AD3 Verslag

Sorteren van geëncodeerde peptiden

Ben De Meurichy
13/12/2023

1 Inleiding

In dit project heb ik een algoritme geïmplementeerd om de orde bewarende prefix-codering van een aantal peptidekettingen te bepalen. Deze prefix-codes kunnen dan gebruikt worden om compressie toe te passen. Aangezien de codes orde bewarend moeten zijn is het ook logisch om de geëncodeerde bestanden te sorteren.
Ik vermoed dat de geëncodeerde bestanden sneller gesorteerd zullen worden omdat hier in totaal minder bits aanwezig zijn dus er moeten minder vergelijkingen doorgevoerd worden.
Of dit wel degelijk het geval is zullen de benchmarks moeten aangeven.

2 Ontwerpbeslissingen

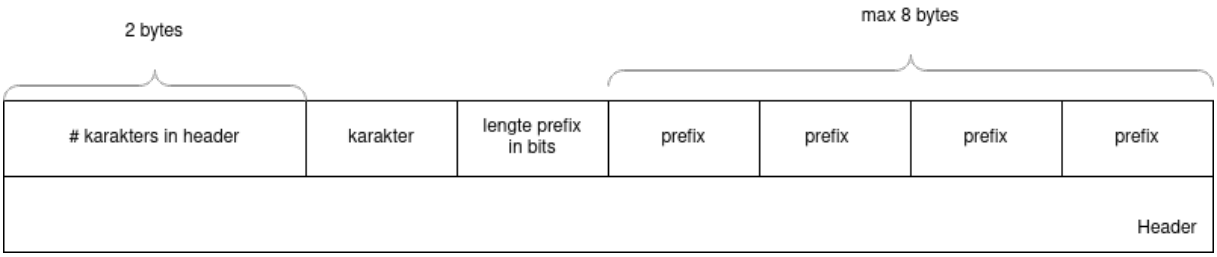
2.1 Data gebruik

In het project is gekozen om een vaste hoeveelheid data te gebruiken voor de volgende delen.

De header:
Elke prefix-code wordt van rechts naar links in de nodige hoeveelheid bytes weggeschreven. Hierdoor worden er nullen achteraan de code geplakt en hebben we de lengte van de prefix nodig die voor de prefix wordt opgeslagen. Als dit niet wordt gedaan geraken we info kwijt en kunnen sommige prefix-codes dubbel voorkomen.

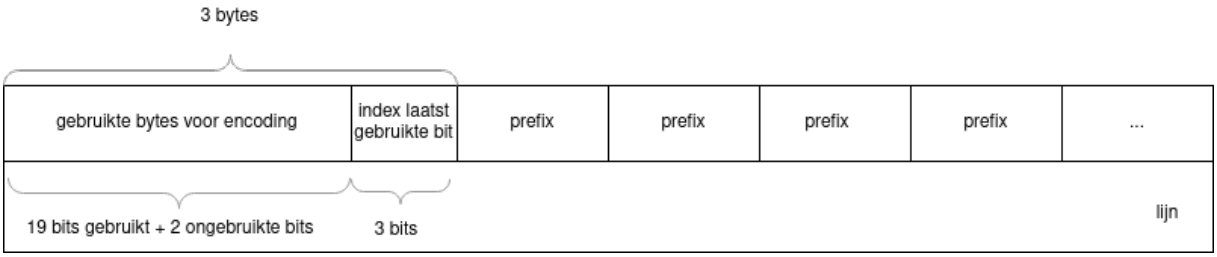
data	gebruikte hoeveelheid bytes
aantal karakters in header	2
karakter	1
prefixlengte in bits	1
prefix	variabel (berekend met prefixlengte): max 8

Tabel 1: gebruikte data header



Figuur 1: Gebruikte formaat header

De geëncodeerde lijnen:
Bij het begin van het project konden er maximaal $(45354\ karakters) * (8\ bytes\ per\ prefix) = 362832$ bytes per lijn voorkomen, hiervoor hebben is dus $\log_2(362832) = 19$ bits nodig om de lengte voor te stellen in bytes. Na de verandering van de opgave waren er maar 16 bits nodig om de maximale lijnlengte op te slaan.
Dit doet er eigenlijk niet toe omdat we nog 3 bits nodig hebben om de index van de laatste bit voor te stellen van de lijn.
Er worden dus voor elke lijn 3 bytes opgeslagen ongeacht de versie van de opgave.
Doordat alle prefix-codes aan elkaar worden gekleefd krijgen we dus in de laatste gebruikte byte van de lijn extra nullen die we moeten negeren. Wanneer we alle prefix-codes een voor een wegschrijven, treedt er logischerwijs geen compressie op, en resulteren de bestanden in grotere formaten vanwege de toegevoegde header.



Figuur 2: Gebruikte formaat geëncodeerde lijnen

2.2 Verdere implementatiedetails

Voor het grootste stuk is het wel een standaard implementatie van de opgave.
Er is voor gekozen om de newline geen code te geven want deze informatie houden we bij via de lengte van de lijn. In de rest van het programma zijn er geen eigenaardigheden tegengekomen.

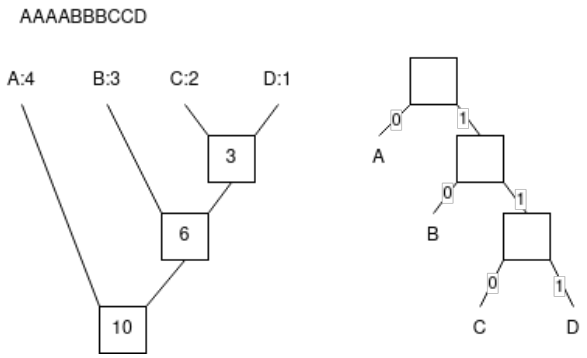
In de implementatie van het extern sorteren zit nog een bug waardoor er vanaf het mergen een aantal lijnen verkeerd geplaatst worden. Deze functie doorloopt wel alle nodige stappen.
Ik geloof dat deze bug komt omdat alle pointers naar de laatst gelezen lijn in een lijst wordt bijgehouden en de volgende lijn wordt gezocht via de index, als er een van de tijdelijke bestanden dus eerder uitgelezen is dan de rest komen hier lege pointers te staan.
De volgende lijn om uit te lezen wordt wel gezocht via de gelinkte lijst waar deze gaten dan niet in zitten.
Een oplossing hiervoor zou zijn om de lijnen bij te houden in de gelinkte lijst met bestand pointers, dan zouden uitgelezen bestanden zichzelf verwijderen en zouden de blokken normaal gezien wel juist gemerged worden.
Dit heb ik jammer genoeg door tijdgebrek niet werkende gekregen zonder segmentationfaults dus hier is niets van te vinden op de subgit indiening.
Het idee achter het sorteren is hetzelfde als in de theorie. Eerst wordt de header weggeschreven naar het output bestand, deze hebben we intact nodig bij de extractie van het bestand. Hierna wordt de basisstap uitgevoerd waar het bestand wordt opgedeeld in blokken van een aantal lijnen. De tijdelijke bestanden die hierdoor worden gecreëerd worden dan gemerged tot er 1 bestand over blijft dat dan volledig wordt weggeschreven naar de output.

De extractie functie is als volgt geïmplementeerd.
Als eerste stap lezen we heel de header uit. Deze gebruiken om de orde bewarende prefix-boom terug op te bouwen. Een 0 in de prefix-code is een tak naar links en een 1 is een tak naar rechts.
Deze kunnen we dan gebruiken om op de normale Huffman manier te decoderen.
We weten waar we de newlines terug moeten plaatsen doordat dit wordt bijgehouden in de 3 bytes voor elke lijn. Het bestand kan zo lijn per lijn gedecodeerd worden.

3 Orde bewarende prefix-codering

3.1 Theorie OPC

Voor sommige teksten kunnen we een Huffman encodering maken die ook orde-bewarend is.
Dit is het gemakkelijkst met teksten waar alle karakters dezelfde frequentie hebben of de frequenties overeen komen met de volgorde van de karakters.
bv:"AAAABBBCCD"= 4 3 2 1 kan makkelijk omgezet worden in een orde bewarende prefix boom Een voorbeeld waarvoor een orde bewarende Huffman prefix-code wel bestaat:"AAAABBBCCD".

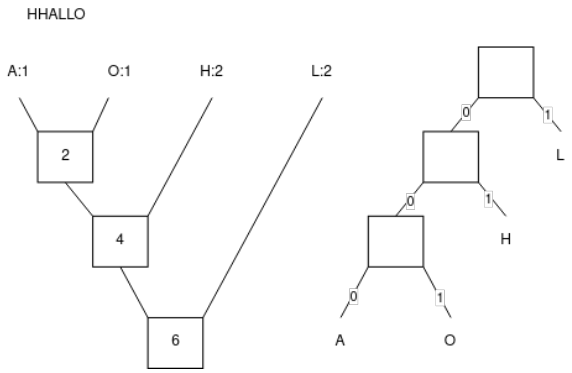


Figuur 3: Huffman encodering "AAAABBBCCD"

wat de volgende prefix-codes geeft:

A:0 C:110
B:10 D:111

Voor de meeste natuurlijke teksten (ook de peptiden in dit project) is dit echter niet mogelijk, we kunnen ook hier makkelijk een voorbeeld vinden waar dit niet kan. Een voorbeeld waarvoor een orde bewarende Huffman prefix-code niet bestaat:"HHALLO".



Figuur 4: Huffman encodering "HHALLO"

Die de volgende codes oplevert:

A:000 H:01
O:001 L:1

Het is duidelijk dat als we eisen dat de prefix-codes altijd orde bewarend moeten zijn maar niet eisen dat de prefix-codes zelf optimaal zijn geen Huffman kunnen gebruiken. Hiervoor kunnen wel verschillende algoritmes gevonden worden die de optimale orde bewarende prefix-boom teruggeven.

3.2 Implementatie OPC

Om de OPC op te stellen heb ik gebruik gemaakt van het algoritme van Hu-Tucker[1].
Deze heeft iets meer ingewikkelde logica dan Garcia-Wachs, maar als we alle stappen een voor een uitwerken vormt dit geen probleem. Dit algoritme bestaat uit 3 grote stappen:

1. Met een lijst van de gesorteerde karakters gematched met hun frequentie in de tekst beginnen we aan de eerste stap.
   Hiervan moeten we een optimale binaire zoekboom maken waarbij deze elementen de bladeren zijn van de boom. Volgens de logica van het algoritme zijn er tijdens het opbouwen van de boom 2 soorten toppen, originele en nieuwe toppen. Originele toppen zijn de toppen die vanaf het begin in de lijst zitten, nieuwe toppen zijn toppen die gemaakt zijn door het samenvoegen van andere toppen. De boom wordt opgebouwd door 2 compatibele toppen samen te nemen met de laagste frequenties. Toppen zijn compatibel wanneer er geen originele toppen tussen zitten in de lijst. Met deze toppen wordt een nieuwe top gemaakt die als frequentie de som heeft van de 2 kindertoppen. Dit herhalen we tot er nog 1 top in de lijst over blijft.

2. Met deze boom kunnen we de diepte van de bladeren bepalen aan de hand van een depht-first-search. Hierna hebben we deze opgestelde boom niet meer nodig, enkel nog de bladeren. Om de volgende stap gemakkelijker te maken houden we ook de grootste diepte gevonden bij.

3. Als laatste stap kunnen we de orde bewarende prefix-boom opstellen met behulp van de gevonden diepte uit de vorige stap. Deze stap begint terug bij de alfabetisch gesorteerde lijst. Hier houdt elke top de diepte die deze moet krijgen in het resultaat bij. Het opbouwen van de boom gebeurt door vanaf de grootste diepte alle toppen op dit niveau per 2 samen te voegen. De samengevoegde top krijgt $diepte - 1$ als frequentie en zo wordt dit herhaald tot we terug een volledige boom hebben en op diepte 0 zitten.
   Voor elk niveau zal er een even aantal toppen zijn door de manier waarop we de boom hebben opgebouwd in stap1. Alle nieuwe toppen worden op dezelfde plaats teruggeplaatst in de lijst als waar het linker kind van de top zat voor het samenvoegen. Zo wordt de alfabetische volgorde gegarandeerd en wordt er uiteindelijk een orde bewarende prefix-boom bekomen.

   Als we op deze implementatie een kleine tijdscomplexiteitsanalyse uitvoeren kan er al gauw opgemerkt worden dat stap 2 en 3 vrij efficiënt kunnen gemaakt worden.

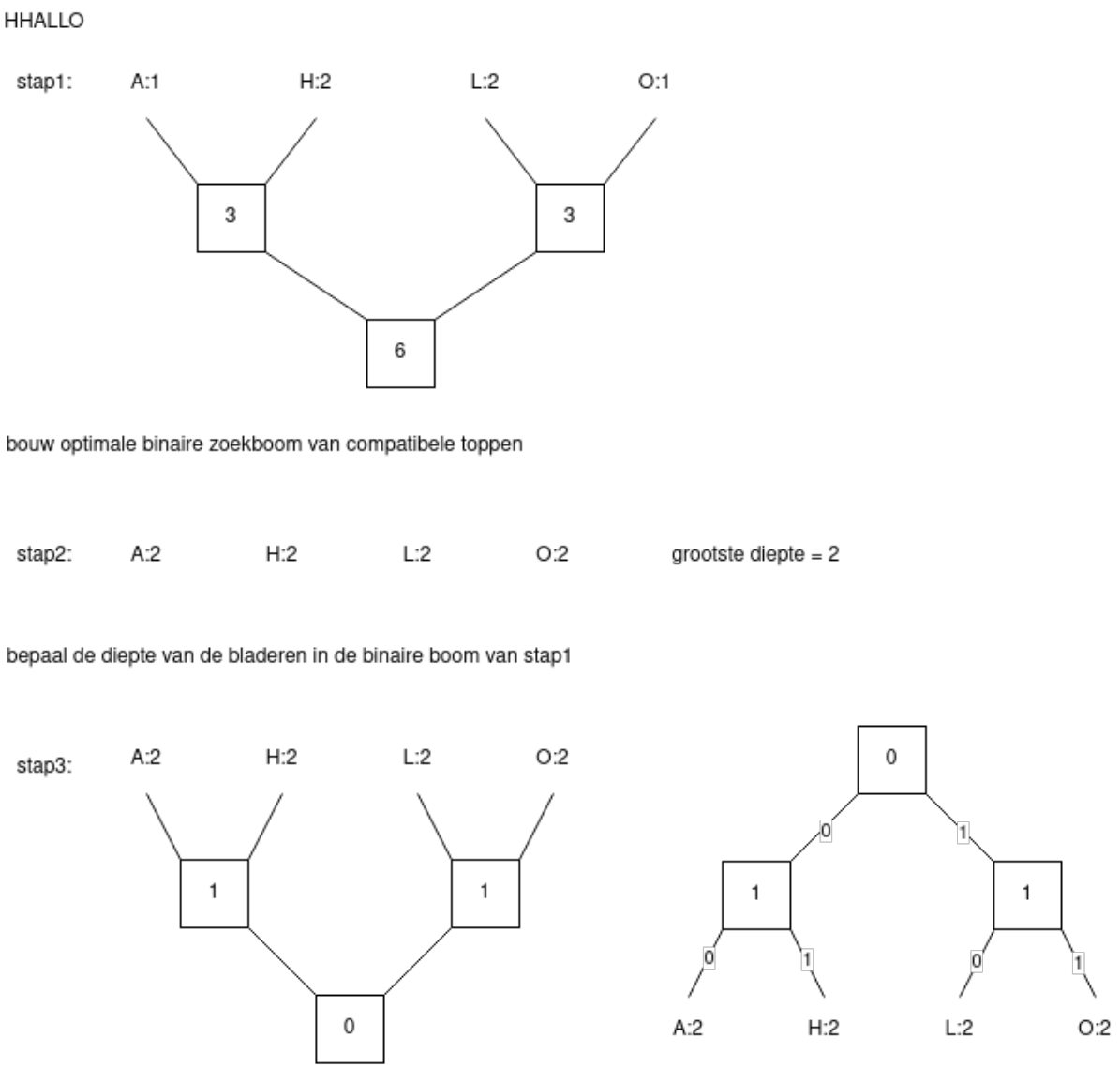
   In stap 2 moet een depth first search uitgevoerd worden, hiervoor moet elke top afgelopen worden, dit kan met $O(n)$. Doordat een binaire boom met n bladeren $2 * n - 1$ toppen heeft is dit $O(n)$ met n het aantal verschillende karakters in de tekst.

   Voor de implementatie van stap 3 heb ik voor elk niveau de lijst doorzocht naar toppen op hetzelfde niveau dus dit zal $O(n \log(n))$ geven. Dit komt doordat een binaire boom met n bladeren en waar alle interne toppen 2 kinderen hebben is de maximale diepte $\log_2(n) + 1$ met n het aantal bladeren. Omdat het aantal toppen op een niveau altijd even is kan je met 1 lus over de lijst het hele niveau opbouwen.

   Voor de eerste stap heb ik op de gemakkelijke manier gezocht naar de 2 compatibele toppen met laagste frequentie. Er is gebruik gemaakt een while lus totdat er nog maar 1 top over blijft in de lijst, dit zijn al n stappen. Binnen in deze lijst wordt er gezocht naar de top met de laagste frequentie, dit is ten hoogste n stappen. Hierna wordt links en rechts van deze top gezocht naar een compatibele top met de kleinste frequentie. Dit is ook ten hoogste n stappen maar gebeurt na het zoeken van de eerste top, dus deze worden opgetelt naar max $2n$. Samen met de buitenste lus voor het opbouwen van de hele boom wordt dit $n * 2n$ wat dus als bovengrens $O(n^2)$ geeft.
   De algemene bovengrens voor de implementatie van dit algoritme is deze 3 stappen opgeteld wat samen komt tot $O(n^2)$.

   In de eerder vermelde paper over Hu-Tucker wordt er nog een andere manier besproken om stap 1 uit te voeren. Deze zou aan de hand van min-heaps het algoritme versnellen tot $O(n \log(n))$.

[1] Hu-Tucker



Figuur 5: voorbeeld Hu-Tucker op string: "HHALLO"

4 Bespreking benchmarks

We zien dat vanaf er betekenisvolle compressie optreed de implementatie die sorteert met compressie er minder lang over doet om het bestand te verwerken dan de sorteer implementatie van bash die de originele bestanden neemt om te sorteren. Over het algemeen is het sorteren met compressie gemiddeld 40% sneller dan die zonder. Dit is logisch omdat we hierbij gewoon minder bytes vergelijken over het algemeen.

De uitvoeringstijd van de andere functies is niet opmerkelijk goed of uitermate slecht. Wat wel opvalt is dat de compressiefunctie een pak trager is dan de extractiefunctie. Deze functies doen op zich hetzelfde maar in de tegengestelde richting dus dit verschil was onverwacht. Dit valt te verklaren doordat de compressie een veel grotere input krijgt dan de extractie en dus over meer bytes zal moeten lezen. IO-operaties kosten veel tijd dus dit veroorzaakt waarschijnlijk de vertraging.

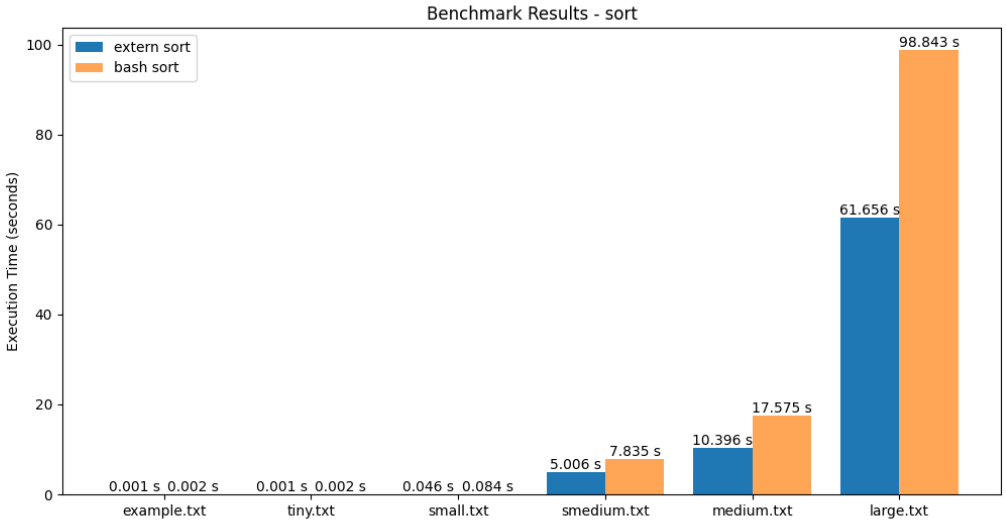
Qua compressie is er telkens ongeveer een compressieratio van $\pm 1,5$ te zien. Dit is vrij goed, zo kunnen we een goede hoeveelheid ruimte besparen op de harde schijf.

bestandsnaam	example.txt	tiny.txt	small.txt	smedium.txt	medium.txt	large.txt
voor compressie	36 B	19,7 kB	2,0 MB	99,2 MB	198,4 MB	991,6 MB
na compressie	62 B	13,3 kB	1,3 MB	67,8 MB	136,6 MB	682,8 MB
compressie ratio	0,581	1,481	1,538	1,463	1,452	1,452

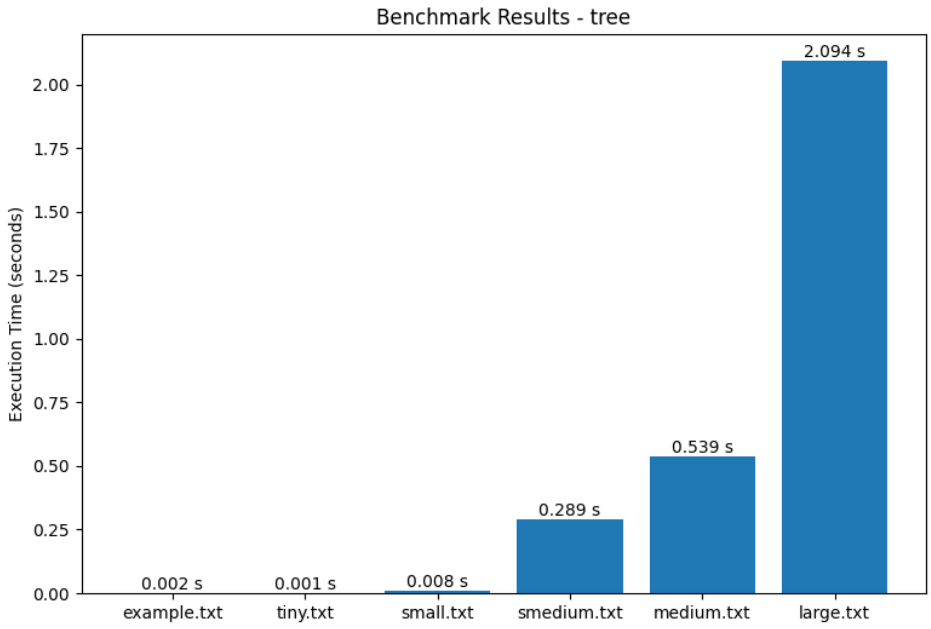
5 Conclusie

Als antwoord op de vraag of het sorteren van data volgens OPC een goed idee is, zien we duidelijk dat dit voordelen heeft. Als de overhead om compressie uit te voeren en extractie gepermiteerd zijn is het zeer makkelijk om dit aan te raden.
Aangezien de compressie op zich al een voordeel geeft door minder schijfruimte in beslag te nemen is het zeker nuttig om dit te doen met OPC. Dan kunnen we tijd een redelijke hoeveelheid tijd uitsparen bij het sorteren.

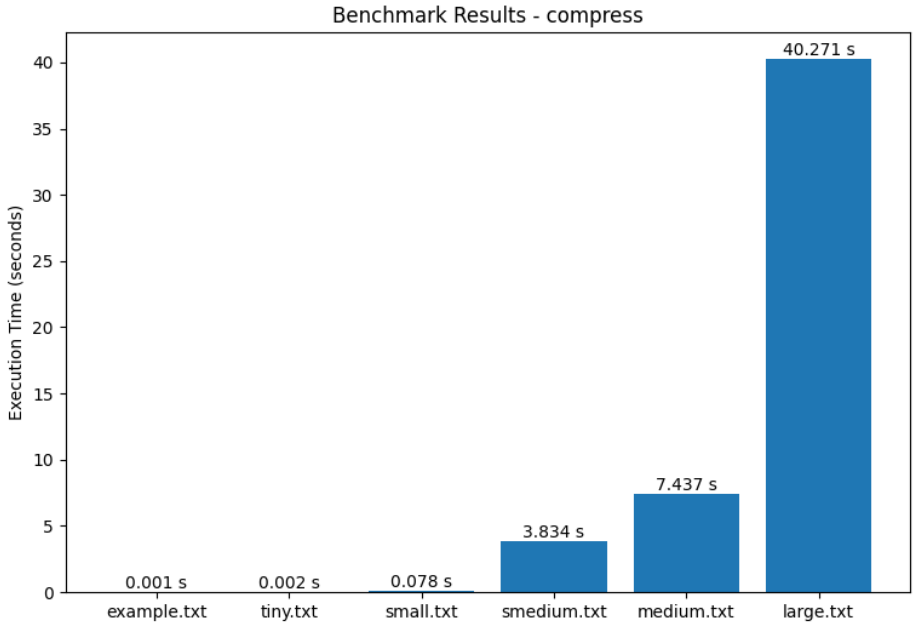
6 Appendix: benchmarks



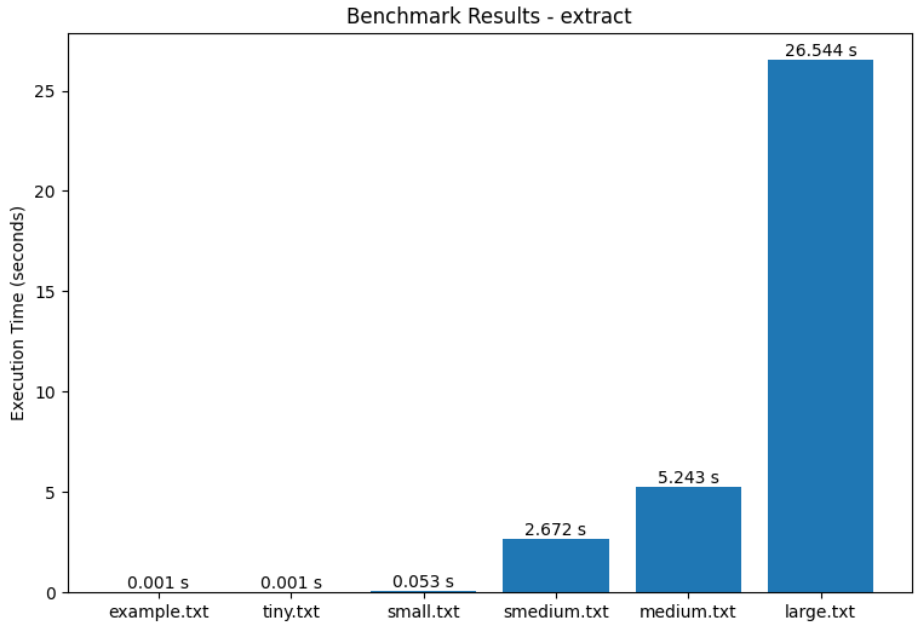
Figuur 6: Vergelijking eigen sortering (geëncodeerd) en bash sortering (niet geëncodeerd)



Figuur 7: Uitvoeringstijd opbouwen OPC



Figuur 8: Uitvoeringstijd compressie



Figuur 9: Uitvoeringstijd extractie