CS124 Programming Assignment 1

Parita Shah, Liya Jin

Due February 23, 2022

No. of late days used on previous psets: 0

No. of late days used after including this pset: 4

Abstract

In this assignment, we attempted to find a function that will determine the average cost, or weight, of a minimum spanning tree for a complete graph whose edge weights are calculated based on varying numbers of points (vertexes) randomly sampled from different dimensions. ($d = 0, 2, 3, 4$) We used C++ in order to implement our version of Prim's algorithm.

1 Implementation methods and modifications

Many of our learnings from the programming assignment are included in this first section, but you'll also find further discussion below our table of results.

1.1 Basic implementation: Prim's

We decided to use Prim's algorithm over Kruskal's because for complete graphs, Prim's yields a slightly better runtime.

Complete graphs are the densest possible undirected graphs: letting $|E|$ and $|V|$ represent the number of edges and vertices in some complete graph, $|E| = |V|^2$. Since the runtime of Kruskal's algorithm is $|E| \log |V|$ and the runtime of Prim's is $|V|^2$, we see that for a complete graph:

$$|E| \log |V| = |V|^2 \log |V| > |V|^2$$

Since Kruskal's runtime is $O(|V|^2 \log |V|)$ for complete graphs, we implemented Prim's which is $O(|V|^2 \log |V|)$.

Overall, we used the basic Prim pseudocode from Lecture 6 as a framework. The modifications we made to Prim's are detailed in section 1.2 below.

1.1.1 Representing the initial graph

We represented the graph as a `numpoints` by `dimension` vector where each "row" represents each vertex's index, each "column" represents one `dimension` of each vertex's coordinate. The contents of each element represent either the distance to that element (in the case where dimension is 1) or its coordinate position along the particular `dimension` indicated by its column (for cases where dimension ≥ 2).

A separate vector, `dist`, keeps track of the total stored cost to a particular vertex, indexed by the row number in our `numpoints` by `dimension` vector. `dist` helps us determine the priority for our priority queue.

Additionally, we treated our undirected graph as a doubly-directed graph where each undirected edge is equivalent to two directed edges of equal weight. For each vertex, once an "incoming" directed edges is found, it will no longer be considered an "outgoing" edge. This allows us to:

1. Ascribe a source and destination for every edge, and
2. Ensure every edge in our resulting MST is only reachable via one path (AKA connected to the tree only once).

1.1.2 Priority queue

We implemented our own `PrioQueue` min heap, complete with functions to retrieve children, parent, insertion, removing the top (minimum) element, changing the priority of a node, and shifting / swapping to rearrange the priority queue.

1.1.3 Implementing Prim's Using our Priority Queue

Essentially, we used C++ Standard Library vectors in order to create different arrays that would work collectively to implement our priority queue.

`dist`: an array indexed by vertex number (each vertex is assigned a number based on it's row in the graph) that keeps track of the minimum weight of an edge incoming that node.

`prioqueue`: a priority queue of vertex numbers that uses `dist` to keep track of the priority of different edges.

`prev`: an array indexed by vertex number that keeps track of the vertex from which the edge is coming.

Once we have instantiated this priority queue, we begin to implement Prim's Algorithm. First, we set all of the distances to be infinity since we do not know the distance to any node. Then we follow the below process:

1. Add a vertex to the priority queue (marking it visited).

2. Calculate the distance from that vertex to every other vertex (computing the Euclidean distance on the fly with `eucl_dist` in the `dimension` ≥ 2 case: see section 1.2.1 below).

3. Check whether there's an incoming stored edge to that vertex, and if there is, check the current value of that stored edge. If that current stored value is less than the threshold cutoff (see section 1.2.2), then we update the priority queue: otherwise, we do nothing.

   If there isn't already a stored value for that vertex, add the edge to the priority queue.

Our `Prims` function, instantiates important variables for edge count, MST cost, whether or not vertices are visited, and other important markers essential for creating the MST. The function then:

1. Iterates through each vertex of the graph, adding the first vertex to the priority queue as well as all of its outgoing edges (represented in the priority queue by the index of the row(vertex) at which the edge is incoming).

2. Picks the vertex with the smallest incoming edge and take it off the priority queue.

3. Adds that edge to a counter for total MST cost

4. Iterates through all outgoing edges from that vertex, updating their "priorities" based on whether or not there exists a smaller edge weight incoming that node. (Since the first iteration of the function already added all the vertices to the priority queue, at any point afterwards, we simply need to update the weight of the edge to that node if we find a different path that is shorter).

5. Repeats, until the number of edges in the MST is equal to $numpoints - 1$.

1.2 Modifications to Prim's

1.2.1 Modification: computing edge weights on the fly

We decided to generate edge weights on the fly in cases where the dimension $= 0$. Thus, when iterating through our "graph" in Prim's at each point, all of the outgoing edges from the current vertex would be randomly generated and stored using an unordered map. This way, we were able to map where the edge was coming from, where it was going, and its weight without needing to precompute these values. We also included a check for whether or not the reverse pair existed in the unordered map–in this case, we did not want to re-generate a random value since these edges are actually undirected (i.e. 2 directed edges both with weight w equals 1 undirected edge with weight w) so rather we use the stored value.

To make our code more memory-efficient, for cases where dimension ≥ 2, we computed the edge weights between each vertex and the vertices to which it is connected "on the fly" (AKA as we explored each vertex, detailed in 1.1.2).

(In the case where dimension = 1, we simply represented the graph by a series of edge weights, since individual vertices had no effect on the MST or edges.)

In Prim's regular algorithm, all of the graph's edge weights are pre-calculated and simply compared as the graph is traversed. However, because we're working with complete graphs, we decided to compute edge weights closer to the time at which they were needed—whether they be randomly generated on $[0, 1]$ or a Euclidean distance on a number of `dimension`s. This prevents us from having to store every edge in the graph at once: since each of n vertices connects $n-1$ edges, this would necessitate storing a total of $\frac{n(n1)}{2}$ edges simultaneously, which takes up abysmally large memory for large n.

This functions the same way as regular Prim's because the edge weights of each random graph remain constant, regardless of when they are computed—we are *computing*, not *calculating*, edge weights on the fly.

1.2.2 Modification: simplifying the graph

To make our code more time-efficient, we wanted to throw out any edge weights that fell above a certain threshold. In geometric terms this threshold would take the form of a radius r (from the point in each dimension that corresponds to $(0, 0)$ in 2D: let us simply call this point 0) for which any randomly-generated graph edge whose Euclidean distance to 0 is greater than the radius is thrown out.

Let us define S as the unit line/square/cube/hypercube from which each edge weight is sampled for ≥ 2 dimensions: this represents a uniform distribution in all dimensions. Each vertex in S represents an edge weight. We want to find some threshold r for which the expected number of edge weights for each vertex whose value (Euclidian distance to 0) is less than r is equal to 1.

We know that

$$E(\textbf{total MST cost}) = n * E(\textbf{1 edge in the MST})$$

We also see that each $E(\textbf{1 edge in the MST})$ can be rewritten as

$$E(\textbf{number of edges in } S <= r) = 1$$

since we only want one edge per vertex to be chosen for our MST after pruning all other connected edges greater than r.

Let us define an indicator random variable I_{jk} which is 1 if the jth edge of some kth vertex in S is less than r, and 0 otherwise. By the fundamental bridge, $E(I_{jk}) = P(\textbf{1 point is } \leq r) = r^d$. where d = number of dimensions. Then by linearity and symmetry, we see that:

$$n * r^d = 1$$

which solving for r, gives us:

$$r = \sqrt[d]{\frac{1}{n}}$$

This is implemented in our code through `threshold_check`, which multiplies the calculated threshold of r by an extra `margin` of 3 before implementing it to ensure that there is probabilistically a near-zero chance that the threshold will completely prune all the edges for a vertex.

Our proof for correctness lies in our code: this threshold r is valid if and only if we end up with an MST with exactly $n-1$ edges. An assert statement in our program makes this check before returning the MST cost: if the number of edges in our MST is *not* $n-1$, the output will not be returned. Therefore, if an MST is returned, we know that there must be at least $n-1$ edges and since each edge connects two unique vertices (via our implementation), we can conclude in this case that our threshold is valid.

1.2.3 Random number generation

In our first attempt, we used `rand()`$/RAND\_MAX$ to generate values between 0 and 1. However, our test results followed a suspiciously patterned "randomization"; we eventually realized that the `rand()` is not truly random. `rand()` can only generate a pre-determined set of "random" values when seeded with an identical value: every time we reseeded our random generator with the same seed, the exact same set of "random" values was generated.

Our solution was to seed our random generation with the `srand` function based on the time the function is called, which changes with every reseeding. We trust our random number generator (for the most part) since each seed is unique and `rand()` for each trial is called using the same seed. Additionally, the seed is only called once before every trial. Thus, the randomly generated values roughly follow similar patterns (thus allowing for the ability to find a function for the average MST cost–especially in the 0D case.) Reseeding the generator more often leads to even more "random" results that seem to not follow the uniform distribution as we want.

2 Average MST size for various n

Average Tree Size				
numpoints	$d = 0$	$d = 2$	$d = 3$	$d = 4$
$n = 128$	0.920026	9.972995	22.367971	35.009117
$n = 256$	0.826699	14.428775	34.920788	58.23415
$n = 512$	0.856207	20.358152	55.404827	95.670273
$n = 1024$	0.845335	28.74696	86.442116	157.657791
$n = 2048$	5	40.178001	135.545349	261.608643
$n = 4096$	545	56.711872	213.425079	434.611725
$n = 8192$	545	79.839188	336.002045	721.532898
$n = 16384$	545	112.509056	531.047363	1204.411377
$n = 32768$	545	159.133835	838.57843	2008.132202
$n = 65536$	545	18744	7560	
$n = 131072$	545	18744	7560	
$n = 262144$	88	788	6344	

Table 1: Average Tree Size for varying n

2.1 Our Data

Below, we've plotted our average MST cost against input size at each of the dimensions from $1, ..., 4$. The functions include a best-fit line, for which we discuss our guess in the next section.

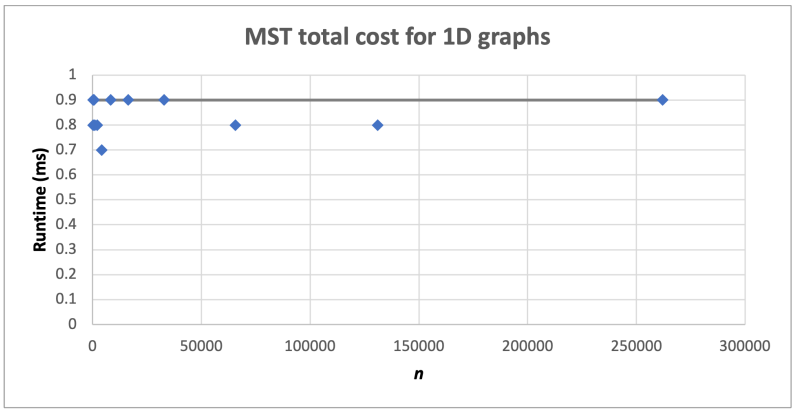

Figure 1: Runtimes and best-fit when n is sampled from 1 dimension.

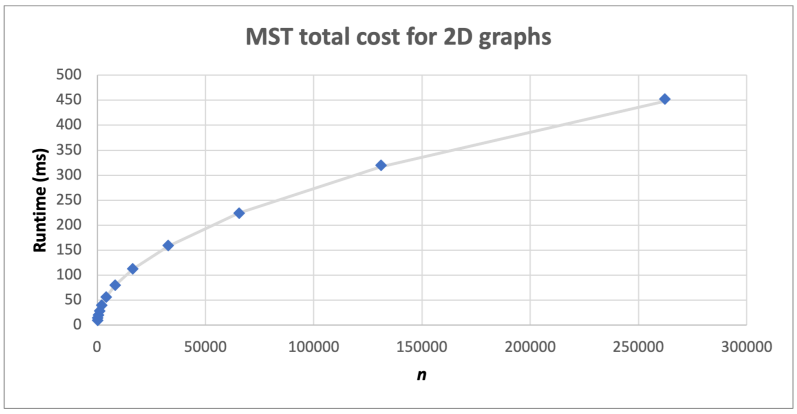

Figure 2: Runtimes and best-fit when n is sampled from 2 dimensions.

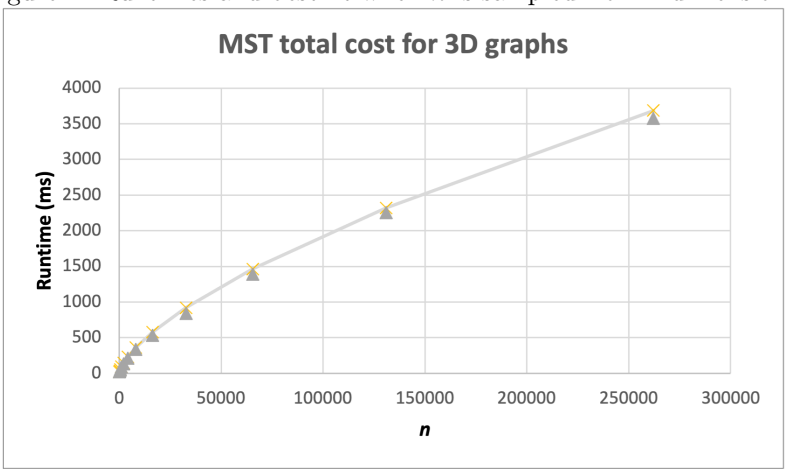

Figure 3: Runtimes and best-fit when n is sampled from 3 dimensions.

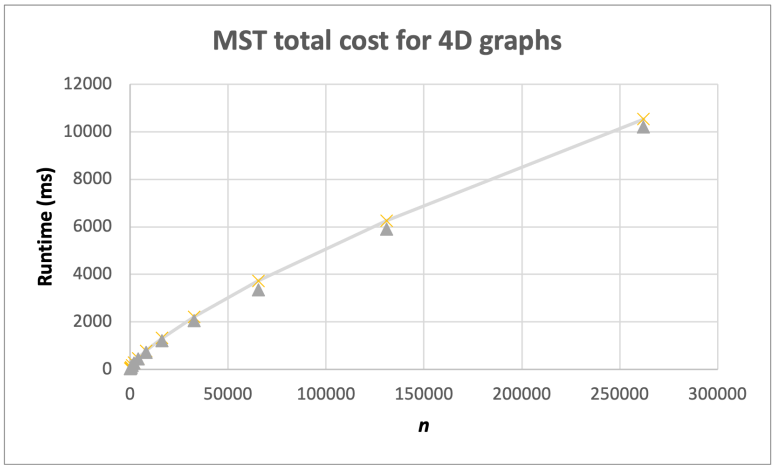

Figure 4: Runtimes and best-fit when n is sampled from 3 dimensions.

3 Discussion of results

3.1 Growth rates (the $f(n)$)

Our guess is that the MST weight grows as a power function of n proportional to the number of dimensions (d) in which its vertices can be generated.

Based on our graphs when plotting n against MST cost, the exponent is likely less than one (the curve follows a root function) and dependent upon d. Any constants other than the exponent seem to interfere only minimally with the function's final value. We began by putting all of our data into an Excel spreadsheet and using best-fit to determine what equation and constants best fit our graph. We realized there must be some multiplicative factor in addition to the power computation so we used a guess and check method where we tried different values until we found one that most closely aligned with the data that we had generated.

We can write more specific guesses, $f_d(n)$, for each dimension d:

$$f_1(n) = \frac{9}{10}n$$

$$f_2(n) = \frac{8}{9}n^{\frac{1}{2}}$$

$$f_3(n) = \frac{9}{10}n^{\frac{2}{3}}$$

$$f_4(n) = \frac{10}{11}n^{\frac{3}{4}}$$

f_1 corresponds to the best-fit function Figure 1 above, f_2 to Figure 2, and so on for f_3 and f_4.

3.2 Runtime, factors affecting it

3.2.1 Runtime Rundown

For the first 5 sizes of n, $(128, ..., 2048)$, our algorithm ran almost instantly. The next three cases $(4096, 8192, 16384)$ collectively took around 15 minutes to run.

The 9th and 10th cases $(32768, 65536)$ took around 1 hour and then a little over 2 hours to run.

The last two cases $(131072, 262144)$ collectively took around 5-6 hours to run.

This generally makes sense: as the input size increases exponentially, as does the time required to run the program. Additionally, as n increases, the number of edges explored increases at a rate of $\frac{n(n-1)}{2}$ due to each graph's completeness, pushing runtime even further upwards. Additionally, as Prim's algorithm was implemented using a minheap priority queue, there was a linear $O(n)$ runtime for building the heap. Essentially, since every time a new weight was found, the heap needed to be "re-heaped" in order to ensure that invariants, such as the parent node always having a smaller weight than the child node,

were being met. Thus, every time `MinHeapify()`was called, the "shift down" or "bubble down" process needed to happen for every set of parent and child nodes beginning from the leaves of the binary tree to the roots, resulting in $O(\log n)$ time. One benefit of using a minimum binary heap approach is that getting the minimum element is $O(1)$ time since the minimum edge weight is always stored at he "0th" index. Additionally, Additionally, the run time for graph generation is linear since we are primarily traversing the 2D array by row number (this is at most, the number of points in the graph.) Similarly, in the 0D case, we traverse the unordered map of pairs, only keeping 1 vertex incoming each node. Thus, traversing in this case is also linear time. Overall, however, our implementation of Prim's algorithm was very slow. This is likely due to the method of keeping values (distances) stored in a list. Even once edges were removed from the priority queue, their values remained in the `dist` vector. Thus, at any one points, in order to determine the priority of an edge, it was necessary to traverse the vector (a linear time operation) before being able to continue.

3.2.2 Graph generation

After looking at the speed to run each part of our code, we saw that the average time in milliseconds taken to generate a random graph (without computing its edges, as that is done on the fly) is negligible and the cases we've seen, essentially zero.

We've learned then that as n grows, the average time for graph generation is entirely dwarfed by the runtime for Prim's. Even as the runtime for Prim's increases drastically, the runtime to generate our graph remains < 0. Note here that this was only possible since our graph generation utilizes an array that only needs to be traversed in one direction. We originally tried to implement this code using an adjacency matrix which led to much higher runtimes for graph generation, and required computing all of the edge weights before hand.

Algorithm Component Runtimes (ms)		
numpoints	Graph Gen	Prim's
$n = 128$	<1	7
$n = 256$	<1	48
$n = 512$	<1	115
$n = 1024$	<1	448
$n = 2048$	<1	1957

Table 2: Average runtimes for generating graphs vs. running Prim's for inputs of size n

Conclusion Ultimately, we learned that the data structure we were using in ordeer to implement Prim's Algorithm had a substantial impact on the results that we got. Initially, attempting to use an IndexedPriorityQueue along with an eager version of Prim's resulted in extremely large runtimes and too much

memory usage since we needed to keep track of multiple different arrays each of which needed to be traversed in linear time. After that, re-implementing using a regular Priority Queue that used an external source for maintaining priority turned out to be much more efficient. We also realized that while other dense graphs may benefit from Prim's algorithm and adjacency matrix representation, finding the MST of complete graphs would actually likely be faster using Kruskal's. This is because a complete undirected graph (when viewed as a directed graph with $2 * edges$) has the same values as weights for each pair of edges. Thus, representation in our adjacency matrix requires double the amount of space that we truly needed to represent our graph. Overall, this assignment really made us think critically about efficiency when designing and implementing algorithms and thinking about how we can improve them in the future.