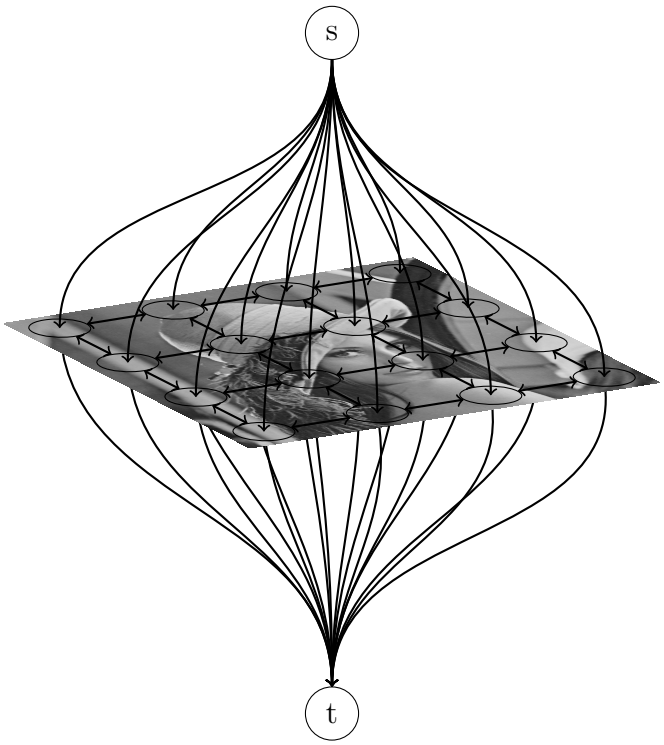
Total variation based image restoration using graph cuts

Bjørn Rustad

November 3, 2014



Supervisor:
Markus Grasmair, Department of Mathematical Sciences, NTNU

Contents

1 Introduction

In everyday life cameras are used to capture a moment and save it for eternity, but imaging technology technology can be used in many other contexts, including medical and astronomical applications. With the advent of computers, tasks previously reserved for the human brain, like recognizing textures, detecting edges and inferring shapes and motion, can now be performed algorithmically. The background of these methods span several fields, including psychology and biology for the study of human vision, statistics and analysis for the mathematical foundations, and computer science for the implementation and performance analysis.

It is possible to roughly spread the tasks of image processing out along a line spanning from the raw capturing of light coming into the camera, a purely physical problem, to the computer vision methods in the other end, semantically interpreting the scene, recognizing objects, their position and their movement. In between we have algorithms working on the captured image, without implying too much about what the image contains. These methods are often categorized as *early vision* and includes but is not limited to image restoration, segmentation, filtering and edge detection.

In the process of capturing the image with our physical apparatus, there is always some noise included. Some might come from the physical nature of how light travels from the scene to the objective, while some might result from inaccuracies in the construction of the capturing apparatus. These noise-inducing processes can be studied, and modeled mathematically, and the process of image restoration looks at how one can remove some – or optimally all – of the noise in the captured image. These methods often take into account how we would expect a "normal" image to look in the capturing conditions, and also what kinds of noise we expect to be a part of the image.

In the next section we will see a small overview of some of the most popular methods in image restoration, but in the rest of the report we will focus on a total variation based method. One of the strong points of these methods are their ability to preserve edges, instead of smoothing over them. Different approaches to total variation image restoration exist, but we will formulate it as an energy minimization problem.

As our input images are digital images, the first step will be to discretize the energy function. Next comes an important part of this project which is to reformulate the minimization problem as a series of minimum cut problems from graph theory. This means we have to carefully construct graphs and verify that finding the minimum cuts actually yield a global minimizer of the original energy function.

A few different algorithms for finding these minumum cuts are considered, but we will mainly focus on the push-relabel algorithm and exploit its ability to re-use some of the results across the separate subproblems.

Towards the end we will look at some image restoration results showing how the method performs for different kinds of noise, and for different input parameters.

2 Methods in image restoration

There are numerous methods for restoring an image, and here we will briefly look at some of the more popular approaches. Given an original image v, the goal is to obtain a somehow denoised output image u. Which method to choose depends on what properties one wants the output image to have, and also on how the original image was obtained.

2.1 Gaussian filtering

Gaussian filtering, or Gaussian blur is a straight forward kind of method where the image is convolved with the Gaussian function

$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2}\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right). \tag{1}$$

In the discrete case of an image consisting of separate pixels, Gaussian blur amounts to setting each pixel value as the weighted average of its neighbors in the original image. This will, in addition to smoothing out possible noise, add blur and remove details from the image.

The Gaussian function happens to be the fundamental solution of the heat equation $\partial_t u = \Delta u$. Convolving it with the original image v is therefore equivalent to solving the heat equation with v as initial value, for some time $t > 0$ depending on σ. Care must be taken on the boundary, and one possibility is to symmetrically extend the image in all directions.

By basic Fourier analysis it is possible to show that the Gaussian filter is a low-pass filter which attenuates high frequencies. See for example [1] for more information.

2.2 Anisotropic diffusion

Since the Gaussian filter will blur out both noise and details of the image, it would be nice to somehow reduce the amount of blurring done around edges, or what we believe to be edges. This can be done by controlling the thermal diffusivity $\alpha(u)$ of the image in the heat equation

$$\begin{cases} \partial_t u = \operatorname{div}(\alpha(u)\nabla u) \\ u|_{t=0} = v. \end{cases} \tag{2}$$

The problem is then, how to detect the edges such that we can reduce $\alpha(u)$ in those parts of the image.

Perona and Malik proposed a modified heat equation

$$\partial_t u = \operatorname{div}\left(\frac{\nabla u}{1+\frac{|\nabla u|^2}{\lambda^2}}\right) \tag{3}$$

which is actually related to the way brightness is percepted by the human visual system. At edges, $|\nabla u|$ is big, and the thermal diffusivity will be small, while at already smooth

portions of the image, the amount of applied smoothing will also be higher. The model has some theoretical problems related to well-posedness, for more information see [1].

A different kind of anisotropic diffusion model is the total variation flow model which can be formulated as

$$\partial_t u = \operatorname{div} \frac{\nabla u}{|\nabla u|}. \tag{4}$$

As the name suggests this model can be related to the total variation formulation considered in this project. One discrete time-step in the solution of this PDE corresponds to the Euler-Lagrange equation of the total variation minimization problem presented later.

Another more advanced approach is to let the thermal diffusivity be a tensor $A(u)$ in the equation

$$\partial_t u = \operatorname{div}(A(u) \cdot \nabla u). \tag{5}$$

This way, it is possible to have a small diffusivity *across* edges, and at the same time a large diffusivity *along* the edges. See for example [1] for more information on these diffusion tensor methods.

2.3 Other filters

The non-local means method of image restoration takes into account the fact that two spatially separated parts of an image might be very similar, for example in the case of a regular texture. As in many other methods, one defines some neighborhood for each pixel, but when restoring a pixel, one considers all other pixels whose neighborhoods are similar to the neighborhood of the current pixel, for some notion of similarity. The new pixel value is then obtained by taking the average of all these similar pixels.

A different filter much used in real-world image processing is the median filter. It replaces the value of each pixel with the median of the values of the neighboring pixels. It can perform very well in practice, especially with salt and pepper noise or images where some pixels are missing.

3 Total variation

The image restoration method considered in this project tries to find an image u which is close to the captured image v, while it at the same time has a low *total variation* by solving

$$\min_u \int_\Omega |u - v|^p + \beta \int_\Omega |\nabla u|. \tag{6}$$

Minimizing the total variation $\int_\Omega |\nabla u|$ will smooth out differences in the image while the term $\int_\Omega |u - v|^p$ will constrain the solution image u to be close to the original image v. Combining the two will give us an image close to the original, but more regular. The parameter β controls how strongly we want to regularize the image.

How we compute and integrate the gradient the image will be clarified later, but first we will look at how this model can be grounded in spatial statistics; a field which forms an important part of image analysis.

3.1 Probabilistic background

Image processing is pushing forward in many fields at once, analysis, spatial statistics and discrete optimization to name a few. Methods can often be related across fields, and in this short section we will see how the total variation method can be related to the maximum a posteriori estimation (MAP) approach in spatial statistics.

There are many ways to model noise in an image, and different models take different physical phenomena into account. We will in this brief introduction only consider the simplest model, namely *additive noise*, which given an image u results in a captured image

$$v = u + \epsilon, \tag{7}$$

where ϵ represents the noise that can be modeled by different probability distributions.

Given a captured noisy image v we want to recover u, so we introduce the likelihood

$$p(u \mid v) = \frac{p(v \mid u) \cdot p(u)}{p(v)} \tag{8}$$

using Bayes' theorem. Maximizing this will give us the image u which is most probable given our noisy image v. The *prior* $p(u)$ represents how probable we think it is to obtain an image like u in the set of all obtainable images, and $p(v \mid u)$ is our noise model, how we think a noisy image v is distributed given an image u.

Since $p(v)$ is constant, and the logarithm is a monotonically increasing function, the maximization problem is equivalent to

$$\min_u \big(-\ln p(v \mid u) - \ln p(u) \big). \tag{9}$$

For discrete images consisting of pixels, we use $x, y \in \mathbb{R}^2$ to denote pixel positions such that u_x and u_y are the values of those pixels, *not to be confused with* partial derivatives $\partial_x u$ and $\partial_y u$ which will not be used. If the noise term $p(v \mid u)$ is modeled to be independently and identically distributed Gaussian noise in each pixel, we have

$$p(v \mid u) \propto \prod_x \exp\left(-\frac{(u_x - v_x)^2}{2\sigma^2} \right). \tag{10}$$

The prior probability $p(u)$ depends on what kinds of images are considered, and how they are obtained. In the theory of Markov random fields, one lets the probability of a pixel value depend only on some neighborhood $\mathcal{N}$ of the pixel. A common choice for the relation between neighboring pixels is the Laplace distribution

$$p(u) \propto \prod_x \prod_{y \in \mathcal{N}(x)} \exp\left(-\frac{|u_x - u_y|}{b} \right). \tag{11}$$

Consult for example [2] for further reading on noise models and different possibilities for the prior probability.

Inserting the prior (11) and the noise model (10) into (9) we obtain

$$\min_u \left(\sum_x |u_x - v_x|^2 + \beta \sum_x \sum_{y \in \mathcal{N}(x)} |u_x - u_y| \right), \tag{12}$$

where β is a combination of the coefficients in the two terms. If the noise is modeled by a Laplace distribution

$$p(u \mid v) \propto \prod_x \exp\left(-\frac{|u_x - v_x|}{b} \right) \tag{13}$$

we obtain the L^1-norm instead of the L^2-norm,

$$\min_u \left(\sum_x |u_x - v_x| + \beta \sum_x \sum_{y \in \mathcal{N}(x)} |u_x - u_y| \right). \tag{14}$$

We see that the two discrete problems (12) and (14) resemble the original formulation in (6). Before presenting the discretization of (6), we will look at a more rigorous presentation of (6), and the definitions, spaces and theorems needed to treat it.

3.2 Continuous formulation

The method of total variation was initially introduced to image restoration by L. Rudin, S. Osher, and E. Fatemi in [3], but is also used in numerous other applications. One of its strong theoretical points is its ability to recover edges, compared to other restoration algorithms which might smooth over edges in the original image. The method can be formulated as minimizing the energy function

$$E_v(u) = \int_\Omega |u - v|^p + \beta \, TV(u) \tag{15}$$

over $L^p(\Omega)$. Here v is the original image, and Ω is the image domain, in our case an open subset of $\mathbb{R}^2$. Normally, p is chosen to be 1 or 2, the two choices that correspond to the two different noise models in (14) and (12).

In the following, assume that our image domain Ω is an open and bounded subset of $\mathbb{R}^2$ with a Lipschitz boundary. Since most images are given on rectangles in $\mathbb{R}^2$, this is not a problem.

The term $TV(u)$ in (15) is the total variation of the image, introduced earlier as $\int_\Omega |\nabla u|$, and often called the regularization term. Since it can be problematic to calculate the traditional gradient, the following definition is introduced, using Green's identity to transfer the derivative to a smooth test function.

Definition 1 (Total variation)**.** *Given a function $u \in L^1(\Omega)$, the total variation of u, often written $\int_\Omega |Du|$, where the D is the gradient taken in the distributional sense, is*

$$TV(u) = \int_\Omega |Du| = \sup \left\{ \int_\Omega u \cdot \operatorname{div} \varphi : \varphi \in C_c^\infty(\Omega, \mathbb{R}^2), \|\varphi\|_{L^\infty(\Omega)} \leq 1 \right\}. \tag{16}$$

The test functions φ are taken from $C_c^\infty(\Omega, \mathbb{R}^2)$, the space of smooth functions from Ω to $\mathbb{R}^2$ with compact support in Ω.

Note that since Ω is open and bounded, the test functions φ vanish on the boundary of Ω. Next we introduce the space of functions with bounded variation.

Definition 2 (Functions of bounded variation)**.** *The space of functions of bounded variation $BV(\Omega)$ is the space of functions $u \in L^1(\Omega)$ for which the total variation is finite, i.e.,*

$$BV(\Omega) = \left\{ u \in L^1(\Omega) : TV(u) < \infty \right\}. \tag{17}$$

Before introducing the coarea formula, we need the formal definition of a set perimeter. Here, $\mathbb{1}_E$ signifies the characteristic function of the set E, the function which is equal to one in every point in E, and zero elsewhere.

Definition 3 (Set perimeter)**.** *The perimeter of a set E is defined as*

$$P(E, \Omega) = \int_\Omega |D\mathbb{1}_E|. \tag{18}$$

A measurable set $E \subset \Omega$ is of finite perimeter in Ω if $\mathbb{1}_E \in BV(\Omega)$.

The set perimeter behaves mostly as one might expect for "nice" sets, just note that if the boundary of E overlaps with the boundary of Ω, then $P(E, \Omega)$ will not include the overlapping parts of the boundary.

For an image u and a level λ we denote the *level set* by $\{u \leq \lambda\}$ defined as the set $\{x \in \Omega : u_x \leq \lambda\}$. The thresholded images are defined as

$$u^\lambda = \mathbb{1}_{u \leq \lambda}. \tag{19}$$

With all these definitions in place, we are ready to introduce the coarea formula, which will let us write the total variation as an integral over the range of the image levels λ.

Theorem 1 (The coarea formula)**.** *Let $u \in BV(\Omega)$. Then for almost every λ the set $\{u \leq \lambda\}$ is of finite perimeter, and one has the coarea formula*

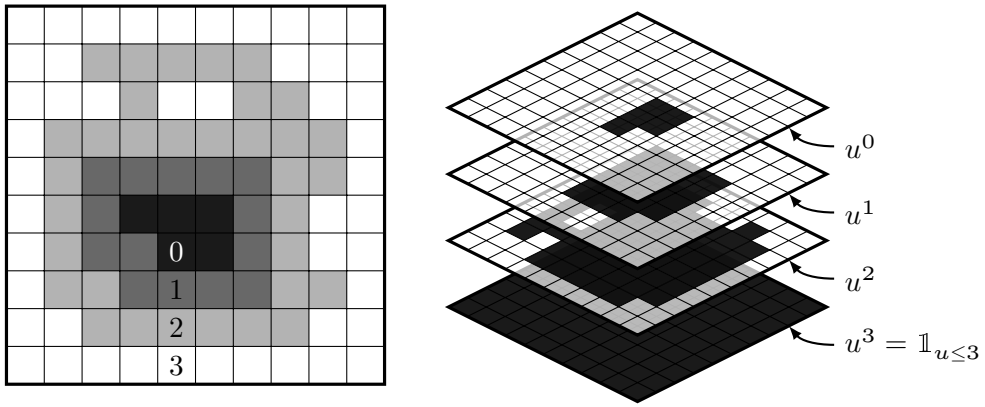
$$TV(u) = \int_\Omega |Du| = \int_{\mathbb{R}} P(\{u \leq \lambda\}, \Omega)\, d\lambda \tag{20}$$

Note that the equality in (20) holds for all $u \in L^1(\Omega)$, not only when the total variation is finite. We can now rewrite (15) as

$$E_v(u) = \int_\Omega |u - v|^p + \beta \int_{\mathbb{R}} P(\{u \leq \lambda\}, \Omega)\, d\lambda. \tag{21}$$

This will be the starting point for our discretization, and it will allow us to decompose the energy into a sum over the discrete set of level values.

More analysis of the continuous problem is given in [4], together with different numerical methods for solving it. Results on how the set of jumps in the resulting image u is contained in the set of jumps in the original image v are also presented there.



(a) An example gray-scale image with four different levels.

(b) The thresholded images, which are elements of $\{0,1\}^{|\Omega|}$.

Figure 1: A sample gray-scale image, and its decomposition into thresholded images. Note that while the image has values from the set $\{0,\dots,3\}$, the thresholded images are binary.

3.3 Discrete formulation

Since digital images are given on a discrete grid with values taken from a discrete and finite set of levels, we want to discretize (21), and solve the resulting discrete problem. We denote $\mathcal{G}$ for the grid of pixels, and assume that the value of each pixel is taken from the set $\mathcal{L} = \{0,\dots,L-1\}$. This is a reasonable assumption for grayscale images.

3.3.1 Regularization term

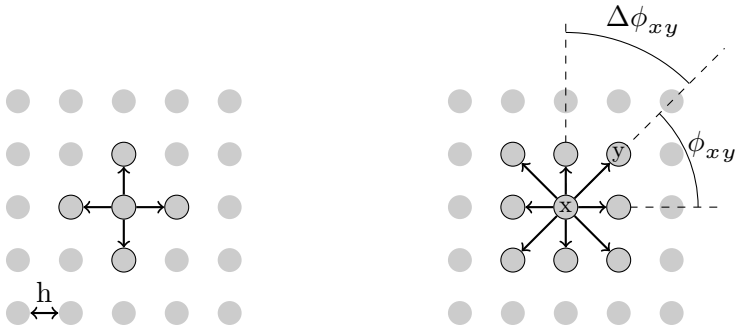
Chambolle discretizes the total variation as

$$TV(u) = \sum_{i,j} \sqrt{\left(u_{i+1,j} - u_{i,j}\right)^2 + \left(u_{i,j+1} - u_{i,j}\right)^2} \tag{22}$$

in his much cited paper [5], where he introduces his dual approach to the continuous minimization problem. This straight-forward way of discretizing the gradient offers, according to Chambolle, a good compromize between isotropy and stability. However, it does not decompose into a sum over the levels of the image, which we need for our graph construction.

Following the notation used in [6], we let u_x denote the value of the image u at position $x \in \mathcal{G}$. With a finite set of pixel values $\mathcal{L}$, we also have a finite number of thresholded images u^λ. Figure 1 visualizes a possible image and its decomposition into thresholded images, one for each level value.

We also need to introduce the notion of pixel neighborhoods. A pixel in the image has a neighborhood which consists of the pixels around it, but not itself. We denote the neighborhood of pixel x by $\mathcal{N}(x)$ and we will call the line going from pixel x to pixel



(a) Size four neighborhood. (b) Size eight neighborhood.

Figure 2: Two common neighborhood stencils.

$y \in \mathcal{N}(x)$ the *edge* between x and y. Two common neighborhood stencils are shown in Figure 2.

Recall that we managed to write the total variation as an integral of the perimeter of the level sets in (20). We estimate the perimeter of the level set u^λ in the discrete setting, such that the total variation can be written as a sum over the level values

$$TV(u) = \sum_{\lambda=0}^{L-2} P(\{u \leq \lambda\}, \Omega) \approx \sum_{\lambda=0}^{L-2} \sum_{(x,y)} w_{xy} |u_x^\lambda - u_y^\lambda|. \tag{23}$$

The sum over (x, y) signifies a sum over all pixels cliques of size two, i.e. all pairs (x, y) such that x and y are in a neighborhood relation. The w_{xy} is a weight parameter, and the sum over λ ends at $L-2$ since u^{L-1} is equal to 1 in every pixel of the image.

Intuitively, the perimeter of the level set $\{u \leq \lambda\}$ is proportional to the number of pixels at the boundary of the set. This is again proportional to the number of neighborhood relations crossing the boundary, and $|u_x^\lambda - u_y^\lambda|$ only contributes to the sum when one pixel is in $\{u \leq \lambda\}$ and the other is not.

Boykov and Kolmogorov argue in [7] that if the weight is chosen as

$$w_{xy} = \frac{h^2 \cdot \Delta\phi_{xy}}{2 \cdot |d_{xy}|}, \tag{24}$$

the discrete perimeter in (23) converges to the continuous perimeter in (20). Here, h is the grid size and d_{xy} is the euclidean distance of the edge from x to y. The number $\Delta\phi_{xy}$ is the difference between the angle of this edge and the next edge, if the edges are sorted by increasing angles. These parameters are also shown in Figure 2. Boykov and Kolmogorov prove we have convergence when all of h, $\sup_{xy} \Delta\phi_{xy}$, and $\sup_{xy} d_{xy}$ go to zero.

3.3.2 Fidelity term

The first integral of (15) is often called the fidelity term as it controls the similarity of the output and input image. As for the regularization term, we want to discretize it and

write it as a sum over the level values λ.

We define the following function for some pixel value k and some pixel position x in the original image v, which is the value of the energy if we were to color pixel x with label k

$$N_x(k) = |k - v_x|^p. \tag{25}$$

This allows us to discretize $\int_\Omega |u - v|^p$ of (21) in the following way

$$\sum_x |u_x - v_x|^p = \sum_x N_x(u_x). \tag{26}$$

We want to write the energy as a sum over the different levels of the image, so we use the following decomposition formula which holds for any function $N(k)$ taking values $k \in \mathcal{L}$:

$$\begin{aligned} N(k) &= \sum_{\lambda=0}^{k-1} \big(N(\lambda+1) - N(\lambda)\big) + N(0) \\ &= \sum_{\lambda=0}^{L-2} \big(N(\lambda+1) - N(\lambda)\big) I(\lambda < k) + N(0). \end{aligned} \tag{27}$$

Here $I(x)$ is an indicator function that takes the value 1 if x is true, and 0 if x is false. Since $I(\lambda < u_x) = (1 - u_x^\lambda)$ we rewrite (26) and obtain

$$\sum_x N_x(u_x) = \sum_{\lambda=0}^{L-2} \sum_x \big(N_x(\lambda+1) - N_x(\lambda)\big)(1 - u_x^\lambda) + N_x(0). \tag{28}$$

3.3.3 Total energy

We have now discretized the energy function and decomposed it into a sum over all the levels λ. Ignoring the constant term $N_x(0)$ we are left with

$$E_v(u) = \sum_{\lambda=0}^{L-2} \sum_x E_\lambda^x(u_x^\lambda) + \beta \sum_{\lambda=0}^{L-2} \sum_{(x,y)} E^{x,y}(u_x^\lambda, u_y^\lambda) =: \sum_{\lambda=0}^{L-2} F_\lambda(u^\lambda) \tag{29}$$

where

$$E_\lambda^x(u_x^\lambda) = \big(N_x(\lambda+1) - N_x(\lambda)\big)(1 - u_x^\lambda) \tag{30}$$

$$E^{x,y}(u_x^\lambda, u_y^\lambda) = w_{xy} |u_x^\lambda - u_y^\lambda|. \tag{31}$$

If we minimize each level separately it is obvious that we also minimize the total energy. In other words if we for every λ find a thresholded image u^λ that minimizes $F_\lambda(u^\lambda)$, the sum in (29) will be minimized. But the question remains if these obtained u^λ can be combined to produce an output image. They were defined as $u^\lambda = \mathbb{1}_{u \le \lambda}$, so we need them to be monotonically increasing in increasing level values, i.e.

$$u_x^\lambda \le u_x^\mu \quad \forall \lambda \le \mu, \quad \forall x \in S. \tag{32}$$

In the following sections we will present a graph cut algorithm that finds thresholded images minimizing each level, *while guaranteeing that they meet this requirement.*

4 Graph cut formulation

We have successfully discretized our original continuous total variation optimization problem, and decomposed it into one smaller problem for each level of the image. These smaller problems are binary in the sense that for each pixel, we want to know if its value should be above the current level λ or not. The only thing tying these problems together is the fact that we should be able to stack the thresholded images in the end.

We will in this section present a graph cut approach to the level subproblem. A graph is constructed containing one vertex per pixel, with carefully chosen edge weights such that a minimum cut of the graph will also minimize the energy function of (29).

4.1 Networks

Using the notation of [8] we will denote a directed graph as $G = (V, E)$ where V is a finite set of vertices, and E is a binary relation on V. If $(u, v) \in E$ we say that there is an edge from u to v in the graph G.

We introduce the non-negative capacity function $c : V \times V \to [0, \infty)$. Only edges $(u, v) \in E$ can have a positive capacity $c(u, v) = q > 0$ and it means that it is possible to send a *flow* of maximum q units from u to v. For convenience we will let $c(u, v) = 0$ for any pair $(u, v) \notin E$, and we do not allow self-loops in our graph. When a directed graph G is equipped with capacity function c, we call it a network, and write $G = (V, E, c)$.

There are two special vertices in the network, the source s and the sink t. Contrary to other vertices, which can neither produce nor receive excess flow, the source can produce and the sink can receive an unlimited amount of flow. The most basic problem in flow network theory is the question of how much flow it is possible to send through the network from the source to the sink.

What we seek in our final network is a minimum s-t-cut, a "minimal" line through the network that cuts a set of edges and divides the vertex set in two, separating the source from the sink.

Definition 4 (s-t-cut)**.** *Given a network $G = (V, E, c)$, an s-t-cut (S, T) of G is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$. The capacity of the cut is*

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v), \tag{33}$$

and a minimum s-t-cut is a cut whose capacity is minimum over all s-t-cuts.

Note that there might exist several minimum s-t-cuts in a network, resulting in different partitions of V. This is why we need to verify later that the cuts we obtain result in stackable thresholded images.

4.2 Network representable energy functions

The next step is to find a way to construct a network such that we can minimize the energy in (29) by finding a minimum s-t-cut. We will do this by creating small and

simple networks representing the separate summands of the energy. For these small networks it will be easy to verify that the minimal cut also minimizes the corresponding part of the energy function, and they can then be merged giving a network representing the complete energy function in (29).

First we need to establish the definition of a network representable function, presented by Kolmogorov and Zabih in [9].

Definition 5 (Network representable functions)**.** *A function $\mathcal{E}(x_1, \dots, x_n)$ of n binary variables is network-representable if there exists a network $G = (V, E, c)$ with terminals s and t, and a subset of vertices $V_0 = \{v_1, \dots, v_n\} \subseteq V - \{s, t\}$ such that, for any configuration $(x_1, \dots, x_n) \in \{0, 1\}^n$, the value of the energy $\mathcal{E}(x_1, \dots, x_n)$ is equal to a constant plus the cost of the minimum s-t-cut among all cuts $C = (S, T)$ where $x_i = 0 \Leftrightarrow v_i \in S$ and $x_i = 1 \Leftrightarrow v_i \in T$, $\forall\, 1 \leq i \leq n$.*

From this definition we see that if we have a network-representable function $\mathcal{E}$ it is possible to find an exact global minimum of $\mathcal{E}$ by finding a minimal s-t-cut in a network representing $\mathcal{E}$.

Furthermore Kolmogorov and Zabih present an important result concerning what kinds of functions are network-representable.

Theorem 2 (Identification of network representable functions)**.** *Given an energy function $\mathcal{E}$ of n binary variables of the form*

$$\mathcal{E}(x_1, \dots, x_n) = \sum_i \mathcal{E}^i(x_i) + \sum_{i<j} \mathcal{E}^{i,j}(x_i, x_j), \tag{34}$$

then $\mathcal{E}$ is graph representable if and only if each term $\mathcal{E}^{i,j}$ satisfies the inequality

$$\mathcal{E}^{i,j}(0,0) + \mathcal{E}^{i,j}(1,1) \leq \mathcal{E}^{i,j}(0,1) + \mathcal{E}^{i,j}(1,0). \tag{35}$$

This theorem will allow us to verify that our energy function actually is network representable.

Finally, the following theorem, proved by Kolmogorov and Zabih in [9], will be crucial in our network construction.

Theorem 3 (Additivity)**.** *The sum of a finite number of network-representable functions*

$$\mathcal{E}(x_1, \dots, x_n) = \sum_k \mathcal{E}^k(x_1, \dots, x_n), \tag{36}$$

each represented by a network $G^k = (V^k, E^k, c^k)$, is network-representable by $G = (V, E, c)$ where $V = \cup_k V^k$, $E = \cup_k E^k$ and $c(u, v) = \sum_k c^k(u, v)$.

It allows us to construct small networks representing the different summands of our energy function (29), before adding them together to create a final network representing the total energy.

Note that when we apply this theorem later, we will assume that all the summands of (29) have the whole picture as their domain. It is unproblematic to extend $E^x_\lambda(u^\lambda_x)$ and $E^{x,y}(u^\lambda_x, u^\lambda_y)$ such that they take all the pixels their argument and then ignore all pixels except the ones they actually depend on.

(a) The network when $E_\lambda^x(0) > 0$, with constant equal to 0.

(b) Network when $E_\lambda^x(0) < 0$, with constant equal to $-E_\lambda^x(0)$.

Figure 3: The network construction for the fidelity term $E_\lambda^x(u_x^\lambda)$. See Table 1 for an overview of the different possible cuts, and on why this construction works.

4.3 Network construction

We will construct a network in such a way that if a variable u_x^λ ends up on the source side of the cut we set $u_x^\lambda = 0$, and if it ends up on the sink side we set $u_x^\lambda = 1$, as in Definition 5. This is an arbitrary choice, but still something we have to keep in mind through the rest of the section.

We will now consider the two kinds of summands in the energy function in (29). The fidelity term coming from our aim to keep the output image close to the original image, and the regularization term coming from our aim to minimize the total variation.

4.3.1 Fidelity term

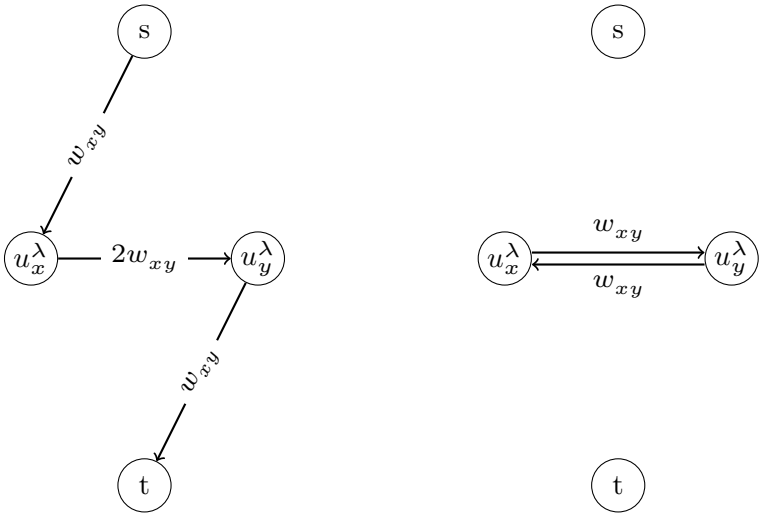
The fidelity term of our energy function in (29) simplifies to

$$E_\lambda^x(0) = N_x(\lambda + 1) - N_x(\lambda) \tag{37}$$

$$E_\lambda^x(1) = 0 \tag{38}$$

Table 1: Each row represents one of the two possible values of $u_x^\lambda \in \{0, 1\}$. The energy $E_\lambda^x(u_x^\lambda)$ and minimum cut obtaining this configuration is shown. The last two columns show the capacities of the cut for each of the two network constructions in Figure 3. We verify that for each of the two network constructions, the cut capacities are equal to the energies, plus a constant.

u_x^λ	$E_\lambda^x(u_x^\lambda)$	Min. cut (S, T)	Network (a) cut cap.	Network (b) cut cap.
0	$E_\lambda^x(0)$	$(\{s, u_x^\lambda\}, \{t\})$	$E_\lambda^x(0)$	0
1	0	$(\{s\}, \{u_x^\lambda, t\})$	0	$-E_\lambda^x(0)$



(a) Representing $E^{x,y}(u_x^\lambda, u_y^\lambda)$ with a constant term of w_{xy}.

(b) Representing $E^{x,y}(u_x^\lambda, u_y^\lambda)$ with a constant term of 0.

Figure 4: Two alternative ways of constructing a network representing the energy term $E^{x,y}(u_x^\lambda, u_y^\lambda)$. See Table 2 for an overview of the different possible configurations of $(u_x^\lambda, u_y^\lambda)$ and how they correspond to cuts through the network.

where $E_\lambda^x(0)$ might be positive or negative depending on λ and the pixel value v_x.

Figure 3 shows how networks can be constructed to represent this part of the total energy. The construction differs depending on whether $E_\lambda^x(0)$ is positive or negative. Table 1 shows how the cuts correspond to the values of u_x^λ and we can easily verify that the constructed network actually represents the fidelity term in the energy function.

4.3.2 Regularization term

For our neighboring relation in (29) of the form

$$E^{x,y}(u_x^\lambda, u_y^\lambda) = w_{xy}|u_x^\lambda - u_y^\lambda| \tag{39}$$

we have

$$\begin{aligned} E^{x,y}(0,0) &= 0, \\ E^{x,y}(0,1) &= w_{xy}, \\ E^{x,y}(1,0) &= w_{xy}, \\ E^{x,y}(1,1) &= 0, \end{aligned} \tag{40}$$

and by Theorem 2 our energy function is graph representable. In [9], Kolmogorov and Zabih presents a way to construct a graph for any graph representable function on the form shown in Theorem 2. Since the energies in (40) are especially simple, the construction and presentation is simplified.

Table 2: An overview of the possible configurations of the variables in the term $E^{x,y}(u_x^\lambda, u_y^\lambda)$. For each configuration the corresponding energy and the cut yielding this configuration is shown. The last two columns show the capacities of the cut in the two alternative network constructions shown in Figure 4. We verify that for each of the two network constructions, the cut capacities are equal to the energies, plus a constant.

$(u_x^\lambda, u_y^\lambda)$	$E^{x,y}(u_x^\lambda, u_y^\lambda)$	Min. cut (S,T)	Alt. (a) cut cap.	Alt. (b) cut cap.
$(0,0)$	0	$(\{s, u_x^\lambda, u_y^\lambda\}, \{t\})$	w_{xy}	0
$(0,1)$	w_{xy}	$(\{s, u_x^\lambda\}, \{u_y^\lambda, t\})$	$2w_{xy}$	w_{xy}
$(1,0)$	w_{xy}	$(\{s, u_y^\lambda\}, \{u_x^\lambda, t\})$	$2w_{xy}$	w_{xy}
$(1,1)$	0	$(\{s\}, \{u_x^\lambda, u_y^\lambda, t\})$	w_{xy}	0

Figure 4 shows two different ways of how a network can be constructed to represent the regularization term. See Table 2 for an overview of how the two values of u_x^λ corresponds to cuts in the network.

Figure 5 shows a visualization of how the final network might look with all its edges. The source will have a lot of outgoing edges, one for each pixel, while the sink has one incoming edge for each pixel. The vertices corresponding to the pixels are only connected to the source, the sink, and their neighboring pixels, so their edge degree is much smaller than for s and t.

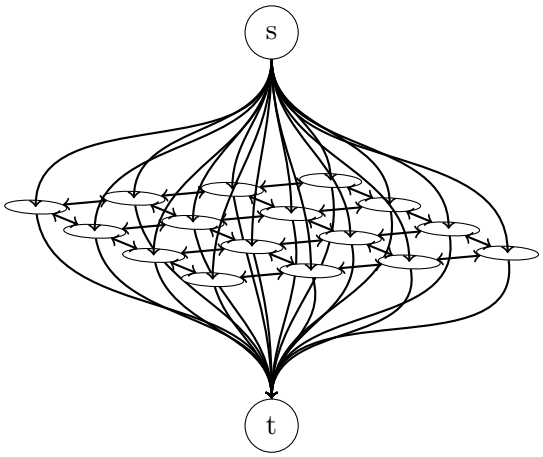


Figure 5: A visualization of the final network, with edges between neighboring pixels, and edges connecting the source and sink to the rest of the network. The edge capacities are not visualized, and of course some of them might be very high.

5 Maximum flow approach

In the previous section we have seen how finding the minimum cut of carefully constructed network can give us the thresholded image minimizing the energy function for one level value λ. We will in this and the next section see how such a minimum cut can be found by sending flow through the network and trying to identify the "bottleneck".

5.1 Flow networks

We have already introduced capacities, and briefly mentioned the notion of flow as something limited by the capacity. In other words, flow is something we can send through our network, but the capacity limits how much we can send along each edge. It is useful to imagine a water supply network with pipes of different sizes and a water source and sink.

Formally, we introduce the *flow* as the function $f : V \times V \to [0, \infty)$. This function keeps count of how much flow we are sending through each edge of our network and must satisfy the following two constraints

Capacity constraint: For all $u, v \in V, 0 \leq f(u, v) \leq c(u, v)$, i.e., for every pair of vertices, the flow is less than or equal to the capacity.

Flow conservation: For all $u \in V - \{s, t\}$

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v), \tag{41}$$

i.e., for any vertex except the source and the sink, the flow into the vertex must be equal to the flow out of the vertex.

Note that we have defined f with all pairs of vertices as its domain, even though it is only non-zero on edges $(u, v) \in E$. This makes it easier to write sums as in the flow conservation constraint. We say that an edge (u, v) is *saturated* if $f(u, v) = c(u, v)$.

We define $|f|$ as the net amount of flow from the source to the sink in the network. Because of the flow conservation constraint, this can be calculated as the net amount of flow going out of the source

$$|f| = \sum_{v \in V} f(s, v) - \sum_{u \in V} f(u, s). \tag{42}$$

Furthermore we denote the *net flow* across an s-t-cut $C = (S, T)$ as

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u). \tag{43}$$

Note how this definition differs from the capacity of a cut $c(S, T)$ in Definition 4. While the capacity of a cut represents how much flow it is maximally possible to send from S to T, the net flow across a cut represents the net amount of flow going across the cut, counting negatively the flow that goes back from T to S.

For any s-t-cut we have that

$$|f| = f(S,T). \tag{44}$$

This is quite intuitive given the flow conservation constraint, and a full chain of arguments can be found in [8].

Following from (43) and (44), we find that

$$\begin{aligned} |f| &= \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{v\in T}\sum_{u\in S} f(v,u) \\ &\leq \sum_{u\in S}\sum_{v\in T} f(u,v) \\ &\leq \sum_{u\in S}\sum_{v\in T} c(u,v) \\ &\leq c(S,T) \end{aligned} \tag{45}$$

for any s-t-cut $C = (S,T)$. A very central result in network flow theory called the max-flow min-cut theorem will be presented later. It states that the inequality of (45) becomes an equality when f is a maximum flow for some cut $C = (S,T)$, and that all such cuts are minimum cuts.

But how does this help us? We know that if we know the maximum flow value, and we have an s-t-cut with capacity equal to the maximum flow, we actually have a minimum cut. The question is then, how do we find a maximum flow, and how do we find a minimum cut?

5.2 Augmenting path algorithms

The family of augmenting flow algorithms represent a popular approach to the maximum flow problem. The idea is simply to look for paths from the source to the sink that can carry additional flow, so-called augmenting paths, and then send the maximum possible amount of flow along such a path. When no such path exists anymore, no more flow can be sent from the source to the sink, and a maximum flow has been reached.

5.2.1 Residual network

When further discussing approaches to solving the maximum flow problem we will need the notion of a residual network $G_f = (V_f, E_f, c_f)$, which is derived from the original network G and contains the edges along which it is possible to send additional flow. This means that E_f contains the edges (u,v) from E where $f(u,v) < c(u,v)$. But that is not all; an important realisation is that it is also possible to push flow *back* along an edge where the flow is already positive. In other words, sending flow from v to u by cancelling some or all of the flow that is already going from u to v.

Thus the capacity function c_f of our residual network becomes

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E, \\ f(v,u) & \text{if } (v,u) \in E, \\ 0 & \text{otherwise.} \end{cases} \tag{46}$$

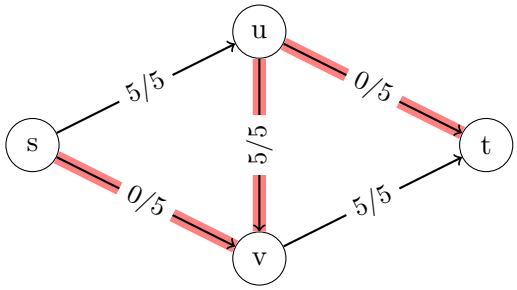


Figure 6: A network with the flow and capacity of each edge shown as flow/-capacity. The marked path is a valid augmenting path from s to t, and by increasing the flow along it, we will push flow back from v to u.

The vertices V_f of G_f are the same as the original network G, while the edges E_f are taken to be all pairs of vertices (u, v) with $c_f(u, v) > 0$. Since we can in E_f at most have all the original edges, and their reversals, we have $|E_f| \leq 2|E|$.

Note that there is ambiguity in the definition of the residual network in the case where the original network contains anti-parallel edges. One could avoid this by defining $c_f(u, v) = f(v, u) + c(u, v) - f(u, v)$ instead, or as they do in [8], disallow anti-parallel edges altogether. In any case it is not something we will have to think about in the implementation, since we will not actually construct the residual network.

With the residual network defined, we are ready to formally present the max-flow min-cut theorem.

Theorem 4 (Max-flow min-cut theorem)**.** *If f is a flow in a network $G = (V, E, c)$ with source s and sink t, then the following statements are equivalent:*

1. *f is a maximum flow in G.*
2. *The residual network G_f contains no augmenting paths.*
3. *$|f| = c(S, T)$ for some cut (S, T) of G.*

See [8] for a proof, and remark that because of the inequality in (45), the cut in Statement 3 is a minimum cut. The theorem does not tell us how to find such a cut, and there are multiple ways. One possibility is to take S to be all vertices reachable from the source in the residual network and $T = V - S$.

Figure 6 shows a simple network which already has five units flowing from s to t. The marked path is a possible augmenting path, and note that it follows an edge in E in the reverse direction, made possible by the construction of the residual network just described.

5.2.2 Ford-Fulkerson

The Ford-Fulkerson algorithm is the most basic augmenting path algorithm, which can be extended to more advanced algorithms. It is stated in pseudocode in Algorithm 1,

and the idea is to augment the flow along paths from s to t until it is no longer possible.

Algorithm 1 The Ford-Fulkerson max-flow algorithm

```
function FORD-FULKERSON(G, s, t)
    while there exists a path p from s to t in the residual network G_f do
        α ← min{c_f(u, v) : (u, v) ∈ p}
        for all (u, v) ∈ p do
            if (u, v) ∈ E then
                f(u, v) += Δf
            else
                f(v, u) −= Δf                          ▷ Push flow back
            end if
        end for
    end while
end function
```

There are different ways to find augmenting paths, and a common choice is to do a breadth-first search from the source until the sink is found, as this will yield the shortest possible augmenting path. This version of the algorithm is called Edmonds-Karp and has a running time of $O(|V||E|^2)$. See for example [8] for a description of the breadth-first search, and a formal proof of the running time of the Edmonds-Karp algorithm.

5.3 Other algorithms

There are many different maximum flow algorithms that fall into the augmenting path category, although we will see a different approach in the next section.

The algorithm of Dinitz, originally published in 1970, later improved on, and described by the original author in [10], is a variant of the augmenting path algorithm. It maintains a distance labeling $d(u)$ of the vertices $u \in V$ in the network, where $d(u)$ is the shortest path from the s to u in the residual network. This can be computed with a simple breadth-first search. The next step is to construct a *blocking flow* f', using only edges in $E'_f = \{(u, v) \in E_f : d(u) + 1 = d(v)\}$. The blocking flow is such that if we augment the flow f by f', there is no longer any paths from s to t following edges in E'_f. After the blocking flow f' has been found and added to f, the distance labels are recalculated, and the label of the sink will be increased by at least one.

Boykov and Kolmogorov present a variant of the augmenting path algorithm in [11], specialized for the kinds of graphs occuring in graphical applications. It keeps two non-overlapping trees of non-saturated edges, one with the source and the other with the sink as its root. These trees are "grown" towards eachother and augmenting paths are found when their leaf nodes touch. In theory, the complexity of this algorithm is not great, but according to Boykov and Kolmogorov, it outperforms the other algorithms in experimental comparisons for this specific application.

6 The push-relabel algorithm

The push-relabel algorithm is a different approach to the maximum flow problem, presented by Goldberg and Tarjan in [12]. Unlike the augmenting flow algorithms, it does not maintain a valid flow f in the network at all times, but still obtains a valid maximum flow when the algorithm terminates.

6.1 Preflow

Instead of maintaining a valid flow, we introduce the concept of a *preflow* by relaxing the flow conservation constraint from earlier. For a preflow f, the capacity constraint still holds so the flow must always be less than the capacity, but we allow positive excess in the vertices. The flow conservation constraint from before then becomes

Preflow conservation: For all $u \in V - \{s, t\}$

$$\sum_{v \in V} f(v, u) \geq \sum_{v \in V} f(u, v), \tag{47}$$

i.e., for any vertex except the source and the sink, the flow into the vertex must greater or equal to the flow out of the vertex.

As in most of the cited push-relabel literature, we define $N = |V|$, and for all vertices $u \in V$ we define the excess

$$e(u) = \sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v), \tag{48}$$

which represents the amount of flow which *disappears* in vertex u. Equivalent to the preflow conservation constraint is stating that $e(u) \geq 0$ for all vertices $u \in V - \{s, t\}$.

The idea of the algorithm is to maintain a height map of the vertices in the network where connected vertices can not have a large height difference. Then we "lift" the source vertex to let as many units as possible flow through the edges of the network towards the sink. When a maximum preflow is reached, there will normally be excess flow in some of the vertices, which has to be pushed back towards the source in order to obtain a valid flow.

The height map $d : V \to \mathbb{N}$ satisfies $d(t) = 0$, and for every edge (u, v) in the residual network, i.e. every edge with $c_f(u, v) > 0$, we require that $d(u) \leq d(v) + 1$. For all vertices u, the label $d(u)$ will be a lower bound on the length from u to t in G_f which is why it is also often called a distance labeling.

For every vertex u for which there is a path through the residual network to the sink, $d(u)$ will be a lower bound on the length of such a path.

A vertex u is *active* if $u \in V - \{s, t\}$, it has positive excess ($e(u) > 0$) and $d(u) < N$. These are the vertices we want to operate on to increase the preflow.

6.2 Basic operations

The algorithm performs two basic operations, the *push* and *relabel* operations, while always maintaining a valid preflow f and a valid distance labeling d.

6.2.1 The push procedure

The push procedure moves excess flow from an active vertex along an edge $(u, v) \in E_f$ for which $d(u) = d(v) + 1$, i.e. to a vertex with a smaller distance label. We call such edges *admissible*. See Algorithm 2 for a pseudocode implementation of the push operation.

Assuming that f is a valid preflow, it is easy to verify that f remains a valid preflow after running the push procedure on some admissible edge (u, v).

The residual network might change during the push procedure. The edge (v, u) will appear if it does not already exist. Initially $d(u) = d(v) + 1$, so for the new edge (v, u) we have

$$d(v) = d(u) - 1 \leq d(u) + 1 \tag{49}$$

and d remains a valid labeling after the push procedure is finished.

Algorithm 2 The push procedure of the Push-Relabel algorithm.

```
function PUSH(u, v)
    Δf ← min(c_f(u, v), e(u))
    if (u, v) ∈ E then
        f(u, v) += Δf
    else
        f(v, u) −= Δf                          ▷ Push flow back
    end if                                     ▷ Excess e(u) and e(v) will also change
end function
```

6.2.2 The relabel procedure

The relabel procedure is our tool for changing the distance labels of the vertices. It changes the label of a vertex to the greatest possible value, which is one more than the lowest label among its neighbors in the residual graph. See Algorithm 3 for a rather mathematical pseudocode implementation.

Algorithm 3 The relabel procedure of the Push-Relabel algorithm

```
function RELABEL(u)
    if there is a v ∈ V such that (u, v) ∈ E_f then
        d(u) ← min{d(v), ∀v ∈ V : (u, v) ∈ E_f} + 1
    else
        d(u) ← N                               ▷ u becomes inactive
    end if
end function
```

If d was a valid labeling before running the relabel procedure, then we still have $d(u) \leq d(v) + 1$ for all neighbors v of u in the residual graph, and d remains a valid labeling. The capacity constraint and preflow constraint remain satisfied assuming they were satisfied before the procedure was started.

6.3 Putting it all together

These basic procedures are then applied to active vertices and admissible edges until we obtain our minimum cut. We will see later that when there are no more active vertices, we can extract the minimum cut from the network.

In the first phase of the algorithm we initialize a valid preflow and distance labeling by saturating all edges out of the source s, and then setting its distance label $d(s) = N$. We then apply the push and relabel procedures where applicable until there are no more active vertices and a maximum preflow is obtained.

A vertex u can only be successfully relabeled to obtain a new label if the outgoing edges of u in the residual network have changed since the previous relabeling. This is why the push and relabel procedures often are combined into a *discharge* procedure as shown in Algorithm 4. When it is run on an active vertex u, we push as much as possible of the excess flow to other vertices before the vertex is relabeled.

Algorithm 4 The discharge procedure of the Push-Relabel algorithm

```
function DISCHARGE(u)
    for all v ∈ V such that (u, v) ∈ E_f do
        if c_f(u, v) > 0 and d(u) = d(v) + 1 then PUSH(u, v)
        end if
    end for
    if e(u) > 0 then RELABEL(u)
    end if
end function
```

In the second phase of the algorithm this maximum preflow is turned into a maximum flow by returning excess flow which did not reach the sink from inside the network back to the source. We can skip this part of the algorithm, as it is possible to identify a minimum cut as soon as the first phase is finished, and the following theorem allows us to do that.

Theorem 5 (Cut identification)**.** *Given a network $G = (V, E, c)$, assume that the first phase of the push-relabel algorithm has terminated so that no more active vertices remain. Then there exists a $k \in \{1, \dots, N-1\}$ such that there is no vertex with label k. For every such k the vertex sets $S = \{u \in V : d(u) > k\}$ and $T = \{u \in V : d(u) < k\}$ define a minimum cut $C = (S, T)$ in G.*

Proof. There are N vertices, the source has label N and the sink has label 0, and the $N-2$ remaining vertices can not occupy all the $N-1$ labels in $\{1, \dots, N-1\}$, so there must exist an k as described.

There can be no edge $(u, v) \in E_f$ such that $u \in S$ and $v \in T$, as this would imply $k \leq d(u) - 1 \leq d(v) < k$. From the construction of E_f we now know that all edges in E from S to T are saturated, and all edges from T to S carry no flow. This implies the capacity of the cut is equal to the flow through the cut, i.e. $c(S, T) = f(S, T)$.

Since the first phase of the algorithm has terminated, there can be no active vertices, and therefore no excess in T, except for the sink. If all flow excess in vertices in S is returned to the source, we can apply the max-flow min-cut theorem to conclude that $C = (S, T)$ is a minimum s-t-cut, since the cut capacity is equal to the flow. □

We will see later that with the gap relabeling heuristic, there will always be a gap at label $k = N - 1$ such that we can construct our cut by taking $S = \{u \in V : d(u) \geq N\}$.

Note that the vertices in S are vertices earlier described as being on the source side of the cut, as no additional flow can go from these vertices to the sink.

6.4 Complexity

In their original article [12], Goldberg and Tarjan analyze the complexity of the push-relabel algoritm by considering the maximum number of basic operations we can possibly do before the algorithm terminates.

The number of relabelings is in $O(|V|^2)$ since every time the procedure is applicable to a vertex u, the label $d(u)$ increases by at least one.

The number of saturating pushes is in $O(|V||E|)$. When a push along (u, v) is saturating, the label of v has to increase with at least 2 before a push can saturate the same edge (in the opposite direction). Since the number of relabelings of a single vertex is bounded by $|V|$, and we have $|E|$ edges, this gives the stated number of saturating pushes.

The number of non-saturating pushes is the most complicated to bound, and will also make up the asymptotic running time of the algorithm. The idea is to define

$$\phi = \sum_{u \text{ active}} d(u), \tag{50}$$

and look at how much this number changes throughout the algorithm. It starts at zero and ends at zero. Every non-saturating push from u to v makes ϕ decrease by at least one since it makes u inactive (but might activate v). The total increase in ϕ due to relabelings is less than $|V|^2$. A saturating push from u to v increases ϕ by at most $|V|$, since v might become active.

Even if ϕ is always increased by relabelings and saturating pushes, we can bound the number of non-saturating pushes by

$$|V|^2 + |V| \underbrace{c|V||E|}_{\#(\text{saturating pushes})} \tag{51}$$

which means that in the general case, the algorithm has a complexity of $O(|E||V|^2)$.

6.5 Vertex selection rules

Until now we have stated that the discharge procedure is run on active vertices until there are no more active vertices left. The choice of the order in which to discharge these active vertices remain, and multiple possibilities exist.

The First In, First Out (FIFO) approach is to always maintain a queue of active vertices. When the vertex from the beginning of the queue is discharged, other vertices might become active, and these are added at the end of the queue. The original article of Goldberg and Tarjan [12] contains a proof that this selection rule gives a complexity of $O(|V|^3)$, which can be very good if you have a dense network.

Using highest level selection rule one always discharges the active vertex with the largest distance label. Goldberg and Tarjan state that this rule also gives a complexity of $O(|V|^3)$ while this bound is improved to $O(|V|^2\sqrt{|E|})$ in an article by Cheriyan and Maheshwari [13].

These are complexity bounds, and the actual running time of the algorithm, which can only be determined by implementing it and running it, varies a lot with the structure of the input network.

Cherkassky and Goldberg describe the algorithm along with different selection rules, heuristics and their implementation in [14].

6.6 Heuristics

Different heuristics exist that can speed up the algorithm considerably. Being heuristics, they are not guaranteed to work, and might perform differently on different kinds of networks. The most used heuristics are the gap and global relabeling heuristics, both aiming to reduce the total number of relabeling steps.

The gap relabeling heuristic aims to find a label k as in Theorem 5 such that no vertex has that label. From vertices u with $d(u) > k$, there are no unsaturated edges going to vertices with smaller distance labels, so no more flow can ever find its way from these vertices to the sink. These vertices are therefore given the label N and never considered again as they will never become active. Algorithm 5 shows a pseudocode representation of what is done once a gap k is found.

Algorithm 5 The gap procedure of the Push-Relabel algorithm

```
function GAP(k)
    for all u ∈ V such that d(u) > k do
        d(u) ← N
    end for
end function
```

Before integrating the gap relabeling procedure into our algorithm we need to verify that it does not invalidate our preflow f or distance labeling d.

Lemma 1 (Gap relabeling validity)**.** *Given a network $G = (V, E, c)$, a distance labeling d and a preflow f, assume there exists a gap k such that no vertex has label k. Running the gap relabeling procedure on label k will yield a valid distance labeling and an unchanged and valid preflow f.*

Proof. No new edges are created, no edges disappear, the preflow is unchanged, so the preflow and capacity constraint remain satisfied after the gap relabeling.

Define $S = \{u \in V : d(u) > k\}$ and $T = V - S$. Initially $d(u) \leq d(v) + 1$ for every edge $(u, v) \in E_f$. These inequalities have to hold after the gap procedure is finished, when $d(u) = N$ for all $u \in S$.

For $(u, v) \in E_f$ we have four possibilities

$u, v \in T$: The labels $d(u)$ and $d(v)$ remain unchanged and the inequality still holds.

$u, v \in S$: After the gap procedure we have $d(u) = d(v)$ so the inequality still holds.

$u \in S, v \in T$: This is not possible as it would imply $d(u) \geq d(v)+2$ and we have assumed an initial valid labeling.

$u \in T, v \in S$: After relabeling we have $d(u) < k < N < d(v) + 1$.

Hence, both the preflow f and distance labeling d are valid. □

When running the push-relabel algorithm with the gap heuristic, we can be sure that there will never be a vertex u with label $d(u) = N - 1$ at the end of the algorithm, i.e. we know that there will always be a gap at label $N - 1$. This can be seen using the same reasoning as in Theorem 5, because if there was a vertex with label $N - 1$, there would only be $N-3$ vertices possibly having labels in $\{1, \dots, N-2\}$, so a gap must exist somewhere in that interval. When using the gap relabeling heuristic, such a gap can not exist, so we can conclude that there is no vertex with label $N - 1$.

Using Theorem 5 we can then conclude that the sets $S = \{u \in V : d(u) \geq N\}$ and $T = V - S$ form a minimum cut of the network.

6.7 Parametric push-relabel algorithm

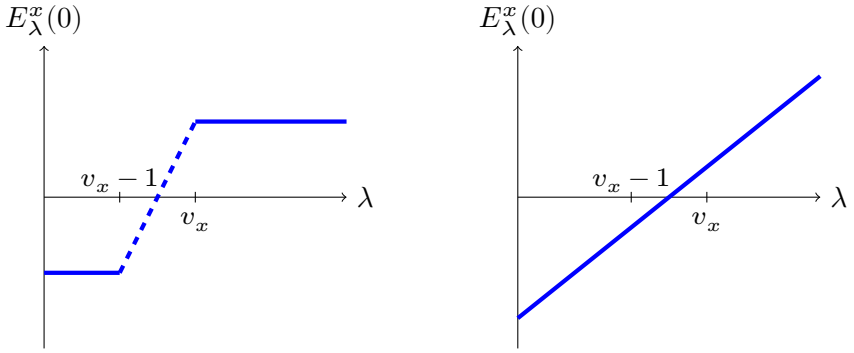
Now we have an algorithm for finding a minimum s-t-cut in a network, so let's return to the network constructed in Section 4.3. For every level $\lambda \in \{0, \dots, L\}$ we want to find a minimum s-t-cut which gives us the thresholded image u^λ. These can then hopefully be stacked together to form the final image u.

6.7.1 Network reuse

Solving L separate minimum cut problems seems like a lot of work, but when using the push-relabel algorithm we will, if we do things in the right order, be able to reuse the network when going from one label to the next.

Going back to the network representations in Figure 3 and Figure 4 we know that only the capacity of edges from sub-networks representing the fidelity term depend on our level parameter λ. From the expression in (37), visualized in Figure 7, we see that the energy term $E^x_\lambda(0)$ increases monotonically with increasing λ parameter. Let $u, v \in V - \{s, t\}$. Since the edges in Figure 3 now are the only ones depending on λ, the following is true for *decreasing* values of λ

Edges from s to u: As seen in Figure 3b the capacity of these edges will increase monotonically with decreasing λ parameter.



(a) L^1 fidelity term. (b) L^2 fidelity term.

Figure 7: Two figures showing how the fidelity energy term $E^x_\lambda(0)$ in (29) increases monotonically with λ.

Edges from u **to** v: These edges have no λ-dependence and will remain unchanged.

Edges from v **to** t: As seen in Figure 3a the capacity of these edges will decrease monotonically with decreasing λ parameter.

After running the push-relabel algorithm for $\lambda = k$, we are left with a network $G = (V, E, c)$, a preflow f and a labeling d. To obtain the network for $\lambda = k-1$ we have to change the capacity of two different kinds of edges, and this is done in the following way to keep the capacity, preflow and labeling constraints satisfied.

Edges from s **to** u: The capacity $c(s, u)$ is increased, and the flow is set to be equal to the capacity $f(s, u) = c(s, u)$. The vertex u might have an increased excess $e(u)$, which might in turn make it active.

Edges from v **to** t: The capacity $c(v, t)$ is decreased. If it is decreased to a value below the current flow value, we set $f(v, t) = c(v, t)$ which will decrease the excess of the sink t, and increase the excess of v.

None of these actions will create new edges in the residual network, and we do not change the labeling d, so the labeling constraints are also satisfied in the new network.

Through this procedure we have easily created the network for $\lambda = k - 1$, and the distance labels remain the same. As these labels always increase monotonically, we have a head start compared to the case where we reset the flow and labels.

6.7.2 Output image construction

We mentioned already in Section 3.3.3 that in order to be able to construct our output image u, the thresholded images u^λ would have to stack one on top of the other as shown in Figure 1. Because of the reuse of the distance labels between the iterations of the push-relabel algorithm, we can guarantee that this is possible.

Consider two subsequent runs of the push-relabel algorithm, for labels λ and $\lambda - 1$ ending with distance labels d^λ and $d^{\lambda-1}$ respectively. We already know that the distance labels d are monotonically increasing. This means that the set $S = \{u \in V : d(u) \geq N\}$ is increasing in size, more precisely, we have the inclusion

$$\{u \in V : d^\lambda(u) \geq N\} \subseteq \{u \in V : d^{\lambda-1}(u) \geq N\}. \tag{52}$$

For a pixel $x \in S$ we will set $u_x^\lambda = 0$, which together with the previous inclusion property implies

$$u_x^\lambda \geq u_x^{\lambda-1} \tag{53}$$

for all $x \in \mathcal{G}$. Being equivalent with the inequality in (32), this means our algorithm produces stackable thresholded images u^λ.

We then construct our output image u by giving each pixel the value

$$u_x = \min\{\lambda \in \{0, \dots, L-1\} : u_x^\lambda = 1\}. \tag{54}$$

This marks the end of the description of the implemented algorithm, but we will further discuss some possible improvements, and also look at the results when using the method on different kinds of noisy images.

6.8 Divide and conquer

The possibility of re-using the network between separate level is a very nice property of the push-relabel algorithm, but there are further room for improvements. Consider one pixel x with value u_x, and imagine we only wanted to find the value of this pixel. One could go through all pixel values $\lambda \in (L-1, \dots, 0)$, and see when u_x^λ changes from 1 to 0, just as we do for all the pixels in the algorithm above. Ignoring network re-use this would have us solve $O(L)$ maximum flow problems.

Improving on this we could employ the idea of binary search to find the value of u_x in only $O(\log_2 L)$ time. After finding one cut, we know whether u_x is above or below the current λ value, and by choosing λ as the midpoint of the current possible range of u_x, we can cut the search space in half for each iteration of the algorithm.

We can extend this idea to the problem of finding all pixel values. Instead of running the algorithm for successively decreasing values of λ, we choose some λ in the middle of the range $\{0, \dots, L-1\}$. The cut we obtain consists of two sets $S = \{u \in V : d(u) \geq N\}$ and $T = V - S$. We know that no more flow can be sent from S to T, even if we decrease the value of λ and adjust the capacities accordingly.

The idea is now that we have halved the possible λ interval for *all* pixels. We continue by considering the two sets S and T separately, and applying the algorithm recursively, at each time halving the λ interval until we have the value of every pixel.

Combining the divide and conquer approach with the parametric push-relabel algorithm is a bit problematic. For all pixels $x \in T$ we know that $u_x \leq \lambda$, and we can reuse the network when decreasing λ. However for pixels $x \in S$, we seek to find $u_x > \lambda$, meaning we have to increase λ which does not allow network re-use.

(a) Original image (b) Gaussian noise, $\mu = 0$, $\sigma = 30$. (c) Laplace noise, $\mu = 0$, $\lambda = 30$.

Figure 8: The classical Lena Söderberg portrait, with two different types of additive noise.

Goldfarb and Yin [15] have found that the divide and conquer approach only yields improved performance when using the L^2 norm in the fidelity term. This has to do with the fact that the fidelity term for the L^1 norm only changes once for each pixel, as we can see in Figure 7, reducing the amount of work that has to be done in each iteration of the regular parametric push-relabel algorithm.

See [16], [17] and [15] for more information.

6.9 Implementation

A C++ implementation can be found in Appendix A. It uses the open computer vision library OpenCV [18] to load and save image files.

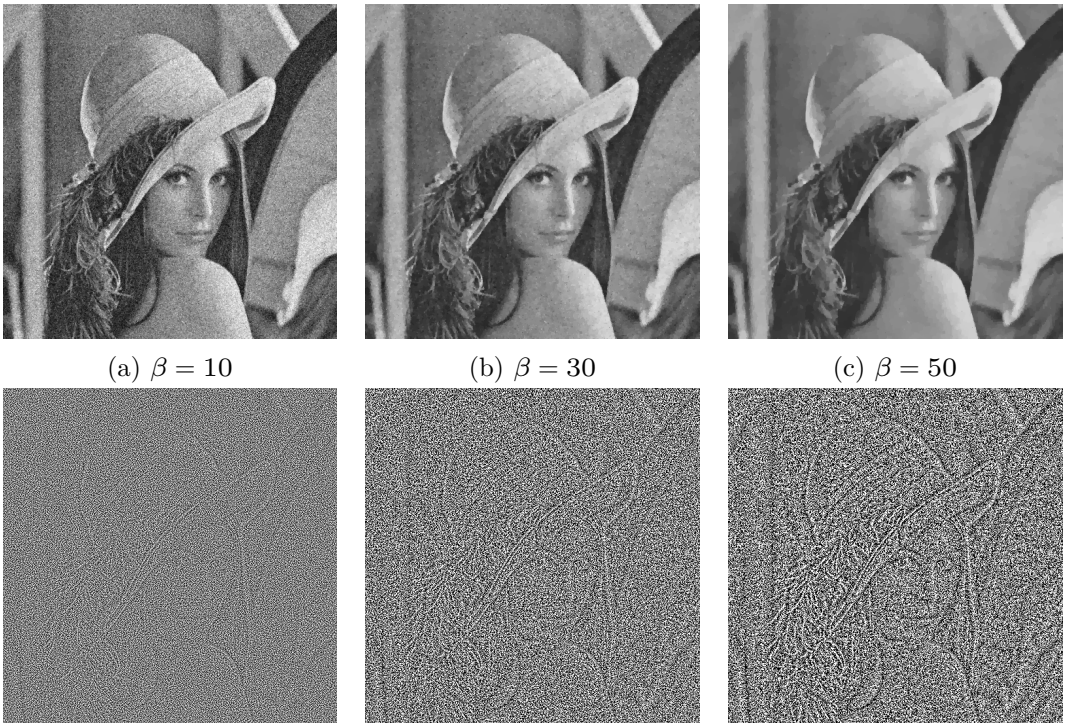
Note that when implementing maximum flow algorithms it is not a good idea, memory- and performance-wise, to actually construct the residual network G_f. Instead, every time we update the flow $f(u, v)$ we set the flow in the opposite direction to its negative value $f(v, u) = -f(u, v)$. Then we can at any time, consider the value $c(u, v) - f(u, v)$ in the place of the residual capacity $c_f(u, v)$.

For the gap relabeling heuristic, we need to have a easy way of finding when a gap occurs. This is done by keeping track of how many vertices exist with each label.

7 Image restoration results

Although the theory behind the graph cut image restoration algorithm has been the main focus of this project, we will look at some results when applying the algorithm to different noisy input images.

The two main parameters available to the user of the algorithm are the β parameter and the neighborhood specification. With β being the weight of the total variation in the energy function, larger values will give more smoothing in the output image. By



(a) $\beta = 10$ (b) $\beta = 30$ (c) $\beta = 50$

Figure 9: The first row shows the image with additional Gaussian noise restored for different values of β, using a size four neighborhood and $p = 2$ as the exponent in the fidelity term. In the second row the method noise is shown around a gray value of 127 with a scaling of 10.

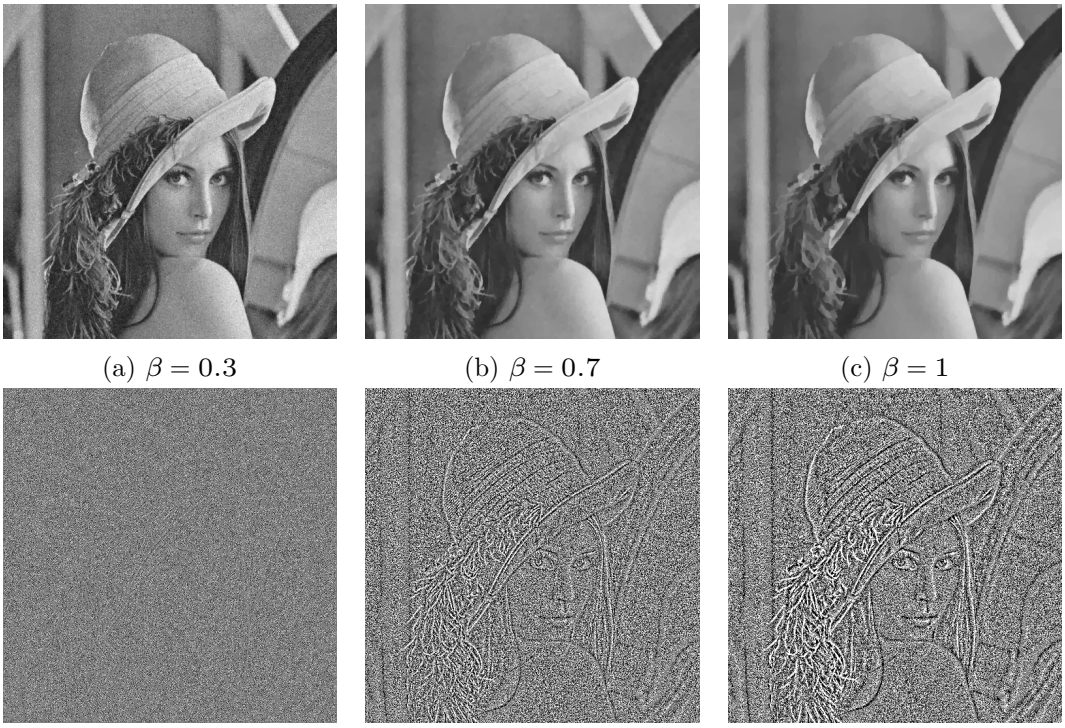
changing the pixel neighborhoods we can change the reach of the smoothing, and maybe try to reduce some visual effects introduced by the discretization.

Measuring the performance of the method is hard, especially as different applications have different measures for what is a "good" output image. We will in this section consider an original image with artificially added noise, and try to remove the noise to obtain an output image as close to the original as possible.

The most obvious approach is to just look at the two images and see how much alike they are, something that makes sense especially if the output is made for the eye to see. One can also consider the *method noise* which is the difference between the noisy image and the de-noised version. If noise is independent of the original image, the method noise would optimally not contain too many features from the original image, since it is only the noise we want to remove.

Since our digitized representation of the images we are restricted to pixel values in $\{0, \dots, 255\}$, we show the method noise as an image with pixel values $127 + \alpha(v - u)$. The scaling α can be tuned depending on the magnitude of difference $v - u$.

Figure 8 shows the Lena Söderberg portrait which has been used as a test image in digital imaging countless times. Gaussian and Laplace noise has been added. Since the



(a) $\beta = 0.3$ (b) $\beta = 0.7$ (c) $\beta = 1$

Figure 10: The first row shows the image with additional Laplace noise restored for different values of β, using a size four neighborhood and $p = 1$ as the exponent in the fidelity term. In the second row the method noise is shown around a gray value of 127 with a scaling of 10.

pixel values in these gray scale images are limited to the range $\{0, \dots, 255\}$, we have to truncate the pixel value if the addition of noise makes it exit the interval.

Figure 9 and 10 shows the noisy images from Figure 8 restored for different parameters of β and the exponent p in the fidelity term. First we note that the method works quite well on smooth surfaces, like Lena's shoulder and the background of the image, while her hair, and the feather boa hanging from her hat has lost details, especially for higher values of β.

We see in the method noise, especially in Figure 10c, that the feather boa in Lena's hat has lost details. Being especially bright, the method noise here tells us that the algorithm has corrected this area more than the rest, all the while the noise was spread evenly over the whole image.

As mentioned earlier, one of the theoretically nice properties of the total variation method is that it preserves edges. Looking at the restored images, we see that the edges in the original image are sharp even for high values of β, taking Lena's shoulder as an example again.

Table 3: Performance comparison of the implemented algorithms, and the algorithm of Darbon and Sigelle [6] using sections of the Lena test image of increasing size, with the L^2 norm in the fidelity term, $\beta = 10$ and a size four neighborhood. The Push-Relabel algorithm is run both using the highest level selection rule (HL) and the first-in first-out selection rule (FIFO). All numbers are in seconds.

Method	64×64	128×128	256×256	512×512
Dinitz	0.55	5.75	36.7	279
Push-Relabel HL	0.070	0.380	2.46	19.3
Push-Relabel FIFO	0.059	0.335	1.83	8.51
Darbon and Sigelle	0.013	0.043	0.162	0.627

7.1 Algorithm performance

Table 3 shows the how the implemented algorithms perform on test images of different sizes. The binary provided by Darbon and Sigelle at [19], implemented following the description in [6], has also been timed for comparison. They use the specialized algorithm of Boykov and Kolmogorov [11], and additionally a divide and conquer approach which could explain the superior performance. All performance experiments were performed on a Lenovo Thinkpad X230 Laptop with an Intel® Core™ i5-3320M CPU at 2.60 GHz and 8 GB of memory.

References

[1] Joachim Weickert. *Anisotropic diffusion in image processing*. European Consortium for Mathematics in Industry. B. G. Teubner, Stuttgart, 1998.

[2] Otmar Scherzer, Markus Grasmair, Harald Grossauer, Markus Haltmeier, and Frank Lenzen. *Variational methods in imaging*, volume 167 of *Applied Mathematical Sciences*. Springer, New York, 2009.

[3] Leonid I Rudin, Stanley Osher, and Emad Fatemi. Nonlinear total variation based noise removal algorithms. *Physica D: Nonlinear Phenomena*, 60(1–4):259–268, 1992.

[4] V. Caselles, A. Chambolle, and M. Novaga. Total variation in imaging. In Otmar Scherzer, editor, *Handbook of Mathematical Methods in Imaging*, pages 1016–1057. Springer New York, 2011.

[5] Antonin Chambolle. An algorithm for total variation minimization and applications. *J. Math. Imaging Vision*, 20(1-2):89–97, 2004. Special issue on mathematics and image analysis.

[6] Jérôme Darbon and Marc Sigelle. Image restoration with discrete constrained total variation. Part I: Fast and exact optimization. *J. Math. Imaging Vision*, 26(3):261–276, 2006.

[7] Yuri Boykov and Vladimir Kolmogorov. Computing geodesics and minimal surfaces via graph cuts. In *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pages 26–33. IEEE, 2003.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, third edition, 2009.

[9] Vladimir Kolmogorov and Ramin Zabih. What energy functions can be minimized via graph cuts? In *Computer Vision—ECCV 2002*, pages 65–81. Springer, 2002.

[10] Yefim Dinitz. Dinitz' algorithm: the original version and Even's version. In *Theoretical computer science*, volume 3895 of *Lecture Notes in Comput. Sci.*, pages 218–240. Springer, Berlin, 2006.

[11] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(9):1124–1137, 2004.

[12] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. Assoc. Comput. Mach.*, 35(4):921–940, 1988.

[13] Joseph Cheriyan and S. N. Maheshwari. Analysis of preflow push algorithms for maximum network flow. *SIAM Journal on Computing*, 18(6):1057–1086, 1989.

[14] B. V. Cherkassky and A. V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[15] Donald Goldfarb and Wotao Yin. Parametric maximum flow algorithms for fast total variation minimization. *SIAM Journal on Scientific Computing*, 31(5):3712–3743, 2009.

[16] Giorgio Gallo, Michael D. Grigoriadis, and Robert E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM J. Comput.*, 18(1):30–55, 1989.

[17] Dorit S. Hochbaum. An efficient algorithm for image segmentation, Markov random fields and related problems. *J. ACM*, 48(4):686–701 (electronic), 2001.

[18] G. Bradski. Opencv library. *Dr. Dobb's Journal of Software Tools*, 2000.

[19] Jérôme Darbon, Total Variation Minimization. `http://jerome.berbiqui.org/total-variation-code/`. Accessed: 2014-02-10.

A C++ implementation

The files `graph.hpp` and `graph.cpp` contains the code related to the graph datastructure and the minimum cut algorithms. The code for the highest label and FIFO selection rules are in `selecitonrule.hpp` and `selectionrule.cpp`, while the neighborhood stencil is stored in a datastructure defined in `neighborhood.hpp`. The files `image.hpp` and `image.cpp` handle the connection between the digital image and the flow network, while command line input is handled in `main.cpp`. A digital copy of the code can be found at `http://github.com/burk/image-restoration`.

A.1 graph.hpp

```
#pragma once

#include <iostream>
#include <queue>
#include <vector>
#include <set>
#include "selectionrule.hpp"

class Edge {
private:

public:
    int from, to;
    int cap;
    int flow;
    int index;

    Edge(int f, int t, int c, int i) :
        from(f), to(t), cap(c), flow(0), index(i) {}
};

class FlowGraph {
private:
    int N;
    std::vector<std::vector<Edge> > G;
    std::vector<int> excess;
    std::vector<int> height;
    std::vector<int> count;
    SelectionRule& rule;

public:
    std::vector<char> cut;

    FlowGraph(int N, SelectionRule& rule) :
        N(N),
        G(N),
        excess(N),
        height(N),
        cut(N),
```

```
        count(N+1),
            rule(rule) {}

    int addEdge(int from, int to, int cap);
    void addDoubleEdge(int from, int to, int cap);
    void changeCapacity(int from, int index, int cap);
    void resetFlow();
    void resetHeights();

    void push(Edge &e);
    void relabel(int u);
    void gap(int h);
    void discharge(int u);
    void minCutPushRelabel(int source, int sink);

    void minCutDinic(int source, int sink);
    int maxFlowDinic(int source, int sink);
    int blockingFlow(std::vector<int> &level, int u,
            int source, int sink, int limit);
    void DFS(int source, int sink);

    int outFlow(int source);
    int inFlow(int sink);

    bool checkExcess(void);
    bool checkCapacity(void);
    bool checkLabels(void);
    bool checkCount(void);
};
```

A.2 graph.cpp

```
#include <iostream>
#include <queue>
#include <stack>
#include <vector>
#include <cassert>

#include "graph.hpp"

using namespace std;

/* Add an edge from one vertex to another. */
int FlowGraph::addEdge(int from, int to, int cap) {
    G[from].push_back(Edge(from, to, cap, G[to].size()));
    if (from == to) G[from].back().index++;
    int index = G[from].size() - 1;
    G[to].push_back(Edge(to, from, 0, index));

    return index;
}

/*
 * Add an edge and at the same time an antiparallel edge
```

```
 * with the same capacity.
 */
void FlowGraph::addDoubleEdge(int from, int to, int cap) {
    G[from].push_back(Edge(from, to, cap, G[to].size()));
    G[to].push_back(Edge(to, from, cap, G[from].size() - 1));
}

/*
 * Change the capacity of an edge. Need the from-vertex and
 * the index of the edge in its edge list (returned from addEdge.
 */
void FlowGraph::changeCapacity(int from, int index, int cap) {
    int to = G[from][index].to;
    int diff = G[from][index].flow - cap;

    G[from][index].cap = cap;

    if (diff > 0) {
        excess[from] += diff;
        excess[to] -= diff;
        G[from][index].flow = cap;
        G[to][G[from][index].index].flow = -cap;
#ifndef DINIC
        rule.add(from, height[from], excess[from]);
#endif
    }
}

/* Reset all flow and excess. */
void FlowGraph::resetFlow() {
    for (int i = 0; i < G.size(); ++i) {
        for (int j = 0; j < G[i].size(); ++j) {
            G[i][j].flow = 0;
        }
    }
    fill(excess.begin(), excess.end(), 0);
}

/* Reset all distance labels. */
void FlowGraph::resetHeights() {
    fill(height.begin(), height.end(), 0);
    fill(count.begin(), count.end(), 0);
}

/* Push along an edge. */
void FlowGraph::push(Edge &e) {
    int flow = min(e.cap - e.flow, excess[e.from]);
    excess[e.from] -= flow;
    excess[e.to]   += flow;
    e.flow += flow;
    G[e.to][e.index].flow -= flow;

    rule.add(e.to, height[e.to], excess[e.to]);
```

```
}

/* Relabel a vertex. */
void FlowGraph::relabel(int u) {
    count[height[u]]--;
    height[u] = 2*N;

    for (int i = 0; i < G[u].size(); ++i) {
        if (G[u][i].cap > G[u][i].flow) {
            height[u] = min(height[u], height[G[u][i].to] + 1);
        }
    }

    if (height[u] >= N) {
        height[u] = N;
    }
    else {
        count[height[u]]++;
        rule.add(u, height[u], excess[u]);
    }
}

/* Relabel all vertices over the gap h to label N. */
void FlowGraph::gap(int h) {
    int c = 0;
    for (int i = 0; i < G.size(); ++i) {
        if (height[i] < h) continue;
        if (height[i] >= N) continue;

        rule.deactivate(i);

        count[height[i]]--;
        height[i] = N;
    }

    rule.gap(h);
}

/* Discharge a vertex. */
void FlowGraph::discharge(int u) {
    int i;
    for (i = 0; i < G[u].size() && excess[u] > 0; ++i) {
        if (G[u][i].cap > G[u][i].flow
                && height[u] == height[G[u][i].to] + 1) {
            push(G[u][i]);
        }
    }

    if (excess[u] > 0) {
        /* Check if a gap will appear. */
        if (count[height[u]] == 1)
            gap(height[u]);
        else
```

```
            relabel(u);
    }
}

/* Run the push-relabel algorithm to find the min-cut. */
void FlowGraph::minCutPushRelabel(int source, int sink) {
    height[source] = N;

    rule.activate(source);
    rule.activate(sink);

    for (int i = 0; i < G[source].size(); ++i) {
        excess[source] = G[source][i].cap;
        push(G[source][i]);
    }
    excess[source] = 0;

    int c = 0;
    /* Loop over active nodes using selection rule. */
    while (!rule.empty()) {
        c++;
        int u = rule.next();
        discharge(u);
    }

    /* Output the cut based on vertex heights. */
    for (int i = 0; i < cut.size(); ++i) {
        cut[i] = height[i] >= N;
    }
}

/*
 * Depth-first-search used to find a cut at the
 * end of Diniz.
 */
void FlowGraph::DFS(int source, int sink) {
    stack<int> s;
    vector<bool> visited(N);
    s.push(source);

    cut[source] = true;
    while (!s.empty()) {
        int u = s.top();
        s.pop();

        for (int i = 0; i < G[u].size(); ++i) {
            if (cut[G[u][i].to]) continue;

            if (G[u][i].cap > G[u][i].flow) {
                assert(G[u][i].to != sink);
                cut[G[u][i].to] = true;
                s.push(G[u][i].to);
            }
```

```
        }
    }
}

/* Diniz min-cut algorithm. */
void FlowGraph::minCutDinic(int source, int sink) {
    maxFlowDinic(source, sink);

    fill(cut.begin(), cut.end(), false);
    DFS(source, sink);
}

/*
 * Recursive blocking flow algorithm, from the current
 * vertex u, from which we can at most send limit flow.
 */
int FlowGraph::blockingFlow(vector<int> &level, int u,
        int source, int sink, int limit) {
    if (limit <= 0) return 0;

    if (u == sink) return limit;

    int throughput = 0;
    for (int i = 0; i < G[u].size(); ++i) {

        int res = G[u][i].cap - G[u][i].flow;

        /* Recurse on vertices one level closer to sink. */
        if (level[G[u][i].to] == level[u] + 1 && res > 0) {
            int aug = blockingFlow(
                    level,
                    G[u][i].to,
                    source,
                    sink,
                    min(limit - throughput, res)
                    );

            throughput += aug;
            G[u][i].flow += aug;
            G[G[u][i].to][G[u][i].index].flow -= aug;
        }
    }

    if (throughput == 0) {
        level[u] = 0;
    }

    return throughput;
}

/* Diniz max-flow algorithm. */
int FlowGraph::maxFlowDinic(int source, int sink) {
```

```
    vector<int> level(N);

    while (true) {
        fill(level.begin(), level.end(), 0);

        queue<int> nq;

        /*
         * Fill the level vector using a basic breadth-first
         * search.
         */
        nq.push(source);
        level[source] = 1;
        while (!nq.empty()) {
            int u = nq.front();
            nq.pop();
            for (int i = 0; i < G[u].size(); ++i) {
                if (level[G[u][i].to] > 0) continue;

                if (G[u][i].cap > G[u][i].flow) {
                    level[G[u][i].to] = level[u] + 1;
                    nq.push(G[u][i].to);
                }
            }
        }

        if (level[sink] == 0) break;

        blockingFlow(level, source, source, sink, 1000000000);
    }

    return outFlow(source);
}

bool FlowGraph::checkExcess(void) {
    for (int i = 0; i < excess.size(); ++i) {
        if (excess[i] < 0) {
            return false;
        }
    }

    return true;
}

bool FlowGraph::checkCapacity(void) {
    for (int i = 0; i < G.size(); ++i) {
        for (int j = 0; j < G[i].size(); ++j) {
            if (G[i][j].flow > G[i][j].cap) return false;
        }
    }

    return true;
}
```

```
bool FlowGraph::checkLabels(void) {
    for (int i = 0; i < G.size(); ++i) {
        for (int j = 0; j < G[i].size(); ++j) {
            if (G[i][j].flow < G[i][j].cap) {
                if (height[i] > height[G[i][j].to] + 1) {
                    return false;
                }
            }
        }
    }

    return true;
}

bool FlowGraph::checkCount(void) {
    for (int i = 0; i < count.size(); ++i) {
        int c = 0;
        for (int j = 0; j < height.size(); ++j) {
            if (height[j] == i) c++;
        }
        if (c != count[i]) {
            cout << "c␣=␣" << c << ",␣count[" << i << "]␣=␣";
            cout << count[i] << endl;
            return false;
        }
    }

    return true;
}

int FlowGraph::outFlow(int source) {
    int c = 0;
    for (int i = 0; i < G[source].size(); ++i) {
        c += G[source][i].flow;
    }
    return c;
}

int FlowGraph::inFlow(int sink) {
    int c = 0;
    for (int i = 0; i < G[sink].size(); ++i) {
        c += G[sink][i].flow;
    }
    return c;
}
```

A.3 selectionrule.hpp

```
#pragma once

#include <iostream>
#include <queue>
#include <vector>
```

```
#include <exception>

class EmptyQueueException : public std::exception {
    virtual const char* what() const throw()
    {
        return "Empty␣Queue␣Exception";
    }
};

class SelectionRule {

private:
    std::vector<char> active;

protected:
    int N;

public:
    virtual int next(void) = 0;
    virtual void add(int u, int height, int excess) = 0;
    virtual bool empty(void) = 0;
    virtual void gap(int h) = 0;

    void activate(int u) { active[u] = 1; }
    void deactivate(int u) { active[u] = 0; }
    bool isActive(int u) { return active[u]; }

    SelectionRule(int N) : N(N), active(N) {}

    virtual ~SelectionRule() {}
};

class HighestLevelRule : virtual public SelectionRule {

private:
    int highest;
    std::vector<std::queue<int> > hq;

    void updateHighest(void);

public:
    virtual int next(void);
    virtual void add(int u, int height, int excess);
    virtual bool empty(void);
    virtual void gap(int h);

    HighestLevelRule(int N) : SelectionRule(N), highest(-1), hq(N) {}
};

class FIFORule : virtual public SelectionRule {

private:
    std::queue<int> q;
```

```
public:
    virtual int next(void);
    virtual void add(int u, int height, int excess);
    virtual bool empty(void);
    virtual void gap(int h);

    FIFORule(int N) : SelectionRule(N) {}
};
```

A.4 selectionrule.cpp

```
#include "selectionrule.hpp"

using namespace std;

void HighestLevelRule::gap(int h) {
    highest = h - 1;
    updateHighest();
}

void HighestLevelRule::updateHighest(void) {
    int s = highest;
    highest = -1;

    for (int i = s; i >= 0; --i) {
        if (hq[i].size() > 0) {
            highest = i;
            break;
        }
    }
}

bool HighestLevelRule::empty(void) {
    return highest < 0;
}

int HighestLevelRule::next(void) {
    if (empty()) throw EmptyQueueException();

    int u = hq[highest].front();
    hq[highest].pop();
    deactivate(u);

    updateHighest();

    return u;
}

void HighestLevelRule::add(int u, int height, int excess) {
    if (isActive(u)) return;
    if (height >= N) return;
    if (excess == 0) return;
```

```
    hq[height].push(u);
    if (height > highest) highest = height;
}

void FIFORule::gap(int h) {
}

int FIFORule::next(void) {
    int u;

    if (!q.empty()) {
        u = q.front();
        q.pop();
        deactivate(u);
    }
    else {
        throw EmptyQueueException();
    }

    return u;
}

void FIFORule::add(int u, int height, int excess) {
    if (isActive(u)) return;
    if (height >= N) return;
    if (excess == 0) return;

    activate(u);
    q.push(u);
}

bool FIFORule::empty(void) {
    return q.empty();
}
```

A.5 neighborhood.hpp

```
#pragma once

#include <iostream>
#include <vector>
#include <cmath>

class Coord {
public:
    int x;
    int y;
    int w;
    mutable double dt;

    Coord() : x(0), y(0), w(0), dt(0.0) {}
    Coord(int x, int y, int w) : x(x), y(y), w(w), dt(0.0) {}

    double angle() {
```

```
        return atan2(y, x);
    }
};

class CoordCompare {
public:
    bool operator()(const Coord& lhs, const Coord& rhs) {
        double p1 = copysign(1.0 - lhs.x/(fabs(lhs.x) + fabs(lhs.y)), lhs.y);
        double p2 = copysign(1.0 - rhs.x/(fabs(rhs.x) + fabs(rhs.y)), rhs.y);

        return p1 < p2;
    }
};

class Neighborhood {
private:
    std::set<Coord, CoordCompare> v;

public:
    void add(int x, int y, int w) {
        v.insert(Coord(x, y, w));
    }

    std::set<Coord, CoordCompare>::iterator begin() { return v.begin(); }
    std::set<Coord, CoordCompare>::iterator end() { return v.end(); }
    std::set<Coord, CoordCompare>::reverse_iterator rbegin() { return v.
    rbegin(); }
    std::set<Coord, CoordCompare>::reverse_iterator rend() { return v.rend();
     }
    size_t size() { return v.size(); }

    typedef std::set<Coord, CoordCompare>::iterator iterator;

    void setupAngles() {
        for (Neighborhood::iterator it = this->begin();
                it != this->end();
                ++it) {
            Coord prev;
            Coord next;

            if (it == this->begin()) {
                prev = *(this->rbegin());
            } else {
                prev = *(--it);
                it++;
            }

            if (++it == this->end()) {
                next = *(this->begin());
            } else {
                next = *it;
            }
            it--;
```

```
            it->dt = next.angle() - prev.angle();
            while (it->dt > 2.0 * M_PI)
                it->dt -= 2.0 * M_PI;
            while (it->dt < 0.0)
                it->dt += 2.0 * M_PI;
            it->dt /= 2.0;
        }
    }
};
```

A.6 image.hpp

```
#pragma once

#include <vector>
#include <set>
#include <opencv2/opencv.hpp>
#include "graph.hpp"
#include "selectionrule.hpp"
#include "neighborhood.hpp"
#include "sobel.hpp"

class Image {
private:
    cv::Mat *in, *out;
    int rows, cols;
    int pixels;
    int source, sink;

    std::vector<int> s_index;
    std::vector<int> t_index;

    std::vector<int> s_caps;
    std::vector<int> t_caps;

    FlowGraph network;

    /* These would preferrably be pointers, not references */
    Neighborhood& neigh;
    SelectionRule& rule;

    void createEdges();
    void createEdgesAnisotropic(int beta, cv::Mat_<Tensor>& tensors);
    void setupSourceSink(int alpha, int label, int p);

public:
    Image(cv::Mat *in, cv::Mat *out, SelectionRule& rule, Neighborhood& neigh
    ) :
        network(in->rows * in->cols + 2, rule),
        in(in),
        out(out),
        rule(rule),
        neigh(neigh),
```

```
        rows(in->rows),
        cols(in->cols),
        pixels(rows * cols),
        source(pixels),
        s_index(pixels),
        t_index(pixels),
        s_caps(pixels),
        t_caps(pixels),
        sink(pixels + 1) {}

    void restore(int alpha, int p);
    void restoreAnisotropicTV(int alpha, int beta, int p, cv::Mat_<Tensor>&
    tensors);
};
```

A.7 image.cpp

```
#include <iostream>
#include <fstream>
#include <algorithm>
#include <set>
#include <iterator>
#include <cmath>
#include <opencv2/opencv.hpp>
#include "image.hpp"
#include "neighborhood.hpp"
#include "sobel.hpp"

using namespace std;
using namespace cv;

int f(int u, int v, int p) {
    return p == 2 ? (u - v) * (u - v) : abs(u - v);
}

/* Fidelity energy term. */
int Ei(int label, int pix, int u, int p) {
    return (f(label+1, pix, p) - f(label, pix, p)) * (1 - u);
}

void Image::createEdges() {
    /*
     * Add sink edges first, so that the first push in discharge
     * will go towards the sink. The capacities are set up in
     * setupSourceSink.
     */
    for (int i = 0; i < pixels; ++i) {
        t_index[i] = network.addEdge(i, sink, 0);
    }

    /*
     * Create internal edges, which do not depend on the current
     * level.
     */
```

```
    for (int j = 0; j < rows; ++j) {
        for (int i = 0; i < cols; ++i) {
            Neighborhood::iterator it;
            for (it = neigh.begin(); it != neigh.end(); ++it) {

                int x = i + it->x;
                int y = j + it->y;

                if (x >= 0 && x < cols && y >= 0 && y < cols)
                    network.addDoubleEdge(
                            j*cols + i,
                            y*cols + x,
                            it->w
                            );
            }
        }
    }

    /*
     * Add edges from the source. Capacities are set up in
     * setupSourceSink.
     */
    for (int i = 0; i < pixels; ++i) {
        s_index[i] = network.addEdge(source, i, 0);
    }
}

void Image::createEdgesAnisotropic(int beta, Mat_<Tensor>& tensors) {
    /*
     * Add sink edges first, so that the first push in discharge
     * will go towards the sink. The capacities are set up in
     * setupSourceSink.
     */
    for (int i = 0; i < pixels; ++i) {
        t_index[i] = network.addEdge(i, sink, 0);
    }

    ofstream wfile;
    wfile.open("weights", ios::out | ios::trunc);

    /*
     * Create internal edges, which do not depend on the current
     * level.
     */
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            Neighborhood::iterator it;

            for (it = neigh.begin(); it != neigh.end(); ++it) {

                int x = j + it->x;
                int y = i + it->y;
```

```
                Mat ee = (Mat_<double>(2, 1) << it->x, it->y);
                Mat M  = Mat(tensors(i,j));

                double w = beta
                    * norm(ee) * norm(ee)
                    * determinant(M)
                    * it->dt
                    / pow(ee.dot(M * ee), 3.0 / 2.0);

                wfile << w << endl;
                if (x >= 0 && x < cols && y >= 0 && y < rows)
                    network.addDoubleEdge(
                            i*cols + j,
                            y*cols + x,
                            w
                            );
            }
        }
    }
    wfile.close();

    /*
     * Add edges from the source. Capacities are set up in
     * setupSourceSink.
     */
    for (int i = 0; i < pixels; ++i) {
        s_index[i] = network.addEdge(source, i, 0);
    }
}

/*
 * Change the capacities of the edges connecting the source
 * and the sink to the rest of the network, as these edges
 * are dependent on the current level.
 */
void Image::setupSourceSink(int alpha, int label, int p) {

    fill(s_caps.begin(), s_caps.end(), 0);
    fill(t_caps.begin(), t_caps.end(), 0);

    for (int j = 0; j < rows; ++j) {
        for (int i = 0; i < cols; ++i) {
            int e0 = Ei(label, in->at<uchar>(j, i), 0, p);
            int e1 = Ei(label, in->at<uchar>(j, i), 1, p);

            if (e0 < e1) {
                s_caps[j*cols + i] += e1 - e0;
            }
            else {
                t_caps[j*cols + i] += e0 - e1;
            }
        }
    }
```

```
    for (int i = 0; i < s_caps.size(); ++i) {
        network.changeCapacity(source, s_index[i], alpha * s_caps[i]);
    }

    for (int i = 0; i < t_caps.size(); ++i) {
        network.changeCapacity(i, t_index[i], alpha * t_caps[i]);
    }
}

void Image::restore(int alpha, int p) {

    createEdges();

    for (int label = 255; label >= 0; --label) {
        cout << "Label: " << label << endl;

        setupSourceSink(alpha, label, p);
        network.minCutPushRelabel(source, sink);

        /* Use the cut to update the output image. */
        for (int j = 0; j < rows; ++j) {
            for (int i = 0; i < cols; ++i) {
                if (!network.cut[j*cols + i])
                    out->at<uchar>(j, i) = label;
            }
        }
    }
}

void Image::restoreAnisotropicTV(int alpha, int beta, int p, Mat_<Tensor>&
    tensors) {

    createEdgesAnisotropic(beta, tensors);

    for (int label = 255; label >= 0; --label) {
        cout << "Label: " << label << endl;

        setupSourceSink(alpha, label, p);
        network.minCutPushRelabel(source, sink);

        /* Use the cut to update the output image. */
        for (int j = 0; j < rows; ++j) {
            for (int i = 0; i < cols; ++i) {
                if (!network.cut[j*cols + i])
                    out->at<uchar>(j, i) = label;
            }
        }
    }
}
```

A.8 main.cpp

```
#include <iostream>
```

```
#include <cstdlib>
#include <cmath>
#include <cstdio>
#include <unistd.h>
#include <opencv2/opencv.hpp>
#include "graph.hpp"
#include "selectionrule.hpp"
#include "neighborhood.hpp"
#include "image.hpp"

using namespace std;
using namespace cv;

int main(int argc, char *argv[])
{
    int p = 2;
    double beta = 10;
    char rarg = 'f';
    int neighbors = 4;
    int index;
    int c;

    /* Read command line parameters beta and p. */
    while ((c = getopt(argc, argv, "b:p:n:fh")) != -1) {
        switch (c)
        {
        case 'p':
            p = atoi(optarg);
            break;
        case 'b':
            beta = atof(optarg);
            break;
        case 'f':
            rarg = 'f';
            break;
        case 'h':
            rarg = 'h';
            break;
        case 'n':
            neighbors = atoi(optarg);
            break;
        case '?':
            if (optopt == 'p')
                fprintf(stderr, "Option␣-%c␣requires␣an␣argument.\n",
                        optopt);
            if (optopt == 'b')
                fprintf(stderr, "Option␣-%c␣requires␣an␣argument.\n",
                        optopt);
            if (optopt == 'n')
                fprintf(stderr, "Option␣-%c␣requires␣an␣argument.\n",
                        optopt);
            else if (isprint(optopt))
                fprintf(stderr, "Unknown␣option␣`-%c'.\n", optopt);
```

```
            else
                fprintf(stderr, "Unknown␣option␣character␣`\\x%x'.\n",
                        optopt);
            return 1;
        default:
            exit(1);
        }
    }

    cout << "Using␣p␣=␣" << p << endl;
    cout << "Using␣beta␣=␣" << beta << endl;

    if (neighbors != 4 && neighbors != 8) {
        cout << "Only␣4␣or␣8␣neighbors␣supported,␣using␣4." << endl;
        neighbors = 4;
    }

    /*
     * Non-option arguments are now in argv from index optind
     * to index argc-1
     */
    Mat image;
    image = imread(argv[optind], CV_LOAD_IMAGE_GRAYSCALE);

    if (!image.data) {
        cout << "Loading␣image␣failed" << endl;
        return -1;
    }

    int rows = image.rows;
    int cols = image.cols;
    int pixels = rows * cols;

#ifdef DINIC
    cout << "Using␣Dinic's␣maximum␣flow␣algorithm." << endl;
#else
    cout << "Using␣the␣Push-Relabel␣maximum␣flow␣algorithm." << endl;
#endif

    /*
     * Network only handles integer edges, so for floating
     * point beta parameters, we cheat a little bit.
     */
    int a;
    int b;
    a = 1;
    b = beta;
    if (beta < 10) {
        b = 100 * beta;
        a = 100;
    }

    /*
```

```
     * Specify one quarter of the neighbors of a pixel. The rest
     * are added symmetrically on the other sides.
     */
    Neighborhood neigh;
    if (neighbors >= 2) {
        neigh.add( 1, 0, b * 1.0);
        neigh.add( 0, 1, b * 1.0);
    }
    if (neighbors >= 4) {
        neigh.add( 1, 1, b * 1.0/sqrt(2.0));
        neigh.add(-1, 1, b * 1.0/sqrt(2.0));
    }

    Mat out = image.clone();

    HighestLevelRule hrule(pixels + 2);
    FIFORule frule(pixels + 2);

    SelectionRule* rule;
    if (rarg == 'h') {
        cout << "Using␣highest␣level␣selection␣rule." << endl;
        rule = &hrule;
    } else {
        cout << "Using␣FIFO␣selection␣rule." << endl;
        rule = &frule;
    }

    Image im(&image, &out, *rule, neigh);
    im.restore(a, p);

    imwrite(argv[optind + 1], out);

    return 0;
}
```