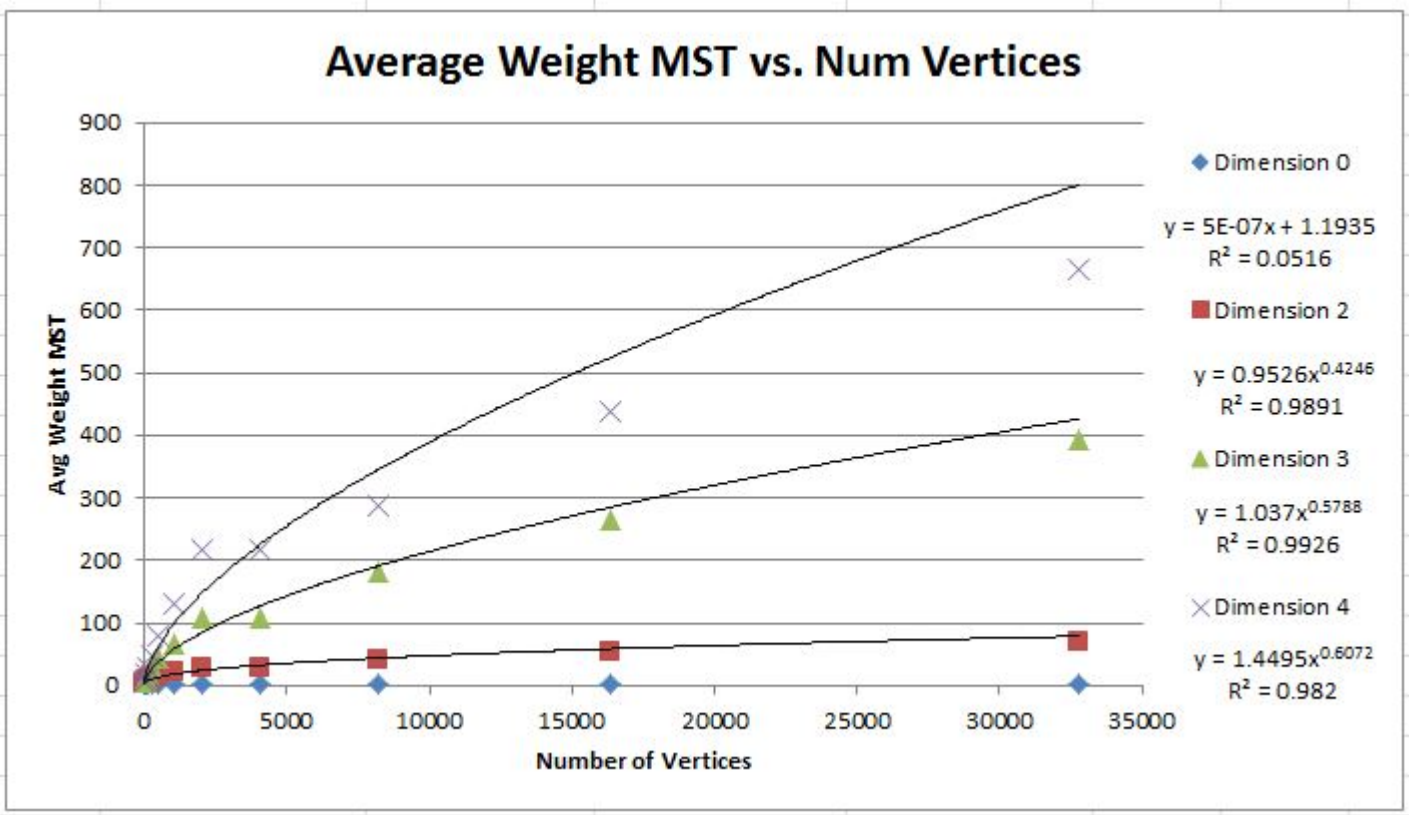
CS124 Programming Assignment 1

Joy Ming and Alisa Nguyen (05 March 2013)

1 Data

Our data for the average tree size for several values of n = number of vertices is displayed in the table and graph below.

Number Vertices	NumTrials	Time (seconds)	0D	2D	3D	4D
16	500	0.598	1.13281423687696	2.68472276916808	4.47010330889035	6.10597287177147
32	500	1.212	1.19362482185003	3.87119711153033	7.18089445267967	10.3090810192296
64	500	3.696	1.19450807445836	5.44843526437119	11.2816375924156	17.1627429156429
128	500	27.957	1.20281889636816	7.63758115930184	17.612546120779	28.5210095573885
256	500	192.254	1.2071159896534	10.7230727663911	27.6005369240705	47.2446845590557
512	5	15.858	1.19684253777017	14.9883540319422	43.3632622367307	78.2080075672136
1024	5	46.656	1.20376858021556	21.1269702396643	67.7807466839505	130.566558745654
2048	5		1.19740357175084	29.7746974697011	107.1396625244	216.843462438744
4096	5		1.21284985294384	29.687314704378	106.674479456663	217.307413415302
8192	5		1.20750737236216	42.1367989945	183.283565684	288.7198502275
16384	5		1.19938723934553	53.403176554987	264.786496451	439.1326492327
32768	5		1.20476802039292	71.05447729432	392.841067392	665.660832854



2 Our guess for function $f(n)$

The growth rates for all the dimensions exept for 0D seem to be polynomial where the power is less than 1. More specific estimates of the growth rates we found are displayed on the graph and table below.

Dimension	$f(n)$
0D	$0.00007x + 1.1935$
1D	$0.9526x^{0.4246}$
3D	$1.037x^{0.5788}$
4D	$1.4495x^{0.6072}$

For Dimension 0, we actually found an almost linear growth rate in which the average weight of the MST remains around 1.2 no matter how large n grows. On the other hand, the other dimensions all grew with a power less than 1. However, our estimates for the growth rates do not look like they fit the points very well, and it seems as if our data beyond 8192 are skewed a little lower than they are supposed to for some reason. Although we tried to debug and many more trials on the large amounts of n, due to time constraints, we were unable to explore this phenomenon further and fix any possible errors related to these specific instances. Given more time and energy, we would return to figure out exactly what was going on for large numbers of n.

3 Discussion of Experiments

1. Prim's Algorithm

   We chose to implement Prim's Algorithm because we wanted to practice implementing a minimum heap using the material we covered in leture and section. Actually implementing the heap in Java gave us a lot of practice with coding and debugging in the language, since neither of us were too familiar with it to begin with. In terms of performance, we also believed that Prim's would work better in this situation because we are working with complete graphs with all possible edges, and since Prim's uses a priority queue instead of the simpler data structure used in Kruskal's to sort the edges, it would theoretically be faster.

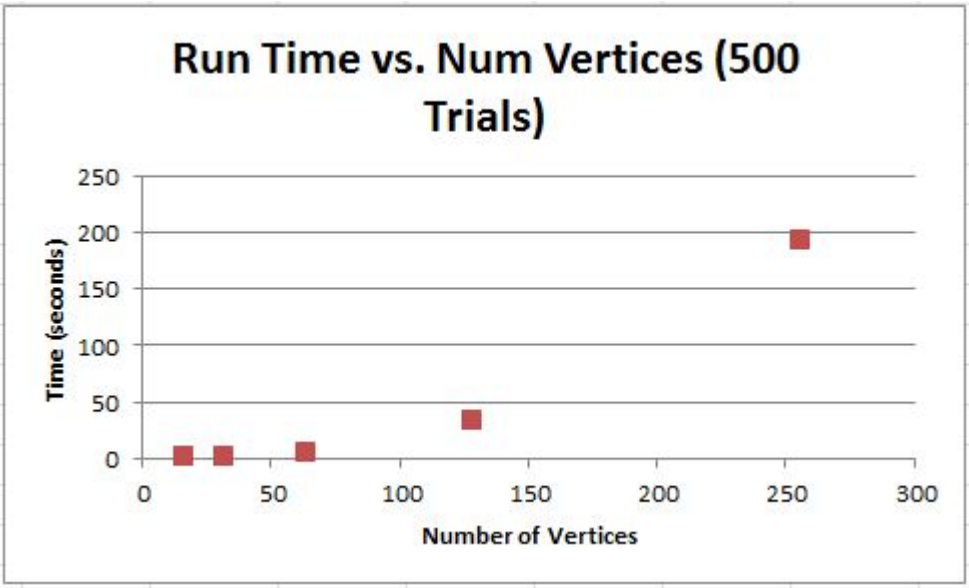
2. Growth Rates

   Initially, our data for the 0 dimension exhibiited a similar growth rate to the other dimensions, which we at first naively generalized as correct. However, upon further inspection of our implementation of Heap, we found that it was not in fact behaving as originally planned. So, after debugging and modifying our Heap, we finally received accurate growth rates for each of the dimensions, which are discussed more in depth in section 2. We were surprised that the growth rate for the 0 dimension graphs were so different from the other dimensions, but after discussing the way in which the Java Random library generates streams of random numbers, we realized that it made sense that the Gaussian distribution of the random weights would not cause the weight of the whole MST to increase by more than a constant factor over time.
   The growth rates seem to increase for higher dimensions, which is somewhat what we expected because there edge weights and the sizes of the MSTs also increase with dimension. However, because our data for large n seem a bit skewed, we are unsure of the actual explanation for this.

3. Algorithm Run Time

   The run time for our algorithm without pruning and simplifying graphs is charted below for the vertices in which we ran 500 trials. For number of vertices beyond 256, we ran 5 trials and the run times are listed in the table below. It makes sense that the run time grows almost exponentially because the number of edges grows exponentially without pruning as a function of n. Before simplifying the graph, running our algorithm on sizes of $n \geq 512$ would increase the CPU usage and slow down our computer's performance signficantly. The first times we tested our algorithm the program

would run for upwards of 15 minutes before stopping or before burning out our computer and forcing us to restart.



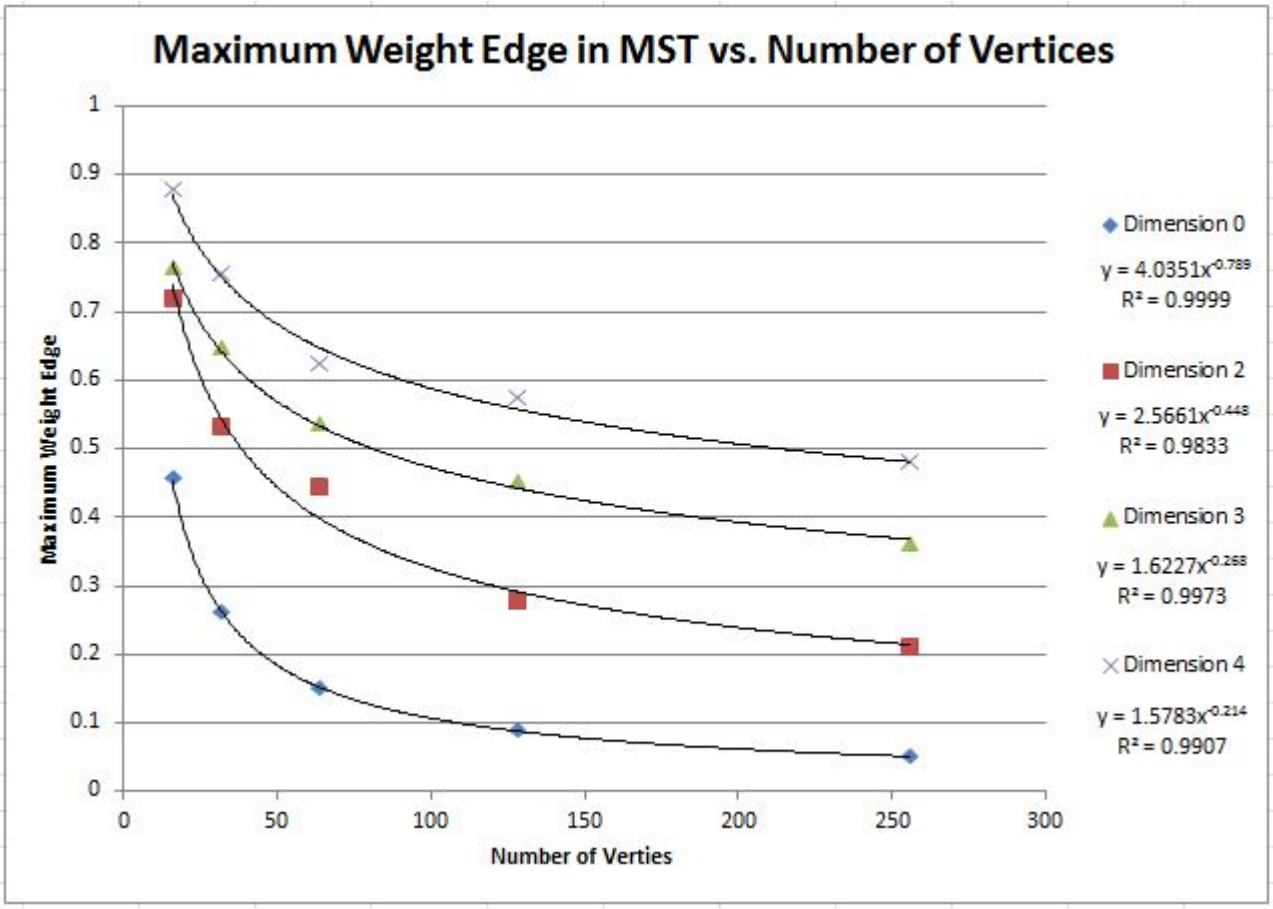
4. Random Number Generator

   We used the random() function from Java's math library, which returns a long from 0 to 1 seeded by current system time. We didn't run into any interesting experiences using this method.

5. Simplifying our Graph for large n

   We ran into quite a few issues dealing with large n and tried many approaches throw away certain edges generated in the tree based on trends we saw in the data.

   - First we tried throwing away edges that are heavier than the expected weight of a single edge in the graph we are considering simply based on a threshold calculation that was a function of only the dimensions of the graph. We had a cutoffs that were based on the hypercube line picking formula.
   - Then we realized that the cutoff should be determined not only based on the dimensions of the graph but also the number of vertices. We calculate expected weight by graphing datapoints of the longest edge in MSTs of each size n for each of the dimensions, over many trials (500), and then finding a function $k(n)$ that best fit these data points, depending on the parameters n and dimension d. The functions we came up with for each of the dimensions are shown on the graph below.



To justify the correctness of these methods, because we are using complete graphs, there is a low probability that any edge higher than the expected weight will be chosen for the MST of that graph, as proven via trial and error. The table below shows our algorithm's performance with and without pruning for small numbers of n. There is no significant difference in performance.

Numpoints	Time (seconds)	0D	2D	3D	4D
16	0.598	1.132814	2.684723	4.470103	6.105973
16 - no pruning	0.624	1.155288	2.731821	4.506055	6.137092
32	1.212	1.193625	3.871197	7.180894	10.30908
32 - no pruning	1.439	1.191222	3.879326	7.108791	10.31346
64	3.696	1.194508	5.448435	11.28164	17.16274
64 - no pruning	5.62	1.203524	5.465693	11.25835	17.26362
128	27.957	1.202819	7.637581	17.61255	28.52101
128 - no pruning	33.682	1.20891	7.65514	17.65306	28.49294

6. Handling large values of n

One aspect of this assignment we had the most trouble with was modifying our algorithm to be able to handle large values of n in a short period of time. Although we tried many different methods to cut down on the running time of our algorithm and the space needed for large values of n, including limiting the amount of edges for each vertex to the minimum 30 because it is most likely

that the edges for the MST will come from there, using a hash table to store our vertices and using an ArrayList of ArrayLists of Vertices, we were still unable to improve our running time by very much. Given more time and energy, we would continue to work through our method for generating the graphs to continue to remove unecessary pieces of information. From this assignment, we learned that space and time constraints are actually important and that we need to take them into consideration early on in the coding process instead of at the very end.