CME 323: Distributed Algorithms and Optimization, Spring 2016
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 5, 4/19/2017. Scribed by Andreas Santucci.

Lecture contents

1. Graph contractions
2. Star contraction
3. Parallel connected components (random mates)
4. Intro to optimization

1 Parallel MST via Boruvka's Algorithm

Since we need to evaluate the edges in order of weight, Kruskal's algorithm is inherently sequential. We need a new approach if we want a parallel MST algorithm. Kruskal's algorithm is instructive in that it uses the Cut Property which we proved last lecture. Our parallel algorithm, which is really Boruvka's algorithm, will also use this property. [2]

1.1 Observation - Smallest Weight Incident Edge on each Node belongs in MST

A single node is a subset of the nodes, i.e. each node can be considered individually to define a cut. And so if you look at a single node and its incident edges, the smallest one must be in the MST by applying the cut property. Realize that if we determined the smallest weight edge incident to *each* node in our graph, we *must* find at least $n/2$ unique edges belonging to the MST. Why $n/2$? When we search for a lowest weight incident edge on each node, realize that it's possible the node to which such an edge connects also selects the same edge for inclusion in the MST. At worst, this could happen for each node. Hence we must find at least $n/2$ edges in our MST.

1.2 Edge Contraction

Before we can recursively apply our algorithm, we must first take each of the (at least) $n/2$ edges we found belonging to the MST and "merge" the two incident nodes into one.

Deleting Duplicate Edges Suppose we add edge (u, v) to our tree. The neighborhood of u and the neighborhood of v may overlap, hence we have two ways to get from super-node u-v to such a neighbor t. When we contract edge (u, v), we would yield a multi-graph, in which two vertices may be connected by more than one edge. We wish to avoid this, and since our algorithm uses the Red-Rule (and is interested in the lowest weight edge in a cut), we can ignore the edge with larger weight. That is, if $\Gamma(u) \cap \Gamma(v) \neq \emptyset$ in our original graph G, then for each node t in the intersection of these two neighborhoods, we compare $w(u, t)$ with $w(v, t)$ and delete the edge with larger weight.

Recursive formulation But notice that when we contract the edges, the resulting "super" node (consisting of multiple nodes) itself defines a cut. Hence we can again apply the Red-Rule and find the smallest weight edge incident to each of the nodes in the *contracted* graph, and again continue to find more edges belonging to the MST. Hence we repeat the algorithm until we realize an MST. Since each time we find at least $n/2$ edges, we only must repeat our algorithm at most $\log_2 n$ times before finding an MST.

1.3 Boruvka's Algorithm

We assume that the graph is stored in an Adjacency-List, i.e. for each node we store its neighbors in a linked list. Since each node can have at most $n-1$ neighbors, each adjacency list can have at most $n-1$ entries. There are n adjacency lists (one for each node).

Algorithm 1: Boruvka's Algorithm

```
  Input  : Graph $G$, represented by adjacency list
1 for each node, in parallel do
2     Compute smallest weight incident edge          // $O(m)$ work, $O(\log n)$ depth
3 end
4 Contract edges
5 Merge Adjacency Lists                              // $O(n)$ work and $O(1)$ depth
6 Recurse
```

1.4 Analysis

1.4.1 Finding Smallest Weight Incident Edges

Examine the initial `for` loop which is executed in parallel. We know that we're looking for smallest weight edges, hence we know that we need to examine each edge at least once, i.e. we require at least $O(m)$ work. Do we require more? No. By the handshake lemma, $\sum_i \deg(v_i) = 2|E|$. Since for each node $v \in V$, we need to find the minimal weight incident edge, we must scan through $2|E|$ neighbors in total. We may use our parallel quickSelect algorithm on each adjacency list which requires $O(\deg(v_i))$ work and $O(\log \deg(v_i))$ depth. Summing across To bound total work, we must sum across all nodes, hence total work is $O(m)$ by handshake lemma. For depth, realize that we are concerned with $\max_i \deg(v_i)$, which is $O(n)$. Hence depth is $O(\log n)$.

1.4.2 Contracting Edges

Although we've shown that we remove $n/2$ isolated vertices in each round, we have not shown anything about how many edges we remove in each iteration. The question remains, how fast are our sub-problems decreasing in size? Notice that the number of edges removed in a contraction is *at least* equal to the number of vertices (since each vertex selects a lowest eight incident edge). Consider a best case possible sequence of events.We cite Blelloch and Maggs, specifically section 16.2.[1]

round	vertices	edges
1	n	m
2	$n/2$	$m - n/2$
3	$n/4)$	$m - n/2 - n/4 = m - 3n/4$
$\vdots$	$\vdots$	$\vdots$
k	$n/2^{k-1}$	$m - (2^{k-1} - 1)n/(2^{k-1})$

Notice that for all $k \in \mathbb{N}$,

$$n\frac{2^{k-1}-1}{2^{k-1}} = n\underbrace{\left(1 - \frac{1}{2^{k-1}}\right)}_{<1}$$

hence the number of edges never quite drops below $m - n$. So as long as there are $m > 2n$ edges to start with, work is $O(m \log n)$.

Crucial Observation: Minimum Weight Edges form a Forest Crucially, in the first stage of our algorithm, each node its own connected component. It's also a tree, since a component of size 1 with 0 edges is connected and acyclic hence it's a tree. Realize that when we expand via minimum weight edges, since these edges are guaranteed to be in the MST, they *cannot* form a cycle, hence the result we get is a *forest*. Hence, our job is now to actually contract an entire tree (such that we may apply this to each tree in the forest).

To contract a tree, have each node (in parallel) flip a coin. If a vertex flips heads, it becomes the *center* of a star. If it flips tails, then the vertex attempts to become a *satellite* of (some) star by finding a neighbor which is a center.[1] If no such neighbor exists, i.e. if all other neighbors flipped tails or the vertex is isolated, then the vertex becomes a center.

When we perform a contraction, we need to select one node as the center, and the rest are satellites. Edges between satellites and centers must be deleted. Edges which cross partitions must be kept.[2]

Claim 1.1 *For a graph G with n non-isolated vertices, let X_n be the random variable indicating the number of satellites in a call to our tree contraction process described above. Then,*

$$\mathbb{E}[X_n] \geq n/4.$$

[1]If multiple a vertex has multiple neighbors which are centers, it may select one arbitrarily. For example, we select the one with lowest node index.

[2]We now turn to Lemma 16.17 of Blelloch and Maggs.

Proof: Consider any non-isolated vertex $v \in V(G)$. By definition, a non-isolated vertex has at least one neighbor, hence the probability that v becomes a satellite is *at least* $\Pr(\text{node } v \text{ flips Tails}) \times \Pr(\text{neighbor flips Heads}) \geq \frac{1}{4}$. Hence by linearity of expectation,

$$\mathbb{E}[\#\text{ Isolated Vertices}] = \sum_{\text{non-isolated vertices}} \mathbb{E}[\text{vertex } i \text{ becomes isolated}] \geq n/4.$$

■

Contracting a tree yields another Tree Since when doing start contraction on a tree, it remains a tree on each step, the number of edges does down with the number of vertices. Further, since a tree on n nodes has $n-1$ edges, the number of edges in a forest is never more than n. Hence the number of edges in our graph decrease geometrically in expectation. Hence total expected work given by

$$\mathbb{E}[W(n,m)] = \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i kn = O(n).$$

The *Depth* of the contraction is given by $\mathbb{E}[D(n,m)] = O(\log n)$ since when we accumulate *neighborhood lists*, adding an edge to it requires $O(\log n)$ depth since the height of the neighbor hood tree is at most $O(\log n)$.

1.4.3 Merge Adjacency Lists

In terms of our data structures, we have n adjacency lists, and if we are contracting edge (u, v), then we must merge their corresponding adjacency lists together. Suppose that our `list` data structure is singly linked with a head to both the head and tail. Since $u < v$ via lexicographical comparison, we choose to make u the "parent" node, i.e. v's neighbors are absorbed into the neighborhood of u. Hence, we may simply take the tail of u's adjacency list and point it to the head of v's. This takes $O(1)$ work.

Since we must contract at least $n/2$ edges each iteration, this requires $O(n)$ work total. Notice that we may contract each edge independently of the rest, hence we have $O(1)$ depth.

1.4.4 Total Work and Depth

Total Work Notice that our algorithm employs *unary tail recursion.* We have a chain of $O(\log n)$ recursive calls to our algorithm. At each recursive call, we require $O(m)$ work to be performed, since our bottleneck is in finding the smallest weight edge incident each component. Hence total work simply $W(n,m) = O(m \log n)$.

Total Depth Note that at each of the $O(\log n)$ recursive calls, we require $O(\log n)$ depth in order to perform a min-scan. Hence total depth given by $D(n,m) = O(\log^2 n)$.

2 Intro to Optimization

2.1 Overview

The purpose of optimization is to minimize some objective function F. Formally stated, we wish to solve the following problem:

$$\min_{\mathbf{w}} F(\mathbf{w}), \text{ where } F(\mathbf{w}) = \sum_{i=1}^{n} F_i(\mathbf{w}, x_i, y_i)$$
$$x_i \in \mathbb{R}^d, y_i \in \mathbb{R} \text{ for } i = 1, \ldots, n$$
$$\mathbf{w} \in \mathbb{R}^d$$

Here, the x_i's can be interpreted as the input data, the y_i's as the output data, and $\mathbf{w}$ as the computed optimal weights. F_i is some objective function that is suitable for the problem at hand. A few commonly used examples include:

- Least squares
- Support vector machine (SVM)
- Exponential function (logistic regression)
- Forward/backward algorithm (neural network)

In optimization, there are 2 main dimensions that can be scaled:

1. n – number of training points ("data parallelism")
2. d – model size ("model parallelism")

Optimization has aspects that are suited for distributed computing like regularization and hyper-parameter tuning, but these things are quite straightforward and not particular interesting from an algorithmic or distributed design perspective.

References

[1] *Parallel algorithms*, Carnegie Mellon.

[2] J. Nesetril, *On minimum spanning tree problem (translation)*, Elsevier, (2001).