

Gevorderde algoritmen

Bert De Saffel

Master in de Industriële Wetenschappen: Informatica Academiejaar 2018–2019

Gecompileerd op 16 december 2018

Inhoudsopgave

I	Gegevensstructuren II	5
1	Efficiënte zoekbomen	6
1.1	Rood-zwarte bomen	6
1.1.1	Operaties	7
1.2	Splaybomen	7
1.2.1	Operaties	7
1.2.2	Performantie	9
1.3	Randomized Search Trees	9
2	Toepassingen van dynamisch programmeren	10
2.1	Langste gemeenschappelijke deelsequentie	10
3	Uitwendige gegevensstructuren	11
3.1	B-trees	11
3.1.1	Definitie	11
3.1.2	Eigenschappen	11
3.1.3	Operaties	12
3.1.4	Varianten	13
3.2	Uitwendige Hashing	13
3.2.1	Extendible hashing	13
3.2.2	Linear hashing	14
4	Meerdimensionale gegevensstructuren	15
4.1	Projectie	15
4.2	Rasterstructuur	15
4.3	Quadrees	15

4.3.1	Point quadtrees	15
4.3.2	Point-Region quadtrees	16
4.3.3	k-d trees	16
5	Samenvoegbare heaps	17
II	Grafen II	19
6	Toepassingen van diepte-eerst zoeken	20
6.1	Enkelvoudige samenhang van grafen	20
6.1.1	Samenhangende componenten van een ongerichte graaf	20
6.1.2	Sterk samenhangende componenten van een gerichte graaf	20
6.2	Dubbele samenhang van ongerichte grafen	21
6.3	Eulergraaf	21
7	Kortste Afstanden II	23
7.1	Kortste afstanden vanuit één knoop	23
7.1.1	Algoritme van Bellman-Ford	23
7.2	Kortste afstanden tussen alle knopenparen	24
7.2.1	Het algoritme van Johnson	24
7.3	Transitieve sluiting	25
8	Stroomnetwerken	26
8.1	Maximale stroomprobleem	26
8.2	Verwante problemen	27
8.3	Meervoudige samenhang in grafen	27
9	Koppelen	28
9.1	Koppelen in tweeledige grafen	28
9.1.1	Ongewogen koppeling	28
III	Strings	30
10	Gegevensstructuren voor strings	31
10.1	Digitale zoekbomen	31
10.2	Tries	31
10.2.1	Binaire tries	31

10.2.2 Meerwegtries	32
10.3 Variabelelengtecodering	33
10.4 De Eliascodes en de Fibonaccicode	34
10.5 Huffman codering	35
10.5.1 Opstellen decoderingsboom	35
10.5.2 Patricia tries	36
10.6 Ternaire zoekbomen	37
11 Zoeken in strings	39
11.1 Formele talen	39
11.1.1 Generatieve grammatica's	39
11.1.2 Reguliere uitdrukkingen	40
11.2 Variabele tekst	40
11.2.1 Eenvoudige methode	41
11.2.2 Knuth-Morris-Pratt	41
11.2.3 Boyer-Moore	42
11.2.4 Karp-Rabin	42
11.2.5 Zoeken met automaten	43
11.2.6 De Shift-AND-methode	44
12 Indexeren van vaste tekst	47
12.1 Suffixbomen	47
12.2 Suffixtabellen	48
12.3 Tekstzoekmachines	48
IV Hardnekkige problemen	50
13 NP	51
13.1 Complexiteit: P en NP	51
13.2 NP-complete problemen	52
13.2.1 Het basisprobleem: SAT (en 3SAT)	52
13.2.2 Graafproblemen	53
13.2.3 Problemen bij verzamelingen	54
14 Metaheuristieken	56
14.1 Combinatorische optimalisatie	56

14.2 Vooronderstellingen	57
14.3 Lokaal versus globaal zoeken	57

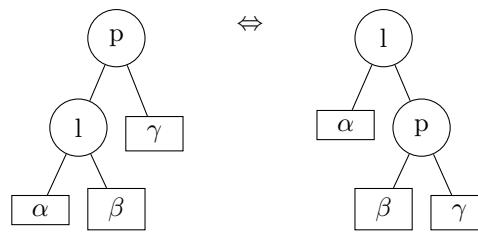
Deel I

Gegevensstructuren II

Hoofdstuk 1

Efficiënte zoekbomen

Rotaties wijzigen de vorm van de boom, maar behouden de inorder volgorde van de sleutels. Dit is nodig omdat de eigenschap van een binaire zoekboom steeds voldaan moet zijn, namelijk dat het linkerkind kleiner is, en het rechterkind groter, dan de ouder. Een rotatie is $O(1)$ omdat enkel pointers verplaatst moet worden.



1.1 Rood-zwarte bomen

$\overline{x_i}$

Een rood-zwarte boom is een **binaire zoekboom** waarbij bovendien:

- Elke knoop rood of zwart gekleurd is.
- Elke virtuele knoop zwart is. Een virtuele knoop is een ontbrekend kind (nullpointer), die geen gegevens bevatten maar wel een kleur hebben (zwart).
- De wortel zwart is (een rode wortel kan zonder problemen zwart gemaakt worden).
- Elke mogelijke weg vanuit een knoop naar een virtuele knoop evenveel zwarte knopen heeft (**zwarte diepte**).
- De hoogte h , kan uit de voorgaande definities, afgeleidt worden aangezien elke deelboom met wortel w en zwarte diepte z tenminste $2^z - 1$ inwendige knopen bevat.

$$1 + 2 + \dots + 2^{z-1} = 2^z - 1$$

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

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

1.1.1 Operaties

Zoeken is equivalent met een gewone binaire boom en is dus $O(\lg n)$. De interessante operaties zijn toevoegen en verwijderen, die beiden zowel bottom-up als top-down kunnen gebeuren:

- **Bottom-up.** Een bottom-up rood-zwarte boom zal eerst een knoop toevoegen of verwijderen, en nadien de boom herstellen.
 - **Toevoegen.** Een knoop toevoegen gebeurt op dezelfde manier als bij een normale binaire zoekboom. Een nieuwe toegevoegde knoop krijgt altijd een rode kleur, omdat de zwarte diepte herstellen moeilijker is. Bij het toevoegen van een knoop kunnen er zich zes gevallen voordoen, waarvan er drie het spiegelbeeld zijn van elkaar. Hier wordt verondersteld dat de ouder p van de toegevoegde knoop c het linkerkind is van grootouder g , en dus de broer b van p het rechterkind is van g .
 1. Is b rood? Maak p en b zwart en g rood. Als g een zwarte ouder heeft is het probleem opgelost. Anders krijgen we opnieuw één van de drie gevallen, waarbij het nu lijkt dat g de toegevoegde rode knoop is. De ligging van c ten opzichte van p heeft in dit geval geen impact.
 2. is b zwart, dan zijn er twee gevallen:
 - (a) Indien c aan de uitwendige kant ligt van p , liggen de drie knopen g , p en c op een lijn en moet er een rotatie naar rechts uitgevoerd worden. Deze rotatie wordt gevolgd door p zwart en g rood te kleuren.
 - (b) Indien c aan de inwendige kant ligt van p , dan moet enkel p en c naar links geroteerd worden, zodat we het vorige geval krijgen (c is nu wel de ouder van p).
 - **Verwijderen.** Ook wordt deze operatie eerst uitgevoerd zoals bij een normale binaire zoekboom. Indien de fysisch te verwijderen knoop rood is, is verwijderen eenvoudig aangezien de zwarte hoogte ongewijzigd blijft. **ToDo: dubbelzwart enz**
- **Top-down.** Een top-down rood-zwarte boom zal op de zoekweg ook al de boom herstellen.
 - **Toevoegen.** Op de weg naar beneden mogen we geen rode broers toelaten, aangezien de nieuwe knoop kind van beide kan zijn. Wanneer er op de zoekweg een zwarte knoop c met twee rode kinderen voorkomt, kan c rood gemaakt worden en zijn kinderen zwart, indien de ouder p van c ook rood is, en c ligt aan de buitenkant, moet p geroteerd worden, zodat p de ouder is van c en g , de oorspronkelijke ouder van p . Ligt c aan de binnenkant, dan wordt eerst c geroteerd, zodat deze de ouder wordt van p , gevolgd door een bijkomende rotatie rond c zodat c als kinderen p en g heeft. De knoop c wordt terug zwart gemaakt en g wordt rood gemaakt.
 - **Verwijderen.**

1.2 Splaybomen

Een splayboom is een normale binaire zoekboom, waarbij er een splayoperatie gedefinieerd is: een reeks van operaties zodat de meest recente opgevraagde knoop α in de wortel staat. Bij splaybomen heeft zoeken ook een bottom-up en top-down versie.

1.2.1 Operaties

- **Bottom-up.** De splayoperatie bij een bottom-up splayboom maakt gebruik van drie rotaties: **zig**, dat enkel uitgevoerd wordt indien de ouder p van α de wortel is van de boom. Deze rotatie is een normale rotatie zodat α de wortel wordt, en p één van de kinderen van α , **zig-zig**, dat

uitgevoerd wordt indien p nog een ouder g heeft en α aan de inwendige kant light. De eerste rotatie roteert α naar de buitenkant, zodat deze de ouder wordt van p , gevolgd door een rotatie die α de ouder maakt van p en g en **zig-zag**, dat uitgevoerd wordt indien α , p en g op dezelfde lijn liggen (dus α is uitwendig). De eerste rotatie roteert g , zodat p de ouder is van α en g , gevolgd door een rotatie die c de ouder maakt van p (die nog steeds g als kind heeft).

- **Zoeken.** Bottom-up zoeken is equivalent met een normale binaire zoekboom, gevolgd door de splayoperatie die de gezochte knoop tot wortel maakt. Als de gezochte sleutel niet bestaat, wordt de splayoperatie uitgevoerd op zijn voorloper of opvolger.
- **Toevoegen.** Toevoegen zal eerst de fysische knoop toevoegen aan de boom. Deze toegevoegde knoop wordt dan met de splayoperatie tot wortel gemaakt.
- **Verwijderen.** Eerst wordt de fysisch te verwijderen knoop verwijderd. De ouder van deze knoop wordt nu via de splayoperatie tot wortel gemaakt. De ouder is ook de laatste knoop op de zoekweg, zodat indien de te verwijderen knoop niet bestaat, nog steeds deze ouder tot wortel gemaakt wordt.
- **Top-down.** Een top-down splayboom maakt geen gebruik van rotaties, zodat er geen nood is aan ouderwijzers of stapels. De splayoperatie bij een top-down splayboom deelt de boom op in drie zoekbomen: L , die alle sleutels kleiner dan de sleutels in M bevat, en R , die alle sleutels groter dan de sleutels in M bevat. Initieel is M de oorspronkelijke boom en zijn R en L ledig. De zoekweg begint bij de wortel van M , en er wordt voor gezorgd dat de huidige knoop op de zoekweg steeds de wortel van M is, zodat op het einde van het zoeken, de gezochte sleutel (of zijn voorloper of opvolger) de wortel is van de uiteindelijke splayboom.

Top-down splaybomen kennen 6 operaties, waarvan er ook weer drie het spiegelbeeld zijn van elkaar. Veronderstel dat we vanuit een knoop p (die op dat moment de wortel van M is) naar het linkerkind c moeten, dan kunnen volgende gevallen zich voordoen:

1. De knoop c is de laatste knoop op de zoekweg. Dit komt enkel voor indien c gezocht wordt, of als hij geen kind heeft in de richting dat gezocht moet worden. Knoop p , samen met zijn rechtste deelboom wordt het nieuwe kleinste element in R . De linkse deelboom van p wordt de nieuwe M met c als wortel. Dit geval wordt ook weer **zig** genoemd, maar heeft dus wel een heel andere implementatie dan de zig bij bottom-up splaybomen.
2. Linkerkind c is niet de laatste knoop op de zoekweg.
 - (a) Indien afgedaald moet worden naar het linkerkind l van c , dan wordt eerst p en c naar rechts geroteerd, daarna wordt c , samen met zijn rechtse deelboom, het nieuwe kleinste element in R . De linkse deelboom van c wordt de nieuwe M . Dit geval heet ook opnieuw **zig-zig**, aangezien de knopen op één lijn liggen.
 - (b) Indien afgedaald moet worden naar het rechterkind r van c , dan wordt p , samen met zijn rechtste deelboom, het nieuwste kleinste element van R . Daarna wordt c het nieuwe grootste element in L , en de rechtste deelboom van c wordt de nieuwe M . Dit geval heet **zig-zag**.

Na deze operaties moeten de deelbomen nog samengevoegd worden, waarbij de wortel c is. Alle sleutels in de linkerdeelboom van c zijn groter dan die van L , dus kan L deze linkerdeelboom opnemen. Analooch zijn alle sleutels in de rechterdeelboom van c kleiner dan die van R , dus kan R deze rechterdeelboom opnemen. Het linkerkind van c wordt nu de wortel van L en het rechterkind wordt de wortel van R . De operaties verlopen nu als volgt:

- **Zoeken.** Deze operatie maakt de gezochte sleutel tot wortel, of indien deze niet gevonden wordt, door zijn voorloper of opvolger.
- **Toevoegen.** De voorloper of opvolger van de nieuwe sleutel wordt wortel, en de nieuwe knoop krijgt als linkerkind de voorloper met zijn linkse deelboom en als rechterkind zijn rechtste deelboom. Of alternatief, de nieuwe knoop krijgt als rechterkind de opvolger met zijn rechtste deelboom en als linkerkind zijn linkse deelboom.

- **Verwijderen.** Eerst wordt de sleutel gezocht, zodat die wortel wordt. Daarna wordt deze wortel verwijderd en worden de twee deelbomen terug samengevoegd.

1.2.2 Performantie

= Aantonen dat elke reeks van m operaties met maximaal n knopen een performantie van $O(m \lg n)$ heeft, waaruit volgt dat er een geamortiseerde efficiëntie van $O(\lg n)$ is per operatie.

Gebruik een potentiaalfunctie Φ_i , die aangeeft hoe goed de vorm van de splayboom is na een operatie i . Hoe hoger het potentiaal, des te beter de vorm van de splayboom. Elke knoop krijgt ook een gewicht $s_i =$ het aantal knopen in de deelboom waarvan hij wortel is. Vaak stelt men de potentiaalfunctie in als de som van de logaritmen van al deze gewichten:

$$\Phi = \sum_{i=1}^n \lg s_i$$

Stel nu t_i de werkelijke tijd van de i -de operatie, a_i de geamortiseerde tijd, en Φ_i de potentiaal na deze operatie, dan is $a_i = t_i + \Phi_i - \Phi_{i-1}$. De geamortiseerde tijd van een reeks van m operaties is dan de som van de individuele geamortiseerde tijden:

$$\sum_{i=1}^m a_i = \sum_{i=1}^m (t_i + \Phi_i - \Phi_{i-1})$$

In de som komen de meeste potentialen met tegengestelde tekens voor.

$$\sum_{i=1}^m a_i = \sum_{i=1}^m t_i + \Phi_m - \Phi_0$$

Stel nu c , de knoop die we zoeken, r_w de rang van de wortel en r_c de rang van knoop c in de originele boom.

- c is de wortel: Er zijn geen rotaties nodig en de potentiaal blijft ongewijzigd. Deze operatie is dan ook $O(1)$.
- c is niet de wortel: **ToDo: moeten we dit eigenlijk kennen?**

1.3 Randomized Search Trees

kans is klein dat hij dit vraagt. is vrij easy ook: gewoon per knoop een prioriteit genereren (met randomgenerator), en steek die knoop in treap op basis van prioriteit. Een treap = heap + tree, een boom waarbij de prioriteit van de knopen aan de heapvoorwaarde voldoen.

Hoofdstuk 2

Toepassingen van dynamisch programmeren

2.1 Langste gemeenschappelijke deelsequentie

Voor twee strings $X = \langle x_0, x_1, \dots, x_{n-1} \rangle$ en $Y = \langle y_0, y_1, \dots, y_{m-1} \rangle$ waarbij x_i en y_i individuele karakters zijn, kan men de langste gemeenschappelijke deelsequentie bepalen door een stringelementen weg te laten, zodat beide strings gelijk zijn. Dit probleem heeft een optimale deelstructuur: de deelproblemen zijn paren prefixen van de twee strings. Stel X_i de prefix met lengte i en X_0 de ledige prefix. Analoog geldt hetzelfde voor Y . Beschouw nu $Z = \langle z_0, z_1, \dots, z_{k-1} \rangle$, dan zijn er drie mogelijkheden:

1. Als $n = 0$ of $m = 0$ dan is $k = 0$.
2. Als $x_{n-1} = y_{m-1}$ dan is $z_{k-1} = x_{n-1} = y_{m-1}$, met gevolg dat Z_{k-1} een LGD is van X_{n-1} en Y_{m-1} .
3. Als $x_{n-1} \neq y_{m-1}$:
 - (a) Z is ofwel een LGD van X_{n-1} en Y , met $z_{k-1} \neq x_{n-1}$, of,
 - (b) Z is een LGD van X en Y_{m-1} , met $z_{k-1} \neq y_{m-1}$.

Dit kan opgesteld worden als een recursieve vergelijking, waarbij $c[i, j]$ de lengte van de LGD voorstelt:

$$c[i, j] = \begin{cases} 0 & \text{als } i = 0 \text{ of } j = 0 \\ c[i-1][j-1] & \text{als } i > 0 \text{ en } j > 0 \text{ en } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{als } i > 0 \text{ en } j > 0 \text{ en } x_i \neq y_j \end{cases}$$

Uitgewerkt op de woorden **LOUIS** en **ALOYSIUS**:

	A	L	O	Y	S	I	U	S
	0	0	0	0	0	0	0	0
L	0	0	1	1	1	1	1	1
O	0	0	1	2	2	2	2	2
U	0	0	1	2	2	2	3	3
I	0	0	1	2	2	3	3	3
S	0	0	1	2	3	3	3	4

Hoofdstuk 3

Uitwendige gegevensstructuren

3.1 B-trees

Een B-tree is een uitwendige evenwichtige meerwegszoekboom met een zeer kleine hoogte waarbij elk blad op dezelfde diepte zit. Meestal wordt er geheugenruimte voorzien voor de wortel en de meest recent gebruikte knopen. Vooraleer knopen kunnen bewerkt worden moeten deze eerst ingeladen worden aan de hand van een welbepaalde paginaindex, en door de grootte van een knoop, bevat het geheugen best zo weinig mogelijk knopen. Overbodige knopen worden terug weggeschreven, indien deze gewijzigd werden, en uit het geheugen verwijderd.

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 inwendige knoop, behalve de wortel, heeft minstens $\lceil m/2 \rceil$ kinderen, tenzij hij een blad is.
- Elk blad zit op hetzelfde niveau

Elke knoop bevat dan:

- Een getal $k = m - 1$ dat aangeeft hoeveel sleutels in de knoop zitten.
- Een tabel met maximaal k sleutels, die niet dalend gerangschikt zijn. Een tweede tabel van dezelfde grootte voor de informatie bij elke sleutel bij te houden.
- Een tabel voor maximaal m wijzers naar de kinderen van de knoop.
- Een logische waarde b die aangeeft of de knoop een blad is.

3.1.2 Eigenschappen

Het minimaal aantal sleutels n kan eenvoudig berekend worden. veronderstel een boom met hoogte h en $g = \lceil m/2 \rceil$. De wortel van de minimale boom heeft slechts één sleutel en twee kinderen. Elk

van die kinderen heeft minimaal g kinderen, die op hun beurt ook minimaal g kinderen hebben, enz. Het **aantal knopen** wordt dus:

$$1 + 2g + \dots + 2g^{h-1} = 1 + 2 \sum_{i=0}^{h-1} g^i$$

Aangezien elke knoop minstens $g - 1$ sleutels heeft, behalve de wortel, kan n geschreven worden als (zie 2.3.3 Afschattingen met sommen Algoritmen I cursus):

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

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

$$h \leq \log_g \frac{n + 1}{2}$$

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

3.1.3 Operaties

Zoeken

Elke knoop op de zoekweg moet ingelezen worden. Allereerst wordt er nagegaan of de sleutel in deze knoop zit (via lineair of binair zoeken). Als de gezochte sleutel niet in de knoop zit, moet de volgende knoop ingeladen worden. De wijzer van de volgende knoop staat op dezelfde index in de kindtabel als waar de niet gevonden sleutel zou moeten zitten in de sleuteltabel. Is de huidige knoop een blad en zit de gezochte sleutel niet in dit blad, dan zit de sleutel niet in de boom.

Toevoegen

Enkel de **bottom-up** methode wordt besproken aangezien de top-down versie minder vaak gebruikt wordt, maar wel handig is indien meerdere gebruikers aan de boom moeten, omdat knopen op de zoekweg dan vroeger worden vrijgegeven. Toevoegen gebeurt altijd in een blad en vormt geen probleem zolang dit blad nog plaats heeft. Is dit blad vol, wordt de knoop gesplitst rond de middelste sleutel van knoop. Sleutels die zich rechts van deze middelste sleutel bevinden, worden in een nieuwe knoop ondergebracht, dat ook een blad wordt. Sleutels die zich links van de middenste sleutel bevinden, blijven in het blad. De middelste sleutel zelf wordt nu verwijderd van de knoop, en toegevoegd bij de ouderknoop van de gesplitste knoop, zodat hetzelfde proces zich kan herhalen. Elke splitsing kost drie schijfoperaties. In het slechtste geval wordt het probleem helemaal tot de wortel opgeschoven, zodat er een nieuwe knoop wordt aangemaakt, met slechts één element, die nu de wortel wordt van de B-tree.

Verwijderen

Ook bij verwijderen wordt enkel de **bottom-up** methode besproken. Een sleutel wordt enkel maar verwijderd indien deze in een blad zit. Deze strategie moet dus de te verwijderen sleutel vervangen met zijn voorloper of opvolger omdat die altijd in een blad zitten, maar de meeste sleutels zitten in bladeren, zodat dit meestal geen probleem vormt. Wanneer een knoop te weinig sleutels heeft $< \lceil m/2 \rceil$ dan kan men proberen sleutels over te nemen van één van de twee broerknopen. De sleutel van de broer gaat naar zijn ouder, een sleutel van de ouder gaat naar de knoop, die ook een kindwijzer van de broer overneemt. Omdat hier drie knopen worden aangepast, is het beter om meerdere sleutels op die manier te roteren, zodat elke knoop evenveel sleutels heeft. Indien geen van

beide broers een sleutel kan afstaan, wordt de knoop samengevoegd met een broer, zodat de ouder een kind verliest. De sleutel die ervoor zorgde dat deze knoop bereikbaar was, wordt toegevoegd aan de samengevoegde knoop en verwijderd uit de ouderknoop.

3.1.4 Varianten

- **B⁺-tree.** Deze variant zal enkel sleutels opslaan in de bladeren zodat inwendige knopen enkel gebruikt worden als index om deze sleutels te lokaliseren. Bovendien is er een gelinkte lijst van alle bladeren in stijgende sleutelvolgorde. Omdat inwendige knopen enkel dienen als index, moeten ze minder informatie bevatten. Bladeren moeten ook geen plaats reserveren voor kindwijzers, zodat ze meer gegevens kunnen bevatten.
- **Prefix B⁺-tree.** Deze variant wordt gebruikt indien de sleutels strings zijn. De inwendige knopen bevatten een zo kort mogelijke string, meestal een prefix van de te onderscheiden strings.
- **B*-tree.** Deze variant zal bij de splitsoperatie de gegevens over drie knopen verdelen, in plaats van twee knopen. Beter gevulde knopen betekent een minder hoge boom.

3.2 Uitwendige Hashing

3.2.1 Extendible hashing

Deze methode bevat een hashtable in het geheugen. Deze tabel bevat wijzers naar de schijfpagina's, die maximaal m sleutels met bijhorende gegevens kunnen bevatten. De hashwaarde zijn gehele getallen, met als bereik de breedte van een processorwoord w . De laatste d bits worden gebruikt als indicies in de hashtable, zodat de tabel 2^d elementen bevat. Deze d is dan ook de globale diepte van de hashtable en komt overeen met de langste prefix. Alle sleutels waarvan de hashwaarde op dezelfde d bits eindigt, komen in dezelfde pagina. Meerdere tabelelementen mogen naar dezelfde pagina verwijzen, daarom wordt er bij elke pagina ook de lokale diepte k bijgehouden: het aantal bits waarmee al haar hashwaarden eindigen. Op die manier bevat elke pagina 2^{d-k} elementen.

- **Zoeken.** Zoeken van de sleutel komt neer op het hashen van deze sleutel, en de overeenkomstige schijfpagina te vinden, en dan deze pagina sequentieel te doorzoeken.
- **Toevoegen.** Toevoegen gebeurt analoog, waarbij gemiddeld de helft van de gegevens in een pagina moeten opschuiven. Indien de pagina vol geraakt moet er gesplitst worden. Deze splitsing gebeurt volgens de waarde van bit $k + 1$. Gegevens waarvoor die bit 1 is worden overgebracht naar een nieuwe gecreëerde pagina, beide met een k dat één groter is als de oorspronkelijke pagina. Nu zijn er nog twee gevallen:
 - * $k - 1 \leq d$: De helft van de wijzers naar de oude pagina moeten vervangen worden door de nieuwe pagina, maakt niet uit de welke.
 - * $k - 1 = d$: Er was slechts één wijzer naar de oude pagina, en omdat k nu groter is dan d , moet d ook met één toenemen, en de grootte van de hashtable moet verdubbeld worden. Elke index wordt nu één bit langer, zodat er twee nieuwe indices ontstaan. De tabelelementen bij beide indicies moeten naar dezelfde pagina verwijzen als de oorspronkelijke index.
- **Verwijderen.** Indien een pagina, na verwijdering van een element, samen met haar broer minder dan m sleutels bevat, moeten deze samengevoegd worden.

3.2.2 Linear hashing

De d eindbits worden niet meer als index in een hashtable, maar rechtstreeks als adres van een pagina gebruikt.

Hoofdstuk 4

Meerdimensionale gegevensstructuren

Notatie: k = aantal dimensies en n = aantal punten.

4.1 Projectie

Deze methode gebruikt per dimensie een gegevensstructuur die alle punten gerangschikt bijhoudt volgens die dimensie. Zoeken gebeurt door een dimensie te kiezen, en alle punten te selecteren die binnen zijn zijde voor die dimensie vallen. Die punten worden dan sequentieel overlopen.

4.2 Rasterstructuur

Deze methode verdeelt de zoekruimte in regelmatige rastergebieden en kan geïmplementeerd worden met een meerdimensionale tabel. Elk rastergebied heeft een gelinkt lijst met punten die in dat gebied liggen, maar kan juist hierdoor onnodig veel geheugen innemen.

4.3 Quadrees

Deze soort bomen verdeelt de zoekruimte in 2^k hyperrechthoeken en was origineel ontworpen voor 2 dimensies. Deze verdeling wordt opgeslagen in een 2^k -wegaanboom. Elke knoop staat voor een gebied, dat onderverdeeld wordt in de 2^k deelgebieden van zijn kinderen. Voor grote k zijn quadrees niet geschikt, daarom wordt enkel twee dimensies besproken.

4.3.1 Point quadrees

In deze versie bevat elke knoop een punt. De coördinaten van dit punt delen het gebied op in 4 rechthoeken. De vorm is afhankelijk van de toevoegvolgorde, zodat slechtste geval $O(1)$ is. Zoeken naar een punt vergelijkt telkens het zoekpunt met de punten bij de opeenvolgende knopen, en daalt eventueel af naar het kind met het gepaste deelgebied.

4.3.2 Point-Region quadtrees

Deze vorm vereist dat de zoekruimte een rechthoek is omdat elke knoop de ruimte in vier gelijke rechthoeken verdeelt, zodat elk deel nul of één punt bevat. Inwendige knopen bevatten geen punten. Hier is de vorm onafhankelijk van de toevoegvolgorde, maar kan wel evenwichtig uitvallen. Het is onmogelijk om de hoogte en grootte in functie van het aantal punten uit te drukken.

4.3.3 k-d trees

Een k-d tree gebruikt een binaire boom, waarbij op elk niveau de dimensies afgewisselt wordt. Elke inwendige knoop bevat dan ook één punt, die de zoekruimte verdeelt in de dimensie voor dat niveau. Ideale opsplitsing bestaat uit gelijkmatige verdeling van de dimensies. In twee dimensies zal elk niveau dus afwisselend de x-dimensie en de y-dimensie beschouwen.

Hoofdstuk 5

Samenvoegbare heaps

= heaps die geoptimaliseerd zijn om de 'join' operatie op uit te voeren. De join operatie voegt twee heaps samen, zodat de heapvoorwaarde nog steeds geldig is.

Een lijst van voorbeelden die niet gekend moeten zijn (In de cursus is er geen uitleg over *hoe* dat je de samenvoegoperatie zou implementeren voor deze heaps):

- Leftist tree.
- Skew heaps.
- Fibonacci heaps.
- Relaxed heaps

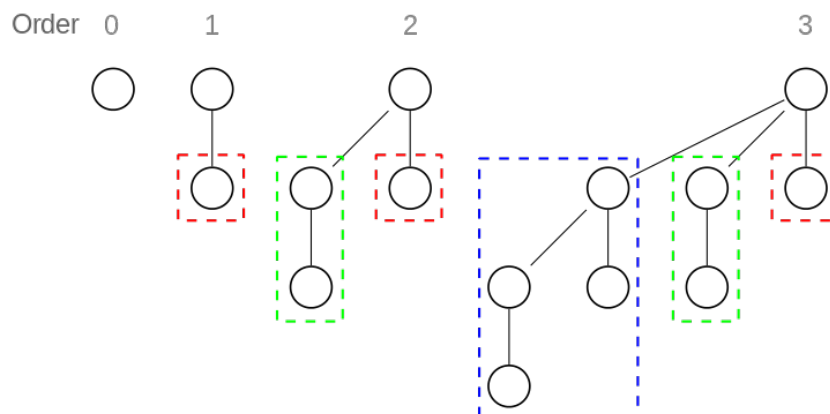
De belangrijke samenvoegbare heaps zijn: Binomial queues en Pairing heaps.

- **Binomial queues:**

- Bestaat uit bos van binomiaalbomen.
- Binomiaalboom B_n bestaat uit twee binomiaalbomen B_{n-1} . B_0 bestaat uit één knoop.
- De tweede binomiaalboom is de meest linkse deelboom van de wortel van de eerste.
- Een binomiaalboom B_n bestaat uit een wortel met als kinderen B_{n-1}, \dots, B_1, B_0 (zie figuur 5.1)
- Op diepte d zijn er $\binom{n}{d}$ knopen.
- Voorbeeld: Een prioriteitswachtrij met 13 elementen wordt voorgesteld als $\langle B_3, B_2, B_0 \rangle$.

De operaties op een binomialqueue:

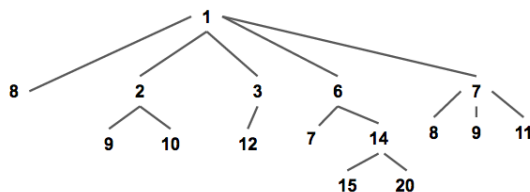
- **Minimum vinden:** Overloop de wortel van elke binomiaalboom. Het minimum kan ook gewoon bijgehouden worden.
- **Samenvoegen:** Tel de bomen met dezelfde hoogte bij elkaar op, $B_h + B_h = B_{h+1}$. Maak de wortel met de grootste sleutel het kind van deze met de kleinste.
- **Toevoegen:** Maak een triviale binomialqueue met één knoop en voeg deze samen met de andere binomialqueue.
- **Minimum verwijderen:** Zoek binomiaalboom B_k met het kleinste wortelelement. Verwijder deze uit de binomialqueue. Verwijder wortel van B_k . Voeg beide binomialqueues terug samen.



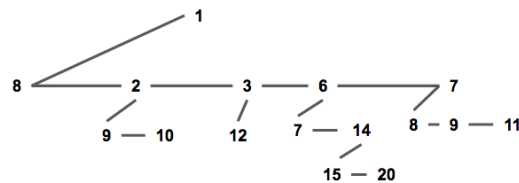
Figuur 5.1: Verschillende ordes van binomiaalbomen.

- **Pairing heaps:**

- Een algemene boom waarvan de sleutels voldoen aan de heapvoorwaarde.
- Elke knoop heeft een wijzer naar zijn linkerkind en rechterbroer (figuur 5.2 en 5.3). Als verminderen van prioriteit moet ondersteund worden, heeft elke knoop als linkerkind een wijzer naar zijn ouder en als rechterkind een wijzer naar zijn linkerkind.



Figuur 5.2: Een pairing heap in boomvorm.



Figuur 5.3: Een pairing heap.

De operaties op een pairing heap.

- **Samenvoegen:** Verlijk het wortelelement van beide heaps. De wortel met het grootste element wordt het linkerkind van deze met het kleinste element.
- **Toevoegen:** Maak een nieuwe pairingheap met één element, en voeg deze samen met de oorspronkelijke heap.
- **Prioriteit wijzigen:** De te wijzigen knoop wordt losgekoppeld, krijgt de prioriteitwijziging, en wordt dan weer samengevoegd met de oorspronkelijke heap.
- **Minimum verwijderen:** De wortel verwijderen levert een collectie van c heaps op. Voeg deze heaps van links naar rechts samen in $O(n)$ of voeg eerst in paren toe, en dan van rechts naar links toevoegen in geamortiseerd $O(\lg n)$.
- **Willekeurige knoop verwijderen:** De te verwijderen knoop wordt losgekoppeld, zodat er twee deelheaps ontstaan. Deze twee deelheaps worden samengevoegd.

Deel II

Grafen II

Hoofdstuk 6

Toepassingen van diepte-eerst zoeken

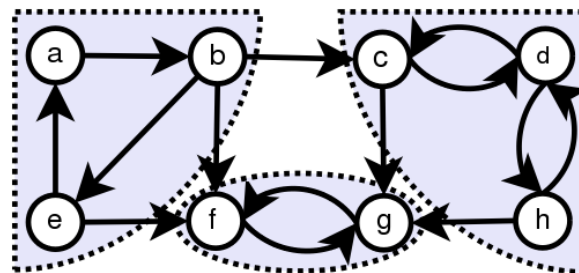
6.1 Enkelvoudige samenhang van grafen

6.1.1 Samenhangende componenten van een ongerichte graaf

- △ Een **samenhangende ongerichte graaf** is een graaf waarbij er een weg bestaat tussen elk paar knopen.
- △ Een **niet samenhangende ongerichte graaf** bestaat dan uit zo groot mogelijke samenhangende componenten.

6.1.2 Sterk samenhangende componenten van een gerichte graaf

- △ Een **sterk samenhangende gerichte graaf** is een graaf waarbij er een weg tussen elk paar knopen in beide richtingen (niet perse dezelfde verbindingen) bestaat (cfr. figuur 6.1).



Figuur 6.1: Een sterk samenhangende graaf.

- △ Een **zwak samenhangende gerichte graaf** is een graaf die niet sterk, maar toch samenhangend is indien de richtingen buiten beschouwing gelaten worden. Een graaf die niet sterk samenhangend is, bestaat uit zo groot mogelijke sterk samenhangende componenten.
- △ Sommige algoritmen gaan ervan uit de een graaf sterk samenhangend is. Men moet dus eerst deze componenten bepalen, meestal via een **componentengraaf** die:
 - △ een knoop heeft voor elk sterk samenhangend component,

- △ en een verbinding van knoop a naar knoop b indien er in de originele graaf een verbinding van één van de knopen van a naar één van de knopen van b is.
- ! De componentgraaf bevat geen lussen, anders wil dit zeggen dat die knoop zelf nog opgesplitst zou kunnen worden in twee sterk samenhangende componenten.
- △ De sterk samenhangende componenten kunnen bekomen worden met behulp van diepte-eerst zoeken (Kosaraju's Algorithm):
 1. Stel de omgekeerde graaf op, door de richting van elke verbinding om te keren.
 2. Pas diepte-eerst zoeken toe op deze omgekeerde graaf, waarbij de knopen in postorder genummerd worden.
 3. Pas diepte-eerst zoeken toe op de originele graaf, met als startknoop de resterende knoop met het hoogste postordernummer. Het resultaat is een diepte-eerst bos, waarvan de bomen sterk samenhangende componenten zijn.
- △ Diepte-eerst zoeken is $\Theta(n + m)$ voor ijle en $\Theta(n^2)$ voor dichte grafen. Het omkeren van de graaf is ook $\Theta(n + m)$ voor ijle en $\Theta(n^2)$ voor dichte grafen.

6.2 Dubbele samenhang van ongerichte grafen

Twee definities:

- △ **Brug.** Een brug is een verbinding dat, indien deze wordt weggenomen, de graaf in twee deelgrafen opsplitst. Een graaf zonder bruggen noemt men dubbel lijnsamenhangend; als er tussen elk paar knopen minstens twee onafhankelijke wegen bestaan, dan is een graaf zeker dubbel lijnsamenhangend.
- △ **Scharnierpunt.** Een scharnierpunt is een knoop dat, indien deze wordt weggenomen, de graaf in ten minste twee deelgrafen opsplitst. Een graaf zonder scharnierpunten noemt men dubbel knoopsamenhangend (of dubbel samenhangend). Een graaf met scharnierpunten kan onderverdeeld worden in dubbel knoopsamenhangende componenten. Als er tussen elk paar knopen twee onafhankelijke wegen bestaan, dan is de graaf dubbel knoopsamenhangend.

Scharnierpunten, dubbel knoopsamenhangende componenten, bruggen en dubbel lijnsamenhangende componenten kunnen opnieuw via diepte-eerst zoeken gevonden worden:

1. Stel de diepte-eerst boom op, waarbij de knopen in postorder genummerd worden.
2. Bepaal voor elke knoop u de laagst genummerde knoop die vanuit u kan bereikt worden via een weg bestaande uit nul of meer dalende boomtakken gevolgd door één terugverbinding.
3. Indien alle kinderen van een knoop op die manier een knoop kunnen bereiken die hoger in de boom ligt dan hemzelf, dan is de knoop zeker niet samenhangend. De wortel is een scharnierpunt indien hij meer dan één kind in heeft. _ToDo: hoe bruggen vinden?

Diepte-eerst zoeken is $\Theta(n + m)$ voor ijle en $\Theta(n^2)$ voor dichte grafen.

6.3 Eulergraaf

Een eulercircuit is een gesloten omloop in een graaf die alle verbindingen éénmaal bevat. Een eulergraaf is een graaf met een eulercircuit, die volgende eigenschappen heeft:

- De graaf is knoopsamenhangend.
- De graad van elke knoop is even.
- De verbindingen kunnen onderverdeeld worden in lussen, waarbij elke verbinding slechts behoort tot één enkel lus.

Hoofdstuk 7

Kortste Afstanden II

7.1 Kortste afstanden vanuit één knoop

7.1.1 Algoritme van Bellman-Ford

Dit algoritme werkt voor verbindingen met negatieve gewichten, iets wat het algoritme van Dijkstra niet kon. Het algoritme is dan natuurlijk ook trager. (Dijkstra gebruikt het feit dat indien een pad naar $A \rightarrow C$ bestaat, er geen korter pad $A \rightarrow B \rightarrow C$ kan zijn, daarom is het algoritme performant, maar er mogen dus geen negatieve verbindingen zijn. Indien B negatief zou zijn dan klopt Dijkstra niet.).

- ✓ Werkt voor negatieve verbindingen.
- ✓ Geen globale kennis nodig van heel het netwerk, zoals bij Dijkstra, maar slechts enkel de burens van een bepaalde knoop. Daarom gebruiken routers Bellman-Ford (distance vector protocol).
- ! Zal niet stoppen indien er een negatieve lus in de graaf zit, aangezien het pad dan zal blijven dalen tot $-\infty$.

Het algoritme berust op een belangrijke eigenschap: Indien een graaf geen negatieve lussen heeft, zullen de kortste wegen evenmin lussen hebben en hoogstens $n - 1$ verbindingen bevatten. Hieruit kan een recursief verband opgesteld worden tussen de kortste wegen met maximaal k verbindingen en de kortste wegen met maximaal $k - 1$ verbindingen.

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

met

- $d_i(k)$ het gewicht van de kortste weg met maximaal k verbindingen vanuit de startknoop naar knoop i ,
- g_{ij} het gewicht van de verbinding (j, i) ,
- $j \in V$ elke buur j van i .

Er bestaan twee goede implementaties:

1. Δ Niet nodig om in elke iteratie alle verbindingen te onderzoeken. Als een iteratie de voorlopige kortste afstand tot een knoop niet aanpast, is het zinloos om bij de volgende iteratie de verbindingen vanuit die knoop te onderzoeken.
 - Δ Plaats enkel de knopen waarvan de afstand door de huidige iteratie gewijzigd werd in een wachtrij.
 - Δ Enkel de burens van deze knopen worden in de volgende iteratie getest.
 - Δ Elke buur moet, indien hij getest wordt en nog niet in de wachtrij zit, ook in de wachtrij gezet worden.
2. Δ Gebruik een deque in plaats van een wachtrij.
 - Δ Als de afstand van een knoop wordt aangepast, en als die knoop reeds vroeger in de deque zat, danst voegt men vooraan toe, anders achteraan.
 - ! Kan in bepaalde gevallen zeer inefficiënt uitvallen.

7.2 Kortste afstanden tussen alle knopenparen

Δ Voor dichte grafen \rightarrow Floyd-Warshall (Algoritmen I).

Δ Voor ijle grafen \rightarrow Johnson.

7.2.1 Het algoritme van Johnson

Δ Maakt gebruik van Bellman-Ford en Dijkstra.

Δ Omdat we Dijkstra gebruiken, moet elk gewicht positief worden.

1. Breidt de graaf uit met een nieuwe knoop s , die verbindingen van gewicht nul krijgt met elke andere knoop.
2. Voer Bellman-Ford uit op de nieuwe graaf om vanuit s de kortste afstand d_i te bepalen tot elke originele knoop i .
3. Het nieuwe gewicht \hat{g}_{ij} van een oorspronkelijke verbinding g_{ij} wordt gegeven door:

$$\hat{g}_{ij} = g_{ij} + d_i - d_j$$

Δ Het algoritme van Dijkstra kan nu worden toegepast op elke originele knoop, die alle kortste wegen zullen vinden. Om de kortste afstanden te bepalen moeten de originele gewichten opgeteld worden op deze wegen.

Δ Dit algoritme is $O(n(n+m) \lg n)$ want:

Δ Graaf uitbreiden is $\Theta(n)$.

Δ Bellman-Ford is $O(nm)$.

Δ De gewichten aanpassen is $\Theta(m)$.

! n maal Dijkstra is $O(n(n+m) \lg n)$. Dit is de belangrijkste term, al de andere termen mogen verwaarloosd worden.

7.3 Transitieve sluiting

Sluiting = algemene methode om één of meerdere verzamelingen op te bouwen. ('als een verzameling deze gegevens bevat, dan moet ze ook de volgende gegevens bevatten').

△ **Fixed point:** Een sluiting wordt fixed point genoemd omdat op een bepaald moment verdere toepassing niets meer verandert, $f(x) = x$.

△ **Least fixed point:** De kleinste x zoeken zodat $f(x) = x$ voldaan wordt.

Transitieve sluiting = 'Als (a, b) en (b, c) aanwezig zijn dan moet ook (a, c) aanwezig zijn.'

△ Transitieve sluiting van een gerichte graaf is opnieuw een gerichte graaf, maar:

△ er wordt een nieuwe verbinding van i naar j toegevoegd indien er een weg bestaat van i naar j in de oorspronkelijke graaf.

• 3 algoritmen:

1. **Diepte-of breedte-eerst zoeken:**

△ Spoor alle knopen op die vanuit een startknoop bereikbaar zijn en herhaal dit met elke knoop.

✓ Voor ijle grafen $\rightarrow \Theta(n(n + m))$.

! Voor dichte grafen $\rightarrow \Theta(n^3)$.

2. **Met de componentengraaf:**

△ Interessant wanneer men verwacht dat de transitieve sluiting een dichte graaf zal zijn, want dan zijn veel knopen onderling bereikbaar, zodat er veel sterk samenhangende componenten zijn. Die kunnen in $\Theta(n + m)$ bepaald worden.

△ Maak dan de componentengraaf (kan in $O(n + m)$).

△ Als nu blijkt dat component j beschikbaar is vanuit component i , dan zijn alle knopen van j bereikbaar vanuit knopen van i .

3. **Het algoritme van Warshall:**

△ Maak een reeks opeenvolgende $n \times n$ matrices $T^{(0)}, T^{(1)}, \dots, T^{(n)}$ die logische waarden bevat.

△ Element $t_{ij}^{(k)}$ duidt aan of er een weg tussen i en j met mogelijke intermediaire knopen $1, 2, \dots, k$ bestaat.

△ Bepalen opeenvolgende matrices:

$$t_{ij}^{(0)} = \begin{cases} \text{onwaar} & \text{als } i \neq j \text{ en } g_{ij} = \infty \\ \text{waar} & \text{als } i = j \text{ of } g_{ij} < \infty \end{cases}$$

en

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \text{ OF } (t_{ik}^{(k-1)} \text{ EN } t_{kj}^{(k-1)}) \quad \text{voor } 1 \leq k \leq n$$

$T^{(n)}$ is de gezochte burenmatrix.

Hoofdstuk 8

Stroomnetwerken

- △ Gerichte graaf met twee speciale knopen: producent en verbruiker.
- △ Producent levert een bepaald materiaal aan verbruiker via de gerichte verbindingen van de graaf.
- △ Elke verbinding heeft een capaciteit $c(i, j)$ dat positief is, en de stroom $s(i, j)$ dat er door loopt ($0 \leq s(i, j) \leq c(i, j)$).
- △ Verbindingen die niet bestaan, worden toch voorgesteld als een verbinding, maar met $c = 0$, zodat er geen stroom kan doorlopen. Dit maakt implementaties eenvoudiger.
- △ Een knoop bevat geen capaciteit, en kunnen hierdoor niets opslaan.

8.1 Maximalestroomprobleem

= zoveel mogelijk materiaal van producent naar verbruiker laten stromen, zonder de capaciteit van de verbinding te overschrijden.

- △ Methode van **Ford-Fulkerson** ('methode' aangezien manier om vergrotende paden te zoeken ontbreekt).
- △ Iteratieve methode. Bij elke iteratie neemt netstroom vanuit producent toe.
- △ Stel de verzameling van knopen K , dan is de totale nettostroom f uit de producent p (De som van het materiaal dat toekomt min het materiaal dat weggaat van elke knoop).

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

- △ De stroomverdeling is het geheel van alle stromen voor alle mogelijke knopenparen.
- △ Het restnetwerk is een nieuw netwerk dat verkregen wordt door uit de huidige stroomverdeling een overzicht op te stellen van de mogelijke stroomtoename tussen elk paar knopen. Het restnetwerk bevat dezelfde knopen, maar niet noodzakelijk dezelfde verbindingen en capaciteit.

Het algoritme verloopt als volgt:

1. Initiële stroomverdeling is nul. Het restnetwerk is initieel de volledige graaf.

2. Bij elke iteratie wordt de huidige stroomverdeling en de verschillende capaciteiten gebruikt om een restnetwerk op te stellen.
3. In dit restnetwerk wordt een vergrotend pad gezocht: een weg van producent naar verbruiker die nog meer stroom tussen beide toelaat.
4. Als er geen vergrotende paden kunnen gevonden worden, stopt de iteratie, en zal de netwerkstroom maximaal zijn.

<https://www.youtube.com/watch?v=TI90tNtKvxs>

Hoe gaan we vergrotende weg zoeken?

- **Performantie afhankelijk van de capaciteiten**

- Stel dat alle capaciteiten geheel zijn en C de grootste capaciteit voorstelt. Restnetwerk bepalen is $O(m)$, en daarin diepte-eerst of breedte-eerst zoeken naar een vergrotend pad is ook $O(m)$. Aantal iteraties in Ford-Fulkerson is $O(nC)$, dus totale performantie is $O(nmC)$, wat niet performant is voor grote C .
- Neem de het vergrotend pad die de grootste stroomtoename mogelijk maakt, dan wordt het aantal iteraties $O(m \lg C)$ (zonder bewijs). Om die weg te vinden volstaat een kleine aanpassing aan het algoritme van Dijkstra: kortste afstand vervangen door grootste capaciteit, zodat de iteratiestap $O(m \lg n)$ (van Dijkstra) is. De totale performantie is nu $O(m^2 \lg C \lg n)$.
- Een vergrotende weg bepalen die een stroomtoename van minsten c , de cut-off, eenheden toelaat, of besluiten dat deze weg niet bestaat, kan in $O(m)$. Indien geen enkele weg meer gevonden wordt, dan is de minimale snedecapaciteit van het restnetwerk lager dan mc . Halveer c en gaan opnieuw op zoek, maar nu naar verbindingen die de helft van de oorspronkelijke c kunnen bevatten. Begin met ondergrens $c = 2^{\lfloor \lg C \rfloor}$, deze in elke fase halveren, en eindigen met één, dan wordt de maximale stroom bereikt met $O(m \lg C)$ iteraties.

- **Performantie onafhankelijk van de capaciteiten**

Deze methode is enkel interessant wanneer C groot is. Als de vergrotende weg steeds het minimum aantal verbindingen heeft, dan kan men aantonen (zonder bewijs) dat de lengte van de vergrotende wegen na hoogsten m iteraties stijgt, en aangezien de maximale lengte $n - 1$ is, volstaan $O(nm)$ iteraties. Deze iteratiestap gebruikt ook breedte-eerst zoeken en is $O(m)$, zodat de totale performantie $O(nm^2)$ wordt.

_ToDo: todo

8.2 Verwante problemen

onbelangrijk

8.3 Meervoudige samenhang in grafen

- △ Knoopsamenhang: indien er tussen elk paar knopen van een graaf k onafhankelijke wegen bestaan, zonder gemeenschappelijke knopen, dan is de graaf k -voudig knoopsamenhangend.
- △ Lijnsamenhang: indien er tussen elk paar knopen van een graaf k onafhankelijke wegen bestaan, zonder gemeenschappelijke verbindingen, dan is de graaf k -voudig lijnsamenhangend.

Hoofdstuk 9

Koppelen

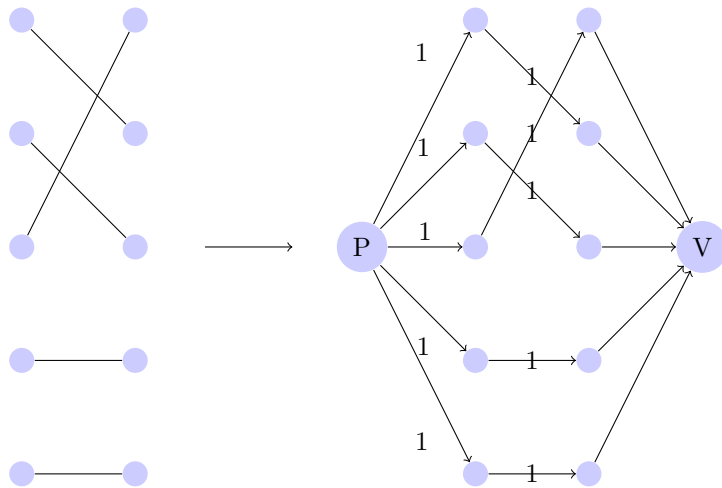
= een deelverzameling van de verbindingen waarin elke knoop hoogstens eenmaal voorkomt. Een maximale koppeling is een koppeling met het grootste aantal verbindingen. Soms hebben verbindingen ook een gewicht.

9.1 Koppelen in tweeledige grafen

Tweeledige graaf (of bipartiete graaf) = 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 knoop uit R verbinden.

9.1.1 Ongewogen koppeling

- Een knoop uit L kan slechts met één knoop uit R verbonden worden. Een maximale ongewogen koppeling zal zoveel mogelijk knopen uit L met R verbinden.
- Is eenvoudiger op te lossen met behulp van een stroomnetwerk. Daarom moet eerst de bipartiete graaf omgezet worden in een stroomnetwerk:
 - Voer een producent P in, die met alle knopen van L verbonden wordt ($P \rightarrow L$).
 - Voer een verbruiker V in, die met alle knopen van R verbonden wordt ($R \rightarrow V$).
 - De oorspronkelijke verbindingen van de graaf krijgen nu een gerichte verbinding van L naar R .
 - De capaciteiten van elke verbinding wordt 1.



Deel III

Strings

Hoofdstuk 10

Gegevensstructuren voor strings

△ Veronderstelling: elke sleutel heeft een ander bitpatroon. Dit impliceert dat geen enkele sleutel een prefix is van een ander.

10.1 Digitale zoekbomen

- △ Analoog met gewone zoekbomen, met één verschil: 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 (van links naar rechts). Bij de wortel gebruiken we het eerste sleutelement, een niveau dieper het tweede sleutelement, enz...
- △ De boom overlopen in **inorder** geeft de sleutels niet noodzakelijk in volgorde. De hoogte van de boom wordt beperkt tot het aantal bits van de grootste sleutel. Voor een groot aantal sleutels met relatief kleine bitlengte is de performantie in het slechtste geval veel beter dan die van een gewone binaire zoekboom, en vergelijkbaar met die van een rood-zwarte boom. In het gemiddelde geval is de kans op een volgend nul- of éénbit steeds gelijk, zodat er gemiddeld evenveel sleutels links als rechts komen. Daardoor is de gemiddelde hoogte van de boom $O(\lg n)$. De woordenboekoperaties zijn ook $O(\lg n)$.
- ! De dikwijls beperkende voorwaarde is een efficiënte toegang tot de bits van de sleutels. Bovendien zijn er enkel woordenboekoperaties mogelijk.

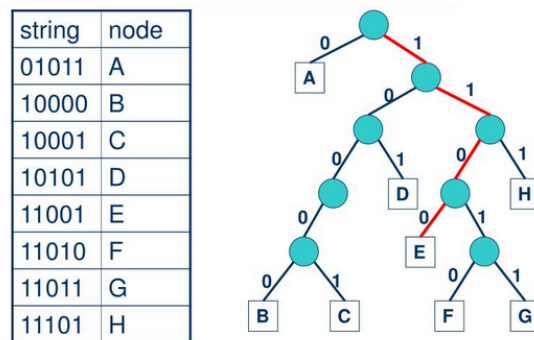
10.2 Tries

Deze familie bomen behouden wel de sleutels in volgorde.

10.2.1 Binaire tries

Cfr. figuur 10.1

- △ Zoekweg wordt bepaald door de opeenvolgende bits van de zoeksleutel.
- △ De sleutels worden enkel opgeslagen in de bladeren, en niet in inwendige knopen.
- ✓ Deze boom in order overlopen geeft wel de sleutels gerangschikt.



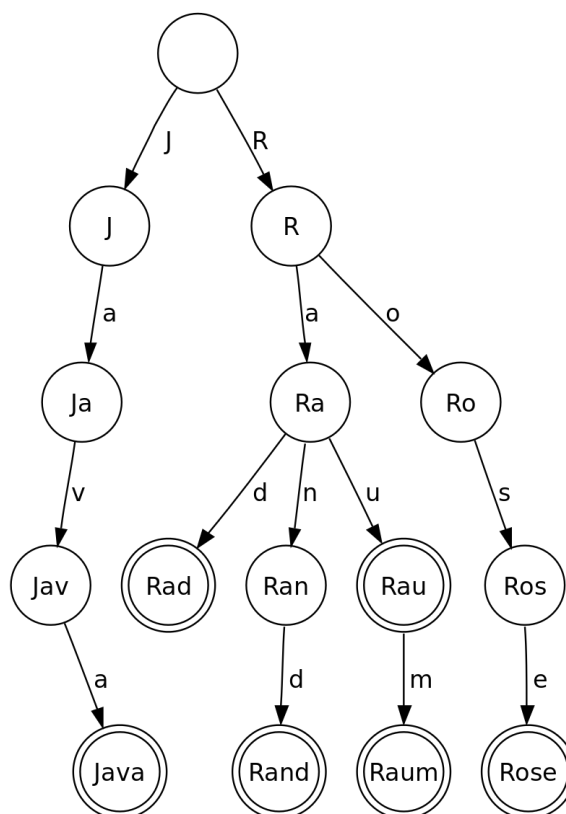
Figuur 10.1: Een binaire trie. De letters worden voorgesteld door een bepaald bitpatroon, die hier niet overeenkomt met het werkelijk bitpatroon voor deze letter, maar een willekeurig gegenereerd bitpatroon.

- ✓ De zoeksleutel moet niet vergeleken worden met elke knoop op de zoekweg, maar met de opeenvolgende bits.
- △ Komen we bij een ledige deelboom terecht (nullwijzer), dan bevat de boom de zoeksleutel niet, en kan dan op deze plaats toegevoegd worden.
- △ Komen we niet bij een ledig deelboom terecht, dan bevat het blad de enige sleutel in die boom die gelijk kan zijn aan de zoeksleutel, aangezien ze dezelfde beginbits hebben.
- ! Wanneer opgeslagen sleutels veel gelijke beginbits hebben zijn er veel knopen met slechts één kind.
- ! De bits van een sleutel mogen geen prefix zijn van een andere sleutel.
- ✓ De structuur is onafhankelijk van de toevoegvolgorde, zodat er slechts één unieke trie is voor elke verzameling sleutels.
- △ Een trie opgebouwd uit n gelijkmatige verdeelde sleutels vraagt voor zoeken of toevoegen $O(\lg n)$ bitoperaties.

10.2.2 Meerwegtries

Cfr. figuur 10.2

- ✓ Beperken hoogte van een trie.
- △ Elke inwendige knoop krijgt potentiaal m kinderen (m = aantal verschillende waarden voor een sleutelement).
- △ Zoeken en toevoegen is analoog aan binaire trie.
- ✓ Kinderen zijn alfabetisch gesorteerd.
- △ Wijzers bijhouden naar de kinderen, die dan gesorteerd in tabel zitten.
- △ Performantie is gelijkaardig aan binaire tries: $O(\log_m n)$.
- ! Wat als we nu veronderstellen dat sleutels toch prefixen mogen zijn van andere sleutels (zie begin dit hoofdstuk).



Figuur 10.2: Een meerwegstrie.

! Hoe moet men aanduiden dat de bits van een kortere sleutel op een langere zoekweg opgebruikt zijn?

→ Door elke sleutel af te sluiten met een speciaal sleutelelement, dat in geen enkele andere voorkomt.

! Meerwegstries gebruiken veel geheugen. Twee verbeteringen:

1. Kindwijzers in gelinkte lijst voorstellen, ten koste van snelheidsverlies.
2. Gebruik enkel trie voor eerste aantal niveaus, en schakel dan over naar korte lijsten.

10.3 Variabelelengtecodering

Twee belangrijke redenen om variabele lengte te voorzien:

1. Verhoogde flexibiliteit
2. Compressie

△ In beide gevallen een alfabet waarbij we niet elke letter door evenveel bits voorstellen.

! Nadeel: een heel blok gegevens moet gedecodeerd worden om te weten waar elk getal begint.

△ Bij decoderen hebben we prefixcode nodig (= codering waarbij codewoord nooit het prefix van een ander codewoord kan zijn).

- ✓ Hierdoor weten we dat we het einde van een codewoord bereikt hebben, zonder het volgende codewoord te moeten analyseren.
- △ Prefixcode natuurlijk getal: Schrijf het in het 128-delig stelsel en sla elk cijfer op in een aparte byte. Tel bij het laatste cijfer 128 op.
- △ Prefixcode voor strings: Elk woord gevolgd door een leesteken.

10.4 De Eliascodes en de Fibonaccicode

= universele codes die onafhankelijk zijn van de gekozen brontekst.

	Elias' gammacode	Elias' deltacode	Fibonacci-code
1	1	1	11
2	0 10	010 0	011
3	0 11	010 1	0011
4	00 100	011 00	1011
5	00 101	011 01	00011
6	00 110	011 10	10011
7	00 111	011 11	01011
8	000 1000	0010 000	000011
9	000 1001	0010 001	100011
...			
23	0000 10111	00101 0111	01000011
45	00000 101101	00110 01101	001010011
...			

Tabel 10.1: Verschillende coderingen voor natuurlijke getallen.

△ Tabel 10.1 toont drie verschillende universele codes:

1. **Elias' gammacode**: Stel de bitrepresentatie van het getal n op (dat k bits bevat), en laat dit voorafgaan door $k - 1$ nulbits. Aan dit aantal nulbits weet het decoderprogramma hoeveel bittekens er daarna nog zullen volgen.
2. **Elias' deltacode**: Gebruik de laatste $k - 1$ bits van het getal, en laat dit voorafgaan door de Elias' gammacode voor k .
3. **Fibonacci-code**: Stel de reeks van Fibonacci:

1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...

De rij heeft als eigenschap dat elk getal n als de som van verschillende Fibonaccigetallen kan geschreven worden, zodanig dat elke term geen onmiddellijke buur is van elkaar.

De Fibonacci-code gebruikt dus het overeenkomstig bitpatroon waarbij de gebruikte termen het bit op 1 staan hebben. Bv voor $n = 7$:

1 2 3 5

0 1 0 1

Aan dit bitpatroon wordt aan het einde nog een 1 toegevoegd, zodat we 01011 krijgen.

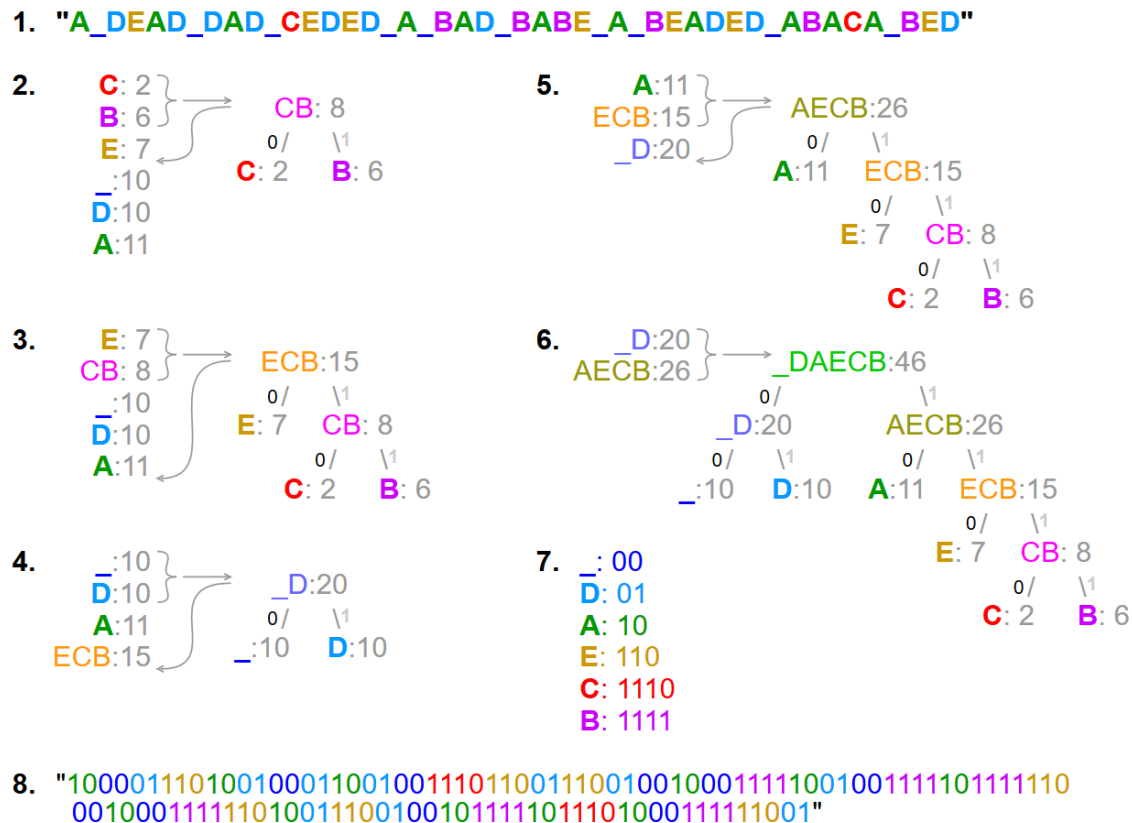
10.5 Huffmancodering

= Minder bittekens gebruiken voor letters die veel voorkomen, zodat de totale lengte ingekort kan worden.

10.5.1 Opstellen decoderingsboom

- △ Prefixcode waarbij elk letter een apart codewoord heeft, dat over het hele bericht dezelfde blijft.
- △ Optimale prefixcode kan enkel gevonden worden indien de frequentie van elk codewoord gekend is.
- △ Bestand met alfabet $\Sigma = \{s_i | i = 0, \dots, d - 1\}$ levert reeks frequenties f_i .
- △ Binaire trie gebruiken met d bladeren, elk met een letter uit Σ .
- △ Ken aan elke knoop een gewicht toe:
 - △ Blad: de frequentie van de overeenkomstige letter.
 - △ Inwendige knoop: De som van de gewichten van zijn kinderen (dit is dan ook het aantal keer dat deze knoop bezocht wordt).
- △ Aantal bits in het gecodeerde bestand is de som van de gewichten van alle knopen samen, met uitzondering van de wortel. De trie met een minimaal gewicht geeft de optimale code en bevat n bladeren.
- △ Hoe optimale trie vinden? Stel knoop k met gewicht w_k en knoop l met gewicht w_l , zodanig dat k niet onder l hangt of l niet onder k hangt. Is het nuttig om deze twee knopen van plaats te verwisselen?
 - Elke knoop boven k (aantal knopen boven $k = d_k$) verliest gewicht w_k en krijgen gewicht w_l .
 - Elke knoop boven l (aantal knopen boven $l = d_l$) verliest gewicht w_l en krijgen gewicht w_k .
 - De gewichtsverandering is

$$(d_k - d_l)(w_l - w_k)$$
 - Als l een groter gewicht heeft en dieper zit dan k , dan is er een betere trie door k en l om te wisselen.
- △ Opbouw boom (cfr. Figuur 10.3). Op elk ogenblik hebben we een bos van deelbomen die we aan elkaar moeten hangen:
 1. Het bos bestaat eerst uit alle bladeren. Dit wordt bekomen door eerst het hele bericht in te lezen, en de frequentie van elke letter te tellen.
 2. Neem de twee bladeren met de minste frequentie (bv met priorityqueue), en maak ze de kinderen van een nieuwe ouderknoop, die als gewicht de som van deze frequenties bevat. Het rechterkind van deze ouderknoop moet deze zijn met het grootste gewicht. Bij gelijk gewicht maakt het niet uit. De resulterende boom steken we terug in het bos.
 3. Blijf dit proces herhalen door telkens 2 bladeren, 2 deeltries, of 1 blad en 1 deeltrie (telkens de twee met het minste gewicht) samen te voegen tot dat er één enkele trie in het bos zit. Dit is de gezochte trie.
 4. De codering voor elke letter kan dan eenvoudig bepaald worden door alle bladeren te zoeken.



Figuur 10.3: Illustratie van de Huffman-codering.

10.5.2 Patricia's

- ! Veel triekknopen hebben maar één kind. Bovendien zijn er twee soorten knopen: bladeren en inwendige knopen.
- ✓ Patricia's vermijden deze problemen door enkel knopen met meer dan één kind te behouden.
- △ Stel een normale m -wegtrie. Indien een knoop maar één kind heeft, kan in plaats van naar dit kind te wijzen, naar het kind van dit kind wijzen. Op die manier wordt er minder geheugen gespendeerd aan het aantal knopen. Er zijn hier wel twee gevolgen:
 1. Elke knoop moet nu bijhouden hoeveel voorouders er ontbreken. Als eenvoudig wordt de index van het te testen karakter, het testkarakter bijgehouden.
 2. Omdat nu een karakter niet getest wordt, kan het zijn dat dit karakter niet overeenkomt met het karakter in het blad.
- △ Een expliciete knoop is een knoop die nog in de resulterende boom zit. Een impliciete knoop is een knoop die wordt aangegeven door een indexprong.
- △ Zoeken:
 - Daal in boom en gebruik het testkarakter van een knoop om de zoekweg te bepalen.
 - Bij nulpointer \rightarrow dan zit string niet in de trie.
 - Bij blad \rightarrow met onzekerheid dat dit het juiste blad is. De strings moeten nog eens vergeleken worden (elk blad moet dus de volledige string bevatten).

△ Toevoegen:

- Bij blad \rightarrow string zit al in de trie.
- Bij nulpointer \rightarrow voeg blad toe. Maar het kan zijn dat deze moet toegevoegd worden aan een impliciete knoop. Daarom wordt het verschilindex berekend, die de eerste plaats aanduidt waar de nieuwe string verschilt van de meest gelijkende string in de trie. Bij een lege trie is deze index nul.
- Drie mogelijkheden:
 1. Zoeken eindigt in een expliciete knoop waarbij $\text{testindex} = \text{verschilindex}$, maar die geen kind heeft voor het karakter in de string aangeduid door de verschilindex. Voeg een blad toe voor de string.
 2. Zoeken eindigt in een expliciete knoop waarbij $\text{testindex} > \text{verschilindex}$, maar die geen kind heeft voor het karakter in de string aangeduid door de verschilindex. Voeg nieuwe expliciete knoop toe met als testindex de verschilindex. De knoop krijgt twee kinderen: de oude expliciete knoop en het nieuwe blad.
 3. Zoeken eindigt in een blad.

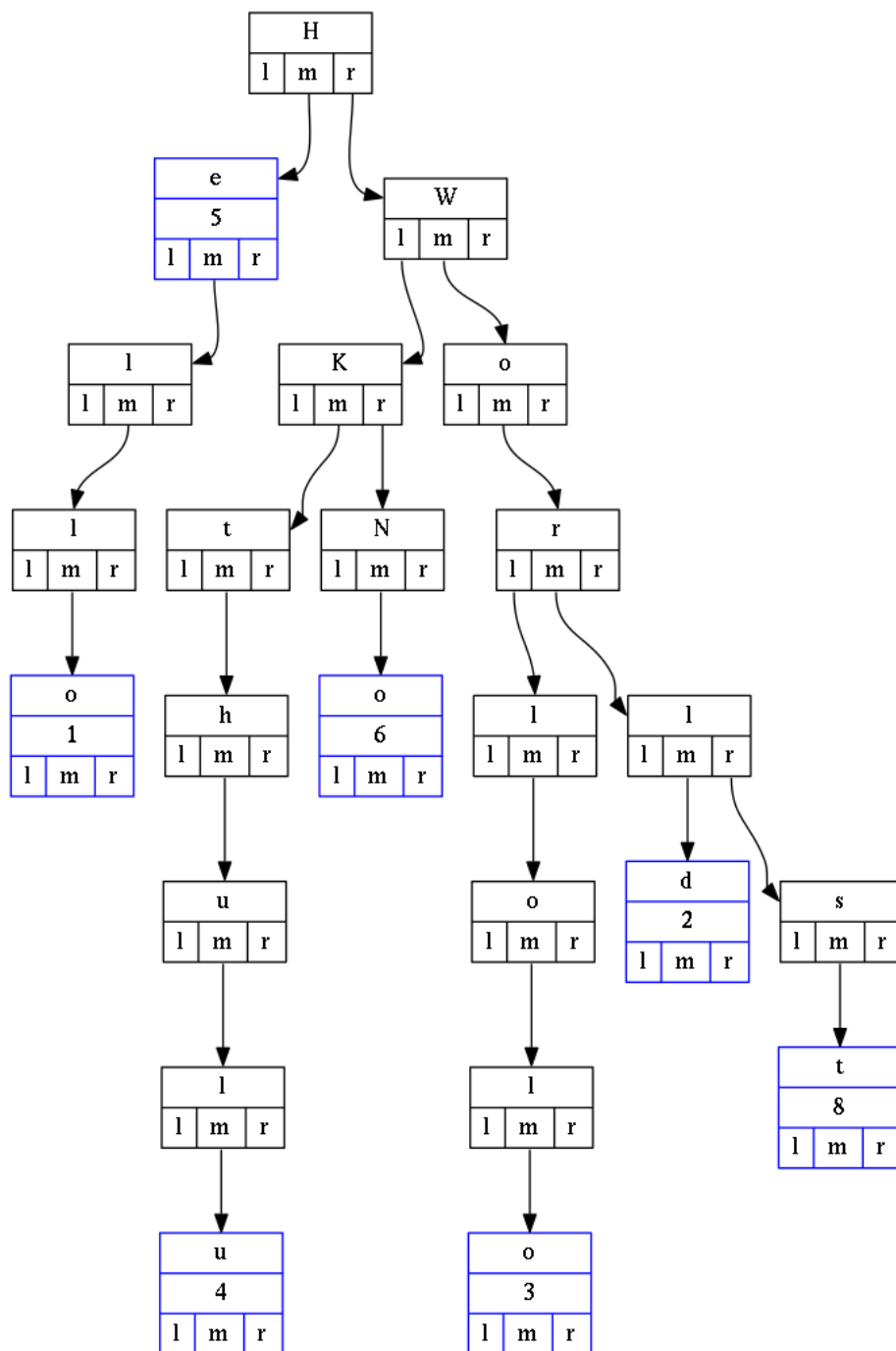
Binaire patriciatricie

- △ Elke expliciete inwendige knoop heeft twee kinderen.

10.6 Ternaire zoekbomen

! m -wegstrie gebruikt tabel met m kindwijzers in elke knoop \rightarrow te veel geheugen.

- △ Ternaire zoekboom bevat sleutelelement, en drie pointers naar knopen.
- △ Is het zoeksleutelelement kleiner/groter, dan zoeken we verder in de linkse/rechtse deelboom met hetzelfde zoeksleutelelement.
- △ Is het element gelijk, dan zoeken we verder in de middelste deelboom, met het volgende element.
- △ Een zoeksleutel wordt gevonden wanneer we met zijn afsluitelement bij een knoop met datzelfde element terechtkomen (de cijfers van de blauwe knopen in figuur 10.4).
- △ 3 voordelen:
- ✓ Past zich goed aan bij onregelmatige zoeksleutels.
 - ✓ Zoeken naar afwezige sleutels is efficiënt. Er worden slechts enkele sleutelementen vergeleken, en volgen slechts enkele wijzers.
 - ✓ Het laat meer complexe zoekoperaties toe: zoeken naar sleutels waarvan bepaalde elementen niet gespecificeerd zijn of alle sleutels opsporen die in niet meer dan één element verschillen van de zoeksleutel.
- △ 2 mogelijke verbeteringen:
1. Beperken aantal knopen door sleutels in bladeren op te slaan (zoals bij tries), en door inwendige knopen met slechts één kind te elimineren (zoals patriciatricies). De zoektijd wordt dan onafhankelijk van de sleutellengte.
 2. Vervang de wortel door een meerwegstrieknoop, zodat we een tabel van ternaire zoekbomen hebben.



Figuur 10.4: Een ternaire zoekboom. Stel dat we het woord "Worst" zoeken. $W > H$, dus we dalen af naar het rechterkind. $W = W$, dus we dalen af naar het middelste kind en vergelijken het volgende zokelement o . Dezelfde procedure gebeurt voor o en r . Dan is $s > m$, dus we dalen af naar het rechterkind. $s = s$ en $t = t$ zodat het woord gevonden wordt.

Hoofdstuk 11

Zoeken in strings

Symbolen die gebruikt worden:

Symbool	Betekenis
Σ	Het gebruikte alfabet
Σ^*	De verzameling strings van eindige lengte van letters uit Σ
d	Aantal karakters in Σ
P	Het patroon (naald)
p	Lengte van P
T	De tekst (hooiberg)
t	Lengte van T

11.1 Formele talen

Formele taal over een alfabet = een verzameling eindige strings over dat alfabet.

In de cursus wordt formele talentheorie beperkt tot wat in de praktijk handig is: generatieve grammatica's en reguliere uitdrukkingen.

11.1.1 Generatieve grammatica's

= Een startsymbool wordt getransformeerd tot een zin van de taal met behulp van substitutieregels. Hiervoor zijn **niet-terminale symbolen**, of kortweg niet-terminalen nodig met als notatie $\langle \dots \rangle$.

Ξ = De verzameling van alle strings van letters uit Σ vermengd met niet-terminalen.

Ξ^* = de bijhorende verzameling strings.

Voorbeeld: stel de volgende niet-terminalen (het symbool $::=$ staat voor "wordt herleidt tot"): Deze

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

grammatica definieert als formele taal de verzameling van alle strings ofwel bestaande uit een rij 'a's gevolgd door een even lange rij 'b's ofwel bestaande uit een rij 'c's gevolgd door een even lange rij

'd's. De afleiding van bijvoorbeeld de string "ccdd" wordt gegeven door

$$\langle S \rangle \Rightarrow \langle CD \rangle \Rightarrow c\langle CD \rangle d \Rightarrow cc\langle CD \rangle dd \Rightarrow ccc\langle CD \rangle ddd \Rightarrow cccdd$$

11.1.2 Reguliere uitdrukkingen

(reguliere uitdrukking = regexp) = een string over het alfabet $\Sigma = \{\sigma_0, \sigma_1, \dots, \sigma_{d-1}\}$ aangevuld met de symbolen \emptyset , ϵ , $*$, $(,)$ en $|$ gedefinieerd door tabel 11.1.

$\langle \text{Regexp} \rangle$::=	$\langle \text{basis} \rangle \langle \text{samengesteld} \rangle$
$\langle \text{basis} \rangle$::=	$\sigma_0 \dots \sigma_{d-1} \emptyset \epsilon$
$\langle \text{samengesteld} \rangle$::=	$\langle \text{plus} \rangle \langle \text{of} \rangle \langle \text{ster} \rangle$
$\langle \text{plus} \rangle$::=	$(\langle \text{Regexp} \rangle \langle \text{Regexp} \rangle)$
$\langle \text{of} \rangle$::=	$(\langle \text{Regexp} \rangle \langle \text{Regexp} \rangle)$
$\langle \text{ster} \rangle$::=	$(\langle \text{Regexp} \rangle)^*$

Tabel 11.1: Definitie reguliere expressie.

- △ Een taal die door een regexp gedefinieerd kan worden heet een reguliere taal.
- △ De formele taal die een regexp R definieert, noteren we $\text{Taal}(R)$.
- △
 1. \emptyset is een regexp, met als taal de lege verzameling.
 2. De lege string, voorgesteld als ϵ , is een regexp met als taal $\text{Taal}(\epsilon) = \{\epsilon\}$.
 3. Voor elke $a \in \Sigma$ is "a" een regexp, met als taal $\text{Taal}(a) = \{a\}$.

Regexps kunnen gecombineerd worden:

Operatie	Regexp	Operatie op taal/talen
Concatenatie	(RS)	$\text{Taal}(R) \cdot \text{Taal}(S)$
of	$(R S)$	$\text{Taal}(R) \cup \text{Taal}(S)$
Kleensluiting	$(R)^*$	$\text{Taal}(R)^*$

~~ToDo: p112 + 113 + 114~~

11.2 Variabele tekst

Zoeken van een patroon P met lengte p in een tekst T met lengte t kan op veel manieren. Wij zien volgende methoden:

- Knuth-Morris-Pratt
- Boyer-Moore
- Karp-Rabin
- Zoeken met automaten
- De Shift-AND-methode

Zie ook het bestand `Extra/zoeken_in_strings.pdf` voor meer methoden en uitleg.

11.2.1 Eenvoudige methode

= Voor elke positie $0 \leq j \leq t - p$ nagaan of P in T voorkomt startend vanaf positie j , en de opeenvolgende p posities. **bro!**

11.2.2 Knuth-Morris-Pratt

Eerst de definitie van een **deelstring** bepalen.

0	1	2	3	4	5	6	7
A	B	C	D	E	F	G	H

Een deelstring $(T[i], T[j])$ begint op positie i en eindigt op positie j . Let op dat net zoals bij iterator, de positie j wijst naar het element dat niet meer behoort tot de deelstring. Op het voorbeeld is de deelstring $(T[2], T[7])$ gelijk aan CDEFG.

De prefixfunctie

De prefixfunctie $q()$ van een string P bepaalt voor elke stringpositie i , $1 \leq i \leq p$, de lengte van het langste prefix van P met lengte kleiner dan i dat we voor i kunnen leggen. [Online ga je hier vaak een andere definitie voor terugvinden, waarbij \$0 \leq i \leq p\$ en dus \$q\(0\) = 0\$.](#)

Hieruit volgt:

- $q(0) = -1$ (ongedefinieerd)
- $q(1) = 0$
- $q(i) < i$

Hoe $q(i + 1)$ berekenen?

- $q(i + 1) = q(i) + 1$ als $P[q(i)] == P[i]$
- Anders, $q(i + 1) = q(i) + 1$ als $P[q(q(i))] == P[i]$
- Anders, $q(i + 1) = q(i) + 1$ als $P[q(q(q(i)))] == P[i]$
- ...
- Als er geen enkele prefix bestaat dat we voor i kunnen leggen kan verlengd worden. Dan is $q(i + 1) = 0$.

Pseudocode prefixfunctie:

```

q[0] = -1
q[1] = 0
for (i = 2; i <= p; i++){
    vorig = q[i - 1]
    while (vorig > 0 && P[i - 1] != P[vorig]){
        prev = q[vorig]
    }
    if (P[i - 1] == P[vorig]){
        vorig++
    }
    q[i] = prev
}

```


- △ De hashfunctie voor de eerste positie mag gerust langer duren.
- △ Beschouw een string als een verzameling elementen die elk d waarden kan aannemen. Elk element kan dan voorgesteld worden als een getal gelegen tussen 0 en $d - 1$ (inbegrepen.)
- △ Het getal dat P dan voorstelt:

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

- △ Om de hashwaarde ('fingerprint') van P te bepalen, wordt er een getal r ($0 < r \leq 2^w$) (wordt later uitgelegd, w is breedte van processorwoord) gebruikt:

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

- △ $H(P)$ kan heel groot worden, maar je kan gebruik maken van deze eigenschap, die ook geldt voor verschil en product:

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

- △ Om nu in $O(1)$ de opeenvolgende fingerprints voor T_{j+1} te bepalen, kunnen we gebruik maken van de fingerprint T_j :

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

- △ Eenmalig T_0 bepalen in $\Theta(p)$ (exact hetzelfde als bepalen $H_r(P)$):

$$H_r(T_0) = \left(\sum_{i=0}^{p-1} P[i] \cdot d^{p-i-1} \right) \mod r$$

- △ Als nu blijkt dat de hashwaarde van $T_j = P$, dan moet enkel nog de posities $T[j], T[j + 1], \dots, T[j + p]$ vergeleken worden, om al dan niet te besluiten dat de strings gelijk zijn.

- △ Hoe r kiezen?

- **Vaste r :** kies een zo groot mogelijk priemgetal, zodat $rd < 2^w$
De berekening van $H_r(T_{j+1})$ wordt licht aangepast:

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

- **Random r :** Kies een willekeurig priemgetal r uit een bepaald bereik. **ToDo: snap ik nie zo goe**

11.2.5 Zoeken met automaten

Een aantal begrippen:

- △ Geheugen: Elk mechanisme dat de toestand van een eenheid verandert.
- △ Staat: Bij een deterministisch automaat is dit een synoniem voor de toestand van een eenheid. Een geheugenmodule van 32 KB bevat 256^{32000} staten.
- △ Invoerkanaal: Op elk discreet tijdstip is een zoeker invoer beschikbaar. De eindige verzameling van alle mogelijke invoerwaarden noemt het invoeralfabet. Bij mensen is het invoeralfabet de combinatie van alle zintuigen, inclusief zenuwsignalen.
- △ Hooiberg: de reeks symbolen die de automaat als invoer krijgt.

Deterministische automaten (DA)

= Een abstract model voor een schakeling die of een programma dat zich in één van een aantal toestanden kan bevinden, in elke toestand een symbool invoert, en eventueel de toestand kan veranderen.

Formeel bestaat een DA uit:

- △ een eindige verzameling invoersymbolen Σ ,
- △ een eindige verzameling toestanden S ,
- △ een begintoestand s_0 behorend tot S ,
- △ een verzameling eindtoestanden F , die een deelverzameling is van S ,
- △ een overgangsfunctie $p(t, a)$ die de nieuwe toestand geeft wanneer de automaat in toestand t symbool a ontvangt. Dit kan dus voorgesteld worden als een geëtiketteerde multigraaf G overgangsgraaf.

Wanneer een DA zich na het invoeren van een string (opvolging van s_i) in een eindtoestand bevindt, zegt men dat de string herkend werd door de DA.

- △ Een taal die herkend wordt door een DA is regulier. De taal is namelijk een verzameling etiketten van $P_G(\{s_0\}, F)$

Niet-deterministische automaten (NA)

= deterministisch mechanisme waarin toeval geen enkele rol speelt. Het heeft geen staat, maar wel statenbits.

- △ De statenbits duiden aan in welke staat een NA zich bevindt. De beginbit, een speciale statenbit, duidt s_0 aan. De eindbits, duiden dan alle bits aan die ervoor zorgen dat de NA zich in een eindtoestand bevindt (1 of meerdere eindbits moeten aan staan).
- △ De overgang naar een volgende staat werkt bit per bit. Een statenbit die één of meerdere signalen krijgt, zet zichzelf aan, terwijl een statenbit die geen signalen krijgt, zichzelf uitdooft. De functie $s(i, a)$ geeft letter a aan statenbit i .
- △ Een ϵ -overgang van statenbit i naar statenbit j , zal ervoor zorgen dat indien i een signaal krijgt met $a = \epsilon$, deze ook naar j wordt doorgestuurd zonder vertraging, zodat ze beide aanstaan.
- △ Een NA herkent een string indien op het einde van de string de NA één of meerdere eindbits heeft aanstaan.

De deelverzamelingconstructie

= Het omzetten van een NA naar het DA-formalisme met expliciete staten.

11.2.6 De Shift-AND-methode

= bitgeoriënteerde methode, die zeer efficiënt werkt voor kleine patronen, en eenvoudig kan uitgebreid worden om patronen te zoeken waarin fouten zitten.

De methode werkt als volgt:

1. Stel een tabel S op, die d (de grootte van het alfabet) bitpatronen bevat. Bijvoorbeeld: Indien we willen zoeken in de tekst die bestaat uit karakters uit de Extended ASCII standaard. Deze standaard bevat 256 karakters. De tabel S zal 256 bitpatronen bevatten. Enkel de karakters die in het patroon voorkomen zullen een bitpatroon verschillend van 0 hebben. Een bit i van het bitpatroon voor karakter k wordt aangezet indien k voorkomt op positie i in P .

Pseudocode:

```

i ← 0
zolang i < p
    Bitpatroon ← S[P[i]]
    Zet bit i van bitpatroon op 1
    i++

```

Een specifiek bit op 1 zetten kan via een OF operatie met het huidige bitpatroon en een bitpatroon die op positie i enkel deze bit op 1 staan heeft.

Stel dat $P = \text{mijnnaald}$, dan ziet de overeenkomstige tabel S er uit zoals tabel 11.2. De overige

$S['a']$	00000110000000000000000000000000
$S['d']$	00000000100000000000000000000000
$S['i']$	01000000000000000000000000000000
$S['j']$	00100000000000000000000000000000
$S['l']$	00000001000000000000000000000000
$S['m']$	10000000000000000000000000000000
$S['n']$	00011000000000000000000000000000

Tabel 11.2: Tabel S voor $P = \text{mijnnaald}$.

plaatsen hebben uiteraard allemaal een bitpatroon van 0.

2. Hoe kan hiermee gezocht worden in T ?

Stel een bitpatroon R op, waarvan alle bits onwaar zijn. Stel ook een bitpatroon M op, waarvan slechts één bit waar is, namelijk het bit op $p-1$. Elk karakter $T[j]$ moet nu overlopen worden. Op basis van j bouwen we een nieuwe R , R_{j+1} op als volgt:

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

De vorige R waarde wordt eerst 1 positie naar rechts geschoven. Aangezien dit een nulbit vooraan zal toevoegen, moet hier ook nog een OF operatie gedaan worden zodat het bit dat vooraan staat 1 wordt. Dit wordt gevolgd door een EN operatie met het bitpatroon voor karakter $T[j+1]$. Er zijn dan drie gevallen:

- Het bitpatroon van $T[j+1]$ is 0. De nieuwe R krijgt terug een nulbitpatroon, wat neerkomt dat er terug in het begin van het patroon zal gezocht worden.
- Het bitpatroon van $T[j+1]$ is verschillend van 0, maar de positie i van het meest rechtste éénbit van R heeft een nulbit in T op positie i . Dit wil zeggen dat R opnieuw bitpatroon 0 krijgt, en er opnieuw in het begin van het patroon moet gezocht worden.
- Het bitpatroon van $T[j+1]$ is verschillend van 0, en de positie i van het meest rechtste éénbit van R heeft ook een éénbit in T op positie i . Dit wil zeggen dat het patroon matcht met de huidige waarde $T[j+1]$, en er dus verder in het patroon (door de bitshift naar rechts uit te voeren) wordt gezocht.

Wanneer $R = M$, dan wil zeggen dat het patroon gevonden is in T . R wordt dan opnieuw geïnitialiseerd tot een nulbitpatroon, en kan er verder gezocht worden in T .

Pseudocode:

```
R ← bitpatroon 0
M ← bitpatroon waarbij bit  $p - 1$  waar is
j ← 0
zolang j < t
    R ← (Schuif(R) OF Bitpatroon 1) EN S[T[j]]
    if (R EN M){
        patroon gevonden
    }
    j++
```

Hoofdstuk 12

Indexeren van vaste tekst

- △ Voorbereidend werk op de tekst, zodat zoeken in tekst tot $O(p)$ kan gereduceerd worden.
- △ Opslaan van de suffixen van de tekst in een bepaalde gegevensstructuur.
- △ Als het patroon in de tekst voorkomt, moet het een prefix zijn van één van die suffixen.
- △ Om te vermijden dat een suffix een prefix van een ander suffix zou zijn, sluiten we de tekst af met een speciaal karakter.
- △ $suff_i$ = suffix dat begint op lokatie i .

12.1 Suffixbomen

- ! Gebruik geen meerwegstrie. De zoektijd zou wel beperkt worden door sleutellengte p , maar toevoegen is $O(t^2)$ en het neemt onnodig veel geheugen in beslag.
- △ De suffixboom bevat alle suffixen van dezelfde tekst.
- △ Het is verwant met de patriciatie, maar met volgende wijzigingen:
 - △ In plaats van $suff_i$ op te slagen, volstaat de beginindex i . De tekst moet wel bijgehouden worden.
 - △ **_ToDo: dumno**
 - △ Er wordt een staartpointer opgenomen in elke inwendige knoop, die constructie en sommige toepassingen veel sneller maakt.
- △ $staart(abcd) = bcd$. De staart van een string is de string bekomen door het eerste karakter te verwijderen.
- △ We starten met een lege suffixboom en voegen elke keer één karakter van T toe, zodat we na k iteraties suffixen hebben van de string $T[0]...T[k-1]$, zonder afsluitteken.
Na k iteraties is er een kleinste index i , die gelijk is aan k , zodanig dat het incomplete $suff_i$ een prefix is van een vorig suffix. Dan geldt:
 1. Alle suffixen $suff_j$ waarvoor $j < i$ zijn geen prefix en worden aangeduid als een blad.
 2. Alle suffixen $suff_j$ met $j \geq i$ zijn een prefix van een andere string en zijn niet zichtbaar in de suffixboom. $suff_i$ noemen we het actieve suffix.

Wat als we $T[k]$ toevoegen? Alle strings in de suffixboom moet met dit karakter verlengd worden en het karakter $T[k]$ moet toegevoegd worden in de boom, tenzij dit karakter al eerder in de string zat, want dan is deze string een prefix dat we niet te zien krijgen.

1. Voor suffixen die aangeduid worden door een blad moeten we niets doen.
 2. Overloop de strings af die een prefix waren, in orde van dalende lengte. Er zijn er nul of meer die door dit extra karakter geen prefix meer zijn. Die moeten een blad krijgen, waarna we naar het volgende suffix gaan.
- △ De toevoegoperatie van hierboven voegt steeds een blad toe, en springt verder naar de volgende impliciet aanwezige suffix, zodat de toevoegoperatie $O(t)$ is.

Om dit te bereiken houden we een pointer bij naar de laatst toegevoegde inwendige knoop en een pointer naar het actieve punt.

Het actieve punt is de laatste expliciete inwendige knoop die we tegenkomen als we het actieve suffix zouden zoeken in de suffixboom.

△ Toepassingen van een suffixboom:

- △ Het deelstringprobleem (Geef alle beginposities van P in T).
- Construeer de suffixboom voor T , en zoek dan de knoop die overeenkomt met P .
1. Bestaat de knoop niet, dan komt P niet in T voor.
 2. Bestaat de knoop wel, dan zijn de gezochte beginposities de indices bij alle bladeren die opvolgers zijn van deze knoop.

Als P k maal voorkomt, kan die gevonden worden in $O(p + k)$ want elke deelboom heeft hoogstens $k - 1$ inwendige knopen.

- △ Langste gemeenschappelijke deelstring (Gegeven een verzameling van k verschillende strings $S = \{s_1, s_2, \dots, s_k\}$, met totale lengte t . Gezocht de langste gemeenschappelijke deelstring van al die strings.)

Construeer een veralgemeende suffixboom, die al de suffixen bevat van al de strings uit deze verzameling. De bladeren bevatten niet enkel de beginpositie van de suffix, maar ook de string waartoe die behoort. De constructietijd voor een verzameling strings met totale lengte t is $O(t)$.

De boom wordt systematisch overlopen om de lengte van alle prefixen en het aantal verschillende strings te bepalen waarin ze voorkomen, en dus meteen ook het langste prefix dat in alle strings voorkomt.

Voorzie voor elke string in S een verschillend afsluitkarakter, op die manier kunnen we zien tot welke string deze behoort, en behoort elk blad maar tot één string.

12.2 Suffixtabellen

- ✓ Ondersteunt dezelfde operaties, met dezelfde performantie, maar met minder geheugen dan een suffixboom.
- △ Gesorteerde tabel van de suffixen van tekst T . Elke index in de tabel komt overeen met de beginpositie van de gesorteerde suffixen. Voor de tekst **banaan** stelt tabel 12.1 in de eerste twee kolommen de beginpositie i voor elke suffix, gesorteerd op beginpositie. De twee volgende kolommen bevat dezelfde informatie, maar gesorteerd op beginpositie j . De suffixtabel bevat voor elke beginpositie i , de overeenkomstige beginpositie j , zodat de suffixtabel $\{6, 4, 3, 1, 0, 5, 2\}$ wordt.

12.3 Tekstzoekmachines

suffix	i	suffix	i
banaan\$	0	\$	6
anaan\$	1	an\$	4
naan\$	2	aan\$	3
aan\$	3	anaan\$	1
an\$	4	banaan\$	0
n\$	5	n\$	5
\$	6	naan\$	2

Tabel 12.1: Constructie van de suffixtabel voor **banaan**.

Deel IV

Hardnekkige problemen

Hoofdstuk 13

NP

13.1 Complexiteit: P en NP

Elk probleem wordt voorgesteld als een beslissingsprobleem, in de vorm van 'Bestaat er een ...?'.

De klasse P(olynomiaal): bevat alle problemen waarvan de uitvoeringstijd begrensd wordt door een veelterm in de grootte van het probleem, bij uitvoering op een realistisch computermodel.

- Grootte = het aantal bits bedoeld dat nodig is om de invoergegevens in een computer voor te stellen. Deze voorstelling is compact, bevat geen overbodige informatie, en stelt getallen voor in een talstelsel met radix groter dan 1.
- Realistisch computermodel = heeft een polynomiale bovengrens voor het werk dat in één tijdseenheid kan verricht worden.

Elk probleem in P wordt als efficiënt oplosbaar beschouwd.

Waarom veeltermen gebruiken, want $O(n^{100})$ is toch niet efficiënt?

1. Als uitvoeringstijd niet begrensd is door een veelterm, is het niet efficiënt oplosbaar. Normaal is de graad ook beperkt (zelden meer dan twee of drie).
2. Veeltermen zijn de kleinste klasse functies die kunnen gecombineerd worden waarbij het resultaat opnieuw een veelterm is.

De klasse N(iet deterministisch) P(olynomiaal): Als het opsplitsen van een verzameling kandidaten en het controleren van één kandidaat beide in polynomiale tijd kunnen gebeuren, dan kan een niet-deterministische computer het antwoord geven in polynomiale tijd.

- Niet-deterministische computer = Een computer met oneindig veel processoren, waarbij elke processor in een tijdsinterval k aantal andere processoren kan aanspreken. In t tijdsintervallen kunnen we t^k processoren aan het werk stellen. Deze processoren werken niet samen, ze kunnen enkel hun deel van het probleem oplossen.
- Kandidaat = Een kandidaatoplossing voor een deelprobleem. Denk bijvoorbeeld aan sudoku; stel dat je een getal in een vakje moet plaatsen, kan je dit gewoon aan 9 processoren doorgeven en wachten op hun resultaat, sommige van deze processoren zullen op het beslissingsprobleem 'nee' antwoorden, en sommige 'ja'.

Elk probleem uit P behoort dus zeker tot NP , want je kan gewoon het probleem aan de hoofdprocessor van de niet-deterministische computer geven. Men is er toch niet in geslaagd om aan te tonen dat de niet-deterministische computer krachtiger is dan de klassieke deterministische computer. Dit heet het $P - versus - NP$ probleem.

NP-harde problemen: Als we een probleem X kunnen reduceren naar een probleem Y , dan is Y minstens even zwaar als X . Een reductie is het omvormen van het ene probleem naar een ander. Er zullen dus problemen zijn die minstens even zwaar zijn als elk NP-probleem. Deze problemen noemen we NP-hard.

NP-complete problemen: Dit zijn NP-harde problemen die zeker in NP zitten. Een NP-complete problemen hebben een typische structuur die aan backtracking doen denken, waardoor het vaak vrij gemakkelijk door reductie van het ene naar het andere probleem over te gaan.

Indien er één NP-compleet probleem tot P zou behoren, dan behoren alle NP-complete problemen tot P . Op die manier wordt $NP = P$.

Als je kan bewijzen dat een bepaald probleem NP-compleet is, weet je dat je niet moet zoeken naar een efficiënte oplossing, maar kan je eerder gebruik maken van volgende technieken:

- Als de invoer van het probleem relatief klein is, kan je toch met backtracking alle mogelijkheden doorzoeken (bv 3x3 sudoku).
- Misschien bestaan er efficiënte algoritmen om speciale gevallen van het probleem op te lossen.
- Misschien is de gemiddelde uitvoeringstijd stukken beter dan het slechtste geval.
- Er bestaan benaderde algoritmen, die de vereisten voor de oplossing wat relaxeren. Sommige benaderingen zijn zelf NP-compleet.
- Maak gebruik van efficiënte heuristische methoden. Deze geven vaak niet de optimale oplossing, maar wel vaak een goede, maar het kan ook de slechtste oplossing zijn.

13.2 NP-complete problemen

13.2.1 Het basisprobleem: SAT (en 3SAT)

Gegeven een verzameling logische variabelen $\chi = \{x_1, \dots, x_n\}$ en een collectie logische uitspraken $\mathcal{F} = \{F_1, \dots, F_m\}$ Elke uitspraak F_j bestaat uit atomaire uitspraken x_i of de inverse $\overline{x_i}$ samengevoegd met ofoperaties zoals:

$$F_1 = x_2 \vee \overline{x_5} \vee x_7 \vee x_8 \vee x_9$$

De vraag is nu: kan men waarden toekennen aan de variabelen, zodat elke uitspraak F_j waar is?

Dit is natuurlijk eenvoudig op te lossen met backtracking, door incrementeel waarden toe te kennen.

Er werd bewezen dat elk probleem uit NP polynomiaal kan gereduceerd worden tot het SAT probleem.

Er is hier nog een uitbreiding op, waarbij elke uitspraak slechts 3 atomaire variabelen mag bevatten. We kunnen F_1 dan omvormen naar:

$$\begin{aligned} F'_1 &= x_2 \vee \overline{x_5} \vee x_n \\ F''_1 &= \overline{x_n} \vee x_7 \vee x_m \\ F'''_1 &= \overline{x_m} \vee x_8 \vee x_9 \end{aligned}$$

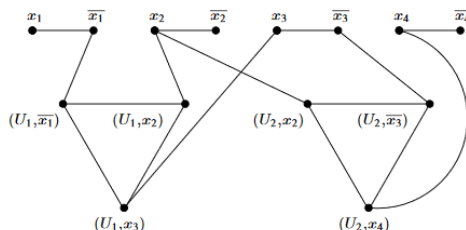
13.2.2 Graafproblemen

Vertex Cover

- △ Gegeven: Een ongerichte graaf.
- △ Gevraagd: De kleinste groep knopen die minstens één eindknoop van elke verbinding bevat.
- △ Reductie 3SAT tot vertex cover:
 - Voor elke logische variabele x_i , maken we twee knopen. De knoop x_i en zijn negatie $\overline{x_i}$. Deze twee zijn verbonden met elkaar.
 - Voor elke uitspraak maken we drie knopen, één voor elk atoom, en worden met elkaar verbonden.
- △ Voor een probleem met twee uitspraken:

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

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



Dominating set

- △ Gegeven: Een ongerichte graaf.
- △ Gevraagd: De kleinste groep knopen zodat elke andere knoop met minstens één van de knopen uit de groep verbonden is.

Graph Coloring

• Vertex coloring:

- △ Gegeven: Een ongerichte graaf.
- △ Gevraagd: Kleur de knopen met een minimum aantal kleuren, zodat de eindknopen van elke verbinding een verschillende kleur hebben.
- △ Enkel oplosbaar via backtracking of via heuristische methoden.

• Edge coloring:

- △ Gegeven: Een ongerichte graaf.
- △ Gevraagd: Kleur de verbindingen met een minimum aantal kleuren, zodat verbindingen met een gemeenschappelijke eindknoop een verschillende kleur hebben.
- △ Benaderend $O(n^2)$ algoritme van Vizing gebruikt hoogstens één kleur teveel.

Clique

- △ Gegeven: Een ongerichte graaf.
- △ Gevraagd: De grootste groep knopen die allemaal met elkaar verbonden zijn.
- △ Enkel oplosbaar via backtracking of heuristische methoden (bv simulated annealing).
- △ SAT herleiden tot clique:
 - Voor elk voorkomen van een atoom in een uitspraak maken we de bijbehorende knopen (F_i, x_j) en $(F_i, \overline{x_j})$.
 - Leg verbindingen tussen knopen horende bij verschillende uitspraken ($i \leq j$)
 - * (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 $k \neq l$.

Independent set

- △ Gegeven: Een ongerichte graaf
- △ Gevraagd: De grootste groep knopen zonder onderlinge verbindingen.

Hamiltonian circuit/path

- △ Gegeven: Een ongerichte of gerichte graaf.
- △ Gevraagd: Bestaat er een circuit dat elke knoop éénmaal bevat?
- △ Kan enkel opgelost worden met backtracking. Het is NP-compleet.

13.2.3 Problemen bij verzamelingen

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 één van de deelverzamelingen uit C' behoort.
- △ Kan efficiënt opgelost worden wanneer de deelverzamelingen in C niet meer dan twee elementen bevatten.
- △ SAT herleiden tot minimum cover:
 - Stel $S = \mathcal{F} \cup \mathcal{X}$
 - Voor elke logische variabele x_i zijn er twee elementen van C :

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

$$C_{\overline{x_i}} = \{\overline{x_i}\} \cup \{F \in \mathcal{F} : \overline{x_i} \in F\}$$
 - Het SAT-probleem heeft een oplossing als er een minimum cover is van grootte $|\mathcal{X}|$.

Subset sum

- △ Gegeven: Een verzameling elementen, elk met een positieve gehele grootte, en een positief geheel getal k .
- △ Gevraagd: Bestaat er een deelverzameling van die elementen, zodat de som van hun grootten gelijk is aan k ?
- △ Oplosbaar met dynamisch programmeren.

Partition

- △ Gegeven: Een zak van positieve getallen.
- △ Gevraagd: Kan de zak 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?
- △ Oplosbaar met dynamisch programmeren.

Hoofdstuk 14

Metaheuristieken

= vuistregels bij het zoeken naar een oplossing, maar geen garantie dat een oplossing gevonden wordt. Ze versnellen meestal de zoektocht ernaar.

14.1 Combinatorische optimalisatie

= Beste (hier het kleinste) element uit een verzameling \mathcal{S} vinden via evaluatiefunctie f . De verzameling \mathcal{S} is:

- een eindige verzameling van strings over een eindig alfabet,
- niet gekarakteriseerd door een opsomming, maar door voorwaarden.

△ Voorbeeld Travelling Salesman Problem:

- Alfabet = verzameling verbindingen.
- \mathcal{S} = reeks verbindingen zodat elke verbinding vertrekt uit het eindpunt van de vorige, en zo dat alle steden bezocht worden.
- Het beste individu is die met de kortse lengte.

△ Soms zijn alle strings in \mathcal{S} even lang, dan kan men de elementen beschrijven met variabelen $s \in \mathcal{S}$. Voorbeeld lessenroosterprobleem:

- Beschrijving: Een lessenrooster is goed wanneer deze weinig negatieve elementen heeft (springuren, onevenwichtige verdelingen, *zelf te kiezen wat 'negatief' is*).
- Elke string in \mathcal{S} beschrijft een lessenrooster zonder conflicten door aan elke les een lokaal en tijdsslot toe te kennen.
- \mathcal{S} heeft dus twee letters voor elke les: de eerste duidt de tijdsslot aan en de tweede het lokaal.

△ Voor elk individu wordt de f -waarde berekend, en het beste wordt natuurlijk bijgehouden.

△ De metaheuristiek stopt bij een bepaalde stopvoorwaarde, meestal is dit een aantal iteraties, tijd of de gevonden oplossing is benaderend goed genoeg.

△ Zeer groot aantal metaheuristieken

14.2 Vooronderstellingen

= De voorwaarden die voldaan moeten zijn zodat de methodieken in de cursus werken.

1. Het moet zinvol zijn om op zoek te gaan naar betere individuen in de buurt van een gegeven individu.
2. Soms is het niet zinvol om, bij het opbouwen nieuwe individuen van \mathcal{S} , uit te gaan van de reeds gevonden individuen.

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

1. Constructie: een individu wordt opgebouwd uit componenten. Een component komt overeen met letters van \mathcal{S} (Niet alle uitbreidingen bekijken zoals bij backtracking, maar één at random kiezen).
2. Individen worden rechtstreeks aangemaakt.

△ Twee manieren om kandidaten te nemen, indien ze opgebouwd worden uit componenten:

1. Shortlisting. Neem de beste kandidaten en kies hieruit at random.
2. Gewogen keuze. Elke keuze krijgt gewicht w_i . De kans $p(i)$ op die keuze wordt dan:

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

△ Drie manieren om een nieuw individu te kiezen uit \mathcal{S} , uitgaande van andere individuen die men reeds bekeken heeft:

1. Kleine wijzigingen aanbrengen in één reeds bekeken individu s van \mathcal{S} .
2. Kruisen van twee al bekeken individuen.
3. Het vermengen van grote hoeveelheden al bekeken individuen.

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

14.3 Lokaal versus globaal zoeken

- △ Lokaal wijzigen: een kleine wijziging (bv één of twee letters van het individu s veranderen).
- △ Voor elke $s \in \mathcal{S}$, is $\mathcal{N}(s)$ de omgeving die bestaat uit s zelf en alle strings uit \mathcal{S} die door één kleine wijziging uit s bekomen kunnen worden.
- △ Van s naar s' door willekeurige aaneenschakeling van wijzigingen.
- △ Vaak zit in de omgevingen van een goed individu, andere goede individuen.
- △ Iteratief toepassen: rij $s_0, s_1, \dots, s_n \in \mathcal{S}$ construeren zodanig dat voor elk individu in die rij in de omgeving zit van s_0 , en waarbij $f(s_{i+1}) < f(s_i)$. Deze rij vinden heet lokale optimalisatie.