KU Leuven

Practicum

Snijdende cirkels

Toepassingen van de meetkunde in de informatica [G0Q37C]

Auteurs:
Xavier Goás Aguililla
Tom Sydney Kerckhove

Professor:
prof. dr. ir. Dirk Roose

27 mei 2014

Inhoudsopgave

Inhoudsopgave **1**

Lijst van figuren **2**

Lijst van tabellen **2**

Lijst van algoritmen **2**

1 Inleiding **4**

2 Algoritmen **6**
2.1 De snijpunten van twee cirkels berekenen . . . 6
2.1.1 Algoritme . . . 6
2.1.2 Theoretische achtergrond . . . 7
2.1.3 Complexiteit . . . 9
2.2 Naïef . . . 9
2.2.1 Algoritme . . . 9
2.2.2 Correctheidsbewijs . . . 9
2.2.3 Complexiteit . . . 10
2.3 Kwadratisch . . . 11
2.3.1 Algoritme . . . 11
2.3.2 Correctheidsbewijs . . . 11
2.3.3 Complexiteit . . . 13
2.4 Linearitmisch . . . 15
2.4.1 Algoritme . . . 15
2.4.2 Correctheidsbewijs . . . 15
2.4.3 Complexiteit . . . 16

3 Implementatie **18**
3.1 Haskell . . . 18
3.1.1 Evaluatie en timing . . . 18
3.1.2 Lijsten . . . 18
3.2 Naief . . . 19
3.3 Kwadratisch . . . 19
3.4 Linearitmisch . . . 20

4 Experimenten **21**
4.1 Correctheidstoetsen . . . 21
4.2 Rauwe data . . . 21
4.2.1 Weinig snijpunten . . . 21
4.2.2 Veel snijpunten . . . 21
4.2.3 3D-plot . . . 21
4.3 Doubling ratio . . . 21
4.3.1 Naïef en kwadratisch . . . 22
4.3.2 Linearitmisch . . . 22

5 Resultaten **23**
5.1 CorrectheidToetsen . . . 23
5.2 Rauwe data . . . 23
5.2.1 Weinig Snijpunten . . . 23
5.2.2 Gemiddeld geval . . . 24
5.2.3 Veel snijpunten . . . 25
5.2.4 3D plot . . . 26
5.3 Doubling ratio . . . 27
5.3.1 Naief . . . 27
5.3.2 Kwadratisch . . . 28
5.3.3 Lineairtmisch . . . 28

6 Besluit **29**

7 Reflectie **29**
7.1 Onverwachte verhouding . . . 29
7.2 Haskell sequence . . . 29
7.3 Compilatie optimalisaties . . . 29

A Voorbeeldgeval **30**

Lijst van figuren

1 Een voorbeeld opgave . . . 5
2 Snijpunten van twee cirkels . . . 7
3 Nagekeken cirkels bij algoritme 1 . . . 10
4 Nagekeken cirkels bij algoritme 2 . . . 14
5 Nagekeken cirkels bij algoritme 3 . . . 17
6 Operaties op lijsten . . . 18
7 Functionele code voor het naïeve algoritme . . . 19
8 Functionele code voor het kwadratische algoritme . . . 19
9 Functionele code voor het linearitmische algoritme . . . 20
10 Uitvoeringstijden bij gevallen met weinig snijpunten. . . . 23
11 Uitvoeringstijden bij een gemiddeld geval snijpunten. . . . 24
12 Uitvoeringstijden bij gevallen met relatief veel snijpunten. . . . 25
13 Een 3D grafiek van de uitvoeringstijden in functie van de scalering en het aantal cirkels 26
14 Aantal snijpunten . . . 27

Lijst van tabellen

1 Doubling ratio 1 . . . 27
2 Doubling ratio 2 . . . 28
3 Doubling ratio 3 . . . 28

Lijst van algoritmen

1 Snijpunten van twee cirkels berekenen . . . 6
2 Nagaan of twee cirkels snijden . . . 6
3 Naïeve aanpak . . . 9

4 Kwadratische aanpak 11
5 Events 11
6 Lineairitmische aanpak 15
7 Breedte-interval van een cirkel 15

1 Inleiding

We bespreken drie verschillende algoritmen voor het vinden van de snijpunten van een verzameling van n cirkels C. De opgave van het practicum herhalen is vrij zinloos, maar alvorens we overgaan tot de werkelijke inhoud van dit verslag, willen we een aantal zaken aankaarten met betrekking tot onze algoritmische notatie en de implementatie van de algoritmen.

Notatie. Wat betreft de notatie van de algoritmen volgen we in het algemeen de wiskundige notatie voor verzamelingen en operaties daarop. De algoritmes volgen globaal een imperatief stramien: de stappen worden sequentieel uitgevoerd zoals bepaald door de besturingsstructuren in het algoritme. Logische expressies en functies worden zoals hoort in standaard logische notatie weergegeven. Dit staat enigszins in contrast met onze implementatie.

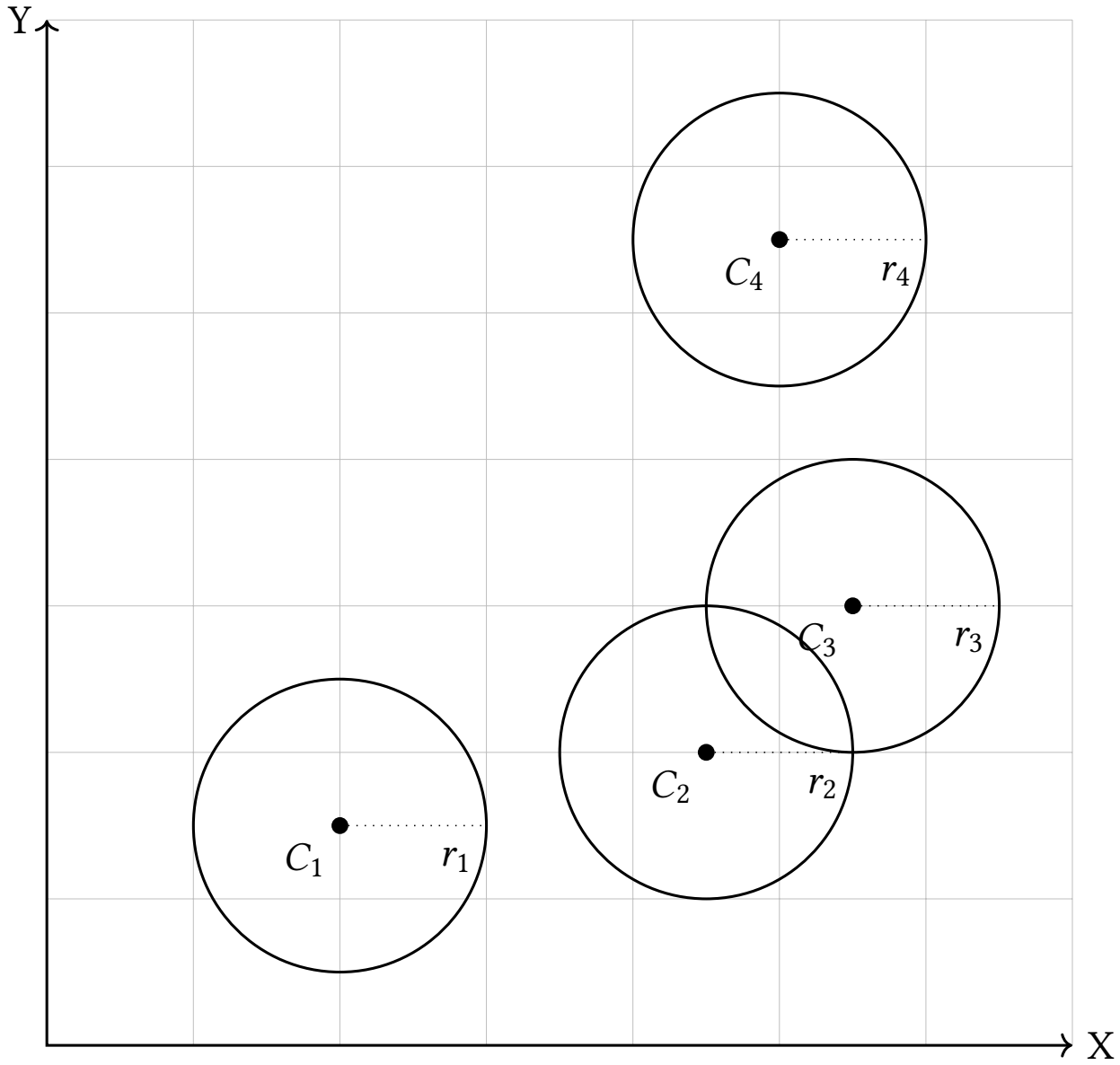
Implementatie. Wij hebben ervoor gekozen de algoritmen te implementeren in Haskell, een functionele programmeertaal. Dit zorgt ervoor dat de algoritmen op een andere manier moeten of kunnen worden uitgedrukt dan in de klassieke notatie. Dit kan code veel leesbaarder en beknopter maken. Anderzijds is het soms moeilijker om een traditionele complexiteitsanalyse te maken: zo kent Haskell geen iteratieve lussen, maar wordt overvloedig gebruik gemaakt van recursie. We belichten sommige aspecten van onze implementatie concreter in sectie 3.

Invoer en uitvoer. De invoer en uitvoer gebeurt via de standaard *streams* voor invoer en uitvoer op Linux. Het programma wordt dus als volgt aangeroepen:

```
Executable < input.txt > output.txt
```

Naamgeving. We zullen de beschreven algoritmes aanduiden overeenkomstig hun complexiteit. Zo noemen we de algoritmes respectievelijk ‘naïef’, ‘kwadratisch’ en ‘linearitmisch’. In de tekst zelf zullen we vaak ook refereren naar hun volgnummer in de lijst van algoritmen in de inhoudstafel. ‘Algoritme 3’ in de tekst komt dus niét overeen met het ‘algoritme 3’ uit de opgave!

Wij wensen u veel lees- en verbeterplezier.



Figuur 1: Een voorbeeld opgave

2 Algoritmen

2.1 De snijpunten van twee cirkels berekenen

2.1.1 Algoritme

Algoritme 1 : Snijpunten van twee cirkels berekenen

Input : twee cirkels, c en c' met resp. middelpunten $p_1 = (x_1, y_1), p_2 = (x_2, y_2)$ en stralen r_1, r_2
Output : geen, één of twee snijpunten van de twee cirkels (de doorsnede van de cirkels)
Procedure *intersections(c,c')*:

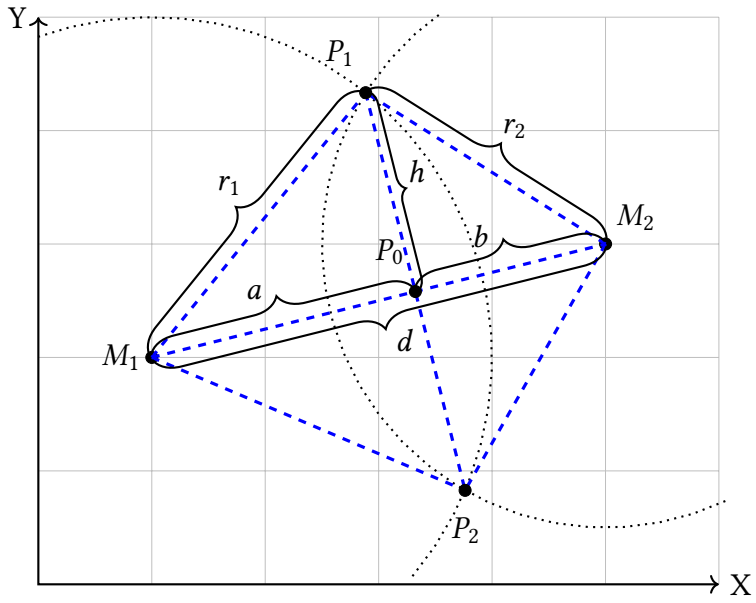
$result \leftarrow \emptyset$
if $intersect(c_1, c_2) \wedge c_1 \neq c_2$ **then**
$d \leftarrow \|p_1 - p_2\|$
$a \leftarrow \frac{r_1^2 - r_2^2 + d^2}{2d}$
$x_0 \leftarrow x_1 + \frac{a}{d}(x_2 - x_1)$
$y_0 \leftarrow y_1 + \frac{a}{d}(y_2 - y_1)$
$h \leftarrow \sqrt{r_1^2 - a^2}$
$x_1' \leftarrow x_0 + \frac{h}{d}(y_2 - y_1)$
$y_1' \leftarrow y_0 - \frac{h}{d}(x_2 - x_1)$
$result \leftarrow result \cup \{(x_1', y_1')\}$
if $a \neq r_1$ **then**
$x_2' \leftarrow x_0 - \frac{h}{d}(y_2 - y_1)$
$y_2' \leftarrow y_0 + \frac{h}{d}(x_2 - x_1)$
$result \leftarrow result \cup \{(x_2', y_2')\}$
end
end
return *result*
end

Algoritme 2 : Nagaan of twee cirkels snijden

Input : twee cirkels, c en c' met resp middelpunten p_1, p_2 en stralen r_1, r_2
Output : waar als en slechts als de twee cirkels snijden
Procedure *intersect(c,c')*:

d $\leftarrow \|p_2 - p_1\|$
if $(d \leq r_1 + r_2) \wedge (d \geq abs(r_1 - r_2))$ **then**
return *true*
else
return *false*
end
end

2.1.2 Theoretische achtergrond



Figuur 2: Snijpunten van twee cirkels

Gegeven twee cirkels C_1 en C_2 zoeken we $C_1 \cap C_2$. Noem r_i de straal van C_i. We gaan er ook van uit dat r steeds positief is. Het is niet nodig om deze voorwaarde te stellen op theoretisch niveau, maar alle theorie is nog steeds geldig wanneer we enkel over positieve stralen spreken.

Mogelijke doorsneden Twee cirkels kunnen op vier specifieke manieren gelegen zijn, met betrekking tot hun doorsnede. Het blijkt zo te zijn dat we de situatie kunnen herkennen aan de waarde van $d = \|M_2 - M_1\|$ wanneer de cirkels niet samenvallen.

- De cirkels vallen samen.

  $$C_1 \cap C_2 = C_1 = C_2$$

  Dit valt voor wanneer de cirkels concentrisch zijn en een even grote straal hebben.

  $$M_1 = M_2 \text{ en } r_1 = r_2$$

  Hier geldt $d = 0$, maar deze situatie zullen we in het programma negeren.

- De cirkels snijden in precies twee punten.

  $$C_1 \cap C_1 = \{P_1, P_2\} \text{ met } P_1, P_2 \in \mathbb{R}^2$$

  In deze situatie geldt zowel $d < r_1 + r_2$ als $d > |r_1 - r_2|$.

- De cirkels raken aan elkaar.

  $$C_1 \cap C_2 = \{P\}$$

  In deze situatie geldt $d = r_1 + r_2$.

- De cirkels raken niet.

  $$C_1 \cap C_2 = \{P_1, P_2\} \text{ met } P_1, P_2 \in \mathbb{C}^2$$

  Hier geldt ofwel $d < |r_1 - r_2|$ ofwel $d > r_1 + r_2$

Benaming De benaming die hier beschreven staat is ook te zien in figuur 2.

We geven de twee stralen standaard de benaming r_1 en r_2 respectievelijk. Noem de middelpunten van de cirkels $M_1(x_1, y_1)$ en $M_2(x_2, y_2)$.

Noem de doorsnede van C_1 en C_2 I. Wanneer de twee cirkels gelijk zijn geldt $C_1 = C_2 = I$. In de rest van dit deel bespreken we enkel het geval waarbij de snijpunt(en) reëel zijn. Noem de snijpunten P_1 en P_2 in de berekening. Wanneer er maar één snijpunt is zal de berekening nog steeds gelden, maar zal P_1 gelijk zijn aan P_2.

Noem de rechte door M_1 en M_2 l_1 en de rechte door P_1 en P_2 l_2. Het snijpunt van l_1 en l_2 noemen we P_0. Noem a de afstand van P_1 tot P_0 en b de afstand van P_2 tot P_0. Noem d de afstand tussen M_1 en M_2.

$$d = a + b$$

Noem h de afstand van P_1 (of P_2) tot P_0.

Snijpunten berekenen We zien dat de driehoeken $M_1P_1P_0$, $M_2P_1P_0$, $M_1P_2P_0$ en $M_2P_2P_0$ rechthoekige driehoeken zijn. We gebruiken nu de regel van Pythagoras om de volgende gelijkheden te bekomen.

$$\begin{cases} r_1^2 = a^2 + h^2 & (1) \\ r_2^2 = b^2 + h^2 & (2) \end{cases}$$

Uit de eerste vergelijking kunnen we een uitdrukking voor h halen.

$$h = \sqrt{r_1^2 - a^2}$$

Trekken we vergelijking (2) van vergelijking (1) af, dan krijgen we volgende vergelijking.

$$\begin{aligned} r_1^2 - r_1^2 &= a^2 - b^2 \\ &= (a - b)(a + b) \\ &= d(a - b) \\ &= d(a - d + a) \\ &= d(2a - d) \end{aligned}$$

Vormen we deze vergelijking om om een uitdrukking voor a te bekomen, dan krijgen we volgende uitdrukking.

$$a = \frac{r_1^2 - r_2^2 + d^2}{2d}$$

Vervolgens kunnen we de coördinaten van P_0 berekenen. P_0 ligt namelijk op de rechte M_1M_2 op afstand $\frac{a}{d}$ van M_1.

$$P_0 = M_1 + \frac{a}{d}(M_2 - M_1) = M_2 + \frac{b}{d}(M_1 - M_2)$$

$$\begin{bmatrix} P_{0x} \\ P_{0y} \end{bmatrix} = \begin{bmatrix} x_1 + \frac{a}{d}(x_2 - x_1) \\ y_1 + \frac{a}{d}(y_2 - y_1) \end{bmatrix} = \begin{bmatrix} x_2 + \frac{a}{d}(x_1 - x_2) \\ y_2 + \frac{a}{d}(y_1 - y_2) \end{bmatrix}$$

Tenslotte kunnen we de coördinaten van de snijpunten berekenen uit de coördinaten van M_1, M_2 en P_0.

$$\begin{bmatrix} P_x \\ P_y \end{bmatrix} = \begin{bmatrix} x_0 \pm \frac{h}{d}(y_2 - y_1) \\ y_0 \mp \frac{h}{d}(x_2 - x_1) \end{bmatrix}$$

2.1.3 Complexiteit

De snijpunten berekenen van twee *verschillende* cirkels heeft een constante uitvoeringstijd. Het is opmerkelijk dat er in, nog steeds constante, kortere tijd kan nagekeken worden of twee cirkels snijden, vóór de snijpunten berekend worden, zodat er minder berekeningen gedaan kunnen worden. Wanneer twee cirkels in het algoritme terecht komen die niet snijden zal er het algoritme proberen complexe getallen te berekenen. Als het systeem daar niet op voorzien is resulteert dat in 'NaN' waarden.

2.2 Naïef

2.2.1 Algoritme

Algoritme 3 : Naïeve aanpak

Input : een lijst van n cirkels C, gegeven door hun middelpunt en straal
Output : een verzameling van S snijpunten R van de cirkels in C
Procedure *intersections1(C)*:
    R $\leftarrow \emptyset$;
    for $i \leftarrow 1$ **to** n **do**
        for $j \leftarrow i$ **to** n **do**
            $R \leftarrow R \cup intersections(C(i), C(j))$
        end
    end
    return R
end

2.2.2 Correctheidsbewijs

Stelling 2.2.1. *Algoritme 3 is correct en eindig.*

Bewijs. Het algoritme eindigt: immers, de instructie in de binnenste for-lus wordt exact

$$\sum_{i=1}^{n} \sum_{j<i}^{n} 1 = \frac{n(n-1)}{2}$$

keer uitgevoerd.

De correctheid van het algoritme bewijzen we met een lusinvariante:

Invariante 1. *Na de i^{de} iteratie van de buitenste lus zijn alle snijpunten van de eerste i cirkels met de andere cirkels toegevoegd aan de verzameling snijpunten.*

Bewijs. We bewijzen dit per inductie.

Basisstap. Het is duidelijk dat voor $i = 1$ de invariante geldt. Immers, na iteratie 1 hebben we cirkel 1 in de lijst nagekeken op snijpunten met alle $n - 1$ andere cirkels, en zijn dus alle snijpunten van de eerste i cirkels met alle andere cirkels toegevoegd aan de verzameling snijpunten.

Inductiestap. Stel dat voor de eerste $i - 1$ iteraties de invariante geldt. We tonen aan dat na de i^{de} iteratie de invariante nog geldt. In de i^{de} iteratie wordt cirkel i nagekeken op snijpunten met alle cirkels die later in de lijst komen. De snijpunten van i met alle cirkels die later in de lijst komen zullen dus zeker in de verzameling snijpunten zitten na afloop van deze iteratie.

Maar ook alle snijpunten met de cirkels die vroeger in de lijst zitten hebben we al gevonden en bijgehouden; immers, in één van de vorige iteraties is het snijpunt van elk van die cirkels met cirkel i al berekend en toegevoegd aan de verzameling snijpunten. Bij krachte van het inductieprincipe

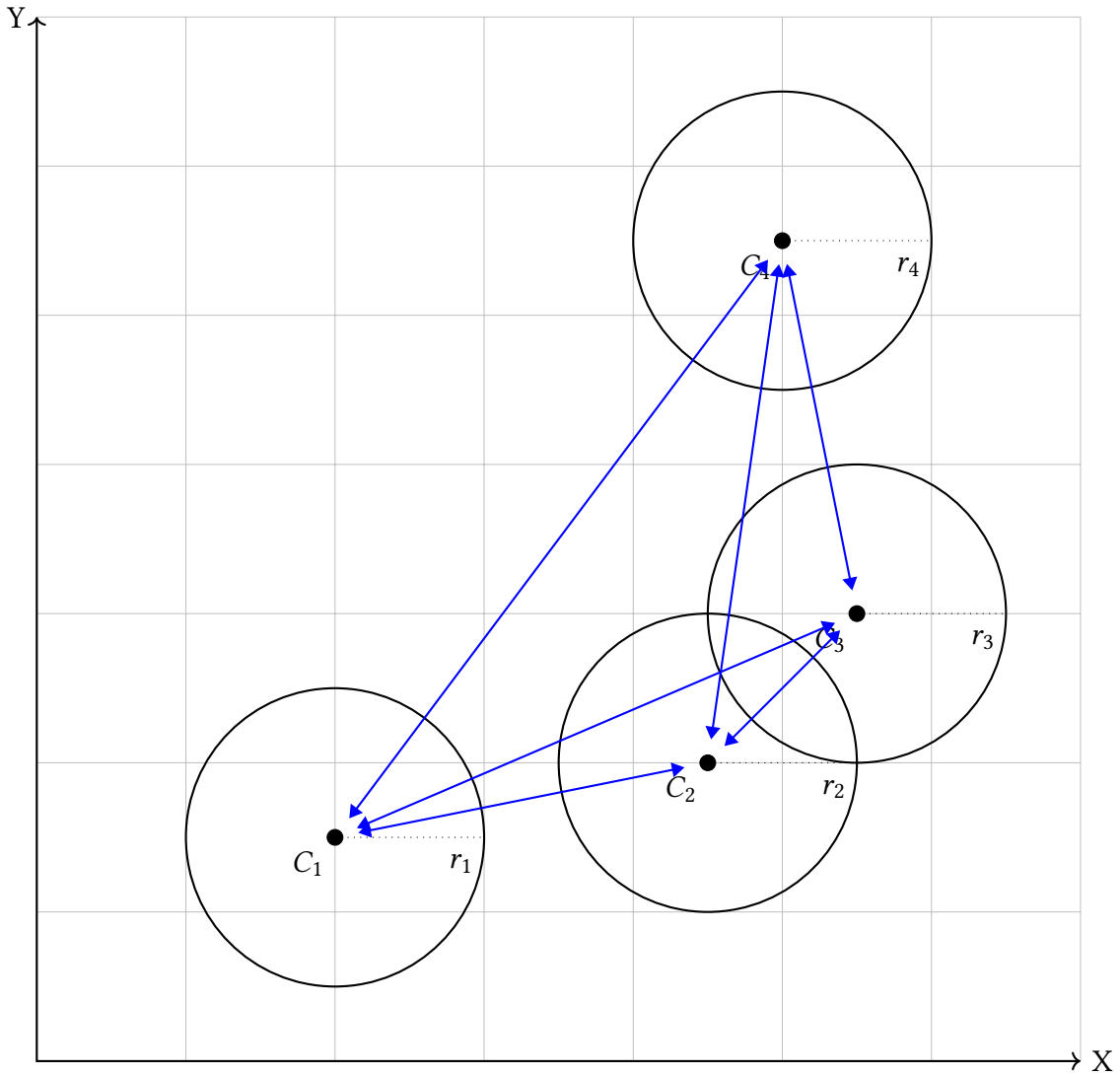
bevat de verzameling snijpunten na iteratie i dus alle snijpunten van de eerste i cirkels met alle andere cirkels. □

Na iteratie n zijn dus alle snijpunten gevonden; het algoritme is correct. □

2.2.3 Complexiteit

De naïeve aanpak heeft complexiteit $O(N^2)$. Dit volgt rechtstreeks uit ons bewijs van eindigheid, immers:

$$\frac{N(N-1)}{2} = O(N^2)$$



Figuur 3: Nagekeken cirkels bij algoritme 1

2.3 Kwadratisch

2.3.1 Algoritme

Algoritme 4 : Kwadratische aanpak

Input : een lijst van n cirkels C, gegeven door hun middelpunt en straal
Output : een verzameling van S snijpunten R van de cirkels in C
Procedure *intersections2(C)*:

```
E ← ∅;
for c ∈ C do
    E ← E ∪ events(c);
end
E ← sort(E);
T, R ← ∅;
for e ∈ E do
    if e == insert c then
        for c' ∈ T do
            R ← R ∪ intersections(c, c')
        end
        T ← T ∪ {c}
    else if e == delete c then
        T ← T \ {c}
    end
end
return R
```

end

Algoritme 5 : Events

Input : een cirkel c met middelpunt (x, y) en straal r
Output : twee *events* e_1 en e_2, corresponderend aan het toevoegen en verwijderen van c aan de doorlooplijn, waarbij elk event ook de y-coördinaat waarop het gebeurt met zich meedraagt
Procedure *events(c)*:

```
y1 ← y − r;
y2 ← y + r;
R ← {(y1, insert c), (y2, delete c)};
return R
```

end

2.3.2 Correctheidsbewijs

Stelling 2.3.1. *Algoritme 4 is correct en eindig.*

Bewijs. We bewijzen eerst dat het algoritme eindigt, dan dat het correct is.

De eerste for-lus is eindig en levert een eindige verzameling van events E op; dit volgt rechtstreeks uit de eindigheid van de verzameling cirkels C en uit het feit dat algoritme 5 altijd een eindige verzameling teruggeeft. De tweede for-lus is genest; we bewijzen dat ook deze eindigt.

De binnenste for-lus past de binnenste lus van het naïeve algoritme dat we in de vorige sectie besproken hebben toe op T, en is dus gegarandeerd eindig als de verzameling T bij elke iteratie eindig is. Om dat aan te tonen, bewijzen we eerst dat de buitenste for-lus eindig is. Die behandelt elk element in E éénmaal; het aantal iteraties ligt dus vast op $|E|$ en de lus eindigt.

Nu kunnen we aantonen dat T altijd eindig is. We doen dit wederom met een invariante.

Invariante 2. *T bevat na elke iteratie van de buitenste lus maximaal n elementen.*

Bewijs. Op het einde van elke iteratie van de buitenste lus wordt exact één cirkel ofwel toegevoegd aan ofwel verwijderd uit T. Elke cirkel wordt exact één keer toegevoegd aan of verwijderd uit T (zie algoritme 5). Dus E bevat exact $2n$ elementen. Stel nu dat T na iteratie i $n+1$ elementen zou bevatten. Er zijn maar n cirkels, dus volgens het duiventilprincipe zou er ergens een cirkel tweemaal moeten toegevoegd zijn, wat in strijd is met het feit dat elke cirkel exact één keer toegevoegd aan of verwijderd uit T. Dit kan niet en dus moet T maximaal n elementen bevatten. □

Hiermee is de eindigheid van het algoritme bewezen. Nu gaan we over tot een correctheidsbewijs. Daarvoor volstaat het aan te tonen dat het voldoende is om een cirkel die we toevoegen tijdens iteratie i te checken met de cirkels in T. We doen dit in meerdere stappen.

We merken eerst op dat we de lijst E voor gebruik sorteren op y-coördinaat. Voor het voorbeeldgeval levert dit de volgende geordende verzameling E op:

$$E = \{insert\ c_1, delete\ c_1, insert\ c_2, insert\ c_4, insert\ c_3, delete\ c_2, delete\ c_4, delete\ c_3\}\,.$$

Als we nader kijken, zien we dat E eigenlijk een reeks van n intervallen op de y-as definieert die elk bepaald worden door de uiterste y-coördinaten van elk van de n cirkels in C. Wanneer we E sorteren, bekomen we dan een geordende verzameling die aangeeft wanneer deze intervallen overlappen. Dit gaan we nu formeel aantonen.

We construeren nu een verzameling T' die bestaat uit alle mogelijke combinaties van cirkels wiens intervallen overlappen, geordend volgens toevoegpunt. Voor ons voorbeeld zou dit bijvoorbeeld de volgende verzameling zijn:

$$T' = \{\{c_2, c_4\}, \{c_2, c_4, c_3\}, \{c_4, c_3\}\}$$

We willen het volgende aantonen:

Lemma 1. *Elk element van T' is een deelverzameling van een configuratie van T.*

Bewijs. We tonen dit per constructie aan. Elke configuratie t van T is gegenereerd door een gesorteerde sequentie van toevoegpunten en verwijderpunten zoals we die in E vinden. Elk tweetal toevoegpunt-verwijderpunt van een cirkel vormt een interval. De i^{de} configuratie van T wordt gegenereerd door de eerste $i-1$ elementen van E.

Neem de i^{de} configuratie van T. We kunnen gemakkelijk aantonen dat die een element uit T' als deelverzameling heeft. Immers, alle cirkels die in een willekeurige configuratie van T zitten hebben overlappende intervallen door de manier waarop T gegenereerd is.

We bewijzen nu inductief per constructie van t dat E een T genereert zodanig dat $t \subseteq T$ vóór iteratie i.

Basisstap. De lege verzameling is triviaal een deelverzameling van elke verzameling. Als basisstap nemen we dus $|t| = 1$. Het is makkelijk in te zien dat elk singleton ook een deelverzameling is van minstens één configuratie van T, gezien elke cirkel exact één keer aan T wordt toegevoegd.

Inductiestap. Stel dat er een $t \in T'$ bestaat met grootte $i-1$ die deelverzameling is van een configuratie van T. Dan tonen we nu aan dat juist één van de volgende gevallen geldt:

Geval 1. Er bestaat een t' van grootte i met $t \subseteq t'$. Dan is die t' ook een deelverzameling van een configuratie van T. Immers, in t' zit juist één cirkel meer dan in t, en deze cirkel moet overlappen met alle cirkels in t. Het toevoegpunt van deze cirkel bevindt zich dus vóór het verwijderpunt van de laatste cirkel in t. Gezien E gesorteerd wordt naar stijgende y-coördinaat, zal deze cirkel toegevoegd worden aan de configuratie van T waarvan t maximaal deelverzameling is en zal t' op zijn beurt deelverzameling zijn van de nieuwe configuratie van T.

Geval 2. Er bestaat geen t' van grootte i met $t \subseteq t'$. We hoeven dan niet aan te tonen dat t' een deelverzameling is van een configuratie van T, gezien t' geen element kan zijn van T.

□

Gevolg. *Elke configuratie van T bevat minstens één element van T' als deelverzameling.*

Nu kunnen we bewijzen dat het algoritme correct is. Daarvoor gebruiken we het het volgende lemma:

Lemma 2. *Elke cirkel die vóór het begin van iteratie i uit T verwijderd is kan onmogelijk snijden met een cirkel die toegevoegd wordt tijdens iteratie i.*

Bewijs. We bewijzen uit het ongerijmde. Stel dat er een cirkel $c' \in C$ bestaat die uit T verwijderd is maar die wel met c snijdt. Na het nakijken op snijpunten wordt c toegevoegd aan T. Gezien c en c' snijden, overlapt het interval op de y-as dat correspondeert aan c' met het interval dat correspondeert aan c.

Nu heeft c' al eens in T gezeten: uit het gevolg van 1 volgt dan dat onze stelling een contradictie is; immers, dan is er een element $t \in T'$ waarin zowel c als c' zitten, maar dit behoort dan tot geen enkele configuratie van T, gezien een cirkel die verwijderd is niet meer toegevoegd kan worden aan T. Daarmee is het lemma bewezen. □

Gevolg. *Als een cirkel aan T wordt toegevoegd aan het einde van iteratie i hoeven tijdens die iteratie enkel snijpunten van die cirkel met de cirkels uit T toegevoegd te worden aan de verzameling snijpunten.*

Bewijs. Als er cirkels zijn die voor iteratie i niet in T zitten, kan dat twee redenen hebben:

Geval 3. De cirkels zijn al toegevoegd en terug verwijderd uit T; dan geldt lemma 2 en hoeven wij niet na te kijken op snijpunten met c.

Geval 4. De cirkels zijn nog niet toegevoegd aan T: dan weten we dat die in een latere iteratie zullen toegevoegd worden. Op dat moment kunnen we terug lemma 2 toepassen.

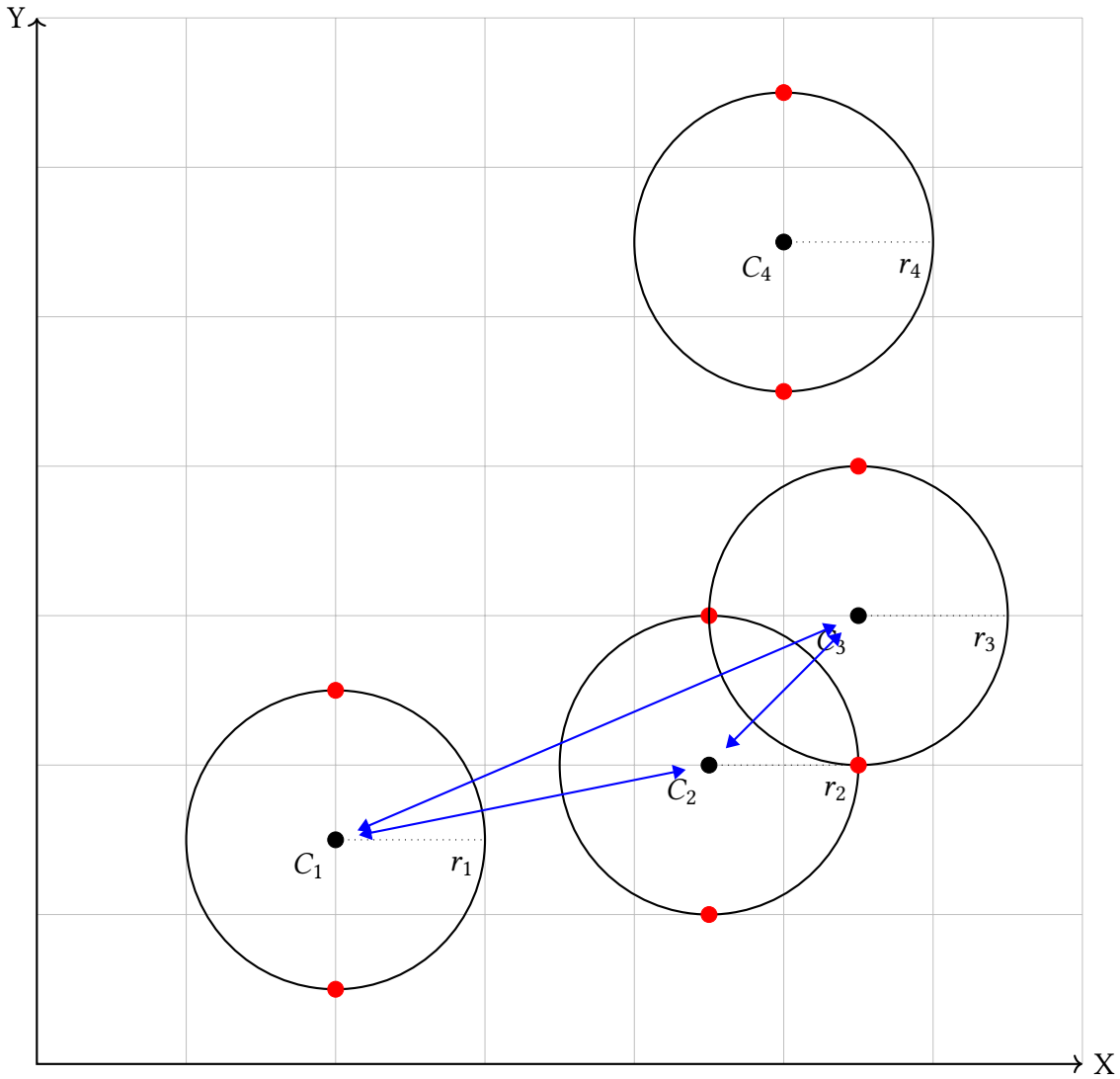
□

Het bewijs dat de snijpunten van alle cirkels op deze manier in de uiteindelijke verzameling snijpunten komen te zitten loopt analoog aan het bewijs van correctheid voor algoritme 3. Daarmee is ons correctheidsbewijs afgerond. □

2.3.3 Complexiteit

De buitenste lus wordt exact $2n$ keer uitgevoerd. In de n iteraties waarin een cirkel verwijderd wordt, wordt exact één instructie uitgevoerd. In de n iteraties waarin een cirkel wordt toegevoegd, worden maximaal $n - 1$ vergelijkingen met andere cirkels uitgevoerd: T kan immers op geen enkel moment meer dan n elementen hebben. Onze uiteindelijke complexiteit komt in het slechtste geval dus uit op:

$$n(n - 1 + 1) = O(n^2)$$



Figuur 4: Nagekeken cirkels bij algoritme 2

2.4 Linearitmisch

2.4.1 Algoritme

Algoritme 6 : Linearitmische aanpak

Input : een lijst van n cirkels C, gegeven door hun middelpunt en straal
Output : een verzameling van S snijpunten R van de cirkels in C
Procedure *intersections3(C)*:

```
E ← ∅;
for c ∈ C do
    E ← E ∪ events(c);
end
E ← sort(E);
T, R ← ∅;
//T is een intervalboom for e ∈ E do
    if e == insert c then
        for c' ∈ overlapping(c, T) do
            R ← R ∪ intersections(c, c')
        end
        T ← T ∪ {c}
    else if e == delete c then
        T ← T \ {c}
    end
end
return R
```

end

Algoritme 7 : Breedte-interval van een cirkel

Input : een cirkel c met middelpunt (x, y) en straal r
Output : een interval I corresponderend aan het interval tussen de uiterste x-coördinaten van c
Procedure *interval(c)*:

```
x1 ← x − r;
x2 ← x + r;
I ← [x1, x2];
return I
```

end

2.4.2 Correctheidsbewijs

Stelling 2.4.1. *Algoritme 6 is correct en eindig.*

Bewijs. Dit algoritme verschilt van algoritme 4 op slechts n punt: in de binnenste for-lus wordt nog extra gepruned door te checken op het overlappen van de intervallen van de cirkels op de x-as. We voegen geen extra lussen toe, dus de eindigheid van het algoritme is gegarandeerd bij krachte van stelling 2.3.1.

Door de check in de binnenste for-lus wordt nog slechts een deelverzameling van T vergeleken met de cirkel c toegevoegd in iteratie i van de tweede for-lus. Noem deze deelverzameling B. Het volstaat aan te tonen dat er in $T \setminus B$ geen enkele cirkel zit die zou kunnen snijden met c. Dat is simpel: het criterium dat we opleggen is dat c over de x-as moet overlappen met elke c'; formeel zien we in dat

$$B = \left\{c' | c' \in T, interval(c) \cap interval(c') \neq \varnothing\right\}$$

en dus is het ook zo dat

$$T \setminus B = \left\{c' | c' \in T, interval(c) \cap interval(c') = \varnothing\right\}$$

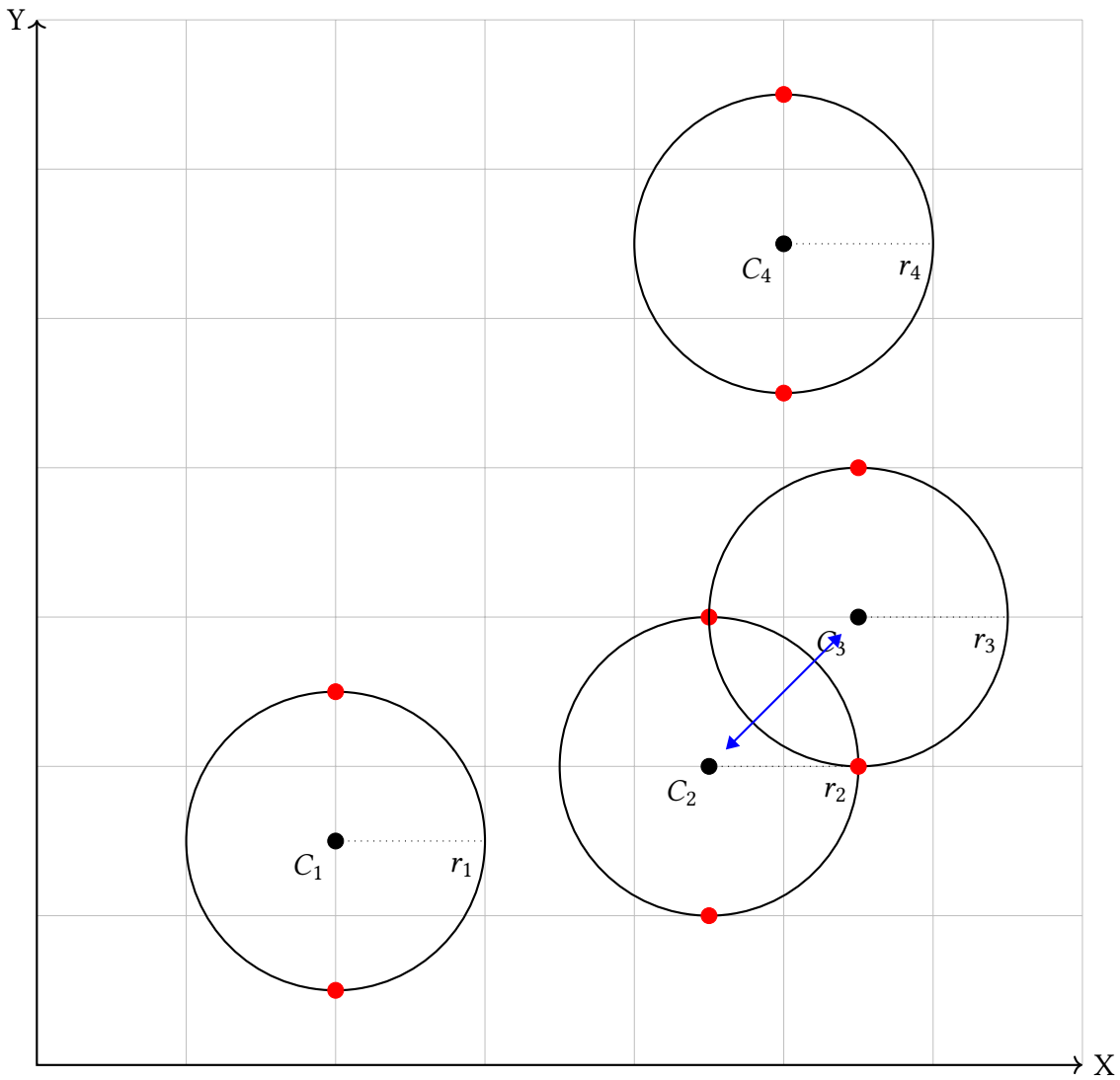
Het is niet moeilijk in te zien dat geen enkele cirkel c' in $T \setminus B$ kan snijden met de nieuw toegevoegde cirkel c: gezien ze niet overlappen op de x-as, is de horizontale afstand tussen hen groter dan de som van hun stralen, wat uitsluit dat ze snijden. Algoritme 6 is dus ook correct. □

2.4.3 Complexiteit

Voor de complexiteitsanalyse van dit algoritme bouwen we voort op de complexiteitsanalyse van het kwadratische algoritme. Het enige dat we toevoegen is een check in de binnenste lus: het is nu dus de kwestie te bepalen hoe vaak we kunnen vermijden dat we twee cirkels nakijken. We gebruiken de functie *overlapping* om aan pruning te doen; in de implementatie van ons algoritme heeft deze operatie $log(n)$ complexiteit omdat we T als boom bijhouden. We kunnen echter niet naïef zeggen dat het algoritme nu complexiteit *nlogn* heeft. Immers, neem nu dat we het slechtste geval voor het kwadratische algoritme doortrekken naar de subcheck van dit algoritme, m.a.w., T bevat alle cirkels en die overlappen ook allemaal op de x-as. Dan neigen we in het allerslechtste geval naar $O(n^2 log(n))$: voor elke iteratie van de binnenste lus in het kwadratische algoritme komt er nog een operatie in $O(log(n))$ tijd bij.

We kunnen de complexiteit zelfs nog verfijnen. Merk op dat hoe groter het aantal snijpunten van de cirkels is, hoe groter de uitvoeringstijd. Het aantal cirkels dat in de binnenste lust wordt nagekeken, het aantal cirkels met overlappende intervallen, noemen we R. De complexiteit is dus $O(N \log(N) + R \log(N))$. We weten dat S, het aantal cirkels, van grootteorde R is. We komen dus uit op de volgende complexiteit.

$$O((N + S) \log(N))$$



Figuur 5: Nagekeken cirkels bij algoritme 3

3 Implementatie

In deze sectie beschrijven we de implementatiedetails van de algoritmes en de gebruikte gegevensstructuren. We tonen eveneens stukjes code. We garanderen niet dat deze code zal compileren wanneer ze gekopieerd wordt. De code die hier staat is enigszins vereenvoudigd om de leesbaarheid te vergoten.

3.1 Haskell

We hebben de code geschreven in Haskell. Haskell is een programmeertaal die gefundeerd is op de volgende principes:

1. Haskell is **lui**: een expressie zal enkel geëvalueerd worden als het resultaat van die evaluatie nodig is.
2. Haskell is **puur**: functies in Haskell hebben geen side-effects. Hierdoor is er een garantie dat voor een bepaalde input een functie altijd dezelfde output zal teruggeven, en kunnen programma's vrij eenvoudig geparallelliseerd worden.
3. Haskell is **functioneel**: functies in Haskell zijn in tegenstelling tot talen als Java zgn. *first-class citizens*: we kunnen functies manipuleren als gewone data.

Dit vereenvoudigde het implementeren van de wiskundige aspecten van dit project en verkortte de code in het geheel. Bepaalde aspecten zorgden evenwel voor een aantal problemen.

3.1.1 Evaluatie en timing

Haskell wordt niet per se sequentieel geëvalueerd. Code wordt doorgaans enkel geëvalueerd wanneer het resultaat nodig is voor uitvoer. Een resultaat dat nooit opgevraagd wordt zal dus ook nooit berekend worden. Voorts is het ook zo dat functies die niet aan I/O doen in Haskell niet beschouwd worden als procedures, maar wel als definities van functies. Wanneer we zo'n functie toepassen op haar argumenten, wordt binnen het paradigma dat Haskell gebruikt eigenlijk niets berekend!

We hebben het algoritme zo geïmplementeerd dat de cirkels eerst worden ingelezen, de snijpunten dan worden berekend en pas daarna de snijpunten worden uitgelezen. Enkel het berekenen van de snijpunten wordt dan getimed. De snijpunten worden dus niet meer op een luie manier berekend. Het programma is hierdoor natuurlijk trager geworden.

Voor het timen van onze code gebruiken we het 'Criterion' pakket op hackage.[1] Voor het garanderen van striktheid van de code gebruiken we het 'DeepSeq' pakket op hackage.[2]

3.1.2 Lijsten

Operaties op de ingebouwde lijsten in Haskell hebben de volgende complexiteiten.

Operatie	Complexiteit
pattern matching	$O(1)$
concat	$O(N + M)$
map	$O(N)$
nub (uniques)	$O(N \log(N))$
sort	$O(N \log(N))$

Figuur 6: Operaties op lijsten

[1] `https://hackage.Haskell.org/package/criterion`
[2] `https://hackage.Haskell.org/package/deepseq`

3.2 Naief

Het naieve algoritme maakt gebruik van staartrecursie zodat er nooit twee cirkels maar dan één keer met elkaar vergeleken worden. Dit algoritme werkt steeds in $O(N^2)$ tijd.

```
intersections :: [Circle] -> [Position]
intersections [] = []
intersections [_] = []
intersections l = nub $ go l
    where
        go [] = []
        go [_] = []
        go (c:cs) = concatMap (circlesIntersections c) cs ++ go cs
```

Figuur 7: Functionele code voor het naïeve algoritme

3.3 Kwadratisch

Het kwadratische algoritme maakt geen gebruik van externe datastructuren. Het gebruikt de ingebouwde lijsten van Haskell om de status bij te houden. Dit algoritme heeft een slechtste-geval complexiteit van $O(N^2)$ en een beste-gevalcomplexiteit van $O(N \log(N)$. De 'events' moeten immers nog steeds gesorteerd worden;

```
intersections :: [Circle] -> [Position]
intersections [] = []
intersections [c] = []
intersections cs = nub $ go (sort $ eventPointss cs) []
    where
        go :: [Event] -> [Circle] -> [Position]
        go [e] act = []
        go (Insert c : evl) act
                = concatMap (circlesIntersections c) act
                ++ go evl (c:act)
        go (Delete c : evl) act = go evl (delete c act)
```

Figuur 8: Functionele code voor het kwadratische algoritme

3.4 Linearitmisch

Het linearitmische algoritme maakt gebruikt van een datastructuur genaamd 'intervalmap' om de status bij te houden. Op deze manier kunnen de cirkels met overlappende intervallen kunnen gezocht worden in $O(R \log N)$ tijd (waarbij R het aantal cirkels met overlappende intervallen voorstelt).

```
intersections :: [Circle] -> [Position]
intersections []  = []
intersections [_] = []
intersections cs = nub $ go (sort' eventPointss cs) I.empty
    where
        go :: [Event] -> IntervalMap Position Circle -> [Position]
        go [e] _ = []
        go (Insert c : es) act = intersects ++ next
            where
                intersects
                        = concatMap
                          (circlesIntersections c)
                          overlapping
                overlapping = intersecting act thisInterval

                next = go es newAct
                newAct = insert (interval c) c act

        go (Delete c : es) act = go es newAct
            where newAct = delete (circleInterval c) act
```

Figuur 9: Functionele code voor het linearitmische algoritme

4 Experimenten

In deze sectie beschrijven we de experimenten die we hebben uitgevoerd, samen met de verwachtingen die we stelden van de resultaten.

4.1 Correctheidstoetsen

Om na te kijken of we de algoritmes correct hebben geïmplementeerd hebben we ook een aantal toetsen geïmplementeerd. Eerst wordt het eerste algoritme getoetst aan de hand van de voorbeeld in- en uitvoer in de opgave. Daarna wordt de uitvoer van de andere algoritmes aan de uitvoer van het eerste algoritme getoetst.

4.2 Rauwe data

De eerste groep experimenten die we hebben uitgevoerd zijn puur qualitatief. We meten de uitvoeringstijden van de verschillende algoritmes bij verschillende scaleringen van de straal bij verschillende aantallen cirkels.

4.2.1 Weinig snijpunten

Wanneer we de stralen van de willekeurig gegenereerde cirkels met een klein getal vermenigvuldigen zullen er minder snijpunten zijn tussen de cirkels. Wij kozen voor dit getal 0.001.

In de resulterende grafiek hopen we te zien dat het eerste algoritme het zich niet aantrekt dat de cirkels verschaald zijn. De grafiek zou er dan als een parabool moeten uitzien. Het tweede en derde algoritme zullen zich hopelijk linearitmisch gedragen. De grafiek van deze algoritmes zou er dan moeten uitzien als een weinig-gebogen rechte. De grafieken van het tweede en derde algoritme zouden onder de grafiek van het eerste algoritme moeten liggen voor grote N.

4.2.2 Veel snijpunten

Wanneer we de stralen scaleren met een groot getal zullen er relatief veel snijpunten zijn tussen de cirkels. Wij kozen voor dit getal 1000.

In de grafieken zouden we dan moeten zien dat algoritme één en twee beide een parabool vormen. De grafiek van algoritme drie zou duidelijk boven de grafiek van de andere twee moeten liggen.

4.2.3 3D-plot

Een 3D-plot zou meer duidelijkheid kunnen scheppen, maar is vaak niet erg leesbaar. We hopen te zien dat het eerste algoritme ongeveer even snel werkt onafhankelijk van de scalering van de stralen. Bovendien hopen we te zien dat algoritme twee en drie beter presteren dan het eerste voor kleine scaleringen, en slechter presteren voor grote scaleringen.

4.3 Doubling ratio

Een doubling ratio experiment voeren we uit om enigszins quantitatieve resultaten te bekomen. In de doubling ratio experimenten verdubbelen we het aantal cirkels en berekenen we de verhoudingen van de uitvoeringstijden. De verhouding van de uitvoeringstijd voor dubbel zo veel cirkels ten opzichte van de vorige uitvoeringstijd vertelt ons of ons vermoeden van de complexiteit juist is.

4.3.1 Naïef en kwadratisch

Het naïeve algoritme zou een tijdscomplexiteit van $O(N^2)$ hebben. Dit houdt in de uitvoeringstijd vier keer groter wordt wanneer we het aantal cirkels verdubbelen. Het tweede algoritme heeft een slechtste-geval tijdscomplexiteit van $O(N^2)$. Het slechtste geval is wanneer elke twee cirkels snijden. Voor grote scaleringen zou de verhouding dus ook naar 4 moeten convergeren.

$$\lim_{N\to\infty} \frac{(2N)^2}{N^2} = \lim_{N\to\infty} \frac{4N^2}{N^2} = 4$$

4.3.2 Linearitmisch

Voor het derde algoritme voorspellen we een tijdscomplexiteit van $O(N + S)\log(N)$ waarbij S het aantal snijpunten is. Wanneer er (bijna) geen snijpunten zijn verwachten we een verhouding van 2.

$$\lim_{N\to\infty} \frac{(2N + S)\log(2N)}{(N + S)\log(N)} = \lim_{N\to\infty} \frac{2N\log(2N)}{N\log(N)} = \lim_{N\to\infty} \frac{2\log(2N)}{\log(N)}$$

$$= \lim_{N\to\infty} 2\frac{\log(2) + \log(N)}{\log(N)} = \lim_{N\to\infty} 2\left(1 + \frac{\log(2)}{\log(N)}\right) = 2$$

Wanneer (bijna) elke twee cirkels elkaar snijden ligt S dicht bij N^2. We verwachten dan ook een verhouding van 4.

$$\lim_{N\to\infty} \frac{(2N)^2\log(2N)}{N^2\log(N)} = \lim_{N\to\infty} 4\frac{N^2\log(2N)}{N^2\log(N)} = \lim_{N\to\infty} 4\frac{\log(2) + \log(N)}{\log(N)}$$

$$= \lim_{N\to\infty} 4\left(1 + \frac{\log(2)}{\log(N)}\right) = 4$$

5 Resultaten

In deze sectie tonen en bespreken we de bekomen resultaten. De resultaten lijken veelbelovend, afgezien van de resultaten van de doubling ratio experimenten. De correctheidstoetsen zijn succesvol, alsook de rauwe data-experimenten. De doubling ratio experimenten geven onverwachte verhoudingen.

5.1 CorrectheidToetsen

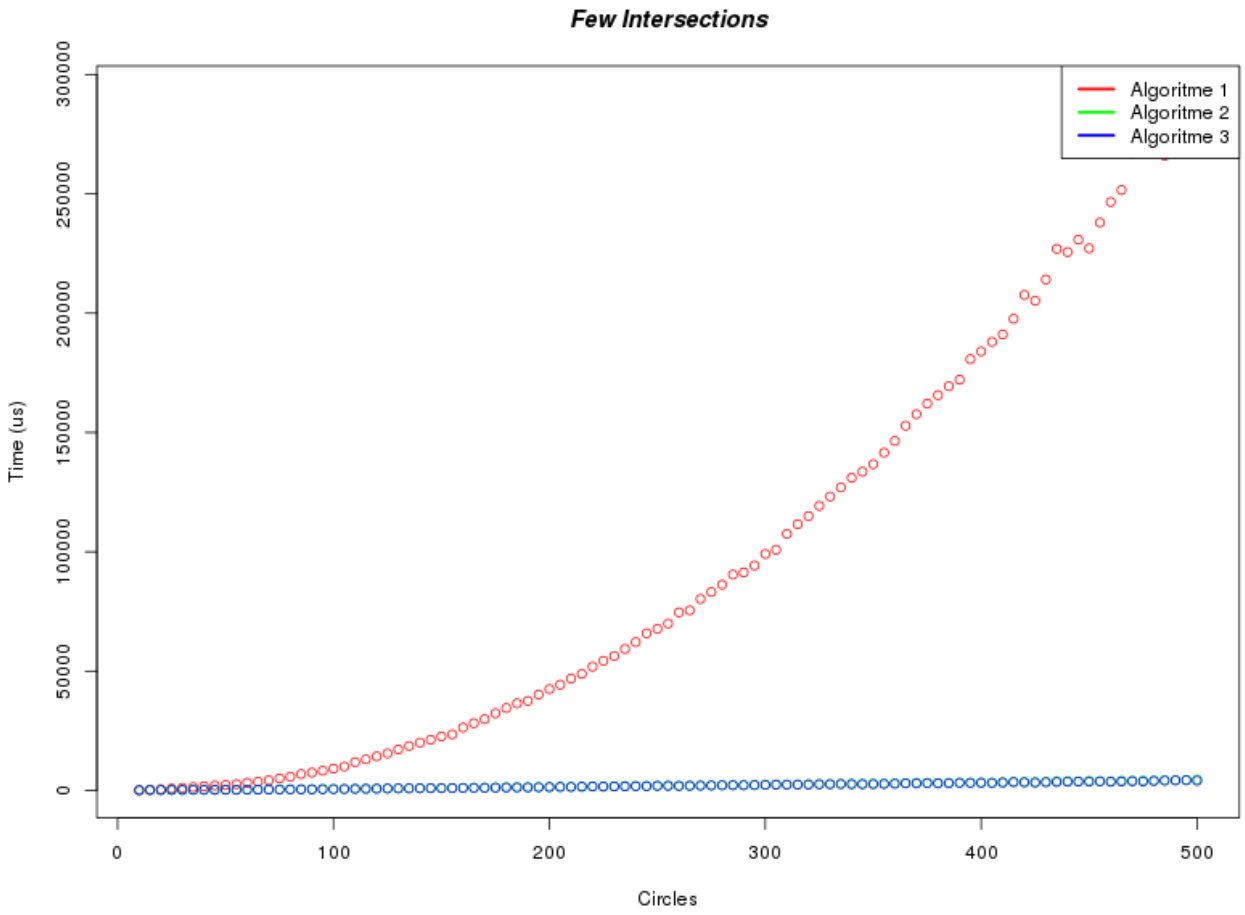
De uitvoer van de correctheidstoetsen is als volgt:

```
BASETEST  SUCCESS
testcase_2_010.txt  SUCCESS  100/100
testcase_3_010.txt  SUCCESS  100/100
```

Empirisch gezien zijn onze implementaties correct en volledig.

5.2 Rauwe data

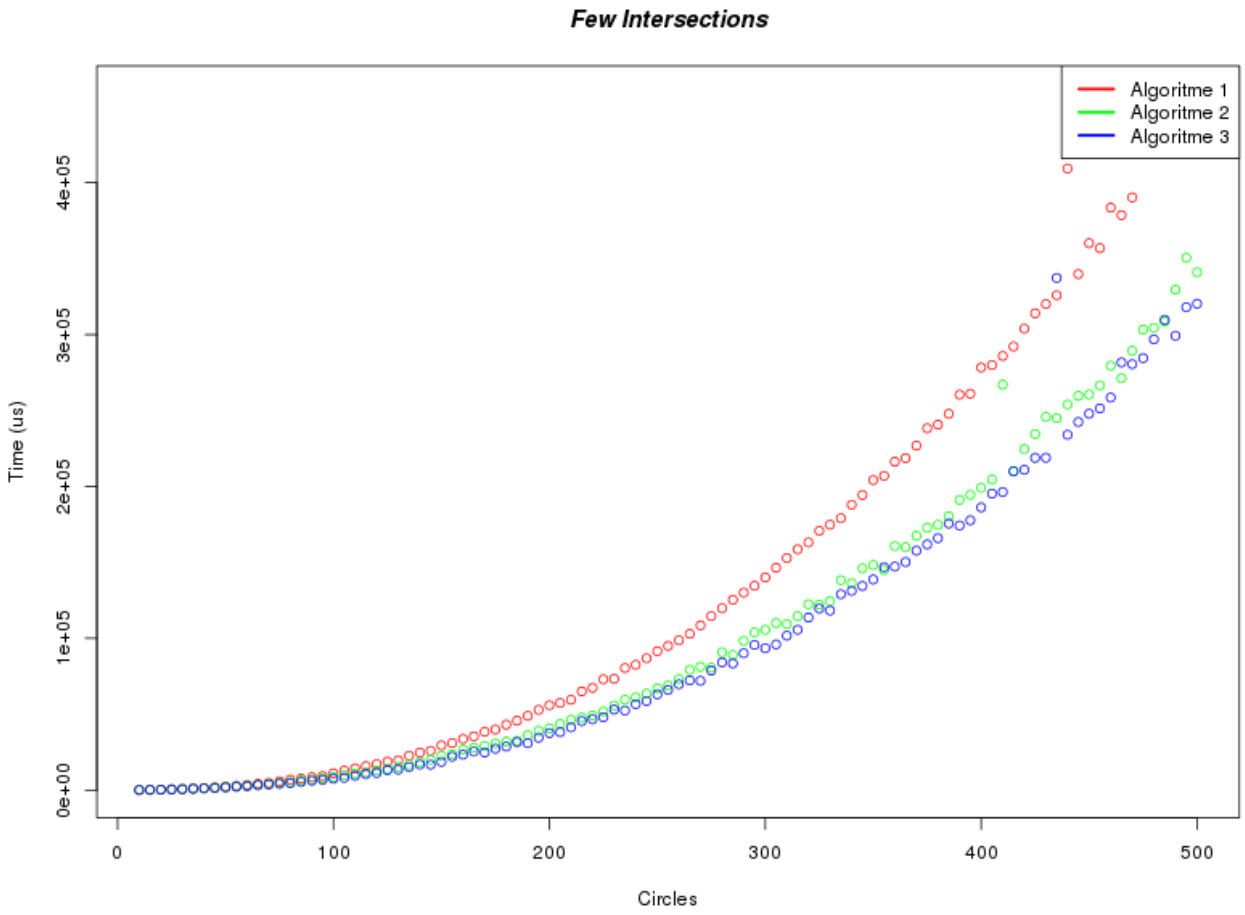
5.2.1 Weinig Snijpunten



Figuur 10: Uitvoeringstijden bij gevallen met weinig snijpunten.

In figuur 10 zijn de resultaten uiteengezet van het eerste rauwe data-experiment. We zien dat het naïeve algoritme veel slechter presteert dan de andere twee. Het eerste algoritme wordt niet beïnvloedt door het aantal snijpunten, lijkt het.

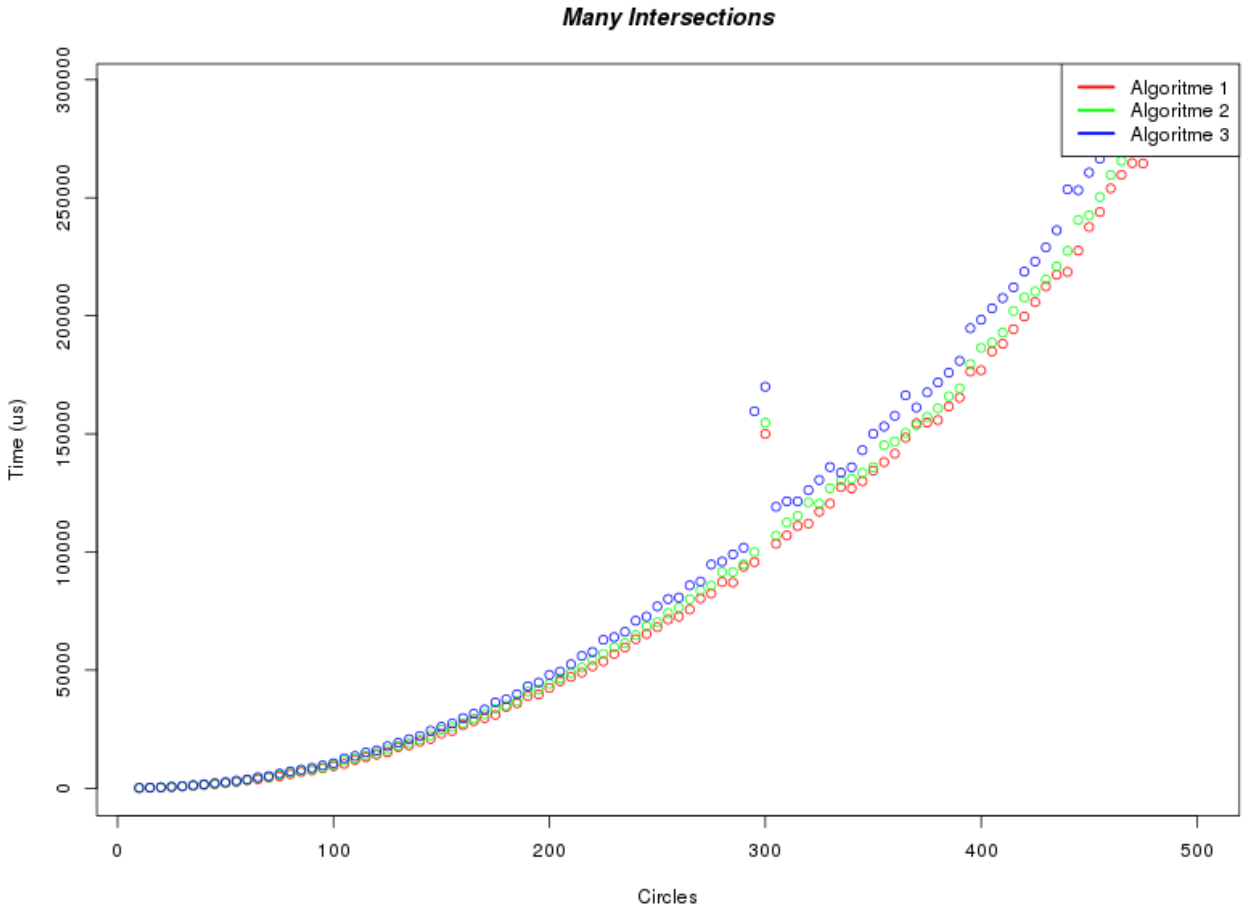
5.2.2 Gemiddeld geval



Figuur 11: Uitvoeringstijden bij een gemiddeld geval snijpunten.

In figuur 11 zijn de resultaten uiteengezet voor een gemiddeld geval. We zien dat algoritme één nog steeds ongeveer even snel werk. We zien ook dat algoritme twee en drie sneller zijn, zoals verwacht.

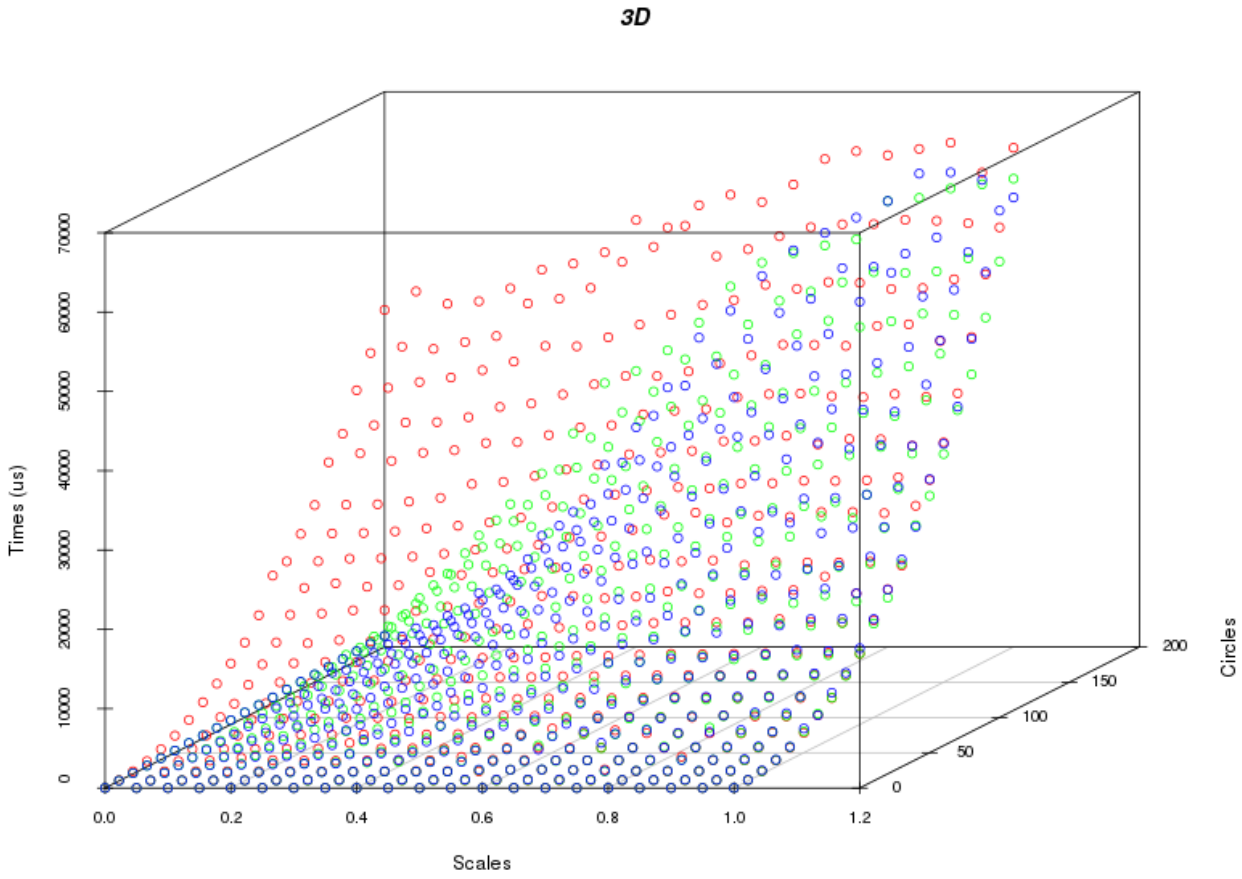
5.2.3 Veel snijpunten



Figuur 12: Uitvoeringstijden bij gevallen met relatief veel snijpunten.

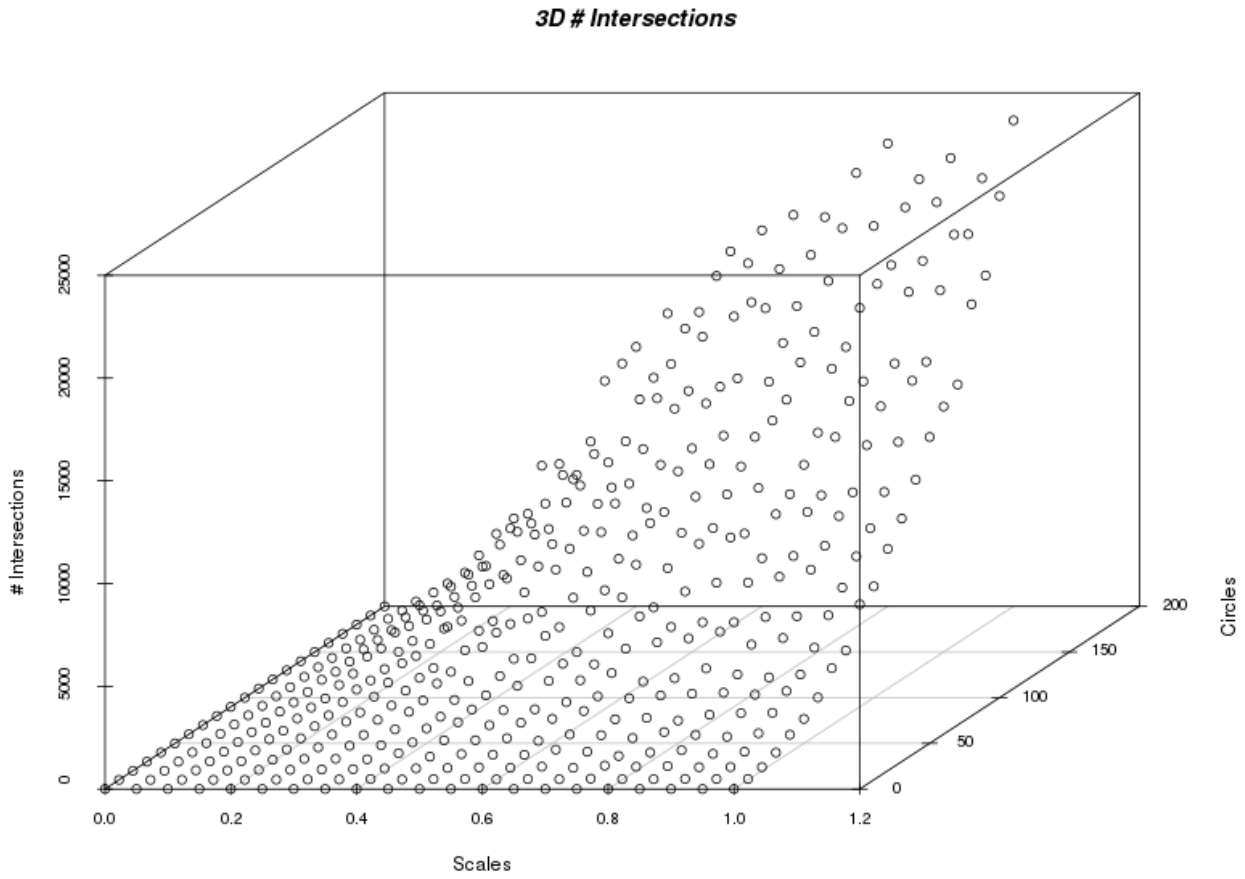
In figuur 12 zijn de resultaten uiteengezet van het tweede rauwe data-experiment. We zien dat algoritme één hier beter presteert dan de andere twee. Algoritme twee en drie moeten dezelfde snijpunten berekenen maar hebben te kampen met enige 'overhead'.

5.2.4 3D plot



Figuur 13: Een 3D grafiek van de uitvoeringstijden in functie van de scalering en het aantal cirkels

De resultaten van het laatste rauwe data-experiment staan in figuur 13. Het is duidelijk dat algoritme één ongeveer evenveel tijd nodig heeft onafhankelijk van de scalering van de stralen. We zien bovendien dat algoritme drie slechter presteert dan algoritme één bij grote scaleringen en beter bij kleine scaleringen.



Figuur 14: Aantal snijpunten

In figuur 14 staan het aantal snijpunten geplot voor elk geval, in plaats van de uitvoeringstijden. We zien dat de grafiek van de uitvoeringstijden van algoritme drie mooi dezelfde vorm heeft.

5.3 Doubling ratio

De resultaten van een doubling ratio experiment zetten we in een tabel. Op de bovenste rij van de tabel staat het aantal cirkels. In de meest linkse kolom staan de scaleringen van de straal. De getallen in de tabel stellen de verhouding voor tussen de uitvoeringstijd bij deze probleemgrootte ten opzichte van het probleemgrootte er links van.

5.3.1 Naief

	20	40	80	160	320	640	1280
0.000	3.7	4.1	4.7	4.4	4.3	4.3	4.3
0.001	3.5	6.5	2.8	4.2	4.5	4.6	4.3
0.500	3.4	3.0	4.4	4.6	5.4	4.9	5.0
1.000	4.1	4.0	4.6	4.8	5.7	5.4	4.9

Tabel 1: Doubling ratio 1

tabel 1 toont de resultaten van het doubling ratio exeriment voor algoritme één. Het is makkelijk te zien dat de ratio zal convergeren naar 4 voor een groot aantal cirkels. Dit bovendien onafhankelijk

van de scalering van de stralen. Wanneer er veel snijpunten zijn lijkt het algoritme last te hebben van enige overhead. We weten niet waarom, maar algoritme één is zodanig eenvoudig dat we denken dat het probleem misschien bij Haskell zelf ligt.

5.3.2 Kwadratisch

	20	40	80	160	320	640	1280
0.000	2.1	2.1	2.2	2.3	2.3	2.1	2.2
0.001	1.9	2.3	2.0	2.5	2.4	2.3	2.5
0.500	3.2	2.8	3.6	4.9	5.1	4.9	5.0
1.000	3.7	4.0	4.5	4.5	5.7	5.1	5.1

Tabel 2: Doubling ratio 2

In tabel 2 de resultaten van het doubling ratio experiment voor algoritme twee. Voor kleine scaleringen van de straal zien we dat de ratio rond 2.2 schommelt. Wanneer er weinig snijpunten zijn is dit algoritme dus inderdaad beter dan kwadratisch. Voor grote scaleringen zou de verhouding naar 4 moeten gaan, maar ook deze erhoudingen lijken rond 5.2 te schommelen. Opnieuw kunnen we niet verklaren waarom.

5.3.3 Linearitmisch

	20	40	80	160	320	640	1280
0.000	2.2	2.2	2.2	2.3	2.2	2.1	2.2
0.001	2.1	2.2	2.2	2.3	2.8	1.8	2.4
0.500	3.3	3.4	4.1	4.4	5.2	5.2	5.0
1.000	3.4	4.0	4.2	4.6	5.5	5.1	5.1

Tabel 3: Doubling ratio 3

In tabel 3 staan de resultaten van het laatste doubling ratio experiment. We zien dat de ratio rond 2.2 schommelt voor kleine scaleringen. Dit komt overeen met onze voorspellingen. Bij kleine S zal de uitvoeringstijd van het algoritme namelijk linearitmisch groeien. Bij grote scaleringen schommelt de ratio boven 4 zoals verwacht van een $O(N^2 \log(N))$ algoritme. We weten echter niet of dit is omdat alles in orde is of omdat hetzelfde probleem zich stelt als bij de eerdere doubling ratio experimenten. Opnieuw zouden we geen verklaring hebben voor de verhoudingen die rond 5.1 schommelen.

6 Besluit

Het is mogelijk om alle snijpunten te vinden van een verzameling cirkels in $O(N + S)\log(N)$ tijd. Dit is echter niet steeds wenselijk. Indien we op voorhand kunnen weten of er eerder veel of eerder weinig snijpunten zullen zijn, kunnen we het meest gepaste algoritme kiezen. Algoritme drie is duidelijk beter wanneer er weinig snijpunten zijn. Algoritme één en twee zijn duidelijk beter wanneer er veen snijpunten zijn. In het algemeen lijkt het het best om voor algoritme twee te kiezen.

7 Reflectie

7.1 Onverwachte verhouding

We zouden graag een verklaring hebben gevonden voor de onverwachte verhoudingen in de doubling ratio experimenten. Hiervoor moeten we echter experimenten kunnen uitvoeren met veel grotere aantallen cirkels. Bovendien hadden we eventueel nog een betere datastructuur kunnen gebruiken.

7.2 Haskell sequence

In heel het programma gebruiken we de ingebouwde lijsten van Haskell. Het datatype 'Sequence' heeft echter, in de meeste gevallen en voor de meeste operaties, een betere slechtste geval-complexiteit. De totale uitvoeringstijd voor grote problemen zouden we zo nog kunnen inkorten.

7.3 Compilatie optimalisaties

De GHC' compiler die we gebruikt hebben biedt nog optimalisaties aan die we niet gebruikt hebben. Zo hebben we geprobeerd om de 'llvm' vlag aan te zetten, maar omdat we daar een ongeteste versie van gebruikten resulteerde dit in compilatie fouten.

A Voorbeeldgeval