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

Lecture 5, 4/11/2016. Scribed by Henry Ehrenberg, Sagar Vare, Andreas Santucci.

Lecture contents

1. QuickSort
2. Parallel algorithm for minimum spanning trees (Boruvka)
3. Parallel connected components (random mates)

1 QuickSort

First, we'll finish the analysis of QuickSort. The algorithm is as follows.

Algorithm 1: QuickSort

Input: An array A
Output: Sorted A

1 $p \leftarrow$ element of A chosen uniformly at random
2 $L \leftarrow [a | a \in A \text{ s.t. } a < p]$ `// Implicitly:` $B_L \leftarrow \mathbb{1}\{a_i < p\}_{i=1}^n$`, prefixSum`$(B_l)$`,`
3 $R \leftarrow [a | a \in A \text{ s.t. } a > p]$ `// which requires` $\Theta(n)$ `work and` $O(\log n)$ `depth.`
4 **return** [QuickSort(L), p, QuickSort(R)]

1.1 Analysis on Memory Management

Recall that in Lecture 4, we designed an algorithm to construct L and R in $O(n)$ work and $O(\log n)$ depth. Since we know the algorithm used to construct L and R (which is the main work required of `QuickSort`), let us take this opportunity to take a closer look at memory management during the algorithm.

Selecting a pivot uniformly at random We denote the size our input array A by n. To be precise, we can perform step 1 in $\Theta(\log n)$ work and $O(1)$ depth. That is, to generate a number uniformly from the set $\{1, 2, \ldots, n\}$ we can assign $\log n$ processors to independently flip a bit "on" with probability $1/2$. The resulting bit-sequence can be interpreted as a $\log_2$ representation of a number from $\{1, \ldots, n\}$.

Allocating storage for L and R Start by making a call to the OS to allocate an array of n elements; this requires $O(1)$ work and depth, since we do not require the elements to be initialized. We compare each element in the array with the pivot, p, and write a 1 to the corresponding element if the element belongs in L (i.e. it's smaller) and a 0 otherwise. This requires $\Theta(n)$ work but can be done in parallel, i.e. $O(1)$ depth. We are left with an array of 1's and 0's indicating whether an element belongs in L or not, call it $\mathbb{1}_L$,

$$\mathbb{1}_L = \mathbb{1}\{a \in A \text{ s.t. } a < p\}.$$

We then apply `PrefixSum` on the indicator array $\mathbb{1}_L$, which requires $O(n)$ work and $O(\log n)$ depth. Then, we may examine the value of the last element in the output array from `prefixSum` to learn the size of L. Looking up the last element in array $\mathbb{1}_L$ requires $O(1)$ work and depth. We can further allocate a new array for L in constant time and depth. Since we know $|L|$ and we know n, we also know $|R| = n - |L|$; computing $|R|$ and allocating corresponding storage requires $O(1)$ work and depth.

Thus, allocating space for L and R requires $O(n)$ work and $O(\log n)$ depth.

Filling L and R Now, we use n processors, assigning each to exactly one element in our input array A, and in parallel we perform the following steps. Each processor $1, 2, \ldots, n$ is assigned to its corresponding entry in A. Suppose we fix attention to the kth processor, which is responsible for assigning the kth entry in A to its appropriate location in either L or R. We first examine $\mathbb{1}_L$`[k]` to determine whether the element belongs in L or R. In addition, examine the corresponding entry in `prefixSum` output, denote this value by $i =$ `prefixSum`$(\mathbb{1}_L)$`[k]`. If the kth entry of A belongs in L, then it may be written to the position i in L immediately, by definition of how what our `prefixSum` output on $\mathbb{1}_L$ means. If the kth entry instead belongs in R, then realize that index i tells us that exactly i entries "before" element k belong in L. Hence exactly $k - i$ elements belong in array R before element $A[k]$. Hence we know exactly where to write the kth element to R if it belongs there.

Clearly, the process of filling L and R requires $O(n)$ work and $O(1)$ depth.

Work and Depth per Iteration Thus we may say that steps 1,2,3 of our algorithm require $O(n)$ work and $O(\log n)$ depth.[1] The last step of the algorithm requires recursive calls to `Quicksort` and a concatenation of several elements. Realize that if we are clever about how we allocate the memory for our arrays to begin with, this "concatenation" (or lack thereof) can be performed without extra work or depth.[2]

[1] It may be tempting to think that we can get away with *not* calculating a `prefixSum` on $\mathbb{1}_L$ in order to determine the sizes of L and R, and instead pass this indicator array $\mathbb{1}_L$ to our recursive call. We might think that we can then avoid incurring $\log n$ depth. However, realize that then when we pass $\mathbb{1}_L$ to our recursive call, it would still be of size n, hence we would not be decreasing the size of our problem with each recursive call. This would result in quite a poor algorithm.

[2] For details, see lecture 4 notes, specifically the section on "Parallel Algorithms and (Seemingly) Trivial Operations".

1.2 Total Expected Work

Now we analyze the total expected work of QuickSort.[3] We assume all our input elements are unique.[4] On a very high level, to compute the work, note that the work done summed across each level of our computational DAG is n. There are $\log_{4/3} n$ levels in our DAG, hence expected work given by

$$\mathbb{E}[T_1] = \mathbb{E}[\#\text{ levels}] \cdot \mathbb{E}[\text{work per level}] = O(n \log n)$$

Details We define the random indicator variable X_{ij} to be one if the algorithm *does compare* the ith *smallest* and the jth *smallest* elements of input array A during the course of its sorting routine, and zero otherwise. Let X denote the *total* number of comparisons made by our algorithm. Then, we have that

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.$$

Number of Comparisons We take a moment to realize why we sum over $\binom{n}{2}$ elements instead of n^2: the only time in our `QuickSort` algorithm whereby we make a comparison of two elements is when we construct L and R. In doing so, we compare each element in the array to a fixed pivot of p, after which all elements in L less than p and all elements in R greater than p. Realize that pivot p is never compared to elements in L and R for the remainder of the algorithm. Hence each of the $\binom{n}{2}$ pairings are considered *at most* once by our `QuickSort` algorithm.

Expected Number of Comparisons Realize that expectation operator $\mathbb{E}$ is monotone, hence

$$\mathbb{E}[X] \leq \mathbb{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}],$$

where the equality follows from linearity of expectation. Since X_{ij} an indicator random variable,

$$\mathbb{E}[X_{ij}] = 0 \cdot \Pr(X_{ij} = 0) + 1 \cdot \Pr(X_{ij} = 1) = \Pr(X_{ij}).$$

Consider any particular X_{ij} for $i < j$ (i.e. one of interest). Denote the ith smallest element of input array A by $\text{rank}_A^{-1}(i)$ for $i = 1, 2, \ldots, n$. For any one call to `QuickSort`, there are exactly *three* cases to consider, depending on how the value of the pivot compares with $A[i]$ and $A[j]$.

- **Case 1: $p \in \{\text{rank}_A^{-1}(i), \text{rank}_A^{-1}(j)\}$** In selecting a number from $\{1, 2, \ldots, n\}$ uniformly at random, we happen to select an element as pivot from A which is either the ith smallest element in A or the jth smallest element in A. In this case, $X_{ij} = 1$ by definition.

[3] We follow the analysis from CMU very closely.[3]

[4] If there are duplicate elements in our input A, then this only cuts down the amount of work required. As the algorithm is written, we look for strict inequality when constructing L and R. If we ever select one of the duplicated elements as a pivot, its duplicates values are not included in recursive calls hence the size of our sub-problems decreases even more than if all elements were unique.

- **Case 2: $\mathbf{rank}_A^{-1}(i) < p < \mathbf{rank}_A^{-1}(j)$**: The value of the pivot element selected lies between the ith smallest element and the jth smallest element. Since $i < j$, element `i` placed in L and `j` placed in R and $X_{ij} = 0$, since the elements will never be compared.
- **Case 3: $p < \mathbf{rank}_A^{-1}(i)$ or $p > \mathbf{rank}_A^{-1}(j)$**: The value of the pivot is either less than the ith smallest element in A or greater than the jth smallest, in which case either both elements placed into R or both placed into L respectively. Hence X_{ij} may still be "flipped on" in another recursive call to `Quicksort`.

It's possible that on any given round of our algorithm, we end up falling into case 3. Ignore this for now. In doing so, we implicitly condition on falling into case 1 or 2, i.e. we condition on our rank p being chosen so that it lies in the set of values $\{\text{rank}_A^{-1}(i), \text{rank}_A^{-1}(i+1), \ldots, \text{rank}_A^{-1}(j)\}$. Then, the probability that $X_{ij} = 1$ (ignoring case 3) is exactly

$$2/(j - i + 1),$$

Thus,

$$\begin{aligned}\mathbb{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} 2 \sum_{j=i+1}^{n} \frac{1}{j-i+1} \\ &= \sum_{i=1}^{n-1} 2\left(\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-i+1}\right).\end{aligned}$$

Now, note that $H_n = \sum_{i=1}^{n} \frac{1}{i}$ is the Harmonic Number. We note that $\sum_{i=1}^{n} \frac{1}{i} < \int_1^n \frac{1}{x}\mathrm{d}x$, from Calculus,[5] and further that $\int_1^n \frac{1}{x}\mathrm{d}x = \ln n$. Hence we see that

$$\mathbb{E}[X] = \sum_{i=1}^{n-1} 2 \sum_{j=2}^{n-i+1} \frac{1}{j} < 2nH_n = O(n \log n).$$

So, in expectation, the number of comparisons (i.e. the work performed by our algorithm) is $O(n \log n)$.

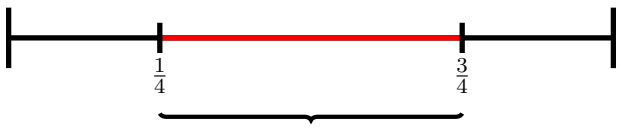
1.3 Total Expected Depth

Recall our analysis in Lecture 4 for `QuickSelect`. In a similar fashion, we may consider the number of *recursive calls* made to our algorithm when our input array is of size

$$\left(\frac{3}{4}\right)^{k+1} n < |A| < \left(\frac{3}{4}\right)^{k} n,$$

and denote this quantity by X_k. We saw that if we select a pivot in the right way, we can reduce input size by 3/4.

We saw that $\mathbb{E}[X_k] = 2$, since it's a geometric random variable (i.e., we continually make calls to `QuickSort` until we successfully reduce our input size by at least 3/4. In expectation, this takes



If we select any of these elements in red as a pivot, where we visualize the elements being in sorted order, *then* both L and R will have size $\leq \frac{3}{4}n$.

Figure 1: Ideal Pivot Selection

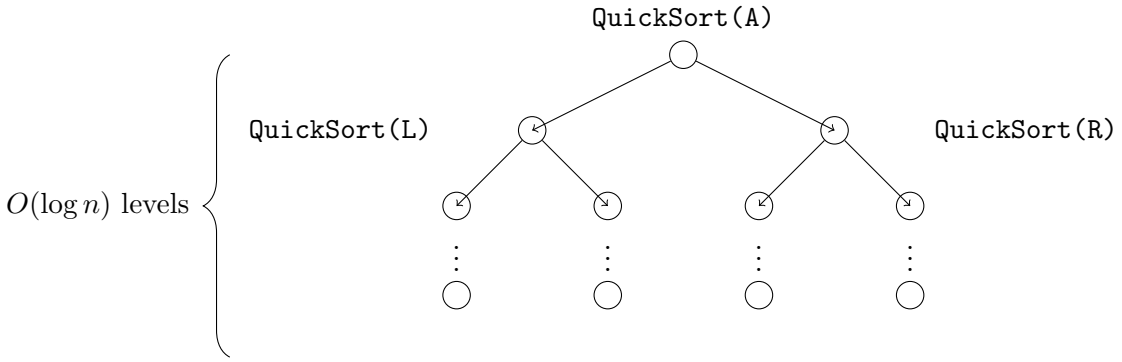


Figure 2: Computational DAG for `QuickSort`

two trials. Hence in expectation we require $2c \log_{4/3} n$ recursive calls to be made before we reach a base case.

Here, we see that in our computational DAG, we have $O(\log_{4/3} n)$ levels. At each level, our algorithm requires $O(\log n)$ depth since the bottleneck is in constructing L and R and specifically in the call to Prefix Sum. Hence in expectation our total depth given by the expected number of levels times the expected depth per level: $\mathbb{E}[D(n)] = O(\log^2 n)$.

1.4 A Shortcut for Bounding Total Expected Work

Examine figure 1.3, the computational DAG for our QuickSort algorithm. We are initially handed an input array of size n elements. At each level of our DAG, the n elements are split into chunks across nodes in that level. Specifically, we're not sure exactly how many elements from A are allocated to L, nor do we know exactly how many elements from A are allocated to R. But we definitely know that there are always n elements we deal with on each level. The most work-intensive operation we perform in each call to QuickSort is in constructing L and R, which requires $\Theta(n)$ work. Hence, for a given level in our computational DAG, we perform exactly $\Theta(n)$ work total after summing across work in all nodes.

So, in this *very specialized analysis*, we have that expected work is

[5]This can easily be seen by comparing a Lower Darboux Sum with a Riemann Integral.

$$\mathbb{E}[W] = (\#\text{ nodes per level}) \times (\#\text{ levels}) = O(n \log n).$$

Since there are $O(\log n)$ levels in our tree in expectation, we perform $O(n \log n)$ work in expectation.

2 Minimum spanning tree algorithms

Now we'll shift our focus to parallel graph algorithms, beginning with minimum spanning trees. In the following sections, we'll denote our *connected* and *undirected* graph by $G = (V, E, w)$. The size of the vertex set $|V| = n$, the size of the edge set $|E| = m$, and we assume that the weights $w : E \to \mathbb{R}$ are distinct for convenience.[6]

A *tree* is an undirected graph G which satisfies any two of the three following statements, such a graph also then satisfies the other property as well.[7]

1. G connected
2. G has no cycles
3. G has exactly $n - 1$ edges

The *minimum spanning tree* (MST) of G is $T^* = (V, E_T, w)$ such that $\sum_{e \in E_{T^*}} w(e) \leq \sum_{e \in E_T} w(e)$ for any other spanning tree T. Under the assumptions on G, the MST T^* is unique and the previous inequality is strict.

2.1 Sequential approaches and Kruskal's algorithm

Sequential algorithms for finding the MST are nice and easy because greedy algorithms work well for this problem. The two clear approaches are Prim's algorithm and Kruskal's algorithm. Here, we'll focus on Kruskal's algorithm, given in Algorithm 2.

Kruskal's algorithm is based on the *cut property* (or *red rule*), which we now prove.

Theorem 2.1 (Cut-Property, Red-Rule) *Let $G(V, E, w)$ be an undirected and connected graph. Then the edge with minimum weight leaving any cut $S \subset V$ is in the MST of G.*

[6] If the edge weights are not distinct, then we may simply perturb each edge weight by a small random factor $\epsilon > 0$, where $\epsilon < 1/n^3$. We no longer have to break ties in our algorithm and at the end, we may simply round back the edge weights to recover the weight of the minimum spanning tree.

[7] To see that $(1, 2) \implies 3$, use induction on the number of nodes n. The base case is trivial. For the inductive step, realize that any acyclic connected graph G must have a leaf v where $d(v) = 1$. Since $G - v$ also acyclic and connected, by the induction hypothesis, $e(G') = n - 2$. Since $d(v) = 1$, we have that $e(G) = n - 1$. To see that $(1, 3) \implies 2$, suppose toward contradiction G has a cycle. To see that $(2, 3) \implies 1$, consider the case that G split into k components. Since G has no cycles, each component G_i both connected and acyclic, hence each is a tree. So total number of edges in G given by $\sum_i (n(G_i) - 1) = n - k$. But since $|E| = n - 1$, $k = 1$.

Algorithm 2: Kruskal's MST algorithm

Input: $G = (V, E, w)$, an undirected and connected graph
Result: T^*, the MST of G

Sort E in increasing order by edge weight $T \leftarrow \emptyset$
while T *not yet a spanning tree* **do**
  $e \leftarrow$ next edge in the queue `// i.e. lightest edge yet to be considered`
  if $(T \cup \{e\})$ *contains no cycles* **then**
    $T \leftarrow T \cup \{e\}$ `// By red rule,` e `belongs in` T^*.
  end
end
return T

Proof: We are given a graph G and a cut $S \subset V$. We say that an edge is in a cut S if exactly one incident node is in S and the other incident node of the edge in $V \setminus S$. Let $e^* = (u, v)$ be the minimum weight edge in cut S. Assume toward contradiction that the edge with minimum weight leaving cut S is *not* in the MST T, i.e. that $e^* \notin T$ where T is the MST of G and

$$e^* = \arg\min_{e \in S} w(e).$$

Suppose $e^* = (u, v)$. Since T is a spanning tree of G, we know that u and v are connected in the edge set of T, i.e. there exists a u-v path in T. Of course, we have assumed that $(u, v) \notin T$. We construct a u-v path using depth first search. Let $(x, y) = e'$ denote the edge in T which crosses cut S in the u-v path. Replace (x, y) with (u, v). Call the result T'.

We claim that $T' = (T \setminus (x, y)) \cup (u, v)$ is still a tree. We first claim T' retains connectivity among all nodes. To see this, realize first that since (x, y) and (u, v) each have exactly one incident node in S and one incident node in $V \setminus S$, then it is *not* possible that either (x, y) or (u, v), when removed from T, could result in S becoming disconnected or $V \setminus S$ becoming disconnected. Hence, without either of these edges, we can get from all nodes in S to all other nodes in S, and same goes for the set $V \setminus S$. It remains to show that we can still traverse from S to $V \setminus S$. Realize that all paths previously going from S to $V \setminus S$ using (x, y) can now simply utilize (u, v) instead.

Further, realize that we have not created any cycles. Specifically, when we add edge (u, v) to the tree T, we induce a cycle. But realize that the edge (x, y) is part of this cycle, and we immediately remove it. Hence in maintaining connectivity and keeping exactly $n - 1$ edges, by definition of a tree we know that the result is acyclic.

Since e^* is the minimum weight edge in S, and since it is not contained in T, we know that $w(e^*) < w(e')$. But then this implies that

$$w(T') = \sum_{e \in T'} w(e) = \sum_{e \in T} w(e) - \underbrace{[w(e') - w(e^*)]}_{>0} < \sum_{e \in T} w(e) = w(T)$$

But this is a contradiction, since T was defined as the MST of G. Thus, e^* is in the MST of G. ■

2.2 Complexity Analysis for Kruskal's

A brief complexity analysis of Kruskal's algorithm is as follows.[8] Recall work is defined as $T_1 = W(n)$.

Sorting Edges Our QuickSort algorithm requires $\mathbb{E}[W(n)] = O(m \log m) = O(m \log n)$.

Checking for Cycles Specifically because we are considering adding a single edge to a tree, we can check if we are inducing a cycle in almost constant time; specifically the work required is $\alpha(m, n)$, where α denotes the Inverse Ackermann function.[2]

We show now quickly a way to check for cycles in $O(\log n)$ work that is a bit more accessible. We must ensure that if edge $e = (u, v)$ to be added, that the component of u is *not* the same component as v. To do this, we keep several data structures around: an array and a Binary Search Tree. We assume that with each component of our graph, we associate with it a binary search tree storing values of nodes in our graph G which are in a particular component k. In addition, we keep an array of length n where in each entry, it tells us which component node i currently in.

Hence, given a candidate (u, v), we go to our length n array and look up in constant time $a[v]$ which tells us that node v in component j. We then go to our BST corresponding to component j, and search in $O(\log n)$ time to check if node u in the same component.

Updating Data Structures if we Add an Edge If we find they share the same component already, we discard the edge and continue. Otherwise, the edge turns out to connect two components, hence we must update our data structures. We go to array A of length n, and write down that u now belongs to v's component.[9] This last step requires $O(1)$ work.

Now, we must merge the component of u with the component of v. Two merge two binary search trees, we first flatten each BST into a sorted linked list with $O(n)$ work using in order traversal. We then merge two sorted lists in $O(n)$ work to get a sorted *array*. We then need to form a BST with our sorted array. To do this, we fix attention to the element in the middle of our array. Realize that all preceding elements in the list are no larger than itself, and all elements following are no smaller. We may make the same observation for the left and right partitions, each time storing pointers from the median element of the list to the left and right children medians. In this way, we re-construct our BST with $O(n)$ work, since we only end up applying a constant amount of work to each element in our sorted array.

Total Work Sorting edges costs $O(m \log n)$ work. Our while loop iterates over at most m edges. Checking for cycles each iteration costs $O(\log n)$ work. Hence we have incurred $O(m \log n)$ work

[8] It's worthwhile to note that since $\binom{n}{2} \leq m \implies m = O(n^2)$, hence any $O(\log m)$ term may actually be replaced by $O(\log n)$, using the fact that $\log m = O(\log n^2) = O(2 \log n) = O(\log n)$. We leave $O(\log m)$ terms in place here for pedagogical purposes.

[9] We resolve the conflict of which node to join to which component by (arbitrarily) choose node index v as the parent since it is larger than node index u.

thus far without even considering how much it costs to update our data structures when we add an edge to our tree.

Realize that we only need to add $n-1$ edges before we construct a tree. Further, each edge we add requires $O(n)$ work. Hence the entire tree construction process takes $O(n^2)$ work. Since $O(n^2) = O(m)$, the work required to sort our edges dominates, and total work required is $O(m \log n)$.

3 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, Theorem 2.1 which we proved above. Our parallel algorithm, which is really Boruvka's algorithm, will also use this property. [4]

3.1 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 theorem 2.1.

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.

3.2 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.

3.3 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 3: 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

3.4 Analysis

3.4.1 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 $O(\log n)$.

3.4.2 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.[10] 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.[11]

Claim 3.1 *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,*

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

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(node v flips Tails) $\times$ Pr(neighbor flips Heads) $\geq \frac{1}{4}$. Hence by linearity of expectation,

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

■

[10] 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.

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

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 $O(\log n)$.

3.4.3 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.

3.4.4 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 $D(n,m) = O(\log^2 n)$.

References

[1] *Parallel algorithms*, Carnegie Mellon.

[2] *A minimum spanning tree algorithm with inverse-ackermann type complexity*, NECI Research Tech Report, (1999).

[3] *Probabilistic analysis and randomized quicksort*, Carnegie Mellon, 15451-s07, (2007).

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