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

Input: Lijst `L` met middelpunten en stralen van cirkels
Result: Lijst `S` van alle snijpunten tussen alle cirkels in `L`
$\texttt{S} \leftarrow \emptyset$
foreach *Cirkel* C_i *in* `L` **do**
  foreach *Circkel* C_j *in* $\texttt{L}_{>i}$ **do**
    if *intersection*(C_i, C_j) $!= \emptyset$ **then**
      Voeg snijpunt(en) van C_i en C_j toe aan `S`
    end
  end
end

Algorithm 1: Brute Force-algoritme

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

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

Algorithm 2: Naïef sweepline-algoritme

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 ∅ 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.

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

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

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

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

Algorithm 3: Sweepline-algoritme

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