

The min-cut problem

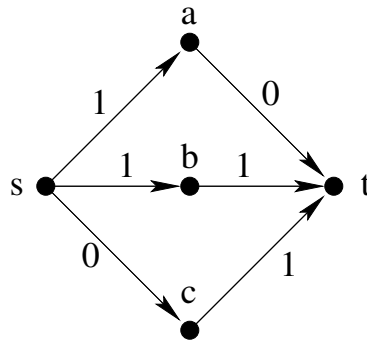
Let $G = (V, E)$ be a directed graph, with a source vertex $s \in V$ and a sink vertex $t \in V$. Assume that edges are labelled with a cost, which can be modelled as a cost function $c : E \rightarrow \mathbb{N}$ that associates a non-negative integral cost $c(e)$ to every edge $e \in E$. A (s, t) -cut (L, R) is a way of partitioning the vertices into two disjoint sets L and R , so that $L \cup R = V$, $s \in L$, and $t \in R$. (From here on, we will simply call this a cut.) The cost of a cut is the sum of the costs of the edges from L to R :

$$c(L, R) = \sum_{\substack{u \in L, v \in R \\ (u, v) \in E}} c(u, v).$$

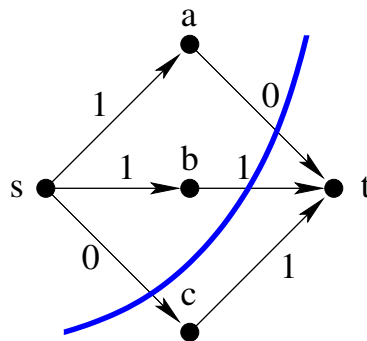
The min-cut problem is to find a minimum-cost cut in G .

Examples

Consider the following graph:

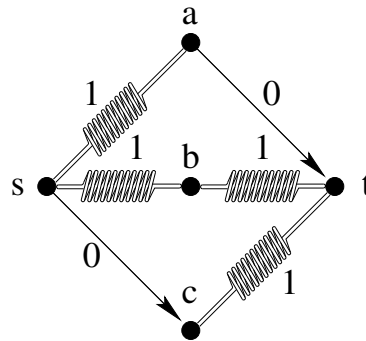


Here is a minimum-cost cut in that graph:

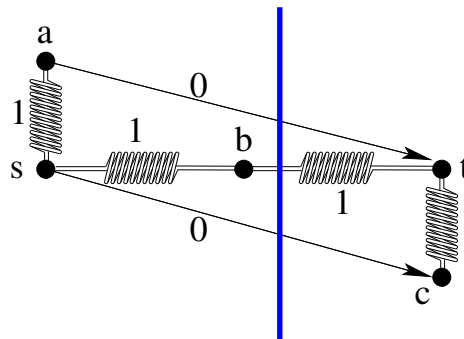


This cut is (L, R) , where $L = \{s, a, b\}$ and $R = \{t, c\}$. The cost of this cut is 1, since it cuts the edge (b, t) , which has cost 1.

Here is another way to look at it. Consider the following physical analogy. We can think of each cost-1 edge as though it were a spring that tries to contract, and each cost-0 edge as though it were an infinitely stretchy elastic band, like this:



Now imagine taking vertex s in your left hand, and vertex t in your right hand, and pulling them apart from each other as far as possible (leaving the other vertices to float freely). What's going to happen? Well, due to the tight spring connecting s and a , vertex a will try to stay as close to s as possible. Similarly, vertex c will try to stay close to t . This pulls a leftwards and c rightwards. Finally, when we draw a cut down the middle, we'll get a picture like this:

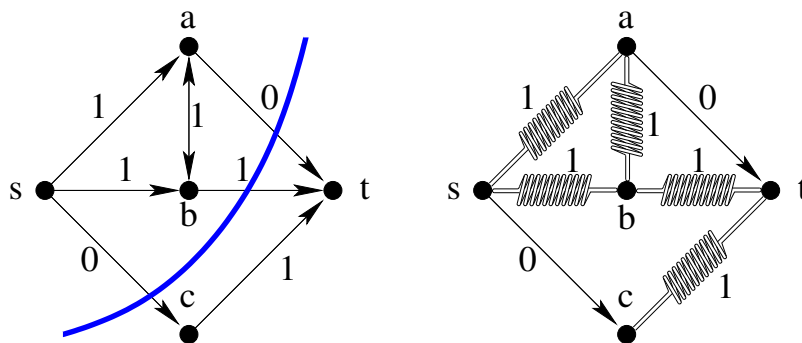


The spring connecting s and a tries to make s and a stay on the same side of the cut; since s is forced to be on the left side, this is equivalent to saying that it tries to make a be on the left side of the cut. Similarly, the spring between c and t tries to pull c towards the right side of the cut. And that's exactly what happens.

So, speaking loosely, we can think of the min-cut problem as though we have a network of springs. Each edge corresponds to a spring of some specific tension:

- An edge labelled with cost 0 places no constraints at all. It's like the edge isn't present at all.
- An edge with a positive cost is like a spring of a certain tension that tries to keep both ends of the spring on the same side of the cut. The larger the cost, the harder the spring pulls, i.e., the larger the cost on an edge, the harder we try to keep both ends of the edge on the same side of the cut.

Here is another example, to illustrate these principles. On the left is a flow network (we use a double-headed line between a and c to represent two directed edges, one from a to b and one in the reverse direction, both with the same cost), and on the right is a corresponding spring network:



In this example, we can see that the spring between s and a tries to pull a towards the left side of the cut; and the spring between a and b tries to keep these two edges on the same side of the cut, thus pulling b towards the left side of the cut. That is exactly what happens when we compute the minimal-cost cut.

However, one word of caution about the spring analogy is in order. In the actual min-cost problem, the direction of each edge matters, since we only count edges that go across the cut from left to right towards the total cost of the cut. Springs don't have a direction, and so can't model that aspect of the min-cut problem. So the spring analogy is a reasonable rough-approximation way to think of the problem, but only as a first guide to the intuition; it doesn't capture everything that is going on in the min-cut problem.

The overall intuition is: when you have an edge (u, v) with a positive cost, it tries to avoid putting u on the left side of the cut and v on the right side, since doing so would incur a cost. Put another way, an edge (u, v) of positive cost tries to either (1) put both u and v on the same side of the cut, if possible, or (2) put u on the right side of the cut and v on the left side, if possible. Of course in a graph with many edges it may not be possible to satisfy all of these competing demands, but the minimum-cost cut somehow tries to satisfy as many of them as possible.

Algorithm

There is a straightforward way to compute the min-cost cut, using a network flow algorithm. We treat the cost on each edge as a capacity, treat the graph as though it were a network of oil pipelines, and find the maximum flow from s to t in the graph. By the max-flow-min-cut theorem, the maximum-value flow corresponds to a minimum-capacity cut, whose capacity will be the same as the value of the flow.

The proof of the max-flow-min-cut theorem tells us how to explicitly identify a minimum-capacity cut. We compute the residual graph G^f , find all of the vertices that are reachable from s in the residual graph (via some sequence of edges of positive residual capacity), and put them on the left side of the cut. All the other vertices go on the right side of the cut. We proved earlier that this forms a minimal-capacity cut. Finally, note that the capacity of the cut is equal to the cost of the cut (as defined earlier). Therefore, this provides an efficient algorithm for the min-cut problem.

A minor caution: this algorithm searches for (s,t) -cuts, i.e., cuts that are constrained to place the source s on the left side of the cut and are constrained to place the sink t on the right side of the cut. (Researchers have also studied a cut problem where there is no source or sink vertex, and where we want to look at all cuts, regardless of where any vertex ends up; that is a different problem and the best algorithm for that problem turns out to be pretty different. We won't consider that problem any further here.)

Application: image segmentation

Here is a neat application of the min-cut problem. We have a $m \times n$ pixel image, $I[1..m, 1..n]$, where $I[i, j]$ denotes the color of the pixel at row i and column j . The picture includes a foreground object (say, a tree), in front of a bunch of background scenery (sky, grass, etc.). We are writing an image processing tool, and we want to identify the set of pixels associated with the foreground object.

This task turns out to be quite challenging to do in a purely automated fashion, but it becomes easier if we ask a human to help us. We ask the user to mark the area of the foreground object, as a hint. Let $H[1..m, 1..n]$ be a $m \times n$ boolean array, where $H[i, j] = \text{true}$ for each pixel (i, j) that the user has marked as part of the foreground object. The user's hints are not perfect, but we can assume that they are correct for the overwhelming majority of pixels. Also, we can assume that if two neighboring pixels have approximately the same color, then they are likely part of the same object, and thus likely either both part of the foreground, or both part of the background.

The object is to classify each pixel as either "foreground" or "background", in a way that is as consistent as possible with the user's hints and also with the same-color information. We can represent this classification as a $m \times n$ boolean matrix $F[1..m, 1..n]$, where $F[i, j] = \text{true}$ if pixel (i, j) is classified as part of the foreground, or false if that pixel is classified as part of the background. Based upon the hints above, we'll define the cost of a classification as the sum of the following charges:

- It costs \$1 for each pixel (i, j) where our classification disagrees with the user's hint, i.e., where $F[i, j] \neq H[i, j]$.
- It costs \$1 for each pair of neighboring pixels $(i, j), (i', j')$ that have a similar color but are classified differently, i.e., where $I[i, j] \approx I[i', j']$ but $F[i, j] \neq F[i', j']$.

The image segmentation problem is as follows: given I and H , find the classification F whose cost is minimal.

Here is a solution. We build a graph with one vertex for each pixel (i, j) in the image. We also add a special source vertex s and a sink vertex v , so that the graph has $nm + 2$ vertices in all. We add the following edges:

- For each pixel (i, j) that the user hinted is part of the foreground (i.e., where $H[i, j] = \text{true}$), we add an edge from s to (i, j) of cost 1.
- For each pixel (i, j) that the user hinted is part of the background (i.e., where $H[i, j] = \text{false}$), we add an edge from (i, j) to t of cost 1.

- For each pair of neighboring pixels $(i, j), (i', j')$ with a similar color (i.e., where $I[i, j] \approx I[i', j']$), we add an edge from (i, j) to (i', j') of cost 1, and an edge in the reverse direction also of cost 1.

Finally, we find the minimal-cost cut (L, R) in this graph. This cut yields a classification: we classify each pixel in L as “foreground”, and classify each pixel in R as “background.”

Note that the cost of any particular cut is the same as the cost of the corresponding classification, due to the way we have constructed the graph. For instance, if the cut puts s and (i, j) on opposite sides of the cut, where the user hinted that (i, j) is foreground (i.e., where $H[i, j] = \text{true}$), this will increase the total cost of the cut by 1, since it cuts the edge from s to (i, j) . This corresponds to a charge of \$1 for classifying a pixel (i, j) as “background” when the user has hinted it should be part of the foreground. And so on. Consequently, the minimal-cost cut is the same as the minimal-cost classification.

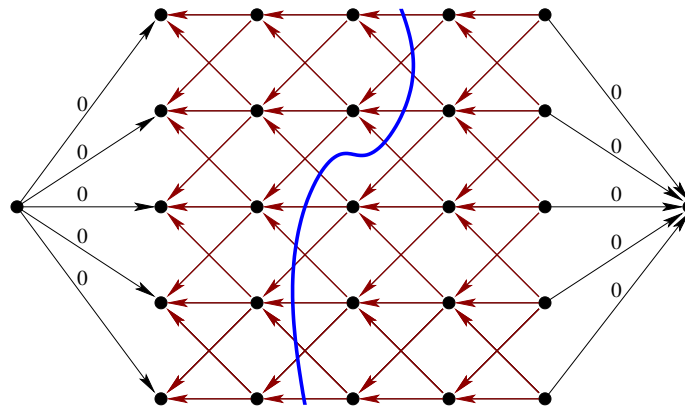
The intuition behind this algorithm is simple. The source vertex s represents the foreground, and the sink vertex t represents the background. When we want a pixel to be part of the foreground, we add an edge between s and that pixel. The effect will be to try to keep them on the same side of the cut (as though they had a spring pulling them together), and in particular, to pull that pixel to the foreground side of the cut. When we want a pixel to be part of the background, we add an edge between that pixel and t , which has the effect of trying to pull them both to the background side of the cut. Finally, when we want two neighboring pixels to be classified the same, we put an edge between them, which has the effect of trying to keep them on the same side of the cut (as though they had a spring between them). Consequently, the graph represents the constraints we are trying to satisfy, and the minimal-cost cut represents the best solution that obeys as many of the constraints as possible.

Image re-sizing

Remember the image re-sizing problem, from Question 4 of Homework 9? I will show you another solution to this problem, based upon network flow and minimum cuts.

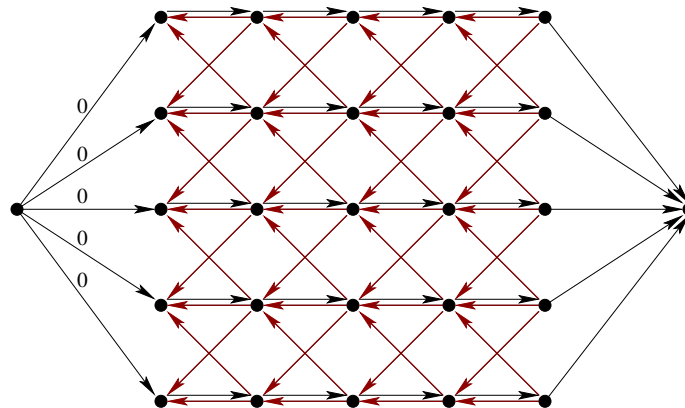
You probably remember the problem statement. We have an image $I[1..m, 1..n]$. A tear is a sequence of pixels that forms a connected path from the top of the image to the bottom of the image. Connected means that the pixel at row $i + 1$ is at most one position to the right or left of the pixel at row i (i.e., the pixel at row $i + 1$ is either southeast, south, or southwest of the pixel at row i). Deleting a tear will re-size the image to a $m \times (n - 1)$ image, yielding a new image that is one pixel narrower. For each pixel (i, j) , we are given the cost of deleting that pixel, namely, $\text{cost}(i, j)$. The cost of a tear is the sum of the costs of the pixels deleted. The goal is to find a least-cost tear.

To solve it, we’ll define a graph where the minimal-cost cut corresponds to the minimal-cost tear. The graph is a grid with one vertex for each pixel, along with a special source vertex s and a sink vertex t . The basic backbone of the graph looks like this:



In summary, the red edges enforce the constraint that a tear can only jump left or right by one position in each row.

To complete the graph we add an edge from each pixel to the neighbor on its right, shown in black below.



In other words, we have added an edge from each pixel (i, j) to its neighbor $(i, j + 1)$ to the immediate right. We assign a cost to this edge according to the cost of deleting pixel (i, j) , i.e., the cost of this edge is $\text{cost}(i, j)$. (In the case of a pixel on the rightmost column of the image, namely pixel (i, n) , we assign this cost to the edge from (i, n) to t .)

Finally, we compute the minimal-cost cut in this graph. The minimal-cost cut must have the following form. In the first row, it places the leftmost x_1 pixels, for some value x_1 , on the left side of the cut, and the remaining pixels on the right. In the second row, it places the leftmost x_2 pixels on the left and the remainder on the right, for some x_2 . And so on. As argued above, we have $|x_1 - x_2| \leq 1$, and generally, $|x_i - x_{i+1}| \leq 1$. Consequently, each cut corresponds to a valid tear, where the tear is defined to be the set of pixels immediately to the left of the tear line (one pixel per row). The cost of the cut is the sum of the right-going edges that it cuts, which corresponds to the sum of the costs of the pixels in the tear. Consequently, the minimal-cost cut in this graph is exactly the minimal-cost tear. This shows how algorithms for the minimum-cut problem can be used to resize images automatically.

Even more beautifully, this idea can be extended in a straightforward way to resizing of videos. We try to find a tear in each frame of video, such that the two tears in two consecutive frames are

shifted from each other by at most ± 1 position in each row. This can be encoded as a minimum-cut problem in a graph where the pixels form a 3-dimensional cube (rather than a 2-dimensional grid). It turns out that this provides a practical and efficient way to automatically resize videos, which is pretty nifty.