Minimum Spanning Trees

CS 124 - Sabrina Hu & Smriti Somasundaram

2/24/23

No. of late days used on previous psets: 2

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

1 Algorithm Approach

1.1 Algorithm Choice

To determine how the average weight of the minimum spanning tree grows as a function of n for graphs in dimension 0, 2, 3, and 4, we decided to approach the problem with Prim's Algorithm. Because Prim's Algorithm grows from a starting vertex and adds the next cheapest vertices to create an MST, while Kruskal's grows from the cheapest edge and adds the next cheapest edges to create an MST, Prim's has a better time complexity when dealing with dense graphs, such as the complete graphs presented in this problem. This motivated us to proceed with Prim's Algorithm.

1.2 Algorithm Setup

Before diving into our algorithm, we first constructed a "Node" structure for each vertex and a "Min-Heap" class to create min-heaps out of these nodes, stored in a header file alled "binary_heap.hpp". The Node struct, which are the vertices in our graph, store a few key attributes: ID (int), visited (bool), value (float), and coords (float array). These attributes are used throughout our algorithm implementation, most important of all being the Node "value" — which is essentially the vertex's weight. Creating our Node structure was important because it allowed us to assign the Nodes many important attributes and allowed us to update these attributes as we went. Additionally, using a binary MinHeap was also beneficial because it allowed us to extract the minimum-value node at the beginning of each loop with $O(1)$ time, and allowed us to insert a new node into the heap with $O(\log n)$ time.

1.3 Algorithm & Modifications

For our algorithm itself, we implemented a variation of Prim's, making certain modifications as we went to make our program more efficient. For a given graph with n vertices, we first initialize a vector of n Nodes. We then initialized each node to have the maximum possible float value, set the "visited" attribute to false, and initialized their IDs to be their index in the vector. For dimensions 2, 3, and 4, we randomly generated coordinates for each Node.

We initially tried initializing a graph by generating an adjacency matrix for the graph, and iterating through it. However, this clearly became too costly in space as n got bigger, which led to one of our **biggest modifications**. Instead of initializing a graph, we generated random edges as we went through our program (for dim 0) and for dim 2, 3, and 4, calculated the distances between randomly generated coordinates as we went through out program. Thus, instead of taking $O(n^2)$ space to store an adjacency matrix, we took $O(n)$ space to generate a random edge each time we iterated on a new Node and only store it if it was cheap.

Another modification we made from Prim's is that we did not use a Set to store unvisited nodes. Instead, we made the boolean "visited" an attribute of each Node such that we could check if a Node was visited in $O(1)$ time. Additionally, we did not use a parent point "prev" like in the standard Prim's algorithm, since we were not concerned with the actually MST we were finding as opposed to just the MST weight.

In our algorithm, we start with an arbitrary Node in our graph (initialized with value 0) and add it to our MinHeap. In every loop of our while loop, we extract the minimum node from our MinHeap as the node we "arrive' at, and mark it as visited. Then we search through every other node in the graph that is unvisited. This is when we either randomly "generate" an edge between the node we are at and the node we are looking at, or calculate the distance between the coordinates. If this edge weight is smaller than the value of the node we are looking at, we update the value of that node and add the node into our MinHeap (which helps us update the shortest paths, a form of edge relaxation). Once we finish that loop we go back to the beginnning and once again extract the minimum node from our MinHeap, which guarantees that we "generate" an edge to the Node that is the shortest distance away from our MST (the minimum node). We also update our "minimum distance" value everytime we arrive at a new minimum node. We keep going through this loop until we have no values left in our MinHeap, at which point we would have reached all nodes in our graph and found the weight of the MST.

2 Results

2.1 Table of Average MST Weights

		Dim 0		**Dim 2**		**Dim 3**		**Dim 4**	
n	**tr**	**w**	**t**	**w**	**t**	**w**	**t**	**w**	**t**
128	100	1.19951	0.056	7.60743	0.111	17.5622	0.091	28.4142	0.082
256	100	1.19272	0.118	10.6699	0.29	27.6621	0.214	47.1353	0.2
512	100	1.1993	0.403	14.9744	0.85	43.3709	0.628	78.2315	0.582
1024	100	1.20111	0.949	21.0637	2.908	68.1734	2.023	130.051	1.889
2048	10	1.18649	0.337	29.6973	1.076	107.562	0.739	216.422	0.704
4096	10	1.20609	1.283	41.9178	3.325	169.444	2.494	361.173	2.417
8192	10	1.20264	4.98	59.0523	12.07	266.954	9.513	602.439	9.033
16384	5	1.20172	10.366	83.1577	24.172	422.508	17.516	1010.32	18.177
32768	5	1.20029	37.106	117.455	89.739	668.663	66.563	1687.47	68.941
65536	5	1.20354	148.338	166.044	323.178	1058.57	253.7	2828.58	265.539
131072	5	1.20262	591.963	234.62	1908.123	1677.74	998.468	4740.75	1095.226
262144	5	1.20184	2342.209	331.284	7712.498	2658.81	3964.894	7950.01	4241.588

For clarification, **n** is the number of vertices in the graph, **tr** is the number of trials, **w** is the average weight of the MST in each case, and **t** is the time for the program to run in seconds.

2.2 Estimation of $f(n)$

Using Google Sheets functions, we approximately fit the average weights of the MSTs to a power function of the form $f(n) = An^B$ (with constants A, B), to estimate $f(n)$ for each dimension.

For dimension 0, we see that the average weight of the MSTs gets closer to 1.2 as n as n grows, so we estimated

$$\mathbf{f(n) = 1.2}.$$

For dimension 2, we have

$$\mathbf{f(n) = 0.7n^{\frac{1}{2}}}.$$

For dimension 3, we have

$$\mathbf{f(n) = 0.7n^{\frac{2}{3}}}.$$

For dimension 4, we have

$$\mathbf{f(n) = 0.7n^{\frac{3}{4}}}.$$

Then growth rates for each $f(n)$ are not too surprising, as they are sublinear with respect to n. This means that the average MST weights for these graphs do not grow linearly with the number of vertices in the graph, but at a slower rate. This makes sense because although the average distance between vertices in a complete graph increases as the number of vertices increase, it does at a slower rate, and more vertices in the graph also allows for shorter edges to exist between subsets of certain vertices. Additionally, it makes sense that the growth rates increase as the dimension increases, as higher dimensional graphs have more edges, and thus possibly more variation in edge weights and a larger average MST weight.

3 Discussion

3.1 Asymptotic Runtime

The time complexity of our algorithm is $\mathbf{O(n^2 \log n)}$. There are $\binom{n}{2}$ "edges" in our complete graph, which we normally do not iterate through completely, but we would in the worst case scenario. The $\log n$ part of the time complexity comes from inserting/updating the binary min-heap every time a node value is updated.

In our actual runtimes through running our program, we can see that the runtimes increase exponentially as n increases by powers of 2, which once again demonstrates the $n \log n$ time complexity. I also found it interesting that the dim 2 graphs took longer to run. I'm still not sure why, since it makes more sense for the program to run slower as dimensions increase.

3.2 Cache Size

Cache size of my computer definitely made a difference for this program. The most obvious one was that when I initially tried using an adjacency matrix (which takes up a n^2 space), my computer couldn't take n values past 4096. Additionally, even with the current algorithm, cache size of computer would make a difference; computers with more memory should run faster.

3.3 Random Number Generator

I did not have any problems with our random number generator. Just like any computer program, it is not actually completely random. But for our purposes, I think it did a good job — I also seeded random number on the current time, so that (at least for the longer trials), there would be an increased factor of randomness.

4 Conclusions

To summarize, we used Prim's algorithm with some alterations to implement this problem. We found that for dim 0, the average weight of MSTs as a function of n was about $f(n) = 1.2$, while for dim 2, 3, and 4, the average weight of MSTs as a function of n was about $f(n) = 0.7n^{\frac{1}{2}}$, $f(n) = 0.7n^{\frac{2}{3}}$, and $f(n) = 0.7n^{\frac{3}{4}}$.

We learned a lot in the assignment, especially about the concepts behind MST algorithms. We also learned a lot from trying to make optimizations; it was interesting to figure out what significantly led to more efficiency and what didn't. Some of the biggest challenges included figuring out how to generate our graph after the adjacency matrix did not work for bigger n, and also figuring out how to construct our Node structure for our program to run as easily as possible. Overall, it was a very interesting and rewarding assignment.