Stochastische algoritmen

Tony BLEYS

promotor :
Prof. dr. Jan VAN DEN BUSSCHE

Eindverhandeling voorgedragen tot het bekomen van de graad
Licentiaat in de informatica afstudeervariant multimedia

universiteit hasselt

Samenvatting

Ik heb tijdens mijn thesis stochastische algoritmen bestudeerd. Ik heb mij vooral gebaseerd op het werk van Rajeev Motwani en Prabhakar Raghavan en hun boek Randomized Algoritmes. Ik ben begonnen met het bestuderen van stochastische algoritmen in het algemeen, tesamen met enkele technieken voor het bestuderen van deze algoritmes.

Daarna heb ik enkele specifieke toepasingen nader bekeken. De eerste toepssing is het gebruik van stochastische algoritmen bij datastructuren. Dit vooral omdat ik mij moeilijk kon voorstellen hoe men dit goed kon toepassen. Mijn verbazin was ook groot toen bleek dat dit niet alleen vrij makkelijk was, maar ook nog eens mooie resultaten behaalde.

De tweede toepassing die ik bestudeerd heb gaat over online algoritmes. Dit vooral omdat ik nog niet veel gezien had over online algoritmes en dit mij wel interessant leek.

Aan het begin van de thesis staat wat kanstheorie en een beetje wiskunde. Deze wordt gebruikt in de rest van de thesis. Voor deze statestiek heb ik gebruik gemaakt van de cursussen van kanstheorie van Prof. dr. Callaert en Prof. dr. Veraverbeke. De bewijzen in de hoofdstukken over stochastische algoritmen zijn in de meeste gevallen geïnspireerd door het boek zelf, maar ze zijn verder uitgewerkt, dikwijls versimpeld en soms ook helemaal zelf gemaakt. Ook de oefeningen zijn allemaal zelf gemaakt.

Inhoudsopgave

I Wiskunde **1**

1 Inleiding tot de Kanstheorie **2**
1.1 Wat is een kans? . . . 2
1.1.1 Universum . . . 2
1.1.2 Gebeurtenis . . . 3
1.1.3 Kans . . . 3
1.2 Voorwaardelijke Kans . . . 8
1.3 Onafhankelijkheid van Kansen . . . 9
1.4 Stochastische Veranderlijke . . . 9
1.4.1 Wat is de Verwachtingswaarde? . . . 10
1.4.2 Kansverdelingen . . . 11

2 Wiskundige Voorkennis **12**
2.1 Notaties, Definities en Eigenschappen . . . 12
2.1.1 Asymptotische Notaties . . . 12
2.1.2 Eigenschappen van Logaritmen . . . 12
2.2 Recursieve Vergelijkingen . . . 13
2.3 Eigenschappen van Harmonische getallen . . . 14
2.3.1 Afschatten van Harmonische Getallen . . . 14
2.3.2 Relatie tussen Harmonische getallen . . . 15

II Algoritmen **16**

3 Inleiding tot Stochastische Algoritmes **17**
3.1 Wat zijn Stochastische Algoritmes? . . . 17
3.1.1 Een Eerste Algoritme . . . 17
3.1.2 Een Tweede Algoritme . . . 21
3.1.3 Een Derde Algoritme . . . 25
3.2 Classificatie van Stochastische Algoritmes . . . 28
3.2.1 Las Vegas en Monte Carlo . . . 28
3.2.2 Omzetten van een Monte Carlo Algoritme . . . 29
3.2.3 Foutverkleining bij Monte Carlo Algoritmes . . . 30

3.2.4 Een Derde Soort 31
3.2.5 Efficiëntie en Stochastische Algoritmes 31
3.3 Computatie Model 31
3.3.1 Turing Machine 31
3.3.2 RAM Model 32
3.4 Complexiteitsklassen 33
3.4.1 Wat zijn Complexiteitsklassen? 33
3.4.2 De verschillende Complexiteitsklassen 34
3.5 Oefeningen 38

4 Speltheorie **44**
4.1 Spelboom Evaluatie 44
4.1.1 Wat is een Spelboom? 44
4.1.2 Benodigde Tijd voor het Evalueren van een Spelboom 44
4.2 Het Minimax Principe 47
4.2.1 Speltheorie 47
4.2.2 Yao's Techniek 49
4.2.3 Ondergrens voor het Evalueren van Spelbomen 50
4.3 Oefeningen 51

5 Data Structuren **54**
5.1 Het Fundamentele Data-Structuur Probleem 54
5.2 Algemene Oplossingen 55
5.2.1 Binaire Zoekbomen 55
5.2.2 Random Treaps 57
5.2.3 Skip Lists 64
5.3 Oefeningen 70

6 Online Algoritmes **72**
6.1 Het Online Paging Probleem 73
6.1.1 Wat is het Online Paging Probleem? 73
6.1.2 Efficiëntie Berekeningen met Deterministische Algoritmen 74
6.1.3 Efficiëntie Berekeningen met Stochastische Algoritmen 76
6.2 Oefeingen 83

Lijst van figuren

3.1 Graaf voor en na Samentrekking 23
3.2 Splitsing van Lijnstukken bij Binaire Planaire Partities 26

5.1 Voorbeeld van een skip list 65

Deel I

Wiskunde

Hoofdstuk 1

Inleiding tot de Kanstheorie

1.1 Wat is een kans?

De meeste mensen hebben een vaag idee wat een kans is. Zo weten de meeste mensen dat bij het opgooien van een eerlijk muntstuk de kans op kop even groot is als de kans op munt. Hetzelfde geldt voor eerlijke dobbelstenen: elk nummer heeft evenveel kans om bovenaan te liggen. Moeilijker wordt het wanneer men overschakelt naar ingewikkelder situaties zoals bijvoorbeeld de kans om de lotto te winnen of het aantal radioactieve deeltjes dat zal vervallen in één seconde. Om verwarring tegen te gaan zullen we nu gaan definiëren wat we nu bedoelen met het begrip kans.

1.1.1 Universum

In de kanstheorie worden situaties bestudeerd die aan het toeval onderhevig zijn. Een aantal van deze situaties staan vermeld in de vorige paragraaf. In het algemeen worden deze situaties experiment genoemd. Nu kunnen we meestal niets zeggen over de exacte uitkomst van een dergelijk experiment —anders konden we makkelijk de lotto winnen, maar we kunnen meestal wel iets zeggen over alle mogelijke uitkomsten van een experiment. Alle mogelijke uitkomsten van een experiment noemen we het universum van dat experiment.

Definitie 1.1.1. Het universum van een experiment is de verzameling van alle mogelijke uitkomsten van dit experiment. We noteren Ω voor het universum.

Noot 1.1.1. De verzameling Ω kan naargelang het experiment aftelbaar(zowel eindig als oneindig) of niet aftelbaar zijn.

Enkele voorbeelden van verschillende universa:

zes zijdige dobbelsteen werpen	$\Omega = \{1, 2, 3, 4, 5, 6\}$
munt opgooien	$\Omega = \{kop, munt\}$
aantal radioactieve desintegraties per uur	$\Omega = \{1, 2, ...\} = \mathbb{N}$
kies een willekeurige getal tussen 0 en 1	$\Omega = [0, 1]$

1.1.2 Gebeurtenis

Nu we weten wat een universum is kunnen we dat gebruiken om de uitkomsten te beschrijven. In het bijzonder kunnen we begrippen uit de verzamelingenleer aanwenden om de resultaten van experimenten te beschrijven. Neem als voorbeeld het klassieke experiment met een zeszijdige dobbelsteen. Daar is $\Omega = \{1, 2, 3, 4, 5, 6\}$. Nu komt een bepaalde eigenschap, vb gooi een getal kleiner dan 3, overeen met een deel verzameling van Ω, in dit geval de deelverzameling $\{1, 2\} \subset \Omega$. Een andere eigenschap komt overeen met een andere deelverzameling. Zo komt gooi een oneven getal overeen met $\{1, 3, 5\}$, ook weer een deelverzameling van Ω.

Definitie 1.1.2. Een gebeurtenis in de kanstheorie is een deelverzameling van het universum Ω

Nu kunnen we ook gaan rekenen met deze eigenschappen. Zij E_1 de gebeurtenis gooi een even getal met een zeskantige dobbelsteen, ofwel de deelverzameling $E_1 = \{2, 4, 6\} \subset \Omega$ waarbij $\Omega = \{1, 2, 3, 4, 5, 6\}$ is. Zij E_2 de gebeurtenis gooi een getal groter of gelijk aan 5, dus de deelverzameling $E_2 = \{5, 6\} \subset \Omega$. Nu kunnen we met behulp van verzamelingenleer ook andere gebeurtenissen beschrijven. Zo wordt gooi geen getal groter of gelijk aan 5 ofwel $\Omega \setminus E_2 = E_2^c$ waarbij de c het complement van de verzameling aanduidt. De gebeurtenis gooi een even getal groter of gelijk aan 5 wordt dan $E_1 \cap E_2 = \{2, 4, 6\} \cap \{5, 6\} = \{6\} \subset \Omega$. We zien dus dat als we willen dat een gebeurtenis voldoet aan twee voorwaarden, we de unie nemen van de deelverzamelingen van beide gebeurtenissen. Indien we de gebeurtenis willen waarbij we een getal gooien dat even is alsook groter of gelijk aan 5, dan komt het neer op het volgende: $E_1 \cap E_2 = \{2, 4, 6\} \cap \{5, 6\} = \{6\} \subset \Omega$, de doorsnede van de twee deelverzamelingen.

1.1.3 Kans

Nu we weten wat gebeurtenissen zijn, valt ons op dat bepaalde gebeurtenissen meer voorkomen dan anderen. Hoeveel meer of hoeveel makkelijker een gebeurtenis te realiseren valt geven we aan met een getal. Dat getal noemen we de kans van een gebeurtenis. We noteren $Pr[E_1]$ als de kans op gebeurtenis E_1. er zijn verschillende manieren om tot een definitie te komen voor $Pr[E_1]$ en deze zijn niet allemaal evenwaardig.

De Naïve Methode

Een eerste methode om tot een definitie te komen is redelijk naïef. Er wordt gebruik gemaakt van de intuïtie. Zo kan men zich bijvoorbeeld afvragen wat is de kans dat iemand een dobbelsteen gooit en het aantal ogen oneven is. De gebeurtenis is dan $E_1 = \{1, 3, 5\}$. De meeste mensen zullen vrij snel correct antwoorden dat de kans op deze gebeurtenis $1/2$ is. Bij deze methode tellen we het aantal gunstige uitkomsten en delen dat getal door het aantal mogelijke uitkomsten. In dit geval geeft dat de juiste uitkomst, maar er zijn een aantal problemen om dit te gebruiken in het algemeen. Ten eerste gaan we er van uit dat we alle mogelijke gebeurtenissen kunnen tellen, dus dat het er een eindig aantal zijn. Dit is niet altijd het geval. Ook gaan we er van uit dat alle mogelijke gevallen evenwaardig zijn. Ook dit is niet altijd het geval: beschouw bijvoorbeeld een niet eerlijke dobbelsteen. Al met al is deze methode niet ideaal.

De Relatieve Methode

Een tweede methode maakt gebruik van de relatieve frequentie. Neem terug het typische voorbeeld van een dobbelsteen, wat is de kans op een oneven getal dus $E_1 = \{1, 3, 5\}$? In dit geval echter weten we niet als we te doen hebben met een eerlijke of met een oneerlijke dobbelsteen. Tellen van het aantal mogelijke gevallen is hier niet nuttig. Wat ons wel een echt beeld kan geven is het herhaaldelijk uitvoeren van het experiment (het gooien met de dobbelsteen) en dan meten hoeveel keer de gebeurtenis E_1 zich voordoet. Als we dan het aantal keren dat gebeurtenis E_1 zich voordeed delen door het aantal keer dat we het experiment uitvoerden komen tot het begrip relatieve frequentie, notatie $\frac{n(E_1)}{n}$ met $n(E_1)$ het aantal keer dat E_1 voorkomt en n het aantal keer dat het experiment uitgevoerd werd. Hoe meer het experiment uitgevoerd wordt, hoe preciezer de resultaten. We definiëren het kansbegrip dan ook als $Pr(E_1) = \lim_{n\to\infty} \frac{nE_1}{n}$. Deze definitie is redelijk goed, zodanig dat hij overeenkomt met onze intuïtie en voor praktische voorbeelden klopt ze ook. Echter is deze definitie niet geheel sluitend en kan ze niet gebruikt worden voor het opstellen van een puur wiskundig axiomatische opbouw van de kans theorie.

De Wiskundig-Axiomatische Methode

De wiskundig axiomatische methode, die gebruikt kan worden voor het opstellen van exacte wiskundige bewijzen, maakt gebruik van verzamelingenleer. We gaan proberen een kans te associëren met deelverzamelingen van het universum Ω. Het is niet altijd mogelijk om dat te doen voor alle deelverzamelingen van Ω. Daarom keizen we een welbepaalde familie van deze deelverzamelingen. Wel moet deze familie groot genoeg voor het

oplossen van praktische problemen. Logisch gezien, als een deelverzameling E_1 overeenkomt met het hebben van een bepaalde eigenschap, zoals het oneven zijn van het resultaat, is het handig om ook de deelverzameling te beschouwen waarin deze eigenschap niet opgaat. Dat is dan de verzameling $\Omega \backslash E_1 = E_1^c$ waarbij we E_1^c het complement van E_1 noemen. Het complement van het universum Ω is de lege verzameling $\emptyset$.

Ook van belang zijn de combinaties van verschillende eigenschappen. Zo staat het hebben van twee verschillende eigenschappen zoals "is oneven" en "is groter dan 4" gelijk aan de doorsnede van de deelverzamelingen voor "oneven" en "groter dan 4". Indien we óf "groter dan 4" óf "oneven" willen kunnen we de unie van deze twee deelverzamelingen nemen. Op deze manier kunnen we een familie deelverzamelingen van Ω definiëren waaraan we dan kansen kunnen toewijzen.

Definitie 1.1.3. Een σ-algebra $\mathbb{F}$ over het universum Ω is een niet-lege familie deelverzamelingen van Ω, zodat:

1: $E \in \mathbb{F} \Rightarrow E^c \in \mathbb{F}$

2: $E_n \in \mathbb{F}, n = 1, 2, ... \Rightarrow \cup_{n=1}^{\infty} E_n \in \mathbb{F}$

Opmerking. De elementen van $\mathbb{F}$ noemen we gebeurtenissen.

De bovenstaande definitie heeft een aantal gevolgen:

1: $E_n \in \mathbb{F}, 1, 2, ..., N \Rightarrow \cup_{n=1}^{N} E_n \in \mathbb{F}$
wegens de tweede eigenschap uit definitie 1.1.3. Neem $A_n = \emptyset$ voor $n > N$.

2: $\Omega \in \mathbb{F}$
wegens eigenschap 1 uit definitie 1.1.3 is zowel A als A^c een element van $\mathbb{F}$. Wegens het eerste gevolg is dus ook $A \cup A^c \in \mathbb{F}$ en $A \cup A^c = \Omega$ en dus $\Omega \in \mathbb{F}$.

3: $\emptyset \in \mathbb{F}$
Uit het vorige gevolg en de eerste eigenschap van definitie 1.1.3 $\emptyset = \Omega^c \in \mathbb{F}$.

4: $A_n \in \mathbb{F}, 1, 2, ... \Rightarrow \cap_{n=1}^{\infty} A_n \in \mathbb{F}$
want $A_n \in \mathbb{F} \Rightarrow A_n^c \in \mathbb{F}$ wegens de eerste eigenschap definitie 1.1.3. Uit de tweede eigenschap: $\cup_{n=1}^{\infty} A_n^c \in \mathbb{F}$. Combineren van het voorgaande leidt tot: $(\cup_{n=1}^{\infty} A_n^c)^c \in \mathbb{F}$. Nu passen we de wetten van de Morgen toe op het eerste deel en dus $\cap_{n=1}^{\infty} A_n \in \mathbb{F}$.

Nu we duidelijk gedefinieerd hebben op welke soort families van deelverzamelingen we gaan werken kunnen we eindelijk de kansen op gebeurtenissen gaan definiëren.

Definitie 1.1.4. Zij Ω een universum en $\mathbb{F}$ een σ-algebra over Ω
Een kansmaat Pr op $\mathbb{F}$ is een functie
$Pr : \mathbb{F} \to \mathbb{R}$
$A \mapsto Pr(A)$ zodanig dat

1: $Pr(\Omega) = 1$

2: $\forall A \in \mathbb{F} : Pr(A) > 0$

3: $A_n \in \mathbb{F}, n = 1, 2, ...,$
$A_n \cap A_m, n \neq m :$
$Pr\left(\cup_{n=1}^{\infty} A_n\right) = \sum_{n=1}^{\infty} Pr(A_n)$

Ook deze definitie heeft een aantal gevolgen.

1: $Pr(\emptyset) = 0$
$\emptyset = \emptyset \cup \emptyset \cup ...$
$Pr(\emptyset) = Pr(\emptyset) + Pr(\emptyset) + ...$ wegens derde eigenschap uit definitie 6
Nu zit $Pr(\emptyset)$ in $\mathbb{R}$ en het enige element uit $\mathbb{R}$ dat bij zichzelf opgeteld zichzelf geeft is het neutraal element van de optelling, namelijk 0 □

2: $\forall A_n \in \mathbb{F}, n = 1, ..., N$, paarsgewijs disjunct, geldt:

$$Pr\left(\cup_{n=1}^{N} A_n\right) = \sum_{n=1}^{N} Pr(A_n)$$

Wegens $\cup_{n=1}^{N} A_n = A_1 \cup ... \cup A_n \cup \emptyset \cup \emptyset \cup ...$ het gevraagde volgt dan uit de derde eigenschap van definitie en bovenstaand gevolg. □

3: $\forall A, B \in \mathbb{F} : Pr(B) = Pr(B \cap A) + Pr(B \cap A^c)$
$B = B \cap \Omega = B \cap (AcupA^c) = (B \cap A) \cup (B \cap A^c)$ pas nu weer de derde eigenschap van definitie toe. □

4: $\forall A \in \mathbb{F} : Pr(A) = 1 - Pr(A^c)$
wegens het voorgaande $Pr(\Omega) = Pr(A) + Pr(A^c)$
$Pr(\Omega) = 1$ wegens het eerste gevolg en dus $1 = Pr(A) + Pr(A^c) \Leftrightarrow$
$Pr(A) = 1 - Pr(A^c)$ □

5: $\forall A, B \in \mathbb{F} : A \subset B \Rightarrow Pr(A) \leq Pr(B)$
$A \subset B \Rightarrow A \cap B = A$
uit gevolg 3 volgt dan $Pr(B) = Pr(A) + Pr(B \cap A^c)$
Nu wegens de tweede eigenschap uit definitie is $Pr(B\ capA^c) \geq 0$ en dus $Pr(A) + Pr(B \cap A^c) \geq Pr(A)$ waaruit het gevraagde volgt □

6: $\forall A \in \mathbb{F} : Pr(A) \leq 1$
$\forall A \in \mathbb{F} : A \subset \Omega$ uit de eerste eigenschap uit definitie en het voorgaande gevolg hebben we rechtstreeks het gevraagde □

7: $\forall A, B \in \mathbb{F} : Pr(A \cup B) = Pr(A) + Pr(B) - Pr(A\ capB)$
$A \cup B = (A \cap B^c) \cup (A \cap B) \cup (A^c \cap B)$
wegens het tweede gevolg $Pr(A \cup B) = Pr(A \cap B^c) + Pr(A \cap B) + Pr(A^c \cap B)$
rekenen met behulp van het derde gevolg geeft ons: $Pr(A \cup B) = Pr(A) - Pr(A \cap B) + Pr(A\ capB) + Pr(B) - Pr(A \cap B)$
verder rekenen geeft: $Pr(A \cup B) = Pr(A) + Pr(B) - Pr(A\ capB)$ $\square$

8: $\forall A_n \in \mathbb{F}, n = 1, 2, ..., N :$

$$Pr\left(\cup_{n=1}^{N} A_n\right) \leq \sum_{n=1}^{N} Pr(A_n)$$

Bewijs. Bewijs door inductie.
Het gevraagde geldt voor $n = 2$. Dit volgt rechtstreeks uit het zevende gevolg.
Inductiestap
We weten nu dat het geld voor alle waarden tot en met N. Nu bewijzen we dat het ook geldt voor $N + 1$.

$$Pr\left(\cup_{n=1}^{N+1} A_n\right) = Pr\left(\cup_{n=1}^{N} A_n \cup A_{N+1}\right)$$

$$\Updownarrow \text{ zij } \cup_{n=1}^{N} A_n = A'$$

$$Pr\left(\cup_{n=1}^{N+1} A_n\right) = Pr\left(A' \cup A_{N+1}\right)$$

$$\Updownarrow \text{ wegens } Pr(A \cup B) = Pr(A) + Pr(B) - Pr(A \cap B) \text{ en } Pr(A' \cap A_{N+1}) \geq 0$$

$$Pr\left(\cup_{n=1}^{N+1} A_n\right) \leq Pr(A') + Pr(A_{N+1})$$

$$\Updownarrow$$

$$Pr\left(\cup_{n=1}^{N+1} A_n\right) \leq Pr(\cup_{n=1}^{N} A_n) + Pr(A_{N+1})$$

$$\Updownarrow \text{ de formule geldt voor } n \leq N$$

$$Pr\left(\cup_{n=1}^{N+1} A_n\right) \leq \sum_{n=1}^{N} Pr(A_n) + Pr(A_{N+1})$$

$$\Updownarrow$$

$$Pr\left(\cup_{n=1}^{N+1} A_n\right) \leq \sum_{n=1}^{N+1} Pr(A_n)$$

□

Opmerking. Het drietal $(\Omega, \mathbb{F}, Pr)$ noemen we een kansruimte.

1.2 Voorwaardelijke Kans

Als we een experiment uitvoeren dan kunnen we nu de kans op een bepaalde gebeurtenis G berekenen met behulp van de kansruimte die dat experiment beschrijft. Als we nu extra informatie krijgen over de uitkomst van het experiment, kunnen we ons afvragen wat dan de kans is op dezelfde gebeurtenis G. De extra informatie die gegeven wordt, laat toe om een deel van het universun uit te sluiten. Het is dan ook logisch om een nieuw universum te definiëren waarin alleen die uitkomsten zitten die niet worden uitgesloten door de extra informatie.

Neem bijvoorbeeld het klassieke geval met de zeszijdige dobbelsteen. Als we ons afvragen wat de kans op het gooien van vier is, dan weten we dat het antwoord 1/6 is. Als we echter ook gegeven krijgen dat er een even aantal ogen gegooid is, dan is deze kans duidelijk hoger. Het universum wordt dan $\Omega_1 = \{2, 4, 6\}$. Nu kunnen we een nieuwe σ-algebra definiëren met deelverzamelingen van de vorm $\Omega_1 \cap C$ met $C \subseteq \Omega$, dus we nemen de doorsnede van het nieuwe universum met de oude deelverzamelingen van de σ-algebra. Dit is opnieuw een σ-algebra. We definiëren ook een nieuwe kansmaat Pr_1. We weten dat $Pr_1(\Omega_1) = 1$ moet zijn. Als Pr de kansmaat is van de oorspronkelijke kansruimte, dan wordt de nieuwe kansmaat $Pr_1(G) = \frac{Pr(G)}{Pr(\Omega_1)}$. Dit geldt natuurlijk alleen wanneer $Pr(\Omega_1) > 0$.

Met bovenstaande definities is het dan zo dat de kans op een gebeurtenis G wanneer we weten dat Ω_1 gebeurd is, gelijk is aan: $Pr_1(G \cap \Omega_1) = \frac{Pr(G \cap \Omega_1)}{Pr(\Omega_1)}$. We noteren $Pr_1(G \cap \Omega_1)$ als $Pr(G|\Omega_1)$, de voorwaardelijke kans op gebeuretenis G, gegeven dat Ω_1. Formeel krijgen we dan de volgende definitie:

Definitie 1.2.1. Zij $(\Omega, \mathbb{F}, P)$ een kansruimte; Zij $\Omega_1 \in \mathbb{F}$ en $Pr(\Omega_1) > 0$. De voorwaardelijke kans op gebeurtenis $G \in \mathbb{F}$, gegeven Ω_1 noteren we als $Pr(G|\Omega_1)$ en gedefinieerd als

$$Pr(G|\Omega_1) = \frac{P(G \cap \Omega_1)}{P(\Omega_1)} \tag{1.1}$$

Als $Pr(\Omega_1) = 0$ dan is Ω_1 niet gedefinieerd.

1.3 Onafhankelijkheid van Kansen

Als we twee munten opgooien, en we weten dat de tweede munt als resultaat KOP heeft, dan weten we nog steeds niet over het resultaat van de eerst munt. Met andere woorden de informatie over de tweede munt zegt niets over de eerste munt. In formule vorm geeft dat:

$$P[E_1 \mid E_2] = P[E_1]$$

Als dit het geval is zeggen we dat de twee gebeurtenissen onafhankelijk van elkaar zijn.

Definitie 1.3.1. Twee gebeurtenissen E_1 en E_2 zijn onafhankelijk als

$$P[E_1 \cap E_2] = P[E_1] \times P[E_2]$$

Definitie 1.3.2. Een gorep gebeurtenissen $E_1, E_2, ..., E_n$ $(n \geq 2)$ is onderling onafhankelijk als voor elke keuze van τ verschillende indices $i_1, i_2, ..., i_r$ uit $1, 2, ..., n$ en voor elke $r = 2, ..., n$ geldt:

$$P[E_{i_1} \cap E_{i_2} \cap ... \cap E_{i_r}] = P[E_{i_1}] \times P[E_{i_2}] \times ... \times P[E_{i_r}]$$

Definitie 1.3.3. We zeggen dat een oneindige familie gebeurtenissen $E_1, E_2, ...$ onderling onafhankelijk is als elke eindige deelfamilie onderling onafhankelijk is in de zin van definitie 1.3.2

1.4 Stochastische Veranderlijke

We willen nu systematisch gaan werken. In principe hebben we het toch altijd over uitkomsten en de kans op deze uitkomst. Als we deze uitkomsten voorstellen door reële getallen hebben we altijd $\mathbb{R}$ of een deel ervan als nieuw universum. Nu kunnen we de overgang van Ω naar $\mathbb{R}$ voorstellen als een reële functie als volgt:

$$X : \Omega \to \mathbb{R} : \omega \mapsto X(\omega)$$

Deze functie noemen we de stoschastische veranderlijke en wordt meestal gewoon voorgesteld door X.

Nu kunnen we het reciproce beeld van een waarde x onder X gaan beschouwen. Het is duidelijk dat dit een deelverzameling is van Ω en dus een gebeurtenis is. Dit kunnen we noteren als $(X = x)$ en nu kunnen we de kans gaan berekenen dat de stochastische veranderlijke de waarde x aanneemt. Deze is natuurlijk gelijk aan de kans op de gebeurtenis die we gevonden hadden onder het reciproce beeld van x onder X.

1.4.1 Wat is de Verwachtingswaarde?

Nu we een functie hebben gedefiniëerd die afhangt van kansen, kunnen we ons gaan afvragen wat de waarde van deze functie waarschijnlijk zal zijn. We definiëren de verwachtingswaarde als volgt:

Definitie 1.4.1. $E[X] = \sum_i x_i P[X = x_i]$

Dit komt neer op het een groot aantal keer evalueren van X en alle resultaten optellen en delen door het aantal. We kunnen dus ook spreken van eht gemiddelde van een stochastische veranderlijke.

Lineariteit van de Verwachting

De verwachtingswaarde van een stochastische veranderlijke heeft als belangrijkste eigenschap dat hij lineair is, dus dat Voor de stochastische veranderlijke $X_1, ..., X_n$ geldt: $E[\sum_i X_i] = \sum_i (E[Xi])$. Om de belangrijkheid hiervan aan te tonen het volgende voorbeeld.

Probleem:
Een schip komt aan in een haven, 40 matrozen mogen aan land. 's Avonds komen deze 40 matrozen beschonken terug en kiezen een willekeurige slaapplaats uit. Hoeveel matrozen slapen op hun eigen slaapplaats?

Oplossing:
Alle 40^40 arragementen nakijken en tellen. Het is duidelijk dat dit niet erg efficiënt is. Daarom een efficiëntere manier: een indicator variabele en gebruik maken van de lineariteit van de verwachtingwaarde. Zij $X_i = 1$ als de $i$de matroos in zijn eigen cabine slaapt en 0 anders. Nu is het verwachte aantal matrozen in hun eigen slaapcabine gelijk aan $E[\sum_{i=1}^{40} X_i]$. Nu door de lineariteit van de verwachtingswaarde toe te passen kunen we dit herschrijven tot $\sum_{i=1}^{40} E[X_i]$, nu de kans dat $X_i = 1$ oftewel de matroos kiest zijn eigen cabine is $\frac{1}{40}$ dus $E[X_i] = \frac{1}{40}$ en dus het verwachte aantal matrozen in hun eigen cabine is 1.

We kunnen dit bewijzen als volgt:

Bewijs.

$$E[\sum_i X_i] = \sum_j \sum_i x_j P[X_i = x_j]$$

uit de definitie. Omwisselen van de somaties geeft:

$$E[\sum_i X_i] = \sum_i \sum_j x_j P[X_i = x_j]$$

dan terug de definitei toepassen.

$$E[\sum_i X_i] = \sum_i E[X_i]$$

□

1.4.2 Kansverdelingen

In deze sectie worden de twee kansverdelingen vermeld die veel gebrukt worden in de rest van de thesis.

De Uniforme Verdeling

Bij deze verdeling heeft elke uitkomst dezelfde kans omzich voor te doen. Als X dus n verschillende waarden kan aannemen dan zal de kans dat X een specifieke waarde aanneemt gelijk zijn aan $\frac{1}{n}$. Als we $x_1, x_2, ..., x_n$ gelijk stellen aan $1, 2, ..., n$ dan zal de verwachtingswaarde gelijk zijn aan $\frac{n+1}{2}$.

De Geometrische Verdeling

Dit is de tweede verdeling die veel voorkomt in deze thesis. Hij komt op het volgende neer. Voor een experiment uit dat twee mogelijk uitkomsten heeft, zoals het opgooien van een munt. Noem de ene mogelijke uitkomst succes en de andere mislukking. De kans op succes is gelijk p, die voor een mislukking is $1 - p = q$. Als we weer $x_1, x_2, ..., x_n$ gelijk stellen aan $1, 2, ..., n$ dan zal de verwachtingswaarde gelijk zijn aan $\frac{1}{p}$

Hoofdstuk 2

Wiskundige Voorkennis

2.1 Notaties, Definities en Eigenschappen

2.1.1 Asymptotische Notaties

Definitie 2.1.1. Laat $f(x)$ en $g(x) : \mathbb{R} \to \mathbb{R}$ twee niet-negatieve functies met reële waarden zijn. Dan:

- $f(x) = O(g(x))$ als er positieve getallen c en X zodat voor alle $x \geq X$, $f(x \leq cg(x))$
- $f(x) = \Omega(g(x))$ als er positieve getallen c en X zodat voor alle $x \geq X$, $f(x \geq cg(x))$
- $f(x) = \Theta(g(x))$ als zowel $f(x) = O(g(x))$ als $f(x) = \Omega(g(x))$ geldig zijn.
- $f(x) = o(g(x))$ als $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$
- $f(x) \sim g(x)$ als $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 1$

2.1.2 Eigenschappen van Logaritmen

Definitie 2.1.2. Een logaritme $\log_b x$ voor een basis b en een getal x is de inverse functie van het verheffen van b tot de macht x. Dus $x = \log_b b^x$.

Veel gebruikte logaritmen zijn deze met als basis e. Deze noemt men natuurlijke logaritmen en worden meestal genoteerd als $\ln(x) = \log_e x$. Andere veel gebruikte logaritmen zijn deze met een basis 10. Deze noemt men de Briggse logaritmen. Deze worden soms genoteerd met $\log x = \log_{10} x$, maar deze notatie is niet algemeen. Wiskundigen gebruiken de noatie $\log x$ ook wel voor de natuurlijke logaritmen. In de informatica wordt ook veel gebruik gemaakt van logaritmen met als basis 2. Informatici gebruiken om

die reden de notatie $\log x = \log_2 x$ ook wel eens. In de rest van deze thesis zal er dan ook met deze notatie gewerkt werden.

Er volgen nu enkele rekenregels voor logaritmen.

- $\log_b xy = \log_b x + \log_b y$
- $\log_b \frac{x}{y} = \log_b x - \log_b y$
- $\log_b x^n = n \log_b x$
- $\log_b x = \frac{\log_n x}{\log_n b}$

2.2 Recursieve Vergelijkingen

De recursieve vergelijking die we op enkele plaatsen gebruiken is

$$T(n) = 2T(\frac{n}{2}) + cn \tag{2.1}$$

We willen nu een meer algemene vorm oplossen. De meer algemene vergelijking is de volgende:

$$T(n) = T(\frac{n}{k}) + T(\frac{(k-1)n}{k}) + cn \tag{2.2}$$

Uit de vergelijking (2.2) halen we dat

$$T(\frac{n}{k}) = T(\frac{n}{k^2}) + T(\frac{(k-1)n}{k^2}) + \frac{cn}{k} \tag{2.3}$$

en

$$T(\frac{(k-1)n}{k}) = T(\frac{(k-1)n}{k^2}) + T(\frac{(k-1)^2 n}{k^2}) + \frac{(k-1)cn}{k} \tag{2.4}$$

Als we (2.3) en (2.4) invullen in (2.2) krijgen we:

$$T(n) = T(\frac{n}{k^2}) + T(\frac{(k-1)n}{k^2}) + \frac{cn}{k} \tag{2.5}$$

$$+T(\frac{(k-1)n}{k^2}) + T(\frac{(k-1)^2 n}{k^2}) + \frac{(k-1)cn}{k} \tag{2.6}$$

$$+cn \tag{2.7}$$

verder uitwerken geeft dan:

$$T(n) = T(\frac{n}{k^2}) + 2T(\frac{(k-1)n}{k^2}) + T(\frac{(k-1)^2 n}{k^2}) + \frac{(k-1)cn}{k} + 2cn \tag{2.8}$$

Nu kunnen we dit process van in vullen blijven herhalen. Op elke stap zullen de argumenten van T kleiner worden. Na verloop van tijd gaan er dus termen met T weg vallen. Tijdens de $i$de stap is $\frac{(k-1)^i n}{k^i}$ het grootste

argument van om het even welke term met T. Deze term zal dus altijd als laatste wegvallen. Bij elke stap komt er hoogstens cn bij de termen die niet afhangen van T. Hieruit volgt dat het resultaat kleiner is dan het aantal stappen dat nodig is om de grootste term in T te laten verdwijnen vermenigvuldigt met cn. Dus:

$$T(n) \leq \log_{\frac{k-1}{k}} ncn \tag{2.9}$$

door gebruik te maken van de eigenschappen van logaritmen bekomen we:

$$T(n) \leq \frac{\log n}{\log \frac{k}{k+1}} cn \tag{2.10}$$

verder uitwerken en $c' = \frac{c}{\log \frac{k}{k+1}}$ geeft dan:

$$T(n) \leq \log nc'n = O(n \log n) \tag{2.11}$$

2.3 Eigenschappen van Harmonische getallen

Definitie 2.3.1. voor elk natuurlijk getal $n \in \mathbb{N}$, is het $n$de Harmonisch getal:

$$H_n = 1 + \frac{1}{2} + ... + \frac{1}{n} = \sum_{i=1}^{n} \frac{1}{i}$$

2.3.1 Afschatten van Harmonische Getallen

Theorema 2.3.1. *We kunnen voor elk natuurlijk getal $n \in \mathbb{N}$ het $n$de Harmonische getal afschatten als:* $H_n = \ln n + \Theta(1)$

Bewijs. Zei $a_n = \int_n^{n+1} \frac{1}{x} dx$
dan geldt:

$$a_n < \frac{1}{n} < a_{n-1} \; \forall n \in \mathbb{N}_0$$

wegens dat $\frac{1}{x}$ een strict dalende functie is, en $\frac{1}{n}$ is de bovengrens voor a_n en de ondergrens voor a_{n-1}
nu nemen we de som van 1 tot en met n. We krijgen dan:

$$a_1 + a_2 + ... + a_n < H_n \leq 1 + a_1 + a_2 + ... + a_{n-1} \; \forall n \in \mathbb{N}_0$$

we weten ook dat wegens de definitie van a_n als we de integraal uitrekenen $a_n = \ln(n+1) - \ln n$ en dus:

$$\ln(n+1) < H_n \leq \ln n + 1 \; \forall n \in \mathbb{N}_0$$

wat reken werk leidt tot:

$$H_n - \ln n > H_n - \ln(n+1) > 0 \; \forall n \in \mathbb{N}_0$$

uit het bovenstaande en:

$$H_{n+1} - \ln(n+1) = H_n + \frac{1}{n+1} - \ln n - a_n < H_n - \ln n \; \forall n \in \mathbb{N}_0$$

volgt dat $H_n - \ln n$ altijd positief is en daalt naarmate n stijgt voor alle natuurlijke getallen groter dan 0. In werkelijkheid gaat deze waarde naar een limiet als n naar oneindig gaat. Men noemt deze limiet de Euler-Mascheroni constante, die men noteert als γ en zij bedraagt ongeveer $0,577215$. Met behulp van deze resultaten bekomen we:

$$\gamma < H_n - \ln n \leq 1 \; \forall n \in \mathbb{N}_0$$

rekenen geeft ons:

$$\gamma + \ln n < H_n \leq 1 + \ln n \; \forall n \in \mathbb{N}_0$$

uit deze vergelijking kunnen we besluiten dat er een functie $g(x)$ bestaat zodat $H_n = \ln n + g(n)$. $g(x) = O(1)$, want $g(x)$ heeft een maximum, namelijk 1. Ook geldt $g(x)\Omega(1)$, want $g(x)$ heeft een minimum, namelijk γ. Daar $g(x) = O(1)$ en $g(x) = \Omega(1)$ geldt dus ook $g(x) = \Theta(1)$. en dus geldt:

$$H_n = \ln n + \Theta(1)$$

□

2.3.2 Relatie tussen Harmonische getallen

Theorema 2.3.2. *Voor alle Harmonische getallen geldt:*

$$\sum_{k=1}^{n} H_k = (n+1)H_(n+1) - (n+1)$$

Bewijs.

$$\sum_{k=1}^{n} H_k = n \times 1 + (n-1) \times \frac{1}{2} + (n-2) \times \frac{1}{3} + ... + 1 \times \frac{1}{n}$$

Dit is gewoon het uitschrijven ven het eerste lid. Vervolgens tellen we bij het tweede lid een deel op, maar trekken we die er ook terug af. dan krijgen we:

$$\sum_{k=1}^{n} H_k = (n+1) \times 1 + (n+1) \times \frac{1}{2} + (n+1) \times \frac{1}{3} + ... + (n+1) \times \frac{1}{n} + (n+1) \times \frac{1}{n+1}$$
$$-1 \times 1 - 2 \times \frac{1}{2} - ... - (n) \times \frac{1}{n} - (n+1) \times \frac{1}{n+1}$$

als we dan weer termen samen nemen bekomen we het volgende resultaat:

$$\sum_{k=1}^{n} H_k = (n+1)H_{n+1} - n + 1 \times 1$$

en dit laatste is bewijs voor het gevraagde. □

Deel II

Algoritmen

Hoofdstuk 3

Inleiding tot Stochastische Algoritmes

In dit hoofdstuk wordt uitgelegd wat randomized algoritmes zijn. Aan de hand van de voorbeelden van deze algoritmes worden de begrippen uit de kanstheorie en de wiskundige voorkennis uit het vorige deel toegepast.

3.1 Wat zijn Stochastische Algoritmes?

Er zijn verschillende soorten van randomized algoritmes. We beginnen met enkele voorbeelden die deze verschillende groepen illustreren, om een idee te krijgen van de verschillende toepassingen.

3.1.1 Een Eerste Algoritme

Een probleem en een oplossing

Neem een verzameling S van n getallen. De bedoeling is om deze verzameling getallen te gaan sorteren van klein naar groot. Er zijn vele manieren bekend om dit soort problemen op te lossen maar beschouw de volgende:

stap 1 Neem een element y uit S

stap 2 Maak 2 verzamelingen S_1 en S_2 en stop alle elementen van S die kleiner zijn dan y in S_1, en alle elementen die groter zijn dan y in S_2

stap 3 Pas dit algoritme recursief toe op S_1 en S_2

stap 4 De output is S_1 gevold door y gevolgd door S_2

Het bovenstaande algoritme zal de meeste wel bekend zijn onder de naam quicksort. Het grootste probleem van quicksort is het maken van een goede keuze voor het element y uit de eerste stap. Er zijn diverse methoden ontwikkeld om snel een goede keuze voor y te bekomen, maar bijna al deze methoden maken het algoritme ingewikkelder.

Looptijd van Quicksort

Als we aannemen dat we y zouden kunnen vinden in cn stappen voor een bepaalde constante c, het opsplitsen van $S\backslash\{y\}$ in S_1 en S_2 kan in $n-1$ extra stappen door elk element van S te vergelijking met y en dan wordt het totaal aantal stappen van quicksort gegeven door de recursieve vergelijking:

$$T(n) \leq 2T(n/2) + (c+1)n \tag{3.1}$$

De oplossing voor deze recursieve vergelijking kan gevonden worden door middel van substitutie en deze oplossing is:

$$T(n) \leq c'n \log n \tag{3.2}$$

De moeilijkheid die hierbij ontstaat, is het goed kiezen van y. Nu moet het niet zijn dat de gekozen y de verzamelingen S_1 en S_2 exact even groot maakt. Indien S_1 en S_2 ongeveer even groot zijn, ongeveer tussen $1/4$ en $3/4$ van de grootte van de oorspronkelijke verzameling S wordt in het algemeen als aanvaardbaar beschouwd. De gemiddelde looptijd van quicksort en varianten is $nlogn$, zoals blijkt uit vergelijking (3.2), terwijl de worst-case n^2 is.

Randomized Quicksort

Nu bestaan er vele manieren om te proberen y goed te kiezen, maar nu stellen we voor om gewoon een y willekeurig te kiezen uit de verzameling S. Het gekozen element y zal niet altijd voldoen aan de gestelde voorwaarde, maar zal in de meeste gevallen wel een redelijke keuze voor y zijn. Het algoritme wordt dan:

Input: een verzameling getallen S

Output: de getallen van S in stijgende volgorde

stap 1 kies op op een volkomen willekeurig, volgens een uniforme verdeling, een element y uit S, zodat elk element uit S evenveel kans heeft om gekozen te worden

stap 2 vergelijk elk element uit S met y en verdeel ze over 2 verzamelingen S_1 en S_2 met alle elementen kleiner dan y in S_1 en alle elementen groter dan y in S_1

stap 3 Sorteer S_1 en S_2 recursief.

stap 4 De output is de gesorteerde versie van S_1 gevolgd door y gevolgd door S_2

Looptijd van Randomized Quicksort

Om aan te tonen dat Randmoized Quicksort (kortweg RandQS) goed presteert, gaan we de verwachte looptijd uitrekenen. De looptijd van een algoritme is evenredig met het aantal vergelijkingen dat het algoritme uitvoert. Alle vergelijkingen die RandQS uitvoert, gebeuren in stap 2 van het bovenstaande algoritme. Om de verwachte looptijd van RandQS te berekenen, berekenen we gewoon het verwachte aantal vergelijkingen in de tweede stap waar we elk element in de verzameling S vergelijken met het random gekozen getal y.

Laat de verzameling Sn elementen bevatten. Dan, voor $1 \leq i \leq n$ noteer $S_{(i)}$ voor het $i$de kleinste element van S. Dus $S_{(1)}$ is het kleinste element van S, $S_{(2)}$ het op één na kleinste enzoverder tot $S_{(n)}$ het grootste element van S. Op deze manier staat elke $S_{(i)}$ voor het element van $i$de rang uit de verzameling S. Definiëer de stochastische veranderlijke X_{ij}, X_{ij} krijgt de waarde 1 indien $S_{(i)}$ en $S_{(j)}$ met elkaar vergeleken worden tijdens de uitvoering van het algoritme. In het andere geval krijgt X_{ij} de waarde 0. Op deze manier vormt X_{ij} een maat voor het aantal vergelijkingen tussen $S_{(i)}$ en $S_{(j)}$, waardoor het totaal aantal vergelijkingen gelijk is aan $\sum_{i=1}^{n} \sum_{j>i} X_{ij}$. (Het is niet $\sum_{i=1}^{n} \sum_{j=1}^{n} X_{ij}$ omdat voor concrete i en j X_{ij} en X_{ji} gelijk zijn aan elkaar en we deze dan dubbel tellen.)

Nu we weten wat het totaal aantal vergelijkingen is, kunnen we het verwacht aantal vergelijkingen tijdens een typische run van RandQS berekenen. Het verwachte aantal vergelijkingen wordt dan:

$$E[\sum_{i=1}^{n} \sum_{j>i} X_{ij}] = \sum_{i=1}^{n} \sum_{j>i} E[X_{ij}] \tag{3.3}$$

de gelijkheid uit vergelijking 3.3 volgt rechtstreeks uit de lineariteit van de verwachting.

Nu laat p_{ij} de kans aanduiden dat $S_{(i)}$ en $S_{(j)}$ met elkaar vergeleken worden. Dan geldt wegens dat X_{ij} alleen de waardes 0 en 1 kan aannemen dat:

$$E[X_{ij}] = p_{ij} \times 1 + (1 - p_{ij}) \times 0 = p_{ij} \tag{3.4}$$

Uit vergelijking (3.4) blijkt duidelijk dat om de verwachte looptijd uit te rekenen we de eerst p_{ij} moeten uitrekenen. Om dit te doen zien we de uitvoering van RandQS als een binaire boom T. Elke node van de boom is gelabeld met één specifiek element uit de verzameling S. De wortel van de boom wordt gelabeld met het element y dat gekozen werd in de eerst stap van het algoritme. De linkerdeelboom bevat alle nodes waarvan de labels elementen zijn van S_1, terwijl de rechterdeelboom alle nodes bevat die als labels elementen uit de verzameling S_2 hebben. De structuur van de deelbomen wordt bepaald door de recursieve aanroepen van RandQS op zowel S_1 als op S_2. Het is duidelijk dat de wortel y vergeleken wordt

met alle elementen van zowel de verzameling S_1 als S_2, maar dat er geen vergelijkingen gebeuren tussen de linker- en de rechterdeelboom. Het is ook duidelijk dat er nooit een vergelijking zal plaatsvinden tussen elementen uit de linker- en de rechterdeelboom, want de recursieve aanroep beschouwt alleen de deelboom waarop hij wordt aangeroepen en niet de andere. Uit het voorgaande kunnen we dus besluiten dat $S_{(i)}$ en $S_{(j}$) vergeleken worden als en slechts als één van deze elementen een voorouder is van het andere element.

Het moge duidelijk zijn dat het in-orde doorlopen van de binaire boom, alle elementen van S in stijgende volgorde tegenkomt, in feite is het de exacte output van het algoritme. Voor de huidige analyse echter zijn we meer geïntereseerd in de niveau-orde doorkruising van de nodes. Het doorlopen van de nodes niveau per niveau en van links naar rechts op elk niveau levert namelijk een permutatie π op. Herinner dat het $i$de niveau van de boom de verzameling is van alle nodes op een afstand i van de wortel.

Nu kunnen we met behulp van twee observaties de kans p_{ij} uit gaan rekenen. Ten eerste is er alleen dan een vergelijkingen tussen $S_{(i)}$ en $S_{(j)}$ als en slechts als $S_{(i)}$ of $S_{(j)}$ eerder in de permutatie π voorkomt dan om het even ander element $S_{(l)}$ met $i < l < j$.

Bewijs. Laat $S_{(k)}$ het eerste voorkomen in de permutatie π van alle elementen met een rang ten minste i en ten hoogste j. Als k niet gelijk is aan ofwel i, ofwel j dan zit $S_{(i)}$ in de linkerdeelboom van $S_{(k)}$, terwijl $S_{(j)}$ in de rechterdeelboom zit. Ook kunnen $S_{(i)}$ en $S_{(j)}$ niet eerder in verschillende deelbomen beland zijn daar beide ofwel kleiner ofwel groter zijn dan elementen die niet in het interval $\{i, i+1, ..., j\}$ liggen. Dit impliceert dat er geen vergelijking plaatsvindt tussen $S_{(i)}$ en $S_{(j)}$. Omgekeerd als k wel gelijk is aan i of j dan is er een voorouder, afstammeling relatie tussen $S_{(i)}$ en $S_{(j)}$. Dit impliceert dan weer dat $S_{(i)}$ en $S_{(j)}$ met elkaar vergeleken worden. □

Ten tweede hebben alle elementen $S_{(i)}, S_{(i+1)}, ..., S_{(j)}$ evenveel kans om eerst gekozen te worden in het RandQS algoritme en daarmee eerst te komen in de permutatie π. Daardoor is de kans dat ofwel $S_{(i)}$, ofwel $S_{(j)}$ eerst gekozen wordt, gelijk aan $2/(i - j + 1)$. Er zijn namelijk $i - j + 1$ getallen waarvan 2 er het juiste resultaat geven. Dit is ook meteen de kans dat $S_{(i)}$ en $S_{(j)}$ met elkaar vergeleken worden.

Dus p_{ij} is gelijk aan $2/(i - j + 1)$. Uit vergelijkingen (3.3) en (3.4) volgt dat het verwachte aantal vergelijkingen tussen 2 elementen van de

verzameling S gelijk is aan:

$$\begin{aligned}\sum_{i=1}^{n}\sum_{j>i} p_{ij} &= \sum_{i=1}^{n}\sum_{j>i}\frac{2}{j-i+1} \\ &\leq \sum_{i=1}^{n}\sum_{j>i}^{n-i+1}\frac{2}{k} \\ &\leq 2\sum_{i=1}^{n}\sum_{k=1}^{n}\frac{1}{k}\end{aligned}$$

Uit het bovenstaande volgt dat het verwacht aantal vergelijkingen gelijk is aan $2nH_n$ waarbij H_n staat voor het $n$de harmonisch getal, gedefiniëerd door $H_n = \sum_{k=1}^{n}\frac{1}{k}$.

We kunnen nu besluiten dat het verwachte aantal vergelijkingen van dat RandQS uitvoert, gelijk is aan ten hoogste $2nH_n$. We weten ook uit sectie 2.3.1 dat $H_n \sim lnn + \Theta(1)$. dit maakt de verwachte looptijd van RandQS gelijk aan $O(n \log n)$

Noot 3.1.1. Uitvoering van het RandQS algoritme laat een random permutatie π van de verzameling S ontstaan. Nu zijn deze permutatie niet uniform verdeeld over alle mogelijke permutatie van de verzameling S. Er zijn zelfs permutatie s die niet kunnen voorkomen bij de uitvoering van het RandQS algoritme. Neem bijvoorbeeld de verzameling $S = \{1, 2, 3\}$ de permutatie $2, 3, 1$ is een geldige permutatie van de verzameling S. Echter deze permutatie kan niet gegenereerd worden met behulp van RandQS daar dan eerst het element 2 gekozen wordt als 'splitter'. Deze splitst de verzameling S op in 2 deel verzamelingen 1 en 3. de eerste is de linkerdeelboom en deze levert dus het 2de element uit de permutatie. Aangezien deze verzameling een singleton is is er slechts één keuze namelijk 1. het is onmogelijk om op de 2de plaats in de permutatie het getal 3 te krijgen. Als niet alle permutaties mogelijk zijn is er geen uniforme verdeling over alle mogelijke permutaties.

3.1.2 Een Tweede Algoritme

Het minimale snede probleem

Beschouw de volgende situatie: G is een samenhangende, ongerichte multigraaf met n knopen. Een snede in G is een verzameling bogen die als ze verwijderd worden G in 2 of meer delen splitst. Een minimale snede is een snede met het kleinste aantal verwijderde bogen. Dit is eigelijk een meer eenvoudige versie van het algemene probleem waar de bogen ook nog kosten hebben. Beschouw nu een eenvoudig algoritme.

Input: een ongerichte, samenhangende multigraaf G met n bogen

Output: een minimale snede van G

stap 1 kies op op een volkomen willekeurig, volgens een uniforme verdeling, een boog uit G, zodat elke boog uit G evenveel kans heeft om gekozen te worden.

stap 2 neem de 2 knopen op de eindpunten van deze boog samen. Alle bogen van elk van de 2 samengenomen knopen komen nu aan op de samengenomen knoop maar alle bogen van de samengenomenknoop naar zichzelf worden verwijderd.

stap 3 herhaal stappen één en twee totdat er nog maar 2 knopen overblijven. De bogen tussen deze knopen vormen een snede van G.

we noemen het samensmelten van knopen over een boog, het samentrekken van deze boog. Met elke samentrekking wordt het aantal knopen in de graaf G met één verminderd. We kunnen er dus zeker van zijn dat het algoritme ooit stopt en wel na $n-2$ herhalingen. Het is belangrijk om in te zien dat de operaties van stap één en twee een nieuwe graaf creëren met de eigenschap dat elke snede in deze nieuwe graaf ook een snede vertegenwoordigt in de oude graaf. Merk op dat alle bogen in de nieuwe graaf een equivalent hebben in de oude graaf. Dus als een verzameling bogen in de nieuwe graaf een snede is, dan kunnen we de equivalente bogen uit de oorspronkelijke graaf verwijderen. Is de oorspronkelijke graaf dan nog samenhangend? Het antwoord op deze is vraag is nee, want de oorspronkelijke graaf bevat alleen die bogen meer die lopen tussen twee knopen die samengesmolten zijn in de nieuwe graaf. Deze bogen zijn 'intern' in de losse groepjes die door de voorgestelde snede gevormd worden en dus zullen ze deze stukken van de graaf niet met elkaar verbinden waardoor de oorspronkelijke graaf na de snede niet meer samenhangend is. De snede is echter niet altijd minimaal. Na de volgende definities zullen we de kans gaan uitrekenen dat deze snede wel minimaal is.

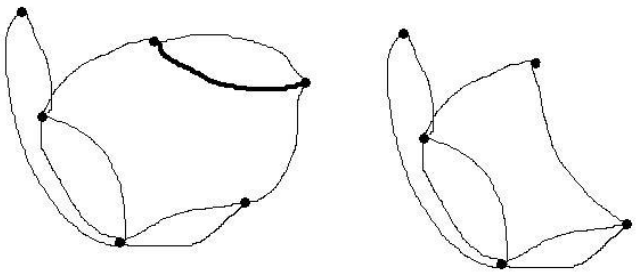
Enkele Definities

Definitie 3.1.1. Voor om het even welke knoop k in een multigraaf G the omgeving van k, genoteerd $\Gamma(k)$, is de verzameling van alle knopen van G die aanliggend zijn aan k.

Definitie 3.1.2. De omgeving van een verzameling knopen S van de graaf G, genoteerd $\Gamma(S)$, is de unie van de omgevingen van de elementen van S.

Definitie 3.1.3. De graad van een knoop k van de graaf G, genoteerd $d(k)$ is het aantal invallende bogen op v.

Opmerking. De cardinaliteit van $\Gamma(k)$ is gelijk aan $d(k)$ als er geen lussen of meervoudige bogen tussen k en om het even welke van zijn buren bestaan.



Figuur 3.1: Tekening van een multigraaf voor en na het samentrekken van de dikke boog.

De Kans op een Minimale Snede?

In deze sectie gaan we de kans uitrekenen dat het beschreven algoritme ook echt een minimale snede terug geeft. Laat om te beginnen de grootte van de minimale snede k zijn. de graaf G is een samenhangende, ongerichte multigraaf met n knopen. We beschouwen een bepaalde minimale snede C met k bogen. Nu heeft de graaf G ten minste $kn/2$ bogen. Als dit niet het geval zou zijn dan is er een knoop met een graad van minder dan k en dan vormen haar invallende bogen een minimale snede met een grootte van minder dan k.

Wat is nu de kans dat het algoritme tijdens haar uitvoering geen enkele boog uit C samentrekt. Als dat gebeurt blijven alle bogen uit C bewaard, en geeft het algoritme C terug als mogelijke oplossing.

Noteer ϵ_i als de gebeurtenis waarbij in de $i$de herhaling van het algoritme er geen boog gekozen wordt die een element is van C waarbij $1 \leq i \leq n-2$. De kans dat op de eerste stap er op een volkomen willekeurige wijze (en uniform verdeelt) een boog uit de minimale snede C gekozen wordt is hoogstens gelijk aan $k/(nk/2) = 2/n$. Er zijn namelijk ten minste $nk/2$ bogen in de graaf G en er zijn k mogelijke bogen die in de snede C zitten. Dus $Pr[\epsilon_1] \geq 1-2/n$. Voor de tweede stap, als we er van uit gaan dat we succes hebben in de eerste herhaling, zijn er minstens $k(n-1)/2$ bogen. Elke knoop moet nog steeds een graad hebben van minimaal k, maar het aantal knopen is op de tweede herhaling verminderd met één. De kans dat we in de tweede herhaling, gegeven dat we succes hadden in de eerste stap, een boog kiezen uit de minimale snede C is ten hoogste $k/\left(k\left(n-1\right)/2\right) = 2/(n-1)$. Dus $Pr[\epsilon_2|\epsilon_1] \geq 1-2/(n-1)$. In de $i$de herhaling zijn er minstens $n-i+1$ knopen over. De grootte van de minimale snede is nog steeds k. De graaf G bevat in de $i$de herhaling nog steeds minstens $k/\left(k\left(n-i+1\right)/2\right) = 2/(n-i+1)$

knopen. De kans dat er in de $i$de herhaling geen boog uit de snede C gekozen wordt, ervan uitgaande dat in alle voorgaande herhalingen geen boog gekozen is uit C wordt dan: Dus $Pr[\epsilon_i | \cap_{j=1}^{i-1} \epsilon_j] = 1 - 2/(n-i+1)$

De kans dat er gedurende het gehele proces geen boog uit de snede C geselecteerd wordt om te worden samengetrokken, wordt dan door de formule voor voorwaardelijke kans (zie sectie 1.2):

$$\begin{aligned} P[\cap_{i=1}^{n-2}\epsilon_i] &\geq \prod_{i=1}^{n-2}\left(1-\frac{2}{n-i+1}\right) \\ &= \left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)\left(1-\frac{2}{n-3}\right)...\left(1-\frac{2}{3}\right) \\ &= \frac{1}{3}\frac{2}{4}\frac{3}{5}\frac{4}{6}...\frac{n-4}{n-2}\frac{n-3}{n-1}\frac{n-2}{n} \\ &= \frac{2}{n(n-1)} \end{aligned}$$

De kans dat het algoritme een bepaalde minimale snede C vindt is groter dan $2/n^2$. Deze minimale snede kan eventueel uniek zijn voor de graaf G maar het kan ook zijn dat de graaf meerdere snedes van dezelfde grootte bevat. Dit algoritme kan echter ook van een niet minimale snede zeggen dat ze toch minimaal is. Het algoritme levert altijd een snede af. Deze snede is echter niet in alle gevallen minimaal. Dit algoritme is dus niet geheel foutloos zoals RandQS dat wel was. Het resultaat van RandQS was altijd betrouwbaar. Bij Dit algoritme kunnen we echter alleen spreken van de kans dat het antwoord juist is. Voor ëën enkele run is deze kans dus $n^2/2$.

Stel dat we het algoritme $n^2/2$ keer laten lopen (met onafhankelijke keuzes bij elke run) dan is de kans dat we geen minimale snede vinden wegens de onafhankelijkheid van kansen (zie sectie 1.3) hoogstens $(1-2/n^2)^{(}n^2/2) < 1/e$. Dit volgt uit de formule voor e zijnde $\lim_{i\to\infty}(1-i)^{1/i}$. Dit resultaat is al een heel stuk beter dan $n^2/2$. Merk op dat het algoritme nog steeds een tijdscomplexiteit heeft die polynomiaal is. We kunnen de kans op een goed resultaat nog meer verhogen door nog steeds meer runs uit te voeren. De looptijd zal echter bij elke verbetering toenemen. Wel blijft de tijdscomplexiteit polynomiaal, maar de benodigde tijd zal wel hoog oplopen.

Noot 3.1.2. Het algoritme werkt door de samentrekking ven een boog. Indien men gewoon twee willekeurige knopen zou samen nemen zonder naar de bogen te kijken dan zal het algoritme voor bepaalde input een marginaal kleine kans hebben om een minimale snede te vinden. Immers neem een graaf waarvan de knopen bestaan uit twee ongeveer even grote groepen. De knopen van één groep zijn allemaal onderling verbonden, maar hebben geen boog naar een knoop van de andere groep. Er is één uitzondering. Er bestaat net één boog die twee knopen verbind die elk in een andere groep zitten. Op deze manier is de gehele graaf samenhangend.

De kans dat het oorspronkelijk algoritme werkt is hoger dan in het algemene geval. De minimale snede is immers die ene boog tussen de twee verschillende groepen. De kans dat deze samengetrokken wordt is kleiner dan $n^2/2$ (een stuk kleiner zelfs maar één boog die niet samen getrokken mag worden en een hoop andere bogen die wel mogen samengetrokken waardoor de kans dat de 'verboden' boog het overleeft een stuk groter wordt.)

Nu als we gewoon willekeurig twee knopen zouden kiezen om samen te trekken in plaats van per boog te werken krijgen we in deze situatie een heel ander beeld. Om dan tot de minimale snede te komen mogen we bij elke samentrekking alleen knopen kiezen die tot dezelfde groep behoren. daar beide groepen even groot zijn zal de kans daarop gemiddeld rond de 50 percent liggen. De kans dat het fout gaat ligt dus ook rond de 50 percent voor elke stap en er zijn $n-1$ stappen. Dus de kans dat er geen twee knopen van verschillende groepen samen genomen worden, neemt snel toe naarmate er meer knopen in de graaf gestoken worden. Bij voldoende knopen wordt hij zelfs marginaal klein.

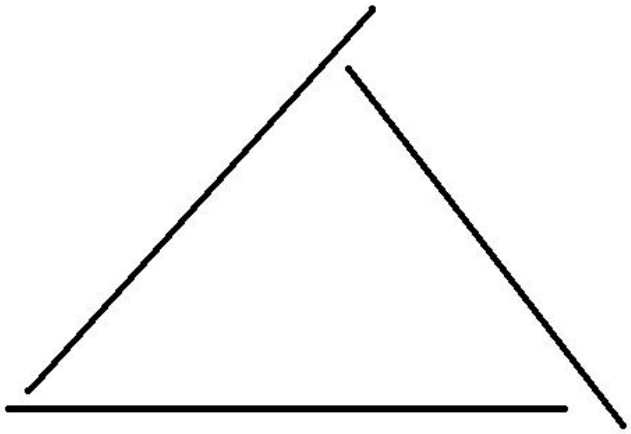
3.1.3 Een Derde Algoritme

Wat zijn Binair Planar Partitions?

Een binair planar partitie verdeeld een vlak in verschillende regio's. het is geen willekeurige verdeling van een vlak maar een zodanige verdeling dat van een gegeven verzameling lijnstukken in elke regio slechts één lijnstuk of deel van een lijnstuk ligt.

Een binair planar partition bestaat uit ten eerste een binaire boom. Echter is er additionele informatie beschikbaar. Elke interne node van de boom heeft twee kinderen. Er is dus geen interne node met slechts één kind. Ten tweede is elke node n geassocieerd met een regio $r(n)$ van het vlak dat we aan het verdelen zijn. Dit geld zowel voor de inwendige nodes als voor de bladeren. Ten derde is elke interne node geassocieerd met een lijn $l(n)$ die de regio $r(n)$ van deze node snijdt. Deze lijn kan een rechte, half rechte dan wel een lijnstuk zijn. Hij heeft in elk geval alleen invloed op de regio van zijn geasocieerde node en deze node zijn kinderen. Ten vierde wordt de regio $r(n)$ door de lijn van zijn node $l(n)$ in twee regio's $r_1(n)$ en $r_2(n)$ verdeeld. $r_1(n)$ en $r_2(n)$ zijn dan de regio's geasocieerd met de kinderen van de node n. De regio van de bladeren worden niet in tweeën verdeeld daar de bladeren niet geasocieerd zijn met een lijn. De regio waarmee de wortel is geasocieerd is het volledige vlak waarmee we begonnen zijn. Op deze manier is elke regio r uit de partitie gebonden door de partitie lijnen van het pad vanaf de wortel tot de node waarmee de regio r geasocieerd is.

Opmerking. Lijnstukken die op de grens tussen 2 regio's liggen, dat wil zeggen dat het lijnstuk op de lijn ligt die twee regioś scheidt, liggen in één van de twee regio's zoals het het beste uitkomt.



Figuur 3.2: voorbeeld waarbij er geen enkele binaire planaire partities bestaat die de lijnstukken niet splitst in meerdere delen.

Nu willen we aan de hand van een gegeven verzameling niet-snijdende lijnstukken een binaire planare partitie vinden zodat elke regio hoogstens 1 lijnstuk (of een deel van een lijnstuk) bevat. De eis dat de verschillende lijnstukken elkaar niet snijden is geen echt sterke eis. In het geval dat de verzameling toch lijnstukken bevat die elkaar snijden kan men deze lijnstukken opsplitsen zodat ze elkaar niet meer snijden.

Noot 3.1.3. Het is niet in alle gevallen mogelijk om een binaire planere partitie te maken dewelke de gegeven lijnstukken niet splitst in kleinere delen. Zie illustratie

Toepassingen van Binaire Planaire Partities

Binaire planaire partities worden vooral gebruikt bij computer graphics. Daar hebbeen binaire planaire twee toepassingen. Ze kunnen gebruikt worden voor het verwijderen van verborgen oppervlakten (hidden surfaces problem) en voor het representeren voor een polyhedraal object met behulp van constructive solid geometry.

Een natuurlijke opdeling voor een verzameling lijnstukken in een binaire planere partitie is de verzameling zogenaamde autopartitions dewelke alleen lijnen gebruikt die het verlengde zijn van de lijnstukken uit de gegeven verzameling lijnstukken. We beschouwen vanaf nu alleen deze partititions. Noteer $l(s_i)$ voor de lijn gevormd door het verlengen van het lijnstuk s_i.

Het Algoritme RandAuto

Input: een verzameling S met n niet snijdende lijnstukken $S = s_1, s_2, ..., s_n$

Output: een binaire autopartitie P_π van S

stap 1 Neem een permutatie p van $V = 1, 2, , n$ volledig willekeurig maar uniform van de $n!$ mogelijkheden

stap 2 Zolang een regio meer dan één lijnstuk of segment van een lijnstuk bevat, snijd deze regio met een lijn $l(s_i)$ waarbij i eerst komt in de gekozen permutatie P_π en $l(s_i)$ de regio daadwerkelijk verdeelt.

Theorema 3.1.1. *De verwachte grootte van de autopartitie geven door RandAuto is* $O(n \log n)$

Bewijs. Voor lijnstukken of segmenten van lijnstukken u en v definieer $index(u, v)$ zodat $index(u, v) = i$ als $l(u)$ (de lijn gevormd door het verlengen van het lijnstuk u) $i - 1$ andere lijnstukken of segmenten van lijnstukken snijdt voordat $l(u)$ het lijnstuk (of segment) v snijdt en $index(u, v) = \infty$ als uv niet snijdt. Het is mogelijk dat $index(u, v) = index(u, w)$ voor twee verschillende lijnstukken v en w daar het lijnstuk u langs beide zijden verlengt kan worden.

Noteer $u \dashv v$ het geval waar $l(u)v$ snijdt in de geconstruëerde partitie Laat $index(u, v) = i$ en $u_1, u_2, ..., u_{i-1}$ zijn de lijnstukken die $l(u)$ snijdt voordat v gesneden wordt. $u \dashv v$ gebeurt alleen als u voor alle $u_1, u_2, ..., u_{i-1}, v$ komt in de willekeurig gekozen permutatie π. De kans dat dit voorkomt is $1/(i+1)$ (Er zijn $i+1$ verschillende getallen, de kans dat één specifiek getal eerst komt staat gelijk met het kiezen van een getal uit een reeks de kans daarop is $i + 1$)

$C_{u,v}$ is een zogenaamde indicator variabele. Deze variabele is gelijk aan 1 als $u \dashv v$ en ze is gelijk aan 0 anders. Dan $E[C_{u,v}] = Pr[u \dashv v] \leq 1/(index(u, v) + 1)$ De grote van de binaire planaire partitie P_π is dan $n+$ het aantal intersecties De verwachte waarde voor de grote van de autopartitie is dan $n + E[\sum_u \sum vC_{u,v}]$ Lineariteit van de verwachting leidt dan tot:

$$n + \sum_u \sum_{v \neq u} Pr[u \dashv v] \leq n + \sum_u \sum_{v \neq u} 1/(index(u, v) + 1) \tag{3.5}$$

Voor elk lijnstuk u en elk eindig natuurlijk getal i zijn er hoogstens twee lijnstukken v en w zodat $index(u, v)$ en $index(u, w)$ gelijk is aan i. Dit omdat u aan twee zijden verlengd kan worden en komt aan éém van de twee kanten, elk lijnstuk hoogstens een keer tegen. Op deze manier ontstaat er in elk van de twee richtingen een totale ordening op de intersectie punten en de index van deze ordening stijgt monotoon. Dat impliceert:

$$\sum_{v \neq u} \frac{1}{(index(u, v) + 1} \leq \sum_{i=1}^{n-1} \frac{2}{i + 1} \tag{3.6}$$

Er zijn aan één zijde van u hoogstens $n-1$ lijnstukken, we gaan al deze lijnstukken af, vullen $index(u, v)$ in en dat voor beide zijden van u. Omdat we niet weten wat de juiste verdeling is tussen de lijnstukken aan de linkerkant van u, de rechterkant van u of de lijnstukken die niet door $l(u)$ gesneden worden, tellen we alsof $l(u)$ aan beide zijden het totaal aantal lijnstukken dat mogelijk is namelijk $n-1$ snijdt. Dit is natuurlijk een overschatting, maar is voldoende voor onze bedoelingen hier.

Als we nu het resultaat bekomen in (3.6) invullen in het eerder bekomen resultaat (3.5) voor de verwachte grootte van P_π geeft ons:

$$n + 2\sum_{u}\sum_{i=1}^{n-1}\frac{1}{i} \leq n + 2nH_n \tag{3.7}$$

nH_n is gelijk aan $O(n \log n)$ door gebruik temaken van sectie 2.3.1 waardoor het geheel ook $O(n \log n)$ is. Dus de verwachte grootte van een autopartitie gegenereerd door het algoritme RandAuto is $O(n \log n)$. □

Opmerking. Het resultaat kan als volgt geïnterpreteerd worden: daar de verwachte grootte van de binaire boom $O(n \log n)$ is voor om het even welke input, moet er voor elke input minstens één boom zijn die hoogstens $O(n \log n)$ groot is. Dit volgt uit de eigenschappen voor de verwachtingswaarde. Er is altijd minstens één mogelijke uitkomst die hoogstens gelijk is aan de verwachtingswaarde. Ook is er minstens één mogelijke uitkomst die ten minstens de verwachtingswaarde is, maar dat is hier niet zo van belang. Op deze manier hebben we met zekerheid kunnen vaststellen dat iets bestaat in plaats van met een bepaalde kans.

3.2 Classificatie van Stochastische Algoritmes

We hebben laten zien in de voorgaande stukjes dat niet alle gerandomizeerde algoritmes hetzelfde zijn. Het eerste algoritme dat we bekeken hebben, RandQS geeft altijd de juiste oplossing. De enige variatie zit hem in de looptijd van het algoritme. Het minimale snede algoritme daarentegen geeft niet altijd de juiste oplossing, maar heeft slechts een bepaalde kans dat het algoritme een juist antwoord geeft.

3.2.1 Las Vegas en Monte Carlo

Het eerste soort algoritme, dus de algoritmes zoals RandQS, noemen we een Las Vegas algoritme. Daar deze algoritmen variabel zijn in hun looptijd is het de verdeling van deze looptijd die we bestuderen. Het tweede soort algoritme, dat zijn de algoritmen zoals het minimale snede algoritme, zijn algoritmen die een oplossing geven die met een zekere kans correct is. Van

deze algoritmes kunnen we dan de kans afschatten dat hun gegeven antwoord fout is. Dit soort algoritmen noemen we Monte Carlo algoritmen.

Een nuttige eigenschap van een Monte Carlo algoritme is dat men de kans op een foute oplossing kan verkleinen door het algoritme meerdere keren uit te voeren met onafhankelijke keuzes voor de random beslissingen tijdens elke uitvoering. De kans op een foute oplossing kan arbitrair klein gemaakt worden door het meerdere keren te laten uitvoeren van het algoritme. Het is wel zo dat de looptijd in dat geval toeneemt. Ook zijn er algoritmen waarvan zowel de looptijd als de kans op een goede oplossing gerepresenteerd worden door een stochastische variabele. Deze algoritmen noemt men meestal ook Monte Carlo Algoritmen.

Beslissingsproblemen en Monte Carlo

Een beslissingsprobleem is een probleem dat slechts twee mogelijke antwoorden heeft: ja of nee. Voor een beslissingsprobleem kunnen we twee soorten Monte Carlo algoritmen onderscheiden. Een beslissingsprobleem met een algoritme dat met een niet nulle waarschijnlijkheid bij zowel een ja- of een nee-antwoord een fout kan maken noemen we een Monte Carlo algoritme met een tweezijdige fout. Als het algoritme slechts een fout maakt bij ofwel een ja-antwoord ofwel een nee-antwoord, spreken we van een eenzijdige fout.

Het is niet mogelijk te zeggen welk soort algoritme beter is Las Vegas of Monte Carlo. Dit hangt van de toepassing af. In sommige toepassingen is het catastrofaal om een fout te maken, bij andere is het van minder belang. Bij definitie is een Las Vegas algoritme een Monte Carlo algoritme met een kans op een fout van nul. Hier volgt een manier om een Monte Carlo algoritme om te zetten naar een Las Vegas algoritme.

3.2.2 Omzetten van een Monte Carlo Algoritme

Gegeven een Monte Carlo algoritme A voor een probleem Π met verwachte looptijd ten hoogste $T(n)$ op elke input van grootte n en een kans op een juiste oplossing $\gamma(n)$. Veronderstel ook dat we gegeven een oplossing voor Π we deze kunnen verifiëren in tijd $t(n)$. Maak een Las Vegas algoritme dat altijd de juiste oplossing geeft voor het probleem Π met een verwachte looptijd van ten hoogste $\frac{(T(n)+t(n))}{\gamma(n)}$.

De oplossing gaat als volgt: voer eerst algoritme A uit, controleer dan de oplossing gegeven door A. Indien deze oplossing correct is, geef dan deze oplossing terug als correct antwoord. Indien de oplossing fout is, herhaal dan de voorgaande stappen tot een juiste oplossing bekomen wordt. Dit zorgt ervoor dat we in elk geval het juiste resultaat krijgen.

Dit geeft de manier aan waarmee we een Monte Carlo algoritme kunnen omzetten naar een Las Vegas algoritme. Nu moeten we alleen nog aantonen dat de verwachte looptijd van ons geconstrueerd Las Vegas algoritme

ten hoogste $\frac{(T(n)+t(n))}{\gamma(n)}$ is. Merk op dat de looptijd een geometrische stochastische variabele is. Dit is in wezen hetzelfde als hoeveel keer moet ik gemiddeld een munt opgooien waarvan de kans op kop p is, tot ik voor de eerste keer kop krijg. De oplossing van dit probleem is $1/p$ dus voor ons Las Vegas algoritme verwachten we $1/\gamma(n)$ herhalingen, dus de running time die we verwachten is $(T(n) + t(n))/\gamma(n)$ daar elke herhaling $T(n) + t(n)$ tijd in beslag neemt.

3.2.3 Foutverkleining bij Monte Carlo Algoritmes

Een ander probleem is hoeveel onafhankelijke herhalingen nodig zijn bij het verkleinen van de fout bij een Monte Carlo algoritme. Zei $0 < \epsilon_2 < \epsilon_1 < 1$. Neem een Monte Carlo algoritme dat het juiste antwoord geeft op een gegeven probleem met een kans van ten minste $1 - \epsilon_1$ on geacht de input. Hoeveel onafhankelijke herhalingen volstaan om de kans op een correcte oplossing te verhogen tot minstens $1 - \epsilon_2$, ongeacht de input.

Elke herhaling van het algoritme heeft een kans van ϵ_1 op een foute oplossing. De kans dat een aantal onafhankelijke herhalingen fout zijn is dan ϵ_1^n, waarbij n staat voor het aantal onafhankelijke herhalingen. We moeten er voor zorgen dat de fout kleiner wordt dan ϵ_2 zodat we een algoritme krijgen dat correct is met een kans van $1 - \epsilon_2$. De vraag is nu hoe groot moet n minstens zijn zodat $\epsilon_1^n \leq \epsilon_2$?

$$\epsilon_1^n \leq \epsilon_2 \tag{3.8}$$

Neem van beide zijde de logaritme. Dan wordt de vergelijking:

$$\log \epsilon_1^n \leq \log \epsilon_2 \tag{3.9}$$

Breng de exponent n naar voor, volgens de rekenregels der logaritmen. Dan krijgen we:

$$n \log \epsilon_1 \leq \log \epsilon_2 \tag{3.10}$$

Deel beide zijden door $\log \epsilon_1$. Dit is een strikt negatief getal waardoor de ongelijkheid omkeerd. Dit leidt tot de volgende vergelijking:

$$n \geq \frac{\log \epsilon_2}{\log \epsilon_1} \tag{3.11}$$

Toepassen van rekenregels van logaritmen geeft dan:

$$n \geq \log_{\epsilon_1} \epsilon_2 \tag{3.12}$$

We kunnen dus besluiten dat we het algoritme minstens $\log_{\epsilon_1} \epsilon_2$ aantal keren moeten uitvoeren om de gewenste precisie te verkrijgen.

3.2.4 Een Derde Soort

Er is ook nog een derde soort random algoritme. Dit soort algoritme wordt niet gebruikt om problemen op te lossen maar om het bestaan van bepaalde eigenschappen van oplossingen aan te tonen. Een voorbeeld van een dergelijk algoritme is het algoritme RandAuto waarmee we hebben kunnen aantonen dat er altijd ten minste één autopartitie is met een grootte van $O(n \log n)$ bestaat. Van deze algoritmes bestuderen we niet de looptijd noch de kansverdeling van de oplossingen, maar we gebruiken ze samen met kanstheorie als bewijs.

3.2.5 Efficiëntie en Stochastische Algoritmes

Nog een laatste opmerking over efficiëntie. We noemen een Las Vegas algoritme efficiënt als het algoritme een verwachte looptijd heeft, op om het even welke input, die looptijd afgeschat kan worden door een veelterm functie van de grootte van de invoer. We zeggen dat een Monte Carlo algoritme efficiënt is als dat algoritme op om het even welke input een worst-case looptijd heeft die gebonden is door een veelterm functie van de input grootte. Het moge duidelijk wezen dat de derde soort van random algoritmes geen overwegingen wat betreft efficiëntie heeft.

3.3 Computatie Model

3.3.1 Turing Machine

Deterministische Turing Machine

Voor discussies over complexiteit maken we gebruik van het Turing machine model zoals gebruikelijk is in de theoretische informatica. we definiëren eerst een Turing machine, wat een abstract model is voor een algoritme.

Definitie 3.3.1. Een deterministische Turing machine is een quadruppel $M = (S, \Sigma, \delta, s)$. S in een eindige verzameling toestanden en s is een element van S en is de initiële toestand van de machine. De machine gebruikt een eindige verzameling symbolen voorgesteld door Σ. Deze verzameling bevat de speciale symbolen BLANK en FIRST. De functie δ is de transitie functie van de Turing machine, een functie van $S \times \Sigma$ naar $(S \cup \{HALT, YES, NO\}) \times S \times \{\leftarrow, \rightarrow, STAY\}$ de machine heeft drie stop toestanden HALT, YES, NO (dit zijn toestanden maar formeel niet in S)

De input van een Turing machine wordt voorgesteld alsof hij geschreven is op een tape. De Turing machine kan tenzij anders vermeld wordt, zowel lezen als schrijven op deze tape. We nemen aan dat HALT, YES, NO en de symbolen $\leftarrow, \rightarrow$, en STAY zich niet in de verzameling $S \cup \Sigma$ bevinden.De

machine begint in de begintoestand s met zijn leeskop op het eerste symbool van de input, dat wil zeggen aan het linkeruiteinde van de tape. Dat symbool is altijd FIRST. De rest van de tape is een oneindige opeenvolging van symbolen uit $\Sigma \setminus \{BLANK, FIRST\}$. De grens van de input wordt aangegeven door het eerste BLANK symbool op de tape. Aan de hand van deze symbolen en de transitie functie bepaald de Turing machine zijn acties, zijn programma als het ware. Op elke stap leest hij het symbool in waar zijn leeskop zich bevindt en aan de hand van de huidige toestand en het ingelezen symbool bepaald hij welk symbool op de huidige positie zal worden geschreven, de nieuwe toestand en een richting waarin de leeskop zich beweegt voor de volgende stap. De Turing machine heeft de optie om het BLANK symbool te overschrijven, hij kan met andere woorden de meer ruimte van de tape gebruiken dan dat de input groot is. Wel kan de leeskop van een Turing machine niet voorbij het symbool FIRST.

Als de machine stopt dan zijn er drie mogelijk heden. Hij stop in de NO toestand. In dat geval heeft de machine de input verworpen. Hij kan ook stoppen in de YES toestand. In dat geval wordt de input aanvaard. Het derde geval treed op als de Turing machine stopt in de HALT toestand. Deze toestand duidt erop dat de Turing machine klaar is en dat de output op de tape geschreven is. Deze toestand wordt gebruikt wanneer het antwoord niet booleaans is.

Probabilistische Turing Machine

Een probabilistische Turing machine is hetzelfde als een normale Turing machine, alleen kan de probabilistische Turing machine een eerlijke munt opgooien in een stap. Dit komt overeen met een stochastisch algoritme. Op elke input x, aanvaardt een probabilistische Turing machine x met een bepaalde kans en we bestuderen die kans.

3.3.2 RAM Model

Voor het bestuderen van algoritmes wordt overgeschakeld op het RAM model. Het RAM of Random Acces Machine model heeft een machine die beschikt over registers en hoofdgeheugen. Met behulp van deze verschillende geheugens kan een RAM volgende operaties uitvoeren: input-output operaties, transfers tussen hoofdgeheugen en registers, indirecte adressering, branching, en aritmische operaties. Elk register of geheugen lokatie kan één integer bevatten, waartoe de machine in één stap toegang heeft. De toegestane aritmische operatoren zijn $+, -, \times$ en $\div$. Ook kan een algoritme twee getallen met elkaar vergelijken en de positieve vierkantswortel van en getal berekenen.

Er worden twee soorten RAM modellen gedefiniëerd, gebaseerd op de tijd op de kosten voor het berekenen van de looptijd van een programma.

Unit-cost RAM

Ten eerste is er unit-cost RAM. Bij dit model kosten alle operaties kosten evenveel en kunnen ze uitgevoerd worden in exact één stap. Dit model wordt in het algemeen gezien als te krachtig. Dit komt omdat er geen algoritme bekent is om dit RAM model te simuleren in polynominale tijd op een Turing machine. Dit probleem treed op omdat dit RAM model in tegenstelling tot de minder flexibele Turing machine vermenigvuldigen van elke grootte kan uitvoeren in constante tijd, ook voor heel grote integers.Door een beperking in te voeren en alleen optellen en aftrekken toe te laten als aritmische operaties is het wel mogelijk om dit RAM model te simuleren op een Turing machine in polynomiale tijd.

Log-cost RAM

De tweede definitie van een RAM is de zogenaamde log-cost RAM. Dit model is realistischer daar elke instructie tijd kost evenredig met de logaritme van zijn operands. Op deze manier wordt het vermenigvuldigingsprobleem van de unit-cost RAM vermeden. De log-cost RAM is dan ook equivalent met een Turing machine zelfs als we de alle aritmische operatoren gebruiken.

3.4 Complexiteitsklassen

3.4.1 Wat zijn Complexiteitsklassen?

Wij zullen complexiteitsklassen gaan definiëren ervan uitgaande dat het onderlinge computatie model, zoals gebruikelijk is, dat van de Turing machine is. We kunnen evengoed het Log-cost RAM model of het unit-cost RAM met beperkingen op de aritmetische bewerkingen gebruiken om tot gelijkwaardige definities te komen voor dezelfde complexiteitsklassen.

Het is gebruikelijk in complexiteitstheorie om zich te concentreren op het beslissingprobleem (een probleem met een ja of nee antwoord) dat gewoonlijk wordt afgeleid van een moeilijk (ook wel "hard") optimalisatie probleem. Dit maakt de wiskunde eenvoudiger en eleganter zonder aan de kern van het probleem zelf te raken.

Elk beslissingsprobleem kan behandeld worden als een taalherkenningsprobleem. Behoort een string tot een bepaalde taal ja of nee? Neem een eindig alfabet Σ en zij Σ^* alle mogelijke strings over dit alfabet. Laat $|s|$ de lengte van een string s voorstellen. Een taal $L \subseteq \Sigma^*$ is een verzameling van strings over Σ. Het overeenstemmende taalherkenningsprobleem is het beslissen dat een gegeven string s een element is van een specifieke taal L. Een algoritme dat dit probleem oplost, aanvaardt indien $s \in L$ en wijst af als $s \notin L$

Een complexiteitsklasse is een verzameling talen waarvan hun herkenningsproblemen opgelost kunnen worden met vooraf bepaalde grenzen aan

hun computationele mogelijkheden. Meestal zijn we geïnteresseerd in efficiënte algoritme, waarbij we polynomiale tijd als efficiënt beschouwen. Een algoritme heeft een polynomiale looptijd als het stopt binnen $n^{O(1)}$ op elke invoer van lengte n.

3.4.2 De verschillende Complexiteitsklassen

We gaan nu enkele complexiteitsklassen definiëren. Eerst enkele gewone en daarnaast ook enkel specifiek bedoelt voor stochastische algoritemes.

Definitie 3.4.1. De klasse P (Polynomiale tijd) bevat alle talen L waarvoor een algoritme A bestaat, dat in polynomiale tijd loopt en voor elke input $x \in \Sigma^*$ geldt:

$x \in L \Rightarrow A(x)$ accepts

$x \notin L \Rightarrow A(x)$ rejects

Definitie 3.4.2. De klasse NP bevat alle talen L waarvoor een algoritme A bestaat, dat in polynominale tijd loopt en voor elke input $x \in \Sigma^*$ geldt:

$x \in L \Rightarrow \exists y \in \Sigma^*, A(x, y)$ accepts en $|y|$ is begrensd door een veelterm in $|x|$

$x \notin L \Rightarrow \forall y \in \Sigma^*, A(x, y)$ rejects

We kunnen deze definities als volgt opvatten: de klasse P bevat alle talen waarvan we het lidmaatschap van een string x makkelijk kunnen aantonen, de klasse NP daarentegen bevat alle talen waarvan we het bewijs van lidmaatschap makkelijk kunnen verifiëren. Een andere definitie maakt gebruik van een Turing machine. Alle talen die berekend kunnen worden met een deterministische Turing machine in polynomiale tijd zitten in de klasse P. Talen die berekend kunnen worden door een niet-deterministische Turing machine in polynomiale tijd behoren tot de klasse NP Het is duidelijk dat $P \subseteq NP$. Het is nog steeds een open vraag als $P = NP$. Over het algemeen wordt aangenomen dat $P = NP$ niet waar is.

Voor elke complexiteitsklasse C definiëren we de complementaire klasse co-C als de verzameling van talen wiens complement in C zit. We weten dat $P \subseteq NP \cap$ co-NP en dat $P =$ co-P.

We kunnen alle vermeldingen in de bovenstaande definities van polynomiale tijd vervangen door exponentiële tijd. Dit leidt tot de klassen EXP en $NEXP$ respectievelijk. Ook hier is het duidelijk dat $EXP \subseteq NEXP$. Maar ook hier weten we niet als $EXP = NEXP$. We weten wel dat als $P = NP$ dan ook $EXP = NEXP$.

We hebben tot hiertoe steeds de looptijd gebruikt om een onderscheidt te maken tussen de verschillende complexiteitsklassen. In plaats van tijd kunnen we ook gebruik maken van ruimte (=geheugen). De hoeveelheid

ruimte die een algoritme gebuikt komt bij een Turing machine overeen met de lengte van de tape die gebruikt wordt tijdens de uitvoering van het algoritme. Bij het RAM model gebruiken we simpelweg het aantal gebruikte woorden in het geheugen van de RAM. Dit leidt tot de klassen $PSPACE$ en $NPSPACE$. Deze klassen gedragen zich anders dan tijdscomplexiteitsklassen. Zo weten we dat hier $PSPACE = NPSPACE$ zeker waar is.

Definitie 3.4.3. Een polynomiale reductie van een taal $L_1 \subseteq \Sigma^*$ naar een taal $L_2 \subseteq \Sigma^*$ is een functie $f : \Sigma^* \to \Sigma^*$ zodat:

1 Er is een algoritme dat f berekent in polynimiale tijd.

2 $\forall x \in \Sigma^*, x \in L_1 \Leftrightarrow f(x) \in L_2$

Opmerking. Bovenstaande definitie houdt in dat als er een polynomiale reductie bestaat van L_1 naar L_2 en $L_2 \in P$ dan ook $L_1 \in P$.

Bewijs. We hebben een algoritme A dat in polynomiale tijd bepaald als een string $x \in L_2$ is of niet. Ook hebben we een algoritme R dat een string x afbeeldt op een string x' zodanig dat $x \in L_1 \Leftrightarrow x' \in L_2$. Het algoritme R loopt in polynomiale tijd. We kunnen nu een algoritme A' construeren dat in polynomiale tijd loopt en het herkenningsprobleem voor de taal L_1 oplost. Dat doen we als volgt. Voer eerst algoritme R uit. Op de uitvoer van R laten we algoritme A los. Allebei de algoritmes lopen in polynomiale tijd, dus samen is het nog steeds polynomiale tijd. En door de constructie geeft het algoritme een antwoord op het herkenningsprobleem van de taal L_1. □

Definitie 3.4.4. Een taal L is NP-hard als voor alle talen $L' \in NP$ er een polynomiale reductie bestaat die L' reduceert naar L.

Opmerking. Uit het bovenstaande volgt dus dat als we een NP-hard probleem kunnen oplossen in polynomiale tijd, we alle problemen in NP kunnen oplossen in polynomiale tijd en dus dat $NP = P$. Tot nu toe is er echter nog niemand in geslaagd om een NP-hard probleem op te lossen in polynomiale tijd.

Definitie 3.4.5. we noemen een taal L NP-compleet als en slechts als $L \in NP$ en L is NP-hard.

Nu we de standaard complexiteitsklassen gedefiniëerd hebben, kunnen we nu hetzelfde doen maar dan voor gerandomiseerde algoritmes.

Definitie 3.4.6. De klasse RP (randomized polynomiaal) bestaat uit alle talen L waarvoor een gerandomiseerd algoritme A bestaat met een worstcase polinomiale looptijd zodat voor elke string x uit Σ^* geldt:

$x \in L \Rightarrow Pr[A(x) \text{ accepts}] \geq \frac{1}{2}$

$$x \notin L \Rightarrow Pr[A(x) \text{ accepts}] = 0$$

Een RP-algoritme is een Monte Carlo algoritme dat alleen fouten maakt indien de string x een element is van L. Het is dus een algoritme met een zogenaamde one-sided error bij het rejecten. Soms wordt een string x verworpen terwijl deze string zic toch in de taal bevindt. Als deze fout zich aan de andere zijde voordoet hebben we een co-RP-algoritme.

Definitie 3.4.7. De klasse ZPP (zero error probalistic polynomial time) is de klasse van alle talen met een Las Vegas algoritme dat loopt in verwachte polynomiale tijd.

Opmerking. Uit deze en voorgaande definities volgt $ZPP = RP \cap \text{co-}RP$.

Bewijs. Zij $L \in RP \cap \text{co-}RP$, dan bestaat er een algoritme A_1 dat L beslist met een one-sided error waardoor af en toe geldige strings verworpen worden. Ook bestaat er een algoritme A_2 dat eveneens L beslist, maar af en toe verkeerde strings aanneemt. Beide algoritmes hebben een worst-case looptijd die polynomial is. Maak nu het algoritme A als volgt. Voer eerst A_1 uit op de string. Als A_1 aanvaardt, aanvaardt de invoer. Als A_1 verwerpt, voer dan A_2 uit op dezelfde input. Als A_2 verwerpt, verwerp dan de invoer. Als er nog geen resultaat is begin dan van voor af aan met het uitvoeren van A_1. Daar de kans op een fout $1/2$ is, en we blijven proberen tot we een juist antwoord krijgen is dit weer een geometrische kansverdeling. Het verwachte aantal keren dat we de lus moete doorlopen is dus twee keer. Vier keer iets polynomiaals is nog steeds polynomiaal. Dus $RP \cap \text{co-}RP \subseteq ZPP$.

Nu de andere kant. Zei $L \in ZPP$ een taal met een algoritme A. Dit algoritme heeft een verwachte looptijd die polynomiaal is. Dit betekent echter niet dat de worst-case ook polynomiaal is. Om dit aan te pakken, passen we A een beetje aan. We genereren een maximale looptijd voor A waarbij we A afbreken zodra A deze tijd gerund heeft. We nemen voor deze tijd de verwachte looptijd van het algoritme A op deze manier zal er in de meeste gevallen tijd genoeg zijn om A uit te rekenen. Nu voor het algoritme uit RP verwerpen we de invoer als A afgebroken wordt, terwijl we voor het co-RP algoritme dan aanvaarden. Daar minder dan de helft wegens de eigenschappen van de verwachtingswaarde, wordt afgebroken blijft de kans op een fout kleiner dan $1/2$ zodat $ZPP \subseteq RP \cap \text{co-}RP$.

Uit het bovenstaande volgt dan $ZPP = RP \cap \text{co-}RP$. □

Definitie 3.4.8. De klasse PP (Probabilistic Polynomial Time) bevat alle talen L waarvoor een stochastisch algoritme A bestaat met een worst-case polynomiale looptijd zodat voor elke string $x \in \Sigma^*$ geldt:

$$x \in L \Rightarrow Pr[A(x) \text{ accepts}] \geq \tfrac{1}{2}$$

$$x \notin L \Rightarrow Pr[A(x) \text{ accepts}] \leq \tfrac{1}{2}$$

Dit zijn de Monte Carlo algoritmes met two-sided errors. Nadeel van deze klasse is dat niet geweten is hoe ver de waardes afliggen van 1/2, hierdoor kunnen we de fout niet altijd arbitrair klein krijgen in polynomiale tijd.

Definitie 3.4.9. De klasse BPP (bounded-error Probabilistic Polynominal Time) bevat alle talen L waarvoor een stochastisch algoritme A bestaat met een worst-case polynomiale looptijd zodat voor elke string $x \in \Sigma^*$geldt:

$x \in L \Rightarrow Pr[A(x) \text{ accepts}] \geq \frac{3}{4}$

$x \notin L \Rightarrow Pr[A(x) \text{ accepts}] \leq \frac{1}{4}$

Deze klasse is handiger, want in dit geval kunnen we de fout wel arbitrair klein maken in polynomiale tijd.

Bewijs. Zij $L \in BPP$ een taal met een algoritme A. Maak een nieuw algoritme A'. Voer daartoe het algoritme A drie keer na elkaar uit. Als output geeft A' het resultaat van de meerderheid terug. Nu moeten we aantonen dat $A' \in BPP$ en dat $\frac{3}{4} \leq Pr[A(x) \text{ accepts}] < Pr[A'(x) \text{ accepts}]$ als $x \in L$, evenals $Pr[A'(x) \text{ accepts}] \leq Pr[A(x) \text{ accepts}] \leq \frac{1}{4}$ voor $x \notin L$. Als we de ongelijkheden aantonen volgt $A' \in BPP$ direct, daar driemaal iets polynomiaal nog teeds polynomiaal is.

We weten zeer weinig over de specifieke kansen van het algoritme A op elke input x. We breken het probleem in twee. Eerst het geval $x \in L$. Er zijn acht mogelijke uitkomsten en in vier ervan aanvaarden we (alledie aavaarden ze x ofwel is er één die x niet aanvaardt.) Noem $y = Pr[A(x) \text{ accepts}] \geq \frac{3}{4}$. De kans dat de input $x \in L$ aanvaardtwordt is dan $\big(y^3 + 3y^2(1-y)\big)$. Beschouwen we dat als functie dan is deze functie f strikt positief in het interval $[\frac{3}{4}, 1]$. Ook zijn afgeleide $f' = \big(-6y^2 + 6y\big)$ is strikt positief in dat interval. Daaruit kunnen we besluiten dat f een strikt stijgende functie is op $[\frac{3}{4}, 1]$. Het minimum bereikt deze functie dan ook op $\frac{3}{4}$. De waarde daar is 0.84375 hetgeen duidelijk hoger is $\frac{3}{4}$. Indien we een hogere waarde invullen zal omdat het het een strikt positieve functie is de uitkomst ook hoger liggen. Op deze manier hebben we de eerste ongelijkheid bewezen.

Nu het geval $x \notin L$. Hier is $y = Pr[A(x) \text{ accepts}] \leq \frac{1}{4}$. De kans op een verwerping is dan $1 - y$. Noem $1 - y$ dan z. Dit leidt tot de functie $\big(z^3 + 3z^2(1-z)\big)$. Als $y \in [0, \frac{1}{4}]$ dan $z \in [\frac{3}{4}, 1]$ gebruikmakend van het resultaat uit het vorige deel is de kans op verwerping dus groter dan $\frac{3}{4}$ en de kans op aanvaarding dus kleiner dan $\frac{1}{4}$. Ook stijgt de kans op verwerping als we beginnen met een hogere kans voor z waarmee overeenkomt dat de kans op aanvaarding daalt als de oorspronkelijke kans y daalt. Hiermee is dus de tweede ongelijkheid bewezen.

We kunnen dit resultaat opnieuw toepassen om de fout nog kleiner te krijgen terwijl de worst-case looptijd polynomiaal blijft. Op deze manier kunnen we de fout dus arbitrair klein maken. □

3.5 Oefeningen

Opgave 3.1. *Gegeven een munt waarvan de kans op KOP, p, ongekend is. Geef een manier waarop men met deze munt onbevoordeelde opgooiingen kan maken (dus zodat $Pr[KOP] = Pr[MUNT] = \frac{1}{2}$.) Zorg ervoor dat het verwachte aantal keren dat de oneerlijke munt opgegooid moet worden minder is dan $\frac{1}{p(1-p)}$. [9]*

Beschouw twee opeenvolgende worpen van de oneerlijke munt. Als het resultaat $KOP, MUNT$ is, noteer dan KOP voor de eerlijke munt. Als het resultaat $MUNT, KOP$ is, noteer dan $MUNT$. Indien de oneerlijke munt tweemaal na elkaar hetzelfde resultaat genereert, dan is er geen geldig resultaat voor de eerlijke munt.

De kans op KOP met de oneerlijk munt is p. Dit maakt de kans op $MUNT$ met de oneerlijke munt gelijk aan $1 - p$. De kans op KOP gevolgd door $MUNT$ is dus $Pr[KOP, MUNT] = p(1 - p)$. De kans op $MUNT$ gevolgd door KOP is eveneens $p(1 - p)$. Dus gegeven een geldige serie (dus niet tweemaal hetzelfde resultaat van de oneerlijk munt) is de kans op KOP of $MUNT$ van de eerlijke munt 50 %.

Nu blijft alleen de vraag over wat het verwachte aantal keren is dat men de oneerlijke munt moet opgooien. Elke serie van twee worpen heeft een universum $\Omega = \{(KOP, KOP), (KOP, MUNT), (MUNT, KOP), (MUNT, MUNT)\}$. We zijn geïntereerd in de kans een geldige worp voor de eerlijke munt, hetgeen overeenstemt met de deelverzameling $e = \{(KOP, MUNT), (MUNT, KOP)\}$. $Pr[e] = 2p(1-p)$. We kunnen dit nu zien als een geometrische verdeling zien. Dan weten we dat we het experiment, zijnde het tweemaal opgooien van de valse munt tot we een geldig resultaat bekomen vor de eerlijke munt, $\frac{1}{2p(1-p)}$ maal moeten uitvoeren. Daar voor elk experiment we de oneerlijke munt tweemaal opgooien moeten we deze munt gemiddeld $\frac{1}{p(1-p)}$ keren opgooien.

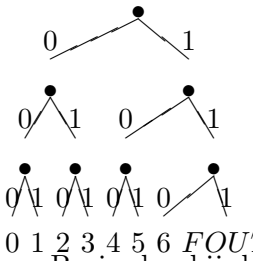
Opgave 3.2. *a) Gegeven een bron van onbevoordeelde random bits. Geef een manier om met deze random bits uniform verdeelde stalen te nemen uit de verzameling $S = \{0, ..., n-1\}$. Bereken ook het verwachte aantal van gebruikte random bits voor deze methode.*

b) Wat is het worst-case aantal random bits dat deze manier nodig heeft. Beschouw zowel het geval waar n een macht van twee is als waar n dat niet is.

c) Los a) en b) op, maar in plaats van gebruik te maken van onbevoordeelde random bits, gebruik uniform verdeelde stalen van de verzameling $\{0, ..., p-1\}$. Voor b) beschouw zowel het geval van een macht van p alsook het geval waar het geen macht is van p. [3]

a) Maak een beslissingsboom op de volgende manier: neem een gebalanceerde binaire boom binaire boom met het aantal bladeren gelijk aan de

kleinste macht van 2 die groter of gelijk is aan n. Noteer op de bladeren de getallen uit de verzameling S. Markeer de bladeren die overblijven met $FOUT$. Om het aantal gebruikte random bits in te perken kan men van elke knoop die 2 bladeren die gemarkeerd zijn met $FOUT$ deze bladeren verwijderen en van die knoop dan zelf een blad maken gemarkeerd met $FOUT$. Deze stap kan men herhalen tot er geen bladeren meer wegvallen. Label elke boog met 0 of 1, zodat er uit elke knop juist één boog vertrket gelabeld 0 en één met label 1. De boom ziet er dan als volgt uit voor $n = 7$:



Begin dan bij de wortel. Neem de eerste bit uit onze bron. Is dit een 0 volg dan de boog met het label 0, is het een 1 dan deze met het label 1. Die dit bij elke knoop tot een blad bereikt wordt. als dit blad een getal waarde heeft is dat de waarde van het sample. Bevat het blad echter de waarde $FOUT$ ga dan terug naar de wortel en ga dan van daaraf verder. Om de boom te door kruisen zijn er $\log n$ naar boven afgerond bits nodig. Ook kan het zijn dat het doorlopen enkele keren herhaald moet worden als er bladeren zijn met de code $FOUT$. Er zijn echter nooit minder dan de helft van de bladeren gelabeld met een getal. Dus in het slechtste geval verwachten we dat de boom tweemaal doorlopen moet worden (dit is weer een geometrische kansverdeling). Dus het verwachte aantal gebruikte random bits is $2 \log n$ naar boven afgerond.

b) In het worst-case scenario met n geen macht van twee zal er bij elke herhaling een blad bereikt worden met de code $FOUT$. In dat geval geval is het aantal gebruikte bits oneindig. Als n wel een macht van twee is zijn er geen bladeren met als label de code $FOUT$ in dat geval zullen er altijd $\log n$ bits gebruikt worden.

c) In dit geval zal het geen binaire boom zijn maar heeft elke interne knoop p uitgaande bogen, elk gelabeld met een waarde van $0, ..., p-1$ zodat elke waarde exact één keer voorkomt. Het aantal bladeren is dan gelijk aan de kleinste macht van p die groter is dan n. In dit geval verwacht men $2 \log_p n$ samples nodig te hebben om tot een resultaat te komen.

Het worst-case scenario is analoog aan dat van het binaire geval, maar hier kan het aantal gebruikte samples oneindig zijn als n geen macht van p is. Als n wel een macht van p is dan is het aantal gebruikte samples altijd $\log_p n$

Opgave 3.3. *Gegeven een bron van onbevoordeelde random bits, ontwerp een manier om random samples te genereren uit de verzameling* $2, 3, ..., 12$

die onstaat door het werpen van twee dobellstenen en het optellen van hun uitkomsten.[3]

Gebruik de voorgande oefening om voor elke dobbelsteen een boompje te maken. Voer beide bomen uit en tel hun reslutaten op. Men verwacht dat elke boom twee keer gerund moet worden voor een geldig antwoord, en het één doorlopen van 1 boom kost 3 random bits. Het verwachte gebruik is dus 12 random bits. Als alterntief kan men ook een boom nemen met 64 bladeren. Het getal 2 komt dan eenmaal voor, het getal 3 tweemal, net als de mogelijkheden zijn om deze getallen te genereeren met 2 dobbelstenen. Voor het doorlopen van deze grotere boom heeft men 6 random bits nodig, en men verwacht dat hij tweemaal doorlopen zal worden dus in het totaal verwachten we een gebruik van 12 bits. Merk op dat voor beide systemen het gebruik van het aantal bits evenhoog is.

Opgave 3.4. *a) Gegeven een onbevoordeelde bron van random bits, genereer een permutatie van grootte n. Efficiëntie hangt af van zowel tijd als het gebruikte aantal bits.*

b) Overweeg de volgende methode voor het genereren van een permutatie van grootte n. Kies n random waarden $X_1, ..., X_n$ onafhankelijk volgens de uniforme verdeling over het interval $[0, 1]$. De sortering van de verschillende X_i geeft een random permutatie van grootte n. Is dit waar? En hoe efficiënt is dize manier?

c) Overweeg de volgende 'luie' implementatie van het systeem vermeldt onder b). De binaire representatie van X_i is een reeks van onbevoordeelde random bits. Op elk moment van het sorteer algoritme hebben we slechts zoveel bits gebruikt als nodig om alle vergelijkingen tot op dat moment op te lossen. Wanneer X_i met X_j vergeleken wordt en de huidige prefix van hun binaire representaties zijn niet toerijkend om een beslissing te nemen, bereiden we deze prefixes uit tot er genoeg bits zijn zodat we een beslissing kunnen nemen. Wat is het verwachte aantal random bits gebruikt met deze implementatie?

a) We gaan een permutatie van lengte n maken op de volgende manier. Voor het eerste element van de permutatie kies volkomen willekeurig, maar met een uniforme verdeling een natuurlijk getal uit de verzameling $1, 2, ..., n$. Dit kan met de methode beschreven in oefening . Voor het tweede element kiezen we op dezelfde manier een natuurlijk getal maar nu uit de verzameling $1, 2, ..., n \setminus i$ waarbij i het element is gekozen op de eerste stap. Bij elke stap wordt de verzameling met keuzemogelijkheden kleiner. Na de $n$de stap is hij leeg en zijn we klaar. Op elke stap hebben we hoogstens $O(\log n)$ random bits nodig. er zijn n stappen, dus in het totaal hebben we hoogstens $O(n \log n)$ random bits nodig. In tijd is er niet meer nodig dan het telkens

beslissen welke boog men moet volgen. Er zijn dus hoogstens $O(n \log n)$ vergelijkingen. De tijdscomlexiteit is dus ook $O(n \log n)$

b) De sortering vormt een permutatie zolang alls X_i verschillend zijn van elkaar. De kans echter dat bij het kiezen van een eindig aantal getallen volgens de uniforme verdeling op een interval twwe keer hetzelfde getal voorkomt is oneinidg klein. Alle elementen hebben evenveel kans om op een bepaalde plaats uit te komen, dus de permutatie is volledig random.

Ervan uitgaande dat het kiezen van de waarden voor de verschillende X_i snel kunnen gebeuren in verhouding tot het sorteren is de tijdscomplexiteit in dit geval gelijk aan die van het sorteren. Gebruiken we RandQS voor het sorteren is de verwachte looptijd $O(n \log n)$

c) Als we een willekeurige splitter kiezen, dan is de eerste bit van de binaire representatie ofwel 1, ofwel 0. Als de splitter haar eerste bit 0 heeft hebben we om te vergelijken met alle elementen die als eerste bit een 1 hebben dus naar één bit nodig. Dit is ongeveer de helft van de totale groep. Had de splitter als eerste bit een 1 dan geldt hetzelfde maar hadden we maar één bit nodig voor de elementen beginnende met 0. De andere helft, die eenzelfde bit hadden als de splitter, hebben een tweede bit nodig, evenals de splitter. Ook hier worden de elementen in twee groepen verdeeld, zij die klaar zijn en zij die verder vergeleken moeten worden. daar de groep die verder vergeleken moet worden met elke nieuwe bit ongeveer halveert worden er $O(\log n)$ bits van de splitter gebruikt. Dit geldt ook voor elke splitter in een recursieve aanroep. Dus als r staat voor de $r$de recusrsieve aanroep heeft de splitter op de $r$de aanroep $O(\log \frac{n}{2^r})$ bits nodig. Alleen zijn er na de eerste r aanroepen al minstens r bits gebruikt. In het totaal heeft een splitter op de $r$de aanroep $O(r + \log \frac{n}{2^r}) = O(\log n)$ bits nodig. De elementen die geen splitter zijn moeten op de $r$de aanroep al minstens r bits en hoogstens evenveel bits als de splitter in hun prefix hebben. Daar het verwachte aantal aanroepen $O(\log n)$ is, hebben deze elementen $O(\log n)$ bits nodig. Er zijn n elementen. Het aantal gebruikte bits wordt dan $O(n \log n)$.

Opgave 3.5. *Beschouw het probleem waarbij we met een behulp van een onbevoordeelde bron van random bits willen kiezen uit de verzameling* $S = \{0, 1, ..., n-1\}$ *zodanig dat het* i*de element gekozen wordt met een kans* p_i*. Leg uit hoe deze keuzes te maken door slechts gebruik te maken van* $O(\log n)$ *bits ongeacht wat de waardes zijn vam* p_i*.*

Zij $q_j = \sum_{i=1}^{j} p_i$. Steek de q_j in een binaire endogene zoekboom, op deze wijze zit er geen enkele q_j in een blad. Elke q_j is geassocieerd met het $j$de element. Maak gebruik van de binaire representatie van de q_j. Als we nu een getal uit S willen kiezen, nemen we een hoeveelheid random bits om te kunnen beslissen als het groter of kleiner is dan de wortel. We gaan zo verder door de nodes tot we in een blad belanden. Als we niet genoeg bits hebben om te kunnen beslissen dan verlengen we de reeks random bits die we al hebben tot we wel kunnen beslissen. We verwachten dat we (zie vorige

opgave) $O(\log n)$ bits in de random reeks nodig hebben. Als de random reeks terecht komt in een blad dat het linkerkind is van een q_j dan wordt het $j$de element uit S gekozen, indien de random reeks terecht komt in een rechterkind dan wordt het $j+1$ste element gekozen. Daar het binaire fracties zijn zal er geen enkele randomreeks in het rechterkind van q_n terechtkomen.

Opgave 3.6. *bewijs de volgende inculisies:*

$$P \subseteq RP \subseteq NP \subseteq PSPACE \subseteq EXP \subseteq NEXP$$

$P \subseteq RP$ De definitie van P is gelijk aan de definitie van RP als we de kans op 1 zetten. De definitie van P is dus duidelijk stricter. Alle algoritmen die voldoen aan de definitie van P voldoen ook aan de definitie van RP. Alle elementen van P zitten dus ook in RP. Dus $P \subseteq RP$.

$RP \subseteq NP$ Met bepaalde willekeurige keuzes zal een algoritme uit RP besluiten dat een bepaalde string x deel is van een taal L als $x \in L$. Nu een algoritme in NP neemt twee strings x en y en een $x \in L$ als er een y bestaat zodat $A(x, y)$ aanvaardt. Neem nu de opeenvolgende willekeurige keuzes waarbij een RP algoritme x aanvaardt. Het NP algoritme is nu hetzelfde als het RP algorimte, maar in plaats van willekeurige keuzes, kiest het algoritme volgens de string y. en dus $RP \subseteq NP$

$NP \subseteq PSPACE$ De strings x en y nemen elk maar polinomiale ruimte in, in het geheugen. Alle bewerkingen die we in polinomiale tijd kunnen uitvoeren zullen, ook maar polinomiale ruimte innemen. Ergo een algoritme uit NP zal altijd beperkt kunnen worden tot het gebruik van polinomiale ruimte en dus als een algoritme uit $PSPACE$. Dit leidt tot $NP \subseteq PSPACE$.

$PSPACE \subseteq EXP$ Alle bewerkingen die uitgevoerd worden in polinomiale ruimte, zullen nooit meer danexponentiele tijd in beslag nemen. Bij gevolg is het algoritme uit $PSPACE$ ook een algoritme uit EXP, en dus $PSPACE \subseteq EXP$

$EXP \subseteq NEXP$ Dit volgt rechtstreeks uit de definitie. Het $NEXP$ algoritme is gelijk aan het EXP algoritme, waarbij y en willekeurige string is.

Opgave 3.7. *bewijs de volgende inculisies:*

$$RP \subseteq BBP \subseteq PP$$

$RP \subseteq BBP$ Daar een RP algoritme maar eenzijdige fouten maakt en op deze wijze altijd strings verwerpt als ze niet tot de te testen taal behoren, is al voldaan aan het tweede deel van de definitie van BPP. Het enige probleem is dat de kans uit het eerste deel van de definitie is dat de kans dat een RP algoritme zegt dat een string niet tot de taal behoort, terwijl dit wel het geval is groter dan voor een BPP algoritme. Dit lossen we als volgt op. Door het RP algoritme te herhalen maken we een nieuw algoritme. Dit

nieuwe algoritme voldoet wel aan de definitie van BPP. We aanvaarden als minstens een van beide runs aanvaard wordt. Anders verwerpen we. De kans dat het RP algoritme tweemaal achter elkaar een foutief verwerpt is ten hoogste $1/4$. De kans op een correcte aanvaring wordt dan $3/4$, hetgeen in de definitie van BPP staat. Bijgevolg $RP \subseteq BBP$.

$BBP \subseteq PP$ Uit de definfinites van BPP en PP blijkt duidelijk dat alle algoritmen die voldoen aan de definitie van BPP zeker voldoen aan de definitie van PP. Dus alle elementen van BPP zitten ook in PP. En dus $BBP \subseteq PP$.

Opgave 3.8. *Toon aan det $PP = co - PP$ en dat $BPP = co - BPP$*

Zij A een algoritme uit PP voor de taal L. Nu maken we een algoritme voor de taal $\bar{L}$, de complementaire taal van L namelijk A'. We voeren gewoon A uit op de invoer, maar als A aanvaardt reject A' en omgekeerd. Dan zal A' voor elk element $x \in \bar{L}$ aanvaarden met kans groter dan $\frac{1}{2}$ want als $x \in \bar{L}$ dan $x \notin L$ en dan zal A aanvaarden met kans kleiner dan $\frac{1}{2}$ waardoor A' zal aanvaarden met kans groter dan $\frac{1}{2}$. Omgekeerd, voor elke $x \notin \bar{L}$ geldt dat $x \in L$, waardoor A zal aanvaarden met een kans groter dan $\frac{1}{2}$. Dit heeft tot gevolg dat A' zal aanvaarden met een kans kleiner dan $\frac{1}{2}$. Zoals we kunnen zien voldoet het algoritme A' aan de definitie van PP. Dus elke taal uit PP haar compliment ligt ook in PP en dus $PP = co - PP$.

De tweede gelijkheid is vrijwel op dezelfde manier te bewijzen. Zij A een algoritme uit BPP voor de taal L. Nu maken we een algoritme voor de taal $\bar{L}$, de complementaire taal van L namelijk A'. We voeren gewoon A uit op de invoer, maar als A aanvaardt reject A' en omgekeerd. Dan zal A' voor elk element $x \in \bar{L}$ aanvaarden met kans groter dan $\frac{3}{4}$ want als $x \in \bar{L}$ dan $x \notin L$ en dan zal A aanvaarden met kans kleiner dan $\frac{1}{4}$ waardoor A' zal rejecten met kans kleiner dan $\frac{1}{4}$ en dus zal aanvaarden met kans groter dan $\frac{3}{4}$. Omgekeerd, voor elke $x \notin \bar{L}$ geldt dat $x \in L$, waardoor A zal aanvaarden met een kans groter dan $\frac{3}{4}$. Dit heeft tot gevolg dat A' zal aanvaarden met een kans kleiner dan $\frac{1}{4}$. Zoals we kunnen zien voldoet het algoritme A' aan de definitie van BPP. Dus elke taal uit BPP haar compliment ligt ook in BPP en dus geldt ook $BPP = co - BPP$.

Hoofdstuk 4

Speltheorie

4.1 Spelboom Evaluatie

4.1.1 Wat is een Spelboom?

Een spelboom is een boom met een wortel. Alle interne knopen van deze boom op even afstand worden met MIN en op oneven afstand met MAX gelabeld. Elk blad is geassocieerd met een reëel getal. Dit noemen we de waarde van dat blad. Nu kunnen we de spelboom gaan evalueren. Het evalueren van de spelboom gaat als volgt:

- Elk blad geeft zijn waarde terug.
- Elke MAX knoop geeft de hoogste waarde van zijn kinderen terug.
- Elke MIN knoop geeft de laagste waarde van zijn kinderen terug.

Spelbomen wordn veel gebruikt in artificile intelligentie, vooral bij het spelen van spellen. Het is mogelijk om alle kinderen van een knoop te associëren met een optie van een speler. De bladeren geven de waarde van het spel weer voor de spelers. Daartoe wordt gebruik gemaakt van een zogenaamde evaluatiefunctie de speltoestand op dat moment analyseert. Het simpelste geval van een dergelijke fuctie geeftt 1 voor winst van de eerste speler, 0 voor verlies van de eerste speler en 1/2 voor gelijkspel. Het is echter niet altijd mogelijk om alle mogelijke zetten te beschouwen tot op het einde van het spel. Dan kan er gebruik gemaakt worden van meer gesofisticeerdere evaluatiefuncties. Eén speler probeert de waarde te maximaliseren, de andere probeert juist de waarde te minimaliseren. Op deze manier kan elke speler zijn beste zet, gegeven de huidige situatie vinden.

4.1.2 Benodigde Tijd voor het Evalueren van een Spelboom

Om de looptijd van een spelboomevaluatie algoritme uit te rekenen gaan we het aantal bladeren tellen dat gelezen moet worden. We beperken ons

tot het speciale geval waarbij de bladeren 0 of 1 zijn. Dan kan elke MIN knoop gezien worden als een booleaanse AND en elke MAX-knoop als een booleaanse OR. Noteer $T_{d,k}$ staat voor een uniforme boom waar elke interne knoop en de wortel d kinderen heeft en elk blad zich op een afstand $2k$ van de wortel bevindt. Op deze manier passert elk pad van de wortel naar een blad k AND-knopen en k OR-knopen.

Een instantie van het evaluatie probleem bestaat uit een boom $T_{d,k}$ samen met een booleaanse waarde voor elk blad. Gegeven een algoritme kunnen we bestuderen hoeveel stappen er nodig zijn om de waarde van de wortel uit te rekenen. Een algoritme zegt op de eerste stap welk blad eruit gelezen zal worden. Voor de volgende bladeren beslist het algoritme aan de hand van de reeds gekozn bladeren en hun waardes. Een deterministisch algoritme gebruikt een deterministische functie om de opeenvolgende bladeren te bepalen. Bij een stochastisch algoritme kan de keuze willekeurig zijn.

We bestuderen nu een eenvoudig gerandomiseerd algoritme. Laat $d = 2$ voor de eenvoudigheid van de voorstelling. Een deterministisch algoritme kan een boom krijgen die alle bladeren moet opvragen. Dit zal niet zo zijn voor elke boom en voor verschillende deterministische aloritmes zullen het verschillende bomen zijn, maar er zullen altijd bomen zijn waarvan elk blad opgevraagd dient te worden. Nu gaan we echter een gerandomiseerd algoritme gebruiken.

Maak de volgende observatie: een AND-knoop met 2 bladeren evalueert tot 0 als en slechts als minstens één van de twee bladeren 0 is, dus onze tegenstander kan de 0 verstoppen in het laatste blad dat het algoritme zal evalueren. Verstoppen is echter niet mogelijk met een gerandomiseerde strategie. Het blad met de waarde 0 heeft één kans op twee om als eerst gekozen te worden om gelezen te worden. Het verwachte aantal stappen is dus 3/2, wat beter is dan de worst-case van een deterministisch algoritme. We kunnen dezelfde redenering maken voor een OR-knoop. In dat geval moeten we wel de verborgen 1 vinden en niet de 0 natuurlijk.

Het gerandomiseerde algoritme zal dus als volgt te werk gaan. Om een AND-knoop v te evalueren kiest het algoritme één van de twee kinderen en evalueert het deze knoop door zichzelf recursief op te roepen op de kinderen. Als het kind een 1 terug geeft wordt het andere kind ook geëvalueerd, als het een 0 is dan geeft het algoritme 0 terug als waarde voor de knoop v. Het algoritme werkt op dezelfde manier voor een OR-knoop, maar dan zal het algoritme het andere kind evalueren als het eerste kind een 0 teruggeeft en meteen 1 terug geven van zodra een kind 1 als waarde krijgt.

Bovenstaande geeft duidelijk aan dat men kan profiteren met een stochastisch algoritme als een AND-knoop geëvalueerd tot 0 zal worden of een OR-knoop tot 1. Nu is de vraag hoe men kan profiteren van een stochastisch algoritme als de AND-knoop 1, of de OR-knoop 0 teruggeeft. Als de kinderen bladeren zijn, moeten duidelijk beide bladeren gëvalueerd worden, maar bij een interne AND-knoop moeten beide kinderen 1 teruggeven. De kinderen

zijn in dat geval OR-Knopen. Nu is dat net het makkelijke geval voor OR-knopen, waar het in random volgorde evalueren een voordeel geeft. We kunnen dezelfde rederning opnieuw maken maar dan voor interne OR-knoppen.

We gaan nu aantonen door middel van inductie dat de verwachte looptijd voor de evaluatie van een $T_{2,k}$ spelboom hoogstens 3^k is. De basis ($k = 1$) volgt uit het bovenstaande. We hebben dan één AND-knoop namelijk de wortel. Deze heeft twee OR-knopen als kinderen. Als de wortel zou evalueren tot 1, moeten beide OR-knopen geëvalueerd worden. We weten dat het verwachte aantal bladeren dat elke OR-knoop moet lezen om zijn waarde te bepalen 3/2 is. Voor beide knopen komt het totaal dan uit op $2 \times 3/2 = 3 = 3^k$ want $k = 1$ in dit geval. Als de wortel zou evalueren tot 0, moet minstens een van de twee OR-knopen de waarde 0 krijgen. Om een OR-knoop te evalueren tot de waarde 0 moeten beide kinderen de waarde 0 hebben. Daar er minstens een kans is van 1/2 om als eerste een OR-knoop te kiezen die als waarde 0 heeft, is het verwachte aantal bladeren dat gelezen moet worden ook in dit geval 3.

Nu volgt de inductie stap. De kost voor het evalueren van $T_{2,k-1}$ is hoogstens 3^{k-1}. Neem een boom met als wortel een OR-knoop. Elk kind is de wortel van een instantie van $T_{2,k-1}$. De wortel kan alleen maar evalueren tot 1 als minstens één van de kinderen tot 1 evalueert.Daar er twee kinderen zijn, is de kans dat dit kind eerst gekozen wordt 1/2. De verwachte kost voor het evalueren van de boom is dan 3^{k-1} want we moeten slechts één instantie van $T_{2,k-1}$ evalueren. Als beide kinderen geëvalueerd moeten worden is de kost $2 \times 3^{k-1}$. De verwachte kost om de gehele boom te evalueren is dus $1/2 \times 3^{k-1} + 1/2 \times 2 \times 3^{k-1} = 3/2 \times 3^{k-1}$ Indien de wortel in het andere geval de waarde 0 zou krijgen, moeten beide subbomen getest worden. Dit geeft een kost van hoogstens $2 \times 3^{k-1}$.

Neem vervolgens een boom $T_{2,k}$. De wortel is nu een AND-knoop. Ook hier zijn twee mogelijkheden. Indien hij evalueert tot 1 dan moeten beide OR-knopen evalueren tot 1. Uit het voorgaande volgt dan dat de verwachte kost dan gelijk aan $2 \times 3/2 \times 3^{k-1} = 3 \times 3^{k-1} = 3^k$. In het andere geval als hij moest evalueren tot 0, dan moet minstens één van de subtrees de waarde 0 krijgen. De kans dat er eerst een subtree gekozen wordt die de waarde 0 heeft, is minstens 1/2. De verwachte kost in dit geval wordt dus $2 \times 3^{k-1} + 1/2 \times 3/2 \times 3^{k-1} \leq 3^k$. De eerste term is de verwachte kost van het evalueren van dde OR-knoop die de waarde 0 krijgt, de tweede term geeft aan dat er met kans 1/2 een extra kost is voor het evalueren van de andere OR-knoop met waarde 1.

We kunnen nu besluiten dat gegeven een willekeurige instantie van $T_{2,k}$ het verwachte aantal opgevraagde bladeren van het gerandomiseerd algoritme ten hoogste 3^k is. Nu daar het aantal bladeren $n = 4^k$ is, is de verwachte looptijd ten hoogste $n^{\log_4 3}$. Deze waarde schatten we af met $n^{0.793}$. Merk op dat deze waarde beter is dan elk deterministisch algoritme, want er zijn voor elk deterministisch algoritme instanties van $T_{2,k}$ waarbij

alle $n = 2^{2k}$ bladeren opgevraagd dienen te worden. Merk ook op dat dit een Las Vegas algoritme is, de uitkomst is namelijk altijd correct.

4.2 Het Minimax Principe

We hebben net een minimale looptijd van eenalgoritme voor het evalueren van een uniforme, binaire AND-OR boom berekend. Nu kunnen we ons afvragen als er andere algoritmes zijn die dezelfde boom sneller kunnen evalueren. Er is een techniek bekent die grbuikt kan worden voor precies dit doel: het minimax principe. In feite is het zelfs de enige bekende techniek. Ze komt voort uit de speltheorie, maar ze is in brede mate toepasbaar. Om tot het minimax principe te komen, bekijken we eerst wat algemene begrippen uit de speltheorie en passen daarna het minimax principe toe op het probleem van het evalueren van de AND-OR boom.

4.2.1 Speltheorie

Neem het spel blad-steen-schaar. We laten de verliezer een bepaalde som betalen aan de winnaar. Dit is een voorbeeld van een tweepersoons zero-sum spel. De naam zero-sum komt door het feit dat het totaal gewonnen bedrag van de twee spelers samen altijd nul is. dit komt doordat wat de ene wint de andere verliest.

De resultaten van dit spel kunnen samengevat worden in een matrix die de keuzes van de ene speler als zijn kolommen heeft en de keuzes van de andere speler worden door de rijen gerepresenteerd, met de payoff, de winst of het verlies als elementen in de matrix. In het algemeen kan dit resultaat uitgebreid worden zodat elk tweepersoons zero-sum spel kan weergegeven worden door een een $n \times m$ matrix. De kolommen vertegenwoordigen de mogelijke strategiën van de eerste speler, terwijl de strategiën van de tweede speler dan in de rijen staan. Het ijde element van de matrix is het bedrag dat de kolom speler betaalt aan de rij speler wanneer de rij speler strategie i en de kolom speler strategie j kiest. De rij speler wil nu de opbrengst maximailiseren, terwijl de kolom speler deze juist wil minimaliseren. Zij $\mathbf{M}$ de payoff matrix. Als de rijspeler een strategie i kiest dan is deze speler zeker van uitbetaling $\min_j M_{ij}$ ongeacht wat de kolom speler doet. een optimale strategie voor de rijspeler maximaliseert $\min_j M_{ij}$. Noteer $V_R = \max_i \min_j M_{ij}$ de ondergrens voor de uitbetaling van de rij speler wanneer deze een optimale strategie gebruikt. Voor de kolom speler geldt een gelijkaardig argument en zijn optimale strategie heeft een payoff van $V_K = \min_j \max_i M_{ij}$.

Er geldt $\max_i \min_j M_{ij} \leq \min_j \max_i M_{ij}$ voor willekeurige uitbetalingsmatrices $\mathbf{M}$.

Bewijs. Dit is makkelijk in te zien. Zij $\max_i \min_j M_{ij} = M_{ab}$ en $\min_j \max_i M_{ij} = M_{cd}$. Dan is M_{ab} het kleinste element van de $b$de kolom. Dus is $M_{cb} \geq M_{ab}$.

Nu geldt ook daar M_{cd} het maximum is van de $c$de rij $M_{cd}\ geq M_{cb}$ en dus geldt door transitiviteit $M_{cd} \geq M_{ab}$. Dit is duidelijk het gevraagde. □

In de meeste gevallen zal de ongelijktheid strikt zijn. Indien $\max_i \min_j \mathbf{M}_{ij} = \min_j \max_i \mathbf{M}_{ij}$ dan zeggen we dat ht spel een oplossing heeft. De waarde van het spel is dan $V = V_R = V_K$. De oplossing (ookwel het zadelpunt) van een dergelijk spel is de specifieke keuze voor de optimale strategiën die leiden tot deze uitbetaling. Het is mogelijk dat een speler meerdere optimale strategiën heeft.

Interessant wordt het indien een spel geen oplossing heeft. Er is dan geen duidelijke optimale strategie voor om het evenwelke speler. In dat geval kan elke aanwijzing over de strategie van de tegenstander de uitbetaling verbeteren. Tot nu toe hebben we gebruik gemaakt van deterministische strategiën, maar laten we eens kijken naar gerandomiseerde of gemixte strategiën. Een gemixte strategie is een kansverdeling over een verzameling mogelijke strategiën. Elke speler heeft een vector die een kansverdeling is over de rijen dan wel kolommen van de matrix $\mathbf{M}$ is. Zij $\mathbf{p} = (p_1, ..., p_n)$ de kansverdeling voor de rij speler, terwijl $\mathbf{q} = (q_1, ..., q_m)$ de kansverdeling is voor de kolom speler. De verwachte uitbetaling is dan een stochastische variabele gegeven door:

$$E[uitbetaling] = mathbfp^T mathbfMq = \sum_{i=1}^{n}\sum_{j=1}^{m} p(i)M(ij)q(j)$$

De ondergrens voor de verwachte uitbetaling voor de rij speler is zoals eerder V_R, de bovengrens voor de kolom speler is V_K.

$$V_R = \max_{\mathbf{p}} \min_{\mathbf{q}} \mathbf{p}^T\mathbf{M}\mathbf{q} V_K = \min_{\mathbf{q}} \max_{\mathbf{p}} \mathbf{p}^T\mathbf{M}\mathbf{q} \tag{4.1}$$

Waarbij min en max gaan over alle mogelijke kansverdelingen. Het minimax theorema van von Neumann impliceert dat dit spel altijd een oplossing heeft en dat de payoff van de rij speler gelijk is aan die van de kolomspeler

Theorema 4.2.1. *(Minimax Theorema van von Neumann) Voor elk tweepersoons zero-sum spel bepaald door de matrix* $\mathbf{M}$ *geldt:*

$$\max_{\mathbf{p}} \min_{\mathbf{q}} \mathbf{p}^T\mathbf{M}\mathbf{q} = \min_{\mathbf{q}} \max_{\mathbf{p}} \mathbf{p}^T\mathbf{M}\mathbf{q}$$

Het is duidelijk dat de hoogste verwachte uitbetaling die de rij spler kan garanderen gelijk is aan de laagste uitbetaling die de kolom speler kan garanderen door gebruik te maken van een gemixte strategie. Deze waarde noteren we met V. Een paar van gemixte strategieën $(\mathbf{p}, \mathbf{q})$, welke respectievelijk de linkerkant maximaliiseren en de rechterkant minimaliseren van de vergelijk uit theorema 4.2.1 noemen we een zadelpunt en de twee kansverdelingen noemen optimale gemixte strategiën.

Merk op dat eens wanneer $\mathbf{p}$ gekozen is, $\mathbf{p}^T\mathbf{Mq}$ een stelsel liniaire vergelijkingen in $\mathbf{q}$ wordt. Het minimaliseren is dan simpel. Voor $\mathbf{q}$ nemen we dan op alle plaatsen 0 behalve op de plaats met de mininale coëfficient q_j, daar zetten we dan een 1. Dus als de kolom speler de strategie van de andere speler kent dan wordt zijn strategie terug een pure strategie.Hetzelfdfe geldt ook voor de rij speler. We kunnen nu ook een simpelere versie van therorema opstellen. Zij $\mathbf{e}_k$ een eenheidsvector met een 1 op de $k$de plaats en voor de rest allemaal nullen.

Theorema 4.2.2. *(Theorema van Loomis) Voor elk tweepersoons zero-sum spel bepaald door de matrix* $\mathbf{M}$ *geldt:*

$$\max_{\mathbf{p}} \min_{j} \mathbf{p}^T\mathbf{Me}_j = \min_{\mathbf{q}} \max_{i} \mathbf{e}_i^T\mathbf{Mq}$$

4.2.2 Yao's Techniek

We gaan nu een techniek beschrijven die gebruikt kan worden om ondergrenzen voor de prestaties van gerandomiseerde algoritmes te bewijzen door toepassing van speltheorie. Laat daartoe de algoritme-ontwerper de kolomspeler zijn en de tegenstander, die de invoer kiest, de rijspeler. De kolommen van de matrix komen overeen met alle mogelijke algoritmes, de rijen met de verzameling van alle mogelijke invoeren van een vaste grootte. De payoff is dan een maat voor de performantie van een algoritme. De performantie kan uitgedrukt zijn in tijd, ruimte, kwaliteit van de oplossing of communicatie kost. In de rest van deze sectie gaan we uit van tijd, maar we kunnen om het even wat gebruiken. Het is duidelijk dat de ontwerper de payoff zo laag mogelijk wil hebben, terwijl de tegenstander er alles aan doet om de payoff te maximaliseren.

Beschouw het probleem waar er een eindig aantal verschillende invoeren van een vaste grootte zijn, als ook waarbij het aantal algoritmes, die een correcte oplossing geven. Een pure strategie voor de ontwerper is een deterministisch algoritme en de payoff is de worst-case looptijd van dat algoritme. Een gemixte strategie voor de ontwerper is een kansverdeling over alle mogelijk correcte deterministische algoritmes. Een gemixte strategie voor de tegenstander bestaat uit een kansverdeling over elke mogelijk input. De optimale mixed strategie voor de ontwerper is een optimaal Las Vegas algoritme.

We herformuleren nu de theoremas van von Neumann en Loomis als volgt voor algoritmen:

Lemma 4.2.3. *Neem een probleem* Π *met een eindige, vaste groep van invoer instanties van een vaste grootte* $\mathcal{I}$*, en een eindige verzameling deterministische algoritmes* (A)*. Voor elke input* $I \in \mathcal{I}$ *en een algoritme* $A \in \mathcal{A}$ *laat* $K(I, A)$ *de looptijd zijn van algoritme* A *op de invoer* I*. Voor kansverdelingen* $\mathbf{p}$ *over* $\mathcal{I}$ *en* $\mathbf{q}$ *over* $\mathcal{A}$ *laat* $I(\mathbf{p})$ *een random input zijn gekozen volgens*

$\mathbf{p}$ *en* $A(\mathbf{q})$ *een random algoritme gekozen volgens* $\mathbf{q}$.
Dan:

$$\max_{\mathbf{p}} \min_{\mathbf{q}} E[K(I_{\mathbf{p}}, A_{\mathbf{q}}] = \min_{\mathbf{q}} \max_{\mathbf{p}} E[k(I_{\mathbf{p}}, A_{\mathbf{q}}]$$

Ook:

$$\max_{\mathbf{p}} \min_{A \in \mathcal{A}} E[K(I_{\mathbf{p}}, A] = \min_{\mathbf{q}} \max_{I \in \mathcal{I}} E[K(I, A_{\mathbf{q}}]$$

Uit voorgaand lemma kunnen we nu makkelijk Yaos minimax principe afleiden.

Theorema 4.2.4. *(Yao's Minimax principe) Voor alle distrubuties* $\mathbf{p}$ *over* $\mathcal{I}$ *en* $\mathbf{q}$ *over* $\mathcal{A}$ *geldt:*

$$\min_{A \in \mathcal{A}} E[K(I_{\mathbf{p}}, A] = \max_{I \in \mathcal{I}} E[K(I, A_{\mathbf{q}}]$$

Met andere woorden, de verwachte looptijd voor een willekeurig gekozen input volgens een kansverdeling $\mathbf{p}$ van het optimale deterministische algoritme is een ondergrens van het optimale Las Vegas algoritme. We merken op dat het algoritme $\mathbf{p}$ kent!

4.2.3 Ondergrens voor het Evalueren van Spelbomen

We vervangen onze AND-OR boom door een equivalente boom, maar deze nieuwe boom maakt gebruikt van NOR-nodes. De evaluatie ven deze boom geeft hetzelfde resultaat als een AND-OR boom met dezelfde waarden op de bladeren. Dit is makelijk in te zien volgens de wetten van Demorgan:

$$(aNORb)NOR(cNORd) = NOT(aNORb)ANDNOT(cNORd) = (aORb)AND(cORd)$$

Het maakt niet uit als a, b, c, d zelf formules of literals zijn. Nu laat $p = \frac{(3-\sqrt{5})}{2}$zijn. Dit is dan de keuze die we maken voor $\mathbf{p}$. Elk blad van de boom heeft kans p dat het de waarde 1 krijgt. De kans dat beide inputs van een NOR node waarvan beide kinderen bladeren zijn en waardes hebben volgens onze gekozen $\mathbf{p}$, nul zijn, is gelijk aan:

$$\left(\frac{\sqrt{5}-1}{2}\right)^2 = (\frac{(3-\sqrt{5})}{2} = p$$

Nu weten we dus dat alle nodes in de boom evalueren tot een waarde van 1 hebben met een kans van p. Laat $W(h)$ het verwachte aantal bladeren zijn die geïnspecteerd worden door het algoritme voor het evalueren van een node op een afstand h van de bladeren. Daar we voor het evalueren van deze node zijn kinderen moeten evaluren en er een kans is van $1-p$ dat het eerste kind dat we kiezen te evaluëren nul is volgt duidelijk dat

$$W(h) = W(h-1) + (1-p) \times W(h-1) = (2-p) \times W(h-1)$$

De hoogte van de wortel van een boom is $\log n$. Daar we weten dat $n = 2^{2k}$ is $\log n = 2k$ volgt dat $W(h) = (2-p)^{2k}$. Dus $W(h) = n^{\log 2-p}$. Dit kunnen we afschatten met $W(h) = n^{0.694}$. In de eerste sectie van dit hoofdstuk hadden we een algoritme met een bovengrens van $W(h) = n^(0.793)$. Nu zou men dus in principe een algoritme kunnen vinden dat beter dan het algoritme dat beschreven is in de eerste sectie van dit hoofdstuk. Een diepere en moeilijkere analyse, met een kansverdeling die geen van de kinderen van om het even welke node tegelijkertijd de waarde 1 laat aannemen geeft een hogere ondergrens. Deze ondergrens is dan inderdaad $W(h) = n^(0.793)$. Deze analyse is echter niet uitvoerbaar zonder gebruik te maken van meer geavanceerde kanstheorie.

4.3 Oefeningen

Opgave 4.1. *Toon aan dat voor elk deterministisch algorimte voor het evalueren van spelbomen er een instantie van $T_{d,k}$ is waarbij dit algoritme alle bladeren moet bekijken.*

We weten in welke volgorde het algoritme de deelbomen gaat evalueren. We kiezen die instantie waarbij de eerste $d-1$ deelbomen altijd evalueren tot 0 als zij een kind zijn van een OR-node, en tot een 1 als zij een kind zijn van een AND-node. Op deze wijze zal het algoritme alle deelbomen van een node moeten evalueren om de node zelf te evalueren. Op deze wijze moeten alle deelbomen met als top een kind van de wortel geëvalueerd worden. Al hun kinderen moeten op hun beurt geëvalueerd worden en hun kinderen weer enzoverder. In het bijzonder zullen van een node die bladeren als kinderen heeft alle bladeren gelezen moeten worden. Op deze wijze worden alle bladeren dus gelezen.

Opgave 4.2. *Beschouw een boom waarvan elk blad zich evenver van de wortel bevindt, noem die afstand h. De wortel en alle inzendige nodes hebben drie kinderen. Elk blad is geassocieerd met een booleaanse waarde. Elke inwendige node heeft als waarde, de waarde van de meerderhijd van zijn kinderen. Het evaluatie probleem zoekt de waarde van de wortel. Op elke stap kan een algoritme de waarde van één blad opvragen. [6]*

> *a) Toon aan dat voor elk deterministisch algoritme er altijd een configuratie van bladeren gevonden kan worden, zodat het algoritme alle bladeren moet lezen.*
>
> *b) Beschouw het recursieve stochastische algoritme dat twee deelbomen van de wortel willekeurig kiest, en deze evalueert. Als de waardes niet overeenstemmen dan evalueert het algoritme ook de derde boom. Toon aan dat het verwachte aantal bladeren dat het algoritme leest ten hoogste $n^{0.9}$*

a) Daar het een determinsitsch algoritme is weet ik in welke volgorde de nodes geëvalueerd worden. Daardoor kan ik ervoor zorgen dat de eerste twee die geëvalueerd worden altijd verschillende waardes zullen hebben. Dus zal het algoritme altijd alle drie de deelbomen moeten evalueren. In het bizonder zullen ook alle nodes wiens kinderen bladeren zijn geëvalueerd worden. Om de waardes van deze nodes te bepalen, kan ik, op dezelfde wijze, ervoor zorgen dat ook alle bladeren gelezen moeten worden.

b) Het ergste geval is het geval waarbij twee kinderen een waarde hebben en het andere kind een andere. Voor de worst case gaan we ervan uit dat we dus altijd in dat geval zitten. Dan moeten we, voor het evalueren van een node altijd twee kinderen evalueren. Dan is er, doordat we in het geval zitten waar er slechts twee kinderen met dezelfde waarde zijn $\frac{2}{3}$ kans dat we ook de derde node moeten evalueren. Dus als $h = 1$ dan verwachten we dat we $\frac{8}{3}$ bladeren moeten lezen.

Voor $h > 1$ bewijzen we het met inductie. We weten nu dat het verwachte aantal bladeren dat we moeten lezen ten hoogste $\left(\frac{8}{3}\right)^{h-1}$ is. We gaan nu aantonen dat dit ook geldt voor h. Er moeten sowiso twee kinderen van de wortel geëvalueerd worden. Dat levert een bijdrage van $2 \times \left(\frac{8}{3}\right)^{h-1}$. Ook is er $\frac{2}{3}$ kans dat we ook de derde node moeten evalueren. Dat geeft ook nog eens $\frac{2}{3} \times \left(\frac{8}{3}\right)^{h-1}$ in het totaal geeft het dan:

$$\frac{8}{3} \times \left(\frac{8}{3}\right)^{h-1} = \left(\frac{8}{3}\right)^{h}$$

Daar $n = 3^h$ is het verwachte aantal bladeren dat we moeten lezen ten hoogste $n^{\log_3 \frac{8}{3}}$ is. Nu $\log_3 \frac{8}{3} = 0.892789 < 0.9$, waaruit de opgave volgt.

Opgave 4.3. *Bepaal de waarde V_R van de volgende 2×2 spelmatrix M en geef optimale gemixte strategieën voor beide spelers.*

$$\begin{bmatrix} 5 & 6 \\ 7 & 4 \end{bmatrix}$$

$V_R = \max_i \min_j M_{ij}$ en dus $V_R = \max\{5, 4\}$. Waaruit duidelijk is dat $V_R = 5$. Voor de optimale gemixte strategieën merken we op dat de ideale keuze voor de kolom speler op de ene diagonaal liggen en de ideale keuze van de rij speler op de andere diagonaal. De beste keuze voor beide spelers is gewoon met kans 1 op 2 een kolom of rij te kiezen. Merk op dat dit alleen geldt zolang de spelers elkaars strategie niet kennen.

Opgave 4.4. *Gebruik Yao's minimax principe om een ondergrens voor alle Las Vegas algoritmes te bepalen voor het sorteren van n getallen.*

Met een uniforme verdeling van de getallen heeft quicksort een kans van $\frac{1}{2}$ dat het getal dat gekozen wordt als splitter een goede keuze is. (elke

vast gekozen grens is in principe goed we kiezen hier arbitrair voor $\frac{1}{2}$) Het algoritme moet dan $O(n \log n)$ vergelijkingen uitvoeren wat ook de keuze is van de kans verdeling. We kiezen de splitter simpel weg als het element in die positie dat de grootste kans heeft om een rang te hebben gelijk aan de halve grote van de verzameling. Door de random verdeling van de getallen zal deze keuze meestal goed zijn. Waardoor we ook nooit meer dan $O(n \log n)$ vergelijkingen moeten uitvoeren. Dit geeft met behulp van Yao's minimax principe een ondergrens voor eht sorteren van een verzameling n getallen.

Hoofdstuk 5

Data Structuren

5.1 Het Fundamentele Data-Structuur Probleem

Eén van de eerste problemen die men op informatica vlak tegenkomt is het efficiënt opslaan van data. We wilen onze data op een zodanige manier opslaan dat we op deze data bepaalde voorgedefiniëerde opdrachten kunnen uitvoeren. We zullen nu aannemen dat onze data bestaat uit een collectie van verzamelingen. Elk item i uit onze data collectie is een record dat geïdentificeerd wordt door zijn key $k(i)$ en dat alle k uit een totaal geordend universum komen. Ook nemen we aan dat alle keys verschillend zijn. Verder hechten we geen wezelijk belang aan de werkelijke data, we houden alleen rekening met de key. In het algemene geval willen we de volgende operaties ondersteunen:

- $MAKESET(S)$: creëer een nieuwe en lege verzameling S.
- $INSERT(i, S)$: voeg een element i aan de verzameling S toe.
- $DELETE(k, S)$: verwijder het element beschreven door k uit de verzameling S.
- $FIND(k, S)$: geef het element beschreven door k uit de verzameling S terug.
- $JOIN(S_1, i, S_2)$: vervang de verzamelingen S_1 en S_2 door een nieuwe verzameling $S = S_1 \cup \{i\} \cup S_2$ met voor alle items $j \in S_1, k(j) < k(i)$ en voor alle items $j \in S_2, k(j) > k(i)$
- $PASTE(S_1, S_2)$: vervang de verzamelingen S_1 en S_2 door een nieuwe verzameling $S = S_1 \cup S_2$ met voor alle items $i \in S_1$ en voor alle items $j \in S_2, k(i) < k(j)$
- $SPLIT(k, S)$:vervang de verzameling S door een nieuwe verzameling S_1 en S_2 met $S_1 = \{j \in S \mid k(j) < k\}$ en $S_2 = \{j \in S \mid k(j) > k\}$

5.2 Algemene Oplossingen

5.2.1 Binaire Zoekbomen

Wat zijn Binaire Zoekbomen?

Een standaard oplossing is het gebruik van binaire zoekbomen om de verzameling S te representeren. Een binaire zoekboom is ten eerste een binaire boom. Dit wil zeggen dat alle nodes ten hoogste twee kinderen hebben. De makkelijkste manier van implementeren geeft elke node drie pointers. Eén naar de oudernode, één naar het linkerkind en één naar het rechterkind. De pointer naar de oudernode is alleen NIL als en slechts als de node de wortel van de boom is. De node is een blad als en slechts als beide kindpointers NIL zijn. Het is ook mogelijk om een binaire boom te implementeren met slechts twee pointers per node. Dan moeten we wel een onderscheid maken tussen de node van het linkerkined en die van het rechterkind. Ook moeten er soms twee pointers gevolgd worden om bij de node te raken die we willen hebben. Indien we twee pointers per node willen heeft elke node een pointer naar zijn linkerkind. De linkerkind node heeft een pointer naar zijn (rechter)broer, maar niet naar de oudernode. De rechternode heeft een pointer naar de oudernode. De oudernode kan aan zijn rechterkind door het volgen van twee pointers: via linkerkind pointer en dan de rechterbroerpointer. De linkernode kan aan zijn ouder door eerst naar zijn rechterbroer te gaan en daar de oudernode op te vragen.

We hebben nu een binaire boom implementatie. We hebben echter een binaire zoekboom nodig. Om van de verzameling S een representatie in de vorm van een binaire zoekboom te maken moeten we de elementen van S op een speciale manier aan de nodes van zoekboom koppelen. We doen dit op de volgende manier: van elke node die een element van S bevat, moet de linkerdeelboom alleen waardes bevaten die kleiner zijn dan de waarde uit de node, terwijl de rechterdeelboom alleen waardes bevat die groter zijn.

Om de beschrijving te vereenvoudigen gaan we ervan uit dat onze binaire zoekbomen endogeen zijn. Dit wil zeggen dat er alleen waardes gekoppeld worden aan interne nodes, niet aan bladeren. Alle bladeren zijn dus leeg. Ook heeft elke interne node exact twee kinderen, zodat de boom voledig is. Zoals gezegd is dit om de beschrijving te vereenvoudigen, het is niet strikt noodzakelijk.

Implementeatie van de Operaties

We kunnen nu de verschillende operaties als volgt ondersteunen. Voor $MAKESET(S)$ maken we simpelweg een lege boom aan. De operatie $FIND(k, S)$ kunnen we uitvoeren door binair te zoeken in de boom, als het gezochte element groter is dan de waarde van de huidige node zoek recursief verder in de rechterdeelboom, als de waarde kleiner ga dan verder

met de linkerdeelboom. Voor de operatie $INSERT(k, S)$ voeren we eerst $FIND(k, S)$ uit. Als k niet gevonden wordt, krijgen we een leeg blad terug. Op deze plaats kunnen we dan k toevoegen (samen met twee nieuwe lege bladeren voor de node die k bevat.) De operatie $DELETE(k, S)$ is iets moeilijker. Als de node die k bevat, minstens één kind heeft dat een blad is, dan kan $DELETE(k, S)$ simpel uitgevoerd worden door de ouder van de node die k bevat te linken met het kind dat geen blad is (indien dit bestaat.)

Als de node die k bevat, geen kinderen heeft die een blad zijn, dan zoeken we in de linkerdeelboom naar de voorganger van k in de verzameling S. De voorganger van k is geen voorouder van de node die k bevat want dan zou het linkerkind van k een blad zijn. Nu het rechterkind van de node van de voorganger van k is een blad, dus kunnen we de node die deze voorganger bevat verwijderen. We vervangen dan de node die k bevat door die met deze voorganger. de kinderen van k hangen we aan de node met de voorganger van k.

De operatie $JOIN(S_1, k, S_2)$ is makkelijk te implementeren. Maak een nieuwe node aan voor k en hang S_1 en S_2 eraan als respectievelijk de linker- en de rechterdeelboom. De operatie $PASTE(S_1, S_2)$ is ook niet moeilijk. Zoek en verwijder de grootste k uit S_1. Dit kan door het uitvoeren $FIND(\infty, S_1)$ dan de gevonden k verwijderen uit S_1. Voer dan $JOIN(S_1, k, S_2)$ uit om het gezochte resultaat te bekomen.

De operatie $SPLIT(k, S)$ is alleen simpel uit te voeren indien k de ortel is van de boom voor S. Dan verwijderen we k en S_1 is dan de linkerdeelboom, S_2 de rechter. De vraag is nu kunnen we onze boom zo transformeren zodat de node van k de wortel wordt en wel op een zodanige wijze dat de boom een binaire zoekboom blijft? Het antwoord daarop is ja. De operatie wordt het roteren van nodes genoemd. Met deze operatie kan men bepaalde nodes dichter bij de wortel van de boom brengen en tegelijkertijd andere nodes er verder van verwijderen. Door het herhaaldelijk uitvoeren van deze operatie kan men dus elke willekeurige node tot wortel maken dus ook de node k. Het roteren gaat als volgt te werk: wissel een ouder node met één van zijn kinderen (of een kindnode met zijn ouder.)De deelboom van de ouder die niet het gebruikte kind als wortel heeft blijft bij deze node. Dus als een node geroteerd wordt met zijn linkerkind blijft zijn rechterdeelboom hetzelfde. De linkerdeelboom wordt in datgeval de rechterdeelboom van de vroegere kindnode. De kindnode behoudt in dat geval zijn linkerdeelboom en als rechterkind krijgt het zijn vroegere oudernode. Als er geroteerd wordt met het rechterkind dan is alles hetzelfde alleen zijn links en rechts dan omgewisseld. Op deze wijze blijft de boom een binaire zoekboem. Zodra de node die k bevat de wortel is kunnen we weer op eenvoudige wijze $SPLIT(k, S)$ uitvoeren.

Tijdscomplexiteiten

Alle operatie zijn gebonden door de diepte van de boom. Sommige operaties zoals $JOIN$ kunnen ook in constante tijd. Nu de gemiddelde diepte van een binaire zoekboom is $\log n$, maar het is niet moeilijk om een opeenvolging van $INSERT$ operaties voor te stellen die ervoor zorgen dat de boom een diepte heeft van n. Voeg alle waardes van de verzameling S simpel weg toe in stijgende volgorde en dit is al het geval. Er zijn enkele manieren bedacht om ervoor te zorgen dat de zoekboom gebalanceerd blijft. Dit wordt meestal gedaan met de eerder beschreven rotaties. Tijdens elke update worden er ook rotaties uit gevoerd zodat de diepte van de boom $O(\log n)$ blijft. Op deze wijze kunnen we dus de operaties begrenzen door $O(\log n)$.

5.2.2 Random Treaps

Wat zijn Treaps?

Er zijn ook andere sorteringen mogelijk in een boom, buiten de zoekboom sortering. Als we ervoor zorgen dat de de waarde van de oudernode hoger is dan die van de kindnodes dan zeggen we dat de boom in heap orde is. Wanneer we nu aan elke node een koppel toekennen en we sorteren de eerste waarde van dit koppel zodat zij een binaire zoekboom vormen, terwijl we er ook voor zorgen dat de tweede waardes van de koppels op heap orde zijn, spreken we van een treap. Wanneer we de koppels aan de boom toevoegen zodat we de zoekvolgorde bewaren, zal het zijn dat we de heap volgorde zullen doorbreken. Door middel van rotaties kunnen we dan door het omhoog roteren van de node de heap volgorde herstellen en tegelijkertijd de zoekvolgorde bewaren.

Gegeven een verzameling koppels, dan is de bijbehorende treap uniek en niet afhankelijk van de volgorde waarin de koppels aan de treap zijn toegevoegd.

Theorema 5.2.1. *Zij $S = \{(k_1, p_1), ..., (k_n, p_n)\}$ een verzameling van key-prioriteit koppels zodanig dat elke key en elke prioriteit verschillend zijn. Dan bestaat er een unieke treap $T(S)$ voor deze verzameling S.*

Bewijs. Dit is een constructief bewijs en de constructie is recursief. Het theorema klopt duidelijk als er minder dan twee koppels zijn. Nu als er twee of meer koppels zijn, dan is er één koppel met de hoogste prioriteit. Stel dat koppel voor als (k_i, p_i). Een treap voor de verzameling S kan un geconstrueerd worden door (k_i, p_i) te gebruiken als wortel. Het is zelfs zo dat(k_i, p_i) de wortel van de treep moet zijn willen we er voor zorgen dat de heapvolgorde intact blijft. Nu kunnen we voor alle koppels waarvan de key kleiner is dan k_i een treap maken en deze gebruiken als de linkerdeelboom van de wortel. Voor alle elementen van S waarvan de key groter is dan k_i kunnen we hetzelfde doen maar dan voor de rechterdeelboom. Op deze wijze

kunnen we met behulp van recursie de voledige treap opstellen, met telkens slechts één mogelijke keuze voor de wortel van de nieuwe deelboom. Daar er slechts één keuze mogelijk is moet de opgebouwde treap dus uniek zijn voor de gegeven verzameling S. □

De vorm van een treap hangt dus af van de relatieve prioriteiten die bij de keys horen. Elke mogelijke vorm kan verkregen worden door een juiste keuze te maken van de prioriteiten bij de keys.

Implementatie

$MAKESET(S)$ en $FIND(k, S)$ implementeren we zoals in het geval van binaire zoekbomen. $INSERT(k, S)$ beginnen we op dezelfde manier als eerst. We voeren $FIND(k, S)$ uit en voegen de node voor k toe op het blad waar de zoekopdracht faalt. We kennen aan de node voor k ook een willekeurige prioriteit toe. Nu moeten we er echter ook voor zorgen dat de heap orde op de prioriteiten gehandhaafd blijft. In het geval dat de prioriteit van de net toegevoegde node hoger is dan net toegevoegde node, dan roteren we deze twee nodes met elkaar. We herhalen dit zolang de prioriteit van de toegevoegde node hoger is dan zijn oudernode. Op deze wijze blijft de treap een treap, want rotaties behouden de zoekvolgorde en we herstellen de heapvolgorde.

De implementatie van $DELETE(k, S)$ is net omgekeerd aan $INSERT(k, S)$. Ook hier voeren we eerst $FIND(k, S)$. Als de gevonden node twee bladeren als kinderen heeft (wegens het gebruik van endogenen bomen) dan kunnen we deze node simpelweg verwijderen. Als dat niet het geval is moeten we eerst deze node omlaag roteren tot hij slechts twee bladeren als kinderen heeft. Om te weten met welk kind we moeten roteren, kijken we naar de prioriteiten van deze kinderen. We roteren dan met dat kind dat de hoogste prioriteit heeft om geen verdere inbreuken op de heap volgorde te maken. Op deze wijze zorgen we ervoor dat de enige node die de heapvolgorde schendt de te verwijderen node is. Na de verwijdering is bijgevolg de heapvolgorde hersteld. De boom is ook nog steeds een zoekboom daar deze eigenschap niet aangetast wordt door een rotatie.

Voor de $JOIN(S_1, k, S_2)$ kunnen we ook weer bijna hetzelfde doen als bij gewone zoekbomen. Maak van het nieuwe koppel de wortel en hang S_1 eraan als linkerdeelboon en S_2 als rechterdeelboom. We geven k een willekeurig gekozen prioriteit. Nu moeten we ook nog rekening houden met de prioriteiten. Als de prioriteit van de node lager is dan die van zijn kinderen moeten we weer rotaties gebruiken totdat de heapvolgorde herstelt is. De operatie $PASTE(S_1, S_2)$ kunnen we hetzelfde doen als bij een gewone binaire boom. Dus eerst deleten we de grootste key uit S_1, en daarna voeren we $JOIN(S_1, k, S_2)$ uit met k het net verwijderde element van S_1. Hier maakt het niet echt uit als de oude prioriteit van k gebruiken of een nieuwe

generen. De vorm van de boom zal waarschijnlijk verschillen maar niet in die mate dat het invloed gaat hebben op de performantie.

De operatie $SPLIT(k, S)$ kunnen we dan weer als volgt uitvoeren. Eerst verwijderen we de node die k bevat. Dan voegen we een nieuwe node toe aan de treap met als key k maar met prioriteit ∞. Het is duidelijk dat deze node de wortel van de treap zal worden. Het is dan makkelijk om de linker- en rechterboom eruit te halen en terug te geven. Beide zijn dan geldige treaps en de linkertreap bevat alle elementen waarvan de key kleiner is dan k, terwijl de rechter alleelement bevat waar deze groter is dan k.

Nu hebben we alle update operaties gedefiniëerd in termen van $INSERT$ en $DELETE$. We kunnen ook echter $INSERT$ en $DELETE$ definiëren aan de hand van $JOIN$, $PASTE$ en $SPLIT$ en daarna een rechtstreekse implementatie geven voor deze operaties.

$INSERT$ kunnen we als volgt uitvoeren. We voeren simpel weg $JOIN(S, k, \emptyset)$ uit, waarbij S de bestaande treap is, k het toe tevoegen element en $\emptyset$ staat voor een lege treap. $DELETE$ voeren we uit door eerst $SPLIT$ uit te voeren met de te verwijderen waarde en daarna de resulterende bomen met behulp van $PASTE$ terug aan elkaar te plakken.

De operatie $JOIN$ is al op directe wijze geïmplementeerd. Nu voor de operatie $SPLIT$. We zoeken eerst de node waarop we gaan splitsen. daarna roteren we deze naarboven totdat hij de wortel geworden is. Zodra hij de wortel is het makkelijk om de linker- en de rechterdeelboom terug te geven al het resultaat. Als laatste is er de operatie $PASTE$. Eerst voeren we $SPLIT$ uit op de eerste treap van de $PASTE$. Zo splitsen we het grootste element van deze treap af, zeg k. Daarna voeren we gewoon $JOIN(S_1, k, S_2)$ uit.

We moeten nu alleen nog een manier vinden om goede prioriteiten toe te kennen aan onze key waarden om zo een goede treap te bekomen. Het idee is om een random treap te maken door het willekeurig toekennen van de prioriteiten aan de key waardes. We kiezen de prioriteiten volgen een een bepaalde kansverdeling. De enige voorwaarde op deze kansverdeling is dat deze er voor moet zorgen dat alle prioriteiten verschillend zijn. In het algemeen volstaat het om een continue verdeling te gebruiken. De kans dat er dan twee dezelfde prioriteiten gegenereerd zorden is dan immers gelijk aan $\frac{1}{\infty}$. De enige moeilijkheid is dat het model van computatie dat wij gebruiken niet toestaat om een continue kansverdeling te samplen. Voorlopig zullen we echter aannemen dat het wel kan, zie oef 5.2 invullen voor hoe men wel makkelijk prioriteiten kan kiezen.

Tijdcomplexiteiten

Om de tijdscomplexiteiten van de verschillende operaties uit te rekenen moeten we eerst nog een paar dingen aantonen. Allereerst heeft een treap een belangrijke eigenschap die we de geheugenloosheid noemen. Dit is een handige eigenschap voor het gebruik bij bewijzen. Wegens theorema 5.2.1

weten we dat zodra de prioriteiten vastliggen ook de treap vastligt. De volgorde waarin het toevoegen gebeurd maakt niet uit voor de vorm van de boom. Daar we in een random treap de prioriteiten onafhankelijk kiezen, kunnen we deze vastleggen voordat we met toevoegen beginnen. Dus kunnen we nu zonder verlies aan algemeenheid stellen dat alle elementen toegevoegd worden in volgorde van afnemende prioriteit. Een groot voordeel van deze visie is dat alle insert gebeuren aan de bladeren en er dus geen rotatie meer nodig zijn.

Als we hier dieper over nadenken dan zijn er overeenkomsten met het gerandomizeerde Quicksort algoritme uit sectie **??**. Een treap neemt als wortel steeds het element met de hoogste prioriteit. Dan wordt de oorspronkelijke verzameling in twee gedeeld, één deel heeft alle elementen die kleiner zijn dan het element van de wortel, het andere deel alle elementen die groter zijn. Dan wordt recursief verder gegaan. Van elk deel wordt weer het element met de grootste prioriteit gekozen en wordt weer verder geplist tot er niks meer overblijft. Het quicksort algoritme doet ongeveer hetzelfde. Alleen kiest het niet de grootste prioriteit uit de (deel)verzameling maar kiest het gewoon een element volkomen willekeurig maar volgens een uniforme verdeling. Daar de prioriteiten op gelijkaardige wijze gekozen worden komt dit echter op hetzelfde neer. Het quicksort algoritme deed een verwacht aantal van $O(\log n)$ stappen, dat komt in dit geval overeen met een boom met een hoogte van $O(\log n)$, althans dat vermoeden we.We kunnen het ook bewijzen.

Definitie 5.2.1. De diepte van een node is de afstand tot de wortel en we noteren $diepte(x)$ om deze lengte voor een node x aan te geven.

Lemma 5.2.2. *Zij T een random treap voor een verzameling S met grootte n. Voor elk element $x \in S$ met een rang k geldt dan:*

$$E[diepte(x)] = H_k + H_{n-k+1} - 1$$

Bewijs. Definiëer de verzamelingen $S^- = \{y \in S \mid y \leq x\}$ en $S^+ = \{y \in S \mid y \geq x\}$. daar de rang van x gelijk is aan k volgt dat de grootte van S^- gelijk is aan k en de grootte van S^+ is dan $n-k+1$. De verzameling Q_x is dan de verzameling die alle elementen bevat die zich op de nodes bevinden op het pad van de wortel naar x. Laat $Q_x^- = Q_x \cap S^-$ en $Q_x^+ = Q_x \cap S^+$ zijn. We zullen nu aantonen dat $\mid Q_x^- \mid = H_k$. Wegens symmetrie door het omkeren van de totale orde op de verzameling S volgt dan ook dat $\mid Q_x^+ \mid = H_{n-k+1}$.

Neem een voorouder $y \in Q_x^-$ van de node x. Doordat we gebruik maken van de geheugenloosheid van een treap volgt hieruit dat y toegevoegd werd voor x. Daar $y < x$ ligt x in de rechterdeelboom van y. We kunnen zelfs zeggen dat alle elementen $z \in S$ en $y < z < x$ in de rechterdeelboom van y zitten. Want het pad om de node x te bereiken volgt tot de node y bereikt wordt hetzelfde pad als dat om de node y te bereiken. Als er nu een z was

die niet in de rechterdeelboom van y zat dan moet het dus een voorouder van y zijn. Maar op dat punt is er een verschil van pad tussen het pad naar x en het pad naar y. Dat is in tegenspraak met het feit dat y een voorouder is van x. Dit houdt in dat y een voorouder is voor alle elementen tussen x en y. Door de geheugenloosheid van treaps houdt dit in dat alle elementen tussen x en y zijn toegevoegd na y en dus een lagere prioriteit hebben.

Door het voorgaande weten we nu dat y een voorouder is van x als en slechts als y het grootste element van S^- in de treap is ten tijde van het invoegen van y. Daar de prioriteiten uniform verdeeld zijn, kunnen we het toeveoegen beschouwen als het uniform sampelen zonder teruglegging van S^-. We tellen nu 1 voor elke keer we het grootste element pakken. Wat is dan het totaal dat we uitkomen? Zij P_k dit totaal. Zodra we een voorouder gekozen hebben met een rang i wordt de verwachte waarde $1 + P_{i-1}$. Daar elke element evenveel kans heeft om als eerste als voorouder gekozen te worden krijgen we de volgende recursieve vergelijking:

$$P_k = \sum_{i=1}^{k} \frac{1 + P_{i-1}}{k} = 1 + \sum_{i=1}^{k} \frac{P_{i-1}}{k}$$

Nu, gebruik makend van het feit dat $P_0 = 0$ en door herschrijven krjgen we:

$$kP_k - k = \sum_{i=1}^{k-1} P_i$$

Uit subsectie 2.3.2 weten we dat de harmonische getallen aan deze vergelijking voldoen. Dus we verwachten H_k keer goed te kiezen. De grootte van Q_x^- is dus ook H_k. Wegens de eerde rvermelde symmetrie geldt dan ook dat Q_x^+ een grootte heeft van H_{n-k+1}. Daar $Q_x^- \cap Q_x^+ = x$ volgt dus dat de verwachte grootte van Q_x gelijk is aan $H_k + H_{n-k+1} - 1$. Hieruit volgt het gevraagde. □

Uit bovenstaand lemma volgt nu dat de verwachte diepte van een element x met rang k gelijk is aan $O(\log k + \log(n - k + 1))$ (zie sectie 2.3.1). Dat laatste is altijd gelijk aan $O(\log n)$

Nu moeten we ook nog het aantal rotaties gaan tellen tijdens $DELETE$ en $INSERT$

Definitie 5.2.2. De linkerspine van een boom is het pad verkregen door te beginnen bj de wortel telkens het linkerkind te nemen tot een blad bereikt wordt. De rechterspine van een boom is gelijkaardig, alleen neemt men dan telkens het rechterkind.

Lemma 5.2.3. *Het aantal rotaties tijdens een uitvoering van de $DELETE$ operatie op een element k is gelijk aan de som van de lengte van de rechterspine van de linkerdeelboom van de node van k met de lengte van de linkerspine van de rechterdeelboom van de node van k.*

Bewijs. Dit is een gevolg van het gedrag van een rotatie. Stel we voeren een rotatie uit met het linkerkind van k. Dan blijft het rechterkind van k en dus ook de rechterdeelboom van k gelijk. De linkerdeelboom van het linkerkind van k wordt nu de linkerdeelboom van k zelf. Hierdoor is de lengte van de rechterspine met 1 afgenomen, terwijl we 1 rotatie hebben uit gevoerd. Als De lengte van bijde spines nul is dan zijn beide kinderen (lege) bladeren. We kunnen voor het rechterkind natuurlijk dezelfde redenering maken als voor het linkerkind. Het aantal rotaties dat we moeten uitvoeren zodat beide kinderen bladeren zijn, is dus de som van de lengtes van beide spines. □

Bovenstaand resultaat geldt ook voor de $INSERT$ operatie. Een toevoeging heeft immer evenveel rotaties nodig als een $DELETE$ nodig heeft om een node terug te verwijderen (mits er ondertussen geen andere nodes zijn toegevoegd.)

Nu moeten we alleen nog de verwachte lengtes van de spines uitrekenen om te weten hoeveel rotaties we verwachten nodig te hebben. Daartoe dient volgend lemma.

Lemma 5.2.4. *Zij T een random treap voor een verzameling S van grootte n. Zij $x \in S$ van rang k. Zij de rechterspine van de linkerdeelboom van de node van x R_x. Zij de linkerspine van de rechterdeelboom van de node van x L_x. Dan geldt:*

$$E[R_x] = 1 - \frac{1}{k}$$

en

$$E[L_x] = 1 - \frac{1}{n-k+1}$$

Bewijs. We bewijzen ook hier alleen de eerste vergelijking. De tweede volgt dan wegens symmetrie en het omkeren van de totale orde op S. We willen dus de lverwachte engte van de rechterspine van de linkerdeelboom berekenen. Merk op dat het laatste element van deze spine de voorganger van het element x is. Als deze voorganger een voorouder is van x dan is het linkerkind van x een blad. Een tweede opmerking is dat het pad naar de voorganger van x zal verlopen via de node van x en dan verder via de rechterspine van de linkerdeelboom van x, tenzij deze voorganger een voorouder is van x. Als we dus alle elementen nemen die kleiner of gelijk zijn aan de voorganger van x uit S op het pad naar de voorgenger van x en daar alle elementen die kleiner of gelijk zijn aan de voorouder van x die ook op het pad naar x zelf liggen afhalen hebben we de lengte van de rechterspine van de linkerdeelboom van x. Namelijk het aantal elementen dat overblijft. Dit gaat zelfs op al de voorganger een voorouder is van x. De elementen zijn dan dezelfden zodat er geen enkele overblijven en we dus een spine van lengte nul hebben. We weten uit het bewijs van lemma 5.2.2 dat de het verwachte aantal elementen op het pad naar de voorganger van x in een random treap, die kleiner of gelijk zijn aan deze voorganger gelijk is aan H_{k-1} want de rang

van de voorganger is één minder dan de rang van x. Het verwachte aantal elementen dat kleiner of gelijk is aan de voorganger van x op het pad naar x is volgens hetzelfde bewijs $H_k - 1$. We moeten er hier namelijk één aftrekken omdat x zelf natuurlijk groter is dan zijn voorganger. Het verwacht aantal elementen dat overblijft is dus $H_{k-1} - H_k + 1$. Dit kunnen we verder uitrekenen en dan bekomen we $E[R_x] = 1 - \frac{1}{k}$. De tweede vergelijking volgt dan zoals eerder opgemerkt uit symmetrie bij het omkeren van de totale orde. □

Nu kunnen we met behulp van deze lemma's de totale kosten gaan uitrekenen voor de operateis van een treap.

Theorema 5.2.5. *Zij T een random treap van een verzamelin S van grootte n dan;*

1. *Het aantal verwachte rotaties tijden een $INSERT$ of een $DELETE$ is ten hoogste twee.*

2. *De verwachte looptijd voor een $FIND$, $INSERT$ of $DELETE$ operatie op T is dan $O(\log n)$*

3. *De verwachte looptijd voor een $JOIN$, $PASTE$ of $SPLIT$ operatie die gebruik maakt van twee verzamelingen S_1 en S_2 met respective groottes n en m is $O(\log(n) + \log(m))$*

Bewijs. 1. We weten wegens lemma 5.2.3 dat het aantal rotaties bij een $DELETE$ en een $INSERT$ gelijk is aan de som van de lengte van de rechterspine van de linkerdeelboom met de lengte van de linkerspine van de rechterdeelboom van het element dat we verwijderen dan wel toevoegen. In lemma 5.2.4 hebben we aangetoond dat de som van deze verwchte lengtes gelijk is aan $(1 - \frac{1}{k}) + (1 - \frac{1}{n-k+1})$. We kunnen beide termen nu afschatten door 1. De maximaal verwachte som is dan dan ook 2 . Dus het verwacht aantal rotaties is dan ook ten hoogste 2.

2. Uit lemma 5.2.2 weten we dat de verwachte diepte van een element $O(\log n)$ is. We weten ook dat bij binaire bomen de looptijd van een $FIND$ operatie evenredig is met de diepte. Dus $FIND$ heeft een verwachte looptijd van $O(\log n)$.

   $DELETE$ en $INSERT$ voeren eerste een $FIND$ uit gevolgd door een aantal rotaties. Uit het voorgaande deel van dit theorema weten we dat het verwacht aantal rotaties twee is. Dit is een constante en dus zal de tijd voor het uitvoeren van $FIND$ overheersen. We kunnen dus besluiten dat de verwachte looptijden ook hier $O(\log n)$ zijn.

3. Bij de operatie $JOIN$ woegen we twee treaps samen. Het samenvoegen opzich gebeurt in constante tijd, gevolgd door een aantal rotaties om

de wortel op zijn plaats te zetten. Het maximaal aantal rotaties hangt af van de rechterspine van S_1 en de linkerspine van S_2. De lengte van een spine is altijd kleiner of gelijk aan de maximale diepte van een boom. We weten dat de verwachte diepte van elk element van een random treap van de orde $O(\log i)$ met i de grootte van de treap is. Het maximaal aantal verwachte rotaties is dan ook $O(\log n + \log m)$. En de verwachte looptijd wordt dan ook $O(\log n + \log m)$.

De $PASTE$ operatie voert eerst een $DELETE$ uit in de eerste boom, gevolgd door een $JOIN$. De verwachte looptijd is dan ook $O(\log n + \log n + \log m) = O(\log n + \log m)$.

Voor het uitvoeren van $SPLIT$ voeren we eerst een $DELETE$ uit, gevolgd door een $INSERT$. Daarna splitsen we de resulterende boom in twee. Dit laatste kan in constante tijd. Het probleem is dat de $INSERT$ geen random prioriteit heeft maar oneindig. Dat wil zeggen dat hij zal roteren naar boven tot hij de wortel wordt. Daar alle rotaties over de spines van de resulterende boom verlopen zijn, zijn er $O(\log n + \log m)$ verwachte rotaties nodig.

□

5.2.3 Skip Lists

Wat zijn Skip Lists?

Definitie 5.2.3. Neem een verzameling S uit een universum waarop een totale orde bestaat. Een leveling met r levels van deze geordende verzameling S is een sequentie van geneste deelverzemalingen (deze noemen we levels)

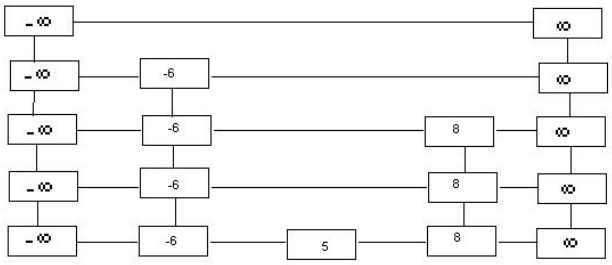
$$L_r \subseteq L_{r-1} \subseteq ... \subseteq L_2 \subseteq L_1$$

zodanig dat $L_r = \emptyset$ en $L_1 = S$.

Definitie 5.2.4. Gegeven een geordende verzameling S en een leveling voor deze verzameling. We definiëren dan het level van een willekeurig element $x \in S$ als

$$l(x) = \max\{i \mid x \in L_i\}$$

Nu gaan we een datastructuur opbouwen en deze zullen we dan een skiplist noemen. ten eerste hebben we een leveling van de geordende verzameling S die we in de datastructuur zullen opslaan nodig. Om het ons makkelijk te maken nemen we het bestaan van twee speciale elementen aan die een element zijn van elk level namelijk het element $-\infty$ dat kleiner is dan alle andere elementen. Het andere element is groter dan ale andere mogelijke elementen en heeft als symbool $+\infty$. Deze elementen zijn lid van elk level van leveling. De gegeven leveling heeft r levels. Het bovenste level is leeg op de speciale elementen $-\infty$ en $+\infty$ na. Het level L_1 is een geordende



Figuur 5.1: Een skip list.

gelinkte lijst van de verzameling S. De tussenliggende levels zijn ook geordende gelinkte lijsten, maar dan van de deelverzameling van S die dat level is. Dezelfdfe elementen van de verschillende levels zijn met elkaar verbond met pointers van boven naar beneden. Zoals een stapel als het ware. Een element x met als level $l(x) = i$ komt voor in alle levels kleiner of gelijk aan i en het element op het $i$de level heeft een pointer naar het element op het $(i-1)$de level, deze heeft weer een pointer naar het volgende niveau lager enzoverder tot op het laagste niveau. Merk op dat de speciale tekens van level r zijn en op alle levels voorkomen. We gebruken ze dan ook om het begin en het einde aan te geven van de gelinkte lijsten en alle niveaus met elkaar te verbinden. Zie figuur 5.1 voor een illustratie van deze datastructuur.

Definitie 5.2.5. Een interval op een level i is de deelverzameling gevormd door alle elementen van de verzameling S die overspannen worden door een specifieke horizontale pointer op level i.

Wij gaan het hier hebben over random skip lists. We maken een skip list random door voor een random leveling te zorgen. Dit doen we als volgt. Elk element krijgt bij toevoeging in de skip list een random level toegewezen. We kennen dit level toe volgens een geometrische verdeling met parameter $\frac{1}{2}$. Nu wanneer we een verzameling hebben en deze hele verzameling willen representeren dan kunnen we dit ook als volgt interpreteren. Het volgende niveau wordt samengesteld door voor elk element van het huidige niveau een munt op te werpen. Is het resultaat kop dan komt dit element ook voor op het volgende niveau, als het resultaat munt is niet.

Implementatie

De operatie $MAKESET$ is simpel. We maken gewoon een skiplist met net één level en dat level bevat alleen de speciale elementen $-\infty$ en $+\infty$.

$FIND$ kan efficieënt geïmplementeerd worden als volgt. We beginnen op het bovenste niveau van de skip list. Op elk level volgen we de horizontale pointers tot we aankomen in dat interval waar het gezochte element deel van uitmaakt. Zodra we op het huidige niveau dit interval gevonden hebben zakken we een level naar beneden. Dit blijven we doen tot we het onderste niveau berekt hebben of totdat we het gezochte element tegen komen als element op een hoger level.

De operatie $DELETE$ gaat als volgt te werk. We doen net hetzelfde als tijdens $FIND$ maar wanneer we zien dat de volgende horizontale pointer wijst naar het element dat we willen verwijderen dan laten we deze horizontale pointer wijzen naar het volgende element en verwijderen op dit niveau het element dat we wilden verwijderen. Dan dalen we een niveau af met de verticale pointer. Nu volgen we weer de horizontale pointers tot we het te verwijderen element weer tegen komen, zetten dan de horizontale pointer weer goed. dit herhalen we totdat we het element verwijderd hebben op het laagste level. Er is altijd een doel voor de horizontale pointer daar we op het einde altijd het speciale element $+\infty$ hebben. Ten slotte worden alle lege levels die eventueel bovenaan de skip list ontstaan zijn,een na een verwijderd, tot er slechts één overblijft.

Voor het uitvoeren van $INSERT$ gaan we als volgt te werk. Eerst genereren volgens een geometrische verdeling een level voor het in te voegen element. Als dat level hoger ligt dan het huidige hoogste level van de skip list voegen we lege levels toe boven op de huidige skip list totdat het level van de skip lsit hoger is dan het level van het toe tevoegen element. Nu beginnen we weer op het hoogte level en volgen de horizontale pointers tot we het interval bereiken dat het toe te voegen element bevat. Zodra we dat gevonden hebben dalen we met behulp van een verticale pointer een level. Dit blijven we doen tot we het level van het toe tevoegen elemetn bereikt hebben. Zodra we dit level bereikt hebben, dalen we niet meer onmiddelijk, maar voegen eerst het element toe op dit niveau. Daarna dalen we pas af en herhalen de procedure. We houden steeds een pointer bij naar het laatst toegevoegde element zodat we de verticale pointers goed kunnen invullen. Zodra het element is ingevoegd op het laagste level zijn we klaar.

De operatie $JOIN$ is vrij gemakkelijk uit te voeren. Zorg er eerst voor dat beide skiplist hetzelfde level hebben. Dit kan door het toeveoegen van lege levels aan de skip list me tde minste levels. Genereer een level voor het tussen te voegen element. Verhoog eventueel de levels van beide skiplist mocht dit level hoger zijn dat dat van hun. Begin weer met het hoogste level. Op elk level ga naar het element van dat level dat wijst naar het speciale element $+\infty$ laat de horizontale pointer van dat element nu wijzen naar het element waar het speciale element $-\infty$ van de tweede skiplist op hetzelfde level naar wijst. Zodra we op het niveau komen van het tussen te voegen element moeten we de procedure lichtjes aanpassen. Laat hier het element van de eerste skiplist wijzen naar het tussen te voegen eleemnt

en laat het tussen te voegen elemetn wijzen naar het eerste element uit de tweede skiplist. Blijf dit herhalen tot op het laagste niveau.

Voor de oepratie $PASTE$ doen we ongeveer hetzelfde. Alleen is er bij deze operatie geen tussen te voegen element dus het tweede deel van de $JOIN$ operatie valt weg, we blijven gewoon telkens het laatste element van de ene skip list linken met het eerste van de tweede skiplist.

De operatie $SPLIT$ begint zoals altijd op het hoogste niveau. Volg op elk niveau de horizontale pointer zoalng het element waar deze pointer naar wijst kleiner is dan dan het element warop we gaan splitten. We maken twee speciale elementen aan. Een $+\infty$, waar we de horizontale pointer naar wijzen. En een $-\infty$, die we laten wijzen naar het element waar de horizontale pointer naar wees als het element waar de horizontale pointer naar wees groter is dan de splitter, als hij wees naar de splitter zelf dan laten we hem wijzen naar het element waar de horizontale pointer van de splitter naar wijst. We houden ook weer pointers bij naar de laatste ingevoegde elementen zodat we verticale pointer juist kunnen zetten. Op deze manier onstaan er twee skip lists. De ene bevat alle elementen van de oorspronkelijke skip list die kleiner zijn da n de splitter, de andere deze die groter zijn.

Tijdcomplexiteiten

We gaan nu de operaties van een random skip list analyseren met betrekking tot hun efficiëntie. Voor we gaan beginnen aan de tijdscompexiteiten van de verschillende operaties, gaan we aantonen dat een random skiplist maar een verwacht geheugengebruik heeft van $O(n)$. Dit kan als volgt. Daar het aantal keren dat eenzelfde element op verschillende niveau's voorkomt gelijk is aan zijn level. Het level van een element wordt gekozen volgen een geometrische verdeling met parameter $\frac{1}{2}$. Het verwachte level van een willekeurig element is dus twee. Ergo We verwachten dat elk element tweemaal opgeslagen dient te worden. Er is dus naar verwachting $2n = O(n)$ geheugen nodig.

Nu gaan we de tijdscomplexiteiten bewijzen. Daarvoor gaan we eerst bewijzen wat het verwachte aantal levels in een random skip list is.

Theorema 5.2.6. *Het verwachte aantal levels r in een random leveling van een verzameling S van een grootte n is $E[r] = O(\log n)$. Het is zelfs zo dat $r = O(\log n)$ met grote kans.*

Bewijs. Het aantal levels $r = 1 + \max_{x \in S} l(x)$ en de levels zijn stochastische variabelen die verdeeld zijn volgend een geometrische verdeling met parameter $p = \frac{1}{2}$. Dat wil zeggen dat het verwacht aantal elementen op een level $i + 1$ de helft is van het aantal elementen op level i. Dus verwachten we dat het hoogste level waarop zich een element bevindt, het level $\log n$ is. Het totaal aantal levels in een skip lsit is 1 meer door het lege bovenste level dus $E[r] = O(\log n)$

Nu kunnen we alle elementen van S ook zien als n onafhankelijke stochastische variabelen $X_1, X_2, ..., X_n$. Het is dan duidelijk dat geldt $P[X_i > t] \leq (1-p)^t$ en dus ook

$$P[max_i X_i > t] \leq n(1-p)^t = \frac{n}{2^t}$$

dat laatste natuurlijk omdat $p = \frac{1}{2}$. Vullen we nu in dat $t = \alpha \log n$ en $r = 1 + \max_i X_i$ dan bekomen we:

$$P[r > \alpha \log n] \leq \frac{1}{n^{\alpha-1}}$$

voor alle $\alpha > 1$ Hieruit kunnen we besluiten dat de kans dat r groter is dan $\log n$ exponentieel afneemt. □

We kunnen een skip list ook beschouwen als een boom. De delen van het interval op het $i$de level zijn dan de kinderen van de node die we kunnen identificeren met het interval. We kunnen alleen het meest linkse kind rechtstreeks bereiken. De andere kinderen kunnen we alleen bereiken door telkens de horizontale pointer te volgen. We zullen nooit door het volgen van een horizontale pointer bij kinderen van een ander interval terecht komen. Dan hadden we namelijk op een hoger niveau al naar een volgend interval moeten gaan voordat we een niveau zakten. We weten ook dat de hoogte van een skip list $O(\log n$ is. Maar de boom die gevormd wordt door deze skip lsit is niet noodzakelijk binair. We moeten dus ook uitzoeken hoevel keer we naar een rechterkind moeten wandelen per level. Het is makkelijk te zien als een element y behoort tot het huidige interval of een interval rechts van het huidige. De kost voor een $FIND$ en dus bij extentie voor de andere operatoren hangt dus niet alleen af van de hoogte van de skip lit maar ook van het aantal nodes die bezocht worden op elk niveau. Het aantal nodes dat bezocht wordt op niveau i, is niet hoger dan het aantal kinderen van het interval dat dat y bevat één niveau hoger. Stel het aantal kinderen van het interval dat y bevat op niveau i voor door $c_i(y)$. Nu kunnen we de kost afschatten met

$$O\left(\sum_{j=1}^{r}(1 + c_i(y))\right)$$

Merk op dat dit een stochastische variabele is, zowel r als het aantal kinderen zijn bepaald door willekeurige keuzes die gemaakt zijn bij het opstellen van de skip list.

Lemma 5.2.7. *Zij y een element van de random skip list S met grootte n. Dan is de verwachte kost voor een $FIND$:*

$$E\left[\sum_{j=1}^{r}(1 + c_i(y))\right] = O(\log n)$$

Bewijs. We gaan eerst aantonen dat $E[c_j(z)] = O(1)$ voor elke willekeurig j en z. Zij J een willeukrig interval van de skip list op level i. We gaan aantonen dat het aantal broers van J gebonden wordt door een constante. Dit duidt erop dat ook het verwachte aantal gebonden zal zijn door een constante. Het is voldoende aan te tonen dat het aantal rechterbroers gebonden is door een constante, als dit geldt voor elk interval, geldt het ook voor het meest linkse kind. Zij de intervallen rechts van J, $J_1 = [x_1, x_2], J_2 = [x_2, x_3], ..., J_k = [x_k, +\infty]$. Deze intervallen bestaan alleen op level i als en slechts als $x_1, ..., x_k \in L_i$. Als J s rechterbroers heeft, dan moet het zo zijn dat $x_1, ..., x_s \notin L_{i+1}$ en dat $x_{s+1} \in L_{i+1}$. Sinds elk element van L_i onafhankelijk van elkaar met kans $\frac{1}{2}$ in L_{i+1} zitten, wordt het aantal rechterbroers dus bepaald door een syochastische variabele die geometrisch verdeeld is met parameter $\frac{1}{2}$. Het verwachte aantal rechterbroers is dus hoogstens twee. We hebben nu $E[c_j(z)] = O(1)$. We kunnen dat nu niet snel vermenigvuldigen met de waarde die we voor r verwachten doordat deze twee stochastische veranderlijken niet onafhankelijk van elkaar zijn. Maar we weten dat $r = O(\log n)$ met hoge waarschijnlijkheid. Daar $r > \frac{1}{n^{\alpha-1}}$ en $\sum_j c_i(y) = O(n)$ zullen de gevallen waar $r > 2 \log n$ niet significant gaan bijdragen. We kunnen dus besluiten dat de kost voor een $FIND$ $O() \log n$ is. □

Theorema 5.2.8. *In een random skip list S met grootte n kunnen de operaties $FIND$, $INSERT$ en $DELETE$ uitgevoerd worden binnen een verwachte tijd van $O(\log n)$*

Bewijs. We weten wegens lemma 5.2.7 dat de operatie $FIND$ uitgevoerd kan worden in een verwachte tijd van $O(\log n)$. We gaan u ook aantonen dat hetzelfde geld voor de operaties $INSERT$ en $DELETE$.

Een $INSERT$ begint met het eventuele invoegen van extra niveaus. Dit aantal is afhankelijk van de hoogte van de skip list en de verwachte hoogte is $O(\log n)$, dat kan dus in $O(\log n)$ tijd. Dan wordt er in wezen een $FIND$ uitgevoerd, alleen worden er na verloop van tijd inttervallen gesplitsd. Dat splitsen brengt op zich slechts enkele opdrachten met zich mee. De verwachte gebruikte tijd is gelijk aan die van een gewone $FIND$, welke $O(\log n)$ is. In het totaal is de verwachte tijd voor een $INSERT$ dan ook $O(\log n)$.

Een $DELETE$ operatie is ook weer een $FIND$ met iets extra. WE doen gewoon find en na verloop van tijd moeten we twee intervallen samennemen. Dit samennemen kost per samenneming ook hier slechts constante tijd daar hetslechts het goedzetten van enkele pointers is. De kost voor dit deel is dan $O(\log n)$. Nu kan het zijn dat een $DELETE$ levels moet verwijderen die leeg geworden zijn. Dit zijn er dus hoogstens r en dit is ook weer gebonden $O(\log n)$. De totale tijd is dus ook hier weer $O(\log n)$.

□

5.3 Oefeningen

Opgave 5.1. *Gegeven een verzameling keys $S = \{k_1, k_2, ..., k_n\}$, construeer een random treap voor S waarbij we geen loze bladeren introduceren om de boom endogeen te maken. Heeft elk element van S evenveel kans om een blad te zijn? Wat impliceert dat voor de performantie van een treap?*

Daar het al of niet blad zijn afhangt van de volgorde van invoegen in een gewone boom en daar deze volgorde volkomen willekeurig bepaald wordt, heeft elk element evenveel kans dat het een blad zal zijn in dit soort treaps. Dit is ook logisch daar we weten dat het al dan niet endogeen maken van een boom geen invloed heeft op de performantie.

Opgave 5.2. *Laten we nu analyseren hoeveel random bits we nodig hebben om de operaties van een random treap te implementeren. Stel dat we de prioriteiten p_i uniform kiezen uit het interval $[0, 1]$. Dan kunnen we elke p_i binair voorstellen als een reeks van bits (mogelijk oneindig lang), waarvan elke bit het resultaat is van het opwerpen van een munt. Het idee is nu om maar net zoveel bits te genereren als we nodig hebben om elke vergelijking tussen prioriteiten op te kunnen lossen. Stel dat we een aantal elementen in onze random treap hebben. Elk van deze elementen heeft een prefix van zijn prioriteit. Als we nu een nieuw element y gaan toevoegen in de treap, dan gaan we vergelijken met zijn prioriteit p_y. Als er genoeg bits zijn in een p_i om een conculsie te kunnen trekken als p_i vergeleken wordt met p_y dan moeten we niks doen. Als het niet mogelijk is dan hebben ze dezelfde prefix en moeten we de korste (of allebei) aanvullen met random bits totdat we een bit tegenkomen die verschilt voor beide prioriteiten. Bereken een bovengrens voor het verwachte aantal bits dat we nodig hebben voor elke prioriteit.*

We weten dat we om n van zulke prefixen te vergelijken dat de verwachte lengte van deze prefixen $O(\log n)$ is (zie opgave 3.5.) We verwachten dan ook $O(\log n)$ bits nodig te hebben voor elke prefix, anders kunnen we niet altijd een onderscheidt maken op plaatsen waar dat nodig was.

Opgave 5.3. *bereken een bovengrens voor het verwachte aantal muntopwerpingen of random bits die nodig zijn voor elke update operatie voor een random skip list.*

We beschouwen alleen de INSERT operatie, voor de DELETE operatie zijn er geen random bits nodig. Voor de INSERT operatie moeten we een niveau kiezen waarop het element in de leveling gaat komen. Deze keuze hangt niet af van het niveau van de andere elementen in de skip list. Deze hoogte wordt gekozen volgens een geometrische verdeling met een parameter van $\frac{1}{2}$. De hoogte is vb het aantal keren dat we een munt opgooien totdat we voor de eerste keer kop krijgen. doordat dit een standaard voorbeeld is weten we dat eht verwachte aantal keren dat we moeten gooien gelijk is aan

2. We verwachten dus dat we twee random bits per INSERT nodig zullen hebben.

Opgave 5.4. *Bij het definiëren van een leveling voor een skip list, kiezen we elementen uit een laag met kans* $\frac{1}{2}$ *om naar het hogere level te gaan. Beschouw nu in plaats daarvan een skip list waarbij we kiezen uit de elementen met een kans* p *waarbij* $0 < p < 1$*.[5]*

a) Bepaal het aantal verwachte levels voor een dergelijke skip list.

b) Wat is de relatie tussen de verwachte kost voor een operatie en de waarde van p*?*

a) We merken eerst op dat hoe groter p wordt des te hoger het hoogste level van de skip list is. Daar we op elk level verwachten om nog $\frac{1}{p}$ van het aantal elementen vorige level over te houden, zullen we dat naar verwachting $\log_{\frac{1}{p}} n$ keer kunnen doen voordat we zonder elementen zitten. We verwachten dan ook dat de hoogte van een dergelijke skip list gelijk is aan $\log_{\frac{1}{p}} n + 1$. We zien dus dat de verwachte hoogte zeer duidelijk afhangt van de parameter p

b) We merken op dat als de parameter p groot is, dat er dan meer levels zijn. Hierdoor zal de performantie van de skip list afnemen doordat er meer levels doorkrijst moeten worden. Het minder aantal nodes per level is niet genoeg om dit te compenseren. Op deze manier gaat de kost voor de FIND operatie en bijgevolg voor alle andere operaties, de hoogte in.

Aan de andere kant als de parameter p klein is, zullen er minder levels zijn. Het probleem hier echter is dat er meer nodes per level zullen zijn. Ook hier neemt daardoor de performantie af naarmate p verder daalt. We kunnen dus concluderen dat de performantie het hoogst is, zolang de parameter p in de buurt van $\frac{1}{2}$ bevindt. Als we wat verder kijken zien we dat in de limieten $p = 1$ en $p = 0$ de slechtste performance krijgen: namelijk in het eerste geval is de skip list oneindig groot waardoor we nooit iets bereiken, in het tweede geval hebben we een gewone linked list.

Hoofdstuk 6

Online Algoritmes

Meestal als algoritmen bestudeerd worden, gaat men uit van algoritmen die alle hun inputs krijgen op hetzelfde moment. Kijk maar naar een Turing machine ; men geefteen tape met daarop alle invoer variabelen op en dan begint de Turing machine te werken op deze tape en gebaseerd op zijn instructies gaat hij door tot hij een output produceert. Terwijl de Turing machine aan het rekenen is komen er niet ineens inputs bij, of veranderen de waardes op de tape ineens van waarde door een kracht van buitenaf. Nu gaan we echter eens kijken naar online algoritmen. In een typisch geval van een online algoritme krijgt dit algoritme verzoeken om een dienst. Elk verzoek moet afgehandeld worden voor er aan het volgende verzoek begonnen kan worden. Tijdens het afhandelen van een elk verzoek kan het algoritem keuzes maken die invloed kunnen hebben op het afhandelen van volgende verzoeken. Elke keuze heeft natuurlijk een kost en door de invloed van keuzes nu op de keuzes van volgende verzoeken hebben keuzes nu ook invloed op de ksot van volgende verzoeken. De bedoeling is natuurlijk om de kost voor het uitvoeren van een reeks verzoeken zo laag mogelijk te houden. Het bekenste voorbeeld van een online algoritme is het beheren van het pagefile van het besturingssysteem.

Bij online algoritmes is het moeilijk om nog te werken met absolute noties met netrekking tot de performantie van dergelijke algoritmes. Dit komt omdat we voor zowat elk algoritme een reeks van verzoeken kunnen samenstellen die ervoor zorgt dat we geen limiet meer op de kost kunnen plakken. In de plaats van een absolute kost zullen we gaan werken door de kost te gaan vergeliken met een offline algoritme dat dezelfde serie verzoeken moet afhandelen. Het offline algorime krijgt natuurlijk alle informatie op voorhand. Dit soort analyses wordt competitive analyse genoemd.

6.1 Het Online Paging Probleem

6.1.1 Wat is het Online Paging Probleem?

Het eerste online probleem dat we gaan bestuderen is het klassieke geval van het paging probleem. Stel het geheugen van een computer is georganiseerd in twee lagen. De bovenste laag bestaat uit snel, maar relatief duur geheugen. Dit geheugen heeft slechts een beperkte opslagcappaciteit. Het kan dus maar een beperkt aantal items bevatten. De onderste laag bestaat uit goedkoper, maar trager geheugen. Deze laag kan zo goed als een onbegrensde hoeveelheid items opslaan. Een paging algoritme bepaald dan welke items er in de bovenste laag zitten. Als er een item opgevraagd wordt en dit item bevindt zich in de bovenste laag dan spreken we van een hit. Indien dit item zich in de onderste laag bevind dan spreken we van een miss en dan krijgt het algoritme een kost aangerekend voor laden van dit item uit de onderste laag geheugen. Dit soort van memory model wordt onderandere gebruikt op een PC. De bovenste laag is dan het werkgeheugen, de onderste laag bestaat dan uit het zogenaamde virtueel geheugen op de harde schijf. Het operating systeem moet dan terwijl verschillende programma's uitgevoerd worden bepalen welke items het in het werkgeheugen (ook wel cache genaamd) wil hebben en welke op de tragere harde schijf in eht virtueel geheugen komen te staan.

Daar dit toch een belangrijk deel is voor een besturingssysteem dat multitasking toelaat is hiernaar redelijk wat onderzoek gebeurt. De moeilijkheid is er voor te zorgen dat items die meerdere keren worden opgevraagd in de cache te houden terwijl items die weinig tot niet gebruikt worden te verhuizen naar het tragere geheugen. Het probleem is natuurlijk dat er altijd te weinig werkgeheugen is om alle items die gebruikt worden in de cache te houden. Er moeten dus goede kandidaten gevonden worden om uit het snelle geheugen gegooid te worden. En deze kandidaten moeten natuurlijk gevonden worden zonder dat het algoritme kennis heeft over de volgende verzoeken.

Er zijn drie deterministische algoritmes die gebruikt worden door besturingssystemen om het paging probleem op te lossen:

- Minst Recent Gebruikt (MRG) laat het item vallen dat het langst niet gebruikt is geweest.
- Eerst In, Eerst Uit (EIEU ookwel FIFO) deze laat dat item uit de cahce vallen dat er het langst inzit.
- Minst Dikwijls Gebruikt MDG hier wordt dat item uit de cache verwijderd dat de minste verzoeken (en dus hits) heeft gekregen.

Er dient te worden opgemerkt dat niet al deze algoritmes een triviale computationele kost hebben. Het eerste algoritme moet bijvoorbeeld een prioriteit-

enrij bijhouden van alle items in de cache om te weten welke de kanditaat is om verwijderd te worden.

6.1.2 Efficiëntie Berekeningen met Deterministische Algoritmen

Neem $\rho = (\rho_1, \rho_2, ..., \rho_N)$ een reeks van verzoeken. Als we nu een online algoritme A nemen en als A deterministisch is, dan kunnen we voor elk verzoek uit ρ berekenen als A mist of niet. We stellen het aantal keren dat het algoritme A mist op de reeks $\rho_1, \rho_2, ..., \rho_N$ voor door $f_A(\rho_1, \rho_2, ..., \rho_N)$. Nu kunnen we ons ook afvragen wat het minimum aantal missen is, over de reeks $\rho_1, \rho_2, ..., \rho_N$. Stel het aantal keren dat een optimaal offline algoritme mist voor door $f_O(\rho_1, \rho_2, ..., \rho_N)$.

Het volgende Algoritme is gekend als het MIN algoritme en het is een optimaal offline algoritme. Als het MIN algoritme een mis heeft, dan verwijdert het dat element uit de cache dat door de reeks die het aan het behandelen is het laatst wordt opgevraagd van alle elementen die momenteel in de cache zitten. Het is duidelijk dat MIN een offline algoritme is, anders kan MIN onmogelijk iets weten over de toekomstige verzoeken.

Voor het uitrekenen van de efficiëntie van online algoritmes hebben we wel een probleem. Dit kan namelijk niet op de klassieke manier. Neem bijvoorbeeld een simple scenario waarbij we een cache grote hebben van k en er in het totaal $k+1$ items zich in het geheugen bevinden. We kunnen dan een reeks verzoeken opstellen waarbij elk deterministisch online algoritme A zal missen op elk verzoek. Dit kan als volgt. We weten op elk moment welk item er niet in de cache zit. Ons volgende verzoek is dus altijd net dat item. Dus voor een reeks van lengte N zal het algoritme dus N keer missen.

Nu het offline algoritme MIN maakt een betere beurt. Doordat het algoritme ook de verzoeken in de toekomst kent, zal MIN dat item uit de cache verwijderen het laatst wordt opgevraagd in de rest van de reeks. Dit houdt in dat er minstens $k-1$ andere verzoeken moeten worden uitgevoerd en deze zullen allemaal hitten, daar deze items zich al in de cache bevinden. Als er geen $k-1$ verzoeken kunnen worden uitgevoerd die hitten dan bestaat er een element in de cache dat later wordt opgevraagd dan het element dat we verwijderd hebben. We hebben echter expliciet dat element gekozen dat het laatst werd opgevraagd. Dat is dus onmogelijk. Er moeten dus $k-1$ verzoeken zonder mis kunnen worden uitgevoerd. Hier uit volgt dat MIN op elke k verzoeken hoogstens 1 mis heeft. Het aantal keren dat MIN mist op een reeks van lengte N is dus $\frac{N}{k}$.

Voor online algoritmen is het dus niet mogelijk om de worst-case van $f_A(\rho_1, \rho_2, ..., \rho_N)$ te gebruiken voor het meten van eficiëntie. We moeten dus een andere manier bedenken om de eficiëntie van het algoritme te bepalen. We gaan dit doen door het aantal keren dat het algoritme mist te vergelijken met het aantal keren dat het optimale, offline algoritme mist op dezelfde

reeks verzoeken.

Definitie 6.1.1. Een deterministisch, online algoritme A noemt men C-competitief als er een constante b bestaat zodanig dat voor elke reeks $\rho = (\rho_1, \rho_2, ..., \rho_N)$

$$f_A(\rho_1, \rho_2, ..., \rho_N) - C \times f_O(\rho_1, \rho_2, ..., \rho_N) \leq b$$

De constante b moet onafhankelijk zijn van de lengte van de reeks N, maar mag afhangen van de grootte van de cache k. De competititeitscoefficiënt van A, genoteerd C_A is het infimum van C zodanig dat A is C-competitief.

Theorema 6.1.1. *Zij A een deterministisch online paging algoritme. Dan zal $C_A \geq k$*

Bewijs. Zowel het deterministisch online algoritme A als het offline algoritme houden elk een aparte cache van grootte k bij voor dezelfde reeks van verzoeken. Om te beginnen gaan we er vanuit dat beide dezelfde items in hun cache hebben.

We stellen nu een reeks van verzoeken op die volledig bepaald is door het gevrag van A. het eerste verzoek is een verzoek voor een item dat zich niet in de cache van A bevindt. Daar beide algoritmes dezelfde items in hun cache hebben, missen beiden op dit verzoek. Nu zij S de verzameling die bestaat uit alle elementen die in de initële cache zaten van zowel A als het optimale offline algoritme samen met het element dat eerste verzoek behelst. S heeft dus $k + 1$ elementen. Op elk moment zal er ook een item dat een element van S is, zich niet in de cache van A bevinden. Elk volgend verzoek gaat om dat item. Dit zorgt ervoor dat A zal missen op elk verzoek van deze reeks.

Nu gaan we de reeks verzoeken opdelen in rondes. We doen dat op de volgende manier. De eerste ronde begint met het eerste verzoek. Een ronde eindigt zodra er nadat er k verschillend items gevraagd zijn, er een om nieuw item ρ dat niet eerder tijdens de ronde is opgevraagd, verzocht wordt. Dat verzoek wordt dan het eerste van de volgende ronde. Sinds er tijdens een ronde minstens k verschillende items opgevraagd worden en het algoritme A op al deze verzoeken zal missen, zal A minsten k keer misssen tijdens een ronde.

Nu bestaat er een offline algoritme dat slechts één keer zal missen tijden een ronde. In feite zal het alleen missen op het eerste verzoek van een ronde. Er worden slechts k verschillende items opgevraagd tijdens een ronde. We vragen alleen elementen van S op. Dit betekent dat er dus één item is dat niet wordt opgevraagd tot de volgende ronde. Noem dit item ρ. Op het eerste verzoek van de reeks verwijdert het offline algoritme ρ uit de cache. tijdens de gehele ronde zijn er dan ook geen verzoeken meer waarop het offline algoritem kan missen. Dit is net wat het MIN algoritme doet.

Doordat A deterministisch is weet het offline algoritme wat A doet tijdens elke ronde. Het offline algoritme kent ook alle elementen uit de initële cache van A namelijk dezelfde elementen als in zijn cache. Het offline algortime kent ook de voledige reeks verzoeken en in het bijzonder dus ook de ρ van elke ronde. Het offline algoritme zal dus maar één mis hebben elke ronde, die minimaal k verzoeken omvat.

Op het einde van een ronde zal de inhoud van de verschillende cache weer gelijk zijn. dit is duidelijk in te zien daar we slechts items kiezen uit een verzameling van $k+1$ elementen en beide algoritmes missen op het eerste verzoek van elke ronde. Op deze manier kunnen we zoveel rondes maken als we willen terwijl dit blijft opgaan. Als we zovee lrondes kunnen maken als we willen kunnen we dus ook reeksen maken die zolang zin als we willen. Bij deze reeksen zal A altijd missen op elk verzoek, terwijl het offline algoritme hoogstens één verzoek per k verzoeken zal missen. A zal dus k keer meer verzoeken missen als het offline algoritme en dus $C_A \leq k$ □

We merken op dat we in het bovenstaande bewijs geen gebruik gemaakt hebben van enige computationele beperkeningen op A slechts van het feit dat A niet op de hoogte is van toekomstige verzoeken. Dit heeft als gevolg dat dit resultaat geldt voor alle deterministisch algoritmen.

Een andere opermerking is dat het offline algoritme beschouwd kan worden als een tegenstander die niet alleen zijn eigen cache beheert maar ook de reeks van verzoeken genereerd. Dit is een terugkomend beginsel voor alle tegenstanders die we zullen beschouwen voor het onderzoeken van stochastische algoritmen. De tegenstander wil een zo groot mogelijk kost voor het stochastische algoritme terwijl hij probeert de kost voor het andere algoritme probeert te minimaliseren.

6.1.3 Efficiëntie Berekeningen met Stochastische Algoritmen

We vragen ons nu af of het resultaat dat we bekomen hebben voor deterministische algoritmen in de vorige sectie kunnen verbeteren door gebruik te maken van stochastische algoritmen. Neem nu een stochatisch algoritme R voor het oplossen van het paging probleem. Gegeven een reeks verzoeken $\rho = (\rho_1, \rho_2, ..., \rho_N)$ wordt het aantal keren dat R mist een stochastische veriabele. We stellen deze variabele naar analogie met deterministische algoritmen voor als $f_R = (\rho_1, \rho_2, ..., \rho_N)$. Nu gaan we het gedrag van R onderzoeken op reeksen van verzoeken die gegenereerd zijn door verschillende tegenstanders. Het is nu echter zo dat er niet langer een uniek begrip is voor "tegenstander"

Dit komt door het volgende. Wat weet de tegenstander van de keuzes van het stochastische algoritme? We kunnen een tegenstander voorstellen die niets weet over de keuzes van R. De tegenstander kent natuurlijk wel het algoritme R opzich. Dit soort tegenstanders kan de reeks verzoeken op

voorhand opschrijven daar de reeks niet beïnvloed kan worden door de keuze die R maakt tijdens de uitvoering van deze reeks verzoeken. De tegenstander zoekt een worst-case reeks voor R en zelf lost hij deze reeks op MIN. De kost van de reeks voor de tegenstander is geen stochastische variabele, want de reeks ligt vast. We stellen deze kost weer voor als $f_O(\rho_1, \rho_2, ..., \rho_N)$. Dit type tegenstander noemen een oblivios of onwetende tegenstander. De naam komt vanwege het feit dat detegentander niet op de hoogte is (of onwetend) van de random keuzes van R.

Definitie 6.1.2. Een stochastisch algoritme R is C-competitief tegen een onwetende tegenstander als voor elke reeks van verzoeken $\rho_1, \rho_2, ..., \rho_N$,

$$E[f_R(\rho_1, \rho_2, ..., \rho_N)] - C \times f_O(\rho_1, \rho_2, ..., \rho_N) \leq b$$

met b weer een constante die niet afhangt van de lengte van de reeks N, maar wel kan afhangen van de grootte van de cache. De onwetende competititeitscoëfficiënt van R noteren we al C_R^{onw} en deze is gelijk aan het infimum van C zodat R C-competitief is.

Een andere tegenstander die we ons kunnen voorstellen is een tegenstander die kijkt naar welke keuzes R maakt en de reeks verzoeken die gegenereerd worden daaraan aanpast. Op deze manier kan de tegenstander de kost voor het stochastische algoritme opdrijven. Dit is geen punt voor het geval met een deterministisch algoritme, daar de antwoorde nvan dit algoritme vast liggen als de reeks bekent en we dus konden aannemen dat de tegenstander op de hoogte is van de toestand van het deterministische algoritme als de tegenstander de keuze maakt voor het volgend item op de lijst van verzoeken. Het antwoord van een stochastich algoritme daarentegen hangt af van de willekeurige keuzes die dit algoritme gemaakt heeft.

Dit gaan we onderzoeken door middel van adaptieve tegenstanders. Zulke tegenstadners kiezen het volgende verzoek in de rij verzoeken op basis van de beslissingen die het stochastische algoritme genomen heeft bij het uitvoeren van de voorgaande verzoeken. De adaptieve tegenstander kent dus het verleden van het stoschastische, offline algoritme R. Het enige waar de adaptive tegenstander geen informatie over heeft, zijn de beslissingen van die het stochastische algoritme R gaat nemen bij het uitvoeren van toekomsitge verzoeken. De kost die R oploopt is nog steeds een stochastische variabele. De kost die het optimale algoritme en dus de tegenstander oploopt is echter iets moeilijker te specifiëren. Voor het definiëren van de competitiviteit van R kan er gedacht worden alsof de adaptieve tegenstander en het optimale algoritme samenwerken.

Nu zijn er twee mogelijk scenario's die elk aanleiding geven tot een andere soort van adaptive tegenstander. Voor het eerste scenario genereert de tegenstander stap na stap de reeks verzoeken $\rho = (\rho_1, \rho_2, ..., \rho_N)$. Zodra de reeks af is laat de tegenstander het optimale algoritme los op de gekende

reeks (dat is dus het MIN algoritme.) Dit noemen we een adaptieve offline tegenstander. De reeks $\rho = (\rho_1, \rho_2, ..., \rho_N)$ hangt af van het gedrag van R en is dus in wezen een willekeurige reeks. Dus niet alleen $f_R(\rho_1, \rho_2, ..., \rho_N)$ is een stochastische variabele in dit geval, ook $f_O(\rho_1, \rho_2, ..., \rho_N)$ is dat.

In het tweede scenario wordt de reeks op dezelfde manier gegenereerd. In tegenstelling met het voorgaande scenario echter moet de tegenstander op hetzelfde moment een cache online bijhouden. Dus zodra de tegenstander een volgende verzoek genereerd, gebaseerd op de voorgaande keuzes van R moet het zijn antwoord op dat verzoek er meteen bijgeven (De tegenstnder zegt dit antwoord niet tegen R.) Ook hier zijn zowel $f_R(\rho_1, \rho_2, ..., \rho_N)$ en $f_O(\rho_1, \rho_2, ..., \rho_N)$ beide stochastische variabelen. Dit type tegenstadner noemen een adaptieve online tegenstander.

Definitie 6.1.3. Een stochastisch algoritme R is C-competitief tegen een adaptive offline tegenstander als voor elke reeks verzoeken $\rho_1, \rho_2, ..., \rho_N$

$$E[f_R(\rho_1, \rho_2, ..., \rho_N)] - C \times E[f_O(\rho_1, \rho_2, ..., \rho_N)] \leq b$$

waarbij b een constante is die niet afhangt vande lengte van de reeks N, maar wel kan afhangen van de grootte van de cache. De adaptive offline competitiviteitscoëfficiënt, die we noteren als C_R^{aof}, is het infimum van van C zodanig dat R C-competitief is.

Definitie 6.1.4. Een stochastisch algoritme R is C-competitief tegen een adaptive online tegenstander als voor elke reeks verzoeken $\rho_1, \rho_2, ..., \rho_N$

$$E[f_R(\rho_1, \rho_2, ..., \rho_N)] - C \times E[f_O(\rho_1, \rho_2, ..., \rho_N)] \leq b$$

waarbij b een constante is die niet afhangt vande lengte van de reeks N, maar wel kan afhangen van de grootte van de cache. De adaptive online competitiviteitscoëfficiënt, die we noteren als C_R^{aon}, is het infimum van van C zodanig dat R C-competitief is.

Opmerking. Het moge duidelijk zijn dat een adaptive offline tegenstander minstens evengoed is als een adaptive online tegenstander. Deze is dan weer minstens even goed als een onwetende tegenstander. Dus geldt er:

$$C_R^{onw} \leq C_R^{onl} \leq C_R^{ofl}$$

Een Onwetende Tegenstander

We gaan nu onderzoeken wat de ondergrens is voor de competitiviteiscoëfficiënt van een stochastisch algoritme tegen een onwetende tegenstander. Voor het bewijs van de competitiviteiscoëfficiënt van een deterministisch algoritme hebben we gebruik gemaakt van het feit dat de tegenstander de antwoorden van het deterministisch algoritme perfect kon voorspellen. Met een

stochastisch algoritme en een onwetende tegenstander is dit niet meer het geval. De reeks verzoeken wordt in het begin door de tegenstander gekozen en daarna niet meer veranderd. De tegenstander geeft ook de optimale respons voor dezelfde reeks. Dan laten we er het stochastische algoritme op los. Op deze wijze kan de tegenstander onmogelijk de inhoud van de cache van het stochastische algoritme kennen. We voelen intuïtief aan dat dit een hulp gaat zijn voor het stochastische algoritme.

Theorema 6.1.2. *Zij R een stochastish algoritme voor het paging probleem. Dan is $C_R^{onw} \geq H_k$ waarbij H_k het $k$de harmonisch getal is.*

Voor het bewijs maken we gebruik van Yao's Minimax Principe (zie sectie 4.2.2.) Maar eerst gaan we nog wat dingen definëren. Zij $\mathcal{P}$ een kansverdeling voor het kiezen van een reeks verzoeken. De verdeling voor ρ_i mag afhangen van $\rho_1, \rho_2, ..., \rho_{i-1}$. De kost van het algoritme alsook de kost voor het optimale algoritme zijn nu stochastische variabelen. Voor een deterministisch online paging algoritme A definiëren we nu de competitiviteitonder $\mathcal{P}$, $C_A^{\mathcal{P}}$ als het infimum van C zodat

$$E[f_A(\rho_1, \rho_2, ..., \rho_N)] - C \times E[f_O(\rho_1, \rho_2, ..., \rho_N)] \leq b$$

waarbij b zoals steeds een constante is die niet afhangt van de lengte van de reeks maar wel kan afhangen de grootte van de cache. Yao's minimax principe impliceert dat

$$\inf_R C_R^{onw} = \sup_{\mathcal{P}} \inf_A C_A^{\mathcal{P}}$$

Het komt er dus op neer dat de competitiviteit van het beste stochastische online paging algoritme gelijk is aan de competitiviteit van het best mogelijke deterministische algoritme A op een reeks verzoeken die gegenereerd worden volgens $\mathcal{P}$, wat een worst-case verdeling is voor verzoek reeksen ρ. We kunnen dus een ondergrens voor C_R^{onw} berekenen door het geven van een verdeling $\mathcal{P}$ en dan een ondergrens voor $C_A^{\mathcal{P}}$ voor om het even welk deterministisch algoritme A.

Bewijs. We gaan gebruik maken van $k+1$ verschillende geheugen items $I = \{I_1, I_2, ..., I_{k+1}\}$ voor het berekenen van een ondergrens. Daar er k verschillende items in de cache passen moeten we voor elk algoritme alleen maar bijhouden wel item zich niet in de cache bevindt. We nemen aan dat N, de lengte van de reeks van verzoeken veel groter is dan de grote van de cache k.

We gaan Yao's minimax pricipe gebruiken als volgt: we geven een kansverdeling voor het generen van reeksen van verzoeken ρ van lengte N en berekenen het aantal keren dat om het even welk deterministisch algoritme zal missen op deze reeks ρ. De reeks ρ kiezen we als volgt. Het eerste verzoek voor een

item, ρ_1 wordt uniform gekozen uit alle mogelijke geheugen items. Voor volgende verzoeken ρ_i met $i > 1$ kiezen we uniform uit de verzameling $I \backslash \{\rho_{i-1}\}$. We gaan nu aantonen dat het offline algoritme de reeks ρ zo kanverdelen dat het alleen zal missen op het laatse verzoek van een ronde.

De eerste ronde begint met het eerste verzoek en eindigt wanneer elk element vn I minstens één keer is opgevraagt. De tweede ronde begint dan met het volgende verzoek. In het algemeen eindigt een ronde na het verzoek voor het $k+1$ste verschillende item in een ronde. We gebruiken het MIN algoritme; Dit laat het laatste item dat verzocht wordt in de volgende ronde uit de cache op het einde van een ronde. Het item dat uit de cache verwijdert werd zal dan het laatse verzoek zijn in de volgende ronde. Op de manier krijgt MIN altijd één mis per ronde als kost.

Nu moeten we ons echter de vraag stellen hoe lang zal een ronde duren? dit kunnen we als volgt berekenen. Op elk moment in de ronde zijn er al een aantal items opgevraagd. Noem alle items die nog niet zijn opgevraagd nieuw, degene die al eens opgevraagd zijn noemen we oud. Op een gegeven moment zijn er i oude items. Dan is er een kans van $\frac{(k+1)-i}{k}$ dat er een nieuw element gekozen wordt. Nu kunnen er een aantal verzoeken komen die oude items opvragen. We kunnen nu het verwachte aantal verzoeken naar oude items tussen de verzoeken naar nieuwe items bepalen. Zij X_i de stochastische variabele die aangeeft hoeveel items er gevraagd worden na het $i$de nieuwe item tot er een nieuw item gevraagd wordt. Deze X_i zijn dan duidelijk verdeeld volgens de geometrische verdeling met parameter $\frac{(k+1)-i}{k}$. De verwachte duur van een ronde is dan $\sum_i E[X_i]$. Dat is dan

$$\sum_{i=1}^{k} \frac{k}{k+1-i} = k \sum_{i=1}^{k} \frac{1}{k+1-i} = kH_k$$

Neem nu het online algoritme A. A moet op elk moment één item uit haar cache laten. Sinds het volgende verzoek uniform gekozen wordt uit alle geheugen items uitgezonderd datgene dat we net gevraagd hadden heeft A een kans van $\frac{1}{k}$ dat het algoritme A mist. Het verwacht aantal keren dat A zal missen per ronde is dan H_k.

Het offline algoritme mist slechts één keer per ronde. Het online algoritme mist dus H_k keer meer tijdens de uitvoering van de reeks ρ dan het offline algoritme. Door Yao's minimax principe volgt ook het resultaat van theorema 6.1.2. □

We hebben nu wel een ondergrens gevonden maar als elk stochstische algoritme geen betere performance haalt dan een deterministisch algoritme dan geldt deze onder grens ook. We stellen daarom nu een algoritme op dat een performantie heeft die dicht bij dit theoretische optimum ligt. Dit is het zogenaamde Marker algoritme. Elk cache lokatie heeft een marker bit (vandaar de naam.) Dit algoritme verloopt in rondes. Bij het begin van alle

rondes worden de marker bits gereset naar nul. Het algoritme gaat dan als volgt. Als een verzoek reeds in de cache zit zet dan de marker bit van de positie waar dit item zich bevindt op één. Bij een mis kiest het algoritme uniform een cache lokatie waarvan de marker bit nul is en laat het item op die positei uit de cache. De marker bit van die lokatie wordt op één gezet en het item dat de mis veroorzaakte wordt op deze lokatie in de cache geladen. Als alle lokatie hun markeer bit op één hebben staan is de ronde voorbij bij het opvragen van het volgende item dat niet in de cache zit.

Theorema 6.1.3. *Het marker algoritme is* $(2H_k)$*-competitief.*

Bewijs. Om het gemakkelijk te maken zullen we soms referen naar items die gemarkeerd zijn in plaats van de cache lokaties. We zeggendat een item gemarkeerd is als zijn lokatie in de cache gemarkeerd is. We gaan nu het Marker algoritme net als eerder vergelijken met het optimale algoritme over een reeks van verzoeken $rho_1, \rho_2, ...$

We nemen ook weer aan dat beide algoritmes starten met dezelfde items in hun cache en dat het eerste verzoek ρ zich niet in deze cache bevindt. Het marker algoritme verdeelt de uitvoering op natuurlijke wijze in rondes. De eerste ronde begint met het eerste verzoek. De ronde die begint met het verzoek ρ_i eindigt met ρ_j waarbij j het kleinste natuurlijke getal is zodat er $k+1$ verschillende items in de reeks $\rho_i, \rho_{i+1}, ..., \rho_{j+1}$. Alle cache lokaties zijn gemarkeerd op he teinde van een ronde, want er zijn dan k verschillende items opgevraagd en items die tijdens een ronde opgevraagd worden kunnen tijdens diezelfde ronde niet verwijderd worden uit de cache. Het eerste verzoek van een ronde is altijd naar een item dat zich niet in de cache bevindt.

Nu gaan we de verzoeken tijdens een ronde bekijken. We noemen een item gebruikt wanneer het niet gemarkeerd is in deze ronde, maar wel gemarkeerd was op he einde van de vorige ronde. Een item dat gemarkeerd noch gebruikt is noemen we schoon. zij l_i het aantal verzoeken om schone items gedurende de $i$de ronde. We gaan nu uitzoeken hoeveel keren het offline algoritme mist op de uitvoering van een reeks ρ.

Zij S_O de verzameling van items in de cache van het offline algoritme en zij S_M de verzameling van items in de cache vanhet Marker algoritme. Zij $d_I^i = \mid S_O \setminus S_M \mid$ bij het begin van de $i$de ronde en zij d_F^i dezelfde waarde mar dan op het einde van de $i$de ronde. Zij M_O^i het aantal keren dat het offline algoritme mist in de $i$de ronde.

Het is duidelijk dat $M_O^i \geq l_i - d_F^i$ daar er minstens $l_i - d_I^i$ van de l_i propere items waarom verzocht werd in de $i$de ronde zich niet in de cache bevonden bij het begin van deze ronde. Op het einde van een ronde zijn alle k gemerkte items in S_M op dat moment de items die opgevraagd werden gedurende deze ronde. Daar er d_F^i items in de cache van het offline algoritme zitten op he teinde van de $i$de ronde die zich op dat moment niet in de cache

van het Marker algoritme bevinden heeft het offline algoritme minstens d_F^i keren gemist tijdens deze ronde. We bekomen dus

$$M_O^i \geq \max(l_i - d_I^i, d_F^i) \geq \frac{l_i - d_I^i + d_F^i}{2}$$

Zij het aantal rondes in de reeks ρ n dan krijgen we voor M_O het totale aantal keren dat het offline algoritme mist

$$M_O \geq \sum_{i=1}^{n} \frac{l_i - d_I^i + d_F^i}{2} \approx n\frac{l}{2}$$

want $d_I^i = d_F^{i+1}$ dus op de eerste en de laatste na vallen deze weg tegenover elkaar.

Nu het Marker algoritme mist ook een aantal keren per ronde. Sowiso zijn alle l schone items een mis. Van de $k - l$ verzoeken naar een gebruikt item is de verwachte kost de kans dat dit item zich niet in de cache bevindt. De kans daarop wordt eht grootst wanneer we eerst de l schone items krijgen in de reeks verzoeken. De kans dat een gebruikt item zich dan niet langer in de cache bevindt is voor het $i$de gebruikte item dat opgevraagd wordt gelijk aan $\frac{l}{k-i+1}$. Als we dit sommeren over alle i zien we dat deverwachte kost per ronde voor het Marker algoritme gelijk is aan $l + l(H_k - H_l) \leq H_k$. Het totale aantal keren dat Marker mist is dus nlH_k. Het is duidelijk dat Marker $(2H_k)$-competitief is. □

Een Adaptieve Offline Tegenstander

We kunnen blijkbaar door gebruik te maken van een stochastisch algoritme beter resultaten neerzetten tegen een onwetende tegenstander, dan een deterministisch algoritme kan halen. Geldt dit nu ook voor een adaptieve offline tegenstander? Het antwoord daarop zal nee blijken te zijn.

Het probleem is dat de adaptive offline tegenstander weet welke beslissingen het stochastische algoritme neemt. Met deze informatie kan de tegenstander er voor zorgen dat het stochastische algoritme mist bij elk verzoek door telkens dat item in de lijst met verzoeken te plaatsen dat het stochastische algoritme net uit de cache verwijdert heeft. De tegenstander zelf zal pas een verzoek missen nadat er minstens k keer een hit heeft opgetreden. Dit wil dus zeggen dat de competitiviteitscoëfficiënt C^{aof} voor elk stochastisch algoritme minstens k is. Merk op dat de reeks verzoeken nog steeds een stochastische variabele, die door het stochastische algoritme bepaald wordt, is. Een stochastisch algoritme heeft dus geen voordeel tegen een adaptive offline tegenstander in vergelijking met een deterministisch algoritme.

Een Adaptieve Online Tegenstander

We hebben gezien dat tegen een adaptieve offline tegenstander een stochastisch algoritme gen betere resultaten haalt dan een deterministisch al-

goritme. De adaptieve offline tegenstander is gewoon te sterk. Wat zal de score zijn tegen de adaptive online tegenstander? Deze is niet zo sterk als een adaptive offline tegenstander, maar wel sterker dan de onwetende tegenstander.

Een adaptieve online tegenstander weet ook in dit geval wat de keuzes zijn die door het stochastische algoritme gemaakt worden. Op deze wijze kan de adaptive online tegenstander, net als de adaptive offline tegenstander er voor zorgen dat om het even welk stochastisch algoritme altijd zal missen. De vraag is nu hoeveel keer zal de tegenstander missen? Als we aannemen dat beide algoritmes met dezelfde cache beginnen, en het eerste verzoek een item is dat niet in de cache zit, hebben we een natuurlijke keuze voor $k+1$ items. De tegenstander zal slechts verzoeken om items uit deze $k+1$ verschillende items. De beste keuze voor de tegenstander om te verwijderen uit zijn cache als hij mist is dat item dat het stochastisch algoritme met de minste kans uit zijn cache zal halen. Dat is omdat we dat item opvragen dat het stochastische algoritme op de vorige ronde uit zijn cache verwijderd heeft. De kans dat de tegenstander mist is dus gelijk aan de kans dat dat item door het stochastische algoritme uit zijn cache vewijderd wordt. Nu deze kans is ten hoogte $\frac{1}{k}$. De som van alle kansen van de verschillende items in de cache om verwijderd te worden, moet 1 bedragen. Er zijn k verschillende items, dus het minimum van al deze kansen is $\frac{1}{2}$. We verachten dus dat de tegenstander één verzoek per k verzoeken zal missen. Het stochastische algoritem mist echter elk verzoek, dus de competitiviteitscoëfficiënt C^{aon} bedracht voor elk stochastisch algorritme ten minste k. We zien dat stochastische algoritmes niet beter zijn tegen een adaptive online tegenstander dan tegen een adaptive offline tegenstander.

6.2 Oefeingen

Opgave 6.1. *Toon aan de Het MRG algoritme k-competitief is.[7]*

We kunnen makkelijk een reeks van verzoeken opstellen zodat het MRG algoritme zal missen op elk verzoek. We nemen aan dat beide algoritmen beginnen met dezelfde items in hun cache. Het eerste verzoek is om een item niet in deze cache. Zij S de verzameling van de k items in de cahce in het begin plus het item van het eerste verzoek. Alle volgende verzoeken is dat item van S dat zich niet in de cahce van het MRG algoritme bevindt. Op deze manier zal het MRG algoritem elk verzoek missen. Diezelfde reeks voeren we ook aan het MIN algoritme, daar dit algoritme het optimale offline algoritme is. Na de initiële mis op het eerste verzoek, zal dit algoritme geen verzoeken meer missen totdat er k verschillende items opgevraagd zijn. Dit geeft aanleiding tot een opdeling in rondes. De eerste ronde begint met het eerste verzoek. Een ronde eindigt wanneer er na het opvragen van k verschillende items deze ronde er voor de eerste keer weer een nieuw item

wordt opgevraagd. Dat vezoek is het eerste verzoek van de nieuwe ronde. Dat eerste verzoek is het item dat MIN uit zijn cache verwijderd bij het eerste verzoek van de vorige ronde. Op deze wijze krijgt MIN slechts één mis per ronde. Het MGR algoritme houdt een item waarom verzocht is, voor minstens de volgende $k-1$ verzoeken in zijn cache. Daar we telkens dat item opvragen dat net uit zijn cache verwijderd is, zal elke ronde exact k verzoeken lang zijn. We kunnen dus besluiten dat het MRG algoritme k-competitief is en dat zijn competititivietscoëfficiënt exact k is.

Opgave 6.2. *Toon aan de Het FIFO algoritme k-competitief is. [7]*

Ook hier kunnen we makkelijk eens reks van verzoeken opstellen waarop het FIFO algoritme elk verzoek zal missen. We veronderstellen dat beide algoritmen met dezelfde items in hun cache beginnen en dat het eerste verzoek een item opvraagt dat niet in de cache zit. Zo krijgen we de verzameling S die $k+1$ items bevat: de k items in de cache plus het item van het eerste verzoek. We stellen nu een reeks van verzoeken samen op de volgende manier. We vragen telkens dat item van S op dat zich niet in de cahce van het FIFO-algoritme bevindt. FIFO zal op deze manier elk verzoek missen. We delen de reeks verzoeken weer op in rondes. De eerste ronde begint met het eerste verzoek. Een ronde eindigt net voor het $k+1$ste verschillende item sinds het begin van de ronde wordt opgevraagd. Op deze manier zal het MIN algoritme hoogstens één keer missne, namelijk op het eerste verzoek van een ronde. Nu is het zo dat het FIFO algoritme er voor zorgt dat de rondes altijd lengte k hebben. Daaruit kunnen we besluiten dat het FIFO algoritme k-competitief is en dat zijn competititivietscoëfficiënt exact k is.

Opgave 6.3. *Toon aan dat het MFG algoritme niet gebonden is door een competitiviteitscoëfficiënt.*

We kunnen een reeks van verzoeken samenstellen op de volgende manier. We vragen eerst naar alle k items die zich in de cahce van eht MFG algoritem bevinden. Daarna vragen we de $k-1$ eerste items nog een keer op. Daarop volgend vragen we een item dat zich niet in de cahce bevindt. We hebben nu gebruik gemaakt van $k+1$ verschillende geheugen items. Nu vragen we telkens dat item op van die $k+1$ items dat zich niet in de cache bevindt. Door de eigenschappen van het MFG algoritme gaan we nu afwisselen tussen twee items. In het totaal zalals we beginnen met de zelfde items in de cache het optimale offline algoritme, MIN, slechts één mis hebben tijdens de uitvoering van de gehele reeks, terwijl het MFG algoritme alleen hit opde $2k-1$ eerste verzoeken, al de rest zullen missen zijn. Het aantal keren dat het MFG algoritme mist zal dus afhangen van de lengte van de reeksverzoeken, terwijl het MIN algoritme altijd maar één keer zal missen. Op deze manier kunnen we geen competitiviteitscoëfficiënt vinden volgens de definitie.

Opgave 6.4. *Gegeven een ongerichte graaf G, dan is een boogkleuring een toekenning van indices $1, 2, ..., C$ aan de bogen van G zodanig dat geen twee*

bogen die toekomen in dezelfde knoop dezelfde index hebben. De indices noemen we kleuren en de kleinste waarde voor C waarmee de graaf G gekleurd kan worden noemen we de chromatische index van G. Een graaf met een maximale graad Δ heeft volgens Vizing's theorema een chromatische index van Δ of $\Delta + 1$. Nu gaan we echter kijken naar online boogkleuring. Veronderstel dat de bogen van de graaf G met een maximale graad Δ één per één aan het algoritem gegeven worden. Zodra een boog aan het algoritme getoond wordt, moet deze boog een kleur toegekend krijgen. Deze kleur ligt daarna onherroepelijk vast, hij kan niet meer veranderd worden.[1]

> *a) Stel een deterministisch online algorimte op dat hoogstens $2\Delta - 1$ kleuren gebruikt om de graaf te kleuren.*
>
> *b) Toon aan dat er geen enkel deterministisch online algoritme bestaat dat minder dan $2\Delta - 1$ kleuren gebruikt in de worst case.*

a) Maak een circulaire array die alle kleuren bevat. Op deze manier kunnen we steeds verder gaan in de array en zmoeten we geen rekening houden met het einde van de array, deze zal gewoon automatisch terug bij zijn begin verder gaan. De eerste boog die getoond wordt, krijgt de eerste kleur toegewezen. De tweede boog de tweede enzoverder. Als we een boog krijgen die aan een knoop hangt waar al andere bogen naartoe lopen dan kennen we de eerste kleur die we tegenkomen en die nog niet toegekent is aan één van de bogen van die knoop, toe aan deze boog. We gebruiken op deze manier hoogstens $2\Delta - 1$ kleuren. Dat gebeurt in het volgende geval. We hebben een knoop met daaraan reeds $\Delta - 1$ gekleurde bogen. We hebben een ander knoop, ook met $\Delta - 1$ gekleurde bogen, maar al deze kleuren zijn verschillend van de kleuren van de eerste knoop. Als we nu een boog onthullen die deze twee knopen verbindt is er slechts één mogelijke kleur die deze boog kan krijgen. Ook kan elke configuratie van de gekleurd worden, daar voorgaande configuratie deze is die de meeste kleuren gebruikt.

b) Er is geen deterministisch online algoritme dat toekomt met minder dan $2\Delta - 1$ kleuren. We kunnen dit als volgt zien. Laat het algoritme voldoende "sterren", knopen met $\Delta - 1$ bogen inkleuren. Met voldoende Sterren zullen er Δ sterren zijn die op dezelfde manier ingekleurd zijn. Verbind het centrum van deze sterren nu met een tot nu toe lege knoop. Elk ster heeft dan een boog naar deze knoop en de kleur moet verschillend zijn voor elke ster. Er zijn Δ van dit soort sterren dus zijn er Δ verschillende kleuren nodig. Daarbij komen ook nog de $\Delta - 1$ kleuren die we al gebruikt hebben om de sterren in te kleuren dus in het totaal zijn er altijd $2\Delta - 1$ kleuren nodig.

Opgave 6.5. *Het random algoritme voor het paging probleem werkt alsvolgt: zodra er een verzoek is om een element dat zich niet in de cache bevindt wordt er willekeurig volgens een uniforme verdeling een element uit de cahce*

verwijderd. Het gevraagde element wordt dan op de vrijgekomen plaats in de cahce geladen. Toon nu aan dat het random algoritme een competitiviteitscoëfficiënt heeft van ten minste kH_k tegen een adpative offline tegenstander.

We voronderstellen dat beide algoritmes beginnen met dezelfde items in hun cahcec en dat eh teerste verzoek is naar een item niet in deze cache. Doordat de adaptive offline tegenstander weet welk geheugen element het random algoritme uit zijn cache gooit, kan de adaptive tegenstander ervoor zorgen dat dit algoritme altijd zal missen. De reeks die op deze manier onstaat, is gelijk aan het volgende. We kiezen op uniforme wijze steeds een item uit de k items die in de cahce van het random algoritme zatten voor de uitvoering van eht huidige verzoek. In essentie werken we dus maar met $k + 1$ geheugen items. Ook zullen geen twee verzoeken na elkaar naar het zelfde item voorkomen. In het bewijs van theorema 6.1.2 hebben we gezien dat op een dergelijke reeks het min algoritme verwacht om een keer te missen op kH_k verzoeken. Hieruit volgt duidelijk dat de competitiviteitscoëfficiënt voor het random algoritme tegen adaptive offline tegenstanders gelijk is aan kH_k.

Bibliografie

[1] A. Bar-Noy, R. Motwani, and J. Naor. The greedy algorithm is optimal for online edge coloring. *Information Processing Letters*, 44:251–253, 1992.

[2] H. Callaert. *Statestiek en Kanstheorie*. Limburgs Universitair Centrum, 2000.

[3] Donald E. Knuth and Andrew C. Yao. The complexity of nonuniform random number generation. 1976.

[4] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambride University Press, 1995.

[5] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[6] Michael Saks and Avi Wigderson. Probabilistic boolean trees and the complexity of evaluating game trees. In *Proceedings of the 27th FOCS*, pages 29–38, 1986.

[7] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985.

[8] Noël Veraverbeke. *Kanstheorie en Statistiek*. Limburgs Universitair Centrum, 2000.

[9] John von Neumann. Various techniques used in connection with random digits. *J. Research Nat. Bur. Stand., Appl. Math. Series*, 12(??):36–38, 1951.

Auteursrechterlijke overeenkomst

Opdat de Universiteit Hasselt uw eindverhandeling wereldwijd kan reproduceren, vertalen en distribueren is uw akkoord voor deze overeenkomst noodzakelijk. Gelieve de tijd te nemen om deze overeenkomst door te nemen en uw akkoord te verlenen.

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Stochastische algoritmen
Richting: **Licentiaat in de informatica** Jaar: **2006**
in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Deze toekenning van het auteursrecht aan de Universiteit Hasselt houdt in dat ik/wij als auteur de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij kan reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

U bevestigt dat de eindverhandeling uw origineel werk is, en dat u het recht heeft om de rechten te verlenen die in deze overeenkomst worden beschreven. U verklaart tevens dat de eindverhandeling, naar uw weten, het auteursrecht van anderen niet overtreedt.

U verklaart tevens dat u voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen hebt verkregen zodat u deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal u als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze licentie

Ik ga akkoord,

Tony BLEYS

Datum: