Mark de Berg
Faculteit Wiskunde en Informatica
Technische Universiteit Eindhoven
Postbus 513
5600 MB Eindhoven
m.t.d.berg@tue.nl

Inaugurele rede

Algoritmiek en

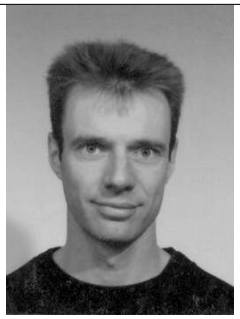
Ruimtelijke data spelen een belangrijke rol in veel toepassingsgebieden van de informatica — denk bijvoorbeeld aan geografische informatiesystemen, robotica, CAD/CAM en virtual reality. De computationele geometrie is het gebied binnen de algoritmiek dat zich bezig houdt met ruimtelijke data. Binnen het vakgebied zijn allerlei theoretisch uitstekende algoritmen ontwikkeld, die helaas te kort schieten bij praktische toepassingen. Mark de Berg, sinds 1 november 2002 hoogleraar aan de Technische Universiteit Eindhoven legt uit hoe dit komt.

Dat het belang van de informatica nog steeds toeneemt behoeft geen betoog. De informatica speelt echter niet alleen een steeds grotere rol in onze moderne maatschappij, het is ook een prachtige wetenschap. Een wetenschap waar ondanks (of misschien wel dankzij) de razendsnelle ontwikkeling die zij al heeft doorgemaakt, nog veel uitdagingen liggen. Ik zal een beeld schetsen van wat in mijn ogen een van de mooiste gebieden binnen de informatica is: de computationele geometrie: een gebied waar de informatica raakt aan de wiskunde, en waar fundamentele vragen raken aan praktische toepassingen. Ik hoop een deel van mijn enthousiasme voor de computationele geometrie op u over te brengen, en u daarnaast een idee te geven van wat ik als grote uitdagingen op dit gebied beschouw.

Geometrie

In 1644 publiceerde Descartes deel III van zijn beroemde *Principia Philosophiae*, waarin hij onder andere de structuur van het heelal beschrijft. Figuur 1 is een illustratie uit zijn werk. Te zien is een verdeling van de ruimte in gebieden. In elk gebied is een aantal concentrische cirkels te zien. Het middelpunt daarvan is de ster die het gebied domineert. Zo staat het label *S* bijvoorbeeld voor de zon. In de gebieden rond de sterren, vooral rond de zon, zijn planeten te zien. Als we iets preciezer naar de gebieden kijken, dan zien we dat het gebied rond een ster bestaat uit dat deel van de ruimte waarvoor die ster de meest dichtbijzijnde is. Een dergelijke opdeling van de ruimte in dichtstbijzijnde gebieden bij een gegeven verzameling punten (de sterren in de tekening van Descartes) wordt tegenwoordig het *Voronoidiagram* genoemd, naar een Russische wiskundige uit het begin van de twintigste eeuw. Voronoidiagrammen lijken misschien weinig met onze moderne maatschappij te maken te hebben, laat staan met de informatica. Maar schijn bedriegt.

Laten we eens een wat moderner onderwerp nemen: digitale hoogtekaarten. Stel dat we de hoogte op een groot aantal punten in, laten we zeggen, de Alpen gemeten hebben. Uit deze gegevens willen we een 3D-hoogtemodel van de Alpen genereren. De cruciale vraag daarbij is welke hoogte we toekennen aan punten waar we de hoogte niet gemeten hebben. We kunnen de hoogte van een punt gelijk kiezen aan de hoogte van het dichtstbijzijnde meetpunt — daarmee zijn we weer terug bij Voronoidiagrammen — maar dit is geen goede oplossing. Op deze manier creëren we namelijk een landschap dat bestaat uit een groot aantal plateaus, en dat is niet erg realistisch. Een betere oplossing is om de hoogte door interpolatie te bepalen. Het bepalen van de hoogte door middel van interpolatie kunnen we doen met behulp van een zogenaamde *triangulatie* van de meetpunten: een opdeling van het gebied in driehoeken waarvan de hoekpunten precies samenvallen met de meetpunten. Vervolgens geven we de hoekpunten de hoogte van de bijbehorende meetpunten. We krijgen dan een 3D-hoogtemodel van de Alpen dat, hopelijk, aardig overeenkomt met de werkelijkheid. Figuur 2 laat een voorbeeld zien. Een 3D-hoogtemodel dat op een triangulatie gebaseerd is wordt in de geografische informatiesystemen een *triangulated irregular network* (TIN), genoemd. Een probleem bij het berekenen van een TIN is dat er heel veel verschillende triangulaties mogelijk zijn. Elke triangulatie geeft een ander hoogtemodel. Wat is een goede triangulatie? Zolang we geen andere informatie hebben dan de hoogte in de meetpunten, is het onmogelijk om met zekerheid te zeggen dat de ene triangulatie een betere weergave van de werkelijkheid is dan de andere. Maar sommige hoogtemodellen zijn wel waarschijnlijker dan andere. Kijk bijvoorbeeld eens naar de verzameling meetpunten in figuur 3, en het punt p, dat geen meetpunt



Mark de Berg

geometrie

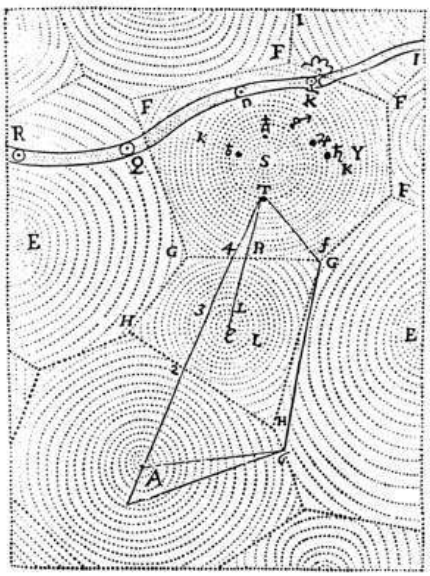
is. Als we de getekende driehoek in de triangulatie opnemen, dan zal de hoogte van p worden bepaald door interpolatie tussen de hoogtes van de meetpunten a, b, c. Maar de hoogte van bijvoorbeeld meetpunt d lijkt veel relevanter voor p dan de hoogte van c. Het is dus goed om lange dunne driehoeken zoveel mogelijk te vermijden. Een andere manier om dit te formuleren is dat we kleine hoeken (zoals de hoek bij punt c in de driehoek abc) zoveel mogelijk willen vermijden in de triangulatie. De triangulatie waarvan de kleinste hoek zo groot mogelijk is, wordt de *Delaunaytriangulatie* genoemd, naar (alweer) een Russische wiskundige. De Delaunaytriangulatie wordt binnen de geografische informatiesystemen veel gebruikt om TIN's van terreindata te genereren.

En nu komt een van de vele verrassingen die de geometrie in petto heeft. Stel dat we het Voronoidiagram van de meetpunten bepalen. Daarna verbinden we de meetpunten waarvan de gebieden aan elkaar grenzen, zoals in figuur 4, en vervolgens laten we het Voronoidiagram weer weg. Dan krijgen we precies de Delaunaytriangulatie van de meetpunten!

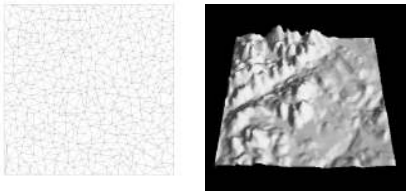
Voronoidiagrammen en Delaunaytriangulaties zijn slechts een voorbeeld van de prachtige structuren die de computationele geometrie zo'n mooi vakgebied maken.

Algoritmiek

Tot nu toe hebben we het vooral gehad over geometrische structuren, zoals ook bestudeerd in de wiskunde. De informatica voegt daar een nieuw aspect aan toe: de berekening, representatie, en manipulatie van dergelijke structuren met behulp van de computer. In het voorbeeld over hoogtemodellen wordt de vraag dan: hoe kunnen we de Delaunaytriangulatie van een verzameling meetpunten automatisch laten berekenen? Nu kan men bewijzen dat een drietal meetpunten a, b, c precies dan een van de driehoeken van de Delaunaytriangulatie vormt, als er geen andere meetpunten binnen de cirkel door a, b, c liggen. We kunnen dus alle driehoeken waaruit de Delaunaytriangulatie bestaat als volgt berekenen: ga alle drietallen meetpunten af, en test voor elk drietal of de bijbehorende cirkel een ander meetpunt bevat. Voor een klein aantal meetpunten is dit wellicht nog wel te doen, maar voor grotere aantallen meetpunten zal dit veel te veel tijd kosten. Voor bijvoorbeeld 1.000 meetpunten moeten we al 166.167.000 drietallen controleren, en voor 10.000 meetpunten loopt dat op tot ruim $1,6 \cdot 10^{11}$. Dit wordt veroorzaakt door het slechte opschaalgedrag van het algoritme. Voor n meetpunten zijn er ongeveer $n^3/6$ drietallen, en dat betekent dat het aantal drietallen met een factor 1.000 toeneemt als het aantal meetpunten met een factor 10 toeneemt. Hier komen we bij de kern van de algoritmiek: het ontwerp van *efficiënte algoritmen*, dat wil zeggen algoritmen met een goed opschaalgedrag. Hiervoor heeft de algoritmicus een groot aantal ontwerptechnieken ter beschikking. De algoritmiek houdt zich aan de ene kant bezig met het ontwikkelen van dergelijke ontwerptechnieken, en aan de andere kant met het (met behulp van die technieken) ontwerpen van efficiënte algoritmen voor specifieke problemen. Hierbij komen in de computationele geometrie vaak allerlei vragen naar voren van combinatorische aard. Er is dan ook een nauwe band tussen de computationele en de combinatorische geometrie.



Figuur 1 Illustratie uit deel III van Descartes' *Principia Philosophiae* (1644)



Figuur 2 Links: een triangulatie van een verzameling meetpunten in het vlak. Rechts: het hoogtemodel dat ontstaat door de hoekpunten van de triangulatie de gemeten hoogte te geven

Men zou kunnen denken dat de efficiëntie van algoritmen minder belangrijk wordt door de toenemende rekensnelheid van computers. Niets is minder waar. Niet alleen de rekensnelheid neemt toe, maar de hoeveelheid data waarmee gerekend wordt neemt minstens even hard toe. Dit maakt dat het opschaalgedrag van algoritmen alleen maar belangrijker wordt. Ik wil hier benadrukken dat het ontwerp van een efficiënt algoritme geen kwestie is van het optimaliseren of het slim implementeren van een inefficiënt algoritme. De manier waarop een algoritme wordt geïmplementeerd kan weliswaar een grote invloed hebben op de looptijd, maar het opschaalgedrag wordt hier niet door beïnvloed. Dat betekent dat inefficiënte oplossingen altijd traag zullen blijven voor grote data sets. In het voorbeeld van de Delaunaytriangulatie: hoe slim men de test of een cirkel een meetpunt bevat ook implementeert, feit blijft dat het algoritme alle drietallen punten controleert. Om dit te veranderen is een fundamenteel nieuwe aanpak nodig. Hiervoor moeten de juiste probleemspecifieke eigenschappen worden ontdekt, bewezen en uitgebuit, en moeten de juiste algoritmische technieken op een creatieve manier worden ingezet. Dit maakt het vinden van efficiënte algoritmen elke keer weer een intellectuele uitdaging van formaat.

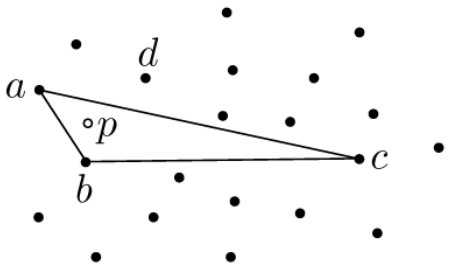
Toepassingen

Zodra we de wereld om ons heen (of juist een andere, virtuele wereld) willen modelleren, krijgen we te maken met ruimtelijke data. Ruimtelijke data spelen dan ook een belangrijke rol in veel toepassingsgebieden van de informatica. Laat ik een paar voorbeelden geven.

Een geografisch informatiesysteem (GIS) is een informatiesysteem dat geografische, en dus ruimtelijke, data opslaat. Bij de opslag en analyse daarvan komt een groot aantal algoritmische vragen naar voren. Eerder zagen we al een voorbeeld van een algoritmische vraag die ontstond bij het automatisch genereren van een hoogtemodel. Een ander voorbeeld is te vinden in het plaatsen van labels op een kaart. Stel dat we een topografische kaart van Nederland willen genereren, met daarop plaatsen, rivieren, en provincies aangegeven. Het geografisch informatiesysteem bevat alle daartoe benodigde ruimtelijke data: de locatie van de plaatsen, de loop van de rivieren, de grenzen tussen de provincies en dergelijke. Als we deze ruimtelijke data visualiseren krijgen we een kaart met daarop de plaatsen, rivieren, en provinciegrenzen. Wat nog ontbreekt zijn de bijbehorende labels: de plaats-, rivier-, en provincienamen. Ook deze informatie zal te vinden zijn in het systeem, maar de positionering van de labels — of het label 'Eindhoven' op de kaart links of rechts, boven of onder Eindhoven wordt geplaatst — is nog onbekend. Deze positionering is van groot belang. Bij een verkeerde plaatsing kunnen labels immers gaan overlappen en daardoor onleesbaar worden, of kan het onduidelijk zijn bij welke plaats een bepaald label hoort. We kunnen nu een cartograaf de labels laten positioneren, en daarna kunnen we de posities in het systeem opslaan. Dit is echter een kostbare oplossing. Daarnaast is deze aanpak niet mogelijk bij interactieve systemen, waarbij gebruikers zelf hun kaart kunnen samenstellen. Wat we nodig hebben is een algoritme dat, gegeven een beschrijving van alle objecten (plaatsen, rivieren) die gelabeld moeten worden, een goede plaatsing van de labels berekent. Ook hier geldt weer dat een naïeve aanpak onwerkbaar is. Zelfs als we slechts vier posities toestaan voor de positie van een plaatsnaam ten opzichte van de bijbehorende plaats (linksboven, rechtsboven, linksonder, en rechtsonder) dan nog is het volstrekt ondoenlijk om alle mogelijke combinaties van posities uit te proberen. Voor bijvoorbeeld 30 plaatsen geeft dit al meer dan 10^{18} mogelijkheden; zelfs met de snelste computer zou het afgaan van al deze mogelijkheden vele tientallen, zo niet honderden jaren in beslag nemen.

Een ander gebied waar ruimtelijke data een prominente rol spelen is de *computer-aided design and manufacturing* (CAD/CAM). Deze systemen zijn een onmisbaar hulpmiddel geworden in het ontwerp- en fabricageproces op een groot aantal terreinen. Ze worden gebruikt voor ontwerp en fabricage van eenvoudige gebruiksvoorwerpen zoals asbakken, maar ook voor complete industriële installaties zoals kerncentrales en boorplatformen. Ook in de CAD/CAM liggen de algoritmische problemen voor het oprapen.

Een voorbeeld: bij het ontwerp van bij-



Figuur 3 Een slechte driehoek: de hoogte van p wordt bepaalt door interpolatie van meetpunten a, b en c. Maar a en c liggen ver uit elkaar

voorbeeld kerncentrales wordt meestal geprobeerd om de centrale zo compact mogelijk te houden. Dit betekent wel dat de aan- en afvoer van grote onderdelen lastiger wordt. Het is dus wenselijk als het CAD-systeem dat gebruikt wordt bij het ontwerp van de centrale hier ondersteuning bij biedt. Het systeem moet dan vragen kunnen beantwoorden als: 'Kan een gegeven onderdeel, zonder gedemonteerd te hoeven worden, vanuit zijn huidige positie naar buiten de centrale gebracht worden, en zo ja, via welke route?' Dat is nuttig tijdens de ontwerpfase, omdat aan de hand van het antwoord eventueel besloten kan worden om het ontwerp aan te passen, maar het is ook nuttig voor planning van onderhoudstaken bij bestaande installaties.

Een laatste voorbeeld dat ik wil noemen is 3D-reconstructie. Het is tegenwoordig mogelijk om met een zogenaamde 3D-scanner het oppervlak van een voorwerp zeer nauwkeurig te meten. Wat preciezer: een 3D-scanner kan de coördinaten bepalen van een groot aantal punten op het oppervlak. Het resultaat is dus een wolk van punten die op het oppervlak van het gescande voorwerp liggen. Het doel van 3D-reconstructie is om uit deze ongestructureerde puntenwolk een 3D-model te berekenen, zoals geïllustreerd in figuur 6. Een mogelijke toepassing hiervan is bij het digitaliseren van belangrijke culturele erfgoederen. Zo is in Italië een aantal beelden van Michelangelo gescand en gereconstrueerd [8]. Ook kan 3D-reconstructie gebruikt worden om een 3D-kopieerapparaat te maken. Zo'n apparaat bestaat uit een 3D-scanner, software om de uitvoer van de scanner om te zetten in een 3D-model van het gescande voorwerp, en een '3D-printer' die 3D-modellen daadwerkelijk kan produceren (bijvoorbeeld met technieken als stereolithografie).

Op bovenstaande gebieden liggen nog talloze andere algoritmische vragen. Dit geldt evenzeer voor gebieden als de robotica, de *computer graphics* en *virtual reality*, de medische technologie, en de moleculaire biologie. Deze grote verscheidenheid aan toepassin-

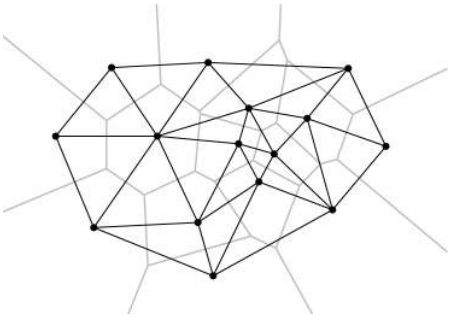
gen maakt, samen met de prachtige geometrische structuren en de elegante algoritmische technieken die gebruikt worden, de computationele geometrie tot een bijzonder boeiend vakgebied.

Natuurlijk is het ontwerp van geometrische algoritmen niet voorbehouden aan de computationele geometrie; ook onderzoekers uit de toepassingsgebieden zelf zijn op dit gebied actief. Het is interessant te ontdekken dat de vraagstellingen die optreden in de verschillende toepassingen vaak sterk gerelateerd zijn. Hier ligt een belangrijke rol van de computationele geometrie: het ontwikkelen van generieke oplossingen, die bruikbaar zijn in verschillende toepassingen. Daarnaast zou de computationele geometrie een theorie moeten bieden die kan voorspellen welke oplossingen effectief zullen zijn in een bepaalde toepassing. Tot nu toe is de computationele geometrie vooral succesvol geweest in het eerste aspect: er zijn allerlei ingenieuze algoritmen en datastructuren ontwikkeld, gebaseerd op elegante generieke concepten en technieken. In het tweede aspect is er echter minder succes geboekt: de huidige theorie voorspelt niet goed welke oplossingen in de praktijk effectief zullen zijn. Soms werken de theoretische oplossingen in de praktijk uitstekend, maar soms blijken volgens de theorie inferieure oplossingen in de praktijk toch beter. Dit te verbeteren is in mijn ogen een van de belangrijkste uitdagingen voor het vakgebied. In het vervolg van mijn rede zal ik wat dieper ingaan op de problemen met de huidige theorie. Daarna zal ik een onderzoeksrichting schetsen die er op gericht is de voorspellende waarde van de theorie te verbeteren.

De efficiëntie van algoritmen

Zoals we al eerder opmerkten is de kern van de algoritmiek gelegen in het ontwerp van efficiënte algoritmen. De vraag die hierbij direct naar voren komt is: wanneer is een algoritme efficiënt? Of beter: hoe kunnen we de efficiëntie van een algoritme kwantificeren, zodat we algoritmen met elkaar kunnen vergelijken? Het kwantificeren van de efficiëntie van een algoritme wordt in de algoritmiek de analyse van een algoritme genoemd. Het analyseren van een algoritme is een krachtig hulpmiddel om de bruikbaarheid ervan in de praktijk te voorspellen. Er is echter een paar problemen met de manier waarop deze analyse meestal gebeurt. Voor het gemak zal ik me bij de bespreking van de huidige manier van analyseren beperken tot de looptijd, dat wil zeggen, de tijd die een algoritme nodig heeft. Voor andere aspecten zoals geheugengebruik gelden dezelfde principes.

Stel dat we de looptijd van een algoritme dat een verzameling getallen sorteert willen analyseren. Een van de eerste observaties is dat de looptijd af zal hangen van het aantal getallen: het algoritme zal meer tijd nodig hebben om een grote dan om een kleine verzameling te sorteren. Daarom wordt de looptijd van een algoritme geanalyseerd als functie van n, het aantal elementen in de invoer. Maar zelfs als het aantal invoerelementen vastligt, dan nog kan de looptijd variëren: misschien zal een sorteeralgoritme sneller werken als de invoerverzameling toevallig al bijna gesorteerd is. En hier komen we bij een belangrijk aspect van de traditionele analyse van algoritmen: wat geanalyseerd wordt is het slechtste geval dat kan optreden. Met andere woorden, voor de looptijd bij een bepaalde waarde van n wordt gekeken naar de maximale tijd die het algoritme nodig heeft over alle mogelijke invoeren ter grootte n. Dit betekent dat men niet voor onplezierige verassingen kan komen te staan: het algoritme zou voor sommige invoeren sneller kunnen zijn dan de analyse voorspelt, maar nooit langzamer. In diverse toepassingen is een dergelijke garantie van groot belang. Blijft over de vraag: hoe kwantificeren we (bij een gegeven waarde van n) de looptijd voor de slechtste invoer? We zouden hiertoe, gesteld dat we de slechtste invoer kennen, het algoritme daadwerkelijk uit kunnen voeren op die invoer. Een nadeel hiervan is dat het algoritme dan eerst geïmplementeerd moet worden. Een fundamenteler bezwaar is dat de uiteindelijke tijd die we meten afhankelijk zal zijn van bijvoorbeeld de gekozen programmeertaal, de compiler, en de computer waarop we werken; allemaal zaken die niets met het algoritme van doen hebben. Daarom wordt de looptijd op een meer abstracte manier gekwantificeerd: in plaats van de echte tijd te meten, wordt het aantal stappen geteld dat het algoritme doet. Iets preciezer: wat geteld wordt is het aantal elementaire operaties — optellingen, vermenigvuldigingen, vergelijkingen, et cetera — dat het algoritme uitvoert. Dit kunnen we doen voor een algoritme zonder het te implementeren. Overigens wordt om de analyse werkbaar te houden, het aantal elementaire operaties niet exact geteld; er wordt alleen gekeken naar het groeigedrag als functie van n. We schrijven dan bijvoorbeeld dat de looptijd van een algoritme $O(n)$ is, als de looptijd lineair groeit met n. Op deze versimpeling, die de voorspellende waarde van de analyse ook negatief kan beïnvloeden, wil ik hier echter niet verder in gaan.



Figuur 4 Het Voronoidiagram (grijs) en de Delaunaytriangulatie (zwart) van een verzameling punten in het vlak

De rol van de geometrie

Bij de analyse van een algoritme wordt uitgegaan van het slechtste geval dat kan optreden. Dat is wenselijk, omdat men op die manier niet voor onplezierige verassingen komt te staan. Vooral bij ruimtelijke data lopen we hier echter tegen een probleem op. Vaak treedt het slechtste geval daar namelijk op bij hypothetische invoeren, die in de praktijk nooit voor zullen komen. Op dat moment is de voorspellende waarde van de analyse nihil.

Laat ik dit illustreren aan de hand van een voorbeeld uit de *virtual reality* (VR). Stel dat we een computermodel hebben van, zeg, het Hoofdgebouw van de Technische Universiteit Eindhoven. Met behulp van een VR-systeem kunnen we dan virtueel door het Hoofdgebouw lopen. Hiervoor zal het systeem moeten detecteren wanneer we ergens tegenop botsen; anders zouden we door muren of voorwerpen heen kunnen lopen. Nu zijn 3D-modellen meestal opgebouwd uit een groot aantal kleine driehoeken. Voor een gedetailleerd model van het hele Hoofdgebouw zijn al gauw miljoenen driehoeken nodig. Het expliciet testen van al die driehoeken om te detecteren of we ergens tegenop botsen zal het systeem veel te traag maken. Om dit proces te versnellen is het dan ook nodig om het model op een geschikte manier op te slaan. Dit kan bijvoorbeeld met behulp van een zogenaamde *binary space partition* (BSP). Hierbij wordt het model eerst in tweeën gedeeld met een vlak. De twee delen worden elk vervolgens weer in tweeën gedeeld, en dit gaat zo door tot de uiteindelijke deelgebieden nog maar enkele driehoeken bevatten. Om te kijken of bijvoorbeeld mijn virtuele voet ergens tegenaan schopt, wordt eerst gekeken in welke deelgebieden mijn voet zich bevindt. Vervolgens hoeven alleen de driehoeken uit die gebieden getest te worden.

Een van de vragen die deze aanpak oproept is: in hoeveel gebieden moeten we het model opdelen om te garanderen dat elk gebied nog maar enkele driehoeken van het mo-

del bevat? Het aantal gebieden is namelijk bepalend voor de hoeveelheid geheugen die we nodig hebben om de BSP op te slaan. Traditioneel gaat de computationele geometrie hier als volgt mee om. Net als bij de analyse van de looptijd wordt er gekeken naar het aantal benodigde gebieden als functie van n, het aantal driehoeken waaruit het model is opgebouwd. Paterson en Yao [7] hebben bewezen dat het altijd mogelijk is om BSP te construeren, waarvoor het aantal gebieden kwadratisch is in n. Daarnaast hebben ze laten zien dat het voor sommige verzamelingen van n driehoeken onvermijdelijk is dat het aantal gebieden kwadratisch is. Met andere woorden: elk mogelijk constructie-algoritme zal in het slechtste geval een kwadratisch aantal gebieden opleveren. Dat betekent dat het algoritme van Paterson en Yao optimaal is wat betreft het gedrag in het slechtste geval. Voor de traditionele computationele geometrie is daarmee de kous af.

Helaas is een kwadratisch aantal gebieden in de praktijk onwerkbaar. Deze analyse lijkt dus aan te geven dat BSP's onbruikbaar zijn. Toch worden ze in de praktijk gebruikt. Wat is er mis?

Het probleem is gelegen in het voorbeeld dat laat zien dat een kwadratisch aantal gebieden onvermijdelijk is. Figuur 7 laat dit voorbeeld zien. Het bestaat uit lange dunne driehoeken — zo lang en dun dat ze er in het plaatje als lijnen uitzien — die in een soort gridpatroon geplaatst zijn. Daarbij moet het grid ook nog eens op een speciale manier krom getrokken worden, zonder dat de driehoeken krom getrokken worden. (In vaktermen: het grid ligt op een *ruled surface*.) Het moge duidelijk zijn dat dit voorbeeld niets te maken heeft met verzamelingen driehoeken die gebouwen modelleren. Vandaar ook dat BSP's in de praktijk vaak prima werken. We hebben dan wel een algoritme nodig waarvan het gedrag in de praktijk beduidend beter is dan het gedrag op het bovenstaande gridvoorbeeld. Helaas kan de traditionele analyse dat gedrag in de praktijk niet voorspellen.

De conclusie is dat de traditionele slechtstegeval analyse voor ruimtelijke data vaak geen relatie heeft tot de efficiëntie van een algoritme in de praktijk. Dit betekent niet dat de prachtige algoritmen en concepten uit de computationele geometrie waardeloos zijn. Het betekent wel dat het onduidelijk is of die algoritmen inderdaad beter zijn dan andere algoritmen die als minder efficiënt terzijde worden geschoven. Het negatieve effect is helaas nog sterker. Omdat efficiëntie wordt afgemeten aan de hand van het gedrag voor het slechtste geval, zal het ontwerp van een algoritme er op gericht zijn het slechtste geval zo snel mogelijk op te lossen. Als het slechtste geval in de praktijk nooit optreedt, dan betekent dit vaak dat het algoritme nodeloos ingewikkeld wordt.

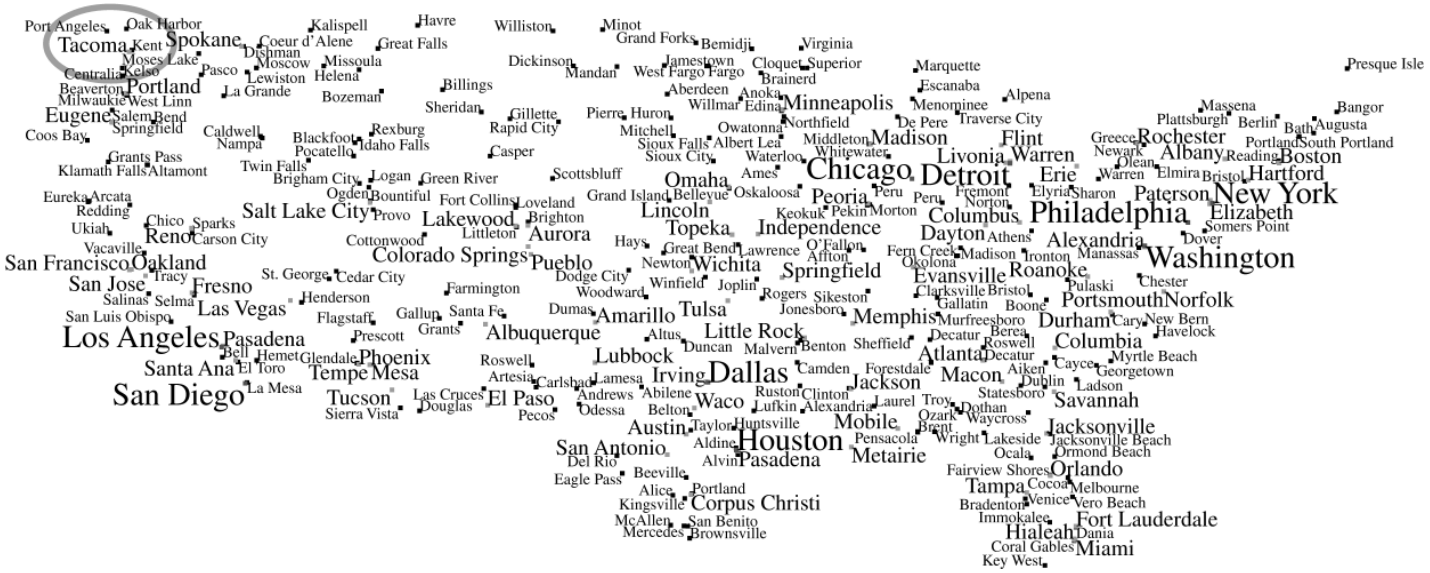
I/O- en caching-gedrag

Eerder hebben we gezien dat bij de analyse van de looptijd van een algoritme wordt gekeken naar het aantal elementaire operaties dat het algoritme uitvoert. Daarbij wordt geen onderscheid gemaakt tussen de operaties: een optelling telt even zwaar als een vermenigvuldiging, of elke andere elementaire operatie. Voor veel operaties is dat redelijk. Maar als de data waarmee het algoritme moet werken niet in het interne geheugen past maar op de harddisk staat, dan is er een duidelijke uitzondering: het inlezen en het wegschrijven van data. Dergelijke I/O-operaties kunnen al gauw meer dan 100.000 keer zoveel tijd kosten als bijvoorbeeld het vermenigvuldigen van twee getallen die al in het interne geheugen staan. Dit verschil zal in nieuwe computersystemen waarschijnlijk nog verder toenemen, omdat de afname van de tijd voor I/O-operaties geen gelijke tred houdt met de toename aan rekensnelheid — de nadruk bij het ontwikkelen van nieuwe harddisks ligt meer op het vergroten van de opslagcapaciteit dan op het verlagen van de *latency*.

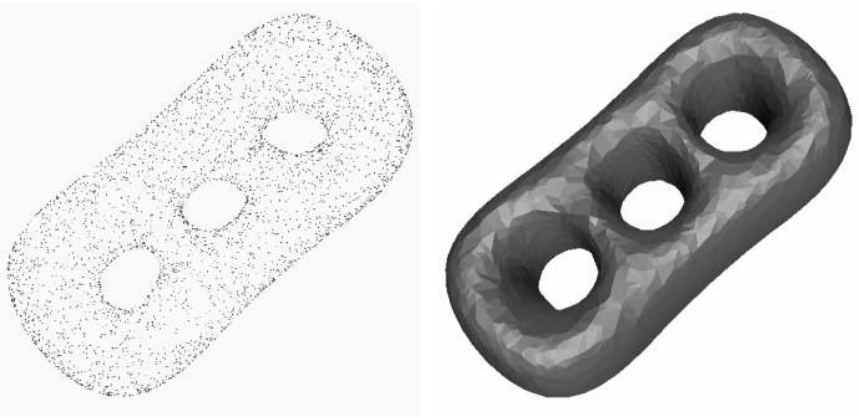
Dit alles betekent dat voor data sets die op disk staan, de looptijd van een algoritme vaak veel meer bepaald wordt door het aantal I/O-operaties dan door de rekentijd. De traditionele analyse houdt hier geen rekening mee. Het gevolg hiervan is ook dat er met dit aspect bij het ontwerp van algoritme geen rekening wordt gehouden. Overigens treedt het bovenstaande probleem ook op (maar dan in mindere mate) als de data wel in het interne geheugen past. In dat geval heeft het zogenaamde *caching-gedrag* vaak een grote invloed op de looptijd van een algoritme. Later zal ik hier nog wat uitgebreider op terug komen.

Beter voorspellen, efficiëntere algoritmen

In de vorige sectie hebben we gezien dat de traditionele analyse van geometrische algoritmen tekortschiet in het voorspellen van hun



Figuur 5 Automatisch gelabelde kaart met de belangrijkste steden in de VS (zie [5])



Figuur 6 Een puntenwolk in 3D en een reconstructie van het bijbehorende object [1]

efficiëntie in de praktijk. Hierdoor gaat de vergelijking van algoritmen vaak mank: het algoritme dat volgens de theoretische analyse het meest efficiënt is, hoeft dat in de praktijk helemaal niet te zijn. De tekortschietende analyse heeft ook een negatieve invloed op het ontwerp van algoritmen: doordat de efficiëntie op de verkeerde manier gemeten wordt, zal ook de aandacht bij het ontwerp naar de verkeerde dingen uitgaan. Maar het kan beter.

Geometrie-gevoelige analyse

Ruimtelijke data kunnen sterk variëren in vorm en verdeling over de ruimte. Deze aspecten kunnen van grote invloed zijn op de looptijd van een algoritme. Het gevolg is dat de slechtste-gevallooptijd gedomineerd wordt door extreme vormen en verdelingen die in de praktijk niet voorkomen. Een manier om dit te voorkomen is extra eisen op te leggen aan de invoer. De eisen moeten zo zijn dat invoeren die in de praktijk voorkomen aan deze eisen voldoen, terwijl hypothetische invoeren uitgesloten worden. Men spreekt dan van *realistische invoermodellen* [1]. Een mogelijkheid is om eisen op te leggen aan de vorm van de voorwerpen, bijvoorbeeld door te verbieden dat de invoer extreem lange en dunne voorwerpen bevat. Deze eis sluit de hypothetische invoer in figuur 8 uit. Een andere mogelijkheid is om eisen te stellen aan de verdeling van de voorwerpen over de ruimte. Natuurlijk moet dit soort eisen precies gedefinieerd worden om in de wiskundige analyse van de looptijd gebruikt te kunnen worden.

De extra eisen aan de invoer kunnen op twee manieren gebruikt worden. Ten eerste in de analyse van een algoritme. Bij de bepaling van de looptijd worden dan de invoeren die niet aan de eisen voldoen uitgesloten. Ten tweede kunnen ze gebruikt worden bij het algoritme-ontwerp. We kunnen ons dan richten op algoritmen die efficiënt zijn voor invoeren die aan de eisen voldoen; het gedrag op andere invoeren is van geen belang. We zouden dus zelfs algoritmen kunnen ontwerpen die niet langer correct werken als er niet aan de eisen voldaan is. Hoewel dit in sommige gevallen de algoritmen misschien zou kunnen versimpelen, is dit een onwenselijke situatie. Er is namelijk vaak een grijs gebied van gevallen die niet aan de eisen voldoen, maar waarvan het niet direct duidelijk is dat ze in de praktijk nooit zullen optreden. Ik pleit dan ook voor een iets andere aanpak.

In plaats van strikte eisen te stellen aan de parameters die de geometrische eigenschappen — de vorm van de voorwerpen en hun verdeling over de ruimte — van de invoer beschrijven, kunnen we die parameters ook meenemen in de analyse. Met andere woorden: we analyseren niet alleen het gedrag van het algoritme als functie van de invoergrootte n, maar ook als functie van een of meer geometrie-parameters. Met een dergelijke verfijnde analyse kunnen we voorspellen onder welke omstandigheden een algoritme goed zal werken. Dit heeft als bijkomend voordeel dat we de beste oplossing voor een bepaalde toepassing kunnen kiezen op grond van de typische waarden van de geometrie-parameters in die toepassing.

Deze aanpak heeft bijvoorbeeld voor BSP's al tot succes geleid. Er is namelijk een bijzonder eenvoudige manier om een BSP te construeren die bestaat uit $O(\lambda^2 n)$ gebieden, waarbij n het aantal driehoeken in de invoer is en λ de zogenaamde dichtheid van de verzameling driehoeken [2]. (De dichtheid van een verzameling voorwerpen zegt iets over de verdeling van de voorwerpen over de ruimte; als er veel relatief grote voorwerpen dicht opeen zijn gepakt, dan is de dichtheid hoog.) Nu zal de dichtheid van een verzameling driehoeken die een enkele kamer modelleert, ongeveer hetzelfde zal zijn als de dichtheid van een verzameling driehoeken die een gebouw van vele verdiepingen modelleert. Dat betekent dat de BSP voor deze toepassing een lineair opschaalgedrag heeft — beter kunnen we ons niet wensen. Deze verfijnde analyse geeft dus verklaring voor het feit dat BSP's in de praktijk vaak uitstekend werken. Het geeft ook aan dat het gebruik van BSP's waarschijnlijk af te raden is voor toepassingen waar de dichtheid groot is.

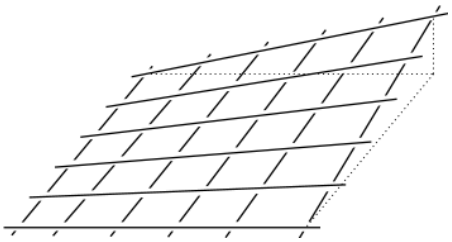
Ook op enkele andere gebieden, zoals het plannen van bewegingen van robots, zijn er al veelbelovende resultaten geboekt. Toch moet er nog veel gebeuren om tot een algemene theorie op dit gebied te komen.

I/O-efficiënte en cache-oblivious algoritmen

Een tweede probleem met de traditionele analyse is dat er geen rekening mee wordt gehouden dat I/O-operaties (lezen van en schrijven naar disk) vele malen langer duren dan operaties op data in het interne geheugen. We zouden I/O-operaties dan ook eigenlijk apart moeten tellen. Dit is echter eenvoudiger gezegd dan gedaan. Om dit fenomeen en mogelijke oplossingen hiervoor beter te begrijpen, moeten we eerst wat dieper ingaan op de werking en architectuur van moderne computer systemen.

Als een algoritme bepaalde data nodig heeft, wordt er eerst gekeken of die data al in het interne geheugen staan. Als dit niet het geval is, dan zal de data van disk moeten worden gelezen. Omdat lezen van disk zo'n kostbare operatie is, wordt er dan niet een enkele waarde ingelezen, maar een heel blok data; het lezen van een heel blok kost namelijk weinig extra tijd ten opzichte van het lezen van een enkele waarde. Het blok blijft daarna een tijd in het interne geheugen staan. Als er hiervoor geen ruimte beschikbaar is, zal een van de andere blokken uit het interne geheugen naar disk teruggeschreven moeten worden, om plaats te maken voor het nieuwe blok. Welke data bij elkaar in een blok gezet worden, en welke blokken in het interne geheugen gehouden worden, is van grote invloed op het aantal I/O-operaties dat een algoritme doet: het is voordelig om data die kort na elkaar nodig is zoveel mogelijk bij elkaar in een blok te zetten, en om blokken die binnenkort weer nodig zijn in het interne geheugen te houden.

Dit alles wordt normaal gesproken verzorgd door het operating systeem. De huidige operating systemen hebben allerlei geavanceerde strategieën om het geheugen-



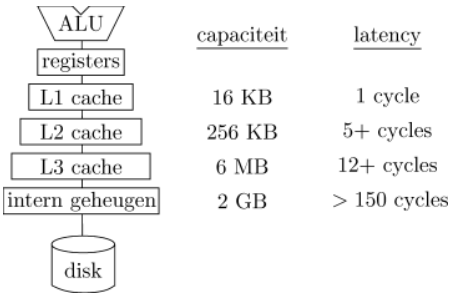
Figuur 7 Elke lijn stelt een lange, dunne driehoek voor. Voor deze verzameling 'driehoeken' is een kwadratisch aantal gebieden nodig, om te garanderen dat elk gebied slechts enkele driehoeken levert

management zo te doen, dat algoritmen zo min mogelijk I/O-operaties hoeven te doen. Over het algemeen werken deze strategieën vrij goed. Omdat het aantal I/O-operaties dat een algoritme doet afhangt van het operating system, is het echter niet mogelijk om dat aantal van te voren te analyseren. Daar komt bij dat de strategieën niet toegesneden zijn op een specifiek algoritme. Vaak is het dan ook mogelijk om winst te boeken door het algoritme zelf het geheugen-management te laten verzorgen. Dat betekent natuurlijk wel extra werk voor de algoritme-ontwerper: die zal expliciet moeten aangeven welke data bij elkaar in een blok geplaatst moet worden, en welke blokken in het geheugen gehouden dienen te worden. Algoritmen die zelf hun geheugen-management verzorgen worden *external-memory algoritmen* of ook wel *I/O-efficiënte algoritmen* genoemd. Een bijkomend voordeel van deze algoritmen is dat het aantal I/O-operaties geanalyseerd kan worden, zodat hun looptijd beter voorspeld kan worden.

Een mooi voorbeeld van wat er bereikt kan worden als het I/O-gedrag tijdens het algoritme-ontwerp de nodige aandacht krijgt is te vinden in [1]. Hier wordt de zogenaamde *randomized incremental construction* (RIC) toegepast: een veelgebruikte ontwerptechniek in de computationele geometrie. RIC-algoritmen zijn vaak verrassend simpel, en toch zeer efficient. Het kenmerk van deze algoritmen is dat ze op een zeer chaotische manier door de data heen springen; het voordeel daarvan is dat de kans op een ongunstige volgorde bij het behandelen van de data bewijsbaar vrijwel nihil is. (Helaas voert het hier te ver om dieper in te gaan op de mooie wiskundige theorie die aan RIC-algoritmen ten grondslag ligt.) Dit chaotische gedrag levert echter een groot probleem op voor het I/O- en caching gedrag. Het *operating system* zal namelijk niet goed kunnen voorspellen welke blokken data in het interne geheugen gehouden moeten worden. Het gevolg is *thrashing*: het operating system raakt de kluts kwijt en is bijna alleen nog bezig met bloktransport tussen intern- en extern geheugen. In [1] werd bijvoorbeeld geëxperimenteerd met het berekenen van 3D-Delaunaytriangulaties. Thrashing trad daar al op bij zo'n 250.000 punten. Door het I/O-gedrag expliciet mee te nemen in het ontwerp, konden ze echter een variant van het RIC-algoritme ontwikkelen die data sets van meer dan 2,5 miljoen punten aankan. Kumar [6] was zelfs in staat 3D-Delaunaytriangulaties van 300 miljoen punten te berekenen.

Voor toepassingen van de computationele geometrie waar de data op disk staat — bij geografische informatiesystemen is dit bijvoorbeeld vaak het geval — is het dus belangrijk om I/O-efficiënte algoritmen te ontwerpen. Maar zelfs als de data in het interne geheugen past, spelen vergelijkbare zaken een rol. Lezen van en schrijven naar het interne geheugen kost namelijk nog steeds duidelijk meer tijd dan bijvoorbeeld het optellen van twee getallen, en daarom zit er tussen het 'gewone' interne geheugen en de ALU (de *arithmetic/logic unit*, die het rekenwerk uitvoert) een kleiner, extra snel geheugen: de *cache*. Net als tussen disk en intern geheugen gebeurt het data transport tussen intern geheugen en cache in blokken — die overigens kleiner zijn dan de blokken voor data transport tussen disk en intern geheugen — en kan de hoeveelheid data transport tussen intern geheugen en cache een grote invloed op de looptijd hebben . In feite is de situatie nog ingewikkelder: tegenwoordig zijn er vaak al drie niveaus van caches. Figuur 8 geeft deze hiërarchische geheugenarchitectuur schematisch weer, met de bijbehorende opslagcapaciteit en latency (de 'wachttijd' voordat data transport van het betreffende geheugen of cache kan beginnen) voor de Intel Itanium 2.

Helaas is het over het algemeen onmogelijk om een algoritme zelf de caches te laten aansturen; deze staan meestal puur onder controle van de hardware. En gesteld dat dit wel mogelijk zou zijn, dan is het, zachtjes uitgedrukt, nogal gecompliceerd om algoritmen te ontwerpen die expliciet het geheugenmanagement voor het interne geheugen en alle caches doen. Het lijkt er dus op dat we ons moeten beperken tot het ontwerp van algoritmen die het aantal disk-operaties proberen te minimaliseren. En zelfs dit brengt al de nodige problemen met zich mee. Om het geheugenmanagement goed te kunnen doen, zullen algoritmen namelijk moeten weten hoeveel data er in een blok past, en hoeveel blokken er in het interne geheugen passen. Dit zijn parameters die afhankelijk zijn van de hardware



Figuur 8 Geheugenhiërarchie van de Intel Itanium 2 (1.5 GHz). Capaciteit van de disk kan oplopen tot 1 TB, met *latencies* van enkele milliseconden (wat overeenkomt met meer dan 106 *clock cycles*.)

waarop het algoritme wordt uitgevoerd. Wat erger is, zelfs op dezelfde hardware staan ze niet vast. Zo zal een algoritme het interne geheugen vaak moeten delen met andere processen; welk deel voor het algoritme beschikbaar is, zal (zelfs tijdens het uitvoeren van het algoritme) kunnen variëren.

Recent is er echter een bijzonder elegante manier gevonden om deze problemen aan te pakken, namelijk door middel van zogenaamde *cache oblivious-algoritmen* [4]. Deze algoritmen gaan uit van een geheugenmodel met maar twee lagen: het 'gewone' geheugen en de cache. Deze lagen kunnen model staan voor extern geheugen en intern geheugen, voor intern geheugen en L3-cache, voor L3-cache en L2-cache, et cetera. De algoritmen proberen het data transport tussen deze twee lagen te minimaliseren. De cruciale eigenschap die ze hebben, is dat hun werking onafhankelijk is van de blokgrootte of de grootte van de cache. Het lijkt dan onmogelijk om het geheugenmanagement te optimaliseren, maar dit is verrassend genoeg niet het geval: onder enkele milde aannames over de werking van de cache, blijkt het mogelijk om algoritmen te ontwerpen waarvan men kan bewijzen dat hun caching gedrag uitstekend is, ongeacht de blok-grootte en de grootte van de cache. En dit laatste heeft een geweldig mooie consequentie: als men kan bewijzen dat een algoritme goed is voor het abstracte geheugenmodel met twee lagen, dan is het algoritme automatisch goed voor alle lagen in de geheugen-hiërarchie. Er zijn nog maar enkele van zulke cache oblivious-algoritmen en datastructuren ontworpen voor ruimtelijke data. Hier liggen dus nog enorme uitdagingen.

Toekomstig onderzoek

De computationele geometrie is een prachtig vakgebied, waarin een scala aan ingenieuze algoritmen, elegante concepten, en algemene technieken zijn ontwikkeld. Helaas is er nog

geen goede theorie die de bruikbaarheid van deze oplossingen in de praktijk goed voorspelt. Ik hoop dat onderzoek langs de hierboven geschetste lijnen daar verbetering in kan brengen. Dergelijk onderzoek zal de komende jaren dan ook een van de zwaartepunten van het werk in mijn groep zijn.

Een van de onderwerpen waar we ons hierbij op richten zijn zogenaamde *multifunctionele geometrische datastructuren*. Dit zijn geometrische datastructuren waar verschillende soorten ruimtelijke data in opgeslagen kunnen worden, en waar verschillende typen zoekoperaties efficiënt op kunnen worden uitgevoerd. Dit in tegenstelling tot datastructuren zoals die meestal in de computationele geometrie worden ontworpen; die zijn geschikt voor één soort data en voor één type zoekoperaties. Om te kunnen bewijzen dat multi-functionele geometrische datastructuren ook efficiënt kunnen zijn, is een verfijnde analyse in termen van geometrieparameters noodzakelijk.

Hoewel het doel van dit onderzoek is om de theorie te verbeteren, zal er ook het nodige experimentele onderzoek moeten gebeuren. Er zal immers geverifieerd moeten worden in hoeverre de nieuwe theorie inderdaad beter aansluit bij de praktijk.

Een van hulpmiddelen voor het onderzoek naar geometrische datastructuren zal een *benchmarking systeem* worden, dat we momenteel aan het opzetten zijn. Een verzameling benchmarks voor geometrische datastructuren is er op dit moment niet, en dat maakt het vergelijken van verschillende experimentele resultaten erg lastig. Ik hoop dan ook dat onze benchmarks het experimentele onderzoek naar geometrische datastructuren een stuk vooruit kunnen helpen.

In veel toepassingen waar ruimtelijke data een rol spelen — denk bijvoorbeeld aan computerspellen en allerlei vormen van simulaties — is beweging een belangrijk aspect. Het efficiënt omgaan met bewegende objecten is dan ook een ander onderwerp waar we de komende jaren de nodige aandacht aan willen geven zijn. Tenslotte willen we ons onderzoek op het gebied van de geografische informatiesystemen (en meer in het bijzonder de geautomatiseerde cartografie) verder uitbouwen.

De bovenstaande onderwerpen geven de grote lijnen in het onderzoek in de groep aan. In de computationele geometrie zijn echter nog veel intrigerende problemen waar men aan kan puzzelen, en ik vind het belangrijk dat daar ruimte voor is. Dergelijk onderzoek scherpt het verstand, levert soms inzichten op die op andere plaatsen plotseling van pas blijken te komen, en — niet onbelangrijk — het is gewoon erg leuk en stimulerend om af en toe eens een uitstapje te maken.

Referenties

1 N. Amenta, S. Choi, G. Rote, 'Incremental construction con BRIO', *Proc. 19th ACM Symp. on Computational Geometry*, (2003), p. 211–219.

2 M. de Berg, 'Linear size binary space partitions for uncluttered scenes', *Algorithmica*, 28, (2000), p. 353–366.

3 M. de Berg, M. Katz, F. van der Stappen, and J. Vleugels, 'Realistic input models for geometric algorithms', *Algorithmica*, 34, (2002), p. 81–97.

4 S. van Dijk, *Genetic Algorithms for Map Labeling*, PhD thesis, Utrecht University, 2001.

5 M. Frigo, C.E. Leisorson, H. Prokop, and S. Ramachandran, 'Cache-oblivious algorithms', *Proc. 40th Symp. Foundations Comp. Science (FOCS)*, (1999), p. 285–298.

6 P. Kumar, *Clustering and Reconstructing Large Data Sets*, PdD thesis, Stony Brook University, 2004.

7 M. S. Paterson and F. F. Yao, 'Efficient binary space partitions for hidden-surface removal and solid modeling', *Discr. Comput. Geom.*, 5 (1990), p. 485–503.

8 *The Digital Michelangelo Project*. Stanford University. See http://www-graphics.stanford.edu/projects/mich

9 *The Stanford 3D Scanning Repository*. Stanford University. See http://graphics.stanford.edu/data/3Dscanrep