transnationale Universiteit Limburg
School voor Informatietechnologie
$\sim$
Universiteit Hasselt

Sequentie Analyse

Thesis voorgedragen tot het behalen van de graad van licentiaat in de informatica / doctorandus in de kennistechnologie, afstudeervariant databases

door

Raf Nulens

Promotor: Prof. dr. Frank Neven
Begeleider: Dieter Van de Craen

Januari 2006

Inhoudsopgave

Hoofdstuk 1

Inleiding

Sequenties kunnen vergelijken en aligneren is van groot belang om de functionaliteit van genen te kunnen achterhalen. Dankzij het aligneren van sequenties door onder meer BLAST of FASTA heeft men veel bijgeleerd over de relaties tussen genomen onderling en de evolutie van genen. De meest gebruikte toepassing voor deze programma's is het vergelijken van een enkel gen, de query-sequentie, ten opzichte van een database van genen.
Bij het merendeel van de query's op een normale database worden indexen gebruikt om snel het antwoord op de query te kunnen vinden. BLAST werkt echter door de volledige sequentie-database lineair te scannen en delen van de query-sequentie te aligneren met stukken sequentie uit de database. Als men weet dat het aantal sequenties in deze databases exponentieel groeit, is het duidelijk dat er voor sequenties ook een soort index gemaakt moet worden.
In deze thesis bespreken we enkele datastructuren om sequenties te indexeren, de suffix tree en de suffix array. Er bestaan een aantal algoritmes om deze indexen te creëren. De belangrijkste doelstelling van deze thesis is na te gaan in hoeverre deze algoritmes geschikt zijn om van een volledig genoom de suffix tree of suffix array te construeren.
Naast de suffix tree en suffix array bespreken we ook de enhanced suffix array, een index die als basis een suffix array heeft maar dankzij extra tabellen voor de meeste toepassingen dezelfde tijdscomplexiteit heeft als een suffix tree. Voor elk van deze index-structuren zien we een aantal constructiealgoritmes en evalueren we enkele implementaties. In het laatste deel van deze thesis overlopen we enkele algoritmes om sequenties te aligneren. We tonen aan dat suffix arrays en suffix trees praktisch toepasbaar zijn, alignerингsprogramma's die van deze indexstructuren gebruik maken zijn in veel gevallen sneller dan BLAST.

Hoofdstuk 2

Suffix trees

2.1 Inleiding

Een suffix tree is een datastructuur die alle suffixen van een bepaalde sequentie S over een alfabet Σ indexeert. De belangrijkste toepassing van de suffix tree is dat het zoeken van alle locaties in S waar een substring s voorkomt, in lineaire tijd $|s|$ kan gebeuren. Het opbouwen van een suffix tree kan - zoals we in hoofdstuk 2.4 gaan aantonen - in $O(n)$ tijd kan gedaan worden, met n de lengte van de sequentie S.
Ook al kunnen suffix trees op belangrijke queries snel een antwoord vinden, toch worden ze nog niet vaak gebruikt in de praktijk. Een mogelijke oorzaak hiervoor is dat suffix trees - in tegenstelling tot normale indexen - meer geheugen innemen dan de data die ze indexeren. Verschillende onderzoekers hebben hiervoor een oplossing proberen te vinden door een suffix tree zo compact mogelijk voor te stellen. De suffix tree implementatie van Kurtz [GKS99] is tot nu toe één van de meest efficiënte qua geheugengebruik. Een overzicht van enkele methoden om een suffix tree op te slaan geven we in hoofdstuk 2.6.
Een tweede mogelijke verklaring dat suffix trees redelijk onpopulair zijn is dat de oorspronkelijke papers van Weiner [Wei73] en McCeight [McC76], waarin een algoritme gegeven wordt om een suffix tree in lineaire tijd te construeren, de reputatie hebben zeer moeilijk te begrijpen zijn. Meer recent heeft Ukkonen [Ukk92] een ander lineair algoritme gemaakt dat gemakkelijker uit te leggen is, en daardoor ook beter te implementeren. De meerderheid van de implementaties die een lineair algoritme gebruiken om de suffix trees te construeren gebruiken de methode van Ukkonen, daarom behandelen we in deze thesis enkel deze methode.
Een ander probleem is dat het vrij lang duurt om een suffix tree te construeren, zelfs indien de volledige suffix tree samen met de sequentie S in het werkgeheugen past. In de bio-informatica hoeft dit geen groot probleem te zijn, DNA-sequenties die men in een database opslaat zijn over het alge-

meen stabiel, ze worden niet snel aangepast. De constructietijd is dus niet het grootste issue.
Ook al bestaan er methodes om een suffix tree in lineaire tijd te construeren toch gebruiken sommige onderzoekers liever alternatieve, niet-lineaire methodes. Lineaire algoritmes lezen data op een semi-random manier uit het geheugen. Indien de suffix tree dan wegens plaatsgebrek voor een deel op de harde schijf moet worden opgeslagen gaat het algoritme zeer traag lopen, toegang tot de harde schijf is - in vergelijking met toegang tot het geheugen - een traag mechanisch proces. Er bestaan zeer eenvoudige niet-lineaire algoritmes die een suffix tree deel per deel construeren, waarbij stukken suffix tree in het werkgeheugen gemaakt worden en pas daarna naar de harde schijf worden geschreven. We zullen aantonen dat zij in de praktijk toch sneller kunnen zijn dan hun lineaire tegenhangers.
Een suffix tree is een veelzijdige index die meer in haar mars heeft dan op een snelle wijze subsequenties te localiseren. Een overzicht van de andere mogelijkheden van een suffix tree, toegespitst op bio-informatica, kan u vinden in hoofstuk 2.8.

2.2 Definities

Als we het in deze paper hebben over het construeren van een suffix tree voor een sequentie, wordt hiermee een biologische sequentie bedoeld over het alfabet `a`,`c`,`g`,`t`. Uiteraard kunnen suffix trees gebouwd worden voor elk willekeurig en eindig alfabet.

Definitie 2.2.1 (Suffix tree)**.** *Een suffix tree τ voor een sequentie S met lengte n bestaande uit tekens uit een alfabet Σ is een boom met exact n bladeren. Elke inwendige node heeft ten minste twee en ten hoogste $|\Sigma|$ kinderen. Het label van een node is een niet lege substring van S waarbij elke twee uitgaande bogen van een node niet met hetzelfde karakter kunnen beginnen.*

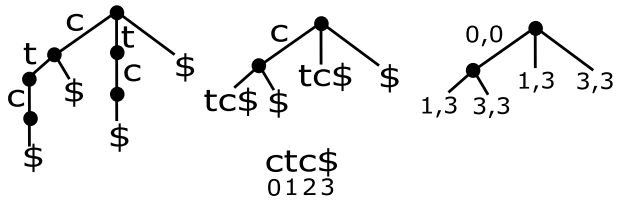
Met bovenstaande definitie is er echter nog een probleem. Neem als voorbeeld de sequentie `attcagttc`, de prefix `ttc` van suffix `ttcagttc` is precies hetzelfde als de suffix `ttc` . Als we van deze sequentie de suffix tree maken, dan heeft de suffix `ttc` geen blad, wat in tegenspraak is met de definitie. Om dit probleem op te lossen wordt er achteraan de sequentie een uniek symbool geplaatst wat niet in het oorspronkelijk alfabet voorkomt. In de literatuur wordt hiervoor meestal het karakter $ gebruikt. Op deze manier kan geen enkele prefix van een suffix gelijk zijn aan een andere suffix, en heeft bijgevolg elke suffix een overeenkomstig blad in de suffix tree. Merk op dat $ wel tot het alfabet Σ gerekend moet worden. Een suffix tree van een sequentie die niet eindigt op een uniek karakter noemt men ook wel een impliciete suffix tree.

De belangrijkste eigenschap van een suffix tree is dat voor elk blad i de concatenatie van alle boog-labels op het pad van de root tot het blad i de suffix van S is die begint op positie i. Deze suffix $S[i..n]$ schrijven we ook wel kortweg als suffix S_i. We noteren deze suffix in de thesis als $S[i..n]$. Het $path(\overline{v})$ is de concatenatie van alle edge-labels van de root van τ tot de node $\overline{v}$, $path(\overline{v})$ komt overeen met substring v. Elk $path(\overline{v})$ in de suffix tree is uiteraard uniek.
Omdat $n \geq 1$ en $S(1) \neq \$$ moet de root kinderen hebben, bijgevolg zijn alle interne nodes branching nodes. Dit betekent eveneens dat er maximaal n interne nodes kunnen zijn, die elk in constante space kunnen worden opgeslagen. Omwille van het toevoegen van het \$-teken heeft, wordt elke suffix voorgesteld door exact één blad in de suffix tree (en omgekeerd). Een suffix tree bevat dus $n + 1$ bladeren. Een suffix tree bevat dus ten hoogste $2n + 1$ nodes, waardoor een suffix tree $O(n)$ space inneemt.
Indien men van meerdere sequenties een suffix tree wil maken, kan men deze in één sequentie plaatsen door bijvoorbeeld het teken \$ tussen alle naburige sequenties te plaatsen. Uiteraard moet ook bij deze samengevoegde sequentie op het einde een uniek karakter worden bijgevoegd, hiervoor gebruikt men meestal het teken #. Merk op dat samenvoegen van de sequenties zonder scheidingsteken niet veel zin heeft, deze sequenties zijn fysiek gescheiden (bijvoorbeeld omdat ze op andere chromosomen liggen). Het is wel nuttig om voor q verschillende sequenties in plaats van q verschillende suffix trees te maken, één grote suffix tree te construeren. Indien bij een query toch al de sequenties doorzocht moeten worden is het efficiënter de grote suffix tree te doorzoeken dan alle aparte suffix trees.

Men moet een onderscheid maken tussen een suffix trie, een gewone suffix tree en een gecomprimeerde suffix tree. Het verschil tussen een suffix trie en een suffix tree is simpel, bij een suffix trie mag een boog slechts één karakter hebben, zodat niet elke inwendige node een branching node is. Bij een suffix tree kan een edge tussen twee branching nodes meer dan één karakter beslaan. Een gecomprimeerde suffix tree heeft als labels van de edges geen karakters meer, maar begin- en eindverwijzingen naar de posities van de karakters in de sequentie. We hebben in figuur 2.1 de verschillen op een rij gezet voor de sequentie `ctc$`. Links staat de suffix trie, in het midden de gewone suffix tree en rechts de gecomprimeerde suffix tree.

2.3 Naïeve implementatie

De meest logische manier om van een sequentie een suffix tree te maken is deze incrementeel op te bouwen. Bij de eerste stap bestaat de boom enkel uit een root en één blad, waarbij de boog tussen root tot blad de volledige sequentie $S[1..n]\$$ is. Daarna worden de andere suffixen $S[i..n]\$$ één voor



Figuur 2.1: Suffix trie, suffix tree en gecomprimeerde suffix tree

één bijgevoegd, met i gaande van 2 tot n. De suffix tree die alle suffixen van 1 tot i bevat noemen we τ_i.
Boom τ_{i+1} wordt uit boom τ_i opgebouwd door vanaf de root van τ_i het langste pad te zoeken dat een prefix is van $S[i+1..n]\$$. Dit pad kan gevonden worden door opeenvolgende labels van bogen tussen nodes te matchen met de karakters van de suffix $S[i+1..n]\$$, todat er geen match meer mogelijk is. Dit pad is uniek omdat er bij een node geen twee uitgaande bogen zijn met labels die met hetzelfde karakter beginnen. Op een gegeven moment moet er een mismatch zijn tussen de labels en $[i+1..n]\$$ omdat geen suffix van $S\$$ een prefix kan zijn van een andere suffix van $S\$$. Die mismatch kan gebeuren in het midden van een label, of aan een node zelf wanneer geen van de labels met het juiste karakter beginnen.
Indien de mismatch in het midden van een label voorkomt, het label van de boog tussen nodes u en v, dan moet die boog in twee worden gebroken en een nieuwe node w worden toegevoegd. Het label van de boog moet juist achter het laatste karakter dat matchte met een karakter in $S[i+1..n]\$$ gesplitst worden. De nieuwe boog tussen nodes u en w is gelabeld met het deel van de boog tussen u en v dat overeenkwam met $S[i+1..n]\$$, de boog tussen w en v is dan gelabeld met het overblijvende deel van de boog tussen u en v. Tot slot moet er een nieuwe boog worden gemaakt tussen node w en het nieuwe blad $i+1$, met als label het niet gematchte deel van suffix $S[i+1..n]\$$.
Indien men vanuit een node geen bogen heeft waarvan het eerste karakter van het label matcht met het volgende karakter van suffix $S[i+1..n]\$$, moet er een nieuwe boog tussen die node en een nieuw blad $i+1$ worden gelegd. Het label van deze boog wordt gevormd door de niet gematchte karakters van suffix $S[i+1..n]\$$.
De boom bevat nu een uniek pad van de root naar blad $i+1$, en dit pad heeft als label $S[i+1..n]\$$. De worst-case tijdscomplexiteit van dit algoritme bedraagt $O(n^2)$, waarbij verondersteld wordt dat de grootte van het alfabet constant is. Dit is gemakkelijk in te zien wanneer we een suffix tree maken van een sequentie zoals `aaaaaaaag`. Telkens moet er een path van de root tot de plaats van de nieuwe node worden afgelopen, in dit voorbeeld is te zien dat er gemiddeld $n/2$ symbolen per suffix worden doorlopen. Er worden n suffixen toegevoegd zodat de tijdscomplexiteit op $O(n^2)$ uitkomt. Indien we suffixen invoegen in een boom die in evenwicht is gaat de average

tijdscomplexiteit $O(n \log n)$ benaderen.

2.4 Lineaire constructie

In dit hoofdstuk presenteren we het algoritme van Ukkonen om een suffix tree in lineaire tijd te construeren [Ukk92]. Ukkonen was niet de eerste die met zulk een algoritme op de proppen kwam, Weiner [Wei73] ontwikkelde in 1973 al het eerste lineaire suffix tree constructiealgoritme. Later, in 1976, publiceerde McCreight [McC76] zijn ietwat efficiënter algoritme. Zowel de methode van McCreight als Weiner construeren de suffix tree van rechts naar links, dus door de kleinste suffix eerst toe te voegen. Het algoritme van Ukkonen werkt door de suffixen van links naar rechts toe te voegen, de methode is ongeveer even efficiënt als het algoritme van McCreight, maar heel wat makkelijker om te begrijpen. Hier beschrijven we Ukkonens algoritme door te beginnen van een inefficiënte implementatie die stap voor stap verbeterd wordt totdat het constructiealgoritme in worst-case tijdcomplexiteit slechts lineaire tijd nodig heeft. Dit hoofdstuk is voor een groot deel gebaseerd op hoofdstuk 6 van het boek *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology* van Gusfield [Gus97], aangevuld met voorbeelden.

2.4.1 Impliciete suffix trees

Een impliciete suffix tree van een string S met lengte n is een suffix tree waarvan het laatste karakter niet noodzakelijk uniek is, wat tot gevolg heeft dat niet elke suffix wordt voorgesteld door een blad. Sommige suffixen kunnen hierdoor eindigen in een node. De impliciete suffix tree van de string S noteren we als I. Als men een gewone suffix tree heeft van string $S\$$ dan kan men een impliciete suffix tree bekomen door alle labels aan de bladeren het symbool \$ weg te laten. Hierdoor kan het gebeuren dat sommige bladeren geen labels meer hebben, deze moeten ook verwijderd worden en vervolgens moeten alle nodes die nog slechts één kind hebben verwijderd worden.

2.4.2 High-level beschrijving

Het algoritme werkt incrementeel in n fasen, in fase $i+1$ wordt uit suffix tree I_i tree I_{i+1} gemaakt, waarbij de impliciete suffix tree I_i overeen komt met de prefix $S[1..i]$. Tijdens elke fase $i+1$ gebeuren dan ook nog $j = 1, \ldots, i$ updates van de suffix tree, elke extentie j zorgt ervoor dat suffix $S[j...i+1]$ in I_{i+1} zit. Dus in fase $i+1$ wordt substring $S[1...i+1]$ eerst in de boom geplaatst, gevolgd door $S[2...i+1]$, $S[3...i+1]$, $\ldots$, $S[i+1]$.

Schematisch kan het algoritme als volgt worden weergegeven:

Algoritme 1. *Lineair suffix tree constructiealgoritme van Ukkonen*

```
Construeer boom I_1
for i from 1 to n − 1 do {
   begin phase i + 1;
   for j from 1 to i {
     begin update j;
     Zoek het einde van het path S[j..i] in de huidige boom,
      vertrekkende van de root. Verleng dat path door het
      karakter S[i + 1] er aan toe te voegen, zodat de boom
      nu string S[j...i + 1] bevat.
   }
}
```

Drie update regels

Nu is de vraag natuurlijk hoe path $\beta = S[j..i]$ verlengd kan worden tot $\beta S[i+1]$. Er zijn drie mogelijkheden:

1. Indien het path β in de huidige tree eindigt op een blad wordt het karakter $S[i+1]$ gewoon toegevoegd aan het einde van het label op die blad-edge.

2. Als er een path β in de suffix tree zit, dat verder gaat met een karakter $q \neq S[i+1]$ kunnen er twee zaken gebeuren. Ofwel eindigt path β op een node, in dat geval moet een blad-node worden toegevoegd met als label $S[i+1]$. Ofwel eindigt het path β midden in een edge, zodat die edge moet gesplitst worden. Op die plaats moet een node worden toegevoegd, het path van de node is dan β, de uitgaande edges van de node leiden dan naar een nieuw blad met edge-label $S[i+1]$ en naar de node onder de gesplitte edge.

3. Soms is het path $\beta S[i+1]$ al aanwezig in de boom. Omdat we te maken hebben met impliciete suffix trees moet niet elke suffix worden voorgesteld door een blad, daarom verandert er in dit geval niets.

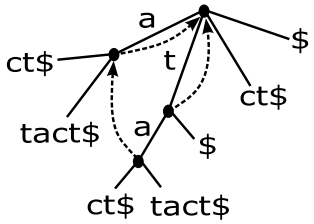
Elk van de hierboven beschreven regels zijn uit te voeren in constante tijd, van zodra het einde van de suffix $S[1..i]$ in de tree gevonden is. We kunnen het einde van elke suffix β in $O(|\beta|)$ tijd vinden door vanaf de root van de tree te zoeken. Door deze benadering kan update j van fase $i+1$ in $O(i+1-j)$ tijd gedaan worden, I_{i+1} kan dan uit I_i in $O(n^2)$ tijd geconstrueerd worden en bijgevolg neemt het maken van I_n dan $O(n^3)$ tijd in beslag. Deze tijdscomplexiteit van $O(n^3)$ kan worden verbeterd tot $O(n)$ door een aantal observaties en verbeteringen in de implementatie. Elk apart gaan die observaties de tijdscomplexiteit niet verbeteren, maar door ze tegelijk te gebruiken wordt het algoritme van Ukkonen lineair.

2.4.3 Suffix links

Om de tijdscomplexiteit van kubisch naar kwadratisch te verminderen moet bij elke update de juiste positie in de suffix tree in constante tijd worden gevonden. Om dit te bereiken definiëren we het begrip *suffix link*.

Definitie 2.4.1 (Suffix link)**.** *Voor elke interne node v van een suffix tree met als path $x\alpha$, waarbij $x \in \Sigma$ en $\alpha \in \Sigma^*$, definieer $s(v)$ als de node met als path α. Een pointer van v naar $s(v)$ is dan de suffix link van v.*

We verduidelijken dit begrip aan de hand van de suffix tree van de sequentie `tatact$`, in figuur 2.2. Deze suffix tree bevat 3 suffix links, waarvan twee uitkomen in de root. Bij deze twee suffix links heeft het path α lengte nul. Merk ook op dat de suffix links van een suffix tree op zich een boom vormen, vanuit een willekeurige node kan men suffix links volgen tot aan de root.



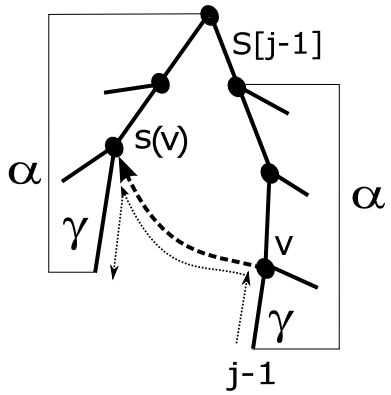
Figuur 2.2: De suffix link

Suffix links kunnen als volgt gebruikt worden. Bij update j van phase $i+1$ wordt het einde van het path $S[j..i]$ gevonden, dit wordt uitgebreid met karakter $S[i+1]$. Update $j+1$ wordt op een gelijkaardige manier gedaan, eerst moet het einde van path $S[j+1..i]$ gezocht worden en daarna uitgebreid. Stel dat v een interne node is met path $S[j]\alpha$ dat een prefix is van substring $S[j..i]$ en $s(v)$ een node is met path α. Door de suffix link te volgen vanuit v naar $s(v)$ kunnen we het α-deel van het path $S[j+1..i]$ afsnijden. Vanuit node $s(v)$ wordt dan het path verder gevolgd om het op het einde te verlengen met karakter $S[i+1]$.
Nu moet nog worden aangetoond dat suffix links altijd bestaan en ze gebruikt kunnen worden wanneer ze nodig zijn. Daarvoor wordt de volgende observatie gebruikt: Als een interne node v gecreëerd wordt tijdens update j in phase $i+1$, dan bestaat node $s(v)$ zeker na de volgende update $j+1$. Laat het path $x\alpha$ eindigen in node v, node v kan enkel gemaakt zijn aan het einde van path $S[j..i]$. Vanuit v moet dus nog een ander path verder lopen beginnende met het karakter $c \neq S[i+1]$. De paden $x\alpha c$ en $x\alpha$ zijn dus al toegevoegd voor phase $i+1$. In update $j+1$ bestaat node $s(v)$ dus al of wordt deze node toegevoegd aan het einde van het path α. Het is niet onmogelijk dat node $s(v)$ al bestond voor phase $i+1$, mogelijk was er al eerder een string met als prefix αd met $d \neq c$ toegevoegd.

In Ukkonens algoritme heeft elke nieuwe interne node een inkomende suffix link pointer na de volgende update. Bijgevolg geldt voor elke impliciete suffix tree I_i dat, indien een interne node v een path-label $x\alpha$ heeft, er een node $s(v)$ in I_i bestaat met path-label α. In elke phase wordt uit een suffix tree I_i suffix tree I_{i+1} geconstrueerd, aan de suffix tree I_i worden sequenties $S[1..i+1]$, $S[2..i+1]$, ..., $S[i+1]$ toegevoegd. Stel nu dat in update j een node v wordt gecreëerd, dan wordt de node $s(v)$ gevonden of gemaakt in update $j+1$ en de suffix link van v naar $s(v)$ wordt gelegd. Als de node $s(v)$ al bestond dan was deze toegevoegd in een eerdere phase, maar door de constructie heeft deze dan al een suffix link naar een node $s(s(v))$. Zoals we in de definitie hebben gezien komt elke ketting van suffix links uit in de root.
Stel dat we in phase i zitten tijdens de eerste update, dus string $S[1..i] = S[1]\alpha$ wordt verlengd met karakter $S[i+1]$. Laat v de laatste inwendige node zijn op het path $S[1]\alpha$, met α mogelijk leeg. Omdat string $S[1..i+1]$ de voorlopig langst toegevoegde string is weten we dat het einde van het path niet uitkomt in een reeds bestaande node met een suffix link. Nadat we $S[i+1]$ hebben toegevoegd aan path $S[1..i]$, moeten we nu op een efficiënte manier het einde van de string $S[2..i]$ vinden in de huidige tree I_i. Vanuit het einde van het path $S[1..i+1]$ stijgen we één label tot aan node v. Node v is ofwel de root, ofwel een interne node. Indien het de root is, moet het path $[2..i]$ worden afgelopen zoals in het naïve algoritme. Indien v een interne node is, heeft het wegens de constructie een suffix link naar node $s(v)$. Wegens de definitie van een suffix link is het path van $s(v)$ een prefix van α, dus moet het einde van string $S[2..i]$ gevonden worden in de subtree onder $s(v)$. Bijgevolg moet het algoritme het path verder aflopen vanaf node $s(v)$, in plaats vanaf de root.
Om aan een string $S[j..i]$, met $j > 2$, het karakter $S[i+1]$ toe te voegen moet ongeveer dezelfde taktiek gehanteerd worden. Startend aan het einde van string $S[j-1..i+1] = S[j-1]\alpha$ in de huidige tree, moet er hooguit één label naar boven worden gegaan tot men aan een node komt die een suffix link heeft. De eerste node met een suffix link die men tegenkomt is ofwel een inwendige node v, ofwel de root. Laat γ het label van de edge zijn tussen het einde van string $S[j-1..i]$ en de bovenliggende node. Als de suffix link van v naar node $s(v)$ gevolgd wordt, moet vanaf deze laatste het path γ gevolgd worden tot het einde van $S[j..i]$. Het volledige path is dan gelijk aan α. Uiteindelijk moet karakter $S[i+1]$ worden toegevoegd volgens de update regels. Zie figuur 2.3 voor een verduidelijking.

2.4.4 Skip/Count truc

String $S[j..i]$ is in update j reeds aan de boom toegevoegd, men kan dit path volgen door de karakters van elke edge te onderzoeken, maar door het juist kiezen van de edges moet niet elk karakter bekeken worden. Laat $S[k]$ het



Figuur 2.3: De juiste plaats localiseren voor een update

volgende karakter zijn dat op het path $S[j..i]$ gematcht moet worden, een edge met als label $S[p..q]$ kan doorlopen worden door enkel te controleren dat $S[p] = S[k]$. De rest van de karakters $S[p+1..q]$ kunnen dan gemakkelijk worden overgeslagen. Indien path $S[j..i]$ niet meer genoeg karakters heeft om te matchen met het volledige label $S[p..q]$, kan men onmiddellijk gaan naar karakter $S[i]$ in label $S[p..q]$ om daar karakter $S[i+1]$ toe te voegen in de boom. Nogmaals, het is dankzij de vaste volgorde waarmee het algoritme strings verlengt dat we volledige labels kunnen overslaan omwille van hun eerste karakter. Als we string $S[j..i]$ willen verlengen zit deze al in de boom en kan het dus niet voorkomen dat er een edge moet gesplitst worden voor we het hele path $S[j..i]$ hebben afgelopen. Het gevolg is dat het aflopen van een path in even veel stappen kan als het aantal nodes op dat path, in plaats van de lengte van de overeenkomstige labels.
Het aantal nodes op een path van de root tot een node u noemen we de node-depth van u. Voor elke node v met een suffix link $s(v)$ geldt dat de node-depth van v ten hoogste één groter is dan de node depth van $s(v)$. Een suffix link volgen heeft dus als gevolg dat men hooguit één niveau dichter bij de root komt. Elke interne node die een voorouder is van v en een path $x\beta$ heeft, heeft een suffix link naar een node met path β. Omdat $x\beta$ een prefix is van het path naar v volgt hieruit dat β een prefix is van het path naar $s(v)$ en bijgevolg gaat elke suffix link die vertrekt vanuit een voorouder van v naar een andere voorouder van $s(v)$. De node depth voor $s(v)$ is ten minste één (de root) plus het aantal interne voorouders van v die path-labels hebben van meer dan één karakter lang (indien het path één karakter lang is wijst de suffix link naar de root). Daarom kan v dus een node-depth hebben die ten hoogste één groter is dan $s(v)$.

Deze truc heeft - samen met de suffix links - belangrijke gevolgen voor de tijdscomplexiteit van het algoritme. Elke phase bedraagt nu nog slechts

$O(n)$ tijd. Er zijn $i+1 \leq n+1$ updates in phase $i+1$, binnen een update gaat het algoritme ten hoogste één node naar boven in de tree, volgt de suffix link, gaat een aantal nodes naar beneden en voert de update uit. De update op zich gebeurt in constante tijd, de vraag is hoeveel nodes er naar beneden gewandeld moeten worden. Het volgen van de suffix link kan de node-depth verminderen met één. Het is belangrijk om in te zien dat de node-depth in een tree nooit groter kan zijn dan n, een extreem voorbeeld is de string `aaaaaaat`, de lengte van de string is hier gelijk aan de maximale node-depth. Aangezien per update de node-depth slechts 2 keer verminderd kan worden en de maximale node-depth n bedraagt, betekent dit dat de bovengrens voor het aantal verhogingen van de node-depth tijdens een phase ten hoogste $3n$ bedraagt.
Door het gebruik van de *skip/count* truc is de tijd om een edge af te lopen constant, zodat de totale tijd voor een phase gelijk is aan $O(n)$. Omdat er n phases zijn bedraagt de tijdscomplexiteit voor het algoritme op dit punt $O(n^2)$, wat even snel is als de naïve methode. Door nog een paar optimalisaties door te voeren kan deze tijdscomplexiteit verlaagd worden tot $O(n)$.

2.4.5 Drie extra observaties

De tot dusver gebruikte voorstelling van de suffix tree gebruikt $\Theta(n^2)$ space. Het is onmogelijk om een $O(n)$ algoritme te maken dat $\Theta(n^2)$ space gebruikt, dus er moet wel iets veranderd worden aan de manier waarop de suffix tree wordt opgeslagen.

Edge-label compression

In plaats van bij elke edge expliciet de substring te schrijven, worden enkel index-paren bijgehouden op die edge. Een index-paar duidt het begin en eindpunt aan van de substring in S. Omdat het algoritme een kopie heeft van de string S, kan - gegeven de positie - in constante tijd worden opgezocht uit welke karakters die edge bestaat. Het aantal edges in een suffix tree is ten hoogste $2n-1$, als er twee integers per edge worden gebruikt heeft de suffix tree slechts $O(n)$ space nodig. We hebben deze voorstelling al gedefiniëerd in hoofdstuk 2.2 onder de naam van een gecomprimeerde suffix tree. We herhalen hier deze voorstelling omdat ze cruciaal is voor het algoritme van Ukkonen.

Regel 3 is een no-op

Als suffix update regel 3 wordt toegepast in update j, dan zal deze regel ook worden toegepast als we de volgende substringen $S[j+1..i+1]$, $S[j+2..i+1]$, $\ldots$, $S[i+1]$ invoegen. De rest van de updates in phase $i+1$ worden impliciet gedaan nadat er een update j is gebeurt waarbij regel 3 werd toegepast.

Eens een blad altijd een blad

Als een blad wordt gecreëerd blijft het altijd een blad. Ten eerste omdat er geen update regel bestaat die kind-nodes aan een blad node hangt, ten tweede omdat wanneer update j een blad toevoegd, bij de updates j in de volgende phases regel 1 wordt toegepast. In elke volgende phase $i+1$ wordt het karakter $S[i+1]$ toegevoegd aan het edge-label van dat blad.
Noem de laatste update van phase i waarbij regel 1 of 2 wordt toegepast j_i. We weten dat van zodra er via regel 2 een blad is gemaakt, de updates j in de volgende phases regel 1 gaan toepassen. Dus van zodra regel 2 wordt toegepast gaat in de volgende phases regel 1 worden toegepast in de overeenkomstige update. Hieruit volgt rechtstreeks dat $j_i \leq j_{i+1}$. Deze observatie suggereert dat in phase $i+1$ alle expliciete updates van 1 tot j_i impliciet kunnen gebeuren.
Wanneer in phase $i+1$ een blad wordt gecreëerd zou het normaal gezien als label de substring $S[p..i+1]$ krijgen. In plaats van de edge de indices $(p, i+1)$ te geven kan men indices (p, e) gebruiken. De e staat symbool voor het huidige einde van de suffix, in elke phase $i+1$ krijgt symbool e als waarde $i+1$.
Als conclusie kunnen we stellen dat in phase $i+1$ expliciete updates enkel nodig zijn tussen update j_i+1 tot aan de eerste update waar regel 3 van toepassing is, of totdat de laatste update $i+1$ van die phase gebeurd is. Alle andere updates kunnen impliciet in constante tijd gedaan worden.
Per phase kan nu het volgende algoritme worden gebruikt:

1. Set index $e := i+1$.
2. Bereken updates $j_i+1,\ldots,j^*$ totdat $j^* > i+1$ of regel 3 werd toegepast in update j^*.
3. Set $j_{i+1} := j^* - 1$.

Door gebruik te maken van suffix links en enkele implementatie-trucs is Ukkonens algoritme in staat een impliciete suffix tree te bouwen in $O(n)$ tijd.
De updates die expliciet worden berekend in twee opeenvolgende phases i en $i+1$ gebeuren op een verschillende positie j, behalve voor update j^*. Deze update kan opnieuw berekend worden in de volgende phase. Deze tweede berekening van update j^* kan gebeuren in constante tijd door het einde van het path te onthouden na de vorige berekening. Stel $\overline{j} = 1, \ldots, n+1$ is de index van de huidige update, dan kan deze index nooit verlaagt worden over de phases $2, \ldots, n+1$, maar ze kan wel dezelfde blijven. Dus er worden hooguit $2n$ updates expliciet berekend.

2.4.6 De volledige suffix tree

Het algoritme maakt een reeks impliciete suffix trees I_1 tot I_n. Deze laatste moet nog worden omgevormd naar een echte suffix tree in $O(n)$ tijd. Eerst moet er een uniek eind-teken \$ worden toegevoegd aan de sequentie S. Ukkonens algoritme moet dan nog een extra phase uitvoeren, zodat er nu geen enkele suffix een prefix kan zijn van een andere suffix. Elke suffix in de suffix tree wordt nu expliciet voorgesteld door een blad. De enige andere verandering die moet gebeuren is om elke index e op elke blad-edge om te zetten in het getal n. Dit kan uiteraard gebeuren in $O(n)$ tijd door de boom éénmaal te doorlopen. Het resultaat van deze aanpassingen is een suffix tree, gebouwd in $O(n)$ tijd.

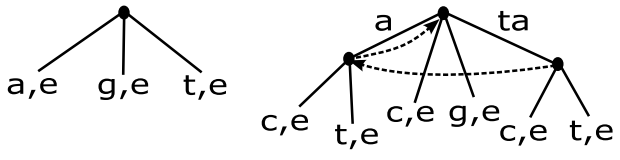
Een voorbeeld

We overlopen het algoritme van Ukkonen aan de hand van de sequentie `gtatacta$`. In tabel 2.1Per phase voeren we alle updates uit, we zeggen erbij of ze al dan niet impliciet worden uitgevoerd. De meest rechtse kolom geeft het nummer van de update aan. Een pijl duidt de laatste expliciete update j^* die wordt uitgevoerd in die phase, alle updates $< j^*$ in de volgende phase worden impliciet uitgevoerd.
In phase 6 wordt de eerste suffix link gelegd die niet uitkomt in de root, we hebben de impliciete suffix tree voorgesteld in figuur 2.4, de linkse boom is deze na phase 5, de rechtse na phase 6. Eerst wordt tijdens phase 6 in update 4 de substring `tac` toegevoegd in de boom, waarbij een nieuwe node $\overline{ta}$ wordt gecreëerd. De volgende update voegt substrinc `ac` toe, vanuit node $\overline{ta}$ bestaat nog geen suffix link, dus gaan we 1 node naar boven en komen zo uit in de root. Vandaar wordt het path dat begint met `a` afgelopen, na het karakter `a` is er een mismatch, op die plaats wordt het label gesplitst en node $\overline{a}$ wordt ingevoegd. Uit update 4 hebben we de positie van node $\overline{ta}$ bijgehouden, nu kunnen we de suffix link leggen van node $\overline{ta}$ naar $\overline{a}$. In update 6 wordt er nog een suffix link gelegd van node $\overline{a}$ naar de root.
De suffix link van node $\overline{ta}$ naar $\overline{a}$ wordt voor het eerst gebruikt tijdens update 8 in phase 9. Nadat substring `ta$` in update 7 is toegevoegd kunnen we vanuit node $\overline{ta}$ de suffix link volgen naar $\overline{a}$, om daar de substring `a$` in te voegen. Tijdens het uitvoeren van het algoritme van Ukkonen wordt de path-lengte bijgehouden, om zo te weten hoeveel karakters van een substring reeds gematcht zijn. Dus in node $\overline{ta}$ weten we dat de path-lengte 2 bedraagt, indien de suffix link naar node $\overline{a}$ gevolgd wordt vermindert de path-lengte met 1. Hierdoor kan het eerste karakter van substring `a$` genegeerd worden.

In hoofdstuk 2.7.2 beschouwen we een systeem, TOP-Q, dat gebruik maakt van het algoritme van Ukkonen. We vergelijken TOP-Q met TDD, wat een niet-lineair constructie algoritme gebruikt in hoofdstuk 2.7.3.

phase 1	g	regel 2	1	←
phase 2	gt	regel 1	1	
	g	regel 2	2	←
phase 3	gta	regel 1 (impliciet)	1	
	ta	regel 1	2	
	a	regel 2	3	←
phase 4	gtat, tat	regel 1 (impliciet)	1, 2	
	at	regel 1	3	
	t	regel 3	4	←
phase 5	gtata, ..., ata	regel 1 (impliciet)	1, 2, 3	
	ta	regel 3	4	←
	a	regel 3 (impliciet)	5	
phase 6	gtatac, ..., atac	regel 1 (impliciet)	1, 2, 3	
	tac	regel 2	4	
	ac	regel 2	5	
	c	regel 2	6	←
phase 7	gtatact, ..., act	regel 1 (impliciet)	1, 2, 3, 4, 5	
	ct	regel 1	6	
	t	regel 3	7	←
phase 8	gtatacta, ..., cta	regel 1 (impliciet)	1, 2, 3, 4, 5, 6	
	ta	regel 3	7	←
	a	regel 3 (impliciet)	8	
phase 9	gtatacta$, ..., cta$	regel 1 (impliciet)	1, 2, 3, 4, 5, 6	
	ta$	regel 2	7	
	a$	regel 2	8	
	$	regel 2	9	//

Tabel 2.1: Berekening van de suffix tree voor sequentie `gtatacta$`



Figuur 2.4: Van update 4 in phase 6 naar update 5

2.5 Incrementele opbouw van suffix trees

In het vorige hoofdstuk zijn suffix links voor het eerst gedefinieerd, hun functie is om snel van de ene plaats naar de andere in de suffix tree te springen. Suffix trees zijn vrij kwistig met geheugen, een normale suffix tree neemt acht tot twaalf keer meer plaats in dan de sequentie die ze indexeert. Als we een suffix tree willen construeren voor een grotere sequentie past de suffix tree al snel niet meer in het werkgeheugen, daarom moeten (onafgewerkte) delen van de suffix tree worden weggeschreven naar de harde schijf. Het volgen van een suffix link die naar een node verwijst die op de harde schijf staat, kost zeer veel tijd. Een optimale voor de hand liggende manier om te voorspellen waar suffix links naar wijzen bestaat niet. Bij het volgen van de suffix links verplaatst men zich op een horizontale manier door de boom, terwijl een path volgen van root naar een bepaalde node op de boom vertikaal doorkruist.
Om deze redenen wordt door verschillende onderzoekers geen gebruik gemaakt van suffix links bij het construeren van de suffix tree. Een mogelijkheid bestaat erin de sequentie meerdere keren te overlopen en telkens een deelboom van de suffix tree op te bouwen. Hierdoor wordt de lineaire tijdscomplexiteit vervangen door een $O(pn^2)$ tijdcomplexiteit in worst case, met p het aantal partities. De average case tijdscomplexiteit hangt in grote mate af van hoe random de karakters verdeeld zijn in een sequentie. Indien er alle karakters ongeveer even veel voorkomen en random verspreid zijn, gaat deze tijdscomplexiteit $\Theta(m \log m)$ benaderen.

In [HAI01] wordt de suffix tree opgesplitst in partities, een partitie bestaat uit suffixen die allemaal een prefix hebben van een bepaalde lengte. Suffixen die beginnen met eenzelfde prefix zitten uiteraard allemaal in dezelfde subtree. Het aantal partities p en dus de lengte van de prefix hangt af van de hoeveelheid geheugen die men verwacht om de volledige suffix tree voor te stellen, Smm genaamd, en de hoeveelheid main memory Amm. Het aantal partities bedraagt dus:

$$\left\lceil \frac{Smm}{Amm} \right\rceil$$

Voor elke partitie p_w worden alle suffixen van de sequentie gescand. Alle sequenties met die bepaalde prefix w worden in de partitie p_w ingevoegd. Nadat een partitie volledig is wordt deze opgeslagen in secundair geheugen. Het is belangrijk om in te zien dat de volledige sequentie in het werkgeheugen blijft en dat ook elk van de subtrees afzonderlijk in het main memory moeten passen. Anders faalt het algoritme.

Het belangrijkste punt van kritiek op deze werkwijze is dat DNA niet altijd pseudo-random verdeeld is en dat hierdoor bepaalde subtrees veel groter zijn

dan andere. Dit heeft tot gevolg dat de verwachte tijd dat het algoritme nodig heeft groter gaat zijn dan $\Theta(n \log n)$. Doordat bepaalde subtrees erg groot zijn, en elke subtree even ver van de root staat, groeit het aantal partities lineair als de sequentie langer wordt. Per subtree moet de volledige sequentie éénmaal doorlopen worden. De combinatie van deze twee gegevens geeft dat de tijd dat het algoritme in de praktijk nodig heeft eerder $O(n^2)$ benadert.
De oplossing voor dit probleem bestaat er in om in plaats van subtrees te creëren die allemaal een even lange prefix $|w|$ hebben, de suffix tree dynamisch op te splitsen [Bro04]. Eerst worden partities gemaakt met een vaste prefix lengte $|w|$. Indien het aantal suffixen binnen die partitie te groot is om een subtree te bouwen die in het geheugen past, wordt die partitie verder opgesplitst, totdat er enkel blokken overblijven die wel in het geheugen passen. Soms komt het voor dat wanneer van een partitie p_w de subtree is gemaakt, er nog plaats in het geheugen overblijft om van een andere partitie de volledige subtree te bouwen. In dat geval wordt de subtree van p_w pas naar disk geschreven als de andere subtree ook is geconstrueerd. Nadat alle subtrees op de harde schijf staan wordt van alle prefixen van die subtrees de stam van de uiteindelijke suffix tree gemaakt.
Het algoritme blijft uiteraard worst case tijdscomplexiteit $O(n^2)$ hebben, maar omdat de sequentie minder moet worden doorlopen gaat dit volgens de auteurs in de praktijk een average case tijdscomplexiteit opleveren van $\Theta(n \log n)$.

2.5.1 Samenvoegen van suffix trees

Dikwijls wordt een sequentie gezocht in een hele set van database-sequenties. Men kan dan een suffix tree construeren voor elke van die sequenties in de database, maar bij een query moeten al die suffix trees worden doorzocht om te zien of ze de gevraagde substring bevatten. Het is heel wat efficiënter om alle suffix trees samen te voegen om zo het zoekproces te versnellen. In dit hoofstuk wordt een manier geschetst om suffix trees samen te voegen tot één geheel.

De meeste onderzoekers gaan er van uit dat de sequentie waarvan een suffix tree moet gebouwd worden in het geheugen past. In [TTHP05] wordt beschreven hoe men door de sequentie in verschillende sukken kan delen en per deel de suffix tree kan construeren. De verschillende suffix trees worden uiteindelijk samengevoegd tot één enkele boom. Zij vergelijken hun algoritme met sort-merge, dat ook in verschillende fasen werkt. Bij sort-merge worden de elementen verdeeld in sets waarna elke set in het geheugen wordt gesorteerd. Daarna worden de gesorteerde subsets in één of meer fasen samengevoegd.
Het algoritme dat door de auteurs beschreven wordt heet ST-Merge. In

principe kunnen de suffixen puur random in groepen worden ingedeeld, maar het is logisch om opeenvolgende suffixen te nemen. Daardoor moet niet de hele sequentie worden ingeladen in het geheugen maar enkel het deel waar de startposities van de suffixen zich bevinden, samen met een beperkt aantal symbolen rechts van het begin van de laatste suffix.
Van elk van die groepen wordt de suffix tree gemaakt, bijvoorbeeld door gebruik te maken van het TDD algoritme van dezelfde auteurs, wat in hoofdstuk 2.7.1 wordt besproken. De aparte suffix trees kunnen allemaal in één keer worden samengevoegd, of in fasen waarbij intermediaire suffix trees worden gecreëerd. Het algoritme zelf bestaat uit twee functies, EdgeMerge en NodeMerge. De functies roepen elkaar recursief op totdat de verschillende bomen zijn samengevoegd.

Werking van het algoritme

Het merge algoritme begint met de verschillende root nodes om te vormen tot de root van de samengevoegde boom. Dit gebeurt door de functie NodeMerge op te roepen. EdgeMerge wordt gebruikt door NodeMerge als er verschillende nodes met uitgaande edges die een gemeenschappelijke prefix hebben, moeten worden samengevoegd.
De NodeMerge functie krijgt als parameters een NodeSet en een ParentEdge mee. Het moet de nodes samenvoegen en onder de ParentEdge plaatsen in de samengestelde suffix tree. Eerst worden alle uitgaande edges van de nodes uit de NodeSet gegroepeerd volgens hun eerste symbool. Dus er ontstaan ten hoogste $|\Sigma|$ groepen, met $|\Sigma|$ de grootte van het alfabet. Omdat de edges van elke node al gesorteerd zijn kan gemakkelijk het count-sort algoritme worden gebruikt. Vervolgens bekijkt het algoritme elke groep van edges, indien er in een groep slechts één edge zit betekent dit dat in de andere suffix trees geen suffixen zitten die dezelfde prefix hebben als de suffixen in deze subtree. Deze subtree kan daarom rechtstreeks in de samengevoegde boom worden gekopiëerd. Indien een groep meer dan één edge bevat, creëert het algoritme een nieuwe uitgaande edge voor de samengevoegde node.
EdgeMerge voegt verschillende edges bij elkaar die beginnen met hetzelfde symbool. Als parameters krijgt het een EdgeSet en een ParentNode mee. Eerst zoekt het de langste gemeenschappelijke prefix (*lgp*) van de set edges en maakt een nieuwe edge in de samengevoegde boom die als label de lgp krijgt. Als sommige edges langere labels hebben dan de lgp, worden die edges gesplitst door een node toe te voegen na de lgp. Alle nodes die eindigen op de lgp kunnen nu worden samengevoegd door NodeMerge aan te roepen, ze eindigen immers allemaal op een edge met hetzelfde label.

Algoritme 2. *NodeMerge(NodeSet, ParentEdge)*

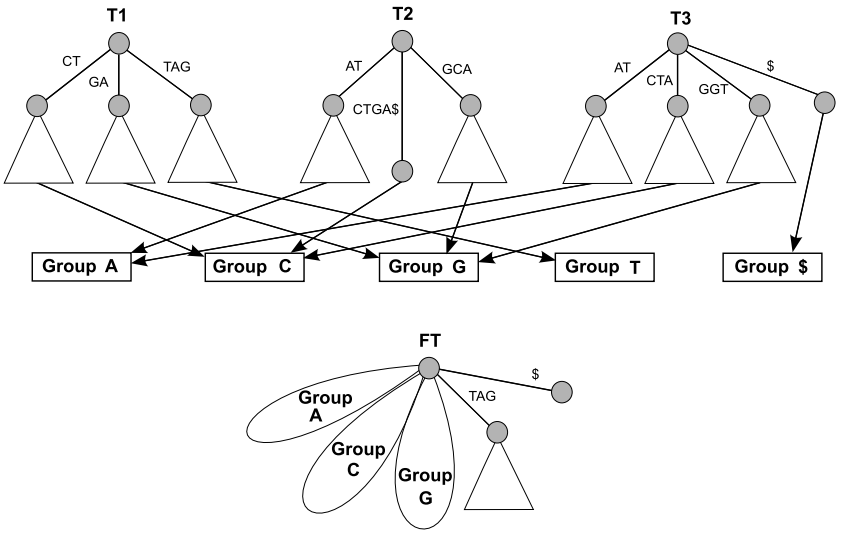
```
If (ParentEdge == NULL)
   Maak nieuwe node N voor samengestelde boom
Else
   Maak nieuwe node N op eind van ParentEdge
Groepeer de nodes in de NodeSet volgens eerste karakter
For each edge group
   If (group.size() == 1)
     Kopieer edge en subtree naar node N van de samengestelde boom
   Else
     EdgeSet := edges in de edge group
     EdgeMerge(EdgeSet, N)
```

Algoritme 3. *EdgeMerge(EdgeSet, ParentNode)*

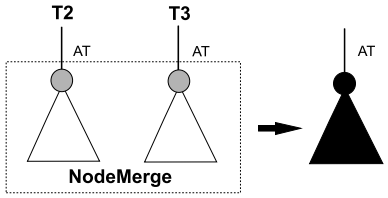
```
L := langste gemeenschappelijke prefix (lgp) van edges in EdgeSet
Maak uitgaande edge vanuit ParentNode met label L
NodeSet := {}
For each edge in EdgeSet
   If lgp < edge label
     Maak nieuwe node in boom door edge te breken na lgp
     Nieuwe node aan NodeSet toevoegen
   else
     Voeg node op einde edge toe aan NodeSet

NodeMerge(NodeSet, E)
```

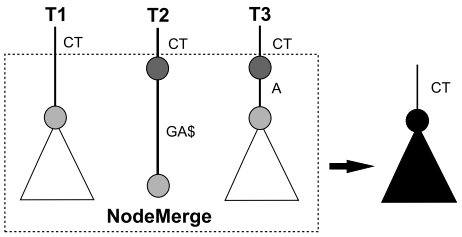
We werken nog een voorbeeld uit van enkele suffix trees die worden samengevoegd.



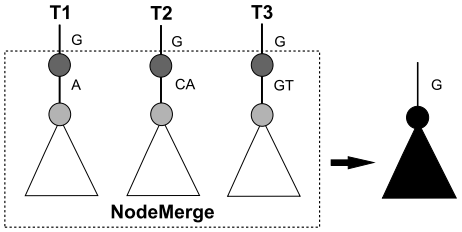
Figuur 2.5: We noemen de drie suffix trees T1, T2 en T3, de uiteindelijke suffix tree wordt FT genoemd. Het algoritme begint door de NodeMerge functie aan te roepen op de drie bomen, waardoor een root-node wordt gecreëerd voor FT. Onder die root-node komen 5 groepen, waarvan de T-groep en de $-groep slechts één edge hebben.



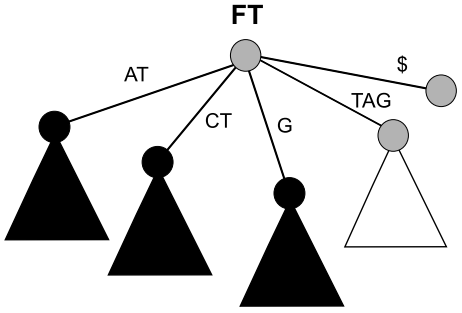
Figuur 2.6: De A- C- en G-groep bevatten meer dan één edge zodat de EdgeMerge functie wordt opgeroepen op elk van deze groepen. Eerst wordt de edge naar de A-groep berekend, omdat voor T2 als T3 de labels hetzelfde zijn, blijft het label naar de A-groep in de samengevoegde suffix tree ook hetzelfde. De overeenkomstige nodes worden samengevoegd door NodeMerge op te roepen.



Figuur 2.7: Voor de C-groep zijn niet alle edges gelijk, dus wordt uit de drie labels de langste gemeenschappelijke prefix berekend en worden de edges gesplitst in T2 en T3. De nieuwe nodes van T2 en T3 worden door NodeMerge samengevoegd met de reeds bestaande node van T1.



Figuur 2.8: Ook bij de G-groep moet een nieuwe node worden gecreëerd met als label de langste gemeenschappelijke prefix van de labels. Er moet een nieuwe node worden geïnsert in T1, T2 en T3, die vervolgens worden samengevoegd door NodeMerge.



Figuur 2.9: De uiteindelijke suffix tree FT.

2.6 Fysieke voorstelling suffix tree

Er bestaan een aantal datastructuren om een suffix tree voor te stellen, elk met hun eigen voor- en nadelen. In dit hoofdstuk geven we eerst de twee basisvoorstellingen, de gelinkte lijst en de array voorstelling. Daarna gaan we dieper in op twee implemenaties van Kurtz et Al. om een suffix tree zo compact mogelijk voor te stellen zonder veel snelheidsverlies tijdens de constructie. De eerste methode is een lineair algoritme dat gebaseerd is op de gelinkte lijst voorstelling. De tweede voorstelling, *wotd* genaamd, vertrekt van een niet lineair algoritme maar stelt de suffix tree nog compacter voor, onder andere door geen suffix links te gebruiken. In de laatste sectie geven we nog een implementatie van Japp, waarbij de top van de suffix tree wordt gecomprimeerd.

2.6.1 Basisvoorstelling

Eén van de meest logische voorstellingen van een suffix tree is een boomstructuur waarbij per interne node een gelinkte lijst met child-nodes wordt bewaard, dit noemt men de gelinkte lijst voorstelling van een suffix tree. Het kost weinig moeite om deze lijst eventueel gesorteerd op te bouwen, wat belangrijk is voor sommige toepassingen op suffix trees. Indien men een vrij groot alfabet heeft, gaat de kost van het doorlopen van de lijst doorwegen tijdens zowel de constructie als het gebruik van de suffix tree. Een ander nadeel van deze voorstelling is dat er ruimte verloren gaat aan de pointers die de lijst bij elkaar moeten houden, zeker indien het aantal kinderen van een node de grootte van het alfabet Σ benadert. In dat geval had men beter een array gealloceerd, wat ons tot de volgende voorstelling brengt.
Een alternatief voor de gelinkte lijst is om elke interne node een array van grootte Σ te geven. Elk element van de array representeert een karakter uit het alfabet, indien een edge begint met dat karakter bevat dat element uit de array een pointer naar een kind-node. Dit soort voorstelling van een suffix tree wordt de array implementatie van een suffix tree genoemd. Het is zeer gemakkelijk om dit soort suffix tree te updaten, maar het belangrijkste nadeel is dat er veel geheugen wordt verspild. Hoe lager men in de boom komt, hoe minder kinderen elke node gemiddeld heeft. Onderaan de boom zijn dan ook de meeste van de pointers in de arrays null-pointers. Het is ook logisch dat bij een groot alfabet er meer plaats gaat verkwist worden dan bij een klein alfabet.
Het is duidelijk dat een aantal methodes samen kunnen worden gebruikt om een suffix tree compacter op te slaan. In hoofdstuk 2.6.4 zien we twee methodes om een suffix tree te construeren waarbij zowel arrays als gelinkte lijsten worden gebruikt om een suffix tree voor te stellen.

2.6.2 Gelinkte lijst

Kurtz heeft veel aandacht besteed om suffix trees te verkleinen, het hoofddoel in zijn paper [Kur99a] was om suffix trees zo compact voor te stellen zodat de volledige suffix tree in het geheugen kon opgebouwd worden. Met de volledige suffix tree in het geheugen kan men immers gemakkelijker lineaire constructiealgoritmes gebruiken, van zodra de suffix tree voor een deel in het secundair geheugen zit gaat - zoals al eerder aangetoond - het volgen van suffix links naar nodes die zich op de harde schijf bevinden bijzonder veel tijd in beslag nemen. Zijn voorstel gebruikt de suffix tree voorstelling van McCreight [McC76], namelijk de gelinkte lijst, waarop Kurtz dadelijk enkele optimalisaties doorvoert. Vanuit deze implementatie vertrekt Kurtz om een nog compactere suffix tree te verkrijgen.

Stel dat we een substring w hebben die in de suffix tree wordt voorgesteld door een branching node $\overline{w}$, dan weten we dat w meerdere keren voorkomt in de sequentie S. De meest logische oplossing is om de node $\overline{w}$ te koppelen aan het eerste voorkomen van w in S. Maar er bestaat ook nog een andere mogelijkheid die niet minder logisch is, namelijk $\overline{w}$ koppelen aan de tweede positie waar w in voorkomt. De tweede keer dat we bij het opbouwen van een suffix tree een prefix van een suffix tegenkomen betekent dat er een branching node wordt ingevoegd.
Laat $head_i$, met $i \in [2, n+1]$, de langste prefix zijn van $S[i..n]$ die ook een prefix is van $S[j..n]$ voor $j \in [1, i-1]$. Voor elke branching node $\overline{w}$ duidt $headposition(\overline{w})$ de kleinste integer $i \in [1, n+1]$ zodat voor de overeenkomstige string w van $\overline{w}$ geldt dat $w = head_i$. Door de constructie van de suffix tree weten we dat voor elke suffix $S[i..n]$ een branching node bestaat in de suffix tree en voor elke branching node $\overline{w}$ een headposition bestaat.
Kurtz geeft een simpele voorstelling voor een suffix tree door middel van een gelinkte lijst. Een suffix tree τ wordt voorgesteld door twee tabellen T_{leaf} en T_{branch}. Voor elke leaf node is slechts één pointer nodig naar de right sibling. De leaf nodes worden lexicografisch opgeslagen, zodat voor alle $i, j \in [0, n]$ met $i < j$ geld dat $T_{leaf}[i] <_{lex} T_{leaf}[j]$. Elke branching node $\overline{w}$, die woord w voorstelt, wordt opgeslagen als vijf waarden die elk in een integer passen:

1. *firstchild* verwijst naar het eerste kind van $\overline{w}$
2. *branchbrother* wijst naar de rechtse sibling van $\overline{w}$, indien deze niet bestaat is het een null-waarde
3. *depth* is de diepte van node $\overline{w}$, de lengte van de overeenkomstige string w
4. *headposition* van $\overline{w}$

5. *suffix link* wijst naar een node $\overline{v}$ die string v voorstelt, waarbij $av = w$ met $a \in \Sigma$

De array van de branching nodes wordt lexicografisch geranschikt volgens de string die elke node voorstelt, dus de volgorde van T_{branch} is dezelfde als wanneer we de suffix tree depth-first zouden doorlopen.
Men kan zich ook afvragen waarom de depth van een branching node, de lengte van het path naar die node, wordt bijgehouden in plaats van de lengte van de inkomende edge. De reden hiervoor is dat tijdens de constructie van de suffix tree de lengte van de inkomende edge kan verkleind worden indien deze edge wordt gesplitst om een branching node toe te voegen. De diepte van een node daarentegen blijft altijd hetzelfde. Dus de *depth* is niet hetzelfde als de node-depth in de beschrijving van het algoritme van Ukkonen in hoofdstuk 2.4.4, daar telt het aantal nodes op het path in plaats van de lengte van dat path.

De waarden van *firstchild*, *branchbrother* en $T_{leaf}[j]$ kunnen verwijzen naar zowel een *leaf* als een *branching* node, om het verschil aan te duiden wordt de meest significante bit van de integer gebruikt om aan te duiden naar welke tabel de pointer verwijst.

Stel dat de suffix tree bestaat uit q interne nodes $b_1, ..., b_q$ gesorteerd op *headposition*, zodat $headposition(b_i) < headposition(b_i + 1)$ voor elke $i \in [1..q-1]$. Elke node b_i krijgt een nummer i, waarbij b_1 uiteraard de root is. In de tabel T_{branch} worden de nodes opgeslagen volgens *headposition*.
De tabel T_{leaf} gebruikt n integers, tabel T_{branch} 5q integers. In het ergste geval hebben we dat $q = n$, zodat de worst-case geheugengebruik $5n$ integers bedraagt. Het is ook mogelijk om de head positie i op index i in T_{branch} te plaatsen . Op die manier is het niet nodig om de head positie van elke node apart bij te houden. Deze implementatie heeft wel tot gevolg dat er altijd $4n$ integers nodig zijn voor de T_{branch} tabel. In de praktijk is q bijna altijd kleiner dan $0.8n$, zodat deze manier van opslag toch meer plaats inneemt.
Men zou altijd de nodes kunnen opslaan in breadth-first of depth-first volgorde, om plaats te sparen voor de sibling of kind referenties. Jammer genoeg weet men bij de constructie van de suffix tree niet hoeveel kinderen of siblings een node gaat hebben, zodat er bij het invoegen van een node een onbepaald aantal nodes moeten verschoven worden in de tabel. Dit in tegenstelling tot de 3 of 4 updates die er bij de huidige implementatie moeten gebeuren indien een node wordt toegevoegd.
We hebben aan de hand van deze voorstelling de sequentie `ataccatc$` uitgewerkt, indien een getal verwijst naar de T_{leaf} tabel staat het in het vet gebruikt.

T_{leaf}		
leaf	leaf number	$T_{leaf}[j]$
ataccatc$	1	**6**
taccatc$	2	**7**
accatc$	3	4
ccatc$	4	**8**
catc$	5	**4**
atc$	6	null
tc$	7	null
c$	8	null
$	9	null

T_{branch}					
branching node	root	$\overline{a}$	$\overline{c}$	$\overline{at}$	$\overline{t}$
node number	1	2	3	4	5
firstchild	2	**3**	**5**	**1**	**2**
branchbrother	null	3	5	null	**9**
depth	0	1	1	2	1
headposition	1	3	5	6	7
suffixlink		1	1	5	1

Enkele verbeteringen

Nu we de gelinkte lijst hebben voorgesteld overlopen we een aantal observaties die hebben geleid tot een nog compactere voorstelling van de suffix tree. De suffix tree is compacter voor te stellen omdat de T_{branch} tabel redundante informatie bevat. Dankzij gebruik te maken van enkele eigenschappen van head positions kan deze redundante informatie worden weggewerkt.

Observatie 1

1. Indien u en v verschillende branching nodes zijn dan volgt hieruit dat $headposition(u) \neq headposition(v)$.
2. Voor elke branching node $\overline{aw}$ met als overeenkomstige string aw, bestaat er een branching node $\overline{w}$, met als overeenkomstige string w, waarvoor geldt $headposition(\overline{aw}) + 1 \geq headposition(\overline{w})$.

De eerste observatie is vanzelfsprekend, een suffix kan niet verantwoordelijk zijn voor twee branching nodes. De tweede observatie is gemakkelijk in te zien dankzij volgende voorbeelden. Neem als eerste voorbeeld de - artificiële - sequentie tawcgawggwt$, in dit geval hebben we het karakter a dat twee keer voorkomt voor een w. Suffix awcgawggwt$ wordt toegevoegd voor suffix awggwt$, waardoor de branching node $\overline{aw}$ ontstaat. In dit voorbeeld wordt de branching node $\overline{aw}$ gemaakt voor de branching node $\overline{w}$, zodat

$headposition(\overline{aw})+1 = headposition(\overline{w})$. Als we de sequentie tawcgbwgawt\$ nemen, zien we dan de branching node $\overline{w}$ voor branching node $\overline{aw}$ ontstaat, de tweede suffix met prefix w wordt ingevoegd voor de tweede suffix met prefix aw zodat $headposition(\overline{aw}) > headposition(\overline{w})$.
Kortom, observatie 1.2 impliceert ook dat voor elke $\overline{aw}$ geldt dat ofwel $headposition(\overline{aw}) + 1 = headposition(\overline{w})$ ofwel $headposition(\overline{aw}) > headposition(\overline{w})$. We delen de nodes nu op in twee groepen: *large* nodes en *small* nodes (het waarom van het opdelen wordt later duidelijk). Een node $\overline{aw}$ is een *small* node indien $headposition(\overline{aw}) + 1 = headposition(\overline{w})$, node $\overline{aw}$ is *large* indien $headposition(\overline{aw}) > headposition(\overline{w})$. De root zelf is *small* noch *large*.

Observatie 2

1. Indien $\overline{aw}$ small is, dan geldt $nodenum(\overline{aw}) + 1 = nodenum(\overline{w})$.
2. Indien $nodenum(\overline{aw}) = q$ en $q > 1$ dan is $\overline{aw}$ large, waarbij q het aantal branching nodes is.

Deze observatie toont aan dat een small node altijd gevolgd wordt door een branching node, en dat de laatste branching node een large node is. Volgens deze observatie kunnen we de sequentie $b_2, ..., b_q$ branching nodes opsplitsen in kettingen van nul of meer opeenvolgende small nodes gevolgd door één enkele large node. Een ketting is een ononderbroken deelsequentie $b_l, ..., b_r$ van $b_2, ..., b_q$ met $r \geq l$ zodat het volgende geldt:

- b_{l-1} is geen small node
- $b_l, \ldots, b_{r-1}$ zijn small nodes
- b_r is een large node

Het is gemakkelijk in te zien dat elke branching node behalve de root een member is van precies één ketting. Volgende observatie toont aan dat small nodes flink wat redundante informatie bevatten.

Observatie 3

Stel dat $b_l, \ldots, b_r$ een ketting is, dan zijn volgende eigenschappen waar voor elke $i \in [l, r-1]$:

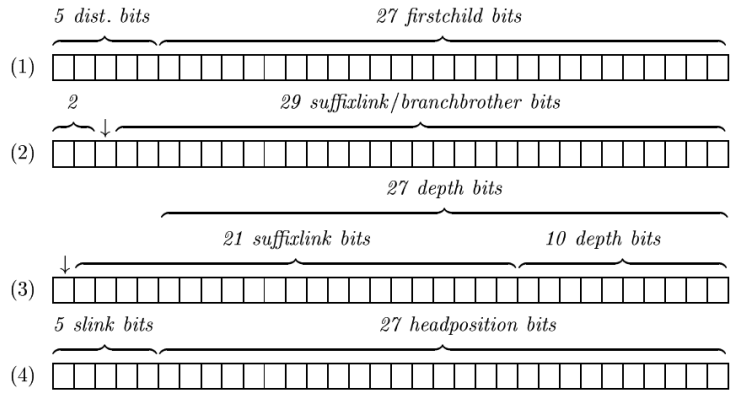
1. $b_i.depth = b_r.depth + (r - i)$
2. $b_i.headposition = b_r.headposition - (r - i)$
3. $b_i.suffixlink = b_{i+1}$

Uit deze vaststelling heeft Kurtz een verbeterde gelinkte lijst implementatie gemaakt, waarbij een small node minder ruimte nodig heeft dan een large node, de informatie kan immers worden afgeleid uit die large node. De laatste observatie toont immers aan dat het niet nodig is om $b_i.depth$, $b_i.headposition$ of $b_i.suffixlink$ voor $i \in [l, r-1]$ op te slaan, $b_i.suffixlink$ verwijst naar de volgende node in de ketting, en indien de afstand $b_i.distance = r - i$ tussen de small node b_i en de large node b_r bekend is, kunnen $b_i.depth$ en $b_i.headposition$ in constante tijd worden achterhaald.
De branching nodes worden dus in twee groepen opgedeeld, *small* en *large* nodes, die elk andere informatie bevatten. Voor elke *small* node $\overline{w}$ wordt $\overline{w}.distance$, $\overline{w}.firstchild$ en $\overline{w}.branchbrother$ opgeslagen. Elke *large* node houdt de waarden van $\overline{w}.firstchild$, $\overline{w}.branchbrother$, $\overline{w}.depth$, $\overline{w}.headposition$ en $\overline{w}.suffixlink$ bij.
Kurtz heeft aangetoond dat een small node kan worden opgeslagen in twee integers en een large node in vier. We overlopen zijn voorstelling en maken hierbij een aantal kritische opmerkingen.

Eerst bekijken we hoe een large node precies wordt voorgesteld. We weten dat deze zoals de small nodes twee integers *firstchild* en *branchbrother* bevat. Dankzij enkele observaties zijn er slechts twee extra integers nodig om de rest van de waarden (*depth*, *headposition* en *suffixlink*) bij te houden. Gegeven een large node $\overline{w}$ die een meest rechtse kind $\overline{v}$ heeft. Dan bestaat een reeks waarden die begint met een *firstchild*-referentie en voor de rest $k-1$ *branchbrother*/T_{leaf}-referenties die $\overline{w}$ en $\overline{v}$ aan elkaar linken. Indien de node v een blad voorstelt voor een bepaalde suffix $S[j..n]$ waarvan $j \in [1, n+1]$ dan is $T_{leaf}[j]$ een null-referentie. Aan de andere kant, indien node $\overline{v}$ een branching node is dan is $\overline{v}.branchbrother$ een null-referentie. Uiteraard kan een null-waarde worden voorgesteld door één bit, dus deze integer kan gebruikt worden om $\overline{w}.suffixlink$ op te slaan. Dit heeft dus tot gevolg dat we de kinderen van node $\overline{w}$ moeten doorlopen om de suffixlink van $\overline{w}$ te achterhalen.
Dus om een suffix link te volgen moeten er in worst case $O(|\Sigma|)$ nodes worden afgelopen, dit betekent gelukkig genoeg dat de suffix tree nog kan gemaakt worden in $O(|\Sigma|n)$ time omdat er bij het opbouwen van een suffix tree slechts n keer een suffix link wordt gevolgd. Bovendien hebben algoritmes die gebruik maken van suffix tree al een alfabet-factor in hun tijdscomplexiteit. In de praktijk kan dit echter toch de opbouw van een suffix tree vertragen, daarom wordt alweer een truc toegepast.
In de meeste gevallen hebben nodes die dicht bij de root liggen het meeste kinderen. Dus vooral voor deze nodes gaat het opzoeken van suffix links het traagste verlopen. Aan de andere kant is hun diepte meestal klein, er worden slechts weinig bits gebruikt om deze diepte op te slaan. Daarom kan de rest van de integer worden gebruikt om de *suffixlink* in op te slaan. Kurtz stelt voor om aan nodes die een diepte kleiner of gelijk aan 2^{10}-1 =

1023 hebben, de suffix link toe te voegen.

Een *small* node $\overline{w}$ bevat $\overline{w}.distance$, $\overline{w}.firstchild$ en $\overline{w}.branchbrother$, maar neemt toch slechts twee integers in beslag. Dit is mogelijk door in een ketting na 31 small nodes een large node toe te voegen. Daardoor kan de *distance* slechts vijf bits in beslag nemen, deze waarde kan dus bijgevoegd worden in één van de andere twee integers. De kans dat een ketting langer dan 32 nodes is, dus dat de distance groter wordt dan 32, wordt vrij laag ingeschat door Kurtz.

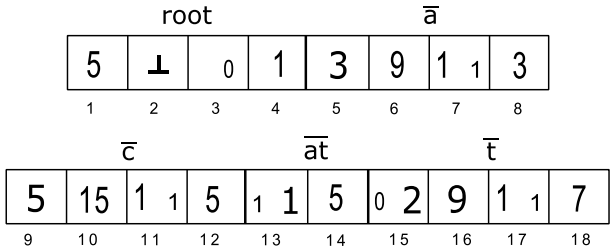


Figuur 2.10: Bit layout voor een small en large node

In figuur 2.10 staat de bit-layout van de branching nodes, (1) en (2) stellen de twee integers van de small node voor en (1)-(4) de vier integers voor de large node. Een small node slaat in integer (1) de depth op in de eerste 5 bits, de depth heeft een waarde tussen 1 en 31, bij een large node is deze waarde altijd 0. Hierdoor is het onmiddellijk duidelijk of we te maken hebben met een small of een large node. De overblijvende 27 bits van integer (1) en de twee eerste bits van integer (2) worden gebruikt om de *firstchild*-referentie op te slaan. De bit in integer (2) gemarkeerd met een ↓ is de null-bit, indien deze geset is, is $\overline{v}$ het meest rechtse kind van $\overline{w}$ en wordt in de overblijvende 29 bits de *suffixlink* van $\overline{w}$ opgeslagen. Indien die null-bit niet geset is, wordt de *branchbrother*-referentie op die plaats opgeslagen.
Een large node heeft nog twee extra integers (3) en (4). Indien de eerste bit van integer (3) is geset, dan is de diepte van de large node ten hoogste $2^{10} - 1 = 1023$ en wordt in de 10 meest rechtse bits de distance-waarde bijgehouden. De overblijvende 21 bits van integer (3) en de vijf eerste bits van integer (4) worden gebruikt om de suffix-link bij te houden. Indien de eerste bit van integer (3) niet is geset, dus indien de depth groter is dan 1023, worden de 27 rechtse bits van integer (3) gebruikt om de *depth* op te slaan. De rechtse 27 bits van integer (4) worden altijd gebruikt voor de *headposition*.

Zowel de small als de large nodes worden in dezelfde tabel T'_{branch} verzameld, ze worden gerangschikt volgens hun headposition. Een verwijzing naar een branching node gebeurt via zijn *base* adres in T'_{branch}, de index van de eerste integer van die node. Stel dat we σ small nodes hebben en λ large nodes, dan bevat de T'_{branch}-tabel $2\sigma + 4\lambda$ integers. Samen met de T_{leaf}-tabel betekent dit dat er $n + 2\sigma + 4\lambda$ integers nodig zijn om de suffix tree op te slaan.
Omdat een small node slechts de helft is van een large node is het belangrijk de verdeling ervan te weten. Volgens Kurtz ligt λ, het aantal large nodes, bij een binair alfabet rond de waarde $0.7n$. Ze kwamen uit op dit getal na enkele proeven, maar kunnen dit niet bewijzen zodat er gerekend moet worden met een logische upper bound van $\lambda \leq n$.
In de voorstelling van de small en large nodes wordt ook het aantal bits dat gebruikt kan worden om naar een blad in de T_{leaf}-tabel te verwijzen, gelimiteerd. Hierdoor kunnen er niet $2^{32} - 1$, de maximale waarde van een integer, bladeren geadresseerd worden. Men kan aantonen dat er suffix trees kunnen gemaakt worden van input sequenties die bestaan uit maximaal $2^{27} - 1 = 134.217.727$ elementen. De beperkende factor is dat een *first-child*-referentie uit 27 bits bestaat, dus er kunnen slechts $2^{27} - 1$ bladeren geadresseerd worden.
We hebben bovenstaand voorbeeld verder uitgewerkt in figuur 2.11. We hebben slechts één small node, namelijk $\overline{at}$. De small node $\overline{at}$ hoort bij large node $\bar{t}$. De afstand tussen de twee nodes is 1, dit is de kleine 1 bij de eerste positie van $\overline{at}$. Bij de eerste positie van de large node $\bar{t}$ staat een kleine 0, om aan te duiden dat het een large node is. Van elke large node uitgezonderd de root wordt de derde positie gedeeld door links de (geringe) depth en rechts de suffix link. In de T_{leaf}-tabel worden in de $T_{leaf}[j]$-waardes die null zijn, de suffix links van hun parents bijgehouden. We duiden ze aan door ze te onderlijnen. Deze suffix links staan ook in de $T_{branch}[j]$ tabel omdat de depth van deze nodes lang geen 1023 bereikt, zodat er nog plaats overblijft om ze te stockeren.

T_{leaf}		
leaf	leaf number	$T_{leaf}[j]$
ataccatc$	1	**6**
taccatc$	2	**7**
accatc$	3	4
ccatc$	4	**8**
catc$	5	**4**
atc$	6	15
tc$	7	1
c$	8	1
$	9	null



Figuur 2.11: De gelinkte lijst voorstelling van Kurtz

2.6.3 Lazy suffix trees

In vorig hoofdstuk hebben we een methode gezien om een suffix tree die in lineaire tijd geconstrueerd wordt zo klein mogelijk te houden. Hier bekijken we een suffix tree constructiealgoritme dat niet lineair is maar nog minder plaats inneemt, de lazy suffix tree [GKS99]. Suffix links worden bij deze voorstelling niet bewaard. Het algoritme is write-only top-down, nodes die weggeschreven zijn naar tabel T worden nooit meer bezocht tijdens de constructie. Ook al werkt het algoritme niet lineair, toch is het in de praktijk goed werkbaar door zijn simpliciteit en zijn goede *locality of reference*. We beschrijven hier twee versies van het algoritme, een *luie* (*lazy*) manier om een suffix tree te construeren en een *gulzige* (*eager*). Bij de luie methode wordt de subtree onder een node pas uitgewerkt als een algoritme expliciet toegang vraagt tot die node. De *eager* methode werkt tijdens de constructie de volledige suffix tree uit.

Werking van het *wotd*-algoritme

Het *wotd*-algoritme gaat uit van het feit dat de subtree onder de branching node $\overline{u}$ volledig bepaald wordt door alle suffixen van de sequentie $S\$$ die u als prefix hebben. Dus als we de set $R(\overline{u}) = \{s \mid us \text{ is een suffix van } S\$\}$ van overblijvende suffixen hebben, kunnen we de node $\overline{u}$ evalueren. Het evalueren van de set $R(\overline{u})$ gebeurt als volgt, eerst wordt de set verdeeld in groepjes volgens het eerste karakter na de prefix u. Voor elk karakter $c \in \Sigma$, laat $groep(\overline{u}, c) = \{w \in \Sigma^* \mid cw \in R(\overline{u})\}$ de *c-groep* van $R(\overline{u})$ zijn. Indien een c-groep slechts één string w telt, dan heeft node $\overline{u}$ een leaf edge met als label cw. Indien een groep($\overline{u}$, c) ten minste twee strings telt dan bestaat er een edge cv naar node $\overline{ucv}$ waarbij v de langste gemeenschappelijke prefix is van alle string in groep($\overline{u}$, c). Node $\overline{ucv}$ kan dan verder geëvalueerd worden via de set $R(\overline{ucv}) = \{w \mid vw \in groep(\overline{u}, c)\}$.
Het *wotd*-algoritme begint met de set $R(root)$ waarin alle suffixen zitten van sequentie $S\$$. Alle nodes van de suffix tree kunnen recusief worden berekend uit de set van de overblijvende suffixen.

De suffix tree data structuur

Om de structuur van de suffix tree uit te leggen moeten we eerst enkele begrippen definiëren.
Een *leaf set* $\ell(\alpha)$ van een node α wordt als volgt gedefinieerd. Voor een blad $\overline{s_j}$ geldt dat $\ell(s_j) = \{j\}$, voor een branching node $\overline{u}$ geldt dat $\ell(\overline{u}) = \{j \mid \overline{u} \xrightarrow{v} \overline{uv}$ is een edge en $j \in \ell(\overline{uv})\}$. We kunnen een orde tussen de kinderen van een branching node definiëren, $\overline{uv} \prec \overline{uw}$ als en slechts als $\min\ell(\overline{uv}) < \min\ell(\overline{uw})$. Sets van bladeren zijn nooit leeg en $\ell(\overline{uv}) \cap \ell(\overline{uw}) = \emptyset$.
Zoals we in hoofdstuk 2.4.5 al hebben uitgelegd is een edge label een substring van $S\$$ en kan deze worden voorgesteld als een paar indexen (i,j) die naar het begin en einde van de substring in $S\$$ wijzen. We tonen nu aan dat het volstaat om enkel de linkse index (li) op te slaan, terwijl de labels van de edges toch nog in constante tijd achterhaald kunnen worden. Stel dat $\overline{u} \xrightarrow{v} \overline{uv}$ een edge is in de suffix tree. We definiëren $li(\overline{uv}) := \min\ell(\overline{uv}) + |u|$ als zijnde de linkse index van $\overline{uv}$. Stel dat $\overline{uv}$ een branching node is en $i = li(\overline{uv})$, en stel dat $\overline{uvw}$ het kleinste kind is van $\overline{uv}$ volgens de '$\prec$' relatie. We weten dus dat $\min\ell(\overline{uv}) = \min\ell(\overline{uvw})$ en dus ook dat $li(\overline{uvw}) = \min\ell(\overline{uvw}) + |uv| = \min\ell(\overline{uv}) + |u| + |v| = li(\overline{uv}) + |v|$. Stel nu dat $r = li(\overline{uvw})$. Dan volgt daaruit dat $v = s_i \dots s_{i+|v|-1} = s_i \dots s_{li(\overline{uvw})-1} = s_i \dots s_{r-1}$.
Concreet komt het er op neer dat we het label van de edge kunnen vinden door te kijken naar de linker index van de parent edge, en de linker index van het meest linkse kind. Uiteraard moet het label van het meest linkse kind in constante tijd opgevraagd kunnen worden. Dit is mogelijk dankzij de referentie *firstchild*$(\overline{u})$ naar het eerste kind $\overline{u}$. Zoals al eerder aangeduid worden de kind-nodes gesorteerd via de $\prec$-relatie en in die volgorde vlak langs elkaar opgeslagen. Enkel de edge naar het eerste kind wordt expliciet via de *firstchild*-waarde aangeduid. Zowel de *li*- als de *firstchild*-integers worden in dezelfde tabel T bewaard.
Naar een node $\overline{u}$ wordt gerefereerd via de index in tabel T waar $li(\overline{u})$ wordt bewaard. Om een volwaardige suffix tree representatie te verkrijgen moet ook nog aangeduid worden wanneer we te maken hebben met een blad. Dit gebeurt via een *leaf bit*, een *rightmost child bit* duidt een node aan die geen rechterbroer heeft volgens de $\prec$-relatie.

Opbouwen van de suffix tree

Bij het evalueren van een niet uitgewerkte node $\overline{u}$ moeten we weten waar de suffixen die onder deze node horen, de set $R(\overline{u})$, precies gelocaliseerd zijn in de sequentie. Daarvoor wordt een globale array *suffixen* gebruikt, deze bevat pointers naar de suffixen. De set $R(\overline{u})$ houdt voor een niet geëvalueerde node $\overline{u}$ pointers bij naar de suffixen. Deze pointers naar suffixen zijn per set gesorteerd op lengte. Nu worden de grenzen *left*$(\overline{u})$ en *right*$(\overline{u})$ van de pointers van set $R(\overline{u})$ opgeslagen in de twee integers die in tabel T

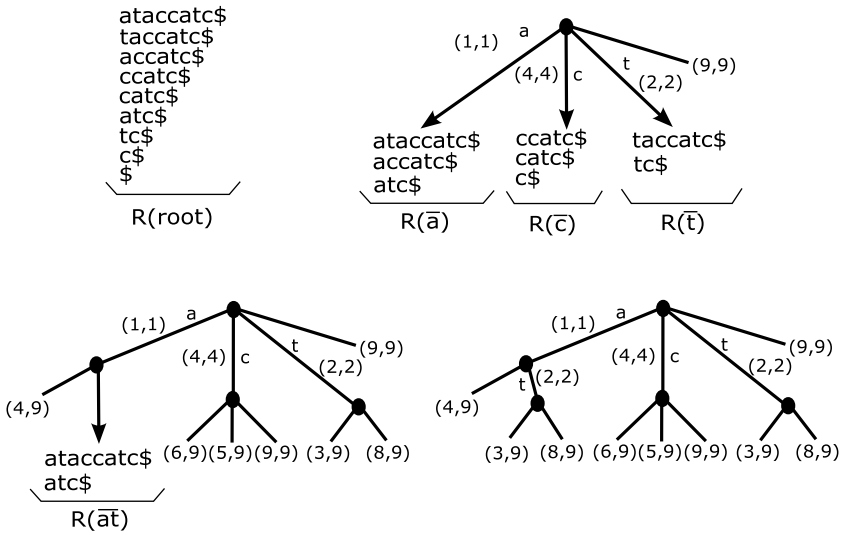
gereserveerd zijn voor de branching node $\overline{u}$. Om nu geëvalueerde en niet geëvalueerde nodes uit elkaar te houden wordt er een derde bit gebruikt, de *unevaluated bit*.
Voor elke node $\overline{u}$ moet worden bepaald hoeveel kinderen dat ze heeft, dit gebeurt door alle eerste karakters van de suffixen in het interval [*left*($\overline{u}$), *right*($\overline{u}$)] uit de tabel *suffixen* te groeperen op hun eerste karakter c. Elk karakter c dat voorkomt komt overeen met een edge die vanuit $\overline{u}$ vertrekt. De suffixen uit een c-groep bepalen volledig de subtree onder die edge. Per c-groep worden de suffixen gesorteerd in een subinterval. Om het label van die edge te vinden moet de langste gemeenschappelijke substring bepaald worden.
De kinderen van node $\overline{u}$ worden in tabel T bewaard, een groep die slechts één suffix telt wordt een blad. Een blad $\overline{b}$ kan worden voorgesteld door een enkele integer, de linker index $li(\overline{b})$. Een groep die uit meerdere suffixen bestaat heeft uiteraard een branching node $\overline{v}$ tot gevolg. Deze node is nog niet uitgewerkt en wordt in tabel T voorgesteld door *left*($\overline{v}$) en *right*($\overline{v}$). De set $R(\overline{v})$ wordt bewaard in tabel *suffixen*. Alle kinderen van node $\overline{u}$ worden vlak na elkaar opgeslagen volgens de relatie $\prec$. Nu we alle kind-nodes van $\overline{u}$ afgelopen hebben, worden de waarden *left*($\overline{u}$) en *right*($\overline{u}$) vervangen door de integers $li(\overline{u})$:= *suffixen*[*left*($\overline{u}$)] en *firstchild*($\overline{u}$). De *unevaluated bit* voor $\overline{u}$ wordt verwijderd.

De nodes in de suffix tree kunnen op verschillende manieren worden uitgebreid. De eerste manier wordt *eager* (=gulzig) genoemd, de nodes worden in depth-first volgorde doorlopen en per node worden de kinderen van links naar rechts toegevoegd. Het algoritme dat de suffix tree op deze wijze opbouwt wordt daarom *wotdeager* genoemd. Een alternatieve manier wordt *lazy* genoemd, een node wordt pas geëvalueerd indien een algoritme voor de eerste keer toegang vraagt tot de subtree. Het algoritme dat deze strategie hanteert wordt *wotdlazy* genoemd.

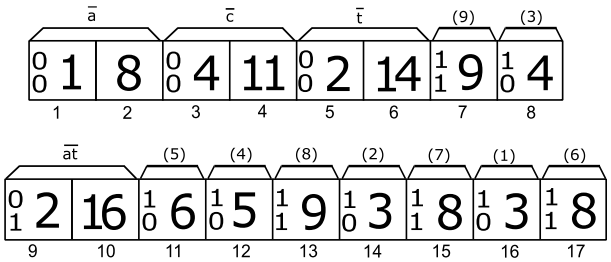
We werken de sequentie `ataccatc$` uit in de wotdeager voorstelling, het eerste karakter van de sequentie heeft index 1. Figuur 2.12 laat de opbouw van de suffix tree niveau per niveau zien, terwijl figuur 2.13 de opgeslagen vorm van de suffix tree voorstelt.

Plaats- en Tijdscomplexiteit

Een suffix tree kan via bovenstaande constructie worden opgeslagen in $2q+n$ integers, waarbij q het aantal branching nodes is. Een boom met n bladeren kan maximaal $n-1$ interne nodes hebben, q kan dus maximaal $n-1$ bedragen. Waar nog rekening mee moet worden gehouden is de extra plaats die nodig is om het algoritme te kunnen laten werken, deze kan maximaal



Figuur 2.12: Het opbouwen van de suffix tree volgens *wotdeager*



Figuur 2.13: De suffix tree volgens *wotdeager*

uit $2.5n$ extra integers bestaan. De *suffixen* array bestaat uit n integers, het algoritme om de suffixen per groep te sorteren op hun eerste karakter neemt nog maximaal $n-1$ integers in beslag, de grootte van de set suffixen die gesorteerd wordt. Daarboven houdt het *wotdeager*-algoritme ook nog een stack bij van ten hoogste $n/2$, om referenties bij te houden naar niet geëvalueerde nodes.

Het is duidelijk dat dit algoritme minder plaats nodig heeft dan de gelinkte lijst voorstelling van Kurtz uit hoofdstuk 2.6.3. Dit verschil zit onder andere in het feit dat het *wotd* algoritme geen suffix links opslaat. Bijgevolg is het ook niet evident om dit algoritme in lineaire tijd te laten werken, de auteurs van [GKS99] gaan niet dieper in op de vraag of dit mogelijk is. De tijdscomplexiteit van het algoritme heeft als average tijdscomplexiteit $\theta(n \log n)$ en als worst-case tijdscomplexiteit $O(n^2)$.
Experimenten tonen aan dat de gebruikte space tussen 9 en 11 bytes per input karakter ligt, afhankelijk van de aard van de sequentie. Vergeleken met de gelinkte lijst voorstelling uit vorig hoofdstuk toegepast op het algoritme van McCreight, betekent het *wotd*-algoritme een verbetering van gemiddeld één byte. Voor DNA is er zelfs een vermindering van 12.80 bytes naar 10.48 bytes voor het genoom van de bacterie *E.coli*. Enkel voor DNA was het gelinkte lijst algoritme significant sneller dan *wotd*. Over het algemeen is de suffix tree van DNA groter dan suffix trees van andere types sequenties. Dit komt omdat het alfabet van DNA zeer kort is, het bevat slechts 4 karakters. Het gevolg is dat er meer branching nodes zijn in de tree, wat neerkomt op een groter geheugengebruik.

Repetitieve sequenties

Bij de bespreking van de performatie van het *wotd*-algoritme wordt ook aangegeven dat het algoritme niet goed overweg kan met stringen waarin veel herhalingen voorkomen. Jammer genoeg is DNA dikwijls repetitief doordat knippen, plakken en recombinatie van stukken DNA vaak voorkomt in de natuur. Lange herhalingen binnen een DNA-sequentie hebben tot gevolg dat er edges zijn met bijzonder lange labels. Op zich heeft dit geen effect op de grootte van de suffix tree, elke substring wordt immers weergegeven door twee indexen die verwijzen naar het begin en einde van de substring in de sequentie. Het probleem is dat de langste gemeenschappelijke prefix van de suffixen van een subtree gezocht moet worden om de edge een label te kunnen geven. Om dit probleem op te lossen kunnen we teruggrijpen naar een observatie uit hoofdstuk 2.4.3. De labels van de edges die uit de nodes $\overline{w}$ en $\overline{aw}$ vertrekken zijn dikwijls dezelfde, ofwel is de edge van $\overline{w}$ een prefix van $\overline{aw}$. De suffixen uit $R(\overline{aw})$ zijn, op een prefix a na, een subset van de suffixen uit set $R(\overline{w})$.
Uit deze observatie kan men gemakkelijk een verbetering afleiden voor het

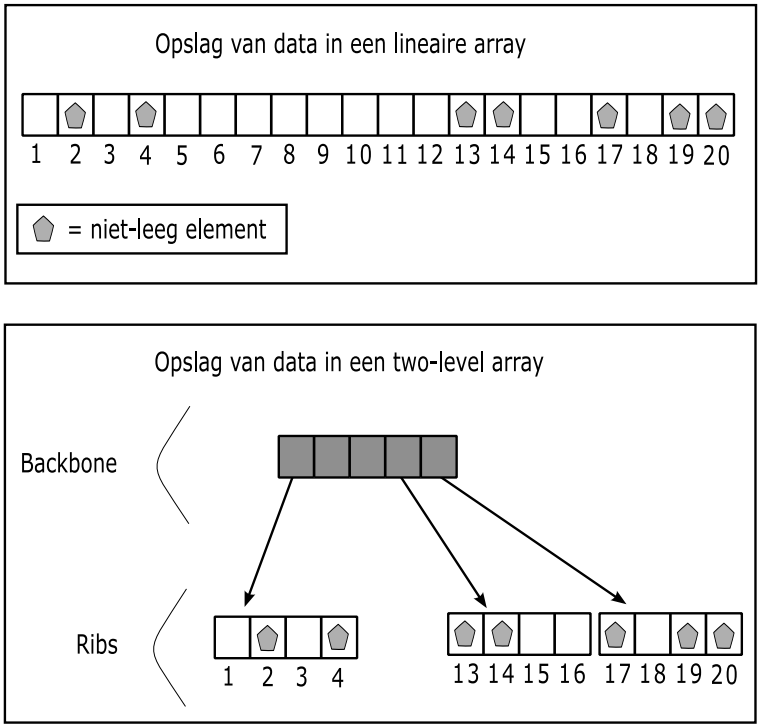
wotd-algoritme. Indien m de lengte is voor de *lgp* die voor node $\overline{w}$ berekend is, dan kan men bij de constructie van node $\overline{aw}$ onmiddellijk de eerste m karakters van de suffixen overslaan. Het write-only karakter van het *wotd*-algoritme gaat hiermee wel verloren, samen met de locality of reference, omdat node $\overline{w}$ opgevraagd moet worden wanneer node $\overline{aw}$ geëvalueerd wordt. Men kan dit idee ook recursief gaan toepassen en zelfs de suffixen gaan toevoegen volgens toenemende suffix lengte. Hiermee creëert men een nieuw algoritme dat fel lijkt op Weiner's *landmark algorithm* [Wei73].

2.6.4 Comprimeren van de top van de suffix tree

Nodes die dicht bij de top van een suffix tree zijn gesitueerd hebben gemiddeld genomen meer kinderen dan nodes die dicht bij de bladeren van een suffix tree geplaatst zijn. Bij de inleiding van dit hoofdstuk hebben we gezien dat nodes met veel kinderen beter worden voorgesteld door middel van arrays met grootte $|\Sigma|$ die pointers bevatten naar alle kinderen, terwijl nodes met relatief weinig kinderen minder plaats innemen indien ze worden opgeslagen in een gelinkte lijst. Het is relatief eenvoudig om hiervoor een datastructuur op te stellen, gewoon de c bovenste niveaus in arrays bewaren en de rest in gelinkte lijsten. Toch kunnen er enkele problemen ontstaan. Ten eerste moet men weten hoeveel niveaus van de suffix tree men gaat bewaren in een array structuur. Dit hangt uiteraard af van de grootte van zowel het alfabet als de sequentie. Een ander probleem is dat een suffix tree niet altijd evenwichtig is, indien niet alle symbolen even veel voorkomen krijgt men een schuine suffix tree. Hierdoor is het moeilijker op voorhand te bepalen hoe groot c moet zijn om de suffix tree optimaal te kunnen opslaan.

Japp heeft een eigen systeem beschreven om een suffix tree voor te stellen, de top-compressed suffix tree [Jap04]. Zijn methode bestaat erin om een suffix tree te zien als een set onafhankelijke subtrees die gelinkt zijn aan een gecomprimeerde top. De c bovenste niveaus van de suffix tree worden gecomprimeerd tot een arrayvoorstelling over twee niveaus.
Stel dat we c niveaus van de suffix tree willen comprimeren, dan heeft elke subtree een prefix van exact c karakters lang. We kunnen die sequenties van c karakters associëren met unieke integers door de karakters te gebruiken als coëfficiënten in een c-de graadse polynomiaal P. Indien we te maken hebben met een alfabet met grootte $|\Sigma|$ betekent dit dat de array met pointers naar mogelijke subtrees $c^{|\Sigma|}$ groot is. Indien de c bovenste niveaus vrij sparse zijn, dus dat de meeste nodes weinig of geen kinderen hebben, zitten er grote opeenvolgende stukken in de array die geen pointers naar subtrees bevatten. Deze leegtes nemen enkel geheugen in en kunnen worden weggewerkt door van de array een two-level array te maken. Als de oorspronkelijke array in s_1 stukken wordt ingedeeld, kan een index-array - in de paper backbone genoemd - met grootte s_1 links bevatten naar die stukken. Indien een bepaald

stuk geen pointers naar subtrees bevat wordt er ook geen geheugen voor gealloceerd. Zie figuur 2.14 voor een verduidelijking.



Figuur 2.14: Opslag van waarden in twee niveaus

Een voor de hand liggend alternatief bestaat erin om een array te maken die per entry een pointer naar een subtree bevat, samen met een integer i die via P is omgezet. Deze array moet gesorteerd zijn volgens i. Nu kan men, gegeven een sequentie, niet onmiddellijk naar de overeenkomstige subtree gaan, maar via een zoek-algoritme kan men deze subtree wel in $\log k$ stappen vinden, met k het aantal subtrees. Andere methodes zijn ook mogelijk, indien c erg groot wordt kan men de arrays op twee niveaus vervangen door een B-Tree.
Naast besparing van geheugen kunnen door deze opsplitsing van de suffix tree queries ook efficiënter worden beantwoord. Men houdt de backbone van de suffix tree in het geheugen en haalt via de ribs de subtrees van de harde schijf van zodra dit nodig is. Per query wordt de harde schijf slechts één of twee keer aangesproken, tegenover meerdere keren indien de suffix tree op een klassieke manier wordt bewaard in het secundair geheugen. Bij deze laatste voorstelling moeten de kinderen van een node worden opgevraagd van harde schijf, daarna opnieuw de kinderen van één van deze nodes, etc. . ., totdat de query is beantwoord. Dit proces kan uiteraard versneld worden door meerdere niveaus per iteratie terug te geven, doch deze voorstelling

is niet zo gecomprimeerd als de methode van Japp zodat er minder in het geheugen kan geladen worden.
Deze voorstelling van een suffix tree heeft jammer genoeg enkele nadelen. Ten eerste is het mogelijk dat een suffix tree niet evenwichtig is, sequenties waarin een aantal symbolen niet evenredig verdeeld zijn of waarbij bepaalde substringen meer voorkomen, hebben een schuine suffix tree. Het resultaat zou dan kunnen zijn dat er deelbomen ontstaan die te groot zijn om in het geheugen geladen te worden.

2.7 Buffering

In de laatste jaren kan men de auteurs die onderzoek verrichten naar de constructie van suffix trees ruwweg in twee groepen indelen, zij die vertrekken vanuit de implementatie van Ukkonen of McCreight en zij die de lineaire constructie laten vallen omwille van de inefficiëntie van suffix links. Bij het algoritme van Ukkonen worden suffix links gebruikt om van de ene locatie naar de andere te gaan in constante tijd. Het probleem is echter dat die locaties op totaal verschillende plaatsen in de boom kunnen liggen. Bij de constructie van een grote suffix tree kan deze niet volledig in het geheugen blijven, waardoor delen van de suffix tree in en uit het geheugen moeten geladen worden wanneer een suffix link naar een positie wijst die zich op de harde schijf bevindt.

In dit hoofdstuk vergelijken we twee verschillende implementaties met elkaar, TOP-Q [BH04] en TDD [TTHP05], wat staat voor Top Down Disk based. We kiezen deze twee systemen om verschillende redenen, ten eerste omdat het recente systemen zijn, de papers over TOP-Q en TDD zijn beide in 2004 gepubliceerd. Daarnaast gaan beide algoritmes uit van een bekend algoritme, TOP-Q gebruikt het algoritme van Ukkonen terwijl TDD het *wotd*-algoritme van Kurtz gebruikt. TOP-Q vertrekt dus van een lineair algoritme terwijl TDD een $\Theta(n \log n)$-algoritme gebruikt. Deze twee systemen zijn ook op de praktijk gericht, het is expliciet de bedoeling om van volledige chromosomen de suffix tree te construeren. Suffix trees van deze omvang kunnen niet meer in het werkgeheugen gebouwd worden, de programma's zijn erop voorzien om de suffix tree gedeeltelijk op de harde schijf op te bouwen. Om hun implementatie zo snel mogelijk te maken wordt er gebruik gemaakt van een buffer management systeem. Dit kan men zien als een teken dat de programma's om een suffix tree te construeren volwassen worden. We hebben deze twee systemen ook test, resultaten kunnen teruggevonden worden in hoofdstuk 2.7.3.

2.7.1 Top-Down Disk-based

De *Top-Down Disk-based techniek*, of kortweg TDD, maakt het mogelijk om met weinig geheugen toch een grote suffix tree te construeren door gebruik te maken van een strategisch buffer-management systeem. Voor het algoritme zijn er een aantal buffers voorzien die nooit allemaal samen in het geheugen passen, behalve bij zeer korte sequenties. Daarom moet het geheugen worden opgedeeld, buffers waar veel random access is kunnen dan een groter deel van het geheugen ter beschikking krijgen. Buffers waarvan de nieuwste elementen het meest worden opgevraagd kunnen dan oudere informatie naar de harde schijf wegschrijven, waardoor deze buffer minder plaats inneemt in het geheugen. De bedoeling van het algoritme is om het aantal disk-accesses tot een minimum te beperken door optimaal gebruik te maken van het werkgeheugen.

De TDD techniek bestaat uit een algoritme dat een suffix tree kan construeren, PWOTD genoemd, en een buffer management strategie. Het PWOTD algoritme, wat staat voor *Partition and Write Only Top Down*, is gebaseerd op het *wotdeager* algoritme van Kurtz. Dit algoritme is al aan bod gekomen in hoofdstuk 2.6.3. We focussen ook speciaal op de datastructuren die nodig zijn tijdens de constructie van de suffix tree, de manier waarop deze structuren gebruikt worden is belangrijk voor het buffer management systeem.
Het PWOTD algoritme bestaat uit twee fasen, in de eerste fase worden de suffixen van de input sequentie ingedeeld in $|\Sigma|^{prefixlen}$ partities, met $|\Sigma|$ de grootte van het alfabet en *prefixlen* de diepte van de partitionering. Het partitioneren gebeurt door de volledige sequentie van links naar rechts te scannen, waarbij op elke index positie i wordt gekeken tot welke partitie deze suffix behoort. In een partitie zitten alle pointers naar suffixen waarvan de prefix met lengte *suffixlen* hetzelfde is.
In de tweede fase van het PWOTD algoritme wordt voor elke partitie de overeenkomstige suffix tree opgebouwd. Deze fase gebruikt vijf verschillende datastructuren: de input sequentie (*String*), een array met suffix pointers (*Suffixen*), een tijdelijke array (*Temp*), een stack (*Stack*) en de suffix tree (*Tree*).
Neem de sequentie `ataccatc$` als voorbeeld, als we *prefixlen* = 1 nemen hebben we in het begin 4 groepen, de A-, C-, T- en $-groep. Alle suffixen die beginnen met hetzelfde karakter vormen samen een subtree, omdat de grootte van deze groep vast ligt wordt er plaats voor gereserveerd in de *Tree*. De A-, C- en T-groep hebben elk meer dan één element, dus we weten dat het branching nodes zijn. In de *Tree* wordt plaats gereserveerd voor elk van deze nodes waarna ze op de stack worden gepushed. De $-groep bestaat slechts uit één suffix en kan daarom rechtstreeks naar de *Tree* geschreven worden.

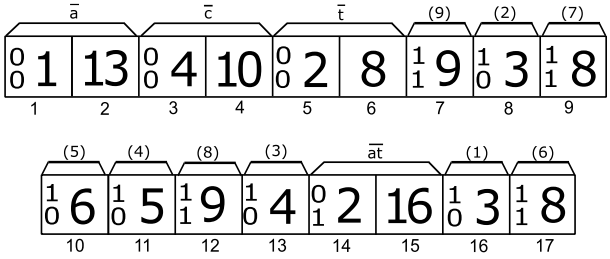
Van zodra alle groepen van de suffix array zijn bekeken, wordt de bovenste node van de stack gepopt en wordt van de overeenkomstige suffix groep de langste gemeenschappelijke prefix (lgp) gezocht. Deze lgp vormt het label van de edge tussen de parent-node en de top-node van de groep. De lgp wordt voor elke suffix weggelaten en de suffix groep wordt opnieuw gesorteerd.
Eerst wordt *Suffixen* gevuld met de suffixen van een partitie, maar zonder de *prefixlen* eerste karakters. Voortgaand op ons voorbeeld betekent dit dat de eerste partitie met prefix A bestaat uit de set suffixen {`ataccatc$`, `accatc$`, `atc$`}. De *Suffixen*-tabel wordt dus gevuld met de suffixen met beginpositie {1,3,6}. De lgp van deze suffixen bedraagt 1, deze prefix wordt dus van de suffixen afgetrokken zodat we de set {2,4,7} verkrijgen. Vervolgens moeten de suffixen in deze array enkel volgens het eerste karakter worden gesorteerd, dit gebeurt via een *count-sort* algoritme wat in lineaire tijd werkt voor een constante alfabetgrootte. Bij het kopiëren van de suffixen naar de *Temp* array wordt geteld hoeveel keer het eerste karakter van de suffixen voorkomt. Als de suffixen in de *Temp* array terug worden gekopieerd naar de *Suffixen*-tabel moet enkel per voorkomend karakter een offset worden bijgehouden. De set suffixen is nu {4,2,7}, wat overeen komt met de suffixen {`ccatc$`, `taccatc$`, `tc$`}. De C-groep heeft één suffix, de T-groep heeft er 2. In de *Tree* wordt dus de ene suffix van de C-groep weggeschreven als blad, voor de T-groep maken we op de *Tree* een branching node die we daarna op de stack pushen.
Het recursief opsplitsen van de suffix groepen blijft duren totdat de *Stack* leeg is en in elke groep slechts één element zit, een leaf-node. Hieronder geven we de pseudo-code voor het algoritme.

Algoritme 4. *PWOTD(Sequentie, prefixlen)*

Fase 1:
Scan *Sequentie* en deel *Suffixen* in volgens de eerste
prefixlen symbolen van elke suffix
Fase 2:
For each partition {
  Vul *Suffixen* met suffix-indexen van de huidige partitie
  Sorteer *Suffixen* op het eerste symbool via *Temp*
  Schrijf branching en bladnodes naar de *Tree*
  Push niet-geëvalueerde nodes op de *Stack*
  while *Stack* niet leeg {
    Pop een node
    Zoek de langste gemeenschappelijke prefix van alle suffixen
    in het overeenkomstige deel van *Suffixen*
    Sorteer deze suffixen op hun eerste karakter via *Temp*

```
        Schrijf branching en bladnodes naar de Tree
        Push niet-geëvalueerde nodes op de Stack
      }
    }
```

Bij dit algoritme moeten we nog een kanttekening maken, de volgorde waarmee nodes in de suffix tree worden bezocht komt niet overeen met de volgorde waarin het *wotdeager* algoritme de nodes afloopt. In het PWOTD-algoritme wordt gebruik gemaakt van een stack, dus indien we bij een set suffixen de groepen A, C en T identificeren, met elk meer dan één suffix, pushen we de node voor A, C en T in deze volgorde op de stack. De eerstvolgende node die gepopt gaat worden is dan de node van de T-groep in plaats van de A-groep. Het *wotdeager* algoritme verwerkt de groepen suffixen van links naar rechts, het doorloopt de tree in breadth-first orde.
Het PWOTD-algoritme geeft dus de suffix tree in figuur 2.15, men kan deze vergelijken met de suffix tree geproduceerd door het *wotdeager* algoritme in figuur 2.13.



Figuur 2.15: De suffix tree volgens het PWOTD-algoritme

Grootte van de datastructuren

In *wotdeager* bedraagt de grootte van de Suffixen en de Temp array $4n$ bytes, voor een sequentie van lengte n. Door de partitionering in fase 1 wordt de hoeveelheid geheugen beperkt tot $4n/|\Sigma|^{prefixlen}$. De suffixen opsplitsen in verschillende groepen vermindert de hoeveelheid geheugen om de suffix tree te construeren, onafhankelijke subtrees kunnen in hun geheel worden opgebouwd in main memory.
Stel bijvoorbeeld dat er een suffix tree van een DNA-sequentie bestaande uit 100 miljoen symbolen moet worden opgebouwd. Door een *prefixlen* van 2 te gebruiken vermindert de hoeveelheid geheugen die de *Suffixen* en *Temp* array nodig hebben van 400MB tot 25MB elk, en de *Tree* array van 1200MB tot 75MB. Hoe hoger *prefixlen* is, hoe meer keer moet de volledige sequentie worden doorlopen op zoek naar suffixen met een bepaalde prefix.
Een ander bekend probleem bij het partitioneren is dat symbolen niet altijd random verdeeld zijn, dus dat bepaalde sequenties meer voorkomen dan

andere. Dit heeft tot gevolg dat sommige partities veel groter zijn. Om dan voor alle partities de datastructuren volledig in het geheugen te krijgen moet *prefixlen* zo groot worden gemaakt tot de datastructuren in het geheugen passen.

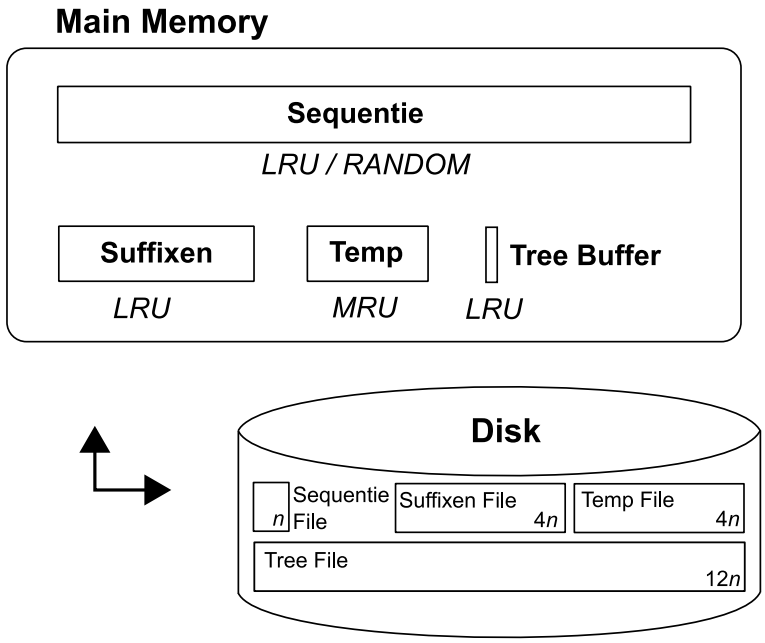
Buffer management

De grote sterkte van het TDD algoritme is dat het datastructuren naar de harde schijf overbrengt als dat nodig is. Hierbij wordt rekening gehouden worden met de aard van de vijf datastructuren van het PWOTD algoritme: *String*, *Suffixen*, *Temp*, *Stack* en *Tree*. Elk van deze datastructuren worden op een andere manier gebruikt tijdens de constructie van de suffix tree.
Het is cruciaal voor een snel algoritme om zo min mogelijk I/O te hebben van en naar de harde schijf. Indien we niet de volledige datastructuur in het geheugen kunnen stockeren, moet er gekeken worden welke data het minst snel gaat opgevraagd worden. We onderscheiden drie verschillende gevallen. Ten eerste is er LRU, *least recently used*. Data die het meest recent is opgevraagd heeft de grootste kans om opnieuw opgevraagd te worden, bij datastructuren waar dit van toepassing is kan men het beste de oudste data naar de harde schijf overbrengen in plaats van de recentste. Ten tweede hebben we MRU, *most recently used*, waarbij data die het recentste is toegevoegd of opgevraagd de minste kans heeft om opnieuw te worden opgevraagd. Dus de meest recente data kan het best worden weggeschreven. Een laatste variant van data ophalen is RANDOM, waarbij men van tevoren niet kan voorspellen welke data gaat gelezen worden. Hier mag men kiezen welke data men wegschrijft.
Als we het in dit hoofstuk hebben over data die wordt weggeschreven bedoelen we niet één of twee integers maar een hele page. De grootte van een page verschilt van systeem tot systeem maar is meestal 4kb. In plaats van stukjes data één voor één weg te schrijven is het efficiënter om blokken data ter grootte van een page naar disk te schrijven.
Over de *Stack* weet men dat de laatst toegevoegde data ook het snelst weer gaat gebruikt worden, data wordt op de stack gepushed en de laatst toegevoegde data wordt het eerst gelezen. Daarom kan best de oudste data op de *Stack* worden weggeschreven naar harde schijf indien het geheugen te vol geraakt, deze data wordt pas veel later terug opgevraagd dan de nieuwst toegevoegde. De *Stack* is de kleinste datastructuur in het algoritme, het aantal elementen dat ze bevat kan maximaal $|\Sigma|$ keer de diepte van de suffix tree worden. Omdat zelfs enorm grote suffix trees meestal slechts een diepte hebben van enkele honderden nodes is het niet de moeite om de *Stack* te bufferen.
Indien een deel van de datastructuren die nodig zijn om een subtree te maken niet in het geheugen past, kan door een juist buffer management veel snelheidswinst geboekt worden. De *Tree* buffer is duidelijk de grootste da-

tastructuur, ze is 8-12 keer zo groot als de oorsprokelijke sequentie. Deze array dient als opslagplaats voor de suffix tree terwijl deze opgebouwd wordt, meestal wordt de data er sequentieel naar toe gescheven wanneer een node recursief wordt onderzocht. Soms wordt een vroeger deel van de boom opnieuw bekeken wanneer een node van de stack wordt gehaald. Het nieuwste deel van de *Tree* wordt dus het meeste bezocht, de oudste pages moeten eerder worden weggeschreven naar harde schijf dan de pages waar het laatst iets is aan toegevoegd.
De *Suffixen* en de *Temp* array zijn de volgende grote datastructuren. Data wordt sequentieel in de *Temp* array gekopieerd, waarna de suffix pointers sequentieel worden overgebracht naar de *Suffixen* array. Het beste taktiek voor de *Temp* array is om de meest recent gebruikte data naar de disk te schrijven. Bij het sorteren worden de pointers gekopieerd naar de suffix array, elke pointer moet in de juiste groep terecht komen. De toegang tot de *Suffixen* array is dus vrij random, in principe kan elke suffix pointer in een andere groep terecht komen. De groepen pointers groeien gelukkig sequentieel aan, suffix pointers worden altijd rechts van de reeds gekopieerde pointers bijgevoegd. Daardoor kan men best de pages waar er het minst recent data is aan toegevoegd wegschrijven naar het geheugen.
De *Sequentie* array is de kleinste van alle datastructuren op de *Stack* na, toegang tot de elementen gebeurt meestal niet sequentieel. Bij de countsort worden alle suffixen één keer opgevraagd, van links naar rechts. Bij het zoeken naar de langste gemeenschappelijke prefix worden die pointers - binnen een groep met dezelfde beginletter - eveneens van links naar rechts opgezocht, ook al zitten er grote gaten tussen twee pointers. Hierdoor is de beste manier om de *Sequentie* array te bufferen de minst recent gebruikte pages naar disk te schrijven.

Buffer grootte

Het is van groot belang dat het beschikbare geheugen tussen de verschillende datastructuren op een goede manier verdeeld wordt. Een goede manier betekent niet noodzakelijk dat het geheugen eerlijk moet worden ingedeeld, door de manier waarop elementen van sommige datastructuren worden opgevraagd, kunnen ze beter volledig in het geheugen worden ingeladen. De *Sequentie* array is zulk een datastructuur, er wordt veel random informatie van opgevraagd en in elk deel van het algoritme speelt de array een sleutelrol. De *Tree* array daarentegen heeft een goede localiteit, dus slechts een fractie van de totale grootte van deze datastructuur moet in het geheugen zitten. We weten dat de *Temp* array sequentieel wordt afgelopen, dus deze heeft minder plaats nodig dan de *Suffixen* array, waar meer random writes op worden uitgevoerd.
Door enkele experimenten uit te voeren stellen de auteurs de volgende in-



Figuur 2.16: Buffer management van TDD

deling van het geheugen voor indien de *Sequentie* array groter is dan het geheugen. Het minimum aantal pages voor de *Temp* en de *Suffixen* array moet $|\Sigma|$ zijn, tijdens de sorteringsfase worden de suffixen één voor één naar de juiste groep geschreven in de de *Suffixen* array. Indien we te maken hebben met een grote hoeveelheid suffixen gaan die $|\Sigma|$ groepen elk in een andere page zitten. Dus wanneer er minder dan $|\Sigma|$ pages voor de *Suffixen* array gebruikt worden weten we dat er dikwijls pages van en naar de harde schijf moeten worden geschreven.
De *Tree* array heeft minimaal twee pages nodig, zo kan men nog toegang krijgen tot een parent node die mogelijk naar een vorige page is geschreven en daarna op de *Stack* is gepushed om later verder uitgewerkt te kunnen worden. Deze situatie komt volgens de auteurs van de paper vrij veel voor en scheelt dus flink wat in de hoeveelheid I/O.
In figuur 2.16 geven we een overzicht hoe het geheugen wordt ingedeeld bij verschillende grootte van *Sequentie*. Ongeacht de grootte van de *Sequentie* worden altijd eerst het minimum benodigde pages voor de *Tree*, *Suffixen* en de *Temp* array gereserveerd. Dit zijn dus respectievelijk 2, $|\Sigma|$ en $|\Sigma|$ pages. Daarna wordt de sequentie ingeladen in het geheugen en overblijvende pages worden in volgorde toegewezen aan *Suffixen* array, de *Temp* array, en de *Tree*.
Uit de experimenten kan men afleiden dat vooral de *Sequentie* array veel wordt bevraagd, als deze niet volledig in het geheugen zit stijgt het disk I/O sterk. Het aantal disk accesses van de *Tree* array is daarentegen bijzonder

klein, naar deze datastructuur wordt vooral geschreven. Wat opvalt bij de experimenten is dat het aantal disk accesses sterk stijgen van zodra er minder dan 20% van de *Sequentie* gebufferd wordt. Het is dus cruciaal om meer dan 20% van bijvoorbeeld de *Sequentie* array in het geheugen te houden om veel snelheidswinst te boeken.

Analyse van het algoritme

Een belangrijk verschil tussen het algoritme van Ukkonen en het PWOTD-algoritme is dat bij Ukkonen de sequentie sequentieel wordt afgelopen maar dat de suffix tree in opbouw semi-random wordt doorlopen door het gebruik van suffix links. Bij TDD is het juist omgekeerd, de suffixen in de sequentie worden random opgevraagd terwijl naar de suffix tree sequentieel wordt geschreven. Voor disk-based constructiealgoritmes is er geen grotere kost dan data die random van de harde schijf moet gelezen worden. Dus efficiënt bufferen van pages die random gelezen worden is essentieel. Omdat bij het TDD-algoritme de hoeveelheid te bufferen data veel kleiner is dan bij het algoritme van Ukkonen, zou TDD veel sneller moeten zijn in de praktijk.

Indien er een suffix tree wordt gebouwd over een korte sequentie dan lijkt het aangewezen om een lineair constructiealgoritme zoals dat van Ukkonen te gebruiken in plaats van het TDD-algoritme. Het aantal disk-accesses is meestal de grootste zorg bij een suffix tree constructiealgoritme, elke toegang tot het geheugen wordt als gelijkwaardig gezien. Dit is echter niet juist, omdat werkgeheugen wordt bijgestaan door cache-geheugen.
Het geheugen van een moderne computer is ingedeeld in verschillende niveaus. Algemeen gezien zijn er twee soorten geheugen, er is het gewone werkgeheugen (RAM) maar ook het cache-geheugen wat heel wat kleiner is dan het werkgeheugen maar veel sneller. Het doel van het cache-geheugen is een buffer vormen tussen het werkgeheugen en de relatief snellere processor. Data ophalen uit het cache-geheugen gaat veel sneller dan data uit het werkgeheugen opvragen. Indien de processor data nodig heeft dat in het cache-geheugen aanwezig is noemt men dit een cache-hit. Wanneer data niet aanwezig is in de cache een cache-miss, de data moet dan uit het werkgeheugen worden opgehaald.
Meestal bestaan er twee soorten cache, in één deel worden instructies geladen, in het andere pure werkdata. Om data efficiënt te bufferen bestaan er verschillende technieken. Meestal komt het er op neer dat data die adresgewijs vlak naast opgevraagde data gesitueerd is, sneller wordt opgevraagd dan andere data in het geheugen. Veel programma's doen bewerkingen met arrays van data die sequentieel worden afgelopen. Dus als data wordt opgevraagd uit het werkgeheugen wordt afhankelijk van het caching schema een aantal naburige datablokken naar de cache gekopieerd.
Dit heeft dus tot gevolg dat op een moderne computer een array sneller

gaat worden doorlopen dan een gelinkte lijst waarvan de elementen over het geheugen liggen verspreid. Het algoritme van Ukkonen heeft dus last van dit probleem omdat de suffix tree gaat doorlopen via suffix links. Data waar de suffix link naar verwijst zit zelden in de cache. Indien men een algoritme gebruikt dat zijn data anders bewaart, in een arraystructuur bijvoorbeeld, kan het in de praktijk heel wat sneller werken. Een voorbeeld hiervan is de array-voorstelling van het algoritme van Ukkonen of McCreight, alle kinderen van een node worden gebundeld in een array van vaste grootte. Het probleem met deze voorstelling is dan weer dat ze veel plaats inneemt wat dan weer een langere uitvoeringstijd tot gevolg heeft.

Dus zelfs bij de constructie van kleinere suffix trees die volledig in het geheugen passen kan een algoritme met een hogere tijdscomplexiteit toch een beter resultaat neerzetten door op een efficiënte manier van het cache-geheugen gebruik te maken. Zoals in de analyse van het TDD-algoritme al is aangeduid, veel van de datastructuren worden sequentieel gelezen of geschreven. Enkel de *Sequentie* array zelf wordt random gelezen.

2.7.2 TOP-Q

In [BH04] beschrijven Bedathur en Haritsa een methode om suffix trees te maken van zeer lange sequenties via het lineaire algoritme van Ukkonen [Ukk92], wat we al hebben besproken in hoofdstuk 2.4. Dankzij een uitgekiend buffer management en een ietwat afwijkende opslag van de suffix tree in secondary memory hebben ze de constructietijd van suffix trees weten te verkleinen. Hierdoor wordt een algoritme dat op het eerste zicht ongeschikt is om gebruikt te worden voor de constructie van een suffix tree die niet in het werkgeheugen past, toch bruikbaar voor bijzonder grote sequenties.

Node access patterns

Het buffer management systeem heet *TOP-Q*, het houdt rekening met de meest waarschijnlijke manier waarop een suffix tree wordt doorlopen tijdens de constructie. Tijdens de constructie worden reeds opgebouwde stukken van de boom opnieuw bezocht via suffix links. Het volgen van die suffix links heeft als gevolg dat de boom schijnbaar random wordt doorlopen. Ook al kan niet worden voorspeld welke de eerstvolgende nodes zijn die zullen worden opgevraagd, toch heeft hun onderzoek aangetoond dat sommige nodes meer worden opgevraagd dan andere. Als we kijken hoe bij het algoritme van Ukkonen werkt bij het zoeken naar de volgende locatie om een suffix toe te voegen, zien we dat er nadat een suffix is toegevoegd er maximaal één node naar boven wordt geklommen, de suffix link wordt gevolgd waardoor het path met 1 wordt verminderd, waarna vanuit deze node de juiste locatie wordt gezocht om de volgende suffix toe te voegen. Het is duidelijk dat

hoger liggende nodes ook meer zullen worden opgevraagd dan nodes die aan het uiteinde van de boom gesitueerd zijn.
De beste taktiek is dus om de hoger liggende nodes te bufferen, om op deze manier de toegang naar de nodes in het algemeen sneller te laten verlopen. Bij de constructie van een suffix tree gaan nodes echter ook van diepte veranderen omdat hoger liggende edges gesplitst kunnen worden. Men kan de diepte per node gaan bijhouden, maar het probleem hiermee is dat indien een hogerliggende edge wordt gesplitst, alle dieptes van nodes onder die edge moeten worden aangepast. Het gevolg is uiteraard dat de constructietijd serieus gaat oplopen. Gelukkig kan de diepte van een node efficiënt worden geschat dankzij volgende observaties.

Stel dat we een symmetrische Bernoulli generator hebben. In dit model worden symbolen van het alfabet onafhankelijk van elkaar gekozen, zodat op een willekeurige positie van de string elk karakter dezelfde kans heeft om voor te komen. Bij een assymmetrisch model hebben sommige karakters meer kans om voor te komen. Een symmetrisch model heeft daarentegen de eigenschap dat substringen van dezelfde lengte evenveel kans hebben een deel van de sequentie te zijn. Stel dat s een subtring is van de sequentie S, dan hangt het aantal substringen elders in S die een gemeenschappelijke prefix hebben met s af van de lengte van S. Dus hoe langer een substring, hoe meer substringen in S die een gemeenschappelijke prefix hebben met s.

Observatie 1. *Hoe langer een edge is in een suffix tree, hoe meer kans er bestaat dat deze gaat gesplitst worden wanneer er nog suffixen worden toegevoegd in de suffix tree.*

Een edge kan zolang worden opgesplitst totdat er tussen elke twee nodes slechts edges overblijven van slechts één karakter lang. De lengte van de edge geeft dus het maximum aan van de eventuele diepte die een node kan bereiken.

Observatie 2. *De uiteindelijke diepte van een node in een suffix tree van een lange sequentie kan geschat worden dankzij de lengte van zijn pad.*

In bovenstaande redenering staat dat er wordt uitgegaan van een symmetrische Bernoulli verdeling, bij DNA sequenties moet men echter uitgaan van een asymmetrische verdeling. Dit kan tot gevolg hebben dat een node v met label $\sigma(v)$ dat minder frequente symbolen bevat, een lagere kans heeft om gesplitst te worden, waardoor de diepte van de node overschat wordt wanneer zijn pad-lengte $L(v)$ wordt gebruikt. Het onderzoek van Haritsa en Bedathur heeft echter aangetoond dat het verschil tussen de lengte van het pad en het aantal nodes nauwelijks groter werd. Met andere woorden, ook al was er sprake van een asymmetrische verdeling toch is bij een voldoende lange sequentie de lengte van het path een goede schatting voor de diepte van een node.

Ontwerp van TOP-Q

Voordat we de buffer strategie van TOP-Q uitleggen, eerst een woordje uitleg over de voorganger van TOP-Q, TOP. Dit laatste systeem gebruikt de observaties dat hoger liggende nodes meer bezocht worden. De suffix tree wordt opgeslagen op de harde schijf, elke page bevat een collectie nodes van de suffix tree, waarbij interne en blad-nodes niet in dezelfde page zitten. Om het gebruikte geheugen minimaal te houden wordt de pad-lengte niet per node bijgehouden, maar per page. De gemiddelde pad-lengte van een page wordt bewaard per page, omdat deze waarde voor elke node invariant is, kan dit gemiddelde worden berekend bij het committen van een page naar de disk. De pages met de nodes die gemiddeld het kortste pad hebben worden zoveel mogelijk in het geheugen bijgehouden. Vandaar de naam TOP, om aan te duiden dat de pages die nodes bevatten die zich de top van de suffix tree bevinden, worden gebufferd.
Het TOP buffering systeem kan worden verbeterd door toegangspatronen op de nodes te onderzoeken en aan de hand daarvan pages anders te gaan bufferen. Als een suffix wordt toegevoegd in de suffix tree dan moet een edge gesplitst worden, waarbij een nieuwe branching node en leaf node en eventueel suffix link moeten worden toegevoegd. Elke node houdt de details van zijn inkomende edge bij, (*begin*, *end*) indexen in de sequentie, lengte en edge-label. Als we een edge $parent(v) \rightarrow v$ splitsen worden volgende stappen uitgevoerd:

1. Maak een nieuwe branching node, v' en een leaf node l,
2. vul de inkomende edge details voor v' en l in,
3. update de inkomende edge details voor v,
4. v' wordt geset als het kind van $parent(v)$ en v is nu een kind van v'.

Dit soort correlaties tussen het opvragen van nodes werd onderzocht door [BH04] en zij kwamen tot de constatatie dat pages waarin juist een node was geupdate snel terug werd opgevraagd. Toch bufferde TOP de pages waarnaar er actieve referenties waren, samen met de pages die veel nodes bevatten die hoog in de suffix tree zitten. Het probleem is dat tijdens de uitvoering van de vier bovenstaande stappen enkel v gegarandeerd gebufferd is, terwijl het onzeker is of de page waarin $parent(v)$ zit in het geheugen is geladen. Naar $parent(v)$ bestaat immers geen actieve referentie.
Daarom werd in TOP-Q een extra buffer gebruikt die pages bewaart die pas uit de buffer pool - geïmplementeerd als een heap - zijn gegooid. Deze buffer pool wordt geïmplementeerd als een FIFO-array.

Suffix tree nodes worden opgeslagen in twee soorten pages, pages die enkel interne nodes bevatten en pages met enkel leaf nodes. In pages met interne

nodes kunnen minder nodes worden opgeslagen dan in pages met leaf nodes, maar pages met interne nodes worden veel meer opgevraagd omdat ze nodig zijn om een plaats voor de volgende suffixen te localiseren. In de buffer pool worden dus meer pages bewaard met interne nodes dan pages met leaf nodes.

Men heeft het TOP-Q buffer management schema vergeleken met twee andere schema's: LRU en 2Q. LRU hebben we reeds besproken in hoofdstuk 2.7.1, indien er een page moet vervangen worden, wordt de minst recent gebruikte page uit de buffer pool weggeschreven. Het buffer management schema 2Q is een verbetering van het LRU/2 schema, waarbij de hoogste prioriteit wordt gegeven aan pages die als tweede-recentste zijn opgevraagd [JS94]. Buffers met de laagste prioriteit worden uit de buffer pool verwijderd. De verbetering in 2Q is dat het in constante tijd kan berekenen welke pagina moet worden weggeschreven, in plaats van in $\log z$ tijd, met z de grootte van de priority queue. Volgens de testresultaten van de makers van TOP-Q, is LRU het traagste buffer management schema en presteert 2Q juist iets beter dan LRU. TOP-Q zelf komt als snelste systeem uit de bus, al is de snelheidswinst niet spectaculair.

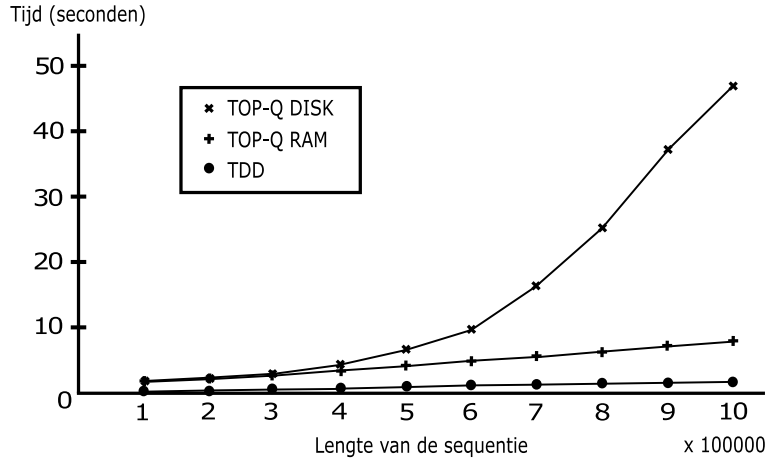
Voorstelling van de suffix tree

In de paper worden twee verschillende manieren besproken om de suffix tree voor te stellen, de array-voorstelling en de gelinkte lijst-voorstelling. We hebben deze methodes al besproken in hoofdstuk 2.6.
Uit experimenten is gebleken dat de gelinkte lijst veel meer I/O tot gevolg heeft dan de implementatie op basis van een array. Deze overhead is te wijten aan het volgen van de pointers om een lijst met siblings te doorlopen. In de array-gebaseerde implementatie zitten alle kinderen in aangrenzende geheugenlocaties en in dezelfde page. Dus ook al worden er procentueel meer pages van de gelinkte lijst implementatie gebufferd, toch is de array gebaseerde methode sneller.
Zelf hebben we deze experimenten niet meer kunnen herhalen omdat het algoritme dat steunt op de gelinkte lijst implementatie niet wou werken. In onze verdere tests werken we dus met de volgens de auteurs optimale array gebaseerde methode.

2.7.3 Vergelijking van TOP-Q en TDD

Om TOP-Q en TDD te vergelijken hebben we het chromosoom 1 uit het menselijk genoom gebruikt. We zijn geïnteresseerd hoe snel TOP-Q en TDD suffix trees kunnen construeren die al dan niet in het werkgeheugen passen. Eerst hebben we korte stukken uit het chromosoom getest, sequenties tussen 100.000 en 1.000.000 base-pairs. De resultaten staan in figuur 2.17. Alle testen zijn uitgevoerd op een Intel centrino 1.7 processor met 512MB geheugen.

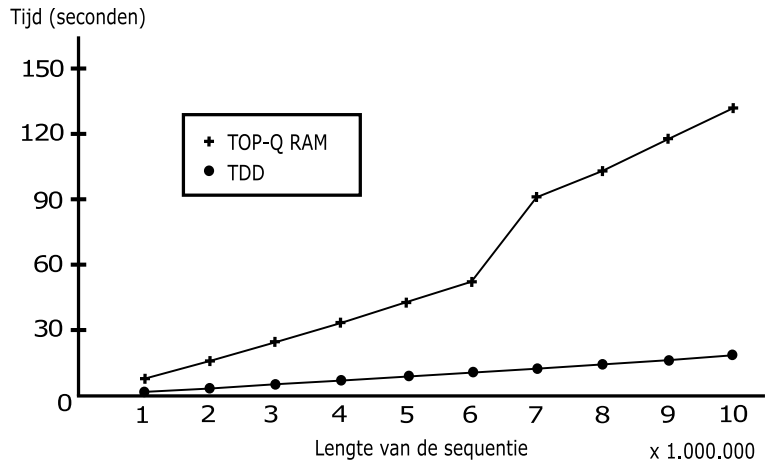
De suffix tree van TOP-Q kan gebaseerd zijn op gelinkte lijsten of arrays, omdat de gelinkte lijst voorstelling niet wou werken en de array gebaseerde suffix tree volgens de auteurs toch het snelste zou zijn, hebben we voor deze laatste implementatie gekozen.
Wat bij het testen onmiddellijk opviel was dat TOP-Q veel trager was dan TDD. Tijdens de constructie van de suffix tree door TDD werd bij de korte sequenties de harde schijf niet aangesproken, pas wanneer het algoritme klaar was werd de suffix tree naar de harde schijf geschreven. TOP-Q daarentegen was zelfs bij de kleinste sequenties volop naar de harde schijf (DISK) aan het schrijven. Daarom hebben we TOP-Q de suffix trees enkel in het werkgeheugen (RAM) laten construeren, wat veel snelheidswinst tot gevolg had. Het wegschrijven van de suffix tree uit het werkgeheugen naar de harde schijf duurt bij TDD slechts enkele seconden, zelfs bij sequenties van 1.000.000 base-pairs. Vermoedelijk worden tijdens het uitvoeren van TOP-Q twee verschillende groepen pages met interne en leaf nodes op de harde schijf aangesproken, zodat er flink wat seek-time verloren gaat als er constant data naar de twee groepen wordt geschreven.
Het is duidelijk dat voor korte sequenties waarvan de suffix tree in het werkgeheugen kan worden geconstrueerd, TDD sneller is dan TOP-Q . De reden hiervoor is waarschijnlijk dat TDD beter gebruik maakt van het cachegeheugen, terwijl TOP-Q suffix links moet volgen waarvan de data bijna nooit in de cache zit.



Figuur 2.17: Vergelijking TOP-Q en TDD voor korte sequenties

Vervolgens hebben we langere sequenties getest, sequenties met tussen de 1.000.000 en 10.000.000 base-pairs. Normaal gezien passen de suffix trees van sequenties van deze grootte nog in het werkgeheugen, maar bij TOP-Q werd al extra swap-geheugen aangesproken vanaf sequenties met een lengte van 7 miljoen base-pairs. In figuur 2.18 is op deze plaats een duidelijke knik in de grafiek te zien. TDD wachtte vanaf een sequentie grootte van 9 miljoen base-

pairs niet meer tot de volledige suffix tree was geconstrueerd, maar begon al eerder stukken van de suffix tree naar de harde schijf te schrijven. Ook al werden stukken suffix tree weggeschreven tijdens de constructie, had dit nauwelijks een weerslag op de snelheid van TDD. Jammer genoeg konden we met TDD geen suffix tree maken van een sequentie van meer dan 50 miljoen base-pairs, het programma crashte bij sequenties van dat formaat. We geven nog mee dat het construeren van een suffix tree van een sequentie van 20 miljoen base-pairs TDD 46 minuten kostte.



Figuur 2.18: Vergelijking TOP-Q en TDD voor langere sequenties

Van zodra de kaap van de 11 miljoen base-pairs was overschreden begon TOP-Q veel trager te werken. Om de suffix tree van een sequentie van 11 miljoen te construeren was er 2,5 minuten tijd nodig, voor een suffix tree van een sequentie van 12 miljoen base-pairs nam TOP-Q al 9 minuten in beslag. Hierbij moeten we opmerken dat we de suffix tree opbouwen zonder deze op de harde schijf op te slaan. Indien we de suffix tree op harde schijf willen hebben moeten we al 5 minuten wachten voor een sequentie van 2 miljoen base-pairs.

Wat in de testen opviel is het verschil in ruimte ingenomen door de suffix trees gemaakt door TOP-Q en TDD. Als voorbeeld nemen we een sequentie van 1 miljoen base-pairs waarvan we de suffix tree laten construeren door TOP-Q en TDD. De suffix tree opgebouwd door TDD gebruikt 12,9 MB terwijl de suffix tree van TOP-Q 41,7 MB ruimte nodig heeft op de harde schijf. Dit betekent dus dat TOP-Q $42n$ bytes gebruikt en TDD ongeveer $13n$ bytes. Deze grote verschillen zijn een logisch gevolg van de manier waarop de suffix tree wordt voorgesteld, TDD stelt de suffix tree als een array voor zoals in het wotdeager algoritme van Kurtz. TOP-Q stelt de kinderen van nodes voor in arrays met grootte $|\Sigma|$, met als resultaat een suffix tree die enorm veel plaats inneemt. In de lagere niveaus van de suffix tree hebben

nodes in het algemeen minder kinderen, hierdoor gaat dus bijzonder veel ruimte verloren. TOP-Q is in staat om de gelinkte lijst voorstelling van een suffix tree te gebruiken, maar dit wou niet werken. Volgens de auteurs is de array gebaseerde methode trouwens sneller om te construeren, waarschijnlijk omdat het volgen van de pointers tussen de kind-nodes vrij lang duurt.

Wat we niet hebben kunnen vergelijken is de snelheid waarmee exacte matches kunnen worden berekend, het algoritme om exacte matches op te sporen werkte niet bij TDD. Voor TOP-Q hebben we getest hoe snel 10000 queries met grootte respectievelijk 10 en 100 karakters lang, uitgevoerd kunnen worden. Als inputdata hebben we een stuk sequentie van 1 miljoen basepairs uit het menselijk chromosoom 1 gebruikt, de query-sequenties kwamen uit hetzelfde stuk chromosoom zodat er per query-sequentie altijd minstens één match is. De suffix tree was al in het werkgeheugen geladen voor het begin van de test. De verwerking van de 10000 sequenties van 10 karakters lang duurde 26 seconden, voor de sequenties van 100 karakters waren er slechts 7 seconden nodig. Deze verschillen in uitvoeringstijd zijn logisch, query-sequenties van 10 karakters hebben meer kans om ook op een andere locatie toevallig voor te komen. Indien een query-sequentie meerdere keren voorkomt in een sequentie, moeten ook alle bladeren van de subtree onder de laatste node op het path van de query-sequentie worden afgelopen. De bladeren van de subtrees zitten echter niet aaneengesloten opgeslagen, iets wat wel het geval is bij TDD. Dus waarschijnlijk is de locality of reference beter bij TDD, waardoor de uitvoeringstijd vermoedelijk kleiner gaat zijn.

We hadden graag nog TDD met ST-Merge uit hoofdstuk 2.5.1 vergeleken, maar jammer genoeg konden de auteurs ons geen broncode van ST-Merge geven. Daarom geven we hier de conclusies uit hun paper. Het ST-Merge algoritme is speciaal gemaakt om van een sequentie die niet in het werkgeheugen past de suffix tree te construeren. TDD voorziet ook in deze mogelijkheid door pages van en naar de harde schijf te swappen. ST-Merge lost dit probleem op door de sequentie in stukken te breken en van de stukken één voor één de suffix tree te maken. Op het einde worden al deze suffix trees gemerged. De constructie van de aparte suffix trees kan gebeuren door TDD. Dus van zodra de sequentie groter wordt dan het werkgeheugen is het aan te raden het ST-Merge algoritme te gebruiken.

2.8 Toepassingen van suffix trees

2.8.1 Zoeken naar een substring

De meest voor de hand liggende toepassing voor een suffix tree is het zoeken naar een bepaalde substring s in een gegeven sequentie S. Indien men de suffix tree voor $S\$$ heeft gemaakt kan men alle plaatsen in S waar s voorkomt

localiseren in een tijd gelijk aan $O(|s| + m)$ met m het aantal matches in de suffix tree. Men moet er immers rekening mee houden dat indien we een substring van grootte $|s|$ zoeken, er gemiddeld $1/\Sigma^{|s|}$ resultaten gevonden worden. Indien men zeer korte substrings zoekt die een groot deel van de boom als resultaat hebben doet men er, afhankelijk van de voorstelling van de suffix tree, beter aan de oorspronkelijke sequentie S serieel te scannen om zo alle voorkomens van die substringen te localiseren.

2.8.2 Langste herhaalde substring

Indien we voor een sequentie S de langst herhaalde substring willen zoeken moeten we in de overeenkomstige subtree de laagste branching node zien te localiseren. Het path van deze node vormt dan de langste herhaalde substring van S. We kunnen deze vinden door de boom in breath-first orde te doorlopen, wat in worst-case tijdscomplexiteit overeen komt met de lengte van de sequentie S.

2.8.3 Langste gemeenschappelijke substring

De langste gemeenschappelijke substring van twee stringen S_1 en S_2 kan op twee manieren gevonden worden. De eerste manier bestaat erin om een gegeneraliseerde suffix tree te bouwen uit alle suffixen van zowel S_1 als S_2. Elke branching node wordt dan gelabeld om aan te duiden of ze tot S_1, S_2 of beide behoort. Vervolgens wordt het algoritme van de langste herhaalde substring toegepast, de diepste branching node die zowel een label van S_1 als S_2 heeft, is de langste gemeenschappelijke substring van S_1 en S_2.
De tweede manier is om een gewone suffix tree te maken van de string $S_1\$S_2\#$, waarbij \$ en # niet voorkomen in S_1 en S_2. De langste gemeenschappelijke substring wordt dan gevormd door het path van de laagste branching node die zowel een suffix $\alpha\$\beta\#$ als een suffix $\gamma\#$ als kinderen heeft.

Hoofdstuk 3

De suffix array

Ook al is de suffix tree één van de belangrijkste datastructuren bij sequentie analyse toch blijft de plaats die een normale suffix tree inneemt erg groot. Qua geheugengebruik bestaan er efficiëntere datastructuren dan de suffix tree. Een valabel alternatief is de suffix array die slechts 4n bytes gebruikt in zijn basisvorm. De suffix array werd in 1989 voorgesteld door Udi Manber en Gene Myers in [MM93].
Een suffix array is niets meer dan een array met indexen die wijzen naar suffixen van de sequentie. De suffixen zijn lexicografisch gesorteerd, elke index wijst naar een unieke positie (suffix) in de sequentie. Als voorbeeld berekenen we de suffix array van de sequentie `atcacccttca$`. In principe moet de sequentie niet worden beëindigd met een $-teken, maar omdat we enkele algoritmes gaan beschrijven die wel een $ toevoegen doen we dit hier ook.
In tabel 3.1 staat de sequentie met zijn indexen, deze indexen worden in tabel 3.2 lexicografisch gesorteerd. De suffix array in tabel 3.3 wordt gevormd door deze gesorteerde suffixen.
Dankzij een suffix tree kunnen we in lineaire tijd volgens de lengte van de query vaststellen of een query-sequentie in de sequentie voorkomt. Bij een suffix array moeten we een binair zoek algoritme toepassen. Doordat de indexen lexicografisch gesorteerd zijn kunnen we door het zoekdomein recursief in twee te splitsen en telkens het deel te selecteren waarin er sequenties kunnen zitten die de gezochte substring als prefix hebben, in $\log n$ tijd de posities vinden van de gevraagde query-sequentie, waarbij n de lengte is van de sequentie waarvan de suffix array is gebouwd.

Index	0	1	2	3	4	5	6	7	8	9	10	11
Sequentie	a	t	c	a	c	c	c	t	t	c	a	$

Tabel 3.1: De sequentie `atcacccttca$`

Suffixen	Index	Gesorteerde suffixen	Index
atcacccttca$	0	acccttca$	3
tcacccttca$	1	atcacccttca$	0
cacccttca$	2	a$	10
acccttca$	3	cacccttca$	2
cccttca$	4	ca$	9
ccttca$	5	cccttca$	4
cttca$	6	ccttca$	5
ttca$	7	cttca$	6
tca$	8	tcacccttca$	1
ca$	9	tca$	8
a$	10	ttca$	7
$	11	$	11

Tabel 3.2: Sorteren van de suffixen

3	0	10	2	9	4	5	6	1	8	7	11

Tabel 3.3: De suffix array van sequentie `atcacccttca$`

3.1 Definitie suffix array

Stel dat er een sequentie S met lengte n gegeven is. Noteer dan de suffix S die begint op positie i als volgt: $S_i = S[i..n-1]$. Een suffix array is een lexicografisch gesorteerde array, SA genaamd, die alle suffixen van S bevat. $SA[k]$ is de startpositie van de k-de kleinste suffix in de set S_0,S_1,...S_{n-1}. In de array SA zitten alle suffixen van S op lexicografische volgorde, $S_{SA[0]} < S_{SA[1]} < ... < S_{SA[n-1]}$.
Een string W met een prefix met lengte p wordt weergegeven als W^p, de relatie $<_p$ geeft dan de lexicografische orde aan van de prefixes met lengte p, $W <_p V$ als en slechts als $W^p < V^p$. Andere relaties zoals $\leq_p$, $=_p$, $\neq_p$ en $>_p$ kunnen op een gelijkaardige manier worden gedefinieerd. Uiteraard is - voor een willekeurige grootte p - de SA array geordend volgens de $\leq_p$ relatie.

3.2 Zoeken in een suffix array

Het localiseren van een substring W met lengte $p \leq n$ in de sequentie S kan als volgt gebeuren [MM93]. Definieer $L_W = \min\,(k \,|\, W \leq_p S_{SA[k]}$ of $k = n)$ en $R_W = \max\,(k \,|\, S_{SA[k]} \leq_p W$ of $k = -1)$. Substring W kan enkel gelijk zijn aan substring $S[i..i+p-1]$ als en slechts als $i = SA[k]$ voor een $k \in [L_W,R_W]$. In de suffix array zijn de posities van de suffixen die als prefix W hebben te vinden op de volgende locaties: $SA[L_W]$, $SA[L_W+1]$, ..., $SA[R_W]$. Omdat de suffix array SA in lexicografische ($\leq_p$) volgorde

staat kan een simpel binair zoekalgoritme worden gebruikt om L_W en R_W te localiseren. De tijdscomplexiteit bedraagt $O(\log n)$ vergelijkingen waarbij elke vergelijking $O(p)$ kost. Dus dankzij de suffix array kunnen alle posities van een substring gelocaliseerd worden in $O(P \log n)$ tijd.
Er is een eenvoudige verbetering van bovenstaand algoritme mogelijk indien we de langste gemeenschappelijke prefix (*lgp*) van ($S_{SA[L]}$, W) en (W, $S_{SA[R]}$) bijhouden. We kunnen $l = \text{lgp}(S_{SA[L]}, W)$ en $r = (W, S_{SA[R]})$ als bijproduct verkrijgen wanneer we $S_{SA[L]}$ en $S_{SA[R]}$ vergelijken met W. Initieel krijgt l de waarde van lgp(W, $S_{SA[0]}$) en r de waarde lgp(W, $S_{SA[n-1]}$). Daarna wordt l of r geupdate volgens lgp(W, $S_{SA[M]}$), waarbij M = (L+R)/2. Men kan per iteratie h = min(l,r) vergelijkingen uitsparen, alle suffixen tussen $S_{SA[L]}$ en $S_{SA[R]}$ hebben immers een prefix van h karakters die gelijk is aan de eerste h karakters van W. Jammer genoeg blijft de worst-case tijdscomplexiteit $O(P \log n)$ omdat moet gezocht worden of $S_{SA[M]}$ lexicografisch kleiner of groter is dan W. Een pathologisch geval voor dit algoritme is de sequentie `agggggggggct`.

Men kan deze tijd verbeteren indien er een tabel wordt aangemaakt die informatie bevat over de langste gemeenschappelijke prefix van bepaalde suffixen. We beschouwen tripletten (L, M, R) die kunnen voorkomen bij een binaire zoektocht. Per iteratie wordt elk stuk van een range in twee gesplitst en wordt er verder gezocht binnen één van die ranges. Omdat er in het begin over de volledige range [1..n] wordt gezocht zijn die tripletten gemakkelijk te bepalen. Er bestaan precies $n-2$ zulke tripletten, elk met een uniek middelpunt $M \in [1, n-2]$. Nu maken we twee arrays Llgp en Rlgp waarin de elementen als volgt gedefinieerd zijn, Llgp[M] = lgp($S_{SA[L_M]}$, $S_{SA[M]}$) en Rlgp[M] = lgp($S_{SA[M]}$, $S_{SA[R_M]}$).
Zoals bij het vorige algoritme worden l en r geïnitialiseerd op respectievelijk lgp(W, $S_{SA[0]}$) en lgp(W, $S_{SA[n-1]}$). Stel dat $l \geq r$, dan kunnen we twee verschillende gevallen onderscheiden: Llgp[M] $\geq$ of $< l$. Voor elk geval moeten we bepalen of L_W in de rechter of linker helft van het interval zit en moet l of r geupdate worden. Indien Llgp[M] $\geq l$ betekent dit dat $S_{SA[M]}$ en $S_{SA[L]}$ een zelfde prefix hebben van minstens $l+1$, maar dat $S[SA[L]+(l+1)]$ en $W[l+1]$ verschillend zijn. Daarom berekenen we de lgp van de suffixen van $S_{SA[M]+l}$ en W_l. Het resultaat hiervan tellen we op bij l en bewaren het in variabele m. Indien $m = P$ of $W[m] \leq S[SA[M]+m]$ dan nemen we het linkse interval, indien $W[m] > S[SA[M]+m]$ het rechtse. Het tweede geval is wanneer Llgp[M] $< l$, dan is het duidelijk dat W in het linkse interval gezocht moet worden, r krijgt dan ook de waarde Llgp[M]. Het geval waarbij we uitgaan van $l < r$ is gelijkaardig.
Het gebruik van de tabellen Llgp en Rlgp reduceert het aantal karakters dat vergeleken moet worden tot maximaal $2p$, er is immers slechts één mismatch per iteratie mogelijk. Het aantal iteraties blijft $\lceil \log_2(n-1) \rceil$, waardoor de tijdscomplexiteit $O(p + \log n)$ bedraagt.

Algoritme 5. *Het $O(p + \log n)$ zoek-algoritme.*

$l := \text{lgp}(S_{SA[0]}, W)$
$r := \text{lgp}(S_{SA[n-1]}, W)$
if($l = p$ || $W[l] \leq S[SA[0] + l]$)
    $L_W := 0$
else if($r = p$ || $W[r] \geq S[SA[n-1] + r]$)
    $L_W := n$
else {
    $(L, R) := (0, n-1)$
    while($R - L > 1$) {
        $M := (L+R)/2$
        if($l \geq r$)
            if($Llgp[M] \geq l$){
                $m := l + \text{lgp}(S_{SA[M]+l}, W_l)$
            else
                $m := Llgp[M]$
        } else {
            if($Rlgp[M] \geq r$)
                $m := r + \text{lgp}(S_{SA[M]+r}, W_r)$
            else
                $m := Rlgp[M]$
        }
        if($m = P$ || $W[m] \leq S[SA[M] + m]$)
            $(R,r) := (M,m)$
        else
            $(L,l) := (M,m)$
    }
    $L_W := R$
}

3.3 Sorteren

Het belangrijkste deel van het algoritme om een suffix array te maken is het sorteren van de suffixen. In [MM93] wordt het sorteren gedaan in $\lceil \log_2(n+1) \rceil$ stappen. In de eerste stap worden de suffixen ingedeeld in buckets volgens hun eerste symbool. Vervolgens wordt in elke volgende stap de buckets verder onderverdeeld, telkens verdubbelt het aantal karakters waarop gesorteerd wordt. Voor de eenvoud zullen we deze stappen nummeren met 1, 2, 4, 8, ..., om zo het aantal karakters waarop gesorteerd wordt

aan te duiden. Dus in de Hde stap worden de suffixen gesorteerd volgens de $\leq_H$-ordening.
In de eerste stap worden dus alle suffixen ingedeeld in een bucket zodat elke bucket alle suffixen bevat die met hetzelfde symbool beginnen. Het resultaat wordt in de array SA opgeslagen. Neem twee suffixen S_i en S_j die na stap H in dezelfde bucket zitten, we weten dan dat $S_i =_H S_j$. In de volgende stap moeten S_i en S_j gesorteerd worden op de volgende H karakters. Omdat we met suffixen te maken hebben weten we dan de volgende H symbolen van S_i en S_j precies dezelfde zijn als de eerste H symbolen van respectievelijk S_{i+H} en S_{j+H}. Met andere woorden, we kennen de $\leq_H$-ordening al tussen S_{i+H} en S_{j+H}.
Na stap H beginnen we met de eerste bucket, deze moet de kleinste suffixen hebben volgens de $\leq_H$-ordening. Neem S_i als eerste suffix in die eerste bucket ($SA[k] = i$) waarvoor geldt dat $i - H \geq 0$ (k wordt vanaf 0 geïncrementeerd totdat een S_i gevonden is). Omdat S_i met prefix met lengte H de kleinste string heeft volgens de $\leq_H$-ordening is het logisch dat S_{i-H} de eerste moet worden in de $2H$-bucket. We zetten S_{i-H} op de eerste plaats in zijn bucket. Voor elke bucket moet worden bijgehouden hoeveel suffixen reeds naar voor zijn verplaatst en in $\leq_{2H}$ volgorde zijn gebracht. Bij het bezoeken van de volgende suffixen S_i wordt S_{i-H}, als deze suffix bestaat, op de volgende beschikbare plaats gezet in bucket H. We geven nu een overzicht van de implementatie.

We houden drie integer arrays bij van lengte n: SA, SA^{-1} en *count*. Daarnaast hebben we nog twee boolean arrays BH en $B2H$. Als alle suffixen gesorteerd zijn volgens de eerst H karakters is $BH[i] = 1$ als $SA[i]$ de meest linkse suffix van een H-bucket bevat, dus als $S_{SA[i]} \neq_H S_{SA[i+1]}$. Het algoritme begint met $H = 1$, we kunnen de waarden van de arrays correct invullen door radix sort toe te passen op het eerste symbool van elke suffix. Stel dat SA, SA^{-1} en BH de correcte waardes hebben na het sorteren op de eerste H symbolen, dan sorteren we nu de suffixen volgens $2H$ symbolen. Eerst laten we $SA^{-1}[i]$ wijzen naar de meest linkse cel van de H-bucket die de i-de suffix bevat, in plaats van de precieze plaats van die suffix in de bucket. De array *count* wordt ook op 0 geïnitialiseerd. We lopen door SA van links naar rechts, bucket per bucket. Laat l en r, met $l \leq r$, respectievelijk de linker en rechter boundary van de huidige H-bucket aanduiden. Voor elke i, waarbij $l \leq i \leq r$, wordt count$[SA^{-1}[SA[i] + H]]$ geïncrementeerd en setten we $SA^{-1}[SA[i] + H]$ += *count*$[SA^{-1}[SA[i] + H]]$ - 1 en $B2H[SA^{-1}[SA[i] + H]] = 1$. Dit heeft als effect dat alle suffixen waarvan de symbolen op posities $[H + 1..2H]$ gelijk zijn met de prefix van de huidige H-bucket op de eerste niet bezette plaats in hun bucket komen te staan. De B2H array wordt gebruikt om de suffixen te markeren die van plaats zijn gewisseld terwijl de *count* array aanduidt op welke plaats de volgende suffix moet komen binnen een bucket. Nadat een bucket is gescand worden de B2H velden zo geset dat

ze de linker grenzen van de $2H$ buckets aanduiden. Na het scannen wordt SA geupdate en worden de velden van $B2H$ toegekend aan BH.
Alle stappen kunnen gedaan worden in $O(n)$ tijd en omdat er ten hoogste $\lceil \log_2(n+1) \rceil$ iteraties zijn heeft het sorteren een worst-case tijdscomplexiteit van $O(n \log n)$. Bovenstaande implementatie kan nog verbeterd worden zodat slechts 2 arrays van grootte n nodig zijn. Meer informatie hierover is te vinden in [MM93].
Naast dit algoritme zijn er nog gelijkaardige implementaties, we geven nog het two-stage algoritme van Itoh-Takana en het copy algoritme van Seward.

3.3.1 Itoh-Takana two-stage algoritme

Itoh en Tanaka beschrijven in [IT99] een sorteeralgoritme dat in twee niveaus werkt. Gegeven een alfabet $\Sigma = \{a, b, \ldots, z\}$ wordt count-sort gebruikt om de suffixen initieel in $|\Sigma|$ buckets $B_a, \ldots, B_z$ te plaatsen. Een bucket B_a bevat alle suffixen die beginnen met karakter a, de suffixen zijn dus gesorteerd volgens hun eerste karakter.
Nu beschouwen we twee types binnen een bucket, suffixen van *Type A* waarbij het tweede karakter lexicografisch gezien kleiner is dan het eerste en suffixen van *Type B* waarbij het tweede karakter groter of gelijk is dan het eerste karakter van de suffix. Binnen elke bucket zijn de *Type A* suffixen kleiner dan *Type B*. We sorteren de *Type B* suffixen en plaatsen ze aan de rechterkant van hun bucket. Indien alle *Type B* suffixen gesorteerd zijn, kunnen we gemakkelijk de volgorde van de *Type A* suffixen afleiden. We lopen de array af van rechts naar links. Indien we een suffix S_i tegenkomen kijken we naar suffix S_{i-1}, indien S_{i-1} een *Type A* suffix is kopiëren we deze naar de eerste lege positie in bucket $B_{S[i-1]}$. Het is eenvoudig in te zien dat het algoritme werkt, *Type B* suffixen staan altijd al op hun plaats. Wanneer we een *Type A* suffix S_i tegenkomen weten we dat deze al op zijn plaats moet staan omdat S_{i-1} al gesorteerd is doordat deze zich in een bucket bevindt met suffixen die beginnen met een lexicografisch kleiner karakter. Deze redenering kunnen we doortrekken tot we aan de eerste bucket B_a komen. Deze bucket kan niet anders dan suffixen van *Type B* bevatten die reeds gesorteerd werden.
Het sorteren van de *Type B* suffixen kan via verschillende algoritmes gebeuren, al naargelang de grootte van de bucket. De auteur stelt voor om grote groepen te sorteren via MSD radix sort [MBM93], middelgrote groepen via Bentley-Sedgewick mulikey quicksort en voor kleine groepen insertion sort. Het two-stage algoritme steunt dus op bekende sorteringsmethoden om ruwweg de helft van de suffixen te sorteren en aan de hand van die gesorteerde groep de rest van de suffixen op een snelle manier toe te voegen aan de suffix array.

3.3.2 Seward copy algoritme

Het Seward *copy* algoritme [Sew00] is gelijkaardig aan het Itoh-Takana two-stage algoritme. Door gebruik te maken van count-sort worden de suffixen gesorteerd volgens de eerste twee karakters in de SA-array opgeslagen. We noemen elk deel van SA waarin suffixen met hetzelfde beginkarakter een bucket, een deel met suffixen die beginnen met dezelfde twee karakters een small bucket. Dus SA bevat $|\Sigma|$ buckets die elk $|\Sigma|$ small buckets bevatten, sommige van de (small) buckets kunnen uiteraard leeg zijn.
Het *copy* algoritme sorteert de buckets één voor één, beginnende bij de kleinste bucket en zo verder werkend naar de grootste. Stel dat we het alfabet $\Sigma = \{\text{a, b, } \ldots, \text{ z}\}$ hebben. Om een bucket B_p te sorteren worden de small buckets B_{pa}, B_{pb}, $\ldots$, B_{pz} gesorteerd. Van zodra B_p volledig gesorteerd is kunnen de small buckets B_{ap}, B_{bp}, $\ldots$, B_{zp} in een enkele pass gesorteerd worden. Deze small buckets worden gemerkt zodat ze kunnen worden overgeslagen als de parent bucket wordt gesorteerd. Het is ook mogelijk om een small bucket B_{pp} te sorteren van zodra de small buckets B_{pa}, $\ldots B_{po}$, B_{pq}, $\ldots$, B_{pz} gesorteerd zijn. Vooral bij files waarin veel opeenvolgende karakters hetzelfde zijn is dit algoritme interessant.

3.4 Doubling

Doubling, oorspronkelijk beschreven door Ferragina, Grossi en Vitter in [AFGV97], is een suffix array constructiealgoritme dat in $O(\log n)$ verdubbelingsstappen een suffix array kan berekenen voor een sequentie S met grootte n. De totale tijd dat het algoritme nodig heeft bedraagt $O(n \log^2 n)$. Het idee achter doubling is om namen te berekenen voor alle substringen $S[i, i + 2^h - 1]$ per doubling stap h, voor $0 \leq i < n$ en $h > 0$. De namen van substringen berekend in stap h worden gebruikt om de namen in stap $h + 1$ te berekenen. Eén doubling stap berekent dus namen voor n paren uit de sequentie, indien nodig wordt wordt de sequentie achteraan bijgevuld met \$-tekens.
In de papers over doubling is \$ lexicografisch kleiner dan de andere karakters, maar omdat we bij andere algoritmes \$ als lexicografisch het grootste element beschouwen, nemen we deze zienswijze over in dit hoofdstuk. Voor de werking van het algoritme maakt het weinig uit, indien we in dit hoofdstuk \$ anders beschouwen krijgen we hier uiteraard een andere suffix array wat eerder verwarrend zou werken.

Neem $n_{h,i}$ als naam voor de substring $S[i, i + 2^h - 1]$, dan kunnen we de naam $n_{h+1,i}$ van substring $S[i, i + 2^{h+1} - 1]$ berekenen dankzij de positie van het koppel namen $\langle n_{h,i}, n_{h,i+2^h} \rangle$. Per positie i wordt dus een naam $n_{h,i}$ bijgehouden, we schrijven dit als het tuple $\langle i, n_{k,i} \rangle$. Een naam kan uit maximaal $\lceil \log_2 n \rceil$ bits bestaan. Omdat de substringen in elke iteratie

verdubbelen heeft het algoritme na $q = \lceil \log_2(n+1) \rceil$ iteraties de namen berekend van de twee substringen met lengte 2^q uit $S[1, 2n]$, waarbij $2^q \geq n$. De volgorde tussen twee suffixen $S[i, n]$ en $S[j, n]$ kan worden bepaald door de lexicografische namen van de substringen $S[i, i+2^q-1]$ en $S[j, j+2^q-1]$ te vergelijken. Dus door de tuples $I{:=}\langle i, n_{k,i} \rangle$ te sorteren volgens $n_{k,i}$ verkrijgen we de suffix array voor S.
Indien we dit op ons voorbeeld `atcacccttca$` toepassen krijgen we volgend resultaat:

Stap 1.

index	0	1	2	3	4	5	6	7	8	9	10	11
S	a	t	c	a	c	c	c	t	t	c	a	$

index	3	0	10	2	9	4	5	6	1	8	7
paren	ac	at	a$	ca	ca	cc	cc	ct	tc	tc	tt
namen	0	1	2	3	3	4	4	5	6	6	7

Stap 2.

index	0	1	2	3	4	5	6	7	8	9	10	11	12
S	1	6	3	0	4	4	5	7	6	3	2	$	$

index	3	0	10	2	9	4	5	6	1	8	7
paren	04	13	2$	34	3$	45	47	56	60	62	73
namen	0	1	2	3	4	5	6	7	8	9	10

Merk op dat we na de eerste stap de suffix array toevallig al hebben, dit is uiteraard niet altijd zo. In stap 1 zouden de paren cc en cc met index respectievelijk 4 en 5 omgewisseld worden in stap 2 indien karakter t met index 7 bijvoorbeeld een a was geweest. Verder in hoofdstuk 3.5 werken we een ander voorbeeld uit waarbij paren in de volgende stap wel worden omgewisseld. De suffix array van de sequentie `atcacccttca$` is dus:

$$\text{SA} = \{3,\ 0,\ 10,\ 2,\ 9,\ 4,\ 5,\ 6,\ 1,\ 8,\ 7\}$$

Bij een concrete implementatie wordt een lijst van n tuples gemaakt die elk uit vier componenten bestaan: $\langle n_{h,i}, n_{h,i+2^h}, i, n_{h+1,i} \rangle$, i blijft constant in dit tuple, het duidt namelijk de positie van de suffix in de sequentie S aan. De volgende vier stappen zijn nodig om van stap h naar $h+1$ te gaan:

1. De n tuples moeten eerst gesorteerd worden volgens hun derde component, zodat het i-de tuple van de vorm $\langle *, *, i, n_{h,i} \rangle$ is.

2. Vervolgens wordt de lijst van links naar rechts doorlopen waarbij de informatie van het i-de tuple wordt veranderd in $\langle n_{h,i}, n_{h,i+2^h}, i, NULL\rangle$, waarbij de waarde n_{i+2^h} gehaald wordt uit de vierde component van het $(i+2^h)$-de tuple in de lijst.

3. De lijst met tuples wordt nu gesorteerd volgens de eerste twee componenten.

4. De gesorteerde array wordt nu van links naar rechts doorlopen en elk tuple krijgt als vierde component een nieuwe lexicografische naam. Een tuple dat lexicografisch gezien (volgens de eerste twee componenten) groter is dan zijn linkse buur krijgt als naam een integer die één eenheid hoger ligt. Indien de twee tuples gelijk zijn volgens de eerste twee componenten $n_{h,i}$ en $n_{h,i+2^h}$ krijgen ook hetzelfde nummer als vierde component. De nummering begint bij 0.

Op het einde van deze stappen kan de suffix array worden geconstrueerd door te sorteren op de lexicografische naam (vierde component). De derde component vormt in die volgorde de suffix array.
In de vier bovenstaande stappen wordt tweemaal de volledige sequentie gesorteerd en twee keer wordt er doorheen de sequentie gescand. Omdat deze stappen $\lceil \log_2(n+1) \rceil$ keer worden uitgevoerd, en het sorteren van de koppels in $n \log_2 n$ tijd kan, bedraagt de tijdscomplexiteit van het basisalgoritme daarom $O(n \log^2 n)$.

In [CF99] worden enkele voor de hand liggende verbeteringen voorgesteld om het algoritme in de praktijk sneller te maken. Allereerst kan men verschillende karakters in de sequentie S samennemen, afhankelijk van de grootte van het alfabet. Men kan bijvoorbeeld de namen per vier encoderen in één naam, waardoor het algoritme minder iteraties moet uitvoeren. Per sorteerfase moet wel meer werk worden verricht. Deze variant van doubling noemt men ook wel quadrupling. Een andere voor de hand liggende verbetering is om niet zomaar $O(\log_2 n)$ iteraties uit te voeren, maar te stoppen van zodra alle tuples verschillende namen hebben gekregen. De hoeveelheid iteraties hangt af van de langste gemeenschappelijke prefix, de namen van de tuples zijn pas uniek als er $\log maxlgp$ verdubbelingsstappen zijn geweest. Door deze laatste optimalisatie vermindert de tijdscomplexiteit tot $O(n \log maxlgp \log n)$.
Het algoritme in zijn huidige vorm heeft $4n$ integers nodig (de extra plaats voor het sorteren uitgezonderd) om de suffix tree te construeren, maar we kunnen dit echter reduceren tot $3n$ integers. Een tuple bevat vier integers, maar als we de vier stappen bekijken zien we dat in stap 2 voor het i-de tuple de vierde component niet meer van belang is. In stap 4 zijn de eerste twee componenten niet meer van belang. De vierde component kunnen we dus in stap 4 opslaan op de plaats van de eerste component, in stap 2 moet

dan wel naar de eerste component van het $(i+2^h)$-de tuple gekeken worden.

3.5 Discarding

Het probleem met het doubling algoritme is dat het aantal iteraties afhangt van enkele suffixen die een lange prefix gemeenschappelijk hebben, *maxlgp*'s genoemd. De discarding techniek (discarding betekent negeren) [DMKS05] heeft als doel bepaalde delen van de suffix array die al een unieke naam hebben buiten beschouwing te laten, zodat heel wat minder namen berekend moeten worden. Als een tuple een unieke naam heeft betekent dit dat zijn positie in de suffix array al vast ligt, dus door deze tuples uit te sluiten van de sorteerstappen kan men een serieuze snelheidswinst boeken. Deze tuples volledig uitsluiten gaat niet, ze kunnen in de verdere stappen nodig zijn om te bepalen welke van twee suffixen voor de andere komt. Jammer genoeg geeft deze verbetering van het doubling algoritme geen betere worst-case tijdscomplexiteit.

Om een beter zicht te krijgen op het probleem werken we hier via het doubling algoritme de sequentie `tagtcatagtcaa$` uit waarin een *maxlgp* (`tagtca`) zit van 6 karakters.

Stap 1.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
S	t	a	g	t	c	a	t	a	g	t	c	a	a	$

Index	11	1	7	5	12	4	10	2	8	0	6	3	9
Paren	aa	ag	ag	at	a$	ca	ca	gt	gt	ta	ta	tc	tc
Namen	0	1	1	2	3	4	4	5	5	6	6	7	7

Stap 2.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
S	6	1	5	7	4	2	6	1	5	7	4	0	3	$

Index	11	1	7	5	12	10	4	2	8	0	6	9	3
Paren	0$	17	17	21	3$	43	46	54	54	65	65	70	72
Namen	0	1	1	2	3	4	5	6	6	7	7	8	9

Stap 3.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
S	7	1	6	9	5	2	7	1	6	8	4	0	3	$

Index	11	7	1	5	12	10	4	8	2	6	0	9	3
Paren	0$	10	12	28	3$	4$	56	63	67	74	75	8$	91
Namen	0	1	2	3	4	5	6	7	8	9	10	11	12

Het nieuwe algoritme houdt twee verschillende lijsten bij: een lijst met *afgewerkte* tuples (D) en *niet-afgewerkte* tuples (U) waarvan de plaats in de uiteindelijke suffix tree dus nog niet vast ligt. De tuples in D zijn van de vorm $\langle pos, -1, i\rangle$, waarbij *pos* de uiteindelijke positie is van suffix $S[i, n]$, zodat SA[*pos*]=i. De verzameling U bestaat uit tuples $\langle x, y, i\rangle$ waarbij x en y lexicografische namen zijn en i naar de positie van de suffix in S verwijst. De array D is in het begin leeg, terwijl U tuples van de vorm $\langle 0, S[i], i\rangle$ bevat, met $1 \leq i \leq n$. De teller h wordt op 0 geïnitialiseerd. Per iteratie worden volgende stappen uitgevoerd:

1. Sorteer alle tuples in U volgens hun eerste twee componenten, indien U leeg is ga naar stap 6.

2. Doorloop de array tuples U van links naar rechts, markeer de afgewerkte tuples en geef nieuwe namen aan de tuples in U. Een tuple t is afgewerkt indien de twee buren in U verschillen in ten minste één van hun twee eerste componenten ten opzichte van t. Een afgewerkt tuple krijgt als tweede component de waarde -1. De nieuwe namen voor

de onafgewerkte tuples worden op een andere manier berekend dan in het oorspronkelijke algoritme, de eerste component van een tuple $t = \langle x, y, * \rangle$ wordt geset op (x+c), waarbij c het aantal tuples is dat links van t ligt en de vorm $\langle x, p, * \rangle$ hebben met $p \neq y$.

3. Sorteer U volgens de derde component, volgens de startpositie van zijn suffix.

4. Merge de lijsten U en D volgens de derde component, U bevat nu terug alle tuples terwijl D leeg is.

5. Doorloop U, voor elk onafgewerkt tuple $t = \langle x, y, i \rangle$ neem het volgende tuple $\langle z, *, i + 2^h \rangle$, dat op afstand 2^h ligt van t, en verander t in $\langle x, z, i \rangle$. Een afgewerkt tuple wordt niet bekeken en onmiddellijk gekopieerd in D. Incrementeer h met 1 en ga naar stap 1.

6. Nu D leeg is wordt de array gesorteerd volgens het eerste component van de tuples. In deze volgorde geeft de derde component de suffix array van S.

De tijdscomplexiteit van doubling gecombineerd met discarding blijft jammer genoeg $O(n \log^2 n)$. De space complexiteit van het algoritme hangt af van de gebruikte methode om te sorteren, in [CF99] wordt multiway mergesort gebruik. Omdat elk tuple bestaat uit 3 integers kost het algoritme $12n$ bytes, maar in combinatie met het multiway mergesort verdubbelt dit tot $24n$ bytes.

3.6 Constructie in L stukken

De hoeveelheid geheugen die nodig is om de suffix array van een volledig chromosoom te maken is vrij aanzienlijk, daarom wordt in [CF99] een techniek voorgesteld om de suffix array incrementeel op te bouwen. Stel dat er een algoritme A_{sa} gegeven is dat een suffix array kan maken, en A_{string} een algoritme dat een set strings kan sorteren. Beide algoritmes moeten in extern geheugen kunnen werken. Eerst worden er L suffix arrays geconstrueerd, SA_0, SA_1, ..., SA_{L-1}, elk met grootte n/L. Elke array SA_i houdt de lexicografisch gesorteerde suffixen S_i, $Si + L$, $Si + 2L$, ... bij. De sequentie S wordt zo nodig aangevuld met karakters \$ om S een grootte n te geven, waarbij n een veelvoud van L is. De basisgedachte is om eerst SA_{L-1} te laten construeren door A_{sa} en A_{string}, waarna de andere arrays SA_{L-2}, SA_{L-3}, ..., SA_0 afgeleid kunnen worden door een algoritme dat in extern geheugen integer tripletten kan sorteren.
De suffix array SA_{L-1} wordt in twee stappen gemaakt, eerst wordt een string set $SS = S[L - 1..2L - 2]$, $S[2L - 1..3L - 2]$, ..., $S[n..n + L - 2]$ lexicografisch gesorteerd via A_{string}. Vervolgens wordt een gecomprimeerde

tekst S' van lengte n/L afgeleid van $S[L-1, n+L-2]$ door elke string met de vorm $S[iL-1, (i+1)L-2]$ met $i \geq 1$ te vervangen door zijn rang in de gesorteerde string set SS. Algoritme A_{sa} maakt dan de suffix array SA' van S' en leidt SA_{L-1} daarvan af door $SA_{L-1}[j]$ gelijk te stellen met $(SA'[j] \times L) - 1$, met $j = 1, 2, \ldots, n/L$.
De reden dat er nog een suffix array SA' van S' moet gemaakt worden is dat de string set SS mogelijk strings bevat die meerdere keren voorkomen. Sommige strings $S[iL-1, (i+1)L-2]$ en $S[jL-1, (j+1)L-2]$ met $i \neq j$ kunnen dus gelijk zijn, waardoor hun rang uiteraard ook gelijk is. Op dit moment is het dus nog onduidelijk of suffix S_{iL} kleiner dan wel groter is dan suffix S_{jL}. Daarom moet van S' nog eens de suffix array SA gemaakt worden zodat de suffixen S_{jL}, met $j = 1, 2, \ldots, n/L$, nu in de juiste volgorde staan. Tijdens de constructie van een suffix array moet dus een suffix array van groote n/L worden geconstrueerd. Dit kan uiteraard recursief gedaan worden, maar door de waarde van L groot genoeg te maken kan men de suffix array SA' in het werkgeheugen constueren.
De volgende $L-1$ suffix arrays kunnen geconstrueerd worden dankzij de observatie dat elke suffix $S[i+kL-1, n]$ in SA_i kan gezien worden als de concatenatie van het karakter $S[i+kL-1]$ en de suffix $S[i+kL, n]$ die in SA_{i+1} zit. Dus als SA_{i+1} is bekend kan de volgorde tussen $S[i+kL-1, n]$ en $S[i+hL-1, n]$ worden berekend door volgende twee integer paren met elkaar te vergelijken: $\langle S[i+kL-1], pos_{i+kL-1} \rangle$ en $\langle S[i+hL-1], pos_{i+hL} \rangle$, waarbij pos_s de positie is van de suffix S_s in SA_{i+1}. Dit betekent dat de constructie van SA_i inhoudt dat van zodra SA_{i+1} bekend is er enkel n/L tuples gesorteerd moeten worden.

We werken het algoritme uit aan de hand van onze voorbeeldsequentie `atcacccttca$`, waarbij $L = 2$.

Index	0	1	2	3	4	5	6	7	8	9	10	11
S	a	t	c	a	c	c	c	t	t	c	a	$

SS = {(t,c),(a,c),(c,c),(t,t),(c,a),($,$)}
SS (gesorteerd) = {(a,c),(c,a),(c,c),(t,c),(t,t),($,$)}
rang S = {4, 1, 3, 5, 2, 6}
SA' = {2, 5, 3, 1, 4, 6}
SA_1 = {3, 9, 5, 1, 7, 11} ((×2)-1)

Index	0	2	4	6	8	10
Koppels	(a,4)	(c,1)	(c,3)	(c,5)	(t,2)	(a,6)

SA_0	0	10	2	4	6	8
Koppels	(a,4)	(a,6)	(c,1)	(c,3)	(c,5)	(t,2)

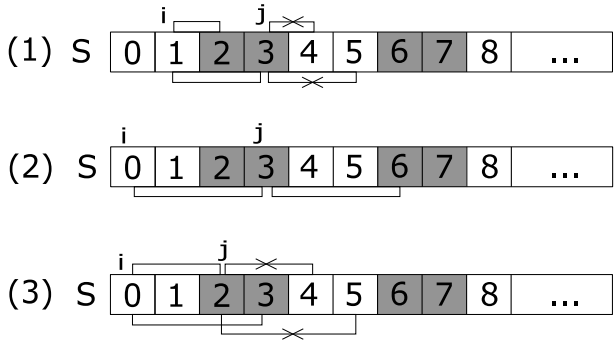
Nu hebben we dus twee suffix arrays, $SA_0 = \{0, 10, 2, 4, 6, 8\}$ en $SA_1 = \{3, 9, 5, 1, 7, 11\}$. In [CF99] wordt geen algoritme gegeven om de suffix arrays samen te voegen, men stelt dat het afhangt van de applicatie of de suffix array in één stuk of gedistribueerd wordt opgeslagen. We kunnen echter zelf een algoritme beschrijven om de suffix arrays om te vormen tot één samenhangende suffix array.

We kunnen twee suffix arrays SA_p en SA_q, met $0 \leq p < q < L$ niet zomaar samenvoegen. Gegeven een element i uit SA_p en j uit SA_q moeten we uitzoeken welk van de twee lexicografisch groter is dan de andere, dus of $S_{SA_p[i]} <_l S_{SA_q[j]}$ dan wel $S_{SA_p[i]} >_l S_{SA_q[j]}$. Van twee willekeurige suffixen uit SA_{L-1} en SA_{L-2} kunnen we niet onmiddellijk de relatieve volgorde te weten komen. Van zodra we SA_{L-2} en SA_{L-1} hebben samengevoegd in array $SA_{L-2} + SA_{L-1}$ kunnen we SA_{L-3} iets efficiënter toevoegen. We moeten immers weten of gegeven een suffix i uit $SA_{L-2} + SA_{L-1}$ kleiner of groter is dan een suffix j uit SA_{L-3}. Dit kunnen we te weten komen door het eerste karakter van suffix i te vergelijken met het eerste karakter van suffix j, indien deze karakters gelijk zijn moeten we volgende twee gevallen beschouwen. We weten dat $SA_{L-2} + SA_{L-1}$ alle suffixen i mod $(L-1)$ en i mod $(L-2)$ bevat. Als i mod $(L-2) = 0$ dan is de volgorde van i en j gelijk aan de volgorde van suffixen $i+1$ en $j+1$. We weten immers dat $i+1$ mod $(L-1) = 0$ en $j+1$ mod $(L-2) = 0$, de relatieve volgorde van suffixen i en j hangt dus af van de volgorde van suffixen $i+1$ en $j+1$ die beide in verzameling $SA_{L-2} + SA_{L-1}$ zitten. Het tweede geval is iets ingewikkelder, indien i mod $(L-1) = 0$ dan kunnen we de relatieve volgorde van $i+1$ en $j+1$ niet zomaar bepalen. Indien $L > 3$ bestaat er geen k zodat $i+k$ en $j+k$ beide in verzameling $SA_{L-2} + SA_{L-1}$ zitten. Indien L=3 dan lukt het wel, dan zitten $i+2$ en $j+2$ beide in de verzameling $SA_{L-2} + SA_{L-1}$. Indien de twee eerste karakters van i en j gelijk zijn kunnen we de onderlinge volgorde bepalen dankzij de onderlinge volgorde van $i+2$ en $j+2$ in de verzameling $SA_{L-2} + SA_{L-1}$.
Het mergen van suffixen uit SA_{L-z-1} aan $SA_{L-z} + \ldots + SA_{L-1}$ gebeurt op een gelijkaardige manier. Ook hier staat niet vast of er een k bestaat zodat we kunnen bepalen of een suffix i uit SA_{L-z-1} kleiner of groter is dan een suffix j uit verzameling $SA_{L-z} + \ldots + SA_{L-1}$ zodat $i+k$ en $j+k$ beide in $SA_{L-z} + \ldots + SA_{L-1}$ zitten. Per L kan wel een z gevonden worden zodat een willekeurige suffix i uit SA_{L-z-1} kan vergeleken worden met een willekeurige suffix j uit $SA_{L-z} + \ldots + SA_{L-1}$ zodat suffixen $i+k$ en $j+k$ beide in $SA_{L-z} + \ldots + SA_{L-1}$ zitten.

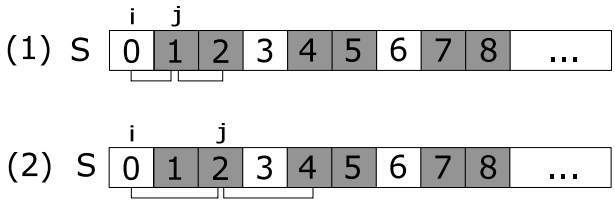
Om het probleem te verduidelijken werken we enkele voorbeelden uit, de situatie in de voorbeelden is simpeler voorgesteld dan in werkelijkheid kan voorkomen. We situeren de suffixen i en j altijd dicht bij elkaar in de buurt, terwijl ze willekeurig ver uit elkaar kunnen liggen. Normaal gezien kan er

dus altijd een veelvoud van L suffixen tussen i en j liggen, maar aan de waarde van k verandert dit niets. In figuur 3.1 nemen we $L = 4$, de suffix arrays SA_2 en SA_3 zijn al samengevoegd in $SA_2 + SA_3$. De suffix i uit SA_1 is aangeduid, net zoals suffix j in SA_3. In figuur 3.1(1) zien we dat we geen enkele k kunnen vinden zodat suffix $i + k$ en suffix $j + k$ beide in verzameling $SA_2 + SA_3$ zitten. In figuur 3.1(2) en 3.1(3) tonen we aan dat eerst de suffix array SA_0 toevoegen evenmin altijd efficiënt kan gebeuren. Als we i en j kiezen zoals in figuur 3.1(2) kunnen we $k = 3$ nemen, suffixen $i+3$ en $j+3$ zitten beide in verzameling SA_2+SA_3 zodat we hun relatieve volgorde kunnen gebruiken om de volgorde van i en j snel te kunnen bepalen indien de eerste 3 karakters van suffixen i en j gelijk zouden zijn. Maar in figuur 3.1(3) hebben we een situatie waarbij er geen k bestaat, eerst SA_0 toevoegen heeft dus geen effect.
Tot slot hebben we enkele voorbeelden gemaakt waarbij de verzameling $SA_{L-z+1}+\ldots+SA_{L-1}$ groot genoeg is om de verzamelingen $SA_0, \ldots, SA_{L-z}$ efficiënt toe te voegen. In figuur 3.2 staat het voorbeeld met $L = 3$, waarbij de suffixen uit SA_1 en SA_2 al zijn samengevoegd. Voor een willekeurige i uit SA_0 en j uit $SA_1 + SA_2$ kan men altijd een $k = 1$ of $k = 2$ vinden zodat suffixen $i + k$ en $j + k$ in $SA_1 + SA_2$ zitten. Figuur 3.3 illustreert het geval waarbij $L = 5$. Hierdoor is duidelijk dat $z = 3$ indien $L = 5$, zodat voor gegeven een willekeurige suffix i uit $SA_0, \ldots, SA_{L-z-1}$ en een suffix j uit $SA_{L-z} + \ldots + SA_{L-1}$ er altijd een k kan gevonden worden zodat $i + k$ en $j + k$ beide in de samengevoegde suffix array $SA_{L-z} + \ldots + SA_{L-1}$ zitten.

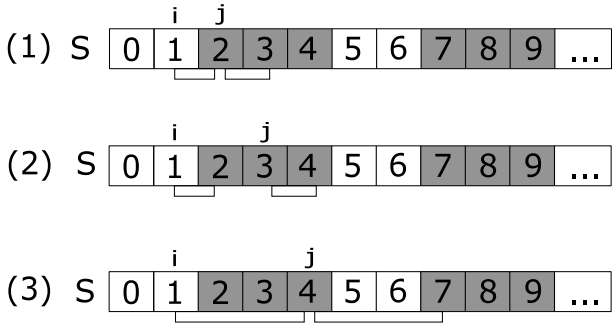
In het volgend hoofdstuk worden deze observaties gebruikt om een lineair algoritme te construeren.



Figuur 3.1: Constructie van de suffix array in 4 stukken



Figuur 3.2: Constructie van de suffix array in 3 stukken



Figuur 3.3: Constructie van de suffix array in 5 stukken

3.7 Lineaire constructie

In 2003 werd in drie onafhankelijke onderzoeken vastgesteld dat suffix arrays rechtstreeks in lineaire tijd kunnen worden geconstrueerd [KS03, KSPP03, KA03]. Theoretisch gezien kon men al veel eerder een suffix array in lineaire tijd construeren, namelijk door de lineair geconstrueerde suffix tree in depth-first volgorde te doorlopen.
J. Kärkkäinen en P. Sanders beschrijven in [KS03] het *skew* algoritme om in lineaire tijd een suffix tree te construeren. Het algoritme gaat uit van een divide-and-conquer aanpak. De suffixen worden in twee groepen ingedeeeld, een groep suffixen die positie i mod $3 \neq 0$ hebben en een groep die positie i mod $3 = 0$ heeft. Eerst worden de suffixen die beginnen op positie i mod $3 \neq 0$ gesorteerd. In de tweede stap worden de suffixen met startpositie i mod $3 = 0$ gesorteerd, gebruik makend van de gesorteerde suffixen uit de vorige stap waardoor het sorteren sneller verloopt. Uiteindelijk worden de twee lijsten samengevoegd tot de suffix array van de volledige sequentie.
Het idee achter het *skew* algoritme wordt verder uitgewerkt in [JKB03], waarbij het concept *difference cover* wordt voorgesteld . Het difference cover 3 (DC3) algoritme werkt zoals het *skew* algoritme, maar het kan worden gegeneraliseerd zodat het voor elke difference cover D modulo v geldt. Hierdoor kan door een bepaalde difference cover D te kiezen space ingeruild worden voor tijd.
We bespreken eerst de basisvoorstelling met DC3, waarbij we het algorit-

me stap voor stap uitwerken aan de hand een voorbeeldsequentie. Daarna bespreken we de tijdscomplexiteit van DC3 waarna we in hoofdstuk 3.8 overgaan tot het bespreken van het begrip difference cover en het algoritme generaliseren.

3.7.1 Difference Cover 3

Zonder eerst te definiëren wat een difference cover precies is werken we het voorbeeld uit, in het volgend hoofdstuk wordt uitgelegd waarom dit algoritme een toepassing is van difference cover 3. De voorbeeldsequentie ziet er als volgt uit:

i	0	1	2	3	4	5	6	7	8	9	10	11
$S[i]$	a	t	c	a	c	c	c	t	t	c	a	$

Allereerst worden de suffixen in drie groepen ingedeeld, voor $k = 0, 1, 2$ definiëren we $B_k = \{i \in [0, n] \mid i \bmod 3 = k\}$. Laat $C = B_1 \cup B_2$ de set sample posities zijn en S_C de set suffixen die beginnen op een positie uit de set sample posities. In ons voorbeeld geeft dit het volgende:

$$B_1 = \{1, 4, 7, 10\},\ B_2 = \{2, 5, 8, 11\},\ \text{C} = \{1, 4, 7, 10, 2, 5, 8, 11\}$$

Nu moeten de suffixen van de sample posities gesorteerd worden. We doen dit door twee stringen R_1 en R_2 te concateneren tot R en deze laatste te sorteren. Hierdoor krijgen we de volgorde van de sample suffixen S_C. Voor $k = 1, 2$ construeren we de stringen

$$R_k = [S[k], S[k+1], S[k+2]][S[k+3], S[k+4], S[k+5]]\ldots \\ [S[maxB_k], S[maxB_{k+1}], S[maxB_{k+2}]],$$

waarvan de karakters tripletten $[S[i], S[i+1], S[i+2]]$ zijn. Laat $\text{R} = R_1 \circ R_2$ de concatenatie zijn van R_1 en R_2. R ziet er in ons voorbeeld dan uit als volgt:

R	[tca]	[ccc]	[ttc]	[a$0]	[cac]	[cct]	[tca]	[$00]
S_C	5	3	6	1	2	4	5	7

Om de suffixen van R te sorteren worden de karakters, we beschouwen de tripletten als karakters, gesorteerd via radix sort. We verkrijgen de set sample suffixen door aan elk karakter een integer waarde te geven die hun onderlinge volgorde uitdrukt, $[S[i], S[i+1], S[i+2]] < [S[j], S[j+1], S[j+2]] \Leftrightarrow {R'}_i < {R'}_j$.
We zien dat er twee karakters hetzelfde zijn, namelijk [tca]. Het gevolg is dat S_C nog niet volledig gesorteerd is, daarvoor kopiëren we S_C naar R'. De suffixen in R' worden vervolgens recursief gesorteerd via DC3. Hiervoor voegen we wel het karakter 0 toe aan R', zodat het aantal karakters een

veelvoud van 3 is. We nemen aan dat het aanroepen van het DC3 algoritme met R' als input de suffix array $SA_{R'} = (8, 3, 4, 1, 5, 0, 6, 2, 7)$ teruggeeft. Nu de sample suffixen gesorteerd zijn moet er aan elke suffix een rang gegeven worden. Voor elke $i \in C$ noemen we $rank(S_i)$ de rang van S_i in de sample set S_C. Geef $rank(S_{n+1}) = rank(S_{n+2}) = 0$. Voor elke $i \in B_0$ is $rank(S_i)$ niet gedefinieerd.

Index	0	1	2	3	4	5	6	7	8
R'	5	3	6	1	2	4	5	7	0

C	1	4	7	10	2	5	8	11
S_C	5	3	7	1	2	4	6	8

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$rank(S_i)$	$\perp$	5	2	$\perp$	3	4	$\perp$	7	6	$\perp$	1	8	$\perp$	0	0

Nu de sample suffixen van B_1 en B_2 gesorteerd zijn, moeten de suffixen waarvan de positie start op een index uit B_0 nog gesorteerd worden. Na elke non-sample suffix S_i staan twee sample suffixen waarvan we de relatieve positie zojuist hebben berekend. Dus door het eerste karakter $S[i]$ van elke non-sample suffix S_i te combineren met de rang van sample suffix S_{i+1} kunnen we de non-sample suffixen sorteren. Formeel gezegd: voor elke i, $j \in B_0$,

$$S_i \leq S_j \iff (S[i],\, rank(S_{i+1})) \leq (S[j],\, rank(S_{j+1}))$$

Alle paren $(S[i],\, rank(S_{i+1}))$ worden via radix sort op volgorde gebracht. In ons voorbeeld geeft dit dus (0,0) < (a,3) < (a,5) < (c,1) < (c,7) waaruit volgt dat $S_{12} < S_3 < S_0 < S_9 < S_6$. De set suffixen $S_i \in S_{B_0}$ is dus gelijk aan 12, 3, 0, 9, 6.
We hebben nu twee lijsten, S_C en S_{B_0}, maar we willen de indices van S_C aflopen volgens rang. Hiervoor gebruiken we de inverse array van S_C, {(14, 13,) 10, 2, 4, 5, 1, 8, 7, 11}. Als laatste stap moeten de twee sets van gesorteerde suffixen samen gevoegd worden tot de uiteindelijke suffix array. Om een suffix $S_i \in S_C$ te vergelijken met $S_j \in S_{B_0}$ hebben we twee gevallen:

$i \in B_1 : S_i \leq S_j \iff (S[i], rank(S_{i+1})) \leq (S[j], rank(S_{j+1}))$
$i \in B_2 : S_i \leq S_j \iff S[i], S[i+1], rank(S_{i+2})) \leq (S[j], S[j+1], rank(S_{j+2}))$

Doordat we bij het uitvoeren van het algoritme extra nullen toevoegen heeft het geen zin om nu nog posities die hoger zijn dan n nog bij te houden. Waardes 12, 13 en 14 vallen dus weg. Het samenvoegen werkt dan door suffixen $S_i \in S_C$ te vergelijken met $S_j \in S_{B_0}$, de kleinste van beide suffixen wordt dan aan de suffix array toegevoegd. Indien $S_i < S_j$ dan wordt vervolgens S_{i+1} met S_j vergeleken, als $S_i > S_j$ wordt S_j toegevoegd aan de

suffix array en worden vervolgens S_{j+1} en S_i met elkaar vergeleken. In het voorbeeld gaat dit dan als volgt:

$$\begin{aligned}
&\text{(a, 3)} < \text{(a, 8)} \iff S_3 < S_{10} \rightarrow 3 \text{ toevoegen,}\\
&\text{(a, 5)} < \text{(a, 8)} \iff S_0 < S_{10} \rightarrow 0 \text{ toevoegen,}\\
&\text{(a, 8)} < \text{(c, 1)} \iff S_{10} < S_9 \rightarrow 10 \text{ toevoegen,}\\
&\text{(c, a, 3)} < \text{(c, a, 8)} \iff S_2 < S_9 \rightarrow 2 \text{ toevoegen,}\\
&\text{(c, 1)} < \text{(c, 4)} \iff S_9 < S_4 \rightarrow 9 \text{ toevoegen,}\\
&\text{(c, 4)} < \text{(c, 7)} \iff S_4 < S_6 \rightarrow 4 \text{ toevoegen,}\\
&\text{(c, c, 7)} < \text{(c, t, 6)} \iff S_5 < S_6 \rightarrow 5 \text{ toevoegen,}\\
&\text{(c, 7)} < \text{(t, 2)} \iff S_6 < S_1 \rightarrow 6 \text{ toevoegen.}
\end{aligned}$$

Omdat alle elementen van S_{B_0} op hun plaats staan in de suffix array, kunnen we de rest van de set S_C achteraan toevoegen. De uiteindelijke suffix array is dus

$$\text{SA} = \{3, 0, 10, 2, 9, 4, 5, 6, 1, 8, 7, 11\}.$$

Plaats- en Tijdscomplexiteit

Nu moeten we nog aantonen dat het algoritme in lineaire tijd kan werken, ondanks de recursie. Het indelen in groepen en de tripletten van twee van deze groepen sorteren gebeurt via radix sort, in lineaire tijd. Het berekenen van de integer waarde van de tripletten kan eveneens lineair, indien bepaalde tripletten meerdere keren voorkomen moet de nieuw gevormde array recursief gesorteerd worden. Elke sorteringsfase is lineair, de vraag is echter of de recursie stopt in lineaire tijd. Om dit te bewijzen hebben we het *master theorem* voor recursie nodig.

Definitie 3.7.1. *Het master theorem voor recursie.*
$T(n) = \sum_{i=1}^{m} T(\alpha_i * n) + O(n^k)$ *met* $0 \leq \alpha_i \leq 1, m \geq 1, k \geq 0$. *Deze vergelijking heeft als oplossing*

$$T(n) = \begin{cases} O(n^k) & \textit{indien } \sum_{i=1}^{m} \alpha_i^k < 1 \\ O(n^k \ log \ n) & \textit{indien } \sum_{i=1}^{m} \alpha_i^k = 1 \\ O(n^c) & \textit{indien } \sum_{i=1}^{m} \alpha_i^k > 1 \end{cases}$$

waarbij c de oplossing is van de vergelijking $\sum_{i+1}^{m} \alpha_i^c = 1$.

Bijgevolg heeft de vergelijking $T(n) = O(n) + T(\lceil 2n/3 \rceil)$ als oplossing $T(n) = O(n)$.
Gegeven de array $rank(S_i)$, kan de inverse in lineaire tijd worden berekend door de array éénmaal te doorlopen. Aangezien elke rang slechts één keer voorkomt en de rang tussen twee gesorteerde suffixen juist 1 is, kunnen we de precieze locatie van elke suffix in de inverse array berekenen.

3.8 Difference cover

Het DC3 algoritme sorteert suffixen waarvan de startpositie in een difference cover sample modulo 3 zitten en gebruikt deze om alle suffixen te sorteren. Dit algoritme kan gegeneraliseerd worden zodat het voor elke difference cover D modulo v geldt.
Het belang van deze generalisatie is dat voor een $v = O(\sqrt{n})$ het DC algoritme kan geïmplementeerd worden om in $O(vn)$ tijd te werken waarbij er $O(n/\sqrt{v})$ extra plaats gebruikt wordt. Dit betekent dus dat men tijd voor space kan inruilen.

In het DC3 algoritme hebben we gesproken over sample suffixen en non-sample suffixen, de keuze van de sample suffixen is een speciaal geval van een difference cover sample. Een difference cover sample moet voldoen aan twee voorwaarden. Ten eerste moeten de sample suffixen efficiënt kunnen worden gesorteerd. In het DC3 algoritme is dit het geval, de tripletten kunnen via radix sort lineair worden gesorteerd. Difference cover samples kunnen snel gesorteerd worden omdat ze periodisch zijn, met een kleine periode. We kunnen bovenstaand algoritme aanpassen zodat we niet met tripletten maar met grotere intervallen werken. Stel dat we een sample met grootte m en een periode v hebben. De sample suffixen kunnen dan gesorteerd worden in $O(vm)$ tijd.
De tweede voorwaarde is dat de non-sample suffixen gesorteerd moeten worden met de hulp van de gesorteerde sample suffixen zodat de volledige suffix array kan worden opgesteld. De set difference cover sample posities hebben de eigenschap dat voor elke i, $j \in [0,\, n-v+1]$ er een kleine ℓ bestaat zodat zowel $i+\ell$ als $j+\ell$ sample posities zijn. Bij het DC3 algoritme is aan deze voorwaarde voldaan, zowel bij het sorteren van de suffixen van S_0 als het samenvoegen van de twee sets.
Een difference cover sample is gebaseerd op difference covers:

Definitie 3.8.1. *Een set $D \subseteq [0, v-1]$ is een difference cover modulo v indien $\{(i-j) \bmod v \mid i, j \in D\} = [0, v-1]$.*

Definitie 3.8.2. *Een v-periodieke sample C van $[0, n]$ met een periode D, $C = \{i \in [0, n] \mid i \bmod v \in D\}$, is een difference cover sample indien D een difference cover modulo v is.*

Een difference cover sample voldoet aan de eerste conditie, namelijk dat ze periodiek is. Volgend lemma toont aan dat ze ook voldoet aan de tweede conditie, dat de non-sample suffixen efficiënt gesorteerd kunnen worden dankzij de sample suffixen.

Lemma 1. *Indien D een difference cover modulo v is, en i en j zijn integers, dan bestaat er een $\ell \in [0, v-1]$ zodat $(i+\ell) \bmod v$ en $(j+\ell) \bmod v$ in D zitten.*

Bewijs. Volgens de definitie van difference cover bestaat er i', $j' \in D$ zodat $i' - j' \equiv i - j \bmod (v)$. Laat $\ell = (i' - i) \bmod v$. Dan $i + \ell \equiv i' \in D \pmod v$ en $j + \ell \equiv i' - (i - j) \equiv j' \in D \pmod v$.

3.8.1 Veralgemeend algoritme

Allereerst moeten we een sample set construeren, voor $k \in [0, v-1]$ definieer $B_k = \{i \in [0,n] \mid i \bmod v = k\}$. De set sample posities is nu $C = \bigcup_{k \in D} B_k$. Laat $\overline{D} = [0,v-1] \ / \ D$ en $\overline{C} = [0, n] \ / \ C$.
Voor $k \in D$ construeer de stringen

$$R_k = [S[k]S[k+1] \dots S[k+v-1]][S[k+v]S[k+v+1] \\ \dots S[k+2v-1]] \dots [S[maxB_k] \dots S[maxB_k + v - 1]].$$

Laat R de concatenatie zijn van alle R_k met $k \in D$. De suffixen van R noemen we de sample suffixen. Deze sample suffixen moeten recursief gesorteerd worden, waarbij de lengte van de periode mag gaan van 3 tot $(1-\epsilon)v^2/|D|$ om zeker te zijn dat de recursie convergeert.
Definieer $rank(S_i)$ zoals bij het DC3 algoritme voor elke $i \in C$ waarbij $rank(S_{n+1}) = rank(S_{n+2}) = \dots = rank(S_{n+v-1}) = 0$ en $rank(S_i) = \perp$ voor $i \in \overline{C}$.
Sorteer elke S_{B_k}, met $k \in \overline{D}$, apart. Laat $\ell \in [0, v-1]$ zodat $(k + \ell) \bmod v \in D$. Om S_{B_k} te sorteren stellen we elke suffix $S_i \in S_{B_k}$ voor door het tuple $(S[i], S[i+1], \dots, S[i+\ell-1], rank(S_{i+\ell}))$. Wat belangrijk is om in te zien is dat de rang altijd gedefinieerd is. Het sorteren van elke S_{B_k} gebeurt in lineaire tijd via radix sort.
In het DC3 algoritme zouden we nu de sets S_{B_k} samenvoegen, maar omdat we hier te maken hebben met v verschillende sets zou het mergen $O(nv \log v)$ tijd kosten omdat er n keer $O(v \log v)$ vergelijkingen nodig zijn om het kleinste element uit v verschillende lijsten te selecteren en aan de suffix array toe te voegen. Daarom moeten we uitgaan van een iets andere aanpak.
We delen de sample suffix set S_C in sets S_{B_k}, met $k \in D$, waarbij elke set gesorteerd blijft. Nu hebben we alle sets S_{B_k}, met $k \in [0, v-1]$, als gesorteerde sequenties. Nu moeten deze sequenties geconcateneerd worden en vervolgens stabiel gesorteerd worden op de eerste v karakters.
Voor $\alpha \in [0,n]^v$, neem S^α de set van suffixen die beginnen met α, en laat $S^\alpha_{B_k} = S^\alpha \cap S_{B_k}$. Het algoritme heeft nu de suffixen in sets $S^\alpha_{B_k}$ ingedeeld. De sets zijn gegroepeerd volgens α en elke set $S^\alpha_{B_k}$ zelf is gesorteerd.
Een voorbeeld met $v = 3$ en alfabet {a,c,g,t} ziet er dan als volgt uit:

$S^{aaa}_{B_0}$	$S^{aaa}_{B_1}$	$S^{aaa}_{B_2}$		$S^{aac}_{B_0}$	$S^{aac}_{B_1}$	$S^{aac}_{B_2}$		$S^{aag}_{B_0}$	$\dots$

Tot slot moeten voor elke $\alpha \in [0,n]^v$ de sets $S^\alpha_{B_k} \in [0, v-1]$ in de set S_α gesorteerd worden. Op het eerste zicht kost dit $O(nv \log v)$ tijd, maar omdat elk element zijn unieke rang heeft moeten er in elke groep van v elementen

v	3	7	13	21	31	32	64	128	256	512	1024	2048
$\|D\|$	2	3	4	5	6	7	9	13	20	28	40	58

Tabel 3.4: Difference cover D modulo v waardes

slechts v vergelijkingen gebeuren om het kleinste element te sorteren. De benodigde tijd bedraagt dus $O(nv)$.
Vervolgens mergen we alle sets. Voor elke $i, j \in [0, n]$, laat $\ell \in [0, v-1]$ zodat $(i + \ell)$ mod v en $(j + \ell)$ mod v beide in D zitten. Suffixen S_i en S_j worden vergeleken dankzij $rank(S_{i+\ell})$ en $rank(S_{j+\ell})$. Dit geeft de juiste volgorde omdat S_i en S_j tot dezelfde set S^α en dus $S[i]S[i+1]\dots S[i+\ell-1] = S[j]S[j+1]\dots S[j+\ell-1]$.

Plaats- en Tijdscomplexiteit

De tijdscomplexiteit van het gegeneraliseerde algoritme bedraagt $O(vn)$. Ook kan het zo worden geïmplementeerd dat er slechts $O(n/\sqrt{v})$ extra space nodig is om de output te berekenen.
De enige vraag die overblijft is welke difference covers D modulo v optimaal zijn. Het is duidelijk dat D niet kleiner dan $\sqrt{v}$ mag zijn, anders wordt aan definitie 3.8.1 niet voldaan. Aan de andere kant mag volgens dezelfde definitie 3.8.1 $D = [0, v-1]$, waardoor alle suffixen gesorteerd moeten worden. We willen dat de groep van non-sample suffixen zo groot mogelijk is, dus dat we de non-sample suffixen kunnen sorteren dankzij de sample suffixen. Indien de set sample suffixen te klein is kunnen de non-sample suffixen niet gesorteerd worden, dit kan echter niet voorkomen als er aan definitie 3.8.1 voldaan is.
Het algoritme wordt niet optimaal uitgevoerd indien D een te grote subset van $[0, v-1]$ is. De lower bound voor D is $\sqrt{v}$, de beste upper bound die we kennen komt van Dolbourn and Ling [CL00], zij hebben volgend lemma bewezen:

Lemma 2. *Voor elke v kan een difference cover modulo v met grootte ten hoogste $\sqrt{1.5v} + 6$ berekend worden in $O(\sqrt{v})$ tijd.*

Dus we weten dat voor een difference cover D modulo v de optimale grootte van D tussen $\sqrt{v}$ en $\sqrt{1.5v} + 6$ ligt. Onderstaande tabel geeft de kleinste bekende difference cover D modulo v voor verschillende periode lengtes v. Voor $v \leq 128$ weten we dat ze optimaal zijn.

3.9 Pipelined suffix array implementatie

Dementiev, Kärkkäinen, Mehnert, Sanders hebben enkele van de hierboven suffix array constructiealgoritmes geïmplementeerd waarbij ze gebruik

maken van pipelining [DMKS05]. Pipelining is een techniek waarbij tussenresultaten niet gematerialiseerd worden, maar van het ene algoritme worden doorgegeven naar het andere. Deze techniek wordt dikwijls toegepast in database-toepassingen, om de hoeveelheid I/O te minimaliseren.
De bedoeling van hun implementatie is om na te gaan of het DC3 algoritme, dat lineair is, ook in de praktijk sneller is dan doubling en discarding, twee niet-lineaire algoritmes. Hun implementatie is er op gericht om grote sequenties te construeren waarvan de suffix array die, samen met extra datastructuren die nodig zijn voor het algoritme, niet meer in het werkgeheugen passen. We hebben deze algortimes, DC3, doubling en discarding getest om te controleren of een aantal conclusies van de auteurs kunnen bevestigen.

Het doubling algoritme is nog voor verbetering vatbaar, in de basisversie van het algoritme worden namen gezocht voor twee aaneengrenzende tuples, maar men kan per iteratie voor verschillende tuples samen één naam zoeken. Voor elke iteratie wordt het sorteren dan zwaarder, maar er moeten minder iteraties worden uitgevoerd, wat een positief effect heeft op het aantal I/O-operaties. Onderzoek heeft uitgewezen dat per iteratie 5 tuples samennemen het meest efficiënt is. In de implementatie van [DMKS05] worden slechts 4 tuples samengevoegd per iteratie, omdat 4 een veelvoud is van 2 en theoretisch nauwelijks trager dan wanneer er 5 tuples worden samengenomen. Deze versie van het doubling algoritme wordt ook wel het *quadrupling* algoritme genoemd. Het aantal I/O operaties zou voor quadrupling ongeveer 30% lager liggen dan voor doubling. Om deze reden hebben we het quadrupling algoritme getest in plaats van doubling.

De piplined versie van het discarding algoritme verschilt weinig van de implementatie van het doubling algoritme. Details over beide algoritmes kunnen teruggevonden worden in [DMKS05]. We gebruiken eveneens het quaddiscarding algoritme, maar we noemen het gewoon discarding.
Ook het piplined DC3 algoritme lijkt sterk op het algoritme voor doubling, het enige probleem is dat de input sequentie twee keer moet worden gescand. Eerst worden de sample suffixen geselecteerd, waarna er door de input moet worden gescand om de namen van de tuples te genereren. Deze twee scans zitten elkaar in de weg, men heeft dit opgelost door een tijdelijke kopie te maken van de input stream. Verdere details over de pipelined algoritmes zijn te vinden in [DMKS05].

3.9.1 Testresultaten

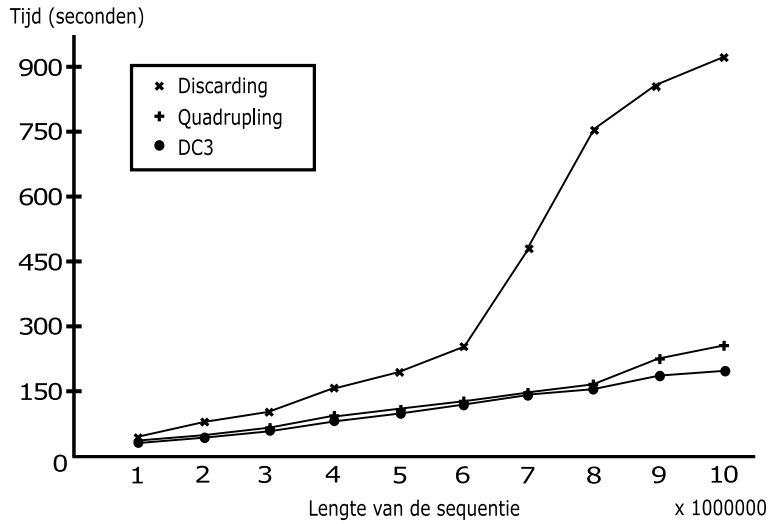
We hebben de pipelined implementatie van het quadrupling, discarding en DC3 algoritme getest op stukken van het menselijk chromosoom nummer 1. Eerst hebben we de verschillende algoritmes getest op een deelsequentie van

1 tot 10MB, om een idee te krijgen van de uitvoersnelheid op korte sequenties. Deze resultaten zijn niet spectaculair, de belangrijkste reden hiervoor is dat de harde schijf gebruikt wordt voor bepaalde sorteer- en eventueel merge-procedures in de algoritmes. Hun implementatie is er op gericht om van zeer grote sequenties de suffix array te construeren, waarbij er te weinig werkgeheugen voorhanden is. Voor korte sequenties kan het algoritme de suffix array beter volledig in het werkgeheugen opbouwen, de grote vertragende factor bij dit soort algoritmes is de I/O naar de harde schijf.

In figuur 3.4 zien duidelijk dat het discarding algoritme trager werkt dan DC3 en quadrupling. Dit strookt niet met de theoretische tijdscomplexiteit, we hadden verwacht dat discarding sneller zou werken dan quadrupling omdat discarding een verbeterde versie is van het quadrupling algoritme. Ook komt dit niet overeen met de resultaten van de auteurs in [DMKS05].
We kunnen verschillende redenen geven voor deze afwijkende resultaten. Er is een verschil in gebruikte hardware, de auteurs hebben een systeem met een dual-core 2.0 GHz Intel Xeon processor, een gigabyte RAM en vier harde schijven van 80 gigabyte gebruikt voor hun experimenten. Wij hebben onze testen uitgevoerd op een eenvoudige laptop met een 1.7 Intel Centrino processor, 512 megabyte RAM en één harde schijf van 60 gigabyte (waarvan 10 gigabyte gereserveerd voor het algoritme). Het meest in het oog springende verschil is het feit dat hun systeem twee processoren heeft, bepaalde algoritmes kunnen immers geparallelliseerd worden zodat ze optimaal gebruik kunnen maken van de verschillende processoren. Andere algoritmes kunnen dan weer nauwelijks profijt halen uit een extra processor. Tussen discarding en quadrupling is er echter weinig verschil, beide algoritmes kunnen vlot geparalleliseerd worden. Het lijkt ons weinig waarschijnlijk dat de verschillende testresultaten hierdoor verklaard kunnen worden.
De external memory library stxxl versie 0.52, die door de pipelined algoritmes gebruikt wordt om data te bufferen, is er op voorzien om met meerdere harde schijven te werken [DS05]. Naast de andere specificaties van het systeem kan dit een invloed hebben op de testresultaten. Het is uiteraard ook mogelijk dat de iets hogere complexiteit van het discarding-algoritme verantwoordelijk is voor een tragere werking van het algoritme.

Theoretisch gezien is vooral de grootte van de langste gemeenschappelijke prefix van belang bij het doubling algoritme, het discarding algoritme zou hier minder last van hebben. Het doubling algoritme is bijzonder gevoelig voor een grote lgp, hoe groter de maximale lgp, hoe meer iteraties er worden uitgevoerd. Genomen van onder andere zoogdieren staan er om bekend dat ongeveer de helft van het DNA bestaat uit herhalingen en reversals. Het discarding algoritme is minder gevoelig aan enkele bijzonder grote herhalingen, maar het algoritme kan minder tuples negeren indien de gemiddelde lgp vrij groot is. In de gebruikte sequenties schommelt de gemiddelde lgp

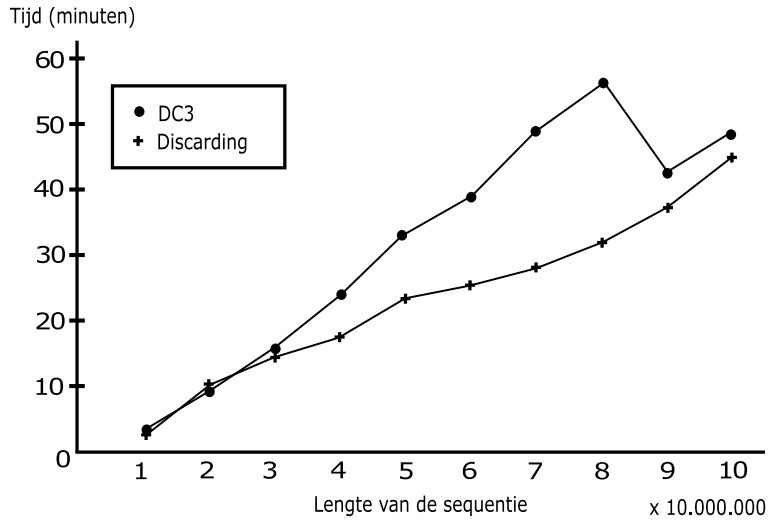
rond de 10 karakters, wat betekent dat er inderdaad een vrij groot aantal herhalingen aanwezig is.



Figuur 3.4: Vergelijking tussen quadrupling, discarding en DC3 met een sequentiegrootte van 1 tot 10 miljoen karakters

Het verschil in snelheid tussen DC3 en discarding is vrij klein en de uitvoeringstijd is redelijk, dus we zijn enkel verder gegaan met deze twee algoritmes bij het testen van sequenties tussen 10 en 100 miljoen base-pairs.
Uit resultaten voorgesteld in figuur 3.5 kunnen we vaststellen dat DC3 en quadrupling niet fel van elkaar verschillen, daarom hebben we beide algoritmes de suffix array van het volledige menselijke chromosoom 1, wat iets minder dan 250 miljoen base-pairs bevat, laten constueren. Quadrupling blijkt de suffix array in 2 uur te kunnen maken, terwijl DC3 40 minuten extra nodig had.

De conclusie van de auteurs van [DMKS05] wijkt af van onze resultaten. Volgens hun testresultaten op stukken van het menselijk genoom met grootte tussen 2^{24} tot 2^{30} karakters blijkt DC3 het snelste, gevolgd door discarding wat ongeveer dubbel zoveel tijd nodig heeft. Traagste van de drie algoritmes was quadrupling, wat tot vier keer trager was dan DC3. DC3 kwam in alle testen op verschillende types tekst als snelste algoritme uit de bus. De geteste teksten waren twee geconcateneerde kopieën random gegenereerde tekst (om een enorme maximale lgp te krijgen), de Gutenberg bijbel, het menselijk genoom en een set html-paginas. Het waarom van de verschillende testresultaten is moeilijk te achterhalen, extra testen op verschillende hardware zou wenselijk zijn.



Figuur 3.5: Vergelijking tussen discarding en DC3 met een sequentiegrootte van 10 tot 100 miljoen karakters

Hoofdstuk 4

De enhanced suffix array

De suffix tree is de belangrijkste data structuur voor string processing algoritmes, maar een suffix tree is veeleisend op het gebied van space. De suffix array daarentegen is heel wat zuiniger dan een suffix tree, maar heeft minder toepassingen op het gebied van string processing en comparative genomics. Verder werkend op de resultaten uit [KLA+01] wordt in [AKO04] aangetoond dat elk algoritme dat een suffix tree als data structuur gebruikt kan omgevormd worden tot een algoritme dat enkel een suffix array nodig heeft, samen met wat extra tabellen. Meer zelfs, de omvorming gebeurt op zulk een manier dat de tijdscomplexiteit gelijk blijft.
Abouelhoda, Kurtz en Ohlebusch noemen hun datastructuur een enhanced suffix array, ze bestaat uit een suffix array en enkele extra arrays. Het grote voordeel van deze structuur is dat algoritmes minder plaats nodig hebben en dat ze volgens de makers ook sneller zijn. Algoritmes kunnen een suffix tree op verschillende manieren gebruiken, door de suffix tree top-down of bottom-up te doorlopen, of door gebruik te maken van suffix links. De enhanced suffix array wordt aangevuld met extra arrays afhankelijk van de wijze waarop de suffix tree doorlopen moet worden.

4.1 Bottom-up simulatie

We geven eerst een methode om met een suffix array en wat informatie over de langste gemeenschappelijke prefixen het bottom-up doorlopen van een suffix tree te simuleren. Oorspronkelijk komt dit idee van Kasai et Al. [KLA+01], maar in [AKO04] heeft men het algoritme verrijkt met het concept *lcp-interval trees*, zij bevatten zowel informatie over de langste gemeenschappelijke prefixen als informatie over kind-nodes in een suffix tree. Eerst worden concepten zoals lcp-interval gedefinieerd waarna we uitleggen hoe een lcp-interval tree gebouwd kan worden.

Definitie 4.1.1. *Een interval* $[i..j]$ *met* $0 \leq i < j \leq n$ *is een lcp-interval (longest common prefix ofwel langste gemeenschappelijke prefix) met lcp-waarde* ℓ *indien:*

1. $lcptab[i] < \ell$,
2. $lcptab[k] \geq \ell$ *voor* k *met* $i+1 \leq k \leq j$,
3. $lcptab[k] = \ell$ *voor ten minste één* k *met* $i+1 \leq k \leq j$,
4. $lcptab[j+1] < \ell$.

Een lcp-interval $[i..j]$ met lcp-waarde ℓ wordt ook wel geschreven als ℓ-interval.

Definitie 4.1.2. *Een* ℓ*-interval* $[i..j]$ *is een lokaal maximum in de lcp-tabel indien* $SA[k] = \ell$ *voor alle* $i+1 \leq k \leq j$.

De lcp-table *lcptab* is een array van integers lopende van 0 tot n, lcptab[0] = 0 en lcptab[i] is gelijk aan de lengte van de langste gemeenschappelijke prefix van de suffixen $S_{SA[i-1]}$ en $S_{SA[i]}$ voor $1 \leq i \leq n$. Omdat $S_{SA[n]} = \$$ geldt dat lcptab[n] = 0. Deze tabel kan berekend worden als een bijproduct tijdens de constructie van de suffix array, ofwel kan ze uit de suffix array worden geconstrueerd in lineaire tijd [KLA$^+$01].
Een probleem met het algoritme van Kasai is dat het vrij veel space gebruikt. Normaal wordt per tekst symbool één byte gebruikt en per element van een suffix of lcp array vier bytes. Het algoritme van Kasai et Al. gebruikt echter $13n$ bytes, wat dus $4n$ bytes overhead betekent. Manzini [Man04] heeft dit algoritme verbeterd zodat er bij het berekenen van de lcp-array nauwelijks overhead bestaat. Zijn oplossing zou volgens zijn eigen berekeningen wel 5 tot 10% trager werken. In dezelfde paper wordt ook een algoritme voorgesteld dat bij het construeren van de lcp-array de suffix array overschrijft. Voor sommige toepassingen is immers enkel de lcp-array van tel en niet de suffix array. Het voordeel is dat er minder space verloren gaat, slechts $(5 + \lambda)n$ bytes waarbij $\lambda << 1$.

De tabel *bwttab* bevat de Burrows en Wheeler transformatie [BW94], gebruikt voor data compressie. Het is een tabel met grootte $n+1$ zodat voor elke i met $0 \leq i \leq n$, bwttab[i] = S[SA[i]-1] als SA[i] $\neq 0$. Indien SA[i] = 0 dan is bwttab[i] niet gedefinieerd. De tabel kan uiteraard in $O(n)$ tijd worden geconstrueerd uit de suffix array [KLA$^+$01].

Als we bovenstaande definities toepassen op de sequentie `atcacccttca$` krijgen we tabel 4.1 als resultaat.

Nu geven we het algoritme om door middel van een suffix array en zijn lcp-table *lcptab* het bottom-up doorlopen van een suffix tree te simuleren.

i	SA	lcptab	bwttab	$S_{SA[i]}$
0	3	0	c	acccttca$
1	0	1		atcacccttca$
2	10	1	c	a$
3	2	0	t	caccctttca$
4	9	2	t	ca$
5	4	1	a	cccttca$
6	5	2	c	ccttca$
7	6	1	c	cttca$
8	1	0	a	tcacccttca$
9	8	3	t	tca$
10	7	1	c	ttca$
11	11	0	a	$

Tabel 4.1: De enhanced suffix array van de sequentie atcacccttca$

Het algoritme is oorspronkelijk van [KLA+01], maar deze aangepaste versie is afkomstig uit [AKO04]. Alle lcp-intervallen van lcptab worden berekend met behulp van een stack. De elementen op de stack zijn lcp-intervallen voorgesteld door tuples $\langle lcp, lb, rb\rangle$, waarbij *lcp* de lcp-waarde is van dat interval, *lb* de linkergrens en *rb* de rechtergrens is. Merk op dat de functie *report(interval)* door het algoritme dat de virtuele suffix tree bottom-up doorloopt moet worden ingevuld.

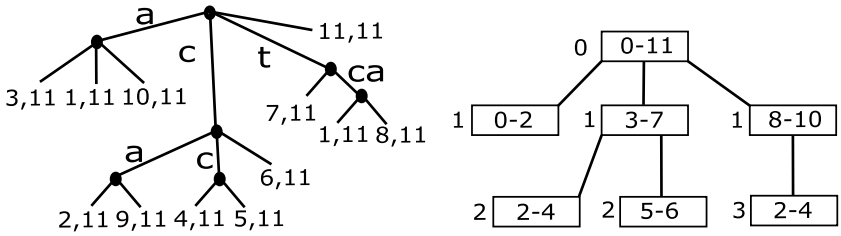
Algoritme 6. *Simulatie bottom-up doorlopen van een suffix tree.*

```
push(⟨0, 0, ⊥⟩)
for i := 1 to n do
   lb := i − 1
   while lcptab[i] < top.lcp
     top.rb := i − 1
     interval := pop
     report(interval)
     lb := interval.lb
   if lcptab[i] > top.lcp then
     push(⟨lcptab[i], lb, ⊥⟩)
```

Er bestaat een aangepast algoritme dat eveneens de kind-intervallen bij elk interval teruggeeft, alvorens we het algoritme presenteren moeten we eerst uitleggen wat een lcp-interval tree is.

Een m-interval $[l..r]$ is *embedded* in een ℓ-interval $[i..j]$ indien het een subinterval is van $[i..j]$, met $i \leq l < r \leq j$ en $m > \ell$. Indien $[l..r]$ bevat zit in

$[i..j]$ en er is geen enkel ander interval in $[i..j]$ dat ook $[l..r]$ omvat, dan is $[l..r]$ een kind interval van $[i..j]$.
Dankzij deze ouder-kind relatie kan men een virtuele boom opstellen die we de lcp-interval tree van de suffix array noemen. Deze virtuele boom is eigenlijk de suffix tree zonder bladeren, concreet betekent dit dus dat een tuple $\langle lcp, lb, rb\rangle$ een node is waarbij alle suffixen tussen de indices lb en rb de bladeren vormen onder die node. Figuur 4.1 geeft de lcp-interval tree weer van ons voorbeeld, vergeleken met de suffix tree. Op deze figuur zien we duidelijk dat de lcp-interval tree de suffix tree is zonder bladeren.



Figuur 4.1: De suffix tree en lcp-interval tree van de sequentie atcacccttca$

Algoritme 6 kunnen we nog wat aanpassen zodat er in een tuple ook nog informatie wordt bijgehouden over de child-nodes. Men kan in principe van elke node apart de lijst van kind-intervallen achterhalen, maar dat kost meer tijd. De enhanced suffix array houdt geen expliciete pointers bij om de ouder-kind relatie aan te duiden, het algoritme 7 werkt zoals algoritme 6 door de lcp-tabel linear te scannen en intervallen op een stack te plaatsen. De elementen op de stack zijn lcp-intervallen voorgesteld door tuples $\langle lcp, lb, rb, childList\rangle$, waarbij *childList* de lijst van kind-intervallen is. Wat het algoritme eigenlijk doet is een virtuele lcp-interval tree aflopen en alle nodes ervan in top-down volgorde returnen.

Algoritme 7. *Simulatie bottom-up doorlopen van een lcp-interval tree met per node een lijst van kind-intervallen.*

```
lastInterval := ⊥
push(⟨0, 0, ⊥, [ ]⟩)
for i := 1 to n do
  lb := i − 1
  while lcptab[i] < top.lcp
    top.rb := i − 1
    lastInterval := pop
    report(lastInterval)
    lb := lastInterval.lb
    if lcptab[i] ≤ top.lcp then
      top.childList := add(top.childList,lastInterval)
```

```
      lastInterval := ⊥
  if lcptab[i] > top.lcp then
    if lastInterval ≠ ⊥ then
      push(⟨lcptab[i], lb, ⊥, [lastInterval]⟩)
      lastInterval := ⊥
    else push(⟨lcptab[i], lb, ⊥, [ ]⟩)
```

4.2 Top-down simulatie

Naast simuleren van het bottom-up doorlopen van een suffix tree is het interessant om voor elk ℓ-interval $[i..j]$ in constante tijd te kunnen bepalen wekle de kind intervallen zijn. Om dit te verwezelijken wordt een extra tabel toegevoegd aan de suffix array: de child-tabel *childtab*. Deze tabel bevat $n+1$ elementen die elk drie waarden hebben: *up*, *down* en *nextℓIndex*. Elk van deze drie waarden kosten 4 bytes in het ergste geval, maar in [AKO04] is aangetoond dat deze drie waarden kunnen worden samengenomen tot één veld.
In essentie slaat een child-tabel de ouder-kind relatie op van lcp-intervallen. Dus voor een ℓ-interval $[i..j]$ met ℓ-indices $i_1 < i_2 < ... < i_k$ wordt de childtab$[i]$.*down* of childtab$[j+1]$.*up* waarde gebruikt om de eerste ℓ-index i_1 te bepalen. De andere ℓ-indices $i_1, ..., i_k$ kunnen gevonden worden via childtab$[i_1]$.*nextℓIndex*, ..., childtab$[i_{k-1}]$.*nextℓIndex*. Van zodra alle ℓ-indices bekend zijn kent men de kind intervallen van $[i..j]$. Als $i_1 < i_2 < ... < i_k$ de ℓ-indices in aflopende volgorde zijn, dan zijn de kind-intervallen van $[i..j]$ gelijk aan $[i..i_1-1]$,$[i_1..i_2-1]$,...,$[i_k..j]$.
Men kan de waarden van *up*, *down* en *nextℓIndex* van elke childtab-entry ook formeel omschrijven:

- childtab$[i]$.*up* $= \min\{q \in [0..i-1] \mid \text{lcptab}[q] > \text{lcptab}[i]$ en $\forall\, k \in [q+1..i-1] : \text{lcptab}[k] \geq \text{lcptab}[q]\}$
- childtab$[i]$.*down* $= \max\{q \in [i+1..n] \mid \text{lcptab}[q] > \text{lcptab}[i]$ en $\forall\, k \in [i+1..q-1] : \text{lcptab}[k] > \text{lcptab}[q]\}$
- childtab$[i]$.*nextℓIndex* $= \min\{q \in [i+1..n] \mid \text{lcptab}[q] = \text{lcptab}[i]$ en $\forall\, k \in [i+1..q-1] : \text{lcptab}[k] > \text{lcptab}[i]\}$

Als voorbeeld nemen we de *up*, *down* en *nextℓIndex* waarden van het interval [3-7], deze staan op positie 3 in tabel 4.2. De waarde 1 staat in childtab[3].*up* en childtab[0].*down*, zodat het duidelijk is dat interval [0-2] een linkerbroer is van interval [3-7]. Op positie 5 staat als nextℓIndex waarde 8, zodat we weten dat de rechterbroer van interval [3-7] begint met [8-?]. Waarde 5 op index 3 staat voor het tweede kind-interval van [3-7], het interval dat begint

op positie [5-?]. Hierdoor weten we dat het eerste kind van interval [3-7] het interval [3-4]is. Op index 5 staat als nextℓIndex de waarde 7 zodat we weten dat het vorige interval het interval [5-6] was en dat het volgende interval begint op positie [7-?]. Op index 7 staat echter geen nextℓIndex meer zodat we kunnen afleiden dat het geen interval is, maar gewoon de 7de suffix in SA. Uit dit alles kunnen we dus opmaken dat interval [3-7] twee kinderen heeft, namelijk de intervallen [3-4] en [5-6].

De constructie van de child-table kan in lineaire tijd uitgevoerd worden door het bottom-up doorlopen van de lcp-interval tree zoals in algoritme 7. Hier geven we voor de duidelijkheid twee verschillende algoritmes om de *up/down* waardes en de *nextℓIndex* waarde van de child-tabel te berekenen.

Algoritme 8. *Berekenen van de up en down waardes.*

```
lastIndex := -1
push(0)
for i := 1 to n do
   while lcptab[i] < lcptab[top]
     lastIndex := pop
     if(lcptab[i] ≤ lcptab[top]) and
          (lcptab[top] ≠ lcptab[lastIndex]) then
       childtab[top].down := lastIndex
   /* nu geldt lcptab[i] ≥ lcptab[top] */
   if lastIndex ≠ -1 then
     childtab[i].up := lastIndex
     lastIndex := -1
   push(i)
```

Het algoritme doorloopt de lcp-tabel en pusht de huidige index op de stack indien zijn lcp-waarde groter of gelijk is aan de lcp-waarde waar de top van de stack naar wijst. Elementen worden van de stack gepopt zolang hun lcp-waarde groter is dan deze op de huidige index. Dus dankzij de lcp-waardes van de top van de stack te vergelijken met de huidige lcp-waardes, worden de *up* en *down* velden van de child-tabel gevuld met elementen die van de stack gepopt worden tijdens het doorlopen van de lcp-tabel.

Het algoritme om de *nextℓIndex* waardes in te vullen is iets gemakkelijker, men moet gewoon controleren of lcptab[i] gelijk is aan lcptab[top]. Indien dit zo is wordt i de waarde van het veld childtab[top].*nextℓIndex*.

Algoritme 9. *Berekenen nextℓIndex-waardes.*

```
push(0)
```

```
for i := 1 to n do
  while lcptab[i] < lcptab[top]
    pop
  if lcptab[i] = lcptab[top] then
    lastIndex := pop
    childtab[lastIndex].nextℓIndex := i
  push(i)
```

Na het berekenen van de waardes van *up*, *down* en *nextℓIndex* op onze voorbeeldsequentie krijgen we tabel 4.2.

			childtab			
i	SA	lcptab	1	2	3	$S_{SA[i]}$
0	3	0		1	3	acccttca$
1	0	1			2	atcacccttca$
2	10	1				a$
3	2	0	1	5	8	cacccttca$
4	9	2				ca$
5	4	1	4	6	7	cccttca$
6	5	2				ccttca$
7	6	1	6			cttca$
8	1	0	5	10	11	tcacccttca$
9	8	3				tca$
10	7	1	9			ttca$
11	11	0	10			$

Tabel 4.2: Enhanced suffix array met childtab; 1 = up, 2 = down en 3 = nextℓIndex

Als we naar de tabel 4.2 kijken zien we dat veel velden leeg zijn en dat de *up* en *down*-waardes dikwijls herhaald worden. De drie velden *up*, *down* en *nextℓIndex* kunnen worden gecomprimeerd tot één enkel veld.

4.2.1 Opzoeken van een kind-interval

Om de kind-intervallen van een ℓ-interval $[i..j]$ te vinden moeten we eerst de voorste ℓ-index in $[i..j]$ localiseren, het minimum van de set $\ell Indices(i,j)$. De eerste ℓ-index in $[i..j]$ kunnen we vinden via de *up* en *down* waarde in de child-tabel.
Voor elk ℓ-interval $[i..j]$ geldt het volgende:

1. $i <$ childtab$[j+1].up \leq j$ of $i <$ childtab$[i].down \leq j$

2. childtab$[j+1].up$ houdt de eerste ℓ-index bij van $[i..j]$ indien $i <$ childtab$[j+1].up \leq j$

3. childtab$[i].down$ houdt de eerste ℓ-index bij van $[i..j]$ indien $i <$ childtab$[i].down \leq j$

Van zodra de eerste ℓ-index i_1 van een ℓ-interval $[i..j]$ gevonden is, kunnen de volgende ℓ-indices $i_2 < i_3 < \ldots < i_k$ uit $[i..j]$, waarbij $1 \leq k \leq |\Sigma|$, opgevraagd worden uit het *nextℓIndex*-veld van achtereenvolgens childtab$[i_1]$, childtab$[i_2]$, ..., childtab$[i_{k-1}]$. Hieruit volgt dat de kind-intervallen van $[i..j]$ de intervallen $[i..i_1-1]$, $[i_1..i_2-1]$, ..., $[i_k..j]$ zijn. Het algoritme om van een ℓ-interval de kind-intervallen op te vragen ziet er als volgt uit:

Algoritme 10. *getChildIntervals(i,j) waarbij lcp-interval* $[i..j] \neq [0..n]$.

```
intervalList = [ ]
if i < childtab[j + 1].up ≤ j then
   i1 := childtab[j + 1].up
else
   i1 := childtab[i].down
intervalList.add(i,i1 − 1)
while childtab[i1].next`Index ≠ ⊥ do
   i2 := childtab[i1].next`Index
   intervalList.add(i1, i2 − 1)
   i1 := i2
intervalList.add(i1, j)
return intervalList
```

De functie *getChildIntervals* heeft een tijdscomplexiteit van $O(|\Sigma|)$, wat we beschouwen als constante tijd. Door deze functie te gebruiken kunnen we een top-down traversal over een suffix tree simuleren. Men kan gemakkelijk een functie *getInterval* schrijven die als input-parameters een ℓ-interval $[i..j]$ en een karakter $a \in \Sigma$ krijgt en als resultaat het kind interval $[l..r]$ van $[i..j]$, waarvan de suffix het karakter a heeft op positie ℓ, returnt. Indien zulk een interval niet bestaat moet *getInterval* de waarde $\perp$ returnen.
Men kan ook een functie *getLcp*(i, j) implementeren die de lcp-waarde van een lcp-interval $[i..j]$ in constante tijd berekent. Indien $i <$ childtab$[j+1].up \leq j$ dan geeft *getlcp*(i, j) de waarde lcptab[childtab$[j+1].up$] terug, of anders de waarde lcptab[childtab$[i].down$].

We passen algoritme 10 toe om de kind-intervallen van het lcp-interval [3..7] te vinden. We weten dat $i = 3$ en $j = 7$, met $3 <$ childtab$[8].up = 7 \leq 7$ waardoor $i_1 :=$ childtab$[8].up = 7$. Aan de intervallijst voegen we 1-interval [3..4] toe. Aangezien childtab$[7].next\ell Index = \perp$ voegen we het 1-interval

[5..6] toe aan de intervallijst. De kind-intervallen van het lcp-interval [3..7] zijn dus [3..4] en [5..6].

4.2.2 Enkele toepassingen

Voor een gewone suffix array heeft het beantwoorden van een vraag zoals 'Is P een substring van S?' een worst-case tijdscomplexiteit van $O(m \log n)$, waarbij m en n de lengtes zijn van respectievelijk P en S. Dankzij de enhanced suffix array kunnen we dit type queries oplossen in $O(m)$ tijd. Queries van het type 'Geef alle z voorkomens van P in S' kunnen in $O(m + z)$ tijd beantwoord worden, dus onafhankelijk van de lengte van S.

Algoritme 11. *Beantwoorden van een beslissingsquery.*

```
c := 0
queryFound := True
(i, j) := getInterval(0, n, P[c])
while (i, j) ≠ ⊥ en c < m en queryFound = True
   if i ≠ j then
      ℓ := getlcp(i, j)
      min := min{ℓ, m}
      queryFound := S[SA[i] + c..SA[i] + min - 1] = P[c..min - 1]
      c := min
      (i, j) := getInterval(i, j, P[c])
   else
      queryFound := S[SA[i] + c..SA[i] + m - 1] = P[c..m - 1]
if queryFound then
   return(i, j) /* gevraagd P-interval */
else
   print 'Pattern P niet gevonden in S'
```

Eerst vindt de functie *getInterval* het juiste lcp-interval $[i..j]$ waarvan de suffixen met het karakter P[0] beginnen. Vervolgens gaat via de while-lus uit het vorige bepaalde lcp-interval telkens een deelinterval gezocht worden totdat het lcp-interval ofwel een singleton interval is geworden, waarbij de voorste karakters gelijk moeten zijn aan de nog niet gematchte karakters van P, ofwel totdat alle karakters van P gematcht zijn via de prefix van de intervallen, zodat alle elementen uit het laatste deelinterval P als prefix hebben. Dit algoritme werkt lineair omdat we via c bijhouden hoeveel karakters er al gematcht zijn via de lcp-waarde van de voorouder-intervallen. Hierdoor wordt elk karakter van P slechts één keer vergeleken met een karakter uit S, zodat de worst-case tijdscomplexiteit $O(m)$ bedraagt.
Queries waarbij alle elementen moeten worden opgesomd kunnen door een kleine uitbreiding van de bovenstaande functie worden beantwoord. Gegeven een sequentie P met lengte m wordt het P-interval $[l..r]$ gezocht via

bovenstaand algoritme, wat $O(m)$ tijd kost. Daarna kunnen de startposities van elk voorkomen van P in S worden opgesomd: SA[l], ..., SA[r]. Dus indien P z keer voorkomt in S, neemt het opzoeken van het P-interval samen met het aflopen van de startposities van elk voorkomen van P in S ten hoogste $O(m + z)$ tijd in beslag.

4.3 Simulatie van suffix links

Het laatste element waarin een enhanced suffix array verschilt van een suffix tree is de suffix link. Bij de suffix tree wijst een suffix link van één node naar een andere, in deze voorstelling is de suffix link een verwijzing van een ℓ-interval $[i..j]$ naar een interval $[l..r]$ met een lcp van $\ell - 1$. We geven eerst enkele definities alvorens over te gaan tot de bespreking van de constructie van de suffix link tabel.

Definitie 4.3.1. *Stel dat $S_{SA[i]} = a\omega$. Indien voor index j, met $0 \leq j < n$, geldt dat $S_{SA[j]} = \omega$, dan noteren we j als link$[i]$ en noemen het de suffix link van i. Indien $SA[i] < n$, dan link$[i] = SA^{-1}[SA[i] + 1]$.*

Definitie 4.3.2. *Gegeven het ℓ-interval $[i..j]$ wordt het kleinste lcp-interval $[l..r]$, waarvoor geldt $l \leq$ link$[i] <$ link$[j] \leq r$, het suffix link interval van $[i..j]$ genoemd.*

Lemma 3. *Gegeven een $a\omega$-interval ℓ-$[i..j]$ is zijn suffix link interval een ω-interval dat een lcp-waarde $\ell - 1$ heeft.*

Bovenstaand lemma kunnen we als volgt bewijzen. De langste gemeenschappelijke prefix van $a\omega$ is $S_{SA[i]}$, ..., $S_{SA[j]}$. Bijgevolg is ω de langste gemeenschappelijke prefix van $S_{SA[link[i]]}$, ..., $S_{SA[link[j]]}$. Omdat $[l..r]$ het kleinste lcp-interval is waarvoor geldt dat $l \leq$ link$[i] <$ link$[j] \leq r$, volgt hieruit dat ω de langste gemeenschappelijke prefix is van $S_{SA[l]}$, ..., $S_{SA[r]}$. Dus is $[l..r]$ het ω-interval en heeft het een lcp-waarde van ℓ - 1.
Voor elk ℓ-interval $[i..j]$ moet een suffix link worden bijgehouden naar het interval $[l..r]$, dit gebeurt door de linker en rechter grens l en r te bewaren bij de eerste ℓ-index van $[i..j]$. De overeenkomstige tabel noemen we *suflink*, zie tabel 4.3. De lcp-waarde van $[l..r]$ moet niet worden bijgehouden omdat deze ℓ - 1 is. De tabel wordt geconstrueerd door de lcp-interval breadth-first van links naar rechts te doorkruisen. Voor elke lcp-waarde die we tegenkomen wordt een lijst intervallen bijgehouden die deze waarde hebben. Toegepast op ons voorbeeld geeft dit dan een set ℓ-waarden met hun bijhorende intervallen.

0-lijst: [0..11]
1-lijst: [0..2], [3..7], [8..10]

2-lijst: [3..4], [5..6]
3-lijst: [8..9]

Deze lijsten zijn automatisch gesorteerd volgens de linker grens van de intervallen doordat de lcp-interval boom in breadth-first volgorde wordt doorlopen. Het aantal ℓ-intervallen in de lijsten is maximaal n. Voor elke lcp-waarde $\ell > 0$ en elk ℓ-interval $[i..j]$ in de ℓ-lijst berekenen we link$[i]$. Door binair zoeken in de (ℓ - 1)-lijst kan het interval $[r..l]$ gevonden worden zodat l de hoogste linkse grens van alle (ℓ - 1)-intervallen waarvoor geldt $l \leq$ link$[i]$. Dit interval is het suffix link interval van $[i..j]$. Bij de eerste ℓ-index van het interval $[i..j]$ worden vervolgens de l en r waarden opgeslagen.
Omdat er minder dan n lcp-intervallen zijn en binair zoeken $O(\log n)$ tijd kost, neemt het algoritme in totaal $O(n \log n)$ tijd in beslag. Tabel SA^{-1}, die we gebruiken om link$[i]$ in constante tijd te berekenen, en de ℓ-lijst hebben elk $O(n)$ space nodig, maar zij kunnen worden verwijderd als de suffix link tabel klaar is.

In ons voorbeeld zoeken we voor twee intervallen, [0..5] en [2..3] het suffix link interval. Voor het eerste interval [0..5] berekenen we de waarde van link[0] = SA^{-1}[SA[0] + 1] = SA^{-1}[3] = 1. Nu zoeken we het interval $[l..r]$ zodat l de grootste linkergrens van alle ℓ - 1 = 0 intervallen waardoor $l \leq$ link[0]. Dit interval moet wel [0..10] zijn, aangezien er geen andere intervallen in de 0-lijst zitten. De waarden $l = 0$ en $r = 10$ worden opgeslagen bij de eerste index van [0..5] waar lcp-waarde 1 staat, index 2. Voor het tweede interval [2..3] doen we dezelfde berekening, zodat link[2] = SA^{-1}[SA[2] + 1] = SA^{-1}[1] = 6. Het gezochte $[l..r]$-interval staat in de 2-lijst, hier gaan we dus via binair zoeken het juiste interval [6..7] vinden. De waarden $l = 6$ en $r = 7$ worden opgeslagen bij de eerste index van [2..3] met een lcp-waarde 3, index 3 dus.

Theoretisch is het mogelijk om de suffix link intervallen te berekenen via de suffix tree, deze kan geconstrueerd worden in $O(n)$ tijd. Het is echter ook mogelijk om zonder de omweg van een suffix tree de suffix link intervallen te berekenen in lineaire tijd. Dit kan gedaan worden door de binaire search te omzeilen en het probleem om de suffix link intervallen te construeren te reduceren tot het beantwoorden van *range minimum queries*. Voor dit algoritme houden we wel de grenzen i en j van elk ℓ-interval $[i..j]$ bij op elke ℓ-index.

Lemma 4. *Gegeven is een ℓ-interval $[i, j]$ met $[l, r]$ zijn suffix link interval. Omdat er een ℓ-index q met $i+1 \leq q \leq j$, is er ook een index k zodat k een $(\ell - 1)$-index is van $[l..r]$ en link$[i] + 1 \leq k \leq$ link$[j]$.*

Omdat $l \leq$ link$[i]$ + 1 $\leq$ link$[j]$ en ℓ - 1 de lengte van de langste gemeenschappelijke prefix is van link$[i]$ en link$[j]$, is ℓ de minimum waarde van de

lcp-tabel in de range [link[i] + 1 ... link[j]]. Hierdoor kunnen we een ($\ell-1$)-index k van interval [$l..r$] met link[i] + 1 $\leq k \leq$ link[j] localiseren door de range minimum query van de range [link[i] + 1 ... link[j]] te beantwoorden.

Definitie 4.3.3. *Laat L een array zijn van groote n met elementen uit de range* $[0, n-1]$. *De* range minimum query *$RMQ(i, j)$, met $0 \leq i < j \leq n-1$, vraagt om een index k zodat $i \leq k \leq j$ en $L[k] = min\ \{L[q] \mid i \leq q \leq j\}$.*

De array L kan in lineaire tijd en space worden geconstrueerd, zoals bij het vorige algoritme wordt de lcp-interval boom doorkruist in breadth-first order. De ℓ-intervallen worden volgens hun waarde in oplopende volgorde afgelopen. Stel dat het ℓ-interval [$i..j$] bekeken wordt en dat alle intervallen met lcp-waarde ℓ - 1 al berekend zijn. Eerst worden de grenzen van het ℓ-interval i en j bij elke ℓ-index van [$i..j$] bewaard. Dan worden link[i] en link[j] bepaald waaruit we de waarde k = RMQ(link[i] + 1, link[j]) berekenen. Omdat k een (ℓ - 1) index van het suffix link interval [$i..j$] is, kunnen we de grenzen l en r van dit suffix link interval op index k opzoeken. Uiteindelijk worden l en r in de suffix link tabel op de eerste ℓ-index van [$i..j$] bewaard.
Elke stap in bovenstaande beschrijving kan in constante tijd worden uitgevoerd, zodat de worst-case tijdscomplexiteit om de suffix link tabel te construeren $O(n)$ tijd bedraagt.

			childtab			suflink			
i	SA	lcptab	1	2	3	l	r	SA^{-1}	$S_{SA[i]}$
0	3	0		1	3			1	acccttca$
1	0	1			2	0	11	8	atcacccttca$
2	10	1						3	a$
3	2	0	1	5	8			0	cacccttca$
4	9	2				0	2	5	ca$
5	4	1	4	6	7	0	11	6	cccttca$
6	5	2				3	7	7	ccttca$
7	6	1	6					10	cttca$
8	1	0	5	10	11			9	tcacccttca$
9	8	3				3	4	4	tca$
10	7	1	9			0	11	2	ttca$
11	11	0	10					11	$

Tabel 4.3: Enhanced suffix array met suffix links

4.4 Vmatch

Vmatch is de software tool die de enhanced suffix array implementeert, het programma kan verschillende soorten matching problemen oplossen zoals zoeken naar maximale repeats, branching tandem repeats, supermaximal repeats of volledige matches.
Verder bestaan er een aantal systemen gebaseerd op de enhanced suffix array, zoals MGA, wat multiple alignments van volledige genomen kan berekenen, ESAsearch, een programma dat zoekt naar position specific scoring matrices of REPuter, een systeem om maximale repeats in genomen te vinden. ESAsearch bespreken we in hoofdstuk 6.6. REPuter had in de eerste versies een suffix tree [Kur99b], deze is echter vervangen door een enhanced suffix array met een significante vermindering van het gebruikte geheugen tot gevolg [AKO02].

4.4.1 Testresultaten

Het algoritme dat gebruikt wordt om de suffix array te construeren is gebaseerd op het suffix sorting algoritme van Bentley en Sedgewick [BS97]. Het zou ongeveer 50% minder plaats innemen dan de normale algoritmes om suffix arrays te construeren, volledig in het werkgeheugen. Het systeem waarop de auteurs de experimenten hebben uitgevoerd is een SUN-Sparc met 32 gigabyte RAM en een 950Mhz CPU. Omdat ons systeem 512MB RAM heeft, kunnen we slechts suffix arrays construeren van sequenties die maximaal 100 miljoen karakters bevatten. De geteste sequenties zijn allemaal aaneengesloten stukken uit het eerste chromosoom van het menselijk genoom, waaruit alle karakters N zijn verwijderd. Vmatch houdt immers enkel rekening met de letters A, C, G, T bij het opstellen van de enhanced suffix array.

Allereerst vergelijken we hoeveel tijd het kost om enkel de suffix array aan te maken met de tijd die nodig is om de volledige index te creëren. De volledige index bestaat uit de suffix array en een set andere arrays waaronder de getransformeerde input sequentie (tistab), de oorspronkelijke input sequentie (oistab), de gereduceerde inverse suffix array (sti1tab), de Burrows-Wheeler transformatie (bwttab), de lengtes van de langste gemeenschappelijke prefixen (lcptab), de bucket boundaries (bcktab) en de skip values (skptab). De belangrijkste van deze arrays zijn besproken in het vorig hoofdstuk.

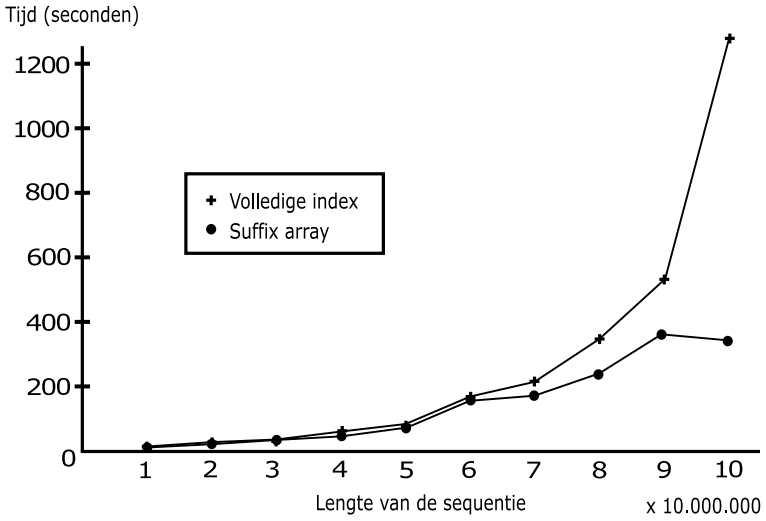
We hebben voor een sequentie van 10 miljoen karakters de groottes van de verschillende arrays samengevat in tabel 4.4, samen met hun relatieve grootte ten opzichte van de sequentie. De totale grootte van de volledige enhanced suffix array bedraagt dus 13,307 MB. De grootte van de suffix

trees geconstrueerd door TOP-Q en TDD (zie hoofdstuk 2.7.3) waren respectievelijk 41,7 MB en 12,9 MB. De volledige enhanced suffix array is dus nauwelijks groter dan de suffix tree geconstrueerd door TDD.
Hierbij moeten we wel nog rekening houden met de toepassing waarvoor de index gebruikt wordt, voor een gewone beslissingsquery hebben we enkel de suffix array, de lcptab, de tistab en bcktab nodig. In totaal zijn er dus slechts $6n + n/2$ bytes nodig om de index voor te stellen indien men enkel beslissingsqueries wil oplossen. In vergelijking met TDD, wat suffix trees van $13n$ bytes construeert, is dit een halvering van de benodigde space.
Een ander voorbeeld is het ESAsearch algoritme dat op zoek gaat naar position specific scoring matrices door gebruik te maken van een enhanced suffix array. Dit programma heeft enkel de suffix array, lcptab en de skptab nodig, voor een totaal van $9n$ bytes.

Naam	grootte (KB)	grootte(n)
tistab	983	n
oistab	983	n
sti1tab	983	n
bwttab	983	n
lcptab	983	n
bcktab	524	$n/2$
skptab	3934	$4n$
SA	3934	$4n$

Tabel 4.4: Grootte van de index arrays bij een sequentie van 10 miljoen karakters

In figuur 4.2 staat het tijdsverschil tussen de creatie van de volledige enhanced suffix array en enkel de suffix array. Eerst hebben we de test uitgevoerd met korte sequenties tussen de 1 en 10 miljoen karakters, in figuur 4.2 met langere sequenties tussen de 10 en 100 miljoen tekens. Voor korte sequenties was het verschil in constructietijd onbeduidend en ook voor de langere sequenties kunnen we vaststellen dat de tijd om enkel de suffix array te maken niet veel verschilt met de constructietijd van de volledige index, ook al neemt de volledige index een veelvoud aan plaats in beslag op de harde schijf. De constructietijd van de volledige index begint echter plots sterk te stijgen bij sequenties die meer dan 80 miljoen karakters bevatten. De oorzaak hiervan is niet meteen duidelijk, de meest logische verklaring is dat voor de constructie van de volledige index het werkgeheugen niet meer toereikend is en er gebruik wordt gemaakt van swap-geheugen. Het opnieuw testen van deze sequenties bevestigden ons vermoeden dat er inderdaad veel swapgeheugen werd aangesproken, we kunnen hieruit besluiten dat indien men een volledige enhanced suffix array wil maken van een sequentie van meer dan 100 miljoen karakters, men meer dan 512MB werkgeheugen nodig heeft.



Figuur 4.2: Constructie van de suffix array en de volledige enhanced suffix array

Indien we deze constructietijden vergelijken met de pipelined algoritmes uit hoofdstuk 3.9, zien we dat de constructietijd van de enhanced suffix array lager ligt dan deze van de pipelined algoritmes DC3, quadrupling en discarding. Eigenlijk is deze vergelijking niet eerlijk, de pipelined constructiealgoritmes gebeuren voor een groot deel op de harde schijf, terwijl de creatie van de enhanced suffix array volledig in het werkgeheugen plaatsvindt.

Tot slot hebben we geprobeerd te achterhalen of er een verschil in snelheid bestaat bij het zoeken naar exacte maches in een suffix tree en een enhanced suffix array. Hiervoor hebben we eerst voor een sequentie van 1 miljoen karakters beide indexen gemaakt en 10000 query-sequenties gegenereerd door random subsequenties met lengte 10 en lengte 100 te kiezen uit de basissequentie. Elk van de query-sequenties komt dus minimaal één keer voor in de sequentie. Deze set queries hebben we uitgevoerd op de suffix tree gegenereerd door TOP-Q en de enhanced suffix array. Beide indexen werden op voorhand in het werkgeheugen geladen.
Het uitvoeren van de query-sequenties van lengte 10 duurde bij de suffix tree 26 seconden en bij de enhanced suffix array 4 seconden. De langere uitvoeringstijd bij de suffix tree is te verklaren door het feit dat bij een query-sequentie die veel voorkomt in de sequentie, alle bladeren onder de node met als path de query-sequentie moeten worden afgelopen. Deze bladeren zijn meestal niet aaneengesloten opgeslagen, in tegenstelling tot een suffix array. De enhanced suffix array kan de queries sneller beantwoorden omdat deze index-structuur een betere locality of memory reference heeft.
Indien we 10000 queries uitvoeren met lengte 100 stijgt de snelheid, de suffix

array weet deze queries te beantwoorden in 7 seconden en de enhanced suffix array in minder dan 1 seconde. Deze betere uitvoeringstijd is te verklaren door het feit dat er veel minder posities moeten worden opgevraagd, de kans dat in een sequentie meerdere keren een subsequentie van 100 tekens voorkomt, is klein. De relatieve snelheidswinst is voor beide systemen gelijk, de queries met lengte 100 worden 4 keer sneller beantwoord dan de queries met lengte 10.

Onze conclusie is dan ook dat de enhanced suffix array een goed alternatief is voor een suffix tree, voor de meeste algoritmes moet slechts een deel van tabellen van de enhanced suffix array worden gebruikt. Hierdoor wordt dikwijls minder geheugen gebruikt dan wanneer men een suffix tree zou gebruiken. Het enige nadeel is dat men voor grotere sequenties de enhanced suffix array op een systeem met extreem veel RAM moet construeren. Aan dit probleem kan echter verholpen worden indien men kiest voor een algoritme dat in staat is de suffix array in extern geheugen te construeren, een algoritme zoals DC3 of quadrupling. De querytijd voor exacte matches is beter in vergelijking met een array-gebaseerde suffix tree, maar verdere testen op andere voorstellingen van suffix trees zouden nuttig zijn.

Hoofdstuk 5

Sequentie analyse

We beschouwen in dit hoofdstuk niet alle aspecten van sequentie analyse maar leggen de nadruk op enkele belangrijke algoritmes om sequenties te aligneren. Enkele van deze algoritmes worden in hoofdstuk 6 gebruikt in systemen om homologieën te vinden tussen sequenties. We bespreken systemen om te illustreren hoe suffix trees en suffix arrays in de praktijk gebruikt kunnen worden om sequenties te analyseren. De definities voor dit hoofdstuk komen voor een groot deel uit [MDS04].

Alignments worden in de moleculaire biologie gebruikt om homologieën tussen twee sequenties op te sporen. Door sequenties te vergelijken en er homologieën in terug te vinden kan men uit één sequentie waarvan de functie bekend is, de biochemische rol van de andere sequentie afleiden. Een andere toepassing is om door de similarity, de maat van overeenkomst, tussen twee sequenties te berekenen af te leiden of de beide organismen een gemeenschappelijke voorouder hebben, en hoe lang geleden die voorouder heeft bestaan. We spreken van een homologie indien volgens statistische methodes is vastgesteld dat twee stukken sequentie erg gelijkaardig zijn. Om te weten hoe gelijkaardig twee sequenties zijn gebruiken we de edit distance, dit model drukt uit hoeveel edit operaties er nodig zijn om één sequentie om te vormen tot een andere sequentie. Er bestaan drie soorten edit operaties: $\alpha \rightarrow \epsilon$ wat de verwijdering van karakter α betekent (deletion), $\epsilon \rightarrow \beta$ wat de invoeging van karakter β betekent (insertion) en $\alpha \rightarrow \beta$ wat wil zeggen dat karakter α door karakter β wordt vervangen (replacement of substitution). Verwijderingen en invoegingen worden ook wel *indels* genoemd, naar insert en delete.

Definitie 5.0.1. *Een alignment A van sequenties u en v is de sequentie $(\alpha_1 \rightarrow \beta_1, \ldots, \alpha_h \rightarrow \beta_h)$ edit operaties waarbij $u = \alpha_1 \ldots \alpha_h$ en $v = \beta_1 \ldots \beta_h$.*

Een alignment wordt meestal voorgesteld door de twee sequenties in kwestie boven elkaar te plaatsen en bij een indel een spatie in de juiste sequentie

```
t g t c c t a c _
_ g t t c _ a t t
```

Figuur 5.1: Voorbeeld van een alignment

te voegen. Zo kan visueel worden vastgesteld op welke plaatsen de sequenties het best matchen. We nemen als voorbeeld de sequenties `tgtcctac` en `gttcatt`, die worden uitgelijnd volgens alignment A = ($t \to \epsilon$, $g \to g$, $t \to t$, $c \to t$, $c \to c$, $t \to \epsilon$, $a \to a$, $c \to t$, $\epsilon \to t$). De alignment wordt visueel voorgesteld in figuur 5.

Definitie 5.0.2. *Een kost functie δ geeft aan elke edit operatie $\alpha \to \beta$, met $\alpha \neq \beta$ een positieve kost $\delta(\alpha \to \beta)$. De kost van een edit operatie $\alpha \to \alpha$ is 0. Indien $\delta(\alpha \to \beta) = \delta(\beta \to \alpha)$ noemen we δ symmetrisch. De kost $\delta(A)$ van een alignment $A = (\alpha_1 \to \beta_1, \ldots, \alpha_h \to \beta_h)$ is de som van alle kosten van de edit operaties binnen A.*

$$\delta(A) = \sum_{i=1}^{h} \delta(\alpha_i \to \beta_i).$$

Definitie 5.0.3. *De edit distance van u en v, $edist_\delta(u, v)$ genoemd, is de kleinste mogelijke kost van een alignment van u en v.*
Kortom: $edist_\delta(u, v) = min\ \{\delta(A) \mid A$ is een alignment van u en $v\}$.

We kunnen de edit distance berekenen door een matrix E_δ met afmetingen $(m + 1) \times (n + 1)$ waarin elke cel als volgt wordt gedefinieerd:

$$E_\delta(i, j) = edist_\delta(u_1 \ldots u_i, v_1 \ldots v_j)$$

Het algoritme dat de edit distance berekent wordt in de bioinformatica het *dynamic programming (DP)* algoritme genoemd, omdat het resultaat wordt berekend door dynamisch programmeren. Bij dynamisch programmeren wordt een optimale oplossing afgeleid van optimale suboplossingen, de optimale oplossing kan bekomen worden door suboplossingen telkens verder uit te werken.
De tabel E_δ kan ingevuld aan de hand van volgende regels:

- $E_\delta[0, 0] = 0$
- $E_\delta[i + 1, 0] = E_\delta[i, 0] + \delta(u_{i+1} \to \epsilon)$
- $E_\delta[0, j + 1] = E_\delta[0, j] + \delta(\epsilon \to v_{j+1})$
- $E_\delta(i + 1, j + 1) = \min \begin{cases} E_\delta[i, j + 1] + \delta(u_{i+1} \to \epsilon) \\ E_\delta[i + 1, j] + \delta(\epsilon \to v_{j+1}) \\ E_\delta[i, j] + \delta(u_{i+1} \to v_{j+1}) \end{cases}$

```
t g t c c t a c          t g t c c t a c
g t t c a t t _          _ g t t c a t t
```

Figuur 5.2: Optimale alignments met edit distance 6

Per definitie geeft $E_\delta[m, n]$ de edit distance van u en v. Om de optimale alignments te berekenen moeten we van uit entry $E_\delta[m, n]$ backtracken naar $E_\delta[0, 0]$. Bij het backtracken vanuit $E_\delta[m, n]$ moet voor elke $E_\delta[i, j]$ de minimale waarde $E_\delta[i', j']$ gekozen worden zodat $E_\delta[i, j] = E_\delta[i', j'] + \delta(\alpha \to \beta)$. Indien $E_\delta[i, j] = E_\delta[i-1, j-1] + \delta(u[i] \to v[j])$ hebben we te maken met een substitutie, als $E_\delta[i, j] = E_\delta[i-1, j] + \delta(u[i] \to \epsilon)$ spreken we over een deletion, indien $E_\delta[i, j] = E_\delta[i, j-1] + \delta(\epsilon \to v[j])$ hebben we een insertion. We werken de alignering van de sequenties `tgtcctac` en `gttcatt` uit, als score-functie gebruike we +1 voor een mismatch en +2 voor een gap. Om de edit distance te berekenen tussen u en v heeft elke match de waarde 0, de edit distance tussen dezelfde karakters is immers nul.

			t	g	t	c	c	t	a	c
$E_\delta[i, j]$		0	1	2	3	4	5	6	7	8
	0	0	2	4	6	8	10	12	14	16
g	1	2	1	2	4	6	8	10	12	14
t	2	4	2	2	2	5	7	8	10	12
t	3	6	4	3	2	3	5	7	9	11
c	4	8	6	5	4	2	3	5	7	9
a	5	10	8	7	6	4	3	4	5	7
t	6	12	10	9	7	6	5	5	5	6
t	7	14	12	11	9	8	7	5	4	6

Tabel 5.1: Berekenen van de optimale edit-distance

Uit deze tabel kunnen we opmaken dat de minimale edit distance voor deze score-functie 6 bedraagt. In figuur 5.2 staan de twee optimale alignments met edit distance 6.
Het bovenstaande DP-algoritme om de edit distance te berekenen kan niet omgaan met positieve scores bij een match, een match heeft een neutrale score. Het Needleman-Wunsch algoritme verschilt met het vorige algoritme in de manier waarmee het omgaat met gaps en mismatches, deze zijn bij het Needleman-Wunsch algoritme namelijk negatief, terwijl een positieve waarde kan gegeven worden aan een match.
Bij het Needleman-Wunsch algoritme wordt de matrix W bepaald door volgende regels:

- $W[0, 0] = 0$
- $W[i+1, 0] = W[i, 0] + \delta(u_{i+1} \to \epsilon)$

- $W[0, j+1] = W[0, j] + \delta(\epsilon \rightarrow v_{j+1})$
- $W(i+1, j+1) = \max \begin{cases} W[i, j+1] + \delta(u_{i+1} \rightarrow \epsilon) \\ W[i+1, j] + \delta(\epsilon \rightarrow v_{j+1}) \\ W[i, j] + \delta(u_{i+1} \rightarrow v_{j+1}) \end{cases}$

Als we als score-functie een -2 voor een gap, een -1 voor een mismatch en +1 voor een match gebruiken, verkrijgen we voor de sequenties `tgtcctac` en `gttcatt` volgende matrix.

			t	g	t	c	c	t	a	c
W		0	1	2	3	4	5	6	7	8
	0	0	-2	-4	-6	-8	-10	-12	-14	-16
g	1	-2	-1	-1	-3	-5	-7	-9	-11	-13
t	2	-4	-1	-2	0	-2	-4	-6	-8	-10
t	3	-6	-3	-2	-1	-1	-3	-3	-5	-7
c	4	-8	-5	-4	-3	0	0	-2	-4	-4
a	5	-10	-7	-6	-5	-2	-1	-1	-1	-3
t	6	-12	-9	-8	-5	-4	-3	0	-2	-2
t	7	-14	-11	-10	-7	-6	-5	-2	-1	-3

Tabel 5.2: Needleman-Wunsch matrix voor een globale alignment

De optimale alignments bepalen werkt zoals bij het DP-algoritme, startend vanuit positie W$[m, n]$ wordt voor elke positie W$[i+1, j+1]$ nagegaan hoe de waarde is berekend, vanuit W$[i, j]$ (substitutie), vanuit W$[i+1, j]$ (deletion) of vanuit W$[i, j+1]$(insertion). De optimale alignments voor deze score-functie staan in figuur 5.3

```
t g t c c t a c     t g t c c t a c     t g t c c t a c
g t t c a t t _     _ g t t c a t t     g t t c a t _ t
```

Figuur 5.3: Optimale alignments voor het Needleman-Wunsch algoritme

5.1 Local alignment

In vorig hoofdstuk werden volledige sequenties gealigneerd, maar om relaties tussen biologische sequenties te zoeken is het soms interessanter om alleen de coding regions met elkaar te vergelijken. Een DNA-sequentie bevat immers veel *junk*-DNA, wat sneller muteert dan DNA in een coding region. Locaal aligneren betekent dat er in twee sequenties gezocht wordt naar alle paren substringen die sterk op elkaar lijken, in plaats van de sequenties volledig met elkaar te aligneren.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	4	-1	-2	-2	0	-1	-1	0	-2	-1	-1	-1	-1	-2	-1	1	0	-3	-2	0
R	-1	5	0	-2	-3	1	0	-2	0	-3	-2	2	-1	-3	-2	-1	-1	-3	-2	-3
N	-2	0	6	1	-3	0	0	0	1	-3	-3	0	-2	-3	-2	1	0	-4	-2	-3
D	-2	-2	1	6	-3	0	2	-1	-1	-3	-4	-1	-3	-3	-1	0	-1	-4	-3	-3
C	0	-3	-3	-3	9	-3	-4	-3	-3	-1	-1	-3	-1	-2	-3	-1	-1	-2	-2	-1
Q	-1	1	0	0	-3	5	2	-2	0	-3	-2	1	0	-3	-1	0	-1	-2	-1	-2
E	-1	0	0	2	-4	2	5	-2	0	-3	-3	1	-2	-3	-1	0	-1	-3	-2	-2
G	0	-2	0	-1	-3	-2	-2	6	-2	-4	-4	-2	-3	-3	-2	0	-2	-2	-3	-3
H	-2	0	1	-1	-3	0	0	-2	8	-3	-3	-1	-2	-1	-2	-1	-2	-2	2	-3
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4	2	-3	1	0	-3	-2	-1	-3	-1	3
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4	-2	2	0	-3	-2	-1	-2	-1	1
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5	-1	-3	-1	0	-1	-3	-2	-2
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5	0	-2	-1	-1	-1	-1	1
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6	-4	-2	-2	1	3	-1
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7	-1	-1	-4	-3	-2
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4	1	-3	-2	-2
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5	-2	-2	0
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11	2	-3
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	-1
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4

Tabel 5.3: BLOSUM62 similarity score matrix voor aminozuren

Definitie 5.1.1. *Een score functie σ geeft aan elke edit operatie $\alpha \to \beta$ een score $\sigma(\alpha, \beta)$ heeft waarbij gelijkaardige symbolenparen een positieve waarde krijgen en ongelijke paren een negatieve. De score van een alignment $A = (\alpha_1 \to \beta_1, \ldots, \alpha_h \to \beta_h)$ wordt als volgt gedefinieerd:*

$$\sigma(A) = \sum_{k=1}^{h} \sigma(\alpha_k \to \beta_k) + g \times \gamma,$$

met g het aantal gaps in A en $\gamma \leq 0$ de kost per gap.

Indien men de scores tussen elk symbolenpaar in een tabel opslaat spreken we van een similarity matrix. Een voorbeeld van een similarity matrix is de BLOSUM62 matrix om proteïneparen te vergelijken, zie tabel 5.3. Om een volledige score functie te hebben heeft men natuurlijk ook nog scores nodig voor invoegingen en verwijderingen. Bij het alignen van DNA-sequenties worden dikwijls ook heel eenvoudige score functies gebruikt, bijvoorbeeld +1 voor een match, -1 voor een mismatch en -2 voor een gap.
Biologisch gezien is komt het minder voor dat in een sequentie g gaps worden gemaakt dan dat op één plaats een gap van g karakters ontstaat, een gap extenden zou dus minder kostelijk moeten zijn dan een gap maken. We kunnen definitie 5.1.1 dan ook aanpassen zodat de kost van een gap van lengte g gelijk is aan $\gamma_1 + \gamma_2 \times (g - 1)$. De kost om een gap te beginnen is dan γ_1, om een gap te extenden bedraagt de kost γ_2, waarbij normaal gezien $\gamma_1 > \gamma_2$.

Definitie 5.1.2. *Het locale alignment probleem bestaat erin een alignment A te bepalen van u' en v' waarbij u' en v' substrings zijn van u en v en de score van A maximaal is:*

$\sigma(u, v) = max\ \{\sigma(A) \mid A\ is\ een\ alignment\ van\ u'\ en\ v'\ \}$
Elke alignment die voldoet aan deze voorwaarde is een locale optimale alignment van u en v, $\sigma(u, v)$ wordt de optimale local alignment score genoemd.

De brute-force oplossing zou er in bestaan voor elk paar substrings (u', v') van u en v de waarde van $score_\sigma(u', v')$ te berekenen. Omdat er $O(n^2m^2)$ paren substringen (u', v') bestaan waarvan de $score_\sigma(u', v')$ berekenen telkens $O(mn)$ kost, bedraagt de totale tijdscomplexiteit $O(n^3m^3)$.

5.1.1 Smith-Waterman

Smith en Waterman hebben een efficiënter algoritme gepresenteerd [SW81] dat uitgaat van de volgende observatie. Als we bij het brute-force algoritme de score van (u', v') hebben kunnen we hieruit de score van ($u'a$, $v'b$) met minimale moeite berekenen, waarbij a en b de karakters zijn in u en v volgend op respectievelijk u' en v'. Het idee is dus om een matrix te maken die per entry (i, j) de score bevat van de tot dan toe optimale alignment van de suffixen van de substrings ($u'_0 \ldots u'_i$, $v'_0 \ldots v'_j$).
We berekenen de matrix L met afmetingen $(m+1) \times (n+1)$ en vullen ze in op de volgende manier:

- Indien j = 0 of i = 0 dan geldt L$(i, j) = 0$
- In de andere gevallen geldt

$$L(i,j) = max \begin{cases} L(i, j-1) + \sigma(\epsilon \rightarrow v_j) \\ L(i-1, j-1) + \sigma(u_i \rightarrow v_j) \\ L(i-1, j) + \sigma(u_i \rightarrow \epsilon) \\ 0 \end{cases}$$

In de matrix L komen dus geen negatieve waarden voor, substrings die similair zijn worden voorgesteld als groepjes positieve waarden. Om nu een locale optimale alignment te vinden moet men op zoek gaan naar hoge waardes $L(i, j)$. Vervolgens moet vanaf deze waarde hetzelfde recursieve algoritme als bij Needleman-Wunsch worden toegepast om de optimale alignment te vinden. Bij Smith-Waterman moet men de locale alignment echter stoppen van zodra men op een waarde $L(i', j') = 0$ komt. De tijdscomplexiteit voor dit algoritme bedraagt eveneens $O(mn)$, of $O(n^2)$ indien m en n ongeveer hetzelfde is.

We illustreren het Smith-Waterman algoritme met een voorbeeld, we zoeken local alignments tussen de sequenties `tgtcctac` en `gctacgtc` .
Uit de matrix in tabel 5.4 kunnen we twee alignments opmaken met een score van 3 of hoger. Deze alignments worden geïllustreerd in figuur 5.4, boven en onder de subsequenties staat de index van de karakters in de oorspronkelijke sequenties.

			t	g	t	c	c	t	a	c
$L[i,j]$		0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
a	1	0	0	0	0	0	0	0	1	0
c	2	0	0	0	0	1	1	0	0	2
t	3	0	1	0	1	0	0	2	0	0
a	4	0	0	0	0	0	0	0	3	1
c	5	0	0	0	0	1	1	0	1	4
g	6	0	0	1	0	0	0	0	0	2
t	7	0	1	0	2	0	0	1	0	0
c	8	0	0	0	0	3	1	0	0	1

Tabel 5.4: Smith-Waterman matrix voor een local alignment

```
2 3 4          5 6 7 8
g t c          c t a c
g t c          c t a c
6 7 8          2 3 4 5
```

Figuur 5.4: Locale alignments van het Smith-Waterman algoritme

5.1.2 Position Specific Scoring Matrices

Een position specific scoring matrix (PSSM) probeert de overeenkomsten en verschillen tussen een set sequentie-patronen op een statistische manier te omschrijven. De bedoeling is om een profiel op te stellen van sequenties die eenzelfde functie hebben. Door de sequenties te aligneren en te bepalen waar de overeenkomsten en verschillen zitten kan men in een query-sequentie subsequenties zoeken die sterke overeenkomsten hebben met het opgestelde profiel. De kans is groot dat de gevonden subsequenties dezelfde functie hebben als de sequenties waarop de position specific scoring matrix is gebaseerd.

Definitie 5.1.3. *Gegeven een set S van n gealigneerde sequenties met lengte ℓ, $S_1, \ldots, S_\ell$ waarbij elke S_k bestaat uit karakters $S_{k0}, \ldots, S_{k(\ell-1)}$, kunnen we de* position weight matrix *als volgt berekenen:*

$$PWM_{b,j} = \sum_{k=1}^{n} I_b(S_{kj})$$

waarbij b de karakters van het alfabet zijn (meestal basen of proteïnen), $j = 1 \ldots \ell$ en

$$I_b(q) = \begin{cases} 1 \text{ indien } b = q \\ 0 \text{ anders} \end{cases}$$

In deze matrix worden de absolute voorkomens van een karakter op een bepaalde plaats weergegeven. Het is echter gebruikerlijker om de relatieve frequentie te gebruiken in een profiel. Hiervoor moet dus elke waarde gedeeld worden door het aantal gebruikte sequenties.
De ratio tussen de waarschijnlijkheid van een sequentie op een functionele site en de waarschijnlijkheid in een random site is de *likelihood ratio*. Indien de ratio nul is betekent dit dat de sequentie evenveel kans heeft om in een random als een functionele site voor te komen. Bij een positieve ratio gaat de sequentie meer voorkomen in een functionele dan een random site. Van deze ratio kan men ook het logaritme berekenen, dan hebben we het over de log-likelihood ratio.

Bovenstaand model gaat uit van een gelijke verdeling tussen alle basen of proteïnes, maar in de praktijk komen basen A en T veel meer voor dan C en G. In de volgende vergelijking wordt hiermee rekening gehouden:

$$M_{b,j} = f_{b,j} \log_2 \frac{f_{b,j}}{p_b}$$

waarbij p_b de frequentie is van base b in het hele genoom, $f_{b,j}$ is de geobserveerde frequentie van een base b op positie j.
Deze vergelijking is een genormaliseerde log-likelihood ratio die kan gebruikt worden om de statistische significantie van een patroon uit te drukken. Gegeven een sequentie van lengte ℓ kan nu de log-likelihood ratio worden uitgerekend door de som te nemen van alle coëfficienten uit de matrix die overeenkomen qua karakter en positie met de sequentie. Normaal wordt dus met een window met grootte ℓ over een sequentie geschoven en wordt van elk window de log-likelihood ratio berekend. Indien deze waarde boven een bepaalde threshold ligt is de kans groot dat de huidige site een biologische verwantschap heeft met de sequenties uit het profiel.

Definitie 5.1.4. *Voor een sequentie w met lengte m definiëren we*

$$sc(w, M) = \sum_{i=0}^{m-1} M(i, w[i])$$

als de match score van w ten opzichte van M.

Definitie 5.1.5. *Gegeven een sequentie S van lengte n over alfabet Σ en een score threshold th, is het* PSSM searching probleem *het vinden van alle posities $j \in [0, n-m]$ in S waarvoor geldt $sc(S[j..j+m-1], M) \geq th$.*

Een voor de hand liggend algoritme om dit probleem op te lossen bestaat er in om met een vester van grootte m langs de sequentie S te schuiven en voor elke subsequentie $w = S[j..j+m-1]$ met $j \in [0, n-m]$ de score $sc(w, M)$ te berekenen. De tijdscomplexiteit van dit algoritme bedraagt uiteraard $O(nm)$ tijd.

Dit algoritme kan versneld worden door gebruik te maken van lookahead scoring [WNMB00]. Bij deze techniek wordt de berekening van $sc(w, M)$ gestaakt wanneer het duidelijk is dat de totale score niet boven de threshold th kan komen.

Definitie 5.1.6. *De prefix score* $\text{pfxsc}_d(w, m)$ *met lengte* d *wordt als volgt gedefinieerd:*

$$\text{pfxsc}_d(w, M) = \sum_{h=0}^{d} M(h, w[h])$$

De maximale score die kan behaald worden in de laatste $m - d - 1$ *posities van de PSSM noemen we* σ_d*:*

$$\sigma_d = \sum_{h=d+1}^{m-1} max_h \text{ met } max_d = max\{M(d, a) \mid d \in \Sigma\} \text{ en } d \in [0, m-1]$$

Indien $th_d = th - \sigma_d$ de intermediaire threshold is op positie d, is het eenvoudig aan te tonen dat $sc(w, M) \geq th$ impliceert dat $pfxsc_d(w, M) \geq th_d$ voor alle $d \in [0, m-1]$. Dus wanneer $sc(w, M)$ wordt berekend moet voor elke d bekeken worden of de intermediaire threshold th_d wordt bereikt.
Met deze methode wordt een praktische snelheidswinst geboekt, we noemen k het gemiddelde aantal PSSM-posities per sequentie die worden berekend. Uiteraard is $k \leq m$, waardoor het algoritme in $O(kn)$ tijd werkt. De worst-case tijdscomplexiteit blijft $O(mn)$.

We werken een voorbeeld uit, stel dat van de sequenties in tabel 5.5 geweten is dat ze een gelijkaardige functie hebben. Van deze sequenties stellen we de position weight matrix samen, zie tabel 5.6. Uit deze matrix berekenen we de log-likelihood matrix in tabel 5.7, we gaan er van uit dat elk karakter een even grote kans heeft om voor te komen.
We kunnen de opgestelde matrix als volgt gebruiken. We schuiven met een window met grootte 9 over een sequentie, per window berekenen we de log-likelihood ratio, indien deze boven de threshold van 2 beschouwen we dit als een match. Neem bijvoorbeeld de sequentie `...ATCTgaggtgaagGAAT...`, de karakters waar het window over staan in kleine letters. We berekenen de score van `gaggtgaag` gegeven het log-likelihood profiel. De berekening van deze score zien we in tabel 5.8. De samengetelde score bedraagt 2.72, wat hoger is dan de threshold.

Stel dat het window over de subsequentie `gcttggagt` geschoven is, dan kunnen we na positie 4 al constateren dat de threshold 2 niet meer gehaald kan worden, de prefix score $pfxsc_3(gcttggtac, 9) = -\infty$. De maximale score σ_3 die kan behaald worden in de laatste 5 posities van de PSSM bedraagt 4.83, wat uiteraard te weinig is om de totale score nog positief te krijgen. Dus na het vierde karakter kan men al stoppen met de score te berekenen. De berekening van de score in elke positie van de sequentie `gcttggagt` staat in tabel 5.9.

Sequentie 1	gaggtaaac
Sequentie 2	tccgtaagt
Sequentie 3	caggttgga
Sequentie 4	acagtcagt
Sequentie 5	taggtcatt
Sequentie 6	taggtactg
Sequentie 7	atggtaact
Sequentie 8	caggtatac
Sequentie 9	tgtgtgagt
Sequentie 10	aaggtaagt

Tabel 5.5: Tien sequenties met een gelijkaardige functie

	0	1	2	3	4	5	6	7	8
A	3	6	1	0	0	6	7	2	1
C	2	2	1	0	0	2	1	1	2
G	1	1	7	10	0	1	1	5	1
T	4	1	1	0	10	1	1	2	6

Tabel 5.6: De position weight matrix

	0	1	2	3	4	5	6	7	8
A	0.18	0.87	-0.91	$-\infty$	$-\infty$	0.87	1.02	-0.22	-0.91
C	-0.22	-0.22	-0.91	$-\infty$	$-\infty$	-0.22	-0.91	-0.91	-0.22
G	-0.91	-0.91	1.02	1.38	$-\infty$	-0.91	-0.91	0.69	-0.91
T	0.47	-0.91	-0.91	$-\infty$	1.38	-0.91	-0.91	-0.22	0.87

Tabel 5.7: Het profiel als een log-likelihood matrix

	g 0	a 1	g 2	g 3	t 4	g 5	a 6	a 7	g 8
A		0.87					1.02	-0.22	
C									
G	-0.91		1.02	1.38		-0.91			-0.91
T					1.38				

Tabel 5.8: Berekening van de score van de sequentie `gaggtgaag`

	g 0	c 1	t 2	t 3	g 4	g 5	a 6	g 7	t 8
A							1.02		
C		-0.22							
G	-0.91				$-\infty$	-0.91		0.69	
T			-0.91	$-\infty$					0.87

Tabel 5.9: Berekening van de score van de sequentie `gcttggagt`

Hoofdstuk 6

Alignment Programma's

In deze laatste sectie beschouwen we een aantal algoritmes en programma's om sequenties te aligneren. Eerst omschrijven we BLAST, wat één van de meest gebruikte tools is om sequenties locaal te aligneren. Vervolgens beschouwen we een aantal systemen die gebruik maken van een suffix tree of suffix array. QUASAR maakt gebruik van een suffix array om snel locaties te kunnen vinden waar een deel van de query mee overeen komt. De locatie zelf wordt verder uitgelijnd door BLAST. De suffix sequoia is een op zichzelf staand systeem dat door middel van een suffix tree-achtige structuur locale alignments gaat zoeken, waarbij de kans dat alignments, die voldoen aan een bepaalde threshold, niet opgemerkt worden zo goed als onbestaand is. Oasis gaat eveneens sequenties locaal aligneren en gebruikt daarbij een suffix tree, het programma geeft resultaten terug volgens hun score. MUMmmer is dan weer een programma om volledige genomen globaal te aligneren, het programma maakt gebruik van een suffix tree. Het laatste systeem dat we bespreken is PSSMsearch, het programma gebruikt een enhanced suffix array om via PSSM-scores sequenties locaal te aligneren.

6.1 BLAST

BLAST, wat staat voor basic local alignment search tool, is een programma om lokale overeenkomsten te zoeken tussen een query-sequence en een andere veel grotere databasesequentie. Het programma aligneert stukken van de query-sequence met de databasesequentie en rangschikt de resultaten volgens S-score. BLAST is bijzonder populair bij biologen omdat het veel sneller is dan het algoritme van Smith-Waterman. Dit laatste algoritme heeft een tijdscomplexiteit van $O(mn)$, met m en n de lengte van respectievelijk de query-sequentie en de database-sequentie. BLAST daarentegen gebruikt een heuristiek om hoog scorende locale alignments te vinden. Hierdoor kan echter geen garantie gegeven worden dat alle alignments met een bepaalde score worden gevonden, maar men kan de kans dat een alignment

met een bepaalde score gemist wordt wel willekeurig klein maken.

6.1.1 De oorspronkelijke BLAST

Als eerste stap gaat BLAST de query-sequentie opsplitsen in woorden met een vaste lengte w die men vervolgens probeert terug te vinden in de database-sequentie. Als deze woord-paren een score hebben van tenminste T spreken we van een hit of een seed. Zulk een woord-paar probeert men vervolgens aan beide zijden uit te breiden om uit te maken of dat woord-paar deel is van een segment pair met een score gelijk aan of groter dan S. Hoe lager de threshold T, hoe meer kans dat er segment pairs gevonden gaan worden met een score van ten minste S. Aan de andere kant gaat een kleine waarde voor T ook veel meer hits produceren waardoor de uitvoertijd van het algoritme de hoogte in schiet. Deze afweging tussen snelheid en precisie komen vaak terug in de discussies over de verschillende BLAST-versies.

In het algemeen gaat een wetenschapper de Maximal Segment Pairs (MSP) zoeken die een score hebben boven een bepaalde threshold S. De belangrijkste vraag is nu: tot welke waarde S gaan er geregeld MSP's zijn die per toeval zijn ontstaan in plaats van een biologische oorzaak te hebben. Anders gezegd, als we tussen twee puur random gegenereerde sequenties met lengte n en m de MSP's gaan zoeken, hoe groot is de kans dat er een MSP gevonden wordt met een score groter dan bijvoorbeeld 100?

Er bestaat een verschil in taktiek om bij proteïne en DNA sequenties hits te zoeken. Bij proteïne sequenties wordt een lijst gemaakt van alle woorden (w-mers genoemd) die een score van ten minste T hebben wanneer ze worden uitgelijnd met een woord uit de query sequentie. Als score matrix wordt meestal de PAM120 matrix gebruikt.
Voor DNA bestaat de word-list uit alle substringen uit de query sequentie met lengte w die wanneer ze met zichzelf worden uitgelijnd een score halen van ten miste T. Voor DNA werd simpelweg de score +5 voor een match en -4 voor een mismatch aangeraden. Een query sequentie met lengte n geeft dus een lijst van $n - w + 1$ woorden, waarbij w meestal 12 is.
Van zodra er een lijst is gemaakt met woorden worden alle voorkomens van deze woorden in de database-sequentie gezocht door een lineaire scan. Alle matches worden dan maximaal verlengd en degene met een score boven S worden als output gegeven.
Dit algoritme werkt vrij goed als we te maken zouden hebben met sequences die uit random gegenereerde karakters bestaan. Jammer genoeg is DNA niet random, bepaalde stukken van DNA zijn repetitief of bepaalde letters komen veel meer voor. Hierdoor worden veel hits gegenereerd die weinig biologische waarde hebben. De ad-hoc oplossing die in de oorspronkelijke paper werd gebruikt is te kijken welke woorden aan een hogere frequentie voorkomen

dan verwacht wordt in een random sequentie. Deze woorden worden dan uit de lijst gefilterd. Er wordt ook een lijst gemaakt van repetitieve elementen, de woorden die in deze substringen voorkomen worden eveneens verwijderd van de lijst. Merk op dat wanneer er naast zulk een repetitieve substring een hit is, het woord-paar wel kan worden uitgebreid tot in die repetitieve substring [MY03].

6.1.2 Statistische betekenis van MSPs

BLAST is gebouwd op een aantal statistische formules. In 1990 publiceerde Samuel Karlin en Stephen Altschul de paper *Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes* [KA90], waarin de basis voor BLAST wordt gelegd. In het kort gaat het erom dat, gegeven een set kansen dat een symbool voorkomt en een set scores om karakter-paren te aligneren, de theorie twee parameters λ en K geeft om de statistische significantie van de score van een Maximal Segment Pair (MSP) te evalueren.

In hun statistische formules worden twee veronderstellingen gemaakt over score matrices, namelijk dat een positieve alignment score mogelijk is maar dat de verwachte score negatief moet zijn. Dit komt overeen met een score matrix voor het Smith-Waterman algoritme. Drie andere veronderstellingen zijn iets minder realistisch in de biologie, namelijk dat de karakters van de sequenties onafhankelijk ten opzichte van elkaar zijn, dat de sequenties oneindig lang zijn en dat er enkel substituties kunnen voorkomen (gaps zijn dus uitgesloten). Jammer genoeg zijn de karakters van biologische sequenties dikwijls afhankelijk van elkaar (bijvoorbeeld na een A komt een T meer voor dan een G), zijn de sequenties eindig en tijdens de evolutie kunnen er basen wegvallen of bijkomen. In het volgend hoofdstuk komen we nog terug op gapped alignments.

Wanneer twee random gegenereerde sequenties van lengte m en n worden vergeleken, is de waarschijnlijkheid om een segment pair te vinden met een score die minstens S bedraag als volgt te berekenen:

$$1 - e^{-E} \; waarbij \; E = Kmne^{-\lambda S} \qquad (1)$$

De parameter λ dient om de gebruikte score-matrix te normaliseren, het is een constante die het inverse van de oorspronkelijke schaleringsfactor moet benaderen. De constante K moet de mogelijkheid in rekening brengen dat optimale locale alignments die op een verschillende plaats in de twee sequenties beginnen, toch aan elkaar relateerd kunnen zijn. Stel bijvoorbeeld dat we een locale alginment met een hoge score hebben die begint op posities x_1 en x_2 van de twee sequenties. De kans is dan groot dat op de respectievelijke posities $x_1 + 3$ en $x_2 + 3$ ook een alignment begint die een hoge score haalt.

De waarde van K ligt meestal rond 0.1, zijn invloed op de statistiek van de alignments is vrij klein.

De kans om c of meer verschillende segment pairs met een score groter dan S te vinden wordt uitgedrukt door de volgende formule:

$$1 - e^{-E} \sum_{i=0}^{c-1} \frac{E^i}{i!} \qquad (2)$$

Dankzij formule (1) kunnen we weten wanneer een MSP toevallig matcht, of dat er sprake is van een biologische verwantschap tussen de sequenties. Alleen kijken naar de p-waarde van een MSP is niet genoeg, men moet de lengte van de sequenties in rekening brengen. Stel bijvoorbeeld dat we een MSP hebben met een p-waarde hebben van 0.001, dan lijkt het op het eerste zicht zeer onwaarschijnlijk dat er segment pairs gevonden kunnen worden in random sequenties. Als we echter te maken hebben met sequentiedatabases met miljarden karakters, is het statistisch zeer waarschijnlijk dat er verschillende segment pairs gevonden gaan worden, puur door toeval.
BLAST steunt op het idee dat een segment pair een woordpaar van lengte w bevat dat een score van tenminste T heeft. Het is interessant om te weten hoeveel segment pairs met een bepaalde score, zulk een woordpaar bevatten. In [AGML90] hadden de auteurs geen exacte formule om deze kans q dat een segment pair geen woord met score ten minste T bevat, uit te drukken. Ze vermoedden wel dat hoe langer een MSP, hoe meer onafhankelijke kansen deze heeft om een woord met score ten minste T te bevatten. Dit impliceert dat q exponentieel daalt als de MSP score S stijgt.

6.1.3 Gapped BLAST and PSI-BLAST

Uit de oorspronkelijke BLAST zijn verschillende programma's gegroeid, specifiek voor proteïne (BLASTP) of DNA sequenties (BLASTN) of versies met afwijkende heuristieken. In dit hoofdstuk bespreken we Gapped BLAST, ook wel bekend als BLAST 2, en PSI-BLAST.

Indien men bij een alignering gaps toelaat is het duidelijk dat er meer Maximal Segment Pairs gaan gevonden worden. Hoeveel meer hangt af van de kost om een gap te maken en dat gap te extenden, ten opzichte van de waarden in de score matrix. Omdat gaps bij alignments zeer nuttig zijn om homologieën tussen sequenties op te sporen schreven Altschul et Al. in 1997 een vervolgpaper over BLAST waarin ze twee nieuwe versies van BLAST voorstelden: Gapped BLAST en PSI-BLAST [AMS+97]. PSI-BLAST is een geïtereerde versie van BLAST, om zwakke homologieën beter te localiseren. Een aligment tussen twee deelsequenties is langer dan één enkel woordpaar, het is logisch dat er binnen een locale alignment meerdere woordparen ge-

vonden kunnen worden die op dezelfde diagonaal liggen. Met de diagonaal tussen twee hits wordt bedoeld dat de twee woorden die beginnen op positie x_1 en x_2 in de query op dezelfde positie + offset beginnen in de database sequentie. Als we nu de twee sequenties gaan uitlijnen op een matrix zien we twee opeenvolgende hits op dezelfde diagonaal. Dankzij deze observatie gaat men een hit pas uitbreiden nadat er eerder binnen een afstand A een andere niet overlappende hit gevonden is. Men noemt dit de two-hit methode.
Doordat de threshold parameter T, die gebruikt wordt om te bepalen wanneer er een woordpaar gevonden is, wordt verlaagd, zullen er in de eerste fase meer hits zijn. Veel hits moeten echter niet worden uitgebreid omdat er dikwijls geen tweede hit kan worden gevonden. Het uitbreiden van hits neemt meer dan 90% van de executietijd van BLAST in beslag dus het aantal uitgevoerde extensies verlagen kan een belangrijke snelheidswinst opleveren.
Naast substituties kunnen ook insertions of deletions voorkomen in een sequentie. Als men bij een locale alignment hiermee rekening houdt noemt men dit een gapped alignment. Het probleem met de gewone BLAST is dat aaneengrenzende regionen van twee sequenties worden gevonden als aparte MSP's, terwijl biologisch gezien dit één aaneengesloten alignment zou moeten zijn. Een gapped alignment is jammer genoeg een vrij tijdsintensieve berekening dus gaat men pas nadat men, via de two-hit methode, een MSP van ten minste S_g bits heeft gevonden, een gapped extention uitvoeren.

Het BLAST algoritme kan geïtereerd worden waarbij een score matrix met specifieke informatie over de posities van alignments, gevonden in vorige iteraties, gebruikt wordt. Deze methode wordt motif of profile search genoemd, deze methode is meestal veel gevoeliger dan paarwijze vergelijkingen en kunnen dus meer homologiën vinden die al verder uit elkaar zijn geëvolueerd. Deze BLAST versie wordt PSI-BLAST genoemd, of voluit Position-Specific Iterated BLAST, en wordt gebruikt bij proteïne database searches. Voor elke alignering tussen aminozuren en een profiel positie wordt een score gedefinieerd, goed bewaarde sequenties (sequenties waarin weinig substituties zijn gebeurd) hebben een hoge positieve of negatieve score, terwijl sequenties die niks met elkaar te maken hebben een score dicht bij nul krijgen. De snelheid van PSI-BLAST is per iteratie hetzelfde als gapped BLAST, maar is zoals gezegd veel gevoeliger om sequenties te ontdekken die biologisch gezien vrij ver uit elkaar zijn geëvolueerd.
In sommige gevallen kan men niet veel iteraties uitvoeren met PSI-BLAST omdat een klein percentage van de queries in de lijst van matches terecht komen terwijl ze toch geen homologiën zijn. Het profiel wordt in de volgende iteraties dus niet verbeterd, wel integendeel. In [SMS$^+$01] hebben de onderzoekers hier een oplossing voor gevonden in de vorm van compositiegebaseerde statistiek. Hierbij wordt de alignment getuned volgens een specifiek profiel.

6.2 QUASAR

In [BCF+99] wordt een nieuw database search algoritme, QUASAR genoemd, gepresenteerd. QUASAR kan zelf geen alignments maken, het spoort subsequenties op in de database die fel overeenkomen met een query sequentie. In plaats van de database lineair te scannen worden deze subsequenties opgezocht in een een suffix array. Daarna wordt BLAST gebruikt om de aangeduide subsequenties verder te aligneren.

6.2.1 Het algoritme

QUASAR is gebaseerd op volgende observatie: als twee sequenties een edit-distance onder een bepaalde grens hebben, kan men garanderen dat ze een aantal korte sequenties van lengte q (*q-grams* genoemd) gemeenschappelijk hebben [BCF+99]. De database wordt opgedeeld in blokken van lengte b, terwijl de query sequentie wordt opgedeeld in blokken van lengte q. Vervolgens wordt er gekeken hoeveel q-grams van de query-sequentie in één blok voorkomen. Van zodra dit aantal boven een bepaalde threshold komt is de kans groot dat dit deeltje van de database zeer fel lijkt op de query-sequentie, en wordt dit verder gecheckt met een alignment-algoritme. Om het zoekproces te versnellen wordt een array van tellers bijgehouden per blok, en een suffix array wordt gebruikt om snel te kunnen localiseren waar de gezochte q-grams zich bevinden in de database.
Een sequentie $d \in D$ is lokaal gelijk aan S, als er ten minste één paar $(\mathrm{S}[i \ldots i+w-1],\ d')$ substringen bestaan met de volgende eigenschappen:

- $\mathrm{S}[i \ldots i+w-1]$ is een substring van S met lengte w en d' is een substring van d.
- De substringen d' en $\mathrm{S}[i \ldots i+w-1]$ hebben een edit distance van ten hoogste k.

Als $\mathrm{S}[1 \ldots w]$ met tenminste k verschillen eindigt op positie j in D weten we dat ten minste $t = w + 1 - (k+1)q$ van de q-grams in S voorkomen in de substring $\mathrm{D}[j-w+1 \ldots j]$.

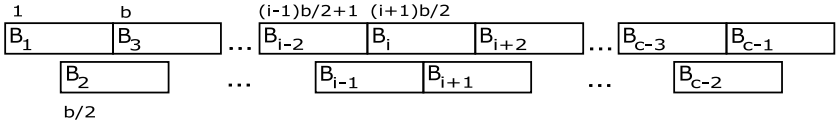
6.2.2 Suffix array als index

Voor elke q-gram Q in S moet de lijst occurrences (een hitlist) in D efficiënt worden opgesteld. Eerst wordt een suffix array SA met lengte $|D|$ opgesteld, waarin alle suffixen van D lexicografisch geordend zijn. Vervolgens worden de posities van de hitlists van alle q-grams opgezocht in de suffix array en in een hulparray van grootte $|\Sigma|^q$ geplaatst. Voor een gegeven q-gram kan dan de startpositie van de hitlist in constante tijd worden opgevraagd. Indien q van grootte zou veranderen moet de suffix array SA niet opnieuw

worden opgebouwd, door een lineaire scan van SA kan de hulparray berekend worden.

6.2.3 Opdelen in blocks

Om alle approximate matches te vinden tussen S en D moeten alle substringen in D die ten minste t q-grams delen met S geïdentificeerd worden. Een simpele benadering zou zijn om een teller bij te houden voor elke substring van lengte w in D, en dan alle tellers te verhogen van blocks die een bepaalde q-gram van S bevatten. Dit kost echter zeer veel geheugen ($|D| - w + 1$ tellers), daarom worden er verschillende aangrenzende substrings met lengte w samen in één block genomen waaraan slechts één teller wordt toegewezen. Hierdoor vermindert uiteraard het gebruikte geheugen, maar stijgt het aantal *false positives* doordat elke teller die boven threshold t ligt moet worden onderzocht.
Omdat het zou kunnen dat q-grams van S over twee naburige blocks verspreid worden, waardoor de twee tellers allebei onder t blijven, wordt er een tweede block decompositie uitgevoerd die over de lengte van een halve block naar rechts is geschoven. Het opdelen van de database D in blocks van lengte b ziet er dan uit zoals in figuur 6.1.



Figuur 6.1: Opdeling van de database in overlappende blocks van grootte b

6.2.4 Alignment

Tot nog toe hebben we enkel de approximate matches voor een window van grootte w gezocht, om de matches van een volgend window $S[2 \ldots w+1]$ te vinden moeten we slechts twee q-grams beschouwen. De q-gram $S[1 \ldots q]$ mag niet meer meetellen, maar de nieuwe q-gram $S[w+2-q \ldots w+1]$ wel. Daarom wordt de teller van alle blocks die $S[1 \ldots q]$ bevatten verlaagd en de blocks die q-gram $S[w+2-q \ldots w+1]$ bevatten verhoogd.
Indien de teller van een block de threshold al bereikt heeft, wordt dat block buiten beschouwing gelaten. Per verschuiving van het window moeten dus slechts twee q-grams in rekening worden gebracht.
Nadat de lijst met blocks die potentiële hits bevatten is opgesteld, wordt BLAST gebruikt om deze blocks te scannen en matching regions te aligneren.

6.2.5 Resultaten

Door de makers van de paper is QUASAR onderzocht met volgende settings: een window lengte van w = 50, q-grams met lengte 11 en een threshold t die garandeert alle windows te vinden met een edit distance van maximum 3. Ze vergeleken QUASAR-BLAST, waarbij BLAST dus enkel de database-secties kreeg die aangewezen waren door QUASAR, met BLAST, die de hele sequentie scant. De BLAST-versie waarmee de onderzoekers werken was NCBI BLAST 2.0.3.
Bij het vergelijken van de sensitivity zou men in het ideale geval E-values vergelijken. QUASAR-BLAST en BLAST kunnen jammer genoeg niet hierop vergeleken worden omdat hun database-grootte verschilt. Men bekijkt daarom per E-value hoeveel alignments gevonden worden door BLAST maar niet door QUASAR-BLAST. Gegeven bovenstaande settings kwam men uit op een E-value van 10^{-5}, wat klein genoeg is om ook homologieën te vinden tussen genomen die al ver uit elkaar zijn geëvolueerd.
Qua performance bleek QUASAR 30 keer sneller te werken dan BLAST, ook al werd er verwacht dat repeats de werking van het algoritme fel zou kunnen vertragen. Proeven op muis-genoom heeft echter aangetoond dat dit slechts weinig effect heeft op de snelheid. Een muis-genoom heeft veel meer repeats en low-complexity regions dan bijvoorbeeld een bacterieel genoom. Herhalingen kunnen gemakkelijk dit soort algoritme vertragen omdat er voor bepaalde q-grams een enorme hit-lijst bestaat waardoor er veel meer false positives onderzocht worden.
Een nadeel van het algoritme is de tijd die nodig is om de suffix array te berekenen, samen met de plaats die deze inneemt in het geheugen of op de harde schijf. Indien er slechts zeer weinig queries worden uitgevoerd op een bepaald genoom kan het nuttiger zijn om het telkens lineair te scannen in plaats van een grote index bij te houden die weinig wordt gebruikt.

6.3 De suffix sequoia

Approximate string matching wordt door twee types algortimen aangepakt: exhaustieve algoritmes zoals Smith-Waterman wat we hebben besproken in hoofdstuk 5.1.1 en heuristieken waaronder BLAST, zie hoofdstuk 6.1. De suffix sequoia [Hun03] is bedoeld voor full-sensitivity searching, zoals het Smith-Waterman algoritme, maar dan via een sequentie-database.
De sequentie wordt geïndexeerd via een aangepaste datastructuur die alle suffixen sorteert op de d eerste karakters. Voor elke window van de query sequentie wordt een similarity matrix opgesteld met alle mogelijke subsequenties van lengte d. De posities van de subsequenties die een minimale score behalen worden vervolgens uit de index opgevraagd en gealigneerd met de query-sequentie. Door gebruik te maken van de index is een volledige scan van de database-sequentie niet nodig.

6.3.1 De datastructuur

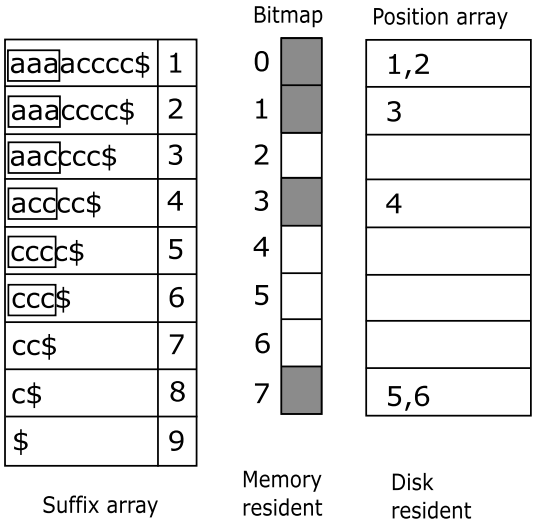
De suffix sequoia is een afgeleide van de suffix tree, waarbij de tree afgekort wordt tot een bepaalde string diepte d. Deze structuur bestaat uit een bitmap en een array van posities. De bitmap geeft weer of een gegeven korte substring al dan niet aanwezig is in de tekst. Een cel in de array van posities (*PositionArray*) duidt aan waar de suffixen, die een gemeenschappelijke prefix van lengte d hebben, zich bevinden in de sequentie.
Om de index te creëren wordt een window van size d over de sequentie geschoven. Dus de sequentie S met lengte n heeft $n - d + 1$ vensters van de vorm $win_j = s_j ... s_{j+d}$. De windows worden lexicographisch gesorteerd door de *PositionArray* te updaten. Deze heeft een grootte van g^d, één voor elke window code:

$$code(win_j) = \sum_{k=0}^{d-1} s_{k+j} g^{d-k-1}$$

In de bitmap wordt dan de overeenkomstige window-code op true gezet:

$$bitMap[code(win_j)] = true$$

De *PositionArray* bevat dan op elke locatie $PositionArray[code(win_j)]$ een lijst van alle posities van vensters met die tekst.

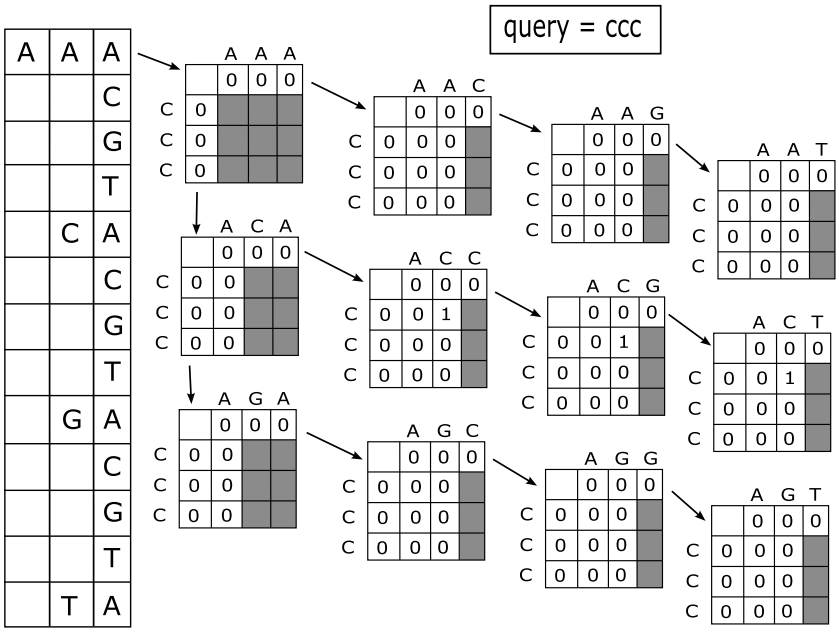


Figuur 6.2: De suffix sequoia

6.3.2 Het algoritme

Het algoritme werkt in drie stappen. Eerst wordt de query gescand met een window dat dezelfde grootte heeft als de index-diepte. Elk window wordt

gealigneerd met alle subsequenties in de index, indien de score boven de threshold komt wordt de overeenkomstige window-code bijgehouden. Per window van de query kunnen dus meerdere verwijzingen naar window-codes worden bewaard. Het tweede deel van het algoritme kijkt voor elke window-code na of de bitmap op true is gezet voor die bepaalde code. Indien ja worden de overeenkomstige posities in de sequentie opgevraagd. In de derde stap wordt voor elk van de bekomen posities de alignment opgesteld.
Het algoritme zoekt dus niet alleen exacte matches. Gegeven een stuk van de query waarvan de windowgrootte even groot is als de diepte van de index, wordt de volledige sequentie gescand en wordt van elke substring in de index uitgelijnd met het stukje query. In deze stap wordt niet eens gekeken of de substring effectief voorkomt in de sequentie.
Omdat de opeenvolgende subsequenties in de index lexicografisch zijn gerangschikt moeten de opeenvolgende matrices niet from scratch worden opgebouwd. Telkens als er een letter verandert tussen twee subsequenties moeten de waarden in de similarity matrix van dat punt opnieuw worden berekend. Dus als enkel het laatste karakter verandert moet alleen de laatste kolom worden herberekend. Dit wordt verduidelijkt in figuur 6.3, de grijze gebieden zijn waardes die berekend moeten worden.



Figuur 6.3: Berekenen van de scores

We zijn enkel geïnteresseerd in similarity matrices waarin een vaste minimale score wordt bereikt, van deze uitlijningen worden de overeenkomstige window-codes bijgehouden. Het is echter mogelijk dat bij een lage minimale score er teveel window-codes worden gereturned, waardoor een te groot deel

van de database moet worden gealigneerd. Als we bijvoorbeeld als minimale score 3 hebben en we zoeken matches voor het query window GTCAG. Stel dat het matchen van een G met een G +3 geeft, dan moeten alle window-codes die beginnen met een G worden gereturned. Om zulke problemen op te lossen wordt een parameter *maxCodes* ingesteld, deze duidt aan hoeveel window-codes er maximaal worden opgeroepen voor elk query window.
Algemeen genomen is er, voor een alfabet Σ van grootte g en window size (index diepte) d, een maximum aantal berekeningen mogelijk om de similarity matrices op te bouwen:

$$g + g^2 + ... + g^d = \frac{g(g^d - 1)}{g - 1}$$

Zoals bij QUASAR, wat we in hoofdstuk 6.2 hebben besproken, gebeurt de eigenlijke aligning door een ander programma. Nadat een set posities is berekend wordt BLAT [Ken02] gebruikt om de eigenlijke aligneringen te maken.
De tijd die nodig is om de index aan te maken komt overeen met de lengte van de te indexeren sequentie, namelijk $O(n)$. De space complexiteit, $O(n + g^d)$, hangt af van windowgrootte, de grootte van het alfabet en uiteraard de lengte van de sequentie. De grootste plaats wordt ingenomen door de *PositionArray*, die lijsten van integers bevat.

6.3.3 Resultaten

Het programma kan garanderen dat geen enkele substring die op een vastgelegde edit distance ligt van een deel van de query, onopgemerkt voorbij gaat. Dit in tegenstelling tot BLAST, dat een heuristiek gebruikt. Afhankelijk van de versie kan BLAST enkel substringen vinden die ten minste een bepaald aantal opeenvolgende karakters gemeenschappelijk hebben, deze sequenties worden gezocht door met een window over de query sequentie te schuiven en de stukjes query te vergelijken met de database-sequentie. Bij proteïne wordt een window met grootte 3 gebruikt, per window worden een aantal gelijkaardige subsequenties gegenereerd binnen een bepaalde edit-distance waarna er naar matches wordt gezocht. Het aantal gelijkaardige subsequenties wordt echter beperkt.
Over de suffix sequoia wordt een full-sensitivity search uitgevoerd, vergelijkbaar met het Smith-Waterman algoritme. Men heeft een aantal queries ingevoerd in BLAST en de suffix sequoia, en kwam op dezelfde grootorde van snelheden uit. Veel hangt uiteraard af van de instellingen van de parameters *maxCodes* en de minimale score.
Uit de resultaten valt moeilijk af te leiden hoe dikwijls de *maxCodes*-waarde gehaald werd bij een query, maar het valt wel op dat een te lage waarde een felle vermindering geeft in het aantal matches.

Om het algoritme te versnellen wordt nagekeken of in de sequentie bepaalde repeats frequent voorkomen. Deze repeats hebben als effect dat bepaalde window-codes enorm veel gaan voorkomen, wat veel vertraging geeft bij het alignen van al deze posities. Veel voorkomende windows kunnen daarom gemaskeerd worden met een aanduiding in de bitmap. Hierdoor kan echter niet meer gegarandeerd worden dat alle alignments met een score die hoger is dan de minimale score, gevonden worden.
Daarnaast kan het opbouwen van de alignments per window versneld worden door toch te kijken of de overeenkomstige code voorkomt in de bitmap. Indien niet kan het interessant zijn om een aantal alignments die toch niet voorkomen over te slaan. Zoals al eerder aangetoond kunnen deze alignments in de volgende stappen dikwijls worden herbruikt, maar als het eerste karakter verandert moet de hele alignment toch opnieuw gebeuren. Indien vanaf de huidige alignment tot aan de code waarin het eerste karakter verandert, er geen overeenkomstige substrings in de sequentie voorkomen kan men de alignment van deze codes overslaan.

6.4 OASIS

OASIS, voluit *An Online and Accurate Technique for Local-alignment Searches on Biological Sequences* [MPK03], is bedoeld om accuraat en efficiënt local-alignment searches te evalueren. Het programma gebruikt een best-first A* algoritme, om zo de beste locale alignments als eerste te vinden. Daarom wordt het programma online genoemd, het begint output te genereren ook al zijn nog niet alle matches gevonden. Dit in contrast met BLAST en Smith-Waterman, zij moeten eerst de volledige query en database-sequentie hebben geëvalueerd alvorens ze output genereren.
Voordat OASIS een query kan berekenen moet eerst de suffix tree van de sequentie database worden gemaakt. Op basis van die suffix tree gaan er bepaalde stukken sequentie gealigneerd worden via het Smith-Waterman algoritme. Naast de suffix tree en de query moeten ook nog een score functie en een minimum alignment score als input worden gegeven. OASIS genereert nu search-nodes, gedeeltelijke alignments tussen de query en delen van de sequentie database. Elke search-node komt precies overeen met een node uit de suffix tree, de gedeeltelijke alignment komt overeen met het path van de root naar die node uit de suffix tree (path). De search-nodes worden gesorteerd in een priority queue (PQ), hoe hoger de maximale score die mogelijk is bij het aligneren van de rest van de query hoe hoger in de PQ. Bij elke stap van het algoritme wordt de bovenste search-node genomen, waarna de kinderen van de overeenkomstige suffix tree node verder worden onderzocht.
De belangrijkste velden van een search-node zijn de volgende:

- de pointer *sp* naar een node in de suffix tree

- een vector $Z = [z_0\ z_1\ ...\ z_m]$, met z_i de score van de sterkste alignment tussen de sequenties path(sp) en elke subsequentie van Q die eindigt in q_i. Deze vector komt ruwweg overeen met een kolom van de Smith-Waterman matrix.
- de maximum score gevonden op path(sp)
- de hoogst mogelijke score f die gevonden kan worden als deze node verder wordt onderzocht
- een tag die aanduidt of de sterkste alignment gevonden is en boven de threshold ligt, of dat er een sterkere alignment mogelijk is dan die tot nu toe gevonden is en boven de threshold kan liggen, of dat er geen enkele alignment kan gevonden worden hoger dan de threshold

De expand functie is de belangrijkste functie van OASIS, de beste search-node wordt uitgebreid met de kinderen van zijn overeenkomstige suffix tree node. Voor elk van die kinderen moet een alignment berekend worden. Omdat Smith-Waterman inductief werkt is slechts de laatste kolom van de vorige alignment nodig, de Z-vector van de search-node. Van zodra de alignment is uitgevoerd kunnen bepaalde search-nodes worden gepruned, ofwel omdat de maximum haalbare score onder de threshold blijft, of omdat de alignment samenvalt met een hoger scorende alignment.
Het algoritme gebruik een suffix tree die bewaard wordt op de harde schijf. Omdat sequenties opzoeken in een suffix tree veel random disk accesses tot gevolg heeft wordt de suffix tree zo opgeslagen dat siblings zo veel mogelijk aaneengesloten worden bewaard. Dit is voordelig omdat OASIS alle kinderen van een node tegelijkertijd gaat uitwerken. Een suffix tree wordt voorgesteld door de sequentie zelf, een array voor de interne nodes en één voor de leaf nodes. De sequentie en de interne nodes worden opgedeeld in stukken die precies passen in één disk block, deze blokken worden sequentieel op de harde schijf geschreven. Interne nodes worden opgeslagen per niveau. Om plaats te sparen worden de leaf nodes zo georganiseerd dat hun positie in de array verwijst naar de overeenkomstige positie in de sequentie. Elke leaf node komt immers overeen met één bepaalde suffix.

6.5 Mummer

Mummer is een systeem om volledige genomen uit te lijnen. Waar de meeste systemen zoals BLAST op zoek gaan naar invoegingen, verwijderingen of substituties gaat dit systeem er van uit dat de sequenties niet veel van elkaar verschillen. Mummer aligneert twee genomen die vrij homoloog zijn, genomen die grote stukken subsequenties nog gemeenschappelijk hebben.
Mummer gaat op zoek naar de verschillen tussen genomen, zoals (tandem) repeats, reversals, grote invoegingen en single nucleotide polymorphisms

(SNP's). Deze laatste term duidt op sequenties die in beide genomen voorkomen en slechts in één nucleotide van elkaar verschillen. Repeats zijn subsequenties in een DNA-sequentie die een aantal keer worden herhaald, indien de subsequenties vlak na elkaar worden herhaald spreken we over een tandem repeat. Tandem repeats en grote reversals of insertions worden niet ontdekt indien gen per gen wordt gealigneerd, enkel bij een volledige alignment van de genomen komen deze fenomenen aan de oppervlakte. In sectie 6.5.3 gaan we dieper in op deze begrippen.
De eerste versie van Mummer is geschreven in 1999 [DKF$^+$99], de output van het systeem bestaat uit een alignment van de input sequenties, waarin de exacte verschillen tussen de genomen worden benadrukt. Het systeem bestaat uit de combinatie van drie ideeën: MUM's zoeken via suffix trees, de longest increasing subsequence (LIS) en Smith-Waterman alignering. De basis van het algoritme is een suffix tree, deze datastructuur laat toe op op een snelle manier alle subsequenties te vinden die in beide genomen voorkomen.

6.5.1 MUM's zoeken via suffix trees

De alignment wordt in verschillende stappen uitgevoerd, allereerst worden MUM's gezocht. Een MUM, wat maximale unieke match betekent, is een subsequentie die precies éénmaal in elk van de twee genomen voorkomt en maximaal is. De theorie is dat indien twee gerelateerde sequenties dezelfde lange subsequentie bevatten, deze subsequentie zo goed als zeker deel uit maakt van de globale alignment. Dus rond deze MUM-alignments wordt de global alignments opgebouwd.
Het probleem van het vinden van een set maximaal uniek matchende substrings in twee lange sequenties is niet triviaal. Om dit probleem op te lossen wordt in Mummer een suffix tree gebruikt. Eerst wordt de suffix tree van het eerste genoom gemaakt, waaraan de suffixen van het tweede genoom worden toegevoegd. Elke leaf node in de suffix tree krijgt een label dat aanduidt tot welk genoom de overeenkomstige suffix behoort. Het is gemakkelijk in te zien dat elke uniek matchende sequentie voorgesteld wordt door een interne node met precies twee leaf-nodes die elk een ander label hebben. Uit deze matchende sequenties moeten uiteraard de maximale matchende sequenties worden geselecteerd. Men kan eenvoudig vaststellen of twee matchende sequenties maximaal zijn door de twee karakters die vlak voor de sequenties liggen te vergelijken. Indien ze gelijk zijn is het geen maximaal matchende sequentie. Een belangrijke parameter bij dit proces is de lengte van de kortste MUM die door het systeem wordt aanvaardt. Deze waarde mag uiteraard niet te klein zijn, om random matches te vermijden.

6.5.2 Longest Increasing Subsequence

Tijdens de evolutie van genomen kunnen transposities en reversals van deelsequenties plaatsvinden zodat de MUM's niet meer in dezelfde volgorde gaan gevonden worden. De MUM's worden gesorteerd en de langste set van matches die in de zelfde volgorde in beide genomen voorkomt wordt bewaard. Deze set wordt gevonden dankzij het LIS-algoritme [Gus97]. Vanuit deze set wordt een geordende MUM-alignment gemaakt, zodat de alignment van links naar rechts gescand kan worden.

6.5.3 Sluiten van de gaps

De laatste stap is het sluiten van de gaps tussen de MUM-alignments door lokale inserts, repeats, gemuteerde stukken, tandem repeats en SNP's te indentificeren. Mummer beschouwt vier verschillende klassen interruptions: SNP's, insertions, polymorphic regions en repeats.
Single nucleotide polymorphisms (SNP's) zijn in de meeste gevallen gemakkelijk te vinden, meestal zijn ze ingesloten tussen twee maximaal uniek matchende subsequenties. Soms zit een SNP echter vlak naast een repeat, dit geval moet worden ontdekt door de procedure om repeats te evalueren.
Er zijn twee verschillende soorten insertions, namelijk transposities en gewone insertions. Transposities zijn subsequenties die van plaats zijn veranderd, zij worden door de MUM-alignment apart behandeld omdat ze niet meer in de globale alignment passen. Gewone insertions zijn subsequenties die slechts in één van de genomen voorkomen, zij worden niet beschouwd in de globale alignment.
Gaps in de MUM-alignment kunnen ook veroorzaakt zijn door sequenties die een groot aantal verschillen vertonen, maar toch nog gealigneerd kunnen worden in de globale alignment. Deze polymorphic regions kunnen op twee verschillende manieren worden gealingneerd, indien de subsequentie vrij kort is kan Smith-Waterman worden toegepast. Voor vrij lange polymorphic regions kan de matching procedure recursief worden toegepast, eventueel met een kleinere minimum MUM lengte.
Repeat sequenties tellen niet mee in de globale MUM-alignment omdat deze enkel de sequenties omvat die precies éénmaal voorkomen in elk genoom.

6.5.4 Mummer 2.0

Ook in de tweede versie van Mummer [DPCS02] worden suffix trees gebruikt, maar in tegenstelling tot de eerste versie worden enkel de suffixen van één genoom door een suffix tree geïndexeerd. Het tweede genoom wordt langs de suffix tree gestreamd, er wordt gesimuleerd dat de suffixen van het tweede genoom worden toegevoegd zonder ze echt toe te voegen.
Omdat er de helft minder suffixen in de suffix tree worden toegevoegd, wordt er uiteraard minder geheugen gebruikt en gaat het algoritme sneller werken.

Om een volgende match te vinden worden suffix links gebruikt, vanuit een node waar een suffix $i + 1$ niet verder gematcht kan worden volgen we de suffix link om vanuit die node de volgende suffix i te matchen. Als men vanuit een node waarvan het path overeen komt met de prefix van suffix $i + 1$ de suffix link volgt komt men in een node die als path de prefix van suffix i heeft. Dit betekent dus dat de karakters van de prefix niet meer moeten vergeleken worden met de labels van de suffix tree.
Daarnaast wordt de suffix tree op een andere manier opgeslagen door de methode van Kurtz [AKO04] te gebruiken. Deze methode is reeds aan bod gekomen in hoofdstuk 2.6.2. De hoeveelheid geheugen wordt hierdoor beperkt tot gemiddeld 20 bytes per base-pair.
De tweede versie van Mummer gaat niet meer proberen één globale alignment te maken, er worden verschillende matches geclusterd waartussen een consistent path wordt gezocht. De oorspronkelijke versie van Mummer ging er van uit dat genomen niet al te fel van elkaar verschillen. Bij soorten die een verre verwantschap vertonen kunnen echter grote stukken van het genoom van plaats worden verwisseld, zodat het niet veel zin heeft om een langste gemeenschappelijk path van MUM's te zoeken. Het clustering gebeurt door matchende paren te zoeken die voldoende dicht bij elkaar liggen en op ongeveer dezelfde diagonaal liggen in een alignment matrix. Per cluster wordt dan een alignment uitgevoerd.

6.5.5 Mummer 3.0

De vorige twee versies van Mummer steunden op maximaal unieke matches (MUM's) om de sequenties te aligneren. MUM's komen in de twee genomen exact één maal voor. In sommige gevallen zal Mummer echter niet alle matches voor een repetitieve subsequentie vinden. Indien bijvoorbeeld één van de twee genomen een extra kopie heeft van een subsequentie is het mogelijk dat het tweede voorkomen van de subsequentie niet wordt opgemerkt. Om dit probleem op te lossen gaat Mummer 3.0 alle maximale matches, waaronder de niet-unieke, opsporen.
Approximate matches worden efficiënter opgespoord in Mummer 3.0 door gebruik te maken van twee packages: Nucmer en Promer. Deze beide packages kunnen een serie local alignments maken. Het verschil tussen de twee programma's is dat Nucmer nucleotide-alignments maakt tussen twee sets DNA sequenties, terwijl Promer aminozuur-alignments maakt. Promer vertaalt eerst beide sequenties in zes frames, zoekt matches in de aminozuur sequenties en mapt die matches terug naar het DNA-alfabet. Op die manier is er een grotere kans dat men de meest waarschijnlijke alignment vindt.

6.5.6 Andere methoden

Er bestaan nog verschillende andere methoden om genomen in hun geheel met elkaar te aligneren, waaronder SSAHA, AVID, MGA, BLASTZ en LAGAN. De meeste van deze programma's werken met ankers, maximale matches met een lengte langer dan l. Eerst worden alle matches met een vaste lengte $k \leq l$ gegenereerd via hashing. Deze matches noemt men k-mers. Elk van deze k-mers wordt verlengd om te zien of er een maximale exacte match is met een lengte van ten minste l. Het verlengen van de matches gebeurt base per base, dus de tijd die nodig is voor de vergelijkingen hangt niet enkel af van het aantal ankers, maar ook van hun lengte. De grote meerderheid van deze k-mers worden verlengd tot maximale matches met minimum lengte l. Mummer gebruikt echter suffix trees in plaats van hashing technieken, zoals we in sectie 6.5.1 al hebben beschreven. Door suffix trees en suffix links verstandig te gebruiken hangt de runtime niet af van de lengte van de maximale matches.

6.6 PossumSearch

In een suffix tree kan men alle subsequenties van S met een vaste lengte m een score geven via een position specific scoring matrix (PSSM) door de suffix tree in depth-first orde te doorlopen. We hebben PSSM's reeds beschreven in hoofdstuk 5.1.2. We herhalen nog even dat M een PSSM is met lengte m, $sc(w, M)$ de match score is van w ten opzichte van M en dat $pfxsc_d(w, M)$ de prefix score is met diepte d. Door lookahead scoring kunnen bepaalde subtrees overgeslagen worden omdat de prefix van die subsequenties niet boven de intermediaire threshold komt. Indien men in een node zit waar de score van het path onder de intermediaire threshold zit, moet men de rest van die subtree niet meer bekijken en kan men het path van de volgende sibling gaan bekijken. Indien een node geen sibling heeft moet men de sibling evalueren van de eerste ouder met sibling.
In hoofdstuk 4 hebben we uitgelegd hoe de enhanced suffix array werkt, in [Bec04] wordt aangetoond dat deze ook kan gebruikt worden om PSSM-scores te bepalen. De enhanced suffix array bestaat uit een aantal tabellen, waaronder de eigenlijke suffix array *SA* en de *lcp*-tabel. Om lookahead scoring toe te laten moeten er soms stukken van de suffix array worden overgeslagen, dit wordt via de *skp*-tabel gesimuleerd.
De tabel *skp* heeft een grootte van n,

$$skp[i] = min(\{n+1\} \cup \{j \in [i+1, n] \,|\, lcp[i] > lcp[j]\}).$$

Met andere woorden, $skp[i]$ duidt op het lexicografische volgende blad dat niet voorkomt in de subtree onder de huidige branching node. De branching node heeft als path de langste gemeenschappelijke prefix van $S_{SA[i-1]}$ en

$S_{SA[i]}$. Deze array kan bij de enhanced suffix tree worden gevoegd door ze in $O(n)$ tijd te berekenen uit de suffix array SA en lcp-tabel.

We geven nu het ESA search-algoritme om PSSM scores te bepalen aan de hand van een enhanced suffix array. Het algoritme probeert dubbele berekeningen zoveel mogelijk te vermijden door te steunen op de lcp-tabel. Intermediaire waarden $pfxsc_d(w, M)$ gelden immers voor alle substrings w die dezelfde prefix hebben met lengte d. Naast intermediare waarden herbruiken gaat het algoritme bij een mismatch ook op zoek naar de volgende suffix die wel een PSSM-score kan hebben die boven de threshold uitkomt. Voor $i \in [0, n]$ noem de suffix $v_i = S_{SA[i]}$, $l_i = min\{m, |v_i|\} - 1$ en

$$d_i = max(\{-1\} \cup \{d \in [0, l_i] \mid pfxsc_d(v_i, M) \geq th_d\}).$$

We weten dat $d_i = m - 1 \Leftrightarrow pfxsc_{m-1}(v_i, M) \geq th_{m-1} \Leftrightarrow sc(v_i, M) \geq th$. Dus enkel indien $d_i = m - 1$ matcht M op positie $j = SA[i]$. Om het PSSM search probleem op te lossen moeten we $d_i = m-1$ bepalen voor alle $i \in [0, n]$.
We berekenen d_i via de array $C_i[d] := pfsxc_d(v_i, M)$ voor elke $d \in [0, d_i]$. We beginnen met d_0 en C_0 te berekenen, wat in $O(m)$ tijd kan. We gaan er van uit dat we d_{i-1} en $C_{i-1}[d]$ hebben berekend voor $d \in [0, d_{i-1}]$. Omdat v_{i-1} en v_i een gemeenschappelijke prefix hebben van lengte $lcp[i]$, kunnen we hieruit afleiden dat $C_i[d] = C_{i-1}[d]$ voor elke $d \in [0, lcp[i] - 1]$. We onderscheiden tijdens de constructie twee gevallen:

- Indien $d_{i-1} + 1 \geq lcp[i]$ moeten we $C_i[d]$ berekenen voor $d \geq lcp[i]$ terwijl $d \leq l_i$ en $C_i[d] \geq th_d$. Dus $d_i = d$.

- Indien $d_{i-1}+1 < lcp[i]$ laat j het minimum zijn van de range $[i+1, n+1]$ zodat alle suffixen $v_i, v_{i+1}, \ldots, v_{j-1}$ een gemeenschappelijke prefix hebben met v_{i-1} die lengte $d_{i-1} + 1$ heeft. Hieruit concluderen we dus dat $pfxsc_d(v_{i-1}, M) = pfxsc_d(v_r, M)$ voor alle $d \in [0, d_{i-1} + 1]$ en $r \in [i, j-1]$. Dus $d_{i-1} = d_r$. Indien $d_{i-1} = m-1$ zijn er PSSM matches op alle posities $SA[r]$, met $r \in [i, j-1]$. Indien echter $d_{i-1} < m - 1$ betekent dit dat er geen PSSM matches zijn op deze posities. Dus kunnen we doorgaan met positie j, die we verkrijgen door een keten van waarden uit de tabel skp te volgen, $j_0 = i$, $j_1 = skp[j_0], \ldots, j_k = skp[j_{k-1}]$ zodat $d_{i-1}+1 < lcp[j_1], \ldots, d_{i-1}+1 < lcp[j_{k-1}]$ en $d_{i-1}+1 \geq lcp[j_k]$. Dus $j = j_k$.

6.6.1 Resultaten

In de experimenten wordt het ESA search algoritme vergeleken met de methode van [DNM00]. Deze methode maakt gebruik van een suffix tree waardoor er 17n bytes geheugen gebruikt worden. Het ESA search algoritme

werkt slechts met een aantal van de tabellen van de enhanced suffix array, namelijk de suffix array, de Burrows-Wheeler transformation array (bwttab) en de tabel met langste gemeenschappelijke prefixen (lcptab), zodat er slechts 9n bytes nodig zijn.
De snelheidswinst die geboekt wordt, een factor 1.5 tot 1.8 volgens de resultaten van [Bec04], schrijft men toe aan het feit dat de enhanced suffix array een beter locality of memory reference heeft dan de suffix tree. Maar ook door het gebruik van de skp tabel kunnen er significante stukken van de suffix array worden overgeslagen, zodat het algoritme in sublineaire tijd over de lengte van de sequentie werkt.

Hoofdstuk 7

Conclusie

We hebben in deze thesis eerst een aantal algoritmes besproken om suffix trees te construeren. Een suffix tree indexeert alle suffixen van een sequentie, zijn belangrijkste eigenschap is het vermogen om bestaansqueries te beantwoorden in lineaire tijd over de lengte van de query. Allereerst hebben we enkele constructiealgoritmes besproken, waaronder het lineaire algoritme van Ukkonen. Omdat de meeste voorstellingen van suffix trees bijzonder veel geheugen innemen heeft hebben we enkele methodes gezien om de grootte van een suffix tree in het geheugen zo klein mogelijk te maken. De beste oplossing die we hebben gevonden is het wotdeager algoritme van Kurtz, dat een suffix tree kan construeren die slechts 12n bytes inneemt.
Vervolgens hebben we twee systemen besproken die in staat zouden moeten zijn om van volledige genomen de suffix tree op te stellen, TDD van Patel et Al. en TOP-Q van Haritsa en Bedathur. TDD is gebaseerd op het wotdeager algoritme van Kurtz, TOP-Q op het lineaire algoritme van Ukkonen. Door het testen van beide systemen zijn we tot de vaststelling gekomen dat TOP-Q ongeschikt is om in extern geheugen de suffix tree te construeren. TDD was in de testen tot vijf keer sneller dan TOP-Q maar kon door een programmeerfout geen sequenties verwerken van meer dan 50 miljoen karakters. Toch kunnen we uit de resultaten concluderen dat het lineaire algoritme trager werkt dan TDD dat een worst-case tijdscomplexiteit van $O(n^2)$ heeft.

Een alternatief voor de suffix tree is de suffix array, deze indexstructuur heeft exact 4n bytes nodig maar heeft als nadeel dat bestaansqueries een factor $\log n$ trager zijn dan bij suffix trees. We hebben eerst enkele algoritmes, zoals doubling en discarding, besproken die een suffix array kunnen construeren. De snelheid van deze algoritmes hangt in grote mate af van het gebruikte sorteeralgoritme. Later hebben we een lineair algoritme besproken, het difference cover 3 algoritme, dat we hebben veralgemeend naar andere difference covers. We hebben een pipelined versie van doubling, discarding en DC3 getest, waarvan de verrassende uitkomst was dat doubling

iets sneller was dan DC3. Discarding zette het slechtste resultaat neer, terwijl het in principe een verbetering is van het doubling algoritme.

Daarna hebben we heel wat aandacht besteed aan de enhanced suffix array, een datastructuur die een brug vormt tussen de suffix array en de suffix tree. Een enhanced suffix array is een indexstructuur gebaseerd op een suffix array aangevuld met een aantal extra tabellen. Hiermee kan men het bottom-up en top-down doorlopen van de suffix tree simuleren alsook het volgen van suffix links. We hebben aangetoond dat door deze eigenschappen elk algoritme dat gebruik maakt van een suffix tree, deze kan vervangen worden door een enhanced suffix array. Uit testen is gebleken dat de grootte van de enhanced suffix array voor de meeste algoritmes kleiner is dan de grootte van de overeenkomstige suffix tree. Qua snelheid in het beantwoorden van bestaansqueries hebben we gunstige resultaten opgetekend, vergeleken met de array-gebaseerde suffix tree.

We hebben enkele begrippen overlopen om sequenties te aligneren, zoals het Smith-Waterman en Needleman-Wunsch algoritme. We hebben eveneens het begrip position specific scoring matrix besproken. Tot slot hebben we enkele systemen overlopen om zowel locale als globale aligneringen te berekenen, die steunen op een suffix tree of suffix array. De systemen voor locale alignments hebben we vergeleken met twee standaard-algoritmes in de bioinformatica: BLAST en het Smith-Waterman algoritme.

Bibliografie

[AFGV97] Lars Arge, Paolo Ferragina, Roberto Grossi, and Jeffrey Scott Vitter. On sorting strings in external memory. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 540–548, New York, NY, USA, 1997. ACM Press.

[AGML90] S. F. Altschul, W. Gish, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, October 1990.

[AKO02] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *2nd Workshop on Algorithms in Bioinformatics*, LNCS, 2002.

[AKO04] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. of Discrete Algorithms*, 2(1):53–86, 2004.

[AMS+97] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Res.*, 25:3389–3402, 1997.

[BCF+99] Stefan Burkhardt, Andreas Crauser, Paolo Ferragina, Hans-Peter Lenhof, Eric Rivals, and Martin Vingron. q-gram based database searching using a suffix array (quasar). In *RECOMB '99: Proceedings of the third annual international conference on Computational molecular biology*, pages 77–83, New York, NY, USA, 1999. ACM Press.

[Bec04] Beckstette, M. and Strothmann, D. and Homann, R. and Giegerich, R. and Kurtz, S. PoSSuMsearch: Fast and Sensitive Matching of Position Specific Scoring Matrices using Enhanced Suffix Arrays. In Giegerich, R. and Stoye, J., editor, *Proc. of the German Conference on Bioinformatics*, GI-Edition, Lecture Notes in Informatics, volume P-53, pages 53–64, 2004.

[BH04] Srikanta J. Bedathur and Jayant R. Haritsa. Engineering a fast online persistent suffix tree construction. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 720, Washington, DC, USA, 2004. IEEE Computer Society.

[Bro04] A. L. Brown. Constructing chromosome scale suffix trees. In *CRPIT '04: Proceedings of the second conference on Asia-Pacific bioinformatics*, pages 105–112, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[BS97] Bentley and Sedgewick. Fast algorithms for sorting and searching strings. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1997.

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, 1994.

[CF99] Andreas Crauser and Paolo Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. Research Report MPI-I-1999-1-001, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, March 1999.

[CL00] Charles J. Colbourn and Alan C. H. Ling. Quorums from difference covers. *Inf. Process. Lett.*, 75(1-2):9–12, 2000.

[DKF+99] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucl. Acids. Res.*, 27(11):2369–2376, 1999.

[DMKS05] R. Dementiev, J. Mehnert, J. Kärkkäinen, and P. Sanders. Better external memory suffix array construction. *Workshop on Algorithm Engineering & Experiments*, 2005.

[DNM00] Bogdan Dorohonceanu and C. G. Nevill-Manning. Accelerating protein classification using suffix trees. In *Proc. of the International Conference on Intelligent Systems for Molecular Biology*, pages 128–133, Menlo Park, CA, 2000. AAAI Press.

[DPCS02] A. L. Delcher, A. Phillippy, J. Carlton, and S. L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research*, 30(11):2478–2483, 2002.

[DS05] R. Dementiev and P. Sanders. The stxxl library. *Documentatie en download op http://i10www.ira.uka.de/dementiev/stxxl.shtml*, 2005.

[GKS99] Robert Giegerich, Stefan Kurtz, and Jens Stoye. Efficient implementation of lazy suffix trees. In *WAE '99: Proceedings of the 3rd International Workshop on Algorithm Engineering*, pages 30–42, London, UK, 1999. Springer-Verlag.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, New York, 1997.

[HAI01] Ela Hunt, Malcolm P. Atkinson, and Robert W. Irving. A database index to large biological sequences. In *The VLDB Journal*, pages 139–148, 2001.

[Hun03] E. Hunt. The suffix sequoia index for approximate string matching. Technical Report TR-2003-135, Department of Computing Science, University of Glasgow, Glasgow, UK, 2003.

[IT99] Hideo Itoh and Hozumi Tanaka. An efficient method for in memory construction of suffix arrays. In *SPIRE '99: Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware*, page 81, Washington, DC, USA, 1999. IEEE Computer Society.

[Jap04] Robert Japp. The top-compressed suffix tree: A disk-resident index for large sequences. Technical Report TR-2004-165, Department of Computing Science, University of Glasgow, July 2004.

[JKB03] Peter Sanders Juha Kärkkäinen and Stefan Burkhardt. Linear work suffix array construction. In *J. ACM.* To appear, 2003.

[JS94] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 439–450. Morgan Kaufmann, 1994.

[KA90] S. Karlin and S. F. Altschul. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proc. Natl. Acad. Sci. USA*, 87:2264–2268, 1990.

[KA03] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. Annual Symposium on Combinatorial Pattern Matching*, volume 2676, pages 200–210, Berlin, 2003. Springer-Verlag.

[Ken02] W. James Kent. Blat: The blast-like alignment tool. *Genome Res.*, 12(4):656–664, 2002.

[KLA+01] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *CPM '01: Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, pages 181–192, London, UK, 2001. Springer-Verlag.

[KS03] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 13th International Conference on Automata, Languages and Programming*. Springer, 2003.

[KSPP03] D.K. Kim, J.S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. Annual Symposium on Combinatorial Pattern Matching*, volume 2676, pages 186–199, Berlin, 2003. Springer-Verlag.

[Kur99a] Stefan Kurtz. Reducing the space requirement of suffix trees. *Software: Practice and Experience*, 29(13):1149–1171, 1999.

[Kur99b] Kurtz, S. and Schleiermacher, C. REPuter: Fast Computation of Maximal Repeats in Complete Genomes. *Bioinformatics*, 15(5):426–427, 1999.

[Kur01] Kurtz, S. and Choudhuri, J.V. and Ohlebusch, E. and Schleiermacher, C. and Stoye, J. and Giegerich, R. REPuter: The Manifold Applications of Repeat Analysis on a Genomic Scale. *Nucleic Acids Res.*, 29(22):4633–4642, 2001.

[Man04] Giovanni Manzini. Two space saving tricks for linear time lcp array computation. In *SWAT*, pages 372–383, 2004.

[MBM93] Peter M. McIlroy, Keith Bostic, and M. Douglas McIlroy. Engineering radix sort. *Computing Systems*, 6(1):5–27, 1993.

[McC76] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[MDS04] Morris Michael, Christoph Dieterich, and Jens Stoye. Suboptimal local alignments across multiple scoring schemes. In *WABI*, pages 99–110, 2004.

[MF02] Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm. In *ESA '02: Proceedings of the 10th Annual European Symposium on Algorithms*, pages 698–710, London, UK, 2002. Springer-Verlag.

[MM93] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[MPK03] Colin Meek, Jignesh M. Patel, and Shruti Kasetty. Oasis: An online and accurate technique for local-alignment searches on biological sequences. In *VLDB*, pages 910–921, 2003.

[MY03] Joseph Bedell Mark Yandell, Ian Korf. *BLAST*. O'Reilly, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2003.

[SA93] Karlin S. and S.F Altschul. Applications and statistics for multiple high-scoring segments in molecular sequences. *Proc. Natl. Acad. Sci. U.S.A.*, 90:5873– 5877, 1993.

[Sew00] Julian Seward. On the performance of bwt sorting algorithms. In *DCC '00: Proceedings of the Conference on Data Compression*, page 173, Washington, DC, USA, 2000. IEEE Computer Society.

[SMS+01] A.A. Schaffer, T.L. Madden, S. Shavirin, J.L. Spouge, Y.I. Wolf, E.V. Koonin, and S.F. Altschul. Improving the accuracy of psi-blast protein database searches with composition-based statistics and other refinements. *Nucleic Acids Res.*, 29:2994–3005, 2001.

[SW81] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[TTHP05] Yuanyuan Tian, Sandeep Tata, A. Hankins, and M. Patel. Practical methods for constructing suffix trees. *The VLDB Journal*, 14(3):281–299, 2005.

[Ukk92] Esko Ukkonen. Constructing suffix trees on-line in linear time. In *Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing '92, Volume 1*, pages 484–492. North-Holland, 1992.

[Wei73] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[WNMB00] Wu, Nevill-Manning, and Brutlag. Fast probabilistic analysis of sequence function using scoring matrices. *BIOINF: Bioinformatics*, 16, 2000.