Practicum: snijdende cirkels

Toepassingen van de meetkunde in de informatica

bestemming: Yvette Vanberghen

Niels De Bock & Mattias De Charleroy
2Ba Informatica

18 april 2013

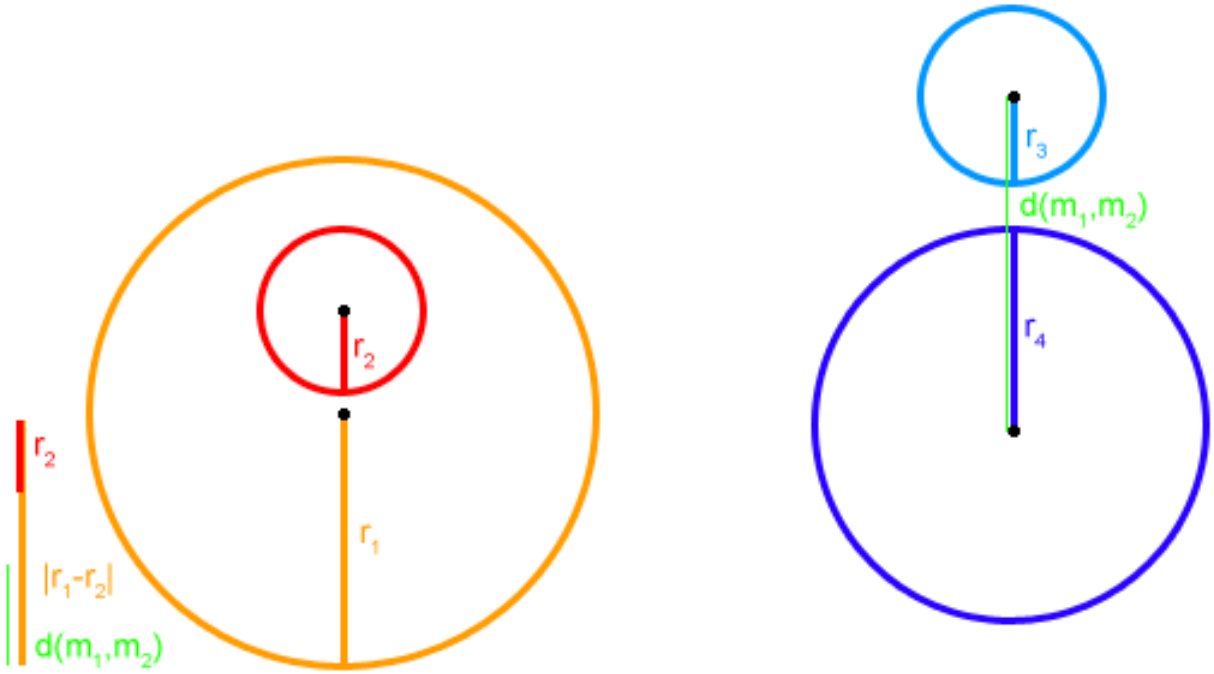
1. Beschrijving van de algoritmen

1.1 Algoritme 1

Algorithm 1 Bereken alle snijpunten van een verzameling cirkels

Input: *cirkels*: Een array met n cirkels
snijpunten $\leftarrow \emptyset$
for $i = 0 \rightarrow n - 1$ **do**
  for $j = i + 1 \rightarrow n - 1$ **do**
    voegToe(*snijpunten*, berekenSnijpunten(*cirkels*[i], *cirkels*[j]))
  end for
end for
return snijpunten

Dit algoritme zal elk paar cirkels één keer met elkaar vergelijken om snijpunten te berekenen, wat neerkomt op $\frac{n(n-1)}{2}$ vergelijkingen. Het berekenen van snijpunten gebeurt op de volgende manier: eerst doen we een snelle check om te bepalen of de twee cirkels daadwerkelijk snijden. Dit kan door de afstand tussen de middelpunten te vergelijken met de som van de stralen. Wanneer deze afstand groter is dan de som van de stralen snijden de cirkels niet. Een cirkel kan echter ook volledig in een andere cirkel liggen. Dit is wanneer de afstand (d) tussen de middelpunten kleiner is dan het lengteverschil tussen de stralen ($|r_1 - r_2|$). Wanneer twee cirkels gelijk zijn en dus samenvallen, zullen er geen snijpunten berekend worden. Dit is echter een randgeval. Wanneer blijkt dat de cirkels snijden, zullen we de snijpunten berekenen zoals beschreven in sectie 3.



Figuur 1: Niet snijdende cirkels

1.2 Algoritme 2

Algorithm 2 Bereken alle snijpunten van een verzameling cirkels met gebruik van een doorlooplijn

Input: *cirkels*: Een lijst met n cirkels
$snijpunten \leftarrow \emptyset$
$actief \leftarrow \emptyset$ (Een verzameling die de actieve cirkels bevat)
evenementen: Een lijst die zowel de x-coördinaat van het uiterste linkse als de x-coördinaat van het uiterste rechtse punt van elke cirkel bevat, gesorteerd op de x-waarde.
for each *evenement* in *evenementen* **do**
  if *evenement* is x-coördinaat van linkerpunt *cirkel* **then**
    for each *actieveCirkel* in $actief$ **do**
      voegToe($snijpunten$, berekenSnijpunten(*cirkel*, $actieveCirkel$))
    end for
    voegToe($actief$, *cirkel*)
  else if *evenement* is x-coördinaat van rechterpunt *cirkel* **then**
    verwijder($actief$, *cirkel*)
  end if
end for
return snijpunten

De doorlooplijn zal van links naar rechts het gebied van cirkels doorlopen. Wanneer de doorlooplijn op een cirkel komt, zal deze vergeleken worden met de andere actieve cirkels om snijpunten te zoeken. Vervolgens wordt deze ook toegevoegd aan de verzameling van actieve cirkels. Wanneer de doorlooplijn een cirkel helemaal heeft doorlopen, zal deze cirkel verwijderd worden uit de verzameling van actieve cirkels.
We gaan de doorlooplijn natuurlijk enkel laten stoppen op linker- of rechterpunten van de cirkels. Dit kunnen we bereiken door vooraf een lijst op te stellen met "evenementen", die linker- en rechterpunten van cirkels voorstellen en gesorteerd is op de x-coördinaten.

1.3 Algoritme 3

Algorithm 3 Bereken alle snijpunten van een verzameling cirkels met gebruik van een doorlooplijn, waarbij de actieve cirkels in een gesorteerde zoekboom geplaatst worden

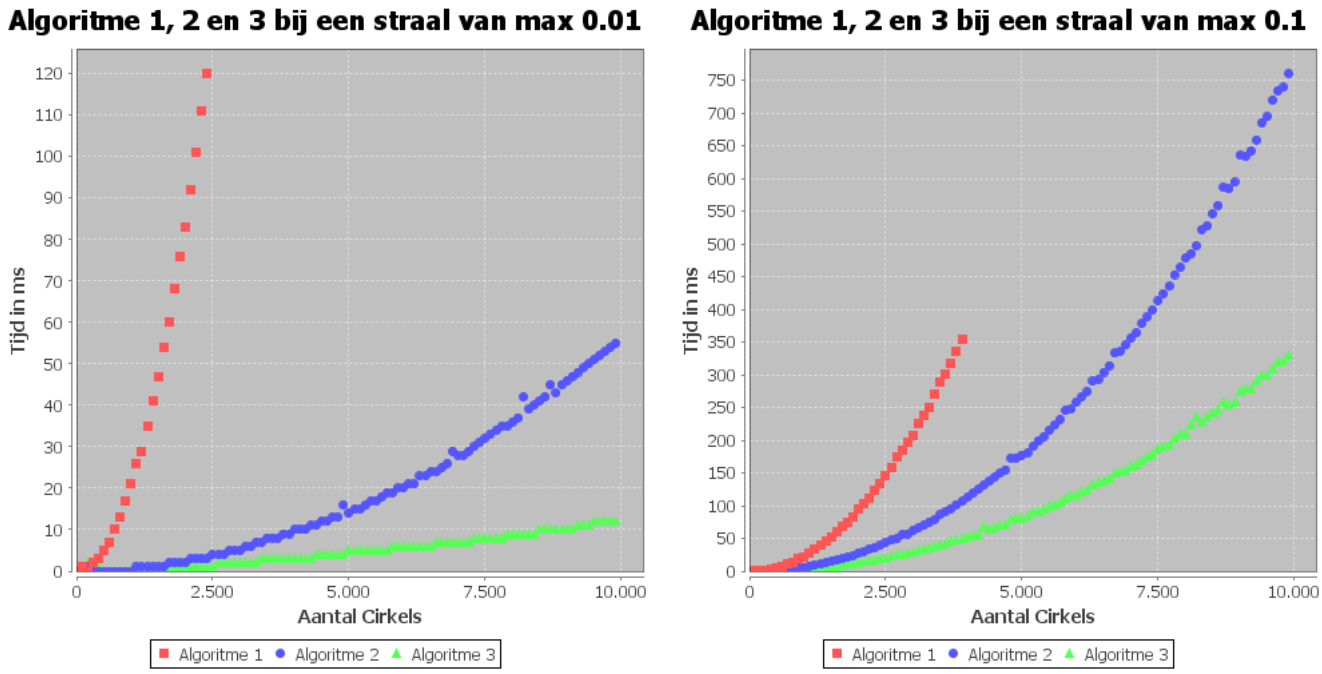
Input: *cirkels*: Een lijst met n cirkels
$snijpunten \leftarrow \emptyset$
$stralen \leftarrow \emptyset$ (Een gesorteerde lijst met stralen van actieve cirkels)
$actief \leftarrow \emptyset$ (Een zoekboom die de actieve cirkels bevat, gesorteerd op de y-coördinaat van hun middelpunt)
evenementen: Een lijst die zowel de x-coördinaat van het uiterste linkse als de x-coördinaat van het uiterste rechtse punt van elke cirkel bevat, gesorteerd op de x-waarde.
$maxStraal \leftarrow 0$ (Deze variabele houdt de straal van de grootste cirkel van de actieve cirkels bij).
for each *evenement* in *evenementen* **do**
  if *evenement* is x-coördinaat van linkerpunt *cirkel* **then**
    for each *actieveCirkel* in deelverzameling($actief$, *cirkel*.top.y - $maxStraal$, *cirkel*.bottom.y + $maxStraal$) **do**
      if *actieveCirkel*.bottom.$y \geq$ *cirkel*.top.y en *actieveCirkel*.top.$y \leq$ *cirkel*.bottom.y **then**
        voegToe($snijpunten$, berekenSnijpunten(*cirkel*, *actieveCirkel*))
      end if
    end for
    if *cirkel*.straal groter dan $maxStraal$ **then**
      $maxStraal \leftarrow$ *cirkel*.straal
    end if
    voegToe($actief$, *cirkel*)
    voegToe($stralen$, *cirkel*.straal)
  else if *evenement* is x-coördinaat van rechterpunt *cirkel* **then**
    verwijder($actief$, *cirkel*)
    verwijder($stralen$, *cirkel*.straal)
    if *cirkel*.straal is $maxStraal$ **then**
      $maxStraal \leftarrow pollFirst(stralen)$
    end if
  end if
end for
return snijpunten

De doorlooplijn zal werken zoals in *Algoritme 2*. Nu vergelijken we de nieuwe actieve cirkel echter niet met elke andere actieve cirkel om snijpunten te berekenen. Dit doen we door een zoekboom te gebruiken waarin we de actieve cirkels plaatsen, gesorteerd op de y-coördinaat van hun middelpunt. Verder gebruiken we ook een zoekboom waarin we de stralen van deze actieve cirkels zetten. Wanneer een nieuwe cirkel toegevoegd wordt, nemen we echter slechts een deelverzameling van de actieve cirkels. Deze deelverzameling bevat de cirkels waarvan het middelpunt minder ver ligt van het top- of het benedenpunt van de nieuwe actieve cirkel verlengd met de maximale straal. Van deze deelverzameling bekijken we enkel de cirkels waarvan een deel (van hun y-coördinaten) tussen de nieuwe actieve cirkel ligt. De maximale straal zal indien nodig aangepast worden bij het toevoegen of verwijderen van een cirkel uit de verzameling van actieve cirkels.

2. Experimenten

We hebben de experimenten uitgevoerd door grafieken te maken van de uitvoeringstijd ten op zichte van het aantal cirkels. De grafieken werden opgesteld door willekeurige cirkels te genereren met een meegegeven maximum straal. We namen de gemiddelde uitvoertijd over 10 tests van deze gegenereerde cirkels. Eerst stelden we de maximale straal in op 0.01 en vervolgens verhoogden we deze systematisch. Hierdoor konden we de evolutie van de algoritmen evalueren voor verschillende stralen. Het aantal keer dat de methode "berekenSnijpunten" werd opgeroepen, werd ook gemeten, alsook het aantal snijpunten.

2.1 Relatief kleine stralen

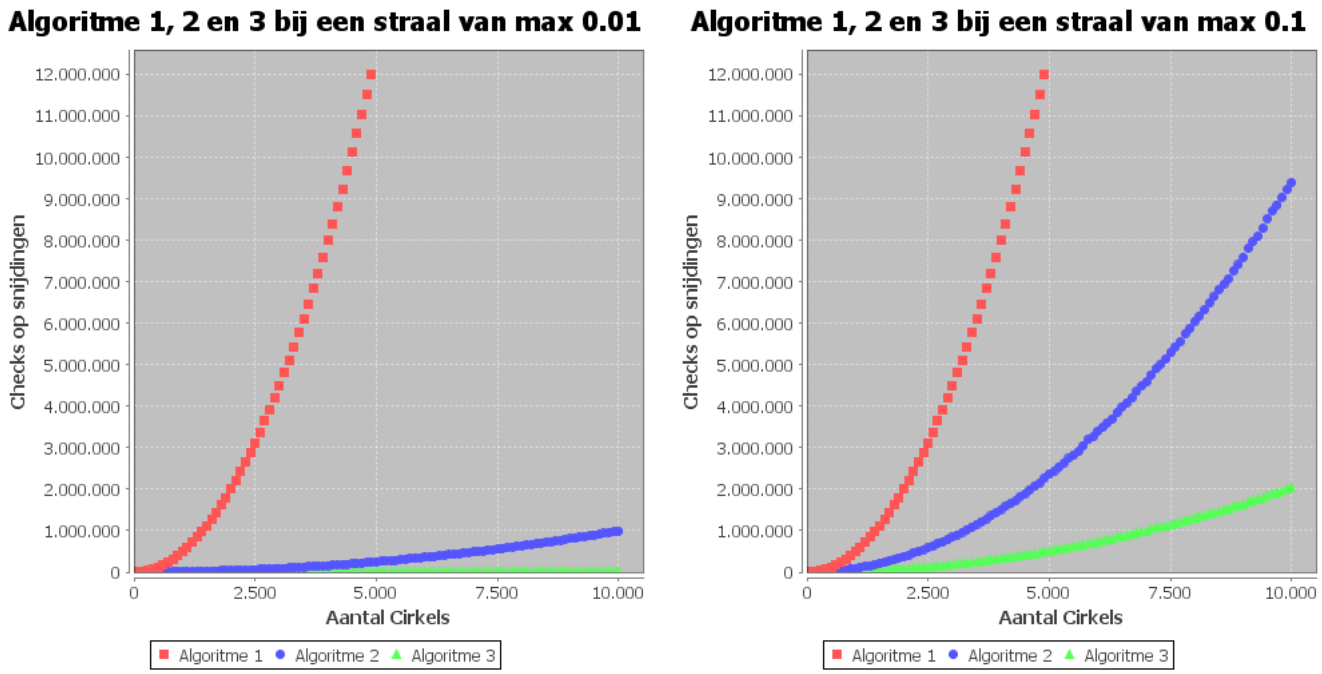


Figuur 2: Uitvoeringstijd van algoritme 1, 2 en 3 bij kleine stralen

In Figuur 2 kunnen we zien dat *Algoritme 1* duidelijk het slechtst presteert. Dit komt omdat het algoritme snijpunten zoekt door elke cirkel te vergelijken met elke andere cirkel. Hierdoor heeft het algoritme een tijdscomplexiteit van $O(n^2)$, aangezien het altijd $\frac{n(n-1)}{2}$ vergelijkingen uitvoert.

Algoritme 2 heeft een 'slechtste-geval' rekencomplexiteit $O(n^2)$. In dit geval zijn de stralen echter zeer klein waardoor het veel beter presteert. Dit is te wijten aan het feit dat er weinig cirkels tegelijk actief zijn en er dus weinig cirkels vergeleken worden wanneer een nieuwe cirkel wordt toegevoegd. Figuur 2 laat zien dat *Algoritme 2* inderdaad veel beter presteert dan *Algoritme 1* voor kleine stralen.

Algoritme 3 presteert beter met kleine stralen dan *Algoritme 2* aangezien dit algoritme ook rekening houdt met de y-coördinaten van de cirkels. De rekencomplexiteit is $O((n+S)log_2(n))$, waarbij n het aantal cirkels is en S het aantal snijpunten. Bij grotere stralen zal dit algoritme dus vertragen omdat er dan gemiddeld gesproken meer overlappende cirkels en dus ook meer snijpunten zijn.

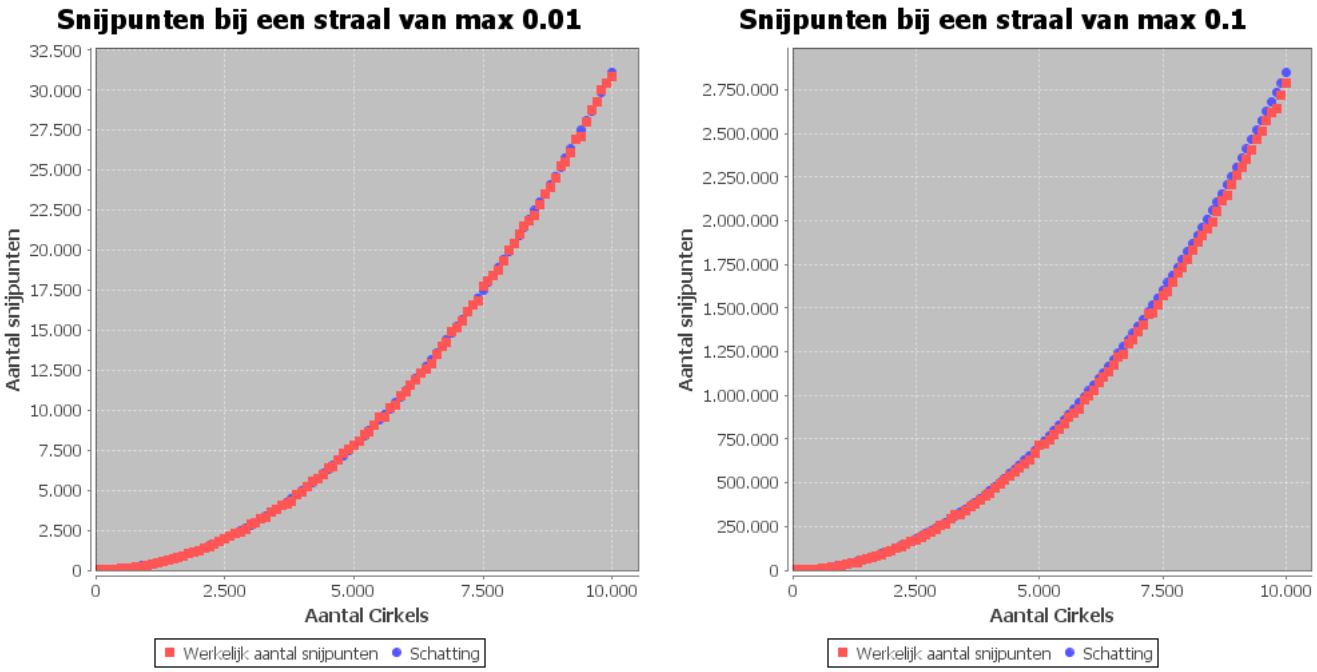


Figuur 3: Aantal keer dat "berekenSnijpunten" wordt opgeroepen door algoritme 1, 2 en 3 bij kleine stralen

Er is een discrepantie tussen het aantal keer dat de "berekenSnijpunten" methode wordt opgeroepen en de uitvoertijd. Dit is te wijten aan de complexe (gesorteerde) datastructuren die nodig zijn voor *Algoritme 2* en *3*. Dit is te zien in Figuur 3.

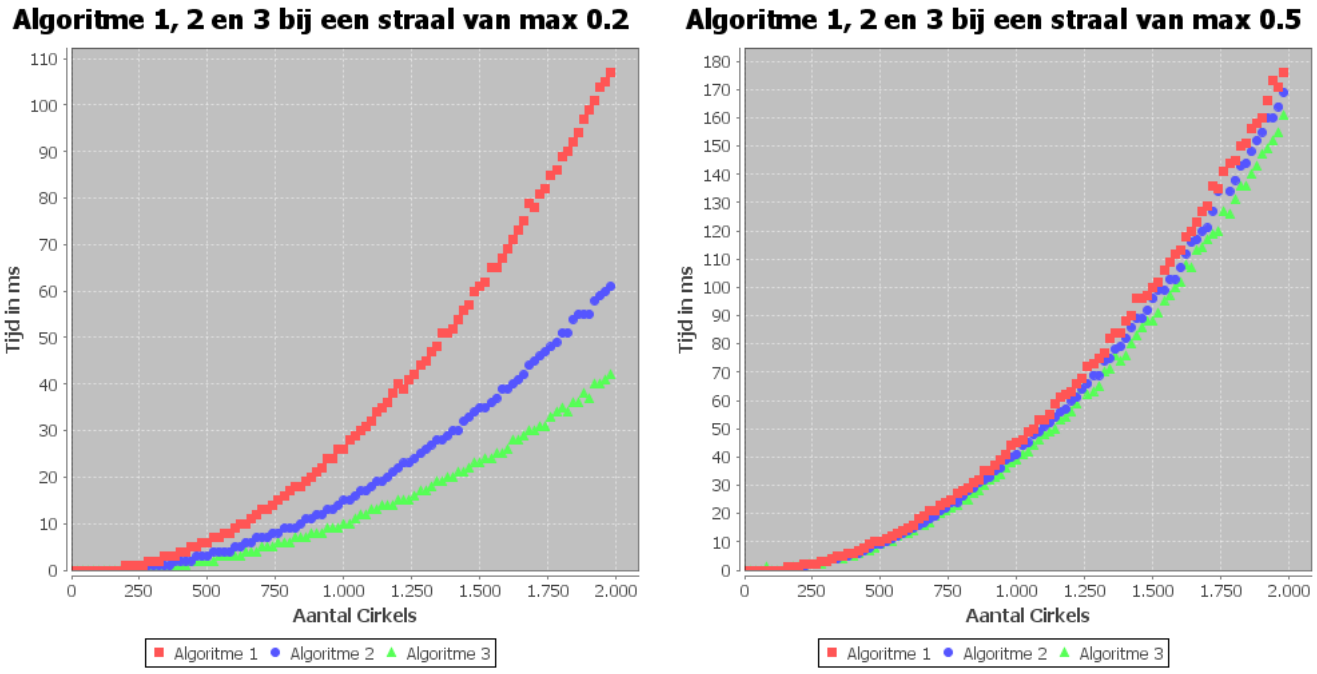
Zoals eerder vermeld, kunnen we in Figuur 4 zien dat het aantal snijpunten zeer snel stijgt voor éénzelfde aantal cirkels indien de straal groter wordt. We zijn erin geslaagd om het aantal snijpunten zeer goed te benaderen alvorens het algoritme uitgevoerd wordt. Dit heeft als gevolg dat we de gegevensstructuur die gebruikt wordt om de snijpunten in op te slagen, een betere initiële capaciteit kunnen meegegeven. Om het verwachte aantal snijpunten te berekenen, berekenen we de oppervlakte waarin een gemiddelde cirkel een andere cirkel snijdt. Een gemiddelde cirkel is bij onze gegenereerde cirkels een cirkel met een straal gelijk aan de maximale straal gedeeld door twee. Vervolgens nemen we de verhouding tussen deze oppervlakte en de totale oppervlakte van het vlak. Dit geeft de kans dat twee cirkels snijden. Om deze te extrapoleren naar de kans dat n cirkels snijden, vermenigvuldigen we met $\frac{n(n-1)}{2}$. Hierna vermenigvuldigen we met twee omdat twee snijdende cirkels bijna altijd twee keer snijden.

$$gemiddeldeStraal = maxStraal/2$$
$$snijdingsOppervlakteGemiddeldeCirkel = \pi \times (2 \times gemiddeldeStraal)^2$$
$$verwachteAantalSnijpunten = \frac{snijdingsOppervlakteGemiddeldeCirkel}{totaleOppervlakte} * \frac{aantalCirkels \times (aantalCirkels-1)}{2} \times 2$$



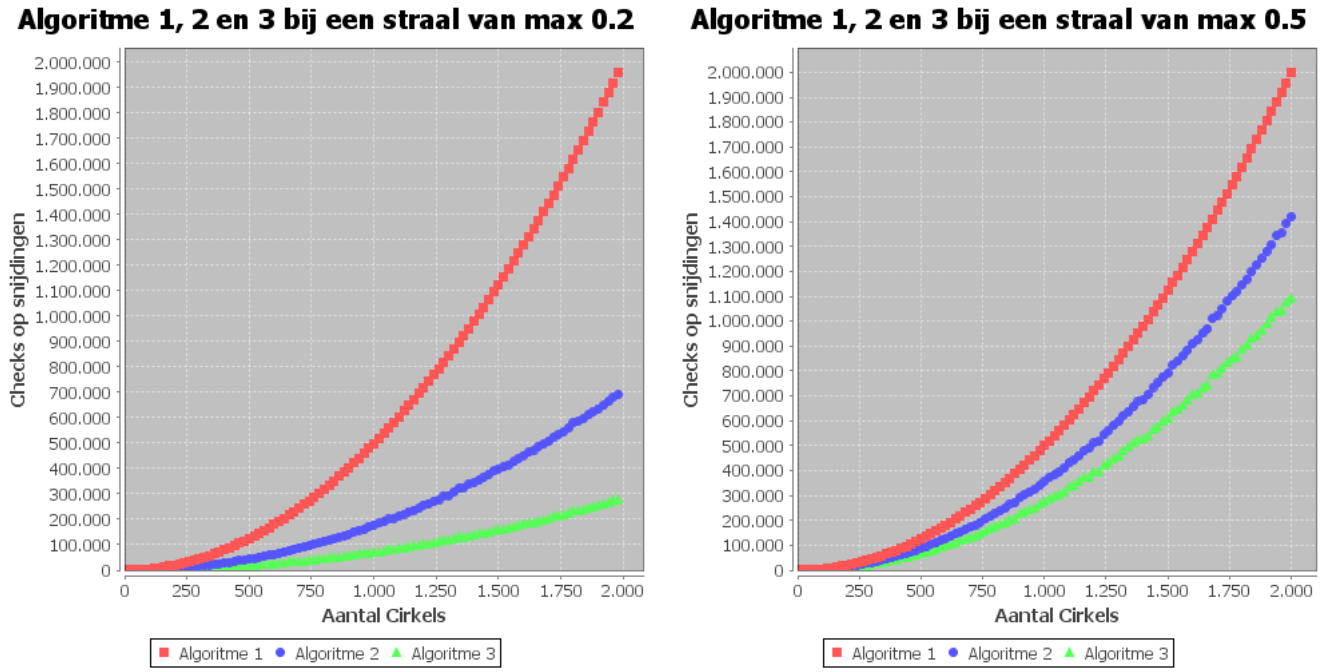
Figuur 4: Schatting van aantal snijpunten samen met het werkelijk aantal snijpunten

2.2 Relatief grote stralen



Figuur 5: Uitvoeringstijd van algoritme 1, 2 en 3 bij grote stralen

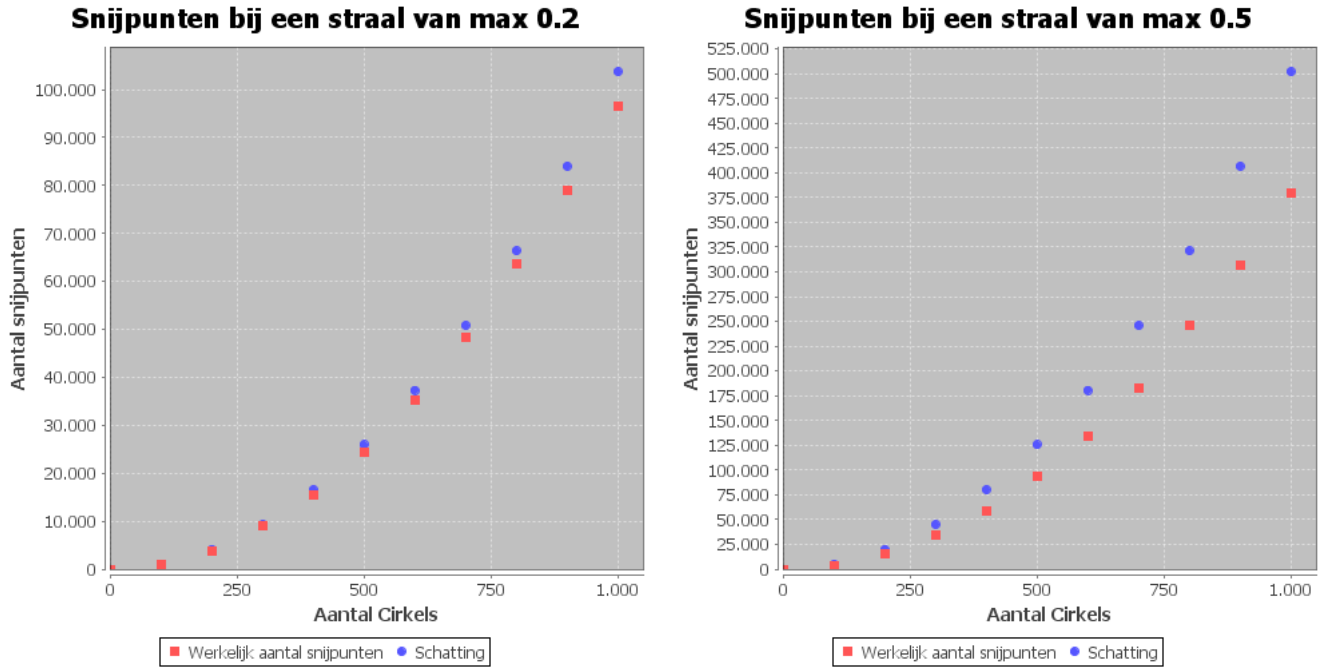
Grote stralen hebben geen invloed op het aantal vergelijkingen dat *Algoritme 1* moet doen om te kijken of cirkels snijden. Zoals blijkt uit Figuur 5 is er echter wel een verschil in uitvoeringstijd. Dit komt doordat het checken of twee cirkels snijden sneller is dan het daadwerkelijk berekenen van de snijpunten. Bij grote stralen zijn er veel meer snijpunten (zoals te zien is wanneer we Figuur 4 en 7 vergelijken), waardoor er meer snijpunten berekend zullen moeten worden.



Figuur 6: Aantal keer dat "berekenSnijpunten" wordt opgeroepen door *Algoritme 1*, *2* en *3* bij grote stralen

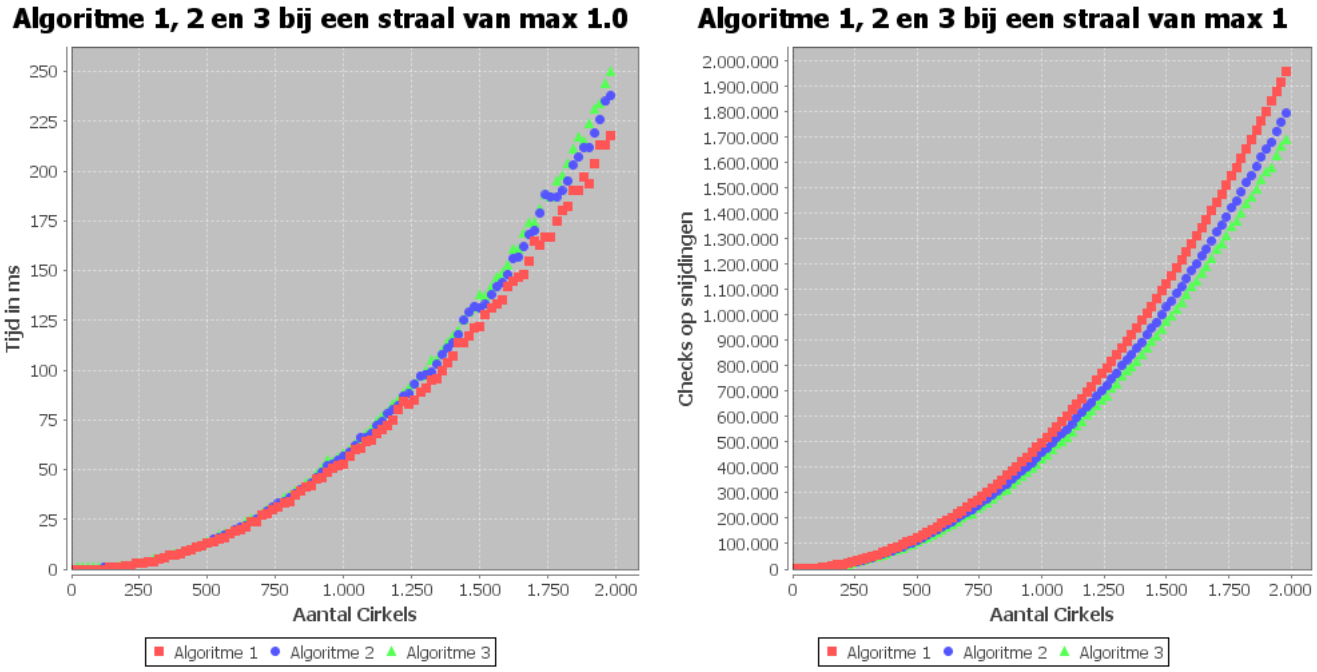
Algoritme 2 is duidelijk trager bij grote cirkels dan bij kleine cirkels. Dit komt doordat cirkels langer actief blijven als ze een grotere straal hebben en er dus meer cirkels tegelijk actief zullen zijn. In het slechtste geval zullen alle cirkels tegelijk actief zijn. Hierdoor zal *Algoritme 2* dezelfde rekencomplexiteit krijgen als *Algoritme 1*, aangezien elke cirkel met elke andere cirkel vergeleken zal worden.

Aangezien *Algoritme 3* een rekencomplexiteit heeft van $O((n+S)log_2(n))$, zal dit algoritme vertragen wanneer het aantal snijpunten toeneemt (zoals te zien in Figuur 5). In het slechtste geval, wanneer elke cirkel elke andere cirkel twee keer snijdt, zijn er $n(n-1)$ snijpunten. Hierdoor wordt de rekencomplexiteit van dit algoritme opgeblazen tot $O((n^2)log_2(n))$, wat duidelijk slechter is dan de rekencomplexiteit van *Algoritme 1*. Dit zal echter nooit voorkomen in een reële situatie.



Figuur 7: Schatting van aantal snijpunten samen met het werkelijk aantal snijpunten

2.3 Straal van maximum één



Figuur 8: Uitvoeringstijd en aantal keer dat "berekenSnijpunten" wordt opgeroepen bij een straal van één

Bij een straal van maximaal één zien we dat hoewel het aantal vergelijken om snijpunten te berekenen bij *Algoritme 3* nog steeds het laagst is, de uitvoeringstijd het hoogst is. Dit komt door de complexe gegevensstructuren die voor *Algoritme 3* nodig zijn.

3. Het berekenen van snijpunten

De methode die we gebruikt hebben om snijpunten te vinden, is gebaseerd op de methode die beschreven wordt in de volgende bron.

Bron

N.N., How to calculate the intersection of two circles. Internet, 3 juli 2011, (http://fypandroid.wordpress.com/2011/to-calculate-the-intersection-of-two-circles-java/).