KU LEUVEN

FACULTEIT WETENSCHAPPEN

Een nieuwe constructieve heuristic voor het plaatsen van cirkels in een cirkel

Gebaseerd op een *best-fit* methodiek

Pablo BOLLANSÉE

Promotor: Prof. P. De Causmaecker
Affiliatie *(facultatief)*

Co-promotor: *(facultatief)*
Affiliatie *(facultatief)*

Begeleider: *(facultatief)*
Affiliatie *(facultatief)*

Proefschrift ingediend tot het behalen van de graad van Master of Science in de toegepaste informatica

Academiejaar 2015-2016

Voorwoord

Het circle-packing probleem bestaat er uit om een aantal cirkels, met gekende radii, in een zo klein mogelijke container te plaatsen. De vorm van deze container kan verschillen, meestal is het een driehoek, rechthoek of cirkel. In deze thesis stel ik een nieuwe *best-fit* gebaseerde heuristiek voor voor het plaatsen van cirkels in een cirkel. Het is een constructieve heuristiek waarin stapsgewijs telkens de best-passende cirkel geplaatst zal worden. Alle code is beschikbaar op GitHub [3]

Wiskundig is dit een relatief eenvoudig probleem om voor te stellen, maar computationeel is het zeer zwaar om exact op te lossen. Bestaande pogingen om dit probleem op te lossen vragen zeer veel tijd om het te berekenen. In deze thesis stel ik een nieuwe heuristiek voor die het mogelijk maakt zéér snel oplossingen te genereren.

Ik wil hierbij Patrick De Causmaeker bedanken voor alle hulp en ondersteuning bij het ontwerpen van deze heuristiek en verwezenlijken van dit werk. Ook wil ik Jim Bollansée en Marie Julia Bollansée bedanken voor hun hulp bij het schrijven van deze tekst.

Abstract

TODO

Inhoud

Lijst van figuren

Lijst van tabellen

Hoofdstuk 1

Inleiding

In deze thesis stel ik een nieuwe *best-fit* gebaseerde heuristiek voor voor het cirkle-packing probleem. De heuristiek is specifiek voor sub-probleem van het plaatsen van cirkels in een cirkel. Ook bespreek ik de implementatie die gemaakt is naast de ontwikkeling van de heuristiek. Dit circle-packing probleem bestaat uit het plaatsen van n cirkels in een zo klein mogelijke cirkelvormige container. Het is de bedoeling om voor de gegeven cirkels de coordinaten van de middelpunten te vinden zodat deze niet overlappen en de radius van de omcirkel zo klein mogelijk is.

Circle-Packing is zowel theoretisch als practisch een zeer interessant probleem. Het kan gebruikt worden om verschillende real-world problemen op te lossen, zoals het plaatsen van zendmasten, stokage van cilindrische voorwerpen, en het combineren van verschillende kabels.

Mathematisch is het redelijk eenvoudig als een optimalisatie probleem te omschrijven:

$$\begin{array}{lll} \text{minimaliseer} & r & \\ \text{onderhevig aan} & x_i^2 + y_i^2 \leq (r - r_i)^2, & i = 1, ..., n \\ & (x_i - x_j)^2 + (y_i + y_j)^2 \geq (r_i + r_j)^2, & i \neq j \end{array}$$

Hierin is r_i de radius, en (x_i, y_i) de coördinaten van het centrum van cirkel i. Hierbij wordt verondersteld dat de omcirkel het nulpunt als middelpunt heeft. De eerste formule verzekerd dat de cirkels in de omcirkel liggen, en de tweede dat ze elkaar niet overlappen. In het geval dat alle cirkels de zelfde grootte hebben wordt meestal r_i altijd gelijk aan 1 genomen. Het circle-packing probleem voor andere containers heeft gelijkaardige, relatief eenvoudige, wiskundige omschrijvingen.

Hoewel dit wiskundig zeer eenvoudig te omschrijven is, blijft het toch een zeer moeilijk probleem om exact op te lossen. Het is een NP-hard probleem. Daarom zoekt men naar andere technieken om het toch op te kunnen lossen. Er is reeds veel onderzoek gebeurt naar het oplossen van het circle-packing probleem voor zowel cirkels van gelijke goottе, als voor cirkels van verschillende grootte. In [6] en [12] probeert men vaste patronen te vinden die een optimale packing van cirkels met gelijke grootte geeft. In [7], [11] en [15] worden fysisch geïnspireerde simulaties gebruikt om packing te bekomen. Ook worden zijn er pogingen om meta-heuristieken te gebruiken om packings te bekomen. In [5] worden verschillende meta-heuristieken, waaronder een genetisch algoritme, uitgeprobeerd en vergeleken. Zij ondervinden dat dit genetisch algoritme en een quasi-random techniek, in vergelijking met de andere uitgeprobeerde meta-heuristieken, de beste resultaten geven. In [9] en [10] worden respectievelijk een genetisch en een simulated-annealing algoritme

voorgesteld. Een recentere poging is het Monotonic Basin Hopping algoritme voorgesteld in [8]. Hierin beschrijven ze dat er teveel lokale optima zijn voor een eenvoudige multi-start behandeling, en stellen een variant voor waarin ze op een slimme manier de begin punten proberen genereren. Ook relatief recent is [2], waarin gebruik wordt gemaakt van de combinatorische eigenschappen van circle-packing in combinatie met een taboo-search en een off-the-shelf non-linear optimizer. Hierin plaatsen ze één voor één elke cirkel en laat de non-linear optimizer hiervoor telkens een lokaal extremum berekenen. Ze zoeken van met de taboo-search naar de beste volgorde om de cirkels te plaatsen.

Constructieve algoritmen voor het oplossen van circle-packings zijn veel minder onderzocht. Eén van de weinig constructieve methoden word beschreven in [1], waar ze een alternatieve vorm van circle-packings oplossen: de grootte van de container ligt vast, en je moet zo veel mogelijk cirkels van gelijke grootte er in plaatsen. In [9] wordt er dan weer een aanpassing gedaan op de klassieke bottom-left-first heuristiek, voor het plaatsen van rechthoeken, om gebruikt te maken van de cirkelvormigheid van de objecten.

Hoewel veel van deze oplossingen zeer goede packings maken, regelmatig verbeteren ze hun voorganger, vragen ze zeer veel reken tijd en beperken ze zich tot een klein aantal cirkels. In deze thesis stel ik een nieuw constructieve heuristiek voor om het circle-packing probleem op te lossen. Deze laat toe in een fractie van de tijd nodig voor andere algoritmen packings te maken. Ook is het mogelijk om veel grotere aantallen cirkels te plaatsen. De packings zijn echter minder dicht dan deze verkregen in voorgenoemd onderzoek. Deze *verslechtering* blijft echter beperkt en laat toe de packings in slecht enkele ogenblikken te maken, waar anderen uren rekentijd vragen.

De heuristiek voorgesteld in deze thesis is een best-fit heuristiek gebaseerd op een oplossing voor het Orthogonal Stock-Cutting Problem voorgesteld in [4]. Zij stellen een heuristiek voor die de volgende balk om te plaatsen kiest uit een lijst, en deze plaatst op de *beste* positie. Dit in tegenstelling tot cirkels plaatsen in een vooraf bepaalde volgorde zoals in [8] en [2]. Op een gelijkaardige manier kiest mijn algoritme de volgende cirkel die best past in de huidige packing.

In hoofdstuk 3 bespreek ik hoe de heuristiek opgebouwd is. Ik bespreek de twee basis concepten voor mijn best-fit heuristiek: *holes* en de *shell*. Ik bespreek hoe deze werken, en op welke manier gekozen wordt welke cirkel best past in de packing. Hierbij bespreek ik ook de implementatie. In hoofdstuk 4 worden de verkregen resultaten besproken. Hier vergelijk ik de packings met de best gekende resultaten zoals gerapporteerd op de Packomania website ([13]). Ik doe hier een vergelijking zowel op omtrek van de verkregen omcirkel, als op nodige tijd om deze packing te berekenen tegenover de best gekende oplossingen. Ook toon ik resultaten voor packings voor veel meer cirkels dan getoond op de Packomania website. In hoofdstuk 5 bespreek ik mogelijke verbeteringen, de *losse eindjes* en ideeën voor verdere uitbreidingen en onderzoek. In hoofdstuk 2 wordt kort verduidelijkt hoe u de visualisaties gebruikt doorheen deze thesis kan interpreteren.

Hoofdstuk 2

Handleiding voor het lezen van de visualisaties

Doorheen deze thesis zal ik gebruik maken visualisaties gegenereerd door de java implementatie van het algoritme. Dit om de concepten grafisch te verduidelijken. Twee voorbeelden van zulke visualisaties zijn figuur 2.1 en figuur 2.2.

Deze figuren kan u op de volgende manier interpreteren:

- De **reeds geplaatste cirkels** worden getoond als **licht blauwe cirkels**.
- De **shell** is een **gele lijn** aan de buitenste rand van de packing. Deze verbind de middelpunten van de cirkels op de shell.
- De kleine **groene bolletjes** op de shell geven de posities aan waarop mogelijk een cirkel geplaatst zal worden.
- **Holes** worden getoond als **rode driehoeken**. De hoekpunten liggen dicht bij de middelpunten van de drie cirkels die het gat definiëren.
- De **omcirkel** van de huidige packing wordt getoond als een **groene cirkel**.

Op figuur 2.1 is zijn er duidelijk drie holes te zien. Elk van de drie holes word gedefinieerd door de centrale cirkel en twee van de buitenste cirkels. Ook is er een kleine shell te zien, die bestaat uit de buitenste drie cirkels. In figuur 2.2 wordt een verder gevorderde packing getoond waarop één hole te zien is, en een veel grotere shell. Op beide figuren kan je ook de omcirkel zien.

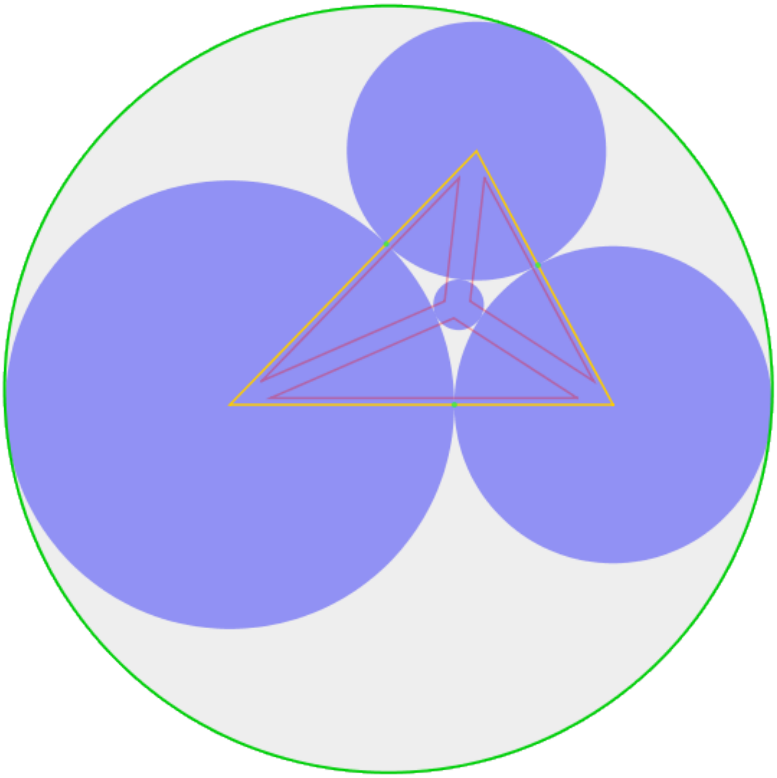

Figure 2.1: Voorbeeld visualisatie met drie duidelijke holes

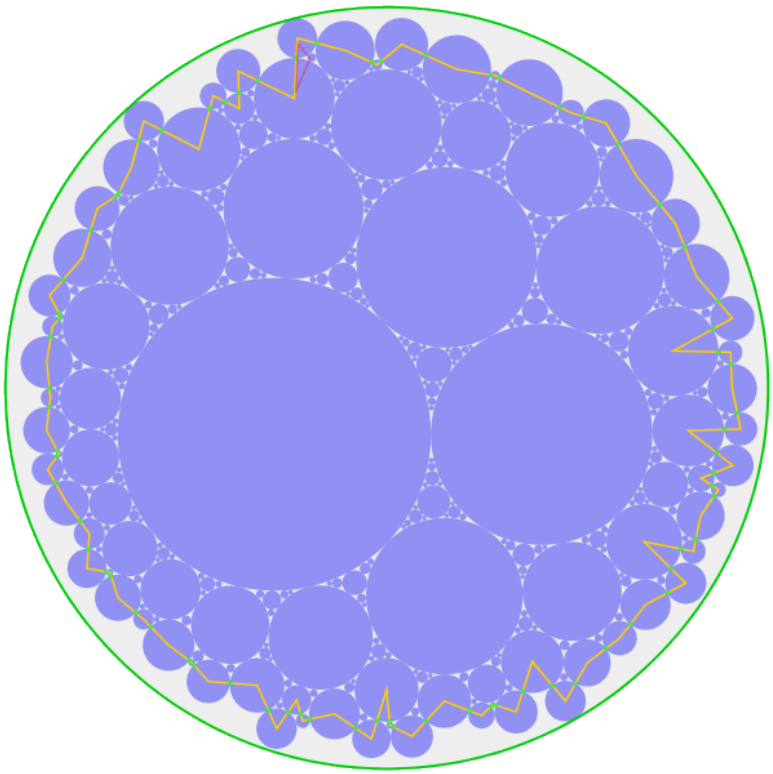

Figure 2.2: Voorbeeld visualisatie met grote shell

Hoofdstuk 3

Algoritme

In dit hoofdstuk bespreek ik de werking van de heuristiek. Eerst geef ik een korte beschrijving van het basis idee van het algoritme, gevolgd door de structuur van de code. Vervolgens leg ik stelselmatig de volledige werking uit, alle veronderstellingen die gemaakt worden en implementatie details waar nodig. De volledige implementatie is beschikbaar op GitHub [3] en is gebeurt in Java.

3.1 Basis idee

Het basis idee van de heuristiek is om stelselmatig een packing op te bouwen, door telkens cirkels te zoeken die het best passen. Bij elke stap wordt telkens eerst een plek gekozen wordt om een cirkel te plaatsen (in een hole, of op de shell, later hier meer over). Hier wordt dan de best-passende cirkel geplaatst. Eenmaal een cirkel geplaatst is wordt deze nooit meer verplaatst. Dit laat toe om intelligente structuren op te bouwen en deze op een zeer efficiënte manier te gebruiken.

Het algoritme bouwt dus cirkel per cirkel een packing op. Dit door in elke stap een positie te kiezen, en hierin een cirkel te proberen plaatsen. Indien er geen cirkel geplaatst kan worden wordt de interne structuur van het probleem vernieuwd om deze nieuwe informatie te reflecteren. Dit gebeurt op verschillende manieren voor de holes en de shell. Meer hierover vindt u terug in sectie 3.6 en sectie 3.7. Als er wel een cirkel geplaatst kan worden dan wordt deze uit de lijst van nog-te-plaatsen cirkels verwijderd, en krijgt deze een permanente positie daar. Dit geeft ook aanleiding tot aanpassen van de holes en/of shell. Hierdoor wordt er een nieuwe tussentijdse packing gemaakt. Deze wordt dan door gegeven naar de volgende stap, waarin het algoritme opnieuw zal proberen een cirkel te plaatsen. Op deze manier word een volledige packing opgebouwd voor alle cirkels.

3.2 Structuur

De implementatie van het algoritme bevat enkele belangrijke classen die regelmatig zullen terug komen in de verdere uit, vooral in code fragmenten:

- Cirkel (Circle)
- Vector2

- Locatie (Location)
- Probleem (Problem)
- Oplossing (Solution)
- Oplosser (Solver)
- Gat (Hole)
- Schil (Shell)

Een *circle* is voor de hand liggend. Deze heeft een radius, maar geen positie.

*Vector*2 is een 2D positie. Deze bevat een x en y coördinaat.

Een *location* of locatie is de combinatie van een cirkel met mijn positie. Deze bevat dus een referentie naar een *circle* en een *vector*2.

Een *problem* of probleem is een lijst van cirkels. Deze hebben nog geen positie, en worden gesorteerd van groot naar klein. Dit is wat de *solver* als input krijgt.

Een *solution* of oplossing is een lijst van cirkels met hun positie. Dit kan een tussen oplossing zijn, waar nog niet voor alle cirkels uit een probleem een positie gevonden is. Ook geeft dit geen garanties van correctheid, er kan dus bijvoorbeeld overlap zijn, maar voorziet functionaliteit om dit na te gaan. Dit is wat de solver als output geeft. Een correcte solver geeft natuurlijk wel altijd goede oplossingen.

Een *solver* of oplosser is het object dat een packing zoekt voor een gegeven probleem. Dit is dus het belangrijkste deel van de code, en hier is de nieuwe heuristiek geïmplementeerd. De best-fit solver, zoals beschreven in deze thesis, doet dit stap voor stap. In elke stap wordt er één cirkel geplaats op zijn finale positie, dit aan de hand van enkele keuzes die verder in dit hoofdstuk toegelicht zullen worden.

Hole en *shell* worden verder uitgelegd in respectievelijk sectie 3.6 en sectie 3.7.

3.3 Structuur van de solver

Zoals hierboven gezegd is de solver het hart van de implementatie. Deze maakt effectief een packing voor een gegeven probleem. De solver bevat een lijst van *holes* en de *shell*. Het bevat ook een lijst van de nog te plaatsen cirkels, en een tussen-oplossing met de cirkels die reeds een plaats gekregen hebben. Ook heeft hij interne omcirkel voor deze oplossing. Een oplossing kan zelf ook een omcirkel berekenen, maar de solver gebruikt een interne omcirkel die enkel vernieuwd wordt als het nodig is. Bovendien heeft de solver extra informatie die de oplossing niet heeft, waardoor deze omcirkel efficiënter berekend kan worden. Zie sectie 3.7 voor meer uitleg hierover.

In de implementatie ziet de code van de solver er als volgt uit (vereenvoudigde versie):

```
1 List<Circle> circlesToPack;
2 Queue<Hole> holes;
3 List<Location> shell;
4 Location enclosingCircle;
5
6 Solution solution;
7
```

```
8  void solve() {
9      init();
10     packFirstThree();
11
12     while(!circlesToPack.isEmpty()) {
13         boolean ok = bestFitStep();
14         if (!ok) break;
15     }
16 }
17
18 boolean bestFitStep() {
19     if (circlesToPack.isEmpty()) {
20         return false;
21     }
22
23     if(!holes.isEmpty()) {
24         ...
25         // Probeer een cirkel in een gat te plaatsen
26         ...
27         return true;
28     }
29     else if (!shell.isEmpty()) {
30         ...
31         // Probeer een cirkel op de shell te plaatsen
32         ...
33         return true;
34     }
35 }
```

De solver bevat alle nodige informatie over de shell en holes, als ook de cirkels die nog geplaatst moeten worden en de huidige tussen-oplossing (lijn 1 tot lijn 6). Om een probleem op te lossen wordt de solve methode (lijn 8) aangeroepen. Deze initialiseert eerst alle nodige variabelen, doet dan de initiële packing (meer hierover in sectie 3.4) en voert dan best-fit-stappen uit tot een oplossing bereikt is (vanaf lijn 12).

De best-fit solver implementatie uit deze thesis kan stap voor stap de oplossing genereren en tussentijdse oplossingen visualiseren. Het is dus niet nodig een packing volledig te maken, het kan zeer nuttig zijn tussentijdse oplossingen te zien, zeker bij het debuggen of implementeren van nieuwe functionaliteit.

3.4 Initialisatie

Zoals eerder gezegd bouwt het algoritme steeds verder op een packing uit de vorige stap. Hierdoor is het dus nodig om een initiële packing te maken van een aantal cirkels waarop de volgende stappen kunnen verder bouwen. Deze initiële packing is de optimale packing van de drie grootste cirkels in het probleem. Deze drie cirkels worden zo geplaatst dat ze alle drie aan elkaar raken, zoals getoond in figuur 3.1. De licht blauwe cirkels tonen de drie eerste-geplaatste cirkels. Meer uitleg over hoe u deze figuur kan interpreteren kan u vinden in hoofdstuk 2

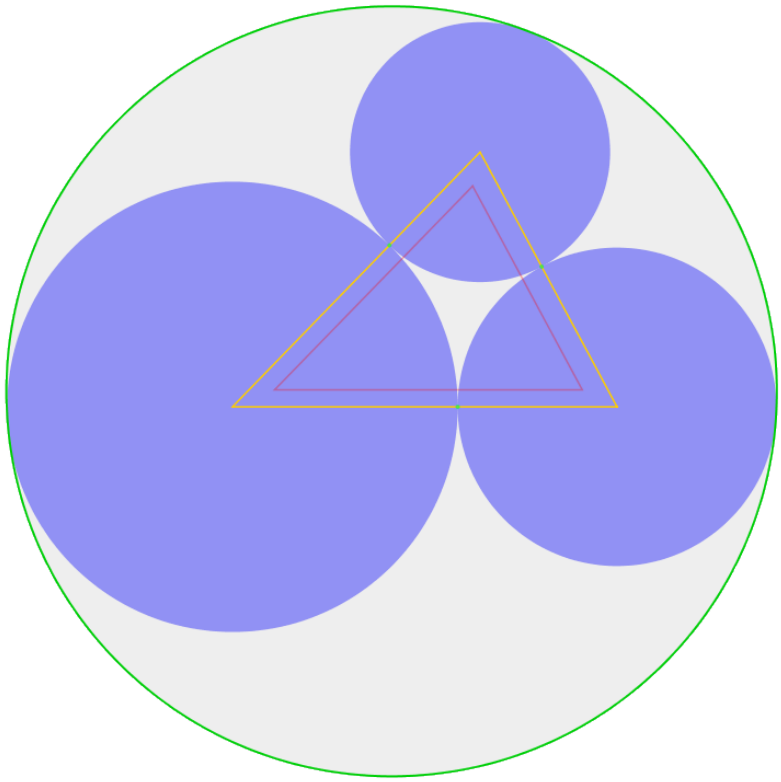

Figure 3.1: Voorbeeld van initiële packing

Het exacte proces om deze initiële packing te bekomen wordt verduidelijkt aan de hand van code uit de implementatie:

```
1  private void packFirstThree() {
2      //Initially place the two biggest circles next to eachother
3      Circle first = circlesToPack.get(0);
4      Circle second = circlesToPack.get(1);
5
6      Vector2 firstPos = new Vector2(0, 0);
7      Vector2 secondPos = new Vector2(first.getRadius() + second.getRadius(),
           0);
8
9      Location firstLoc = new Location(firstPos, first);
10     Location secondLoc = new Location(secondPos, second);
11     getSolution().add(firstLoc);
12     getSolution().add(secondLoc);
13
14     // Place the third biggest circle on top of the first two (assuming they
           are positioned clockwise)
15     Circle third = circlesToPack.get(2);
16     Vector2 thirdPos = Helpers.getMountPositionFor(third, firstLoc,
           secondLoc);
17     Location thirdLoc = new Location(thirdPos, third);
18     getSolution().add(thirdLoc);
19
```

```
20      circlesToPack.remove(first);
21      circlesToPack.remove(second);
22      circlesToPack.remove(third);
23
24      // Create first hole
25      holes.add(new NHole(firstLoc, secondLoc, thirdLoc));
26      // Create the initial shell
27      // IMPORTANT: must be clock-wise
28      shell.add(firstLoc);
29      shell.add(thirdLoc);
30      shell.add(secondLoc);
31
32      enclosingCircle =
            Location.calculateEnclosingCircle(Arrays.asList(firstLoc, secondLoc,
            thirdLoc));
33  }
```

Eerst worden de twee grootste cirkels naast elkaar geplaatst. Vanaf lijn 3 tot lijn 7 worden eerst de twee grootste cirkels uit het probleem opgevraagd. De lijst *circlesToPack* is gesorteerd van groot naar klein, dus dit zijn de eerste twee cirkels in deze lijst. De eerste wordt in de oorsprong geplaatst, en de tweede er tegen op de horizontale as. Deze worden ook reeds aan de tussentijdse oplossing toegevoegd (vanaf lijn 11). Vervolgend wordt de positie berekend voor de derde aan de hand van een helper functie op lijn 16. Deze helper functie komt regelmatig terug, en wordt verduidelijkt in sectie 3.5.

In de initialisatie wordt ook het eerste gat gemaakt, gedefinieerd door de eerste drie cirkels. Dit gat wordt toegevoegd aan de lijst van gaten in de solver op lijn 25. Ook wordt de shell aangemaakt, vanaf lijn 28. Deze wordt met de klok mee (gezien vanuit het centrum van de huidige packing) bij gehouden. Verdere uitleg hierover is te vinden in sectie 3.7.

3.5 Een cirkel tegen twee andere plaatsen

In verschillende delen van de heuristiek is het nodig om een cirkel c_i, met radius r_i, tegen twee andere cirkels te plaatsen. Deze twee cirkels noemen we c_g1, c_g2, en hun radii r_g1, r_g2. Het punt vinden waarop deze cirkel moet staan om beide andere cirkels te raken wordt bepaald door een eenvoudige cirkel-cirkel intersectie, tussen twee cirkels met hun middelpunt gelijk aan het middelpunt van de cirkels c_g1 en c_g2 en als radii $r_g1 + r_i$ en $r_g2 + r_i$:

```
1   Vector2 getMountPositionFor(Circle cir, Location first, Location second) {
2       double x0 = first.getPosition().getX();
3       double y0 = first.getPosition().getY();
4       double r0 = first.getCircle().getRadius() + cir.getRadius();
5
6       double x1 = second.getPosition().getX();
7       double y1 = second.getPosition().getY();
8       double r1 = second.getCircle().getRadius() + cir.getRadius();
9
```

```
10      // dx en dy zijn de verticale en horizontale afstand tussen de
            cirkel-centra.
11      double dx = x1 - x0;
12      double dy = y1 - y0;
13
14      // Bepaalde de afstand tussen de centra
15      //d = sqrt((dy*dy) + (dx*dx));
16      double d = Math.hypot(dx, dy);
17
18      // 'Punt 2' is het punt waar de lijn door de cirkel-intersectie punten de
            lijn tussen de cirkel-centra kruist
19      // We berekenen hier de coordinaten x2 en y2 van dit punt
20
21      // Bepaal eerst de afstand van tussen Punt 2 en het centrum van de eerste
            cirkel
22      double a = ((r0*r0) - (r1*r1) + (d*d)) / (2.0 * d);
23
24      // Bepaal dan de coordinaten van Punt 2.
25      double x2 = x0 + (dx * a/d);
26      double y2 = y0 + (dy * a/d);
27
28      // Bepaal nu de afstand van Punt 2 naar een van de intersectie-punten
29      // Het tweede intersectie-punt ligt even ver
30      double h = Math.sqrt((r0*r0) - (a*a));
31
32      // Zet deze afstand om naar een vector met de juiste richting
33      double rx = -dy * (h/d);
34      double ry = dx * (h/d);
35
36      // Bepaal een van de tweede intersectie punten
37      return new Vector2(x2 - rx, y2 - ry);
38  }
```

In deze code wordt één van de intersectie punten bepaald. Dit intersectie punt is steeds het negatieve punt. Als op lijn 37 + gebruikt wordt in plaats van − kan het tweede punt bekomen worden. Het is ook mogelijk het andere intersectie punt te verkrijgen door de twee *location* parameters om te wisselen.

Het bekomen punt is steeds het punt dat aan uw linker kant zou liggen indien je wandelt van het centrum van de eerste cirkel naar het centrum van de tweede cirkel. Dit is verduidelijkt in figuur 3.2, de onderste cirkel is de eerste, de bovenste cirkel de tweede. De pijl tussen deze cirkels geeft de *wandel richting* aan.

3.6 Holes

Het eerste van de twee belangrijkste concepten van de heuristiek is *holes* of *gaten*. Dit zijn plaatsen tussen andere, reeds geplaatste, cirkels waar potentieel nog een cirkel tussen kan passen. De heuristiek zal telkens eerst deze gaten proberen op te vullen, alvorens cirkels op de shell te plaatsen.

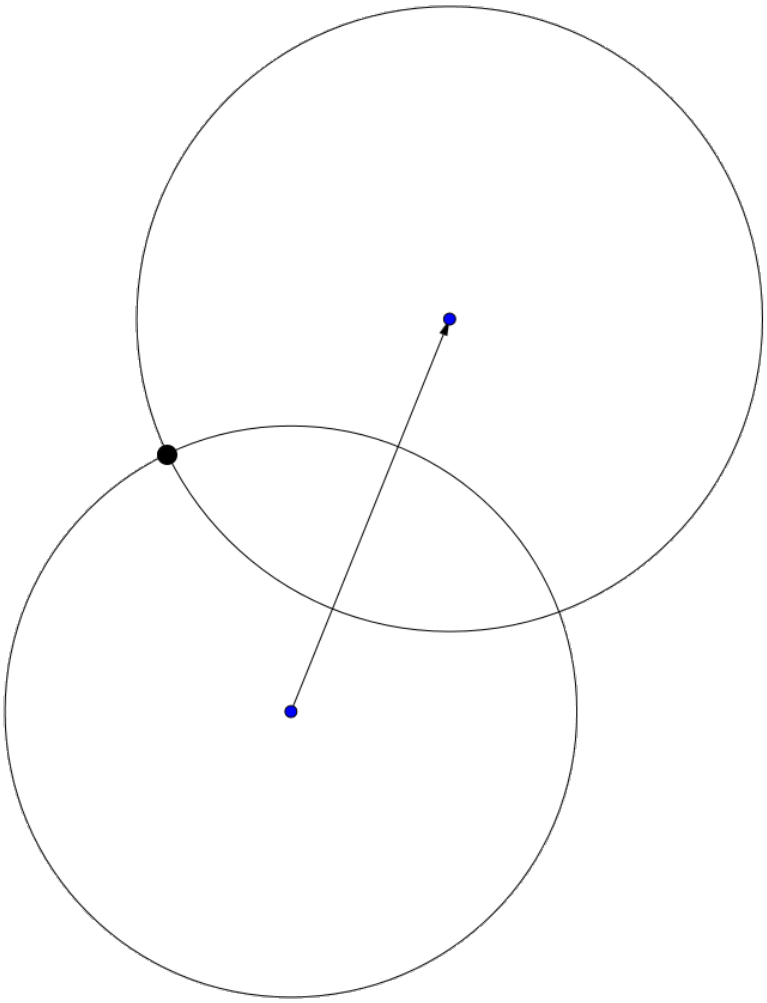

Figure 3.2: Verkregen intersectie punt van *getMountPositionFor*

(Gemaakt met web.geogebra.org)

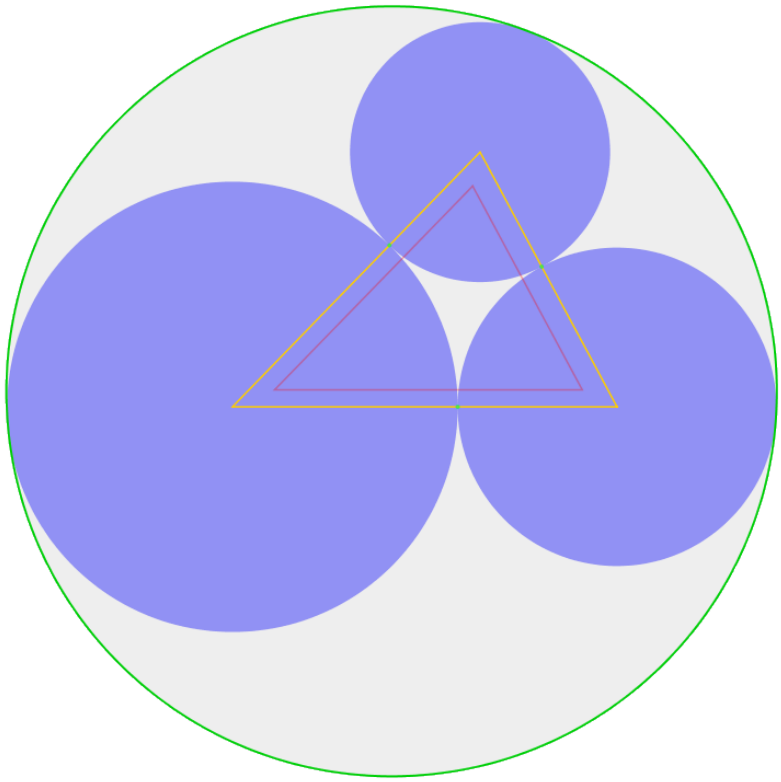

Figure 3.3: Packing voor het opvullen van een gat

Gaten worden gedefinieerd door exact drie cirkels in de huidige packing. De solver houdt informatie bij voor elk gat waar mogelijk nog een cirkel in kan passen. Bij elke stap van de solver zal er eerst gekeken worden of er nog gaten in de oplossing zijn. Indien er nog gaten zijn zal hij deze dus eerst hier een cirkel in proberen plaatsen. Indien het gat te klein is voor alle nog-te-plaatsen cirkels wordt dit gat simpelweg verwijderd uit de lijst van gaten in de solver. Op deze manier weet de solver in de volgende stap dat hij daar niet meer moet proberen om een cirkel te plaatsen, en zal hij een ander gat uitproberen. Indien er wel een cirkel in het gat past wordt deze daar in geplaatst. Dit zal leiden tot het creëren van drie nieuwe gaten, zoals getoond in figuur 3.3 en figuur 3.4. De eerste figuur toont het gat waarin een cirkel geplaatst zal worden (de rode driehoek). De tweede figuur toont de nieuwe packing, nadat een cirkel geplaatst is in dit gat. Er zijn drie nieuwe kleinere gaten gemaakt, die in de volgende stappen ook zullen opgevuld worden indien mogelijk. Het algoritme zal deze gaten ook terug proberen opvullen. In figuur 3.5 en figuur 3.6 wordt respectievelijk de tussen-oplossing getoond voor wanneer er nog een cirkel is die klein genoeg is, en wanneer dit niet het geval is, om het onderste gat op te vullen.

3.6.1 Grootste cirkel zoeken die past in een *hole*

Bij het plaatsen van een cirkel in een gat wordt een zo groot mogelijke cirkel gezocht die in dit gat past. Dit is bij wijze van spreken de best-passende cirkel, vandaar *best-fit*. Meer uitleg over hoe bepaald wordt of een cirkel past vind je in sectie 3.6.2. Hier wordt het proces om te vinden *welke* cirkel past verder verduidelijkt. Er wordt logaritmisch gezocht

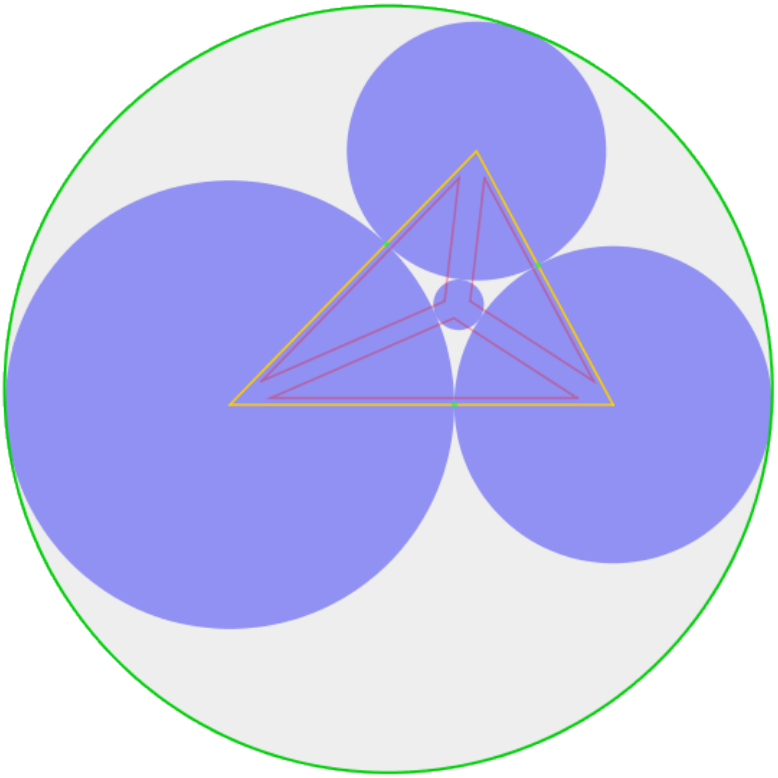

Figure 3.4: Packing na het opvullen van een gat

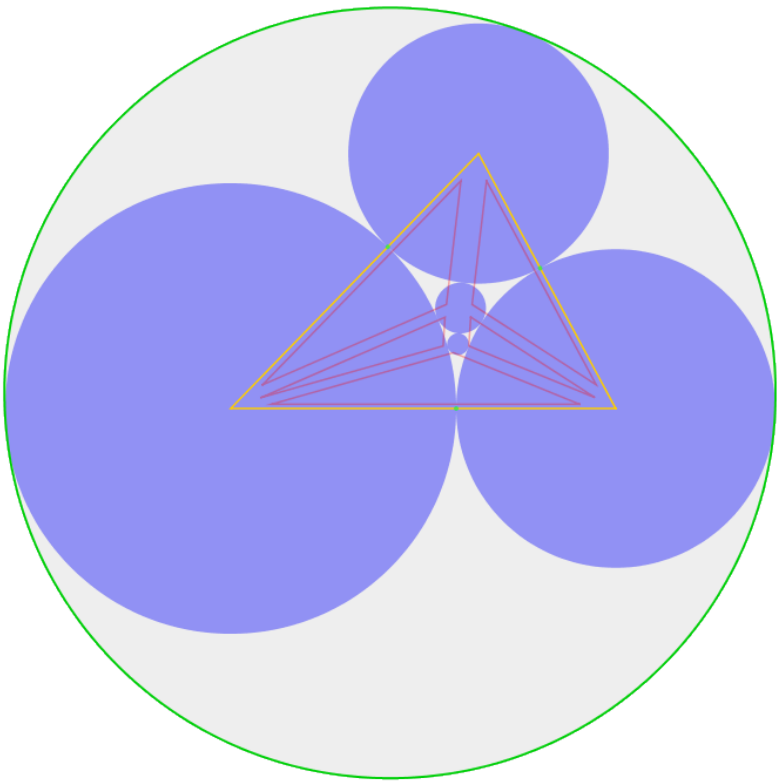

Figure 3.5: Packing na het opvullen van een tweede gat

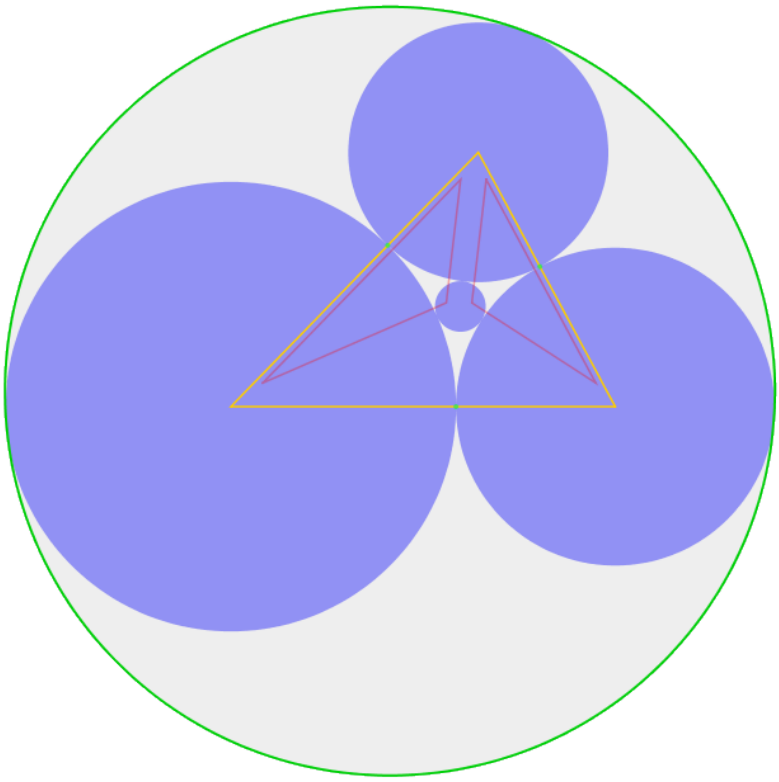

Figure 3.6: Packing als het opvullen van een tweede gat mislukt

door de lijst van cirkels om te bepalen welke cirkel de grootste is die past. Onderstaande code verduidelijkt dit proces.

```
Location findBestFitFor(Hole hole, List<Circle> sortedBigToSmall) {
    // Probeer eerst de kleinste cirkel
    int lower = sortedBigToSmall.size() - 1;
    Circle smallestCir = sortedBigToSmall.get(lower);
    Vector2 smallestPos = hole.tryFit(smallestCir);
    if (smallestPos == null) {
        return null;
    }

    // Probeer dan de grootste cirkel
    int upper = 0;
    Circle biggestCir = sortedBigToSmall.get(upper);
    Vector2 biggestPos = hole.tryFit(biggestCir);
    if (biggestPos != null) {
        return new Location(biggestPos, biggestCir);
    }

    // Logaritmisch zoeken
    Circle cir = null;
    Vector2 pos = null;
    while (lower - upper > 1) {
        int middle = (upper + lower) / 2;
```

```
23          cir = sortedBigToSmall.get(middle);
24          pos = hole.tryFit(cir);
25
26          if (pos == null) {
27              upper = middle;
28          }
29          else {
30              lower = middle;
31          }
32      }
33
34      cir = sortedBigToSmall.get(lower);
35      pos = hole.tryFit(cir);
36      if (pos != null) {
37          return new Location(pos, cir);
38      }
39      else {
40          return null;
41      }
42  }
```

De solver houdt de lijst van cirkels bij gesorteerd op grootte, dat is cruciaal om snel de beste-passende cirkel te vinden. Eerst worden de grootste en kleinste cirkel uitgeprobeerd (lijn 3). Indien de kleinste niet past zal het algoritme direct rapporteren dat dit gat te klein is. Het gat zal dan, zoals vermeld in sectie 3.6, verwijderd worden uit de lijst van mogelijk holes. Indien de grootste past (lijn 11) zal het algoritme onmiddellijk deze cirkel plaatsen in het gat. Er zijn immers geen grotere cirkels, dus deze is de cirkel die verondersteld wordt best te passen. Vervolgens word er een gebied bepaald in de overblijvende cirkels, waarin de best-passende cirkel zich bevind. Initieel ligt de boven-en ondergrens van dit gebied op de uiteinden van de overblijvende cirkels. De cirkel in de midden van dit gebied wordt dan uitgeprobeerd. Afhangende of deze wel of niet past zal de boven-of ondergrens aangepast worden. Dit wordt telkens herhaald tot er nog maar één cirkel over blijft. Dit is dan de grootste cirkel die past in het gat.

3.6.2 Bepalen of een cirkel past in een *hole*

Er is geen exact definitie van de grootte van een gat. Dit is niet mogelijk omdat de cirkels die het gat bepalen niet altijd aan elkaar raken. Het is echter wel mogelijk om te bepalen of een cirkel past.

Dit gebeurt door de cirkel die we willen testen te plaatsen in het gat. Eerst wordt de cirkel tegen twee van de cirkels in het gat geplaatst, met een cirkel-cirkel intersectie zoals beschreven in sectie 3.5. Een cirkel-cirkel intersectie heeft natuurlijk altijd twee punten. Hiervan moet er één gekozen worden. Door systematisch de gaten op te bouwen, wordt verzekerd dat telkens het punt in het gat gekozen wordt. Eenmaal dit punt bepaald is word de cirkel op deze plek gezet. Dan wordt gekeken of deze cirkel wel effectief in het gat geplaatst is, en of deze niet overlapt met de derde cirkel die het gat definieert.

```
1   public Vector2 tryFit(Circle cir) {
2       // Try to place circle
```

```
3       Vector2 pos = Helpers.getMountPositionFor(cir, first, second);
4
5       // Check that is inside the hole
6       boolean inside = Vector2.isInsideTriableBy(first.getPosition(),
            second.getPosition(), third.getPosition(), pos);
7       if (!inside) {
8           return null;
9       }
10
11      // Test for overlap
12      Location loc = new Location(pos, cir);
13      if (third.overlaps(loc)) {
14          return null;
15      }
16      return pos;
17  }
```

Op lijn 6 wordt verzekerd dat het middelpunt van de cirkel in het gat ligt. Dit voorkomt dat de geplaatste cirkel buiten het gat geplaatst wordt, en dus zeker niet kan overlappen met cirkels buiten het gat zonder ook te overlappen met één van de cirkels die het gat definiëren. Op lijn 13 wordt dan de overlap met de derde gat-definiërende cirkel nagekeken. Er kan geen overlap zijn met de eerste twee, de methode *getMountPositionFor* plaatst de cirkel zodanig dat deze de twee andere cirkels raakt, maar niet overlapt. Indien de cirkel in het gat past, en dus alle checks doorstaat, wordt de positie voor deze cirkel terug gegeven. De solver zal deze cirkel kan plaatsen in zijn oplossing.

Het is niet nodig om na te gaan of er overlap is met andere cirkels in de oplossing. Als er met een andere overlap zou zijn, moet dit zijn omdat de cirkel buiten het gat geplaatst is, of er is ook overlap met één van de cirkels in het gat zelf. Dit zorgt er voor dat er zeer weinig overlap-checks gedaan moeten worden, wat het algoritme zeer snel maakt.

3.7 Shell

De *shell* of *schil* is het tweede van de belangrijkste concepten van de heuristiek. Dit is de buitenste laag van cirkels in een (tussen-)oplossing van de solver. Deze wordt in de implementatie simpelweg bijgehouden als een geordende lijst van cirkels. Cirkels die naast elkaar staan in de lijst, grenzen ook aan elkaar in de shell. De cirkels in deze laag zijn met de klok mee gesorteerd, ten opzichte van het middelpunt van de omcirkel. De eerste en laatste cirkel in de lijst grenzen ook aan elkaar in de shell.

De heuristiek voorgesteld in deze thesis probeert steeds eerst alle gaten op te vullen. Maar wanneer er geen cirkels meer over zijn die klein genoeg zijn om te passen in gaten, word er een cirkel op de shell geplaatst. Op de shell worden alle posities tegen twee cirkels van de shell overwogen. Dit is een goede heuristiek voor de optimale positie, beperkt het aantal mogelijke posities voor de geplaatste cirkel enorm en maakt het algoritme dus zeer snel. Er wordt steeds een cirkel zo dicht mogelijk bij het centrum van de huidige tussen-oplossing geplaatst. Dit om het uitbreiden van de omcirkel zo min mogelijk te houden.

Eerst worden twee cirkels gekozen waartegen de nieuwe geplaatst zal worden. Hiervoor

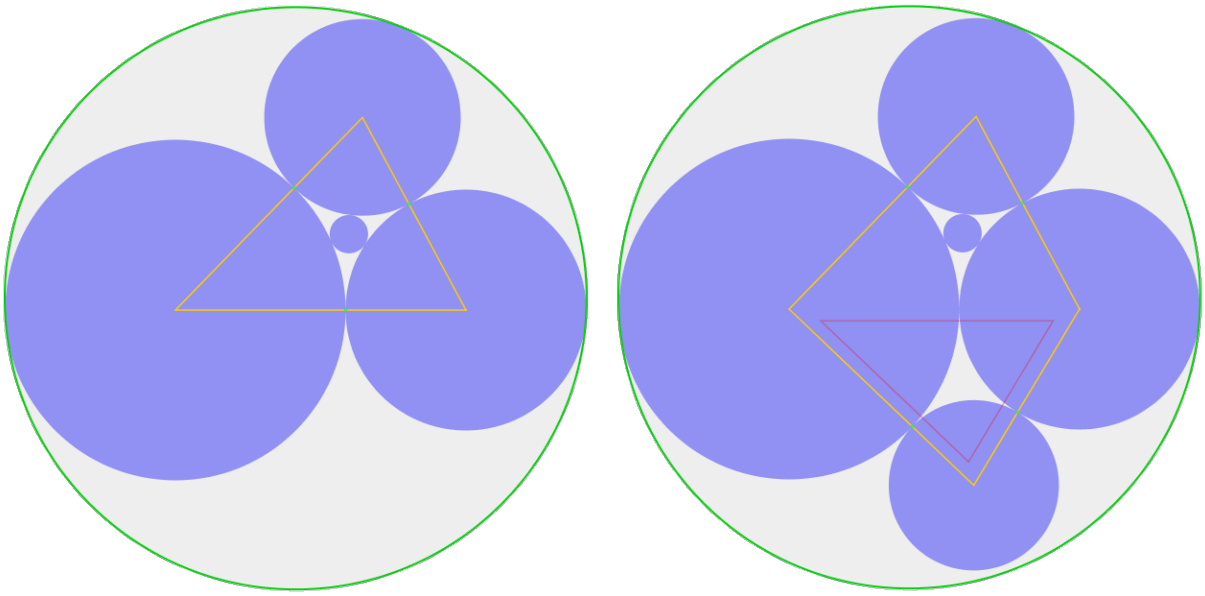

Figure 3.7: Het plaatsen van de grootste cirkel op de shell

wordt gekeken naar het middelpunt tussen alle cirkels die naast elkaar staan op de shell. De twee cirkels waarvoor het middelpunt het dichtst bij het centrum van de huidige omcirkel ligt worden gekozen als kandidaten om de derde cirkel tegen te plaatsen. Dan wordt er gezocht naar een zo groot mogelijke cirkel die daar past op de shell. Indien geen enkele cirkel past wordt één van de twee kandidaat cirkels verwijderd uit de shell. Welke verwijderd wordt verduidelijkt in sectie 3.7.2. Dit zorgt er voor dat de shell veranderd en *groeit* naar buiten. In de volgende stap van het algoritme zal dan ook een andere positie voor een cirkel uitgeprobeerd worden. Indien er wel een cirkel past wordt deze toegevoegd aan de shell, en geeft deze aanleiding tot een nieuw gat. Dit gat zal vervolgens opgevuld worden indien mogelijk, zoals beschreven in sectie 3.6. Op figuur 3.7 wordt getoond hoe de packing veranderd wanneer een cirkel geplaatst wordt op de shell. Eerst wordt een packing zonder gaten getoond en vervolgens de packing nadat een cirkel op de shell geplaatst is. Hierop is duidelijk te zien hoe de shell aangepast is geweest, en deze plaatsing geleid heeft tot een nieuw gat. Zoals eerder gezegd is het mogelijk dat op een positie op de shell de grootste cirkel niet past. Dit wordt geïllustreerd in figuur 3.8. Het is ook mogelijk dat geen enkele cirkel nog past op de shell, zoals getoond in figuur 3.9.

3.7.1 Slim de omcirkel berekenen gebaseerd op de shell

Bij het kiezen van de kandidaat cirkels om een cirkel tegen te plaatsen op de shell wordt gebruik gemaakt van de omcirkel van de huidige tussen-oplossing. Zoals eerder vermeld kan voor elke (tussen-)oplossing de omcirkel berekend worden. Dit gebeurt door een aangepaste versie van het Welz algoritme voor de omcirkel van punten beschreven in [16]. De implementatie is gebaseerd op de implementatie in [14]. Het is een recursief algoritme dat in lineaire tijd de omcirkel kan berekenen. Het idee is dat de omcirkel van een aantal cirkels (of punten) volledig gedefinieerd is door maximum drie cirkels. Het algoritme vind deze twee of drie cirkels.

Het is eenvoudig in te zien dat deze drie cirkels aan de buitenkant van een (tussen-)oplossing zullen liggen. Dit is ook net wat de shell is, de cirkels aan de buitenkant van

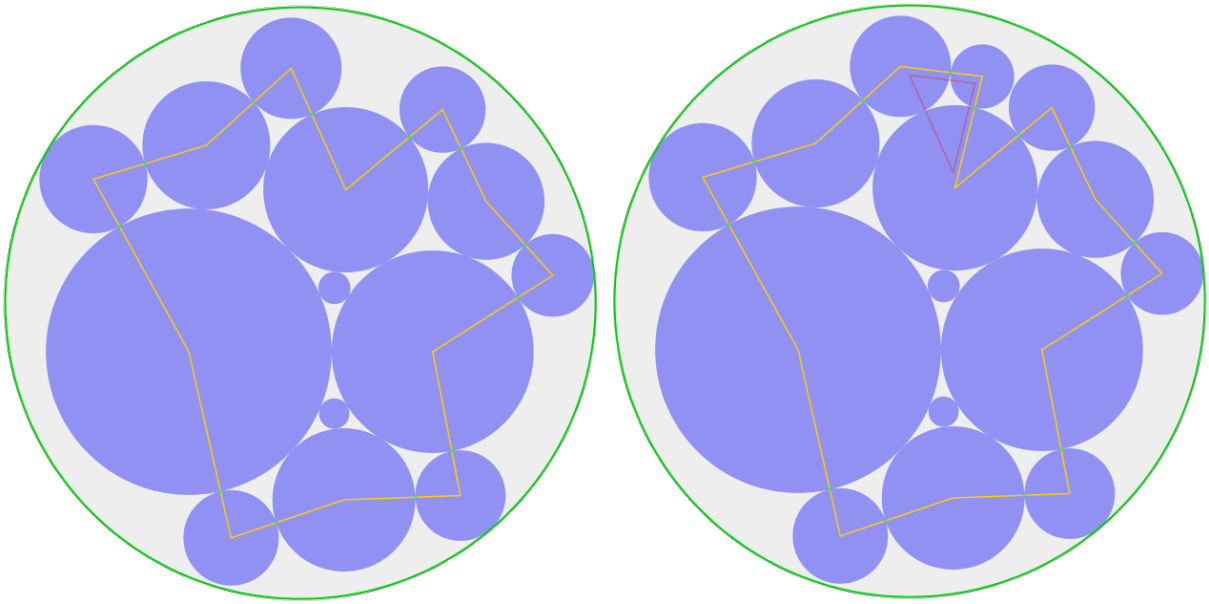

Figure 3.8: Het plaatsen van een kleinere cirkel op de shell

Focus: midden bovenaan

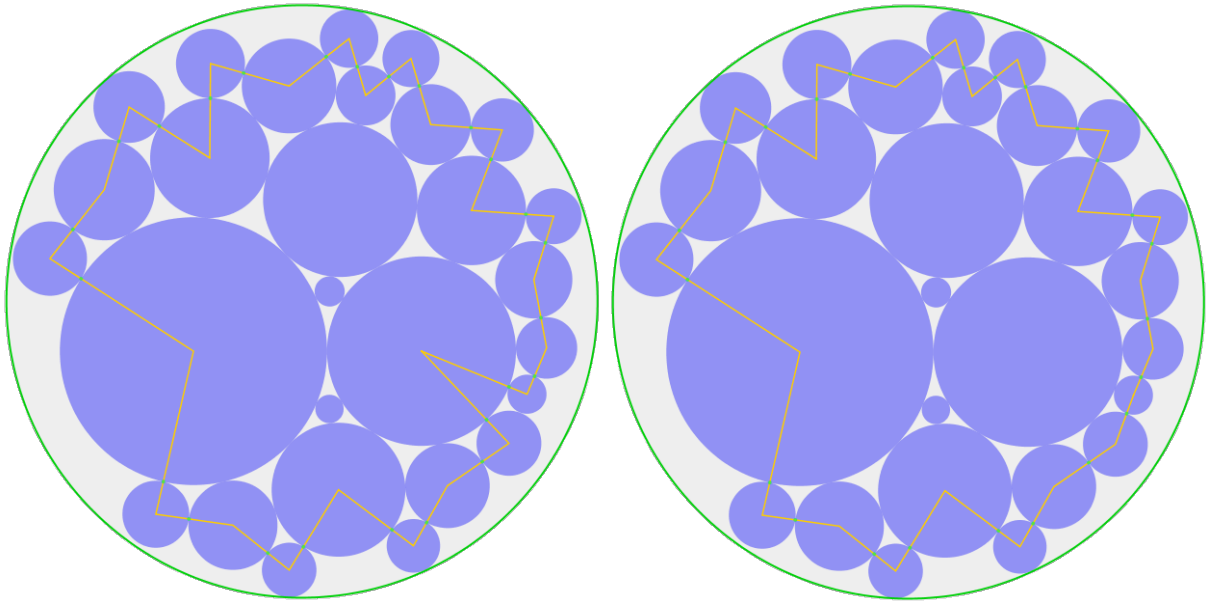

Figure 3.9: Shell aanpassen als geen enkele cirkel past

Focus: rechts onderaan

een oplossing. Het is dus mogelijk om in elke stap van de solver de omcirkel zeer efficient te berekenen. De complexiteit is dan slechts lineair in de omtrek van de huidige (tussen-)oplossing, en niet in het totaal aantal cirkels in de oplossing. Aangezien de omcirkel regelmatig moet herberekend worden doorheen het algoritme is dit een zeer interessante optimalisatie.

3.7.2 Bepalen of een cirkel past op de shell

Om te bepalen of een cirkel op de shell past plaatsen we de cirkel eerst tegen twee cirkels op de shell. Dit gebeurt op de zelfde manier als het plaatsen van een cirkel tegen twee cirkels van een gat. De exacte methode is reeds uitgelegd in sectie 3.5. Eenmaal deze positie gekend is, wordt er nagekeken of dit niet tot overlap leidt. Indien er overlap is is het niet mogelijk om de cirkel daar op de shell te plaatsen, en wordt er informatie terug gegeven over welke cirkel op de shell voor problemen zorgt.

Om na te gaan of er overlap is, wordt er systematisch een aantal cirkels op de shell nagekeken. Het is niet nodig andere cirkels, die niet op de shell zitten, na te kijken voor overlap. Een nieuwe cirkel wordt steeds op de shell geplaatst zodanig dat als er overlap zou zijn met cirkels in de oplossing, minstens één van die cirkels op de shell ligt. Doordat ik geen exacte definitie van de shell heb, ze is enkel gedefinieerd door het algoritme dat ze opbouwt, is het niet mogelijk om hier een echt bewijs van te geven. Uit empirische tests lijkt dit echter wel te kloppen. Verdere bedenkingen hierbij kan u terug vinden in hoofdstuk 5. Om deze overlappings-test zeer snel te kunnen uitvoeren worden eerst cirkels dicht bij nagekeken. Er staat ook een limiet op het aantal cirkels die getest worden. Uit tests, tot 5000 cirkels, blijkt dat drie cirkels aan elke kant van de shell nakijken genoeg is. Voor de meeste gevallen is één cirkel nakijken genoeg, maar sommige randgevallen (wanneer meerdere kleine cirkels dicht bij elkaar staan op de shell) vragen deze extra checks.

Het aantal cirkels dat nagekeken wordt is een hyperparameter *checkRadius* van het algoritme. De volgende code toont hoe de overlap nagekeken wordt:

```
for (int i = 1; i <= checkRadius; ++i) {
    int prevIndex = (firstIndex + shell.size() - i) % shell.size();
    Location prev = shell.get(prevIndex);
    if (loc.overlaps(prev)) {
        toRemove = first;
        break;
    }
    int nextIndex = (secondIndex + i) % shell.size();
    Location next = shell.get(nextIndex);
    if (loc.overlaps(next)) {
        toRemove = second;
        break;
    }
}
```

Er wordt dus om te beurt links en rechts van de huidige positie op de shell nakeken voor overlap. Indien er overlap gevonden is, wordt bijgehouden aan welke kant van de shell dit gebeurt is. *toRemove* is dus uiteindelijk $first$ of $secons$, wat aan geeft of er aan

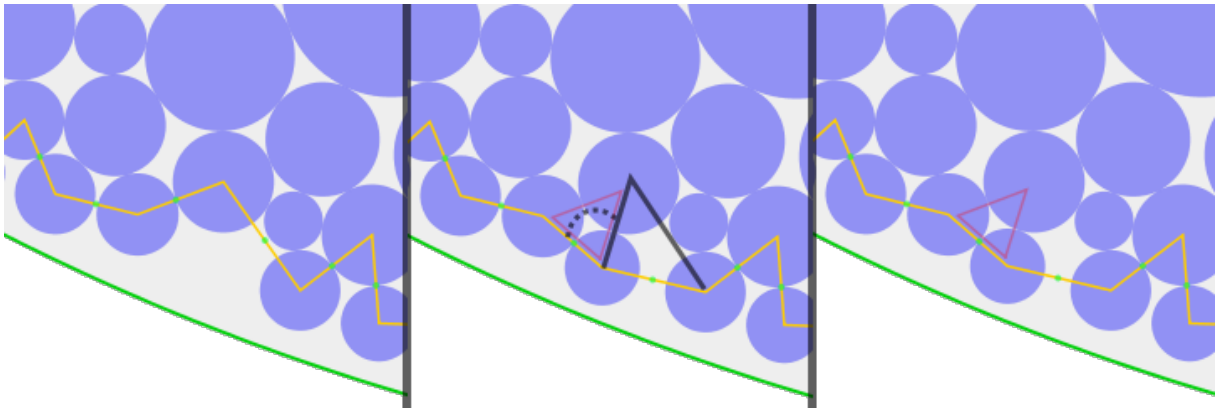

Figure 3.10: Een cirkel veroorzaakt een tegen-ge-klok shell

de kant van de eerste cirkel, of van de tweede cirkel, overlap voorkomt. *toRemove* kan ook *null* zijn indien er geen overlap is. Deze informatie wordt dan gebruikt door de solver om te bepalen of de cirkel geplaatst kan worden ($toRemove = null$), of dat de shell aangepast moet worden. Meer informatie hierover in sectie 3.7.3

3.7.3 Een cirkel plaatsen op de shell

Om een cirkel te plaatsen op de shell wordt eerst de grootste cirkel is die past op de shell logaritmisch gezocht in de lijst van nog-te-plaatsen cirkels. Dit gebeurt op analoge manier als het zoeken naar de grootste cirkel die past in een gat, zoals beschreven in sectie 3.6.1. Indien er een cirkel gevonden wordt die past op de shell wordt de shell uitgebreid met deze cirkel. De cirkel wordt dan tussen de cirkels, waartegen hij geplaatst is, gezet in de shell. Dit wordt geïllustreerd in figuur 3.7, figuur 3.8. De gele lijn (die de shell voorstelt, zoals beschreven in hoofdstuk 2) wordt uitgebreid met een extra punt. Ook zorgt dit er voor dat een nieuw gat gemaakt wordt waar mogelijk kleinere cirkels in passen.

Een laatste check die gebeurt is om na te gaan of door het plaatsen van de cirkel de shell nog juist gevormd is. Indien een cirkel geplaatst wordt zodat de cirkels niet meer met de klok mee gesorteerd zijn, zou dit er voor kunnen zorgen dat foutieve plaatsingen voorkomen. Om dit te verzekeren wordt de gerichte hoek tussen de geplaatste cirkel en de cirkels waartegen hij geplaatst is nagekeken. Indien deze hoek negatief is wordt de shell hier voor aangepast. Dit wordt getoond in figuur 3.10. Het eerste deel toont de configuratie waarop een cirkel geplaatst zal worden. Het laatste deel de toteindelijke configuratie van de shell. Het middelste deel toont met een zwarte lijn het deel van de shell dat verwijderd is om er voor te zorgen dat deze kloksgewijs gesorteerd blijft. Op figuur 3.11 wordt getoond wat er kan gebeuren als deze check niet gebeurt.

3.8 Conclusie

Dit hoofdstuk gaf een volledig overzicht van de nieuwe constructieve heuristiek. Waar nodig werd het uitleg verduidelijkt met code uit de implementatie. Er werden ook de verschillende veronderstellingen geformuleerd die gebruikt worden om het algoritme zeer snel te maken. Zie ook hoofdstuk 5 voor verdere bedenkingen hieromtrent.

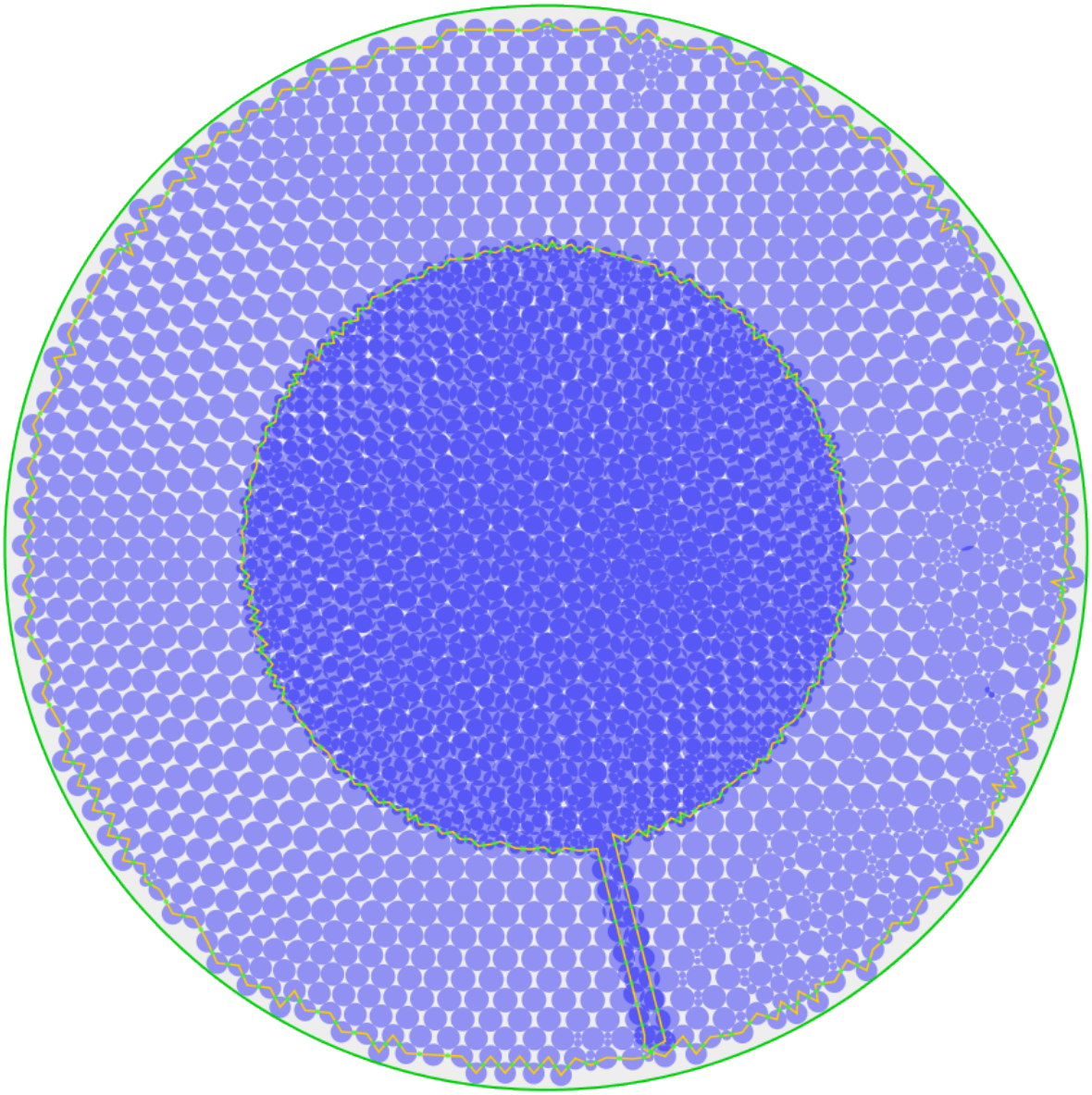

Figure 3.11: Mogelijke fout indien de shell niet met de klok mee gesorteerd is

Hoofdstuk 4

Resultaten

Hoofdstuk 5

Opmerkingen en verder werk

// TODO Bespreking van verbeteringen/andere mogelijke strategieën/losse eindjes // TODO Bedenkingen bij veronderstellingen in het algorime, deze lijken te kloppen maar hebben geen bewijs. // TODO verder uitweiden over clockwise/counterclockwise en dat hier tussen gewisseld wordt voor een stabielere fout?

Hoofdstuk 6

Conclusie

Bibliography

[1] Hakim Akeb and Yu Li. A basic heuristic for packing equal circles into a circular container. *Comput. Oper. Res*, 33:2125–2142, 2006.

[2] I Al-Mudahka, Mhand Hifi, and Rym M'Hallah. Packing circles in the smallest circle: an adaptive hybrid algorithm. *Journal of the Operational Research Society*, 62(11):1917–1930, 2011.

[3] P. Bollansée. Circle packing. `https://github.com/circle-packing/best-fit`.

[4] Edmund K Burke, Graham Kendall, and Glenn Whitwell. A new placement heuristic for the orthogonal stock-cutting problem. *Operations Research*, 52(4):655–671, 2004.

[5] John A George, Jennifer M George, and Bruce W Lamar. Packing different-sized circles into a rectangular container. *European Journal of Operational Research*, 84(3):693–712, 1995.

[6] Ronald L Graham and Boris D Lubachevsky. Repeated patterns of dense packings of equal disks in a square. *the electronic journal of combinatorics*, 3(1):R16, 1996.

[7] Ronald L Graham, Boris D Lubachevsky, Kari J Nurmela, and Patric RJ Östergård. Dense packings of congruent circles in a circle. *Discrete Mathematics*, 181(1):139–154, 1998.

[8] Andrea Grosso, ARMJU Jamali, Marco Locatelli, and Fabio Schoen. Solving the problem of packing equal and unequal circles in a circular container. *Journal of Global Optimization*, 47(1):63–81, 2010.

[9] Mhand Hifi and Rym M'Hallah. Approximate algorithms for constrained circular cutting problems. *Computers & Operations Research*, 31(5):675–694, 2004.

[10] Mhand Hifi, Vangelis Th Paschos, and Vassilis Zissimopoulos. A simulated annealing approach for the circular cutting problem. *European Journal of Operational Research*, 159(2):430–448, 2004.

[11] Boris D Lubachevsky. How to simulate billiards and similar systems. *Journal of Computational Physics*, 94(2):255–283, 1991.

[12] Boris D Lubachevsky and Ronald L Graham. Curved hexagonal packings of equal disks in a circle. *Discrete & Computational Geometry*, 18(2):179–194, 1997.

[13] E. Specht. Packomania. `http://www.packomania.com/`. Accessed: 2016-05-23.

[14] Sunshine. Computing the smallest enclosing disk in 2d. `http://www.sunshine2k.de/coding/java/Welzl/Welzl.html`, 2008.

[15] Huaiqing Wang, Wenqi Huang, Quan Zhang, and Dongming Xu. An improved algorithm for the packing of unequal circles within a larger containing circle. *European Journal of Operational Research*, 141(2):440–453, 2002.

[16] Emo Welzl. *Smallest enclosing disks (balls and ellipsoids)*. Springer, 1991.

AFDELING
Straat nr bus 0000
3000 LEUVEN, BELGIË
tel. + 32 16 00 00 00
fax + 32 16 00 00 00
www.kuleuven.be