Randomized min-cut algorithm

Consider the randomized min-cut algorithm discussed in class. We have seen that its probability of success is at least $1/\binom{n}{2}$, where n is the number of its vertices.

- Describe how to implement the algorithm when the graph is represented by adjacency lists, and analyze its running time. In particular, a contraction step can be done in $O(n)$ time.
- A weighted graph has a weight $w(e)$ on each edge e, which is a positive real number. The min-cut in this case is meant to be min-weighted cut, where the sum of the weights in the cut edges is minimum. Describe how to extend the algorithm to weighted graphs, and show that the probability of success is still $\geq 1/\binom{n}{2}$. [hint: define the weighted degree of a node]
- Show that running the algorithm multiple times independently at random, and taking the minimum among the min-cuts thus produced, the probability of success can be made at least $1 - 1/n^c$ or a constant $c > 0$ (hence, with high probability).

SOLUTION

First Point

We shall keep nodes in an array. We expect to run $N - 2$ contractions, spawning a new node for each of them, thus the array shall be of length $2N - 2$, the last $N - 2$ elements initially NULL.

Nodes will be annotated with their *degree* and a *new* field with the ID of the node they're currently "contained" in (initially themselves, then the nodes they get contracted to).

An edge (i, j) shall be a vector of three elements $< dest, mult, reverse >$:

- *dest*: the node ID of the destination (i.e. j);
- *mult*: recall we're in a multigraph, so there can be many edges $(i, j)_k$. We represent them as a single edge annotated with a *multiplicity* value, i.e. $(i, j).mult = |\{(i, j)_k\}|$.

  This way, each adjacency list contains $O(N)$ elements;
- *reverse*: as we're in an undirected graph, for each edge (i, j) there is the reverse (j, i). In *reverse* we store a pointer to (j, i), as we'll need it for bookkeeping during contraction operations.

Adjacency lists shall be doubly linked and sorted on *dest* fields.

Random choice of edge

Let N be the initial number of nodes, n be the number of remaining unique nodes.
Let M be the initial number of edges, m be the number of remaining edges.
Let $\delta(i)$ be the degree of node i (NB: contracted nodes will have degree 0). Then:

function ChooseEdge
  $r \leftarrow rand(0, 2m - 1)$ ▷ note $\sum_{i=1}^{n} \delta(i) = 2m$
  $i \leftarrow 0$
  while $nodes[i].degree < r$ **do** ▷ note we'll never reach the NULL tail of $nodes[]$
    $r \leftarrow r - nodes[i].degree$
    $i \leftarrow i + 1$
  end while
  $j \leftarrow 0$
  while $nodes[i].adj[j].mult < r$ **do**
    $r \leftarrow r - nodes[i].adj[j].mult$
    $j \leftarrow j + 1$
  end while
  return $(i,\ j)$

end function

You can think of what we're doing as having concatenated the adjacency lists and then crawled through them to edge r. This would require $O(n^2)$ steps done naively, but saving the degree of the node (the length of the adjacency list) allows us to skip to the next node in one step (a sort of skip-pointer).

In fact, what we do is crawl the list of nodes ($O(n)$) and then one adjacency list ($O(n)$ again).

This procedures results in a random uniform choice of the edge. In fact, the chance that a specific (directed!) edge $(i, j)_k$ is chosen ends up being:

$$P_r[(i, j_k)] = \frac{\delta(i)}{2 \times m} \times \frac{mult(i, j)}{\delta(i)} \times \frac{1}{mult(i, j)} = \frac{1}{2 \times m}$$

where

$$\frac{\delta(i)}{2 \times m} = P_r[node\ i\ is\ chosen]$$

$$\frac{mult(i, j)}{\delta(i)} = P_r[\{(i, j)_k\}|\ node\ i]$$

$$\frac{1}{mult(i, j)} = P_r[(i, j)_k|\ \{(i, j)_k\}]$$

$\{(i, j)_k\}$ is the set of all edges going from i to j

$mult(i, j) = |\{(i, j)_k\}|$ is the multiplicity of edge (i,j)

Thus, the probability that an (undirected!) edge $(i, j)_k$ is chosen results

$$\frac{1}{2m} + \frac{1}{2m} = \frac{1}{m}$$

Contraction

General idea for contracting over edge (i, j):

- Spawn a new node that will represent i and j contracted;
- Update the *new* field of nodes "contained" in i or j;
- Crawl through i and j's adjacency lists: nodes pointed by one but not the other get appended to the new node's list; nodes pointed by both are also appended, but their multiplicity is the sum of the old egdes' multiplicity.

  This can be done in linear time w.r.t. the length of the adjacency lists ($O(N)$) because they're sorted, so we proceed as in a *merge* operation of a MergeSort.

  In all cases, we take advantage of $(i, j).reverse$ to update the adjacency list of the pointed node with the new node. As the lists are doubly linked and assuming a pointer to the last element is kept, this can be done in $O(1)$ time.

function CONTRACT(i, j)
    $nID \leftarrow$ position of next NULL element in the tail of $nodes[]$
    $nodes[n] \leftarrow new\ Node()$
    $\forall x$ in $nodes \mid x.new = i \vee x.new = j\ .\ x.new \leftarrow nID$
    $it_i \leftarrow$ iterator over $nodes[i].adj$
    $it_j \leftarrow$ iterator over $nodes[j].adj$
    while $it_i.hasNext()$ and $it_j.hasNext()$ **do**
        if $it_i.dest = j$ **then**
            $it_i.next()$
        else if $it_j.dest = i$ **then**
            $it_j.next()$
        else if $it_i.dest = it_j.dest$ **then**
            /* create a pair of edges, one the reverse of the other */
            $e \leftarrow new\ Edge(it_i.dest, it_i.mult{+}it_j.mult, new\ Edge(nID, it_i.mult{+}it_j.mult, NULL))$

$e.reverse \leftarrow e$
/* remove old edges from the destination's adjacency list */
$it_i.reverse.remove()$
$it_j.reverse.remove()$
/* put reverse in destinations; note nID is highest so it goes to the tail*/
$nodes[e.dest].adj.append(e.reverse)$
/* put e at the tail of $nodes[nID].adj$ */
$nodes[nID].adj.append(e)$
$it_i.next()$
$it_j.next()$
/* increment $nodes[nID]$'s degree by $e.mult$ */
$nodes[nID].degree \leftarrow nodes[nID].degree + e.mult$
else if $it_i.dest < it_j.dest$ **then**
$e \leftarrow new\ Edge(it_i.dest, it_i.mult, new\ Edge(nID, it_i.mult, NULL))$
$e.reverse \leftarrow e$
$it_i.reverse.remove()$
$nodes[e.dest].adj.append(e.reverse)$
$nodes[nID].adj.append(e)$
$it_i.next()$
$nodes[nID].degree \leftarrow nodes[nID].degree + e.mult$
else ▷ $it_j.dest < it_i.dest$
$e \leftarrow new\ Edge(it_j.dest, it_j.mult, new\ Edge(nID, it_j.mult, NULL))$
$e.reverse \leftarrow e$
$it_j.reverse.remove()$
$nodes[e.dest].adj.append(e.reverse)$
$nodes[nID].adj.append(e)$
$it_j.next()$
$nodes[nID].degree \leftarrow nodes[nID].degree + e.mult$
end if
end while
Process leftovers from either i or j's adjacency lists, if there are any, in a similar fashion.
$nodes[i].degree \leftarrow 0$ ▷ degrees of i and j are put to 0 for ChooseEdge's conveniency
$nodes[j].degree \leftarrow 0$ ▷ we could also detach the adjacency lists if we liked, but it's not necessary
end function

Computing the cut

After running $N-2$ contractions, you'll end up with two nodes hl_1 and hl_2 (highlanders) left over in the graph.

The size of the cut will be $hl_1.degree = hl_2.degree$.

The *new* field of the first N elements of *nodes* will be either hl_1 or hl_2.

The two classes will thus define the cut. Should you need the actual edges making up the cut, you'll crawl through the original graph and output edges connecting nodes of the two classes ($O(N^2)$).

Second Point

In the algorithm we change the way we choose an edge to contract: edges with high weight will have a greater probability to be chosen with respect to others with lower weight. We define $weight : E \longrightarrow \mathbb{R}$ as the function that associate the weight to a given edge. We will use the *weight* function also on subsets of E, intending for $weight(X)$ with $X \subseteq E$ the sum of the weight of all the edges in X.

$$weight(X) = \sum_{e \in X} weight(e) \qquad\qquad where\ X \subseteq E$$

As seen in class, the *min cut* is not unique, nevertheless the sum of weights in all *min_ cuts* is the same, than with a notation abuse we will call it $weight(min\_cut)$. For each edge $x \in E$ the probability to be choose for a contraction is given by its weights normalized with the total sum of the weights of all the edges of the graph.

$$Pr[extract\ x] = \frac{weight(x)}{weight(E)}$$

Now the probability of making an error, when extracting an edge at random, is the probability of extracting one of the "bad" edges, i.e. edges such that their contraction cause a variation in $weight(min\_cut)$. We name BAD the set of "bad" edges.

$$Pr[error] = \sum_{e \in BAD} Pr[extract\ e] = \sum_{e \in BAD} \frac{weight(e)}{weight(E)} = \frac{weight(BAD)}{weight(E)} \tag{1}$$

As usual bad edges are those belonging to every *min cut*, so the total weight of bad edges is less or equal to the weight of *min cut*.

$$weight(BAD) \leq weight(min\_cut) \tag{2}$$

Given a node v, we define $star(v)$ as the set of all the edges touching v. For the same reasons explained in class, for each node v the weight of *min cut* must be less or equal to the sum of the weights of the edges touching v. This is because otherwise we would have a cut, $star(n)$, with weight less than *min cut*, which is absurd.

$$\forall\ v \in V\ .\ weight(min\_cut) \leq weight(star(v)) \tag{3}$$

We can reformulate the *handshaking lemma* for weighted graph in the following way, in which $weight(star(v))$ is something like the "weighted degree" of the node v.

$$weight(E) = \frac{\sum_{v \in V} weight(star(v))}{2} \tag{4}$$

Now we can derive:

$$\begin{aligned} weight(E) &= \frac{\sum_{v \in V} weight(star(v))}{2} && \text{(from 4)} \\ &\geq \frac{|V| weight(min\_cut)}{2} && \text{(from 3)} \end{aligned}$$

$$weight(min\_cut) \leq \frac{2\ weight(E)}{|V|} \tag{5}$$

Finally we have all the ingredients for the proof:

Theorem 1. *Selecting the edges for the contraction according to the probability described, each time we choose an edge the probability of choosing a "bad" edge is less or equal than* $2/|V|$.

$$Pr[error] \leq \frac{2}{|V|}$$

Proof.

$$\begin{aligned} Pr[error] &= \frac{weight(BAD)}{weight(E)} && \text{(from 1)} \\ &\leq \frac{weight(min\_cut)}{weight(E)} && \text{(from 2)} \\ &\leq \frac{\frac{2\ weight(E)}{|V|}}{weight(E)} && \text{(from 5)} \\ &= \frac{2\ weight(E)}{|V| weight(E)} = \frac{2}{|V|} \end{aligned}$$

□

The probability of making an error when selecting an edge is the same as for the case of a normal graph seen in class, so the probability of success for the algorithm (i.e. the probability of choosing well all the times) is still $\geq 1/\binom{n}{2}$.

Third Point

We know that $P(n) \geq 1/\binom{n}{2}$ is the probability of choose well for N times. We have seen also that the probability of error is $\leq (1 - 1/\binom{n}{2})$. Then the probability of having an error each time in N repetitions of the algorithm is

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

Given a c we want the probability of success to be $\geq 1 - 1/n^c$. This is equivalent to ask that the probability of error is $\leq 1/n^c$. Making some calculations:

$$\begin{aligned} \frac{1}{e^{\frac{2N}{n(n-1)}}} = e^{-\frac{2N}{n(n-1)}} \leq \frac{1}{n^c} \\ n^c \leq e^{\frac{2N}{n(n-1)}} \\ ln(n^c) \leq ln(e^{\frac{2N}{n(n-1)}}) = \frac{2N}{n(n-1)} \\ N \geq \frac{n(n-1)ln(n^c)}{2} \end{aligned}$$

So, for each given c, to make the probability of success at least $1 - 1/n^c$ is sufficient to take N grater or equal than $\frac{1}{2}n(n-1)ln(n^c)$.