4 Min-Cut Algorithms

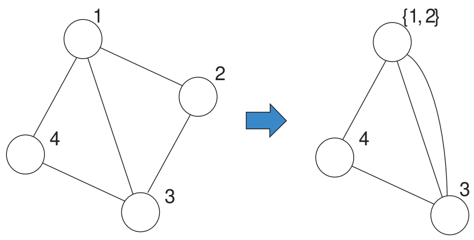
In this lecture, we discuss the min-cut problem for undirected graphs. Consider a graph $G = (V, E)$. A cut in G is a partition of V into two nonempty subsets, denoted $(X, \overline{X})$, where $X \subseteq V$ and $\overline{X} = V - X$. The size of the cut is the number of edges in E that cross the cut, or if G is edge-weighted, the size of the cut is the sum of the weights of the crossing edges. The min-cut problem can be stated as follows: Given $G = (V, E)$ and weights w_e for each $e \in E$, compute a minimum cut in G defined as the cut of least weight in the graph. It's worth noting that the same graph may have several minimum cuts.

Traditionally, min-cut has been computed using max-flow algorithms, since, by the max-flow min-cut theorem in **CLRS**, the value of a maximum flow is equal to the capacity of a minimum cut. However, algorithms for max-flow give the min *s-t cut* of a graph, or the cut of $G = (V, E)$ that partitions V into S and $T = V - S$ such that $s \in S$ and $t \in T$. However, our problem statement requires that we find the *global* min-cut of G. We can still do this using a max-flow algorithm, by fixing a vertex s, and varying t over all other nodes in the graph. We then calculate the s-t min-cut for each t, and take the minimum of all such s-t min-cuts. This takes $(n - 1)$ max-flow computations, and while a naive implementation might take $O(n\times$ max flow time), with cleverness we can do all $n - 1$ computations in the time it would take to do a single max flow computation.

Our intuition tells us that since max flow is a more complicated entity than min-cut, we should be able to compute the min-cut more quickly than O(max flow time). In the following sections, we'll see several faster algorithms that use a randomized approach, as well as a simple deterministic algorithm for computing the min-cut.

4.1 Contract

All of the algorithms we'll consider rely on the contract operation. As defined in **CLRS**, a *contraction* of an undirected graph $G = (V, E)$ by an edge $e = (u, v)$ is a graph $G' = (V', E')$ where $V' = V - \{u, v\} \cup \{x\}$ and x is a new vertex. The set of edges E' is formed from E by deleting the edge (u, v) and, for each vertex w incident to u or v, deleting whichever of (u, v) and (v, w) is in E and adding the new edge (x, w). The following picutre illustrates this:



The running time for this operation is $O(n)$. For a weighted graph, the new weight of a consolidated edge is the sum of the two edges that now coincide: $w_{\{1,2\},3} = w_{1,3} + w_{2,3}$.

4.2 Karger's randomized contraction algorithm

We first introduce an algorithm RANDCONTRACT, which simply picks an edge at random and contracts it.

RANDCONTRACT:
while $|V| > 2$
  do choose an edge e at random,
    Pr[choosing e] $= \frac{w_e}{\sum_{f \in E} w_f}$
    CONTRACT e
return resulting cut $(X, \overline{X})$

The running time of RANDCONTRACT is $O(n^2)$, since both choosing an edge at random and running CONTRACT take $O(n)$ and both these operations are in a loop that executes $O(n)$ times.

Observe that this algorithm does nothing to minimize the cut. What is the probability that it actually outputs a min-cut of the graph? Fix a particular min-cut $(X, \overline{X})$ in G.

$$\Pr[\text{CONTRACT destroys } (X, \overline{X})] = \Pr[\text{We pick an edge crossing } (X, \overline{X})]$$

If the size of a min-cut in G is k, then every vertex has degree $\geq k$. So, $m \geq nk/2$.

$$\begin{aligned} \Pr[\text{CONTRACT destroys } (X, \overline{X})] &\leq \frac{k}{nk/2} \\ &= \frac{2}{n} \end{aligned}$$

If n is large, $2/n$ is small, so the probability of killing the cut using this algorithm is small. However, since n decreases as the algorithm runs, on each run of the algorithm there is a greater likelihood that we will kill the cut. In fact, the last contraction destroys $(X, \overline{X})$ with probability $\leq 2/3$.

$$\begin{aligned} \Pr[(X, \overline{X}) \text{ survives all } (n-2) \text{ contractions}] &\leq 1 - \Pr[\text{CONTRACT destroys } (X, \overline{X})] \\ &= \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right)\cdots\left(1 - \frac{2}{3}\right) \\ &= \frac{2}{n(n-1)} \\ &= \Theta\left(\frac{1}{n^2}\right) \end{aligned}$$

If we run RANDCONTRACT n^2 times, we expect to see a min-cut, which we can easily recognize, since it is the smallest of the output cuts. This leads us to the following simple algorithm discovered by Karger.

KARGER:
Repeat RANDCONTRACT $c\binom{n}{2} \ln n$ times
Output the least weight cut obtained.

Recall RANDCONTRACT takes $O(n^2)$ to run, and we're running it $O(n^2 \log n)$ times, so the running time of KARGER is $O(n^4 \log n) = \tilde{O}(n^4)$. Comparing this to the $O(n^3)$ running time of our max-flow algorithm, KARGER is not an improvement.

4.2.1 Success Probability

$$
\begin{aligned}
\Pr[\text{KARGER fails to find } (X,\overline{X})] &\le \left(1-\frac{2}{n(n-1)}\right)^{\left(c\binom{n}{2}\ln n\right)} \\
&\le e^{\frac{-2}{n(n-1)}\left(c\binom{n}{2}\ln n\right)} && \forall x \in \mathcal{R},\ 1+x \le e^x. \\
&= e^{-c\ln n} \\
&= n^{-c}
\end{aligned}
$$

Thus, failure happens with polynomially small probability which implies it succeeds with high probability. More formally, $\Pr[\text{KARGER succeeds}] \ge 1 - n^{-c}$ ∎

4.3 A recursive contraction-based algorithm

Returning to RANDCONTRACT, recall that its success probability is based on the fact that $\Pr[\text{CONTRACT destroys } (X,\overline{X})] \le 2/n$, so the more vertices there are in the graph, the less likely we are to pick an edge that crosses the min-cut. What if we didn't randomly pick edges to CONTRACT until a single cut remained, but instead we contracted the graph down to t vertices for some t, and then applied an $O(t^3)$ deterministic min-cut algorithm to the resulting graph? Then, the chances we succeed improve to

$$
\begin{aligned}
\Pr[(X,\overline{X}) \text{ survives contraction to } t \text{ vertices}] &\ge \left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)\cdots\left(1-\frac{2}{t+1}\right) \\
&= \frac{t(t-1)}{n(n-1)} \\
&\approx \frac{t^2}{n^2}
\end{aligned}
$$

The best we can do using this idea is $\Theta(n^{8/3})$, which is an improvement over the $O(n^3)$ time it takes using a max-flow method. However, we can improve further by observing that the $O(t^3)$ running time for the deterministic phase contributes the greatest cost to the algorithm's running time. The following algorithm uses the same idea, but makes it quicker with recursion.

```
RECURSIVECUT(G):
    if n = 2
        then return the only cut in the graph
        else H ← G
             CONTRACT G downto ⌈n/√2⌉ vertices
             RECURSIVECUT G
             CONTRACT H downto ⌈n/√2⌉ vertices
             RECURSIVECUT H
             return minimum of the two resulting cuts
```

Drawing out the recursion tree shows us that the running time can be expressed by the recurrence: $T(n) = 2T\left(\frac{n}{\sqrt{2}}\right) + O(n^2)$. We can apply the Master theorem to find the solution $T(n) = O(n^2 \log n)$.

4.3.1 Success Probability

We can also observe that since the number of leaves in the tree corresponds to the number of cuts we've tried out, which is $2^{2\log n} = n^2$, this recursive algorithm is good because it tries out a lot of cuts! However, since many of the nodes have common ancestors, we lose the independence of the events of the different cuts being chosen. The probability that $(X, \overline{X})$ survives one branch of the recursion tree is the probability that it survives the initial contraction to $n/(\sqrt{2})$ vertices times the probability it survives RecursiveCut starting at $n/(\sqrt{2})$ vertices given it has survived so far. If we let α be the probability $(X, \overline{X})$ survives the right branch, then

$$\Pr[(X, \overline{X}) \text{ survives at least one branch}] = 1 - (1-\alpha)^2$$

Let $p(n) = \Pr[(X, \overline{X}) \text{ is returned by RecursiveCut}]$. Then,

$$p(n) = 1 - \left(1 - \frac{1}{2}p\left(\frac{n}{\sqrt{2}}\right)\right)^2 - O\left(\frac{1}{n}\right)$$

We can prove that the final $O(1/n)$ term is insignificant. Instead of solving the recurrence, we will guess a solution and prove it by induction. We begin by substituting variables as follows $q_k = p(\sqrt{2^k}); n = \sqrt{2^k}; k = 2\log n$. This gives

$$q_k = q_{k-1} - (1/4)(q_{k-1}^2)$$

Substituting again with $z_k = 4/(q_k); q_k = 4/(z_k+1)$, we get

$$\begin{aligned} \frac{4}{z_k+1} &= 4\left(\frac{1}{z_{k-1}+1} - \frac{1}{(z_{k-1}+1)^2}\right) \\ &= \frac{4z_{k-1}}{(z_{k-1}+1)^2} \end{aligned}$$

Moving variables around will give us

$$z_k = z_{k-1} + 1 + \frac{1}{z_{k-1}}$$

Claim 4.1 $k < z_k < k + H_{k-1} + 3$

The proof of this is by induction, and is not particularly interesting, so we won't show it here. After reverse substitution,

$$p(n) = q_{2\log n} = \frac{4}{z_{2\log n} + 1} = \Theta\left(\frac{1}{\log n}\right)$$

Note that this probability of success is still not very good, but it's exponentially better than $1/n^2$. Repeating RecursiveCut $O(\log^2 n)$ times gives us a good probability of success: $1 - (1/(n^c))$. The overall running time of this algorithm is

$$\begin{aligned} &= O(n^2 \log n) \times O(\log^2 n) \text{ repetitions} \\ &= O(n^2 \log^3 n) \end{aligned}$$

This remains the best known algorithm so far.

4.4 Stoer-Wagner's deterministic min-cut algorithm

Now let us introduce a deterministic algorithm for calculating the min cut of a graph.

Stoer-Wagner's first theorem is as follows. Let s and t be 2 vertices of a graph $G = (V, E)$. Let $G/\{s, t\}$ be the graph obtained by contracting s and t. Then, a min-cut of G can be obtained by taking the smaller of a min s-t-cut and a min-cut of $G/\{s, t\}$. The theorem holds by a simple proof stating either there exists a min cut of G that separates s and t, or such a cut does not exist, so a global min-cut of G must be in $G/\{s, t\}$. The implication this theorem has for Stoer-Wagner's algorithm is that, on the first iteration, we find a min s-t cut of the graph and either it is the global min-cut, or the global min-cut exists in our contracted graph.

So, we need a function to give us the min s-t-cut of a graph G. MinCutPhase gives us this, by defining a set A that begins with a single starting node, and adding to A the most tightly connected vertex until no vertices remain in the graph, where the *most tightly connected vertex* x is

$$x \notin A \text{ such that } w(A, x) = \max\{w(A, y) | y \notin A\} \tag{1}$$

It's important to understand that tightly connected refers to the summation of the weights of the edges from A to x, not just the weight of a single edge.

```
MinCutPhase(G, w, a):
    A ← {a}
    while A ≠ V
            x ← MTC
            if |A| == |V| − 2
                    s ← x
            if |A| == |V| − 1
                    t ← x
                    CP ← (A, Ā)
            add x to A
    Contract(s, t)
    return CP
```

It's interesting to observe the analogy to Prim's algorithm for finding the minimum spanning tree, which essentially takes the most loosely connected vertex into a starting set, in the same way that MinCutPhase takes the most tightly connected vertex. However, Prim is concerned with taking the vertex connected to the light edge crossing the cut rather than the sum of the weights of edges from the starting set to the vertex.

Note that the cut-of-the-phase CP that is returned by MinCutPhase is the minimum s-t-cut of G, which is to say the min-cut that separates vertices s and t, where s and t are the last two vertices added to A. It then contracts the vertices s and t into a super-vertex. We then use the theorem we've stated to devise an algorithm to find a global min-cut for G.

```
MinCut(G, w, a):
    A ← {a}
    while |V| > 1
            CP ← MinCutPhase(G, w, a)
            if w(CP) < w(MC)
                    MC ← CP
```

Basically, we just store the lightest cut-of-the-phase for each run of MinCutPhase. By our theorem, this algorithm returns the min-cut.

4.4.1 Running Time

We analyze the running time of the algorithm by considering an implementation using a priority queue. Recall that a priority queue has operations ExtractMax which runs in $\log n$ time, and IncreaseKey which runs in constant time. In our algorithm, we use ExtractMax to get the most tightly connected vertex, and we do this for each vertex in the graph, in $O(n \log n)$ time. For each vertex selected, we update the weights of all the edges from the vertex to any vertices remaining in $\overline{X}$. Since we do this for each edge in the graph, it takes $O(m)$ time. Thus, the overall running time of MinCut is $O(n) \times O(m + n \log n)$, or $O(mn + n^2 \log n)$.

4.4.2 Proof of correctness

We must prove that MinCutPhase is in fact a min s-t-cut of the graph, where s and t are the two vertices last added in the phase. The proof of the following lemma requires that we show that any abitrary s-t-cut C of G is at least as heavy as the cut of the phase.

Lemma 4.2 MinCutPhase *returns a min s-t-cut of G.*

We prove this by induction on the set of active vertices. Let $C = (X, \overline{X})$ be an arbitrary s-t-cut, and CP be the cut of the phase. We must show $w(C) \geq w(CP)$.

Observe that a run of MinCutPhase gives us a permutation of all the vertices in the graph (where s and t are the two vertices added last in the phase). We say that a vertex v is *active* if $v \in X$, the vertex before v in the ordering of vertices produced by MinCutPhase is in $\overline{X}$ or vice versa, which is to say, they're on opposite sides of the cut. We define A_v to be the set of vertices added to A before v and C_v to be the cut of the set $A_v \cup \{v\}$ induced by C. Let v_0 be the first active vertex.

Base Case: $w(A_{v_0}, v_0) \leq w(C_{v_0})$

Let v_0 be the first active vertex. By the definition of these two quantities, they are equivalent. A_{v_0} is simply all vertices added to A before v_0, and the edges between these vertices and v_0 are the only edges that cross the cut C.

Let u, v be active vertices, such that v is added to A before u.

$$\begin{aligned}
w(A_u, u) &= w(A_v, u) + w(A_u - A_v, u) \\
&\leq w(C_v) + w(A_u - A_v, u) && \text{by induction, } w(A_v, u) \leq w(A_v, v) \leq w(C_v) \\
&\leq w(C_u) && w(A_u - A_v, u) \text{ contributes to } w(C_u) \text{ but not to } w(C_v)
\end{aligned}$$

Thus, since t is always an active vertex since the last cut of the phase separates s from t by definition, for any active vertex t

$$w(A_t, t) \leq w(C_t) = w(C)$$

So, the cut of the phase is at most as heavy as C. ∎