1 Minimum Cut Problem

Today, we introduce the minimum cut problem. This problem has many motivations, one of which comes from image segmentation. Imagine that we have an image made up of pixels – we want to segregate the image into two dissimilar portions. If we think of the pixels as nodes in the graph and add in edges between similar pixels, the min cut will correspond to a partition of the pixels where the two parts are most dissimilar.

Let us start with the definition of a cut. A *cut* S of a graph $G = (V, E)$ is a proper subset of V ($S \subset V$, $S \neq \emptyset$, $S \neq V$). The size of a cut w.r.t. S is the number of edges between S and the rest of the graph ($V \setminus S$). In the example below, where S is the set of black nodes and $V \setminus S$ is the set of white nodes, the size of the cut is 2.

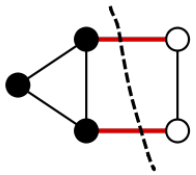


Figure 1: Cut size = 2 (Image Source: Wikipedia)

The minimum cut problem (abbreviated as "min cut"), is defined as followed:
Input: Undirected graph $G = (V, E)$
Output: A minimum cut S– that is a partition of the nodes in G into S and $V \setminus S$ that minimizes the number of edges running across the partition.
Intuitively, we want to "destroy" the smallest number of edges possible.

To find the min-cut, a trivial solution is to enumerate over all $O(2^n)$ subsets. However, this is way too time consuming. Thus we present a more efficient algorithm, as follows.

2 Karger's Algorithm

A note on randomized algorithms:
Karger's Algorithm is a randomized algorithm. It is different from the randomized algorithms that we have seen before. The randomized algorithms we've seen so far (such as Quicksort and the HW question to find colinear points) have good runtimes in expectation, but may occasionally run significantly longer than this (and may not even be guaranteed to terminate!). Nevertheless, these algorithms provide a guarantee that the solution produced upon termination is *always correct*. Algorithms with the properties above are known as "Las Vegas" algorithms.

This is not the case for Karger's Algorithm. Karger's Algorithm is a randomized algorithm whose runtime is deterministic; that is, on every run, the time to execute will be bounded by a fixed time function in the size of the input (i.e. a worst-case runtime bound), but the algorithm may return a wrong answer with a small probability. Such an algorithm is called a "Monte Carlo" algorithm.

2.1 Finding a Min-Cut

Like some of the other graph algorithms we've seen before, Karger's Algorithm will use the notions of "supernodes" and "superedges". A supernode is a group of nodes. A superedge connecting two supernodes X and Y consists of all edges between a pair of nodes, one from X and one from Y. Initially, all nodes will start as their own supernode and every superedge just contains a single edge. The intuition behind Karger's Algorithm is to pick any edge at random (among all edges contained in superedges), merge its endpoints, and repeat the process until there are only two supernodes left. These supernodes define the cut.

Algorithm 1: IntuitiveKarger(G)

while *there are more than 2 supernodes:* **do**
    Pick an edge $(u, v) \in E(G)$ uniformly at random;
    Merge u and v;
Output edges between remaining two supernodes

The goal of this lecture will be to show that this simple algorithm can be made to work with good probability. In what follows, we will use the following notation: We will refer to the nodes within a supernode u as $V(u)$, and the set of edges running between two supernodes u, v as E_{uv} (this is the superedge between u and v).

Here is how we initialize the algorithm:

Algorithm 2: Initialize(G)

$\Gamma \leftarrow \emptyset$; // the set of supernodes
$F \leftarrow \emptyset$; // the set of sets of edges
foreach $v \in V$ **do**
    $\bar{v} \leftarrow$ new supernode;
    $V(\bar{v}) \leftarrow \{v\}$;
    $\Gamma \leftarrow \Gamma \cup \{\bar{v}\}$;
foreach $(u, v) \in E$ **do**
    $E_{uv} \leftarrow \{(u, v)\}$;
    $F \leftarrow F \cup \{(u, v)\}$;

and here is how we merge two supernodes:

Algorithm 3: Merge(a, b, Γ) //Γ is the set of supernodes with $a, b \in P$

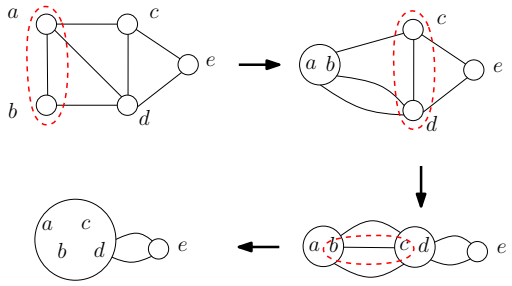
$x \leftarrow$ new supernode ;
$V(x) \leftarrow V(a) \cup V(b)$; //merge the vertices of a and b
foreach $d \in \Gamma \backslash \{a, b\}$ // $O(n)$ *iterations* **do**
    $E_{xd} \leftarrow E_{ad} \cup E_{bd}$; //$O(1)$ operation using linked lists
$\Gamma \leftarrow (\Gamma \backslash \{a, b\}) \cup \{x\}$;

Now, we can present the Karger's algorithm in full detail.

Algorithm 4: Karger(G)

Initialize(G); //Γ is the set of supernodes, F is the set of superedges
while $|\Gamma| > 2$ **do**
    $(u, v) \leftarrow$ uniform random edge from F;
    Merge($\bar{u}, \bar{v}, \Gamma$); //$u \in \bar{u}, v \in \bar{v}$
    $F \leftarrow F \backslash E_{\bar{u}\bar{v}}$;
Return one of the supernodes in Γ and $|E_{xy}|$; //$\Gamma = \{x, y\}$

The following example illustrates one possible execution of Karger's Algorithm.



The example above happens to give the correct minimum cut, but only because we carefully picked the edges to contract. There are many other choices of edges to contract, so it's possible we could have ended with a cut with > 2 edges.

The runtime of the algorithm is $O(n^2)$ since each merge operation takes $O(n)$ time (going through at most $O(n)$ edges and vertices), and there are $n-2$ merges until there are 2 supernodes left.

3 Analysis

We prove that we obtain the correct answer with sufficiently high probability under uniformly random selection of edges.

Claim 1. *The probability that Karger's algorithm returns a min-cut is at least* $\frac{1}{\binom{n}{2}}$.

Proof. Fix a particular min-cut S^*. If Karger's algorithm picks any edge across this cut to do a merge on, then S^* will not be output. However, if all edges that the algorithm selects are not across the cut, then S^* will be output.

$$\begin{aligned}
&P(\text{Karger outputs } S^*)\\
=&P(1^{st} \text{ edge not across } S^*) \cdot P(2^{nd} \text{ edge not across } S^* \mid 1^{st} \text{ edge not across } S^*)\\
&\cdot \cdots \cdot P((n-2)^{th} \text{edge not across } S^* \mid 1^{st}, 2^{nd}, \ldots, (n-3)^{th} \text{ edges all not across } S^*)
\end{aligned}$$

We say that an edge is *good* (w.r.t. S^*) if it is not across the cut $S* - (V \backslash S*)$.
Note that

$$P(1^{st} \text{ edge is good}) = 1 - \frac{K}{m} \geq 1 - \frac{K}{\frac{n \cdot K}{2}} \leq \frac{n-2}{n}$$

where K is the number of edges across the min-cut S^*, and m is the total number of edges in the graph. Note that if the min-cut size is K, then $m \geq \frac{n \cdot K}{2}$. This is because any node v in the graph is a cut of size $deg(v)$, and so for all v, $deg(v) \geq K$. Thus, $m = \sum_v deg(v)/2 \geq nK/2$.

We now show that in the multigraph, after the first $j-1$ edges are merged, if none of these edges are across S^*, then the min-cut size is still K: Call the multigraph after the first $j-1$ edges are merged, G_j. Every cut in G_j is a valid cut in G, so the min cut of G_j has value at least that of S^*. If the first $j-1$ edges are not across S^*, then for any supernode x in G_j, the edges in $V(x)$ must be on the same side of the min cut S^* (either all in S^* or all in $V \backslash S^*$). Because of this, S^* is a valid cut in G_j as well, and the size of the min cut of G_j is the same as the min cut size of G.

Consider $j \geq 1$, then define P_j as the probability that the jth edge is good, given that the first $j-1$

edges are also good:

$$\begin{aligned} P_j &= P(j^{th} \text{ edge good} \mid 1^{st}, 2^{nd}, \ldots, (j-1)^{th} \text{ edges all good}) \\ &= 1 - \frac{K}{\text{number of edges in multigraph}} \\ &\geq 1 - \frac{K}{\frac{K}{2} \cdot \text{number of supernodes}} \end{aligned}$$

The first equality is a definition. The second equality comes from the fact that the first $j-1$ edges are good, so that the min cut is still a valid cut in G_j, so that the probability that the jth edge is good is the probability that none of the K min cut edges are picked. The third equality holds because (just as in our earlier argument) every supernode represents a cut, so that every supernode must have degree $\geq K$, and hence the number of edges in the multigraph is at least the number of supernodes $\cdot K/2$.

Every call to the Merge procedure decreases the number of supernodes by 1, thus the number of super nodes after $j-1$ merges is $n-j+1$.
Then, for all $j \geq 1$,

$$\begin{aligned} P_j &\geq 1 - \frac{K}{\frac{K}{2} \cdot (n-j+1)} \\ &= \frac{n-j+1-2}{n-j+1} \\ &= \frac{n-j-1}{n-j+1} \end{aligned}$$

Thus, we've proved that $P_j \geq \frac{n-j-1}{n-j+1}$. But what we really care about is $P_1 \cdot P_2 \cdot \cdots \cdot P_{n-2}$.
Using the expression of P_j calculated, we have

$$P(\text{Karger outputs } S^*) = P_1 \cdot P_2 \cdot \cdots \cdot P_{n-2} \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \cdots \cdot \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3} = \frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}}.$$

We obtained the quantity above since all numerators except for 2 cancel and all denominators except for $n(n-1)$ cancel. □

A success probability of $1/\Theta(n^2)$ might seem very small. However, we'll see that we can boost this probability to an arbitrarily large probability, by performing repeated independent trials of Karger's algorithm.

Let $C > 0$ be an arbitrarily large constant.

Algorithm 5: AmplifiedKarger(G) //C is a constant

Run $C \cdot \binom{n}{2} \cdot \ln n$ independent Karger procedures, keep track of min cut so far, output the best one.

The runtime of AmplifiedKarger is clearly $O(n^4 \log n)$ since we run $O(n^2 \log n)$ trials of an $O(n^2)$ time algorithm.

Claim 2.

$$P(\text{AmplifiedKarger } \textit{is correct}) \geq 1 - \frac{1}{n^C}$$

where C is the constant used in the amplification algorithm.

Remark 1 (Useful fact)**.** *For $x > 0$, $\left(1 - \frac{1}{x}\right)^x \leq \frac{1}{e}$.*

Proof.

$$
\begin{aligned}
P(\text{AmplifiedKarger is incorrect}) &= P(\text{Karger is incorrect for all the } C \cdot \binom{n}{2} \cdot \ln n \text{ independent runs}) \\
&= (P(\text{Karger is incorrect}))^{C \cdot \binom{n}{2} \cdot \ln n} \\
&\leq \left(1 - \frac{1}{\binom{n}{2}}\right)^{C \cdot \binom{n}{2} \cdot \ln n} \\
&= \left(\left(1 - \frac{1}{\binom{n}{2}}\right)^{\binom{n}{2}}\right)^{C \ln n} \\
&\leq \left(\frac{1}{e}\right)^{C \ln n} \\
&= \frac{1}{n^C}.
\end{aligned}
$$

□

Remark 2 (General Way Of Boosting The Success Rate of Monte Carlo Algorithms)**.** *If an algorithm is correct with probability (w.p.) P, we can run it $c \cdot (1/P) \ln n$ times and output the best result found, so that the amplified algorithm is correct w.p. $1 - \frac{1}{n^c}$.*

4 Karger-Stein Algorithm

While we may be happy with our polynomial time algorithm for finding the min cut as compared to the trivial exponential algorithm, $\Omega(n^4)$ is slower than we'd like. In particular, this algortihm would not be practical on large networks encountered in many of today's applications. Can we do better?

In fact, there is a better way to run Karger's Algorithm than running n^2 independent trials. This algorithm is known as the Karger-Stein algorithm, and detailed in `this paper`.

Here is the pseudocode of the Karger-Stein Algorithm:

Algorithm 6: KargerStein(G)

```
n ← |G| ;
if n < 2√2 then
    Karger(G) ;
else
    Run Karger's procedure twice, each time until n/√2 supernodes remain; get two multigraphs G_1, G_2 ;
    (S_1, e_1) ← KargerStein(G_1) ;
    (S_2, e_2) ← KargerStein(G_2) ;
    Output min{(S_1, e_1), (S_2, e_2)}
```

While we don't include the fairly technical proof, the following bound can be shown for the probability

of success of Karger-Stein.

$$P(\text{Karger-Stein is correct}) \geq \frac{1}{\Theta(\log n)}.$$

So, by our remark above, if we run $O(\log^2 n)$ trials, the new amplified algorithm will be correct w.p.

$$1 - \frac{1}{\text{poly}(n)}$$

Now we are experts in divide-and-conquer. We thus perform the runtime analysis:

$$T(n) = O(n^2) + 2T\left(\frac{n}{\sqrt{2}}\right),$$

by the Master's Theorem,

$$T(n) = \Theta(n^2 \log n).$$

Thus the final runtime of the AmplifiedKargerStein algorithm is

$$\Theta(n^2 \log n) \cdot O(\log^2 n) = O(n^2 \log^3 n).$$