Table 1: Results for Values of n With $numtrials = 5, dimension = 4$

n	average $f(n)$	total time (seconds)
128	28.627	0.0165
256	46.509	0.075
512	76.25	0.300
1024	130.42	1.259
2048	216.420	5.378
4096	360.030	23.28
8192	602.38	97.43
16384	1009.21	403.647
32768	1686.93	1512.621

Table 2: Results for Values of n With $numtrials = 5, dimension = 0$

n	average $f(n)$	total time (seconds)
128	1.175	0.0099
256	1.269	0.0417
512	1.288	0.182
1024	1.279	0.862
2048	1.221	3.626
4096	1.201	15.69
8192	1.200	68.305

Estimated $f(n)$: $f(n) = 1.24$ (constant).

Table 3: Results for Values of n With $numtrials = 5, dimension = 2$

n	average $f(n)$	total time (seconds)
128	7.209	0.0146
256	10.637	0.0564
512	15.090	0.248
1024	20.890	1.040
2048	29.718	4.493
4096	41.84	19.319
8192	59.025	81.468

Table 4: Results for Values of n With $numtrials = 5, dimension = 3$

n	average $f(n)$	total time (seconds)
128	17.598	0.0149
256	27.813	0.0634
512	43.433	0.266
1024	67.536	1.140
2048	107.44	4.941
4096	168.67	20.70
8192	266.96	88.464

Our implementation used Kruskal's algorithm. We chose this over Prim's algorithm because it only requires tracking a list of all the edges in the tree as candidates, rather than updating the list of candidates on each iteration, which we found easier to implement. We also learned some optimizing tricks for Kruskal's algorithm in class, such as path compression and union by rank, both of which were implemented in our code.

Times for each run are listed in the tables above. Each doubling of n corresponded to approximately a 4x increase in time. This makes sense because the number of edges that must be generated, sorted, and checked is quadratic in n, so it is reasonable for the time to increase quadratically as well. Times are slightly lower for smaller dimensions, as the calculation of Euclidean distance has fewer terms in smaller dimensions, and although this function runs in linear time in n it is called a quadratic number of times, so even a slight increase is noticeable. In the case of dimension = 0 it is never called (weights are random uniform between 0 and 1), so this version is noticeably faster.

The most surprising $f(n)$ was for dimension = 0, where we found that the weight was essentially constant. This is because, since the edge weights are all independent, increasing the number of points also increased the number of very small edge weights. Since number of edges in the graph grows faster than number of edges needed for an MST (the former is quadratic, while the latter is linear), while each larger MST contains more edges, it also contains a lower average edge cost. It seems that these two factors essentially cancel out, and leave a constant MST weight. The rest of the functions increase with n, with higher dimensions having higher rates of increase as well as higher base weights. This made more sense intuitively.

At no point did we find the random number generator to create any aberrant behavior. We seeded with the current time on each iteration, so while the numbers were generated pseudorandomly, the sequence was different each time.