Fakultät für Mathematik mit Schwerpunkt Digitale Bildverarbeitung

Graphbasierte Segmentierung von CT-Scans

Bachelorarbeit im Fachbereich Informatik

Katrin Schmelz

Prüfer
Prof. Dr. Tomas Sauer

---

6. Oktober 2021

Abstract

Vor allem im medizinischen Bereich ist die Segmentierung von CT-Scans ein entscheidender Schritt um Leben zu retten, aber auch in anderen Bereichen, wie der Bildbearbeitung oder der Qualitätskontrolle, ist die Segmentierung nicht mehr wegzudenken.
Doch so vielfältig die Anwendungsmöglichkeiten sind, so groß ist auch die Fülle an Segmentierungstechniken.
Ziel dieser Arbeit ist es einen graphbasierten Segmentierungsansatz für 3D-Bilder zu implementieren und zu analysieren. Dazu wurde ein graphbasierter Segmentierungsalgorithmus nach Vorbild der Image Foresting Transform in MATLAB programmiert und schrittweise erklärt. Abschließend wurde der Algorithmus untersucht und Tests bezüglich Laufzeit und Qualität durchgeführt.

Inhaltsverzeichnis

1 Einleitung **7**

2 Definitionen **9**
2.1 Computertomographie 9
2.1.1 CT-Scans 9
2.2 Segmentierung 10
2.2.1 Pixelorientierte Verfahren 11
2.2.2 Kantenorientierte Verfahren 11
2.2.3 Regionenorientierte Verfahren 11

3 Mathematische Grundlagen **13**
3.1 Matrizen 13
3.2 Graphen 13

4 Graphbasierte Segmentierung **17**
4.1 Grundprinzip 17
4.2 Image Foresting Transform 18
4.3 Optimierungsmöglichkeiten 22

5 Algorithmus **23**
5.1 3D-Implementierung 23
5.1.1 Directions.m 23
5.1.2 getAdjacency.m 24
5.1.3 ift.m 27
5.1.4 showSegment.m 30
5.2 2D-Implementierung 31

6 Analyse **33**
6.1 Wahl der seed points 33
6.2 Einfluss der Bildgröße 33
6.3 Laufzeit 33
6.4 Stärken & Schwächen 37

7 Fazit **41**

Anhang: Quellcode **43**
A.1 Directions.m 43
A.2 getAdjacency.m 44
A.3 ift.m 46
A.4 showSegment.m 48
A.5 demo.m 48
A.6 Directions2D.m 49
A.7 getAdjacency2D.m 50
A.8 showSegment2D.m 51

Literaturverzeichnis **53**

1 Einleitung

Lassen Sie uns folgendes Bild analysieren:

Abbildung 1.1: Quokka [5]

Im Hintergrund sehen wir viele Äste, Blätter und den unteren Teil eines Baumstammes, einen Waldboden vermutlich. Im Vordergrund befindet sich ein etwa katzengroßes Tier mit braunem Fell. Die Kopfform und der Schwanz erinnern an eine Ratte, die Körperhaltung an ein kleines Känguru.
Tatsächlich handelt es sich um ein *Kurzschwanzkänguru*, auch *Quokka* genannt, welches im Südwesten Australiens zu Hause ist.
Obwohl der Großteil von uns dieses Tier vorher wahrscheinlich noch nie gesehen hat, konnten wir es gut identifizieren und klassifizieren. Auch wenn es sich dank seiner Fellfarbe, die dem trockenen Waldboden ähnelt, gut tarnen kann, war es doch ein Leichtes es von den Blättern abzugrenzen und zu bestimmen was zum Tier gehört und was nicht. Wenn uns jemand bitten würde dieses Quokka auszuschneiden, hätten wir, bis auf eventuell mangelndes Fingerspitzengefühl, wohl keine großen Probleme damit. Viele würden diese Fähigkeit nun als Kleinigkeit abtun. Schließlich schneiden wir schon Formen aus seit wir eine Schere halten können. Objekte erkennen und zuordnen, auch das können wir seit Kindesbeinen. Sogar in diesem Moment, beim Lesen dieses Textes, trennt unser Gehirn die einzelnen schwarzen Buchstaben vom weißen Hintergrund und ordnet ihnen eine Bedeutung zu, ohne dass wir uns dessen bewusst sind. Sobald wir

die Augen öffnen und Informationen aufnehmen, separiert unser Hirn das Gesehene in verschiedene Objekte. Es betreibt somit eine Form der *Segmentierung*.
Was für uns selbstverständlich ist, ist für den Computer eine Herausforderung. Der "sieht" beim Anblick des Bildes kein Känguru vor sich, sondern eine rasterförmige Ansammlung von Zahlen, ohne wirklichen Zusammenhang. Den müssen wir ihm erst beibringen. Mithilfe von Segmentierung, einem Teilgebiet der Bildverarbeitung und der *Computer Vision*, können zusammenhängende Regionen erkannt werden. Es ähnelt dem Ausschneiden aller (Teil-)Objekte des Bildes, sodass später jedes Objekt, unabhängig vom Rest des Bildes, separat bearbeitet werden kann. Diese Fähigkeit ist nicht nur für Fotografen äußert nützlich, sondern spielt auch in der Medizin eine tragende Rolle. Vor allem wenn es darum geht CT-Scans oder andere 3D-Modelle zu analysieren, ist Segmentierung ein beliebtes Hilfsmittel.
Genau mit diesem Bereich beschäftigt sich die vorliegende Arbeit: Der Implementierung und Analyse einer graphbasierten Segmentierung für 3D-Modelle. Dazu werden zuerst die Grundzüge der Segmentierung und der Computertomographie erläutert. Anschließend werden einige mathematische Grundlagen vermittelt, bevor im nächsten Kapitel auf die graphbasierte Segmentierung, im Speziellen die *Image Foresting Transform*, eingegangen wird. Abschließend wird eine solche graphbasierte Implementierung vorgestellt und analysiert.

2 Definitionen

2.1 Computertomographie

Die Computertomografie (CT), welche sich aus den altgriechischen Worten $\tau o\mu\acute{\eta}$ (*tomé*) "Schnitt" und $\gamma\rho\acute{\alpha}\phi\varepsilon\iota\nu$ (*gráphein*) "schreiben" zusammensetzt, bezeichnet ein bildgebendes Verfahren in der Radiologie. Sie ist eine Weiterentwicklung der Röntgentechnik und wurde in den 1970er Jahren von dem britischen Ingenieur Sir Godfrey Newbold Hounsfield entwickelt [3].
Das Ergebnis sind digitale Schnittbilder, aus denen mithilfe eines Computers 3D-Bilder erstellt werden können. Dies wird vor allem in der Medizin angewandt und hat gegenüber dem Röntgenverfahrens, neben der Darstellung im Drei- und Vierdimensionalen, den Vorteil die Organe ohne Überlagerungen darzustellen.

2.1.1 CT-Scans

Um einen CT-Scan zu generieren, werden mehrere Röntgenaufnahmen eines Objekts erstellt. Ein CT-Scanner besteht in der Regel aus einem Ringtunnel, auch Gantry genannt, und einer beweglichen Plattform auf dem das Untersuchungsobjekt platziert wird. In der Gantry befinden sich Röntgenröhren und Blenden, die den Röntgenstrahl auffächern, sodass er eine ganze Ebene durchleuchtet [1].
Sendet man einen Röntgenstrahl durch das zu untersuchende Objekt, wird ein Teil des Strahls, abhängig von der Struktur des Objekts, verschieden stark absorbiert und somit abgeschwächt. Bei einem Menschen würden z.B. Organe oder Gewebe den Strahl weniger abschwächen als Knochen.
Auf der gegenüberliegenden Seite sind Detektoren verbaut, die die ankommende Röntgenstrahlung messen und in elektrische Signale umwandeln.
Während der Aufnahme dreht sich die Gantry um das Objekt und der Prozess wird mehrmals wiederholt.
Die Signale werden dann an einen Computer weitergeleitet, der daraus ein Grauwertstufenquerschnittsbild *(Tomogramm)* erstellt [1]. Je nachdem wie viel von der Strahlung an den Detektoren ankommt, werden die Stellen im Röntgenbild heller oder dunkler dargestellt.

Da Röntgenstrahlen das Objekt immer komplett durchdringen, kommen in herkömmlichen Röntgenbildern häufig Überlagerungen von Objekten vor. Durch die zahlreichen Aufnahmen der Objekte aus verschiedenen Winkeln und die spätere Verarbeitung am Computer sind die entstandenen CT-Scans überlagerungsfrei.

Aus technischer Sicht kann man sich eine solche CT-Datei als dreidimensionale Matrix vorstellen. Deren Einträge bezeichnet man als *Voxel*, eine Zusammensetzung der englischen Wörter *volume* und *element*, im Zweidimensionalen als *Pixel (picture element)*. Die Werte dieser Voxel bzw. Pixel werden aus den Absorptionswerten berechnet und schließlich als Grauwerte interpretiert.

2.2 Segmentierung

Bei einer Segmentierung handelt es sich um eine Art Trennung von Objekten eines Bildes. Genauer beschreibt sie die Prüfung jedes Pixels oder Voxels auf Zugehörigkeit zu einem Objekt. Wir geben den Pixeln, welche uns vorher nur einen Grauwert liefern, sozusagen einen Zusammenhang.
Dies spielt vor allem im medizinischen Bereich eine wichtige Rolle, um einzelne Organe und Nerven herauszuarbeiten. Aber auch in andern Bereichen, wie der Qualitätskontrolle von Bauteilen und der Verarbeitung von Satellitenbildern, wird Segmentierung genutzt.
Will man als Vorbereitung auf eine Operation einen CT-Scan von einer Leber machen, so sind, aufgrund der verwendeten Röntgentechnik, auch angrenzende Strukturen enthalten. In einem solchen Fall kann dann Segmentierung genutzt werden um das gewünschte Organ z.B. einzufärben und es dadurch von anderen Organen hervorzuheben oder um nicht relevante Strukturen auszublenden.
Eine solche Segmentierung kann manuell vorgenommen werden, indem der Benutzer die Objektkontur für jedes Schnittbild selbst nachzieht. Da der Zeit- und Arbeitsaufwand aber mit jedem Schnittbild steigt und die Aufnahme eines Kopfes über 600 Schichten beinhalten kann, wird für solche Fälle ein Segmentierungsalgorithmus benötigt.

Für die Segmentierung gibt es verschiedene Verfahren, welche jeweils ein anderes Merkmal zur Differenzierung von Objekten betrachten oder stärker gewichten.
Im Folgenden werden die drei verbreitetsten Ansätze kurz vorgestellt.

2.2.1 Pixelorientierte Verfahren

Bei *pixelorientierten Verfahren* wird für jedes Pixel einzeln entschieden zu welchem Objekt es gehört, sodass kein Pixel ignoriert wird [4].
Für die Entscheidung zu welchem Objekt es angehört gibt es mehrere Möglichkeiten. Eine beliebte Methode dafür ist das Schwellwert-Verfahren, in dem z.B. bestimmte Grauwerte als Schwellwerte für Segmente definiert werden und das Pixel entsprechend des eigenen Grauwertes dem jeweiligen Segment zugeordnet wird.

Dieses Verfahren ist aufgrund seiner Einfachheit sehr effizient, jedoch ist es aufgrund der fixen Schwellwerte anfällig für "Ausreißer".

2.2.2 Kantenorientierte Verfahren

In *kantenorientierten Verfahren* wird im Bild nach Kanten und Konturen gesucht.
Um eine Kante zu erkennen, wird nach starken Grauwertänderungen bzw. großen Gradienten Ausschau gehalten. Dazu stehen verschiedene Hilfmittel, wie der Sobel-Operator oder Laplace-Filter, zur Verfügung [6].
Meist haben die Kantenzüge nach einem solchen Verfahren noch Lücken, die mit einem separaten Algorithmus geschlossen werden müssen.

2.2.3 Regionenorientierte Verfahren

Im Vergleich zu pixelorientierten Verfahren wird bei *regionenorientierten Verfahren* der eigene Pixelwert nicht isoliert betrachtet, sondern im Zusammenhang mit den Nachbarpixeln. Hier wird der Durchschnittsgrauwert einer Region betrachtet und anhand dessen versucht zusammenhängende Objekte zu finden [4].

Wie man sieht führen letztendlich viele Wege zum Ziel und oftmals ist es abhängig von der Struktur des zu segmentierenden Bildes, welches Verfahren das Bessere ist.

3 Mathematische Grundlagen

Um den verwendeten Algorithmus zu verstehen, werden im Folgenden einige relevante mathematische Grundlagen erklärt.

3.1 Matrizen

Definition 3.1 (Dünnbesetzte Matrix)
Eine *dünnbesetzte Matrix* oder *schwachbesetzte Matrix* ist eine Matrix, in der sehr wenige Einträge ungleich Null sind.

Zugegebenermaßen erscheint diese Formulierung etwas schwammig. Für *sehr wenige Einträge* gibt es keine einheitliche Definition, jedoch wird eine Matrix der Größe n^2 und nur n Einträgen ungleich Null meist schon als dünnbesetzt bezeichnet.

3.2 Graphen

Da im Algorithmus später das Ausgangsbild als Graph betrachtet wird, wobei jedes Voxel einem Knoten entspricht, definieren wir zuerst was ein Graph ist.

Definition 3.2 (Ungerichteter Graph)
Ein *ungerichteter Graph* ist ein Paar $G = (V, E)$, wobei V eine endliche, nichtleere Knotenmenge und $E \subseteq \binom{V}{2} = \Big\{\{u, v\} \subseteq V | u \neq v\Big\}$ eine endliche Kantenmenge ist.
Eine Kante $\{u, v\} \in E$ ist ein ungeordnetes Paar von Knoten u und v, wobei $\{u, v\}$ und $\{v, u\}$ in einem ungerichteten Graph die selbe Kante beschreiben.
Ist $e_{u,v} = \{u, v\} \in E$ eine Kante von G, so sind u und v zueinander *adjazent* oder *benachbart*.

Definition 3.3 (Gewichteter Graph)
Ein *gewichteter Graph* ist ein Tripel $G = (V, E, f_w)$, wobei $f_w : E \to \mathbb{R}$ eine *Gewichts-* oder *Kostenfunktion* ist. Für alle $e \in E$ heißt $f_w(e)$ das *Gewicht* oder die *Kosten* von e unter f_w.

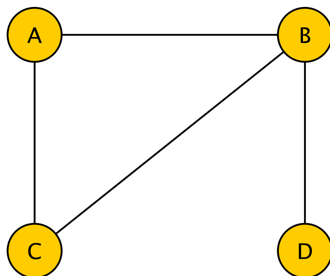
In dieser Arbeit werden ausschließlich gewichtete, ungerichtete Graphen benutzt.

Definition 3.4 (Adjazenzmatrix)
Sei n die Anzahl der Knoten in einem Graph $G = (V, E)$ mit $V = (v_1, ..., v_n)$. Eine *Adjazenzmatrix* zu G ist eine $n \times n$ Matrix $A(G) = (a_{i,j})$, $i, j \in \{1...n\}$ mit

$$a_{i,j} = \begin{cases} 1 & \text{falls } \{v_i, v_j\} \in E \\ 0 & \text{falls } \{v_i, v_j\} \notin E \end{cases} \tag{3.1}$$

Gilt $a_{i,j} = 1$, so sind die Knoten v_i und v_j zueinander *adjazent*.
Ist G ungerichtet und $\{v_i, v_j\} \in E$, so gilt $a_{i,j} = 1 \Leftrightarrow a_{j,i} = 1$. Somit ist sie entlang der Diagonalen für ungerichtete Graphen symmetrisch.

Beispiel 3.5
Zur Verdeutlichung ein ungerichteter Graph und seine zugehörige Adjazenzmatrix:



	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	0
D	0	1	0	0

Definition 3.6 (Adjazenzliste)
Eine *Adjazenzliste* für einen ungerichteten Graph $G = (V, E)$ ist eine Liste aller Nachbarn von $v \in V$ mit $\{u \in V : \{v, u\} \in E\}$.

Beispiel 3.7
Eine Adjazenzliste für den Graph aus Beispiel 3.5.

A: B C
B: A C D
C: A B
D: B

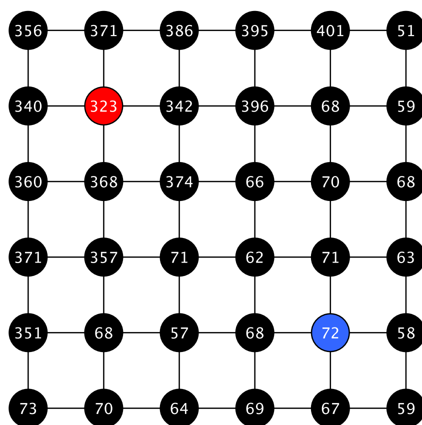
Definition 3.8 (Kürzester Pfad)
Ein *kürzester Pfad* in einem Graph $G = (V, E, f_w)$ ist ein Pfad zwischen zwei Knoten $u, v \in V$, dessen Kantengewichte bzgl. der Gewichtsfunktion f_w minimal sind.

4 Graphbasierte Segmentierung

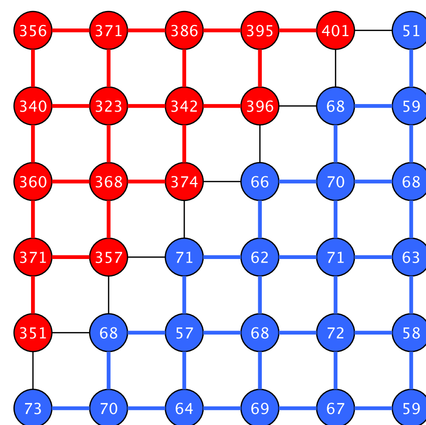
Neben den in Kapitel 2 genannten Segmentierungstechniken versprechen auch graphbasierte Methoden gute Resultate. Der Algorithmus in Kapitel 5, mit dem sich diese Arbeit hauptsächlich befasst, bedient sich ebendiesem Verfahren.
Das Prinzip der *Image Foresting Transform* mit User-Interaktion erscheint hier als guter Ansatz und wird im Folgenden erläutert.

4.1 Grundprinzip

Eine Segmentierung unternimmt eine "Beschriftung" von Pixeln bzw. Voxeln.
Würde man ein Bild in Vorder- und Hintergrund einteilen wollen, so wäre ein möglicher Segmentierungsoutput eine Liste aller Voxel des Bildes, wobei jedem Voxel ein Label zugeordnet ist. Diese Label können beliebige Formen haben, beispielsweise logische Werte wie „true“ für „Vordergrund“ und „false“ für „Hintergrund“.
Für eine Segmentierung mehrerer Objekte könnte das Label auch eine Zahl sein, sodass jeweils alle Pixel mit derselben Zahl zusammengehören oder auch eine Beschriftung durch Character oder Strings ist denkbar.



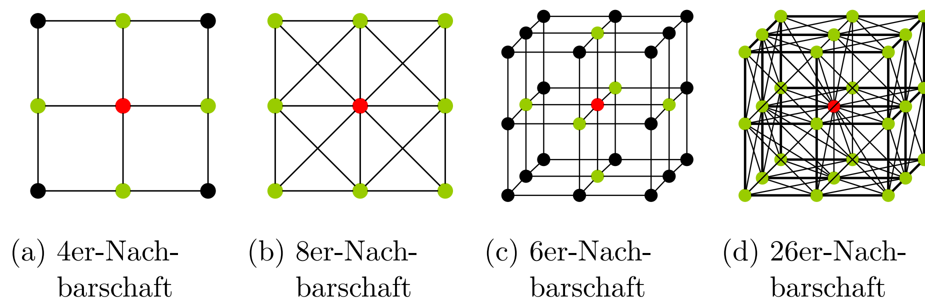
(a) Möglicher Segmentierungsinput mit zwei vom User gekennzeichneten Voxeln *(seeds)* und den Labeln *rot* und *blau*.



(b) Möglicher Segmentierungsoutput.

Graphbasierte Segmentierungsverfahren betrachten jedes Pixel bzw. Voxel als Knoten eines Graphen, wobei benachbarte Voxel mit Kanten verbunden sind. Den Kantengewichten liegt in der Regel eine Kostenfunktion zu Grunde, welche sich meist aus

den Farbwerten der Voxel und anderen Parametern wie Position oder Gradient zusammensetzt.
Die Wahl der Nachbarschaftgröße eines Voxels beeinflusst die Effizienz und Genauigkeit. Hier differenziert man meist Nachbarschaften in denen die Kanten der Voxel aufeinanderliegen müssen um Nachbarn zu sein und Nachbarschaften in denen sich auch lediglich die Ecken berühren können. Im Zweidimensionalen wären eine 4er- bzw. 8er-Nachbarschaft möglich, im Dreidimensionalen eine 6er- bzw. 26er-Nachbarschaft.



(a) 4er-Nachbarschaft (b) 8er-Nachbarschaft (c) 6er-Nachbarschaft (d) 26er-Nachbarschaft

4.2 Image Foresting Transform

Die *Image Foresting Transform (IFT)* [2] ist ein optimierter Dijkstra-Algorithmus für kürzeste Wege mit mehreren Eingaben.
Der CT-Scan wird hier als Graph interpretiert. Will man von einem Knoten bzw. Voxel zu einem anderen wandern, darf man jeweils nur über die Nachbarn eines Voxels entlanggehen. Dieses Besuchen eines anderen Voxels birgt sogenannte *(Pfad-)kosten*, welche z.B. die Grauwertdifferenzen dieser beiden Voxel bilden. Zwischen zwei Voxeln gibt es meist mehrere mögliche Pfade, von denen einige günstig und andere teuer sind. Benachbarte Voxel deren Grauwerte nahe beieinanderliegen, die also eine ähnliche Farbe haben, haben eine höhere Wahrscheinlichkeit zum gleichen Objekt zu gehören als solche, deren Grauwertdifferenz groß ist. Da ein Voxel nur dann einem solchen *kürzesten Pfad* angehört, wenn es von einem anderen Pfadvoxel aus, im Gegensatz zu seinen anderen Nachbarn, am günstigsten zu erreichen ist, gehören alle Voxel eines kürzesten Pfades mit hoher Wahrscheinlichkeit zum selben Objekt, insofern Anfangs- und Schlussvoxel zum selben Objekt gehören.
Diese Überlegungen bilden das Grundgerüst der IFT. Ziel ist es von jedem Voxel im Bild einen kürzesten Pfad zu einem Voxel aus einer vom User definierten Menge an Anfangsvoxeln, den *seed points* oder kurz *seeds*, zu finden, also jedem Voxel den billigsten seed zuzuordnen. Die Pfade die dabei entstehen und an den seeds, den sogenannten

Wurzeln, zusammenlaufen, bilden einen *optimum-path forest*, eine Liste die jedem Voxel seinen Pfad bis zu einem seed oder *nil*, falls das Voxel selbst ein seed ist, zuordnet. Dieser optimum-path forest ist der Output des Algorithmus.
Die seeds werden vom User gewählt und sollten mindestens ein Voxel jedes Segments beinhalten. Damit man seeds, die zum selben Objekt gehören und nicht zusammengehörige seeds unterscheiden kann, wird jedem seed ein *Label*, z.B. eine Zahl, angefügt. Haben zwei seeds die gleichen Label, gehören sie dem selben Objekt an.

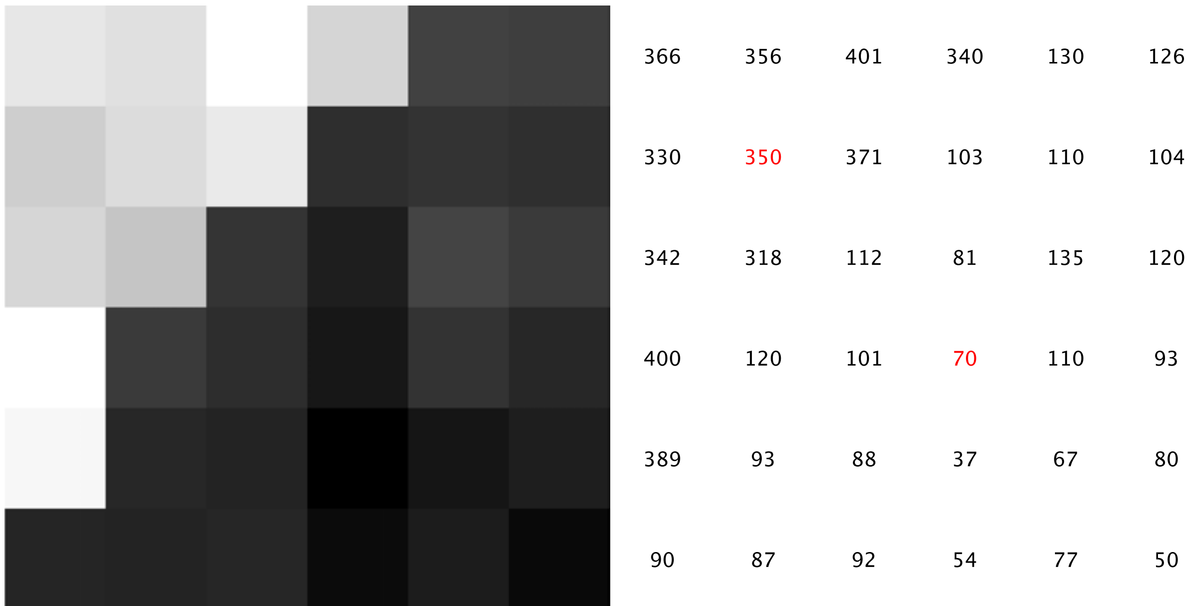
Jedem Voxel werden Anfangskosten zugeordnet. Wie diese Kosten berechnet werden und mit welchen Pfadkosten die Voxel initialisiert werden, hängt von der verwendeten Kostenfunktion ab. Im Folgenden wird hierfür der Absolutbetrag der Grauwertdifferenz verwendet, wie es auch in Kapitel 5 der Fall ist. Außerdem sind die Pfadkosten hier *additv*. Will man also die Pfadkosten von Voxel B über das benachbarte Voxel A berechnen, so gilt folgendes:

$$Pfadkosten(B) = Pfadkosten(A) + |Grauwertdifferenz(A, B)|$$

Somit werden die seeds mit 0 und die restlichen Voxel mit $+\infty$ initialisiert.
Nun wird das Voxel v mit den kleinsten Pfadkosten betrachtet. Ausgehend von diesem Voxel werden dessen Nachbarvoxel berechnet und jeweils die Pfadkosten von v zu jedem Nachbar n kalkuliert.
Ist nun der Weg zu einem Nachbar n über v kürzer als die aktuellen Pfadkosten von n, wird v als Vorgänger von n im optimum-path forest gesetzt und die neuen Pfadkosten von n entsprechend aktualisiert. Hat v schon Vorgänger, ist also das Ende eines Pfades und kein seed, so wird der gesamte Pfad von v als Vorgänger von n gesetzt. Dies wird wiederholt bis alle Voxel abgearbeitet sind.

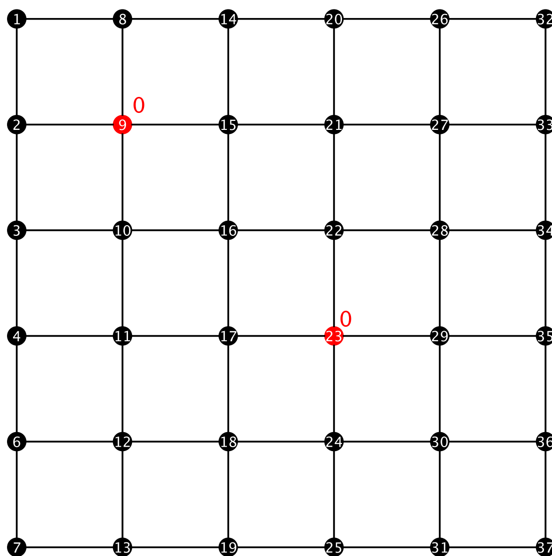
Betrachtet man den Algorithmus etwas genauer, wird schnell klar warum er auch jedem Voxel einen seeds zuordnet. Zur Verdeutlichung werden die ersten Schritte des Algorithmus graphisch dargestellt. Abbildung 4.3 zeigt den Graph eines gegebenen Bildes nach der Initialisierung der IFT. Einfachheitshalber wird hier ein zweidimensionales Bild mit einer 4er-Nachbarschaft verwendet. In nebenstehender Kostentabelle befinden sich die Voxel-Indizes aufsteigend sortiert nach ihren zugehörigen Pfadkosten. Die seeds sind als rote Knoten mit ihren Pfadkosten eingezeichnet, die schwarzen Knoten haben unendliche Pfadkosten und werden in der Kostentabelle übersichtshalber nicht aufgeführt.

Da immer das Voxel mit den geringsten Kosten als nächstes abzuarbeitendes Voxel ausgewählt wird und die seeds zu Beginn mit 0 initialisiert werden, ist das erste ausgewählte Voxel ein seed. Dessen Nachbarn, sofern sie kein weiterer seed sind, haben



(a) Gegebenes Bild.

(b) Grauwerte des Eingabebildes.



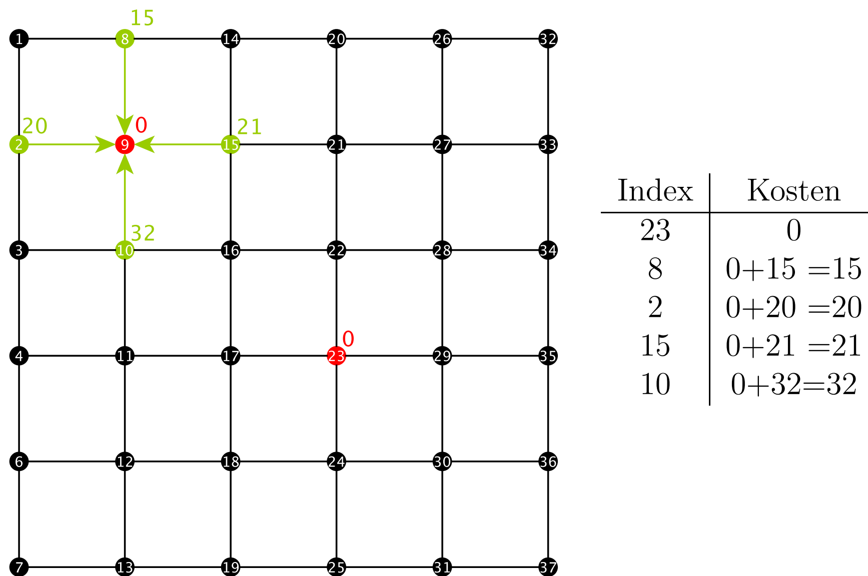
(c) Graph nach der Initialisierung der IFT.

Index	Kosten
9	0
23	0

(d) Kostentabelle

Abbildung 4.3: Beispiel für die IFT im Zweidimensionalen mit einer 4er-Nachbarschaft.

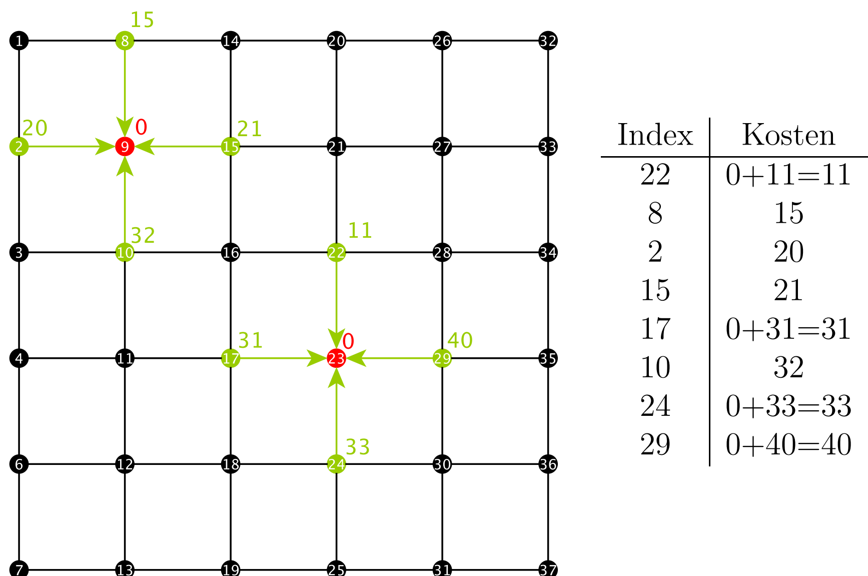
Pfadkosten von $+\infty$, wodurch der Weg zu allen Nachbarn über den seed der kürzeste ist. Somit wird jedem Nachbar der seed als Vorgänger zugewiesen und es werden die Pfadkosten aller Nachbarn neu berechnet. Da der seed Pfadkosten von 0 hat sind die neuen Pfadkosten der Nachbarn die Absolutbeträge der Grauwertdiffernzen zwischen seed und Nachbar (Abb. 4.4).
Nun werden alle weiteren seeds abgearbeitet, da sie mit 0 stets die geringsten Kosten



Index	Kosten
23	0
8	0+15 =15
2	0+20 =20
15	0+21 =21
10	0+32=32

Abbildung 4.4: Zuerst wird einer der rot gefärbten seeds betrachtet und die Kosten der Nachbarn in grün aktualisiert. Der seed wird als Vorgänger aller Nachbarn gesetzt.

haben (Abb. 4.5).

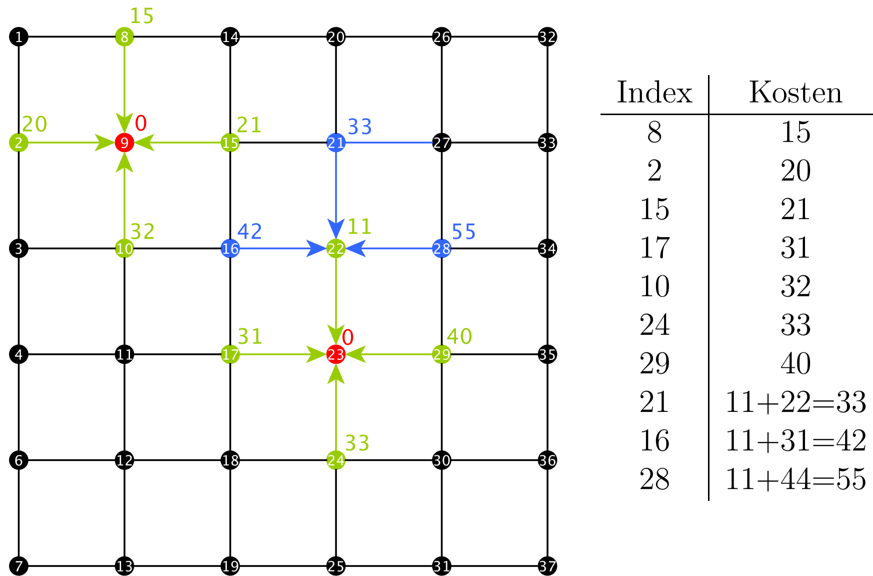


Index	Kosten
22	0+11=11
8	15
2	20
15	21
17	0+31=31
10	32
24	0+33=33
29	0+40=40

Abbildung 4.5: Der Zweite seed (23) wird betrachtet und dessen Nachbarn aktualisiert.

Hat der Algorithmus alle seeds durchlaufen, ist das nächste abzuarbeitende Voxel s ein Nachbar eines seeds, da alle anderen Voxel immer noch Kosten von $+\infty$ haben. Nun wird der Nachbar mit dem kürzesten Weg zu s gesucht und diesem Nachbar wird

s wieder als Vorgänger zugeordnet. Da s aber das Ende eines Pfades mit einem seed als Startpunkt ist, wird somit dem Nachbar auch der seed zugeordnet (Abb. 4.6). Alle



Index	Kosten
8	15
2	20
15	21
17	31
10	32
24	33
29	40
21	11+22=33
16	11+31=42
28	11+44=55

Abbildung 4.6: Nachbar Nr. 22 wird betrachtet, da er die kleinsten Kosten hat und die Kosten seiner Nachbarn werden aktualisiert.

Voxel mit Kosten $\neq +\infty$ haben somit einen seed als Pfadbeginn.

4.3 Optimierungsmöglichkeiten

Damit die Pfadkosten eines Voxels nicht jedes Mal neu berechnet werden müssen, sind sie in einer Kostentabelle gespeichert, die bei einem günstigeren Pfad aktualisiert wird. Außerdem kann das Finden von seeds auch von einem separaten Algorithmus vorgenommen werden, welcher seine Ergebnisse an den IFT-Algorithmus weitergibt, falls Benutzerinteraktion nicht gewünscht oder möglich ist.

5 Algorithmus

Im Folgenden wird ein MATLAB-Algorithmus zur Segmentierung von CT-Scans vorgestellt. Anschließend wird die Funktionsweise diskutiert und deren Stärken und Schwächen gezeigt.

5.1 3D-Implementierung

Das Grundgerüst für die Implementierung bilden die beiden Dateien *ift.m* und *getAdjacency.m*.

5.1.1 Directions.m

Das Enum *Directions.m* wird in *getAdjacency.m* benötigt um die Koordinaten für alle Nachbarvoxel zu berechnen. Es beinhaltet alle Richtungsvektoren für eine 26er Nachbarschaft.

```
classdef Directions
```

Als Eigenschaften eines Direction-Objekts werden x-, y- und z-Richtung definiert.

```
    properties
        xDir
        yDir
        zDir
    end
```

Die Klasse beinhaltet einen Konstruktor und die Methode *getVector(obj)* mit der man die Richtungsvektoren für eine gegebene Richtung erhält.

```
    methods
        % Constructor
        function obj = Directions(x, y, z)
            obj.xDir = x; obj.yDir = y; obj.zDir = z;
        end

        % Get coordinates of vector
        function [X, Y, Z] = getVector(obj)
```

```
9            X = obj.xDir;
10           Y = obj.yDir;
11           Z = obj.zDir;
12       end
13   end
```

Die 26 Richtungsvektoren sind nach folgendem Schema definiert.

```
1    enumeration
2        B(0,0,-1)      % Back
3        F(0,0,1)       % Front
4
5        NWB(-1,-1,-1) % North West Back
6        NW(-1,-1,0)   % North West
7        NWF(-1,-1,1)  % North West Front
8
9        NB(0,-1,-1)   % North Back
10       N(0,-1,0)     % North
11       NF(0,-1,1)    % North Front
12
13       NEB(1,-1,-1)  % North East Back
14       NE(1,-1,0)    % North East
15       NEF(1,-1,1)   % North East Front
```

Die vollständige Liste der Richtungsvektoren befindet sich im Anhang (A.1).

5.1.2 getAdjacency.m

Die Datei *getAdjacency.m* erstellt, für ein gegebenes Bild, eine Adjazenzmatrix, welche als Eingabeparameter für die IFT benötigt wird.

Da CT-Scans sehr viele Voxel beinhalten können und somit die zugehörige Adjazenzmatrix potenziell sehr groß werden kann, muss schon zu Beginn auf eine effiziente Speicherung geachtet werden. Ein Bild mit n Voxeln benötigt in einer normalen Adjazenzmatrix n^2 viel Speicher, wenn für jedes der n Voxel die Kanten zu allen n Voxeln eingetragen werden. Jedoch betrachtet man hier für jedes Voxel nur seine 26 Nachbarn, weshalb lediglich $26n$ Einträge, von den n^2 Einträgen, von 0 verschieden sind. Eine solche Matrix mit sehr vielen "Null-Einträgen" nennt man *dünnbesetzte Matrix* oder *schwachbesetzte Matrix* (Def. 3.1).
Um uns das zu verdeutlichen, ein kleines Rechenbeispiel: Angenommen wir haben einen 16x16x16 großen CT-Block. Das wären insgesamt 4.096 Voxel und die entsprechende Adjazenzmatrix hätte $4.096^2 = 16.777.216$ Einträge. Davon enthalten aber nur $4.096 \cdot$

26 = 106.496 Einträge für uns relevante Informationen, in den restlichen steht eine Null. Also ist die Matrix nur zu ca. 0,63% belegt.
Diesen Ballast an Nullen, der uns keinen wirklichen Mehrwert bietet und somit unnötig viel Platz und Rechenkapazität verbraucht, wollen wir nicht mitspeichern. Dafür bietet sich in Matlab das Matrizen-Format *sparse* an, welches, im Gegensatz zum normalen Matrix-Format, nur Einträge speichert, die von Null verschieden sind. Auch Adjazenzlisten würden sich hier gut eignen.

Nachdem nun ein geeignetes Datenformat zur Speicherung der Adjazenzmatrix ausgewählt wurde, wird nun erläutert wie die Adjazenzmatrix erstellt wird:

Als Eingabe wird nur das zu segmentierende Bild, in Form einer dreidimensionalen Matrix, benötigt.

```
1 function adja = getAdjacency(data)
```

Zuerst wird ein Enum erstellt, mit dem man Richtungsvektoren erhalten kann (siehe 5.1.1). Außerdem werden aus Performanzgründen zwei Hilfs-Matrizen initialisiert, aus denen später die sparse Adjazenzmatrix erstellt wird.

```
1 directions = enumeration('Directions');
2
3 AllVoxel = [];
4 AllNeighbors =[];
```

Ziel ist es nun die Hilfsmatrizen so zu befüllen, dass die nebeneinander-Konkatenation der Matrizen alle benachbarten Knotenpaare enthält:

AllVoxel	AllNeighbors
voxel1	neighbor1.1
voxel1	neighbor1.2
voxel1	neighbor1.3
voxel1	neighbor1.4
...	...
voxel2	neighbor2.1
voxel2	neighbor2.2
...	...

Danach erfolgt eine Iteration über den linearen Index aller Voxel, in der zunächst die Koordinaten des aktuellen Voxels gespeichert und ein Array *currentNeighbors* initialisiert wird, in der später die Nachbarn des aktuell betrachteten Voxels zwischengespeichert werden.

```
% Iterate over linear index of all voxels
for currentInd = 1:numel(data)

    % Get coordinates of current voxel
    [x,y,z] = ind2sub(size(data),currentInd);

    % Create empty list of neighbors for current voxel
    currentNeighbors = [];
```

Um nun alle Nachbarvoxel des aktuellen Voxels zu erhalten, werden mithilfe des *Direction*-Enums sämtliche Richtungsvektoren berechnet, die auf einen Nachbarn zeigen und dieser dann, falls die Koordinaten existieren, in *currentNeighbors* zwischengespeichert.

```
    % Iterate over all possible neighbor coordinates
    for i = 1:size(directions,1)

        % Get coordinates of neighbor
        [x1, y1, z1] = getVector(directions(i));

        % Check if indices of neighbor are out of range
        if(x+x1>0 && x+x1 <= size(data,1) && y+y1>0 && y+y1 <= size(data
    ,2) && z+z1>0 && z+z1 <= size(data,3))

            % Add linear index of neighbor
            currentNeighbors = [currentNeighbors; sub2ind(size(data),x+x1
    ,y+y1,z+z1)];
        end
    end
```

Im nächsten Schritt werden die Hilfsmatrizen so befüllt, dass in *AllVoxel* das aktuelle Voxel so oft eingefügt wird, wie es Nachbarn besitzt und in *AllNeighbors* alle Nachbarn dieses Voxels angehängt werden.

```
    % Add current voxel
    AllVoxel = [AllVoxel; transpose(repelem(currentInd,size(
    currentNeighbors,1)))];

    % Add neighbors of current voxel
    AllNeighbors = [AllNeighbors;currentNeighbors];
end
```

Zuletzt wird aus den Hilfsmatrizen eine sparse Adjazenzmatrix erstellt, welche benachbarte Voxelpaare mit 1 speichert und nicht benachbarte Voxelpaare nicht speichert.

```
% Create a sparse adjacency matrix
adja = sparse(AllVoxel,AllNeighbors,1);
end
```

Die fertige sparse Adjazenzmatrix sieht nun so aus:

$$\begin{matrix} (\text{voxel1} & \text{neighbor1.1}) & 1 \\ (\text{voxel1} & \text{neighbor1.2}) & 1 \\ (\text{voxel1} & \text{neighbor1.3}) & 1 \\ (\text{voxel1} & \text{neighbor1.4}) & 1 \\ \dots & \dots & \dots \\ (\text{voxel2} & \text{neighbor2.1}) & 1 \\ (\text{voxel2} & \text{neighbor2.2}) & 1 \\ \dots & \dots & \dots \end{matrix}$$

5.1.3 ift.m

Die Datei *ift.m* führt die eigentliche Segmentierung durch.

Zuerst sollte man sich noch einmal ins Gedächtnis rufen, was wir mit einer Segmentierung erreichen wollen. Wir ordnen jedem Voxel ein Objekt zu, welches in unserem Fall durch ein Label dargestellt wird. Diese Information können wir aus einem optimum-path forest ablesen, welcher eine Liste aller Voxel enthält und für jedes Voxel ein Pfad seiner Vorgänger bis zu einem seed point gespeichert ist.

Als Eingabe benötigt der Algorithmus das Bild in Form einer 3D-Matrix, die zugehörige sparse Adjazenzmatrix und ein Array das vom User gewählte seeds und dazugehörige Label beinhaltet. Zurückgegeben wird ein optimum-path forest und eine Kostentabelle aller Voxel.

```
function [optPathForest, costTable] = ift(data, adja, seeds)
```

Zuerst wird eine Matrix *Queue* erstellt in der die linearen Indizes aller Voxel der 3D-Matrix eingefügt werden. Außerdem wird der optimum-path forest als leeres Cell Array initialisiert.

```
N = numel(data);

% Store linear indices of data in Q
Queue = 1:numel(data);
Queue = transpose(Queue);
```

```
6
7 % Prelocate space for predecessor map
8 optPathForest = cell(N,1);
```

Dann wird eine Kostentabelle erstellt, in der alle Pfadkosten jedes Voxels zum seed festgehalten werden. Anfangs haben alle Voxel unendliche Kosten, bis auf die seed Voxel, welche mit 0 minimale Kosten haben. Im Algorithmus werden nur positive Kosten verwendet.

```
1 % Init cost table
2 costTable = Inf(N,1);
3 costTable(seeds(:,1)) = 0;
```

Für eine bessere Performanz werden jedem Voxel in der *Queue* auch seine Kosten hinzugefügt. *Key* wird benötigt um spätere Suchen effizienter zu gestalten.

```
1 % Add costs to Q
2 Queue = [Queue costTable];
3
4 Key = 1:numel(data);
```

Solange sich in unserer "Warteschlange" *Queue* Voxel befinden, wird das Voxel v mit den kleinsten Pfadkosten aus der Warteschlange entfernt und sein linearer Index in *currentInd* gespeichert. Sämtlicher nachfolgender Code wird für jedes Voxel ausgeführt bis *Queue* leer ist, also jedes Voxel abgearbeitet ist.
Da zu Beginn *Queue* mit allen Voxeln befüllt wurde und die seeds minimale Pfadkosten haben, sind dies die ersten Voxel die entfernt werden. Anschließend werden die linearen Indizes aller Nachbarvoxel von v in *neighbors* gespeichert.

```
1  % While Q not empty
2  while ~isempty(Queue)
3
4      % Get voxel with smallest cost from Q
5      [~,QInd] = min(Queue(:,2));
6
7      % Get index of current voxel
8      currentInd = Queue(QInd,1);
9
10     % Remove voxel with smallest cost from Queue
11     Queue(QInd,:) = [];
12
13     % Get neighbors of voxel
14     [neighbors, ~,~] = find(adja(:,currentInd));
```

Für jeden Nachbar von v wird der Index in der *Queue*-Matrix ermittelt falls er dort noch vorhanden ist.

```
1     % For every neighbor of voxel
2     for n = 1:size(neighbors,1)
3
4         % Get linear index of neighbor
5         neighborInd = neighbors(n);
6
7         % Search Index of neighbor in Queue
8         QNeighborInd = Key(Queue(:,1)== neighborInd);
```

Ist der Nachbar n nicht in *Queue* so wird er nicht weiter betrachtet, sondern der nächste Nachbar von v ermittelt.

Ist er vorhanden, so werden die Pfadkosten zu n über v berechnet und das Ergebnis in *tmpCost* festgehalten.

Nun wird verglichen ob der Pfad zu n über v kürzer ist als die vorherigen Pfadkosten von n. Wenn dies der Fall ist, wird der Pfad zu v als Vorgänger von n im optimum-path forest gesetzt. Ist v ein seed, so ist v der alleinige Vorgänger. Außerdem werden die Kosten in *costTable* und die Kosten von n in *Queue* zu den neuen, niedrigeren Kosten aktualisiert.

```
1         % If neighbor is in Q
2         if  ~isempty(Queue(QNeighborInd))
3
4             % Calculate costs from neighbor via voxel
5             tmpCost = abs(double(data(currentInd)) - double(data(
    neighborInd)));
6
7             % If the path to neighbor via voxel is shorter than current
8             % path to neighbor
9             if tmpCost <= costTable(neighborInd)
10
11                % Update predecessor to current voxel
12                optPathForest{neighborInd} =  [currentInd  optPathForest{
    currentInd}];
13
14                % Update neighbor's cost in costTable
15                costTable(neighborInd) = tmpCost;
16
17                % Update neighbor's cost in queue
18                Queue(QNeighborInd,2)= tmpCost;
19            end
```

```
        end
    end
end
```

5.1.4 showSegment.m

Mit der Datei *showSegment.m* können einzelne Segmente eines ausgewählten Labels graphisch dargestellt werden.

Dazu werden das Ausgangsbild als 3D-Matrix, der optimum-path forest aus der IFT, die seed points und ein Label als Eingabe benötigt.

```
function showSegment(data,p,seeds,label)
```

Das Eingabebild wird kopiert und die seeds mit dem ausgewählten Label werden gespeichert.

```
segment= data;

% Get seeds with chosen label
labelSeeds = seeds(seeds(:,2) == label,1);
```

Nun wird über alle Voxel iteriert und alle, die nicht dem ausgewählten Label entsprechen werden auf 0 gesetzt und somit nicht angezeigt.

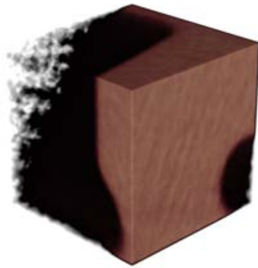
```
% Set voxels with different labels to 0
for i = 1:numel(segment)
    if ~isempty(p{i,1}) && ~ismember(p{i,1}(end), labelSeeds, 'rows')
        segment(i) = 0;
    elseif isempty(p{i,1}) && ismember(i,seeds(:,1)) && ~ismember(i,
   labelSeeds)
        % Set seeds with different labels to 0
        segment(i) = 0;
    end
end
```

Optional wird eine Colormap erstellt.

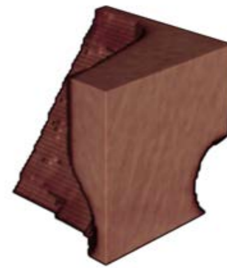
```
% Create optional colormap
intensity = [-3024,-16.45,641.38,3071];
color = ([0 0 0; 186 65 77; 231 208 141; 255 255 255]) ./ 255;
queryPoints = linspace(min(intensity),max(intensity),256);
colormap = interp1(intensity,color,queryPoints);
```

Zuletzt wird das Segment ausgegeben und mit seinem Label beschriftet.

```
% Display segment
figure('Name',['Label ', num2str(label)],'NumberTitle','off');
volshow(segment,'Colormap',colormap,'BackgroundColor', [1 1 1]);
```



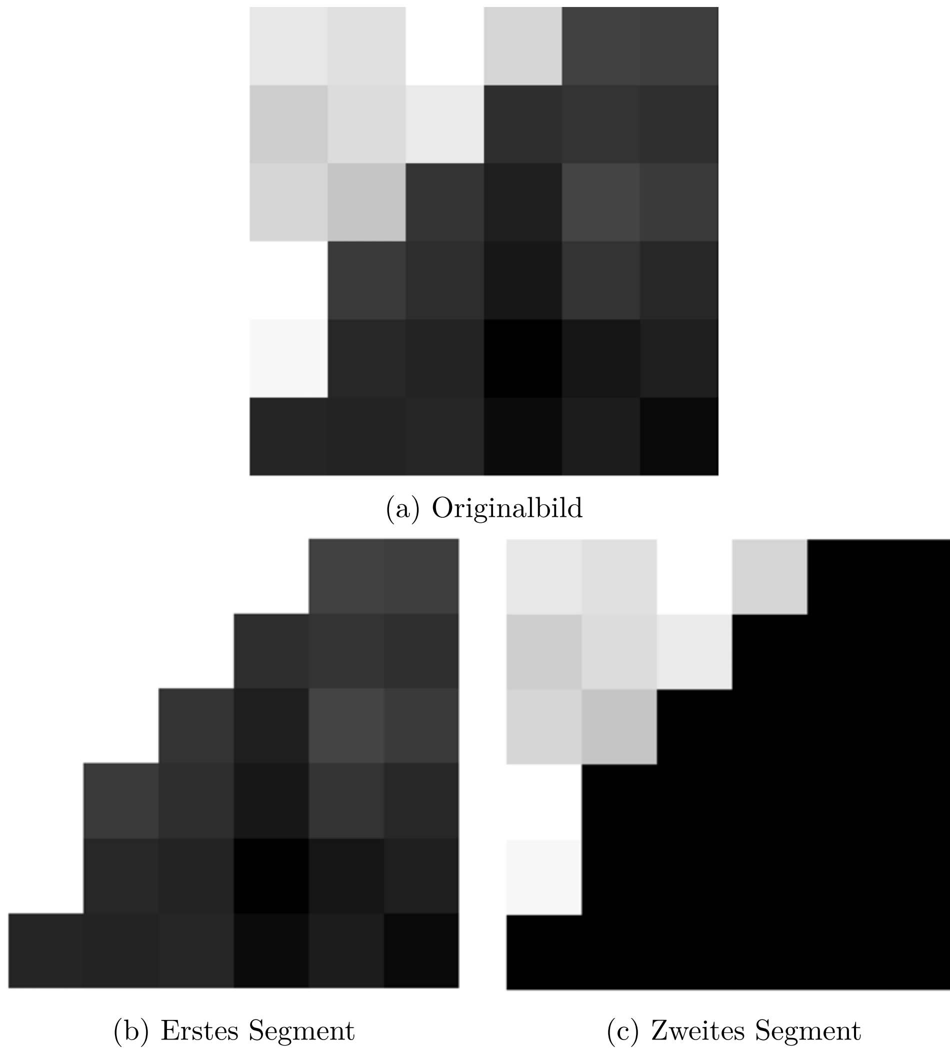
(a) Originalbild



(b) Segment mit dem Label '2'

5.2 2D-Implementierung

Der Algorithmus ist auch für zweidimensionale Bilder geeignet. Am Code von *ift.m* ändert sich hierdurch nichts, nur muss statt einem 3D-Bild ein 2D-Bild mit zugehöriger Adjazenzmatrix für zweidimensionale Bilder übergeben werden. Auch die Methode mit der die Segmente angezeigt werden muss überschrieben werden, da die darin verwendete Funktion *volshow()* nur auf 3D-Bildern angewendet werden kann.
Somit wurden die Dateien *Directions.m*, *getAdjacency.m* und *showSegment.m* angepasst. Da diese Modifikationen klein und intuitiv sind, wird an dieser Stelle auf den Anhang verwiesen (siehe A.6, A.7, A.8).
In *Directions2D.m* und *getAdjacency.m* wurde lediglich die dritte Koordinate entfernt, sodass nun eine 8er-Nachbarschaft ausgegeben wird.
In *showSegment.m* wurde die Funktion *volshow()* durch *imshow()* ersetzt, um 2D-Bilder anzeigen zu können. Außerdem werden die nicht zum gewählten Segment gehörigen Voxel nicht auf 0 gesetzt, sondern auf den Maximalpixelwert des Bildes, sodass sie weiß erscheinen 5.2b. Um sie schwarz erscheinen zu lassen, muss die Variable *invisible* auf *min(data(:))* gesetzt werden 5.2c. Bei Grauwertbildern kann es vorkommen, dass weder schwarz noch weiß eine geeignete Farbe für "ausgeblendete" Pixel ist. Ist dies der Fall, so kann statt *imshow()* auch *image(segment, 'AlphaData', segment)* verwendet werden, wobei dann die Variable *invisible* auf *0* gesetzt sein muss.



(a) Originalbild

(b) Erstes Segment

(c) Zweites Segment

Abbildung 5.2: IFT für 2D-Bilder

6 Analyse

In diesem Kapitel wird obiger Segmentierungs-Algorithmus genauer betrachtet und auf Schwächen und Stärken getestet.

6.1 Wahl der seed points

Die seed points sind für den User die größten Stellschrauben am Algorithmus. Je nach Anzahl und Position können sie das Ergebnis deutlich verbessern. Damit alles reibungslos funktioniert, müssen mindestens so viele seeds wie Segmente gewählt werden und alle seeds desselben Objekts mit dem gleichen Label versehen werden. Das Setzen von mehreren seeds auf demselben Segment, sofern sie das gleiche Label besitzen, ist vor allem bei größeren Dateien und Objekten sinnvoll, wie später gezeigt wird.

6.2 Einfluss der Bildgröße

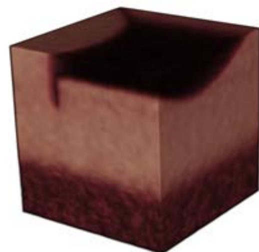
Sind die Dateien sehr groß und somit die Pfadkosten potenziell größer, ist es ratsam diese Kosten im *double* Format und nicht in Formaten wie *uint16* zu speichern. Bei *uint16* beispielsweise ist die größte Zahl, die man in diesem Format speichern kann, 65535. Werden dann auf großen Segmenten wenige seeds gesetzt, und somit die Pfade immer länger, kann es vorkommen, dass dieser Maximalwert erreicht wird. Das hat zur Folge, dass ab diesem Punkt alle folgenden Pfadkosten falsch berechnet werden, da z.B. im Falle von uint16 gilt $65535 + i = 65535$ für ein beliebiges $i \in \mathbb{N}$. Dadurch haben alle folgenden Knoten Kosten von 65535, wodurch die kürzesten Wege nicht mehr korrekt berechnet werden können. Dem entgegenwirken kann man natürlich auch durch das Setzen von entsprechend vielen seeds, damit die Pfade kurz bleiben, was jedoch sehr zeitaufwändig sein kann.

6.3 Laufzeit

Die Laufzeit des Algorithmus ist von der Bildgröße abhängig. Um herauszufinden in welchem Verhältnis die beiden Größen stehen, wurden einige Messungen mit verschiede-

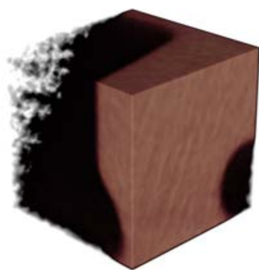
nen Bildgrößen getätigt.
Als Referenzmodell dient ein MacBook Pro 15" (Mitte 2015) mit 2,2 GHZ Quad-Core Intel Core i7, 16 GB RAM und der Grafikkarte Intel Iris Pro 1536 MB.

Folgende 3D-Bilder wurden verwendet:

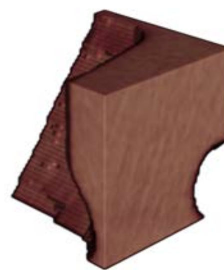


(a) Bild 1

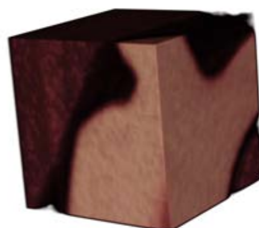
(b) Hauptsegment von Bild 1



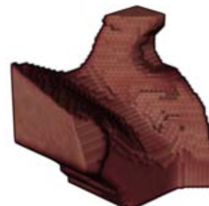
(c) Bild 2



(d) Hauptsegment von Bild 2



(e) Bild 3



(f) Hauptsegment von Bild 3

Abbildung 6.1: Verwendete Beispielmatrizen.

Obwohl die Laufzeit für *getAdjacency.m* nicht von den Grauwerten sondern ausschließlich von der Bildgröße abhängt, wurden dennoch verschiedene Bilder getestet (Tabelle 6.1). Bilder mit denselben Abmessungen produzieren auch die gleiche Adjazenzmatrix.

Um zu prüfen wie sich die Laufzeit von *ift.m* während eines Durchlaufs verändert, wurde immer nachdem 5000 neue Voxel abgearbeitet wurden, die verstrichene Zeit gemessen (Tabelle 6.2). Für die Messungen wurde Bild 6.1c verwendet.

#Voxel	Abmessungen	Bild	Zeit
10.000	10x20x50	-	3,608 s
10.000	10x20x50	-	3,517 s
10.000	10x20x50	-	3,825 s
25.000	50x50x10	-	9,020 s
25.000	50x50x10	-	8,938 s
25.000	50x50x10	-	8,777 s
50.000	50x50x50	1	55,430 s
50.000	50x50x50	2	47,998 s
50.000	50x50x50	3	47,543 s
1.000.000	100x100x100	-	381,745 s

Tabelle 6.1: Laufzeit des getAdjacency.m

#Voxel	Abmessungen	Bild	# seeds	Zeit
10.000	10x20x50	-	2	2,106 s
10.000	10x20x50	-	2	1,964 s
10.000	10x20x50	-	2	1,852 s
25.000	50x50x10	-	2	8,659 s
25.000	50x50x10	-	2	8,595 s
25.000	50x50x10	-	2	8,594 s
50.000	50x50x50	1	3	165,749 s
50.000	50x50x50	1	3	190,412 s
50.000	50x50x50	2	5	172,247 s
50.000	50x50x50	2	5	188,770 s
50.000	50x50x50	3	4	172,617 s
50.000	50x50x50	3	4	172,364 s
50.000	50x50x50	3	4	172,364 s
50.000	50x50x50	Abb. 6.3a	2	169,233 s
50.000	50x50x50	Abb. 6.3a	2	169,701 s

Tabelle 6.2: Laufzeit des ift.m

# abgearbeitete Voxel	Zeit insgesamt	Zeitdifferenz
5.000	15,35 s	15,35 s
10.000	28,68 s	13,33 s
15.000	40,44 s	11,76 s
20.000	51,79 s	11,35 s
25.000	62,23 s	10,44 s
30.000	72,69 s	10,46 s
35.000	83,24 s	10,55 s
40.000	93,20 s	9,96 s
45.000	102,38 s	9,18 s

50.000	110,82 s	8,44 s
55.000	118,82 s	7,99 s
60.000	126,02 s	7,21 s
65.000	132,52 s	6,50 s
70.000	138,77 s	6,25 s
75.000	144,34 s	5,57 s
80.000	149,46 s	5,12 s
85.000	154,12 s	4,66 s
90.000	158,36 s	4,24 s
95.000	162,04 s	3,68 s
100.000	165,17 s	3,13 s
105.000	167,80 s	2,63 s
110.000	169,86 s	2,06 s
115.000	171,49 s	1,63 s
120.000	172,74 s	1,25 s
125.000	173,34 s	0,60 s

Tabelle 6.3: Laufzeit des ift.m mit Bild 6.1c

Wie man in Abb. 6.2 gut erkennen kann wird der Algorithmus schneller, je weiter er fortgeschritten ist. Das liegt daran, dass sich mit größerem Fortschritt weniger Voxel in der Queue befinden. Dadurch werden zeitintensive Such-Operationen beschleunigt, da sie nun weniger Voxel durchsuchen müssen und auch Änderungsoperationen, wie Löschen oder Aktualisierungen, gehen nun schneller von statten. Wird eine Zeile in einer Matrix gelöscht, so werden alle nachfolgenden Einträge "nachgerückt". Sind weniger Voxel in der Matrix müssen auch weniger "nachgerückt" werden. Auch mit anderen Bildgrößen ergibt sich ein ähnlicher Graph.

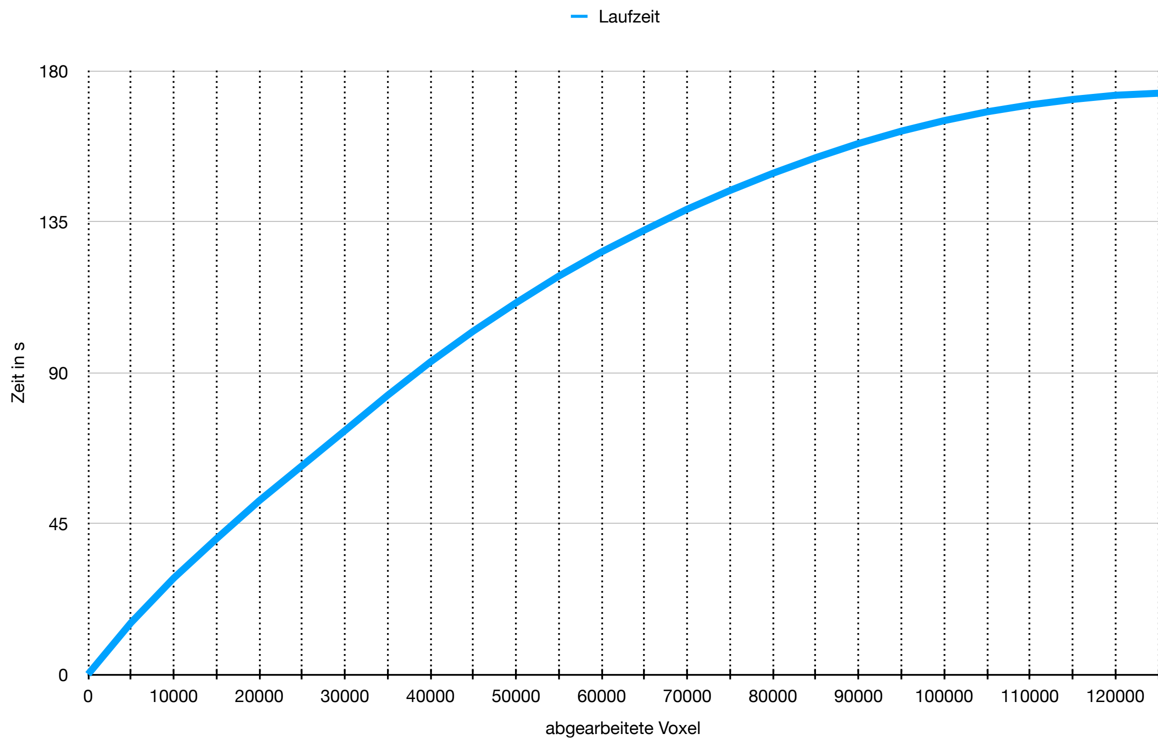


Abbildung 6.2: Graph zu Tabelle 6.3

6.4 Stärken & Schwächen

Um zu testen wie gut der Algorithmus feine Strukturen erkennt, wird ein 50x50x50 Würfel mit zufälligen Grauwerten, im Intervall zwischen 100 und 300, erstellt. Danach werden drei Achsen, ähnlich denen in einem dreidimensionalen Koordinatensystem, in den Würfel gelegt, wobei jede Achse Abmessungen von 1x3x50 Voxeln hat. Der Schnittpunkt der drei Achsen und ein Voxel im Würfel werden als seed points gewählt. In den folgenden Beispielen bleibt das Grauwertintervall des Würfels gleich, während sich das der Achsen mit jedem Bild etwas mehr an das des Würfels annähert.

Bei genauerer Betrachtung merkt man, dass die Oberfläche in Abb. 6.3a glatter ist als die der übrigen Achsenkonstruktionen, also besser segmentiert wurde, und dass diese Glätte mit geringerem Abstand zum Grauwertintervall des Würfels abnimmt. Dies ist nicht verwunderlich da der Algorithmus die Zugehörigkeit zum Objekt anhand der Grauwertdifferenz zu einem Objektvoxel auswählt. Dadurch gilt: Je weiter die Grauwertintervalle von angrenzenden Segmenten auseinanderliegen und je kleiner die Intervalle sind, desto klarer wird segmentiert. Trotz der Tatsache dass in Abbildung 6.3d die Grauwertintervalle mit 200 (Würfel) bzw. 100 (Achsen) Stufen im Vergleich

(a) Extrahierte Achsen mit Grauwerten von 800 bis 1000

(b) Extrahierte Achsen mit Grauwerten von 500 bis 800.

(c) Extrahierte Achsen mit Grauwerten von 400 bis 600

(d) Extrahierte Achsen mit Grauwerten von 350 bis 450

Abbildung 6.3: Segmentierungsergebnisse mit verschiedenen Grauwertintervallen.

zur Distanz der Intervalle mit 50 deutlich größer sind, wurden die Achsen trotzdem gut segmentiert.

In Abb. 6.4a hingegen, in denen beide Grauwertintervalle 200 Stufen groß sind, die Distanz aber nur 50 beträgt, wird das Objekt nur noch teilweise segmentiert. Diese unvollständige Segmentierung kann man etwas kompensieren, indem man mehrere seeds auf den Achsen platziert, sodass die Pfade kürzer werden (Abb. 6.4b).

(a) Extrahierte Achsen mit Grauwerten von 350 bis 550 (2 seeds).

(b) Extrahierte Achsen mit Grauwerten von 350 bis 550 (8 seeds).

Abbildung 6.4: Verbesserung des Ergebnisses durch mehr seeds.

7 Fazit

Im Rahmen dieser Bachelorarbeit wurde ein graphbasierter Algorithmus nach Vorbild der *Image Foresting Transform* implementiert. Ziel dieser Arbeit war es die Funktionsweise eines solch graphbasierten Segmentierungsalgorithmus auf dreidimensionalen Bildern verständlich darzulegen.
Dazu wurden zunächst Grundlagen in den Bereichen Segmentierung, Computertomographie und Mathematik vermittelt. Es wurden verschiedene Segmentierungstechniken vorgestellt und gezeigt wie ein CT-Scan, ein klassisches Segmentierungsobjekt, in der Praxis aufgenommen wird. In Kapitel 4 wurde näher auf das Prinzip der graphbasierten Segmentierung eingegangen, bevor es in Kapitel 5 in einem Algorithmus angewandt wurde. Abschließend wurde der Algorithmus analysiert und auf mehreren 3D-Bildern getestet.
Wie aus den Analysen hervorgegangen ist, eignet sich ein graphbasierter Ansatz gut zur Segmentierung. Durch die Wahl der seeds kann der User den Algorithmus individuell, nach Bildbeschaffenheit, anpassen und ist nicht an feste Schwellwerte gebunden. Somit können nicht nur große Objekte, sondern auch feine Strukturen zuverlässig herausgearbeitet werden.

Anhang: Quellcode

Im Folgenden befindet sich der MATLAB-Quellcode für die in Kapitel 5 beschriebenen Algorithmen und eine Demo-Version.

A.1 Directions.m

```
classdef Directions
    % Contains direction vectors for neighbors of a 26-connected voxel
    % adjacency graph.

    properties
        xDir
        yDir
        zDir
    end


    methods
        % Constructor
        function obj = Directions(x, y, z)
            obj.xDir = x; obj.yDir = y; obj.zDir = z;
        end

        % Get coordinates of vector
        function [X, Y, Z] = getVector(obj)
            X = obj.xDir;
            Y = obj.yDir;
            Z = obj.zDir;
        end
    end


    enumeration
        B(0,0,-1)      % Back
        F(0,0,1)       % Front

        NWB(-1,-1,-1) % North West Back
        NW(-1,-1,0)   % North West
        NWF(-1,-1,1)  % North West Front
```

```
34
35         NB(0,-1,-1)   % North Back
36         N(0,-1,0)     % North
37         NF(0,-1,1)    % North Front
38
39         NEB(1,-1,-1)  % North East Back
40         NE(1,-1,0)    % North East
41         NEF(1,-1,1)   % North East Front
42
43         EB(1,0,-1)    % East Back
44         E(1,0,0)      % East
45         EF(1,0,1)     % East Front
46
47         SEB(1,1,-1)   % South East Back
48         SE(1,1,0)     % South East
49         SEF(1,1,1)    % South East Front
50
51         SB(0,1,-1)    % South Back
52         S(0,1,0)      % South
53         SF(0,1,1)     % South Front
54
55         SWB(-1,1,-1)  % South West Back
56         SW(-1,1,0)    % South West
57         SWF(-1,1,1)   % South West Front
58
59         WB(-1,0,-1)   % West Back
60         W(-1,0,0)     % West
61         WF(-1,0,1)    % West Front
62     end
63 end
```

A.2 getAdjacency.m

```
1  function adja = getAdjacency(data)
2  % Creates a sparse adjacency matrix for the given data of the form:
3  % (node1,node2) = 1
4  % The nodes are stored by their linear indices
5
6  % Input:
7  %   data : a 3D matrix
8  %
9  % Output:
10 %   adja : a sparse adjacency matrix
11
```

```
12
13 directions = enumeration('Directions');
14
15 AllVoxel = [];
16 AllNeighbors =[];
17
18 % Iterate over linear index of all voxels
19 for currentInd = 1:numel(data)
20
21     % Get coordinates of current voxel
22     [x,y,z] = ind2sub(size(data),currentInd);
23
24     % Create empty list of neighbors for current voxel
25     currentNeighbors = [];
26
27     % Iterate over all possible neighbor coordinates
28     for i = 1:size(directions,1)
29
30         % Get coordinates of neighbor
31         [x1, y1, z1] = getVector(directions(i));
32
33         % Check if indices of neighbor are out of range
34         if(x+x1>0 && x+x1 <= size(data,1) && y+y1>0 && y+y1 <= size(data
    ,2) && z+z1>0 && z+z1 <= size(data,3))
35
36             % Add linear index of neighbor
37             currentNeighbors = [currentNeighbors; sub2ind(size(data),x+x1
    ,y+y1,z+z1)];
38         end
39     end
40     % Add current voxel
41     AllVoxel = [AllVoxel; transpose(repelem(currentInd,size(
    currentNeighbors,1)))];
42
43     % Add neighbors of current voxel
44     AllNeighbors = [AllNeighbors;currentNeighbors];
45 end
46
47 % Create a sparse adjacency matrix
48 adja = sparse(AllVoxel,AllNeighbors,1);
49 end
```

A.3 ift.m

```
function [optPathForest, costTable] = ift(data, adja, seeds)
% Executes an image foresting transform on the given data and adjacency
    list with the seeds.

% Input:
%   data : a 2D or 3D matrix
%   adja : a sparse adjacency matrix of the 2D or 3D matrix
%   seeds: an array of seed points with their related labels
%
% Output:
%   optPathForest : an optimum path forest
%   costTable     : a cost table of all voxel


N = numel(data);

% Store linear indices of data in Q
Queue = 1:numel(data);
Queue = transpose(Queue);

% Prelocate space for predecessor map
optPathForest = cell(N,1);

% Init cost table
costTable = Inf(N,1);
costTable(seeds(:,1)) = 0;

% Add costs to Q
Queue = [Queue costTable];

Key = 1:numel(data);

% While Q not empty
while ~isempty(Queue)

    % Get voxel with smallest cost from Q
    [~,QInd] = min(Queue(:,2));

    % Get index of current voxel
    currentInd = Queue(QInd,1);

    % Remove voxel with smallest cost from Queue
    Queue(QInd,:) = [];
```

```
43
44    % Get neighbors of voxel
45    [neighbors, ~,~] = find(adja(:,currentInd));
46
47
48    % For every neighbor of voxel
49    for n = 1:size(neighbors,1)
50
51        % Get linear index of neighbor
52        neighborInd = neighbors(n);
53
54        % Search Index of neighbor in Queue
55        QNeighborInd = Key(Queue(:,1)== neighborInd);
56
57
58        % If neighbor is in Q
59        if  ~isempty(Queue(QNeighborInd))
60
61            % Calculate costs from neighbor via voxel
62            tmpCost = abs(double(data(currentInd)) - double(data(
   neighborInd)));
63
64            % If the path to neighbor via voxel is shorter than current
65            % path to neighbor
66            if tmpCost <= costTable(neighborInd)
67
68                % Update predecessor to current voxel
69                optPathForest{neighborInd} =  [currentInd  optPathForest{
   currentInd}];
70
71                % Update neighbor's cost in costTable
72                costTable(neighborInd) = tmpCost;
73
74                % Update neighbor's cost in queue
75                Queue(QNeighborInd,2)= tmpCost;
76            end
77        end
78    end
79 end
80
81 end
```

A.4 showSegment.m

```
function showSegment(data,p,seeds,label)
% Displays segments with chosen label
%
% Input:
%     data: a 3D matrix
%     seeds: an optimum-path-forest
%     label: label of the segment to be shown

segment= data;

% Get seeds with chosen label
labelSeeds = seeds(seeds(:,2) == label,1);

% Set voxels with different labels to 0
for i = 1:numel(segment)
    if ~isempty(p{i,1}) && ~ismember(p{i,1}(end), labelSeeds, 'rows')
        segment(i) = 0;
    elseif isempty(p{i,1}) && ismember(i,seeds(:,1)) && ~ismember(i,
    labelSeeds)
        % Set seeds with different labels to 0
        segment(i) = 0;
    end
end

% Create optional colormap
intensity = [-3024,-16.45,641.38,3071];
color = ([0 0 0; 186 65 77; 231 208 141; 255 255 255]) ./ 255;
queryPoints = linspace(min(intensity),max(intensity),256);
colormap = interp1(intensity,color,queryPoints);

% Display segment
figure('Name',['Label ', num2str(label)],'NumberTitle','off');
volshow(segment,'Colormap',colormap,'BackgroundColor', [1 1 1]);
end
```

A.5 demo.m

```
%% Segmentation
% Load the image
load('cross_800_1000.mat');

% Create a sparse adjacency list of the image
```

```
adja = getAdjacency(data);

% Set seeds
seeds = [61225 1; 22715 2];

% Get an optimum-path forest and the cost table
[p,costTable] = ift(data, adja, seeds);


%% Optional:

% Only display the segments with a specific label
showSegment(data,p,seeds,1)
```

A.6 Directions2D.m

```
classdef Directions2D
    % Contains direction vectors for neighbors of a 8-connected voxel
    % adjacency graph.

    properties
        xDir
        yDir
    end


    methods
        % Constructor
        function obj = Directions2D(x, y)
            obj.xDir = x; obj.yDir = y;
        end

        % Get coordinates of vector
        function [X, Y] = getVector(obj)
            X = obj.xDir;
            Y = obj.yDir;
        end
    end


    enumeration
        NW(-1,-1)    % North West
        N(0,-1)      % North
        NE(1,-1)     % North East
```

```
        E(1,0)      % East
        SE(1,1)     % South East
        S(0,1)      % South
        SW(-1,1)    % South West
        W(-1,0)     % West
    end
end
```

A.7 getAdjacency2D.m

```
function adja = getAdjacency2D(data)
% Creates a sparse adjacency matrix for the given data of the form:
% (node1,node2) = 1
% The nodes are stored by their linear indices

% Input:
%   data : a 2D matrix
%
% Output:
%   adja : a sparse adjacency matrix


directions = enumeration('Directions2D');

AllVoxel = [];
AllNeighbors =[];

% Iterate over linear index of all voxels
for currentInd = 1:numel(data)

    % Get coordinates of current voxel
    [x,y] = ind2sub(size(data),currentInd);

    % Create empty list of neighbors for current voxel
    currentNeighbors = [];

    % Iterate over all possible neighbor coordinates
    for i = 1:size(directions,1)

        % Get coordinates of neighbor
        [x1, y1] = getVector(directions(i));

        % Check if indices of neighbor are out of range
```

```
34          if(x+x1>0 && x+x1 <= size(data,1) && y+y1>0 && y+y1 <= size(data
    ,2))
35
36              % Add linear index of neighbor
37              currentNeighbors = [currentNeighbors; sub2ind(size(data),x+x1
    ,y+y1)];
38          end
39      end
40      % Add current voxel
41      AllVoxel = [AllVoxel; transpose(repelem(currentInd,size(
    currentNeighbors,1)))];
42
43      % Add neighbors of current voxel
44      AllNeighbors = [AllNeighbors;currentNeighbors];
45  end
46
47  % Create a sparse adjacency matrix
48  adja = sparse(AllVoxel,AllNeighbors,1);
49  end
```

A.8 showSegment2D.m

```
1   function showSegment2D(data,p,seeds,label)
2   % Displays segments with chosen label
3   %
4   % Input:
5   %    data: a 2D matrix
6   %    seeds: an optimum-path-forest
7   %    label: label of the segment to be shown
8
9   segment= data;
10
11  % Set color of "turned of" pixels
12  invisible= max(data(:));
13
14  % Get seeds with chosen label
15  labelSeeds = seeds(seeds(:,2) == label,1);
16
17  % Set pixels with different labels to 0
18  for i = 1:numel(segment)
19      if ~isempty(p{i,1}) && ~ismember(p{i,1}(end), labelSeeds, 'rows')
20          segment(i) = invisible;
21
```

```
22      elseif isempty(p{i,1}) && ismember(i,seeds(:,1)) && ~ismember(i,
        labelSeeds)
23          segment(i) = invisible;
24      end
25  end
26
27  % Display segment
28  figure('Name',['Label ', num2str(label)],'NumberTitle','off');
29  imshow(segment, [min(data(:)) max(data(:))]);
30  end
```

Literaturverzeichnis

[1] Susanne Andreae, Peter Avelini, Melanie Berg, Ingo Blank, and Annelie Burk. *Lexikon der Krankheiten und Untersuchungen*, pages 1286--1288. Thieme, 2 edition, 2006.

[2] A.X. Falcao, J. Stolfi, and R. de Alencar Lotufo. The image foresting transform: theory, algorithms, and applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(1):19--29, 2004. doi:10.1109/TPAMI.2004.1261076.

[3] Heinz Handels. *Medizinische Bildverarbeitung - Bildanalyse, Mustererkennung und Visualisierung für die computergestützte ärztliche Diagnostik und Therapie*, page 12. Springer-Verlag, 2 edition, 2009.

[4] Jens Kürbig and Martina Sauter. Bildsegmentierung: Übersicht, 2006. URL: http://www.mathematik.uni-ulm.de/stochastik/lehre/ws05_06/seminar/ausarbeitung_sauter.pdf.

[5] SKY-PuLeun. Own work. CC BY-SA 4.0. URL: https://commons.wikimedia.org/w/index.php?curid=104995714.

[6] Song Yuheng and Yan Hao. Image segmentation algorithms overview. *ArXiv*, abs/1707.02051, 2017. URL: https://arxiv.org/pdf/1707.02051.pdf.