Genereren van een 6-positie postcodebestand op basis van de kadastrale registratie

Afstudeerscriptie Friso Penninga
sectie GIS -Technologie
Afdeling Geodesie

TU Delft

Faculteit Civiele Techniek en Geowetenschappen

Technische Universiteit Delft

Genereren van een 6-positie postcodebestand op basis van de kadastrale registratie

Afstudeerscriptie Friso Penninga

Afstudeerhoogleraar:	prof. dr. ir. P.J.M. van Oosterom
Begeleiders:	ir. E. Verbree
	drs. C.W. Quak

Delft, mei 2003

Sectie GIS-Technologie
Afdeling Geodesie
Faculteit Civiele Techniek en Geowetenschappen
Technische Universiteit Delft

Voorwoord

Deze scriptie is het meest tastbare resultaat van mijn afstudeeronderzoek, zoals ik dat van september 2002 tot mei 2003 heb uitgevoerd bij de sectie GIS-Technologie, afdeling Geodesie, TU Delft. Het afstuderen is het laatste onderdeel in mijn opleiding tot geodetisch ingenieur, die ik in september 1996 gestart ben. Binnenkort eindigt mijn vertrouwde studentenleven om met ir. voor mijn naam de grotemensenwereld binnen te stappen.

In de afgelopen negen maanden ben ik door veel mensen op de een of andere manier geholpen. Sommigen hebben mijn scriptie kritisch doorgelezen, anderen lieten zich gewillig verslaan met een potje tafelvoetbal en weer anderen waren uitstekend gezelschap bij een biertje ter ontspanning op de Beestenmarkt. Drie mensen wil ik echter wel bij name noemen. Ten eerste Peter van Oosterom, die ondanks zijn (voor mij als afstudeerder als verrassing komende) verblijf in Brisbane het project geïnteresseerd heeft gevolgd en bijgestuurd. Ten tweede Edward Verbree, die al bij menig project mijn begeleider is geweest en desondanks weer bereid was deze taak op zich te nemen. Hij heeft continue de voortgang van het project bewaakt. Als laatste wil ik Wilko Quak noemen, die met name op programmeergebied onmisbaar is geweest. Het maakt niet uit met welk probleem je aan komt zetten, telkens weer blijkt Wilko in het verleden al eens wat regels code geschreven te hebben om het probleem op te lossen of te omzeilen. Maar belangrijker dan al deze praktische hulp was de aangenaam ontspannen sfeer waarin Edward, Wilko en ik hebben samengewerkt. Door deze sfeer ben ik mijn afstuderen tot het laatste moment leuk blijven vinden.

Delft, 28 april 2003

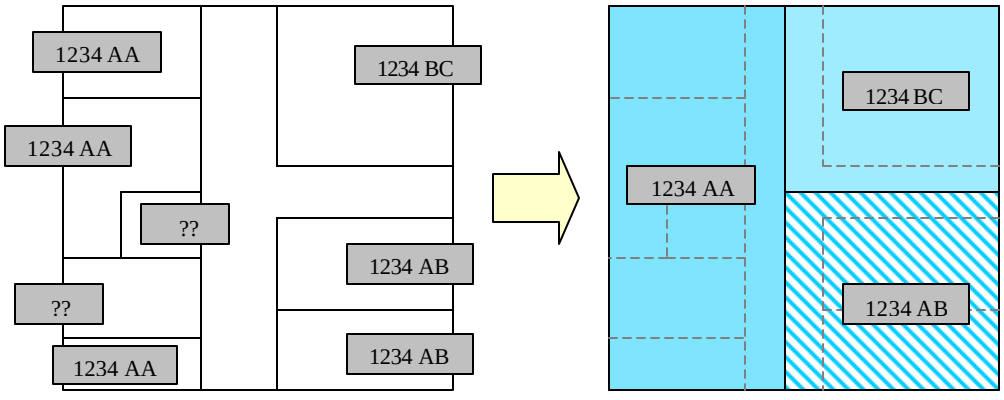
Friso Penninga

Inhoudsopgave

Samenvatting

Kwalitatief goede postcodebestanden spelen een belangrijke rol in veel GIS-toepassingen, aangezien de postcode gebruikt kan worden als koppeling tussen administratieve gegevens enerzijds en een geografische locatie anderzijds. Deze koppeling werkt twee kanten op: van administratieve gegevens naar geografische locatie (geocoderen) en vice versa (adres matching). Geocoderen is de bekendste toepassing, maar adres matching wordt steeds belangrijker, mede door het toegenomen GPS-gebruik. Hierdoor groeit de vraag naar adressen bij een bepaalde geografische locatie. Voor adres matching zijn postcodevlakbestanden het meest geschikt, hierbij is een postcode aan elk gebied gekoppeld. Met een point-in-polygon test kan bepaald worden in welk postcodegebied de geografische locatie valt.

Momenteel laat echter met name de kwaliteit van de 6-positie (vier cijfers + twee letters) postcodevlakbestanden te wensen over. Bij deze postcodebestanden worden de grenzen van de postcodegebieden berekend door gebruik te maken van Thiessen polygonen rond de postcodepunten (gemiddelde ligging van alle adressen met dezelfde postcode). Hierdoor vallen deze berekende grenzen niet samen met in het landschap herkenbare grenzen, zoals wegen en waterlopen. Ook resulteert deze aanpak in postcodegebieden waarvan de grenzen vaak gebouwen doorsnijden. Om deze problemen te ondervangen, is in dit afstudeeronderzoek een algoritme ontwikkeld dat uit gegevens uit de kadastrale registratie (LKI en AKR) een 6-positie postcodevlakbestand genereert. Het idee is dat de eigendomsgrenzen uit het LKI in veel gevallen in het terrein herkenbaar zullen zijn. Op veel percelen staan bovendien objecten waarvan een adres met postcode bekend is. Door nu alle percelen met eenzelfde postcode samen te voegen, ontstaan postcodevlakken die begrensd worden door fysieke grenzen (infrastructuur). Hiertoe speelt skeletteren een belangrijke rol binnen het algoritme: met behulp van deze techniek worden infrastructuur percelen gesplitst en in delen toegevoegd aan de omliggende postcodegebieden. Het idee hierachter is dat huizen aan de ene kant van de weg vaak een andere postcode hebben dan de huizen aan de andere kant. De grens tussen deze twee postcodegebieden ligt gevoelsmatig in het midden van de weg. Daarom worden deze percelen op de hartlijn gesplitst en de twee helften ieder aan het aangrenzende postcodegebied gekoppeld. Het idee is hieronder gevisualiseerd. In deze figuur is ook te zien dat aan percelen zonder postcode de meest waarschijnlijke postcode wordt toegekend.



Figuur 1 Concept: van kadastrale situatie naar postcodegebieden

Doel van het onderzoek was het beantwoorden van de volgende hoofdvraag:

Op welke wijze kan, gebruikmakend van de in de kadastrale registratie aanwezige perceels- en eigendomsgegevens, een kwalitatief betere planaire partitie van door infrastructuur begrensde 6-positie postcodegebieden worden gecreëerd en in welk formaat kan het resultaat, inclusief betrouwbaarheidsindicator, het best opgeslagen worden ?

De wijze waarop deze postcodegebieden gecreëerd kunnen worden, is geïmplementeerd in een algoritme, bestaand uit vijf stappen:

1 Inlezen gegevens uit de kadastrale registratie

Uit de kadastrale registratie wordt voor alle percelen de geometrie en een aantal attributen opgevraagd. Via de AKR-objecten op het perceel zijn nul, één of meerdere postcodes bekend. Aan de hand van de cultuur- en bebouwingscode wordt vastgesteld of het gaat om infrastructuurpercelen die geskeletteerd en gesplitst gaan worden.

2 Skeletteren en splitsen van infrastructuurpercelen

Doel van het skeletteren en splitsen van de percelen is om delen van infrastructuurpercelen te kunnen samenvoegen met omliggende postcodegebieden, analoog aan een region growing operatie. De percelen worden eerst getrianguleerd met behulp van een constrained Delaunay triangulatie, waarbij de perceelsgrenzen als constrained edges worden ingevoegd. Vervolgens wordt op basis van deze driehoeken het skelet berekend door gebruik te maken van de rekenregels van DeLucia en Black (1987), die gebaseerd zijn op het aantal constrained edges in een driehoek. Vervolgens worden een aantal zijtakken berekend die het skelet verbinden met perceelsgrenzen tussen buurpercelen om zo het te splitsen perceel in meer delen op te kunnen splitsen. Met het uitgebreide skelet wordt vervolgens het perceel gesplitst in een aantal kleinere delen. In onderstaande figuur is links de kadastrale situatie weergegeven. Er is een aantal zeer grote percelen zonder postcode (grijs). Deze percelen grenzen aan percelen met tal van postcodes (in pasteltinten). Rechts is het resultaat te zien van skeletteren, splitsen en de volgende stap: het toekennen van postcodes aan (delen van) percelen zonder postcode.

Figuur 2 Links de kadastrale situatie, rechts de resultaten van skeletteren en toekennen van postcodes

3 Toekennen postcodes aan percelen en berekenen betrouwbaarheidindicator

Aan alle (delen van opgesplitste) percelen zonder postcode wordt een postcode toegekend. Hiertoe wordt in een iteratief proces de meest geschikte postcode bepaald aan de hand van de postcodes van de omliggende percelen. Als criterium wordt de lengte van de gezamenlijke grens gehanteerd.

Voor elk postcodegebied wordt de betrouwbaarheidsindicator berekend, die een beeld geeft van de relatieve betrouwbaarheid van de toegekende postcode. Het idee hierachter is dat de betrouwbaarheid van een postcode hoger is als de meeste percelen in het postcodegebied hun postcode rechtstreeks aan de kadastrale registratie ontlenen dan wanneer de meeste percelen pas na meerdere iteratiestappen een postcode toegekend hebben gekregen.

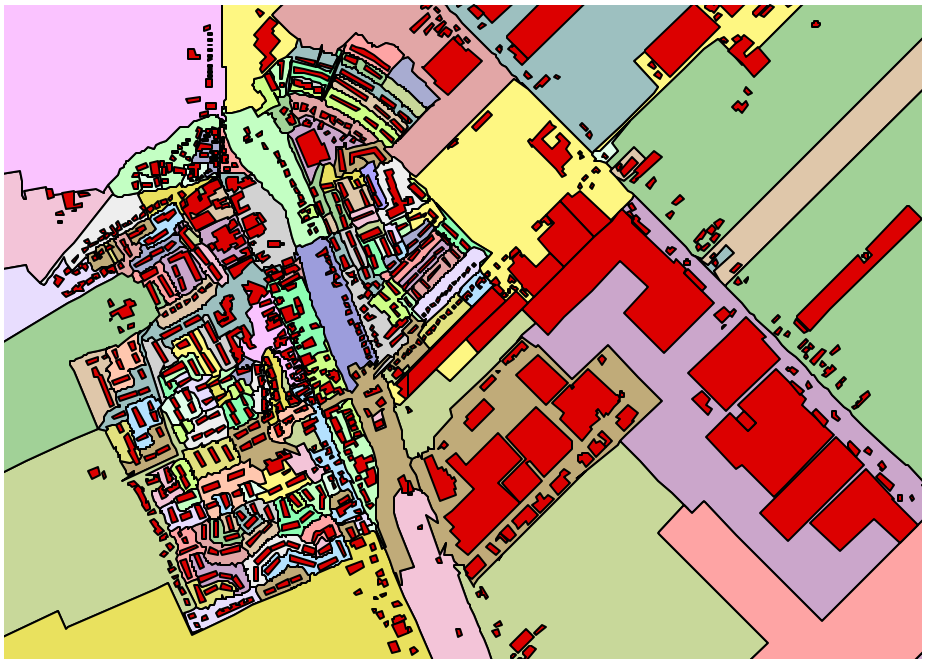
4 Creëren 6-positie postcodegebieden

Nu aan alle (delen van) percelen een postcode is toegekend, kunnen hieruit postcodegebieden worden gegenereerd door alle percelen met gelijke postcode samen te voegen. Aan deze gebieden wordt de betrouwbaarheidsindicator gekoppeld.

5 Exporteren postcodegebieden

De berekende postcodegebieden worden opgeslagen in Oracle Spatial. Op basis van deze tabel kunnen de verschillende door de klant gewenste formaten worden geleverd.

De postcodevlakbestanden die de ontwikkelde methode opleveren, voldoen aan de vooraf gestelde eisen: de meeste postcodegebieden worden begrensd door infrastructuur en bovendien wordt de bebouwing nu vrijwel nooit meer doorsneden door de grenzen van de postcodegebieden. Ter illustratie is in figuur 3 het berekende postcodebestand weergegeven, gecombineerd met de bebouwing uit de Top10Vector.



Figuur 3 Berekende postcodegebieden gecombineerd met de bebouwing uit de Top10Vector

Het algoritme is ontwikkeld in C++, waarbij veelvuldig gebruik is gemaakt van de Computational Geometry Algorithms Library (CGAL). CGAL biedt veel datastructuren en bewerkingen op ruimtelijke objecten. Grootste voordeel van C++ / CGAL is de flexibiliteit waarmee gebruikers de klasses kunnen uitbreiden. Nadeel is echter wel dat het gebruik van CGAL niet altijd even eenvoudig is; ruime ervaring in het programmeren in C++ is een vereiste. Daarnaast geeft CGAL in berekeningen regelmatig precisieproblemen, ook als alle voorgeschreven maatregelen in acht worden genomen. Hierdoor ziet CGAL identieke punten soms als twee verschillende punten, waardoor de Planar Map niet meer valide is.

Summary

Accurate postal code maps play an important role within the geographical information systems as the postal code is used as a link between an address and a geographical location. This linking is known in both directions: linking a location to an address (geocoding) and linking an address to a location (address matching). As geocoding is the most widespread application, the importance of address matching is increasing due to the growing popularity of GPS-receivers. As a result more users need to link an address to their location.

The current postal code map products do not fit to the topography (infrastructure) and are not very accurate. This lack of accuracy is caused by the current method of calculating postal code maps, as these maps are derived by calculating Thiessen polygons based on weighted postal code points. As a result most boundaries of these postal code areas do not fit to actual existing boundaries, such as roads. Often these boundaries intersect buildings. This masters thesis describes an algorithm which is designed to cope with these problems. It derives postal code areas from the Cadastral map and its associated administrative data. Basic idea is that the parcel ownership boundaries are often visible in the terrain. On these parcels objects occur with known addresses including postal code. By joining these parcels with the same postal code, postal code areas are created with boundaries that are visible in the terrain. Skeletonization is one of the key features of the new algorithm as it is used to split road parcels in order to add these separate parts to different postal code areas. This concept is based on the idea that in many cases houses located at one side of a road have a different postal code in comparison to the houses on the opposite side. Instinctively the boundary between these two postal code areas is located at the road centreline. Therefore the left part of the road is added to the postal code area at the left side of the road and the right part to the postal code area at the right side. This concept is visualised in figure 1. It can also be seen that parcels without a postal code are added to the most likely postal code area.

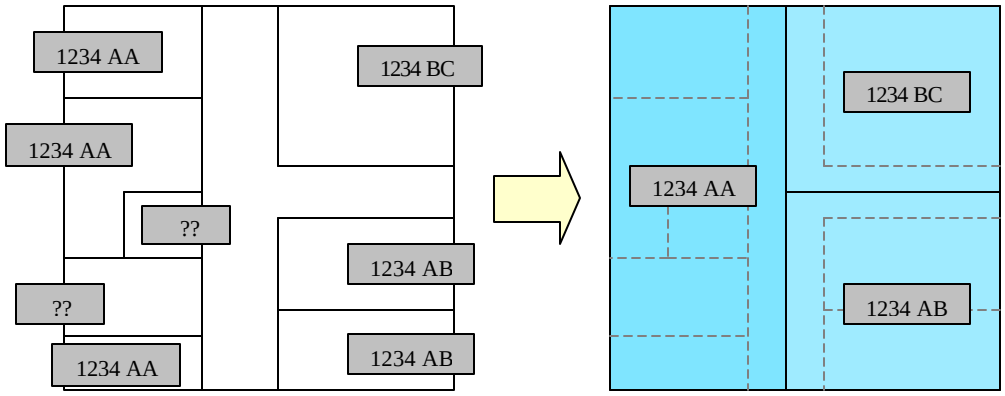


Figure 1 Concept: Cadastral situation (left) and its derived postal code areas (right)

The research objective was to answer the following question:

In which way could an improved planar subdivision of postal code areas, bordered by infrastructure, be derived from the Cadastral map and in which way could the result, including a reliability parameter, be stored most efficiently?

The way in which these postal code areas can be derived is implemented in an algorithm consisting of five steps:

1 Querying the Cadastral map and administrative data

The geometry of each parcel is stored in the database table lki_boundary. Each parcel can be linked to zero, one or multiple postal codes. Based on its 'cultuurcode' (which indicates the land use of the parcel) or its 'bebouwingscode' (which indicates the land use less specific in case there is no 'cultuurcode' added) it is determined for each parcel whether this parcel is infrastructure and therefore has to be skeletonized.

2 Skeletonization

Main objective of skeletonization is to split infrastructure parcels and add the different parts to the neighbouring postal code areas in the same way as a region growing operation would. Each parcel is triangulated in a constrained Delaunay triangulation, inserting the parcel boundaries as constrained edges. As a next step the skeleton is calculated using the method of DeLucia and Black. This method is parameterised by the number of constrained edges in a triangle. After these calculations some extra skeleton branches are calculated to connect the skeleton with the external boundary to enable splitting the parcel in more separate pieces. With this extended skeleton the original parcel is split. Figure 2 illustrates the original Cadastral situation (left) en the results of skeletonization and the next step in the algorithm: adding postal codes to (parts of) parcels without associated postal codes.

Figure 2 Original Cadastral situation (left) and results of skeletonization and adding postal codes (right)

3 Adding postal codes and calculating reliability parameter

For each (part of a) parcel without a postal code the most likely postal code is determined. To do so the postal code is determined based on the postal codes of the adjacent parcels. As a criterion the largest sum of common boundary lengths is used. Based on the number of iteration steps needed the reliability of each postal code area is calculated.

4 Creating 6-position postal code areas

After step 3 all parcels have a postal code. Postal code areas are obtained by merging all areas with the same postal code.

5 Exporting results

All postal code areas are exported to a database table in Oracle Spatial. From this table the postal code map products can be extracted in different GIS-formats, depending on the customers wishes.

The postal code map created with the newly developed algorithm meets the requirements defined in advance. Most postal code areas are bordered by infrastructure. Buildings no longer intersect with the boundaries of the postal code areas. Figure 3 shows the calculated results, combined with the buildings layer from the Top10Vector.

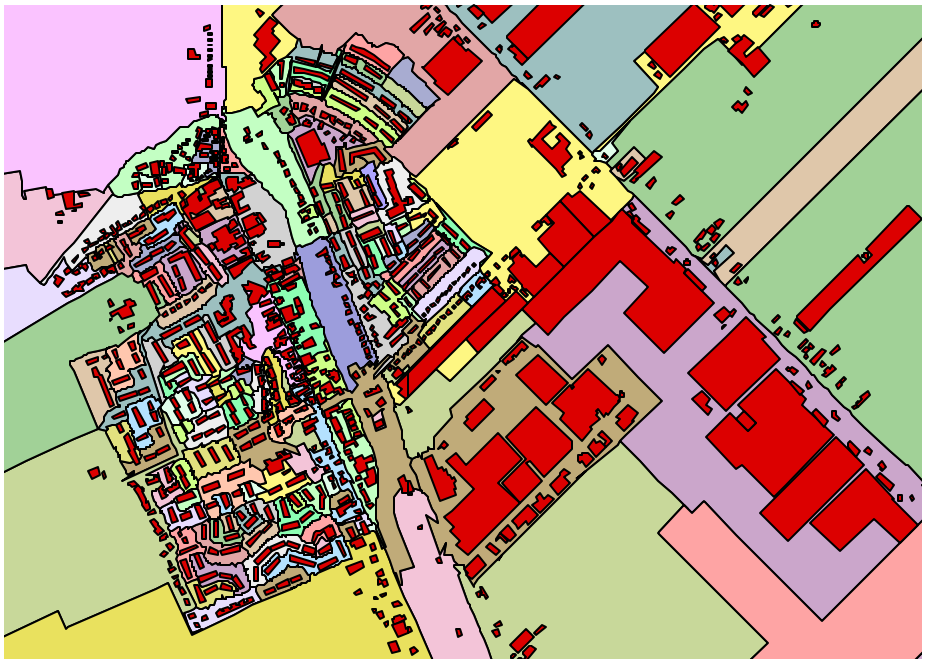


Figure 3 Calculated postal code areas combined with the buildings layer from the Top10Vector

The algorithm is implemented using the Computational Geometry Algorithms Library (CGAL). CGAL supplies several geometric datastructures, a large number of geometric objects and many operations on these objects. The major advantage of C++ / CGAL is its flexibility. Major disadvantage is its complexity. Due to rounding errors CGAL considered identical points as not identical with an invalid Planar Map as a result. It's not easy to use for programmers with little C++ experience.

1 Inleiding

1.1 Aanleiding voor het onderzoek

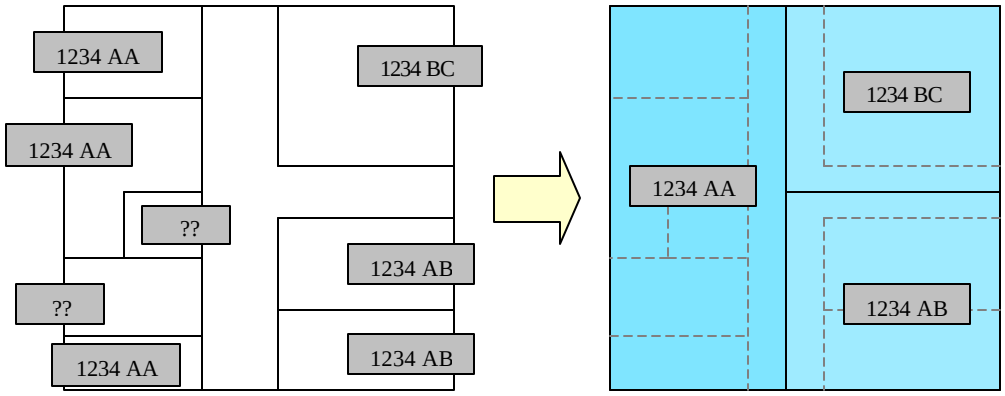
Kwalitatief goede postcodebestanden spelen een belangrijke rol in veel GIS-toepassingen, aangezien met behulp van deze postcodebestanden een koppeling kan worden gelegd tussen administratieve gegevens enerzijds en een geografische locatie anderzijds. Hierdoor wordt het mogelijk gemaakt om ruimtelijke analyses op administratieve gegevens uit te voeren. Er bestaan twee typen postcodebestanden: vlakbestanden, waarin een postcode aan een gebied gekoppeld wordt, en puntbestanden, waarin een postcode aan een punt wordt gekoppeld. Deze bestanden zijn verkrijgbaar in verschillende detailniveaus, namelijk de 4-, 5- en 6-positie postcodebestanden. Dit wil zeggen dat ze gebruik maken van de vier cijfers uit de postcode respectievelijk van vier cijfers en de eerste letter respectievelijk van de gehele postcode.

Momenteel laat echter met name de kwaliteit van de 6-positie postcodevlakbestanden te wensen over. Bij deze postcodebestanden worden de grenzen van de postcodegebieden berekend door gebruik te maken van Thiessen polygonen rond de postcodepunten. Hierdoor vallen deze berekende grenzen niet samen met fysieke grenzen, zoals wegen en waterlopen. Bovendien doorsnijden deze berekende grenzen vaak gebouwen.

1.2 Doel van het onderzoek

Om deze problemen te ondervangen, wordt in dit onderzoek een algoritme ontwikkeld dat uit gegevens uit de kadastrale registratie (LKI en AKR) een 6-positie postcodevlakbestand genereert. Doordat binnen dit algoritme een vlakgericht bestand (een postcodevlakbestand) wordt afgeleidt uit een ander vlakgericht bestand (de kadastrale registratie), is het mogelijk om de postcodevlakken te laten begrenzen door fysieke grenzen, dit in tegenstelling tot de huidige praktijk waarin dit door het gebruik van puntbestanden als uitgangspunt niet mogelijk is. Door deze nieuwe aanpak moet het mogelijk zijn een kwalitatief betere postcodekaart af te leiden.

Het idee is dat in de kadastrale database gedetailleerde informatie beschikbaar is over percelen, hun ligging en via de eigendomsregistratie ook de bijbehorende adressen. Met behulp van de aan percelen gekoppelde cultuurcode kan verder worden vastgesteld of een perceel deel uitmaakt van de infrastructuur die we nu als grens van een postcodegebied willen gebruiken. Het moet mogelijk zijn om uit deze informatie een postcodevlakbestand af te leiden. Dit idee is geïllustreerd in figuur 1.1, waarin ook te zien is dat percelen zonder postcode worden toegedeeld aan het postcodevlak waar ze het meest waarschijnlijk bijhoren.



Figuur 1.1 Afleiden postcodevlakken uit kadastrale registratie

Deze werkwijze wordt ook voorgesteld in Schut en Maessen (2000). Met het voorgaande in gedachten is de volgende hoofdvraag van het onderzoek gedefinieerd:

Op welke wijze kan, gebruikmakend van de in de kadastrale registratie aanwezige perceels- en eigendomsgegevens, een kwalitatief betere planaire partitie van door infrastructuur begrensde 6-positie postcodegebieden worden gecreëerd en in welk formaat kan het resultaat, inclusief betrouwbaarheidsindicator, het best opgeslagen worden ?

1.3 Structuurbeschrijving

Zoals gezegd is het doel van het onderzoek het ontwerpen en implementeren van een algoritme dat een 6-positie postcodebestand berekent op basis van de kadastrale registratie. Voordat de resultaten hiervan worden gepresenteerd, zal in hoofdstuk 2 eerst dieper ingegaan worden op postcodebestanden en hun rol binnen de geografische informatiesystemen. Binnen het algoritme ligt de nadruk op het skeletteren en splitsen van infrastructuurpercelen om zo tot fraaiere grenzen te komen. Vanwege deze prominente rol wordt in hoofdstuk 3 de keuze voor en theorie van skeletteren behandelt. Het algoritme wordt geïmplementeerd met behulp van CGAL, een wetenschappelijke algoritmebibliotheek in C++. Hoofdstuk 4 gaat in op de achtergronden en kenmerken van deze algoritmebibliotheek. Na deze drie theoretische hoofdstukken wordt het ontwikkelde algoritme gepresenteerd. Allereerst worden de benodigde gegevens uit de kadastrale registratie bevraagd, onderwerp van hoofdstuk 5. Op basis van deze gegevens worden in hoofdstuk 6 in een aantal stappen de daadwerkelijke 6-positie postcodegebieden berekend. Deze resultaten worden geëxporteerd en geanalyseerd in hoofdstuk 7. De scriptie sluit af met conclusies en aanbevelingen in hoofdstuk 8. De volledige C++-code is te vinden in Appendix A. Appendix B gaat verder in op de opgedane ervaringen met CGAL.

2 Postcode & GIS

2.1 Inleiding

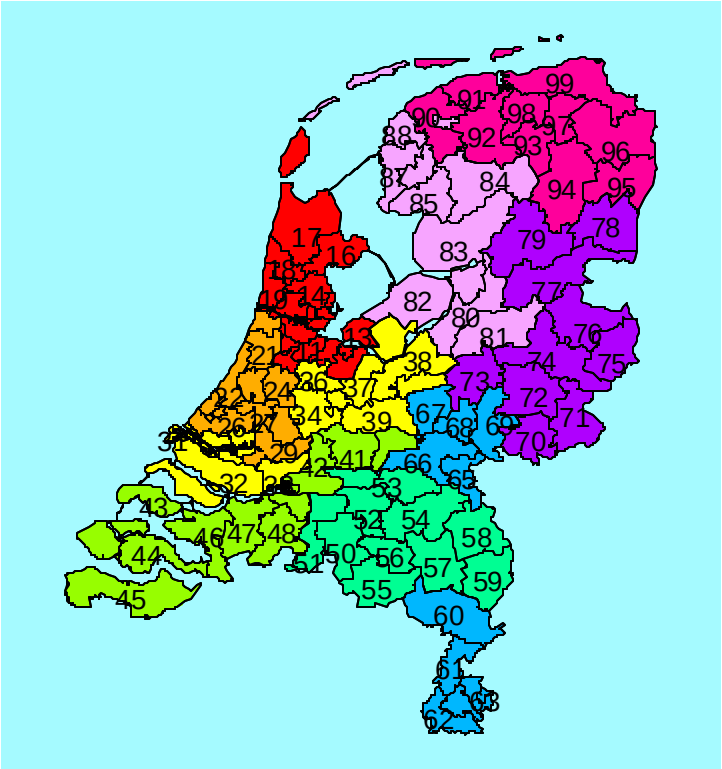
Dit hoofdstuk gaat nader in op het gebruik van postcodes binnen geografische informatiesystemen. De postcode is ontwikkeld met het oog op het automatisch sorteren van post. Deze ontstaansgeschiedenis van de postcode wordt behandeld in paragraaf 2.2. Sinds de invoering van de postcode zijn ook tal van geografische toepassingen bedacht; deze toepassingen en de daarvoor benodigde bestanden die postcodes koppelen aan een locatie komen in paragraaf 2.3 aan bod. Paragraaf 2.4 geeft een overzicht van de belangrijkste aanbieders van postcodebestanden en hun producten. Naar aanleiding van de beperkingen van deze producten wordt in paragraaf 2.5 een alternatieve methode geïntroduceerd. In paragraaf 2.6 komen een aantal commerciële toepassingen van administratieve postcodebestanden aan de orde, omdat deze niet-ruimtelijke toepassingen hebben bijgedragen aan de populariteit van de postcode als ontsluiting van informatie. Het hoofdstuk sluit af met een samenvattende conclusie; paragraaf 2.7.

2.2 Geschiedenis van de postcode

In Nederland is eind jaren zeventig een fijnmazig systeem van postcodes ingevoerd om de afhandeling van post te versnellen. Sinds een aantal jaren was de PTT aan het experimenteren met de invoering van postsorteermachines. Toen het succes van dergelijke machines boven handmatige postsortering bleek, ontstond de behoefte aan een methode voor het automatisch herkennen van de adressering. Probleem hierbij was de veelheid aan schrijfwijzen voor straatnamen en huisnummers (http://members.home.nl/jtv/postcode3.html). Om de automatische adresherkenning te vereenvoudigen, is gekozen voor invoering van de postcode; een combinatie van vier cijfers en twee letters. Door deze keuze zijn er ruim zes miljoen verschillende postcodes beschikbaar (volgens afspraak mag de postcode niet met een 0 beginnen). De postcodes zijn op zodanige wijze aan adressen toegekend, dat elke postcode ongeveer even veel post ontvangt. Dit is gedaan door voor de invoering te onderzoeken welke indeling gehanteerd werd bij de handmatige sortering in de postkantoren in het land. Het idee was dat de handmatige sorteerervaring had geleid tot een systeem waarin de vakken in de houten sorteerkasten optimaal benut werden. Tegenwoordig gaan achter elke postcode gemiddeld 16 postadressen schuil. Dit gegeven, gecombineerd met het aantal mogelijke postcodecombinaties en het gegeven dat er in Nederland momenteel zo'n 7,5 miljoen huishoudens zijn, geeft aan dat het systeem van postcodes nog lang mee kan zonder dat uitbreidingen noodzakelijk zijn.

Bij de invoering van de postcode is ervoor gekozen om een van groot-naar-klein-benadering te gebruiken. Nederland is eerst in negen gebieden ingedeeld (het eerste cijfer van de postcode), elk gebied is weer onderverdeeld in tien gebieden (het tweede cijfer), die elk op hun beurt weer zijn onderverdeeld, enzovoort. Om een idee te geven van de indeling in postcodegebieden van Nederland is in figuur 2.1 een kaart van Nederland te zien, opgesplitst op basis van de eerste twee cijfers van de postcode.

Afhankelijk van het niveau van indeling wordt gesproken over 4-positie postcodes (alleen de vier cijfers), 5-positie postcodes (vier cijfers plus de eerste letter) en 6-positie postcodes (de volledige postcode). Momenteel zijn er in Nederland ruim 420.000 verschillende (6-positie) postcodes in gebruik.



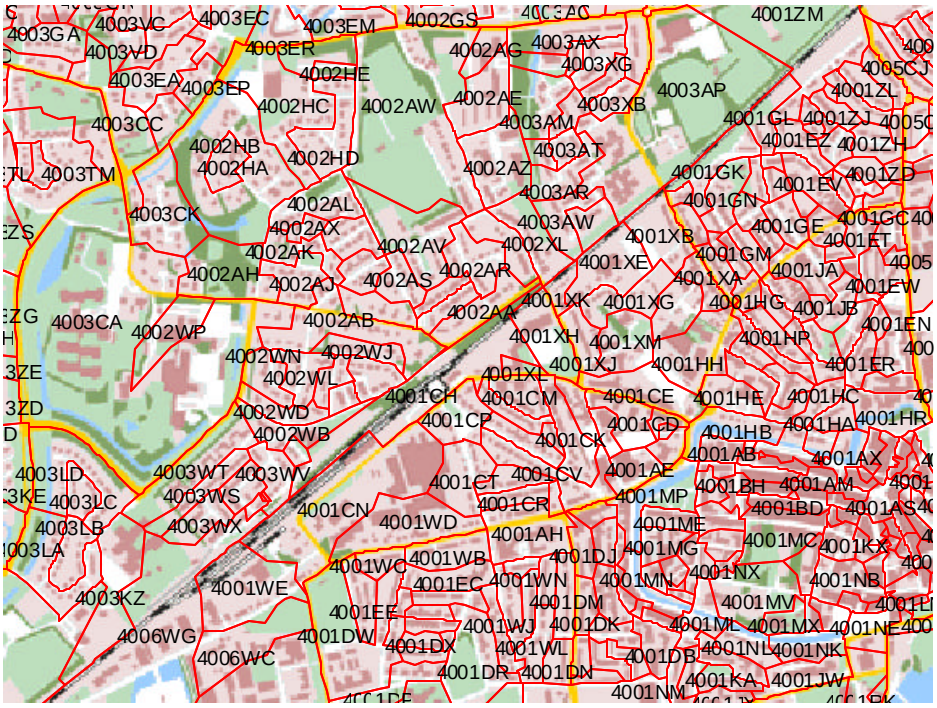
Figuur 2.1 Postcodekaart Nederland op basis van de eerste twee cijfers van de postcode (ingekleurd op basis van het eerste cijfer) (bron: Geodan)

2.3 Postcodebestanden: geografische toepassingen van de postcode

De ruimtelijke component van gegevens staat centraal in de geografische informatiesystemen. Aangezien aan veel gegevens niet direct een geografische locatie is gekoppeld, bijvoorbeeld in de vorm van een x,y-coördinaat, is het nodig om op een andere manier de koppeling tussen gegevens en de geografische locatie vast te leggen. De postcode bleek hiervoor zeer geschikt. Veel gegevens zijn immers aan een adres gekoppeld en van dat adres is de postcode bekend. Het enige wat nu nodig is om de koppeling tussen de administratieve gegevens en de geografische locatie tot stand te brengen, is een bestand dat aan elke postcode een locatie koppelt: de geografische postcodebestanden.

Geografische postcodebestanden zijn er in twee typen: de postcodepuntbestanden en de postcodevlakbestanden. Bij het eerste type wordt voor elke postcode het gemiddelde x,y-coördinaat berekend, zodat een verzameling punten ontstaat die elk aan een postcode zijn gekoppeld. In paragraaf 2.4 wordt nader ingegaan op de wijze waarop deze postcodepunten worden berekend. Bij het tweede type is Nederland opgedeeld in vlakken en is aan elk vlak een postcode gekoppeld.

In de figuren 2.2 en 2.3 is van beide typen een voorbeeld afgebeeld. Afhankelijk van de gewenste nauwkeurigheid van de geografische locatie kunnen 4-, 5- of 6-positie postcodebestanden gebruikt worden. Deze bestanden bevatten echter niet alle postcodes; de postcodes van de ruim 30.000 postbussen ontbreken.

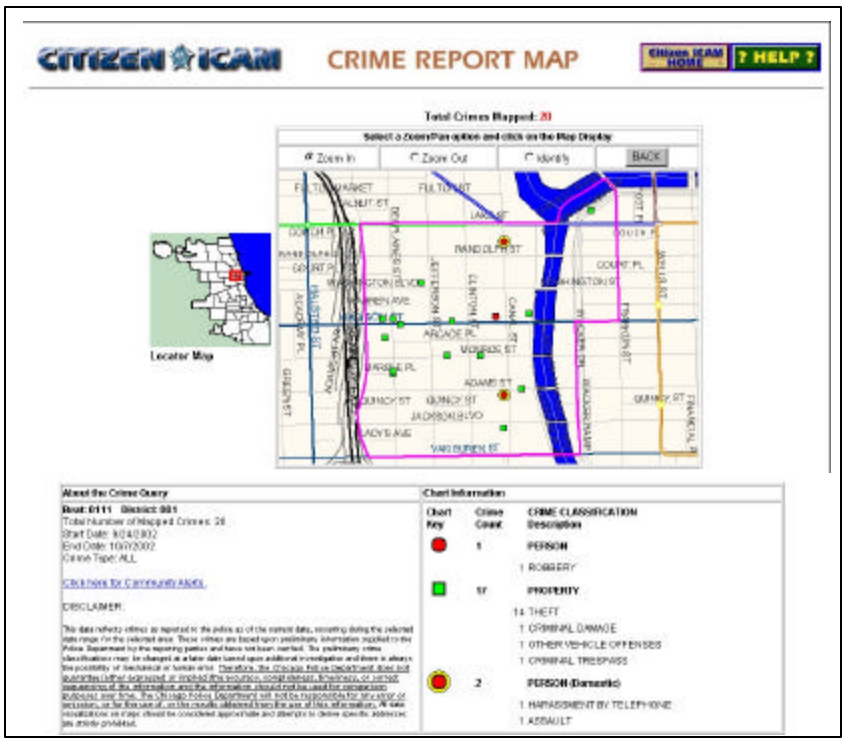


Figuur 2.2 6-positie postcodevlakbestand van Tiel met Top50-achtergrond (bron: Bridgis)

Figuur 2.3 6-positie postcodepuntbestand van Tiel met Top50-achtergrond (bron: Bridgis)

De beschreven koppeling tussen administratieve gegevens en geografische locatie kan twee kanten op werken: van adres naar locatie (het zogenaamde geocoderen) en van locatie naar adres (het zogenaamde adres matching). De eerste toepassing, het vinden van een locatie bij een adres, wordt voor tal van toepassingen gebruikt. In onderstaand kader is een voorbeeld weergegeven, afkomstig van een website van de politie van Chicago. Andere bekende applicaties die gebruik maken van geocoderen zijn routeplanners: met behulp van de postcode worden het aankomst- en vertrekadres omgezet in x,y-coördinaten, die vervolgens als invoer worden gebruikt in een netwerkanalyse, om zo tot een kortste of snelste route te komen.

Voorbeeld: Citizen Icam

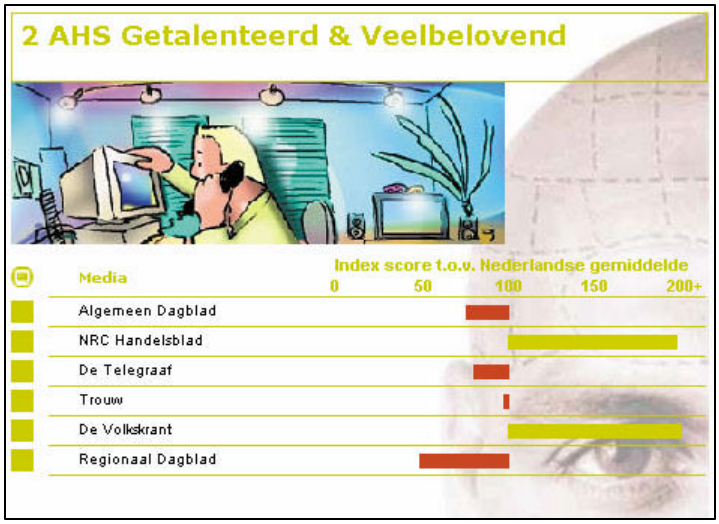


Met behulp van de web-GIS applicatie Citizen Icam (http://12.17.79.6/) kunnen inwoners van Chicago per tijdsblok van 14 dagen een ruimtelijk overzicht krijgen van alle gepleegde delicten. Zo is te zien dat in het geselecteerde gebied één persoon is beroofd, 14 diefstallen zijn gepleegd en één iemand telefonisch is lastig gevallen.

Naast het gebruik van postcodes om gegevens aan één punt te koppelen, wordt de postcode ook vaak gebruikt om gegevens aan een gebied te koppelen. Vaak gaat het hierbij om statistische gegevens, zoals bijvoorbeeld gemiddeld inkomen, gezinssamenstelling of lifestyle. In het kader op de volgende pagina wordt verder ingegaan op de Prizm Viewer van Claritas, een commerciële aanbieder van consumentengegevens (http://www.claritas.nl).

Andere populaire toepassingen van het koppelen van gegevens aan een postcodegebied liggen op de huizenmarkt. Zo is het dankzij de Kadata 'gemiddelde koopsom per 6-positie postcodegebied'-internetservice voor potentiële huizenkopers mogelijk om een overzicht te krijgen van de transactiesommen in de afgelopen vijf jaar (http://kadata.kadaster.nl). Hierdoor kan men zich een goed beeld vormen van de prijzen in het desbetreffende postcodegebied.

Voorbeeld: PRiZM Lifestyle Viewer

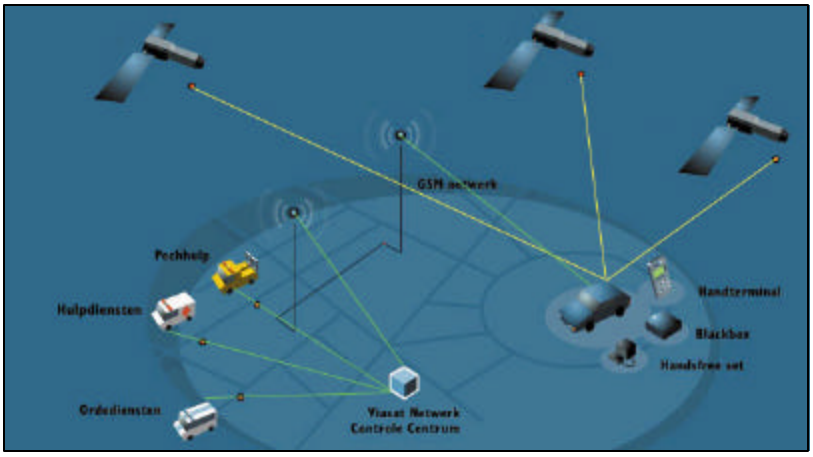


De Prizm Viewer (http://www.prizm.nl) is een applicatie die per 6-positie postcode de top-5 meest voorkomende lifestyles beschrijft (van de 41 verschillende lifestyles). Als test is de postcode van de auteur (woonachtig in een studentencomplex) ingevoerd. In dit postcodegebied blijkt de lifestyle "Getalenteerd en Veelbelovend" het meest voor te komen. Deze lifestyle wordt onder andere gekenmerkt door "zeer jonge huishoudens zonder kinderen", "inkomen beneden modaal" en "hoog opgeleid". In de categorie Media blijken NRC Handelsblad en de Volkskrant zeer veel gelezen te worden en is de VPRO de populairste omroep. Tot slot leert de beschrijving dat het percentage caravan- en vouwwagenbezit flink onder het landelijk gemiddelde ligt, maar het PC-bezit juist weer ruim daarboven.

De tweede internetservice is te vinden op funda.nl (http://www.funda.nl), de internetsite waarop het volledige woningaanbod van NVM-makelaars is samengebracht. Bij elke woning is het mogelijk om op basis van de 6-positie postcode statistieken over de buurt op te vragen, zoals gezinssamenstelling, opleidingsniveau en inkomen. Dergelijke profielen dienen ook als uitgangspunt voor vestigingsplaats-onderzoek. Hierbij wordt gezocht naar de optimale vestigingslocatie voor een bedrijf op basis van afstand tot de doelgroep. Zo zal McDonalds zich graag dicht bij gezinnen met jonge kinderen vestigen en mikt een duurdere supermarkt als Albert Heijn op de ruime aanwezigheid van de hogere inkomens.

Zoals reeds gezegd kunnen postcodebestanden ook gebruikt worden om juist een adres aan een locatie te koppelen, het adres matching. Weliswaar wordt deze toepassing nog niet zo vaak gebruikt als de koppeling locatie aan adres, maar deze toepassing wordt steeds belangrijker. Dit is met name toe te schrijven aan het toegenomen GPS-gebruik, zowel in de vorm van hand-held ontvangers als in autonavigatiesystemen. Hierdoor komen vaker verzoeken om hulp binnen in de vorm van een geografische locatie (een x,y-coördinaat). De hulpverleners zijn echter gewend aan een beschrijvende locatie (een adres). Dankzij een postcodebestand kan de geografische locatie worden omgezet in een beschrijvende locatie. Deze toepassing maakt deel uit van de Viasat-service (zie het kader op de volgende pagina).

Voorbeeld: ViaSat



Viasat (http://www.viasat.nl) is een service waarbij een GPS en GSM in een auto worden ingebouwd. In geval van pech kan via de centrale een garage worden gewaarschuwd. De bestuurder van de takelwagen heeft echter niets aan een geografische locatie (het x,y-coördinaat), maar prefereert een beschrijvende locatie (een straatnaam). Via dezelfde service zijn ook de politie of een ambulance te alarmeren, waarbij de positie ook weer in een adres wordt omgezet. Het is zelfs mogelijk om na diefstal op afstand de startonderbreking te activeren, waarna dankzij het GPS-systeem het adres kan worden gemeld waar de auto is terug te vinden.

Een andere toepassing is te vinden in de watersport. Veel booteigenaren hebben inmiddels de beschikking over een GPS-ontvanger. Mocht men op een rivier in de problemen komen, dan is het nu vaak moeilijk om de locatie aan wal te bepalen. Aan de hand van een GPS-coördinaat is het mogelijk om via postcodebestanden de dichtstbijzijnde straatnaam aan de hulpdiensten door te geven.

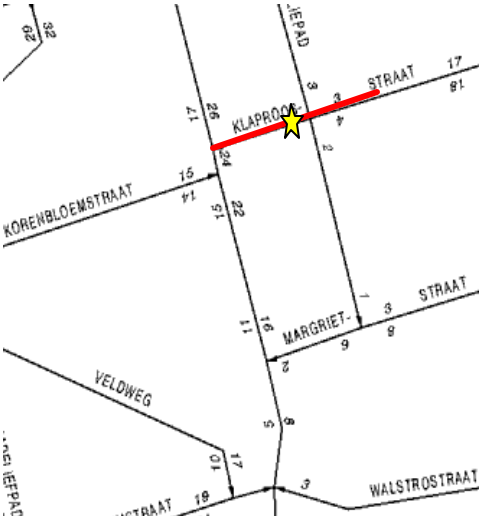
Nu bekend is dat de koppeling tussen beschrijvende locatie en geografische locatie in beide richtingen functioneert, rijst de vraag welke vorm (puntbestand of vlakbestand) de postcodebestanden dienen te hebben om deze koppelingen mogelijk te maken. Geocoderen, het koppelen van een geografische locatie aan een beschrijvende locatie, is zowel met een postcodepuntbestand als met een postcodevlakbestand mogelijk, afhankelijk van de vraag of men een x,y-coördinaat wil (bijvoorbeeld routeplanners) of een gebied (bijvoorbeeld de Prizm Viewer). Voor adres matching, het koppelen van een beschrijvende locatie aan een geografische locatie, is een vlakbestand gewenst. Het is weliswaar ook mogelijk om voor een geografische locatie te berekenen welk postcodepunt het dichtste bij ligt, maar het is eenvoudiger en nauwkeuriger om te kijken in welk postcodegebied de geografische locatie valt. Doordat adres matching steeds vaker wordt toegepast, is de vraag naar goede postcodevlakbestanden gegroeid.

2.4 Overzicht Nederlandse markt voor postcodebestanden

De Nederlandse markt voor postcodeproducten wordt bepaald door een aantal grote aanbieders, die zowel zelf postcodebestanden produceren als bestaande producten doorverkopen. Vrijwel alle verkrijgbare producten zijn gebaseerd op één van de volgende drie basisbestanden (http://www.geodan.nl/nl/product/nlpc/index.html):

Andes 6-positie postcodepunten (VSP -Centroid 6S)

Andes werkt nauw samen met de TPG Post Groep en heeft zo direct toegang tot de bron van de postcodes, die in tabelvorm wordt opgesteld. In deze tabellen staat aangegeven welke adressen welke postcode krijgen toegedeeld. Als producent beschikt Andes ook over GeoStreets. GeoStreets is een bestand op straatniveau waarin aan alle wegsegmenten van Nederland een straatnaam en een huisnummerrange is gekoppeld. De huisnummerrange geeft aan welke huisnummers in een (deel van een) straat voorkomen, bijvoorbeeld 31-89. Door nu de tabellen met adressen en postcodes te combineren met de straatnamen en huisnummerranges, kan de ligging van een postcode geïnterpoleerd worden langs de wegsegmenten. In figuur 2.4 is een voorbeeld weergegeven. Stel dat in de Klaproosstraat de huisnummers 2-10 dezelfde postcode hebben. Op basis van de huisnummerrange kan worden berekend waar deze adressen liggen (langs de rode lijn) en vervolgens wordt het zwaartepunt van deze adressen berekend (de gele ster). Dit punt wordt opgenomen in het postcodepuntbestand.



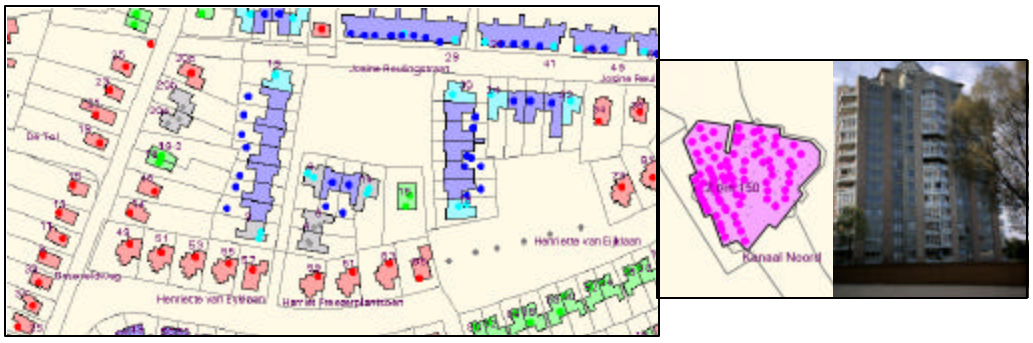
Figuur 2.4 Interpoleren langs een huisnummer-range

Kadata 6-positie postcodepunten

Kadata is een onderdeel van het Kadaster en richt zich op het afleiden van producten uit de vastgoedinformatie waarover het Kadaster beschikt. De postcodepunten zijn berekend door het zwaartepunt te berekenen van alle perceelscentroïden met dezelfde postcode.

Adres Coördinaten Nederland (ACN)

Dit bestand is ook een product van Kadata. De punten geven een adres aan, oftewel de coördinaten van elke combinatie van een 6-positie postcode met een huisnummer. De exacte ligging van de punten is echter niet consequent. Zo is in onderstaande afbeeldingen te zien dat punten op willekeurige plaatsen in of rond een pand gelegen zijn. Zeker de adreslocaties in het appartementencomplex rechts hebben weinig realiteitswaarde. Toch zijn de posities in het ACN daarmee nauwkeuriger dan de postcodepunten die (zoals bij Andes) op basis van huisnummerranges zijn geïnterpoleerd.



Figuur 2.5 Voorbeeld willekeurige plaatsing adrescoördinaten in het ACN (bron: Rengelink, 2000)

Naast de genoemde bestanden bieden Geodan en Bridgis (de twee grote spelers op de Nederlandse markt) ook eigen postcodeproducten aan, die grotendeels op één van de drie basisbestanden zijn gebaseerd. Een overzicht:

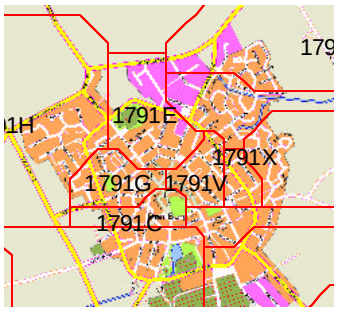
GEODAN

NLP4:

4-positie postcodevlakbestand. Dit bestand wordt door Geodan zelf geproduceerd op basis van de postcodetabellen van de TPG Post Groep. De grenzen worden zo getrokkken dat de vlakken zoveel mogelijk begrensd worden door natuurlijke grenzen als water, wegen en bebouwing. Verder wordt gezorgd dat de grenzen waar mogelijk samenvallen met de burgerlijke gemeentegrenzen. Het bestand bevat ongeveer 4000 postcodevlakken.

NLP5:

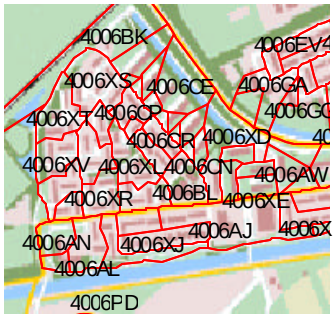
5-positie postcodevlakbestand. Dit product komt tot stand door het combineren van NLP4 met het 5-positie puntenbestand van Andes (een afgeleide van het eerder beschreven 6-positie puntenbestand). Om tot de 5-positie postcodevlakken te komen, wordt binnen de grenzen van de 4-positie postcodevlakken een Voronoi-diagram berekend rond de 5-positie punten. Dit resulteert uiteindelijk in een bestand met ongeveer 35.000 gebieden.



BRIDGIS

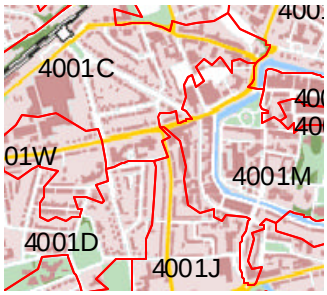
6-positie postcodebestand:

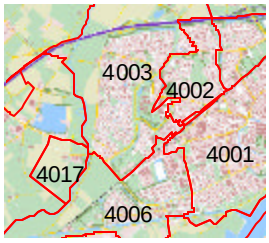
Dit bestand bevat zowel de 6-positie postcodepunten als postcodevlakken. Uitgangspunt voor de productie is het ACN. Waar mogelijk probeert Bridgis de vlakken door infrastructuur te laten begrenzen door gebruik te maken van een wegenbestand van Navigation Technologies. Vaak lukt dit echter niet en herkent men duidelijk grenzen die afkomstig zijn uit een Voronoi-diagram op basis van de 6-positie postcodepunten. Bij nieuwbouwwijken zijn de grenzen echter wel goed, wat doet vermoeden dat bij de jaarlijkse update de nieuwe grenzen handmatig getrokken worden, zodat door infrastructuur begrensde vlakken ontstaan.



5-positie postcodebestand:

Dit bestand bevat wederom postcodepunten en –vlakken. Het is afgeleid uit het 6-positie postcodebestand door vlakken samen te voegen met dezelfde 5-positie postcode. De 5-positie punten worden berekend door alle 6-positie postcodepunten met dezelfde 5-positie postcode te middelen.





4-positie postcodebestand:

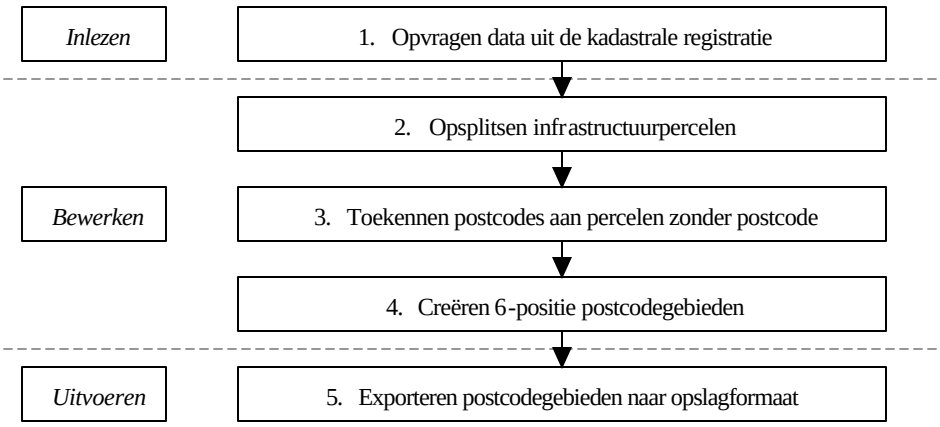
Hier geldt hetzelfde als voor het 5-positie postcodebestand.

2.5 Alternatieve methode voor het bepalen van een postcodevlakbestand

In de vorige paragraaf bleek dat het 6-positie postcodevlakbestand van Bridgis grenzen bevat die duidelijk op basis van een Voronoi-diagram zijn berekend. Dit heeft onder andere tot gevolg dat de grenzen van de postcodegebieden gebouwen doorsnijden. Daarnaast hebben veel van deze grenzen geen duidelijke relatie met de werkelijke situatie. Om dit te voorkomen wordt in dit onderzoek een nieuwe methode ontwikkeld. Deze nieuwe methode, die uitgaat van de kadastrale registratie, komt in paragraaf 2.5.1 aan bod. De keuze om uit te gaan van de kadastrale registratie leidt echter ook tot een aantal beperkingen en randvoorwaarden. Deze zullen in paragraaf 2.5.2 behandeld worden.

2.5.1 Opzet nieuwe methode

Er is gekozen om uit te gaan van de kadastrale registratie, om zo een vlakgericht bestand (het postcodebestand) af te kunnen leiden uit een ander vlakgericht bestand (de kadastrale registratie). De gedachte hierachter is dat de eigendomsgrenzen in de kadastrale registratie in veel gevallen in het terrein terug te vinden zullen zijn. Als de postcode vlakken nu door deze eigendomsgrenzen worden begrensd, zullen de postcodegrenzen daarmee dus ook beter overeenkomen met de werkelijkheid. Doordat op veel kadastrale percelen objecten voorkomen waarvan het adres bekend is, is het mogelijk om voor veel percelen de postcode uit de kadastrale registratie af te leiden. Als nu alle percelen met dezelfde postcode worden samengevoegd, ontstaan vanzelf postcodegebieden. Een beperking hierbij is echter wel dat niet aan alle percelen een postcode gekoppeld is. Daarom zal, voordat de percelen tot postcodegebieden samengevoegd kunnen worden, eerst aan alle percelen zonder postcode een postcode toegekend moeten worden. Daarnaast is het wenselijk om infrastructuurpercelen op te splitsen en de afzonderlijke delen aan postcodegebieden toe te delen, om zo de grenzen van de postcodevlakken zoveel mogelijk te laten samenvallen met de infrastructuur (zie nogmaals figuur 1.1). Dit alles leidt tot een alternatieve methode voor het berekenen van postcodevlakken. Deze nieuwe methode is in figuur 2.6 schematisch weergegeven.



Figuur 2.6 Schematische weergave van het berekenen van postcodegebieden op basis van de kadastrale registratie

2.5.2 Beperkingen en randvoorwaarden

Het is de verwachting dat de alternatieve methode leidt tot een kwalitatief beter postcodevlakbestand, waarbij de vlakgrenzen zo vaak mogelijk overeenkomen met werkelijk bestaande grenzen en deze grenzen niet snijden met bebouwing. De keuze voor de kadastrale registratie als uitgangspunt veroorzaakt echter ook een paar specifieke problemen.

Het belangrijkste probleem is dat er percelen voorkomen waaraan meerdere postcodes zijn gekoppeld. Dit wordt veroorzaakt doordat op één perceel meerdere AKR-objectadressen kunnen voorkomen en deze niet per se dezelfde postcode hebben. Binnen dit probleem kunnen twee situaties onderscheiden worden. In het eerste geval komen er meerdere postcodes voor op één perceel doordat er hoogbouw op het perceel is gerealiseerd. In het tweede geval bevat een perceel zoveel huurhuizen dat er meerdere postcodes voorkomen.

De vraag is of en hoe deze problemen ondervangen kunnen worden. In de ideale situatie is aan elk vlak één postcode gekoppeld. De oplossing kan daarom gezocht worden in het splitsen van het perceel. Additionele informatie zou dan moeten aangeven hoe deze splitsing kan worden uitgevoerd. In het eerste geval, bij hoogbouw, is dit niet mogelijk. Het onderscheid tussen de verschillende postcodegebieden kan alleen gemaakt worden in de derde dimensie en niet in het twee dimensionale postcodevlakbestand. Alhoewel dit probleem dus niet daadwerkelijk is op te lossen, kan de situatie cartografisch wel worden verduidelijkt. Vlakken waaraan meerdere postcodes zijn gekoppeld, kunnen zodanig worden weergegeven dat direct duidelijk is dat er meerdere postcodes zijn gekoppeld. Zo zou het perceel in taartpunten opgedeeld kunnen worden, met aan elke taartpunt een postcode gekoppeld. Een symbool in de kaart kan aangeven dat dit perceel is opgedeeld en de getekende situatie niet in werkelijkheid bestaat. Het geval van de huurwoningen kan gedeeltelijk wel worden opgelost. Zolang er sprake is van laagbouw kan het perceel gesplitst worden door gebruik te maken van het ACN (Adres Coördinaten Nederland). In dit bestand is voor elk adres een positie opgeslagen. Aan de hand van deze posities is af te leiden waar de grens tussen twee postcodegebieden binnen het perceel met huurwoningen getrokken zou moeten worden. Deze methode werkt echter alleen voor laagbouw, bij hoogbouw blijven de problemen bestaan en zou er voor de cartografische verduidelijking gekozen kunnen worden.

Een ander probleem is dat door het uitgaan van kadastrale percelen in plaats van het berekenen van Thiessen polygonen er meerdere vlakken kunnen ontstaan met dezelfde postcode. Dit is met name ongewenst voor het koppelen van een locatie aan een postcode, want welk van de vlakken moet gebruikt worden bij een gegeven postcode? Een eenduidige oplossing hiervoor is niet te geven, met name omdat vooraf niet te zeggen is of dit vaak zal voorkomen. Een laatste beperking is dat niet alle postcodes zullen voorkomen in het postcodebestand. Deze bestanden bevatten namelijk niet de postcodes van de ruim 30.000 postbussen in Nederland, omdat er geen directe relatie bestaat tussen de locatie van de postbus en de achterliggende persoon of instantie. Deze laatste beperking geldt overigens voor alle postcodebestanden, onafhankelijk van de wijze van productie.

2.6 Commerciële toepassingen van administratieve postcodebestanden

De grootste commerciële toepassing van postcodebestanden ligt bij de bedrijven die zich bezig houden met handelsinformatievoorziening. Deze bedrijfstak richt zich op het implementeren van marketing- en klantmanagementsystemen. Uitgangspunt van beide typen systemen is de aanwezigheid van een zeer uitgebreide database waarin per postcodegebied tal van consumenteneigenschappen zijn vastgelegd, waaronder koopkracht, gezinssamenstelling en autobezit. De analysemogelijkheden en effectiviteit van dergelijke systemen nemen toe met de mate van detail van het postcodebestand.

Marketingmanagement systemen kunnen bedrijven helpen om met marketing-activiteiten een hogere penetratie van de doelgroep te bereiken. Zo biedt Experian (http://www.experian.nl), de marktleider in Nederland, applicaties aan om op basis van 4, 5- en 6-positie postcodegebieden reclamemailings zo te adresseren dat het bereik onder de doelgroep gemaximaliseerd wordt bij een gegeven aantal exemplaren. Diverse op Experian's

internetsite beschikbare rekenvoorbeelden laten zien dat met behulp van deze applicatie het bereik onder de doelgroep met ongeveer 25% verhoogd kan worden ten opzichte van een verspreiding in een bepaalde cirkel om een winkelbedrijf. Deze vorm van marketing, die uitgaat van kennis van consumentenprofielen en –gedrag, wordt vaak aangeduid als Consumer Relationship Marketing.

De klantmanagementsystemen gaan nog een stapje verder dan de marketing-managementsystemen. Binnen deze systemen worden de gegevens in de beschikbare consumentenprofielen aangevuld met gegevens over het consumentengedrag, die verkregen worden via de bekende klantenkaarten. Uit deze uitgebreide dataset kunnen huidige en toekomstige consumentenbehoeften worden afgeleid, op basis waarvan bijvoorbeeld het assortiment of reclames kunnen worden toegesneden op de behoefte van de klant. Bekende gebruikers van dergelijke analyses zijn onder andere Albert Heijn, Shell en Mc Donalds. Naast alle gegevens over het klantgedrag richten veel klantmanagementsystemen zich ook op risicomanagement door het toevoegen van kredietinformatie op basis van postcodegebieden. Experian heeft de beschikking over bestanden op 5 positie postcode niveau met betaalgedrag. Op basis van dergelijke informatie kunnen ondernemers aanvullende maatregelen nemen, bijvoorbeeld door niet op krediet te leveren. Deze toepassingen van postcodebestanden zijn niet onomstreden, zoals blijkt uit het krantenartikel in onderstaand kader. Wel dient opgemerkt te worden dat deze toepassingen de postcode als administratief gegeven gebruiken en niet direct als link naar een geografische locatie.

Voorbeeld: Libertel weigert klanten op basis van postcode

door Kees Quaadgras

ROTTERDAM – Naast de Postbank weert ook communicatiebedrijf Libertel op basis van postcodes klanten in zogenoemde achterstandswijken. Nieuwe Westenbewoner W. de Jonge heeft een week lang geprobeerd Libertel ertoe te bewegen hem een abonnement te verlenen, maar dat werd botweg geweigerd. Bij KPN had hij binnen enkele uren wel een abonnement. Een medewerker van telefoonbedrijf BelCompany bevestigt dat Libertel op postcode kan selecteren. Libertels woordvoerder R. Stevens wil het bevestigen noch ontkennen: 'Wij doen geen enkele mededeling over de manier waarop wij de kredietwaardigheid van klanten controleren.' De Consumentenbond is fel tegen het selecteren aan de hand van postcodes.

De Libertelwinkel aan het Beursplein.

Twee weken geleden berichtte deze krant als eerste dat de Postbank een ontheemstanet hanteert voor inboedelverzekeringen, met een daar aan gekoppeld maximaal ruimte. Omdat de risico's in oude wijken groter zijn dan het Nederlandse gemiddelde sluit de Postbank in die wijken geen verzekeringen af. De selectie gebeurt aan de hand van postcodes. W. de Jonge las het verhaal en dacht 'hé, zoiets had ik ook'.

Hij belde Libertel.

De Jonge probeerde vorig jaar voor zijn architectenbureau bij Libertel een abonnement te nemen voor een mobiele telefoon. Hij vulde een aanmeldformulier in en kreeg als antwoord een computerbriefje met de mededeling dat hij was afgewezen. Er stond niet bij waarom. Als hij meer wilde weten kon hij schriftelijk op de afwijzing reageren. 'Of wellicht vindt Libertel Izi uitkomst', aldus een aanvullende mededeling. De laatste betreft mobiele telefonie waarbij voorafbetaald.

De Jonge probeerde telefonisch te weten te komen wat de reden van afwijzing was. 'Nou dat hebben we niet. Daar valt op geen enkele manier over te praten. Je krijgt een of anderen uitzendkracht aan de lijn die je op een botte en kwetsende manier te woord staat. Ze kijken niet of het zakelijk of privé is.

'Dat zou discriminatie zijn'

Desondanks kreeg De Jonge enige informatie los. Bij Libertel Business werd hem meegedeeld dat er twee criteria zijn: kredietwaardigheid en de buurt waar het bedrijf is gevestigd. Met een groot eigen huis, een eigen bedrijf en een gevulde aandelenportefeuille was De Jonge meer dan kredietwaardig. Toch kreeg hij geen abonnement. De conclusie lag voor de hand.

'Ik ben daar een tijd boos over geweest. Als ik schriftelijk had gereageerd zou ik binnen dertig dagen antwoord krijgen. Dan kon ik niet op wachten. Ik moest voor mijn bedrijf bereikbaar zijn. Toen heb ik bij KPN een abonnement genomen. Dat was binnen enkele uren geregeld.'

Bij enkele bedrijven die mobiele telefonie verkopen wordt ongevraagd. 'Daar hebben we nooit van gehoord'. En: 'Dat kan niet. Dat mag discriminatie zijn.'

Bij het Libertel City Point aan het Beursplein: 'Dat kan alleen als iemand niet te verhuizen was en een adres waar een wanbetaler heeft gewoond.'

Bij de BelCompany bevestigt een medewerker dat Libertel inderdaad op postcode kan selecteren. 'Maar daar kan schriftelijk bezwaar tegen worden aangetekend, en in de meeste gevallen wordt het abonnement dan alsnog verleend.' De medewerker noemt vooral hetzelfde argument als bij het Libertel City Point. En: 'Er wordt gekeken of iemand nog openstaande rekeningen heeft.'

'Heel normaal'

R. Stevens van Libertel vertelt dat er een standaard procedure bestaat om de kredietwaardigheid van toekomstige abonnementhouders vast te stellen. Dat is heel normaal. Anderen doen dat ook. Het is een interne procedure waar ik verder geen mededeling over ga doen. En, nadrukkelijk gevraagd naar het gebruik van postcodes: 'Nee, dat wil ik niet ontkennen en ook niet bevestigen.'

P. van den Brandhof van de Consumentenbond laat weten dat het verschijnsel betreffende de selectie naar postcode bij de bond bekend is, al gaat het dan niet specifiek om Libertel, maar om meerdere aanbieders van telefonische diensten. 'Wij zijn daar een fel tegenstander van. Het mag niet zo zijn dat je geen abonnement krijgt, alleen omdat je in zo'n achterstandswijk woont.'

(bron: de Havenloods 10-8-2000)

2.7 Conclusie

In dit hoofdstuk is beschreven hoe de postcode voor steeds meer verschillende doeleinden gebruikt wordt, waarbij de nadruk ligt op de toepassingen binnen de geografische informatiesystemen. De daarvoor benodigde postcodebestanden zijn onderverdeeld in twee soorten: de punt- en de vlakbestanden. Welk van beide typen het meest bruikbaar is, is afhankelijk van het type gebruik. In dit hoofdstuk zijn twee typen gebruik onderscheiden: geocoderen (administratieve gegevens omzetten in een geografische locatie) en adres matching (geografische locatie omzetten in administratieve gegevens). Geocoderen speelt onder andere een belangrijke rol in routeplanners en web-GIS-systemen zoals die van de politie van Chicago. Adres matching wordt nog lang niet zo vaak gebruikt als geocoderen, maar met name door het groeiend GPS-gebruik zijn deze toepassingen aan een opmars bezig. Voor de toepassingen die gebruik maken van geocoderen voldoen postcodepuntbestanden; voor adres matching voldoen deze bestanden echter niet. Postcodevlakbestanden maken het eenvoudiger om te bepalen in welk postcodegebied een geografische locatie valt. De op dit moment verkrijgbare vlakbestanden zijn echter allemaal afgeleid uit puntbestanden, waardoor de berekende grenzen van de postcodegebieden niet overeenkomen met de werkelijke grenzen. Daarnaast snijden deze grenzen vaak met bebouwing. Het gegeven dat de vraag naar goede postcodevlakbestanden toeneemt door het groeiende gebruik van adres matching, is aanleiding om in dit onderzoek te proberen tot een beter postcodevlakbestand te komen door nu niet uit te gaan van een puntbestand, maar van een vlakgericht (kadastraal) bestand.

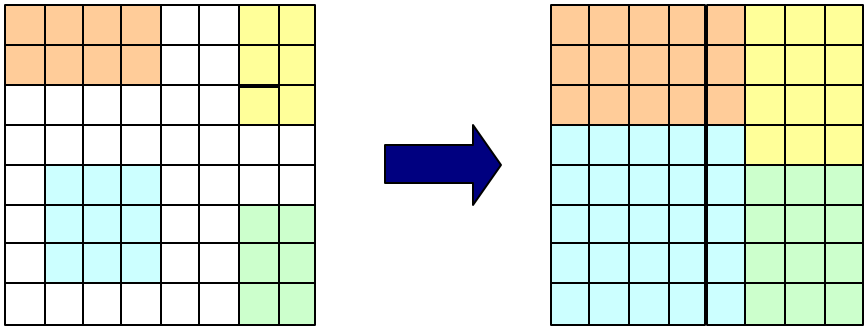
3 Methode voor het splitsen van infrastructuurpercelen

3.1 Inleiding

In hoofdstuk 1 is in een eenvoudige figuur het idee weergegeven achter het algoritme om uit de kadastrale registratie postcodegebieden af te leiden. Doel hierbij is het berekenen van een planaire partitie van door infrastructuur begrensde postcodegebieden. De keuze om de postcodevlakken zoveel mogelijk door infrastructuur te laten begrenzen, sluit aan bij de ervaring dat huizen aan de ene kant van de straat een andere postcode hebben dan de huizen aan de andere kant van de straat. Het is nu de vraag waar de grens tussen die twee postcodegebieden getrokken moet worden, aangezien er door de TPG Post Groep geen echte grens gedefinieerd is. Voor de planaire partitie is echter wel een grens nodig en gevoelsmatig loopt die grens tussen twee postcodegebieden over het midden van de weg. Deze hartlijnen van wegen zijn echter niet bekend in de kadastrale registratie en moeten dus op een andere wijze verkregen worden. Uiteindelijk is gekozen voor het skeletteren van infrastructuur-percelen om zo de hartlijnen te bepalen. In paragraaf 3.2 worden de achtergronden van deze keuze toegelicht. Vervolgens behandelt paragraaf 3.3 twee technieken om een skelet te berekenen en wordt de meest geschikte techniek gekozen. De resultaten van het skeletteren zijn met behulp van verschillende filtertechnieken verder te verbeteren; onderwerp van paragraaf 3.4. Het hoofdstuk sluit in paragraaf 3.5 af met samenvattende conclusies.

3.2 Achtergronden bij de keuze voor skeletteren

De uitgangssituatie voor het berekenen van postcodegebieden is een aantal percelen, waaraan via het AKR-objectadres een postcode is gekoppeld. Vanuit deze situatie moet een planaire partitie van postcodegebieden ontstaan. Intuïtief kan dit gebeuren door alle percelen met een postcode steeds iets groter te laten worden, totdat al deze percelen elkaar raken. In een rasteromgeving is een dergelijke operatie bekend onder de naam region growing. Hierbij groeien objecten steeds doordat de aangrenzende pixels worden toegevoegd aan deze objecten. In figuur 3.1 is hiervan een eenvoudig voorbeeld gegeven, waarin na één stap van toevoegen van pixels alle objecten elkaar al raken. Vaak zullen meerdere iteratiestappen benodigd zijn voordat een volledige planaire partitie is ontstaan.

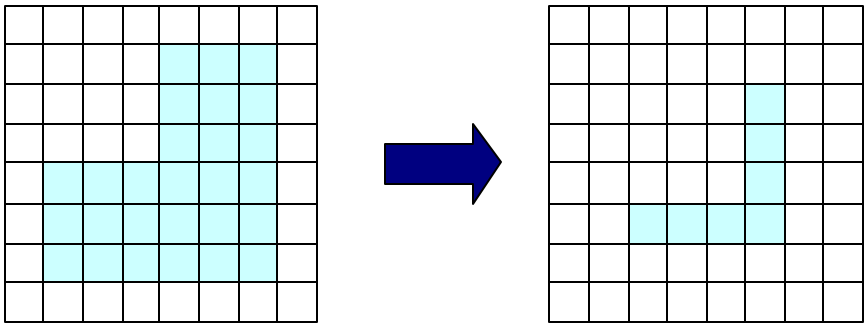


Figuur 3.1 Region growing: vier objecten (in kleur) worden uitgebreid met de aangrenzende pixels

Als in figuur 3.1 de vier gekleurde objecten percelen met verschillende postcodes en de witte pixels infrastructuur voorstellen, vormt het resultaat van deze region growing-operatie een postcodevlakbestand. De vraag is nu hoe een dergelijke operatie er in vectorformaat uit zou

zien. Er zijn duidelijke parallellen zichtbaar met het berekenen van voronoi-diagrammen. Hierbij worden vlakken rond punten gecreëerd die het gebied bevatten dat dichterbij het punt in het vlak ligt dan bij één van de punten buiten het vlak. Region growing doet in principe hetzelfde, alleen wordt nu het gebied bepaald dat het dichtste bij een vlakobject ligt in plaats van bij een puntobject. Door dit verschil is het moeilijk deze operatie in vectorformaat te implementeren. Daarnaast zijn de resultaten van region growing minder fraai als er percelen zijn zonder postcode. In dit geval is het namelijk mogelijk dat delen van een perceel zonder postcode wordt toegedeeld aan verschillende postcodegebieden, waardoor de grenzen van de postcodevlakken geen relatie meer hebben met de werkelijkheid. Aangezien aan veel percelen geen postcode gekoppeld is en het juist het doel van het nieuwe algoritme is om grenzen te laten samenvallen met in werkelijkheid voorkomende grenzen, is deze techniek niet geschikt.

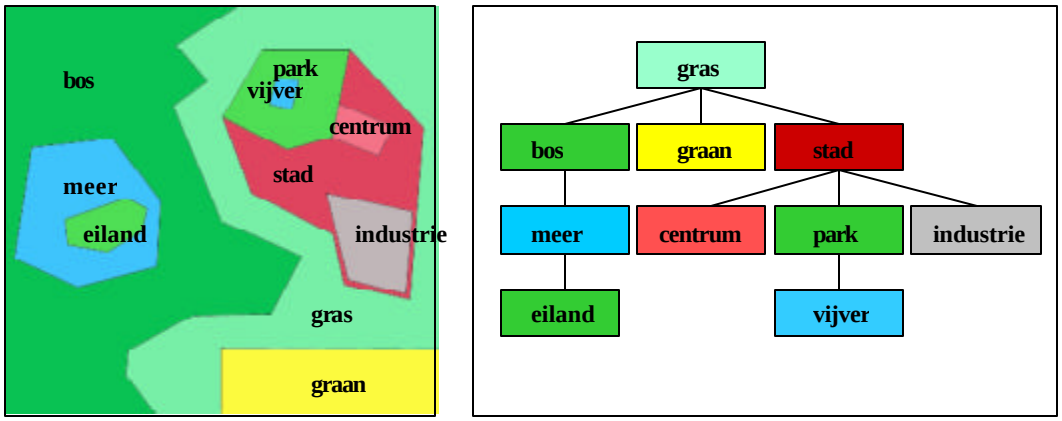
De oplossing is nu om te kiezen voor een combinatie van twee technieken, namelijk collapse en amalgamatie. Dit komt neer op het eerst splitsen van infrastructuurpercelen en vervolgens het toedelen van de afzonderlijke delen aan de omliggende postcodegebieden. Collapse is de vectorequivalent van de rasteroperatie thinning. Thinning is eigenlijk het omgekeerde van region growing: in plaats van het toevoegen van de aangrenzende pixels wordt juist steeds de buitenste rand pixels verwijderd, net zo lang totdat een object één pixel breed is. Een eenvoudig voorbeeld hiervan is weergegeven in figuur 3.2, waarin één iteratiestap nodig is.



Figuur 3.2 Thinning met één iteratiestap

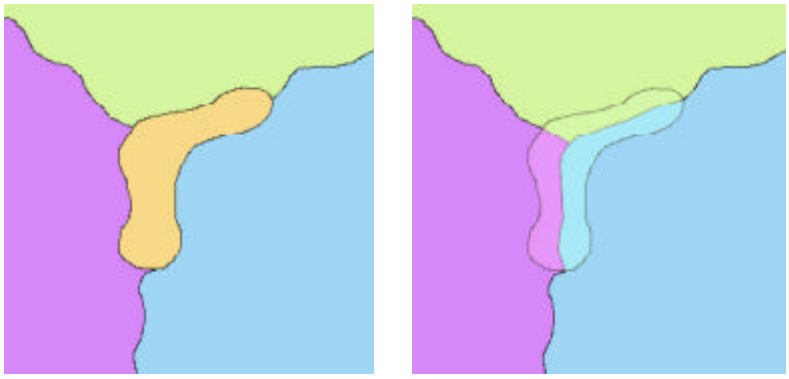
Zoals gezegd is collapse de vectorvariant van thinning. Met behulp van de collapse operatie kunnen de hartlijnen van de infrastructuurpercelen worden bepaald. Door nu deze hartlijnen te gebruiken om de infrastructuurpercelen te splitsen, wordt het mogelijk om deze afzonderlijke delen aan verschillende postcodegebieden toe te kennen. Dit opdelen en toekennen is bekend onder de naam amalgamatie. Binnen kaartgeneralisatie wordt deze operatie gebruikt voor het samenvoegen van objecten op lagere schaalniveaus. Een bekend voorbeeld van deze operatie is de weergave van huizen. Op grote schaal zijn de individuele huizen of huizenblokken te herkennen, terwijl op kleine schaal de gehele bebouwde kom één vlak is. Om dit te bereiken moeten onbelangrijke objecten worden toegevoegd aan belangrijke objecten.

Eén van de beschikbare methoden hiervoor is de GAP-tree (van Oosterom, 1995), een methode waarmee on-the-fly kan worden gegeneraliseerd. Een voorbeeld van een GAP-tree is te vinden in figuur 3.3. Binnen deze tree is vastgelegd hoe de relatie van objecten onderling is. Op het meest gedetailleerde niveau worden alle objecten getoond, op minder gedetailleerde niveaus steeds minder. Zo zal bij een eerste generalisatiestap het eiland niet meer worden afgebeeld, dit object wordt toegedeeld aan het object meer. Weer een generalisatiestap verder zal het meer niet meer worden getoond, maar aan het object bos worden toegevoegd. In de laatste stap zal het hele object bos zelfs verdwijnen en aan gras worden toegedeeld.



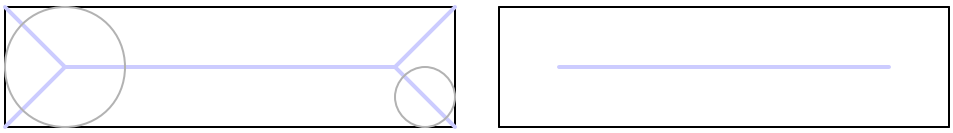
Figuur 3.3 Kaart, bestaande uit 10 objecten, met bijbehorende GAP-tree

Als een object zoals in figuur 3.4 aan meerdere objecten grenst, is het een goede optie om het object met behulp van een skelettering op te delen en vervolgens de afzonderlijke delen aan de buurobjecten toe te voegen. Dit opdelen en splitsen kan ook in real-time gebeuren voor generalisatie toepassingen (Ai en Van Oosterom, 2002). Dit artikel beschrijft ook de amalgamatie van objecten die niet aan elkaar grenzen, maar alleen in elkaars nabijheid liggen. Een toepassing hiervoor is het genereren van een vlakobject gebouwen uit een aantal losse gebouwen. Binnen het uitgevoerde onderzoek is dit interessant voor het opdelen van wegpercelen die aan meer dan twee postcodevlakken grenzen. Opgemerkt wordt dat er parallellen zichtbaar zijn met de rasteroperatoren opening en closing. Opening en closing bestaan beide uit de morfologische transformaties dilatie en erosie. Voor meer details over deze operaties wordt verwezen naar Sonka et al. (1993).



Figuur 3.4 Skeletteren en opdelen aan meerdere buren (amalgamatie)

De hierboven beschreven combinatie van opsplitsen en toedelen is geschikt voor het toepassen op infrastructuurpercelen. Daarom is een methode nodig om de hartlijnen van deze percelen te berekenen. Skeletteren is hiervoor geschikt. Het skelet van een vlak is de lijn (of kromme) waarop de middelpunten van alle cirkels liggen die minimaal twee vlakgrenzen raken. Door deze definitie is een skelet net iets groter dan het resultaat van een collapse operatie. In figuur 3.5 is dit geïllustreerd. Het verschil zit in het begin en einde van het skelet. Het skelet raakt de vier buitenhoeken van het vlak, het resultaat van de collapse operatie niet. Niet alle verschillende methoden onder de noemer ‘skeletteren’ leveren overigens exact een skelet dat aan bovenstaande definitie voldoet.



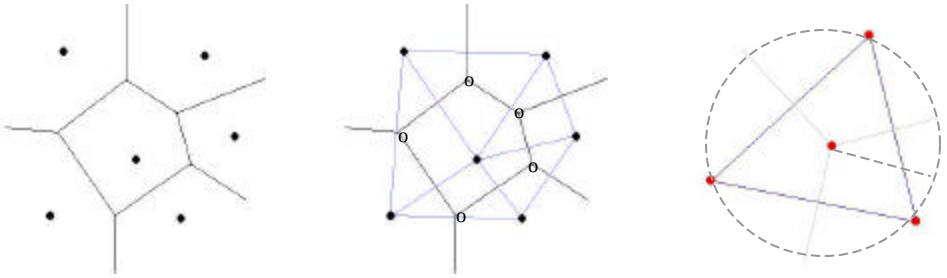
Figuur 3.5 Resultaten skeletteren (links) en collapse (rechts) met in lichtblauw het skelet van het zwarte object

3.3 Twee methoden voor het skeletteren van vlakken

In deze paragraaf komen de twee bekendste methoden voor skeletteren aan bod, namelijk die van Chithambaram (1991) en die van DeLucia en Black (1987). Beide methoden maken gebruik van Voronoi-diagrammen en Delaunay-triangulaties. De achtergronden hiervan worden voorafgaand aan de skeletteer-methoden besproken.

3.3.1 Achtergrond: Voronoi-diagram en Delaunay-triangulatie

Een triangulatie is een onregelmatige vlakverdeling die gebaseerd is op driehoeken. Een dergelijke datastructuur kan op verschillende manieren worden geconstrueerd; in deze paragraaf zal echter alleen de Delaunay-triangulatie worden besproken. Dit is één van de meest gangbare triangulatie-methoden. Om aan te geven hoe op basis van een verzameling punten de triangulatie wordt geconstrueerd, is het nodig om eerst in te gaan op Voronoi-diagrammen. Uitgangspunt voor een Voronoi-diagram is een puntenset. Op basis van deze puntenset wordt een planaire partitie berekend. Rond elk punt wordt het vlak gezocht dat zodanig begrensd is dat elke willekeurige locatie in dat vlak dichterbij het punt ín de veelhoek ligt dan bij de punten buiten de veelhoek. De vlakgrenzen rond een punt zijn hiertoe opgebouwd uit de middelloodlijnen tussen het punt en zijn nabuurpunten. De punten waar deze middelloodlijnen elkaar kruisen, worden Voronoipunten genoemd. Deze geometrische constructie wordt ook wel aangeduid als een Dirichlet-verdeling of als Thiessen polygonen. De relatie tussen een Delaunay-triangulatie en een Voronoi-diagram zit erin dat twee punten in de triangulatie met elkaar verbonden worden als ze in het Voronoi-diagram buren zijn (zie figuur 3.6a. en 3.6b.). Als gevolg van deze constructiemethode geldt het volgende: de drie punten van een Delaunay-driehoek liggen op een cirkel met een Voronoipunt als middelpunt (figuur 3.6c).



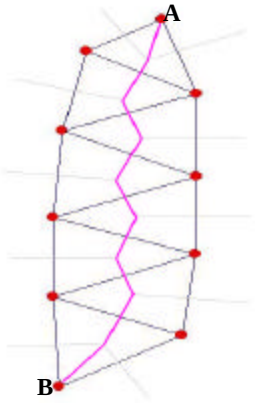
Figuur 3.6a Voronoi-diagram b Delaunay-triangulatie c Relatie Voronoi-punt en Delaunay driehoek

De hierna te bespreken methoden voor het skeletteren van polygonen gaan uit van de verzameling punten die de grenzen van de te skeletteren polygonen opspannen. Op basis van deze punten worden Delaunay-triangulaties en Voronoi-grenzen bepaald. De tweede methode maakt gebruik van een constrained Delaunay triangulatie. Hierin worden de grenzen van het polygoon als edge opgenomen, ook als volgens de Delaunay regels deze edges eigenlijk geen onderdeel van de triangulatie zouden horen te zijn. Hierdoor geldt een belangrijke eigenschap van Delaunay triangulaties niet altijd voor constrained Delaunay triangulaties. Voor Delaunay triangulaties geldt namelijk dat binnen de omschrijvende cirkel

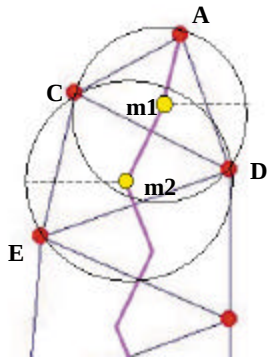
van een driehoek (zoals in figuur 3.6c te zien) geen andere punten van de triangulatie liggen. In de constrained Delaunay triangulatie komen twee soorten edges voor: D-edges, interne edges die punten op verschillende grenzen verbinden en G-edges (de constrained edges), die de grens vormen van het polygoon (Uitermark, 1999; van Oosterom, 1999).

3.3.2 Methode 1: Chithambaram

De eerste skeletteringsmethode wordt gepresenteerd in Chithambaram et al. (1991). Chithambaram hanteert als uitgangspunt dat het skelet van een polygoon verkregen wordt door de grenzen van de polygoon met gelijke snelheid naar binnen te laten bewegen, totdat twee grenzen elkaar raken. Op deze manier stort het polygoon in tot het minder-dimensionale skelet; een bewerking identiek aan de collapse-operator. Dit ineenstorten leidt tot een lijn waarvoor geldt dat alle punten op de lijn op gelijke afstand tot beide grenzen liggen. In de methode Chithambaram laat men ondanks dit uitgangspunt echter niet de grenzen met gelijke snelheid naar binnen bewegen, maar de punten die de grenzen opspannen. Dit komt overeen met het berekenen van Voronoi-diagrammen, aangezien de grenzen in een Voronoi-diagram op gelijke afstand liggen tot de punten aan weerszijden van die grenzen. Alle Voronoi-grenzen die binnen het te skeletteren vlak liggen, maken deel uit van het skelet. Tot slot worden aan het skelet nog twee lijnstukken toegevoegd, namelijk de lijnstukken die de twee eindpunten A en B verbinden met het skelet van Voronoi-grenzen. De punten A en B maken dus ook deel uit van het skelet. Dit is geïllustreerd in figuur 3.7.



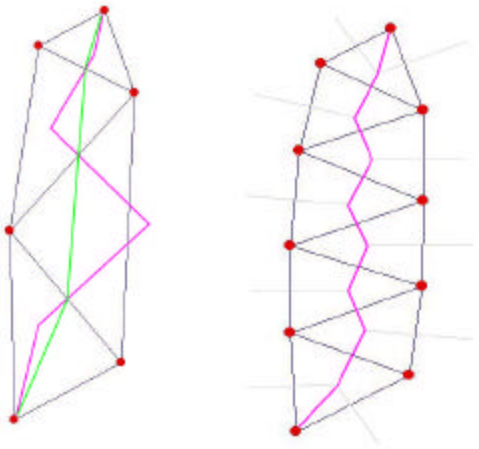
Figuur 3.7 Skelet (paars) volgens de methode Chithambaram



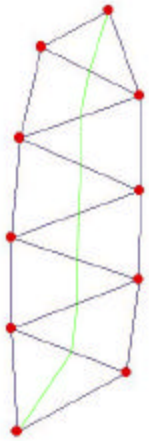
Figuur 3.8 Regenereren van de punten van het vlak aan de hand van het skelet

Het skelet dat resulteert uit de methode Chithambaram heeft een fraaie eigenschap; uit het skelet kan namelijk het oorspronkelijke vlak worden geregenereerd (Chithambaram et al. 1991). Om dit te begrijpen is het goed om te kijken naar de relatie tussen het Voronoi-diagram en de Delaunay-triangulatie. Het Voronoipunt m1 (zie figuur 3.8) is het middelpunt van de cirkel waarop de hoekpunten van de Delaunay-driehoek A, C en D liggen. Deze cirkel heeft straal m1A en middelpunt m1 (punt A is bekend, want het is onderdeel van het skelet). Eenzelfde relatie geldt ook voor het volgende punt van het skelet, m2; de punten C, D en E liggen op een cirkel met middelpunt m2 en straal m2C. Chithambaram stelt voor om bij het skeletteren aan elk punt m ook de straal van de bijbehorende cirkel als attribuut toe te voegen. Met deze informatie is het nu mogelijk om alle punten van het oorspronkelijke vlak te reconstrueren en daarmee het vlak zelf terug te vinden. De werkwijze is als volgt: om punten m1 en m2 worden cirkels getrokken met straal m1A en m2C. Deze cirkels snijden op twee punten: punten C en D. Op deze wijze kan punt voor punt het gehele skelet worden afgelopen, waarna alle punten van het oorspronkelijke vlak bekend zijn.

De methode Chithambaram heeft ook een duidelijk nadeel, dat men zelf niet beschrijft. Men wat goede wil is in het vlak in figuur 3.7 een weg te herkennen als men punten A en B even wegdenkt. Het is nu zeer twijfelachtig of het gevonden skelet een mooie representatie van deze weg is. Immers, de weg is vrijwel recht, maar het skelet heeft een zig-zag-vorm. Deze zig-zag-vorm wordt veroorzaakt door het gebruik van delen van de Voronoi-grenzen. De ligging van de grenzen is afhankelijk van de ligging van de punten. Om de gevolgen hiervan te verkennen is in figuur 3.9 een vrijwel identiek vlak te zien, dat echter opgespannen wordt door een andere puntenverzameling. Voor de vorm van het skelet heeft dit vergaande gevolgen! Twee (vrijwel) identieke vlakken kunnen zeer verschillend worden geskeletteerd, afhankelijk van de verdeling van de punten over dit vlak. Dit leidt tot een belangrijke conclusie t.a.v. de methode Chithambaram: het skelet representeert de punten die een vlak opspannen (het Voronoidiagram wordt immers berekend op basis van die punten) en niet zozeer het vlak zelf. Voor toepassingen waarbij het skelet een (visuele) representatie dient te vormen van een vlak is de methode Chithambaram dus minder geschikt.



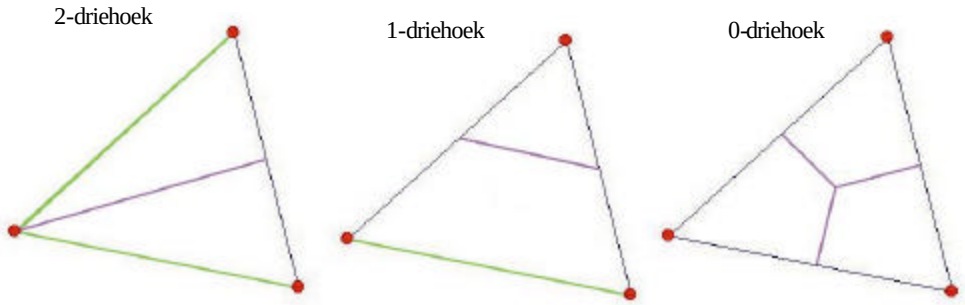
Figuur 3.9 Skeletten volgens Chithambaram (paars) en DeLucia en Black (groen) bij een gewijzigde puntenset, rechts met de originele puntenset



Figuur 3.10 Skelet (groen) volgens methode DeLucia en Black

3.3.3 Methode 2: DeLucia en Black

Een tweede methode om te skeletteren wordt gepresenteerd in DeLucia en Black (1987). Ook bij deze methode is het centrale idee dat de grenzen van een polygoon met gelijke snelheid naar binnen schuiven, waarna het skelet wordt vastgelegd op de punten waar de grenzen elkaar raken. Zowel de methode DeLucia en Black als de methode Chithambaram berekenen eerst de middelpunten op de interne edges. Chithambaram berekende vervolgens echter aan de hand van dit punt de middelloodlijn om zo tot een skelet te komen dat bestaat uit Voronoi-grenzen. Hier wordt dus telkens een lijnstuk toegevoegd aan het skelet. Bij DeLucia en Black worden deze punten echter direct verbonden en aan het skelet toegevoegd. De wijze waarop deze regels met elkaar verbonden worden, hangen af van de hoeveelheid G-edges. In principe zijn er vier mogelijkheden; van elke driehoek kunnen 0, 1, 2 of 3 edges deel uitmaken van de buitengrens van het te skeletteren polygoon. De laatste mogelijkheid, waarbij elke edge deel uitmaakt van de buitengrens, is voor skelettering niet van belang. Een dergelijk vlak zal als punt worden gerepresenteerd. De drie wel ter zake doende mogelijkheden zijn geïllustreerd in figuur 3.11.

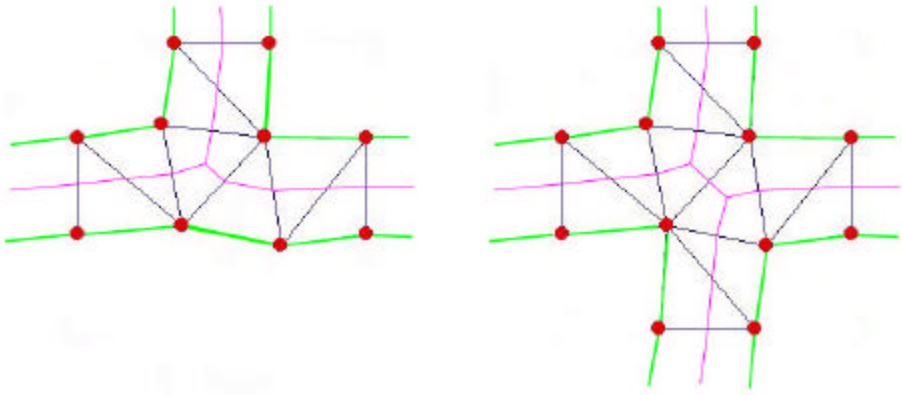


Figuur 3.11 Skeletteerregels van DeLucia en Black (groene edges maken deel uit van de buitengrens)

Uitgangspunt voor alle gevallen is dat het middelpunt van de interne edge(s) wordt berekend. In een 2-driehoek wordt het middelpunt van de interne edge verbonden met het punt van de driehoek dat geen deel uitmaakt van de interne edge. De 1-driehoek illustreert het beste het idee van skelettering. De twee middelpunten van de interne edges worden met elkaar verbonden, waardoor dit deel van het skelet parallel loopt met de buitengrens. Voor het bepalen van het skelet in een 0-driehoek wordt naast het drietal middelpunten van de interne edges ook het middelpunt van de driehoek zelf berekend. De drie middelpunten worden verbonden met het middelpunt van de driehoek.

Om de resultaten van DeLucia en Black met Chithambaram te kunnen vergelijken, is in figuur 3.10 dezelfde vorm geskeletteerd. Duidelijk zichtbaar is dat de zig-zag-vorm die Chithambaram kenmerkte, volledig ontbreekt. In figuur 3.9 is getest hoe het skelet verandert als een vrijwel identieke vorm door een andere puntenset wordt opgespannen. Duidelijk zichtbaar is dat de methode DeLucia en Black niet gevoelig is voor en wijziging van de puntenset, terwijl bij de methode Chithambaram dit tot een ingrijpende wijziging van het skelet leidde. Voor toepassingen van skeletteren waarbij het met name van belang is om de vorm van de polygoon te representeren in plaats van de specifieke punten op de polygoon, is de methode DeLucia en Black duidelijk geschikter dan de methode Chithambaram.

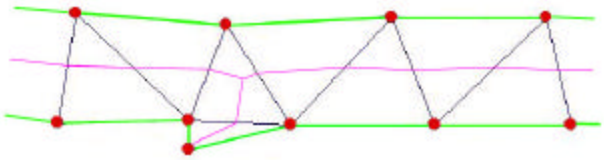
Nu duidelijk is dat de methode DeLucia en Black de betere skeletteermethode is als doel van de skelettering een beschrijving van de vorm van de polygoon is, is het aardig om te kijken naar de werking van het algoritme voor het skeletteren van wegen. In figuur 3.12 zijn een T-splitsing en een kruising geskeletteerd. Aan de hand van deze illustraties kunnen uitspraken gedaan worden over het voorkomen van 0-, 1- en 2-driehoeken. Duidelijk is dat de 1-driehoek het meest voorkomt, eigenlijk overal waar sprake is van twee parallelle grenzen. Op plaatsen waar meerdere wegen bijeen komen, komen 0-driehoeken voor. Een T-splitsing leidt tot één 0-driehoek, een kruising tot twee 0-driehoeken.



Figuur 3.12 DeLucia en Black skeletten van een T-splitsing en een kruising

3.4 Filtertechnieken ter verbetering van de skeletteer-resultaten

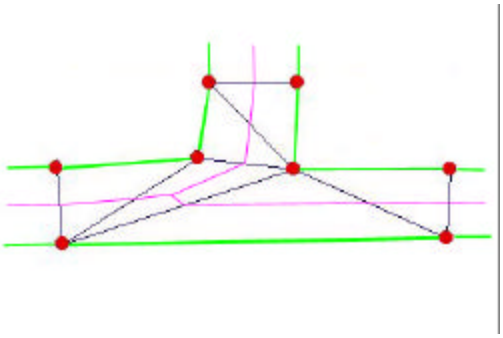
Overigens is het niet zo dat de collapse-operator alleen voor visuele doeleinden wordt ingezet. In Uitermark et al. (1999) wordt beschreven hoe uit de (vlakgerichte) Top 10 Vector en de Grootschalige Basiskaart Nederland (GBKN) met behulp van skelettering een wegennetwerk wordt afgeleid. Doel hiervan is in een later stadium te komen tot (semi-) automatische mutatiepropagatie tussen beide bestanden. In het licht van dit afstudeeronderzoek is het beschreven onderzoek met name interessant doordat de standaard methoden voor skelettering niet direct tot een bevredigend resultaat leiden. Op rechte wegen ontstaan er nauwelijks problemen, maar kruisingen blijken tot complexe situaties te leiden. In de ideale situatie zou elke T-splitsing moeten resulteren in één 0-driehoek en een kruising in twee (aangrenzende) 0-driehoeken. Helaas blijkt dit in de praktijk lang niet altijd te gebeuren doordat wegen niet mooi recht zijn. Zo is in figuur 3.13 een voorbeeld van een 'valse 0-driehoek' (Uitermark et al., 1999) te zien, veroorzaakt doordat één van de punten die de weg vastleggen, niet in lijn ligt met de overige punten. Gevolg hiervan is een afwijkend skelet.



Figuur 3.13 Een valse 2-driehoek veroorzaakt uitsteeksel skelet

Er lijkt nu immers een kort doodlopend zijweggetje te bestaan. Dit kan opgelost worden door te filteren op de oppervlakte van de drie driehoeken die aan de 0-driehoek grenzen. In de figuur is goed te zien dat de driehoek die het zijweggetje veroorzaakt, kleiner is dan de overige driehoeken. Deze kleinere oppervlakte wordt veroorzaakt door het gegeven dat deze driehoek (een 'valse 0-driehoek' (Uitermark et al., 1999)) niet de volle wegbreedte beslaat. Door een minimumoppervlakte in te stellen voor de aan een 0-driehoek grenzende driehoeken kan het kleine driehoekje verwijderd worden door de 0-driehoek als 'virtuele 1-driehoek' (Uitermark et al., 1999) te beschouwen.

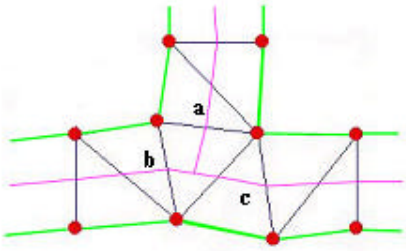
Figuur 3.14 illustreert een ander probleem dat tot een ongewenst skelet leidt: de 'verschoven 0-driehoek' (Uitermark et al., 1999). Doordat de driehoek niet samenvalt met de daadwerkelijke T-splitsing, vertoont het skelet een bocht die niet overeenkomt met de werkelijkheid.



Figuur 3.14 Een verschoven 0-driehoek beïnvloedt de vorm van het skelet

Dit probleem kan ondervangen worden door voor de triangulatie meer punten te gebruiken om de grens van de weg te beschrijven. Uitermark stelt een filtering voor op basis van de

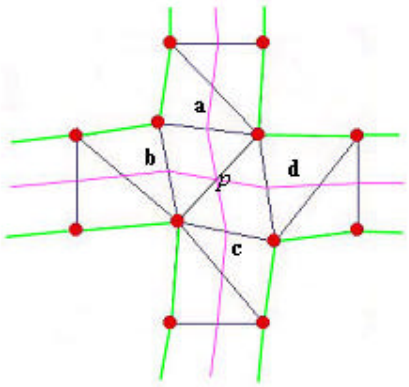
afstand tussen twee punten op een grens. Als deze afstand groter is dan de gemiddelde wegbreedte (15 meter), worden tussenpunten ingevoegd met een tussenafstand van 15 meter. Het toevoegen van tussenpunten heeft echter wel een nadelig effect op de benodigde rekentijd. Als oplossing wordt in het artikel van Uitermark het uitvoeren van een lijngeneralisatie op de wegkanten voorafgaand aan het toevoegen van de tussenpunten genoemd, met als gunstig neveneffect dat hierdoor ook een deel van de valse 2-driehoeken verdwijnt. Eventueel zou op het skelet nogmaals lijngeneralisatie toegepast kunnen worden.



Figuur 3.15 Fraaier skelet door toepassen van Jones' regels voor 0-driehoeken

Een skelet kan nog verder worden verfraaid door andere regels toe te passen voor het skeletteren van 0-driehoeken dan DeLucia en Black. Hun regels zorgen voor een soort puntje in een 0-driehoek. In figuur 3.15 is de skelettering te zien volgens de regels van Jones et al. (1995). Door figuur 3.12 en deze figuur te vergelijken is zichtbaar dat Jones' methode een fraaier resultaat geeft. De wijze waarop de 0-driehoek geskeletteerd wordt, is afhankelijk van de skeletsegmenten in de aangrenzende driehoeken. Voor lijnstukken a, b en c wordt gekeken welke twee lijnstukken het meest in dezelfde richting liggen. In de figuur liggen lijnstukken b en c ongeveer in elkaars verlengde en staat a daar min of meer haaks op. Volgens de methode Jones et al. worden nu de twee lijnstukken die het meest in dezelfde richting liggen met elkaar verbonden (verbinding tussen b en c). Van dit lijnstuk wordt het middelpunt berekend; dit middelpunt wordt verbonden met lijnstuk a.

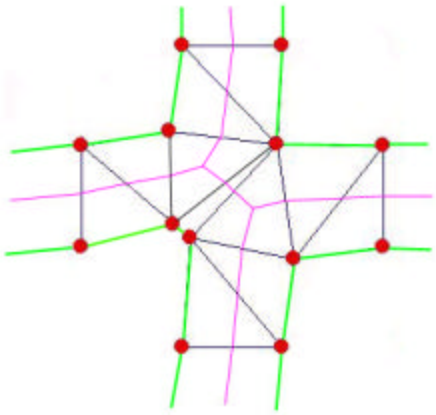
De beschrijving van de regels voor het skeletteren van 0-driehoeken in het artikel van Jones et al. gaat voorbij aan de mogelijkheid dat twee 0-driehoeken aan elkaar grenzen, waardoor niet alle drie de richtingen bepaald kunnen worden. Dit doet zich voor op kruisingen, zie nogmaals figuur 3.12. In deze figuur is te zien dat de regels van DeLucia en Black niet echt een mooie kruising opleveren. Doorredenerend op het uitgangspunt van Jones et al., namelijk dat de skeletsegmenten met gelijke richting met elkaar verbonden dienen te worden, kan een constructie als in figuur 3.16 worden ontworpen.



Figuur 3.16 Mooier skelet door het toepassen van op Jones gebaseerde rekenregels

Dit kan gerealiseerd worden door te detecteren dat twee 0-driehoeken aan elkaar grenzen, om vervolgens de vier aangrenzende lijnsegmenten a t/m d qua richting met elkaar te vergelijken. Dit leidt tot twee keer twee segmenten (a-c en b-d) die met elkaar verbonden worden. Het berekenen en vergelijken van vier richtingen kost echter rekencapaciteit en kan handiger worden opgelost. Geometrisch gezien komt deze constructie namelijk overeen met het resultaat van een eenvoudigere rekenregel. Het snijpunt p van de skeletsegmenten is immers het middelpunt van de grens die beide 0-driehoeken gezamenlijk hebben. Nu kunnen de regels van Jones et al. voor 0-driehoeken iets aangepast worden:

- als de 0-driehoek alleen aan 1- of 2-driehoeken grenst, worden de regels van Jones et al. gebruikt;
- als de 0-driehoek aan een andere 0-driehoek grenst, wordt het middelpunt van deze gezamenlijke grens berekend. Dit middelpunt wordt verbonden met de skeletsegmenten in de andere twee aangrenzende driehoeken.



Figuur 3.17 Probleem bij detectie kruispunt: de twee 0-driehoeken grenzen niet aan elkaar

In figuur 3.17 is een kruising afgebeeld waar door de vorm van de wegen de twee 0-driehoeken niet aan elkaar grenzen. Dergelijke situaties kunnen worden voorkomen door het uitvoeren van lijngeneralisatie (Uitermark et al., 1999). Daarnaast is het mogelijk om voor 0-driehoeken die niet aan elkaar grenzen, maar die wel binnen een bepaalde afstand van elkaar liggen, alternatieve regels toe te passen. Zo kunnen de drie skeletsegmenten in de driehoek bijvoorbeeld vervangen worden door twee andere segmenten, door het middelpunt van de edge die bijna aan de andere 0-driehoek grenst, direct te verbinden met de middelpunten van de andere twee edges.

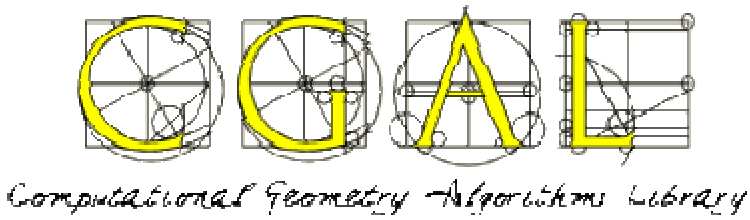
3.5 Conclusie

Gezocht is naar een methode om de grenzen van de postcodegebieden zoveel mogelijk te laten samenvallen met infrastructuur. Region growing ligt voor de hand, maar is moeilijk toepasbaar in een vectoromgeving. Daarnaast kan region growing er voor zorgen dat percelen zonder postcode opgesplitst en aan verschillende postcodegebieden worden toegedeeld, waardoor grenzen ontstaan die niet in de werkelijkheid voorkomen. Daarom is gekozen voor een aanpak in twee stappen: eerst wordt een infrastructuurperceel gesplitst, daarna worden de delen afzonderlijk toegedeeld aan postcodegebieden. Het splitsen gebeurt op hartlijnen, die bepaald worden door een collapse-operatie. De gangbare methode hiervoor in het vectordomein is skeletteren. Van de twee behandelde skeletteermethodes is de methode DeLucia en Black het meest geschikt voor toepassing in het algoritme.

4 De Computational Geometry Algorithms Library

4.1 Inleiding

Dit hoofdstuk gaat in op de Computational Geometry Algorithms Library (CGAL). Deze algoritmebibliotheek is gebruikt om het in dit onderzoek ontworpen algoritme te implementeren. In paragraaf 4.2 worden de achtergronden van deze bibliotheek behandeld. Vervolgens worden in paragraaf 4.3 de twee in het onderzoek meest gebruikte datatypen behandeld; de Planar Map en de Constrained Delaunay triangulatie. Het hoofdstuk sluit af met een samenvatting en conclusies in paragraaf 4.4.



4.2 Achtergrond van de Computational Geometry Algorithms Library

In deze paragraaf zal eerst worden ingegaan op de ontstaansgeschiedenis van CGAL. Vervolgens wordt de keuze om CGAL in C++ te ontwikkelen toegelicht. De paragraaf sluit af met een beschrijving van de structuur van CGAL.

4.2.1 Ontstaansgeschiedenis CGAL

De Computational Geometry Algorithms Library (CGAL) is een in C++ geschreven bibliotheek die veel datatypen en algoritmes bevat op het gebied van de Computational Geometry. Computational Geometry kan gedefinieerd worden als de “systematische studie van algoritmes en datastructuren voor geometrische objecten, met de nadruk op exacte algoritmes die asymptotisch snel zijn” (de Berg et al., 1997). Toepassingen zijn te vinden in tal van velden binnen de informatica, waaronder computer graphics en virtual reality, robotica, CAD/CAM en geografische informatiesystemen. In de literatuur zijn meerdere doelen van CGAL geformuleerd, waaronder “het beschikbaar maken van het geheel aan geometrische algoritmes uit de computational geometry voor industriële toepassingen” (Fabri et al., 1998) en geven van “eenvoudige toegang tot bruikbare, betrouwbare geometrische algoritmes” (http://www.cgal.org). Om deze doelen te realiseren hebben de ontwerpers bij het ontwerpen van CGAL vijf uitgangspunten gehanteerd: CGAL moest flexibel, correct, robuust, eenvoudig in gebruik en efficiënt worden.

CGAL wordt ontwikkeld door onderzoekers van acht universiteiten, namelijk de Universiteit van Utrecht, ETH Zürich, Freies Universität Berlin, Martin-Luther-Universität Halle-Wittenberg, INRIA Sophia-Antipolis, Max-Plack-Institut für Informatik, RISC Linz en de Tel Aviv University. Deze universiteiten waren al actief op het gebied van de computational geometry, wat onder andere resulteerde in een aantal voorlopers van CGAL: XYZ-library (ETH Zürich), PlaGeo/SpaGeo (Universiteit van Utrecht), C++GAL (INRIA Sophia-Antipolis) en LEDA (Max-Plack-Institut für Informatik). Delen van deze bibliotheken worden gebruikt binnen CGAL. De ontwikkeling van CGAL begon in januari 1995. In juli

1998 kwam de eerste versie van CGAL beschikbaar (versie 1.1), de meest recente versie (2.4) is sinds mei 2002 verkrijgbaar. De nieuwste versie wordt in de lo op van 2003 verwacht.

Ongeveer tegelijk met de aanvang van de ontwikkeling van CGAL, startte in de Verenigde Staten een vergelijkbaar project bij het Center for Geometric Computing (Fabri et al., 1998). In dit centrum werken onderzoeksgroepen van Brown University, Duke University en John Hopkins University samen aan een geometrische bibliotheek in Java met de naam GeomLib. Ten tijde van dit schrijven (voorjaar 2003) is de GeomLib-website “still under construction” (http://www.cs.jhu.edu/~cduncan/old/geomLib.html) en lijkt het project stil te liggen.

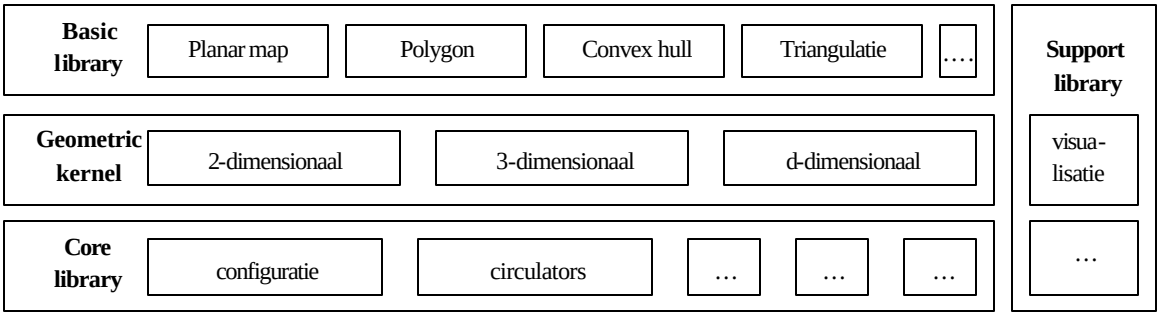
4.2.2 Keuze voor C++

Er zijn twee belangrijke redenen om CGAL als C++-bibliotheek te ontwikkelen (Fabri et al. 1998). De eerste reden is de wijde acceptatie van C++. C++ wordt zowel in academische als industriële omgeving veel gebruikt; en specifieker ook in sommige oudere bibliotheken als LEDA. Verder kan C++ eenvoudig gekoppeld worden aan bestaande C en FORTRAN code. Java werd ook gezien als breed geaccepteerde taal, maar werd als te langzaam beoordeeld.

De tweede reden is een meer inhoudelijke. C++ biedt de mogelijkheid om generiek te programmeren door gebruik te maken van templates. Templates (letterlijk sjabloon of model) maken het mogelijk om functies en klasses te definiëren zonder dat alle datatypes exact hoeven worden vastgelegd. C++ onderscheid twee typen templates, namelijk functie templates en klasse templates. In een functie template kan een functie geïmplementeerd worden die dezelfde bewerkingen uitvoert op argumenten van steeds andere types. Een eenvoudig voorbeeld is een functie die het gemiddelde van twee getallen berekent, waarbij van te voren niet vastgelegd hoeft te worden of de argumenten twee integers of twee doubles zijn. Voor klasse templating geldt hetzelfde; het is mogelijk om klasses zo te definiëren dat ze met argumenten van verschillende types kunnen werken. Als voorbeeld kan gedacht worden aan de klasse die een 2-dimensionaal punt definieert: het is mogelijk om met deze klasse zowel het punt (1,3) (twee integers) als het punt (36.692, 6.9945)(twee doubles) te creëren. Dit gebruik van templates maakt het mogelijk om CGAL flexibel en efficiënt te bouwen; twee van de vijf doelen van de ontwerpers.

4.2.3 Structuur CGAL

CGAL is opgebouwd uit drie lagen, namelijk de core library, de geometric kernel en de basic library (zie figuur 4.1). De geometric kernel gebruikt functionaliteit uit de core library, de basic library werkt weer met objecten en functies uit de geometric kernel en de core library. De support library staat eigenlijk los van deze drie lagen van CGAL en regelt onder andere de visualisatie. Het gaat hierbij vooral om export naar bestaande softwarepakketten (GeomView, LEDA) en dataformaten (VRML en Postscript). Verder regelt de support library het gebruik van number types uit andere bibliotheken, bijvoorbeeld vanuit LEDA.



Figuur 4.1 Schematische weergave van de structuur van CGAL (bron: Fabri et al., 1998)

De core library bevat een aantal ondersteunende componenten die de geometric kernel en de basic library nodig hebben om goed te kunnen functioneren. Zo is onder andere de ondersteuning voor diverse C++-compilers hierin geregeld, maar ook bijvoorbeeld circulators en random number generators. De echte geometrische functionaliteit bevindt zich in de twee hogere lagen. De geometric kernel bevat de geometrische primitieven zoals punten, lijnen, driehoeken en tetrahedra. Daarnaast bevat het simpele geometrische objecten (bijv. vectoren) en een aantal bewerkingen op deze objecten, waaronder affiene transformaties, afstandsberekeningen en intersecties. Deze objecten en bewerkingen zijn geïmplementeerd in 2D, 3D en in d-D. Hierdoor is de geometric kernel op te delen in drie aparte onderdelen, namelijk het 2-dimensionale deel, het 3-dimensionale deel en het d-dimensionale deel. Belangrijkste reden voor de speciale aandacht voor 2- en 3-dimensionale bewerkingen is de hoeveelheid onderzoek die verricht is naar specifieke 2- en 3-dimensionale bewerkingen, aangezien deze specifieke gevallen veel realistische toepassingen hebben. Dit onderzoek heeft ertoe geleidt dat er voor deze specifieke gevallen efficiëntere algoritmen bestaand dan voor de generieke d-D situatie.

Boven op de geometric kernel functioneert de basic library. Deze basic library bevat complexere geometrische objecten en datastructuren, waaronder polygonen, planaire partities en (Delaunay) triangulaties. Om CGAL zo flexibel mogelijk te maken, is ervoor gekozen om de verschillende componenten zo onafhankelijk mogelijk te maken. Niet alleen onderling zijn deze componenten onafhankelijk, ze zijn ook zo geprogrammeerd dat ze onafhankelijk zijn van een bepaald kernel. Dit betekent dat de componenten zowel met de 2-, 3- als d-dimensionale kernel kunnen werken. Dit is gedaan door alle bewerkingen zodanig te programmeren dat de geometrische primitieven en de bewerkingen daarop gescheiden zijn van de algemenere bewerkingen. Deze bewerkingen zijn hiermee onafhankelijk geworden van het geometrische kernel. De informatie over welke primitieven gebruikt worden (de parameters), is vastgelegd in de *traits*(kenmerk) klasse.

De geometric kernel en de basic library zijn zo ontworpen dat het mogelijk is om de klasses zelf uit te breiden met extra attributen. Hierdoor is het bijvoorbeeld mogelijk om een kleur aan een vlak te koppelen.

```
class Face_with_color : public CGAL::Pm_face_base
{
   long face_color;
public:
   Face_with_color() : CGAL::Pm_face_base(), face_color(0) {}
   long get_color() {return face_color; }
   void set_color(long fcol) { face_color = fcol; }
};
```

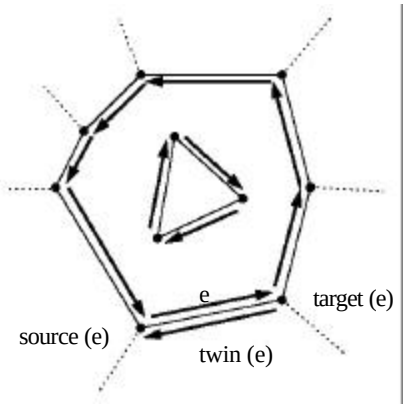
In bovenstaand voorbeeld wordt een klasse gedefinieerd op basis van de pm_face_base klasse met de naam Face_with_color. Naast de gebruikelijke attributen, die middels overerving ook aan de nieuwe klasse worden gekoppeld, wordt nu een kleur aan het vlak toegevoegd met als defaultwaarde 0 (zwart). Vervolgens worden twee functies gedefinieerd die aangeven hoe het attribuut kleur respectievelijk kan worden opgevraagd en gewijzigd. Op deze wijze kan de gebruiker CGAL heel flexibel aanpassen aan zijn concrete toepassing.

4.3 Meest gebruikte datastructuren

Tijdens het onderzoek is uitgebreid gebruik gemaakt van twee datastructuren, namelijk de Planar Map en de Constrained Delaunay triangulatie. In deze paragraaf worden deze datastructuren beschreven en wordt dieper ingegaan op het creëren en bevragen hiervan.

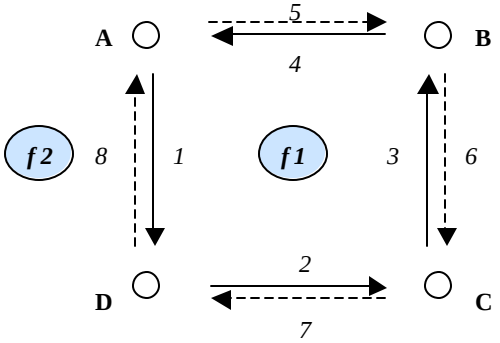
4.3.1 Planar Map

De Planar Map klasse is afgeleidt uit de Topological Map klasse. Een Topological Map is gedefinieerd als een "graaf die bestaat uit vertices V, edges E, faces F en hun onderlinge relaties" (CGAL Basic Library Manual, 2002). Elke edge in deze Topological Map bestaat uit twee halfedges met tegengestelde oriëntering. Elke face is vastgelegd als een geordende lijst met halfedges die de binnen- en buitengrens vormen. Een belangrijk begrip in dit verband is de Connected Component of the Boundary (CCB) ofwel ring. Dit is een geordende lijst met halfedges die samen één van de grenzen vormen. Eén ring is de buitengrens (outer CCB). De overige ringen (inner CCB's) representeren elk een gat (hole) in het vlak. Een face kan nul, één of meerdere gaten bevatten. De halfedges zijn zodanig georiënteerd dat het bijbehorende vlak altijd links van de halfedge ligt. Hierdoor zijn de halfedges op de buitengrens tegen de klok in georiënteerd en de halfedges op de binnengrenzen juist met de klok mee. In figuur 4.2 is dit geïllustreerd.



Figuur 4.2 Ordening halfedges op binnen- en buitengrenzen (bron: CGAL Basic Library Manual, 2002)

Een Topological Map wordt opgeslagen door alle CCB's in een DCEL-representatie op te slaan. Deze Doubly Connected Edge List is een lijst van halfedges, aangevuld met informatie over buurrelaties. Zo worden voor elke halfedge pointers naar de twin halfedge, de volgende halfedge, het bij de halfedge horende face en de target vertex opgeslagen. Hiervoor is in figuur 4.3 een klein voorbeeld uitgewerkt met een vlak f1 dat aan alle kanten wordt omsloten door vlak f2. Voor elke face wordt een pointer opgeslagen naar de outer CCB van het vlak en een iterator over de inner CCB's. Voor elke vertex wordt een pointer opgeslagen naar één van de halfedges die die vertex als target heeft.



Edge	*Twin*	*Next*	*Target*	*Face*
1	8	2	D	f1
2	7	3	C	f1
3	6	4	B	f1
4	5	1	A	f1
5	4	6	B	f2
6	3	7	C	f2
7	2	8	D	f2
8	1	5	A	f2

Figuur 4.3 DCEL representatie van de halfedges van vlakken f1 en f2

De Planar Map is een "geometrische inbedding van een Topological Map in het Euclidische vlak, zodanig dat elke edge uit de Topological Map wordt ingebed als een x-monotone curve en elke vertex als een punt" (CGAL Basic Library Manual, 2002). Een curve is x-monotoon als deze curve elke verticale lijn hoogstens één keer snijdt of als de curve een verticale lijn is. Geen enkele edge mag een andere edge raken of snijden, behalve op begin- en eindpunt. Door deze definitie komt de Planar Map overeen met een planaire partitie. Een planaire partitie is een onderverdeling in vlakken, zodanig dat de vlakken elkaar niet overlappen en er geen gaten tussen de vlakken bestaan. De perceelsgrenzen zoals ze in dit onderzoek gebruikt worden, zijn opgeslagen als polylines (te zien als meerdere curves) en zijn niet x-monotoon.

De Planar Map wordt opgebouwd door het invoegen van x-monotone curves. Op het eerste gezicht lijkt het wellicht vreemd dat het opbouwen van een planaire partitie gebeurt door het invoegen van curves en niet door het invoegen van vlakken. Dit is echter het gevolg van de keuze om de Planar Map te zien als geometrische inbedding van een Topological Map, die weer als graaf is gedefinieerd. Het opbouwen van de Planar Map komt dus neer op het opbouwen van een graaf. Dit opbouwen kan met de standaard functie die alleen de curve als parameter heeft, maar ook met een aantal geavanceerde functies die voorkennis vereisen van de relatie tussen de in te voegen curve en de structuur van de Planar Map.

```
Planar_Map pm;
pm.insert(X_curve_2 cv);
pm.insert_in_face_interior(X_curve_2 cv, Face_handle f);
pm.insert_from_vertex(X_curve_2 cv, Vertex_handle v);
pm.insert_at_vertices(X_curve_2 cv, Vertex_handle v1, Vertex_handle v2);
```

Voordeel van de geavanceerde invoegfuncties is dat ze sneller werken, doordat ze geen point location query voor de gegeven curve hoeven uit te voeren. Deze point location queries zijn ook beschikbaar als afzonderlijke functies. De standaard geïmplementeerde functie is de incremental randomized trapezoidal map. Het belangrijkste voordeel van deze strategie is de snelheid, het nadeel is echter dat voortdurend een zoekstructuur moet worden bijgehouden om deze snelheid mogelijk te maken. Naast deze standaard strategie zijn ook de naive_point_location strategie en de walk-along-a-line strategie beschikbaar.

```
pm.locate(Point_2 p, Locate_type & lt);
```

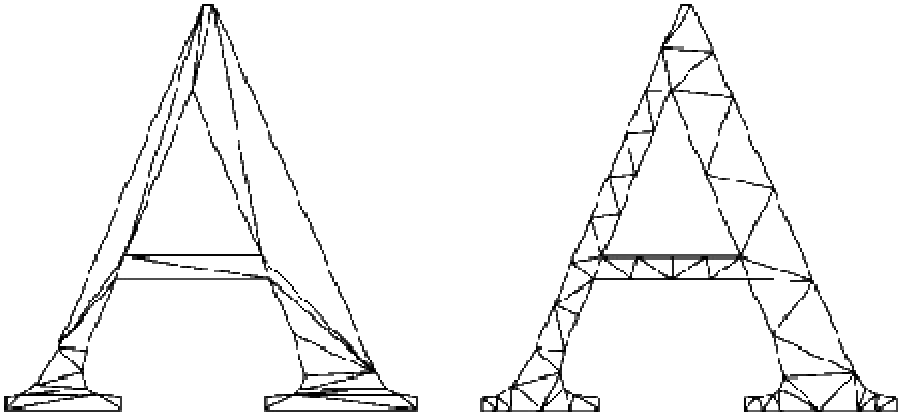
Het Locate_type geeft aan of punt p overeenkomt met een vertex, edge, face, unbounded vertex, unbounded edge of unbounded face. De functie levert een halfedge op, die afhankelijk van het in het locate_type vastgelegde zoekresultaat verwijst naar het resultaat van de point location operatie. Als het zoekresultaat een vertex is, zal de functie een halfedge opleveren die die vertex als eindpunt heeft. Indien het punt p op een edge ligt, levert de functie deze edge. Als punt p in een vlak ligt, zal de halfedge op de grens van dit vlak liggen.

De toegankelijkheid van de datastructuur is vooral geregeld in een aantal iterators en circulators. Zo zijn er iterators over alle vertices, halfedges, edges en faces van de Planar Map. Daarnaast zijn er circulators over alle edges die een bepaalde vertex als eindpunt hebben en over alle halfedges in een CCB. Om alle CCB's af te lopen is er een iterator beschikbaar die alle inner CCB's afloopt en voor elke CCB een halfedge circulator oplevert waarmee de hele CCB kan worden afgelopen.

4.3.2 Constrained Delaunay triangulatie

De constrained Delaunay triangulatie is één van de binnen CGAL beschikbare triangulaties, naast de standaard triangulatie, de Delaunay triangulatie en de constrained triangulatie. In tegenstelling tot de Delaunay triangulatie is de constrained Delaunay triangulatie geen

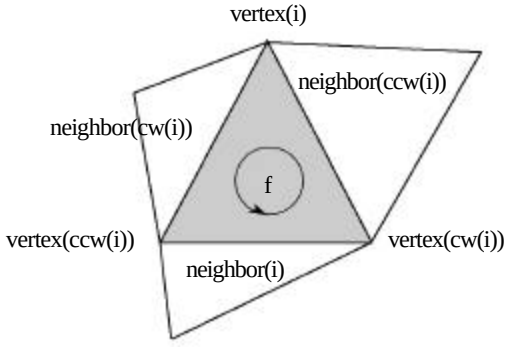
zuivere Delaunay triangulatie. Door de aanwezigheid van constrained edges geldt de regel niet altijd dat binnen de omschrijvende cirkel van een driehoek geen andere vertices liggen. Het is echter wel mogelijk om een constrained Delaunay triangulatie zo aan te passen dat deze weer wel voldoet aan de regel dat binnen de omschrijvende cirkel van een driehoek geen andere vertices liggen. Hiertoe worden extra vertices toegevoegd op de constrained edges totdat de triangulatie weer aan de Delaunay regels voldoet. Een dergelijke aangepaste triangulatie wordt een conforming Delaunay triangulatie genoemd. Dit is geïllustreerd in figuur 4.4. In deze figuur is de hoofdletter A getrianguleerd. De buitengrenzen van de letter zijn als constrained edges ingevoerd. Duidelijk zichtbaar is dat op de lange constrained edges extra vertices worden toegevoegd om zo te voldoen aan het principe van de lege omschrijvende cirkel. Binnen CGAL is overigens alleen de constrained Delaunay triangulatie geïmplementeerd.



Figuur 4.4 Een constrained Delaunay triangulatie (links) en een conforming Delaunay triangulatie (bron: http://www-2.cs.cmu.edu)

```
Constrained_Delaunay_triangulation cdt;
cdt.insert(Point p, Face_handle f);
cdt.insert(Point a, Point b);
```

In bovenstaand voorbeeld wordt een constrained Delaunay triangulatie gemaakt. Vervolgens wordt punt p ingevoegd, waarbij optioneel kan worden aangegeven in welk vlak f punt p valt. In de laatste regel wordt een constrained edge ingevoerd van punt a naar b. Alhoewel in deze triangulatie de constrained edges een belangrijke rol spelen, is de datastructuur toch vooral gericht op vlakken (de driehoeken) en de punten. Dit werkt door in de toegankelijkheid van de datastructuur. In figuur 4.5 is geïllustreerd welke bevragingen op de datastructuur mogelijk zijn.



Figuur 4.5 De vertices en buurfaces van face f (bron: CGAL Basic Library Manual, 2002)

Face f heeft drie vertices, intern genummerd 0, 1 en 2. Deze vertices zijn tegen de klok in genummerd. Doordat deze volgorde vastligt, is het ook mogelijk de vertices met de klok mee of tegen de klok in af te lopen. De buurfaces zijn zo gedefinieerd dat neighbor(i) het vlak is dat tegenover vertex(i) ligt. In onderstaand voorbeeld is te zien hoe deze vlakken en vertices kunnen worden opgevraagd. In de laatste regel wordt een edge gedefinieerd, zodanig dat edge (f,i) de edge is die vlak f en vlak f.neighbor(i) gemeen hebben.

```
Constrained_Delaunay_triangulation::Face_handle f;
f.vertex(i);
f.neighbor(i);
typedef pair <Face_handle f, int> Edge;
```

Naast de hierboven beschreven functies is de datastructuur ook toegankelijk via een aantal iterators en circulators. Zo zijn er iterators over alle faces, edges en vertices. Verder zijn er circulators over alle faces om een bepaalde vertex, over alle edges die een vertex als eindpunt hebben en over alle vertices die samen met een bepaalde vertex een edge opspannen.

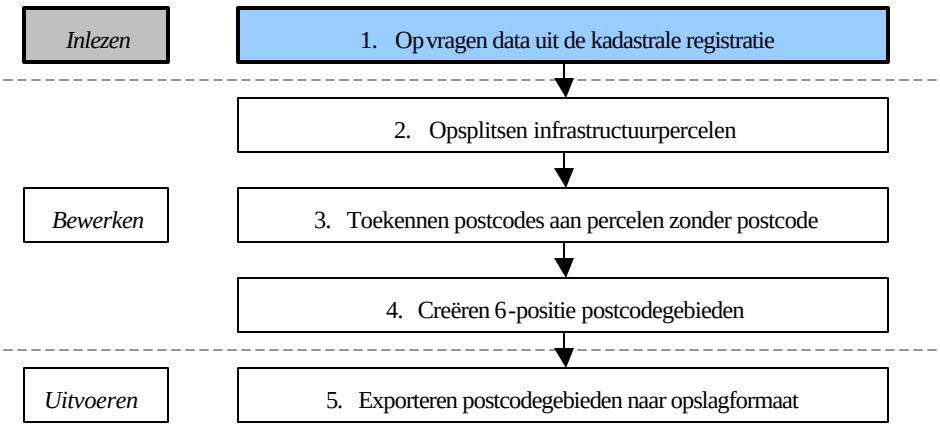
4.4 Conclusie

In dit hoofdstuk is de Computational Geometry Algorithms Library geïntroduceerd. Deze C++-bibliotheek implementeert een aantal veel gebruikte datastructuren en geometrische objecten. CGAL is flexibel geprogrammeerd en kan door de gebruiker eenvoudig worden aangepast aan zijn specifieke wensen. De datastructuren zijn goed toegankelijk, met name door de vele beschikbare iterators en circulators. Voor een beschrijving van de ervaringen van de auteur met CGAL wordt verwezen naar Appendix B.

5 Inlezen gegeven uit de kadastrale registratie

5.1 Inleiding

Het Kadaster bevordert op grond van de wet de rechtszekerheid in het maatschappelijk verkeer van vastgoed. Hiertoe worden notariële akten geregistreerd in de Openbare registers. De twee meest voorkomende typen zijn overdrachtsakten (waarbij rechten op een registergoed worden overgedragen) en hypotheekakten. De meest essentiële zaken uit de openbare registers worden opgenomen in de kadastrale registratie, zoals bijvoorbeeld verwijzingen naar akten in de openbare registers, eigenaar- en adresgegevens, kadastrale aanduiding, zakelijke rechten en de koopsom. Hiermee functioneert de kadastrale registratie als een index op de openbare registers. Uit deze kadastrale registratie zijn de gegevens afkomstig op basis waarvan het 6-positie postcodebestand berekend zal worden. In paragraaf 2.5 is de methode uitgewerkt die dit postcodebestand kan berekenen. De vijf stappen van dit algoritme zijn weergegeven in figuur 5.1. In dit hoofdstuk wordt ingegaan op de eerste stap van het algoritme: het opvragen van de benodigde gegevens uit de kadastrale registratie. Paragraaf 5.2 gaat over de benodigde perceelsgegevens. De grensgegevens zijn los opgeslagen en komen in paragraaf 5.3 aan bod. Vervolgens wordt in paragraaf 5.4 met deze gegevens in CGAL een Planar Map opgebouwd. Het hoofdstuk sluit af met conclusies in paragraaf 5.5



Figuur 5.1 De vijf stappen om tot een postcodebestand te komen

5.2 Opvragen perceelsgegevens

De kadastrale registratie bestaat uit twee afzonderlijke systemen, namelijk het LKI (Landmeetkundig Kartografisch Informatiesysteem) en de AKR (Automatisering Kadastrale Registratie). De benodigde geometrische gegevens zijn opgeslagen in het LKI en komen aan bod in paragraaf 5.2.1. De AKR bevat de benodigde administratieve perceelsgegevens en is onderwerp van paragraaf 5.2.2.

5.2.1 Perceelsgegevens uit het LKI

Binnen het onderzoek was de kadastrale registratie beschikbaar in een Ingres database. Deze registratie is met behulp van de kadastrale querytool verkend om zo te bepalen welke gegevens bruikbaar zijn en in welke tabellen en views deze gegevens zijn opgeslagen. De benodigde perceelsgegevens zijn opgeslagen in de view lki_parcel. Deze view geeft de huidige perceelssituatie aan, dit in tegenstelling tot de tabel xfio_parcel, waarin ook de historie is opgeslagen. Zoals in de inleiding al is opgemerkt, zijn de perceelsgrenzen niet in deze tabellen opgeslagen, maar in een aparte tabel. Hierdoor is redundante dataopslag vermeden om zo inconsistenties te voorkomen. In plaats van de grenzen zelf is in de tabel nu een verwijzing opgenomen naar één van de segmenten van de buitengrens. Doordat de grenzen zelf in een topologische structuur zijn opgeslagen, is het mogelijk om op basis van dat ene segment de gehele perceelsgrens te reconstrueren. Ook naar eventuele binnengrenzen zijn verwijzingen opgenomen.

In onderstaand SQL-statement worden de benodigde perceelsgegevens uit de tabel lki_parcel opgevraagd.

```
create table friso_output
as
select object_id, x_akr_objectnummer, l_num, line_id1, line_id2, 0 as split,
char(location) as location
from lki_parcel
where (municip = 'ZVH02')
```

Er wordt een nieuwe tabel aangemaakt (friso_output) waarin een aantal attributen voor elk perceel wordt opgeslagen, als het perceel in de kadastrale gemeente ZVH02 (testgebied Zevenhuizen Z.H., geboorteplaats van de auteur) ligt. De geselecteerde attributen zijn:

object_id
: Uniek identificatienummer, bijvoorbeeld 340070263.

x_akr_objectnummer
: Het kadastrale AKR objectnummer, bijvoorbeeld ZVH02 C 02761 G0000. Dit kadastrale objectnummer is samengesteld uit de kadastrale gemeentenaam (ZVH02), de kadastrale sectie (C), het kadastraal bladnummer (02761) en een perceelsnummer (G0000). In dit perceelsnummer geeft de G aan dat het om een geheel perceel gaat, in plaats van om een deelperceel (D) of appartement (A).

l_num
: Aantal lijnreferenties, oftewel het aantal referenties naar een grens. Bij een perceel zonder gaten erin is l_num gelijk aan 1, bij een perceel met één gat 2, enz.

line_id1
: Referentie naar een grens op de buitengrens via het unieke identificatie-nummer van de grens.

line_id2
: Referentie naar een grens op de eerste binnengrens. Dit veld is toegevoegd omdat veel percelen met een gat vaak maar één gat hebben. Als een perceel meer gaten heeft en er dus meer verwijzingen naar grenzen nodig zijn, worden deze opgeslagen in de aparte tabel xfio parcelover. Door het toevoegen van line_id2 hoeft deze tabel niet voor elk perceel met een gat te worden geraadpleegd, maar alleen bij percelen met meer dan één gat.

split

Dit is een veld dat niet voorkomt in de kadastrale registratie. Het is toegevoegd om een boolean-waarde op te slaan die aangeeft of een perceel geskeletteerd dient te worden. De waarde wordt later bepaald, maar nu geïnitialiseerd op 0 (false).

location

Coördinaat van een twee-dimensionaal punt dat in het perceel ligt. Het nut van dit punt komt in paragraaf 5.4 in aan de orde.

5.2.2 Perceelsgegevens uit de AKR

Nu de geometrische gegevens zijn ingelezen en opgeslagen in de nieuwe tabel friso_output, kan de split_waarde worden bewerkt. Deze waarde staat nu immers nog voor alle percelen op 0 (niet skeletteren), terwijl het de bedoeling is dat infrastructuurpercelen worden gesplitst. Met een drietal update-statements wordt de waarde op 1 (true) gezet voor alle percelen die op basis van hun cultuur- of bebouwingscode daarvoor in aanmerking komen. Deze cultuur- en bebouwingscode zijn opgeslagen in tabel akr_objectkul. De cultuurcode geeft een indicatie van het gebruik van het perceel. Er zijn 80 verschillende codes gedefinieerd. De codes die in de eerste twee update-statements worden gebruikt, hebben betrekking op infrastructuur. In onderstaand kader zijn deze codes gegeven.

Cultuurcodes (infrastructuur)

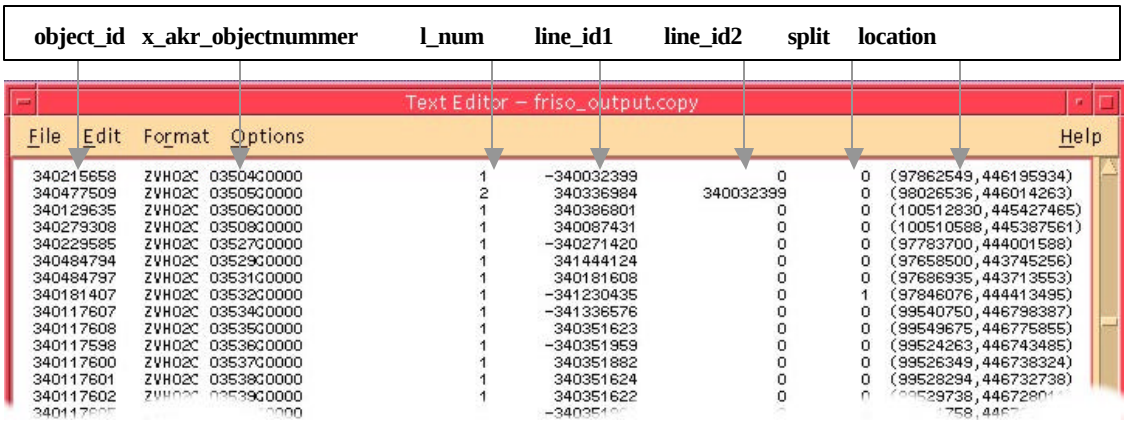
41	Verharde wegen	*81*	Buitenwater	*86*	Zeehavens
42	Semi-verharde wegen	*82*	Waterreservoirs	*87*	Waterwerken
43	Niet verharde wegen	*83*	Gracht, vaart, kanalen	*88*	Rivieren
44	Spoorwegen	*84*	Meren, plassen, vennen	*89*	Overig water
45	Tram- of metrowegen	*85*	Binnenvaarthavens		

In de praktijk bleken veel grote, complexe percelen met meerdere functies cultuurcode 0 (onbekend) te hebben. De bebouwingcode stond dan op 1 (onbebouwd). Omdat het wenselijk is om deze complexe percelen wel te kunnen splitsen, is het derde update-statement toegevoegd. Er is bewust gebruik gemaakt van deze drie statements in plaats van één OR-statement, omdat OR-statements bij gebruik van de Ingres database zeer traag bleken te werken.

```
update friso_output
set split = 1
where (object_id in (select a.object_id from akr_objectkul a where
(a.municip='ZVH02') and ((int1(a.soort_cult)>40) AND
        (int1(a.soort_cult)<46))))

update friso_output
set split = 1
where (object_id in (select a.object_id from akr_objectkul a where
(a.municip='ZVH02') and ((int1(a.soort_cult)>80) AND
        (int1(a.soort_cult)<90))))

update friso_output
set split = 1
where (object_id in (select a.object_id from akr_objectkul a where
(a.municip='ZVH02') and ((int1(a.soort_cult)=0) AND
        (int1(a.beb_code)=1))))
```

De zo gecreëerde tabel is vervolgens geëxporteerd naar een ASCII-file.

```
copy friso_output(
   object_id=c0tab,
   x_akr_objectnummer=c0tab,
   l_num=c0tab,
   line_id1=c0tab,
   line_id2=c0tab,
   split=c0tab,
   location=c0nl
) into '$PWD/friso_output.copy'
```

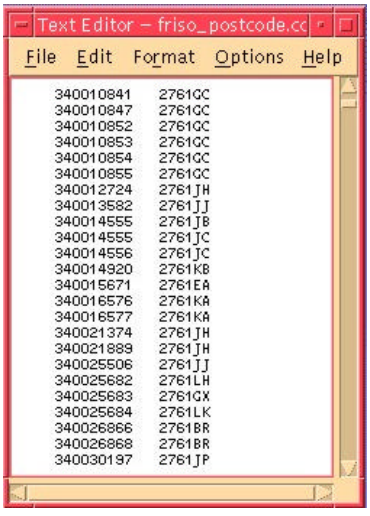


Figuur 5.2 Perceelsgegevens in het ASCII-bestand

In de bovenstaande file staan nog geen postcodes vermeld. De relatie tussen een perceel en een postcode is geen 1:1 relatie, doordat de relatie alleen via akr-objectadressen bestaat. Als aan een perceel een akr-objectadres is gekoppeld, is zo de postcode bekend. Aan één perceel kunnen echter meerdere akr-objectadressen gekoppeld zijn die verschillende postcodes hebben. Hierdoor is er sprake van een m:n relatie tussen een perceel en een postcode. Doel is nu om tot een lijst te komen waarbij per perceelnummer één of meerdere postcodes zijn opgeslagen. Als een perceel niet voorkomt in deze lijst impliceert dat automatisch dat het perceel geen (akr-objectadres met een) postcode heeft. Deze lijst is niet in één stap te maken, daarom is de bewerking in twee stappen opgedeeld. Eerst het SQL-statement:

```
create table friso_postcode
as
select distinct object_id, postcode
from akr_objectadres
where (municip = 'ZVH02') and
   (postcode != '0000')
```

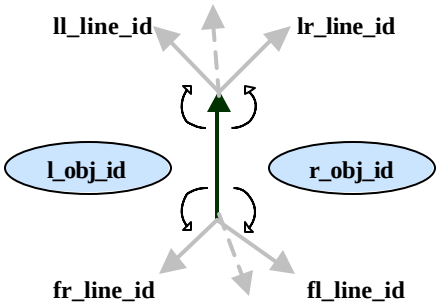
Dit resulteert in een file waarin elke in de kadastrale registratie voorkomende combinatie van perceelnummer en postcode staat. In figuur 5.3 is dit te zien. De postcode 2761 GC komt bij zes verschillende percelen voor en iets verderop blijkt aan perceel 340014555 zowel postcode 2761 JB als 2761 JC gekoppeld te zijn. De tweede bewerkingsstap komt in paragraaf 5.4 aan de orde. Deze stap heeft als resultaat dat elk perceelsnummer één keer in de lijst voorkomt, eventueel met meerdere postcodes (bijvoorbeeld de regel “340014555 2761JB 2761 JC”).



Figuur 5.3 File met perceelnummers en postcode

5.3 Opvragen grensgegevens

In de vorige paragraaf is melding gemaakt van de afzonderlijke opslag van percelen en de perceelsgrenzen. Voor het onderzoek zijn de grenzen uit tabel lki_boundary gebruikt. De grenzen zijn als polylines opgeslagen in een topologische structuur. Deze winged-edge representatie is een uitbreiding op de doubly connected edge list (DCEL). In figuur 5.4 is deze structuur weergegeven.



Figuur 5.4 De winged-edge structuur zoals gebruikt in lki_boundary

Voor elke edge zijn de perceelnummers van de percelen links en rechts opgenomen (l_obj_id en r_obj_id). De edge heeft twee vertices, de first (f) en last (l) vertex. Voor beide vertices is gekeken van welke edges die vertex nog meer deel uitmaakt. De eerste edge links en rechts van de edge (met de klok mee respectievelijk tegen de klok in) worden opgenomen. Belangrijk om op te merken is dat de kadastrale registratie geen gebruik maakt van halfedges. Om nu toch eenduidig te kunnen verwijzen van een perceel naar een grenssegment op de grens wordt eventueel naar een negatief edge_id verwezen als de oriëntering tegengesteld zou moeten zijn.

In het onderzoek wordt de volledige topologische structuur bevraagd en opgeslagen in een file. Met onderstaand SQL-statement worden alle grenzen opgevraagd die links en/of rechts aan de gewenste gemeente grenzen. Omdat na de union-operatie de grenzen die zowel links als rechts de gewenste kadastrale gemeente hebben, dubbel zijn opgeslagen, worden deze met het select distinct-commando eruit gefilterd.

```
sql kadnl << EOF
create table friso_boundary
as
(select l.* from lki_boundary l where (l_municip = 'ZVH02'))

union all

(select l.* from lki_boundary l where (r_municip = 'ZVH02'))

create table friso_boundary_distinct
as
select distinct * from friso_boundary
```

Dit resulteert na een schrijfstatement in een ASCII-file met de volgende attributen: object_id, ll_line_id, lr_line_id, fl_line_id, fr_line_id, l_obj_id, r_obj_id en geo_polyline. Geo_polyline is een string met lengte 1150 die alle coördinaten bevat (bijvoorbeeld ((99517294, 447711423), (99559195,447723559), (99580621,447726417)) voor een grens bestaand uit drie punten.

5.4 Opbouwen Planar Map

Nu alle benodigde gegevens zijn opgevraagd uit de kadastrale database, kan de data in CGAL ingelezen worden. Er is voor gekozen om de data in de in paragraaf 4.3.1 beschreven Planar Map-datastructuur op te slaan. Het algoritme dat de daadwerkelijke 6-positie postcodegebieden berekent, werkt op basis van deze Planar Map. In hoofdstuk 4 is vermeld dat het invoegen van data in een Planar Map edge-gewijs gaat en niet per vlak, wat men wellicht zou verwachten op basis van de naam. Dit komt doordat de Planar Map binnen CGAL als geometrische inbedding van een graaf wordt gezien. Voor de implementatie heeft dit als gevolg dat het niet mogelijk is een perceel met de bijbehorende geometrie en attributen ineens in te laden. Het proces van inlezen en opbouwen bestaat nu uit drie stappen: eerst worden de drie ASCII-files (één met perceelsgegevens, één met postcodes en één met grensgegevens) ingelezen, vervolgens worden de grenzen ingevoegd en als laatste worden aan de ontstane faces in de Planar Map de bijbehorende attributen gekoppeld. In dit proces wordt geen gebruik gemaakt van de topologie die in de kadastrale database is opgeslagen en die in voorgaande paragrafen is opgevraagd. Deze topologie maakt het mogelijk om bij de perceelsgegevens direct de goede grenzen te vinden. Er is voor gekozen deze topologie toch te beschrijven en te bevragen, aangezien deze gegevens wel nodig zijn als het algoritme in een andere omgeving dan CGAL wordt geïmplementeerd.

De eerste stap is het inlezen van de drie ASCII-files. Deze files worden regel voor regel ingelezen en de data wordt opgeslagen in een map. Een map is een opslagstructuur uit de Standard Template Library (STL) waarin bepaalde waarden worden opgeslagen, die ontsloten worden middels een sleutelwaarde. Behalve via de sleutel kan de map ook met iterators ontsloten worden. In de eerste map worden de perceelnummers als sleutelwaarde opgeslagen en gekoppeld aan een string met de bijbehorende postcodes. Als in de ASCII-file meerdere postcodes bij één perceelnummer horen, worden deze postcodes achter elkaar gezet in één lange string. De tweede map koppelt het perceelnummer als sleutel aan een object parcel. Dit is object van een zelfgedefinieerde klasse. Het object heeft alle attributen die in de ASCII-file met perceelgegevens opgeslagen zijn, maar wordt nu ook uitgebreid met de lange string met postcodes. Als het perceelnummer van het perceel niet in de map met postcodes voorkomt, heeft het perceel nog geen postcode en krijgt het als initialisatiewaarde een string met lengte nul “”. In de map met grenzen worden de grensnummers gekoppeld aan een object van de zelf gedefinieerde klasse boundary. Dit object bevat alle attributen zoals ze in de ASCII-file opgeslagen zijn.

De tweede stap is het invoegen van de grenzen in de Planar Map. Hiertoe worden alle geo_polylines gesplitst in segmenten. In de geo_polyline zijn de polylines opgeslagen als lijst van punten, bijvoorbeeld punten a-b-c-d. Zoals in hoofdstuk 4 is vermeldt, kan de Planar Map hier niet mee omgaan, omdat deze polylines geen x-monotone curves zijn. Daarom wordt de polyline a-b-c-d gesplitst in de x-curves a-b, b-c en c-d. Deze X-curves worden afzonderlijk in de Planar Map ingevoegd.

De derde en laatste stap is het toevoegen van de perceelsattributen aan de faces in de Planar Map. Hiervoor wordt de map met perceelsgegevens gebruikt. Via een iterator wordt elk perceel in de map afgelopen. Voor elk perceel wordt met behulp van het location-attribuut (dat een willekeurig punt binnen het perceel bevat) een point location query op de Planar Map uitgevoerd.

```
bool spl=false;
parcel& p= *(map_iter->second);
if (p.split) spl = true;
Planar_map::Locate_type lt;
PM_Face_handle f = pm.locate(ins_point,lt)->face();
f->set_id(pid);
f->set_split(spl);
```

```
if (spl == true)
{
   f->set_pc("");
   spl = false;
}
else
{
   f->set_pc(postcode);
};
```

Deze point location query levert een face_handle f op. Na een aantal (hier voor de overzichtelijkheid weggelaten) controles of deze face de gezochte face is, wordt aan de face het perceelnummer, de split-waarde en de postcode gekoppeld. Aan percelen die in een later stadium geskeletteerd en gesplitst gaan worden, wordt geen postcode gekoppeld, ook al is er één bekend. Dit oogt vreemd, maar bleek in de praktijk nodig. Grote percelen met veel wegen erin zijn in regel eigendom van de gemeente. Bij meerdere van dit soort percelen bleek een huurhuis op gemeentegrond te staan. Dit huis stond op een deel van het perceel, maar de postcode die dit akr-objectadres heeft, wordt aan het gehele perceel gekoppeld. Deze postcode is niet valide voor het gehele perceel, dit perceel gaat juist in kleinere delen worden opgesplitst om aan elk stuk afzonderlijk een betere postcode toe te kennen.

5.5 Conclusie

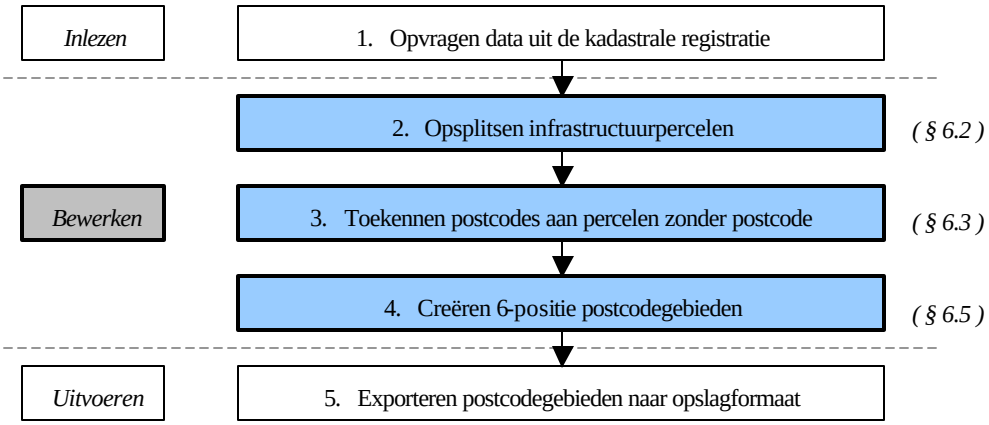
Het in dit onderzoek ontwikkelde algoritme maakt gebruik van gegevens uit de kadastrale registratie. Zowel uit het LKI als uit het AKR zijn gegevens nodig. Perceel- en grensgegevens zijn gescheiden opgeslagen, maar zijn te koppelen doordat beide in een topologische datastructuur zijn opgeslagen, de winged-edge representatie. De topologie wordt dan ook opgevraagd en geëxporteerd naar een drietal ASCII-files. Al bij het inlezen wordt bepaald welke percelen in aanmerking komen om geskeletteerd te worden. Dit wordt aan de hand van de cultuur- en bebouwingscode gedaan.

De gegevens worden in een Planar Map ingelezen. Eerst worden alle grenzen ingevoegd. Aan de faces die hierdoor ontstaan worden een aantal perceelsattributen gekoppeld, namelijk het perceelnummer, de boolean die aangeeft of het perceel geskeletteerd en gesplitst moet worden en de postcode. Bij het inlezen wordt geen gebruik gemaakt van de topologische structuur, aangezien CGAL deze zelf opbouwt.

6 Berekenen 6-positie postcodegebieden

6.1 Inleiding

In het vorige hoofdstuk zijn de benodigde gegevens uit de kadastrale registratie opgevraagd en is met die data een Planar Map opgebouwd. Deze Planar Map en de daaraan gekoppelde attributen vormen het uitgangspunt voor het berekenen van de 6-positie postcodegebieden. Het nieuwe algoritme voor het berekenen van de 6-positie postcodegebieden is in paragraaf 2.5 geïntroduceerd. In dit hoofdstuk komen de daadwerkelijke bewerkingsstappen (stap 2 t/m 4) van het algoritme aan bod. De complete C++-code is in Appendix A te vinden, waarbij de stappen dezelfde nummering hebben.



Figuur 6.1 Structuur van het ontwikkelde algoritme

In paragraaf 6.2 wordt in vier deelstappen het proces van skeletteren en opsplitsen van percelen op basis van dat skelet behandeld. Als de skelettering is afgerond, wordt aan alle (delen van) percelen zonder postcode de meest waarschijnlijke postcode toegekend, onderwerp van paragraaf 6.3. Om een beeld te krijgen van de relatieve betrouwbaarheid wordt aan alle postcodegebieden een betrouwbaarheidsindicator gekoppeld. Het berekenen van deze betrouwbaarheidsindicator komt in paragraaf 6.4 aan bod. Als aan alle (delen van) percelen de benodigde attributen zijn gekoppeld, kunnen de percelen met gelijke postcode worden samengevoegd om zo tot 6-positie postcodegebieden te komen. Deze laatste bewerkingsstap in het berekenen van de postcodegebieden wordt uitgewerkt in paragraaf 6.5. Het hoofdstuk sluit af met conclusies in paragraaf 6.6.

6.2 Skeletteren en splitsen van percelen

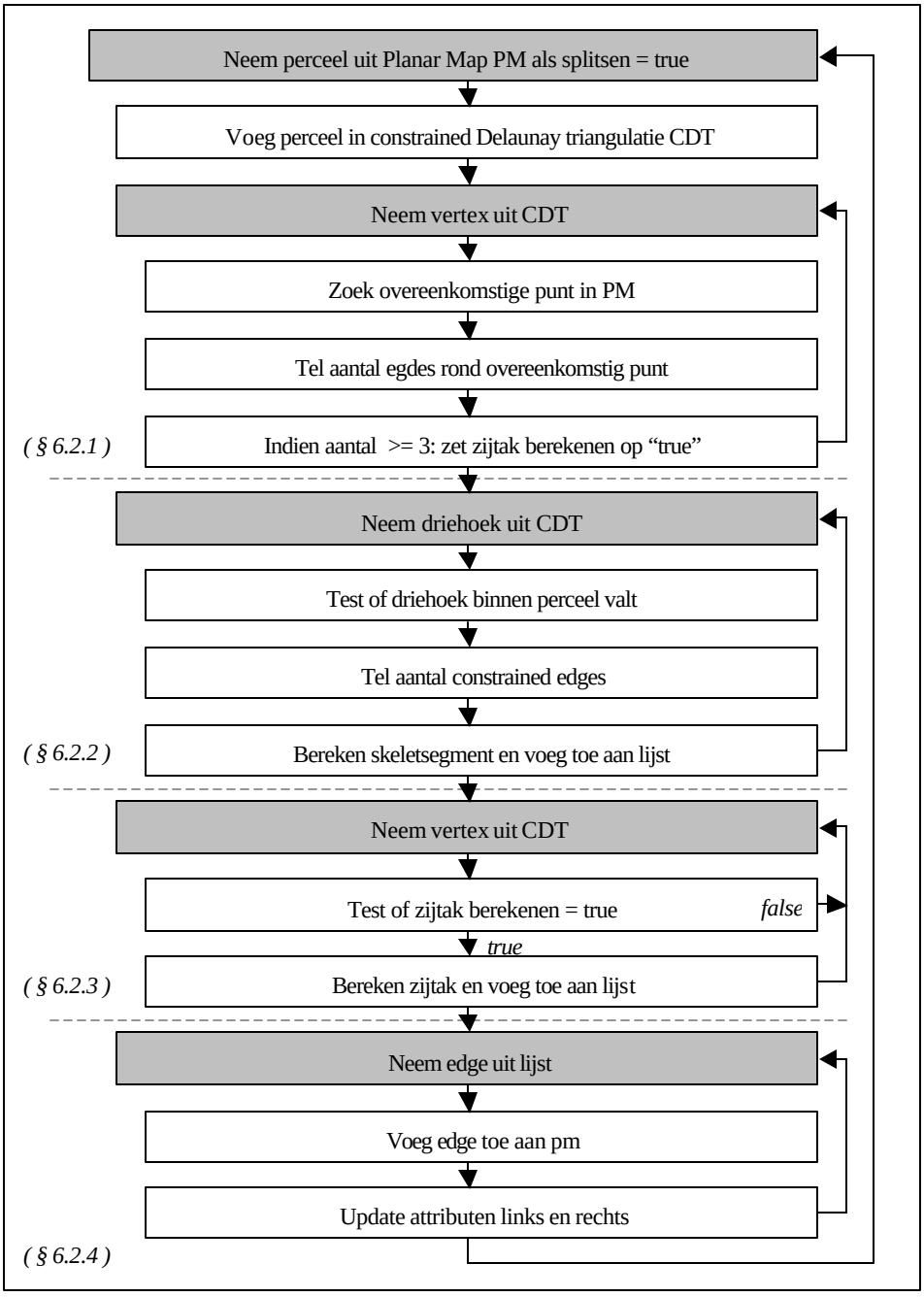
In hoofdstuk 1 is de gedachte dat skeletteren wenselijk is gemotiveerd met het idee dat huizen aan de ene kant van de straat vaak een andere postcode hebben dan de huizen aan de andere kant. De grens tussen die twee postcodegebieden loopt gevoelsmatig over de hartlijn van de weg. Door de weg te skeletteren en te splitsen, krijg je fraaiere postcodegebieden. Naast deze theoretische noodzaak is er ook een heel duidelijke praktische reden om te skeletteren.

Figuur 6.2 Een groot gemeentelijk perceel (boven, geel) wordt geskeletteerd en opgesplitst (onder)

In figuur 6.2 is boven de kadastrale situatie weergegeven. Het gemeentelijke perceel (in geel) is zeer groot, alle verkochte percelen zijn er als gaten uitgeknipt. Hierdoor grenst het gemeentelijke perceel aan percelen met bijna twintig verschillende postcodes (in pasteltinten weergegeven). Als dit perceel niet geskeletteerd en gesplitst zou worden, wordt er één postcode aan toegekend. Deze situatie zou niet overeenkomen met de werkelijkheid. Gevolg hiervan zou bijvoorbeeld kunnen zijn dat een routeplanner die gebruik maakt van het postcodebestand een route geeft die vijf straten verderop eindigt in plaats van op de gewenste bestemming. In figuur 6.2 is onder te zien hoe skeletteren en splitsen zorgt voor een realistischer beeld. In deze figuur is het gemeentelijke perceel gesplitst en is per deel de meest waarschijnlijke postcode bepaald.

Het proces om percelen te ske letteren en op te splitsen bestaat uit vier stappen. Eerst wordt het te skeletteren perceel getrianguleerd (paragraaf 6.2.1). Vervolgens worden de in hoofdstuk 3 geïntroduceerde rekenregels van DeLucia en Black toegepast op de driehoeken van de triangulatie (paragraaf 6.2.2). Op basis van het skelet dat nu berekend is, kan het perceel nog niet gesplitst worden. Hiervoor is een aantal zijtakken nodig, die aansluiten op aangrenzende

perceelsgrenzen (paragraaf 6.2.3). Uiteindelijk kan met het skelet het perceel gesplitst worden in een aantal nieuwe vlakken (paragraaf 6.2.4). Deze aanpak is in figuur 6.3 in een stroomschema uitgewerkt.



Figuur 6.3 Stroomschema bewerkingsstappen skeletteren en splitsen van infrastructuurpercelen

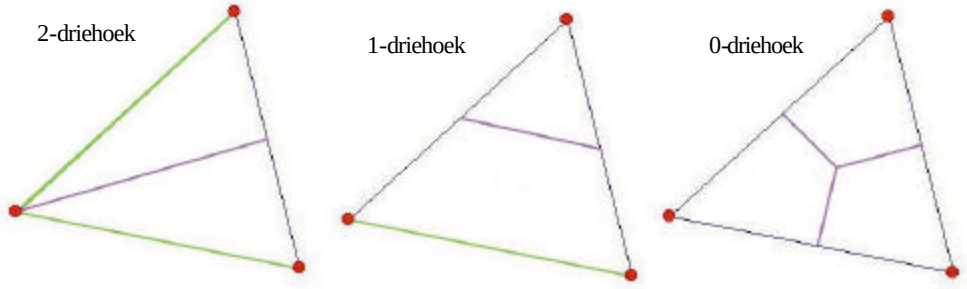
6.2.1 Trianguleren

Uitgangspunt voor het skeletteren is een triangulatie en meer specifiek een Constrained Delaunay triangulatie. In deze triangulatie worden de perceelsgrenzen als constrained edge opgeslagen. Het gehele proces van skeletteren en splitsen wordt per perceel doorlopen. In paragraaf 5.2.2 is op basis van de aan een perceel gekoppelde cultuur- en bebouwingscode

bepaald of een perceel geskeletteerd en gesplitst moet worden. Deze informatie is als attribuut aan elke face in de Planar Map gekoppeld. Met behulp van een face iterator worden alle faces (percelen) in de Planar Map afgelopen. Als het attribuut split_parcel true is, wordt het perceel geskeletteerd. Hiertoe worden alle grenzen opgevraagd. Eerst wordt de buitengrens afgelopen met een circulator en wordt elke edge zoals in paragraaf 4.3.2 als constrained edge ingevoegd. Na het invoegen van de buitengrens wordt er geïtereerd over alle gaten in het perceel. Over elke grens van een gat wordt weer gecirculeerd om zo alle edges in te voegen in de Constrained Delaunay triangulatie.

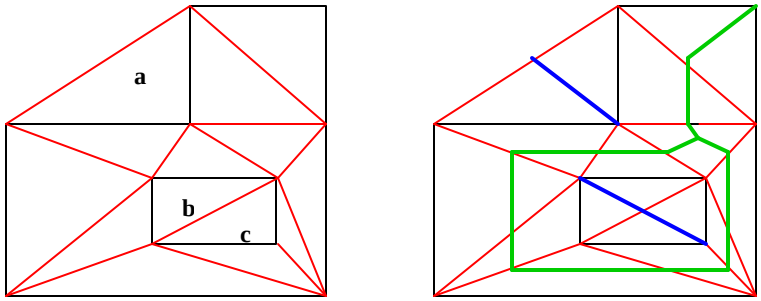
6.2.2 Skeletsegmenten berekenen

Nu het te skeletteren perceel is getrianguleerd, kunnen de skeletsegmenten berekend worden. Hiervoor worden de rekenregels van DeLucia en Black gebruikt, zoals die in hoofdstuk 3 zijn geformuleerd. In figuur 6.4 zijn deze regels nogmaals weergegeven. Het aantal constrained edges (groen) in een driehoek is bepalend voor de wijze waarop het skeletsegment wordt berekend.



Figuur 6.4 Skeletteerregels DeLucia en Black (in groen de constrained edges)

Om het skelet te berekenen worden alle driehoeken in de triangulatie afgelopen en voor elke driehoek wordt het bijbehorende stukje skelet berekend. In twee gevallen moet de driehoek echter niet gebruikt worden, dit is in figuur 6.5 geïllustreerd.



Figuur 6.5 Getrianguleerd perceel met gat (unconstrained edges rood, constrained edges zwart)

Driehoek a is een voorbeeld van het eerste geval. Het gaat hier om een driehoek die buiten het perceel ligt. Dergelijke driehoeken ontstaan doordat een Delaunay triangulatie altijd convex is. Deze driehoeken moeten niet worden meegenomen in de berekening van een skelet. De driehoeken b en c zijn voorbeelden van het tweede geval. Hierbij ligt de driehoek ook niet in het perceel, maar nu doordat er gaten in het perceel voorkomen. Deze gaten worden echter ook getrianguleerd. Ook deze driehoeken moeten niet gebruikt worden. Rechts in de figuur is het gewenste skelet ingetekend (groen) en in blauw de ongewenste skeletsegmenten.

De twee gevallen waarin geen skelet moet worden berekend, hebben gemeen dat in beide gevallen de driehoek buiten het vlak ligt. In het algoritme wordt dit gedetecteerd door in de triangulatie een punt in de driehoek te berekenen. Met dit punt wordt in de Planar Map een point-in-polygon-test uitgevoerd:

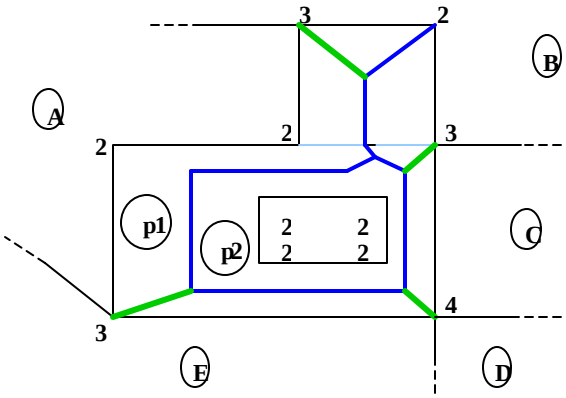
```
p0 = cdt_f_iter->vertex(0)->point();
p1 = cdt_f_iter->vertex(1)->point();
p2 = cdt_f_iter->vertex(2)->point();

p01 = midpoint (p0, p1);
inside_p = midpoint (p01, p2);
triangle_in_face = pm.is_point_in_face(inside_p, pm_f_iter);
```

De punten p0, p1 en p2 zijn de drie hoekpunten van de driehoek in de triangulatie. Punt inside_p is een punt dat altijd binnen de driehoek ligt. Uiteindelijk geeft de boolean triangle_in_face aan of de driehoek in het vlak valt. Alleen als dat het geval is, dient van deze driehoek het skelet berekend te worden. Als alle driehoeken zijn bezocht, resulteert dat in een lijst met skeletsegmenten, de groene lijnstukken in figuur 6.5.

Alhoewel de point-in-polygon-test niet heel veel rekentijd kost, is het mogelijk om deze test te vermijden om zo het algoritme sneller te maken. Het is mogelijk om de driehoeken op basis van de buurrelaties zodanig af te lopen dat alleen driehoeken bezocht worden die binnen het perceel liggen. Dit kan gebeuren door eerst een grens te zoeken die ook een constrained edge (perceelsgrens) is, daar de bijbehorende (in het perceel liggende) driehoek bij op te vragen en vervolgens steeds de buurdriehoeken op te vragen, zolang hierbij maar geen buren worden opgevraagd die aan de andere zijde van een constrained edge (perceelsgrens) liggen.

6.2.3 Zijtakken skelet berekenen

Resultaat van de vorige stap is het skelet zoals dat in figuur 6.6 in blauw is weergegeven. Dit is echter nog niet voldoende om het lichtblauwe perceel op te kunnen splitsen. Het skelet zou in dit geval het perceel in twee delen splitsen, één deel (p1) dat aan de buitengrens ligt en één deel (p2) dat aan de binnengrens ligt. Als nu echter het perceel vijf buurpercelen A t/m E heeft, is het de vraag aan welk van deze percelen het buitenste deel zou moeten worden gekoppeld. Het ligt meer voor de hand om nu het deel p1 zo op te splitsen dat de delen van p1 aan de verschillende percelen gekoppeld kunnen worden. Dit wordt gedaan door het skelet uit te breiden met een aantal zijtakken, in de figuur in groen weergegeven. Deze zijtakken vormen een verbinding tussen het skelet en een punt op de perceelsgrens. De vraag is nu hoe bepaald wordt of er een zijtak berekend moet worden.



Figuur 6.6 Perceel met skelet (blauw) en zijtakken skelet (groen)

Niet elk punt op de perceelsgrens hoeft immers verbonden te worden met het skelet. De voorwaarde die gesteld wordt om te bepalen of een punt verbonden hoort te worden met het skelet, is of het punt op een grens tussen twee buurpercelen ligt. In de figuur maakt bijvoorbeeld het punt rechtsonder deel uit van de grenzen van het blauwe perceel en de percelen C, D en E. Een dergelijk punt wordt op wel een topologische node genoemd. Het punt rechtsboven maakt geen deel uit van een grens tussen twee buurpercelen, maar alleen van de grens tussen het blauwe perceel en buurperceel B. Dit soort punten zijn vormpunten. Op basis van dit verschil wordt het punt rechtsonder wel verbonden met het skelet en het punt rechtsboven niet.

Dit idee is in een rekenregel gevat door voor elk punt in de triangulatie de corresponderende vertex in de Planar Map te zoeken. Voor deze vertex wordt een halfedge around vertex circulator gedefinieerd. Deze circulator loopt alle halfedges af die de vertex als eindpunt hebben. Door nu dit aantal edges te tellen, kan bepaald worden of een punt verbonden dient te worden met het skelet. Als het aantal edges gelijk is aan twee, is er sprake van een punt dat alleen de vorm van de grens tussen perceel en buurperceel vastlegt. Als het aantal edges drie of meer is, maakt het punt ook deel uit van een grens tussen twee buurpercelen en wordt er een zijtak van het skelet berekend. Het aantal edges dat bij een vertex hoort is in de figuur weergegeven. Merk op dat deze telling ook gebeurt voor punten op de binnengrens. Als er in een gat meerdere percelen liggen, zijn ook hier zijtakken nodig.

Dankzij de telling van het aantal edges rond een vertex in de Planar Map kan worden vastgesteld of voor dat punt een zijtak berekend moet worden. Als dit het geval is, wordt binnen de triangulatie een edge around vertex circulator gedefinieerd. Deze circuleert over alle edges die mede door de vertex worden opgespannen. Voor elke edge wordt het middelpunt berekend en gekeken of deze edge binnen het perceel valt. Een edge kan ook buiten het perceel vallen, zoals de edge helemaal linksboven in de figuur. Met de eerste edge die de circulator bezoekt die binnen het perceel ligt, wordt de zijtak bepaald door het punt te verbinden met het middelpunt van deze edge. De keuze om de eerste edge te nemen is tijdens het onderzoek gemaakt om zo snel mogelijk een werkende versie van het algoritme te kunnen realiseren. Door tijdgebrek is hiervoor geen fraaiere oplossing geïmplementeerd. Het is waarschijnlijk fraaier om van alle edges die binnen het perceel liggen de kortste te nemen, om zo de kans op uitsteeksels te verkleinen. Om het aantal zijtakken terug te brengen (en zo de bestandsomvang en rekentijd), is het ook mogelijk om alleen de zijtakken te berekenen naar topologische nodes waar de extra grens een postcodevlakgrens is.

6.2.4 Percelen splitsen op basis van het skelet

Nu het volledige skelet bekend is, kan het worden ingevoegd in de Planar Map. Dit gaat identiek aan het invoegen van de perceelsgrenzen zoals beschreven in pararaaf 5.4. Het skelet, inclusief de zijtakken, is opgeslagen als een lijst met X-curves. Deze curves worden één voor één in de Planar Map ingevoegd. Op het moment dat door het invoegen van een curve nieuwe vlakken ontstaan, krijgen deze de initialisatiewaarden. Hierdoor is niet meer vast te stellen uit welk perceel het deelgebied afkomstig is. Dit wordt opgelost door na elke ingevoegde edge de attributen opnieuw aan de vlakken links en rechts van de ingevoegde edge te koppelen.

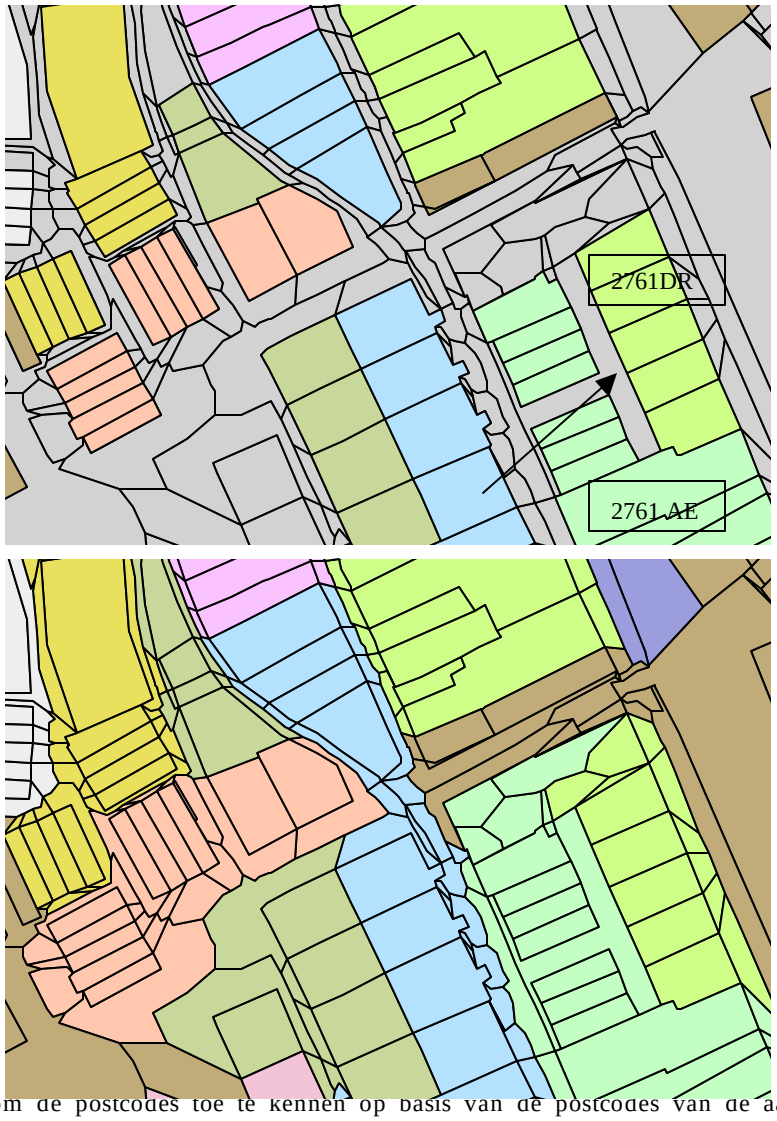
```
PM_Halfedge_handle e = pm.insert(edges[j]);
PM_Face_handle f = e->face();
PM_Face_handle ft = e->twin()->face();

f->set_id(parcel_id);
f->set_split(to_be_split);
f->set_pc("");
f->set_splitted(true);

ft->set_id(parcel_id);
ft->set_split(to_be_split);
ft->set_pc("");
```

```
ft->set_splitted(true);
```

Dit is mogelijk doordat de insert-operatie de halfedge e oplevert die overeenkomt met de X-curve. Van deze halfedge worden de face f en de bij de tegengesteld georiënteerde halfedge horende face ft opgevraagd. Aan deze twee vlakken worden de oorspronkelijke perceelsattributen gekoppeld.

6.3 Toekennen postcodes

Aan veel percelen is niet direct een postcode te koppelen, omdat er geen akr-objectadres aan het perceel is gekoppeld of omdat het perceel net is opgesplitst in kleinere delen. Om toch tot een planaire partitie van postcodegebieden te komen, zal voor elk (deel van een) perceel bepaald moeten worden welke postcode dit perceel het meest waarschijnlijk heeft. Deze meest waarschijnlijke postcode wordt aan het perceel gekoppeld en op deze wijze hebben uiteindelijk alle (delen van) percelen een postcode. In figuur 6.7 is weergegeven welk resultaat dit heeft. Boven is de oorspronkelijke situatie te zien met in grijs de percelen zonder postcode. Onder is voor elk perceel de meest waarschijnlijke postcode bepaald en toegekend.

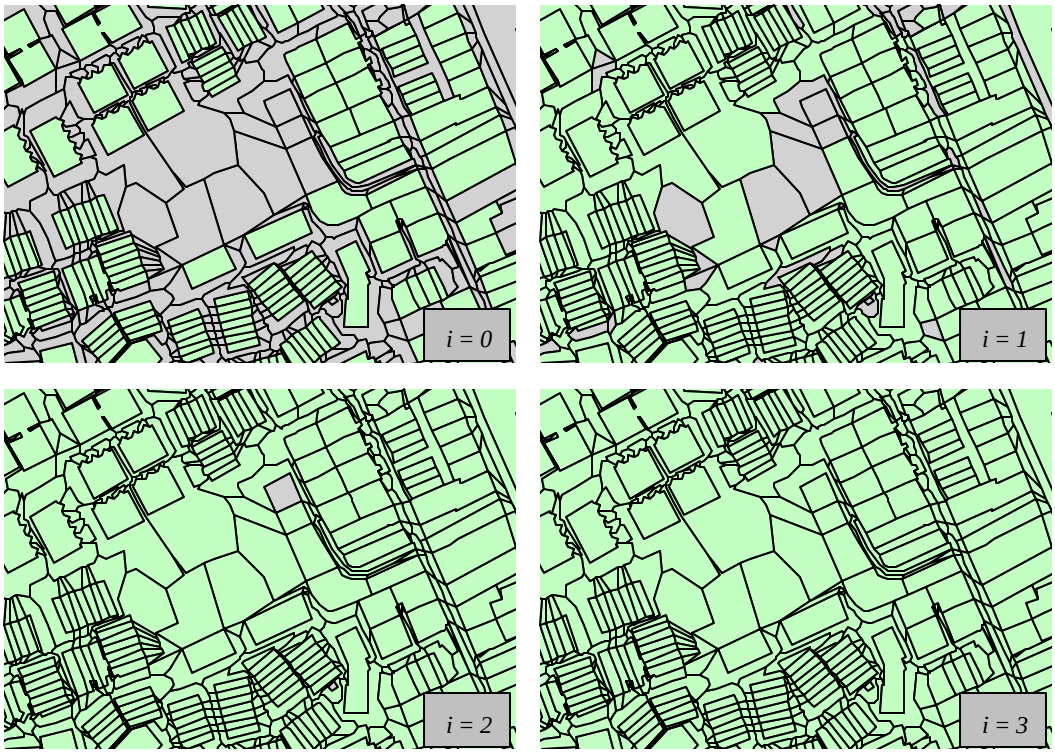


Figuur 6.7 Situatie voor (boven) en na (onder) het toekennen van de meest waarschijnlijke postcode

De keuze om de postcodes toe te kennen op basis van de postcodes van de aangrenzende percelen is gemaakt vanwege de (gelijkenis van deze methode met de resultaten van region

growing. In paragraaf 3.2 is beschreven hoe deze methode intuïtief het beste aansluit op de werkelijkheid. De vraag is nu op welke wijze bepaald kan worden wat de meest waarschijnlijke postcode voor een perceel is. Gezocht wordt naar een methode om op basis van de postcodes van de omliggende percelen te bepalen welke postcode het perceel moet krijgen. Eén mogelijkheid is het kiezen van de postcode die de meeste omliggende percelen hebben. Aangezien dit nog een redelijk grove methode is, is er voor gekozen om postcodes toe te kennen op basis van de grootst gezamenlijke grenslengte. In figuur 6.7 is dit idee weergegeven. Voor het T-vormige perceel, gemerkt met de pijl, wordt gekeken hoe lang de grenzen zijn die het perceel gezamenlijk heeft met omliggende percelen met een postcode. Het perceel grenst aan zes percelen met postcode 2761 DR (de percelen oostelijk van het perceel) en aan acht percelen met postcode 2761 AE (de percelen ten westen en zuiden). Omdat de som van de lengte van de grenzen met de percelen met postcode 2761 AE groter is dan de som van de lengte van de grenzen met de percelen met postcode 2761 DR, krijgt het perceel postcode 2761 AE toegekend.

Het kan natuurlijk ook voorkomen dat geen van de omliggende percelen een postcode heeft. In dit geval wordt geen postcode toegevoegd. Het proces van toekennen van postcodes is daarom een iteratief proces. In figuur 6.8 zijn drie iteratiestappen weergegeven.



Figuur 6.8 Iteratief postcodes toekennen. Percelen met postcode in groen, zonder postcode in grijs

Het aantal iteratiestappen dat nodig is om aan een perceel een postcode toe te kennen, komt overeen met de minimale afstand (in percelen geteld) tot een perceel mét postcode in de oorspronkelijke situatie. In het gegeven voorbeeld bestaan er ook na de vijfde iteratiestap nog percelen zonder postcode.

6.4 Berekenen betrouwbaarheidsindicator

In de vorige paragraaf is het toekennen van postcodes aan percelen zonder postcode aan de orde geweest. Dit resulteert in een Planar Map waarin aan elke face een postcode gekoppeld is. Faces met dezelfde postcode kunnen worden samengevoegd tot 6-positie postcodevlakken. Met name door het toekennen van de postcode is het de vraag hoe betrouwbaar deze postcodegebieden zijn, oftewel hoe aannemelijk het is dat de toegekende postcode de juiste is. Uitspraken over de absolute betrouwbaarheid zijn niet te geven, aangezien er geen sprake is van één correcte manier. De postcode wordt immers door TPG Post toegekend aan huisadressen en niet direct aan gebieden. Over de relatieve betrouwbaarheid kunnen wel enkele uitspraken gedaan worden. In het algemeen mag men er van uitgaan dat de in de kadastrale registratie vermelde postcodes correct zijn. Als een 6-positie postcodegebied vooral percelen bevat die hun postcode rechtstreeks uit de kadastrale registratie ontlenen, is de relatieve betrouwbaarheid van deze postcode hoog. Als echter de meeste percelen pas na meerdere iteratiestappen een postcode toegekend hebben gekregen, is het aannemelijker dat hierin enkele fouten worden gemaakt, doordat de rekenregels voor het toekennen niet altijd overeenkomen met de werkelijkheid. De betrouwbaarheid van de postcode van een dergelijk gebied is lager ten opzichte van het postcodegebied waarbinnen nauwelijks iteratiestappen nodig waren.

De betrouwbaarheidsindicator q beoogt een beeld te geven van deze relatieve betrouwbaarheid. Benadrukt wordt dat deze waarden q op een ordinale meetschaal berekend worden; er kunnen op basis van q slechts relatieve uitspraken gedaan worden. In onderstaand kader is de gebruikte formule weergegeven.

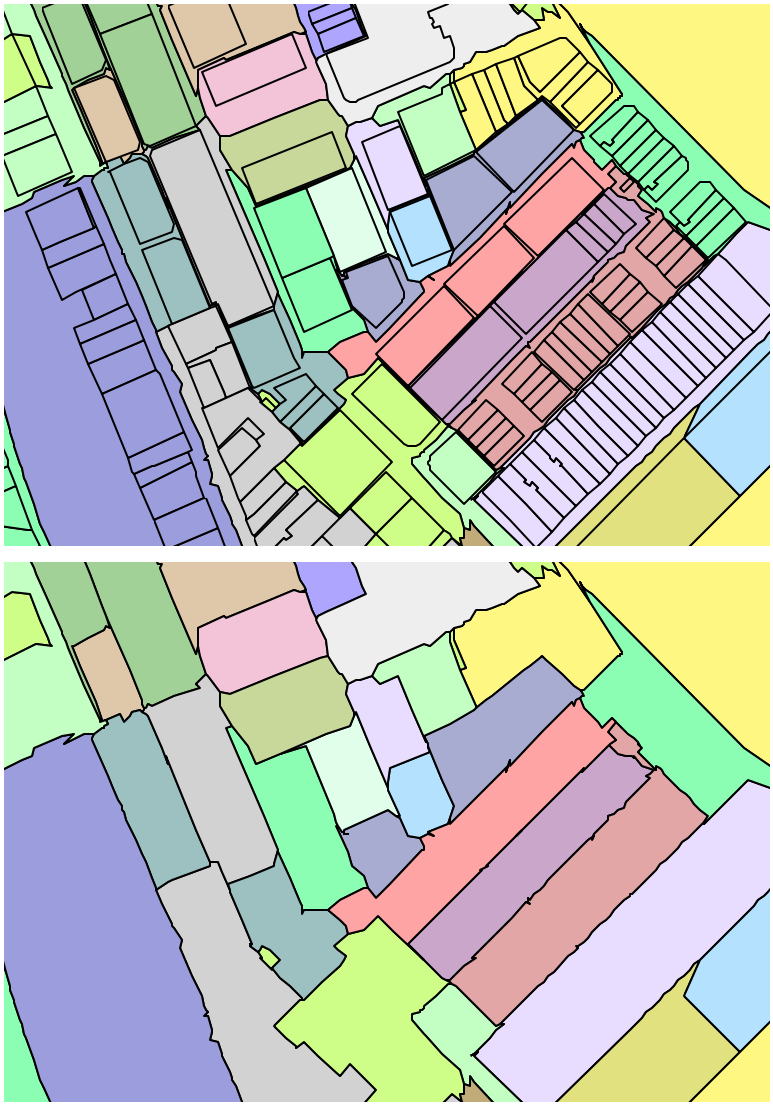
$$\text{Betrouwbaarheidsindicator } q = \frac{1}{n}\left(\sum_{m=1}^{n}\frac{1}{\sqrt{i_m + 1}}\right) \quad \text{met} \begin{cases} n = \text{aantal percelen in postcodegebied} \\ i = \text{aantal iteratiestappen} \\ 0 \le q \le 1 \end{cases}$$

6.5 Creëren 6-positie postcodegebieden

Nu alle (delen van) percelen postcodes hebben en de betrouwbaarheidsindicator q is berekend, kunnen de percelen met dezelfde postcode worden samengevoegd tot 6-positie postcodegebieden. Dit komt overeen met het verwijderen van alle grenzen die aan beide zijden dezelfde postcode hebben. Om dit te implementeren wordt er in de Planar Map geïtereerd over alle faces. Voor elke face wordt weer geïtereerd over de binnen- en buitengrenzen. Van elk segment op de grens wordt de postcode van het aangrenzende perceel vergeleken met de postcode van de face zelf. Als deze postcodes identiek zijn, wordt de edge toegevoegd aan de lijst edges die verwijderd kunnen worden. Omdat op deze manier elke edge twee keer verwijderd zou worden (de grens tussen faces a en b wordt twee keer bekeken, één keer vanuit face a en één keer vanuit face b), wordt hier voor gecorrigeerd.

```
vector<Planar_map::Ccb_halfedge_circulator> edges;
for (faces = pm.faces_begin(); faces != pm.faces_end(); faces++)
{
   if (!faces->is_unbounded())
   {
      faces->set_visited(true);
      h_circ2 = faces->outer_ccb();
      do
      {
         if (((h_circ2->twin()->face()->pc()) == (faces->pc())) &&
                ((h_circ2->twin()->face()->visited()) == false))
         {
             edges.push_back(h_circ2);
```

Dit leidt uiteindelijk tot een lijst met edges die verwijderd worden met een remove-statement. Dit resulteert in een Planar Map met 6-positie postcodegebieden, zie figuur 6.9.



Figuur 6.9 Verwijderen van edges om tot 6-positie postcodegebieden te komen

6.6 Conclusie

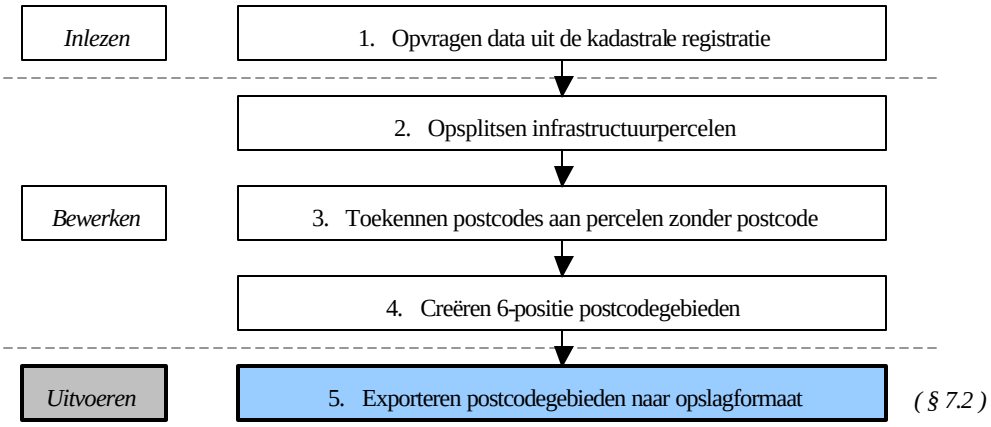
In dit hoofdstuk zijn de bewerkingsstappen op de Planar Map aan bod gekomen. Uitgangspunt is de kadastrale situatie. Om het mogelijk te maken om afzonderlijke delen van een infrastructuurperceel aan de omliggende postcodegebieden toe te delen, is het nodig om deze percelen eerst te skeletteren en op basis van het zo verkregen skelet te splitsen. Het skelet zoals dat berekend kan worden door gebruik te maken van de rekenregels van DeLucia en Black, maakt het niet mogelijk om een perceel in meer dan twee delen te splitsen. Omdat dit vaak wel gewenst is, wordt het skelet uitgebreid met een aantal zijtakken om het perceel in meer delen op te kunnen splitsen. Deze zijtakken verbinden topologische nodes met het skelet. Dit uitgebreide skelet kan ingevoegd worden in de Planar Map, waardoor een aantal nieuwe vlakken ontstaat. Aan deze nieuw ontstane vlakken worden vervolgens de

oorspronkelijke attributen gekoppeld. Na deze stap kunnen de meest geschikte postcodes worden bepaald voor percelen zonder postcode. Hiervoor is het criterium van de langst gemeenschappelijke grens gehanteerd. Er zijn meerdere iteratiestappen nodig om aan alle vlakken een postcode toe te kennen. Deze iteratiestappen zijn het uitgangspunt voor het berekenen van de betrouwbaarheidsindicator. Deze indicator geeft aan in hoeverre een postcodegebied percelen bevat die hun postcode rechtstreeks aan de kadastrale registratie ontlenen, aangezien dit de postcode van het gebied betrouwbaarder maakt dan wanneer het postcodegebied veel percelen bevat die slechts na meerdere iteratiestappen een postcode toegekend kregen. Door nu alle grenzen die geen postcodegebied begrenzen te verwijderen, ontstaan de gewenste 6-positie postcodegebieden.

7 Uitvoer & analyse

7.1 Inleiding

In het vorige hoofdstuk is beschreven op welke wijze 6-positie postcodegebieden kunnen worden berekend op basis van de opgevraagde gegevens uit de kadastrale registratie. Deze postcodegebieden kunnen nu worden geëxporteerd naar een geschikt opslagformaat; de vijfde en laatste stap in het algoritme. De exportmethode en de keuze voor een opslagformaat zijn onderwerp van paragraaf 7.2. Het eindresultaat wordt in paragraaf 7.3 per algoritmestap geanalyseerd en waar mogelijk worden verbeteringen gesuggereerd. Vervolgens komt in paragraaf 7.4 de complexiteit van het algoritme aan bod. Het hoofdstuk sluit af met conclusies in paragraaf 7.5.



Figuur 7.1 Structuur algoritme

7.2 Exporteren naar Oracle Spatial

Nu de Planar Map 6-positie postcodegebieden bevat, kunnen deze worden geëxporteerd naar het definitieve opslagformaat. De vraag is nu in welke vorm deze data geleverd moet kunnen worden. Op basis van de datasets die marktleiders Bridgis en Geodan leveren, kan gesteld worden dat de postcodegebieden als polygonen in de meest gangbare GIS-formaten geleverd moeten kunnen worden. Beide leveren datasets voor gebruik in ArcView en MapInfo, Bridgis levert daarnaast ook in AutoCad-formaat. Om flexibel te kunnen zijn in het te leveren formaat en het te leveren gebied, ligt het voor de hand om te kiezen voor het creëren van een moederbestand, op basis waarvan aan alle gebruikerswensen voldaan kan worden. Hierom is gekozen voor opslag in een databasetabel en meer specifiek voor één tabel in Oracle Spatial. Voordeel van het opslaan in Oracle Spatial ten opzichte van in een niet ruimtelijke database is ten eerste de validatie van de geometrie. Een tweede voordeel is dat steeds meer GIS-pakketten, waaronder bijvoorbeeld ArcGIS, direct een Spatial-tabel kunnen inlezen.

Het exporteren van de data vanuit CGAL naar de databasetabel verloopt in twee stappen. Eerst wordt een ASCII-bestand aangemaakt, waarin de data per postcodevlak wordt weggeschreven. Binnen het onderzoek worden naast de geometrie ook alle aan het postcodevlak gekoppelde attributen geëxporteerd, dit om het mogelijk te maken

tussenresultaten te visualiseren. In een productieomgeving volstaat het koppelen van een postcode aan de geometrie, de overige attributen zijn dan minder relevant.
Voor het aanmaken van het postcodevlaksgewijze ASCII-bestand wordt in de Planar Map over alle faces geïtereerd. Van elke face worden de volgende attributen opgevraagd en geëxporteerd: perceelnummer, postcode, betrouwbaarheidsindicator, aantal benodigde iteratiestappen, een boolean die aangeeft of het perceel gesplitst is en de bijbehorende polygoon. Deze polygoon moet eerst worden opgebouwd, aangezien de Planar Map alleen uit edges bestaat (a-b, b-c, c-d). Deze edges worden tot een polygoon a-b-c-d aaneen gekoppeld. De polygoon die zo ontstaat, wordt in het OpenGIS Well Known Text (WKT) formaat weggeschreven naar de ASCII-file.

De tweede stap is het inlezen van het ASCII-bestand in een databasetabel. Daartoe wordt deze tabel eerst aangemaakt:

```
create table pm_faces_friso
(
   parcel_id number(10),
   is_splitted number(1),
   postal_code varchar2(70),
   to_split number(1),
   step number(2),
   quality number(5,3),
   shape mdsys.sdo_geometry,
   shapewkt clob
);
```

In de onderste twee regels is te zien dat er in meerdere velden geometrie wordt opgeslagen. Dit is gedaan omdat bij het inlezen bleek dat een aantal polygonen in WKT-formaat te lang waren om door SQL*Loader direct te worden omgezet in een polygoon in het veld mdsys.sdo_geometry. De string in WKT-formaat wordt daarom als clob (character large object) ingelezen.

```
load data
infile 'PlanarMap.wkt' "str"
append
into table pm_faces_friso
fields terminated by '\t'
trailing nullcols (
   parcel_id float external,
   is_splitted float external,
   postal_code char(70),
   to_split float external,
   step float external,
   quality float external,
   shapewkt char(500000)
)
```

Het veld mdsys.sdo_geometry blijft bij het inlezen leeg. Na het inlezen wordt dit veld alsnog gevuld met een update-statement:

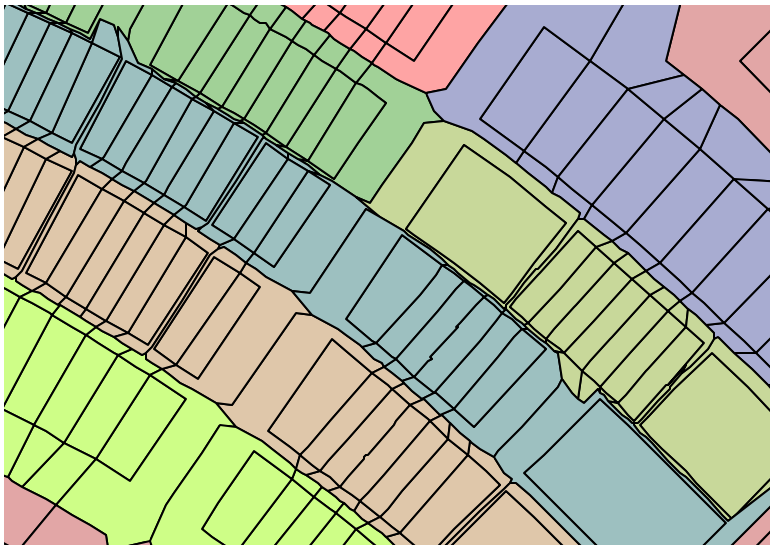
```
update pm_faces_friso
set shape = fromwkt2(shapewkt);
```

7.3 Analyse resultaten

Vanuit de tabel in Oracle Spatial zijn de resultaten gevisualiseerd met ArcView. In deze paragraaf zullen achtereenvolgens de resultaten van het skeletteren, het toekennen van postcodes, het berekenen van de betrouwbaarheidsindicator en het totale eindresultaat worden beoordeeld.

7.3.1 Resultaten skeletteren

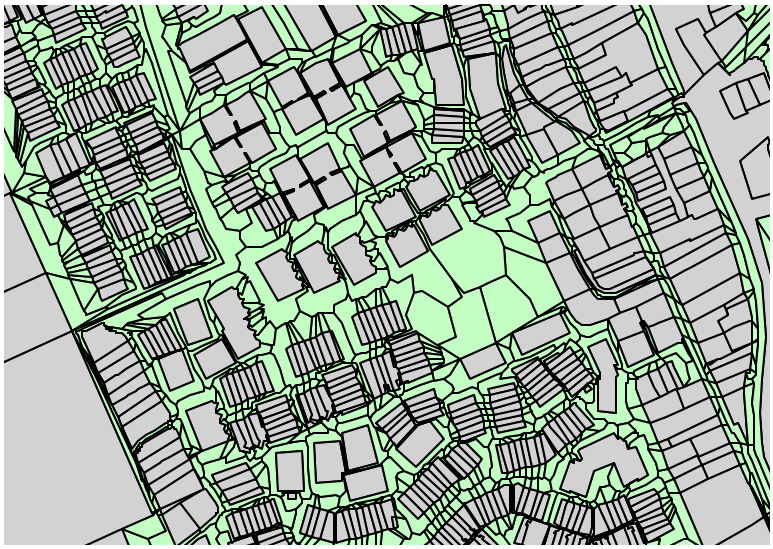
In paragraaf 6.2 is beschreven hoe percelen geskeletteerd en gesplitst kunnen worden met behulp van de rekenregels van DeLucia en Black. In de vorm van de skeletten zijn met name de pieken herkenbaar die berekend worden in de 0-driehoeken. Deze pieken zijn duidelijk herkenbaar in figuur 7.2, waarin het gaat om 0-driehoeken op kruisingen. Hier liggen twee 0-driehoeken naast elkaar. Op elke kruising is hierdoor een combinatie van een piek naar links en een piek naar rechts te zien.



Figuur 7.2 Ongewenste pieken op kruisingen door de oplossing voor 0-driehoeken

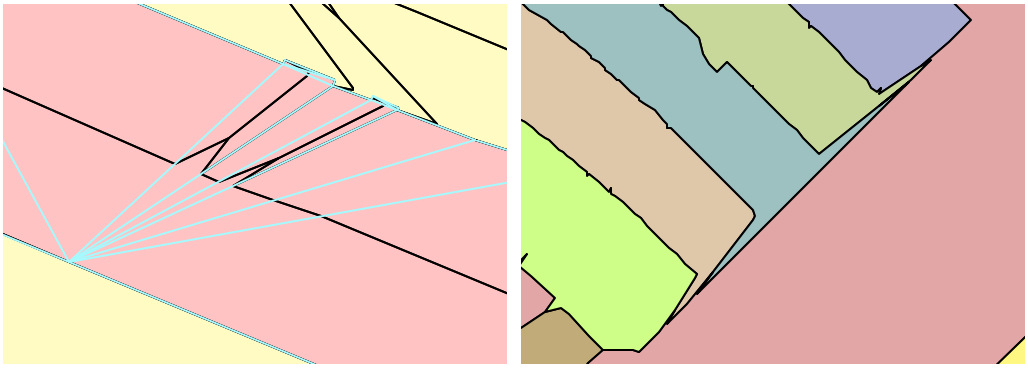
Het is aan te bevelen om dergelijke pieken te verwijderen. Hiervoor zijn twee mogelijkheden, namelijk het verwijderen van deze pieken in een post-processing stap en het voorkomen van deze pieken door het hanteren van andere rekenregels voor 0-driehoeken. De eerste mogelijkheid is om als post-processing stap een lijngeneralisatie of een spike filter uit te voeren. De tweede mogelijkheid is het hanteren van de in paragraaf 3.4 geïntroduceerde alternatieve rekenregels van Jones voor 0-driehoeken. Zijn techniek kijkt naar de richting van de drie lijnstukken die in de omliggende driehoeken worden berekend. De twee lijnstukken die het meeste in elkaars verlengde liggen, worden direct met elkaar verbonden. Vervolgens wordt het midden van dit verbindende lijnstuk verbonden met het derde lijnstuk (zie nogmaals figuur 3.14). In dezelfde paragraaf is dit idee verder uitgewerkt voor kruisingen, waarbij andere rekenregels gelden als twee 0-driehoeken (bijna) aan elkaar grenzen (zie nogmaals figuur 3.15). Implementeren van deze methode geeft een fraaier resultaat en is binnen CGAL goed mogelijk, doordat de Constrained Delaunay triangulatie goed toegankelijk is door functies die gebruik maken van buurrelaties.

Het toepassen van de regels van Jones voor 0-driehoeken (inclusief de ontwikkelde variant voor kruisingen) is te prefereren boven de eerste mogelijkheid, omdat de uitkomsten van Jones' regels en de daarop ontwikkelde varianten voorspelbaarder zijn en het direct toepassen sneller werkt dan het achteraf filteren. Verder onderzoek hiernaar is wenselijk, mede ook omdat toepassen van deze alternatieve regels naast de kruisingen ook veel pieken op T-splitsingen zal verwijderen. Deze pieken zijn op veel plaatsen te vinden in figuur 7.3, waar de resultaten van skeletteren op een groter gebied zijn weergegeven. In groen de geskeletteerde percelen, in grijs de niet geskeletteerde.



Figuur 7.3 Resultaten van skeletteren binnen de bebouwde kom (geskeletteerde percelen in groen).

In hoofdstuk 3 zijn nog twee problemen beschreven die kunnen optreden bij het skeletteren, namelijk het voorkomen van valse 0-driehoeken en van verschoven 0-driehoeken. In figuur 7.4 is links een voorbeeld te zien van valse 0-driehoeken. Door de twee kleine uitstulpsels in de bovengrens van het perceel (een deel van de snelweg A12) ontstaan twee keer twee hele kleine driehoeken. Door deze extra driehoeken ontstaan twee 0-driehoeken, alsof het om twee T-splitsingen gaat. Hierdoor krijgt het skelet twee zijtakken. In hoofdstuk 3 is reeds Uitermarks oplossing gegeven: het negeren van driehoeken met een oppervlakte kleiner dan een minimum. Idee hierachter is dat normaal gesproken driehoeken altijd de gehele wegbreedte beslaan en daarmee een oppervlakte zullen hebben die duidelijk groter is dan de oppervlakte van deze kleine driehoeken.



Figuur 7.4 Valse 0-driehoeken (links) en de gevolgen van verschoven 0-driehoeken (rechts)

In de figuur zijn rechts de gevolgen van verschoven 0-driehoeken te zien: de pieken wijken erg uit. Ook hiervoor is Uitermarks oplossing uitgewerkt in hoofdstuk 3. Het idee is dat eerst een lijngeneralisatie wordt uitgevoerd op de perceelsgrenzen, vervolgens worden extra punten toegevoegd (maximale tussenafstand 15 m.) en dan pas getrianguleerd voor het berekenen van het skelet. Door de extra punten worden de driehoeken in de triangulatie gelijkvormiger, waardoor verschoven pieken in het skelet niet meer voorkomen.

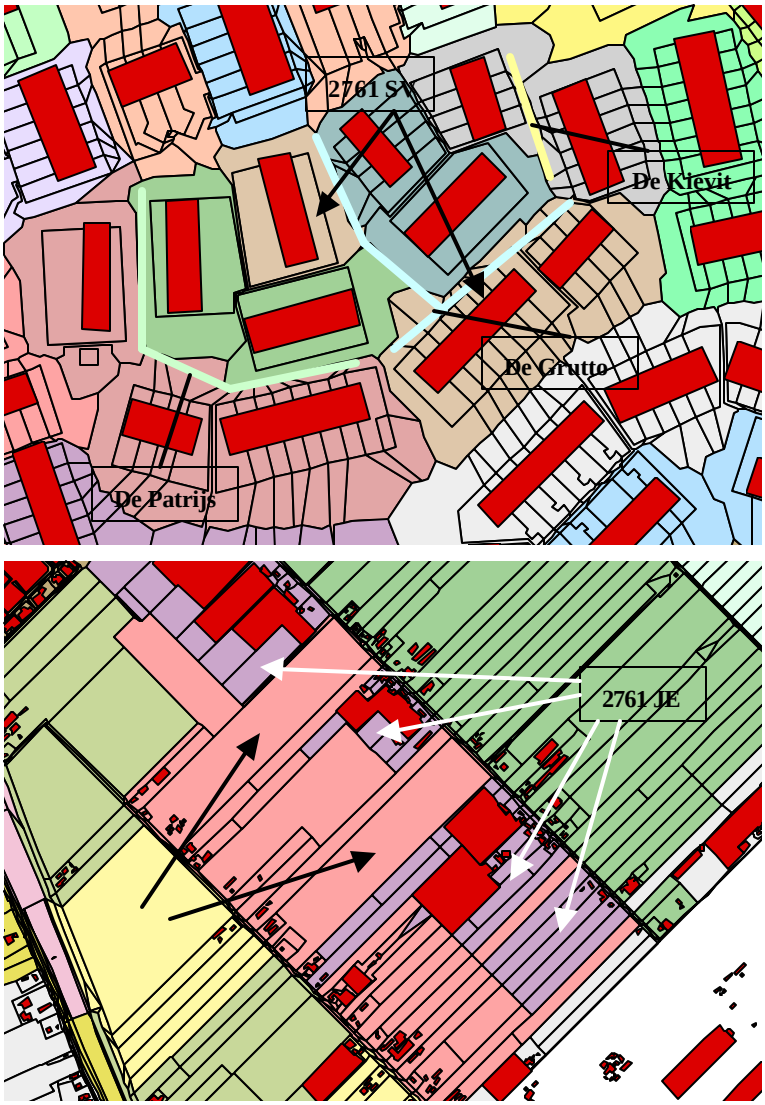
Binnen het onderzoek zijn beide filteringen wegens tijdgebrek niet geïmplementeerd. Gezien de resultaten is het wel aan te bevelen deze filteringen uit te voeren.

7.3.2 Resultaten toekennen postcodes

Voor percelen zonder postcode is de meest waarschijnlijke postcode bepaald, waarbij de langst gemeenschappelijke grens als criterium is gehanteerd. In paragraaf 6.3 is deze methode behandeld, waarbij meerdere iteratiestappen nodig bleken. In figuur 7.5 zijn de resultaten gevisualiseerd. Boven is de oorspronkelijke kadastrale situatie weergegeven met in grijs de percelen zonder postcode. Onder is het resultaat te zien na skeletteren en bepalen van de meest waarschijnlijke postcode.

Figuur 7.5 Oorspronkelijke kadastrale situatie en resultaat na opsplitsen en toekennen postcode

Op het eerste gezicht zijn er fraaie postcodegebieden te herkennen. Er blijkt echter een ongewenst effect op te treden: er zijn postcodegebieden die uit meerdere vlakken bestaan. In figuur 7.6 zijn hier twee voorbeelden van gegeven; één voorbeeld binnen de bebouwde kom en één buiten de bebouwde kom. Om de situatie te verduidelijken zijn de huizen en wegen uit de Top10Vector aan het beeld toegevoegd.



Figuur 7.6 Twee voorbeelden van postcodegebieden die uit meerdere vlakken bestaan

In het geval links ontstaan er twee vlakken voor postcode 2761 SV doordat het stratenplan wat ingewikkeld loopt. Deze wijk is deels per auto bereikbaar en deels alleen over het trottoir. De straatnamen (en daarmee de postcodes) volgen deze trottoirs. Zo staan de huizen langs de blauwe lijn aan De Grutto en de huizen langs de groene lijn aan De Patrijs. Doordat het rechter blok huizen van De Patrijs tussen de huizenblokken van De Grutto ligt, wordt een deel van de infrastructuur toegedeeld aan het postcodegebied van De Patrijs.

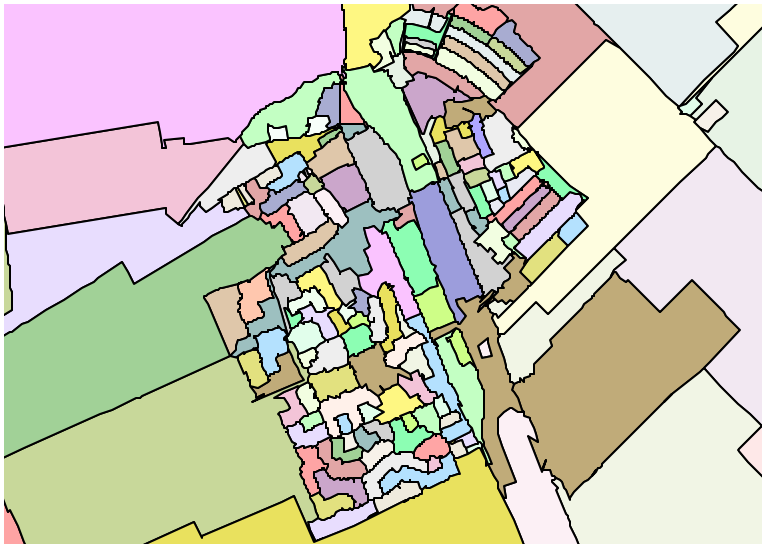
In het geval rechts lopen twee parallelle tochtwegen diagonaal door het beeld (van midden-links naar midden-onder en van midden-boven naar midden-rechts). Aan deze wegen zijn een aantal land- en tuinbouwbedrijven gevestigd. De twee grote agrarische percelen (met zwarte pijlen gemarkeerd in de figuur) grenzen aan beide wegen en scheiden zo de verschillende gebieden met postcode 2761 JE.

De oplossing voor dergelijke gevallen ligt niet direct voor de hand. In het geval in de bebouwde kom zijn een aantal delen van het grote infrastructuurperceel toegewezen aan het verkeerde postcodegebied. Additionele informatie over het stratenplan zou kunnen

voorkomen dat perceelsdelen die binnen een bepaalde straat vallen, worden toegedeeld aan een andere straat. De implementatie van dit idee zal niet eenvoudig zijn.

In het geval buiten de bebouwde kom ligt het probleem niet in het toewijzen van opgesplitste delen, maar juist in de niet gesplitste percelen. Wellicht kan dit opgelost worden door het splitsen van deze percelen, bijvoorbeeld als de oppervlakte een bepaald maximum overschrijdt. Het is dan echter nog de vraag hoe deze splitsing uitgevoerd kan worden. In tegenstelling tot bij het skeletteren en splitsen van infrastructuurpercelen zou je dit perceel niet in de lengterichting maar in de dwarsrichting (parallel aan de wegen) willen splitsen.

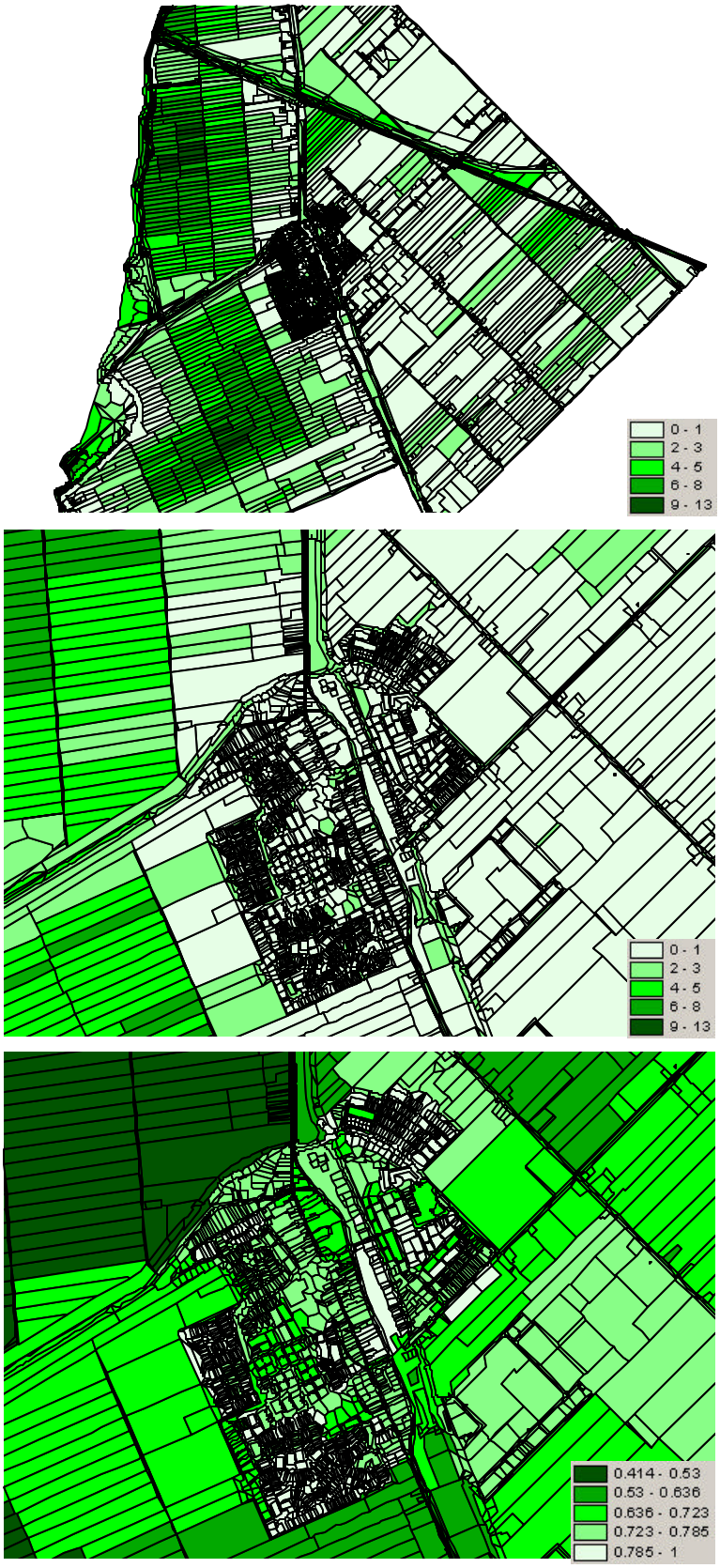
Voor de gesuggereerde oplossingen voor beide gevallen is het twijfelachtig of deze oplossingen altijd het gewenste resultaat hebben. Daarom ligt het meer voor de hand om de oplossing te zoeken in een post-processing stap na het toekennen van de postcodes. In deze stap zou gekeken kunnen worden of alle (deel)percelen met dezelfde postcode met elkaar in verbinding staan. Als dat niet het geval is, zou opnieuw gekeken moeten worden naar de toegekende postcodes van de (deel)percelen die de verschillende delen van het postcodegebied van elkaar scheiden. Hierbij zou beoordeeld moeten worden of de toegekende postcode te veranderen is in de postcode van het gescheiden postcodegebied. In figuur 7.7 zijn de postcodegebieden die uit meerdere vlakken bestaan, gearceerd.



Figuur 7.7 Postcodegebieden bestaand uit meerdere vlakken (gearceerd)

7.3.3 Resultaten betrouwbaarheidsindicator

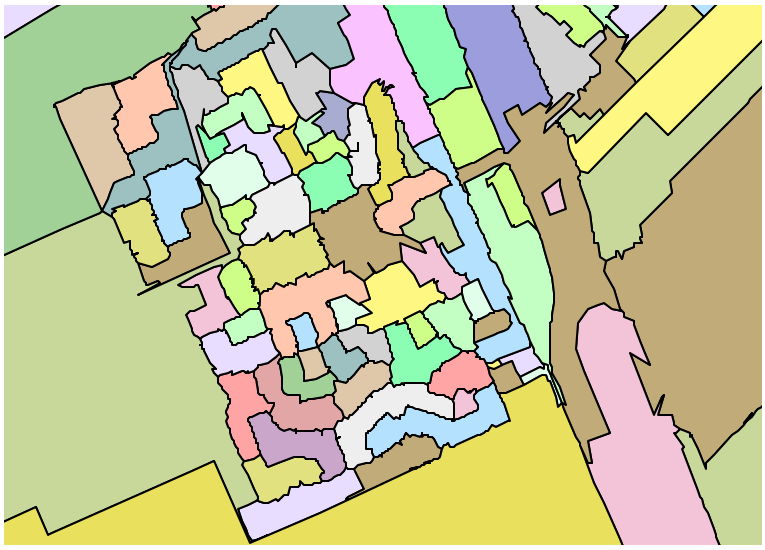
De betrouwbaarheidsindicator beoogt een beeld te geven van de relatieve betrouwbaarheid door inzichtelijk te maken in hoeverre de postcode van een postcodegebied afkomstig is uit de kadastrale registratie. Het idee hierachter is dat de postcode minder betrouwbaar wordt als deze is toegekend omdat het buurperceel van een buurperceel (etc., etc.) van een perceel een bepaalde postcode heeft. Dit uitgangspunt is verwerkt in de in paragraaf 6.4 geïntroduceerde formule. In figuur 7.8 is voor zowel het buitengebied (boven) als voor de bebouwde kom (midden) per perceel gevisualiseerd hoeveel iteratiestappen nodig waren voor het toekennen van een bepaalde postcode. Goed zichtbaar is dat alleen in landelijk gebied dit aantal flink kan oplopen. Binnen de bebouwde kom volstaan drie iteratiestappen vrijwel altijd. Dit is terug te zien in de op basis van dit aantal iteratiestappen berekende betrouwbaarheidsindicator, waarvan de resultaten ook in figuur 7.8 (onder) zijn weergegeven. Op basis hiervan kan aangenomen worden dat de methode voldoende betrouwbaar is voor bebouwd gebied.



Figuur 7.8 Aantal iteratiestappen (boven en midden) en de op basis daarvan berekende betrouwbaarheidsindicator q

7.3.4 Eindresultaat

Het uiteindelijk resultaat is een bestand met polygonen, zoals in figuur 7.9 is afgebeeld. In de figuur zijn een aantal spikes te zien die gefilterd kunnen worden zoals beschreven in paragraaf 7.3.1. De cruciale vraag is nu of de nieuw ontwikkelde methode om postcodevlakken te berekenen goed werkt. In hoofdstuk 2 was immers te zien hoe in één van de huidige bestanden de grenzen van de postcodegebieden dwars door gebouwen lopen.



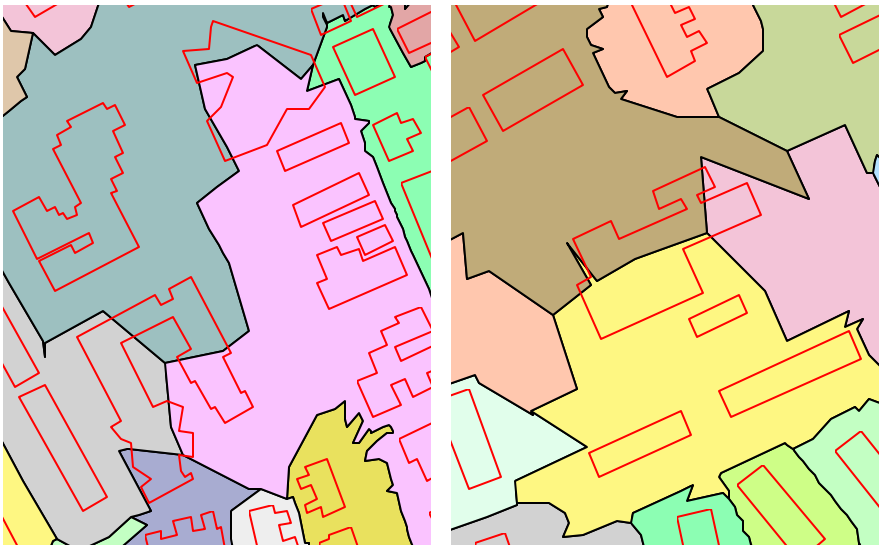
Figuur 7.9 De berekende 6-positie postcodevlakken

Figuur 7.10 Berekende postcodegebieden gecombineerd met de bebouwing uit de Top10Vector

In figuur 7.10 is goed te zien dat de nieuwe postcodegebieden keurig bij de bebouwing passen. Slechts in drie gevallen snijden de grenzen met een gebouw; deze drie situaties zijn afgebeeld in figuur 7.11. In het linkerbeeld snijden twee gebouwen met de grenzen van de postcodegebieden. Het bovenste van de twee gebouwen is het gemeentehuis, het onderste is de openbare basisschool. Beide staan op gemeentegrond. Het gaat hierbij om een zeer groot perceel waarop onder andere negen wegen, het marktplein, de basisschool en het gemeentehuis gelegen zijn. Vanwege de verschillende functies is in de kadastrale registratie geen cultuurcode ingevoerd, maar staat de bebouwingscode op onbebouwd. Daarom wordt

dit perceel geskeletteerd en gesplitst, waarbij niet voorkomen kan worden dat de grenzen deze gebouwen snijden. Gezien de voorkennis van de auteur t.a.v. het testgebied was het geen verrassing dat deze beide gebouwen doorsneden werden door postcodegebieden.

De situatie rechts in dezelfde figuur was echter wel een verrassing. Het grote gebouw bestaat uit een boerderij en een aantal daaraan gebouwde schuren. Deze oude boerderij is midden in de bebouwde kom gelegen, maar nog wel in gebruik, waarbij de landbouwgrond buiten het dorp gelegen is. Navraag bij de bewoners leerde dat de gemeente ruim 20 jaar geleden, toen het dorp aanzienlijk kon uitbreiden, de voor de nieuwbouw benodigde grond heeft aangekocht, inclusief de grond waar de boerderij op stond. De boerderij werd verpacht. Indertijd ging de gemeente er echter van uit dat ook de overige grond van de boer snel voor woningbouw bestemd zou worden, wat de boer tot bedrijfsbeëindiging zou dwingen. Dat zou ook reden zijn om de pachtovereenkomst op te kunnen zeggen en zo de vrijkomende grond voor woningbouw te bestemmen. Door een restrictief bouwbeleid in het landelijk gebied is dit echter nooit gebeurd en bestaat deze bijzondere situatie nog steeds. De boerderij staat nu op een groot perceel waarbinnen ook alle wegen in de omgeving vallen. Net als bij het andere geval is voor dit perceel geen cultuurcode bekend, maar staat de bebouwingscode op onbebouwd. Er kan hier met recht gesteld worden dat het een uitzonderingssituatie betreft. Het is daarom de vraag of gepoogd moet worden dit op te lossen. Het zou eventueel mogelijk zijn om voor alle te skeletteren perceel een overlay uit te voeren met de bebouwingslaag uit de Top10, GBKN of kadastrale registratie zelf om zo eerst alle bebouwing uit het te skeletteren perceel te snijden. Deze extra vlakken zouden dan niet geskeletteerd en gesplitst moeten worden.

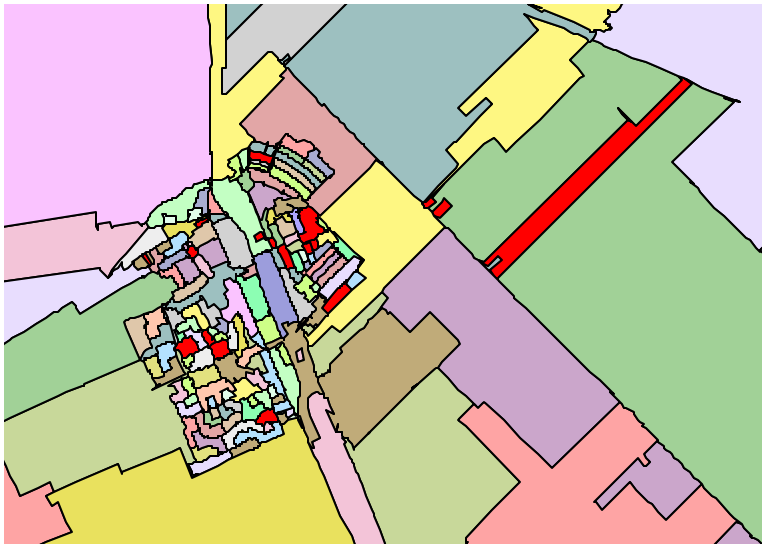


Figuur 7.11 Drie gebouwen worden doorsneden door de grenzen van de postcodegebieden

Op de volgende pagina zijn in figuur 7.12 twee voorbeelden van de resultaten opgenomen waarin naast de postcodegebieden ook de bebouwing te zien is. Deze bebouwing is afkomstig uit de Top10Vector. Om een vergelijking mogelijk te maken is ook een voorbeeld van het Bridgis 6-positie vlakbestand toegevoegd, gecombineerd met de Top25. Nu is goed zichtbaar dat in het Bridgisbestand (het enige nu verkrijgbare 6-positie postcodevlakbestand) veel bebouwing doorsneden wordt door de postcodevlakgrenzen. In de resultaten van het nieuwe algoritme komt dit vrijwel niet voor.

Figuur 7.12 Twee illustraties van de resultaten van het ontwikkelde algoritme. Ter vergelijking de nu op de markt verkrijgbare Bridgis 6-positie postcodevlakken, gecombineerd met de Top25

Het probleem van meerdere postcodes per vlak is binnen het onderzoek niet opgelost. Dit probleem wordt veroorzaakt doordat aan sommige percelen meerdere akr-objectadressen gekoppeld zijn, waarbij die adressen verschillende postcodes kunnen hebben. Door tijdgebrek is de in paragraaf 1.3 voorgestelde cartografische oplossing (gebied opdelen in taartpunten) niet geïmplementeerd. Wel is in figuur 7.13 geïllustreerd welke postcodegebieden meerdere postcodes hebben. Deze gebieden zijn in rood weergegeven.



Figuur 7.13 Postcodegebieden met meer dan één postcode (rood)

7.4 Complexiteit van het algoritme

Binnen het ontwikkelde algoritme zijn de verschillende point location queries op de Planar Map de duurste operaties qua rekentijd. De rekentijd neemt logaritmisch toe met het aantal x-curves in de Planar Map. Het algoritme kan efficienter gemaakt worden door het aantal point location queries en het aantal x-curves tot een minimum te beperken. Naast de point location queries die in de code van het algoritme zijn terug te vinden, gebruikt CGAL binnen de insert-operatie van x-curves in de Planar Map zelf ook een point location query. Met deze query wordt bepaald in welke face de curve komt te liggen en of één of beide eindpunten van de curve reeds deel uitmaken van de Planar Map. Het aantal point location queries dat in de huidige implementatie wordt gebruikt, kan worden gereduceerd om zo tot een efficiëntere implementatie te komen. Zo kan er voor gekozen worden om gebruik te maken van de topologische relaties van de perceelsgrenzen bij het invoegen van deze grenzen als x-curve in de Planar Map. Hierdoor kunnen de grenzen als een ketting van x-curves worden ingevoegd, waarbij steeds gebruik gemaakt kan worden van het insert_at_vertex-commando. Met deze aanpassing kan tevens de point location query die gebruikt wordt om de attributen aan de ontstane faces te koppelen, voorkomen worden. Het insert-commando levert immers een halfedge handle op, waarvan de bijbehorende face kan worden opgevraagd. Aan deze face kunnen dan de perceelsattributen gekoppeld worden.

Aangezien de rekentijd van de point location strategy logaritmisch toeneemt met het aantal x-curves in de Planar Map, is het belangrijk om te zorgen dat er geen overbodige x-curves in de Planar Map voorkomen. In de huidige implementatie is dit nog niet optimaal, aangezien er meer zijtakken van het skelet worden berekend dan strikt noodzakelijk is. Eigenlijk zijn alleen de zijtakken nodig naar topologische nodes die twee verschillende postcodegebieden van elkaar scheiden.

7.5 Conclusie

In dit hoofdstuk is de Planar Map met de 6-positie postcodevlakken geëxporteerd naar een tabel in Oracle Spatial. Op basis van deze tabel kunnen de door de gebruiker gewenste formaten worden afgeleid, waarbij de huidige marktpartijen de formaten van ArcView en MapInfo leveren. In de tabel wordt de geometrie in polygoonvorm opgeslagen. Na deze laatste stap in de productie van de 6-positie postcodevlakken zijn de resultaten geanalyseerd.

De resultaten van het skeletteren en splitsen zijn goed, alleen de pieken ten gevolg van de 0-driehoeken zorgen soms voor rommelige grenzen. Aannemelijk is dat de resultaten verder verbeteren na het implementeren van de regels van Jones voor 0-driehoeken, zeker als ook de op Jones gebaseerde variant voor kruisingen (twee aangrenzende 0-driehoeken) wordt meegenomen. Verder is nog enige verbetering te verwachten door het toevoegen van extra punten op perceelsgrenzen om de maximale tussenafstand te reduceren tot 15 m. Lijngeneralisatie kan ook uitkomst bieden, maar heeft als risico dat buiten- en binnengrenzen elkaar gaan snijden. Daarnaast bestaat ook het risico dat juist de ongewenste punten behouden blijven doordat deze als kenmerkend worden geclassificeerd.

Het proces van iteratief postcodes toekennen functioneert goed, maar heeft in combinatie met de vlakgerichte aanpak soms tot gevolg dat een postcodegebied uit meerdere vlakken bestaat. Er is niet direct één duidelijke oplossing aan te wijzen. Mogelijk zou een postprocessing stap na het toekennen van de postcodes, maar voor het creëren van de postcodegebieden soelaas bieden. In deze stap zou gekeken kunnen worden of alle percelen met dezelfde postcode met elkaar verbonden zijn. Als dat niet het geval is, kan geprobeerd worden om de afzonderlijke gebieden met elkaar te verbinden door aan tussenliggende percelen een andere postcode toe te kennen.
De betrouwbaarheidsindicator is gebaseerd op het aantal benodigde iteratiestappen en geeft een beeld van de relatieve betrouwbaarheid van de postcodegebieden. Alhoewel er geen absolute uitspraken gedaan kunnen worden, lijkt de betrouwbaarheid in de bebouwde kom in orde. Er bestaan voldoende percelen mét postcode om tot aannemelijke postcodegebieden te komen.

Het uiteindelijke resultaat is een 6-positie postcodevlakbestand dat voldoet aan de vooraf gestelde wensen: het is van goede kwaliteit, grenzen vallen samen met infrastructuur en alleen in uitzonderingsgevallen worden gebouwen doorsneden. Het enige echte minpunt is het gegeven dat postcodegebieden niet altijd uit één gebied bestaan.

8 Conclusies en aanbevelingen

8.1 Conclusies

In deze scriptie is een nieuw algoritme beschreven dat op basis van gegevens uit de kadastrale registratie 6-positie postcodegebieden berekend. Het is nu belangrijk om de vraag te stellen in hoeverre de resultaten van het algoritme voldoen aan de verwachtingen. De resultaten worden daarom per stap behandeld:

Skeletteren en splitsen

Het skeletteren en splitsen leidt tot realistische postcodegebieden, waarbij de grenzen vaak samenvallen met infrastructuur. Ten aanzien van het skeletteren zelf kan gesteld worden dat het skeletteren in principe het gewenste resultaat heeft, alleen de pieken in de grenzen zijn niet fraai. Deze pieken worden veroorzaakt door de gehanteerde rekenregels van DeLucia en Black voor 0-driehoeken.

Toekennen postcodes

Op basis van visuele controle kan gesteld worden dat het proces van iteratief postcodes toekennen goed functioneert. Met name in de bebouwde kom worden de delen van de geskeletteerde percelen aan de beoogde postcodegebieden toegedeeld. Duidelijk minpunt is dat de huidige methode van toekennen leidt tot meerdere gebieden met dezelfde postcode. De gebruikte methode probeert dit niet expliciet te voorkomen. Het is vermoedelijk zo dat dit verschijnsel deels inherent is aan de vlakgerichte aanpak.

Koppelen betrouwbaarheidsindicator

De betrouwbaarheidsindicator is gebaseerd op het aantal benodigde iteratiestappen en geeft een beeld van de relatieve betrouwbaarheid van de postcodegebieden. Aangezien hierdoor geen absolute uitspraken gedaan kunnen worden over de betrouwbaarheid, geeft de betrouwbaarheidsindicator alleen een beeld van de effecten van de gehanteerde methode voor het toekennen van postcodes. Daarmee is deze indicator met name tijdens het onderzoek van belang geweest. Op basis van de resultaten van de indicator kan gesteld worden dat (zeker in de bebouwde kom) de kadastrale registratie voldoende houvast biedt voor het realistisch bepalen van de postcodegebieden.

Al het voorgaande samenvattend kan geconcludeerd worden dat de ontworpen methode in staat is om kwalitatief betere postcodevlakbestanden te creëren dan de huidige methoden. Het uiteindelijke 6-positie postcodevlakbestand voldoet aan de vooraf gestelde wensen: het is van goede kwaliteit, grenzen vallen samen met infrastructuur en alleen in uitzonderingsgevallen worden gebouwen doorsneden. Het enige echte minpunt is het gegeven dat postcodegebieden niet altijd uit één gebied bestaan.

Bij de ontwikkeling van het algoritme is gebruik gemaakt van de Computational Geometry Algorithms Library (CGAL). Dit is een in C++ geschreven bibliotheek die veel datatypen en algoritmes bevat op het gebied van de Computational Geometry. Pluspunten van CGAL zijn de flexibele opzet, waarbij alle klasses eenvoudig zijn uit te breiden, en de goede toegankelijkheid van de datastructuren door specifieke bevragingen, iterators en circulators. Minpunten zijn de steeds opduikende precisieproblemen, de soms onvolledige manuals en

het gebrek aan commentaar in de source code van CGAL. Geconcludeerd kan worden dat CGAL ook voor toepassingen binnen de geografische informatiesystemen geschikt is, maar dat hierbij ruime ervaring met C++ en het opbouwen van voldoende ervaring met CGAL kritieke succesfactoren zijn. Een uitgebreidere beschrijving van de ervaringen met CGAL is te vinden in Appendix B.

8.2 Aanbevelingen

Op basis van de conclusies in de vorige paragraaf kunnen een aantal aanbevelingen worden gedaan om de resultaten van het ontwikkelde algoritme verder te verbeteren. Deze aanbevelingen staan (in de ogen van de auteur) gerangschikt in volgorde van noodzaak:

- Verricht nader onderzoek naar uitbreiding van het algoritme om zo te voorkomen dat aan één postcode meerdere gebieden gekoppeld worden. De oplossing kan wellicht gezocht worden in het analyseren en wijzigen van de toegekende postcodes om zo de afzonderlijke gebieden alsnog met elkaar te verbinden of in het opdelen van zeer grote percelen die potentieel als buffer tussen de afzonderlijke gebieden functioneren. Bij dit opdelen zou per driehoek in de triangulatie een postcode bepaald kunnen worden in plaats van per perceel. Duidelijk nadeel hiervan is wel dat wordt afgeweken van het principe van gebruik maken van in werkelijkheid bestaande grenzen.

- Implementeer de alternatieve regels van Jones voor 0-driehoeken in plaats van de regels van DeLucia en Black (zie figuur 3.14). Hiermee kunnen de pieken uit de skeletten verwijderd worden zonder de risico's die kleven aan bijvoorbeeld lijngeneralisatie als post-processing stap.

- Implementeer ook de op Jones gebaseerde variant voor kruisingen. Deze alternatieve regels voor twee aangrenzende 0-driehoeken (zie figuur 3.15) zijn nu alleen in theorie ontwikkeld.

- Verricht nader onderzoek naar mogelijke oplossingen voor percelen met meerdere postcodes, waarbij zeer kritisch beschouwd moet worden of de oplossing niet erger is dan de kwaal. De aanpak in het algoritme staat juist in het teken van het vermijden van 'berekende' grenzen; splitsen op basis van soms onnauwkeurige bestanden als het ACN zou dit pluspunt teniet doen. Onderzoek daarom ook welk belang de praktijk hecht aan het verwijderen van deze gebieden met meerdere postcodes. Immers, zo lang het om een gebied met beperkte omvang gaat, is de situatie in het postcodebestand wél een betrouwbare weergave van de werkelijkheid.

- Implementeer het invoegen van extra tussenpunten op perceelsgrenzen voor het trianguleren zoals Uitermark die heeft voorgesteld, om zo verschoven 0-driehoeken te voorkomen. Hierbij kan geëxperimenteerd worden met de door Uitermark gehanteerde maximale afstand van 15 m. Kritisch moet gekeken worden of het uitvoeren van een lijngeneralisatie voor het invoegen van extra punten, zoals Uitermark die voorstelt, wel gewenst is. Het risico bestaat dat grenzen elkaar gaan snijden. Als de nadelen van lijngeneralisatie niet opwegen tegen de voordelen, kan de grotere bestandsomvang beter voor lief worden genomen. Om valse 0-driehoeken te voorkomen is dan wel de filtering op minimale oppervlakte van een driehoek aan te bevelen, tenzij het hanteren van de alternatieve rekenregels van Jones afdoende blijken voor het voorkomen van valse 0-driehoeken.

Naast deze verbeteringen en uitbreidingen van het ontwikkelde algoritme zijn een aantal algemenere aanbevelingen te doen:

- Test het algoritme uitgebreider met meer verschillende datasets. Binnen het onderzoek is geen ruimte geweest voor uitgebreide tests die een kwantitatieve vergelijking met de bestaande 6-positie postcodebestanden mogelijk maken.

- Leidt uit het 6-positie postcodebestand ook de 5- en 4-positie postcodegebieden af en beoordeel of het nieuwe algoritme ook voor deze minder gedetailleerde producten tot een beter resultaat leiden dan de huidige productiemethoden.

- Onderzoek of CGAL de beste omgeving is om dit algoritme in te implementeren. Wellicht zijn deze bewerkingen ook mogelijk in commerciële GIS-pakketten.

- Analyseer de invloed van de kwaliteit van de AKR. De correctheid van de cultuur- en bebouwingscode en de volledigheid van de akr-objectadressen is van belang voor de kwaliteit van de resultaten van het algoritme.

Literatuurlijst

- Ai, T., en P.J.M. van Oosterom, 'GAP-Tree Extensions Based on Skeletons'. In: D. Richardson en P.J.M. van Oosterom (eds), *Advances in Spatial Data Handling, 10th International Symposium on Spatial Data handling*. Berlin: Springer-Verlag, 2002, p.501-513.

- Berg, M. de, et al., *Computational geometry, algorithms and applications*. Berlin: Springer-Verlag Berlin Heidelberg New York, 1997.

- *CGAL Basic library manual*. CGAL Consortium, 2002

- Chithambaram, R. et al., 'Skeletonizing polygons for map generalization'. In: *Technical papers, ACSM-ASPRS Convention, Cartography and GIS/LIS, Volume 2* Bethesda: ACSM, 1991, p.44-54.

- DeLucia, A., en T. Black, 'A comprehensive approach to automatic feature generalization'. In: *Proceedings of 13th International Cartographic Conference*. International Cartographic Association, 1987, p.169-191.

- Fabri, A. et al., *On the design of CGAL, the Computational Geometry Algorithms Library (research report)*. Saarbrücken: Max-Planck-Institut für Informatik, 1998.

- Jones, C.B. et al., 'Map Generalization with a Triangulated Data Structure'. *Cartography and Geographic Information Systems*, 22-4 (1995), p.317-331.

- Oosterom, P.J.M. van, 'The GAP-tree, an approach to on-the-fly map generalization of an area partitioning'. In: J.C. Müller et al. (eds), *GIS and Generalization; methodology and practice*. London: Taylor and Francis, 1995, p.120-132.

- Oosterom, P.J.M. van, 'Hartlijnen van wegennetwerken'. *Informatieblad Kadaster*, 7 (1999), p.10-11.

- Rengelink, M., *Automatisch afleiden en classificeren van woningen uit kadastrale gegevens*. Delft, 2000.

- Schut, T.G., en B. Maessen, *Haalbaarheid 6PPC-vlakkenkaart (eindrapportage)*. Apeldoorn: Kadaster, 2000.

- Sonka, M. et al., *Image processing, analysis and machine vision*. London: Chapman & Hall, 1993, p.422-442.

- Uitermark, H. et al., 'Semantic and Geometric Aspects of Integrating Road networks'. In: A. Vckovski et al. (eds), *Interoperating geographic information system: second international conference, INTEROP '99*. Berlin: Springer-Verlag, 1999, p.177-188.

Datasets:

- *Bridgis GIS starterkit*. Tiel: Bridgis, 2002
- *Geodan Startdata CD*. Amsterdam: Geodan, 2002

Internetsites:

- http://kadata.kadaster.nl
- http://members.home.nl/jtv/postcode3
- http://www.bridgis.nl
- http://www.cgal.org
- http://www.claritas.nl
- http://www.experian.nl
- http://www.funda.nl
- http://www.geodan.nl
- http://www.prizm.nl

Appendix A: municip.C

```
/* municip.C    by Friso Penninga

=========================================================================

STRUCTURE OF MUNICIP.C
======================

1. Loading data
        1.1 Reading friso_postcode.copy into map postcode
        1.2 Reading friso_ouput.copy into map parcels
        1.3 Reading friso_boundary.copy into map boundaries
                1.3.1 Building the map boundaries
                1.3.2 Building a PlanarMap pm
        1.4 Adding parcel attributes to the PlanarMap faces

2. Skeletonization
        2.1 Triangulation of faces
                2.1.1 Building the constrained Delaunay triangulation
                2.1.2 Update point attributes
        2.2 Computing skeleton segments
        2.3 Computing skeleton branches
        2.4 Inserting the skeleton in the planar map

3. Adding postal codes and quality indicator
        3.1 Adding postal codes
        3.2 Calculating quality indicators

4. Creating 6 ppc regions

5. Exporting the planar map
=========================================================================
*/

#include <iterator>
#include <algorithm>
#include <string>
#include <cassert>
#include <map>
#include <list>
#include <vector>
#include <fstream>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

#include "parcel.h"
#include "boundary.h"

#include <CGAL/basic.h>
#include <CGAL/Cartesian.h>
#include <CGAL/Quotient.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/cartesian_homogeneous_conversion.h>
#include <CGAL/Filtered_exact.h>
#include <CGAL/leda_real.h>
#include <CGAL/double.h>

#include <CGAL/Pm_walk_along_line_point_location.h>
#include <CGAL/Pm_segment_traits_2.h>
#include <CGAL/Pm_default_dcel.h>
#include <CGAL/Topological_map_bases.h>
#include <CGAL/Planar_map_2.h>

#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/Constrained_Delaunay_triangulation_2.h>
#include <CGAL/Constrained_triangulation_2.h>
#include <CGAL/Triangulation_2.h>
```

```
class Face_with_id : public CGAL::Pm_face_base
{
   long p_id;
   bool split_parcel;
   bool has_been_splitted;
   std::string postcode;
   int pc_step;
   bool visited_for_removal;
public:
   Face_with_id() :
      CGAL::Pm_face_base(),
      p_id(-1),
      split_parcel(false),
      has_been_splitted(false),
      postcode(),
      pc_step(0),
      visited_for_removal(false)
   {
      cerr << "face_with_id " << p_id << " and postcode (" << postcode << ") created"
        << endl;
   }

   long id() {return p_id; }
   bool split() {return split_parcel; }
   bool splitted() {return has_been_splitted;}
   std::string pc() const {return postcode; }
   int step() {return pc_step; }
   bool visited() {return visited_for_removal; }
   void set_id(long pid) { p_id = pid; }
   void set_split(bool spl) { split_parcel = spl;}
   void set_splitted(bool hbs) { has_been_splitted = hbs; }
   void set_pc(const std::string & c) {
      cerr << "parcel " << p_id << " gets pc (" << c << ")" << endl;
      postcode = c;}
   void set_step(int st) {pc_step = st;}
   void set_visited(bool vis) {visited_for_removal = vis;}
};


template < class Kernel >
class Vertex_with_info : public CGAL::Triangulation_vertex_base_2<Kernel>
{
   bool parcel_point;
public:
   Vertex_with_info() : CGAL::Triangulation_vertex_base_2<Kernel>(),
parcel_point(false) {}

   bool is_parcel_point() { return parcel_point; }
   void set_parcel_point(bool p_p) { parcel_point = p_p;}
};

typedef CGAL::Filtered_exact<double,leda_real>                  Number_type;
typedef CGAL::Cartesian<Number_type>                            KernelCartesian;
typedef CGAL::Homogeneous<Number_type>                          KernelHomogeneous;
typedef CGAL::Point_2<KernelCartesian>                          Point_2;
typedef CGAL::Point_2<KernelHomogeneous>                        H_Point_2;
typedef CGAL::Segment_2<KernelHomogeneous>                      Segment_2;

typedef CGAL::Pm_segment_traits_2<KernelHomogeneous>            PM_Traits;
typedef CGAL::Pm_dcel<CGAL::Pm_vertex_base<PM_Traits::Point>,
                      CGAL::Pm_halfedge_base<PM_Traits::X_curve>,
                      Face_with_id>                             Dcel;
typedef CGAL::Planar_map_2<Dcel,PM_Traits>                      Planar_map;
typedef PM_Traits::X_curve_2                                    X_curve_2;
typedef Planar_map::Vertex_handle                               PM_Vertex_handle;
typedef Planar_map::Face_handle                                 PM_Face_handle;
typedef Planar_map::Halfedge_handle                             PM_Halfedge_handle;
typedef Planar_map::Face                                        PM_Face;
typedef Planar_map::Halfedge                                    PM_Halfedge;

typedef Vertex_with_info<KernelHomogeneous>                     Vb;
typedef CGAL::Constrained_triangulation_face_base_2<KernelHomogeneous>    Fb;
typedef CGAL::Triangulation_data_structure_2<Vb, Fb>        Tds;
typedef CGAL::No_intersection_tag                               Itg;
typedef CGAL::Constrained_Delaunay_triangulation_2<KernelHomogeneous,Tds,Itg>  CDT;
typedef CDT::Constraint                                         Constraint;
```

```
typedef pair<CDT::Face_handle,int>                              CDT_Edge;
typedef CDT::Face_handle                                        CDT_fh;

// translation to origin, only valid for testarea Zevenhuizen
const long XDISPLACEMENT = 100000000;
const long YDISPLACEMENT = 445000000;


/*
*********************************************************
* STEP 1.1 Reading friso_postcode.copy into map postcode *
*********************************************************

What?
=====
Reading friso_postcode.copy. This is a dump from akr_objectadres with two
columns: object_id and postcode. Object_id is not unique, so an object_id
can be linked to multiple postcodes. This part of the script converts this
dump into a map called postcodes where each object_id is linked to a char
with all linked postcodes, seperated by a blank space.

How?
=====
Each line is read from the file, the first token is object_id, the second
temp_short. Temp_short is concetenated to temp_long. Next step: the next
line is read. If object_id is the same as the previous, the new temp_short
is concetenated to temp_long. If not, the temp_long is added to the map with
the previous object_id as key value. Temp_long is cleared and again concetenated
with the new temp_short.

*/
void readPostcodes(
   map<int,char*> &postcodes,
   char* postcodefile)
{
   ifstream postcode_data (postcodefile);
   cout << "Reading " << postcodefile << " ... ";
   long c, p_id, old_id;

   char line[200], temp_short[6], temp_long[70];
   char * ppc;
   c=0;

   while (!postcode_data.eof())
   {
      postcode_data.getline(line, 200);

      if (strlen(line) == 0)
      {
         break;
      }

      ppc = strtok (line," \t");
      p_id = atol (ppc);

      if (c == 0)
      {
        old_id = p_id;
      }

      ppc = strtok (NULL, " \t");
      strcpy(temp_short, ppc);

      if (p_id != old_id)
      {
         postcodes[old_id] = strdup(temp_long);
         strcpy (temp_long, "");
      }

      c=c+1;
      old_id = p_id;

      if ( strlen(temp_long) > 5)
      {
         strcat (temp_long, " ");
      }
```

```
      strcat (temp_long, temp_short);
      strcpy (temp_short, "");
   }

   postcodes [p_id] = strdup(temp_long);

   cout << "   READY" << endl;
}


/*
*****************************************************
* STEP 1.2 Reading friso_ouput.copy into map parcels *
*****************************************************

What?
=====
Reading friso_output.copy. This is a dump from lki_parcel and akr_objectkul.
Columns: object_id (unique), x_akr_objectnummer, l_num, line_id1, line_id2
and split. Split indicates if a parcel can be split, based on its value in
soort_cult or beb_code. '1' indicates to split, '0' not to split. With these
attributes an new object parcel is created. This parcel is stored in a map
called parcels with its object_id as key value.

How?
=====
Each line is read from the file. The first token is the object_id, the second
x_akr_objectnummer an so on. After splitting the line and getting all attributes,
it is tested if the object_id appears in the map postcodes. If so, the string
postcodes is read. If not: postcode is an empty string: "". With this attributes
a new object called parcel is created (see also parcel.C and parcel.h). This
parcel is stored in the map with its object_id as key value.

*/
void readParcels(
   map<int,parcel*> &parcels,
   map<int,char*> &postcodes,
   char* parcelfile)
{
   ifstream parcel_data (parcelfile);
   cout << "Reading " << parcelfile << " ... ";

   int columncounter, l_num, split;
   long parcel_id, line_id1, line_id2;
   char record[900];
   char * pch;

   while (!parcel_data.eof())
   {
      char g_akr_objectnr[18], akr_half1[7], akr_half2[11], postcd[70], location[25];
      parcel_data.getline(record, 900);

      if (strlen(record) == 0)
      {
         break;
      }

      pch = strtok (record," \t");

      columncounter = 0;

#ifdef FRISO_CODE
      while (pch != NULL)
      {
         switch( columncounter)
         {
            case 0: parcel_id = atol (pch);
                    break;
            case 1: strcpy (akr_half1, pch);
                    break;
            case 2: strcpy (akr_half2, pch);
                    g_akr_objectnr[0] = 0;
                    strcat (g_akr_objectnr, akr_half1);
                    strcat (g_akr_objectnr, " ");
                    strcat (g_akr_objectnr, akr_half2);
                    break;
```

```
            case 3: l_num = atoi (pch);
                    break;
            case 4: line_id1 = atol (pch);
                    break;
            case 5: line_id2 = atol (pch);
                    break;
            case 6: split = atoi (pch);
                    break;
            case 7: strcpy (location, pch);
                    break;
            default: cout << "Error in splitting LKI_parcel data" << endl;
                    exit(1);
         }

         pch = strtok (NULL, " \t");
         columncounter++;
      }
#else
      parcel_id = atol (pch);
      pch = strtok (NULL, " \t");
      strcpy (akr_half1, pch);
      pch = strtok (NULL, " \t");
      strcpy (akr_half2, pch);
      g_akr_objectnr[0] = 0;
      strcat (g_akr_objectnr, akr_half1);
      strcat (g_akr_objectnr, " ");
      strcat (g_akr_objectnr, akr_half2);
      pch = strtok (NULL, " \t");
      l_num = atoi (pch);
      pch = strtok (NULL, " \t");
      line_id1 = atol (pch);
      pch = strtok (NULL, " \t");
      line_id2 = atol (pch);
      pch = strtok (NULL, " \t");
      split = atoi (pch);
      pch = strtok (NULL, " \t");
      strcpy (location, pch);
#endif

      if (postcodes[parcel_id] != NULL)
      {
         strcpy (postcd, postcodes[parcel_id]);
      }
      else
      {
         strcpy (postcd, "");
      }
      parcels[parcel_id] = new parcel(parcel_id,g_akr_objectnr,l_num,line_id1,line_id2,
                                      split, postcd, location);
      strcpy (postcd, "");

   }
   cout << "   READY" << endl;
}

/*

************************************************************
* STEP 1.3 Reading friso_boundary.copy into map boundaries *
************************************************************

What?
=====
Read file friso_boundary.copy, a dump from lki_boundary with all boundaries
in the selected municipality. Inserting all boundaries in Planar Map pm.

How?
=====
Read each line, tokens are: line_id, geo_polyline (char[1150], identical to
datatype in lki_boundary), ll_line_id, lr_line_id, fl_line_id, fr_line_id,
l_obj_id, r_obj_id. l_obj_id and r_obj_id are parcel object_id's. With these
attributes a new object called boundary is created. All boundaries are stored
in a map called boundaries with the line_id as key value. All geo_polylines
are split, coordinate by coordinate. Pairs of points are used to create X-curves.
These X-curves are inserted in the Planar Map pm.
```

```
        Topology in lki_bounary:
     ----------------------------

       ll_line_id  lr_line_id
               \   /
                \ /
                /|\
l_obj_id         |        r_obj_id
                 |
                / \
               /   \
      fr_line_id   fl_line_id

all l/f/l/r_line_id's are positive if the line points away from line(line_id) and
negative if they point towards the line(line_id)


****************************************
* STEP 1.3.1 Building the map boundaries *
****************************************

*/
void readBoundary(
   Planar_map& pm,
   char* boundaryfile)
{
      ifstream boundary_data (boundaryfile);
      cout << "Reading " << boundaryfile << " ... " << endl;

      int countboun;
      long line_id, ll_line_id, lr_line_id, fl_line_id, fr_line_id, l_obj_id, r_obj_id;
      char bounline[1500];

      cout << "   Inserting data... " << endl;

      int tl=0;
      while (!boundary_data.eof())
      {
         char geo_polyline[1500];
         boundary_data.getline(bounline, 1500);

         if (strlen(bounline) == 0)
         {
            break;
         }

        char* pboun = strtok (bounline, " \t");

         countboun = 0;

         while (pboun != NULL)
         {
            switch( countboun )
            {
               case 0: line_id = atol (pboun);
                       break;
               case 1: strcpy (geo_polyline, pboun);
                       break;
               case 2: ll_line_id = atol (pboun);
                       break;
               case 3: lr_line_id = atol (pboun);
                       break;
               case 4: fl_line_id = atol (pboun);
                       break;
               case 5: fr_line_id = atol (pboun);
                       break;
               case 6: l_obj_id = atol (pboun);
                       break;
               case 7: r_obj_id = atol (pboun);
                       break;
               default: cout << "Error in splitting boundary data" << endl;
                        exit(1);
            }

            pboun = strtok (NULL, " \t");
            countboun++;
```

```
        }

/*

************************************
* STEP 1.3.2 Building a PlanarMap pm *
************************************

   Splitting the string geo_polyline in tokens. First part is the x-coordinate
   of the first point, second part is the y-cordinate of the first point, the
   third part is the x-coordinate of the second point, etc., etc. CGAL Planar
   Map requires x-curves, defined by a surce and a target point. As a result, the
   polyline is split into seperate points. Point 1 and 2 form an x-curve,
   point 2 and 3, etc. etc.
*/
        long xcoord, ycoord;
        bool firsttime = true, isx = true;
        H_Point_2 prev_point, p;
        X_curve_2 cv;

        char* pch = strtok (geo_polyline,"),(");

        while (pch != NULL)
        {
           if (strlen(pch) > 5)
           {
              if (isx == true)
              {
                 xcoord = atol(pch);
                 isx = false;
              }
              else
              {
                 ycoord = atol(pch);
                 H_Point_2 p(xcoord - XDISPLACEMENT,ycoord - YDISPLACEMENT,1);
                 isx = true;

                 if (firsttime == true)
                 {
                    prev_point = p;
                    firsttime = false;
                 }
                 else
                 {
                    cv = X_curve_2(prev_point,p);
                    tl++;
                    cout << tl << " inserting: " << cv << endl ;

                    pm.insert(cv);
                    prev_point = p;
                 }
              }
           }
           pch = strtok (NULL,"),(");
        }
     }
     cout << "pm.number_of_faces() = " << pm.number_of_faces() << endl;
     cout << "   READY " << endl;
}

/*
************************************************************
* STEP 1.4 Adding parcel attributes to the PlanarMap faces *
************************************************************

What?
=====
Adding additional data to the faces in the planar map. For each parcel in the
map parcels the corresponding face in the Planar map is searched. The parcel_id,
split boolean and postcode are added.

How?
====
Iterating over each entry in the map parcels. Using the location field (from
lki_parcel), the locate-statement is used to find the corresponding face in the
map. With set_id, set_split and set_pc the attributes are added to the face.
```

```
*/
 void addParcelAttributes(
    Planar_map& pm,
    map<int,parcel*> &parcels)
{
   cout << "Start adding parcel attributes..." << endl;

   int sspl;
   bool spl=false;
   long int pid;
   char*  location;
   char* part_p;
   map<int, parcel*>::const_iterator map_iter;

   for ( map_iter = parcels.begin(); map_iter != parcels.end(); map_iter++)
   {
      int countr=0;
      long ins_x, ins_y;
      H_Point_2 ins_point;

      pid = map_iter->first;
      parcel& p= *(map_iter->second) ;



      sspl = p.split;
      if (sspl == 1) spl = true;

      location = strdup(p.location.c_str());
      std::string postcode  = strdup(p.postcode.c_str());
      part_p = strtok (location, "),(");

      while (part_p != NULL)
      {
         if (strlen(part_p) > 5)
         {
            if (countr == 0)
            {
               ins_x = atol( part_p );
               countr++;
            }
            else
            {
               ins_y = atol( part_p );
               ins_point = H_Point_2
                              ( ins_x - XDISPLACEMENT , ins_y - YDISPLACEMENT, 1 );
               countr++;
            }
         }

         part_p = strtok (NULL, "),(");
      }
      Planar_map::Locate_type lt;
      PM_Face_handle f = pm.locate(ins_point,lt)->face();
      if (lt == Planar_map::UNBOUNDED_FACE)
      {
         cerr << "point is located in unbounded face" << endl;
      }
      else if (lt == Planar_map::VERTEX)
      {
         cerr << "point is located on vertex" << endl;
      }
      else if (lt != Planar_map::FACE)
      {
         cerr << "point is not located in face" << endl;
      }
      else
      {
         if (f->id() != -1)
         {
            cerr << "ERROR: face " << f->id() << " already has a zipcode" << endl;
         }

         f->set_id(pid);
         f->set_split(spl);
```

```
         if (spl == true)
         {
            f->set_pc("");
            spl = false;
         }
         else
         {
            f->set_pc(postcode);
         }
      }
   }

   cout << "   READY" << endl;
}

/*

*********************************************************************
*                STEP 2     SKELETONIZATION                         *
*********************************************************************


**********************************
* STEP 2.1 Triangulation of faces *
**********************************

What?
=====
Triangulating each face that has to be skeletonized.

How?
====
A face iterator iterates over each face in the planar map pm. Each face where
to_be_split is true and been_splitted (which indicates if a face is a result of
earlier skeletonization) is false, is being skeletonized. The boundaries of this
face are added as constraint edges to an empty constrained Delaunay triangulation.
To do this, the outer boundary is circulated and each x-curve is added to the CDT
as constrained edge. A hole iterator iterates over each hole in the face and returns
a circulator over the boundary of the hole. Each x-curve is again being added to
the CDT as constrained edge.


************************************************************
* STEP 2.1.1 Building the constrained Delaunay triangulation *
************************************************************

*/

/*

**************************************************
* STEP 2.4 Inserting the skeleton in the planar map *
**************************************************

What?
=====
Adding the skeleton to the Planar map


How?
====
The Point_map with the points of the skeleton is read. Each two points form a
skeleton segment. With these two points an x-curve is build and these x-curves
are added to the planar map. To each new face the old parcel_id is added and the
boolean has_been_splitted is switched to true

*/
void addEdges(
   Planar_map& pm,
   vector<X_curve_2> edges,
   int parcel_id,
   bool to_be_split)
{
   cout << "adding edges to pm" << endl;
```

```
   for (unsigned int j = 0; j < edges.size(); j++)
   {
      cout << "inserting curve " << edges[j] << endl;
      PM_Halfedge_handle e = pm.insert(edges[j]);

      PM_Face_handle f = e->face();
      PM_Face_handle ft = e->twin()->face();

      f->set_id(parcel_id);
      f->set_split(to_be_split);
      f->set_pc("");
      f->set_splitted(true);

      ft->set_id(parcel_id);
      ft->set_split(to_be_split);
      ft->set_pc("");
      ft->set_splitted(true);
   }
   cout << "done" << endl;
}

void createCDT(
   CDT &cdt,
   Planar_map::Face& face)
{
   PM_Traits pm_traits;
   Planar_map::Ccb_halfedge_circulator h_circ = face.outer_ccb();
   cdt.clear();

   //
   // circulate through all edges of the boundary of the parcel and add
   // them to the CDT.
   //
   do
   {
      X_curve_2 cur = h_circ->curve();

      H_Point_2 hr =  pm_traits.curve_source(cur);
      H_Point_2 hs =  pm_traits.curve_target(cur);
      cout << "inserting " << hr << " " << hs << " in cdt";
      cdt.insert(hr,hs);
      cout << "." << endl;
   } while (++h_circ != face.outer_ccb());
   cout << "outer ccb inserted in cdt" << endl;

   //
   // circulate through all edges of all holes of the parcel and add
   // them to the CDT.
   //
   Planar_map::Holes_iterator f_holes_iter;
   for ( f_holes_iter = face.holes_begin(); f_holes_iter != face.holes_end();
                        f_holes_iter++)
   {
      Planar_map::Ccb_halfedge_circulator island_circulator;

      island_circulator = *f_holes_iter;
      do
      {
         X_curve_2 cur = island_circulator->curve();
         H_Point_2 hr =  pm_traits.curve_source(cur);
         H_Point_2 hs =  pm_traits.curve_target(cur);
         cdt.insert(hr,hs);
         //cout << cdt.is_valid();
      }
      while (++island_circulator != *f_holes_iter);
      cout << "inner ccb's inserted in cdt" << endl;
   }
}

void skeletonize(Planar_map& pm)
{
   cout << "Start skeletonization... " << endl;

   vector<X_curve_2> edge_vector;

   Planar_map::Face_iterator pm_f_iter;
```

```
   for (pm_f_iter=pm.faces_begin(); pm_f_iter != pm.faces_end(); pm_f_iter++ )
   {
      long parcel_id = pm_f_iter->id();
      bool to_be_split = pm_f_iter->split();
      bool been_splitted = pm_f_iter->splitted();
      cerr <<"working on parcel: id=" << parcel_id <<
             " to_be_split=" << to_be_split <<
             " been_splitted=" << been_splitted << endl;

      CDT cdt;

      if (( to_be_split == true ) && (been_splitted == false))
      {
         cout << "Parcel " << parcel_id << " has to be skeletonized" << endl;
         createCDT(cdt,*pm_f_iter);


//         assert(cdt.is_valid());

/*

************************************
* STEP 2.1.2 Update point attributes *
************************************

To each point in the CDT an indicator is added, which indicates if the point
needs to be connected to the skeleton in a later phase. This is the case when
the point represents a point in the planar map where at least three parcels
border on one another. To check this, for each point in the CDT its corresponding
point in the planar map is calculated. An incident halfedge circulator is used
to count the number of edges (=borders) the point is part of. If this number
equals or exceeds three, the point has to be connected to the skeleton in a
later phase.

*/

         CDT::Finite_vertices_iterator cdt_v_iter1;
         for (cdt_v_iter1=cdt.finite_vertices_begin();
                         cdt_v_iter1!=cdt.finite_vertices_end();
                         cdt_v_iter1++ )
         {
            Planar_map::Locate_type lt;
            PM_Halfedge_handle hh = pm.locate(cdt_v_iter1->point(), lt);
            PM_Vertex_handle p_v = hh->target();

            int no_of_boundaries=0;
            Planar_map::Halfedge_around_vertex_circulator
               hav_circ = p_v->incident_halfedges();
            do
            {
               no_of_boundaries++;
            } while ( ++hav_circ != p_v->incident_halfedges());

            if (no_of_boundaries >= 3)
            {
               cdt_v_iter1->set_parcel_point(true);
            }
         }

/*

**************************************
* STEP 2.2 Computing skeleton segments *
**************************************

What?
=====
Computing the skeleton of the triangulated face

How?
====
Now the CDT is complete, an iterator over each finite triangle is defined. If this
triangle is inside the face that has to be skeletonized, the number of constraint
edges in the triangle is counted. According to the rules of Black and DeLucia the
parts of the skeleton are computed:
```

```
-  2 comstrained egdes: the midpoint of the non-contrained edge is connected with
              the vertex opposite to this edge.
-  1 constrained edge : the midpoints of the two non-constrained edges are connected
-  0 constrained edge : the midpoints of the three non-constrained edges are connected
              with the midpoint of the triangle

*/



        edge_vector.clear();


        CDT::Finite_vertices_iterator cdt_v_iter;
        for (cdt_v_iter=cdt.finite_vertices_begin();
             cdt_v_iter!=cdt.finite_vertices_end();
             cdt_v_iter++ )
        {
           bool add_point_to_skeleton;
           add_point_to_skeleton = cdt_v_iter->is_parcel_point();
           if (add_point_to_skeleton == true)
           {
              bool ready = false;
              CDT::Edge_circulator v_e_circ;
              v_e_circ = cdt.incident_edges(cdt_v_iter);

              while (ready == false)
              {
                 bool ed_constr, ed_infinite;
                 ed_constr = cdt.is_constrained(*v_e_circ);
                 ed_infinite = cdt.is_infinite(*v_e_circ);
                 if ((ed_constr == false) && (ed_infinite == false))
                 {

                    H_Point_2 hseg_middle;
                    Segment_2 seg;
                    seg = cdt.segment(v_e_circ);
                    hseg_middle = midpoint( seg.source() , seg.target() );

                    bool seg_m_isinface = pm.is_point_in_face( hseg_middle, pm_f_iter);

                    if (seg_m_isinface == true)
                    {
                       edge_vector.push_back(X_curve_2
                                             (hseg_middle,cdt_v_iter->point()));
                       ready = true;
                    }
                 }
                 v_e_circ++;
                 if (v_e_circ == cdt.incident_edges(cdt_v_iter)) ready = true;
              }
           }
        }



        CDT::Finite_faces_iterator cdt_f_iter;
        for (cdt_f_iter=cdt.finite_faces_begin();
             cdt_f_iter!=cdt.finite_faces_end();
             cdt_f_iter++ )
        {
           H_Point_2 p0 = cdt_f_iter->vertex(0)->point();
           H_Point_2 p1 = cdt_f_iter->vertex(1)->point();
           H_Point_2 p2 = cdt_f_iter->vertex(2)->point();

           //
           // check whether an internal point of the triangle is inside the
           // face. If this is not the case we can skip this triangle.
           //
           if (pm.is_point_in_face(midpoint(midpoint(p0,p1), p2), pm_f_iter))
           {
              int nr_g_edges=0, v_id=-1, pp=-1;

              for (int edgenr=0; edgenr <=2; edgenr++)
              {
                 CDT_Edge cur_edge(cdt_f_iter,edgenr);
```

```
                    if (cdt.is_constrained(cur_edge))
                    {
                       nr_g_edges++;
                    }
                    else
                    {
                       if (edgenr == 0)
                       {
                          pp = 0;
                       }
                       v_id = edgenr;
                    }
                 }

                 H_Point_2 mp0 = midpoint(p1,p2);
                 H_Point_2 mp1 = midpoint(p0,p2);
                 H_Point_2 mp2 = midpoint(p0,p1);

                 if (nr_g_edges == 0)
                 {
                    H_Point_2 centerpoint = H_Point_2(
                      to_double(p0.hx()) + to_double(p1.hx()) + to_double(p2.hx())  ,
                      to_double(p0.hy()) + to_double(p1.hy()) + to_double(p2.hy())  ,
                      p0.hw() * p1.hw() * p2.hw() * 3
                    );

                    edge_vector.push_back(X_curve_2(centerpoint,mp0));
                    edge_vector.push_back(X_curve_2(centerpoint,mp1));
                    edge_vector.push_back(X_curve_2(centerpoint,mp2));
                 }
                 else if (nr_g_edges == 1)
                 {
                   if (v_id == 1)
                   {
                      edge_vector.push_back(X_curve_2(mp0,mp1));
                   }
                   else
                   {
                      if (pp == 0)
                      {
                         edge_vector.push_back(X_curve_2(mp0,mp2));
                      }
                      else
                      {
                         edge_vector.push_back(X_curve_2(mp1,mp2));
                      }
                   }
                 }
                 else if (nr_g_edges == 2)
                 {
                   H_Point_2 ep = cdt_f_iter->vertex(v_id)->point();
                   H_Point_2 otherp;
                   switch (v_id)
                   {
                      case 0: otherp = mp0;
                              break;
                      case 1: otherp = mp1;
                              break;
                      case 2: otherp = mp2;
                              break;
                   }
                   edge_vector.push_back(X_curve_2(ep,otherp));
                 }
                 else if (nr_g_edges == 3)
                 {
                    cerr << "found triangular face" << endl;
                 }
                 else
                    cerr << "unexpeted number of g_edges " << endl;

              }
           }

/*
```

```
*************************************
* STEP 2.3 Computing skeleton branches *
*************************************

What?
=====
Computing skeleton branches

How?
====
For each point that has to be added to the skeleton (determined in step 2.1.2) a
skeleton segment is calculated. A halfedge circulator around the point is used to
find an internal non-constrained edge. The point is connected with the midpoint of
this non-constrained edge. If there are multiple internal non-constrained edges, the
first visited edge is used.

*/

         addEdges(pm,edge_vector,parcel_id,to_be_split);
      }
   }
   cout << "pm.number_of_faces() = " << pm.number_of_faces() << endl;
   cout << "   READY" << endl;
}

/*
*****************************************************************
*      STEP 3     ADDING POSTAL CODES AND QUALITY INDICATOR     *
*****************************************************************

*******************************
* STEP 3.1 Adding postal codes *
*******************************

What?
=====
For each parcel without postal code the most likely postal code is searched.
The most likely postal code is the neighboring postal code which shares the
longest common boundary (or sum of common boundaries).

How?
====
As long as parcels exist without postal code, these parcels without postal code
are visited. For each face all edges (inner and outer boundaries) are visited.
For these edges the neighboring face and its postal code are found. The length
of the edge is calculated and stored together with the postal code in a map.
If the postal code exists already in the map, the two lengths are added. The
postal code with the largest sum of lengths is added to the
*/
void addPostcodes(Planar_map& pm)
{
   cout << "Start adding most likely postal codes..." << endl;
   PM_Traits pm_traits;
   int step_count = 0;
   bool all_parcels_have_pc;
   do
   {
      bool assigned_pc = false;
      ++step_count ;
      cerr << "doing step " << step_count << endl;
      all_parcels_have_pc = true;
      Planar_map::Face_iterator pm_f_iter;
      for (pm_f_iter=pm.faces_begin(); pm_f_iter != pm.faces_end(); pm_f_iter++ )
      {
         bool unb;
         unb = pm_f_iter->is_unbounded();

         if (( unb == false) && ( pm_f_iter->pc() == "" ))
         {
            map < std::string, double> pc_length_map;

            Planar_map::Ccb_halfedge_circulator h_circ = pm_f_iter->outer_ccb();
            do
            {
               double current_length;
               Point_2 curv_source, curv_target;
```

```
   H_Point_2 hcurv_source, hcurv_target;
   X_curve_2 curv = h_circ->curve();

   hcurv_source = pm_traits.curve_source(curv);
   hcurv_target = pm_traits.curve_target(curv);
   curv_source = homogeneous_to_cartesian( hcurv_source );
   curv_target = homogeneous_to_cartesian( hcurv_target );

   double xdiff = curv_source.x().value() - curv_target.x().value()  ;
   double ydiff = curv_source.y().value() - curv_target.y().value()  ;
   double edge_length = sqrt((xdiff * xdiff) + (ydiff * ydiff));
   current_length = pc_length_map[h_circ->twin()->face()->pc()] ;


   if ((h_circ->twin()->face()->pc()) != "")
   {
      if ((h_circ->twin()->face()->step()) == step_count)
      {
         current_length = pc_length_map["0"] ;
         pc_length_map["0"] = current_length + edge_length;
      }
      else
      {
         pc_length_map[h_circ->twin()->face()->pc()] =
            current_length + edge_length;
      }
   }
} while ( ++h_circ != pm_f_iter->outer_ccb() );

Planar_map::Holes_iterator f_holes_iter;

for ( f_holes_iter = pm_f_iter->holes_begin();
     f_holes_iter != pm_f_iter->holes_end(); f_holes_iter++)
{
   Planar_map::Ccb_halfedge_circulator island_circulator;
   island_circulator = *f_holes_iter;
   X_curve_2 first_cur = island_circulator->curve();

   do
   {
      double current_length;
      Point_2 curv_source, curv_target;
      H_Point_2 hcurv_source, hcurv_target;
      X_curve_2 curv = island_circulator->curve();

      hcurv_source = pm_traits.curve_source(curv);
      hcurv_target = pm_traits.curve_target(curv);
      curv_source = homogeneous_to_cartesian( hcurv_source );
      curv_target = homogeneous_to_cartesian( hcurv_target );

      double xdiff = curv_source.x().value() - curv_target.x().value()  ;
      double ydiff = curv_source.y().value() - curv_target.y().value()  ;
      double edge_length = sqrt((xdiff * xdiff) + (ydiff * ydiff));
      current_length =
         pc_length_map[island_circulator->twin()->face()->pc()] ;
      if ((island_circulator->twin()->face()->pc()) != "")
      {
         if ((island_circulator->twin()->face()->step()) == step_count)
         {
            current_length = pc_length_map["0"] ;
            pc_length_map["0"] = current_length + edge_length;
         }
         else
         {
            pc_length_map[island_circulator->twin()->face()->pc()] =
               current_length + edge_length;
         }
      }
   } while ((++island_circulator)->curve() !=  first_cur);
}

double sum_length, largest_sum = 0;
std::string most_likely_pc, this_pc;
map < std::string, double>::const_iterator pc_map_iter;

for ( pc_map_iter = pc_length_map.begin();
```

```
                pc_map_iter != pc_length_map.end();
                pc_map_iter++ )
            {
               sum_length = pc_map_iter->second;
               this_pc = pc_map_iter->first;

               if (( sum_length > largest_sum) && ( this_pc != "0"))
               {
                  largest_sum = sum_length;
                  most_likely_pc = this_pc;
               }
            }
            if (largest_sum > 0)
            {
               assigned_pc = true;
               pm_f_iter->set_pc(most_likely_pc);
               pm_f_iter->set_step(step_count);
            }
            else
            {
               all_parcels_have_pc = false;
            }
         }
      }

      if (!assigned_pc)
      {
         cerr << "NO POSTCODES ASSIGNED" << endl;
         break;
      }
   } while ( all_parcels_have_pc == false);
   cout << "   READY" << endl;
}

/*
***********************************
* STEP 3.2 Adding quality indicator *
***********************************

What?
=====
Adding a quality indicator which reflects the reliability of the
postal code. Basic idea is that as the number of iterations (to
add a postal code) increases, the reliability drops.

How?
====
An iterator visists every parcel. Postal code and number of steps are
derived. In a map, organized by postal code, the sum of fractions is
keeped:

1 / square root ( number of steps + 1 )

In a second map the number of parcels with a certain postal code is
registered. After this iteration the quality indicator is being
calculated by deviding the sum of fractions by the number of parcels
for each postal code. These quality indicators are stored in a map.
These indicators are not added to the parcels but added in the
exportfile in step 5.

*/
void addQualityIndicator(
   Planar_map& pm,
   map<std::string,double>& quality
   )
{
   cout << "Start computing quality indicators..." << endl;

   map < std::string, double > no_of_parcels;
   map < std::string, double > sum_of_fractions;

   Planar_map::Face_iterator pm_f_iter;
   for (pm_f_iter=pm.faces_begin(); pm_f_iter != pm.faces_end(); pm_f_iter++ )
   {
      double no_so_far, sum_so_far, new_sum;
      no_so_far = no_of_parcels[pm_f_iter->pc()];
```

```
      sum_so_far = sum_of_fractions[pm_f_iter->pc()];
      new_sum = (1 / sqrt(( (pm_f_iter->step()) + 1))) + sum_so_far;
      no_of_parcels[pm_f_iter->pc()] = no_so_far + 1;
      sum_of_fractions[pm_f_iter->pc()] = new_sum;
   }
   map < std::string, double >::const_iterator no_iter;

   for ( no_iter = no_of_parcels.begin(); no_iter != no_of_parcels.end(); no_iter++ )
   {
      quality[no_iter->first] =
         (floor(((sum_of_fractions[ no_iter->first ]) / (no_iter->second))*1000))/1000;
   }
   cout << "   READY" << endl;
}

/*
*****************************************************************
*                  STEP 4     CREATE 6 PPC AREAS                 *
*****************************************************************
*/
void createPpcAreas( Planar_map& pm)
{
   cout << "Start creating 6 ppc areas..." << endl;

   Planar_map::Face_iterator faces;
   Planar_map::Ccb_halfedge_circulator h_circ2;
   vector<Planar_map::Ccb_halfedge_circulator> edges;

   for (faces = pm.faces_begin(); faces != pm.faces_end(); faces++)
   {
      if (!faces->is_unbounded())
      {
         faces->set_visited(true);

         h_circ2 = faces->outer_ccb();
         do
         {
            if (((h_circ2->twin()->face()->pc()) == (faces->pc())) &&
                 ((h_circ2->twin()->face()->visited()) == false))
            {
               edges.push_back(h_circ2);
            }
         } while ( ++h_circ2 != faces->outer_ccb() );

         Planar_map::Holes_iterator f_holes_iter2;

         for ( f_holes_iter2 = faces->holes_begin();
               f_holes_iter2 != faces->holes_end();
               f_holes_iter2++)
         {
            Planar_map::Ccb_halfedge_circulator island_circulator2;
            island_circulator2 = *f_holes_iter2;
            X_curve_2 first_cur = island_circulator2->curve();

            do
            {
               if (( (island_circulator2->twin()->face()->pc()) == (faces->pc()) ) &&
                        ( (island_circulator2->twin()->face()->visited()) == false))
               {
                  edges.push_back(island_circulator2);
               }
            } while ( (++island_circulator2)->curve() != first_cur);
         }
      }
   }

   cout << "going to delete " << edges.size() << "edges" << endl;
   for (unsigned int i=0; i < edges.size(); i++)
   {
      cout << "deleting edges[" << i << "/" << edges.size() << "] :"
           << edges[i]->curve() << endl;
      pm.remove_edge(edges[i]);
   }

   cout << "   READY" << endl;
}
```

```
/*

************************************************************
*         STEP 5   EXPORTING THE PLANAR MAP                *
************************************************************

What?
=====
Exporting the planar map.

How?
====
An exportfile is created. An iterator iterates over all faces. For each bounded
face its parcel_id, is_splitted and postcode are exported to the tab-seperated
ASCII-file. A circulator visits all boundary x-curves and transforms the boundary
to a polygon in OpenGIS Well Known Text-format.

*/
static inline void exportPoint(
   ofstream& exportfile,
   const H_Point_2& hp)
{
   Point_2 p = homogeneous_to_cartesian ( hp );
   exportfile << p.x() + XDISPLACEMENT  << " " << p.y() + YDISPLACEMENT;
}

void exportRing(
   ofstream& exportfile,
   Planar_map::Ccb_halfedge_circulator startedge)
{
   H_Point_2 startpoint = startedge->source()->point();
   exportfile << "(";

   Planar_map::Ccb_halfedge_circulator h_circ = startedge;
   do
   {
      exportPoint(exportfile,h_circ->source()->point());
      exportfile << ",";

   } while (++h_circ != startedge);

   exportPoint(exportfile,startpoint);

   exportfile << ")";
}


void exportPlanarmap(
   Planar_map& pm,
   map <std::string,double>& qualitymap,
   char* filename)
{
   cout << "Start exporting..." << endl;
   cout << "pm.number_of_faces() = " << pm.number_of_faces() << endl;

   ofstream exportfile(filename);
   exportfile.precision(12);

   //
   // Loop over all faces of the Planar Map.
   //
   Planar_map::Face_iterator pm_f_iter;
   for (pm_f_iter=pm.faces_begin(); pm_f_iter != pm.faces_end(); pm_f_iter++)
   {
      cerr << "working on face with id " << pm_f_iter->id() << endl;

      if (!pm_f_iter->is_unbounded())
      {
         cerr << "exporting face with id " << pm_f_iter->id() << endl;
         double  quality = qualitymap[(pm_f_iter->pc())];



         exportfile << pm_f_iter->id() << "\t"
```

```
                    << pm_f_iter->splitted() << "\t"
                    << pm_f_iter->pc() << "\t"
                    << pm_f_iter->split() << "\t"
                    << pm_f_iter->step() << "\t"
                    << quality << "\tPOLYGON(";

         exportRing(exportfile,pm_f_iter->outer_ccb());

         Planar_map::Holes_iterator f_holes_iter;

         for ( f_holes_iter = pm_f_iter->holes_begin();
               f_holes_iter != pm_f_iter->holes_end();
               f_holes_iter++)
         {
            exportfile << ",";
            Planar_map::Ccb_halfedge_circulator island_circulator;
            island_circulator = *f_holes_iter;
            exportRing(exportfile,island_circulator);
         }
         exportfile << ")" << endl;
         cout << "   READY" << endl;

      }
      else
      {
         cerr << "skipping unbounded face with id " << pm_f_iter->id() << endl;
      }
   }
}


void exportEdges(
   Planar_map& pm,
   char* filename)
{
   cout << "Start exporting edges..." << endl;

   ofstream exportfile(filename);
   exportfile.precision(12);

   //
   // Loop over all faces of the Planar Map.
   //
   Planar_map::Halfedge_iterator edge;
   for (edge=pm.halfedges_begin(); edge != pm.halfedges_end(); edge++)
   {
      //
      // The iterator goes over every halfedge and we want all edges once,
      // so we skip all edges where face()->id() <= twin()->face()->id().
      //
      if (edge->face()->id() > edge->twin()->face()->id())
      {
         exportfile << "LINESTRING (";
         exportPoint(exportfile,edge->source()->point());
         exportfile << " , ";
         exportPoint(exportfile,edge->target()->point());
         exportfile <<
            ")\t" <<
            edge->face()->id() <<
            "\t" <<
            edge->twin()->face()->id() <<
            "\t" <<
            edge->face()->id() << "-" <<
            edge->twin()->face()->id() <<
            endl;
      }
   }
   cout << "End exporting edges..." << endl;
}




/*
```

```
*****************************************************************
*               STEP 1     LOADING DATA                         *
*****************************************************************
*/



int main(int argc, char* argv[])
{
   char* postcodefile = 0;
   char* parcelfile = 0;
   char* boundaryfile = 0;

   if(argc == 4)
   {
      postcodefile = argv[1];
      parcelfile = argv[2];
      boundaryfile = argv[3];
   }
   else
   {
      cerr << "usage: " << argv[0] << " postcodefile parcelfile boundaryfile" << endl;
      exit(1);
   }

   cout << setiosflags( ios :: fixed );
   setiosflags( ios :: fixed );


   map<int,char*> postcodes;
   map<int,parcel*> parcels;
   Planar_map pm;

   readPostcodes(postcodes,postcodefile);
   readParcels(parcels,postcodes,parcelfile);
   readBoundary(pm,boundaryfile);
   addParcelAttributes(pm,parcels);
   skeletonize(pm);
   addPostcodes(pm);

   //
   // This map goes from postcode to quality.
   //
   map < std::string, double > quality;

   addQualityIndicator(pm,quality);
   createPpcAreas(pm);
   exportPlanarmap(pm,quality,"data/PlanarMap.wkt");
   exportEdges(pm,"data/edges.txt");

   cout << endl <<" ---" << endl
        << "Program ended normally. Results are in ../ data / PlanarMap.wkt" << endl
         << "---" << endl;
   return 0;
}
```

Appendix B: ervaringen met CGAL

Bij aanvang van het afstudeerproject is bewust gekozen voor implementatie binnen C++ en CGAL om zo te kunnen beoordelen of CGAL een omgeving is waar in de toekomst meer mee gedaan zou moeten worden. Mijn houding ten opzichte van CGAL is gedurende het onderzoek heel wisselend geweest en nog steeds sta ik ambivalent tegenover het gebruik van CGAL. Uiteindelijk heb ik binnen CGAL een oervorm werkend kunnen krijgen van mijn algoritme, waarin uitgebreid gebruik is gemaakt van de mogelijkheden van CGAL. In dat opzicht ben ik tevreden over CGAL, maar tegelijkertijd hebben een aantal CGAL-gerelateerde problemen mijn onderzoek ernstig vertraagd. Door die vertraging zijn uiteindelijk ook nauwelijks verbeteringen op de oervorm van het algoritme geïmplementeerd.

Om enigzins gestructureerd een aantal voor- en nadelen van CGAL te kunnen bespreken, heb ik mijn commentaar in vier punten onderverdeeld:

Flexibiliteit: zowel een plus- als minpunt

De flexibiliteit van CGAL komt eigenlijk op twee manieren tot uiting, namelijk in de gelaagde opzet van CGAL zelf en in de mogelijkheden tot aanpassing en uitbreiding van in CGAL gedefinieerde klasses. De flexibiliteit in de opzet zorgt er voor dat alle klasses uit de basic library kunnen werken met de verschillende kernels. Daarnaast is de afhankelijkheid tussen klasses in de basic library zo klein mogelijk gehouden. Vanuit ontwerpoogpunt is dit erg mooi, maar sommige binnen GIS gangbare bewerkingen zijn hierdoor echter niet mogelijk. Zo zou het voor de implementatie van mijn algoritme prettig zijn als ik direct een Planar Map (of een andere datastructuur) zou kunnen trianguleren. Door de onafhankelijkheid van de verschillende klasses in de basic library was ik gedwongen om eerst de Planar Map edge gewijs te bevragen en deze edges stuk voor stuk in de triangulatie in te voegen. Dit is natuurlijk niet onoverkomelijk, maar dit is in sommige GIS-pakketten wel beter (eenvoudiger) geregeld. Oorzaak ligt in de beoogde brede range van toepassingsgebieden van CGAL, dit in tegenstelling tot de commerciële GIS-pakketten. De mogelijkheid om klasses zelf uit te breiden is een groot voordeel van CGAL. Ik heb hier zowel binnen de Planar Map als binnen de constrained Delaunay triangulatie gebruik van gemaakt om zo extra attributen aan faces en vertices te kunnen koppelen.

Precisieproblemen: belangrijkste oorzaak van mijn vertraging

De grootste problemen bleken veroorzaakt te worden door precisieproblemen. In mijn geval werkte het invoegen van skeletten in de Planar Map vaak niet. Soms bleef het programma hangen, soms crashte het. Aangezien in deze periode alle begeleiders in het buitenland verbleven, heb ik een eenvoudig voorbeeldje gepost op de CGAL Mailinglist. Hierop kreeg ik meerdere reacties, die allen wezen in de richting van precisieproblemen. Indertijd gebruikte ik nog doubles in een cartesiaans coördinatenstelsel. Dit werd me van verschillende kanten afgeraden in verband met de risico's van problemen met afrondingen. De voorgestelde oplossing was om over te stappen op homogene coördinaten en op leda_reals binnen een filtered_exact als numbertype. Dit bleek voor het eenvoudige voorbeeldje de oplossing, maar voor het algoritme zelf niet. Hierbij bleef het invoegen van skeletsegmenten problematisch. Waarschijnlijk zijn twee skeletsegmenten met een gezamenlijke vertex elkaar gaan snijden doordat door kleine afrondingsverschillen het gezamenlijke punt als twee

verschillende punten werd gezien. Deze precisieproblemen zouden theoretisch echter niet kunnen voorkomen en dat is een nadeel van CGAL. Toen uit pure balorigheid van alle coördinaten de decimalen werden verwijderd, bleek een deel van de problemen verholpen. Uiteindelijk is om het probleem heen gewerkt door de RD-coördinaten (tot tien cijfers voor de komma) te transleren naar de oorsprong om zo met kleinere getallen te kunnen werken.

Leren van CGAL

De leercurve bij het leren van CGAL is zeer steil, zeker als CGAL aangeleerd wordt door direct met het eigenlijke probleem aan de slag te gaan op basis van één van de met CGAL meegeleverde voorbeeldscripts. Deze scripts verduidelijken de mogelijkheden van de verschillende klasses en maken de keuze voor de goede klasse eenvoudiger. De verschillende CGAL manuals zijn pas bruikbaar als een bepaalde hoeveelheid ervaring met CGAL is opgedaan, tot die tijd zijn ze te beperkt. Bovendien is de hoeveelheid uitleg bij bepaalde klasses of functies wel zeer summier. Vaak is de enige mogelijkheid dan om zelf te proberen om de CGAL sourcecode te doorgronden. Doordat deze code door zeer ervaren C++-programmeurs is geschreven en slechts zelden van commentaar is voorzien, is het voor onervaren C++-programmeurs vaak moeilijk om de mogelijkheden en werking van functies te doorgronden. De gemiddelde programmeerkennis van een student geodesie in de eindfase van de opleiding is hiervoor eigenlijk te beperkt.

CGAL + Oracle = GIS ?

Dit is eigenlijk de belangrijkste vraag: is de combinatie van CGAL en Oracle niet gewoon een minder gebruiksvriendelijke variant van een GIS-pakket? Zijn er binnen het algoritme bewerkingen uitgevoerd die niet in een commercieel GIS-pakket hadden kunnen worden uitgevoerd? Ik heb op dit moment het idee dat dit niet zo is. Dit onderzoek had ook kunnen worden uitgevoerd binnen een GIS-pakket, als binnen dit pakket de constrained Delaunay triangulatie en een Planar Map-achtige datastructuur beschikbaar zijn. Als deze datastructuren zelf geïmplementeerd zouden moeten worden, is CGAL wel een aantrekkelijkere optie.

Concluderend kan gesteld worden dat CGAL weliswaar een aantal fraaie eigenschappen heeft, maar dat voor veel toepassingen volstaan kan worden met het gebruik van een standaard GIS-pakket. Deze omgeving is gebruiksvriendelijker en vereist geen gedegen C++-kennis, waarmee de keuze om met een GIS-pakket te werken zeker voor studenten geodesie aantrekkelijker is. Zeker nu steeds meer GIS-pakketten direct met een database kunnen communiceren, zie ik de noodzaak voor het gebruik van CGAL niet in.