CS261: A Second Course in Algorithms
Lecture #4: Applications of Maximum Flows and Minimum Cuts*

Tim Roughgarden†

January 14, 2016

1 From Algorithms to Applications

The first three lectures covered four maximum flow algorithms (Ford-Fulkerson, Edmonds-Karp, Dinic's blocking flow-based algorithm, and the Goldberg-Tarjan push-relabel algorithm). We could talk about maximum flow algorithms til the cows come home — there has been decades of intense work on the problem, including some interesting breakthroughs just in the last couple of years. But four algorithms is enough for a course like CS261; it's time to move on to applications of the algorithms, and then on to study other fundamental problems.

Let's remind ourselves why we studied these algorithms.

1. Often the best way to get a good understanding of a computational problem is to study algorithms for it. For example, the Ford-Fulkerson algorithm introduced the crucial concept of a residual network, and gave us an excellent initial feel for the maximum flow problem.

2. These algorithms are part of the canon, among the greatest hits of algorithms. So it's fun to know how they work.

3. Maximum flow problems really do come up in practice, so it good to how you might solve them quickly. The push-relabel algorithm is an excellent starting point for implementing fast maximum flow algorithms.

The above reasons assume that we care about the maximum flow problem. And why do we care? Because like all central algorithmic problems, it directly models several well-motivated

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

problems (traffic in transportation networks, oil in a distribution network, data packets in a communication network), and also a surprising number of problems are really just maximum flow in disguise. The lecture gives two examples, in computer vision and in graph matching, and the exercise and problem sets contain several more. Perhaps the most useful skill you can learn in CS261, for both practical and theoretical work, is how to recognize when the tools of the course apply. Hopefully, practice makes perfect.

2 The Minimum Cut Problem

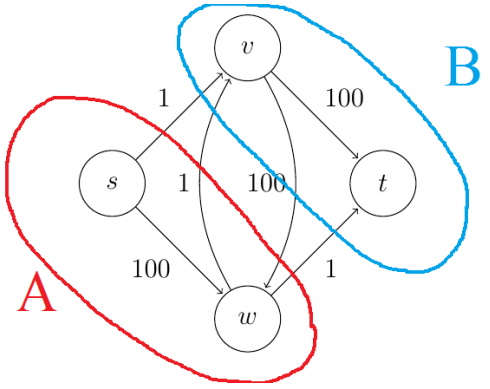

Figure 1: Example of an (s,t)-cut.

The minimum (s,t)-cut problem made a brief cameo in Lecture #2. It is the "dual" problem to maximum flow, in a sense we'll make precise in later lectures, and it is just as ubiquitous in applications. In the minimum (s,t)-cut problem, the input is the same as in the maximum flow problem (a directed graph, source and sink vertices, and edge capacities). The feasible solutions are the (s,t)-cuts, meaning the partitions of the vertex V into two sets A and B with $s \in A$ and $t \in B$ (Figure 1). The objective is to compute the s-t cut with the minimum capacity, meaning the total capacity on edges sticking out of the source-side of the cut (those sticking in don't count):

$$\text{capacity of } (A,B) = \sum_{e \in \delta^+(a)} u_e.$$

In Lecture #2 we noted a simple but extremely useful fact.

Corollary 2.1 *The minimum s-t cut problem reduces in linear time to the maximum flow problem.*

Recall the argument: given a maximum flow, just do breadth- or depth-first search from s in the residual graph (in linear time). We proved that if this search gets stuck at A, then $(A, V-A)$ is an (s,t)-cut with capacity equal to that of the flow; since no cut has capacity less than any flow, the cut $(A, V-A)$ must be a minimum cut.

While there are some algorithms for solving the minimum (s,t)-cut problem without going through maximum flows (especially for undirected graphs), in practice it is very common to solve it via this reduction. Next is an application of the problem to a basic image segmentation task.

3 Image Segmentation

3.1 The Problem

We consider the problem of classifying the pixels of an image as either foreground or background. We model the problem as follows. The input is an undirected graph $G = (V, E)$, where V is the set of pixels. The edges E designate pairs of pixels as neighbors. For example, a common input is a grid graph (Figure 2(a)), with an edge between two pixels that different by 1 in of the two coordinates. (Sometimes one also throws in the diagonals.) In any case, the solution we present works no matter than the graph G is.

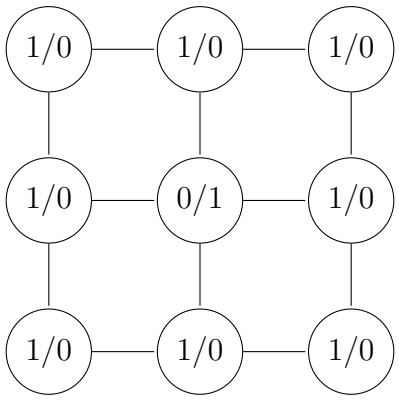

Figure 2: Example of a grid network. In each vertex, first value denotes a_v and second value denotes b_v.

The input also contains $2|V| + |E|$ parameter values. Each vertex v is annotated with two nonnegative numbers a_v and b_v, and each edge e has a nonnegative value p_e. We discuss the semantics of these shortly.

The feasible outputs are the partitions V into a foreground X and background Y; it's OK if X or Y is empty. We assess the quality of a solution by the objective function

$$\sum_{v \in X} a_v + \sum_{v \in Y} b_v - \sum_{e \in \delta(X)} p_e, \tag{1}$$

which we want to make as large as possible. ($\delta(X)$ denotes the edges cut by the partition (X, Y), with one endpoint on each side.)

We see that a vertex v earns a "prize" of a_v if it is included in X and b_v otherwise. In practice, these parameter values come from a prior as to whether a pixel v is more "likely" to be in the foreground (in which case a_v is big and b_v small) or in the background (leading to a big b_v and small a_v. It's not important for our purposes how this prior or these parameter are chosen, but it's easy to imagine examples. Perhaps a light blue pixel is typically part of the background (namely, the sky). Or perhaps one already knows a similar image that has already been segmented, like one taken earlier from the same position, and then declares that each pixel's region is likely to be the same as in the reference image.

If all we had were the a's and b's, the problem would be trivial — independently for each pixel, you would just assign it optimally to either X (if $a_v > b_v$) or Y (if $b_v > a_v$). The point of the neighboring relation E is that we also expect that images are mostly "smooth," with neighboring pixels much more likely to be in the same region than in different regions. The penalty p_e is incurred whenever the endpoints of e violate this prior belief. In machine learning terminology, the final objective (1) corresponds to a massaged version of the "maximum likelihood" objective function.

For example, suppose all p_e's are 0 in Figure 2(a). Then, the optimal solution assigns the entire boundary to the foreground and the middle pixel to the background. The objective function would be 9. If all the p_e's were 1, however, then this feasible solution would have value only 5 (because of the four cut edges). The optimal solution assigns all 9 pixels to the foreground, for a value of 8. The latter computation effectively recovers a corrupted pixel inside some homogeneous region.

3.2 Toward a Reduction

Theorem 3.1 *The image segmentation problem reduces, in linear time, to the minimum (s,t)-cut problem (and hence to the maximum flow problem).*

How would one ever suspect that such a reduction exists? The big clue is the form of the output of the image segmentation problem, as the partition of a vertex set into two pieces. This sure sounds like a cut problem. The coolest thing that could be true is that the problem reduces to a cut problem that we already know how to solve, like the minimum (s,t)-cut problem.

Digging deeper, there are several differences between image segmentation and (s,t)-cut that might give us pause (Table 1). For example, while both problems have one parameter per edge, the image segmentation problem has two parameters per vertex that seem to have no analog in the minimum (s,t)-cut problem. Happily, all of these issues can be addressed with the right reduction.

Minimum (s,t)-cut	Image segmentation
minimization objective	maximization objective
source s, sink t	no source, sink vertices
directed	undirected
no vertex parameters	a_v, b_v for each $v \in V$

Table 1: Differences between the image segmentation problem and the minimum (s,t)-cut problem.

3.3 Transforming the Objective Function

First, it's easy to convert the maximization objective function into a minimization one by multiplying through by -1:

$$\min_{(X,Y)} \sum_{e \in \delta(X)} p_e - \sum_{v \in X} a_v - \sum_{vinY} b_v.$$

Clearly, the optimal solution under this objective is the same as under the original objective.

It's hard not to be a little spooked by the negative numbers in this objective function (e.g., in max flow or min cut, edge capacities are always nonnegative). This is also easy to fix. We just shift the objective function by adding the constant value $\sum_{v \in V} a_v + \sum_{v \in V} b_v$ to every feasible solution. This gives the objective function

$$\min_{(X,Y)} \sum_{e \in \delta(X)} p_e + \sum_{v \in Y} a_v + \sum_{vinX} b_v. \tag{2}$$

Since we shifted all feasible solutions by the same amount, the optimal solution remains unchanged.

3.4 Transforming the Graph

We use tricks familiar from Exercise Set #1. Given the undirected graph $G = (V, E)$, we construct a directed graph $G' = (V', E')$ as follows:

- $V' = V \cup \{s, t\}$ (i.e., add a new source and sink)
- E' has two bidirected edges for each edge e in E (i.e., a directed edge in either direction). The capacity of both directed edges is defined to be p_e, the given penalty of edge e (Figure 3).

Figure 3: The (undirected) edges of G are bidirected in G'.

- E' also has an edge (s, v) for every pixel $v \in V$, with capacity $u_{sv} = a_v$.
- E' has an edge (v, t) for every pixel $v \in V$, with capacity $u_{vt} = b_v$.

See Figure 4 for a small example of the transformation.

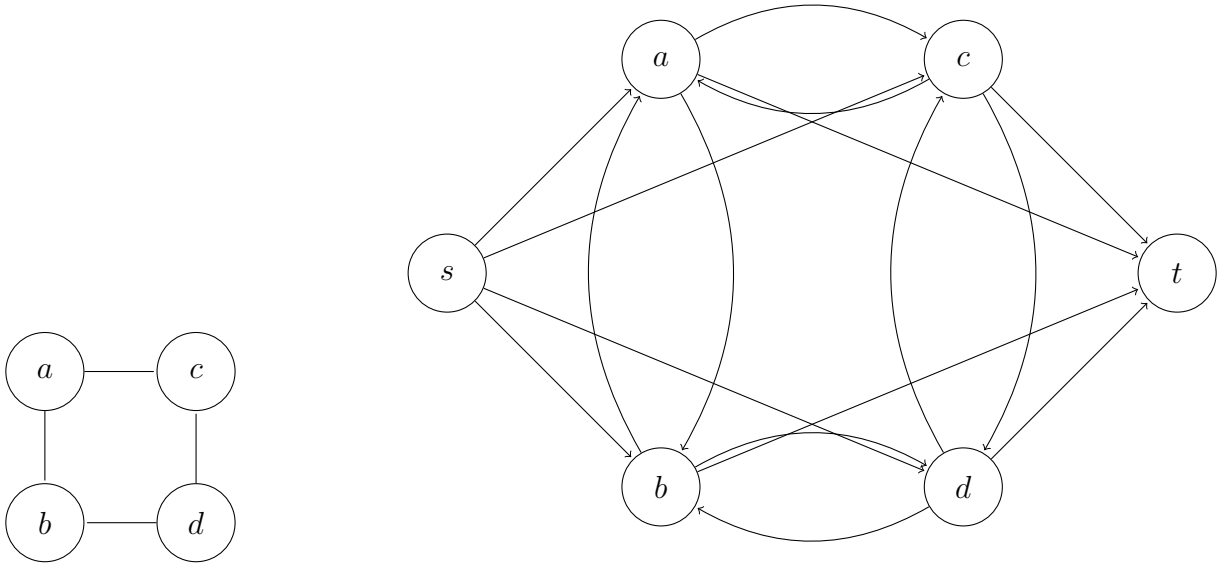

Figure 4: (a) initial network and (b) the transformation

3.5 Proof of Theorem 3.1

Consider an input $G = (V, E)$ to the image segmentation problem and directed graph $G' = (V', E')$ constructed by the reduction above. There is a natural bijection between partitions (X, Y) of V and (s, t)-cut (A, B) of G', with $A \leftrightarrow X \cup \{s\}$ and $B \leftrightarrow Y \cup \{t\}$. The key claim is that this correspondence preserves objective function value — that the capacity of every (s, t)-cut (A, B) of G' is precisely the objective function value (under (2))of the partition $(A \setminus \{s\}, B \setminus \{t\})$.

So fix an (s, t)-cut $(X \cup \{s\}, Y \cup \{t\})$ of G'. Here are the edges sticking out of $X \cup \{s\}$:

1. for every $v \in Y$, $\delta^+(X \cup \{s\})$ contains the edge (s, v), which has capacity a_v;

2. for every $v \in X$, $\delta^+(X \cup \{s\})$ contains the edge (v, t), which has capacity b_v;

3. for every edge $e \in \delta(X)$, $\delta^+(X \cup \{s\})$ contains exactly one of the two corresponding directed edges of G' (the other one goes backward), and it has capacity p_e.

These are precisely the edges of $\delta^+(X \cup \{s\})$. We compute the cut's capacity just be summing up, for a total of

$$\sum_{v \in Y} a_v + \sum_{v \in X} b_v + \sum_{e \in \delta(X)} p_e.$$

This is identical to the objective function value (2) of the partition (X, Y). We conclude that computing the optimal such partition reduces to computing a minimum (s, t)-cut of G'. The reduction can be implemented in linear time.

4 Bipartite Matching

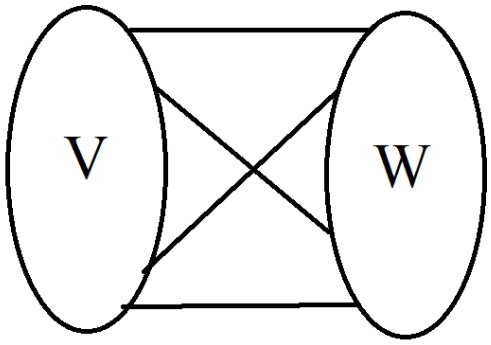

Figure 5: Visualization of bipartite graph. Edges exist only between the partitions V and W.

We next give a famous application of maximum flow. This application also serves as a segue between the first two major topics of the course, the maximum flow problem and graph matching problems.

In the bipartite matching problem, the input is an undirected bipartite graph $G = (V \cup W, E)$, with every edge of E having one endpoint in each of V and W. That is, no edges internal to V or W are allowed (Figure 5). The feasible solutions are the *matchings* of the graph, meaning subsets $S \subseteq E$ of edges that share no endpoints. The goal of the problem is to compute a matching with the maximum-possible cardinality. Said differently, the goal is to pair up as many vertices as possible (using edges of E).

For example, the square graph (Figure 6(a)) is bipartite, and the maximum-cardinality matching has size 2. It matches all of the vertices, which is obviously the best-case scenario. Such a matching is called *perfect*.

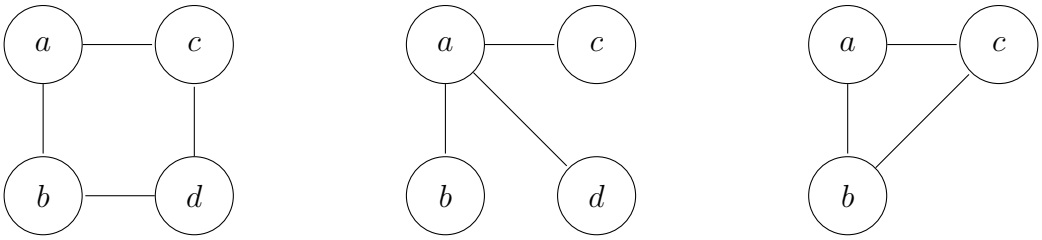

Figure 6: (a) square graph with perfect matching of 2. (b) star graph with maximum-cardinality matching of 1. (c) non-bipartite graph with maximum matching of 1.

Not all graphs have perfect matchings. For example, in the star graph (Figure 6(b)), which is also bipartite, no many how many vertices there are, the maximum-cardinality matching has size only 1.

It's also interesting to discuss the maximum-cardinality matching problem in general (non-bipartite) graphs (like Figure 6(c)), but this is a harder topic that we won't cover here. While one can of course consider the bipartite special case of any graph problem, in matching bipartite graphs play a particularly fundamental role. First, matching theory is nicer and matching algorithms are faster for bipartite graphs than for non-bipartite graphs. Second, a majority of the applications of already in the bipartite special case — assigning workers to jobs, courses to room/time slots, medical residents to hospitals, etc.

Claim: maximum-cardinality matching reduces in linear time to maximum flow.

Proof sketch: Given an undirected bipartite graph $(V \cup W, E)$, construct a directed graph G' as in Figure 7(b). We add a source and sink, so the new vertex set is $V' = V \cup W \cup \{s, t\}$. To obtain E' from E, we direct all edges of G from V to W and also add edges from s to every vertex of V and from every vertex of W to t. Edges incident to s or t have capacity 1, reflecting the constraints the each vertex of $V \cup W$ can only be matches to one other vertex. Each edge (v, w) directed from V to W can be given any capacity that is at least 1 (v can only receive one unit of flow, anyway); for simplicity, give all these edges infinite capacity.

You should check that there is a one-to-one correspondence between matchings of G and integer-valued flows in G', with edge (v, w) corresponding to one unit of flow on the path $s \to v \to w \to t$ in G' (Figure 7). This bijection preserves the objective function value. Thus, given an integral maximum flow in G', the edges from V to W that carry flow form a maximum matching.[1]

[1] All of the maximum flow algorithms that we've discussed return an integral maximum flow provided all the edge capacities are integers. The reason is that inductively, the current (pre)flow, and hence the residual capacities, and hence the augmentation amount, stay integral throughout these algorithms.

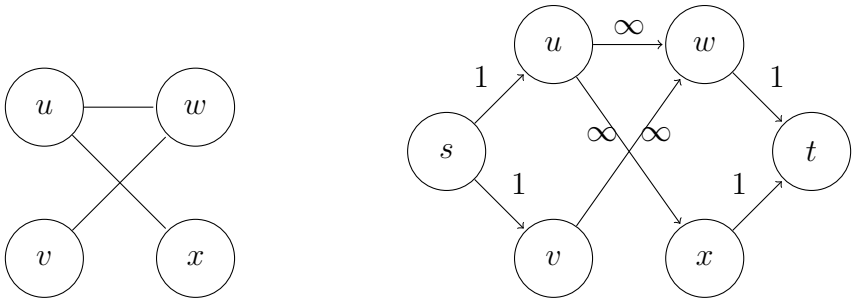

Figure 7: (a) original bipartite graph G and (b) the constructed directed graph G. There is one-to-one correspondence between matchings of G and integer-valued flows of G' e.g. (v, w) in G corresponds to one unit of flow on $s \to v \to w \to t$ in G'.

5 Hall's Theorem

In this final section we tie together a number of courses ongoing themes. We previously asked the question

> How do we know when we're done (i.e., optimal)?

for the maximum flow problem. Let's ask it again for the maximum-cardinality bipartite matching problem. Using the reduction in Section 4, we can translate the optimality conditions for the maximum flow problem (i.e., the max-flow/min-cut theorem) into a famous optimality condition for bipartite matchings.

Consider a bipartite graph $G = (V \cup W, E)$ with $|V| \leq |W|$, renaming V, W if necessary. Call a matching of G *perfect* if it matches every vertex in V; clearly, a perfect matching is a maximum matching. Let's first understand which bipartite graphs admit a perfect matching.

Some notation: for a subset $S \subseteq V$, let $N(S)$ denote the union of the neighborhoods of the vertices of S: $N(S) = \{w \in W : \exists v \in S \text{ s.t. } (v, w) \in E\}$. See Figure 8 for two examples of such neighbor sets.

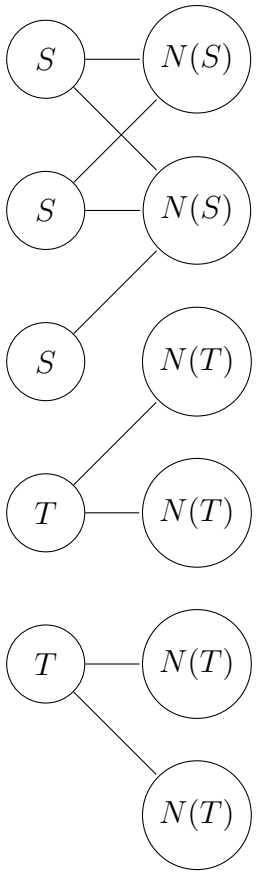

Figure 8: Two examples of vertex sets S and T and their respective neighbour sets $N(S)$ and $N(T)$.

Does the graph in Figure 8 have a perfect matching? A little thought shows that the answer is "no." The three vertices of S have only two distinct neighbors between them. Since each vertex can only be matched to one other vertex, there is no hope of matching more than two of the three vertices of S.

More generally, if a bipartite graph has a *constricting set* $S \subseteq V$, meaning one with $|N(S)| < |S|$, then it has no perfect matching. But what about the converse? If a bipartite graph admits no perfect matching, can you always find a short convincing argument of this fact, in the form of a constricting set? Or could there be obstructions to perfect matchings beyond just constricting sets? *Hall's Theorem* gives the beautiful answer that constricting sets are the only obstacles to perfect matchings.[2]

Theorem 5.1 (Hall's Theorem) *A bipartite graph $(V \cup W, E)$ with $|V| \leq |W|$ has a perfect matching if and only if, for every subset $S \subseteq V$, $|N(S)| \geq |S|$.*

[2]Hall's theorem actually predates the max-flow/min-cut theorem by 20 years.

Thus, it's not only easy to convince someone that a graph has a perfect matching (just exhibit a matching), it's also easy to convince someone that a graph does *not* have a perfect matching (just exhibit a constricting set).

Proof of Theorem 5.1: We already argued the easy "only if" direction. For the "if" direction, suppose that $|N(S)| \geq |S|$ for every $S \subseteq V$.

Claim: in the flow network G' that corresponds to G (Figure 7), every (s,t)-cut has capacity at least $|V|$.

To see why the claim implies the theorem, note that it implies that the minimum cut value in G' is at least $|V|$, so the maximum flow in G' is at least $|V|$ (by the max-flow/min-cut theorem), and an integral flow with value $|V|$ corresponds to a perfect matching of G.

Proof of claim: Fix an (s,t)-cut (A,B) of G'. Let $S = A \cap V$ denote the vertices of V that lie on the source side. Since $s \in A$, all (unit-capacity) edges from s to vertices of $V - A$ contribute to the capacity of (A,B). Recall that we gave the edges directed from V to W infinite capacity. Thus, if some vertex w of $N(S)$ fails to also be in A, then the cut (A,B) has infinite capacity (because of the edge from S to w) and there is nothing to prove. So suppose all of $N(S)$ belongs to A. Then all of the (unit-capacity) edges from vertices of $N(S)$ to t contribute to the capacity of (A,B). Summing up, we have

$$\begin{aligned} \text{capacity of } (A,B) &\geq \underbrace{(|V| - |S|)}_{\text{edges from } s \text{ to } V-S} + \underbrace{|N(S)|}_{\text{edges from } N(S) \text{ to } t} \\ &\geq |V|, \end{aligned} \tag{3}$$

where (3) follows from the assumption that $|N(S)| \geq |S|$ for every $S \subseteq V$. ■

On Exercise Set #2 you will extend this proof to show that, more generally, for every bipartite graph $(V \cup W, E)$ with $|V| \leq |W|$,

$$\text{size of maximum matching} = \min_{S \subseteq V} \left(|V| - (|S| - |N(S)|)\right).$$

Note that at least $|S| - |N(S)|$ vertices of S are unmatched in every matching.