Algoritmen en datastructuren 2

Anoek Strumane

December 24, 2019

Contents

1 Efficiënt zoeken

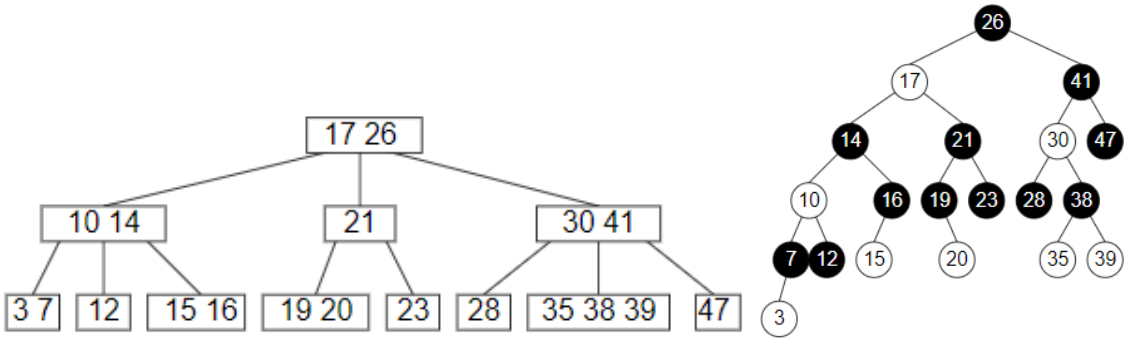
1.1 Inleiding

De uitvoeringstijd van operaties op een binaire zoekboom zijn meestal evenredig met de hoogte, dus $O(\lg h)$. Dit is dus gemiddeld $O(\lg n)$ voor n gegevens maar in het slechtste geval $O(n)$.
Er bestaan verschillende mogelijkheden om zoekbomen er voor te zorgen dat elke toevoegvolgorde even waarschijnlijk is:

- **Elke operatie steeds efficiënt maken**, dit door evenwichtige zoekbomen
  - **AVL-bomen:** hoogteverschil tussen twee deelbomen van elke knoop mag nooit groter zijn dan 2. De hoogte wordt in elke knoop opgeslagen.
  - **2-3-bomen:** elke knoop heeft 2 of 3 kinderen, en elk blad dezelfde hoogte. Dit zijn de voorlopgers van **B-trees** (H3). 2-3-4-bomen analoog. Deze bomen kunnen worden voorgesteld als binaire bomen, waar het perfecte evenwicht wordt opgegeven maar de hoogte blijft $O\lg(n)$. Hieruit werden **rood-zwarte bomen** afgeleid.
- **Elke reeks operaties steeds efficiënt maken**
  - Vb: **splay-trees**: de vorm van de boom wordt steeds aangepast waardoor deze nooit slecht blijft. Geamortiseerde performantie blijft dus goed.
- **Gemiddelde efficiëntie onafhankelijk maken van de operatievolgorde**, door gebruik te maken van een random generator
  - **Randomized search trees** zorgt dat het random blijft, onafhankelijk van operaties
  - **Treap** is de eenvoudigste vorm

1.2 Rood-zwarte bomen

1.2.1 Definitie en eigenschappen



1.2.1.1 Definitie

- = Een binaire zoekboom die een 2-3-4-boom simuleert
  - Een 3-knoop wordt vervangen door 2 binaire knopen en een 4 door 3 binaire knopen.
  - Originele verbindingen zijn zwart, nieuwe rood, deze wordt opgeslagen bij de kinderen
- Ontbrekende kinderen, virtuele knopen, zijn nulwijzers naar eenzelfde "nullknoop" die altijd zwart is.
- Om de hoogtes ongeveer gelijk te houden worden de volgende restricties gelegd:
  1. Elke knoop is rood of zwart
  2. Virtuele knopen zijn zwart
  3. Een rode knoop heeft altijd 2 zwarte kinderen
  4. Elke mogelijke weg van 1 knoop naar zijn virtuele knopen heeft even veel zwarte knopen, de **zwarte hoogte**
  5. De wortel is zwart

1.2.1.2 Eigenschappen

Elke knoop heeft 2 tot 4 kinderen en zijn enkel virtueel op niveau $z-1$.
Op niveau d zijn er dus minstens 2^d en hoogstens 4^d knopen.
Er zijn dus minimaal 2^z-1 knopen:

$$1+2+...+2^{z-1}=2^z-1\leq n$$

Er kunnen geen 2 opeenvolgende rode knopen zijn waardoor de equivalente rood-zwarte boom een h zal hebben dat $h\leq 2z$

$$n\geq 2^z-1\geq 2^{h/2}-1$$

$$h\leq 2\lg(n+1)$$

De hoogte van de rood zwarte boom is dus nog steeds $O(\lg n)$

1.2.2 Zoeken

Gegarandeerd $O(\lg n)$ (geen rekening houden met kleuren)

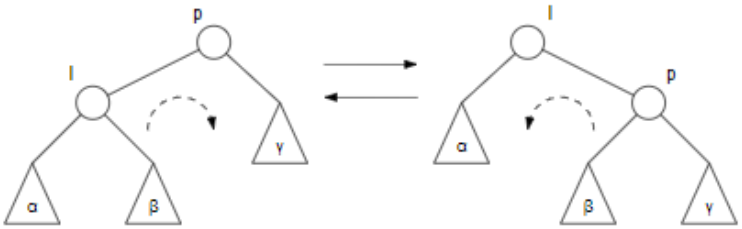
1.2.3 Toevoegen en verwijderen

Bij toevoegen of verwijderen kan men op 2 manieren werken:

1. **Bottom-up:** eerst afdalen toevoegen en daarna terug naar boven om de kleuren te herstellen. Stack of ouderwijzers zijn nodig
2. **Top-down:** boom aanpassen langs dalende zoekweg. Geen stack of ouderwijzers nodig.

1.2.3.1 Rotaties

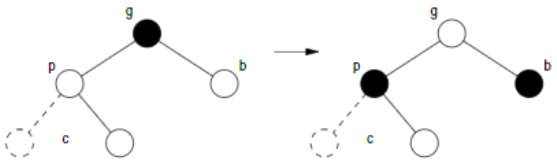
Rotaties wijzigen de vorm maar houden de inorder volgorde. 2 inwendige knopen worden betrokken bij een rotatie, kind en ouder. Een rotatie moet 3 ouder-kind verbindingen aanpassen.



1.2.3.2 Bottom-up rood-zwarte bomen

- **Toevoegen:**
  Nieuwe knoop maken we altijd rood. Er is dus een probleem als de ouder ook al rood is. Aangezien de ouder p van de toe te voegen knoop c rood is, is hij geen wortel en heeft p dus een zwarte ouder dus c een zwarte grootouder g.
  Er zijn 6 mogelijke gevallen: 2 groepen van drie, naargelang dat p een linker of rechter kind is van g. De 2 groepen zijn analoog. We behandelen p als linkerkind van g:

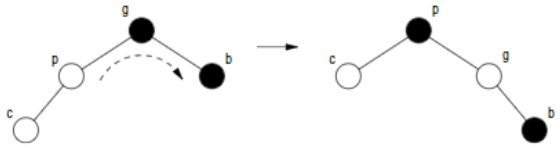
  1. **De broer b van p is rood**

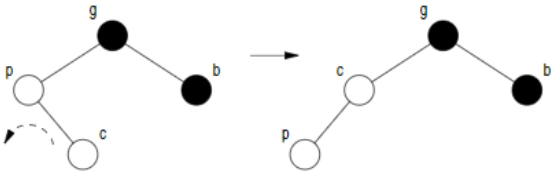
     - De ligging van c tov zijn ouder p is onbelangrijk.
     - p mag zwart worden als zijn broer ook zwart wordt ter compensatie. p en b worden dus zwart, en de grootouder wordt rood om zwarte-hoogte niet te verstoren.
     - Indien echter g een rode ouder heeft, wordt het probleem naar boven opgeschoven of opgelost door een van de andere gevallen.

  2. **De broer b van p is zwart**
     - De rode ouder wordt geëlimineerd door rotatie naar rechts.
     - p komt bovenaan en heeft invloed op beide wegen dus neemt de zwarte kleur van g over
     - g wordt dan rood om de zwarte-hoogte niet te verstoren
     - Dit werkt echter enkel als c, p en g op 1 lijn liggen, dus dit wordt opgesplitst in 2 gevallen:

     (a) **Knoop c is een linkerkind van p**

     (b) **Knoop c is een rechterkind van p**

     p en c worden naar links geroteerd om het voirge geval te bekomen

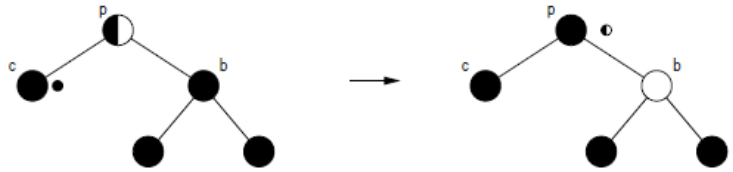
  Efficiëntie: er zijn dus hoogstens 2 rotaties nodig, eventueel voorafgegaan door $O(\lg n)$ keer opschuiven. Roteren en opschuiven zijn $O(1)$ en het initiëel afdalen is $O(\lg n)$ dus **toevoegen is steeds** $O(\lg n)$

- **Verwijderen:**
  Eerst verwijderen we de knoop zoals bij een gewone zoekboom.
  Als de te verwijderen knoop rood is, dan heeft hij enkel virtuele kinderen. Verwijderen is dan eenvoudig en heeft geen invloed op de zwarte hoogte.
  Als de knoop zwart is dan heeft hij ofwel geen 'echte' kinderen of 1 rood kind.
  Een rood kind kan de zwarte kleur overnemen.
  Bij een geen "echte" kinderen krijgt een van die kinderen de kleur "dubbelzwart"
  In bepaalde gevallen zal het probleem worden opgeschoven, waardoor een inwendige knoop c ook dubbelzwart kan worden.

  - Is de dubbelzwarte knoop c de wortel, dan kan deze gewoon verwijderd worden.
  - Stel dat c een ouder p heeft, dan zijn er acht mogelijkheden. Opgesplitst in 2 groepen van 4 (c is linker of rechterkind van p)
  - **Onderstel dat c een linkerkind is van p**. We kunnen oftewel het overtollige zwart samen met een vrijgekomen zwarte kleur van de rechterdeelboom naar boven schuiven OF de linkerboom een extra knoop bezorgen via rotatie naar links en die zwart maken

    1. **Broer b van c is zwart**
       Of b zijn kleur afstaat of roteert hangt af van de kleur van zijn (virtuele) kinderen

       (a) **Broer b heeft 2 zwarte kinderen**

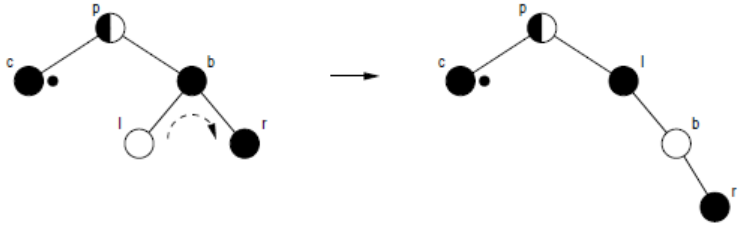
       b kan rood worden en de vrijgekomen zwarte kleur wordt samen met het overtollig zwart van c naar ouder p geschoven.
       Een zwarte ouder p zou nu dubbelzwartworden, waardoor het probleem naar boven wordt geschoven

       (b) **Broer b heeft een rood rechterkind**

       We bezorgen de linkerdeelboom een extra knoop, p, via rotatie naar links, zodat die de overtollige zwarte kleur van c kan overnemen.
       De broer b neemt de oorspronkelijke kleur van p over.
       De rechter boom verliest dan een zwarte knoop, waardoor rode rechterkind van b nu zwart wordt.

       (c) **Broer b heeft een zwart rechterkind en een rood linkerkind**

       We herleiden dit naar het vorige geval door b en zijn rood linkerkind l naar rechts te roteren.
       b wordt zwart en l wordt rood.

2. **Broer b van c is rood**

Ouder p is dan natuurlijk zwart
We herleiden dit naar het eerste hoofdgeval.
c moet een nieuwe zwarte broer kirjgen, die bovendien een inwedige knoop moet zijn omdat we zijn kinderen nodig hebben.
Hiervoor gebruiken we het linkerkind l van b, die heeft 2 zwarte kinderen omdat zijn hoogte minstens 2 is door het dubbelzwart zijn van c.
p en b worden links geroteerd en b wordt zwart en p rood.

- **Efficiëntie:**
- Er zijn dus max 3 rotaties of het probleem wordt naar boven geschoven. (1ste hoofdgeval is max 2 rotaties, 2de hoofgeval is 1 rotatie om naar 1ste hoofdgeval te komen)
- Verwijderen is dus maximaal 3 keer roteren $O(1)$, en $O(\lg n)$ keer roteren/opschuiven $O(1)$. Het initieel afdalen is ook $O(\lg n)$.
- **Verwijderen is dus ook** $O(\lg n)$

1.2.3.3 Top-down rood-zwarte bomen

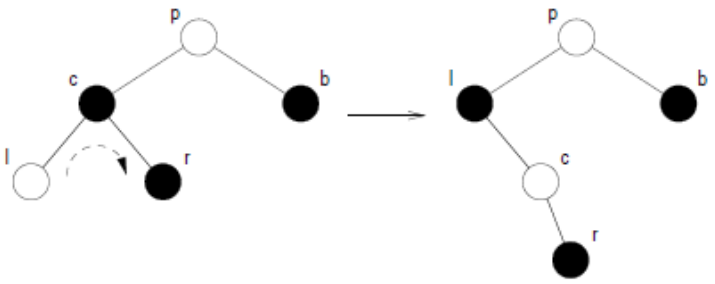
- **Toevoegen:**
  Opnieuw vervangen we een virtuele knoop, maken we hem rood en geeft dit enkel problemen indien de ouder p rood is.
  We hebben gezien dat dit met een rotatie en kleurwijziging kan opgelost worden indien de oom van de nieuwe knoop zwart is.
  Bij een rode oom zouden we moeten terugkeren in de boom, dus tijdens het afdalen moeten we zorgen dat de oom steeds zwart is.

  – Op de weg naar beneden worden dus geen 2 rode broers toegelaten.
  – Als we dit tegenkomen, wordt de ouder rood, en de 2 kinderen zwart
  – Als de ouder van deze rood gemaakte knoop rood is, is er een probleem en kan dit opgelost worden met rotaties en kleurwijzigingen omdat zijn oom al gegarandeerd zwart is.
  – Bij een rotatie zijn 3 verschillende generaties nodig, er moeten dus altijd 3 boompointers bijgehouden worden

  Efficiëntie: Toevoegen daalt in de boom en er gebeuren kleurwijzigingen en rotaties. **Toevoegen is dus** $O(\lg n)$

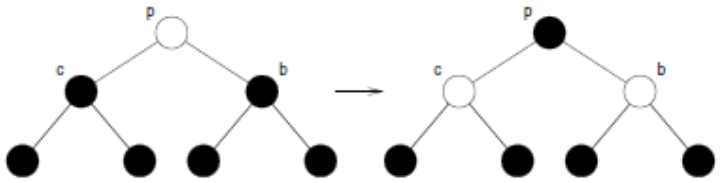
- **Verwijderen:**
  Een te verwijderen knoop wordt vervangen door zijn opvolger of voorganger tot het vanonder komt te staan en dan verwijderd.
  De hoogte van de te verwijderen knoop is 1, omdat minstens 1 van zijn kinderen virtueel is
  Om de zwarte hoogt geldig te houden moet deze te verwijderen knoop rood zijn, en zijn kinderen moeten beiden virtueel zijn
  We maken dus elke knoop dat we onderweg tegenkomen rood.
  We onderscheiden twee groepen van mogelijkheden en behandelen er 1:

  1. **Knoop c heeft minstens 1 rood kind**

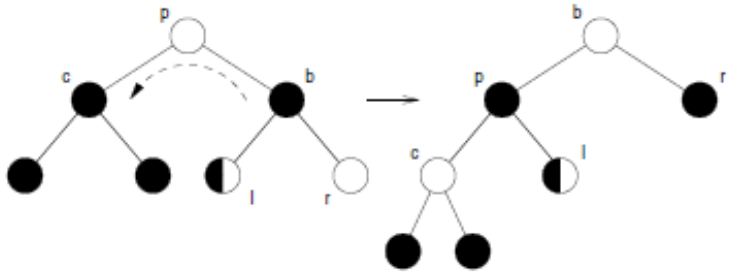


  Als we naar het rood kind moeten afdalen komen we terug in de beginsituatie.
  Als we naar het zwart kind moeten afdalen of c de fysisch te verwijderen knoop is, dan maken we c rood door hem te roteren met zijn rood kind, en beiden hun kleuren om te wisselen.
  Zo komen we opnieuw in de beginsituatie of kunnen we c gewoon verwijderen

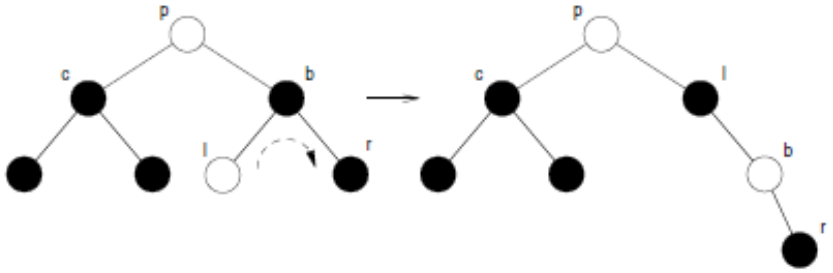
2. **Knoop c heeft 2 zwarte kinderen**
   Om c rood te maken moeten we beroep doen op zijn zwarte broer b (c heeft er een omdat we zijn ouder reeds rood hebben gemaakt)
   (a) **Broer b heeft twee zwarte kinderen**



   b en c worden rood, p zwart
   (b) **Broer b heeft een rood rechterkind**



   c moet rood worden, dus er moet een zwarte knoop bij komen in de linker deelboom
   Er wordt geroteerd naar links, c wordt rood en de kleuren van de oorspronkelijke broer van c en zijn rechterkind worden omgewisseld
   (c) **Broer b heeft een zwart rechterkind en een rood linkerkind**



   We roteren l en b naar rechts en verwisselen hun kleuren
   Dan roteren we p en l naar links, maken c rood en p zwart

Efficiëntie: verwijderen gebeurt door af te dalen in de boom, waarbij rotaties en kleurwijzigingen gebeuren. **De performantie is dus ook steeds** $O(\lg n)$

1.2.4 Vereenvoudigde rood-zwarte bomen

- AA-bomen: enkel rechterkinderen mogen rood zijn
- Binary B-trees
- left-leaning red-black tree: knoop mag enkel een rood rechterkind hebben als hij ook een rood linkerkind heeft

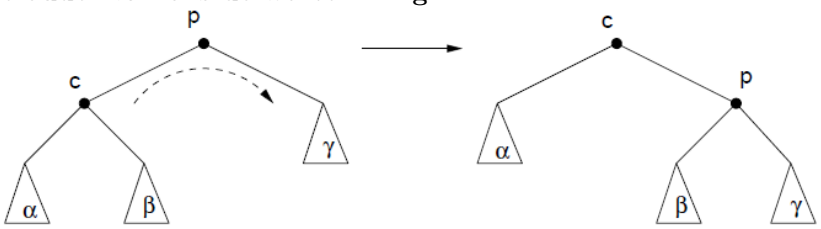
1.3 Splay trees

Een splay tree is een binaire zoekboom met een extra operatie, de **splay operatie**. Dit zet de laatst gebruikte knoop als wortel. Het verschil is dus dat nieuw toegevoegde en laatste gebruikte knopen altijd vanboven staan idpv vanonder zoals bij RZ-bomen en gewone binaire bomen.
Splay trees hebben 1 zekerheid: als een reeks operaties lang is, dan is hun geamortiseerde efficiëntie gegarandeerd $O(\lg n)$.
Er moet geen hoogte of andere informatie over het evenwicht worden bijgehouden.

1.3.1 Bottom-up splay trees

Eerst zoeken we de knoop, en dan pas wordt de boom naar boven aangepast. Ouderwijzers of stapel zijn dus nodig.
Rotaties moeten zorgvuldiger aangepakt worden, zodat de situatie van de knopen op de zoekweg ook verbetert en bv niet het slechtste geval kan worden, een gelinkte lijst.
We onderscheiden 3 mogelijkheden om knoop c vanboven te krijgen:

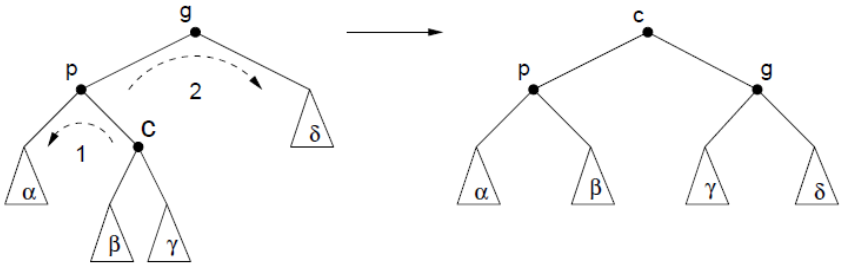
1. **De ouder van c is de wortel = zig**



We roteren beiden knopen

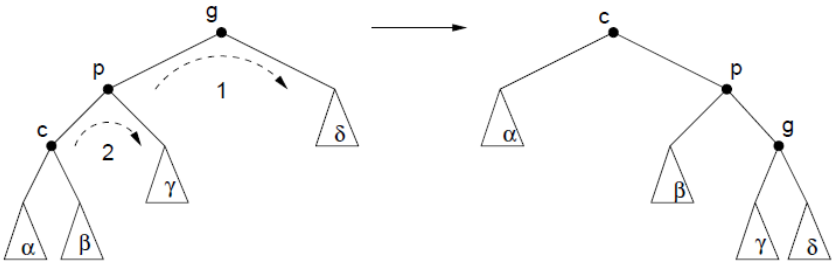
2. **Knoop c heeft een grootouder g**
Stel dat p linkerkind is van g, dan onderscheiden we 2 gevallen

   (a) **Knoop c is rechterkind van p = zig-zag**



   p en c naar **links** roteren
   g en c naar **rechts** roteren

   (b) **Knoop c is linkerkind van p = zig-zig**



   g en p naar **rechts** roteren
   p en c naar **rechts** roteren

Ondiepe knopen zakken maximaal 2 niveaus.

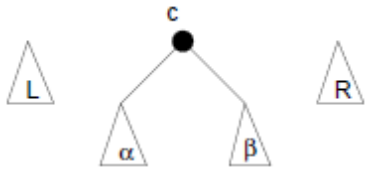
1.3.1.1 Woordenboek operaties

- **Zoeken:** zoals een gewone zoekboom zoeken, en dan de gezochte knoop de wortel maken door de splay-operatie.
- **Toevoegen:**
  1. **Optie 1:** gewoon toevoegen en dan nieuwe knoop naar boven verplaatsen
  2. **Optie 2:** we gaan eerst opzoek naar de knoop waaraan we moeten toevoegen en zetten deze vanboven. Hierdoor staat ofwel de opvolger of de voorloper van de toe te voegen knoop vanboven. Daarna de wordt de wortel vervangen door onze toe te voegen knoop.
- **Verwijderen:**
  1. **Optie 1:** zoals normaal verwijderen en dan de ouder de wortel maken
  2. **Optie 2:** eerste de te verwijderen knoop zoeken, deze wortel maken en dan verwijderen. Linker of rechterkind wordt dan de nieuwe wortel

1.3.2 Top-down splay trees

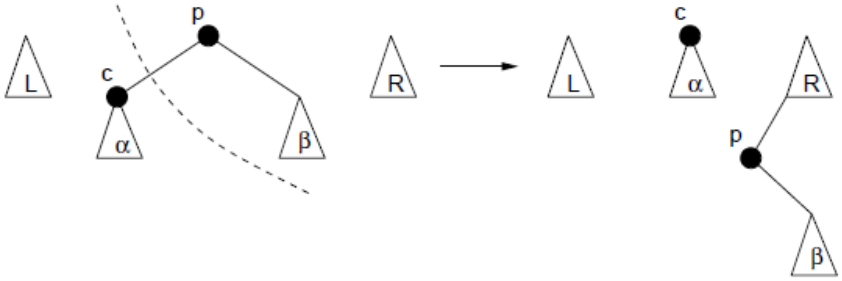
De splay operatie wordt uitgevoerd tijdens de afdaling, zodat de gezochte knoop wortel is wanneer we hem bereiken.
Tijdens het afdalen wordt de boom in 3 zoekbomen opgedeeld:

1. **Links:** alles dat ik weet dat kleiner is dan wat ik zoek
2. **Midden:** alles dat ik nog niet onderzocht heb
3. **Rechts:** alles dat ik weer dat groter is dan wat ik zoek



Wat moet ik doen om stukken aan L of R toe te voegen? Er zijn dan 3 mogelijkheden.
We moeten vanuit p naar zijn linkerkind c

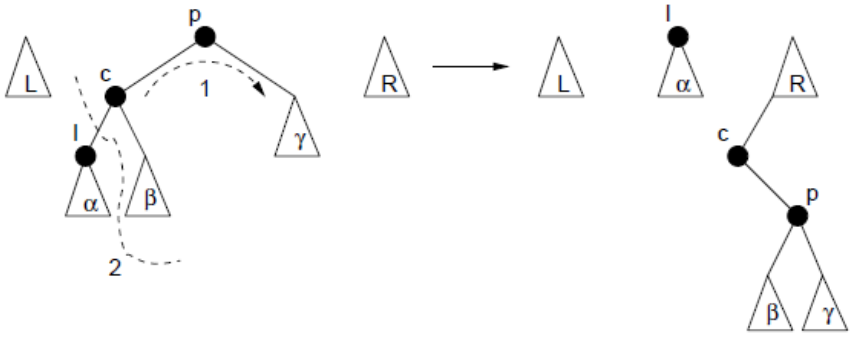
1. **Linkerkind c is de laatste knoop op de zoekweg = zig**



Dit gebeurt indien c de gezochte knoop is, of we naar de richting moeten waar c geen kinderen meer heeft.
Knoop p wordt het nieuwe kleinste element van R (de rechtse deelboom van p gaat dus ook mee).
De linkse deelboom van p wordt de nieuwe M met c als wortel

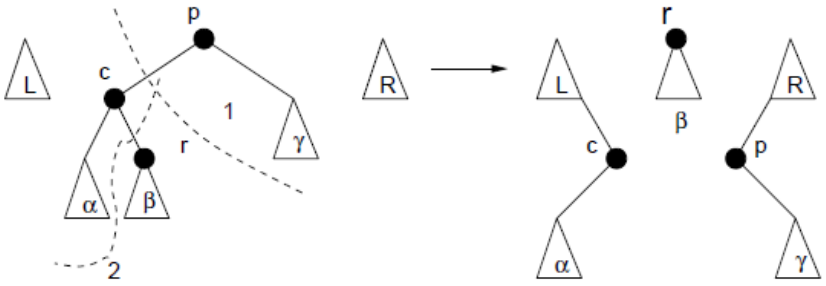
2. **Linkerkind c is niet de laatste knoop op de zoekweg**

   (a) **We moeten naar het linkerkind l van c = zig-zig**



   p en c naar rechts roteren
   c is nu het nieuwe kleinste element van R
   De linkse deelboom van c wordt de nieuwe M met l als wortel

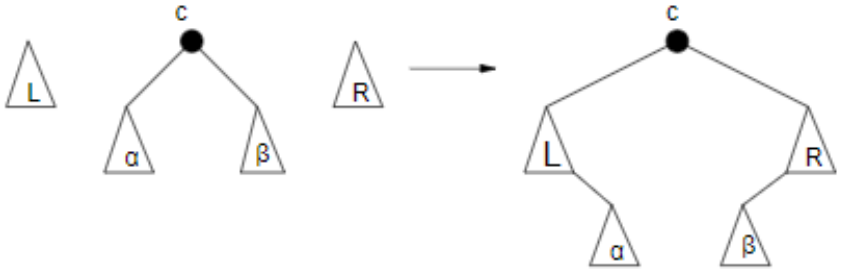
   (b) **We moeten naar het rechterkind r van c = zig-zag**



   p wordt het nieuwe kleinste element van R en de rechtse deelboom van p gaat mee
   c wordt het nieuwe grootste element van L
   De rechtse deelboom van c wordt de nieuwe M
   Alternatief:

   - Beschouwen het als zig, p wordt het kleinste element van R, maar nu wordt de linkse deelboom van p de nieuwe M met c als wortel
   - Voordeel: minder gevallen te onderscheiden
   - Nadeel: slechts 1 niveau afgedaald idpv 2

Wanneer c de wortel van M is geworden, voltooien we de splay om de 3 bomen samen te voegen tot 1 boom met als wortel c.
Alle sleutels in de linkse deelboom van c zijn groter dan de sleutels in L , dus brengen we deze deelboom volledig onder in L. IDEM voor rechts.
L en R worden nu deelbomen van c.



1.3.2.1 Woordenboek operaties

- **Zoeken:** knoop met de wortel wordt sleutel
- **Toevoegen:** analoog aan de alternatieve versie bottom-up
- **Verwijderen:** analoog aan de alternatieve versie bottom-up

1.3.3 Performantie van splay trees

We tonen aan dat een splay tree met maximaal n knopen een performantie van $O(m \lg n)$ heeft. Dus een geamortiseerde efficiëntie van $O(\lg n)$
Omdat de vorm van de boom kan veranderen, krijgt elke mogelijke vorm een reëel getal toegewezen, zijn **potentiaal**.
Efficiënte operaties verhogen de potentiaal

BEWIJS
De geamortiseerde tijd

- t_i is de werkelijke tijd van de i-de operatie
- a_i haar geamortiseerde tijd
- Φ_i de potentiaal na deze operatie
- De geamortiseerde tijd voor een operatie wordt dan
  (met Φ_{i-1} de potentiaal zoals de vorige operatie de boom heeft achtergelaten, dus voor de i-de operatie)
  $$a_i = t_i + \Phi_i - \Phi_{i-1}$$
- De geamortiseerde tijd van een reeks van m operaties wordt dan
  $$\sum_{i-1}^{m} a_i = \sum_{i=1}^{m} (t_i + \Phi_i - \Phi_{i-1})$$
- In deze som komen de meeste potentialen in tegengestelde tekens voor, dus
  (Met Φ_m de eindpotentiaal en Φ_0 beginpotentiaal)
  $$\sum_{i-1}^{m} a_i = \sum_{i=1}^{m} t_i + \Phi_m - \Phi_0$$

De juiste keuze van de potentiaalfunctie

- Als Φ_0 klein is (of Φ_m niet kleiner is dan Φ_0), dan geeft een afschatting van de geamortiseerde tijd een *bovengrens* voor de werkelijke tijd.
- Aan elke knoop i wordt een gewicht s_i gegeven, gelijk aan het aantal knopen in zijn deelboom
- De potentiaal wordt dan de som van de logaritmen van deze gewichten
  $$\Phi = \sum_{i=1}^{n} \lg s_i$$
- We noteren $r_i = \lg s_i$ als de *rang* van knoop i

Performantie analyse van bottom-up splay trees
(Volledig analoog aan top-down)

- De performantie van een operatie is evenredig met de diepte van de knoop, dus het aantal rotaties
- We tonen aan dat zoeken naar een knoop c, gevolgd door een splay-operatie een geamortiseerde tijd heeft van
  (r_w = rang wortel)
  $$O(1 + 3(r_w - r_c))$$
  (bewijs is niet te kennen)

Bovengrenzen

- De bovengrenzen van de 3 soorten operaties bevatten allemaal de positieve term $r'_c - r_c$ ($r'_c > r_c$)
- We maken alle coëfficiënten gelijk aan de grootste waarden (want we zoeken een bovengrens), dus die van de *zig-zig*.
- Aan het einde is c de wortel, met als rang idd r_w, waarmee de eigenschap is aangetoond

Woordenboekoperaties

- **Zoeken:** $O(1 + 3 \lg n)$ want $s_w = n$ en voor een blad is $s_c = 1$
- **Toevoegen:** (eerst afdalen zoals zoeken, en de rang van de knopen op de zoekweg $(p_1, p_2, ..., p_k)$ worden aangepast)
  - s_{p_i} = gewicht van de knoop p_i voor het toevoegen van de nieuwe knoop
  - s'_{p_i} = gewicht erna
  - Potentiaal toename is dan gelijk aan
    $$\lg\left(\frac{s'_{p_1}}{s_{p_1}}\right) + \left(\frac{s'_{p_2}}{s_{p_2}}\right) + ... + \left(\frac{s'_{p_k}}{s_{p_k}}\right) = lg\left(\frac{s'_{p_1}}{s_{p_1}}\frac{s'_{p_2}}{s_{p_2}}...\frac{s'_{p_k}}{s_{p_k}}\right)$$
  - Nu is zowel $s_{p_{i-1}} \geq s_{p_i} + 1$ als $s'_{p_i} = s_{p_i} + 1$ en dus ook $s_{p_{i-1}} \geq s'_{p_i}$
    $$\lg\left(\frac{s'_{p_1}}{s_{p_k}}\right) \leq \lg(n+1)$$
  - De geamortiseerde tijd van toevoegen is dus
    $$O(1 + 4 \lg n)$$
- **Verwijderen** doet de rang van de knopen op de zoekweg dalen zodat het effect nooit positief is

De geamortiseerde tijd van woordenboek operaties

- n_i is het aantal knopen bij de i-de operatie:
  $$O(m + 4 \sum_{i=1}^{m} \lg n_i)$$
- En als de boom maximaal n knopen bevat is dat resultaat zeker $O(m + 4m \lg n)$ of $O(m \lg n)$

1.4 Randomized search trees

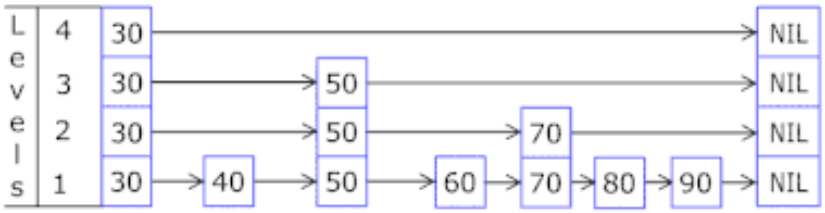
Randomized search trees maken gebruik van een random generator om het effect van operatievolgorde te neutraliseren.
Een **treap** is de eenvoudigste vorm hiervan. Elke knoop krijgt naast een sleutel ook een random prioriteit toegekend, waaraan de heap voorwaarde van de treap moet voldoen. (Het blijft ook nog steeds een binaire zoekboom)
Als alle sleutels en prioriteiten anders zijn, is er maar 1 treap mogelijk, met als gevolg dat de vorm van de treap niet afhangt van de toevoegvolgorde van de sleutels

Woordenboekoperaties op een treap

- **Zoeken:** identiek aan gewone zoekboom
- **Toevoegen:** gebeurt als blad zoals bij gewone zoekbomen, daarna houdt men rekening met de prioriteiten. Daarna wordt de knoop dus geroteerd om aan de heapvoorwaarde te voldoen. Het aantal gevallen bij rotaties is hier heel beperkt want het heeft geen gevolgen voor heapvoorwaarden elders in de boom.
- **Verwijderen:** we geven hem de laagste prioriteit, roteren hem naar onder en verwijderen hem.

De verwachtingswaarde van de efficiëntie bij een goede randomgenerator is hier dus $O(\lg n)$

1.5 Skip lists



Een skip list wordt beschouwd als een vector van gelinkte lijsten
Het onderste niveau bevat alle sleutels (de boom is dus evenwichtig), de lijst daarboven bevat een gedeelte van die sleutels, en zo verder. Knopen op een niveau hebben een wijzer naar hun overeenkomsige knoop op het niveau onder hun. **Woordenboek operaties**

- **Zoeken:** sleutel is niet de bovenste? Niveau lager en naar rechts tot ik een sleutel vind die kleiner is dan degene die ik zoek, en zo verder naar beneden.
- **Toevoegen:** Sleutel wordt aan een random niveau toegekend en toegevoegd aan alle niveaus eronder.
- **Verwijderen:** onderste verwijderen, dan de niveau's erboven

2 Toepassingen van dynamisch programmeren

Inleiding van in de les: Indien je problemen kan opdelen in kleinere problemen kan je ze makkelijk voorstellen als een boom. Indien hetzelfde probleem meerdere keren voorkomt, kun je het bijhouden om dan later te checken of je een bepaald probleem al eens hebt opgelost. Deze bomen behandel je best *bottum-up* indien je al goed weet welke problemen je gaat moeten oplossen (deeloplossingen zijn gekend) en *top-down* indien je dit niet weet.

2.1 Optimale binaire zoekbomen

Stel dat we op voorhand weten welke gegevens er moeten worden opgeslaan en we de waarschijnlijkheid weten waarmee ze worden opgezocht. We proberen dan de boom te schikken zodat de verwachtingswaarde van de zoektijd minimaal wordt.

De zoektijd wordt bepaald door de lengte van de zoekweg, dus de diepte van een knoop of de diepte van een ledige deelboom (voor afwezige sleutels). We beschouwen ledige deelbomen als bladeren en aanwezig gegevens zijn dan inwendige knopen.

BEWIJS
De verwachtingswaarde van de zoektijd

- De gerangschikte sleutels van de n aanwezige sleutels noemen we $s_1, s_2, ..., s_n$
- De $n+1$ bladeren noemen we $b_0, b_1, ..., b_n$
- b_0 staat voor alle afwezige sleutels die kleiner zijn dan s_1, b_n voor alle groter dan s_n, en b_i voor alle sleutels tussen s_i en s_{i+1}
- De waarschijnlijkheid dat de i-de *aanwezige* sleutel gezocht wordt is p_i
- De waarschijnlijkheid dat de i-de *ontbrekende* sleutel gezocht wordt is q_i
- Zoeken levert altijd een knoop of blad op dus

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

- We stellen de zoektijd gelijk aan het aantal knopen op de zoekweg (diepte + 1), dan is de verwachtingswaarde van de zoektijd

$$\sum_{i=1}^{n} (diepte(s_i) + 1)p_i + \sum_{i=0}^{n} (diepte(b_i) + 1)q_i$$

Nagaan of dit probleem in aanmerking komt tot dynamisch programmeren
We moeten de zoekboom zoeken die deze uitdrukking minimaliseert. Dynamisch programmeren biedt hier een oplossing.

1. **De deelproblemen zijn onafhankelijk:**
   Hiervoor moeten alle deelproblemen optimaal zijn, dit is voldaan want deelbomen hebben geen gemeenschappelijke knopen

2. **De deelproblemen zijn overlappend:**
   We bedenken eerst een oplossingsstrategie

   - Elke deelboom bevat een reeks opeenvolgende sleutels $s_i, ..., s_j$
   - Met bijhorende bladeren $b_{i-1}, ..., b_j$
   - Een van die sleutels s_w $(i \leq w \leq j)$ zal de wortel zijn van de optimale deelboom
     - De *linkse* deelboom van s_w bevat $s_i, ..., s_{w-1}$ en $b_{i-1}, ..., b_{w-1}$
     - De *rechtse* deelboom bavet $s_{w+1}, ..., s_j$ en $b_w, ..., b_j$
   - Opdat s_w een optimale boom is moeten beiden zijn deelbomen optimaal zijn

De verwachte zoektijd voor deelbomen
We moeten dus de kleinste verwachte zoektijd $z(i, j)$ bepalen voor een deelboom met sleutels $s_i, ..., s_j$ en bijhorende bladeren $b_{i-1}, ..., b_j$ voor alle $1 \leq i \leq n + 1$ en $0 \leq j \leq i - 1$ met $j \geq i - 1$

- Om $z_w(i, j)$ te zoeken bepalen we die van zijn deelbomen $z(i, w - 1)$ en $z(w + 1, j)$
- De bijdrage tot de zoektijd van deelboom $s_i, ..., s_j$ wordt gegeven door

$$g(i, j) = \sum_{k=i}^{j} p_k + \sum_{k=i-1}^{j} q_k$$

- Met s_w als wortel krijgen we dat

$$z_w(i, j) = p_w + (z(i, w - 1) + g(i, w - 1)) + (z(w + 1, j) + g(w + 1, j))$$

- Of eenvoudiger

$$z(i, j) = z(i, w - 1) + z(w + 1, j) + g(i, j)$$

- Index w doorloopt alle waarde tussen i en j (inbegrepen) en we nemen de kleinste waarde

$$z(i, j) = \min_{i \leq w \leq j} \{z_w(i, j)\} = \min_{i \leq w \leq j} \{z(i, w - 1) + z(w + 1, j)\} + g(i, j)$$

- Indien de deelboom geen sleutels bevat is $j = i - 1$ of en heeft de deelboom enkel b_{i-1}
- Om de volledige optimale boom te hebben houden we de indexen w van de wortel van elke optimale deelboom bij in $r(i, j)$

Implementatie van het algoritme
We bepalen de deeloplossing bottom-up en slaan de resultaten op in tabellen zodat ze hogerop beschikbaar zijn.
De implementatie gebruikt 3 tabellen:

1. 2D tabel $z[1...n+1][0...n]$ voor $z(i,j)$ (we hebben $z(1,0)$ nodig voor boom met enkel b_0 en $z(n+1,n)$ voor boom met b_n)
2. 2D tabel $g[1...n+1][0...n]$ voor $g(i,j)$. Om deze tabel in te vullen gebruiken we $g(i,j) = g(i,j-1) + p_j + q_j$ voor $i \leq j$ en $g(i,i-1) = q_{i-1}$
3. 2D tabel $r[1...n][1...n]$ voor $r(i,j)$

$z(i,j)$ en $g(i,j)$ moeten ook in de juiste volgorde bepaald worden: eerste de diagonalen vullen $(i,i-1)$ en daarna de elk evenwijdige diagonaal, in de richting van de tabelhoek linksboven (enkel linker-boven halve driehoek van de tabel moet ingevuld worden)
$g(i,j)$ wordt enkel berekend met $g(i,j-1)$ en daarna kunnen we hiermee $z(i,j)$ berekenen
Zie cursus pagina 32 voor pseudocode
De performantie

- **Bovengrens:** De drie vernestelde herhalingen kunnen elk niet meer dan n iteraties uitvoeren is het zeker $O(n^3)$
- **Ondergrens:** Een deelboom met sleutels $s_i, ..., s_j$ met $1 \leq i \leq j \leq n$ heeft $j-i+1$ mogelijke wortels en elke test is $O(1)$.
  //Aan te vullen, denk niet dat we dit moeten kennen (maar ook $O(n^3)$)

2.2 Langste gemeenschappelijke deelsequentie

Dit soort probleem heeft een *optimale deelstructuur.*
Gegeven zijn twee strings $X = \langle x_0, x_1, ...x_{n-1}\rangle$ en $Y = \langle x_0, x_1, ...x_{m-1}\rangle$.
De deelproblemen zijn paren prefixen van de twee strings: het prefix van X met lengte i noemen we X_i en het ledige prefix X_0.
Als $Z = \langle x_0, x_1, ...x_{k-1}\rangle$ een LGD is van X en Y, dan zijn er 3 mogelijkheden

- Als $n = 0$ of $m = 0$ dan is $k = 0$
- Als $x_{n-1} = y_{m-1}$ dan is $z_{k-1} = x_{n-1} = y_{m-1}$ en is Z_{k-1} een LGD van X_{n-1} en Y_{m-1}
- Als $x_{n-1} \neq y_{m-1}$ dan is Z een LGD van X_{n-1} en Y met $z_{k-1} \neq x_{n-1}$ of een LGD van Y_{m-1} en X met $z_{k-1} \neq y_{m-1}$

Lengte van de LGD
We stellen een recursieve vergelijking op voor de lengte van een LGD. $c[i,j]$ (dimensies $(n+1)*(m+1)$) is de lengte van de LGD van X_i en Y_i:

$$c[i,j] = 0 \ \forall \ i = 0 \ of \ j = 0$$

$$c[i,j] = c[i-1,j-1] + 1 \ \forall \ i > 0 \ en \ j > 0 \ en \ x_i = y_j$$

$$c[i,j] = max(c[i,j-1], c[i-1,j]) \ \forall \ i > 0 \ en \ j > 0 \ en \ x_i \neq y_j$$

De waarden worden rij per rij van links naar rechts berekend. De eerste rij en kolom worden geïnitialiseerd met nullen. Er zijn dus $\Theta(nm)$ verschillende deelproblemen
De effectieve LGD
In een $n*m$ tabel b in $b[i,j]$ houden we de plaats bij van de c-waarde die gebruikt werd om $c[i,j]$ te bepalen. Vertrekkend uit $b[n,m]$ vinden we de LGD in omgekeerde volgorde: wanneer $c[i,j]$ gebruik maakt van $c[i-1,j-1]$ is $x_i = y_j$
Performantie
De reconstructie van de LGD is $O(n+m)$, bij elke iteratie wordt i of j geïncrementeerd. Constructie van de c-tabel was (nm). De vereiste plaats is ook $\Theta(nm)$.
Indien echter enkel de lengte gewenst is, volstaat enkel c. En aangezien elke rij enkel waarden van de vorige rij gebruikt, volstaan twee rijen. De vereiste plaats is dan $\Theta(min(n,m))$

3 Uitwendige gegevensstructuren

Wanneer de omvang van gegevens te groot is voor het intern geheugen, moeten ze in een extern geheugen worden opgeslaan. Lees en schrijf operaties hiernaartoe zijn niet performant en moeten dus zo veel mogelijke geminimaliseerd worden.

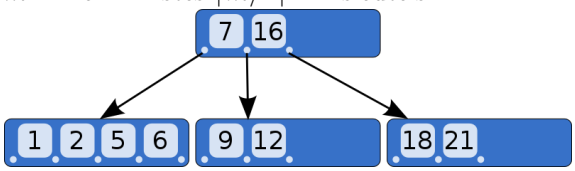
3.1 B-trees

= Een uitwendige evenwichtige zoekboom
De efficiëntie van de meeste operaties wordt begrensd door de hoogte van de boom
De informatie-overdracht van en naar een schijf gebeurt in pagina's, dus we nemen knopen die een volledige schijfpagina beslaan, en we geven ze zoveel kinderen als ze kunnen bevatten.
We houden de wortel en soms ook het volledige eerste niveau van knopen in het geheugen (omdat die vaak gebruikt worden en anders moeten we die altijd ophalen)
De bladeren van een B-tree moeten altijd op dezelfde hoogte blijven

3.1.1 Definitie

Een B-tree van orde m, waarbij $m > 2$, wordt als volgt gedefinieerd:

- Elke inwendige knoop heeft hoogstens m kinderen
- Elke inwedige knoop (behalve de wortel) heeft minstens $\lceil m/2 \rceil$ kinderen. De wortel minstens 2, tenzij het een blad is
- Elke inwendige knoop met $k + 1$ kinderen bevat k sleutels. De bladeren bevatten hoogstens $m - 1$ en minstes $\lceil m/2 \rceil - 1$ sleutels



- Alle bladeren bevinden zich op hetzelfde niveau

Elke knoop bevat dan:

- Een geheel getal k dat het huidig aantal sleutels aanduidt
- Een tabel voor m wijzers naar kinderen van de knoop
- Een tabel voor maximaal $m-1$ sleutels die ze bevatten, stijgend gerangschikt. En een tweede tabel van dezelfde grote met bijhorende informatie of wijzers ernaar.
- Een logische waarde b die aanduid of het een blad is of niet

3.1.2 Minimale hoogte

Stel dat de boom niet ledig is en hoogte h heeft. De wortel heeft slechts minstens 1 sleutel en dus 2 kinderen. Elk van die kinderen heeft minimaal $g = \lceil m/2 \rceil$ kinderen.

- Het aantal minimaal aantal *knopen* is

$$1 + 2 + 2g + ... + 2g^{h-1} = 1 + 2\sum_{i=1}^{h-1} g^i$$

- Elke knoop heeft minstens $g - 1$ sleutels, behalve de wortel (minstens 1)

$$n \geq 1 + 2(g-1)\left(\frac{g^h - 1}{g - 1}\right)$$

- Zodat

$$n \geq 2g^h - 1$$

- Na het nemen van het logaritme met basis $\lceil m/2 \rceil$

$$h \leq \log_{\lceil m/2 \rceil} \frac{n+1}{2}$$

De hoogte is dus $O(\lg n)$

3.1.3 Woordenboek operaties

3.1.3.1 Zoeken

Knoop inlezen, zoeken in gerangschikte tabel, vinden of een volgende knoop inlezen. Als we het niet vinden is het natuurlijk niet aanwezig.
Het aantal schijfoperaties is dus $O(h) = O(\log_{\lceil m/2 \rceil} n)$ en de processortijd per knoop is $O(m)$, en dus in totaal $O(m \log_{\lceil m/2 \rceil} n)$

3.1.3.2 Toevoegen

We bespreken de bottom-up versie
Initialisatie
De wortelknoop in het geheugen aanmaken en gedeeltelijk invullen, dan kopiëren naar een schijf. Een verwijzing naar die plaats moet permanent bijgehouden worden (de wortel kan veranderen)
Toevoegen

- Vanuit de wortel zoeken we het juiste blad waar hij in moet en we voegen hem toe
- Als dat blad al m sleutels bevat, wordt het geplitst bij de middenste sleutel. Een nieuwe knoop wordt aangemaakt op dat niveau, en de middenste sleutel zelf gaat naar de ouder. Een deel van de tabel van de ouder moet dus opgeschoven worden
- Als die ouder ook al m sleutels bevat moet er verder opgeschoven worden naar boven
- Splitsen wordt soms uitgesteld door gegevens over te brengen naar een broer knoop(als die plaats heeft). Dit gebeurt weer adhv een soort rotatie: een sleutel wordt naar de ouder gestuurd, en ouder staat een sleutel af aan de broer
- Slechtste geval: splitsen tot aan de wortel, dan moet er een nieuwe wortel worden toegevoegd. **B-trees groeien dus vanboven**

Performantie

- In het slechtste geval wordt er dus $h+1$ keer gesplitst, maar dit is heel zeldzaam. Een knoop splitsen vereist 3 schrijfoperaties en een processortijd van $O(m)$. In het slechtste geval moeten we 2x de hoogte van de boom doorlopen (zoeken naar beneden en splitsen naar boven).
- Het totaal aantal schijfoperaties is dus $\Theta(h) = \Theta(\log_{\lceil m/2 \rceil} n)$
- **Dit geeft dus een totaal van** $O(mh) = O(m \log_{\lceil m/2 \rceil} n)$

3.1.3.3 Verwijderen

- We moeten altijd uit een blad verwijderen.
- Indien het in een inwendige knoop zit, vervangen we het door zijn voorloper of opvolger (die zit swso in een blad) en dan verwijderen we de sleutel. Dus eerst helemaal naar beneden en dan naar boven. (Zeldzaam want de meeste zitten in bladeren)
- Wanneer een knoop te weinig sleutels overhoudt, kunnen we proberen om er een te lenen van een broer, adhv dezelfde rotatie als eerder besproken. Men brengt soms meerdere sleutels over zodat ze ongeveer hetzelfde aantal hebben.
- Als er geen broers zijn die sleutels kunnen missen dan doen we het omgekeerde van splitsen: de knoop wordt samengevoegd met een broer, zodat een knoop uit de boom verdwijnt.
- Kan weer tot de wortel gaan

Performantie

- In het slechtste geval: de boom 2 maal helemaal doorlopen met een constant aantal schijfoperaties per niveau
- Het totaal aantal schijfoperaties is dus $\Theta(h) = \Theta(\log_{\lceil m/2 \rceil} n)$
- **Dit geeft dus een totaal van** $O(mh) = O(m \log_{\lceil m/2 \rceil} n)$

3.1.4 Varianten van B-trees

3.1.4.1 B^+-tree

Nadelen van B-trees: bladeren moeten plaats reserveren voor kinderen die niet gebruikt worden, inwendige knopen die gegevens bevatten maakt verwijderen ingewikkelder en zoeken naar de opvolger van een sleutel kan $O(\log_{\lceil m/2 \rceil} n)$ schijfoperaties vereisen.

B^+-tree slaat alle gegevens op in bladeren. Inwendige knopen zijn dan gewoon *indexen* en bladeren zijn een gelinkte lijst in sleutelvolgorde.

Maximale graad van inwendige knopen kan groter worden (ze moeten enkel sleutel en kindwijzers bevatten) en ze moeten geen plaats reserveren voor kinderwijzers.
Door de knopen beter te gebruiken wordt de hoogte kleiner en de gelinkte lijst (sequentie) van de bladeren zorgt ervoor dat zoeken naar een opvolger/voorloper 1 schijfoperatie vereist.
Woordenboek operaties gebeurt nagenoeg zoals bij een gewone B-tree

3.1.4.2 Prefix B^+-tree

Als de sleutels strings zijn, gebruiken we als sleutel een zo kort mogelijke string om te onderscheiden. Dit zorgt ervoor dat er minder plaats nodig is voor sleutels.

3.1.4.3 B^*-trees

Een B^*-tree verdeelt (wanneer er gesplitst moet worden) de sleutels van een knoop en de volle buur over 3 knopen, zodat die elk voor ongeveer een twee-derden gevuld zijn. Betere gevulde knopen betekent een minder hoge boom.

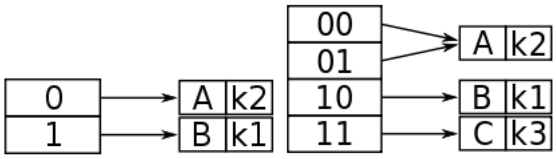
3.2 Uitwendige hashing

Voor wanneer men niet geïnteresseerd is in de volgorde van de sleutels.
Elke pagina in het uitwendig geheugen komt overeen met een hashwaarde

3.2.1 Extendible hashing

We onderstellen dat de hashtabel opgeslagen is in het inwendig geheugen. Deze bevat dan wijzers naar de pagina's, die maximaal m gerangschikte sleutels met bijhorende gegevens bevatten.

- De laatste (prefix) d bits van het gehashte woord met lengte w gebruikt als indices in de hashtabel.
- De tabel bevat dus 2^d elementen met $d < w$, met d de *globale diepte*
- Alle sleutels waarvan de hashwaarden met dezelfde d bits eindigt worden in overeenkomstige pagina's opgeslagen
- Meerdere tabel elementen mogen naar dezelfde pagina wijzen. Er moet dus een k bijgehouden worden die het aantal bits aangeeft waarmee al haar hashwaarden overeenkomen.
- **Woordenboekoperaties**
  1. **Zoeken**
     - Berekenen van de hashwaarde, de pagina inlezen en de sleutels sequentieel overlopen
  2. **Toevoegen**
     - Analoog zolang dat de pagina niet vol zit, anders splitsen
     - Alle hashwaarden in de pagina eindigen met dezelfde k bits, dus men splitst men de elementen volgens $k + 1$
     - Als k kleiner was dan d, moet de helft van de wijzers naar de oude pagina verwijzen naar de nieuwe
     - Als $k = d$ moet d 1 groter worden en verdubbelt de grootte van de hashtabel



     - Problemen: als ze niet te onderscheiden zijn adhv k-de bit, moeten we verder splitsen. Als dezelfde hashwaarden veel voorkomen kan de hashtabel dus oneindig groot worden. De hashfunctie moet dus goed gekozen worden.
  3. **Verwijderen**
     - Wanneer een pagina en haar ooit afgesplitse buur samen minder dan m sleutels bevatten, kunnen ze worden samengevoegd.
     - Dan verdwijnt er een pagina en haar hashwaarde moer naar de andere pagina wijzen
     - Als de hashtabel minstens 2 verwijzingen naar elke pagina bevat, kan d gehalveerd worden
- Als de hashtabel te groot wordt voor het inwendig geheugen zijn er andere mogelijkheden
  1. De hashtabel kan in 2 niveau's geïmplementeerd worden. De wortel wordt in het geheugen gehouden, en indexeren hem met een kleiner aantal eindbits die verwijzen naar hashtabelen in het geheugen. Zoeken vereist dan een extra schijf operatie.
  2. Het maximaal aantal gegevens m in elke pagina vergroten door enkel sleutels met wijzers naar gegevens er in op te slaan idpv de gegevens zelf. Er is opnieuw een extra schijfoperatie nodig.

3.2.2 Lineair hashing

Heeft geen hashtabel meer nodig doordat het pagina's met opeenvolgende adressen gebruikt. De d eindbits worden rechtstreeks als adres van de pagina gebruikt. We moeten dus 2^d overeenkomstige pagina's hebben. We gaan anders tewerk, anders moet het aantal pagina's verdubbelen als we $d+1$ moeten doen.

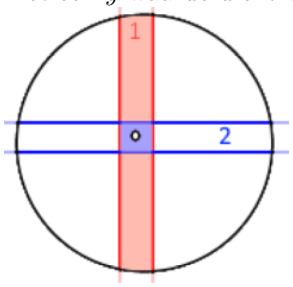
- Wanneer een pagina vol is, wordt er gesplitst, maar niet noodzakelijk de volle pagina. Een loper p houdt bij welke pagina mag gesplitst worden, want dit gebeurt in volgorde. De effectieve volle pagina zet gegevens over in een *overflow pagina* tot de p bij hem is en hij kan gaan splitsen.
- **Woordenboekoperaties**
  1. **Zoeken**
     - We moeten weten hoeveel eindbits van de hashwaarde nodig zijn om de pagina te adresseren
     - Het adres dat gevormd wordt door de d eindbits wordt vergeleken met p: als het kleiner is dan p, dan is de pagina al gesplitst en moet met $d+1$ bits gebruiken
     - Dan wordt er lineair of binair gezocht, desnoods tot in de overflow pagina
     - Dit vereist normaal 1 schijfoperatie en 2 indien er de gezochte sleutel in de overflowpagina zitten
  2. **Toevoegen**
     - Lokaliseren zoals bij zoeken en tussenvoegen indien hij niet vol zit
     - Zit deze vol? Dan moet er gesplitst worden op pagina p
     - Als de volle pagina niet gesplitst werd, wordt het gegeven toegevoegd aan een overflowpagina
  3. **Verwijderen**
     - Lokaliseren zoals bij zoeken en er uit halen.
     - Onbezette pagina's verdwijnen in omgekeerde volgorde als waarin ze gecrëeerd werden. (maar enkel indien hun gezamelijk aantal gegevens kleiner is dan m)
     - De gegevens van zo'n verwijderde pagina komen terecht in de pagina waarvan ze origineel afgesplitst werden

4 Meerdimensionale gegevensstructuren

Probleem: We hebben coördinaten in een meerdimensionale ruimte. We zoeken gegevensstructuren om efficiënt te gaan zoeken welke punten dicht bij elkaar liggen OF welke punten liggen in het gebied van een bepaald punt?
Er bestaan simpele gegevensstructuren (projectie, raster) en bomen (quad, kd) om dit probleem aan te pakken.
(Ik geef altijd een korte uitleg van in de les met het aantal dimensies $k = 2$ ter illustratie en daarna de theorie van in de cursus)

4.1 Projectie

Les: ik heb een punt (x, y). Ik kies bijvoorbeeld x als mijn belangrijke as en ga op zoek naar alle punten die dicht bij deze x as liggen. Daarna zoek ik in die verzameling van punten alle punten met een y waarde dicht bij ons punt.

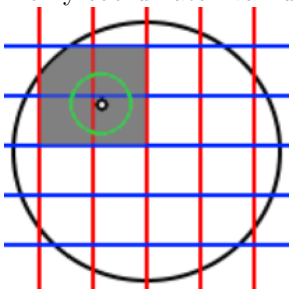


Theorie:

- Per dimensie wordt er een gegevensstructuur bijgehoude die alle punten gerangschikt bijhoudt volgens die dimensie = *projectie op elke dimensie*
- Zoeken gebeurt door een dimensie te kiezen en in die overeenkomstige gesorteerde gegevensstructuur alle punten zoeken die er dichtbij liggen
- Deze punten worden dan sequentieel overlopen om te testen of ze in het gebied liggen
- Als de punten gelijkmatig verdeeld zijn, hebben we maar 1 dimensie gegevensstructuur nodig met 1 dimensie in volgorde, en gebruiken we altijd deze
- Gemiddelde performantie: $O(n^{1-1/k})$

4.2 Rasterstructuur

Les: het gebied is opgedeeld in rasters. Ik maak een omtrek rond mijn punt, en bekijk alle rasters waar mijn omtrek in voorkomt.
Indien de punten homogeen verdeeld zijn: homogene rasterstructuur voorgesteld als een 2D tabel met gelinkte lijstjes die alle punten van dat gebied bevatten
Indien de verdeling ongelijk is, kunnen we het raster ook ongelijk verdelen, en dan moeten we de x en y coördinaten van de raster-verdeling in volgorde bijhouden



Theorie:

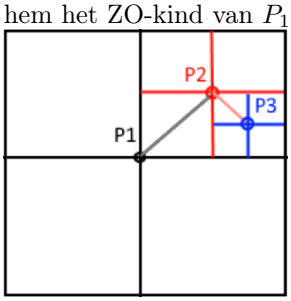
- Voor elk van de rastergebieden wordt er een gelinkte lijst van punt die erin liggen bijgehouden
- Alle punten in een gebied opzoeken komt neer op het vinden van de rastergebieden die dit gebied overlappen, en nagaan welke punten in die rastergebieden ons gebied overlappen
- Er moet een compromis gezocht worden tussen de grootte van rastergebieden en de lengte van hun gelinkte lijsten.
- Dit veronderstelt statische gegevens. In geval van dynamische gegevens moet men ledige of overvolle gebieden tegengaan.
  - Ofwel een andere gegevensstructuur met enkel niet-ledige gebieden
  - Adaptief, zodat gebieden kunnen splitsen (= basis van quadtrees)

4.3 Quadtrees

Verdelen de zoekruimte in 2^k (hyper)rechthoeken, waarvan de zijden evenwijdig met het assenstelsel. Dit wordt opgeslagen in een 2^k-meerwegsboom. Dit is niet geschikt voor veel dimensies (want heel veel kinderen) dus we houden het bij 2D.

4.3.1 Point quad trees

Les: oorspronkelijk hebben we een lege boom. Het eerste punt dat we toevoegen is de wortel (stel $P_0(0,0)$).
Het tweede punt is $P_1(2,2)$, hiervan is de x waarde groter dan P_0 en de y waarde ook. We zetten P_1 dus in het NO gebied.
Het derde punt is $P_2(3,1)$. Dit zou ook in het NO gebied van P_0 moeten, maar hier staat P_1 al dus we gaan verder richting P_1. P_2 moet ten ZO van P_1 komen, hier staat nog niets, dus we maken hem het ZO-kind van P_1.

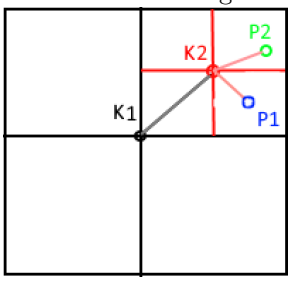


Theorie:

- Elke inwendige knoop bevat een punt, waarvan de coördinaten de (deel)zoekruimte opdelen in 4 rechthoeken, die dus niet noodzakelijk even groot zijn
- De vorm is afhankelijk van de toevoegvolgorde. Als elke toevoegvolgorde even waarschijnlijk is, is zoeken en toevoegen $O(\lg n)$ maar in het slechtste geval $O(n)$
- Indien we de punten op voorhand weten kunnen we ervoor zorgen dat geen enkele deelboom meer dan de helft van de punten bevat van de punten die zijn ouder bevat. Dit door de punten bv te sorteren op x, de mediaan te nemen, en deze als volgend punt toevoegen. En dit voor steeds kleinere intervallen van punten.
  - Hierdoor wordt de hoogte $O(\lg n)$ en de constructie $O(n \lg n)$
- **Zoeken:** gebeurt recursief
- **Verwijderen:** best dat punt verwijderen en al zijn kinderen opnieuw toevoegen op de methode dat we net besproken hebben

4.3.2 PR quadtrees

Les: dit is bijna hetzelfde maar nu bevatten inwedige knopen enkel een verdeling van de regio, en zitten effectieve punten in de bladeren. Elke inwendige knoop verdeelt het (deel)gebied in 4 gelijke delen.
We beginnen met een wortel toe te voegen die ons gebied $x, y \in [-4, 4]$ op plaats $K_1(0, 0)$ die ons gebied in 4 gelijke delen verdeelt.
We voegen een punt $P_1(3, 1)$ toe, dit komt in het NO-gebied van K_1.
We voegen een tweede punt $P_2(3.5, 2.5)$ toe. Dit zou ook in het NO-gebied van K_1 moeten komen maar hier staat P_1 al. We voegen dus aan K_1 in het NO-gebied een nieuwe inwedige knoop toe, die het deelgebied in 4 gelijke delen verdeelt, dus $K_2(2, 2)$.
We kijken nu of P_1 en P_2 in hetzelfde deelgebied van k_2 terechtkomen, dit is niet het geval, dus kunnen we ze gewoon aan K_2 toevoegen als NO- en ZO-kinderen
Indien dit wel het geval was, moesten we verder splitsen

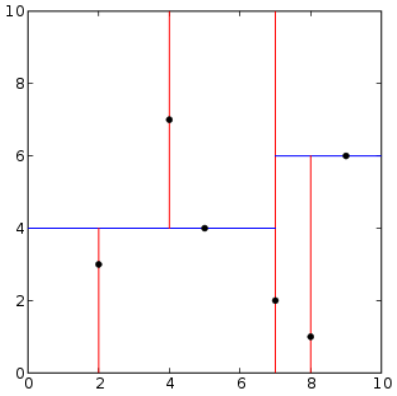


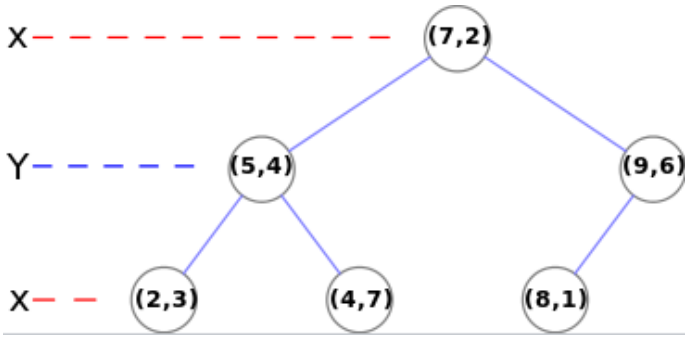
Theorie:

- In tegenstelling tot point quadtrees vereist deze boom dat de zoekruimte een rechthoek is. Elke knoop verdeelt een deel van die ruimte in 4 gelijke delen. Deze opdeling gaat door tot elk deel nog 1 punt bevat.
- **Zoeken en toevoegen:** vanuit de wortel zoeken, indien het gevonden punt verschilt en we willen ons punt toevoegen, dan moet het gebied opgesplitst worden tot elk van de punten in een eigen gebied ligt
- De vorm is onafhankelijk van de toevoegvolgorde maar kan wel onevenwichtig uitvallen

4.3.3 k-d trees

- Vermijdt de grote vertakkingsgraad van een point quadtree door een binaire boom te gebruiken.
- Elke inwendige knoop bevat een punt dat de zoekruimte verdeelt in slechts 1 dimensie
- Opeenvolgende knopen kunnen opeenvolgende dimensies gebruiken om te splitsen zodat de zoekruimte in k-dimensionale rechthoeken verdeeld wordt





- De opdeling kan door gaan tot elk gebied 1 punt bevat, of vroeger stoppen en per gebied een gelinkte lijst bijhouden
- Bij een statisch geval (we kennen de punten op voorhand) dan kunnen we een een boom van hoogte $O(\lg n)$ opbouwen in $O(kn \lg n)$ tijd.
- Bij dynamisch toevoegen, als elke toevoegvolgorde even waarschijnlijk is, heeft de boom een *gemiddelde* hoogte van $O(\lg n)$
- Er bestaan geen rotaties om de boom evenwichtiger te maken, het beste dat men kan doen is af en toe de boom reconstrueren.
- Een punt zoeken of toevoegen in een boom met hoogte h is $O(\lg h)$

5 Prioriteitswachtrijen

5.1 Samenvoegbare heaps: een overzicht

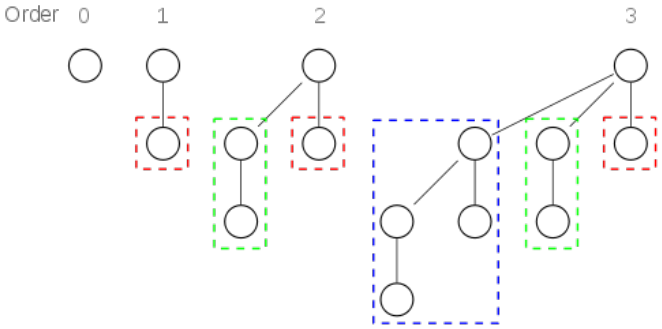
Sommige algoritmen vereisen nog een operatie op prioriteitswachtrijen: het samenvoegen van 2 prioriteitswachtrijen.

5.2 Binomial queues

5.2.1 Structuur

Bestaat uit een bos van binomiaalbomen, elk met de heapvoorwaarde. Een binomiaalboom wordt gekenmerkt door zijn hoogte

- Er is slechts 1 binomiaalboom met hoogte h mogelijk
- B_0 bestaat uit 1 knoop
- B_h bestaat uit 2 B_{h-1} bomen
- B_h bestaat dus uit een wortel met als kinderen (van links naar rechts) $B_0, B_1, ...B_{h-1}$



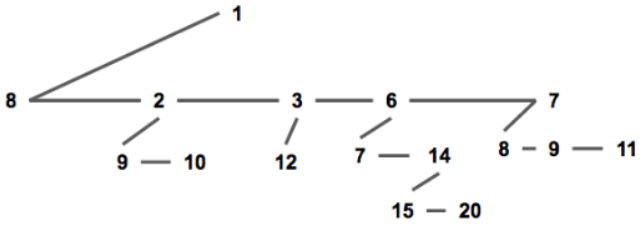
5.2.2 Operaties

- **Minimum vinden**: alle wortels van de bomen overlopen, er zijn minstens $\lg n$ bomen dus dit is $O(\lg n)$ (of $O(1)$ als we de plaats ervan bijhouden)
- **Samenvoegen** telt bomen met dezelfde hoogte bij elkaar op. Twee bomen B_h worden opgeteld door de wortel met de grootste sleutel kind te maken van de kleinste. Samenvoegen van binomiale bomen is $O(1)$ en van binomiale queues is dit dan $O(\lg n)$
- **Toevoegen** is een speciaal geval van samenvoegen: de knoop met het toe te voegen element wordt een triviale binomial queue
- **Minimum verwijderen**: binomiaalboom met de kleinste wortel zoeken en de wortel verwijderen en dit levert een nieuwe binomial queue op. De performantie is $O(\lg n)$
- Bijkomende operaties die men kan ondersteunen: prioriteit verminderen door naar de wortel toe te bewegen, verwijderen is prioriteit verminderen en minimum verwijderen

Implementatie: Voorgesteld als een tabel van binomiaalbomen. Elke knoop bevat een wijzer naar zijn linkerkind en een naar zijn rechterbroer

5.3 Pairing heaps

Elke knoop bevat een wijzer naar zijn linkerkind en een naar zijn rechterbroer. Wanneer verminderen van de prioriteit moet ondersteund worden, heeft elke knoop een extra wijzer nodig: een linkerkind naar zijn ouder, een rechterbroer naar zijn linkerbroer.



- **Samenvoegen:** Wortel met het grootste element wordt linkerkind van deze met het kleinste element
- **Toevoegen:** speciaal geval van samenvoegen
- **Prioriteit van een knoop verminderen**: de knoop wordt losgekoppeld van zijn ouder, zodat hij de wortel wordt van een nieuwe heap en dan samenvoegen bij de oorspronkelijke heap
- **Minimum verwijderen:** door de wortel te verwijderen, wat c heaps oplevert en dan $c - 1$ keer samenvoegen.
- **Willekeurige knoop verwijderen:** de knoop loskoppelen van zijn ouder, dit geeft een aparte heap, hiervan het minimum verwijderen en dan de heaps samenvoegen met de oorspronkelijke heap

6 Toepassingen van diepte eerst zoeken

6.1 Enkelvoudige samenhang in grafen

6.1.1 Samenhangende componenten van een ongerichte graaf

Een ongerichte graaf heet *samenhangend*, wanneer er een weg bestaat tussen elk paar knopen. Diepte eerst zoeken vindt alle knopen die met de wortel van de diepte-eerst boom verbonden zijn. De graaf is dus samenhangend wanneer de diepte eerst boom alle knopen bevat. Testen of een graaf samenhangend is of de samenhangende componenten zoeken is $\Theta(n+m)$ (ijle graaf) of $\Theta(n^2)$ (dichte graaf)

6.1.2 Sterk samenhangend

Een gerichte graaf met een weg tussen elk paar knopen, in *beide* richtingen, noemt men *sterk samenhangend*.
Sterk samenhangende componenten kunnen opnieuw gevonden worden met diepte-eerst zoeken

1. Door eerst de *omgekeerde graaf* opstellen door de richting van elke verbinding om te draaien
2. Op deze graaf diepte-eerst zoeken toepassen en de knopen in postorder nummeren
3. Dan diepte eerst zoeken op de oorspronkelijke graaf, waarbij men als startknoop steeds de resterende knoop met hoogste postordernummer neemt.

elk paar knopen is in beide richtingen verbonden met elkaar via de wortel

- Via de boomtakken is er zeker een weg w naar elk van de knopen u in de boom, en dus is er een weg van u naar w in de omgekeerde graaf.
- w is steeds een voorrouder van u in een diepte-eerst boom van de omgekeerde graaf (de twee knopen kunnen immers niet tot verschillende bomen behoren)
- Daaruit volgt dat er een weg van w naar u bestaan in de omgekeerde graaf, via de boomtakken zodat de knopen in beide richtingen verbonden zijn.

Twee keer diepte eerst zoeken is $O(n+m)$ bij ijle en $O(n^2)$ bij dichte grafen. Het omkeren wijzigt dit gedrag niet.

6.2 Dubbele samenhang van ongerichte grafen

Er zijn 2 definities van dubbele samenhang:

- **Dubbel lijnsamenhangend:** een *verbinding* die een ongerichte graaf doet splitsen in 2 delen wanneer ze wordt weggenomen = *brug*. Een graaf zonder bruggen is *dubbel lijnsamenhangend*. Tussen elk paar knopen van een dubbel lijnsamenhangende graaf bestaan er minstens 2 wegen zonder gemeenschappelijke verbindingen.
- **Dubbel samenhangend:** een knoop die een graaf doet uiteenvallen in *minstens* 2 delen wanneer hij wordt weggenomen = *scharnietpunt*. Een graaf zonder scharnierpunten is *dubbel samenhangend*. Tussen elk paar knopen van een dubbel samenhangende graaf bestaan er minstens 2 wegen zonder gemeenschappelijke *knopen*.

Al deze componenten kunnen opnieuw gevonden worden via diepte-eerst zoeken:

1. Eerst de diepte-eerst boom opstellen en de knopen in *preorder* nummeren
2. Voor elke knoop u wordt de laagst genummerde knoop die vanuit u bereikt kan worden via een weg bestaande uit nul of meer dalende boomtakken gevolgd door 1 terugverbinding bepaald. (tijdens een 2de postorder doorgang van de boom)

Als alle kinderen van een knoop op die manier een knoop kunnen bereiken die hoger in de boom ligt dan hemzelf, dan is hij zeker geen scharnierpunt. De **wortel** is een scharnierpunt wanneer hij meer dan 1 kind in de boom heeft.
(codevoorbeeld scharnierpunten zoeken pagina 64)
Dit algoritme is in essentie diepte-eerst zoeken, zodat de uitvoeringstijd $O(n+m)$ of $O(n^2)$ bedraagt.

6.3 Eulercircuits

Een Eulercircuit is een gesloten omloop in een (multi)graaf die alle *verbindingen* exact eenmaal bevat.

6.3.1 Ongerichte grafen

Een **Eulergraaf** is een (multi)graaf met een Eulercircuit. De volgende 3 eigenschappen zijn equivalent:

1. Een samenhangende (multi)graaf G is een Eulergraaf
2. De graad van elke knoop van G is even ***(//Dit is toch de graad van elke knoop in het circuit en niet de ganse graaf?)-> In een Eulergraaf behoort elke knoop tot het Eulercircuit.***
3. De verbindingen van G kunnen onderverdeeld worden in lussen

Eigenschap 2 volgt uit 1: telkens een knoop k voorkomt in een Eulercircuit draagt hij 2 bij tot zijn graad en elke verbinding komt precies eenmaal voor op dat circuit.
Eigenschap 3 volgt uit 2:

- Stel dat G n knopen heeft. Er zijn dan minstens $n-1$ verbindingen, want G is samenhangend
- Geen enkel knoopgraad is 1 zodat er minstens n verbindingen zijn
- Dus G bevat minstens 1 lus
- Elke samenhangende component van G kan op deze manier verdeeld worden in lussen

Eigenschap 1 volgt uit 3:

- Stel dat L 1 van de lussen uit G is
- Als L een Eulercircuit is dan is G een Eulergraaf
- Zoniet bestaat er een andere lus L' die een gemeenschappelijke knoop k heeft met L
- Lus L' kan bij knoop k tussengevoegd worden in lus L zodat een grotere lus ontstaat
- Als we dit proces herhalen komen we aan een Eulercircuit aangezien elke verbinding tot 1 lus behoort

Constructie van een Eulercircuit:

- Om de eerste lus te vinden: bij willekeurige knoop beginnen en verder tot we weer in die knoop zitten
- De volgende lus begint bij een knoop van L waarvan nog niet alle verbindingen doorlopen zijn
- We testen best in volgorde van langste lus

6.3.2 Gerichte grafen

Exact hetzelfde maar dan met ingraad van elke knoop is gelijk aan de uitgraad

7 Kortste afstanden

7.1 Kortste afstanden vanuit één knoop

In algo 1 zagen we Dijkstra, maar daar konden we geen negatieve gewichten gebruiken. Nu zien we het algoritme van Bellman-Ford die dit wel toelaat.
We onderstellen de grafen *gericht*

7.1.1 Het algoritme van Bellman-Ford

= een algoritme dat dynamisch programmeren gebruikt en kan beschouwd worden als een combinatie van breedte-eerst zoeken met kortste afstanden.
Er mogen *geen* negatieve *lussen* zijn, de kortste wegen hebben *geen* lussen en er zijn hoogstens $n-1$ verbindingen.

- $d_i(k)$ is het gewicht van de kortste weg met hoogstens k verbindingen vanuit de startknoop naar knoop i, en g_{ji} is het gewicht van de verbinding (j,i) dan

$$d_i(k) = \min(d_i(k-1), \min_{j \in V}(d_j(k-1) + g_{ji}))$$

- De gezochte kortste afstanden zijn de waarden $d_i(n-1)$ voor elkle knoop i

Dit kan dus opgelost worden via dynamisch programmeren.

- optimale deelstructuur.
- deeloplossingen zijn onafhankelijk.
- deeloplossingen zijn overlappend.

Het bestaat uit $n-1$ iteraties, waarbij de kortste weg telkens 1 verbinding langer mag worden
De voorlopige kortste afstanden worden in een tabel bijgehouden en hun voorlopige voorloper

Elke iteratie moet alle m verbindingen testen dus de performantie is $O(nm)$

Er zijn 2 goede implementaties:

1. **Manier 1:**
   - Enkel de knopen waarvan de afstand door de huidige iteratie gewijzigd werd worden in een wachtrij geplaatst, en bij de volgende iteratie worden enkel de buren van deze knopen getest
   - We moet dus niet nagaan of de afstand waarvan een knoop buur is verbeterd werd (*pull based*) maar eerder verwittigd worden wanneer een buur gewijzigd is (*push based*)
   - Als een knoop in de wachtrij moet worden gezet en hij is er al uit gehaald: gewoon terug in zetten, als hij er nog in zit: gewoon in laten zitten
   - **Beveiligen tegen negatieve lussen:** aantal iteraties bijhouden, en als de wachtrij leeg is na n iteraties, heeft de graaf geen negatieve lus
2. **Manier 2:**
   - In de plaats van in een wachtrij worden de nog te onderzoeken knopen in een *deque* bijgehouden: er uit nemen gebeurt vooraan, toevoegen gebeurt aan beide kanten
   - Als de knoop die werd aangepast al vroeger in de deque zat, voegt men hem vooraan toe, anders achteraan
   - Logica hierachter: als knoop i in de deque zat en werd verwijderd zijn daarbij al zijn buren onderzocht, dan is de kans groot dat zijn buren onderzocht werden zodat sommigen daarvan ook in de deque kwamen. Als de afstand van de huidige knoop nog eens wijzigt, is het best dat dit gebeurt voordat we naar de buren gaan, want dit kan hun afstand later opnieuw wijzigen

Wordt gebruikt in netwerken om de kortste afstand te zoeken en elke knoop moet maar 1 voorloper/opvolger kennen: *distance vector protocol* (*shout out to ma boi Joris*)

7.2 Kortste afstanden tussen alle knopenparen

Voor *ijle* grafen met negatieve gewichten kan men beter het algoritme van Johnson gebruiken met performantie $O(nm \lg n)$ (idpv Floyd-Warshall, dat trouwens ook negatieve gewichten toelaat maar $\Theta(n^3)$ is)

7.2.1 Het algoritme van Johnson

We gebruiken Dijkstra en Bellman-Ford. We geven alle verbindingen een nieuw positief gewicht om Dijkstra te kunnen gebruiken

- We breiden de graaf uit met s en geven die verbindingen met gewicht 0 naar alle andere knopen
- We kunnen dan het algoritme van Bellman-Ford toepassen op de nieuwe graaf om vanuit s de kortste afstand d_i te bepalen tot elke originele knoop i
- Deze afstanden gebruiken we dan om de nieuwe gewichten te bepalen
  - g_{ij} is het oorspronkelijk gewicht van verbinding (i, j)
  - Het nieuw gewicht wordt dan $\hat{g_{ij}} = g_{ij} + d_i - d_j$
  - Aangezien d_i en d_j kortste afstanden izjn, geldt $d_j \leq d_i + g_{ij}$ dus $\hat{g_{ij}}$ is nooit negatief
- Zijn de kortste wegen met de nieuwe gewichten hetzelfde als bij de originele?
  - w_{ij} was het originele gewicht van de *kortste weg* tussen i en j en $\hat{w_{ij}}$ is het nieuwe gewicht van diezelfde weg
  - Als er met de nieuwe gewichten een kortere weg zou bestaan $\hat{w_{ij}}' < \hat{wij}$
  - Dan zou $w'_{ij} + d_i - d_j < w_{ij} + d_i - d_j$
  - en bestaat er dus een kleinere weg w'_{ij} $(w'_{ij} < w_{ij})$
  - wat zou betekenen dat de eerste weg (met originele gewichten) toch niet de kortste was.
- Dijkstra geeft dan de kortste wegen weer
- **Performantie:**
  - graaf uitbreiden $\Theta(n)$
  - Bellman-Ford $O(nm)$
  - Gewichten aanpassen $\Theta(m)$
  - n maal Dijkstra $O(n(n+m) \lg n)$
  - Deze laatste is de belangrijkste waardoor deze methode $O(n(n+m) \lg n)$ wordt, te vereenvoudigen tot $O(nm \lg n)$

7.3 Transitieve sluiting

- *sluiting* = algemene methode om een of meerdere verzamelingen op te bouwen
- Herhaaldelijk "als een verzameling deze gegevens bevat, dan moet ze ook de volgende gegevens bevatten" tot er niets meer kan toegevoegd worden
- Bijkomende voorwaarde: er mag niets anders in dan wat de regels vermelden
- *Transitieve sluiten* is een geval waarbij de regels van de vorm 'als (a,b) en (b,c) aanwezig zijn, dan moet ook (a,c) aanwezig zijn' zijn
- De transitieve sluiting van een gerichte graaf is een nieuwe gerichte graaf met dezelfde knopen, en een verbinding van knoop i naar knoop j als er een weg van i naar j bestaat in de oorspronkelijke graaf.
- Er zijn 3 manieren om de transitieve sluiting t bepalen:
  - Herhaaldelijk diepte of breedte eerst zoeken met elke knoop als startknoop. (vooral voor ijle grafen)
  - Als men verwacht dat de transitieve sluiting een dichte graaf zal zijn: De componentengraaf wordt opgesteld en het probleem herleid zich dan. Als nu blijkt dat component j bereikbaar is vanuit component i, dan zijn alle knopen van j bereikbaar vanuit alle knopen van i
  - Als de graaf zelf dicht is: de methode van **Warshall** $\cong$ Floyd-Warshall
    - Een reeks opeenvolgende $n \times n$ matrices $T^0, T^1, , T^n$ die nu logische waarde bevatten
    - Element $t_{ij}^{(k)}$ duidt aan of er een weg bestaat tussen i en j met als *mogelijke* intermediaire knopen $1, 2, ..., k$
    - Verbinding (i, j) behoort dan tot de transitieve sluiten als $t_{ij}^{(n)}$ waar is
    - De opeenvolgende matrices worden als volgt bepaald:

$$t_{ij}^{(0)} = \ false \ \forall \ i \neq j \ en \ g_{ij} = \infty$$

$$t_{ij}^{(0)} = \ true \ \forall \ i = j \ en \ g_{ij} < \infty$$

      En

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} \right) \ \forall \ 1 \leq k \leq n$$

    - De efficiëntie is $\Theta(n^3)$ en de logische waarden vereisen minder plaats dan getallen.

8 Stroomnetwerken

Een *stroomnetwerk* is een gerichte graaf met twee speciale knopen, een *producent* en een *verbruiker*. 'Materiaal' stroomt van de producent naar verbruiker. De graaf mag lussen bevatten.

Elke gerichte verbinding bevat een *capaciteit*, die het grootste debiet voorstelt dat er doorheen kan stromen. Knopen kunnen geen materiaal opslaan dus alles dat er in stroomt moet er weer uit stromen (*conservatief*).

8.1 Maximalestroomprobleem

= Zo veel mogelijk materiaal van P naar V laten stromen zonder de capaciteit van de verbindingen te overschrijden. De methode (niet algoritme) om dit op te lossen is die van Ford-Fulkerson, en deze laat meerdere implementaties toe.

- $c(i, j)$ is de positieve capaciteit van elke verbinding
- $s(i, j)$ de stroom die loopt op een verbinding (beperkt door de capaciteit $0 \leq s(i, j) \leq c(i, j)$)
- K is de verzameling knopen, dan is de totale nettostroom f uit de producent p (niet bestaande verbindingen krijgen capaciteit 0)

$$f = \sum_{j \in K} (s(p, j) - s(j, p))$$

- Deze netwerkstroom moet maximaal worden
- **Ford-Fulkerson:**
  - Begint met een stroomverdeling die overal nul is
  - Tijdens elke iteratie wordt de huidige stroomverdeling en de capaciteiten bekeken om een mogelijke stroomtoename te zoeken
  - In dit overzicht, *het restnetwerk*, wordt gezocht naar een weg van producent naar verbruiker die stroom tussen beiden toelaat, de *vergrotende weg*
  - Met die stroom wordt de stroomverdeling in het oorspronkelijk netwerk aangepast
- **Het restnetwerk:**
  - Het restnetwerk is ook een stroomnetwerk met dezelfde knopen maar enkel met de verbindingen die meer stroom toelaten dan het voorlopige gevonden stroomnetwerk
  - Dit betekent dat de capaciteit niet volledig gebruikt werd $s(i, j) < c(i, j)$ OF er stroom loopt van j naar i die kleiner kan gemaakt worden
  - Verbinding (i, j) krijgt een restcapaciteit $c_r(i, j) = c(i, j) - s(i, j) + s(j, i)$
  - De verbindingen van het restnetwerk vormen niet noodzakelijk een deelverzameling van de originele verbinding ($c(i, j) = 0$ maar $c(j, i) > 0$)
- **De vergrotende weg**
  - Een vergrotende (dus positieve restcapaciteit) enkelvoudige (zonder lussen) weg van producent naar verbruiker
  - Tussen p en v is extra stroom mogelijk, gelijk aan de kleinste restcapaciteit op die weg
  - Een stroomtoename kunnen we dan toepassen door een toename van $s(i, j)$ of afname van $s(j, i)$

- **Aantonen dat de methode van Ford-Fulkerson correct is**
  - **Gebruikte terminologie**
    - Een *snede* (P, V) is de verzameling verbindingen dat de graaf in 2 niet-ledige stukken P en V verdeelt
    - Bij gebruiken snedes waarbij $p \in P$ en $v \in V$
    - De capaciteit $c(P, V) = \sum c(i, j)\ \forall\ i \in P, j \in V$
    - De nettostroom $f(P, V) = \sum s(i, j) - s(j, i)\ \forall\ i \in P, j \in V$
    - Aangezien $s(i, j) \leq c(i, j)$ is $f(P, V) \leq c(P, V)$
  - **Uit de *conservatieve eigenschap* volgt dat de netwerkstroom f gelijk is aan de nettostroom $f(P, V)$ van *elke mogelijke snede*. We tonen dit aan:**
    - De stroom van het netwerk is de nettostroom vanuit p:
    $$f = \sum_{j \in K} (s(p, j) - s(j, p))$$
    - In alle andere knope i van P is de stroom conservatief:
    $$\sum_{j \in K} (s(i, j) - s(j, i)) = 0$$
    - Als we de som maken hiervan voor alle knopen van P dan
    $$f = \sum_{i \in P} \sum_{j \in K} (s(i, j) - s(j, i))$$
    - Voor alle knopen j uit P komt elke stroom $s(i, j)$ tweemaal voor in deze dubbele som, met tegengesteld teken, enkel knopen j uit $V = L$
    P blijven over, wat precies de nettostroom van snede (P, V) is:
    $$f = \sum_{i \in P} \sum_{j \in K} (s(i, j) - s(j, i)) = f(P, V)$$
  - **De maximale netwerkstroom wordt bereikt als het overeenkomstige netwerk geen vergrotende weg meer heeft, de volgende eigenschappen zijn immers equivalent**
    1. De netwerkstroom f is maximaal
    2. Er is geen vergrotende weg meer te vinden in het restnetwerk
    3. De netwerkstroom f is gelijk aan de capaciteit van *een* snede in de oorspronkelijke graaf
  - **Verklaring van de eigenschappen**
    - **2 volgt uit 1:** als er nog een vergrotende weg overblijft kan f zeker groter gemaakt worden
    - **3 volgt uit 2:**
    Beswhouw snede (P, V) waarvan P alle knopen bevat die in het restnetwerk bereikbaar zijn vanuit p.
    Er is geen vergrotende weg dus v behoort zeker niet tot P, de restcapaciteit tussen (P, V) moet dus 0 zijn
    In het stroomnetwerk moet dan $f(P, V) = c(P, V)$ want als
    $$\sum_{i \in P} \sum_{j \in V} c_r(i, j) = \sum_{i \in P} \sum_{j \in V} (c(i, j) - s(i, j) + s(j, i)) = 0$$
    Dan is
    $$c(P, V) = \sum_{i \in P} \sum_{j \in V} c(i, j) = \sum_{i \in P} \sum_{j \in V} (s(i, j) - s(j, i)) = f(P, V)$$
    - **3 volgt uit 1:** want we weten al dat de netwerkstroom gelijk is aan de nettostroom doorheen elke snede, en de nettostroom van de snede hierboven heeft zijn maximum bereikt. Overigens moet dat de snede zijn met de kleinste capaciteit, vandaar de naam van deze stelling.

- **Verschillende mogelijke implementaties**
  - **Performantie afhankelijk van de capaciteiten**
    - Stel C de grootste capaciteit en capaciteiten zijn gehele getallen. De maximale netwerkstroom is dan $O(nC)$. Bij Ford-Fulkerson is het **aantal iteraties** dan $O(nC)$. Een **restnetwerk bepalen** is $O(m)$ en daarin de **vergrotende weg zoeken** (met diepte of breedte eerst zoeken) is $O(m)$. De **totale performantie** wordt dus $O(nmC)$
    - Als men steeds de vergrotende weg neemt die de grootste stroomtoename mogelijk maakt, zijn er $O(m \lg C)$ **iteraties** nodig. De **iteratiestap** (+- algoritme van Dijkstra) wordt $O(m \lg n)$. De totale performantie wordt $O(m^2 \lg n \lg C)$
    - Snellere variant: een **vergrotende weg vinden** die een stroomtoename van minstens c eenheden toelaat kan in $O(m)$. Als er geen meer gevonden wordt is de minimale snedecapaciteit van het restnetwerk lager dan mc. Dan wordt de ondergrens gehalveerd enzoverder. Als we beginnen bij $c = 2^{\lfloor \lg C \rfloor}$ dan wordt de **maximale stroom bereikt met** $O(m \lg C)$ **iteraties**.
  - **Performantie onafhankelijk van de capaciteit**
    - Als de vergrotende weg steeds het minimum aantal verbindingen heeft, stijgt de lengte van de vergrotende weg na hoogstens m iteraties en aangezien de maximale lengte $n-1$ is, **volstaan** $O(nm)$ **iteraties**. De iteratiestap gebruikt **breedte-eerst** zoeken en is dus $O(m)$. De **totale performantie** wordt $O(nm^2)$
  - **Preflow-push methode**
    - Vindt de maximale stroom door stroomtoename langs *individuele verbindingen*, idpv langs een vergrotende weg.
    - Als gevolg is de stroom niet altijd conservatief tijdens het uitvoeren van het algoritme:
    - Knopen met een positieve overschot van stroom heten *actief*, en zolang er zo'n knopen bestaan, voldoet de oplossing niet
    - De basisoperatie is een actieve knoop selecteren en trachten om zijn overschot weg te werken via zijn buren
    - Als er geen actieve knopen meer zijn, voldoet de stroom aan de vereisten en blijkt die bovendien maximaal

8.2 Verwante problemen

- **Stroomnetwerk met meerdere producenten en verbruikers**
  - We voeren een fictieve *totaalproducent* en *totaalverbruiker* in
  - Vanuit de totaalproducent komen er verbindingen met onbeperkte capaciteit naar de producenten, verbruiker analoog.
  - De maximale stroomverdeling in dit nieuw netwerk komt dan overeen met die in het originele netwerk
- **Knopen die ook een capaciteitsbeperking hebben**
  - De knopen worden ontdubbeld in 2 knoppen en een verbinding met de knoopcapaciteit wordt voorzien tussen die 2 knopen
- **Bij een *ongericht* stroomnetwerk**
  - Elke verbinding wordt vervangen door een paar gerichte verbindingen, één in elke richting, en beide verbindingen krijgen de originele capaciteit
- **Verbindingen hebben ook ondergrenzen**
  - 2 fasen: eerst onderzoeken of dit wel mogelijk is en daarna transformeren in een maximalestroomprobleem.
- **Meerdere soorten materiaal mogelijk**
  - Voor elke soort is er 1 producent en 1 verbruiker, en in elke knoop is de stroom voor elke soort apart conservatief.
- **Minimale kost probleem**
  - Als er aan elke stroom ook een kost per stroomeenheid wordt gegeven
  - Dit zoekt dus de maximale stroom met de minimale kost

8.2.1 Meervoudige samenhang in grafen

Vormen van samenhang:

- *k-voudig samenhangend:* als er tussen elk paar knopen k onafhankelijke wegen bestaan, zonder gemeenschappelijke *knopen*
- *k-voudig lijnsamenhangend:* als er tussen elk paar knopen k onafhankelijke wegen bestaan, zonder gemeenschappelijke *verbindingen*

Om meervoudige samenhang op te sporen maken we gebruik van stroomnetwerken: als alle capaciteiten 1 zijn, dan is de capaciteit van een snede (P, V) gelijk aan het aantal verbindingen van knopen in P naar knopen in V. Het aantal onafhankelijke wegen is gelijk aan de minimale snedecapaciteit, en dus gelijk aan de maximale netwerkstroom. Bij netwerken met $C = 1$ kan dit gevonden worden in $O(nm)$.

Stelling van Menger (4 versies hiervan bestaan: gericht en ongericht, meervoudig samenhangend en meervoudig lijnsamenhangend):

Het minimum aantal verbindingen dat moet verwijderd worden om een knoop v van een gerichte graaf onbereikbaar te maken vanuit een andere knoop p is gelijk aan het maximaal aantal lijnonafhankelijke wegen van p naar v.

(Aan beide versies van meervoudige samenhang voegt men de beperking toe dat v geen buur mag zijn van p.)

Deze versie van de stelling volgt uit de eigenschappen van een stroomnetwerk met eenheidscapaciteiten:

- De stroom over een maximaal aantal lijnonafhankelijke wegen van p naar v is de maximale stroom en dus ook de capaciteit van de minimale snede
- Een minimale verzameling verbindingen M die v onbereikbaar maakt vanuit p vormt een snede (P, V). Want stel P bevat alle knopen vanuit p die geen verbindingen van M bevatten en V alle andere knopen, dan bevat de snede (P, V) alle verbindingen van M. M is ook minimaal dus het kan geen andere verbindingen bevatten.

Om een verband te leggen tussen onafhankelijke wegen zonder gemeenschappelijke *knopen* en vergrotende wegen ontdubbelt men alle knopen met een verbinding van capaciteit 1 ertussen. Nu kan slechts 1 vergrotende weg gebruik maken van elke knoop. Om knopen te vinden die v onbereikbaar maken vanuit p wanneer ze weg worden gehaald, maken we alle capaciteit oneindig en die tussen ontdubbelde knopen 1.

Als de originele graaf n knopen en m verbindingen had, dan blijft het aantal knopen $O(n)$ en verbindingen $O(m)$

De stelling van Menger spreekt slechts over 1 paar knopen. **De stelling van Whitney** past die eigenschap toe op elk paar knopen van een graaf om voorwaarden op te stellen voor zijn samenhang en lijnsamenhang (en heeft dus ook vier versies):

Een graaf (al dan niet gericht) is k-voudig samenhangend (k-voudig lijnsamenhangend) als en enkel als er tenminste k knopen (verbindingen) moeten verwijderd worden om hem te doen uiteenvallen.

Om de graad k van samenhang of lijnsamenhang van een graaf te vinden, kan men dus een maximalestroomprobleem oplossen voor elk knopenpaar (van een eventueel getransformeerde graaf), en het minimum van al die oplossingen nemen. Aangezien er $Q(n^2)$ knopenparen zijn is de performantie $O(n^3m)$.

Gelukkig blijken $Q(n)$ stroomproblemen te volstaan. Want stel dat r en s producent en verbruiker zijn van het stroomnetwerk met de kleinste minimale snede (R, S) onder alle $Q(n^2)$ knopenparen. Dan blijkt dat we hetzelfde resultaat zouden gevonden hebben voor een stroomnetwerk met een willekeurige knoop x uit R als producent, en een willekeurige knoop y uit S als verbruiker. Immers, (R, S) is ook een snede voor dat stroomnetwerk. Want als men de verbindingen uit (R, S) wegneemt wordt y zeker onbereikbaar vanuit x, anders zou s ook bereikbaar blijven vanuit r. En dat stroomnetwerk kan geen kleinere snede hebben, omdat (R, S) bij onderstelling de kleinste minimale snede was. Het probleem is natuurlijk dat we de snede (R, S) niet kennen, en dus ook x en $y$niet kunnen kiezen! Maar als we alle knopen in een willekeurige volgorde $x_1, x_2, ..., x_n$ zetten (cyclisch, zodat x_1 volgt op x_n), dan zijn er altijd twee opeenvolgende knopen x_i en x_{i+1} zodat x_i in R ligt, en x_{i+1} in S. We moeten dus slechts n stroomnetwerken oplossen, voor alle paren opeenvolgende knopen, en het minimum van al die oplossingen bijhouden.

9 Koppelen

- Een **koppeling** in een ongerichte graaf is een deelverzameling van verbindingen waarin elke knoop hoogstens eenmaal voorkomt. (geen 2 verbindingen mogen een gemeenschappelijke knoop hebben).
- De eindknopen hiervan heten **gekoppeld**
- De koppeling met het grootste aantal verbindingen is de **maximale koppeling**
- Het **gewogen koppelingsprobleem** zoekt de koppeling met het grootste totale gewicht

9.1 Koppelen in tweeledige grafen

Een *tweedelige graaf* is een ongerichte graaf waarbij de knopen in twee deelverzamelingen L en R kunnen verdeeld worden, zodat alle verbindingen steeds een knoop uit L met een knop uit R verbinden.

9.1.1 Ongewogen koppeling

De tweeledige graaf moet eerst getransformeerd worden tot een stroomnetwerk: een producent die met alle knopen van L verbonden wordt en v met alle knopen van R worden ingevoerd. Alle oorspronkelijke verbindingen gaan van L naar R en hebben capaciteit 1.

Er geldt:

- Voor elke koppeling k bestaat er een overeenkomstige *gehele stroomverdeling* in het stroomnetwerk, met als netwerkstroom k.
- Voor elke gehele stroomverdeling in het stroomnetwerk met gehele netwerkstroom k, is er een koppeling met k verbindingen.
- Een maximale koppeling komt dan ook overeen met een maximale gehele stroomverdeling

Het aantal verbdingingen k is niet groter dan het aantal knopen in de kleinste verzameling L of R en dus zeker $O(n)$. Met breedte eerste zoeken voor de vergrotende wegen is dit $O(km)$ dus $O(nm)$.

9.2 Stabiele koppeling

Gegeven één of twee verzamelingen van elementen. Elk element heeft een gerangschikte voorkeurslijst van andere elementen. Deze elementen moeten gekoppeld worden, rekening houdend met hun voorkeuren, en zodanig dat de koppeling stabiel is. Een koppeling is onstabiel wanneer ze twee niet met elkaar gekoppelde elementen bevat, die liever met elkaar zouden gekoppeld zijn dan in hun huidige toestand te blijven.

Dit vormt de basis voor drie verwante problemen: *stable marriage*, *hospitals/residents* en *stable roommates*

9.2.1 Stable marriage

Hier zijn er twee verzamelingen, met dezelfde grootte. Elke man (vrouw) heeft een voorkeurslijst die alle vrouwen (mannen) bevat. Elke man moet gekoppeld worden aan een vrouw, zodat de koppeling stabiel is.

9.2.1.1 Het Gale-Shapley-algoritme

Dit algoritme zorgt ervoor dat alle mannen de beste partner krijgen die ze kunnen hebben in een stabiele koppeling.

Voorgestelde oplossing controleren: alle paren niet-gekoppelde elementen moeten hun partners verkiezen boven elkaar. Het volstaat om elk element uit een van de verzamelingen te testen met de elementen uit de andere verzameling die hoger op zijn voorkeurslijst staan dan zijn partner. Dit is $\Theta(n^2)$

Het algoritme:

- Iedereen is op elk ogenblik verloofd of vrij en een man kan tussen vrij en verloofd afwisselen, een vrouw blijft eens verloofd, altijd verloofd
- Als een vrije man een aanzoek doet aan een vrije vrouw, verloven ze. Aan een verloofde vrouw: de vrouw vergelijkt hem met haar partner, en verwerpt de laagst geklasseerde
- Elke man doet aanzoeken in volgorde van zijn voorkeurslijst, tot hij kan verloven. Bij een verbroken verloving gaat de man naar de volgende in de lijst.
- Het algoritme stopt wanneer iedereen verloofd is, en dat gebeurt zeker voor een man het einde van zijn lijst bereikt.

9.2.1.2 Eigenschappen van de oplossing

- Het algoritme stopt altijd, en is stabiel
  - Niemand wordt afgewezen door alle vrouwen: een man kan niet afgewezen worden door de laatste vrouw op zijn lijst
  - Elke man doet max 1 aanzoek per vrouw, er zijn dus maximaal n^2 aanzoeken
  - Stabiel: Want als er een man m bestaat die vrouw v verkiest boven zijn partner, dan moet v hem ooit afgewezen hebben. Kortom, m verkiest een andere vrouw v, maar niet omgekeerd. Er bestaat dus geen ongekoppeld paar dat de stabiliteit van de koppeling in gevaar kan brengen.
- Elke mogelijke aanzoekvolgorde van mannen geeft dezelfde oplossing:
  - Stel m krijgt niet zijn beste stabiele partner v, dan werd m ooit afgewezen door v
  - Onderstel dat m de eerste man is die door een stabiele partner wordt afgewezen
  - m werd dus afgewezen omdat v een andere m' verkiest, maar m' werd nog niet door een stabiele partner afgewezen, dus alle andere stabiele partners van m' staan lager dan v
  - Omdat v een stabiele partner van m is, bestaat er een andere stabiele koppeling K waarin m en v partners zijn, en waarin de partner van m' bijvoorbeeld v' is, dus deze v' staat lager dan v op de lijst van m'
  - Dan staat v' lager dan v op m' zijn lijst, dus de koppels (m, v) en (m', v') uit K zijn niet stabiel, want zowel v als m' verkiezen elkaar boven hun partners
- In deze man-georiënteerde versie krijgt elke vrouw de slechtst mogelijke partner die ze kan hebben in een stabiele koppeling
  - Onderstel dat v niet haar slechtste stabiele partner m krijgt, maar een hogere m'
  - Omdat m een stabiele partner van v is, bestaat er een andere stabiele koppeling K waarin m en v partners zijn, en waarin de partner van m' bijvoorbeeld v' is. Zowel v als v' zijn dus stabiele partners van m'.
  - Omdat elke *an* de best mogelijke stabiele partner krijgt, staat v hoger dan v' op de lijst van m', maar ook m' staat hoger dan m op de lijst van v
  - Dat betekent dat de koppels (m, v) en (m', v') uit K niet stabiel zijn, want v en m' verkiezen elkaar boven hun partners

9.2.1.3 Implementatie

- De voorkeurslijsten en ranglijst van de vrouwen (om snel 2 mannen te kunnen vergelijken) opstellen is $\Theta(n^2)$
- Een lijst wordt bijgehouden van vrije mannen. Er wordt nooit aan toegevoegd want een afgewezen man doet gewoon direct een nieuw aanzoek.
- Het algoritme stopt als de laatste vrije vrouw een aanzoek krijgt. $n-1$ vrouwen kunnen maximaal n aanzoeken krijgen $O(n^2)$. **Gale-Shapley is dus** $\Theta(n^2)$

9.2.1.4 Uitbreidingen

- **Verzamelingen van ongelijke grootte**
  Een koppeing M wordt nu als onstabiel beschouwd wanneer er een $m$en v bestaat zodat:
  1. m en v geen partners zijn
  2. m ofwel ongekoppeld blijft, ofwel v verkiest boven zijn partner
  3. v ofwel ongekoppeld blijft, ofwel m verkiest boven haar partner
- **Onaanvaardbare partners**
  De voorkeurslijsten moeten niet langer de volledige andere groep bevatten. Een koppeing M wordt nu als onstabiel beschouwd wanneer er een $m$en v bestaat zodat:
  1. m en v geen partners zijn, maar wel aanvaarbaar voor elkaar
  2. m ofwel ongekoppeld blijft, ofwel v verkiest boven zijn partner
  3. v ofwel ongekoppeld blijft, ofwel m verkiest boven haar partner
- **Gelijke voorkeuren**
  Er zijn 3 mogelijke manieren om stabiliteit te definiëren:
  1. **Superstabiliteit:** Een koppeling is onstabiel als er een man en een vrouw bestaan die geen partners zijn, en die elkaar minstens evenzeer verkiezen als hun partners. Er zijn nu gevallen waarbij geen (super)stabiele koppeling mogelijk is.
  2. **Sterke stabiliteit:** Een koppeling is onstabiel als er een man en een vrouw bestaan die geen partners zijn, waarvan de ene de andere strikt verkiest boven de partner, en de andere de eerste minstens even graag heeft als de partner. Ook hier bestaat er niet altijd een (sterk) stabiele koppeling.
  3. **Zwakke stabiliteit:** Een koppeling is onstabiel als er een man en een vrouw bestaan die geen partners zijn, en die elkaar strikt verkiezen boven hun partners. Dit geval kan getransformeerd worden tot het standaardprobleem door gelijke voorkeuren arbitrair te ordenen. Die is niet noodzakelijk uniek, omdat ze afhangt van de arbitraire ordening.

9.2.2 Hospitals/residents

Elk hospitaal moet stabiel gekoppeld worden aan een aantal stagiairs. Verschillen met stable marriage: het totaal aantal plaatsen moet niet gelijk zijn aan het aantal stagiairs, en de voorkeurslijsten mogen onvolledig zijn. Er zijn wel geen gelijke voorkeuren. Een koppeling is onstabiel wanneer er een hospitaal h en een stagiair s zijn, zodat:

1. h en s geen partners zijn, maar wel aanvaardbaar voor elkaar
2. s ofwel ongekoppeld blijft, ofwel h verkiest boven zijn toegewezen hospitaal
3. h ofwel onbezette stageplaatsen overhoudt, ofwel s verkiest boven tenminste een van zijn toegewezen stagiairs

We transformeren dit in een gewone een-eenduidige koppeling:

- Elke h met p plaatsen wordt vervangen door $h_1, h_2, ...h_p$ met elk 1 stageplaats, de s vervangen ook deze h in hun voorkeurslijst door $h_1, h_2, ...h_p$
- De oplossing van de hospitaalversie is gelijktijdig optimaal voor alle hospitalen, de oplossing is steeds dezelfde
- Omdat stagiairs die (levens)partners zijn meestal dezelfde hospitalen verkiezen, bestaat er een versie voor dergelijke paren. Elke stagairpaar (s_1, s_2) dient een gezamenlijke voorkeurslijst in. Deze bestaat uit hospitaalparen (h_1, h_2). Het is nu niet langer zeker dat er stabiele oplossingen bestaan, en zelfs als dat zo is, zijn ze soms moeilijk te vinden.

9.2.3 Stable rommates

Dit is een veralgemening van het stable marriageprobleem: er is slechts één groep met n personen (n even), die elk een voorkeurslijst opmaken van al de anderen als potentiële kamergenoten. Een koppeling verdeelt de groep in paren, en is onstabiel wanneer twee personen elkaar verkiezen boven hun toegewezen kamergenoten.

Er zijn gevallen waarvoor geen stabiele koppeling mogelijk is. Testen of er een bestaat, en zo ja, er een vinden, is $O(n2)$.

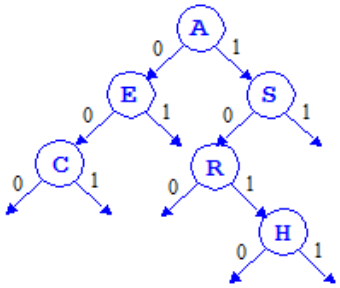
10 Gegevensstructuren voor strings

We zullen steeds onderstellen dat geen enkele sleutel een prefix is van een andere.

10.1 Digitale zoekbomen

Er is één belangrijk verschil met zoekbomen: de juiste deelboom wordt niet bepaald door de zoeksleutel te vergelijken met de sleutel in de knoop, maar enkel door het volgende element van de zoeksleutel.
We beperken ons tot bits en gebruiken dus binaire didigtale zoekbomen.

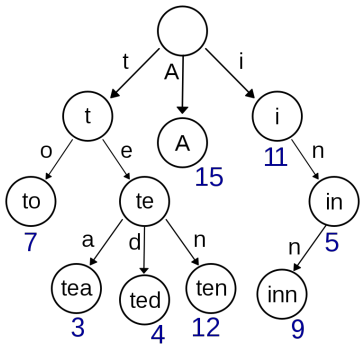
A	00001
S	10011
E	00101
R	10010
C	00011
H	10100



- Het inorder overlopen van de boom geeft niet noodzakelijk de sleutels in volgorde. Alle sleutels van een linkerdeelboom zijn wel kleiner dan de rechterdeelboom, want het vershil werd gemaakt door die $(i+1)$-de bit.
- De langste weg op de boom, h, wordt beperkt door het aantal bits van de langste opgeslagen sleutel
- Bij gelijkmatig verdeelde sleutels wordt de gemiddelde weglengte $O(\lg n)$, zodat zoeken of toevoegen $O(\lg n)$ vergelijkingen vereist.
- De performantie is dus vergelijkbaar met RZ-bomen, maar de implementatie is veel makkelijker

10.2 Tries

Een trie is een digitale zoekboomstructuur die wél de volgorde van de opgeslagen sleutels behoudt.



10.2.1 Binaire tries

Het belangrijkste verschil met digitale zoekbomen: de sleutels zitten enkel in de bladeren, inorder zoeken levert nu wel gerangschikte sleutels op en de sleutels moeten niet vergeleken worden in elke knoop van de zoekweg.
Zoeken en toevoegen:

- Komen we bij een ledige deelboem terecht, dan bevat de boom de sleutel niet
- Komen we in een blad terecht, dan bevat dit blad de enige sleutel in de boom die gelijk kan zijn aan de zoeksleutel, aangezien ze dezelfde beginbits hebben. Een vergelijking van beide sleutels geeft uitsluitsel.
  Als we moeten toevoegen moeten we het gevonden blad splitsen tot we een bit zijn tegengekomen dat verschillend is, en het ene sleutelwoord als links blad en het ander als rechts blad zetten.

Eigenschappen van binaire tries:

- wanneer opgeslagen sleutels veel gelijke beginbits hebben, zijn er veel knopen met slechts 1 kind.
- Een sleutel mag geen prefix zijn van een andere sleutel, want die eerste sleutel moet in een blad zitten.
- De structuur is onafhankelijk van de toevoegvolgorde, er is steeds maar 1 trie mogelijk
- In een trie opgebouwd uit n gelijkmatig verdeelde sleutels vereist zoeken of toevoegen van een willekeurige sleutel gemiddeld $O(\lg n)$ *bit*operaties.
- **De bitlengte van de langste sleutel is de bovengrens voor het aantal operaties bij zoeken en toevoegen in slechtse geval**

10.2.2 Meerwegstries

Om de hoogte van een trie met lange sleutels te beperken kan men in elke knoop meerdere (opeenvolgende) sleutelbits tegelijk behandelen. Als elk sleutelelement m waarden kan aannemen, krijgt hij potentieel m kinderen.

Zoeken en toevoegen gebeurt analoog als bij een binaire trie. Om het vergelijken van het volgende sleutelelement $O(1)$ te houden worden de kinderen bijgehouden in een tabel en geïndexeerd.

De sleutels worden in 'alfabetische' volgorde bijgehouden.

De performantie is analoog aan binaire tries. Zoeken vereist gemiddeld $O(\lg_m n)$ testen, en dat aantal is nooit groter dan de lengte van de zoeksleutel. Zo'n trie heeft gemiddeld $n/\ln m$ inwendige knopen en het aantal wijzers in die knopen wordt $mn/\ln m$

Wat als sleutels wel een prefix zijn van anderen? Elke sleutel afsluiten met een speciaal afsluitelement, dat dan altijd een blad is.

Mogelijke verbeteringen:

- Idpv tabellen wordt er gebruik gemaakt van gelinkte lijsten van kinderen, er is minder geheugenplaats nodig ten koste van snelheidsverlies. Soms bij heel grote m: hoogste niveaus hebben tabellen, intermediare bv hashtabellen en laagste niveau's gebruiken gelinkte lijsten.
- Er kan ook bv enkel een trie gebruikt worden voor de bovenste niveaus en daarna een andere structuur,bv gewoon een lijst die sequentieel doorzocht wordt.

10.3 Variabelelengtecodering

In dit geval hebben we een alfabet waarbij we niet elke letter door evenveel bits voorstellen (bv bij compressie).

Bij het decoderen is het praktisch om een *prefixcode* te hebben. Dit is een codering waarbij een codewoord nooit het prefix van een ander codewoord kan zijn. Dit zorgt er voor dat we weten wanneer we het einde van een codewoord bereikt hebben.

Een **trie** wordt gebruikt voor de decodering van een invoerstroom gecodeerd met een prefixcode. Eerst slaan we alle codewoorden in de trie op. Aan het begin van een codewoord vertrekken we aan de wortel en per ingelezen symbool gaan we een niveau omlaag in de trie. Als we bij een blad komen weten we dat het codewoord volledig is. De bijhorende data kunnen opgeslagen zijn in dat blad.

10.3.1 Universele codes

(Kunnen beschouw worden als huffmancodes voor een oneindig alfabet (zie later))

- **Elias-γcode**
  Het getal n wordt voorgesteld door zo weinig mogelijk bits (k bittekens), en begint dus met een 1. We laten dit dan vooraf gaan door $k-1$ nulbits. Het aantal nulbits voor de eerste 1-bit geeft het aantal bits van het getal aan. n wordt dus voorgesteld door $2\lfloor \log_2 n \rfloor + 1$ bittekens.

- **Elias-δcode**
  De laatste $k-1$ bits van het getal n (dus zonder de eerste 1-bit), en deze wordt vooraf gegaan door elias-γcode voor k. n wordt hier voorgesteld door $\lfloor \log_2 n \rfloor + 2\lfloor \log_2(\log_2 +1) \rfloor + 1$

- **Fibonaccicode**
  De fibonaccireeks heeft als eigenschap dat elk getal n op precies 1 manier kan geschreven worden als som van verschillende Fibonaccigetallen die *geen buren van elkaar zijn.*

  Om de Fibonaccicode van een getal te bepalen, overloop je de reeks van klein naar groot en gebruik je een 1-bit voor elk getal dat in de berekende som voorkomt en een nulbit voor de anderen. Als afsluiting wordt een extra 1-bit geplaatst, wat het einde van het getal aanduid, aangezien enkel hier 2 opeenvolgende 1'en kunnen voorkomen.

  Het getal n met $F_k \leq n \leq F_{k+1}$ kan worden voorgesteld met $k+1$ bits.

10.4 Huffmancodering

= om minder bittekens te gebruiken voor letters die veel voorkomen (totale lengte van bestanden inkorten)

10.4.1 Opstellen van een decoderingsboom

Bij huffmancodering wordt een prefixcode toegepast waarbij elke letter een apart codewoord heeft dat over de het hele bericht hetzelfde blijft. We gebruiken in de uitleg bitcodes en we stellen dus dat de trie binaire is.

Uitgaande van een alfabet $\Sigma = \{s_i | i = 0, ..., d - -1\}$ krijgen we een reeks van frequenties f_i (aantal keer dat een codewoord gebruikt wordt). We zoeken dus een trie met n bladeren die de optimale code oplevert.

We nemen een willekeurige volledige (= geen uitwendige knopen met maar 1 kind, want dit kan verkort worden naar een korter codewoord) binaire trie met d bladeren, elk met een letter uit Σ.

We kennen gewichten toe:

- Een blad krijgt als gewicht de frequentie van de overeenkomstige letter
- Een inwendige knoop krijgt als gewicht de som van de gewichten van zijn kinderen. Dit is ook de som van de gewichten van de bladeren die onder de knoop hangen

Het aantal keer dat we naar een bepaald blad gaan wordt gegeven door zijn gewicht, het aantal keer dat wel langs knoop k gaan ook door zijn gewicht. Het aantal bits in het gecodeerde bestand is dus de som van de gewichten van alle knopen samen, met uitzondering van de wortel. We zoeken dus een trie met minimaal gewicht.

Verbeteringen die we kunnen maken aan de trie

- Stel dat we een trie hebben met een knoop k met gewicht w_k op diept d_k en een knoop l met gewicht w_l op diepte d_l.
- Wat gebeurt er als we een nieuwe trie maken door k, l en hun deelbomen om te wisselen?
  - Er waren d_k kopen boven k in de trie. Deze verliezen gewicht w_k, maar krijgen gewicht w_l bij.
  - analoog bij l
- Damen krijgen we dus een gewichtsverandering van

$$(d_k - d_l)(w_l - w_k)$$

- Heeft l een *groter* gewicht maar een *kleinere* diepte dan k, dan hebben we een betere trie gevonden. (als de dieptes gelijk zijn mogen we de verwisseling doen zonder een slechter resultaat te bekomen)
- We kunnen dus een optimale trie opstellen met de volgende eigenschappen:
  - Geen enkele knoop heeft een groter gewicht dan een knoop op een kleinere diepte
  - Geen enkele knoop heeft een groter gewicht dan een knoop links van hem op dezelfde diepte

Constructie van zo'n optimale trie

- We bouwen de boom op van onder naar boven: op elk ogenblik hebben we een *bos* van deelbomen die we verder aan mekaar moeten hangen. In het begin bestaat het bos uit alle bladeren.
- We weten nog niet wat de diepte h gaat zijn van de boom die we construeren. Wel weten we
  - dat alle knopen op niveau h bladeren zijn
  - dat dat aantal even is
  - dat de bladeren op dat niveau alle vooraan staan, en wel in een volgorde die van rechts naar links gaat in de trie.
- We beginnen met de bladeren twee aan twee samen in een boom te steken, elke keer de twee lichtste overblijvende bladeren nemend
- Vermits deze boom op niveau $h-1$ in het resultaat komt, staat hij verder naar achter in ons gesorteerde bos dan de overblijvende knopen van niveau h.
- Op het ogenblik dat we klaar zijn met het niveau h steken alle knopen/bomen van niveau $h-1$: ofwel hebben we ze geconstrueerd door knopen uit het vorige niveau samen te nemen ofwel zijn het bladeren die we al hadden.
- We kunnen dus verder gaan met elke keer de twee lichtste bomen samen te nemen in een nieuwe boom.
- Hoe weten we ofdat we klaar zijn met niveau h? Dit weten we niet maar uiteindelijk gaan we maar 1 boom overhouden, dit is dan onze trie.

In toepassingen moet men de Huffmantrie zelf op een of andere manier bij het bestand toevoegen ofwel maakt men gebruik van een vaste, vooraf afgesproken Huffmantrie.

10.5 Patriciatries

Patricitatries vermijdt onnodig geheugengebruik (tries hebben knopen met maar 1 kind) en het gebruik van 2 soorten knopen.

Patricia behoudt enkel knopen met meer dan 1 kind. Bij binaire tries wordt het dan mogelijk om slechts één soort knopen te gebruiken: **Sleutels worden in inwendige knopen opgeslagen, en wijzers naar bladeren worden vervangen door wijzers naar inwendige knopen.**

We behandelen eerst het weglaten van knopen: knopen met maar 1 kind kunnen we weglaten en het kind in de plaats zetten, dit heeft 2 gevolgen:

1. Om de juiste weg te vinden in het kind moeten we een karakter van de string overslaan. Het kind moet dus aanduiden hoeveel ouders hij ontbreekt. We slaan de index van het te testen karakter op in de knoop, de *testindex*.
2. Het kan zijn dat het karakter dat we nu overslaan in de zoekstring niet overeenkomt met het karakter dat in de verdwenen knoop naar zijn kind leidde

We noemen een knoop expliciet als hij nog voorkomt in de resulterende boom en impliciet als hij alleen wordt aangeduid door een indexsprong aangegeven in de nakomeling.

Woordenboekoperaties op een niet ledige trie met enkel expliciete knopen

- **Zoeken:**
  We gaan naar beneden en nemen het karakter aangegeven door de testindex om de weg te volgen. Als we in een blad komen vergelijken we de zoekstring met de string in het blad.

  Soms heeft elke inwendige knoop een verwijzing naar een willekeurig onderliggend blad. Zo kunnen we bij een inwendige knoop met impliciete voorouders al kijken of de beginbits overeenkomen met dat willekeurig blad.

- **Toevoegen:**
  Het kan zijn dat we een blad moeten toevoegen aan een impliciete knoop. Als we de wijzers hebben die in "zoeken" besproken werden, kunnen we meteen controleren of we in tussenliggende impliciete knopen wel de goede kant op gaan.

  In elk geval krijgen we een verschilindex die de eerste plaats aanduidt waar de nieuwe string verschilt van de meest gelijkende string in de trie. Als de boom leeg is dan zetten we de verschilindex op nul.

  Als de nieuwe string nog niet in de trie zit hebben we drie mogelijkheden:

  1. Zoeken eindigt in een expliciete knoop waar testindex = verschilindex, maar die geen kind heeft voor het karakter in de string aangeduid door de verschilindex. Deze is geen blad, want dan zou een prefix van de nieuwe string al in de trie zitten. We kunnen dus gewoon een blad toevoegen voor de nieuwe string.
  2. We komen uit in een expliciete knoop waar de testindex groter is dan de verschilindex. Er moet een expliciete knoop worden toegevoegd met als testindex de verschilindex. Deze krijgt twee kinderen: de oude expliciete knoop (via het oude karakter) en het nieuwe blad (via het nieuwe karakter).
  3. We geraken opnieuw tot in het blad. Als we een blad opvatten als een knoop met oneindig grote testindex dan is dit een speciaal geval van het vorige.

  Toevoeg geval 1 kan nooit voorkomen aangezien expliciete inwendige knopen altijd 2 kinderen heeft. We moeten dus altijd zowel een blad als een expliciete inwendige knoop toevoegen, dus deze voegen we samen toe als 1 structuur zodat de noodzaak van bladpointers verdwijnt.

Het nieuwe afdalen en toevoegen

Bij afdalen in de boom kijkt men alleen naar het inwendigeknoopgedeelte. Men weet dat men in een blad aankomt als men bij afdalen stuit op een knoop met een testindex die niet groter is dan de vorige. Voor n sleutels zijn er dus n knopen vereist, die $2n$ wijzers bevatten. Daarvan verwijzen er n naar een sleutel, en $n-1$ naar een kindknoop.

Als we een string toevoegen wordt er dus ´e´en structuur bijgemaakt, een versmelting van een inwendige knoop met een blad. Aanvankelijk is de inwendige knoop de directe ouder van het blad, maar latere wijzigingen kunnen daar nog knopen tussenvoegen.

De structuur hangt af van de toevoegvolgorde

Welk blad versmolten is met een inwendige knoop hangt af van de volgorde: het is steeds het tweede blad onder de inwendige knoop, behalve bij de wortel. Dit komt omdat een expliciete inwendige knoop wordt samengeplakt met het blad dat tot het ontstaan van de inwendige knoop leidde.

Ledige bomen

We moeten bij zoeken en toevoegen een onderscheid maken tussen een ledige en een niet-ledige boom omdat elke knoop een plaats voor een sleutel bevat. Bij een boom met ´e´en sleutel is de wortel een blad met testindex -1. Een ledige boom heeft testindex.

Performantie

De hoogte van de boom en dus het maximaal aantal bitvergelijkingen worden opnieuw beperkt door de lengte van de langste opgeslagen sleutel. Patriciatries testen echter meteen (en enkel) de belangrijke karakters, zodat de zoektijd niet toeneemt met de sleutellengte. Ze zijn dus zeer geschikt voor lange sleutels.

Kortom, patriciatries combineren de voordelen van digitale zoekbomen en tries:

- Zoals digitale (en gewone) zoekbomen gebruiken ze niet meer geheugen dan nodig. Er zijn n bladeren en hoogstens $n-1$ inwendige knopen.
- Ze zijn even effici¨ent als tries: zoeken vereist gemiddeld $O(\lg n)$ karaktervergelijkingen, gevolgd door ´e´en sleutelvergelijking.
- Net zoals tries (en gewone zoekbomen) respecteren ze de volgorde van de sleutels, zodat bijkomende operaties mogelijk zijn.

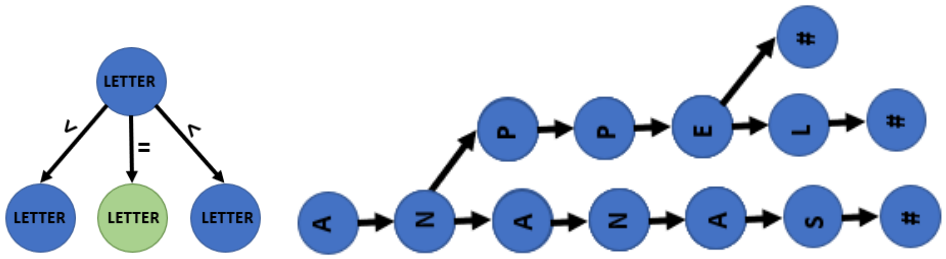
10.6 Ternaire zoekbomen

Deze zijn alternatieve voorstelling voor meerwegstries. Bij meerwegstries gebruikten we een tabel met m kindwijzers. Dit geheugenverlies gaan we vermijden door een ternaire boom te gebruiken waar elke knoop een *sleutelelement* bevat.

Zoeken vergelijkt een zoeksleutelelement met het element in de huidige knoop:

- Is het zoeksleutelelement kleiner, dan zoeken we verder in de linkse deelboom
- Is het zoeksleutelemenent groter, dan zoeken we verder in de rechtse deelboom
- Is het zoeksleutel element hetzelfde, dan gaan we naar de middenste deelboom en onderzoeken we het **volgende** element van de zoeksleutel met deze volgende knoop

We gebruiken opnieuw een geschikt afsluitelement om er voor te zorgen dat geen enkele sleutel een prefix van een andere kan zijn.



Woordenboekoperaties

- **Zoeken:** een sleutel wordt gevonden wanneer we met zijn afsluitelement bij een knoop met datzelfde element terechtkomen. Wanneer we een nulwijzer of een niet-afsluitelement tegenkomen bij ons laatste element, zit de sleutel niet in de boom.
- **Toevoegen:** zoeken en dan het aanmaken van een reeks opeenvolgende knopen.

Het aantal knopen hangt enkel af van de opgeslagen sleutels, niet van de toevoegvolgorde.

Voordelen van ternaire zoekbomen:

- Aan onregelmatig verdeelde zoeksleutels passen ze zich goed aan:
  - In dit geval zouden meerwegtries met hun tabellen te veel plaats gebruiken
  - Sleutels hebben vaan een gestructureerd formaat (volgorde van ons alfabet bv). De overeenkomstige deelbomen bij ternaire zoekbomen gaan zich automatisch gedragen als zoekbomen met 26 knopen.
- Zoeken naar afwezige sleutels is meestal zeer efficiënt. Ternaire zoekbomen vergelijken dikwijls slechts enkele sleutelelementen, en volgen slechts enkele wijzers.
- Ze laten complexere zoekoperaties toe, zoals zoeken naar een sleutel waarvan bepaalde elementen niet gespecifieerd zijn (*don't care karakters*)

Mogelijke verbeteringen aan ternaire zoekbomen:

- Het aantal knopen kan beperkt worden
  - sleutels in bladeren op te slaan van zodra men ze kan onderscheiden
  - inwendige knopen met slechts 1 kind te elimineren via een sleutelindex
  - Zo wordt de zoekweg onafhankelijke van de sleutellengte
- De wortel wordt vervangen door een meerwegstrieknoop met tabel van lengte m of zelfs m^2 zodat we de eerste 2 sleutelelementen direct juist kunnen plaatsen.

11 Zoeken in strings

Algemeen gebruikte symbolen in dit hoofdstuk

Σ	Het gebruikte alfabet
$\Sigma*$	De verzameling van strings van eindigde lengte uit Σ
d	Het aantal karakters in Σ
P	Het gezochte patroon (de naald)
p	De lengte van P
T	De tekst waarin gezocht wordt
t	De lengte van T

11.1 Formele talen

Een formele taal over een alfabet is een verzameling eindige strings over dat alfabet. Formele talentheorie bestudeert de vraag hoe men een taal kan beschrijven.

Formele talen kunnen op 2 manieren beschreven worden

1. Contextvrije grammatica/generatieve grammatica (bijvoorbeeld bij parsers)
2. Reguliere expressies

11.1.1 Generatieve grammatica's

Hierbij gaan men uit van een startsymbool dat kan getransformeerd worden tot een zin van de taal met behulp van *substitutieregels*. Hiervoor gebruikt met buiten het alfabet Σ ook een andere symbolen: **niet-terminalen** (deze worden omgezet in andere strings door substitutieregels en zijn dus nooit finaal).

Niet-terminalen worden aangeduid met $\langle ... \rangle$ en de verzamling van Σ met de niet-terminalen wordt aangeduid met Ξ.

Contextvrije talen is een geval waarbij men grammatica's van een bepaalde vorm kan gebruiken. Het is contextvrij omdat de substitutie onafhankelijk is van wat voor of achter de niet-terminaal staat.

Voorbeeld van een contextvrije taal

$$\begin{array}{lll} \langle S \rangle & ::= & \langle AB \rangle | \langle CD \rangle \\ \langle AB \rangle & ::= & a\langle AB \rangle b | \epsilon \\ \langle CD \rangle & ::= & c\langle CD \rangle d | \epsilon \end{array}$$

(Hierin is ϵ niets). Met deze grammatica kunnen we bijvoorbeeld de volgende zin vormen:

$$\langle S \rangle \to \langle CD \rangle \to c\langle CD \rangle d \to ccc\langle CD \rangle dd \to cccc\langle CD \rangle ddd \to cccddd$$

11.1.2 Reguliere uitdrukkingen

Een regex is een string over ons alfabet $\Sigma = \{\sigma_0, \sigma_1, ..., \sigma_{d-1}\}$ aangevuld met de symbolen $\emptyset$, ϵ, $*$, (,), en $\underline{|}$ gedefinieerd door

$$\begin{array}{lll} \langle \textbf{regex} \rangle & ::= & \langle \textbf{basis} \rangle | \langle \textbf{samengesteld} \rangle \\ \langle \textbf{basis} \rangle & ::= & \sigma_0 | \sigma_1 | ... | \sigma_{d-1} | \emptyset | \epsilon \\ \langle \textbf{samengesteld} \rangle & ::= & \langle \textbf{plus} \rangle | \langle \textbf{of} \rangle | \langle \textbf{ster} \rangle \\ \langle \textbf{plus} \rangle & ::= & (\langle \textbf{regex} \rangle \langle \textbf{regex} \rangle) \\ \langle \textbf{of} \rangle & ::= & (\langle \textbf{regex} \rangle \underline{|} \langle \textbf{regex} \rangle) \\ \langle \textbf{ster} \rangle & ::= & (\langle \textbf{regex} \rangle) * \end{array}$$

Elke regex definieert een formele taal, die we aanduiden als $Taal(R)$ voor een regen R. De definitie van een regex verloopt recursief. Op het eerste niveau hebben we primitieve reguliere talen

1. $\emptyset$ is een regex met als taal de lege verzameling
2. De lege string ϵ is een rexep met als taal Taal$(R) = \{\epsilon\}$
3. Voor elke $a \in \Sigma$ is "a" een regex, met als taal Taal$(R) = \{"a"\}$

Regex kunnen dan gecombineerd worden adhv concatenatie, of en *Kleenesluiting*

- Concatenatie: $(RS) = Taal(R) \cdot Taal(S)$, alle strings uit de eerste verzameling gevolgd door de 2de verzameling
- of: $(R|S) = Taal(R) \cup Taal(S)$
- Kleenesluiting: $(R)* = Taal(R)*$, verzameling van alle opeenvolgingen van een eindig aantal strings mogelijk in R

Deze definitie levert alle nodige reguliere expressies.

Regex als graaf
Dat regexs verbonden kunnen worden met graafproblemen blijkt uit de volgende stelling:

Stelling:

Zij G een gerichte multigraaf met verzameling takken Σ. Als a en b twee knopen van G zijn dan is de verzameling $P_G(a, b)$ van paden beginnend in a en eindigend in b een reguliere taal over Σ

Bewijs:

- We bewijzen dit door inductie op het aantal verbindingen m van G
- **Voor $\mathbf{m = 0}$**:
  - $a \neq b$ dan is $P(a, b) = \emptyset$
  - $a = b$ dan is $P(a, b) = \epsilon$
- **Met 1 verbinding**: We breiden dit uit naar multigraaf G' door aan G 1 verbinding toe te voegen: verbinding v_{xy} die van knoop x naar y loopt. Alle paden van a naar b zijn dan van de 2 volgende vormen:
  1. Paden die v_{xy} niet bevatten, deze vormen de reguliere taal $P_G(a, b)$
  2. Paden die v_{xy} 1 of meer keer bevatten. Deze verzameling wordt gegeven door

$$P_G(a, x) \cdot \{v_{xy}\} \cdot (P_G(y, x) \cdot \{v_{xy}\}) * \cdot P_G(y, b)$$

- Als unie van deze twee reguliere talen is $P_{G'}(a, b)$ regulier. Hiermee is de stelling bewezen.

Dit geldt ook voor een verzameling knopen met $P_G(A, B)$ voor alle $a \in A$ en $b \in B$, want dit is een unie van de verzamelingen $P_G(a, b)$

Een belangrijk idee voor sommige toepassingen is het **substitutieprincipe:**
Als we in een rexep elke letter van het basisalfabet vervangen door een reguliere uitdrukking over hetzelfde alfabet of over een ander alfabet, dan krijgen we opnieuw een regex.

Toepassing van het substitutieprincipe:

- Verbindingen van grafen vervangen door etiketten: een letter of string over een alfabet
- Het etiket van een pas is de string over het alfabet van etiketten.
- Volgens het *substitutieprincipe* is het zo, dat als we twee verzamelingen A en B van knopen van een geëtiketteerde graaf hebben, dat de verzameling van de etiketten van al de paden van A naar B een regulire taal is

Als we een reguliere taal hebben kunnen we er een contextvrije grammatica voor bedenken:

- De primitieve rexeps leiden tot de volgende grammatica's:

Regex	Grammatica
$\emptyset$	$\langle S\rangle ::= \langle S\rangle$
ϵ	$\langle S\rangle ::= \epsilon$
a	$\langle S\rangle ::= a$

- Zijn R en S twee regex en hebben de bijhorende contextvrije grammatica's de startsymbolen $\langle R\rangle$ en $\langle S\rangle$
  - Dan kunnen we grammatica's voor $(R|S)$ en RS construeren door de grammatica's van R en S onder mekaar te schrijven en ze te laten vooraf gaan door de regel $\langle T\rangle ::= \langle R\rangle|\langle S\rangle$ dan wel $\langle T\rangle ::= \langle R\rangle\langle S\rangle$
  - Een grammatica voor $R*$ wordt gegeven door $\langle T\rangle ::= \langle R\rangle\langle T\rangle|\epsilon$

Alle reguliere talen zij dus contextvrij.

11.2 Variabele tekst

11.2.1 Een eenvoudige methode

De meest voor de hand liggende methode: test of P vanaf positie j in T voorkomt door overeenkomstige karakters van beide te vergelijken.
Als de strings in T random zijn dan stopt het testen bij j meestal na 1 vergelijking. De **gemiddelde uitvoeringstijd wordt dus O(t)**. In het **slechtste geval wordt dit O(tp)**

11.2.2 Zoeken met de prefixfunctie: Knuth-Morris-Pratt

11.2.2.1 De prefixfunctie

Nemen we een string P en i met $i \leq p$. We zeggen dat we een string Q voor i op P kunnen leggen als $i \geq Q.size()$ en als Q overeenkomt met de even lange deelstring van P eindigend voor i.

De prefixfucntie $q()$ van een string P bepaalt voor elke stringpositie $i, 1 \leq i \leq p$, de lengte van de langste prefix van P met lengte kleiner dan i dat we voor i kunnen leggen.

$q(i)$ is altijd kleiner dan i. $q(0)$ is niet gedefinieerd en $q(1)$ is altijd 0.

Voorbeeld: "aabaaab" geeft $[-1, 0, 1, 0, 1, 2, 2, 3]$

De bepaling van $\mathbf{q(i+1)}$

- $q(i+1)$ kan zeker niet groter zijn dan $q(i)+1$, en alleen als $P[q(i)]$ en $P[i]$ overeenkomen.
- Als ze niet overeenkomen, proberen we een korter prefix te verlengen dat we voor positie i kunnen leggen. Maar een prefix korter dan $q(i)$ dat we voor i kunnen leggen, kunnen we ook voor $q(i)$ leggen. Dus het kan niet langer zijn dan $q(q(i))$
- Kunnen we dit met 1 karakter verlengen? Enkel als $P[q(i)] = \nexists i]$, ... enzoverder met $q(q(q(i)))...$
- De prefixen kunnen dus door stijgende indices berekend worden, aangezien $q(1)$ steeds nul is

Efficiëntie

- Er moeten p prefixwaarden berekend worden, en voor elke waarde is er mogelijks een herhaling die de vorige prefixwaarden overloopt. Op het eerste zicht is dit $O(p^2)$
- Het algoritme levert een dubbele lus op. De binnenste lus wordt p keer uitgevoerd. De buitenste lus wordt q steeds met hoogstens 1 verhoogd zen in de binnenste met minstens 1 verminderd.
- De binnenlus wordt dus hoogstens $p - q(p)$ keer uitgevoerd met $q(p) \geq 0$.
- Aangezien er $p-1$ buitenste herhalingen zijn, wordt de totale performantie $\Theta(p)$

11.2.2.2 Een eenvoudige lineaire methode

Een string wordt samengesteld bestaande uit P gevolgd door T, gescheiden door een speciaal karakter dat in geen van beiden voorkomt. Daarna bepaalt men de prefixfunctie $\Theta(t+p)$. Wanneer $p = q(i)$, werd P gevonden, beginnend bij index $i-p$ in T. We moeten nu enkel de eerste p waarden van de prefixfunctie bewaren, dus deze methode is (p)

11.2.2.3 Het Knuth-Morris-Prattalgoritme

Hoe ver kunnen we opschuiven voor de volgende vergelijking?

Stel dat P op een bepaalde positie vergeleken wordt met T en dat $P[i] \neq T[j]$. Als $i > 0$: als we P verschuiven met een stap s kleiner dan i dan hebben we overlapping tussen het begin van P en het prefix van P dat we al in T hebben gevonden. Deze overlappende delen hebben nu lengte $i - s$ en moeten natuurlijk overeenkomen. De kleinste waarde waarvoor dit kan is $i-s = q(i)$. **We kunnen P dus $i-q(i)$ plaatsen opschuiven en dan verder gaan door T[j] te vergelijken met** $P[q(i)]$

Nog een verbetering

We weten ook dat $P[i]$ en $T[j]$ verschillen. Als $P[q(i)] = P[i]$ dan treedt er onmiddelijk een fout op en dit weten we zonder nog naar $T[j]$ te kijken.

Het algoritme
Opschuiven met stap s is dus enkel zinvol als het prefix van P met lengte $i - s$ een prefix is van P dat we voor i kunnen leggen *en* als $P[i - s] \neq P[i]$.

- Bij een bepaalde i kunnen we de kleinst mogelijke waarde s berekenen die hieraan voldoet
- Een functie q' wordt berekend zodat $i - q'(i)$ de kleinste zinvolle s-waarde geeft
- Of $T[j] = P[q'(i)]$ weten we niet, dit wordt de eerstvolgende karaktervergelijking
- Aangezien de bijkomende vereiste voor het nieuwe prefix enkel met P te maken heeft, kunnen deze nieuwe prefixwaarden $q'(i)$ voor elke positie in P op voorhand bepaald worden. We kunnen ook $q'(p)$ zo definiëren dat $p - q'(p)$ de spring is na het einde van P.

Efficiëntie

Het aantal karaktervergelijkingen is $\Theta(t)$. Want na elke verschuicing van P wordt hoogstens 1 karakter van T getest dat vroeger reeds getest werd. Aangezien de voorbereiding $\Theta(p)$ is (prefixtabel opstellen), wordt de totale performantie $\Theta(t + p)$

11.2.3 Het Boyer-Moorealgoritme

Een tekst T en een patroon P worden op verschillende beginposities karakter per karakter vergeleken. Het patroon wordt hier echter *van achter naar voor* overlopen. Daarnaast maat Boyer-Moore gebruik van 2 heuristieken die groteren verschuivingen mogelijk maakt.

11.2.3.1 De heuristiek van het verkeerd karakter

We vergelijken patroon en tekst op een bepaalde beginpositie in de tekst. We beginnen achteraan in P, en stel dat dit karakter, dat we f noemen, een fout geeft. P naar rechts verschuiven doen we dan enkel tov f.

Stel dat we *de meest rechtse positie* in P kennen van elke letter in het alfabet. We moeten dan de positie van f opzoeken, en P opschuiven zodat f overeen komt. Als f er niet in voorkomt, kunnen we opschuiven tot de positie na f.

De fout treedt niet altijd achteraan op, maar bv op positie $i(0 \leq i < p)$. Als j de meest rechtse positie in P is van f, dan schuiven we over $i - j$ posities. Als dit getal echter negatief is, wordt er geen rekening mee gehouden en wordt er gewoon minstens 1 plaats (zie volgende heuristiek) opgeschoven.

Efficiëntie: het voorbereidend werk vereist een tabel die in $\mathbf{\Theta(p + d)}$ kan ingevuld worden.

Varianten van deze heuristiek

1. **Uitgebreide heuristiek van het verkeerde karakter**
    - Om negatieve verschuivingen te vermeiden: de meest rechtse positie j *links* van de patroonpositie i waar de fout werd gevonden wordt gebruikt
    - Nadeel: we moeten nu j voor alle f maar ook voor alle i's bepalen
    - Om dit snel te kunnen opzoeken kunnen we een 2D tabel gebruiken, met d rijden en p kolommen.
    - *Compromis tussen plaats en tijd:* voor elke f wordt een dalende lijst met alle posities waar f voorkomt in P bijgehouden. We kunnen deze in $\Theta(p+d)$ opstellen en de gebruikte plaats is $\Theta(p)$

2. **Variant van Horspool**

   - Voor elk karakter van het alfabet bevat de tabel de meest rechtse positie j van het karakter in P, links van positie $p-1$. ($j=-1$ als het daar niet voorkomt)
   - Wanneer er bij het vergelijken een fout optreedt bij patroonpositie i en tekstpositie k, dan moet P opgeschoven worden. Tegenover tekstkarakter $T[k+p-1]$ mag enkel een nieuw patroonkarakter terechtkomen dat gelijk is aan dat tekstkarakter, anders is er meteen weer een fout.
   - Nu ligt dat patroonkarakter linkst van $P[p-1]$, en zijn positie j vinden we in de tabel met als indexkarakter $T[k+p-1]$. De verschuiving van P is dan $p-1-j$ en dus steeds positief
   - Deze verschuiving is onafhankelijk van de foutieve patroonpositie i. Ook als het patroon in de tekst voorkomt, kan de verschuiving groter zijn dan 1.
   - Omdat de eerste heuristiek dit zo snel maakt en de tweede nu niet meer nodig is om een positieve verschuiving te garanderen, gebruikt de **Boyer-Moore-Horspool methode** enkel de eerste heuristiek
   - In het slechtste geval is deze methode $O(pt)$

3. **Variant van Sunday**

   - Dit gebruikt bijna dezelfde tabel als in de originele methode, maar als er een fout optreedt tussen patroonpositie i en tekstpositie k, dan zorgt dit ervoor dat bij verschuiving het gepaste patroonkarakter tegenover tekstkarakter $T[k+p]$ terechtkomt.
   - De verschuiving is opnieuw onafhankeloijk van de foutieve patroonpositie i waardoor dat de volgorde waarin de karakters van P en T vergeleken wordt geen rol speelt.

11.2.3.2 De heuristiek van de jusite suffix

Wanneer nagenoeg alle beginposities van P in T moeten getest worden, en er op elke beginpositie veel testen nodig zijn cooraleer er een fout gevonden wordt (bv bij zeer klein alfabet (DNA)), dan zou Boyer-Moore met enkel de eerste heuristiek in het slechtste geval een performantie van $O(pt)$ hebben.

Als we bij positie i in P een verkeerd karakter f in T vinden, dan hebben we een suffix van P in T gevonden, met lengte $p-i-1$. P moet dus verschoven worden zodat de suffix met het deel in T overeenkomt. Dit geeft ons dus **twee mogelijke verschuivingen** waarvan we altijd de grootste nemen.

We moeten dus te weten komen of dit juiste suffix s nog ergens in P voorkomt en waar. Als het meermaals voorkomt nemen we het meest rechtse. Als het niet meer voorkomt zouden we een correct beginpositie voor P kunnen overslaan.

De suffixfunctie

Stel dat we, analoog met de prefixfucntie, voor elke index j in P de lengte $s(j)$ van het grootste *suffix* va P kennen, dat daar *begint*.

- Als $s(j) < p-i-1$ (lengte van het juiste suffix), dan komt j niet in aanmerking voor de gezochte positie k. Maar ook niet als $s(j) > p-i-1$, want het kortere juiste suffix is rechts van j in P terug te vinden.
- Dus enkel j waar $s(j) = p-i-1$ komen in aanmerking voor k, en aangezien we de grootste nodig hebben, moet k de grootste van de j waarden hebben.
- De gezochte tweede verschuiving voor foutpositie i wordt in een tabel v opgeslagen.

Speciale gevallen

- **Het patroon P werd gevonden**
  - De 'foutieve' patroonpositie i is dan -1 (dus de eerste heuristiek geeft geen verschuiving). Het juist suffix is nu P en komt natuurlijk maar 1x in P voor.
  - We mogen (zoals we volgens de 2de heuristiek zouden doen) niet gewoon P p karakters opschuiven.
  - De maximale overlapping is het langst mogelijke suffix van P, dat overeenkomt met P beginnend bij positie 0. De lengte van dit suffix is $s(0)$, met als overeenkomstige verschuiving van $p - s(0)$. Deze slaan we op in een extra tabelelement $v[-1]$.
  - Deze is natuurlijk gelijk aan p als er geen overlapping is
- **Er is geen juist suffix**
  - Als de fout optreedt bij patroonpositie $p - 1$, dan is er geen juist suffix, dus we kunnen hier geen verschuiving bepalen
  - De verschuiving bepaald via de eerste heuristiek is nu zeker positief, waardoor we $v[p-1]$ een waarde van minimaal 1 kunnen geven
- **Het juiste suffix komt niet meer in P voor**
  - We moeten P dan over meer dan i posities verschuiven, maar niet noodzakelijk p posities
  - Als er dan meerdere suffixen zijn die overeenkomen met een prefix van P, nemen we het grootste, want dit geeft de kleinste verschuiving
  - De lengte van het gezochte suffix is dus weer $s(0)$, zodat de overeenkomstige verschuiving $v[i]$ opnieuw $p - s(0)$ wordt.

Performantie

Het opstellen van v lijkt traag omdat voor elke foutpositie i de indices j waarvoor $s(j) = p - i - 1$ lineair moesten gezocht worden in de suffixtabel van P.

We kunnen ook: de suffixtabel 1 maal overlopen, voor elke index j de foutpositie $i = p - s(j) - 1$ en de verschuiving $v[i] = i + 1 - j$ bepalen, en de kleinste verschuiving voor elke i bijhouden. Dit komt overeen met de grootste j waarde, dus als we gewoon j van groot naar klein overlopen en overschrijven wat er staat (zonder controle), bekomen we ons antwoord.

We vullen heel de tabel v eerst op met $p - s(0)$ (de waarde voor foutposities waarvoor geen suffix gevonden wordt)

- Voor foutposities $0 \leq i < p$ waarbij het juiste suffix nog voorkomt, wordt de suffix overschreven
- "" niet meer voorkomt, is $p - s(0)$ de vereiste verschuiving
- Voor $i = p - 1$, waarbij er dus geen juist suffix bestaat, moet $v[p - 1]$ de waarde 1 krijgen.

Deze verbeterde versie overloopt maar 1 keer de suffixtabel, en per index j is het werk $O(1)$, ze wordt dus $\Theta(p)$. Het voorbereidend werk voor de 2de heuristiek wordt dus ook $\Theta(p)$ met $\Theta(p)$ geheugenplaats.

Performantie van het ganse algoritme

Indien P niet in T zit $O(t + p)$. Indien wel blijft het **slechtste geval O(tp)**. We kunnen dit echter verhelpen:

Wanneer P voorkomt in T moet de verschuiving $p - s(0)$ zijn, om het grootst mogelijke overlappende suffix $s(0)$ van P niet te missen. Daarna begint de vergelijking van P en T weer achteraan P. Wanneer deze vergelijking echter bij index $s(0)$ van P is gekomen, kan ze stoppen, want het suffix werd reeds gevonden. Daarmee is het algoritme nu in alle gevallen lineair.

11.2.4 Onzeker algoritmen

Onzeker algoritmen zijn algoritmen die soms radicaal fout gaan. Er zijn twee soorten toepassingen waar zo'n algoritme handig is:

1. Vb van koeien testen op besmetting: De altijd juiste test kost 200/test en de onzeker 10. De onzeker test geeft nooit een foutief *negatief* antwoord. Dus Het onzeker wordt altijd gebruikt, en als een koe positief getest wordt, wordt het zekere algoritme toegepast op die koe.

   - Dit noemt met een scheef algoritme, oftewel kan enkel soms fout zijn, oftewel enkel false (dit zijn *waar-scheve* en *onwaar-scheve* algoritmen)

2. Je hebt de keuze tussen

   - Een klassiek deterministisch algoritme, dat 2 milliseconden nodig heeft
   - Een Monte Carloalgoritme (onzeker algoritme) dat 1 keer op 2^100 een fout antwoord geeft, maar de berekeningen in 1 milliseconde uitvoert.

   Dit algoritme draait op een computer waar 1 keer op de duizend jaar een fout optreedt. Het blijkt dat het 2de dan veel minder kans heeft op fouten.

11.2.5 Het Karp-Rabinalgoritme

Deze methode herleidt strings naar getallen die dan vergeleken worden. Het heeft enkel zin als getallen sneller kunnen vergeleken worden dan strings.

Principe: aan elke mogelijke string van lengte P kent men een uniek geheel getal toe. Gelijke strings betekenen dan gelijke getallen. Maar er zijn d^p verschillende strings van lengte p en deze kunnen enkel efficiënt vergeleken worden als ze in een processorwoord (lengte w) passen. We gaan er dus voor zorgen dat we de strings door 2^w verschillende getallen (die we *fingerprints* noemen) voorstellen (=hashing) zodat we ze in $O(1)$ kunnen vergelijken. Verschillende strings gaan nu hetzelfde getal hebben, dus als we een deelstring hebben gevonden met dezelfde fingerprint als P, moeten we nog nakijken of het dezelfde string is (= onzeker algoritme).

De fingerprint van P berekenen

Elk stringelement kan d verschillende waarden aannemen, dus elke "letter" wordt voorgesteld als een cijfer tussen $[0, d-1]$. Het groot getal dat P voorstelt wordt dan

$$H(P) = \sum_{i=0}^{p-1} P[i]d^{p-i-1}$$

Dit vereist p optelling en p vermenigvuldigingen.

Om er voor te zorgen dat het woord in een processorwoord past, gebruiken we de rest bij deling door een getal r $0 < r \leq 2^w$. Het volgende is dan de *fingerprint* van P:

$$H_r(P) = H(P) \mod r$$

Eerst het grote getal $H(P)$ bepalen en daarna pas de rest bepalen is niet efficiënt. Door gebruik te maken van de volgende eigenschap, kunnen we van alle tussenresultaten de rest nemen, en de berekeningen doen met deze restwaarden idpv grote getallen. (geldt ook voor verschil en product)

$$(a+b) \mod r = (a \mod r + b \mod r) \mod r$$

Deelbewerkingen vereisen dan $O(1)$ waardoor het berekenen van de fingerprint voor P $\Theta(p)$ wordt

De fingerprints van de deelstrings van T bepalen

Als we op de vorige manier alle deelstrings van T bepalen, komen we aan $\Theta(tp)$, wat dus niet goed is. Gelukkig is er een eenvoudig verband tussen T_{j+1} en T_j (j is de beginpositie):

$$H(T_{j+1}) = (H(T_j) - T[j]d^{p-1})d + T[j+p]$$

De fingerprint voor T_{j+1} wordt dan

$$H_r(T_{j+1}) = ((H(T_j) - T[j]d^{p-1})d + T[j+p]) \mod r$$

We mogen opnieuw de resten gebruiken bij berekening. De waarde van $d^{p-1} \mod r$ moet slechts 1 maal bepaald worden. Deze fingerprint kan dus (behalve T_0) in $O(1)$ berekend worden.

De fingerprint van T_0 bepalen

analoog aan $H_r(P)$ en is dus ook $\Theta(p)$:

$$H_r(T_O) = \left(\sum_{i=0}^{p-1} T[i]d^{p-i-1} \right) \mod r$$

Performantie

De berekening van $H_r(P)$, $H_r(T_0)$ en $d^{p-1} \mod r$ vereist dus $\Theta(p)$ operaties. De berekening van alle fingerprints vereist $\Theta(t)$, zodat dit algoritme $\Theta(t+p)$ is.

We moeten natuurlijk strings met dezelfde fingerprint nog vergelijken, wat $O(p)$ vereist, waardoor het algoritme in slechtste geval $O(tp)$ wordt.

De keuze van r

- **Vaste r**
  Kies voor r een zo **groot mogelijk priemgetal** zodat $rd \leq 2^w$ (dit zodat er geen overflow is bij het berekenen van tussenresultaten). Het verband tussen opeenvolgende fingerprints wordt: ($r(d-1)$ om negatief tussenresultaat te vermijden)

  $$H_r(T_{j+1}) = \Big(\big((H_r(T_j) + r(d-1) - T[j](d^{p-1} \mod r)) \mod r \big) d + T[j+p] \Big) \mod r$$

  .

- **Random r**
  Dit zorgt er voor dat de kans op vergissingen niet bepaald wordt door de waarschijnlijkheidsverdeling van het patroon/tekst.

  De foutkans wordt willekeurig klein gemaakt door gebruik te maken van een groot bereik. Hiervoor wordt de eigenschap gebruikt dat het aantal priemgetallen kleiner dan k ongeveer $k/\ln k$ is. Als we k groot genoeg kiezen, zijn er slechts een klein aantal priemen dat een fout veroorzaken, dus de kans dat r een van die priemen is, is de foutkans, en wordt dus klein. Met $k = t^2$ wordt dit $O(1/t)$

  Om hiervan een *Las Vegas* algoritme te maken kunnen we:

  - Overgaan op een eenvoudigere methode als de $O(t)$ test een fout signaleert. De gemiddelde uitvoeringstijd voor een foutloos algoritme wordt dan $(1-1/t)(\Theta(t+p)+O(t)) + (1/t)(\Theta(t+p) + O(t) + O(tp)) = \Theta(t+p)$
  - Herbeginnen met een nieuwe random priem r wanneer de test een fout signaleert. Met $O(1/t)$ moeten we het algoritme dan gemiddeld $O(t/(t-1))$ keer uitvoeren en wordt dit $\Theta(t+p)$

11.2.6 Zoeken met automaten

Automaten hebben staten. Bij een deterministisch automaat beschrijft dit de gehele toestand van de eenheid. (beetje anders bij niet-deterministische)

Het automaat wordt voorzien van een invoerkanaal dat de veranderende omgeving voorstelt. Tijd wordt als een discrete eenheid beschouwd dus op elk ogenblik is er een invoer. De verzameling van alle mogelijke invoerwaarden noemt het alfabet.

Al wat een automaat doet als het een karakter als invoer krijgt is veranderen van staat en eventuele uitvoer genereren. Nieuwe staten hangen enkel af van oude staten.

Stings zijn nu een reeks symbolen dat het automaat als invoer krijgt. We gaan nu op zoek naar een verzameling strings. Als zo'n string wordt tegenegekomen, wordt er een uitvoersignaal gegenereerd.

11.2.6.1 Deterministische automaten

Een DA is een abstract model voor een schakeling die of een programma dat izch in 1 van een aantal toestanden kan bevinden, in elke toestand een symbool invoert, en afhankelijk van dat symbool van toestand kan veranderen.

Een DA bestaat uit

Symbool	Betekenins
Σ	een eindige verzameling invoersymbolen (alfabet)
S	een eindige verzameling toestanden
s_0	een een begintoestand behorende tot S
F	een verzameling eindtoestenden ($\in S$)
$p(t, a)$	een overgangsfunctie

De overgangsfuntcie geeft de nieuwe toestand wanneer het automaat in toestand t symbool a ontvangt.

Een DA kan voorgesteld worden door een gerichte geëtiketteerde multigraaf G, de *overgangsgraaf*, met als knopen de toestanden en als verbindingen de overgangen, met als etiket het overeenkomstig invoersymbool.

Een string wordt herkend als de DA zich in een eindtoestand bevindt. De etiketten op de weg tussen s_0 en deze eindtoestand geven de opeenvolgende symbolen van deze string. De verzameling strings die door de DA herkend worden is de taal ervan.

Deze taal is regulier: het is een verzameling etiketten van $P_G(\{s_0\}, F)$. Dit is de helft van de stelling van Kleene, die zegt dat de verzameling van reguliere talen gelijk is aan de verzameling talen die herkenbaar zijn door DA.

Softwareimplementatie: tweedimensionale overganstabel met als rijen de toestanden en als kolommen de symbolen.

11.2.6.2 Niet-deterministische automaten

Een NA is een speciale manier om een DA voor te stellen. Een NA heeft een aantal statenbits, die de de waarden 'aan' en 'uit' kunnen aannemen. De staat van de NA is dan de verzameling statenbits die aan staan. Een NA is in een eindstaat als een of meerdere eindbits aan staan, de andere bits zijn van geen belang.

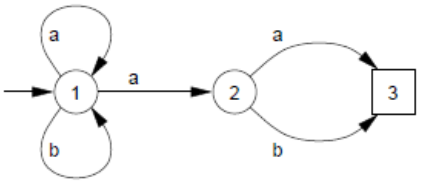
Een statenbit die een of meer signalen binnenkrijgt zet zichzelf aan, een statenbit die geen signaal binnenkrijgt dooft uit of blijft uit. Is i een statenbit en a een letter uit het alfabet, dan is $s(i, a)$ de verzameling statenbits die rechtstreeks een signaal van i krijgen als de inkomende letter a is.

ϵ-overgangen: Als er een ϵ-overgang is van i naar j, en i krijgt een signaal binnen, dan stuurt i direct, zonder vertraging, een signaal naar j, zodanig dat ze beide aangezet worden. (eigenlijk niet nodig want we kunnen dit vervangen door overgangen te leggen van de voorgangers van i naar j zelf, maar dit maakt het ontwerp makkelijker)

Softwareimplementatie: de overgangstabel zal verzamelingen van statenbits moeten bevatten ipdv eenvoudige staten, en een extre kolom voor het 'invoersymbool' ϵ.

Bij elk invoersymbool moet een *verzameling* statenbits bepaald worden. Met elke string van invoersymbolen kunnen er dus meerdere wegen in de overgangsgraaf overeenstemmen. Een string wordt in de overgangsgraaf herkend als eer *een* weg bestaat tussen beginbit s_0 ene een eindtoestand.

NA voor $(a|b) * a(a|b)$:

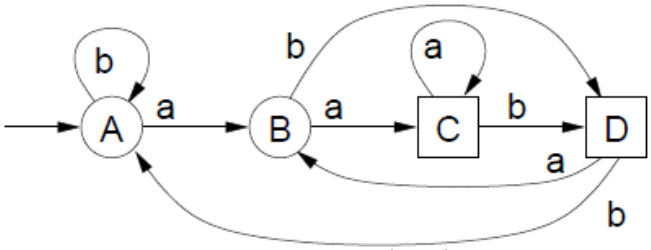


		a	b
$\rightarrow$	1	$\{1,2\}$	$\{1\}$
	2	$\{3\}$	$\{3\}$
$\leftarrow$	3		

11.2.6.3 De deelverzamelingconstructie

= Het omzetten van een NA naar een DA. Elke staan van een NA is een verzameling statenbits die aanstaan en is dus impliciet gedefinieerd. Het omzetten naar een DA met expliciete staten is de *deelverzamelingenconstructie*.

DA geconstrueerd uit vorige NA:



		a	b
$\rightarrow$	$A = \{1\}$	B	A
	$B = \{1,2\}$	C	D
$\leftarrow$	$C = \{1,2,3\}$	C	D
$\leftarrow$	$D = \{1,3\}$	B	A

Als er k statenbits zijn dan zijn er 2^k mogelijke deelverzameling, maar de NA bereikt meestal maar een klein aantal van deze deeelverzamelingen, waardoor het aantal toestanden in de DA vergelijkbaar is met de NA.

We gaan de impliciet gegeven multigraaf met 2^k knopen doorlopen en DEZ of BEZ, vertrekkend van de beginstaat om te zien welke staten bereikt worden. De beginstaat waar b_0 aan staat en al de rest uit is niet altijd de beginstaat van de DA (als we ϵ-overgangen hebben vanuit b_0).

Hulpoperaties: buren in de NA kunnen we niet opzoeken in een burenlijst, we hebben 2 hulpoperaties nodig:

- De deelverzameling van statenbits bereikbaar via ϵ-overgangen vanuit een verzameling statenbits T noemt met de ϵ-sluiting(T)
  - Het bepalen van de ϵ-sluiting komt neer op het zoeken van alle statenbits bereikbaar via ϵ-overgang vanuit T (psuedocode pg 134)
- De overgangsfunctie $p(t, a)$ kunnen we uitbreiden voor een *verzameling* statenbits T tot $p(T, a)$.

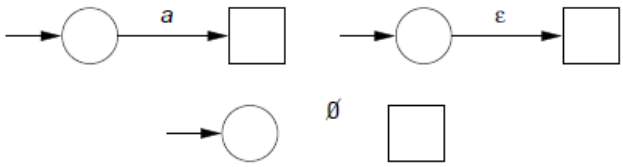
Constructie van de DA:

- we moeten de verzameling van de toestanden D en zijn overgangstabel M opstellen. De begintoestand van de DA is de ϵ-sluiting.
- Om de overgang vanuit toestand T voor invoer a te bepalen moeten we alle statenbits (en ϵ-overgangen) vinden voor $s(T, a)$.
- De nieuwe DA-toestand is dus ϵ-sluiting($s(T, a)$)
- (pseudocode pg 135) Er wordt een stapel gebruikt en het algoritme stopt omdat het aantal mogelijke DA-toestanden eindig is, en elke DA-toestand slechts eenmaal op de stapel terecht komt.

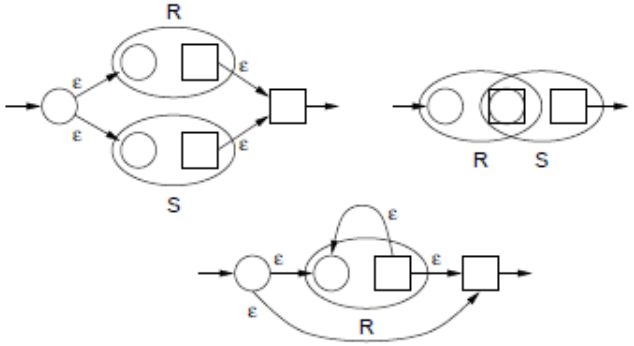
11.2.6.4 Automaten voor regex

De constructie van **thomson** gebruikt de structuur van de regex als leidraad om een NA op te bouwen. De methode werkt *bottom-up*: eerst worden primitieve elementen gedefinieerd, daarna de basisoperatoren en ten slotte wordt een complexere regex opgebouwd hieruit.

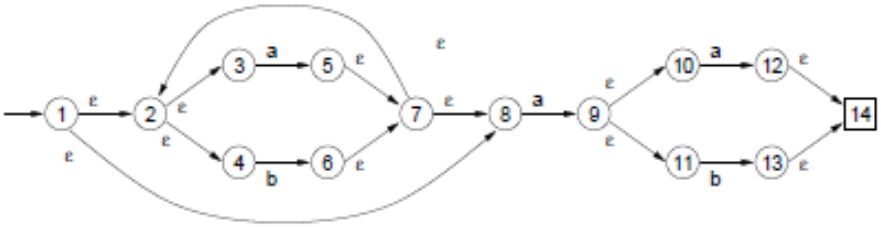
- **Basiselementen**
  Dit zijn de symbolen uit Σ (a), ϵ en $\emptyset$



- **Basisoperator**
  Stel r en s regexen waarvoor de overeenkomstige NA's R en S geconstrueerd werden.
  De constructie van Thomson voor $r|s$, $r \cdot s$ en $r*$:



- **Constructie van een complexere regex**
  De NA die bekomen wordt voor $(a|b) * a(a|b)$



Stelling 2 (Kleene): *Een taal kan herkend worden door een eindige deterministische automaat als en slechts als ze regulier is*

Bewijs: we hebben al opgemerkt dat een taal die herkend wordt door een automaat zeker regulier is, omdat taal bestaande uit etiketten van paden vertrekkend uit de beginstaat en eindigend in een eindstaat regulier is.

Is omgekeerd een taal regulier, dan wordt ze beschreven door een regex en voor deze regex kunnen we met de bovenstaande methode een DA construeren.

Gevolg: *niet alle contextvrije talen zijn regulier.* Bijvoorbeeld een taal bestaande uit strings beginnend met 'a's gevolgd door evenveel 'b's kan niet herkend worden.

Eigenschappen van de NA uit de constructie van Thomson

- De NA heeft slechts 1 eindbit. Er zijn geen overgangen naar de beginbit en ook niet vanuit de eindbit.
- Het aantal bits van de NA is niet groter dan tweemaal het aantal elementen in de regex (elke constructiestap bestaat uit hoogstens 2 bits)
- Vanuit elke bit van de NA vertrekken niet meer dan twee overgangen

Efficiëntie: De constructie is $O(|r|)$ met $|r|$ de lengte van r. Want de syntactische ontleding van regex r is $O(|r|)$ en met elk gevonden symbool of operator komt 1 contructiestap overeen die $O(1)$ is. Het aantal bits is ook $O(|r|)$

Implementatie mogelijkheden van het abstracte model NA

1. **Een DA kan geïmplementeerd worden** met een programma of digitale schakeling. We transformeren de NA volgens de deelverzamelingconstrucite. Een DA zo geconstrueerd, heeft als herkenningstijd van een stirng s met lengte $|s|$ $O(|s|)$
2. **De NA rechtstreeks simmuleren:**
   het bepalen van alle bits van de NA die aanstaan na het inlezen van elke symbool zoals bij deelverzamelingenconstructie gebeurt. Daarbij worden alle overgangen bepaald die de DA voor strings zou maken.
   De grootte van de deelverzamelingen is $O(|r|)$ en met goede gegevensstructuren voor de deelverzamelingen kan de bepaling van de functies $p(T, a)$ en ϵ-sluiting(T) ook $O(|r|)$ worden. Omdat er vanuit elke bit hoogstens 2 overgangen zijn vereist de overgangstabel slechts $O(|r|)$ plaats.
   Voor het herkennen van een string s moeten er $|s|$ overgangen bepaald worden, zodat de totale uitvoeringstijd $O(|s| \times |r|)$ bedraagt

3. **Hybridische methode:** (combineert snelheid eerste en plaatsgebruik tweede)
   De deelverzamelingenconstructie gebeurt bij elke string, maar reeds bepaalde overgangen worden bijgehouden in het cachegeheugen. In plaats van de volledige DA te construeren, gebeurt er vaak slechts een deel daarvan omdat enkel werkelijk gebruikte overgangen (eenmalig) bepaald worden. Als een overgang moet gebeuren zoeken we dan eerst in de cache. Als het zoeken in de cache efficiënt gebeurt, blijkt herkennen van een string met deze methode bijna even snel als (1). De constructie is erbij vaak sneller omdat niet gebruikte overgangen nooit bepaald worden. Haar plaatsgebruik is zoals (2) vermeerder met de cache.

11.2.6.5 Minimaliseren van een automaat

Elke DA kan getransformeerd worden in een equivalente DA, die dezelfde taal herkent, maar met een minimaal aantal toestanden.

Hierbij gaan we proberen om equivalente toestanden te groeperen, elke groep bevat toestanden die hetzelfde gedrag vertonen. *Twee toestanden zijn **equivalent** wanneer men vanuit beide toestanden, na invoer van om het even welke string, ofwel in twee gewone toestanden terechtkomen, ofwel in twee eindtoestanden (niet noodzakelijk dezelfde).*

Opsporen van equivalente toestanden (eenvoudige implementatie)

Om na te gaan of twee toestanden equivalent zijn zou men alle mogelijke strings moeten uitproberen. We gaan dus omgekeerd werken en de toestanden opsporen die niet equivalent zijn.

- Stel dat t voor invoersymbool a overgaat naar toestand v, en u voor a naar w.
- Als v en w niet equivalent zijn, dan zijn t en u dat evenmin
- Voor elk paar toestanden gaat men na naar welke paren toestanden ze overgaan, en dat voor elk invoersymbool
- Deze testen gaan door tot er niets meer verandert
- We moeten natuurlijk wel met een paar niet-equivalente toestanden beginnen, maar dit is eenvoudig: elke eindtoestant onderscheid zich van elke gewone toestand
- Wanneer men geen niet-equivalente toestanden meer kan vinden, zijn de overblijvende paren equivalent

Opsporen van equivalente toestanden (betere implementatie)

Er zijn $O(t^2)$ paren toestanden en na elke iteratie wordt er minstens 1 nieuw paar niet-equivalente toestanden gevonden wordt de implementatie van het vorige $O(|\Sigma|t^4)$. We kunnen dit echter verbeteren:

- Voor elk paar toestanden wordt een gelinkte lijst bijgehouden met toestanden die ervan afhangen: als een paar niet-equivalente bevonden wordt, geldt dat voor alle paren op zijn lijst
- Het eenmalig opstellen van de lijsten is $O(|\Sigma|t^2)$, en elke lijst wordt slechts 1 maal overlopen, wanneer het overeenkomstige paar toestanden niet-equivalent blijkt
- De totale lengte van alle lijsten is ook $O(|\Sigma|t^2)$ omdat elk paar tot hoogstens $|\Sigma|$ lijsten kan behoren
- Dit wordt dus $O(|\Sigma|t^2)$

Nog sneller maken

Dit gebeurt door een extra verfijning: telkens wordt nagegaan of de toestanden van eenzelfde groep nog steeds equivalent zijn, en indien niet wordt de groep gesplitst. Dit stopt als er geen groepen meer gesplitst worden, de groepen zijn dan equivalent. Initieel zijn er 2 groepen: de eindtoestanden en de rest.

Een goede implementatie van deze methode is $O(|\Sigma| t \lg t)$

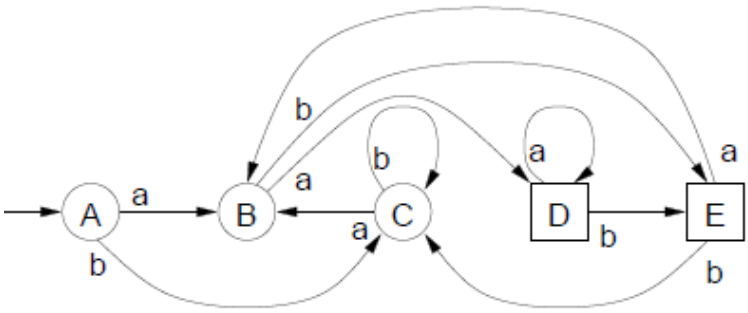
Opstellen van de gereduceerde DA

Eens de partitie gekend is gebeurt het volgende:

- Per groep wordt er een *vertegenwoordiger* gekozen, deze wordten de toestanden in de nieuwe DA.
- Elke groep van de partitie bestaat volledig uit eindtoestanden OF bevat geen enkele eindtoestand.
- Overgangen tussen toestanden worden overgangen tussen vertegenwoordigers
- Ten slotte worden alle toestanden uit de nieuwe DA verwijderd die onbereikbaar zijn vanuit de begintoestand

Voorbeeld:

Volgende automaat



		a	b
$\rightarrow$	$A = \{1,2,3,4,8\}$	B	C
	$B = \{2,3,4,5,7,8,9,10,11\}$	D	E
	$C = \{2,3,4,6,7,8\}$	B	C
$\leftarrow$	$D = \{2,3,4,5,7,8,9,10,11,12,14\}$	D	E
$\leftarrow$	$E = \{2,3,4,6,7,8,13,14\}$	B	C

wordt gereduceerd tot

		a	b
$\rightarrow$	$A = \{A,C\}$	B	A
	B	D	E
$\leftarrow$	D	D	E
$\leftarrow$	E	B	A

11.2.7 De Shift-And-Methode

Deze bitgeoriënteerde methode houdt voor elke positie j in de tekst T bij welke prefixen van het patroon P overeenkomen met de tekst, *eindigend* op positie j. Daartoe gebruikt men een tabel R met p logische waarden. Het i-de element van de tabel komt overeen met het prefix van lengte i.

Principe

R_j is de tabel bij tekstpositie j, dan is $R_j[i-1]$ waar als de eerste i karakters van P overeenkomen met de i tekstkarakters eindigend in j. De nieuwe tabel R_{j+1} kan uit de vorige afgeleid worden:

$$R_{j+1}[0] = waar\ als\ P[0] = T[j+1]$$

$$R_{j+1}[i] = waar\ als\ R_j[i-1]\ waar\ is\ en\ P[i] = T[j+1]$$

De voorstelling van P en T

Aangezien de elementen logische waarden zijn en processorinstructies op alle bits van processorwoorden tegelijk kunnen werken, gebruikt men de afzonderlijke bits van een processorwoord om karakters voor te stellen.

Bij berekening van R_{j+1} moeten we weten of karakter $T[j+1]$ overeenkomt gelijk is aan $P[i]$. Dit willen we in een processorwoord kunnen steken en aangezien P nooit verandert kunnen we dat processorwoord op voorhand bepalen. Daarvoor stellen we een tabel S op met d woorden. Een bit i van woord $S[s]$ is waar als het karakter s op plaats i in P voorkomt.

Bepaling van $\mathbf{R_{j+1}}$

Om alle bits van R_{j+1} gelijktijdig te berekenen voeren we eerst een *schuifoperatie* naar rechts op R_j uit, gevolgd door een bit-per-bit EN operatie met $S[T[j+1]]$:

$$R_{j+1} = Schuif(R_j)\ EN\ S[T[j+1]]$$

Bij de schuifoperatie wordt vooraan een waar-bit ingeschoven. De waarde $R_{j+1}[0]$ wordt dan bepaald door de eerste bit van $S[T[j+1]]$, dus ze is waar als $T[j+1] = P[0]$.

Dit kan in 1 processorinstructie gebeuren waarden R_{j+1} bepalen $O(1)$ wordt

Initialisatie van R_0

We kunnen dezelfde schuif- en EN-operaties gebruiken als we een extra tabel R_{-1} invoeren, waar elke bit onwaar is.

De performantie

Er gebeuren $\Theta(tp)$ bitoperaties maar dit wordt $O(t)$ woordoperaties als $p \leq w$ (of p een klein veelvoud van w). Tabel S opstellen duurt $\Theta(d)$ voor initialisatie en $\Theta(p)$ voor het invullen. **De totale performantie wordt dus $\mathbf{\Theta(t+p)}$**. Dit is dus een goede methode voor kleine patronen.

11.3 De shift-and-methode: benaderende overeenkomst

De shift-and-methode kan makkelijk aangepast worden om fouten in het gevonden patroon toe te laten.

11.3.1 1 extra karakter op een willekeurige plaats

We voeren een tweede andaloge tabel R_j^1 in (R_j wordt nu R_j^0), die alle prefixen aanduidt met hoogstens 1 inlassing. Dus $R_j^1[i]$ is waar als de eerste i karakters van P overeenkomen met i van de eerste $i+1$ karakters die in de tekst eindigen op j.

Voor de bepaling van $R_{j+1}^1[i]$ maken we onderscheid tussen inlassing aan het einde van de prefix of eventueel op een andere plaats:

1. De eerste i patroonkarakters komen exact overeen tot tekstpositie j. Als we dan tekstkarakter $T[j+1]$ na die i karakters inlassen
2. De eerste $i-1$ patroonkarakters komen overeen tot tekstpositie j met hoogstens 1 inlassing, en $T[j+1] = P[i]$ (inlassing dus niet aan het einde van de prefix)

Beide gevallen kunnen samen of apart voorkomen dus ze worden gecombineerd met een OF operatie:

$$R_{j+1}^1[i] = R_j^0[i] \; OF \; (R_j^1[i-1] \; EN \; T[j+1] = P[i])$$

Dit wordt herschreven als

$$R_{j+1}^1 = R_j^0 \; OF \; (Schuif(R_j^1) \; EN \; S[T[j+1]])$$

11.3.2 1 karakter verwijderd op een willekeurige plaats

R_j^1 duidt nu de tabel voor 1 verwijdering aan. Er zijn weer twee mogelijke gevallen:

1. De eerste $i-1$ patroonkarakters komen exact overeen tot tekstpositie $j+1$. Hier werd dus $P[i]$ verwijderd. Dan is bit $R_{j+1}^0[i-1]$ uit de nieuwe tabel R^0 waar.
2. De eerste $i-1$ patroonkarakters met hoogstens 1 verwijdering komen overeen met tekstpositie j, en $T[j+1] = P[i]$. De verwijdering gebeurt opnieuw ergens binnenin en niet aan het einde.

Dit wordt weer gecombineerd met een OF-operatie

$$R_{j+1}^1[i] = R_{j+1}^0[i-1] \; OF \; (R_j^1[i-1] \; EN \; T[j+1] = P[i])$$

Dit wordt herschreven als

$$R_{j+1}^1 = Schuif(R_{j+1}^0) \; OF \; (Schuif(R_j^1) \; EN \; S[T[j+1]])$$

11.3.3 Een karakter vervangen op willekeurige plaats

Er zijn weer twee gevallen

1. De eerste $i-1$ patroonkarakters komen exact overeen tot bij tekstpositie j. Karakter $P[i]$ werd dus vervangen door $T[j+1]$
2. De eerste $i-1$ patroonkarakters komen met hoogstens 1 vervanging overeen tot bij tekstpositie j, en $P[i] = T[j+1]$. De vervanging gebeurt weer binnenin.

Dit wordt weer gecombineerd met een OF-operatie

$$R_{j+1}^1[i] = R_j^0[i-1] \; OF \; (R_j^1[i-1] \; EN \; T[j+1] = P[i])$$

Dit wordt herschreven als

$$R_{j+1}^1 = Schuif(R_j^0) \; OF \; (Schuif(R_j^1) \; EN \; S[T[j+1]])$$

11.3.4 Hoogstens 1 van de bovenste fouten toelaten

Dit combineert de vorige 3 gevallen

$$R^1_{j+1} = R^0_j \; OF \; Schuif(R^0_{j+1}) \; OF \; Schuif(R^0_j) \; OF \; (Schuif(R^1_j) \; EN \; S[T[j+1]])$$

11.3.5 Hoogstens f fouten toelaten

//Vrij zeker dat Cnops gezegd heeft dat dit niet te kennen is.

We gebruiken nu f tabellen $R^1, R^2, ..., R^f$ waarbij R^k alle mogelijke overeenkomde prefixen aanduidt met *hoogstens* k fouten.

Er zijn nu 4 mogelijke gevallen om een overeenkomst te vinden voor de eerste i karakters met hoogstens k fouten, eindigend bij tekstpositie $j+1$

1. Er bestaat een overeenkomst van de eerste i patroonkarakters met hoogstes $k-1$ fouten bij tekstpositie j. Hierbij wordt tekstkarakter $T[j+1]$ ingelast na $P[i]$.
2. Er bestaat een overeenkomst van de eerste $i-1$ patroonkarakters met hoogstes $k-1$ fouten bij tekstpositie $j+1$. Dat betekent de verwijdering van $P[i]$
3. Er bestaat een overeenkomst van de eerste $i-1$ patroonkarakters met hoogstes $k-1$ fouten bij tekstpositie j. Dat komt neer op een vervanging van $P[i]$ door $T[j+1]$
4. Er bestaat een overeenkomst van de eerste $i-1$ patroonkarakters met hoogstens k fouten tot bij tekstpositie j, en $P[i] = T[j+1]$

Alles samen geeft dit Dit wordt herschreven als

$$R^k_{j+1} = R^{k-1}_j \; OF \; Schuif(R^{k-1}_{j+1}) \; OF \; Schuif(R^{k-1}_j) \; OF \; (Schuif(R^{k-1}_j) \; EN \; S[T[j+1]])$$

$$R^k_0 = 11...100...0$$

(met k 1'en en m bits in totaal)

Performantie

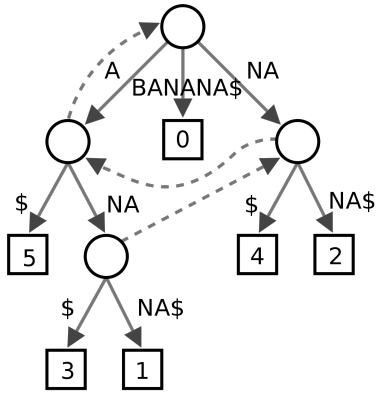
Het voorbereidend werk is S opstellen en $f+1$ tabellen R_0 initialiseren wat $O(f\lceil p/w \rceil)$ is. Daarna gebeuren er $O(f\lceil p/w \rceil)$ zodat **de totale performantie** $O(tf\lceil p/w \rceil)$ **wordt**. Voor $p << w$ wordt dit $O(tf)$

12 Indexeren van vaste tekst

Voor zoekoperaties op vaste tekst T waarin frequent gezocht moet worden. Als voorbereidend werk slaat men alle suffixen van de tekst in een gegevensstructuur op.

$suff_i$ is het suffix dat begint op plaats i.

12.1 Suffixbomen



Een optie voor de opslag van T is de **meerwegstrie**, maar: constructietijd is $O(t^2)$ en omdat suffixen gemiddeld $t/2$ lang zijn vereist de trie $O(|\Sigma|t^2)$

Nieuwe structuur werd bedacht: de suffixboom (verwant aan de patriciatrie/meerwegstrie). Zoals bij patriciatries worden knopen met maar 1 kind weggelaten waardoor de nodige inwendige knopen $O(t)$ wordt en **het nodige geheugen O($|\Sigma|$t)**

Wijzigingen tegenover patricia

1. Bij patricia werden strings opgeslagen in bladeren. Nu hebben is de tekst altijd vast dus slaan we er gewoon de beginindex i van $suff_i$ in op.
2. We gaan nu in de plaats van testindexen gebruiken, gewoon in elke knoop de begin en eindindex van de substring opslaan in de knopen.
3. Elke inwendige knoop krijgt nu ook een staartpointer om opbouw en sommige toepassingen sneller te maken

De staartpointer

We noemen de **staart** van een string de string bekomen door het eerste karakter te verwijderen. De staart noteren we als **staart(s)**.

De staart van een suffix is zelf ook een suffix en zit ook in de boom. Er is in de trie enkel een *expliciete* knoop aanwezig horend bij niet-ledige string α, als de trie twee strings bevat $\alpha\beta$ en $\alpha\gamma$, zodat de eerste letter van β verschilt van de eerste letter van γ. $staart(\alpha\beta)$ en $staart(\alpha\gamma)$ zitten dan ook in de boom, en dus is er een expliciete knoop aanwezig $staart(\alpha)$. De staartpointer van de knoop met prefix α wijst nu naar die tweede inwendige knoop.

Toepassingen van suffixbomen

- **Het klassieke deelstringprobleem**
  - Alle beginposities van P in T worden gevraagd
  - De suffixboom wordt opgesteld en daarin zoeken we de knoop die overeenkomt met P
  - Als die niet bestaat, staat P er niet in, zo wel, dan zijn de beginposities de indices bij alle bladeren die opvolgers zijn van deze knoop
  - Als P k maal voorkomt, kunnen die dus gevonden worden in $O(P + k)$
  - Als we er slechts 1 zoeken, kunnen we in de inwendige knoop een index van een van zijn bladeren opslaan. Dit voorbereidend werk is $O(t)$ (als we een blad aanmaken bij het aanmaken van een expliciete knoop zetten we de index er in). Dan wordt de hele operatie $O(p)$
- **Langste gemeenschappelijke deelstring**
  - Gegeven een verzameling van k strings $S = \{s_1, s_2, ..., s_k\}$, met totale lengte t. Gezocht de langste gemeenschappelijke deelstring van al die strings.
  - Een *veralgemeende* suffixboom wordt gebruikt: bladeren bevatten naast beginindex i ook tot welke string het suffix behoort. De constructie tijd is slechts $O(t)$
  - We construeren dus een veralgemeende suffixboom voor de strings S.
  - Elke inwendige knoop van de boom komt overeen met een prefix van een suffic, en deze deelstring komt voor in elke string die vermeld wordt bij een blad dat opvolger is van die knoop
  - De boom wordt dan overlopen om de lengte van de prefixen en het aantal *verschillende* strings waar ze in voorkomen te vinden, en dus meteen ook het langste prefix dat in alle strings voorkomt.
  - De boom heeft $O(t)$ knopen en kan dus in $O(t)$ doorlopen worden, zodat de oplossing ook in $O(t)$ gevonden wordt.
  - Handig: elke string van S krijgt een verschillend afsluitkarakter, zo zien we direct tot welke het behoort

12.2 Suffixtabellen

Een suffixtabel is een tabel met 'alfabetisch' gerangschikte suffixen van teskt T. De tabel bevat de beginindices van de suffixen. De tabel bevat geen informatie over het gebruikte alfabet wat de nodige geheugenruimte kleiner maakt.

De suffixtabel $A[]$ van de string bananas$

i =	1	2	3	4	5	6	7
$A[i]$ =	7	6	4	2	1	5	3
1	$	a	a	a	b	n	n
2		$	n	n	a	a	a
3			a	a	n	$	n
4			$	n	a		a
5				a	n		$
6				$	a		
7					$		

Constructie van de suffixtabel

Eerst de suffixboom construeren (als daar tijdelijk genoeg geheugen voor is) en daarna inorder overlopen om ze gerangschikt te hebben. Beide deeloperaties zijn $O(t)$ dus de constructie is ook $O(t)$.

De langste Gemeenschappelijke Prefix

De LGP-tabel is een belangrijke hulpstructuur die nuttig is bij het gebruikt van suffixtabellen.

Voor een gegeven suffix $suff_i$ is $LGP[i]$ de lengte van het langste gemeenschappelijke prefix van $suff_i$ met zijn opvolger in alfabetische volgorde $opvolger(suff_i)$.

De LGP-tabel behorend bij de suffixtabel op de vorige blz:

i	0	1	2	3	4	5	6
H[i]	$\perp$	0	1	3	0	0	2

De opbouw van de LGP-tabel

- We beginnen met het suffix $suff_i$ met $i = 0$.
- In de SA zoeken we de opvolger, dus we moeten de j bepalen met $SA[j] = 0$ zodat we $suff_{SA[j+1]}$ kennen.
- We bepalen het langste gemeenschappelijk suffix op de gewone manier: startend met $l = 0$ berhogen we l tot $T[i + l]$ niet meer overeenkomt met de overeenkomstige waarde van de opvolger. Op dat ogenblik is $l = LGP[i]$
- Voor de verdere waarden van LGP maken we gebruik van het staartprincipe. We tonen aan dat voor $i < t - 1, LGP[i + 1] \leq LGP[i] - 1$
  - stel dat $LGP[i] = l > 0$
  - Nu is $staart(suff_i) = suff_{i+1}$
  - Vermits de opvolger van $suff_i$ een string is groter dan $suff_i$, is $staart(opvolger(suff_i))$ een string groter dan $suff_{i+1}$ die een prefix van lengte $l - 1$ gemeenschappelijk heeft met $suff_{i+1}$
  - Hij hoeft niet zelf de opvolger van $suff_i + 1$ te zijn, maar minstens even groot en heeft dus hoogstens evenveel letters gemeen met $suff_{i+1}$
  - Bijgevolg moeten we, om $LGP[i+1]$ te vinden, pas beginnen vergelijken vanaf $suff_{i+1}[l-1]$, wat overeenkomt met $T[i + 1]$.
- We hebben $O(t)$ vergelijkingen (gelijke karakters: l verhogen en $i + l \leq t$ dus hoogstens t keer, ongelijke: i verhogen en dit gebeurt $t - 1$ keer).

Suffixboom omzetten in een suffixtabel

Om dit te doen maken we gebruik van de volgende eigenschap:

Is k een expliciete inwendige knoop van de suffixboom van T. Dan is er een suffix $suff_i$ waarbij $LGP[i] = l$ zodanig dat de padstring voor k het prefix met lengte l is van $suff_i$. Omgekeerd hoort bij elke suffix $suff_i$ een knoop met padstring $T[i, ..., i + LGP[i] - 1]$

Patronen opzoeken in de suffixtabel (letterlijk overgetypt, snap er effe geen klote van)

Dit gebeurt via binair zoeken wat $O(\lg t)$ *string* vergelijkingen vereist (dus $O(p \lg t)$ karaktervgl). We kunnen hier $O(p + \lg t)$ karaktervergelijkingen maken door de volgende principes:

1. **Nutteloze testen bij binair zoeken vermijden door gemeenschappelijke suffixen**
   - Binair zoeken gebruikt linkerindex l, rechter r en middelste $c = \lfloor (l+r)/2 \rfloor$ in suffixtabel A
   - Stel p_l is de lengte van het langste gemeenschappelijk prefix van P en suffix $A[l]$ en p_r dat met $A[r]$ met p_{min} het minimum van de 2
   - P heeft dan een prefix met lengte p_{min} gemeen met alle suffixen tussen l en r, zodat de stringvergelijking tussen P en het middelste suffic $A[c]$ kan beginnen bij karakter $P[p_{min}]$
   - Dit wordt $O(t + \lg p)$ maar in slechtste geval nog steeds $O(p \lg t)$
2. **Bijkomende gegevens op voorhand bepalen om in slechtste geval ook $O(t + \lg p)$ te hebben**
   - Om p_l en p_r te bepalen moet men de karakters van P testen tot 1 voorbij het *maximum* van die twee waarden, $P[p_{max}]$, zodat alle karakters an $P[p_{min}]$ tot en met $P[p_{max}]$ dan ook reeds getest zijn
   - 1 redundante test per herhaling van binair zoeken betekent dat de volgende test op P moet beginnen bij $P[p_{max}]$
   - Bij elke iteratie gebruiken we de lengte $p_{l,c}$ (of $p_{c,r}$, wat analoog is) van het langste gemeenschappelijk prefix van suffix $A[l]$ en het middelste suffix $A[c]$
   - Deze waarden kunnen allemaal op voorhand bepaald worden, ($O(t)$ waarden) en ze kunnen in $O(t)$ afgeleid worden uit de suffixtabel
   - De bijkomende gegevens worden als volgt bepaald:
     - Als $p_l = p_r$ dan is $p_{min} = p_{max}$ en is er slechts 1 redundante test
     - Als $p_l > p_r$ hebbenn we 3 mogelijkheden:
       * $p_{l,c} > p_l$ dan zijn de eerste p_l karakters van P en $A[c]$ gelijk, en aangezien P rechts van l ligt is het patroonkarakter $P[p_l]$ groter dan het karakter van $A[l]$ op dezelfde positie, dat echter gelijk is aan het overeenkomstig karakter van $A[c]$. Dus we weten dat P rechts ligt van c, zonder enige test. We maken l gelijk aan c, terwijl p_l niet verandert.
       * Als $p_{l,c} < p_l$ dan is het karakter $A[c][p_{l,c}]$ groter dan het overeenkomstig karakter van $A[l]$, dat gelijk is aan het overeenkomstig karakter van P. We weten dus dat P links ligt van c zonder enige test. We maken nu $r = c$ en $p_r = p_{l,c}$
       * Als $p_{l,c} = p_l$ dan komen de eerste p_l karakters van P en $A[c]$ reeds overeen. We moeten ze echter verder vergelijken vanaf $P[p_l]$, waarin $p_l = p_{max}$, om te beslissen of P links of rechts van c ligt. Enkel de eerste test is redundant, en als er meer testen gebeuren wordt p_{max} navenant groot. Het laatst geteste karakter van P is dus steeds $P[p_{max}]$
     - Als $p_l < p_r$, analoog vorige
   - Er zijn dan maximaal p niet-redundante testen op P mogelijk, en elk van de $O(\lg t)$ iteraties van binair zoeken verricht hoogstens 1 redundante test

12.3 Testzoekmachines

Hier komt het artikel over inverted files

13 NP

13.1 Complexiteit: P en NP

Problemen worden onderverdeeld in *complexiteitsklassen*, naargelang hun uitvoeringstijd of geheugenvereiste:

P (polynomiaal)

De klasse P bevat alle problemen waarvan de uitvoeringstijd begrensd wordt door een *veelterm* (bv $O(mn^2)$), bij uitvoering op een 'realistisch' computermodel (dit computermodel heeft een polynomiale bovengrens voor het werk dat in 1 tijdseenheid kan verricht worden). Alle problemen in P worden als *efficiënt oplosbaar* beschouwd.

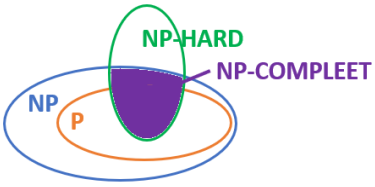
Eigenschappen: de efficiëntie blijft hier onafhankelijk van het computermodel en de klasse P is 'gesloten': een som of product van twee veeltermen is ook een veelterm, dus combinatie van meerdere P problemen geeft ook een P probleem.

NP (niet-deterministisch polynomiaal)

Niet-deterministische computer = comuter met oneindig veel processoren die elk per tijdsstap k andere processen kan aanspreken. Dit levert in t tijdsstappen op dat we t^k processoren aan het werk hebben. (elke processor kan hun deel van het probleem oplossen)

Als bij een beslissingsprobleem (ja-nee vraag: "bestaat er een...") het opsplitsen can de verzameling kandidaten én het controleren van 1 van de kandidaten beide in polynomiale tijd kan gebeuren, dan kan een niet-deterministische computer antwoord geven in polynomiale tijd. Zo'n problemen behoren tot de **NP-klasse**.

Situeren van N en NP problemen



Elke P probleem is natuurlijk ook NP.

Als we een probleem X kunnen **reduceren** naar een probleem Y dan noemen we Y minstens even zwaar als X (bv koppelingsprobleem omzetten in stroomnetwerkprobleem). Er zijn natuurlijk problemen die minstens even zwaar zijn als *elk* NP-probleem, deze noemt met **NP-hard**. Dit zijn problemen die de NP klasse als het ware omvatten.

Er zijn NP-harde problemen die zeker in NP zitten, deze noemt men **NP-compleet**. *Wikipedia-definitie:* In formele zin is een probleem NP-compleet als en slechts als het probleem tot de complexiteitsklasse NP behoort EN elk ander probleem in NP in polynomiale tijd naar dit probleem kan worden gereduceerd.

Wat zijn we met NP-complete problemen te kennen?

Vanaf dat we 1 NP-compleet probleem kenden (**SAT**), konden we dit probleem reduceren naar andere problemen om zo aan te tonen dat die andere problemen ook NP-compleet zijn. Als we nu maar ook 1 NP-compleet probleem efficiënt kunnen oplossen, kunnen we dat ook doen met de anderen. Er is nog nooit zo een manier gevonden.

Als we nu een probleem hebben en kunnen bewijzen dat het NP-compleet is, dan weten we dat er geen efficiënte oplossing is. Wel zijn er dan een aantal mogelijkheden:

- Als de afmetingen van het probleem klein zijn, kan men toch alle mogelijkheden onderzoeken (met backtracking), en daarbij trachten zoveel mogelijk te beperken
- Misschien bestaan er efficiënte algoritmen om speciale gevallen van het problemen op te lossen
- De begrenzing is voor het slechtste geval, het is mogelijk dat het gemiddeld meevalt
- Benaderende algoritmen gebruiken die de vereisten voor de oplossing relaxeren
- Efficiënte *heuristische methoden* gebruiken die meestal een goede maar niet noodzakelijk optimale oplossing vinden

13.2 NP-complete problemen

Een kort overzicht van enkele belangrijke NP-complete problemen.

13.2.1 Het basisprobleem: SAT (en 3SAT)

= Een of andere constructie vinden om aan alle voorwaarden te volgoen

Gegeven een verzameling logische variabelen $\chi = \{x_1, ..., x_{|\chi|}\}$ en een collectie logische uitspraken $F = \{f_1, ..., f_{|F|}\}$. Elke uitspraak bestaat uit atomaire uitspraken x_i of de inverse $\overline{x_i}$ samengevoegd door of-operaties zoals:

$$F_j = x_2 \vee \overline{x_5} \vee x_7 \vee x_8$$

Kan men waarden toekennen aan de variabelen, zodat al deze uitspraken tegelijk waar zijn?

Cook heeft aangetoond dat elk probleem uit NP polynomiaal kan gereduceerd worden tot SAT, dus **NP-compleet**.

We zien in dat elke uitspraak van meer dan 3 atomen kan herleid worden naar een reeks uitspraken met elk drie atomen zodat de uitspraak waar is als en slechts als de 2-uitspraken waar zijn. Dit doen we iteratief door telkens de eerste twee atomen af te splitsen en een nieuwe variabele toe te voegen.

$$x_2 \vee \overline{x_5} \vee x_n$$

$$x_7 \vee x_8 \vee \overline{x_n}$$

Dit leidt tot het **3SAT** probleem, waarbij elke uitspraak hoogstens 3 atomen mag hebben.

Nu gaan we een aantal problemen bespreken waar we 3SAT naar kunnen reduceren

13.2.2 Graafproblemen

13.2.2.1 Vertex cover

Gegeven: ongerichte graaf. **Gevraagd:** een kleinste groep knopen die minstens 1 eindknoop van elke verbinding bevat (verwant met edge cover).

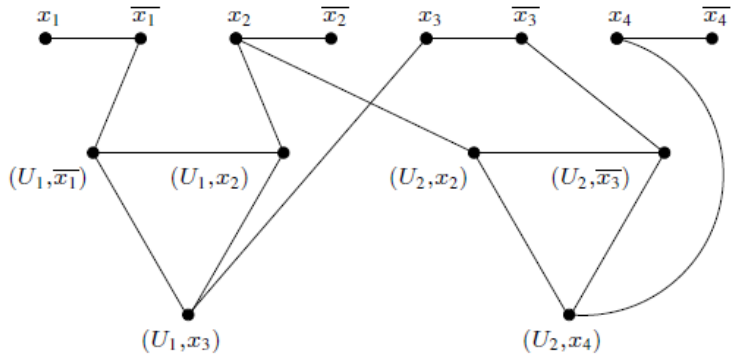
Hebben we een instantie van 3SAT dan construeren we de graaf erbij als volgt:

- Voor elke logische waarde maken we 2 knopen (waarde en inverse waarde) en verbinden die met elkaar
- Voor elke uitspraak maken we 3 knopen, 1 voor elk atoom, en deze verbinden we met elkaar en met bijhorende atoom uit het vorige

We stellen de graaf op voor

$$U_1 = \overline{x_1} \vee x_2 \vee x_3$$

$$U_2 = x_2 \vee \overline{x_3} \vee x_4$$



Een vertex cover zoeken voor zo'n graaf.

- Voor elke logische varianele moeten we al zeker een van de twee knopen x_i of $\overline{x_i}$ opnemen om de verbinding ertussen te hebben
- Voor elke uitspraak moeten we zeker 2 knopen opnemen
- Er zijn dus minstens $|\chi|/2|F|$ knopen nodig

Dit aantal is voldoende als en slechts als het 3SAT-probleem een oplossing heeft: de cover geeft aan welke waarde ik moet geven aan de variabelen, de toegevoegde knopen in mijn vertex cover zijn waar in het 3SAT probleem.

13.2.2.2 Dominating set

Gegeven een ongerichte graaf. **Gevraagd** een kleinste groep knopen zodat elke andere knoop met minstens 1 knoop van de groep knopen verbonden is

13.2.2.3 Graph coloring

Vertex covering

gegeven een ongerichte graad. Gevraagd kleur de knopen met een minimum aantal kleuren, zodat de eindknopen van elke verbdingin een verschillende kleure heeft. Hier moet men beroep doen op **heuristische methoden**.

Edge covering

Gegeven een ongerichte graaf. **Gevraagd** kleur de verbindingen met een minimum aantal kleuren, zodat verbindingen met een gemeenschappelijke eindknoop een verschillende kleur hebben.

13.2.2.4 Clique

Gegeven een ongerichte graaf. **Gevraagd** de grootste groep knopen die allemaal met elkaar verbonden zijn. Heuristische methoden zoals simulated annealing werken vaak redelijk goed.

SAT herleiden tot clique

Voor elk voorkomen van een atoom in een uitspraak maken we een bijhorende knoop (F_i, x_j) dan wel $(F_i, \overline{x_j})$ en leggen volgende verbindingen tussen knopen horende bij *verschillende uitspraken* $(j \neq i)$ als de atomen elkaar niet uitsluiten:

- (F_i, x_k) met (F_j, x_l) voor alle k en l
- $(F_i, \overline{x_k})$ met $(F_j, \overline{x_l})$ voor alle k en l
- (F_i, x_k) met $(F_j, \overline{x_l})$ als en slechts als $k \neq l$

Een clique kan hoogstens $|F|$ elementen bevatten, want knopen horende bij dezelfde uitspraak zijn niet verbonden. Een clique met exact $|F|$ elementen leidt duidelijk tot een oplossing van SAT (maak alle atomen horende bij leden van de clique waar; de waarde van andere logische variabelen mag willekeurig gekozen worden).

13.2.2.5 Independent set

Gegeven een ongerichte graaf. **Gevraagd** de grootste groep knopen zonder onderlinge verbindingen.

Is nauw verwant met kleuren van knopen (verdeelt een graaf in een klein aantal groepen van knopen die onafhankelijk zijn), en identiek aan zoeken van een clique in de complementaire graaf

13.2.2.6 Hamilton circuit/path

Gegeven een al dan niet gerichte graaf. **Gevraagd** bestaat er een circuit dat elke knoop eenmaal bevat?

De enige oplossingsmethode is backtracking.

13.2.3 Problemen bij verzamelingen

13.2.3.1 Minimum cover

Gegeven een verzameling S en een collectie deelverzamelingen C van S. Gevraagd de kleinste deelverzameling C′ van C zodat elk element van S tot minstens een van de deelverzamelingen uit C′ behoort.

Een aantal andere problemen kan in deze vorm gegoten worden:

- **Vertex cover:** S is de verzameling verbindingen; C is de verzameling knopen, waarbij elke knoop wordt opgevat de verzameling van de verbindingen die hij heeft.
- **Edge cover**
- **Dominating set:** S is de verzameling knopen. Voor elke knoop is er een element van C bestaande uit de knoop met al zijn buren.

Minimum cover kan efficiënt opgelost worden wanneer de deelverzamelingen in C niet meer dan twee elementen bevatten. Als elke C ∈ C exact twee elementen bevat is het duidelijk equivalent met edge cover; als er C ∈ C zijn met maar ´e´en element kan men deze (voorlopig) schrappen, het edge coverprobleem oplossen en eventueel nodige singletons terug toevoegen.

SAT herleiden tot minimum cover

Neem $S = F \cup \chi$. Voor elke logische variabele x_i zijn er twee elementen van C:

$$C_{x_i} = \{x_i\} \cup \{f \in F : x_i \in f\}$$

$$C_{\overline{x_i}} = \{x_i\} \cup \{f \in F : \overline{x_i} \in f\}$$

Het SAT-probleem heeft een oplossing als en slechts als er een minimum cover is van grootte $|\chi|$.

Als S uiteenvalt in twee verzamelingen

Het gebeurt vaak dat S uiteenvalt in twee verzamelingen $S = F \cup \chi$ waarbij elk element van C juist 1 element van χ bevat en we een cover met grootte $|\chi|$ zoeken. We kunnen χ opvatten als een lijst van variabelen x_i die een waarde moeten krijgen, waarbij een element $c \in C$ overeenkomt met een keuze van een waarde die x_i waarvoor $x_i \in c$.

Een voorbeeld hiervan wordt gegeven door betegelingsproblemen (tiling puzzles). Hierbij hebben we een speelbord F met vakjes, vaak een deel van een oneindig raster zoals een schaakbord, en een aantal tegels van verschillende of gelijke vorm. Bedoeling is om het bord te bedekken met de tegels. Met elke c ∈ C komt dan een toegelaten positie van een tegel op het bord overeen: c bevat zowel de tegel als de bedekte vakjes.

Een minimum coverprobleem kan beschreven worden door een 0/1-matrix met |S| rijen en |C| kolommen. Als een probleem klein genoeg is kan het opgelost worden met backtracking.

13.2.3.2 Subset sum

Gegeven een verzameling elementen, elk met een positieve gehele grootte, en een positief geheel getal k. **Gevraagd** estaat er een deelverzameling van die elementen, zodat de som van hun grootten gelijk is aan k?

13.2.3.3 Partition

Gegeven een zak (een verzameling waarbij meerdere keren hetzelfde element kan voorkomen) van positieve gehele getallen. **Gevraagd** kan die opgesplitst worden in twee deelverzamelingen, zodat de som van de grootten van de ene gelijk is aan de som van de grootten van de andere?

13.3 Netwerken

13.3.0.1 TSP

Gegeven een aantal steden, en de afstand tussen elk paar steden. **Gevraagd** het kortste circuit dat elke stad eenmaal bevat. Heuristische methoden zijn aangewezen, en het resultaat is vaak zeer goed.

13.3.0.2 Longest path

Gegeven een gerichte of ongerichte gewogen graaf, en twee knopen s en t. **Gevraagd** de langste simpele weg (zonder lussen) van s naar t.

13.3.1 Gegevensopslag

13.3.1.1 Bin packing

Gegeven een verzameling van n objecten met afmetingen $s_1, ..., s_n$, en een verzameling van m bakken met capaciteiten $c_1, ..., c_m$. **Gevraagd** alle objecten op te slaan in zo weinig mogelijk bakken. Er kunnen beperkingen opgelegd worden aan oriëntatie en plaatsing van de objecten.

13.3.1.2 Knapsack

Gegeven een verzameling van n objecten, met elk een afmeting en een waarde, en een knapzak met zekere capaciteit. **Gevraagd** objecten in de knapzak te stoppen, zonder zijn capaciteit te overschrijden, zodat hun totale waarde zo groot mogelijk is.

14 Metaheuristieken

Metaheuristieken zijn vuistregels bij het zoeken naar een oplossing van een probleem. Ze garanderen niet altijd een oplossing, maar ze versnellen wel de zoektocht.

14.1 Combinatorische optimalisatie

Optimalisatie houdt in dat we uit een verzameling S een individu halen dat een beste element is uit die verzameling. S is een eindige verzameling strings die aan een aantal voorwaarden voldoen. Het beste individu wordt bepaald door *evaluatiefunctie* f.

Deze problemen kunnen we in principe aflopen met backtracking, dus de methodes die we hier gaan beschrijven zijn heuristieken om het beste individu te benaderen voor een probleem dat te groot is voor backtracking.

Metaheuristieken nemen daarvoor een steekproef van S, berekenen voor elk individu uit de steekproef f en bewaren het beste individu, tot een stopvoorwaarde bereikt wordt.

De verschillende metaheuristieken verschillen in de manier waarop individuen worden uitgekozen en bekeken.

14.2 Constructie van een enkel individu uit S

De methoden die we gaan bekijken werken niet voor alle mogelijke problemen ze moeten doen aan bepaalde voorwaarden zoals: het moet zinvol zijn om op zoek te gaan naar betere individuen in de buurt van een gegeven individu. Bij sommigen in het zinvol om bij het opbouwen van nieuwe individuen van S, uit te gaan van reeds gevonden individuen, en soms niet, dan wordt er gewoon een random steekproef genomen.

We bespreken eerst de aanmaak van een nieuw element (voor we bespreken hoe we nuttige informatie kunnen halen uit reeds gevonde individuen)

Als men een nieuw element van de grond opbouwt zijn er twee belangrijke mogelijkheden:

1. Constructie: het individu wordt opgebouwd uit componenten. Deze componenten komen overeen met de letters van de strings in S. In tegenstelling tot backtracking gaat men hier niet alle uitbreidingsmogelijkheden bekijken maar kiest men er één uit. Dit gebeurt at random, al kan er een heuristiek zijn die de keuze be¨ınvloedt.
2. Individuen worden rechtstreeks aangemaakt

14.2.1 Constructie vanaf de grond

Sommige methodes hebben verschillende startindividuen nodig (als het bv niet meer zinvol wordt om aan een component toe te voegen). Dan is het handig om een zekere **randomisatie** te hebben. Als er geen heuristiek is maakt men een *ongewogen keuze*, alle overblijvende mogelijkheden zijn dan even waarschijnlijk. Als er wel een heuristiek is (in de vorm van een kwaliteitsfunctie) kan men 2 mogelijkheden toepassen:

1. **Shortlisting:** de beste kandidaten worden genomen en daaruit wordt random gekozen
2. **Gewogen keuze:** elke keuze krijgt een gewicht w_i. De kans $p(i)$ dat i gekozen wordt is dan evenredig met w_i

$$p(i) = \frac{w_i}{\sum_j w_j}$$

14.2.2 Afleiden uit andere individuen

Voor het kiezen van een nieuw individu uit S, uitgaande van andere individuen die men al bekeken heeft, zijn er fundamenteel er drie methodes:

1. Kleine wijzigingen aanbrengen in 1 reeds bekeken individuen s van S (lokaal zoeken)
2. Kruisen van 2 bekeken individuen (genetische algoritmen)
3. Het vermengen van grote hoeveelheden al bekeken individuen (niet vaak gebruikt)

14.3 Lokaal versus globaal zoeken

Een belangrijke factor bij zoeken is of dat het zin heeft om een individu te proberen verbeteren door het aanbrengen van kleine veranderingen. We gebruiken hier het begrip **omgeving**: voor elke $s \in S$ is $N(s)$ de omgeving van s die bestaat uit s zelf en alle strings uit S die door 1 kleine wijziging uit s kunnen bekomen worden.

We moeten hier wel met verschillende dingen rekening houden zoals als bv alle strings niet even lang zijn, moeten er kleine wijzigingen zijn die de string langer of korter maken. Ook moet men er natuurlijk altijd voor zorgen dat de kleine wijzigingen weer een geldig individu uit S opleveren, zo niet 'repareert' men het of wordt het weggegooid. Of men kan ze toelaten maar een zeer grote evaluatiewaarde geven waardoor ze niet toegelaten worden voor de eindoplossing.

Het wordt pas echt interessant als het zinvol is om lokale verbeteringen iteratief toe te passen, m.a.w. als het vaak voorkomt dat, uitgaande van een individu $s_0 \in S$, we een rij van individuen $s_0, s_1, ..., s_n \in S$ kunnen construeren zodanig dat voor alle $i = 0, ..., n+1$, $s_{i+1} \in N(s_i)$ en $f(s_{i+1}) < f(s_i)$. Deze rij eindigt zeker in een **lokaal minimum** s_n zodat.

Het vinden van zo'n rij wordt **lokale optimalisatie** genoemd. In een aantal gevallen is het makkelijk om, gegeven s, een element in $N(s)$ te vinden met de beste f-waarde. Dit noemen we **steepest slope desscending** (of indien f zo groot mogelijk moet zijn, **steepest hill climbing**).

Exploratie

Bij problemen waar er maar 1 lokaal minimum is is dit ook het beste individu. Het kan zijn dat er veel lokale minima in S kunnen zijn en dat er geen enkele garantie is dat al die lokale minima dezelfde f-waarde hebben. Bovendien kan het construeren van zo'n rij zeer veel tijd in beslag nemen, vooral als de startwaarde s0 zeer slecht is. We moeten dus ontsnappen aan dit lokaal minimum, **exploratie**. Dit kan op 2 manieren:

1. Exploratie door helemaal opnieuw te beginnen. Hierbij worden individuen uit S random opgebouwd
2. Exploratie door grotere wijzigingen aan te brengen aan gekende individuen

14.4 Methodes zonder recombinatie

(Dit gaat over methodes die gebruik maken van random zoeken maar dan met een aantal verbeteringen, 2 voorbeelden: simulated annaeling en tabu search). Aan het gewone random zoeken kunnen twee elementen toegevoegd worden die de efficiëntie kunnen opvoeren:

1. Lokaal zoeken verbetert vaak random aangemaakte individuen
2. Voor sommige problemen zijn er heuristieken die constructie van individuen kunnen sturen

14.4.1 Simulated annaeling

Bij simulated annaeling is er een kleine wijziging ten opzicht van het random kiezen van s' uit $N(s)$ (en bewaren als het beter is): de *kans* dat s' gekozen wordt is nu niet enkel afhankelijk van de evaluatiewaarden van s en s' maar ook van een zogenaamde **temperatuur** T. Een vaak gekozen functie is:

$$\rho(T, f(s'), f(s)) = exp\left(\frac{f(s) - f(s')}{T}\right)$$

De begintemperatuur wordt vrij hoog gekozen, zodat exploratie bevorderd wordt. Het klassieke patroon is om T te laten dalen tot nul, zo wordt de eindfase gelijk aan lokaal zoeken.

Voor voorbeelcode zie blz 176

14.4.2 Tabu search

In zijn eenvoudigste vorm houdt Tabu Search een taboelijst bij: een lijst van al gecontroleerde individuen die niet opnieuw mogen bezocht worden. De lijst wordt in grootte beperkt en het oudste wordt verwijderd als het vol zit.

Het gebruik van de taboelijst moet ervoor zorgen dat de methode voldoende exploreert: met een taboelijst met grootte 0 wordt Tabu Search gewoon lokale verbetering. De taboelijst moet echter zo groot zijn dat men ver genoeg uit de buurt van een lokaal optimum kan komen om het *basin of attraction* te verlaten. Dit is meestal niet haalbaar, zodat men vaak taboelijsten gebruikt die geen individuen bevatten maar wel eigenschappen.

Voor voorbeelcode individu zie blz 177, voor voorbeelcode component zie blz 178

14.5 Genetische algoritmen

14.5.1 Kruising

Bij kruising gaan we twee individuen vermengen. Dit is alleen zinvol als er reden is om aan te nemen dat zo'n vermenging kan leiden tot individuen die beter zijn dan de ouders. Onderliggend is meestal de notie dat er combinaties zijn van componenten die 'goed' kunnen zijn, ook al heeft het individu een slechte f-waarde. Er zijn 2 mogelijkheden:

1. **Gerichte kruising:** Als men een idee heeft van de algemene structuur van individuen dan kan men deze structuur gebruiken om zinvolle combinaties van componenten te vinden.
2. **Blinde kruising:** De meestgebruikte is die waarbij een individu wordt voorgesteld als een bitreeks van bepaalde lengte. Een manier om toch nog enige structuur te behouden is om ervoor te zorgen dat verwante bits dicht bij mekaar staan. Kruising van twee bitstrings gebeurt dan door de suffixen van de twee strings om te wisselen.

Vermits bij kruising twee individuen (de ouders) nodig zijn is moeten we verschillende individuen samen bijhouden. Men spreekt van een *populatie* van individuen. Bij genetische algoritmen spreekt men van generaties. Gegeven een populatie gaat men over naar een volgende generatie. De overgang verloopt in twee stappen:

1. *Survival of the fittest:* de populatie wordt uitgedund door een aantal individuen te verwijderen.
2. Gebaseerd op de overblijvende individuen wordt populatie terug aangevuld tot haar oorspronkelijke grootte door kruising en mutatie.

Twee belangrijke verfijningen:

- Als componenten van individuen met mekaar interageren dan kan het voordelig zijn om de kans te vergroten dat interagerende componenten bij elkaar blijven.
- Kruising moet niet alleen zorgen voor betere individuen, maar ook voor diversificatie en zo voor exploratie, iets wat door de voorgaande techniek beperkt wordt. Om toch meer exploratie te krijgen gaat men soms gebruik maken van niching, waarbij de fitheid van individuen verm

14.6 Vermenging

Bij vermengingsmethodes gaat men niet meer twee individuen kruisen. Wel gaat men uit van de globale eigenschappen van een populatie om de volgende te nemen. Hierbij kan men werken op twee niveau's:

1. **Componentniveau:** Hierbij gaat men ervan uit dat het succes van een individu afhangt van de individuele componenten die het bevat.
2. **Combinatieniveau:** Hierbij gaat men ervan uit dat het succes van een individu afhangt van combinaties van componenten die het bevat.

14.6.1 Recombinatie op componentniveau

Er zijn twee belangrijke, bijna identieke, metaheuristieken die gebruik maken van van dit soort recombinatie: *Gene Pool Recombination (GPR)* en *Ant Colony Optimisation (ACO)*. Zoals een genetisch algoritme werken beide met een populatie die telkens vervangen wordt door een nieuwe generatie. Bij beide gaat men een fitheid toekennen niet alleen aan individuen maar ook aan componenten.

Bij het opbouwen van een nieuwe populatie gaat men ongeveer te werk zoals bij een gerandomiseerd inhalige constructie waarbij de fitheid w_i van een component i dient als gewicht voor een gewogen keuze. De gewogen random keuze kiest uit een verzameling V van componenten een element waarbij de waarschijnlijkheid om een component i te kiezen evenredig is met w_i

$$p(i) = \frac{w_i}{\sum_{j \in V} w_j}$$

Het verschil tussen GPR en ACO zit in de keuze van de gewichten w_i:

- Bij GPR wordt alleen rekening gehouden met de laatste generatie.
- Bij ACO speelt het oude gewicht wel een rol. Bovendien worden slechte individuen uit de laatste generatie niet uitgesloten. Wel krijgt elke s een kwaliteitswaarde $F(s)$. Deze waarde is groter naarmate $f(s)$ kleiner is. Het nieuwe gewicht voor een component i is dan

  $$w_i \leftarrow (1-\rho)w_i + \sum_{s \ bevat \ i} F(s)$$

  Hierbij is ρ een vergeetfactor

14.6.2 Recombinatie op combinatieniveau

Deze vorm van recombinatie vereist dat het probleem zich ertoe leent. Daarvoor zijn er twee eigenschappen belangrijk:

1. Opeenvolgende letters in een individu moeten een sterk onderling verband vertonen
2. Een deelstring van een goed individu gebruiken in een ander individu verhoogt de kans dat het andere individu ook goed is

We stellen dit voor als een graaf. De componenten van onze individuen worden daarbij de knopen van een ongerichte gewogen graaf. Indien twee componenten samen kunnen voorkomen in een individu is er een verbinding.

We hebben nu een aantal mieren die proberen individuen uit S te construeren. Zo'n individu wordt voorgesteld als een pad met een bepaald vertrekpunt. Elke mier vertrekt vanuit dit punt en construeert een weg als volgt:

1. Op een bepaald punt aangekomen bepaalt de mier of het pad dat ze al heeft gedefinieerd een individu definieert, Zo ja, dan stopt ze.
2. Zo nee bepaalt ze welke volgende componenten nog in aanmerking komen.
3. Als er geen mogelijkheden meer zijn stopt ze
4. Anders kiest ze er 1 door een gewogen random keuze met als gewichten de gewichten van de verbindingen

We spreken hier van *feromonen* in de plaats van gewichten.

Nadat elke mier geprobeerd heeft om een individu te construeren worden de hoeveelheden feromonen aangepast. Eerst wordt het feromoon bij elke verbinding vermenigvuldigd met een factor $1 - \rho$. Hier is ρ de vergeetfactor. Daarna voegt elke mier die een individu s heeft gevonden een hoeveelheid feromonen $F(s)$ toe aan elke verbinding die ze gebruikt heeft. Ook hier is $F(s)$ de kwaliteitswaarde van het individu s. In formule wordt dit

$$\tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij} + \sum_{s \in A_{ij}} F(s)$$

$\forall i, j$ waar A_{ij} de verzameling individuen is gevonden door mieren die verbinding v_{ij} gebruikt hebben.

Bij het begin van het algoritmen kan men alle waarden τ_{ij} gelijk nemen. Wanneer men echter een heuristiek heeft kan men andere beginwaarden nemen, zodat bepaalde verbindingen initieel bevorderd worden.