Image segmentation via graph cuts without network flow overhead

Rushik Anil Vartak
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
rv9981@g.rit.edu

***Abstract*—The algorithm proposed by Rachel Silva [1] computes the count of the minimum (s,t) - cuts in an undirected weighted planar graph without preprocessing the graph using a max-flow algorithm. It can be used for image segmentation, where each minimum (s,t) - cut in the planar graph representing the image is equivalent to a distinct segmentation in the image. In this capstone project, Rachel Silva's algorithm [1] is implemented to count the number of minimum (s,t) - cuts in an image, and sample through the cuts to produce an image segmentation. In addition, experiments are performed to evaluate the performance of the algorithm and to prove the hypothesis that 'generating a single minimum (s,t)-cut might be sufficient for most of the images with almost identical segmentations; but there exist images with a number of distinct segmentations, for which it is necessary to count and sample through the minimum (s,t)-cuts'.**

I. Introduction

One of the applications of image segmentation is to separate an object in an image from its background. This can be achieved using graph cuts [2], where a minimum (*s*,*t*) - cut is computed for the network flow graph representing the image to be segmented. In this graph, each pixel in the image is represented as a vertex. The edges in the graph connect the pixels with their neighboring pixels; where the neighboring pixels of a pixel can be the pixels above, below, to the left or to right of the pixel. Each edge is associated with a weight which represents the function of similarity of intensities between two neighboring pixels. The image is segmented by performing a minimum (*s*,*t*) - cut on this graph by selecting the source *s* as one of the pixels inside the object and the sink *t* as one of the pixels in the background.

As there can be several distinct minimum (*s*,*t*) - cuts in a graph, there can be several distinct segmentations of the image represented by that graph. A selected segmentation of an image, resulting from a minimum (*s*,*t*) - cut of the graph representing the image, might not be the segmentation expected by the user. Providing the user with all the possible minimum (*s*,*t*) - cuts is not a practical approach as the number of minimum (*s*,*t*) - cuts in a graph can be exponential [3]. An approach that solves this problem is to count the number of minimum (*s*,*t*) - cuts in a graph and sample from several minimum cuts, which enables the user to select the desired segmentation. But, counting the number of minimum (*s*,*t*) - cuts in a graph is a #P-complete problem [3].

Researchers have presented polynomial-time algorithms [4][3] that solve the problem of counting the number of minimum (*s*,*t*) - cuts in a planar graph. These algorithms require that the planar graph is preprocessed using a max-flow algorithm to get the residual graph. The actual computations, required to count the number of minimum (*s*,*t*) - cuts, are performed on this residual graph.

The algorithm presented by Silva [1] computes the number of minimum (*s*,*t*) - cuts in an undirected weighted planar graph without the need of preprocessing the graph using a max-flow algorithm. Though, this algorithm is not computationally faster than the algorithms that preprocess the planar graph using max-flow algorithm [4][3] to compute the number of minimum (*s*,*t*) - cuts; it is computationally and conceptually simpler [1]. This algorithm depends on the correlation of the cuts in a planar graph with certain cycles in the dual of that graph. Using this correlation, the number of minimum (*s*,*t*) - cuts can be efficiently counted by computing the number of cycles in the dual graph that represent the minimum (*s*,*t*) - cuts in the planar graph.

This capstone project focuses on implementing the algorithm presented by Silva [1] and using it to perform image segmentation. This implementation is used to perform experiments on benchmark images to evaluate how well the image is segmented using the algorithm presented by Silva [1]. The algorithm is tested against numerous benchmark images in an attempt to prove the proposed hypothesis, which states that 'Generating a single minimum (s,t)-cut might be sufficient for the images with almost identical segmentations represented by all its minimum (s,t)-cuts; but there exist images with more than one distinct segmentations, for which it is necessary to count and sample through the minimum (s,t)-cuts'.

II. Related work

Ball and Powan devised an algorithm that computes the number of minimum cuts in a directed unweighted planar

graph in polynomial time [4]. Their algorithm reduces the problem of counting the number of minimum cuts to the problem of counting the number of maximal antichains in the processed graph; where an antichain is a set of vertices such that the predecessor of each vertex in the set is not present in the set and a maximal antichain is an antichain which is not a proper subset of any other antichain [5]. But, it assumes that both source *s* and sink *t* lie on the same face.

Bezáková and Friedlander extends the algorithm presented by Ball and Powan [4] by designing an algorithm that computes the number of minimum cuts in any weighted planar graph in polynomial time [3]. This algorithm has a runtime complexity of $O(nd + nlogn)$, where n is the number of vertices in the graph and d is the length of the shortest path between source *s* and sink *t* of the contracted residual graph. It does not depend on the locations of source *s* and sink *t*, but it does assume that all the vertices lie on some path from *s* to *t*. It reduces the problem of finding the number of minimum cuts in a planar graph to the problem of finding the number of paths between certain faces in the dual of the residual graph obtained by running the max-flow algorithm [6].

Joshi in his Master's project [7], under the guidance of Bezáková, implemented the algorithm presented by Bezáková and Friedlander [3], and performed experiments using the implementation to determine the number of minimum cuts present in an image and the relation of the number of minimum cuts with the size of the image. The results of these experiment show that the number of minimum cuts are directly proportional to the size of the image. Additionally, image segmented by different minimum cuts were compared to analyze the differences between each segmentation.

Gomsale in his Master's project [8], under the guidance of Bezáková, extended Joshi's work [7] by implementing probabilistic sampling in the algorithm presented by Bezáková and Friedlander [3]. In the sampling step of this implementation, instead of randomly selecting the edges that form paths from source *s* to sink *t* in the dual graph, the edges are selected based on their probabilities. The probability of each edge is measured by evaluating the number of times the edge appears in all the possible minimum cuts. In this project, experiments were also performed to compare the performance of the algorithm presented by Bezáková and Friedlander [3] with the performance of its extension that uses probabilistic sampling [7], and based on the results of these experiments it was concluded that using probabilistic sampling produced more realistic segmentation boundaries.

Shah in his Master's project [9], under the guidance of Bezáková, evaluated the algorithm presented by Bezáková and Friedlander [3] and performed experiments using measures of precision and Jaccard index to compare its results with other image segmentation algorithms. The evaluation and experiments are performed using the implementations of the algorithm developed by Gomsale [8] and Joshi [7]. Based on the results of the experiments, it was concluded that the algorithm does not perform very well with segmentation of objects with vast color differences in it, or if the colors in the object are very similar to the background colors, or if the object is very close to the border.

Silva in her Master's thesis [1] proposed an alternate algorithm to count the number of minimum cuts in undirected weighted planar graphs without running the max-flow algorithm [6]. This algorithm has a runtime complexity of $O(V^2 logV)$, where V is the number of vertices in the graph. Though, this algorithm does not run faster than the previously proposed algorithms [4][3][8], it does simplify the process of finding the minimum cuts, both computationally and conceptually, by reducing the pre-processing steps performed on the graph to get the residual graph. It depends on the correlation of the cuts in the planar graph with certain cycles in the dual of that graph, using which the number of minimum cuts can be efficiently counted.

III. Preliminaries

A. Planar graphs

A planar graph is a graph that can be embedded in a plane without its edges intersecting each other. Fig. 1. shows an example of a planar graph on the left and a non-planar graph on the right. Unless otherwise mentioned, all the graphs discussed in this project are planar graphs.

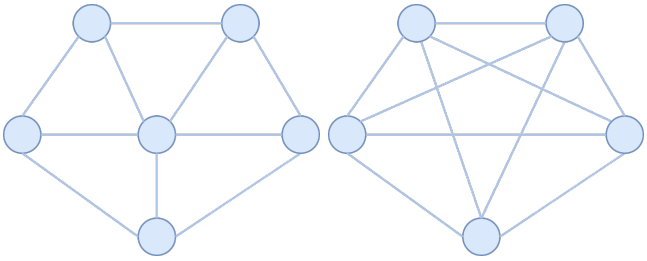

Fig. 1: Examples of planar graph (left) and non-planar graph (right)

A dual of a graph G with the set of faces F and the set of edges E, is a graph G_d with the set of vertices V_d and the set of edges E_d; such that every vertex $v_d \in V_d$ is represented by a face $f \in F$ and every edge $e_d \in E_d$ is represented by an edge $e \in E$ adjacent to the faces in G. Fig. 2. shows a dual representation of a planar graph, where the green vertices are the vertices of the dual graph (faces of the original graph) and the green edges are the edges of the dual graph.
Euler's formula for planar graphs is as follows

$$V - E + F = 2 \tag{1}$$

where V is the number of vertices, E is the number of edges, and F is the number of faces in the planar graph.

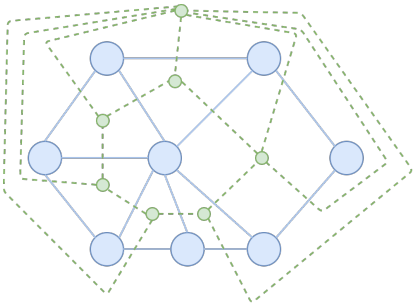

Fig. 2: Example of the dual representation of an undirected planar graph

B. Minimum (s,t) - cut

Given a graph G with the set of vertices V, the set of weighted edges E, a source vertex s such that $s \in V$, and a sink vertex t such that $t \in V$; an (s,t) - cut is a set of vertices $S \subseteq V$ such that $s \in S$ and $t \notin S$. Weight of an (s,t) - cut is the sum of the weights of all the edges $\{u, v\}$ such that $u \in S$ and $v \notin S$. A minimum (s,t) - cut is an (s,t) - cut with the minimum weight. There can be more than one minimum (s,t) - cuts in a graph, as shown in Fig. 3. where there are 5 minimum (s,t) - cuts with the weight of 7.

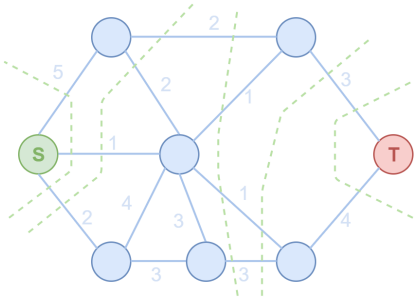

Fig. 3: Examples of minimum (s,t) cuts in an undirected weighted planar graph

C. Rightmost shortest path

Rightmost shortest path (clockwise shortest path) is a shortest path between two vertices such that no other shortest path branches out from it in the clockwise direction [1]. There can be more than one rightmost shortest paths in a graph. Fig. 4. shows a graph with edges having unit weights, and some of the shortest paths from the starting vertex S to the destination vertex T. The shortest path $S \Rightarrow 3 \Rightarrow 2 \Rightarrow T$ is not a rightmost shortest path as there is another shortest path $S \Rightarrow 3 \Rightarrow 5 \Rightarrow T$ that branches out in the clockwise direction, which is a rightmost shortest path. The shortest paths $S \Rightarrow 4 \Rightarrow 5 \Rightarrow T$ and $S \Rightarrow 1 \Rightarrow 2 \Rightarrow T$ are also rightmost shortest paths in this graph.

D. Edges / vertices below a path

Consider the position of the planar graph such that the paths from S to T go from left to right. Here, the edges and vertices

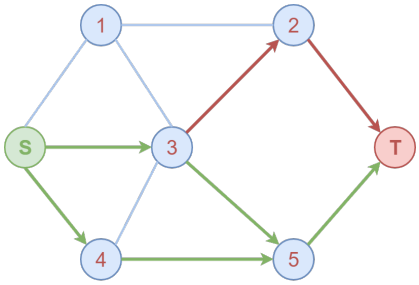

Fig. 4: Examples of rightmost shortest path (green) branching out from another shortest path (red)

are said to be below a path p if they lie below the path p with respect to the considered position. In Fig. 4., consider the path $S \Rightarrow 3 \Rightarrow 2 \Rightarrow T$; here the vertices below the path are 4 and 5, and the edges below the path are $\{3, 4\}$ and $\{3, 5\}$.

E. Algorithm

Algorithm 1 states the algorithm being implemented in this project [1] which is used to count the number of minimum cuts in an undirected weighted planar graph. The input to the algorithm is an undirected weighted graph G, as shown in Fig. 5., with the set of vertices V, the set of weighted edges E, a source vertex s such that $s \in V$, and a sink vertex t such that $t \in V$. The steps 1-4 are the preprocessing steps and the step 5 is the step where the number of minimum cuts is computed.

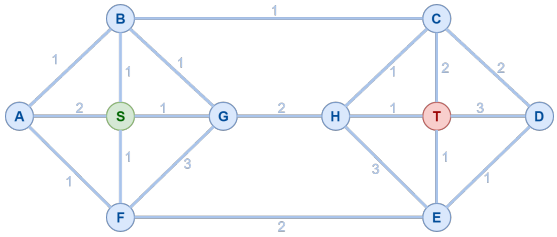

Fig. 5: Example of an undirected weighted planar graph

In step 1, a graph G' is created as a copy of graph G. In G', s' is the source vertex and t' is the sink vertex. The aim of this step is to make each of the s' and t' vertices incident to a single face. For source s', each edge incident to s' is bisected with a new vertex s_i. These newly formed vertices are then connected to each other in a clockwise-cyclic order to form a cycle around s' with edge weights equal to the sum of all edges in G'. The edges from s' to the newly formed vertices are then deleted. This results in s' being incident to only a single face f_s as shown in Fig. 6. Similarly, t' is also made incident to a single face f_t.

In step 2, a dual graph G'_d is created which is the dual representation of the graph G' as shown in Fig. 8. The vertices s_d and t_d in G'_d represent the faces f_s and f_t respectively in G'. The weights of the edges in G'_d are equivalent to

Algorithm 1: Counting minimum (s,t) - cuts [1]

$G \leftarrow$ undirected weighted planar graph
$V \leftarrow$ set of vertices of G
$E \leftarrow$ set of edges of G
$s \leftarrow$ source vertex in V
$t \leftarrow$ sink vertex in V

1. $G' \leftarrow$ copy of G
   make s' incident to a single face f_s
   make t' incident to a single face f_t
2. $G'_d \leftarrow$ dual of G'
   where $f_s \leftarrow s_d$ & $f_t \leftarrow t_d$
3. $p \leftarrow$ rightmost shortest path between s_d & t_d
4. $\hat{G} \leftarrow$ copy of G'_d

for *every vertex* v_i *in* $\hat{p}$ *except* $\hat{s}$,$\hat{t}$ **do**
  $u_i \leftarrow$ copy of v_i
  for *every edge* $\{v_i, x\}$ *below* $\hat{p}$ **do**
    create edge $\{u_i, x\}$
    $weight(u_i, x) = weight(v_i, x)$
    delete edge $\{v_i, x\}$
  end
end

5. $count_{cuts} \leftarrow 0$
$weight_{min} \leftarrow \infty$

for *every pair of vertices* $\{v_i, u_i\}$ *in* $\hat{p}$ **do**
  $count_i \leftarrow$ number of shortest paths between $\{v_i, u_i\}$
  $weight_i \leftarrow$ weight of shortest paths between $\{v_i, u_i\}$
  if $weight_i < weight_{min}$ **then**
    $count_{cuts} \leftarrow count_i$
    $weight_{min} \leftarrow weight_i$
  else if $weight_i = weight_{min}$ **then**
    $count_{cuts} \leftarrow count_{cuts} + count_i$
  else
    continue;
  end
end
return $count_{cuts}$

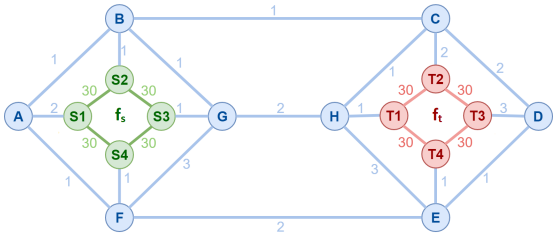

Fig. 6: Graph G' after making source s and sink t each incident to a single face f_s and f_t respectively

the weights of the edges they intersect in G' as shown in Fig. 7.

In step 3, a rightmost shortest path p is computed from s_d to t_d in G'_d. For graphs with more than one rightmost shortest paths, p is selected arbitrarily. In Fig. 9., an arbitrarily chosen rightmost shortest path p from s_d to t_d in G'_d is highlighted.

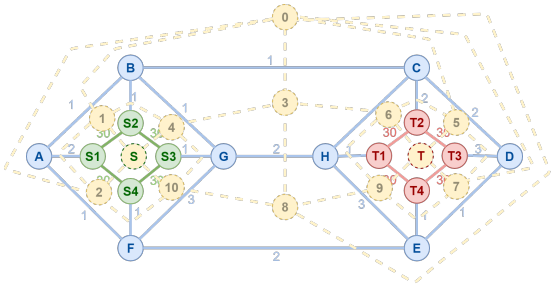

Fig. 7: G' with the dual representation of the graph G'

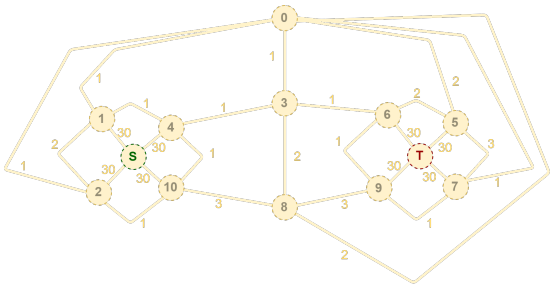

Fig. 8: Graph G'_d: dual representation of the graph G'

In step 4, a graph $\hat{G}$ is created as as copy of the graph G'_d. In $\hat{G}$, a copy u_i is created for each vertex v_i in p except source $\hat{s}$ and sink $\hat{t}$. For every edge $\{v_i, x\}$ below the rightmost shortest path $\hat{p}$, an edge is created between x and the copy u_i with weight equal to that of edge $\{v_i, x\}$; and the edge $\{v_i, x\}$ is deleted. Fig. 10. shows the newly formed copies of the vertices in $\hat{p}$ except s_d and t_d.

In step 5, for each vertex v_i in $\hat{p}$ except s_d and t_d, the number of shortest paths between v_i and its copy u_i is counted using a shortest path algorithm like Dijkstra's algorithm. If the weight of the shortest path between v_i and u_i is the minimum, then the sum of counts is updated by adding the count of shortest paths between v_i and u_i. Fig. 11. shows all the 5 shortest paths with minimum weight of 5 for the different pairs of v_i and u_i in different colors. The final sum of the counts of shortest paths is equivalent to the number of minimum (s,t) - cuts in the original graph G. Fig. 12. shows the minimum (s,t) - cuts in the original graph G that are equivalent to the shortest paths found in $\hat{G}$.

F. Time complexity

For a planar graph V vertices, the run-time complexity of the graph modification steps 1, 2, and 4 of algorithm 1 is $O(V)$. Finding the right-most shortest path in the step 3 of algorithm 1 takes $O(VlogV)$ time using Dijkstra's algorithm with min priority queue. Counting the number of minimum separating cycles in step 5 of the algorithm 1 takes $O(d(VlogV))$ time, where d is the number of vertices in right-most shortest path. As d is bounded by V, the overall time complexity of the algorithm is $O(V^2logV)$.

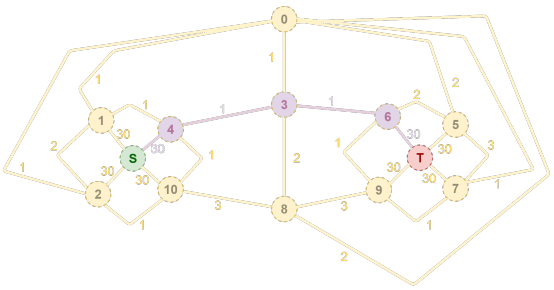

Fig. 9: Rightmost shortest path p from s_d to t_d in G'_d

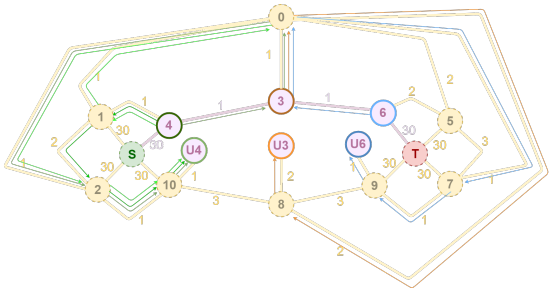

Fig. 11: Shortest paths between each pair of vertices $\{v_i, u_i\}$

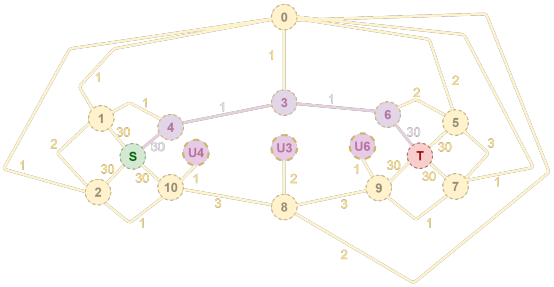

Fig. 10: Copies of vertices in p except s_d and t_d are created

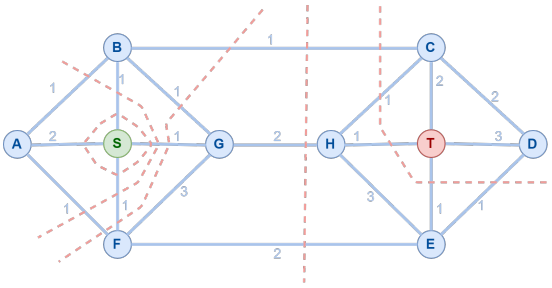

Fig. 12: Minimum (s,t) - cuts in the original graph G

IV. Implementation

A. Image representation

The image chosen for segmentation is represented as an undirected weighted planar graph, where each vertex represents a pixel in the image and the edges between the vertices represent the similarity between the pixels. The weight of the edge is determined by an edge weight function which is a function of the difference between the luminance of the pixels represented by the vertices incident to the edge.

B. Data Structure

The data structure used to store the planar graph is a modified adjacency list which also stores the planar embedding of the graph [1]. As in an adjacency list, for every vertex v in the graph there exists a list of nodes $\{nbr_1, nbr_2, ..., nbr_n\}$ which represent the edges to the neighbors of v, where n is the number of neighbors of v. These nodes are arranged in a clockwise-cyclic order in the list, i.e. nbr_2 is to the right of nbr_1, nbr_{n-1} is to the right of nbr_n, nbr_1 is to the right of nbr_n, etc. These nodes contain the information about the edges like the edge weight, and the other endpoint of the edge; in addition to this information, the nodes also have 3 pointers viz. a pointer to the left neighbor, a pointer to the right neighbor, and a pointer to the node that v represents in the list associated to the nbr_i. Fig. 13. shows a planar graph with the modified adjacency list that preserves the planar embedding of the graph. For every vertex v in the graph, the list of nodes arranged in clockwise cyclic order is shown; the green arrows in the list represent pointers to the left and right nodes in a list, and the red arrows in the list represent pointers to the node that represents v in the list associated to the nbr_i.

C. Edge weight function

The edge weight function is a very important factor in image segmentation as the quality of the segmentation depends largely on it. An edge weight function should output weights which have a large range (to guarantee that two weights vary greatly), which will ensure that the minimum (s,t) - cut is not limited to the edges incident to the source pixel or sink pixel, but instead results in a large cut.
The edge weight function is a function of the difference between the luminance of two pixels. The luminance of a pixel is the brightness value of the pixel in gray-scale. It can be defined by the following formula [7]:

$$luminance(r,g,b) = 0.2126 * r + 0.7152 * g + 0.0722 * b \quad (2)$$

where r is the red color intensity value, g is the green color intensity value, and b is the blue color intensity value. The edge weight functions considered for this project were [7]:

$$f(LumDiff) = (256 - LumDiff) \quad (3)$$

$$f(LumDiff) = (255 - LumDiff)^8 + 1 \quad (4)$$

$$f(LumDiff) = \frac{1}{(LumDiff + 1)^2} \quad (5)$$

$$f(LumDiff) = (\frac{1}{LumDiff + 1} * 1000)^4 * (W * H) \quad (6)$$

$$f(LumDiff) = (\frac{1}{LumDiff + 1} * 1000)^8 * (W^{10} * H^{10}) \quad (7)$$

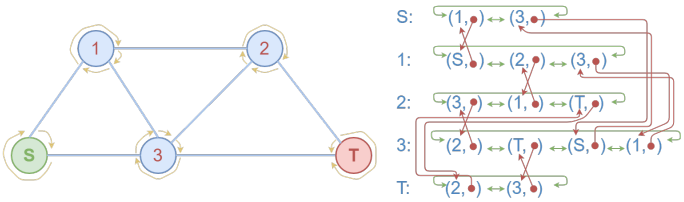

Fig. 13: Modified adjacency list that stores planar embedding

where $LumDiff$ is the difference between the luminance of two pixels, W is the width of the image, and H is the height of the image.

The edge weight function mentioned in Eq. (3) is not a suitable function as it produces weights in the range of 1 to 255. This often results in a segmentation which includes pixels that are the neighboring pixels of the source or the sink; as due to the low variance in the weights, the cut with minimum weight is often the one that includes the edges incident to the source or the sink.

The edge weight functions stated by Eq. (4) and Eq. (5) perform well as the variance between two weights is large enough i.e. it has a large output range. The edge weight function stated by Eq. (6) works better as it scales the range of output based on the width W and height H of the image. Eq. (7) is a modified version of the Eq. (6) which is used in this project.

The maximum threshold $MaxDiff$ is chosen for the difference between the luminance of two pixels $LumDiff$ is 20, as there is no need to consider any difference in luminance more than 20 due to the nature of images in real world[7].

D. Image segmentation

To perform image segmentation, source pixel(s) and sink pixel(s) should be selected. The source pixel(s) should be located inside the object being segmented in the image, while the sink pixel(s) should be located outside the object in the background of the image.

E. Probabilistic sampling

The number of minimum (s,t) - cuts in an image can be exponential, and thus it is necessary to sample through them to choose a single minimum (s,t) - cut representing an image segmentation. Choosing a left-most / right-most minimum (s,t) - cut or an arbitrary minimum (s,t) - cut might not give the best result every time. Thus, probabilistic sampling [8] is used where a minimum (s,t) - cut is chosen probabilistically, i.e. the chances of a segmentation being chosen from a dense set of segmentations (set of large number of identical segmentations) are more as compared to the chances of a segmentation being chosen from a sparse set (set of small number of identical segmentations). Initially, a pair of vertices $\{v_x, u_x\}$ is chosen probabilistically from all the $\{v_i, u_i\}$ pairs obtained in step 4 of algorithm 1. The probability used for choosing each pair is computed based on the count of shortest paths that exist between that pair and the total count of shortest paths. In Fig. 14., the pair $\{4, u4\}$ may be chosen with a probability of $\frac{3}{5}$; and the pairs $\{3, u3\}$ and $\{6, u6\}$ may each be chosen with a probability of $\frac{1}{5}$.

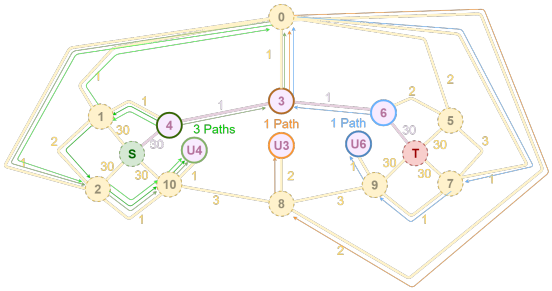

Fig. 14: Probabilistically choosing a pair of vertices $\{v_i, u_i\}$

Once a pair of vertices $\{v_x, u_x\}$ is chosen, the next task is to probabilistically choose a shortest path that exists between the vertices $\{v_x, u_x\}$. Fig. 15 shows a graph with a source S and a sink T, and the values above the vertices represent the number of shortest paths that exist from that vertex to the sink T. A shortest path is probabilistically chosen by walking from source S to sink T and choosing the next vertex v probabilistically based on the number of shortest paths that go from v to T. Probability for each next vertex x is computed based on the count of shortest paths that exist from x to T and the count of shortest paths that exist between the current vertex and T. In Fig. 15. after starting with S, the vertex A may be chosen with a probability of $\frac{2}{3}$ or the vertex D may be chosen with a probability of $\frac{1}{3}$. Assuming A is probabilistically chosen, the next vertices B and C can each be chosen with a probability of $\frac{1}{2}$. Similarly, in each iteration the next vertex is chosen probabilistically until the sink T is encountered.

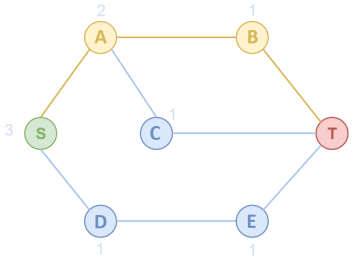

Fig. 15: Probabilistically choosing a shortest path between two vertices

F. Programming languages used

The algorithm 1. was initially implemented in Python, and was later implemented in Java for better performance. The image segmentations generated by both the implementations are similar, but the running times of the Python implementation are extremely high as compared to the running times of the Java implementation as seen in table I.

Image resolution	Python	Java
$800 * 800$	$> 30mins$	$2mins50secs$
$400 * 400$	$19mins42secs$	$33secs$
$200 * 200$	$2mins7secs$	$2secs$
$100 * 100$	$25secs$	$< 0sec$

TABLE I: Run times for implementations in Python vs. Java

Algorithm 2: Modified Dijkstra's algorithm

```
G ⇐ Graph
S ⇐ Source vertex
PQ ⇐ Min Priority Queue with all vertices in G
Dist ⇐ Distances for each vertex in G
Count ⇐ Counts for each vertex in G
PQ.insert(S, 0)
while PQ is not empty do
    curr ⇐ PQ.extractMin()
    for each neighbor nbr of curr do
        weight = curr.weight + nbr.weight
        if weight == Dist[nbr] then
            Count[nbr]= Count[nbr]+Count[curr]
        else
            weight ¡ Dist[nbr]
        Dist[nbr] = weight
        Count[nbr] = Count[curr] if PQ contains nbr then
            PQ.decreaseKey(nbr, weight)
        else
            PQ.insert(nbr,weight)
        end
    end
end
return Dist, Count
```

G. Shortest path algorithm

The Dijkstra's algorithm is used in the Algorithm 1 for finding the rightmost shortest path in step 3 and finding the number of shortest paths in step 5. In addition to find the shortest distance to each vertex in the graph, the Dijkstra's algorithm is modified to also keep a count of the shortest paths as shown in algorithm 2. An implementation of min priority queue [10] is used in the Dijkstra's algorithm to achieve the time complexity of $O(VlogV)$ where V is the number of vertices in the graph.

H. Scribbles for source and sink

Instead of choosing a single pixel each for source and sink, a better image segmentation can be achieved by choosing multiple pixels (scribbles) for the source and the sink as shown in Fig. 16. The pixels representing the source or the sink should be connected i.e. should be a single scribble; and the scribbles should not cross over itself or each other. Instead of making just a single vertex s' and t' each incident to a single face in step 1 of algorithm 1, all the vertices representing the scribble for source and sink are made incident to the faces f_s and f_t respectively.

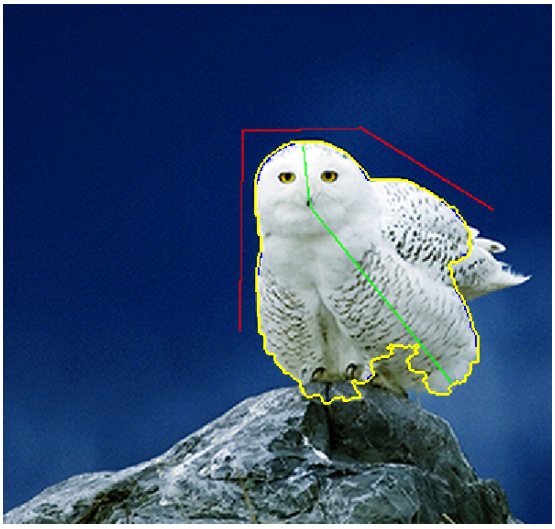

Fig. 16: Image segmentation using scribbles for source and sink. Image ref. [11]

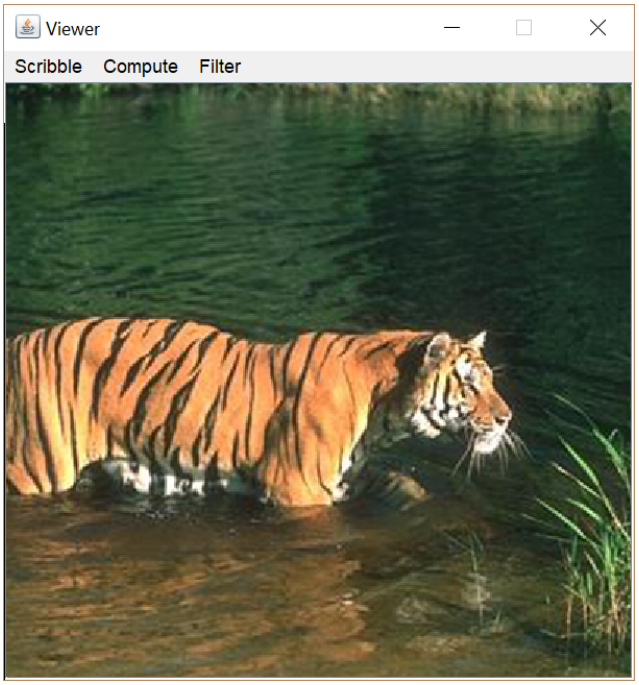

Fig. 17: Java implementation User Interface

I. Running the implementation

The code for the implementation is available at https://github.com/rushikv/ImageSegmentationGraphCuts.
The $UI.java$ file is used to run the implementation user interface, which is as shown in Fig. 17. To run the implementation, all the java files should be first compiled using '$javac * .java$' and then the command '$java\ UI\ image\_file\_name$' can be executed in the command line interface / terminal; where $image\_file\_name$ is the name of the image file.
The user interface has a menu bar with 3 sub menus, viz.

Scribble, Compute, and Filter. The options Draw_S and Draw_T under the Scribble menu can be used to draw the scribbles for source and sink respectively. The option Left_Right_Most under Compute menu is used to compute the left-most and right-most image segmentations, whereas, the Prob_Sample option under Compute menu is used to find display n probabilistically chosen samples, where n is taken using an input dialog box. The options Grayscale and RGB under Filter can be used to display the image in Gray-scale or RGB mode respectively.

V. Experimentation

All the experiments were performed using the images from the Berkeley segmentation dataset [12]. Fig. 18. shows the image segmentations of 10 different images with the source (in green) and sink (in red) scribbles and the total number of minimum (s,t) - cuts. The 2^{nd} column shows the left-most segmentation (blue) and the right-most segmentation (yellow) of the images, whereas the 3^{rd} column shows 1000 samples out of all the segmentations (fuzzy yellow: where brighter yellow pixels indicates that they are part of more segmentation samples than the faint yellow pixels) chosen using probabilistic sampling.

A. Experimenting with different image dimensions

The image in Fig. 16 [11]. is used to test the algorithm against different image resolutions. The results as shown in Table II, show that as the resolution of the image decreases the count of minimum (s,t)-cuts (segmentations) decreases. The quality of the image segmentations also reduces as the resolution decreases, especially for images with lower resolutions as seen in Fig. 19.

Image resolution	No. of min cuts	Run-time
$1000 * 1000$	$7.16 * 10^{147}$	$8mins2secs$
$800 * 800$	$4.54 * 10^{112}$	$1mins42secs$
$400 * 400$	$9.47 * 10^{47}$	$16secs$
$200 * 200$	$9.93 * 10^{16}$	$2secs$
$100 * 100$	8460	$< 0sec$
$50 * 50$	1	$< 0sec$

TABLE II: No. of min cuts and run-times for different image resolutions

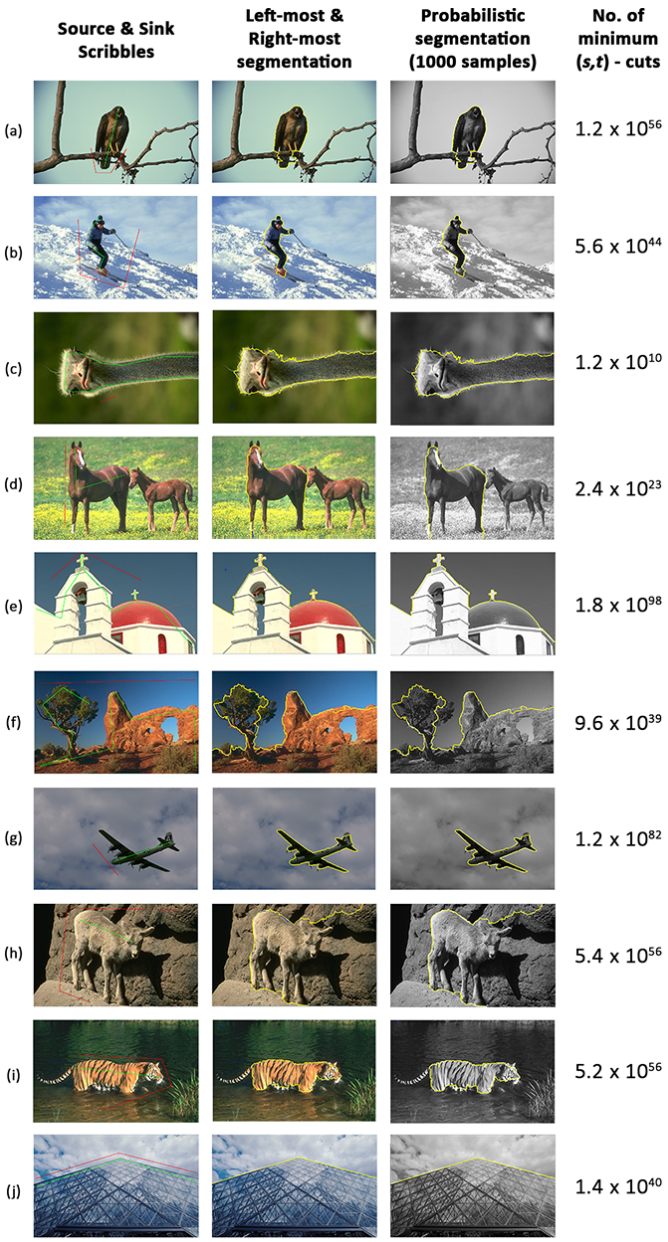

Fig. 18: Segmentations of 10 images of size 481 x 321 pixels from the Berkeley dataset [12] with the number of minimum cuts

B. Testing with different edge weight functions

The image [11] was used to test the image segmentations obtained by using the different edge weight functions (Eq. (3-7)) mentioned in section IV-C; and the source and sink scribbles used in Fig. 16. The edge weight functions in Eq. (3) and Eq. (4), shown in Fig. 20(a) and (b) respectively, generate poor quality segmentations as the variance between two edge weights is not sufficient. The edge weight functions in Eq. (5-7), shown in Fig. 20(c-e) respectively, generate much better quality segmentations as they produce edge weights with high variance. Especially, the segmentations in Fig. 20(d) and (e) have the best quality as the edge weight functions used scale the output based on the height and width of the image, in turn scaling the variance based on the image size. Thus, as the variance between the edge weights increases, the quality of the image segmentation increases; and furthermore, the number of minimum (s,t) - cuts increases too as shown in Fig. 20.

C. Testing with different thresholds for maximum difference in luminance of two pixels

Setting the maximum luminance difference to ∞ results in the algorithm generating just a single minimum (s,t)-cut. The huge values of the edge weights resulting from the luminance differences as high as 255 introduce floating point precision issues, due to which the modified Dijkstra's

Fig. 19: Image segmentation (blue and yellow) for a 200 x 200 px image (left), a 100 x 100 px image (middle), and a 50 x 50 px image (right)

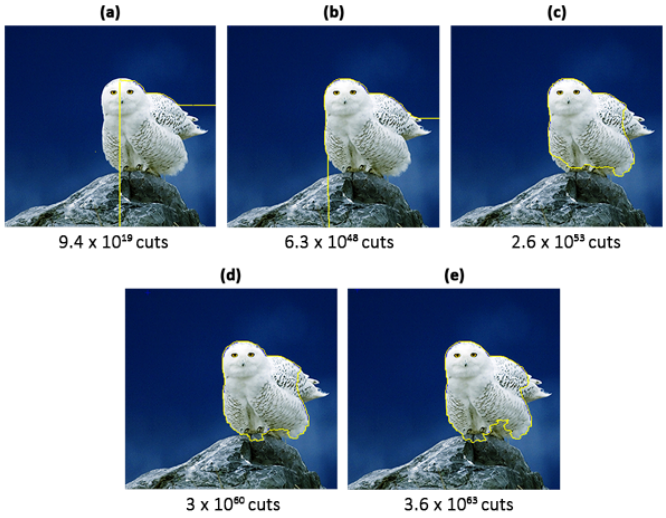

Fig. 20: Segmentations with minimum (s,t) - cuts for the image [11] using 5 different edge weight functions; (a) using Eq. (3), (b) using Eq. (4), (c) using Eq. (5), (d) using Eq. (6), and (e) using Eq. (7).

implementation (algorithm 2) fails to find more than one shortest path. This in turn results in a single minimum (s,t)-cut generated by algorithm 1; and thus to solve this issue the maximum difference between luminance of two pixels should be limited.

Different thresholds were tested using the image in Fig. 16. to determine the effect of the change of maximum luminance threshold on the count of minimum (s,t)-cuts and the quality of image segmentation. As shown in table III, the count of minimum (s,t)-cuts increases as the maximum $LumDiff$ decreases; except for $LumDiff = 5$ where the count of minimum (s,t)-cuts drastically drops to 10. The quality of image segmentation remains almost similar for the maximum $LumDiff >= 20$, as shown in Fig. 21., but the quality deteriorates as the maximum $LumDiff$ decreases from $15, 10$ to 5.

Thus, the threshold of 20 was chosen as it is the minimum threshold at which the quality of image segmentation is the best and the count of minimum (s,t)-cuts is the maximum.

Maximum $LumDiff$	No. of min cuts
∞	1
50	$1.58 * 10^{27}$
20	$9.47 * 10^{47}$
15	$6.18 * 10^{54}$
10	$4.53 * 10^{62}$
5	10

TABLE III: Maximum luminance difference vs. number of cuts

Fig. 21: Image segmentations for max $LumDiff >= 20$ (left), max $LumDiff = 15, 10$ (middle), and max $LumDiff = 5$ (right)

D. Scribbles vs single pixels for source and sink

Selecting multiple pixels (scribbles) for source and sink rather than selecting a single pixel each for source and sink is very helpful in generating a desired image segmentation. Scribbles enables customizing the image segmentation by drawing a rough scribble inside (source) and outside (sink) the desired object to be segmented, to get a better image segmentation as shown in Fig. 22.

Fig. 22: Image segmentation by selecting single pixel (left) vs. selecting multiple pixels - scribbles (right) for the source and the sink

E. Generating a single minimum (s,t)-cut vs. sampling multiple minimum (s,t)-cuts

The hypothesis of this capstone project states that generating a single minimum (s,t)-cut is sufficient for most of the images with almost identical minimum (s,t)-cuts, but for some images with a number of distinct minimum (s,t)-cuts it is necessary to count and sample through the minimum (s,t)-cuts. To find such images with distinct minimum (s,t)-cuts, the left-most and right-most minimum (s,t)-cuts were generated. The left-most minimum (s,t)-cut is the left-most shortest path (inverse of right-most shortest path explained in section III-C) between the first pair of vertices v_i, u_i that has

the minimum distance in step 5 of algorithm 1. The right-most minimum (s,t)-cut is the right-most shortest path between the last pair of vertices v_i, u_i that has the minimum distance in step 5 of algorithm 1.
All the minimum (s,t)-cuts lie between the left-most and the right-most minimum (s,t)-cuts, and thus these cuts are useful to determine if the image contains identical or distinct cuts. Fig. 23. shows two images with distinct left-most minimum (s,t)-cut (in yellow) and right-most minimum (s,t)-cut (in blue). Though, the two cuts are not significantly distinct, but these images prove that such images exist. Thus, counting and sampling through the minimum (s,t)-cuts is important.

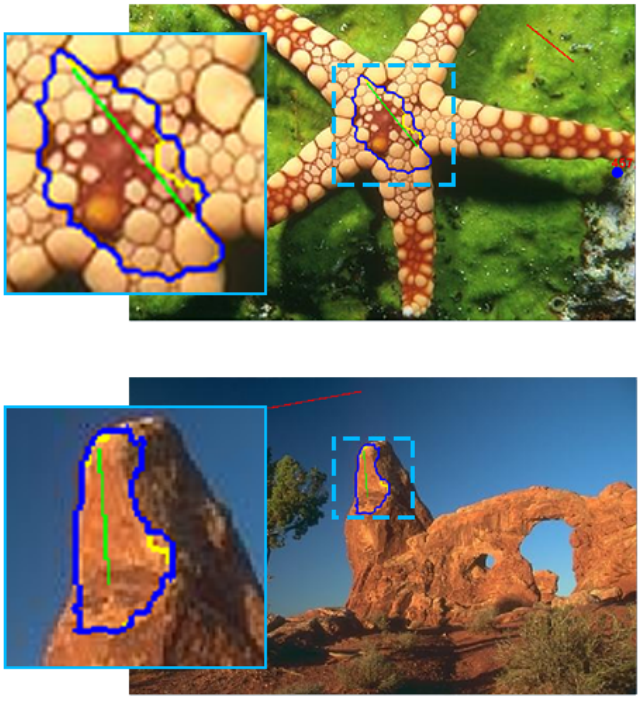

Fig. 23: Images with distinct left-most (yellow) and right-most (blue) minimum (s,t)-cuts

VI. Results

- The algorithm computes good quality image segmentations for images with both the object and background having small amount of color differences in it. The algorithm does not compute good quality image segmentations for images having similar colored object and background as seen in Fig. 18(h).
- If the object is very close to the border, the segmentation goes outside the border (cutting the image in half) rather than segmenting the object completely as seen in Fig. 18(d). This occurs because there exist edges in the planar graph that connects the vertex representing the infinite face with the vertices representing border faces. Giving large weights to such edges might solve this problem for some cases.
- The algorithm works very well for images of size up to 1000 x 1000 pixels; for images larger than 1000 x 1000 pixels, the Java implementation crashes as the program runs out of heap space.
- As the image size increases, the number of minimum (s,t)-cuts and the running time of the algorithm increases. The quality of the image segmentation decreases as the size of the image decreases for resolutions lower than 200 x 200.
- The count of minimum (s,t)-cuts increases as the maximum threshold $MaxDiff$ for the difference between luminance of two pixels $LumDiff$ decreases. The quality of the segmentations remains constant for $MaxDiff > 20$ and it decreases as the $MaxDiff$ starts decreasing below 20.
- Edge weight functions play a major role in defining the quality of the image segmentations. The edge weight functions that produce weights with a larger variance between them tend to generate better quality image segmentations.
- Selecting multiple pixels (scribbles) rather than selecting a single pixel each for source and sink is very helpful in generating a better desired image segmentation as it enables customizing the segmentation by drawing outlines inside the object and the background.
- Most of the images tested in the Berkeley dataset [12] had identical segmentations, but a very few images had distinct segmentations. Though this distinction between was not significant, it still proves that images with distinct segmentations do exist and thus, it is necessary to count and sample through the minimum (s,t)-cuts.

VII. Future scope

- Currently, the edge weights are calculated based on the differences between the gray-scale luminance between the pixels. Having edge weight functions that are based on the color differences rather than the gray-scale differences, might improve the quality of image segmentations.
- Testing the algorithm against many more images from various image segmentation datasets might yield more images with distinct minimum (s,t)-cuts, especially significant ones, which will help in strengthening the proof for the proposed hypothesis.
- Implementing the algorithm in C++ might lead to further decrease in running times of the algorithm as compared to the current Java implementation.

VIII. Conclusion

Image segmentation using Rachel Silva's algorithm [1] is a much simpler to understand and implement approach; which can be used to count and sample through the minimum (s,t)-cuts representing the segmentations in the image. This algorithm will be especially important in the segmentation of the images with multiple distinct minimum (s,t)-cuts, for which these cuts can be counted and probabilistically sampled to get a desired image segmentation.

REFERENCES

[1] R. E. Silva, "An alternative approach to counting minimum (s; t)-cuts in planar graphs," Master's thesis, Rochester Institute of Technology, Rochester, 2017.

[2] Y. Y. Boykov and M. P. Jolly, "Interactive graph cuts for optimal boundary and region segmentation of objects in n-d images," in *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*, vol. 1, 2001, pp. 105–112 vol.1.

[3] I. Bezáková and A. J. Friedlander, "Counting and sampling minimum (s,t)-cuts in weighted planar graphs in polynomial time," *Theoretical Computer Science*, vol. 417, pp. 2 – 11, 2012, mathematical Foundations of Computer Science (MFCS 2010). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397511003999

[4] M. O. Ball and J. S. Provan, "Calculating bounds on reachability and connectedness in stochastic networks," *Networks*, vol. 13, no. 2, pp. 253–278, 1983. [Online]. Available: http://dx.doi.org/10.1002/net.3230130210

[5] J. Wildstrom, "Lecture notes in math 681," November 2009. [Online]. Available: http://aleph.math.louisville.edu/teaching/2009FA-681/notes-091119.pdf

[6] G. Borradaile and P. Klein, "An o(n log n) algorithm for maximum st-flow in a directed planar graph," *J. ACM*, vol. 56, no. 2, pp. 9:1–9:30, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1502793.1502798

[7] G. P. Joshi and I. Bezáková, "Image segmentation using min (s,t)-cuts," Rochester Institute of Technology, Rochester, Tech. Rep., 2015.

[8] A. B. Gomsale and I. Bezáková, "Probabilistic image segmentation using min (s,t)-cuts," Rochester Institute of Technology, Rochester, Tech. Rep., 2016.

[9] A. Shah and I. Bezáková, "Evaluation of randomized minimum cut based image segmentation," Rochester Institute of Technology, Rochester, Tech. Rep., 2017.

[10] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Addison-Wesley Professional, 2011. [Online]. Available: https://algs4.cs.princeton.edu/24pq/

[11] F. Schulz, "Snowy owl." [Online]. Available: https://www.naturesbestphotography.com/competition_conservation.php

[12] D. Martin, C. Fowlkes, D. Tal, and J. Malik, "A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics," in *Proc. 8th Int'l Conf. Computer Vision*, vol. 2, July 2001, pp. 416–423.