Algorithms Programming PS1

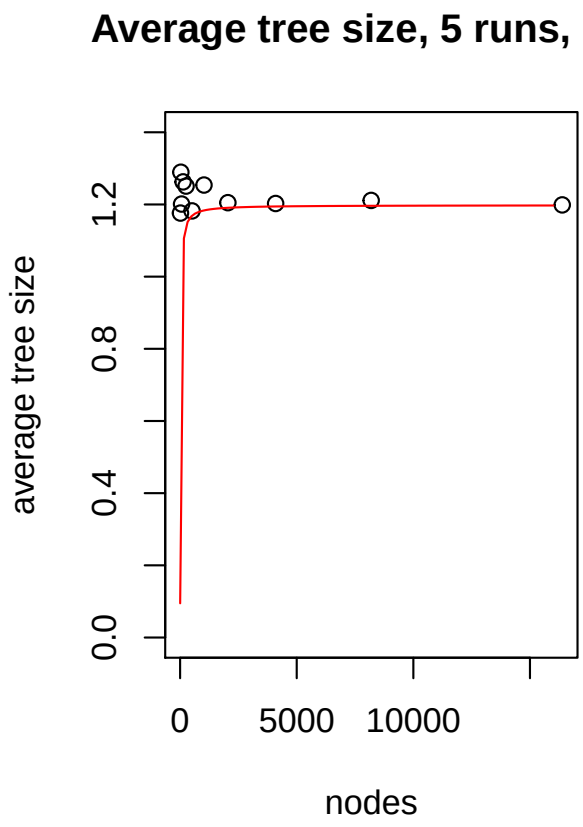
Josh Kaplan & John Hotchkiss

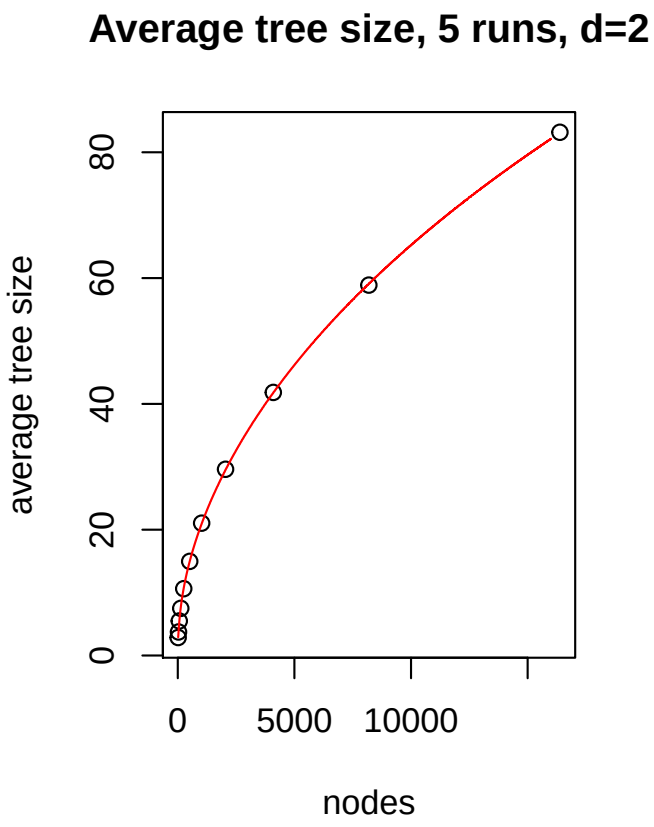
3/2/2017

Algorithm Choice

To begin our search for an algorithm for this assignment, we decided it would be safest to stick to those that we learned in class. That meant we needed to decide between Kruskal's and Prim's algorithms. After considering our abilities (we could only reasonably expect to use a binary heap for Prim's), we looked at the complexities for each algorithm. In particular, Prim's with a binary heap runs in $O\left((|E| + |V|) \log |V|\right) = O\left(|E| \log |V|\right)$ time. Kruskal's runs in $O\left(|E| \log |V|^2\right) = O\left(2 * |E| \log |V|\right) = O\left(|E| \log |V|\right)$ time. Clearly, then, Prim's and Kruskal's are asymptotically the same given our intended implementation strategies. As such, we decided to pursue Prim's algorithm because we felt more comfortable with a binary heap than we did with a UnionFind data structure.

Part 1





For $d = 0$, the tree size basically stops increasing after about 32 nodes, settling down to about 1.2. We tested smaller graph sizes than 16 in order to fit the curve, and found that arctan works particularly well because it

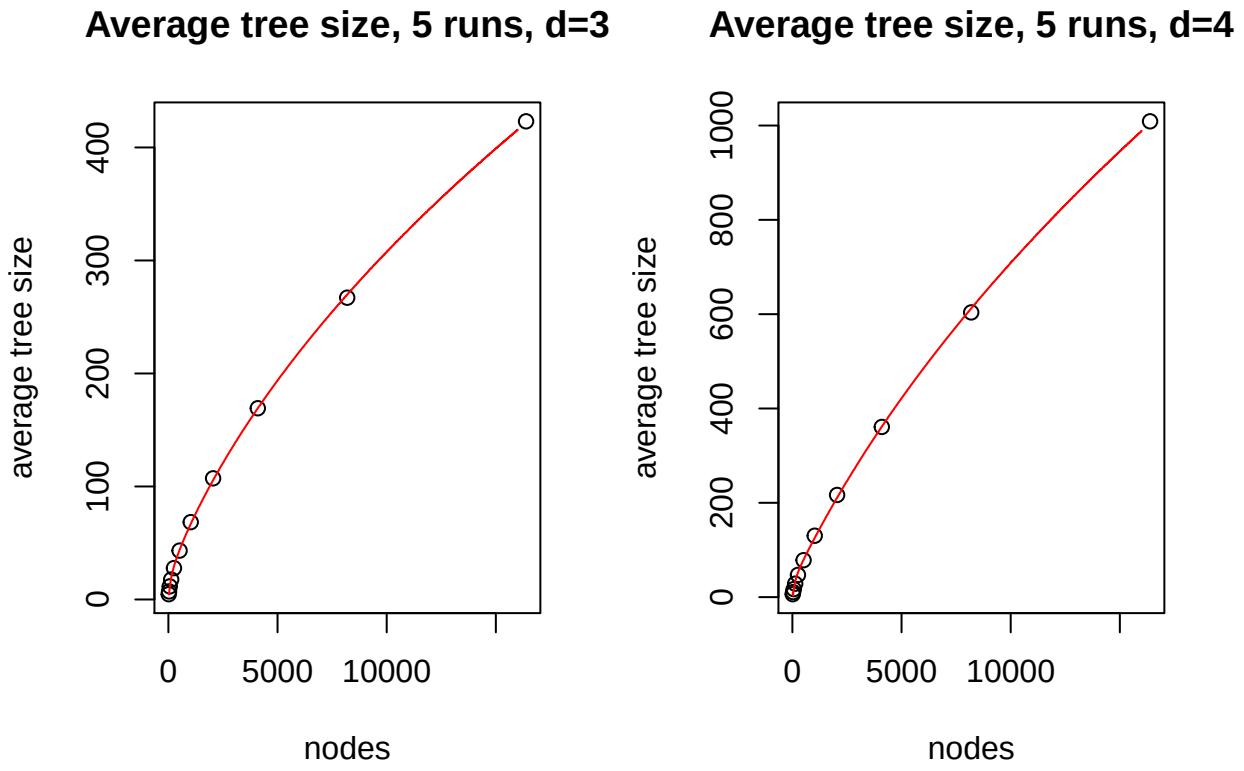
also settles down very quickly (An analysis of applying arctangent follows in Part 2). For $d = 0$,

$$f(n) = .75 * \arctan(n/20) + .02$$

$d = 2$ clearly exhibits different behavior - the tree size increases with the nodes at a decreasing rate, characteristic of a logarithmic function. We used regression to find a function that exactly fits our data points (The R^2 was $> .999$). The result was:

$$f(n) = 19.9223713 + -16.6133158 * \log(x) + 5.5573793 * \log(x)^2 + -0.7673005 * \log(x)^3 + 0.0453579 * \log(x)^4$$

The y-intercept is not very meaningful in this cases; it is set so that the model fits, but it doesn't make a lot of sense because the model is not based on any data from $n = 0 - 15$.



Again, we used regression to find functions that exactly fit our data points. The functional form for d=3,4 was basically identical to d=2, only the model coefficients changed. For d=3,

$$f(n) = 211.2809282 + -186.5858728 * \log(x) + 60.2758222 * \log(x)^2 + -8.3623451 * \log(x)^3 + 0.4495789 * \log(x)^4$$

and for d=4,

$$f(n) = 667.4062802 + -588.7571167 * \log(x) + 188.1568835 * \log(x)^2 + -25.8921268 * \log(x)^3 + 0.4495789 * \log(x)^4$$

Again, the y-intercepts are not especially meaningful in these cases because the model does not include data from $n = 0 - 15$.

Part 2

Growth rates of f(n)

For d=2,3, and 4, the logarithmic behavior of f(n) makes sense. As n increases, each node has more edges coming off of it (since the graph is complete), so Prim's algorithm has exponentially more edges to choose from in selecting edges of minimum weight. Thus, you would expect the average edge weight to decrease as n increases. On the other hand, there are more nodes in the graph, so the MST must contain more edges in order to connect them all. So you would expect the average tree size to increase. The nodes (and the number of edges in the tree) increases linearly, while the average edge weight decreases logarithmically. Thus, we see logarithmic growth in the size of the tree. This pattern holds for graphs in any dimension >1, but higher dimensions result in overall higher edge weights, so the average tree size is a constant factor higher for all n.

We were surprised to see the quick convergence of the tree size in d=0. After about 30 nodes, the tree doesn't grow no matter how many nodes are added to the graph. After some internet research, we discovered that the limit on the weight of the spanning tree, as n goes to infinity, is equal to Riemann zeta(3) ≈ 1.2.

General Implementation Strategy/Experience

Always, we simply tried to implement aspects of the algorithm in the most straightforward way possible, but this led to many inefficiencies we eventually needed to remove. For example, we began by learning how to generate a graph with a list containing all of the vertices and a list containing all of the edges and edgeweight information. Then, we fed these things to our implementation of Prim's. Because edges were a simple list (node1,node2,edgeweight), which worked fine for graph representation, we had to expend a lot of runtime and effort extracting the information back out of that data format in Prim's.

In particular, consider one of the deeper lines of pseudo code for Prim's: for $(v, w) \in E$ and $w \notin S'$. This required looping through all of our edges, E, extracting the node information, checking to see if the source node, v, was either of the nodes associated with the edge, u OR w, then assigning w to whichever the source was not, all before continuing to the next line of the algorithm.

This issue stemmed from not approaching the task more holistically. Many efficiencies were gained when we stopped treating the tasks (generating a graph and creating a minimum spanning tree) as two separate tasks and instead one larger task.

Finally, our implementation made clear that the devil is in the details. It is one thing to consider an algorithm as pseudocode and entirely another to make it functional. For example, we needed to change our heap so that it would notice whether or not a node that was trying to be inserted was already on the heap, and if so, would change the value of that node instead of adding a duplicate. This required adding a dictionary to the internals of the heap that would keep track of the index of particular nodes within the heap.

Optimization Explanations

We made a few optimizations. Some of them were quite small and probably made little difference. However, the two largest optimizations are below:

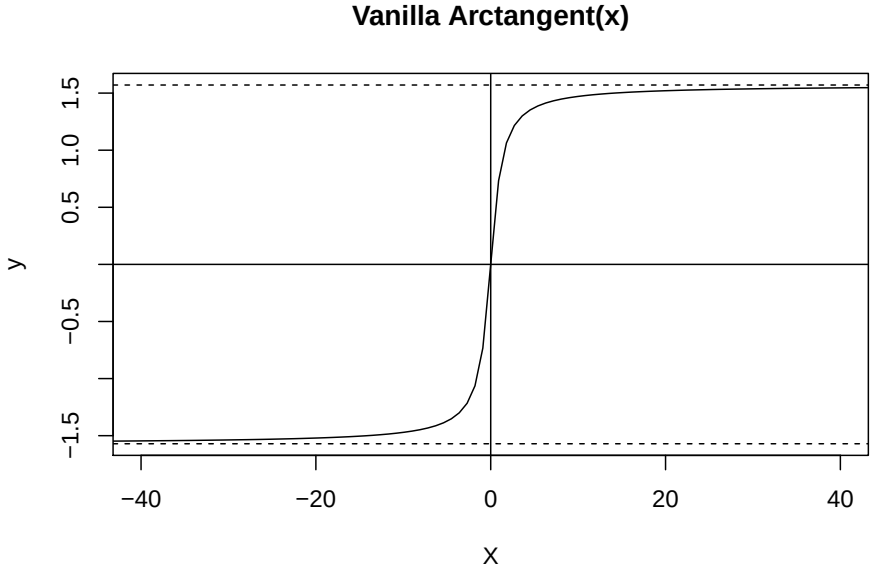
1. Following the hint in the assignment, we attempted to find a weight threshold as a function of n, $k(n)$.
2. Storing edges in a dictionary format. This stores all of the edges that we keep twice, once for each node belonging to the edge. Entries in the dictionary are (node, edgeweight).

Determining edge weight threshold k(n)

To find a threshold function, $k(n)$, we first gathered data from the algorithm. We collected the maximum edge weight over 5 runs for n as a power of 2 at the same levels as those we used for the other graphs. After

putting those values over a scatter plot, we fit a curve to them using the strategy described below.

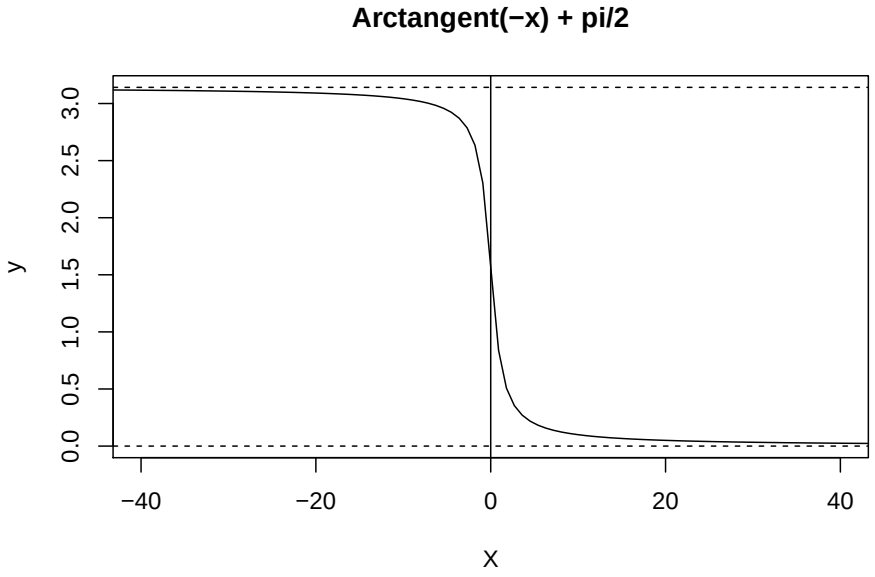
We chose to model the curve using a tangent function due to its desirable mathematical properties. In particular, we thought it would be helpful to have a curve with built in asymptotes. The asymptotic properties can help to mitigate any odd behavior at extremely large values of n that could occur in equations without such properties (such as an eventual droop to a negative threshold or a threshold equal to zero). Recall an arctangent curve without modification,



Note the asymptotes at $\frac{\pi}{2}$ and $-\frac{\pi}{2}$. By negating x we can flip the function over the x-axis. Then, by adding $\frac{\pi}{2}$ the bottom asymptote will lie on $y = 0$. So for

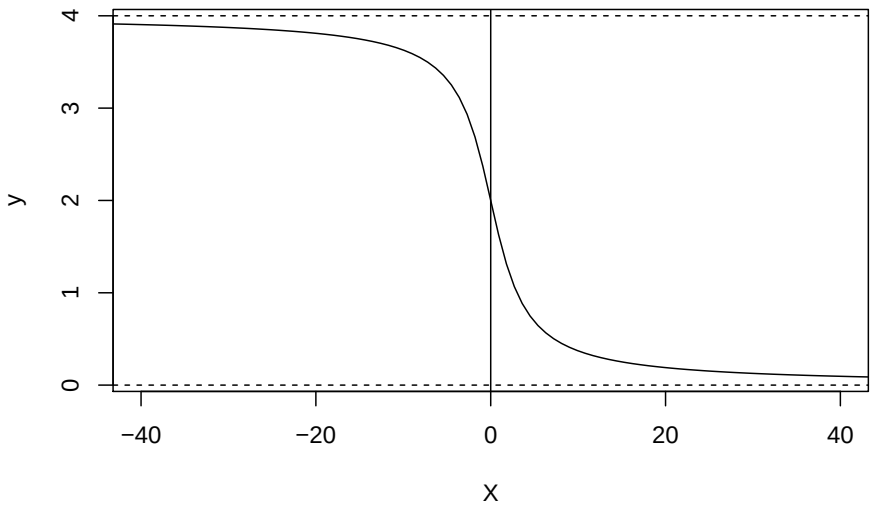
$$\arctan(-x) + \frac{\pi}{2},$$

we have:



Now consider that by dividing x we can make the slope less steep. Furthermore, by multiplying the entire equation by a value we can change the height of the bend (and increase the y-intercept) without moving the asymptote at $y = 0$. Consider

$$\frac{4}{\pi}\left(\arctan\left(-\frac{x}{2}\right)+\frac{\pi}{2}\right).$$



Using the sorts of manipulations described above, we could modify the tangent curves to fit the plots, as you can see below for each dimension:

Dimension 0:

$$\frac{6}{\pi}\left(\arctan\left(-\frac{x}{20}\right)+\frac{\pi}{2}\right)$$

Dimension 2:

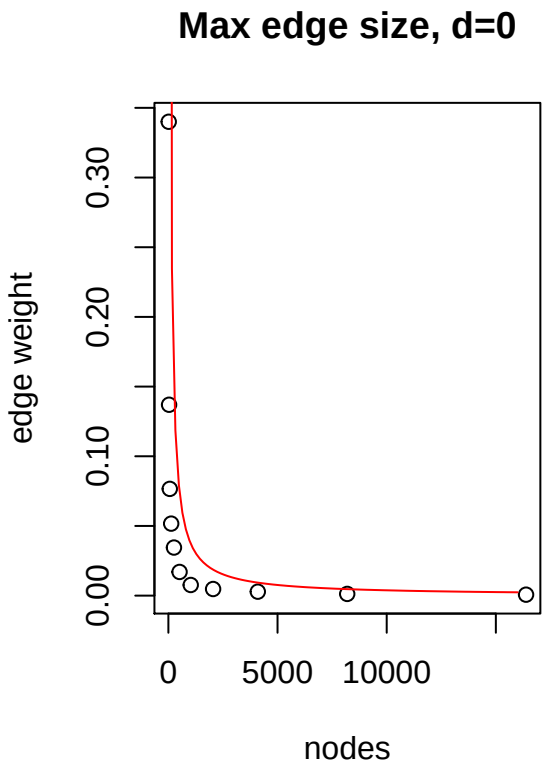
$$\frac{3}{\pi}\left(\arctan\left(-\frac{x}{35}\right)+\frac{\pi}{2}\right)+0.03$$

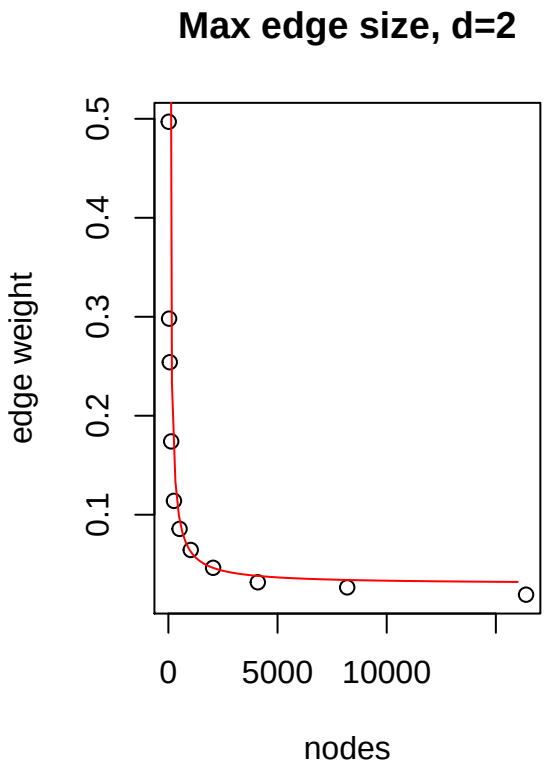
Dimension 3:

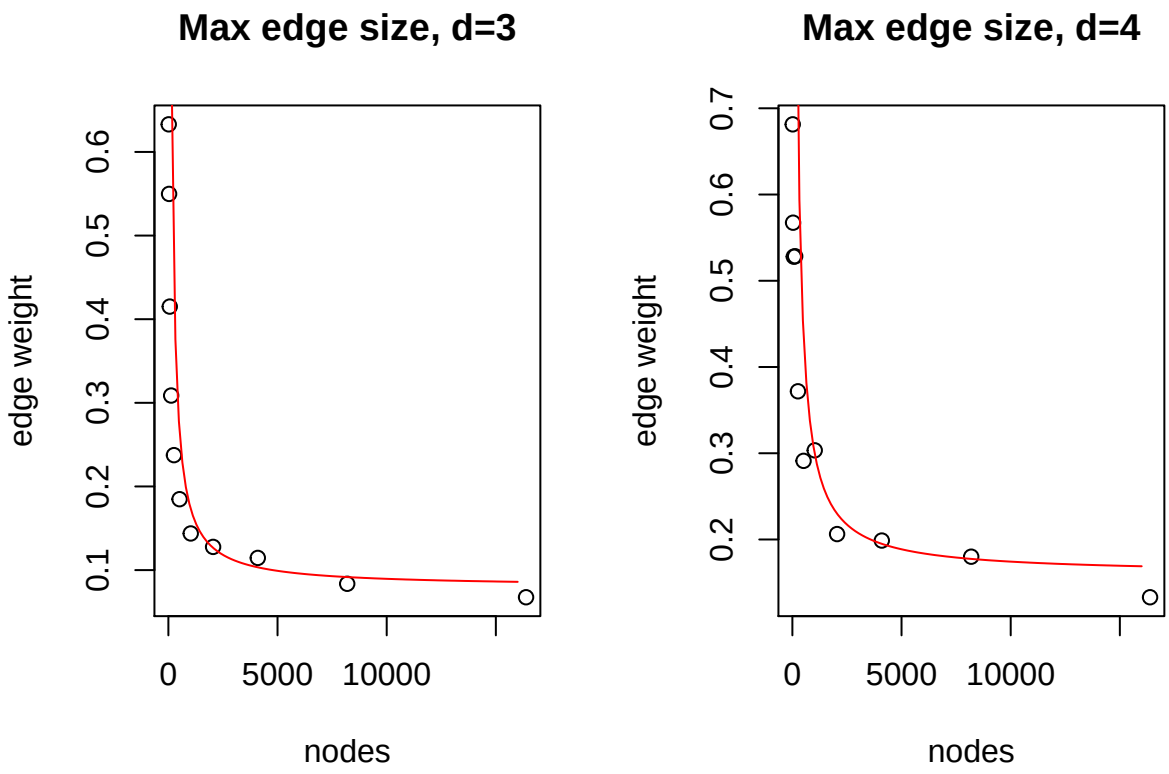
$$\frac{10}{\pi}\left(\arctan\left(-\frac{x}{30}\right)+\frac{\pi}{2}\right)+0.08$$

Dimension 4:

$$\frac{5}{\pi}\left(\arctan\left(-\frac{x}{90}\right)+\frac{\pi}{2}\right)+0.16$$







In the event that the pruned graph is too pruned, our algorithm will call itself with a higher threshold (previous threshold + 0.01) and try again. This was a necessary addition because it is possible to remove too many edges. In the event that too many edges are removed, Prim's algorithm will be unable to create a spanning tree, and will stop before all nodes have been reached (there will be nodes with minimum distances still equivalent to infinity). It does not, however, fail by creating the wrong tree. This is because if there is not an edge to connect the node to the tree, it is because they were all thrown out. If there *is* an edge or multiple edges, they must be the shortest edges that connect to that node, in which will be included the shortest. It is not possible to keep a longer edge than the shortest, because by definition the shortest would have been kept also.

Applying a Dictionary to Store Edges

Another big improvement in runtime came when we stopped having to loop over all edges in order to find those that were involved in a particular node, v. This applies to the line referenced earlier from the pseudo code, for $(v, w) \in E$ and $w \notin S'$, and is a response to looking at the task at hand more holistically. Instead of restricting ourselves to a full list of edges, we simply save edges from the time of generation into a dictionary and only if they pass the threshold. In this way, we never actually have the complete list of edges stored. For the dictionary, keys are nodes. Values for the dictionary are the partner node and the weight of the edge they share. That means each edge appears twice in the dictionary (once for each node).

This and other small changes changed our "for edges"" loop from

```
for e in E:
    if d!=0:
        enodes = [tuple(i) for i in e[0]]
    else:
```

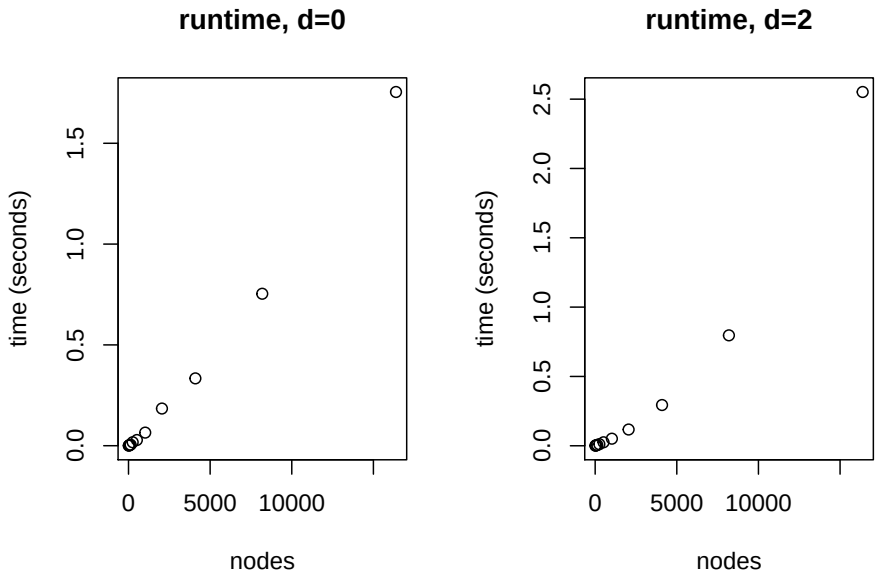
```
        enodes = e[0]
    if v[0] in enodes:
        w = enodes[1-enodes.index(v[0])]
        if w in SP:
            if distd[w] > e[1]:
                distd[w] = e[1]
                prevd[w] = v[0]
                H.insert((w,distd[w]))
```

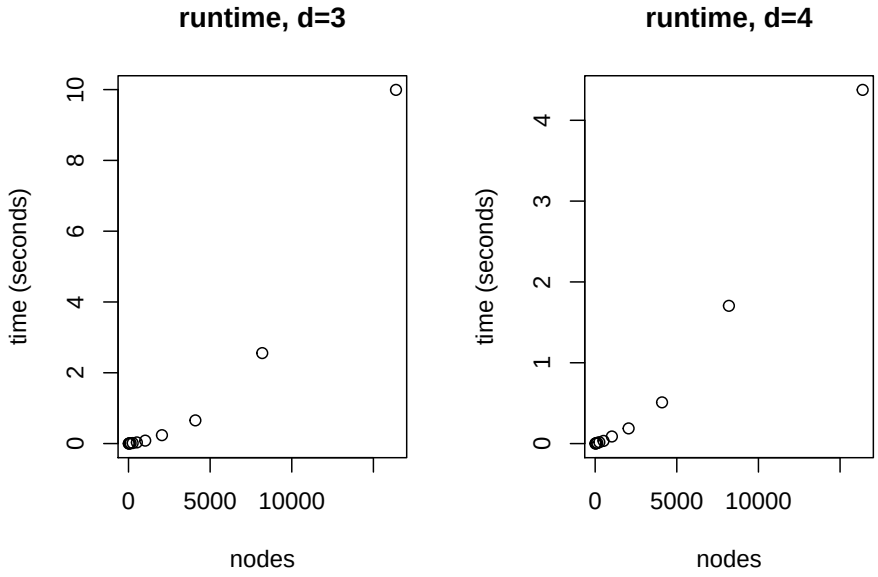
to

```
for e in E[v[0]]:
    if e[0] in SP:
        if distd[e[0]] > e[1]:
            distd[e[0]] = e[1]
            H.insert((e[0],distd[e[0]]))
```

Runtime

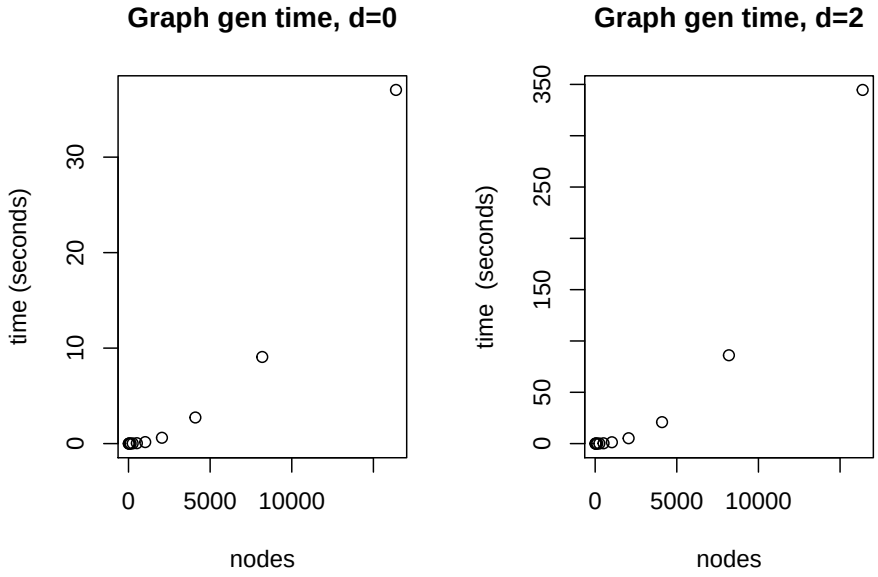
Algorithm

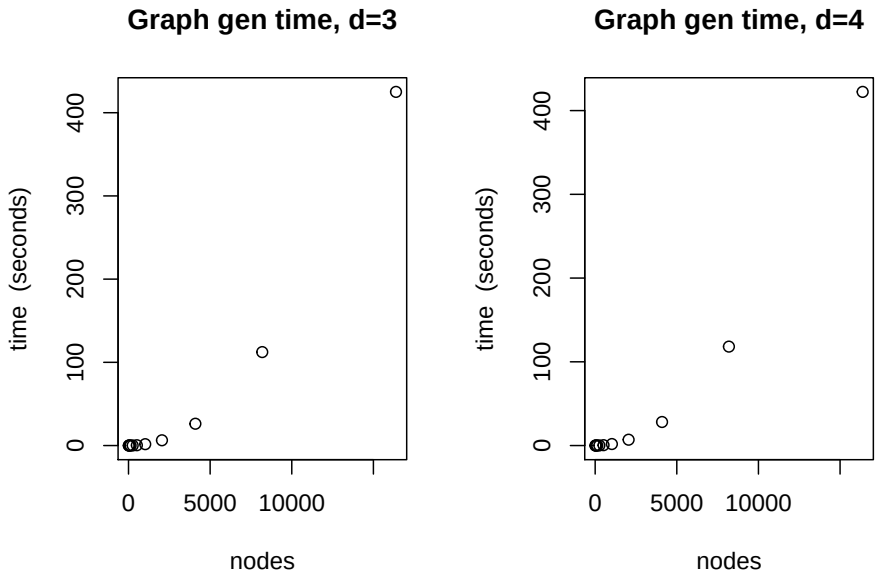




All of the runtime plots are based on 5 trial runs for each value of n. For all choices of d, algorithm runtime exhibits the same functional form with respect to the number of nodes. This is the characteristic O(n log n) curve, which makes sense as, due to our optimization reducing the number of edges submitted to Prim's algorithm, |E| is O(n). The runtime seems to increase with d, as well; this is a result of sub-optimal edge weight thresholds - we are passing a few more edges than we need to to Prim's algorithm. It is just a fluke that d=3 has higher runtime than d=4 - one of the runs here must have resulted in an incomplete MST (due to the threshold being a bit too low for the particular set of edges generated), so we generated a new graph and tried again.

Random graph generation





Most of the total run time for our program comes from the graph generation stage, which exhibits a functional form closer to O(n^2). Graph generation time slows considerably for all d not equal to 0, since for d=0 edge weights can be generated without any calculation. Graph generation time increases a bit from d=2 to d=4, since it is slightly more costly to calculate Euclidean distances in more dimensions. The worse performance of d=3 relative to d=4 is again a result of the edge weight threshold being set too low for one of the d=3 runs.