Problem Set 2 Report[1]

Billy Cox[2]
CS125
9/19/14

Implementation

I chose to implement Kruskal's Algorithm. My original implementation contained classes implemented as follows:

- Edge: a class to represent edges in the graph. This class encapsulates a float value for the weight of the edge, as well as two integer values to represent the endpoints of the edge in the graph. One edge object is compared to another simply by comparing their weights, to allow for sorting later on.
- DisjointSet: a class to represent a collection of disjoint sets, which supports the union and find operations. This is implemented as an array, where the elements (1 through n) correspond to the indices, and the actual array slot at a given index denotes the parent of that element.
- Graph: contains an array of Edge objects of length n (n – 1) / 2, which is the total number of edges in the graph. The class constructor creates a random complete graph on n vertices with a given dimension (an auxiliary array is used to represented Euclidean points for dimensions 2, 3, and 4). The class also had a method to perform Kruskal's Algorithm, which began with Java's built in O(n log n) sort.
- Main: a main class to actually build graphs and find MST's.

Optimization

1 The accompanying code can be run with ./randmst. Joy said that this format was acceptable.
2 I collaborated and discussed with Cameron Clarke, but we each completed the problem set separately.

After verifying the correctness of the algorithm on small inputs by doing Kruskal's Algorithm by hand, I realized that my program would never terminate on inputs larger than about 10,000. I inserted a few lines to actually time various portions of the code, and determined that the bottleneck was building and sorting the graph; the rest of Kruskal's Algorithm was taking negligible time.

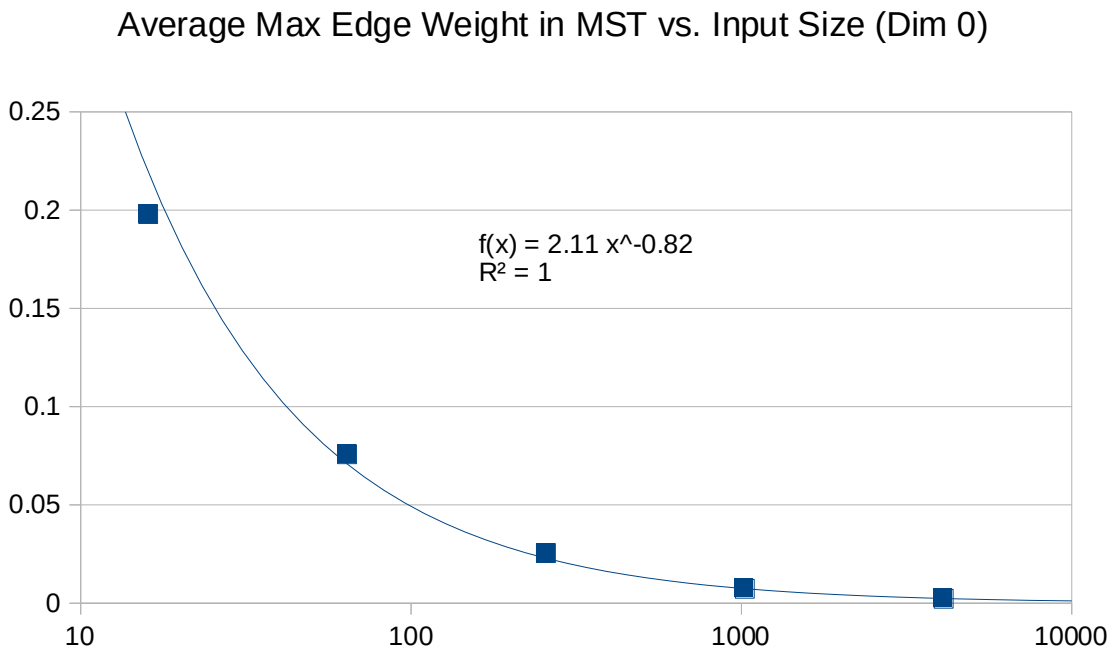
My first idea was to implement a faster sort, rather than use Java's built in sort. I thought of Bucket Sort, which would have worked well for dimension zero, given the evenly distributed inputs. I realized, however, that BucketSort would present difficulties with higher dimensions, because the distance between two random Euclidean points is not evenly distributed over the interval of possible values it can obtain. For example, there do exist pairs of points almost sqrt(3) distance apart within the unit cube, but two random points are not very likely to be this far apart, since they would have to be situated in the very corners of the cube.

I then considered implementing a min heap to represent the edges, rather than simply an array. The problem with the min heap, however, is that actually removing the min from the heap requires O(log n) time; this is much more costly than simply traversing a sorted array. On top of O(log n) insertion into the heap, building the heap could also be somewhat costly. Though this would likely improve on the O(n log n) time needed to sort the array, I feared that the difference would not be large enough to allow the program to run on my system for large n.

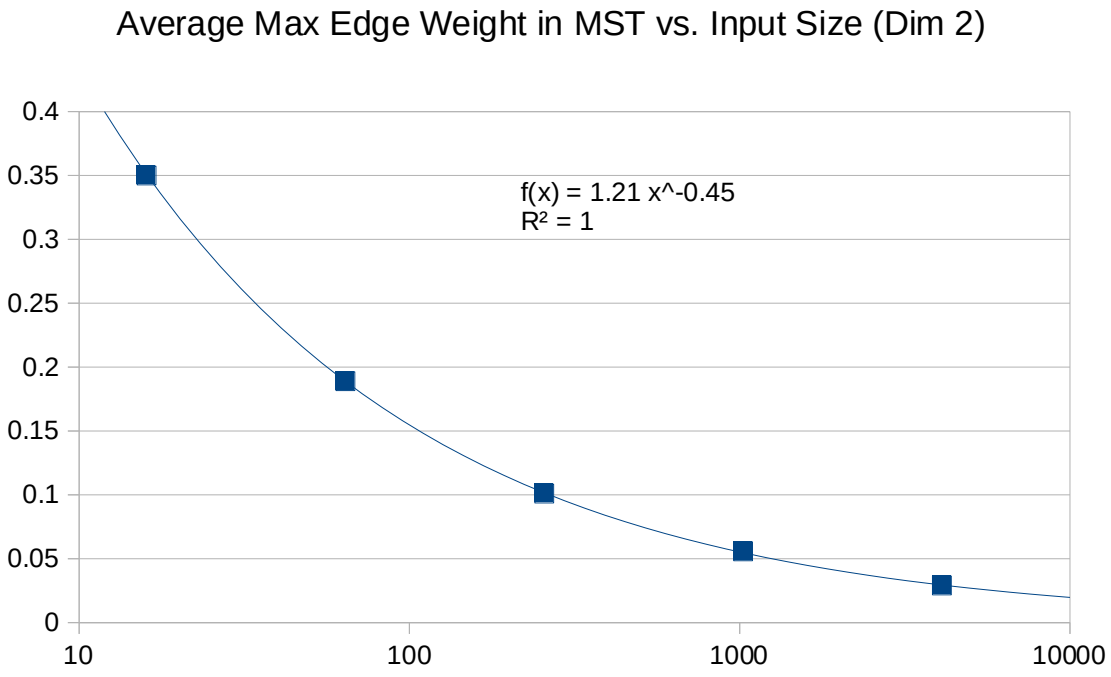
I then realized that the best way to alleviate some of the strain from sorting was to reduce the size of the input. I set out to determine the largest possible edge that could end up in the MST for various inputs. I had my program print the full list of edge weights in the MST, so that I could see the heaviest edge that was included at various n. I averaged ten trials for each data point, and then found a best fit curve for the average heaviest edge at each input size. A more

accurate best fit curve a*x^b is given below each graph:

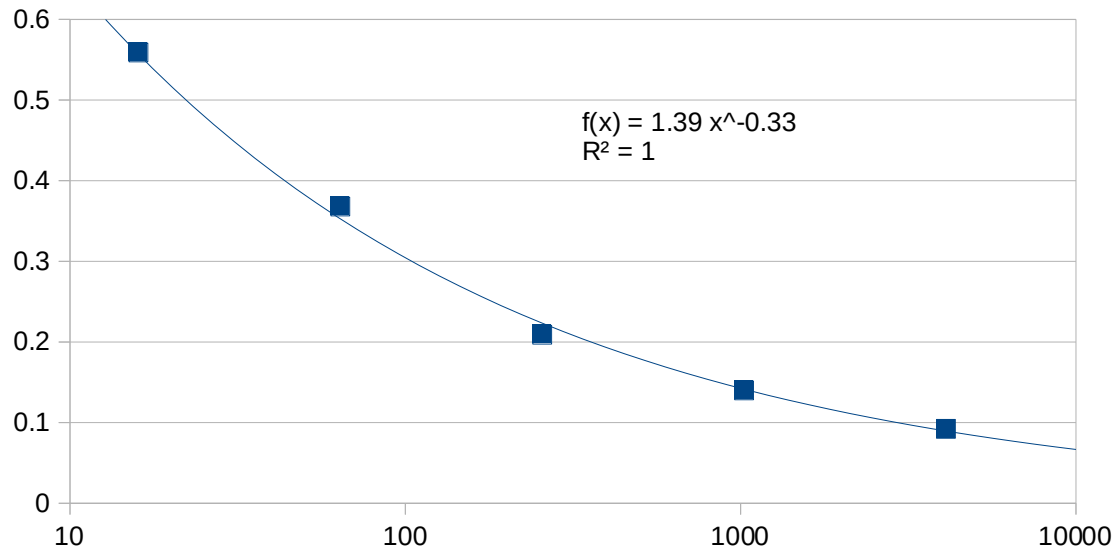
Input size	*Average max edge weight (dim 0)*	*Average max edge weight (dim 2)*	*Average max edge weight (dim 3)*	*Average max edge weight (dim 4)*
16	0.198	0.350	0.559	0.582
64	0.0758	0.189	0.368	0.440
256	0.0255	0.101	0.209	0.334
1024	0.00733	0.0558	0.140	0.253
4096	0.00223	0.0291	0.0923	0.179



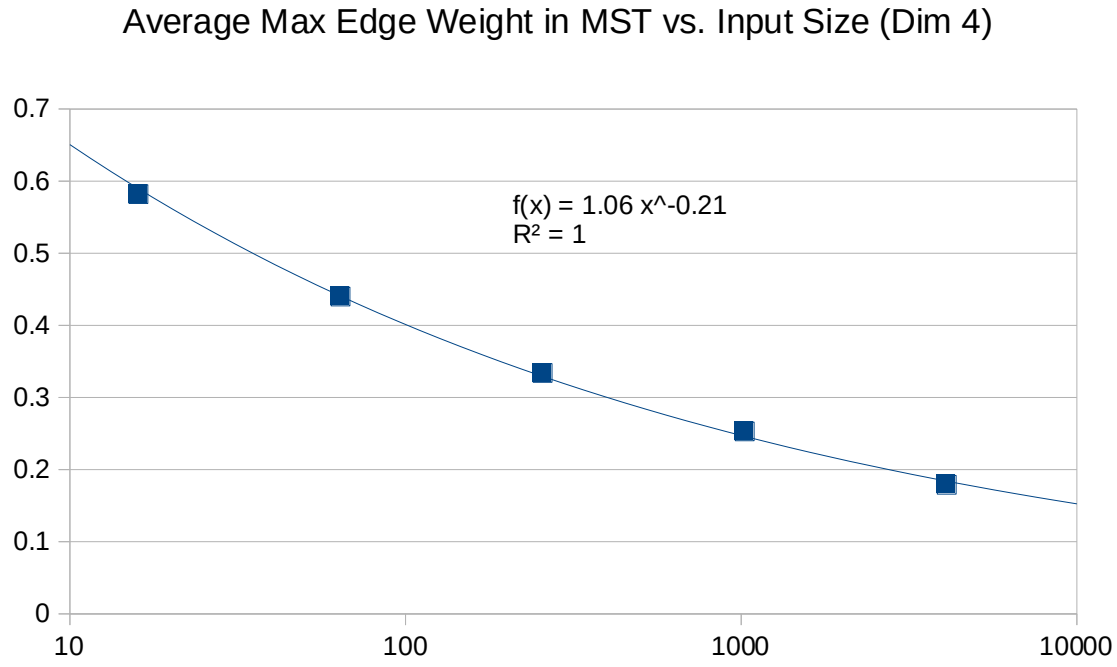
f(x) = 2.1075 x ^-0.81575, R^2 = 0.9975



f(x) = 1.2110 x^-.4468, R^2 = 0.9998



f(x) = 1.3887 x^-0.3296, R^2 = 0.9965



f(x) = 1.0554 x^-0.2100, R^2 = 0.9978

I included the constants for these functions in my code, to determine whether edges were likely to be part of the MST. Since these were only averages, I allowed the user to define a tolerance factor c. In this way, all the edges with weight greater than the c * f(n) for input size n would be discarded. Rather than combing through the edges after they were all generated to discard weights, I simply never added the edges that were too heavy to the graph, in order to save time and space building the graph as well as sorting it. I elected to use Java's built in ArrayList class, which allows an arbitrary amount of elements to be added[3]. The tolerance factor is the first command line input. If the value is zero, the program selects a default value that works nearly all the time. The user may choose, however, to enter a tolerance factor very close to 1 in order to

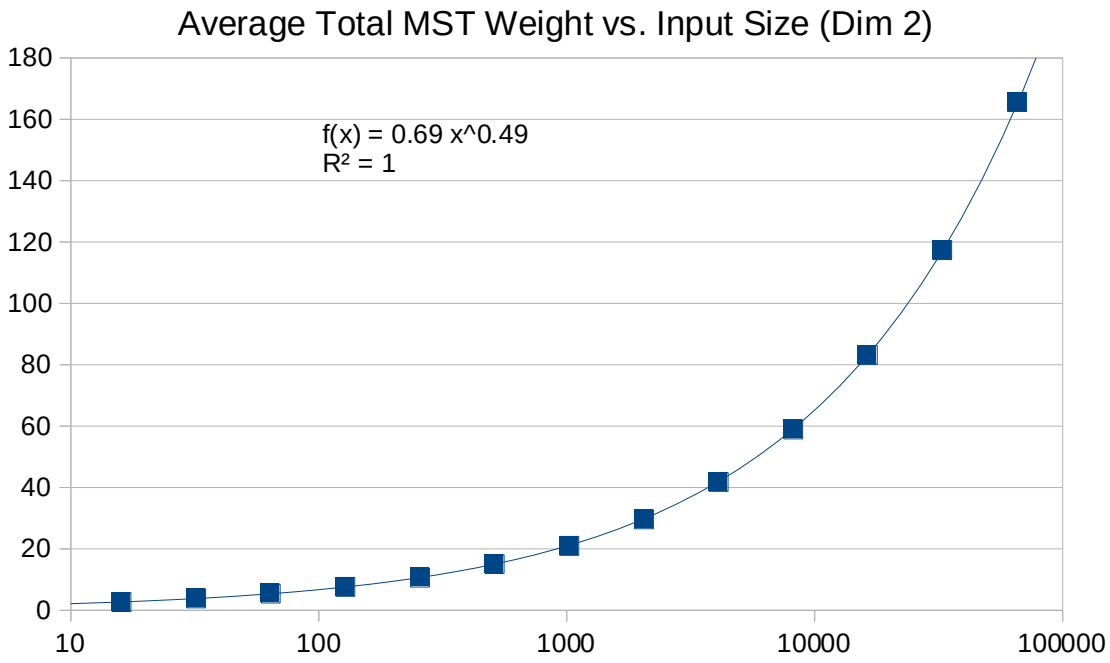
3 Under the hood, there is some cost associated with reallocating and copying the array as it expands to accommodate more values, but I found these extra operations not to be prohibitive in the runtime of my implementation.

speed up the program (more edges will be discarded). As n gets very large, the largest weight in the MST gets more and more likely to be close to the average, which allows for smaller tolerance factors. If the user enters a tolerance factor that is too small (the graph ends up disconnected), the program will quit and inform the user that a larger tolerance is needed.

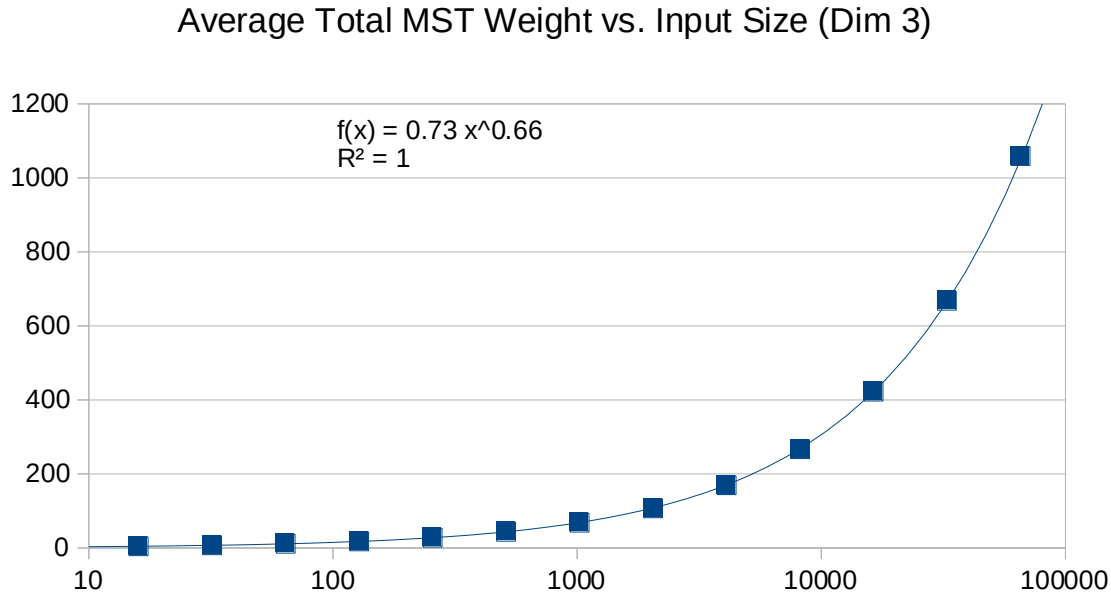
Results

After optimizing per the above, I was able to run the program on my system and determine the average total weight of the MST for various input sizes. Each of the following data points is the result of 20 trials averaged together. Graphs are included, along with best fit functions a * x^b, for dimensions 2, 3, and 4:

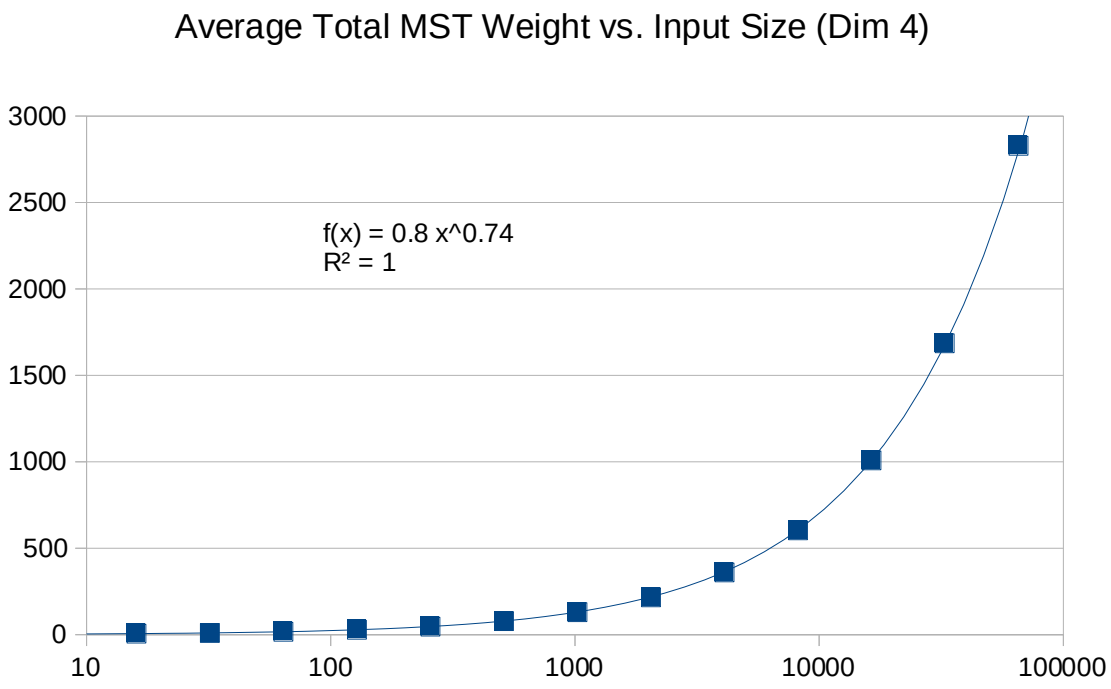
Input size	*Average MST weight (dim 0)*	*Average MST weight (dim 2)*	*Average MST weight (dim 3)*	*Average MST weight (dim 4)*
16	1.1792	2.6739	4.5018	6.2036
32	1.1797	3.8741	7.0626	10.351
64	1.1450	5.4534	11.289	17.214
128	1.1776	7.6203	17.582	28.356
256	1.1838	10.659	27.710	46.991
512	1.1999	15.040	43.453	78.411
1024	1.1963	20.983	68.138	130.28
2048	1.1957	29.646	106.96	216.40
4096	1.1988	41.831	169.35	360.61
8192	1.2009	58.918	266.82	603.96
16384	1.1985	83.203	422.39	1008.6
32678	1.1998	117.37	667.73	1685.6
65536	1.1989	165.62	1058.9	2828.2



f(x) = 0.69155 x^0.49359, R^2 = 1.0000



f(x) = 0.73013 x^0.65544, R^2 = 1.0000



f(x) = 0.80274 x^0.73527, R^2 = 1.0000

Discussion

For the zero-dimensional case, the values seem to approach a limit of about 1.2. This makes sense, as there is no limit to how small the weights between any two edges can be; as n gets bigger, the probability of finding extremely light edges increases, resulting in the overall weight of the graph remaining relatively constant. For the other cases, it makes sense that the weight would grow with n. Even though a very large number of points in k-dimensional space will be very close together, connecting them all would still require a lot of distance, because they are actual points in space.