CSE 203A- Lecture 3: Randomized Algorithm for finding the Min-Cut in a Graph

Prof. Ramamohan Paturi

January 14, 2014

Scribe Notes by: Hani Altwaijry

Definitions

For an undirected graph $G = (V, E)$, a *Cut* is a partition of V into two sets $A, B \subseteq V$. Such that $A \cup B = V$ and $A \cap B = \Phi$, where $A \neq \Phi$ and $B \neq \Phi$. An example is shown in Figure 1

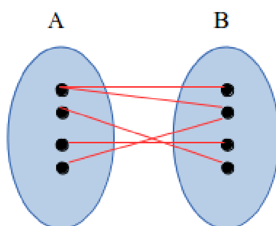

Figure 1: An illustration of a graph cut.

The edges crossing the cut are known as the *Cut-Edges*, and are defined as:

$$\text{Cut-Edges}(A, B) = \{e = (x, y) | x \in A, y \in B\}$$

The definition of the *Min-Cut* follows as:

$$\text{Min-Cut}(V) = \underset{\text{Cut}(A,B)}{\arg\min} |\text{Cut-Edges}(A, B)|$$

Relationship to vertex degrees

The degree of a vertex refers to the number of edges incident to that vertex. It is clear that the Min-Cut can have a smaller cardinality than maximum vertex degree in the graph, however, it cannot have a smaller cardinality than the minimum vertex degree, because selecting the vertex alone, results in a cut of a smaller cardinality.

Example:

For a connected graph, give an example of a min-cut, where the size of the cut is less than the maximum degree 3. An example is shown in Figure 2.

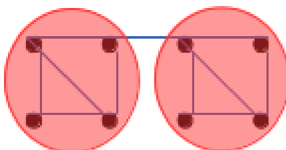

Figure 2: An example of a graph-cut less than the maximum degree 3.

Max-Flow Min-Cut

The Max-Flow Min-Cut problem is defined over a directed graph $G = (V, E)$, with two special nodes designated as s and it is a "source" without any incoming edges, and t for "sink" without any outgoing edges. An example of a directed graph with s, t nodes is shown in Figure 3

This problem was first solved by the Ford and Fulkerson Algorithm described in detail in [1]. The problem is named as "Max-Flow Min-Cut" referring to the relationship of the maximum flow in the network and the minimum cut, where finding the maximum flow corresponds to finding a minimum cut in the graph. Further details can be found in [1].

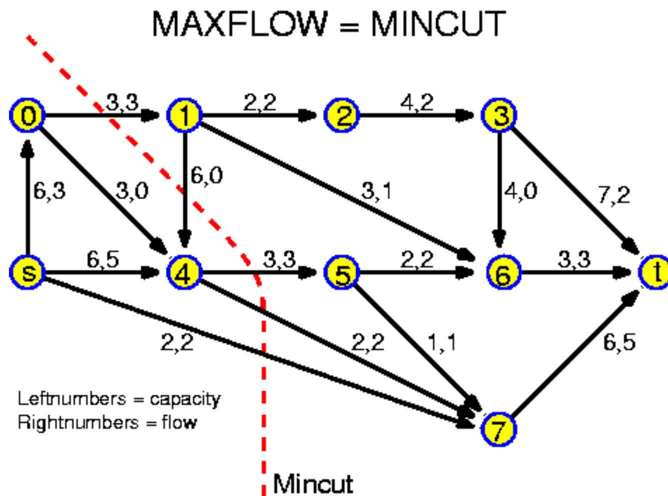

Figure 3: A directed graph with s, t nodes for the max-flow min-cut problem. Example obtained from cs.umu.es

In the case of an undirected graph without a source and a sink, we need to convert our graph to fit the Ford-Fulkerson algorithm by: (1) Substituting two directed edges for each undirected edge, (2) Assigning a capacity of 1 for all edges, and (3) Specifying the source and sink nodes, i.e. s, t. In the simplest approach, the algorithm will try all possible combinations which contributes a multiplicative $O(V^2)$ to the complexity of the Ford-Fulkerson algorithm leading to an overall complexity of $O(V^3\,E^2)$. An example of a converted undirected graph with source and sink nodes is shown in Figure 4.

Further details of finding the max-flow min-cut in an undirected graph can be found in Karger and Levine [2].

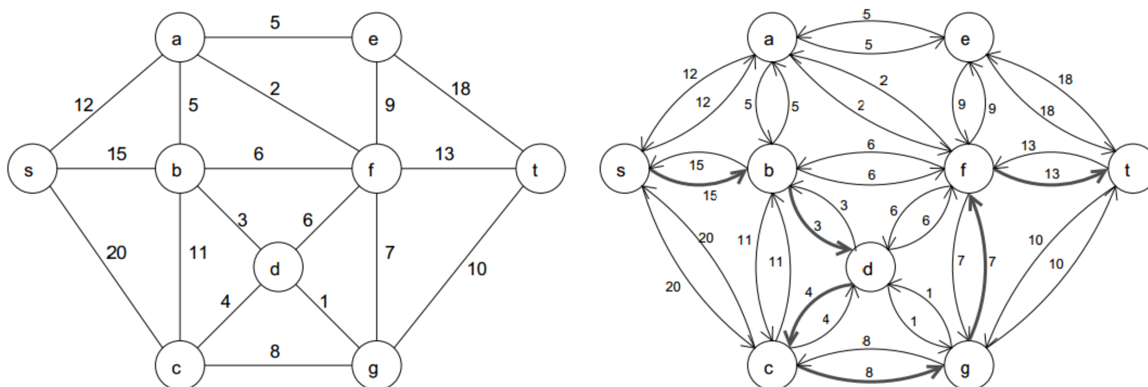

Figure 4: An undirected graph with s, t nodes and its converted counterpart. Example obtained from inf.ufpr.br

Karger's Algorithm

Karger's Algorithm is randomized algorithm that starts with an undirected graph $G = (V, E)$ and proceeds to finding a cut as described in Algorithm 1. In the while-loop body, the main operation, *Edge Contraction* is performed. In this step, the algorithm removes a randomly selected edge, and fuses the vertices which were connected by the removed edge, as well as removing any self-loops. An illustration of that step is in Figure 5.

Algorithm 1 Randomized Min-Cut Algorithm (K)

Input: $G = (V, E)$
Output: Cut=(A, B)
  $G' = (V', E') \leftarrow G$
  while $|V'| > 2$ **do**
    $e = (x, y) \leftarrow Random(E')$
    $E' = E' - e$
    $V' = V' - \{x, y\} + \{xy\}$
    $DeleteSelfLoops(E')$
  end while
  $A = V'(1)$, $B = V'(2)$ // indexing through the vertices

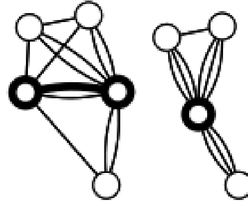

Figure 5: An illustration of an edge contraction. Here, the bold edge was contracted.

The algorithm terminates with two remaining vertices that constitute the cut. The number of edges going between those two vertices is the number of Cut-Edges. Note that in each step, the union of the vertices constitutes V. Also note, parallel edges are kept, therefore we do not decrease any vertex's degree.

Observation: A cut in the graph at any step during the algorithm execution is a cut in the original graph, however, a cut in the original graph is not necessarily a cut in an intermediate graph during execution.

To verify this observation, let's consider the first part of the observation, "a cut in the graph at any step during the algorithm execution is a cut in the original graph". Consider a cut in an intermediate graph during a given iteration. All cut edges in this cut can be found in the original graph, and selecting those edges will form a cut. Note there cannot be anymore edges crossing the cut that were not in the intermediate graph but in the original graph, because the algorithm retains all parallel edges. Now, let's consider the second part, "a cut in the original graph is not necessarily a cut in an intermediate graph during execution". We can see that by considering the cut edges of a cut in the original graph, and finding those edges in the intermediate graph. It is clear that some edges might have been contracted by the algorithm during its execution, and hence the original graph's cut is not an intermediary graph cut.

This algorithm does not always find the min-cut. Two examples of the algorithms' execution is shown in Figure 6 with an example showing the algorithm finding a min-cut, and in the second example, showing the algorithm failing.

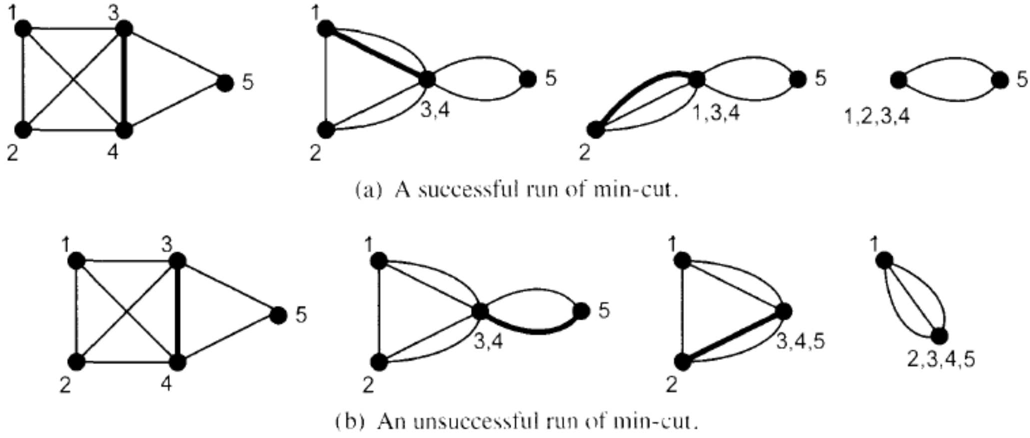

Figure 6: Two examples showing the algorithm execution and two different cases of finding and not finding the min-cut.

The probability this algorithm produces a min-cut is given by:

$$Pr(K \text{ outputs a min-cut}) \geq \frac{2}{n(n-1)}$$

Where $n = |V|$. The analysis of the algorithm and the derivation of this probability is given in the next section. We can improve our result by repetition, where each run is conducted independently. If we run the algorithm T times, we output the answer carrying the minimum number of edges as the min-cut.
The probability that the algorithm does not output a min-cut is:

$$Pr(K \text{ does not output a min-cut}) \leq 1 - \frac{2}{n(n-1)}$$

Running the algorithm T times leads to:

$$Pr(\bigwedge_i K_i \text{ does not output a min-cut}) \leq \left(1 - \frac{2}{n(n-1)}\right)^T, \quad 1 \leq i \leq T$$

Note the effect of conditional independence in the result above. Therefore, running K for T repetitions, yields a probability of success amounting to:

$$Pr(\bigwedge_i K_i \text{outputs a min-cut}) \geq 1 - Pr(\bigwedge_i K_i \text{ does not output a min-cut}) = 1 - \left(1 - \frac{2}{n(n-1)}\right)^T$$

This leads to the question of, *how large a T is required for a probability close to 1?*
We will use the identity:

$$1 - x \leq e^{-x} \quad \forall x \in \mathbb{R}$$

We can verify the validity of this identity using calculus as follows:

$$\begin{aligned} f(x) = &\quad e^{-x} + x - 1 \geq 0 \\ \rightarrow &\quad \frac{\delta}{\delta x}(e^{-x} + x - 1) = -e^{-x} + 1 \\ \rightarrow &\quad -e^{-x} + 1 = 0 \\ \rightarrow &\quad e^{-x} = 1 \\ \rightarrow &\quad -x = \ln(1) \\ \rightarrow &\quad x = 0 \end{aligned}$$

We have a critical point at $x = 0$, and we need to verify that it is a minimum for the inequality to hold. Using the second-derivative test, we find

$$\begin{aligned} & \tfrac{\delta}{\delta x}(-e^{-x}+1) = e^{-x} \\ \rightarrow \quad & f''(x=0) = e^0 = 1 \end{aligned}$$

Now since $f''(x = 0) > 0$, and $x = 0$ is the only critical point in the function, we conclude that $x = 0$ is the global minimum for the function, and therefore, $f(x) \geq 0, \quad \forall x \in \mathbb{R}$.
Substituting the identity in the joint probability of finding a min-cut, we have:

$$1 - \frac{2}{n(n-1)} \leq e^{-\frac{2}{n(n-1)}}$$

Our probability for success becomes:

$$Pr(K^T \text{outputs a min-cut}) \geq 1 - e^{-\frac{2t}{n(n-1)}}$$

By setting $t = C\frac{n(n-1)}{2} \ln n$ we get:

$$1 - e^{-\frac{2t}{n(n-1)}} = 1 - e^{-\frac{2Cn(n-1)\ln n}{2n(n-1)}} = 1 - e^{-C\ln n} = 1 - \frac{1}{n^C}$$

Which means, the probability of success is bounded by $\frac{1}{n^C}$.

$$Pr(K^T \text{outputs a min-cut}) \geq 1 - \frac{1}{n^C}$$

Analysis

When we select an edge, what is the probability that it is a cut-edge?
Let's assume that the size of the min-cut is k. Let $m = |E|$. In the first iteration, our probability of picking a cut-edge is then:

$$Pr(\text{A cut edge is selected}) \leq \frac{k}{m}$$

As noted earlier, the minimum degree of each vertex is at least k, because if a vertex had a degree less than k, it would form a smaller min-cut. Therefore, from the degree sum relationship, we have

$$\sum_{v \in V} deg(v) = 2|E| = 2m$$

We get:

$$m \geq \frac{kn}{2}$$

which relates to the probability of selecting a cut-edge:

$$Pr(\text{A cut edge is selected}) \leq \frac{k}{m} \leq \frac{2}{n}$$

During the execution of the algorithm, the degree of any vertex is always more than k. Henceforth, assuming we did not select a cut-edge in the first iteration.

$$Pr(\text{A cut edge is selected in the second iteration}) \leq \frac{2}{n-1}$$

In our analysis, we assume that in each iteration, we did not select a cut-edge, and therefore, our statements are conditionally *dependent*. Let's define the following:

$$E_i = \text{A cut-edge is not selected during step i}$$

$$F_i = \bigwedge_{j \leq i} E_j$$

The base case is:

$$Pr(F_1) = Pr(E_1) \geq 1 - \frac{2}{n}$$

The next iteration is then:

$$Pr(F_2) = Pr(E_2 \bigwedge F_1) = Pr(E_2|F_1)Pr(F_1)$$

$$Pr(E_i|F_i) \geq 1 - \frac{2}{n-i+1}$$

Our goal is to calculate $Pr(F_{n-2})$

$$\begin{aligned}
Pr(F_{n-2}) &= Pr(E_{n-2} \bigwedge F_{n-3}) = Pr(E_{n-2}|F_{n-3}) \cdot Pr(F_{n-3}) \\
&= Pr(E_{n-2}|F_{n-3})\dot{P}r(E_{n-3}|F_{n-4})...Pr(E_2|F_1)Pr(F_1) \\
&\geq \prod_{i=1}^{n-2} \left(1 - \frac{2}{n-i+1}\right) \\
&= \prod_{i=1}^{n-2} \left(\frac{n-i-1}{n-i+1}\right) \\
&= \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right)...\left(\frac{3}{5}\right)\left(\frac{2}{4}\right)\left(\frac{1}{3}\right) \\
&= \frac{2}{n(n-1)}
\end{aligned}$$

References

[1] L. R. Ford, Jr. and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956. 2

[2] D. R. Karger and M. S. Levine. Finding maximum flows in undirected graphs seems easier than bipartite matching. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 69–78, New York, NY, USA, 1998. ACM. 2