Jeremy Elkayam

Minimum Spanning Tree Assignment - Writeup

n	d=0	d=2	d=3	d=4
16	1.7053	3.2654	5.1394	6.8868
32	1.8636	4.8798	8.3446	11.6218
64	1.8256	6.9720	13.1967	19.3112
128	1.8072	9.8845	20.5533	31.7244
256	1.8043	14.0147	32.0931	52.3300
512	1.8298	19.8051	50.3370	86.6359
1024	1.8504	28.0322	79.0597	143.8732
2048	1.8587	39.6393	124.4083	239.1005
4096	1.8724	55.9550	195.9091	398.3578
8192	1.8732	79.0474	309.2099	664.4503
16834	1.8692	113.1849	496.985	1131.9582
33668	1.8846	160.8147	788.8199	1898.8110

All of these values are obtained from running 250 trials per n-value per dimension, with the exception of 16834 and 33668, which were obtained from running 25 and 5 per dimension, respectively.

f(x) can be approximated relatively closely using $f(x)=0.8x^{(d-1)/(d)}$, with the exception of $d=0$; where $f(x)=1.9$.

I used Prim's MST algorithm (also seen in CS301) since it performs better than Kruskal's in densely populated cases. Complete graphs (like the ones seen here) are densely populated, so Prim's is the best option here. An adjacency matrix ended up being a more time-efficient solution as compared to the adjacency list, due to its advantage of having constant lookup time.

The algorithm's running time grows with graph size, which is to be expected. For graph sizes lower than n=2048, runtimes are so small on my machine that comparisons are difficult to make. At n=2048, runtimes start to hit roughly 100ms on my machine. Interestingly, runtimes for all four values of d are consistent at this level, but occasionally, the runtime will be roughly 15 milliseconds higher or lower. I have yet to find runtimes for n=2048 that are within more than 2ms of 70ms, 85ms, 100ms, 115ms, etc. I don't know exactly what operation it is that's taking

the JVM 15 milliseconds to perform, but it's clearly having a large impact on the test. Even more interestingly, at higher values of n, d=0 performs far more poorly than all other values of d, coming in at ~10 seconds for n=8192 as opposed to the ~6-8 seconds that d=2,3, and 4 provide. The only explanation I have for this is that the `nextDouble()` method in `Random` is a slower operation than the calculation of the distance formula, which uses array retrieval as well as the built-in `sqrt` and `pow` methods, all of which have constant runtime. I really don't know why d=0 is so much slower, especially considering *f(x)* at d=0 approaches a constant limit.

The Java random number generator is used by creating an instance of the built-in class `Random`, which takes an optional seed in its constructor. Without a constructor, it obtains its own seed. The documentation is pretty opaque as to how the default constructor picks a seed, so I opted to use `System.currentTimeMillis()` as a seed. The RNG is given a new seed when it's constructed and never again in my program (though there is a method in `Random` to change the seed after instantiation, I did not use it). Since my `Random` instance is defined in its declaration as an instance variable of `RandMST`, it gets its seed when the JRE first runs the program and never again. To make calculations easier on myself, I used a PowerShell `For` loop to run `java RandMST` multiple times and calculate the average of 10 runs of the program for every dimension and size, with 25 trials each, for a total of 250 trials, with a new seed being fed into the RNG every 25 trials. The exceptions to this are the values 16834 and 33886 respectively. 16834 averages ~3 seconds per trial on my machine, while 33886 averages 10-15 seconds per trial, making 250 trials a prohibitively long endeavor. In these cases, 25 trials were run for 16834, and 5 were run for 33886, each with a separate instance of `RandMST`. Even through all of this, I still got quite varied results, particularly for small sizes. Running the program with dimension 0 and n=16 gave me results ranging from as low as 1.6 up to nearly 2.0. It's probably due to the small amount of nodes in the graph, but it still intrigued me.

The results seemed to be relatively uniform, but I'm still not sure how much I trust Java's `Random` class and the usage of `currentTimeMillis` as a seed. For one, even when repeating trials with PowerShell, each `Random` instance was still being fed pretty similar seeds simply due to my running trials in such quick succession. When I've had to use random numbers in C, I tend to feed its pseudo-RNG some data from physical aspects of the system, such as heat. This is admittedly still deterministic, as it can be predicted using the physics of the computer's surroundings, it's obscure enough to be difficult to do so. It's been so long since the last time I had to use an RNG in C that I don't remember what data I actually used, and I also don't know how to access that data in Java, which is why I used `currentTimeMillis`. So, I don't totally trust the RNG here, not really because I dislike Java's `Random` class because I feel like I could've done better in providing it a seed. I don't know what the RNG does to make its own seed, but since the documentation provides basically no information on that front, I can't really trust that either. The `currentTimeMillis` seed is hard to trust because, as stated above, it doesn't change very much across successive trials. Thus, it's important to consider that the Java RNG, like any other machine-based RNG, isn't completely trustworthy, as it's deterministic and is therefore really only as random as your seeds are.