Programming assignment 1 - CS 124
Kruskal's algorithm on randomly created graphs

Anam Javed, Lora Stoyanova

February 2017

1 Introduction

2 Methodology

1. Kruskal vs. Prim
   Our algorithm of choice was Kruskal's, because, unlike Prim's it uses simpler data structures (arrays, linked lists, etc as opposed to Fibonnaci heaps), which makes it easier to implement and debug, and it takes up less memory space (linear space).
   Both of us know Java, C and Python. We did not use Python because of TF's recommendation to not use a scripting language. We chose Java over C for a few reasons:

   (a) Java has its own garbage collector unlike C
   (b) Java is higher-level than C, meaning we won't have to worry about memory allocation, and we still have mechanisms of managing memory, for example by modifying our data structures.
   (c) We just had a better experience debugging Java

2. Optimizations of our code to make up for the time complexity difference between Kruskal's and Prim's

   (a) We realized early on that keeping different classes in separate files is more helpful for easier coding and optimization. Taking advantage of Java's object-orientation made our code much more readable and organized.
   (b) When creating the graph, we managed its size by not creating double edges. Our unoptimized algorithm created $n - 1$ edges for every vertex, where n is the number of vertices, thus creating the same edge multiple times (because our graph is undirected, the edge (v_i, v_j) is

the same as the edge (v_j, v_i)). We removed the redundant edges by only creating edges that are not yet in the graph, thus reducing the number of edges from $n(n-1)$ to $\frac{n(n-1)}{2}$, which is the maximum number of edges we need.

(c) At the end of each iteration, our code deletes the contents of the data structures it has used so far, hereby freeing space it is no longer using for the next iteration and saving time for the initialization of data structures (however small this time is for an array - O(1) performed n times is O(n)).

(d) Pruning
We calculated the average edge weight maximum in a number of final MSTs, then fit it to a curve and scaled the curve to ensure that the curve is an effective upper bound of the statistical maximum edge of the MSTs. Then, we used this function to "prune" the graph (aka remove edges with very low chance of being a part of the MST from the graph while it is being created). Instead of adding edges and then removing them, which would have been very time and space consuming (an O(1) insertion and deletion time per edge turns into an O(n), when done for n edges n*1 = n. Also, we would have needed to literally create an arraylist of n elements before deleting some of them, which would have defeated the purpose of pruning for the sake of space savings), we instead checked the edge weight before adding the edge to the graph and if the edge weight was above f(n), we simply did not add it to the graph.

3. Calculating f(n) and pruning

(a) Calculating f(n)

i. f(n) is a function for the maximum edge included in the Minimum Spanning Tree produced by our implementation of Kruskal's algorithm

ii. We plugged in different numbers of edges for 4 different dimensions and 6 different values in groups of 5 trials per value per dimension (around 120 trials) and took the maximum edges from each group of 5 trials and plotted them to fit them to a curve (f(n)) that would be their tight bound.

iii. We then multiplied f(n) by 1.1 to add an additional 10 percent to make sure that even values within a few standard deviations are taken into account.

(b) Proof of correctness: pruning
Pruning basically means only adding edges to the graph only if their weights are less than the average greatest weight of an edge of to the MST.
Proof of correctness: The proof is probabilistic - we did tens of iterations and took the highest edge weight we would get for each dimension and number of vertices in order to maximize the edge weight.

We then plotted the values we got and assigned a curve to them, which we will refer to as the tight bound. We then multiplied the tight bound by a constant - 1.1, thus turning it into a strict upper bound for the maximum edge weights by size. Then, we took the strict upper bound and used it as a lower bound for the edges that we will exclude (aka an upper bound for the edge weights we included in the pruned graph). This works because statistically, the chance that an edge above the upper bound of $cf(n)$ is included in the MST is very close to zero, because

4. MST size
   We calculated the average MST size for each dimension and for each number of nodes in groups of 5 trials, then fitted the results to a curve.

3 Results

Result tables are in the appendix - results for MST size and average maximum edge weight in MST

4 Conclusions

1. General
   (a) We found out that memory was the greatest issue with our system, which was a surprise for us because we had expected that our greatest problem would be the time complexity of our overall algorithm. However, we found out that the heapsize we had access to was very small (4096 MB), hence we had to prune to get our code to work quickly. Our overall algorithm runs in time O(v^2d), where v is the number of vertices and d - the number of dimensions, and the space it takes to do so is close to linear. We were considering using a heap implementation of Kruskal's instead of the Arraylist, however, we realized that our runtime was not a problem - before 4096, our algorithm took miliseconds per iteration, and a few minutes for greater values. However, a heap would not have managed space more efficiently. We ran our code on a university computer to get to the higher values.
   (b) We noticed that our random number generator tended towards the bigger side of its bounds as our number of vertices, trials and dimensions increased, which we then researched and found out that Math.rand tends to be slightly biased towards its higher bound of 1. However, with the number of trials we have used and the fitting of the graph tightly, this did not matter as much, since we needed an upper bound anyway. Were we to be looking for a lower bound, we

could have avoided the biasness by excluding the highest values we got from our calculations.

2. MST size
   As expected the average weight of the MST grew with the increase in number of vertices and dimensions.

3. Average Maximum edge weight
   The maximum edge weights for different graph sizes became smaller and smaller with increasing n. This is understandable, because with large number of vertices, a large number of edges are produced with very small weights and these are added to the MST.

Figure 1: Average Weight of MST Over 5 Trials Before Optimization

Average Weight of MST Over 5 Runtrials Before Optimization						
dim/n	128	256	512	1024	2048	4096
0	1.099216637	1.110146722	1.225628141	1.205023543	1.213903255	1.205671086
2	7.690817022	10.73609478	14.89433789	21.01008571	29.77345266	41.8574527
3	17.63773092	27.82181385	43.10894041	68.13384895	107.205818	169.4922495
4	28.45372646	47.44939318	78.33491422	130.5905898	215.9030419	361.351425

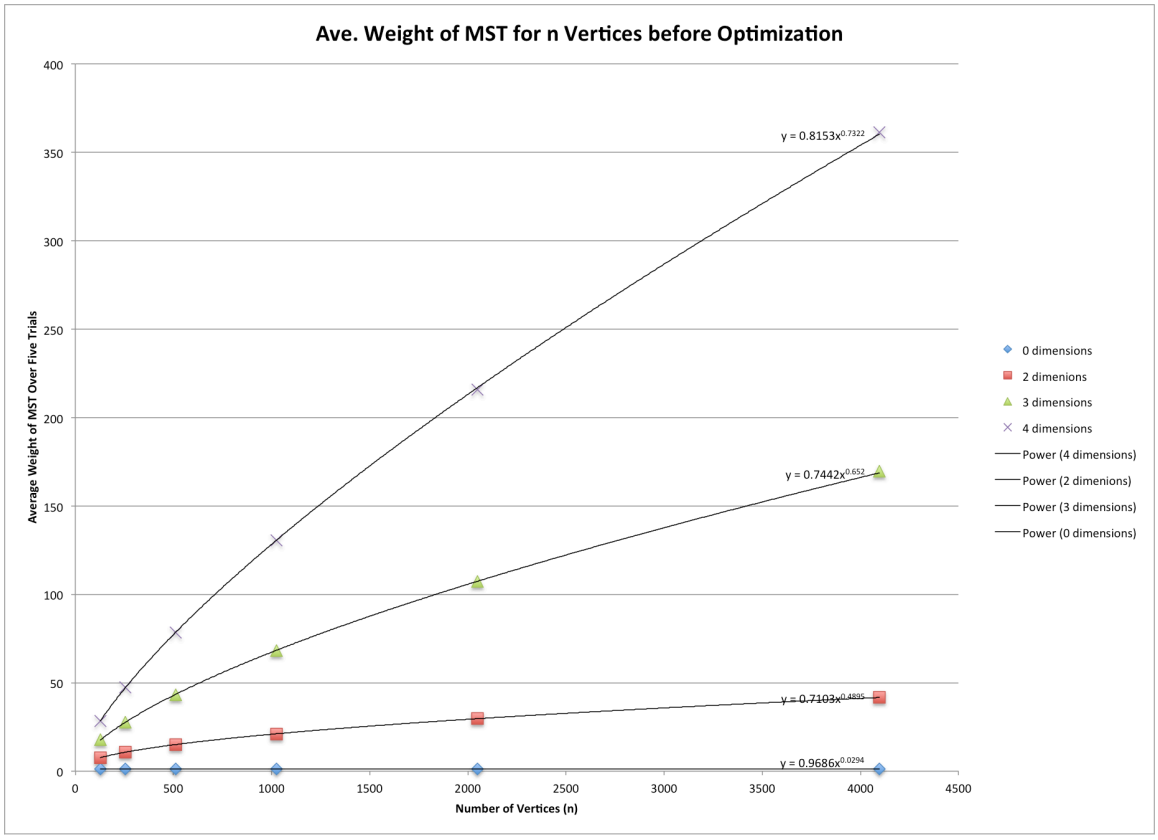

Figure 2: Pruning Function Showing Maximum Edge in MST Over Five Trials

Dimension/Vertices	128	256	512	1024	2048	4096
0	0.066699717	0.033231093	0.014829299	0.009642732	0.004842656	0.002240403
2	0.174728504	0.117953497	0.079082711	0.065073961	0.047525985	0.030294875
3	0.287002727	0.222751257	0.195764656	0.15138177	0.120963573	0.097705378
4	0.525784232	0.423688606	0.308356352	0.271016961	0.234704896	0.190234378

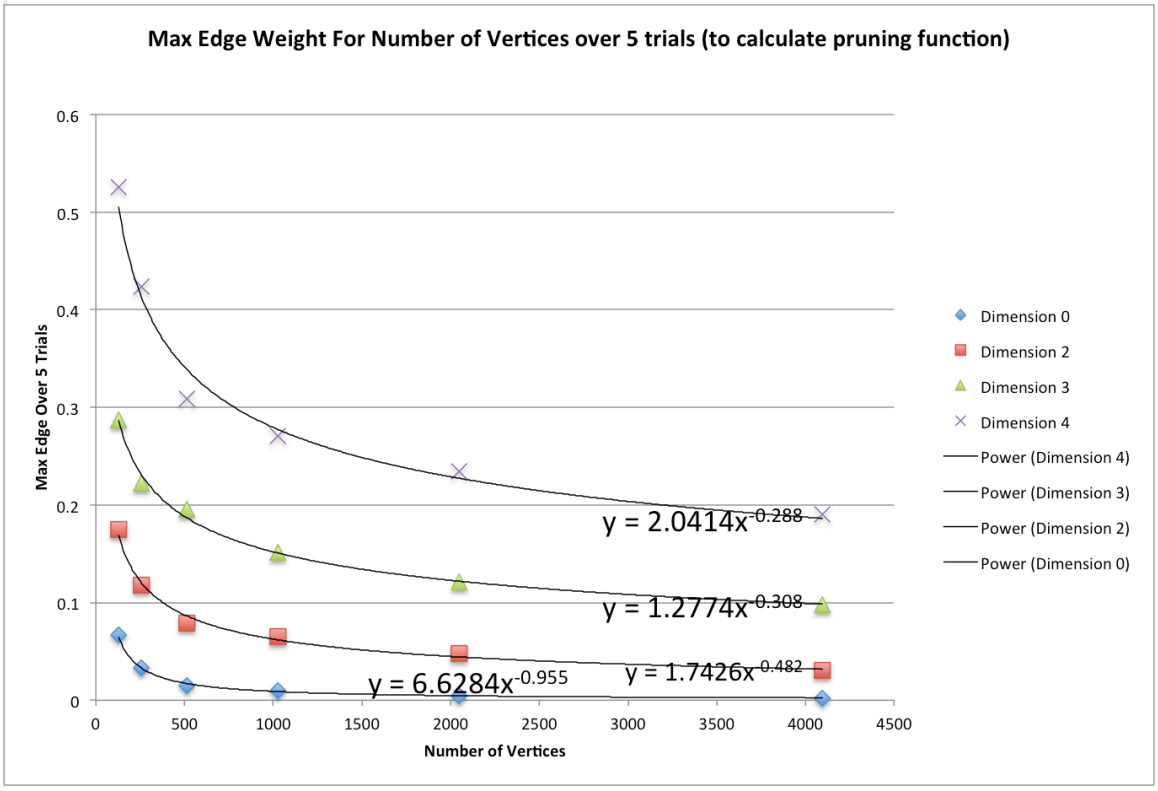

Figure 3: Average Weight of MST Over 5 Trials After Optimization

dim/n	8192	16384	32768	65536	131072
0	1.20766612	1.28839316	1.2895234	1.2054381	1.2853274
2	58.4652286	82.120817	115.280157	161.858231	227.22856
3	264.985157	416.381539	654.279163	1028.09916	1615.4969
4	598.000516	993.38071	1650.1745	2741.22098	4553.6349