Counting shared nearest neighbors

IN3200/IN4200 Home Exam 1, Spring 2021

Note: Each student should independently do the programming and write her/his own short note that accompanies the code. The detailed submission information can be found at the end of this document.

1 Introduction

1.1 Connectivity graph

In many branches of data science, a *connectivity graph* can be used to show how the data objects are directly connected with each other. Each data object is represented by a node in such a graph, and an edge connecting a pair of nodes means that the two nodes are *nearest neighbors* (that is, directly connected). We consider all the edges as *unweighted* and *"two-way"*, that is, if node v is a nearest neighbor of u, then node u is a nearest neighbor of v. An example of a connectivity graph can be see in Figure 1.

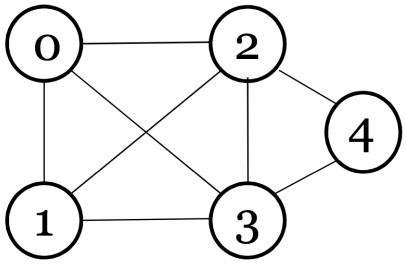

Figure 1: An example connectivity graph involving five nodes and eight edges.

1.2 Data storage formats for connectivity graph

There are two commonly used ways of storing a connectivity graph in program code. The first way, which is very convenient but memory-usage unfriendly, is to use a *2D table* (a 2D array of `char` values). The dimension of the 2D table is $N \times N$, where N denotes the number of nodes in the graph. The values in the table are either 0 or 1. Specifically, if on row i and column j the value is 1, it means nodes i and j form a nearest neighbor pair. Note that the 2D table is symmetric, that is, `table2D[i][j]` always equals `table2D[j][i]`, and by convention we have `table2D[i][i]` equals

0 (a node cannot be a nearest neighbor of itself). The 2D table corresponding to the connectivity graph in Figure 1 is as follows:

row 0	0	1	1	1	0
row 1	1	0	1	1	0
row 2	1	1	0	1	1
row 3	1	1	1	0	1
row 4	0	0	1	1	0

It can be noticed that the number of values of 1 in the 2D table is twice the number of edges in the connectivity graph (see Figure 1).

The second strategy, named *compressed rows*, is more memory efficient for the cases where the number of edges N_{edges} is smaller than N^2 by orders of magnitude. (For comparison, the corresponding 2D table will mostly have values of 0.) Specifically, the compressed row storage (CRS) format for a connectivity graph will use two 1D arrays of integer values: `row_ptr` and `col_idx`. The array `col_idx` is of length $2N_{\text{edges}}$, it consecutively stores for all the nodes the indices of their nearest neighbors. The length of array `row_ptr` is $N+1$, it is for "dissecting" the array `col_idx` with respect to the different nodes. For more information about CRS, please read Section 3.6.1 of the textbook. Note: There is a typo in the textbook about the length of `row_ptr`, which should be $N+1$ (not N). The CRS format for the connectivity graph in Figure 1 will be as follows:

`row_ptr`: 0, 3, 6, 10, 14, 16

`col_idx`: 1, 2, 3, 0, 2, 3, 0, 1, 3, 4, 0, 1, 2, 4, 2, 3

1.3 Number of shared nearest neighbors

One measure of "similarity" between two directly connected nodes u and v is the number of their shared nearest neighbors (SNNs). That is, how many other nodes are directly connected with both u and v? For the simple connectivity graph in Figure 1, the numbers of SNNs are depicted in Figure 2. In general, this info can be represented by a so-called SNN graph, which has the same number of nodes and edges as the connectivity graph, but each edge now has a *non-negative* weight.

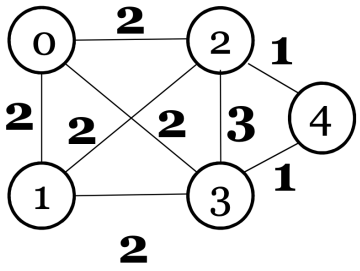

Figure 2: An example SNN graph corresponding to the connectivity graph in Figure 1.

Similarly, an SNN graph can be stored as either a 2D table or in the CRS format. The difference is that the 2D table corresponding to an SNN graph must use a 2D array of `int` values to record the numbers of SNNs. The CRS format for an SNN graph must use an additional array containing non-negative integer values, named such as `SNN_val`, which is of length $2N_{\text{edges}}$ (same as `col_idx`).

1.4 Clustering based on SNN (only relevant for IN4200 students)

Grouping nodes with similarity into clusters is a typical task of unsupervised machine learning. Clustering based on SNN, with a threshold value τ, aims to produce one or several clusters. Here, a cluster is a subset of the nodes, where each node in the cluster is directly connected with *at least* another node in the same cluster, with the number of SNNs between them being equal or larger than the threshold value τ. Take for instance the SNN graph in Figure 2, if the threshold value τ is chosen as 2, nodes 0,1,2,3 will form a cluster. For the case of $\tau = 3$, nodes 2 and 3 will form a cluster. (For $\tau \geq 4$, no clusters can be found for the SNN graph in Figure 2.)

2 Assignments of the home exam

2.1 Reading a connectivity graph from file

You are requested to implement two functions with the following syntax:

- `void read_graph_from_file1 (char *filename, int *N, char ***table2D)` which reads a text file (filename prescribed as input) that contains a connectivity graph, outputs the number nodes through `N`, allocates a 2D table of correct dimension inside the function, fills its content and passes it out (thus triple pointer as type for `table2D`).
- `void read_graph_from_file2 (char *filename, int *N, int **row_ptr, int **col_idx)` which has the same purpose as above except that the CRS format is used. Note: The arrays `row_ptr` and `col_idx` are to be allocated inside this function (thus double pointer as type), and you should avoid internally allocating/using a 2D table.

The format of a text file that contains a connectivity graph is described in Section 3.

2.2 Creating an SNN graph

You are requested to implement two functions for creating the corresponding SNN graph for a given connectivity graph. The syntax of the functions is as follows:

- `void create_SNN_graph1 (int N, char **table2D, int ***SNN_table)` which has `N` and `table2D` as input, and `SNN_table` as output (to be allocated inside the function, thus triple pointer as type).

- `void create_SNN_graph2 (int N, int *row_ptr, int *col_idx, int **SNN_val)` which has `N`, `row_ptr` and `col_idx` as input, and `SNN_val` as output (to be allocated inside this function, thus double pointer as type).

OpenMP parallelization: You are requested to parallelize the above two functions using OpenMP programming. You can either directly insert suitable OpenMP directives (and additional code if necessary), or write two separate OpenMP-specific functions. (Note: The standard macro `#ifdef _OPENMP ... #endif` can be useful when inserting OpenMP-specific code directly into sequential code.)

2.3 Only for IN4200 students: Checking whether a node can be in a cluster

Each IN4200 student is requested to implement the following function:

- `void check_node (int node_id, int tau, int N, int *row_ptr, int *col_idx, int *SNN_val)`

This function checks for a given node (with index `node_idx`) and a threshold value τ, whether it can be inside a cluster based on the input SNN graph (in the CRS format). If yes, the function should print out the other nodes inside the cluster.

2.4 Test program

You are requested to write a single test program to test all the above functions (both the sequential and OpenMP implementations), or alternatively an additional test program for the OpenMP versions.

3 File format of a connectivity graph

It can be assumed that a text file containing a connectivity graph has the following format:

- The first two lines both start with the `#` symbol and contain free text (listing the name of the data file, authors etc.);
- Line 3 is of the form "`# Nodes: integer1 Edges: integer2`", where `integer1` is the total number of nodes, and `integer2` is the total number of unique edges;
- Line 4 is of the form "`# FromNodeId ToNodeId`";
- The remaining part of the file consists of a number of lines, the total number equals the number of edges. Each line simply contains two integers: the index pair of a nearest neighbor pair;
- Some of the edges can be self-links (same `FromNodeId` as `ToNodeId`), these should be excluded (not used in the data storage later);

- Some of the edges may have "illegal" values for `FromNodeId` and/or `ToNodeId`, these should be excluded (not used in the data storage later);
- Note: all the indices start from 0 (C convention).
- It can be assumed that each edge is uniquely listed, that is, there will NOT be multiple text lines describing the same undirected edge.
- You may NOT assume that the edges are sorted with respect to `FromNodeId`.
- For each `FromNodeId`, you may NOT assume that the edges are sorted with respect to `ToNodeId`.

An example connectivity graph file corresponding to Figure 1 is as follows:

```
# Undirected graph: simple-graph.txt
# Just an example of connectivity graph
# Nodes: 5 Edges: 8
# FromNodeId    ToNodeId
0       1
0       3
0       2
2       4
2       1
2       3
1       3
3       4
```

An example real-world connectivity graph file, `facebook_combined.txt`, can be downloaded from the following semester webpage folder
`https://www.uio.no/studier/emner/matnat/ifi/IN3200/v21/teaching-material/`

4 Submission

Each student should submit a single tarball (`.tar`) or a single zip file (`.zip`). Upon unpacking/unzipping it should produce a folder named `IN3200_HE1_xxx` or `IN4200_HE1_xxx`, where `xxx` should be the candidate number of the student (can be found in StudentWeb). Inside the folder, there should be the following files:

- One `*.c` file for the implementation of each of the functions. The filename should be same as the function name.
- A `README.txt` or `README.md` file explaining how the compilation should be done, with additional comments if relevant. You are strongly advised to make sure that your implemented code can be compiled on a standard Linux computer.
- Preferably a Makefile (but not mandatory).
- A PDF file containing a short note that explains the basic idea (and algorithm) behind each function, as well as efficiency considerations (if relevant).

5 Grading

The grade of the submission will constitute 20% of the final grade of IN3200/IN4200. Grading of the submission will be based on the correctness, conformability (of filenames and function syntax), readability and speed of the implementations, in addition to the quality of the short note. **Hint:** In order to make sure that your code compiles on a standard Linux computer, please test the compilation on any of Ifi's Linux servers (e.g. `login.ifi.uio.no`.)