevolg dus betere compressie krijgt.

De grootte van de triple heeft dus een groot effect op compressie. Als je veilig wilt spelen kies je deze groot, maar in de praktijk heeft een groot bereik weinig effect.

Referenties

[1] Ukkonen's suffix tree algorithm in plain English *https://stackoverflow.com/questions/9452701/ukkonens-suffix-tree-algorithm-in-plain-english/9513423#9513423*. 2