Image segmentation via graph cuts with connectivity priors

Celline Butuem Soares
Rochester Institute of Technology
cxb6747@rit.edu

ABSTRACT

Graph cuts can be applied to many different ends and one possible application is in Computer Vision, where they can be employed in the extraction of an object from its background within an image. Most image segmentation methods can present fragmented results when the object have thin elongated structure, such as blood vessels in a medical image or the branches of a tree. In this project we implemented the first part of the *Minimum Planar Multi-Sink Cuts with Connectivity Priors* [2] in image segmentation. This is an interactive method where the user selects a pixel in the image background and several pixels inside the object to be extracted. This method imposes connectivity conditions between pixels inside the object and finds a minimum-cut that respects these settings.

1. INTRODUCTION

There are many approaches to image segmentation, such as color intensity threshold, edge detection, watershed and graph-cut segmentation [9]. Most of these methods can present fragmented results when dealing with thin, elongated structures [8, 3] and one way of dealing with this problem is to impose connectivity conditions between pixels inside the object when performing the segmentation.

The purpose of this project is to implement the first part of the graph-cut with connectivity priors algorithm presented in [2], which is the *Minimum Planar Multi-Sink Cuts with Connectivity Priors* method, and apply this algorithm to automated image segmentation and experiment with different edge weight functions aiming a better segmentation accuracy.

Related work includes a research of another graph cut method that also uses connectivity priors in order to avoid *"shrinking bias"*, it uses an algorithm inspired by Dijkstra's algorithm, but it is merged with graph cuts. Another research was done in the evaluation of the segmentation [10] and it provides different ways to better evaluate segmentation methods, which can be useful once all of the parts of the implementation of the method in [2] are finished and merged together.

1.1 Background

The first step in the graph-cut segmentation methods is to construct a graph that represents the image. In this graph, each pixel of the image is represented by a vertex and edges represent the connection between neighboring pixels The weight of the edges is calculated by a function of the color similarity between the pixels. Figures 1 and 2 represent the image and its corresponding grid graph.

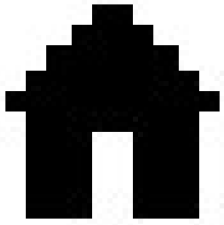

Figure 1: Original image.

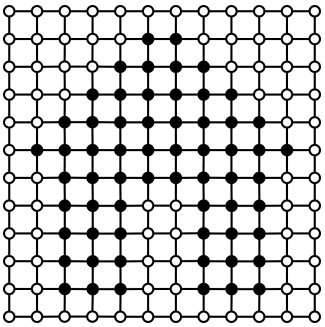

Figure 2: Grid graph constructed from Fig.1.

The goal of interactive image segmentation is to extract an object from its background using input from the user [8]. For this implementation the user chooses a pixel in the background (source) and a multiple pixels inside the object

(sinks). Figure 3 illustrates the selected source (s) and sinks (t_1, t_2 and t_3).

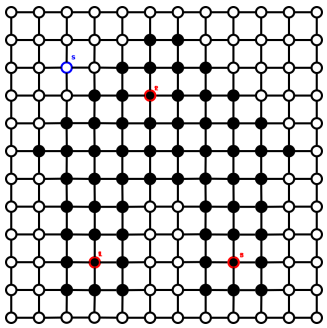

Figure 3: Grid graph constructed from Fig. 1. Selected source (s) highlighted in blue and selected sinks (t_1, t_2 and t_3) highlighted in red.

The constructed graph is a edge-weighted planar undirected grid graph. The weight of the edges are calculated by a edge-weight function of the color difference between the neighboring pixels. This weight function, $w(l_1, l_2)$, where l_1 and l_2 represent the luminance value of each pixel whose vertices are connected by the edge in question, can be a variety of functions in which similar luminance values result in higher weight, and different luminance values result in lower weight. Equations 1 and 2 are a few examples of such functions. The luminance value for each pixel was calculated using Equation 3, from [7].

$$w(l_1, l_2) = e^{\frac{1}{|l1-l2|}} \tag{1}$$

and

$$w(l_1, l_2) = \frac{1}{|l_1 - l_2|}, \tag{2}$$

where l_1 and l_2 are the luminance values of pixel 1 and 2.

$$l_i = 0.2125r + 0.7154g + 0.0721b, \tag{3}$$

where l_i is the luminance value of a pixel i in a RGB image and r, g and b are the intensity of the red, green and blue channels.

Most of the data processing of the implemented method takes place in the *Dual* of the original planar graph. The dual (H) of a planar graph (G) has one vertex for each face in G and edges connect vertices in H corresponding to neighboring faces in G. The weight of the edges in H are the same as the edge which is of the common boundary of the faces in G. For instance, if the faces f_1 and f_2 in g have in common the edge e with weight w_e, the corresponding vertices in H, f_1^* and f_2^*, are connected by the edge e^* with weight w_e [1]. Figure 1.1 illustrates a graph and its dual.

1.2 Minimum planar multi-sink cuts with connectivity priors algorithm

The method implemented in the scope of this project is the first part of the algorithm *Minimum planar multi-sink cuts with connectivity priors* presented in [2].

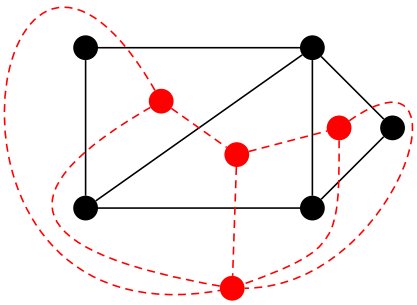

Figure 4: Planar graph in black and its corresponding dual graph in red.

This method operates mostly in the dual (H) of the original planar graph (G). In order to maintain the information about the location of of the original graph's source (s) and sinks ($T = t_1, t_2, ...$) in the dual graph, before constructing H, for each s and $t_i \in T$, we create a small face in G with ∞-weight edges. Now, constructing H, there exists vertices s^* and $T^* = t_1^*, t_2^*, ...$ representing s and T.

In H we compute the shortest path for each pair of sinks (t_i^*, t_j^*), t_i^* and $t_j^* \in T^*$ and $i \neq j$, which can be done using Dijkstra's algorithm [5] or any other shortest path algorithm.

After computing all shortest paths between sinks, we combine the paths and form a region τ, which we cut along so that a separating cycle in H (corresponding to a cut in G), will not cross τ, fragmenting the result.

Subsequently, we need to find the region containing s but not any vertex from τ. This can be computed using s search algorithm, such as breadth first search of depth first search.

2. METHODOLOGY

There were two main parts to this project: the first part of the algorithm from [2] and applying this algorithm to image segmentation. Each of these parts were then subdivided into smaller steps as follows:

1. Read the image to be segmented;
2. Obtain the chosen pixels in the background (source s) and inside of the object (sinks t_1, t_2...) from user input;
3. Construct the graph G from the image;
4. Construct the dual graph H from G;
5. In H, compute the shortest path between each pair (t_i^*, t_j^*) of sinks;
6. Combine the shortest paths to obtain the region τ;

7. Obtain a set (V^*) of all of the vertices reachable from s^* and that are not in τ;
8. Find the faces in G that correspond to the vertices in V^*;
9. Retrieve the vertices from the faces found in the previous step;
10. Remove from the image the pixels correspondent to the vertices retrieved in the previous step;
11. Display the resulting image.

Figures 5, 6, 7, 8, 9 and 10 illustrate steps $2-3$, 4, 5, 7, 9 and $10-11$, respectively.

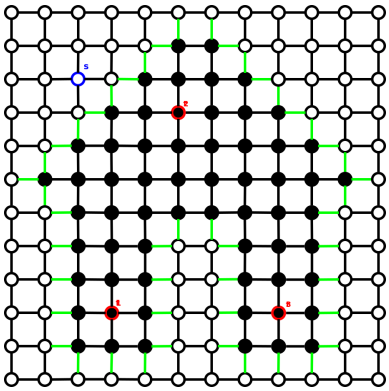

Figure 5: Obtain s and $T = \{t_1, t_2...\}$ and construct the graph G from the image (Steps 2 and 3). Lower weight edges highlighted in green.

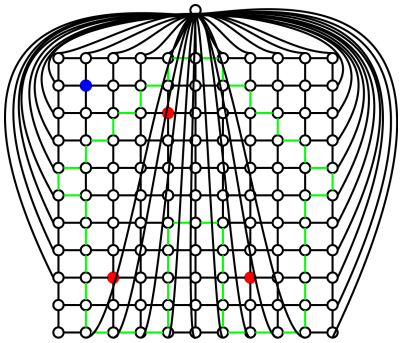

Figure 6: Construct the dual graph H from G (Step 4).

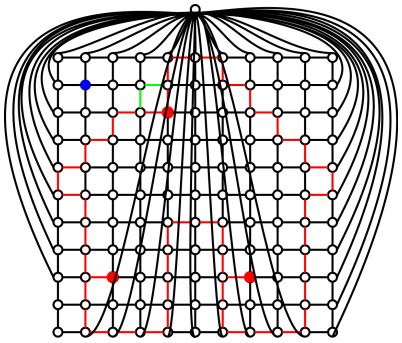

Figure 7: In H, compute the shortest path between each pair (t_i^*, t_j^*) of sinks (Step 5). Shortest paths highlighted in red

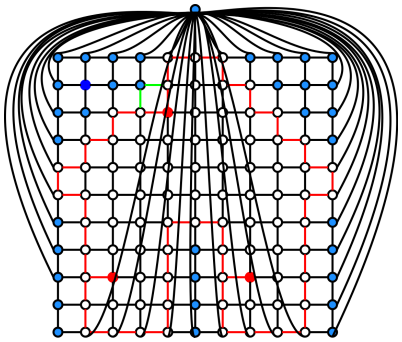

Figure 8: Obtain a set (V^*) of all of the vertices (in light blue) reachable from s^* and that are not in τ (Step 7).

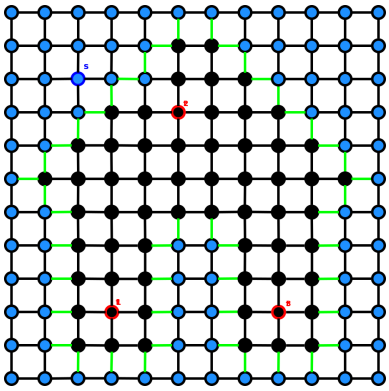

Figure 9: Retrieve the vertices (in light blue) from the faces found in the step 8 (Step 9).

2.1 Implementation details

The programming language used was $C++$, and *OpenCV* library [4] was used to process the images. The program consists of two files: `Graph.h` and `segmentation.cpp`. The header file contains the the class `Graph` definitions and `segmentation.cpp` contains the implementation of all functions used in the segmentation and also the `main` function.

`Graph` class has the following methods:

- `void addVertex(int newVert)`: adds a vertex to the graph structure.
- `void addSink(int newSink)`: adds a vertex to the list of sinks.
- `void addSource(int gS)`: add a vertex as source of the graph.
- `void addEdge(int newEdge[2], double w)`: adds a new edge with weight w to the graph structure.
- `bool isEdge(int v, int u)`: verifies if there is an edge between vertices v and u.

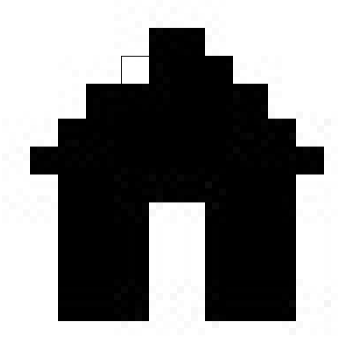

Figure 10: Retrieve the resulting image. (Step 11).

- `double getEW(int v, int u)`: returns the weight value of the edge between vertices v and u.
- `void findFaces()`: search the graph for faces and creates a list of faces.
- `void createDual()`: creates the dual graph of an existing graph.

The functions outside of class `Graph` there are the following functions:

- `list<int> shortestPath(Graph g, int s, int t)`: returns the shortest path between vertices s and t in the graph g.
- `vector<int> runDFS(Graph d, int u, vector<int> s, set<int> paths)`: returns a vector of vertices of g that are reachable from u, but do not contain the vertices in `paths`.
- `list<int> mergeList(list<int> l1, list<int> l2)`: returns a merged list of vertices.
- `list<int> findRegion(Graph g, Graph dual)`: this function calls the `shortestPath`, `mergeList` and `runDFS` functions, and returns a list with all of the vertices in the dual graph corresponding to the faces in g and the background of the image.
- `double compWeight(int func, double l1, double l2)`: receives the edge weight function option and returns the weight value calculated by the chosen function.
- `Graph getGraph(Mat image, vector<Point> pts)`: given an image and a vector of points in the background and foreground of the image, creates and returns a graph, adding all of the vertices and edges to it.
- `Point getPoint(int p, int cols)`: receives a vertex index and and returns the position of the corresponding pixel.
- `vector<Point> background(list<int> reg, Graph g, Graph dual, int cols)`: returns a vector of the points that are in the background of the image.
- `Mat getObj(Mat im, vector<Point>out)`: returns the image with the points in the background in gray.
- `void CallBackFunc(int event, int x, int y, int flags, void* param)`: mouse callback function fro the GUI. Currently not in use.
- `int main(int argc, char** argv )`: main function of the program, takes the image file name as command line argument.

The implemented graph representation structure was *Adjacency List*, since the number of vertices in the graph is large and an *Adjacency Matrix* would take up too much space.

A GUI is not yet implemented for obtaining the user input, but a mouse callback function is implemented. The Points for the source and sinks of the image are hard-coded for testing purposes, but can easily be altered to accept command line arguments.

The selected edge weight function is also hard-coded and can be changed in the function `getGraph` at the variable `func`.

The following list are the steps to segment an image using the current program:

3. EXPERIMENTAL RESULTS

Before using the program to segment larger images, a few tests were made to make sure that there were no errors in the creation of the graphs or calculation of the edge-weight. The images used in these tests can be seen in Figures 11 and 12.

Figure 11: Image used to perform tests with the program (left) and resulting segmentation (right). This segmentation was obtained with the edge-weight function in Equation 2

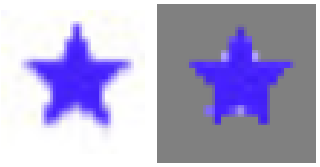

Figure 12: Image used to perform tests with the program (left) and resulting segmentation (right). This segmentation was obtained with the edge-weight function in Equation 1

Original images and segmentation results presented here were cropped in order to better obtain and show the segmentation. Figures 14, 16, 18, 20 and 22 are from [6]. Figures 14, 15, 17, 19, 21, 23 and 24 are the results after the segmentation.

Figure 13: Original image.

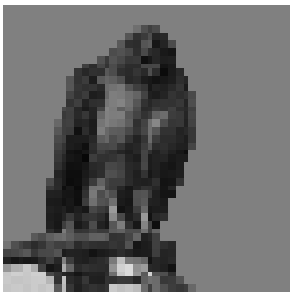

Figure 14: Segmentation of the image in Figure 13. This segmentation was obtained with the edge-weight function in Equation 2 and 6 different selected sinks.

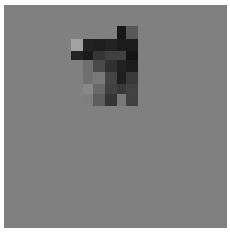

Figure 15: Segmentation of the image in Figure 13. This segmentation was obtained with the edge-weight function in Equation 2 and 2 selected sinks.

Figure 16: Original image.

Figure 17: Segmentation of the image in Figure 16. This segmentation was obtained with the edge-weight function in Equation 2 and 6 different selected sinks.

Figure 18: Original image.

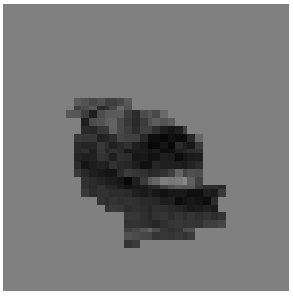

Figure 19: Segmentation of the image in Figure 18. This segmentation was obtained with the edge-weight function in Equation 2 and 5 different selected sinks.

4. DISCUSSION

The resulting images presented satisfactory segmentation, and even though it is not the final segmentation. From

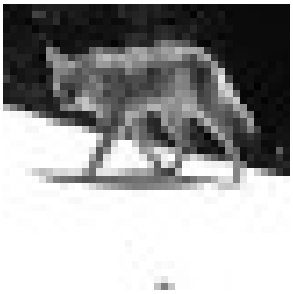

Figure 20: Original image.

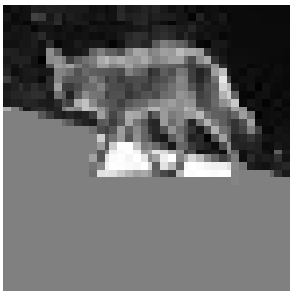

Figure 21: Segmentation of the image in Figure 20. This segmentation was obtained with the edge-weight function in Equation 2 and 6 different selected sinks.

Figure 22: Original image.

the results in Figure 15 and Figure 14, it is clear that the amount of pixels selected as sinks in the graph can make a lot of difference. Selecting more sinks can even make up for the fact that the used weight function was not the one that presented better separation between of values for similar colors and different colors, which can mean that this method of image segmentation can also be used in cases where it is difficult to identity the object within the image because the luminance values are too similar.

For the image in Figure 20, the method could not efficiently segment the wolf from its background. Images in Figures 24 and 14 presented the best segmentation among all of the tests.

5. FUTURE WORK

Figure 23: Segmentation of the image in Figure 22, but with lower resolution. This segmentation was obtained with the edge-weight function in Equation 2 and 6 different selected sinks.

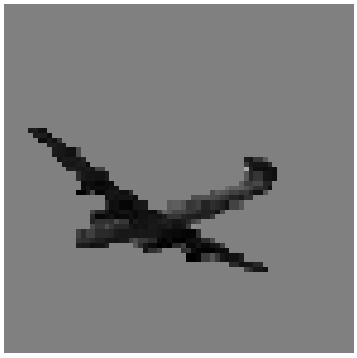

Figure 24: Segmentation of the image in Figure 22. This segmentation was obtained with the edge-weight function in Equation 2 and 6 different selected sinks.

Future work includes the implementation of the second part of the algorithm from [2] and integration between both parts. Also includes experimenting with a variety of edge weight functions to increase segmentation accuracy, and heuristically decrease the running time by testing different pre-processing steps.

6. CONCLUSIONS

It is not possible to completely evaluate the results of this project since it is not the integral algorithm implementation, however, it was clear in the resulting images that the connectivity conditions were essential in obtaining an object that is not fragmented.

Nonetheless, the implementation of the first part of the algorithm presented good results for all of the tested images, when a sufficient amount of pixels inside the object were selected, therefore we expect that the complete implementation will improve the quality of the segmentation.

7. ACKNOWLEDGMENTS

The author would like to greatly thank the Brazilian mobility program *Science Without Borders* and *Coordination for*

the Improvement of Higher Education Personnel - CAPES (Brazil) for the opportunity of completing this graduate study and for the financial support.

8. REFERENCES

[1] R. Balakrishnan and K. Ranganathan. *A Textbook of Graph Theory*. Springer-Verlag New York, Inc., New York, NY, USA, 2nd edition, 2012.

[2] I. Bezáková and Z. Langley. *Mathematical Foundations of Computer Science 2014: 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part II*, chapter Minimum Planar Multi-sink Cuts with Connectivity Priors, pages 94–105. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[3] Y. Boykov and O. Veksler. *Handbook of Mathematical Models in Computer Vision*, chapter Graph Cuts in Vision and Graphics: Theories and Applications, pages 79–96. Springer US, Boston, MA, 2006.

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

[5] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, Dec. 1959.

[6] 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*, volume 2, pages 416–423, July 2001.

[7] R. Szeliski. *Computer Vision: Algorithms and Applications*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[8] S. Vicente, V. Kolmogorov, and C. Rother. Graph cut based image segmentation with connectivity priors. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8, June 2008.

[9] F. Yi and I. Moon. Image segmentation: A survey of graph-cut methods. In *Systems and Informatics (ICSAI), 2012 International Conference on*, pages 1936–1941, May 2012.

[10] Y. J. Zhang. A survey on evaluation methods for image segmentation. *Pattern Recognition*, 29(8):1335–1346, Aug. 1996.