Graph Cuts Segmentation

by

Queenie Leung

Undergraduate Honours Thesis
Faculty of Science (Applied and Industrial Mathematics)
University of Ontario Institute of Technology

Supervisor(s): Dr. Mehran Ebrahimi

Copyright © 2016 by Queenie Leung

Abstract

Graph Cuts Segmentation

Queenie Leung

Undergraduate Honours Thesis

Faculty of Science (Applied and Industrial Mathematics)

University of Ontario Institute of Technology

2016

This paper presents a framework for the segmentation of 2D medical scans with both a user-interactive optimization approach and an algorithm of clustering. In this case, interactivity stems from the user marking their chosen pixels as "seeds" to act as hard constraints. The method then relates to the application of weighted directed graphs in order to embody soft constraints in an energy function incorporating boundary and regional penalties. The goal is to separate objects with a high inter-similarity from the image. In our exploration, an algorithm which utilizes the duality between a minimum cut and a maximum flow is chosen.

Contents

1 Introduction **1**

1.1 Mathematical Formulation 3

1.2 Max-Flow Min-Cut Theorem 9

1.3 Interactive Graph-Cut 14

1.4 K-Means Clustering Graph-Cut 27

2 Software Instructions **36**

2.1 Interactive Graph Cut 36

2.2 K-Means Clustering 42

3 Future Work **45**

Bibliography **48**

Chapter 1

Introduction

Segmentation is given its name based on its purpose to split an image into two or more segments: typically the "object(s)" and the "background(s)." Applications of segmentation are vital to medical imaging as a tool to both diagnose patients and research functionalities of organs. One example is the early detection of Alzheimer's disease through computing the volume of the hippocampus as presented in magnetic resonance images [12]. While digital mammograms are generally affective in detecting varying stages of breast cancer, image segmentation draws attention to less noticeable regions of risk resulting in an increase to early detection [2]. In both cases, the diseased organ of interest must be segmented from the rest of the scan for further analysis. Correct and early diagnosis is clearly a key factor to improving health care.

There exist many methods of image segmentation, two of which will be the focus of this paper. The first is the idea of interactive graph cuts [7]. In such a method, the algorithm begins by storing information provided by the user in the form of what are known as <u>hard constraints</u>. These seeds are actually two or more groups of pixels, some belonging to the background and the others belonging to the object. A typical method of obtaining this information is through brush strokes or specifying polygonal regions of interest. It is important to note that the background sets must be part of the background

segment while the object sets must be part of the object segment. Information obtained from hard constraints are additionally implemented into the soft constraint data relating to region penalties. Neighborhood similarity is obtained by the program itself, then the image is automatically segmented. The difference between the two constraints lies in their effect in the problem. While all hard constraints must be satisfied for the cut to be feasible, the soft constraints simply contribute towards an energy that is minimized when finding the global optimal.

An advantage of this interactive algorithm is that it is very compatible to an iterative approach. While other methods using automated cuts are also used, interactive segmentation corrects many of the complications that may arise. After the first run, another optimal segmentation may be recomputed by having the user select additional background or object seeds located in imperfect regions. The revised cut is calculated from the result of the previous iteration rather than from the very beginning. This feature is desirable because it not only leads to time and space efficiency, but can be carried out intuitively by a third party with little to no prior training.

On the other hand, segmentation can also be done without any interactive input from the user. In these cases, hard constraints typically do not exist and the image is only optimized over an energy function. To obtain vital clues in absence of such hard constraints, one method is the use of k-means clustering [1, 3]. K-means clustering aims to provide a "rough segmentation" of the image through an interactive process. Depending on how many segments are desired, a number of initial centroids are randomly chosen as the representative of each group. Using a normed distance from each pixel intensity to the centroids, the pixels are then partitioned into a group corresponding to the minimized distance. The algorithm updates the centroids as the average all pixels belonging to its cluster, and the process is repeated until no relabeling between iterations occurs. While the main purpose of hard constraints is not viable here, these clusters will still act in its place in terms of its role in contributing to finding the region penalties. Similarly, the

image is then automatically segmented as a global optimal of an energy function.

To gain an idea of both processes, the fundamental step is understanding the background information needed in splitting a 2D image into two segments. This base case is much easier to grasp in this sense as the graph is easy to visualize. The set up of the problem is provided in section 1.1. Section 1.2 will introduce the maximum flow algorithm as an important contributor to solving minimum cut due to their duality. Sections 1.3 and 1.4 cover our interactive graph cut method and k-means clustering method respectively. In both explorations, the alteration to the graph as well as samples of the segmentation administered on very simple impractical graphs meant to gain understanding will be displayed. This paper will then present various segmentation results on medical images in chapter 2. In order for the user to be able to experiment on their own images, chapter 3 presents software instructions along with the source codes corresponding to the theorem in sections 1.3 and 1.4. Finally, concluding statements and future work are provided in chapter 4.

1.1 Mathematical Formulation

To perform a graph cut of an image into the "object" and "background" regions, a graph G must be built. Suppose a gray scale image $\mathbf{u}$(x,y) of height m and width n is provided. Structurally, the image is viewed as a 3D discrete function where the first dimension x=[1,2,...,n] is the x-coordinate of the pixel, the second dimension y=[1,2,...,m] is the y-coordinate of the pixel, and the third dimension $\mathbf{u}$(x,y) takes on values depending on the intensity of the pixel in that (x,y) location. Each pixel acts as a vertex in the set V. To complete the set, two vertices $\{s,\ t\}$ known as terminal nodes are added into V. s corresponds to "object" and t corresponds to "background." For this reason, we often call this algorithm the "s-t cut." s is thought of as the node corresponding to the object group while t corresponds to the background group.

The second component of graphs involves introducing directed edges: a set Ω made up of two main partitions, t-links and n-links. $(s,\ p)$ and $(p,\ t)$ for all p$\in V\backslash\{s,t\}$ are known as the t-link edges. Evidently, they connect all pixels to the terminal nodes. $(p,\ q)$ and $(q,\ p)$ for all p,q$\in V\backslash\{s,\ t\}$ where q is in the neighborhood N_p of p are known as the n-link edges which connect all pixels to their neighbors. For this algorithm, a 4-neighborhood system is chosen. Ω is described as the union of all t-links and n-links. A graph can then be defined as $G = \{V, \Omega\}$. An example of a graph is depicted in figure 1.1 (a). This paper will use [10] to draw its graphs.

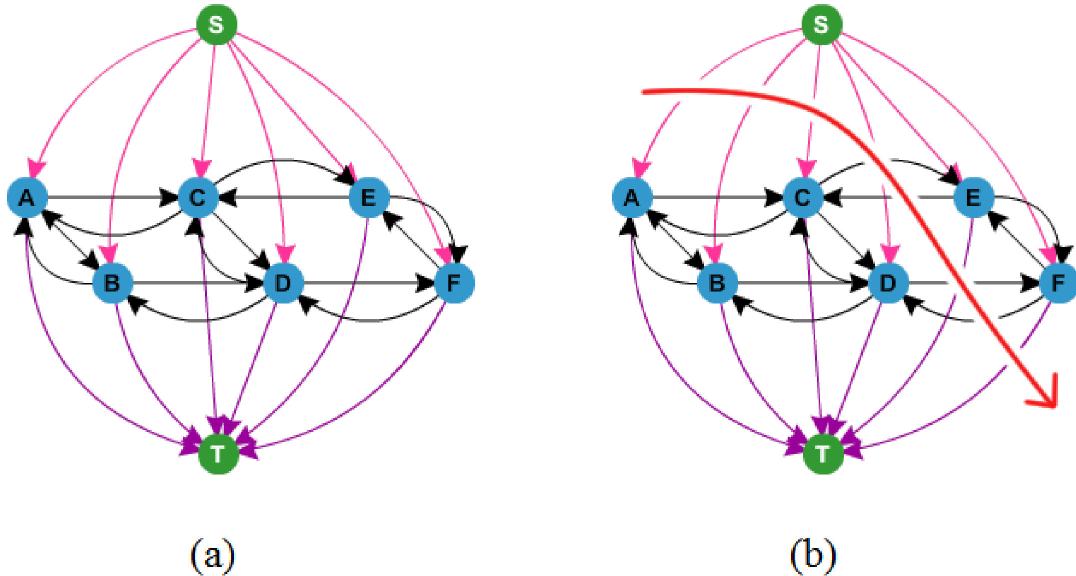


Figure 1.1: (a)The graph corresponding to an arbitrary *2*x*3* image. Vertices in green represent the terminal nodes while vertices in blue represent the pixels. Black edges are the n-links while the pink and purple edges are the t-links. (b) A random feasible cut of G. In this case, C = {(s,A), (s,B), (s,C), (s,D), (E,t), (F,t), (C,E), (E,C), (D,F), (F,D)} where the last 4 elements are n-links and the first 6 elements are t-links. The corresponding A is $\{A_A, A_B, A_C, A_D, A_E, A_F\}$ = {t,t,t,t,s,s}.

The goal of this algorithm is to assign a labeling $\mathbf{L} = \{s,\ t\} = \{L_1, L_2\}$ = {0,1} to each random variable of pixels in $V\backslash\{s,t\}$. The final result should be a binary array A

$= (A_1, A_2, ..., A_{|V|})$ where A_i $\forall$ i=1,2,...,$|V|$ is either s or t. Another way of looking at this problem is to say that we "cut" the graph into two parts. A graph cut is a subset of edges $C \subset \Omega$ such that the resulting graph $G(C) = \{V, \Omega \backslash C\}$ has separated terminals. However, there are rules which must be followed. Any solution cut must be feasible, as defined in the following theorem [7]:

Theorem 1.1. *Assume that the set Ψ denotes all feasible cuts C on graph G. Then Ψ is all possible C such that*

✓ *$\forall p \in V \backslash \{s,t\}$, p must be left with exactly one t-link (L_1,p) or (p,L_2) [9]*

✓ *$(p,q) \in C$ iff p and q are linked to different terminals*

✓ *A_p=s iff $(p,t) \in C$*

✓ *A_p=t iff $(s,p) \in C$*

The graph cut given in figure 1.1 (b) is an example of a feasible cut. As depicted, for any feasible graph cut $C \subset \Omega$, a unique corresponding segmentation $A(C)$ exists. Such an $A(C)$ can be defined as

$$A_p(C) = \begin{cases} \text{"}s\text{"} = 0 & if\{p,t\} \in C \\ \text{"}t\text{"} = 1 & if\{s,p\} \in C \end{cases} \tag{1.1}$$

Evidently, C would be made up of all edges that must be cut in G [1]. The issue now is the question of which feasible cut is chosen and why. To differentiate such an $m \times n$ image from another of the same size, it is necessary to somehow input the data from the pixel intensities. These intensities affect the problem in the form of edge capacities or edge weights; a non-negative function $w : \Omega \rightarrow \mathbb{R}$ which essentially attaches a price tag to each edge. If we think of the cut C as a shopping trip where all its parts must be purchased, then the total cost of cut C is

$$|C| = \sum_{e \in C} w(e) = \sum_{(a,b) \in \Omega} w(e = (a,b)) d_{e=(a,b)} \tag{1.2}$$

as long as $d_{e=(a,b)}$ is one when pixel a and b are in different groups and zero otherwise. This problem now grows from finding any cut C to finding a unique cut C such that $|C|$ is minimized, an optimization problem. Similar to how C and A share an equivalence relation, finding C such that $|C|$ is minimized is the same as finding A such that the energy function $E(A)$ is minimized. Specifically,

$$E(A) = \lambda \cdot R(A) + B(A) \tag{1.3}$$

is the sum of the region and boundary penalties. λ is a parameter that controls which term has a stronger influence on the solution.

$$R(A) = \sum_{p \in V \setminus \{s,t\}} R_p(A_p) \tag{1.4}$$

is the region penalty given a labeling A. For each pixel p, this term can assign two possible values: one such value is given if p is in the object (t-linked to s) and the other is given if p is in the background (t-linked to t). A higher $R_p(A_p)$ when $A_p = s$ than when $A_p = t$ indicates a larger "penalty" for placing pixel p in the object group than the background group, and vice versa.

$$B(A) = \sum_{\{p,q\} \in N} B_{p,q} \cdot \delta(A_p, A_q) \tag{1.5}$$

where

$$\delta(A_p, A_q) = \begin{cases} 1 & A_p \neq A_q \\ 0 & otherwise \end{cases} \tag{1.6}$$

is the boundary penalty; a value that is nonzero if A places two neighboring pixels (p,q) such that p∈object and q∈background, or p∈background and q∈object. In other

words, B(A) penalizes the grouping of neighboring pixels into two different partitions, and increases in value when p and q have very similar intensities. Obviously, two images with the same dimensions but different intensities will yield distinct energy functions.

Lemma 1.1. *The minimum cut $\hat{C}$ on G is feasible, so $\hat{C} \in \Psi$.*

The above lemma is a consequence of $\hat{C}$ being a minimum cut. $\hat{C}$ must sever at least one t-link at each pixel by definition of a graph cut. Opting to cleave both t-links means to spend money on unnecessary purchases, raising $|C|$. Hence for $\hat{C}$ to be a minimum cut, it must also sever at most one t-link at each pixel and the first property of a feasible cut is satisfied. The second condition is automatically satisfied because $\hat{C}$ must separate the terminals by definition.

By finding the A such that such an energy E is minimized, the optimal segmentation is obtained. Such information can be transferred over to the graph cut problem given that the previously mentioned function $w : \Omega \to \mathbb{R}$ is well defined. If e$\in\Omega$ is an n-link, then w(e) contributes to the boundary penalty, and if e$\in\Omega$ is a t-link, then w(e) contributes to the region penalty. The exact assignment of these weights are given in table 1.1.

e $\in \Omega$	w(e)	for
{p,q} n-link	$B_{\{p,q\}}$	q $\in N_p$
	0	q $\notin N_p$
{s,p} t-link	$\lambda \cdot R_p(t)$	p $\in V$
{p,t} t-link	$\lambda \cdot R_p(s)$	p $\in V$

Table 1.1: The weights of the edges Ω in G. Note that the t-link edge corresponding to object (s) is given the region boundary for background while the t-link edge corresponding to background (t) is given the region boundary for object. The reasoning behind this inverse relation is given in equations 1.7 and 1.8.

The weights are assigned so then edges with larger weights are less likely to be "purchased" into minimum cut $\hat{C}$. In fact, while the region and boundary terms act as penalties in $E(A)$, they take on the role of similarity values in w(e). The general-purpose derivation of the region penalties are given as the negative log-likelihoods

$$R_p(s) = -\ln\big(Pr(I_p|s)\big) \tag{1.7}$$

$$R_p(t) = -\ln\big(Pr(I_p|t)\big) \tag{1.8}$$

where the prior probability Pr($I_p|j$) $\in$ [0,1] $\forall$ j=1,2,...,|**L**| depends on the method of graph-cut segmentation: interactive or k-means clustering. While other approaches exist, these are the main two which will be covered in this paper. In both cases, there exists a positive correlation describing the relation between the similarity of pixel p to group j. As Pr($I_p|j$) $\rightarrow$ 1, pixel p is more likely to belong to group j, and as Pr($I_p|j$) $\rightarrow$ 0, pixel p is less likely to belong to group j. By taking the negative log-likelihood, $R_p(j) \rightarrow 0$ as pixel p is more likely to belong to group j and $R_p \rightarrow \infty$ as p is less likely to belong to group j. Therefore, R_p(j) acts as a penalty because a large value indicates that we do not want to place pixel p in group j. Since w(e) must consistently denote similarity, that is, a large value indicates that we do want to place pixel p in group j, then the assignment is swapped as indicated in table 1.1.

On the other hand, the boundary term is the continuous Gaussian probability distribution multiplied by the parameter inverse Euclidean distance

$$B_{\{p,q\}} = exp\Bigg(-\frac{(I_p - I_q)^2}{2\sigma^2}\Bigg)\cdot\frac{1}{||p-q||_2} \tag{1.9}$$

when I_p denotes the intensity of pixel p and I_q denotes the intensity of pixel q $\in N_p$. This probability distribution function has a characteristic "bell curve" shape with the properties that I_q is the position of the center of the peak and the variance σ^2 controls

the width of the bell. Unlike the region term $R_p(j)$ which can return any value between 0 and ∞, true to being a probability, the boundary term only produces outputs between 0 and 1. A number close to 1 is judged as meaning that the intensity of pixel p is similar to the intensity of pixel q. Therefore, $B_{\{p,q\}}$ can be directly assigned to w(e={p,q}) as a term which measures similarity. For our 4-neighborhood model, the Euclidean distance between pixel p and q denoted $||p - q||_2$ is always 1.

To summarize, the minimum-cut problem can be formulated as an optimization problem as follows:

$$
\begin{array}{lll}
\text{Minimize} & \sum_{(a,b)\in\Omega} w(e = (a,b))d_{e=(a,b)}, & \\
\text{subjected to} & d_{e=(a,b)} - A_a + A_b \geq 0, & \text{(a,b)} \in \Omega \\
 & d_{e=(a,b)} \geq 0, & \text{(a,b)} \in \Omega \\
 & A_s - A_t \geq 1 & \\
 & A_p \geq 0, & \text{p} \in V
\end{array}
$$

which is an LP(linear programming) system because the objective function is linear and optimized under linear inequalities.

Theorem 1.2. *The minimum cut and maximum flow problem always has a unique optimal solution [10].*

PROOF:

The cut $C = \{(s,p) \mid p \in V\}$ *is always feasible so a solution must exist.*

In addition, $\sum_{(a,b)\in\Omega} w(e = (a,b))d_{e=(a,b)} = \sum_{e\in C} w(e)$ *is bounded above by the case* $e \in C \;\forall\; e \in \Omega$ *so the solution must be unique.*

With our problem well-defined, we can now work on an efficient algorithm in the next section.

1.2 Max-Flow Min-Cut Theorem

Instead of interpreting $w : \Omega \to \mathbb{R}$ as a cost function to buy an edge for a cut C, we now think of w(e) as the capacity which represents the maximum amount of flow that can pass through the pipeline e $\in \Omega$. A flow can then be defined in place of a cut by

Definition 1.1. *A flow is a mapping $f : \Omega \to \mathbb{R}^+$, written as f((a,b)) or $f_{a,b}$ when (a,b) $\in \Omega$, which satisfies the two conditions:*

1. ***Capacity Constraint**: $f(e) \leq w(e) \; \forall \; e \in \Omega$*
2. ***Conservation Constraint**: $\sum_{\{p:(a,p)\in\Omega\}} f_{a,p} = \sum_{\{p:(p,a)\in\Omega\}} f_{p,a}$*

The corresponding value of flow from source (terminal) s to p is

$$|f| = \sum_{p \in V} f_{s,p} \quad \forall p \in V \tag{1.10}$$

The capacity constraint ensures that the amount of flow passing through an edge never surpasses its maximum capacity. The conservation constraint is equivalent to the idea that the amount of flow rushing into pixel p must equal the amount of flow rushing out of pixel p. The maximum flow problem is to find the route from the source(s) to the sink(t) such that |f| is maximized. The set of saturated edges appear after the maximum flow is computed and is defined as $S = \{$e | f(e) = w(e)$\}$.

Theorem 1.3. *The maximum flow from s to t is equal to the minimum cut. Additionally, $S = C$ where S is the set of saturated edges corresponding to the maximum flow and C is the minimum cut [4].*

The equality of the max-flow primal and the min-cut dual problems is a consequence of the strong duality theorem in linear programming. The solution to the dual provides a lower bound to the solution of the primal, yielding a duality gap $|\,|f| - |C|\,|$ that equals zero in the case where both optimization problems are convex.

By the max-flow min-cut theorem as presented in [15], an equivalent maximum-flow problem can be formulated as an optimization problem as follows:

$$
\begin{array}{lll}
\text{Maximize} & |f| = \sum_{p \in V} f_{s,p}, & \forall \text{ p} \in V \\
\text{subjected to} & f_{a,b} \leq w_{a,b}, & \text{(a,b)} \in \Omega \\
 & f_{a,b} \geq 0, & \text{(a,b)} \in \Omega \\
 & \sum_{\{a:(a,b)\in\Omega\}} f_{a,b} - \sum_{\{a:(b,a)\in\Omega\}} f_{b,a} \leq 0, & \text{b} \in V, \text{ b} \neq \text{s,t} \\
 & |f| + \sum_{\{a:(a,s)\in\Omega\}} f_{a,s} - \sum_{\{a:(s,a)\in\Omega\}} f_{s,a} \leq 0, & \\
 & -|f| + \sum_{\{a:(a,t)\in\Omega\}} f_{a,t} - \sum_{\{a:(t,a)\in\Omega\}} f_{t,a} \leq 0, &
\end{array}
$$

The first two expressions binding the problem ensure that the flow through an edge is neither negative nor surpasses the maximum capacity of that same edge. Referring back to figure 1.1 (a), each n-link would have two capacities flowing in opposite directions. The third constraint states that for any non-terminal vertex b, the total inflow to b is less than or equal to the total outflow from b. Finally, the last two conditions enforce the idea that the maximum s-t flow must equal the amount of flow coming out of s, as well as the amount of flow coming into t.

Definition 1.2. *A residual capacity is defined as* $w_k(e) = w_{k-1}(e) - |f_k|$ *where* $w_0(e) = w(e)$ *and* $|f_k|$ *is the total amount of flow sent through the graph at the kth iteration. In other words, it is the total amount of flow that can still be sent through pipeline e in the next iteration. We denote the set of all residual capacities by the kth iteration as* $\Omega_k = \{e \in \Omega | w_k(e) > 0\}$.

Definition 1.3. *The residual graph at the kth iteration is* $G_k = \{V, \Omega_k\}$.

The above definitions are crucial because the algorithm to solve the maximum flow problem (and hence the minimum cut segmentation) relies on a structure of performing numerous iterations until a certain condition is met. The most common of these is given in the theorem below.

Theorem 1.4. *Assume the input consists of the graph* G=G_0 *and the function of initial capacities* $w_0 : \Omega \rightarrow \mathbb{R}$*. The output should be the final residual graph* G_h *and maximum flow* $|f|$*, where* h *is the total number of iterations. Given this information, the* ***Ford-Fulkerson Algorithm*** *from [9] is as follows.*

1. *Set* k=*1,* $|f_k|$=*0, and* $|f|$=*0.*
2. *Find an s-t path* P *in the residual graph* G_{k-1} *s.t.* $w_{k-1}(e) > 0 \; \forall \; e \in P$.
   If none exists, then ***stop*** *and set* h=k, G_h=G_k.
3. *Compute* $|f_k| = \min_{e \in P}\big(w_k(e)\big)$*. Augment the path* P *by* $|f_k|$*.*
   Let k=k+*1 and* $|f|$= $|f|$ + $|f_k|$*. Return to step 2.*

Suppose an s-t path P is defined as a set of edges that lead from terminal s to terminal t. We let $\tilde{\text{P}}$ be the same path but from terminal t to terminal s. To augment this path by $|f_k|$ means that if e $\in$ P then let w_k(e) $= w_{k-1}(e) - |f_k|$, and if e $\in \tilde{\text{P}}$ then let w_k(e) $= w_{k-1}(e) + |f_k|$. If Ω_k is the set of all the updated edges, then the resulting residual graph would be $G_k = \{V,\Omega_k\}$. We explore the use of this algorithm on a simple graph in the next example.

Problem 1.1. *Suppose there are 3 projects* $\{A,B,C\}$ *and 3 pieces of equipment* $\{D,E,F\}$ *such that each project yields a revenue of* {*$100 ,$200 ,$150*} *respectively and each equipment costs* {*$200 ,$100 ,$50*} *to buy respectively. Project A requires the purchase of equipment D and E, project B requires the purchase of equipment E, and project C requires the purchase of equipment F. The problem is to find which projects and equipments should be selected to maximize profit.*

Considering each equipment and project as vertices, we let the s terminal represent the group of plans selected, and t represent the group of plans which are scrapped. Evidently, any t-link weights connected to s will have to be related to revenue and any t-link weights connected to t will have to be related to costs. Assuming that the equipment can be shared by several projects, the resulting graph in figure 1.2 (a) can be drawn.

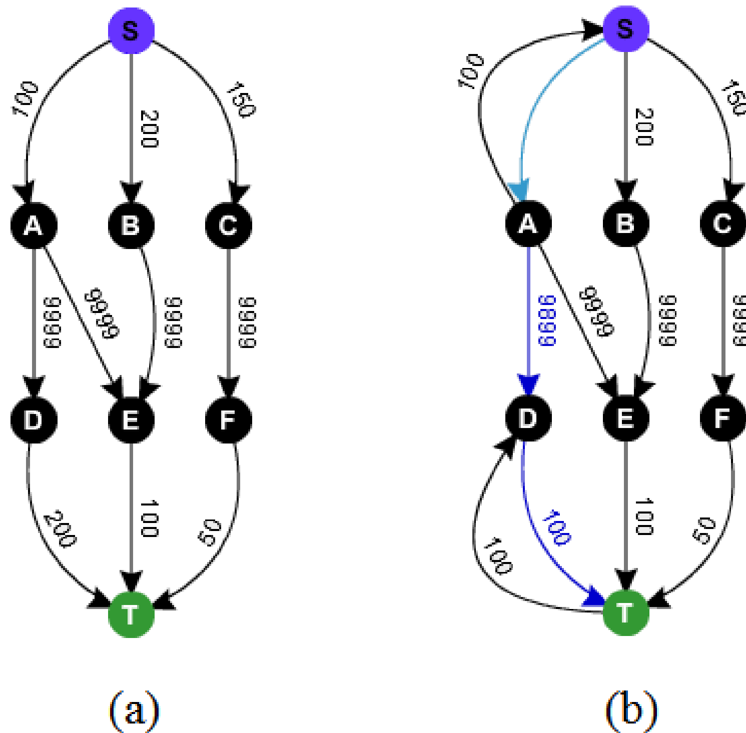


Figure 1.2: Our project selection problem. (a) The graph G = {V,Ω} along with the weights $w : \Omega \to \mathbb{R}$ to represent the system. (b) The residual graph G_1 after the first iteration of the Ford-Fulkerson algorithm.

Note that the n-links are set as an arbitrarily large number as indication that each project will require their respective equipment no matter what. It is not important to keep track of the residual capacities for such edges. For the sake of clarity, the reverse edges of these n-links will not be included in the residual graphs.

The solution is found through the Ford-Fulkerson algorithm. For the first iteration, the s-t path selected is $s \to A \to D \to t$*, where the maximum flow which can be sent through this path is 100 and hence* $|f_1|$*=100. Results yield the saturated edge* $\{s,A\}$ *and residual graph* G_1 *as shown in figure 1.2 (b). Because another s-t path* $s \to B \to E \to t$ *can be found, then this is not the last iteration and the algorithm loops.*

Repeating this process with the path $s \to B \to E \to t$*, we determine that* $|f_2|$*=100*

with the saturated edge {E,t} in G_2 as depicted in figure 1.3 (c). Finally, $s \rightarrow C \rightarrow F \rightarrow t$ is selected. This time, $|f_3|$=50 along with the residual graph G_3 and saturated edge {F,t} in figure 1.3 (d). At this point, it can be deducted that no more s-t paths exist from which flow can be sent from terminal s to terminal t and the code terminates.

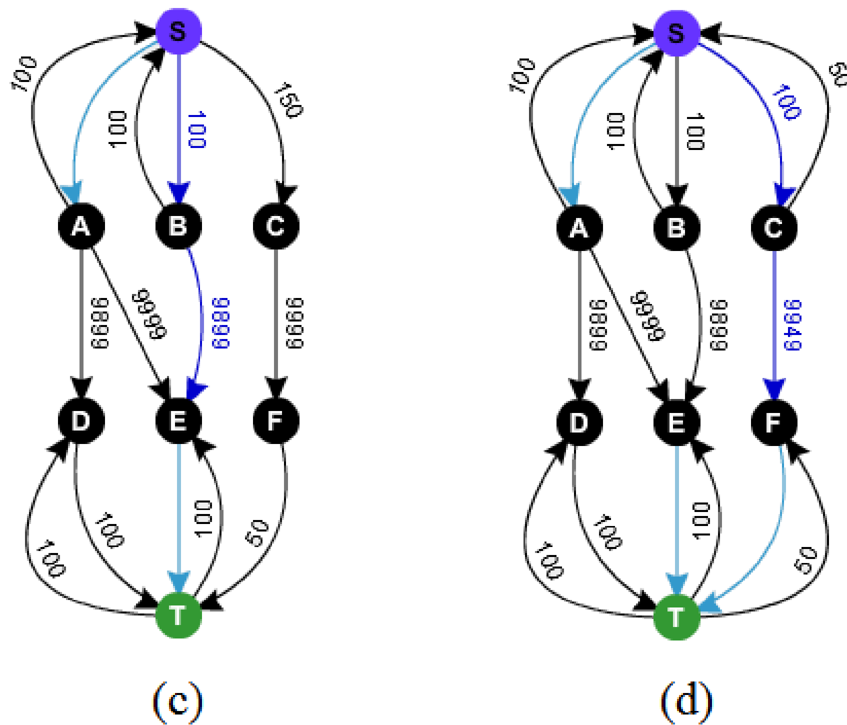


Figure 1.3: Our project selection problem continued. (c) The residual graph G_2 after the second iteration of the Ford-Fulkerson algorithm. (b) The residual graph G_3 after the third iteration of the Ford-Fulkerson algorithm.

The final output would be a maximum flow of $|f| = |f_1|+|f_2|+|f_3| =$ 250 with the set of saturated edges $S = \{$(s,A), (E,t), (F,t)$\}$. By the previously discussed duality in the max-flow min-cut theorem, the cost of the minimum cut would therefore be $|\hat{C}|$ = 250 where $\hat{C} = \{$(s,A), (E,t), (F,t), (A,E)$\}$. The addition n-link (A,E) is a consequence of $\hat{C}$ being a feasible cut. By figure 1.4, the optimal segmentation is $A = \{A_A, A_B, A_C, A_D, A_E, A_F\}$ = {t,s,s,t,s,s} = {1,0,0,1,0,0}. In other words, the projects which should be carried out

and the corresponding equipment to be bought in order to maximize profit are B and C along with E and F.

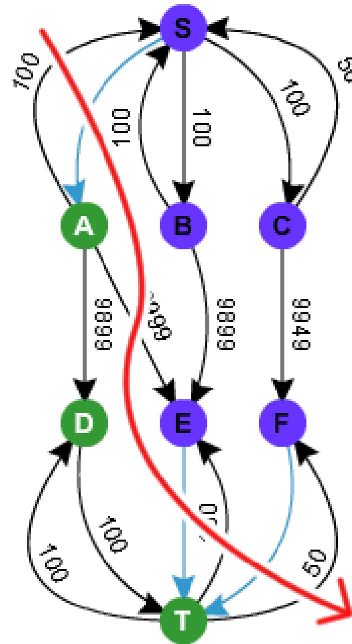


Figure 1.4: The corresponding minimum cut to the maximum flow.

The time complexity of the Ford-Fulkerson algorithm is $\mathcal{O}(|\Omega||f|)$ or in other words, the multiplication of the number of edges with the maximum flow. The reasoning behind the term $|\Omega|$ is simple. For each iteration of the algorithm, the maximum amount of edges that the program can search through in order to find an s-t path is $|\Omega|$. In the worst case scenario, the number of iterations will equal the total maximum flow $|f|$ if $|f_k|=1$ $\forall$ iterations. This concept is illustrated in figures 1.5 and 1.6 which represent the best and worst case scenarios respectively. Evidently, in the case of figure 1.6, there are $|f|=1000$ iterations. Hence, the time complexity is the absolute worst-case where for each iteration out of a total $|f|$ performed, every edge $|\Omega|$ must be searched.

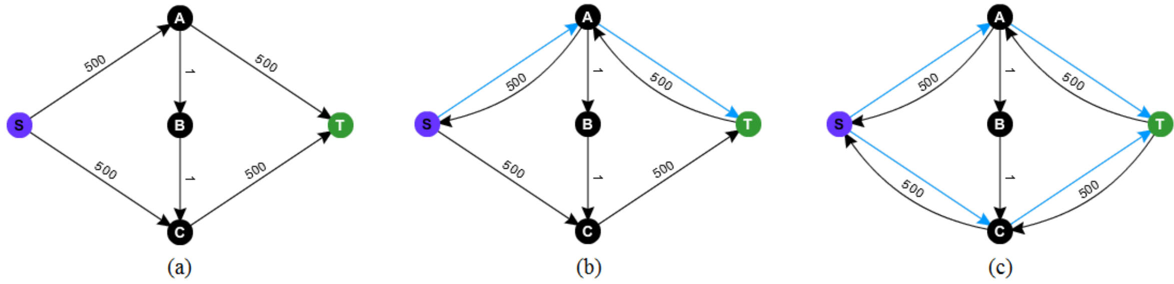


Figure 1.5: The best case scenario. (a) The given max-flow problem. (b) The first iteration chooses the path s $\rightarrow$ A $\rightarrow$ t so $|f_1|$=500. (c) The second iteration chooses the path s $\rightarrow$ C $\rightarrow$ t so $|f_2|$=500. In total, $|f|$=1000.

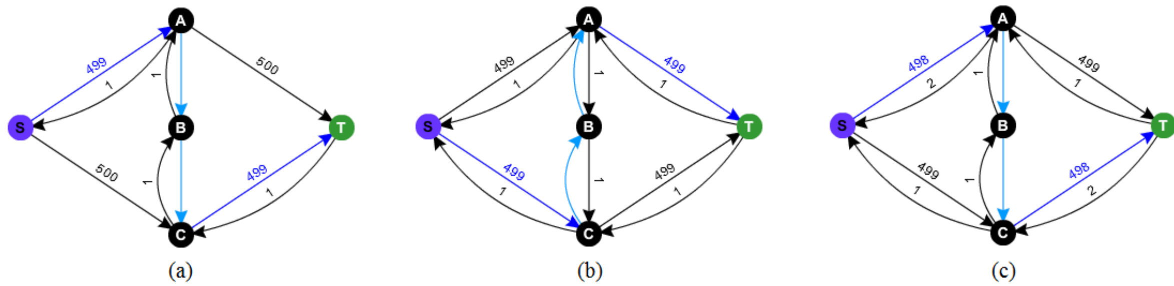


Figure 1.6: The worst case scenario. (a) The first iteration chooses the path s $\rightarrow$ A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ t so $|f_1|$=1. (b) The second iteration chooses the path s $\rightarrow$ C $\rightarrow$ B $\rightarrow$ A $\rightarrow$ t so $|f_2|$=1. (c) The third iteration chooses the path s $\rightarrow$ A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ t so $|f_3|$=1.

1.3 Interactive Graph-Cut

Recall that although it is true $w : \Omega \rightarrow \mathbb{R}$ is well-defined, the method of deriving the boundary term from equations 1.7 and 1.8 had still not yet been specified. The main reason is that the prior probability $\Pr(I_p|j) \in [0,1]$ $\forall$ j=1,2,...,$|\mathbf{L}|$ has many schemes for being obtained. The first of these schemes in which will be focused on in this paper is the interactive graph-cut method.

As mentioned in the introduction, the algorithm first gathers information provided by the user in the form of seeds: a group of pixels selected to be in the object or background.

Let $\theta \subset V \backslash \{s,t\}$ be the set of object seeds and $\beta \subset V \backslash \{s,t\}$ be the set of background seeds. The new assignment of weights are given in table 1.2, while figure 1.7 displays an example of the alterations made to figure 1.1 (a) to accommodate the new information.

e ∈ Ω	w(e)	for
{p,q} n-link	$B_{\{p,q\}}$	q ∈ N_p
	0	q ∉ N_p
{s,p} t-link	∞	p ∈ θ
	0	otherwise
{p,t} t-link	∞	p ∈ β
	0	otherwise

Table 1.2: The updated weight assignment of the edges Ω in G, when using the interactive graph-cut method. $B_{\{p,q\}}$ is as defined in equation 1.9.

It can be concluded then that this particular method uses an energy function which heavily relies on the boundary term B(A). Suppose an $m \times n$ image is inputted. The t-link weights are represented as a very sparse $mn \times 2$ matrix T consisting almost entirely of zeros. Suppose we count the pixels in the image in column-major order. An example of a *3×3* with pixels indexed in column-major order is given in problem 1.2. If entry T(i,1) = a then the ith pixel is t-linked to s with a weight of a. If entry T(j,2) = b then the jth pixel is t-linked to t with a weight of b. Evidently, the location of the object seeds θ and the background seeds β is known and the program can easily set the corresponding entries in T to an arbitrarily large number, and zero otherwise.

On the other hand, cataloging the n-link weights is considerably more difficult. Keeping in mind that a 4-neighborhood system is maintained, it is crucial to introduce a separate $|\hat{N}| \times 2$ matrix ε where $|\hat{N}|$ is the number of neighborhoods. Indexing the pixels in column-major order, each row of ε contains two elements, and represents an edge con-

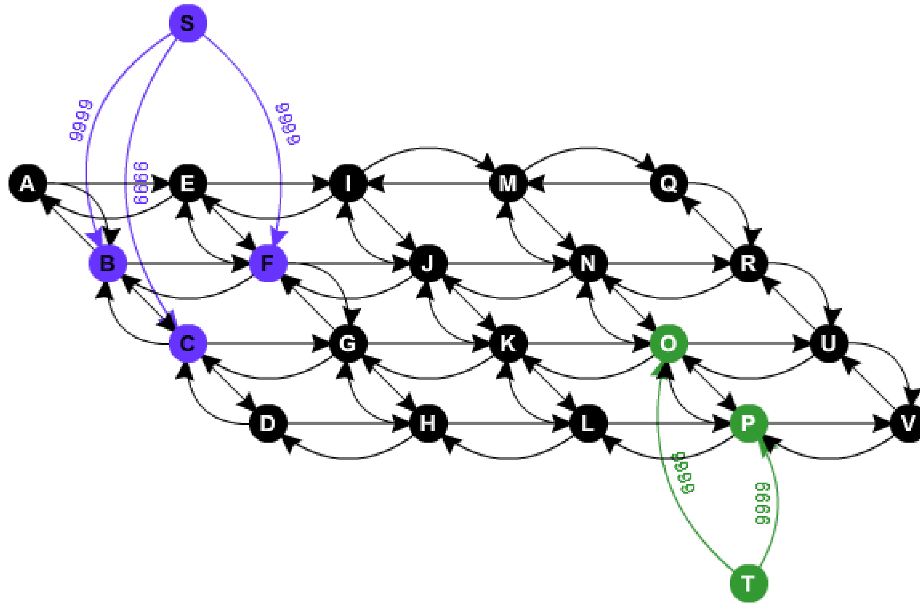


Figure 1.7: G for any 4×5 image when θ={B,F,C} and β={O,P}. The black edges have weights of $B_{\{p,q\}}$ while the purple and green edges are given an arbitrarily high weight of 9999.

necting those two corresponding pixels. Specifically, in the case of the program in listing 1.1 found in [11], the n-links are listed in the order: downward arrows, upwards arrows, rightward arrows, then leftward arrows. With this tool in hand, a sparse $mn \times mn$ matrix Ê can be defined to represent the n-link weights.

Listing 1.1: The program edges4connected.m forms a $mn \times mn$ matrix ε which can store information about the structure of the neighborhood

```
function E = edges4connected(height,width)

% EDGE4CONNECTED Creates edges where each node
%   is connected to its four adjacent neighbors on a
%   height x width grid.
```

```
%   E - a vector in which each row i represents an edge
%   E(i,1) --> E(i,2). The edges are listed is in the following
%   neighbor order: down,up,right,left, where nodes
%   indices are taken column-major.
%
%   (c) 2008 Michael Rubinstein, WDI R&D and IDC
%   $Revision$
%   $Date$
%

N = height*width; % total number of pixels
I = []; J = []; % I is the first column, J is the second column
% connect vertically (down, then up)

% GET DOWNWARDS THEN UPWARDS LINKS
is = [1:N]'; % index all the pixels
is([height:height:N])=[];
% every multiple of 'height'th pixel index is removed
% is = all pixels that have downwards neighbors

js = is+1;
% js = all pixels that have upwards neighbors

I = [I;is;js];
% I = pixels with downwards then upwards neighbors
% concatenate I row-wise

J = [J;js;is];
% J = pixels with upwards then downwards neighbors


% GET RIGHTWARDS THEN LEFTWARDS LINKS
```

```
is = [1:N-height]';
% is = all pixels that have rightward neighbors

js = is+height;
% js = all pixels that have leftwards neighbors

I = [I;is;js];
% I = pixels with downwards then upwards
%     then rightwards then leftwards neighbors

J = [J;js;is];
% J = pixels with upwards then downwards
%     then leftwards then rightwards neighbors

E = [I,J]; % concatenate column-wise
% each row is an n-link between pixels
% E = downward then upwards then rightwards then leftwards edges

end
```

We take a deeper look at the structure of this program by analyzing its output on a 3×3 image of arbitrary intensities.

Problem 1.2. *Consider the 3×3 image matrix A below counted in column-major order. This ordering means for instance that the entry located at $A(2,3)$ is known as the eighth pixel. The input is the height and width of the image, in this case, 3 and 3 while the output end goal is to create an $|N|\times 2$ matrix ε (=E in the code) corresponding to the system of neighborhoods.*

Let I be the first column of ε (=E in the code) and J be the second column of ε (=E in the code). In this case, $N = 9$ total pixels. "is = [1:N]';"' first takes a column array of pixels 1 through 9, after which "is([height:height:N])=[];" empties out pixels 3, 5, and 9. In other words, at this point, this is an array of all pixels who have downward

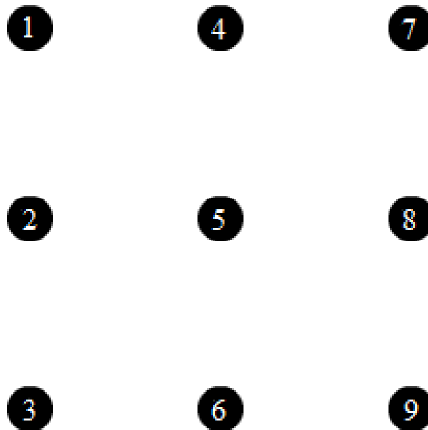


Figure 1.8: A 3×3 image assigned to matrix A. Pixels are indexed in column-major order.

neighbors. "js = is+1;'" shifts the pixel indexes by one and is an array of all pixels who have upward neighbors. Therefore, is = [1,2,4,5,6,8]' and js = [2,3,5,6,8,9]'. By concatenating the empty I in "I = [I;is;js];'," we receive a column array of all pixels with upward neighbors followed by all pixels with downward neighbors. In contrast, "J = [J;js;is];'" is a column array of all pixels with downward neighbors followed by all pixels with upward neighbors.

At this point, if I and J are concatenated to create an $|\hat{N}|\times 2$ *matrix, then the first six rows would represent the downward facing edges as depicted in green on figure 1.9 (a). As expected, the last six rows are the upward facing edges as depicted in blue on figure 1.9 (b).*

As can be predicted, the remainder of the code seeks to include 12 more rows for the rightward and leftward facing edges. For the 3×3 example, "is = [1:N-height]';" gives an output of is = [1,2,3,4,5,6] while "js = is+height;" gives an output of js = [4,5,6,7,8,9] so these arrays represent the indexes of pixels with rightward neighbors and leftward neighbors respectively. Now when we concatenate "I = [I;is;js];'," entry 13 through 18 are indexes of pixels with rightward neighbors while entry 19 through 24 are indexes of

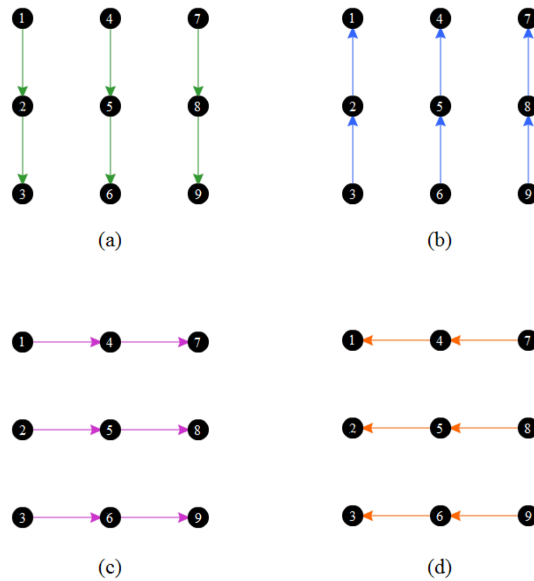


Figure 1.9: A 3×3 image assigned to matrix A. Pixels are indexed in column-major order.

pixels with leftward neighbors. For J, entry 13 to 18 are indexes of pixels with leftward neighbors then entry 19 to 24 are indexes of pixels with rightward neighbors.

Now when ε (=E in the code) is defined as the concatenation of I and J, we receive a 24×2 matrix where the number of neighborhoods is $\hat{N} = 24$. The next six rows are rightward facing edges as shown in purple in figure 1.9 (c) while the last six rows are leftward facing edges corresponding to the orange in figure 1.9 (d).

Now with the function **edges4connected.m** built, the code provided by [11] in listing 1.2 is much more easily manageable. In this case, the n-link weights $\hat{E}$ (i,j) (=A in the code) represents the cost of cutting pixel i from pixel j or in other words, placing pixel i and j into two different segments. Although not necessary, this code uses a variable V in order to define the n-link weights as the absolute value of the difference in intensities as in equation 1.11.

$$B_{\{p,q\}} = |I_p - I_q| \tag{1.11}$$

Similarly, T(i,l) is the cost of cutting the link between pixel i and label l or in other words, not placing pixel i in label l. A “black box” program titled **maxflow.m** which performs the Ford-Fulkerson algorithm is then called.

Listing 1.2: The program test2.m which performs a non-interactive segmentation on a waterfall. Additional comments were added to the source code for understanding.

```
% TEST2 Demonstrates gradient-based image
%   min-cut partitioning.
%
%   (c) 2008 Michael Rubinstein, WDI R&D and IDC
%   $Revision: 140 $
%   $Date: 2008-09-15 15:35:01 -0700 (Mon, 15 Sep 2008) $
%

im = imread('waterfall.bmp'); % Get 3D matrix of intensities
% Three dimensions as the third dimension has 3 elements (Red, Green, Blue)

m = double(rgb2gray(im));
% convert to a 2D grayscale matrix so intensities are double
[height,width] = size(m); % get height and width of picture

disp('building graph');
N = height*width; % total number of pixels

% construct graph
E = edges4connected(height,width); % Contains n-link system
%V = exp((-(m(E(:,1))-m(E(:,2))).^2))+eps;
V = abs(m(E(:,1))-m(E(:,2)))+eps; % difference between intensities
%  of first and second column, plus eps
   % eps returns the distance from 1.0 to the next largest
```

```
    %    double-precision number, that is eps = 2^(-52)
A = sparse(E(:,1),E(:,2),V,N,N,4*N);
%S = sparse(i,j,v) generates a sparse matrix S from the triplets i, j, and
%    v such that S(i(k),j(k)) = v(k).
%S = sparse(i,j,v,m,n) specifies the size of S as m-by-n.
%S = sparse(i,j,v,m,n,nz) allocates space for nz nonzero elements.
% Use this syntax to allocate extra space for nonzero values to
%      be filled in after construction.

% terminal weights
% connect source to leftmost column.
% connect rightmost column to target.
T = sparse([1:height;N-height+1:N]',[ones(height,1);ones(height,1)*2],...
ones(2*height,1)*9e9); % 9*(e^9) so infinity
% What sparse does is only store the location of the '1' entires since most
%        of the entires are '0'
% The leftmost pixels are automatically background (entry 1)
% First, concatenate two column vectors, vector of one to height and vector
%        of N-height+1 to N, transpose
% Second, concatenate two column vectors, vector of ones size
%        (height) and vector of ones size (2xheight)
% Third, output is a vector of one s size (2xheight) times 9e9

disp('calculating maximum flow');

[flow,labels] = maxflow(A,T);
labels = reshape(labels,[height width]);

imagesc(labels); title('labels');
```

Which yield the results of segmentation in figure 1.10.

Note that unlike the weights in table 1.2, the t-link weights for the object s which are set to infinity are those corresponding to pixels on the very left edge of the image.

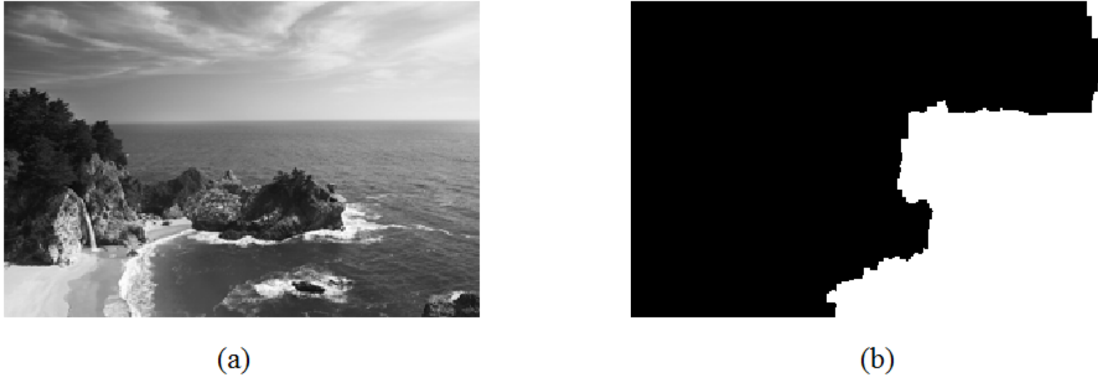


Figure 1.10: The graph cut provided in the source code. (a) The original image. (b) The resulting segmentation B = $[B_1, B_2, ..., B_N]$ where $B_k \forall k = 1, 2, ..., N$ are either 1 ("s object") or 2 ("t background") provided that N is the total number of pixels.

Meanwhile, the t-link weights for the background t which are set to infinity are those corresponding to the pixels on the very right edge of the image. A portrayal of such a graph is given in figure 1.11.

For a more efficient cut, the code must be modified such that object and background seeds are accepted. Depending on the location of these seeds, the infinity t-link weights are redistributed according to table 1.2. In layman's terms, instead of a graph as shown in figure 1.11, we want a graph alike to figure 1.7. An alteration to the source code such that the object seeds are grouped into a square located in the middle of a circle and the background seeds are in a square located outside the circle is produced in listing 1.3.

Listing 1.3: An altered code which performs an interactive segmentation on a circle of gradient. Additional comments were added to the source code for understanding.

```
% Uses maxflowmex to segment a an image of a circle
% Alteration of the known program which,
%      instead of connecting infinitely weighed T-links
```

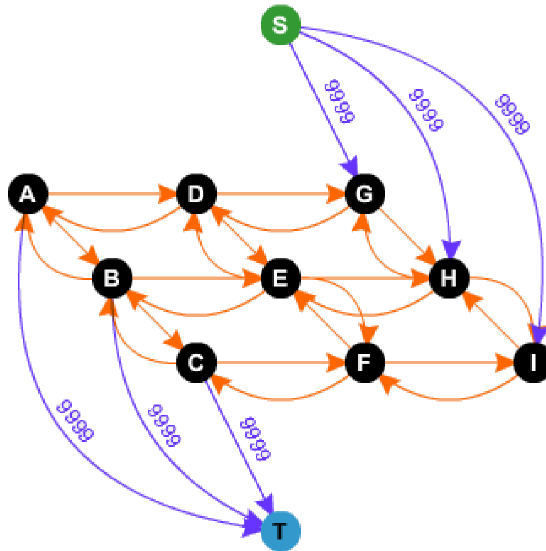


Figure 1.11: The graph built according to table 1.2. 9999 represents an arbitrarily large number.

```
% to the leftmost and rightmost pixels, connects to pixels inside the
% polygons chosen by the user.

% Create Circle
I=zeros(256,256); centre=128; radi=50; % 256x256 image of zeros (black)
for x=1:256
    for y=1:256
      I(x,y)= ((x-centre)^2+(y-centre)^2<radi^2);
      % If this statement is true, set to one (white)
    end;
end;
I=I*256;
% Create Chosen Polygons
for x=1:256
    for y=1:256
       O(x,y)=  (x<(128+5)).*(x>(128-5)).*(y<(128+5)).*(y>(128-5));
            % Object polygon inside circle
```

```
       B(x,y)= (x<250).*(x>240).*(y<250).*(y>240);
            % Background polygon outside circle
    end;
end;
%O=O*256;
%B=B*256;
figure; imshow(O,[]);
figure; imshow(B,[]);

% Graph Cut
disp('building graph');
[height,width] = size(I);
N = height*width; % total number of pixels
NO=0; NB=0; % represents number of pixels
%          in the object and background polygons
for x=1:N
    if O(x)==1
        NO=NO+1;
    end
    if B(x)==1
        NB=NB+1;
    end
end

E = edges4connected(height,width);
% V = exp((-(m(E(:,1))-m(E(:,2)))^2))+eps; % the one on the paper
V = abs(I(E(:,1))-I(E(:,2)))+0.0001;
A = sparse(E(:,1),E(:,2),V,N,N,4*N);

% Alteration to T-links
% get matrix of pixel INDEXES in the object and background polygons
PO=ones(NO,1); PB=ones(NB,1);
```

```
mO=1; mB=1;
for x=1:N % counts in column-major order
    if O(x)==1
        PO(mO)=x;
        mO=mO+1;
    end
    if B(x)==1
        PB(mB)=x;
        mB=mB+1;
    end
end;



T = sparse([PO;PB],[ones(NO,1);ones(NB,1)*2],...
ones(NO+NB,1)*9^4,N,2); % 9*(e^9) so infinity
% corresponding to notes:
% C should be a column vector

%T = sparse([1:height;N-height+1:N]',[ones(height,1);
%   ones(height,1)*2],ones(2*height,1)*9e9); % 9*(e^9) so infinity

disp('calculating maximum flow');

[flow,labels] = maxflow(A,T);
labels = reshape(labels,[height width]);

imagesc(labels); title('labels');
```

The results of listing 1.3 would then be

To summarize, Ê (a,b) is the penalty of placing pixel a and pixel b in two different groups and would be equal to $B_{\{a,b\}}$=$B_{\{b,a\}}$. T(a,1) is the penalty of cutting the t-link between the a-th pixel (as counted in column-major order) from terminal s while T(a,2)

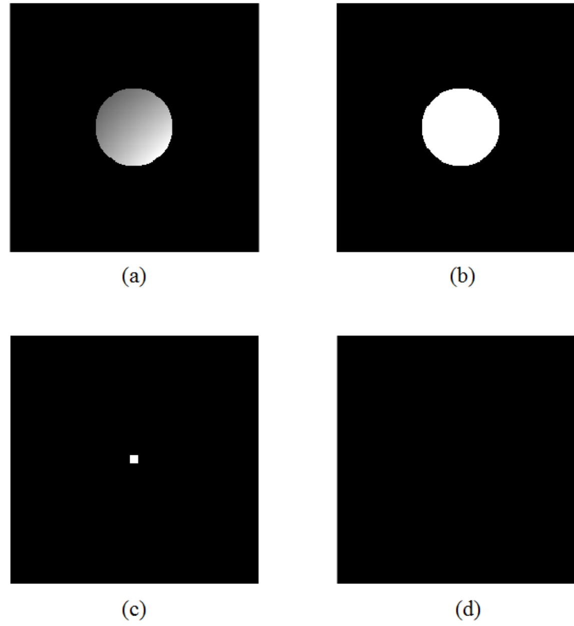


Figure 1.12: The modified graph cut. (a) The original image. (b) The resulting segmentation. (c) The polygon made up of all the object seeds. (d) The polygon made up of all the background seeds.

is the penalty of cutting the t-link between the a-th pixel from the terminal t. With the matrices of n-links and t-links now well-defined, we end off by testing the validity of our black box on a very simple graph.

Problem 1.3. *We test out our black box program through running it over a known graph cut. A graph of a 2×3 image is provided in figure 1.13 (a). Its corresponding optimal segmentation in figure 1.13 (b) is found through the Ford-Fulkerson algorithm.*

In this case, we correctly define a sparse 6×2 matrix T such that $T(1,1)=\infty$ *and* $T(6,2)=\infty$. *As for the n-link weights, they are stored in the sparse 6×6 matrix T2 such that:*

1. $\hat{E}(1,2) = \hat{E}(2,1) = 9$.

2. $\hat{E}(1,3) = \hat{E}(3,1) = 14$.

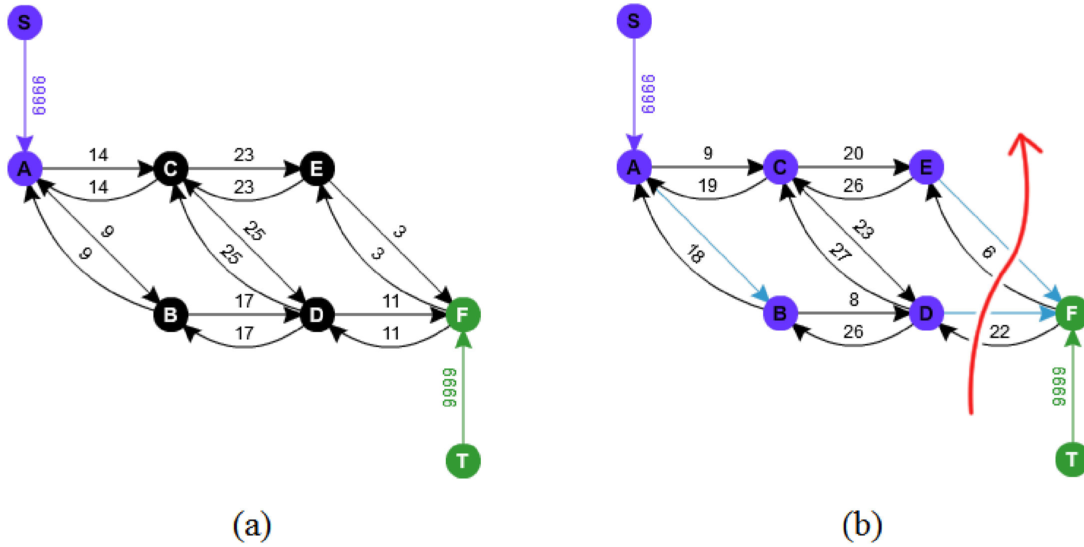


Figure 1.13: A posed graph cut problem for the interactive method. (a) G for a 2×3 image when θ={A}={1} and β={F}={6} when indexing in column-major order. (b) The final minimum cut solution, found using three iterations through the paths $s \rightarrow A \rightarrow B \rightarrow D \rightarrow F \rightarrow t$, then $s \rightarrow A \rightarrow C \rightarrow E \rightarrow F \rightarrow t$, and finally $s \rightarrow A \rightarrow C \rightarrow D \rightarrow F \rightarrow t$. The cost of the minimum cut is $|f|$=$|\hat{\text{C}}|$=14.

3. $\hat{E}(2,4) = \hat{E}(4,2) = 17$.
4. $\hat{E}(3,4) = \hat{E}(4,3) = 25$.
5. $\hat{E}(4,6) = \hat{E}(6,4) = 11$.
6. $\hat{E}(3,5) = \hat{E}(5,3) = 23$.
7. $\hat{E}(5,6) = \hat{E}(6,5) = 3$.

Using these weights, the black box program gives the final answer as $|\hat{C}|$=*14 and* A={$A_A, A_B, A_C, A_D, A_E, A_F$}={*0,0,0,0,0,1*}. *Since our written answer and the black box are equivalent, then the interactive graph cut black box works.*

When working with such an algorithm, it is important to note that an image can only

be segmented into simply connected regions. In other words, two segments which do not touch cannot both be considered the object or both be considered the background. The reason for this setback stems from the structure of the t-links. Fortunately, the coded t-links are relatively easy to alter in comparison to the n-links. In the next section, this paper will explore a technique in gathering region term data *R(A)* as well as its implementation in graph-cuts.

Software instructions to run the interactive graph-cut are provided in section 2.1.

1.4 K-Means Clustering Graph-Cut

In order to be able to apply this technique to graph-cuts, one must first understand the idea of K-means clustering itself. Suppose there are several items. K-means clustering is an algorithm which separates those items into K groups based on their attributes. Evidently, its name is born from the idea of sorting the data into "clusters." In our case, the items are all the pixels and the attribute is that pixel's corresponding intensity. Analogous to how every crowd has a leader, each group will have a representative called the centroid. The centroid will be taken as the average intensity of all its members. The k-means algorithm outlines four main steps [14].

Definition 1.4. *Given the goal of sorting a set* $N=\{N_1, N_2, ...N_n\}$ *of n items into k number of clusters based on a feature m, the k-means algorithm can be implemented.*

1. *Choose k random centroids. The set of centroids become* $\{c_1, c_2, ..., c_k\}$.
2. *Calculate the "distances" between all items and the centroids. These distances should depend on* $\{m_1, m_2, ..., m_p\}$.
3. *Group all items to its corresponding centroid with the shortest distance.*
4. *Calculate the new centroid* $c_i \forall i=1,2,...k$ *as the average c value of all members of its group.*

5. *Group all items to its corresponding centroid with the shortest distance.*

6. *If no items transfer to a new group in two consecutive iterations, then* ***stop****. Otherwise, go back to step 4.*

The definition of "distance" depends on the feature m. If m is a vector of p elements, in other words the sorting is based off of p features, then the distance between two items x and y is defined using the discrete p norm:

$$||x - y||_p = \left(\sum_{i=1}^{p} |m_{x,i} - m_{y,i}|^p \right)^{1/p} \tag{1.12}$$

Where $m_{a,b}$ is the b-th feature of item a. For the purpose of image segmentation, because m is only taken as {intensity, position}, then p=2 and the distance simplifies to the Euclidean norm. We test this algorithm on a simple example.

Problem 1.4. *Suppose we have 4 balls, each with features in weight and size as prescribed below. The goal is to group all items into $K = 2$ clusters based on these two attributes.*

Ball	Weight (g)	Size (dm^2)
A	2	6
B	1	5
C	3	3
D	4	1

Table 1.3: The four items we want to cluster are these balls with the following traits.

Because each ball has two features, then p=2 and the distance used will be defined as the Euclidean norm. To cluster these items into two groups, the k-means algorithm is used.

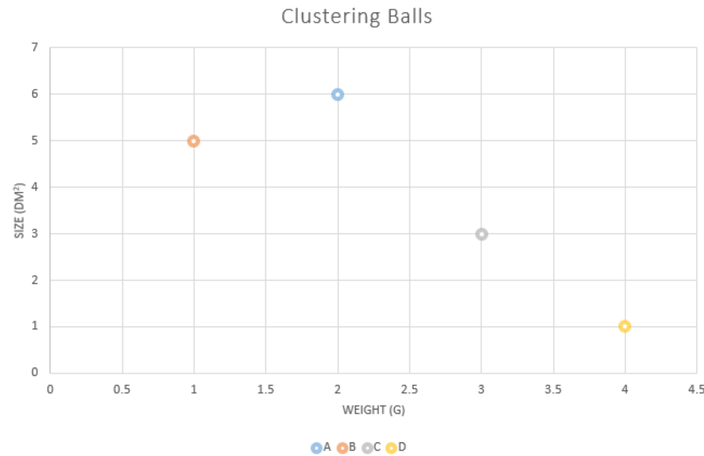


Figure 1.14: The location of the balls in the space of their attributes.

1. *Choose k=2 random centroids. Let* $A = c_1 = (2{,}6)$ *and* $B = c_2 = (1{,}5)$ *be the first centroids.*

2. *Calculating Euclidean distance, then* $D_A = \{0{,}1.41\}$, $D_B = \{1.41{,}0\}$, $D_C = \{3.16{,}2.83\}$, *and* $D_D = \{5.39{,}5\}$, *where* $D_j = \{d_1{,}d_2\}$ *is the distance from ball j to the centroids.*

3. *Obviously, balls A and B would remain in cluster 1 and 2 respectively. Balls C and D are assigned cluster 2 because they both have a shorter Euclidean distance to* c_2.

4. c_1 *remains as (2,6). The updated* c_2 *becomes* $c_2 = \left(\frac{1+3+4}{3}, \frac{5+3+1}{3}\right) = (\frac{8}{3}, 3)$.

5. *Calculating Euclidean distance, then* $D_A = \{0{,}3.07\}$, $D_B = \{1.41{,}2.60\}$, $D_C = \{3.16{,}0.33\}$, *and* $D_D = \{5.39{,}2.40\}$. *Ball B would transfer to cluster 1 while all other balls remain stationary.*

6. *Because ball B had to be moved to a different cluster, then we return to step 4.*

7. *The updated* c_1 *becomes* $c_1 = \left(\frac{2+1}{2}, \frac{6+5}{2}\right) = (\frac{3}{2}, \frac{11}{2})$. *The updated* c_2 *becomes* $c_2 = \left(\frac{3+4}{2}, \frac{3+1}{2}\right) = (\frac{7}{2}, 2)$.

8. *Calculating Euclidean distance, then* $D_A = \{2.55{,}4.27\}$, $D_B = \{0.71{,}2.69\}$, $D_C =$

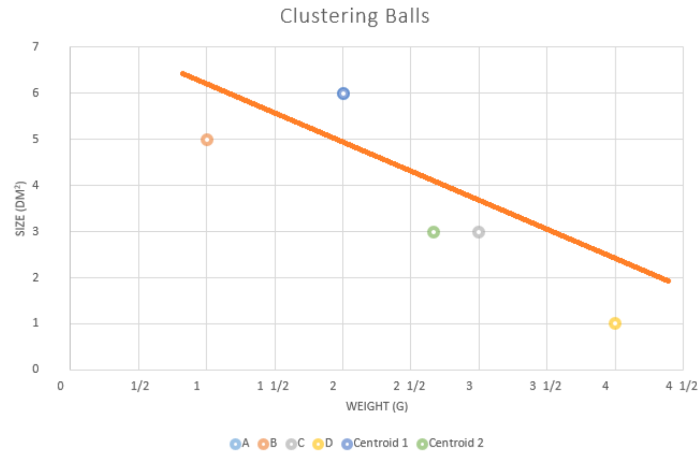


Figure 1.15: The centroids and clusters as of iteration 1. The orange line separates the clusters.

{2.92,1.12}, and D_D = {5.15,1.12}. None of the balls would transfer to a new group.

9. *Because no items transfer to a new group in two consecutive iterations, then we* ***stop****.*

The final result is {A,B,C,D} = {1,1,0,0}. As can be seen in figure 1.16, the centroids are of equal distance to all items in their cluster.

In MATLAB, the steps to performing this algorithm is wrapped up in a built-in function **kmeans.m**. Given an input of an $n \times 1$ column vector of items and the desired k number of clusters, this function outputs an $n \times 1$ column vector of indices 1,2,...,k-1, or k corresponding to that item's group and a $k \times 1$ array to store the centroids. In relation to graph-cuts, these clusters will act as data in order to derive an equation for $R(A_p)$ from equation 1.7 and 1.8 by using the Gaussian distribution. The result is equation 1.13 where n is the total number of items in group Q.

$$R(A_p) = -\ln\left(Pr(I_p|s)\right) = -\ln\left(\exp^{-\frac{||I_p - I_Q||^2}{2\sigma_Q^2}}\right) = \frac{||I_p - I_Q||^2}{2\sigma_Q^2} \qquad (1.13)$$

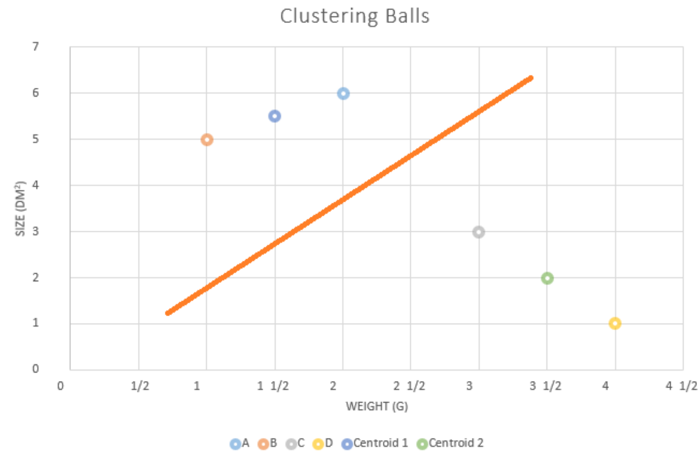


Figure 1.16: The centroids and clusters as of iteration 2. The orange line separates the clusters.

where σ_Q is the variance such that

$$\sigma_Q^2 = \frac{\sum_{p \in Q} ||I_p - I_Q||^2}{n - 1} \tag{1.14}$$

and I_Q is the centroid of group Q, given that A_p places pixel p in the group Q. Keep in mind that the distance between p and Q is defined according to equation 1.12. Outlined in listing 1.4 is the graph-cut program applied to a very simple matrix [1,5/6; 4/6,3/6; 2/6,1/6]. All t-link weights are stored in the matrix Dc while all the n-link weights $B_{\{p,q\}}$ in Sc are simply defined as 1 if p $\neq$ q and 0 otherwise.

Listing 1.4: The program gc_example_BW which performs a k-means clustering graph cut on a simple $3{\times}2$ image.

```
function gc_example_BW()
% Grayscale version of gc_example

close all
```

```
% read an image in grayscale
%im = im2double(rgb2gray(imread('original_ball.jpg')));
im=[1,5/6; 4/6,3/6; 2/6,1/6]; % 3x2 intensity matrix
% means [1.00 , 0.83]
%       [0.67 , 0.50]
%       {0.33 , 0.17]

sz = size(im); % becomes an array [ny, nx]

% try to segment the image into k different regions
k = 2;

% color space distance
distance = 'sqEuclidean'; % *

% cluster the image colors into k regions
data = ToVector(im); % CALLS FUNCTION AT THE BOTTOM
% converts im from (ny)x(nx)x(nc) to (ny*nx)x(nc)
[idx, c] = kmeans(data, k, 'distance', distance,'maxiter',200);
% clusters data into k classes,
% idx returns 6 x 1 array corresponding to pixels closest to centroids
%     c = [0.3333, 0.8333] where intensities 1, 0.67, 0.83 are closer to
%     0.8333 and intensities 0.33, 0.50, 0.17 are closer to 0.3333
%     idx = [2; 2; 1; 2; 1; 1]
%     NOTE: MAY SWITCH LABELS 1 AND 2 WITH EVERY RUN OF CODE

% calculate the data cost per cluster center
Dc = zeros([sz(1:2), k],'single');
% creates (ny)x(nx)x(k) 3D matrix of zeros
for ci=1:k % for each k class
    % use covariance matrix per cluster
    a = data(idx==ci,:); % collects pixels in that group
```

```
   b = cov(a); % calculates variance
   icv = inv(b); % takes inverse of variance 1/b
% a is data(idx==1,:) = [0.33; 0.50; 0.17] when ci = 1 and c(1) = 0.333
%      data(idx==2,:) = [1.000; 0.67; 0.83] when ci = 2 and c(2) = 0.833
% b is 0.0278 = [(0.33-0.33)^2 + (0.50 - 0.33)^2 + (0.167 - 0.33)^2] /3-1
%               when ci = 1 and c(1) = 0.3333
%   is 0.0278 = Var(a) = [(R1-av)^2 + (R2-av)^2 + (R3-av)^2] / N-1
%             = [(1-0.83)^2 + (0.67 - 0.83)^2 + (0.83 - 0.83)^2] / 3-1
%               when ci = 2 and c(2) = 0.8333
% icv is 36 = 1/b when ci = 1 and c(1) = 0.333
%        36 = 1/b when ci = 2 and c(2) = 0.833
% SO IVC IS 1 / (variance of pixels in one segment)

   d = repmat(c(ci,:), [size(data,1), 1]);
% d is [0.33; 0.33; 0.33; 0.33; 0.33; 0.33] when ci = 1 and c(1)=0.33
%      [0.83; 0.83; 0.83; 0.83; 0.83; 0.83] when ci = 2 and c(2)=0.83
%   so repeats c(ci) (ny)x(ny) times
   dif = data - d;
% dif = [0.67; 0.33; 0; 0.5; 0.17; -0.17] when ci = 1 and c(1) = 0.333
%       = [1-0.33; 0.67-0.33; 0.33-0.33; 0.83-0.33; 0.50-0.33; 0.17-0.33]

 %     = [0.17; -0.17; -0.5; 0; -0.33; -0.67] when ci = 2 and c(2) = 0.833
%       = [1-0.83; 0.67-0.83; 0.33-0.83; 0.83-0.83; 0.50-0.83; 0.17-0.83]
% SO DIF IS DIFFERENCE BETWEEN ALL
% PIXEL INTENSITIES AND THE CENTROID ci

   f = (dif*icv).*dif./2; % icv is scalar
   % g = sum(f,2); % not necessary since it sums over all channels (dim=2)
   Dc(:,:,ci) = reshape(f,sz(1:2));
   % f is [8; 2; 0; 4.5; 0.5; 0.5] when ci = 1 and c(1) = 0.333
   %      = (distance between all pixels...
       %        to centroid of ci).^2 / 2*(variance of ci)
```

```
    %   is [0.5;0.5;4.5;0;2.0;8.0] when ci = 2 and c(2) = 0.833
    % Dc reshapes into original image shape
    %    is [8,4.5; 2,0.5; 0,0.5] when ci = 1 and c(1) = 0.333
    %    is [0.5,0; 0.5,2; 4.5, 8] for ci = 2 and c(2) = 0.8333

end

% cut the graph

% smoothness term:
% constant part
Sc = ones(k) - eye(k);
% spatialy varying part
% [Hc Vc] = gradient(imfilter(rgb2gray(im),...
%           fspecial('gauss',[3 3]),'symmetric'));
[Hc Vc] = SpatialCues(im);

% graphCut knows how many regions k to split into
%            because Dc is (ny)x(nx)x(k)
gch = GraphCut('open', Dc, 1*Sc); %exp(-Vc*5), exp(-Hc*5)
[gch, L] = GraphCut('expand',gch);
% L is an (ny)x(nx) matrix of {0,1,2,3} corresponding to k label
figure; imshow(L,[]); % show results
gch = GraphCut('close', gch);



%---------------- Aux Functions ----------------%
function v = ToVector(im)
% takes MxNx3 picture and returns (MN)x3 vector
sz = size(im); % [ny, nx, nc]
v = reshape(im, [prod(sz(1:2)), 1]);
% reshapes im into product of (ny*nx) x 1
```

The altered assignment of weights are given in table 1.4.

$e \in \Omega$	w(e)	for
{p,q} n-link	1	$q \neq p$
	0	$q = p$
{s,p} t-link	$\frac{(I_p - I_s)^2}{2\sigma_s^2}$	$\forall\ p \in V$
{p,t} t-link	$\frac{(I_p - I_t)^2}{2\sigma_t^2}$	$\forall\ p \in V$

Table 1.4: The updated weight assignment of the edges Ω in G, when using the k-means clustering graph-cut method.

As can be seen, unlike the interactive graph cut, this algorithm uses an energy function *E(A)* that has a dominant region term. A graph similar to figure 1.17 is yielded. One disadvantage to using k-means clustering is that the user has no influence on what objects to segment out. Rather, the pixels are clustered together automatically. Throughout this paper, we will be comparing the effectiveness between scaling a dominant region term or a dominant boundary term in experimental graph-cut segmentation of images.

Software instructions are provided in section 2.2.

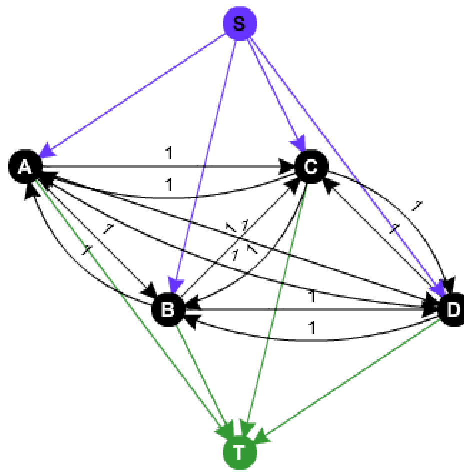


Figure 1.17: G for any 2×2 image. The black edges have weights of 1, the purple edges have weight R(A_p=s) and green edges have weight R(A_p=t).

Chapter 2

Software Instructions

Now that the basic concepts and results of graph-cut segmentation have been covered, one may wish to perform the algorithm themselves. Fortunately, the codes used in this paper are all modified versions of packages which are readily-available online. This chapter strives to be a guideline on the very use and implementation of the interactive graph cut and K-means clustering graph cut whose theorems were coverd in sections 1.3 and 1.4. Keep in mind that a fully-licensed version of MATLAB is required.

2.1 Interactive Graph Cut

Now that the theorem of section 1.3 is understood, the question now is how to begin segmenting images using a program. The source code can be accessed [11] and hence used as intended through the following steps:

1. Visit `http://www.mathworks.com/matlabcentral/fileexchange/21310-maxflow` and downlown the provided zip file to a designated folder.

2. Extract the library to some directory, say we rename this directory as *lib_home*.

3. Open **make.m** on MATLAB and run the compiler.

4. A typical usage of the function on the workspace is the command:

   Listing 2.1: A very basic call of **maxflow.m** once the wrapper runs smoothly.

   ```
   [flow,labels] = maxflow(A,T);
   ```

   Where A represents the n-link weights, A(i,j) being the cost of cutting pixel i from piel j, and T represents the t-link weights, T(i,j) being the cost of cutting pixel i from terminal j. The output "flow" is the maximum flow or minimum cut real-valued number while "labels" is the desired segmentation presented as a binary mask.

5. For a very simple example, try running **test1.m** provided in the library. Because there are only two pixels and two terminals, the values of A and T can be entered manually.

6. A much more complicated segmentation of a grayscale waterfall is provided in **test2.m**. Without altering any values, the results in figure 1.10 should be obtained. Notice that the leftmost pixels will be in the background and the rightmost pixels will be in the object no matter what.

As mentioned previously, the source code behind this method was not originally designed to accept input from the user. This notion is strongly supported by the listing 1.2 and the automatic placement of t-links indicated in figure 1.11. Letting |V| be the number of pixels and k be the number of terminal nodes, the obvious subject to be experimented on is the $|V|\times k$ matrix T, where T(i,j) is the cost of cutting the weight between pixel i and terminal j. In order to alter these starting conditions, the program must be modified to accept object and background polygons from the user in the form of binary masks.

A handy built-in function to handle this job is **roipoly.m**. A quick search of roipoly

in the help page reveals the function to be a graphical user interface which allows the user to draw region of interests (ROI) with the mouse then output it as a binary mask. The visual in figure 2.1 will be shown. A binary mask is an image matrix that is zero everywhere and one at the region of interest. With these masks as a new addition to the input, the goal now is to modify the matrix T(i,s) to be zero everywhere except when i $\in \theta$ and T(i,t) to be zero everywhere except when i $\in \beta$.

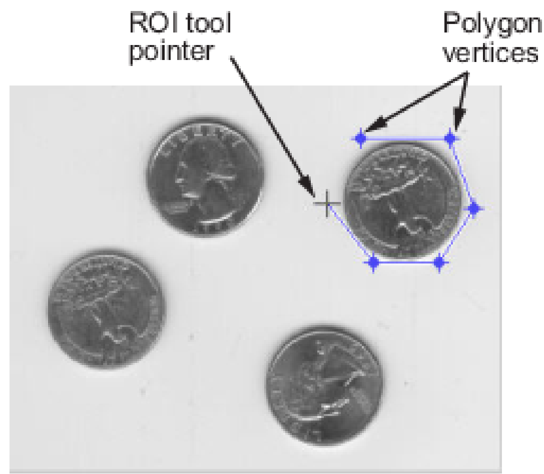


Figure 2.1: A visual of the graphical user interface in the **roipoly.m** function as provided by the official MATLAB help page.

1. Open MATLAB and prompt a new script. Remember to save your work as a .m file in an appropriate directory. For the sake of this procedure, say the name is *interactive.m* under the folder *graph_cut*.

2. Retrieve the matrix of intensities of the desired image, say *someimage.jpg* by using **imread.m**. If it is not a part of MATLAB's built-in demo images, then remember to copy the image file into *graph_cut*.

3. Two **roipoly.m** calls are needed: one for designating an object polygon and one for the background. The resulting masks will have the same dimensions as *someimage.jpg*.

Listing 2.2: Using the function roipoly.m.

```
image = imread('someimage.jpg'); % name.extension
O = roipoly(image);
B = roipoly(image);
```

Note that **roipoly** only provides a graphical interface that is user-friendly. The presence of this function was absent in listing 1.3 because the object and background masks were coded as for loops instead.

4. Obviously, more information about these polygons is required. Firstly, the exact number of pixels in groups θ and β can be obtained as in listing 2.3.

Listing 2.3: Obtain the number of the pixels in each polygon.

```
[height,width] = size(image);
N = height*width; % total number of pixels
NO=0; NB=0; % number of pixels in the objt and bkgd polygons
for x=1:N
    if O(x)==1
        NO=NO+1;
    end
    if B(x)==1
        NB=NB+1;
    end
end
```

5. Recall that we "count" pixels in column-major order. The element in the 20th row and 211th column in a *256×256* image would have an index of $(256 \times 19) + 211 = 5075$. In other words, the pixel in position (i,j) of an image with height h would have index $(h \times (i-1)) + j$. The next piece of information extracted from the object and background polygons is an array of indices of all pixels in θ and β, say we name them PO and PB.

Listing 2.4: Obtain the indices of the pixels in each polygon.

```
% get matrix of pixel INDICES in the object and background polygons
PO=ones(NO,1); PB=ones(NB,1);
mO=1; mB=1;
for x=1:N % counts in column-major order
    if O(x)==1
        PO(mO)=x;
        mO=mO+1;
    end
    if B(x)==1
        PB(mB)=x;
        mB=mB+1;
    end
end;
```

Where O(x) and B(x) are the pixels in the object mask and background mask respectively with pixel index x. The parameters mO and mB are used as counters which increase by one each time a new element is placed in PO or PB in order to ensure correct overwriting.

6. Looking back on **test2.m**, T is coded to be a sparse matrix of size N$\times$2 such that most elements of T are zero. T = sparse(i,j,v) depends on the triplets i, j, and v such that T(i(k),j(k)) = v(k). In this case v(k) would need to be an array of size NO+NB where all entries are an arbitrarily large number. This allows the built-in function **sparse.m** to output infinity as the t-link weight for all instances that are nonzero.

   Alongside, i(k) represents the array of indices of all pixels that are in the object concatenated with the array of all pixels that are in the background. To ensure that NO does not have to equal NB, they are concatenated vertically. Meanwhile, j(k) is an array of size NO+NB, where the first NO entries are 1 and the next NB

entries are 2. In this case, 1 is the label corresponding to the object and 2 is the label corresponding to the background.

In laymen's terms, i(k) stores the pixels that are in a polygon, j(k) keeps track of which terminal these pixels are hard linked to, and v(k) outputs the weights of these links. For example, suppose the pixel with index b is in θ, then $T(b,1) = T(i(b),j(b)) = v(b) = \infty$ where T(d,e) is the cost of cutting the link between pixel d and label e. With this information in mind, the t-link weights can be reassigned as in listing 1.3.

Listing 2.5: Build the T matrix.

```
T = sparse([PB,PO],[ones(NO,1)*1;ones(NB,1)*2],ones(NO+NB,1)*(9^9));
```

Keep in mind that if there doesn't exist a k such that $T(k,l) = T(i(k),j(k)) = v(k)$, then T(k,l) outputs zero so there is no t-link.

7. Because the last remaining steps require the use of the wrapper library downloaded previously, it is important that MATLAB has access to that library. The MATLAB software uses a search path to efficiently locate files called within codes, so *lib_home* must be added to that path. If MATLAB 2015 is the version being used, this can be accomplished by clicking "Set Path" under the home tab in figure 2.2.

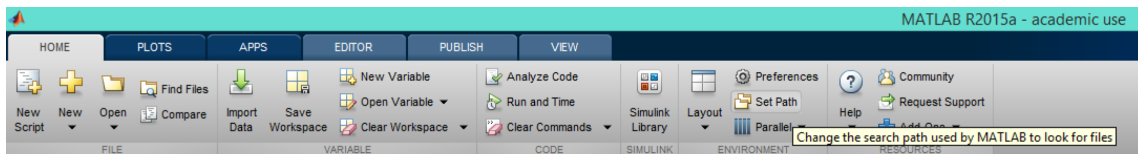


Figure 2.2: The location of the "set path" button.

8. The rest of the algorithm is easy, as the n-link weights remain unchanged from the code listing 1.2 provided in the library.

Listing 2.6: Build the A matrix.

```
E = edges4connected(height,width);
V = -abs (I(E(:,1))-I(E(:,2)))+eps;
A = sparse(E(:,1),E(:,2),V,N,N); % specifies size of A as NxN
```

Where A(p,q) is the cost of severing the link between pixel p and pixel q.

9. Finally, the **maxflow.m** call can be made. Since "labels" is outputted as an N $\times$ 1 column vector, the task is to rearrange it back into a height $\times$ width image through the use of the **reshape.m** function.

Listing 2.7: Using the wrapper to call the function.

```
[flow,labels] = maxflow(A,T);
labels = reshape(labels,[height width]);
```

2.2 K-Means Clustering

The goal is to look at the procedure for segmenting images using k-means clustering based off of the theorem covered in section 1.4. Previously, a simple segmentation was executed in listing 1.4 to test the validity of the source code **??**. The approach this time around is to provide insight on the wrapper library so the user can perform k-means clustering segmentation and compare the results themselves.

1. Open MATLAB.

2. Visit `http://www.wisdom.weizmann.ac.il/~bagon/matlab.html` and scroll down to "Matlab Wrapper for Graph Cuts." Download both the library *GCmex2.0.tar.gz* as well as *example.zip*.

3. Extract the library to a directory with an appropriate name such as *Shai_Bagon*. Repeat the action with *example.zip*, either in the same folder or a different one. In

the latter case, *Shai_Bagon* must be added to the search path through MATLAB. Assuming MATLAB 2015 is the version being used, this can be accomplished by clicking "Set Path" under the home tab in figure 2.3.

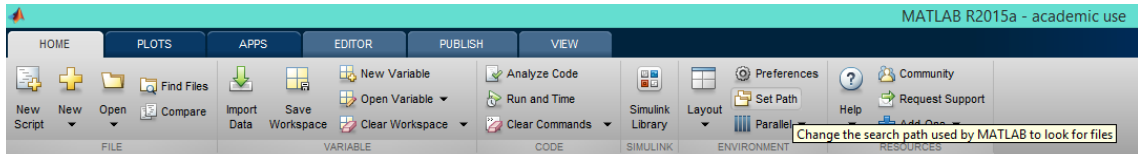


Figure 2.3: The location of the "set path" button.

4. On MATLAB, run the **compile_gc.m** function provided in the library. This wrapper will allow the use of algorithms computed on C++ to be usable from MATLAB.

5. A typical usage of the function on the workspace is the command:

Listing 2.8: A call of **GraphCut.m** once the wrapper runs smoothly.

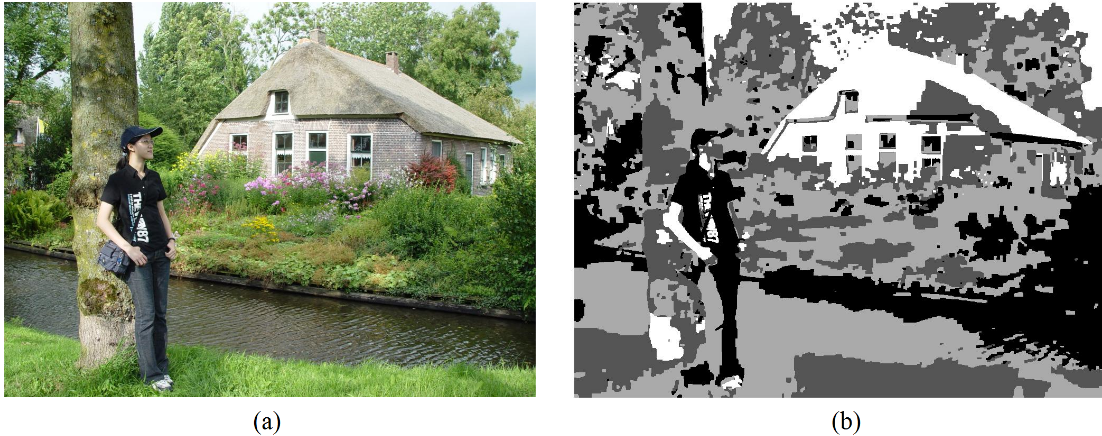
```
gch = GraphCut('open', Dc, Sc);
[gch, L] = GraphCut('expand',gch);
% L is an (ny)x(nx) matrix of {0,1,2,3} corresponding to k label
gch = GraphCut('close', gch);
```

Here, gch is the function handle representing an association with a function. Such a data type allows the passing of a function to another function. The second line performs the optimization while the last line releases the memory.

6. Open **gc_example.m** and try to run the function to obtain the results in figure 2.4 where k is the number of segments.

Notice that unlike interactive segmentation, the different structuring of t-link weights allows groups to be disconnected. Also, we are not limited to two terminal nodes.

Rather, the simple choice of k allows the image to be split into as many segments as desired. For the context of this paper, k will always be 2.



(a) (b)

Figure 2.4: The function **gc_example.m**. (a) The original image. (b) The results.

7. For the sake of comparison with interactive graph cuts, the code can easily be converted to grayscale through two modifications. Firstly, add in an extra line of code underneath **imread.m** as demonstrated by listing 2.9. Then, scroll down to the "Aux Functions" and alter **ToVector.m** as in listing 2.10.

Listing 2.9: The function **gc_example.m** accommodates both colour and grayscale images.

```
im1 = im2double(imread('outdoor_small.jpg'));
im = rgb2gray(im1);
```

Listing 2.10: The function **gc_example.m** accommodates both colour and grayscale images.

```
        %---------------- Aux Functions ----------------%
function v = ToVector(im)
```

```
% takes MxNx1 picture and returns (MN)x1 vector
sz = size(im); % [ny, nx, nc]
v = reshape(im, [prod(sz(1:2)), 1]); % reshapes into (ny*nx) x (nc=1)
```

The significance of turning the colour problem into a grayscale problem is that icv returns the variance of each cluster rather than a 3×3 covariance matrix corresponding to the relation between the red, green, and blue channels. In accordance to equation 1.12, the distance between pixels in a colour image is a 3-norm with its red, green, and blue values as its three attributes.

Letting Dc(i,j,l) be the cost of cutting off pixel in position (i,j) from label l, and Sc(p,q) be the cost of placing pixel p and q into different groups, then the weights are assigned as given in table 1.4.

Chapter 3

Future Work

A difficult problem which usually arises in optimization is the case where the algorithm terminates at a local minimum. To ensure that the solution is a global minimum, there must be repercussions taken to restrain the problem. This issue is not present in graph-cuts because it is a linear problem, and hence gains the advantage over other segmentation methods. Linear problems are convex in nature and the global minimum is always the output. Despite this favorable property, there are significant setbacks which arise from either k-means clustering graph-cuts, interactive graph-cuts, or both.

Interactive Graph-Cut Segmentation

- The soft constraint data of interactive graph-cut segmentation only includes penalties with respect to the boundary term. After setting the hard constraints, this is our solution when λ=0 in the energy function. This segmentation heavily depends on similarity between neighboring pixels as well as the seeds, while ignoring the similarity between the pixel itself and the object and background chosen.

- Possible solution: In the future, it will be useful to incorporate region penalties into the optimization. One method of defining such a function involves gathering information from the "seeds" themselves then using the intensities of the pixels

in the seeds to form two distributions: one for object and one for background. This distribution can then act as a prior probability when trying to calculate the probability of a pixel outside θ and β to be in the object or background.

- Assuming the user only selects one object polygon in interactive graph-cut segmentation, then the object segment must be a simply connected region. For instance, if an image has two apples on opposite sides of the screen and one wants to define the object as apples, this algorithm will only be able to discern whichever apple was chosen by the user as the sample seed θ.

- Possible solution: An improvement to the code would be to allow the user to choose a finite number n of object seeds and a finite number m of background seeds. The matrix of t-link weights T can be modified to hard-link terminals s to all pixels p $\in \theta = \theta_1 \cup \theta_2 \cup ... \cup \theta_n$ and t to all pixels p $\in \beta = \beta_1 \cup \beta_2 \cup ... \cup \beta_m$, then the graph-cut performed as normal.

K-means Clustering Graph-Cut Segmentation

- K-means clustering is an example of an unsupervised segmentation, a method of cutting which is automatically performed by the computer with no input from the user. In many cases, this lack of flexibility can render the results useless when a user wants to segment a specific object in the image. For example, a graphic designer with a photo of a rose garden may wish to segment a single flower rather than the entire bush.

- The soft constraint data of interactive graph-cut segmentation only includes penalties with respect to the region term. After setting the hard constraints, this is our solution when $\lambda \to \infty$ in the energy function. This segmentation heavily depends on similarity between the pixel itself and the corresponding centroid, while ignoring the similarity between neighboring pixels.

- Possible solution: Given the methods explored so far, a reasonable solution is to combine the structure of the interactive segmentation with the k-means clustering segmentation. The new algorithm will begin by collecting object θ and background β seeds from the user. As in the interactive method, pixels in these seeds will then be hard-linked to the s and t terminals respectively. However, instead of setting the t-link weights of the remaining pixels as 0, the data collected from k-means clustering is used. The boundary region penalties $B_{\{p,q\}}$ are unchanged from equation 1.9. The summary of the new graph is given in table 3.1 and figure 3.1.

e $\in \Omega$	w(e)	for
{p,q} n-link	$B_{\{p,q\}}$	q $\in N_p$
	0	q $\notin N_p$
{s,p} t-link	$\frac{(I_p - I_s)^2}{2\sigma_s^2}$	$\forall$ p $\in V \backslash \theta$
	∞	$\forall$ p $\in \theta$
{p,t} t-link	$\frac{(I_p - I_t)^2}{2\sigma_t^2}$	$\forall$ p $\in V \backslash \beta$
	∞	$\forall$ p $\in \beta$

Table 3.1: The updated weight assignment of the edges formed as a combination of interactive and k-means clustering segmentation. The colors indicate corresponding edges in figure 3.1.

Keep in mind that this paper uses very simplistic models for the boundary $B_{\{p,q\}}$ and region $R(A_p)$ penalties. The beauty of graph-cuts segmentation is that a large variety of mathematical models can be used in combination with the theory discussed in this paper. For instance, the programmer may implement k-means clustering to obtain statistical information on pixels with similar intensities in order to form more dynamic t-link weights. The efficiency of the results from above are by no means the endpoint of graph-cuts in image segmentation. On the contrary, the use of these models will continue

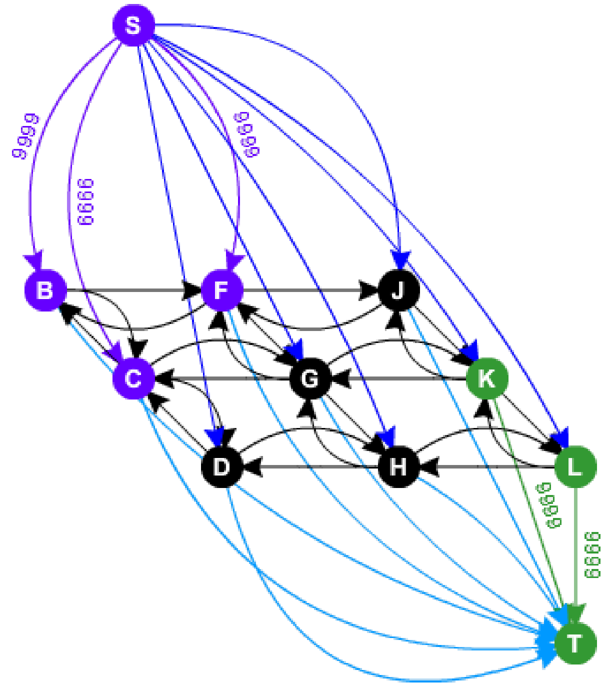


Figure 3.1: The new 3×3 graph formed as a combination of interactive and k-means clustering segmentation, where $\theta = \{B, C, F\}$ and $\beta = \{K, L\}$.

to improve and grow as more complex problems are presented.

Bibliography

[1] Shai Bagon. Matlab wrapper for graph cut, December 2006.

[2] Samir Kumar Bandyopadhyay. Survey on segmentation methods for locating masses in a mammogram image. *International Journal of Computer Applications*, 9(11):25–28, 2010.

[3] Leon Bottou and Yoshua Bengio. Convergence properties of the k-means algorithms. In *Advances in Neural Information Processing Systems 7*. Citeseer, 1995.

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

[5] Yuri Boykov, Olga Veksler, and Ramin Zabih. Markov random fields with efficient approximations. In *Computer vision and pattern recognition, 1998. Proceedings. 1998 IEEE computer society conference on*, pages 648–655. IEEE, 1998.

[6] Yuri Boykov, Olga Veksler, and Ramin Zabih. Efficient approximate energy minimization via graph cuts. *IEEE transactions on Pattern Analysis and Machine Intelligence*, 20(12):1222–1239, November 2001.

[7] Yuri Y Boykov and Marie-Pierre Jolly. Interactive graph cuts for optimal boundary & region segmentation of objects in nd images. In *Computer Vision, 2001. ICCV*

2001. Proceedings. Eighth IEEE International Conference on, volume 1, pages 105–112. IEEE, 2001.

[8] Vladimir Kolmogorov and Ramin Zabih. What energy functions can be minimized via graph cuts? *IEEE transactions on Pattern Analysis and Machine Intelligence*, 26(2):147–159, February 2004.

[9] Bernhard Korte, Jens Vygen, B Korte, and J Vygen. *Combinatorial Optimization.* Springer, 2002.

[10] National Council of Teachers of Mathematics. Graph creator. http://illuminations.nctm.org/Activity.aspx?id=3550, cited February 2016.

[11] Miki Rubinstein. Wrapper library for graph cut, September 2008.

[12] KJ Shanthi, DK Ravish, and M Sasikumar. Image segmentation an early detection to alzheimer's disease. In *India Conference (INDICON), 2013 Annual IEEE*, pages 1–4. IEEE, 2013.

[13] Hlavac Svoboda, Kybic. *Image processing, analysis, and machine vision: A MATLAB companion.* Thomson Learning, 2008.

[14] Kardi Teknomo. K-means clustering. http://croce.ggf.br/dados/, cited February 2016.

[15] "Wikipedia". Max-flow min-cut theorem — wikipedia, the free encyclopedia, February 2016.