Counting shared nearest neighbors
IN4200 Home Exam 1

Candidate number 15835
(Dated: March 30, 2021)

I. IMPLEMENTATION

In this note i will explain some of the rationale behind my design of the core functionalities of the program. In addition to the functions described here, there are also a few extra utilities present in the `src/` subdirectory. These are minor utility-type functions, and their usage should be quite obvious from their naming as well as the documentation in their docstrings. Thus, for the sake of brevity i will not go into detail on these here.

Also, as a courtesy to the examiner i have included output of my program ran on both my own laptop, as well as one of the IFI shared machines in the text files `out_m1.txt` and `out_ifi_aftur.txt` located in the root directory.

A. Reading a connectivity graph from file

1. *read_graph_from_file1: 2D table approach*

The main consideration for this function is that each entry into the `table2D` array that is read from the file is to be duplicated such that the array is symmetric. That is, for each 1 assigned to indices (i, j), we assign another 1 to indices (j, i). I was initially suspicious that this symmetry may have been redundant, and that we could for example store only the upper-half of the array, but as we will see later symmetry can be exploited to compute the SNN graphs with efficient stride-1 access to the array as opposed to stride-N, discussed further in section I B.

It is also worth to note that the table is ensured to be stored contiguously in memory using the functions implemented in `char2d.c`.

2. *read_graph_from_file2: CRS approach*

In order to store the graph in the CRS format, we read through the file line-by-line, this time storing the `FromNodeID` and `ToNodeID` pairs (and again, their symmetric counterparts) in two separate arrays of length $2N_{edges}$. Given that the number of legal edges may be $< 2N_{edges}$, we also keep a counter `c` that is incremented twice per line in the file. Prior to reading the file, we also initialize the `row_ptr` array with zeroes, incrementing the `row_ptr[FromNodeId+1]` and `row_ptr[ToNodeId+1]` entries[1] by 1 for each line.

After the entirety of the file has been read, we run a cumulative sum in the entries of `row_ptr`. That is, we for $i = 1, \ldots, N+1$ let `row_ptr[i] += row_ptr[i-1]` meaning that an index $i \in 0, N-1$ in points to the number at which the $i-th$ row in `col_idx` begins, and Nth entry, the last entry in `col_idx`. So for example; if i wish to find the entries in `col_idx` corresponding the the 3rd row[2] in the matrix, that is found in indices `row_ptr[3]` to `row_ptr[4]` in `col_idx`.

Lastly, each row-wise subsection of `col_idx` is sorted individually in ascending order as required by chosen algorithm for computing the SNN graph.

B. Creating an SNN graph

We wish to compute an $[N \times N]$ matrix S representing the number of shared nearest neighbors (SNN) between each connected pair u, v in a graph using the corresponding connectivity graph, represented by an $[N \times N]$ matrix C. To understand how we may construct S, we consider the example graph defined by Figure 1 in the problem text. For this graph we have the connectivity graph

$$C = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{1}$$

From the same figure, we also deduce that the SNN graph may be written as

$$S = \begin{bmatrix} 0 & 2 & 2 & 2 & 0 \\ 2 & 0 & 2 & 2 & 0 \\ 2 & 2 & 0 & 3 & 1 \\ 2 & 2 & 3 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{2}$$

In order to deduce how we may construct S from C, we start by considering how to find S_{01}, the number of shared neighbors between nodes 0 and 1.

---

[1] Again, because of symmetry.
[2] counting from 0

After looking long and hard at C, we observe that the product between the 0th row and the 1st column in C yields 2. That is;

$$\sum_{k=0}^{N-1} C_{0,k}C_{k,1} = 2 \tag{3}$$

By computing a few more entries in S, we may quickly convince ourselves that this (almost) generalizes to

$$\sum_{k=0}^{N-1} C_{ik}C_{kj} \tag{4}$$

failing only in the sense that this sum also finds the shared neighbors between nodes that are not directly connected. It also does not produce a non-zero diagonal. We can remedy this by zeroing out all these non-neighboring entries with C_{ij} like

$$C_{ij} \sum_{k=0}^{N-1} C_{ik}C_{kj} \tag{5}$$

We also recall that the connectivity graph is symmetric, meaning that $C_{kj} = C_{jk}$; so we can rewrite this computation slightly as

$$S_{ij} = C_{ij} \sum_{k=0}^{N-1} C_{ik}C_{jk} \tag{6}$$

which facilitates stride-1 access when summing over k for row-major storage, like we have in the C language[3].

Having outlined the basic computation that needs to be performed, we may then further optimize it to suit each of the two representations of the graph.

1. *create_SNN_graph1: 2D table approach*

The SNN graph may be computed by Eqn. 6 in a simple way by directly looping over indices

$$i, j, k = \{0, \dots, N-1\}$$

yielding a time complexity of $\mathcal{O}(N^3)$. However, given that we know that the all the diagonal elements $S_{ii} = 0$ and that the matrix is symmetric $S_{ij} = S_{ji}$, we can limit ourselves to only looping above the diagonal. That is;

$$i = \{0, \dots N-1\}, \;\; j = \{i+1, \dots N-1\}$$

and for each of these indices (i, j) check if C_{ij} is non-zero, and only compute the sum (Eqn. 6) if that is the case. We then for each computed (i, j) copy the same value to the corresponding indices (j, i) to retain symmetry.

As is perhaps more clearly observed from the code itself, the loops over the indices i, j are, as they say; embarrassingly parallel. That is, any mutations done for any pair of indices (i, j) are entirely thread safe. However, to minimize loop-overhead we opt to only run the outer loop over i in parallel. We also take care to ensure that the workload is distributed in a sequential order due to the fact that there is an increased load for large i. That is; for each i we loop over $N-(i+1)$ number of indices of j. As such, we use a dynamic schedule to distribute the workload among the threads rather than a static one; where the threads would end up with an uneven distribution of the total workload, meaning that the threads obtaining the lighter workloads would sit idle while there are still free computations waiting in queue to be performed.

Lastly, i should note that while i did create a separate serial and parallel version of the function, these are essentially identical, and could be easily merged to one using `#ifdef _OPENMP` to exclude out the `omp.h` include as well as the omp pragma if the program were to be compiled without linking OpenMP. However, i deemed this impractical for the purposes of this exam; as keeping them separate was helpful for my initial development of the program, as well as subsequent unit testing.

2. *create_SNN_graph2: CRS approach*

For the CRS approach, we again utilize Eqn. 6, but in a somewhat different way. Given that none of the zeros in the connectivity graph is included in the CRS format, computing the matrix elements (i, j) in the SNN graph. The problem can instead be thought of as counting the number of common elements in the two sub-arrays of `col_idx` corresponding to rows i and j, which is exactly equal to the summing $C_{ik}C_{jk}$ over k as all elements in the connectivity graphs are either 0 or 1, where only the latter are actually stored in `col_idx`.

To compute the number of shared elements in two subsections of `col_idx` corresponding to two different rows, we define two temporary variables a, b initialized as the indices pointing to the start of the i and j-th row in `col_idx` respectively. We then check if

- `col_idx[a] == col_idx[b]`: In which case we have found a match, and either a or b can be incremented by 1.
- `col_idx[a] < col_idx[b]`: No match, and a is incremented by 1
- `col_idx[a] > col_idx[b]`: No match, and b is incremented by 1

which is repeated until either a or b reached the value corresponding to the end of the current row.

As we can see, this algorithm relies on the symmetric entries in the graph being saved, as well as each of the row sub-arrays `col_idx` being sorted in ascending order, thus

[3] not to be comfused with the C matrix.

justifying the price that is paid to sort the sub-arrays, and all the "duplicate" symmetric entries.

Again, parallelizing this algorithm is entirely trivial as the outer loop is entirely thread safe with respect to `SNN_val`, but again, a serial and parallel version were created for the same reasons as before.

C. Checking Whether a node can be in a cluster

In order to check if a node is part of a cluster, and to seek out all of the connected nodes for a given tolerance τ, we define two char arrays of length N; `in_cluster` and `checked`. Which as the names suggest keeps track of the nodes which are confirmed to be inside of the cluster, and which nodes have been checked for neighbors to add to the cluster.

We also define the function `find_nodes`, which loops through the range of indices i in `SNN_val` corresponding to the `node_id`-th row in the matrix. For each index i, we then check if

- `SNN_val[i]` $\geq \tau$
- `!in_cluster[col_idx[i]]`

if both of these are true, set `in_cluster[col_idx[i]] = 1`, adding the node to the current cluster. However, note that the same index in the `checked` will still be zero; meaning that we will later have to check if the same node that was just added to the cluster also has some neighbors that can be added to the cluster.

After `find_nodes` has finished running for a given node id, we loop through `in_cluster` and `checked` in order to determine if there are any nodes confirmed to be in the cluster that has not had its neighbors checked. If any such nodes are found, we run `find_nodes` with the corresponding node id, if not, we have found all the nodes in the cluster and their indices are printed using the information stored in the `in_cluster` array.

II. TESTING

In order to ensure correct functionality of my programs, i wrote a set of unit tests in `tests.c` drawing on the "simple-graph" example, for which the connectivity graph in both representations is provided in the problem set. In addition, i also computed the corresponding SNN graphs by hand in both the table and CRS representations. I used these expected results to ensure the correct functionality of `read_graph_from_file1`, `read_graph_from_file1`, `create_SNN_graph1` and `crate_SNN_graph2`.

I also created an additional test for the larger facebook dataset, for which no ground truth is provided. As such, i instead tested for self-consistency between the table and CRS representations. That is; i imported the connectivity graph and computed the SNN graph in both representations and checked that they produced equivalent results.

For further details of the unit tests, i refer the reader to the `tests.c` program, which is well commented and should give more insight to how the tests work.

Additionally, i also performed a "manual" test of the `check_node` function. That is; i visually inspected its output of running the function for $\tau = 1, 2, 3$ on all of the nodes in the graph defined by `simple-graph.txt`. In doing so, i confirmed that the expected results provided in the problem set were reproduced. Furthermore, for $\tau = 1$ and `node_id`=0 and 2, i observed that node 4 was connected to the cluster; meaning that the function is able to add nodes which are not directly connected to the input `node_id` to the cluster, and is able to seek out its neighbors neighbors, as it should.

III. PERFORMANCE BENCHMARKS

Benchmarks of each function, with the exception of `check_node`, is provided Table I. The benchmarks were ran on my Macbook air with the M1 chip, compiled with a native[4] version of GCC 10. A noteworthy quirk of this particular processor is that 4 of its 8 total CPU cores are low-power, low-performance cores. And so running the program with more than 4 cores is expected to yield some diminishing performance because of the processor. The benchmarks were performed using the `facebook_combined.txt` dataset, and as we can see in table I the performance drop as the number of threads is increased. I suspect this is due to the overhead associated with spawning new threads, and distributing the data, outweighs the performance gain of running the computation in parallel due to the relatively small size of the graph. Presumably, for a sufficiently large graph the cost of spawning new threads may be considered negligible, and the parallel code would run faster, and scale with the number of threads.

[4] i.e not through Rosetta; a compatibility layer that emulates x86 binaries to the ARM architecture native to this processor

TABLE I: Benchmarks (Macbook air, M1)

	Average runtime of function			
Number of Threads	read_graph_from_file1	read_graph_from_file2	create_SNN_graph1	create_SNN_grahp2
1	12.8 ms	7.9 ms	124.2 ms	88.9 ms
2	-	-	127.3 ms	92.0 ms
3	-	-	131.5 ms	92.1 ms
4	-	-	131.6 ms	92.9 ms
5	-	-	148.3 ms	96.5 ms
6	-	-	162.6 ms	99.0 ms
7	-	-	174.4 ms	101.1 ms
8	-	-	184.7 ms	101.4 ms