GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

XX XXX XX
YYY YYY Y

Bachelorarbeit
im Studiengang „Angewandte Informatik“

SUSAN

Ein Ansatz zur Strukturerkennung in Bildern

Julian Lüken
julian.lueken@stud.uni-goettingen.de

Institut für Numerische und Angewandte
Mathematik

Bachelor und Masterarbeiten des Zentrums für
angewandte Informatik an der
Georg-August-Universität Göttingen

25. September 2019

Inhaltsverzeichnis

Kapitel 1

Einführung

Kantendetektoren sind ein wichtiges Werkzeug der Bildverarbeitung...

Kapitel 2

Mathematische Grundlagen

2.1 Analytische Grundlagen

2.2 Varianzanalyse

2.3 Bildverarbeitung

Kapitel 3

Der SUSAN Kantendetektor

In diesem Kapitel wird der Algorithmus angegeben, danach werden einige Details zur Implementation des Algorithmus in Python 3 erläutert. Im dritten Abschnitt wird bewiesen, dass der SUSAN-Kantendetektor funktioniert und im letzten Teil werden ein paar numerische Experimente durchgeführt.

3.1 Der Algorithmus

Der SUSAN Kantendetektor aus [ref] besteht im Grunde aus drei großen Schritten:

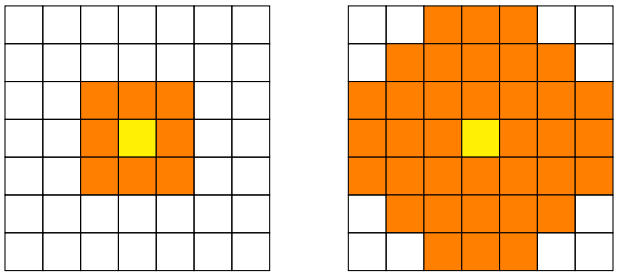
Im ersten Schritt wird eine Maske über jedes Pixel gelegt. Mithilfe dieser Maske berechnet man für jedes Pixel eine Kennzahl A, die im weiteren Verlauf dieser Arbeit „Antwort“ heißt. Die lokalen Maxima von A liegen genau auf den Nullstellen der partiellen Ableitungen in horizontale und vertikale Richtung unseres Eingangsbildes I, wie später noch gezeigt wird.

Der zweite Schritt ist die sogenannte Non-Maximum-Suppression, bei der mithilfe der Richtung der im ersten Schritt berechneten Antwort die Kanten im Eingangsbild I noch genauer lokalisiert werden. Die Lokalisation von Kanten auf Bildern ohne Rauschen ist nach dem zweiten Schritt abgeschlossen.

Da der erste Schritt des SUSAN-Kantendetektors auf einem Schwellenwert t basiert, vernachlässigt er Intensitätsdifferenzen in I, die unterhalb von t liegen. Falschpositive und falschnegative Ergebnisse sind auf mit Rauschen behafteten Digitalbildern keine Seltenheit. Aus diesem Grund gibt es noch einen dritten Ausdünnungsschritt, in welchem räumlich isolierte Antworten entfernt werden und Kanten vervollständigt werden. Dieser Schritt wird näher in Kapitel [implementation] behandelt

3.1.1 Das SUSAN-Prinzip

Sei I ein Eingangsbild. Um jedes Pixel im Bild wird eine Maske gelegt. Für unseren Zweck betrachten wir lediglich die Masken



wobei das gelbe Pixel der Mittelpunkt der Maske ist, die orangenen Pixel in der Maske und die weißen Pixel außerhalb der Maske liegen.

Das SUSAN-Prinzip funktioniert wie folgt: Für jedes Pixel r_0 in I (wobei $I(r_0)$ der Grauwert am Pixel r_0 ist), berechne die Antwort

$$A(r_0) = \max\{0, g - n(r_0)\}.$$

Dabei ist n definiert als

$$n(r_0) = \sum_r c_t(r, r_0),$$

wobei r die Pixel in der respektiven Maske sind und

$$c_t(r, r_0) = \exp\Big(- \Big(\frac{I(r) - I(r_0)}{t} \Big)^6 \Big)$$

eine Vergleichsfunktion für zwei Pixel ist. Statt der obigen Vergleichsfunktion kann auch die Näherung

$$c_t(r, r_0) \approx \begin{cases} 1 & \text{falls } |I(r) - I(r_0)| \leq t \\ 0 & \text{sonst} \end{cases}$$

verwendet werden. Als USAN bezeichnen wir genau diejenigen Pixel r aus der Maske, für die $c_t(r, r_0) > 0$, jedes Pixel hat also seine eigene USAN.

3.1.2 Non-Maximum-Suppression

Das oben genannte Prinzip findet Kanten innerhalb von Bildern mit nur bedürftiger Genauigkeit. Im Bereich der eigentlichen Kante stellt man fest, dass die Antwort A ungleich 0 ist. Um die Kante genauer zu lokalisieren, verwenden wir das Prinzip der Non-Maximum-Suppression, bei der nur die maximale Antwort entlang einer Kante erhalten bleibt. Zu diesem Zweck wird die Richtung eines jeden Pixels $r_0 = (x_0, y_0)$, für welches $A(x_0, y_0) \neq 0$ gilt, durch folgende Fallunterscheidung berechnet:

1. **Inter-Pixel:**

   Falls die Größe der USAN die des Maskendurchmessers übersteigt und die Distanz zwischen $\mathrm{COG}(r_0)$ und r_0 größer als 1 Pixel ist, so ist die Richtung $D(r_0)$ gegeben

durch

$$D(r_0) = \begin{cases} \arctan\left(\frac{x_0 - \mathrm{COG}(x_0)}{y_0 - \mathrm{COG}(y_0)}\right) & \text{falls } \mathrm{COG}(y_0) \neq y_0 \\ \frac{\pi}{2} & \text{sonst} \end{cases},$$

wobei

$$\mathrm{COG}(r_0) := \frac{\sum_r r\, c(r, r_0)}{\sum_r c(r, r_0)}.$$

2. **Intra-Pixel:**

   Andernfalls müssen wir die zweiten Momente der USAN folgendermaßen berechnen:

$$\begin{aligned} d_{x_0} &:= \sum_r (x - x_0)^2\, c_t(r, r_0) \\ d_{y_0} &:= \sum_r (y - y_0)^2\, c_t(r, r_0) \\ \sigma &:= \mathrm{sgn}\left(\sum_r (x - x_0)\,(y - y_0)\, c_t(r, r_0)\right) \end{aligned}$$

   Dabei ergibt sich die Kantenrichtung als

$$D(r_0) = \begin{cases} \sigma \arctan \frac{d_{y_0}}{d_{x_0}} & \text{falls } d_{x_0} \neq 0 \\ \frac{\pi}{2} & \text{sonst} \end{cases}$$

   Falls allerdings $d_{x_0} = 0$, so ist $D(r_0) = \frac{\pi}{2}$

Die Berechnungen für die jeweiligen Kantenrichtungen an jedem Pixel werden in der Implementation gleichzeitig mit den Berechnungen für das SUSAN-Prinzip durchgeführt. Auf diese Weise müssen c_t-Werte an jeder Stelle nur einmal berechnet werden.

Die Richtung lohnt sich nur für diejenigen Pixel (i, j) zu bestimmen, für die $A(i, j) > 0$ gilt. Ist die Richtung der Kante bestimmt, so können wir die lokalen Maxima von A entlang der Richtung, die senkrecht zur Kantenrichtung steht, erhalten. Alles, was kein lokales Maximum entlang dieser Richtung ist, wird verworfen. Wir erhalten so ein neues Bild. In Kapitel 3.2 wird darauf eingegangen, wie genau dieser Prozess implementiert wurde.

3.1.3 Ausdünnen

Da viele digitale Bilder eingangs mit Rauschen behaftet sind, ist es manchmal hilfreich, einzelne Antworten zu entfernen und dafür andere hinzuzufügen. So zum Beispiel, falls eine Kante Zusätzlich zur Non-Maximum Suppression ist es hilfreich, einzelne Pixel zu entfernen, zu verschieben und

3.2 Implementation

Für meine persönliche Implementation des SUSAN-Kantendetektors in Python 3 habe ich eine Reihe von ausgewählten Gütekriterien aus der Softwaretechnik herangezogen:

1. Funktionalität
2. Effizienz
3. Benutzbarkeit
4. Änderbarkeit

Grundsätzlich lässt sich die Struktur der Software folgendermaßen darstellen: ..... Es sei zu jedem Unterpunkt dieses Kapitels gesagt, dass sich die Implementation im Anhang [ref] befindet.

3.2.1 Masken

3.2.2 Vergleichsfunktion

Zu Gunsten der Effizienz habe ich für die Vergleichsfunktion eine Lookup-Tabelle implementiert, wie es auch in [quelle] nahgelegt wurde. Beim Initialisieren des `Susan`-Objekts wird ein Feld erzeugt. Das Ziel ist es, alle möglichen Werte, die c_t (siehe 3.1.1) annehmen kann, zu speichern. In unserem Fall ist c_t folgendermaßen definiert:

$$c_t(a,b) := \exp\left(-\left(\frac{I(a)-I(b)}{t}\right)^6\right)$$

Ein Pixel in einem 8-bit Graustufenbild kann $2^8 = 256$ mögliche Intensitäten annehmen, demnach braucht das Feld für die Differenz zweier solcher Graustufenintensitäten 512 Plätze. Regulär werden Felder entweder ab 0 oder 1 indiziert. In Python werden Felder ab 0 indiziert, allerdings bietet Python den Vorteil, dass man mit Index $-i$ das i-te Element von hinten abrufen kann. In der Implementation ist diese Lookup-Tabelle dank dieser speziellen Art der Indizierung einfach und elegant: Der Index $a-b$ des Feldes steht für $c_t(a,b)$.

3.2.3 Non-Maximum-Suppression

Die Non-Maximum-Suppression ist ein Vorgang, bei der jede Kante genauer lokalisiert wird. Entlang einer Kante soll immer nur die maximale Antwort erhalten bleiben. Um die Non-Maximum-Suppression durchzuführen, wird zunächst für jedes Pixel (i,j) mit $A(i,j) > 0$ die Kantenrichtung $D(i,j)$ bestimmt. Die möglichen Kantenrichtungen werden dann für einen Zwischenschritt kategorisiert. Die Kategorien sind *negativ diagonal*, *vertikal*,

positiv diagonal und *horizontal*. Die Kategorie bestimmt, welche zwei adjazenten Pixel C für die Non-Maximum-Suppression interessant sind.

Bedingung	**Kategorie**	**Adjazente C**
$D(i,j) \leq -\frac{3}{8}\pi$	negativ diagonal	$\{(i+1,j-1), \quad (i-1,j+1)\}$
$D(i,j) > -\frac{3}{8}\pi, \quad D(i,j) \leq -\frac{1}{8}\pi$	vertikal	$\{(i-1,j), \quad (i+1,j)\}$
$D(i,j) > -\frac{1}{8}\pi, \quad D(i,j) \leq \frac{1}{8}\pi$	positiv diagonal	$\{(i,j-1), \quad (i,j+1)\}$
$D(i,j) > -\frac{3}{8}\pi$	horizontal	$\{(i+1,j-1), \quad (i-1,j+1)\}$

Für jedes Pixel (i,j) wird nun überprüft, ob $(i,j) = \max(C \cup \{(i,j)\})$ gilt. Gilt es nicht, so wird $A(i,j)$ unterdrückt.

3.2.4 Parallelisierung

Das SUSAN-Prinzip sieht vor, die Antwort $A(i,j)$ und die Kantenrichtung $D(i,j)$ aus immer nur mithilfe von Pixeln aus der Maske auf (i,j) zu berechnen. An dieser Stelle kann die Berechnung von A und D parallelisiert werden. In meiner Implementation liefert das Paket `multiprocessing` die nötigen Werkzeuge, um die vorhandenen Ressourcen zusammenzuschließen und die Berechnung von A und D parallel abzuarbeiten.

Der Einfachheit halber habe ich das Bild nur der Höhe nach partitioniert. Eine Partitionierung $\mathcal{S}$ ist eine Menge von nichtnegativen ganzen Zahlen $\{S_1, S_2, ..., S_n, S_{n+1}\}$. Für alle $i \in \{1, 2, ..., n\}$ arbeitet der Job J_i die Partition $(S_i, S_{i+1}]_{\mathbb{Z}}$ ab.

Unter der stark vereinfachten Annahme, dass alle Kerne gleich viele Fließkommazahloperationen pro Zeiteinheit abarbeiten können und gleichgroße Partitionen etwa gleichviel Rechenzeit benötigen, gelte folgende Aussage: Eine Partitionierung $\mathcal{S}$ für die gilt $\#\mathcal{S} = n + 1$ ist genau dann optimal, wenn für alle Partitionen gilt

$$|\#(S_i, S_{i+1}]_{\mathbb{Z}} - \#(S_j, S_{j+1}]_{\mathbb{Z}}| \leq 1, \quad \forall i \neq j, \quad i,j \in \{1, 2, ..., n\}$$

Angenommen der Computer, auf dem die Software läuft, besitzt n Kerne, kann also n Jobs $J_1, J_2, ..., J_n$ gleichzeitig abarbeiten, und ein Bild der Höhe H wird eingegeben. Sei $k := H \bmod n$ und $z := \left\lfloor \frac{H}{n} \right\rfloor$. Dann ist die optimale Partitionierung $\mathcal{S}$ nach der Höhe gegeben durch:

$$
\begin{aligned}
S_1 &= 0 \\
S_2 &= S_1 + z + 1 \\
S_3 &= S_2 + z + 1 \\
&\vdots \\
S_{k-1} &= S_{k-2} + z + 1 \\
S_k &= S_{k-1} + z + 1 \\
S_{k+1} &= S_k + z \\
&\vdots \\
S_n &= S_{n-1} + z \\
S_{n+1} &= S_n + z = H
\end{aligned}
$$

Die Jobs $J_1, J_2, ..., J_n$ arbeiten das komplette Bild ab, denn es gilt

$$\bigcup_{i=1}^{n} (S_i, S_{i+1}]_{\mathbb{Z}} = (0, H]_{\mathbb{Z}} = [1, H]_{\mathbb{Z}}$$

3.2.5 Ausdünnen

Das Ausdünnen, welches in [ref] näher beschrieben wird,

3.3 Rauschunterdrückung

3.4 Numerische Experimente