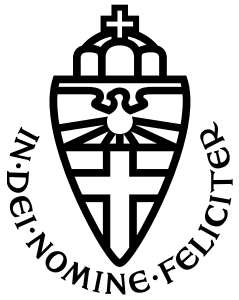
Radboud Universiteit Nijmegen



Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Algoritmen voor het maken van *string art*

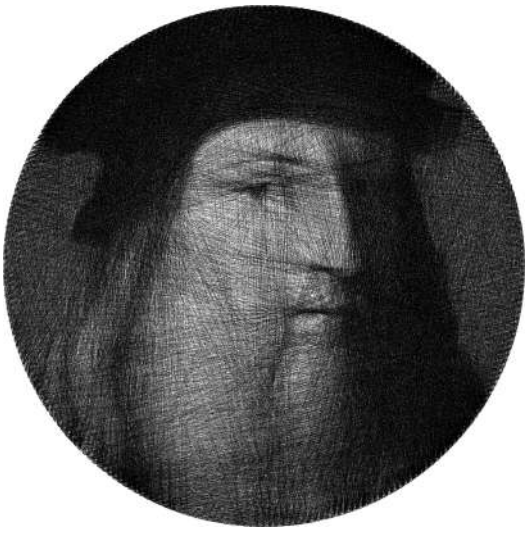
Een toepassing van wiskunde in de kunst

Bachelorscriptie Wiskunde

Auteur:
Maaike Schouten

Begeleider:
dr. W. Bosma

Tweede lezer:
dr. H. Don



Oktober 2022

Voorwoord

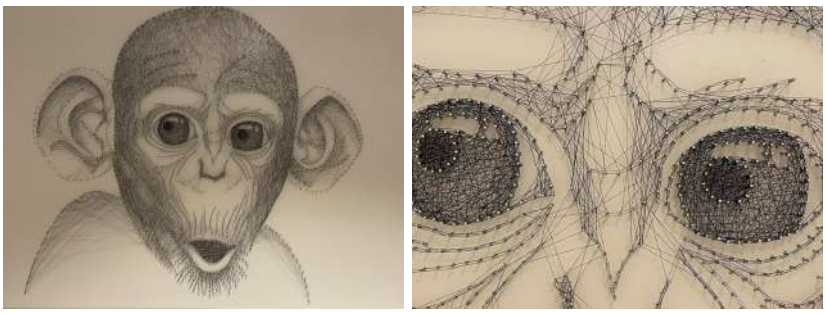
Deze scriptie gaat over *string arts*, schilderijen waarbij alleen spijkertjes en draad is gebruikt. Bij *string arts* worden er spijkers in een houten plank geslagen en daarna worden de spijkers met elkaar verbonden met draad. Op die manier maken kunstenaars afbeeldingen van bijvoorbeeld mensen of dieren. Sinds een paar jaar maak ik zelf ook *string arts*, in Figuur 1 is een voorbeeld te zien van een *string art* die ik heb gemaakt.

Een jaar geleden kwam ik een kunstenaar tegen op het internet die ook *string arts* maakte, maar dan op een bijzondere manier. Hij had een algoritme geschreven dat bepaalt welke spijkers met elkaar verbonden moeten worden. Zijn *string arts* zijn altijd cirkelvormig, met de spijkers gelijk verdeeld over de rand van de cirkel. Met het blote oog is niet te zien welke spijkers je met elkaar moet verbinden om er bijvoorbeeld een portret van te maken. Maar een computer kan dit wel.

Na nog wat surfen op het internet vond ik een tweede algoritme om dezelfde *string arts* te maken. Het leek me leuk om mijn studie wiskunde te combineren met mijn grote passie voor kunst en specifiek mijn interesse in *string arts*. Wieb Bosma zag dit ook wel zitten en samen zijn we aan de slag gegaan om de twee algoritmen te bestuderen en te onderzoeken.

Hoewel de algoritmen geen hele moeilijke wiskunde bleken te bevatten, heb ik wel ervaren dat ik dankzij mijn studie wiskunde de benodigde manier van denken en redeneren bezat om de algoritmen te begrijpen en te implementeren. Het artikel dat ik had gevonden was niet altijd even duidelijk, het vergde veel puzzelen en discussiëren met mijn begeleider Wieb Bosma om te begrijpen wat er precies werd gedaan. Verder had ik nog niet heel veel ervaring met programmeren, waardoor dat soms veel tijd kostte, maar ik wel heb ontdekt dat ik het erg leuk vind om te doen.

Al met al was het een interessant proces waar ik met veel plezier op terugkijk en ben ik erg blij met de resultaten die te zien zijn in deze scriptie.



Figuur 1: Een *string art* gemaakt zonder computer

Inhoudsopgave

Hoofdstuk 1

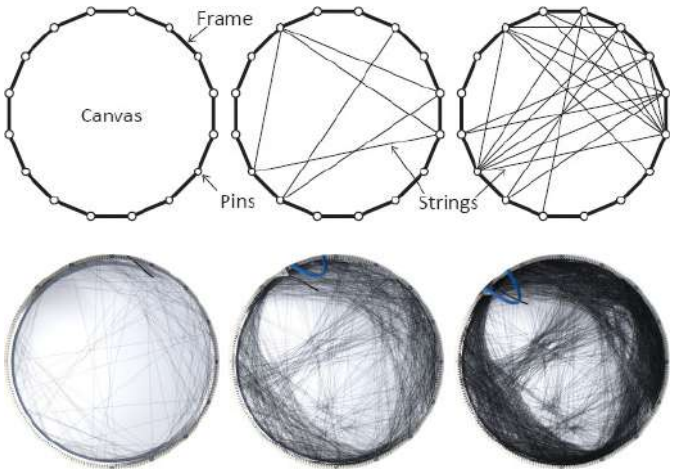
Inleiding

String art is een vorm van kunst waarbij gebruik wordt gemaakt van spijkertjes en draad. Spijkers worden in een houten plank geslagen en vervolgens in een bepaald patroon met elkaar verbonden door een draad. Kunstenaars maken op deze manier schilderijen, ze "schilderen" met draad.

In 2016 bedacht Griekse kunstenaar Petros Vrellis een nieuwe manier om *string art* te maken [8]. Hij schreef een algoritme waarin hij de computer laat bepalen welke spijkers met elkaar verbonden moeten worden voor de beste resultaten. Deze *string arts* hebben altijd de vorm van een cirkel waarbij de spijkers gelijk verdeeld zijn over de rand van de cirkel. In Figuur 2 is te zien hoe zo'n *string art* eruit ziet. Het is heel moeilijk, of onmogelijk, om zelf te zien welke spijkers met elkaar verbonden moeten worden om er bijvoorbeeld een portret van te maken. Maar Vrellis vond een manier om de computer dit te laten bepalen.

Voor het maken van een algoritme om *string art* te maken, is het nodig om lijnen om te zetten naar pixels. Een computer werkt niet met lijnen, maar alleen met pixels, kleine vierkante blokjes die samen een digitale foto vormen. Lijnen moeten benaderd worden met de pixels en dat gebeurt met een lijnalgoritme. Voor deze scriptie bestuderen we het lijnalgoritme van Bresenham [3] en het lijnalgoritme van Wu [9]. Dat van Bresenham maakt alleen gebruik van witte en zwarte pixels, terwijl dat van Wu ook gebruik maakt van verschillende tinten grijs ertussenin. Met behulp van de lijnalgoritmen kunnen we het algoritme van Vrellis implementeren. We onderzoeken hoe goed het algoritme werkt en we bekijken de invloed van verschillende variabelen op het eindresultaat.

Ter vergelijking bestuderen we nog een tweede algoritme uit een artikel dat is geschreven door Michael Birsak in 2018 [1]. Waar het algoritme van Vrellis in elke iteratie alleen de mogelijke lijnen vanuit één spijker bekijkt, overweegt Birsak in elke iteratie álle mogelijke lijnen van de *string art*. Dit schept de verwachting dat het algoritme van Birsak betere resultaten geeft, dat wil zeggen: een *string art* waarin de originele portretfoto beter in te herkennen is. Ook het algoritme van Birsak implementeren we om de resultaten te kunnen bekijken.

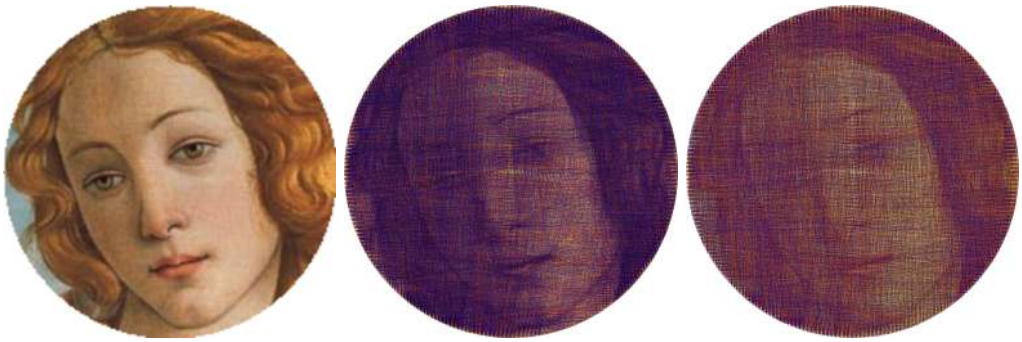


Figuur 2: Een schematische weergave van een *string art* gemaakt met een algoritme

Ondanks onze verwachtingen, blijkt het algoritme van Birsak geen directe verbetering op het algoritme van Vrellis. Sterker nog, de implementatie van het algoritme van Vrellis werkt een stuk sneller en lijkt op het eerste gezicht beter herkenbare resultaten te geven. In een vervolgonderzoek zou het nuttig zijn om de implementatie van het algoritme van Birsak nog verder te verbeteren. Op dit moment werkt het erg traag en worden bovendien niet alle stappen goed uitgevoerd, zoals later te lezen is. Het sneller maken van de implementatie zou het mogelijk maken om meer resultaten van dit algoritme te produceren. Bovendien zou een verbetering van de implementatie als gevolg kunnen hebben dat de resultaten beter worden dan die van Vrellis.

Een andere suggestie voor een vervolgonderzoek is het onderzoeken van het gebruik van kleur. Beide algoritmen gebruiken nu foto's met grijswaarden als input en de *string arts* worden gemaakt met zwart draad. Maar Vrellis heeft op zijn website één voorbeeld van een *string art* gemaakt met zwart, rood, geel en blauw draad [8]. We hebben geprobeerd deze gekleurde *string art* te reproduceren, door de vier kleuren te isoleren uit een gekleurde foto en het algoritme vier keer te runnen op deze aparte kleuren. Het geeft nog geen denderende resultaten, want meestal overheerst één kleur heel erg, zoals te zien is in Figuur 3. Met meer tijd en onderzoek is dit wellicht te verbeteren.

Deze scriptie is als volgt opgebouwd: in Hoofdstuk 2 en 3 bespreken we respectievelijk de lijnalgoritmen van Bresenham en Wu. In Hoofdstuk 4 leggen we het algoritme van Vrellis uit en in Hoofdstuk 5 bespreken we de implementatie en de resultaten van het algoritme. Hetzelfde doen we voor het algoritme van Birsak in Hoofdstuk 6 en 7. In Hoofdstuk 6 maken we tevens een vergelijking tussen beide algoritmen, waarbij we in Hoofdstuk 7 ook resultaten van beide algoritmen tegenover elkaar zetten.



(a) Gezicht van Venus van De geboorte van Venus (b) Output met blauw als laatste kleur (c) Output met rood als laatste kleur

Figuur 3: Twee *string arts* met kleur van het gezicht van Venus uit het schilderij De geboorte van Venus

Hoofdstuk 2

Lijnalgoritme van Bresenham

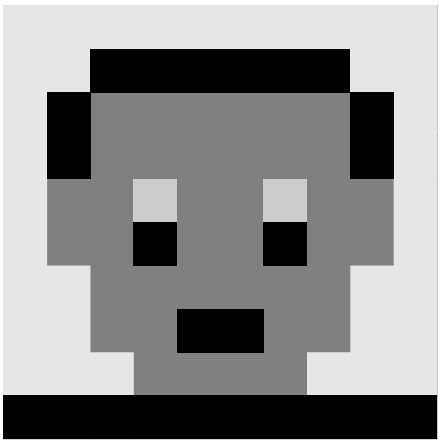
In Hoofdstuk 4 en 6 zullen we twee algoritmen om *string arts* te maken bespreken. Een belangrijke stap in deze twee algoritmen is het bepalen van de pixels onder een lijn. Digitale afbeeldingen of foto's bestaan uit kleine vierkante blokjes, pixels. Een schuine lijn kan daardoor nooit precies worden weergegeven, maar moet worden benaderd door de pixels. Er zijn verschillende manieren om dat te doen. Het bekendste algoritme is dat van Jack Bresenham, waarbij een lijn benaderd wordt door enkel zwarte pixels [3]. Een uitbreiding van dit algoritme is gemaakt door Xiaolin Wu, die naast zwarte pixels ook pixels met verschillende grijswaarden gebruikte [9]. De lijn die hierdoor ontstaat noemen we *anti-aliased*. In dit hoofdstuk bespreken we het algoritme van Bresenham en in het volgende hoofdstuk komt het algoritme van Xiaolin Wu aan bod.

2.1 Digitale afbeeldingen

Voordat we het algoritme uitleggen, bespreken we een aantal conventies over digitale afbeeldingen of plaatjes. Zo'n afbeelding bestaat dus uit pixels en elke pixel heeft een kleur, of een grijswaarde als het een afbeelding is zonder kleur. Bij onze *string arts* werken we alleen met de grijswaarden van de pixels. We gebruiken hiervoor het meest toegankelijke systeem, namelijk de 8-bits verwerking met 256 ($= 2^8$) verschillende gradaties. Elke pixel krijgt een waarde van 0 tot en met 255, waarbij pixels met waarde 0 zwart zijn en pixels met waarde 255 wit zijn. Met het algoritme van Bresenham zullen de pixels dus een waarde van 0 of 255 hebben, terwijl bij het algoritme van Wu de pixels alle waarden tussen 0 en 255 kunnen hebben.

We kunnen een rechthoekige digitale afbeelding beschouwen als een matrix. Elke plek van de matrix heeft als waarde de grijswaarde van de pixel op die plek. Een pixel kun je aangeven met een paar (x, y) waarbij x de kolom en y de rij in de matrix aangeeft. Merk op dat we hierbij dus eerst de kolom en daarna de rij aanduiden, in plaats van andersom zoals we gewend zijn bij matrices. De pixel links bovenin is $(0, 0)$, de pixel ernaast $(1, 0)$, de pixel eronder $(0, 1)$, etc. In Figuur 4 is een voorbeeld te zien van een digitale afbeelding met ernaast de matrix met grijswaarden.

Naast een matrix kunnen we een afbeelding ook beschouwen als een assenstelsel gelegen in het vlak, waarbij de middens van de pixels coördinaten hebben met gehele getallen. De pixels worden nu aangeduid met hun middens en zijn nog steeds dezelfde paren als bij de matrix: de pixels links bovenin heeft als midden $(0, 0)$, de pixel ernaast heeft als midden $(1, 0)$, etc. Ook hier werkt het net anders dan gewoonlijk bij assenstelsels: de y-waarde loopt op in plaats van af als je naar beneden gaat. Voor het algoritme van Bresenham is deze beschouwing van een digitale afbeelding erg handig, omdat we een lijn in een afbeelding nu kunnen beschouwen als een lijn ten opzichte van een assenstelsel. We spreken af dat een lijn met dikte maximaal 1 pixel tussen twee pixels loopt, en nog specifieker tussen de middens van de twee pixels.



	0	1	2	3	4	5	6	7	8	9
0	231	231	231	231	231	231	231	231	231	231
1	231	231	0	0	0	0	0	0	231	231
2	231	0	128	128	128	128	128	128	0	231
3	231	0	128	128	128	128	128	128	0	231
4	231	128	128	205	128	128	205	128	128	231
5	231	128	128	0	128	128	0	128	128	231
6	231	231	128	128	128	128	128	128	231	231
7	231	231	128	128	128	128	128	128	231	231
8	231	231	231	128	128	128	128	231	231	231
9	0	0	0	0	0	0	0	0	0	0

Figuur 4: Een afbeelding bestaande uit pixels met grijswaarden

2.2 Algoritme

Het algoritme van Bresenham neemt de begin- en eindpixel, $P_1 = (x_1, y_1)$ en $P_2 = (x_2, y_2)$, als input. De eerste stap van het algoritme is het bepalen van $dx = x_2 - x_1$ en $dy = y_2 - y_1$. De rest van het algoritme wordt uitgevoerd in de richting die het grootst is. Als bijvoorbeeld dx het grootst is, wordt er in elke kolom tussen de begin- en eindpixel precies één pixel zwart gekleurd. In de rijen kunnen meerdere pixels zwart worden. Als dy het grootst is, wordt er in elke rij precies één pixel zwart gekleurd en kunnen in de kolommen meerdere pixels zwart worden.

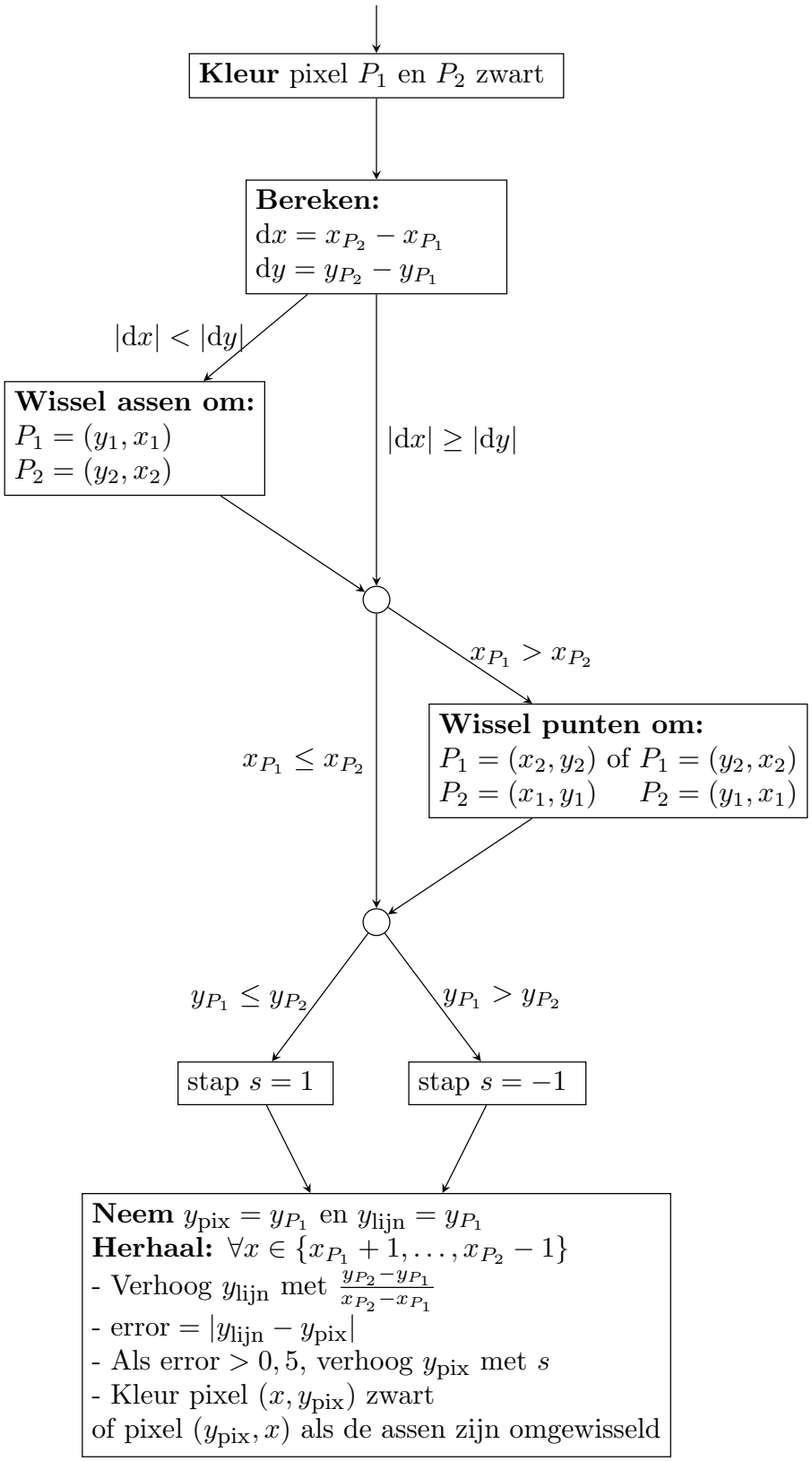
Om te bepalen welke pixel in een kolom (of rij) zwart gekleurd moet worden, berekenen we de waarde van de lijn in het midden van die kolom of rij. Stel, we zijn in de derde kolom van de afbeelding en willen bepalen welke pixel in die kolom zwart moet worden. We bepalen dan eerst de y-waarde van de lijn voor $x = 2$ (de derde kolom heeft in het midden x-waarde 2). Vervolgens kiezen we de pixel waarbij het midden van de pixel het dichtst bij de lijn is. Daarvoor zijn maar twee opties: de pixel met het midden vlak boven de lijn of de pixel met het midden vlak onder de lijn.

De y-waarde van de lijn, y_{lijn} kunnen we berekenen met behulp van de richtingscoëfficient $\frac{\mathrm{d}y}{\mathrm{d}x}$. Aan het begin van het algoritme is y_{lijn} gelijk aan y_1, daarna berekenen we bij elke stap y_{lijn} door $\frac{\mathrm{d}y}{\mathrm{d}x}$ erbij op te tellen. Als het verschil tussen y_{lijn} en de y-waarde van de pixel erboven, y_{pix}, groter wordt dan $0,5$, dan weten we dat de pixel eronder dichterbij is en verhogen we y_{pix} met 1 (let op: verhogen, want we tellen van boven naar beneden). Dit herhalen we voor elke kolom tussen de begin- en eindpixel, dus voor elke $x \in \{x_1 + 1, \ldots, x_2 - 1\}$. Hieronder zijn de stappen nog eens op een rijtje te zien:

1. y_{lijn} verhogen we met $\frac{\mathrm{d}y}{\mathrm{d}x}$;
2. error $= |y_{\text{lijn}} - y_{\text{pix}}|$;
3. Als error $> 0,5$, dan verhogen we y_{pix} met 1;
4. We kleuren pixel (x, y_{pix}) zwart.

Er zijn nog wel wat verschillende gevallen waar we rekening mee moeten houden. De bovenstaande stappen horen bij het geval dat dx het grootst is. Als dy het grootst is, lopen we alle rijen in plaats van kolommen langs. Om toch de stappen zoals ze hierboven staan uit te voeren, kunnen we de y-as en de x-as omwisselen. P_1 wordt dan (y_1, x_1) en P_2 wordt (y_2, x_2). Op die manier hebben we het geval dat dy het grootst is omgezet naar een geval dat dx het grootst is. Alleen in de laatste stap maken we nog een wijziging: we kleuren pixel (y_{pix}, x) zwart in plaats van pixel (x, y_{pix}).

Verder moeten we nog rekening houden met het begin- en eindpunt. Bovenstaande stappen gaan ervan uit dat $x_{P_1} \leq x_{P_2}$ en $y_{P_1} \leq y_{P_2}$. Als geldt dat $x_{P_1} > x_{P_2}$, moeten we de kolommen



Figuur 5: Het Bresenham algoritme

van rechts naar links doorlopen in plaats van andersom. Een andere optie is om het begin- en eindpunt om te wisselen. Op die manier geldt dat $x_{P_1} < x_{P_2}$ en kunnen we alsnog de kolommen van links naar rechts langslopen en bovenstaande stappen uitvoeren. Ten slotte het geval dat $y_{P_1} > y_{P_2}$: hiervoor geldt dat y_{lijn} steeds kleiner wordt. In plaats van y_{pix} dus constant groter te maken als het verschil groter dan $0,5$ wordt, moeten we y_{pix} nu kleiner maken. We spreken af dat we y_{pix} verhogen met stap s. Afhankelijk van y_{P_1} en y_{P_2} is s dan 1 of -1. In Figuur 5 is het totale algoritme stap voor stap te zien en in Voorbeeld 2.1 laten we een voorbeeld van een lijn zien waarvoor we stap voor stap de pixels bepalen.

Voorbeeld 2.1. We willen een lijn trekken van pixel $(1, 2)$ naar $(8, 6)$, zie Figuur 6a. De lijn loopt van het midden van de beginpixel, $(1, 2)$, tot het midden van de eindpixel, $(8, 6)$.

De eerste stap is dx en dy berekenen:

$$dx = x_{P_2} - x_{P_1} = 8 - 1 = 7;$$
$$dy = y_{P_2} - y_{P_1} = 6 - 2 = 4.$$

In dit voorbeeld is dx het grootst, dus we hoeven de x-as en de y-as niet om te wisselen. Bovendien is $x_{P_1} \leq x_{P_2}$, dus we wisselen het begin- en eindpunt ook niet om. Verder is $y_{P_1} \leq y_{P_2}$, dus $s = 1$. We nemen $y_{\text{pix}} = y_{P_1} = 2$, $y_{\text{lijn}} = y_{P_1} = 2$ en lopen alle $x \in \{2, 3, \ldots, 7\}$ langs. Voor $x = 2$ geldt:

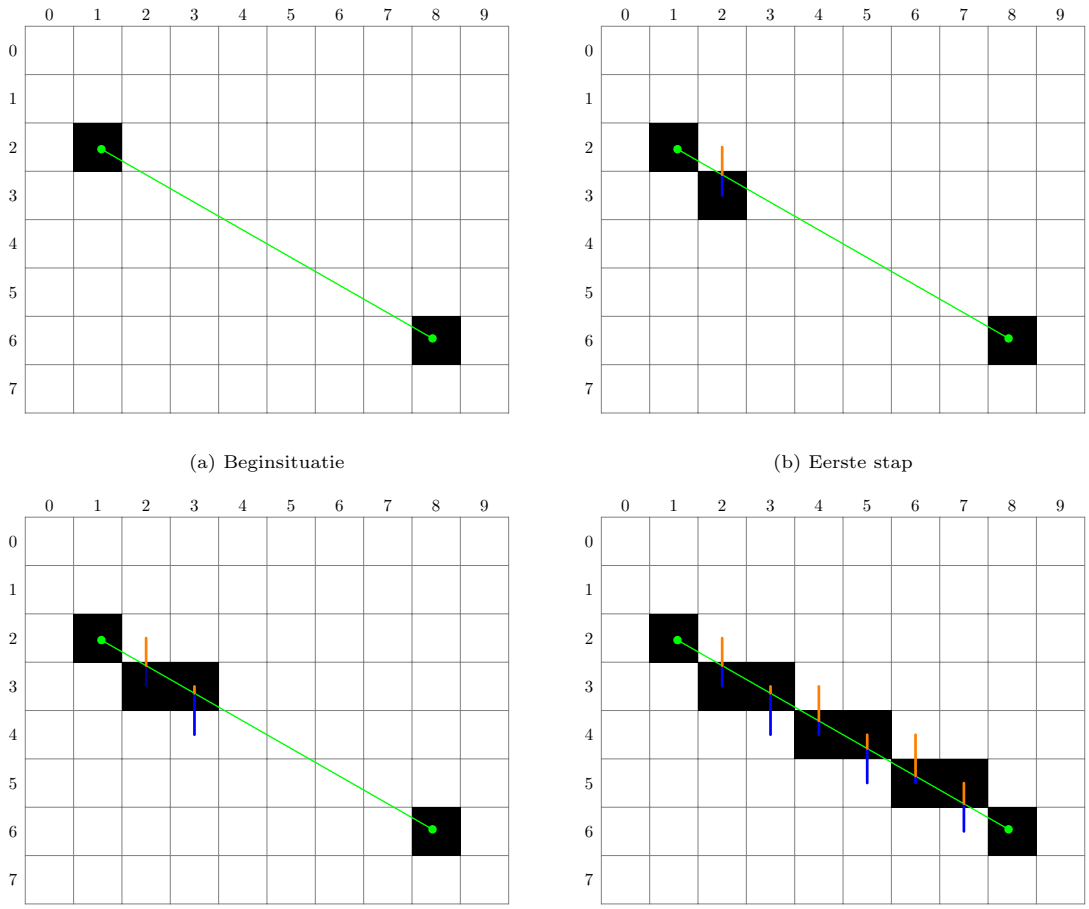
$$y_{\text{lijn}} = 2 + \frac{x_{P_2} - x_{P_1}}{y_{P_2} - y_{P_1}} = 2 + \frac{4}{7} = 2,57142857;$$
$$\text{error} = |y_{\text{lijn}} - y_{\text{pix}}| = |2,57142857 - 2| = 0,57142857.$$

De error komt overeen met het oranje lijnstuk in Figuur 6b. De error is groter dan $0,5$. Het betekent dat het blauwe lijnstuk kleiner is dan $0,5$ (het oranje en het blauwe lijnstuk zijn samen precies 1) en dat het verschil met het midden van de pixel met y-waarde 3 kleiner is. We verhogen y_{pix} dus met 1. De pixel die we zwart kleuren is $(2, 3)$.

De volgende stap ziet er als volgt uit:

$$x = 3,\ y_{\text{pix}} = 3;$$
$$y_{\text{lijn}} = 2,57142857 + \frac{4}{7} = 3,14285714;$$
$$\text{error} = |3,14285714 - 3| = 0,14285714;$$
$$\text{error} < 0,5 \Rightarrow y_{\text{pix}} \text{ blijft } 3.$$

De error is nu kleiner dan 0,5. Dat betekent dat de pixel met y-waarde 3 het dichtst bij de lijn is. We hoeven y_{pix} niet te verhogen en voegen pixel $(3, 3)$ toe aan de lijn, zie Figuur 6c. Als we deze stap herhalen voor elke x krijgen we de lijn Figuur 6d.



(a) Beginsituatie

(b) Eerste stap

(c) Tweede stap

(d) Lijn met pixels

Figuur 6: Bepalen van de pixels van de lijn met behulp van Bresenham

2.3 Implementatie

Hieronder is een implementatie van het algoritme van Bresenham in Python [7] te zien. In Hoofdstuk 5 en Hoofdstuk 7 laten we de implementaties zien van de algoritmen van Vrellis en Birsak. De onderstaande functie om de pixels van een lijn te bepalen gebruiken we in die implementaties. De onderstaande functie geeft als output twee lijsten, eentje met de x-coördinaten en eentje met de y-coördinaten van de zwarte pixels. Voor de implementaties van de algoritmen van Vrellis en Birsak is het handig om de output op deze manier te geven in plaats van één lijst met paren.

```
# this functions returns two lists:
# - xpix with x-coordinates of the black pixels
# - ypix with y-coordinates of the black pixels

def Bresenham(pin1, pin2):
    (x1, y1), (x2, y2) = pin1, pin2
    xpix = [x1, x2]
    ypix = [y1, y2]

    switch = abs(x2-x1) < abs(y2-y1)
    if switch:
        x1, y1 = y1, x1
        x2, y2 = y2, x2

    if x1 > x2:
        x1, x2 = x2, x1
        y1, y2 = y2, y1

    s = 1 if y1 <= y2 else -1

    y_pix = y_lijn = y1
    slope = (y2-y1)/(x2-x1)
    for x in range(x1+1, x2):
        y_lijn += slope
        error = abs(y_lijn - y_pix)
        if error > 0.5:
            y_pix += s
        if switch:
            xpix += [y_pix]
            ypix += [x]
        else:
            xpix += [x]
            ypix += [y_pix]

    return ypix, xpix
```

Hoofdstuk 3

Lijnalgoritme van Xiaolin Wu

Xiaolin Wu borduurt voort op het algoritme van Bresenham, maar introduceert het gebruik van verschillende grijswaarden om de lijn nog beter te benaderen. In het algoritme van Bresenham wordt de pixel met de kleinste error gekozen en helemaal zwart gekleurd. De andere pixel blijft wit. In het algoritme van Wu krijgen beide pixels een grijswaarde, afhankelijk van de error. De waarde 255 wordt als het ware verdeeld over de twee pixels, waarbij de dichtstbijzijnde pixel het donkerst wordt en de kleinste waarde krijgt en de andere pixel lichter met een grotere waarde.

3.1 Algoritme

De grijswaarden lopen van 0 tot en met 255, waarbij 0 een zwarte pixel is en 255 een witte pixel is. De grijswaarde van een pixel van de lijn wordt in het algoritme berekend met de formule $\lfloor \text{error} \cdot 255 \rceil$, waarbij geldt dat $\lfloor x \rceil$ het gehele getal in $\mathbb{Z}$ is dat het dichtst bij x ligt. Hoe kleiner de error (en hoe dichter de pixel dus bij de lijn), hoe kleiner de grijswaarde en hoe donkerder de pixel dus wordt. Als de error met de pixel boven de lijn groter wordt dan $0,5$, dan schuiven we een pixel op. Bij het algoritme van Bresenham deden we dit *voor* het zwart maken van de pixel, want de error bepaalde welke pixel zwart moest worden. Bij Wu doen we het *na* het kleuren van de pixels. Immers, beide pixels rondom de lijn worden gekleurd bij Wu. Als de error groter wordt dan 0,5, zijn het een kolom verder twee pixels van één rij lager die rondom de lijn liggen. Dus voor de volgende stap verhogen we y_{pix}, maar nog niet voor deze stap. Voor de verschillende gevallen passen we dezelfde veranderingen toe als bij het algoritme van Bresenham in Hoofdstuk 2. In Figuur 7 is het algoritme stap voor stap te zien en in Voorbeeld 3.1 wederom een voorbeeld.

Voorbeeld 3.1. We laten het opnieuw zien met dezelfde lijn als in Voorbeeld 2.1 bij Bresenham. We willen dus weer een lijn tekenen van pixel (1,2) naar (8,6). Het algoritme van Wu gaat precies hetzelfde totdat de error is berekend. Dus in de eerste stap hebben we voor $x = 2$ en $y_{\text{pix}} = 2$:

$$y_{\text{lijn}} = 2,57142857;$$
$$\text{error}_1 = 0,57142857.$$

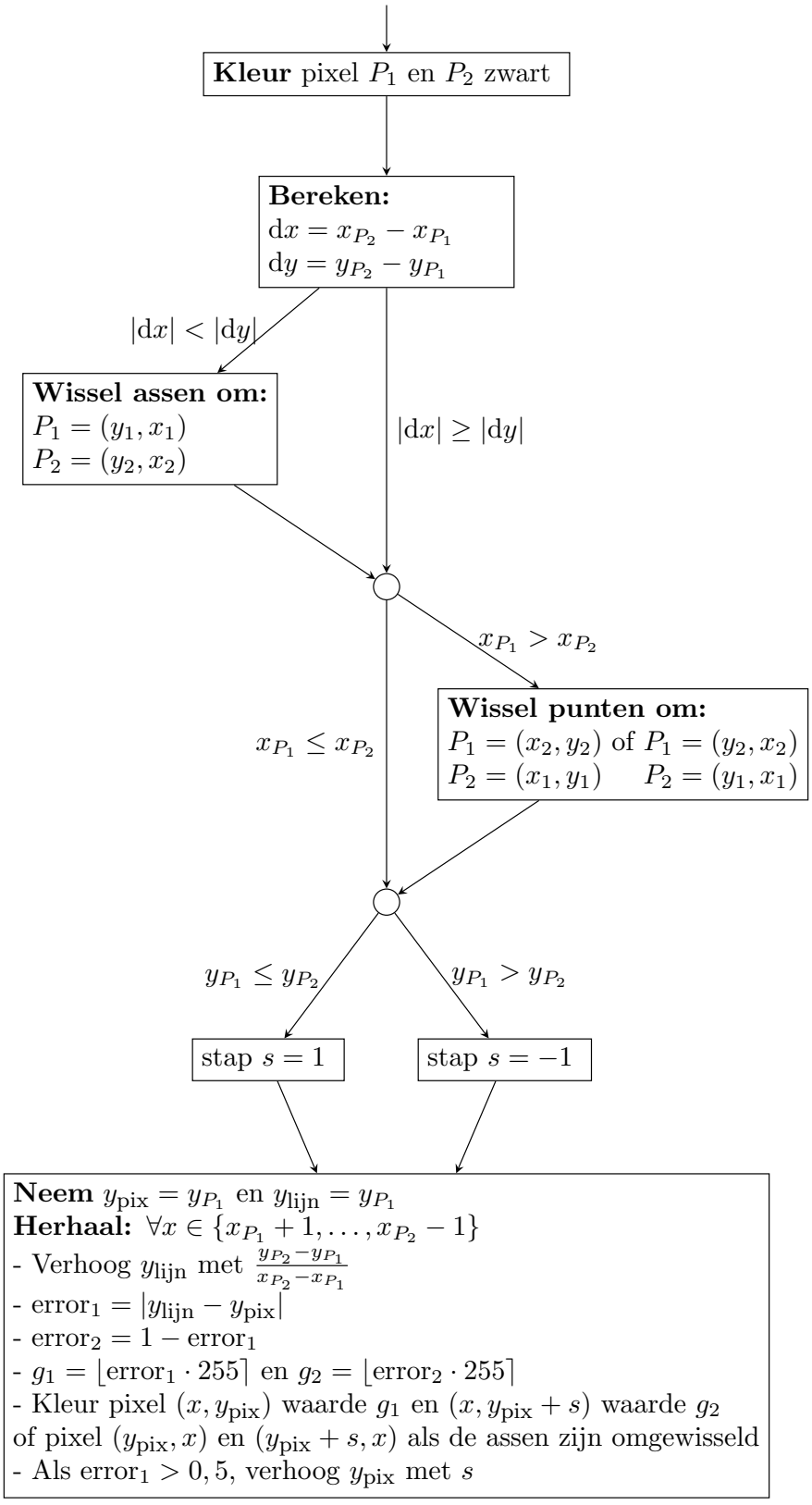
We berekenen nu nog een extra error, namelijk het verschil tussen de lijn en het midden van de pixel eronder. In Figuur 8b is dit het blauwe lijnstuk. Omdat het midden precies 1 onder het midden van de pixel erboven ligt, kunnen we error_2 berekenen met:

$$\text{error}_2 = 1 - \text{error}_1 = 1 - 0,57142857 = 0,42857143.$$

Vervolgens bepalen we de grijswaarden voor de pixels $(2, 2)$ en $(2, 3)$:

$$g_1 = \lfloor \text{error}_1 \cdot 255 \rceil = \lfloor 0,57142857 \cdot 255 \rceil = 146;$$
$$g_2 = \lfloor \text{error}_2 \cdot 255 \rceil = \lfloor 0,42857143 \cdot 255 \rceil = 109.$$

Pixel $(2, 2)$ krijgt grijswaarde 146 en pixel $(2, 3)$ krijgt grijswaarde 109. Dit is te zien in Figuur 8b. Aan de hand van de waarde van error_1 bepalen we of we y_{pix} met 1 moeten verhogen (hoewel we pixel $(2, 3)$ al wel gekleurd hebben, is y_{pix} nog steeds 2). We doen dit op dezelfde manier als

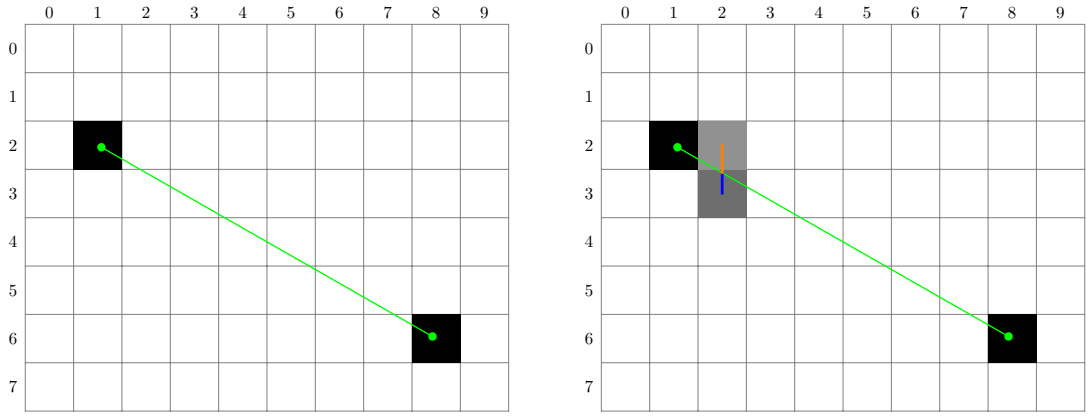


Figuur 7: Het algoritme van Xiaolin Wu

bij Bresenham: $\text{error}_1 > 0,5$, dus in dit geval moet y_{pix} met 1 verhoogd worden. Stap 2 ziet er als volgt uit:

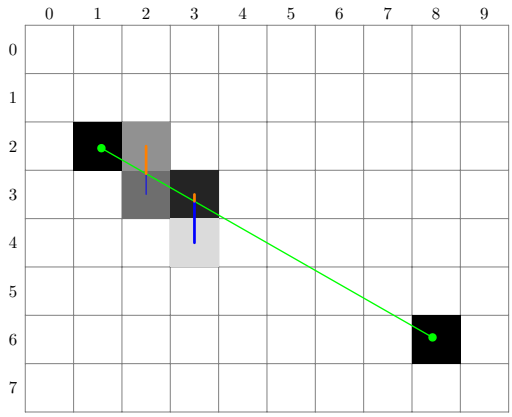
$$x = 3,\ y_{\text{pix}} = 3;$$
$$y_{\text{lijn}} = 2,57142857 + \frac{4}{7} = 3,14285714;$$
$$\text{error}_1 = |3,14285714 - 3| = 0,14285714;$$
$$\text{error}_2 = 1 - 0,14285714 = 0,85714286;$$
$$g_1 = \lfloor 0,14285714 \cdot 255 \rceil = 36;$$
$$g_2 = \lfloor 0,85714286 \cdot 255 \rceil = 219;$$
$$\text{error}_1 < 0,5 \Rightarrow y_{\text{pix}} = 3.$$

In Figuur 8c is te zien dat pixel $(3, 3)$ grijswaarde 36 heeft en pixel $(3, 4)$ heeft grijswaarde 219. Als we deze stap herhalen voor alle $x \in \{2, 3, \ldots, 7\}$ krijgen we het resultaat in Figuur 8d.

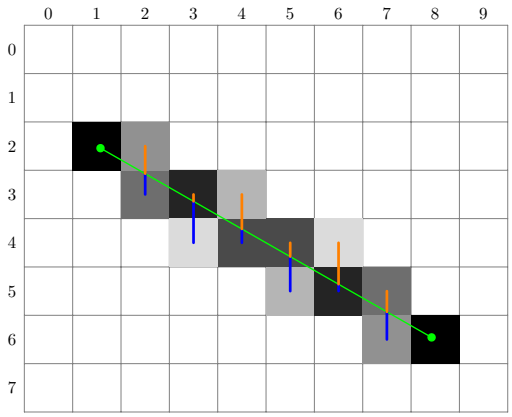


(a) Beginsituatie

(b) Eerste stap



(c) Tweede stap



(d) Lijn met pixels

Figuur 8: Bepalen van de pixels van de lijn met behulp van Xiaolin Wu

3.2 Implementatie

Hieronder is de implementatie van het algoritme van Wu te zien. Als output geeft deze functie drie lijsten, twee met de x- en y-coördinaten van de pixels die gekleurd moeten worden, en eentje met de bijbehorende error_1 en error_2. We kiezen ervoor om niet meteen de grijswaarde te berekenen en die als output mee te geven. Voor de algoritmen van Vrellis en Birsak is het van belang om specifiek de errors te weten in plaats van de grijswaarden. In de volgende hoofdstukken wordt duidelijk waarom.

```
# this functions returns three lists:
# - xpix with x-coordinates of the colored pixels
# - ypix with y-coordinates of the colored pixels
# - errors with the errors between the pixels and the line

def Wu(pin1, pin2):
    (x1, y1), (x2, y2) = pin1, pin2
    xpix = [x1, x2]
    ypix = [y1, y2]
    errors = [0, 0]

    switch = abs(x2-x1) < abs(y2-y1)
    if switch:
        x1, y1 = y1, x1
        x2, y2 = y2, x2

    if x1 > x2:
        x1, x2 = x2, x1
        y1, y2 = y2, y1

    s = 1 if y1 <= y2 else -1

    y_pix = y_lijn = y1
    slope = (y2-y1)/(x2-x1)
    for x in range(x1+1, x2):
        y_lijn += slope
        error1 = abs(y_lijn - y_pix)
        error2 = 1 - error1
        if switch:
            xpix += [y_pix, y_pix + s]
            ypix += [x, x]
        else:
            xpix += [x, x]
            ypix += [y_pix, y_pix + s]
        errors += [error1, error2]
        if error1 > 0.5:
            y_pix += s

    return ypix, xpix, errors
```

Hoofdstuk 4

Algoritme van Petros Vrellis

In dit hoofdstuk bespreken we het algoritme van Petros Vrellis [8]. Het is een *greedy* algoritme waarbij in elke stap de beste lijn op dat moment wordt gekozen. De input van het algoritme is een afbeelding, vaak een portretfoto. Portretten lijken het best te werken door de verdeling van schaduw en licht in het gezicht. Het algoritme is in drie fasen te verdelen en bespreken we hieronder:

1. Bewerken van de afbeelding;
2. Bepalen van de spijkers;
3. Bepalen van de lijnen.

4.1 Bewerken van de afbeelding

De input van het algoritme is een afbeelding waarvan de *string art* wordt gemaakt. De *string arts* zijn altijd cirkelvormig, dus uit deze afbeelding moet ten eerste een cirkelvorm worden gesneden. Bij portretfoto's valt het gezicht meestal in het midden van de foto, maar soms is het handig om handmatig de cirkel in de afbeelding aan te passen.

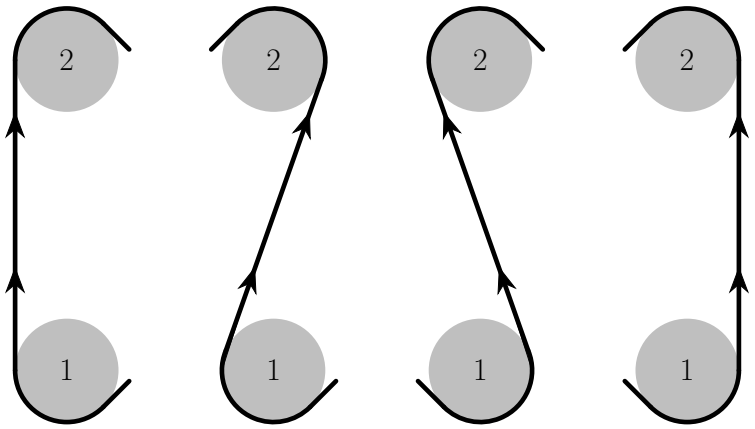
Verder moet de foto worden aangepast naar grijswaarden. Voor dit algoritme zijn alleen de grijswaarden van de pixels belangrijk en niet de kleuren. Zoals eerder gezegd hebben pixels met grijswaarden een waarde tussen 0 en 255, waarbij 0 een volledig zwarte pixel is en 255 een witte pixel is. In dit algoritme inverteren we de grijswaarden, een grijswaarde g wordt nu $255 - g$. Nu geldt: hoe donkerder een pixel, hoe groter de grijswaarde. Dit werkt intuïtiever en zorgt ervoor dat we later in het algoritme op zoek gaan naar een maximum in plaats van een minimum, wat net iets makkelijker is.

4.2 Bepalen van de spijkers

De tweede fase is de positie van de spijkers bepalen. Zoals eerder gezegd zijn de spijkers gelijk verdeeld over de rand van de cirkel. Vrellis gebruikt een aantal spijkers dat tussen de 200 en 300 ligt, afhankelijk van de afbeelding en de grootte van de *string art*. Minder spijkers geeft geen goed resultaat omdat er niet veel verschillende lijnen mogelijk zijn. Meer dan 300 spijkers wordt krap langs de rand van de *string art*, die meestal een diameter tussen 60 cm en 80 cm heeft. Twee spijkers hebben dan ongeveer een centimeter ruimte ertussen. In de implementatie van het algoritme worden de spijkers genummerd en de coördinaten van de spijkers opgeslagen.

We moeten rekening houden met de dikte van de spijker ten opzichte van de draad. Een draad gaat om de spijker heen en kan dit in twee richtingen doen, waardoor er vier verschillende lijnen tussen twee spijkers mogelijk zijn. In Figuur 9 wordt dit geïllustreerd. Als de draad dun genoeg is en de spijker breed genoeg, denk bijvoorbeeld aan een draad van 0,5 mm dik en een spijker van minstens 1,6 mm breed, zullen deze lijnen goed van elkaar te onderscheiden zijn en kunnen we ze beschouwen als afzonderlijke lijnen. Voor elke spijker slaan we twee punten met

coördinaten op, de linkerkant en de rechterkant van de spijker, waarbij we vanuit het middelpunt van de cirkel bepalen wat links en wat rechts is.



Figuur 9: Twee spijkers kunnen op vier verschillende manieren met elkaar verbonden worden. Van elke spijker worden de coördinaten van de linkerkant en de rechterkant van de spijker opgeslagen.

4.3 Bepalen van de lijnen

Als we alle coördinaten van de spijkers hebben opgeslagen, kunnen we door naar de derde fase van het algoritme. Deze fase van het algoritme begint bij een willekeurige spijker en bij een willekeurige kant van deze spijker. Vanuit dit punt wordt elke mogelijke lijn naar de andere punten van de spijkers bekeken. Naar elke spijker zijn er twee mogelijke lijnen: naar de linkerkant van de spijker en naar de rechterkant van de spijker. Omdat het aantal spijkers niet heel groot is, duurt het niet lang om alle lijnen te bekijken. Van elke lijn wordt berekend hoe donker deze lijn is in de input afbeelding. Dit wordt gedaan met behulp van het algoritme van Bresenham (Hoofdstuk 2) of Wu (Hoofdstuk 3). Deze algoritmen bepalen welke pixels van de afbeelding bij de lijn horen. In het geval van Bresenham worden de grijswaarden van deze pixels bij elkaar opgeteld. In het geval van Wu worden de grijswaarden van de pixels met 1 minus de bijbehorende error vermenigvuldigd voordat ze bij elkaar worden opgeteld. De error geeft aan hoe dicht een pixel bij een lijn ligt. Afhankelijk daarvan wordt een bepaald percentage van de grijswaarde meegerekend voor de som, in plaats van de hele grijswaarde. Hoe dichter de pixel bij de lijn, hoe groter we dit percentage willen hebben. Tegelijkertijd is de error juist kleiner als de pixel dichter bij de lijn zit, daarom trekken we de error van 1 af en vermenigvuldigen we het dan met de grijswaarde.

Het algoritme kiest de donkerste lijn. Omdat we de grijswaarden geïnverteerd hebben, is dit de lijn met de grootste som van grijswaarden. Dit wordt de eerste lijn van de *string art*. De spijker aan de andere kant van de lijn wordt in een lijst opgeslagen, evenals of de lijn naar de linker- of rechterkant gaat. Daarnaast wordt de lijn die gekozen is in de portretfoto lichter gemaakt. Dit is nodig, want als we niks zouden doen met de lijn in de portretfoto, blijft het algoritme dezelfde lijnen kiezen. Bovendien is deze lijn donkerder geworden in onze *string art*. We hebben de donkerte van de portretfoto als het ware verschoven naar de *string art*. Het doel van het algoritme is om dit zo precies mogelijk te doen met de hele portretfoto, zodat de *string art* een goede benadering van de foto wordt. Het lichter maken van de lijn in de portretfoto gebeurt door de grijswaarden van de pixels van de lijn aan te passen. Het meest voor de hand liggend is om de pixels helemaal wit te maken, aangezien we een lijn trekken in onze *string art* die helemaal zwart is. Toch blijkt dit niet te werken. Het blijkt dat we een bepaalde constante eraf moeten trekken, waarna de pixels nog aardig donker blijven. Bovendien doen we dit niet alleen voor de pixels van de lijn, maar ook voor pixels om de lijn heen. In het hoofdstuk over de implementatie en de resultaten, Hoofdstuk 5, vertellen we hier meer over.

Als de bovenstaande dingen zijn uitgevoerd, gaat het algoritme opnieuw op zoek naar de

donkerste lijn. Deze keer vanuit een andere spijker, namelijk de spijker die we zojuist hebben opgeslagen. We zijn dus bij een nieuwe spijker aangekomen en herhalen de bovenstaande stap. Het algoritme blijft dit doen, totdat er geen donkere lijnen meer te vinden zijn. We zijn dan bij een spijker aangekomen en vanuit deze spijker zijn alle mogelijke lijnen al helemaal wit geworden. De som van elke lijn is 0 en het algoritme eindigt.

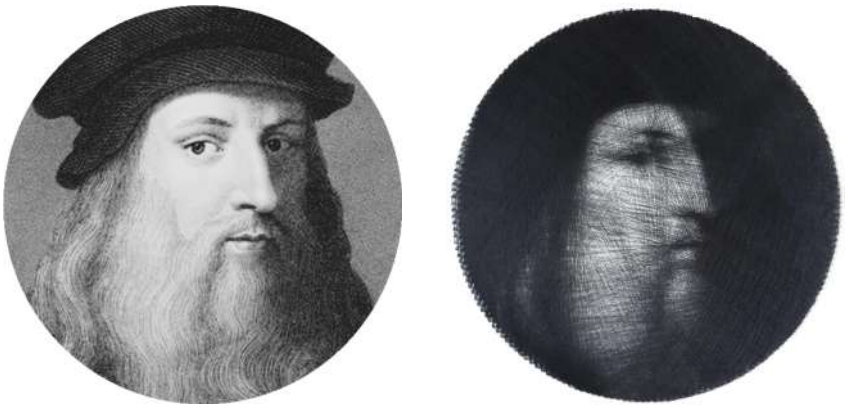
4.4 Output

De output van het algoritme is de lijst met spijkers die is opgeslagen. Deze lijst geeft alle spijkers weer, in de volgorde waarin ze met elkaar moeten worden verbonden met de draad. Bovendien geeft het voor elke spijker weer of de draad naar de linker- of rechterkant van de spijker gaat. Dat wordt weergegeven met een 0 of een 1, waarbij een 0 links betekent en een 1 rechts. Stel, het begin van de lijst ziet er als volgt uit: $\{(0,0),(54,0),(209,1),(29,0),(31,1),\dots\}$ Dit betekent dat de draad bij spijker 0 begint, en vervolgens naar de linkerkant van spijker 54 gaat en de spijker aan de rechterkant weer verlaat. Vanaf spijker 54 gaat de draad naar 209 en hier gaat de draad naar rechts, vanaf daar naar spijker 29 en links, etc.

Hoofdstuk 5

Implementatie en resultaten Vrellis

In dit hoofdstuk bespreken we de implementatie en de resultaten van ons eigen geïmplementeerde algoritme. We hebben het algoritme geïmplementeerd in Python [7]. Om te beginnen laten we een foto zien van een *string art* die we hebben gemaakt aan de hand van het algoritme. We hebben als input een portrettekening van Leonardo da Vinci gebruikt. Met behulp van de lijst met spijkers hebben we de *string art* in Figuur 10b gemaakt. Het portret is een beetje donker uitgevallen, waarschijnlijk omdat de dikte van de gebruikte draad niet helemaal overeenkomt met de instellingen van het algoritme (hierover later meer). Toch is Leonardo goed te herkennen. Dit is een *string art* met 180 spijkers, een diameter van 610 mm en spijkers van 1,6 mm dik. De dikte van de draad is onbekend, maar zal tussen 0,3 en 0,5 mm liggen. Om een nog completer beeld te geven: er zit bijna 4 km draad in deze *string art* verwerkt en het bevat meer dan 7000 lijnen.



(a) Input (b) *String art*

Figuur 10: Een *string art* gemaakt met het algoritme van Vrellis, aan de hand van een portrettekening van Leonardo da Vinci

De rest van de *string arts* in dit hoofdstuk zijn, in tegenstelling tot die van Leonardo, simulaties. Het zijn digitale afbeeldingen van hoe de *string arts* er in het echt uit zouden zien. Op die manier kunnen we de resultaten gemakkelijker bekijken, zonder dat we de *string arts* elke keer fysiek moeten maken. Om ervoor te zorgen dat deze digitale afbeeldingen de echte *string arts* goed genoeg simuleren, gebruiken we bij de afmetingen van de simulaties de volgende formule [1]:

$$Z = \frac{1}{t} \cdot d,$$

waarbij Z de diameter van de simulatie in pixels is, t is de dikte van de draad in mm en d is de diameter van de *string art* in mm. In de simulatie tekenen we lijnen van 1 pixel dik. Deze

formule zorgt ervoor dat die dikte ten opzichte van de afmetingen van de simulatie overeenkomt met de dikte van de draad ten opzichte van de afmetingen van de *string art*. In de onderstaande resultaten hebben we telkens een diameter van 4096 pixels gebruikt in de simulatie (tenzij anders aangegeven). Dit komt overeen met een *string art* met een diameter van 614,4 mm en een draad van 0,15 mm.

We gaan in dit hoofdstuk verschillende variabelen van het algoritme bespreken en bekijken hoe ze invloed hebben op het resultaat. We zullen geen perfecte waarden voor de variabelen geven. Het verschilt namelijk per portretfoto bij welke variabele de kwaliteit van de *string art* het "beste" is. Bovendien kunnen we ons afvragen wanneer een *string art* een goede kwaliteit heeft, daarom zetten we "beste" tussen aanhalingstekens. Het algoritme gaat op zoek naar de beste benadering van een portretfoto, het zet een portretfoto zo goed mogelijk om naar lijnen. Om iets te kunnen zeggen over hoe goed een *string art* een portretfoto benadert, kunnen we de pixels van de simulatie van de *string art* en de portretfoto vergelijken. Dat doen we in het stukje code hieronder, we berekenen het verschil tussen de grijswaarden en zetten het om naar een percentage. Merk op dat we hier de simulatie met de portretfoto vergelijken en dus niet de *string art* zelf.

```
import numpy as np

def approx(img1, img2):
    height, width = img1.shape[:2]
    total = height*width*255
    diff = np.sum(abs(img1-img2).astype(int))
    approx = (1-diff/total)*100

    return approx
```

Deze percentages zijn een manier om een *string art* te beoordelen. Maar als we deze manier toepassen op onze simulaties, blijkt al snel dat het niet een hele goede manier is. De verschillende simulaties van wiskundige John Nash in Figuur 11 komen allemaal rond de 50% en 51% overeen met de portretfoto. Toch zijn met name de simulaties in Figuur 11b, 11c en 11d niet herkenbaar als John Nash. We kunnen dus misschien de *string arts* beter beoordelen naar *herkenbaarheid* in plaats van de *overeenkomst* met een portretfoto. Maar herkenbaarheid is natuurlijk subjectief, net zoals de schoonheid of de diepzinnigheid van een portret. We zullen daarom in dit hoofdstuk de waarden van de variabelen variëren en laten zien hoe de *string arts* veranderen, maar geen directe conclusies trekken.

5.1 Resolutie van de input

De eerste stap van het algoritme is het bewerken van de afbeelding. Om met afbeeldingen te werken gebruiken we de software van OpenCV [2]. Zoals al genoemd in Hoofdstuk 4 zetten we de eventuele kleuren van de afbeelding om in grijswaarden. Bovendien inverteren we deze grijswaarden, omdat dit later in het algoritme handiger is. Hieronder is te zien welke functies van OpenCV we gebruiken om dit te doen.

```
import cv2  # we use openCV to process images

img = cv2.imread(portrait)
img_grayscales = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
img_inverted = cv2.bitwise_not(image)
```

Verder hebben we de vorm van een cirkel nodig voor het algoritme. We zetten een wit masker in deze vorm over de foto heen, waardoor alle delen van de foto buiten de cirkel wit worden.

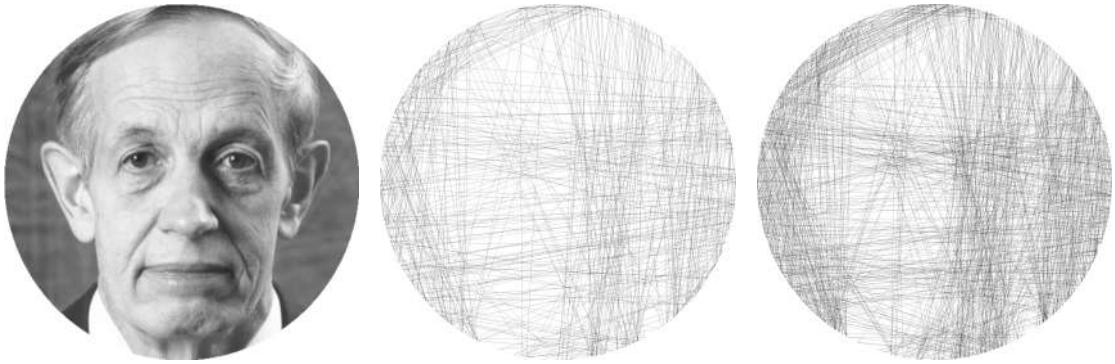
```
def maskImage(image, center, radius):
    height, width = image.shape[:2]
    mask = np.zeros((height, width), dtype = "uint8")
    cv2.circle(mask, center, radius, 255, -1)
    img_masked = cv2.bitwise_and(image, image, mask=mask)

    return img_masked
```

Nadat we dit masker erop hebben gezet, kunnen we de overbodige delen van de afbeelding eraf knippen. We houden een vierkante afbeelding over, met in het midden een cirkel waarin

het portret te zien is. Verder veranderen we eventueel de resolutie van de foto. Het blijkt dat de resolutie van grote invloed is op het eindresultaat. Hoe kleiner de resolutie van de foto, hoe lichter de *string art* wordt. Dat is eenvoudig te verklaren: de foto heeft weinig pixels, dus met het lichter maken van een lijn, wordt een relatief groot deel van de foto lichter gemaakt. Het algoritme is hierdoor snel klaar en heeft weinig lijnen toegevoegd aan de *string art*. Hieronder is te zien hoe we de afbeelding bijsnijden en de resolutie veranderen naar een `width` naar keuze. In Figuur 11 zijn verschillende *string arts* te zien van een portretfoto van de wiskundige John Nash. Bij elke *string art* hebben we een andere resolutie van de portretfoto gebruikt.

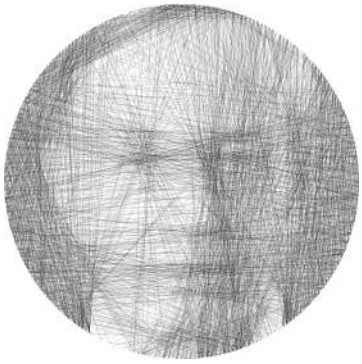
```
img_cut = img_masked[c2-radius : c2+radius, c1-radius : c1+radius]
img_resized = cv2.resize(img_cut, (width, width))
```



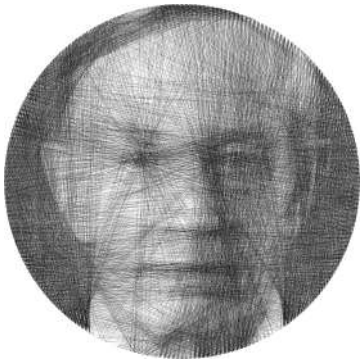
(a) Portretfoto van John Nash

(b) Resolutie foto: 256 × 256

(c) Resolutie foto: 512 × 512



(d) Resolutie foto: 1024 × 1024



(e) Resolutie: 2048 × 2048

(f) Resolutie: 4096 × 4096

Figuur 11: Verschillende *string arts* bij een portretfoto van John Nash

5.2 Aantal spijkers

De tweede stap van het algoritme is de positie van de spijkers bepalen. We slaan de coördinaten van de spijkers op in een lijst. Zoals besproken in het vorige hoofdstuk houden we rekening met de dikte van de spijkers, `width_pin`. We slaan daarom telkens de linkerkant van de spijker op, `pin_left`, en de rechterkant van de spijker, `pin_right`, gezien vanaf het middelpunt van de cirkel. We berekenen de coördinaten met behulp van de cosinus en de sinus en de hoeken in radialen. We gebruiken daarbij de straal `r` maar rekenen met `r-1`, omdat de coördinaten anders buiten de afbeelding kunnen vallen. Hieronder is de implementatie te zien.

```
import math

def pinCoords(nPins, center, r, width_pin):
    (c1, c2) = center
    angles = np.linspace(0, 2*np.pi, nPins+1)
    b = 2*math.asin(width_pin/(2*r))
    pins = []
    for a in angles[0:-1]:
        left_x = int(c1+(r-1)*np.cos(a))
        left_y = int(c2+(r-1)*np.sin(a))

        right_x = int(c1+(r-1)*np.cos(a+b))
        right_y = int(c2+(r-1)*np.sin(a+b))

        pins += [(left_x, left_y), (right_x, right_y)]

    return pins
```

Het aantal spijkers, `nPins`, is een variabele die invloed heeft op het resultaat. Hoe meer spijkers, hoe accurater de *string art*. Er zit natuurlijk een maximum aan het aantal spijkers, afhankelijk van de grootte van de *string art*. In Figuur 12 zijn de verschillende *string arts* te zien bij `nPins=64`, `nPins=128` en `nPins=256`. Bij elke simulatie gebruiken we een input foto met een resolutie van 4096 bij 4096 pixels.

(a) $p = 64$ (b) $p = 128$ (c) $p = 256$

Figuur 12: Verschillende *string arts* van John Nash

5.3 Bresenham en Wu

Uiteindelijk komen we aan bij het belangrijkste gedeelte van het algoritme, het bepalen van de lijnen. We berekenen de donkerte onder elke lijn en voegen de donkerste lijn toe aan de *string art*. Hiervoor gebruiken we de implementaties van de algoritmen van Bresenham en Wu uit Hoofdstuk 2 en 3. Hieronder is de implementatie van het proces te zien als we het algoritme van Bresenham gebruiken.

```
def findLine(image, currentPin, pins):
    maximum = 0
    bestPin = None
    for pin in pins:
        ypix, xpix = Bresenham(currentPin, pin)
        sum = np.sum(image[ypix, xpix])
        if sum > maximum:
            maximum = sum
            bestPin = pin

    return bestPin
```

De implementatie van Wu geeft naast de lijsten `ypix` en `xpix` ook een lijst met errors, `errors`, terug. De errors geven aan hoe ver de pixels van de lijn liggen. We tellen niet de hele grijswaarden van de pixels bij elkaar op, maar bepaalde percentages van de grijswaarden. Dat berekenen we

met behulp van de error: we tellen telkens `(1-error)*image[y,x]` bij de som op, voor een pixel (x, y).

```
def findLine(image, currentPin, pins):
    maximum = 0
    bestPin = None
    for pin in pins:
        ypix, xpix, errors = Wu(currentPin, pin)
        errors = np.array([errors])
        sum = np.sum((image[ypix, xpix]*(1-errors)).astype(float))
        if sum > maximum:
            maximum = sum
            bestPin = pin

    return bestPin
```

In Figuur 13 staan twee simulaties van Leonardo da Vinci. Figuur 13a is gemaakt met het algoritme van Bresenham, Figuur 13b is gemaakt met het algoritme van Wu. Beide simulaties zijn gemaakt met een input afbeelding van 4096 bij 4096 pixels en met 256 spijkers. We zien eigenlijk bijna geen tot geen verschil. Het algoritme van Bresenham werkt iets sneller dan dat van Wu, want er worden dan minder berekeningen gemaakt. Daarom gebruiken we in de rest van dit hoofdstuk steeds het algoritme van Bresenham.

(a) Leonardo da Vinci met Bresenham (b) Leonardo da Vinci met Wu

Figuur 13: Twee *string arts* met verschillende lijnalgoritmen

5.4 Lijn aftrekken van de input

Als we hebben bepaald wat de donkerste lijn is, voegen we `bestPin` toe aan de lijst van spijkers die de output van het algoritme wordt. Verder moet de lijn die gekozen is in de portretfoto lichter gemaakt worden. Dat doen we met behulp van de onderstaande functie. We maken een tijdelijke afbeelding met alleen maar witte pixels, waarin we de betreffende lijn trekken. Met de `subtract` functie van `numpy` kunnen we deze afbeelding van de portretfoto aftrekken.

```
def subtractLine(image, pin1, pin2, darkness, width_line):
    temp = np.zeros((image.shape))
    cv2.line(temp, pin1, pin2, darkness, width_line)
    image = np.subtract(image, temp)

    return image
```

In deze functie zitten twee variabelen die we kunnen variëren en die invloed hebben op het resultaat. Ten eerste de donkerte van de lijn, `darkness`, die we van de portretfoto aftrekken. Hoe donkerder deze lijn, hoe lichter de portretfoto wordt. Het algoritme duurt dan korter en de *string art* heeft minder lijnen. Daarnaast kunnen we de dikte van de lijn, `width_line`, variëren. Het lijkt voor de hand te liggen dat we gewoon een dikte van 1 pixel moeten nemen. Dit komt immers overeen met de dikte van de draad. Toch blijkt dit niet voor de beste resultaten te zorgen. De *string art* wordt erg donker bij een dikte van 1. Het lijkt beter te werken om een

dikte van een paar pixels meer te nemen, de *string art* wordt zo iets lichter en de afbeelding is beter zichtbaar. In Figuur 15 zijn de verschillende simulaties te zien, waarbij de resolutie van de input steeds 4096 bij 4096 pixels is en het aantal spijkers 256.

5.5 Dikte van de draad

Een laatste variabele die invloed heeft op het resultaat is de dikte van de draad. Het spreekt voor zich: hoe dunner de draad, hoe gedetailleerder het resultaat kan worden. In Figuur 14 zijn verschillende resultaten te zien bij een foto van Einstein. De dikte van de draad varieert tussen 0,15 mm en 1,2 mm. De resolutie van de input is 4096 bij 4096 pixels, het aantal spijkers is 256 en we hebben een donkerte van 15 en een dikte van 3 gebruikt in de `subtractLine` functie.

In de simulatie *string arts* kunnen we de dikte van de draad simuleren door de resolutie van de simulatie aan te passen, volgens de formule $Z = \frac{1}{t} \cdot d$. We hebben de diameter op 614,4 mm gehouden en Z en t veranderd om de verschillende diktes te kunnen bekijken. Hieronder is te zien hoe we de simulatie hebben gemaakt.

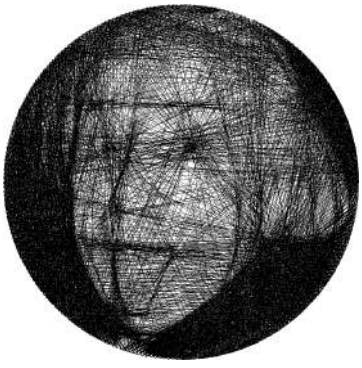
```
# we start with a white image
height, width = Z
simulation = np.ones((height, width), dtype = "uint8")*255

# every round we update the simulation by drawing a black line
cv2.line(simulation, currentPin, bestPin, 0, 1)
```

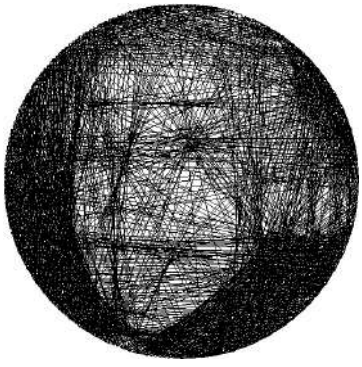
(a) Foto van Einstein

(b) Dikte draad: 0,15mm

(c) Dikte draad: 0,3 mm

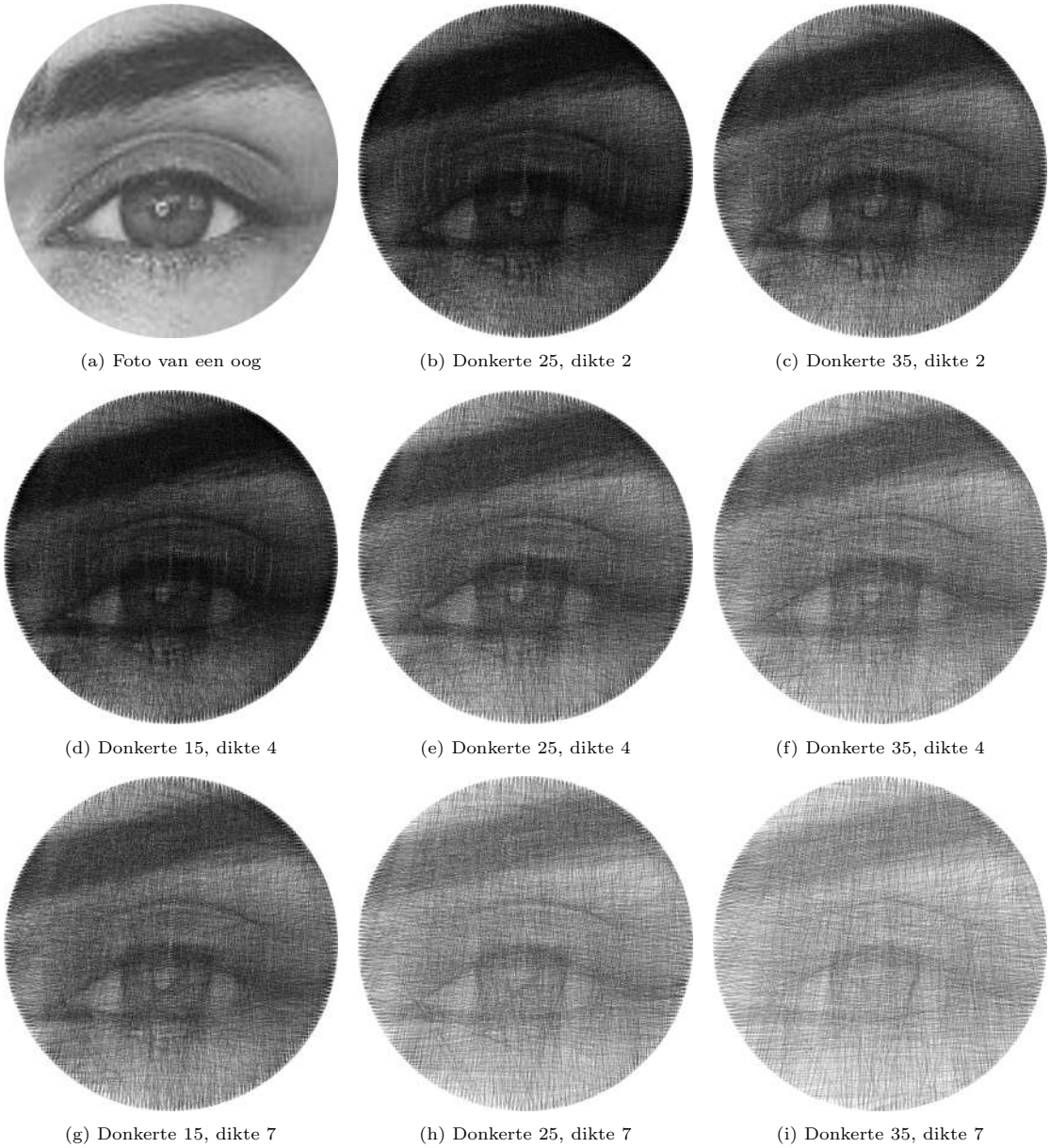


(d) Dikte draad: 0,6 mm



(e) Dikte draad: 1,2 mm

Figuur 14: Verschillende *string arts* bij een foto van Einstein



(a) Foto van een oog (b) Donkerte 25, dikte 2 (c) Donkerte 35, dikte 2

(d) Donkerte 15, dikte 4 (e) Donkerte 25, dikte 4 (f) Donkerte 35, dikte 4

(g) Donkerte 15, dikte 7 (h) Donkerte 25, dikte 7 (i) Donkerte 35, dikte 7

Figuur 15: Verschillende *string arts* bij een foto van een oog

Hoofdstuk 6

Algoritme van Michael Birsak

In dit hoofdstuk bespreken we een andere manier om een algoritme te maken voor *string arts*, beschreven door onder andere Michael Birsak [1]. Bij het algoritme van Vrellis worden telkens alleen de lijnen vanuit één bepaalde spijker bekeken. Hierdoor is de keuze erg beperkt. Het algoritme van Birsak doet dit anders en is daarom de moeite waard om naast het algoritme van Vrellis ook te onderzoeken.

Een aantal dingen zijn hetzelfde: ook in het algoritme van Birsak wordt er een cirkelvormige *string art* gemaakt, met de spijkers verdeeld over de rand van de cirkel. Als input neemt het algoritme een portretfoto, als output geeft het algoritme een lijst van spijkers die met elkaar verbonden moeten worden. De eerste twee fasen, het bewerken van de input afbeelding en het bepalen van de positie van de spijkers, zijn hetzelfde, met uitzondering van het inverteren van de grijswaarden. Dat is voor dit algoritme niet nodig. In de derde fase, het bepalen van de lijnen, wijkt dit algoritme verder af. Waar Vrellis vanuit één spijker de mogelijke lijnen bekijkt, bekijkt Birsak álle mogelijke lijnen, precies de reden waarom het algoritme van Birsak wellicht een resultaat geeft dat de foto nóg beter benadert. Verder gebruikt Birsak de euclidische norm van een vector om voor een bepaalde lijn te kiezen in plaats van de donkerte van een lijn. Omdat Birsak eerst de beste combinatie van lijnen kiest, moet hij daarna nog bepalen in welke volgorde hij de lijnen met elkaar verbindt. Dit is de vierde fase van het algoritme, het bepalen van een Eulercykel.

6.1 *String arts* als vectoren

Voor het bepalen van de lijnen maakt dit algoritme gebruik van de euclidische norm van een vector. Het algoritme geeft de input afbeelding en een mogelijke *string art* weer als vectoren en berekent vervolgens de norm van het verschil van deze twee vectoren. Door deze norm te minimaliseren wordt een zo goed mogelijke *string art* gemaakt. Het algoritme kiest dus telkens de lijn die de norm van het verschil het meest vermindert. Maar voordat we dat gaan doen, leggen we uit hoe we de input afbeelding en de *string art* weergeven als vectoren. In Hoofdstuk 2 is uitgelegd dat we digitale afbeeldingen zoals de input afbeelding kunnen zien als een matrix. Om er een vector van te maken zetten we simpelweg alle rijen achter elkaar. De input afbeelding kunnen we dan als volgt weergeven:

$$y \in [0, 255]^m \subseteq \mathbb{N}^m,$$

waarbij m het aantal pixels van de afbeelding is.

Het weergeven van een *string art* als vector is iets ingewikkelder, want de vector hangt af van de gekozen lijnen voor de *string art*. Een mogelijke keuze van een combinatie van lijnen geven we vanaf nu weer met

$$x \in \{0, 1\}^n,$$

waarbij n het totaal aantal mogelijke lijnen is. We kunnen n berekenen met:

$$n = 4 \cdot \binom{p}{2},$$

waarbij p het aantal spijkers is. De formule is niet moeilijk te begrijpen: voor elke 2 spijkers uit het totale aantal spijkers geldt dat ze op vier verschillende manieren met elkaar verbonden kunnen worden (Figuur 9). De vector x is een binaire vector, een 1 betekent dat de lijn gekozen wordt voor de *string art* en een 0 betekent dat de lijn niet gekozen wordt. In dit algoritme kan elke lijn dus maar één keer getrokken worden in de *string art*.

Het opstellen van een *string art* vector bij een combinatie x van lijnen kunnen we zien als een functie:

$$F : \{0,1\}^n \to \mathbb{N}^m$$
$$x \mapsto F(x) \in [0,255]^m.$$

De *string art* vector geeft op elke plek de grijswaarde van de pixel in de *string art* weer. De grijswaarde van een pixel hangt naast de combinatie van lijnen ook af van het gekozen lijnalgoritme. We kunnen het algoritme van Bresenham uit Hoofdstuk 2 of het algoritme van Wu uit Hoofdstuk 3 gebruiken. In die twee hoofdstukken hebben we laten zien hoe we de grijswaarde van een pixel van een lijn bepalen. Nu willen we weten wat de grijswaarde van een pixel wordt als we een heleboel lijnen tekenen, waarvan sommigen misschien over de pixel lopen. We laten voor beide algoritmen apart zien wat de waarde van $F(x)_i$ voor een willekeurige $i \in \{1,2,\ldots,m\}$ en een willekeurige $x \in \{0,1\}^n$ wordt.

Bij het algoritme van Bresenham geldt dat een pixel zwart of wit is. De mogelijke grijswaarden voor een pixel in een *string art* vector zijn dus 0 en 255. Een pixel is wit in de *string art* vector als er geen enkele lijn over de pixel heen loopt. Alle lijnen die gekozen zijn voor de *string art* lopen dus niet in de buurt van de pixel en de pixel blijft wit. Een pixel is zwart in de *string art* vector als er ten minste één lijn overheen loopt. Het maakt niet uit of er nog andere lijnen over de pixel heen lopen, want de pixel kan niet donkerder dan zwart worden. We krijgen dus het volgende onderscheid:

$$F(x)_i = \begin{cases} 255 & \text{Als er geen enkele lijn over pixel } i \text{ loopt} \\ 0 & \text{Als er ten minste 1 lijn over pixel } i \text{ loopt.} \end{cases}$$

Voor het algoritme van Wu is het iets ingewikkelder. Hierbij maakt het wel uit hoeveel lijnen er precies over de pixel lopen. Hoe meer lijnen er over een pixel lopen, hoe donkerder de pixel wordt. Als er één lijn over de pixel loopt, kunnen we de grijswaarde berekenen met $\lfloor \text{error}_{ij} \cdot 255 \rceil$, waarbij error_{ij} de error tussen pixel i en lijn j voorstelt zoals we die in Hoofdstuk 3 berekenden. We spreken af dat error_{ij} de waarde 1 krijgt als de pixel niet gekleurd wordt bij een lijn, de pixel ligt te ver van de lijn af en blijft wit met grijswaarde 255.

Als er meerdere lijnen over een pixel lopen, wordt de uiteindelijke grijswaarde van de pixel bepaald door de grijswaarden bij de afzonderlijke lijnen. Om goed te begrijpen hoe die afzonderlijke grijswaarden samenwerken, inverteren we de grijswaarden tijdelijk: een grijswaarde g wordt nu $255 - g$. Nu geldt: hoe hoger de grijswaarde, hoe donkerder de pixel. We berekenen een afzonderlijke grijswaarde nu met $\lfloor 255 - \text{error}_{ij} \cdot 255 \rceil$. Merk op dat we pas aan het eind afronden, om een zo nauwkeurig mogelijk resultaat te krijgen. Het is nu intuïtief goed te begrijpen dat we de uiteindelijke grijswaarden van een pixel kunnen berekenen door de afzonderlijke grijswaarden bij elkaar op te tellen. Stel, pixel i ligt onder 3 gekozen lijnen van de *string art*, en krijgt bij de 3 afzonderlijke lijnen de grijswaarden 60, 75 en 38. De uiteindelijke grijswaarde van i wordt dan $60 + 75 + 38 = 173$. Het maximum is 255, de pixel is dan zwart en kan niet nog donkerder worden. Als de som boven 255 uitkomt, kiezen we dus als grijswaarde 255. Om de grijswaarde van i te berekenen, sommeren we dus over alle lijnen die over i lopen. We kunnen ook sommeren over alle lijnen van de *string art*, dus alle j waarvoor geldt dat $x_j = 1$, aangezien de afzonderlijke grijswaarde 0 is als een lijn niet over de pixel loopt (want de error is 1). We zeggen dat $F_{\text{inv}}(x)_i$ de waarde is van $F(x)_i$ als de grijswaarden geïnverteerd zijn. Verder ronden we opnieuw pas aan het eind af. Er geldt:

$$F_{\text{inv}}(x)_i = \max(255, \lfloor \sum_{\substack{j \in \{1,\ldots,n\}: \\ x_j = 1}} 255 - \text{error}_{ij} \cdot 255 \rceil)$$
$$= \max(255, \lfloor 255 \cdot \sum_{\substack{j \in \{1,\ldots,n\}: \\ x_j = 1}} 1 - \text{error}_{ij} \rceil).$$

We moeten niet vergeten om nog een keer te inverteren om weer bij de oude grijswaarden uit te komen. Een grijswaarde g wordt weer $255 - g$. Dus dan krijgen we:

$$F(x)_i = \max(0, \lfloor 255 - 255 \cdot \sum_{\substack{j \in \{1,\ldots,n\} \\ x_j = 1}} 1 - \text{error}_{ij} \rceil).$$

Ten slotte noemen we $\sum_{j \in \{1,\ldots,n\}: x_j = 1} 1 - \text{error}_{ij}$ voor het gemak vanaf nu $s(x)_i$. We kunnen $F(x)_i$ dan als volgt noteren:

$$F(x)_i = \begin{cases} 0 & \text{Als } s(x)_i > 1 \\ \lfloor 255 - 255 \cdot s(x)_i \rceil & \text{Anders.} \end{cases}$$

6.1.1 Alternatieve notatie

Nu hebben we voor het algoritme van Bresenham en voor het algoritme van Wu twee uitkomsten voor de grijswaarde van een pixel i van de *string art* vector. Maar met name de uitkomst bij het algoritme van Wu is niet een handige om mee te werken. Daarom gaan we op zoek naar een manier om de bovenstaande uitkomsten te bundelen. Hiervoor introduceren we een matrix $A \in \mathbb{R}^{m \times n}$.

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}.$$

Als we het lijnalgoritme van Wu gebruiken nemen we voor a_{ij} de waarde $1 - \text{error}_{ij}$. Als we het lijnalgoritme van Bresenham gebruiken definiëren we a_{ij} als volgt:

$$a_{ij} = \begin{cases} 1 & \text{Als pixel } i \text{ onder lijn } j \\ 0 & \text{Als pixel } i \text{ niet onder lijn } j. \end{cases}$$

De vector $A \cdot x$ ligt al in de buurt van $F(x)$. Maar nog niet alle waarden zijn gehele getallen en de grijswaarden moeten weer geïnverteerd worden en ten slotte kunnen de waarden nu boven 255 liggen. Daarom introduceren we een functie C die de waarden inverteert en afrondt en ervoor zorgt dat de waarden maximaal 255 zijn:

$$C : \mathbb{R}^m \to [0, 255]^m$$
$$(u)_{i=1}^m \mapsto [\lfloor (1 - \min(1, u_i)) \cdot 255 \rceil]_{i=1}^m.$$

We bekijken de vector $C(A \cdot x)$. We gaan nu laten zien dat dit precies de *string art* vector is die bij een combinatie x van lijnen hoort, dus dat $F(x) = C(Ax)$. We doen dit opnieuw apart voor beide lijnalgoritmen.

Claim: Bij gebruik van Bresenham geldt: $\forall x \in \{0, 1\}^n : F(x) = C(Ax)$.
Bewijs: We weten dat voor een willekeurige $i \in \{1, \ldots, m\}$ geldt:

$$F(x)_i = \begin{cases} 255 & \text{Als er geen enkele lijn over pixel } i \text{ loopt} \\ 0 & \text{Als er ten minste 1 lijn over pixel } i \text{ loopt.} \end{cases}$$

We bekijken wat er in beide gevallen met $C(Ax)_i$ gebeurt. We maken de onderstaande berekening voor het eerste geval. Belangrijk voor deze berekening is om te bedenken dat in $a_{i1}x_1 + \cdots + a_{in}x_n$ elke term 0 is. Voor elke j geldt dat $x_j = 0$ of $x_j = 1$. Als $x_j = 0$, dan is de term $a_{ij}x_j$ uiteraard ook 0. Als $x_j = 1$, dan betekent dit dat lijn j gekozen is voor de *string art*. We zijn nu bezig met het geval dat er geen enkele lijn uit de *string art* over pixel i loopt, dus lijn j loopt **niet** over pixel i. Volgens de definitie van matrix A geldt dan: $a_{ij} = 0$, dus de term $a_{ij}x_j = 0$. Elke

term is 0, dus de som $a_{i1}x_1 + \cdots + a_{in}x_n$ is 0. $C(Ax)_i$ berekenen we dan als volgt:

$$\begin{aligned} C(Ax)_i &= \lfloor (1 - \min(1, (Ax)_i)) \cdot 255 \rceil \\ &= (1 - \min(1, a_{i1}x_1 + \cdots + a_{in}x_n)) \cdot 255 \\ &= (1 - \min(1, 0)) \cdot 255 \\ &= (1 - 0) \cdot 255 \\ &= 255. \end{aligned}$$

In het eerste geval geldt dus dat $F(x)_i = C(Ax)_i$.

In het tweede geval is er minstens 1 lijn gekozen voor de *string art* die over pixel i loopt. Er is dus ten minste 1 term $a_{ij}x_j$ waarvoor geldt dat $a_{ij} = 1$ en $x_j = 1$. Verder geldt dat elke term ≥ 0 is, want de matrix A en de vector x bevatten alleen waarden ≥ 0. Dus er geldt dat $a_{i1}x_1 + \cdots + a_{in}x_n \geq 1$. We berekenen $C(Ax)_i$ als volgt:

$$\begin{aligned} C(Ax)_i &= \lfloor (1 - \min(1, (Ax)_i)) \cdot 255 \rceil \\ &= (1 - \min(1, a_{i1}x_1 + \cdots + a_{in}x_n)) \cdot 255 \\ &= (1 - 1) \cdot 255 \\ &= 0. \end{aligned}$$

Ook in het tweede geval geldt dus dat $F(x)_i = C(Ax)_i$. □

Claim: Bij gebruik van Wu geldt: $\forall x \in \{0,1\}^n : F(x) = C(Ax)$.
Bewijs: We weten dat voor een willekeurige $i \in \{1, \ldots, m\}$ geldt dat:

$$F(x)_i = \begin{cases} 0 & \text{Als } s(x)_i > 1 \\ \lfloor 255 - 255 \cdot s(x)_i \rceil & \text{Anders.} \end{cases}$$

Bij het algoritme van Wu heeft a_{ij} de waarde $1 - \text{error}_{ij}$. Voor x_j geldt dat $x_j = 0$ of $x_j = 1$. We kunnen de som $a_{i1}x_1 + \cdots + a_{in}x_n$ daarom schrijven als

$$\begin{aligned} (Ax)_i &= \sum_{j=1}^{n} a_{ij}x_j \\ &= \sum_{\substack{j \in \{1,\ldots,n\}: \\ x_j = 1}} a_{ij} \\ &= \sum_{\substack{j \in \{1,\ldots,n\}: \\ x_j = 1}} 1 - \text{error}_{ij}. \end{aligned}$$

Deze som hebben we hiervoor precies als $s(x)_i$ gedefinieerd. Dus dan kunnen we $C(Ax)_i$ als volgt berekenen:

$$\begin{aligned} C(Ax)_i &= \lfloor (1 - \min(1, (Ax)_i)) \cdot 255 \rceil \\ &= \lfloor (1 - \min(1, s(x)_i)) \cdot 255 \rceil \\ &= \begin{cases} (1 - 1) \cdot 255 & \text{Als } s(x)_i > 1 \\ \lfloor (1 - s(x)_i) \cdot 255 \rceil & \text{Anders} \end{cases} \\ &= \begin{cases} 0 & \text{Als } s(x)_i > 1 \\ \lfloor 255 - 255 \cdot s(x)_i \rceil & \text{Anders.} \end{cases} \end{aligned}$$

Dit is precies hetzelfde als $F(x)_i$ bij het algoritme van Wu. Dus ook bij het algoritme van Wu geldt $F(x)_i = C(Ax)_i$ voor alle $i \in \{1, \ldots, m\}$ en $x \in \{0,1\}^n$. □

We hebben nu dus een compacte functie F die x naar een vector stuurt die de *string art* weergeeft. Nu kunnen we deze vector vergelijken met de vector y en een zo goed mogelijke combinatie x zoeken zodat het verschil tussen y en $F(x)$ zo klein mogelijk wordt. In de volgende paragraaf bespreken we hoe we op zoek gaan naar deze x.

6.2 Bepalen van de lijnen

We willen een combinatie x van lijnen vinden zodat het verschil tussen y en $F(x)$ zo klein mogelijk wordt. Vergelijken gaan we doen met de norm van het verschil. We zoeken een vector x zodat deze norm zo klein mogelijk wordt:

$$\min_{x \in \{0,1\}^n} \|C(Ax) - y\|.$$

Birsak gebruikt in eerste instantie de euclidische norm ofwel de L^2-norm. Later in zijn artikel gebruikt hij ook de L^1-norm, dat een soortgelijk resultaat lijkt te geven. In principe kan dit algoritme worden uitgevoerd met elke norm en het is een interessant onderzoek of het uitmaakt welke norm wordt gebruikt. In deze scriptie hebben we ons beperkt tot de euclidische norm. Verder is het goed om te noemen dat, aangezien we te maken hebben met een optimaliseringsprobleem, lineair programmeren een goede methode is om de beste x te vinden. Birsak onderzoekt deze methode ook en gebruikt de software van Gurobi Optimizer [5] om x te vinden. Dit geeft wederom een vergelijkbaar resultaat als de manier die we hieronder zullen beschrijven om een goede x te vinden. Om de software van Gurobi niet aan te hoeven schaffen en bovendien vanwege een gebrek aan tijd hebben we ons alleen op de onderstaande manier gefocust.

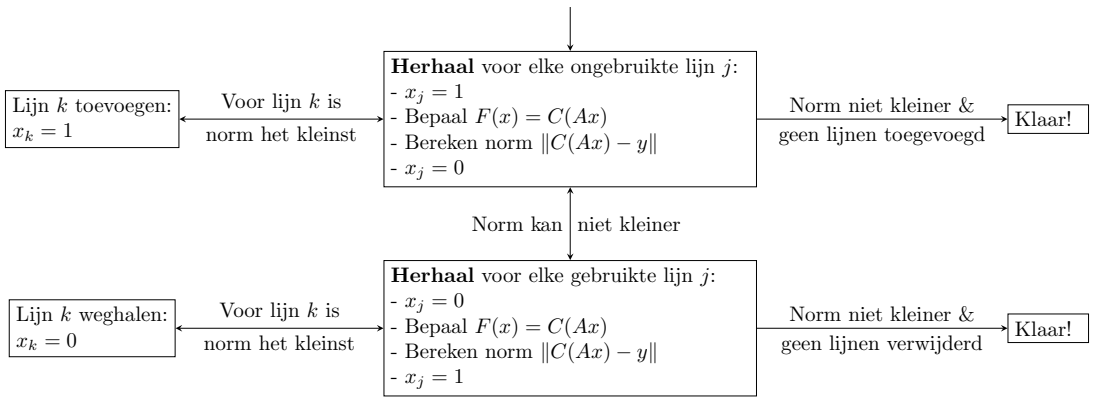
Om de beste x te vinden, gebruiken we een *greedy* algoritme. We herhalen twee stappen, net zolang totdat de norm niet meer kleiner kan worden.

1. In de eerste stap lopen we alle mogelijke lijnen langs. Voor elke lijn berekenen we de norm van het verschil als deze lijn wordt toegevoegd. Voor lijn j veranderen we tijdelijk x_j van 0 naar 1 en berekenen $\|C(Ax) - y\|$. Als we dit voor alle lijnen hebben gedaan, kiezen we de lijn waarbij $\|C(Ax) - y\|$ het kleinste wordt. Deze lijn, zeg k, voegen we toe aan de combinatie van lijnen, dus $x_k = 1$. Daarna herhalen we deze stap, we lopen opnieuw alle lijnen langs en kiezen de lijn die de norm het kleinste maakt. We stoppen als we geen lijn meer kunnen vinden die de norm kleiner maakt.

2. In de tweede stap gaan we lijnen weghalen. Omdat dit algoritme een *greedy* algoritme is, kan het zijn dat er vroeg in het algoritme een keuze wordt gemaakt die later in het algoritme geen goede keuze blijkt te zijn. We lopen daarom alle lijnen die zijn toegevoegd aan de *string art* langs en berekenen de norm als we deze lijn zouden weghalen. Voor elke lijn j veranderen we dus tijdelijk x_j naar 0 en berekenen $\|C(Ax) - y\|$. We kiezen de lijn die de norm het kleinste maakt en halen hem uit de combinatie van lijnen. We herhalen deze stap totdat er geen lijnen meer te verwijderen zijn zodanig dat de norm kleiner wordt.

3. Stap 1 en stap 2 worden herhaald tot de norm niet meer kleiner kan worden.

In Figuur 16 staan de bovenstaande stappen nog eens duidelijk op een rijtje.

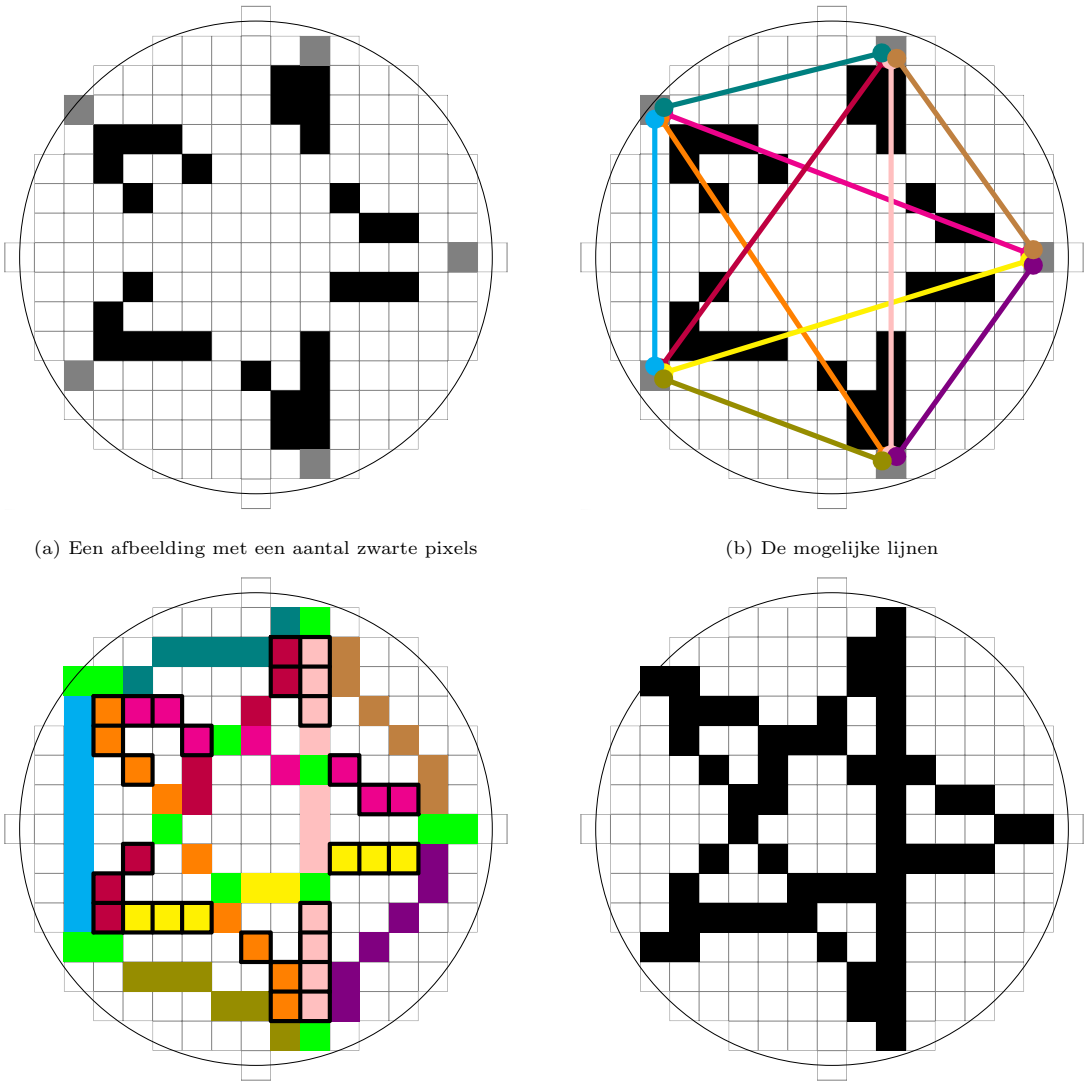


Figuur 16: Algoritme om de combinatie van lijnen te bepalen

Aan het eind van deze stappen hebben we een combinatie x van lijnen gevonden waarvoor het verschil met de input afbeelding erg klein is. Het is niet zo dat dit algoritme altijd de x

vindt waarvoor het verschil het *kleinst* is. Om dit aan te tonen laten we in Voorbeeld 6.1 een afbeelding zien waarvoor dit algoritme niet de beste x vindt.

Voorbeeld 6.1. In Figuur 17a is een cirkelvormige afbeelding te zien met een straal van 8 pixels. We plaatsen 5 spijkers over de rand van de cirkel. Als we de spijkers dun genoeg nemen en de draad dik genoeg, kunnen we verwaarlozen dat de spijkers een linker- en een rechterkant hebben. De vier verschillende lijnen vallen haast samen als we twee spijkers met elkaar verbinden. Neem bijvoorbeeld spijkers van een millimeter dik en draad van 5 millimeter dik. De dikte van de spijker, `width_pin` zoals we in het vorige hoofdstuk zagen, is dan slechts 0,2 pixels. De coördinaten van de linker- en de rechterkant van de spijkers vallen dan op dezelfde pixel. De vijf grijze pixels in Figuur 17a stellen de spijkers voor. Merk op dat de cirkel moet worden benaderd door de vierkante pixels, evenals de coördinaten van de spijkers.



(a) Een afbeelding met een aantal zwarte pixels

(b) De mogelijke lijnen

(c) Pixels bij de mogelijke lijnen

(d) Een combinatie van lijnen met kleinere norm

Figuur 17: Een afbeelding waarbij het algoritme om de beste combinatie x van lijnen te vinden niet goed werkt

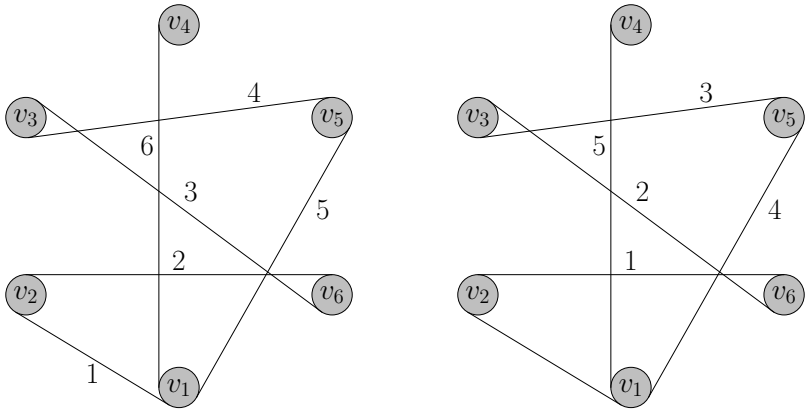
Als we van deze afbeelding een *string art* willen maken, zijn er 10 mogelijke lijnen, zie Figuur 17b. In Figuur 17c is te zien welke pixels bij de lijnen horen volgens het algoritme van Bresenham, waarbij we de pixels die bij meerdere lijnen horen groen hebben gekleurd. De pixels met een dikke zwarte omranding zijn de pixels in de originele afbeelding. Het algoritme loopt al deze 10 lijnen langs om te kijken of de norm vermindert als de lijnen worden toegevoegd. Maar van alle lijnen is minder dan de helft van de pixels zwart gekleurd in de originele afbeelding. De

norm van het verschil met de nulvector (als x de nulvector is, is er geen enkele lijn getekend) is dus altijd kleiner dan de norm van het verschil met een afbeelding met één lijn. Het algoritme zal dus geen van de lijnen kiezen. De combinatie x die het algoritme als beste x geeft is de nulvector. De originele afbeelding bevat 30 zwarte pixels, dus de norm van het verschil is dan: $\|\mathbf{0} - y\| = \sqrt{30 \cdot (255-0)^2} \approx 1396,7$.

Toch bestaat er een x waarvoor de norm wel verminderd kan worden. In Figuur 17d is een afbeelding te zien waarbij 5 lijnen zijn gekozen. De lijnen bevatten alle zwarte pixels van de originele afbeelding. De lijnen bevatten ook witte pixels uit de originele afbeelding, maar dat aantal is kleiner dan het aantal zwarte pixels. De lijnen zijn zo gekozen dat ze overlappen in een deel van de witte pixels van de originele afbeelding, maar niet overlappen in de zwarte pixels. Als je elke lijn afzonderlijk bekijkt, is het aantal witte pixels te groot. Maar als je de lijnen samen bekijkt, tellen we nog steeds evenveel zwarte pixels (daar zit geen overlap in), maar minder witte pixels. Om het preciezer te maken: de originele afbeelding bevat dus 30 zwarte pixels, de afbeelding met de 5 lijnen bevat 57 zwarte pixels, waaronder precies de 30 uit de originele afbeelding. De norm van het verschil met de afbeelding met 5 lijnen is $\|y - F(x)\| = \sqrt{27 \cdot (0-255)^2} \approx 1325,0$.

6.3 Bepalen van een Eulercykel

Nu hebben we de beste combinatie x van lijnen bepaald. Maar we zijn nog niet klaar. Bij het maken van een *string art* willen we dat de hele *string art* uit één draad bestaat. In Figuur 18 zien we een voorbeeld van een *string art* met 6 spijkers en 6 gekozen lijnen. Links zien we een volgorde van lijnen waarin we alle lijnen kunnen trekken met één draad. We beginnen bij v_1 en eindigen bij v_4. Als je echter bij bijvoorbeeld spijker v_2 begint, zoals in de figuur rechts, lukt het niet om alle lijnen met één draad te trekken. We moeten dus een manier vinden om de lijnen in een volgorde achter elkaar te zetten, zodat we de lijnen kunnen trekken met één draad.



Figuur 18: Een *string art* met 6 spijkers en 6 gekozen lijnen, afhankelijk van de volgorde van lijnen lukt het om de lijnen met 1 draad te trekken

Hiervoor beschouwen we de combinatie van lijnen als een graaf met knopen en zijden. De spijkers kunnen we zien als knopen en de getrokken lijnen tussen spijkers zien we als zijden. Op deze manier hebben we een *ongerichte* - het maakt niet uit of een lijn van spijker 1 naar spijker 2 of van spijker 2 naar spijker 1 wordt getrokken - en een *meervoudige graaf* - spijkers kunnen op vier verschillende manieren verbonden worden met elkaar.

Het trekken van de lijnen met de draad kunnen we zien als een *wandeling* door de graaf. Een wandeling door een graaf is een rij knopen $(v_1, \ldots, v_n)$, waarvoor geldt dat twee opeenvolgende knopen in de rij verbonden zijn in de graaf, dus er is ten minste één zijde tussen de twee knopen. Het trekken van de lijnen in Figuur 18 kunnen we bijvoorbeeld zien als de wandeling $(v_1, v_2, v_6, v_3, v_5, v_1, v_4)$. Omdat we alle lijnen uit onze combinatie x willen trekken, willen we dus dat elke zijde van de graaf die bij x hoort in de wandeling voorkomt. Daarnaast is x een binaire vector, elke lijn wordt maximaal één keer getrokken. Dus elke zijde van onze graaf moet niet

alleen *minstens* één keer voorkomen, maar ook *maximaal* één keer. We kunnen dus concluderen dat we een wandeling zoeken waarbij elke zijde *precies* één keer voorkomt. Zulke wandelingen noemen we Eulerwandelingen en Eulercykels.

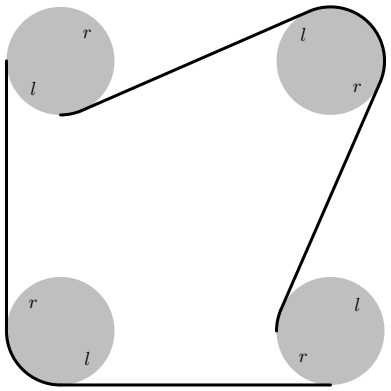
Definitie 6.2. Een *Eulerwandeling* tussen twee knopen in een graaf is een wandeling waarbij elke zijde van de graaf precies één keer voorkomt. Een graaf met een Eulerwandeling noemen we een *semi-Eulergraaf*.

Definitie 6.3. Een *Eulercykel* is een Eulerwandeling waarbij geldt dat het beginpunt gelijk is aan het eindpunt. Een graaf met een Eulercykel noemen we een *Eulergraaf*.

Volgens Definitie 6.2 en 6.3 willen we dus dat onze graaf een semi-Eulergraaf of een Eulergraaf is. Als dat zo is, dan is er een Eulerwandeling of een Eulercykel te vinden in de graaf. We kunnen de lijnen dan zo aan elkaar schakelen dat elke lijn precies één keer getrokken wordt. Natuurlijk is onze graaf niet altijd toevallig een (semi-)Eulergraaf. In dat geval zullen we extra lijnen moeten toevoegen om er een (semi-)Eulergraaf van te maken. Die lijnen trekken we om de spijkers heen aan de buitenkant van de cirkel, zodat ze niet het resultaat bederven.

6.3.1 Eulergrafen bij *string arts*

Normaal gesproken geldt dat een samenhangende graaf (tussen elke twee knopen bestaat een wandeling) een Eulergraaf is als elke knoop een even graad heeft, dus aan elke knoop zit een even aantal zijden [4]. Bij een semi-Eulergraaf heeft bijna elke knoop een even graad, behalve het begin- en eindpunt. Maar onze graaf heeft één afwijking ten opzichte van normale grafen. De draad gaat om de spijker heen. Een lijn komt dus altijd aan één kant bij de spijker aan en vertrekt vervolgens vanuit de andere kant van de spijker. Hierdoor kan het voorkomen dat elke knoop in onze graaf een even graad heeft maar de *string art* toch niet te maken is met één draad. In Figuur 19 is een voorbeeld te zien van een combinatie van lijnen met vier spijkers. Aan elke spijker zitten twee lijnen, dus elke spijker heeft een even graad. Toch kunnen we de *string art* niet met één draad maken, je zou altijd vastlopen bij een spijker.



Figuur 19: Hoewel elke spijker een even graad heeft, kan deze *string art* niet met één draad gemaakt worden

Het voldoet dus niet om alleen te controleren of de graden van alle knopen (of alle knopen behalve twee) even zijn. We zullen de Eulergraaf en de semi-Eulergraaf anders moeten definiëren voor onze *string arts*. Omdat elke knoop in een Eulercykel een even graad heeft kun je altijd aankomen in een knoop en weer vertrekken via een ongebruikte zijde in de Eulercykel. Het aantal ingaande zijden is dus even groot als het aantal uitgaande zijden van een knoop in een Eulercykel. En dat is precies waar het misgaat in onze grafen, bijvoorbeeld in Figuur 19. Zodra je aankomt in de spijker linksboven, kun je niet vertrekken, omdat de enige andere ongebruikte lijn aan de spijker aan dezelfde kant zit, namelijk de linkerkant (l). Voor de spijker rechtsonder geldt hetzelfde, beide lijnen zitten nu aan de rechterkant van de spijker (r). We moeten bij onze grafen dus onderscheid maken tussen lijnen vanuit de rechter- en de linkerkant van de spijker. Als we willen vertrekken uit een spijker, moet er een ongebruikte zijde aan de *andere* kant van de spijker zitten. Het aantal lijnen aan de linkerkant moet dus gelijk zijn aan het aantal lijnen aan de rechterkant van een spijker.

Definitie 6.4. Een *Eulergraaf voor een string art* is een combinatie van lijnen waarbij elke spijker evenveel lijnen aan de rechterkant als aan de linkerkant heeft.

Definitie 6.5. Een *semi-Eulergraaf voor een string art* is een combinatie van lijnen waarbij, op twee spijkers na, elke spijker evenveel lijnen aan de linkerkant als aan de rechterkant heeft. De overige twee spijkers hebben 1 lijn meer aan één van de twee kanten.

6.3.2 Lijnen toevoegen

Als onze *string art* een semi-Eulergraaf of een Eulergraaf is volgens bovenstaande definities, dan hoeven we geen lijnen toe te voegen. Als onze *string art* niet een semi-Eulergraaf of een Eulergraaf is, zullen we lijnen moeten toevoegen om er een (semi-)Eulergraaf van te maken. Hiervoor ordenen we eerst alle spijkers die niet aan bovenstaande eis voldoen, naar het aantal lijnen dat ze missen. Een spijker i kan bijvoorbeeld 15 lijnen aan de linkerkant hebben en 5 lijnen aan de rechterkant. De spijker mist dan dus 10 lijnen (aan de rechterkant). Als we alle spijkers hebben geordend, beginnen we bij de spijker die het grootste aantal lijnen mist. Deze verbinden we met de spijker die daarna het grootste aantal lijnen mist, we verbinden deze twee spijkers zo vaak als dat de tweede spijker lijnen mist. We voegen de lijnen toe aan de kanten waar de spijkers de lijnen missen. Op die manier hebben we de tweede spijker in evenwicht gebracht, de spijker heeft evenveel lijnen aan de rechterkant als aan de linkerkant. Als de eerste spijker nu nog steeds lijnen mist, verbinden we de spijker met de derde spijker in de lijst, en zo nodig met de vierde spijker, etc. Ten slotte hebben we genoeg lijnen toegevoegd zodat deze spijker evenveel lijnen links en rechts heeft. Hierna gaan we door naar de eerstvolgende spijker in de lijst die nog lijnen mist en herhalen we het proces.

Het kan niet voorkomen dat we één spijker overhouden die nog een oneven aantal lijnen mist. Als alle andere spijkers in evenwicht zijn, hebben die spijkers allemaal een even graad. Een spijker die nog een oneven aantal lijnen mist, heeft een oneven graad op dat moment. Zouden we de graden van alle spijkers bij elkaar optellen, dan krijgen we dus een oneven getal. Maar met elke lijn die gekozen is voor de *string art*, zijn er twee spijkers waarvan de graad met 1 stijgt. Dus voor elke *string art* graaf (en voor elke graaf in het algemeen) geldt dat alle graden bij elkaar opgeteld een even getal moet geven.

Het kan wel gebeuren dat we één spijker overhouden die nog een even aantal lijnen mist. Dat is eenvoudig op te lossen: we kiezen een willekeurige andere spijker en verbinden deze twee spijkers zo vaak met elkaar zodat de laatste spijker ook in evenwicht is. Om de andere spijker in evenwicht te houden, voegen we de helft van de lijnen aan de ene kant van de spijker toe en de andere helft van de lijnen aan de andere kant. Hieronder staan de stappen nog eens op een rij: Stel, er zijn k spijkers die niet evenveel ingaande als uitgaande lijnen hebben.

1. Order de spijkers naar het aantal lijnen dat elke spijker mist:
   $\{(v_1, x_1), (v_2, x_2), \ldots, (v_{k-1}, x_{k-1}), (v_k, x_k)\}$, waarbij x_i het aantal lijnen is dat spijker v_i mist. Er geldt: $x_1 \geq x_2 \geq \cdots \geq x_k$.

2. Voor elke eerste spijker v_i in de lijst die nog niet evenveel ingaande als uitgaande lijnen heeft, herhalen we de onderstaande stappen, totdat we alle spijkers hebben gehad of er nog 1 overhouden:

   (a) Zoek het kleinste gehele getal $n \geq 1$ zodat $x_i \leq x_{i+1} + x_{i+2} + \cdots + x_{i+n}$.

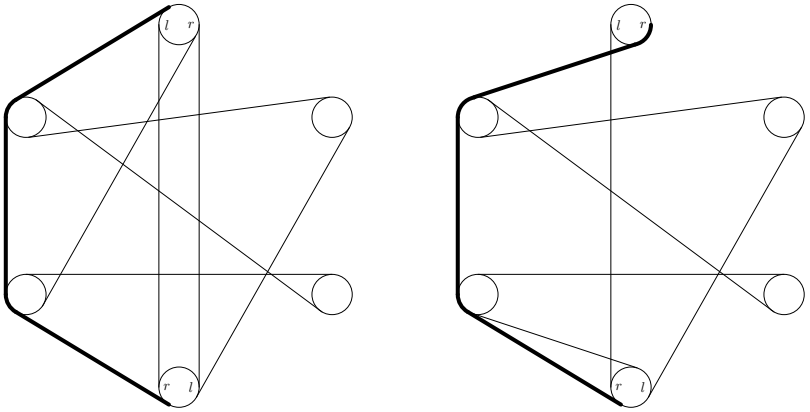
   (b) Voor alle $j \in \{1, 2, \ldots, n-2, n-1\}$: verbind spijker v_i en v_{i+j} met elkaar met x_{i+j} lijnen, aan de kanten waar ze lijnen missen.

   (c) Verbind spijker v_i en v_{i+n} met elkaar met $y = x_i - x_{i+1} - \cdots - x_{i+(n-1)}$ lijnen, aan de kanten waar ze lijnen missen.

3. Als er nu nog 1 spijker v is die x lijnen mist, dan verbinden we v met een willekeurige spijker w met $\frac{x}{2}$ lijnen aan de rechterkant van w en $\frac{x}{2}$ lijnen aan de linkerkant van w.

Merk op dat we het algoritme ook kunnen stoppen als er twee spijkers over zijn die allebei 1 lijn missen. Deze twee spijkers worden dan het begin- en eindpunt van de Eulerwandeling. Als we in het algoritme ervoor zorgen dat alle spijkers in evenwicht zijn, dan kunnen we een Eulercykel maken.

Bij het verbinden van de spijkers moeten we er dus op letten dat de lijnen aan de goede kanten van de spijkers zitten. Verder trekken we de draad aan de buitenkant van de cirkel, om de spijkers heen, zodat het geen invloed heeft op het resultaat. In Figuur 20 is te zien hoe we twee spijkers langs de buitenkant van de cirkel verbinden met elkaar, allebei aan dezelfde kant van de spijker of aan twee verschillende kanten van de spijkers.



Figuur 20: Extra lijnen trekken we langs de buitenkant van de cirkel

6.3.3 Hierholzer algoritme

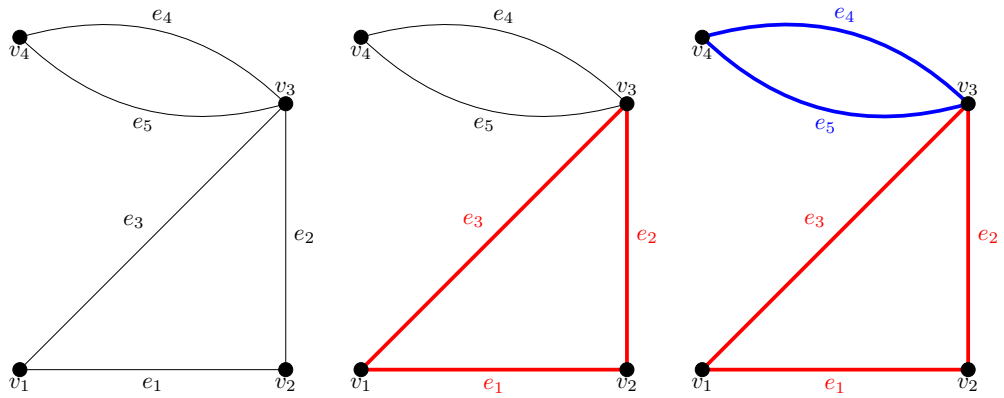
Nu we een semi-Eulergraaf of een Eulergraaf hebben, kunnen we de Eulercykel of -wandeling zoeken. Als we weten in welke volgorde de zijden worden doorlopen zodat ze een Eulercykel of Eulerwandeling vormen, weten we de volgorde waarin we de spijkers met elkaar moeten verbinden zodat het met één draad gedaan kan worden. Het vinden van een Eulercykel of Eulerwandeling in een graaf kan met behulp van het algoritme van Hierholzer [6].

Het algoritme van Hierholzer begint bij een willekeurige knoop (of het beginpunt als we slechts een Eulerwandeling zoeken). Vanuit deze knoop maken we een wandeling door de graaf, met telkens ongebruikte zijden (dit is mogelijk omdat de knopen een even graad hebben), totdat we weer bij de knoop zijn waar we zijn begonnen (of bij het eindpunt van de Eulerwandeling). Als we nu de hele cykel of wandeling hebben, zijn we klaar. Als dat niet het geval is, kiezen we een knoop v in de cykel of wandeling die nog ongebruikte zijden heeft. Vanuit deze knoop maken we een nieuwe wandeling van ongebruikte zijden totdat we weer terug zijn bij v. Deze nieuwe cykel stoppen we in de oude cykel (of wandeling). We knippen de cykel (of wandeling) door bij v en plaatsen op die plek de nieuwe cykel. Op die manier wordt het één grote cykel (of wandeling). Als we nu nog steeds niet klaar zijn, blijven we dit herhalen totdat alle zijden zijn gebruikt en we de Eulercykel of -wandeling hebben gevonden. Hieronder nog eens duidelijk op een rij de stappen:

1. Begin bij een willekeurige knoop en maak een cykel (of wandeling) van ongebruikte zijden: $c_1 = (v_1, v_2, \ldots, v_k, v_1)$.
2. Herhaal onderstaande stappen totdat alle zijden voorkomen in de cykel:
   (a) Kies een knoop v_i uit de cykel (of wandeling) die nog ongebruikte zijden heeft.
   (b) Maak vanuit v_i een cykel van ongebruikte zijden: $c_2 = (v_i, w_1, w_2, \ldots, w_l, v_i)$.
   (c) Plaats de nieuwe cykel c_2 in de oude cykel c_1 bij v_i: $c_{12} = (v_1, \ldots, v_{i-1}, v_i, w_1, \ldots, w_l, v_i, v_{i+1}, \ldots, v_k, v_1)$.

Voorbeeld 6.6. Alle knopen in de graaf in Figuur 21a hebben een even graad, dus deze graaf is een Eulergraaf en bevat een Eulercykel. In Stap 1 van het Hierholzer algoritme nemen we een willekeurige knoop, bijvoorbeeld knoop v_1. We maken vanuit v_1 een wandeling met ongebruikte

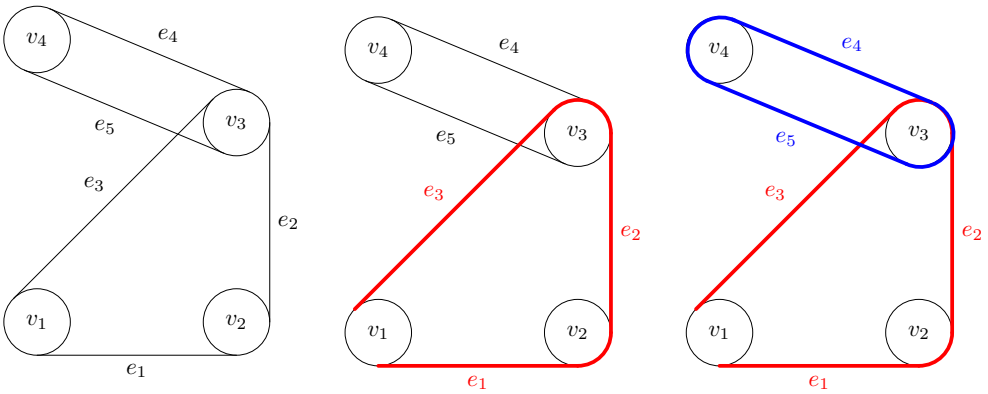
zijden. We kunnen bijvoorbeeld eerst zijde e_1, dan e_2 en daarna e_3 kiezen. Dan zijn we weer terug bij v_1 en hebben we een cykel (v_1, v_2, v_3, v_1), rood gekleurd in Figuur 21b. In Stap 2 kiezen we knoop v_3, deze knoop heeft nog ongebruikte zijden. Vanuit deze knoop maken we de blauwe cykel in Figuur 21c: (v_3, v_4, v_3). We plaatsen deze cykel in de originele cykel bij knoop v_3. Nu hebben we alle zijden gebruikt en de Eulercykel gevonden: $(v_1, v_2, v_3, v_4, v_3, v_1)$.



(a) Een multigraaf (b) Eerste stap (c) Tweede stap

Figuur 21: Bepalen van een Eulercykel van een graaf met het algoritme van Hierholzer

Bij onze grafen moeten we het Hierholzer algoritme een klein beetje aanpassen, zodat er rekening wordt gehouden met de verschillende kanten van spijkers. Als we bij een spijker aangekomen zijn, kunnen we niet simpelweg een ongebruikte zijde kiezen, maar moeten we een ongebruikte zijde aan de *andere* kant van de spijker kiezen. Zolang we dat doen werkt het algoritme verder hetzelfde en zullen we een Eulercykel of een Eulerwandeling vinden. We laten Voorbeeld 6.6 nog een keer zien, maar dan met een graaf voor een *string art*.



(a) Een multigraaf voor een *string art* (b) Eerste stap (c) Tweede stap

Figuur 22: Bepalen van een Eulercykel van een graaf voor een *string art* met het algoritme van Hierholzer

Voorbeeld 6.7. Alle knopen in de graaf in Figuur 22a hebben evenveel lijnen aan de rechter- en aan de linkerkant. Deze graaf is dus een Eulergraaf voor een *string art* en bevat een Eulercykel. In Stap 1 van het Hierholzer algoritme nemen we weer knoop v_1 en we kiezen ongebruikte zijden e_1 en e_2. Bij v_3 kunnen we alleen ongebruikte zijden aan de andere kant van de spijker kiezen, dus e_3 of e_4. We kiezen e_3 en krijgen de rode cykel in Figuur 22b: (v_1, v_2, v_3, v_1). In Stap 2 kiezen we knoop v_3, deze knoop heeft nog een ongebruikte zijde aan de andere kant van de spijker, namelijk e_4. Zo krijgen we de blauwe cykel in Figuur 22c: (v_3, v_4, v_3). We plaatsen deze cykel in de originele cykel bij v_3 Nu hebben we alle zijden gebruikt en de Eulercykel gevonden: $(v_1, v_2, v_3, v_4, v_3, v_1)$.

In de vorige fase van het algoritme hadden we een combinatie van lijnen bepaald die de best mogelijke *string art* maakt. In deze fase hebben we met die lijnen een Eulercykel of Eulerwandeling gemaakt. Deze cykel of wandeling geeft ons de volgorde waarin we de lijnen met elkaar moeten verbinden zodat we het met één draad kunnen doen. Dit voltooit het algoritme van Birsak om een *string art* te maken die een portretfoto zo goed mogelijk benadert.

6.4 Vergelijking met Vrellis

We hebben nu beide algoritmen besproken. Hoewel het algoritme van Birsak een heleboel meer uitleg vergde, lijken de twee algoritmen best op elkaar. We kunnen het algoritme van Vrellis namelijk ook zien als het optimaliseren van een norm.

We passen de principes van het algoritme van Birsak toe op dat van Vrellis en schrijven de portretfoto als een vector y. Bij Vrellis gaan we bij elke stap op zoek naar de donkerste lijn in de foto. We tellen de grijswaarden bij elkaar op en kiezen de lijn met de grootste som. Nadat een lijn is gekozen, wordt er een bepaalde waarde van de grijswaarden van de pixels van de lijn afgetrokken, in Hoofdstuk 5 noemden we deze waarde de `darkness`. De vector y wordt bij Vrellis dus bij elke stap aangepast, waar die bij Birsak constant blijft.

In plaats van telkens van een lijn de som van de grijswaarden te berekenen, kunnen we ook bekijken wat het verschil is tussen de (geüpdatete) vector y en een vector y'. Die vector y' construeren we als volgt: als we gebruikmaken van het lijnalgoritme van Bresenham, dan maken we alle pixels van de lijn wit, dus waarde 0 (bij Vrellis hadden we de grijswaarden geïnverteerd). Als we gebruikmaken van het lijnalgoritme van Wu, dan geven we alle pixels van de lijn de waarde error $\cdot\, g$, waarbij g de grijswaarde van de pixel van de lijn is en error de bijbehorende error zoals we in Hoofdstuk 3 berekenden. Net als bij Birsak kunnen we nu de norm van dit verschil berekenen. Omdat bij Vrellis lijnen meerdere keren gekozen mogen worden, kunnen we niet het verschil nemen tussen y en de *string art* vector, zoals we bij Birsak deden. Een lijn die voor de tweede keer getrokken wordt heeft namelijk geen invloed op de *string art* vector, aangezien de pixels van die lijn allemaal al zwart zijn. Verder gebruikten we bij Birsak de L^2-norm om de norm van het verschil te berekenen, maar bij Vrellis gebruiken we de L^1-norm, waarbij de norm van een vector $x = (x_1, \ldots, x_n)$ berekend wordt met $\sum_{i=1}^{n} |x_i|$. Op deze manier geeft de norm van het verschil $y' - y$ precies hetzelfde resultaat als de som van de grijswaarden (of de som van de grijswaarden vermenigvuldigd met $(1 - \text{error})$ in het geval van Wu). We willen dat deze som zo groot mogelijk is, dus we zoeken naar de lijn die de norm van het verschil maximaliseert.

Dus hoewel er kleine verschillen zijn, zoals de soort norm die gebruikt wordt en de vectoren die van elkaar worden afgetrokken, wordt bij Vrellis net zoals bij Birsak in elke stap gezocht naar de lijn die een norm optimaliseert. Het derde, en misschien wel grootste verschil is het aantal lijnen dat onderzocht wordt. Bij Birsak worden alle mogelijke lijnen van de *string art* geprobeerd, bij Vrellis zijn het alleen de lijnen vanuit één bepaalde spijker. Op die manier is de volgorde van de spijkers voor het maken van de *string art* meteen duidelijk, maar kunnen we niet meer lijnen weghalen om de norm te verbeteren, terwijl we bij Birsak de norm misschien iets meer kunnen verbeteren, maar we nog wel het Hierholzer algoritme moeten toepassen om de volgorde te achterhalen.

Hieronder vatten we beide algoritmen heel kort samen, waardoor de verschillen tussen de algoritmen meteen duidelijk zijn.

Vrellis	**Birsak**
Voor alle lijnen vanuit 1 spijker ($2(p-1)$): - Bereken $\|y' - y\|_1$	Voor alle mogelijke lijnen (maximaal $4\binom{p}{2}$): - Bereken $\|C(Ax) - y\|_2$
Kies de lijn met de *grootste* norm: - voeg spijker toe aan lijst - trek `darkness` van de grijswaarden af in y	Kies de lijn met de *kleinste* norm: - x-coördinaat wordt 1 bij toevoegen - x-coördinaat wordt 0 bij verwijderen
Herhaal bovenstaande stappen: - voor de volgende spijker - totdat de norm 0 is voor alle lijnen	Herhaal bovenstaande stappen: - wissel toevoegen en verwijderen af - totdat de norm geminimaliseerd is
Lijst met spijkers geeft volgorde	Maak een (semi-)Eulergraaf Hierholzer algoritme geeft volgorde

Hoofdstuk 7

Implementatie en resultaten Birsak

In dit hoofdstuk bespreken we de implementatie en de resultaten van het algoritme van Birsak. Net als bij het algoritme van Vrellis bewerken we de afbeelding en bepalen we de coördinaten van de spijkers. De afbeelding, eerst een matrix van lengte en breedte `width`, wordt in dit algoritme omgezet in een vector. Verder hebben we voor dit algoritme een lijst nodig met alle mogelijke lijnen.

```
from itertools import combinations

y = img_resized.reshape(width**2)
lines = [(pins[x], pins[y]) for (x,y) in combinations(range(2*nPins), 2)]
```

Het algoritme van Birsak begint met de functie F. Hiervoor hebben we de matrix A nodig, zoals die gedefinieerd is in het vorige hoofdstuk. Hieronder is de implementatie daarvoor te zien. De meest ingewikkelde stap is het omzetten van de nummers van de pixels. Het Bresenham algoritme (of Wu algoritme) geeft de pixels weer met x- en y-coördinaten. Voor het maken van matrix A moeten we weten welk nummer een pixel heeft als alle rijen van een afbeelding achter elkaar worden gezet in een vector. Hiervoor gebruiken we de oorspronkelijke breedte (tevens de lengte) van de afbeelding, `width`. Het nummer van de pixel berekenen we dan met `ypixel*width + xpixel`.

```
def matrix(lines, pins, width):
    nRows = width**2  # total number of pixels
    nColumns = len(lines) # total number of lines
    matrix = np.zeros((nRows, nColumns))
    for j in range(nColumns):
        (pin1, pin2) = lines[j]
        ypix, xpix = Bresenham(pin1, pin2)
        # or ypix, xpix, errors = Wu(pin1, pin2)
        pix = [y*width+x for (y,x) in zip(ypix, xpix)]
        matrix[pix, [j]*len(pix)] = 1
        # or matrix[pix, [j]*len(pix)] = 1-np.array(errors)

    return matrix
```

Nu we de matrix hebben gemaakt, kunnen we voor elke vector `x` de bijbehorende `Fx` berekenen. Dat doen we met de onderstaande functie.

```
def Fx(A, x):
    Ax = np.matmul(A,x)
    Fx = [round((1-min(1,el))*255) for el in Ax]

    return Fx
```

Vervolgens kunnen we beginnen met het toevoegen en verwijderen van lijnen. Hieronder is de implementatie te zien van een functie die bepaalt welke lijn de norm het meest vermindert als die wordt toegevoegd aan de *string art*. Voor het verwijderen van lijnen is de functie bijna hetzelfde, in de opmerkingen in het groen is te zien waar en hoe de functie afwijkt.

```
def addLine(unused, A, x, image, currentNorm):
    # unused contains numbers that represent the lines that are not used in the
    string art. In removeLine we use a list called used
    j_best = None
    for j in unused:
        x[j] = 1 # in removeLine x[j] = 0
        Fx = stringartvector(A, x)
        norm = np.linalg.norm(Fx - image)
        if norm < currentNorm:
            currentNorm = norm
            j_best = j
        x[j] = 0 # in removeLine x[j] = 1

    return (j_best, currentNorm)
```

Als de `currentNorm` niet veranderd is, dan weten we dat de norm niet verder verminderd kan worden door lijnen toe te voegen. Als dit ook geldt voor het verwijderen van lijnen, dan zijn we klaar. In de vector `x` kunnen we nu precies zien welke lijnen zijn toegevoegd aan de *string art*. Voor het maken van een simulatie is het niet nodig om het gedeelte over de Eulercykels en -wandelingen te implementeren. Het maken van een simulatie *string art* gebeurt met de volgende implementatie:

```
def simulation(Z, x, lines):
    simulation = np.ones((Z,Z), dtype="uint8")*255
    for i in range(len(x)):
        if x[i] == 1:
            (pin1, pin2) = lines[i]
            cv2.line(simulation, pin1, pin2, 0, 1)

    return simulation
```

7.1 Een alternatieve implementatie

Bij het *runnen* van de bovenstaande code blijkt echter dat het veel te traag is. Alle mogelijke lijnen moeten worden langsgelopen en bovendien kost elke stap ook nog eens veel tijd omdat het een matrixvermenigvuldiging met een enorm grote matrix bevat. We hebben het algoritme in Python geïmplementeerd. Ongetwijfeld zijn er andere programmeertalen waarin het algoritme sneller werkt. Toch hebben we eerst geprobeerd om een oplossing te vinden in Python zelf.

We hebben een manier gevonden om in ieder geval de matrixvermenigvuldiging te vermijden. In plaats van de functie F gebruiken we de functies van OpenCV om te bepalen hoe een *string art* eruit komt te zien als we lijnen toevoegen of verwijderen. We kunnen een lijn tekenen in een tijdelijke afbeelding die de *string art* voorstelt en vervolgens de norm berekenen. We gebruiken dus geen matrixvermenigvuldiging om de grijswaarden van de pixels te berekenen, maar tekenen de lijn daadwerkelijk, waardoor de pixels de juiste grijswaarden krijgen. We hoeven hiervoor de portretfoto ook niet meer om te zetten naar een vector, we blijven werken met matrices en berekenen de norm van het verschil van twee matrices in plaats van twee vectoren. We beginnen met een lege *string art*, dus `Fx` is een witte afbeelding:

```
Fx = np.ones((width, width))*255
```

Hieronder is te zien hoe de functie die de beste lijn kiest er nu uit komt te zien. We kopiëren `Fx`, zodat we daar tijdelijk een lijn aan kunnen toevoegen en de nieuwe norm kunnen berekenen. Dit gaat een stuk sneller dan de trage matrixvermenigvuldiging in de vorige implementatie.

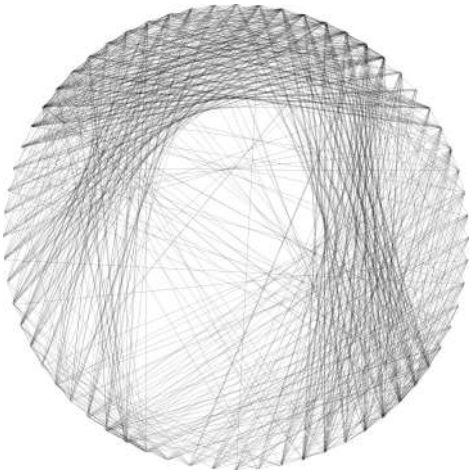
```
def addLine(unused, lines, Fx, image, pins, currentNorm):
    line_best = None
    temp_best = None
    for j in unused:
        (pin1, pin2) = lines[j]
        temp = Fx.copy()
        temp = cv2.blur(temp, (3,3))
        cv2.line(temp, pin1, pin2, 0, 1)
        norm = np.linalg.norm(temp - image)
        if norm < currentNorm:
            currentNorm = norm
            j_best = j
            temp_best = temp

    return (j_best, currentNorm, temp_best)
```

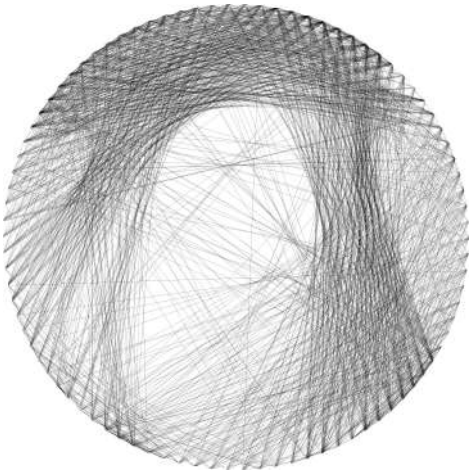
Er is één grote maar: we kunnen geen lijnen meer weghalen. Met OpenCV is het mogelijk om lijnen te tekenen, maar niet om lijnen weg te halen. Natuurlijk kun je een lijn weer wit laten maken, maar de pixels van een lijn zouden eventueel ook bij andere lijnen kunnen horen en daarom zwart of grijs moeten blijven. En natuurlijk is ook dat te achterhalen, maar dat kost veel tijd. Er is geen snelle manier om een lijn weg te halen. We kunnen dus niet `Fx` bepalen als we lijnen weghalen en moeten deze stap van het algoritme weglaten. Verder is het nodig om de functie `blur` van OpenCV toe te passen op de tijdelijke kopie van `Fx`. Deze functie zorgt ervoor dat de lijnen in `Fx` wat waziger worden gemaakt. De lijnen worden als het ware iets uitgesmeerd over de pixels eromheen. Hierdoor is het contrast minder groot en worden de *string arts* niet te donker. Je zou het kunnen vergelijken met de dikte van de lijn die we in het vorige algoritme van de portretfoto aftrokken. Hoewel een dikte van 1 het meest logisch lijkt, bleek dat een dikte van bijvoorbeeld 3 veel beter werkte. Hierdoor werden de *string arts* minder donker.

7.2 Vergelijking met Vrellis

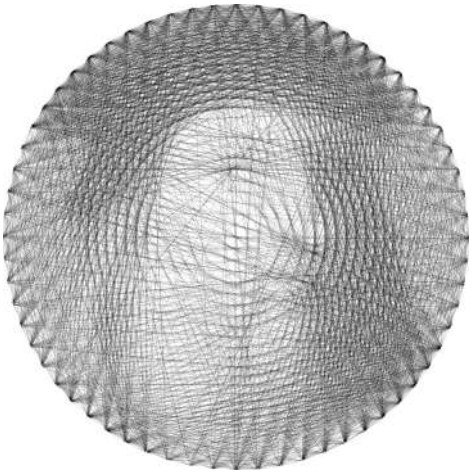
Hoewel de bovenstaande wijzigingen ervoor zorgen dat het algoritme een stuk sneller werkt, is het nog steeds heel traag. Het duurt meerdere weken om de implementatie met goede variabelen te runnen. Hieronder staan een aantal resultaten met wat minder goede variabelen. Het zijn geen perfecte *string arts*, maar ze laten wel zien dat het de goede kant op gaat. Ter vergelijking staan er twee *string arts* bij die gemaakt zijn met het algoritme van Vrellis, met dezelfde variabelen. De resultaten zijn duidelijk verschillend. Verrassend is dat in de *string arts* van Vrellis de vormen van het gezicht van Leonardo duidelijker herkenbaar zijn. Deze *string art* is donkerder, het bevat meer lijnen waardoor de vormen duidelijker worden. Een verklaring voor de tegenvallende resultaten van Birsak zou te vinden kunnen zijn in de overgeslagen stap van het lijnen weghalen in de implementatie. Het zou interessant zijn om te kijken wat het algoritme van Birsak doet bij andere variabelen. Blijft de *string art* van Vrellis dan beter herkenbaar of worden de *string arts* van Birsak beter? Aan de andere kant zouden we op basis van deze resultaten ook kunnen concluderen dat het algoritme van Vrellis beter werkt: het lijkt op het eerste gezicht resultaten te geven die beter herkenbaar zijn en het algoritme werkt een stuk sneller, het resultaat in Figuur 23d is bijvoorbeeld gegeven binnen 314 seconden, terwijl we het algoritme van Birsak meer dan 3 dagen moesten laten runnen voor het resultaat in Figuur 23b.



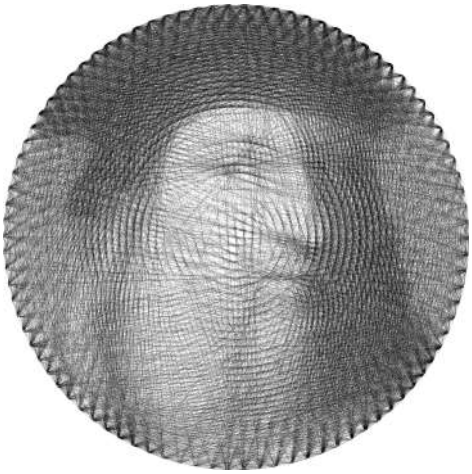
(a) Resolutie foto 1024×1024, aantal spijkers $p = 64$, algoritme van Birsak



(b) Resolutie foto 1600×1600, aantal spijkers $p = 90$, algoritme van Birsak



(c) Resolutie foto 1024×1024, aantal spijkers $p = 64$, algoritme van Vrellis



(d) Resolutie foto 1600×1600, aantal spijkers $p = 90$, algoritme van Vrellis

Figuur 23: String arts van Leonardo da Vinci met verschillende variabelen en algoritmen

Bibliografie

[1] Birsak, Michael, Florian Rist, Peter Wonka, en Przemyslaw Musialski: *String Art: Towards Computational Fabrication of String Images.* Computer Graphics Forum, 37(2):263–274, 2018.

[2] Bradski, G.: *The OpenCV Library.* Dr. Dobb's Journal of Software Tools for the Professional Programmer, 25(11):120–123, 2000.

[3] Bresenham, J. E.: *Algorithm for computer control of a digital plotter.* IBM Systems Journal, 4(1):25–30, 1965.

[4] Euler, Leonhard: *Leonhard Euler and the Koenigsberg bridges.* Scientific American, 189(1):66–72, 1953.

[5] Gurobi: *Gurobi Optimizer Reference Manual*, 2009. `https://www.gurobi.com/wp-content/plugins/hd_documentations/documentation/9.0/refman.pdf`, Bezocht op: 2022-07-12.

[6] Hierholzer, C. en C. Wiener: *Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren.* Mathematische Annalen, 6(1):30–32, 1873.

[7] Rossum, Guido van en Fred L. Drake: *Python 3 Reference Manual.* CreateSpace, Scotts Valley, CA, 2009, ISBN 1441412697.

[8] Vrellis, Petros: *A new way to knit*, 2016. `http://artof01.com/vrellis/works/knit.html`, Bezocht op: 2022-07-09.

[9] Wu, X.: *An efficient antialiasing technique.* ACM SIGGRAPH Computer Graphics, 25(4):143–152, 1991.

Bronnen figuren

Figuur 1: `https://www.artbymaaike.nl`

Figuur 2: Artikel van Michael Birsak [1]

Figuur 3a: `https://nl.m.wikipedia.org/wiki/Bestand:Venus_botticelli_detail.jpg`

Figuur 10a: `https://www.gettyimages.be/fotos/leonardo-da-vinci`

Figuur 10b: Gemaakt door auteur

Figuur 11a: `https://www.nobelprize.org/prizes/economic-sciences/1994/nash`

Figuur 15a: `https://nl.pinterest.com/pin/449867450286133530/`

Figuur 14a: `https://en.wikipedia.org/wiki/File:Einstein-tongue.jpg`

Simulaties: Gemaakt door auteur met behulp van OpenCV in Python

Overige figuren: Gemaakt door auteur met behulp van TikZ in LaTeX