Faculteit Toegepaste Wetenschappen
Vakgroep Telecommunications and Information Processing
(TELIN)Telecommunicatie en Informatieverwerking
Voorzitter: Prof. dr. ir. H. Bruneel

Computeralgoritmen voor het oplossen van legpuzzels

door Johan De Bock

Promotor: prof. dr. ir. W. Philips
Co-promotor: prof. dr. ir. J. D'Haeyer
Thesisbegeleider: dr. ir. P. De Smet

Afstudeerwerk ingediend tot het behalen van de academische graad van burgerlijk ingenieur in de computerwetenschappen

Academiejaar 2002–2003

Toelating tot bruikleen

De auteur geeft de toelating dit afstudeerwerk voor consultatie beschikbaar te stellen en delen van het afstudeerwerk te kopiëren voor persoonlijk gebruik. Elk ander gebruik valt onder de beperkingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit dit afstudeerwerk.

Johan De Bock 31 mei 2003

Dankwoord

Graag zou ik iedereen willen bedanken die heeft bijgedragen tot de verwezenlijking van deze thesis, in het bijzonder dank ik:

- mijn promotoren prof. dr. ir. W. Philips en prof. dr. ir. J. D'Haeyer en thesisbegeleider dr. ir. P. De Smet voor het scheppen van de mogelijkheid dit onderzoek te verrichten;
- mijn ouders voor de morele steun en het nalezen van de thesistekst;
- familieleden en kennissen voor het lenen van hun legpuzzels;
- alle medestudenten die nuttige tips geleverd hebben.

Computeralgoritmen voor het oplossen van legpuzzels

door
Johan De Bock

Afstudeerwerk ingediend tot het behalen van de academische graad van burgerlijk ingenieur in de computerwetenschappen

Academiejaar 2002–2003

Universiteit Gent
Faculteit Toegepaste Wetenschappen

Promotor: prof. dr. ir. W. Philips
Co-promotor: prof. dr. ir. J. D'Haeyer

Samenvatting

Voor vele mensen is het oplossen van legpuzzels een intrigerende bezigheid. Daarom zijn er al een reeks van artikels verschenen die beschrijven hoe men met behulp van de computer deze legpuzzels automatisch kan oplossen. In deze thesis wordt een programma ontwikkeld dat startend van de individuele puzzelstukken van een legpuzzel een topologische oplossing van deze legpuzzel zoekt via verschillende basisstappen. De gebruikte algoritmen zijn deels eigen algoritmen en deels interpretaties van reeds bestaande algoritmen. Verder tonen we aan dat het algoritme om de passende delen van twee puzzelstukken te zoeken, ook kan gebruikt worden voor het zoeken van de passende delen van objecten met een irreguliere rand zoals gescheurde papierfragmenten. Hiermee leggen we dan de basis van een semi-automatische reconstructie methode voor allerlei objecten.

Trefwoorden: puzzel, matching, reconstructie, automatisering.

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

Hoofdstuk 1

Inleiding

Het oplossen van legpuzzels is vooral een leuke bezigheid op jongere leeftijd door de creativiteit die vereist is bij het zoeken van puzzelstukjes die in elkaar passen. Maar het is ook een zeer interessant probleem als we het bekijken vanuit een meer technisch standpunt: hoe kunnen we vertrekkend van de individuele puzzelstukken een legpuzzel automatisch oplossen via de computer. Hiervoor zijn een reeks van beeldverwerking, beeldanalyse en recompositie technieken nodig. Sinds 1964 zijn er sporadisch artikels verschenen die specifiek handelden over het oplossen van legpuzzels met behulp van computeralgoritmen. We hebben ons voor het thesisonderzoek vooral laten inspireren door [1], [2], [3] en [4]. Hiervan is [2] het recentste artikel waarvan wij kennis hebben.

De interesse van onderzoekers in dit onderwerp kunnen we enerzijds verklaren door de uitdaging van het probleem op zich en anderzijds door de verschillende toepassingen waarvan de technieken die men bij het oplossen van legpuzzels gebruikt de basis zijn. Deze toepassingen situeren zich vooral in het domein van de archeologie: de reconstructie van allerlei archeologische objecten zoals tegels, schalen, ... Hier kunnen we de algoritmen om passende delen van figuren te zoeken direct toepassen op de scherven. Ook bij forensisch onderzoek kunnen we deze algoritmen gebruiken: de reconstructie van in stukken gescheurde papieren. Zelfs in de medische sector vindt men een toepassing onder de vorm van het zoeken van passende moleculen.

Het hoofddoel dat we voorop gesteld hebben is het schrijven van een programma dat startend van de scans van de individuele puzzelstukken de topologische oplossing van een legpuzzel kan vinden. De verschillende hoofdstappen die we hierbij moeten doorlopen worden in de komende hoofdstukken beschreven.

Eerst moeten we een digitale representatie hebben van de individuele puzzelstukken onder de vorm van een contour, dit wordt besproken in hoofdstuk 2. Eens we deze contour hebben is het zeer nuttig om een representatie van de contour te definiëren die we verder kunnen gebruiken als basis van de verschillende algoritmen, dit verkrijgen we door de benadering van de contour door middel van een polygoon. We baseren ons dan op de polygonale benadering om kleur- en vormkenmerken te extraheren. Deze polygonale benadering en kenmerkenextractie worden besproken in hoofdstuk 3. De vormextractie is een eigen interpretatie van de vormextractie in [1] en de kleurextractie is gebaseerd op de kleurextractie in [3] en [4]. De kenmerken worden dan gebruikt bij het zoeken van de passende delen van twee puzzelstukken, wat beschreven wordt in hoofdstuk 4. De resultaten van dit zoekproces worden dan gebruikt bij het zoeken naar de globale topologische oplossing voor een legpuzzel. De gebruikte oplosalgoritmen en de uiteindelijke resultaten

voor een paar legpuzzels vindt men terug in hoofdstuk 5. Enkele van de gebruikte algoritmen zijn gebaseerd op [2].

Het algoritme voor het zoeken van passende delen is niet specifiek geconstrueerd voor puzzelstukken, het kan gebruikt worden voor algemene figuren. Om dit te illustreren gebruiken we het in hoofdstuk 6 voor het zoeken van passende delen van papierfragmenten.

Uiteindelijk worden enkele besluiten getrokken uit het onderzoek van dit thesisonderwerp in hoofdstuk 7.

Hoofdstuk 2

Digitale acquisitie en contourextractie

Vooraleer over te gaan naar het oplossen van legpuzzels met computeralgoritmen, moeten we natuurlijk een digitale representatie hebben van van de verschillende puzzelstukken. Hierbij hebben we gekozen voor traditionele legpuzzels die makkelijk verkrijgbaar zijn. Deze puzzelstukken hebben we dan ingescand met een HP-Laserjet 4p op 300 dpi. Deze manier van werken is te verkiezen boven het artificieel aanmaken van puzzelstukken. Het inscannen van de puzzelstukken geeft een realistischer beeld: de algoritmen moeten dan aangepast zijn aan de reële problemen zoals scannerruis, vuiltjes, beschadigde stukken en uitstekende randvezels. De puzzelstukken zijn ingescand in totaal willekeurige posities omdat we niet op voorhand weten in welke positie het puzzelstuk zich uiteindelijk zal bevinden.

Eens we duidelijke digitale beelden hebben van de puzzelstukken, moet we de rand van een puzzelstuk op een scan kunnen extraheren. Met rand of contour bedoelen we in deze context een gesloten kromme in het tweedimensionaal vlak. Equivalent hiermee is het vereenvoudigen van het beeld van een puzzelstuk naar een beeld met slechts twee samenhangende gebieden, namelijk een voorgrondgebied voor het puzzelstuk en een achtergrondgebied voor de gekozen achtergrond. We hebben dus een algoritme nodig dat éénduidig een beeld scheidt in een samenhangende voor- en achtergrond. Het oppervlakte-uitbreiding algoritme is hiervoor uiterst geschikt. Dit algoritme wordt in de volgende paragraaf beschreven voor één puzzelstuk per scan. Daarna wordt meer de nadruk gelegd op het automatiseren van het extractie proces in de paragrafen “Automatisering met histogramanalyse” en “Meerdere figuren per scan: automatisering voor legpuzzels”.

2.1 Oppervlakte-uitbreiding algoritme

We nemen bij de komende uitleg aan dat de achtergrond een kleinere grijswaarde heeft dan de voorgrond. De gebruikte datastructuren en parameters van het algoritme voor een beeld met NxN pixels zijn de volgende:

- input:
  - *check* : NxN matrix van booleaanse variabelen die weergeven of de grijswaarde van een pixel al dan niet kleiner is dan een drempelgrijswaarde

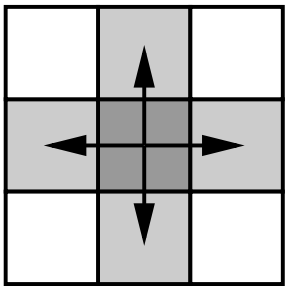
  - *start* : eerste pixel waarvan de buurpixels zullen onderzocht worden

- lokaal:
  - *queue* : queue van pixels waarvan de buurpixels gecontroleerd moeten worden

- output:
  - *segment* : NxN matrix van booleaanse variabelen die weergeven of een pixel al dan niet tot de achtergrond behoort, initieel zijn alle variabelen vals

De indices van de matrices corresponderen hierbij met de pixels van het tweedimensionale beeld.

Het algoritme werkt nu als volgt:
Voor een bepaald beeld wordt eerst een drempelgrijswaarde opgegeven. Deze drempelgrijswaarde is de scheiding tussen voor- en achtergrond uitgedrukt in grijswaarde. Voor elke pixel van het beeld wordt dan de grijswaarde berekend volgens deze formule: $grijswaarde = 0.2125 \times rood + 0.7154 \times groen + 0.0721 \times blauw$. Deze grijswaarde wordt gecontroleerd op het al dan niet kleiner zijn dan de opgegeven drempelgrijswaarde en het resultaat hiervan wordt opgeslagen in *check* zodat dit resultaat zeer snel gecontroleerd kan worden voor de verschillende pixels.
De pixel *start* wordt nu achteraan de *queue* geplaatst. Vanaf dan wordt telkens één pixel vooraan van de queue gehaald en van die pixel worden de buurpixels gezocht volgens het 4 buren principe weergegeven in figuur 2.1. De buurpixels die als corresponderende waarde



Figuur 2.1: Het 4 buren principe

van *check*, waar hebben, worden achteraan de *queue* geplaatst. Voor elke pixel die van de *queue* gehaald wordt, wordt de corresponderende waarde van *segment* op waar gezet. Deze *segment* matrix kan men dan ook gebruiken om te bepalen of een pixel al niet eerder behandeld is. De uiteindelijke scheiding van voor- en achtergrond vindt men dan in *segment*.
Het buren principe zorgt ervoor dat de achtergrond geleidelijk groter wordt en steeds een samenhangend oppervlak is, vandaar de naam oppervlakte-uitbreiding algoritme. Een voorbeeld van dit gradueel proces ziet men in figuur 2.2.

Op figuur 2.2 is ook te zien dat de achterkant van het puzzelstuk ingescand is. De eerste pogingen voor contourextractie werden gedaan op scans van de voorkanten van de puzzelstukken, maar hierbij bleek de hinder van de schaduw te groot te zijn om automatisering toe te laten. De achterkanten hebben een egale kleur die sterk verschillend is van

Figuur 2.2: Gradueel proces van het oppervlakte-uitbreiding algoritme

zwart, zodoende dat men makkelijk de schaduw kan beschouwen als een deel van de gebruikte zwarte achtergrond. Dit is niet mogelijk bij de voorkanten door het soms aanwezig zijn van zwart-componenten die identiek zijn aan de schaduw.

2.2 Automatisering met histogramanalyse

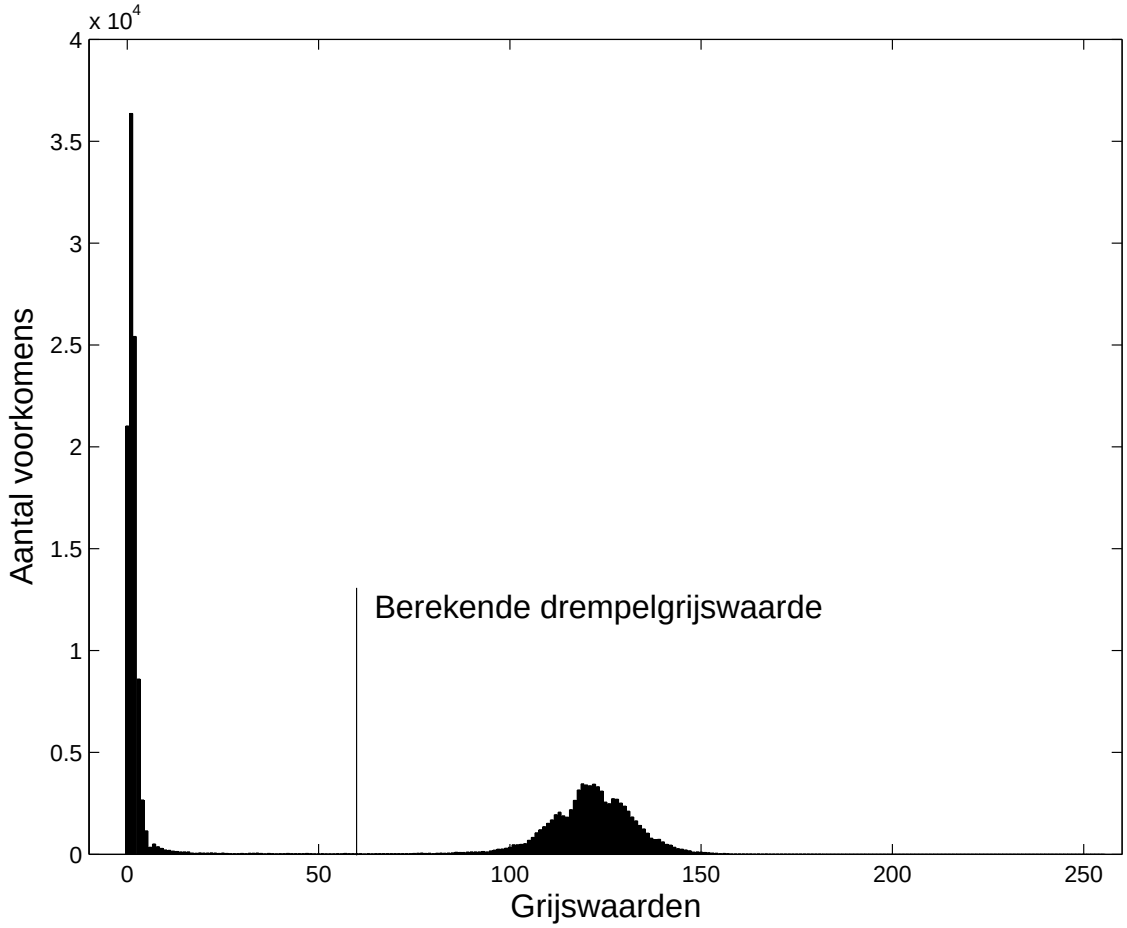
In de beschrijving van het oppervlakte-uitbreiding algoritme werd de drempelgrijswaarde voor scheiding van voor- en achtergrond verondersteld op voorhand gekend te zijn. Om automatisering toe te laten zou het dus zeer nuttig zijn om een goede drempelwaarde te berekenen uitgaande van de informatie die in het beeld van het puzzelstuk vervat zit.

Een makkelijke methode om dit te bewerkstelligen is een eenvoudige histogramanalyse. Tijdens het berekenen van de grijswaarden van alle pixels in het oppervlakte-uitbreiding algoritme wordt van elke van deze grijswaarden het geheel deel genomen. Deze getallen vallen dan in het bereik [0-255]. Voor elke gehele waarde uit dit bereik wordt dan bijgehouden hoeveel maal ze voorkwam. Uiteindelijk kan men deze tellingen grafisch weergeven in een histogram met 256 staven. In dit histogram bevinden zich twee maxima: één maximum voor de belangrijkste voorgrondgrijswaarde en één maximum voor de belangrijkste achtergrondgrijswaarde. Van de abscissen van deze twee maxima wordt dan het gemiddelde genomen en dit getal gebruikt men dan als drempelwaarde. Een voorbeeld van zo een histogram ziet men in figuur 2.3.

2.3 Meerdere figuren per scan: automatisering voor legpuzzels

Het is mogelijk om puzzelstuk per puzzelstuk in te scannen en telkens hiervan de contourextractie te doen. Dan heeft men direct per puzzelstuk de contour. Maar voor een groot aantal puzzelstukken is dit te tijdrovend. De oplossing is dus scannen van meerdere puzzelstukken per scan. Hierbij is een automatische extractie van de verschillende aparte contouren van de puzzelstukken per scan wenselijk.

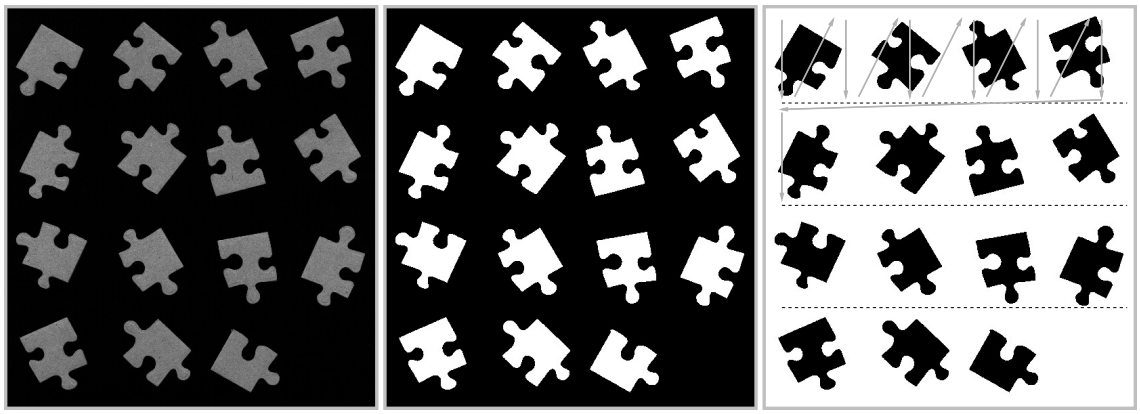
Per rij in de juiste oplossing van de legpuzzel wordt er een scan genomen. Hierbij worden de puzzelstukken van links naar rechts en van boven naar onder geplaatst, in de oorspronkelijke volgorde. Als we nu de contourextractie van de aparte puzzelstukken ook in deze volgorde doen, kunnen we exact de oorspronkelijke rij en kolom positie bijhouden van de verschillende puzzelstukken. Deze rij en kolom worden dan later gebruikt om de gevonden oplossing van de legpuzzel te controleren met de juiste oplossing. Dit is



Figuur 2.3: Berekening van de drempelgrijswaarde via histogramanalyse

weergegeven op het eerste beeld van figuur 2.4.

Om aan deze twee vereisten te voldoen passen we het oppervlakte-uitbreiding algoritme meerdere keren toe en wel op de volgende manier:
Eerst passen we het oppervlakte-uitbreiding algoritme op de volledige scan, wat men kan zien op het tweede beeld van figuur 2.4. Nu kunnen we de verschillende puzzelstukken apart extraheren door voor elk puzzelstuk een pixel te zoeken die zeker tot de voorgrond behoort en die als *start* pixel gebruiken bij het toepassen van het oppervlakte-uitbreiding algoritme met een *check* matrix die de inverse is van de eerder bekomen *segment* matrix van booleaanse waarden (invers: waar wordt vals en vals wordt waar). Wanneer men dan voor de bekomen nieuwe *segment* matrix de minima en de maxima van de pixels bijhoudt, kan men gemakkelijk een bounding box construeren die de volledige contour van het puzzelstuk bevat. Het voordeel van de extra toepassing van het oppervlakte-uitbreiding algoritme is het verwijderen van eventuele stofdeeltjes in de achtergrond. Het behoud van de volgorde van de puzzelstukken wordt bekomen door de *start* pixels voor de opeenvolgende puzzelstukken te zoeken op de manier geïllustreerd in het derde beeld van figuur 2.4: er wordt per rij van boven naar onder en van links naar rechts gezocht. De dikte van deze rijen is op voorhand ingesteld en natuurlijk afhankelijk van de grootte van



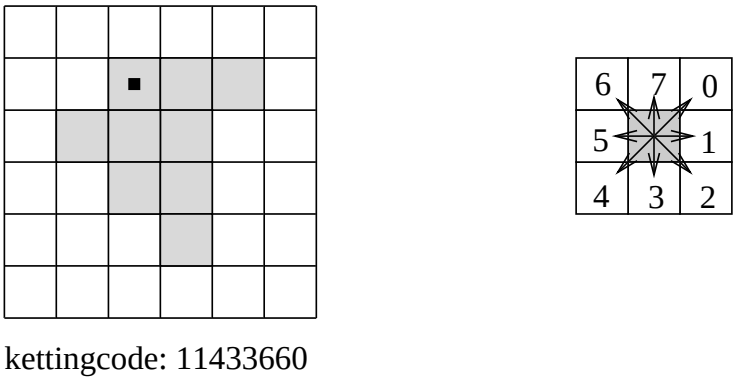
Figuur 2.4: Contourextractie voor meerdere puzzelstukken per scan

de puzzelstukken.

2.4 Contourcodering

Na de contourextractie is de contour van elk puzzelstuk éénduidig bepaald onder de vorm van een binair beeld: een samenhangend oppervlak dat het puzzelstuk voorstelt en daarrond de achtergrond. Dit zou men kunnen gebruiken als structuur om de contour permanent bij te houden voor verder gebruik, maar deze structuur bevat teveel informatie buiten de contour en is bovendien niet handig voor de algoritmen die verder in dit werk de contourinformatie gebruiken. Deze algoritmen lopen de contour sequentieel af. Men zou dan dus telkens als men de contour nodig heeft deze sequentieel moeten aflopen op het originele binaire beeld. Het is dus beter dit één maal te doen en dan deze info op te slaan in een geschikt formaat voor later gebruik in de algoritmen.

Zo een geschikt formaat vindt men onder de vorm van kettingcodes. Een kettingcode is een vector van gehele getallen in het bereik [0-7] die elke pixel van de contour beschrijft relatief tegenover de vorige pixel. Een voorbeeld van een eenvoudige figuur samen met zijn kettingcode vindt men in figuur 2.5. De mapping tussen de getallen en de buren is



Figuur 2.5: De kettingcode

ook opgenomen in de figuur. De contour van de figuur wordt dus beginnend vanaf een bepaald punt in wijzerzin afgelopen en naargelang de positie van de volgende pixel wordt overeenkomstig de burenmapping het correcte getal gekozen. Het beginpunt wordt samen met de vector van gehele getallen opgeslagen voor later gebruik.

Hoofdstuk 3

Kenmerkenextractie

Eens de extractie van de contour voltooid is, hebben we alles voor de volgende stap: de kenmerkenextractie. Bij het uiteindelijke algoritme voor het zoeken van passende contourdelen zal het nodig blijken om de contour te herleiden tot een reeks van punten op de contour. Deze punten worden dan gebruikt om verschillende kenmerken van de figuur te extraheren. De verschillende manieren waarop men deze punten kan vastleggen worden beschreven in de paragraaf "Polygonale benadering". De extractie van de vormkenmerken via de gekozen punten wordt uiteengezet in de paragraaf "Vormextractie" en de extractie van de kleurkenmerken wordt uiteengezet in de paragraaf "Kleurextractie". De uiteindelijke resultaten van deze extracties worden opgeslagen in vectoren die we de kenmerkvectoren noemen.

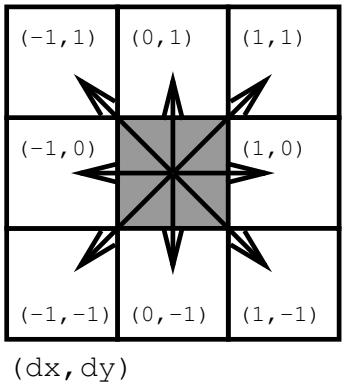
Voor het oplossen van legpuzzels is het nuttig om de vier kanten van de contour te kunnen onderscheiden, zodat algoritmes voor de globale oplossing van legpuzzels van deze extra info gebruik kunnen maken. Dit onderscheid tussen de 4 deelcontouren wordt verkregen door de scheidingspunten op de contour te zoeken. Deze scheidingspunten noemen we de hoekpunten van het puzzelstuk omdat ze gelegen zijn op een deel van de contour dat quasi een rechte hoek voorstelt. De extractie van deze vier hoekpunten wordt beschreven in de paragraaf "Hoekpuntextractie".

3.1 Polygonale benadering

Hetgeen we willen verkrijgen in de polygonale benadering zijn punten die we achteraf, bij het zoeken van passende delen van twee verschillende figuren, kunnen gebruiken om de kenmerken van de ene figuur te kunnen vergelijken met de kenmerken van de andere figuur. We moeten deze punten dus zo kiezen dat in het ideale geval van een perfecte ineenpassing van twee figuren, de punten behorend tot het passende deel van de twee figuren op dezelfde plaats liggen als we de figuren effectief ineenplaatsen. De makkelijkste manier om dit te verkrijgen is het kiezen van een reeks punten op de contour die op een vaste afstand van elkaar liggen. Voor beide figuren kiezen we dan dezelfde afstand. Hierbij wordt verondersteld dat de offset tussen de twee sets van punten van het passende deel nul is. Hoe men een offset die bijna nul is kan verkrijgen wordt uitgelegd in het hoofdstuk "Zoeken passende contourdelen". De logica van het op deze manier kiezen van de punten zal in dat hoofdstuk ook duidelijker worden.

Het berekenen van de afstanden tussen de punten van de contour wordt gedaan door

de kettingcode sequentieel af te lopen. Voor elke overgang worden de verplaatsing in de x-richting en in de y-richting op pixelniveau bijgehouden zoals op figuur 3.1 weergegeven. Het eerste vastgelegde punt wordt het beginpunt van de kettingcode. Dan berekenen we



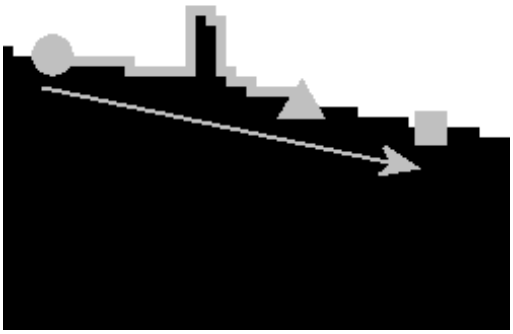
Figuur 3.1: Vertaling van de kettingcode naar verplaatsingen

voor elke volgende pixel van de contour de afstand tot het vorige al vastgelegde punt. Wanneer de opgegeven afstand overschreden wordt door de berekende afstand, wordt de laatst behandelde pixel van de contour gekozen als volgende punt van de polygonale benadering. Afstanden op pixelniveau worden hierbij berekend met de klassieke Euclidische afstand: $\sqrt{dx^2 + dy^2}$. Naargelang de interpretatie van het begrip afstand kunnen we nu op twee verschillende manieren deze afstand berekenen via de berekende dx en dy voor elke pixelovergang:

- **Afstand naar de volgende pixel expliciet langs de contour**: de afstand wordt berekend als de som van de individuele afstanden van de elementaire pixelovergangen, 1 voor horizontale en verticale overgangen en $\sqrt{2}$ voor diagonale overgangen.
  Samengevat: $\sum(\sqrt{dx^2 + dy^2})$

- **Vogelvlucht afstand naar de volgende pixel**: hier wordt de som van de dx en de som van de dy apart bijgehouden en met deze sommen wordt dan de afstand berekend. Dit geeft een rechtstreekse afstand van een pixel tot een pixel en niet de afstand langs de contour.
  Samengevat: $\sqrt{(\sum dx)^2 + (\sum dy)^2}$

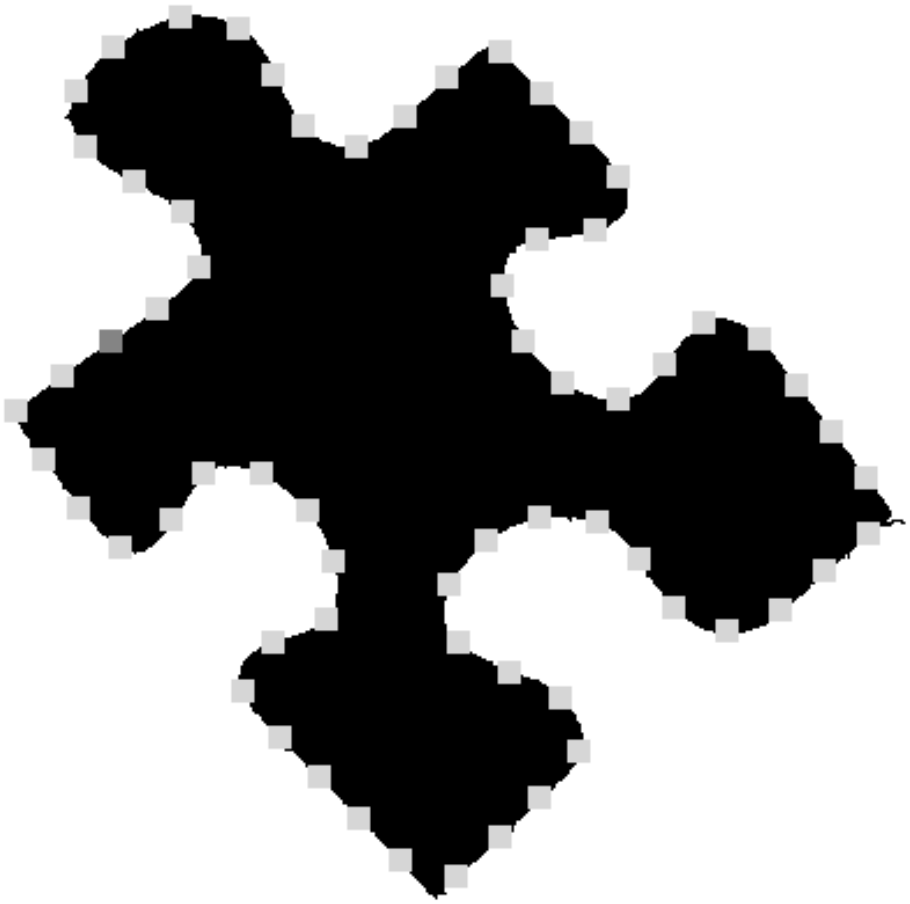
Het voordeel van de tweede benadering tegenover de eerste wordt weergegeven in figuur 3.2. Deze figuur stelt een stukje van een rechte rand van een puzzelstuk voor, echter is er door het meescannen van een vezel een uitstulping ontstaan bij de contourextractie. Deze uitstulping is in het beeld van een ideale rand natuurlijk niet aanwezig en het is dus zeer wenselijk dat de uitstulping de berekeningen voor gelijke afstanden niet verstoort. De twee methoden starten beide aan de bol. De eerste methode loopt dan langs de rand tot aan de driehoek en neemt hierbij de lengte van de uitstulping mee in de afstandsberekening. De tweede methode is weergegeven door de pijl tot aan het vierkant: hier wordt de lengte van de uitstulping niet meegenomen door het nemen van de vogelvlucht afstand. De tweede methode is dus duidelijk beter dan de eerste omdat de tweede geen effect ondervindt van de uitstulping en de eerste wel. Als men zulke ruisobjecten toch zou



Figuur 3.2: Vergelijking van twee methoden om de afstand langs de contour te meten

meenemen in de afstandsberekeningen, zouden we later bij de kenmerkenextracties een propagatie van fouten krijgen.

Het uiteindelijke resultaat van de polygonale benadering is weergegeven in figuur 3.3. Het donkerder vierkant geeft het beginpunt aan van de kettingcode en dus ook het eerste



Figuur 3.3: De polygonale benadering

punt van de polygonale benadering.

3.2 Vormextractie

Na de polygonale benadering, kunnen we ons toeleggen op de extractie van vormkenmerken. Eerst moeten we ons bezinnen over bruikbare vormkenmerken in de paragraaf "theoretisch wiskundige vormbeschrijving", dan worden deze vormkenmerken vertaald naar hun discrete implementaties die gebruik maken van de eerder gekozen punten op de contour. Deze implementaties worden beschreven in de paragraaf "discretisering vormkenmerken".

3.2.1 Theoretisch wiskundige vormbeschrijving

Met vormbeschrijving bedoelt men meestal globale de vormbeschrijving van een figuur: die vorm is eerder rond of die vorm is eerder hoekig. Maar hier spreken we over de lokale vormbeschrijving langs de contour van de figuur, want alleen deze kunnen we betrouwbaar gebruiken om passende figuren te zoeken. We moeten dus een wiskundig model hebben dat deze lokale vorm langs de contour beschrijft en bovendien rotatie-onafhankelijk is. Deze rotatie-onafhankelijkheid is noodzakelijk omdat de puzzelstukken in totaal willekeurige posities zijn ingescand. Het zou veel te omslachtig zijn om een rotatie-afhankelijk model te gebruiken en dan voor een bepaalde set van rotaties van een puzzelstuk telkens dit model toe te passen om toch een bijna onafhankelijkheid te krijgen. Wiskundige entiteiten die deze rotatie-onafhankelijkheid bezitten en langs een contour kunnen berekend worden zijn de volgende:

- **Richtingverandering** afgeleid naar de booglengte: $\frac{d\varphi}{ds}$
- **Koorde afstand** afgeleid naar de booglengte: $\overline{\frac{d\vec{r}}{ds}}$ (waarbij $\overline{\ }$ de norm voorstelt)

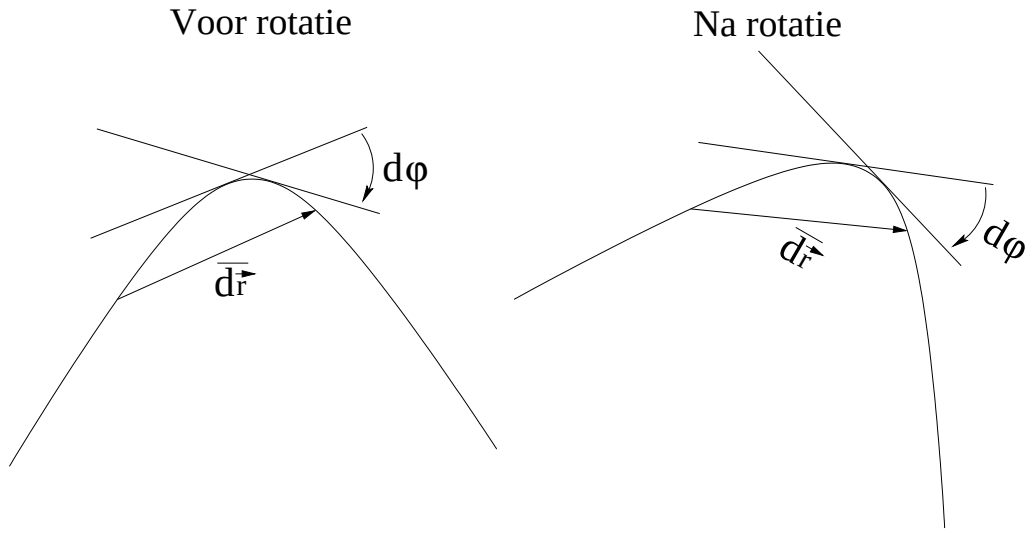
De rotatie-onafhankelijkheid is duidelijk weergegeven in figuur 3.4.

3.2.2 Discretisering vormkenmerken

In een algoritme is het niet aangewezen om deze zonet beschreven continue kenmerken te gebruiken op een continue manier door bijvoorbeeld het interpoleren van krommen. Dit zou teveel rekenkracht eisen. Om dus de continue vormkenmerken te kunnen gebruiken op een efficiënte manier, moeten we ze vertalen naar een discrete implementatie die we makkelijk kunnen gebruiken in een algoritme. Deze vertaling wordt beschreven in de volgende twee punten.

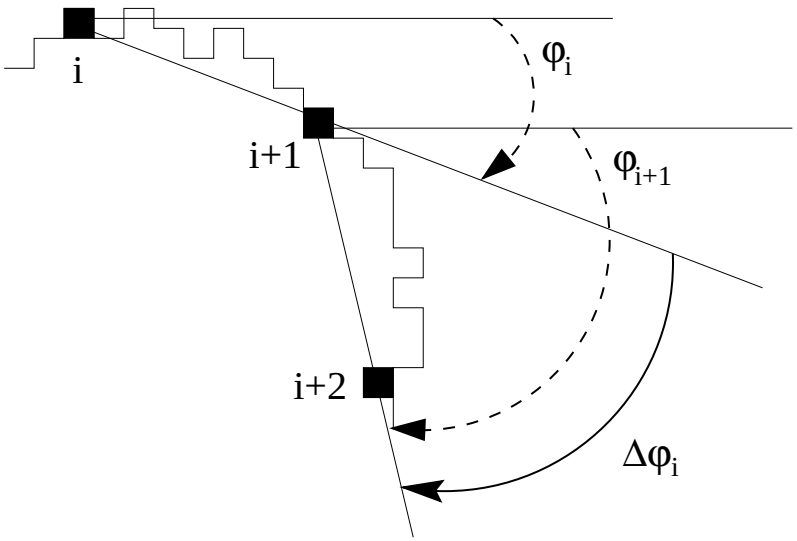
Differentiële hoeken

De richtingverandering afgeleid naar de booglengte of kromming vertaalt zich naar differentiële hoeken. Zoals eerder vermeld worden de punten van de polygonale benadering gebruikt voor de discrete implementatie van de vormkenmerken. De discrete benadering van $\frac{d\varphi}{ds}$ is $\frac{\Delta\varphi}{\Delta s}$, waarbij Δ een differentie is gebaseerd op de punten van de polygonale benadering. Omdat deze punten voor alle puzzelstukken op een vaste afstand van elkaar gekozen zijn, kunnen we de noemer Δs buiten beschouwing laten. Uiteindelijk krijgen we dus $\Delta\varphi$ wat we de differentiële hoek noemen.



Figuur 3.4: Rotatie-onafhankelijke wiskundige entiteiten

De berekening van de differentiële hoek is weergegeven in figuur 3.5. Sequentieel wordt er per paar punten een hoek berekend in het interval $]-180, 180]$ graden. Het aantal punten liggend tussen het paar punten kan via een parameter ingesteld worden (op figuur 3.5 is dit aantal 0). Zo kan men door het tezamen regelen van de afstand tussen de punten van de polygonale benadering en het aantal punten tussen de paren waarvan met de hoek berekent het gewenste detailniveau in de vormbeschrijving van de contour bereiken zonder hierbij hoeken te verkrijgen die te ruisafhankelijk zijn. De differentiële hoeken worden uiteindelijk verkregen door de opeenvolgende berekende hoeken van elkaar af te trekken: $\Delta\varphi = \varphi_{i+1} - \varphi_i$. Deze rij van getallen wordt uiteindelijk opgeslagen in een kenmerkvector



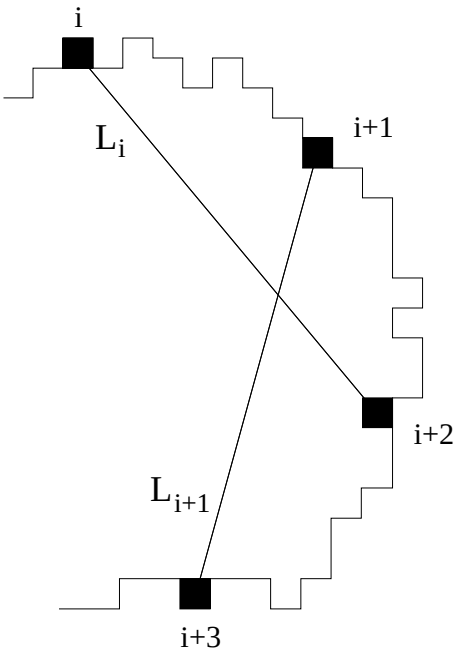
Figuur 3.5: Berekening van de differentiële hoek

van reële getallen.

Koorde afstanden

De koorde afstand afgeleid naar de booglengte vertaalt zich naar koorde afstanden. De discrete benadering van $\frac{\overrightarrow{dr}}{ds}$ is $\frac{\overrightarrow{\Delta r}}{\Delta s}$, waarbij Δ een differentie is gebaseerd op de punten van de polygonale benadering. Omdat deze punten voor alle puzzelstukken op een vaste afstand van elkaar gekozen zijn, kunnen we de noemer Δs buiten beschouwing laten. Uiteindelijk krijgen we dus $\overrightarrow{\Delta\ r}$ wat een koorde afstand is.

De berekening van de koorde afstand is weergegeven in figuur 3.6. Hier is de koorde afstand weergegeven door de letter L. Sequentieel wordt er tussen een paar punten een



Figuur 3.6: Berekening van de koorde afstand

afstand berekend: de koorde afstand. Het aantal punten liggend tussen het paar punten kan via een parameter ingesteld worden (op figuur 3.6 is dit aantal 1). Zo kan men door het tezamen regelen van de afstand tussen de punten van de polygonale benadering en het aantal punten tussen de paren waarvan met de hoek berekent het gewenste detailniveau in de vormbeschrijving van de contour bereiken zonder hierbij afstanden te verkrijgen die te ruisafhankelijk zijn. De bekomen reeks afstanden wordt uiteindelijk opgeslagen in een kenmerkvector van positieve reële getallen.

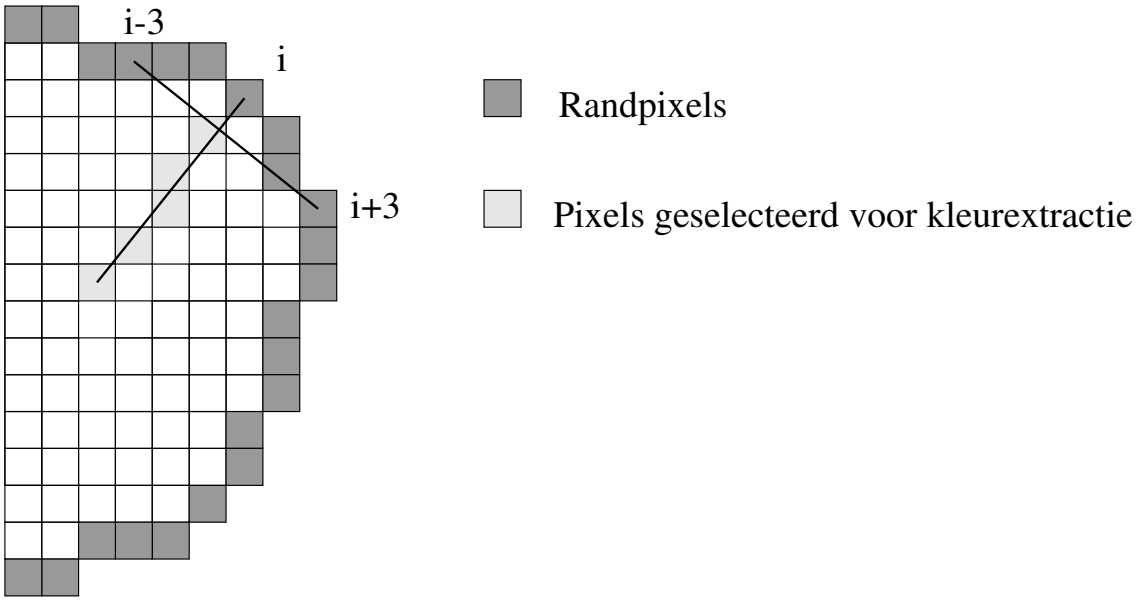
3.3 Kleurextractie

Buiten de vorm kan ook de kleur van de rand van figuren belangrijke informatie bevatten over het al dan niet passen van twee figuren. Omdat het bij de puzzelstukken zoals eerder vermeld te moeilijk was (door de schaduw) om op een snelle manier de scan van

de voorkant te gebruiken om de rand te extraheren, is bij het oplossen van de puzzel de kleurinformatie (die zich op de voorkant bevindt) niet gebruikt. De voorbeelden die in deze paragraaf zullen voorkomen zijn dan ook toegepast op papierfragmenten waarbij men geen last heeft van schaduweffecten. Hierbij is de originele scan van de gekleurde voorkant gebruikt voor de contourextractie, zodat we later makkelijk de gekleurde randpixels uit diezelfde scan kunnen halen.

Zeer belangrijk bij deze kleurextractie is dat ze als resultaat ook een vector van reële getallen moet hebben die informatie geeft over de randkleur in de buurt van polygonale punten. Zo heeft men dan voor elk polygonaal punt één of meerdere reële waarden die informatie bevatten over de lokale contourvorm en één reële waarde die informatie bevat over de lokale randkleur. Als we deze perfecte positie overeenkomst hebben tussen de kenmerkvectoren van vorm en kleur kunnen we op een makkelijke manier de passende contourdelen zoeken.

Op figuur 3.7 is afgebeeld hoe we te werk gaan bij de kleurextractie om aan de gestelde eisen te voldoen. Na de contourextractie van een figuur op een scan die kleurinformatie

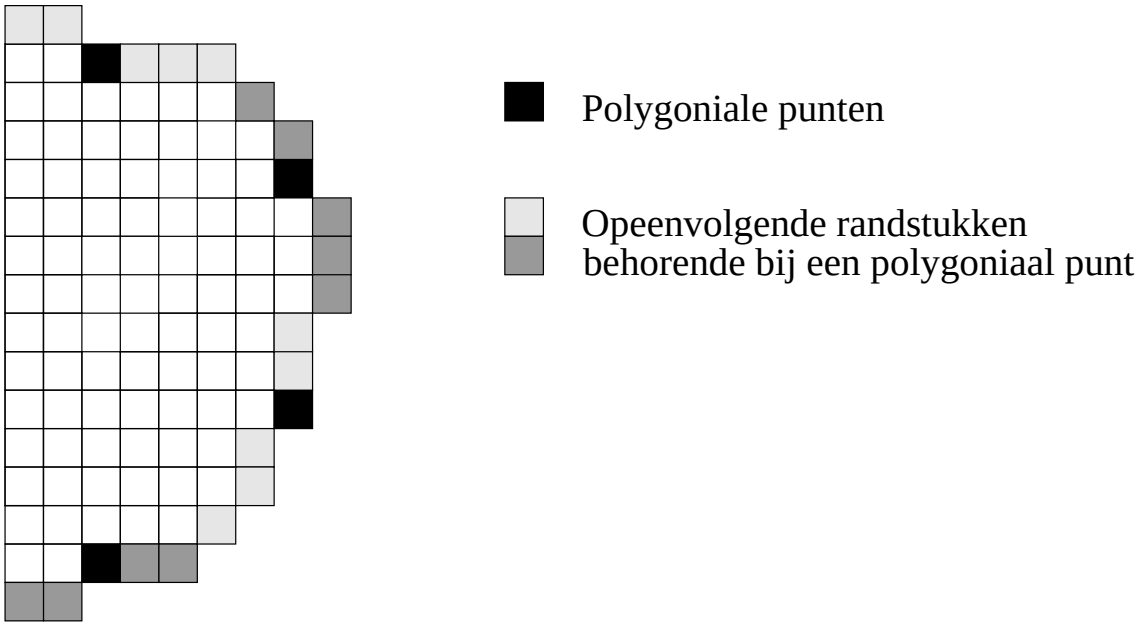


Figuur 3.7: De kleurextractie op pixelniveau

bevat wordt de contour opnieuw afgelopen langs de verkregen kettingcode. Per pixel i van de rand wordt een lijnstuk bepaald tussen pixel $i - k$ en pixel $i + k$. Dan wordt een lijnstuk bepaald dat loodrecht staat op het eerste lijnstuk, als beginpunt pixel i heeft, binnen de figuur gelegen is en een vaste norm heeft. Door middel van het Bresenham algoritme wordt dit lijnstuk benaderd op pixel niveau. Nu wordt het gemiddelde van de reële grijswaarden van de verkregen pixels berekend en dit reëel getal wordt opgeslagen in een vector. Deze pixels zijn natuurlijk uit de originele gekleurde scan gehaald. We krijgen dus uiteindelijk een vector met voor elke pixel i van de rand een reëel getal dat de kleurinformatie weergeeft op die positie. Deze vector wordt samen met de kettingcode opgeslagen in een structuur omdat deze logisch gezien op het zelfde pixelniveau staan.

Zoals bij de vormextractie moeten we nu nog de kleurinformatie bepalen op polygonaal

niveau. Hiervoor hebben we bij de bepaling van de polygonale punten de posities van deze punten bijgehouden op de kettingcode. Deze kunnen we nu gebruiken om de vector met de kleurinformatie op pixelniveau op te splitsen in gelijke gebieden rond de polygonale punten zoals in figuur 3.8 is afgebeeld. Voor deze gebieden wordt telkens het gemiddelde genomen



Figuur 3.8: De kleurextractie op polygonaal niveau

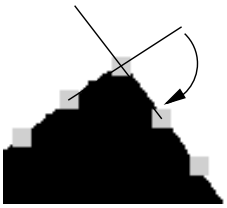
van de corresponderende reële getallen (in de vector met kleurinformatie op pixelniveau) en dit resultaat wordt opgeslagen op de juiste positie in de kenmerkvector van de kleur zodat we een perfecte positie overeenkomst hebben tussen vorm en kleur. Het uiteindelijke resultaat is dus een kenmerkvector van positieve reële getallen.

3.4 Hoekpuntextractie

Zoals in de korte inleiding van dit hoofdstuk beschreven werd, is het noodzakelijk om de vier hoekpunten te vinden van een puzzelstuk om de contour te splitsen in de vier kanten van het puzzelstuk. We baseren ons hierbij op het feit dat binnen bepaalde grenzen het hoekpunt een rechte hoek voorstelt. We moeten een soort algoritme vinden dat de rechte hoeken die de contour bevat lokaliseert. De differentiële hoeken berekend via de polygonale benadering zijn hiervoor het perfecte startpunt omdat ze de vorm in hoeken uitdrukt.

Een voorbeeld van een willekeurige polygonale benadering van een rechte hoek vindt men in figuur 3.9. Als we meerdere polygonale benaderingen maken door het startpunt van de polygonale benadering in de kettingcode telkens met één positie te verschuiven, dan is er voor elke rechte hoek wel een polygonale benadering waarvan één van de punten op het uiterste puntje van de rechte hoek gelegen is, zoals op figuur 3.9 weergegeven. De differentiële hoek die aangeduid is op de figuur is nu ongeveer -90 graden en de differentiële hoeken ervoor en erna zijn bij benadering 0 graden. Nu controleren we alle

Figuur 3.9: Voorbeeld van een willekeurige polygonale benadering van een rechte hoek



Figuur 3.10: Voorbeeld van een ideale polygonale benadering van een rechte hoek

bekomen sequenties van differentiële hoeken op het volgende patroon:

$$|h(i-1)| < \epsilon_1 \ , \ |h(i)+90| < \epsilon_2 \ , \ |h(i+1)| < \epsilon_3$$

met h(i) de differentiële hoek van positie i

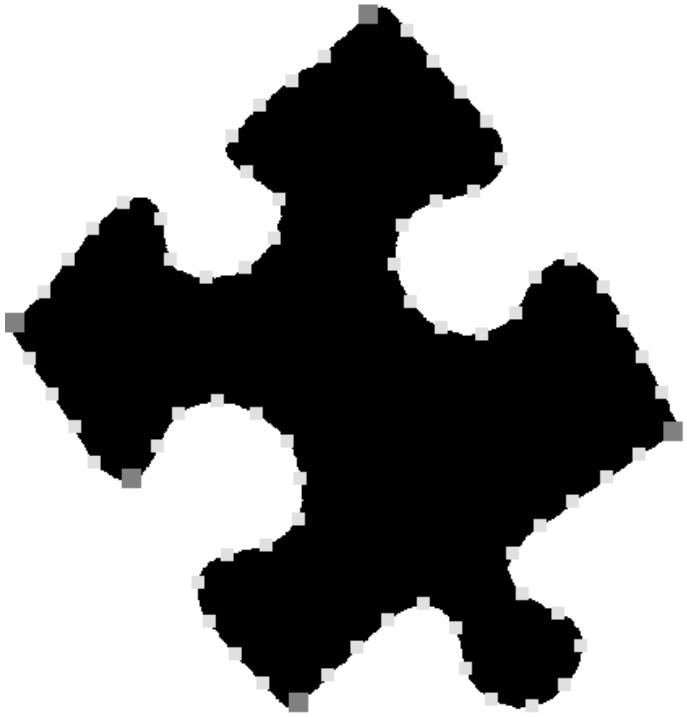
Diegene die er aan voldoen binnen de vastgelegde drempels beschouwen we als één van de vier te vinden hoeken. Om zeker te zijn dat de juiste vier hoeken zeker binnen deze drempels vallen, kiezen we de drempels voldoende groot. Het resultaat daarvan ziet men in figuur 3.11. De vier hoekpunten zijn bij benadering juist bepaald, maar één punt van één van de inkepingen is ook beschouwd als een hoekpunt. Dit moeten we op één of andere manier kunnen verhelpen.

De oplossing hiervoor is het zoeken van de inkepingen en uitsteeksels van het puzzelstuk. We doen dit door terug de sequenties van differentiële hoeken te onderzoeken op de volgende eigenschap:

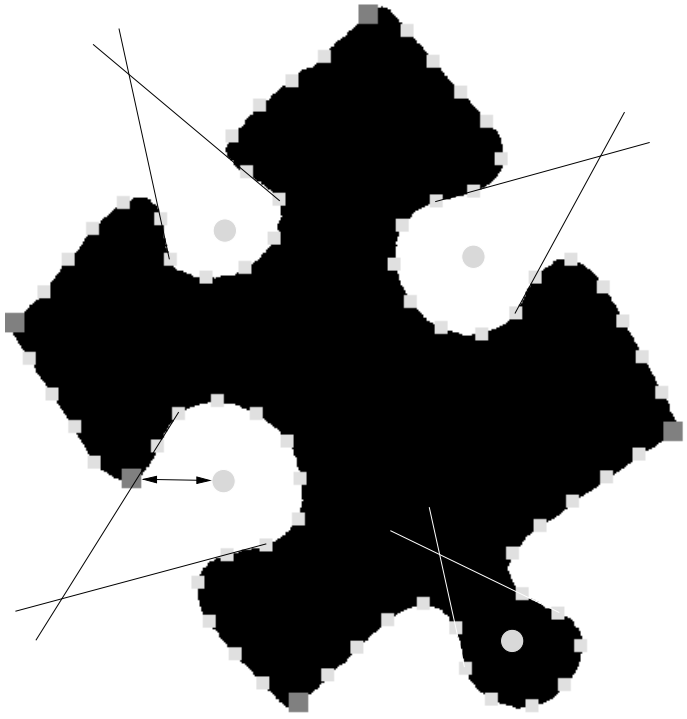
$$\left| \sum_{k=0}^{n} h(i+k) \right| > 180$$

met h(i) de differentiële hoek van positie i

Wat men hier dus eigenlijk doet is de totale hoekverandering berekenen over $n+1$ punten en controleren of deze groter is dan 180 graden of kleiner is dan -180 graden. Als men de parameter n klein genoeg neemt zal alleen bij de inkepingen het geval groter dan 180 graden waar zijn en zal alleen bij de uitsteeksels het geval kleiner dan -180 graden waar zijn. Dit proces is weergegeven in figuur 3.12. Het voldoen aan groter zijn dan 180 graden of kleiner zijn dan -180 graden is hier dus weergegeven door het snijden van de getekende lijnstukken. De cirkels geven hier het midden aan tussen het begin en het einde van een sequentie die voldoet aan het patroon, ze geven de centrale positie aan van de inkepingen en uitsteeksels. Uiteindelijk controleren we voor alle gevonden hoekpunten of ze ver genoeg liggen van deze centrale posities. De verkeerde hoekpunten zullen altijd dicht bij zo een centrale positie liggen en worden dus uit de gevonden set verwijderd. Zo is men dus in staat om de vier hoekpunten perfect te vinden.



Figuur 3.11: De gevonden hoekpunten



Figuur 3.12: Het zoeken van de inkepingen en uitsteeksels

Hoofdstuk 4

Zoeken passende contourdelen

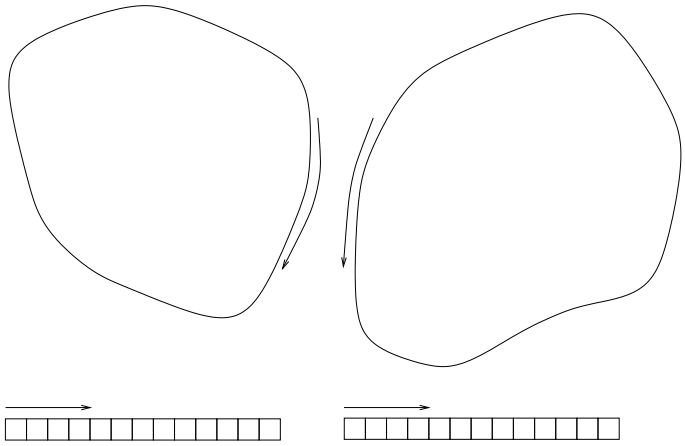
In het vorige hoofdstuk is beschreven hoe men uit één figuur drie kenmerkvectoren extraheert die de volledige contour bestrijken. In dit hoofdstuk zal een algoritme beschreven worden dat van twee figuren met een willekeurige contour (dus niet alleen puzzelstukken) via deze drie kenmerkvectoren de beste passende delen van de contouren zoekt en er een kost aan geeft in de vorm van een reëel getal. Eerst wordt uitgelegd op welke manier met de kenmerkvectoren zal gebruiken in de paragraaf "Gebruik van de kenmerkvectoren". Eens we dit weten wordt het algemene algoritme uitgelegd in de paragraaf "Benaderende stringmatching" en uiteindelijk wordt beschreven hoe met dit algoritme sterk kan versnellen in de paragraaf "Optimalisatie van het algoritme". De Engelse term "match" wordt verder in het hoofdstuk gebruikt in de betekenis van een passend deel.

4.1 Gebruik van de kenmerkvectoren

Bij één van de twee figuren moeten de kenmerkvectoren niet aangepast worden en kunnen we ze dus als directe input gebruiken bij het benaderende stringmatching algoritme. Bij de andere figuur moeten dan wel een paar aanpassingen gedaan worden aan de kenmerkvectoren vooraleer we ze kunnen gebruiken als input van het algoritme. Deze aanpassingen en de reden ervoor worden nu uitgelegd:
Het doel van het benaderende stringmatching algoritme is de twee best gelijkende stukken op beide contouren te vinden. We zullen dus de kenmerkvectoren van beide contouren sequentieel moeten aflopen en telkens stukken van de vectoren moeten vergelijken met elkaar. Als we dit symbolisch voorstellen op de figuren krijgen we figuur 4.1. Hier zien we duidelijk dat als één van de figuren in wijzerzin doorlopen wordt, dat dan de andere in tegenwijzerzin moet doorlopen worden. Aangezien tijdens het aanmaken van de kenmerkvectoren de figuren telkens in wijzerzin zijn doorlopen, moet voor alle kenmerkvectoren van één van de twee figuren de inhoud volledig omgedraaid worden op volgende wijze:

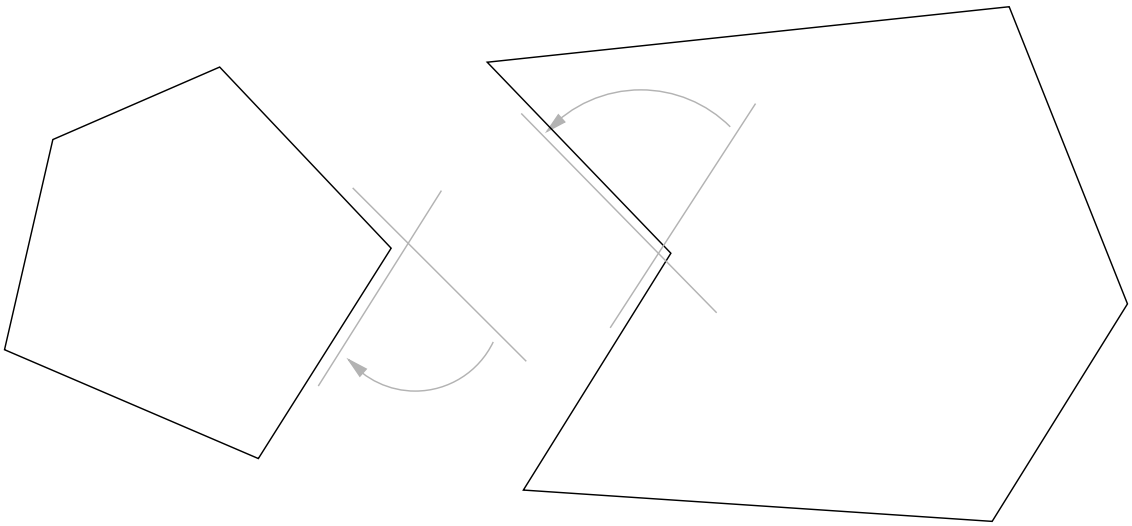
```
for ( int i=0 ; i<n ; i++ ) w[i]=v[n-1-i];

met w :  de nieuwe kenmerkvector
    v :  de oude kenmerkvector
    n :  lengte van de vectoren
```

Voor de kleur kenmerkvector en koorde afstand kenmerkvector is dit genoeg om in het ideale geval van een perfecte ineenpassing, volledig dezelfde deelvectoren te hebben op de



Figuur 4.1: Het afzoeken van de contour bij de benaderende stringmatching

plaats van de perfecte ineenpassing. Bij de differentiële hoek kenmerkvector moeten we nog iets extra doe om dit te bereiken wat op figuur 4.2 is aangetoond. Hier ziet men twee



Figuur 4.2: Verklaring van de omkering van het teken bij de differentiële hoek

figuren die op de beschouwde positie perfect ineenpassen. Doordat bij beide figuren de differentiële hoeken zijn berekend tijdens het in wijzerzin aflopen van de contour, is de differentiële hoek van de twee figuren op de beschouwde positie gelijk op het teken na. Dit betekent dat men bij de differentiële hoek kenmerkvector naast het omdraaien van de inhoud, ook het teken van de individuele reële getallen moet omkeren.
Nu kunnen we de originele kenmerkvectoren van de ene figuur en de aangepaste kenmerkvectoren van de andere figuur gebruiken als input van de benaderende stringmatching.

4.2 Benaderende stringmatching

We zullen in deze paragraaf eerst de algemene code van het benaderende stringmatching algoritme geven en dan de belangrijkste stappen uitleggen:

```
gebruikte datastructuren:
typedef struct{
         int lengte;          // lengte van de vectoren
         float diffhoek[100]; // vector van differentiele hoeken
         float koorde[100];   // vector van koorde afstanden
         float kleur[100];    // vector van grijswaarden
         } Contourinfo;

input:
float c1; // gewicht voor de differentiele hoeken
float c2; // gewicht voor de koorde afstanden
float c3; // gewicht voor de grijswaarden
int minl; // minimale matchlengte
int maxl; // maximale matchlengte
int cfactor; // compensatie factor bij variabele lengte
Contourinfo fig1,fig2; // de kenmerkvectoren voor de figuren

algoritme:
float best[100]; // bijhouden van de laagste kost per lengte
int pos1[100];   // bijhouden van de beste positie in figuur 1
int pos2[100];   // bijhouden van de beste positie in figuur 2

//initialisatie met 'oneindige' waarde:
for ( int i=minl-1 ; i<maxl ; i++) best[i]=100000;

for ( int i=0 ; i<fig1.lengte ; i++ ) {
   for ( int j=0 ; j<fig2.lengte ; j++ )
      float kost=0;
      for ( int l=0 ; l<minl-1 ; l++) {
         kost+=c1*fabs( fig1.diffhoek[ (i+l) % fig1.lengte ]
                      -fig2.diffhoek[ (j+l) % fig2.lengte ] )
              +c2*fabs( fig1.koorde[ (i+l) % fig1.lengte ]
                      -fig2.koorde[ (j+l) % fig2.lengte ] )
              +c3*fabs( fig1.kleur[ (i+l) % fig1.lengte ]
                      -fig2.kleur[ (j+l) % fig2.lengte ] );
         }

      for ( int l=minl-1 ; l<maxl ; l++ ) {
         kost+=c1*fabs( fig1.diffhoek[ (i+l) % fig1.lengte ]
                      -fig2.diffhoek[ (j+l) % fig2.lengte ] )
              +c2*fabs( fig1.koorde[ (i+l) % fig1.lengte ]
                      -fig2.koorde[ (j+l) % fig2.lengte ] )
              +c3*fabs( fig1.kleur[ (i+l) % fig1.lengte ]
```

```
                        -fig2.kleur[ (j+l) % fig2.lengte ] );

          if ( kost<best[l] ) {
             best[l]=kost;
             pos1[l]=i;
             pos2[l]=j;
             }
          }
       }
   }


float best_kost; // gegevens van de uiteindelijke beste match
int best_pos1;   // en output van het algoritme
int best_pos2;   //

if (minl!=maxl) {
   best_kost=100000; // initialisatie
   int bestl=-1;     //
   float kost;
   for (int l=minl-1 ; l<maxl ; l++ ) {
      kost=best[l]/pow(l+1,cfactor);
      if ( kost<best_kost ) {
         bestl=l;
         best_kost=kost;
         }
      }
   best_kost=best;
   best_pos1=data.pos1[bestl];
   best_pos2=data.pos2[bestl];
   } else {
   best_kost=best[minl-1]/minl;
   best_pos1=pos1[minl-1];
   best_pos2=pos2[minl-1];
   }
```

In het algoritme wordt dus per startpositie van een mogelijke match in de ene figuur en per startpositie van een mogelijke match in de andere figuur, voor de verschillende opeenvolgende matchlengtes l, de som van de absolute verschillen berekend tussen alle kenmerkvectoren. De belangrijkheid van de kenmerkvectoren kan men instellen via de gewichten. Vanaf de minimale matchlengte $minl$ tot de maximale matchlengte $maxl$ wordt per matchlengte de gegevens van de beste gevonden match bijgehouden (laagste kost). Bij één vaste matchlengte ($minl = maxl$) wordt het beste resultaat dan gedeeld door de lengte en dit geeft dan de uiteindelijke gemiddelde kost per lengte. Maar wanneer $minl$ verschillend is van $maxl$ is er sprake van een variabele lengte van de beste mogelijke match. Als men dan de gemiddelde kost per lengte van de verschillende matchlengtes

gewoonweg zou vergelijken, dan zou de kleinste matchlengte natuurlijk altijd de laagste kost geven. Daarom is het mogelijk de langere matchlengtes te bevoordelen via de parameter $cfactor$. De kost voor een bepaalde lengte wordt dan als volgt berekend: $\frac{best[l]}{(l+1)^{cfactor}}$, bij een $cfactor > 1$ kan men dan dus de grotere matchlengtes bevoordelen.

Het is nu nog altijd mogelijk dat er een "offset" bestaat tussen punten van de polygonale benaderingen van de twee figuren op de plaats van de beste match. Om deze offset te verminderen, wordt het benaderende stringmatching algoritme meerdere keren uitgevoerd voor een reeks van kenmerkvectoren (in plaats van één maal voor één set van kenmerkvectoren). Deze reeks wordt bekomen door het startpunt van de polygonale benadering in de kettingcodes bij beide figuren telkens met één positie te verschuiven en dan de kenmerkvectoren te berekenen met de nieuwe polygonale benadering. Hierdoor wordt de grootst mogelijke offset telkens iets kleiner per extra benadering: we krijgen meer en meer "detail" in het matchingproces. Het aantal keer dat dit gedaan wordt kan ingesteld worden met een parameter die we de detailparameter noemen.

4.3 Optimalisatie van het algoritme

In het algoritme beschreven in de vorige paragraaf zijn er vele mogelijkheden tot optimalisatie. Omdat het zoeken van de passende contourdelen procentueel een groot deel van de totale tijd van het volledige programma in beslag neemt loont het dus zeker de moeite om het sterk te optimaliseren. We zullen in deze paragraaf eerst de geoptimaliseerde code van het benaderende stringmatching algoritme geven en dan de veranderingen tegenover de niet geoptimaliseerde code uitleggen:

```
gebruikte datastructuren:
typedef struct{
         int lengte;          // lengte van de vectoren
         float diffhoek[300]; // vector van differentiele hoeken
         float koorde[300];   // vector van koorde afstanden
         float kleur[300];    // vector van grijswaarden
         } Contourinfo;

input:
float& algemeen_best; //de laagste kost over alle oproepen van het
                      //algoritme, dus ook output van het algoritme
float c1; // gewicht voor de differentiele hoeken
float c2; // gewicht voor de koorde afstanden
float c3; // gewicht voor de grijswaarden
int minl; // minimale matchlengte
int maxl; // maximale matchlengte
int cfactor; // compensatie factor bij variabele lengte
Contourinfo fig1,fig2; // de kenmerkvectoren voor de figuren

algoritme:
float best[100]; // bijhouden van de laagste kost per lengte
int pos1[100];   // bijhouden van de beste positie in figuur 1
int pos2[100];   // bijhouden van de beste positie in figuur 2
```

```
// de kenmerkvectoren 2 maal kopieren:
int size=sizeof(float)*fig1.lengte;
memcpy( &fig1.diffhoek[ shape.lengte ] , fig1.diffhoek , size );
memcpy( &fig1.diffhoek[ shape.lengte*2 ] , fig1.diffhoek , size );
memcpy( &fig1.koorde[ shape.lengte ] , fig1.koorde , size );
memcpy( &fig1.koorde[ shape.lengte*2 ] , fig1.koorde , size );
memcpy( &fig1.kleur[ shape.lengte ] , fig1.kleur , size );
memcpy( &fig1.kleur[ shape.lengte*2 ] , fig1.kleur , size );

//initialisatie met 'oneindige' waarde:
for ( int i=minl-1 ; i<maxl-1 ; i++) best[i]=100000;

best[maxl-1]=algemeen_best;

// matrix om de op voorhand berekende absolute verschillen in op te slaan:
float index[100][200];

for ( int i=0 ; i<fig1.lengte ; i++ ) {
   for ( int j=0,int i_plus_j=i ; j<fig2.lengte ; j++,i_plus_j++ ) {
      index[i][j]=c1*fabs( fig1.diffhoek[ i_plus_j ]
                          -fig2.diffhoek[ j ] )
                 +c2*fabs( fig1.koorde[ i_plus_j ]
                          -fig2.koorde[ j ] )
                 +c3*fabs( fig1.kleur[ i_plus_j ]
                          -fig2.kleur[ j ] );
      }
}

size=sizeof(float)*fig2.lengte;
for ( int i=0,int j=fig2.lengte ; i<fig1.lengte ; i++,j++ ) {
   memcpy( &index[i][fig2.lengte] , index[ j%fig1.lengte ] , size );
   }

int stop;
for ( int i=0 ; i<fig1.lengte ; i++ ) {
   for ( int j=0 ; j<fig2.lengte ; j++ ) {
      stop=0;
      int kost=0;
      for ( int j_plus_l=j ; j_plus_l<minl-1+j ; j_plus_l++ ) {
         kost+=index[i][j_plus_l];
         if ( kost>=best[maxl-1] ) {
            stop=1;
            break;
            }
         }
      if ( !stop ) {
```

```
         for ( l=minl-1,j_plus_l=minl-1+j ; l<maxl ; l++,j_plus_l++ ) {
            kost+=index[i][j_plus_l];
            if ( kost>=best[maxl-1] ) {
               break;
               }
            if ( kost<best[l] ) {
               best[l]=kost;
               pos1[l]=(i+j) % fig1.lengte;
               pos2[l]=j;
               }
            }
         }
      }
   }

float best_kost; // gegevens van de uiteindelijke beste match
int best_pos1;   // en output van het algoritme
int best_pos2;   //

if (minl!=maxl) {
   best_kost=100000; // initialisatie
   int bestl=-1;     //
   float kost;
   for (int l=minl-1 ; l<maxl ; l++ ) {
      kost=best[l]/pow(l+1,cfactor);
      if ( kost<best_kost ) {
         bestl=l;
         best_kost=kost;
         }
      }
   best_kost=best;
   best_pos1=data.pos1[bestl];
   best_pos2=data.pos2[bestl];
   } else {
   best_kost=best[minl-1]/minl;
   best_pos1=pos1[minl-1];
   best_pos2=pos2[minl-1];
   }

if ( best[maxl-1]<algemeen_best ) {
   algemeen_best=best[maxl-1];
   }
```

Een algemene methode om vectoren cyclisch te doorlopen is het gebruik van modulo berekeningen (% berekeningen) bij het berekenen van de indexen. Deze modulo berekeningen zijn in de originele code veel gebruikt. Modulo berekeningen zijn echter zeer reken-

intensief, ze worden dus best vervangen door gewone indexen die een uitgebreide vector doorlopen. Deze uitgebreide vector wordt als volgt bekomen:

$$\begin{pmatrix} a & b & c \end{pmatrix} \rightarrow \begin{pmatrix} a & b & c & a & b & c & a & b & c & ... \end{pmatrix}$$

Het cyclisch herhalen is dus verplaatst van de indexen naar de inhoud van de vector. Het aantal keer dat de vector moet herhaalt wordt is afhankelijk van de hoogst mogelijke index. Deze uitbreidingen zijn in de geoptimaliseerde code toegepast bij de kenmerkvectoren van $fig1$ (via de *memcpy* functie).

De belangrijkste aanpassing tegenover de originele code is het op voorhand berekenen en indexeren van de absolute verschillen tussen de individuele elementen van de kenmerkvectoren van de twee figuren. In de originele code worden vele berekeningen tussen koppels individuele elementen herhaald. Stel dat het aantal punten van de polygonale benadering n is en maximale matchlengte $maxl = \frac{n}{4}$ (wat ongeveer overeen komt met één kant van een puzzelstuk), dan komt het totale aantal berekeningen op $\frac{n^3}{4}$ omdat elk koppel punten $maxl$ keer wordt herberekend. Door de berekeningen op voorhand te doen valt de factor $maxl$ weg en krijgen we dus n^2 berekeningen.
De berekeningen zijn op de volgende manier opgeslagen in de matrix $index$:
Hierbij is $fig1.lengte = 5$ en $fig2.lengte = 7$ genomen.

$$\begin{pmatrix} v(0,0) & v(1,1) & v(2,2) & v(3,3) & v(4,4) & v(0,5) & v(1,6) \\ v(1,0) & v(2,1) & v(3,2) & v(4,3) & v(0,4) & v(1,5) & v(2,6) \\ v(2,0) & v(3,1) & v(4,2) & v(0,3) & v(1,4) & v(2,5) & v(3,6) \\ v(3,0) & v(4,1) & v(0,2) & v(1,3) & v(2,4) & v(3,5) & v(4,6) \\ v(4,0) & v(0,1) & v(1,2) & v(2,3) & v(3,4) & v(4,5) & v(0,6) \end{pmatrix}$$

met v(k,l) het absolute verschil tussen de getallen met index k en l in de originele kenmerkvectoren

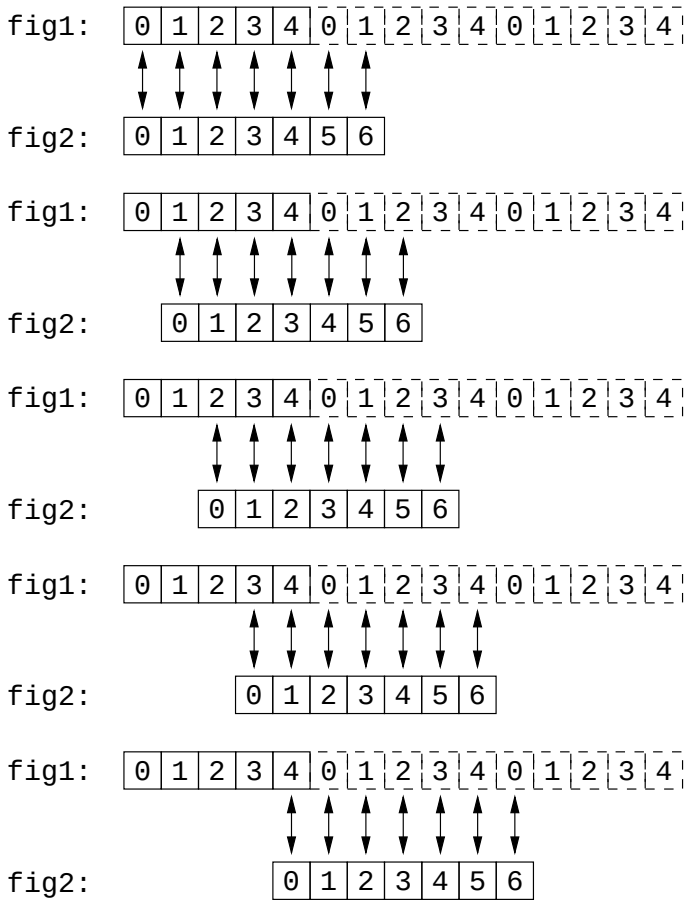
Per startpositie i in $fig1$ worden dus de opeenvolgende absolute verschillen genomen met $fig2$. Dit is weergegeven in figuur 4.3. Deze matrix bevat nu alle mogelijke koppels tussen de twee figuren, maar dit is niet genoeg om achteraf op een gemakkelijke manier de gegevens uit de matrix te halen. Hiervoor maken we van elke rij nog een kopie op de volgende manier:

$$\begin{pmatrix} rij\ 0 & rij\ 2 \\ rij\ 1 & rij\ 3 \\ rij\ 2 & rij\ 4 \\ rij\ 3 & rij\ 1 \\ rij\ 4 & rij\ 0 \end{pmatrix}$$

Uiteindelijk bekomt men dan de volgende matrix:

$$\begin{pmatrix} v(0,0) & v(1,1) & v(2,2) & v(3,3) & v(4,4) & v(0,5) & v(1,6) & v(2,0) & v(3,1) & v(4,2) & v(0,3) & v(1,4) & v(2,5) & v(3,6) \\ v(1,0) & v(2,1) & v(3,2) & v(4,3) & v(0,4) & v(1,5) & v(2,6) & v(3,0) & v(4,1) & v(0,2) & v(1,3) & v(2,4) & v(3,5) & v(4,6) \\ v(2,0) & v(3,1) & v(4,2) & v(0,3) & v(1,4) & v(2,5) & v(3,6) & v(4,0) & v(0,1) & v(1,2) & v(2,3) & v(3,4) & v(4,5) & v(0,6) \\ v(3,0) & v(4,1) & v(0,2) & v(1,3) & v(2,4) & v(3,5) & v(4,6) & v(0,0) & v(1,1) & v(2,2) & v(3,3) & v(4,4) & v(0,5) & v(1,6) \\ v(4,0) & v(0,1) & v(1,2) & v(2,3) & v(3,4) & v(4,5) & v(0,6) & v(1,0) & v(2,1) & v(3,2) & v(4,3) & v(0,4) & v(1,5) & v(2,6) \end{pmatrix}$$

met v(k,l) het absolute verschil tussen de getallen met index k en l in de originele kenmerkvectoren

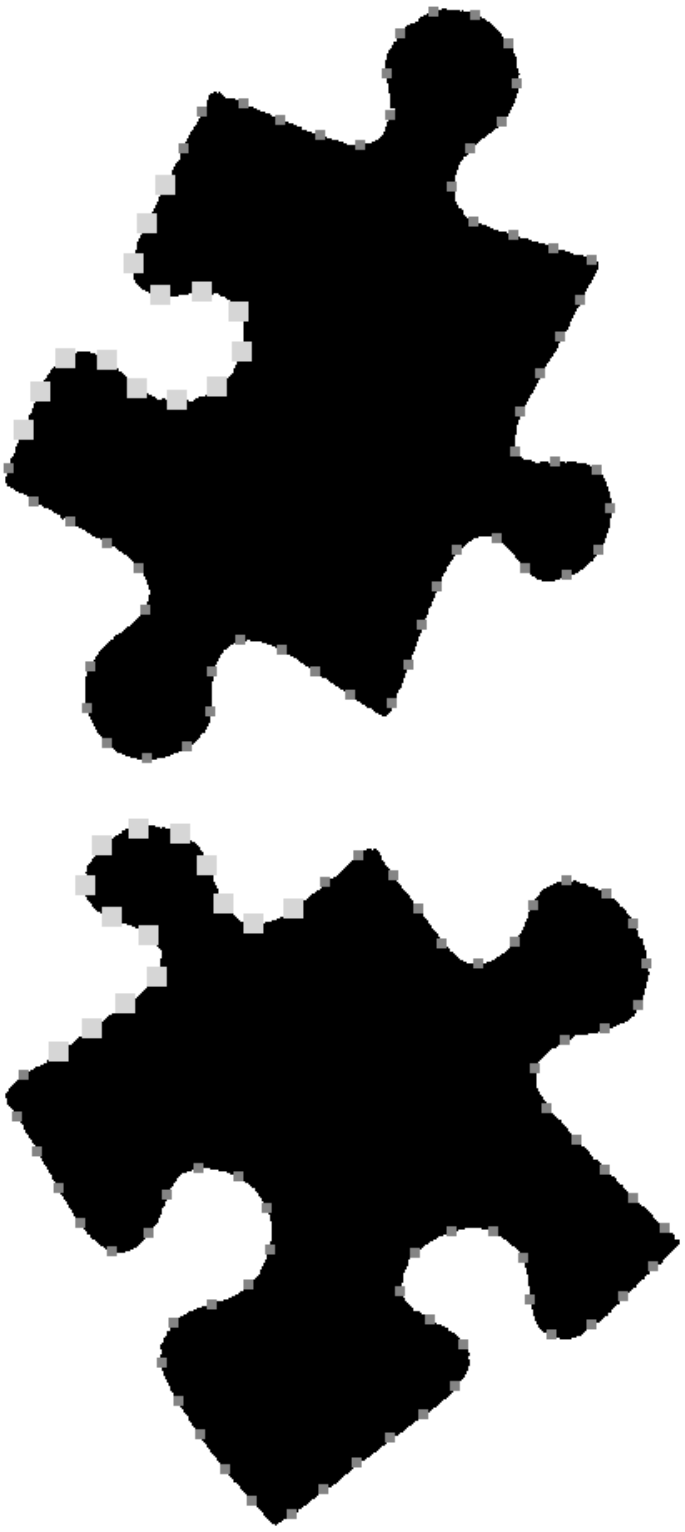


Figuur 4.3: De grafische voorstelling van de berekeningen voor de index matrix

Voor de uiteindelijke kostberekeningen wordt nu voor elke (i, j) een reeks van optellingen gedaan bij variërende l. Hierbij varieert alleen de kolom index met de lengte l: kost+=index[i][j+l]. De elementen van een rij i worden dus één voor één opgeteld startend vanaf kolom j. De index j komt hierbij overeen met de echte startpositie van de match in de kenmerkvectoren van $fig2$, maar de startpositie van de match in de kenmerkvectoren van $fig1$ wordt gegeven door (i+j) modulo fig1.lengte.

Een laatste gebruikte methode om de matching te versnellen is het afbreken van de kostoptellingen als de laagste kost tot dan toe al overschreden is, deze kost kan dan zeker niet meer de uiteindelijk laagste kost worden. De kost waarmee vergeleken wordt is best[maxl-1] zodat men de lange matchlengte zeker niet onterecht overslaat. Deze laagste kost wordt ook bijgehouden over de oproepen van het algoritme heen, zodat men bij meerdere oproepen van het algoritme (om meer detail te verkrijgen) kan profiteren van een nog scherpere kostdrempel.

Een voorbeeld van de best passende delen van twee puzzelstukken gevonden door het algoritme vindt men in figuur 4.4.



Figuur 4.4: Het best passende deel van twee puzzelstukken

Hoofdstuk 5

Globale topologische oplossing voor legpuzzels

Nu we met het benaderende stringmatching algoritme een methode hebben om het beste passende deel van twee puzzelstukken te zoeken en dit uit te drukken in een kost, kunnen we deze kosten op verschillende manieren gebruiken om tot een topologische oplossing te komen van een legpuzzel. De eerste stap die hierbij genomen wordt is het scheiden van de puzzelstukken in verschillende klassen naargelang het hebben van een rechte rand of niet, dit wordt besproken in de eerste paragraaf. Daarna moeten we eerst een topologische oplossing zoeken binnen de randstukken en dan met behulp van deze randtopologie een topologie zoeken voor de interne stukken. Deze laatste twee punten kunnen we op twee manieren benaderen naargelang het beschouwen van een puzzelstuk als één gesloten contour of als vier deelcontouren. Uiteindelijk worden nog de resultaten weergegeven voor verschillende geteste puzzels.

5.1 Indeling van de puzzelstukken in klassen

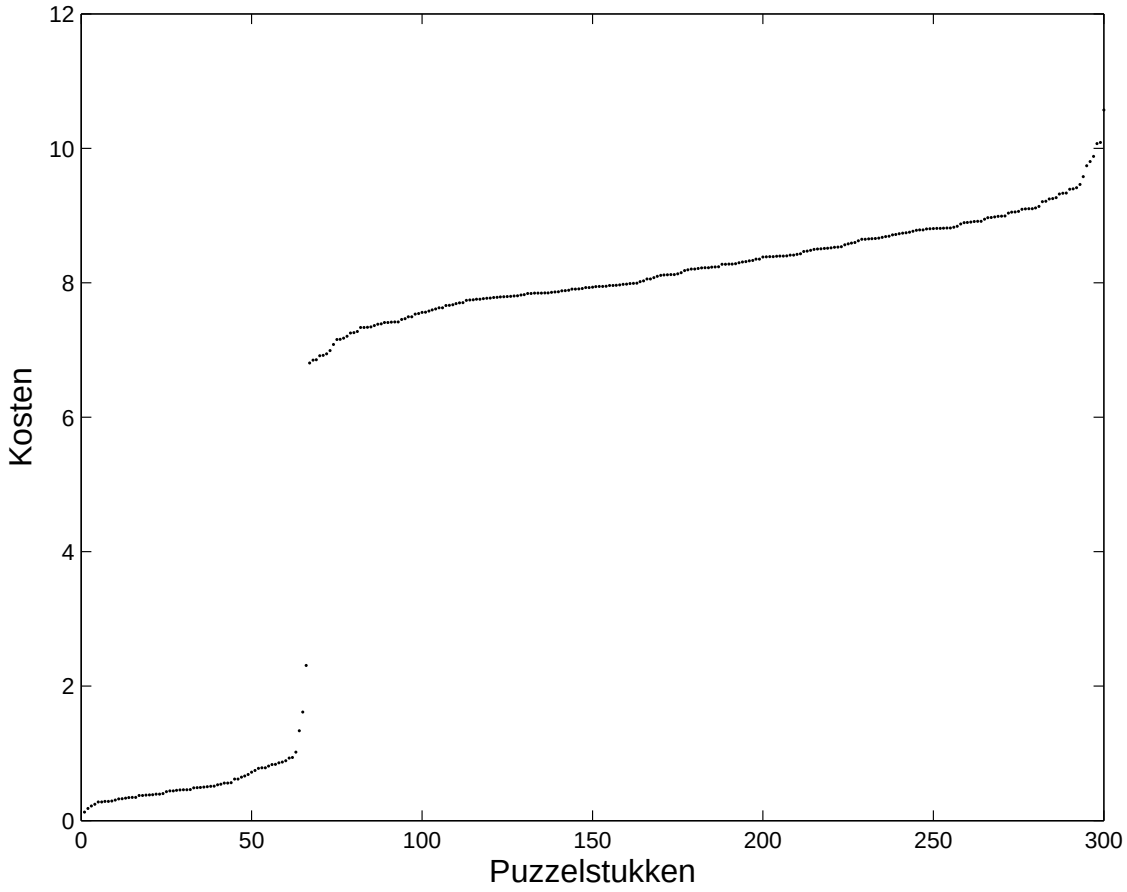
De indeling in klassen gebeurt volgens het aantal rechte randen van het puzzelstuk:

- **hoek-randstuk**: 2 rechte randen, de vier hoeken van de legpuzzel.
- **één-randstuk**: 1 rechte rand, vormen samen met de hoek-randstukken het kader van de legpuzzel.
- **intern stuk**: geen rechte rand, de puzzelstukken binnen het kader.

Voor de indeling zelf in deze klassen hergebruiken we het matching concept. We doen eerst een matching van alle puzzelstukken met een artificiële rechte hoek. Dit wordt gedaan over een vaste matchlengte: 2 maal de gemiddelde rechte rand lengte. Deze rechte rand lengte kunnen we schatten uit de gemiddelde lengte van de kettingcodes. Een voorbeeld van deze rechte hoek ziet men in figuur 5.1. De puzzelstukken die bij deze matching de vier laagste kosten geven lijken het beste op een rechte hoek. Ze zijn dan dus de vier hoek-randstukken. Na deze matching met een artificiële rechte hoek doen we hetzelfde met een artificiële rechte rand over een vaste matchlengte van 1 maal de gemiddelde rechte rand lengte. Een voorbeeld van deze rechte rand ziet men in figuur 5.1. De puzzelstukken die bij deze matching de laagste kosten hebben, hebben een rand die goed overeenkomt

Figuur 5.1: Artificiële rechte hoek en rand gebruikt bij de klassenindeling

met een rechte rand. Als we deze kosten sorteren dan blijkt volgens figuur 5.2 dat er een duidelijke scheiding bestaat tussen randstukken en interne stukken. Met deze duidelijke

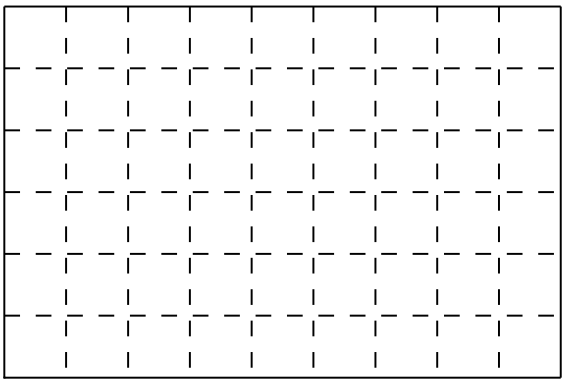


Figuur 5.2: Gesorteerde kosten na matching van alle puzzelstukken met een rechte rand

scheiding kunnen we de puzzelstukken betrouwbaar indelen in de verschillende klassen.

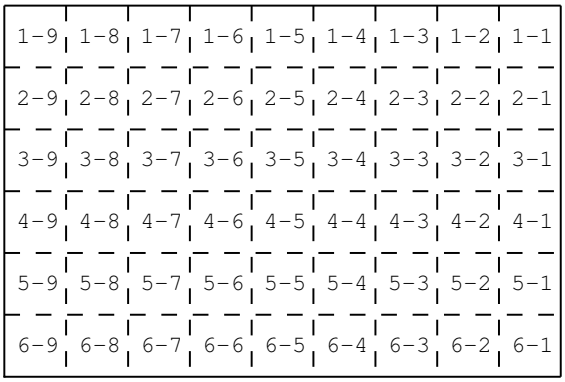
5.2 Algemene oplossingsmethode

Voor we kunnen spreken over een oplossing moeten we definiëren wat we verstaan onder een oplossing. We beschouwen de legpuzzel als opgelost als we voor alle puzzelstukken de juist positie kunnen bepalen in een rechthoekig oplossingsraster. De afmetingen van dit oplossingsraster worden bepaald tijdens het zoeken van de randtopologie. In het vervolg van deze uitleg stellen we dit oplossingsraster voor zoals op figuur 5.3. Bij het inlezen van



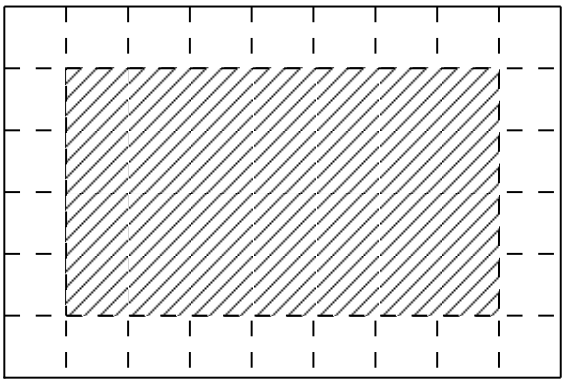
Figuur 5.3: Oplossingsraster gebruikt bij het zoeken van de oplossing van een legpuzzel

de puzzelstukken wordt de rij en kolom positie van het puzzelstuk in de correcte oplossing van de legpuzzel opgeslagen samen met de andere gegevens van het respectievelijke puzzelstuk. Als we na alle oplossingsalgoritmes voor elke puzzelstuk een unieke positie gevonden hebben binnen het oplossingsraster dan krijgen we voor elke positie in het oplossingsraster een rij en kolom waarde: dit noemen we een topologische oplossing. Als men nu voor elke rij en kolom van het oplossingsraster een monotone stijging of daling heeft van de rij of kolom waarden in stappen van 1 dan beschouwen we de oplossing als correct. Een voorbeeld van een correcte topologie is weergegeven in figuur 5.4.



Figuur 5.4: Voorbeeld van een correcte topologie binnen het oplossingsraster

Als we dit zoeken naar een topologische oplossing binnen het oplossingsraster willen vereenvoudigen is het best dat we het probleem opsplitsen in deelproblemen. Dit doen we door eerst een topologische oplossing binnen de randstukken te zoeken: de randtopologie. De mogelijke posities voor de randstukken binnen het oplossingsraster zijn weergegeven in figuur 5.5. De verzameling van al deze mogelijke posities noemen we het randkader. Dit



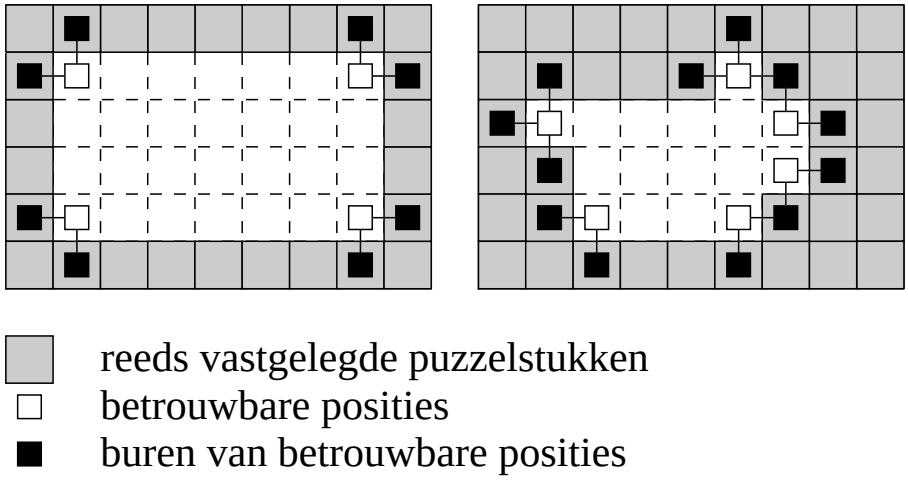
Figuur 5.5: Oplossingsraster voor de randstukken: randkader

opsplitsen in rand gedeelte en intern gedeelte is mogelijk omdat we eerder een perfecte indeling van de puzzelstukken hebben gemaakt in randstukken en interne stukken. Naast de vermindering van de logische complexiteit is er ook een vermindering in berekeningscomplexiteit omdat we totaal geen rekening moeten houden met de interne stukken. Men ziet op figuur 5.5 dat elke positie binnen het randkader telkens 2 buurposities heeft die ook binnen het randkader gelegen zijn (volgens het 4 buren principe weergegeven in figuur 2.1). We kunnen nu twee vereisten definiëren:

- De uiteindelijke randtopologie is een ring van alle randstukken.
- Randstukken die buur zijn van elkaar in de uiteindelijke randtopologie hebben een gemeenschappelijk contourdeel en moeten dus een zo laag mogelijke matchingskost met elkaar hebben.

De in de volgende paragrafen besproken algoritmen voor het zoeken van de randtopologie baseren zich op deze vereisten en hebben als input alle randstukken en geven als output een cyclisch geordende lijst van randstukken: een ring van randstukken. Door te kijken waar de hoek-randstukken zich in deze ring bevinden kunnen we de afmetingen van het volledige oplossingsraster bepalen. Op dit moment wordt dus pas het oplossingsraster opgebouwd. Nu draait men de ring binnen het oplossingsraster voor randstukken zodat de vier hoek-randstukken op de vier hoekpunten van het raster vallen. Hiermee is de randtopologie volledig vastgelegd.

Nu gaan we de reeds vastgelegde randtopologie gebruiken als beginpunt voor het zoeken van de interne topologie. Voor elke iteratie van het oplosalgoritme voor de interne stukken worden de "betrouwbare posities" bepaald. Met een betrouwbare positie bedoelt men een positie in het raster met twee of meer reeds vastgelegde buurpuzzelstukken (volgens het 4 buren principe weergegeven in figuur 2.1). Een paar voorbeelden zijn weergegeven in figuur 5.6. De benaming betrouwbaar is gebruikt omdat men alleen voor deze posities



Figuur 5.6: Voorbeelden van betrouwbare posities

genoeg buren heeft waarmee men de matchingskost kan berekenen. Het linkse oplossingsraster geeft de beginsituatie weer, het rechtse oplossingsraster de situatie na reeds een paar puzzelstukken vastgelegd te hebben. De in de volgende paragrafen besproken algoritmen voor het zoeken van een interne topologie hebben als input deze betrouwbare posities samen met de gegevens van hun buurpuzzelstukken. Als output geven ze één van de betrouwbare posities samen met één van de nog vast te leggen puzzelstukken. Dit puzzelstuk wordt dan vastgelegd op die positie in het oplossingsraster, per iteratie wordt dus één puzzelstuk vastgelegd. Uiteindelijk is het oplossingsraster volledig ingevuld met alle puzzelstukken en is er dus een volledige topologische oplossing gevonden.

5.3 Puzzelstuk beschouwen als één gesloten contour

In de algoritmen die in deze paragraaf beschreven worden zal de contour van elk puzzelstuk beschouwd worden als één gesloten contour. Hiermee bedoelen we dat deze contour in zijn geheel gebruikt wordt in het matching algoritme. Er is dus maar één kost per paar puzzelstukken. Het gegeven dat men de contour van elk puzzelstuk in vier gescheiden deelcontouren kan opsplitsen wordt hier dus niet gebruikt.

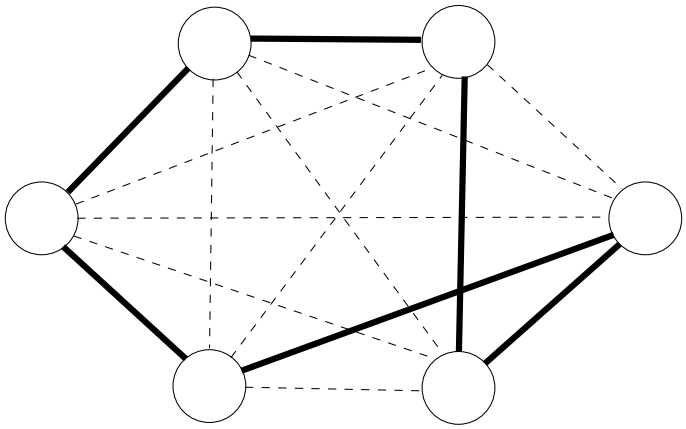
5.3.1 Oplosalgoritmes voor de randtopologie

De beginstap bij alle algoritmen is het matchen van alle randstukken met elkaar. Dit geeft per paar randstukken één kost. Als we dit conceptueel voorstellen krijgen we een volledig geconnecteerde graaf met de volgende inhoud:

- **knopen**: de conceptuele voorstelling van de randstukken
- **takken**: de conceptuele voorstelling van de match van twee randstukken, ze hebben dus als kost de matchingskost

Nu moet met in deze graaf een cyclisch pad zoeken dat elke knoop één maal bezoekt (vertaling van de ring vereiste). Verder in de bespreking noemen we zo een pad een

complete ronde. De kosten van de verschillende takken van de complete ronde moeten zo laag mogelijk zijn (vertaling van de vereiste van een zo laag mogelijke matchingskost tussen buren). Zo een graaf samen met een mogelijke complete ronde is weergegeven in figuur 5.7. De complete ronde geeft dan uiteindelijk een cyclische geordende lijst van



Figuur 5.7: Graaf met een complete ronde

randstukken.

Het hebben van een zo laag mogelijke kost van de verschillende takken van een complete ronde kan men op verschillende manieren interpreteren. Zo bekomt men dan de volgende algoritmen voor het zoeken van een complete ronde:

STSP algoritme

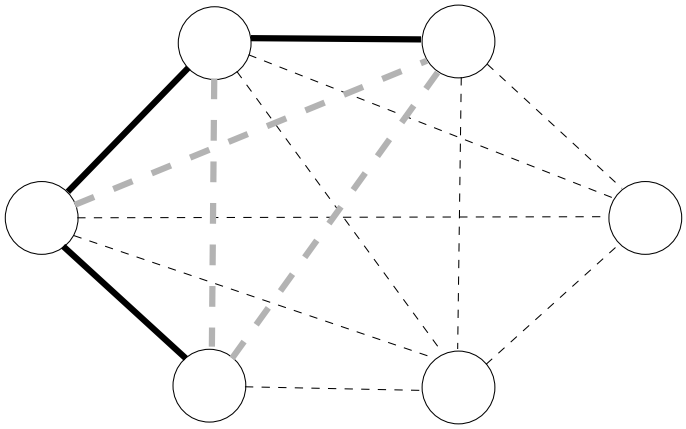
Hier zoekt men een complete ronde waarbij de som van de kosten van alle takken van de complete ronde zo laag mogelijk is. De complete ronde wordt dus in zijn geheel geëvalueerd. Hier bestaat een standaard algoritme voor dat men het Symmetrisch Travelling Salesman Problem algoritme noemt. In het programma werd een "branch en bound" implementatie gebruikt die altijd de beste complete ronde geeft.

Best first algoritme

Hier zijn we vertrokken van een lijst van alle takken gesorteerd op kost (met laagste kost eerst in de lijst). We lopen deze lijst dan af en selecteren de takken één voor één voor de complete ronde. Het idee hierachter is dat de takken met een zeer lage kost zeker deel uitmaken van de complete ronde. Terwijl we deze takken selecteren doen we een lusdetectie zodat we geen lussen creëren voor de uiteindelijke complete ronde gevonden is. Takken zoals de grijs gestreepte takken in figuur 5.8 worden dus overgeslagen bij de selectie. Na het aflopen van genoeg takken bekomt men de uiteindelijke complete ronde.

Best confidence first algoritme

Dit is een verbeterde versie van het best first algoritme. In elke cyclus van de takselectie worden eerst alle nog *vrije* knopen k gezocht. *Vrije* knopen zijn knopen waarvan er minder dan twee takken geselecteerd zijn voor de complete ronde. Voor elke knoop k wordt dan



Figuur 5.8: De lusdetectie

een lijst van takken van die knoop opgebouwd, stel dat we deze lijst L_k noemen. Uit die lijsten L_k worden dan de takken verwijderd die al deel uit maken van de complete ronde of die als tweede knoop een knoop hebben die niet meer *vrij* is. Vervolgens wordt voor elke L_k de twee takken met laagste kost gezocht en wordt het verschil berekend tussen deze twee kosten. De lijst L_k waarvan dit verschil het grootst is wordt dan $(L_k)_{best}$ genoemd. De tak met de laagste kost uit de lijst $(L_K)_{best}$ wordt dan geselecteerd voor de complete ronde. Deze cyclus wordt herhaald tot de uiteindelijke complete ronde gevonden is. Ook hier wordt er in elke cyclus een lusdetectie gedaan.

In de praktijk blijkt het best confidence first algoritme de beste keuze te zijn voor het zoeken van de complete ronde die de juiste randtopologie geeft.

5.3.2 Oplosalgoritmes voor de interne topologie

Zoals eerder vermeld moet hier één van de nog vast te leggen puzzelstukken gekozen worden samen met één betrouwbare positie. Bij alle algoritmen wordt er voor elke betrouwbare positie i een lijst L_i van kosten opgesteld. Deze lijst L_i wordt bekomen door voor elk nog vast te leggen puzzelstuk k de kost te berekenen van het inpassen in de betrouwbare positie i. Deze kost is de gemiddelde matchingskost van k met alle buren van de betrouwbare positie i. Nu kan men verschillende algoritmen definiëren die deze lijsten L_i gebruiken om één nog vast te leggen puzzelstuk en één betrouwbare positie te kiezen:

Best first algoritme

Hier wordt globaal over alle lijsten L_i de laagste kost gezocht. Stel dat deze kost zich bevindt in lijst L_k, dan is k de uiteindelijk gekozen betrouwbare positie en het uiteindelijk gekozen puzzelstuk is het puzzelstuk dat de laagste kost opgeleverd heeft.

Best confidence first algoritme

Hier wordt voor elke lijst L_i het verschil berekend tussen de twee beste kosten. Stel dat bij lijst L_k dit verschil het grootst is, dan is k de uiteindelijk gekozen betrouwbare positie en

het uiteindelijk gekozen puzzelstuk is het puzzelstuk dat de laagste kost opgeleverd heeft in lijst L_k.

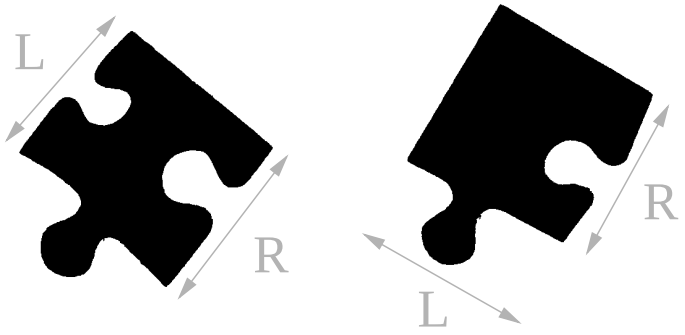
In de praktijk blijkt het best confidence first algoritme de beste keuze te zijn voor het zoeken van de juiste interne topologie.

5.4 Puzzelstuk opsplitsen in vier deelcontouren

In de algoritmen die in deze paragraaf beschreven worden zal de contour van elk puzzelstuk opgesplitst worden in vier deelcontouren. Dit kan men doen door de contour te splitsen op de vier hoekpunten die we bij de kenmerkenextractie hebben bekomen. Per paar puzzelstukken krijgen we nu 16 kosten: de kosten van het matchen van elke deelcontour van het ene puzzelstuk met elke deelcontour van het andere puzzelstuk.

5.4.1 Oplosalgoritmes voor de randtopologie

Ook hier is de beginstap bij alle algoritmen het matchen van alle randstukken met elkaar. Maar nu kunnen we gebruik maken van de extra informatie die het opsplitsen in deelcontouren biedt. We kunnen ons voor elk randstuk beperken tot de deelcontouren die gemeenschappelijk kunnen zijn met andere randstukken. Deze zijn voor hoek-randstukken de twee deelcontouren die geen rechte randen zijn en voor de één-randstukken de twee deelcontouren die grenzen aan de rechte rand. Deze randen zijn weergegeven op figuur 5.9. De deelcontour links van van de rechte rand heeft het label L gekregen en de deelcontour

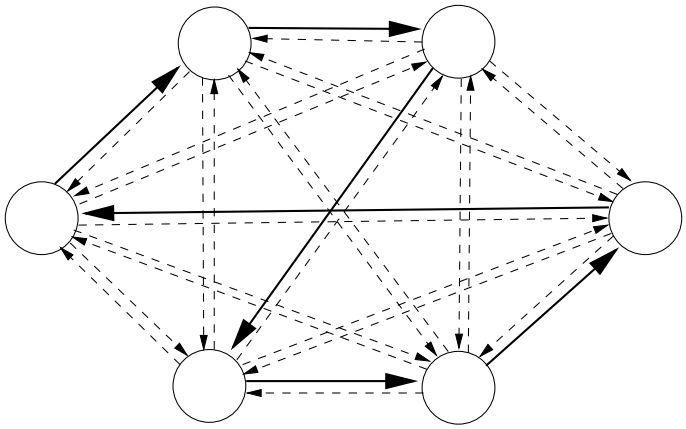


Figuur 5.9: De mogelijk gemeenschappelijke deelcontouren van de randstukken

rechts van de rand heeft het label R gekregen. De rechte rand voor twee randstukken die buren zijn moet voor beide randstukken aan dezelfde kant liggen. Als men hiermee rekening houdt, dan zijn er maar twee mogelijke combinaties voor randstukken die buren zijn namelijk: deelcontour L van het ene randstuk past in deelcontour R van het andere randstuk of deelcontour R van het ene randstuk past in deelcontour L van het andere randstuk. Bij het matchen van twee randstukken beperken we ons dus tot het matchen van deelcontour L van het ene randstuk met deelcontour R van het andere randstuk en het matchen deelcontour R van het ene randstuk met deelcontour L van het andere randstuk. Dit geeft dus twee kosten per paar randstukken.

Ook hier kan men het zoeken van de randtopologie voorstellen door het zoeken van een complete ronde in een volledig geconnecteerde graaf. De kost van de verschillende takken van de complete ronde moet ook hier zo laag mogelijk zijn. Maar hier is het een graaf

met gerichte takken en de complete ronde moet de richting van deze takken respecteren. We krijgen tussen elk paar knopen twee tegengesteld gerichte takken: de kost van de ene tak correspondeert met de R met L matching en de kost van de andere met de L met R matching. Zo een gerichte graaf samen met een mogelijke complete ronde is weergegeven in figuur 5.10.



Figuur 5.10: Gerichte graaf met een complete ronde

Het hebben van een zo laag mogelijke kost van de verschillende takken van een complete ronde kan men ook hier op verschillende manieren interpreteren. We krijgen dezelfde algoritmen maar dan specifiek aangepast voor de gerichte graaf:

ATSP algoritme

Hier zoekt men een complete ronde waarbij de som van de kosten van alle gerichte takken van de complete ronde zo laag mogelijk is. De complete ronde wordt dus in zijn geheel geëvalueerd. Hier bestaat een standaard algoritme voor dat men het Assymetrisch Travelling Salesman Problem algoritme noemt. In het programma werd een "branch en bound" implementatie gebruikt die altijd de beste complete ronde geeft.

Best first algoritme

Het eerder beschreven best first algoritme voor niet gerichte grafen is hier licht aangepast voor de gerichte grafen. De controle op het volgen van de richting van de takken is er aan toegevoegd.

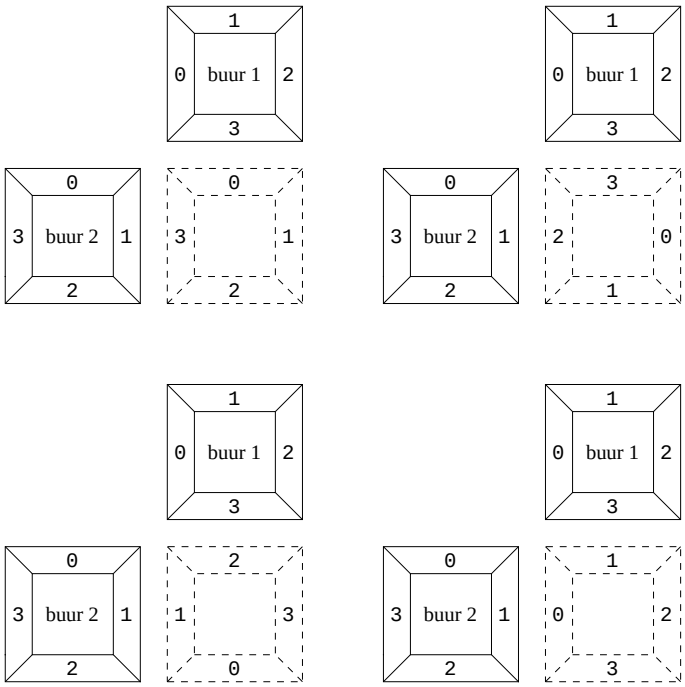
Best confidence first algoritme

Het eerder beschreven best confidence first algoritme voor niet gerichte grafen is ook hier licht aangepast voor de gerichte grafen. De controle op het volgen van de richting van de takken is er aan toegevoegd.

In de praktijk blijkt het best confidence first algoritme de beste keuze te zijn voor het zoeken van de complete ronde die de juiste randtopologie geeft.

5.4.2 Oplosalgoritmes voor de interne topologie

Ook hier moet één van de nog vast te leggen puzzelstukken gekozen worden samen met één betrouwbare positie. Maar nu maken we gebruik van de extra informatie die het splitsen in deelcontouren biedt. Het algoritme loopt volledig analoog met het algoritme voor niet opgesplitste contouren, alleen de berekening van de inpassingskost in een betrouwbare positie is veranderd. Dit kunnen we best illustreren met figuur 5.11. Hier zijn de



Figuur 5.11: Berekening van de matchingskost met de buren bij splitsing in 4 deelcontouren

puzzelstukken symbolisch voorgesteld door een vierkant en de vier deelcontouren van het puzzelstuk zijn aangeduid en genummerd. Door het splitsen in deelcontouren kan men voor alle reeds vastgelegde puzzelstukken ook de oriëntatie bijhouden. Dit doen we door het nummer van de deelcontour die naar boven gericht is bij te houden. Buur 1 en buur 2 zijn de twee reeds vastgelegde buurpuzzelstukken van de betrouwbare positie. Als men nu voor een willekeurig puzzelstuk de inpassingskost wilt berekenen, dan doet men het volgende. We kiezen een oriëntatie voor het puzzelstuk en matchen de koppels deelcontouren die dan gemeenschappelijk moeten zijn. Deze koppels deelcontouren zijn voor de eerste oriëntatie in figuur 5.11 het koppel (deelcontour 3 van buur 1,deelcontour 0 van het puzzelstuk) en het koppel (deelcontour 1 van buur 2,deelcontour 3 van het puzzelstuk). Dit matchen geeft dus per koppel één kost. De uiteindelijke kost voor de gekozen oriëntatie is dan het gemiddelde van de kosten van alle koppels. Dit herhalen we voor de drie andere oriëntaties, wat dan uiteindelijk vier kosten opleverd. De laagste kost van deze vier kosten wordt dan beschouwd als de uiteindelijke inpassingskost.

Het gebruik van de eerder besproken lijsten L_i is hetzelfde gebleven. We hebben dus weer een best first algoritme en een best confidence first algoritme. In de praktijk blijkt het best confidence first algoritme de beste keuze te zijn voor het zoeken van de juiste

interne topologie.

5.5 Resultaten voor verschillende legpuzzels

Alle stappen die nodig zijn om van de individuele puzzelstukken te komen tot de globale topologische oplossing hebben we uitgevoerd voor vijf verschillende puzzels. De totale uitvoeringstijd nodig voor het bekomen van de correcte oplossing is dan opgemeten voor de versie zonder splitsing van de contour en voor de versie met splitsing van de contour. Deze resultaten zijn weergegeven in tabel 5.1. De tijden zijn opgemeten met een AMD

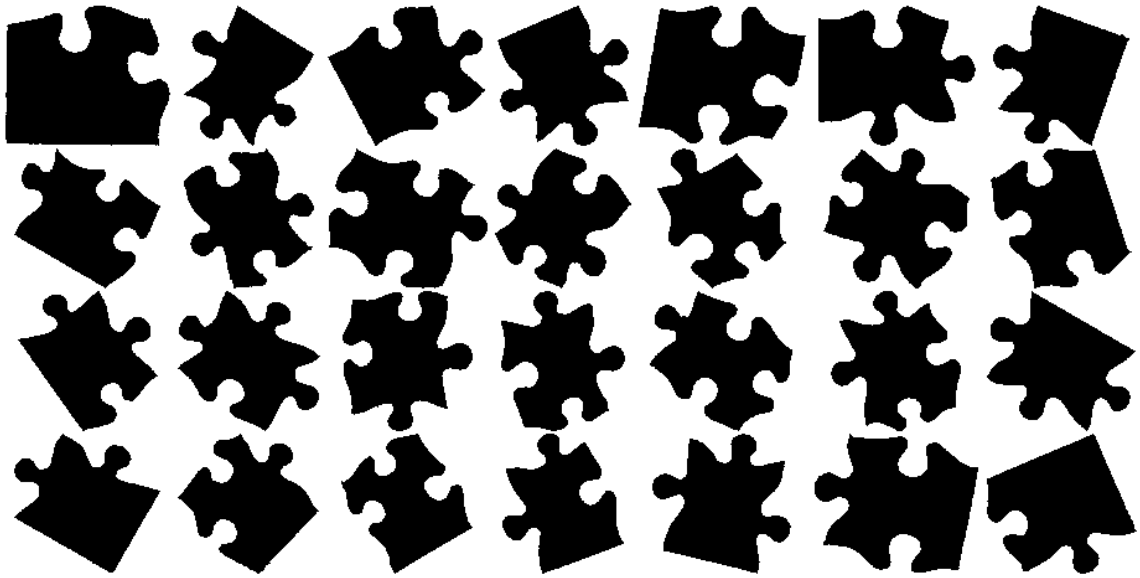
	Niet Gesplitst	Gesplitst
Legpuzzel 1: figuur 5.12	4	8
Legpuzzel 2: figuur 5.13	33	12
Legpuzzel 3: figuur 5.14	153	38
Legpuzzel 4: figuur 5.15	81	76
Legpuzzel 5: figuur 5.16	-	244

Tabel 5.1: De uitvoeringstijden voor de verschillende geteste legpuzzels (in seconden)

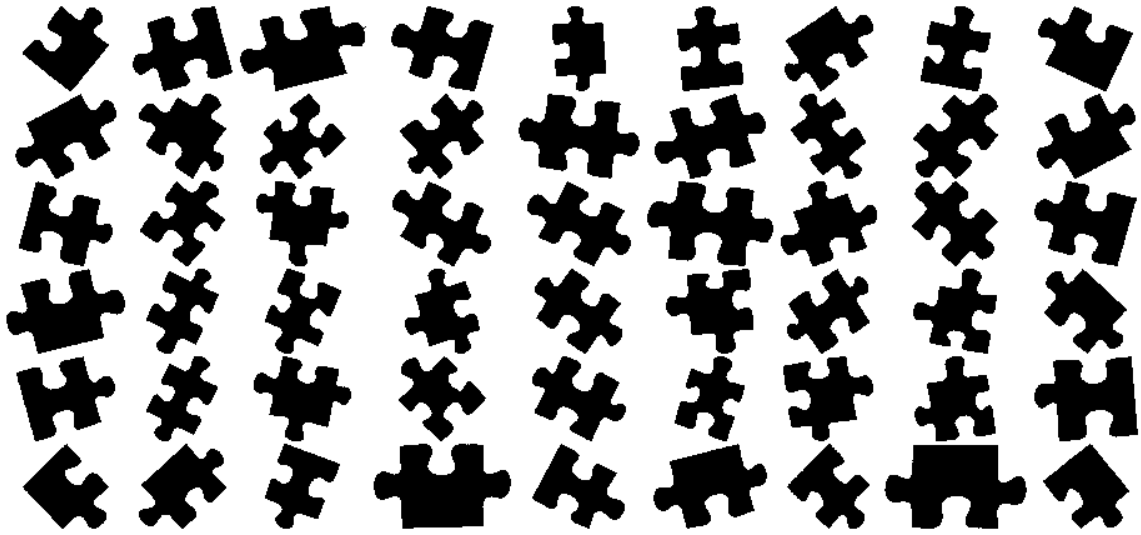
Athlon XP 1700+ (1466 Mhz).

Bij legpuzzel 5 was het bij de niet gesplitste versie onmogelijk om de correcte globale topologische oplossing te bekomen. Bij de meeste puzzels is de gesplitste versie beduidend sneller dan de niet gesplitste versie. Alleen legpuzzel 1 is hierop een uitzondering omdat de tijdwinst in het oplossingsproces niet opweegt tegen de extra tijd nodig om de hoekpunten te zoeken voor de gesplitste versie. Dit is te verklaren door de zeer geringe complexiteit van het oplossingsproces bij legpuzzels met een klein aantal puzzelstukken zoals legpuzzel 1.

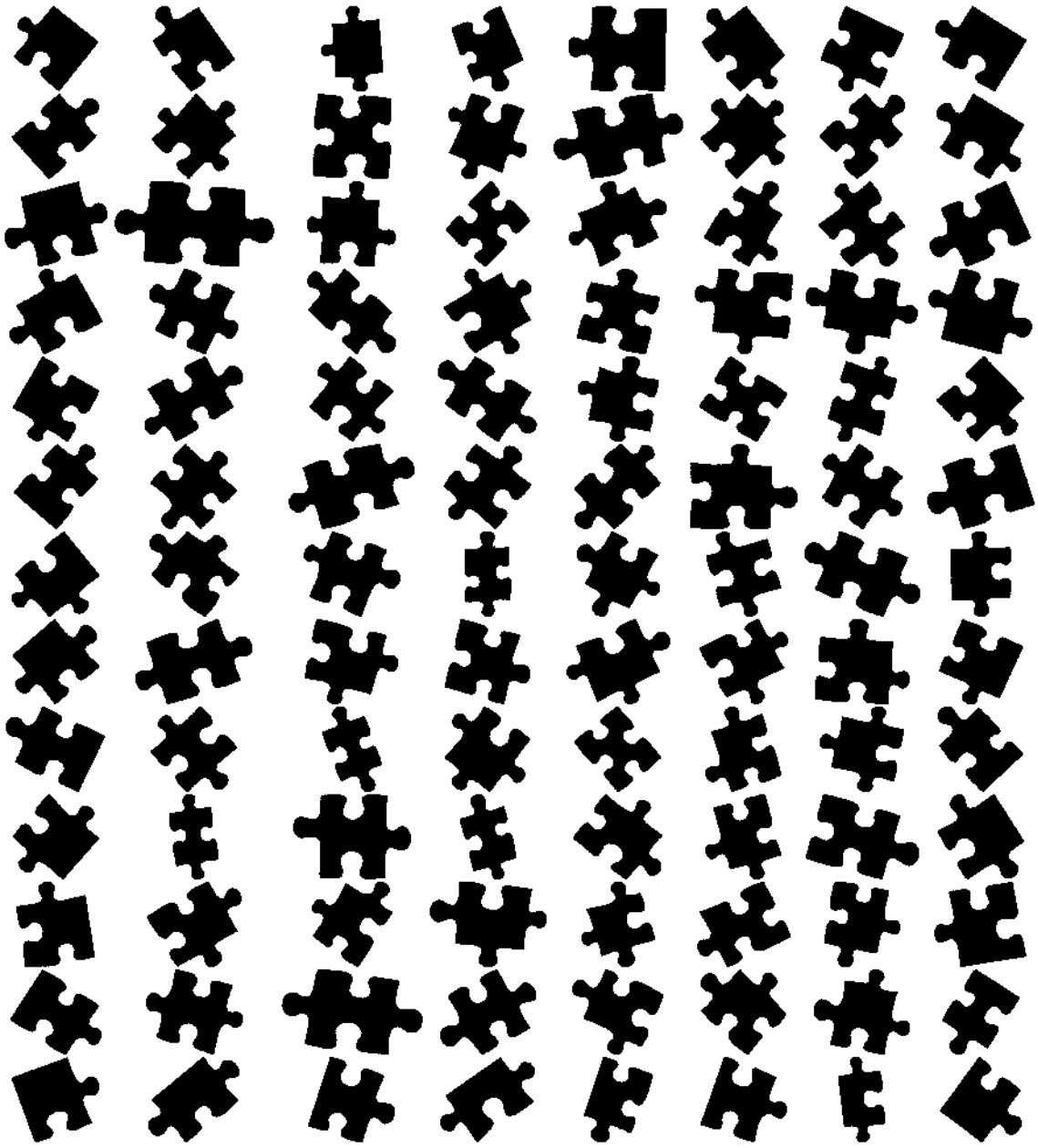
Legpuzzel 5 is op dit moment de grootste puzzel die automatisch is opgelost met behulp van een computer. De legpuzzel bevat 300 puzzelstukken en heeft als dimensie 15x20. De vorige beste prestatie was het oplossen van een legpuzzel van 204 puzzelstukken op 19 juli 2002. Hier gebruikte men een matching methode die alleen voor puzzelstukken kon gebruikt worden, de methode is beschreven in [2].



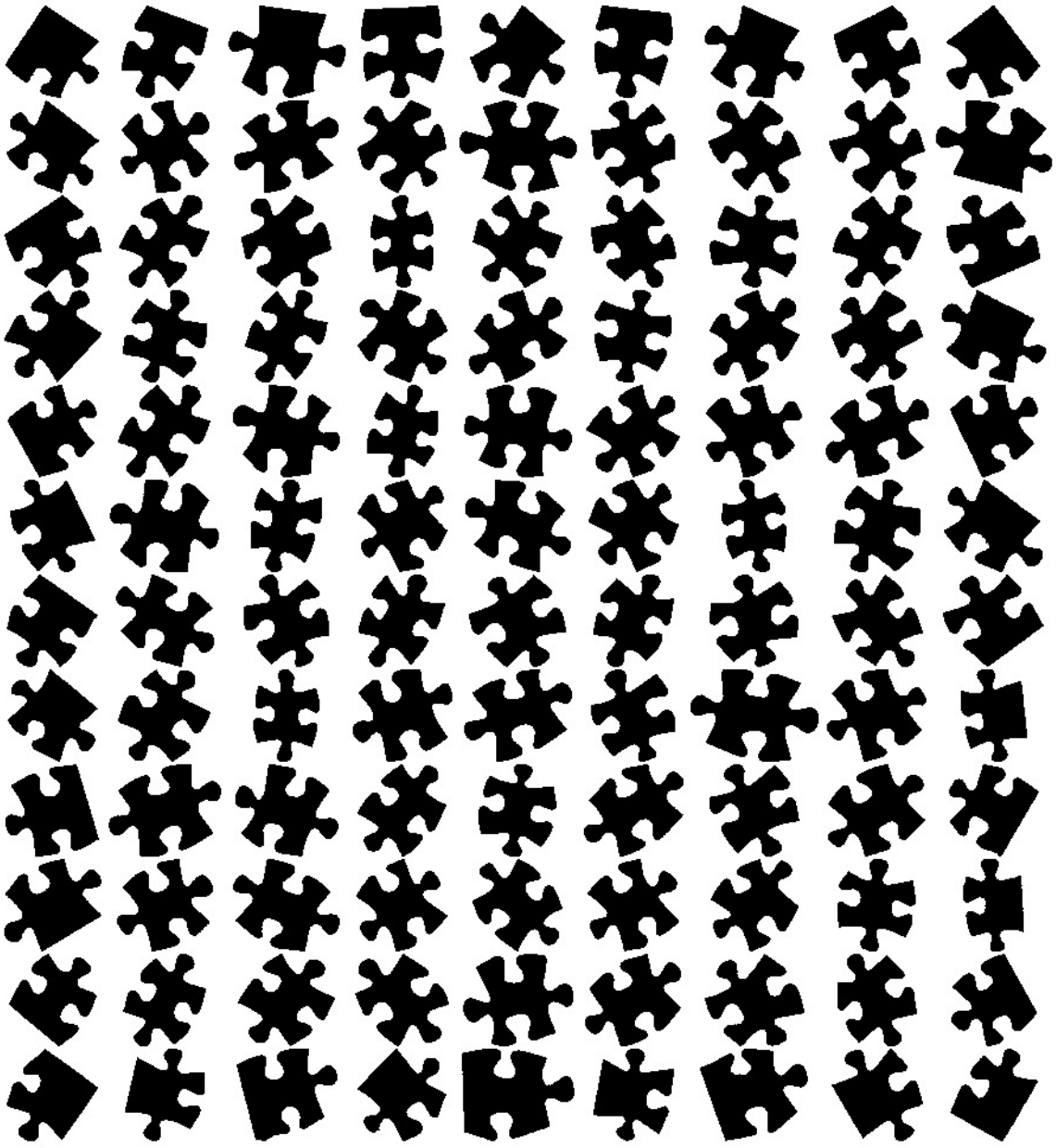
Figuur 5.12: Legpuzzel 1: 28 puzzelstukken, dimensie 4x7



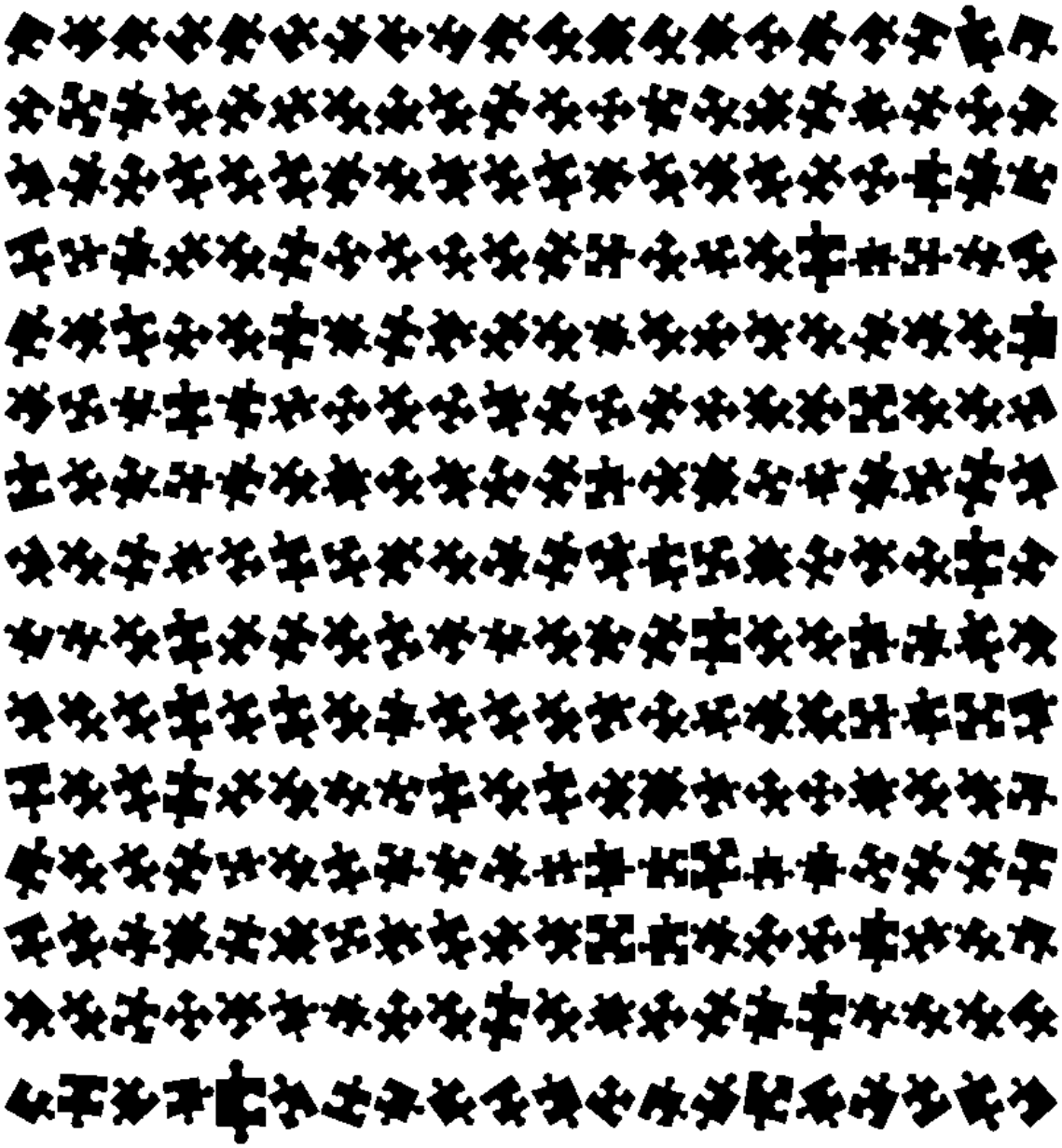
Figuur 5.13: Legpuzzel 2: 54 puzzelstukken, dimensie 6x9



Figuur 5.14: Legpuzzel 3: 104 puzzelstukken, dimensie 13x8



Figuur 5.15: Legpuzzel 4: 108 puzzelstukken, dimensie 12x9



Figuur 5.16: Legpuzzel 5: 300 puzzelstukken, dimensie 15x20

Hoofdstuk 6

Toepassing op contouren met irreguliere vorm: papierfragmenten

Het essentiële verschil tussen puzzelstukken en papierfragmenten is het verschil in de structuur van de contour: een puzzelstuk heeft een continue contour en een papierfragment heeft normaal een niet continue contour of anders gezegd een contour met irreguliere vorm. Om aan te tonen dat het matching algoritme dat gebruikt is om de legpuzzels topologisch op te lossen ook kan gebruikt worden bij het matchen van papierfragmenten, gebruiken we het in dit hoofdstuk als een eerste stap in de reconstructie van een in stukken gescheurd papierfragment.

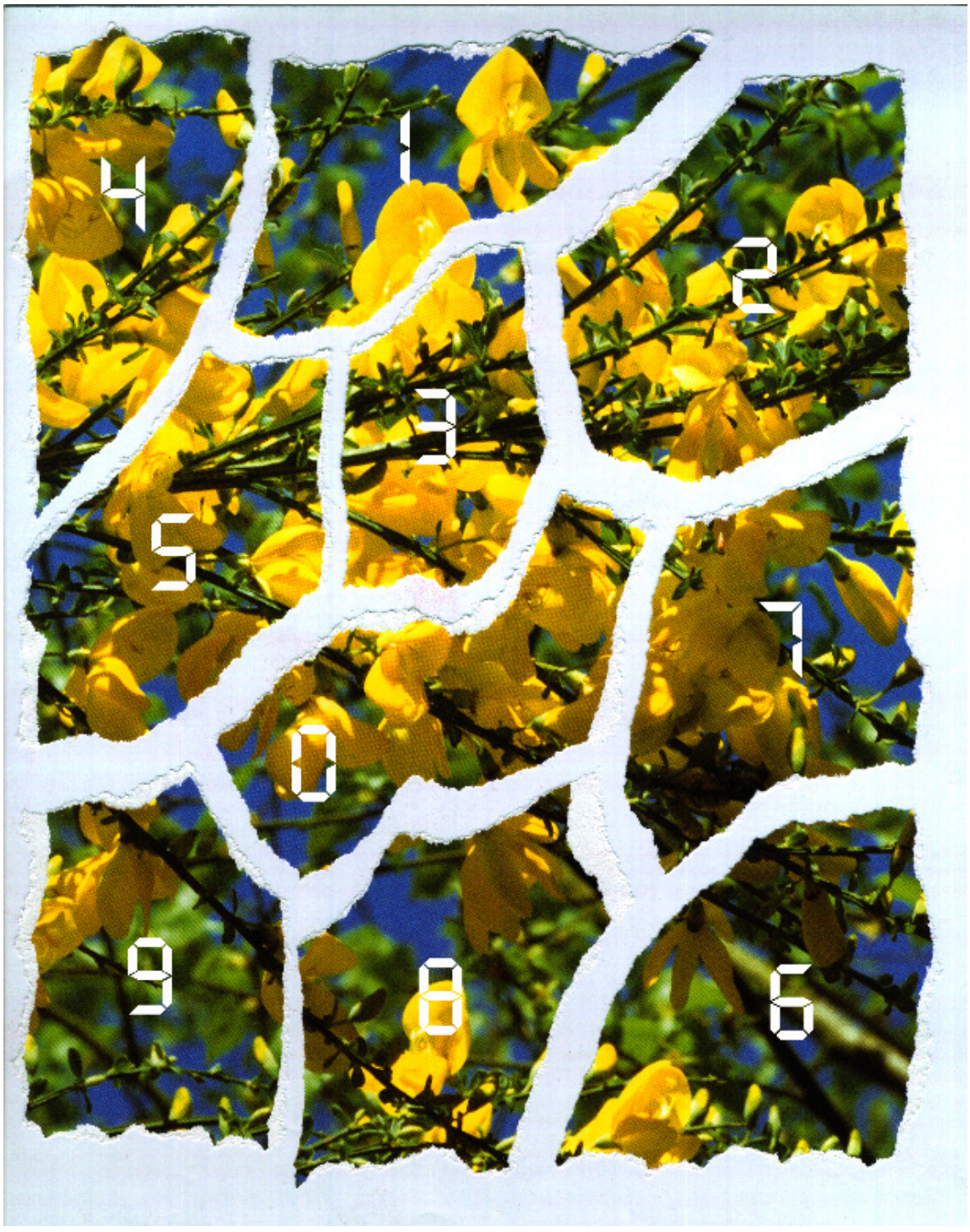
We scheuren een groot papierfragment (figuur 6.1) in tien kleinere papierfragmenten. De posities van de papierfragmenten in het originele grote papierfragment zijn weergegeven in figuur 6.2. De papierfragmenten zijn genummerd van 0 tot 9 zodat we er verder makkelijk naar kunnen verwijzen. Nu matchen we alle paren papierfragmenten over een vaste matchlengte met het beschreven matching algoritme. Nu wordt ook de kleur kenmerkvector gebruikt in het matching proces. Met de matchlengte bedoelen we voor alle duidelijkheid de lengte van de best passende contourdelen. Het is belangrijk dat deze matchlengte groot genoeg is zodat het algoritme over genoeg informatie beschikt om duidelijk onderscheid te maken tussen verschillende mogelijke contourvormen. De resultaten die we bekomen voor alle paren sorteren we nu op de matchingskost, dit geeft dan tabel 6.1. De kolom “paar” geeft aan over welke twee papierfragmenten het gaat, de kolom “kost” geeft hun matchingskost en de kolom “buren” geeft weer of de papierfragmenten al dan niet aan elkaar hingen in het originele grote papierfragment. De papierfragmenten die in het originele papierfragment aan elkaar hingen hebben een duidelijk lagere kost dan diegene die niet aan elkaar hingen. De uitzonderingen hierop zijn vooral te wijten aan een te kort gemeenschappelijk stuk contour: dit was bijvoorbeeld het geval bij het paar 0-2.

We lopen nu de paren van de lijst één voor één af (de laagste kost eerst) en zetten de respectievelijke papierfragmenten aan elkaar op de best passende delen die we verkregen hebben bij de matching van de paren papierfragmenten. Voorbeelden van de best passende delen zijn weergegeven in figuur 6.3. Hier zijn de best passende delen van de eerste twee paren uit de tabel weergegeven in cyaan. Uiteindelijk kan men zo het volledige originele papierfragment reconstrueren.

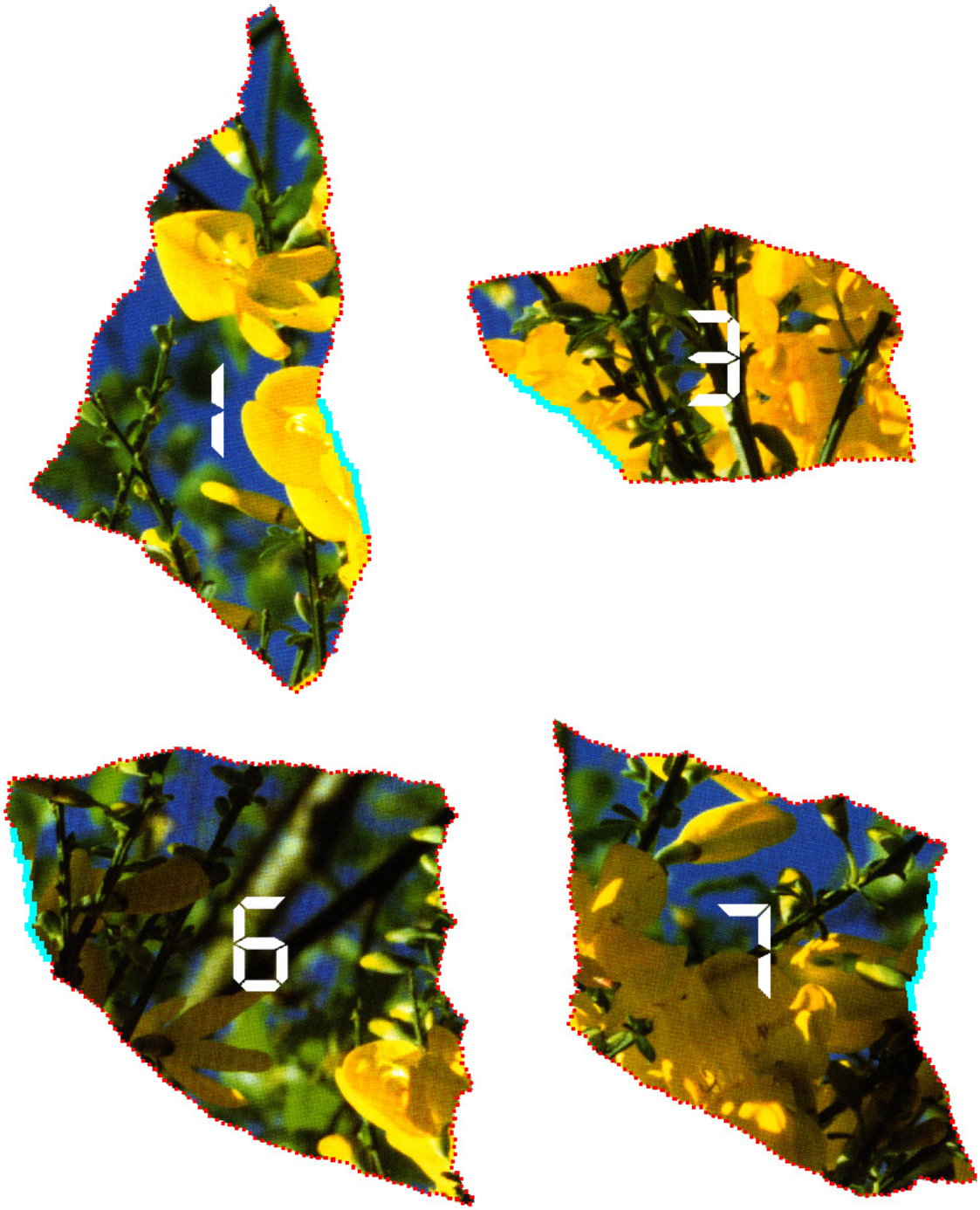
Paar	Kost	Buren
1-3	14,71	ja
6-7	16,26	ja
0-7	16,44	ja
0-8	16,45	ja
0-3	16,93	ja
8-9	17,03	ja
4-5	18,80	ja
7-8	20,97	ja
6-8	21,37	ja
0-9	22,75	ja
1-2	22,82	ja
2-7	23,40	ja
2-3	24,71	ja
5-9	25,29	ja
1-4	26,90	ja
0-5	26,96	ja
0-1	31,66	nee
6-9	32,79	nee
2-6	33,41	nee
3-8	33,51	nee
1-7	34,25	nee
1-8	34,43	nee
2-9	35,23	nee
7-9	35,89	nee
5-8	36,47	nee
5-7	36,58	nee
1-9	36,82	nee
2-8	37,32	nee
3-5	37,42	ja
0-4	39,48	nee
4-8	39,83	nee
4-7	40,04	nee
4-6	40,16	nee
3-7	40,36	nee
4-9	40,53	nee
1-5	40,58	ja
0-2	40,79	ja
2-4	41,49	nee
3-9	41,62	nee
0-6	41,65	nee
2-5	42,15	nee
5-6	42,67	nee
3-4	42,92	nee
1-6	42,95	nee
3-6	45,15	nee

Tabel 6.1: De verschillende paren papierfragmenten gesorteerd op hun kost

Figuur 6.1: Het papierfragment gebruikt bij het testen



Figuur 6.2: De handmatige reconstructie van het papierfragment



Figuur 6.3: Voorbeelden van best passende delen bij papierfragmenten

Hoofdstuk 7

Besluit

Het hoofddoel dat we voorop gesteld hadden is het schrijven van een programma dat startend van de scans van de individuele puzzelstukken de topologische oplossing van een legpuzzel kan vinden. De algoritmen en datastructuren die we hiervoor gebruikt hebben werden beschreven in de verschillende hoofdstukken.

We hebben eerst een **oppervlakte-uitbreiding algoritme** geïmplementeerd voor het zoeken van de contour van een individueel puzzelstuk. Hierbij werden geen parameters handmatig ingesteld, de enige parameter werd geschat via een **histogramanalyse**. Omdat het individueel inscannen van de puzzelstukken te tijdrovend bleek voor grotere legpuzzels werden een paar **automatiseringen** doorgevoerd zodat we een scan met meerdere puzzelstukken zonder enige tussenkomst konden verwerken. Dit spaarde veel tijd uit bij het scanproces. Eens we die contouren hadden werd ze opgeslagen in **kettingcodes**. De contour werd dan benaderd door een **polygoon**. Hierbij hebben we rekening gehouden met eventuele ruisobjecten. De polygoon is dan als basis gebruikt voor de **vormextractie**. Dit leverde een kenmerkvector met **differentiële hoeken** en een kenmerkvector met **koorde afstanden** die beide rotatie en translatie onafhankelijk zijn. Er werd ook een **kleurextractie** geïmplementeerd, waarvan de output gesampled werd op het niveau van dezelfde polygoon als bij de vormextractie. Zo waren alle kenmerkvectoren gesynchroniseerd en konden we ze tezamen gebruiken als input van het **benaderende stringmatching algoritme**. Dit benaderende stringmatching algoritme hebben we zeer sterk **geoptimaliseerd** omdat het procentueel een groot deel van de totale tijd van het volledige programma in beslag neemt en we door optimalisatie dus veel tijdwinst maken. Dan hebben we de kosten die we kregen als output bij het benaderende stringmatching algoritme gebruikt bij het zoeken van een **globale topologische oplossing** van de legpuzzel. De eerste stap hierbij was het **indelen in klassen** van de puzzelstukken. De tweede stap was het construeren van de **randtopologie**. Hiervoor werden drie algoritmen geïmplementeerd: het **TSP algoritme**, een eigen **best first algoritme** en een eigen **best confidence first algoritme**. De derde stap was dan het opbouwen van de **interne topologie**. Hiervoor werden twee eigen algoritmen geïmplementeerd: een best first algoritme en een best confidence first algoritme. Het zoeken van de globale topologische oplossing werd ook over de hele lijn opgesplitst in twee versies: een versie waar de **contour als één gesloten geheel** werd beschouwd en een versie waar de **contour opgesplitst** werd **in vier deelcontouren**.

Er zijn dus heel wat algoritmen geïmplementeerd waarbij de rode draad de snelheid en de éénvoud was. Dit leidde uiteindelijk tot het kunnen oplossen van een legpuzzel van 300 puzzelstukken, wat op dit moment de grootste legpuzzel is die automatisch is opgelost met

behulp van een computer. Het matching algoritme dat hierbij gebruikt is, is niet specifiek voor puzzelstukken geschreven. Het is dus een bijzonder goed resultaat vergeleken met de vorige beste prestatie van [2] die een matching algoritme gebruikten dat alleen voor puzzelstukken kon gebruikt worden.

Om te illustreren dat het matching algoritme niet specifiek voor puzzelstukken geconstrueerd is, werd het gebruikt om papierfragmenten met irreguliere vorm met elkaar te matchen. Er werd een procedure beschreven die de eerste stap kan zijn in een semi-automatische reconstructie van allerlei objecten.

De beschreven algoritmen zijn gericht op tweedimensionale figuren, een logische uitbreiding van het geleverde werk kan dus het toevoegen van de derde dimensie zijn. De gebruikte wiskundige ideëen kunnen dan vertaald worden naar hun 3D variant. Ook het ontwikkelen van een volledige tool met grafische interface voor de semi-automatische reconstructie van objecten kan een zeer nuttige uitbreiding zijn.

Voor verdere informatie en voor een overzicht van de gebruikte scripts en originele scans kan u terecht op: *http://telin.ugent.be/~jdebock* .

Bibliografie

[1] H. Bunke, G. Kaufmann, "Jigsaw puzzle solving using approximate string matching and best-first search", Lecture Notes in Computer Science, Computer Analysis of Images and Patterns, D. Chetverikov, W. G. Kropatsch (Eds.), pp. 299–308, Springer-Verlag.

[2] D. Goldberg, C. Mallon, M. Bern, "A Global Approach to Automatic Solution of Jigsaw Puzzles", Proc. of the eighteenth annual symposium on Computational geometry, pp. 82–87, Barcelona, Spain, 2002.

[3] G. F. Shao, F. H. Yao, H. Yamada, K. Kato, "The Computer Solution of Jigsaw Puzzles (Part I – Extraction, Corner Point Detection, and Piece Classification and Recognition", IPSJ SIG Notes: Computer Vision and Image Media (CVIM), 125-1, pp. 1–11, 2001.

[4] F. H. Yao, G. F. Shao, H. Yamada, K. Kato,"The Computer Solution of Jigsaw Puzzles (Part II) – Boundary Shape Matching, Image Merging, and Recovery of Connection Relationships", IPSJ SIG Notes: Computer Vision and Image Media (CVIM), 125-2, pp. 13–23, 2001.