Algoritmen voor Strings

Project Algoritmen en Datastructuren III

Academiejaar 2010-2011

1.Implementatie

Eerst en vooral dien ik te vermelden dat de code voor het inlezen van bestanden telkens op dezelfde manier gebeurt. Deze code is echter niet gecentraliseerd in een .c- bestand aangezien er telkens kleine wijzigingen aangebracht werden, bijvoorbeeld de functie die het preprocessen van de zoekstring omvat. Maar in principe is het mogelijk om de code voor het inlezen in een aparte .c file onder te brengen. In een vorige implementatie werden de karakters ingelezen met behulp van getchar(). Ik merkte echter dat deze functie enkel geoptimaliseerd is voor het inlezen van 1 karakter. De file werd veel sneller ingelezen met behulp van het fread() commando.

In deze sectie leg ik kort de werking van de 3 verschillende implementaties uit. Als keuze voor het 3e algoritme ging mijn voorkeur uit naar Boyer-Moore. Bij het implementeren echter had ik behoorlijk veel moeite bij de good-suffix heuristiek. Ik ben er niet in geslaagd deze in lineaire tijd te kunnen berekenen en dacht dat mijn naïevere implementatie waarin de good-suffix heuristiek $O(n^2)$ tijd kostte voldoende snel zou zijn. Achteraf besloot ik om nog eens Boyer-Moore-Horspool te implementeren. Nu bleek dit algoritme in de praktijk vaak beter te presteren. Boyer-Moore-Horspool werd dan maar opgenomen als search3. Ik heb wel een lineaire implementatie van de good-suffix heuristiek opgezocht en deze toegevoegd aan mijn Boyer-Moore algoritme. Boyer-Moore maakt dus geen deel uit van de 3 search algoritmen, maar wordt enkel gebruikt om te vergelijken. In de map src2 vind je echter ook de sourcecode voor Boyer-Moore*(waarvan enkel de implementatie van de good-suffix heuristiek niet van mij afkomstig is)*.

1.a Shift-Or

Deze implementatie is vrij analoog aan de cursus. Enkel bij zoekstrings > 64 tekens had ik de keuze tussen ofwel meerdere malen shift-or toe te passen, ofwel enkel de eerste 64 tekens te zoeken m.b.v. shift-or, en indien die gematched worden, verder te werken aan de hand van Brute-Force. Beide werden geïmplementeerd, maar ik kwam tot de conclusie dat de 1e implementatie vrij ingewikkeld werd, en daarentegen meestal geen significante snelheidswinst opleverde. Dit is ook vrij logisch: Indien je een zoekstring hebt die meer dan 64 tekens bevat en die reeds matched met 64 tekens in een zoekstring, is de kans vrij groot dat je een match zult vinden op die positie. Enkel voor gevallen met een beperkt alfabet(bv. DNA-sequenties) zou je nog kunnen twijfelen. Uiteindelijk besloot ik om voor de brute-force methode te kiezen.

1.b Knutt-Morris-Pratt(KMP)

Ook hier is de implementatie vrij analoog aan de cursus. Ik heb enkel nog de functie 'vergelijk_strings' wat verder geoptimaliseerd aangezien deze functie het meest opgeroepen wordt tijdens de uitvoering. Ik heb een variabele weggelaten en de lusconditie gewijzigd. Achteraf werd de functie inline gezet, wat een snelheidswinst van 10% opleverde.

Het idee achter het KMP-algoritme kan je als volgt samenvatten:

Wanneer we een mismatch hebben op een bepaalde plaats, vergeet het naïef brute-force algoritme alle informatie omtrent de vorige vergelijkingen(bijvoorbeeld, 3 karakters die wel al matchten). Brute-Force zal deze vergelijkingen keer op keer opnieuw uitvoeren. KMP maakt echter gebruik van die informatie om de volgende positie te bepalen(aan de hand van de verschuivingstabel). KMP doet dus geen 'overbodige' vergelijkingen.

1.c Horspool

Dit algoritme is in feite een vereenvoudiging van het Boyer-Moore algoritme dat enkel gebruikt maakt van de Bad character heuristiek. Voor de uitleg i.v.m. de bad character heuristiek verwijs ik naar sectie 1.d . Ik schets kort de werking van dit algoritme aan de hand van een voorbeeld.

We definiëren de functie f(x) als de functie die het meest rechtse voorkomen van karakter 'x' in het zoekpatroon weergeeft. We laten het laatste karakter buiten beschouwing. Stel dat we een patroon 'GAGC' hebben die we moeten zoeken in 'CTGCAGCTA'. Dan is het vrij duidelijk dat voor elk karakter $x \notin \{G,A\}$ =f(x) = -1. Voor de letters die wel in ons zoekpatroon voorkomen:

Karakter	f(x)
A	1
G	2

In de while lus kijken we eerst of het laatste en eerste karakter van het zoekpatroon met de tekst overeenkomen, pas dan gaan we vergelijken, dit is een kleine optimalisatie die ervoor zorgt dat we slechts in enkele gevallen de 2e while-lus moeten doorlopen. In die lus vergelijken we van rechts naar links en gaan door zolang alles goed is. Wanneer we een mismatch op positie i tegenkomen, dan shiften we eerst over de volledige lengte van het zoekpatroon. Vervolgens moeten we er nog ervoor zorgen dat er correct gealligneerd wordt. Hiervoor gebruiken we de jumptable die we eerder aangemaakt hebben. Hieronder ziet u een uitwerking (zonder de kleine optimalisatie). Cursief staat aangegeven hoever we springen bij een mismatch (iedere keer de lengte van het zoekpatroon -1). Vervolgens moeten we nog correct alligneren m.b.v. de jumptable.

Uitwerking:

C	T	G	C	A	G	C	G	A	G	C
G	A	G	C							
			G	*A*	*G*	*C*				
				G	A	G	C			
							G	*A*	*G*	*C*
					G	A	G	C		
								G	*A*	*G*
							G	A	G	C

1.d Boyer-Moore

De kracht van Boyer-Moore en Horspool schuilt hem hoofdzakelijk in het feit dat men de vergelijkingen van rechts naar links uitvoert. Dit in combinatie met 2 verschuivingstabellen zorgt ervoor dat Boyer-Moore het snelste van de 4 algoritmes is.

Omdat de implementatie van Boyer-Moore vrij 'droog' is(in essentie werken we constant met verschuivingstabellen) worden hier kort de verschillende verschuivingstabellen toegelicht. We onderscheiden de bad character en de good suffix heuristiek:

Bad character

Dit is de eenvoudigste heuristiek. Met bad character bedoelen we het karakter in de tekst dat een mismatch veroorzaakte. Indien dit karakter nog ergens anders voorkomt in het zoekpatroon, kunnen we deze aligneren. We moeten er wel voor zorgen dat er **gealigneerd** wordt met **het meest rechtse voorkomen** in het patroon. Bijvoorbeeld:

Positie	0	1	2	3	4	5	6	7
Tekst	C	G	A	T	T	C	A	A
VOOR	A	T	C	C				
NA			A	T	C	C		

In de eerste stap veroorzaakt T(=bad character) op positie 3 reeds een mismatch(=VOOR), maar T komt voor op positie 1 van ons zoekpatroon, zodat we deze kunnen allingneren(=NA), we zijn dus in 1 stap 2 posities opgeschoven.

De bad charcter verschuivingstabel werd geïmplementeerd in de methode makejumptable() en de uiteindelijke verschuivingstabel komt in de variabele 'tab' terecht.

Stel nu dat onze zoekstring niet 'ATCC' was, maar 'TGAT'. Indien we dezelfde tekst beschouwen zouden we na 3 succesvolle vergelijkingen te maken hebben met een negatieve shift, en dus terugspringen. We zouden kunnen eisen om geen negatieve shifts te ondernemen, maar hier maken we nog gebruik van een extra heuristiek nl. de good suffix heuristiek.

Good suffix

Indien we met een negatieve shift te maken hebben bij de bad character heuristiek is het beter deze niet uit te voeren, maar gebruik te maken van eerdere matches om de maximale verschuiving te bepalen. We beschouwen 2 gevallen.

Positie	0	1	2	3	4	5	6	7
Tekst	C	A	A	T	T	C	A	A
Zoek1	A	T	A	T				
Zoek2			A	T	A	T		

Hier hebben we na 2 matches een mismatch op positie 1. Normaal zouden we hier een negatieve verschuiving hebben. Maar wij weten immers dat we het patroon 'AT' al kunnen matchen op positie 2 en 3. Patroon 'AT' komt nogmaals voor in onze zoekstring, zodat we dit kunnen alligneren(zoek2). Nu hebben we verschoven over 2 posities i.d.p.v. -1.

Positie	0	1	2	3	4	5	6	7
Tekst	C	T	A	T	T	C	A	A
Zoek1	A	T	A	T				
Zoek2			A	T	A	T		

Hier is het duidelijk dat het patroon 'TAT' niet 2x in de zoekstring voorkomt. Je zou verwachten dat je dus mag verschuiven tot over positie 3. Bemerk echter dat we een deelstring van 'TAT' namelijk 'AT' in het begin van de zoekstring voorkomt. We moeten hier dus extra voorzichtig zijn en ervoor zorgen dat dit deel correct gealigneerd wordt.

De implementatie van deze heuristiek is niet zo eenvoudig, en lijkt misschien op het eerste gezicht niet veel performantiewinst op te leveren. Dit is niet helemaal waar. Indien het alfabet klein is, zoals het geval in DNA sequenties, wordt vrij veel gebruik gemaakt van de good-suffix heuristiek omdat de kans dat herhaalde patronen in de zoekstring voorkomen veel groter is.

Verder spreekt het voor zich dat deze heuristieken veel beter zullen werken wanneer de lengte van het zoekpatroon toeneemt. Dit wordt nagegaan in paragraaf 4.

2. Testfiles

Om er zeker van te zijn dat de 3 algoritmes correct werken, heb ik een aantal verschillende testcases opgesteld. Er moet met name goed gekeken worden naar het aantal matches, en de positie van de matches. Ook is het belangrijk dat het algoritme goed kan omgaan met speciale karakters zoals 'ë'. Ik heb een klein perl scriptje('/src2/randomgenerator') geconstrueerd dat zowel random tekst, als random zoekquerys kan genereren. Ik ben begonnen met een klein alfabet, maar je kan het script makkelijk uitbereiden naar een groter alfabet. Om ervoor te zorgen dat mijn eerste algoritme correct werkte, kon ik op bepaalde plaatsen mijn zoekquery invoegen en dan kijken of dat matchte.

Om ervoor te zorgen dat de andere algoritmen exact dezelfde output genereren als het eerste algoritme, kan je de output vergelijken met het cmp-commando onder linux(zie src/cmp.sh).

3. Complexiteit

Vooraleer ik verschillende testen toelicht zal ik eerst kort de complexiteit van de verschillende algoritmen proberen te verklaren. Ik beschouw telkens de complexiteit van de preprocessing fase en de zoekfase afzonderlijk. De lengte van het zoekpatroon stellen we voor als 'm', en de lengte van de tekst gelijk aan 'n'. Het alfabet wordt voorgesteld door Σ.

Shift-Or

De preprocessing fase is begrensd door de grootte van het alfabet. Voor elke letter moet immers de binaire karakteristieke vector berekend worden. De complexiteit bedraagt dus $O(|\Sigma|)$.

Het zoeken zelf kan dan in principe in O(m*n) gebeuren. En doordat we gebruik maken van bitvectoren kunnen de bitoperaties zeer snel uitgevoerd worden.

Knutt-Morris-Pratt

De complexiteit van KMP vraagt O(n).De berekening van complexiteit werd gegeven in de cursus en daar wordt dus niet dieper op ingaan.

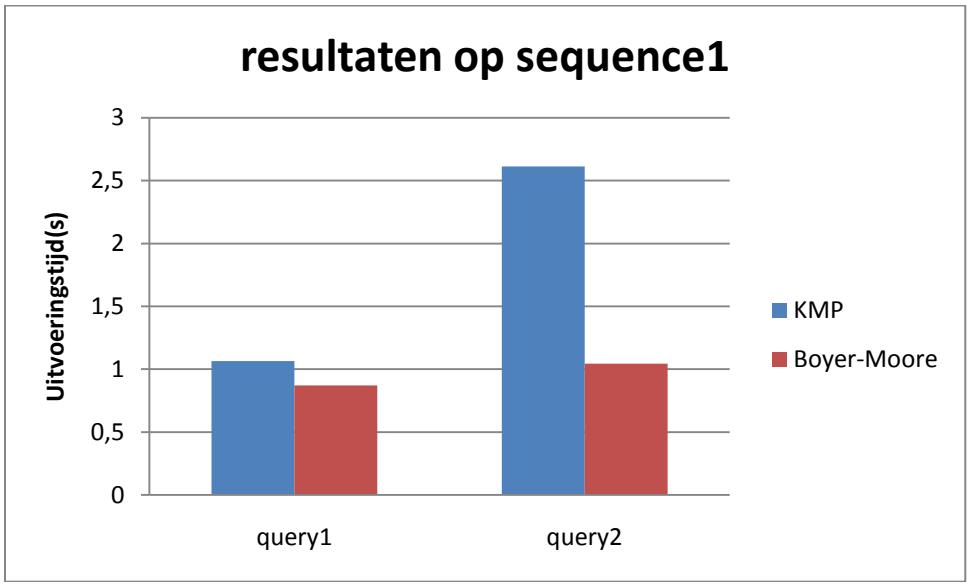
Het slechtste geval bij KMP komt voor wanneer het patroon er als volgt uit ziet:

Patroon Verschuivingstabel	aaaa 1111
Sequentie	aaabaaabaaab…aaab

Hoewel het opbouwen van de verschuivingstabel hier snel gaat, kan er slechts in elk geval geshift worden over exact 1 positie. Verder wordt de lus in vergelijk_strings() maximaal doorlopen. Er zijn dus n*(m-1) vergelijkingen nodig bij het zoeken.

opmerking: Dit scenario is een optimaal scenario bij Boyer-Moore & Horspool, waarbij in dit geval slechts n/m vergelijkingen nodig zijn. Om dit te illustreren heb ik 2 verschillende querys geconstrueerd en uitgevoerd op 1 sequentie van de vorm (aaab)^100000.De eerste query is (aaab).

We zien dat het verschil tussen beide algoritmes vrij klein blijft. De tweede query is een worst-case scenario voor KMP (zoekpatroon 'aaaa').Hier is het verschil enorm.



De fundamentele reden waarom KMP zoveel trager is dan Boyer-Moore in de praktijk ligt in het feit dat de meeste mismatches tijdens het zoeken gebeuren bij het eerste karakter of het begin van het zoekpatroon. Hierdoor zal KMP zijn verschuivingstabel niet optimaal kunnen benutten.

Boyer-Moore & Boyer-Moore-Horspool

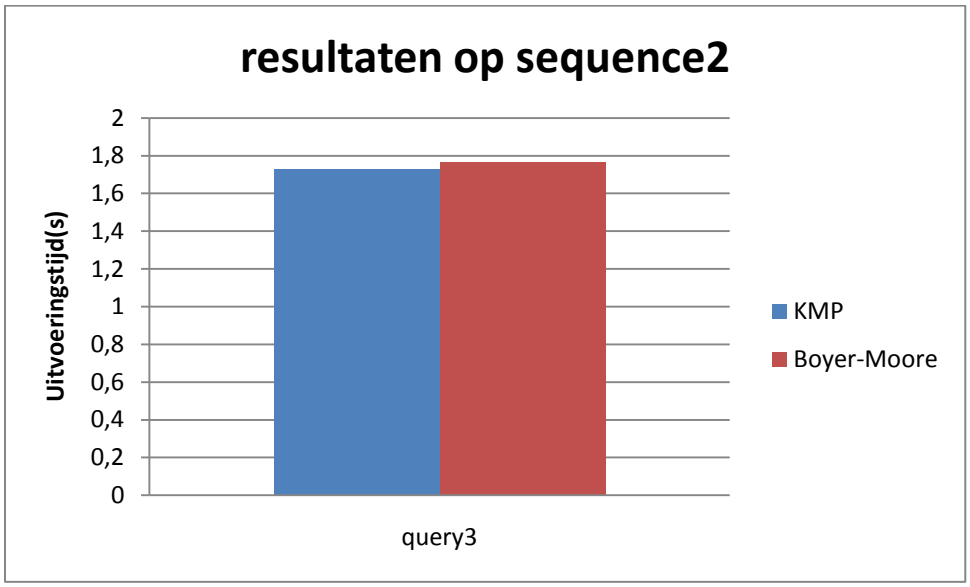
Ook hier is de complexiteit van de preprocessing fase afhankelijk van de grootte van het alfabet(met name bij de bad character heuristiek). We kunnen stellen dat de complexiteit gelijk is aan Θ(m + |Σ|).

Het zoeken kan wel in lineaire tijd gebeuren. Het is bewezen1 dat het volledig doorzoeken, ongeacht welk patroon zich in de tekst bevindt, in 3n vergelijkingen kan gebeuren bij Boyer-Moore. Horspool zal de tekst volledig doorlopen in O(n*m).

Het worst case scenario van beide algoritmen ziet er als volgt uit:

Zoekpatroon	Baaa
sequentie	aaa..a

Dit is een van de weinige gevallen waar Boyer-Moore-Horspool slechter presteerde dan KMP.

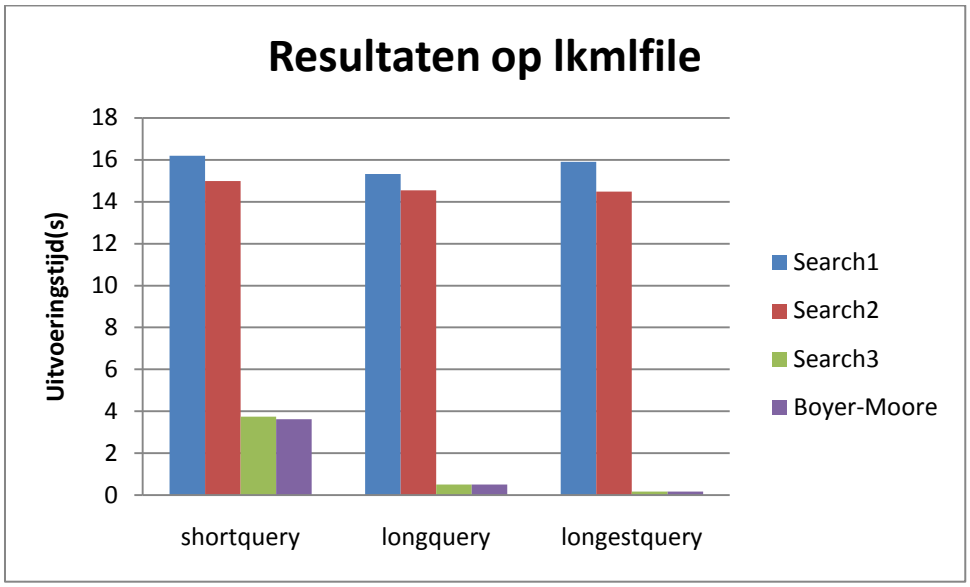


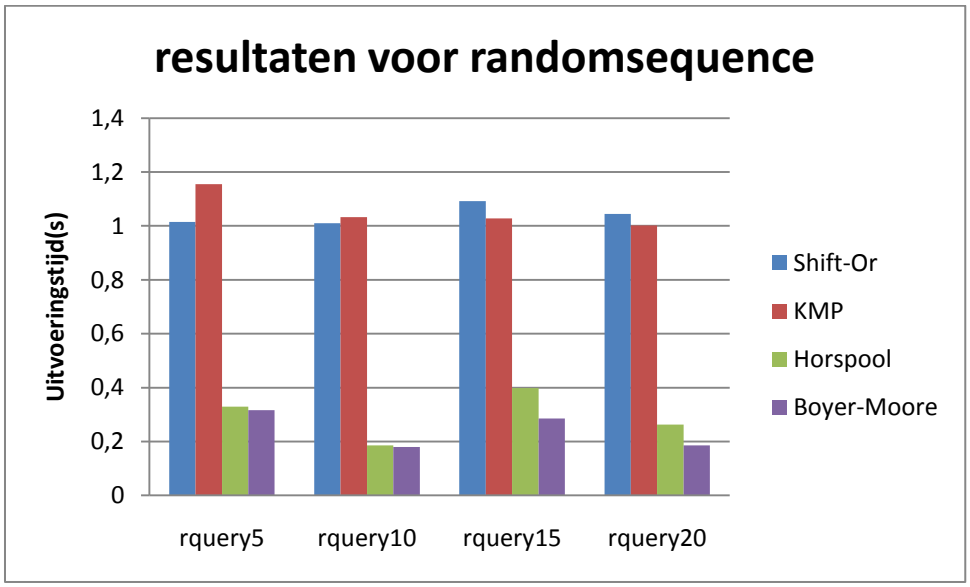
De kans dat dit slechtste geval voorkomt in gewone tekst is echter vrijwel onbestaand.

[1] ^ Richard Cole (1991). "Tight bounds on the complexity of the Boyer-Moore algorithm". *Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms*. pp. 224–233.

4. Tijdsmetingen

Alle testen werden uitgevoerd op Genix. Tijdsmetingen werden bekomen met behulp van het 'time' commando onder linux. Daarbij werd enkel gekeken naar de 'user'-time. Dit is de tijd gebruikt door het eigenlijke programma en alle subroutines dat het programma oproept. Eerst werden alle algoritmen met elkaar vergeleken. Eerst met een gewoon alfabet (256 karakters in de map testcase1) en vervolgens met een beperkt alfabet(4 karakters in de map testcase2) zoals bijvoorbeeld DNA-sequenties. We zien dat in beide gevallen KMP en Shift-Or ongeveer even goed presteren. Boyer-Moore en Horspool presteren daarentegen veel beter. Merk op dat bij de 2e grafiek het verschil tussen Boyer-Moore en Horspool soms groter is, door de extra good-sufffix heuristiek.





Patroonlengte Boyer-Moore en Horspool

Een belangrijke eigenschap van deze algoritmen is dat ze beter presteren bij een langer zoekpatroon. Ik heb dit eens nagegaan.

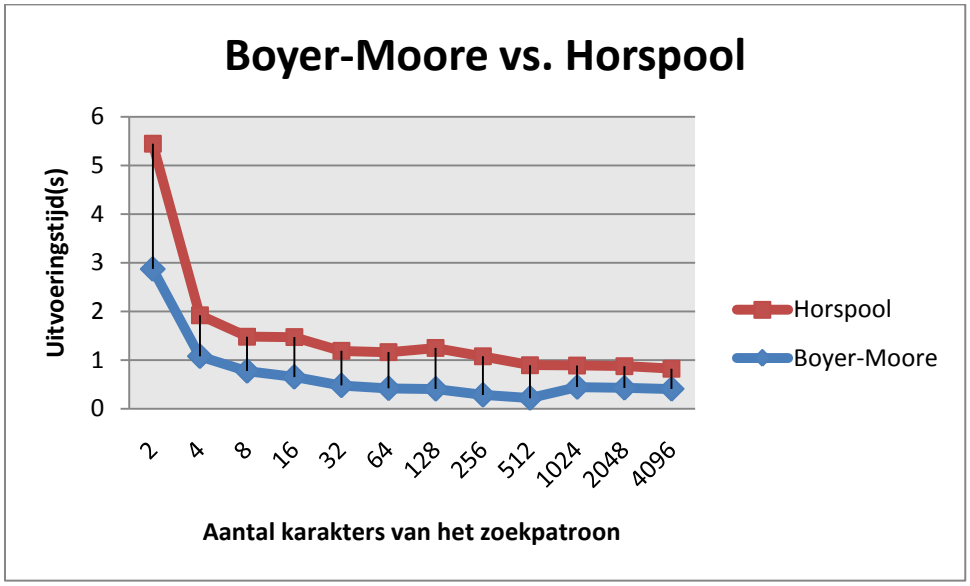
Als inputlengte heb ik een log2 schaal genomen. Het eerste bestand had een input van lengte 2, 4, enz... De patronen zijn eenvoudige random DNA-querys en werden uitgevoerd op het bestand sequencemodified.txt, dat 2x het bestand sequence.txt bevatte(testcase3). Hoewel je natuurlijk in beschouwing moet nemen dat het zoekpatroon verschillend is waardoor de shifts variëren kun je toch een aantal belangrijke vaststellingen doen.

Allereerst klopt het dat hoe langer de inputstring wordt, hoe sneller beide algoritmes werken.

Doordat we met een klein alfabet werken heeft Boyer-Moore het bijkomend voordeel dat hij goed gebruik kan maken van de good-suffix heuristiek, waardoor deze altijd sneller zal presteren dan Horspool.

Indien de lengte echt zeer lang wordt zal je merken dat de preprocessing van onze zoekpatronen de belangrijkste bottleneck zal worden. Doordat Boyer-Moore hier 2 verschuivingstabellen zal moeten aanmaken, verkleint het verschil in snelheid tussen 2 algoritmes. Dit kunnen we duidelijk waarnemen bij de drie laatste tijdmetingen.

De knik in de Boyer-Moore functie die ontstaat wanneer de inputlengte van 512 naar 1024 karakters overgaat kan verklaard worden doordat de preprocessing van het zoekpatroon langer zal beginnen duren dan het eigenlijke zoeken. Hierdoor zal het algoritme trager beginnen werken.



Geheugencomplexiteit

Ten slotte ben ik nog eens gaan kijken naar de geheugencomplexiteit van mijn algoritmes. Allereerst is het belangrijk op te merken dat de heap er ongeveer exact hetzelfde zal uitzien. Het alloceren van de zoekbuffer en de tekstbuffer gebeurt op een gelijkaardige manier met behulp van malloc en realloc. Het meten van het geheugengebruik heb ik gedaan met behulp van het programma valgrind. Valgrind is een programma dat gebruikt kan worden om geheugenlekken op te sporen in processen. Daarnaast biedt het echter ook een mogelijkheid om aan geheugenprofiling te doen m.b.v. de massif optie. Met behulp van onderstaande commando wordt het totale aantal geheugen dat het programma verbruikt gemeten op bepaalde 'snapshots'.

```
valgrind --tool=massif --pages-as-heap-yes --time-unit=B ./searchX patroon zoekfile
```

Met behulp van het commando ms_print geheugensearchX kan er dan vervolgens een grafische weergave bekomen. Ik merkte op dat alle algoritmen ongeveer evenveel geheugen gebruikten. Enkel wanneer de zoekstring echt zeer groot is, merkte ik dat KMP en Boyer-Moore meer geheugen nodig hadden. Dit is logisch aangezien zij een verschuivingstabel bevatten waarvan de lengte afhankelijk is van de lengte van het zoekpatroon. De tabellen waar Horspool en Shift-Or van gebruik maken zijn enkel afhankelijk van de grootte van het alfabet.