Verslag: Practicum TMI

Andreas Hinderyckx
r0760777

December 2020

1 Beschrijving Algoritmen

1.1 Brute Force

1.1.1 Beschrijving

De werking van het brute-force algoritme is eenvoudig: elke cirkel uit de invoer wordt op snijpunten gecontroleerd met alle cirkels die nog volgen uit de invoer. Dit leidt tot een tijdscomplexiteit

1.1.2 Pseudo-Code

Algorithm 1: Brute Force-algoritme

```
Input: Lijst L met middelpunten en stralen van cirkels
Result: Lijst S van alle snijpunten tussen alle cirkels in L
S ← ∅
foreach Cirkel C_i in L do
    foreach Circkel C_j in L_>i do
        if intersection(C_i, C_j) != ∅ then
            Voeg snijpunt(en) van C_i en C_j toe aan S
        end
    end
end
```

Waarbij de notatie $\texttt{L}_{>i}$ gebruiken om de cirkels aan te duiden uit de lijst `L` met een index groter dan `i`.

1.1.3 Tijdscomplexiteit

Aanpak analyse tijdscomplexiteit

We bespreken de tijdscomplexiteit in functie van het aantal cirkels uit de invoer, nl.: N. Daar waar in complexiteitsanalyse van sorteeralgoritmes bijvoorbeeld het aantal `compare`-operaties tussen twee elementen uit de invoer wordt gebruikt als maatstaf voor de uitvoeringstijd, zullen we hier gebruikmaken van het aantal `intersect`-operaties. Een `intersect`-operatie krijgt als invoer twee cirkels en berekent het aantal snijpunten tussen deze twee cirkels. De verantwoording achter de `intersect`-operatie als maatstaf voor de uitvoeringskost te kiezen, is dat deze methode het 'duurste' deel is van alledrie de algoritmes. De rest van de algoritmes bestaat uit het opbouwen en doorlopen van gegevensstructuren, wat constant is in kost onafgezien van de grootte van de invoer N.

1.1.4 Tijdscomplexiteit Brute Force-algoritme

Aangezien voor elke cirkel die behandeld wordt, zal vergeleken worden met alle cirkels uit de invoer die op deze huidige cirkel volgen, kunnen we het aantal `intersect`-operaties als volgt noteren:

$$\begin{aligned} C &= N + (N-1) + (N-2) + \ldots + 2 + 1 \\ &= \frac{N(N+1)}{2} \\ &\sim \mathcal{O}(N^2) \end{aligned}$$

Waarbij we C gebruiken om de totale kost uit te drukken en gebruikmaakten van de somformule van Gauss: $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$

1.2 Naïeve sweepline

1.2.1 Beschrijving

Het idee achter deze naïeve implementatie van een sweepline-algoritme voor het detecteren van snijpunten, is dat we het aantal `intersection`-oproepen proberen te beperken. Door gebruik te maken van het idee van een (symbolische) sweepline waarbij we de x-as voorstellen als tijds-as, kunnen we ervoor zorgen dat we niet alle cirkels met elkaar moeten vergelijken, maar enkel degene die op een bepaald tijdstip 'actief' zijn. Hierbij is een cirkel C 'actief' enkel en alleen indien de sweepline zich op een tijdstip bevindt ná dat de sweepline het meest linkse punt van C is gepasseerd, en vóór dat de sweepline het meest rechtse punt van C gepasseerd is. Indien we dit idee implementeren, kunnen we een groot aantal `intersect`-operaties met cirkels die niet in elkaars buurt liggen vermijden.

1.2.2 Pseudo-code

Algorithm 2: Naïef sweepline-algoritme

```
Input: Lijst L met middelpunten en stralen van cirkels
Result: Lijst S van alle snijpunten tussen alle cirkels in L
EventPoints ← Gesorteerde lijst EventPoints met start- en
  eindpunten van alle cirkels
S ← ∅
Lijst Actief ← ∅
foreach Punt P in EventPoints do
    if P is het begin van een cirkel then
        foreach Cirkel C in Actief do
            if intersection(C, P.Cirkel) != ∅ then
                Voeg snijpunt(en) van C_i en C_j toe aan S
            end
        end
        Voeg C toe aan Actief
    else
        Verwijder C uit Actief
    end
end
```

Waarbij we met `P.Cirkel` de cirkel bedoelen waarvan het punt `P` het start- of eindpunt is.

1.2.3 Tijdscomplexiteit

Een algemene uitdrukking geven voor de tijdscomplexiteit is niet mogelijk voor dit algoritme, aangezien dit sterk afhankelijk is van de ligging van de cirkels. Omwille van deze reden, delen we de analyse op in drie scenario's:

1. **Best case: Disjuncte x-intervallen**
   Indien alle cirkels zodanig gepositioneerd zijn dat ze zich allemaal uitstrekken over een interval van x-coördinaten waarbinnen zich geen enkele andere cirkel bevindt, zal het naïeve sweepline algoritme de beste performantie vertonen. In dit geval zal de `Actief`-lijst namelijk gelijk blijven aan de lege lijst $\emptyset$ doorheen de ganse uitvoering van het algoritme, waardoor er geen enkele `intersection`-oproep gedaan moet worden. Dit levert een tijdscomplexiteit op die lineair evolueert in het

aantal cirkels N: $\sim \mathcal{O}(N)$

2. **Worst case: gedegenereerd scenario**
   Noem de grootste cirkel van de invoer `C` en stel dat deze een straal `r` heeft en een middelpunt met x-coördinaat `x` heeft. Het gedegenereerde scenario waarin alle andere cirkels uit de invoer zich binnen het interval $I = [\texttt{x-r}, \texttt{x+r}]$ bevinden, is het worst-case scenario voor het naïeve sweepline-algoritme.

   Dit scenario impliceert namelijk dat alle cirkels zich gedurende een bepaalde periode - wanneer de sweepline zich binnen interval I bevindt - tegelijk in de `Actief`-lijst zullen bevinden. Hierdoor zal het naïeve sweepline-algoritme reduceren tot het Brute Force-algoritme, aangezien álle cirkels aanwezig zijn in de `Actief`-lijst en bijgevolg moeten gecontroleerd worden op snijpunten met alle andere cirkels uit de invoer. Dit levert analoog een tijdscomplexiteit op die $\sim \mathcal{O}(N^2)$ is.

3. **Algemeen geval**
   Zoals eerder vermeld, is een algemene uitdrukking geven voor de tijdscomplexiteit van dit algoritme niet mogelijk. Indien we voor de $x-$ en $y-$ coördinaten van de middelpunten van de cirkels uitgaan van een uniforme kansverdeling over $\mathbb{R}^2$, kunnen we stellen dat het algoritme zich asymptotisch 'beter' dan $\sim \mathcal{O}(N^2)$ zal gedragen, aangezien de kans dat de spreiding van de cirkels zich in een van de twee vorige gevallen bevindt, verwaarloosbaar klein is.

1.3 Efficiënt sweepline-algoritme

1.3.1 Beschrijving

De werkwijze van het vorige algoritme kan nog verder verfijnd worden. In de vorige implementatie beschouwen we namelijk enkel de x-coordinaten om het aantal `intersection`-oproepen te beperken en filteren we niet op y-coördinaten. Om dit te kunnen implementeren, moeten we een totale orde op de y-coördinaten van de cirkels definiëren. Aangezien een cirkel echter altijd twee snijpunten heeft met de sweepline [1], is het niet voor de hand liggend om deze orde op volledige cirekels te definiëren.

[1] Behalve op de linker- en rechtereindpunten van de cirkel, waarbij deze twee snijpunten samenvallen

Om dit probleem op te lossen, splitsen we elke cirkel op in een bovenste en onderste halfcirkel, en nemen we als y-coördinaat voor een halfcirkel de y-coördinaat van het resp. hoogste en laagste punt op de cirkel. Op deze manier kunnen we een totale orde op de halfcirkels definiëren wanneer te sweepline zich op x-coördinaat a bevindt. Deze orde noteren we als: $<_a$. Door het probleem op deze manier aan te passen, reduceert het probleem zich tot het vinden van snijpunten in een verzameling rechten, waarvoor we het geziene sweepline-algoritme kunnen gebruiken. Dit resulteert dan ook in zeer gelijkaardige code als deze die gezien is in de cursus, zoals is weergegeven in volgende paragraaf.

1.3.2 Pseudo-code

Algorithm 3: Sweepline-algoritme

```
Input: Lijst L met middelpunten en stralen van cirkels
Result: Lijst S van alle snijpunten tussen alle cirkels in L
EventPoints ← Gesorteerde lijst EventPoints met start- en
  eindpunten van alle cirkels
S ← ∅
Rood-Zwart-boom Actief ← ∅
HalfCirkel boven, onder, huidig
foreach Punt P in EventPoints do
    if Eventpunt P is een startpunt then
        // Geval bovenste halfcirkel
        huidig = bovenste halfcirkel van de cirkel die P omvat
        Voeg huidig toe aan Actief
        Voeg snijpunten van halfcirkel huidig met Boven(Actief,
          huidig) toe aan S
        Voeg snijpunten van halfcirkel huidig met Onder(Actief,
          huidig) toe aan S
        // Geval van onderste halfcirkel
        huidig = onderste halfcrikel van cirkel die P omvat
        Voeg huidig toe aan Actief
        Voeg snijpunten van halfcirkel huidig met
          Boven(Actief,huidig) to aan S
        Voeg snijpunten van halfcirkel huidig met Onder(Actief,
          huidig) toe aan S
    else
        // Eventpunt P is een eindpunt
        // Geval bovenste halfcirkel
        huidig = bovenste halfcirkel van de cirkel die P omvat
        Voeg snijpunten van Boven(Actief, huidig) en
          Onder(Actief, huidig) aan S toe
        Verwijder huidig uit Actief
        // Geval onderste halfcirkel
        huidig = onderste halfcirkel van de cirkel die P omvat
        Voeg snijpunten van Boven(Actief, huidig) en
          Onder(Actief, huidig) aan S toe
        Verwijder huidig uit Actief
    end
end
```

waarbij de methodes `Boven(T,c)` en `Onder(T,c)` de methodes zoals in de cursus zijn die respectievelijk de buur boven `c` in data-structuur `T` en de buur onder `c` in data-structuur `T` teruggeven.

Om de actieve half-cirkels bij te houden, moeten we gebruik maken van een data-structuur die *insert*- en *delete*-operaties in logaritmische tijd kan uitvoeren, om binnen de perken van de gevraagde tijdscomplexiteit te blijven. Hiervoor maken we gebruik van een Rood-Zwart-boom: een binaire boomstructuur die zichzelf balanceert en waarvan we bijgvolg kunnen afleiden dat ze deze operaties in de gewenste logaritmische tijd kan uitvoeren. In *Java* wordt deze structuur geïmplementeerd door de `TreeMap`-class, waarvan we in de implementatie ook gebruikmaken.

1.4 Tijdscomplexiteit

We analyseren het algoritme stap voor stap om de tijdscomplexiteit ervan te bepalen en veronderstellen een input van N cirkels.

- We starten bij lijn 1: hier wordt de lijst van eindpunten van de cirkels gesorteerd, met behulp van mergesort weten we dat we dit in $\mathcal{O}(\log N)$ tijd kunnen realiseren.

- Op lijn 2 start de `for`-lus over alle eindupnten van de cirkels: dit zijn er $2N$. Bijgevolg zal deze lus hoogstens $2N$ keer uitgevoerd worden. In deze `for`-lus worden drie soorten operaties uitgevoerd:

  - Ten eerste `insert-, remove-, Boven-` en `Onder`-operaties op de Rood-Zwart-boom `Actief`. Hiervan weten we dat ze alledrie in $\mathcal{O}(\log N)$ tijd kunnen worden uitgevoerd.
  - Ten tweede: `intersect`-oproepen. De uitvoeringstijd hiervan is onafhankelijk van het aantal cirkels N, aangezien ze telkens slechts op twee cirkels wordt uitgevoerd, i.e. $\mathcal{O}(1)$
  - Ten slotte: operaties die de bovenste of onderste halfcirkel genereren die waartoe `huidig` behoort. Dit kan in constante tijd gebeuren, aangezien voor elk punt uit `EventPoints` een verwijzing naar zijn 'parent'-cirkel wordt bijgehouden: $\mathcal{O}(1)$.

We stellen vast dat er één sorteeroperatie plaatsgrijpt: $\mathcal{O}(N \log N)$, en dat alle operaties binnen de `for`-lus maximum $\mathcal{O}(\log N)$ zijn en deze hoogstens

$2N$ keer worden uitgevoerd. Op elk gegeven moment kunnen er zich maximaal N cirkels in `Actief` bevinden.

De `for`-loop die start op lijn 2 omvat twee soorten instructies:

1. Enerzijds instructies die enkel moeten uitgevoerd worden indien er die iteratie snijpunten gevonden worden tussen `huidig` en `Boven(Actief, huidig)` of `Onder(Actief, huidig)`, nl.: het toevoegen van deze snijpunten aan `Actief`, en

2. Anderzijds instructies die elke iteratie worden uitgevoerd, ongeacht of er al dan niet snijpunten gevonden worden.

We analyseren beide gevallen apart en vatten samen als volgt:

1. Van de instructies omschreven in bovenstaand puntje 1., weten we dat deze - zoals we omschreven in het begin van deze paragraaf - zich $\mathcal{O}(\log N)$ gedragen.

2. Van de instructies omschreven in puntje 2., weten we dat ook deze zich $\mathcal{O}(\log N)$ gedragen zoals voordien besproken. Deze instructies worden slechts voor elk snijpunt uitgevoerd. Stel - zonder verlies van algemeenheid - dat er bij invoer van N cirkels, in het totaal S snijpunten zijn.

Indien we beide gevallen samennemen, stellen we vast dat in het totaal hoogstens $2N+S$ keer een instrucie wordt uitgevoerd die zich hoogstens gedraagd als $\mathcal{O}(\log N)$. Met andere woorden, de totale tijdscomplexiteit van het efficiënte sweep line-algoritme reduceert tot:

$$\begin{aligned} & \mathcal{O}((2N+S)\log N) \\ \equiv\ & \mathcal{O}((N+S)\log N) \end{aligned}$$

Vermits we bij $\mathcal{O}$-notatie constante factoren mogen verwaarlozen.

2 Experimenten

2.1 Maatstaf Complexiteit

Zoals eerder vermeld, hebben we in dit onderzoek de keuze gemaakt om de complexiteit uit te drukken in functie van het aantal `intersect`-oproepen, i.p.v. de ruwe rekentijd. In dit onderdeel zullen we hiervoor een argumentatie en motivatie geven.

Een eerste zaak die we moeten aantonen is dat in het algemene geval de rekentijd grotendeels bepaald wordt door het aantal `intersect`-oproepen dat gedaan wordt bij de uitvoering van een algoritme. Theoretisch gezien stamt dit idee van het feit dat de enige twee aspecten van de drie algoritmes die in uitvoeringstijd zullen toenemen naargelang de invoer toeneemt, de volgende zijn:

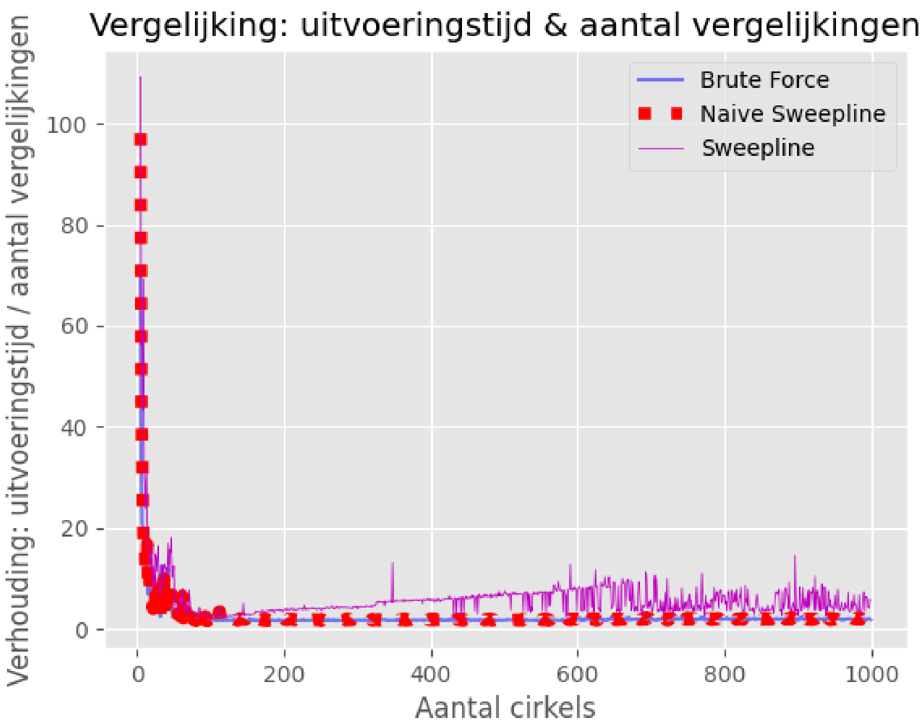
- De zoekoperaties in de Rood-Zwart-Boom (enkel van toepassing voor het finale sweep line-algoritme) enerzijds, en
- Het aantal berekeningen op snijpunten tussen twee cirkels (`intersect`-oproepen) anderzijds.

Nu maken we de veronderstelling dat voor grote inputs. van deze twee factoren, het aantal berekeningen op snijpunten het meeste zal doorwegen in de totale benodigde uitvoeringstijd. Immers, bij kleine inputs zal de tijd benodigd om de alle benodigde datastructuren te initialiseren, de tijd van het kleine aantal `intersect`-operaties overschaduwen. Naargelang de input groeit, verwachten dat hiermee ook het aandeel van de aantal `intersect`-operaties in de totale uitvoeringstijd groeit. Om deze veronderstelling te bevestigen, testen we dit experimenteel.

We laten de drie algoritmes lopen op willekeurige verzamelingen cirkels in het $(1,1) \times (1,1)$-vlak. Deze verzamelingen van cirkels laten we toenemen in grootte, gaande van 5 tot en met 1000. Vervolgens plotten we de verhouding

$$\frac{\text{Uitvoeringstijd}}{\text{Aantal } \texttt{intersect}\text{-oproepen}}$$

in functie van de inputgrootte, N. We bekomen de plot weergegeven in figuur 1.



Figuur 1: Verhouding: $\frac{\text{Uitvoeringstijd}}{\text{Aantal } \texttt{intersect}\text{-oproepen}}$

We zien dat inderdaad onze vermoedens bevestigd worden: voor kleine inputs overheerst de tijd vereist om de datastructuren e.d. te initialiseren, maar naarmate de input groeit, wordt het aantal `intersect`-oproepen dominant in de uitvoeringstijd. Bijgevolg zal de berekende verhouding steeds krimpen en 1 benaderen.

Met deze argumentatie op zak kunnen we onze volgende experimenten verderzetten in functie van het aantal compares, in het achterhoofd houdende dat dit asymptotisch evenredig is met de totale uitvoeringstijd.

2.2 Opbouw

Om een diverse opbouw van experimenten op te bouwen, delen we ze op in volgende deel-experimenten die elks een verschillend aspect van de algoritmes proberen toe te lichten:

- **Positionering Cirkels**
  Hier testen we uit hoe verschillende onderlinge liggingen van cirkels ten opzichte van elkaar invloed hebben op de efficiëntie van de algoritmen, zoals we ook hebben besproken in de vorige respectievelijke secties over tijdscomplexiteit.

- **Aantal Cirkels**
  In dit eerste experiment analyseren we eenvoudigweg wat het aantal `intersect`-oproepen is in functie van het aantal cirkels dat als input gegeven wordt.

2.3 Positionering Cirkels

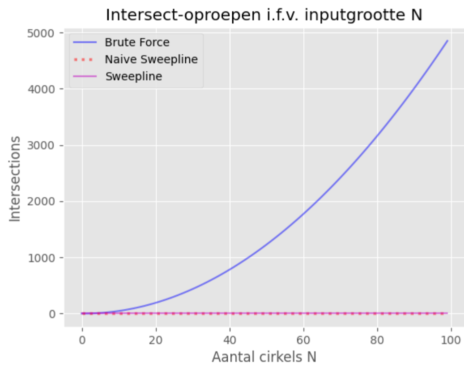
We bekijken enkel de effecten van specifieke positionering van de invoercirkels op de verschillende algoritmes.

2.3.1 Cirkels in disjuncte x-intervallen

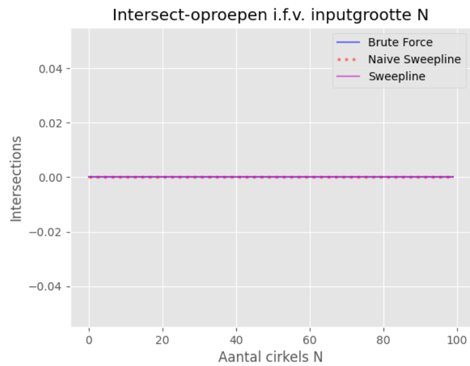
In dit onderdeel gaan we na wat de efficiëntie van de algoritmes is op een invoer waarbij alle cirkels gelegen zijn in disjuncte x-intervallen. Hiermee bedoelen we dat voor elke cirkel C_1 geldt dat geen enkele andere cirkel C_2 start of eindigt binnen de x-coördinaten waarover C_1 zich uitstrekt.

2.3.1.1 Naïeve Intersection-methode

Eerst kijken we wat het effect van het gebruik van een naïeve implementatie van de `intersect`-methode is, naïef in de zin dat we geen gebruik maken van quick-rejection tests en we dus bij elke oproep van de methode de volledige methode en bijbehorende berekeningen uitvoeren. We plotten het aantal `intersect`-oproepen in functie van de inputgrootte N en bekomen de plot uit figuur 2.



Figuur 2: Aantal `intersect` -oproepen i.f.v. inputgrootte N



Figuur 3: Aantal `intersect`-oproepen i.f.v. inputgrootte N

Deze resultaten stroken onze verwachtingen: aangezien het brute-force algoritme elk paar van cirkels onderling test op snijpunten, verwachten we een aantal `intersect`-oproepen dat nog steeds kwadratisch zal toenemen: de uitvoering van het brute-force algoritme is namelijk onafhankelijk van de ligging van de cirkels.

Beide sweep line-algoritmes zullen echter geen enkele `intersection`-oproep maken, aangezien de x-intervallen waarbinnen de cirkels liggen onderling disjunct zijn, en bijgevolg elke cirkel individueel steeds als enige element aanwezig zal zijn in de `Actief`-datastructuur. Deze argumentatie komt overeen met de resultaten uit figuur 2.

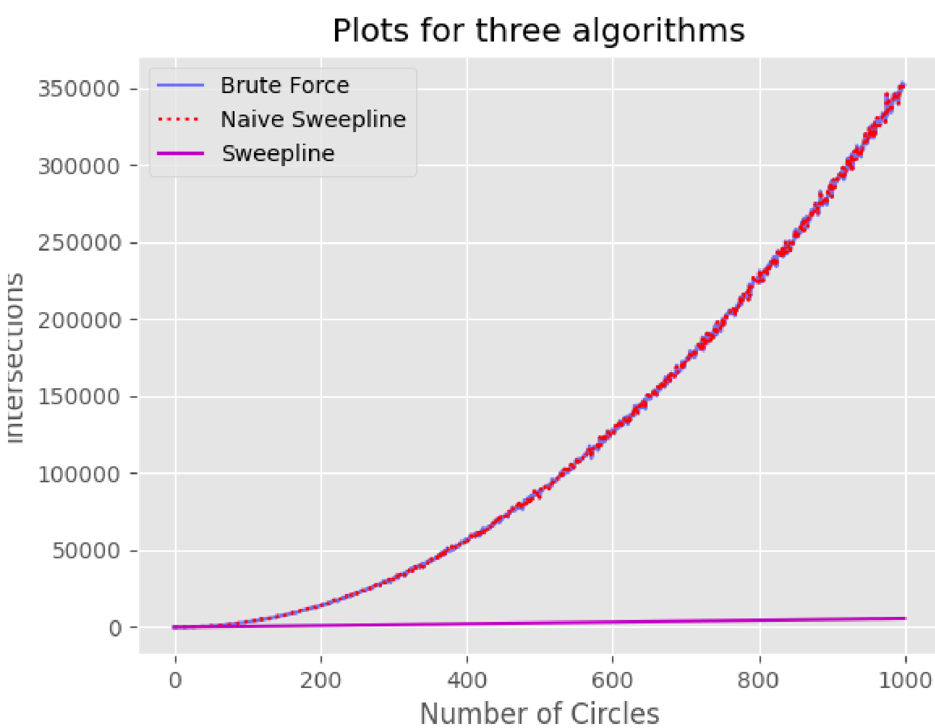
2.3.1.2 Efficiëntere Intersection-methode

In deze efficiëntere implementatie van de `intersection`-methode maken we gebruik van een quick-rejection test om paren van cirkels die onmogelijk kunnen snijden, meteen te verwerpen zonder dat we hiervoor alle berekeningen moeten uitvoeren. Deze quick-rejection test verwerpt de mogelijkheid dat twee cirkels snijden indien de afstand tussen hun middelpunten d kleiner dan het verschil van hun stralen, of groter dan de som van hun stralen is. Indien we nu de resultaten plotten zoals we bij figuur 2 deden, bekomen we de plot die weergegeven is in figuur 3.

Nu is ook het aantal `intersection`-oproepen van het brute force-algoritme constant en gelijk aan 0. Dit valt te verklaren doordat de quick-rejection test bij deze specifieke invoer alle paren van cirkels direct kan verwerpen op mogelijke snijpunten, doordat elke cirkel zich bevindt in een geïsloeerd x-interval. Deze versie van de `intersect`-methode is dan ook degene die we zullen gebruiken doorheen de rest van de experimenten, alsook de versie die in de beschrijving van de werking van de algoritmes is gebruikt.

2.4 Aantal Cirkels

In dit eerste experiment testen we de drie algoritmes op eenzelfde input, die toeneemt tot een aantal van zo'n 1000 input-cirkels waarvan de snijpunten moeten gevonden worden. We plotten de bekomen data en bekomen het resultaat weergegeven in figuur 4.



Figuur 4: Aantal `intersection`-oproepen i.f.v. invoergrootte N

Steunend op de argumentatie van de vorige paragraaf, kunnen we deze resultaten interpreteren als volgt. De naïeve implementatie van de sweepline en het brute force-algoritme gedragen zich asymptotisch gezien gelijkaardig.

We bepalen de groei-orde van het aantal `intersect`-oproepen met behulp van het doubling ratio experiment. Noem $T(N)$ het aantal `intersect`-oproepen voor een invoergrootte van N. Uit onze data, of door af te lezen op de plot, weten we dat:

$$T(1000) \approx 350000 \quad \text{en}$$
$$T(500) \approx 87000$$

Hieruit volgt:

$$\frac{T(2N)}{T(N)} = \frac{T(1000)}{T(500)} \approx \frac{350000}{87000} \approx 4$$

Bij een verdubbeling van de invoergrootte, stijgt het aantal `intersect`-operaties dus met een factor $4 = 2^b$. Aangezien volgens het Doubling Ratio Experiment de groei-orde bij benadering wordt gegeven door N^b en $4 = 2^2$, besluiten we dat zowel het brute-force als naïef sweepline-algoritme een kwardratsiche tijdscomplexiteit hebben:, of met andere woorden: ze gedragen zich $\mathcal{O}(N^2)$ met N de inputgrootte. Dit bevestigt onze argumentatie die we gemaakt hebben in secties 1.1.4 en 1.2.3.

Het verschil in groei van grootte-orde tussen de naïeve en efficiënte implementatie van het sweep line-algoritme kunnen we verklaren doordat het naïeve sweep line-algoritme essentieel enkel filtert op de x-intervallen van de cirkels om `intersect`-oproepen tussen cirkels die onmogelijk kunenn snijden te vermijden. De efficiënte implementatie daarentegen, maakt ook gebruik van de totale orde relatie $>_a$, wat het mogelijk maakt om op elke gegeven x-coördinaat hoogstens twee `intersect`-oproepen te moeten maken: een voor `Boven(Actief, huidig)` en een voor `Onder(Actief, huidig)`.

Aangezien alle cirkels binnen het interval $[0, 1]$ werden gegenereerd, is er een zeer grote graad van overlap, wat betreft x-coördinaten van cirkels. Bijgevolg zal het naïeve sweep line-algoritme slechts enkele `intersection`-oproepen kunnen vermijden, terwijl de efficiënte versie het grootste deel van de oproepen kan uitsluiten door ook op y-coördinaat efficiënt de `intersection`-oproepen te selecteren.

3 Randgevallen

We omschrijven ten slotte voor de verschillende algoritmes welke randgevallen incorrecte resultaten kunnen opleveren.

3.1 Brute Force Algoritme

3.2 Naïef Sweep Line-algoritme

3.3 Sweep Line-algoritme

3.3.1 Drievoudig Snijpunt

Indien de cirkels zodanig geplaatst zijn dat er een drievoudig snijpunt optreedt, zal het efficiënte sweep line-algoritme overbodige oplossingen voor snijpunten geven. Het zal het drievoudig snijpunt drie keer vinden, in combinatie met de overige snijpunten, wat correct is, maar het zal elk individueel snijpunt nog een extra keer als snijpunt classificeren. Een mogelijke oplossing hiervoor zou zijn