Zoeken van deellijsten door middel van suffixbomen

Project Algoritmen en Datastructuren II

Bart Middag

Academiejaar 2012-2013

1. TESTS

1.1 Correctheid

Om de correctheid van de construct-, contains-, count-en locate-algoritmen te testen, heb ik in de testklasse twee aparte methoden gemaakt: `test1(AbstractSuffixTree)` en `test2(AbstractSuffixTree)`. Beide methoden nemen een `AbstractSuffixTree` als argument en kunnen dus met beide `SuffixTree`-klassen uitgevoerd worden. Deze twee methoden testen voor zelfgemaakte bomen verschillende mogelijkheden en printen de bomen ook uit via mijn `printDetails()`-methode, die mooi van elke `TreeNode` in de boom het suffix en de index uitprint en dus heel nuttig was voor het debuggen van de algoritmen.

Ook heb ik terwijl ik in de grote run() methode van de testklasse programmeerde, laten uitprinten of de resultaten van contains, count en locate dezelfde waren als die van `AltIntervalSearcher`. Dit heb ik later achterwege gelaten omdat dit toch altijd zo was.

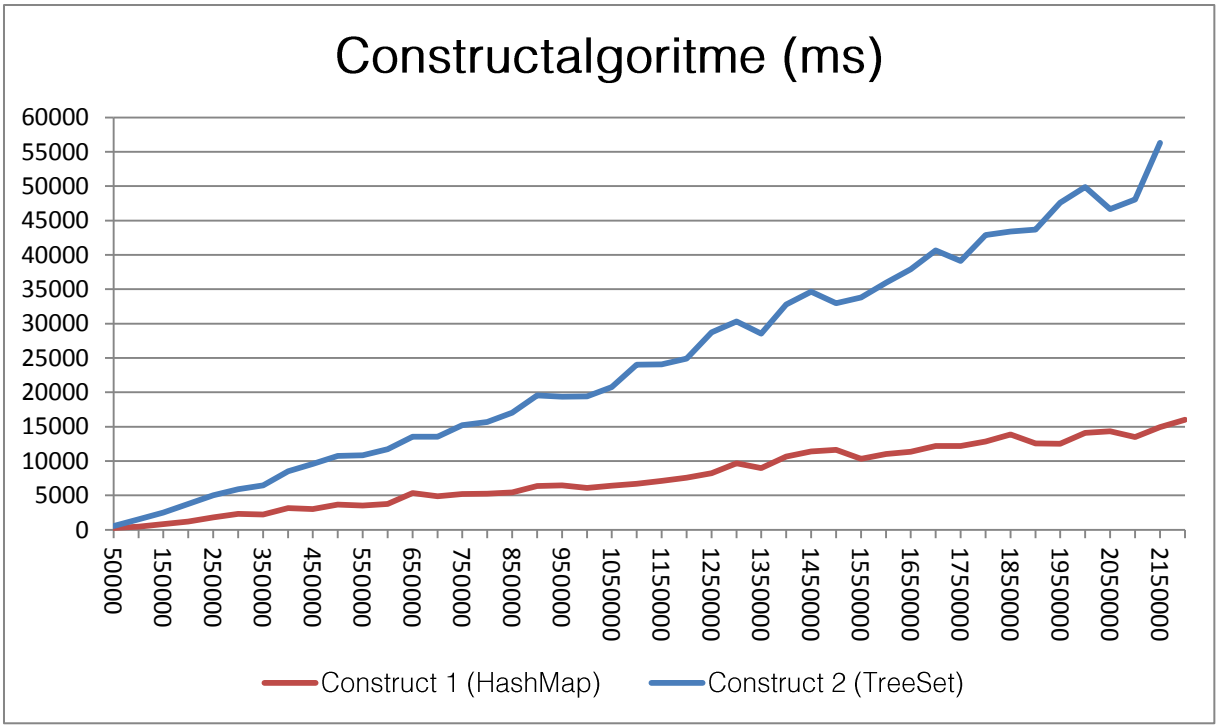
1.2 Tijdsmetingen

Na veel optimalisaties heb ik besloten om mijn algoritmen te testen op Helios, de Unix-server van de Universiteit Gent. Voor de tijdsmetingen heb ik gebruik gemaakt van `ThreadMXBean` in plaats van `System.currentTimeMillis()`. Dit meet de CPU-tijden en zorgt zo voor meer accurate resultaten.

De tests werden uitgevoerd met een alfabetgrootte van `Short.MAX_Value` (32767) en 20 willekeurige queries (lijsten die opgezocht werden) die maximaal even groot waren als de lijst zelf.

Voor kleinere alfabetgroottes worden alle algoritmen trager, voor grotere alfabetgroottes worden ze sneller. Dit kan men gemakkelijk testen door het bijhorende argument van de run-methode aan te passen. Voor meer hierover, zie de commentaar in de main-methode.

Voor de tests van contains, count en locate heb ik gekozen voor willekeurige queries omdat het realistischer is dat we niet weten of de lijst de query wel degelijk bevat. Dit gaf echter bijna steeds tijden van 0 ms voor 20 willekeurige queries. De tijden voor queries die subLists zijn van de originele lijst liggen een stuk hoger, maar ook daarvoor zijn mijn algoritmen duidelijk efficiënter dan die van de `AltIntervalSearcher`. Om dit te testen, kan men gewoon één argument veranderen in de run-methode van de test. Voor meer hierover, zie de commentaar in de main-methode.



Grafiek 1: Het constructiealgoritme. Op de x-as staat de grootte van de lijst waarvoor de boom wordt opgebouwd - op de y-as staat de tijd in ms.

1.3 Tijdscomplexiteit

De tijdscomplexiteit van het constructalgoritme lijkt bij beide implementaties lineair te zijn, maar schijn bedriegt: vermits het algoritme bij elk van de *n* mogelijke suffixen eerst moet zoeken of een deel van de suffix nog niet in de boom staat – en daarbij overloopt het alle toppen op een bepaald pad – zal de complexiteit in het slechtste geval (alfabetgrootte 1) kwadratisch zijn.

Daarbij komt dan nog de kost van het opzoeken in de lijst van de kinderen van elke top: in de implementatie met de `HashMap` gebeurt het opzoeken van de volgende top op dit pad constant (dus daar is de totale kost kwadratisch voor alfabetgrootte 1), maar een `TreeSet` doet er *O(log m)* (met *m* het aantal kinderen) tijd over om het juiste kind te vinden. Bij alfabetgrootte 1 is dit verwaarloosbaar, maar hierdoor zal wel de bestegevalcomplexiteit verschillen, namelijk als op alle posities van de lijst een ander short-getal staat.

Bij een `HashMap` is het opzoeken constant, dus daar zal de bestegevalcomplexiteit lineair zijn. Bij een `TreeSet` is het opzoeken echter logaritmisch, dus daar zal de bestegevalcomplexiteit *O(n log n)* zijn. Gelukkig ligt, zoals te zien op de grafiek, de complexiteit van het gemiddelde geval héél dicht bij de complexiteit van het beste geval.

2. ALGORITMEN

2.1 Algemeen

2.1.1 Construct

Het constructiealgoritme begint met een kopie te maken van de originele lijst en de sentinel achteraan toe te voegen. Deze kopie is nodig, anders blijft de sentinel in de originele lijst voor mogelijke andere bewerkingen met deze lijst.

Daarna maak ik één `TreeNode` aan met defaultwaarden. Dit wordt de root. Hierna overloop ik de lijst van achter naar voor en voeg ik voor elke `Short` in die lijst het suffix vanaf die waarde toe als kind van de root met de methode `addChild`.

De methode `addChild` probeert eerst te kijken of de huidige node (in het begin dus de root) al een kind heeft waarvan de eerste short van het suffix dezelfde is als de eerste short van het kind dat we willen toevoegen. Dat doen we met `quickGetChild`.

- Als dit `null` teruggeeft en dit dus niet bestaat, mogen we het kind gewoon aan de huidige node toevoegen, zonder enige wijzigingen.
- Anders moeten we het suffix van deze node van voor naar achter overlopen en kijken tot waar dat gelijk is aan het suffix van het kind dat we willen toevoegen.
    - Als dit het einde van het suffix bereikt, herhaalt het algoritme zich voor deze node als huidige node en trekken we het al bekeken deel van het suffix af van het kind dat we willen toevoegen.
    - Als het suffix maar deels gelijk is het suffix van het kind dat we willen toevoegen, wordt de huidige node gesplitst op de plaats waar het niet meer gelijk is. Het gelijke deel blijft bestaan op de positie van de huidige node, en dit krijgt als kinderen de niet-gelijke delen van de huidige node en het kind dat we willen toevoegen.

2.1.2 Contains

Eerst controleren we of de query leeg is, want dan moeten we sowieso altijd `true` teruggeven. Daarna zoeken we met de hulp van `quickGetChild` het kind van de huidige node (nu nog de root) waarvan de eerste short van het suffix gelijk is aan de eerste short van de query.

- Als dit niet bestaat, komt het de query niet voor in de sequence en geven we dus false terug.
- Anders moeten we het suffix van deze node van voor naar achter overlopen.
    - Als dit het einde van het suffix bereikt, herhaalt het algoritme zich voor deze node als huidige node en trekken we het al bekeken deel van de query af.
    - Als de query verschilt van het suffix voor het einde van dit suffix, komt het niet voor in de sequence. We kunnen dus `false` teruggeven.

Zodra het einde van de query bereikt wordt, weten we dat het in de sequence zit. Dan kunnen we `true` teruggeven.

2.1.3 Count

Dit algoritme steunt op ongeveer hetzelfde principe als contains, maar in plaats van `true` terug te geven, gaat het vanaf de eerstvolgende top alle bladeren van die deelboom teruggeven door die recursief op te tellen.

2.1.4 Locate

Het locate-algoritme steunt op hetzelfde principe als count, maar in plaats van alle bladeren op te tellen gaat het de index van elk blad in de `Set` steken en de `Set` teruggeven.

2.2 Implementaties

Het grote verschil tussen beide implementaties van `AbstractSuffixTree` is dat bij `SuffixTree1` de kinderen van elke `TreeNode` bijgehouden worden in een `HashMap` en bij `SuffixTree2` in een `TreeSet`.

Daarom gebeuren `quickGetChild` en `quickPutChild` van `SuffixTree1` in constante tijd, terwijl dezelfde methoden van `SuffixTree2` eigenlijk in logaritmische tijd gebeuren. `TreeNode1` moet ook geen `short`-waarde bijhouden, terwijl `TreeNode2` dat wel moet doen, want de `TreeSet` is gesorteerd in stijgende volgorde van deze waarden.

3. THEORIEVRAGEN

3.1 Minimaal en maximaal aantal toppen

3.1.1 Minimaal aantal toppen

Het minimaal aantal toppen van een suffixboom is *n+1*, met *n* het aantal elementen in de sequence. Vermits elke suffixboom exact n bladeren heeft, kunnen we hieruit besluiten dat we alleen meer toppen kunnen hebben als we interne toppen toevoegen. Interne toppen voegen we alleen toe als er twee suffixen beginnen met hetzelfde element, dus als alle elementen verschillend zijn, zullen we een suffixboom hebben met een minimaal aantal elementen.

3.1.2 Maximaal aantal toppen

Het maximaal aantal toppen van een suffixboom is *2n-1*. Dit doet zich voor als alle elementen in de sequence dezelfde zijn. We weten dat er exact *n* bladeren zijn in een suffixboom voor een sequence van *n* elementen. Nu moeten we dus juist nog het aantal interne toppen kennen. Een interne top heeft minstens (en als alle elementen gelijk zijn, exact) 2 kinderen. Als alle elementen dezelfde zijn, is minstens één van deze kinderen is een blad. Beide kinderen van de diepste top zijn bladeren. In dit geval zijn er dus voor *m*

interne toppen exact *m+1* bladeren. We weten dat er n bladeren zijn, dus zijn er *n-1* interne toppen (we rekenen de wortel mee als interne top). De som is dus gemakkelijk gemaakt: *n* bladeren + *n-1* interne toppen = maximaal *2n-1* toppen in een suffixboom.

3.2 Circulaire lijsten

Enkel de zoekalgoritmen moeten worden aangepast.

In plaats van `false` terug te geven bij contains als het algoritme bij de sentinel terechtkomt, zouden we gewoon de sequence vanaf het begin kunnen overlopen en kijken of de rest van de query gelijk is aan het begin van de sequence.

We zouden dus naast onze normale `quickGetChild` telkens `quickGetChild` moeten uitvoeren om te kijken of er een kind is dat begint met de sentinel en als dat zo is telkens de controle doen of het begin van de sequence gelijk is aan de rest van de query.

Bij de recursieve count hetzelfde, maar enkel als de query nog niet leeg is. Telkens als we bij een interne top aankomen, wordt dus `quickGetChild` opgeroepen voor de sentinel. Dan doen we dezelfde controle als bij contains. Als die `true` teruggeeft, tellen we er 1 bij op, anders niets. Dan gaan we verder met de normale count.

Hetzelfde bij locate. In plaats van wat we bij count doen, zetten we gewoon de index van de top die de sentinel bevat in de `Set` en gaan we verder met de normale locate.

De complexiteit wordt hierdoor niet erg aangetast. Het vergelijken van de twee lijsten vanaf het begin gebeurt met in lineaire kost, terwijl bij het normale zoekalgoritme voor een SuffixTree naast het vergelijken met lineaire kost ook telkens nog het goede kind moet bepaald worden met `quickGetChild` (constant bij `HashMap` en logaritmisch bij `TreeSet`).