IN3200 Home exam

candiate number: 15720

25.03.2021

read-graph-from-file1:

The idea and algorithm is pretty simple. We skip the lines that don't contain the information we are after, and store the number of nodes and edges as integers in the function.
We use the number of nodes (N) to allocate a 2D array with size NxN, and set each element to '0'. Then we read through the edges in the list, and while doing so we store each edge in the 2D matrix twice. This is because every node has its own row and column, so each edge unique edge will appear twice in the matrix. During the algorithm we also test for edges between equal nodes, or other illegal edges, since we don't want to store those. Since we use the number of legal edges, found in line 3 of the list, to loop through the list, we have to subtract 1 (we write this as i–) for each illegal node we run through, so that the list will finish, and then we continue to read the next line.
After reading through the file and making the 2D array, we then set the variable table2D as a pointer to the 2D array we created, so that it can exist outside the function. We also do the same with N as number of nodes.

Note: Time usage here both relies on the amount of edges and on the amount of nodes (since we read through the file once, and make a NxN matrix).

read-graph-from-file2:

The idea behind this function is very similar to read graph from file1. We start by reading through the file and storing number of nodes and edges as previous function did. Then we allocate two 1D arrays, temp row and temp col, where row has size N + 1, where N is number of nodes, and col has size 2 * number of edges.
We start by finding the row idx values by reading through the list and for every node found we add 1 to the index value corresponding to the node.
Then to make the list give us the index corresponding to the nodes, as row ptr is supposed to do, we loop over the array and add the previous element to the current element, effectively making each index a sum of the number of edges up until that index value.

Then the tricky part of this function is to make the col idx array:
We start by making another temporary 1D array called count, which has same shape as row ptr, and is used to keep track over where we should place new elements in col idx, since col idx effectively is an array containing multiple lists, where each list represents a unique node, and contains data of which node it has edges with.
The algorithm works by reading a node (from node), using row ptr to find out the index (the list) that belongs to that node and adding the amount

of nodes already placed in that list (found from count[from-node]) to the index value, and then setting that index value as the node which it has an edge with (the to-node), and then doing the same but swapping to-node and from-node, and finish by adding 1 in the count of to-node and from-node. After that i implemented a sorting algorithm to sort while we add values to the lists, which in short checks if the newly added element (node) is smaller than the previous one, and if it is, uses a simple swap function to swap the values of these two nodes, and then keeps checking until either it is at the back of the list, or until it finds a node that it is not smaller than, which will break the loop. Therefore the runtime of this sorting algorithm at max will loop through the individual list once, and at min it will break immediately. This implementation is much faster than sorting after the list is done, and sorting the list is smart because it will save a lot of time in later exercise, but will also make the col-idx more readable.

Note: We also use the same test as read-graph-from-file1 to check for illegal nodes. Time usage here is only reliant on the amount of edges, and not really as much the amount of nodes as the 2Dtable version was, since the heaviest operation is to make col-idx, which has length 2 * number of edges.

create-SNN-graph1:

This function wants to go through the table2D, which we made in read-from-graph1, and for every edge between two nodes, it wants to check how many common nodes these two nodes share edges with.

Therefore we start by making a 2D array with size NxN, and setting each element to 0. Then we loop over the table2D:

Note: (i and j in the loops are the same as from-node and to-node)

The outer loop represents the nodes row, from node 0 to node N-2 (we can exclude the last node, which is node N-1, since every other node will have done their calculations on that node).

The inner loop represents the column-nodes in the 2D array. Here the idea is that for each node we do calculations on, we can skip calculating this exact node further, so therefore looping over the columns starts at j=i. We can also skip the diagonal, since we don't count an edge between two identical nodes, so the entire diagonal is 0, therefore we end up with a loop from j=i+1 to j¡N.

While looping over, we check for each i and j if there is an edge between node i and node j. If we find an edge, we want to check how many common edges they have, so we loop over the i-th row and the j-th row, and count the amount of edges shared. Then we store this count in the matrix on the index [i][j] and [j][i], since those represent the connection between node i and j.

Note: The speed-up from looping j from i+1 instead of i is effectively a double speed-up, and is a bit more than double if the matrix is small, since we cut out more than half the matrix, since we exclude everything on the left of and including the diagonal, and also because we cut out the last row by setting the upper limit of i as N-1.

Also openmp parallelization works perfectly here, however we have to set a few privates, such as column element j, the k element when finding similar edges, and the count. This is to avoid multiple threads overlapping each others work, and getting faulty answers.

So summarized, the speed-up effectively doubles with the smart indexing, which in my case, running on facebook-combined.txt, made the function go from using 2.8 seconds to 1.4 seconds, and by using OMP with 8 threads, i got the speed all the way down to 0.45, which in total is a 6.2 times speed-up!

create-SNN-graph2:

In this function we want to use the col-idx and row-ptr, which were calculated in read-graph-from-file2, to make a SNN-graph on the CRS format (same format as col-idx and row-ptr, where row-ptr is the same as before). Therefore we start by allocating a 1D array with lenght 2 * number of edges, so same as col-idx.

We then want to start the outer loop by looping over every Node, from 0 to N-1 (as in create-SNN-graph1), and we call this variable from-node. Then we want to loop over the list in col-idx corresponding to the current from-node, which is done by using the row-ptr, and we call this variable j.

While looping over each sub-list in col-idx, we check if the node has already been worked on before or not, which is easily done by testing if the j-th node in col-idx is greater than from-node. If the node passes this requirement, then the actual algorithm starts.

We start then by setting count to 0, and finding the value of the node we are working with, which we call to-node. Then we do a simple while loop for the variables k and z, such that k starts at the beginning of the index to from-node, and same with z just with to-node. We make it so that the while loop breaks if k or z exceeds their upper index value, which means that if they go outside of the lists to the node they represent, the loop breaks.

We then start by checking if the node in to-nodes list equals from-node, in which we mark this index to use later.
Then we do a simple if, else if, else test, where first test checks if the node in the list to from-node is smaller than the node in the list of to-node, in which case we add one to k. If its the opposite case, we add 1 to z. And if neither test is true, that means the nodes are equal, and we add both to k, z and count.
In simpler terms, what we are doing is effectively using the fact that the list is sorted to our advantage, by checking which list has the smallest element, and then keep advancing the list with the smallest element. Repeating this process will eventually loop through both lists, and by counting every time an element in both lists are equal to each other, we find the amount of common edges between from-node and to-node. Then we set the value we found from count in the arrays index for j, which is where to-node is found in from-nodes list, and the variable mark, which we found earlier in the while loop, which is from-nodes index in to-nodes list (Where we need to remember that these sub-lists exist in col-idx). So the result of this algorithm is that we created a SNN-graph with same format as col-idx.

Notes: The speed-up here comes down to two parts of the algorithm, namely the test to check if the node has already been worked on, which reduces amount of work down to half. Also the while loop when checking for similar edges in from-node and to-node. If the lists were not sorted, the worst case scenario here would be to do a double for-loop, which would increase the time usage immensely. So since we use the fact that the lists are sorted, we get an insane speed-up from only needing a single loop to go through the lists.

openmp parallelization also works great here, but we have to set the variables j,k,z, count, mark and to-node as private, to avoid any overlapping calculations.

So as to summarize, we get a double speed-up from skipping nodes already calculated, and an immense speed-up from using a single while-loop instead of a double for-loop. Running on my own computer, with OMP the runtime was about 0.03 seconds for facebook-combined.txt, which compared to the speed of table2D without OMP and without smart indexing, is a 93 times speed-up! I
It is also good to note that the amount of edges in facebook-combined was 88234, and amount of nodes was 4039, which means storing SNN in table2D format would require both the normal table2D and SNN-table2D, which is $2*N^2 = 2*4039^2 = 32.627.042$ elements, while storing the SNN values as CRS format, including col-idx and row-ptr, would be $2*2*numberofedges+N+1 = 4*88234+4040 = 356.976$. So the amount of elements needed to be stored using CRS format in this case is 91 times less, and the computation

time (using best results for both), is about 15 times faster!