UNIVERSITEIT GENT

Zoeken van deellijsten door middel van suffixbomen

Project Algoritmen en Datastructuren II

Academiejaar 2012-2013

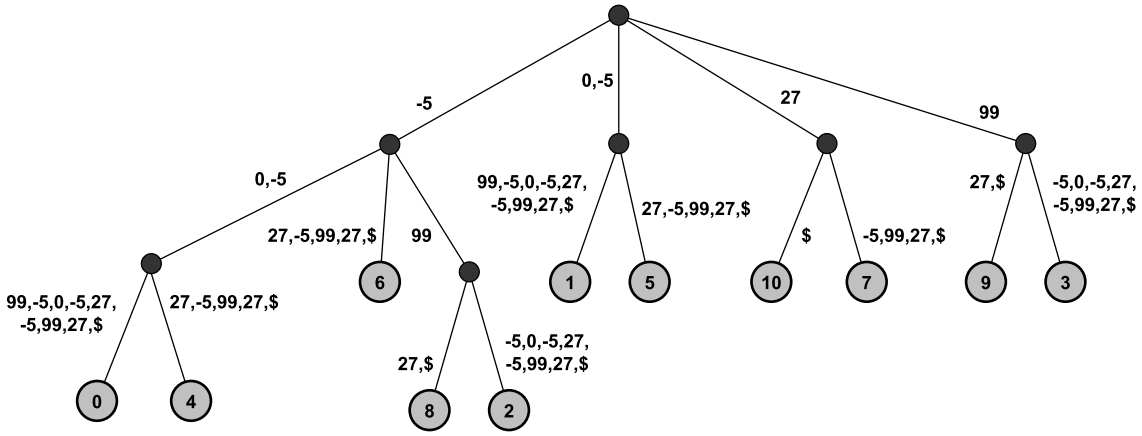
1 Inleiding

Dit project omvat het implementeren van algoritmen voor het zoeken van deellijsten in lijsten van short-getallen door middel van gespecialiseerde datastructuren, namelijk suffixbomen. Suffixbomen zijn krachtige zoekbomen die gebruikt worden om snel allerhande opzoekingen te verrichten op strings, sequenties of lijsten. Het project bestaat uit het voorzien van meerdere efficiënte implementaties van suffixbomen. Dit omvat ook het implementeren van een constructiealgoritme en algoritmen om sub- of deellijsten te vinden binnen lijsten van short-getallen. Het is belangrijk om voor- en nadelen van de verschillende implementaties te onderzoeken, alsook de suffixboomimplementaties te vergelijken met algoritmen die geen gebruik maken van deze datastructuur.

De suffixboom is een heel krachtige, maar toch eenvoudige, datastructuur die veel wordt gebruikt voor het oplossen van zoekproblemen op strings, sequenties en lijsten. Deze populariteit is vooral te danken aan het feit dat vele verschillende zoekproblemen in theoretisch optimale tijdscomplexiteit kunnen worden uitgevoerd op suffixbomen. Zo kunnen sequenties van miljoenen of zelf miljarden karakters snel worden doorzocht. Praktische toepassingen van suffixbomen zijn onder andere te vinden in datamining, datacompressie, detectie van plagiaat in teksten en in de bioinformatica. Eén van jullie assistenten maakt bijvoorbeeld gebruik van (varianten van) suffixbomen voor het afbeelden van korte DNA-fragmenten op genoomsequenties.

Het belangrijkste verschil tussen suffixbomen en de zoekbomen die in de cursus staan beschreven is de datastructuur waarin gezocht wordt. De zoekbomen uit de cursus opereren namelijk op verzamelingen van tekens en suffixbomen opereren op lijsten of strings van tekens. Meestal worden suffixbomen gebruikt voor strings over een zeker alfabet Σ met grootte $\sigma = |\Sigma|$. Voor DNA-sequenties is dit bijvoorbeeld het alfabet {`A`,`C`,`G`,`T`}, waarbij de vier letters staan voor de vier nucleotiden *Adenine*, *Cytosine*, *Guanine* en *Thymine*.
Andere toepassingen vereisen echter een veel groter alfabet. Om alle courante toepassingen met grote alfabetten te omvatten worden in dit project de strings voorgesteld als lijsten van short-getallen. Alle strings of lijsten zullen elementen bevatten uit het alfabet $\Sigma = \{-2^{15} \dots 2^{15} - 2\}$ en de alfabetgrootte kan oplopen tot $\sigma = 2^{16} - 1$.

De naam suffixboom verwijst enerzijds naar de vorm van de datastructuur (een boom) en anderzijds naar het feit dat de boom informatie bevat over alle suffixen van de lijst waarvoor hij is opgebouwd. Ter herinnering: een suffix van een lijst $S = s_0, s_1, \dots, s_{n-1}$ is een lijst $s_{n-m}, \dots, s_{n-1}$, met $0 < m \leq n$. Analoog is een prefix van S een lijst $s_0, \dots, s_{m-1}$, met $0 < m \leq n$. Algemeen is een lijst P een deellijst van S indien er indices i en j bestaan, $0 \leq i \leq j < n$, waarvoor $P = s_i, \dots, s_j$.



Figuur 1: Suffixboom voor de lijst $[-5, 0, -5, 99, -5, 0, -5, 27, -5, 99, 27]$. Het \$-teken in de figuur is een speciaal stopsymbool dat gelijk is aan `SHORT_MAX` in de implementatie. Merk op dat elk suffix van de lijst kan worden teruggevonden op een pad van de wortel tot een blad.

Definitie 1 *Een suffixboom $\mathcal{T}$ voor een lijst $S = s_0, \ldots, s_{n-1}$ over een alfabet Σ is een gewortelde gerichte boom met exact n bladeren die elk een uniek label hebben uit de verzameling $\{0 \ldots n-1\}$. Elke inwendige top (dit is een top met minstens 1 kind), uitgezonderd de wortel, heeft minstens twee kinderen en elke boog tussen twee toppen wordt gelabeld met een niet-lege deellijst van S. Het eerste teken van het label van twee verschillende bogen die vertrekken uit dezelfde top is steeds verschillend. Verder geldt dat voor elk blad met label i, de samenvoeging van alle labels op de bogen van het unieke pad van de wortel tot dat blad het suffix $s_i, \ldots, s_{n-1}$ uitspelt.*

Figuur 1 geeft een voorbeeld van een suffixboom, namelijk voor de lijst $[-5, 0, -5, 99, -5, 0, -5, 27, -5, 99, 27]$. Deze grafische voorstelling van een suffixboom bevat echter niet alle implementatiedetails en een letterlijke vertaling van deze grafische voorstelling kan inefficiënt zijn in de praktijk. Bij de implementatie is het dus belangrijk om aandacht te hebben voor de datastructuren die de verschillende componenten (toppen, bogen, labels) opslaan.

Het stopsymbool \$ in figuur 1 stelt een karakter voor dat uniek is in de lijst en enkel als laatste symbool wordt gebruikt. Dit symbool wordt gebruikt om technische redenen. Immers, als er geen \$-teken gebruikt zou worden, zou suffix nummer 10 in figuur 1 niet eindigen in een blad. Maak in je implementatie best ook gebruik van dit stopsymbool. Je mag er vanuit gaan dat geteste lijsten nooit dit stopsymbool zullen bevatten.

2 Implementatie en verslag

2.1 Algemeen

Constructie-algoritme
Vooraleer er zoekoperaties op een suffixboom kunnen worden uitgevoerd, dient de suffixboom eerst opgebouwd te worden. Dit opbouwen hoeft slechts één keer te gebeuren voor een gegeven sequentie van short-getallen. Dit algoritme neemt meteen een volledige lijst als invoer, en er wordt niet verwacht dat deze lijst op een later tijdstip uitgebreid of gewijzigd zal worden.

Implementeer een constructie-algoritme voor suffixbomen en leg in je **verslag** grondig uit hoe je constructie-algoritme werkt.

Zoekoperaties
Daarnaast dienen onder andere ook de meest voorkomende zoekopdrachten op suffixbomen geïmplementeerd te worden, namelijk: (*i*) het bepalen of een lijst een deellijst is van de referentielijst (`contains`), (*ii*) het tellen van het aantal keer dat een lijst kan worden teruggevonden in de referentielijst (`count`) en (*iii*) het vinden van de posities waar een lijst kan worden teruggevonden in de referentielijst (`locate`).

De volledige lijst van operaties die dienen geïmplementeerd te worden kunnen jullie vinden in sectie 2.3.

Belangrijk: beschrijf in je **verslag** daarnaast ook grondig het algoritme dat je gebruikt om te tellen hoe vaak een gegeven suffix voorkomt in de suffixboom. De algoritmes die je gebruikt voor de andere operaties (zie sectie 2.3) hoeven in je verslag niet beschreven te worden.

Verschillende varianten
Bij de meeste bomen die jullie in de les gezien hebben of nog zullen zien (bijvoorbeeld AVL- of rood-zwart bomen), hebben de toppen ten hoogste 2 kinderen. Bij suffixbomen kunnen de toppen echter veel meer kinderen hebben, daarom is het belangrijk dat de kinderen van een top op een efficiënte manier geraadpleegd kunnen worden. Er zijn verschillende datastructuren mogelijk om de kinderen van een top bij te houden, bijvoorbeeld:

- Een HashMap.
- Een TreeMap.
- Een ArrayList.
- Een gesorteerde ArrayList.
- Een TreeSet.
- ...

Het is de bedoeling dat jullie **minstens 2** varianten kiezen om de kinderen van een top bij te houden en voor elk van deze varianten de constructie- en zoekoperaties te implementeren. Probeer hierbij duplicatie van code te vermijden.

Jullie dienen ook de efficiëntie van de varianten die jullie gekozen hebben grondig te vergelijken en te bespreken in jullie **verslag**. Hierover kunnen jullie meer informatie vinden in sectie 2.2.

2.2 Tests

Vermeld in je **verslag** hoe je de **correctheid** van alle onderdelen van je implementatie getest hebt en hoe je ervoor gezorgd hebt dat je **tijdsmetingen** accuraat waren. Vermeld bij je tijdsmetingen ook duidelijk welke soort input je gebruikt hebt: genereer verschillende soorten testdata met bepaalde eigenschappen. **Vergelijk zoveel mogelijk aspecten** van de verschillende varianten die jullie geïmplementeerd hebben, dus niet alleen de efficiëntie van jullie constructie-algoritmen maar ook die van de verschillende zoek-operaties. Mogelijke parameters waarop getest kan worden zijn het aantal en de grootte van de queries, de alfabetgrootte van de lijsten,... Bedenk zelf nog andere parameters die de vergelijking mogelijks beïnvloeden. De testcode zal ook ingediend moeten worden in een afzonderlijke map (zie sectie 3).

Op `minerva` kunnen jullie de klasse `AltIntervalSearcher.java` vinden. Jullie mogen dit bestand niet wijzigen. Het bevat een implementatie van een alternatief algoritme om verschillende operaties op deellijsten uit te voeren. Dit algoritme maakt geen gebruikt van suffixbomen en vraagt ook geen initiële constructiefase. Vergelijk de efficiëntie van de verschillende varianten van jullie algoritme voor suffixbomen ook met dit alternatief algoritme en bespreek dit ook in jullie **verslag**. Jullie mogen deze gegeven klasse ook gebruiken bij jullie correctheidstesten.

Voorzie duidelijke en correcte grafieken en tabellen, maar becommentarieer en verklaar je resultaten ook voldoende. Zorg ervoor dat je ook een duidelijk besluit vormt. Komen je testresultaten overeen met wat je verwacht had? Indien niet, **verklaar** de verschillen. Bespreek in het besluit zeker de voor- en nadelen van de alternatieve methode, de suffixboommethode en de verschillende suffixboomimplementaties.

2.3 Concrete implementatie

Al je suffixboomimplementaties dienen de interface `IntervalSearcher` te implementeren en andere datastructuren dienen de interface `TreeNode` te implementeren (zie `minerva`). Deze interfaces mogen niet gewijzigd worden! De interface `IntervalSearcher` beschikt over volgende methoden:

`void construct(List<Short> sequence);`	bouwt de suffixboom voor de gegeven lijst op. Dit is de referentiesequentie.
`boolean contains(List<Short> query);`	geeft `true` terug indien de suffixboom de gegeven deellijst bevat, anders `false`.
`int count(List<Short> query);`	geeft het aantal keer dat de deellijst in de suffixboom voorkomt terug.
`Set<Integer> locate(List<Short> query);`	geeft de posities terug waar de deellijst voorkomt in de suffixboom (zie code voor een voorbeeld).
`TreeNode getRoot();`	geeft de wortel van de suffixboom terug.

Opgepast: aangezien de invoer bestaat uit lijsten van `Short`-**objecten** in plaats van het primitieve type `short`, is het belangrijk dat je de juiste vergelijkingsmethode gebruikt om de gelijkheid van twee objecten te bepalen (bijvoorbeeld door de `equals(Short o)` methode op te roepen). Als alternatief kunnen jullie de `Short`-**objecten** ook omzetten naar `short`'s door

middel van de `shortValue()`-methode.

Lees aandachtig de commentaar die zich in de code bevindt! In de commentaar staat ook hoe jullie methodes zich moeten gedragen wanneer ze opgeroepen worden met **lege queries**. De interface `IntervalSearcher` definieert ook de constante `SENTINEL`. Dit is een karakter waarvan gegarandeerd wordt dat het nooit zal voorkomen in de inputsequenties. Jullie mogen hiervan gebruik maken in jullie implementatie. De interface `TreeNode` stelt een top van de suffixboom voor.

Opdat we je project op een efficiënte manier zouden kunnen testen, vragen we je om alle klassen die de interface `IntervalSearcher` implementeren van een **default constructor** te voorzien en deze klassen `SuffixTree1`, `SuffixTree2`, ..., `SuffixTreeN` te noemen.

Vermeld in je **verslag** ook met welke algoritmes `SuffixTree[1-N]` overeenkomen.

Een volledig project dient **tenminste** één klasse te bevatten per geïmplementeerde variant voor het bijhouden van de kinderen van een top in de suffixboom. Vermijd duplicatie van code en werk met overerving!

2.4 Theoretische vragen

Werk in je **verslag** ook volgende theoretische opgaven uit:

- *Gegeven een lijst van n getallen. Wat is het **minimaal** en **maximaal** aantal toppen dat de suffixboom voor deze lijst kan hebben. Bewijs deze grenzen en geef voorbeelden waarbij deze grenzen bereikt worden.*
- *Voor sommige toepassingen is het mogelijk dat de referentielijst **circulair** is. Dit wil zeggen dat het laatste karakter van de lijst opgevolgd wordt door het eerste, enzovoort. Deze toepassing komt bijvoorbeeld voor bij het zoeken in bacteriële genoomsequenties die vaak circulair zijn. Beschrijf hoe de suffixboom en/of de zoekalgoritmen moeten worden aangepast zodat deellijsten kunnen worden gevonden in circulaire lijsten. De aanpassingen mogen de asymptotische tijd- en geheugencomplexiteiten van de originele methode niet wijzigen. Hierbij mag je ervan uitgaan dat er nooit deellijsten worden gezocht die langer zijn dan de referentielijst.*
  Beschrijf duidelijk je algoritme en toon aan dat het correct is en binnen de vereiste geheugen- en tijdscomplexiteit valt.

2.5 Overige

Presenteer je resultaten in overzichtelijke grafieken en bespreek ze ook grondig. Probeer al je resultaten te verklaren. Voorzie je code ook van commentaar en licht al je ontwerpsbeslissingen toe in het verslag. Hou je verslag beknopt, maar zorg ervoor dat alles toch grondig besproken wordt.

3 Indienen

Er moet verplicht tweemaal worden ingediend. Een eerste versie dient tussentijds elektronisch ingediend te worden **vóór vrijdag 2 november 2012 om 17u00**. Deze versie dient ten minste een constructie-algoritme voor één variant van een suffixboom te bevatten. Je uiteindelijke project dien je in **vóór maandag 26 november 2012 om 17u00**. Code en verslag worden elektronisch ingediend. Van het verslag verwachten we ook een **hardcopy**. Nadien zal je mondeling je werk verdedigen; het tijdstip hiervoor wordt later advalvas bekendgemaakt.

Elektronisch indienen Voor de automatische verbetering is het belangrijk dat je bestanden correct ingediend worden. Pak de elektronische bestanden precies in zoals beschreven. Je dient één zipfile in via `http://indianio.ugent.be` met de volgende inhoud:

- `src/` bevat alle broncode inclusief de ongewijzigde interfaces die gegeven werden. Behoud de package structuur en maak ook GEEN subpackages of subdirectories van `suffixtree` aan!
- `tests/` alle testcode. De testcode mag wel in een ander package zitten.
- `extra/verslag.pdf` de elektronische versie van je verslag. In de map `extra` kan je ook extra bijlagen plaatsen.

Algemene richtlijnen

- Zorg ervoor dat je code volledig compileert met een Sun Java 1.6 compiler. Extra libraries zijn niet toegestaan (je kan echter voor alle duidelijkheid wel gebruik maken van de standaard JRE 1.6 library). Voor de tests mag **wel** gebruik gemaakt worden van **JUnit 4.5**. Niet compileerbare code en projecten die niet correct verpakt werden **worden niet beoordeeld**.
- Schrijf efficiënte code maar ga niet overoptimaliseren: **geef voorrang aan elegante, goed leesbare code**. Kies zinvolle namen voor klassen, methoden, velden en variabelen (gebruik de correcte conventies) en voorzie voldoende documentatie.
- Het project wordt gequoteerd op **4** van de 20 te behalen punten voor dit vak, en deze punten worden ongewijzigd overgenomen naar de tweede examenperiode.
- Het is **strikt noodzakelijk** twee keer in te dienen: het niet indienen van de eerste tussentijdse versie betekent sowieso het verlies van alle te behalen punten voor het project.
- Projecten die ons niet bereiken voor de deadline worden niet meer verbeterd: dit betekent het verlies van alle te behalen punten voor het project.
- Dit is een individueel project en dient dus door jou persoonlijk gemaakt te worden. Het is toegestaan om andere studenten te helpen of om ideeën uit te wisselen, maar **het is ten strengste verboden code uit te wisselen of over te nemen van het internet**, op welke manier dan ook. Het overnemen van code beschouwen we als fraude (van **beide** betrokken partijen) en zal in overeenstemming met het examenreglement behandeld worden.
- Essentiële vragen worden **niet** meer beantwoord tijdens de laatste week voor de deadline.