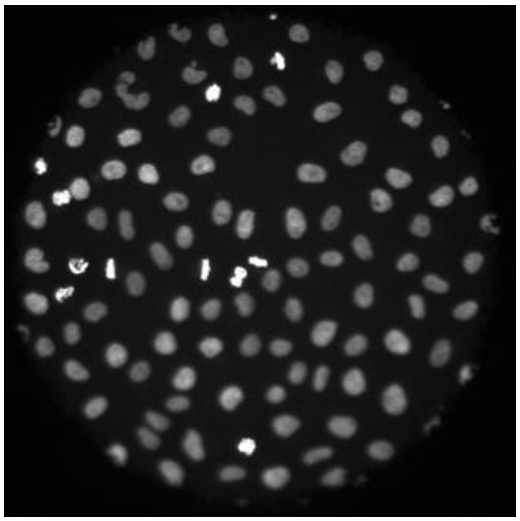
Übung 10 Abgabe 8.7.2014

Aufgabe 1 – Bildverarbeitung mit Graphen 18 Punkte

Gegeben ist das folgende Mikroskop-Bild aus einer Hochdurchsatz-Studie am BioQuant (siehe http://hci.iwr.uni-heidelberg.de/Staff/ukoethe/download/cells.pgm):



Das Bild zeigt Zellkerne, die unterschiedlich gefärbt wurden (grau und weiß). Die Aufgabe besteht darin, die Zellkerne beider Arten zu finden und zu zählen. Für Menschen ist dies im Prinzip kein Problem, aber moderne automatische Mikroskope erzeugen in kurzer Zeit Tausende oder sogar Millionen solcher Bilder, so dass eine manuelle Auswertung nicht mehr möglich ist.

Das Bild ist im PGM-Format abgespeichert und kann z.B. mit den Programmen `gimp` und IrfanView angezeigt werden. PGM-Bilder bieten sich für diese Übung an, weil sie leicht gelesen und gespeichert werden können. Dazu verwenden Sie das Modul http://hci.iwr.uni-heidelberg.de/Staff/ukoethe/download/pgm.py, das Sie am Anfang Ihres Programms einfach importieren können:

```
from pgm import readPGM, writePGM
width, height, data = readPGM('cells.pgm')
writePGM(width, height, data, 'mein_bild.pgm')
```

Dabei geben `width` und `height` die Breite und Höhe des Bildes an, und `data` ist ein Pythonarray (Typ `list`) mit den Intensitätswerten der Pixel (von 0 = schwarz bis 255 = weiß). Auf das Pixel mit der Koordinate (x,y) greift man genauso zu wie auf die Matrix in Übung 4:

```
value = data[x + y*width]            # lesender Zugriff
data[x + y*width] = value            # schreibender Zugriff
index = x + y*width                  # Umrechnung zwischen Koordinaten ...
x, y = index % width, index / width  # ... und Pixelindex
```

Die letzten beiden Zeilen zeigen, wie man zwischen den Koordinaten eines Pixels und seinem Index im Array `data` umrechnet. Alle Funktionen sollen im File `image.py` abgegeben werden.

a) Es ist bekannt, dass alle dunklen Pixel mit einem Wert unter 60 zum Hintergrund gehören, während alle anderen zum Vordergrund (also zu Zellkernen) gehören. Schreiben Sie eine Funktion `mask = createMask(width, height, data, threshold)`, die ein Array erzeugt, wo alle Hintergrundpixel den Wert 0 und alle Vordergrundpixel den Wert 255 haben. Dabei bezeichnet `threshold` die gewählte Schwelle zwischen Vorder- und Hintergrund (hier: 60). Exportieren Sie das resultierende Maskenbild mittels `writePNG(width, height, mask, 'mask.pgm')` in ein PGM-File und geben Sie dieses File ebenfalls ab. *2 Punkte*

b) Schreiben Sie eine Funktion `graph = createGraph(width, height, mask)`, die aus dem Maskenbild einen Graphen in Adjazenzlisten-Darstellung nach folgenden Regeln erzeugt: *6 Punkte*

- Jeder Pixel stellt einen Knoten des Graphen dar, wobei die Knotennummer dem Pixelindex entspricht.
- Die Kanten des Graphen verbinden Pixel (=Knoten) mit ihren nächsten Nachbarn, d.h. Pixel (x,y) kann mit den Pixeln (x-1,y), (x+1,y), (x,y-1), (x,y+1) verbunden werden. Dies bezeichnet man als "Vierer-Nachbarschaft". Beachten Sie, dass bei Randpixeln nicht alle Nachbarn existieren (z.B. hat Pixel (0,0) keinen Nachbarn (-1,0), denn der wäre außerhalb des Bildes).
- Eine mögliche Kante wird nur dann tatsächlich eingefügt, wenn ihre Endknoten entweder beide zum Vordergrund oder beide zum Hintergrund gehören.

c) Bestimmen Sie mit einer Funktion `anchors, labeling = connectedComponents(graph)` die Zusammenhangskomponenten des Graphen aus b). Wenn Sie diese Funktion mittels Tiefensuche implementieren, müssen Sie die Stack-basierte Version aus Übung 9 verwenden, um den Fehler "maximum recursion depth exceeded" zu vermeiden. Alternativ können Sie den Union-Find-Algorithmus aus der Vorlesung verwenden. Neben der Ankerliste soll die Funktion ein Array `labeling` zurückgeben, das jedem Knoten die laufende Nummer der Zusammenhangskomponente zuordnet, zu der der Knoten gehört. Dabei soll die Hintergrundregion das Label 0 erhalten, der erste Zellkern das Label 1 usw. Exportieren Sie dieses Array als Bild in ein File `labeling.pgm` und geben Sie dieses File ebenfalls ab. Wie viele Zellkerne hat Ihr Algorithmus gefunden? *4 Punkte*

d) Schreiben Sie Funktionen `size = getSize(labeling)` und `intensity = getMaxIntensity(data, labeling)`, die für jede Region die Größe (=Anzahl der Pixel) und die maximale Intensität berechnen und in einem Array zurückgeben, so dass z.B. `size[regionLabel]` die Größe der Region mit der Nummer `regionLabel` angibt. Wie viel Prozent des Bildes sind Hintergrund? Wie viele Regionen haben weniger als 30 Pixel (und sind dann wahrscheinlich keine Zellkerne, sondern Detektionsfehler)? Wie viele weiße Zellkerne gibt es (maximale Intensität > 220)? Schreiben Sie eine Funktion `output = createOutput(labeling, size, intensity)`, die ein Bild mit folgenden Eigenschaften erzeugt: Alle Hintergrundpixel haben den Wert 0, alle Pixel in kleinen Regionen (Detektionsfehler) haben den Wert 255, alle Pixel in weißen Zellkernen haben den Wert 160, und alle Pixel in grauen Zellkernen haben den Wert 80. Exportieren Sie das Bild in ein File `output.pgm` und geben Sie das File ebenfalls ab. *6 Punkte*

Aufgabe 2 – Kürzeste Wege mit Dijkstra's Algorithmus 20 Punkte

Für diese Aufgabe benötigen Sie wieder das File "`entfernungen.json`" (siehe http://hci.iwr.uni-heidelberg.de/Staff/ukoethe/download/entfernungen.json) mit 154 deutschen Städten, ihren Koordinaten, sowie den Straßenentfernungen zu umliegenden Städten:

```
{
    "Name der Stadt": {
        "Koordinaten":  {
            "Breite":  "50N47",
            "Länge":   "06E05"
        },
        "Nachbarn":  {
            "Nachbarstadt_1": 16.8,
            "Nachbarstadt_2": 32.4,
            ...
        }
    },
    ...
}
```

Die Koordinaten sind als Geokoordinaten mit Breiten- und Längengrad (hier: 50° 47 Minuten Nord, 6° 5 Minuten Ost), die Nachbarstädte mit Entfernungsangabe in Kilometern angegeben.

a) Implementieren Sie eine Funktion *5 Punkte*

```
graph, names, weights = createGraph(distanceDict)
```

die aus den Entfernungsdaten einen gewichteten Graphen generiert. Dabei ist `distanceDict` das Dictionary, das Sie aus dem File eingelesen haben. `graph` ist die Adjazenzliste des Graphen, `names` ist eine *property map*, die jeder Knotennummer den jeweiligen Städtenamen zuordnet, und `weights` eine *property map*, die die Kantengewichte speichert, so dass `weights[(i,j)]` das Gewicht der Kante von Knoten `i` nach Knoten `j` angibt. Geben Sie Ihre Lösung im File "`shortest_path.py`" ab.

b) Verwenden Sie Dijkstras Algorithmus, um die kürzesten Wege und die Straßenentfernung zwischen *5 Punkte*

I. Aachen und Passau
II. Saarbrücken und Leipzig
III. München und Greifswald
IV. Konstanz und Kassel

zu bestimmen. Implementieren Sie zu diesem Zweck in "`shortest_path.py`" eine Funktion

```
path, distance = dijkstra(graph, weights, start, destination)
```

die den Weg als Folge von Kanten (jeweils mit Anfangs-, Endknoten sowie Entfernung) und die Gesamtentfernung zurückgibt. Verwenden Sie dann die *property map* `names` aus a), um den Pfad in die Darstellung "Startstadt => x km => Stadt2 => y km => Stadt3 => … => Zielstadt (insgesamt: z km)" umzuwandeln.

c) Implementieren Sie auch den A*-Algorithmus, wobei als Schätzung für den Restweg zum Ziel die Luftlinienentfernung zwischen den betreffenden Städten benutzt werden soll. Eine ausführliche Erklärung, wie man die Luftlinienentfernung aus den Geokoordinaten berechnet, finden Sie auf der Webseite http://www.koordinaten.de/informationen/formel.shtml. Schreiben Sie einen Test, der prüft, dass die Luftlinienentfernungen stets kleiner sind als die im File angegebenen Straßenentfernungen. Implementieren Sie nun eine Funktion *10 Punkte*

```
path, distance = a_star(graph, weights, start, destination)
```

und testen Sie für die obigen Städtepaare, dass der A*-Algorithmus dieselben kürzesten Wege findet wie der Dijkstra-Algorithmus. Vergleichen Sie, wie viele Knoten bei beiden Algorithmen jeweils besucht werden – führt der A*-Algorithmus tatsächlich zu einer Beschleunigung? Geben Sie Ihre Lösung ebenfalls im File "`shortest_path.py`" ab.