CS 124 Programming Assignment 1: Spring 2023

Your name(s) (up to two): Salvador Blanco, James Bardin

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: 3
No. of late days used after including this pset: 5

Homework is due Wednesday Feb. 22 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 submit a single copy of the report, with both of you listed as submitters, in Gradescope.)

We 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. We might advise you not to use a scripting language like Python, although many students successfully do the assignment in Python.

For any algorithms or data structures taught in CS 124 that you wish to use, you must write your own code. For instance, you may not use external libraries for heaps, priority queues, or finding MSTs outright, but you may use external libraries for, e.g., binary search trees, hashing (e.g. built-in Python dictionaries), or routine operations on arrays.

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 goal in this Programming Assignment is to determine, in each of the three cases above, how the expected (average) weight of the minimum spanning tree (not an edge, the whole MST) grows as a function of n.

We give further guidelines on how to get to this goal in the rest of this handout. 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, we 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, we prefer that your code accepts the following command line form:

```
./randmst 0 numpoints numtrials dimension
```

Accepting that command line form is optional. If your code doesn't and we need to test your code, we may need to contact you.

The flag 0 is meant to provide you some flexibility; you may use other values for your own testing, debugging, or extensions. 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, and that "dimension 0" is not actually the 0-dimensional version of the 2-, 3-, and 4-dimensional cases.) 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.

It is convenient for us in grading to be able to run the programs without any special per-student attention. The following three instructions make that easier for us, but don't spend more than a few minutes trying to satisfy them—we can contact you to ask about seeing the code run on your machine if necessary.

- If possible, for compatibility reasons, the code should run on a Unix/Linux system, even if you code on another system.
- We expect the code as described above in its own file(s) separate from your results/writeup.
- 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.

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.

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.)
- 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 your experiments use independently generated randomness!)

In addition, you are expected to discuss your experiments in more depth. This discussion should reflect what you have learned from this assignment. You should, at a minimum, discuss the asymptotic runtime of your algorithm(s) and the correctness of any modifications from the standard algorithms presented in class; otherwise, 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!

Writeup:

For this programming assignment, we decided to use Kruskal's Algorithm. We did not apply any modifications to the algorithm itself, but we did include some extra pre-processing steps which will be elaborated on. Our reasoning behind choosing Kruskal's over Prim's was its simplicity and efficiency. Kruskal's runs in $O(E * log(V))$ time, whereas Prim's runs in $O(V^2)$ time. Prim's however, can be improved to attain $O(E * log(V))$ through optimization. Additionally, the fact that Kruskal's begins with the edge with the smallest weight made intuitive sense for the goal of this MST assignment. The total asymptotic runtime of our implementation takes into account the following processes:

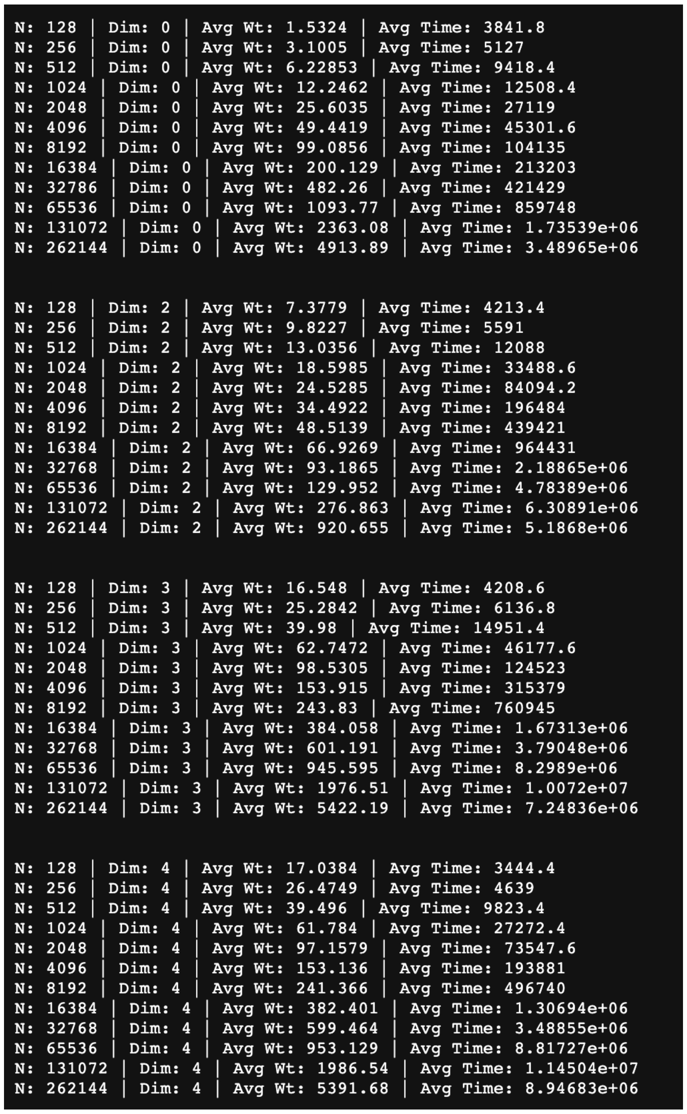
Pre-processing:

We implemented a helper function to determine the threshold for our weights. In other words, if an edge was assigned a weight greater than said threshold, this edge was removed as we could conclude that it will not play a role in our MST. Since we are looking for the spanning tree with the smallest cumulative sum of edge weights, certain values can be ommitted since they will only make this process slower. This function ran in linear time since it was a result of arithmetic operations on scalar values. Additionally, for the 2D/3D/4D cases, we employed a Euclidian distance function which also runs in linear time; this runtime, however, exponentially increases as we run it on large instances of n. The random number generator behaved in a very interesting manner. Since we know that it samples uniformly, we were able to use the idea of a threshold since for very large n, we would have an excess amount of edges which would not play a role in our MST. Our graph generator also ran in linear time due to our created data structure. We simply assign each vertex an id, parent, and rank. Additionally, our edges are created using pre-existing vertices and the randomly generated edge weights. It is important to note that linear time was achieved solely because of our use of parallel processing; if this were not the case, runtimes would have been in the order of $O(n^x)$, where x is the dimension we are handling. This is because of our need for calculating Euclidian distance which requires the use of nested for loops.

Kruskal's Algorithm:

As stated before, Kruskal's algorithm runs in $O(E * log(V))$ time. Since we did not make any modifications to the algorithm itself, it should be bounded by this same value. Our sorting algorithm (from C++'s std library) runs in time $O(n * log(n))$, so this did not play a role is worsening our asymptotic runtime. It is also important to note that our makefile included the -O3 flag. This further optimized our program and ensured that it ran efficiently.

Data:



```
N: 128 | Dim: 0 | Avg Wt: 1.5324 | Avg Time: 3841.8
N: 256 | Dim: 0 | Avg Wt: 3.1005 | Avg Time: 5127
N: 512 | Dim: 0 | Avg Wt: 6.22853 | Avg Time: 9418.4
N: 1024 | Dim: 0 | Avg Wt: 12.2462 | Avg Time: 12508.4
N: 2048 | Dim: 0 | Avg Wt: 25.6035 | Avg Time: 27119
N: 4096 | Dim: 0 | Avg Wt: 49.4419 | Avg Time: 45301.6
N: 8192 | Dim: 0 | Avg Wt: 99.0856 | Avg Time: 104135
N: 16384 | Dim: 0 | Avg Wt: 200.129 | Avg Time: 213203
N: 32786 | Dim: 0 | Avg Wt: 482.26 | Avg Time: 421429
N: 65536 | Dim: 0 | Avg Wt: 1093.77 | Avg Time: 859748
N: 131072 | Dim: 0 | Avg Wt: 2363.08 | Avg Time: 1.73539e+06
N: 262144 | Dim: 0 | Avg Wt: 4913.89 | Avg Time: 3.48965e+06


N: 128 | Dim: 2 | Avg Wt: 7.3779 | Avg Time: 4213.4
N: 256 | Dim: 2 | Avg Wt: 9.8227 | Avg Time: 5591
N: 512 | Dim: 2 | Avg Wt: 13.0356 | Avg Time: 12088
N: 1024 | Dim: 2 | Avg Wt: 18.5985 | Avg Time: 33488.6
N: 2048 | Dim: 2 | Avg Wt: 24.5285 | Avg Time: 84094.2
N: 4096 | Dim: 2 | Avg Wt: 34.4922 | Avg Time: 196484
N: 8192 | Dim: 2 | Avg Wt: 48.5139 | Avg Time: 439421
N: 16384 | Dim: 2 | Avg Wt: 66.9269 | Avg Time: 964431
N: 32768 | Dim: 2 | Avg Wt: 93.1865 | Avg Time: 2.18865e+06
N: 65536 | Dim: 2 | Avg Wt: 129.952 | Avg Time: 4.78389e+06
N: 131072 | Dim: 2 | Avg Wt: 276.863 | Avg Time: 6.30891e+06
N: 262144 | Dim: 2 | Avg Wt: 920.655 | Avg Time: 5.1868e+06


N: 128 | Dim: 3 | Avg Wt: 16.548 | Avg Time: 4208.6
N: 256 | Dim: 3 | Avg Wt: 25.2842 | Avg Time: 6136.8
N: 512 | Dim: 3 | Avg Wt: 39.98 | Avg Time: 14951.4
N: 1024 | Dim: 3 | Avg Wt: 62.7472 | Avg Time: 46177.6
N: 2048 | Dim: 3 | Avg Wt: 98.5305 | Avg Time: 124523
N: 4096 | Dim: 3 | Avg Wt: 153.915 | Avg Time: 315379
N: 8192 | Dim: 3 | Avg Wt: 243.83 | Avg Time: 760945
N: 16384 | Dim: 3 | Avg Wt: 384.058 | Avg Time: 1.67313e+06
N: 32768 | Dim: 3 | Avg Wt: 601.191 | Avg Time: 3.79048e+06
N: 65536 | Dim: 3 | Avg Wt: 945.595 | Avg Time: 8.2989e+06
N: 131072 | Dim: 3 | Avg Wt: 1976.51 | Avg Time: 1.0072e+07
N: 262144 | Dim: 3 | Avg Wt: 5422.19 | Avg Time: 7.24836e+06


N: 128 | Dim: 4 | Avg Wt: 17.0384 | Avg Time: 3444.4
N: 256 | Dim: 4 | Avg Wt: 26.4749 | Avg Time: 4639
N: 512 | Dim: 4 | Avg Wt: 39.496 | Avg Time: 9823.4
N: 1024 | Dim: 4 | Avg Wt: 61.784 | Avg Time: 27272.4
N: 2048 | Dim: 4 | Avg Wt: 97.1579 | Avg Time: 73547.6
N: 4096 | Dim: 4 | Avg Wt: 153.136 | Avg Time: 193881
N: 8192 | Dim: 4 | Avg Wt: 241.366 | Avg Time: 496740
N: 16384 | Dim: 4 | Avg Wt: 382.401 | Avg Time: 1.30694e+06
N: 32768 | Dim: 4 | Avg Wt: 599.464 | Avg Time: 3.48855e+06
N: 65536 | Dim: 4 | Avg Wt: 953.129 | Avg Time: 8.81727e+06
N: 131072 | Dim: 4 | Avg Wt: 1986.54 | Avg Time: 1.14504e+07
N: 262144 | Dim: 4 | Avg Wt: 5391.68 | Avg Time: 8.94683e+06
```

NOTE RUNTIMES ARE IN MICROSECONDS

After plotting our average runtimes as a function of N, we concluded that $f(N)$ lies somewhere in between $y = x^1.33$ and $y = x^1.5$. This function is polynomial in nature, and it makes sense that it would take on this form. Our runtime increases as our number of vertices increases, and the total increase in average runtime is contingent on how many vertices each instance has.