CS 124 Programming Assignment 1: Spring 2022

Your name(s) (up to two): Divya Amirtharaj and Charles Ma

Collaborators: (You shouldn't have any collaborators but the up-to-two of you, but tell us if you did.)

No. of late days used on previous psets:
No. of late days used after including this pset:

Homework is due Wednesday at 11:59pm ET. You are allowed up to **twelve** (college)/**forty** (extension school) late days through the semester, but the number of late days you take on each assignment must be a nonnegative integer at most **two** (college)/**four** (extension school).

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: You may work in groups of two, or by yourself. Both partners will receive the same grade and turn in a single joint report. (You should each submit a copy of the report in Gradescope.)

I recommend using a common programming language such as Java, C, or C++. If you use a more obscure language, that is fine, but if there are errors that are correspondingly harder to find it may cause your grade to be lower. I might advise you not to use a scripting language like Python, although many students successfully do the assignment in Python.

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!

1 Results

Below is a table of the average size of the MST found (over multiple trials) for complete graphs with varying dimension and number of points.

Numpoints (n)	**dim = 0**	**dim = 2**	**dim = 3**	**dim = 4**
128	1.184449	7.577097	17.450769	27.472811
256	1.135384	10.205153	27.899649	47.352406
512	1.173326	15.161670	44.509750	78.659111
1024	1.205634	21.044874	68.636055	129.599808
2048	1.200240	29.750153	107.047554	216.947266
4096	1.207501	41.896603	168.987457	360.417847
8192	1.211114	59.140800	267.706116	604.439819
16384	1.228650	82.990036	422.947266	1009.7752693
32768	1.222313	117.688385	669.216919	1691.187988
65536	1.222579	166.069702	1059.939209	2833.087158
131072	1.221724	228.192479	1678.628052	4749.805664

Our graph unfortunately wasn't able to handle above 131072 (not necessarily because of timing but sometimes we would run out of memory and seg fault or for a few other problems). Bottom line is that our implementation was expedient until the $n = 262144$ where it could not handle the case anymore, but we are ok with the 1-2 point dock because we are confident in our results for our other trials (and we hope that you are too!).

1.1 Function Representation

After plotting our results, we see that the growth rate for average size is the function $f(n) = a * n^{-c}$ where n is the number of vertices and a and c are constants.

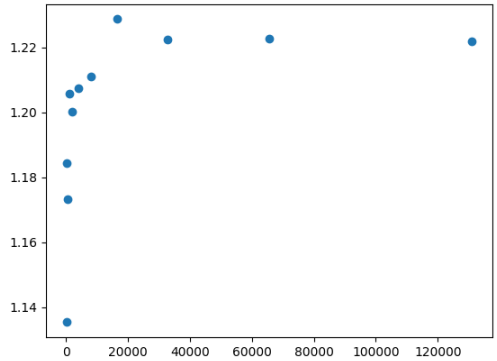

Figure 1: Dimension 0, relationship between numbers of vertices and average weight of MST

There is some very rough correlation for dimension 0 (we assign random weights to all edges). Notice that our MST total values are decently smaller for smaller n but increase rapidly between $0 < n < 20000$.

For there it caps of at an MST weight of about 1.22. I can say that the function representing dimension 0 is about $n^{\frac{1}{4}}$. Since the correlation for results for dimension is so low, it is hard for us to definitively identify a function that models our results, but plotting $n^{\frac{1}{4}}$ decently matches the trend of dimension 0 results.

Instead of consistently growing, dimension 0 seems to reach a carrying capacity, which makes sense since its edge weights are assigned randomly and as there are more and more vertices we need to add to the MST, there is also a proportionally higher chance that we will get edges with lower weights. Once n increases past a certain point, it seems the added edges with lower randomly assigned weights balances out the fact that we need to have more edges contribute to the MST weight.

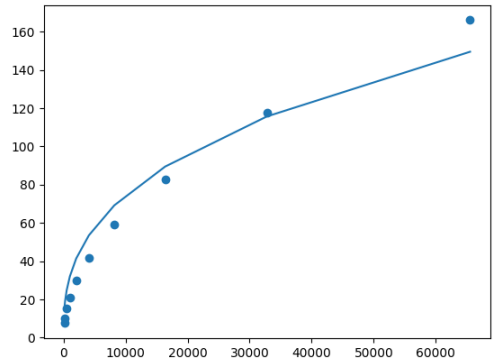

Figure 2: Dimension 2, relationship between numbers of vertices and average weight of MST

We see a clear correlation here on the order of $f(n) = a * n^{-c}$ where n is the number of points and a, c are some constants. After playing around with potential functions, we decided that the correlation roughly matches $f(n) = e^{0.9} * n^{2.7}$.

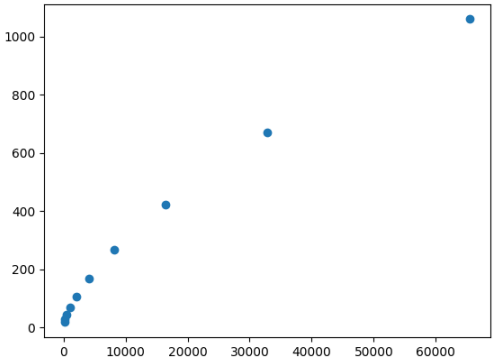

Figure 3: Dimension 3, relationship between numbers of vertices and average weight of MST

We see a clear correlation here on the order of $f(n) = a * n^{-c}$ where n is the number of points and a, c are some constants. The order of growth looks similar to the other dimensions, but with a different constant factor which we discover through trial and error.

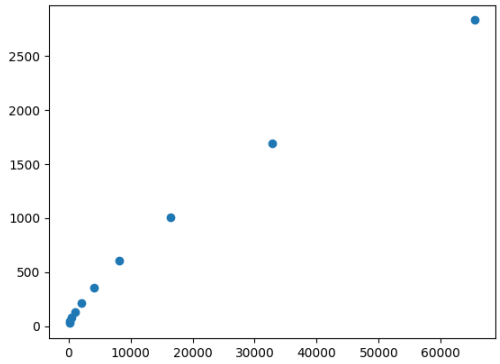

Figure 4: Dimension 4, relationship between numbers of vertices and average weight of MST

We see a clear correlation here on the order of $f(n) = n^{-c}$ where n is the number of points and c is some constant. The order of growth looks similar to the other dimensions, but with a different constant factor which we discover through trial and error. This graph growth shows a less clear diminishing rate of growth than dimension 2 and dimension 3, but that's likely because we need more data points for larger n to see the graph start to cap its growth.

1.2 Speed

Our algorithm was actually quite speedy without edge pruning. Generally we didn't have much problem with speed until maybe around the $n = 16384$ point but at that point we needed to do some pruning. We talk about our pruning process at the Optimization section of our report.

One thing we also noticed was that freeing memory made our algorithm noticeably faster. We were coding together and trying to debug something, and guessed (incorrectly) that freeing memory of variables that would no longer be in use might fix the bug. Turns out our bug was indeed something else but freeing memory made our algorithm run much faster, likely because the machine didn't have to store so much space especially for a larger n. We can see our memory freeing when we free the edges stored after running Prim's.

2 Algorithm

In order to implement the MST, we chose to use Prim's Algorithm with heaps since the runtime is dependent on the number of vertices rather than for example Kruskal's which grows with respect to number of edges (square of vertices since it's a connected graph).

As the number of vertices grows, the number of edges grows exponentially which results in dense graphs for which Prim's performs more efficiently than Kruskal's. Knowing that we would choose to trim the number of edges in the graph (using the hint provided in the assignment), we implemented Prim's algorithm using a heap the performance of the heaps is better on a sparser graph.

2.1 Optimization

Our algorithm was surprisingly fast with some of the smaller values of n. Once we got to larger values of n, we decided it was going to take too long (also we were running out of application memory), and we decided to start doing edge pruning. Our first challenge was figuring out what a threshold would be such that we wouldn't add the edge to our adjacency list (and hence not add to the graph/be evaluated in the runtime of Prim).

We decided that we couldn't use some constant threshold (our if we used some constant value like 0.05, that would make some of our smaller n value MSTs fail while our very large n values could be pruned much more heavily).

One example of just how fast pruning made our algorithm: for the case where $n = 8192$, our algorithm unpruned took a time value of 41.304871, whereas pruned our graph took a time value of only 0.236754. We plotted the average weight for the graphs up until $n = 4096$ and observed the trend line for each. Then we chose an upper bound limit that was generously above our our trend line in order to avoid overpruning. In Prim's algorithm, we search for the smallest edge between our two sets which means that after a certain upper bound (the limit we chose above our trend line), the MST will almost never or never choose an edge length of that size.

However, overpruning was still an issue for us when we used our bound on graphs with fewer than 2048 vertices since the graphs were sparser and there was more variability in edge weight. Thus, we restricted our pruning to graphs that had more than 2048 vertices since the graphs were denser to begin with and ensured that an MST was still able to be found with a generous pruning rate, and since the runtime was not overwhelming before that size.

3 Implementation

We utilied the pseudocode provided in lecture to build our Prim's Algorithm, and implemented it using heaps. Our heap definitions are described below. We took in four arguments (flag, numpoints, numtrials, and dimension) and began with our random point and edge generation which were put into adjacency lists. We go more into detail about our struggles with randomization later in this report. We then called our implementation of Prim's on the adjacency lists and outputted the average MST size, as well as the average time taken for each trial.

3.1 Heaps

We implemented five functions to assist with our heap struct and functionality.

1. buildminheap() is our first function. This function takes no inputs and returns a heap with values of 0 (or for the heapArray, allocating memory to the heapArray).
2. addNode() is our function that takes a vertex id, a weight, and a heap to add the node to. We call this function during Prim in order to add nodes to the min heap as we discover vertices not yet added to the MST that are adjacent to vertices in the MST already.
3. fixHeap() heapifies our heap once we pop nodes from our heap. Note that no where in Prim do we call fixHeap outright.

4. popMin() pops the top element from our min heap and calls fixHeap to reheapify our heap after popping the min. It return the min node that it popped so we can use it in Prims.

5. findCurrNode() finds a node given a starting node that is connected to it via the adjacency list. This is mainly use to iterate through the MST and ensure all vertices are connected, as well as to get the weights for the MST.

3.2 Randomization

Randomization extremely problematic for us since we were coding out in C. We had to seed at the right point in order to ensure we got semi-pure randomness. We would srand() with our current time (for an always unique seed) at the beginning of main so that each time we run, we get unique edgeArr. We know that our randomness at least was not repeating since every time we would run the same configuration (same number of dimensions and trials), we would get slightly different MST weights which we saw while debugging and printing each MST separately.