EC504 Final Project Report
Charles Mo, Darren Eldredge, Do Hun Ji
Team 3 - https://github.com/Flymindo/EC504_FinalProject

Max Flow Image Segmentation

Image segmentation is the task of partitioning pixels of an image into an arbitrary number of sets based on some criterion. In our project, we explore two methods of binary segmentation, wherein an image is segmented into foreground and background, using maximum flow: Edmonds-Karp and Push-relabel. To achieve this, we first construct a pixel similarity graph between neighboring pixels on a gray-scaled image and thereafter apply a probability model such as Mixture of Gaussians or Kmeans to estimate foreground and background probabilities to construct source and sink nodes. Then we apply each maximum flow algorithm to compute minimum cuts to between pixels, creating a segmented graph. In this report, we explain the details for graph construction and each algorithm. Additionally, we will discuss advantages and disadvantages of each algorithm as well as analyze their time and space complexities.

Graph Construction

In order to perform maximum flow segmentation on an image, we constructed a pixel similarity graph and subsequent source and sink nodes to perform maximum flow calculations. Edge weights of our similarity graph are based on the similarity in gray levels between neighboring pixels. To achieve this, we mapped each difference in gray level between neighboring pixels (which become edge weights between pixels) to a gaussian-like function:

$$E(i,j) \;=\; 100\, exp\left(-\frac{(I(i)-I(j))^2}{2\sigma^2}\right),\;\; \sigma \;=\; 30$$

Larger differences in gray level results in smaller values assigned to edges and the converse for similar gray levels. From this, we observed the Space complexity of this, for an MxN image to be O(M^2N^2) for an adjacency matrix implementation and O(4MN) using an adjacency matrix since each pixel is really connected to at most four other pixels in our graph representation. Next we constructed source and sink edges to perform maximum flow. We began by computing the weights on those edges.

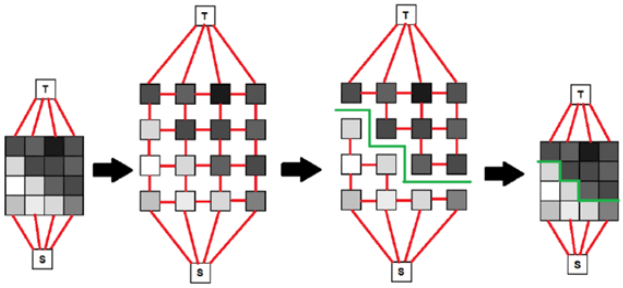


Image of constructed graph with T (sink node) and S (source node) with undirected edges towards pixel similarity graph.

We did this with a probability model, Mixture of Gaussians with two components, due to its generalizability over another clustering algorithm, K-Means. We used two components because we performed binary image segmentation (i.e. foreground and background). Given user points, we initialized to foreground and background means and fit the model and get foreground and background probabilities for each pixel. We then use the foreground and background probabilities as weights for source and sink edges respectively. This increases the space complexity to be O((M+2)^2(N+2)^2) and O(6MN) for the adjacency matrix and list respectively. Next, we built two maximum flow algorithms.

Edmonds-Karp

The Edmonds-Karp algorithm was first published by Yefim Dinitz in 1970 and independently published by Jack Edmonds and Richard Karp in 1972. The algorithm is basically implementation of Ford-Fulkerson method for computing possible maximum flow in a network. Hence, the basic idea on computing maximum flow in the Edmonds-Karp algorithm is very similar to that of the Ford-Fuller algorithm. The Edmonds-Karp uses Breadth First Search method for finding an augmenting path from source to sink which reduces the probability of finding a path with several bottle-neck edges.

Time Complexity

When the Breadth First Search method is used, the Edmonds-Karp algorithm has a time complexity of $O(VE^2)$ where V represents a number of vertices and E represents a number of edges.

Implementation

Our implementation of the Edmonds-Karp algorithm required two loops for iteration. One iteration is for Breadth First Search to find a path from the source node to the sink node and calculating an amount of flow went through over the path. The other

iteration is for updating the remaining capacities of edges based on the flow went through over the path. First, before starting the Breadth First Search, we enqueue the source vertex as the start. While the queue is not empty, we compute all the neighboring vertices and enqueue the vertices to the queue if they are unmarked as visited. Before the process of enqueuing nodes to the queue, we compute the maximum amount of the flow that can pass by comparing the current amount of flow up to the previous vertex and the capacity of the edge from the previous to the current. If the amount of already flowed flow is greater than the capacity of the edge, then we reduce the amount of flow to pass. If not, we flow the same amount of the flow to the next neighboring node. While in the process of the Breadth First Search, if the next node is the sink node, then returns the maximum possible amount of the flow. When the flow is determined over the iteration, then remark all the nodes to be unvisited and subtracts the amount of the flow from all the capacities of edges on the path. We repeat this process until flow cannot be passed from the source to the sink. After all the process, we can find out the min cut of the graph, which is our primary goal, by determining reachable nodes from source by using the BFS. All reachable nodes are marked as the foreground and the other as the background.

Push-Relabel Algorithm

The push-relabel algorithm, designed by Andrew Goldberg and Robert Tarjan, was first presented in 1986 and published in 1988. The name push-relabel refers to the two primary operations of the algorithm, gradually converting a “preflow” into a maximum flow by “pushing” flow between neighboring nodes, with the height of each node maintained by the “relabel” function. It is easiest to think of the graph as a series of connected pipes with defined capacity, and the height of each node affects the ability of a fluid to flow to any neighboring node.

Time Complexity

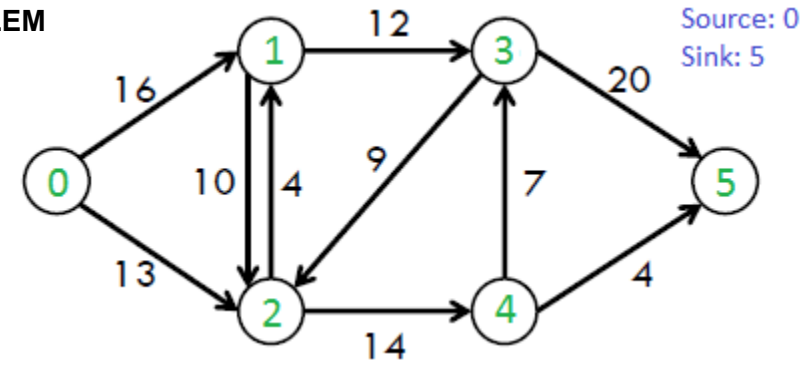
The push-relabel algorithm has polynomial time complexity $O(N^2A)$ or $O(V^2E)$. The relabel operation can be run at most $(2 | N | - 1) (| N | - 2) < 2 | N |^2$ because the height of any node can only increase $(2 | N | - 1)$ times, and the relabel operation can be performed on all nodes {N} except the source and sink node {s,t} which is $(| N | - 2)$ times. The relabel operation therefore has a time complexity of $O(N^2)$. The push operation, on the other hand can be run at most $(2 | N | - 1) (| N | - 2) + (2 | N | - 1) (2 | N || A |) \leq 4 | N |^2 | A |$ times, which gives a time complexity of $O(N^2A)$ or $O(V^2E)$.

Implementation

Our implementation of the push-relabel algorithm uses example (a) from page 726 of the CLRS textbook (Cormen, 2001). We, (specifically me, Darren Eldredge), did

not have time to complete the image segmentation application. Instead, we focused on completing and demonstrating the algorithm by returning an integer max flow. The integer max flow of the graph is calculated and printed to the terminal. Here is a visual representation of the operation of the algorithm on the aforementioned graph from CLRS:

INITIAL PROBLEM



RESULT

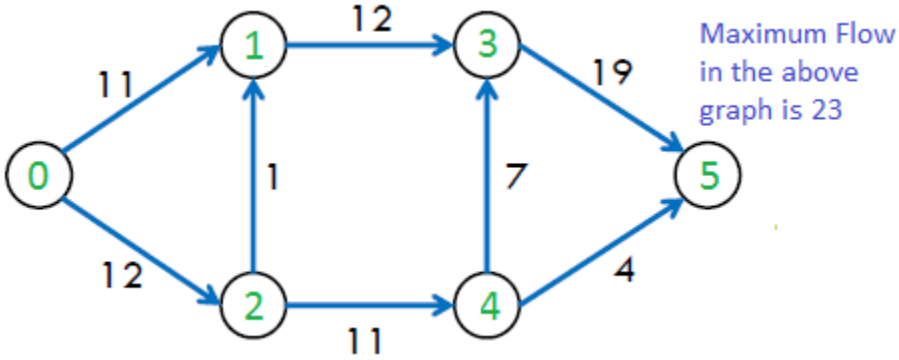


Image Source: https://www.geeksforgeeks.org/push-relabel-algorithm-set-2-implementation/

Our implementation includes a section of incomplete code that has been commented out. This code includes DLIB, a popular computer vision library for C++ with functions to load an image from a file as a 2D-array of pixel objects. Each pixel has a Red, Green, and Blue integer value from 0 - 255 which collectively represent the color of the pixel. The pixels can easily be accessed in the array with .x() and .y() operators. It is also possible to take the average of the RGB values to obtain grayscale intensity.

To calculate the weight, or capacity of arcs/edges, we applied the pythagorean theorem to find the change of color intensity for Red, Green, or Blue values. We can then plug the total change of color intensity into an equation to find the capacity of the edge between two pixels. Selecting sigma to be 30, the final capacity of the edge will be large if the two pixels have a similar color, and small if the colors are very different. Additionally, the balance of RGB colors can be weighted to calculate the total change in color intensity to more closely mimic the unique nature of the human eye, which sees green more strongly than red and blue, as follows:

$$\Delta R = \frac{R_1 - R_2}{255}, \quad \Delta C = \sqrt{2\Delta R^2 + 4\Delta G^2 + 3\Delta B^2}, \quad capacity = 100\, exp\left(-\frac{(\Delta C)^2}{2\sigma^2}\right)$$

The DLIB library includes powerful functions to segment images, generate heatmaps, create bounding boxes, and so forth. We tested some of those functions for fun, but ultimately utilized DLIB functions to display the specified image in a window and collect user input. To collect the foreground and background swatch for image segmentation, the user needs only to double click on two pixels from the displayed image window in the requisite order. The program then creates the requisite source and sink nodes {s,t} based on the RGB information, performs preflow on the graph, and proceeds with the push-relabel algorithm to calculate and return the integer value of maximum flow. However, the algorithm succeeds only part of the time, and is highly dependent on which pixels are initially selected to generate the source and sink nodes. This is where I got hung up, debugging this issue, and that is why we were unable to proceed with the next step, which is to implement the probabilities from s and t to perform a min-cut as explained previously to segment the image into background and foreground objects.

When we presented in class, Professor Brower explained that the infinite looping of the algorithm and inability to settle on max flow could be because the pixels in the image are too similar, or because we did not implement a “reservoir” to preserve the integrity of the system.

Comparative Analysis of Max Flow Algorithms

The time complexity of the push-relabel algorithm is asymptotically more efficient than the Edmonds-Karp algorithm discussed previously (Edmonds, J., Karp, 1972), though even faster algorithms have been developed. In theory, the fastest max flow algorithm achieves O(NM). However, the overhead of such theoretically superior algorithms and the rarity of the worst cases means that “slower” algorithms are frequently preferred (Boykov and Kolmogorov, 2004; Goldberg et al., 2011). Table 1 summarizes the running time of some practical approaches to max flow calculations. Notice that the push-relabel algorithm by Golberg and Tarjan improves efficiency over our approach and achieves time complexity of $O(N^3)$ or less.

Table 1

Flow algorithms

Algorithm	Running Time	Notes
Edmonds-Karp (Edmonds and Karp, 1972)	$O(nm^2)$	Shortest augmenting path
Dinitz (Dinitz, 1970)	$O(n^2m)$	Even's version with optimizations suggested in Dinitz (2006)
Preflow-Push (Goldberg and Tarjan, 1988)	$O(n^3)$	Goldberg and Tarjan's preflow-push algorithm with the highest-label selection rule. A simple, but inefficient, selection implementation process yields the slower runtime
Preflow-Push (Gap) (Goldberg and Tarjan, 1988)	$O(n^2\sqrt{m})$	Goldberg and Tarjan's push relabel method with the highest-label selection rule. Implemented with $O(1)$ selection and the gap relabeling heuristic suggested in Cherkassy and Goldberg (1995)
Improved-SAP (Orlin and Ahuja, 1987)	$O(n^2m)$	Orlin's improved shortest augmenting path method

(Image Source: https://ioinformatics.org/journal/v12_2018_25_41.pdf)

Statement of Work

Charles Mo

Worked a bit on graph construction at the beginning, but primarily worked on GUI for user selection of means for constructing sink and source edge weights in our graph. Organized repository and debugged code. Finalized user interface with Do Hun.

Do Hun Ji

Mainly worked on graph construction and implementation of Edmonds-Karp algorithm. Finalized user interface with Charles as well. Furthermore, I worked to create a single python script that runs all the required codes.

Darren Eldredge

Implemented push-relabel algorithm and tested DLIB in C++, didn't complete image segmentation but was able to return max flow for simple graphs. Wrote push-relabel, and comparative analysis sections of the paper. I would say I contributed less than Charles and Do Hun, but did as well as I could with constrained time. Thanks for a great semester!

References

Boykov, Y., Kolmogorov, V. (2004). An experimental comparison of min- cut/max- flow algorithms for energy minimization in vision. IEEE Transactions on Pattern Analysis and Machine Intelligence, 26(9), 1124– 1137.

Goldberg, A.V., Hed, S., Kaplan, H., Tarjan, R.E., Werneck, R.F. (2011). Maximum flows by incremental breadth-first search. In: Demetrescu, C. and Halldórsson, M. M. (Eds.), Algorithms – ESA 2011. Berlin, Heidelberg. Springer Berlin Heidelberg, 457–468.

Edmonds, J., Karp, R.M. (1972). Theoretical improvements in algorithmic efficiency for network flow problems. J. ACM, 19(2), 248–264.

Ford, L.R., Fulkerson, D.R. (1955). Maximal flow through a network. Canadian Journal of Mathematics, 8, 399–404.

Cormen, T. H., & Cormen, T. H. (2001). *Introduction to algorithms*. Cambridge, Mass: MIT Press.