Algoritmen en Datastructuren II

Academiejaar 2012-2013

Zoeken van deellijsten door middel van suffixbomen

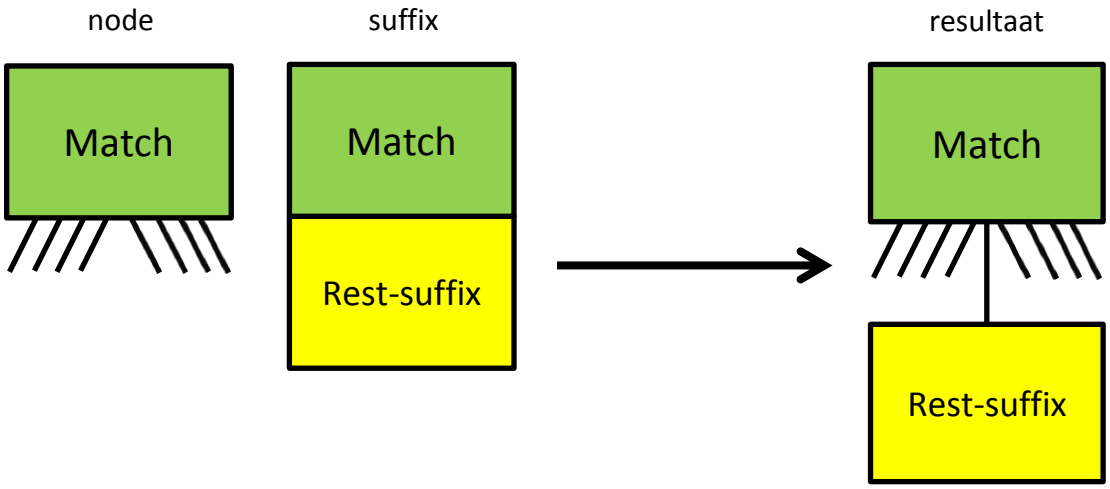
Michiel De Witte

Construct:

Het constructie-algoritme bij de verschillende implementaties is quasi hetzelfde. Het begint met aan de gegeven lijst met shorts de sentinel toe te voegen. Dan worden alle suffixen toegevoegd aan de root node, van klein naar groot. Hierdoor moet men geen rekening houden dat de toe te voegen suffix kleiner is dan de node waarmee men vergelijkt.

Het toevoegen aan de root node gebeurt met het recursieve add-algoritme. Hierbij wordt de positie (index in de lijst waar de suffix start) en de node waaraan men de suffix moet toevoegen(oudernode) meegegeven. Bij die node die meegegeven wordt aan de methode wordt er een kind gezocht waarbij de eerste short van het label matcht met de short in de sequentielijst op de meegegeven positie. Hier kunnen zich 2 gevallen voordoen: ofwel wordt een node gevonden ofwel niet.

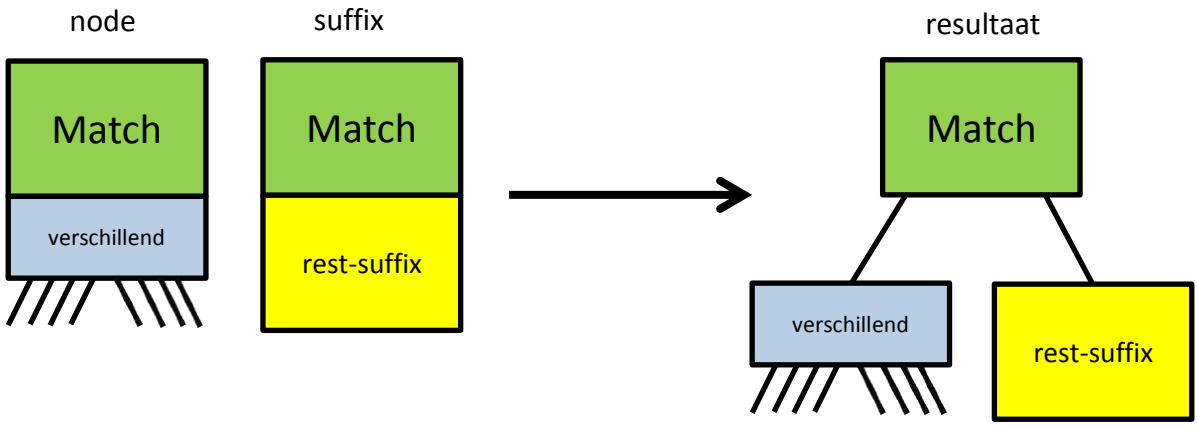
In het geval er geen node wordt gevonden voegt men een node toe aan de oudernode met als label de positie die meegegeven werd en de positie van de sentinel in de sequentie. In een node wordt er bijgehouden als het een leaf is en of het een root is. Dus is de oudernode geen leaf en de toegevoegde node wel.



In het geval er wel een node gevonden wordt, vergelijkt het algoritme het label van die node met de rest van de suffix na de positie. Hier zijn er terug 2 gevallen: er is een volledige match van de label naar de suffix of er is een gedeeltelijke match.

Bij een volledige match voert men een recursieve add uit met de gematchte node en als positie de vorige positie + de lengte van de node-label.

De bewerking die gebeurt bij een gedeeltelijke match splitst de node waarmee gematcht wordt op in 2 stukken, het stuk die overeenkomt met de suffix en het stuk die niet matcht. Hierbij wordt het gematchte deel de ouder van het niet gematchte deel. Het niet gematchte deel krijgt alle kinderen van de originele node. Dan wordt bij de ouder ook nog een nieuwe node toegevoegd die de rest van de suffix voorstelt.



Count:

Als de query leeg is geef ik de grootte terug van de sequentie van de suffix tree. Anders ga ik vanaf de root node zoeken naar het eerste kind die met de eerste positie van de query matcht.

Indien er geen kind is return ik 0.

Indien er wel een kind is zijn er weer 2 mogelijkheden: als de query maar 1 short bevat is er een volledige match, en roep ik de methode searchChildren op met als argument de huidige node.

Anders is de query groter en roep ik de methode subCount op met als argumenten de query, de node en de index (in dit geval 1).

De methode subCount is weer een recursieve methode. Eerst wordt er gecontroleerd als de label in de node 1 lang is.

Indien dit zo is wordt er een kind gezocht waarvan de eerste short van de label matcht met die op de index positie van de query. Als er een kind gevonden wordt dan word er gekeken als de index matcht met het aantal elementen in de query, dan roep ik de methode searchChildren op met als argument de node.

Indien label van de node langer is dan 1 wordt gans het label afgelopen en met de query gematcht. Als er een short niet matcht dan return ik 0, als de short wel matcht overloopt hij de query verder. Als de laatste short in de query matcht wordt de methode searchChildren opgeroepen met als argument de node.

Als gans de label van de node matcht met de query wordt opnieuw een kind gezocht waarvan de eerste short in de label matcht met de volgende short in de query. Indien er

geen kind meer bestaat return ik een 0. Als er een kind bestaat wordt gekeken of het de laatste short van de query was indien zo wordt de methode searchChildren opgeroepen. Als het niet zo was wordt met die node de subCount recursief terug opgeroepen.

De methode searchChildren is ook weer recursief en geeft het aantal bladeren terug die zich bevinden onder de meegegeven node. Hierbij wordt als de node een leaf is een 1 terug gegeven. En indien een ouder worden alle searchChildren waarden van zijn kinderen opgeteld en teruggegeven.

Een andere implementatie van searchChildren is bijvoorbeeld dat elke node in de constructie zijn aantal bladeren bijhoud. Wat het constructie algoritme niet in tijdscomplexiteit zou doen stijgen.

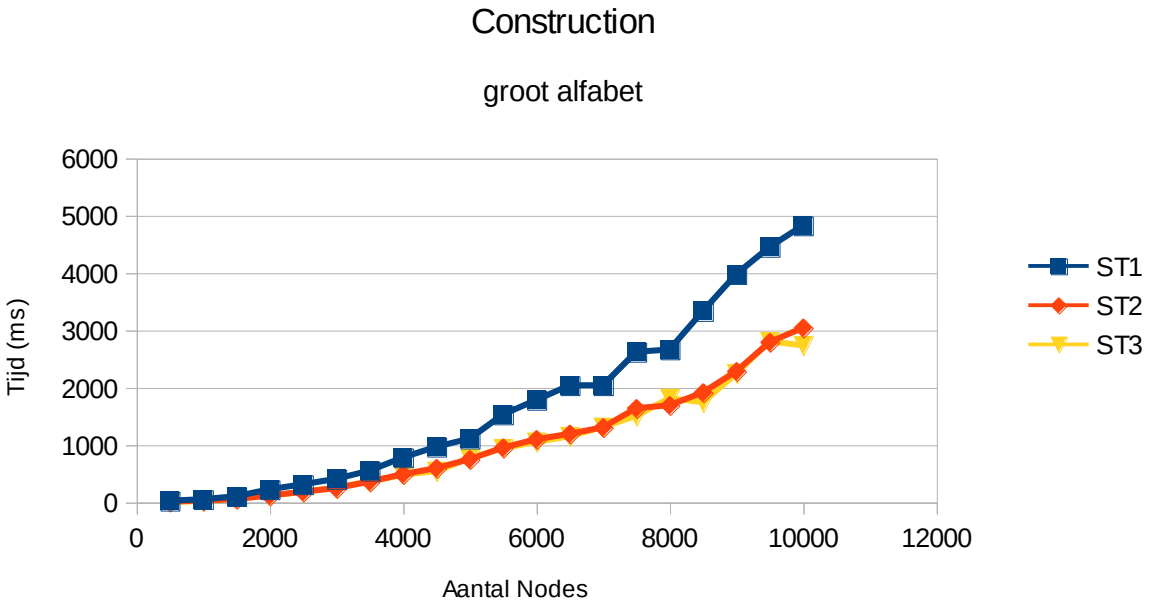
Varianten en efficiëntie bijhouden van kinderen:

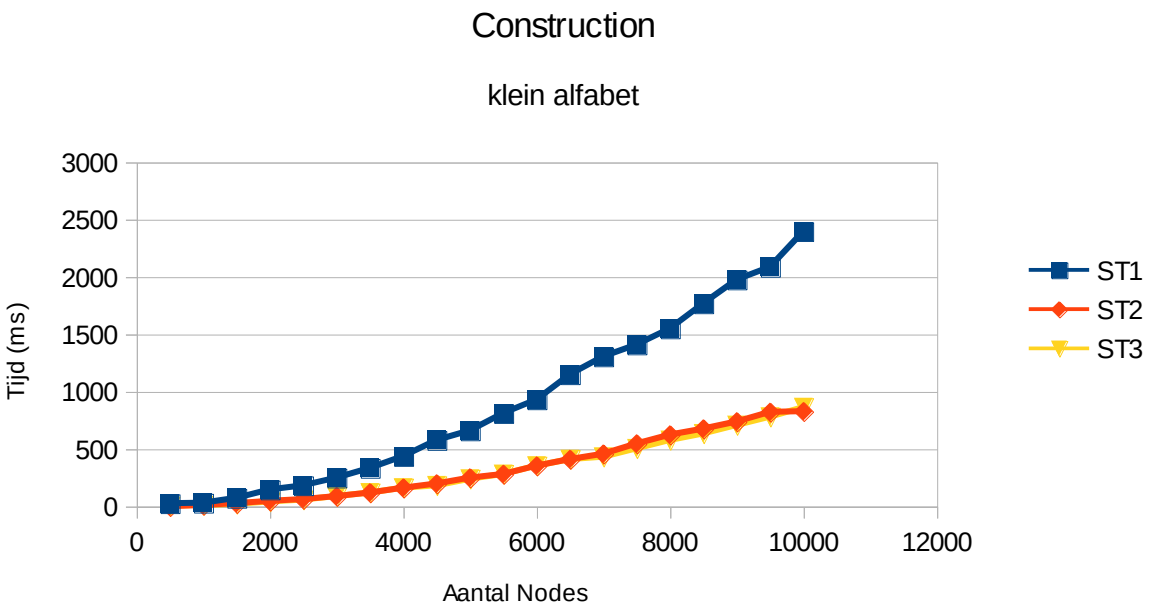
Varianten:

Ik heb 3 verschillende implementaties gemaakt:
SuffixTree1 = een ArrayList implementatie waarbij de label als ArrayList wordt opgeslagen
SuffixTree2 = een HashMap implementatie met de label als begin en eind index van een lijst
SuffixTree3 = een ArrayList implementatie met de label als begin en eind index van een lijst

De gebruikte resultaten van tijdsmetingen zijn het gemiddelde van 3 keer de test uit te voeren. De tijdsmetingen, gemiddelden inclusief de grafieken kunt u vinden in “extra/resultaten.ods”.

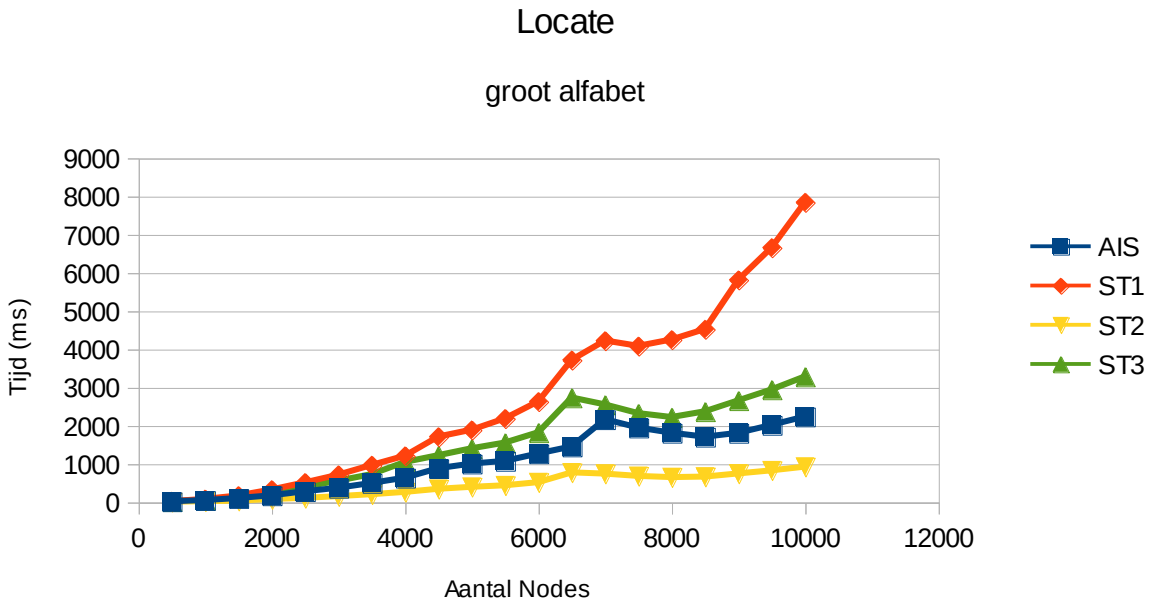
Constructie:
Aangezien AltIntervalSeacher geen constructie nodig heeft heb ik deze niet in de tijdsmetingen opgenomen van de constructie.

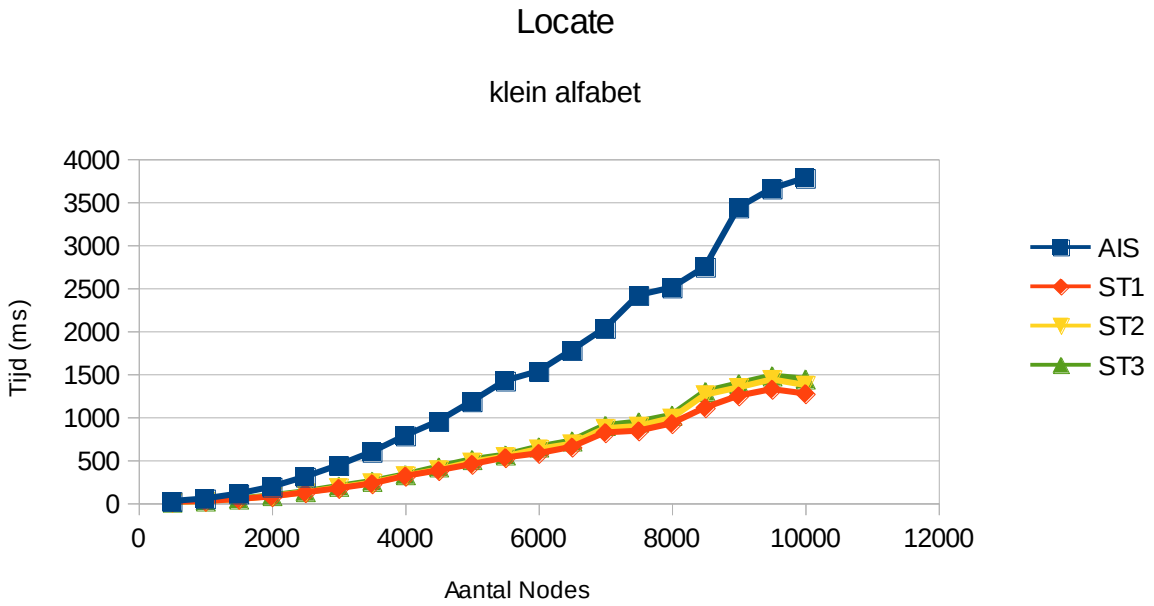




Hier ziet men telkens duidelijk dat de constructie bij de ST1 implementatie langer duurt. Dit vanwege het aanmaken van de vele ArrayLists die gebruikt worden als label van de node. Ook ziet u dat het alfabet een invloed heeft op de snelheid van de constructie, bij een klein alfabet is de constructie dubbel zo snel.

Locate:

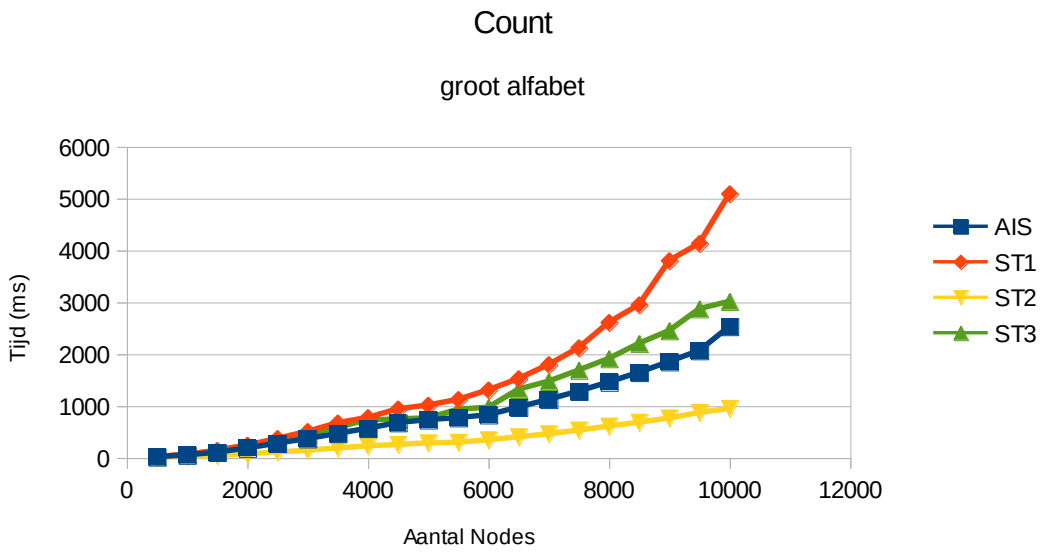


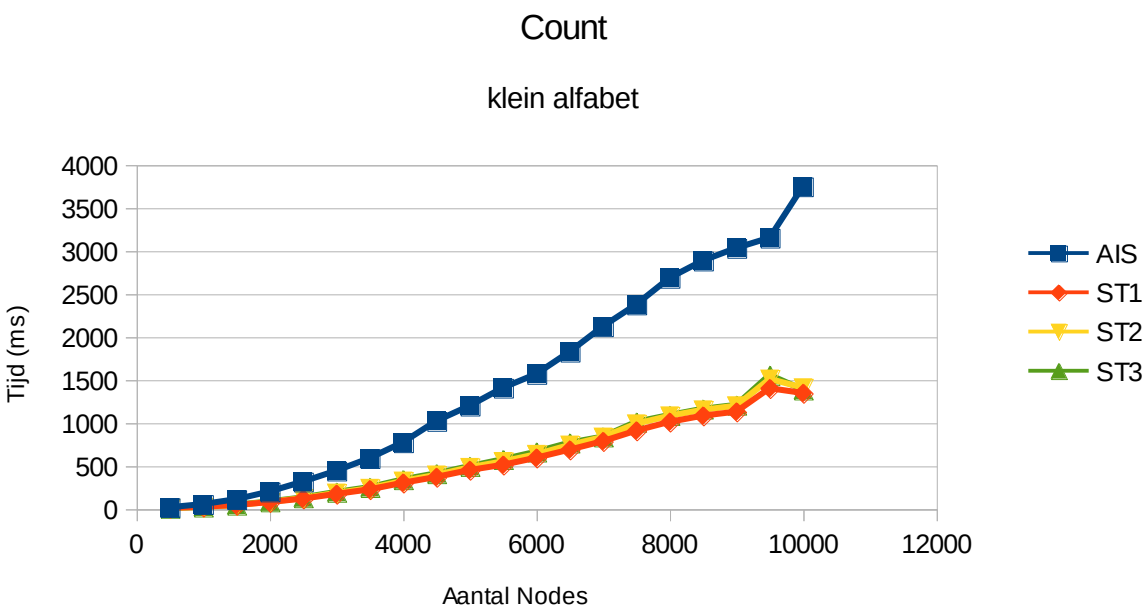


Hier is duidelijk te zien dat een boom bij een klein alfabet duidelijk voordelig is. Dit doordat de boom ervoor zorgt dat elke locate ruwweg lineair is t.o.v. het aantal shorts in de query. Dit omdat elke ouder weinig kinderen heeft, en dus kun je hier niet zien dat je telkens het passende kind moet zoeken een grote array van kinderen moet itereren. Bij een groot alfabet zie je duidelijk dat de array-implementaties dan weer in de problemen komen. De HashMap implementatie blijft in beide gevallen eigenlijk de beste keuze want het opzoeken van het juiste kind gebeurt in constante tijd dus heeft HashMap dat nadeel niet. Dit kun je

goed zijn bij een groot alfabet waar elke ouder veel kinderen heeft. En dus de eigenschap van in constante tijd het juiste kind op te zoeken duidelijk grote voordelen bied.

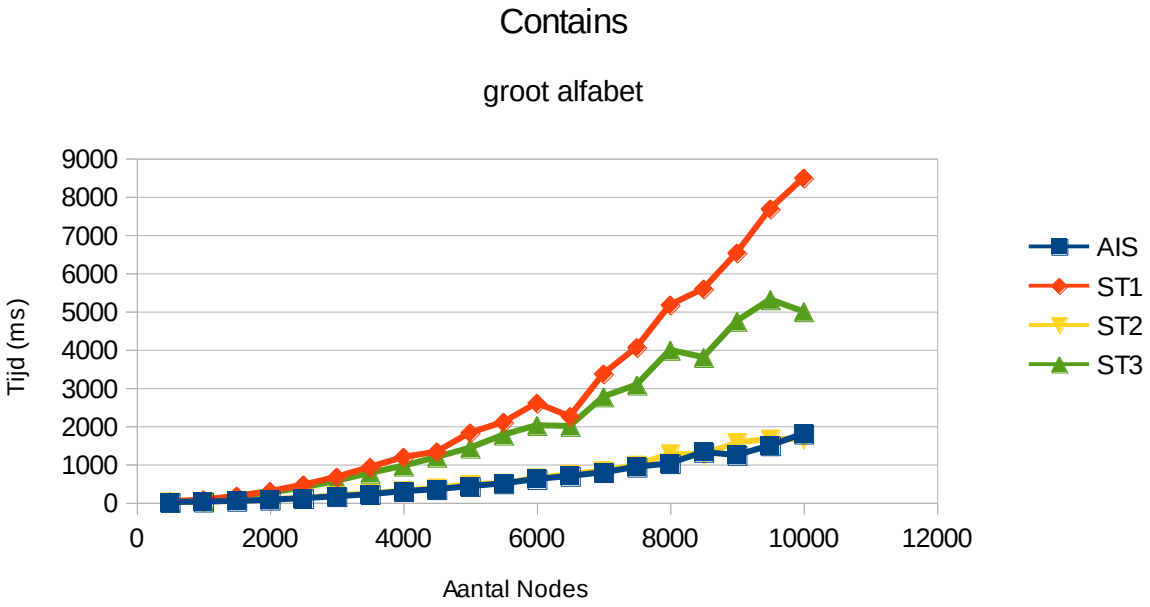
Count:

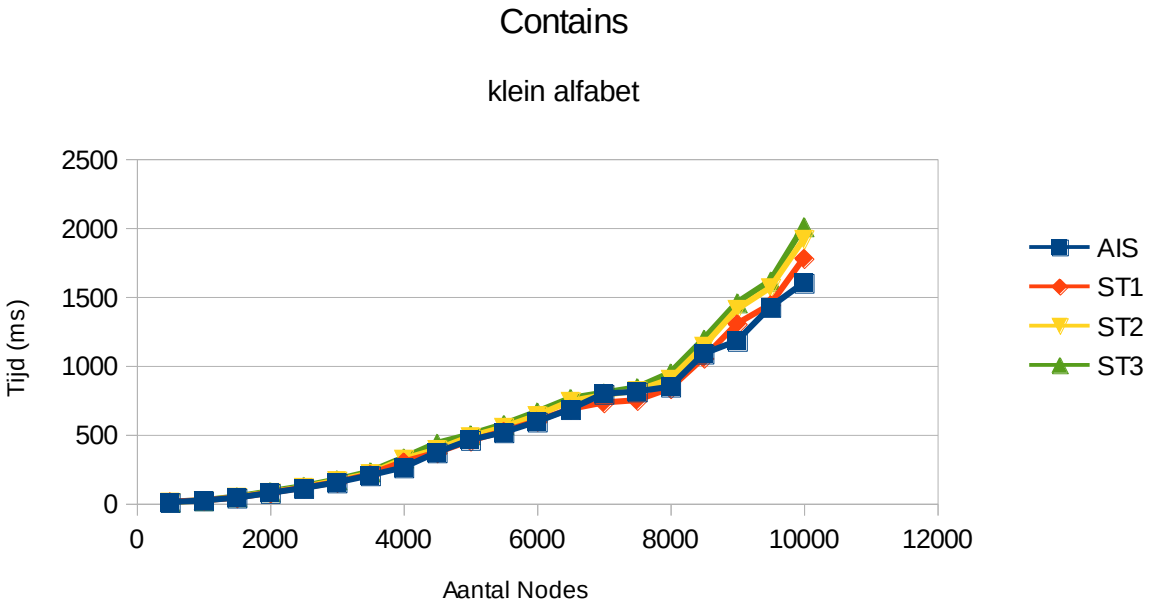




Hier kun je duidelijk opnieuw het voordeel van de HashMap implementatie t.o.v. een ArrayList implementatie zien bij een groot alfabet. Hierbij zijn de tijdsmetingen analoog als bij Locate, dit vanwege het bijna identieke algoritme.

Contains:





Hier zijn ook weer de verschillen te merken bij een groot alfabet

Besluit:

Tussen de varianten van ArrayList is het belangrijkste verschil de geheugengrootte, en de constructietijd. De HashMap implementatie is duidelijk in alle gevallen beter dan ArrayList. De constructie tijd Het is duidelijk dat de HashMap implementatie de snelste is. Dit omdat het tegenover arraylists alleen voordelen bied door de constante opzoekingstijd en de constante toevoegingscomplexiteit. Deze datastructuur is enkel nuttig als men meerdere

opzoekingen gaat doen, door de constructietijd. Ook zullen toepassingen met kleine alfabetgroottes de meeste performantie-verhoging zien.

Tests:

Correctheid:

Om de correctheid van mijn algoritmes aan te tonen heb ik gebruik gemaakt van jUnit tests die de uitvoer van de AltIntervalSearcher vergelijken met die van mijn algoritmes. Dit gebeurt in SuffixTreeTester.
Hierbij heb ik random short sequenties aangemaakt en alle deellijsten overlopen en de resultaten van de functies met als argument die deellijsten met elkaar vergeleken. Ook heb ik random lijsten getest en de uitvoer van de functies vergeleken.

Ook heb ik tests uitgevoerd om de geldigheid van de boom te testen. Zoals dat elke inwendige top uitgezonderd de wortel minstens 2 kinderen heeft, en elk label uitgezonderd van de wortel niet leeg is. Dit gebeurt in SuffixTreeValidator minstens2kinderen().
Nog een test is een algoritme die kijkt hoeveel leafs er zijn, dit moet gelijk zijn aan het aantal suffixen, dus gelijk zijn aan de lijstgrootte. Dit gebeurt in SuffixTreeValidator countLeafs().
Een andere eigenschap die ik gevalideerd heb is, is dat de eerste short van de label van de kindnode uniek moet zijn samen met die van de andere kinderen van elke node.
Dit gebeurt in SuffixTreeValidator firstShortTest().

Theoretische vragen:

Minimum aantal toppen in een suffixboom van een lijst van n getallen:

Aangezien we een minimum aantal toppen willen, moeten we het deel van het algoritme vermijden dat een node en een suffix ombouwt naar 3 toppen. Aangezien er een ander deel van het algoritme een node en een suffix ombouwt naar 2 toppen is dit wenselijk. De sentinel voorkomt dat er een volledige match komt bij een toe te voegen suffix dus een recursieve add volgt altijd op een splitsing naar 3 toppen. Dit proberen we te vermijden, dus moet de suffix altijd direct aan de root node als kind toegevoegd worden.

Een lijst die aan de voorwaarde voldoet waarbij dat alleen dat stuk van het algoritme wordt opgeroepen, vereist dat de eerste short in de toe te voegen suffix niet voorkomt als eerste short in een label van een kind van de root node. Hieruit volgt dat elk eerste element van de suffix verschillend moet zijn dan de rest van de suffixen. En dus dat elk element in de lijst uniek voorkomt. Bv. [1,2,3,4,…] .

Hieruit volgt dat aangezien men eerst een root-node heeft en een lijst met n elementen, en dus n toe te voegen suffixen (de sentinel alleen wordt niet toegevoegd), dat men n+1 nodes minimaal heeft in de boom.

Maximum aantal toppen in een suffixboom van een lijst van n getallen:

Dit is juist het omgekeerde verhaal, aangezien men het maximum aantal toppen wilt hebben moet men proberen het stuk van het algoritme te bereiken dat altijd een node en een suffix splitst naar 3 nodes. Bij het toevoegen van de eerste suffix kan dit niet omdat de root node nog geen kinderen heeft en dus wordt de suffix als node toegevoegd. Bij volgende bewerkingen kan dit wel. Dan wordt bij elke toevoeging van een suffix van 1 node 3 gemaakt. Dus komt er per toegevoegde suffix (bij uitzondering de eerste) 2 toppen bij. Het maximum aantal nodes in een suffixboom van een lijst van n shorts is dus 2n uitgezonderd bij een lege lijst waar het 1 is (de root node).

Een lijst kunnen we creëren door de eigenschap toe te passen dat het eerste element van de suffix ervoor moet matchen met het eerste element van de volgende suffix. Hieruit volgt dat alle elementen in de lijst gelijk zijn. Bv. [1,1,1,1,…] .

Circulaire referentielijst:

Aangezien er nooit deellijsten worden gezocht die langer zijn dan de referentielijst kunnen we 2 mogelijke implementaties gebruiken. We zetten de lijst 2 keer na elkaar en voegen we alle suffixen hiervan toe, hierbij is de maximale diepte 2n. Ofwel voegt men alle suffixen toe, en als men halverwege zit en de suffixen worden langer dan de originele lijst, vervangt men het overige gedeelte met de sentinel. Dus kan gegarandeerd worden dat de diepte maximaal n is. Het add algoritme zal men ook moeten aanpassen want hier werd geen rekening gehouden als er zich een volledige match voordoet.
Geheugencomplexiteit is maximaal de grootte van 2 keer de lijst erin te steken dus 2n dus als geheugencomplexiteit blijft dit lineair. De tijdscomplexiteit blijft ook dezelfde want dit is ook gewoon 2 keer de complexiteit van de originele lijst toe te voegen.