Image segmentation using min (s,t)-cuts

By
Gaurav Pradip Joshi

Advisor
Dr. Ivona Bezáková
## Contents

1 **Introduction**  

2 **Background**  
   2.1 Planar graphs  
   2.2 Max flow  
   2.3 Algorithm  

3 **Implementation details**  
   3.1 Reading images  
   3.2 Representing planar graphs  
   3.3 Edge weight function  
   3.4 Finding max flow  
   3.5 Approaches for contracting SCC  
   3.6 Custom linked list  
   3.7 Test case design  
   3.8 Testing correctness with Euler’s formula  
   3.9 Sampling algorithm  

4 **Instructions for running the software**  
   4.1 Options available at run time  
   4.2 Options available at compile time  
   4.3 Sample commands  
   4.4 Troubleshooting  

5 **Experiments and Results**  
   5.1 Multiple segmentations  
   5.2 Image size  
   5.3 Epsilon change  
   5.4 Other experimentations with pre-processing  

6 **Future scope**  

7 **Conclusions**
1 Introduction

Min cuts are useful in computer vision when we want to segment an image represented as a grid graph. But there can be a number of min cuts in an image. Since segmentation is very subjective to a user it might be the case that the presented min cut may not be the “best cut” that user wants. Though finding all the possible cuts and providing the user with different options sounds interesting at first, it is not practical, as there can be exponentially many min cuts. But what can be done is sampling through these cuts and presenting the user with some of the samples.

Counting min cuts in a graph is a #P-Complete problem. But with planar graphs, it can be done in polynomial time, which is shown by Dr. Bezáková and Adam J. Friedlander [10]. The algorithm is $O(n \log n + n^d)$, where $d$ is path length between sink and source in a graph obtained after contracting all strongly connected components in the residual graph. This residual graph is the outcome of the initial run of the max flow algorithm. This run of the max flow algorithm contributes $O(n \log n)$ in the overall running time of the algorithm. For planar graphs as shown by ref. [11] max flow can be found out in $O(n \log n)$.

For the project, the questions of prime interest were, how different are the min cuts from each other? How many min cuts are there in an image? Can we use some pre-processing stage to improve the segmentation?

The report will focus on implementation details of this algorithm, experimentation, results of these experiments and finally the takeaway points. The report at some occasions refers to the code from the authors of ref. [12] as Schmidt segmentation or Schmidt’s segmentation, as they have named source code that way.

2 Background

2.1 Planar graphs

Planar graphs are simple graphs which can be drawn on a plane without any of its edges crossing [6]. Tree, wheel, grid graphs are some special cases of planar graphs. Graphs in Fig. 1 are planar, whereas the graph in the Fig. 2 is not.

Embedding of a planar graph or a plane graph is nothing but a specific drawing of the planar graph on the plane.

Dual of a planar graph is a graph created using vertices as faces of the planar graph, and then connecting vertices represented by adjacent faces. Dual of a graph can be different for different embeddings of the graph.
2.2 Max flow

The Max-flow min-cut theorem is used to find out “a” segmentation of an image. To do this, the image is represented as a grid graph. Each pixel of the image becomes a vertex, and bidirectional edges are added between neighboring vertices in horizontal and vertical direction. Diagonal edges (from the top left vertex to the bottom right vertex) can also be added, but it does not add any value. Each edge is given some weight according to the luminance difference between endpoints. The sink is selected by the user inside the object to be segmented out and the source is selected outside the object, (preferably close to the object), the min (s,t)-cut obtained is the boundary of the object. Of course, this is highly dependent on edge weight function that is being used.

2.3 Algorithm

Following are the steps of the algorithm:

1. Input the planar graph. See Fig.3.
2. Find out the max flow.
3. Find and contract all the strongly connected components in the residual graph. See Fig. 5
4. Duplicate the path between the sink and the source. See Fig. 5
5. Find out the dual of the graph. While finding dual do not add any edges between the faces opposite to the edges in the t-s path.
6. Count the number of paths between all pairs of vertices which represent opposite faces of an edge in a t-s path in the graph from step 4.
Figure 3: Step 1: Input the planar graph

Figure 4: Step 2: Find out the max flow

Figure 5: Step 3: Contract all the strongly connected components, Step 4: Duplicate a path between t-s, also find out all the faces
3 Implementation details

3.1 Reading images

libOpenImageIO was initially used for reading “jpeg” and “png” images. Installation of libOpenImageIO was a bit problematic for OS X. As after running “make install” it just compiles itself and does not copy the .lib files in desired “PATH” folder. On CS machines, libOpenImageIO is not installed. Hence, the code had to be modified to be able to compile without libOpenImageIO and follow a completely different flow. This also introduced a limitation of reading only “.ppm” files on CS machines. The code for reading and writing “.ppm” files has been used from Schmidt segmentation code.

3.2 Representing planar graphs

While representing planar graphs, it is important that its embedding is either stored or at least when needed it can be retrieved quickly. To do that, all the edges in the adjacency list of a particular vertex needs to be stored in either clockwise or counterclockwise order. Faces can be traced, by picking next edge in adjacency list of neighboring vertex till original vertex is reached again. Fig. 8 is a visual representation of how a face is traced. While finding this face, leftmost vertical edge (call it edge1) is
picked first. Topmost horizontal edge is next in adjacency list of the vertex at the end of edge1. So this edge is picked next. When the same vertex is reached with which face tracing was started, it means that the boundary of the face is completely traced. Similarly, all other faces can be found out. Each edge, at the max, can be in two faces (one containing it in the forward direction, other in the backward direction).

Figure 8: Face finding

3.3 Edge weight function

Edge weight function plays a critical role in finding out a good segmentation. If edges coming out of the source is not able to account for all the edge weights around the object boundary, proper segmentation is not achieved.

Following are some edge weight functions which were tried. They worked well for smaller images but as images grew in size the segmentation quality deteriorated. In all the following equations $diff = \text{luminance difference between the pixels}$, where luminance is calculated according to equation (1).

\[
luminance = 0.2126 \times \text{red} + 0.7152 \times \text{green} + 0.0722 \times \text{blue}
\]  

(1)

\[
EdgeWeight = 256 - diff
\]  

(2)

\[
EdgeWeight = 100000000 - (diff \times 1000)
\]  

(3)

\[
EdgeWeight = \frac{255}{\log(diff + 2)} + 1
\]  

(4)

\[
EdgeWeight = 10000000000 - diff^4
\]  

(5)

\[
EdgeWeight = (255 - diff)^8 + 1
\]  

(6)

\[
EdgeWeight = \frac{1}{diff^2}
\]  

(7)
Finally, edge weights are decided using function (9). This function is a slightly modified version of the function used in ref [12].

\[
\text{EdgeWeight} = \frac{1}{\text{diff} + 1} \times 1000^4 \times (x\text{resolution} \times y\text{resolution})
\] (8)

where \(\text{difference} = \text{difference between luminance values of pixels. In our case } \epsilon = 1.43e^{-39} (\text{for difference = 20}).\)

\(\epsilon\) value indicates the maximum luminance difference that is being considered while assigning edge weights. Due to the nature of the images in the real world, it is not needed to account for large differences. A threshold value can be chosen, after which all larger differences are ignored. This improves the running time of the code.

### 3.4 Finding max flow

For finding max flow, following approaches were followed.

1. Own implementation of the Ford-Fulkerson algorithm: The problem with Ford-Fulkerson is, since we are dealing with irrational numbers as edge weights and max flow, there is a possibility that the algorithm might never terminate.

2. Own implementation of the Edmonds-Karp algorithm: Edmonds-Karp works very well. But the only problem is, it does not scale very well in terms of running time for larger images.

3. Integrate C++ code written by authors of ref. [12]: It worked really well for smaller images. But this code introduces few floating point errors. As image size increases, the number of floating point errors introduced also grows. If the residual graph is used without checking for floating point errors, due to some erroneous edge weights, the next step of the algorithm detects the whole graph as one strongly connected component. On the other hand, if edge weights less than \(\epsilon\) value are ignored, the graph, no more satisfies the Euler’s formula.

Hence, the second approach was used while experimenting. The other approaches are still in the current version of the code and can be used by setting options specified in section 4.

### 3.5 Approaches for contracting SCC

Following are all the approaches which were implemented. The third approach is currently being used.

1. Find all SCCs and delete edges between all vertices one by one: This approach is very simple. But does not work as it messes up the edge ordering which is crucial for planar graphs. This leads to disturbing the embedding of the graph and we would not be able to find faces of the planar graph properly.

   Edge contraction is the way to keep edge ordering intact. It is indicated in Fig. 9

2. Find all SCCs, in a component to contract, pick first vertex, iterate through adjacency list of vertex 1, contract edge which has an endpoint in the same strongly connected component, merge lists and continue going through adjacency list. Do this for all SCCs: Though in first go the approach doesn’t seem like working, it actually works. As whenever we merge an adjacency list we scan through the list which was merged and then at the end come back to the initial list. This approach won’t miss
out on any edge and would be $O(n)$. The only problem with this approach is, it is complex. The edge which is supposed to be contracted needs to be deleted from adjacency list of both vertices and edge list for the graph. This yielded a lot of boundary conditions. Though they were implemented successfully, from the future perspective if anyone wants to use or debug this code it would have been a tough task.

3. Find all SCCs, then in a component to contract, pick the last vertex, iterate through all vertices in that, find the edge between two vertices, contract that edge. Do this for all SCCs: This approach initially does not seem like $O(n)$, because of “finding the edge part”. But here we are scanning adjacency list of the vertex which is not merged with any other vertex before, which can have at the max 8 edges in its adjacency list, making it a constant time operation. Hence, in total, this approach takes $O(n)$ time.

Here, to improve running time of the code we can ignore duplicate edges. As, if there are duplicate edges between two vertices, then it will be counted as a small face and dual graph will have more vertices. So we go through graph’s edges once again to remove duplicate edges if any. Note that, duplicate edges which are adjacent should only be removed. There can be a case when there are multiple edges between two vertices but they are not adjacent, in which case they should not be removed, as then Euler’s formula won’t be satisfied.

With all of the above approaches, it was also important to make sure merging lists is a constant time operation. For that custom doubly circular linked list was implemented. The details of which will follow in sub section 3.6.

It was also important to make sure that changing endpoints of the edges which have one endpoint in SCC and other outside of that, should be constant time. This was achieved by just changing the pointer at the location where “Edge” thinks it’s endpoints lie.

Fig.10 show how “verticesArray” for a graph looks like:

<table>
<thead>
<tr>
<th>Pointer to V1</th>
<th>Pointer to V2</th>
<th>Pointer to V3</th>
<th>Pointer to V4</th>
<th>Pointer to V5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 10: Initial array
Each edge stores information in terms of where its endpoints are. For e.g. edge (1,2) will store its endpoints as, vertex1ID = 1, vertex2ID = 2.
Now if suppose there is one more edge (2,3) which we want to contract and vertex 3 is going to take over vertex 2. Fig.11 shows how verticesArray will look.

<table>
<thead>
<tr>
<th>Pointer to V1</th>
<th>Pointer to V3</th>
<th>Pointer to V3</th>
<th>Pointer to V4</th>
<th>Pointer to V5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 11: Changed array

Now edge (1,2) still has vertex1ID=1 and vertex2ID=2 but when it is referenced in the array it will return vertex 3. This approach though memory consuming makes sure edge contraction is constant time.

The only disadvantage of this approach is space requirement.

3.6 Custom linked list

Implementation of node deletion in linked list in STD package according to reference [2] is O(n). Erase could not be used, as again it would have required traversing adjacency list of the other vertex to find out the position.

Plus most importantly, for this algorithm, to achieve desired running time doubly circular linked list was needed. The implementation of “list” in STD package is not circular [7]. Hence, a custom implementation of doubly circular linked list was needed.

3.7 Test case design

Figures 12, 13, and 14 are some example test cases that were used in initial development phase. A little modification to every test case will yield different cut count. Test case 3 was specifically designed to be used for part of the algorithm, after all the strongly connected components are contracted. The expected outputs are indicated in the caption of the images. For Fig. 12, the planar embedding is exactly same except for the edge (2,3), which should be drawn from the left side of vertex 0 to get proper planar embedding.

3.8 Testing correctness with Euler’s formula

After contracting strongly connected components the given graph still should be planar. Hence, it should still satisfy Euler’s formula for planar graphs, which is \(|V| - |E| + |F| = 2\). This evaluation helped understand errors in code. Specifically, that the graph extracted from Schmidt’s segmentation code can not be used with this algorithm as due to floating point errors the graph either has some erroneous edges or has some edges removed, which changes the results.

3.9 Sampling algorithm

After the count of min (s,t)-cuts is obtained, a min-cut can be sampled at random. This sampling is done by tracing a path in the dual graph, where a vertex which has more paths coming to it from source, is picked with a higher probability than a vertex having lesser number of paths coming to it. For each edge in this path, endpoints of corresponding non dual edges are picked. These vertices are then checked
to see if they belonged to any of the SCCs which were contracted. If they were part of an SCC then all other vertices in that SCC are added to the sample.

![Graph](image)

Figure 12: Min-cuts test case 1, min \((s,t)\)-cuts = 2
Figure 13: Min-cuts test case 2, min (s,t)-cuts = 21

Figure 14: Test case to test counting of paths and finding faces. No. of paths from 0 to 4 = 5
4 Instructions for running the software

The code has been designed to be as generic as possible. Certain options are available at the run time and for some options new binary should be compiled. Make file can be obtained using “gmakemake” utility on RIT CS machines.

4.1 Options available at run time

1. Specify source and sink (default is 0th row and 0th column for both sink and source).

2. Use custom weight function (only applicable for Schmidt, if using Edmonds-Karp this option is redundant).

3. Use Schmidt segmentation (default is Edmonds-Karp).

4.2 Options available at compile time

Flags in Constants.h are indicated on the right side.

1. Turn logs on: DEBUG_ON (currently OFF)

2. Use Edmonds-Karp: USE_EDMONDS_KARP (currently ON, Max flow uses BFS instead of DFS)

3. When sampling a min cut, turn on or off picking vertex using probability of occurrence (if off a vertex is picked totally at random): WEIGHTED_PROBABILITY (currently ON)

4. Use diagonal edges : TRY_DIAGONAL_EDGES (currently OFF)

5. Leave source unpainted : PAINT_SOURCE_RED (currently ON)

6. Write intermediate graph to file, available only for Schmidt segmentation: WRITE_INTERMEDIATE_TO_FILE (currently OFF)

4.3 Sample commands

./main ../Pokemon.ppm 0 0 93 273 75 248

Breaking down the command:

• ./main : Name of the program

• ../Pokemon.ppm: Path of the image. The image must be a ”.ppm” image if we want to run code on CS machines.

• 0 : Indicates do not use custom weight function. (Of course applicable only when using Schmidt segmentation.)

• 0 : Indicates use Edmonds-Karp. (1 for Schmidt).

• 93 : Sink row.

• 273 : Sink column.

• 75 : Source row.

• 248 : Source column.
4.4 Troubleshooting

- If the program crashes giving a “segmentation fault”, it is most probably because enough stack space is not allocated. Try fixing it by using command:

  \[\texttt{ulimit -s unlimited}\]

- If the program does not compile after fetching the code from git, make sure all the files including the ones for Schmidt’s code [12] are in the same folder.

5 Experiments and Results

5.1 Multiple segmentations

Fig.15 shows the original image used to demonstrate how we can obtain multiple segmentations for an image. The image is of size 455x468. Number of samples taken were 10. Fig.16, 17, 18 are three random samples from those 10. The segmented part is zoomed in.

Figure 15: Original image with source(red) and sink(blue) Image reference: [8]

Figure 16: Sample 1

Figure 17: Sample 2

Figure 18: Sample 3
5.2 Image size

To study the effect of compression, a random image of size 1280x532 was compressed to different sizes and the code was tested on them. These images were obtained by using “Preview” utility on OS X, which has a feature called “Adjust Size”. Source and sink were roughly kept in the same area. Table 1 shows the observations. Going from smaller size to larger size, it was observed that after a point segmentations did not improve visually. But, due to increase in the number of vertices and edges in the graph, running time increased. The image in Fig. 19 was used to obtain this plot. Similar behavior is observed for the image in Fig. 24.

Figure 19: Image used to study min (s,t)-cut count change with change in image size, image ref: [9]

<table>
<thead>
<tr>
<th>Image size</th>
<th>Sinkrow</th>
<th>SinkColumn</th>
<th>TotalEdges</th>
<th>TotalVertices</th>
<th>Cuts</th>
</tr>
</thead>
<tbody>
<tr>
<td>60x25</td>
<td>5</td>
<td>22</td>
<td>2915</td>
<td>1500</td>
<td>1</td>
</tr>
<tr>
<td>256x106</td>
<td>21</td>
<td>94</td>
<td>53910</td>
<td>27136</td>
<td>1.59e+10</td>
</tr>
<tr>
<td>320x133</td>
<td>28</td>
<td>117</td>
<td>84667</td>
<td>42560</td>
<td>2.00e+11</td>
</tr>
<tr>
<td>426x177</td>
<td>37</td>
<td>156</td>
<td>150201</td>
<td>75402</td>
<td>5.63e+16</td>
</tr>
<tr>
<td>640x266</td>
<td>55</td>
<td>234</td>
<td>339574</td>
<td>170240</td>
<td>1.80e+20</td>
</tr>
<tr>
<td>853x355</td>
<td>75</td>
<td>311</td>
<td>604422</td>
<td>302815</td>
<td>2.61e+28</td>
</tr>
<tr>
<td>1280x532</td>
<td>112</td>
<td>468</td>
<td>1360108</td>
<td>680960</td>
<td>1.95e+38</td>
</tr>
</tbody>
</table>

Table 1: Impact of increasing image size with same source and sink on number of cuts keeping the $\epsilon$ value constant
5.3  Epsilon change

The image in Fig. 24 was used to study the effect of change of $\epsilon$ value. The image is 512x512. Source and sink were kept roughly in the same area. This experiment was also ran on benchmark images. Fig. 22, 23 shows results for one of the benchmark images [1].

<table>
<thead>
<tr>
<th>Max Diff (luminance)</th>
<th>$\epsilon$</th>
<th>Running time</th>
<th>Cuts</th>
<th>Max flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1.43E-39</td>
<td>233 sec</td>
<td>4.90E+82</td>
<td>3.83e^{-12}</td>
</tr>
<tr>
<td>15</td>
<td>5.88E-26</td>
<td>145 sec</td>
<td>5.24E+102</td>
<td>3.83e^{-12}</td>
</tr>
<tr>
<td>12</td>
<td>8.84E-19</td>
<td>115 sec</td>
<td>1.49E+130</td>
<td>3.83e^{-12}</td>
</tr>
<tr>
<td>11</td>
<td>1.43E-16</td>
<td>106 sec</td>
<td>2.89E+135</td>
<td>3.96e^{-12}</td>
</tr>
<tr>
<td>10</td>
<td>1.64E-14</td>
<td>85 sec</td>
<td>1.57E+135</td>
<td>1.99e^{-11}</td>
</tr>
</tbody>
</table>

Table 2: Impact of changing $\epsilon$ value on number of cuts and running time of the code.
Figure 21: Impact of choice of $\epsilon$ on greyscale image of size 512x512

Figure 22: $\epsilon = 1.43e^{-14}$, time=25sec, Image reference [1]

Figure 23: $\epsilon = 1.43e^{-39}$, time=77sec, cuts=9.09835e64, Image reference [1]
Figure 24: Change in segmentation with the change in $\epsilon$. Decreasing $\epsilon$ increases largest difference we are accounting for, Image reference: [5]

5.4 Other experimentations with pre-processing

Some other pre-processing techniques were tried on the images mentioned in previous sections, keeping $\epsilon = 1.43e^{-39}$. Techniques like applying histogram equalization [3], mean, median filter [4] etc. reduces the min (s,t)-cut count as they help eliminate noise. As the number of min (s,t)-cuts decreased, running time improved. Negating and blurring the image did not yield a good segmentation.

6 Future scope

- Schmidt’s segmentation code is licensed under LGPL. Which means it should not be modified. Plus, the planar graph obtained after calculating max flow, has a few “bad” edges which affect the results. Hence, own implementation of $O(n \log n)$ algorithm for max flow is needed. It will also help to process the higher resolution images faster. The main concern that should be kept in mind while implementing it, is to find out ways by which floating point errors can be minimized.

- Averaging over all the min (s,t)-cuts, can be an interesting result to observe. This averaging will obviously dependent on $\epsilon$ value chosen. Different averages are possible with different $\epsilon$ values.

- Testing the code on benchmark dataset images will yield more insights.

7 Conclusions

Following conclusions are based on the experiments ran on the images and test cases mentioned in previous sections.

- Counting cuts algorithm can be used to find out multiple segmentations of an object in an image.
• All the min (s,t)-cuts are visually similar to each other.

• There can be an exponential number of min (s,t)-cuts in an image, though visually similar.

• Depending on the image, a pre-processing step may improve running time, as it removes noise which gives lesser min (s,t)-cut count.

• Min (s,t)-cut count grows exponentially with image size. Hence, compression can be used to improve running time and still get a decent segmentation.

• Edge weight function has a huge impact on the segmentation. Weight function should be independent of image size and should be some positive power of inverse exponential of the difference between pixel luminance.

• $\epsilon$ value chosen, makes a huge impact on the final outcome. For a faster running time, larger values should be chosen. But this means we might lose on finer segmentation, as larger $\epsilon$ values mean accounting for only a small difference between luminance values of neighboring pixels. It’s essentially a trade-off between finer segmentation vs faster running time.

References


