Binary Image Segmentation Using Graph Cuts

6.854 Advanced Algorithm Term Project

Ying Yin
yingyin@csail.mit.edu

Abstract

We implemented several maximum-flow algorithms, and applied them for segmentation of a degraded binary image. The purpose of the segmentation is to track the position of the hand in camera images for gestural interaction. The segmentation is based on the maximum *a posteriori* estimation which can be reformulated as a network flow problem. We compare the running time of the different algorithms we implemented together with an existing fast implementation. The results show that the fastest implementation is suitable for solving the segmentation problem in real-time.

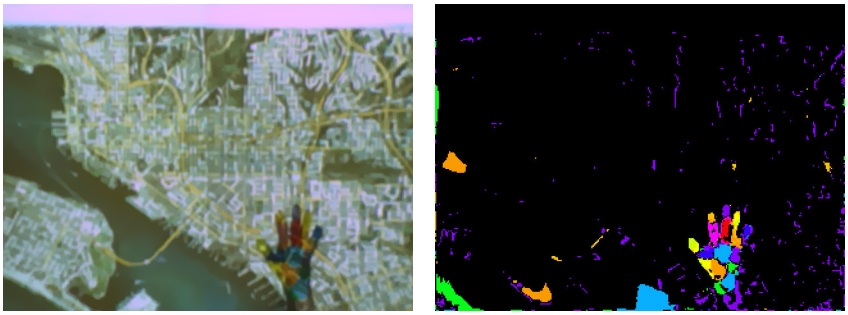
1 Introduction

In this project, we apply maximum-flow (max-flow) minimum-cut (min-cut) algorithms to image segmentation of a degraded binary image. The purpose of the segmentation is to track the position of the hand on a tabletop display for gestural interaction. The colored glove allows easy reconstruction of 3D hand postures. We need to track the hand in real-time for user interaction. We color classify the camera image (Figure 1(a)) in order to extract the hand efficiently. If the color classification is perfect, i.e., only the hand is classified as colored pixels and the rest pixels in the background are classified as black, we can find the location of the hand in the image easily by calculating the centroid of all the colored pixel. However, due to the complex background, there are many mis-classifications (Figure 1(b)) in the background. These misclassified colored pixels can adversely affect the centroid calculation.

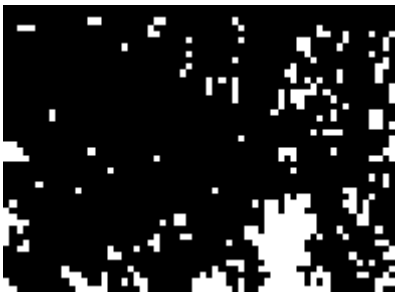
For tracking the position of the hand, we can ignore the actual color of the pixels, and convert the image to binary (Figure 1(c)). The mis-classifications can be considered as noise added to the binary image. To remove the noise, we find the maximum *a posteriori* (MAP) estimate of the true scene from the degraded binary image. The direct calculation of the MAP estimate is in general computationally prohibitive, however Greig et al. [4] shows that the problem can be reformulated to a minimum cut problem in a certain capacitated network, and max-flow min-cut algorithms can be used to find the MAP estimate exactly and efficiently. In Section 2, we describe the reformulation of the problem, and in Section 3, we outline several algorithms we implemented (including the Ford-Fulkerson and the blocking flows algorithms) or used (Cherkassky and Goldberg's implementation of the push-relabel method) for solving the min-cut problem. The experimental results appear in Section 4 with an analysis of the running times.

2 Graph Cuts Formulation of MAP Estimation

In this section, we present the details of a network reformulation of the MAP estimation problem for binary images as explained by Greig et al. [4]. Let $x = (x_1, ..., x_n)$ denote a binary image with n pixels, and each pixel x_i is either white (1) or black (0). Let x^* be the unknown true scene and $y = (y_1, ..., y_n)$ be the observed records of x^*. The MAP estimate of x^* is the image $\hat{x}$ which maximizes the *a posteriori*



(a) (b)



(c)

Figure 1: (a) Camera image (b) Color classified image (c) Black and white image

distribution $p(x|y)$. Let $l(y|x)$ be the likelihood of any image x and $p(x)$ be the *a priori* distribution over all allowable images, then $p(x|y) \propto l(y|x)p(x)$.

The observed records $(y_1, ..., y_n)$ are assumed to be conditionally independent given x [4], and each has known conditional density function $f(y_i|x_i)$, dependent on x only through x_i. Thus the likelihood function for x may be written as

$$l(y|x) = \prod_{i=1}^{n} f(y_i|x_i) = \prod_{i=1}^{n} f(y_i|1)^{x_i} f(y_i|0)^{1-x_i}.$$

The prior distribution $p(x)$ is modeled as a pairwise interaction Markov random field (MRF) of the form

$$p(x) \propto \exp[\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \beta_{ij}(x_i x_j + (1-x_i)(1-x_j))]$$

where $\beta_{ij} = \beta > 0$ if i and j are neighbors (each pixel has eight neighbors except those at the edges), and $\beta_{ij} = 0$ otherwise [4]. This model quantifies the belief that the unknown true scene x^* consists of large homogeneous patches.

Thus, apart from an additive constant, $\ln p(x|y)$ can be written as

$$L(x|y) = \sum_{i=1}^{n} \lambda_i x_i + \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \beta_{ij}(x_i x_j + (1-x_i)(1-x_j)) \quad (1)$$

where $\lambda_i = \ln(f(y_i|1)/f(y_i|0))$, a log-likelihood ratio at pixel i [4]. The MAP estimate is the image $\hat{x}$ which maximizes L. There are 2^n possible values of L making direct search for $\hat{x}$ infeasible even for a small image of $n = 64 \times 64$.

Reformulating the problem as a minimum cut problem in a network provides an efficient way to find the exact MAP estimate. Consider a directed graph G (Figure 2) comprising n+2 veritices, being a

source s, a sink t and the n pixels. There is an edge $e(s,i)$ with capacity $u_{si} = \lambda_i$, if $\lambda_i > 0$; otherwise, there is an edge $e(i,t)$ with capacity $u_{it} = -\lambda_i$. The value of λ_i depends on the value of y_i. There are two edges $e(i,j)$ and $e(j,i)$ between two internal vertices (pixels) i and j with capacity $u_{ij} = u_{ji} = \beta$ if the corresponding pixels are neighbors.

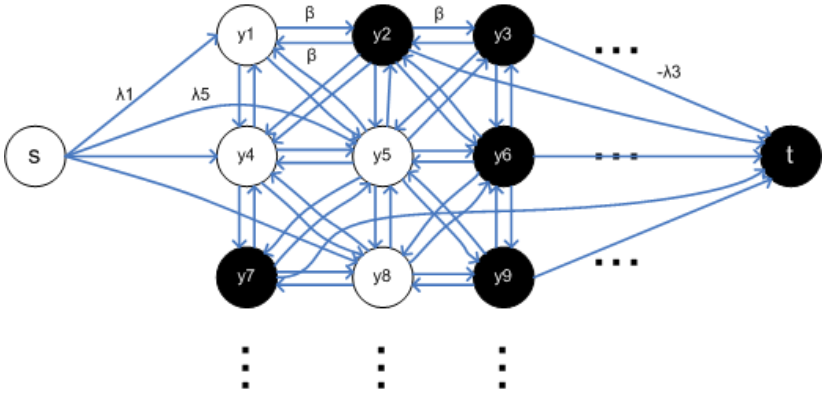


Figure 2: Graph G from network reformulation of the MAP estimate. The color of the nodes indicates the value of the observed record y_i of each pixel.

For any binary image $x = (x_1, ..., x_n)$ let $W = \{s\} \cup \{i : x_i = 1\}$ and $B = \{i : x_i = 0\} \cup \{t\}$. Then (W, B) is a cut for G. The capacity of the cut can be written as

$$u(W) = u(x) = \sum_{i=1}^{n} x_i \max(0, -\lambda_i) + \sum_{i=1}^{n} (1 - x_i) \max(0, \lambda_i) + \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \beta_{ij} (x_i - x_j)^2$$

which differs from $-L(x|y)$ by a term which doest not depend on x. Thus, maximizing $L(x|y)$ is equivalent to finding the minimum cut in G, i.e., in the MAP estimate pixels are white if they are on the source side of the minimum cut, and are black otherwise.

3 Implementation of Min-Cut Algorithms

There are many algorithms for solving the max-flow min-cut problem with different running times. In order to evaluate the performance of these algorithms on our particular image segmentation problem, we implemented the Ford-Fulkerson algorithm with different methods of finding an augmenting path, the blocking flow algorithm by Dinic, and also used Cherkassky and Goldberg's implementation of the push-relabel method. All implementations use the adjacency list representation of the input graph. We give a brief description of these algorithms here.

3.1 Ford-Fulkerson Algorithm

The Ford-Fulkerson algorithm solves the problem by repeatedly finding an augmenting path in the residual network G_f until there is none. We implemented three methods for find the augmenting path, namely depth-first-search (DFS), breadth-first-search (BFS, also called shortest augmenting path), and maximum-capacity augmenting path (MC).

The implementation for DFS and BFS are relatively straight forward. The MC method is similar to the Dijkstra method. We assign a priority to each vertex the minimum capacity of a path in the residual network from the source to the vertex, and process the vertices in decreasing order of priorities. We use a priority queue to efficiently find the vertex with the highest priority, and increase the priority of a vertex. The following is the pseudocode for finding the maximum-capacity augmenting path [5].

```
//define a node structure
struct Node
  vertex    //id of the node
  priority  //the bottleneck capacity of the path from source to vertex
  prev      //previous vertex on the path

int MC()
  priority queue pq
  from[] //an array initialized to -1 and keeps track of the visited
         //node and the previous node leading to the current node
  push Node(source, infinity, -2) to pq
  flow = 0

  while pq is not empty
    node = pop from pq
    current = node.vertex, cost = node.priority
    from[current] = node.prev //mark current vertex as visited

    if(current == sink)
      flow = cost
      break from while loop

    for each vertex next adjacent to current
      if(from[next] == -1 and capacity[current][next]>0)
        new_cost = min(cost, capacity[current][next])
        push Node(next, new_cost, current) to pq
    end for

  end while

  //update the residual network
  current = sink
  while from[current] > -1
    prev = from[current]
    capacity[prev][current] -= flow
    capacity[current][prev] += flow
    current = flow
  end while

  return path_cap
```

Listing 1: Pseudocode for finding maximum-capacity augmenting path.

3.2 Blocking Flows Algorithm

The blocking flow algorithm by Dinic (DINIC) is an extension of the shortest augmenting path method. Both methods use breadth-first-search to find an augmenting path. When we do breadth-first-search, we obtain many layers in the graph. We can find multiple paths through the layers of the graph and augment along many of them at once. The DINIC algorithm does this by conserving information about edges once traversed, and only updates the residual network after pushing flow through all admissible paths. The running time of DINIC on a general graph without capacity scaling is $O(mn^2)$.

The following (Listing 2) is the pseudocode of a relative simple implementation of Dinic's algorithm without using any fancy data structure. During the breadth-first-search, we use an array prev[] to store the previous node u that leads to the current node v in the shortest path from s to v. After an augmenting path is found, we push flow through all admissible paths that have the same shortest length by back-

tracking using prev[]. The flow pushed in each admissible path equals to the bottleneck capacity of the path.

```
int DINIC()
  queque q
  prev[]

  flow = 0

  while true
    initialize prev[] to -1
    push source to q
    prev[source] = -2 //mark source as visited
    //breadth-first-search to create an layered graph
    while q is not empty and prev[sink] == -1
      u = pop from q
      for each vertex v adjacent to u
        if v is not visited and capacity[u][v]>0
          push v to q
          prev[v] = u
      end for
    end while

    if prev[sink] == -1 //cannot find path anymore
      break while loop

    //augment through all admissible paths
    for each vertex z
      if m[z][sink] > 0 and prev[z] != -1
        //find bottle_neck capacity of the path from sink to z to source
        bottle_neck = capacity[sink][z]
        v = z and back-track using prev[v] to update bottle_neck

        if bottle_neck > 0
          update capacities in path from sink to z to source in G_f

      flow += bottle_neck
    end for
  end while
  return flow
```

Listing 2: Pseudocode for DINIC.

3.3 Push-Relabel Algorithm

The push-rebel method again improves on the blocking flows method. One improvement is being lazy, and only updating the distance labels when necessary. The other is making the work count by pushing flow along each augmentable edge found instead of waiting to find an augmentation path.

The efficiency of the push-relabel method depends on the ordering of the update (push and relabel) operations [1]. For the generic push-relabel method, there is no particular ordering of selecting an applicable push or relabel operation, and the algorithm runs in $O(mn^2)$ time. This time bound is not better or worse than DINIC. We used Cherkassky and Goldberg's implementation of the highest-label (HLPR) variant of the push-relabel method[1]. The HLPR algorithm repeatedly selects an active node

[1]http://www.avglab.com/andrew/soft/hipr.tar

(node with excess flow) u with the highest label, and then "discharges" it, that is, performing push and relabel operations until u no longer has a positive excess [3]. This results in a time bound of $O(n^2\sqrt{m})$.

It would not be efficient if we use a heap to keep track of the vertex with the highest label. As the distance label is bounded by n, using an array and buckets will be more efficient. The HLPR algorithm maintains an array of sets B_i (buckets), $0 \leq i \leq n-1$, and an index b into the array. Set B_i consists of all active nodes with label i, so that insertion and deletion take $O(1)$. The index b is the largest label of an active node [1].

The HLPR algorithm has a much better time bound. However, it still has poor practical performance because relabel is a local operation, and the method loses the global picture of the distances [1]. Two heuristics, the global relabeling and gap relabeling heuristics, are used in Cherkassky and Goldberg's implementation which drastically improves the running time.

The global relabeling heuristics updates the distances in the residual graph from all nodes to the sink in linear time. As this update is computationally expensive compared to the push and relabel operations, it is performed only periodically [1].

The gap relabeling heuristics updates the labels of nodes that are disconnected from the sink to n so that they will no longer be active. These nodes can be found efficiently by detecting the gap among the distance labels of all the nodes.

4 Experimental Results

4.1 Computing Environment

Our experiments were conducted on a Dell Vostro 420 Tower desktop with a Intel Core2 Quad processor at 2.83GHz running WindowsXP Professional. The desktop has 3GB of RAM. All codes we implemented are written in C++. The code by Cherkassky and Goldberg is written in C. All codes are compiled with GCC compiler version 3.4.4 (cygming special).

4.2 Input

Given a binary image (the observed records) $y = (y_1, ..., y_n)$ with noise (Figure 1(c)) as an input, we want to estimate the true scene x^* by labeling the hand as white (1) and the background as black (0). An analysis of such input images yields the statistics in Table 1.

$f(y_i\|x_i)$	value
$f(1\|1)$	0.99
$f(0\|1)$	0.01
$f(1\|0)$	0.1
$f(0\|0)$	0.9

Table 1: Values for the conditional density functions.

To construct the network G from y according to the reformulation in Section 2, we need to find the corresponding capacities based on $\lambda_i = \ln(f(y_i|1)/f(y_i|0))$ and β. From the above statistics, if $y_i = 1$, $\lambda_i = 2.3$; if $y_i = 0$, $\lambda_i = -4.5$. We choose $\beta = 0.3, 0.7$ or 1.1 (the same values as chosen by Greig at el. [4]) to investigate the effect of varying β. We scale all these values by 10 to have integral capacities.

The original image is 480×640 pixel in size. We scale the image down to various sizes to investigate the effect of different input sizes on the running time of the algorithms.

4.3 Output

The min-cut algorithms separate the pixels into two sets. The pixels in the same set as the source are labeled white, and the pixels in the same set of the sink are labeled black. For an input image of size 44×60, obtained by scaling the original image (see Figure 1(c)) by a factor of 0.1 and removing the edges, the exact MAP estimates of the true scene are shown in Figure 3 for different choices of β. The result is best when $\beta = 1.1$. This is because the larger value of β increases the weight of the prior distribution $p(x)$ in the MAP estimation (Equation (1)), making the result smoother.

(a) β=0.3 (b) β=0.7 (c) β=1.1

Figure 3: Exact MAP estimates with different β values.

4.4 Running Times

Table 2 shows the running times of the five algorithms mentioned in Section 3 for different input sizes by scaling down the input image. The scale factor refers to the scale in one dimension of the image, hence, the actual scale down will be the square of the scale factor. The results reported here all use $\beta = 1.1$. When tabulating results of our experiments, we give the running times in milliseconds. The running time excludes the input and output times. We also plot the running times against the number of nodes in log scale (Figure 4) to visualize the results better.

scale	nodes (n)	edges (m)	max flow (f)	MC	DFS	BFS	DINIC	HLPR
0.1	2642	23140	7137	897	658	265	12	1
0.2	10562	93796	25332	15300	6688	4131	112	3
0.3	23762	211972	53285	166515	32000	21263	544	8
0.4	42242	377668	85923	475810	120552	69861	1040	17

Table 2: Running times for different input sizes in ms.

The three algorithms (MC, DFS, and BFS) in the Ford-Fulkerson family are close in performance, although max-capacity augmenting path (MC) is asymptotically slower. MC has running time $O(m^2 \log f)$ which is quadratic in the number of edges. The graph is very dense with most of the vertices having degree 9 as each pixel vertex is connected to the source or the sink, and eight neighboring pixel vertices (except those on the edges). As a result, a time bound which is quadratic in m will drastically affect the performance. Moreover, as we use a Dijkstra-like algorithm for finding the maximum-capacity augmenting path, it takes $O(m \log n)$ time to find such a path. Hence the actual running time of MC is $O(m^2 log f log n)$, making it even slower. The running time of BFS is $O(m^2 n)$ which is also quadratic in m, but in practice it is much faster than MC, and is also faster than DFS which has a time bound of $O(mf)$.

From those time bounds, it is not obvious to reason the results in Table 2 for MC, DFS and BFS. This is because these are worst case time bounds. In practice the actual topology of the network, and the implementation of the algorithms will also affect the performance. The experimental results show that

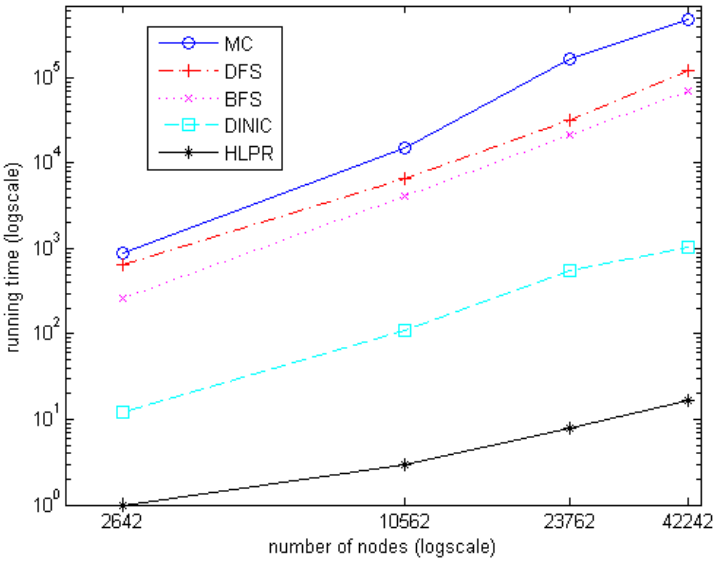


Figure 4: Plot of running time vs. number of nodes in log scale.

DFS and BFS have similar performance with BFS being slightly faster by about a factor of two. Both DFS and BFS take $O(m)$ time to find an augmenting path. However, BFS makes fewer iterations of finding augmenting paths. For an input with scale factor 0.1, BFS takes 984 iterations, while DFS and MC takes 6753 and 2057 iterations respectively. BFS is more efficient by making the work count.

If we look at the input image (Figure 1(c)), the small white patches of pixels are the noise we want to remove. As the white pixels are only connected to the source, and the black ones are only connected to the sink, any path must be from a while pixel to a black pixel. We convert the noisy white pixel to black by saturating the edge from the source to the white pixel. These white pixels corresponds to vertices on shortest paths because they are very close to black pixels which are connected to the sink. Instead of being directly "primal" greedy as in MC, BFS is being greedy in the "dual" sense by augmenting through shortest paths to increase the distance between the sink and the source.

Intuitively, there could be many short paths with the same length in the graph, so DINIC saturates all shortest paths in one blocking flow, making it much faster than BFS. The experiments show that, for an input size with scale factor 0.1, DINIC makes 25 iterations of blocking flow. This is much fewer iterations than BFS. The running time of the blocking flows algorithm for a general graph is $O(mn^2)$ which is quadratic in the number of nodes. For a dense graph, the number of nodes is much smaller than the number of edges like in our case, so DINIC is orders of magnitude faster than the Ford-Fulkerson family algorithms. For an input with scale factor 0.1, it takes 12 ms to finish which makes it possible to use it in a real-time application.

The highest-label push-relabel (HLPR) is the fastest by orders of magnitude, and it scales well with increasing input size. By discharging from the highest label, HLPR "accumulates" flow into fewer piles when moving towards the sink. This reduces the number of nonsaturating pushes, making the work done more efficient. Without using heuristics, HLPR runs in $O(n^2\sqrt{m})$ time in the worst case [2]. Hence, HLPR should have better performance than DINIC asymptotically especially for dense graphs. As the results show, with the global and gap relabeling heuristics, HLPR is superior to DINIC by a wide margin.

Setting $\beta = 1.1$ gives the best image estimation result. However, different β values change the capacities of the graph, and hence, the total maximum flow f. We investigate the effect of changing f on the running time. We run the algorithms on the input with scale factor 0.1. The results are shown in Table 3 and Figure 5. The increase in the maximum flow does not affect the running times much, but it is interesting to observe that the running times of MC and BFS decrease with increasing f. This is because

the number of iterations in BFS and MC increases with decreasing β. As β is usually the bottleneck capacity in the augmenting paths, it requires more augmenting paths to saturate the edge from the source to a pixel vertex when β is smaller.

nodes	arcs	max flow (f)	MC	DFS	BFS	DINIC	HLPR
2642	23140	4196	1747	519	520	8	0.74
2642	23140	6376	915	621	333	10	0.89
2642	23140	7137	897	658	265	12	0.92

Table 3: Running times with the same number of nodes but different maximum flow in ms.

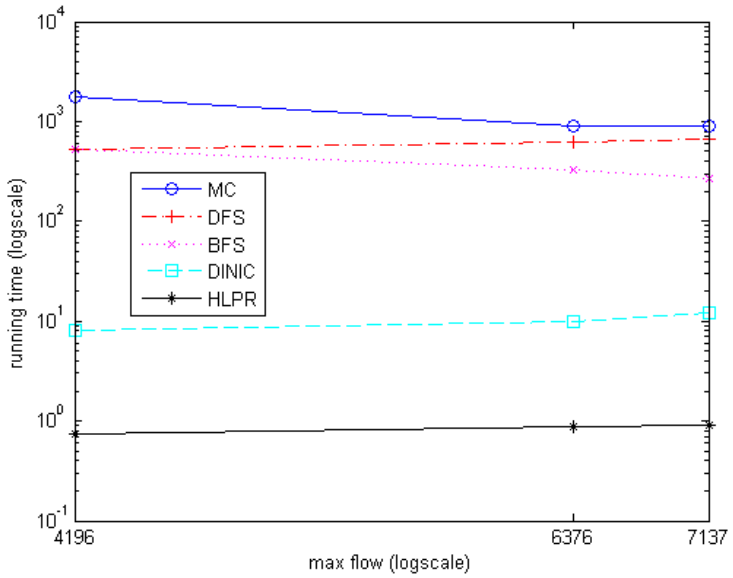


Figure 5: Plot of running time vs. maximum flow in log scale.

5 Conclusion

The results show that the exact MAP estimation gives accurate restoration of the true scene from a degraded binary image. Using this technique, we can remove the mis-classifications in the background, and segment out the hand. We can extract the location of the hand accurately by calculating the centroid of the white pixels. The network reformulation, together with the max-flow min-cut algorithms, provides an efficient way to find the exact MAP estimation. However, the Ford-Fulkerson algorithms are not fast enough for real-time applications. The HLPR with heuristics is fantastic in practice, and is the best choice for real-time applications. However, the simple DINIC algorithm also have good performance on relatively small problem size, and can also be considered for real-time implementation. As our goal is to find the location of the hand, scaling the image to small size will not affect the result. The DINIC algorithm without fancy data structure is much simpler to implement.

References

[1] B. V. Cherkassky and A. V. Goldberg. On implementing push-relabel method for the maximum flow problem. *Algorithmica*, 19:390–410, 1994.

[2] E. Cohen and N. Megiddo. Strongly polynomial-time and nc algorithms for detecting cycles in dynamic graphs. In *STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 523–534, New York, NY, USA, 1989. ACM.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd revised edition, September 2001.

[4] D. M. Greig, B. T. Porteous, and A. H. Seheult. Exact maximum a posteriori estimation for binary images. 1989.

[5] _efer_. Maximum flow. TopCoder Algorithm Tutorials, Dec 2009. URL `http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=maxFlow/`.