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 willekeurge 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 willekeurge coördinaten. De functie zal de gegeven parameters eerst neerschrijven in de inFile. Daarna gebeurd 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 willeukeurige coördinaten kunnen waardes aannemen tussen $[0, 5[$.

2 Opdracht 1: Hoog-niveau beschrijving van verscheidene algoritmes

2.1 Brute-force algorithm

2.1.1 hoogniveau beschrijving

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()
$StringoutString =$
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()

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**
    $currentDist$ = 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

2.3 Opmerking

Als we tijdens de behandeling van p_i tegenkomen dat buiten de verticale strook V ligt, heeft het zin om dit punt uit de zoekboom te verwijderen. Deze bewerking neemt $O(logK)$ elementaire operaties in beslag, en kan voor iedere keer dat het punt nogmaals in de horizontale strook terecht komt $O(logK)$ elementaire operaties (voor het zoeken) besparen. Door het verwijderen van deze elementen wordt K_{gem} ook veel kleiner, en dus worden er minder *boven* en *onder* bewerkingen uitgevoerd.
Het is niet nodig, maar zal het aantal bewerkingen wel verminderen.

2.4 worst-case puntenverzameling

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 rekencomplexiteit zal dus $O(n^2)$ zijn. Voor Kavg kunnen we concluderen dat per N het aantal kavg voor de worst-case gelijk zal zijn aan $\frac{N-1}{2}$. Kavg zal dus voor de worst-case in 2 dimesies een stijgend karakter hebben, wat in tegenstelling is tot willekeurige puntenverzamelingen in 2 dimensies (zie figuur 3). Dat is ook logisch. We hebben daarnet geconcludeerd dat de rekentijd van variant 1 van de worst-case zal toenemen in verband met het aantal punten. De rekencomplexiteit is gelijk aan $(N * K_{avg})$, hieruit kunnen we zien dat voor een stijgende rekencomplexiteit we ook een stijgende Kavg nodig zullen hebben. Kmax zal ook een stijgend karakter hebben. We kunne afleiden dat voor iedere N de Kmax steeds gelijk zal zijn aan $N - 1$. Voor de worst-case zal het laatste punt dat gecontroleerd moet worden de afstand berekenen met alle vorige punten.

2.5 Variant 2 algorithm

2.5.1 hoogniveau beschrijving

2.6 rekencomplexiteit

De behandeling van ieder punt p zal gemiddeld $O(logN)$ bewerkingen nodig hebben. We maken bij onze variant 2 gebruik van een gebalanceerde zoekboom. We weten dat alle nodige bewerkingen in $O(logK)$ operaties kunnen uitgevoerd worden in de boom. Het algoritme voegt bij elke iteratie het punt dat we aan het behandelen zijn toe aan de boom. Bij de behandeling van het laatste punt zullen we alle punten toegevoegd hebben aan de boom. Voor het laatste punt zal dus elke bewerking $log(N)$ tijd kosten.

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-coordinaat
for $i = 1$ to $length(rij) - 1$ **do**
  voegtoe$(t, rij[i-1])$
  $low = rij[i]$
  $low.y = low.y - d$
  $next = boven(t, low)$
  while $next \neq Null$ AND $next.y < rij[i].y + d$ **do**
    if $next.x < rij[i].x - d$ **then**
      verwijder$(t, next)$
      $next = boven(t, next)$
      continue
    end if
    $currentDist =$ calculate_dist($rij[i]$, $next$)
    if $currentDist < d$ **then**
      $dpp1 = rij[i]$
      $dpp2 = next$
      $d = currentDist$
    end if
    $next = boven(t, next)$
  end while
end for

3 Grafieken

Alle grafieken zijn achteraan te vinden.

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.

3.2 figuur 1

In Figuur 1 zullen we de rekentijden tussen het brute-force algoritme en variant 1 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 zal worden.

3.3 figuur 2

In figuur 2 zullen we het verband tussen de K_{max} en het aantal punten plotten voor 2 dimensies. 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 K_{max} 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 K_{max} een licht stijgend karakter heeft. We kunnen wel zien dat steeds K_{max} relatief klein zal blijven ten opzichte van het aantal punten.

3.4 figuur 3 en 4

In figuur 3 zullen we het verband tussen de K_{avg} en het aantal punten plotten voor 2 dimensies. 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 K_{avg} uit. Op deze plot is duidelijk te herkennen dat K_{avg} ongeveer constant zal blijven ongeacht het aantal punten. In ons experiment was deze constante 1.35. We kunnen concluderen dat voor willekeurige puntenverzameling in 2 dimensies de K_{avg} zeer klein zal zijn in vergelijkign met N.

In figuur 4 zullen we het verband tussen de K_{avg} en het aantal punten plotten voor 3 dimensies. 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 K_{avg} uit. Op deze plot zien we een groot contrast als we vergelijken met figuur 3. Terwijl wij bij figuur 3 spraken van een een relatief kleine Kavg voor een willekeurge aantal punten, hebben we hier een positief lineair verband. We zullen in drie dimensies significant meer afstanden moeten berekenen dan in twee dimensies. We kunnen dus zeggen dat vanaf 3 dimensies de efficiëntie van het algoritme al zal afnemen.

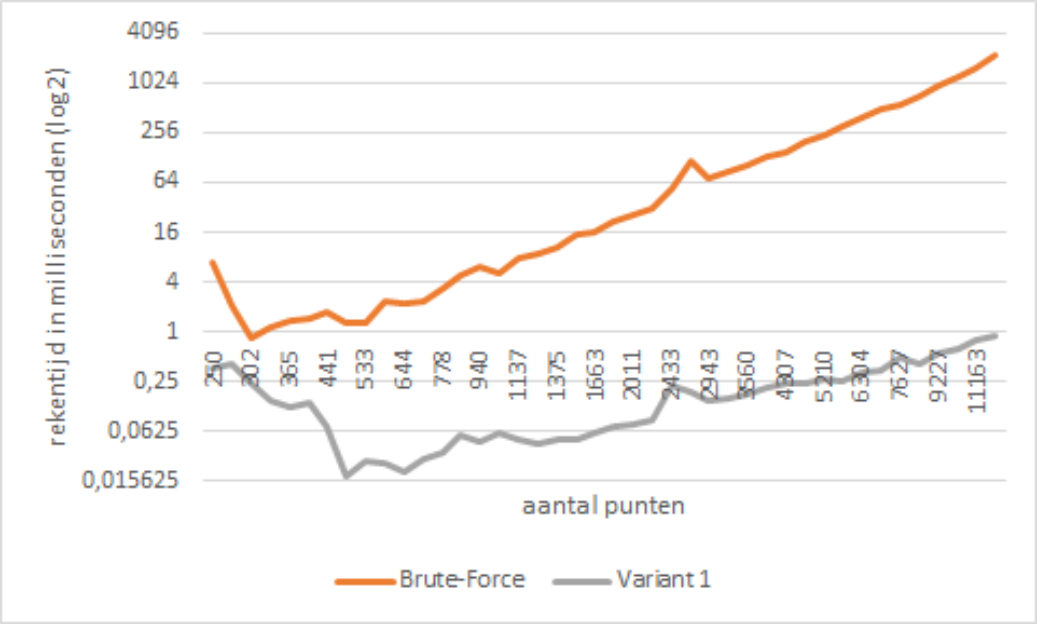
3.5 figuur 5 en figuur 6

In figuur 5 zullen we het verband tussen de K_{avg} en het aantal dimensies plotten. Op de x-as zetten we het aantal dimensies uit van 2 tot 19. Op de y-as zetten we de K_{avg} uit. Op deze plot is duidelijk te herkennen dat K_{avg} een positief lineair verband heeft met het aantal dimensies. Dit zal invloed hebben op de rekentijd van variant 1 voor hogere dimensies. In figuur 6 zullen we het aantal dimensies plotten met de rekentijd van variant 1 voor 2500 punten. We zullen op de x-as het aantal dimensies uitzetten van 2 tot en met 19. Op de y-as zetten we de rekentijd uit in milliseconden. We zien in figuur 6 ook dat voor hogere dimensies het voordeel van een doorlooplijn verloren gaat. we kunnen dit ook afleiden uit figuur 5 en de rekencomplexiteit van variant 1, namelijk $O(N * K_{avg})$. Omdat K_{avg} een positief lineair verband heeft met het aantal dimensies zal ook de rekentijd in hogere dimensies een positiefe trend vertonen.

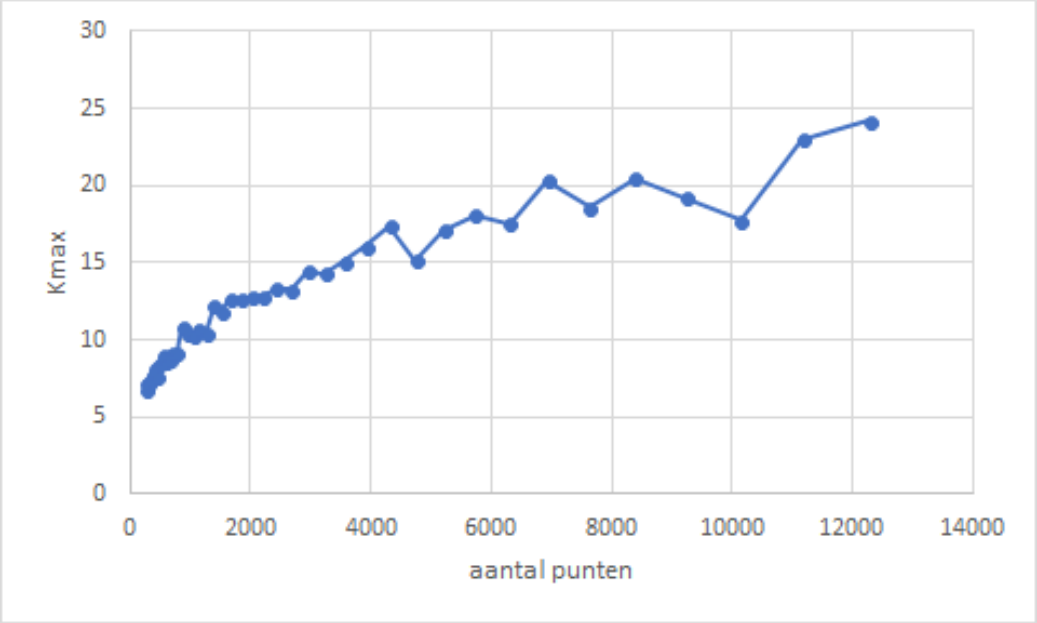
In figuur 3 en 4 kunnen we al zien dat bij dimensie 2 K_{avg} ongeveer constant blijft, terwijl bij dimensie 3 er een positief verband is tussen K_{avg} en het aantal punten. Hierbij gaat een deel van het voordeel verloren, maar het is nog steeds beter dan het *brute-force* algoritme, omdat we nog steeds uit 1 dimensie 'selecteren' voor de verticale strook. Het voordeel zal er dus altijd zijn, t.o.v. *brute-force*, maar zal veel kleiner worden voor stijgende M.

3.6 figuur 7

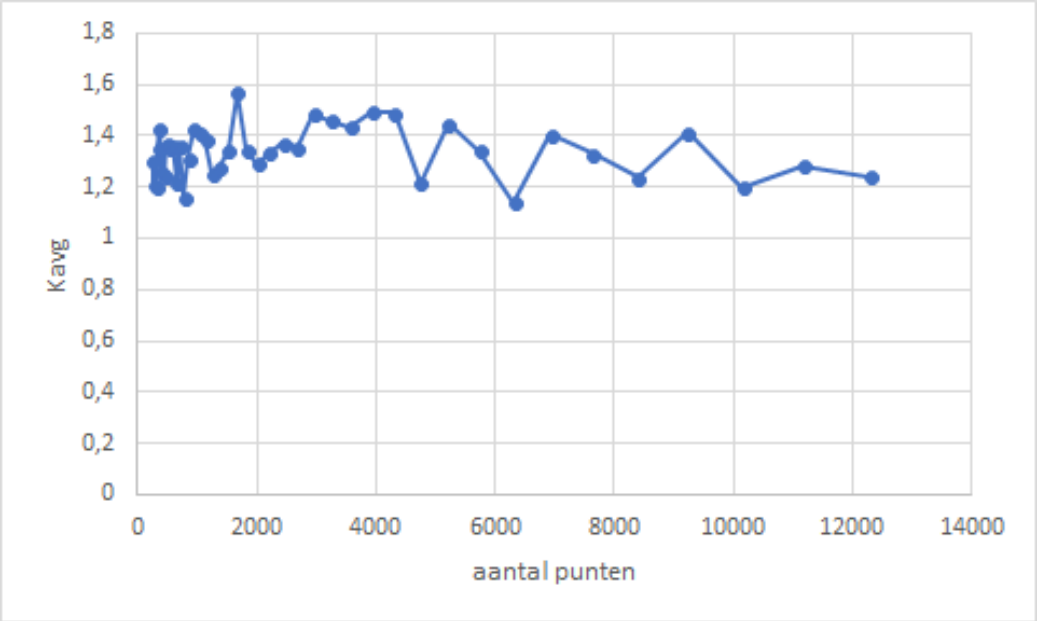
In figuur 7 zullen we het verband tussen het aantal dimensies en K_{max} plotten. Op de x-as zetten we het aantal dimensies uit van 2 tot 19. Op de y-as zetten we de K_{max} uit. We kunnen een duidelijk positief lineair verband tussen K_{max} en het aantal dimensies waarnemen.



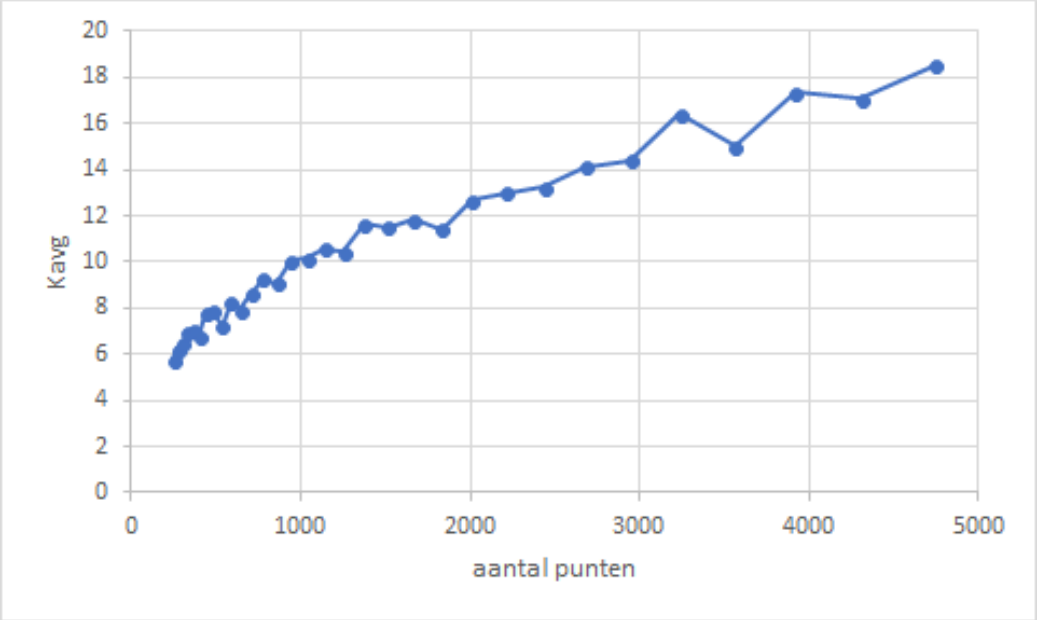
Figuur 1: De plot van de rekentijden tussen het brute-force algoritme en variant 1 voor 2 dimensies



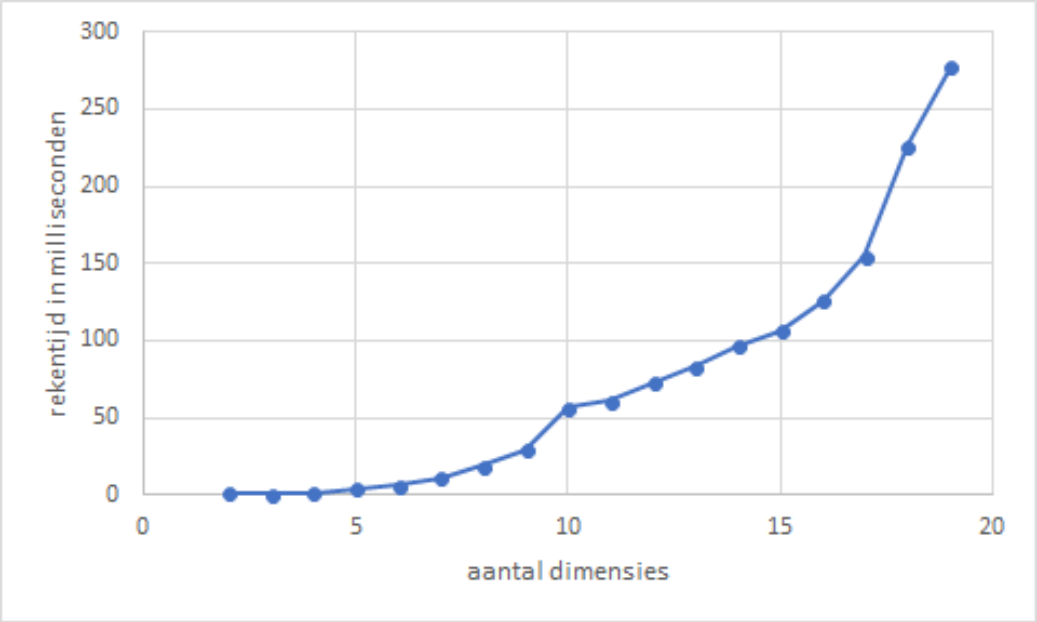
Figuur 2: De plot tussen het aantal punten en Kmax voor 2 dimensies



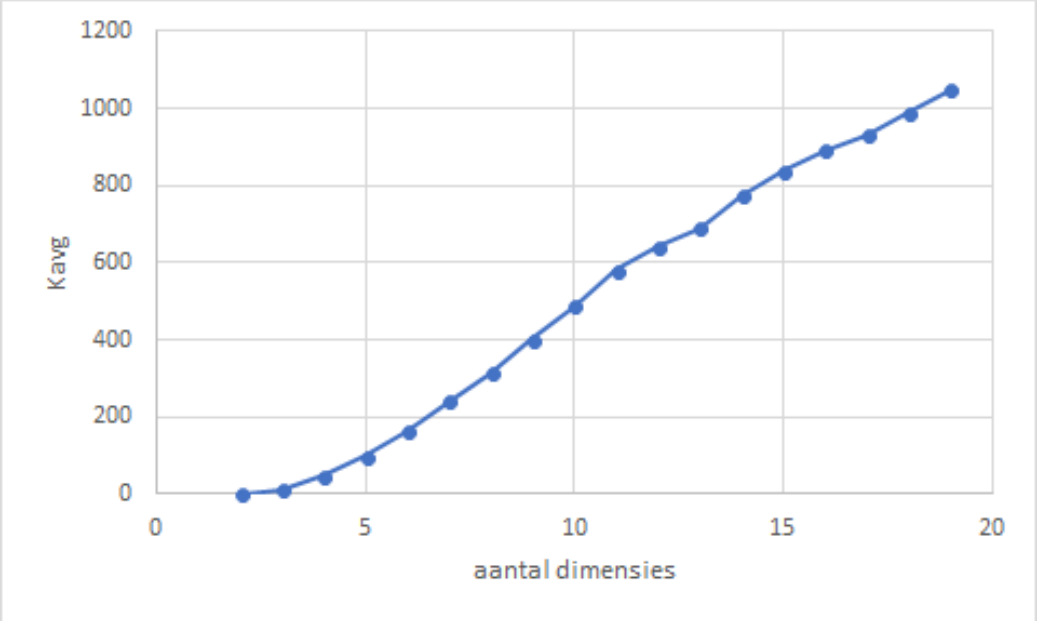
Figuur 3: De plot tussen het aantal punten en Kavg voor 2 dimensies



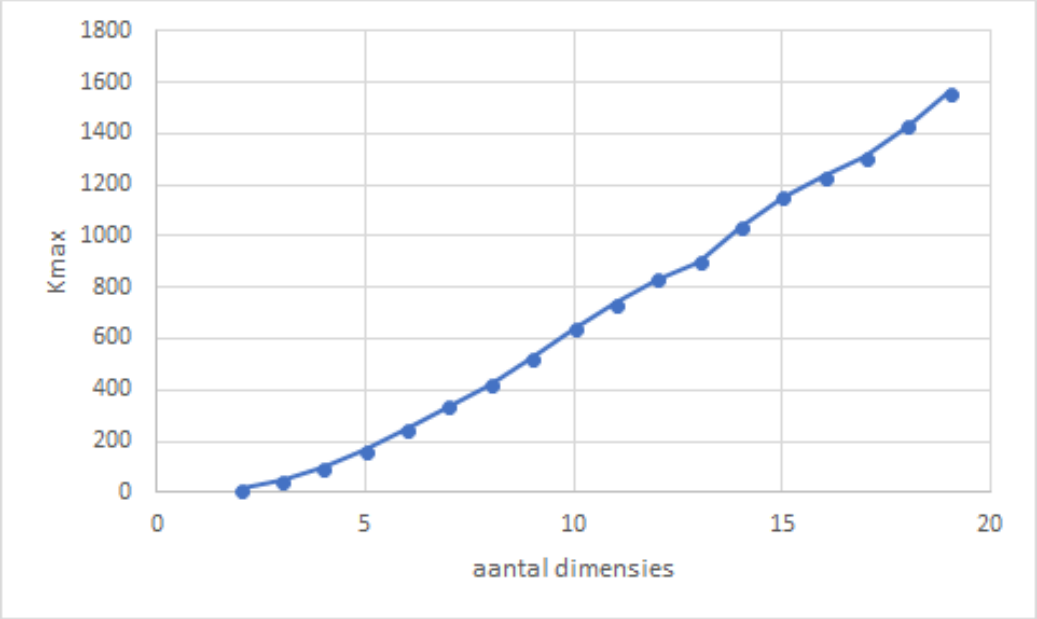
Figuur 4: De plot tussen het aantal punten en Kavg voor 3 dimensies



Figuur 5: De plot tussen het aantal dimensies en de rekentijd van variant 1 (voor 2500 punten)



Figuur 6: De plot tussen het aantal dimensies en Kavg (voor 2500 punten)



Figuur 7: De plot tussen het aantal dimensies en Kmax (voor 2500 punten)