Programming Assignment

Your name(s): Di Zhen

Overview:
The purpose of this assignment is to experience some of the problems involved with implementing an algorithm (in this case, a minimum spanning tree algorithm) in practice. As an added benefit, we will explore how minimum spanning trees behave in random graphs.

Assignment:

We will be considering *complete, undirected* graphs. A graph with n vertices is complete if all $\binom{n}{2}$ pairs of vertices are edges in the graph.

Consider the following types of graphs:

- Complete graphs on n vertices, where the weight of each edge is a real number chosen uniformly at random on $[0, 1]$.
- Complete graphs on n vertices, where the vertices are points chosen uniformly at random inside the unit square. (That is, the points are (x, y), with x and y each a real number chosen uniformly at random from $[0, 1]$.) The weight of an edge is just the Euclidean distance between its endpoints.
- Complete graphs on n vertices, where the vertices are points chosen uniformly at random inside the unit cube (3 dimensions) and hypercube (4 dimensions). As with the unit square case above, the weight of an edge is just the Euclidean distance between its endpoints.

Your first goal is to determine in each case how the expected (average) weight of the minimum spanning tree (not an edge, the whole MST) grows as a function of n. This will require implementing an MST algorithm, as well as procedures that generate the appropriate random graphs. You may implement any MST algorithm (or algorithms!) you wish; however, I suggest you choose carefully.

For each type of graph, you must choose several values of n to test. For each value of n, you must run your code on several randomly chosen instances of the same size n, and compute the average value for your runs. Plot your values vs. n, and interpret your results by giving a simple function $f(n)$ that describes your plot. For example, your answer might be $f(n) = \log n$, $f(n) = 1.5\sqrt{n}$, or $f(n) = \frac{2n}{\log n}$. Try to make your answer as accurate as possible; this includes determining the constant factors as well as you can. On the other hand, please try to make sure your answer seems reasonable.

Code setup:

So that we may test your code ourselves as necessary, please make sure your code accepts the following command line form:

./randmst 0 numpoints numtrials dimension

The flag 0 is meant to provide you some flexibility; you may use other values for your own testing, debugging, or extensions. Note: you should test your program – design tests to make sure your program is working, for example by checking it on some small examples (or find other tests of your choosing). You don't need to put your tests in your writeup, that is for you.

The value numpoints is n, the number of points; the value numtrials is the number of runs to be done; the value dimension gives the dimension. (Use dimension 2 for the square, and 3 and 4 for cube and hypercube, respectively; use dimension 0 for the case where weights are assigned randomly. Notice that dimension 1 is just not that interesting.) The output for the above command line should be the following:

average numpoints numtrials dimension

where average is the average minimum spanning tree weight over the trials.

Please pay attention to the following requirements. In order to grade appropriately, our objective is to ensure that we can run the programs without any special per-student attention. Hence we require:

- If possible, for compatibility reasons, the code should run on the Harvard system (nice.fas.harvard.edu cluster of Linux machines – you can use an ssh client to log in if you have a Harvard account), even if you code on another system.
- We expect an executable with the code as described above (input as described, answers should go to standard output as described).
- The code should compile with make; no instructions for humans. That is, the command "make randmst" should produce an executable from your directory. You may need to read up on makefiles to make this happen.

What to hand in: Besides *submitting a copy of the code you created*, your group should hand in a single well organized and clearly written report describing your results. For the first part of the assignment, this report must contain the following quantitative results (for each graph type):

- A table listing the average tree size for several values of n. (A *graph* is insufficient, although you can have that too; we need to see the actual numbers. If you have a graph but no table of actual numbers, I will take points off.)
- A description of your guess for the function $f(n)$.

Run your program for n = 128; 256; 512; 1024; 2048; 4096; 8192; 16384; 32768; 65536; 131072; 262144; and larger values, if your program runs fast enough. (Having your code handle up to at least n = 262144 vertices is one of the assignment requirements; however, handling n up to 131072 will result in only losing 1-2 points. Providing results only for smaller n will hurt your score on the assignment.) Run each value of n at least five times and take the average. (Make sure you re-seed the random number generator appropriately!)

In addition, you are expected to discuss your experiments in more depth. This discussion should reflect what you have learned from this assignment; the actual issues you choose to discuss are up to you. Here are some possible suggestions for the second part:

- Which algorithm did you use, and why?
- Are the growth rates (the $f(n)$) surprising? Can you come up with an explanation for them?
- How long does it take your algorithm to run? Does this make sense? Do you notice things like the cache size of your computer having an effect?
- Did you have any interesting experiences with the random number generator? Do you trust it?

Your grade will be based primarily on the correctness of your program and your discussion of the experiments. Other considerations will include the size of n your program can handle. Please do a careful job of solid writing in your writeup. Length will not earn you a higher grade, but clear descriptions of what you did, why you did it, and what you learned by doing it will go far.

Hints:

To handle large n, you may want to consider simplifying the graph. For example, for the graphs in this assignment, the minimum spanning tree is extremely unlikely to use any edge of weight greater than $k(n)$, for some function $k(n)$. We can estimate $k(n)$ using small values of n, and then try to throw away edges of weight larger than $k(n)$ as we increase the input size. Notice that throwing away too many edges may cause problems. Why will throwing away edges in this manner never lead to a situation where the program returns the wrong tree?

You may invent any other techniques you like, as long as they give the same results as a non-optimized program. Be sure to explain any techniques you use as part of your discussion and attempt to justify why they should give the same results as a non-optimized program!

Answer:

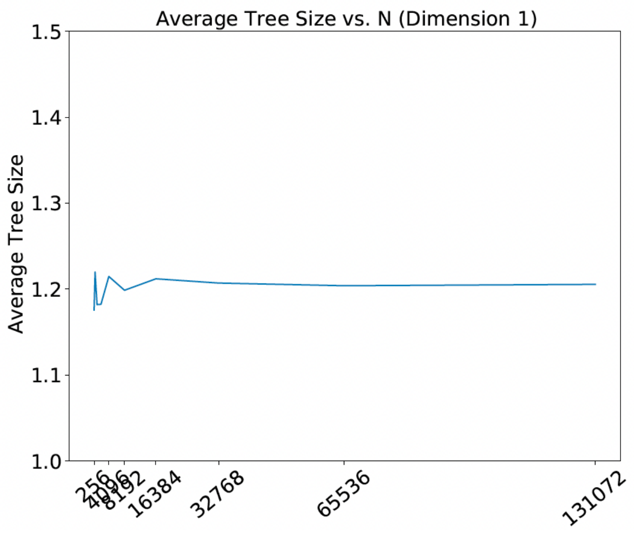
n / average tree size	**1D**	**2D**	**3D**	**4D**
128	1.18434	7.38176	16.14594	27.01182
256	1.17561	9.15099	26.74171	43.29308
512	1.21985	12.90942	40.31291	75.52401
1024	1.18189	18.45081	63.63972	122.64023
2048	1.18220	25.70701	100.70772	206.90013
4096	1.21460	36.50370	157.97822	345.76889
8192	1.19995	51.57175	251.09888	578.11931
16384	1.21189	82.84412	419.25141	997.43527
32768	1.20694	116.32509	659.97832	1672.37541
65536	1.20383	164.76741	1048.17693	2813.04273
131072	1.20541	233.16860	1671.64372	4721.13877

Table 1: The average tree size for different values of n.

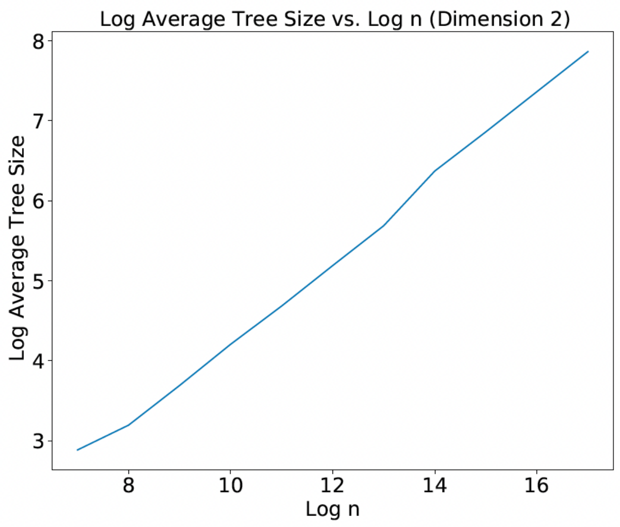
Three types of graphs are generated and represented by adjacency lists. I used Prim's algorithm with to find the MST. The reason why I choose Prim's algorithm rather than Kruskal's algorithm is that Kruskal's would take $O(|V|^2)$ space and require probably $O(|E|\log|E|)$ to sort the edges, which is more expensive than Prim's. I used a priority queue to implement the Prim's algorithm. Considering the graph is dense with $|E| = \Omega(|V|^2)$, and the runtime of Prim's depends on the data structure used, specifically how 'deletemin' and 'insert' operations are implemented. I decided not to use heap which requires $\log|V|$ for both operations and ends up with $O((|V|+|E|)\log|V|) = O((|V|^2)\log|V|)$ runtime. My choice is to implement a priority queue which requires $O(1)$ to insert and $O(|V|)$ to deletemin. The 'insert' function inserts node and length into a Hash table. If the node exists, only update the length is the length is smaller than the previous length of that node. The deletemin function returns the node and length pair with the minimal length, and delete that pair from the Hash table. So the total runtion is $O(|V|^2 + |E|) = O(|V|^2)$.

To handle large n, for each type of graph, I calculated a upper bound of the edge weight, above which the Prim's would hardly or never choose to include in the MST, and then used the upper bound to exclude some edges from the graphs before implementing Prim's to find MST. The upper bound were found manually (improve.py). I gave the formula of the upper bound for each dimension: $m^{-1/b}$ where b is a constant that needs to be found. I iterated through $m = 4, 5, 6..., 200$ to calculate the upper bound as well as the maximum length of the MST. I adjusted the value of b to make sure that the upper bound $m^{-1/b}$ is indeed larger than the max length of edge for all m. Thus, the final choices of b is $1.661, 2.660, 3.869, 5.446$ for $d = 1, 2, 3, 4$. The program ran for n = 128; 256; 512; 1024; 2048; 4096; 8192; 16384; 32768; 65536; 131072, and reported the average tree size for each n over 5 repeats (MST.py). Python(3.8) is the programming language of choice for this assignment.

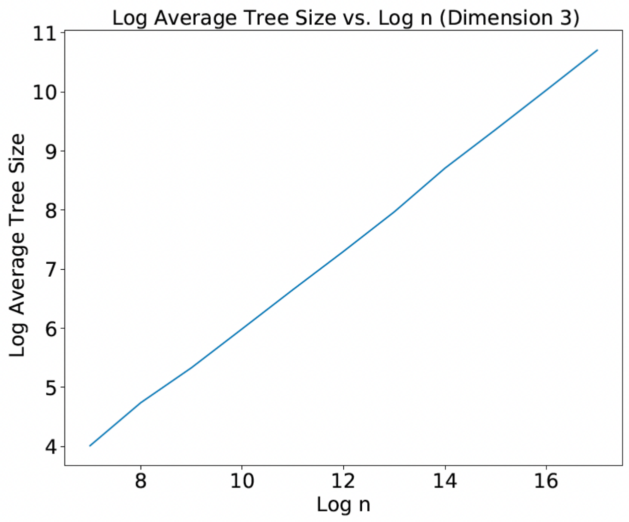
Table 1 gives the average tree sizes for different values of n. Figure 1 visualizes the data in Table 1 (plot.py). When dimension is 1, the average tree sizes are around 1.2 which is independent of n. When n is small, the curve is a little fluctuated. As n goes large, the curve is relatively flat. This might imply that the average tree size goes to a constant around 1.2 when n goes to infinity. When the dimension is larger than 1, relationship between average tree size and n is a curve with a decreasing coefficient. After taking the log (base 2) of both average tree size and n, there is a linear relationship between then as shown in



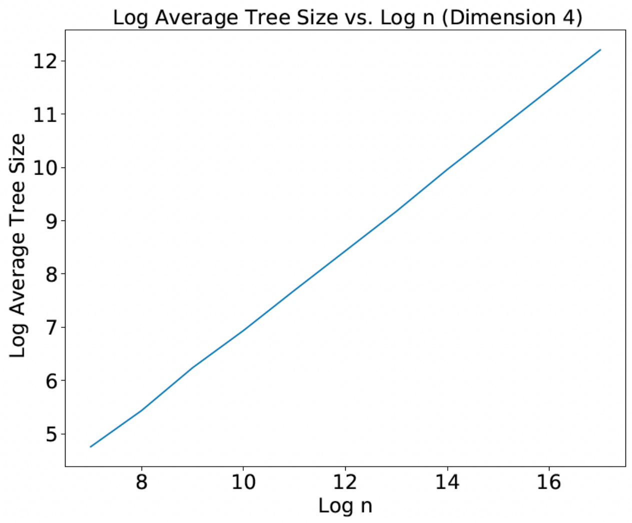
(a)



(b)



(c)



(d)

Figure 1: Plots show the relationship between average tree size and n.

the plots. Then I fit linear regression to identify the intercept and coefficients for these three dimensions (regression.py). For dimension 2, the formula is $\log_2 f(n) = -0.88440 + 0.51310 \log_2 n$. For dimension 3, the formula is $\log_2 f(n) = -0.66994 + 0.66816 \log_2 n$. For dimension 4, the formula is $\log_2 f(n) = -0.52019 + 0.74792 \log_2 n$. Then I tried to figure out the general form of $f(n)$. By exponentiating (base 2) both side. For dimension 2, the formula is $f(n) = 2^{-0.88440} n^{0.51310}$. For dimension 3, the formula is $f(n) = 2^{-0.66994} n^{0.66816}$. For dimension 4, the formula is $f(n) = 2^{-0.52019} n^{0.74792}$. By observing the power of n $(0.51310, 0.66816, 0.74792)$ I figured out the formula for $d > 1$ is $f(n) = cn^{1-1/d}$ where c is some constant which increases as d increases and d is the number of dimension $(d > 1)$.

When $d = 1$, the average tree size is a constant around 1.2. A MST with n vertices has $n - 1$ edges, so the MST size must be at least the sum of all $n - 1$ minimal edges in the graph with $N = n(n-1)/2$ edges. Since the lengths of edges are uniform distributed, the expected length of the ith edge is i/N, the sum of $n - 1$ minimal edges is $\sum_{i=1}^{n-1} i/N = 1$. Since the sum of $n - 1$ minimal edges is independent of n, it makes sense that the average tree size is independent of n. This formula makes sense because as d goes large, the maximum length of edge that would appear in the MST increases.