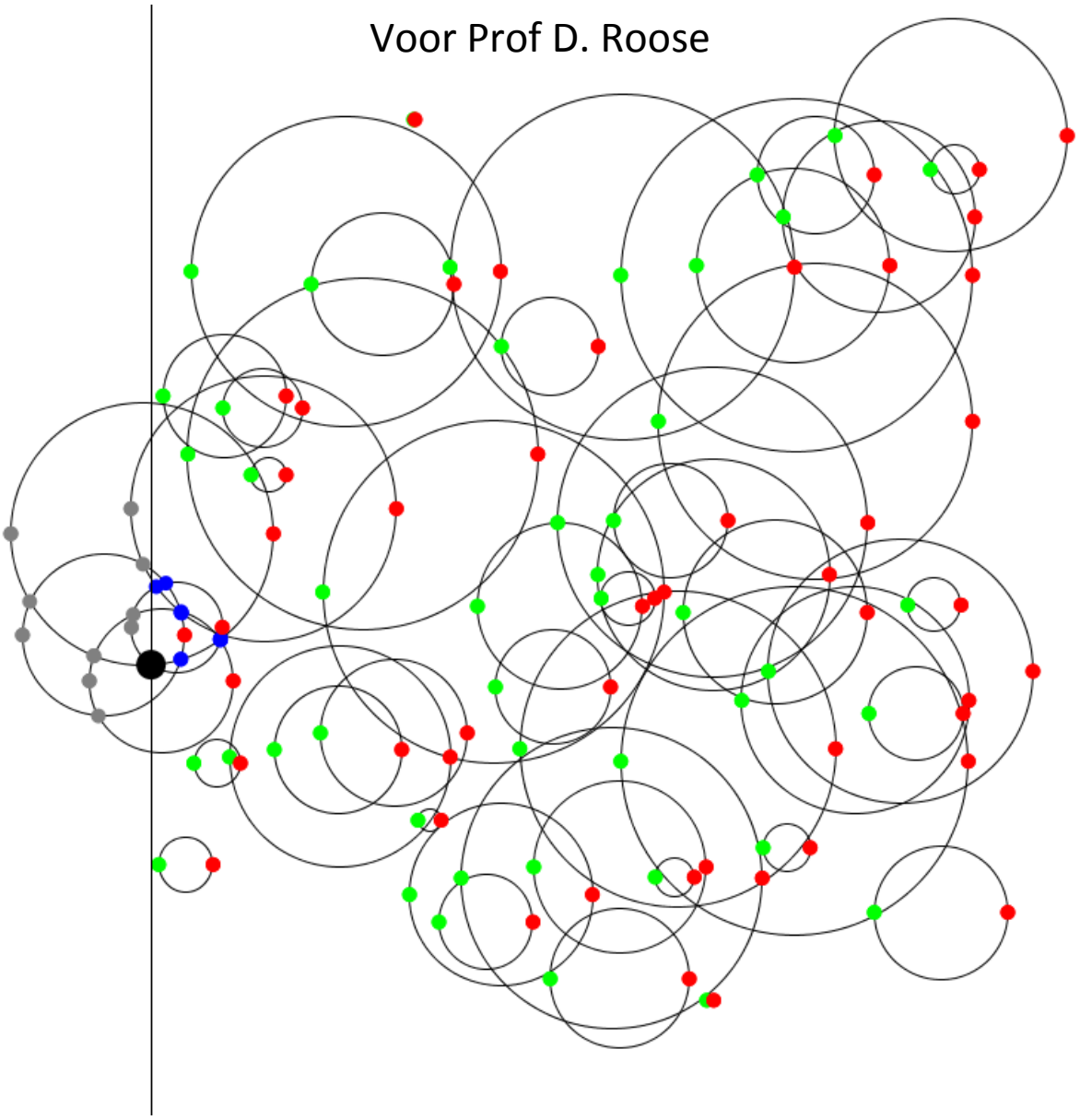
VERSLAG PRACTICUM: SNIJDENDE CIRKELS

Toepassingen van meetkunde in de informatica

Van Jorik De Waen (r0303087, 2Ba Inf)

Voor Prof D. Roose



Code Algoritmes

Algoritme 1

```
function solve(Circles)
  n ← #Circles
  I ← ∅
  for i ← 1 to n do

    for j ← i + 1 to n do
      I ← I ∪ intersections(Circles_i, Circles_j)
    endfor

  endfor

  return I

end
```

Het eerste algoritme is het meest eenvoudige algoritme. Het houdt geen rekening met de posities van de aparte cirkels en controleert voor elke cirkel of er snijpunten zijn met alle andere cirkels. De code krijgt een lijst van alle cirkels $Circles$ binnen. n is het aantal cirkels in die lijst en I wordt geïnitialiseerd als een lege lijst die de snijpunten gaat bevatten. De buitenste for-lus gaat de indexen van alle cirkels in de lijst afgaan. De binnenste for-lus gaat voor elke index van de buitenste lus alle indexen die *hoger* zijn afgaan. In de binnenste for-lus worden alle snijpunten tussen de twee cirkels met de indexen van de binnenste en buitenste for-lus toegevoegd aan de lijst van snijpunten.

De correctheid van deze code is vrij evident: Na elke iteratie van de buitenste lus zijn alle snijpunten die op de cirkel liggen die bij die iteratie hoort, toegevoegd. Bij de eerste iteratie gaat de tweede lus alle cirkels buiten de eerste af. Alle snijpunten op de eerste cirkel zijn dus bekend. Bij de volgende iteratie kunnen we de eerste cirkel overslaan aangezien elk snijpunt met die cirkel al gevonden is. Bij elke iteratie is dus gekend dat alle snijpunten op cirkels met een index kleiner dan de huidige cirkel al bekend zijn. Het volstaat dus om enkel de snijpunten te berekenen met cirkels die een hogere index hebben. Hiermee is het dus bewezen dat alle snijpunten zijn worden na afloop van de buitenste lus.

Algoritme 2

```
function solve(Circles)
  PriorityQueue events ← ∅
  active ← ∅
  I ← ∅

  foreach C in Circles do
    events ← events ∪ event(ADD, C)
    events ← events ∪ event(REMOVE, C)
  endforeach

  while not empty(events) do
    E ← pop(events)
    C ← circle(E)
    if type(E) = ADD do
      foreach A in active do
        I ← I ∪ intersections(C, A)
      endforeach
      active ← active ∪ C
    else do
      active ← active \ C
    endif
  endwhile

  return I

end
```

Het tweede algoritme maakt gebruik van een sweepline. Deze is niet expliciet geïmplementeerd maar het gedrag hiervan wordt gemodelleerd met een PriorityQueue. Het algoritme wordt geïnitialiseerd met een lege PriorityQueue van events, een lege lijst van actieve cirkels en een lege lijst met de gevonden snijpunten.

De eerste stap is het toevoegen van alle events. De sweepline gaat van links naar rechts over het veld bewegen. Wanneer deze een cirkel tegenkomt wordt die cirkel actief. Wanneer de sweepline een cirkel verlaat wordt de cirkel weer inactief. Voor elke cirkel zijn er dus twee events: Eén wanneer een cirkel actief wordt en een tweede wanneer de cirkel terug inactief wordt. Deze twee events worden voor elke cirkel toegevoegd in de eerste foreach lus. Deze events worden gesorteerd op de positie van de sweepline wanneer ze voorkomen. Het ADD event zal aan de linkerkant van de cirkel liggen terwijl het REMOVE event aan de rechterkant ligt.

In de tweede stap wordt elke event in de PriorityQueue afgegaan. Als het een ADD event is worden de snijpunten tussen de bijbehorende cirkel van het event en alle actieve cirkels berekend en toegevoegd aan de lijst van snijpunten. De cirkel wordt vervolgens aan de lijst van actieve cirkels toegevoegd. Als het een REMOVE event is dan wordt de bijbehorende cirkel uit de lijst van actieve cirkels gehaald. Als alle events afgehandeld zijn wordt tenslotte de lijst van gevonden snijpunten teruggegeven.

De correctheid van de code kan als volgt bewezen worden:

Iedere cirkel krijgt exact één ADD en REMOVE event. Het volstaat om te bewijzen dat na het REMOVE event van een cirkel alle snijpunten met die cirkel bekend zijn. De eerste cirkel wordt aan de lijst van actieve cirkels toegevoegd zonder dat er snijpunten berekend worden. Voor elke cirkel die toegevoegd wordt gaan de snijpunten met alle actieve cirkels worden berekend. Terwijl een cirkel in de lijst van actieve cirkels zit zullen dus zeker alle snijpunten worden gevonden met cirkels die dan worden toegevoegd. Wanneer het REMOVE event van een cirkel aan bod komt liggen alle cirkels die nog niet toegevoegd zijn geweest zeker meer naar rechts dan het meest rechtse punt van die cirkel. Moest dit niet zo zijn zou het ADD event van die cirkel al geweest zijn. Bijgevolg zijn alle snijpunten voor een cirkel gevonden wanneer het REMOVE event aan bod komt. Aangezien iedere cirkel een REMOVE event heeft zullen dus alle snijpunten gevonden worden.

Algoritme 3

```
function solve(Circles)
  PriorityQueue events ← ∅
  BST active ← ∅
  I ← ∅

  foreach C in Circles do
    events ← events ∪ event(ADD, C_TOP)
    events ← events ∪ event(REMOVE, C_TOP)
    events ← events ∪ event(ADD, C_BOTTOM)
    events ← events ∪ event(REMOVE, C_BOTTOM)
  endforeach

  while not empty(events) do
    E ← pop(events)
    if type(E) = ADD do
      S ← segment(E)
      active ← active ∪ S
      I ← I ∪ intersections(S, above(S)) ∪ intersections(S, below(S))
      add_swap_events(intersections(S, above(S)) ∪ intersections(S, below(S)))
    else if type(E) = REMOVE do
      S ← segment(E)
      I ← I ∪ intersections(below(S), above(S))
      active ← active \ S
      add_swap_events(below(S), above(S))
    else if type(E) = SWAP do
      top ← top_segment(E)
      bottom ← bottom_segment(E)
      I ← I ∪ intersections(bottom, above(top))
      I ← I ∪ intersections(top, below(bottom))
      add_swap_events(intersections(bottom, above(top)))
      add_swap_events(intersections(top, below(bottom)) )
    endif
  endwhile

  return I

end
```

Het derde algoritme is het meest geavanceerde. Er wordt weer een sweepline gebruikt zoals in het tweede algoritme, maar deze keer worden de snijdingen anders berekend. Eerst en vooral worden de cirkels opgedeeld in segmenten: de bovenste helft en de onderste helft. Dit is nodig omdat bij dit algoritme de actieve segmenten gesorteerd worden op de y waarde van het snijpunt met de sweepline. Gebruik makende van dit moet er bij het toevoegen enkel de snijpunten met de directe bovenbuur en onderbuur van het segment. Voor elk gevonden snijpunt wordt er een nieuw SWAP Event aangemaakt. Dit is een event die de snijding voorstelt. Het is enkel mogelijk om te snijden met een segment dat een directe boven- of onderbuur is vlak voor het snijpunt. De enige manier om de

boven- en onderburen te veranderen is door nieuwe segmenten toe te voegen, segmenten te verwijderen of bij een snijpunt. Enkel bij de drie events is het dus nodig om buren te berekenen en enkel de directe boven en onderbuur moeten berekend worden. Bij het toevoegen worden zoals eerder vermeld de buren van het nieuwe segment gecontroleerd. Bij het verwijderen worden de buren van het segment dat verwijderd wordt onderling gecontroleerd. Bij een snijpunt wordt de bovenbuur van het bovenste segment met het onderste segment, en de onderbuur van het onderste segment met het bovenste segment van de snijding gecontroleerd. Door dit te doen zullen alle snijpunten gevonden worden.

De tijdscomplexiteit van dit algoritme hangt fel af van het aantal snijpunten. Met een PriorityQueue is het mogelijk om in log(n) tijd toe te voegen en te verwijderen. Aangezien elke cirkel in twee segmenten wordt opgedeeld en elk segment twee events krijgt (ADD en REMOVE) zijn er al $4N$ toevoegingen aan de PriorityQueue nodig. Verder wordt ook elk snijpunt toegevoegd, dus in totaal worden er $4N + S$ events aan de PriorityQueue. Elke event wordt ook nog uitgelezen dus dat brengt het totaal op $2(4N + S)$ interacties met de PriorityQueue. Door het $log(n)$ karakter wordt dit dus $O((N + S)log(N))$.

Bij de implementatie van het derde algoritme kwamen er veel bugs naar boven en het algoritme geeft dus niet alle snijpunten terug. Vermoedelijk ligt het aan sorteerfouten in de boom die alle segmenten bijhoudt. Hierdoor worden de verkeerde buren gecontroleerd en worden er dus snijpunten gemist. Als poging om de fout te vinden heb ik een visualisatie ingebouwd die toont wat het algoritme doet. ADD events worden weergegeven als groene bollen, REMOVE als rood en SWAP als blauw. Het actieve event is de grote zwarte bol. Events die al geweest zijn worden grijs. Het voorblad toont een voorbeeld van de visualisatie.

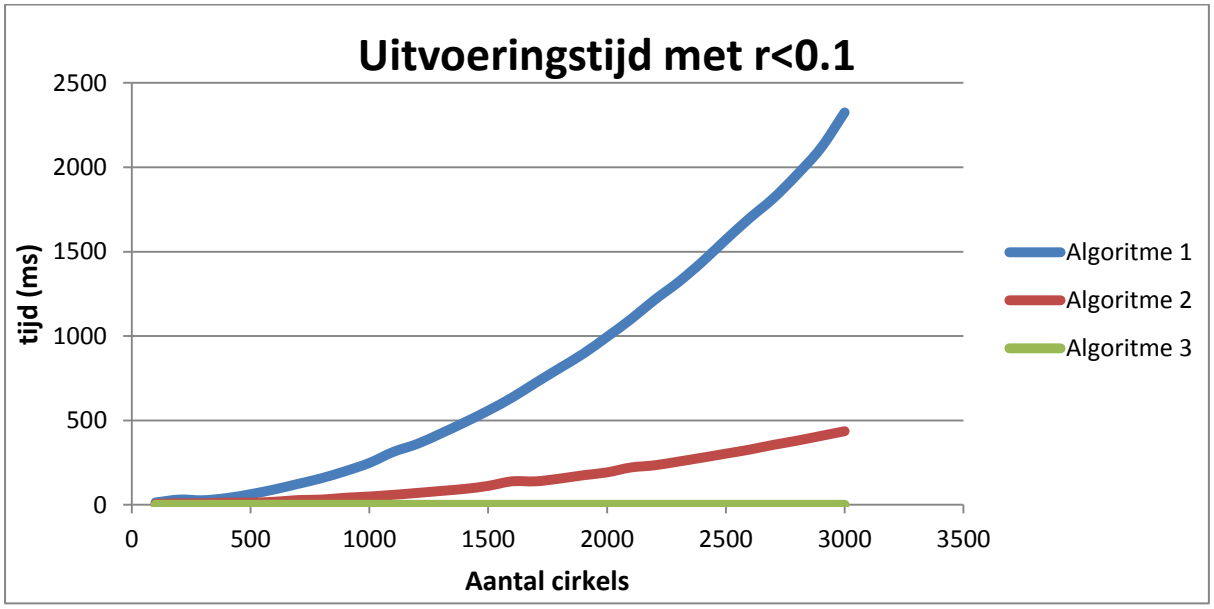
De visualisatie kan worden opgeroepen door een image argument mee te geven na het inputbestand: “java –jar code.jar input.txt image”. De visualisatie speelt vanzelf af, om te kunnen pauzeren na elke stap moet er met breakpoints in een IDE gewerkt worden. Ook wordt er na afloop een afbeelding met alle gevonden snijpunten opgeslagen (en weergegeven) als output.png.

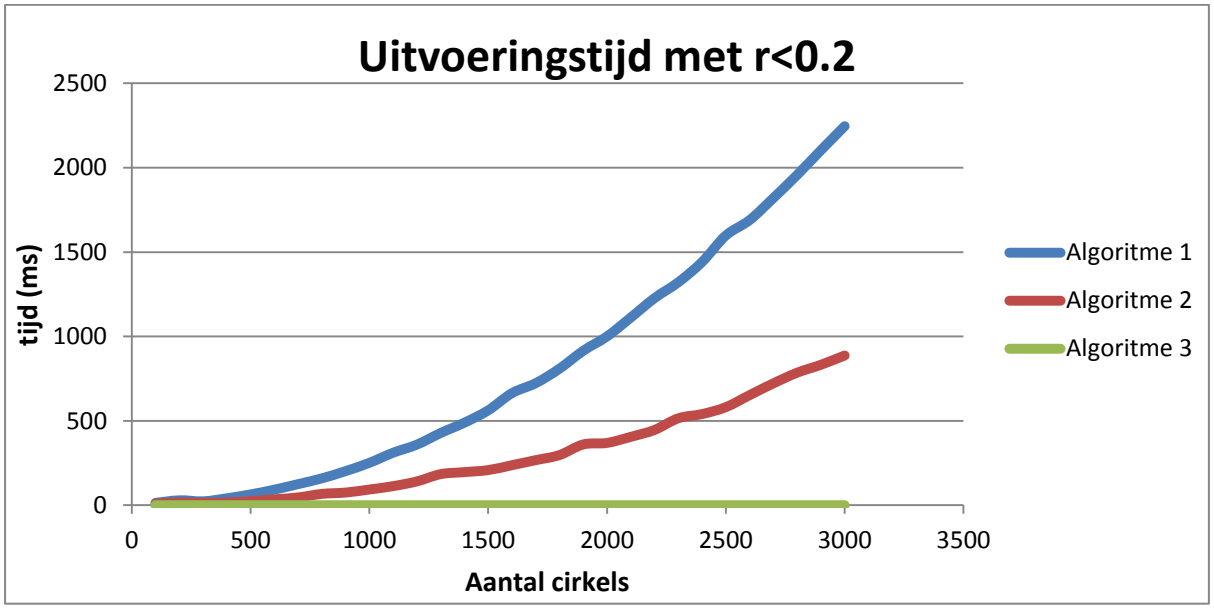
Aangezien het algoritme niet correct werkt zijn tijdsmetingen niet zinnig. De resultaten van het derde algoritme zijn dus niet gebruikt in de experimenten.

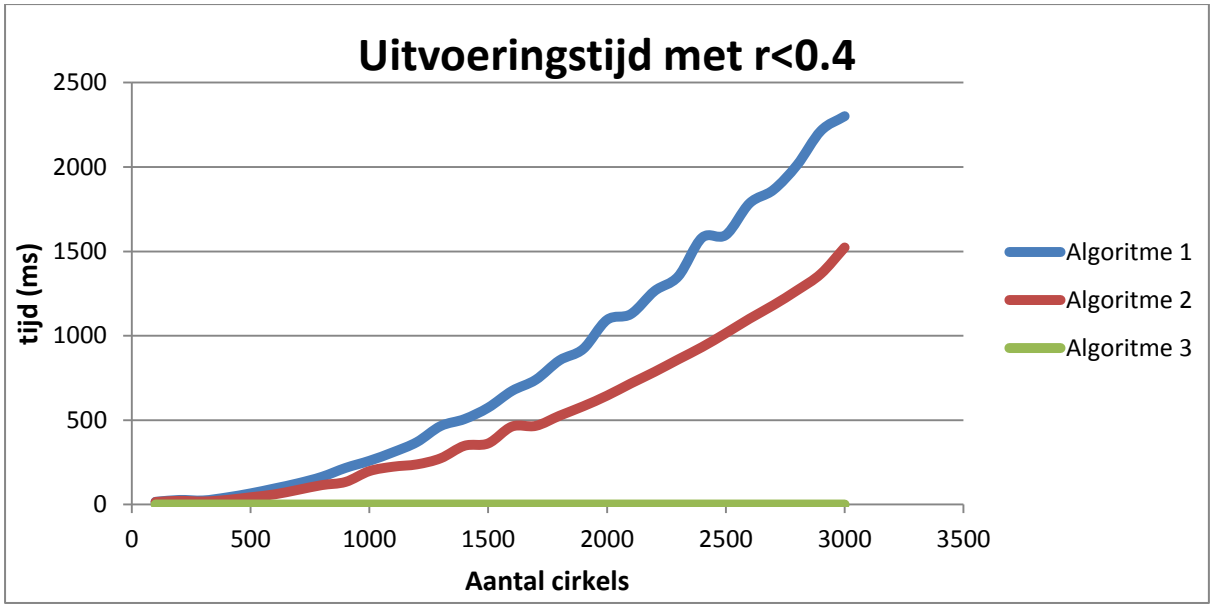
Experimenten

Experiment 1

Bij het eerste experiment worden er variabele hoeveelheden cirkels willekeurig gegenereerd met een x en y coördinaat van het middelpunt tussen 0 en 1. De straal van de cirkels wordt ook willekeurig gegenereerd. Het experiment wordt drie keer herhaald voor een maximum straal van 0.1, 0.2 en 0.4 om het effect van een grotere straal (en dus meer snijpunten) te analyseren. Om de efficiëntie van de aanpak van de algoritmes te kunnen analyseren moet de uitvoeringstijd van elke actie stabiel worden gehouden. Hierdoor wordt er een belangrijke optimalisatie niet gebruikt: Bij het berekenen van de snijpunten tussen twee cirkels kan doormiddel van de locaties en stralen van de cirkels heel snel berekend worden of er wel een snijding zal zijn.



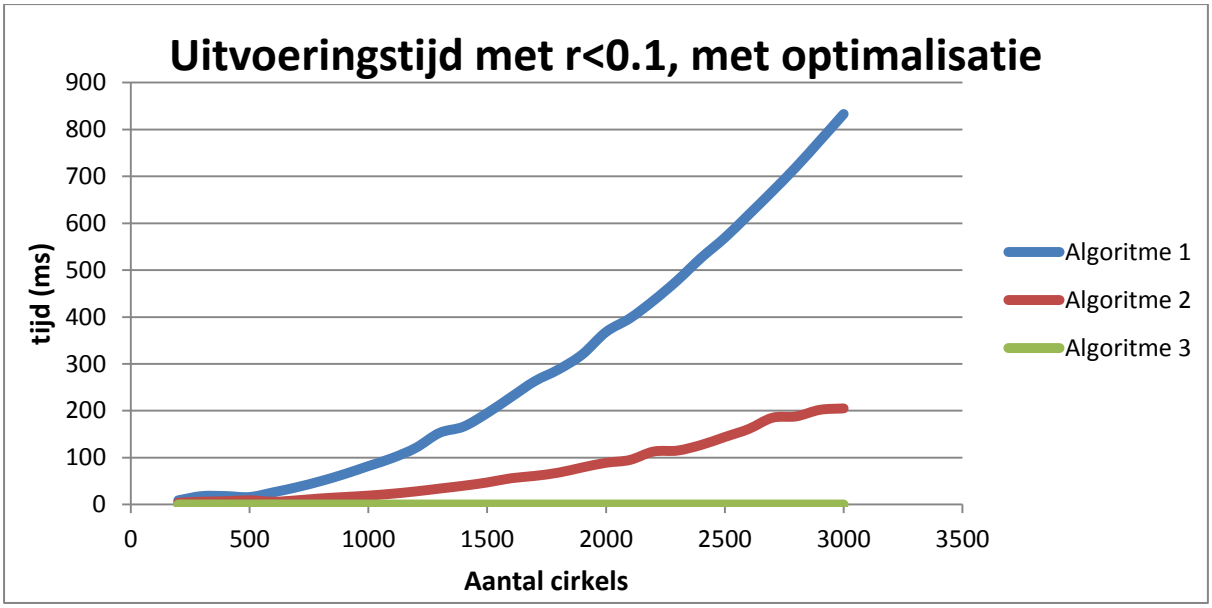


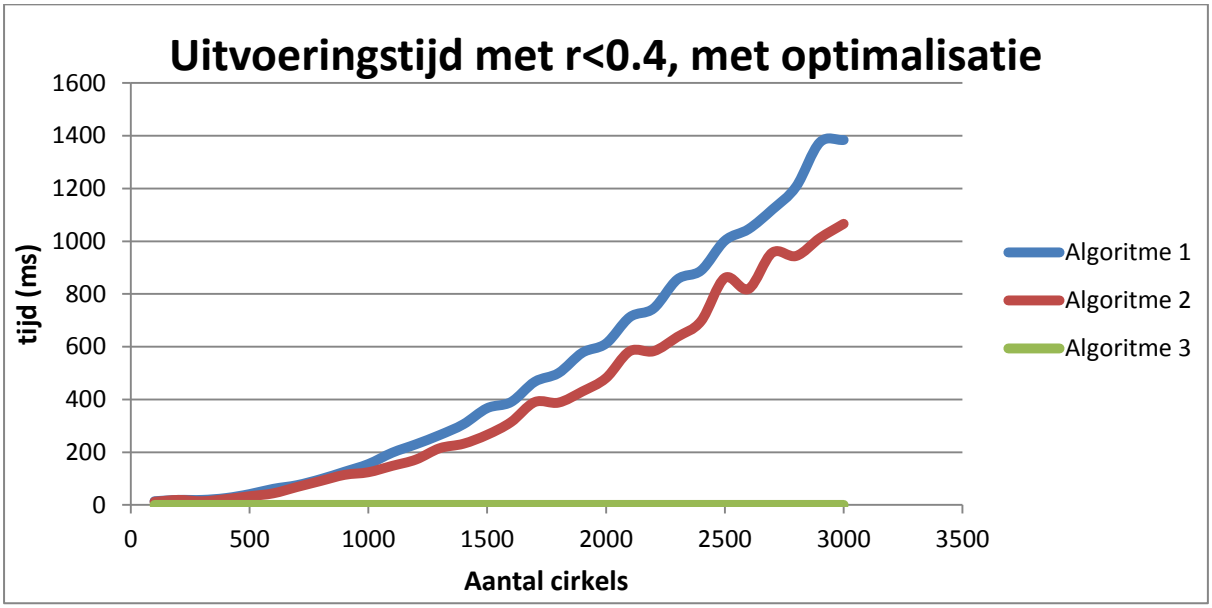


De straal van de cirkels heeft duidelijk een invloed op de uitvoeringstijd van het tweede algoritme, maar niet op de uitvoeringstijd van het eerste. Als de straal klein is werkt algoritme 2 veel sneller dan het eerste algoritme. Het tweede algoritme stijgt bijna lineair bij een straal van maximum 0.1. Dit kan verklaard worden door het feit dat het tweede algoritme actieve cirkels bijhoudt en dus veel minder snijpunten moet berekenen. Wanneer de stralen van de cirkels groter worden gaan de cirkels steeds meer overlappen en verdwijnt dit voordeel. Bij een maximum straal van 0.2 liggen de uitvoeringstijden al een stuk dichter tegen elkaar en bij een straal van 0.4 werken ze bijna even snel. De absolute uitvoeringstijd van het eerste algoritme is zoals verwacht onafhankelijk van de straal van de cirkels.

Experiment 2

In het tweede experiment wordt de impact van de optimalisatie die weggelaten was uit het eerste experiment berekend. Het experiment is volledig identiek, alleen wordt er bij het berekenen van de snijpunten eerst gecontroleerd of de cirkels wel daadwerkelijk kunnen snijden. Het experiment is uitgevoerd met een maximum straal van 0.1 en 0.4



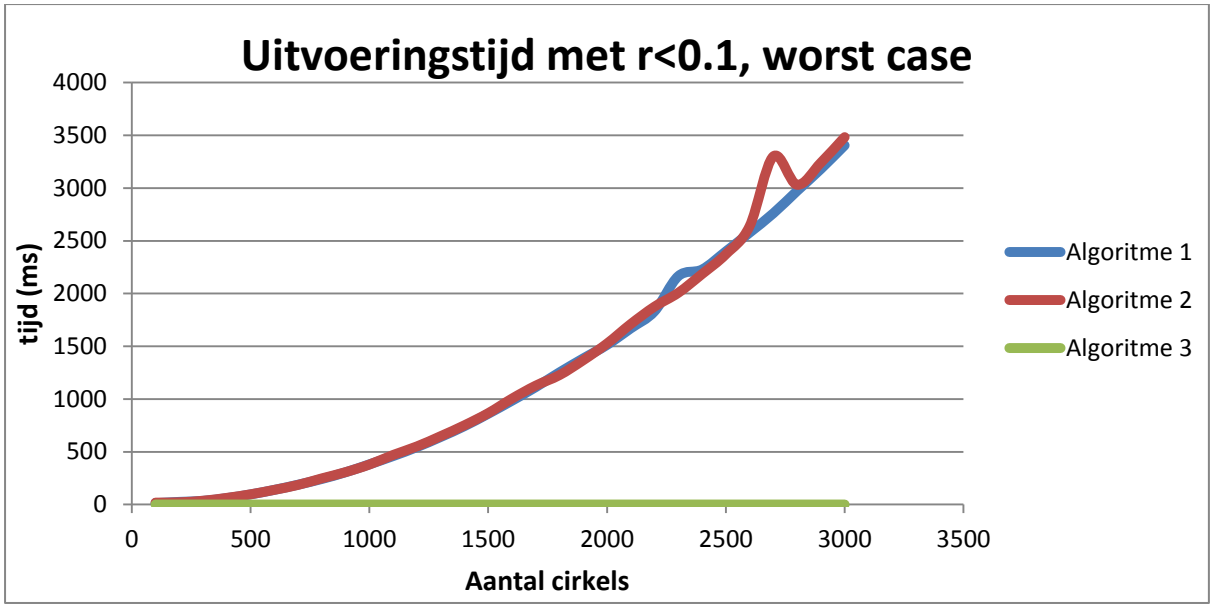


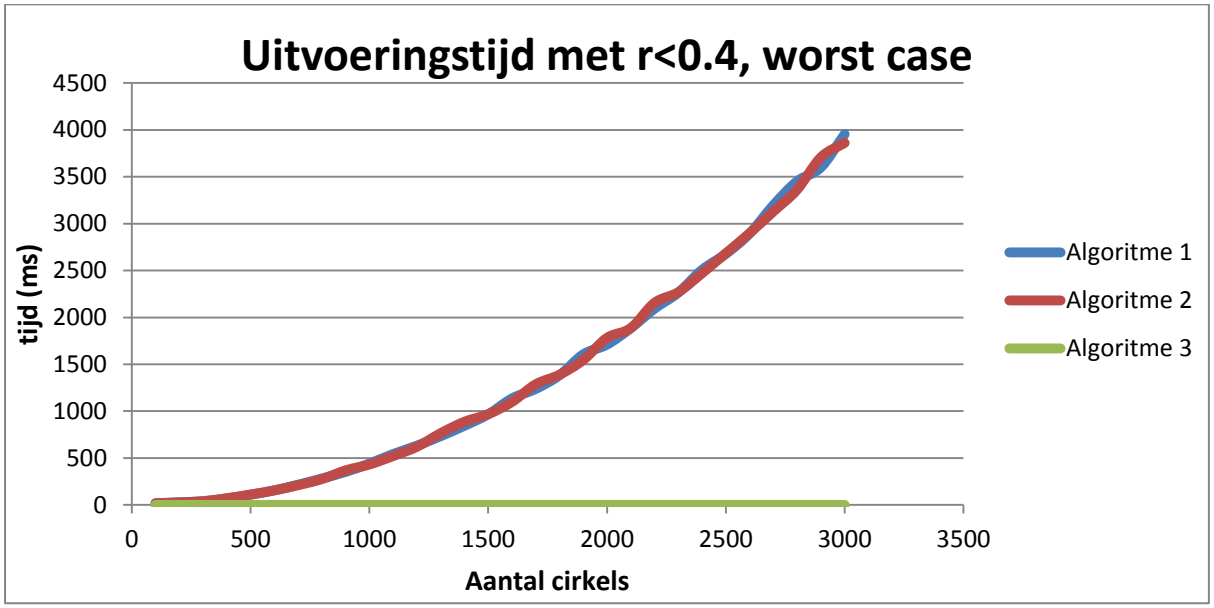
Bij een kleine straal geeft deze optimalisatie voor beide algoritmes een groot verschil. Het eerste algoritme is ruwweg 2.5x sneller en het tweede algoritme is ongeveer dubbel zo snel. Hoewel het absolute verschil tussen de uitvoeringstijden sterk gedaald is blijft het relatieve verschil ongeveer hetzelfde. Bij een straal van maximum 0.4 is er een grotere verandering: Het eerste algoritme scoort bijna even goed als het tweede.

Verder valt het ook op dat de absolute rekentijd van beide algoritmes gestegen is bij de grotere straal. Dankzij de optimalisatie kan veel rekenwerk worden overgeslagen als de cirkels niet snijden. Als er minder snijpunten zijn zullen er dus meer berekeningen kunnen worden overgeslagen. Met deze optimalisatie is het eerste algoritme dus wel gevoelig voor het aantal snijpunten.

Experiment 3

In dit experiment wordt het slechtste geval scenario voor het tweede algoritme getest. Hierbij zijn de x coördinaten van de middelpunten van alle cirkels gelijk. De verwachting is dat de sweepline hier geen voordeel oplevert.





De uitvoeringstijden van de twee algoritmes zijn identiek, zoals verwacht. Omdat alle cirkels boven elkaar liggen moeten ze toch nog altijd met alle andere cirkels vergeleken worden. In dit geval is het tweede algoritme effectief hetzelfde als het eerste.