CS125	Lecture 10

Network Flows

Suppose that we are given the network in top of Figure 10.1, where the numbers indicate capacities, that is, the amount of flow that can go through the edge in unit time. We wish to find the maximum amount of flow that can go through this network, from S to T.

This problem can also be reduced to linear programming. We have a nonnegative variable for each edge, representing the flow through this edge. These variables are denoted $f_{SA}, f_{SB}, \ldots$ We have two kinds of constraints: capacity constraints such as $f_{SA} \leq 5$ (a total of 9 such constraints, one for each edge), and flow conservation constraints (one for each node except S and T), such as $f_{AD} + f_{BD} = f_{DC} + f_{DT}$ (a total of 4 such constraints). We wish to maximize $f_{SA} + f_{SB}$, the amount of flow that leaves S, subject to these constraints. It is easy to see that this linear program is equivalent to the max-flow problem. The simplex method would correctly solve it.

In the case of max-flow, it is very instructive to "simulate" the simplex method, to see what effect its various iterations would have on the given network. Simplex would start with the all-zero flow, and would try to improve it. How can it find a small improvement in the flow? Answer: it finds a path from S to T (say, by depth-first search), and moves flow along this path of total value equal to the *minimum* capacity of an edge on the path (it can obviously do no better). This is the first iteration of simplex (see Figure 10.1).

How would simplex continue? It would look for another path from S to T. Since this time we already partially (or totally) use some of the edges, we should do depth-first search on the edges that have some *residual capacity*, above and beyond the flow they already carry. Thus, the edge CT would be ignored, as if it were not there. The depth-first search would now find the path $S - A - D - T$, and augment the flow by two more units, as shown in Figure 10.1.

Next, simplex would again try to find a path from S to T. The path is now $S - A - B - D - T$ (the edges $C - T$ and $A - D$ are full are are therefore ignored), and we augment the flow as shown in the bottom of Figure 10.1.

Next simplex would again try to find a path. But since edges $A - D$, $C - T$, and $S - B$ are full, they must be ignored, and therefore depth-first search would fail to find a path, after marking the nodes S, A, C as reachable from S. *Simplex then returns the flow shown, of value 6, as maximum.*

How can we be sure that it is the maximum? Notice that these reachable nodes define a *cut* (a set of nodes

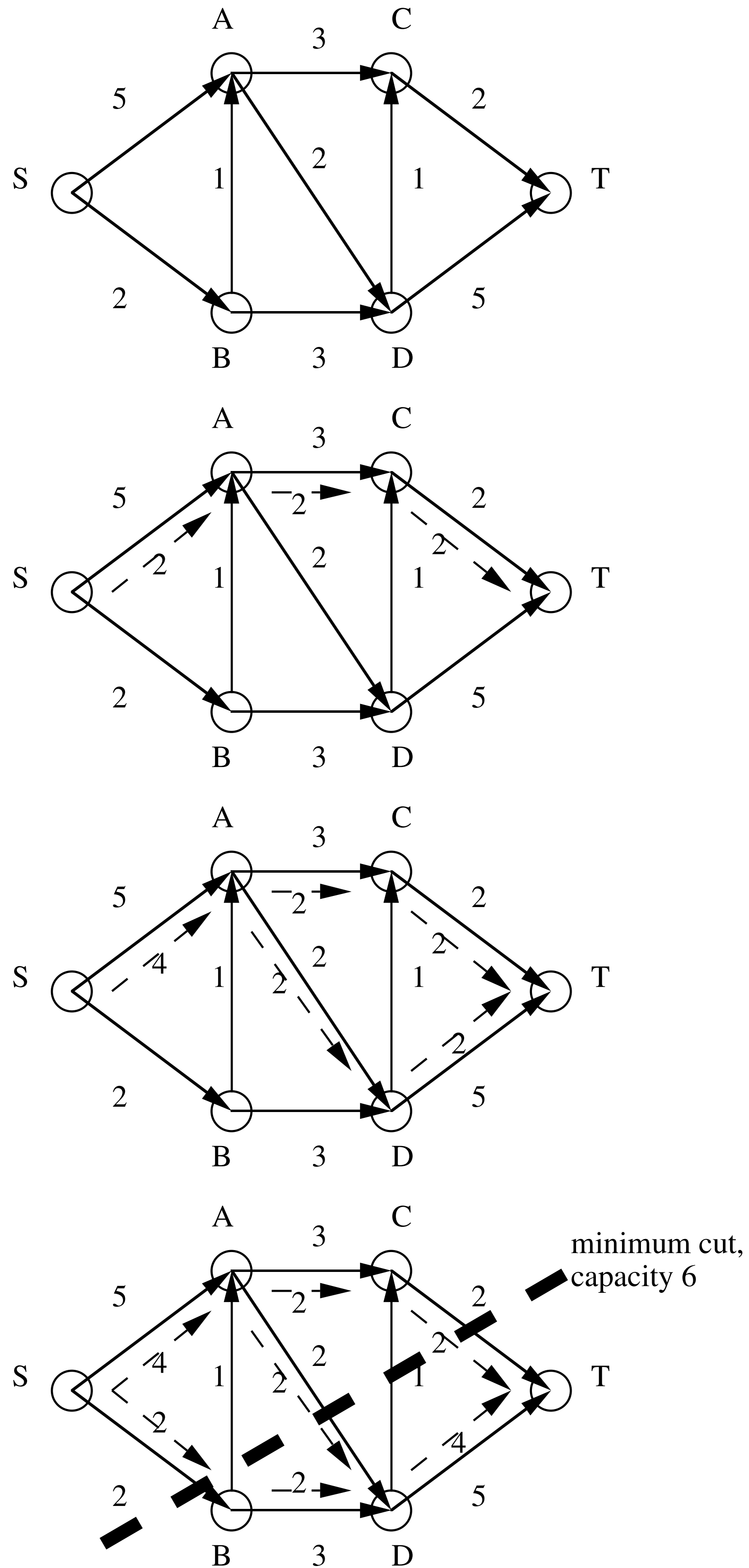

Figure 10.1: Max flow

containing S but not T), and the *capacity* of this cut (the sum of the capacities of the edges going out of this set) is 6, the same as the max-flow value. (It must be the same, since this flow passes through this cut.) The existence of this cut establishes that the flow is optimum!

There is a complication that we have swept under the rug so far: when we do depth-first search looking for a path, we use not only the edges that are not completely full, but we must also traverse *in the opposite direction* all edges that already have some non-zero flow. This would have the effect of canceling some flow; canceling may be necessary to achieve optimality, see Figure 10.2. In this figure the only way to augment the current flow is via the path $S-B-A-T$, which traverses the edge $A-B$ in the reverse direction (a legal traversal, since $A-B$ is carrying non-zero flow).

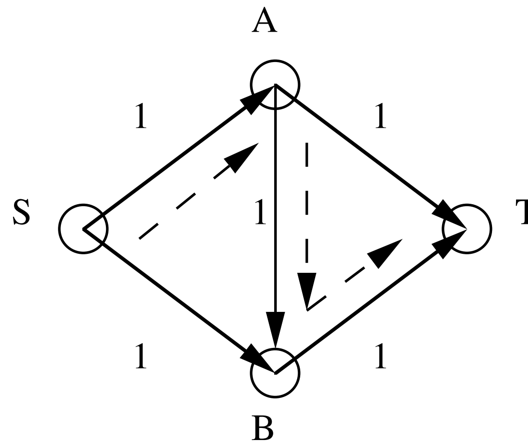

Figure 10.2: Flows may have to be canceled

In general, a path from the source to the sink along which we can increase the flow is called an *augmenting path*. We can look for an augmenting path by doing for example a depth first search along the *residual network*, which we now describe. For an edge (u,v), let $c(u,v)$ be its capacity, and let $f(u,v)$ be the flow across the edge. Note that we adopt the following convention: if 4 units flow from u to v, then $f(u,v) = 4$, and $f(v,u) = -4$. That is, we interpret the fact that we could reverse the flow across an edge as being equivalent to a “negative flow”. Then the *residual capacity* of an edge (u,v) is just

$$c(u,v) - f(u,v).$$

The residual network has the same vertices as the original graph; the edges of the residual network consist of all weighted edges with strictly positive residual capacity. The idea is then if we find a path from the source to the sink in the residual network, we have an augmenting path to increase the flow in the original network. As an exercise, you may want to consider the residual network at each step in Figure 10.1.

Suppose we look for a path in the residual network using depth first search. In the case where the capacities are integers, we will always be able to push an integral amount of flow along an augmenting path. Hence, if the maximum flow is f^*, the total time to find the maximum flow is $O(mf^*)$, since we may have to do an $O(m)$ depth first search up to f^* times. This is not so great.

Note that we do not have to do a depth-first search to find an augmenting path in the residual network. In fact, using a breadth-first search each time yields an algorithm that provably runs in $O(nm^2)$ time, regardless of whether or not the capacities are integers. We will not prove this here. There are also other algorithms and approaches to the max-flow problem as well that improve on this running time.

To summarize: the max-flow problem can be easily reduced to linear programming and solved by simplex. But it is easier to understand what simplex would do by following its iterations directly on the network. It repeatedly finds a path from a source s to a sink t along edges that are not yet full (have non-zero residual capacity), and also along any reverse edges with non-zero flow. If an s-t path is found, we augment the flow along this path, and repeat. When a path cannot be found, the set of nodes reachable from s defines a cut of capacity equal to the max-flow. Thus, *the value of the maximum flow is always equal to the capacity of the minimum cut*. This is the important *max-flow min-cut theorem*. One direction (that max-flow$\leq$min-cut) is easy (think about it: *any* cut is larger than *any* flow); the other direction is proved by taking advantage of the algorithm just described.

Duality Again

As it turns out, the max-flow min-cut theorem is a special case of a more general phenomenon called *duality*. Recall duality means that for each maximization problem there is a corresponding minimizations problem with the property that any feasible solution of the min problem is greater than or equal any feasible solution of the max problem. Furthermore, and more importantly, *they have the same optimum*.

Consider the network shown in Figure 10.3, and the corresponding max-flow problem. We know that it can be written as a linear program as follows:

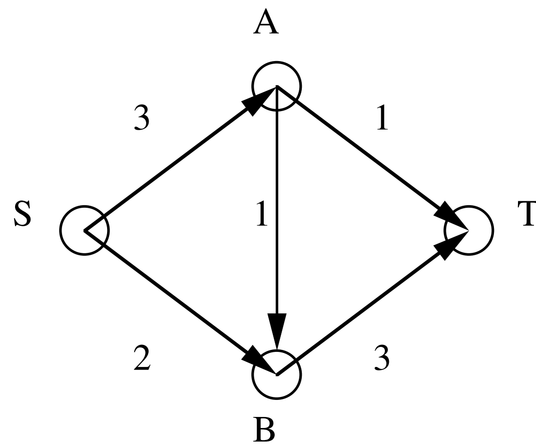

Figure 10.3: A simple max-flow problem

$$
\begin{array}{lcccccl}
\max & f_{SA} & +f_{SB} & & & & \\
 & f_{SA} & & & & & \leq 3 \\
 & & f_{SB} & & & & \leq 2 \\
 & & & f_{AB} & & & \leq 1 \\
 & & & & f_{AT} & & \leq 1 \\
 & & & & & f_{BT} & \leq 3 \\
 & f_{SA} & & -f_{AB} & -f_{AT} & & = 0 \\
 & & f_{SA} & +f_{AB} & & -f_{BT} & = 0 \\
 & & & & & & f \geq 0
\end{array}
\qquad P
$$

Consider now the following linear program:

$$
\begin{array}{lccccccl}
\min 3y_{SA} & +2y_{SB} & +y_{AB} & +y_{AT} & +3y_{BT} & & & \\
y_{SA} & & & & & +u_A & & \geq 1 \\
 & y_{SB} & & & & & +u_B & \geq 1 \\
 & & y_{AB} & & & -u_A & +u_B & \geq 0 \\
 & & & y_{AT} & & -u_A & & \geq 0 \\
 & & & & y_{BT} & & -u_B & \geq 0 \\
 & & & & & & & y \geq 0
\end{array}
\qquad D
$$

This LP describes the *s*-*t* min-cut problem (or at least, a fractional version)! To see why, suppose that the u_A

variable is meant to be 1 if A is in the cut with S, and 0 otherwise, and similarly for B (naturally, by the definition of a cut, S will always be with S in the cut, and T will never be with S). Each of the y variables is to be 1 if the corresponding edge contributes to the cut capacity, and 0 otherwise. Then the constraints make sure that these variables behave exactly as they should. For example, the second constraint states that *if A is not with S, then SA must be added to the cut.* The third one states that *if A is with S and B is not* (this is the only case in which the sum $-u_A + u_B$ becomes -1), *then AB must contribute to the cut.* And so on. Although the y and u's are free to take values larger than one, they will be "slammed" by the minimization down to 1 or 0. Of course in reality, our linear program solver might screw things up for us by setting some of the y or u variables to be fractions as opposed to integers in $\{0,1\}$. It turns out that even in this case, there is a randomized rounding algorithm which rounds such fractional solutions to $\{0,1\}$ in such a way that the expectation of the objective function does not increase, implying in particular that there is an optimal solution which corresponds to an actual cut. We will not give the proof in these notes however.

These two linear programs are in fact, duals of each other. This fact is most easily seen by putting the linear programs in matrix form. The first program, which we call the primal (P), we write as:

max	1	1	0	0	0		
	1	0	0	0	0	$\leq$	3
	0	1	0	0	0	$\leq$	2
	0	0	1	0	0	$\leq$	1
	0	0	0	1	0	$\leq$	1
	0	0	0	0	1	$\leq$	3
	1	0	-1	-1	1	$=$	0
	0	1	1	0	-1	$=$	0
	$\geq$	$\geq$	$\geq$	$\geq$	$\geq$		

Here we have removed the actual variable names, and we have included an additional row at the bottom denoting that all the variables are non-negative. (An unrestricted variable will be denoted by unr.

The second program, which we call the dual (D), we write as:

min	3	2	1	1	3	0	0		
	1	0	0	0	0	1	0	$\geq$	1
	0	1	0	0	0	0	1	$\geq$	1
	0	0	1	0	0	-1	1	$\geq$	0
	0	0	0	1	0	-1	0	$\geq$	0
	0	0	0	0	1	0	-1	$\geq$	0
	$\geq$	$\geq$	$\geq$	$\geq$	$\geq$	unr	unr		

Each variable of P corresponds to a constraint of D, and vice-versa. Equality constraints correspond to unrestricted variables (the u's), and inequality constraints to restricted variables. Minimization becomes maximization. The matrices are transpose of one another, and the roles of right-hand side and objective function are interchanged.

Repeating from the past lecture notes, it is a mechanical process, given an LP, to form its dual. Suppose we start with a maximization problem. Change all inequality constraints into $\leq$ constraints, negating both sides of an equation if necessary. Then

- transpose the coefficient matrix
- invert maximization to minimization
- interchange the roles of the right-hand side and the objective function
- introduce a nonnegative variable for each inequality, and an unrestricted one for each equality
- for each nonnegative variable introduce a $\geq$ constraint, and for each unrestricted variable introduce an equality constraint.

As we have stated before, an LP and its dual, as long as both have bounded optimum solutions, will have the same optimum value by strong duality. The max-flow min-cut theorem could therefore also have been derived by showing that, in general, the max-flow linear program and fractional min-cut linear program are always dual linear programs, and then by exhibiting the randomized rounding algorithm described above.

Matching

It is often useful to *compose* reductions. That is, we can reduce a problem A to B, and B to C, and since C we know how to solve, we end up solving A. A good example is the matching problem.

Suppose that the *bipartite* graph shown in Figure 10.4 records the compatibility relation between four boys and four girls. We seek a maximum matching, that is, a set of edges that is as large as possible, and in which no two edges share a node. For example, in Figure 10.4 there is a *complete* matching (a matching that involves all nodes).

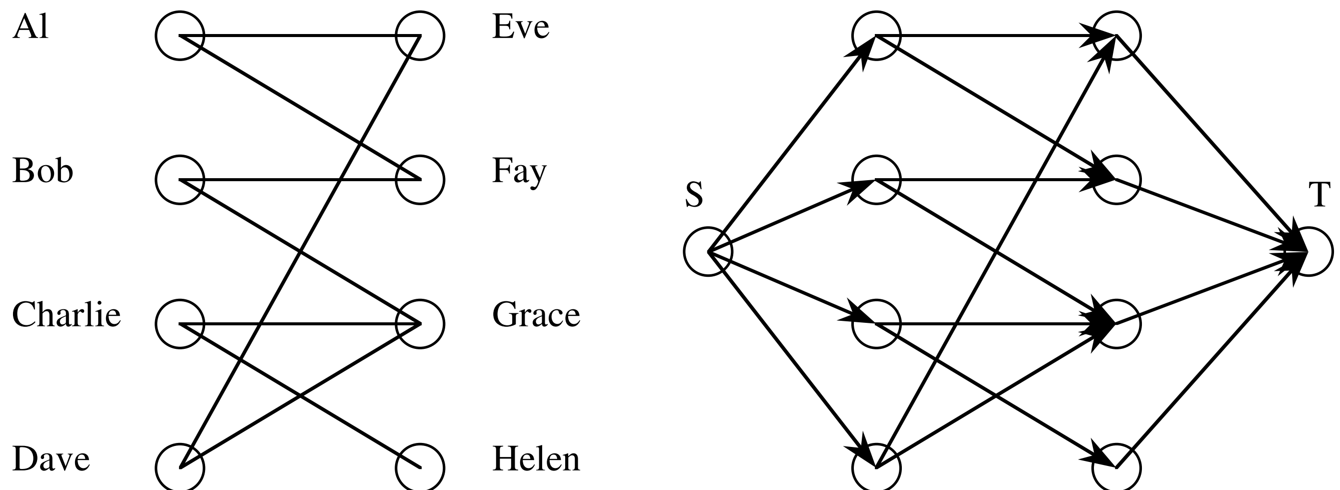

Figure 10.4: Reduction from matching to max-flow (all capacities are 1)

To reduce this problem to max-flow, we create a new source and a new sink, connect the source with all boys and all girls with the sinks, and direct all edges of the original bipartite graph from the boys to the girls. All edges have capacity one. It is easy to see that the maximum flow in this network corresponds to the maximum matching.

Well, the situation is slightly more complicated than was stated above: what is easy to see is that the optimum *integer-valued* flow corresponds to the optimum matching. We would be at a loss interpreting as a matching a flow that ships .7 units along the edge Al-Eve! Fortunately, what the algorithm in the previous section establishes is that *if the capacities are integers, then the maximum flow is integer.* This is because we only deal with integers throughout the algorithm. Hence *integrality comes for free in the max-flow problem*.

Unfortunately, max-flow is about the only problem for which integrality comes for free. It is a very difficult problem to find the optimum solution (or *any* solution) of a general linear program with the additional constraint that (some or all of) the variables be integers.

Another max flow application: baseball

Suppose there are n baseball teams, and team 1 is our favorite. It is the middle of baseball season, and some games have been played but not all have. We would like to know: is it possible for the outcome of the remaining games to be such that team 1 ends up winning the season? Or is team 1 basically "drawing dead", i.e. there is mathematically no way possible for team 1 to win the season? For the purposes of this problem, winning the season means having the most number of wins in the league once all games are complete (if there's a tie, all tied teams are winners!).

Clearly to make team 1 win the season, we should make it win all its remaining games. But then, what about the other games in which team 1 is not involved? For each of those games, *one* of the two teams involved in that game will be the winner, so who should we make the winner be in each of these games to make sure that no team accumulates so many wins that it surpasses team 1? It turns out that we can determine the feasibility of distributing these wins to make team 1 the season winner by solving the maximum flow in a particular graph and checking whether it's big enough! For details, see the excellent notes on the subject in Section 24.5 of `http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/24-maxflowapps.pdf`.

Global min-cut

Above we saw how to solve the $s-t$ min-cut using a maximum flow computation. Recall a *cut* is a partition of the vertices V into non-empty sets S and $V\backslash S$. An $s-t$ min-cut is a cut for which $s \in S$ and $t \notin S$.

Consider the following global min-cut problem: we are given an undirected, unweighted graph $G = (V,E)$, and we would like to find a cut $C = (S, V\backslash S)$ with value as small as possible. The *weight* of a cut $w(C)$ is simply the number of edges with one endpoint in S and the other endpoint in $V\backslash S$. One motivation for studying minimum cut is that it is some measure of the reliability of a network. Imagine that each vertex represents a network switch, and edges corresond direct links between switches. Let us assume our graph is connected (which hopefully our communication network is!). Then the weight of a minimum cut answers the following question: *what is the minimum number of links that need to go down in our network to make our graph disconnected, i.e. so that not every switch can still talk to every other switch*? The minimum cut doesn't deal with switches themselves failing (i.e. vertex failures), but that's a problem we won't consider today.

There is an obvious brute-force algorithm for the minimum cut problem: try all cuts then take the one with the minimum weight. There are $2^{n-1}-1$ cuts to try. To see this, write $V = \{1,\ldots,n\}$ and put cuts in correspondence with length-$(n-1)$ binary strings. The ith bit (1-indexed) is 1 if vertex $i+1$ is on the same side of the cut as vertex 1. There are 2^{n-1} such strings, but the all-1s string is not allowed since that would mean one side of the cut would

be empty.

A more efficient solution follows by observing that the global min-cut problem can be solved by reduction to $n-1$ separate *s*-*t* min-cut problems. In particular, we know that in any min-cut, the vertex $s=1$ must be in a different partition for *some* vertex $t \in V \backslash \{s\}$. We thus solve the problem by first making the graph directed and capacitated (for each undirected edge $e=(u,v)$, replace it with two directed edges (u,v) and (v,u) each with capacity 1). We then compute the minimum $s-t$ cut for $s=1$ and each $t \neq s$; that is, we perform $n-1$ minimum s-t cut computations. We then return the smallest of all the cuts found. Since none of these minimum cuts have value more than $n-1$ (one can always separate s from t by removing all neighbors of s), the runtime of Ford-Fulkerson for each choice of t is $O(mn)$. Thus the overall runtime is $O(mn^2)$.

Another approach is to use a randomized algorithm. Here we will demonstrate the power of randomness in designing simple algorithms.

Karger's contraction algorithm for global min-cut. A more efficient Monte Carlo algorithm, due to David Karger (a Harvard alum '89!) [Karger93], is the following (to say that an algorithm is *Monte Carlo* means that its running time is bounded by some deterministic quantity with probability 1, but its probability of returning an incorrect answer is nonzero). It is known as Karger's contraction algorithm. In what follows, G may be a multigraph (i.e. there may be multiple parallel edges between the same two vertices; even if G doesn't start this way, it may become this way in later recursive calls).

procedure CONTRACT($G=(V,E)$)
    if $|V|=2$ then output the only cut
    else
      pick a random edge e
      contract the edge e to form $G'=(V',E')$
      return CONTRACT(G')
end

What does it mean to *contract* an edge $e=(u,v)$? It means we remove the vertices (u,v) from the graph and insert a new vertex w. Any edge in G of the form (u,x) for $x \neq v$ is replaced by the edge (w,x). Similarly any edge of the form (v,x) for $x \neq u$ is replaced by (w,x). All edges of the form (u,v) are removed. Figure 10.5 gives an example of obtaining some G' by contracting an edge in G. Essentially when we contract (u,v), we are saying that we are promising to place u,v on the same side of the final cut we output. Thus when $|V|=2$ in the last level of recursion, the two vertices actually represent the vertices placed into the two sides of the cut.

Lemma 10.1 *Let C be some minimum cut of G with $w(C)=k$. Then the probability that Karger's contraction algorithm outputs C is at least $1/\binom{n}{2}$.*

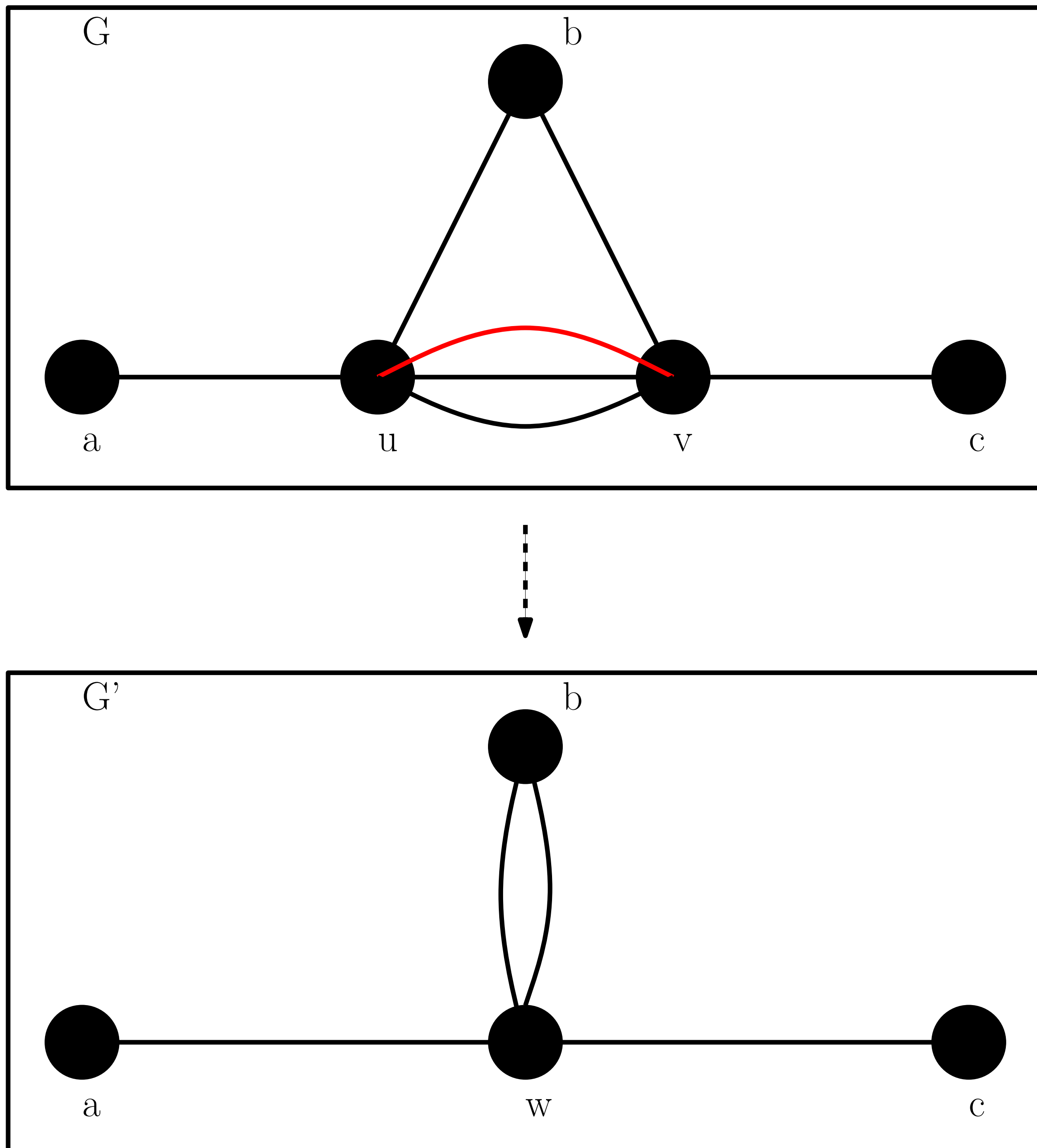

Figure 10.5: Contracting one of the (u, v) edges highlighted in red to merge u, v into w.

Proof: We say C survives the ith level of recursion if no edge in C is contracted in recursive level i. Then C is output as long as C survives every level. Thus

$$\Pr(C \text{ is output}) = \Pr(C \text{ survives every recursive level}) = \prod_{i=0}^{|V|-2} \Pr(C \text{ survives level } i | C \text{ survived levels } 0,1,\ldots,i-1)$$

where $\Pr(A|B)$ is the conditional probability of event A, given that event B occurred. Let m_i be the number of edges in our current graph G_i in recursive level i (note this graph is a multigraph: there can be parallel edges between vertices). Then the probability C survives recursive level i conditioned on it having already survived up until this point is $1-|C|/m_i = 1-k/m_i$. Since C has survived up until this point, the minimum cut of G_i is also still k. Thus the minimum degree of any vertex in G_i is at least k, since otherwise the cut separating a low-degree vertex from everyone else would have smaller cut value. Since m_i is just half the sum of all degrees, we have $m_i \ge n_i k/2 = (n-i)k/2$. Thus $1-k/m_i \ge 1-2/(n-i)$. Thus

$$\Pr(C \text{ is output}) \ge \left(1-\frac{2}{n}\right)\cdot\left(1-\frac{2}{n-1}\right)\cdot\ldots\cdot\frac{1}{3} = \frac{n-2}{n}\cdot\frac{n-3}{n-1}\cdot\frac{n-4}{n-2}\cdot\frac{n-5}{n-3}\cdot\ldots\cdot\frac{3}{5}\cdot\frac{2}{4}\cdot\frac{1}{3} = \frac{2}{n(n-1)}. \quad (10.1)$$

due to cancellation. ■

You might ask: can we improve the $1/\binom{n}{2}$ success probability? The answer is no as the lemma is stated, since if there are t minimum cuts then one of them will be output with probability at most $1/t$. We can obtain $t=\binom{n}{2}$ by letting G be the cycle on n vertices (it has $\binom{n}{2}$ minimum cuts: namely choose any 2 of its n edges to cut).

Now, $1/\binom{n}{2}$ success probability seems low, but we can alleviate this by running the algorithm $\binom{n}{2}$ times and outputting the smallest weight cut we ever find. Then the probabilty that we never see a particular minimum cut is at most $(1-1/\binom{n}{2})^{\binom{n}{2}} \le 1/e$, using our favorite approximation $1+x \le e^x$ for all real x. Thus if we repeat this $\lceil \ln(1/P) \rceil$ times for some $0<P<1/2$, the probability of never seeing a minimum cut is at most $e^{-\ln(1/P)} = P$.

We also leave it as an exercise to show that the contraction algorithm above can be implemented to run in time $O(n^2)$, so that overall the running time when repeating $\binom{n}{2}\lceil \ln(1/P) \rceil$ times is $O(n^4 \log(1/P))$. Note this is slightly slower than the $n-1$ calls to Ford-Fulkerson mentioned above (by the $\log(1/P)$ factor), but is *much* simpler.

Karger-Stein Shortly after Karger's contraction algorithm described above, Karger and Stein developed a variant with improved running time [KargerS93]. The key observation comes from staring at (10.1). Notice that in the first round of contraction the probability that C survives is pretty good: at least $1-2/n$. Meanwhile toward the end, the probability C survives starts getting much farther away from one: we can only say it is at least $1/3$ in the last iteration.

The idea of Karger and Stein was to run contraction not until we get down to two vertices, but rather until we get down to t vertices. Then the probability that C has still survived up until this point, by (10.1), is at least $t(t-1)/(n(n-1))$. By picking $t=\lceil n/\sqrt{2} \rceil + 1$, we have that the probability that C survives after contracting down to t vertices is at least $1/2$. The Karger-Stein algorithm, henceforth known as KS, is then as follows:

procedure KS($G=(V,E)$)
    if $|V|<6$ then try all $2^{|V|-1}$ cuts and return the smallest one
    else
        $t \leftarrow \lceil n/\sqrt{2} \rceil + 1$
        Obtain G_1 by contracting G $n-t$ times to obtain t vertices
        Obtain G_2 by contracting G $n-t$ times to obtain t vertices (using independent randomness from last step)

$C_1 \leftarrow KS(G_1)$
$C_2 \leftarrow KS(G_2)$
return the smaller of the two cuts C_1 and C_2
end

The reason we make $|V| < 6$ a special case is that $\lceil n/\sqrt{2}\rceil + 1 \geq n$ for $n < 6$, and thus the recursive calls in the else statement would not actually shrink the problem size for $n < 6$.

First, what is the running time of the above algorithm? We mentioned that the original contraction algorithm can be implemented to run in time $O(n^2)$. Thus the running time $T(n)$ of KS satisfies the recurrence

$$T(n) \leq 2T(\lceil n/\sqrt{2}\rceil + 1) + O(n^2).$$

The master Master theorem then implies $T(n) = O(n^2 \log n)$.

The key benefit though will be that KS has much better success probability. Whereas the vanilla contraction algorithm had success probability at least $1/\binom{n}{2}$, we will now show that KS has success probability at least $\Omega(1/\log n)$.

Lemma 10.2 *KS outputs a minimum cut with probability* $\Omega(1/\log n)$.

Proof: Let p_k be a lower bound on the probability that KS returns a minimum cut on its input at the kth level of recursion, where $k = 0$ corresponds to the base case $n < 6$. Then clearly $p_0 = 1$ is a valid lower bound since we try all cuts. What about p_{k+1}? By our choice of t, the probability that G_1 still contains a minimum cut of G is at least $1/2$ (and similarly for G_2). Thus the probability that C_1 is a minimum cut of G is at least $(1/2)\cdot p_k$. The same is true for C_2. Since G_1 and G_2 are independent, the probability that *neither* C_1 nor C_2 is a minimum cut of G is thus at most $(1-(1/2)p_k)^2$. Therefore, a valid lower bound on the probability that KS returns a minimum cut at the $(k+1)$st level of recursion is

$$p_{k+1} = 1 - (1-(1/2)p_k)^2 = p_k - (1/4)p_k^2. \tag{10.2}$$

Define new variables z_k by $p_k = 4/(z_k+1)$. Then $z_0 = 3$ and a substitution into (10.2) shows that $z_{k+1} = 1 + z_k + 1/z_k$. Induction on k then implies that $z_k \geq k$ for all $k \geq 0$. Then, given this, induction on k then implies $z_k \leq 3 + 2k$ for all $k \geq 0$. Notice the highest level of recursion of interest is some $k^* = O(\log n)$, since in each recursive call n decreases by a factor of roughly $\sqrt{2}$ (which can only happen $\log_{\sqrt{2}} n$ times). Thus $z_{k^*} = O(\log n)$; remembering the definition of z_k, this implies p_{k^*}, the success probability at the root of the recursion tree, is $\Omega(1/\log n)$. ■

Given the above lemma, as with the analysis for the vanilla contraction algorithm, we see that it suffices to run KS $\Theta(\log n \log(1/P))$ times to have success probability $1-P$. Since the KS running time itself is $O(n^2 \log n)$, taht implies that the total running time to obtain success probability $1-P$ is $O(n^2(\log n)^2 \log(1/P))$.

After Karger-Stein It was later shown by Karger how to obtain a randomized Monte Carlo algorithm with running time $O(m \log^c n)$ for some constant $c > 0$, which is nearly optimal [Karger00]. That algorithm is not based on contraction. A deterministic algorithm with running time $O(m \log^c n)$ for simple, unweighted graphs was very recently (in 2015!) given by Ken-ichi Kawarabayashi and Mikkel Thorup [KawarabayashiT15]. Note the contraction-based algorithms above can easily handle positive integer weights since one can then sample an edge for contraction with probability proportional to its weight W, which is correct by the previously given analysis since it has the same effect as interpreting that edge as W parallel unit-weight edges. The algorithm [Karger00] can also handle weighted graphs. Unfortunately [KawarabayashiT15] cannot handle weights, and it is still an open problem to give a deterministic nearly-linear time algorithm for global min-cut which can handle weighted graphs.

References

[1] David R. Karger. Minimum cuts in near-linear time. *Journal of the ACM*, 47(1), pages 46–76, 2000.

[2] David R. Karger. Global Min-cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm. *Proceedings of the 4th Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms (SODA)*, pages 21–30, 1993.

[3] David R. Karger, Clifford Stein. An $\tilde{O}(n^2)$ algorithm for minimum cuts. *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC)* , pages 75–765, 1993.

[4] Ken-ichi Kawarabayashi, Mikkel Thorup. Deterministic Global Minimum Cut of a Simple Graph in Near-Linear Time. *Proceedings of the 47th Annual ACM Symposium on Theory of Computing (STOC)* , pages 665–674, 2015.