Assignment 1

Instructions

To run, navigate to the downloaded directory containing the source code files and run the command `sh make.sh`. This should compile the source files and create an executable jar file. To run this file, run the command `java -jar randmst.jar [vertices] [trials] [dimensions]`.

Methods

The primary objective of this assignment was to understand the output and behavior of developed algorithms on finding minimum spanning trees (MSTs) on complete, randomized, undirected graphs of various sizes. Here, we list some characteristics of our environment and machine:

1. We utilize JRE v13 with the standard libraries.

2. We utilize multithreading on all available cores of the machine. Here, we present results on a 8 core machine.

The algorithm is split into two main parts: one to run the first case (randomized edge weights from [0, 1]) and another to run the other cases (randomized vertices on the unit square, cube, and hypercube). In either case, we utilize modified versions of Prim's algorithm to find the MST since the graph is extremely dense (in which the standard version of Kruskal's algorithm would take over $O(V^2 log(V^2))$ to sort the edges alone). Adjacency matrices and linked list representations of graph edges/connectivity were abandoned very quickly, as it becomes infeasible to store every edge in memory (for 262,144 nodes, assuming each edge takes 32 bits as a float, then we have (262,144)(262,144 - 1)/2 edges requiring roughly 135 GB - a size that no ordinary computer could handle). Instead, we utilize the fact that edges and nodes are randomized in order to harness computational power/speed instead of space restrictions. In each case, however, we still utilize arrays of length V for storing the current minimum edge to every node and markings for if the node has been visited. We utilize lists for searching the minimum, as this provided an ease of implementation and brought the runtime to $O(|V|^2)$. Floats were utilized as the primary data types (rather than doubles), since the numbers generated are small and the precision is high enough to distinguish edges from one another while drastically saving space in memory. The output of the algorithms can be seen in Table 1, where "0D" refers to the first case of uniformly randomized edge weights.

	$f(n)$	$T(n)$
0D	$\frac{1.71n}{1.93+1.39x}$	$(8 \cdot 10^{-9})n^2 - (4 \cdot 10^{-6})n$
2D	$0.99n^{0.28}$	$(7 \cdot 10^{-9})n^2 + (5 \cdot 10^{-5})n$
3D	$1.01n^{0.36}$	$(1 \cdot 10^{-8})n^2 + (2 \cdot 10^{-4})n$
4D	$1.02n^{0.39}$	$(2 \cdot 10^{-8})n^2 + (2 \cdot 10^{-4})n$

Table 1: Asymptotic growth of MST sum and running time vs. n.

n	0D	2D	3D	4D
2	0.338764	0.5278138	0.2278655	0.05622058
4	1.12755	1.0626127	0.15256295	0.20968239
8	1.3830154	1.6528709	0.5416203	1.1530685
16	1.1930493	2.7995706	1.4325746	1.9128752
32	1.4244978	3.3711743	1.6064961	3.7781093
64	1.0892603	4.585989	4.1572404	4.739301
128	1.1881205	5.3431606	4.526414	7.201525
256	1.2204834	6.843225	7.1178246	10.514885
512	1.1842735	7.039305	9.154314	14.727646
1024	1.1878529	6.025258	11.647881	19.08251
2048	1.1934688	7.512478	14.506622	23.173382
4096	1.2128168	9.1531315	18.36922	29.209885
8192	1.1950417	11.606143	23.845863	38.026646
16384	1.2009975	15.080137	31.816586	48.822716
32768	1.1996164	19.116573	43.53785	63.65795
65536	1.1984191	23.456223	57.096397	82.66597
131072	1.1941906	28.664349	72.75606	107.56417
262144	1.1950694	34.33602	90.45366	136.01978

Table 2: Computed values as a function of n, for different randomized graphs.

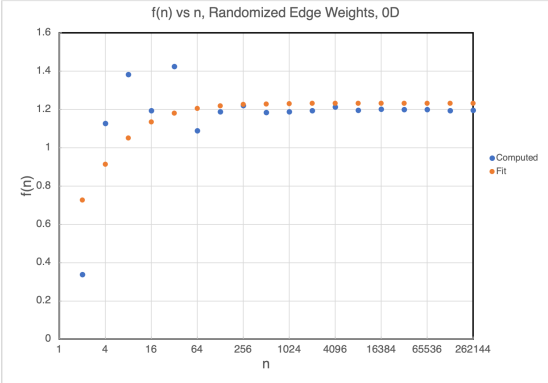
For the first case (0D), the edges are randomized lengths drawn from a uniform distribution of [0, 1]. Instead of computing every edge and storing it in memory until Prim's algorithm reaches that edge for consideration, we only calculate the edge (ie. sample randomly) when we reach that edge. Prim's algorithm examines edges based on neighboring node connectivity and what has been previously examined, and then takes the minimum of that cut; since each edge is examined only once, and since the graph is complete, we only need to sample an edge when we reach it. This eliminates the need to store any edge weights and does not introduce any extra computation other than what is needed, thus reducing the runtime dramatically.

For the second case, we cannot implement the same strategy as the first case. As the vertices themselves have randomized coordinates, the probability distribution of edge weights is no longer uniform and rather quite complex. It is not possible either to re-sample vertices every time we require (thus still saving memory in not storing edges), as this will break Prim's algorithm since the computed edge distances will change iteration to iteration. Instead, we first generate uniformly random coordinates upon initialization of the graph and store it in a list; every time we require the examination of an edge, we compute the Euclidean distance based on these stored values. This is obviously more computationally expensive as the number of dimensions increases; however, when Prim's algorithm must find and possibly update a vertex's neighbors based on its edge lengths, this can be done in parallel. We multithread this in Java based on the maximum number of cores on the current machine and see a large improvement in performance.

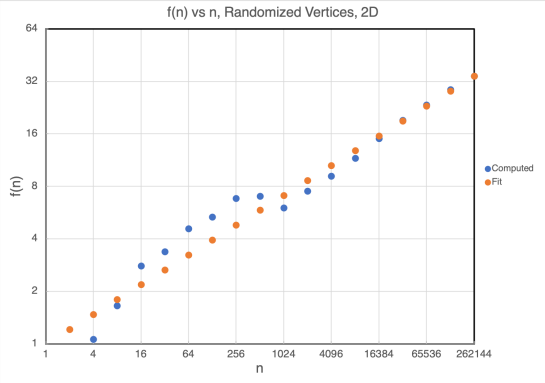
Discussion

For the first case, we see that the growth $f(n)$ appears to be nonlinear with respect to low n, but with increasing powers of n seems to steady at 1.2. The dependence on n seems to eventually disappear. This surprising result can be explained by statistics; as the number of nodes n increases, more and more edges are sampled from a uniform distribution. A higher sampling rate should lead to better recovery of the original distribution (ie. uniform), and thus a greater probability that lower and lower edge weights will appear. Because we choose the lowest edge out of all these edges, then by the cut property we should (generally) expect to keep adding lower and lower edge weights with increasing n. Certainly there may be randomness at low n, as it becomes more probable to draw consecutively large edge weights with fewer draws. But with more and more draws (ie. more edges), we expect to recover the uniform distribution accordingly. This increase is balanced by the increasing number of nodes (and consequently edges) in the MST. The running time of this algorithm for this case is $O(|V|^2$, the same as Prim's algorithm implemented using a simple list for graph representation. This is expected, as the algorithm implementation contains few other computationally expensive operations other than Prim's algorithm. Looking specifically at the coefficients and the lower order terms, we see that with low n the runtime is mainly linear and largely turns quadratic at much higher n.

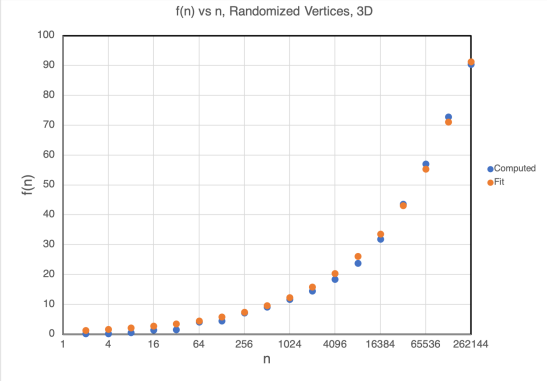
For the second case, we see that for each set of dimensions (2D, 3D, and 4D) the growth $f(n)$ appears to be polynomial. We fit each set of data accordingly to the function An^B using nonlinear curve-fitting optimization in Excel in order to derive coefficients A and B, shown in Figure 1 and Table 1, restricting the curve to start at (0, 0) as we expect no sum



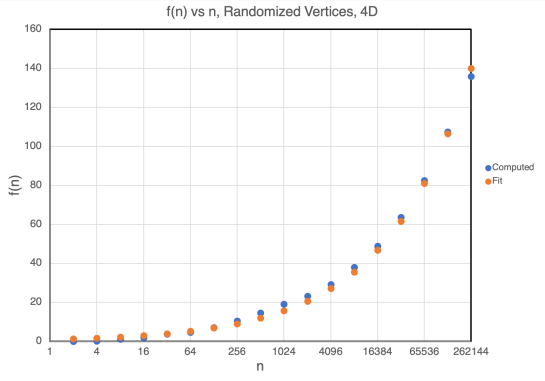
(i) Randomized Edge Weights (0D)



(ii) Randomized Coordinates (2D)



(iii) Randomized Coordinates (3D)



(iv) Randomized Coordinates (4D)

Figure 1: $f(n)$ vs. n for different randomized complete graphs

with no vertices. As seen in Table 1, the coefficient A largely remains the same across all dimensions, and more notably remains essentially one, implying that there is no dependence on scaling in all cases. Rather, we see a difference in the exponent, implying that higher dimensions lead to greater sums for the same number of vertices; this is expected, as the range of possible edge values is greater in higher dimensions. The general trend for this case is likely due to the fact that at higher n, we expect the edge lengths of the MST begin to shrink as random points fill up space. This is balanced, however, by the increasing number of vertices/edges that make up the MST with increasing n. Since the edge weights are not distributed uniformly as with the first case, we do not see the same behavior and see instead this fractional exponent behavior. In terms of the runtime, we see very similar asymptotic runtime with the first case; however, we see that the coefficients and lower order terms are higher, as expected, since Euclidean distance computations should increase the runtime per edge by a constant amount proportional to the dimension.

Some other less method-critical observations involved the Java libraries and hardware. In particular, the Random class provided for by the standard libraries produced different output depending on the seeding method. Seeding using `System.currentTimeMillis()` versus using `System.nanoTime()` produced drastically different results in terms of growth

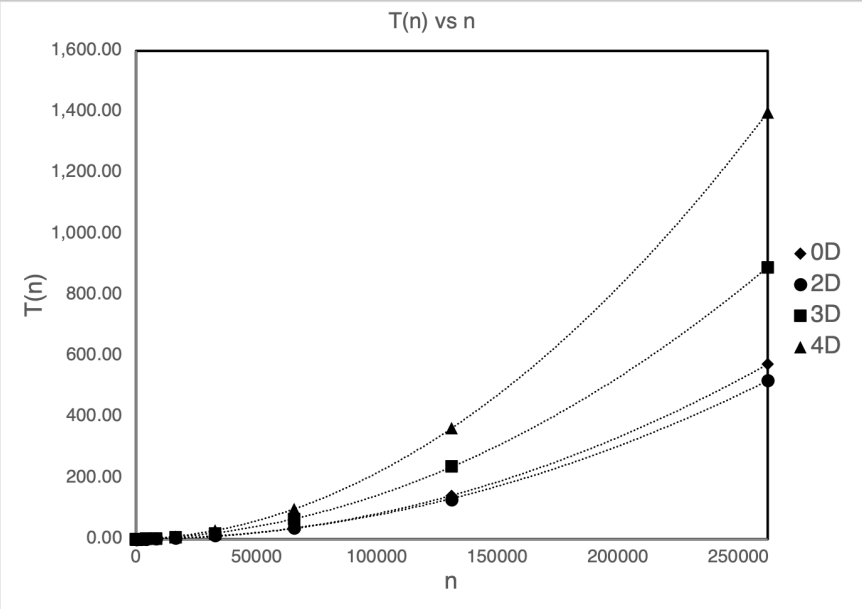


Figure 2: $T(n)$ vs. n for different randomized complete graphs

rate and variance in the data. This is because the millisecond clock time did not update fast enough compared to the runtime of the algorithm; as the algorithm ran much faster each iteration a RNG was produced, edges were seeded with the same value and hence had duplicate values, thus leading to highly biased and incorrect results. Using nanoTime, the default seeding methodology of Java and is what is implemeneted, proved much more conclusive of results.

We also noticed that, per the Java SDK API, the Random class provides a random number "0.0f (inclusive) to 1.0f (exclusive)". It is thus impossible to generate the number 1; however, considering that these are continuous random distributions of edge length and vertex dimension, the probability of obtaining 1 is essentially zero regardless.

References

[1] Java 13 Docs, Oracle.

[2] Computer Science 124, Harvard University.