Extraktion relevanter Bildkanten für die Gebäudedetektion (umgesetzt in MATLAB)

Kerstin Siemes

TR-IGG-P-2007-05

15. Oktober 2007

Technical Report Nr. 6, 2007

Department of Photogrammetry
Institute of Geodesy and Geoinformation
University of Bonn

Available at
http://www.ipb.uni-bonn.de/papers/

Extraktion relevanter Bildkanten für die Gebäudedetektion (umgesetzt in MATLAB)

Kerstin Siemes

Zusammenfassung

Um Gebäude in Bildern detektieren detektieren zu können, greifen wir als ein Merkmal auf Bildkanten zurück. Kanten liegen zum einen aus der Bildsegmentierung vor, können aber auch gezielt mit einer Kantenextraktion aus dem quadratisch Gradientenbild gewonnen werden. Durch die Auswahl von Kanten, die in beiden Fällen auftreten, wollen wir eine Einschränkung auf möglichst relevante Kanten vornehmen. Diese Arbeit beschäftigt sich mit dem Auffinden (und der Gewichtung) von Kantenzügen eines segmentierten Bildes, die einer Kante im quadratidschen Gradientenbild zugeordnet werden können.

1 Einleitung

Das langfristige Ziel ist, Gebäude in Bildern anhand deren Struktur automatisch zu detektieren und zu extrahieren.
Um Bildstrukturen effizient verwalten zu können, wurde bereits in einer vorangehenden Arbeit [Her07] eine Datenbank erstellt, die diese Aufgabe übernimmt. Die vorhandenen Strukturen sollen analysiert werden, um geeignete Merkmale zu finden, die ein Gebäude auszeichnen. Die Überlegung, dass der Mensch Gebäude anhand von langen parallelen oder orthogonalen Kanten erkennen kann, führt in dieser Arbeit zunächst auf die Untersuchung von Bildkanten. Jedoch sind nicht alle Kanten gleichermaßen für die Gebäudedetektion geeignet. Wir werden im Folgenden eine mögliche Methode vorstellen, mit Hilfe derer solche Kanten, die wir als geeignet erachten, (aus der oben erwähnten Datenbank heraus) bestimmt werden können.

2 Kantenextraktion

Wir unterscheiden zwischen Algorithmen, die ein Bild vollständig in Segmente zerlegen und dadurch auch ein Netz von Bildkantenstücken liefern und solchen, die lediglich einzelne Kanten detektieren. Zur Bestimmung geeigneter Kanten für die Gebäudedetektion, wollen wir beide Methoden kombinieren.
Eine vollständige Partitionierung eines Bildes liegt uns bereits aus vorangehenden Arbeiten vor (siehe beispielsweise [DSF06]). Sie erfolgte bislang mit Hilfe eines Wasserscheidenalgorithmus. Die dabei entstandenen Knoten, Kanten und Maschen wurden in Form einer 'Winged Edge' Struktur in einer Datenbank abgelegt. Bei dieser Struktur werden zu jeder Kante deren adjazente Maschen und Knoten gespeichert sowie ihre Vorgänger- und Nachfolgerkante.
Wir wollen uns nun nicht mehr auf einen speziellen Segmentierungsalgorithmus beschränken, sondern Kanten aus einem beliebigen vollständig segmentierten Bild gewinnen. Daher werden wir für diesen allgemeinen Fall in Kapitel 2.1 noch einmal kurz auf die Kantengewinnung eingehen.
Zur Detektion einzelner Kanten bedienen wir uns eines in der Photogrammetrie der Universität Bonn entwickelten Programms mit dem Namen 'Feature Operators' (FOP). Dieses soll in Kapitel 2.2 kurz erläutert werden.

2.1 'Crack Edges'

Ohne eine Einschränkung auf einen speziellen Segmentierungsalgorithmus, können wir für die Kantenextraktion vollständig partitionierte Bilder der folgenden Form voraussetzen.

$$\boldsymbol{I} = \bigcup_i \mathcal{R}_i \text{ mit } \mathcal{R}_i \cap \mathcal{R}_j = \emptyset,\ \forall i \neq j \tag{1}$$

Die einzelnen Regionen $\mathcal{R}_i$ grenzen unmittelbar aneinander. Das bedeutet, dass keine trennenden Kanten im Bild auftreten (siehe Abbildung 1).

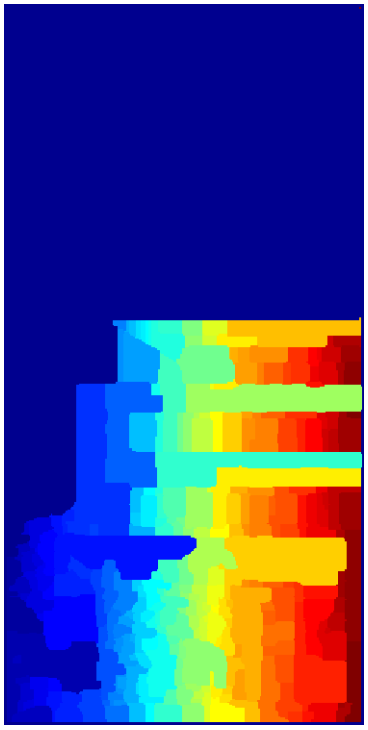


Abbildung 1: Labelbild der bei der Segmentierung entstandenen Regionen. Jede Region bekommt einen eindeutigen Farbwert zugewiesen.

Manche Segmentierungsalgorithmen liefern jedoch zusätzlich zu diesen Regionen einen (in der Regel ein Pixel breiten) Rand. Ein Beispiel hierfür stellt der Wasserscheidenalgorithmus dar. In diesem Fall müssen die Randpixel zuvor eliminiert werden. Wir verwenden dazu die im Anhang A beschriebene Funktion `removeWatershed.m`. Das Entfernen der Randpixel hat den Vorteil, dass nicht unnötig viele Sonderfälle für den Aufbau einer 'Winged Edge' Struktur betrachtet werden müssen. Dies würde die Laufzeiten erheblich verlängern.
Um aus der oben aufgeführten Partitionierung (1) eine 'Winged Edge' Struktur ableiten zu können, benötigen wir eindeutige Koordinaten der Knoten, durch welche die Kanten im Bild beschrieben werden. Am einfachsten ist es hierbei mit ganzzahligen Pixelwerten zu arbeiten, die in dieser Art der Darstellung jedoch nicht gegeben sind. Die Kanten liegen genau zwischen den Pixeln. Aus diesem Grund führen wir 'Crack Edges' ein. Das bedeutet, dass wir künstliche Ränder der einzelnen Maschen in das segmentierte Bild $\boldsymbol{I}$ einfügen. Hierfür wird das Bild in jeder zweiten Zeile und Spalte um eine Zeile bzw. Spalte Nullen erweitert. Diese Aufgabe übernimmt die Funktion `CrackEdge.m`. Nach Eingabe des segmentierten und gelabelten Bildes liefert

diese Funktion die Koordinaten aller Knoten im erweiterten 'Crack Edge' Bild (also dem Bild mit zusätzlichen Nullzeilen und -spalten), getrennt nach Kreuzungs- und Eckpunkten.

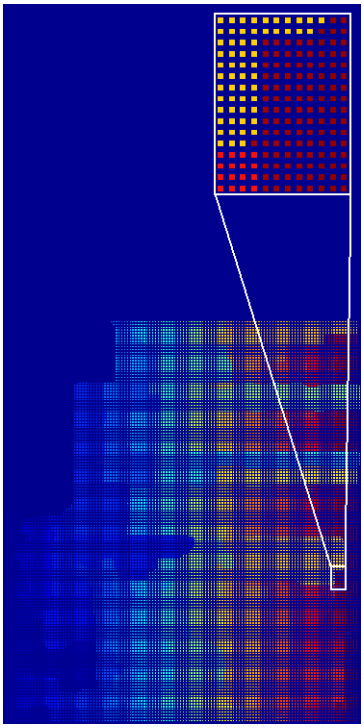
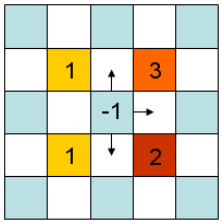
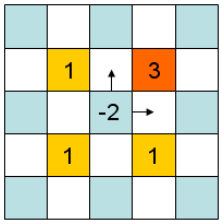


Abbildung 2: Bild der 'Crack Edges'. Das Labelbild wurde in jeder zweiten Zeile um eine Nullzeile erweitert und in jeder zweiten Spalte um eine Nullspalte erweitert.

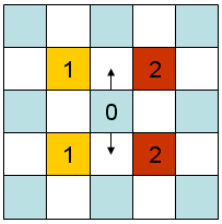
Das Bild der 'Crack Edges' (Abbildung 2) enthält neben den Nullzeilen und -spalten negative Werte an allen Knotenpunkten. Hiermit wollen wir direkt Gruppen von Knotenpunkten unterscheiden. Wir differenzieren zwischen Kreuzungspunkten und Knickpunkten. Unter Kreuzungspunkten verstehen wir diejenigen Knoten, an denen drei oder vier verschiedene Maschen aneinaderstoßen. Knoten, an denen nur zwei verschiedene Maschen zusammenstoßen, können entweder zu einer glatten Kante oder zu einem Knickpunkt gehören, abhängig davon in wievielen Nachbarpixeln eine der beiden Maschen vertreten ist (Abbildung 3).



(a) Kreuzungspunkt mit drei angrenzenden Maschen.



(b) Knickpunkt zwischen zwei verschiedenen Maschen.



(c) Nicht relevanter Punkt zwischen zwei Maschen.

Abbildung 3: Schematische Darstellung der 'Crack Edges'. Gespeichert werden Knotenpunkte. Wir unterscheiden hierbei Kreuzungspunkte (Abbildung 3(a)) und Knickpunkte (Abbildung 3(b)) in Abhängigkeit von der Zahl der angrenzenden Maschen. Pixel, an denen nur zwei Maschen zusammentreffen, sind nur relevant, wenn dort zwei Kanten orthogonal aufeinander treffen (Abbildung 3(b)). Andernfalls werden diese Pixel nicht als Knotenpunkte gespeichert (Abbildung 3(c)).

Weiterhin werden die Koordinaten der Knotenpunkte getrennt nach der Art der Knotenpunkte in je einer Matrix (`ecoord`) gespeichert. Zu jedem Knotenpunkt wird ebenfalls gespeichert, in welche Richtungen Kanten verlaufen (vgl. Abbildung 4). Hierfür wird eine Matrix (`nesw`) angelegt. Jede Zeile steht für einen Knoten. In der ersten Spalte steht eine 1, falls eine Kante in Richtung Norden (`n`) verläuft, sonst eine 0. Für die Spalten 2 bis 4 sind die Einträge analog für die Richtungen Osten (`e`), Süden (`s`) und Westen (`w`).
Anschließend können die Knoten, Kanten und Maschen der 'Winged Edge' Struktur bestimmt und in eine Datenbank geschrieben werden. Dies geschieht mit Hilfe von `WingedEgde.m`. Diese Funktion arbeitet mit dem Bild der 'Crack Edges' sowie den daraus abgeleiteten Daten (`ecoord` und `nesw`), der Angabe von Bildnummer und Skala, sowie mit der Verbindung zur Datenbank (`con = JavaSQLVerbindung()`).

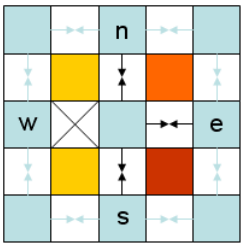


Abbildung 4: Ausschnitt aus der 'Crack Edge' Struktur. Richtungen, in denen von den blau eingefärbten Pixeln aus Kanten verlaufen, sind durch Pfeile gekennzeichnet. Da das Auftreten von Kanten am Rand des Ausschnitts noch nicht gesichert ist, sind diese Kanten in blau dargestellt. Das Kreuz markiert die Richtung, in der keine Kante verläuft. Hier tritt auf beiden Seiten die gleiche Masche auf. Für das mittlere Pixel sind somit die Richtungen Norden, Osten und Süden zulässig. Die Matrix `nesw` erhält für diese Pixel den Eintrag [1 1 1 0].

Die drei Funktionen `removeWatershed.m`, `CrackEdge.m` und `WingedEdge.m` sind in `WingedEdgeAusfuehren.m` zusammengefasst. Durch den Anwender sind lediglich die gewünschte Bildnummer und Skala einzustellen. Gegebenenfalls ist der Dateiname in `ladeBilder.m` zu ändern, falls dieser nicht der Form `Bild'Bildnummer'_regions_at_scale'Skalennummer'.tif` entspricht. Mit der Funktion `WingedEdgeAusfuehren.m` werden die im Bild gefundenen 'Crack Edges' direkt in eine Datenbank geschrieben. Der Aufbau der Datenbank ist ausführlich in [Her07] nachzulesen.

2.2 'Feature Operators' (FOP)

Bei 'Feature Operators' (FOP) handelt es sich um ein Progamm zur Extraktion von Punkten, Kanten und Regionen. Die Erkennung dieser Objekte beruht auf dem mittleren quadratischen Gradienten

$$\overline{\Gamma_\sigma g} \;=\; G_\sigma * \Gamma g. \tag{2}$$

Hierbei bezeichnet G_σ eine rotationssymmetrischen Gauß-Funktion mit Mittelwert 0 und Standardabweichung σ, sowie $\Gamma g = \nabla g \, \nabla g^T$ den quadratischen Gradienten der Intensitäten im Bild [För94].
Zur Unterscheidung, ob eine homogene Region oder eine andere Struktur vorliegt, verwendet FOP als Homogenitätsmaß die Spur des mittleren quadratischen Gradienten $h = tr(\overline{\Gamma_\sigma g}) = \lambda_1 + \lambda_2$. Mit λ_1, λ_2 seien die Eigenwerte bezeichnet. Liegt keine homogene Region vor, so kann über das Verhältnis der Eigenwerte der Grad der Isotropie (der Unabhängigkeit von der Richtung) bestimmt werden. Damit kann zwischen punkt- und linienförmigen Objekten

differenziert werden.
Für unsere Untersuchung sind ausschließlich die Kanten relevant. Wir wollen die mit Hilfe von FOP extrahierten Kanten mit den 'Crack Edges' aus Kapitel 2.1 vergleichen und dadurch mögliche relevante 'Crack Edges' für die Gebäudedetektion finden. Diejenigen 'Crack Edges' werden hoch gewichtet, die in der Nähe von FOP-Kanten und in gleicher Richtung verlaufen.
Zum Ausführen von FOP werden die Funktionen `ip_fop_matlab.m` und `ip_plot_fop_matlab.m` benötigt. Deren Bedienung bedarf einiger Einstellungen:
Zunächst benötigen wir die Information, welche Objekte extrahiert werden sollen. Um Punkte und Regionen (Blobs) auszuschließen, werden diese Parameter auf 0 gesetzt.

```
params.point_start_point_extraction = 0
params.blob_start_blob_extraction = 0
params.edge_start_edge_extraction = 1
```

Weiterhin sind die Parameter des Rauschens einzustellen. Wir verwenden Rauschen als eine lineare Funktion des Grauwertes g: $a + b \cdot g$ mit den Parametern

```
params.noise_a=25
params.noise_b=0.
```

Falls ein informationserhaltender Filter (ipf = information reserving filter) verwendet werde soll, muss der folgende Parameter auf 1 gesetzt werden. Für unsere Aufgabe ist dies nicht relevant. Daher setzen wir

```
params.edge_ipf_iterations = 0
```

Wir müssen zwei Arten der Glättung berücksichtigen. Zum einen erfolgt eine Glättung bei der Gradientenberechnung (σ_1). Des Weiteren wird die Quadrtische-Gradienten-Matrix geglättet (σ_2). Das beinflusst, inwieweit eine Kante als eine oder zwei Linien erkannt wird.

```
params.edge_sigma1 = 1
params.edge_sigma2 = 1
```

Wir wollen zudem nur lange, aussagekräftige Kanten verwenden. Empirische Untersuchungen haben gezeigt, dass bei den von uns betrachteten Bildern eine Kantenlänge von 70 Pixeln geeignet ist. Die minimale Kantenlänge, ab der eine Kantenextraktion erfolgen soll, lässt sich über folgenden Parameter einstellen.

```
params.edge_min_pix_len = 70
```

In der Funktion `Kantenextraktion.m` sind die von uns verwendeten Einstellungen bereits enthalten. Sie wird ausgeführt, um eine Matrix (`fedge`) der FOP-Kanten zu erhalten. Diese Matrix beinhaltet in den ersten beiden Spalten, die Koordinaten des ersten Endpunktes einer Kante und in den letzten beiden Spalten die Koordinaten des zweiten Endpunktes der gleichen Kante.

Die Ergebnisse der Kantenextraktion mittels FOP ist in Abbildung 5. Die schwarzen Kanten stellen die 'Crack Edges' dar. In Rot sind die FOP-Kanten eingezeichnet.

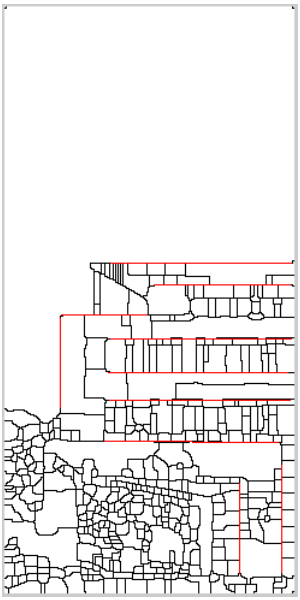


Abbildung 5: Bild der 'Crack Edges' (schwarz) und der Kanten, die mit Hilfe des FOP-Algorithmus unter den oben aufgeführten Einstellungen gefunden werden (rot).

3 Matching von Crack Edges und FOP-Kanten zur Auswahl relevanter Kanten

Kanten erachten wir für die Gebäudedetektion als relevant, wenn sie sowohl bei der Segmentierung auftreten, als auch bei der Kantenextraktion (als lange Kanten von mindestens 70 Pixeln) gefunden werden. Diese Einstellung wurde bereits für den FOP-Algorithmus vorgenommen. Wir wollen nun diejenigen 'Crack Edges' finden, die in der Nähe der mit Hilfe des FOP-Algorithmus gefundenen Kanten liegen. Dies übernimmt die Funktion `edgematching.m`.
In `edgematching.m` fließen als Eingabeparameter die Ergebnisse aus der `Kantenextraktion.m` mit Hilfe von FOP sowie die 'Crack Edges' ein. Letztere stammen aus der Datenbank der 'Winged-Edge'-Struktur (siehe [Her07]), so dass hier lediglich die Bild- und Skalennummer benötigt werden.
Innerhalb eines Fensters um die FOP-Kanten suchen wir nach geeigneten 'Crack Edges' (Kapitel 3.1), die wir anschließend zu einem Kantenzug verbinden (Kapitel 3.2).

3.1 'Searchbox'

Da FOP-Kanten nicht immer exakt dort liegen, wo 'Crack Edges' auftreten, suchen wir in einer Umgebung der FOP-Kanten nach passenden 'Crack Edges'. Dazu wird ein Fenster ('Searchbox') um jede FOP-Kante herum aufgebaut (Abbildung 6), in dem die gültigen 'Crack Edges' liegen. Zwei Kanten der 'Searchbox' verlaufen parallel zur FOP-Kante. Der Versuch, die Größe der 'Searchbox' aus den Kovarianzen der FOP-Kante zu bestimmen, ist an deren geringen Größe gescheitert. Als geeignet hat sich der empirische Wert von 5 Pixeln Abstand zur FOP-Kante erwiesen.

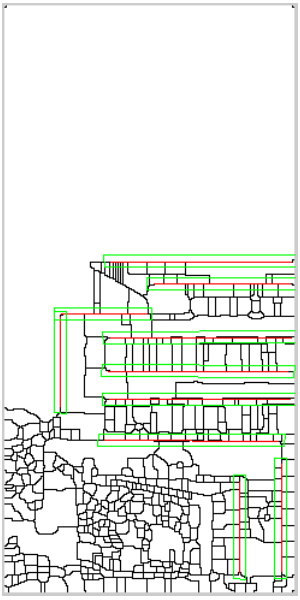


Abbildung 6: Neben den schwarzen 'Crack Edges' und den roten FOP-Kanten, ist um jede FOP-Kante eine grüne 'Searchbox' im Abstand von 5 Pixeln eingezeichnet.

Verwendet werden nur Crack Edges, die innerhalb dieser 'Searchbox' liegen. Sie bekommen zunächst das Gewicht 0.5, während alle anderen 'Crack Edges' das Gewicht 0 erhalten. Wir haben zusätzliche Gewichte für die Ausrichtung der Kanten eingeführt, so dass die Summe der Gewichte maximal 1 ergeben kann. Kanten innerhalb der 'Searchbox', die parallel zur FOP-Kante verlaufen bekommen ein zusätzliches Gewicht von 0.25 (also insgesmat 0.75). Abhängig von ihrem Längenverhältnis bezüglich der FOP-Kante ist ein weiteres Gewicht von $w * 0.25$ $(0 \leq w \leq 1)$ vorgesehen.
Diese Gewichte sind bislang allerdings noch nicht in den Algorithmus zur Auswahl eines Kantenzuges eingeflossen (siehe Kapitel 3.2).

3.2 'Edge Tracking' mit Hilfe des A* Algorithmus

Der A*-Algorithmus sucht nach der kürzesten Verbindung zwischen zwei Punkten in einem Graphen. Dabei werden von einem Knoten v die Kosten $h(v)$ bis zum Zielknoten geschätzt. Wir verwenden die euklidische Distanz als Kostenfunktion. Diese ist monoton steigend und gewährleistet daher, dass der kürzeste Weg gefunden wird, sofern kein Loch im Graphen auftritt. Zusammen mit der bereits zurückgelegten Distanz $g(v)$ (der Summe der Pfadgewichte) vom Startknoten zum Knoten v ergibt dies die Gesamtkosten $f(v) = g(v) + h(v)$. Der Pfad wird vom Startknoten aus in diejenige Richtung weiter verfolgt, die die geringsten Gesamtkosten $f(v)$ aufweist.
Wir wenden den A*-Algorithmus an, um in der Menge von 'Crack Edges' in einer 'Searchbox' (siehe Kapitel 3.1), diejenigen herauszufinden, die einen möglichst direkten Pfad entlang der FOP-Kante beschreiben. Start- und Endpunkt diese Pfades bilden dabei diejenigen Knoten der 'Crack Edges', die am nächsten beim Start- bzw. Endpunkt der FOP-Kante liegen.
Wir benötigen neben Start- und Endknoten auch eine Liste aller möglichen Knoten, einschließlich deren Nachbarknoten und Pfadgewichte. Diese Information ist im Cell-Array `node_list` gespeichert.
In der Funktion `AStar.m` wird eine nach Kosten aufsteigend sortierte Liste (`open_list`) der aktuellen Knoten angelegt, die mögliche Kandidaten auf dem Weg vom Start- zum Endknoten sind, jedoch noch nicht besucht wurden. Im ersten Schritt enthält diese Liste nur den Startknoten. In weiteren Schritten werden je die Nachbarn des gerade aktuellen Knoten aufgenommen. Der erste Knoten aus `open_list` (also der Knoten mit den geringsten Kosten) wird aus dieser Liste entfernt und in eine Liste der besuchten Knoten (`ancess_list`) geschrieben. Er wird damit zum aktuellen Knoten, dessen Nachfolger bestimmt und zusammen mit ihren geschätzten Kosten in die Liste `open_list` aufgenommen werden. Dies wird wiederholt, bis der Zielknoten erreicht ist. Von dort aus kann man nun die `ancess_list` zurückverfolgen bis zum Startknoten und erhält somit die Knoten des kürzesten Pfades in umgekehrter Reihenfolge (`path_list`).

Durch die Auswahl eines Fensters (der 'Searchbox') ist es nicht immer gewährleistet, dass genügend 'Crack Edges' vorliegen, um den Weg vom Start- zum Endpunkt zu verfolgen. Der A*-Algorithmus gibt in diesem Fall den Parameter `a=1` aus. Beim Auftreten solcher Löcher, führen wir zwei neue Endpunkte ein und wenden den A*-Algorithmus segmentweise an. Schließlich erhalten wir alle Kanten ('Crack Edges'), die entlang der FOP-Kante verlaufen.

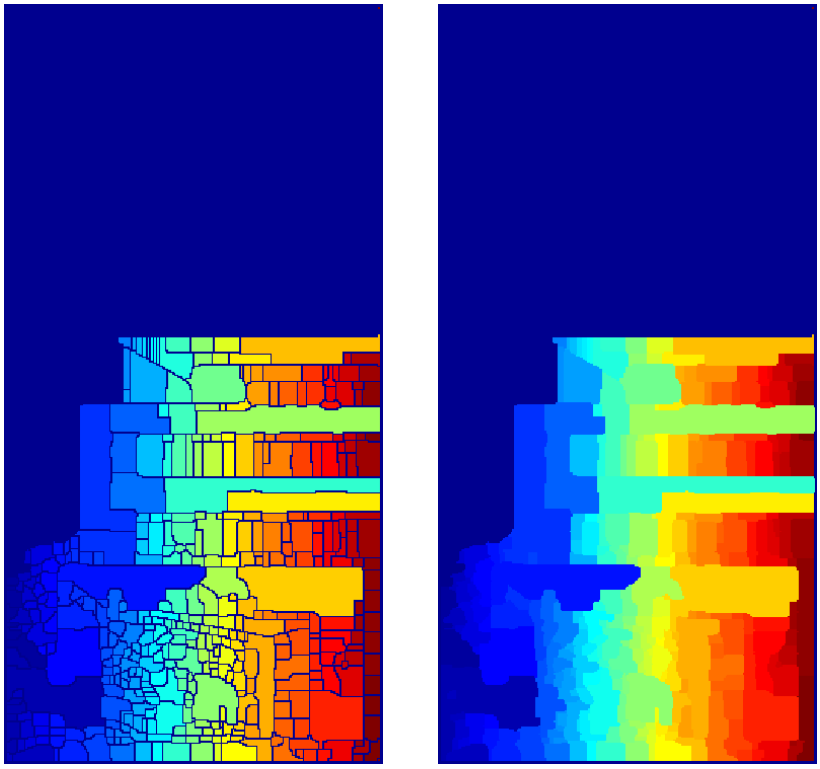
Die Funktion `edgetracking.m` stellt die Liste der Knoten für den A*-Algorithmus auf, führt diesen aus und verwaltet zusätzlich das Auftreten von Löchern im Graphen. Als Eingabeparameter benötigen wir Verbindung zur Datenbank (`con = JavaSQLVerbindung()`), um alle beteiligten Knoten zu ermitteln, sowie die FOP-Kanten (`fedges`), die Koordinaten (`kn`) und ID's (`knID`) der 'Crack Edges'. Des Weiteren wird eine Matrix (`inbox`) eingelesen, die Information darüber enthält, ob eine 'Crack Edge' in einer 'Searchbox' enthalten ist und wenn ja in welcher. Ausgegeben werden die Knoten des kürzesten Pfades vom Start- zum Zielpunkt (`edgetrack`), gegebenenfalls in Teilstücke zerlegt. Treten Löcher im Pfad auf, so sind diese in `edgetrack` durch eine Null gekennzeichnet.

4 Ausblick

Bislang wurden nur die Kanten in Hinblick auf eine Gebäudedetetion aus Bildern gewonnen und nach bestimmten Eigenschaften eingegrenzt. Ob sich lange Kanten, die sowohl bei der Bildsegemntierung als auch bei der Kantenexraktion mit Hilfe des FOP-Algorithmus auftreten, tatsächlich für die Gebäudeextraktion eignen, muss noch geprüft werden.
Wird dieser Ansatz weiter verfolgt, ist es eventuell sinnvoll, die Pfadgewichte im A*-Algorithmus zu variieren. Hier könnten die Gewichte der 'Crack Edges' (Kapitel 3.1) in den mit einbezogen werden, die bereits Informationen darüber enthalten, ob eine 'Crack Edge' parallel oder senkrecht zur FOP-Kante verläuft.

A Gewinnung der 'Crack Edge' Struktur

Zur Gewinnung der in Kapitel 2.1 beschriebenen 'Crack Edge' Struktur aus einem mit Hilfe eine Wasserscheidenalgorithmus segmentierten Bildes verwenden wir die Funktion `removeWatershed.m`. Die Funktion arbeitet mit einem Binärbild der Wasserscheiden (Wasserscheidenpixel = 0, Wasserscheidenregionen = 1).
Wir weisen die Wasserscheidenpixel den bereits vorhandenen Regionen zu. Dazu wird zunächst ein Labeling des Binärbildes vorgenommen. An einer Wasserscheide grenzen in der Regel zwei Regionen (mit unterschiedlichen Labels) aneinander. Mehrere Regionen teffen nur an Kreuzungspunkten bzw. Blockpixeln (das sind Kreuzungspunkte, die größer als ein Pixel sind) zusammen. Bevorzugt werden die Wasserscheidenpixel derjenigen Region zugeschrieben, die bereits einen größeren Flächeininhalt besitzt. Wir beginnen mit der Zuweisung der Kanten und weisen zuletzt die Kreuzungspunkte und Blockpixel den einzelnen Regionen zu.
Als Ergebnis erhalten wir wiederum ein gelabeltes Bild, das alleine durch die Regionen vollständig partitioniert ist. Die Ausgangssituation und das Ergebnis sind in Abbildung 7 dargestellt.
Es ist darauf zu achten, dass durch die Zuweisungen der Wasserscheiden zu den Regionen keine ungültigen Regionen entstehen. Es kann nach dem Löschen der Wasserscheiden der Fall auftreten, dass zwei Regionen, die nur über eine 8er Nachbarschaft verbunden sind, das gleiche Label aufweisen. Dieser Defekt wird in der Funktion `WingedEdgeAusfuehren.m` nach Anwendung von `removeWatershed.m` durch Hinzufügen eines weiteren Labels und Umbenennen einer 'Teil'-Masche behoben.



(a) (b)

Abbildung 7: Vom gelabelten Wasserscheidenbild (7(a)) zur 'Crack Edge' Struktur (7(b)).

Literatur

[DSF06] Martin Drauschke, Hanns-Florian Schuster, and Wolfgang Förstner. Detectibility of buildings in aerial images over scale space. In Wolfgang Förstner and Richard Steffen, editors, *Symposium of ISPRS Commission III: Photogrammetric Computer Vision*, volume XXXVI Part 3, pages 7–12, Bonn, 20-22 September 2006 2006. ISPRS, ISPRS.

[För94] W. Förstner. A framework for low-level feature extraction, 1994.

[Her07] K. Herms. Aufbau einer datenbank zur verwaltung von bildsegmenten. Technical Report 5, Institut für Geodäsie und Geoinformation, Abteilung Photogrammetrie, 2007.