Will Long
Javier Cuan-Martinez
CS 124

Programming Assignment #1: Minimum Spanning Trees

The table below gives the average weight of the minimum spanning trees after five executions of our algorithm at at each value of n, where n is the a number of edges in the entire tree:

	Average weight of trees			
n	**Dimension 0**	**Dimension 2**	**Dimension 3**	**Dimension 4**
16	4.844	5.277	6.522	8.246
32	8.098	7.722	12.218	15.123
64	13.972	13.209	18.879	26.203
128	21.186	21.678	34.055	47.730
256	32.139	33.198	58.270	81.465
512	52.529	52.378	97.880	140.415
1024	80.823	80.146	152.802	235.948
2048	134.972	128.668	283.097	424.042
4096	174.361	203.492	458.190	711.857
8192	329.459	312.862	757.439	1266.500
16384	942.41	983.42	1231.618	2107.183
32768				
65536				

Will Long
Javier Cuan-Martinez
CS 124

	Average time (minutes)			
n	**Dimension 0**	**Dimension 2**	**Dimension 3**	**Dimension 4**
16	0	0	0	0
32	0	0	0	0
64	0	0	0	0
128	0	0	0	0
256	.000427	0	0	0
512	.002840	0.003	2.200	2.000
1024	0.025903	0.028	21.000	21.000
2048	0.358511	0.326	0.256	0.250
4096	3.937	3.000	3.374	3.003
8192	34.112	36.654	56.423	41.271
16384	507.014	532.52	611.684	517.368
32768				
65536				

Are the growth rates (the f(n)) surprising? Can you come up with an explanation for them?
The final running time of our algorithm is $O(n^4 + n^2 logn^2)$. Taking our equation into account, the growth rates make complete sense. We will begin by giving a running time analysis of our algorithm. The bulk of our algorithm lies in an encompassing for-loop that iterates through every vertex in our array of vertices, which is $O(n)$. Within that for-loop we call our function prim, which is where the logic for the algorithm is implemented. Within prim, we call a function called find. Our find function iterates with a for-loop through all of the edges to search for the smallest valid edge, which takes $O(n)$. The prim function runs for the number of nodes in the graph and for each node in our queue; this process takes $O(n^2)$ of running time. Putting all of these iterations together, our prim function runs at $O(n^3)$, and since our prim function is inside our encompassing for-loop, this means that the entire process of running prim's algorithm on our data structure is $O(n^4)$.

Will Long
Javier Cuan-Martinez
CS 124

We figured out that the number of edges in a graph is represented by $\frac{n(n-1)}{2}$ where n is the number of nodes. Knowing this, we get that creating our data structure of n vertices and our coordinate array of n vertices will take $O(\frac{n(n-1)}{2})$, which is the same as $O(n^2)$.

Next, we have our merge-sort. We know that merge-sort runs $O(ElogE)$, where E is the number of edges. We discovered that $E = \frac{n(n-1)}{2}$, so we can say that merge-sort runs $O(\frac{n(n-1)}{2}log\frac{n(n-1)}{2})$, which, after using log rules and taking away any unnecessary terms, turns into $O(n^2logn)$.

Thus, at this point our algorithm runs on a running time of $O(n^4 + n^2 + n^2logn)$. Because the n^2 term is negligible, we have a final running time of $O(n^4 + n^2logn^2)$.

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?
Above is a table of how fast our algorithm works in minutes. There might be discrepancies because my partner and I were using two completely different operating systems and hardware. We timed the algorithm five times at each dimension and took the average of the times. The times that contain zero do not really take zero time to run. We hypothesize that the algorithm ran too fast for any difference in time to be calculated, for this reason, the values are zero. We don't start seeing any significant growth rate until a graph size of about 512.

As we began to run the larger values of n, the cache sizes of our computers were increasing at incredible rates. We immediately started noticing our computers slowing down from the large amounts of inputs that they were taking. Our computers also began to heat up as we ran our programs.

Did you have any interesting experiences with the random number generator? Do you trust it?
Using the built in random number library that C has, we used time as the seed by which to generate our random numbers. For the smaller sizes of n, we quickly saw that we wouldn't be able to seed our random numbers every time because the program computed the time so fast that it outputted the same weighted edges every time. We fixed this by only computing the seed once when we were building the data structure. Once this was done, our program outputs new random numbers each time the program is executed.

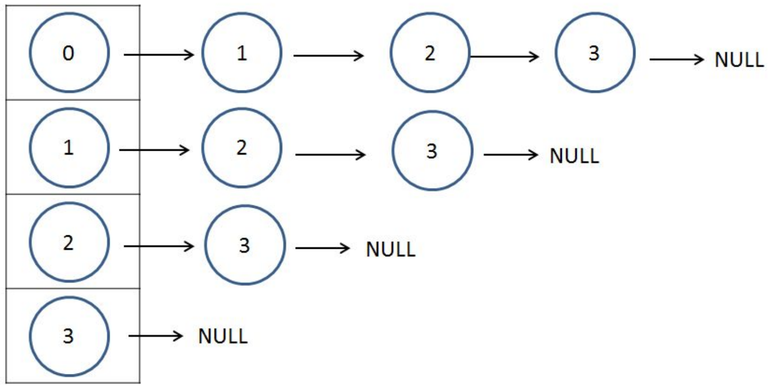
Implementation
We chose to code the assignment in the C programming language because of the freedom that it provides in regards to memory management. Having the ability to manipulate memory to such a high precision allows our program to compute values faster by letting the program know exactly

how much memory is needed to execute each command. This filters any unnecessary memory allocations that script languages like Python tend to overlook.

There are indeed some drawbacks to manually allocating memory while coding in C. Because our memory was not automatically done for us, we had to produce an enormous amount of effort in making sure that we were not accessing memory that was outside of our bounds. Segmentation faults were a common problem throughout our experience with the assignment, and we quickly learned how to spot exactly where our code was producing a segmentation fault using printf() statements and how to handle such situations using pointers.

We have various helper functions in our code that helped us test our program. printList lets us print entire linked-lists into the command prompt. This was extremely helpful in debugging our code, allowing us to see which nodes were being added correctly into the queue and which nodes were correctly being marches as searched. Push is another function that allowed us to enqueue nodes.

Data Structure:
The data structure that we decided to utilize is a modified version of an array with linked lists at each index. Each index of the array from zero to n (where n is the number of nodes in our graph) contains a distinct node from our graph. The nodes in the array each point to their respective linked list. The linked lists consist of nodes that the node in the array is connected to. Because we are working with complete graphs, each node is connected to all of the other nodes. In order to ensure that we don't account for the same edge in our minimum spanning tree twice, we make sure that we don't have the same combo of nodes in different linked lists. For example, with a graph with four nodes, our data structure would look like the following (the numbers represent the name of the nodes, not the weight of the edges):



We designed a struct node to contain four value fields: name, weight, searched, and next. Each node in our array is passed a number from zero to n into their 'name' field. Making their name an integer value makes it easier for us to compare two nodes and name them (it's harder to name a node in something like characters because there are only 26 letters in the alphabet). The 'weight' field represents the weight of the edge between the node and its respective node in the array (nodes that are in the array have a weight of zero, since the distance between it and itself is zero). In order to keep track of which edges we have already chosen, we have a boolean value called 'searched'. The 'next' field is just a pointer to the next node in the linked list.

Our array of linked lists is modified because we chose to use a merge-sort algorithm to sort the linked lists in increasing order with respect to their weights. This allows the smallest weighted edge to always be the first node in its respective linked list.

Algorithm:

We chose to use Prim's algorithm because it works cohesively with the type of data structure that we decided to use. The fact that we can choose the smallest weighted edge at each linked list in constant time (the merge-sort implementation lets us to do this) allows us to execute Prim's algorithm with optimal time. We can look up the vertices in our array in constant time and traverse its linked list of edges in linear time.

We will abide to the rules that Prim's algorithm provides, which entail always searching for the smallest weighted edge that is adjacent to the current spanning tree. Our algorithm works in the following way: Once the user inputs the value for the number of edges in the graph, our data structure that represents the graph is constructed in the manner that was previously explained in the Data Structure section.

We start at index zero of the array and initialize a variable LeastValueEdge to the weight of the first node in the first linked list. We then traverse through the linked list to see which node has a value of false in the 'searched' field. Once we find a node with this criteria, we place a new node with the exact same credentials as the node that we just visited into a queue that will contain all of the nodes whose linked lists we have visited but have not yet depleted. We then set the 'searched' value of the node we just visited to 'true'. If we ever traverse a linked list whose nodes all have a 'searched' field of 'true', then we delete the node in the queue with the credentials that are the same as the node in the array whose linked list has no more edges that we can search.

We then iterate through the queue and search through each of the linked lists of each of the indexes in the array that the nodes in the queue correspond to. We then compare the weights of each of the nodes in each linked list of the nodes that are in the queue to see which one is the smallest. This is the same as looking through a graph and seeing which adjacent edge to the current MST is the smallest. Once we have found the smallest one, we set LeastValueEdge equal to this number and update our result value. We continue to do this until there are no more nodes in the queue. We then return the sum of all of the LeastValueEdge values that we have accumulated.