

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

MC SYLLABUS 48.1

**COLLOQUIUM
COMPLEXITEIT EN ALGORITMEN**

DEEL 1

P.M.B. VITÁNYI (red.)

J. van LEEUWEN (red.)

P. van EMDE BOAS (red.)

MATHEMATISCH CENTRUM AMSTERDAM 1982

1980 Mathematics subject classification: 68C05, 68C25, 68C40, 68H05

ACM-Computing Reviews-category: 5.25, 5.26, 4.34

ISBN 90 6196 237 4

INHOUD

Inhoud	1
Adressen van auteurs	v
Voorwoord	vii

COMPLEXITEIT EN ALGORITMEN, DEEL 1.

A. INLEIDING IN DE COMPLEXITEITSLEER

I. COMPUTERS EN (ON)DOENLIJKE PROBLEMEN,	J. van Leeuwen
1. Algoritmen	3
2. Ontwerpen	6
3. Wat heet doenlijk	12
4. Doenlijk, maar toch te doen?	19
5. Referenties	21

II. BEREKENINGSMODEL EN COMPLEXITEIT,	P.M.B. Vitányi
1. Inleiding	27
2. Seriële berekeningsmodellen	30
2.1. De registerautomaat	31
2.2. P versus NP	37
2.3. Vermenigvuldigings RAMs	39
2.4. Abstracties van RAMs	41
1) Rechthoekige programma's	41
2) Bitgewijze berekeningen	42
3) Beslissingsbomen	43
2.5. De Turing machine	44
2.6. Onachtzaamheid en echte tijd	48
3. Parallelle berekeningsmodellen	51
3.1. Parallelle RAMs	51
3.2. VLSI berekeningsmodellen	53
4. Referenties	61

B. DATASTRUCTUREN

III. BALANCED TREES AS A DATASTRUCTURE FOR

REPRESENTING SORTED LISTS,

K. Mehlhorn

1. Introduction	69
2. Search trees	70
3. Some non-standard applications of balanced trees	77
4. A sequence of operations analysis of (a,b)-trees, $b \geq 2a$	79
5. A representation for linear lists with fingers	86
6. Applications of level-linked (a,b)-trees	88
7. Conclusion	89
8. References	90

IV. VERZAMELINGSMANIPULATIE OP EEN KLEINE COMPUTER, Th.P. van der Weide

0. Inleiding	93
1. Sorteren en selecteren	95
1.1. Sorteren	96
1.2. Selectie	96
1.2.1. On-line selectie	97
1.2.2. Selectie in klein werkgeheugen	100
1.2.3. Probabilistische methodes	103
2. Verzamelingsmanipulaties	104
3. Referenties	107

V. DYNAMISCHE ZOEKSTRUCTUREN DIE HUN GESCHIEDENIS

ONTHOUDEN

M.H. Overmars

1. Zoekproblemen	109
2. Statische zoekstructuren	113
3. Dynamische zoekstructuren	118
4. Zoeken in het verleden	126
5. Referenties	133

C. PARALLELE BEREKENINGEN EN VLSI

VI. VLSI EN DE COMPLEXITEIT VAN BEREKENINGEN

M. Rem

0. Inleiding	139
1. MOS-chips	140
2. CMOS	142
3. Tijd	145
4. Ruimte	150
5. Complexiteitsmaten	152
6. Referenties	153

VII. A COMBINATORIAL LIMIT TO THE COMPUTING POWER
OF VLSI CIRCUITS

J. Vuillemin

1. Introduction	155
2. Transitive Boolean functions	157
2.1. Transitivity of functions in the problem list	158
2.2. On the extend of the class of transitive functions	160
3. The computational model	160
3.1. Technological assumptions	161
3.1.1. Size of elementary devices	163
3.1.2. Speed of elementary devices	163
3.2. Logical assumptions	163
3.3. Global circuit measures	164
4. Area versus data rate	165
4.1. On the number of I/O ports	165
4.2. On the number of gates	166
4.3. On the area occupied by wires	167
4.4. On the total circuit area	171
5. Conclusion	172
5.1. Relations between VLSI and other models of computation	172
5.2. On detailed mathematical analysis of VLSI circuits	173
5.3. On the computational model	173
6. References	174

VIII. BOOMMACHINES EN VERKAVELDE BEREKENINGEN,	F.J. Peters
0. Inleiding	179
1. Doelmatigheid van boommachines	179
2. Voorbeelden	181
2.1. Recursive doubling	181
2.2. Meerdimensionale verdeel- en heers-algoritmen	183
2.3. Eindige-elementen berekeningen	186
3. Realisatie	188
4. Slotopmerkingen	190
5. Referenties	191

COMPLEXITEIT EN ALGORITMEN, DEEL 2.

D. ALGEBRAÏSCHE EN ANALYTISCHE COMPLEXITEIT	
IX BEREKENINGSCOMPLEXITEIT VAN BILINEAIRE EN KWADRATISCHE VORMEN,	P. van Emde Boas
X PRIMALITEIT EN FACTORIZATIE,	H.W. Lenstra jr.
XI EFFICIËNTIE VERSUS NAUWKEURIGHEID,	C.G. van der Laan
E. BESLISKUNDE	
XII KHACHIAN'S ELLIPSOÏDE-METHODE VOOR LINEAIRE PROGRAMMERING,	A. Schrijver
XIII EEN GEAUTOMATISEERDE COMPLEXITEITSCLASSIFICATIE VAN COMBINATORISCHE PROBLEMEN,	B.J. Lageweg, E.L. Lawler, J.K. Lenstra & A.H.G. Rinnooy Kan

ADRESSEN VAN AUTEURS

- J. van Leeuwen Rijksuniversiteit Utrecht, Vakgroep Informatica,
Princetonplein 5, Postbus 80.002, 3508 TA Utrecht.
- K. Mehlhorn Fachbereich 10 der Universität des Saarlandes,
Angewandte Mathematik und Informatik,
D-6600 Saarbrücken, West-Duitsland.
- M.H. Overmars Rijksuniversiteit Utrecht, Vakgroep Informatica,
Princetonplein 5, Postbus 80.002, 3508 TA Utrecht.
- F.J. Peters Technische Hogeschool Eindhoven, Onderafdeling
der Wiskunde, Postbus 513, 5600 MB Eindhoven.
- M. Rem Technische Hogeschool Eindhoven, Onderafdeling
der Wiskunde, Postbus 513, 5600 MB Eindhoven.
- P.M.B. Vitányi Mathematisch Centrum, Kruislaan 413,
1098 SJ Amsterdam.
- J. Vuillemin Laboratoire de Recherche en Informatique,
Université de Paris-Sud, 91405 Orsay, France.
- Th.P. van der Weide Daisy Systems Holland B.V., 6600 AC Wijchen.*)

*) Huidig adres: Zonnenberg Computerservice B.V., Oss.

VOORWOORD

In het academisch jaar 1980/1981 werden op het Mathematisch Centrum een dertiental voordrachten gehouden over *Complexiteit en Algoritmen*. Bij de voordrachten werden syllabi uitgereikt. Deze bundel bevat de door de auteurs bijgewerkte en gecorrigeerde versies daarvan. De bijdragen zijn naar onderwerp gebundeld in vijf groepen verdeeld over twee delen, waarvan dit het eerste is. In plaats van op de verschillende onderwerpen in te gaan laten wij de inhoudsopgave voor zichzelf spreken en wijden dit voorwoord aan een korte inleiding tot het onderhavige vakgebied.

Wie rekent zal gewoonlijk een bepaald algoritme volgen. Door preciese analyse kan men soms een uitspraak doen over het aantal handelingen (bewerkingen, assembler instructies) dat nodig is om een probleem van formaat "n" volgens het gekozen algoritme tot een oplossing te brengen. Tesamen met overwegingen inzake de vereiste I/O transporten, verkrijgt men zo een redelijke indruk van de te verwachten rekentijd (en/of geheugengebruik) van een algoritme op een realistisch machinemodel. Zo'n analyse is vaak nodig om tot een duidelijk onderscheid te komen tussen efficiënte en niet zo efficiënte algoritmen voor eenzelfde probleem.

Sinds de vijftiger jaren is men zich er meer en meer van bewust geworden dat een nauwkeurige evaluatie van de efficiëntie van gebruikte rekenmethodes doorslaggevend kan zijn voor het welslagen van te maken software. Een steeds terugkerende ervaring is echter dat er, hoe slim men ook programmeert, soms geen noemenswaardige vooruitgang in efficiëntie te behalen lijkt in de algoritmische oplossing van een probleem. We schijnen dan op een intrinsieke *complexiteit* van het probleem te stuiten, die ieder denkbaar algoritme dwingt tot het uitvoeren van een zeker minimum aantal bewerkingen. Deze indruk kan onjuist zijn, maar alleen door preciese analyse kan men erachter komen of er geen betere oplossingen bestaan. De voortdurende pogingen om steeds maar weer algoritmen te vinden die hun doel met mindere bewerkingen dan tot nog toe bekend bereiken, zijn een methode om de complexiteit van problemen te bepalen en om uiteindelijk tot praktisch efficiënte rekenmethodes te komen. Wat vroeger misschien een slimme programmeertruuk leek, blijkt dan niet zelden een algemene techniek te zijn om goede algoritmen toch weer te versnellen. Zo kent men thans tal van technieken (zoals depth-first search, path compression, Fast Fourier Transform, dynamization) die, hoewel alleen door de theorie op hun juiste waarde getaxeerd, in praktisch

programmeerwerk hun diensten kunnen bewijzen. En nieuwe berekeningsmodellen (zoals de hardware realisatie van algoritmen op chips) geven steeds weer nieuwe uitgangspunten voor een onderzoek van de wezenlijk haalbare efficiëntie van algoritmen.

In het colloquium is een poging gedaan om de vele facetten van *complexiteit en algoritmen* wat grotere bekendheid te geven. In een aantal voordrachten werden door de sprekers vooral recente vorderingen in dit gebied geëxposeerd, al dan niet met kritische kanttekeningen over hun betekenis zoals die thans gezien wordt. De bedoeling is dat dit zal bijdragen tot een duidelijk beeld van de mogelijkheden en beperkingen van algoritmische analyse voor de ontwerper van rekenmethoden.

Amsterdam, december 1981.

A. INLEIDING IN DE COMPLEXITEITSLEER

COMPUTERS EN (ON)DOENLIJKE PROBLEMEN

J. VAN LEEUWEN

1. ALGORITMEN

Voor tal van praktische vraagstukken is het niet voldoende te weten dat een oplossing bestaat, maar moet men een oplossing ook daadwerkelijk kunnen uitrekenen. Een algoritme is een duidelijke specificatie van de stappen (instructies) die men in een berekening dient uit te voeren om het gewenste "antwoord" te verkrijgen. Men kan algoritmen met veel woorden omschrijven of in een geschikte programmeertaal uitdrukken, afhankelijk van de beoogde toepassing. In ieder geval moeten de individuele instructies tot het elementaire repertoire behoren van de machine waarop men de algoritmen uitvoeren wil.

Het is om allerlei redenen gewenst algoritmen te meten. Zo is de uitvoering van een algoritme bijvoorbeeld zelden kostenloos en wil men van tevoren een indruk hebben van de benodigde hoeveelheid tijd en/of geheugenruimte (of zelfs van het benodigde aantal processoren). Het is echt niet altijd zo dat het goedkoopste algoritme voor een vraagstuk ook het "beste" is, omdat andere eigenschappen (zoals de mate van numerieke stabiliteit) wel eens minder gunstig kunnen uitpakken. Niettemin zijn precieze uitspraken en schattingen over de duurte van algoritmen een belangrijk en vaak doorslaggevend gegeven, indien men uit verschillende algoritmen voor een vraagstuk kiezen wil. Alleen door analyse kan men er achter komen in welke mate een zeker algoritme beslag legt op de middelen (resources) van een computer.

Het is gebruikelijk om de duurte van een algoritme uit te drukken in termen van parameters die bepalend zijn voor de grootte van de input. Zo kan men bijvoorbeeld van een algoritme zeggen dat het n getallen in $O(n \log n)$ stappen sorteert. (Men kan dergelijke algoritmen vinden in b.v. KNUTH [27] of LORIN [32].) Het komt ook voor dat men alleen bewerkingen van een bepaalde soort telt. Zo is er een algoritme dat een n^e graads polynoom $p(x)$ en al zijn genormaliseerde afgeleiden $p^{(i)}(x)/i!$ (voor

$1 \leq i \leq n$) kan evalueren in $2,5 n$ vermenigvuldigingen en delingen (DE JONG & VAN LEEUWEN [15]). Maar dat algoritme gebruikt nog wel $O(n^2)$ optellingen en aftrekkingen.

Men moet uitspraken over algoritmen vaak zorgvuldig lezen. Zo zegt men bijvoorbeeld snel dat het gewone algoritme voor de vermenigvuldiging van twee $n \times n$ matrices $O(n^3)$ stappen vergt. Een stap is dan een optelling, een vermenigvuldiging of het opvragen/opslaan van een waarde uit/in een geheugenplaats. In de analyse van andere algoritmen gebruikt men misschien een ander stapbegrip. Het is dus van belang te weten welk berekeningsmodel gehanteerd wordt, dus wat als "enkele stap" mag doorgaan. We houden het in het vervolg op het veelal gebruikte model van de random access machine (zie b.v. AHO, HOPCROFT & ULLMAN [1] of VITÁNYI [63]), dat in wezen een eenvoudige processor en een oneindig random access geheugen biedt. Er ligt nog een heel terrein van onderzoek open om uitspraken over algoritmen te krijgen waarin op realistische wijze rekening wordt gehouden met de doorgaans optredende overhead door I/O transporten op echte hardware.

De analyse van algoritmen heeft zich in de laatste jaren in versterkte mate gemanifesteerd, wellicht omdat men op veel ruimere schaal computers gebruikt en efficiëntere algoritmen verlangt om economische redenen. Zo worden regelmatig snellere algoritmen voor bestaande vraagstukken geponeerd, al wordt de "winst" soms pas behaald voorbij elke praktische waarde van n . De betekenis van dergelijke resultaten ligt dan in een iets andere richting, maar daarover later.

We bezien het testen van $A \subseteq B$ als voorbeeld van de afweging van mogelijke algoritmen door analyse (ontleend aan BENTLEY [2]). Stel dat twee ongeordende verzamelingen A (m elementen) en B (n elementen) als arrays gegeven zijn. We willen verifiëren of elk element van A in B voorkomt. Neem aan dat $m \leq n$.

Het naieve algoritme zal eruit bestaan dat voor elk element x van A de verzameling B sequentiëel doorlopen wordt tot x (al of niet) gevonden is. Men mag aannemen dat voor het eenmaal doorzoeken van B gemiddeld $n/2$ stappen nodig zijn, zodat de totale duurte van het naieve algoritme op ongeveer $m \cdot n/2$ stappen uitkomt. Men kan het steeds opnieuw doorzoeken van B voorkomen door A en B eerst te sorteren. Dat kost weliswaar zo'n $m \log m + n \log n$ stappen, maar daarna kan men volstaan met het eenmaal gelijktijdig doorlopen van de beide arrays. Dit tweede algoritme vergt dus ongeveer $m \log m + n \log n + m + n$ stappen. Vervolgens kan men bedenken dat het sorteren van B al voldoende was om het zoeken erin door gebruik van binary search tot

zo'n $\log n/2$ per keer te versnellen. Het hierop gebaseerde algoritme vergt zo'n $n \log n + m \log n/2$ stappen. Als $m \ll n$ dan kan het slimmer zijn niet B maar A te sorteren! Test elk element van B op A met binary search en vlag elk element (in A) dat raak is. Aan het eind dient men louter nog te verifiëren dat elk element van A gevlagd is. Het aldus besloten derde algoritme vergt zo'n $m \log m + n \log m/2 + m$ stappen.

Het hangt kennelijk van de grootte van $n \log m/2$ in vergelijking tot $n \log n + n$ af of het derde algoritme wezenlijk sneller is dan het tweede. Er is echter nog een vierde algoritme denkbaar, gebaseerd op de techniek van hashing (KNUTH [27]). Men dient dan wel over de nodige geheugenruimte te beschikken voor een hash-table, waarin de elementen van (zeg) B zonder groot gevaar van clustering ingevoegd kunnen worden. Theorie leert dat zo'n table minstens ongeveer $1,5n$ posities lang moet zijn om met een geschikte hash-functie gemiddeld hoogstens 2 collisions per invoeging te mogen verwachten. Het hashen van B en vervolgens, ook met hashing, opzoeken van elke x uit A zal naar alle waarschijnlijkheid dus niet meer dan ongeveer $2n + 2m$ stappen vergen, de snelste manier tot nog toe. Maar we hebben wel heel wat extra geheugenruimte nodig. (Als $m \ll n$, dan moet men natuurlijk niet een hash-table voor B maar voor A opzetten.)

In de praktijk lukt het vaak niet om een sneller algoritme voor een vraagstuk te vinden en lijkt er geen noemenswaardige vooruitgang meer te behalen. We schijnen dan op een intrinsieke complexiteit van het vraagstuk te stuiten, die ieder algoritme dwingt tot het uitvoeren van een zeker minimum aantal bewerkingen. Alleen door analyse van de diepere structuur van het vraagstuk kan men er achter komen of het ons alleen aan slimheid heeft ontbroken óf binnen het gehanteerde berekeningsmodel inderdaad geen efficiëntere algoritmen bestaan. Zo is het bekend dat in een model waarin elementen alleen "vergeleken" mogen worden elk algoritme voor het testen van $A \subseteq B$ in voorkomende gevallen zeker $\Omega(n \log n)$ vergelijkingen maken moet (REINGOLD [40]). In die zin kan men " $n \log n$ " de "complexiteit" van het set-inclusie vraagstuk noemen. Merk op dat het algoritme gebaseerd op hashing buiten het kader van dit berekeningsmodel valt.

Er zijn maar weinig vraagstukken waarvan de algoritmische complexiteit precies bekend is. Zelfs goede ondergrenzen ontbreken vaak, wegens onvoldoende inzicht in de wiskundige structuur die in vele vraagstukken verschoonen ligt. Zo ligt er nog een grote kloof tussen de bewezen ondergrens van ongeveer $2n^2$ vermenigvuldigingen aan $n \times n$ matrix multiplikatie (b.v. VAN LEEUWEN & VAN EMDE BOAS [61]) en het recent gevonden $O(n^{2,51...})$ algoritme

voor deze taak (SCHÖNHAGE et al. [48]), dat het wellicht bekendere $O(n^{2,80..})$ algoritme (STRASSEN [54]) inmiddels heeft verslagen. De voortdurende pogingen om steeds maar weer algoritmen te vinden die een antwoord met minder stappen dan tot nu toe bekend bereiken hebben zo het tweeledig doel gekregen om zowel economischer rekenmethoden te leveren voor de praktijk als "van boven af" de intrinsieke complexiteit van vraagstukken te benaderen. De dubbele rol die hieruit spreekt voor algoritmen pleegt nogal eens verwarrend te zijn voor de louter praktisch ingestelden.

De moderne paradigmas in het ontwerpen en analyseren van algoritmen vindt men in AHO, HOPCROFT & ULLMAN [1] of in de leesbare surveys van bijvoorbeeld HOPCROFT [18], MEYER [37], RABIN [44] en TARJAN [55]. De navolgende notities pogen een indruk te geven van enkele fundamentele themas in de wereld van complexiteit en algoritmen.

2. ONTWERPEN

Een belangrijk motief in de systematische studie van efficiënte algoritmen is de voortdurende ontdekking van in wezen probleem-onafhankelijke methoden en benaderingen. Wat misschien eens voor zomaar een programmeertruuk doorging, is niet zelden een algemene techniek gebleken die ook in veel andere problemen toepasbaar is en tot snellere algoritmen voert. Een ontwerper (programmeur) zal met een redelijk aantal van die algoritmische technieken vertrouwd moeten zijn.

Het is bekend dat de efficiëncy van een algoritme in aanzienlijke mate afhankelijk is van de gebruikte datastructuren voor gegevens. Vertrouwdheid met de vele mogelijkheden van linked lists is vereist (zie b.v. HOROWITZ & SAHNI [19] of STANDISH [52]). Men heeft geleerd om bij het ontwerpen niet meteen de datastructuren te fixeren, maar eerst de fundamentele operaties te ontdekken die men op de data wil uitvoeren. Deze benadering middels "abstrakte datastructuren" stelt ons in staat de efficiënte implementatie redelijk los van andere zaken te besluiten. Zo zijn er nogal wat problemen met eindige verzamelingen V waarop men in wezen de volgende bewerkingen moet kunnen uitvoeren:

INSERT (x) : voeg x toe aan V

DELETE (x) : laat x weg uit V

SEARCH (x) : bezoek x in V (zo x element van V is).

Door gebruik te maken van een geschikt type van gebalanceerde zoekbomen (b.v. [1]) toont men

STELLING 2.1. *Er is een datastructuur voor V zodat INSERT, DELETE en SEARCH instructies nooit meer dan $O(\log n)$ stappen vergen ($n = \# V$).*

Als men sommige elementen van V vaker bezoekt dan andere, dan zal men een zoekboom wensen waarin de gemiddelde zoektijd minimaal is. MEHLHORN [36] heeft getoond hoe men in die zin optimale zoekbomen ook dynamisch bijna optimaal kan houden.

Bij ontwerpen stelt men verzamelingen meestal maar meteen door gebalanceerde bomen voor "omdat het daar toch altijd op uit draait". Maar stel dat we een soort lijst willen hebben waarop de volgende instructies zijn uit te voeren:

INSERT (k, x) : voeg x in juist vóór het k^e element

DELETE (k) : laat het k^e element weg

SEARCH (k) : bezoek het k^e element.

(k willekeurig). Met een gewone gebalanceerde zoekboom komt men aan $O(\log n)$ per instructie, met $n = \# V$. Men kan echter bewijzen (zie b.v. CLANCY & KNUTH [13])

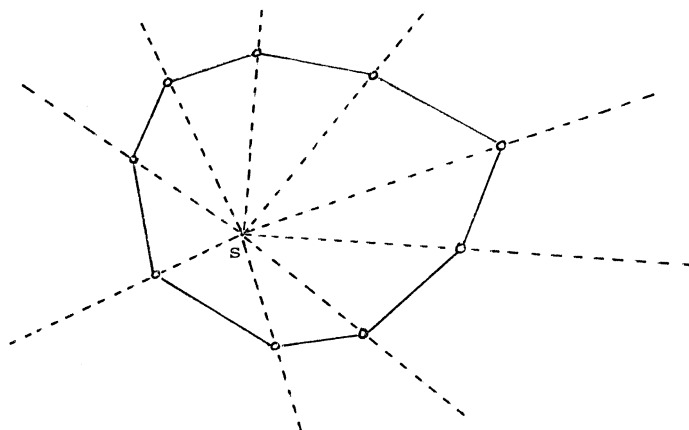
STELLING 2.2. *Er is een datastructuur voor V zodat een INSERT (k, x), DELETE (k) of SEARCH (k) instructie nooit meer dan $O(\log k)$ stappen vergt, onafhankelijk van de grootte van V .*

Vervolgens is het bekend dat veel efficiënte algoritmen uiteindelijk zijn verkregen na een verder onderzoek van de structuur van een probleem. Men heeft een aantal technieken leren onderscheiden die, mits een probleem zich leent voor toepassing, vaak tot extra tijdwinst voeren. We noemen en illustreren enkele van deze algoritmische technieken.

Een eerste techniek is *binary search*, vooral bekend geworden om in $O(\log n)$ stappen een specifieke waarde in een gesorteerd n -plaatsig array op te zoeken. (Om binary search op lijsten toe te passen, moet men weer van gebalanceerde zoekbomen gebruik maken.) Maar binary search is in feite altijd toepasbaar, als men maar een geschikte zoekruimte in een probleem kan onderkennen. Het volgende voorbeeld is ontleend aan SHAMOS [51].

STELLING 2.3. Men kan een konvexe n -hoek zodanig representeren dat voor elk willekeurig punt p in het vlak in slechts $O(\log n)$ stappen beslist kan worden of p in het binnen- of in het buitengebied van de n -hoek ligt.

BEWIJS. Kies een punt s in het binnengebied van de konvexe n -hoek en trek rechten vanuit s door alle hoekpunten (zie figuur).



Op deze wijze wordt het vlak in sectoren verdeeld, die op ruimtehoek bij s kunnen worden opgeslagen. Gegeven een punt p , dan kan men met binary search in $O(\log n)$ stappen de sektor bepalen waartoe p behoort. Men hoeft dan alleen nog maar de positie van p ten opzichte van de in de sektor aanwezige randlijn te bepalen om het gewenste antwoord te krijgen. $\square$

Met wat moeite kan men de stelling tot willekeurige, enkelvoudige n -hoeken uitbreiden (zie [51]). KIRKPATRICK [25] verkreeg onlangs een eenvoudig algoritme om voor elke planaire vlakverdeling met n knooppunten in maar $O(\log n)$ stappen te bepalen tot welk gebied van de verdeling een punt behoort.

Een tweede techniek is *divide-and-conquer*. Hieronder verstaat men de verdeling (splitsing) van een probleem in twee of meer deelproblemen van kleiner formaat, waarvan de oplossingen eenvoudig samengesteld kunnen worden tot een oplossing van het oorspronkelijke probleem. Men komt typisch op rekursieve algoritmen uit. Diverse sorteer-algoritmen zoals merge-sort en quicksort (zie [1]) berusten op *divide-and-conquer*. Ook het volgende resultaat van BLUM et al. [6] berust op een slimme toepassing van deze strategie.

STELLING 2.4. Voor elke t kan het op t na grootste element van een ongeordende verzameling V in $O(n)$ stappen gevonden worden ($n = \# V$).

Het volgende voorbeeld is ontleend aan BENTLEY & SHAMOS [4].

STELLING 2.5. Het konvex omhulsel van een verzameling van n punten in het vlak is te berekenen in $O(n \log n)$ stappen.

BEWIJS. Verdeel de punten over twee deelverzamelingen van $n/2$ punten elk en bepaal het konvex omhulsel van de beide deelverzamelingen door rekursieve toepassing van het algoritme. Men kan bewijzen dat de resulterende konvexe i - en j -hoeken ($i, j \leq n/2$) in $O(n)$ stappen tot het konvex omhulsel van de gehele verzameling zijn samen te stellen. Is $T(n)$ het benodigd aantal stappen van het algoritme op een verzameling van n punten, dan geldt voor zekere c :

$$T(n) \leq 2T(n/2) + c.n.$$

Er volgt dat $T(n) = O(n \log n)$. $\square$

YAO [65] bewees dat de stelling in redelijke zin optimaal is, PREPARATA & HONG [42] bewezen dat zij ook in drie dimensies doorgaat.

Een derde techniek is *dynamic programming*. Dit betreft strategieën die, om tot de konstruktie van een oplossing te kunnen overgaan, in feite eerst een "rap" kwalitatief onderzoek moeten plegen van alle deelproblemen van kleiner formaat. Het $O(n^3)$ algoritme van MURAOKA & KUCK [38] voor bepaling van de optimale evaluatie volgorde van een produkt van n matrices van willekeurige formaten is een typisch voorbeeld van dynamic programming. Om de n^3 -grens te verklaren die in deze problemen nogal eens gevonden wordt, bedenke men dat dynamic programming nogal eens neerkomt op het "invullen" van een $n \times n$ tabel c waarvoor

$$\begin{aligned} c[i,i] &= 0 \\ c[i,j] &= w(i,j) + \min_{i < k \leq j} (c[i,k-1] + c[k,j]) \text{ voor } i < j \end{aligned}$$

met een of andere goedkope increment funktie w . Stel eens dat w voldoet aan de volgende vierhoeksongelijkheid

$$w(i,j) + w(i',j') \leq w(i',j) + w(i,j') \text{ voor } i \leq i' \leq j \leq j'.$$

YAO [66] heeft getoond dat in dat geval dynamic programming versneld kan worden, want zij bewees

STELLING 2.6. *Indien w aan de voornoemde vierhoeksongelijkheid voldoet en bovendien interval-monotoon is (d.w.z. dat $w(i,j) \leq w(i',j')$ als $i' \leq i \leq j \leq j'$) dan kan men c in $O(n^2)$ stappen invullen.*

Het bekende $O(n^2)$ algoritme voor konstruktie van optimale zoekbomen (KNUTH [26]) is uiteindelijk een vrij direkt gevolg van deze stelling gebleken.

Een vierde techniek is *dynamisatie*. Daarbij beperkt men zich aanvanke-lijk tot het vinden van een efficiënte statische datastructuur voor een pro-bleem en poogt men vervolgens daaruit langs direkte weg een efficiënte dy-namische oplossing te maken. Soms lukt dat met lokaal bijwerken (zoals in AVL en 2-3 bomen [1]) of door het geregeld totaal herbouwen van uit balans geraakte delen van de datastructuur (zoals in b.v. OVERMARS & VAN LEEUWEN [39]). Een recente benadering is een dynamisch systeem van statische componenten bij te houden. Dat vereist dat een zoekprobleem in zekere zin "ontbindbaar" is (zie BENTLEY [3]).

DEFINITIE. Een zoekprobleem $Q(V)$ op verzamelingen V heet ontbindbaar indien voor elke partitie $V_1 \cup V_2$ van V geldt dat

$$Q(V) = \square(Q(V_1), Q(V_2))$$

met een eenvoudige bewerking $\square$.

Zo is het naaste-buur-probleem, dat voor een willekeurig punt vraagt het dichtstbijgelegen element van V te vinden, een ontbindbaar zoekprobleem. Stel dat de statische versie van een ontbindbaar zoekprobleem op n punten in $Q(n)$ stappen op te lossen is, door gebruik van een datastructuur die $B(n)$ stappen kostte om te bouwen. Onder milde aannamen over Q en B volgt uit VAN LEEUWEN & WOOD [62] bijvoorbeeld:

STELLING 2.7. *Er is een volledig dynamische oplossing van het bedoelde ont-bindbare zoekprobleem, waarin updates maar $O(B(n/\log n))$ stappen vergen en zoekakties altijd binnen $O(\log n \cdot Q(n/\log n))$ stappen het gewenste ant-word leveren.*

Zo is bijvoorbeeld een statische oplossing van het naaste-buur-probleem bekend met $Q(n) = \log n$ en $B(n) = n \log n$ (LIPTON & TARJAN [31]). Uit de

telling volgt dat er dan een dynamische oplossing is die "maar" $O(n)$ per update en $O(\log^2 n)$ per zoekactie vergt.

Het is bij het ontwerpen van algoritmen vaak helemaal niet duidelijk of een bepaalde algoritmische techniek wel toepasbaar is. Een dieper onderzoek van de onderliggende structuur van een probleem kan dan gewenst zijn, hetgeen niet zelden tot onontdekte stukjes wiskunde voert. Zo hebben de Fast Fourier Transform en Newton Iteratie uiteindelijk toepassing gevonden in de computer-arithmetiek van integers (zie b.v. BORODIN & MUNRO [7]). Een ander voorbeeld is de "planar separator" stelling van LIPTON & TARJAN [30].

STELLING 2.8. *Men kan elke n-knopige planaire graph in $O(n)$ stappen in drie stukken A, B en S partitioneren zodanig dat geen knoop van A direkt met een knoop van B verbonden is, A en B elk ten hoogste $\frac{2}{3}n$ knopen bezitten en S uit ten hoogste $2\sqrt{2} \cdot \sqrt{n}$ knopen bestaat.*

Met deze stelling wordt een krachtige vorm van divide-and-conquer op planaire graphen mogelijk (zie [31]). PHILIPP & PRAUSS [41] bewezen onlangs dat er n-knopige planaire graphen bestaan waarin elke separator ten minste $\sqrt{\frac{7}{6}} \cdot \sqrt{n}$ knopen telt. Hoe het gebruik van beschikbare algoritmische technieken tot een geheel nieuwe visie op klassieke problemen kan voeren is onlangs nog weer bewezen in SHAMOS [50]

Een belangrijk hulpmiddel bij ontwerpen is de beschikking over efficiënte subroutines, dus de parate kennis dat zekere taken binnen een bepaalde efficiency uitvoerbaar zijn. Niet zelden ziet men dat een ontwerper naar het gebruik van dergelijke subroutines toewerkt "omdat die tenminste een scherpe tijdgrens halen". Zo kan men een LU-dekompositie algoritme formuleren dat geregeld matrix multiplikaties aanroept en uiteindelijk binnen dezelfde $O(n^\alpha)$ tijdgrens blijft (mits $\alpha > 2$, zie BUNCH & HOPCROFT [9]). Een interessant gevolg is

STELLING 2.9. *Er is een algoritme dat binnen $O(n^{2,51\dots})$ stappen de determinant van een $n \times n$ matrix levert.*

Dit algoritme behoeft nog niet numeriek aantrekkelijk te zijn, maar leert in ieder geval iets over de complexiteit van het determinant probleem. Vindt men een betere tijdgrens voor matrix multiplikatie, dan verandert de stelling mee.

Om een efficiënter algoritme te vinden is het niet altijd nodig een geheel nieuwe benadering tot het oorspronkelijke probleem te vinden. Vele algoritmen zijn uiteindelijk het resultaat van sleutelen aan eerdere algoritmen en/of hun implementaties. Een mooi voorbeeld van wat door algoritmische engineering bereikt kan worden toont de volgende "geschiedenis" van het maximum-flow probleem voor netwerken met V knopen en E kanten:

ALGORITME	TIJDGRENS
Ford en Fulkerson (1956)	-
Edmonds en Karp (1969)	$E^2 V$
Dinic (1970)	EV^2
Karzanov (1973)	V^3
Malhotra et al. (1978)	V^3
Cherkasky (1976)	$E^{1/2} \cdot V^2$
Galil (1978)	$E^{2/3} V^{5/3}$
Galil en Naamad (1979)	$EV \log^2 V$
Sleator en Tarjan (1980)	$EV \log V$

ITAI & SHILOACH [20] vonden een $O(V^2 \log V)$ algoritme voor het maximum-flow probleem in ongerichte planaire graphen.

Tenslotte kan het de ontwerper gelukken een vraagstuk in zijn geheel te transformeren tot een ander, waarvoor een efficiënt algoritme reeds bestaat. Zo worden vragen in de operations research nogal eens tot een lineair programmeringsprobleem herleid, dat vervolgens met de simplex methode wordt opgelost. (Of Khachian's algoritme, zie b.v. LOVÁSZ [33] of SCHRIJVER [49], hier praktische verbetering brengt, valt nog te bezien.) Interessante voorbeelden van algoritmische transformatie vindt men ook in BROWN [8]. Maar het grootste belang van transformaties is wellicht dat daarmee de complexiteiten van geheel uiteenlopende problemen aan elkaar gerelateerd kunnen worden (zie b.v. KARP [22] of de volgende sectie). Het kan leren dat de hoop op een efficiënter algoritme zinloos is, indien dit reeds zo is voor het getransformeerde vraagstuk.

3. WAT HEET DOENLIJK

Een voor een zekere taak bestemd algoritme is soms helemaal niet realistisch, gelet op de benodigde hoeveelheden rekentijd en geheugen om het ermee korresponderende komputerprogramma daadwerkelijk te kunnen executeren.

Voor kleine inputs gaat het misschien wel, maar bij oplopend formaat kan het algoritme snel "ondoenlijk" worden.

VOORBEELD. Een chemikus houdt een file van structuren bij en zoekt een algoritme voor het GRAPH-ISOMORPHISM probleem als onderdeel van een retrieval procedure. U suggereert hem alle knoop-identifikaties af te testen, maar belooft het probleem verder te onderzoeken.

Of het zoeken naar een beter algoritme zinvol is, zal van de intrinsieke complexiteit van een probleem afhangen. Men kan proberen een probleem in een bekende hiërarchie van complexiteitsklassen in te schalen, ook al geeft dit maar heel ruwe informatie. We zullen in deze notities daar niet op in gaan en ons zelfs niet bekommeren om een al te preciese definitie van verder benodigde begrippen (zie b.v. GAREY & JOHNSON [17]).

Men noemt een probleem wel "doenlijk" indien er een algoritme voor bestaat waarvan de geschatte rekentijd T voldoet aan

$$T(n+i) \leq c \cdot T(n) \quad \text{voor } 0 \leq i \leq n$$

voor zekere konstante c . Er volgt dat T begrensd moet zijn door een polynoom in n .

DEFINITIE. P is de klasse van problemen die door algoritmen met polynomiaal begrensde rekentijd oplosbaar zijn (uitgaande van een natuurlijke komputer representatie voor de inputs).

Men kan zich nu bijvoorbeeld afvragen of een praktisch vraagstuk voor graphen zoals

TRAVELLING-SALESMAN: bestaat er een gesloten pad van begrensde lengte dat alle punten van de graph juist één keer treft

in P zit. Het opsommen en vervolgens testen van alle paden in een graph is stellig niet polynomiaal begrensd en voert tot een algoritme dat al voor graphen van 20 knopen meer sekonden rekentijd kan vergen dan sinds het begin der jaartelling zijn verlopen (cf. KARP [24]). De moeilijkheid komt beter uit als we het algoritme formuleren als: raad de opvolgende kanten en verifiër of het resulterende pad aan de eisen voldoet. Het verifiëren kan

eenvoudig in polynomiale tijd geschieden, maar geen vorm van backtracking is bekend waarmee het "raden" (als substituut voor opsommen) tot een polynomiaal begreind deterministisch zoek-proces is om te vormen.

Laten we vanaf nu alle vraagstukken als ja/nee beslissingen formuleren. Een algoritme heet non-deterministisch indien is toegelaten dat in (sommige) stappen uit een begreind aantal mogelijke akties gekozen wordt. Een non-deterministisch algoritme moet "ja" kunnen opleveren (door een rij van keuzen) dan en precies dan als de gekodeerde input als probleem inderdaad oplosbaar is. De "rekentijd" van een nondeterministisch algoritme is gedefinieerd als het kleinste aantal stappen nodig om op "ja" uit te komen, aannemend dat zo'n antwoord voor de gegeven input bestaat.

DEFINITIE. NP is de klasse van problemen die door nondeterministische algoritmen met polynomiaal begreinde rekentijd oplosbaar zijn (uitgaande van een natuurlijke komputer representatie voor de inputs).

De vraag of NP-problemen zoals TRAVELLING-SALESMAN doenlijk zijn kan kortweg worden samengevat in

Open probleem : is $P = NP$?

Ga na dat ook GRAPH-ISOMORPHISM in NP zit en dus automatisch "doenlijk" zou zijn als P gelijk aan NP mocht blijken.

Bij het ontwerpen van algoritmen stuit men nogal eens op taken die wel in NP maar kennelijk niet in P zitten. Een bekend voorbeeld uit de propositie-logika is

3SAT: is een gegeven formule in konjunktieve normaalvorm met precies 3 literals per klausule vervulbaar

en er zijn vele andere te noemen. Het klinkt misschien oneerlijk om de vraag naar een doenlijk algoritme voor TRAVELLING-SALESMAN of 3SAT op één lijn te stellen met het hele $P = NP$ probleem. Toch is dit voor sommige problemen juist.

DEFINITIE. M heet p-reduceerbaar tot L indien een in polynomiale tijd berekenbare afbeelding f bestaat zodat voor elke x : x is oplosbaar als M-probleem $\iff f(x)$ is oplosbaar als L-probleem.

Is M p-reduceerbaar tot L en $L \in P$ (NP), dan is ook $M \in P$ (NP).

DEFINITIE. L heet NP-volledig indien $L \in NP$ en iedere $M \in NP$ p-reduceerbaar tot L is.

Is L NP-volledig, dan geldt natuurlijk: $P = NP \iff L \in P$. Een belangrijk resultaat van COOK [14] uit 1971 toont dat NP-volledige problemen inderdaad bestaan!

STELLING 3.1. 3SAT is NP-volledig.

KARP [22] gaf in 1972 al een lange lijst van bekende problemen, waaronder TRAVELLING-SALESMAN, die NP-volledig bleken te zijn. En er zijn er sindsdien vele aan toegevoegd (zie [17]).

Om een probleem NP-volledig te bewijzen is het meestentijds onnodig de preciese achtergronden van de theorie te leren, en kan men zich eenvoudigweg beroepen op

STELLING 3.2. Zij L NP-volledig. Is $M \in NP$ en L p-reduceerbaar tot M , dan is M NP-volledig.

Het wordt daarmee een kwestie van het ene probleem in het andere "vertalen" (door een p-reduktie). Zo wordt in [17] uit 3SAT de NP-volledigheid be-
wezen van

3DM: gegeven eindige verzamelingen X, Y, Z met $|X| = |Y| = |Z| = q$ en een $M \subseteq X \times Y \times Z$, bestaat er een $M' \subseteq M$ zodat elk element van X, Y en Z in precies één triple van M' voorkomt.

We illustreren hoe uit 3DM de NP-volledigheid van weer andere problemen is te bewijzen, gebruik makend van 3.2 en een geschikte p-reduktie. Bezie de volgende variant van het "knapsack" probleem:

KNAPSACK1: gegeven positieve natuurlijke getallen $a_1, \dots, a_n$ en b , is er een $J \subseteq \{1, \dots, n\}$ zodat $\sum_{i \in J} a_i = b$.

STELLING 3.3. KNAPSACK1 is NP-volledig.

BEWIJS. Natuurlijk is KNAPSACK1 in NP: raad maar een J en verifieer dat het een oplossing geeft. Dat kan in "nondeterministische" polynomiale tijd. We tonen voorts dat 3DM p-reduceerbaar is tot KNAPSACK1. Representer de

elementen van X , Y en Z door de getallen ("cijfers") 0 t/m $q-1$. Kies $\beta = q^3 + 1$. Vertaal naar een KNAPSACK1 probleem door elke triple $(i, j, k) \in M$ om te vormen tot een $a_{ijk} = \beta^i + \beta^{q+j} + \beta^{2q+k}$ en $b = \beta^{3q-1}/\beta - 1 = 1 + \beta + \beta^2 + \dots + \beta^{3q-1}$ te nemen. Getallen binair schrijvend, is dit inderdaad een p-redukatie. Stellen we ons de getallen even voor in het " β -tallig" stelsel, dan is duidelijk dat de oplosbaarheid van 3DM gelijk geworden is aan de oplosbaarheid van het vertaalde KNAPSACK1 probleem. Immers, bestaat een $M' \subseteq M$ zodat elk element van X , Y en Z in juist één triple van M' voorkomt, dan geldt $\sum_{(i,j,k) \in M'} a_{ijk} = b$. Bestaat omgekeerd een M' (noodzakelijk met ten hoogste q^3 elementen) met $\sum_{(i,j,k) \in M'} a_{ijk} = b$, merk dan op dat in de linker som voor elke macht van β een "cijfer" hoogstens q^3 groot kan komen. Omdat $\beta > q^3$, geldt de wet van de unieke getal representatie en kan de linker som alleen gelijk aan b zijn als alle machten β^i , β^{q+j} en β^{2q+k} ($0 \leq i, j, k \leq q-1$) maar precies één keer worden bijgedragen door de triples in M' . $\square$

Men kan KNAPSACK1 ook formuleren als de vraag of een vergelijking $a_1 x_1 + \dots + a_n x_n = b$ oplosbaar is met $x_i \in \{0, 1\}$. Het stellen van wat minder strakke eisen voor oplosbaarheid behoeft de moeilijkheden nog niet te vermijden. Bezie eens

KNAPSACK2: gegeven positieve gehele getallen $a_1, \dots, a_n$, b en d , bestaat er een oplossing van $a_1 x_1 + \dots + a_n x_n = b$ met natuurlijke getallen x_i zodat $\sum_i x_i = d$.

Voor vaste d is KNAPSACK2 met een "polynomiaal begrensd" algoritme op te lossen: som maar alle $O(n^d)$ rijtjes $\{x_i\}_i$ met $0 \leq x_i \leq d$ en $\sum x_i = d$ op en test of er een oplossing bij zit. Maar laten we d met de overige variabelen vrij, dan geldt

STELLING 3.4. KNAPSACK2 is NP-volledig.

BEWIJS. Weer is KNAPSACK2 $\in$ NP. Op nagenoeg dezelfde wijze als in 3.3 blijkt 3DM ook p-reduceerbaar tot KNAPSACK2. Kies $\beta = q^4 + 1$ en vertaal als tevoren en neem $d = q$. Omdat een oplossing M' van 3DM noodzakelijk q triples heeft, zal daaruit meteen een oplossing van $\sum_{(i,j,k) \in M'} a_{ijk} x_{ijk} = b$ volgen met $\sum_{(i,j,k) \in M'} x_{ijk} = q$. (Tot zover lijkt het nog erg op KNAPSACK1.) Bestaat omgekeerd een oplossing van $\sum_{(i,j,k) \in M} a_{ijk} x_{ijk} = b$ onder de

gestelde eis aan $\sum_{(i,j,k) \in M} x_{ijk}$, merk dan op dat in de linker som nu elke macht van β hoogstens q^3 keren (de grens op het aantal triples) met een factor q (de grens van elke x -waarde) kan worden aangedragen en dus een "cijfer" hoogstens q^4 groot voor zich kan krijgen. Omdat we nu $\beta > q^4$ kozen, geldt toch weer de wet van de unieke getal representatie en kan men concluderen dat noodzakelijk elke $x_{ijk} \in \{0,1\}$ en dat $M' = \{(i,j,k) \mid x_{ijk} = 1\}$ en oplossing van het oorspronkelijke 3DM probleem moet zijn. $\square$

Het begint er op te lijken dat het oplossen van een eenvoudige lineaire vergelijking $a_1 x_1 + \dots + a_n x_n = b$ minder simpel is dan wellicht werd aangenomen als we een oplossing over $\mathbb{N}$ zoeken. Over $\mathbb{Z}$ is het in polynomiale tijd te doen door testen dat $\text{ggd}(a_1, \dots, a_n) \mid b$, maar over $\mathbb{N}$ bestaat een soortgelijk criterium (nog) niet. We omschrijven het probleem als

SLEQ: gegeven positieve natuurlijke getallen $a_1, \dots, a_n$ en b , bestaat er een oplossing van $a_1 x_1 + \dots + a_n x_n = b$ met natuurlijke getallen x_i .

In het volgend resultaat (cf. VAN EMDE BOAS [59]) wordt ons vermoeden bevestigd.

STELLING 3.5. SLEQ is NP-volledig.

BEWIJS. SLEQ $\in$ NP door op te merken dat de x_i door b begrensd (moeten) zijn en het "raden" dus polynomiaal begrensd kan blijven. We tonen nu dat KNAPSACK2 p-reduceerbaar is tot SLEQ. Gegeven een KNAPSACK2 probleem, kies $G = 1 + \max\{b, d \sum_i a_i - b\}$ en stel $A_i = a_i + B$ en $B = b + dG$. Het SLEQ-probleem $\sum_i A_i x_i = B$ kan men na invullen omschrijven tot $G(\sum_i x_i - d) = b - \sum_i a_i x_i$. Het is duidelijk dat een oplossing van KNAPSACK2 meteen een oplossing van het vertaalde SLEQ-probleem geeft. Stel omgekeerd dat een oplossing van $G \cdot (\sum_i x_i - d) = b - \sum_i a_i x_i$ in natuurlijke getallen bestaat. Omdat $G > b$, kan $\sum_i x_i - d$ niet > 0 zijn. Maar $\sum_i x_i - d$ kan ook niet < 0 zijn, omdat dan $G \leq G \cdot (d - \sum_i x_i) = \sum_i a_i x_i - b \leq d \sum_i a_i - b$ zou zijn. Dus moet $\sum_i x_i = d$ en (dus) $\sum_i a_i x_i = b$ zijn, een oplossing van het oorspronkelijke KNAPSACK2 probleem. Ga na dat de afbeelding van KNAPSACK2 naar SLEQ inderdaad een p-reductie is. $\square$

Lukt het dus om voor oplossing van lineaire vergelijkingen $a_1 x_1 + \dots + a_n x_n = b$ over $\mathbb{N}$ een deterministisch algoritme met polynomiaal begrensde rekentijd te vinden, dan zou men meteen het hele $P = NP$ probleem hebben opgelost!

Men kan de theorie van NP-volledigheid verfijnen door een strikter reductie-begrip te hanteren. BERMAN & HARTMANIS [5] bewezen dat in de huidige theorie alle bekende NP-volledige problemen zelfs p-isomorf zijn. Verder, merk op dat we als we weten dat een oplossing "bestaat" we nog geen oplossing "hebben" en een NP-volledig probleem zo nog wel eens extra moeilijkheden bij berekening kan geven (zie VALIANT [56]). Een probleem kan voorts extra moeilijk zijn omdat we "alle" oplossingen willen tellen. Is #P de klasse van "tel-varianten" van NP-problemen (voor een preciese definitie, zie VALIANT [58]), dan heet L #P-volledig indien $L \in \#P$ en elk probleem $M \in \#P$ in polynomiale tijd oplosbaar is als een subroutine aanroep aan L als "een stap" mag gelden. Het analogon van 3.1 blijkt waar ([58]).

STELLING 3.6. *Het tellen van de mogelijke "vervullingen" van een 3SAT probleem is #P-volledig.*

Ook het tellen van alle oplossingen van een SLEQ probleem is #P-volledig. Een lijst van overige, bekende #P-problemen vindt men in [58]. Interessant is dat we hier het "vanouds bekende" probleem van de permanent berekening kunnen plaatsen (VALIANT [57])

PERMANENT: gegeven een integer matrix $A = (a_{ij})$, bereken $\sum_{\sigma} \prod_i a_{i\sigma(i)}$ gesommeerd over alle permutaties $\sigma \in S_n$.

STELLING 3.7. *Berekening van PERMANENT is #P-volledig.*

Dat men er nog altijd niet in geslaagd is permanenten even efficiënt te berekenen als determinanten (vgl. 2.9), blijkt dus nauw verweven met de algemene P-versus-NP problematiek.

Een bewijs dat een bekend NP-volledig probleem te reduceren is tot een gegeven probleem L wordt tegenwoordig vaak als argument (of exkuus) voor de "ondoenlijkheid" van zo'n probleem aangevoerd. Blijkt ooit nog eens dat $P = NP$, hetgeen onwaarschijnlijk lijkt, dan zal men de status van vele problemen moeten herbezien. Van maar betrekkelijk weinig konkrete problemen uit de praktijk is bekend dat elk algoritme ervoor zeker, zeg, exponentiëel of erger toenemende rekentijd moet hebben (zie b.v. STOCKMEYER & CHANDRA [53]).

En hoe zit het nu met GRAPH-ISOMORPHISM? Helaas is hiervan nóch bekend of het in P zit, nóch of het NP-volledig is! LADNER [28] bewees dat er dergelijke problemen moeten zijn als $P \neq NP$. Voor GRAPH-ISOMORPHISM ontbreekt nochtans het antwoord of het "doenlijk" is of niet. Onlangs bewees LUKS [34] dat isomorfie in polynomiale tijd beslisbaar is voor elke klasse van grafen waarin knopen begrensde graad hebben.

4. ONDOENLIJK, MAAR TOCH TE DOEN?

Al is een probleem ondoenlijk voor toenemende n (bijvoorbeeld door de bewezen NP-volledigheid), dan behoeft men de haalbaarheid van een goed algoritme voor de praktijk niet zomaar uit te sluiten. Blijven de verwachte inputs "klein" dan is van een explosie in de nodige rekentijd misschien wel niets te merken. Heuristische methoden kunnen bovendien tot verdere bezuinigingen in rekentijd leiden en de praktische reikwijdte van een algoritme verder vergroten. Met het beschikbaar komen van de nieuwe multiprocessor-technologie (b.v. MEAD & CONWAY [35]), wordt het zelfs realistisch alle NP-problemen als "doenlijk" te gaan zien. Natuurlijk blijven andere problemen even ondoenlijk als tevoren (zie b.v. VITANYI [63]).

In de laatste jaren zijn allerlei manieren bedacht om onder de complexiteit van problemen uit te komen. Zo kan men voor sommige problemen (waaronder ook NP-volledige) algoritmen vinden die "meestal" snel zijn (zie e.g. KARP [23]). De vereiste probabilistische analyses zijn vaak moeilijk. WEIDE [64] biedt een gedegen inleiding tot het onderwerp. Een geheel andere mogelijkheid is een algoritme te vinden dat altijd snel maar slechts beperkt betrouwbaar is (zie b.v. RABIN [43]).

In nogal wat optimaliseringsvraagstukken die zelf NP-volledig (of erger) zijn, blijkt het mogelijk om snel (lees: met een polynomiaal begrensde algoritme) een antwoord te bepalen dat "dicht" bij het gezochte optimum moet liggen. Er is eigenlijk geen coherente theorie van "approximatie algoritmen", maar er zijn enkele interessante inzichten bereikt. We beperken ons eerst tot optimaliseringsvragen waarin een "minimum" gezocht wordt.

DEFINITIE. Een approximatie algoritme voor een probleem haalt een performance factor K indien voor alle toegelaten inputs geldt:

$$|output| / |optimum| \leq K.$$

Herdefiniëer TRAVELLING-SALESMAN nu zo dat om een kortste gesloten pad gezocht wordt dat alle punten in een graph precies één keer treft. SAHNI & GONZALEZ [47] merkten op

STELLING 4.1. *Is $P \neq NP$, dan bestaat er voor geen enkele konstante K een polynomiaal begremsd approximatie algoritme dat een performance faktor K haalt op TRAVELLING-SALESMAN.*

Anders is het bij GEOMETRIC-TRAVELLING-SALESMAN, waarin om een kortste gesloten pad door een stel punten in het vlak (met de euklidische metriek) gevraagd wordt. Het probleem zelf is NP-volledig (PAPADIMITRIOU [40]), maar CHRISTOFIDES [10] (zie ook [11]) bewees

STELLING 4.2. *Er is een polynomiaal begremsd approximatie algoritme dat op GEOMETRIC-TRAVELLING-SALESMAN een performance faktor $3/2$ haalt.*

BEWIJS. Beschouw in gedachten een kortste tour T , van lengte OPT . Laat men een kant weg, dan ontstaat een opspannende boom. Elke minimale opspannende boom heeft dus een lengte $< OPT$. Neem zo'n minimale opspannende boom en beschouw de verzameling S van knopen van oneven graad daarin. S heeft noodzakelijk een even aantal elementen. Laat uit T alle punten niet in S weg, trek de verbindingslijnen recht en kleur kanten afwisselend rood en wit. (Omdat $|S|$ = even, krijgt elke kant precies één kleur.) Zowel de rode als de witte kanten vormen een perfecte matching (paring) op S . Elke minimale lengte perfecte matching kan dus niet meer dan $\frac{1}{2}OPT$ lang zijn. Nu kan men zowel een minimale opspannende boom B als een minimale lengte perfecte matching M op zijn knopen van oneven graad in polynomiale tijd bepalen (e.g. LAWLER [29]). De graph die ontstaat door de kanten van M aan B toe te voegen is samenhangend, heeft louter knopen van even graad en bezit dus een Euler-pad. Zo'n pad is in polynomiale tijd te vinden (e.g. EVEN [16]) en zal lengte $< OPT + \frac{1}{2}OPT = \frac{3}{2}OPT$ moeten hebben. Sla tijdens het doorlopen van het Euler-pad punten over zodra ze voor een tweede of latere keer bezocht worden en trek pas weer een directe "verbinding" als een nieuw punt wordt bereikt. De resulterende travelling salesman tour heeft lengte $< \frac{3}{2}OPT$ en het geschetste polynomiale approximatie algoritme haalt een performance factor $3/2$. $\square$

CHRISTOFIDES [11] geeft een goed overzicht van de thans bekende heuristieken voor (GEOMETRIC-)TRAVELLING-SALESMAN. Optimale tours in het vlak doorsnijden zichzelf nooit, maar bijna-optimale tours die door approximatie-

algoritmen worden afgeleverd voldoen vaak niet aan dat criterium. VAN LEEUWEN & SCHOONE [60] bewezen echter dat uit elk pad alle doorsnijdingen in polynomiale tijd verwijderd kunnen worden, indien zulks gewenst is.

Over KNAPSACK1 is ook wel wat te zeggen, indien we het omschrijven tot een optimaliseringsvraagstuk

KNAPSACK1: gegeven positieve natuurlijke getallen $a_1, \dots, a_n$ en b , wat is de maximum waarde $\leq b$ die een som $\sum_{i \in J} a_i$ met $J \subseteq \{1, \dots, n\}$ kan hebben.

DEFINITIE. Een approximatie algoritme voor een probleem heet een ϵ -approximatie algoritme indien voor alle toegelaten inputs

$$|\text{output-optimum}| / |\text{optimum}| \leq \epsilon.$$

JOHNSON [21] bewees

STELLING 4.3. Voor elke $\epsilon > 0$ bestaat er een polynomiaal begrensde ϵ -approximatie algoritme voor KNAPSACK1.

Meer over approximatie algoritmen vindt men in [17]. SAHNI [46] geeft een overzicht van handige technieken die in approximatie algoritmen worden toegepast. In CHRISTOFIDES, MINGOZZI, TOTH & SANDI [12] worden heldere overzichten geboden van de vele exacte en heuristische methoden voor oplossing van optimaliseringsvraagstukken.

5. REFERENTIES

- [1] AHO, A.V., HOPCROFT, J.E. & J.D. ULLMAN, *The design and analysis of computer algorithms*, Addison-Wesley Publ. Comp., Reading, Mass., 1974.
- [2] BENTLEY, J.L., *An introduction to algorithm design*, CMU-CS-78-121, Carnegie Mellon University, Pittsburgh, 1978.
- [3] BENTLEY, J.L., *Decomposable searching problems*, Inf. Proc. Lett. 8 (1979) 244-251.
- [4] BENTLEY, J.L. & M.I. SHAMOS, *Divide and conquer for linear expected time*, Inf. Proc. Lett. 7 (1978) 87-91.

- [5] BERMAN, L. & J. HARTMANIS, *On isomorphisms and density of NP- and other complete sets*, SIAM J. Computing 6 (1977) 305-323.
- [6] BLUM, M., FLOYD, R.W., PRATT, V.R., RIVEST, R.L. & R.E. TARJAN, *Time bounds for selection*, J. Comp. Syst. Su. 7 (1972) 448-461.
- [7] BORODIN, A.B. & I. MUNRO, *The computational complexity of algebraic and numeric problems*, Theory of Comput. Series, American Elsevier Publ. Comp., New York, 1975.
- [8] BROWN, K.Q., *Geometric transforms for fast geometric algorithms*, Ph. D. Thesis, CMU-CS-80-01, Carnegie Mellon University, Pittsburgh, 1980.
- [9] BUNCH, J. & J.E. HOPCROFT, *Triangular factorization and inversion by fast matrix multiplication*, Math. Comp. 28 (1974) 231-236.
- [10] CHRISTOFIDES, N., *Worst case analysis of a new heuristic for the travelling salesman problem*, Tech. Rep., Grad. School of Industr. Admin., Carnegie Mellon University, Pittsburgh, 1976.
- [11] CHRISTOFIDES, N., *The travelling salesman problem*, Chapter 6, pp. 131-149 in [12]
- [12] CHRISTOFIDES, N., A. MINGOZZI, P. TOTH & C. SANDI, *Combinatorial optimization*, Wiley & Sons, Chichester, 1979.
- [13] CLANCY, M.J. & D.E. KNUTH, *A programming and problem solving seminar*, STAN-CS-77-606, Stanford University, Stanford, 1977.
- [14] COOK, S.A., *The complexity of theorem proving procedures*, Proc. 3rd Annual ACM Symp. on Theory of Computing, 1971, pp. 151-158.
- [15] DE JONG, L.S. & J. VAN LEEUWEN, *An improved bound on the number of multiplications and divisions necessary to evaluate a polynomial and all its derivatives*, SIGACT NEWS 7 (1975) 32-34.
- [16] EVEN, S., *Graph algorithms*, Comp. Sci. Press, Potomac, 1979.
- [17] GAREY, M.R. & D.S. JOHNSON, *Computers and intractability: a guide to the theory of NP-completeness*, Freeman & Company, San Fransisco, 1979.
- [18] HOPCROFT, J.E., *Complexity of computer computations*, Proc. IFIP Congress 74, North Holland Publ. Comp., Amsterdam, 1975, pp. 620-626.

- [19] HOROWITZ, E. & S. SAHNI, *Fundamentals of data structures*, Computer Science Press Inc., Potomac, 1976.
- [20] ITAI, A. & Y. SHILOACH, *Maximum flow in planar graphs*, Techn. Rep. 88, Technion, Haifa, 1979.
- [21] JOHNSON, D.S., *Approximation algorithms for combinatorial problems*, J. Comp. Syst. Sci. 9 (1974) 256-278.
- [22] KARP, R.M., *Reducibility among combinatorial problems*, in: R.E. MILLER & J.W. THATCHER (eds), *Complexity of computer computations*, Plenum Press, New York, 1972, pp. 85-103.
- [23] KARP, R.M., *Probabilistic analysis of partitioning algorithms for the travelling salesman problem in the plane*, Math. Oper. Res. 2 (1977) 209-224.
- [24] KARP, R.M., *Combinatorial search problems: is exponential growth inevitable?*, UC Electronics Newsletter, University of California, Berkeley, 1979, pp. 6-9.
- [25] KIRKPATRICK, D.G., *Optimal search in planar subdivisions*, Techn. Rep., Dept of Computer Science, University of British Columbia, Vancouver, 1979.
- [26] KNUTH, D.E., *Optimum binary search trees*, Acta Inform. 1 (1971) 14-25.
- [27] KNUTH, D.E., *The art of computer programming*, vol 3: *searching and sorting*, Addison-Wesley Publ. Comp., Reading, Mass., 1973.
- [28] LADNER, R.E., *On the structure of polynomial time reducibility*, J. ACM 22 (1975) 155-171.
- [29] LAWLER, E.L., *Combinatorial optimization: networks and matroids*, Holt, Rinehart & Winston, New York, 1976.
- [30] LIPTON, R.J. & R.E. TARJAN, *A separator theorem for planar graphs*, STAN-CS-77-627, Stanford University, Stanford, 1977.
- [31] LIPTON, R.J. & R.E. TARJAN, *Applications of a planar separator theorem*, STAN-CS-77-628, Stanford University, Stanford, 1977.
- [32] LORIN, H., *Sorting and sort systems*, Syst. Progr. Series, Addison-Wesley Publ. Comp., Reading, Mass., 1975.
- [33] LOVÁSZ, L., *A new linear programming algorithm - better or worse than the simplex method*, Math. Intell. 2 (1980) 141-146.

- [34] LUKS, E.M., *Isomorphism of graphs of bounded valence can be tested in polynomial time*, Proc. 21st Ann. IEEE Symp. on Foundations of Computer Science, 1980, pp. 42-49.
- [35] MEAD, C. & L. CONWAY, *Introduction to VLSI systems*, Addison-Wesley Publ. Comp., Reading, Mass., 1980.
- [36] MEHLHORN, K., *Dynamic binary search*, SIAM J. Computing 9 (1979) 175-198.
- [37] MEYER, A.R., *Discrete computation: theory and open problems*, MAC Techn. Memo 39, Lab. for Computer Science, MIT, Cambridge, 1974.
- [38] MURAOKA, Y. & D.J. KUCK, *On the time required for a sequence of matrix products*, C. ACM 16 (1973) 22-26.
- [39] OVERMARS, M.H. & J. VAN LEEUWEN, *Dynamic multidimensional data-structures based on quad- and k-d trees*, RUU-CS-80-2, University of Utrecht, Utrecht, 1980.
- [40] PAPADIMITRIOU, C.H., *The euclidean travelling salesman problem is NP-complete*, Theor. Comp. Sci. 4 (1977) 237-244.
- [41] PHILIPP, R. & E.J. PRAUSS, *Über Separatoren in planaren Graphen*, Acta Inform. 14 (1980) 87-106.
- [42] PREPARATA, F.P. & S.J. HONG, *Convex hulls of finite sets of points in two and three dimensions*, C. ACM 20 (1977) 87-93.
- [43] RABIN, M.O., *Probabilistic algorithms*, in: J.F. TRAUB (ed.), *Algorithms and complexity: new directions and recent results*, Acad. Press, New York, 1976, pp. 21-39.
- [44] RABIN, M.O., *Complexity of computations*, C. ACM 20 (1977) 625-633.
- [45] REINGOLD, E.M., *On the optimality of some set algorithms*, J. ACM 19 (1972) 649-659.
- [46] SAHNI, S., *General techniques for combinatorial approximation*, Oper. Res. 25 (1977) 920-936.
- [47] SAHNI, S. & T. GONZALEZ, *P-complete approximation problems*, J. ACM 23 (1976) 555-565.
- [48] SCHÖNHAGE, A. en anderen, te verschijnen.
- [49] SCHRIJVER, A., *Khachian's ellipsoïde-methode voor lineaire programmering; dit colloquium.*

- [50] SHAMOS. M.I., *Geometry and statistics: problems at the interface*, in: J.F. TRAUB (ed.), *Recent results and new directions in algorithms and complexity*, Acad. Press, New York, 1976, pp. 251-280.
- [51] SHAMOS, M.I., *Computational geometry*, Ph. D. Thesis, Dept of Computer Science, Yale University, New Haven, 1978 (te verschijnen).
- [52] STANDISH, T.A., *Data structure techniques*, Addison-Wesley Publ. Comp., Reading, Mass., 1980.
- [53] STOCKMEYER, L.J. & A.K. CHANDRA, *Intrinsically difficult problems*, Scientific American (1979) 140-159.
- [54] STRASSEN, V., *Gaussian elimination is not optimal*, Num. Math. 13 (1969) 354-456.
- [55] TARJAN, R.E., *Complexity of combinatorial problems*, STAN-CS-77-609, Stanford University, Stanford, 1977.
- [56] VALIANT, L.G., *Relative complexity of checking and evaluating*, Inf. Proc. Lett. 5 (1976) 20-23.
- [57] VALIANT, L.G., *The complexity of computing the permanent*, Theor. Comp. Sci. 8 (1979) 189-202.
- [58] VALIANT, L.G., *The complexity of enumeration and reliability problems*, SIAM J. Computing 8 (1979) 410-421.
- [59] VAN EMDE BOAS, P., Private communication (1979).
- [60] VAN LEEUWEN, J. & A.A. SCHOONE, *Untangling a travelling salesman tour in the plane*, RUU-CS-80-11, University of Utrecht, Utrecht, 1980.
- [61] VAN LEEUWEN, J. & P. VAN EMDE BOAS, *Some elementary proofs of lower-bounds in complexity theory*, J. Lin. Alg. and its Appl. 19 (1978) 63-80.
- [62] VAN LEEUWEN, J. & D. WOOD, *Dynamization of decomposable searching problems*, Inf. Proc. Lett. 10 (1980) 51-56.
- [63] VITANYI, P.M.B., *Berekeningsmodel en complexiteit*, dit colloquium.
- [64] WEIDE, B.W., *Statistical methods in algorithm design and analysis*, Ph. D. Thesis, CMU-CS-78-142, Carnegie Mellon University, Pittsburgh, 1978.

- [65] YAO, A.C., *A lower bound to finding convex hulls*, STAN-CS-79-733, Stanford University, Stanford, 1979.
- [66] YAO, F.F., *Efficient dynamic programming using quadrangle inequalities*, Proc. 12th Annual ACM Symp. on Theory of Computing, 1980, pp. 429-435.

BEREKENINGSMODEL EN COMPLEXITEIT

P.M.B. VITÁNYI

A major concern which is frequently voiced in connection with very fast computing machines, particularly in view of the extremely high speeds which may now be hoped for, is that they will do themselves out of business rapidly; that is, that they will outrun the planning and coding which they require and, therefore, run out of work.

J. von Neumann [1949].

1. INLEIDING

Naar aanleiding van de hierboven gesignaleerde zorg merkt von Neumann op, dat het zo'n vaart niet zal lopen. De komst van snellere machines zal, zonder twijfel, druk uitoefenen tot het aanpakken van problemen van grotere omvang, zodat het evenwicht tussen voorbereiding en programmering aan de ene kant, en rekentijd aan de andere kant, zich op een redelijk niveau instelt. De tegenwoordige problematiek is eerder van omgekeerde aard. Problemen die men wil oplossen kunnen, zelfs als zij van bescheiden omvang zijn, door de snelste huidige machines vaak niet binnen aanvaardbare tijd tot een slotsom gebracht worden. Opvoering van de rekensnelheid blijkt evenmin soelaas te bieden. De omvang en de aard van de aangepakte vraagstukken wordt beperkt door het vermogen van de rekenapparatuur. Deze stand van zaken kan op twee manieren verbeterd worden. Ten eerste door betere, in deze context snellere, algoritmen te ontwikkelen. Ten tweede door alternatieven te geven voor de gangbare seriële rekenapparatuur, bijvoorbeeld parallel rekenende machines. De theorie van de analyse van algoritmen en complexiteit poogt boven- en ondergrenzen te geven aan de inherente berekeningscomplexiteit van problemen. In het voorgaande werd gesuggereerd, dat deze inherente complexiteit afhangt van de aard van de veronderstelde rekenapparatuur. Formalisering van deze apparaturen noemen wij *berekeningsmodellen* (models of computation). Om de prestatie van een algoritme te analyseren, en hierdoor tot nauwkeurige uitspraken te komen over de benodigde rekentijd en geheugen verbruik, is zulk een berekeningsmodel noodzakelijk. Omgekeerd, zullen verschillende berekeningsmodellen verschillende algoritmen oproepen, en voor hetzelfde algoritme een verschillende complexiteit geven. De in de

nabije toekomst te verwachten rekenmachines met vele parallel werkende processoren, bijvoorbeeld, noodzaken nieuwe programmeerconcepten en ook een totale herwaardering van onze ideeën over de inherente complexiteit van problemen.

In deze voordracht zullen we verschillende computermodellen formuleren die simpel genoeg zijn om analytische resultaten op te leveren, maar die evenzo de belangrijkste kenmerken van bestaande en te voorziene machines waarheidsgetrouw weerspiegelen. In sectie 2 bekijken we verschillende modellen van *seriële* aard, en vergelijken die onderling met betrekking tot berekeningstijd en geheugenruimte verbruik. In sectie 3 komen modellen voor *parallelle* berekeningen aan de orde. Het zal blijken, dat de berekeningscomplexiteit hier vaak drastisch anders is dan in het seriële geval. Tenslotte zullen we enigszins ingaan op complexiteitsaspecten van de, door de huidige technologie mogelijk geworden, zeer grootschalige integratie (VLSI).

Wanneer we tijd en (geheugen) ruimte meten, geven we deze grootheden altijd als een functie van de probleemgrootte. Voor de probleemgrootte n wordt meestal de lengte van het probleem in binaire codering genomen. De probleemgrootte kan echter ook in andere parameters uitgedrukt worden. Bij een graafprobleem, bijvoorbeeld, kan de probleemgrootte uitgedrukt zijn in het aantal knopen p en het aantal kanten e van de te onderzoeken graaf G . Aannemende dat we de knopen coderen als binaire getallen tussen 1 en p , zou dit voor de probleemgrootte n kunnen leiden tot een $n \in \Theta(p^2 \log_2 p)$. Als we zeggen dat een probleem opgelost kan worden in tijd of ruimte $g(n)$, dan bedoelen we dat het programma, voor zulk een probleem van formaat n , ten hoogste $g(n)$ eenheden aan tijd of ruimte gebruikt. Bijvoorbeeld, als we zeggen dat de planairiteit van een graaf bepaald kan worden in lineaire tijd, bedoelen we dat er een algoritme is en een constante c , zodanig dat, gegeven een graaf van grootte n , de algoritme bepaalt of de graaf planair is en dat doet in ten hoogste cn tijdseenheden. Voor tijd- en geheugenverbruik zullen we in het algemeen alleen asymptotische grenzen beschouwen. Volgens de in gebruik zijnde notationale conventies definiëren we:

$$\begin{aligned} f(n) \in O(g(n)) & \text{ als er een constante } c \text{ is} \\ \text{zodat } f(n) & \leq c \cdot g(n) \text{ voor alle } n. \\ f(n) \in \Omega(g(n)) & \text{ als er een constante } c \text{ is} \\ \text{zodat } f(n) & \geq c \cdot g(n) \text{ voor alle } n. \\ f(n) \in \Theta(g(n)) & \text{ als } f(n) \in O(g(n)) \cap \Omega(g(n)). \end{aligned}$$

Op het eerste gezicht zou men kunnen denken dat de geweldige toename in rekensnelheid van de huidige, of te voorziene, apparatuur het belang van de efficiëntie van algoritmen doet afnemen. Het tegendeel is echter waar. Naarmate de machines sneller worden, en wij grotere problemen aankunnen, bepaalt de complexiteit van de algoritme steeds meer de toename in formaat van problemen die de snellere machines aankunnen. Veronderstel, wij hebben de beschikking over vijf algoritmen, $A_1, A_2, \dots, A_5$, met de onderstaande tijdscomplexiteit. (Neem aan dat de tijdeenheid een milliseconde is.)

Algoritme	Tijdscomplexiteit	Maximale probleemgrootte		
		1 sec.	1 min	1 uur
A_1	n	1000	6×10^4	$3,6 \times 10^6$
A_2	$n \log n$	140	4893	$2,0 \times 10^5$
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

Stel nu, dat onze machines 10x zo snel worden

Algoritme	Tijdscomplexiteit	Maximale probleem grootte vóór versnelling	Maximale probleem grootte na versnelling
A_1	n	s_1	$10 s_1$
A_2	$n \log n$	s_2	$\approx 10 s_2$ voor grote s_2
A_3	n^2	s_3	$3,16 s_3$
A_4	n^3	s_4	$2,15 s_4$
A_5	2^n	s_5	$s_5 + 3,3$

In de volgende secties, specificëren we modellen voor rekenapparaten, die algoritmen uitvoeren, en definiëren we wat een enkelvoudige stap in een berekening is. Ongelukkigerwijze, bestaat er geen berekeningsmodel dat voor alle omstandigheden even geschikt is. Grote moeilijkheden rijzen door de grootte van computerwoorden. Als we, bijvoorbeeld, aannemen dat een computerwoord een getal van willekeurige grootte kan bevatten kan een probleem

in een enkel woord gecodeerd worden. Van de andere kant, als we aannemen dat een computerwoord niet meer dan een vast aantal bits heeft, moeten we rekening gaan houden met de moeilijkheid van het opslaan van grote getallen, en met nog ander ongerief, dat vaak vermeden kan worden bij problemen van bescheiden omvang. Eigenlijk, zouden we voor iedere vraagstelling een bijpassend berekeningsmodel moeten zoeken. Er bestaan verschillende soorten complexiteit. Bijvoorbeeld, de *slechtste-geval* (worst-case) complexiteit en de *gemiddelde-geval* (average-case) complexiteit. Als voor een gegeven formaat n van een probleem de complexiteit bepaald wordt als het maximum van de complexiteit over alle invoer van grootte n , dan is dat de slechtste-geval complexiteit. Als, daarentegen, de complexiteit bepaald wordt als het gemiddelde van de complexiteit over alle invoer van grootte n , dan is dat de gemiddelde-geval complexiteit. In het volgende zullen we ons uitsluitend bezig houden met de slechtste-geval complexiteit.

De bedoeling van deze bijdrage is een inzicht te geven in de invloed van het berekeningsmodel op de complexiteit van problemen. Vanwege de omvang van deze taakstelling, en de beperktheid van het aantal ter beschikking staande pagina's, werd gekozen voor het vermelden van veel resultaten en het weglaten van alle bewijzen. De geïnteresseerde lezer volg de verwijzingen. Een ander punt is, dat dit geschrift in het nederlands gesteld is. De in zwang zijnde terminologie is zoveel mogelijk vertaald; om verwarring te voorkomen is bij het eerste gebruik van een nieuwe term de engelse versie er tussen haakjes achter geplaatst.

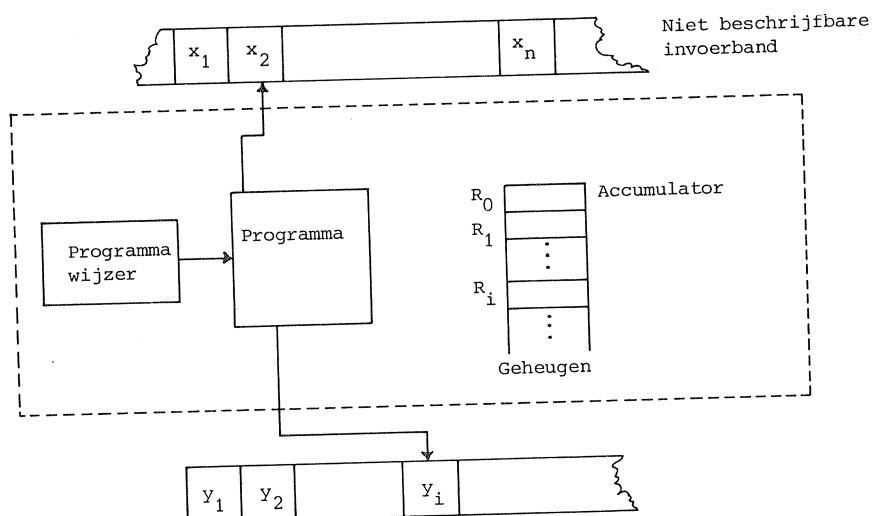
2. SERIËLE BEREKENINGSMODELLEN

Er zijn in de literatuur tal van seriële berekeningsmodellen voorhande. De belangrijkste hiervan zijn de *registerautomaat* (Random Access Machine of RAM) en de *Turing machine*. De registerautomaat lijkt het meest op gangbare rekenmachines, en leent zich daarom uitstekend als maatstaf voor het toetsen van de complexiteit van specifieke algoritmen, en daardoor, voor het bepalen van bovengrenzen aan de inherente complexiteit van problemen. Voor het bepalen van ondergrenzen aan de complexiteit voor problemen, hebben we echter een model nodig, dat even algemeen is als de RAM, of daarvan afgeleide modellen, maar primitiever. Om te laten zien dat er geen algoritme is dat een gegeven taak in minder dan een zekere hoeveelheid tijd kan uitvoeren, hebben we een preciese, en vaak gestileerde, definitie nodig van wat een algoritme is. De Turing machine, en meer nog de

onachtzame (obvious) Turing machine, is hiervan een voorbeeld. In deze sectie volgen we grotendeels AHO, HOPCROFT en ULLMAN [1974] en SAVITCH [1978].

2.1. DE REGISTERAUTOMAAT

De registerautomaat of RAM, zoals wij die behandelen, werd ingevoerd door COOK & RECKHOW [1973]. Zij modelleert, min of meer waarheidsgetrouw, een rekenmachine met één accumulator, waarvan het programma zichzelf niet kan wijzigen. Een RAM bestaat uit een invoerband waarvan alleen gelezen kan worden, een geheugen, en een uitvoerband waarop alleen geschreven kan worden: zie Figuur 1. De invoerband bestaat uit een rij vakjes, die elk een geheel getal kunnen bevatten. Iedere keer dat er een getal van de invoer gelezen wordt, beweegt de leeskop zich een vakje naar rechts. De uitvoerband bestaat, net als de invoerband, uit een rij vakjes, die in het begin van de berekening alle onbeschreven zijn. Het uitvoeren van een schrijf-instructie bestaat uit het afdrukken van een geheel getal in het vakje dat zich op dat moment onder de schrijfkop bevindt, en het verschuiven van de schrijfkop naar het rechts aangrenzende vakje. Het geheugen bestaat uit een (onbegrensde) rij registers, $R_0, R_1, \dots, R_i, \dots$, die ieder een geheel getal van willekeurige grootte kunnen bevatten. De onbeperkte van het aantal registers, en van de getallen die in één register passen, is een redelijke aanname in die gevallen waarin de grootte van het probleem klein genoeg is om in het snelgeheugen van een computer te passen, en de in de berekening gebruikte getallen klein genoeg zijn om in een computerwoord te passen.



Figuur 1. Een register automaat

Het programma van de RAM wordt niet in het geheugen opgeslagen. We kunnen daarom aannemen dat het programma zichzelf niet wijzigt. Het programma bestaat uit een rij van, mogelijk van een label voorziene, instructies. Deze instructies lijken op die welke wij gewoonlijk in de assembleercode van echte computers aantreffen. We nemen hier aan dat we de beschikking hebben over aritmetische instructies, invoer/uitvoer instructies, indirecte adressering en vertakkingsinstructies. Alle berekeningen vinden plaats in het eerste register R_0 , de *accumulator*, die, net zoals alle andere registers, een willekeurig geheel getal kan bevatten. We nemen voorlopig aan dat we de beschikking hebben over het instructie repertoire van Figuur 2.

Aard	code	kosten in tijdeenheden
1) directe toewijzing	$R_i \leftarrow b$	$\ell(b)$
2) indirecte toewijzing	$R(R_i) \leftarrow b$	$\ell(R_i) + \ell(b)$
3) indirecte toewijzing	$R_i \leftarrow R(R_j)$	$\ell(R_j) + \ell(R(R_j))$
4) optelling	$R_i \leftarrow a + b$	$\ell(a) + \ell(b)$
5) aftrekking	$R_i \leftarrow a - b$	$\ell(a) + \ell(b)$
6) voorwaardelijke vertakking	IF a COMP b THEN LABEL ₁ ELSE LABEL ₂	$\ell(a) + \ell(b)$
7) onvoorwaardelijke vertakking	GOTO LABEL	1
8) leesopdracht	READ (R_i)	$\ell(R_i)$
9) schrijfopdracht	WRITE (R_i)	$\ell(R_i)$
10) accepteer invoer	ACCEPT	1

Figuur 2. Tafel met RAM instructies plus kosten in tijdeenheden.

In deze tafel mogen a en b operanden zijn van de vorm i (een geheel getal) of R_i (de inhoud van het register R_i); de relatie COMP mag $<, \leq, \geq, >, =$ of $\neq$ zijn (met de gewone betekenis). $R(R_j)$ betekend indirecte adressering; $R(R_j)$ staat voor R_i als i de inhoud is van register R_j . Als $R_j < 0$ terwijl de RAM een instructie uitvoert, die de evaluatie van $R(R_j)$ veroorzaakt, wordt de berekening beëindigd met een foutmelding. Bij de uitvoering van een instructie van type 1-5, 8 en 9 wordt de programmawijzer met 1 opgehoogd. Dientengevolge, worden de programma-instructies in volgorde uitgevoerd, tot een instructie van type 6 of 7 verschijnt. Bij een instructie van het type GOTO LABEL wordt de programmawijzer gezet op de

instructie gemerkt door LABEL; hierna worden de instructies weer in volgorde uitgevoerd. Wordt een voorwaardelijke vertakking met a COMP b ontmoet, en is a COMP b waar, dan is die instructie gelijkwaardig met GOTO LABEL₁. Is a COMP b onwaar, dan is de instructie gelijkwaardig met GOTO LABEL₂. Het programma termineert als ACCEPT ontmoet wordt, dan wel een vertakking die equivalent is met GOTO LABEL terwijl LABEL in het programma niet voorkomt, of een instructie die een negatief direct adres inhoudt. READ R_i betekent dat de inhoud van het vakje onder de leeskop op de invoerband in register R_i geplaatst wordt; WRITE R_i betekent dat de inhoud van register R_i in het vakje onder de schrijfkop op de uitvoerband geschreven wordt. De berekening begint met de eerste instructie van het programma, de leeskop op het eerste vakje van de invoerband, de registers met inhoud 0, de uitvoerband onbeschreven en met de schrijfkop op het meest linkse vakje. In het algemeen definieert een RAM programma een afbeelding van invoer naar uitvoer. Aangezien de berekening niet voor alle mogelijke invoer hoeft te termineren is de afbeelding een partiële. (Dat wil zeggen, de afbeelding kan voor een invoer ongedefinieerd zijn.) Er zijn twee belangrijke interpretaties van de door het RAM programma gedefinieerde afbeelding mogelijk: als functie en als taal.

Stel dat een programma P altijd n gehele getallen van de invoerband leest, en altijd 1 geheel getal uitvoert. Als P, met $x_1, x_2, \dots, x_n$ op de invoerband, y op de uitvoerband schrijft en dan de berekening beëindigt, dan zeggen we dat P de functie $f(x_1, x_2, \dots, x_n) = y$ uitrekent. RAMs kunnen vanzelfsprekend alle partieel recursieve functies uitrekenen.

Een andere manier is, om de RAM als taalherkenner te beschouwen. Hierbij denken we aan een alfabet als een eindige nietlege verzameling symbolen, en bestaat een taal uit een verzameling symboolrijen over dit alfabet. De symbolen kunnen voorgesteld worden als gehele getallen $1, 2, \dots, k$, voor een of andere eindige k. In dit geval nemen we meestal aan dat elk invoerbandvakje één symbool bevat. Afgezien van deze verschillende doeleinden waarvoor een RAM (en andere berekeningsmodellen) gebruikt kunnen worden, welke doeleinden in zekere mate equivalent zijn, bestaan er twee, beslist niet equivalente, manieren waarop de berekening uitgevoerd kan worden. Zowel in theoretische als in meer praktijkgerichte informatica staan die bekend als on-line en off-line. We verhelderen dit voor de gevolgde procedures bij taalherkenning; het analogon voor het berekenen van functies laat zich dan soortgelijk definiëren.

Een RAM kan een taal op de volgende twee manieren herkennen.

- 1) Een invoersymboolrij $s = a_1 a_2 \dots a_n$ wordt op de invoerband geplaatst met a_i in vakje i , $1 \leq i \leq n$. Het vakje $n+1$ bevat het, einde- invoersymbool $\emptyset$. s wordt geaccepteerd door het RAM programma P als P s geheel leest, en ook het einde - invoersymbool $\emptyset$, vervolgens een 1 schrijft in het eerste vakje van de uitvoerband en dan stopt. De door P geaccepteerde taal is dan de verzameling geaccepteerde invoersymboolrijen.
 - 2) Een, potentieel oneindige, invoersymboolrij $s = a_1 a_2 \dots a_i \dots$ wordt op de invoerband geplaatst met a_i in vakje i , voor $i \geq 1$. $a_1 a_2 \dots a_i$, $i \geq 1$, wordt door het RAM programma P geaccepteerd, als P na voor het eerst a_i gelezen te hebben, en precies vóór de eerste maal a_{i+1} te lezen, een 1 schrijft in het i -de vakje van de uitvoerband. De geaccepteerde taal bestaat dan uit alle geaccepteerde symboolrijen.
- De onder 2) beschreven manier van accepteren wordt *on-line* genoemd, alle andere manieren (dus ook 1)) worden *off-line* genoemd.

Als voorbeeld van een RAM programma berekenen we, *off-line*, de functie $f(n) = n^n$ voor $n \geq 1$ en $f(n) = 0$ voor $n \leq 0$, n een geheel getal.

```

      READ  R1
      IF R1 < 1 THEN L1 ELSE L2
L1: GOTO L6
L2: R2 ← R1
      R3 ← 1
L3: R1 ← R1 - 1
      R4 ← R2
L4: R4 ← R4 - 1
      R5 ← R3 + R5
      IF R4 < 1 THEN L5 ELSE L4
L5: R3 ← R5
      R5 ← 0
      IF R1 < 1 THEN L6 ELSE L3
L6: WRITE R3
L7: ACCEPT

```

FIGUUR 3. RAM programma

De looptijd van een RAM programma wordt bepaald met gebruikmaking van een functie ℓ die de kosten in tijdeenheden per uitgevoerde instructie aangeeft. Deze functie ℓ van de gehele getallen in de natuurlijke getallen is,

per instructie, aangegeven in Figuur 2. Verschillende keuzen voor ℓ kunnen leiden tot geheel verschillende looptijden voor een en hetzelfde programma. De slechtste-geval tijdcomplexiteit (of alleen maar tijdcomplexiteit) van een RAM programma is de functie $T(n)$ die het maximum is, genomen over alle invoer van grootte n , van de som van de "tijden" zoals die per uitgevoerde instructie gevegd worden. De meest voorkomende waarden voor ℓ zijn $\ell(x)=1$ voor alle x : het *uniforme kosten criterium*; en $\ell(x)$ is gelijk aan de lengte van x uitgeschreven als binair getal: het *logaritmische kosten criterium*. Voor het geheugenruimteverbruik van een RAM programma, de *ruimte complexiteit* $S(n)$, geldt een soortgelijke kostenfunctie ℓ als in Figuur 2, en wordt de som genomen over alle registers, inclusief de accumulator R_0 , van $\ell(x_i)$, waarbij x_i het grootste getal is dat op enig moment van de berekening in register R_i , $i \geq 0$, is opgeslagen.

Als we de tijdcomplexiteit van het RAM programma van Figuur 3 bekijken zien we het volgende. De tijdskosten worden bepaald door een buitenlus en een binnenlus. Elke maal dat we de buitenlus met label L_3 doorlopen, wordt de binnenlus met label L_4 n maal doorlopen. De buitenlus zelf wordt ook n maal doorlopen, en zodoende komen we op een tijdcomplexiteit van $O(n^2)$ onder het uniforme kosten criterium. De bepaling van de logaritmische tijdcomplexiteit is lastiger. De kosten om de i -de buitenlus te doorlopen zijn gelijk aan ($1 \leq i \leq n$):

$$\begin{aligned} &\ell(n-i) + \ell(n) + \text{kosten binnenlus met } n^i \text{ in } R_3 \\ &+ \ell(n^{i+1}) + \ell(n-i) + \text{constante} \cdot \ell(n). \end{aligned}$$

De kosten om de j -de binnenlus te doorlopen in de i -de buitenlus worden gegeven door ($1 \leq i \leq n, 1 \leq j \leq n$):

$$\ell(n-j) + \ell(j \cdot n^i) + \ell(n-j).$$

In totaal krijgen we dus:

$$\begin{aligned} T(n) &\in O\left(\sum_{i=1}^n (\ell(n-i) + \ell(n) + \ell(n^{i+1}) + \sum_{j=1}^n (\ell(n-j) + \ell(j \cdot n^i)))\right) \\ &= O\left(\sum_{i=1}^n ((i+1) \log n + \sum_{j=1}^n (i \log n + \log j))\right) \\ &= O\left(\sum_{i=1}^n ((i+1) \log n + i n \log n)\right) = O(n^3 \log n). \end{aligned}$$

Onder het uniforme kostenkriterium is de ruimtecomplexiteit $O(1)$, omdat we maar 6 registers gebruiken; terwijl onder het logaritmische kostenkriterium de ruimtecomplexiteit $O(n \log n)$ is, daar het grootste getal in een der 6 registers, gedurende de berekening, niet groter is dan n^n . Voor het programma van Figuur 2 geldt daarom:

	Uniforme kosten	Logaritmische kosten
Tijdcomplexiteit	$O(n^2)$	$O(n^3 \log n)$
Ruimtecomplexiteit	$O(1)$	$O(n \log n)$

De uniforme kostenmaat is realistisch voor het onderhavige programma, indien aangenomen kan worden dat een enkel computerwoord getallen van orde n^n kan bevatten. Als n^n groter is, dan wat in een enkel computerwoord past, is de logaritmische kostenmaat realistisch, aangezien het bekend is dat twee getallen i en j kunnen worden opgeteld in tijd $O(\log i + \log j)$.

Het uniforme kostenkriterium is gebaseerd op de aanname dat het uitvoeren van iedere RAM instructie één tijdeenheid kost, en dat ieder register in één geheugen eenheid past. Het logaritmische kostenkriterium houdt rekening met de beperkte lengte van een echt computerwoord. Daar registers willekeurig grote getallen kunnen bevatten, neemt deze kostenmaat in aanmerking dat er $\lfloor \log n \rfloor + 1$ bits nodig zijn om een getal n in een register te representeren; verder dat de kosten verbonden met de uitvoering van een instructie proportioneel zijn met de lengte van de operanden. De volgende stelling geeft de relatie tussen de verschillende tijdskosten aan, cf. HARTMANIS [1971].

STELLING 1. (i) Een berekening die op een RAM onder het logaritmische kostenkriterium tijd $T(n)$ vergt, kan door een RAM onder het uniforme kostenkriterium in tijd $O(T(n))$ uitgevoerd worden.

(ii) Een berekening die op een RAM onder het uniforme kostenkriterium tijd $T(n)$ vergt, kan door een RAM onder het logaritmische kostenkriterium in tijd $O(T(n)^2)$ uitgevoerd worden.

Met betrekking tot de ruimtecomplexiteit is een zinvol analogon voor Stelling 1 niet te geven, omdat elke RAM berekening uitgevoerd kan worden door een RAM die slechts twee registers gebruikt, MINSKY [1961]. Daarom hebben alle problemen een ruimtecomplexiteit $O(1)$ onder het uniforme

kostenkriterium! Een bewijs voor dit feit wordt gegeven door middel van een simulatie algoritme die buitengewoon grote getallen genereert. Op een echte computer echter, zouden deze getallen niet in één computerwoord passen, maar zouden vele computerwoorden nodig zijn om de inhoud van de betrokken twee registers te bevatten. Daarom zullen we de ruimtecomplexiteit altijd meten onder het logaritmische kostenkriterium.

2.2. P VERSUS NP

Wanneer wij ons beperken tot algoritmen, met een door een polynoom begrensde tijdcomplexiteit, doet het er in het algemeen niet toe welk (deterministisch) berekeningsmodel gekozen wordt. Om deze observatie te formaliseren, definiëren we P als de klasse van talen die door een RAM in polynomiale tijd herkend kunnen worden. Beide kostencriteria leveren dan, als gevolg van Stelling 1, dezelfde klasse op. De invariantie van P onder verandering van machine model, tot in feite elk redelijk deterministisch model van een computer, is het eerst onderkend door COBHAM [1964]. De laatste tijd ontvangt dit fenomeen veel aandacht, daar de klasse P beschouwd wordt als een benadering van de klasse van praktisch uitvoerbare problemen.

Het tot dusver beschouwde model wordt deterministisch genoemd, omdat er in ieder stadium van de berekening een eenduidige volgende stap is. Alle echte computers zijn deterministisch. Er zijn echter goede redenen om modellen te beschouwen waarbij de aanname verzwakt wordt tot het toelaten van eindig veel mogelijke volgende stappen. Modellen waarin een machineconfiguratie in één stap mag overgaan in eindig veel nieuwe configuraties, dit in tegenstelling tot het geval met een enkele volgende configuratie, worden *niet-deterministisch* genoemd. De *niet-deterministische RAM* is net zo gedefinieerd als een deterministische RAM, behalve dat we niet meer eisen dat de instructies unieke labels hebben: twee of meer opdrachten mogen dezelfde label bezitten. Een berekening op zo'n niet-deterministische RAM loopt hetzelfde als een berekening op een deterministische RAM tot een sprong opdracht naar een label L ontmoet wordt, en twee of meer instructies deze label hebben. De invoer wordt geaccepteerd, of leidt tot een functiewaarde, als er tenminste één toegelaten volgorde van uitgevoerde instructies is die eindigt met de ACCEPT instructie. Merk op, dat als een niet-deterministische machine een toegestane volgorde van instructies afwerkt, die niet in de ACCEPT instructie eindigt, dit niet betekent dat de invoer in een of ander opzicht afgewezen wordt. In feite geeft dit geen informatie, daar een andere

volgorde wel tot de ACCEPT instructie kan leiden. Onder deze afspraak, van wat acceptatie door een niet-deterministische RAM inhoudt, kunnen we de tijdcomplexiteit als volgt definiëren.

Zij M een niet-deterministische RAM, A een verzameling natuurlijke getallen en $T(n)$ een functie op de natuurlijke getallen.

M accepteert A onder het kosten criterium ℓ in tijd $T(n)$ als

- (i) M precies die invoer accepteert die tot A behoort;
- (ii) er voor iedere $x \in A$ een berekening (volgorde van keuzen) van M met invoer x is, die tot de ACCEPT instructie leidt, en die ten hoogste $T(n)$ tijdeenheden gebruikt onder het ℓ kosten criterium, waarbij n de lengte van x is.

Voor het geheugenverbruik $S(n)$ onder het kosten criterium ℓ gelden soortgelijke definities.

Als voorbeeld, stel dat we de priemaliteit van een getal x willen bepalen. Een niet-deterministische RAM gist twee factoren x_1 en x_2 en berekent $x_1 \cdot x_2$. Is $x_1 \cdot x_2 = x$ voor een keuze van x_1 en x_2 , $x_1, x_2 \neq 1$, dan is x samengesteld. Zodoende kan een niet-deterministische RAM de verzameling van samengestelde natuurlijke getallen in $O(n^2)$ tijd accepteren onder het logaritmische kosten criterium. Alle bekende algoritmen hiervoor kosten op een deterministische RAM meer dan polynomiale tijd. (In de voordracht van H.W. LENSTRA jr. zal het bepalen van priemaliteit nader aan de orde komen.)

Hoe verhoudt zich nu het geheugen- en tijdverbruik tussen deterministische en niet-deterministische RAMs? Iedere niet-deterministische RAM kan gesimuleerd worden door een deterministische RAM. Alles wat de deterministische RAM moet doen is systematisch alle toegestane berekeningsvolgorden van i instructies aflopen, voor $i = 1, 2, \dots$. Zo'n simulatie kost zeer veel tijd- en geheugenruimte; deterministische ruimte $k^{S(n)}$ voor niet-deterministische ruimte $S(n)$, en deterministische tijd $k^{T(n)}$ voor niet-deterministische tijd $T(n)$, waarbij k een passende constante is. Wat betreft ruimtecomplexiteit kunnen we met een methode van SAVITCH [1970] een beter resultaat krijgen:

STELLING 2. Als A door een niet-deterministische RAM in ruimte $S(n)$ wordt geaccepteerd, dan kunnen we een deterministische RAM vinden die A in ruimte $O(S(n)^2)$ accepteert als $S(n) \geq \log n$.

Voor het tijdsverbruik is echter geen betere methode bekend dan het deterministisch simuleren van een niet-deterministische machine, met tijd-

complexiteit $T(n)$, in $O(k^{T(n)})$ tijd, $k > 1$. Men vermoedt dat dit het best mogelijke is. Dit vermoeden wordt vaak geuit in termen van polynomiaal begrensde programma's. Naar analogie van de klasse P definiëren we NP als de klasse talen, die in niet-deterministische polynomiale tijd geaccepteerd worden. Zoals met P leiden in het algemeen alle redelijke niet-deterministische modellen tot dezelfde klasse NP. Bij definitie geldt $P \subseteq NP$. Men vermoedt $P \neq NP$. Er is veel belangstelling voor deze problematiek, omdat vele belangrijke (praktijk) problemen tot de klasse NP blijken te behoren, en het zeer nuttig zou zijn om deterministische polynomiale algoritmen voor zulke problemen te hebben. Werk van COOK [1971], KARP [1972,1975], en anderen heeft aangetoond dat als $P \neq NP$ vele, in de praktijk voorkomende, problemen buiten P liggen en dus geen polynomiaal begrensde algoritme toelaten. Zulke problemen vinden we in dat geval bijvoorbeeld in de NP-compleete problemen waarvoor geldt: (i) het probleem ligt in NP, en (ii) als het probleem in P ligt dan is $P = NP$. Voor de goede orde: als voor een probleem (i) en (ii) geldt dan hoeft het nog niet NP-compleet te zijn. Zie voor de P versus NP kwestie ook van LEEUWEN's bijdrage tot dit colloquium.

2.3. VERMENIGVULDIGINGS RAMs

In de voorgaande definities stonden we als aritmetische operaties alleen optellen en aftrekken toe, maar niet vermenigvuldigen of delen. Veel resultaten blijven ongewijzigd onder het toevoegen van de vermenigvuldigingsoperatie, zoals bijvoorbeeld die met betrekking tot het geheugenruimteverbruik. Voor de tijdgrenzen ligt de zaak echter anders. Onder het logaritmische kostenkriterium wordt het karakter van de resultaten echter niet aangetast. In sommige gevallen wordt de looptijd van een simulatieprogramma (op de oude RAM) het kwadraat, of een andere kleine macht, van die van het programma op de nieuwe machine, maar modellen die polynomiaal gerelateerd waren blijven dat, ook als we vermenigvuldiging toestaan. In het bijzonder, als we de klassen P en NP definiëren voor de nieuwe machines, krijgen we dezelfde twee klassen als met de oude machines.

Als we het uniforme kostenkriterium hanteren wordt de zaak geheel anders. HARTMANIS en SIMON [1974] tonen aan dat, onder het uniforme kostenkriterium, een kleine verandering in aritmetische operaties de berekeningskracht van het RAM model opvallend lijkt te veranderen. Definieer nu een *vermenigvuldigings RAM* (multiplication RAM) of *MRAM* als een RAM met de volgende toegevoegde operaties:

$$R_i \leftarrow a * b \text{ en } R_i \leftarrow a \text{ BOOL } b,$$

waarbij $a * b$ een vermenigvuldiging voorstelt, en $a \text{ BOOL } b$ een willekeurige bitgewijze boolse operatie. (Om BOOL uit te voeren beschouwen we getallen als rijen bits en vergeten het teken en de leidende nullen. Bijvoorbeeld, $\text{BOOL} = \text{XOR}$ (exclusive or) en $1011 \text{ XOR } 101 = 1110$) Beide instructies hebben uitvoeringstijd $\ell(a) + \ell(b)$; onder het uniforme kostenkriterium: $O(1)$. Evenals bij RAMs, onderscheiden we bij MRAMs deterministische en niet-deterministische modellen.

STELLING 3. *Onder het uniforme kostenkriterium geldt voor MRAMs:*

- (i) *A wordt door een niet-deterministische MRAM in polynomiale tijd geaccepteerd dan en slechts dan als A door een deterministische MRAM in polynomiale tijd wordt geaccepteerd.*
- (ii) *A wordt door een RAM (met logaritmische kosten) in polynomiale ruimte geaccepteerd dan en slechts dan als A door een MRAM in polynomiale tijd wordt geaccepteerd.*

Merk op, dat in (ii) het al dan niet deterministisch zijn van de machines buiten beschouwing blijft. Dit is een gevolg van Stelling 2 en Stelling 3(i): deterministische en niet-deterministische polynomiale ruimte zijn equivalent. Zoals gezegd, wordt algemeen aangenomen $P \neq NP$, als P en NP gedefinieerd worden op andere modellen dan MRAMs met uniforme kosten. Volgens Stelling 3 accepteren MRAMs (met uniforme kosten) alle talen in NP in deterministisch polynomiale tijd. Daarom lijkt het dat, onder de polynomiale tijd restrictie, deterministische MRAMs krachtiger zijn dan deterministische RAMs. Omdat een MRAM meer op een gewone computer lijkt dan een RAM, lijkt dit resultaat tegenstrijdig met eerdere uitspraken. Immers, als MRAMs krachtiger zijn en ook realistischer, waarom dan niet MRAMs gebruikt als standaard model. Analyse van het bewijs van Stelling 3 laat zien dat dit onverstandig zou zijn. Het simulatie (MRAM) model dat hierbij gebruikt wordt genereert zeer grote getallen, en daar het uniforme kostenmodel gehanteerd wordt, kunnen deze getallen met elkaar vermenigvuldigd worden in één machine stap. Bij implementatie van het algoritme op een echte machine zouden deze vermenigvuldigingen vele machine stappen kosten, aangezien deze zeer grote getallen met volledige precisie vermenigvuldigd moeten worden. Om te vermijden schijnbaar efficiënte algoritmen te schrijven, die niet geïmplementeerd kunnen worden, moeten we daarom MRAMs met logaritmische

kosten of de eerdere modellen gebruiken. Niettemin, is Stelling 3 een belangrijk resultaat. Zij geeft ons een alternatieve, verrasende en bruikbare karakterisering van de klasse van talen herkenbaar in polynomiale ruimte. Zij verschaft ons ook een argument voor het belang van het hanteren van het logaritmische kostenkriterium, en toont aan dat er een wezenlijk verschil bestaat tussen het uniforme en het logaritmische kostenkriterium.

2.4. ABSTRACTIES VAN RAMs

In veel situaties zijn we beter af, als we van sommige aspecten van RAMs abstraheren, en andere verwaarlozen.

1) RECHTLIJNIGE PROGRAMMA's (straight line programs)

Voor veel problemen is het zinvol om onze aandacht te beperken tot RAM programma's waarbij de vertakkingsinstructies slechts gebruikt worden om een rij instructies een aantal malen, proportioneel met de lengte van de invoer n , te herhalen. In dat geval kunnen we het programma, voor iedere n , "uitrollen" door deze rij instructies het gepaste aantal malen te dupliceren. Zulk een handelswijze mondt uit in een aantal rechlijnige (lus-vrije) programma's met toenemende lengte voor grotere n . Zie ook de bijdrage van P. van EMDE BOAS in dit colloquium, en het verband met onachtzame Turing machines in sectie 2.6.

Het uitrollen van een programma in een rechte lijn stelt ons in staat de vertakkingsinstructies overboord te zetten. Via soortgelijke redeneringen kunnen we indirecte adressering, lees, schrijf en andere instructies verwaarlozen, en tenslotte een repertoire overhouden bestaande uit:

$$\begin{aligned} x &\leftarrow y + z, \\ x &\leftarrow y - z, \\ z &\leftarrow y * z, \\ z &\leftarrow y/z, \\ x &\leftarrow i, \end{aligned}$$

waarbij x, y en z symbolische adressen of variabelen zijn, en i een constante is. Een rechtljnig programma kent twee daarmee geassocieerde verzamelingen: de invoer-variabelen en de uitvoer-variabelen. De functie die door een rechtljnig programma wordt berekend is een verzameling waarden van de uitvoer-variabelen uitgedrukt in termen van de invoer-variabelen. Een mooi

voorbeeld is het bepalen van

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0.$$

De invoer-variabelen zijn de waarden van $a_0, a_1, \dots, a_n$ en een onbekende x , de uitvoer is de waarde van $p(x)$ voor aangeboden x . Horner's schema evalueert $p(x)$ als volgt:

$$\begin{aligned} n = 1 : & a_1 x + a_0, \\ n = 2 : & (a_2 x + a_1) x + a_0, \\ n = 3 : & ((a_3 x + a_2) x + a_1) x + a_0, \text{ etc.} \end{aligned}$$

Met deze uitdrukkingen corresponderen de volgende rechte lijn programma's:

$n = 1$	$n = 2$	$n = 3$
$t \leftarrow a_1 * x$	$t \leftarrow a_2 * x$	$t \leftarrow a_3 * x$
$p \leftarrow t + a_0$	$t \leftarrow t + a_1$	$t \leftarrow t + a_2$
	$t \leftarrow t * x$	$t \leftarrow t * x$
	$p \leftarrow t + a_0$	$t \leftarrow t + a_1$
		$t \leftarrow t * x$
		$p \leftarrow t + a_0.$

Voor iedere n bestaat er zo een rechtlijnig programma van $2n$ stappen die een algemene n -e graads polynoom evalueert.

STELLING 4. *n vermenigvuldigingen en n optellingen zijn noodzakelijk en voldoende voor het bepalen van de waarde van een n -de graads polynoom, als de coëfficiënten gegeven zijn als invoer-variabelen, door een rechtlijnig programma. Dus, HORNER's schema is optimaal voor rechtlijnige programma's.*

2) BITGEWIJZE BEREKENINGEN

Het rechtlijnige programma model is duidelijk gebaseerd op het uniforme kosten model. Zoals we (in bijvoorbeeld sectie 2.3) gezien hebben, is dit niet altijd redelijk, tenzij we aan kunnen nemen dat de berekende getallen altijd van redelijke grootte zijn. Een eenvoudige wijziging van het model geeft de logaritmische kosten. Het resulterende bitgewijze berekeningsmodel is gelijk aan het rechtlijnige programma behalve:

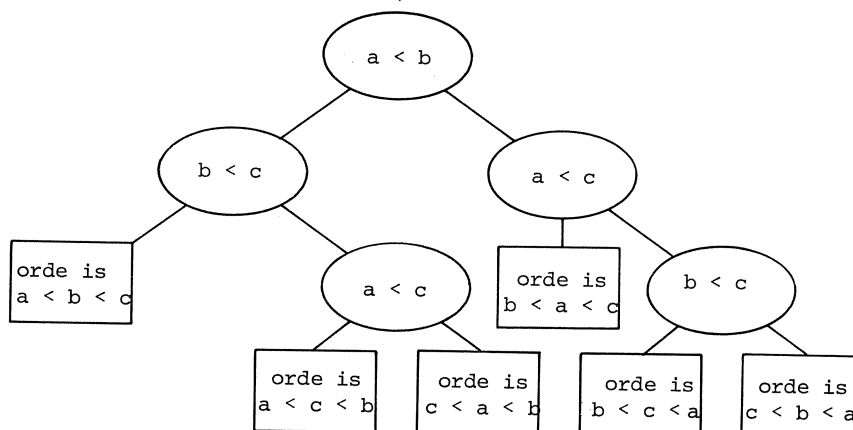
- (i) Alle variabelen zijn bits.
- (ii) De gebruikte operaties zijn logisch in plaats van aritmetisch. Bijvoorbeeld; $\wedge$ (en), $\vee$ (of) $\oplus$ (exclusief of), $\neg$ (niet).

Onder dit model kosten aritmetische operaties op gehele getallen i en j tenminste $\log |i| + \log |j|$ stappen, hetgeen de logaritmische kostenmaat weer spiegelt. De kosten om twee getallen van n bits te vermenigvuldigen zijn bijv.: $O(1)$ voor rechtlijnige programma's met uniforme kosten, $O(\log n)$ voor rechtlijnige programma's met logaritmische kosten, terwijl het beste bekende algoritme voor het bitgewijze model $O(n \cdot \log n \cdot \log \log n)$ oplevert.

Een nuttig aspect van het bitgewijze model ligt in het verband met *logische netwerken*. Rechtlijnige programma's met bitgewijze invoer en operaties kunnen één-éénduidig afgebeeld worden op logische netwerken die boolse functies uitrekenen. Het aantal door het programma uitgevoerde stappen is gelijk aan het aantal elementen in het netwerk. Omdat het vaak mogelijk is het minimaal nodige aantal elementen van een netwerk voor het uitrekenen van een gegeven functie te bepalen, kunnen we zo ook ondergrenzen bepalen aan de tijdcomplexiteit voor het bitgewijze model. Zie SAVAGE [1976].

3) BESLISSINGSBOMEN (decision trees)

Bij de voorgaande RAM-abstracties werden alleen stappen die berekeningen inhielden geteld. Voor veel berekeningen is het meer realistisch het aantal vertakkingen als dominante faktor te beschouwen, bijvoorbeeld bij sorteren. Daarom bekijken we nu een model waarbij alle stappen twee-wegs-vertakkingen, gebaseerd op het vergelijken van twee grootheden, zijn. De gewone representatie voor een programma, bestaande uit zulke vertakkingen, is een binaire boom: de beslissingsboom. Iedere interne knoop stelt een beslissing voor. De test corresponderend met de wortel wordt het eerst uitgevoerd, waarna de test in de, door de uitslag van de voorgaande test aangewezen, zoon uitgevoerd wordt. In het algemeen wordt door een berekening een pad in de boom doorlopen, net zolang tot een blad is bereikt. De gewenste uitvoer is hier dan beschikbaar. Figuur 4 geeft een beslissingsboom voor een programma dat de getallen a, b en c sorteert. Testen worden voorgesteld als knopen; de rechterzoon wordt bezocht als het antwoord bevestigend is, en de linkerzoon als het ontkennend luidt.



Figuur 4. Beslissingsboom

De tijdcomplexiteit van een beslissingsboom is de hoogte van de boom als functie van de grootte van het probleem. Vaak willen we het maximale aantal vergelijkingen weten, dat gemaakt moet worden om de weg te vinden van de wortel tot een blad. Het aantal knopen in de boom kan ver uitstijgen boven de hoogte. Een beslissingsboom om n getallen te sorteren moet tenminste $n!$ bladeren hebben, ofschoon de hoogte van de boom maar $n \log n$ hoeft te zijn. Zulke overwegingen leiden tot een ondergrens: $\Omega(n \log n)$ vergelijkingen zijn noodzakelijk om n items te sorteren.

2.5. DE TURING MACHINE

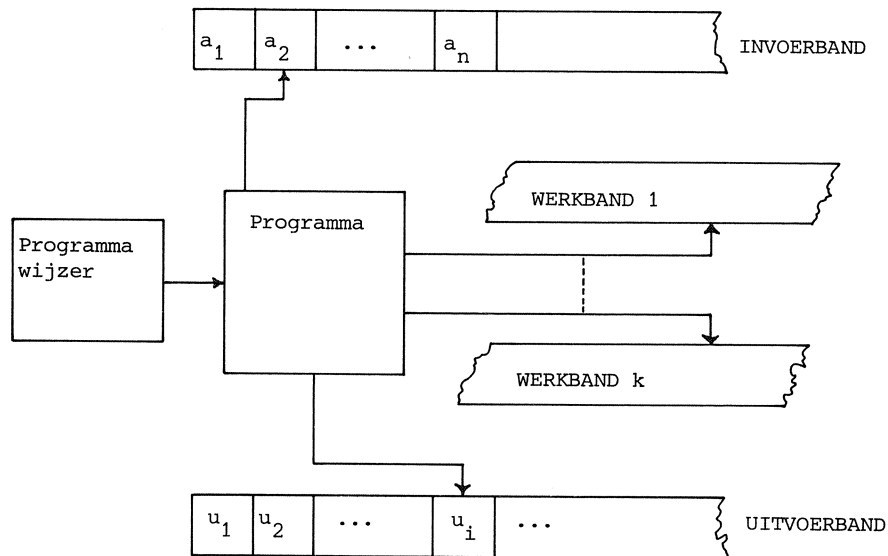
Om te bewijzen dat een gegeven functie een zekere minimale berekeningstijd vergt, hebben we een model nodig dat even algemeen is als de RAM, primitiever en met een beperkt instructie repertoire, maar toch bijna even snel. Voor "bijna" gebruiken we "polynomiaal gerelateerd". Twee functies f_1, f_2 zijn polynomiaal gerelateerd als er polynomen $p_1(x)$ en $p_2(x)$ zijn, zodanig dat $f_1(n) \leq p_1(f_2(n))$ en $f_2(n) \leq p_2(f_1(n))$ voor alle waarden van n . Er bestaan, zij het niet veel, resultaten die ondergrenzen geven aan de berekeningstijd van problemen door Turing machines. Als zulke berekeningsgrenzen exponentieel zijn, dan gelden zij in essentie ook voor modellen met polynomiaal gerelateerde looptijden. De, in het onderstaande gedefinieerde, meerbands (multitape) Turing machine simuleert een RAM, met tijd

complexiteit $T(n)$ onder het logaritmische kostenkriterium, in tijd $O(T(n)^2)$; en vice versa in $O(T(n) \log T(n))$, dus ten koste van nog niet een kwadratering van de looptijd. Logaritmische RAMs en Turing machines zijn polynomiaal gerelateerd.

Een k-band Turing machine bestaat uit een eindig programma (finite-state control) verbonden met een invoerband waarvan alleen gelezen kan worden, een uitvoerband waarop alleen geschreven kan worden, en k werkbanden waarvan gelezen en waarop geschreven kan worden. De banden zijn verdeeld in vierkantjes. Elk vierkantje kan een symbool uit een eindig alfabet bevatten. De geheugenbanden zijn in beide richtingen onbegrensd. De invoer- en uitvoerband zijn als bij de RAM. Figuur 5 toont ons dit apparaat. De werking van de machine is als volgt. Bij iedere stap begint de machine te lezen welke symbolen zich bevinden in de vakjes onder de koppen op de invoerband en de k werkbanden. Afhankelijk van de gelezen symbolen, en de programma instructie waarheen de programmawijzer wijst, zal de machine in één stap het volgende doen.

- (a) De symbolen in de gelezen vakjes op de werkbanden vervangen door andere symbolen (de machine kan een symbool ook ongewijzigd laten).
- (b) De koppen op de invoerband en de werkbanden een vakje naar links, naar rechts, of niet, verschuiven.
- (c) Eventueel een symbool op de uitvoerband schrijven; in dit geval wordt de kop op deze band een vakje naar rechts geschoven zodat de machine klaar is om een volgend symbool te schrijven.
- (d) De programmawijzer veranderen.

De machine begint met een daartoe aangewezen start-instructie in het programma. Op dat moment staat de invoer op de invoerband, de kop op die band leest het meest linkse symbool van de invoer, alle vakjes op de werkbanden en uitvoerband bevatten speciale blanco symbolen, en de kop op de uitvoerband staat op het meest linkse vakje. Een Turing machine kan, net zo als een RAM, een taal herkennen, doordat het programma speciale ACCEPT en REJECT instructies bevat; of hij kan een functie van de invoer naar de uitvoer uitrekenen. Een Turing machine berekent een functie f van invoersymboolrijen naar uitvoersymboolrijen als, gestart met invoer w , de machine uiteindelijk een ACCEPT-instructie ontmoet met $f(w)$ geschreven op de uitvoerband. We kunnen de conventie gebruiken dat de machine dan stopt. Als f partieel is dan ontmoet de machine, voor de waarden waarvoor f niet gedefinieerd is, nooit de ACCEPT instructie en termineert niet.



Figuur 5. Turing machine met k werkbanden

Deze conventies zijn voldoende voor off-line berekeningen. Voor een on-line berekening wordt de machine gestart op invoer $z = wv$, en juist vóór het eerste symbool van v gelezen wordt, doch na het lezen van het laatste symbool van w , schrijft de machine $f(w)$ op de uitvoerband, voor elke partitie van z in w en v . Voor machines gebruikt als taalherkenners nemen we aan dat het waardebereik van f gelijk is aan $\{0,1\}$, waarbij $f(w) = 1$ als w geaccepteerd wordt, en $f(w) = 0$ als w niet geaccepteerd wordt; dit laatste voor het geval de machine deterministisch is en altijd stopt c.q. doorleest.

De Turing machine is het oudste berekeningsmodel. Zij werd door TURING [1936] ingevoerd als een formalisatie van het begrip berekenbare functie, en speelt een vooraanstaande rol in de theorie der recursieve functies. Turing machines berekenen precies alle partieel recursieve functies, en accepteren precies alle recursief opsombare verzamelingen. Door hun eenvoud en overzichtelijkheid zijn zij ook binnen de complexiteitstheorie waardevol, bijvoorbeeld, voor het bepalen van ondergrenzen aan de complexiteit van problemen, en het toetsen en onderzoeken van nieuwe begrippen. Net zoals bij de RAM worden de maten voor berekeningstijd en geheugenverbruik gedefinieerd

als functies van de lengte van de invoer. We geven de definities voor het gebruik van de machine als taalherkenner; voor machines die functies uitrekenen zijn de definities analoog.

Zij M een Turing machine, A een verzameling woorden over het invoeralfabet van M , en $T(n)$ en $S(n)$ functies over de natuurlijke getallen.

(1) M accepteert A binnen tijd $T(n)$ als

- (i) M precies de woorden in A accepteert, en
- (ii) voor ieder woord w in A de berekening van M op w $T(n)$ of minder stappen vergt. Hierbij is n de lengte van w .

(2) M accepteert A binnen geheugenruimte $S(n)$ als

- (i) M precies de woorden in A accepteert, en
- (ii) voor ieder woord w in A de berekening van M op w niet meer dan $S(n)$ vakjes op werkbanden verg ($n=|w|$), dat wil zeggen, tijdens de berekening op w wordt een totaal van ten hoogste $S(n)$ vakjes op de werkbanden gelezen.

Merk op, dat ook hier alleen het gebruik van hulpbronnen gemeten wordt voor die gevallen waarbij de invoer geaccepteerd wordt.

Duidelijk is, dat het verschil tussen Turing machines en RAMs zit in het geheugengebruik: RAMs modelleren berekeningen die in het snelgeheugen passen, terwijl Turing machines meer berekeningen modelleren met betrekking tot het gebruik van grootschalige lineair geordende achtergrond geheugens zoals magnetische banden of schijven. In beide gevallen betreft zo'n vergelijking natuurlijk een geïdealiseerde voorstelling van zaken, maar geeft toch inzicht in de mogelijkheden en beperkingen van zulke geheugens. Vergelijking van beide typen leert ons:

STELLING 5. *Stel $S(n) \geq n$. Een verzameling A , over een eindig alfabet, wordt door een Turing machine in geheugenruimte $S(n)$ geaccepteerd, dan en slechts dan als A door een RAM (met logaritmische kosten) geaccepteerd wordt in geheugenruimte $O(S(n))$.*

STELLING 6. *Stel $T(n) > n$.*

- (i) *Als A door een RAM met logaritmische kosten geaccepteerd wordt in tijd $T(n)$ dan wordt A door een Turing machine geaccepteerd in tijd $O(T(n)^2)$.*
- (ii) *Als A door een Turing machine geaccepteerd wordt in tijd $T(n)$ dan wordt A door een RAM met logaritmische kosten geaccepteerd in tijd $O(T(n) \log T(n))$.*

STELLING 7. Stel $T(n) > n$.

- (i) Als A door een RAM met uniforme kosten geaccepteerd wordt in tijd $T(n)$ dan wordt A door een Turing machine geaccepteerd in tijd $O(T(n)^3)$.
- (ii) Als A door een Turing machine geaccepteerd wordt in tijd $T(n)$ dan wordt A door een RAM met uniforme kosten geaccepteerd in tijd $O(T(n))$.

Uit de bovenstaande stellingen, zie ook COOK en RECKHOW [1973], volgt dat het verbruik van geheugenruimte stabiel is onder wisselingen tussen Turing machines en RAMs. (De uniforme kosten RAM blijft hierbij buiten beschouwing, vanwege het onrealistische gegeven dat hierbij alle berekeningen uitgevoerd kunnen worden met gebruik van twee registers.) Wat betreft berekeningstijd is de situatie anders. Alle modellen geven verschillende rekentijden voor eenzelfde algoritme. Al deze rekentijden zijn echter polynomiaal gerelateerd (behalve voor MRAMs). Daarom is de klasse P hetzelfde voor Turing machines en RAMs, en evenzo, de klasse NP hetzelfde voor niet-deterministische RAMs, en niet-deterministische Turing machines, waarbij de laatste machine de voor de hand liggende uitbreiding is van de beschreven Turing machine.

2.6. ONACHTZAAMHEID EN ECHTE TIJD

In deze sectie worden soorten algoritmen bekeken, die in veel computer toepassingen gangbaar zijn. Als berekeningsmodel wordt de k-bands Turing machine gebruikt. Deze misschien onrealistische abstractie biedt de mogelijkheid om resultaten te verkrijgen, die inzicht geven voor het gedrag van dergelijke algoritmen op machines die grote hoeveelheden gegevens door middel van een achtergrondgeheugen manipuleren, of die evenzo gelden voor modellen als RAMs.

1. Echte tijd (real-time)

Veel computer toepassingen betreffen 'on-line processing' en 'real-time control'. On-line processing betreft het verwerken van een reeks gegevens, elke gegeven onmiddellijk als het aangeboden wordt. Als nu ook de computer antwoord geeft, of een andere daarvoor in aanmerking komende actie uitvoert, na ontvangst van een gegeven en voordat het volgende aangeboden wordt, komt dit begrip overeen met de eerder gedefinieerde on-line berekeningen. Denk hierbij aan bijvoorbeeld interactief computergebruik. Is het aantal (door de computer uitgevoerde) stappen tussen aanbod van een gegeven en het

resultaat van de verwerking hiervan gedurende de verwerking van een willekeurig lange reeks, beperkt door een constante, dan spreken we van een echte-tijd berekening. Dit is het geval bij real-time control, zoals gebruikt bij het besturen van processen, etc. Echte-tijd berekeningen vormen een der oudste onderwerpen van studie in de complexiteitstheorie, bijvoorbeeld YAMADA [1960], RABIN [1963]. Daar de tijdgrens vastligt, zijn de onderzochte vraagstukken meer in de trant van: zijn machines van verschillende specificatie ook verschillend in berekeningskracht. RABIN [1963] toonde aan dat echte-tijd Turing machines met 2 werkbanden krachtiger zijn dan echte-tijd Turing machines met 1 werkband. AANDERAA [1974] generaliseerde dit resultaat tot $k+1$ versus k werkbanden. Ofschoon fysische beperkingen het efficiënt werken van k koppen op één magneetband onmogelijk kunnen maken, laten sommige algoritmen zich makkelijker formuleren voor een Turing machine met één werkband waarop k koppen rekenen. SLISENKO [1970] bewees dat zo'n veelkoppige Turing machine palindromen (zonder gemarkeerd centrum) in echte-tijd kunnen herkennen. FISCHER, MEYER en ROSENBERG [1972] lieten zien dat, door slim programmeren, een k -kops Turing machine in echte-tijd gesimuleerd kan worden door een $(11k-9)$ -bands Turing machine. VITÁNYI [1979, 1980a] bewees dat $(k+1)$ -kops Turing machines méér kunnen in echte tijd dan k -kops Turing machines. In VITÁNYI [1980b] wordt het oude vermoeden, dat 2 koppen op één band meer kunnen dan 2 éénkoppige banden, als geheugen voor een Turing machine die in echte tijd werkt, wel erg aannemelijk gemaakt. Het belang van bovenstaande onderzoeken, betreffende verschillen in kracht tussen verschillende gespecificeerde machines die in echte-tijd werken, ligt hierin, dat eerder bestaande informatie theoretische en diagonalisatie technieken te grof zijn om te werken; nieuw ontwikkelde technieken, bestaande uit precieze analyse van de berekeningsaspecten die beschikbaar zijn, geven een verdergaand inzicht in de aard van berekeningen.

2. Onachtzaamheid (obliviousness)

We noemen een $(k$ -bands) Turing machine onachtzaam als de bewegingen van de invoer-, de uitvoer-, en werkbandkoppen functies zijn van de verstreken tijd, onafhankelijk van de invoer. We zouden bij zo'n machine kunnen denken dat de bewegingen van de koppen bestuurd worden door een tweede, autonome, machine met werkbanden maar zonder invoer- of uitvoerbanden. Een gelijksoortige definitie voor RAMs zou stipuleren, dat de rij uitgevoerde instructies en de rij bezochte geheugenplaatsen, in een berekening,

onafhankelijk zijn van de invoer. Het begrip onachtzame machine is om verschillende redenen interessant. Ten eerste, omdat net zoals een berekeningsmodel een zekere formalisering verschaft voor het begrip algoritme, het begrip onachtzame machine een zekere formalisering verschaft voor de onachtzame algoritme. Bijvoorbeeld, het zoeken in een tafel met de sequentieële zoek algoritme kan onachtzaam geprogrammeerd worden (als het zoekproces niet stopt, als het gezochte item gevonden is, maar doorgaat tot het eind van de tafel), maar een binair zoek-algoritme kan niet onachtzaam geprogrammeerd worden, daar het aantal items dat onderzocht word klein is in verhouding tot de grootte van de hele tafel, en aangezien het afhangt van het gezochte item welke items onderzocht worden. Ten tweede, voor het vinden van ondergrenzen aan de complexiteit van problemen is het makkelijker onachtzame machines te beschouwen, dan machines in het algemeen. Door middel van een efficiënte simulatie van niet-onachtzame machines door onachtzame zouden zo algemeen geldende ondergrenzen kunnen worden gevonden. Ten derde, omdat berekeningen van onachtzame machines efficiënt op een combinatoriek logisch netwerk geïmplementeerd kunnen worden. Een berekening van n stappen door een onachtzame machine kan geïmplementeerd worden op een netwerk met $O(n)$ elementen, bijv. op het bitgewijze berekeningsmodel van Sectie 2.4. Omgekeerd, zijn er snelle onachtzame machines voor veel netwerk constructies. FISCHER en PIPPENGER [1979] lieten zien dat, voor on-line berekeningen:

STELLING 8. *Als A door een Turing machine geaccepteerd wordt in tijd $T(n)$, dan wordt A door een onachtzame Turing machine (met twee werkbanden) geaccepteerd in tijd $O(T(n) \log T(n))$. Er zijn verzamelingen A die door een Turing machine in echte-tijd geaccepteerd kunnen worden, maar waarvoor iedere onachtzame Turing machine tenminste $n \log n$ tijd nodig heeft.*

Voor veel problemen blijkt dat niet-onachtzaam gedrag van een algoritme slechts spaarzaam nodig is: we kunnen dan verschillende aspecten van het probleem isoleren ten opzicht waarvan het algoritme zich onachtzaam kan gedragen. Dit wijst op een heel skala van gedeeltelijke onachtzaamheid tussen totaal onachtzaam en helemaal niet onachtzaam. VITÁNYI [1980c] toonde aan dat, onder verschillende formalisering van het begrip "gedeeltelijke onachtzaamheid", er problemen met een inherente graad van onachtzaamheid bestaan zodat het kleinste teveel aan onachtzaamheid van een accepterende Turing machine evenveel toename in berekeningstijd met zich meebrengt als het totaal onachtzaam zijn.

3. PARALLELE BEREKENINGSMODELLEN

In the future, very large quantities of computing machinery may be placed on a single chip. Such chips will be easily and quickly designed, and rapidly implemented. This capability will present both a great opportunity and a great challenge. How are we to organize and program such a wealth of hardware? Certainly not the way we do now.

C. Mead & L. Conway [1980].

Het bovenstaande citaat vertoont een verrassende gelijkenis met dat in de aanhef van dit artikel. Het vertoont ook het vertrouwen dat wij wegen zullen vinden om de nieuwe machines efficiënt en volledig te benutten.

Als gevolg van verbeterde fabricatie technieken wordt het mogelijk om steeds kleinere geïntegreerde elektronische netwerken te vervaardigen, en daardoor meer compacte, en meer complexe, functies op een enkele siliconen schilfer samen te brengen. Aan het eind van de tachtiger jaren zullen we in staat zijn schilfers te fabriceren met daarop 10^7 - 10^8 individuele transistors. Een enkele schilfer van dat soort zal meer berekeningskracht in zich bergen dan de grootste computer die nu bestaat. Deze aan VLSI (Very Large Scale Integration) inherente mogelijkheden roepen nieuwe problemen op.

De programmeringstechnieken die nu in gebruik zijn, evenals de gangbare complexiteitstheorie, hebben voornamelijk betrekking op de tijd dat een rekenmachine in essentie (of grotendeels) een sequentieel apparaat was. De nieuwe technieken maken het echter mogelijk om in hoge mate gelijktijdig rekenende machine assemblages te vervaardigen, bestaande uit duizenden verbonden componenten die ieder op zich even krachtig zijn als een huidige grote computer. In deze sectie zullen we bekijken hoe de overgang op een parallel berekeningsmodel de, in het voorgaande ontwikkelde, complexiteitstheorie beïnvloedt.

3.1. PARALLELE RAMS

In de recente literatuur over de complexiteitstheorie kan men tenminste drie hoofdklassen van parallelle machine modellen onderscheiden. De eerste klasse wordt vertegenwoordigd door de alternerende machines van CHANDRA, KOZEN en STOCKMEYER [1979] en door het k-PRAM model van SAVITCH en STIMSON [1979]. De modellen in deze klasse bestaan uit een boom met processoren op de knopen. Iedere processor is een min of meer realistisch

model van een seriële computer en heeft zijn eigen privé geheugen, dat wil zeggen, iedere processor is een RAM. Er is geen gemeenschappelijk geheugen waartoe iedere processor toegang heeft. De tweede klasse bestaat uit de MRAMS, zie Sectie 2.3, van HARTMANIS en SIMON [1974], en de vector machines van PRATT en STOCKMEYER [1976]. Het parallelle karakter van deze modellen is min of meer verborgen. Zij bestaan uit seriële RAMs met toevoeging van zeer krachtige instructies. In het bijzonder betreft dit de volgende twee mogelijkheden: het zeer snel genereren van zeer grote getallen en het manipuleren van de afzonderlijke bits van de in het geheugen opgeslagen getallen. Het vermogen om snel grote getallen te genereren is het gevolg van het hanteren van het uniforme kostenkriterium (zie Secties 2.1 en 2.3). Deze machines worden wel als parallelle modellen beschouwd, omdat de enige mogelijkheid om zo'n machine in de praktijk te realiseren bestaat in het implementeren van iedere operatie als een parallelle routine, uit te voeren door vele afzonderlijke processoren. De derde klasse wordt gerepresenteerd door de P-RAMS van FORTUNE en WYLLIE [1978]. Modellen in deze klasse bestaan uit een onbegrensde collectie processoren die allen verbonden zijn met een gemeenschappelijk geheugen. Iedere processor is weer een RAM. Een manier om tegen machines in deze derde klasse aan te kijken, is ze te beschouwen als die van de eerste klasse maar met de toevoeging van een onbegrensd gemeenschappelijk RAM geheugen. Een model dat in kracht vergelijkbaar is met de P-RAMS wordt gevormd door de L-PRAMS van SAVITCH [1978b, 1980]. Hierbij nemen we als basis het k-PRAM model, en breiden de instructieverzameling van de individuele processors uit met de vermenigvuldigingsoperatie en enige 'List processing' instructies, gelijkende op de CONS, CAR en CADR instructies van LISP. Dit model gebruikt het uniforme kostenkriterium, en daarom bestaat iedere individuele processor, zo ongeveer, uit een krachtige register machine van de tweede klasse.

We zullen niet ingaan op de details van bovenstaande modellen, maar geven hieronder het globale effect aan van het parallelisme op de geassocieerde complexiteitsklassen.

STELLING 9. (i) *Voor modellen in de eerste twee klassen, nl. alternerende RAMs, k-PRAMS, en MRAMS, zijn de deterministische en de niet deterministische versies die in polynomiale tijd werken even krachtig; gelijk in kracht aan gewone RAMs onder het logaritmische kostenkriterium die in polynomiale ruimte werken.*

(ii) De deterministische versies van de P-RAM en L-PRAM, die in polynomiale tijd werken, zijn in kracht gelijk aan de gewone RAMs, onder het logaritmische kostenkriterium, die in polynomiale ruimte werken.

(iii) De niet-deterministische versies van de P-RAM en de L-PRAM, die in polynomiale tijd werken, zijn in kracht gelijk aan de gewone niet-deterministische RAMs die in exponentiële (in een polynoom) tijd werken.

Om verwarring te voorkomen, merken we het volgende op. Alternierende RAMs zijn een uitbreiding van niet-deterministische RAMs, en daarom is het onderscheid tussen determinisme en niet-determinisme op alternierende RAMs niet van toepassing. Voor RAMs met het logaritmische kostenkriterium, die in polynomiale ruimte werken, zijn de deterministische en niet-deterministische versies gelijk in kracht volgens Stelling 2.

Als gevolg van bovenstaande stelling blijkt dat parallellisme een meer dan polynomiale toename in berekeningssnelheid kan geven, dan en slechts dan, als de polynomiale tijdcomplexiteitsklasse (voor deterministische RAMs) ongelijk is aan de polynomiale ruimtecomplexiteitsklasse (voor RAMs met logaritmische kosten). Het is niet bekend of niet-determinisme krachtiger is dan determinisme voor P-RAMs en L-PRAMs; het beslissen van deze kwestie zou ook uitmaken of polynomiale ruimtecomplexiteit (voor RAMs met logaritmische kosten) gelijk is aan exponentiële tijdcomplexiteit voor niet-deterministische RAMs.

Tenslotte zij de lezer erop gewezen dat de verbluffende snelheid die te bereiken schijnt met parallelle machines, bijvoorbeeld het oplossen van NP-complete problemen in deterministische polynomiale tijd volgens Stelling 9(i), ten koste gaat van het gebruik van een enorm aantal processoren. Soms wordt daarom het tijdprocessor produkt als een realistischer kostenmaat voor zulke machines beschouwd.

3.2. VLSI BEREKENINGSMODELLEN

De meer theoretische berekeningsmodellen van Sectie 3.1, ofschoon door de huidige ontwikkelingen van VLSI (Very Large Scale Integration) geïnspireerd, gaan vrijwel voorbij aan de technologische beperkingen van VLSI, speciaal met betrekking tot de communicatie tussen de verschillende processoren. Het getuigt van meer werkelijkheidszin om bij een parallel berekeningsmodel te denken aan een, op een siliconen schilfer te realiseren, verzameling modules die op praktisch te verwezenlijke wijze met elkaar verbonden zijn. Hierbij bestaat dan iedere module uit een (kleine) RAM.

Voor de geïnteresseerde lezer verwijzen we naar MEAD en CONWAY [1980] die "het skala van abstrakties bestrijken tussen de onderliggende natuurkundige principes en complete VLSI digitale computer systemen", naar THOMPSON [1979a] voor de complexiteitstheorie van een realistisch geformaliseerd VLSI berekeningsmodel, en naar de bijdragen van M. REM, J. VUILLEMIN en F. PETERS in dit colloquium.

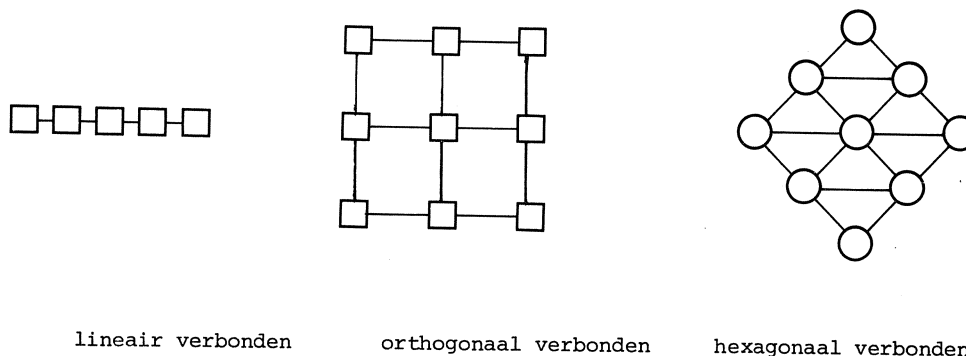
De architectuur van conventionele machines stuit gewoonlijk op twee moeilijkheden die wij proberen te vermijden bij het ontwerp van VLSI systemen. (1) De processor wordt van het bijbehorend geheugen gescheiden door lange communicatielijnen. Deze verbindingen zijn lang genoeg om de uitwisseling van gegevens tussen processor en geheugen aanzienlijk te vertragen. (2) De "van Neumann" machine bevat een enkele processor die sequentieel instructies ophaalt en uitvoert: zij biedt weinig gelegenheid voor gelijktijdige verwerkingsactiviteiten. In deze sectie bekijken we enige mogelijkheden om de verbindingskosten te verlagen, en om meerdere processoren gelijktijdig te gebruiken.

De conventionele computer bestaat vaak uit één processor, verbonden met een groot homogeen (random-access) geheugen. De processor haalt een instructie uit dit geheugen, decodeert die, voert haar uit, en herhaalt de cyclus. Veel van de instructies betreffen het nogmaals raadplegen van het geheugen. De prestaties van zo'n computer hangen af van de snelheid waarmee toegang tot het geheugen verkregen kan worden. Als nu een m -bits geheugen in matrixvorm op een schilfer geïmplementeerd is, vereist dit draden van ongeveer $\sqrt{m}$ lengte-eenheden om de gegevens tussen processor en geheugencel te transporteren. Lange draden kosten veel ruimte op een schilfer, en gebruiken aanzienlijke hoeveelheden energie om een signaal door te geven. Bij de huidige grote computers wordt de prestatie verder verminderd door de verschillende niveaus, die nodig zijn om een flink groot geheugen te krijgen: schilfer, plaat met voorbedrukte bedrading en stapelrek. De hoeveelheid bedrading op de eerste twee groeit als $\sqrt{m}$, maar de lengte van de bedrading voor de stapelrekken groeit lineair met de geheugengrootte. Bij zo'n organisatie wordt het grootste deel van de kosten, tijd en energie, die voor het rekenen nodig zijn, uitgegeven aan het transporteren van gegevens over lange afstanden. De situatie ligt beter bij het gebruik van een *geueghierarchie*, bestaande uit, vaak geraadpleegde, zeer snelle "registers" of "accumulators" (gewoonlijk voor tussenresultaten van aritmetische berekeningen), een "cache" geheugen (voor vaak geraadpleegde gegevens of instructies), en een algemeen snelgeheugen (bestaande uit enkele miljoenen bit) gecomplementeerd door een achtergrondgeheugen (van schijven). Onder redelijke aannamen

van de snelheid, en de frequentie, waarmee de verschillende geheugens in de hiërarchie geraadpleegd worden, zal zo'n hiërarchisch opgebouwd geheugen 10 tot 20 maal zo snel werken als een niet-hiërarchisch geheugen. De tijdwinst kan vervolgens verbeterd worden door de gelijktijdigheid (concurrency) in de computers toe te laten nemen. We noemen hier pipeline- en multiprocessor-structuren. Zie verder ook STONE [1975]. *Pipelined processoren* werken volgens het lopende band principe: er zijn verschillende processoren, waarvan ieder een gedeelte van het werk doet en het resultaat overhandigt aan de volgende processor in de productielijn. Als in zo'n opzet twee processoren gelijktijdig functioneren, waarbij ieder het gebruik krijgt over de helft van het totaal geheugen van m bits, werkt dit arrangement meer dan verdubbelend op de berekeningssnelheid. De tijd om een instructie uit te voeren wordt, met hier en daar enige verwaarlozingen, $\frac{1}{2} \sqrt{m/2}$, ongeveer een-derde van die in het geval van een enkele processor. Net als bij een lopende band kunnen meerdere berekeningen gelijktijdig in de pipeline in bewerking zijn. Multiprocessoren bieden een schakelstructuur die iedere processor in staat stelt met iedere andere processor te communiceren. Men hoopt dan dat algoritmen, die in verband met de vereiste gegevenscommunicatie minder geschikt zijn voor pipelines, wel goed op multiprocessoren uitgevoerd kunnen worden. Al deze voorbeelden van processor-geheugen structuren wijzen erop dat we, om de mogelijkheden van VLSI te benutten, moeten streven naar het gebruik van zoveel mogelijk processoren, die gelijktijdig bezig gehouden kunnen worden, en die zo dicht mogelijk bij het geheugen dat zij nodig hebben geplaatst zijn. In het volgende houden we ons bezig met structuren, die opgebouwd kunnen worden uit goedkope hardware, waarop parallele algoritmen met hoge prestaties direct geïmplementeerd kunnen worden. Met prestatie bedoelen we hier niet zozeer de som van de elementaire operaties, zoals in de conventionele complexiteitstheorie, maar eerder de doorvoer (throughput) die we kunnen verkrijgen als zo'n gespecialiseerd randapparaat aan een gastheer-computer geschakeld wordt. Dit betreft dan de tijd voor invoer/uitvoer, besturing, gegevensbewegingen en aritmetiek. Voor VLSI leiden simpele en regelmatige verbindingen tot goedkope implementaties en dichte pakkingen, en dit laatste betekent hoge prestaties. Daarom zijn we voornamelijk geïnteresseerd in algoritmen met simpele en regelmatige gegevensstromen. Verder kan pipelining gebruikt worden om de gegevensverwerking gelijktijdig met in- en uitvoer te doen verlopen. Omdat in een parallel algoritme meer dan één taak gelijktijdig uitgevoerd kan worden, is gelijktijdigheidsbesturing (concurrency control) noodzakelijk om de gewenste interactie tussen verschillende taken

af te dwingen, zodat de uitvoering van het algoritme als geheel correct verloopt. In het volgende bekijken wij alleen synchrone parallelle algoritmen en machine-architecturen waarbij modulen in de pas gehouden worden door een centrale klok, of door andere meer gedistribueerde middelen, zodanig dat de juiste gegevens de juiste plaats op de juiste tijd, voor de gewenste interactie, bereiken.

(1) In bijvoorbeeld KUNG [1979a,b] of MEAD en CONWAY [1980] worden *net-verbonden* (mesh-connected) structuren bekeken, gebaseerd op de architectuur van de ILLIAC IV, zie BARNES et.al. [1968]. Enige voorbeelden van net-verbonden structuren worden gegeven in Figuur 5.



Figuur 5. Net-verbonden structuren

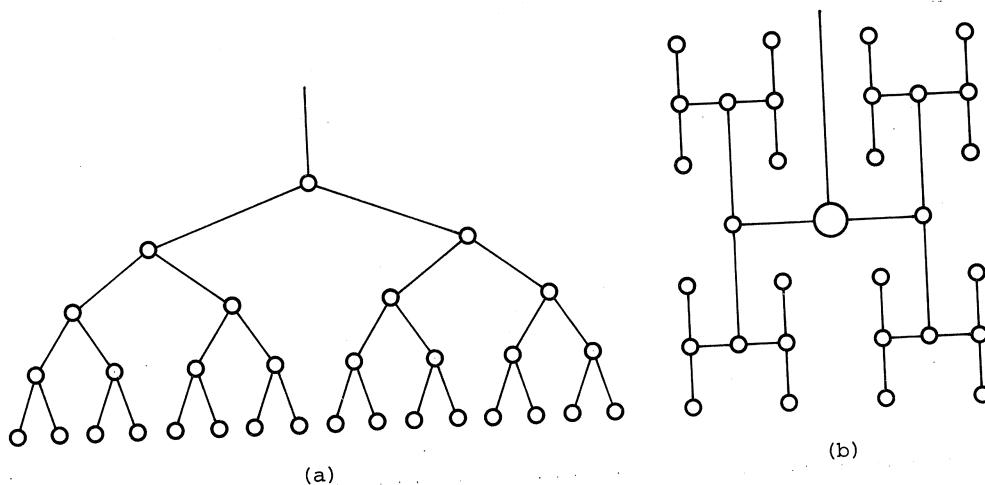
Iedere knoop stelt een processor met wat geheugen voor. De vereisten aan (VLSI) hardware voor zulke processoren zijn bescheiden. De informatie-verwerkende modulen op de knopen zijn uniform; de verbindingen zijn simpel, regelmatig en kort; en de uitwendige verbindingen naar de gastheer-computer (voor in- en uitvoer) liggen bij de randmodulen en zijn geminimaliseerd. De structuur is geometrisch recurrent. Zonder in te gaan op de details, blijkt dat onder meer de volgende prestaties (met behulp van bijvoorbeeld pipelining) bereikt kunnen worden.

Lineair net: - matrix-vector product in n dimensies in $3n$ tijdeenheden.
 - n insert-, delete- en extract-operaties in een priority queue van grootte n in $O(n)$ tijdeenheden.
 - sorteren van n items in $O(n)$ tijdeenheden.

Orthogonaal net: - oplossen van $Ax = b$ (in n dimensies) in $3n$ tijdeenheden.

Hexagonaal net: - 2 $n \times n$ matrices vermenigvuldigen in $4n$ tijdeenheden.

(2) *Boomverbonden* (tree-connected) structuren zoals voorgesteld door MEAD en REM [1979], zie Figuur 6a. In tegenstelling tot de net-verbonden structuren, zijn de verbindingen in een boomstructuur, ingebed in het platte vlak, niet uniform. De afstand tussen twee verbonden processoren neemt toe naarmate zij dichterbij de wortel zijn gelegen. Bij implementatie op schilfers kan de tijd voor signaaloverdracht tussen aanliggende modules, niettemin, grotendeels onafhankelijk gemaakt worden van de relatieve positie in de boom, door grotere aandrijvers aan langere draden te koppelen.^{*)} In MEAD en REM [1979] (zie ook MEAD en CONWAY [1980]) wordt aangetoond dat dit, ondanks het feit dat grotere aandrijvers een grotere oppervlakte in beslag nemen, mogelijk is met de opmaak van Figuur 6b, waarbij een boom op een schilfer wordt geïmplementeerd zodat de gebruikte oppervlakte lineair is in het aantal processoren.



Figuur 6. Boom-verbonden structuur

Het blijkt, dat de tijd, om een gegevens-item over een verbinding te sturen in een boom van zulk een architectuur, min of meer onafhankelijk is van de diepte van de verbinding in de boom, en dat we de wortel van de boom kunnen gebruiken voor in- en uitvoer communicatie met de buitenwereld. Deze boomstructuur is ideaal voor zoekprocessen, zie BENTLEY en KUNG [1979]. Algoritmen die eerst een groot aantal kandidaten voor een oplossing van een probleem

^{*)} Zie in dit verband de bijdrage van M.REM.

genereren en vervolgens hieruit de goede oplossing selecteren, kunnen door de boomstructuur efficiënt verwezenlijkt worden. In CONWAY en MEAD [1980] worden algoritmen gegeven zodat:

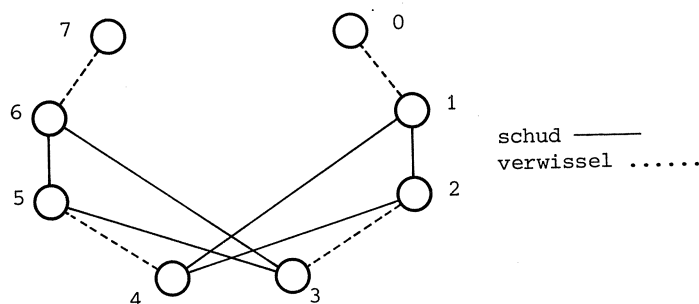
PROBLEEM	BENODIGDE TIJD	BENODIGDE PROCESSOREN
Sorteren n items	n	n
$n \times n$ matrix vermenigvuldiging	n^2	$2n^2 - 1$
Vinden van clique in graaf op n knopen	n^2	$2^n - 1$

De processor op het hoogste niveau, die vanuit de schilfer met de buitenwereld communiceert, heeft de grootste aandrijver en is daarom in staat deze communicatie te verzorgen zonder noemenswaardige vermindering in prestatie. Daarom kan zulk een machine tot een groot aantal individuele schilfers uitgebreid worden, en toch ongeveer de snelheid behouden van de individuele processoren. Het vinden van een clique in een graaf is NP-compleet. We merken op dat de boomstructuur, voor het oplossen van NP-complete problemen in polynomiële tijd (door het weerspiegelen, van een niet-deterministische RAM berekening, in de boomstructuur), een exponentieel aantal processoren vergt. Door dat we verschillende boomstructuren van bescheiden omvang, zonder verlies van prestatie, tot een grote boom aaneen kunnen schakelen, zou deze brute-kracht methode met de ontwikkeling van VLSI zeer wel aan belang kunnen winnen.

(3) *Schud-verwissel* (shuffle-exchange) structuren. Stel dat we een netwerk hebben, bestaande uit $n = 2^m$ knopen (processor-geheugen modulen). De knopen zijn genummerd als $0, 1, 2, \dots, 2^m - 1$. Zij $i_{m-1} i_{m-2} \dots i_1$ de binaire representatie van het getal i , $0 \leq i \leq 2^m - 1$. De schudfunctie wordt gedefinieerd als

$S(i_{m-1} i_{m-2} \dots i_1) = i_{m-1} i_{m-2} \dots i_1 i_m$, en de verwisselfunctie als $E(i_{m-1} i_{m-2} \dots i_1) = i_{m-1} i_{m-2} \dots \bar{i}_1$, waarbij $\bar{i}_1 = 0$ als $i_1 = 1$ en $\bar{i}_1 = 1$ als $i_1 = 0$.

Het netwerk is een schud-verwissel netwerk als knoop i met knoop $S(i)$ verbonden is voor alle i , en met knoop $E(i)$ voor even i . Figuur 7 laat een schud-verwissel netwerk zien van grootte $n = 8$.



Figuur 7. Schud-verwissel structuur

Het blijkt dat, door i maal om de beurt verwissel- en schudverbindingen te gebruiken, gegevens die origineel beschikbaar waren in twee knopen, waarvan de namen 2^i verschillen, in één knoop bij elkaar gebracht kunnen worden, voor alle $i = 0, 1, \dots, m-1$. Deze communicatie structuur wordt door veel algoritmen gedeeld. In BATCHER [1968] wordt getoond dat n elementen op zo'n netwerk (bitonisch) gesorteerd kunnen worden in $O(\log^2 n)$ stappen, als de modules vergelijkings-verwisselings operaties kunnen uitvoeren. In PEASE [1968] blijkt dat de n -punts snelle Fourier transformatie (FFT) op dit netwerk in $O(\log n)$ stappen kan worden uitgevoerd, als de modules kunnen optellen en vermenigvuldigen. Andere toepassingen zoals matrix transpositie en het evalueren van lineaire recurrente betrekkingen worden gegeven in STONE [1971, 1975], waar ook heldere uiteenzettingen en achtergronden voor deze algoritmen worden gegeven. SCHWARTZ [1979] bouwde deze aanpak uit, en liet zien dat zulke netwerken van n processoren zo $\sqrt{n}$ items kunnen sorteren in $O(\log n)$ stappen; $\sqrt[3]{n} \times \sqrt[3]{n}$ matrices kunnen vermenigvuldigen in $O(\log n)$ stappen, enz. Deze laatste grenzen zijn optimaal. Dit kan als volgt worden beargumenteerd. Als voor een probleem met n invoervariabelen tenminste één uitvoervariabele van alle invoervariabelen afhangt, is het een gevolg van de fysische fan-in beperkingen dat we per cyclus slechts een constant aantal, zeg k , eenheden op één plaats bij elkaar kunnen brengen. Daarom zal zo'n berekening tenminste $\log_k n$ cycli vergen.

Het schud-verwissel netwerk is erg onregelmatig en weinig modulair opgebouwd. Misschien is dit een onvermijdelijk gegeven door het grote permutatievermogen. THOMPSON [1979a] toont aan dat het netwerk niet planair is, en niet op een schilfer ingebed kan worden op een oppervlak dat lineair is in het aantal processoren. Om de draden voor een schud-verwissel netwerk

uit te leggen op een schilfer zou een oppervlakte nodig zijn van $\Omega(n^2/\log^2 n)$. MEERTENS [1979] toont aan, dat geen fysisch realiseerbaar netwerk van processoren dat voldoet aan modulariteit en regulariteits criteria (d.w.z. dat een groter netwerk gemaakt kan worden door het simpelweg aaneenkoppelen van kleinere netwerken) de ondergrens van $\log n$ kan realiseren.

De nadelen, verbonden aan het schud-verwissel netwerk, worden niet gedeeld door de, door PREPARATA en VUILLEMIN [1979] voorgestelde, *kubus-verbonden-cycli* (cube-connected-cycles of CCC) structuren, die tot dezelfde prestaties komen als de schud-verwissel structuren. Deze structuren voldoen echter wel aan de voorwaarden voor VLSI realisatie: modulariteit, gemak van opmaak (lay out), eenvoud van communicatie, en eenvoud van timing en controle over het hele systeem, zie MEAD en CONWAY [1980]. De auteurs stellen een draad opmaak voor, die fysiek gerealiseerd kan worden door twee orthogonale lagen, en die voor verschillende problemen (zoals FFT) in het VLSI model van THOMPSON [1979a] optimaal is. In dit model heeft iedere draad een dikte van één eenheid, op de siliconen schilfer, en kan één eenheid informatie per tijd-eenheid tussen de modules transporteren. THOMPSON [1979b] bewees bijvoorbeeld dat, binnen zijn VLSI berekeningsmodel, voor de FFT op n punten geldt dat $AT^2 \geq n^2/16$, waarbij A de oppervlakte van de schilfer is, en T de rekentijd. Dergelijke afruil relaties tussen schilfer oppervlakte en rekentijd uitgedrukt in de grootte van het probleem, werden bijvoorbeeld door BRENT en KUNG [1979a] verkregen voor het vermenigvuldigen van binaire getallen ($AT^{2\alpha} \geq cn^{1+\alpha}$, waarbij n de lengte is van de betrokken binaire getallen, c een constante afhankelijk van de technologische gegevens, en α een willekeurig getal tussen 0 en 1.) In BRENT en KUNG [1979b] wordt een bovengrens gegeven aan A en T (namelijk $A \in O(n \log n)$ en $T \in O(\log n)$) voor het probleem van het optellen van twee n -bits getallen. Uit dit resultaat volgt dat binaire vermenigvuldiging "moeilijker" is dan binaire optelling voor de complexiteitsmaat $AT^{2\alpha}$ (voor iedere $\alpha \geq 0$). In MEAD en REM [1979] wordt het verband tussen oppervlak een schilfer, realiseerbare cyclusduur, en mogelijke positionering van modules en bedrading diepgaand geanalyseerd. Hun resultaten suggereren richtlijnen voor de organisatie van grote geïntegreerde systemen, waarbij de bedrading vereist voor de informatie overdracht over de schilfer het meeste oppervlak in beslag neemt en het aandrijven van de signaaloverdracht door de draden verantwoordelijk is voor het merendeel van het tijdverbruik.

De literatuur over de complexiteit van VLSI berekeningen neemt momenteel razendsnel toe. In tegenstelling tot de situatie bij sequentiële berekeningsmodellen lijkt het of hier ondergrenzen aan de complexiteit van problemen relaties makkelijk te geven zijn, alsmede VLSI constructies die deze ondergrenzen realiseren.

LIPTON en SEDGEWICK [1981] tonen aan dat de gebruikte technieken om zulke ondergrenzen te bewijzen, in wezen alle variaties op één thema, uiteindelijk neerkomen op het "crossing-sequence" argument voor Turing machines, geïntroduceerd door RABIN [1963] en HENNIE [1966], die toen leidden tot enkele van die schaars verkregen ondergrenzen aan de complexiteit van problemen op Turing machines.

4. REFERENTIES

- [1] AANDERAA, S.O. [1974], *On k-tape versus (k-1)-tape real-time computation*, SIAM-AMS Proceedings, Vol.7 (Complexity of Computation) 75-96.
- [2] AHO, A., J.E. HOPCROFT & J.D. ULLMAN [1974], *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass.
- [3] BARNES, G.H., R. BROWN, M. KATO, D.J. KUCK, D.L. SLOTNICK & R.A. STOKES [1968], *The ILLIAC IV computer*, IEEE Transactions on Computers C-17, 746-757.
- [4] BATCHER, K.E. [1968], *Sorting networks and their applications*, Proceedings Spring Joint Computer Conference 32, 307-314.
- [5] BENTLEY, J.L. & H.T. KUNG [1979], *Two papers on a tree-structured parallel computer*, Tech. Rept., Dept. Comp. Sci., Carnegie-Mellon Univ., Pittsburgh, Pennsylvania.
- [6] BRENT, R.P. & H.T. KUNG [1979a], *The area-time complexity of binary multiplication*, Techn. Rept., Dept. Comp. Sci., Carnegie-Mellon Univ., Pittsburgh, Pennsylvania.
- [7] BRENT, R.P., & H.T. KUNG [1979b], *A regular layout for parallel adders*, Techn. Rept., Dept. Comp. Sci., Carnegie-Mellon Univ., Pittsburgh, Pennsylvania.

- [8] CHANDRA, A.K., D.C. KOZEN & L.J. STOCKMEYER [1979], *Alternation*, I.B.M., T.J. Watson Research Centre Techn. Rept. RC 7489(#32286).
- [9] COBHAM, A., [1965], *The intrinsic computational difficulty of functions*, Logic, Methodology and Philosophy (Proc. 1964 Int. Congr.), North-Holland, Amsterdam, 24-30.
- [10] COOK, S.A., [1971], *The complexity of theorem proving procedures*, Proc. 3rd Annual ACM Symposium on Theory of Computing, 151-158.
- [11] COOK, S.A. & R.A. RECKHOW [1973], *Time-bounded random access machines*, J. Computer and System Sciences 7, 354-375.
- [12] FISCHER, P.C., A.R. MEYER & A.L. ROSENBERG [1972], *Real-time simulation of multihead tape units*, J. ACM, 19, 590-607.
- [13] FORTUNE, S. & J. WYLLIE [1978], *Parallelism in random access machines*, Proc. 10th ACM Symposium on Theory of Computing, 114-118.
- [14] HARTMANIS, J. [1971], *Computational complexity of random access stored program machines*, Mathematical Systems Theory 5, 232-245.
- [15] HARTMANIS, J. & J. SIMON [1974], *On the power of multiplication in random-access machines*, Proc. 15th Annual IEEE Symp. on Switching and Automata Theory, 13-23.
- [16] HENNIE, F.C. [1966], *On-line Turing machine computations*, IEEE Transactions on Electronic Computers, EC-15, 35-44.
- [17] KARP, R.M. [1972], *Reducibility among combinatorial problems*, In: R.E. Miller, J.W. Thatcher (eds.) *Complexity of Computer Computations*, Plenum Press, New York, 85-103.
- [18] KARP, R.M. [1975], *On the computational complexity of combinatorial problems*, Networks 5, 45-68.
- [19] KUNG, H.T., [1979a], *Let's design algorithms for VLSI systems*, Techn. Rept., Dept. Comp. Sci., Carnegie-Mellon Univ., Pittsburgh, Pennsylvania.
- [20] KUNG, H.T. [1979b], *The structure of parallel algorithms*, Techn. Rept., Dept. Comp. Sci., Carnegie-Mellon Univ., Pittsburgh, Pennsylvania.

- [21] MEAD, C.A. & M. REM, [1979], *Cost and performance of VLSI computing structures*, IEEE Journal of Solid State Circuits SC-14, 455-462.
- [22] LIPTON, R.J. & R. SEDGEWICK, [1981], *Lower bounds for VLSI*, Proc. 13th ACM Symposium on Theory of Computing, 300-307.
- [23] MEAD, C.A. & L.A. CONWAY, [1980], *Introduction to VLSI systems*, Addison-Wesley, Reading, Massachusetts.
- [24] MEERTENS, L.G.T.M., [1979], *Recurrent ultra computers are not log n-fast*, Techn. Rept. IW 118, Mathematisch Centrum, Amsterdam.
- [25] MINSKY, M.L., [1961], *Recursive unsolvability of Post's problem of "Tag" and other topics in the theory of Turing machines*, Annals of Mathematics, 74, 437-455.
- [26] NEUMANN, J. von, [1949], *The future of high-speed computing*, Collected Works, Vol. V, Pergamon Press (1963), 236.
- [27] PEASE, M.C. [1968], *An adaptation of the Fast Fourier Transform for parallel processing*, J. ACM 15, 252-264.
- [28] PIPPENGER, N. & M.J. FISCHER, [1979], *Relations among complexity measures*, J. ACM 26, 361-381.
- [29] PRATT, V.R. & L.J. STOCKMEYER, [1976], *A characterization of the power of vector machines*, J. Computer and System Sciences, 12, 198-221.
- [30] PREPARATA, F.P. & J. VUILLEMIN, [1979], *The Cube-Connected Cycles: a versatile network for parallel computations*, Proceedings 20th Annual IEEE Symposium on Fundamentals of Computer Science, 140-147.
- [31] RABIN, M.O., [1963], *Real-time computation*, Israel Journal of Mathematics, 1, 203-211.
- [32] SAVAGE, J.E., [1976], *The Complexity of Computing*, Wiley & Sons, New York.
- [33] SAVITCH, W.J. [1970], *Relationship between nondeterministic and deterministic tape complexities*, J. Computer and System Sciences, 4, 177-192.

- [34] SAVITCH, W.J. [1978a], *The influence of the machine model on computational complexity*, In: *Interfaces Between Computer Science and Operations Research*, MC Tract 99, Mathematisch Centrum, Amsterdam, 1-32.
- [35] SAVITCH, W.J. [1978b], *Parallel and nondeterministic time complexity classes*, Lecture Notes in Computer Science 62 (ICALP 5), Springer Verlag, Berlin, 411-424.
- [36] SAVITCH, W.J. [1980], *Parallel random access machines with powerful instruction sets*, Manuscript, Dept. APIS, Univ. Cal. San Diego, California.
- [37] SAVITCH, W.J. & M.J. STIMSON [1979], *Time bounded random access machines with parallel processing*, J. ACM, 26, 103-118.
- [38] SCHWARTZ, J.T. [1979], *Ultracomputers*, Manuscript, Comp. Sci. Dept., New York Univ.
- [39] SLISENKO, A.O. [1973], *Recognition of palindromes by multihead Turing machines*, In: *Problems in the Constructive Trend in Mathematics VI*, 30-202 (Proc. Steklov Inst. Math. 129; V.P. Orverkov & N.A. Sonin eds.), Academy of Sciences USSR. English translation by R.A. Silverman, Am. Math. Soc., Providence, R.I., 1976.
- [40] STONE, H.S. [1971], *Parallel processing with the perfect shuffle*, IEEE Transactions on Computers C-20, 153-161.
- [41] STONE, H.S. [1975], *Parallel Computation*, In: H.S. Stone ed., *Introduction to Computer Architecture*, Science Research Associates, Inc., Chicago, 318-374.
- [42] THOMPSON, C.D. [1979a], *A Complexity Theory for VLSI*, Ph.D. Thesis, Dept. Comp. Sci., Carnegie-Mellon Univ., Pittsburgh, Pennsylvania.
- [43] THOMPSON, C.D. [1979b], *Area-time complexity for VLSI*, Proc. 11th Annual ACM Symposium on Theory of Computing, 81-88.
- [44] TURING, A.M. [1936], *On computable numbers with an application to the Entscheidungsproblem*, Proc. London Math. Soc. (2) 46, 230-265; Correction, *ibid.* (2) 43 (1937), 544-546.

- [45] VITÁNYI, P.M.B. [1979], *Multihead and multitape real-time Turing machines*, Techn. Rept. IW 111, Mathematisch Centrum, Amsterdam.
- [46] VITÁNYI, P.M.B. [1980a], *On the power of real-time Turing machines under varying specifications*, Lecture Notes in Computer Science 85 (ICALP 7), Springer Verlag, Berlin, 645-658.
- [47] VITÁNYI, P.M.B. [1980b], *Relativized obliviousness*, Lecture Notes in Computer Science 88 (MFCS '80), Springer Verlag, Berlin, 665-672.
- [48] VITÁNYI, P.M.B. [1980c], *Two-tape real-time computation*, Techn. Rept. IW 139, Mathematisch Centrum, Amsterdam. (To appear in J. Comp. Syst. Sci.)
- [49] YAMADA, H. [1962], *Real-time computation and recursive functions not real-time computable*, IRE-Trans. EC-11, 753-760.

B. DATASTRUCTUREN

BALANCED TREES AS A DATASTRUCTURE FOR REPRESENTING SORTED LISTS

K. MEHLHORN

1. INTRODUCTION

Sets and operations on sets (insert, delete, test for membership (= search), ...) are one of the fundamental topics of algorithm design and computer science in general. Many algorithms manipulate sets. Think of data bank systems and of combinatorial algorithms.

Trees are one of the most versatile methods for representing sets of data. The first usage of trees for the representation of sets was in the 50's. In the early 60's it was shown by Hibbard, that binary trees give logarithmic search time on the average. Unfortunately, the "naive" use of tree structures can lead to degeneracy, i.e. to trees whose depth is nearly equal to their size. This observation led to the development of balanced tree schemes in the 60's. In balanced trees the depth is guaranteed to be logarithmic in the size of the tree and hence search time is logarithmic in the worst case. However, insertions and deletions are more complicated in balanced trees; they require rebalancing after an insertion or deletion. Fortunately, rebalancing is restricted to the path of search and thus rebalancing cost is dominated by the cost of the preceeding search. The basics of balanced trees are reviewed in Section 2.

In Sections 3, 4 and 5 some nonstandard applications of trees are described: concurrent tree manipulation, sorting presorted files and finger searches. In these applications the rebalancing behavior of trees is a conceivable bottleneck either with respect to degree of parallelism attainable or with respect to efficiency. These applications call for a refined analysis of the rebalancing aspect of balanced trees.

Such an analysis is carried out in Section 4. In the first theorem of that section it is shown that the total number of rebalancing operations is linear in the number of insertions/deletions when the initial tree is empty. This analysis leads to a new adaptive sorting algorithm (described in Section

3) whose running time (never worse than $n \log n$) depends on "the degree of sortedness" of the input sequence. Later in the section, the theorem is refined in two ways. It is first shown that the number of rebalancing operations at height h decreases exponentially with h , i.e. most rebalancing operations occur near the leaves. This result can be used to prove that balanced trees behave well in a parallel environment. In a second extension, the case of an arbitrary initial tree is treated.

The latter extension is used in Section 5 to derive a new data structure: level-linked (a,b)-trees. This data structure supports finger searches, which may have sub-logarithmic running times. In Section 6 some applications of this data structure are described and optimal algorithms for many set operations (union, intersection, difference, ...) are given.

This paper summarizes the results of Huddleston of Mehlhorn: A new data structure for representing sorted lists. It owes a lot to previous work by Guibas, McCreight, Plass, Roberts and Brown, Tarjan.

2. SEARCH TREES

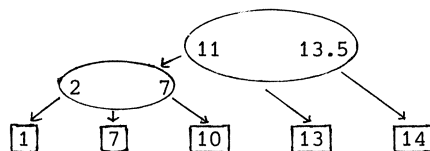
For us, a tree either consists of a single leaf (= a node of arity 0) or of a node v of arity $\rho(v)$ and $\rho(v)$ trees $T_1, \dots, T_{\rho(v)}$. T_i is called the i -th subtree at v . In our drawings, leaves are drawn as boxes and (interior = non-leaf) nodes are drawn as circles.

Let $(U, \leq)$ be a linearly ordered set, the universe. In our examples U is always the reals with the ordinary less than or equal relation. Let $S \subseteq U$, $|S| = n$, be a finite subset of U .

A *search tree* for S consists of

- 1) a tree T with n leaves;
- 2) a bijection cont between the leaves of T and the elements of set S ;
 $\text{cont}(w)$ is the content of leaf w .
- 3) for each node v of T a sequence $k_1(v), \dots, k_{\rho(v)-1}(v)$ of $\rho(v) - 1$ elements of U such that $\forall i (1 \leq i \leq \rho(v) - 1): k_{i-1}(v) < k_i(v)$ and for all leaves w in the i -th subtree at v $k_{i-1}(v) < \text{cont}(w) \leq k_i(v)$

EXAMPLE 1. $S = \{1, 7, 10, 13, 14\}$



Searching for an element X in a search tree T (with root r) for set S is a simple task. We search down the tree starting at the root until we reach a leaf. In each node v , we use the sequence $k_1(v), \dots, k_{\rho(v)-1}(v)$ in order to guide the search to the proper subtree.

```

proc search (X: element of U; T: search tree)
begin v ← root of T;
  while v is not a leaf
  do find i,  $1 \leq i \leq \rho(v) - 1$ ,
    such that  $k_{i-1}(v) < X \leq k_i(v)$ 
    v ← i-th son of v
  od
  if X = cont(v)
  then co success else co failure
end

```

In order to analyse the search algorithm we need some more notation.

DEFINITION. Let T be a tree and v a node of T . The depth $d(v)$ of node v is the length of the path from the root to v , the height $h(v)$ of v is the length of the longest path from v to a leaf in the subtree rooted at v and the height $h(T)$ of tree T is the height of the root.

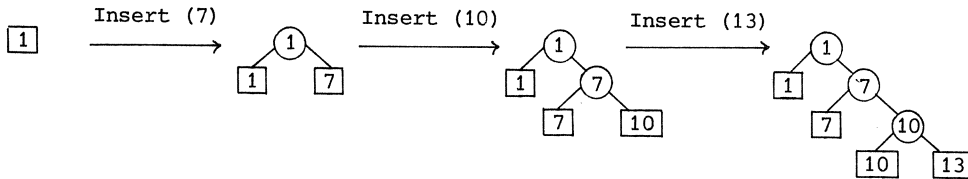
LEMMA 1. The cost of a search in tree T is

$$O(h(T) \cdot \text{maxarity}(T))$$

where $\text{maxarity}(T) = \max\{\rho(v); v \text{ node of } T\}$.

PROOF. The body of the while-loop is executed at most $h(T)$ times. The cost of finding i is certainly bounded by the arity of node v . $\square$

Insertions into and deletions from a tree T are also very simple. In order to insert X we first search for X . The search will end in some leaf w . We then replace this leaf by a subtree with two leaves with appropriate contents.

EXAMPLE 2.

Unfortunately, the naive approach to insertion can lead to degenerate trees, i.e. to trees whose height is linear in the number of leaves. In order to avoid degeneracy, one turns to balanced trees. In balanced trees the height is always logarithmic in the numbers of leaves and hence searches are efficient (cf. Lemma 1). Balanced trees come in many different kinds: AVL-trees [Adelson-Velskii-Landis], 2-3 trees [Aho-Hopcroft-Ullman], B-trees [Bayer-McGreight], $BB[\alpha]$ -trees [Nievergelt-Reingold], to name only a few. In this paper we will treat (a,b) -trees.

DEFINITION. Let a and b be integers with $a \geq 2$ and $2a-1 \leq b$. A tree T is an (a,b) -tree if

- a) all leaves of T have the same depth
- b) all nodes v of T satisfy $\rho(v) \leq b$
- c) all nodes v except the root satisfy $\rho(v) \geq a$
- d) the root r of T satisfies $\rho(r) \geq 2$.

For $b = 2a-1$ the class of (a,b) -trees is identical to the class of B-trees of order a . For $b \geq 2a$ we will refer to the class of (a,b) -trees as a class of *weak B-trees*. In our examples we will always use $(2,4)$ -trees.

LEMMA 2. Let T be an (a,b) -tree of height h . Let n be the number of leaves of T . Then

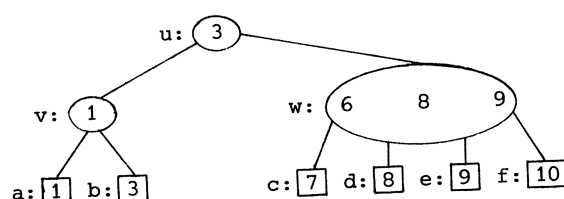
$$2 \cdot a^{h-1} \leq n \leq b^h.$$

PROOF. Obvious. $\square$

We infer from Lemma 1 that the depth of (a,b) -trees is logarithmic in the number of leaves. There are several ways of representing an (a,b) -tree in storage. A commonly used method is to represent each node by a record with

2b fields, b fields for elements of U and b fields for pointers. For node v of arity $\rho(v)$ only the first $2\rho(v)$ fields are actually used, in order to store the sequence $k_1(v), \dots, k_{\rho(v)-1}(v)$ and $k_{\rho(v)}(v) := \infty$ [this infinite key simplifies some programs] and $\rho(v)$ pointers to the sons of v. The leaves are stored separately.

EXAMPLE 3. $a = 2, b = 4,$



	pointer	key	pointer	key	pointer	key	pointer	key
u:	v	3	w	∞				
w:	c	6	d	8	e	g	f	∞
v:	a	1	b	∞				

LEMMA 3. Let T be an (a,b)-tree with n leaves. Then a search takes time $O(\log n)$.

PROOF. By Lemmas 1 and 2 and the observation that $\max \text{arity}(T)$ is bounded by constant b. $\square$

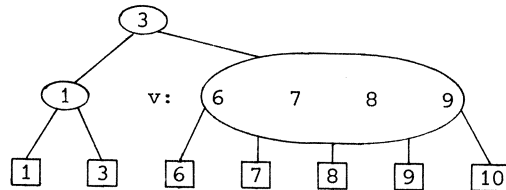
We note in passing, that every comparison-based search method must use logarithmic time. A careful treatment of this question can be found in Yao. Hence (a,b)-trees give optimal search times (up to a constant).

Next we treat the update operations insertion and deletion.

INSERTION. Operation Insert(X) will first search for element X; the search will end in some leaf w. Let v be the father of w.

(1) Expand v by giving it an additional leaf to the right of w ($\hat{=}$ split w), store $\text{cont}(w)$ and X in w and the new leaf in appropriate order and store $\min(\text{cont}(w), X)$ in node v at the appropriate position.

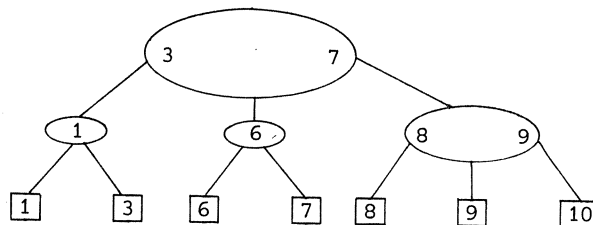
EXAMPLE 4. Insertion of 6 into the tree of the previous example yields



Expanding v increases the arity of v by 1. If it still $\leq b$ then the insertion is completed. Otherwise v needs to be split, splitting may propagate.

```
(2)  while  $\rho(v) = b+1$ 
      do if  $v$  is the root of  $T$ 
        then let  $x$  be a new node and make  $v$  the only son of  $x$ 
        else let  $x$  be the father of  $v$ 
        fi;
      let  $v'$  be a new node;
      expand  $x$ , i.e. make  $v'$  an additional sone of  $x$  immediately to the
      right of  $v$ , also move the  $\lfloor (b+1)/2 \rfloor$ -th key from  $v$  to  $x$ ;
      split  $v$ , i.e. take the rightmost  $\lceil (b+1)/2 \rceil$  sons (and the appropriate
      keys) away from  $v$  and make them sons of  $v'$ .
       $v \leftarrow x$ 
    od
```

EXAMPLE 5. (continuation of 4). Splitting v yields

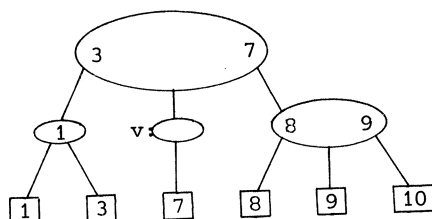


LEMMA 4. Let T be an (a,b) -tree with n leaves. Then $\text{Insert}(X)$ takes time $O(\log n)$.

PROOF. The search for X takes time $O(\log n)$ by Lemma 3. Also we store the path of search in a pushdown store during the search. After adding a new leaf to hold X we walk back (using the push-down-store) to the root and restore balance by some number of splitting operations. Since a single split takes constant time and splits are restricted to (a final segment of) the path of search, the lemma follows. $\square$

DELETION. Operation $\text{Deletion}(X)$ will first search for X ; the search ends in a leaf w with $\text{cont}(w) = X$. Let v be the father of w , and let w be the i -th son of v for some i , $1 \leq i \leq \rho(v)$. We *shrink* v by deleting w and one of the keys $k_{i-1}(v)$ or $k_i(v)$ (to be specific, we could delete $k_i(v)$ for $i < \rho(v)$ and $k_{i-1}(v)$ for $i = \rho(v)$).

EXAMPLE. Deleting 6 from the leaf of Example 5 yields.



Shrinking v decreases $\rho(v)$ by 1. If $\rho(v)$ is still $\geq a$ then rebalancing is completed. Otherwise v needs to be rebalanced by either fusing or sharing. Let y be any brother of v .

```

while ( $\rho(v) = a-1$  and  $\rho(y) = a$ )
do let  $x$  be the father of  $v$ ;
  fuse  $v$  and  $y$ , i.e. make all sons of  $y$  sons of  $v$  and delete  $y$ ; and
  move one key from  $x$  down to  $v$ .
co this will simultaneously shrink  $x$ , i.e. decrease its arity by
one;
 $v \leftarrow x$ ;
let  $y$  be a brother of  $x$ ;
if  $x$  does not have a brother
then begin co  $x$  is the root of  $T$ ;
  if  $\rho(x) = 1$ 

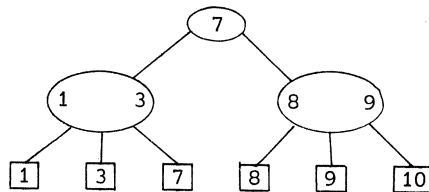
```

```

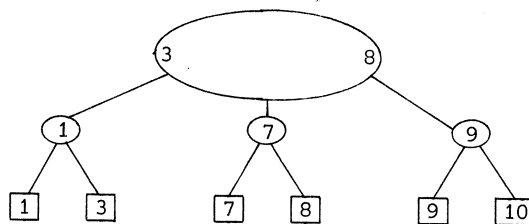
      then delete x
    fi;
    goto completed;
  end
fi
od;
co at this point we either have  $\rho(v) \geq a$  and rebalancing is completed
or  $\rho(v) = a-1$  and  $\rho(y) > a$  and rebalancing will be completed by sharing.
if  $\rho(v) = a-1$ 
  then take 1 son away from y and make it an additional son of v, also
    move one key from x to v and replace it by the leftmost key of y
  fi
completed:

```

EXAMPLE 7. (continuation of 6). The tree of example can be rebalanced by either sharing or fusing depending on the choice of y. If y is the left brother of v, then fusing yields



If y is the right brother of v, then sharing yields



LEMMA 5. Let T be an (a,b) -tree with n leaves. Then Delete(x) takes time $O(\log n)$.

PROOF. Similar to Lemma 4.

We can summarize Lemmas 3, 4 and 5 in

THEOREM 1. Let $S \subseteq U$, $|S| = n$. If S is represented by an (a,b) -tree then the three dictionary operations $\text{Search}(X)$, $\text{Insert}(X)$, $\text{Delete}(X)$ take logarithmic time.

Theorem 1 captures the basic knowledge about balanced trees, which can be found in almost every textbook about data structures and efficient algorithms.

3. SOME NON-STANDARD APPLICATIONS OF BALANCED TREES

In this section, we describe some nonstandard applications of balanced trees. We will learn, that these applications call for a refined analysis of the rebalancing process.

3.1. Concurrent tree manipulation

Many applications, in particular real-time data bank applications, call for a very high transaction rate. Consider for example a computerized banking system, where all transactions are processed on-line. Very high transaction rates can only be supported by concurrent tree manipulation, i.e. by many processors working on the same tree. Concurrent searches cause no problem, however, concurrent insertions/deletions do. Note that the rebalancing operations (splitting, fusing, sharings) require locking of nodes, i.e. whilst node v is rebalanced by one process, the other processes cannot use node v and have to wait if they want to. Also note, that locking node v will (on the average) block many other processes if v is close to the root and that it will block (almost) no other processes if v is close to the leaves. A solution to the concurrency problem must proceed in two steps (historically, the second step was done before the first).

- (a) An analysis of the number and the distribution of rebalancing operations after an insertion and deletion. (MEHLHORN [80]); cf. Theorems 3 and 4 of this paper. Such an analysis will show the feasibility of (a,b) -trees in a parallel environment.
- (b) clever locking protocols for actually achieving a high degree of parallelism (Bayer-Schkolnik)

3.2. Sorting presorted files, merging and finger searches

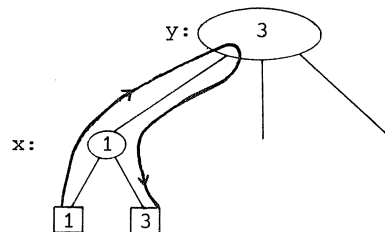
Traditionally, searches start at the root of a tree. However, this is not compulsory.

Consider the following problem. Let $x_1, \dots, x_n$ be a sequence of elements of U , which are to be sorted in increasing order. Suppose, that we know that these sequence is not random, but rather possesses some degree of sortedness. A good measure for sortedness is number of inversions. Let

$$f_i = |\{x_j; j > i \text{ and } x_j < x_i\}|$$

and $F = \sum_{i=1}^n f_i$. Then $0 \leq F \leq n(n-1)/2$, F is called the number of inversions of the sequence $x_1, \dots, x_n$. We can sort this sequence by insertion sort, i.e. we start with the sorted sequence x_n represented by an (a,b) -tree and then insert $x_{n-1}, x_{n-2}, \dots, x_1$ in turn. By Theorem 1 this will take total time $O(n \log n)$. Note however, when we have sorted $x_{i+1}, \dots, x_n$ already and next insert x_i , then x_i will be inserted at the f_i -th position in that sequence. If $F \ll n^2$, then $f_i \ll n$ on the average and hence most x_i 's will be inserted near the beginning. For our search algorithm this implies that it will move down the left spine (the path to the leftmost leaf) for quite some time and only then depart from it. It is cheaper to search for that point by moving the spine *upwards* from the leftmost leaf.

EXAMPLE 7. A search for 2 starting at the leftmost leaf would move through node x to node y , find out that $2 \leq k_1(y)$, turn around and then proceed like



an ordinary search.

So consider the insertion of x_i . We move up the left spine (we assume that son-to-father pointers exist on the left spine) through nodes $v_1, v_2, \dots$ until we hit node v_j with $x_i \leq k_1(v_j)$. We turn around at v_j and proceed as

in an ordinary search. Note that the search for x_i will cost us $O(\text{height}(v_j))$ time units. Since all leaves stored in the subtree rooted at v_{j-2} must have content less than x_i we conclude that $f_i \geq$ number of leaves in subtree rooted at $v_{j-2} \geq 2 \cdot a^{h(v_{j-2})-1}$ by Lemma 2 and hence

$$\text{height}(v_j) = O(\log f_i)$$

The actual insertion of x_i costs $O(s_i)$ time units, where s_i is the number of splits required after adding a new leaf for x_i . This gives the following time bound for the insertion sort

$$\begin{aligned} \sum_{i=1}^n [O(\log f_i) + O(s_i)] &= O(n + \sum_{i=1}^n \log f_i + \sum_{i=1}^n s_i) \\ &= O(n + n \log(F/n) + \sum_{i=1}^n s_i) \end{aligned}$$

since $\sum \log f_i$ is maximal if all f_i 's are equal. It remains to bound the total number of rebalancing operations. In Theorem 2 below it will be shown that $\sum_{i=1}^n s_i = O(n)$. Hence we obtain.

THEOREM 2. Using an (a,b) -tree, $b \geq 2a$, a sequence $x_1, \dots, x_n$ can be sorted in time $O(n(1 + \log F/n))$.

The sorting algorithm of Theorem 2 may be called adaptive. The more sorted sequence $x_1, \dots, x_n$ is, the faster the algorithm. For $F = n \log n$, the running time is $O(n \log \log n)$. This is in marked contrast to all other sorting algorithms, e.g. Heapsort, Quicksort. A careful analysis shows that the algorithm is superior to Quicksort for $F \leq 0.01 \cdot n^{1.5}$ (Mehlhorn, Tsakalidis).

More examples of non-standard searches will be given in Sections 4, 5 and 6.

4. A SEQUENCE OF OPERATIONS ANALYSIS OF (a,b) -TREES, $b \geq 2a$

In this section we study the total cost of sequences of insertions and deletions into (a,b) -trees under the assumption that we start with an initially empty tree. We will show that the total cost is linear in the length of the sequence. The proof follows a general paradigm for analysing the cost of sequences of operations, the bank account paradigm. We will associate a bank account with the tree; the balance of that account measures the balance of

the tree. We will then show that adding or pruning a leaf corresponds to the withdrawal of a fixed amount from that account and that the rebalancing operations (fusing, sharing and splitting) correspond to deposits. Using finally the obvious bounds for the balance of the account, we obtain the result.

THEOREM 3. Let $b \geq 2a$ and $a \geq 2$. Consider an arbitrary sequence of i insertions and d deletions ($n=i+d$) into an initially empty (a,b) -tree. Let SP be the total number of node splittings, F the total number of node fusings and SH be the total number of node sharings. Then

$$(a) \quad SH \leq d \leq n$$

$$(b) \quad (2c-1) \cdot SP + c \cdot F \leq n + c + \frac{c}{a+c-1} (i-d-2)$$

where $c = \min(\min(2a-1, \lceil (b+1)/2 \rceil - a, b - \max(2a-1, \lfloor b+1/2 \rfloor))$. (Note that $c \geq 1$ for $b \geq 2a$ and hence $SP+F \leq n/c + 1 + (n-2)/a$.)

PROOF.

- (a) Node sharing is executed at most once for each deletion. Hence $SH \leq d$.
 (b) We follow the paradigm outlined above.

For a node v (unequal the root) of an (a,b) -tree let the balance $b(v)$ of v be

$$b(v) = \min(\rho(v) - a, b - \rho(v), c)$$

where c is defined as above. Note that $c \geq 1$ for $b \geq 2a$ and $a \geq 2$ and that $\rho(v') = \lfloor (b+1)/2 \rfloor$ and $\rho(v'') = \lceil (b+1)/2 \rceil$ implies $b(v') + b(v'') \geq 2c - 1$ and that $\rho(v) = 2a - 1$ implies $b(v) = c$.

For root r (r will always denote the root in the sequel) the balance is defined as

$$b(r) = \min(\rho(r) - 2, b - \rho(r), c)$$

DEFINITION. (T,v) is a partially rebalanced (a,b) -tree where v is a node of T if

$$(a) \quad \begin{aligned} a-1 &\leq \rho(v) \leq b+1 && \text{if } v \neq r \\ 1 &\leq \rho(v) \leq b+1 && \text{if } v = r \end{aligned}$$

- (b) $a \leq \rho(w) \leq b$ for all $w \neq v, r$
- (c) $2 \leq \rho(r) \leq b$ if $v \neq r$.

Let (T, v) be a partially rebalanced (a, b) -tree. Then the balance of T is defined as the sum of the balance of its nodes

$$b(T) = \sum_{\substack{v \text{ is node of } T \\ v \neq r}} b(v) + b^*(r).$$

FACT 1. Let T be an (a, b) -tree. Let T' be obtained from T by adding a leaf or pruning a leaf. Then $b(T') \geq b(T) - 1$.

PROOF. Obvious. $\square$

FACT 2. Let (T, v) be a partially rebalanced (a, b) -tree with $\rho(v) = b + 1$.

Splitting v and expanding v 's father generates a tree T' with $b(T') \geq b(T) + (2c - 1)$.

PROOF. Since $\rho(v) = b + 1$ we have $b(v) = b^*(v) = -1$.

CASE 1. v is not the root of T . Let x be the father of v . v is split into nodes, v' and v'' say, of arity $\lfloor (b+1)/2 \rfloor$ and $\lceil (b+1)/2 \rceil$ respectively, and the arity of x is increased by one. Hence the balance of x decreases by at most one and we have

$$b(T') \geq b(T) + b(v') + b(v'') - b(v) - 1$$

Furthermore, $b(v) = -1$ and $b(v') + b(v'') \geq 2c - 1$ by the remark above. This shows

$$\begin{aligned} b(T') &\geq b(T) + (2c - 1) - (-1) - 1 \\ &\geq b(T) + (2c - 1). \end{aligned}$$

CASE 2. v is the root of T . Then the father x of v is newly created and hence has arity 2 after splitting of v . Hence

$$\begin{aligned}
b(T') &\geq b(T) + b(v') + b(v'') - b^*(v) + b^*(x) \\
&\geq b(T) + (2c-1) - (-1) + 0.
\end{aligned}$$

FACT 3. Let (T, v) be a partially rebalanced (a, b) -tree with $\rho(v) = a-1$ and $v \neq r$ the root of T . Let y be a brother of v and let x be the father of v .

- (a) if $\rho(y) = a$ then let T' be the tree obtained by fusing v and y and shrinking x . Furthermore, if x is the root and has degree 1 after the shrinking, then x is deleted. Then $b(T') \geq b(T) + c$.
- (b) if $\rho(y) > a$ then let T' be the tree obtained by sharing, i.e. by taking 1 son away from y and making it a son of x . Then $b(T') \geq b(T)$.

PROOF. y and v are fused to a node, say w , of arity $\rho(w) = 2a-1$. Hence $b(w) = b^*(w) = c$.

CASE 1. x is not the root of T . Then the arity of x is decreased by one and hence the balance of x is decreased by at most one. Hence

$$\begin{aligned}
b(T') &\geq b(T) + (b(w) - b(v) - b(y)) - 1 \\
&\geq b(T) + (c - (-1) - 0) - 1 \\
&\geq b(T) + c.
\end{aligned}$$

CASE 2. x is the root of T . If $\rho(x) \geq 3$ before the fusing then the analysis of case 1 applies. Otherwise we have $\rho(x) = 2$ and hence $b^*(x) = 0$ before the fusing, and x is deleted after the fusing and w is the new root. Hence

$$\begin{aligned}
b(T') &= b(T) + (b^*(w) - b(v) - b(y)) - b^*(x) \\
&= b(T) + c - (-1) - 0 - 0 \\
&= b(T) + c + 1.
\end{aligned}$$

In either case we have shown $b(T') \geq b(T) + c$.

- (b) Taking one son away from y decreases the balance of y by at most one. Giving v (of arity $a-1$) an additional son increases $b(v)$ by one. Hence $b(T') \geq b(T)$.

FACT 4. Let T be an (a,b) -tree with m leaves. Then $0 \leq b(T) \leq c + (m-2)\frac{c}{a+c-1}$.

PROOF. For $0 \leq j < c$, let m_j be the number of interior nodes unequal the root of degree $a+j$ and let m_c be the number of interior nodes unequal the root of degree at least $a+c$. Then $b(T) \leq c + \sum_{j=0}^c m_j j$ since the balance of the root is at most c . Furthermore, $2 + \sum_{j=0}^c (a+j)m_j \leq m + \sum_{j=1}^c m_j$ since the expression of the right hand side gives the number of edges in tree T and the expression on the left hand side is a lower bound on that number. Hence

$$\sum_{j=0}^c (a+j-1)m_j \leq m-2$$

Since

$$j/(a+j-1) \leq c/(a+c-1) \quad \text{for } 0 \leq j \leq c$$

we conclude

$$b(T) \leq c + (m-2)c/(a+c-1).$$

In order to finish the banking account paradigm we need to relate deposits, withdrawals and initial and final balance. Since we start with an empty tree T_0 , the initial balance $b(T_0)$ is 0. Next we perform i insertions and d deletions and obtain T_n ($n=i+d$). Since T_n has $i-d$ leaves we conclude $b(T_n) \leq c + (i-d-2)c/(a+c-1)$. Also the total withdrawals are at most n by Fact 1 and the total deposits are at least $(2c-1) \cdot SP + c \cdot F$ by facts 2 and 3. Hence

$$b(T_0) + (2c-1) \cdot SP + c \cdot F - n \leq b(T_n)$$

and thus

$$(2c-1) \cdot SP + c \cdot F \leq n + c + (i-d-2)c/(a+c-1). \quad \square$$

Theorem 3 makes the application to sorting work. For the application to concurrent tree searching, we need a refinement of it. We say that a splitting (fusing, sharing) operation occurs at height h , if node v which is to be split (which is to be fused with its brother y or shares a son with its brother y) has height h ; the height of a leaf being 0. A splitting (fusing)

operation at height h expands (shrinks) a node at height $h+1$. An insertion (deletion) of a leaf expands (shrinks) a node at height 1.

THEOREM 5. (jointly with Ch. Backes). *Let $b \geq 2a$ and $a > 2$. Consider an arbitrary sequence of i insertions and d deletions ($n=i+d$) into an initially empty (a,b) -tree. Let $SP_h (F_h, SH_h)$ be the total number of node splittings (fusings, sharings) at height h . Then*

$$SP_h + F_h + SH_h \leq 2(c+2)n/(c+1)^h$$

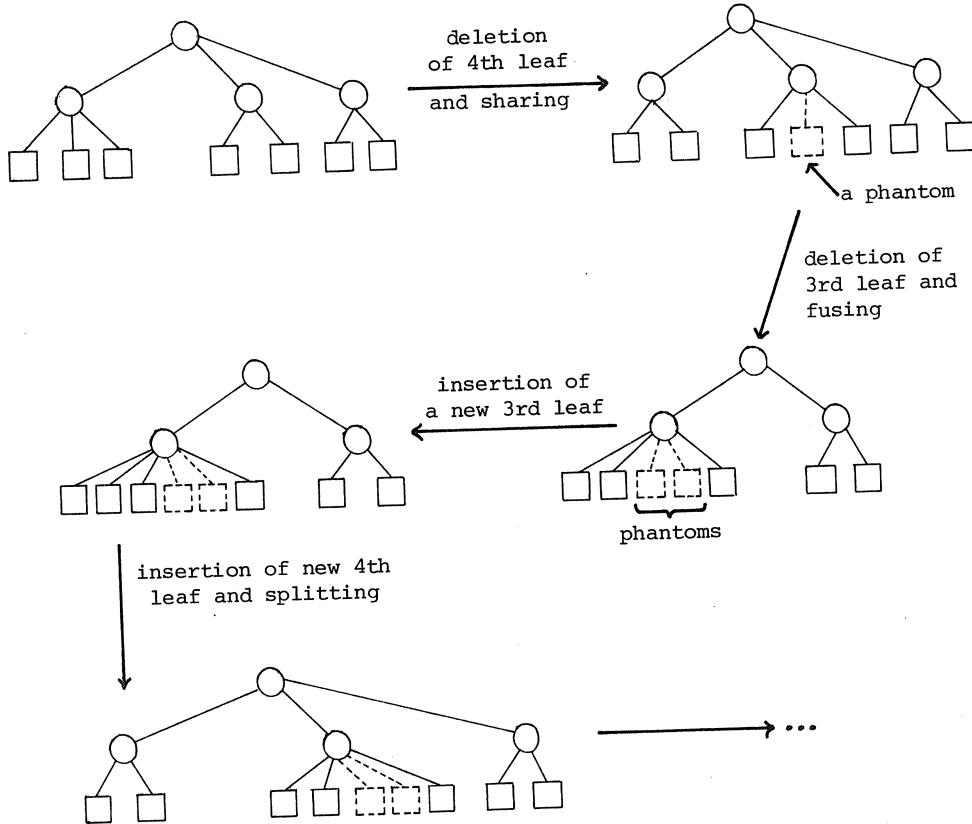
where c is defined as in Theorem 4.

$SP_h + F_h + SH_h$ is the number of insertions and deletions which require rebalancing up to height h or higher. Theorem 5 shows that this number is exponentially decreasing with h (recall $c \geq 1$) and hence that most rebalancing operations occur near the leaves. We conclude that the analysis of Bayer - Schkolnik can be used to show that (a,b) -tree, $b \geq 2a$, behave well in a parallel environment in the presence of *insertions and deletions*.

Theorem 5 makes (a,b) -trees work in a concurrent environment. For the applications of Section 5, we need another extension of Theorem 4, the case of an arbitrary initial tree.

In order to facilitate the following discussion we introduce the following convention: if a leaf is deleted from an (a,b) -tree, it (conceptually) stays as a phantom. The rebalancing algorithms do not see the phantoms, i.e. the arity of nodes is solely determined by the non-phantom leaves.

EXAMPLE 8. A (2,4)-tree with phantoms



THEROEM 6. Let $b \geq 2a$ and let T be an (a,b) -tree with n leaves. Suppose that a sequence of s insertions and t deletions is performed and tree T' is obtained. Then tree T' has $n+s$ leaves, t of which are phantoms. Let $p_1, p_2, \dots, p_{s+t}$ be the positions of the $s+t$ new leaves and phantoms in T' , $p_1 \leq p_2 \leq \dots \leq p_{s+t}$. Let SH (SP, F) be the total number of sharings (splittings, fusions) required to process this sequence. Then

$$SH + SP + F \leq 9(s+t) + 6(\lceil \log(n+s) \rceil + \sum_{i=2}^{s+t} \lfloor \log_a(p_i - p_{i-1} + 1) \rfloor)$$

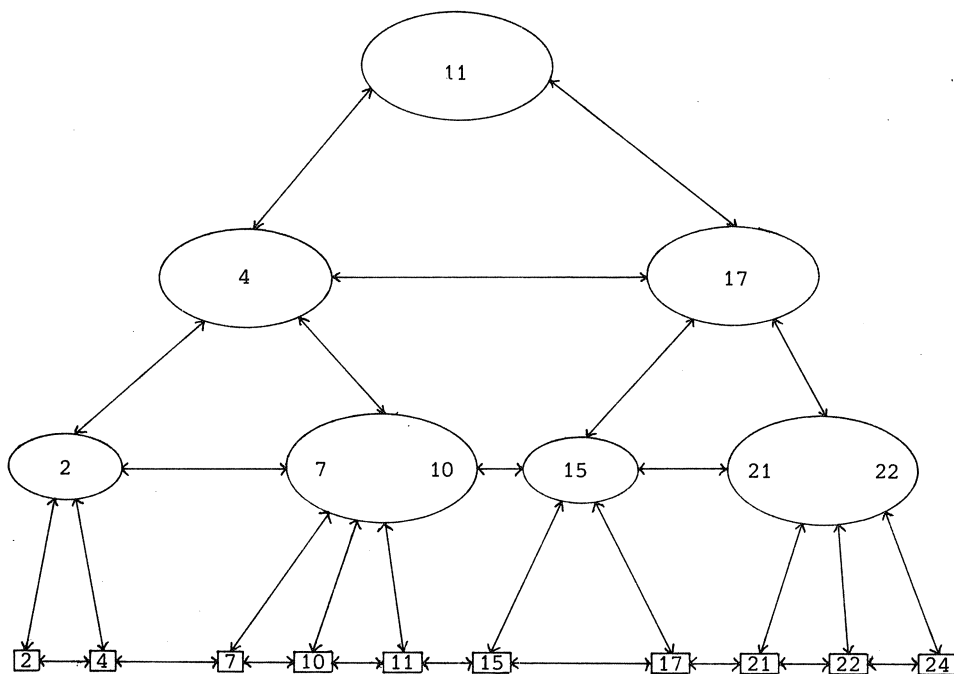
5. A REPRESENTATION FOR LINEAR LISTS WITH FINGERS

In this section we use (a,b) -trees in order to represent linear lists. This section follows Brown-Tarjan very closely.

Let S be a subset of ordered universe U . A finger into set S is a pointer to an element of S . Fingers may be used to indicate areas of high activity in list S . (a,b) -trees as they stand do not support efficient search in the vicinity of fingers. This is due to the fact that neighbouring leaves may be connected only by a very long path. Therefore we introduce *level-linked* (a,b) -trees.

In level linked (a,b) -trees all tree edges are made traversable in both directions (i.e. there are also pointers from sons to fathers); in addition each node has pointers to the two neighbouring nodes on the same level.

EXAMPLE 9. A $(2,4)$ -level linked tree for list 2, 4, 7, 10, 11, 15, 17, 21, 22, 24.



A finger into a level-linked (a,b) -tree is a pointer to a leaf. Level-linked (a,b) -trees allow very fast searching in the vicinity of fingers.

LEMMA 6. Let p be a finger in an (a,b) -tree T . A search for a key k which is

d keys away from p takes time $\theta(1 + \log d)$.

PROOF. We first check whether k is to the left or right of p , say k is to the right of p . Then we walk towards the root, say we reached node v . We check whether k is a descendant of v or v 's right neighbour on the same level. If not, then we proceed to v 's father. Otherwise we turn around and search for k in the ordinary way.

Suppose that we turn around at node w of height h . Let u be that son of w which is on path to the finger p . Then all descendants of u 's right neighbour lie between the finger p and key k . Hence the distance d is at least 2^{h-1} . The time bound follows. $\square$

LEMMA 7. A new leaf can be inserted in a given position of a level-linked (a,b) -tree in time $\theta(1+s)$, where s is the number of splittings caused by the insertion.

LEMMA 8. A leaf can be deleted from a level-linked (a,b) -tree in time $\theta(1+f)$, where f is the number of node fusings caused by the deletion.

LEMMA 9. Creation or removal of a finger in a level-linked (a,b) -tree takes time $\theta(1)$.

Now we apply our results of Section 4 and show that even though the search time in level linked (a,b) -trees can be greatly reduced by maintaining fingers, it still dominates the total execution time, provided that $b \geq 2a$.

THEOREM 7. Let $b \geq 2a$. Then any sequence of searches, finger creations, finger removals, insertions and deletions starting with an empty list takes time

$$O(\text{total cost of searches})$$

if a level-linked (a,b) -tree is used to represent the list.

PROOF. Let n be the length of the sequence. Then the total cost for the searches is $\Omega(n)$ by Lemma 1. On the other hand the total cost for the finger creations and removals is $O(n)$ by Lemma 4 and the total cost of insertions and deletions is $O(n)$ by Lemma 2 and 3 and Theorem 1. $\square$

THEOREM 8. *Let $b \geq 2a$. Let L be a sorted list of n elements represented as a level-linked (a,b) -tree with one finger established. Then in any sequence of searches, finger creations, insertions, and deletions, the total cost of the sequence is*

$$O(\log n + \text{total cost of searches})$$

PROOF. Let S be any sequence of searches, finger creations, insertions and deletions containing exactly s insertions and d deletions. Let T_{final} be the (a,b) -tree, which represents a list L after S is performed. Assume that we keep deleted elements as phantoms. Let the $n+s$ leaves (real leaves and phantoms) of T_{final} be named $0, 1, \dots, n+s$ from left to right. Assign to each leaf p a label $\ell(p)$, whose value is the number of leaves lying strictly to the left of p which were present initially and which did not become phantoms. These labels lie in the range $0, \dots, n-1$.

We now proceed precisely as in the proof of Theorem 4 in Brown-Tarjan. The details are left to the reader. $\square$

6. APPLICATIONS OF LEVEL-LINKED (a,b) -TREES

In general, we advise to use level-linked (a,b) -trees ($b \geq 2a$) whenever there are (maybe time-varying) areas of high activity. The finger will make the searches very fast, and finger creations, insertions and deletions take constant time on the average. This situation occurs quite frequently in the implementation of event lists.

More specifically, level-linked (a,b) -trees permit the optimal realization of many set operations.

THEOREM 9. *Let A and B be sets represented as level-linked (a,b) -trees, $b \geq 2a$.*

- a) *Insert(A, x), Delete(A, x), Search(A, x), Concatenate(A, B) and Split(A, x) can be done in logarithmic time. (Here Concatenate(A, B) = $A \cup B$ if $\max A < \min B$ and undefined otherwise and Split(A, x) = (A_1, A_2) where $A_1 = \{a; a \in A \text{ and } a \leq x\}$ and $A_2 = \{a; a \in A \text{ and } a > x\}$).*
- b) *Let $n = \max(|A|, |B|)$ and $m = \min(|A|, |B|)$. Then $A \cup B$, $A \oplus B$, $A \cap B$, $A \setminus B$ can be constructed in time $O(\log(\frac{n+m}{m}))$.*

PROOF.

- (a) The algorithms are the same as for 2-3 trees. We refer the reader to MEHLHORN [77] for details.
- (b) we first show how to perform $A \oplus B$.

Assume w.l.o.g. $|A| \geq |B|$. The algorithm is as follows:

- (a) establish a finger at the first element of A
- (b) while B not exhausted
- do
- (b.1) take the next element, say x, of B and search for it in A starting at the finger
- (b.2) insert or delete x from A, whatever is appropriate
- (b.3) establish a finger at the position of x in A and destroy the old finger
- od

Let $p_1, \dots, p_m$, $m = |B|$, be the positions of the elements of B in the set $A \cup B$. Then the above program takes time ($|A| = n$)

$$O(m + \log(n+m) + \sum_{i=1}^{m-1} \log(p_{i+1} - p_i + 1))$$

This expression is maximized for $p_{i+1} - p_i = (n+m)/m$ for all i, and then has value $O(\log(n+m) + m \log((n+m)/m)) = O(m \log((n+m)/m)) = O(\log \binom{n+m}{m})$.

In the case of $A \cup B$, we only do insertions in line (b.2). In the case of $A \cap B$, we collect the elements of $A \cap B$ in line (b.2) (there are at most m of them) and construct a level-linked (a,b)-tree for them afterwards.

Finally we have to consider $A \setminus B$. If $|A| \geq |B|$, then we use the program above. If $|A| < |B|$ then we scan through A linearly, search for the elements of A in B as described above (roles of A and B reversed) and delete the appropriate elements from A. Apparently, the same time bound holds.

Note that there are $\binom{n+m}{m}$ possibilities for B as a subset of $A \cup B$. Hence $\log \binom{n+m}{m}$ is also a lower bound on the complexity of union and symmetric difference.

7. CONCLUSION

We carried out a detailed sequence of operations analysis of (a,b)-trees in the case $b \geq 2a$. Our analysis shows that weak B-trees are superior to ordinary B-trees ($b = 2a - 1$) in two areas:

- (a) concurrent usage of trees: in weak B-trees rebalancing operations are concentrated near the leaves even in the presence of insertions and deletions. Thus a high degree of parallelism is guaranteed.
- (b) finger searches: level-linked weak B-trees support finger searches. Although finger searches are usually more efficient (in particular if there is locality of reference) than ordinary searches, total search time still dominates the cost of sequences of operations. Level-linked weak B-trees allow the efficient realization of many set operations. In particular, they support optimal update of a master file against a file of updates.

8. REFERENCES

- ADEL'SON-VELSKII, . LAUDIS, *An algorithm for the organisation of information*, Soviet Math. Dokl. 3, (1962), 1259-1262.
- BAYER, R. & E. MCCREIGHT, *Organization and maintenance of large ordered indices*, Acta Informatica 1, (1972), 173-189.
- BAYER, R. & M. SCHKOLNIK, *Concurrency of operations on B-trees*, Acta Informatica 9, (1977), 1-21.
- AHO, HOPCROFT & ULLMAN, *The design and analysis of compute algorithms*, Addison-Wesley, 1974.
- BROWN, M. & R. TARJAN, *A representation for linear lists with movable fingers*, 10th ACM Symposium on Theory of Computing, 1978, 19-28.
- GUIBAS, L., E. MCCREIGHT, M. PLASS & J. ROBERTS, *A new representation for linear lists*, 9th ACM Symposium on Theory of Computing, 1977, 49-60.
- HUDDLESTONE, & K. MEHLHORN, *A new data structure for representing sorted lists*, to appear in Acta Informatica.
- MEHLHORN, K., *Effiziente Algorithmen*, Teubner-Verlag, Studienbücher Informatik, 1977.
- MEHLHORN, K. & A. TSAKALIDIS, *Sorting presorted files*, in preparation; a preliminary version appeared as: K. Mehlhorn (ed.), *Sorting Presorted Files*, 4th GI-Conference on Theoretical Computer Science, Aachen 1979, LNCS, Vol. 67, 199-212.

- NIEVERGELT, J. & E.M. REINGOLD, *Binary search trees of bounded balance*,
SIAM J. of Computing, Vol. 2, No. 1, March 1973.
- YAO, A.C.C., *Should a table be sorted*, IEEE FOCS, 1978.

VERZAMELINGSMANIPULATIE OP EEN KLEINE COMPUTER

TH. P. VAN DER WEIDE

0. INLEIDING

Een microcomputer wordt gekenmerkt door een (relatief) trage processor met een klein adresseringsbereik. Zeer gangbaar is de 8-bit processor, die een dubbel register gebruikt om te adresseren zodat het adresseringsbereik 64K wordt. Aanvankelijk werd de microcomputer uitgerust met floppy disks als achtergrondgeheugen. Dit medium is goedkoop, zeer betrouwbaar, eenvoudig hanteerbaar, heeft een redelijke capaciteit ($\frac{1}{2}$ M) en is eenvoudig programmeerbaar dankzij de goede tussenliggende hardware (de controller). Met de komst van double density ($\frac{1}{2}$ M) en double sided (1M) wordt tegenwoordig een acceptabele capaciteit bereikt. De limiet voor dit medium lijkt nog niet bereikt te zijn en er wordt al gesproken over een capaciteit van 25M. De microcomputer uitgerust met (meestal) 2 tot 4 floppy disk drives wordt typisch gebruikt in een zogenaamde single-user omgeving. De beperkte grootte van de floppy disk maakt het medium minder bruikbaar in de praktijk om er redelijk grote verzamelingen op bij te houden. De machine zal meestal gebruikt worden als eenvoudige kantoormachine, wordprocessor, hobby computer of als ontwikkelsysteem.

De beschikbaarheid van de zogenaamde hard disk zou het beeld van de microcomputer veranderen. Behoudens de eenvoudige overdraagbaarheid van de floppy disk bleven de goede floppy eigenschappen gehandhaafd, terwijl de capaciteit en snelheid beduidend groter zijn. Het wordt nu wel mogelijk grote verzamelingen bij te houden. In de praktijk is men nu niet meer tevreden met het single-user concept, en zal men de verzameling vanaf meerdere plaatsen willen kunnen inspecteren en/of bijwerken, waarmee het multi-user idee geïntroduceerd is. De relatieve traagheid van de microprocessor maar vooral het kleine adresseringsbereik lijken dit evenwel in de weg te staan. De traagheid van de microprocessor is het minste probleem, aangezien enerzijds de bezettingsgraad in de praktijk toch niet al te groot is,

terwijl de toepassingen erop gericht zijn records uit het achtergrondgeheugen te manipuleren, zodat de diverse processen in eerste instantie voornamelijk elkaars idle time gaan opvullen. Het beperkte adresseringsbereik is wel een probleem. Om een idee te krijgen, van de 64K geheugen zal het operating systeem zeker wel 16K gebruiken. Het runtime system van een hogere programmeertaal kan best zo'n 20K kosten. Om microcomputers toch in een multi-user omgeving te kunnen toepassen zijn oplossingen ontwikkeld waarvan we bankswitching en de multiprocessor configuratie noemen.

Wat zijn nu de beperkingen waar de programmeur op een microcomputer tegenaan loopt. Allereerst zal hij om een redelijk groot datagebied te hebben de programmagrootte dienen te beperken. Segmentatie van het programma is de hierbij gebruikte techniek, waarbij steeds alleen de relevante programma segmenten in het geheugen aanwezig zijn. Dan zal de programmeur moeten beslissen hoe het datagebied te gebruiken. Om een file efficiënt te kunnen manipuleren dient (de) informatie over de disk allocatie aanwezig te zijn in het geheugen. Het inlezen van deze informatie noemt men het openen van de file. Bij een file zal men een buffer hebben ten behoeve van de disk IO. Het datagebied zal derhalve voldoende groot moeten zijn om een redelijk aantal files geopend te kunnen hebben. Hoe groot dient men de recordgrootte van een file te kiezen? Allereerst zijn de tijdskenmerken van het achtergrondmedium nu van belang. De accesstijd naar het achtergrondgeheugen splitst men in twee componenten: seektijd en transfertijd. De seektijd dient hierbij opgevat te worden als de tijd tussen het starten van de IO operatie en het moment waarop de transfer begint. De verhouding van seektijd ten opzichte van transfertijd is nu van belang. Is deze verhouding groot, dan zal men gebaat zijn bij een grote recordgrootte, anders bij een kleine.

Het volgende probleem dat zich voordoet is, welke datastructuur te kiezen om de verzameling in te representeren. Uiteraard hangt deze keuze sterk af van de operaties die men op de verzameling uit wil voeren. Van belang is verder de efficiëntie waarmee de operaties binnen de datastructuur geïmplementeerd kunnen worden, en de grootte van deze implementaties. In de complexiteitstheorie wordt meestal de implementatie grootte niet beschouwd terwijl de complexiteit gemeten wordt in CPU tijd. Dat laatste is redelijk op grote computersystemen. In de microcomputer wereld kan men echter, vanwege de beperkte geheugengrootte vaak beter de complexiteit meten in het aantal disk accesses. Het zal duidelijk zijn dat hiermee vele resultaten uit de complexiteitstheorie hun waarde gaan verliezen. Een volgende punt

wordt nu de volgorde waarin in de implementatie de records op de disk gemanipuleerd worden. Door een volgorde te kiezen waarbij de arm van de disk steeds in één richting bewogen wordt kan men sterk besparen op seek tijden. Vanwege de reeds genoemde laag veronderstelde bezettingsgraad zal dit zelfs in een multi-user omgeving winst opleveren. Een volgend punt vormen vanwege de kleine recordgrootte de extra onderdelen (pointers) die in de records opgenomen dienen te worden om de samenhang in de datastructuur vast te leggen. In dit verband dienen de zogenaamde impliciete datastructuren genoemd te worden, waarbij geen pointers nodig zijn om de samenhang vast te leggen (denk bijv. aan een array). Zie hiertoe [1]. Afhankelijk van de recordgrootte bepaalt men ingeval van zoekbomen de branching degree. Merk op dat deze dus sterk hardware gebonden zal zijn!

1. SORTEREN EN SELECTEREN

In dit hoofdstuk bekijken we wat de gevolgen zijn van een beperkt RAM geheugen op de constructie en analyse van algoritmen voor problemen waarbij alle elementen van de verzameling in kwestie een rol spelen bij de oplossing van het probleem. Deze problemen zijn voornamelijk onderzocht voor het geval dat voldoende RAM geheugen aanwezig is. Voor een kleine computer is dit evenwel niet realistisch. We zijn nu gedwongen het beschikbare geheugen verstandig te beheren, door bijvoorbeeld recycling van geheugenplaatsen waarvan de inhoud niet meer relevant is. Wanneer we voldoende geheugen hebben kunnen we eenvoudig aan het begin van het algoritme de gehele verzameling inlezen om dan vervolgens de oplossing van het probleem te gaan berekenen. Dit staat bekend als een off-line methode. Is het geheugen te klein voor een off-line methode, dan kunnen we een on-line methode proberen. We lezen de verzameling nu stapsgewijs in, en elimineren steeds tenminste zoveel elementen uit de mogelijke oplossingsverzameling dat we weer een nieuw element kunnen inlezen. Wanneer een on-line oplossing mogelijk is spreken we van een *beperkt* werkgeheugen. Is ook een on-line methode niet mogelijk (*klein* werkgeheugen) dan zijn we niet in staat om tijdens het lezen van de verzameling voldoende informatie te onthouden om aan het eind de mogelijke oplossingsverzameling tot (hooguit) een singleton te reduceren. We zullen de verzameling dan nogmaals moeten inspecteren om een verdere reductie te bewerkstelligen. We zullen spreken van een *p-pass* algoritme, als de verzameling (hooguit) *p* maal gelezen dient te worden.

We veronderstellen in dit geval dat het achtergrondgeheugen niet beschreven mag worden. Is dit wel het geval dan kunnen tussenresultaten tijdelijk naar het achtergrondgeheugen geplaatst worden, waardoor een groter (maar gemiddeld beduidend trager) werkgeheugen geïmplementeerd wordt.

1.1. Sorteren

Als eerste voorbeeld bekijken we het sorteerprobleem. Het is bekend dat de tijd om een verzameling van n elementen te sorteren (worst case) $\theta(n \log n)$ bedraagt. De volgende stelling geeft aan hoeveel werkgeheugen vereist is voor een p -pass sorteeralgoritme van n elementen (zie [2]).

STELLING 1.1. Een p -pass sorteeralgoritme vereist om een verzameling van n elementen te sorteren $\theta(\frac{n}{p})$ werkgeheugen.

Veronderstel nu dat de grootte van het werkgeheugen s ($s \geq 2$) bedraagt. Om een verzameling ter grootte n ($n > s$) te sorteren, bepalen we in de eerste pass de $s-1$ grootste elementen en onthouden het kleinste gevonden element om in de volgende pass als filter dienst te doen, waardoor reeds gesorteerde elementen genegeerd kunnen worden. In deze volgende slag bepalen we de volgende $s-1$ grootste elementen, etcetera. Dit algoritme vereist $p = \frac{\lceil n-1 \rceil}{s-1}$ passes, en de tijdscomplexiteit bedraagt $O(pn + pn \log s) = O(\frac{n^2}{s} \log s)$.

1.2. Selectie

Het selectieprobleem bestaat uit het bepalen van het r -de element van een verzameling. Het is bekend dat dit element bepaald kan worden in lineaire tijd, wanneer voldoende werkgeheugen voorhanden is.

We zullen nu eerst een on-line oplossing bekijken voor de selectie van het r -de element. Uiteraard veronderstellen we $r \leq \frac{\lceil n \rceil}{2}$ als n de grootte van de verzameling is. De eerste r elementen zijn alle kandidaten om het r -de element te zijn. Pas bij het $(r+1)$ -ste element kunnen we een element van verdere deelname uitsluiten. Als het werkgeheugen niet $r+1$ elementen kan bevatten, hebben we na r elementen gelezen te hebben al een gelezen element moeten overschrijven, en zullen we in het algemeen een volgende pass voor de verzameling nodig hebben. Wil een on-line oplossing dus mogelijk zijn, dan moet het werkgeheugen grootte $\geq r+1$ hebben.

Laat het werkgeheugen grootte $r+1$ hebben. We lezen nu de eerste

r elementen in. Voor alle volgende elementen x voeren we het volgende uit:

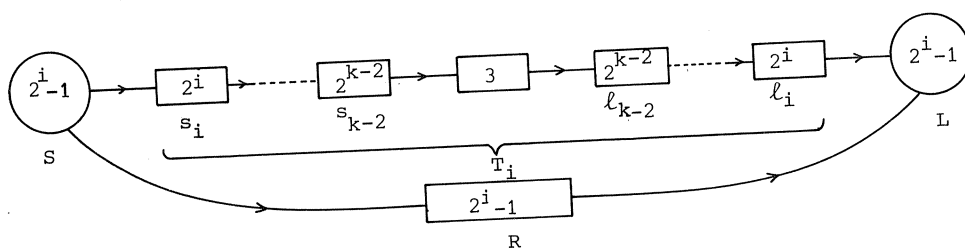
laat m = minimale der aanwezige r elementen
als $x < m$ dan gooi x weg,
gooi in het andere geval m weg, en laat x aanwezig

Nadat we alle elementen gelezen hebben is het minimale der aanwezige r elementen het gezochte element. Door nu de aanwezige elementen als een heap ([3]) te organiseren volgt eenvoudig dat dit algoritme tijd $O(n \log(r+1))$ verstookt. Het algoritme kenmerkt zich doordat steeds te kleine elementen worden uitgesloten. In geval van de mediaan ($r = \frac{\lceil n \rceil}{2}$) is evenwel tweezijdige uitsluiting mogelijk, waardoor een lineair algoritme mogelijk wordt. We beschrijven het algoritme uit [4].

1.2.1. On-line selectie

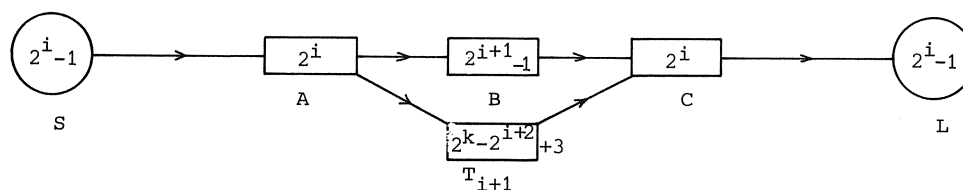
Dit algoritme leest, na initialisatie, in elke slag de elementen in steeds grotere groepen binnen. Met behulp van de opgebouwde datastructuur kan dan op eenvoudige wijze, aan beide zijden, een groep elementen ter grootte van het aantal in deze slag ingelezen elementen uitgesloten worden.

Veronderstel voor het gemak $n = 2^{k+1} + 1$, zodat $r = 2^k + 1$, terwijl het werkgeheugen grootte $2^k + 2$ heeft. Aan het begin van de i^e slag ziet de datastructuur er als volgt uit:

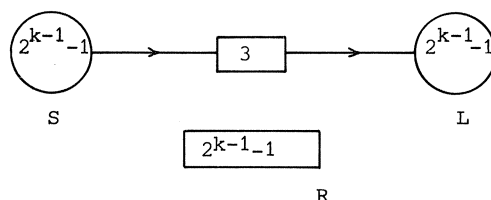


We hebben tot nog toe $2^k + 2^i$ elementen gelezen. Hiervan zijn de elementen uit S en R uitgesloten, omdat ze $2^k + 1$ grotere, resp. kleinere elementen hebben. Aanwezig zijn de elementen uit T_i en R .

Het aantal vrije plaatsen in het werkgeheugen is derhalve $(2^k+2) - (2^k-2^i+2) = 2^i$. Van de verzamelingen s_j wordt het maximale element bekend verondersteld, van de verzamelingen ℓ_j het minimale element. Van de elementen uit R wordt niets verondersteld. Aangezien er 2^i vrije plaatsen zijn, kunnen we 2^i nieuwe elementen (N) inlezen. We gaan nu van de verzameling $\ell_i \cup s_i \cup R \cup N$ de 2^i kleinste (A) en 2^i grootste (C) bepalen, door m.b.v. 3 mediaan bepalingen deze (grootweg) in vieren te delen.

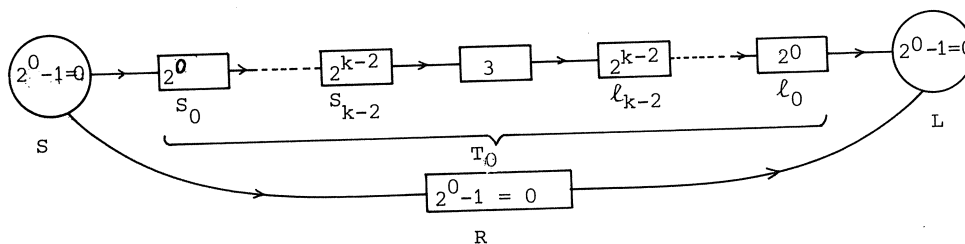


Elk element uit A heeft nu $(2^{i+1}-1) + (2^{k-2^{i+2}}+3) + 2^i + (2^i-1) = 2^{k+1}$ grotere elementen. Bij gevolg kunnen we de elementen uit A weggooien (in S stoppen). Analoog voor de elementen uit C . B wordt nu de verzameling R , waarna de datastructuur weer de gewenste vorm heeft. Na $k-2$ slagen resulteert de structuur:

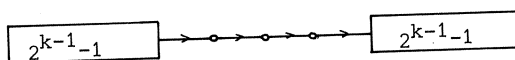


We hebben nu 2^k+2^{k-1} elementen gelezen, zodat nog $2^{k-1}+1$ elementen ongelezen zijn. In het werkgeheugen hebben we nu 2^{k-1} vrije plaatsen. We lezen 2^{k-1} elementen in, en bepalen van de aanwezige elementen de twee in het midden. De middelste van deze twee en het nog ongelezen (laatste) element is de gevraagde mediaan.

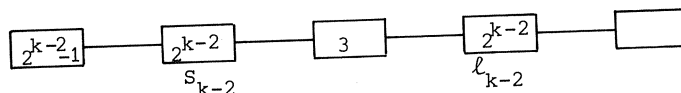
Blijft nog de initialisatie van het proces. Aan het begin van de 0-de slag zijn 2^k+1 elementen gelezen, en geordend volgens de structuur:



Tijdens de initialisatie dienen we derhalve 2^k+1 elementen te lezen en hiervan de 2^i grootste en kleinste elementen ($i = k-2, \dots, 0$) te bepalen. Dit doen we door allereerst (m.b.v. een imaginair element) in twee mediaanbepalingen het 2^{k-1} -de kleinste en grootste element te bepalen, en vinden de structuur:



Van de $2^{k-1}-1$ kleinste (resp. grootste) elementen bepalen we de mediaan, en vinden:



Vervolgens bepalen we de mediaan van de $2^{k-2}-1$ kleinste (resp. grootste) elementen, etcetera.

Tenslotte bepalen we de complexiteit van het beschreven algoritme. Zij $M(j)$ de kosten van een lineair algoritme om de mediaan uit j elementen te bepalen. De initialisatiekosten bedragen:

$$2M(2^k+2) + \sum_{j=0}^{k-1} M(2^j-1) \leq M(3 \cdot (2^k+1) - k)$$

Dan maakt het algoritme in slag i kosten:

$$M(4 \cdot 2^i - 1) + 2 \cdot M(2 \cdot 2^i - 1) \leq M(8 \cdot 2^i - 3)$$

zodat de slagen samen kosten:

$$\sum_{i=0}^{k-2} M(8 \cdot 2^i - 3) \leq M(8 \cdot (2^{k-1} - 1) - 3 \cdot (k-1)) = M(4 \cdot (2^k + 1) - 3k - 9) .$$

Tenslotte worden nog kosten $2M(2^k + 2)$ gemaakt

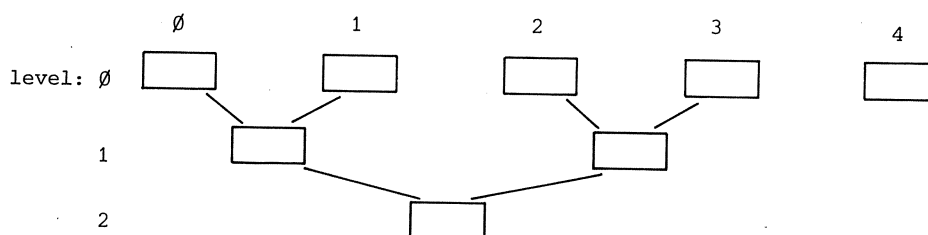
Samen levert dit kosten:

$$\begin{aligned} M(3(2^k + 1) - k) + M(4 \cdot (2^k + 1) - 3k - 9) + M(2(2^k + 1) + 2) \\ \leq M(9(2^k + 1) - 4k - 7) = O(n) \end{aligned}$$

1.2.2 Selectie in klein werkgeheugen

Wanneer we het r -de element van een verzameling moeten bepalen in een werkgeheugen kleiner dan $r+1$ kunnen we, zoals reeds gezegd, geen on-line algoritme gebruiken. We hebben dan onvoldoende geheugen om alle elementen die nog in aanmerking komen om het gezochte element te zijn, in het werkgeheugen te bewaren. We zullen de verzameling nu meerdere malen moeten lezen, en proberen in elke pass zoveel mogelijk elementen uit te sluiten. We beschrijven de methode uit [2]. Dit algoritme houdt twee variabelen u en v bij, zodanig dat aan het begin van elke pass geldt. $\rho(u) \leq r \leq \rho(v)$, waarbij $\rho(j)$ de rang van j in de verzameling is. Tijdens de pass wordt de precieze rang van u en v berekend, en op het eind van de pass worden nieuwe u en v gekozen.

Het algoritme probeert tijdens de pass een evenwichtige sample uit de gereduceerde verzameling te trekken en kiest de nieuwe u en v vervolgens uit deze sample. Het algoritme leest steeds k elementen en doet een stap verder ter bepaling van de sample. Nadat de t -de groep van k elementen is binnen gelezen, wordt deze groep allereerst gesorteerd. We noemen dit een level $\emptyset$ sample. Twee opeenvolgende level $\emptyset$ samples worden steeds gecombineerd tot een level 1 sample, evenzo voor level 1 samples, enzovoort. De combinatie van twee level j samples tot een level $(j+1)$ sample geschiedt door de even geordende elementen uit beide level j samples te mergen.



Wanneer de t -de sample van level $\emptyset$ geconstrueerd is kan men eenvoudig aan de binaire ontwikkeling van t zien of deze sample met de vorige level $\emptyset$ sample gecombineerd dient te worden, en zo ja of dit proces zich herhaalt voor hogere levels.

Op het eind van de pass vullen we, indien nodig, aan met imaginaire elementen tot één enkele sample van zeg level i resteert.

Wat kunnen we zeggen over de rangorde van de elementen uit deze sample in de gereduceerde verzameling? Daartoe voeren we in:

$$\begin{aligned} L(i,j) &= \text{minimale rang van } j\text{-de element uit sample van level } i \\ M(i,j) &= \text{maximale rang van } j\text{-de element uit sample van level } i \end{aligned}$$

Dan is eenvoudig aan te tonen:

$$\begin{aligned} L(i,j) &= j \cdot 2^i \\ M(i,j) &= (i+j-1) \cdot 2^i + 1. \end{aligned}$$

Met $\rho(u)$ en $\rho(v)$ kunnen we de rang in de gereduceerde verzameling eenvoudig uit de rang in de oorspronkelijke verzameling bepalen. Van de nieuw te kiezen elementen u en v moeten we kunnen garanderen.

$$\rho(u) \leq r \leq \rho(v).$$

Met andere woorden, we moeten zorgen dat:

$$\begin{aligned} M(i,u) &\leq r, \\ L(i,v) &\geq r. \end{aligned}$$

We kiezen derhalve in de sample van level i :

$$\begin{aligned} \text{voor } u \text{ het element met rang } \frac{\lceil r \rceil}{2^i} - i, \\ \text{voor } v \text{ het element met rang } \frac{\lceil r \rceil}{2^i}. \end{aligned}$$

Wanneer we de verzameling dusdanig gereduceerd hebben na een aantal passes, dat deze in het werkgeheugen past, kunnen we het gevraagde element met een rechtstreekse selectiemethode bepalen. Het aantal passes dat het algoritme maakt kunnen we bepalen als we weten hoe groot de reductiesnelheid van de verzameling is. Als we uitgaan van n elementen, dan bedraagt het aantal elementen in de gereduceerde verzameling:

$$M(i,v) - L(i,u) - 1 = (2i-1) \cdot 2^i = O\left(\frac{n}{k} \log \frac{n}{k}\right) = O\left(\frac{n}{k} \log n\right).$$

na p passes is de verzameling derhalve gereduceerd tot een grootte

$$O(n \cdot (\frac{\log n}{k})^p).$$

Resteert nog de vraag hoe groot k is. Tijdens de pass hebben we maximaal werkgeheugen nodig voor $(i+1)$ samples, d.w.z. $(i+1) \cdot k$ locaties. We kunnen hierop besparen door daarvan in samples met level > 0 de oneven gerangschikte elementen weg te laten. Verder hebben we twee locaties voor de filterelementen u en v nodig. Samengevat heeft het algoritme dus $k+2+ik/2$ geheugenlocaties nodig. Als de grootte van het werkgeheugen S bedraagt moeten we k dus maximaal kiezen, onder de voorwaarden.

$$k+2+ik/2 \leq S,$$

$$2^{i-1} \cdot k < n \leq 2^i k.$$

We kiezen nu $k = \frac{\lceil S/2 \rceil}{\log n}$. Verder vinden we voor S de restrictie $S = \Omega(\log^2 n)$

Samengevat:

STELLING. Een p -pass algoritme om het r -de element uit een verzameling van n elementen te kiezen vereist hooguit

$$O\left(\left(\frac{n}{\log^2 n}\right)^{1/p} \cdot \log^2 n\right) \text{ geheugen.}$$

Wat betreft de complexiteit van dit algoritme:

STELLING. Het boven beschreven algoritme heeft complexiteit

$$O(pn \log^2 n \cdot \left(\frac{n}{\log^2 n}\right)^{1/p})$$

BEWIJS. De kosten per pass bedragen $O(2^{i-1} \cdot k \log k) = O(n \log k) = O(n \log S)$.

$$\text{Samen: } O(pn \log S) = O(pn \cdot \left(\frac{n}{\log^2 n}\right)^{1/p} \cdot \log^2 n). \quad \square$$

Voor erg kleine geheugens lijkt men aangewezen op de sorteermethode.

Kosten:

$$O\left(\frac{n^2}{s} \log s\right) = O(n^2 \log(\log^2 n)) = O(n^2 \log \log n)$$

Passes: $O\left(\frac{n}{s}\right) = O(n)$

1.2.3 Probabilistische methodes

De laatste tijd wordt veel aandacht besteed aan z.g. probabilistic algorithms. Dit zijn algoritmes die bijna altijd zeer snel de oplossing bepalen, en in de andere gevallen of zeer traag zijn, of mislukking rapporteren. Het welbekende quicksort algoritme om een verzameling te sorteren is hier een voorbeeld van: quicksort zal de verzameling bijna altijd (in de praktijk) zeer snel sorteren, hoewel het algoritme een slechte worst case complexiteit heeft.

Ook voor het selectieprobleem is een aanzet gegeven tot dergelijke algoritme's in kleine geheugens (zie [2]). We beschrijven het mediaan algoritme uit [2]. Wanneer we de verzameling uniform verdeeld beschouwen, vindt dit algoritme met kans $> 1-\epsilon$ voor zekere $\epsilon > 0$ in p passes het gevraagde element, terwijl het vereiste werkgeheugen slechts $O(n^{1/2p})$ bedraagt. Met kans $< \epsilon$ mislukt het algoritme.

Laat de grootte van het werkgeheugen s bedragen. Van de reeds gelezen elementen zal het algoritme nu steeds de $s-1$ elementen bewaren die vermoedelijk het dichtst bij de mediaan liggen. Laat hierbij h (resp. ℓ) het aantal elementen dat we als te groot (resp. te klein) beschouwd hebben. Als nu nadat alle elementen ingelezen zijn $h+1 \leq \frac{\lceil n \rceil}{2} \leq n-\ell$ geldt, dan is het element met rang $\frac{\lceil n \rceil}{2} - h$ uit de aanwezige $s-1$ elementen de gezochte mediaan. In het andere geval hebben we de mediaan ten onrechte als te groot of te klein beschouwd, en is het algoritme mislukt.

Bij de analyse van dit algoritme beschouwt men de waarden die de variable $d = h-\ell$ aanneemt. De succes conditie voor het algoritme luidt dan: $|d| < s-1$. De voortgang van deze variable kan men beschouwen als een random walk uitgaande van het nulpunt. Aldus kan men beargumenteren dat bij elke $\epsilon > 0$ er een constante $c > 0$ bestaat, zodat de kans dat de variable d na $c \cdot s^2$ stappen de waarde $s-1$ bereikt $< \epsilon$ is. Derhalve is het vereiste werkgeheugen $\theta(n^{\frac{1}{2}})$.

In de multipass methode leest men voor geschikt gekozen constante $c > 0$, afhankelijk van ϵ , de eerste $c \cdot s^2$ elementen binnen. De rest van de elementen

wordt nu gelezen om de precieze rang van de aanwezige elementen te bepalen. Is de mediaan niet aanwezig, dan voert men in de volgende stap het algoritme uit voor het interval waarbinnen de mediaan moet liggen. Met grote kans zal de verzameling nu gereduceerd zijn tot hooguit $d \frac{n}{s^2}$ elementen, waarbij d van ϵ afhangt. Deze subset is eveneens uniform verdeeld.

2. VERZAMELINGSMANIPULATIES

We zullen nu kijken naar de verzamelingsoperaties INSERT & FIND waarbij we de complexiteit meten als het aantal accesses naar het achtergrondgeheugen. Keys zijn onderling vergelijkbaar. De datastructuur die we zullen kiezen zal in de vorm van een boom zijn. Om de diepte van de boom beperkt te houden kunnen we gebruik maken van de diverse balanceer technieken, eventueel ten koste van wat extra informatie in het record van elk element. Deze technieken worden gekarakteriseerd door de boom-typen die ze voortbrengen. We noemen AVL-bomen en B-bomen. Om de boom bij toevoeging van elementen in balans te houden zal hij af en toe geherstructureerd moeten worden. Eventueel kan de boom ook geherstructureerd worden bij inspectie van elementen, opdat veel gevraagde elementen snel gevonden zullen worden, en de boom zich aanpast wanneer de belangstelling voor de diverse elementen zich wijzigt. Een nadeel van het herstructureren is, dat extra accesses naar het achtergrondgeheugen noodzakelijk worden. Dit zou de netto winst van het balanceren van de boom teniet kunnen doen. Een ander punt is welke graad we voor de boom kiezen. Een hoge graad betekent dat de records van de knopen in de boom groot zullen zijn, terwijl een lage graad een grotere diepte in de boom oplevert. Bij gegeven grootte van de records waarin we de elementen van de verzameling representeren gaan we nu eerst de optimale graad van de boom bepalen, bij de gegeven karakteristieken van het achtergrondgeheugen. Laat daartoe S de tijd zijn die verloopt tussen het begin van de diskoperatie en het moment waarop de transfer begint. De parameter S zal gedeeltelijk van de toepassing afhangen. Zo kan men voor disk geheugens hierin opnemen of de verzameling aaneengesloten op de disk staat, en ook de te verwachten volgorde van disk accesses kan een rol spelen. Laat T de transfertijd zijn van een elementair element uit het record. We kiezen $\rho = \frac{S}{T}$ de zogenaamde seek-transfer ratio. Het zal nu een tijd $S+TR$ kosten om een record ter grootte R uit het achtergrondgeheugen te accessen (lezen of schrijven). Deze grootte R zal voor disks slechts een beperkt aantal waarden aan kunnen nemen, met de grootte van het deel van het centrale geheugen dat vrij is voor bufferruimte als bovengrens.

Wanneer we de elementen van de verzameling in records ter grootte v representeren, en de graad van de boom $m (\geq 2)$ bedraagt, zal de tijd om een bepaald record uit de verzameling te accessen $O((S+Tmv)^m \log n) = \log n \cdot O(\frac{\rho + mv}{\log m})$ bedragen. De optimale waarde van de graad m van de boom bij gegeven ρ en v wordt dus gevonden als oplossing van $v \cdot \log m = (\rho + mv) \cdot \frac{1}{m}$, met inachtname van de beschreven restrictie aan de grootte van records in het achtergrond geheugen. Meestal zal het lonend zijn op het achtergrondgeheugen een speciale indexfile bij te houden, waarmee men de locatie in het achtergrondgeheugen van het betreffende element uit de verzameling eerst bepaalt, de indexfile zal dan een veel kleinere recordgrootte hebben.

Device	Example	Average seek time (ms)	Transfer time (us/bit)	Seek-transfer ratio	Optimal order of tree		
					record size = 8	record size = 80	record size = 400
Disk pack	IBM 3350	33.0	0.1	330	890	132	39
Cartridge disk	DEC RL 01	63.0	0.3	210	607	93	28
Drum	IBM 2301	8.6	0.1	86	288	47	15
Floppy disk	DEC RX 01	360.0	6.0	60	214	36	12
Bubble memory	TI TIB 0303	7.3	10.0	0.73	9	3	2
Bubble memory	TI TIB 0103	4.0	20.0	0.2	5	2	2
MCS RAM		0.0003	0.5	0.0006			

Nadat men de optimale graad van de boom gevonden heeft dient men het optimale boomtype te bepalen. Opgemerkt dient nog te worden dat, indien de optimale graad van de boom in de buurt van 2 ligt, het meestal toch beter is binaire bomen te gebruiken. Dit omdat binaire bomen eenvoudiger te doorzoeken en te updaten zijn, terwijl ze efficiënter te balanceren zijn. De meest bekende methode om multiway (graad > 2) bomen te implementeren is in de vorm van B-bomen [5]. In een niet lege B-boom van type $\tau (k, h)$ heeft elke knoop ($\neq$ wortel) tenminste $k+1$ zonen, maar niet meer dan $2k+1$. Elk pad van de wortel tot een blad heeft lengte k . Het is dus duidelijk dat niet elke knoop volledig gevuld zal zijn (voor grote waarden van k is het geheugen gebruik voor random B-bomen ongeveer 69% [6]. Toevoegen van elementen kan overflow in een knoop opleveren, waarna de knoop gesplitst wordt.

Bij verwijdering kan het omgekeerde gebeuren, en worden twee knopen verenigd. Op deze manier wordt in B-bomen de balans bewaard. B-bomen staan bekend als zeer efficiënt, ondanks het partiele geheugen gebruik, hetgeen de B-boom bovendien wat dieper maakt dan een optimale multiway boom van dezelfde graad $(2k+1)$.

We zullen ons nu verder bezig houden met de binaire bomen. We zullen allereerst rechtvaardigen dat het zin heeft om de boom gebalanceerd te houden, dat wil zeggen dat de door balancerings geïmpliceerde overhead in diskaccesses terugverdiend wordt aan de hierdoor korter geworden gemiddelde padlengte in de binaire boom. Hiertoe beschouwen we alleen de verzamelingsoperaties INSERT & FIND.

Als gebalanceerde bomen kiezen we de AVL bomen. De AVL boom is op hoogte gebalanceerd. Het verschil in hoogte tussen de linker en rechter-subboom van elke knoop in de AVL boom mag hooguit 1 bedragen. Om deze informatie lokaal in de knoop aanwezig te hebben worden balansfactoren bijgehouden. Wanneer een nieuwe knoop wordt toegevoegd, wordt de boom allereerst afgelopen tot de knoop gevonden is waaraan het nieuwe element gehangen wordt. Door de toevoeging kan de balans evenwel verstoord worden. Om deze te herstellen voeren we ter plaatse een rotatie uit. Op een slotstuk van het reeds afgelegde pad (vanaf de zogenaamde critical node) moeten we nu de balansfactoren gaan aanpassen. Dit betekent dat alle bijbehorende records geschreven moeten worden naar het achtergrondgeheugen, na eventueel eerst nogmaals ingelezen te zijn. Empirisch blijkt het gemiddeld aantal knopen op het kritische pad < 4 te zijn, zodat het meestal mogelijk zal zijn om opnieuw inlezen te vermijden.

Andere empirische gegevens voor de AVL bomen (zie [7]) zijn:

$$\begin{aligned} W &= 1.08 \log_2 n && \text{gemiddelde padlengte in AVL boom} \\ C &= 2.8 && \text{gemiddelde lengte kritische pad} \\ &&& \text{met kans } 0.46 \text{ is een herstructurering nodig na een insertie} \end{aligned}$$

Het gemiddeld aantal accesses naar het achtergrondgeheugen wordt dan:

$$A = 1.08 \log_2 n + p (2.8 + 0.46)$$

als p de kans op een toevoeging is. Indien het kritische pad opnieuw ingelezen moet worden dient de coëfficiënt van p verdubbeld te worden.

Vergelijken we dit met de bomen waarbij elementen toegevoegd worden zonder herstructuering toe te passen dan blijkt empirisch:

$$W = 1.40 \cdot 2^{\log n}$$

zodat $A = 1.40 \cdot 2^{\log n} + p.$

Het gemiddeld aantal accesses is derhalve kleiner voor de AVL boom als:

$$n > 2^{7.1 p}.$$

Is het opnieuw inlezen van het kritische pad wel nodig, dan is de AVL boom beter voor $n > 2^{14.1 p}.$

Voor de praktijk is het dus de moeite waard om de AVL boom te gebruiken. Dit is in contrast met de empirische ervaringen in een RAM omgeving (lettend op CPU tijd). De AVL boom is daar beduidend trager voor een grote reeks van datatypen!

Een andere manier om de boom in balans te houden is het gewicht van de beide subbomen van elke knoop in de boom niet te ver uiteen te laten lopen door de verhouding hiervan tussen α en $1-\alpha$ te laten voor zekere $0 \leq \alpha \leq \frac{1}{2}$. Hierdoor ontstaan de zogenaamde BB-bomen. De gewogen padlengte van BB-bomen blijkt vrijwel overeen te stemmen met die voor AVL-bomen. Daar staat tegenover dat BB-bomen gemiddeld een traceback path hebben met lengte in de orde $\log n$. Hierbij is in het algemeen naast beschrijven ook herlezen noodzakelijk. Verder bezoeken de BB algoritme's ook knopen die niet op het zoekpad liggen. Met andere woorden, AVL bomen zullen beter werken dan BB-bomen.

Wat betreft self-organizing technieken om de boom te balanceren, volstaan we met de opmerking dat AVL-bomen een betere implementatie zijn (zie [7]).

3. REFERENTIES

- [1] MUNRO, J.I. & H. SUWANDA, *Implicit Data Structures*, STOC 11, 1979 pp. 108-117.
- [2] MUNRO, J.I. & M.S. PATERSON, *Selection and sorting with limited storage*, Theoretical Computer Science 12, 1980, pp. 315-323.

- [3] AHO, A.V., J.E. HOPCROFT & J.D. ULLMAN, *The design and analysis of computer algorithms*, Addison-Wesley, Reading. Mass., 1974.
- [4] DOBKIN, D. & J.I. MUNRO, *A minimal space selection algorithm that runs in linear time*, Yale technical report, 1977.
- [5] BAYER, R. & E. MCCREIGHT, *Organization and maintenance of large ordered indexes*, Acta Informatica 1, 1972, pp. 173-189.
- [6] YAO, A.C., *On random 2-3 trees*, Acta Informatica 9, 1978, pp. 159-170.
- [7] WRIGHT, W.E., *Binary search trees in secondary memory*, Acta Informatica 15, 1981, pp. 3-17.

DYNAMISCHE ZOEKSTRUCTUREN DIE HUN GESCHIEDENIS ONTHOUDEN^{*)}

M.H. OVERMARS

1. ZOEKPROBLEMEN

Veel problemen waar de analyse van algoritmen zich mee bezig houdt, kunnen gezien worden als zoekproblemen. Heel in het algemeen is een zoekprobleem een probleem waarin we een vraag stellen over een zoekobject ten opzichte van een verzameling objecten (die meestal punten worden genoemd). Wat formeler is het een afbeelding die een object van type T_1 en een verzameling punten van type T_2 afbeeldt op een object (antwoord) van type T_3 . Zo'n zoekprobleem Q kunnen we dus noteren als

$$Q: T_1 \times T_2 \rightarrow T_3.$$

Om gevoel te krijgen voor het soort problemen dat we nu precies bedoelen zullen we eerst enige voorbeelden behandelen.

a) Member searching

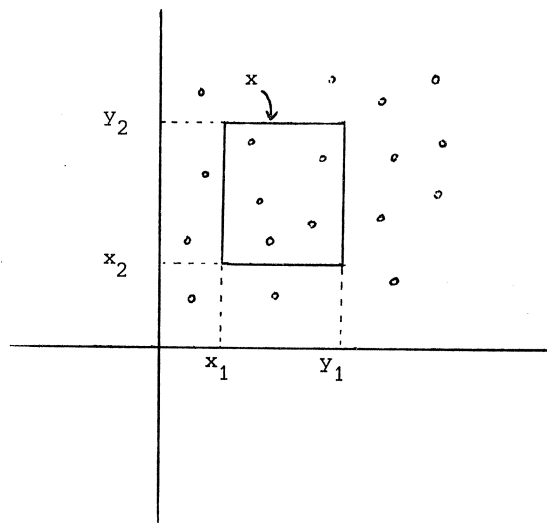
Het meest eenvoudige voorbeeld van een zoekprobleem is de zogenaamde "member searching". Hierbij hebben we een verzameling punten V en een ander punt x en vragen of $x \in V$. In dit geval geldt dus $T_1 = T_2$ en $T_3 = \text{boolean}$.

b) Range searching

Een belangrijk zoekprobleem, met veel toepassingen in bijvoorbeeld databases, is de zogenaamde "range search". Hierbij bestaat de verzameling V

^{*)} Dit werk werd gesubsidieerd door de Nederlandse Organisatie voor Zuiver-Wetenschappelijk Onderzoek (ZWO).

uit punten $p = (p_1, \dots, p_d)$ in een d -dimensionale ruimte. Het zoekobject x bestaat uit een "range" $([x_1:y_1], [x_2:y_2], \dots, [x_d:y_d])$ en we vragen naar alle punten $p \in V$ met $x_1 \leq p_1 \leq y_1, x_2 \leq p_2 \leq y_2, \dots, x_d \leq p_d \leq y_d$. Geometrisch gezien vragen we dus naar alle punten p die in een orthogonale d -dimensionale rechthoek (= rechthoek met zijden evenwijdig aan de coördinaatassen) liggen. Zie Figuur 1 voor een 2-dimensionaal voorbeeld.

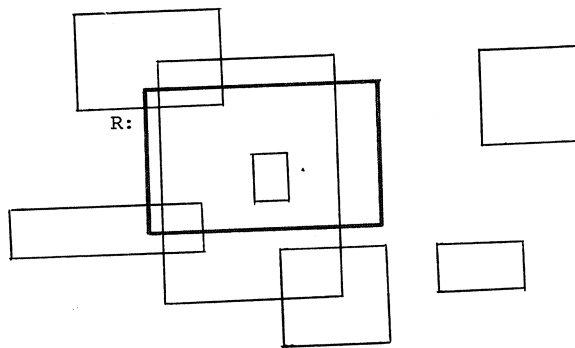


Figuur 1.

Een belangrijk toepassingsgebied van dit probleem is de database. Als we bijvoorbeeld in een database van een groep mensen leeftijd en salaris hebben opgeslagen, willen we vragen kunnen stellen als: welke mensen met leeftijd tussen 20 en 25 hebben een salaris tussen f 30.000,- en f 40.000,- per jaar? Als we nu leeftijd als eerste coördinaat zien en salaris als tweede coördinaat, dan vragen we dus naar die punten die liggen in de range $([20:25], [30.000:40.000])$.

c) Rechthoek intersectie

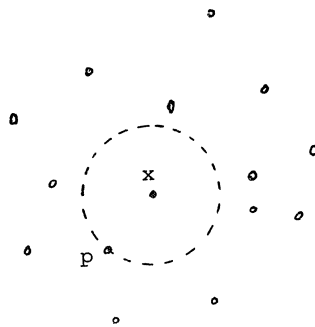
Het rechthoek intersectieprobleem is het volgende: gegeven een verzameling V van orthogonale hyper-rechthoeken in d -dimensionale ruimte en een andere hyper-rechthoek R ; bepaal alle rechthoeken in V die R doorsnijden (= die tenminste één punt met R gemeen hebben). Zie Figuur 2.



Figuur 2

d) Naaste buur zoeken

Het "naaste buur" zoekprobleem is het volgende: gegeven een verzameling punten V in een d -dimensionale ruimte en een ander punt x ; bepaal een punt p in V met kleinste afstand tot x met betrekking tot een zekere metriek (zie Figuur 3).

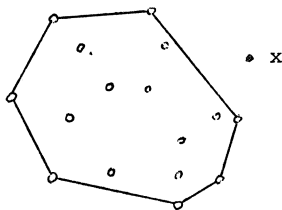


Figuur 3

e) Convex omhulsel zoeken

Gegeven een verzameling punten V in een d -dimensionale ruimte en een punt x . Het "convexe omhulsel" zoekprobleem vraagt of x binnen of buiten

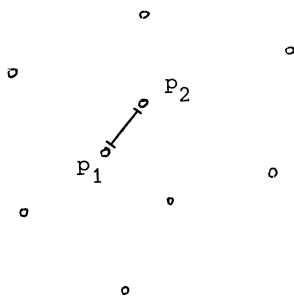
(of op) het convexe omhulsel van V ligt. (Het convexe omhulsel van een verzameling punten V is het kleinste convexe figuur dat alle punten in V bevat.) Zie Figuur 4 voor een 2-dimensionaal voorbeeld.



Figuur 4

f) Closest pair

Niet alle zoekproblemen hebben een werkelijk zoekobject. In een aantal gevallen wordt er alleen een vraag over de verzameling gesteld. Een voorbeeld hiervan is het "closest pair" probleem dat vraagt om twee punten in een d -dimensionale puntverzameling met minimale afstand (zie Figuur 5).



Figuur 5

De hierboven beschreven voorbeelden vormen slechts een kleine greep uit de vele bekende zoekproblemen.

Een zoekprobleem oplossen bestaat uit het ontwikkelen van een data-structuur om de verzameling punten op te slaan, zodanig dat vragen snel beantwoord kunnen worden. Deze zoekstructuren zijn of statisch of dynamisch. Een statische structuur bevat een bepaalde verzameling punten,

zodanig dat vragen snel beantwoord kunnen worden. Een dynamische zoekstructuur laat bovendien toe dat punten aan de verzameling (en dus aan de structuur) worden toegevoegd of uit de verzameling worden weggelaten. In Sectie 2 zullen we enige statische zoekstructuren voor de gegeven voorbeelden bespreken. In Sectie 3 zullen we ons concentreren op dynamische structuren. Als belangrijkste resultaat zullen we laten zien dat voor een grote klasse van zoekproblemen, statische zoekstructuren getransformeerd kunnen worden tot dynamische zoekstructuren. Gegeven een dynamisch veranderende verzameling V , kan het van belang zijn vragen te stellen over V op momenten in het verleden. Als we bijvoorbeeld een database hebben, kan het van belang zijn te weten of meneer Jansen drie jaar geleden meer dan 30.000 gulden verdiende. Alle bekende dynamische zoekstructuren zijn niet in staat dit soort informatie te leveren. In Sectie 4 zullen we laten zien hoe zoekstructuren zodanig getransformeerd kunnen worden dat zij hun geschiedenis onthouden, zodat we queries (vragen) over elk tijdstip (in het verleden) uit kunnen voeren.

2. STATISCHE ZOEKSTRUCTUREN

Gegeven een zoekprobleem Q . Een eerste probleem is het ontwerpen van een "efficiënte" statische zoekstructuur voor Q . De vraag wat efficiënt is hangt af van het gebruik van de structuur. De volgende maten zijn van belang.

DEFINITIE. Laat S een zoekstructuur zijn die n punten bevat.

$Q_S(n)$ = de zoektijd op S .

$P_S(n)$ = de bouwtijd van S .

$S_S(n)$ = de hoeveelheid benodigd geheugen voor S .

Laten we de voorbeelden van Sectie 1 nogmaals bekijken.

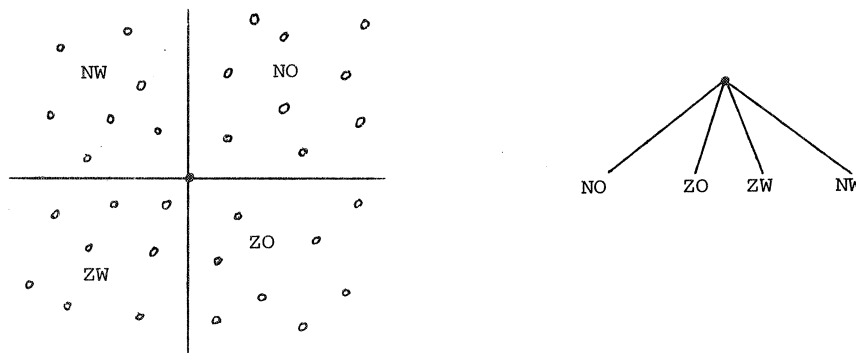
a) Member searching: Er bestaan vele zoekstructuren voor het "member search" probleem, zoals b.v. AVL-bomen (zie b.v. AHO, HOPCROFT & ULLMAN [1] of KNUTH [14]), B-bomen (zie b.v. COMER [6] en MEHLHORN [19]), of $BB[\alpha]$ -bomen. Al deze structuren bereiken de volgende grenzen:

$$Q_S(n) = O(\log n)$$

$$P_S(n) = O(n \log n)$$

$$S_S(n) = O(n).$$

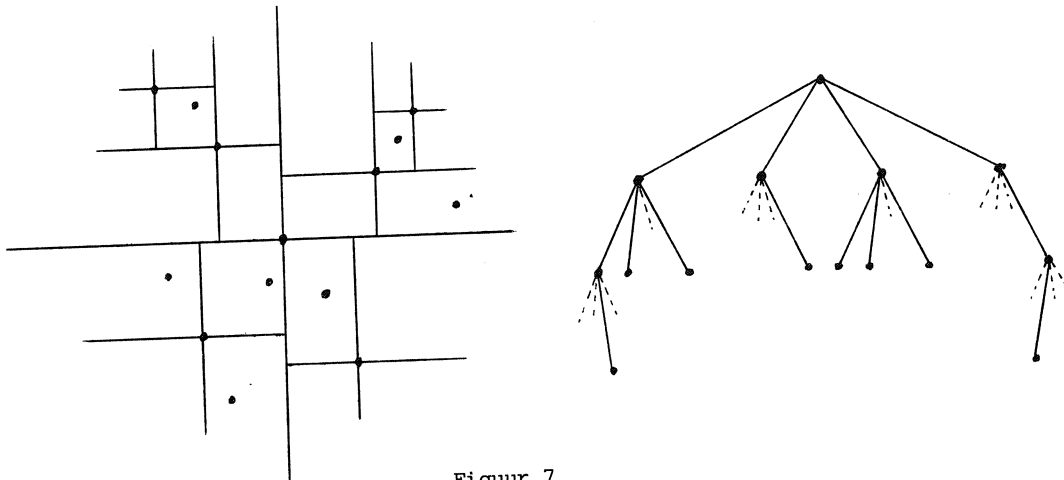
b) Range searching: Voor "range searching" waren tot voor kort geen zoekstructuren bekend met lage worst-case zoektijd en een redelijke bouwtijd. Wel waren er structuren ontworpen waarvoor de verwachte zoektijd laag was ($O(\log n)$). De belangrijkste hiervan was de zogenaamde quad-boom (zie FINKEL & BENTLEY [10] en BENTLEY & STANAT [5]). Een quad-boom wordt op de volgende wijze gebouwd. Eén van de punten van de verzameling wordt als wortel van de quad-boom genomen. Deze verdeelt het vlak in vier kwadranten NO, ZO, ZW en NW, en dus de verzameling in vier deelverzamelingen (zie Figuur 6). Deze vier deelverzamelingen vormen de zonen van de wortel. Elk



Figuur 6

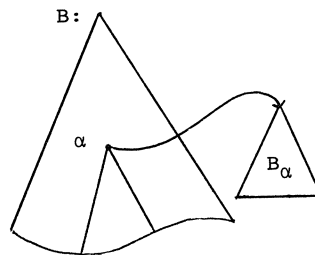
kwadrant wordt weer in vieren gesplitst door een punt van de verzameling te kiezen. Op deze wijze gaan we door tot elk deel-kwadrant ten hoogste één punt van de verzameling bevat. Zie Figuur 7 voor een voorbeeld van een quad-boom en de bijbehorende verdeling van het vlak in subkwadranten. Een quad-boom kan gebouwd worden in $P_S(n) = O(n \log n)$ en gebruikt $O(n)$ geheugen. Een andere structuur met een goede verwachte zoektijd is de k-d boom (zie BENTLEY [2] en SILVA-FILHO [36]). Ook deze is te bouwen in $P_S(n) = O(n \log n)$ en gebruikt $O(n)$ geheugen.

Kort geleden werden enige nieuwe datastructuren voor het d-dimensionale range searching probleem ontworpen (zie WILLARD [39,40] en LUEKER [16]). Deze structuren gebruiken meer geheugen en hebben een hogere bouw-
tijd dan de quad-boom en de k-d boom, maar hebben lage worst-case zoektijden.



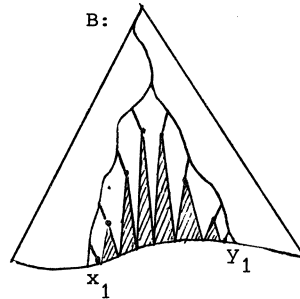
Figuur 7

We zullen hier alleen het 2-dimensionale geval behandelen. De basisstructuur die we gebruiken bestaat uit een binaire zoekboom B waarin we de punten uit de verzameling, geordend naar 1^e coördinaat, in de bladen opslaan. Nu voegen we aan elke interne knoop α een binaire zoekboom B_α toe waarin we de punten in de deelboom onder α , gesorteerd op 2^e coördinaat opslaan (zie Figuur 8).



Figuur 8

Als we nu een range query uit willen voeren met een range $([x_1:y_1], [x_2:y_2])$, zoeken we met zowel x_1 als y_1 in B . In het begin zullen x_1 en y_1 hetzelfde pad volgen, maar op een gegeven moment zullen ze verschillende richtingen uitgaan. We weten dat van nu af aan punten onder knopen α die rechts naast het zoekpad van x_1 liggen (i.e. de vader van α ligt op het zoekpad maar α zelf niet) of links naast het zoekpad van y_1 liggen, binnen de range $[x_1:y_1]$ vallen (zie Figuur 9). We voeren nu op B_α een zoekactie uit met x_2 en y_2 .



Figuur 9

De punten die hiertussen liggen, liggen dus in de volledige range. Men ziet eenvoudig in dat op deze wijze precies elk punt van de verzameling dat in de range ligt, één keer verkregen wordt. Het aantal structuren B_α dat doorzocht moet worden is begrensd door $2 \log n$ en de zoektijd op B_α is begrensd door $O(\log n)$ (+ het aantal antwoorden dat afgedrukt moet worden). De totale zoektijd is dus begrensd door $O(\log^2 n)$. De structuur kan in $O(n \log n)$ gebouwd worden en gebruikt $O(n \log n)$ geheugen.

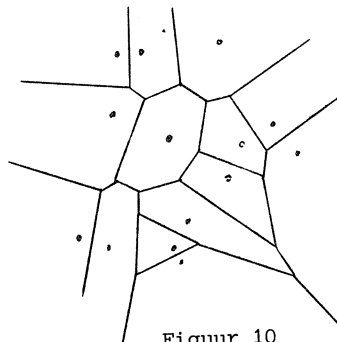
c) Rechthoek intersectie: Ook voor het rechthoek intersectie probleem waren tot voor kort geen goede zoekstructuren bekend. EDELSBRUNNER [8] (zie ook LEE & WONG [15]) toonde aan dat het d -dimensionale rechthoek intersectie probleem te transformeren is naar het $2d$ -dimensionale range searching probleem. Hierdoor wordt een oplossing verkregen met

$$\begin{aligned} Q_S(n) &= O(\log^{2d} n) \\ P_S(n) &= O(n \log^{2d-1} n) \\ S_S(n) &= O(n \log^{2d-1} n). \end{aligned}$$

Kort hierna toonden EDELSBRUNNER & MAURER [9] aan dat het probleem ook direct op te lossen is (met gebruikmaking van een zeer complexe zoekstructuur), zodanig dat

$$\begin{aligned} Q_S(n) &= O(\log^d n) \\ P_S(n) &= O(n \log^d n) \\ S_S(n) &= O(n \log^d n). \end{aligned}$$

d) Naaste buur zoeken: Om het naaste buur zoekprobleem in het vlak op te lossen wordt voor de verzameling V het zogenaamde Voronoi diagram gemaakt. Dit Voronoi diagram geeft bij elk punt $p \in V$ het gebied van punten die het dichtst bij p liggen (zie Figuur 10). Elk Voronoi-gebied is een convexe veelhoek. Om nu voor een gegeven

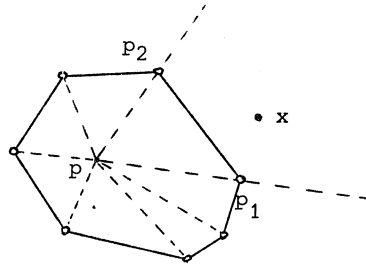


Figuur 10

punt x te bepalen welk element $p \in V$ het dichtst bij x ligt, bepalen we het gebied waar x in ligt. Hiermee is p bepaald. Het maken van het Voronoi diagram kost $O(n \log n)$ (zie SHAMOS [34], SHAMOS & HOEY [35]). Het bouwen van de zoekstructuur op het Voronoi diagram om het gebied te bepalen waar een punt x in ligt, kan ook in $O(n \log n)$ en het zoeken kost $O(\log n)$ (zie KIRKPATRICK [13]). Op deze wijze verkrijgen we een oplossing voor het 2-dimensionale naaste buur zoekprobleem, met

$$\begin{aligned} Q_S(n) &= O(\log n) \\ P_S(n) &= O(n \log n) \\ S_S(n) &= O(n). \end{aligned}$$

e) Convex omhulsel zoeken: Er zijn vele verschillende algoritmen ontwikkeld voor het bepalen van het convexe omhulsel van een verzameling punten in het vlak (zie b.v. GRAHAM [12], PREPARATA [30] en BENTLEY [4]). Het merendeel van deze algoritmen werkt in $O(n \log n)$. De punten op het convexe omhulsel kunnen gezien worden als zijnde geordend bij hoek rond een inwendig punt p . Door deze punten in deze volgorde op te slaan kunnen we met binair zoeken in $O(\log n)$ voor elk punt x , twee punten p_1 en p_2 op het convexe omhulsel bepalen, zodanig dat x vanuit p gezien tussen p_1 en p_2 ligt (zie Figuur 11). Hierna is met een eenvoudige test de positie van x ten opzichte van de lijn $\overline{p_1 p_2}$ te bepalen en hiermee is bepaald of x binnen of buiten het convexe omhulsel ligt.



Figuur 11

Ook in een 3-dimensionale ruimte is het convexe omhulsel van een verzameling punten in $O(n \log n)$ te bepalen (zie bijvoorbeeld PREPARATA & HONG [31]) en ook in dit geval kunnen we een zoekstructuur maken die in $O(\log n)$ bepaalt of een punt x binnen of buiten het convexe omhulsel ligt, zij het op een veel ingewikkelder manier. In hoger dimensionale ruimte zijn geen efficiënte convexe omhulsel algoritmen bekend.

f) Closest pair: Bij zoekproblemen zonder zoekobject bestaat het statisch oplossen van het probleem uit het geven van een algoritme dat bij een gegeven verzameling de juiste oplossing bepaalt. Het closest pair probleem kan met behulp van het Voronoi diagram opgelost worden in $O(n \log n)$ tijd (zie SHAMOS & HOEY [35]).

3. DYNAMISCHE ZOEKSTRUCTUREN

Vaak is het bij zoekproblemen van belang dat er punten aan de verzameling kunnen worden toegevoegd en/of uit de verzameling kunnen worden weggelaten. Indien het bij een zoekstructuur mogelijk is punten toe te voegen en/of weg te laten, noemen we deze structuur dynamisch. Het ontwerpen van een dynamische zoekstructuur voor een zoekprobleem is vaak moeilijker dan het ontwerpen van een statische zoekstructuur.

DEFINITIE. Laat S een zoekstructuur zijn die n punten bevat.

$I_S(n)$ = de tijd, nodig om een punt aan S toe te voegen.

$D_S(n)$ = de tijd, nodig om een punt uit S weg te laten.

In een aantal gevallen zullen dit geen worst-case, maar average (gemiddelde) maten zijn, d.w.z., de totale tijd, nodig voor een serie

updates (insertions of deletions) gedeeld door het aantal updates.
In dit geval gebruiken we de notatie $I_S^a(n)$ resp. $D_S^a(n)$.

Laten we opnieuw naar de voorbeelden kijken.

a) Member searching: Vrijwel alle zoekstructuren voor dit probleem zijn volledig dynamisch, met

$$I_S(n) = O(\log n)$$

$$D_S(n) = O(\log n).$$

b) Range searching: De quad-boom, die beschreven werd in Sectie 2, is niet dynamisch. Indien men bijvoorbeeld de wortel weg wil laten, zal er een nieuw punt van de verzameling als wortel gekozen moeten worden. Dit nieuwe punt splitst het vlak in andere kwadranten en dus de verzameling in andere deelverzamelingen. Dit betekent dat de quad-boom voor een groot stuk herbouwd zal moeten worden. (Zie SAMET [32] voor resultaten betreffende deletions in quad-bomen.) Als we de structuur van de quad-boom echter enigszins wijzigen is het wel mogelijk punten efficiënt toe te voegen en weg te laten. OVERMARS & VAN LEEUWEN [25] definiëren hiertoe een zogenaamde pseudo quad-boom. Deze verschilt hierin van de quad-boom dat interne knopen van de boom willekeurige punten zijn, in plaats van punten uit de verzameling. De punten van de verzameling komen alleen aan de bladen van de boom. In dit geval kan men punten zowel toevoegen als weglaten. Het enige probleem is dat de structuur niet uit balans mag raken. In [25] wordt aangetoond dat een pseudo quad-boom in balans gehouden kan worden in

$$I_S^a(n) = O(\log^2 n)$$

$$D_S^a(n) = O(\log^2 n).$$

Evenzo kan men k-d bomen transformeren in pseudo k-d bomen (zie [25]).

De structuur beschreven in LUEKER [16] en WILLARD [40] is dynamisch en bereikt (zij het op zeer ingewikkelde wijze)

$$Q_S(n) = O(\log^d n)$$

$$P_S(n) = O(n \log^{d-1} n)$$

$$S_S(n) = O(n \log^{d-1} n)$$

$$I_S(n) = O(\log^d n)$$

$$D_S(n) = O(\log^d n).$$

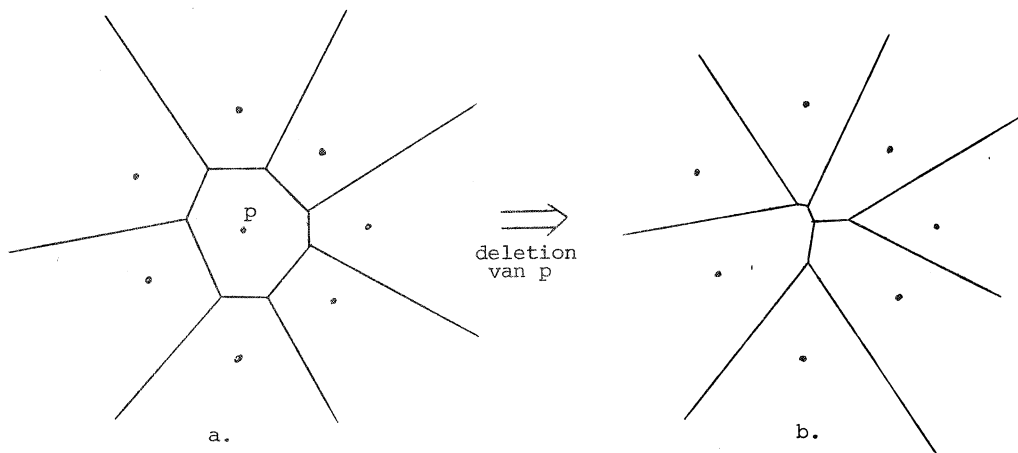
Met een techniek beschreven in OVERMARS & VAN LEEUWEN [29] is de deletion-tijd zelfs terug te brengen tot $O(\log^{d-1} n)$.

c) Rechthoek intersectie: De structuur van EDELSBRUNNER & MAURER [9] is dynamisch en bereikt

$$I_S(n) = O(\log^d n)$$

$$D_S(n) = O(\log^d n).$$

d) Naaste buur zoeken: Het dynamisch bijhouden van het Voronoi diagram van een verzameling punten is niet eenvoudig. Indien we bijvoorbeeld in Figuur 12a het punt p weglaten, verandert het Voronoi diagram volledig (zie Figuur 12b). OVERMARS [21] beschrijft een structuur om



Figuur 12

configuraties in het vlak (of meerdimensionale ruimte) dynamisch bij te houden. Gebruik makend van deze algemene methode verkrijgen we een dynamische datastructuur voor het naaste buur zoekprobleem met

$$Q_S(n) = O(\log n)$$

$$P_S(n) = O(n \log n)$$

$$S_S(n) = O(n \log n)$$

$$I_S(n) = O(n)$$

$$D_S(n) = O(n).$$

Het is mogelijk om de benodigde hoeveelheid geheugen nog te verlagen tot

$O(n \log \log n)$ (zie GOWDA & KIRKPATRICK [11]).

e) Convexe omhulsel zoeken: Ook voor het convexe omhulsel zoekprobleem kan de methode uit OVERMARS [21] gebruikt worden (zie ook OVERMARS & VAN LEEUWEN [26,27]). In het 2-dimensionale geval geeft dit een structuur met

$$\begin{aligned} Q_S(n) &= O(\log n) \\ P_S(n) &= O(n \log n) \\ S_S(n) &= O(n) \\ I_S(n) &= O(\log^2 n) \\ D_S(n) &= O(\log^2 n). \end{aligned}$$

In het 3-dimensionale geval krijgt men een structuur met

$$\begin{aligned} Q_S(n) &= O(\log n) \\ P_S(n) &= O(n \log n) \\ S_S(n) &= O(n \log n) \\ I_S(n) &= O(n) \\ D_S(n) &= O(n). \end{aligned}$$

f) Closest pair: Opnieuw kan de methode in [25] gebruikt worden om het closest pair van een verzameling punten bij te houden in $O(n)$ per update.

Uit bovenstaande voorbeelden moge blijken dat voor een aantal zoekproblemen geen echt efficiënte dynamische zoekstructuren bekend zijn. Voor een aantal andere zoekproblemen zijn deze structuren zeer complex. We zullen nu laten zien dat voor een grote klasse van zoekproblemen statische zoekstructuren op eenvoudige wijze te transformeren zijn in dynamische zoekstructuren.

DEFINITIE. Een zoekprobleem Q heet *ontbindbaar* (decomposable) als voor elke partitie $A \cup B = V$ en elk zoekobject x

$$Q(x, V) = \square(Q(x, A), Q(x, B)),$$

waarbij $\square$ in $O(1)$ te berekenen is.

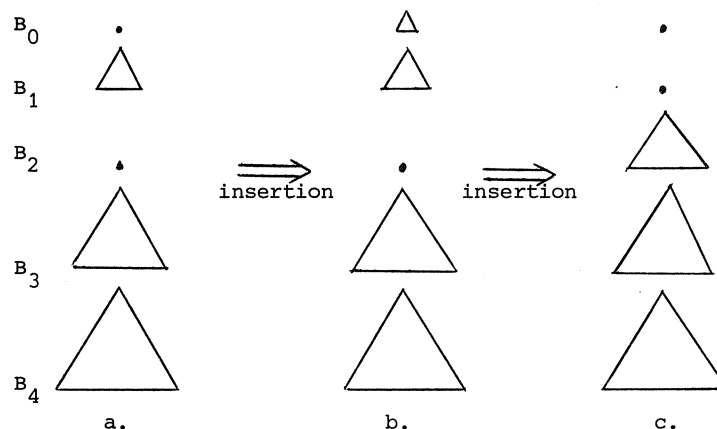
BENTLEY [3] was de eerste die opmerkte dat veel zoekproblemen deze eigenschap bezitten. Voor ontbindbare zoekproblemen hoeft men de verzameling

V niet in één structuur op te slaan, maar kan men in plaats daarvan de verzameling splitsen in disjuncte deelverzamelingen en voor elk van deze deelverzamelingen een zoekstructuur maken. Het antwoord op een query over de totale verzameling kan nu, met behulp van $\square$, verkregen worden uit de antwoorden op de query over de deelverzamelingen.

VOORBEELDEN.

- a) *Member searching* is decomposable met $\square = \vee$ (of).
- b) *Range searching* is decomposable met $\square = \cup$ (vereniging).
- c) *Rechthoek intersectie* is decomposable met $\square = \cap$.
- d) *Naaste buur zoeken* is decomposable met $\square = \min$.
- e) *Convexe omhulsel zoeken* is niet decomposable.
- f) *Closest pair* is niet decomposable.

BENTLEY [3] (zie ook SAXE & BENTLEY [33]) liet zien dat statische zoekstructuren voor ontbindbare zoekproblemen te transformeren zijn naar structuren waarin men ook punten kan toevoegen zonder veel verlies in efficiëntie. Hiertoe wordt de verzameling V gesplitst in deelverzamelingen $V_0, V_1, \dots$, waarbij elke V_i of leeg is, of 2^i punten bevat. Als bijvoorbeeld $|V| = 26$ zijn V_1, V_3 en V_4 gevuld. Voor elke gevulde verzameling V_i bouwen we een statische structuur (blok) B_i (zie Figuur 13a). Als we nu een punt p aan V willen toevoegen, hoeven we alleen een blok B_0 van p te bouwen (zie Figuur 13b). Om opnieuw een punt p' toe te voegen moeten we B_0 en B_1 afbreken en uit p' en de punten uit B_0 en B_1 een blok B_2 bouwen. In het algemeen gaan we, als we een punt p willen toevoegen, als volgt te werk:



Figuur 13

- a) bepaal de kleinste i met B_i leeg;
 b) breek $B_0 \dots B_{i-1}$ af;
 c) bouw B_i uit de punten uit $B_0 \dots B_{i-1}$ en het nieuwe punt p .
 Soms is het mogelijk dat een insertion erg veel werk kost (als er een groot blok gebouwd moet worden) maar vaak kost een insertion erg weinig moeite. Daardoor wordt de gemiddelde insertientijd laag.

STELLING. Voor een ontbindbaar zoekprobleem met statische zoekstructuur S bestaat er een structuur D met

$$Q_D(n) = O(\log n) \cdot Q_S(n).$$

$$I_D^a(n) = O(\log n) \cdot P_S(n)/n.$$

BEWIJS. We gebruiken als structuur D de hierboven beschreven structuur. Laten we eerst de zoektijd bekijken. Aangezien een gevulde B_i 2^i punten bevat, is, als de totale structuur n punten bevat, het grootste gevulde blok $B_{\lfloor \log n \rfloor}$. Aangezien elk blok $\leq n$ punten bevat, kost een query over een blok hoogstens $Q_S(n)$ (aannemend dat Q_S niet dalend is). Er zijn ten hoogste $\lfloor \log n \rfloor + 1$ gevulde blokken, dus de queries over de blokken kosten samen ten hoogste $O(\log n) Q_S(n)$. Het samenstellen van de antwoorden met gebruikmaking van $\square$ kost ten hoogste $O(\log n)$.

Als we een blok bouwen komen al de punten die erin komen uit "lagere" blokken. Het volgt dat elk punt slechts ten hoogste één keer in elk blok wordt gebouwd, dus in ten hoogste $\lfloor \log n \rfloor + 1$ blokken. Als we de kosten voor het bouwen van een blok B_i verdelen over de punten waaruit het blok gebouwd is wordt dit $P_S(2^i) / 2^i \leq P_S(n)/n$ per punt (aannemend dat $P_S(n) = \Omega(n)$). De totale tijd per punt is dus $O(\log n) P_S(n)/n$. $\square$

Ten koste van een factor $O(\log n)$ in zoektijd kunnen we dus punten toevoegen in een redelijke gemiddelde tijd.

VOORBEELD. d. Er bestaat een structuur D voor het naaste buur zoekprobleem, met

$$Q_D(n) = O(\log^2 n)$$

$$I_D^a(n) = O(\log^2 n).$$

De hierboven beschreven structuur laat alleen insertions toe. Deletions leveren grote problemen op. Bijvoorbeeld, na 2^i insertions bestaat de structuur uit één blok B_i . Als we nu een punt weglaten moeten we

dat blok splitsen in $B_0, \dots, B_{i-1}$. Voegen we nu weer een punt toe, dan moeten we opnieuw één blok B_i bouwen uit de andere blokken en het nieuwe punt, enz. Dit kost zeer veel tijd. Het is wel mogelijk voor bepaalde deelklassen van ontbindbare zoekproblemen deletion-technieken te geven (zie SAXE & BENTLEY [33], OVERMARS & VAN LEEUWEN [23] en VAN LEEUWEN & MAURER [37]), maar daar zullen we hier verder niet op ingaan.

Een tweede methode voor het transformeren van statische zoekstructuren naar dynamische zoekstructuren is de zogenaamde "equal block" methode (zie MAURER & OTTMANN [17], VAN LEEUWEN & WOOD [38] en VAN LEEUWEN & MAURER [37]). Hierbij wordt de verzameling V gesplitst in deelverzamelingen $V_1, \dots, V_i$ van ongeveer gelijke grootte. Het antwoord op een query bepalen we weer door de query op elk van de deelverzamelingen uit te voeren. Als we een punt willen invoegen doen we dit door de kleinste structuur te herbouwen met het nieuwe punt erbij in. Om een punt weg te laten bepalen we eerst de structuur waar het punt in zit, door middel van een simpele zoekboom waarin we deze informatie bij elk punt opslaan, en herbouwen deze structuur zonder het weggelaten punt. We willen graag dat i een functie van n is en dat de structuren inderdaad ongeveer even groot blijven. Daartoe leggen we de volgende eisen op.

a) Er is een monotoon stijgende functie f met $0 \leq f(n) \leq n$, zodat $f(\frac{1}{2}n) \leq i \leq f(2n)$.

b) Voor elke V_j ($1 \leq j \leq i$) geldt: $|V_j| \leq 2n/i$.

Eis a) garandeert dat i ongeveer een functie van n is en eis b) stelt dat geen enkel blok te groot mag worden. Zodra als gevolg van een update aan een van beide eisen niet meer wordt voldaan, herbouwen we de volledige dynamische structuur met als nieuwe $i = f(n)$ en $|V_j| = n/i$ voor elke $1 \leq j \leq i$. Dit kost veel tijd ($O(P_S(n))$), maar we zullen zien dat, vanuit een net gebouwde structuur, het lang duurt (veel updates vergt) voordat aan een van beide eisen niet meer voldaan wordt. Stel dat, op het moment dat we de structuur bouwden, er n_0 punten in V waren en nu zijn er n punten in V en aan een van beide eisen wordt niet meer voldaan. Er zijn de volgende mogelijkheden:

- a¹) $i < f(\frac{1}{2}n)$. We weten $i = f(n_0)$, dus $f(\frac{1}{2}n) > f(n_0)$. Omdat f monotoon stijgend is, geldt dus $\frac{1}{2}n > n_0$, dus $n > 2n_0$. Er zijn dus minstens $\frac{1}{2}n$ insertions geweest sinds de (totale) structuur werd gebouwd.
- a²) $i > f(2n)$. Dan $f(n_0) > f(2n)$ en omdat f stijgend is $n_0 > 2n \Rightarrow \frac{1}{2}n_0 > n$. Er zijn dus minstens n deletions geweest.

- b) $|V_j| > 2n/i$ voor een of andere $1 \leq j \leq i$. Aangezien insertions altijd in het kleinste blok worden uitgevoerd en dit blok dus $\leq n/i$ is, kan een blok V_j alleen dan groot worden als er veel deletions in andere blokken plaatsvinden. Stel dat er de laatste keer dat er een insertion in V_j werd uitgevoerd n' punten in de verzameling aanwezig waren ($n' = n_0$ indien er geen insertions in V_j plaatsvonden). Dan geldt dus $|V_j| \leq n'/i$. We moeten herbouwen omdat $|V_j| > 2n/i$, dus $n < \frac{1}{2}n'$. Er zijn dus minstens $\frac{1}{2}n' > n$ deletions geweest.

Er zijn dus altijd tenminste $\frac{1}{2}n$ insertions of deletions geweest sinds de vorige keer dat de totale structuur gebouwd werd. Verdelen we de $P_S(n)$ bouwkosten over deze updates, dan is dit $O(P_S(n)/n)$ per update.

STELLING. Voor elk ontbindbaar zoekprobleem Q met statische structuur S en elke "nette" stijgende functie f (i.e. een functie f met $f(O(n)) = O(f(n))$), bestaat er een dynamische structuur D met

$$Q_D(n) = O(f(n)) Q_S(n/f(n))$$

$$I_D^a(n) = O(P_S(n/f(n)) + P_S(n)/n)$$

$$D_D^a(n) = O(P_S(n/f(n)) + P_S(n)/n + \log n).$$

BEWIJS. Aangezien het aantal structuren, i , altijd tussen $f(\frac{1}{2}n)$ en $f(2n)$ in ligt en f "net" is, geldt dat $i = O(f(n))$. Elke structuur bevat maximaal $2n/i$ punten, dus de querytijd per structuur is $Q_S(2n/i) = O(Q_S(n/f(n)))$. Dus de totale querytijd is $O(f(n))Q_S(n/f(n))$.

Zowel een insertion als een deletion bestaat uit het herbouwen van één statische structuur. Deze bevat hooguit $2n/i$ punten. De kosten zijn dus $P_S(2n/i) = O(P_S(n/f(n)))$. Bovendien wordt gemiddeld $P_S(n)/n$ per update besteed aan het herbouwen van de totale dynamische structuur. Daarnaast moeten we bij een deletion nog de structuur opzoeken waar het punt in zit. Dit kost $O(\log n)$. $\square$

VOORBEELD. d. Er bestaat een structuur D voor naaste buur zoeken met (neem $f(n) = \sqrt{n}$):

$$Q_D(n) = O(\sqrt{n} \log n)$$

$$I_D^a(n) = O(\sqrt{n} \log n)$$

$$D_D^a(n) = O(\sqrt{n} \log n).$$

Sinds de presentatie van bovenstaande twee methoden voor het dynamiseren

van ontbindbare zoekproblemen, zijn vele uitbreidingen voorgesteld. Zoals reeds genoemd, werden deletion methoden ontwikkeld voor de eerste methode. Bovendien werden verschillende trade-offs tussen query- en insertiontijd aangetoond (i.e. mogelijkheden om een betere insertiontijd te krijgen ten koste van een slechtere querytijd of omgekeerd) (zie OVERMARS & VAN LEEUWEN [24] en MEHLHORN & OVERMARS [20]). Ook werden er ondergrenzen voor de efficiëntie van de transformaties bewezen (zie MEHLHORN [18]). Een belangrijk resultaat is dat alle average grenzen ook als worst-case grenzen verkregen kunnen worden (zie OVERMARS & VAN LEEUWEN [28]). Het meest algemene - en in feite optimale - resultaat wordt verkregen in [29] door beide dynamisatiemethoden door elkaar te mengen.

STELLING. *Gegeven een ontbindbaar zoekprobleem Q met een statische structuur S en twee "nette" functies f en g ; dan bestaat er een dynamische structuur D voor Q met*

$$\begin{aligned} Q_D(n) &= O(f(n) + g(n))Q_S(n), \\ I_D(n) &= \begin{cases} O(\log n / \log \frac{f(n)}{\log n}) P_S(n)/n & \text{als } f(n) = \Omega(\log n), \\ O(f(n)n^{(1/f(n))}) P_S(n)/n & \text{als } f(n) = O(\log n), \end{cases} \\ D_D(n) &= O(D_S(n/g(n)) + \log n + P_S(n)/n), \end{aligned}$$

waarin $D_S(n)$ de tijd is, nodig om een deletion op S uit te voeren (dus in sommige gevallen $D_S(n) = P_S(n)$).

VOORBEELD. d. Gebruikmakend van de structuur voor naaste buur zoeken in [21] (met $D_S(n) = O(n)$) kunnen we een dynamische structuur D maken voor naaste buur zoeken (neem $f(n) = g(n) = \sqrt{n/\log n}$) met

$$\begin{aligned} Q_D(n) &= O(\sqrt{n \log n}) \\ I_D(n) &= O(\log n) \\ D_D(n) &= O(\sqrt{n \log n}). \end{aligned}$$

4. ZOEKEN IN HET VERLEDEN

Normaal gesproken houden we in een dynamische zoekstructuur de informatie "up to date". Het gevolg is dat we altijd alleen maar vragen kunnen stellen over de situatie op het moment. Toch kan het in een aantal gevallen

van belang zijn om vragen te kunnen stellen over hoe de situatie op een bepaald tijdstip in het verleden was. Een eerste stap in de richting van zoekstructuren die hun geschiedenis onthouden werd recent gemaakt door DOBKIN & MUNRO [7] die zich echter alleen met een specifieke toepassing bezig hielden. Meer algemene resultaten worden gegeven in OVERMARS [22], waarvan we er hier enige zullen behandelen.

DEFINITIE. T_i = het moment waarop de i -de update op een structuur werd gedaan.

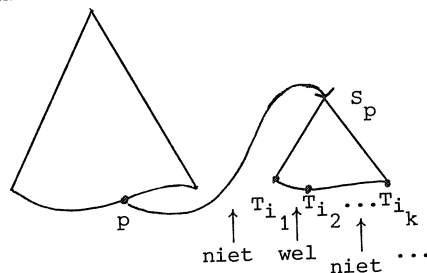
In werkelijkheid bestaat er natuurlijk niet zo'n moment, omdat een update tijd kost, maar we kunnen bijvoorbeeld het moment nemen waarop met de update begonnen wordt, of het moment waarop hij klaar is. Als we dus een query uit willen voeren op tijdstip T en

$$T_i \leq T < T_{i+1}$$

willen we een query uitvoeren op de structuur na de i -de update. N geeft vanaf nu altijd het aantal updates dat er tot nu toe geweest is aan. We zullen eerst een simpel voorbeeld behandelen.

VOORBEELD. a. Member searching.

Om member searching in het verleden uit te kunnen voeren, slaan we alle objecten die eens in de verzameling hebben gezeten op in een zoekboom. Dit zijn maximaal N punten. We kunnen nu dus in $O(\log N)$ bepalen of het punt dat we zoeken ooit in de verzameling heeft gezeten. Als dit niet zo was zijn we klaar. In het andere geval willen we graag weten of het punt op tijdstip T aanwezig was. Hiertoe slaan we bij het punt p de tijdstippen waarop p werd ingevoegd en weggelaten op in een zoekboom S_p en geven we bij elk tussenliggend interval aan of p daar wel of niet aanwezig was (zie Figuur 14). (We gaan ervan uit dat p alleen dan ingevoegd wordt als hij niet aanwezig is, dus p komt altijd hoogstens één keer voor.)



Figuur 14

Als we p gevonden hebben kunnen we dus met T in S_p zoeken en vinden we zo (in maximaal $O(\log N)$) of p op tijdstip T aanwezig was. Om een punt p (op tijdstip T_N) in te voegen, bepalen we eerst of p reeds ooit aanwezig geweest is). In dat geval voegen we T_N in in S_p (met de vermelding dat p nu weer aanwezig is). Als p nog nooit aanwezig geweest is, voegen we p toe in de hoofdstructuur en bouwen een S_p die alleen uit T_N bestaat. Al deze operaties kosten maximaal $O(\log N)$: Om een punt p weg te laten, zoeken we p in de hoofdstructuur en vervolgens voegen we T_N in in S_p (met de vermelding dat p nu niet meer aanwezig is). Ook dit kost maximaal $(\log N)$. We hebben dus een structuur voor member searching in het verleden met

$$Q(N) = O(\log N)$$

$$I(N) = O(\log N)$$

$$D(N) = O(\log N).$$

Voor member searching is dus tamelijk eenvoudig een structuur te maken voor zoeken in het verleden. We zullen nu een algemene methode bespreken die toepasbaar is voor zoekproblemen, waarvoor een dynamische zoekstructuur bestaat.

Het idee achter de algemene methode is tamelijk eenvoudig, namelijk het volgende: Van tijd tot tijd bouwen we een structuur die de situatie op dat moment geeft en bij alle andere updates noteren we alleen wat er gedaan moet worden. Als we nu op een gegeven moment in de tijd willen zoeken, nemen we de dichtsbij zijnde structuur, voeren de noodzakelijke updates uit en vervolgens de query, waarna we de structuur weer in zijn oorspronkelijke vorm herstellen. Wat formeler: Laat f een "nette" integer functie zijn. De momenten waarop we een volledige structuur bouwen zijn $T_{i_1}, T_{i_2}, \dots, T_{i_k}$, waarbij $i_1 = 1$ en $i_{k+1} = i_k + f(i_k)$. Als we nu op moment T met $T_j \leq T < T_{j+1}$ willen zoeken, bepalen we de grootste k met $i_k \leq j$. Daarna voeren we alle updates tussen T_{i_k} en T_j uit op de structuur (bij T_{i_k}). Dit zijn maximaal $f(i_k) \leq f(N)$ updates. Hierna voeren we een query uit op de structuur, die nu de situatie op moment T geeft en daarna herstellen we de structuur weer. Het volgt dat de zoektijd begrensd is door $O(f(N)U(N) + Q(N))$, waarbij $U(N) = \max(I(N), D(N))$. Het uitvoeren van een update kost soms veel tijd (als $T_N = T_{i_k}$) maar kost meestal slechts $O(1)$. Het is eenvoudig in te zien dat, als we de kosten voor het herbouwen gelijkmatig verdelen over de voorafgegane goedkope updates we een gemiddelde updatetijd krijgen van $P(N)/f(N)$. We hebben getoond:

STELLING. Gegeven een zoekprobleem Q met dynamische structuur D , dan bestaat er voor elke "nette" f een structuur D_1 om Q in het verleden op te lossen met

$$Q_{D_1}(N) = O(f(N)U_D(N) + Q_D(N))$$

$$I_{D_1}^a(N) = O(P_D(N)/f(N))$$

$$D_{D_1}^a(N) = O(P_D(N)/f(N)).$$

VOORBEELD. b. Er bestaat een structuur voor d -dimensionale range queries in het verleden met (neem $f(N) = \sqrt{N/\log N}$)

$$Q(N) = O(\sqrt{N \log N \log^{d-1} N})$$

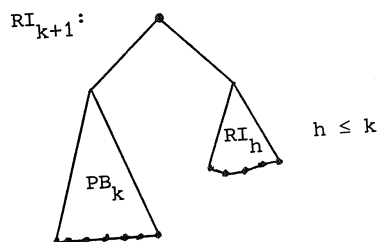
$$I^a(N) = O(\sqrt{N \log N \log^{d-1} N})$$

$$D^a(N) = O(\sqrt{N \log N \log^{d-1} N}).$$

Het is mogelijk de average grenzen te veranderen in worst-case grenzen (zie [22,28]).

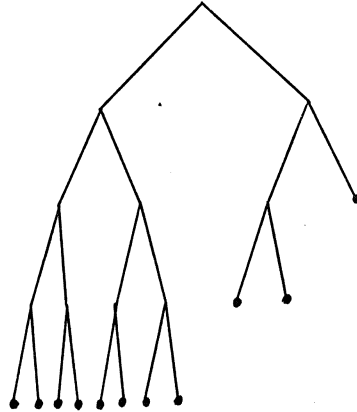
Ook voor ontbindbare zoekproblemen is het mogelijk een algemene transformatie van zoekstructuren te geven. We zullen alleen het geval behandelen waarin alleen insertions optreden. Als basisstructuur gebruiken we een boom waarin alle punten in de verzameling zitten opgeslagen, geordend naar moment van invoeging. In deze structuur hoeven we dus alleen aan de rechterkant te kunnen invoegen.

DEFINITIE. Een RI-boom (Right Insertion) van diepte 0 bestaat uit één punt. Een RI-boom van diepte $k+1$ bestaat uit een wortel, met als linker deel-boom een perfecte binaire boom van diepte k (PB_k) en als rechter deelboom een RI-boom van diepte $\leq k$. Punten zijn opgeslagen in de bladen (zie Figuur 15).



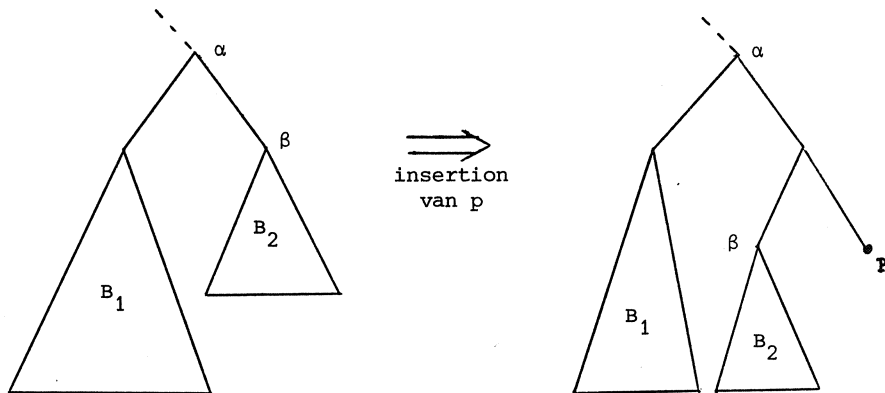
Figuur 15

Figuur 16 geeft een voorbeeld van een RI-boom van diepte 4, die 11 punten bevat. Men ziet eenvoudig in dat er voor elk aantal punten precies één RI-boom is die zoveel punten bevat.



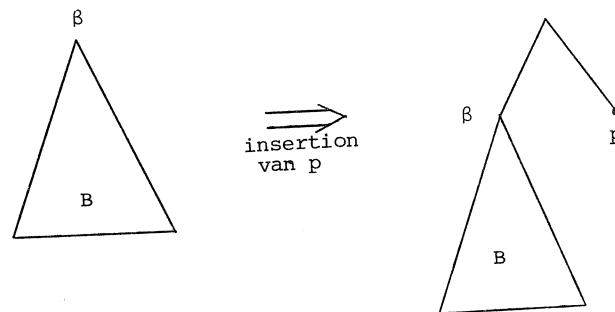
Figuur 16

Om een nieuw punt p aan de rechter kant van de boom in te voegen gaan we als volgt te werk: Bepaal eerst de dieptste knoop α op het meest rechtse pad met diepte linker deelboom $>$ diepte rechter deelboom. Noem de linker deelboom B_1 en de rechter deelboom B_2 . Zowel B_1 als B_2 zijn nu perfect binair en dus kunnen we de actie in Figuur 17 uitvoeren. Als zo'n knoop α



Figuur 17

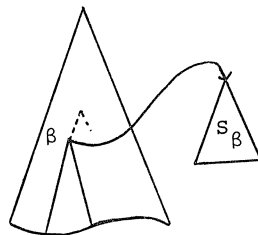
niet bestaat, is de hele boom perfect en moeten we de actie in Figuur 18 uitvoeren.



Figuur 18

Nadat α bekend is kost dit slechts $O(1)$. Ook het vinden van de juiste knoop α kan in $O(1)$ geschieden (zie [22]).

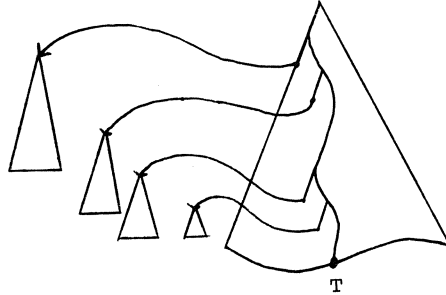
Als we nu een ontbindbaar zoekprobleem, met statische structuur S , in het verleden willen gaan oplossen, slaan wij alle punten in de verzameling op in een RI-boom, geordend naar tijdstip van insertion. In de interne knopen van de boom kan dus als zoekcriterium de tijd worden genomen. Als we nu met een bepaald tijdstip T de boom doorlopen, weten we dat alle punten in de deelboom links van het zoekpad op tijdstip T aanwezig waren. Over deze punten zouden we dus graag de query willen uitvoeren. Hiertoe breiden we de structuur uit door aan elke interne knoop β , die linker zoon van zijn vader is, een S -structuur S_β toe te voegen die alle punten onder β bevat (zie Figuur 19). Als we nu met een tijdstip T door



Figuur 19

de boom zoeken, bevatten de S_β -structuren van knopen β die linker zoon

zijn van een knoop op het zoekpad, maar daar zelf niet op liggen, precies alle punten links van het pad, en dus de punten waarop we een query willen uitvoeren (zie Figuur 20).



Figuur 20

Dit zijn maximaal $O(\log N)$ S-structuren. Op iedere S-structuur is de zoek-tijd begrensd door $Q_S(N)$ en dus is de totale zoektijd begrensd door $O(\log N)Q_S(n)$.

Als we een punt p willen invoegen betekent dit dat we een S-structuur moeten bouwen van alle punten onder β (zie Figuur 17, 18). Dit kost soms erg veel tijd, namelijk als B_2 groot is, maar meestal is B_2 erg klein. Dat de gemiddelde insertiontijd laag blijft is eenvoudig als volgt in te zien. Elk punt p in de verzameling komt in ten hoogste $\log N$ structuren voor. Aangezien er nooit structuren worden weggegooid, betekent dit dat hij in maximaal $\log N$ structuren werd gebouwd. Als we de tijd, nodig voor het bouwen van een S-structuur verdelen over de punten waaruit hij gebouwd werd, is dit maximaal $P_S(N)/N$ per punt.

STELLING. Gegeven een ontbindbaar zoekprobleem Q met statische structuur S , dan bestaat er een structuur S_1 om Q in het verleden op te lossen met

$$Q_{S_1}(N) = O(\log N)Q_S(N)$$

$$I_{S_1}^a(N) = O(\log N)P_S(N)/N.$$

De average insertion tijd kan worden veranderd in een worst-case tijd.

VOORBEELDEN. b. Range searching in het verleden kan gedaan worden in

$$Q(N) = O(\log^{d+1} N)$$

$$I(N) = O(\log^d N).$$

d. Naaste buur zoeken in het verleden kan gedaan worden in

$$Q(N) = O(\log^2 N)$$

$$I(N) = O(\log^2 N).$$

Ook in het geval dat zowel insertions als deletions optreden is het mogelijk algemene transformaties naar "in het verleden" problemen te geven voor ontbindbare zoekproblemen (zie [22] voor resultaten van deze strekking).

5. REFERENTIES

- [1] AHO, A.V., J. HOPCROFT & J.D. ULLMAN, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Mass. (1974).
- [2] BENTLEY, J.L., *Multidimensional binary search trees used for associated searching*, Comm. of the ACM 18 (1975), pp. 509-517.
- [3] BENTLEY, J.L., *Decomposable searching problems*, Inf. Proc. Lett. 8 (1979) 244-251.
- [4] BENTLEY, J.L., *Notes on a taxonomy of planar convex hull algorithms*, Techn. Rep., Carnegie Mellon University, 1980.
- [5] BENTLEY, J.L. & D.F. STANAT, *Analysis of range searches in quad-trees*, Inf. Proc. Let 3 (1975) pp. 170-173.
- [6] COMER, D., *The ubiquitous B-tree*, ACM Comput. Surveys 11 (1979) 121-137.
- [7] DOBKIN, D.P. & J.I. MUNRO, *Efficient uses of the past*, Proc. of the 21th IEEE Symp. on Foundations of Computer Science, 1980, 200-206.
- [8] EDELSBRUNNER, H., *Dynamic rectangle intersection searching*, Bericht 47, Inst. f. Informationsverarb., TU Graz, 1980.
- [9] EDELSBRUNNER, H. & H.A. MAURER, *On the intersection of orthogonal objects*, Bericht 60, Inst. f. Informationsverarb., TU Graz, 1980.
- [10] FINKEL, R.A. & J.L. BENTLEY, *Quad-trees: a data structure for retrieval on composite keys*, Acta Informatica 4 (1974) pp. 1-9.

- [11] GOWDA, I.G. & D.G. KIRKPATRICK, *Exploiting linear merging and extra storage in the maintenance of fully dynamic geometric data structures*, Proc. of the 19th Annual Allerton Conference on Communication, Control and Computing, 1980.
- [12] GRAHAM, P.J., *An efficient algorithm for determining the convex hull of a finite planar set*, Inform. Proc. Lett. 1 (1972) 132-133.
- [13] KIRKPATRICK, D.G., *Optimal search in planar subdivisions*, preprint, Dept. of Computer Science, UBC, Vancouver, Canada, 1979.
- [14] KNUTH, D.E., *The art of computer programming, vol. 3: sorting and searching*, Addison-Wesley Publ. Comp., Reading, Mass., 1973.
- [15] LEE, D.T. & C.K. WONG, *Finding intersection of rectangles by range search*, Techn. report, 1980.
- [16] LUEKER, G.S., *A transformation for adding range restriction capability to dynamic data structures for decomposable searching problems*, Techn. Report #129, Dept. of Inform. and Computer Science, University of California, Irving, 1978.
- [17] MAURER, H.A. & TH. OTTMANN, *Dynamic solutions of decomposable searching problems*, Bericht 33, Inst. f. Informationsverarb., TU Graz, 1979.
- [18] MEHLHORN, K., *Lowerbounds on the efficiency of static to dynamic transformations of data structures*, Techn. report A 80/05, Fachber. 10, Universität des Saarlandes, Saarbrücken, 1980.
- [19] MEHLHORN, K., *Balanced trees as a data structure for representing sorted lists*, dit colloquium.
- [20] MEHLHORN, K. & M.H. OVERMARS, *Optimal dynamization of decomposable searching problems*, Inf. Proc. Lett. 12 (1981) 93-98.
- [21] OVERMARS, M.H., *Dynamization of order decomposable set problems*, Techn. report RUU-CS-80-9, University of Utrecht, 1980. (To appear in J. of Algorithms.)
- [22] OVERMARS, M.H., *Searching in the past*, in preparation.
- [23] OVERMARS, M.H. & J. VAN LEEUWEN, *Two general methods for dynamizing decomposable searching problems*, Computing 26 (1981) 155-166.

- [24] OVERMARS, M.H. & J. VAN LEEUWEN, *Some principles for dynamizing decomposable searching problems*, Inf. Proc. Lett. 12 (1981) 49-53.
- [25] OVERMARS, M.H. & J. VAN LEEUWEN, *Dynamic multidimensional data structures based on quad- and k-d trees*, Techn. report RUU-CS-80-2, Dept. of Computer Science, University of Utrecht, 1980.
- [26] OVERMARS, M.H. & J. VAN LEEUWEN, *Maintenance of configurations in the plane*, Techn. report RUU-CS-79-9, Dept. of Computer Science, University of Utrecht, 1979/1980. (To appear in J.Comp. Syst.Sci.)
- [27] OVERMARS, M.H. & J. VAN LEEUWEN, *Notes on maintenance of configurations in the plane*, Techn. report RUU-CS-80-5, Dept. of Computer Science, University of Utrecht, 1980.
- [28] OVERMARS, M.H. & J. VAN LEEUWEN, *Dynamizations of decomposable searching problems yielding good worst case bounds*, in: P. Deussen (ed.), Theoretical Computer Science (5th GI-Conf.), Lect. Notes in Comp. Sci. 104, Springer-Verlag, 1981, 224-233.
- [29] OVERMARS, M.H. & J. VAN LEEUWEN, *Worst case optimal insertion and deletion methods for decomposable searching problems*, Techn. report RUU-CS-80-10, Dept. of Computer Science, University of Utrecht, 1980. (To appear in Inf. Proc. Lett.)
- [30] PREPARATA, F.P., *An optimal real-time algorithm for planar convex hulls*, C. ACM 22 (1979) 402-405.
- [31] PREPARATA, F.P. & S.J. HONG, *Convex hulls of finite sets of points in two and three dimensions*, C. ACM 20 (1977) 87-93.
- [32] SAMET, H., *Deletion in two dimensional quad trees*, C. ACM 23(1980) 703-710.
- [33] SAXE, J.B. & J.L. BENTLEY, *Transforming static data structures to dynamic structures*, Proc. of the 20th Annual IEEE Symp. on Foundations of Computer Science, San Juan, Puerto Rico, 1979, 148-168.
- [34] SHAMOS, M.I., *Computational geometry*, Ph.D. Thesis Yale University, 1978 (to be published).
- [35] SHAMOS, M.I. & D. HOEY, *Closest-point problems*, Proc. of the 16th Annual IEEE Symp. on Foundations of Computer Science, 1976, 151-162.

- [36] SILVA-FILHO, Y.V., *Average case analysis of region search in balanced k-d trees*, Inform. Proc. Lett. 8 (1979) 219-223.
- [37] VAN LEEUWEN, J. & H.A. MAURER, *Dynamic systems of static data structures*, Bericht 42, Inst. f. Informationsverarbeitung, TU Graz, 1980.
- [38] VAN LEEUWEN, J. & D. WOOD, *Dynamization of decomposable searching problems*, Inform. Proc. Lett. 10 (1980) pp. 51-56.
- [39] WILLARD, D.E., *New data structures for orthogonal queries*, TR-22-78, Aiken Computation Lab., Harvard University, 1978.
- [40] WILLARD, D.E., *The super B-tree algorithm*, TR-03-79, Aiken Computation Lab., Harvard University, 1979.

C. PARALLELE BEREKENINGEN EN VLSI

VLSI EN DE COMPLEXITEIT VAN BEREKENINGEN

M. REM

0. INLEIDING

VLSI (Very Large Scale Integration) is de techniek waarmee volledige rekenmachines gerealiseerd kunnen worden op een klein plakje silicium, de zg. chip. De opslag van informatie en de verwerking ervan worden op een silicium-chip door dezelfde bouwstenen (transistors en verbindingen daartussen) gerealiseerd en er is dan ook geen enkele technische reden om deze twee taken ruimtelijk te scheiden. Integendeel, de kosten van een berekening in VLSI zitten hem voornamelijk in het verplaatsen van informatie over de chip en er is alles voor te zeggen om de informatieopslag en -verwerking zoveel mogelijk ruimtelijk te vermengen om op die manier een hoge graad van parallellisme bij het uitvoeren van berekeningen te bewerkstelligen.

Niet alle algorithmen zijn even geschikt om de nieuwe mogelijkheden van het VLSI-medium uit te buiten. Aangezien met de toename van het aantal transistors per chip de verplaatsing van de informatie ten opzichte van de verwerking ervan steeds duurder wordt, zijn vooral berekeningen met een sterk "lokaal gedrag" aantrekkelijk voor dit medium. Het zal duidelijk zijn dat het beschouwen van de kosten van het verplaatsen van informatie een complexiteitsmaat oplevert die afwijkt van de thans gebruikelijke (op de Turing machine gebaseerde) maat.

In dit artikel ga ik in op de vraag hoe een fysisch verantwoorde complexiteitsmaat voor in VLSI gerealiseerde berekeningen eruit zou kunnen zien. In hoofdstuk 1 geef ik een zeer globale behandeling van de fysische eigenschappen van MOS-chips. In hoofdstuk 2 laat ik zien hoe een bepaalde MOS-techniek, CMOS, kan worden opgevat als een discreet universum, waarin de berekeningen slechts 0 en 1 als operanden en aan/uit-schakelaars als operatoren hebben. We kunnen door deze abstractie van de fysische eigenschappen het ontwerpen van CMOS-chips dan opvatten als een programmeertaak. In de drie daarop volgende hoofdstukken ga ik in op de vraag wat we kunnen zeggen

over de complexiteit van dergelijke in CMOS gerealiseerde programma's.

1. MOS-CHIPS

We geven een vereenvoudigde, maar conceptueel juiste, behandeling van Metal-Oxide-Semiconductor-chips. Een MOS-chip is een dun plakje silicium van ongeveer 5 bij 5 mm, waarop in een aantal lagen geleidende paden zijn aangebracht. Deze lagen zijn onderling gescheiden door isolerend materiaal, siliciumoxide, met gaten op sommige plaatsen om paden in de verschillende lagen met elkaar te verbinden. Van boven naar beneden onderscheiden we achtereenvolgens een laag van metaal (meestal aluminium), een laag van polysilicium en een laag van "diffusie" (silicium met een doping van ionen). Daar waar een polysilicium-pad een diffusie-pad kruist ontstaat een transistor: de spanning op het polysilicium-pad bepaalt of het diffusie-pad ter plaatse al dan niet geleidend is. Een dergelijke transistor wordt wel een FET (Field-Effect Transistor) genoemd.

Het zal duidelijk zijn dat een transistor nauwelijks ruimte inneemt: het is gewoon de kruising van twee geleidende paden. De meeste ruimte op de chip wordt ingenomen door de verbindingen tussen de transistors. Het fabricageproces is dusdanig dat slechts ongeveer één op de vijf chips goed vervaardigd wordt. Fouten worden o.a. veroorzaakt door het niet precies op elkaar passen van de verschillende lagen. De werkende chips worden in de fabriek door testen geselecteerd. De rest wordt weggegooid (of bereiken door de achterdeur soms toch nog klanten!). De fabricagetechniek wordt voortdurend verbeterd. Dat leidt niet tot vergroting van de bovengenoemde factor 1/5, maar tot smallere geleidende paden en/of chips met een grotere oppervlakte, zodat er steeds meer transistors op een chip passen.

Met de huidige fabricagetechniek is de minimale breedte van de geleidende paden ongeveer $5 \cdot 10^{-6}$ m. Dan passen er zo'n 10.000 tot 100.000 transistors op een chip, genoeg om bijvoorbeeld een geheugen van 64K bits te realiseren op een chip. De verwachting is dat het aantal transistors per chip in de komende tien jaar zal toenemen tot 10 à 20 miljoen [1]. Aan de rand van de chip bevinden zich de aansluitpunten voor communicatie met de omgeving. Door de grootte van de chip is het aantal aansluitpunten beperkt, 80 is thans ongeveer het maximum. De aansluitpunten worden behalve voor het uitwisselen van gegevens ook gebruikt voor de ontvangst van spanning (de "logische 1", bijv. 5 V), aarde (de "logische 0") en de klok. De laatste lijkt meer op een metronoom dan op een klok. Hij levert een snelle afwisseling van logische enen en nullen en wordt gebruikt voor de synchronisatie

van de berekeningen op de chip.

We gaan nu nader in op de werking van de transistor. Doordat tijdens de fabricage de diffusie-laag wordt aangebracht na de polysilicium-laag is op plaatsen waar een polysilicium-pad een diffusie-pad kruist het onderliggende diffusie-pad onderbroken. We noemen dit missende stukje diffusie-pad het "channel" van de transistor. Het diffusie-pad is slechts geleidend indien de ladingsdragers in het ene deel van het diffusie-pad door het channel naar het andere deel van het diffusie-pad kunnen worden "getrokken".

We onderscheiden twee soorten diffusie: diffusie met negatieve ladingsdragers (electronen) and diffusie met positieve ladingsdragers ("holes", afwezige electronen). We noemen deze twee soorten respectievelijk N-type en P-type. De valentie van de ionen die gebruikt zijn bij de doping bepaalt tot welke soort de diffusie behoort. Het diffusie-pad van een transistor zal natuurlijk alleen een stroom doorlaten wanneer zich op het ene deel van het diffusie-pad meer ladingsdragers bevinden dan op het andere deel. We noemen het deel met de meeste ladingsdragers de "source" en het andere deel de "drain". (Let erop dat dit in het algemeen een dynamisch onderscheid is, tijdens de werking kunnen de rollen van source en drain wisselen.) Het kruisende polysilicium-pad noemen we de gate van de transistor.

Een transistor is "aan" (d.w.z. het diffusie-pad is geleidend) wanneer de gate voldoende aantrekking uitoefent op de ladingsdragers in de source om ze via het channel naar de drain te trekken. Het is dus het spanningsverschil tussen gate en source ($V_g - V_s$) dat bepaalt of de transistor aan is. Voor iedere transistor is er een drempelspanning V_t die bepaalt voor welke waarde van $V_g - V_s$ de transistor schakelt tussen aan en uit. Een N-type transistor, een transistor met N-type diffusie, moet om aan te zijn negatieve ladingsdragers aantrekken. Dit zal dus een grote waarde van $V_g - V_s$ vereisen. Een P-type transistor, anderzijds, moet om aan te zijn positieve ladingsdragers aantrekken en zal dus aan zijn voor kleine waarden van $V_g - V_s$. In formule:

		aan	uit
(1)	N-type	$V_g - V_s > V_t$	$V_g - V_s < V_t$
	P-type	$V_g - V_s < V_t$	$V_g - V_s > V_t$

De waarde van V_t kan voor iedere transistor tijdens fabricage beïnvloed worden door de mate waarin het channel van een doping met N-type of P-type ionen wordt voorzien. Zo kan bepaald worden welke transistors in rust ($V_g = V_s$) aan en welke uit zijn. Een transistor die voor $V_g = V_s$ uit

is wordt een enhancement transistor genoemd, een transistor die voor $V_g = V_s$ aan is een depletion transistor. Lettend op formule (1) zien we dat een enhancement transistor een transistor is waarvan V_t en de lading van de ladingsdragers van tegengesteld teken zijn en een depletion transistor een waarbij ze van hetzelfde teken zijn.

2. CMOS

De naam CMOS staat voor Complementary MOS, omdat op dezelfde chip zowel N-type als P-type transistors voorkomen. CMOS^{*)} is de techniek die gebruikt wordt voor chips in horloges, "credit card" zakrekenmachines, pace makers, enz. Dit vanwege zijn zuinigheid met energie. In CMOS gebruiken we slechts enhancement transistors. Hoe goed werken de N-type en P-type enhancement transistors als schakelaars?

N-type

Deze heeft $V_t > 0$. Hij is aan als V_g een hoge waarde heeft. Kies $V_g = 1$, de spanning van de spanningsbron. Volgens formule (1) is de transistor dan aan als

$$(2) \quad 1 - V_s > V_t$$

In het bijzonder is hij uit als $V_s = 1$. Deze transistor is niet in staat een 1 op de source naar de drain te transporteren. Precieser, de N-type transistor is slechts een goede geleider voor waarden V_s die voldoen aan

$$(3) \quad 0 \leq V_s < 1 - V_t$$

Als $V_g = 1$ en V_s voldoet aan (3) zal V_d gelijk aan V_s worden, maar voor grotere waarden van V_s kunnen we slechts garanderen dat V_d ten hoogste de drempelspanning V_t van 1 zal verschillen: er komt geen betrouwbare 1 meer uit. Nog erger wordt het als we zo'n zwakke 1, zeg $1 - V_t$, weer gebruiken als waarde op de gate van een volgende N-type transistor. Zoals de lezer gemakkelijk kan verifiëren, hoeft er uit die volgende transistor nog maar $1 - 2V_t$ te komen. Een gebruikelijke drempelwaarde is 0.2 (als we de spanningsbron 1 noemen), zodat met dergelijke schakelaars nauwelijks betrouwbaar te schakelen valt. Zo erg is het gelukkig niet: we zullen laten zien dat we het gebruik van de N-type transistor kunnen beperken tot het gebied

^{*)} Voor de technische ingewijde lezer merken we op dat we hier niet refereren aan "bulk CMOS" maar aan CMOS op een isolerende laag, zoals CMOS/SOS.

waarin hij wel betrouwbaar schakelt.

P-type

Voor P-type transistors geldt, mutatis mutandis, hetzelfde verhaal. Hij heeft $V_t < 0$. Kies $V_g = 0$. De transistor is dan aan als

$$(4) \quad V_s > -V_t$$

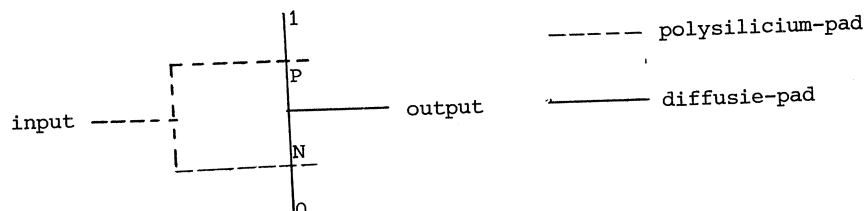
De transistor geleidt slechts waarden V_s die voldoen aan

$$(5) \quad -V_t < V_s \leq 1$$

Als $0 \leq V_s < -V_t$ weten we slechts dat V_d zal voldoen aan $0 \leq V_d < -V_t$.

(Niet $V_d > -V_t$ want dan zou de drain de source worden.)

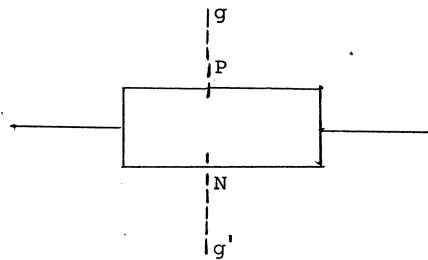
Uit bovenstaande beschouwingen leren we dat de ene transistor eigenlijk alleen bruikbaar is voor het (al dan niet) doorlaten van nullen en de andere van enen. Het lijkt erop dat ze daarmee niet erg bruikbaar zijn, maar dat valt mee. Lat eens op de volgende constructie:



Het diffusie-pad van een P-type transistor is enerzijds verbonden met de spanningsbron ("1") en anderzijds met het diffusie-pad van een N-type transistor. De andere zijde van het diffusie-pad van de N-type transistor is verbonden met aarde ("0"). De gates van de transistors zijn verbonden met de input, de plaats waar de twee diffusie-paden elkaar ontmoeten vormt de output.

Als de input de waarde 1 heeft is de N-type transistor aan en de andere uit. De output is dan verbonden met aarde, de output is dan 0. Als de input de waarde 0 heeft is de P-type transistor aan en de N-type uit: De output is 1. Dit is een inverter. Maar wat meer is: de N-type transistor wordt slechts gebruikt voor het transport van een 0 en de P-type van een 1. Beide transistors worden gebruikt in het gebied waarin ze betrouwbaar zijn.

Hoe maken we nu een schakelaar die zowel nullen als enen betrouwbaar doorlaat? Beschouw de volgende combinatie van een N-type en een P-type transistor:



We gebruiken deze combinatie slechts met als waarden op de gates (g en g') elkaars inversen, d.w.z. als de een de waarde 0 heeft, heeft the andere de waarde 1. Voor $g = 1$ (en dus $g' = 0$) zijn beide transistors uit, voor $g = 0$ zijn ze beide aan.

We gaan voor het geval $g = 0$ na hoe goed deze combinatie nullen en enen over het diffusie-pad geleidt. Laat een van beide kanten (links of rechts op het plaatje) een 0 of een 1 bevatten, d.w.z. verbonden zijn met aarde of de spanningsbron. Als de waarde 0 is dan is de P-type transistor uit en de N-type aan: de andere kant wordt ook 0. Voor de waarde 1 is de N-type transistor uit en de P-type aan, de andere kant wordt ook 1. Zo hebben we door beide typen transistors te combineren een perfecte schakelaar gemaakt.

Nu onze schakelaars betrouwbaar schakelen met zowel nullen als enen kunnen we de output van de ene schakelaar weer gebruiken om een andere schakelaar te openen en te sluiten. Zo kunnen we hele netwerken van schakelaars maken en dat is in feite wat een CMOS-chip is: een groot netwerk van binaire schakelaars. We hebben zo een discreet universum geschapen waarin alle waarden binair zijn en waarin we rekenen door schakelaars te openen en te sluiten.

We kunnen het ontwerpen van dergelijke netwerken opvatten als een programmeertaak. Het speelt zich immers af in een wereld waarin we geabstraheerd hebben van de continue fysische verschijnselen^{*)}. Een notatie voor dergelijke programma's is het onderwerp van [5]. Daarin wordt ook o.a. ingegaan op het voorkómen van kortsluiting (verbinden van 0 en 1) door middel van syntactische regels. In de daar ingevoerde notatie heeft onze inverter

^{*)} Dit is niet helemaal waar: we hebben de tijd nog niet gediscrètiseerd.

het volgende programma:

```
com inverter(in?,out!):
    in' → out = 1; in → out = 0
moc
```

Twee inverters kunnen zo met elkaar verbonden worden dat ze een boolean variabele vormen:

```
com flipflop(in?,ld?,val!):
    sub i1,i2: inverter;
    i2.in = i1.out;
    ld' → i1.in = i2.out; ld → i1.in = in;
    val = i2.out
moc
```

U ziet aan deze twee voorbeelden dat we in hetzelfde medium zowel de operaties op waarden als de opslag van waarden kunnen realiseren. U ziet er ook aan dat dit wel een heel andere manier van programmeren is dan we gewend zijn.

Alles goed en wel, maar hoe efficiënt is zo'n berekening nu? Het is duidelijk dat er tijdens een berekening veel tegelijkertijd kan gebeuren, wat dat betreft gaat het erg snel. Maar de informatie moet wel van de ene schakelaar naar de andere "stromen". Hoeveel tijd kost dat? Dat is het onderwerp van het volgende hoofdstuk.

3. TIJD

Wat kunnen we zeggen over de tijdsduur van een berekening? Het tellen van het aantal omzettingen van schakelaars is uiteraard geen goede maat, omdat het de gelijktijdigheid waarmee deze omzettingen kunnen plaatsvinden niet in beschouwing neemt. Verder kost het schakelen nauwelijks tijd, of beter: een belangrijk gedeelte van de tijd gaat zitten in het transport van informatie over de paden. Met het kleiner worden van de transistors op de chip wordt de tijd die nodig is om een signaal van de ene kant van de chip naar de andere te krijgen steeds groter. Dit omdat het kleiner maken van de transistors, teneinde de energiedissipatie per oppervlakte-eenheid niet te laten toenemen, gepaard moet gaan met een evenredige verlaging van het voltage van de spanningsbron.

Laat s_0 de minimale breedte van een pad zijn, zodat een minimale transistor oppervlakte s_0^2 heeft. (In de literatuur komen we ook wel een maat λ tegen, $s_0 = 2\lambda$.) We gaan na hoe lang het duurt een signaal te transporteren over een pad ter lengte l als het signaal afkomstig is van een minimale transistor. We nemen aan dat het pad dikte en breedte s heeft en ook een afstand s tot naburige paden en tot de onderliggende laag.

De totale tijdsduur voor het transport van het signaal is

$$(6) \quad (R_t + R_p) C_p$$

waarin R_t de weerstand van de transistor is die het signaal voortbrengt, R_p de weerstand van het pad en C_p de capaciteit van het pad. De weerstand van het pad is evenredig met de lengte en omgekeerd evenredig met de doorsnede:

$$(7) \quad R_p = \rho \frac{l}{s}$$

De capaciteit van een pad is omgekeerd evenredig met de afstand tot naburige paden en lagen en evenredig met de oppervlakte van de zijde die naar het naburige pad of the naburige laag is gekeerd:

$$(8) \quad C_p = \epsilon \frac{s l}{s} = \epsilon l$$

Het product van R_p en C_p is dus al kwadratisch in l . Dat is slecht nieuws: de tijdsduur om een signaal over een pad te sturen is kwadratisch in de lengte van het pad. Nu is dat minder erg dan het lijkt; de R_t , de weerstand van een minimale transistor, is namelijk zeer groot en vormt daarmee de dominerende term. We kunnen hier iets aan doen door een grotere transistor te gebruiken. Maar daarvoor geldt dat die dan eerst moeten worden opgeladen door de minimale transistor en het lijkt erop dat we zo het probleem alleen maar verschuiven. Maar dat is niet zo, we kunnen namelijk een rij van steeds groter wordende transistors gebruiken. Dergelijke transistors worden dan meestal drivers genoemd.

Er is een eenvoudige rekenregel om uit te rekenen hoeveel tijd het kost om een transistor een andere transistor te laten opladen [1]. Laat τ de tijdsduur zijn om een minimale transistor de gate van een andere er direct naast gelegen minimale transistor te laten opladen. De grootheid τ is de

eenheid van tijd die doorgaans gebruikt wordt by beschouwingen over de werking van MOS-chips. De rekenregel is dat de tijd die een driver met capaciteit C_1 nodig heeft om een andere er direct naast gelegen driver met capaciteit C_2 ($C_2 > C_1$) op te laden gelijk is aan

$$(9) \quad \tau \frac{C_2}{C_1}$$

Noem de capaciteit van een minimale transistor even C . Dan laten we deze een driver opladen met capaciteit αC , deze tweede driver laten we een driver opladen met capaciteit $\alpha^2 C$, enz. We gaan hiermee door tot de laatste een capaciteit heeft die ongeveer gelijk is aan C_P , de capaciteit van het pad. Het aantal drivers dat we dan nodig hebben is

$$\alpha \log \frac{C_P}{C}$$

De tijdsduur voor iedere driver om de volgende op te laden is $\alpha \tau$, zodat we voor de vertraging in de drivers vinden:

$$(10) \quad T_d = \alpha \tau \alpha \log \frac{C_P}{C}$$

De capaciteit van een minimale transistor is ϵs_0 , zodat uit (8) en (10) volgt

$$(11) \quad T_d = \alpha \tau \alpha \log \frac{l}{s_0}$$

We kunnen nu in plaats van (6) de volgende formule gebruiken. Voor de totale tijdsduur T van het transport geldt

$$(12) \quad T = T_d + R_P C_P$$

Uit (7), (8), (11) en (12) volgt dan

$$(13) \quad T = \alpha \tau \alpha \log \frac{l}{s_0} + \rho \epsilon \frac{l^2}{s^2}$$

We hebben nu een formule met een term die logaritmisch is in de lengte l en een kwadratische term. De kwadratische term kunnen we echter verkleinen door s te vergroten, d.w.z. door het signaal via een dikker en breder pad te leiden. Dit kunnen we bijvoorbeeld bewerkstelligen door de chip uit te

rusten met meer dan één metaallaag, waarbij op een hogere metaallaag de paden breder, dikker en langer zijn. Zo zien we nu al chips met een tweede metaallaag voor de grotere paden. Het aantal metaallagen zal in de toekomst verder toenemen. We zien thans ook de introductie van een extra verbindingslaag tussen de chip en het pc-board: een laag van weer wat dikkere paden waarin een klein aantal chips direct met elkaar verbonden worden, een techniek die bekend staat als "multi layer ceramics". We zullen bij gegeven ℓ de s dusdanig kiezen dat de kwadratische term niet overheerst. We gaan daarbij voorbij aan het feit dat s uit een discrete verzameling gekozen moet worden.

We beperken ons tot die waarden van ℓ waarbij de kwadratische term in (13) minder snel toeneemt dan de logaritmische. We bepalen daartoe de waarde van ℓ waarbij de afgeleiden naar ℓ van de twee termen gelijk zijn:

$$(14) \quad \frac{d}{d\ell} \alpha \tau^\alpha \log \frac{\ell}{s_0} = \frac{\alpha \tau}{\ell \ln \alpha}$$

$$(15) \quad \frac{d}{d\ell} \rho \epsilon \frac{\ell^2}{s} = \frac{2\rho \epsilon \ell}{s}$$

Als we een gegeven afstand ℓ moeten overbruggen kiezen we daartoe een pad met een s waarvoor de afgeleiden in (14) en (15) gelijk zijn:

$$(16) \quad s = \ell \sqrt{\left(\frac{2\rho \epsilon \ln \alpha}{\alpha \tau} \right)}$$

Als we (16) invullen in (13) vinden we voor de tijdsduur T

$$T = \tau \frac{\alpha}{\ln \alpha} \left(\ln \frac{\ell}{s_0} + \frac{1}{2} \right)$$

Of ongeveer:

$$(17) \quad T = \tau \alpha^\alpha \log \frac{\ell}{s_0}$$

De conclusie is dat het mogelijk is een signaal over een afstand ℓ te krijgen in een tijd die logaritmisch is in ℓ door een pad te kiezen op een laag met voldoende grote s . Het interessante fenomeen is dat als we de lagen op die manier kiezen, dat dan de paden op dezelfde laag ongeveer even lang zijn en dat, let op formule (16), alle paden ongeveer dezelfde verhouding tussen lengte en breedte hebben. De verschillende lagen zien er ongeveer hetzelfde uit, op een volgende laag is alleen alles met eenzelfde factor vergroot.

We hebben in het voorgaande de aanname gemaakt dat de vertragingstijd niet beperkt wordt door de lichtsnelheid. We gaan thans na hoe reëel die aanname is.

Laat v de maximum snelheid zijn van ladingsdragers in een halfgeleider (bijv. electronen in silicium). De tijd τ wordt voornamelijk bepaald door de tijdsduur die de ladingsdragers nodig hebben om het channel (met lengte s_0) te passeren, zodat ongeveer geldt

$$(18) \quad \tau v = s_0.$$

Dit invullend in (17) vinden we

$$(19) \quad T = \frac{\alpha s_0}{v} \alpha \log \frac{l}{s_0}.$$

De snelheid $\frac{dl}{dT}$ van het signaal wordt begrensd door de lichtsnelheid c :

$$(20) \quad \frac{dl}{dT} < c.$$

Of, met behulp van (19):

$$(21) \quad l < \frac{c \alpha s_0}{v \ln \alpha}.$$

De grootheden c en v hebben ruwweg de volgende waarden:

$$c = 1.5 \times 10^8 \text{ m/s}$$

$$v = 10^4 \text{ m/s}.$$

Voor chips is $\alpha / \ln \alpha$ ongeveer 6, zodat we vinden

$$(22) \quad l < 10^5 s_0.$$

Bij de huidige MOS-techniek is s_0 ongeveer 5×10^{-6} m. Dat levert op

$$l < 50 \text{ cm}.$$

We ondervinden dus thans zelfs op een pc-board nog niet de beperkende invloed van de lichtsnelheid. Volgens [1] ligt de fysische ondergrens voor

de padbreedte in MOS-chips op 2.5×10^{-7} m. Deze waarde levert voor (22) op

$$l < 2.5 \text{ cm.}$$

Dat is een lengte die vijf maal die van de huidige chips is. Vergeleken met de huidige chips zou dat een chip zijn met een 25 maal zo grote oppervlakte en per oppervlakte-eenheid 400 maal zo veel transistors, zo'n miljard transistors in totaal. We hoeven niet bang te zijn dat we die spoedig gaan maken.

Op een printed circuit board zullen we wel last krijgen van de licht-snelheidsbeperking. Op een chip blijft de beperking door weerstand en capaciteit belangrijker.

Er is echter nog een beperkende factor waardoor het niet voor iedere berekening mogelijk zal zijn het transport van signalen in een logaritmische tijd uit te voeren. Alle drivers bevinden zich namelijk op het laagste niveau: ze moeten worden gerealiseerd in de diffusie- en in de polysilicium-laag. In het volgende hoofdstuk zullen we berekenen hoeveel oppervlakte dat vergt. Als we niet in staat zijn voldoende drivers te herbergen op het laagste niveau hoeven we niet direct in een kwadratische tijd te vervallen. Er is nog het compromis om langs het pad op equidistante punten een versterker te plaatsen. Dat maakt de tijdsduur die nodig is om een signaal over een pad te transporteren lineair in de lengte van het pad. Een inverter is een voorbeeld van een versterker, of twee inverters als we een versterker niet het signaal willen laten invertieren. Ieder stukje tussen twee versterkers vergt dezelfde tijd en zo wordt de totale tijdsduur lineair in de lengte van het pad. De coëfficiënt van deze lineaire tijdsfactor hangt uiteraard weer af van hoeveel oppervlakte we beschikbaar hebben op het laagste niveau om de drivers te herbergen.

4. RUIMTE

Laat van een transistor het diffusie-pad breedte d hebben, het kruisende polysilicium-pad breedte p en de isolerende siliciumoxide daartussen (het channel) dikte h . Dan geldt voor de capaciteit C van de transistor

$$(23) \quad C = \epsilon \frac{dp}{h}$$

De h is voor alle transistors gelijk en de capaciteit van transistors is dus evenredig met hun oppervlakte dp . De minimale transistor heeft opper-

vlakke s_0^2 . In de serie transistors die we gebruiken om een signaal over een pad te sturen heeft de volgende oppervlakte αs_0^2 , de derde oppervlakte $\alpha^2 s_0^2$, enz. De totale oppervlakte A van de drivers wordt dus

$$A = s_0^2 (1 + \alpha + \alpha^2 + \dots) \quad (\alpha \log l \text{ termen})$$

$$A = \frac{s_0^2 (l-1)}{\alpha-1}$$

Of ongeveer:

$$(24) \quad A = \frac{s_0^2 l}{\alpha-1}$$

Teneinde de oppervlakte te meten die een CMOS-schakeling inneemt moeten we de transistors afbeelden op het platte vlak. Dit moet zodanig gebeuren dat een transistor die een signaal over een verbinding met lengte l moet sturen een oppervlakte $s_0^2 l / (\alpha - 1)$ krijgt toegewezen. Verder moet er ruimte zijn voor de verbindingen. Afhankelijk van de lengte van een verbinding kan deze worden uitgevoerd in diffusie, polysilicium of in een van de metaallagen. Diffusie en polysilicium vormen daarbij in feite één laag, kruisingen scheppen immers nieuwe transistors. Verbindingen op dit laagste niveau zijn uiteraard alleen mogelijk voorzover we daarvoor ruimte tussen de transistors openlaten.

Dus de oppervlakte voor de drivers is evenredig met de padlengte. Dat komt goed uit. Stel dat we de lengte en de breedte van een chip verdubbelen, dan verdubbelt ook de lengte van de langste paden (paden van de ene naar de andere kant van de chip) en de oppervlakte die hun drivers innemen. Maar de oppervlakte van de hele chip is verviervoudigd, zodat we gelukkig ook het aantal paden kunnen verdubbelen.

Voordat we nu zonder meer mogen aannemen dat de tijd logaritmisch is in de afstand, moeten we wel opmerken dat dat alleen maar zo is als we niet te veel lange paden hebben. Immers deze komen op hogere lagen waar de paden een grotere oppervlakte hebben, maar de oppervlakte van iedere laag is gelijk. Als op het volgende niveau de paden β maal zo lang, dik en breed zijn, dan mogen daar maar $1/\beta^2$ maal zoveel paden voorkomen. Laat het laagste niveau nummer 0 zijn en N_i het aantal paden op niveau i ($i \geq 0$), dan moet gelden

$$(25) \quad N_i = N_0 \beta^{-2i}$$

Het aantal paden van een bepaalde lengte moet dus exponentieel snel afnemen met de lengte. Dit is een vrij strenge eis. Hij suggereert dat efficiënte chips een boom-achtige structuur moeten hebben. Dit is opnieuw een argument dat pleit voor hiërarchische chips [2,4].

5. COMPLEXITEITSMATEN

Ik ben nogal uitvoerig ingegaan op de vertragingen in paden en het zal de lezer opgevallen zijn hoeveel moeite het ons gekost heeft om deze logaritmisch in de padlengte te laten zijn. Alleen een weloverwogen ontwerp met zeer lokale communicatie tussen transistors kan leiden tot logaritmische vertragingen. Ik ben hier zo precies in geweest omdat in het werk van anderen [3,6] doorgaans wordt aangenomen dat de vertraging in een pad onafhankelijk is van de lengte. Deze misvatting is mogelijk terug te voeren tot complexiteitsberekeningen aan hiërarchische geheugens [2], waaruit de oppervlakkige lezer deze conclusie zou kunnen trekken. Zo schrijft Thompson in [6]:

"The propagation time can be made independent of the length of the wire, by fitting larger drivers to longer wires. Larger drivers of course occupy more area, but need not take more than 10% of the area of the wire they drive. By fudging λ upwards by 5%, the area of the driver is thus absorbed into the area of its wire."

We hebben gezien dat de oppervlakte van de drivers inderdaad evenredig is met de lengte van het pad, maar wat Thompson vergeet is dat het opladen van de grotere driver ook tijd kost. Door de keuze van een rij exponentieel groeiende drivers waren we in staat dit een tijd te doen zijn die logaritmisch is in de lengte van het pad, een techniek waarvoor we alleen voldoende ruimte hebben als er weinig lange paden zijn.

We kunnen oppervlakte en tijd tegen elkaar uitwisselen. Door kleinere drivers te gebruiken kost een berekening meer tijd maar minder oppervlakte. Een maat voor de complexiteit van een berekening zal daarom zowel op oppervlakte als tijd betrekking moeten hebben.

Laat A de oppervlakte van een berekening zijn en T de tijdsduur. Thompson introduceert als maat voor de complexiteit AT^2 . De reden dat hij deze maat kiest is gelegen in de manier waarop hij de oppervlakte van een berekening vaststelt. Hij introduceert hiertoe het begrip "minimale bisectiebreedte" (mbb) van een graaf. De mbb van een graaf is het minimaal aantal

takken dat verwijderd moet worden om de graaf in twee onverbonden delen met (ongeveer) evenveel punten te verdelen. De mbb van een boom is 1, de mbb van een volledige graaf van $2n$ punten is n^2 .

Er geldt dat de oppervlakte van een chip waarvan de paden een graaf met mbb ω vormen tenminste $\omega^2/4$ is. Voor de tijdsduur van berekeningen vinden we, althans wat betreft de berekeningen in [6], een uitdrukking met ω in de noemer. Door dan als complexiteitsmaat AT^2 te nemen verdwijnt de, lastig te bepalen, ω weer. Ik kan mij zeer wel voorstellen dat het adopteren van een meer realistische maat voor de vertraging van signalen in paden zal leiden tot ander complexiteitsmaten dan AT^2 .

Een ander verband tussen oppervlakte en tijd dat wel als complexiteitsmaat wordt gekozen is Te^A . Deze maat neemt de eerder gesignaleerde moeilijkheden bij de fabricage van chips in beschouwing. We zijn niet in staat grote chips betrouwbaar te vervaardigen. Neem even aan dat de productiefouten random over de oppervlakte van de chip verspreid liggen. Eén fout is voldoende om de chip waardeloos te doen zijn. De kans dat een chip met oppervlakte A goed is is dan $e^{-\mu A}$. Vandaar dat bij deze maat niet op de oppervlakte A gelet wordt maar op e^A .

Mead propageert als complexiteitsmaat de hoeveelheid energie die nodig is om een berekening uit te voeren, uitgedrukt als veelvoud van de hoeveelheid energie die nodig is om een elementaire logische operatie uit te voeren. Een uitwerking van dit idee kan gevonden worden in hoofdstuk 9 van [1]. Hierin wordt ook een verband gelegd tussen deze maat en entropie. Volgens [6] kan als eerste benadering van energie het produkt van oppervlakte en tijd genomen worden. Dat is ook de maat die gebruikt is in [2].

6. REFERENTIES

- [1] MEAD, C. & L. CONWAY, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass. 1980.
- [2] MEAD, C.A. & M. REM, *Cost and performance of VLSI computing structures*, IEEE Journal of Solid-State Circuits, Vol. SC-14, No. 2, april 1979. pp. 455-462.
- [3] PREPARATA, F.P. & J.E. VUILLEMIN, *Area-Time Optimal VLSI Networks Based on the Cube-Connected-Cycles*, INRIA Rapports de Recherche No.13, Le Chesnay, Frankrijk, maart 1980.

- [4] REM, M., *Wiskunde en Programmeren*, Intreerede, Technische Hogeschool Eindhoven, december 1979.
- [5] REM, M. & C. MEAD, *A notation for designing restoring logic circuitry in CMOS*, Second Caltech Conference on VLSI, California Institute of Technology, Pasadena, Cal. januari 1981.
- [6] THOMPSON, C.D., *Area-time complexity for VLSI*, Proc. of the 11th Annual ACM Symposium on the Theory of Computing, mei 1979. pp. 81-88.

A COMBINATORIAL LIMIT TO THE COMPUTING POWER OF V.L.S.I. CIRCUITS

J. VUILLEMIN

We introduce a property of boolean functions, called *transitivity* which holds of integer, polynomial, and matrix products as well as of many interesting related computational problems. We show that the area of any circuit computing a transitive function grows quadratically with the circuit's maximum data-rate, expressed in bit/second. This result provides a precise analytic expression of an area-time tradeoff for a wide class of V.L.S.I. circuits. Furthermore, (as shown elsewhere), this tradeoff is achievable. We have thus matching (to within a constant multiplicative factor) upper and lower complexity bounds for the three above products, in the V.L.S.I. circuits computational model.

1. INTRODUCTION

Consider the following *problem list*:

1. Cyclic Shifts:

input $(x_0, \dots, x_{p-1}, s)$ with $x_i \in \{0, 1\}$, $0 \leq s < p$;
output $(z_0, \dots, z_{p-1})$ with $z_i = x_{i+s \bmod p}$;
size $N = p$.

2. Integer Product:

input $(x_0, \dots, x_{p-1}, y_0, \dots, y_{p-1})$ with $x_i, y_i \in \{0, 1\}$;
output $(z_0, \dots, z_{2p-1})$ with $z_i \in \{0, 1\}$ and
 $x = \sum_{0 \leq i} x_i \cdot 2^i$, $y = \sum_{0 \leq i} y_i \cdot 2^i$, $z = x \cdot y = \sum_{0 \leq i} z_i \cdot 2^i$;
size $N = 2p$.

3. Convolution (product of two polynomials with integer coefficients):

input $(x_0, \dots, x_p, y_0, \dots, y_p)$ with $0 \leq x_i, y_i < 2^k$;
output $(z_0, \dots, z_{2p})$ with $x(t) = \sum_{0 \leq i} x_i \cdot t^i$, $y(t) = \sum_{0 \leq i} y_i \cdot t^i$,

$Z(t) = X(t) \cdot Y(t) = \sum_{0 \leq i} z_i \cdot t^i$ for any formal t ;
 size $N = (2p+1)(2k+\log_2 p)$.

4. Linear Transform (product of a matrix by a vector):

input $(x_1, \dots, x_p)$ with $0 \leq x_i < 2^k$;
 output $(z_1, \dots, z_p)$ with $z_i = \sum_j m_{i,j} \cdot x_j$ for some fixed matrix $[m_{i,j}]$,
 with integer coefficients $0 \leq m_{i,j} < 2^k$;
 size $N = p(2k+\log_2 p)$.

5. Matrix Product:

input $(m_{1,1}, \dots, m_{p,p}, m'_{1,1}, \dots, m'_{p,p})$ with $0 \leq m_{r,s}, m'_{r,s} < 2^k$;
 output $(z_{1,1}, \dots, z_{p,p})$ with $z_{r,s} = \sum_j m_{r,j} \cdot m'_{j,s}$;
 size $N = p^2(2k+\log_2 p)$. ■

The purpose of this paper is to:

Show that all problems in our *problem list*, and many related ones, share a common simple mathematical property which we call *transitivity* (Section 2).

Discuss and slightly refine the general yet realistic *computational model*, introduced by MEAD & CONWAY [1], BRENT & KUNG [2], THOMPSON [3] for VLSI circuits (Section 3).

Prove a general *theorem* stating that, for any circuit solving a transitive problem of size N , area A and data rate D are related by $A \geq a_g \cdot N + a_{I/O} \cdot D + a_w \cdot D^2$ where $a_g, a_{I/O}, a_w$ are technology dependent constants, respectively related to the physical sizes of gates, Input/Output ports and wires (Section 4). As we go along, we sketch the design of explicit circuits matching these bounds to within a (small) constant factor.

Some special cases of this *theorem* (usually expressed under the weaker form $A \cdot T^{2\alpha} \geq a \cdot N^{1+\alpha}$, for $0 \leq \alpha \leq 1$) have previously been observed - cyclic shifts in [4], integer multiplication in [2], matrix product in [5].

The main contribution of this paper lies in a generalization as well as strengthening of these earlier results. Interestingly enough, this is achieved by simplifying rather than complicating the mathematical techniques and concepts involved.

On a more general level, we expect here to contribute to the quickly emerging *mathematical theory of VLSI circuits* to which [1], [2] and [3] (among many others) have made pioneering contributions.

Indeed, circuits having area $A = O(a_g \cdot N + a_{I/O} \cdot D + a_w \cdot D^2)$ have been shown to exist for all functions in the above *problem list*: in [6] and [7] for matrix products, and in [8] for cyclic shifts, integer and polynomial products.

This rather pleasant situation should be contrasted with that regarding more widely studied computational models, such as *Random Access* or *Turing Machines*, where it appears to be exceedingly difficult to obtain matching upper and lower complexity bounds for any problem in the above list.

2. TRANSITIVE BOOLEAN FUNCTIONS

Although they are mathematically quite different, all functions in the above problem list reduce, when we specialize some of their arguments to specific values, to a common computational problem. Namely, in these special cases, they still retain the ability to permute the remaining unspecialized input data in a rich enough manner, which we call *transitive*. The present paragraph provides an abstract mathematical statement and proof of this simple idea. To keep his intuition alive, the reader is invited to particularize the definitions and theorems to the simplest problem mentioned above, namely *cyclic shifts*.

Let $G \subseteq S_N$ be a permutation group of degree N ; for any $g \in G$, we note $g(1), \dots, g(N)$ the associated permutation. The order of G is denoted by $|G|$. Note that $\log_2(|G|)$ bits are sufficient to encode each of G 's elements and, in the following definition, we assume that $b \geq \log_2(|G|)$.

DEFINITION. A boolean function $f(x_1, \dots, x_N, s_1, \dots, s_b)$ computes group G if, for any group element $g \in G$, there exist binary values of $s_1, \dots, s_b$ for which function f is permuting $x_1, \dots, x_N$ according to g , i.e.,

$$\forall g \in G, \exists s_1, \dots, s_b \in \{0,1\}^b$$

such that

$$\forall x_1, \dots, x_N \in \{0,1\}^N: f(x_1, \dots, x_N, s_1, \dots, s_b) = (x_{g(1)}, \dots, x_{g(N)}).$$

Problems in the above list all compute permutation groups, with the interesting property that any input bit can be mapped into any output position. To make this statement precise, we say that a permutation group is *transitive* if, for each pair i, j of index positions $1 \leq i, j \leq N$, there

exists a group element $g \in G$ which maps i into $g(i) = j$.

DEFINITION. A boolean function $f(x_1, \dots, x_N, s_1, \dots, s_b) = (z_1, \dots, z_N)$ is *transitive of degree N* if it computes a transitive permutation group acting on N elements.

2.1. Transitivity of functions in the problem list

Note that the *degree of transitivity* of function f indicates the number of bits which are permuted; some of the other input bits are used to specify (encode) the specific permutation applied. The remaining input and output bits are merely ignored for the purpose of this definition. This is clarified by checking the transitivity of specific examples:

PROPOSITION 1. *Cyclic Shifts, Integer Product, Convolution and Linear Transform are transitive problems of respective degrees: $p, p/2, (p+1) \cdot k/2, p \cdot k/2$. Computing the Product of Three Matrices is a transitive problem of degree $p^2 \cdot k/2$.*

PROOF. The result is obvious for cyclic shifts. This implies that shifting n bits $X = (x_0, \dots, x_{n-1})$ by s positions, $0 \leq s < n$, yielding

$$Z = (z_0, \dots, z_{2n-1})$$

with

$$[z_i = 0 \text{ iff } (i < s \text{ or } i \geq s+n) \text{ and } z_i = x_{i-s} \text{ otherwise}]$$

is also a transitive problem of degree $n/2$. Indeed, ordinary shift reduces to cyclic shift when applied to inputs X of the special form

$(x_0, \dots, x_{n/2-1}, x_0, \dots, x_{n/2-1})$, and shifts s such that $0 \leq s \leq n/2$; in the output Z , we only retrain bits $n/2$ through $n-1$, i.e., $z' = (z_{n/2}, \dots, z_{n-1})$. $\square$

Integer multiplication $X \cdot Y$ reduces to (ordinary) shifting if we choose $Y = 2^s$ with $0 \leq s < p$ which means that all the bits in Y are set to 0 except for $y_s = 1$. Such a multiplication is merely shifting $x_0, \dots, x_{p-1}$ by s positions. An equivalent way to look at this is to consider the product $Z = X \cdot Y \bmod 2^p - 1$, in which Y is a power of 2.

Using the same trick as above, we first reduce convolution to *cyclic convolution*, i.e., compute the product $Z(t) = X(t) \cdot Y(t) \bmod t^{p+1} - 1$. Choosing

all y_i 's to be 0 except y_s results in applying a cyclic permutation (by s positions) to the x_i 's, each being also multiplied by y_s . To shift the bits of x_i , we choose y_s to be a power of 2, as in the integer multiplication above, and constrain the x_i 's to be of the form $x' + 2^{k/2}x'$, thus effectively computing mod $2^{k/2}-1$. Mathematically, the effect of this product is to cyclically shift the coefficients of $X(t)$ and simultaneously to cyclically shift the individual bits of each coefficient; the group computed by such a boolean function is the product $C_{p+1} \times C_{k/2}$ of two cyclic groups, which is indeed transitive of degree $(p+1) \cdot k/2$.

A linear transform in which matrix $[m_{ij}]$ is a permutation matrix obviously permutes the input $x_1, \dots, x_p$ in a transitive manner, the underlying group being the symmetric group S_p ; to further shift the individual bits of x_i , we use the now familiar method of multiplying coefficients by a power of 2 mod $2^{k/2}-1$. The resulting group $C_{k/2} \times S_p$, product of a cyclic group by the symmetric group is transitive of degree $p \cdot k/2$.

As for matrix product, the problem is not obviously transitive: if we choose either of the two operands to be a permutation matrix, the global effect will be to permute lines or columns rather than the coefficients themselves. We return to this particular problem in Section 4, showing that matrix product has enough of the transitive features for our purposes. For the time being, changing slightly the problem and computing the product of three rather than two matrices simplifies matters: in the matrix product $P \cdot M \cdot Q$, choose both P and Q to be permutation matrices; such products can now map any coefficient of M into any location of the resulting matrix; to further shift the individual bits of each coefficient, we multiply as above by a power of 2 mod $2^{k/2}$. This argument shows that computing $P \cdot M \cdot Q$ is indeed a transitive problem of degree $p^2 \cdot k/2$. ■

Note that in all four cases, the degree of transitivity is greater than $N/8$, where N is the problem size measured in number of output bits, provided that the number of bits used for encoding the coefficients is large enough with respect to the number of such coefficients, namely $k \geq \log_2 p$. The theorem we shall prove in Section 4 holds under the weak hypothesis that a boolean function having N output bits is transitive of degree N/c , for some constant c (in our case, $c = 8$).

2.2. On the extent of the class of transitive functions

It should be clear by now that transitivity holds true of a wide class of computational problems.

Among the problems mentioned above, we can particularize: computing *modular products*, using special forms of the arguments, etc...; in particular, computing the product of three matrices remains transitive, even if we insist that some of the operands be *circulants*, *symmetric matrices*, ..., thus answering a question of SAVAGE [5].

In the other direction, Proposition 1 implies directly the transitivity of problems which are known to be *harder*, in that they have some degenerate case in our above list.

This holds true of all floating point operations, which, in their normalization stage, require shifting the mantissa. It is obviously also true of functions computing richer transitive classes of input permutations; computing such permutations is a crucial part in the design of switching networks or parallel machines interconnections.

Similarly, a list of arithmetical problems known to be harder than Integer Multiplication can be found in [9].

Signal and image processing also provide a wealth of problems, with great practical significance, which all reduce to computing convolutions (see [13] for example).

Finally, Linear Algebra also abounds in problems closely related to Matrix Multiplication or Linear Transform (see [9] again), and the VLSI complexity of some of these has already been studied in [6] or [10] for example.

3. THE COMPUTATIONAL MODEL

We adopt here a computational model for VLSI circuits similar to the one proposed in [2]. Any such model must inherently be a gross simplification of the present rather complex circuit design and fabrication process, as described (and greatly simplified) for example by MEAD & CONWAY [1].

We nevertheless see two main reasons for studying such models: firstly, they appear to have inherent *theoretical interest*; secondly, they provide a *high level of abstraction* at which to start the design of VLSI circuits, thus allowing us to think in algorithmic rather than electrical terms, and we observe that very efficient practical circuits can be designed using

this approach.

3.1. Technological assumptions ^{*)}

A circuit is a collection of wires. Such wires are connected together by gates; they are also connected to the outside world by I/O ports.

Gates are used to switch on or off certain wire connections. In n-MOS technology, this transistor effect of gates is merely obtained by running the *controlling* wire on the polysilicon level and the *controlled* wire connection on the diffusion level (see [1]) as shown in Fig. 3.1.

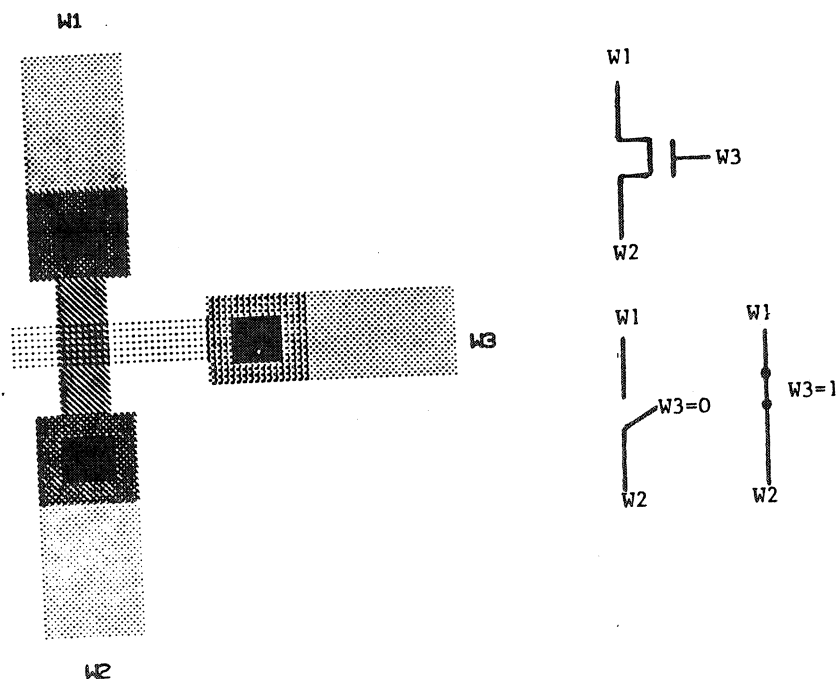


Fig. 3.1. An n-MOS gate. Scale = 1 μ m.

If we think of our circuit as constituting an entire chip (see Fig. 3.2), I/O ports are the *pads* of the chip, used for bonding off-chip connections. If we think of our circuit as merely being part of a larger circuit designed within the same technology, then I/O ports are merely *loose wire connections*, awaiting to be hooked onto the same type of wire.

^{*)} See also the contribution of M REM in this volume

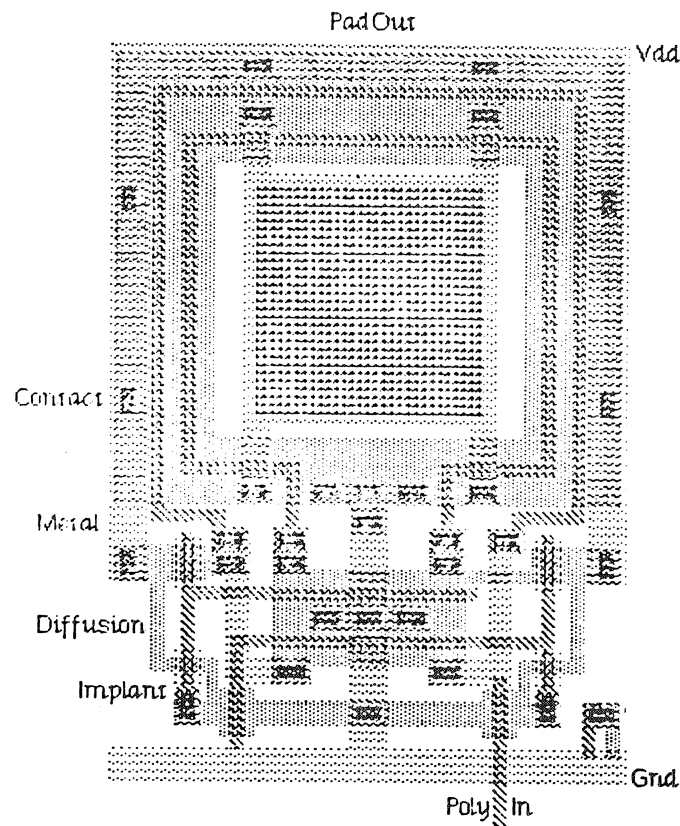


Fig. 3.2. An n-MOS input pad (scale ...)

The most important underlying technological assumption made here is that a circuit is *planar*, meaning that its constituting elements are laid side by side rather than on top of each other. Actually, present technologies allow to superpose a small number of planar levels; a typical n-MOS process has three wire levels: metal, diffusion and polysilicon; in *printed circuit* technology, the number of wire levels is higher (say 10). Such considerations only affects our arguments by a constant multiplicative factor v , representing the number of layers.

Because of planarity, we may assume w.l.o.g. that I/O ports are located at the periphery of the chip: should this not be the case, say with a pad located in the middle of the chip, we can conceptually include in our

circuit the portion of wire bonded to this particular pad and located within the chip's periphery (thus pretending that the number of wire levels is now $v+1$).

3.1.1. Size of elementary devices

The physical size of circuits is dependant upon the basic process resolution λ (in current n-MOS, $\lambda = 2 \sim 3$ microns) as defined in [1].

The distance between the middle of two parallel wires must be at least $\lambda_w = c_1 \times \lambda$ (typical $c_1 = 4 \sim 6$). Wires may only cross on different layers.

Each gate occupies some minimal area $a_g = (c_2 \times \lambda)^2$ (typical $c_2 = 10 \sim 20$).

I/O ports occupy an area $a_{i/o}$. If I/O means going off-chip,

$a_{i/o} = (c_3 \times \lambda)^2$ is typically greater than a_g ($c_3 = 50 \sim 150$).

3.1.2. Speed of elementary devices

As described in [1], all of the elementary devices transitions require a time which is proportional to the transit time τ of the basic process (in current n-MOS, $\tau = 2 \sim 4 \cdot 10^{-1}$ nanosec.).

Transmitting a signal along a wire, i.e. charging or discharging this wire, requires a minimum delay $\tau_w = c_4 \times \tau$ (typical $c_4 = 1 \sim 10$). This time grows with the wire size.

A minimal size gate controlling another minimal size gate switches in time $\tau_g = c_5 \times \tau$ (typical $c_5 = 1 \sim 10$). For larger gates, time is proportional to the gate and controlled wire size.

Each I/O port introduces a minimal delay $\tau_{i/o} = c_6 \times \tau$ (typical $c_6 = 10 \sim 100$ if I/O is off-chip) between passage of two successive bits of information (signals). The ratio $\tau_{i/o}/\tau_w$ is the I/O penalty (typically 10) required for off-chip signal amplification (I/O pads have capacitance which is much larger than that of minimum size wire).

3.2. Logical assumptions

The main assumption made here is that computation within the circuit is performed in a *digital* rather than *analog* manner. It is certainly not obvious how to give a precise technical meaning to the preceding sentence, and some of the issues involved are discussed in [1, ch.9]. For our purposes it simply means that signals representing 0 or 1 enter through Input gates, are processed as such through the circuit, and eventually flow

out through the Output ports. The convention as to which *physical parameter* is used for representing 0 and 1 must be consistent throughout the computation; one *acceptable* convention is to represent logical 0 by a potential less than 1.0 Volt, and 1 by a potential greater than 4.0 Volts; one *un-acceptable* convention is to represent numbers by current intensity proportional to their values, so that addition in particular becomes very simple. This is not to say that the latter idea is not feasible or interesting, but simply that the results in Section 4 would no longer apply. Once this is granted, the *state* of a circuit is defined by the values (0 or 1) of its constituting wires at a particular *time*. Any given circuit has finitely many states (although this number can be very large).

We must also assume that the *times* and *locations* at which the input and output bits are available on the I/O gates are fixed once and for all, and are thus *independent* of the *values* of the inputs. Loosely speaking, these represent the circuit's I/O *format*. Furthermore, each input bit is *available exactly once*, so that we have no *free memory* outside the circuit. It is irrelevant here to talk about *absolute* time (as specified by an external clock) or *relative* time, determined by the instants of the state transitions in the circuit (thus effectively driven by an internal clocking mechanism).

3.3. Global circuit measures

Performances of a specific circuit computing boolean function f are measured with respect to:

The global area A occupied by the circuit, measured in square microns. It is constituted of gates, wires, I/O ports and *empty space* which typically appear in layouts which do not tightly pack the other elements.

The computation time T of the circuit measured in nanoseconds. It measures the amount of time separating the appearance of the first input bit on some input port, from that of the last output bit on some output port.

Equally important (both from a theoretical and a practical point of view) is the maximum data rate D , expressed in bit/second. It measures the number of bits which the circuit is able to process, when it repeatedly computes function f on successive inputs, using the maximum achievable amount of *pipelining*. When necessary, we distinguish between the *input data rate* D_{in} and the *output data rate* D_{out} , which need not necessarily be the

same.

If N_{in} and N_{out} are respectively the numbers of input and output bits for f , we define $P = (N_{in}/D_{in}) = (N_{out}/D_{out})$, the period of the circuit. It measures the minimal time interval separating the first input (resp. last output) signals for two successive (pipelined) computations of f .

To emphasize the difference between time T and period P , note that P can be as small as $\tau_{i/o}$, provided that the circuit has N_{in} and N_{out} input and output ports, and that each of the successive elementary computation stages can be *completely pipelined*; on the other hand, as soon as some specific output bit is effectively dependant upon all input bits of f , computation time must be at least $T \geq 2\tau_{i/o} + \tau_g \log_2(N_{in})$, based on using a tree of gates to combine the n inputs to form the output.

It is obvious that the following inequality holds true of any circuit:

$$\tau_{i/o} \leq P \leq T.$$

4. AREA VERSUS DATA RATE

In this section, we successively analyze the contributions of *I/O ports*, *gates* and *wires* to the global area A of a circuit computing a boolean function f at data rate D . To simplify notation, we assume from now on that $N_{in} = N_{out} = N$ and $D_{in} = D_{out} = D$.

4.1. On the number of I/O ports

Each *I/O port* has maximum data rate $1/\tau_{i/o}$; to achieve a global data rate D , at least $D \times \tau_{i/o}$ Input and Output ports are required. It follows that:

PROPOSITION 2. A circuit computing any function f at data rate D has I/O area at least $A_{I/O} \geq 2D \cdot (a_{i/o} \times \tau_{i/o})$.

This result does not depend upon the *transitivity* of the function computed by the circuit.

To check that it cannot be improved without introducing further hypothesis, the reader is invited to design binary *adders* having an area proportional to their data rate, for all values $O(1) \leq D \leq O(N)$; this can be achieved by simply combining *serial adders* (fig. 4.1) $A = O(1)$, $D = O(1)$, $T = O(N)$ and *carry propagate adders* (fig. 4.2) $A = O(N)$, $D = O(N)$, $T = O(N)$. The latter adder is *slow* even though it has *high* data rate. Using more

sophisticated *carry look ahead adders* in a suitable manner (see [2]), reduces time to $T = O(\log_2 N)$, while increasing the area to $A = O(N \log_2 N) = O(D \log_2 D)$. It is not clear that a *fast adder* $T = O(\log_2 N)$ can be designed which retains $A = O(D)$.

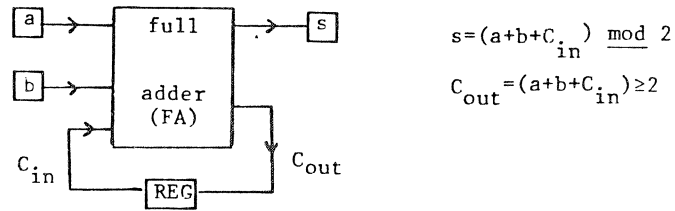


fig. 4.1. A serial adder with 3 I/O ports, and data rate $d = 1/\tau_{i/o}$

To achieve data rate $D = N/\tau_{i/o}$, i.e. period $P = \tau_{i/o}$, inputs (a_i, b_i) must be available when c_i has been computed, so that only adder A_i is active for that particular addition; the other A_j 's are thus available for working on different addition operands.

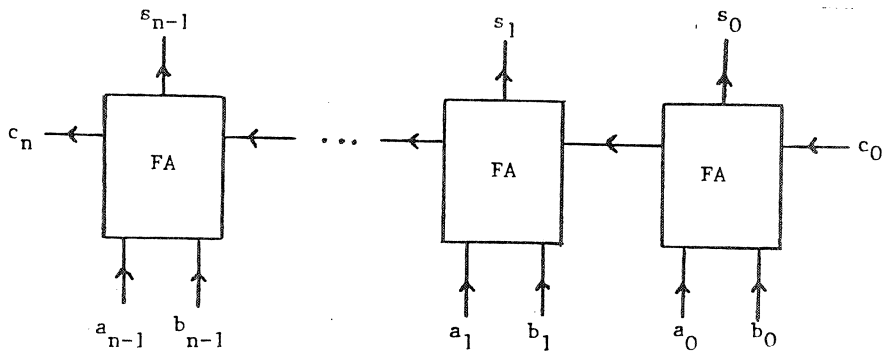


fig. 4.2. A carry propagate adder

4.2. On the number of gates

PROPOSITION 3. Any circuit computing a transitive function f of degree N has gate area at least $A_g \geq N \cdot a_g$.

PROOF. Consider the bit position (among the N which f has the ability to permute) which is *first* to get out of the circuit, according to the input format. Function f being *transitive*, we can specify some of its input parameters (other than those counted in the degree N) so that the *last input bit* is the *first one* to get out. In this situation, f merely permutes N bits and, since no output can be initiated before the last input bit is read in, our circuit must possess 2^N different internal states. In other words, it has the capability of *memorizing* N bits which requires at least N active gates, thus area $N \cdot a_g$. ■

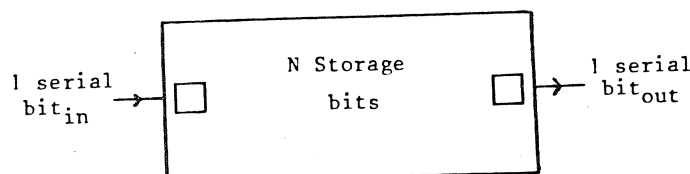


fig. 4.3. A small $A = O(N)$ and slow $D = O(1)$ (large $P = O(N)$, $T = O(N)$) circuit

A simple shift register is described in [1], plate 9, whose area is $N \times (m+c)$ where m is the area used for memorizing one bit, and c is the area of the gate controlling a one bit shift. Our bound is thus tight to within a (small) constant factor.

As a corollary of Proposition 3, we see that multiplying two N bits integers requires at least N memory cells, a result already obtained in [2] through a non elementary number theoretic argument. Note that the well known *shift & add* multiplier is classically implemented with $O(N)$ gates (see [13], pp. 239-246).

4.3. On the area occupied by wires

In the beginning of this section, we need to state a result of BRENT and KUNG [2], establishing an interesting way to cut a given circuit in two roughly equal parts (their result refines an earlier technique of THOMPSON [3]).

Consider a circuit computing some boolean function f . Without (serious) loss of generality, we can assume that the circuit is shaped as a *rectangle*

of area $A = X \times Y$, whose larger side X is horizontal (fig. 4.4). For the sake of this argument, we only consider the part A_w of this region constituted of wires, defined by conceptually shrinking I/O ports and gates to mathematical points having a null area. It proves convenient to assume that I/O port are shrunk into points which all have different x coordinates.

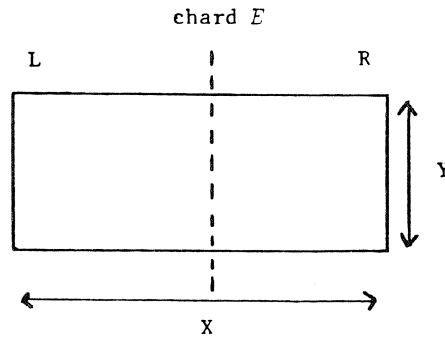


fig. 4.4. Circuit A.

By sliding a vertical line across A , we can find at least one vertical chord C which partitions the output ports in two disjoint sets L and R through which approximately the same number of output bits flow. Indeed, through any given I/O port of a circuit achieving data rate D , at most $N/D \cdot \tau_{i/o}$ bits of the computed function can flow. It follows that we can find a chord C for which the number L_{out} and R_{out} of bits passing respectively through L and R satisfy $L_{out} \geq R_{out} \geq N(1/2 - 1/D \cdot \tau_{i/o})$.

Let ω be the number of circuit wires crossing C . Since wires are spaced at least λ_w apart on each of the V wire levels, the length Y of C must exceed $\omega \times \lambda_w / V$. Since $A = X \times Y$ and $X \leq Y$, it follows that $A \geq c \cdot \omega^2$ for $c = (\lambda_w / V)^2$.

To summarize, BRENT and KUNG [2] prove a result which can be expressed as follows:

LEMMA 1. *There exists a line C cutting ω wires and partitioning the circuit in two parts L and R such that: $A_w \geq c \cdot \omega^2$ and $L_{out} \geq R_{out} \geq N(1/2 - 1/D \cdot \tau_{i/o})$, where D is the circuit data rate.*

We now demonstrate that, in order to achieve a given data rate $D > 2/\tau_{i/o}$, a circuit computing a transitive function must have many wires.

Intuitively, these wires need to be there in order to allow data entering on one side to be broadcast quickly to the other side of the circuit.

LEMMA 2. *If the function computed by a circuit as in Lemma 1 is transitive, its data rate D is bounded by $D \leq 2(\omega + \tau_w / \tau_{i/o}) / \tau_w$.*

PROOF. Assume that f computes the permutation $f(x_1, \dots, x_N) = (x_{g(1)}, \dots, x_{g(N)})$ for $g \in G$, an element of the transitive group G . We say that bit x_i crosses the line C constructed in Lemma 1 if x_i enters an input port in L (resp. R) while $x_{g(i)}$ exits through an output port in R (resp. L).

Let us count the total number of crossings, summed over the N bit positions $(x_1, \dots, x_N)$ and all $|G|$ group elements $g \in G$. Since G is transitive, elementary group theory teaches us that the sets $G_{ij} = \{g: g(i) = j\}$ all have the same size $|G_{ij}| = |G|/N = S$, with $|G| = N \cdot S$. It follows that each input bit is mapped exactly S times into each output position. If it was assigned (according to the I/O format) an input port in L , it contributes $S \cdot R_{out}$ crossings, and $S \cdot L_{out}$ otherwise.

The total number of crossings, during the computation of $|G|$ permutations, is thus greater than $N \cdot S \cdot \min(R_{out}, L_{out}) = |G| \cdot R_{out}$. By the pigeon hole principle, there must therefore be at least one specific permutation, say g' , which accounts for R_{out} crossings. In the course of the computation $(x_1, \dots, x_N) \rightarrow (x_{g'(1)}, \dots, x_{g'(N)})$, at least R_{out} binary values (encoding $2^{R_{out}}$ possible different states of the circuit) must necessarily flow through the ω wires cut by line C . This data movement requires time at least $\tau_w \cdot R_{out} / \omega$, thus the circuit global data rate cannot exceed $D \leq N \cdot \omega / (\tau_w \cdot R_{out})$. Since by Lemma 1, $R_{out} \geq N(1/2 - 1/D \cdot \tau_{i/o})$ we get $\omega \geq \tau_w (D/2 - 1/\tau_{i/o})$. Note the interesting appearance of the I/O penalty $\tau_w / \tau_{i/o}$ in this formula. ■

Combining Lemma 1 and 2 yields, by an elementary case analysis:

PROPOSITION 4. *Any circuit computing a transitive function f at data rate D has wire area at least $A_w \geq a_w \cdot D^2$ for some appropriate constant $a_w > 0$.*

Proposition 4 is not expressed here in its strongest mathematical form. The notion of transitive function which we use here for its conceptual simplicity has been used by [14] in a completely different context. It serves the purpose here to make the counting argument in the above proof clean and simple. It is possible to relax such an hypothesis quite a bit at the price of more complex counting arguments.

In itself, Proposition 4 is expressing what seems like a very weak result: the area needed for computing a transitive function at data rate $D = N/P$ is proportionnal to $a_w \cdot D^2$, i.e. the area of D input wires bending 90° to yield D output wires.

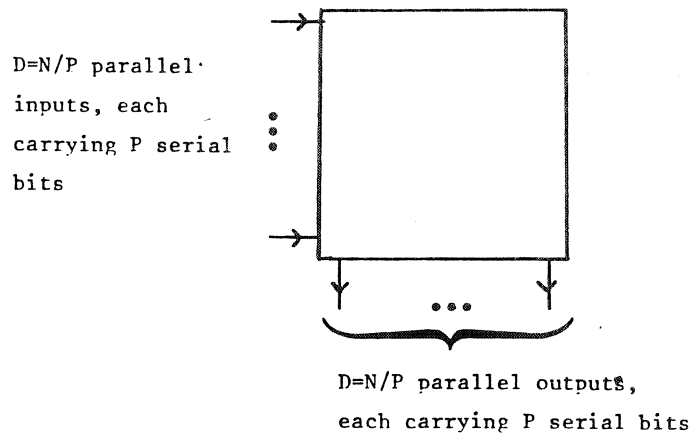
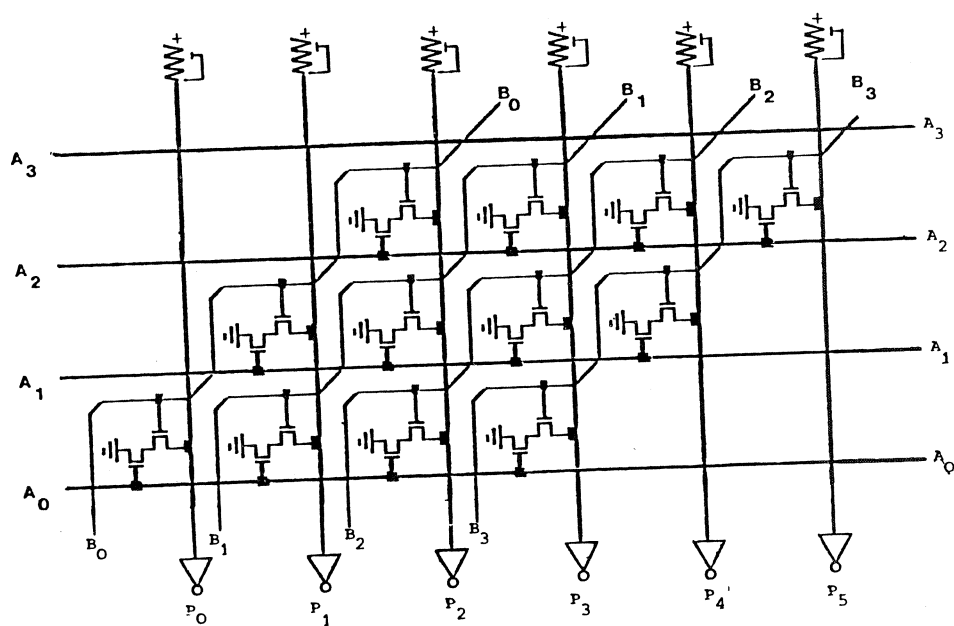


fig. 4.5.

Can anything interesting be computed within such a constraining bound? Surprisingly enough, the answer is yes, and fig. 4.6 provides one example of a non-trivial circuit, (due to C. Thacker) computing shifts over M inputs in area M^2 . This circuit being purely combinatorial (no loops), it can be pipelined in a straightforward fashion, in order to compute successive shifts at data rate $D = O(N^2/P^2)$, where P is the time for one elementary computation.

Thacker's and-or Convolver.



$$p_0 = a_0 b_0 ; p_1 = a_0 b_1 + a_1 b_0 ; p_2 = a_0 b_2 + a_1 b_1 + a_2 b_0 ; \dots$$

fig. 4.6. This circuit for computing and-or convolution reduces to a shifting the a_i 's when exactly one of the b_i 's is non-zero.

4.4. On the total circuit area

Since the circuit is assumed to be planar (except for the wires), its total area is greater than $(A_{I/O} + A_g + A_w)$, and we can conclude from propositions 2, 3 and 4 that:

THEOREM. Any circuit computing a transitive function of degree N at data rate D occupies an area A which is at least $A \geq a_g \cdot N + a_{I/O} \cdot D + a_w \cdot D^2$, where $a_g, a_{I/O}, a_w$ are some positive constants dependent upon the technology.

As a corollary of Proposition 1, the above Theorem directly applies to Cyclic Shifts, Integer Product, Convolution, Linear Transform and computing the Product of Three Matrices.

By refining the counting arguments used implicitly in proving Proposition 3 and 4, one can extend this result to Matrix Product, Merge, Sort and specific instances of Linear Transform, such as the Discrete Fourier, Fermat or Mersenne Transforms ([5],[3],[15],...).

5. CONCLUSION

5.1. Relations between the VLSI and other models of computation

As mentioned in the introduction, we now have matching (to within a constant factor) upper and lower complexity bounds for a variety of computational problems in the VLSI model. For Turing Machines or RAMs, no such result is known for any nonlinear time problem. Why?

Traditionally, boolean circuit complexity has been expressed with respect to time and the number of gates. These in turn are related to the time/space measures of Turing and other Machines (see [12] for example).

The only novelty in the VLSI model is accounting for the wire area. Since any circuit having N gates can be laid out in a straightforward manner within area $O(N^2)$ -used for wires-, our $A = \Omega(N+D^2)$ lower bound does not imply that the circuit has a non-linear number of gates (a $A = \Omega(D^3)$ lower bound for any specific problem would have non trivial implications).

To conclude, one cannot expect the VLSI model to provide new results regarding, say Turing Machines, unless someone is clever enough to solve what we know are difficult problems, such as $P = ?NP$.

On the other hand, it raises an interesting question: which computational problems are in the $A = O(N^2)$ class?

Can they be characterized in a mathematically satisfactory manner? It is not even clear that the class is closed under composition (if you do not believe that, try and connect two arbitrary circuits in a way which makes

the outputs of the first one match the inputs of the second at the *proper* times and locations). Using results from [8], one can prove that the class $A = O(N \log N + D^2)$ is closed under composition. Since it contains basic problems (multiply, merge,...) of great interest for Computer Science, such a complexity class deserves a great deal of attention (the same question can be raised for the $A = O(N)$ class, which contains addition).

5.2. On detailed mathematical analysis of VLSI circuits

Although *optimal circuits* (to within a constant factor) have been shown to exist for multiplying integers, polynomials and matrices, it is not yet clear that they are of practical interest. They can only claim to be asymptotically better than the straightforward implementations commonly used. Where the cutoff point really is remains to be determined by a very detailed design and analysis of such circuits.

There is a clear need here to develop a theory for analysing algorithms for VLSI circuits; eventually it must attain the level of mathematical sophistication which KNUTH [11] and others use in the analysis of sequential algorithms.

5.3. On the computational model

The lower bounds derived in the computational model of section 3 assume that the delay introduced by signal propagation along the wires is independent of the length of these wires. Designing circuits which match these entails a careful accounting of the effect of wire lengths and power limitations. Whether or not one can still retain matching upper and lower complexity bounds in this case is unclear.

Acknowledgements

I have greatly benefited from many discussions with Gérard Baudet (who contributed Proposition 3), and with Franco Preparata. Many thanks to Dick Lyon for his comments and careful reading of this paper.

6. REFERENCES

1. MEAD, C. & L. CONWAY, *Introduction to VLSI Systems*, Addison-Wesley, 1980.
2. BRENT, R.P. & H.T. KUNG, *The Chip Complexity of Binary Arithmetic*, proc. 12th Annual ACM Symposium on Theory of Computing, ACM, pp. 190-200, May 1980.
3. THOMPSON, C.D., *Area-Time Complexity for VLSI*, proc. 11th Annual ACM Symposium on Theory of Computing, ACM, pp. 81-88, May 1979.
4. ABELSON, H. & P. ANDREA, *Information Transfer and Area-Time Tradeoffs for VLSI Multiplication*, CACM, vol. 23, pp. 20-23, 1980.
5. SAVAGE, J.E., *Area-Time Tradeoffs for Matrix Multiplication and Related Problems in VLSI Models*, proc. 17th Annual Allerton Conference on Communications, Control and Computing, 1979.
6. KUNG, H.T. & C.E. LEICERSON, *Systolic Arrays for VLSI*, appeared in [1], Addison-Wesley, pp. 260-292, 1980.
7. PREPARATA, F.P. & J.E. VUILLEMIN, *Area-Time Optimal VLSI Networks for Multiplying Matrices*, proc. 14th Princeton Conference on Information Sciences and Systems, 1980.
8. PREPARATA, F.P. & J.E. VUILLEMIN, *Area-Time optimal VLSI Networks Based on the Cube Connected Cycles*, Rapport INRIA #13, Rocquencourt France, 1980.
9. AHO, A., J. HOPCROFT, J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
10. GUIBAS, L.J., H.T. KUNG, C.D. THOMPSON, *Direct VLSI Implementations of Combinatorial Algorithms*, proc. Conference on VLSI Architecture Design and Fabrication, CALTECH, 1979.
11. KNUTH, D.E., *The Art of Computer Programming*, vol. 1, 2, 3, ..., Addison-Wesley, 1973.
12. RUSSO, W.L., *On Uniform Circuit Complexity*, proc. 20-th Annual Symposium on Foundations of Computer Science.
13. SALAZAR, A.C., (ed.), *Digital Signal Computers & Processors*, IEEE Press, 1977.

14. RIVEST, R.L., J.E. VUILLEMIN, in preparation.

15. HEINTZ, *Thèse de 3^o Cycle*, to appear in 1981.

BOOMMACHINES EN VERKAVELDE BEREKENINGEN

F.J. PETERS

0. INLEIDING

Indien een berekening met één rekenmachine te veel tijd vergt, kan men proberen de rekentijd te reduceren door de berekening te verkavelen over een aantal machines. Dan is het noodzakelijk dat deze machines, die samen één karwei moeten klaren, onderling verbonden zijn om informatie te kunnen uitwisselen. Als het om een groot aantal machines gaat is het technisch niet goed mogelijk om iedere machine met alle andere te verbinden. Een machine kan nu eenmaal slechts verbonden worden met een beperkt aantal andere.

Een boommachine (BENTLEY & KUNG [1979], BROWNING [1980]) is zo'n conglomeraat van machines, die onderling zo verbonden zijn dat het geheel een binaire boom vormt. Iedere machine is dan met hooguit drie andere verbonden, te weten één voorganger en twee opvolgers. Boommachines lijken bij uitstek geschikt voor algoritmen van het verdeel-en-heers type: een probleem wordt in twee deelproblemen verdeeld, vervolgens worden beide deelproblemen opgelost en tenslotte worden de oplossingen van de deelproblemen gecombineerd tot een oplossing van het oorspronkelijke probleem.

Meestal doet de volgorde van de deelproblemen er niet toe en dus kunnen die deelproblemen tegelijkertijd worden opgelost. Beide deelproblemen kunnen elk afzonderlijk worden uitbesteed aan de opvolgers van de machine die belast is met het oorspronkelijke probleem.

We zullen laten zien hoe verdeel-en-heers algoritmen op een boom geïmplementeerd kunnen worden. Daarbij zullen we de notatie gebruiken zoals die door HOARE [1978] is geïntroduceerd om communicatie tussen processoren te specificeren. De syntaxregels beschreven in BNF^{*)} zijn de volgende:

*) Accolades "{...}" moeten als volgt gelezen worden: "nul of meer keer wat er tussen de accolades staat".

```

<invoeropdracht> ::= <bron>?<variable>{,<variabele>}
<uitvoeropdracht> ::= <doel>!<expressie>{,<expressie>}
<bron>                ::= <machinenaam>
<doel>                ::= <machinenaam>

```

Er vindt communicatie tussen twee machines plaats als

- 1) een invoeropdracht voor de ene machine als bron de naam van de andere machine bevat,
- 2) een uitvoeropdracht voor die andere machine als doel de naam van de eerste machine bevat,
- 3) de lijst van variabelen van de invoeropdracht past bij de lijst van expressies van de uitvoeropdracht.

Een boommachine is ook heel geschikt voor allerlei zoekopdrachten (BENTLEY & KUNG [1979]). Neem aan dat er getallen zijn opgeslagen in de bladeren van de boommachine. De volgende vragen kunnen dan snel beantwoord worden: "Welk getal ligt het dichtst bij een gegeven ander getal?", "Wat is het rangnummer van een gegeven getal?". De antwoorden worden als volgt bepaald (KUNG [1980]):

- i) het gegeven getal wordt van de wortel naar de bladeren gestuurd;
- ii) in elk blad wordt het getal vergeleken met het in dat blad opgeslagen getal;
- iii) het resultaat van die vergelijking wordt naar de voorganger - zeg PP - gestuurd, waar het wordt gecombineerd met het antwoord van de andere opvolger van PP; het resultaat van die combinatie wordt naar de voorganger van PP gestuurd, enz.;
- iv) het proces is beëindigd op het moment dat de wortel de antwoorden van zijn twee opvolgers gecombineerd heeft.

Het zal duidelijk zijn dat op deze manier zoekopdrachten in een logaritmische hoeveelheid tijd worden uitgevoerd. Behalve programma's voor boommachines zullen we in deze bijdrage ook formules afleiden die laten zien welke tijdwinsten er geboekt kunnen worden wanneer we boommachines in plaats van "gewone" sequentiële machines gebruiken.

1. DOELMATIGHEID VAN BOOMMACHINES

Beschouw een recursief sequentieel algoritme van de volgende vorm:

```

1. program S; var e1,...,en, v1,...,vm;
2. procedure p (in e1,...,en; out v1,...,vm);
3. var l1,...,lk, v1',...,vm'; v1'',...,vm'';
4. begin partitioneer;
5.     if BE1 then p(e1',...,en'; v1',...,vm') fi;
6.     if BE2 then p(e1'',...,en''; v1'',...,vm'') fi;
7.     combineer
8. end;
9. begin ...
10.    p(e1,...,en; v1,...,vm)
11.    ...
12. end

```

In het bovenstaande programma onderscheiden we twee soorten parameters: bij procedure entry hoort er bij iedere parameter van het type in een waarde; parameters van het type out hebben een waarde gekregen op het moment van procedure exit. In regel 4 is partitioneer een (mogelijkerwijs lege) rij statements om o.a. de waarden te berekenen van de in parameters voor de recursieve aanroepen in de regels 5 en 6. In regel 7 is combineer een rij statements om uit de out parameters v1',...,vm' en v1'',...,vm'' de waarden voor de out parameters v1,...,vm te berekenen. De in regel 3 gedeclareerde variabelen l1,...,lk zijn lokale variabelen om in de procedure body te gebruiken.

Een equivalent programma voor een boommachine kan als volgt gecodeerd worden. Als P een knoop is van de boommachine, dan moet P het volgende programma uitvoeren:

```

1. program T; var e1,...,en, v1,...,vm;
2.     processor PP, PS1, PS2;
3.     var l1,...,lk, v1',...,vm', v1'',...,vm'';
4. begin PP? e1,...,en;
5.     partitioneer;
6.     if BE1 then PS1! e1',...,en' fi;
7.     if BE2 then PS2! e1'',...,en'' fi;
8.     if BE1 then PS1? v1',...,vm' fi;

```

```

9.      if BE2 then PS2? v1",...,vm" fi;
10.     combineer;
11.     PP! v1,...,vm
12. end

```

In het bovenstaande staat PP voor de voorganger van P, behalve als P de wortel is; in dat geval duidt PP de omgeving aan. De machines PS1 en PS2 zijn de opvolgers van P; als P echter een bladmachine is dan zijn er geen opvolgers en moeten de regels 6-9 vervangen worden door aanroepen van de procedure p zoals die in program S gedeclareerd is:

```

6'.     if BE1 then p(e1',...,en'; v1',...,vm') fi;
7'.     if BE2 then p(e1",...,vn"; v1",...,vm") fi;

```

We zullen nu de rekentijden van program S en program T met elkaar vergelijken. Stel dat de omvang van het probleem dat moet worden opgelost, gekarakteriseerd kan worden met een geheel getal n. Neem verder aan dat de omvang deelproblemen die opgelost worden door de recursieve aanroepen in de regels 5 en 6 van S gekarakteriseerd wordt met n/2 (eenvoudigheids-halve gaan we er van uit dat n een twee-macht is). De tijd die nodig is om S uit te voeren zullen we aangeven met s(n). Als g(n) de hoeveelheid tijd is die nodig is om de partitioneer- en combineerstappen uit te voeren plus de tijd die de parameteroverdracht vergt, dan geldt de volgende recurrente betrekking:

$$(1) \quad s(n) = 2 s(n/2) + g(n).$$

Laat nu t(n) de hoeveelheid tijd zijn die nodig is om T uit te voeren. Na uitvoering van regel 7 moet machine P wachten tot PS1 en PS2 hun programma's beëindigd hebben; pas daarna kan P verder gaan met de uitvoering van regel 10. PS1 en PS2 voeren hun programma's tegelijkertijd uit. We hebben derhalve, als P geen bladmachine is, de volgende betrekking:

$$(2) \quad t(n) = t(n/2) + g(n).$$

Als $g(n) = \alpha n^p$ dan hebben (1) en (2) als oplossingen (PETERS [1980a], ROSE & WHITTEN [1976]):

$$(3) \quad \left. \begin{aligned} s(n) &= \frac{2^{p-1}}{2^{p-1} - 1} \alpha n^p + c_1 n \\ t(n) &= \frac{2^p}{2^p - 1} \alpha n^p + c_2 \end{aligned} \right\} \text{ voor } p \neq 0, p \neq 1,$$

$$(4) \quad \left. \begin{aligned} s(n) &= \alpha n \log_2 n + c_1 n \\ t(n) &= 2\alpha n + c_2 \end{aligned} \right\} \text{ voor } p = 1,$$

$$(5) \quad \left. \begin{aligned} s(n) &= -\alpha + c_1 n \\ t(n) &= \alpha \log_2 n + c_2 \end{aligned} \right\} \text{ voor } p = 0.$$

c_1 en c_2 zijn constanten die door randvoorwaarden bepaald worden; c_2 hangt onder meer af van de hoeveelheid werk die de bladmachines verrichten.

Als $p > 1$ dan volgt uit (3), (4) en (5) dat een boommachine ten opzichte van een sequentiële machine slechts een constante factor f in rekentijd bespaart; bovendien ligt die factor dicht bij 1. Als $p \geq 3$ dan geldt: $1 < f \leq \frac{7}{6}$. Het is dus niet erg zinvol om in die gevallen boommachines te gebruiken.

Als $p = 1$ dan bespaart een boommachine een factor $\frac{1}{2} \log_2 n$, terwijl als $0 < p < 1$ een factor $O(n^{1-p})$ bespaard wordt.

Het meest interessante geval echter is $p = 0$; dan wordt de rekentijd van $O(n)$ gereduceerd tot $O(\log n)$.

2. VOORBEELDEN

2.1. Recursive doubling

Recursive doubling is een bij ontwerpers van parallelle algoritmen bekende techniek om gegeven de objecten $a_1, \dots, a_n$ en een associatieve operator $\circ$ de waarde van $a_1 \circ a_2 \circ \dots \circ a_n$ op efficiënte wijze te bepalen. Bij $\circ$ kan men zich een aritmetische operatie $(+, \cdot)$ voorstellen, maar ook operaties als maximum en minimum bepalen.

We beschouwen nu het volgende probleem: gegeven de machines $B_1, \dots, B_n$. Iedere machine B_j bevat een getal v_j . Bewerkstellig dat in iedere machine B_j

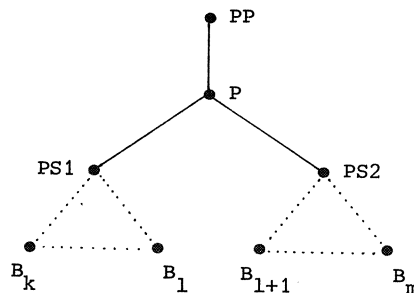
de waarde van $v_1 \circ \dots \circ v_j$ terecht komt. STONE [1975] heeft de oplossing van dit probleem beschreven voor het geval dat de machines $B_1, \dots, B_n$ door middel van een zogenaamd "shuffle-exchange" netwerk onderling verbonden zijn. Die oplossing vereist $O(\log n)$ tijd. Zo'n efficiënte oplossing is ook mogelijk als de machines $B_1, \dots, B_n$ in symmetrische volgorde de bladeren zijn van een boommachine. De algoritme werkt als volgt. Iedere processor P (behalve de bladeren) voert het volgende programma uit:

```

1. program R; var p1,p2,p;
2.           processor PP,PS1,PS2;
3. begin PS1? p1; PS2? p2;
4.       PP! p1  $\circ$  p2;
5.       PP? p;
6.       PS1! p; PS2! p1  $\circ$  p
7. end

```

De processoren PP, PS1 en PS2 zijn weer de voorganger en opvolgers van P . Als P de wortel is dan krijgt de variabele p in regel 5 de neutrale waarde van operator $\circ$. (Als $\circ$ b.v. de vermenigvuldigingsoperator is, dan is de neutrale waarde 1.) De correctheid van de algoritme is als volgt in te zien. Stel $B_k, B_{k+1}, \dots, B_l$ zijn alle bladeren van de deelboom met PS1 als wortel en $B_{l+1}, \dots, B_m$ zijn alle bladeren van de deelboom met PS2 als wortel



Figuur 1

(zie Figuur 1), dan geldt voor de variabelen $p1$ en $p2$ na regel 3 van bovenstaand programma:

$$\begin{aligned}
 (6) \quad p_1 &= v_k \circ v_{k+1} \circ \dots \circ v_1 \\
 p_2 &= v_{1+1} \circ v_{1+2} \circ \dots \circ v_m
 \end{aligned}$$

en voor de variabele p geldt na regel 5:

$$(7) \quad p = v_1 \circ v_2 \circ \dots \circ v_{k-1}.$$

De relaties (6) en (7) zijn te verifiëren met behulp van een inductief argument en het gegeven dat de berekening gestart wordt door alle bladmachines B_j het volgende programma te laten uitvoeren:

```

Bj: program Rj; var v,p,r,...
      processor PP,...
begin .
      .
      .
      PP! v;      {v=vj}
      .
      .
      PP? p;      {p=v1∘...∘vj-1}
      .
      .
      r := v∘p; {r=v1∘...∘vj}
      .
end

```

Globaal gezien worden er eerst gegevens van de bladeren naar de wortel gestuurd, daarna worden er gegevens van de wortel terug naar de bladeren gestuurd (door tussenliggende processoren worden de gegevens gemodificeerd). De algoritme is dus vrij van deadlock en de rekentijd is $O(\log n)$.

2.2. Meerdimensionale verdeel-en-heers algoritmen

Als we te maken hebben met een probleem dat betrekking heeft op punten in een meerdimensionale ruimte kunnen we als volgt te werk gaan (BENTLEY [1980]). Om een probleem van N punten in een k -dimensionale ruimte op te lossen, lossen we eerst recursief twee problemen op van elk $N/2$ punten in een k -dimensionale ruimte en vervolgens lossen we, weer recursief, één probleem op van hooguit N punten in een $k-1$ dimensionale ruimte. Voorbeelden en een meer gedetailleerde beschrijving zijn te vinden in (BENTLEY [1980]).

Ook bij meerdimensionale verdeel-en-heers algoritmen zijn er drie stappen te onderscheiden: 1) het partitioneren, 2) het recursief oplossen van deelproblemen, en 3) het combineren van deeloplossingen. Een procedure om een probleem op te lossen dat betrekking heeft op een collectie S bestaande uit N punten in een k -dimensionale ruimte, ziet er als volgt uit:

```

1. procedure MDC (in  $S, N, K$ ; out  $S$ );
2. var  $S1, S2$ : collectie punten;
3. begin partitioneer ( $S; S1, S2$ );
4.   if  $N \geq 2$  then MDC ( $S1, N/2, k$ ;  $S1$ ); MDC ( $S2, N/2, k$ ;  $S2$ ) fi;
5.   combineer ( $S1, S2$ ;  $S$ );
6.   if  $k > 1$  then MDC ( $S, N, k-1$ ;  $S$ ) fi
7. end

```

Een equivalent programma voor een boommachine ziet er als volgt uit:

```

1. program TMDC; var  $N, k$ : integer;  $S, S1, S2$ : collectie punten;
2.   processor PP, PS1, PS2;
3. begin PP?  $N, k, S$ ;
4.   do partitioneer ( $S; S1, S2$ );
5.   if  $N \geq 2$  then PS1!  $N/2, k, S1$ ; PS2!  $N/2, k, S2$ ;
6.   PS1?  $S1$ ; PS2?  $S2$ ;
7.   fi;
8.   combineer ( $S1, S2$ ;  $S$ );
9.    $k := k-1$ 
10.  until  $k < 1$ ;
11.  PP!  $S$ 
12. end

```

In het bovenstaande hebben we aangenomen dat behalve coördinaten nog andere relevante informatie opgenomen kan zijn in de collecties punten. Deze informatie kan gewijzigd worden door uitvoering van het programma.

Als we aannemen dat de rekentijden van de partitioneer- en combineer-stappen $O(N)$ zijn, dan krijgen we de volgende recurrente betrekking voor de rekentijd $T(N, k)$ van de recursive procedure MDC:

$$T(N, k) = 2T(N/2, k) + T(N, k-1) + \beta N.$$

Gebruiken we

$$T(N,1) = \alpha N$$

als basis voor inductie over k , dan krijgen we:

$$(8) \quad T(N,k) = O(N \log^{k-1} N).$$

Laat $TT(N,k)$ de rekentijd van programma TMDC zijn; dan geldt de recurrente betrekking:

$$TT(N,k) = TT(N/2,k) + TT(N,k-1) + \beta N.$$

Gebruiken we wederom

$$TT(N,1) = \alpha N$$

als basis voor inductie over k , dan krijgen we:

$$(9) \quad TT(N,k) = (\alpha 2^{k-1} + \beta(2^k - 2))N = O(2^k N).$$

Voor een aantal toepassingen weten we dat in regel 8 van TMDC en in regel 5 van MDC de collectie S hooguit $N/2$ punten heeft die van belang zijn. Laat $T'(N,k)$ en $TT'(N,k)$ de rekentijden zijn voor dergelijke toepassingen; dan hebben we:

$$T'(N,k) = 2T'(N/2,k) + T'(N/2,k-1) + \beta N$$

en

$$TT'(N,k) = TT'(N/2,k) + TT'(N/2,k-1) + \beta N$$

waaruit volgt:

$$(10) \quad T'(N,k) = O(N \log^{k-1} N)$$

en

$$(11) \quad TT'(N,k) = (\alpha + 2(k-1)\beta)N = O(kN).$$

Formules (10) en (11) geven een aanwijzing welke besparingen in rekentijd behaald kunnen worden wanneer met boommachines meerdimensionale verdeel-en-heers algoritmen worden uitgevoerd.

2.3. Eindige-elementenberekeningen

De eindige-elementenmethode wordt veel gebruikt om bepaalde typen partiële differentiaalvergelijkingen op te lossen (STRANG & FIX [1973], ZIENKIEWICZ [1977]). De methode wordt toegepast voor het oplossen van problemen op het gebied van sterkte-berekeningen, vloeistof-mechanica, warmte-overdracht, electro-magnetisme en vele andere.

In wezen bestaat de methode uit twee delen:

- het bepalen van een - zeg $n \times n$ - structuurmatrix Q en een structuurvector f
- het oplossen van de vector w uit het stelsel vergelijkingen

$$(12) \quad Qw = f.$$

Het stelsel vergelijkingen heeft betrekking op iets wat wel structuur of gebied genoemd wordt. Die structuur of dat gebied is verdeeld in zogenaamde elementen. Bij elk element e hoort een $k_e \times k_e$ matrix Q^e , de zogenaamde elementmatrix. De structuurmatrix Q wordt verkregen door de elementenmatrices Q^e te assembleren:

$$Q = \sum_e C_e Q^e C_e^T.$$

De matrices C_e (met afmetingen $n \times k_e$) worden verbindingsmatrices genoemd en geven de manier aan waarop de naar verhouding kleine elementmatrices moeten worden opgeblazen en bij elkaar moeten worden opgeteld om de grote structuurmatrix Q te verkrijgen. Op dezelfde manier wordt de structuurvector uit elementenvectoren f^e verkregen:

$$f = \sum_e C_e f^e.$$

Gewoonlijk is Q een symmetrische, positief definitieve matrix. Bovendien is Q over het algemeen nogal ijl. De oplossingsvector w wordt gewoonlijk als volgt uit (12) berekend. Eerst wordt de matrix Q gedeclineerd: $Q = LL^T$, waarbij L een benedendriehoeksmatrix voorstelt. Vervolgens wordt bij de voorwaartse substitutie w' opgelost uit $Lw' = f$ en ten slotte wordt w door achterwaartse substitutie bepaald uit $L^T w = w'$.

Een verdeel-en-heers algoritme voor het oplossen van eindige-elementenproblemen kunnen we verkrijgen door de bekende substructureringstechniek

(zie b.v. (PETERS [1980a])) toe te passen. In dat geval wordt niet de structuurmatrix Q bepaald, maar in plaats daarvan wordt de structuur S verdeeld in een aantal substructuren S_j ($j = 1, \dots, k$) zodanig dat elk element tot precies één substructuur behoort. Vervolgens worden de substructuurmatrices Q_j die bij S_j behoren, berekend en (gedeeltelijk) gedeclineerd. De zo verkregen gereduceerde substructuurmatrices worden geassembleerd; dit levert de matrix $\tilde{Q}$. Vervolgens wordt $\tilde{Q}$ gedeclineerd, waarna door aangepaste voorwaartse en achterwaartse substitutie de oplossingsvector w berekend kan worden. Voor precieze details kan men terecht in (PETERS [1980a]). De substructuren mogen op hun beurt (tenzij ze uit slechts één element bestaan) ook weer opgedeeld worden. Zo verkrijgt men een hiërarchie van substructuren.

Als iedere structuur in slechts twee substructuren is verdeeld dan hebben we te maken met een binaire boom. Een boommachine lijkt dan ook bij uitstek geschikt voor dergelijke eindige-elementenberekeningen. Een beschrijving van zo'n boommachineprogramma is in (PETERS [1980b]) te vinden. Om inzicht te krijgen in de doelmatigheid ervan schetsen we nu een recursieve procedure om de gereduceerde structuurmatrix van een rechthoekig $q \times r$ netwerk R te berekenen:

```

procedure ur (in q,r: integer; out a: array);
var q1,q2,r1,r2: integer; a1,a2: array;
begin if (q=1) and (r=1)
  then bereken elementmatrix
  else begin verdeel R in twee rechthoeken R1 en R2
    (d.w.z. bereken q1,q2,r1,r2);
    ur (q1,r1; a1);
    ur (q2,r2; a2);
    assembleer
  end;
  decomponeer
end

```

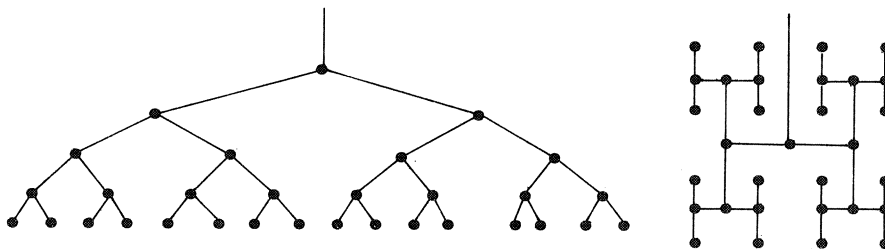
Op de manier, zoals we dat in paragraaf 1 beschreven hebben, is deze recursieve procedure eenvoudig te transformeren tot een programma voor een boommachine. Als ur gebruikt wordt om een vierkant netwerk met $O(n)$ elementen door te rekenen, dan zijn er $O(n^{3/2})$ aritmetische operaties vereist voor de stappen assembleer en decomponeer. Uit formule (3) volgt nu dat een boommachine een factor 2.2 in tijd spaart. Uiteraard kan een vierkant niet in

twee vierkanten verdeeld worden, dus er is een wat nauwkeuriger analyse vereist. Daaruit blijkt dat er een factor 5 aan tijdwinst te behalen is (PETERS [1980b]). Belangrijker echter is de observatie dat de tijdwinst constant is, onafhankelijk van n , de omvang van het probleem.

Opgemerkt kan worden dat de eindige-elementenmachine geen grote hoeveelheden gegevens uitwisselt met zijn omgeving. De machine begint zo gauw de wortel de waarden van q en r ontvangt. Er hoeft niet zo iets als een grote matrix van de buitenwereld naar de machine overgebracht te worden. De machine genereert zelf de grote hoeveelheden gegevens die een rol spelen. De berekeningen zijn beëindigd op het moment dat alle delen van de oplossingsvector bewerkt zijn door de bladmachines. Deze resultaten hoeven niet noodzakelijkerwijs via de wortel naar de buitenwereld getransporteerd te worden; er kunnen directe verbindingen tussen de bladeren en de buitenwereld gelegd worden.

3. REALISATIE

Figuur 2 laat twee mogelijke representaties van boomstructuren in het platte vlak zien. Merk op dat bij beide representaties de verbindingslijnen

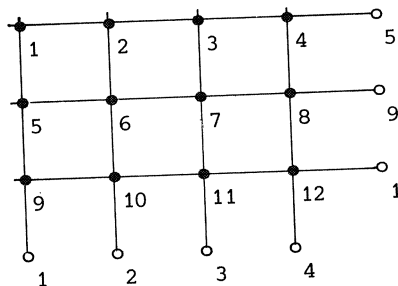


Figuur 2

niet uniform zijn. Naarmate twee naburige processoren dicht bij de wortel liggen, is hun verbindingsdraad langer. In het voorgaande hebben we steeds aangenomen dat de communicatietijd van twee naburige processoren onafhankelijk is van hun plaats in de boom. Deze aanname is toch realistisch omdat het mogelijk is om de tijd die een signaal nodig heeft om zich langs een draad voort te planten, onafhankelijk te maken van de lengte van de draad. Dit kan bewerkstelligd worden door langere draden van grotere "drivers" te voorzien (KUNG [1980], MEAD & REM [1979]). Grotere drivers hebben echter

meer ruimte nodig. Desalniettemin hebben Mead en Rem aangetoond dat het mogelijk is om een boommachine zo te realiseren dat de benodigde oppervlakte in wezen evenredig is met het aantal processoren en de communicatietijd van twee naburige processoren onafhankelijk is van hun plaats in de boom.

Een bezwaar van de hierboven geschetste realisatie van een boommachine is dat tijdens een recursieve berekening op elk moment hooguit de helft van de processoren aan het werk is. Aan dit bezwaar kan worden tegemoet gekomen door een geringer aantal processoren in een netwerk te rangschikken en iedere processor programma's te laten uitvoeren die oorspronkelijk aan een aantal verschillende processoren waren toegedracht. MARTIN [1979] heeft beschreven hoe een binaire "computation tree", zoals die wordt voortgebracht door een recursieve berekening, kan worden afgebeeld op een eindig netwerk. Topologisch gezien is dat netwerk een torus; iedere processor is met vier andere verbonden (zie Figuur 3). Door ervoor te zorgen dat twee



Figuur 3

naburige punten van de binaire boom afgebeeld worden op twee naburige processoren van het netwerk bereik je dat de communicatietijden niet wezenlijk toenemen. Wel wordt er enig parallelisme opgeofferd. Een analyse van de rekentijden wordt daardoor veel ingewikkelder.

Op de door Martin beschreven wijze kan een onbegrensde berekeningsboom op een eindig netwerk worden afgebeeld. Daardoor wordt een ander bezwaar van een boomachtige organisatie vermeden. Doorgaans is namelijk de recursiediepte van een berekening niet bij voorbaat bekend en dus is het aantal benodigde processoren onbekend. In Hoofdstuk 1 is al aangegeven dat dit probleem ook opgelost kan worden door de bladmachines recursieve procedures te laten uitvoeren.

4. SLOTOPMERKINGEN

Om een aantal processoren gezamenlijk één karwei te laten uitvoeren moeten ze onderling verbonden worden. Er zijn tal van verbindingspatronen te bedenken. Welk verbindingspatroon het best gekozen kan worden hangt af van het soort problemen of de klasse van algoritmen waar de machine voor bedoeld is. Voor recursieve berekeningen is de boommachine een voor de hand liggende architectuur. Voorbeelden van andere verbindingspatronen zijn: de door Kung en Leiserson geïntroduceerde hexagonale structuren, b.v. voor het oplossen van een stelsel lineaire vergelijkingen (KUNG [1980]) en de door STONE [1975] beschreven "shuffle-exchange" netwerken om te sorteren en voor FFT berekeningen. Deze netwerken zijn ontworpen voor bepaalde toepassingen en zijn niet bedoeld als general-purpose rekenapparatuur. Bij de boommachine ligt dat wat anders. Browning claimt dat de boommachine een general-purpose rekenmachine is, waarvoor reeds tal van algoritmen zijn ontworpen. Desondanks heeft de boommachine een aantal beperkingen waardoor zij niet bijster geschikt is voor alle mogelijke toepassingen. In het voorgaande hebben we al laten zien dat er zelfs recursieve berekeningen zijn waarvoor een boommachine weinig vruchten afwerpt. Bovendien lijkt een boommachine ook niet erg geschikt om b.v. te sorteren. Sorteeralgoritmen voor boommachines vergen $O(n)$ tijd (als n het aantal te sorteren objecten is), terwijl het "shuffle-exchange" netwerk slechts $O(\log^2 n)$ tijd nodig heeft (STONE [1975]).

Tenslotte, voor vele recursieve algoritmen is het niet doelmatig de boommachine met meer dan slechts enkele processoren uit te rusten. Zo gauw de taak die een subboom moet verrichten klein is in vergelijking met de taak van de wortel, dan is de tijdwinst die verkregen kan worden door zoveel mogelijk parallellisme te exploiteren, verwaarloosbaar. Bijvoorbeeld, als de wortel αN^3 operaties moet uitvoeren en de boomhoogte is twee (d.w.z. de boom bestaat uit 7 processoren), dan is het aantal operaties dat door een bladmachine L moet worden uitgevoerd slechts $\frac{\alpha}{56} N^3$, ofwel de rekentijd van L is minder dan 2% van de rekentijd van de wortel. Dus versnelling van L zet geen zoden aan de dijk. Er zijn echter ook recursieve berekeningen waarvoor het wél de moeite loont om de boommachine van zoveel processoren te voorzien dat de taken van de bladmachines niet verder gesplitst kunnen worden; dat zijn namelijk de berekeningen die gekarakteriseerd worden door het feit dat de tijd die gemoeid is met de partitioneer- en combineerstappen, onafhankelijk is van de omvang van het op te lossen probleem.

5. REFERENTIES

- BENTLEY, J.L., *Multidimensional divide-and-conquer*, Comm. ACM 23 (1980), pp.214-229.
- BENTLEY, J.L. & H.T. KUNG, *A tree machine for searching problems*, in: Proceedings IEEE Internat. Conf. on Parallel Processing, New York, 1979, pp.257-266.
- BROWNING, S.A., *A tree machine*, Lambda 1 (1980), pp.31-36.
- HOARE, C.A.R., *Communicating sequential processes*, Comm. ACM 21 (1978), pp.666-677.
- KUNG, H.T., *The structure of parallel algorithms*, in: Advances in Computers, Vol. 19, Academic Press, 1980, pp.65-112.
- MARTIN, A.J., *A distributed architecture for parallel recursive computations*, Intern rapport, AJM 18, Eindhoven, Philips Nat. Lab., 1979.
- MEAD, C.A. & M. REM, *Cost and performance of VLSI computing structures*, IEEE J. Solid State Circuits sc-14 (1979), pp.455-462.
- PETERS, F.J., *Sparse matrices and substructures*, Mathematical Centre Tracts 119, Amsterdam, Mathematisch Centrum, 1980a.
- PETERS, F.J., *Parallel large scale finite element computations*, in: Proceedings IEEE Conference on Circuits and Computers for Large Scale Systems, New York, 1980b, pp.992-995.
- ROSE, D.J. & G.F. WHITTEN, *A recursive analysis of dissection strategies*, in: Sparse matrix computations (J.R. Bunch and D.J. Rose eds), New York, Academic Press, 1976.
- STONE, H.S., *Parallel computers*, in: Introduction to computer architecture (H.S. Stone ed.), Chicago, Science Research Associates, 1975, pp.318-374.
- STRANG, G. & G.J. FIX, *An analysis of the finite element method*, Englewood Cliffs, Prentice-Hall, 1973.
- ZIENKIEWICZ, O.C., *The finite element method*, third edition, London, McGraw-Hill, 1977.

UITGAVEN IN DE SERIE MC SYLLABUS

Onderstaande uitgaven zijn verkrijgbaar bij het Mathematisch Centrum,
2e Boerhaavestraat 49 te Amsterdam-1005, tel. 020-947272.

-
- | | |
|----------|---|
| MCS 1.1 | F. GÖBEL & J. VAN DE LUNE, <i>Leergang Besliskunde, deel 1: Wiskundige basiskennis</i> , 1965. ISBN 90 6196 014 2. |
| MCS 1.2 | J. HEMELRIJK & J. KRIENS, <i>Leergang Besliskunde, deel 2: Kansberekening</i> , 1965. ISBN 90 6196 015 0. |
| MCS 1.3 | J. HEMELRIJK & J. KRIENS, <i>Leergang Besliskunde, deel 3: Statistiek</i> , 1966. ISBN 90 6196 016 9. |
| MCS 1.4 | G. DE LEVE & W. MOLENAAR, <i>Leergang Besliskunde, deel 4: Markovketens en wachttijden</i> , 1966. ISBN 90 6196 017 7. |
| MCS 1.5 | J. KRIENS & G. DE LEVE, <i>Leergang Besliskunde, deel 5: Inleiding tot de mathematische besliskunde</i> , 1966. ISBN 90 6196 018 5. |
| MCS 1.6a | B. DORHOUT & J. KRIENS, <i>Leergang Besliskunde, deel 6a: Wiskundige programmering 1</i> , 1968. ISBN 90 6196 032 0. |
| MCS 1.6b | B. DORHOUT, J. KRIENS & J.TH. VAN LIESHOUT, <i>Leergang Besliskunde, deel 6b: Wiskundige programmering 2</i> , 1977. ISBN 90 6196 150 5. |
| MCS 1.7a | G. DE LEVE, <i>Leergang Besliskunde, deel 7a: Dynamische programmering 1</i> , 1968. ISBN 90 6196 033 9. |
| MCS 1.7b | G. DE LEVE & H.C. TIJMS, <i>Leergang Besliskunde, deel 7b: Dynamische programmering 2</i> , 1970. ISBN 90 6196 055 X. |
| MCS 1.7c | G. DE LEVE & H.C. TIJMS, <i>Leergang Besliskunde, deel 7c: Dynamische programmering 3</i> , 1971. ISBN 90 6196 066 5. |
| MCS 1.8 | J. KRIENS, F. GÖBEL & W. MOLENAAR, <i>Leergang Besliskunde, deel 8: Minimaxmethode, netwerkplanning, simulatie</i> , 1968. ISBN 90 6196 034 7. |
| MCS 2.1 | G.J.R. FÖRCH, P.J. VAN DER HOUWEN & R.P. VAN DE RIET, <i>Colloquium Stabiliteit van differentieschema's, deel 1</i> , 1967. ISBN 90 6196 023 1. |
| MCS 2.2 | L. DEKKER, T.J. DEKKER, P.J. VAN DER HOUWEN & M.N. SPIJKER, <i>Colloquium Stabiliteit van differentieschema's, deel 2</i> , 1968. ISBN 90 6196 035 5. |
| MCS 3.1 | H.A. LAUWERIER, <i>Randwaardeproblemen, deel 1</i> , 1967. ISBN 90 6196 024 X. |
| MCS 3.2 | H.A. LAUWERIER, <i>Randwaardeproblemen, deel 2</i> , 1968. ISBN 90 6196 036 3. |
| MCS 3.3 | H.A. LAUWERIER, <i>Randwaardeproblemen, deel 3</i> , 1968. ISBN 90 6196 043 6. |
| MCS 4 | H.A. LAUWERIER, <i>Representaties van groepen</i> , 1968. ISBN 90 6196 037 1. |

- MCS 5 J.H. VAN LINT, J.J. SEIDEL & P.C. BAAYEN, *Colloquium Discrete wiskunde*, 1968. ISBN 90 6196 044 4.
- MCS 6 K.K. KOKSMA, *Cursus ALGOL 60*, 1969. ISBN 90 6196 045 2.
- MCS 7.1 *Colloquium Moderne rekenmachines, deel 1*, 1969. ISBN 90 6196 046 0.
- MCS 7.2 *Colloquium Moderne rekenmachines, deel 2*, 1969. ISBN 90 6196 047 9.
- MCS 8 H. BAVINCK & J. GRASMAN, *Relaxatietrillingen*, 1969. ISBN 90 6196 056 8.
- MCS 9.1 T.M.T. COOLEN, G.J.R. FÖRCH, E.M. DE JAGER & H.G.J. PIJLS, *Elliptische differentiaalvergelijkingen, deel 1*, 1970. ISBN 90 6196 048 7.
- MCS 9.2 W.P. VAN DEN BRINK, T.M.T. COOLEN, B. DIJKHUIS, P.P.N. DE GROEN, P.J. VAN DER HOUWEN, E.M. DE JAGER, N.M. TEMME & R.J. DE VOGELAERE, *Colloquium Elliptische differentiaalvergelijkingen, deel 2*, 1970. ISBN 90 6196 049 5.
- MCS 10 J. FABIUS & W.R. VAN ZWET, *Grondbegrippen van de waarschijnlijkheidsrekening*, 1970. ISBN 90 6196 057 6.
- MCS 11 H. BART, M.A. KAASHOEK, H.G.J. PIJLS, W.J. DE SCHIPPER & J. DE VRIES, *Colloquium Halfalgebra's en positieve operatoren*, 1971. ISBN 90 6196 067 3.
- MCS 12 T.J. DEKKER, *Numerieke algebra*, 1971. ISBN 90 6196 068 1.
- MCS 13 F.E.J. KRUSEMAN ARETZ, *Programmeren voor rekenautomaten; De MC ALGOL 60 vertaler voor de EL X8*, 1971. ISBN 90 6196 069 x.
- MCS 14 H. BAVINCK, W. GAUTSCHI & G.M. WILLEMS, *Colloquium Approximatie-theorie*, 1971. ISBN 90 6196 070 3.
- MCS 15.1 T.J. DEKKER, P.W. HEMKER & P.J. VAN DER HOUWEN, *Colloquium Stijve differentiaalvergelijkingen, deel 1*, 1972. ISBN 90 6196 078 9.
- MCS 15.2 P.A. BEENTJES, K. DEKKER, H.C. HEMKER, S.P.N. VAN KAMPEN & G.M. WILLEMS, *Colloquium Stijve differentiaalvergelijkingen, deel 2*, 1973. ISBN 90 6196 079 7.
- MCS 15.3 P.A. BEENTJES, K. DEKKER, P.W. HEMKER & M. VAN VELDHIJZEN, *Colloquium Stijve differentiaalvergelijkingen, deel 3*, 1975. ISBN 90 6196 118 1.
- MCS 16.1 L. GEURTS, *Cursus Programmeren, deel 1: De elementen van het programmeren*, 1973. ISBN 90 6196 080 0.
- MCS 16.2 L. GEURTS, *Cursus Programmeren, deel 2: De programmeertaal ALGOL 60*, 1973. ISBN 90 6196 087 8.
- MCS 17.1 P.S. STOBBE, *Lineaire algebra, deel 1*, 1974. ISBN 90 6196 090 8.
- MCS 17.2 P.S. STOBBE, *Lineaire algebra, deel 2*, 1974. ISBN 90 6196 091 6.
- MCS 17.3 N.M. TEMME, *Lineaire algebra, deel 3*, 1976. ISBN 90 6196 123 8.
- MCS 18 F. VAN DER BLIJ, H. FREUDENTHAL, J.J. DE IONGH, J.J. SEIDEL & A. VAN WIJNGAARDEN, *Een kwart eeuw wiskunde 1946-1971, Syllabus van de Vakantiecursus 1971*, 1974. ISBN 90 6196 092 4.
- MCS 19 A. HORDIJK, R. POTHARST & J.Th. RUNNENBURG, *Optimaal stoppen van Markovketens*, 1974. ISBN 90 6196 093 2.

- MCS 20 T.M.T. COOLEN, P.W. HEMKER, P.J. VAN DER HOUWEN & E. SLAGT, *ALGOL 60 procedures voor begin- en randwaardeproblemen*, 1976. ISBN 90 6196 094 0.
- MCS 21 J.W. DE BAKKER (red.), *Colloquium Programmacorrectheid*, 1975. ISBN 90 6196 103 3.
- MCS 22 R. HELMERS, F.H. RUYMGAART, M.C.A. VAN ZUYLEN & J. OOSTERHOFF, *Asymptotische methoden in de toetsingstheorie; Toepassingen van naburigheid*, 1976. ISBN 90 6196 104 1.
- MCS 23.1 J.W. DE ROEVER (red.), *Colloquium Onderwerpen uit de biomathematica, deel 1*, 1976. ISBN 90 6196 105 X.
- MCS 23.2 J.W. DE ROEVER (red.), *Colloquium Onderwerpen uit de biomathematica, deel 2*, 1976. ISBN 90 6196 115 7.
- MCS 24.1 P.J. VAN DER HOUWEN, *Numerieke integratie van differentiaalvergelijkingen, deel 1: Eenstapsmethoden*, 1974. ISBN 90 6196 106 8.
- MCS 25 *Colloquium Structuur van programmeertalen*, 1976. ISBN 90 6196 116 5.
- MCS 26.1 N.M. TEMME (ed.), *Nonlinear analysis, volume 1*, 1976. ISBN 90 6196 117 3.
- MCS 26.2 N.M. TEMME (ed.), *Nonlinear analysis, volume 2*, 1976. ISBN 90 6196 121 1.
- MCS 27 M. BAKKER, P.W. HEMKER, P.J. VAN DER HOUWEN, S.J. POLAK & M. VAN VELDHUIZEN, *Colloquium Discretiseringsmethoden*, 1976. ISBN 90 6196 124 6.
- MCS 28 O. DIEKMANN, N.M. TEMME (EDS), *Nonlinear Diffusion Problems*, 1976. ISBN 90 6196 126 2.
- MCS 29.1 J.C.P. BUS (red.), *Colloquium Numerieke programmatuur, deel 1A, deel 1B*, 1976. ISBN 90 6196 128 9.
- MCS 29.2 H.J.J. TE RIELE (red.), *Colloquium Numerieke programmatuur, deel 2*, 1976. ISBN 90 6196 144 0
- * MCS 30 P. GROENEBOOM, R. HELMERS, J. OOSTERHOFF & R. POTHAARST, *Efficiency begrippen in de statistiek*, . ISBN 90 6196 149 1.
- MCS 31 J.H. VAN LINT (red.), *Inleiding in de coderingstheorie*, 1976. ISBN 90 6196 136 X.
- MCS 32 L. GEURTS (red.), *Colloquium Bedrijfssystemen*, 1976. ISBN 90 6196 137 8.
- MCS 33 P.J. VAN DER HOUWEN, *Differentieschema's voor de berekening van waterstanden in zeeën en rivieren*, 1977. ISBN 90 6196 138 6.
- MCS 34 J. HEMELRIJK, *Oriënterende cursus mathematische statistiek*, ISBN 90 6196 139 4.
- MCS 35 P.J.W. TEN HAGEN (red.), *Colloquium Computer Graphics*, 1977. ISBN 90 6196 142 4.
- MCS 36 J.M. AARTS, J. DE VRIES, *Colloquium Topologische Dynamische Systemen*, 1977. ISBN 90 6196 143 2.
- MCS 37 J.C. van Vliet (red.), *Colloquium Capita Datastructuren*, 1978. ISBN 90 6196 159 9.

- MCS 38.1 T.H. KOORNWINDER (ED.), *Representations of locally compact groups with applications*, 1979. ISBN 90 6196 161 0.
- MCS 38.2 T.H. KOORNWINDER (ED.), *Representations of locally compact groups with applications*, 1979. ISBN 90 6196 181 5.
- MCS 39 O.J. VRIEZE & G.L. WANROOIJ, *Colloquium Stochastische Spelen*, 1979. ISBN 90 6196 167 X.
- MCS 40 J. VAN TIEL, *Convexe Analyse*, 1979. ISBN 90 6196 187 4.
- MCS 41 H.J.J. TE RIELE (ED.), *Colloquium Numerical Treatment of Integral Equations*, 1979. ISBN 90 6196 189 0.
- MCS 42 J.C. VAN VLIET (RED.), *Colloquium Capita Implementatie van Programmeertalen*, 1980. ISBN 90 6196 191 2.
- MCS 43 A.M. COHEN & H.A. WILBRINK, *Eindige groepen (Een inleidende cursus)*, 1980. ISBN 90 6196 203 X.
- MCS 44 J.G. VERWER (ED.), *Numerical solution of partial differential equations*, 1980. ISBN 90 6196 205 6.
- MCS 45 P. KLINT (RED.), *Colloquium hogere programmeertalen en computer-architectuur*, 1980. ISBN 90 6196 206 4.
- MCS 46.1 P.M.G. APERS (RED.), *Colloquium Databankorganisatie*, 1981. ISBN 90 6196 212 9.
- MCS 46.2 P.M.G. APERS (RED.), *Colloquium Databankorganisatie*, 1981. ISBN 90 6196 232 3.
- MCS 47.1 P.W. HEMKER (ED.), *NUMAL, numerical procedures in ALGOL 60, Part I: General information and indices*, 1981. ISBN 90 6196 217 X.
- MCS 47.2 P.W. HEMKER (ED.), *NUMAL, numerical procedures in ALGOL 60, Part II: Elementary procedures, algebraic evaluations*, 1981. ISBN 90 6196 217 X.
- MCS 47.3 P.W. HEMKER (ED.), *NUMAL, numerical procedures in ALGOL 60, Part III: Linear algebra, part I*, 1981. ISBN 90 6196 217 X.
- MCS 47.4 P.W. HEMKER (ED.), *NUMAL, numerical procedures in ALGOL 60, Part IV: Linear algebra, part II*, 1981. ISBN 90 6196 217 X.
- MCS 47.5 P.W. HEMKER (ED.), *NUMAL, numerical procedures in ALGOL 60, Part V: Analytical evaluations, analytical problems, part I*, 1981. ISBN 90 6196 217 X.
- MCS 47.6 P.W. HEMKER (ED.), *NUMAL, numerical procedures in ALGOL 60, Part VI: Analytical problems, part II*, 1981. ISBN 90 6196 217 X.
- MCS 47.7 P.W. HEMKER (ED.), *NUMAL, numerical procedures in ALGOL 60, Part VII: Special functions and constants, interpolation and approximation*, 1981. ISBN 90 6196 217 X.
- MCS 48.1 P.M.B. VITANYI, J. VAN LEEUWEN & P. VAN EMBE BOAS (RED.), *Colloquium complexiteit en algoritmen, deel I*, 1982. ISBN 90 6196 237 4

- MCS 49 T.H. KOORNWINDER (ED.), *The structure of real semisimple Lie groups*,
1982. ISBN 90 6196 239 0.
- MCS 50 H. NIJMEIJER, *Inleiding systeemtheorie*, 1982.
ISBN 90 6196 240 4.

De met een * gemerkte uitgaven moeten nog verschijnen.

