CME 305: Discrete Mathematics and Algorithms
Instructor: Professor Aaron Sidford (sidford@stanford.edu)
January 16, 2018

Lecture 3 - Running Times and Spanning Trees[1]

There are two main topics we cover in this lecture. First, we perform a slightly more formal analysis of the algorithms we provided last week and discuss how they yield provably efficient algorithms for all the problems we have considered so far: global minimum cut, computing connected components, bipartiteness, articulation points, etc. Second, we introduce a new problem, the minimum spanning tree problem and show how to solve it in nearly linear time. We show that the algorithm here and its analysis are actually much more general.

1 Running Times

In the last two lectures we have provided algorithms for efficiently solving several natural graph optimization problems including computing the global minimum cut, computing connected components, computing articulation points, etc. However, in these lectures we were fairly informal with what we meant by *efficiently*. We discussed how for these problems there were trivial brute force algorithms and how our algorithms were more efficient, but we were not very formal in our analysis. Here we discuss a little bit how we will measure algorithm efficiency more formally.

As has been standard in the analysis of algorithms for decades we will focus on the *asymptotic analysis* of algorithms and our analysis will typically be *worst case*. By *asymptotic analysis*, we mean that we will be primarily concerned about how the running times of our algorithms scale as the size of the input grows and be less concerned with constant factors that may arise from minor algorithmic difference. By *worst case*, we mean that we will typically focus on bounding the running times of our algorithms over all inputs that we could possibly receive. Intuitively, we do not want our algorithmic analysis too tailored to the particular computational model or architecture we are working with or assumptions we make about the input. In short we want our analysis to ignore constant factors in the running time that might be tuned to particular architectural details or minor code optimization and we want our algorithm to always run fast, for all input. This is not the only way to analyze algorithms, but it is a classic and important one that has been instrumental in how we view algorithms and optimization problems. There is important work that relaxes these assumptions, but they will not occur much in this course.

Now, in this course when we analyze algorithms we will try to avoid being too pedandic or making too many assumptions about our precise computational model as this is not the primary emphasis of the course. However, we will need to make some assumptions and introduce some standard conventions so that we can talk about and compare algorithm running times. This is something we expect you to have some competence in by the completion of the course. Here, we briefly review our computational model and how we will analyze algorithms. If you haven't seen this before or are not very comfortable with this computational model, at the minimum please read the suggested reading on this subject immediately. If you would like further help, the TAs and I will be happy to discuss this with you and there are a number of additional resources we can and will point you too.

Now, the main mathematical tool we use to analyze running times is something you likely have heard of known as "big-O" notation. By itself, "big-O" notation is simply a mathematical term used to study the asymptotics of functions.

[1] These lecture notes are a work in progress and there may be typos, awkward language, omitted proof details, etc. Moreover, content from lectures might be missing. These notes are intended to converge to a polished superset of the material covered in class so if you would like anything clarified, please do not hesitate to post to Piazza and ask.

Definition 1 (Big-O Notation)**.** For functions $f, g : \mathbb{Z}_{>0} \to \mathbb{Z}_{>0}$ we say that $f = O(g)$, pronounced "f is big-O of g" if for some sufficiently large value n_0 and some constant $c \geq 0$ we have $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. Similarly we say $f = \Omega(g)$, pronounced "f is omega of g," if for some sufficiently large value n_0 and some constant $c \geq 0$ we have $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$. Lastly, we say "$f = \Theta(g)$, pronounced "f is theta of g" if $f = O(g)$ and $f = \Omega(g)$.

Note that we may use the same terminology for functions that take multiple non-negative integers as input with the interpretation that similar bounds should hold for sufficiently large input, i.e. if we have $f, g : \mathbb{Z}_{>0} \times \mathbb{Z}_{>0} \to \mathbb{Z}_{>0}$ then we write $f = O(g)$ if and only if for some $n_0, c \geq 0$ we have $f(n, m) \leq c \cdot g(n, m)$ for all $n \geq n_0$ and $m \geq n_0$. This notation is relevant when we talk about problems parameterized by multiple values, e.g. graphs.

The appeal of this notation is that it provides a natural way to ignore constants and simply focus on the asymptotics or growth rate of functions. Whenever we talk about running time, we will always use big-O notation as we wish to avoid specifying the actual units by which we measure time or the cost of an operation as these may be extremely dependent on the architecture.. We may also use big-O notation to talk about things other than running time as often we just care about asymptotics and the freedom to drop constants is quite useful.

With this notation in hand, I can describe the computational model we will use for the bulk of the class. We will analyze algorithms primarily in the standard *RAM model of computation*. We assume that the input and the state of the algorithm is written down as integers in some long *arrary* or *vector* of integers we call. We then assume that can perform basic operations, like reading from a location in memory, writing to a location of memory, performing arithmetic operations on two memory locations, etc. all in constant, i.e. $O(1)$ time. We then think of our running times as actual functions of the input size, i.e. for graphs with m-edges and n-nodes we write our running times as $T(m, n)$ and bound them with big-O notation imagining that both m or n are growing to infinite.

It is worth noting that in general one needs to be a little careful with these definitions. If we are not careful about how large we let the integers be or how many memory cells get we might get incredibly impractical algorithms. It is known that without these sorts of restrictions intractable problems may look easily solvable. These issues won't generally come up in the class and they are avoidable by making restrictions on each, but we won't really discuss this in the class.

Before analyzing the algorithms we have seen so far, there is one more point worth making. What do we actually mean when we say solve a problem *efficiently*? As we have discussed, when our input is of size n for a combinatorial problem it is often easy to solve the problem in time that is almost *exponential* in the size of the input, i.e. $2^{O(n)}$ time. However, as the input grows the running time of such a procedure can get much worse. The running time we will often first aim for to show that we can do much better than brute force search is to obtain a polynomial running time, i.e. a running time of $O(n^c)$ for a constant c. This is a classic notion of a problem being efficient and it has been extremely useful historically. Obtaining such a running time corresponds to having a much better structural understanding of a problem than brute force search and implies that if the input size doubles for large input, the running time changes by at most multiplicative constant. Also, obtaining such a running time, has often (but not always) been suggestive of obtaining practical and fairly fast algorithms both in theory and in practice.

In this class we will often consider this classic notion of efficiency and strive to obtain polynomial time algorithms. However, we will often attempt to obtain even faster running times. A holy grail for many combinatorial optimization problems is to obtain *linear time algorithms*, that is algorithms with running time $O(n)$ where the input size n. Such a running time implies that the time it takes to solve the problem is not too much more than the time it takes to read the input. Since for many problems it is easy to show that reading the input is necessary, such a running time is optimal. Often we may also attempt to obtain

running times which are close to this bound, or run in *nearly linear time*, i.e. $O(n \log^c n)$ for some absolute constant c. Here these running times differ from linear only by a term that grows slowly asymptotically.

There is active research and important open problems in getting all sorts of running times, i.e. occasionally solving problems in *sublinear time* (which is asymptotically less than the input size) or *quasipolynomial time* $2^{O(\log^c n)}$ but these will not appear too much in this course.

2 Graph Representation

Now before we analyze the graph algorithms we have seen so far, we should discuss a little bit how our graph is represented. Depending on how a graph is specified we may get different running times. There are two natural ways that a graph could be specified.

The first is the one that we will use by default and that is the *adjacency list* representation of a graph. In this representation, to specify a graph $G = (V, E)$ with $n = |V|$ and $m = |E|$, we assume that we have the nodes of the graph $v_1, ..., v_n$ given in a list where we start with the memory location of v_1 and its name together with a pointer to the next node v_2, etc. Further, we assume that at each node, we have a list of its out-edges, i.e. for node u we have a list of (u, v) for all $(u, v) \in E$. If the graph is weighted we further assume that for each edge we have a pointer to its weight. Note that storing this data structure takes $O(n + m)$ space and we can quickly enumerate the edges of a vertex. Note that if we would rather have the in-edge, i.e. who points to a node, we can make this data structure easily in $O(n + m)$ time simply by iterating over the nodes and edges and adding them to the appropriate lists. Consequently, we won't draw a big distinction between keeping track of incoming or outcoming edges.Note that the space taken for this representation is $O(m + n)$ and thus for these problems $m + n$ is the input size.

Another standard representation of graphs as the input that we will work with a little less is known as the *adjacency matrix* representation of a graph. Here the graph is simply stored as a matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ where $\mathbf{A}_{i,j} = 1$ if $(i, j) \in E$ and 0 otherwise. One advantage of this representation over the adjacency list representation is that checking if $(i, j) \in E$ can easily be done in $O(1)$ time (whereas the time for such a query in adjacency lists is a bit more complicated). However, storing it takes $O(n^2)$ space which can be much larger if the graph G is sparse, meaning (informally) that n^2 is much larger than m. OWe will discuss the adjacency matrix further in the class, but we will typically assume that our input graph is specified in adjacency list.

3 Reachability Running Times

Now that we have our basic tools for analyzing algorithms and representing graphs, lets analyze the running time of breadth first search (BFS) and depth first search (DFS) introduced in the last lecture. Rather than going through the code for each, below we provide the following meta algorithm for computing a *reachability tree* from a given start node. Recall that a reachability tree for s is a subgraph where for every node reachable in the graph from s there is a unique path from s to that node.

Now this algorithm is slightly underspecified in that we have not specified which edge (u, v) is removed from F. Depending which edge is removed we get different algorithm, e.g. BFS or DFS. However, before we discuss this, we first quickly prove that the algorithm always returns a reachability tree.

Lemma 2. *The subgraph T returned by the above meta-algorithm is a reachability tree from s.*

Proof. First, we claim that at the start of each iteration of the while loop T is a reachability tree for the

Algorithm 1 Graph Exploration Meta Algorithm

Input: graph $G = (V, E)$ and start vertex $s \in V$
Mark all vertices $v \in V$ as `unexplored` and initialize $T = \emptyset$
Let $F = \{(u, v) \in E \mid u = s\}$ and mark s as `explored`
while $F \neq \emptyset$ **do**
  Remove edge (u, v) from F
  if v is `unexplored` **then**
    Add (u, v) to T
    Mark v as `explored`
    Add (v, w) to F for all $(v, w) \in E$
  end if
end while
Return the reachability tree T

subgraph induced by the vertices marked as `explored`. This can be improved by induction. The claim starts as trivially true when $T = \emptyset$. Furthermore, if it is true at the start of the while loop we either leave the tree unchanged or add a single edge from the vertices marked as `explored` to a new vertex (that that we then mark as `explored`) the claim stays true.

With this claim established, it only remains to show that set of vertices marked as `explored`, denoted S, contains all vertices reachable from s, denoted R, at the termination of the algorithm. Proceed by contradiction and suppose there is a vertex $v \in R \setminus S$. Since $v \in R$ there is a s to v path and this path must have some edge $(a, b) \in E$ with $a \in S$ and $b \notin S$ as $s \in S$ and $v \notin S$. However, when we marked a as `explored` we would have added (a, b) to F and therefore would have eventually marked b as explored adding b to S, contradicting to the definition of S. □

Now let us analyze the running time of the graph exploration algorithm. Marking the vertices explored or unexplored takes $O(n)$ time in total as there are n vertices and each is marked as explored once and unexplored once. The number of iterations of the while loop is at most $2m$ as every undirected edge is added to F at most twice (once if the edge is directed). Furthermore, we only add at most $n-1$ edges to T so that takes $O(n)$ time. Consequently we see that our running time is $O(n + m)$ plus the time it takes to perform the following operations on a set F $O(m)$ times, (1) check if F is empty, (2) add an element to F, (3) remove an element from F. Now there are many data strutures that can implement each of these operations in $O(1)$ time each. I will not spend too much time in these notes discussing how to implement such data structures, but I will point to references for further information about them. With this, we see that we can compute a reachability tree in $O(m+n)$ time and it is not hard to see that if we apply it repeatedly for different vertices, removing all vertices and edges they can reach, this will ultimately compute all the connected components of an undirected graph in $O(m + n)$ time. We simply need to show how to use this algorithm to implement BFS and DFS.

Suppose that F is implemented in what is called *FIFO*, *first-in first-out*, this means that the element removed from F is the element that was added earliest. In this case, we see that the reachability algorithm will first explore vertices reachble from s with one edge, then the vertices reachable from those in one step, etc. Consequently, this algorithm will be precisely BFS. Such a FIFO F can be implement so that all operations take $O(1)$ time and such a data-structure is known as a *queue*. Thus, BFS can be implement in linear, $O(m + n)$, time.

On the other hand suppose that F is implemented in what is called *LIFO*, *last-in first-out*, this means that the element removed from F is the element that was added most recently. In this case, we see that the reachability algorithm will follow an edge to an `unexplored` vertex, follow an edge from that vertex to an

`unexplored`, etc. backtracking only when no edge would bring the algorithm to an `unexplored` vertex. Consequently, this algorithm will be precisely DFS. Such a LIFO F can be implement so that all operations take $O(1)$ time and such a data-structure is known as a *stack*. Thus, DFS can be implement in linear, $O(m+n)$, time.

4 Strongly Connected Component / Articulation Point Algorithm

... coming soon ... (we touched on this briefly in lecture #2 and it will be part of these notes shortly) ...

5 Minimum Cut Running Time

With our algorithm analysis conventions in hand, let us take a closer look at the running time for Karger's algorithm and discuss how to improve it. Recall that the trivial, brute force running time for min-cut is $O(2^n m)$ which is achieved by looking at all $2^n - 2$ possible cuts, computing each of there sizes, which takes $O(m)$ time for each, and then outputting the best cut.

5.1 Karger's Algorithm

Recall Karger's algorithm is as follows:

Algorithm 2 Karger's randomized global min-cut

repeat
  Remove all self-loops
  Choose an edge $\{u, v\}$ uniformly at random from E.
  Contract vertices u and v
until G has only 2 vertices.
Report the corresponding cut in the original graph

We assume that we can sample a random edge from a list in $O(1)$ time (this isn't too hard to make happen). With this each contraction can be implemented to takes $O(1)$ time. When a contraction happens we can simply enumerate the edges edges and renumber everything to encode the contraction, this takes at most $O(m)$ time. We then contract $O(n)$ times for one execution of Karger, causing Karger to run in $O(mn)$ time.

Now, as we discussed before Karger's algorithm succeeds in finding a min-cut with probability $\geq \frac{1}{O(n^2)}$. Thus, if we run Karger t times and return the smallest cut found, this will takes us $O(tmn)$ time for the t executions of Karger, plus $O(tm)$ time to find the best cut, causing a total running time of $O(tmn)$. Thus as we discussed made the success probability

$$1 - \left(1 - \frac{1}{O(n^2)}\right)^t \geq 1 - \exp\left(-\frac{t}{O(n^2)}\right)$$

and therefore if we picked $t = O(n^2 \log n)$ then we obtain a success probability of $1 - \frac{1}{n^c}$ where c is any constant we wish, as it affects the running time only through the constant in the big-O. This is known as succeeding with high probability (whp) in n and thus we have just shown that we can compute global min cut whp in time $O(mn^3 \log n)$. This is a polynomial running time much better than the trival brute force.

Note that we can improve the implementation of Karger a little bit, by storing multi-edges just as weighted edges and noting that picking a random edge in the graph is same as picking a random vertex with probability

proportional to the degree of that vertex (the total weight of the edges incident to it), and then picking a random edge incident to that vertex. Consequently, with $O(m)$ preprocessing we can compute the degrees of all the vertices and then implement a contraction in $O(n)$ rather than $O(m)$ time simply by sampling the right vertex in $O(n)$ time, sampling the right edge in $O(n)$ time and then contracting and updating the information to do this again in $O(n)$ time. Consequently, Karger can be implemented in $O(m+n^2)$ time, which I will just write as $O(n^2)$ (under the implicit assumption that the original possibly multi-graph is given as a weighted graph). Thus, Karger's algorithm appropriately used gives $O(n^4 \log n)$ time algorithm.

5.2 Improving the Running Time (Karger Stein)

One way to improve the running time is a beautiful idea of Karger and Stein. We sketch this in the remainder of this section. The idea is that we if we contract only some of the edges of the graph (rather than all), we can still contract many edges while maintaining that that the success probability is still fairly high. Suppose we contract until the number of vertices in the graph is some n_{new}, then failure probability is

$$\left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)...\left(1-\frac{2}{n_{new}+1}\right)=\left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)...\left(\frac{n_{new}-1}{n_{new}+1}\right)=\frac{n_{new}(n_{new}-1)}{n(n-1)}\approx\frac{n_{new}^2}{n^2}.$$

Consequently if we contract until $n_{new} \approx \frac{n}{\sqrt{2}}$ then we actually succeeded in preserving a mincut with probability $\frac{1}{2}$. How can we then improve our success probability? One natural idea would be instead of running the same procedure on the resulting graph once, we can run it twice. Since these trials are independent this will improve our success probability. If we use this procedure recursively, it could improve performance by a lot. This motivates the following algorithm.

Algorithm 3 Karger and Stein' Randomized Global Min-cut

Define `solve`$(G=(V,E))$ as follows.
Let $n=|V|$ and if $n=O(1)$ compute the global min-cut and return its corresponding cut in original graph.
Note: any algorithm work for above step, even brute force, since the graph is constant size.
while $|V| > \frac{n}{\sqrt{2}}$ **do**
  Remove all self-loops
  Choose an edge $\{u,v\}$ uniformly at random from E.
  Contract vertices u and v
end while
Return the corresponding cut in the original graph for the smaller cut size of `solve(G)`and `solve(G)`

First, let's roughly compute the success probability the entire procedure succeeds? Let p_k denote the probably of successfully outputting a minimum cut when calling `solve`$(G=(V,E))$ twice and outputting the smaller cut, on a graph where we will need to recurse k times before the graph is constant size. Clearly, $p_0 = 1$ as discussed. To bound p_k note that contracting until the number of vertices decreases by $1/\sqrt{2}$ and then calling the procedure twice in each invocation of `solve`$(G=(V,E))$ will succeed with probability at least $\frac{1}{2}p_{k-1}$. Since we do this twice independently we will fail only if they both fail which happens with probability

$$p_k \geq 1-\left(1-\frac{1}{2}p_{k-1}\right)^2 = p_{k-1}-\frac{1}{4}p_{k-1}^2 = p_{k-1}\left(1-\frac{p_{k-1}}{4}\right).$$

Now, since $p_{k-1} \in [0,1]$ we have that

$$\left(1-\frac{p_{k-1}}{4}\right)\left(1+\frac{p_{k-1}}{2}\right)=1+\frac{1}{4}p_{k-1}-\frac{1}{8}p_{k-1}^2 \geq 1$$

and therefore

$$\frac{1}{p_k} \leq \frac{1}{p_{k-1}\left(1-\frac{p_{k-1}}{4}\right)} \leq \frac{1+\frac{p_{k-1}}{2}}{p_{k-1}} = \frac{1}{p_{k-1}} + \frac{1}{2}\,.$$

Consequently, $\frac{1}{p_k} \leq k$ and $p_k \geq \frac{1}{k}$. Since we recurse $\log_{\sqrt{2}}(n) = O(\log n)$ times this means the overall success probability is $\frac{1}{O(\log n)}$ and therefore, we succeed with high probability if we run the procedure $O(\log^2 n)$ times!

What is the overall running time of the procedure? Note, that as we discussed earlier the contractions can be implemented in $O(n^2)$ time and therefore the running time of this procedure on a n-vertex graph is $T(n)$ where $T(n)$ obeys

$$T(n) = O(n^2) + 2 \cdot T(n/\sqrt{2})\,.$$

There are formulas for solving these recursions and the well known "master formula" yields that $T(n) = O(n^2 \log n)$. Putting everything together yields that we can compute a minimum cut with high probability in $O(n^2 \log^3 n)$ time. Note this is a substantial improvement. When the graph is dense, i.e. $m = \Omega(n^2)$, this running time is nearly linear!

5.3 State-of-the-art

Now a natural question is can we do even better? For example can we get a linear running time even when the graph is not dense, i.e. m is much less than n^2? Note that the algorithm would have to change substantially to achieve this. By similar arguments to earlier we see that this procedure will output every single global minimum cut. There are $\Omega(n^2)$ of them so unless we do something to represent them better this will trivially take us $\Omega(n^2)$ time to get. There was a breakthrough result of Karger that got around this bottleneck and achieved this running time, but we will likely not get to discuss it further in this course. If you would like further information about it, let me know.

6 The Minimum Spanning Tree Problem

We conclude this lecture by taking a look at a new problem known as the minimum spanning tree (MST) problem.

Definition 3. Given an undirected graph $G = (V, E)$ and weights $w \in \mathbb{R}^E$ a *minimum spanning tree (MST)* is a spanning tree $T = (V, E_T)$ of G with $\sum_{e \in E_T} w_e$ minimized. The *MST problem* asks to efficiently compute a MST.

This is an extremely natural and well studied problem in combinatorial optimization. As we have shown, a spanning tree consists of the minimum number of edges that one could hope to keep and maintain that a graph is connected and therefore this problem asks for the cheapest such certificate. One might also imagine a settings where nodes are things that need to be connected (e.g. computers) and the edges correspond to links we could possibly purchase to compute them (at some known cost), in this context the MST problem asks for the minimum cost we need to pay to connect the entire network. The study of this problem has also lead to numerous algorithmic insights and as we will discuss next lecture the MST problem constitutes one of the simplest where natural algorithmic paradigms, like *greedy*, work.

So how do we solve the MST problem? As we just suggested, one of the most natural algorithms one might thing of is a *greedy* algorithm: start with an empty set and repeatedly add the minimum weight edge that doesn't induce a cycle until the graph is connected (i.e. we have a tree). This algorithm in the context of MST is known as Kruskal's algorithm:

Algorithm 4 Greedy MST Algorithm (Kruskal's Algorithm)

Input: graph $G = (V, E)$ and weights $w \in \mathbb{R}^E$
Let $T := \emptyset$
while $|T| < n - 1$ **do**
  add to T the smallest weight edge in G that does not create a cycle in T
end while
Return T

This algorithm is greedy as it greedily repeatedly adds the best edge it can. The algorithm never looks back and removes an edge; once it determines which edge to add it is final. Before we prove this algorithm works, let's show how it can be implemented efficiently, i.e. in nearly linear time. Consider the following:

Algorithm 5 Greedy MST Algorithm Implementation

Input: graph $G = (V, E)$ and weights $w \in \mathbb{R}^E$
Set $T = \emptyset$
Set components $S_i = \{v_i\}$ for all $i \in [n]$
Set pointers $C[v_i] = S_i$
Sort edges $E = \{e_1, ..., e_m\}$ so that $w_{e_1} \leq w_{e_2} \leq ... \leq w_{e_m}$.
for $i = 1$ to m **do**
  if If $e_i = \{u_i, v_i\}$ and $C[u_i] \neq C[v_i]$ **then**
    Add e_i to T
    Swap u_i and v_i so that $|C[u_i]| \leq |C[v_i]|$
    $C[v_i] = C[v_i] \cup C[u_i]$
    Set $C[u] = C[v_i]$ for all $u \in C[u_i]$
  end if
end for
Return T

This procedure simply assign each vertex v_i to a connected component $C[v_i]$ initialized to be $\{v_i\}$, as the connected components are just the vertices to start. Then the edges are sorted and considered in increasing order of weight. Whenever an edge is encountered that is between two different connected components ($C[u_i] \neq C[v_i]$) the smaller connected component is then merged into the larger connected component and the edge $\{u_i, v_i\}$ is added to the tree. Since edges do not form cycles if and only if they are between connected components this algorithm works the same as Kruskal's algorithm To bound the running time, note that sorting can be performed in $O(m \log m)$ time (which can be made $O(m + m \log n) = O(m \log n)$ time simply by taking the minimum weight edge of any multi-edge and then sorting the remaining at most n^2 edges), and then the each of the operations can be performed in $O(1)$ so long as the sets are stored in linked lists. The loop over the edges proceeds $O(m)$ times and thus the running time is simply $O(m \log n)$ plus the time needed to do all the changing of $C[u] = C[v_i]$. However, a vertex is set to a new connected component in this way if and only if the connected component it is in at least doubles in size (as we always merge the smaller component into the larger). Consequently this step can only happen $O(\log n)$ times per vertex and thus $O(n \log n)$ time in total. Consequently, the total running time is $O(m \log n)$.

Note that throughout we assumed that $m \geq n$ as we were assuming there is a spanning tree and the graph is connected, otherwise there would be an additive n in the running time. Also note that there are much faster procedures to implement this algorithm (ignoring the time to sort), in particular a data structure known as *union-find* can be used to almost remove the $\log n$ factor.

To show that this algorithm is correct we provide a fairly general lemma about the structure of edges in a MST.

Lemma 4. *If for some non-trivial $S \subseteq V$ and $e \in \partial(S)$, where $\partial(S) = \{\{i,j\} \in E | i \in S, j \notin S\}$, we have $w_e < w_{e'}$ for all $e' \neq e$ in $\partial(S)$ then e is in every MST.*

Proof. Let $e = \{i,j\}$ for $i \in S$ and $j \notin S$ and proceed by contradiction supposing that $T = (V, E_T)$ is a MST not containing e. Then there is path connecting i to j in T and one of these edges e' must also be in $\partial(S)$ as $i \in S$ and $j \notin S$. However, if we remove edge e' from the tree and add edge e then the total weight of the tree decreases as $w_e < w_{e'}$ and the graph must still be a tree since everything that was connected before is still connected (as there is a path between the endpoints of e' in the new graph. Consequently, T is not minimum weight and not the MST. □

Now, when we run Kruskal's algorithm every time we add an edge it is a minimum in the cut induced by the connected component it connects. Thus if we just define the weight of this edge to be infinitesimally smaller then the other edges of the same weight not included but larger than those included we can apply the above lemma and have that the resulting tree is a MST. Note that this lemma can be used to prove correctness of many possible MST algorithms. For example, if we took the graph exploration meta algorithm and deleted from F the minimum weight edge in F (this can be done in $O(\log n)$ time using a data structure known as a heap or priority queue) then this algorithm (known as Prim's algorithm) also yields a MST. In the next lecture we will talk a little more broadly about when such algorithms work from the perspective of matroids.