April 15th, 2020

Matan Segal

CS 3353

PA03 – Analysis Paper

For my trivial implementation of the disjoint set data structure using set of linked lists, I used an outer linked list, which each node contains a linked list of the vertexes.

- The Find() function has a outer for loop which went through the outer list and inner for loop for the list of the vertexes. If it finds the vertex, it returns its location in the outer loop (the location of its list), if not, it will return -1.
- The Union() function will accept two int which represent the location of the two lists where the vertexes in. First, it will check which is the larger list, then merge the smaller to the larger one. Lastly, it will erase the smaller list. Another version of it is unionIdx() which will add a vertex to an existing list. It will accept int which is the location of the list and the vertex which we need to add.
- The makeSet() function will create a list contain the two vertexes and insert it to the outer list.

For the more sophisticated implementation of a disjoint set, I used a set of trees, or "Forest of Trees". The implementation based on a node structure. Each node contains the vertex name, pointer to the parent, rank which is the height of the tree if it is the root, number of element in this tree (if it is the root), and the location in the array. When initialize the disjoint tree, it will allocate an array of pointer to node. Each vertex gets its location in the array according to its alphabetically order. All the pointers initialized to null. The disjoint list has a handler class - disjointTreeIndex, which has a function to convert the vertexes to their location in the array.

- The Find() function will receive int which is the location of the vertex in the array. If it is null, it will return -1, like before, for not finding. If it is not null, it will look to at its parent. Since the root node parent's is itself, it will go through a while look until the pointer equal to the parent, then it returns the location in the array. For future efficiency, if the while loop need to go trough several iterators, when it finds the root, it changes the parent of the node to the root. In this way, next time we will search for this vertex, it will find its root immediately.
- The Union() function will receive two int which represent the location of the two vertexes in the array. First, it will check for the root with the smaller rank and add its tree to the other one by assigning the smaller rank root's parent to the higher rank root. In this way we make sure that the union add to the tree width and not to its depth, which will keep our searching faster. Also here, I have a version for unionIdx() which will allocate memory for a new node contains the new vertex and assign its parent to the root of the tree.
- The makeSet() function will allocate memory for two new nodes which will contain the new vertexes. It will assign one of them to be a root and the other's parent to the first node.

My Kruskal's algorithm implementation function accept the multiset with all the edges of the graph. Since it is a set, it will sort them in ascending order based on their weight. It will go through a while loop until the disjoint set will have one inner list/tree and the number of vertexes it contains it equal to the number of vertexes in the given graph (I could also do the loop until the number of edges in the disjoint set is one less the number of vertexes, but I chose the first option because it should also check my implementation for both disjoint set structures). First the loop will get the edge from the multiset, get the two vertexes, and will send both thorough the Find() function. If both -1, it will call the makeSet() function. If one is not -1 and the other is, it will use the unionIdx(). If both not -1 and different from each other, it will call the Union() function. If both find calls will return the same number, it will skip it since it means that both vertexes in the same list/tree. At the end of the loop, it will increment the iterator to the next edge in the multiset.

For testing my program, I used five different number of vertexes 10,50,100,200,500. For each number, I had two different densities – low density and high density, which gave me ten different graphs. For example, the graph with the 200 nodes, will have a version of random number between 1 to 15 of edges for each node and a second version with random number between 15 to 30 of edges for each node. The number of the weight for each edge also was given randomly numbers between 1 to 100. All the ten files are in the folder inputFiles. For each graph I have an output file, which will be in outputFiles directory and each have the MST result from the tree and the list implementation of disjoint set.

After running the program several times, using all the files and recording the time it takes for each graph going through the Kruskal's algorithm for both the tree implementation and the list one, I came up with the following results:

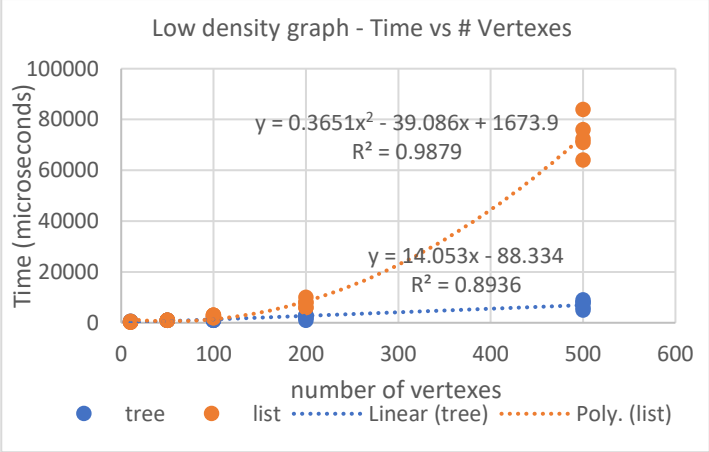

Figure 1 – Low density – Time vs. # vertexes

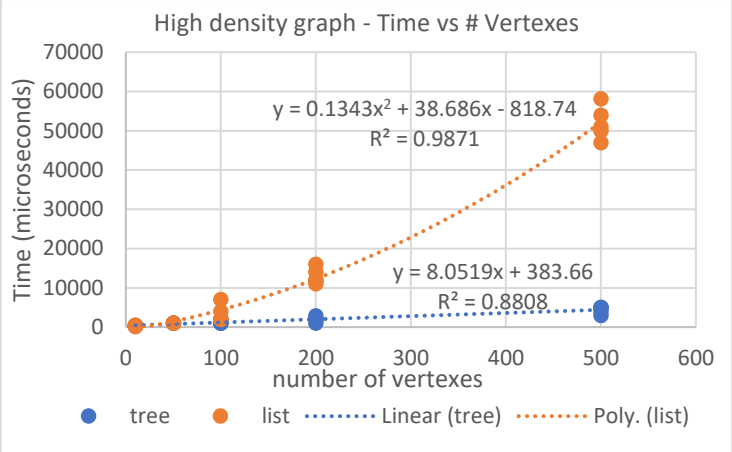

Figure 2 – High density – Time vs. # vertexes

We can clearly see that the "Forest of Trees" implementation of disjoint set is way more time efficient than the set of link list implementation. In both graphs, the low density and the high density, we see that the tree implementation time grows linearly as the number of vertex increase, which referring to a big O(n), while the set of link lists implementation time grows fit very good polynomial line from second degree, gives it a big $O(n^2)$. since the find() function of the set of lists need to have two for loops, one trough the outer list and the second trough the inner one, we get the n times n run time. However, the "Forest of Trees" implementation "skips" the outer loop by having the referring array which send me directly to the tree I need, leave me with only one loop of finding the root of the tree. In addition, as we increasing the number of the vertex the tree does not get much deeper since each time I search for a vertex it will change its parent to the root of the tree and make it wider rather than deeper, which make the number of iterations always minimal.

Since the times in the graph do not include the time it took to read from the graph files, we see that for the set of list implementation it actually was a little bit faster for higher density graphs. My guess, the reason for that is since we have more variety of edges, we get more options with lower weight getting to other vertexes, which causes us searching less for the right edge and completing the MST. In the "Forest of trees" graphs, we see very similar result, but it probably will be the case once we will have higher number of vertexes in the graph.