

Toepassingen van meetkunde in de informatica

Project: Bepaling van het Dichtste Puntenpaar

Wolfgang Möllmann & Robbe Degrève

25 mei 2018

1 Beschrijving van opstelling van puntenverzameling

Voor het opstellen van een input-file met een aantal gegeven parameters om willekeurige punten te creëren gebruiken we de functie "makeRandom". Deze functie zal de volgende parameters als input vragen:

1. inFile: De puntenverzameling zal in dit bestand weggeschreven worden, indien het bestand niet bestaat zal het aangemaakt worden.
2. alg: het nummer van het algoritme dat gebruikt moet worden: 1 (eenvoudige algoritme), 2 (eerste variante van het doorlooplijnalgoritme) of 3 (tweede variante van het doorlooplijnalgoritme)
3. dim: de dimensie van de punten $M \geq 2$
4. size: het aantal punten N

Deze functie heeft als taak om een inputbestand op te stellen met willekeurige coördinaten. De functie zal de gegeven parameters eerst neerschrijven in de inFile. Daarna gebeurt een initialisatie van `Random r`. Daarna zullen we per regel van het bestand itereren van 0 tot en met de dimensie en voor iedere dimensie een willekeurige waarde neerschrijven in het bestand. De willekeurige coördinaten kunnen waardes aannemen tussen $[0, 5[$.

Voor de *worst-case* puntenverzameling voor de eerste variant van het doorlooplijnalgoritme voor $M = 2$ hebben we een javafunctie `makeWorstCase`. In deze functie maken een puntenverzameling aan van één kolom, waarbij alle punten dezelfde x-waarde hebben. De y-waarde speelt hierbij geen rol. In onze rij-implementatie sorteren we de punten met dezelfde x-waarden volgens de y-waarde. Het tweede punt zal dus enkel vergelijken met het punt boven hem. Het derde punt zal nadien vergelijken met het eerste en het tweede punt. Uiteindelijk zal het laatste (onderste) punt zal dus met alle punten vergelijken. Dit zal ons uiteindelijk $\frac{(n+1)}{2}$ vergelijkingen geven. De complexiteit zal dus $O(n^2/2)$ zijn.

Algorithm 1 De javafunctie makeRandom

Input: *inFile*, *alg*, *dim*, *size*

infile.println(*alg*)

if *dim* < 2 **then**

print Dimension needs to be greater or equal to 2.

 Exit

end if

Random *r* = new Random()

Iterator < *Double* > *i* = *r*.doubles(*size* * *dim*, 0.0, 5.0).iterator()

*String**outString* =

while *i.hasNext()* **do**

outString =

for *j* = 0; *j* < *dim*; *j*++ **do**

outString += String.format(Locale.US, "%17.16f ", *i.next()*)

end for

infile.println(*outString*)

end while

infile.flush()

infile.close()

2 Opdracht 1: Hoog-niveau beschrijving van verscheidene algoritmes

2.1 Brute-force algorithm

2.1.1 hoogniveau beschrijving

Algorithm 2 Bereken het dichtste puntenpaar met brute-force

Input: rij : Array met N punten (gesorteerd naar stijgende x-coördinaat)

$d = +\infty$

$dpp1 = 0, dpp2 = 0$

$currentDist = 0$

for i to $length(rij) - 1$ **do**

for $j = i + 1$ to $length(rij)$ **do**

$currentDist = \text{calculate_dist}(rij[i], rij[j])$

if $currentDist < d$ **then**

$dpp1 = rij[i]$

$dpp2 = rij[j]$

$d = currentDist$

end if

end for

end for

return $dpp1, dpp2, d$

2.2 Variant 1 algoritme

2.2.1 hoogniveau beschrijving

Algorithm 3 Bereken het dichtste Puntenpaar volgens variant 1

Input: *rij*: Array met N punten (gesorteerd naar stijgende x-coördinaat)
 $d = +\infty$
 $dpp1 = 0, dpp2 = 0$
 $currentDist = 0$
for $i = 1$ to $length(rij)$ **do**
 for $j = i - 1$ to 0 **do**
 if $rij[i].x - rij[j].x > d$ **then**
 break
 end if
 $currentDist = calculate_dist(rij[i], rij[j])$
 if $currentDist < d$ **then**
 $dpp1 = rij[i]$
 $dpp2 = rij[j]$
 $d = currentDist$
 end if
 end for
end for
return $dpp1, dpp2, d$

2.2.2 rekencomplexiteit

We zullen dus nagaan dat de rekencomplexiteit van dit algoritme gelijk is aan $O(N \cdot K_{gem})$. We kunnen in het hoogniveau algoritme zien dat de eerste for-lus zal itereren over $N-1$ punten. Voor elk van die punten zullen we dus de afstand bepalen van de punten die binnen de verticale strook V . Indien we het K_{gem} bepalen (uitgemiddeld over $i=1, \dots, N$) zullen we een goede schatting hebben hoeveel punten er worden vergeleken in iedere iteratie van de doorlooptijd. We rekenen bij K_{gem} ook het eerste element bij die een K zal hebben van 0 , dus kunnen we bij onze doorlooptijd iteratie het eerste punt ook meetellen. We kunnen dus concluderen dat de rekencomplexiteit van variant 1 gelijk zal zijn aan $O(N \cdot K_{gem})$.

2.3 Variant 2 algorithm

2.3.1 hoogniveau beschrijving

Algorithm 4 Bereken het dichtste Puntenpaar volgens variant 2

Input: *rij*: Array met N punten (gesorteerd naar stijgende x-coördinaat)

$d = +\infty$

$dpp1 = 0, dpp2 = 0$

$currentDist = 0$

t: gegevensstructuur waarin punten links van de doorlooplijn opgeslagen zijn, gesorteerd naar stijgende y-coördinaat

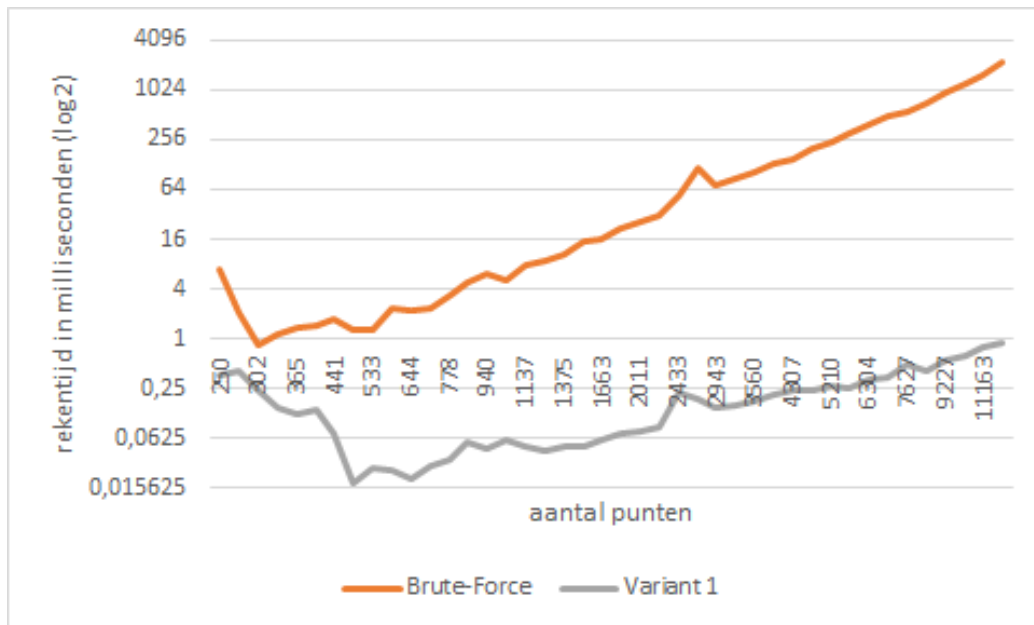
3 Grafieken

3.1 veronderstelling

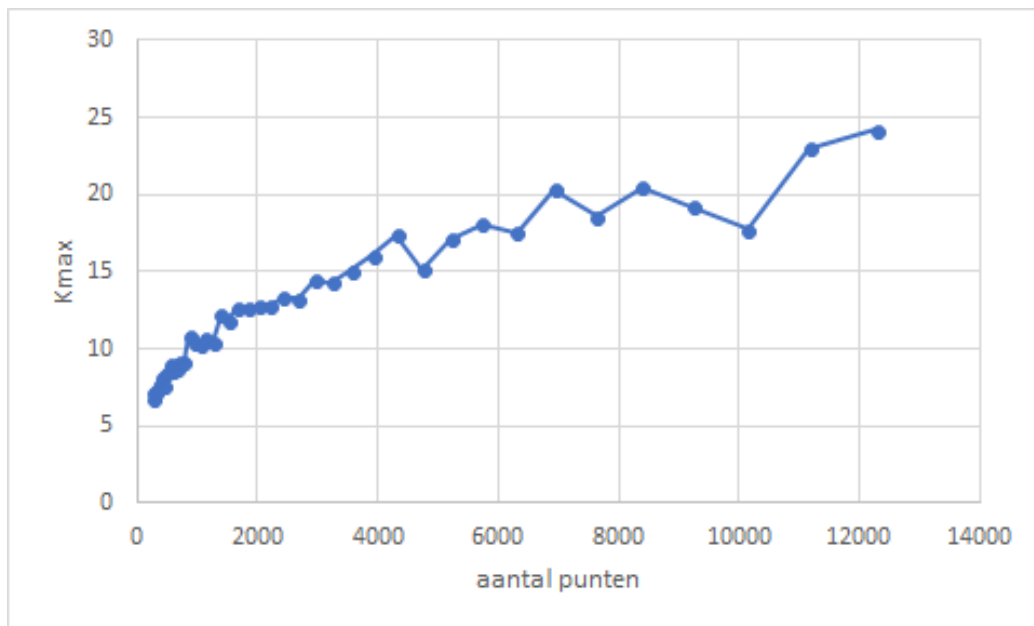
Indien we spreken over rekencomplexiteit bedoelen we de complexiteit van het algoritme zonder de rekentijd van het sorteren van de input. Bij tijdscomplexiteit maken we de som tussen rekentijd van het sorteren en de rekentijd van het algoritme zelf.

In Figuur 1 zullen we de rekentijden tussen het brute-force algoritme en variant1 vergelijken. We zullen op de x-as het aantal punten uitzetten. De gekozen aantal punten zijn logaritmisch bepaald met als basis 1.1 beginnend vanaf 302 tot en met 12279. Op de y-as zetten we de rekentijd uit in milliseconden. Hiervoor gebruiken we een logaritmische functie met basis 2. We kunnen concluderen uit deze plot dat variant 1 voor alle onze gekozen punten een efficiënter algoritme is. We kunnen ook zien dat de het brute-force algoritme een steiler verloop heeft, hieruit kunnen we afleiden dat voor zeer grote aantallen punten het verschil tussen de twee algoritmes steeds groter worden.

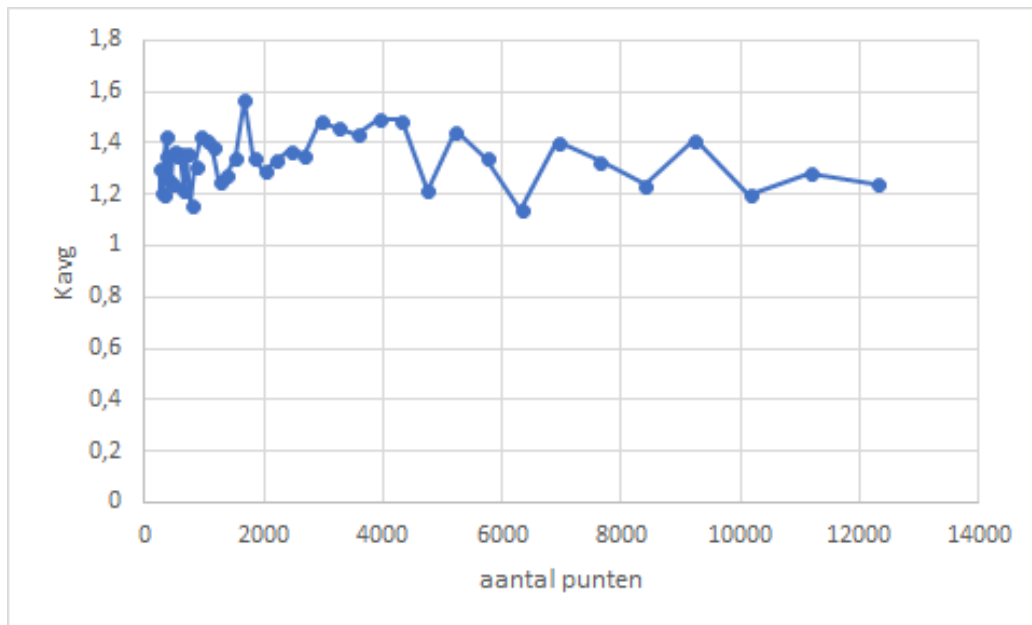
In figuur 2 zullen we het verband tussen de Kmax en het aantal punten plotten. Op de x-as zetten we het aantal punten uit van 250 tot 12279 met een logaritmische functie met basis 1,1. Op de y-as zetten we de Kmax uit die in dit geval zal gaan van een minimum van 2 tot en met een maximum van 25. We kunnen op de plot zien dat de grafiek Kmax een licht stijgend karakter heeft.



Figuur 1: De plot van de rekestijden tussen het brute-force algoritme en variant 1



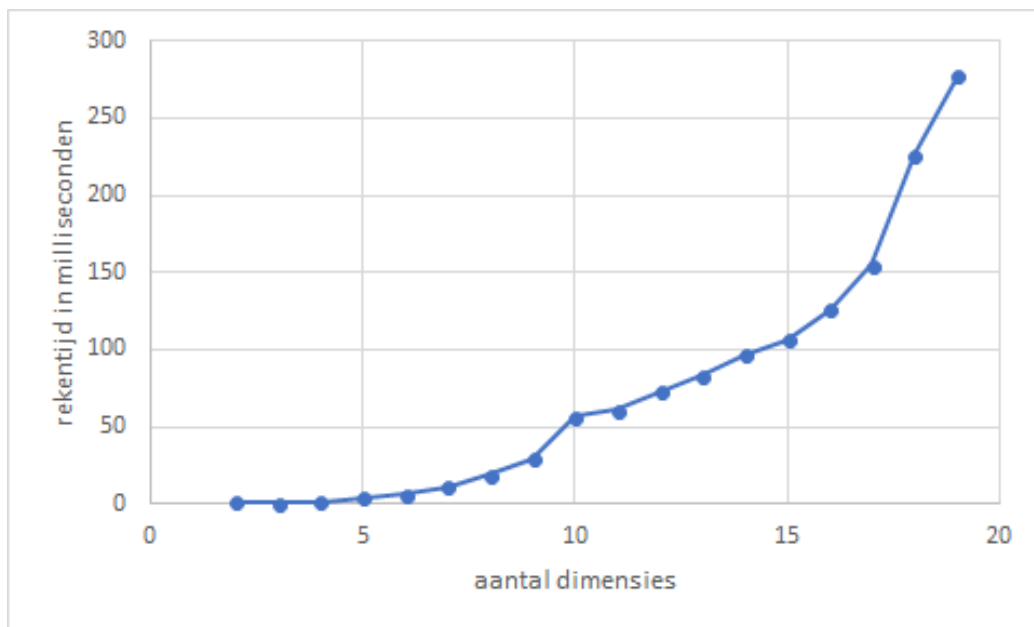
Figuur 2: De plot tussen het aantal punten en Kmax



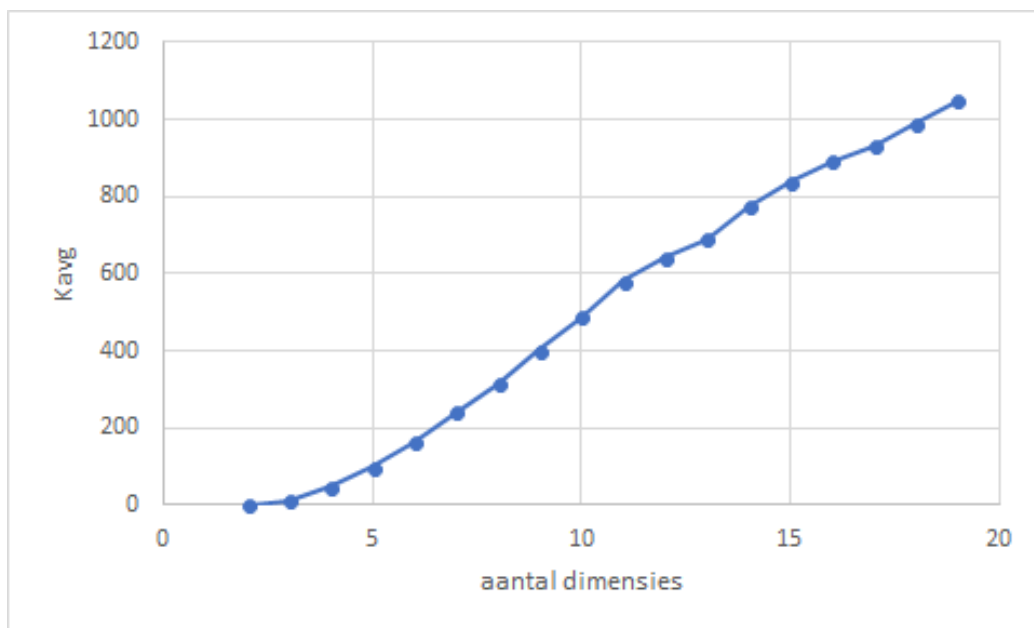
Figuur 3: De plot tussen het aantal punten en Kavg

In figuur 3 zullen we het verband tussen de Kavg en het aantal punten plotten. Op de x-as zetten we het aantal punten uit van 250 tot 12279 met een logaritmische functie met basis 1.1. Op de y-as zetten we de Kmax uit die in dit geval zal gaan van een minimum van 0,1480 tot en met een maximum van 2,14186. Op deze plot is duidelijk te herkennen dat Kavg ongeveer constant zal blijven ongeacht het aantal punten. In ons experiment was deze constante rond 1,35.

In Figuur 4 zullen we de het aantal dimensies plotten in functie van de rekestijd voor variant 1. We zullen op de x-as het aantal dimensies uitzetten van 2 tot en met 19. Op de y-as zetten we de rekestijd uit in milliseconden. We zien duidelijk dat voor een stijgende M gaat het voordeel van het doorlooptijd verloren. Dit kunnen we eenvoudig bepalen aan de hand van de rekencomplexiteit en figuur 5. In figuur 5 zien we dat Kavg een lineair verband heeft met het aantal dimensies. We weten ook dat de rekencomplexiteit van variant 1 gelijk is aan $O(N \cdot Kavg)$. Hierdoor kunnen we dus afleiden dat de rekencomplexiteit ook zal stijgen in functie het aantal dimensies. TODO Waarom? Voor welke waarde van M is er geen voordeel meer? Waarom?



Figuur 4: De plot tussen het aantal dimensies en de rekeningtijd van variant 1 (voor 2500 punten)



Figuur 5: De plot tussen het aantal dimensies en Kavg (voor 2500 punten)