CS 124 Programming Asst. 1

Joseph Kahn — Aron Szanto

February 22, 2016

Instructions

The code has detailed comments relating to implementation, testing, and so on.

In order to make our executable simply run **make randmst** from the command line. You will then be able to execute the program using **./randmst 0 numpoints numtrials dimension**.

Results

Average MST weight per input size for all graph types

Graph Size	Random Weights	Unit Square	Unit Cube	Unit Hypercube
16	0.923766	2.6467	4.68364	5.87211
32	1.2608	3.84693	7.03527	10.9462
64	1.17521	5.46695	11.2448	17.4935
128	1.21201	7.56559	17.6732	28.2733
256	1.11779	10.6306	27.9885	47.3703
512	1.21966	14.9417	43.682	77.751
1024	1.19882	21.0851	68.0085	130.3971
2048	1.20425	29.6526	107.389	216.0091
4096	1.2063	41.7697	169.335	361.871
8192	1.19731	59.0312	267.812	604.0351
16384	1.20743	83.2613	422.338	1009.951
32768	1.20452	117.509	668.31	1689
65536	1.20089	166.041	1059.3	2828.99
Estimated $f(n)$		$0.685n^{0.4947}$	$0.7427n^{0.6536}$	$0.8039n^{0.7353}$

The above "precise" estimates were calculated using the power fitting function provided by Excel. Our best actual estimates are included in the discussion section.

Discussion

Algorithm Choice

We chose to implement Kruskal's over Prim's. Note that Kruskal's has a best case running time of $O(E \log^* V)$ (if given a sorted edge list and implemented with path compression and union by rank) while Prim's has a best case running time of $O(E + V \log V)$ amortized time (if implemented with a Fibonacci heap).

There were two main reasons for our decision, one algorithmic and the other pedagogical. First, Kruskal's is comparatively faster on sparse graphs. After effective pruning, we only consider a tiny subset of the total edges of the formerly connected graph. Indeed, our analysis of $k(n)$ showed that as the number of vertices increases, the threshold for the maximum weight that could reasonably belong to an MST decreases sharply. At a value as small as 100 vertices, edges with weights larger than 0.1 could be pruned with high confidence, quickly eliminating 90 percent of the edges. Second, by using simpler and more intuitive data structures and mechanisms than we would have with Prim's (in the case of implementing a Fib heap) we were able to quickly obtain a fast running time and opportunities for optimization. Prim's algorithm only really competes with Kruskal's if it is implemented via a Fibonacci heap, and we decided that the expected learning we'd get out of a poorly tested project with a working Fib heap was strictly less than that which we'd get out of an elegantly implemented version of Kruskal's on a pruned graph, with lots of optimization and rigorous testing.

Graph Pruning

To estimate a value for $k(n)$ we wrote the function **testMaxWeight**. This runs 100 trials of our MST algorithm (without pruning) on graphs of up to $n = 400$ vertices and returns the absolute maximum edge included in the MST (over all trials, for each value of n, in each possible dimension). We then exported this data to Excel and fitted the curves, finding that power functions best minimized residuals, and calculated reasonable error bounds. The error bounds are calculated by subtracting the estimated value predicted by $k(n)$ from the observed value. You can see the graphs we made, as well as our calculations for $k(n)$ in the appendix (along with

graphs showing the growth rates of $f(n)$. In order to guarantee that the graphs we generate are complete (even in the worst case where a "correct" edge for the MST would be thrown out given $k(n)$, we add a buffer to the pruning threshold that maintains the computational advance given by pruning while eliminating the possibility of erroneously excising an edge.

Possible Improvements

Given more time we would have wanted to implement spacial hashing as well as multithreading.

The former involves placing vertices into buckets based on their position in n-dimensional space. Currently we need to consider every edge before throwing out those which are too large. By using spacial hashing one is able to not only throw out unnecessary edges, but also limit the search space to begin with. This improvement would have associated each vertex with a subspace index and would eliminate the necessity of considering edges that are too far apart in space to possibly be included in the MST.

The latter improvement would have made the collection of data easier while also significantly decreasing running time; since our operations are nearly entirely processor bound, spawning threads to calculate the average weights of various dimensions in parallel or processing multiple subsections of the edge list simultaneously would be feasible. Though concurrency would not improve our asymptotic bound, it would cut down on the runtime significantly.

Understanding the MST's Average Weight — f(n,d)

In the case of randomly assigning edge weights (with a Unif(0,1) distribution) we found that the total weight converged extremely quickly to 1.20. In the 2D, 3D and 4D cases, while we did not find convergence, we did observe that the curves we fitted to our data clearly show that the growth rate decreases as the size of the graph increases (but do tend towards infinity as n grows extremely large). Upon analyzing the best fit for our average weights by n, we found that we can approximate the following functional form:

$$f(n, d) = c(d)n^{\frac{d-1}{d}}$$

where n is the number of vertices, d is the dimensionality of the graph, and $c(d)$ is a

constant factor that depends on the d. Through linear regression, we estimated $c(d)$ to be

$$c(d) = 0.057(d + 10)$$

We recognize that 0.057 is a strange number and have been trying to figure out what its significance could be. Our best guess as yet is that it is a few hundredths away from 1/16 and that this could be the true value. In any case, the key insight here is the $n^{\frac{d-1}{d}}$ term, which dictates that as the number of dimensions grows, the growth rate of the expected total weight of the tree tends towards linearity. Our experimental values of $\frac{d-1}{d}$ bear out this observation, with values of 0.00, 0.49, 0.65, and 0.74 for $d = 1, 2, 3, 4$ respectively. [1]

Running Time

After completing the assignment itself we decided to run our code (fairly arbitrarily) on even larger values of n. Here are some results.

A graph with 200000 vertices (2^{34} edges) in 4D took on average 186 seconds to be generated. Including sorting, it took our MST algorithm 25.83 seconds to find a path.

A graph with 500000 vertices (2^{37} edges) in 4D took on average 1037.38 seconds to be generated. Including sorting, it took our MST algorithm 144.824 seconds to find a path. Note that the increase in graph generation time roughly corresponds to a linear factor in the number of edges. This is a good sanity check to demonstrate that our code is handling the correct number of edges even on very large graphs and not doing any excess work.

It should be clear from these examples that the real bottleneck in our code relates to the generation of graphs and the throwing out of edges. Even with a significant number of vertices, our MST algorithm performs reasonably fast. Thankfully, the process of graph generation is easily multithreaded and perhaps calls for future optimization.

Our Experience With The Code

We learned quite a bit about the C++ random number libraries over the course of the project. Most notably, when generating hundreds of thousands of vertices and

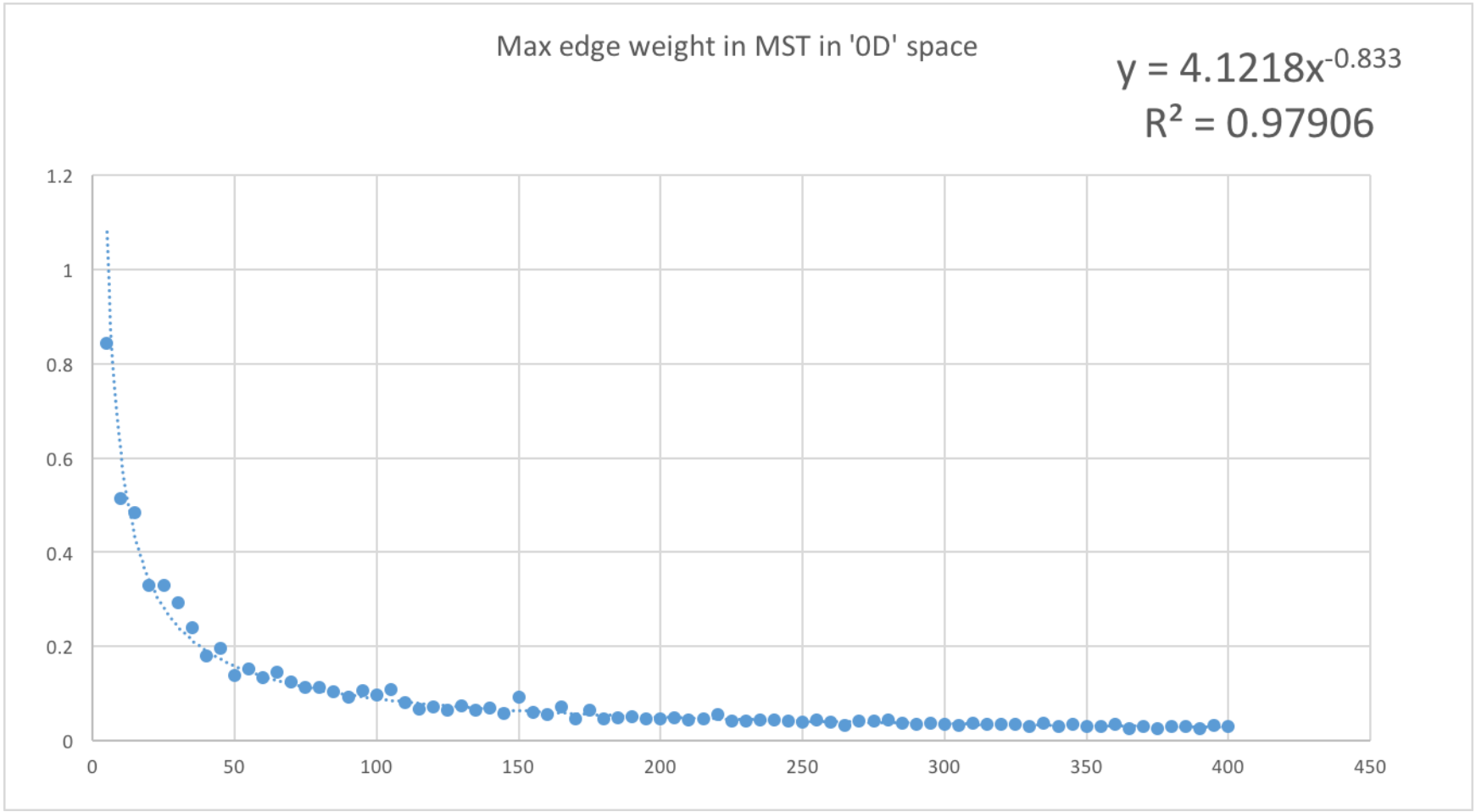
[1] We recognize a minor inconsistency here with terminology- in every other case we refer to graphs with randomly assigned edges that aren't generated stochastically from their relative location in euclidean space as "0-Dimensional")

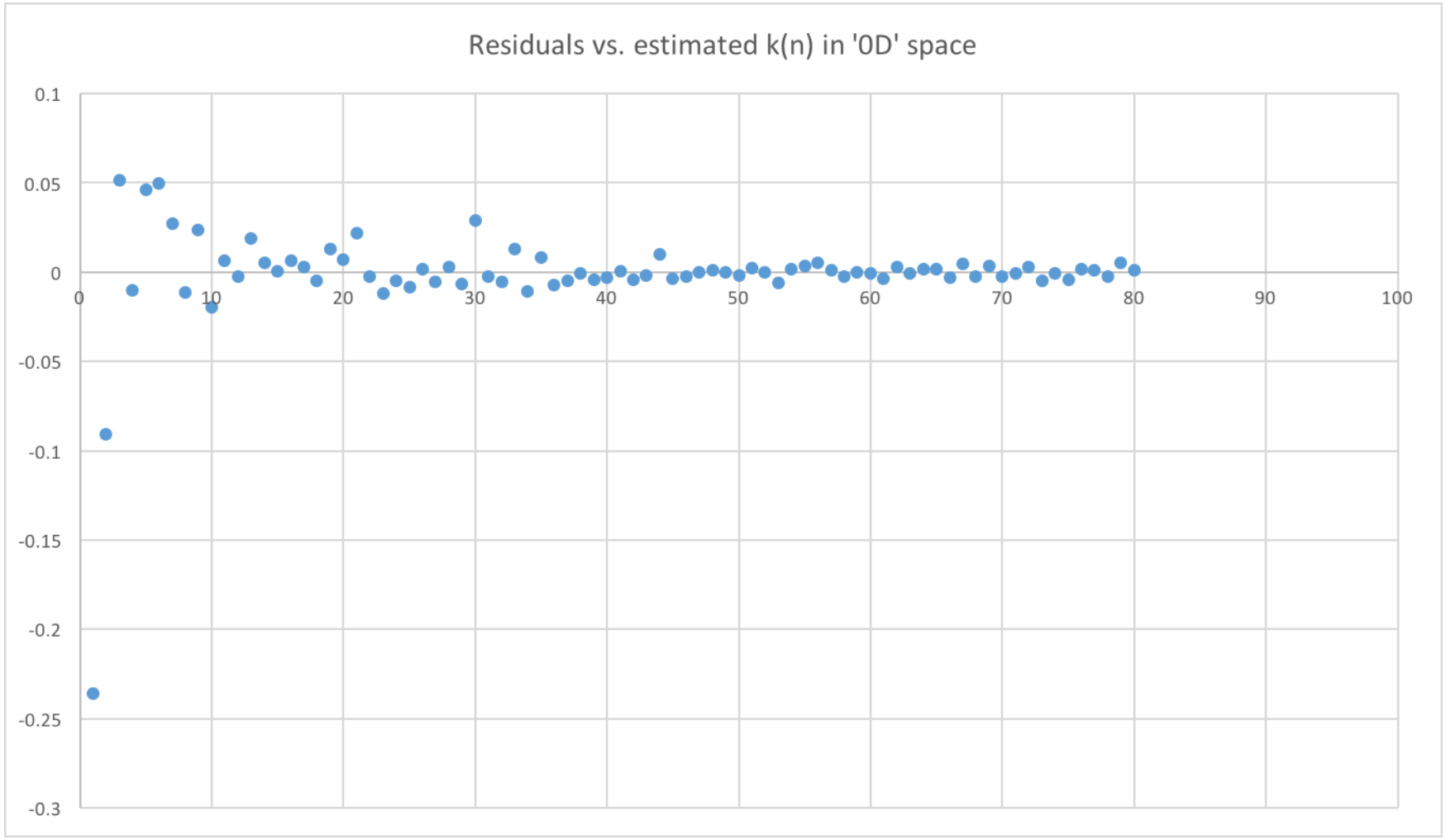
thus edges numbering in the neighborhood of 10^{10}, the C++98 standard of 13 bits of randomness wasn't going to cut it. We calculated that we would roll over on the state of the stock PRNG several times. Instead, we used the new(er) standard for C++ and decided on a Mersenne Twister generator given its large $(2^{19937} - 1)$ period, fast performance, and ability to generate up to the maximum (53) bits of randomness for a C++11 double. In this way, we were guaranteed to have far superior performance with respect to the quality of our random numbers.

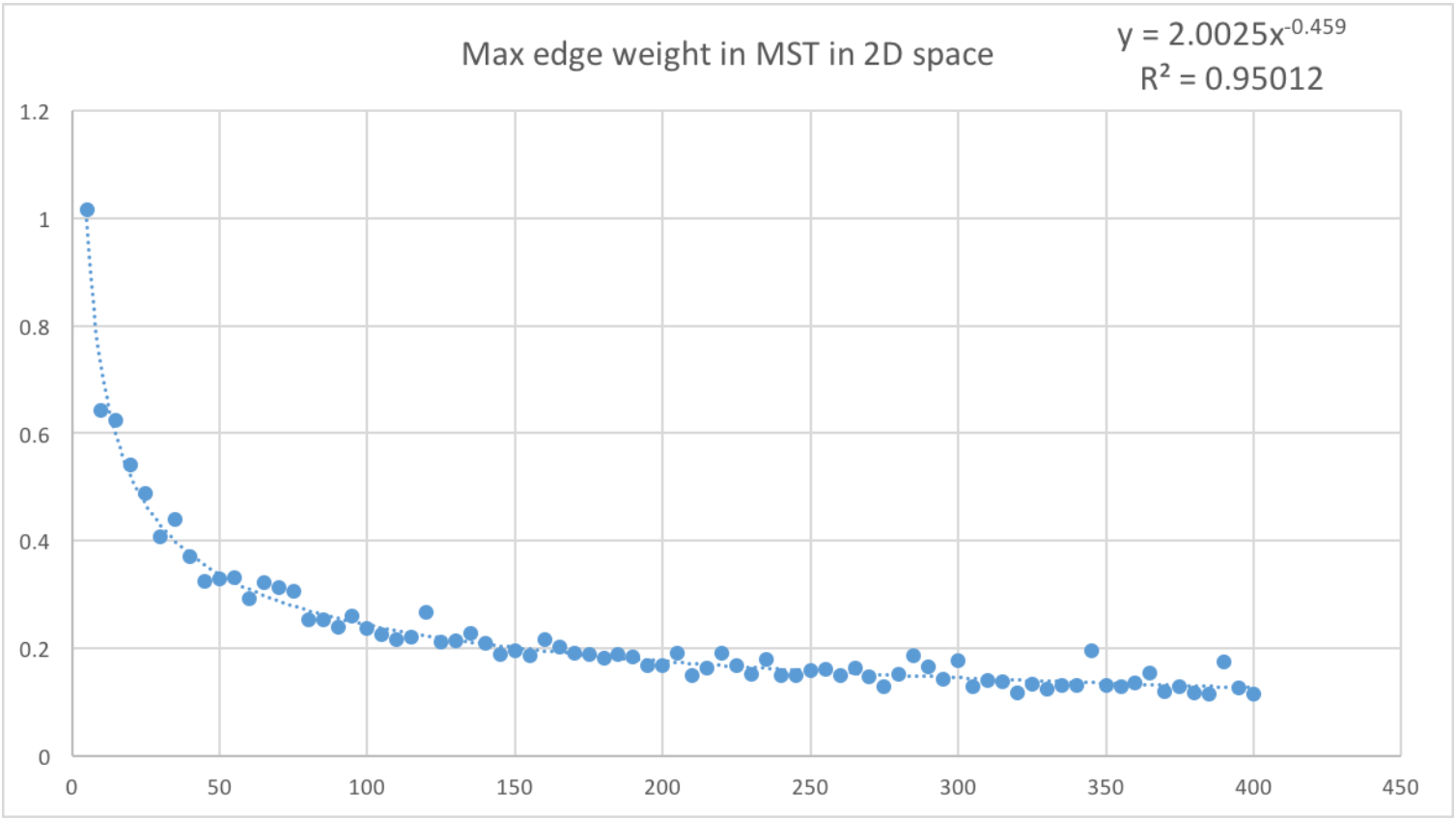
We also learned about combining coding with Excel analysis to improve the speed of our algorithm. Running the two sets of curve fitting was illuminating and it was especially interesting to see how closely our functional approximations came to a beautiful closed form solution to the exercise.

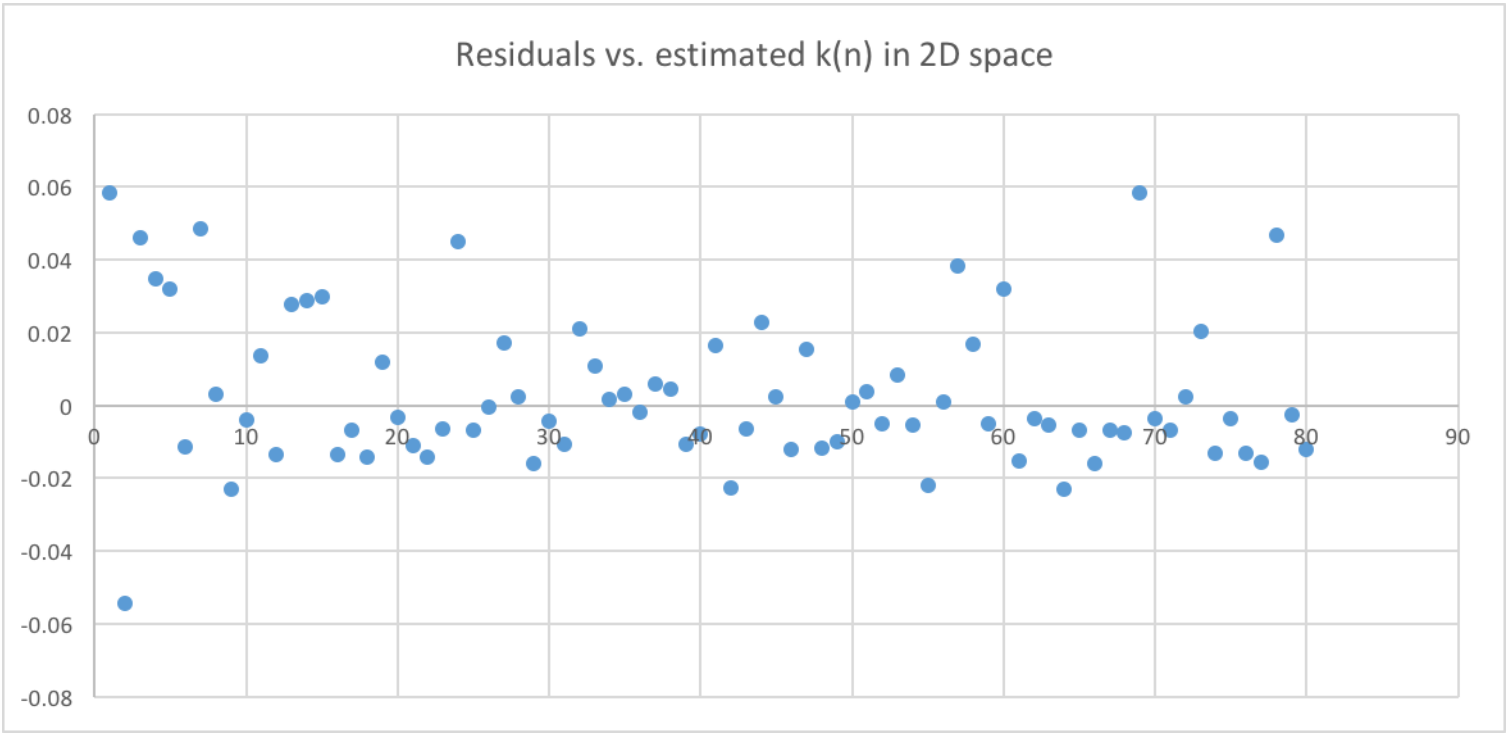
Last, we learned about writing code for the general case rather than for specific ones. Being able to test on graph sizes far beyond 2^{16} and in arbitrary dimensions was instrumental in figuring out our functional estimate (we ran tests in up to 100,000 dimensions).

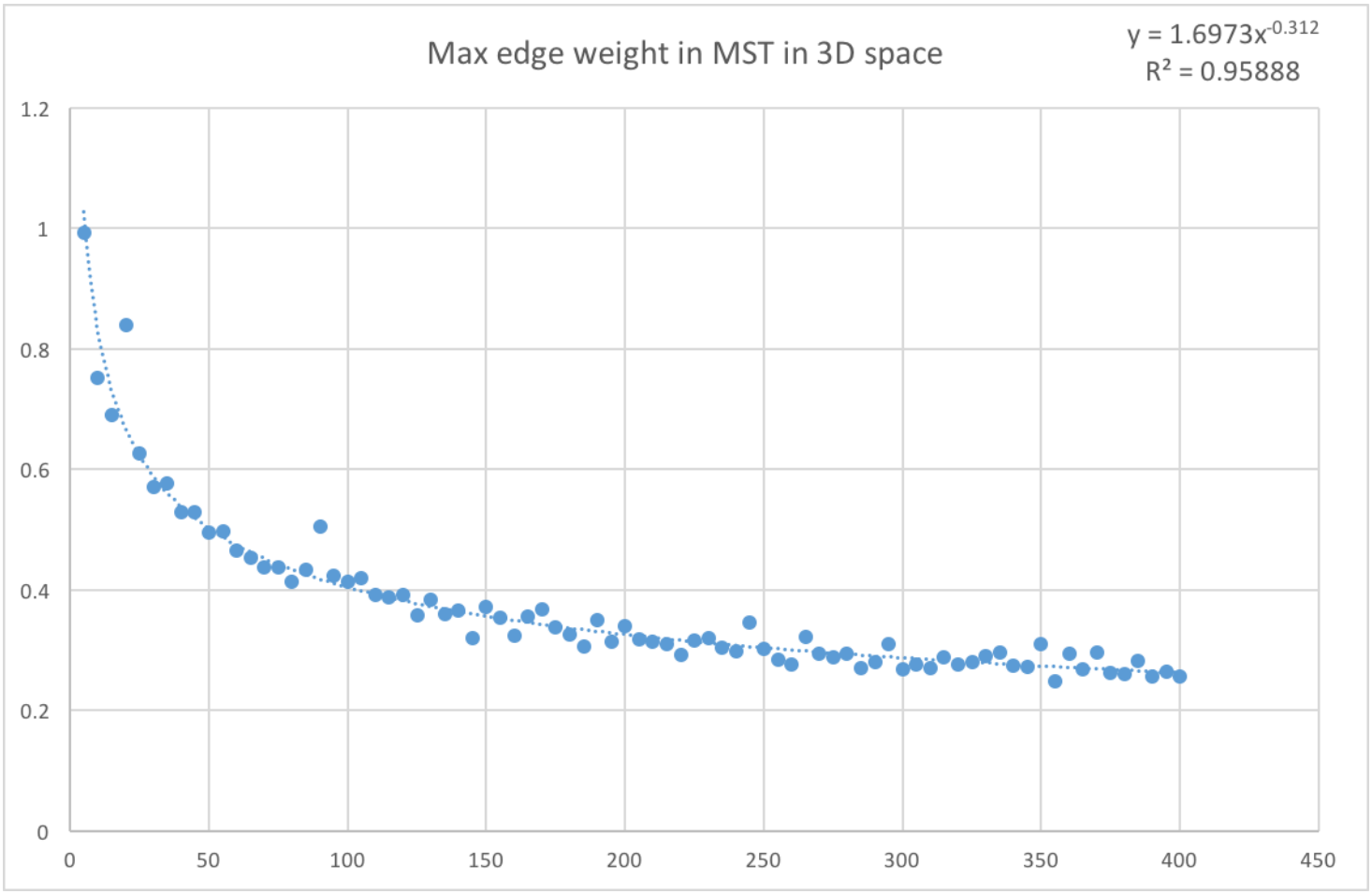
This concludes our report. The appendix is long, but simply includes graphs generated in Excel (one per page).

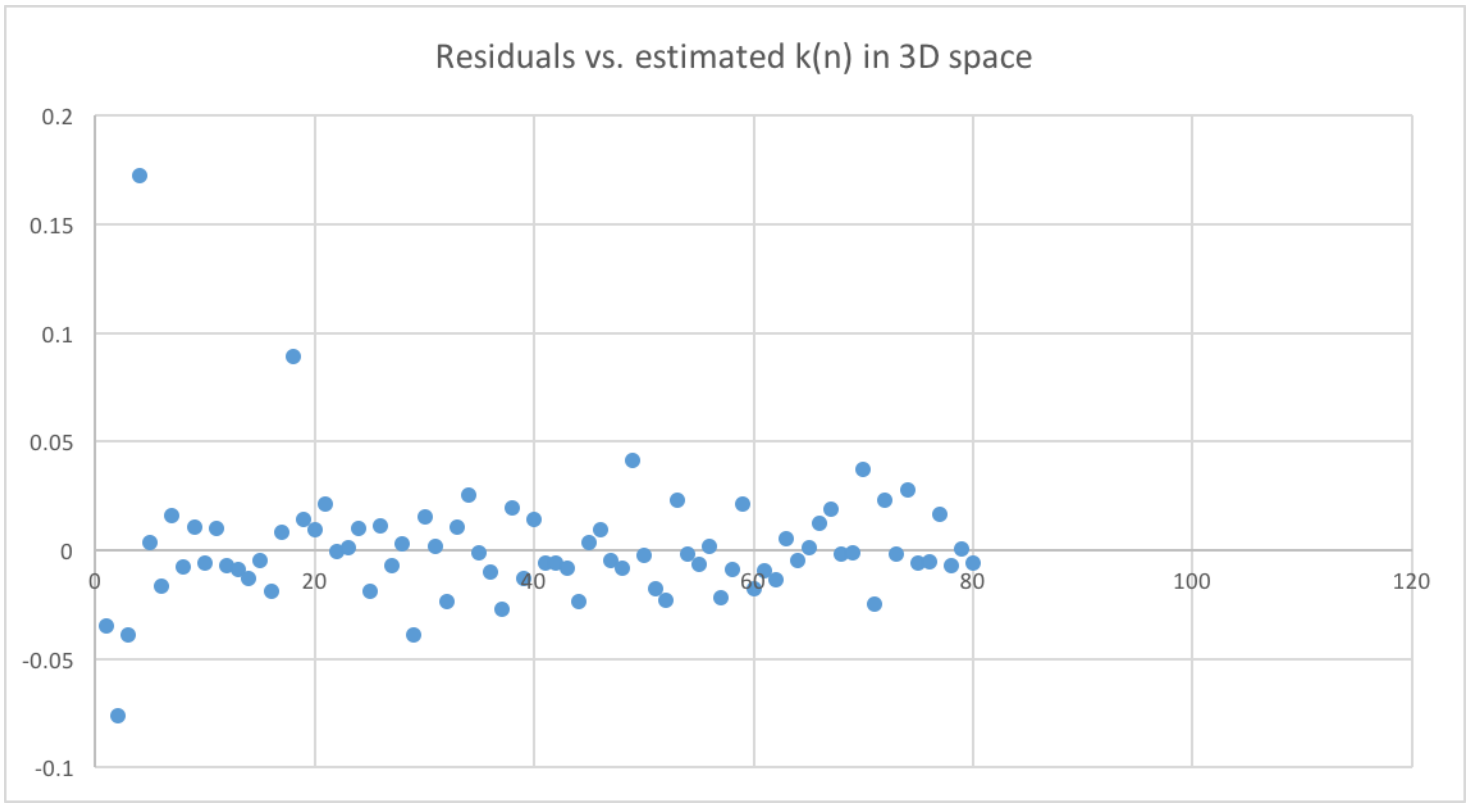

Max edge weight in MST in '0D' space
y = 4.1218x^-0.833
R² = 0.97906
1.2
1
0.8
0.6
0.4
0.2
0
0
50
100
150
200
250
300
350
400
450

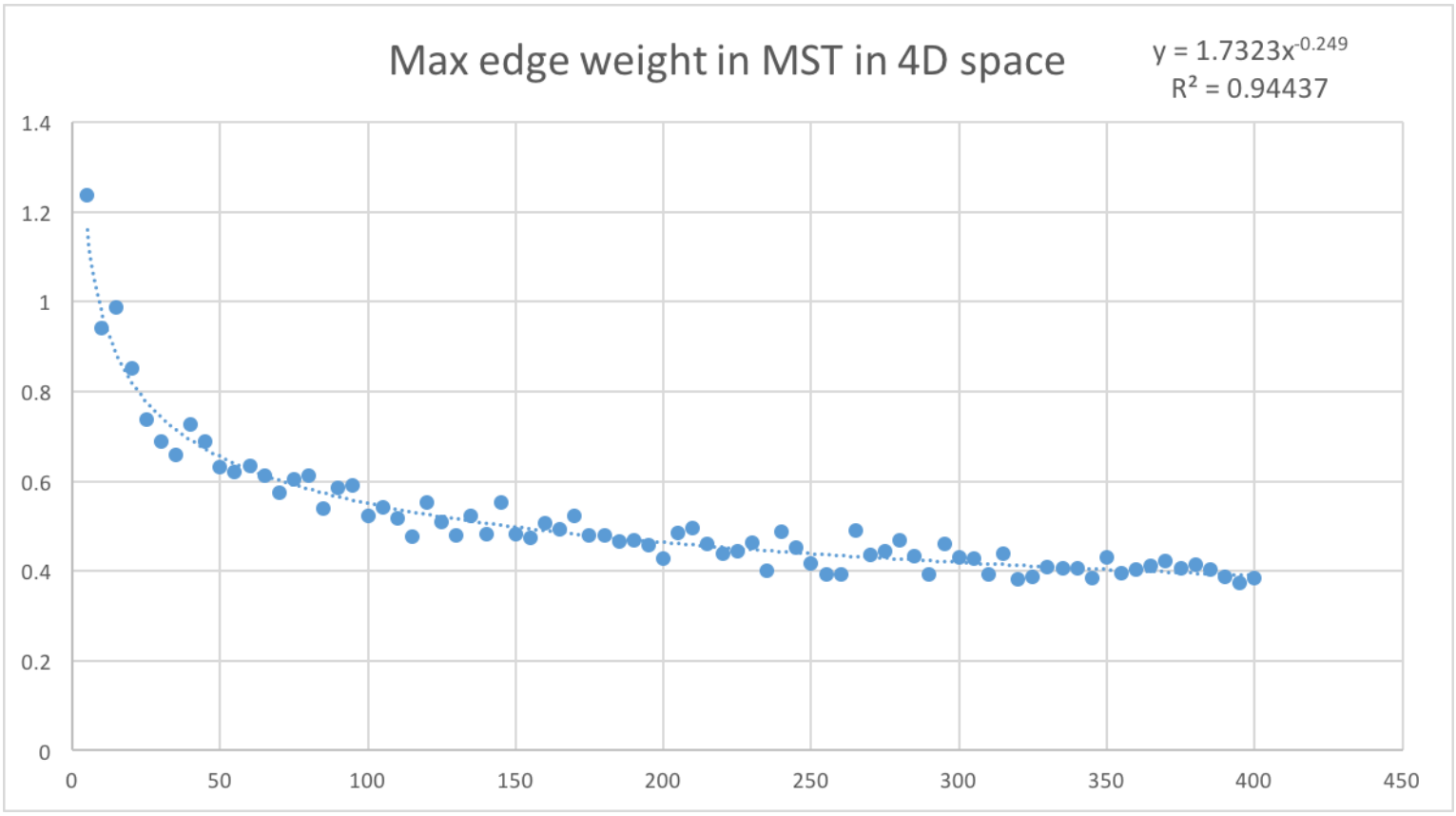

Residuals vs. estimated k(n) in '0D' space
0.1
0.05
0
-0.05
-0.1
-0.15
-0.2
-0.25
-0.3
0
10
20
30
40
50
60
70
80
90
100

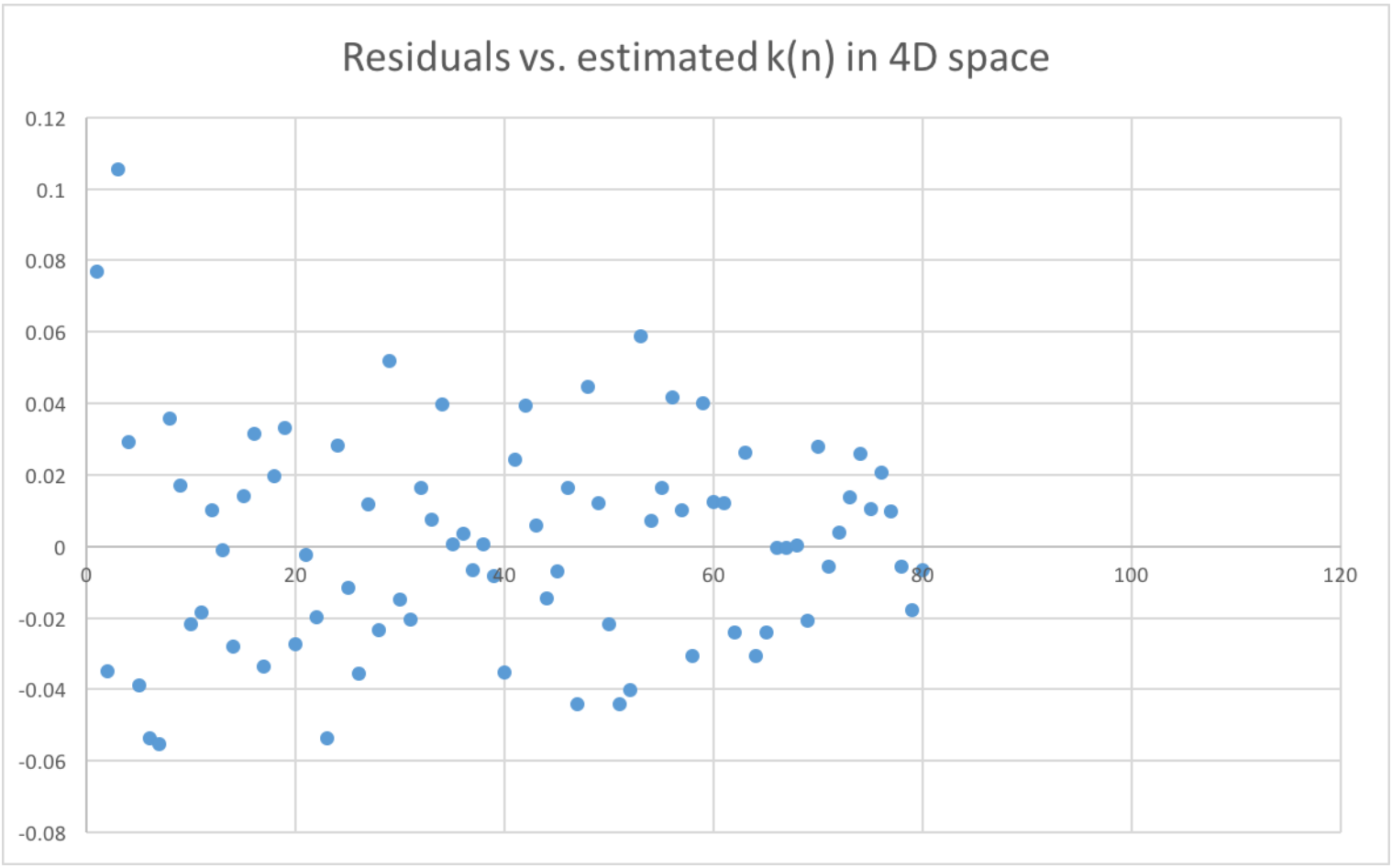

Max edge weight in MST in 2D space
y = 2.0025x^-0.459
R² = 0.95012
1.2
1
0.8
0.6
0.4
0.2
0
0
50
100
150
200
250
300
350
400
450

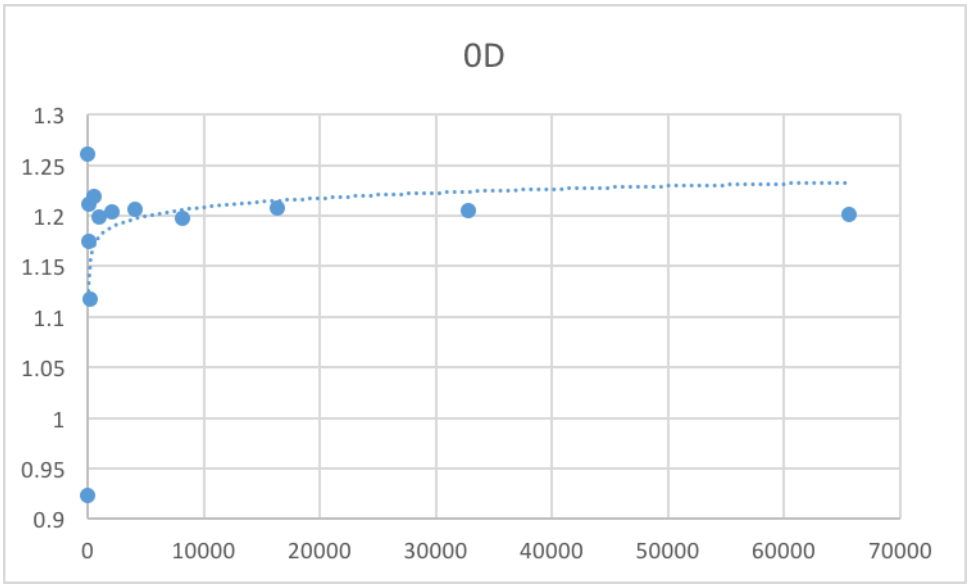

Residuals vs. estimated k(n) in 2D space
0.08
0.06
0.04
0.02
0
-0.02
-0.04
-0.06
-0.08
0
10
20
30
40
50
60
70
80
90

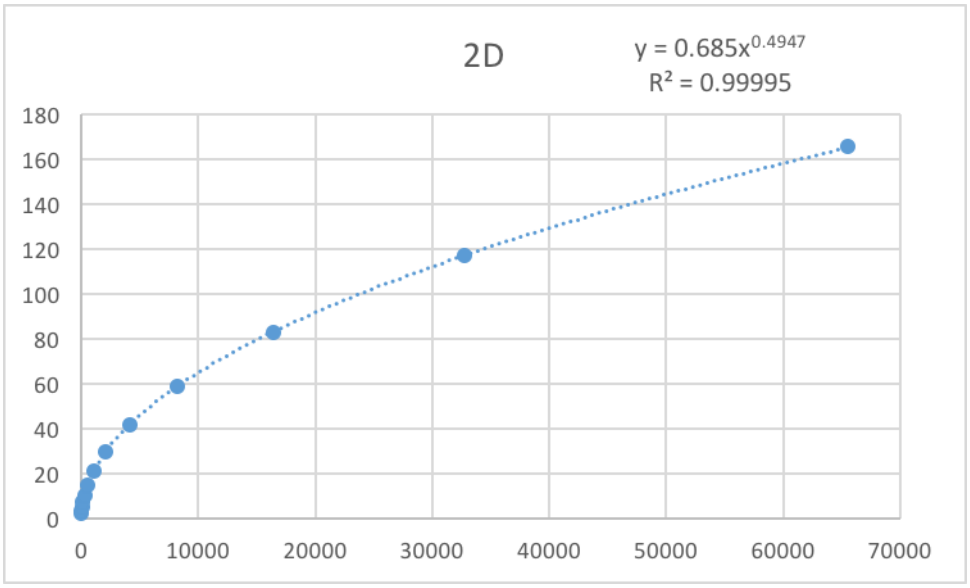

Max edge weight in MST in 3D space
y = 1.6973x^-0.312
R² = 0.95888
1.2
1
0.8
0.6
0.4
0.2
0
0
50
100
150
200
250
300
350
400
450

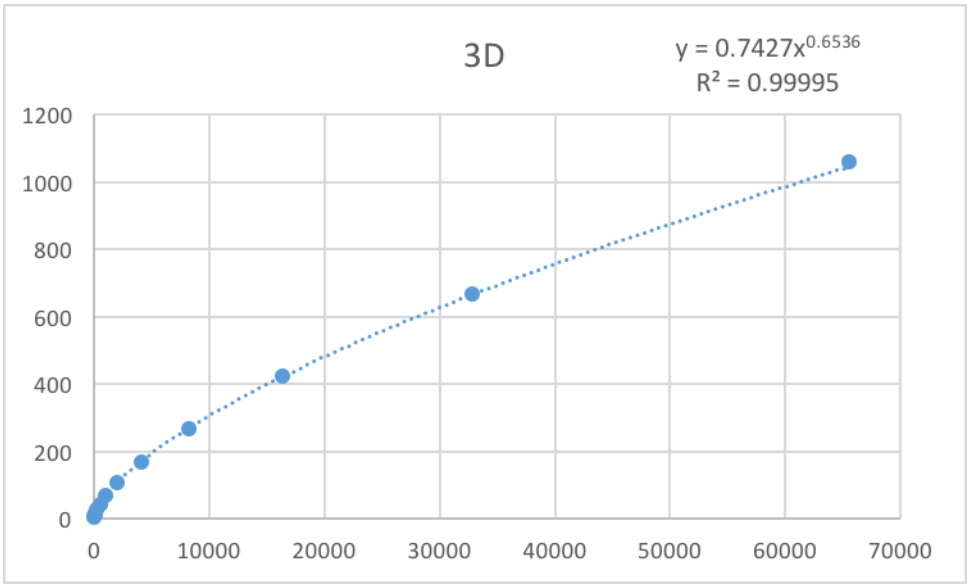

Residuals vs. estimated k(n) in 3D space
0.2
0.15
0.1
0.05
0
-0.05
-0.1
0
20
40
60
80
100
120

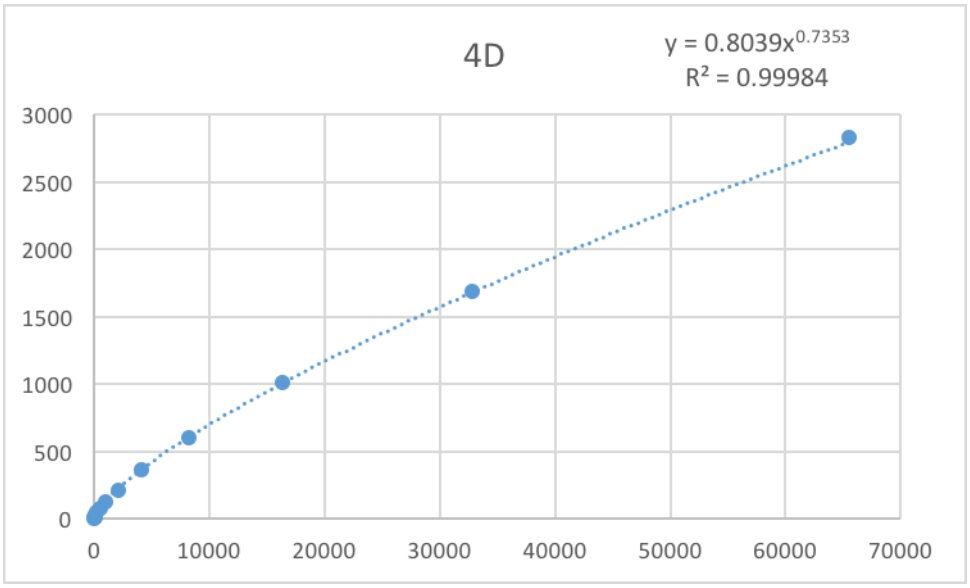

Max edge weight in MST in 4D space
y = 1.7323x^-0.249
R² = 0.94437
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
50
100
150
200
250
300
350
400
450


Residuals vs. estimated k(n) in 4D space
0.12
0.1
0.08
0.06
0.04
0.02
0
-0.02
-0.04
-0.06
-0.08
0
20
40
60
80
100
120


0D
1.3
1.25
1.2
1.15
1.1
1.05
1
0.95
0.9
0
10000
20000
30000
40000
50000
60000
70000


2D
y = 0.685x^0.4947
R² = 0.99995
180
160
140
120
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000
70000


3D
y = 0.7427x^0.6536
R² = 0.99995
1200
1000
800
600
400
200
0
0
10000
20000
30000
40000
50000
60000
70000


4D
y = 0.8039x^0.7353
R² = 0.99984
3000
2500
2000
1500
1000
500
0
0
10000
20000
30000
40000
50000
60000
70000