CME 323: Distributed Algorithms and Optimization, Spring 2017
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 5, 4/17/2017. Scribed by 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.[?]

[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 $O(n \log n)$.

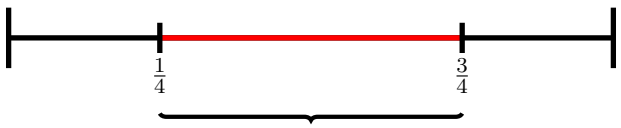
1.3 Total Expected Depth

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

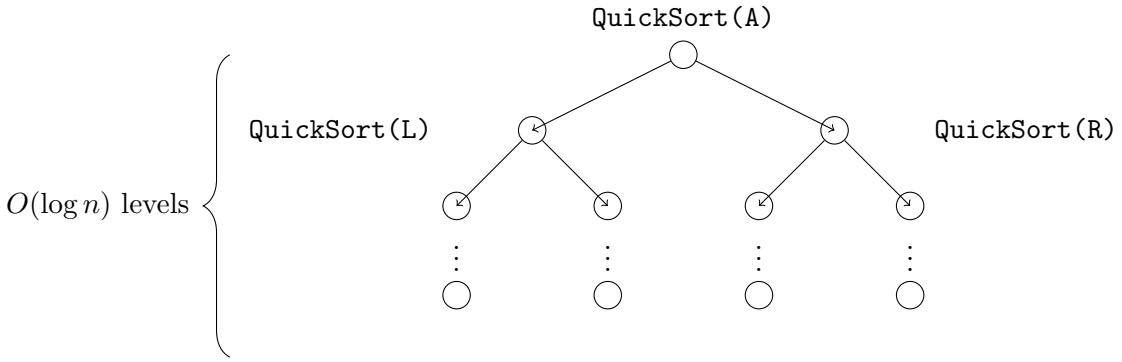


Figure 2: Computational DAG for `QuickSort`

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 Matrix multiplication: Strassen's algorithm

We've all learned the naive way to perform matrix multiplies in $O(n^3)$ time.[6] In today's lecture, we review Strassen's sequential algorithm for matrix multiplication which requires $O(n^{\log_2 7}) = O(n^{2.81})$ operations; the algorithm is amenable to parallelization.[**?**]

A variant of Strassen's sequential algorithm was developed by Coppersmith and Winograd, they achieved a run time of $O(n^{2.375})$.[**?**] The current best algorithm for matrix multiplication $O(n^{2.373})$ was developed by Stanford's own Virginia Williams[**?**].

2.1 Idea - Block Matrix Multiplication

The idea behind Strassen's algorithm is in the formulation of matrix multiplication as a recursive problem. We first cover a variant of the naive algorithm, formulated in terms of block matrices, and then parallelize it. Assume $A, B \in \mathbb{R}^{n \times n}$ and $C = AB$, where n is a power of two.[7]

We write A and B as block matrices,

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},$$

where block matrices A_{ij} are of size $n/2 \times n/2$ (same with respect to block entries of B and C). Trivially, we may apply the definition of block-matrix multiplication to write down a formula for the block-entries of C, i.e.

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

2.2 Parallelizing the Algorithm

Realize that A_{ij} and $B_{k\ell}$ are smaller matrices, hence we have broken down our initial problem of multiplying two $n \times n$ matrices into a problem requiring 8 matrix multiplies between matrices of

[6] Refresher, to compute $C = AB$, we need to compute c_{ij}, of which there are n^2 entries. Each one may be computed via $c_{ij} = \langle a_i^T, b_j \rangle$ in $2n - 1 = \Theta(n)$ operations. Hence total work is $O(n^3)$.

[7] If n is *not* a power of two, then from a theoretical perspective we may simply pad the matrix with additional zeros. From a practical perspective, we would simply use un-equal size blocks.

size $n/2 \times n/2$, as well as a total of 4 matrix additions. There is nothing fundamentally different between the matrix multiplies that we need to compute at this level relative to our original problem.

Further, realize that the four block entries of C may be computed independently from one another, hence we may come up with the following recurrence for *work:*

$$W(n) = 8W(n/2) + O(n^2)$$

By the Master Theorem,[8] $W(n) = O(n^{\log_2 8}) = O(n^3)$. So we have *not* made any progress (other than making our algorithm parallel). We already saw in lecture two that we can naively parallelize matrix-multiplies very simply to yield $O(n^3)$ work and $O(\log n)$ depth.

2.3 Strassen's Algorithm

We now turn toward Strassen's algorithm, such that we will be able to reduce the number of sub-calls to matrix-multiplies to 7, using just a bit of algebra. In this way, we bring the work down to $O(n^{\log_2 7})$.

How do we do this? We use the following factoring scheme. We write down C_{ij}'s in terms of block matrices M_k's. Each M_k may be calculated simply from products and sums of sub-blocks of A and B. That is, we let

$$\begin{aligned}
M_1 &= (A_{11} + A_{22})\,(B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22})B_{11} \\
M_3 &= A_{11}(B_{12} - B_{22}) \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12})B_{22} \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}$$

Crucially, each of the above factors can be evaluated using exactly *one* matrix multiplication. And yet, since each of the M_k's expands by the distributive property of matrix multiplication, they capture additional information. Also important, is that these matrices M_k may be computed independently of one another, i.e. this is where the parallelization of our algorithm occurs.

It can be verified that

$$\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M3 + M_6
\end{aligned}$$

[8] Case 1: $f(n) = O(n^2)$, so $c = 2 < 3 = \log_2(8)$.

Realize that our algorithm requires quite a few summations, however, this number is a constant independent of the size of our matrix multiples. Hence, the work is given by a recurrence of the form

$$W(n) = 7W(n/2) + O(n^2) \implies W(n) = O(n^{\log_2 7}).$$

What about the depth of this algorithm? Since all of our recursive matrix-multiplies may be computed in parallel, and since we can add matrices together in unit depth,[9] we see that depth is given by

$$D(n) = D(n/2) + O(1) \implies D(n) = O(\log n)$$

By Brent's theorem, $T_p \leq \frac{n^{2.81}}{p} + O(\log n)$. In the years since Strassen published his paper, people have been playing this game to bring down the work required marginally, but nobody has come up with a fundamentally different approach.

2.4 Drawbacks of Divide and Conquer

We now discuss some bottleneck's of Strassen's algorithm (and Divide and Conquer algorithms in general).

- We haven't considered communication bottlenecks; in real life communication is expensive.
- Disk/RAM differences are a bottleneck for recursive algorithms, and
- PRAM assumes perfect scheduling.

Communication Cost Our PRAM model assumes zero communication costs between processors. The reason is because the PRAM model assumes a shared memory model, in which each processor has fast access to a single memory bank. Realistically, we never have efficient communication, since often times in the real world we have clusters of computers, each with its own private bank of memory. In these cases, divide and conquer is often impractical.

It is true that when our data are split across multiple machines, having an algorithm operate on blocks of data at a time can be useful. However, as Strassen's algorithm continues to chop up matrices into smaller and smaller chunks, this places a large communication burden on distributed set ups because after the first iteration, it is likely that we will incur a shuffle cost as we are forced to send data between machines.

[9] We note that to perform matrix addition of two $n \times n$ matrices $X + Y = W$, we may calculate each of the n^2 entries $W_{ij} = X_{ij} + Y_{ij}$ in parallel using n^2 processors. Each entry requires only one fundamental unit of computation, hence the work for matrix addition is $O(n^2)$ and the depth is $O(1)$.

Caveat - Big $\mathcal{O}$ and Big Constants One last caveat specific to Strassen's Algorithm is that in practice, the $O(n^2)$ term requires $20 \cdot n^2$ operations, which is quite a large constant to hide. If our data is large enough that it must be distributed across machines in order to store it all, then really we can often only afford to pass through the entire data set one time. If each matrix-multiply requires twenty passes through the data, we're in big trouble. Big $\mathcal{O}$ notation is great to get you started, and tells us to throw away egregiously inefficient algorithms. But once we get down to comparing two reasonable algorithms, we often have to look at the algorithms more closely.

When is Strassen's worth it? If we're actually in the PRAM model, i.e. we have a shared memory cluster, then Strassen's algorithm tends to be advantageous only if $n \geq 1,000$, *assuming no communication costs*. Higher communication costs drive up the n at which Strassen's becomes useful very quickly. Even at $n = 1,000$, naive matrix-multiply requires $1e9$ operations; we can't really do much more than this with a single processor. Strassen's is mainly interesting as a theoretical idea. For more on Strassen in distributed models, see [**?**].

Disk Vs. RAM Trade-off What is the reason that we can only pass through our data once? There is a big trade-off between having data in ram and having it on disk. If we have tons of data, our data is stored on disk. We also have an additional constraint that with respect to *streaming data*, as the data are coming in they are being stored in memory, i.e. we have fast random access, but once we store the data to disk retrieving it again is expensive.

3 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.[10]

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.[11]

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

[10] 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.

[11] 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 $|E| = n - 1$, $k = 1$.

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.

3.1 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.

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 ← ∅
while T not yet a spanning tree do
    e ← next edge in the queue          // i.e. lightest edge yet to be considered
    if (T ∪ {e}) contains no cycles then
        T ← T ∪ {e}                     // By red rule, e belongs in T*.
    end
end
return T
```

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

Theorem 3.1 (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.*

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. ■

3.2 Complexity Analysis for Kruskal's

A brief complexity analysis of Kruskal's algorithm is as follows.[12] 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.[**?**]

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.[13] This last step requires $O(1)$ work.

[12] 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.

[13] 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.

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 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 $O(m \log n)$.