Intelligent Scissors with Efficient Planar Graph Cuts

Andrew Farmer
University of Kansas
Lawrence, KS 66045
afarmer@eecs.ku.edu

Abstract

Graph Cuts is a technique that is well suited for making binary classification decisions. These types of decisions are common in computer vision. Examples include image segmentation, shape matching, and denoising. Most implementations solve the equivalent problem of Maximum Flow, but traditional generalized Maximum Flow algorithms, while having polynomial complexity, are far too slow for computer vision applications. Thankfully, we can take advantage of the planar nature of an image, and apply specialized algorithms that perform much faster. In this paper, we present a brief overview of the history of Graph Cuts, then describe and implement the current state of the art algorithm for planar Graph Cuts, and present experimental performance results for a basic implementation of Intelligent Scissors using our algorithm.

1. Introduction

Graph Cuts have been applied to computer vision problems for over 20 years. The first application was by Greig *et al*. to smooth noisy images [5]. In general, any problem that can be formulated in terms of energy minimization can be solved using Graph Cuts. Binary classification problems are solved exactly, while problems with multi-valued solution domains can be solved approximately by decomposing them into a series of binary decisions and iteratively solving each one. In practice, these approximate solutions are near the global optimum.

1.1. Intelligent Scissors

The basic idea behind intelligent scissors is to present the user with an image, allow him to select an object in that image, and automatically find the borders of that object. This application was pioneered by Mortensen and Barrett [6] and fundamentally is an energy minimization problem.

In Mortensen and Barrett's paper, the energy function was the weighted sum of three edge features: the Laplacian zero-crossing, gradient magnitude, and gradient direction. This can be formulated as:

$$l(p,q) = \omega_Z \cdot f_Z(q) + \omega_G \cdot f_G(q) + \omega_D \cdot f_D(p,q)$$

where $l(p,q)$ is the local cost of the edge between two pixel-nodes p and q and ω is a vector of weights to give each feature.

The Laplacian feature f_Z approximates the second derivative of the image, and is zero where the gradient of the intensity is maximal. This is good for edge detection, as it doesn't penalize weak edges. If $I_L(p)$ is the Laplacian of pixel p in image I, then f_Z is given by:

$$f_Z(p) = \begin{cases} 0 & \text{if } I_L(p) = 0, \\ 1 & \text{if } I_L(p) \neq 0 \end{cases}$$

In practice, the discrete sampling of an image means that rarely is a pixel p actually on the zero-crossing. To compensate for this, we usually implement $f_Z(p)$ such that the value is 0 if the Laplacian value at p and its neighbor q have opposing signs, indicating a zero crossing between them.

The gradient magnitude feature is computed with:

$$G = \sqrt{I_x^2 + I_y^2}$$

where I_x is the discrete horizontal derivative and I_y is the discrete vertical derivative. This value is then scaled and inverted, such that high gradient values between pixels yield low cost edges, and vice versa:

$$f_G(p) = 1 - \frac{G}{\max(G)}$$

Finally, the gradient direction feature is computed by:

$$f_D(p,q) = \frac{2}{3\pi} \left\{ \arccos[d_p(p,q)] + \arccos[d_q(p,q)] \right\}$$

$$d_p(p,q) = D(p) \cdot L(p,q)$$
$$d_q(p,q) = L(p,q) \cdot D(q)$$

$$L(p,q) = \begin{cases} q-p & \text{if } D(p)\cdot(q-p) \geq 0, \\ p-q & \text{if } D(p)\cdot(q-p) < 0 \end{cases}$$

where $D(p)$ is a unit vector perpendicular to the gradient direction at p. This can often be given by:

$$D(p) = (I_y(p), -I_x(p))$$

In essence, f_D assigns a high cost to edges between pixels whose gradient directions are similar but perpendicular to the link direction and a low cost to edges between pixels whose gradient directions are similar and parallel to the link direction.

1.2. Graph Cuts

Graph Cuts is the the common name for finding the minimum cut in a network. Two nodes are designated as the source and sink (s and t respectively). These nodes are known to be in separate classes, and the task to is assign all nodes to either the class of s or the class of t. If the edge costs in the graph represent the energy required to place neighboring nodes in different classes, then the minimization of this energy function can be found by finding the minimum cut (the minimum-cost way to partition the graph such that there is no path from s to t).

In practice, the minimum cut problem is solved by implementing a maximum flow algorithm. In fact, the insight that these two problems are equivalent [4] provided the first known polynomial time algorithms for minimum cuts.

The best maximum flow algorithms for general graphs have running times that are dominated by the number of nodes. Unfortunately, most computer vision problems are naturally formulated such that each pixel becomes a node, connected to its neighboring pixel-nodes in a pairwise fashion. In practical applications with real images, the nodes often number in the millions, meaning runtime is prohibitively slow.

As an example, the Relabel-To-Front algorithm, while easy to understand and implement, runs in $O(n^3)$ time. Given a graph with 10^6 nodes, the worst case execution involves 10^{18} discharge operations. Put another way, doubling each dimension of an image (quadrupling the number of pixels) multiplies the running time by a factor of 64!

The current asymptotically fastest algorithm for general graphs is based on blocking flows, and runs in $O\left(\min(n^{2/3}, m^{1/2}) m \lg \frac{n^2}{m+2} \lg C\right)$ where C is the maximum flow over a single edge in the graph [3].

However, as Table 1 shows, restricting the problem to finding maximum flows in graphs with a planar embedding allows for vast improvements to complexity. This is useful, because representing an image as a pairwise-connected Markov Random Field results in a planar graph.

Year	Restriction	Complexity
1956	st-planar	$O(n^2)$
1979	st-planar	$O(n \lg n)$
1982	flow limited	$O(n\sqrt{n} \lg n)$
1983	flow limited, undirected	$O(n \lg^2 n)$
1985	undirected	$O(n \lg^2 n)$
1987	st-planar	$O(n\sqrt{\lg n})$
1997	undirected	$O(n \lg n)$
2001		$O(n \lg^3 n \lg C)$
2008		$O(n \lg n)$

Table 1. History of Maximum Flow algorithms for planar graphs with restrictions to applicability [1].

2. Efficient Planar Graph Cuts

The first algorithm claiming to solve the maximum flow problem for directed planar graphs in $O(n \lg n)$ time was by Weihe in 1997 [9]. However, the algorithm was complicated and, while running in $O(n \lg n)$, depended on a $O(n^2)$ initialization step. This problem was overlooked by Weihe and remains unresolved, preventing effective implementation.

The original st-planar maximum flow algorithm introduced by Ford and Fulkerson in 1956 ran in $O(n^2)$ time and depended on saturating the uppermost residual path at each step, where the notion of uppermost is intuitively the residual path which has no other residual paths above it when the graph is viewed with the source on the left and the sink on the right. This notion only makes sense in the context of st-planar graphs, where both the source and sink are adjacent to the infinite face.

The algorithm that is the subject of this paper [2] generalizes the notion of 'uppermost', using more modern notions of 'left' and 'right' in a planar graph. Given two edges x and w leaving vertex v, x is left of w if it is between y and w when examining edges clockwise from y to w, where y is the edge entering v from v's parent u. The definition of right is defined likewise, but scanning counter-clockwise.

With this generalization, the algorithm always saturates the leftmost path from the source to the sink, removing the limitation to st-planar graphs. By using a dynamic tree[1] data structure [8], the running time is improved to $O(n \lg n)$.

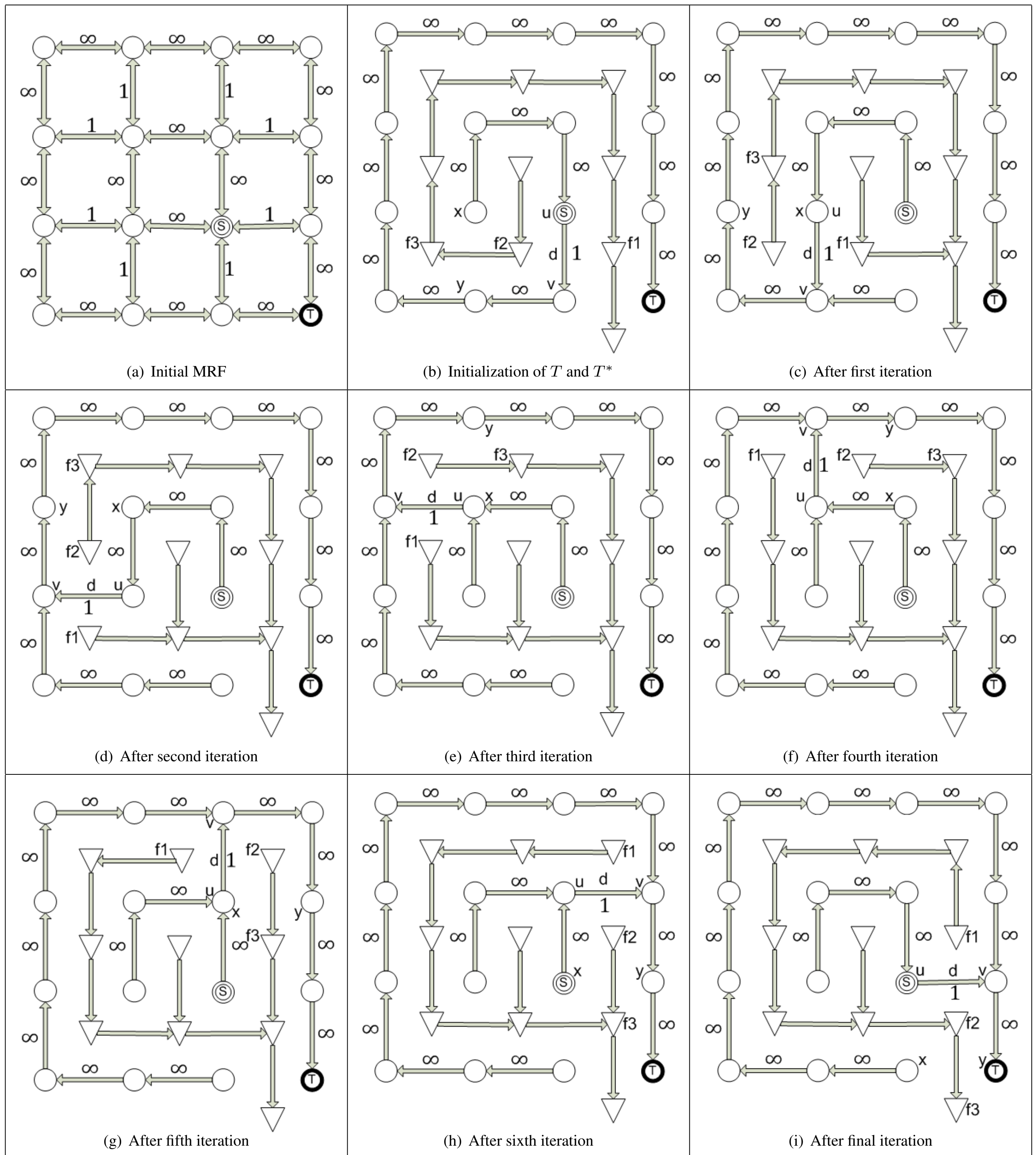
2.1. The Algorithm

The pseudocode for Borradaile and Klein's algorithm can be seen in Table 2.

In her Ph.D. thesis, Borradaile proves that the loop runs at most m times, and m is at most $3n$ in a planar graph. Each operation inside the loop requires at most $O(\lg n)$ time, yielding a total running time of $O(n \lg n)$ [1].

Figure 1 provides a walk-through on a small graph,

[1] also known as a link/cut tree



(a) Initial MRF

(b) Initialization of T and T^*

(c) After first iteration

(d) After second iteration

(e) After third iteration

(f) After fourth iteration

(g) After fifth iteration

(h) After sixth iteration

(i) After final iteration

Figure 1. A walk-through of Borradaile and Klein's algorithm on a simple flow network. The right-first search tree T is represented by circular nodes, while the planar dual T^* is represented by triangular nodes. At each step, the nodes u, v, x, and y and faces f_1, f_2, and f_3 are marked. Notice that in the last step, after the path from s to t is saturated, f_1 is found to be a descendent of f_2, and the algorithm terminates. Connecting f_2 to f_1 creates a cycle in the planar dual, representing the minimum cut.

1. $f = 0$
2. Initialize T to be the right-first search tree backwards from t
3. Initialize T^* to be all edges not in T
4. If path from s to t is residual in T, saturate it, modifying f
5. Let $d = \{u, v\}$ be the closest non-residual edge to t
6. Let f_1 be the face to the left of d
7. Let f_2 be the face to the right of d
8. If f_1 is a descendent of f_2 in T^* then return f
9. Let f_3 be the parent face of f_2 in T^*
10. Let x be the node to the right of the edge $\{f_2, f_3\}$
11. Let y be the node to the left of the edge $\{f_2, f_3\}$
12. $T = T - \{u, v\} + \{x, y\}$
13. $T^* = T^* - \{f_2, f_3\} + \{f_2, f_1\}$
14. Reverse edges along path from x to u
15. Go to Step 4

Table 2. Borradaile and Klein's Algorithm

showing the rightmost search tree and its planar dual after each iteration of the loop.

2.2. Implementation

Borradaile and Klein's algorithm calls for a dynamic tree data structure so the operations in Steps 4, 8, and 14 can be computed in $O(\lg n)$ time. Since this structure is used exclusively to find paths from an interior node to one of its ancestors in a tree, we used a simple array of parent pointers instead. Implementing the rest of the functionality provided by dynamic trees was unnecessary for this application.

Our implementation decisions are described in more detail in Section 4.

3. Intelligent Scissors

For our implementation, the user's selection is indicated by a pixel on the object, which we will label s. Using this pixel and one known to not be on the object t, the scissors software can attempt to minimize some cost function which reflects the cost of labeling each pixel P as part of the class including s or the class including t.

Since the primary focus of this paper is implementing Borradaile and Klein's algorithm, our intelligent scissors are not actually very intelligent. In fact, while Mortensen and Barrett's intelligent scissors accepted multiple points and minimized a curve fitting those points, what we attempt here is somewhat more difficult. We are trying to find an object using only a single pair of points, neither of which may be near the edge.

As you can see in Figure 2, we were only partially successful in scissoring the Jayhawk's beak, or even the whole Jayhawk. This was mainly due to our impoverished local cost function[2] which is only based on the gradient magnitude. Assuming $0 \leq \Delta I_x < 1$, this can be defined for the horizontal edges as:

$$l(p, q) = \frac{1 + \epsilon}{(\Delta I_x + \epsilon)}$$

Also, the scaling factor ϵ used to create "infinite" edge weights was not small enough to overcome the large size of the image, allowing the algorithm to take shortcuts.

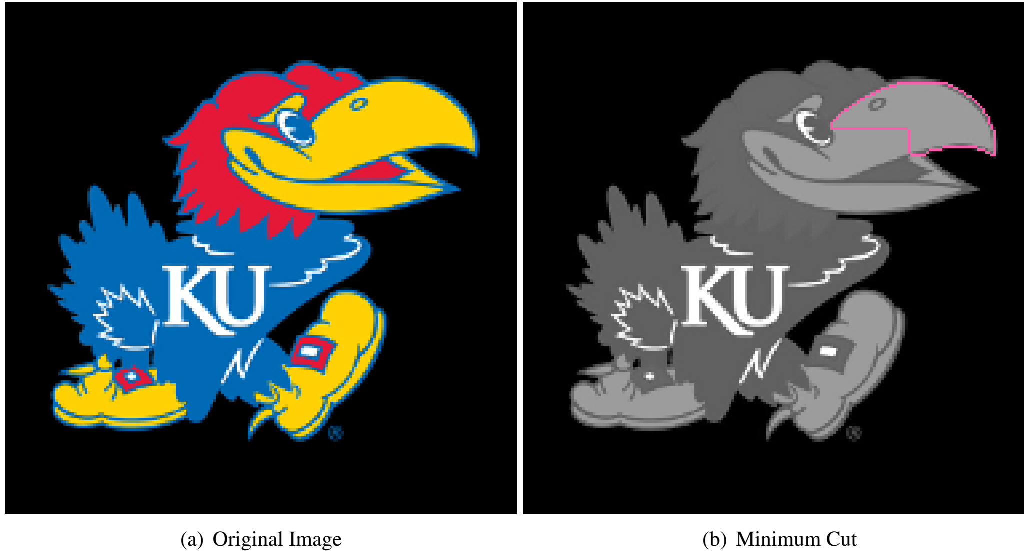
We implemented the algorithm as a Matlab extension in C++. The C++ portion takes a pair of matrices representing the rightward and downward edges between pixels, and returns a boolean heat map of pixels around the edge of the cut. To create this heat map, we use the cycle created in the planar dual of the graph at the termination of the algorithm. We walk this cycle, marking all nodes to the left of each dual edge. A pixel in the heat map is 1 if is on the edge of the object containing the source, and 0 otherwise. This interface allows us to handle less performance critical aspects in Matlab code and leverage some of the built-in image handling capabilities of the Image Toolkit.

4. Performance

To analyze the performance of our implementation, we will examine the complexity of each step in the algorithm.

1. Initializing f to 0 is obviously $O(1)$.
2. Creating the right-first search tree is done via a DFS of the graph, starting at t. At each node, the rightmost unvisited child (relative to the incoming edge) is chosen and the search recurses. When no child of node v is unvisited, control is returned to the parent node u, which continues by examining children to the left of v. Since the graph is bounded and planar, meaning it has at most $3n$ edges, this step requires $O(n)$ time.
3. Creating the planar dual T^* simply requires examining all edges to determine whether they are part of T. The order of the search matters, in order to build a proper array of parent pointers, but each face is only examined once. Since each face has at most four adjacent edges, and there are $O(n)$ faces in our graph, the running time is $O(n)$.
4. Saturating the path from s to t in T requires a single loop to follow parent pointers from s (an interior node in T) to t (the root of T), finding the maximum possible augmenting flow along the way. A second loop traverses the same path, augmenting the flow. Since T is a tree, and the average path from any node to the root in a tree is logarithmic, this step requires $O(\lg n)$ time.

[2]This was mainly due to time constraints. We are confident the scissors could be greatly improved with some work on the cost function.



(a) Original Image

(b) Minimum Cut

Figure 2. A sample result. The source was placed on the Jayhawk's beak, and the sink in the upper left corner. The pink border represents the minimum cut.

5. The edge d is found during one of the loops in Step 4, and requires no additional time.
6. Due to our implementation of the graph as a pair of arrays (one of size n for nodes and another of size $2n$ for edges)[3] the face f_1 can be found in constant time using index arithmetic and the index of d.
7. Finding the face f_2 is also $O(1)$, by the same method we used to find f_1.
8. Determining whether f_1 is a descendent of f_2 in T^* simply requires following parent pointers from f_1, stopping when we reach either f_2 or the root of T^*. Since T^* is a tree containing $O(n)$ faces, this operation takes $O(\lg n)$ time in the average case.
9. Finding f_3 is done in constant time by simply following the parent pointer of f_2.
10. The node x is found in constant time using index arithmetic and the index of the dual edge $\{f_2, f_3\}$.
11. The node y is found in the same manner as x.
12. Since T is really just an array, removing $\{u, v\}$ and adding $\{x, y\}$ is a pair of constant time array accesses.
13. T^* is also an array, so removing $\{f_2, f_3\}$ and adding $\{f_2, f_1\}$ is also $O(1)$.
14. Reversing the edges along the path from x to u, takes at most $O(\lg n)$ time, since x and u are both nodes in the tree T.

As we can see, the maximum time consumed by any step in the loop formed by Steps 4-14 is $O(\lg n)$. As previously mentioned, Borradaile proves that the loop runs at most n times, yielding a total runtime of $O(n \lg n)$. Figure 3 plots some sample running times for various sizes of the Jayhawk image.

[3]For a pairwise-connected MRF of height h and width w it's easy to see that there are $n = h * w$ nodes and $m = 2 * h * w - h - w$ edges. It follows that $2n > m$, so an array of size $2n$ is sufficient to hold every edge.

5. Conclusion

A common approach to computer vision problems is to minimize a specially formulated energy function. As we have seen, graph cuts is a powerful tool in our quest for minimization. While most naturally suited for binary decisions, graph cuts can be applied to other problems by breaking them down into a series of binary subproblems. With recent breakthroughs in the area of planar maximum flow algorithms, many problems that were previously computationally difficult are now feasible.

The intelligent scissors problem represented a good vehicle for testing our implementation of Borradaile and Klein's algorithm. While this author lacked the time to properly explore all the techniques that are applicable to formulating the local edge cost function, we feel that our naive approach served its purpose.

5.1. Improvements

There are numerous things we would like to flesh out or improve with this work.

First and most pressing is to write an adequate edge cost function, as mentioned above. There are several techniques that come to mind, including taking into account color and texture. Edge detection was initially thought to be helpful here, but the edges often proved too noisy or full of holes, which allowed the algorithm to find "short cuts". Laplacian

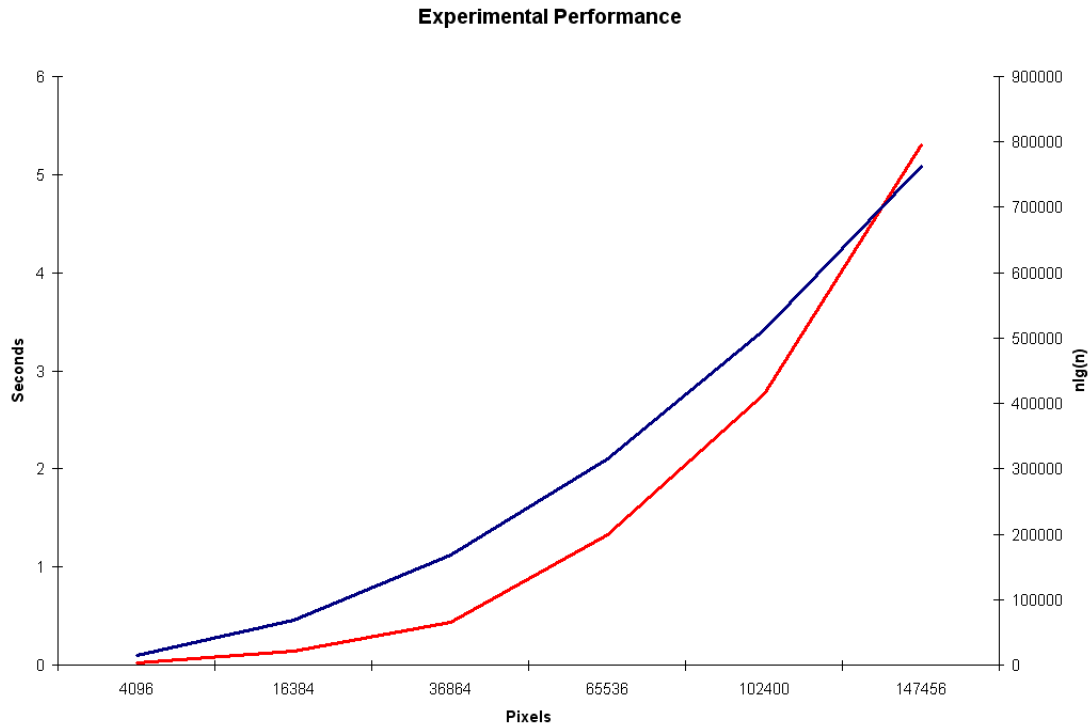


Figure 3. Experimental runtime performance of our implementation. The time taken is plotted alongside an $n \lg n$ curve for comparison. The timing data is somewhat noisy, and looks suspiciously like an n^2 curve would be a better fit, but it's difficult to tell without more data points.

edge detection with a threshold of zero gives closed curves, but was found to be too noisy. As explored by Mortensen and Barrett, some combination of multiple metrics is probably the best approach. Some helpful leads are offered by Schmidt *et al*. [7]. One particular approach that came to mind late in the process was to express the network as a series of higher-order factor nodes. Each factor node would be based on some property of a local group of pixels, and the edges of the network would be between each factor node. This has two properties that make it appealing. Foremost, it allows for higher-order relationships to be represented. In addition, it reduces the effective size of the network for which the minimum cut must be computed, at a slight cost to accuracy at the pixel level.

Second, the Matlab-to-C++ interface could use some work. We ran into several pitfalls getting the initial interface working, not least of which was an incompatibility between the Matlab runtime and GCC on the EECS computers. Getting past that, there seem to be some memory limitations imposed by the Matlab garbage collection system that need to be worked around. This became apparent when trying to test large images, and is the reason that we only have six data points on our performance graph.

Third, more extensive consideration of the actual performance seems in order. The (red) curve representing running time in our graph looks suspiciously quadratic. This is either caused by noise in our small sample size, or an implementation oversight. It's also possible that our particular MRF formulation is problematic. One assumption in the $O(n \lg n)$ running time is that the T and T^* trees remained fairly balanced. However, we can see, even in our walk-through example, that the initial trees are always degenerate. While they become more balanced as the algorithm continues, it's difficult to say whether performance of the tree traversal is ever really $O(\lg n)$ in practice.

Finally, the C++ code itself could most definitely be cleaned up. To paraphrase the famous saying: "I sat down to write a small program, but ran out of time, so I wrote a big one."

References

[1] G. Borradaile. *Exploiting Planarity for Network Flow and Connectivity Problems*. PhD thesis, Brown University, May 2008.

[2] G. Borradaile and P. Klein. An $O(n \log n)$-time algorithm for maximum st-flow in a directed planar graph. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 524–533, 2006.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[4] C. Ford and D. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[5] D. M. Greig, B. T. Porteous, and A. H. Seheult. Exact maximum a posteriori estimation for binary images. *Journal of the Royal Statistical Society Series B*, 51:271–279, 1989.

[6] E. N. Mortensen and W. A. Barrett. Intelligent Scissors for Image Composition. In *Computer Graphics (SIGGRAPH '95)*, pages 191–198, Los Angeles, California, August 1995.

[7] F. R. Schmidt, E. Toeppe, and D. Cremers. Efficient planar graph cuts with applications in computer vision. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Miami, Florida, June 2009.

[8] D. D. Sleator and R. E. Tarjan. A Data Structure for Dynamic Trees. *Journal of Computer and System Sciences*, 26(3):362–391, June 1983.

[9] K. Weihe. Maximum (s, t)-flows in planar networks in $O(|V|\log|V|)$ time. *Journal of Computer and System Sciences*, 55(3):454–476, 1997.