IN4200 Home Exam 1 - Counting shared nearest neighbors (C++)

Candidate no.: 15817
(Dated: March 30, 2021)

I. IDEAS AND ALGORITHMS

A. Reading the data files

I will use the standard function `fscanf` to process the data files. The function takes as argument the pointer to the `FILE` object we want to read and moves through it until it reaches the character or data type specified by the second argument. Matching decimal integers and assigning their value to variables already declared lets us read the numbers in the file.

B. `read_graph_from_file1`

This function is rather simple. Let N denote the number of nodes in the network. We want to register the edges in an $N \times N$ matrix of `char` elemements. I will use an underlying contiguous array of length N^2 to do this. This can be achieved by

```
char *contig_array = new char[N*N];
table2D = new char*[N];
for (int i=0; i<*N; i++)
  table2D[i] = &contig_array[i*N];
```

This can now be indexed by `[row][column]` like a normal 2D-matrix. Letting `table2D[i][j]` represent the link between nodes `i` and `j`, we will simply let 0 mean that there is no link between them and 1 mean that they are linked. Starting with an array of zeros, we then just set the relevant values to 1 as we read through the data file.

C. `read_graph_from_file2`

Here, we want to store the information in the data file as two one-dimensional arrays using the CRS format. Again N denotes the number of nodes, and N_{edges} denotes the number of edges (connections between nodes). I will start by declaring an array of zeros, `node_links`, to hold the number of nodes connected to each node. I.e. if node 2 is connected to three other nodes, the value of `node_links[2]` would be 3. This is easily counted while reading through the file. I will also store the information in the text file in two arrays `from_array` and `to_array`, both of length N_{edges}.

After reading through the file we want to store the information in two arrays: `row_ptr` of length $N + 1$ and `col_idx` of length $2N_{\text{edges}}$. The values of `row_ptr` are easily calculated from the `node_links` array. The array should start at 0. The second element should be the number of nodes connected to node 0, the third elemenent should be the sum of the second element and the number of nodes connected to node 1, the fourth element should be the sum of the third element and the number of nodes connected to node 2, etc. This is achieved by the following loop:

```
row_ptr[0] = 0;
for (int i=1; i<N+1; i++)
  row_ptr[i] = row_ptr[i-1] + node_links[i-1];
```

After `row_ptr` has been filled, I use the following code to fill `col_idx` (which is currently full of zeros) with the correct values:

```
int from, to, col;
for (int i=0; i<N_edges; i++) {
  from = from_array[i];
  to = to_array[i];

  col = row_ptr[from+1] - 1;
  while (col_idx[col] > to)
    col -= 1;
  for (int j=row_ptr[from]; j<col; j++)
    col_idx[j] = col_idx[j+1];
  col_idx[col] = to;

  col = row_ptr[to+1] - 1;
  while (col_idx[col] > from)
    col -= 1;
  for (int j=row_ptr[to]; j<col; j++)
    col_idx[j] = col_idx[j+1];
  col_idx[col] = from;
}
```

The integers `from` and `to` are the indices of two nodes that are connected. `col` is the index for where in `col_idx` to place the connection. To show that node `from` is connected to node `to` we must place the value `to` in `col_idx` in the area between indices `row_ptr[from]` and `row_ptr[from+1] - 1` (inclusive). Starting at the end of this interval, the first while loop moves the counter toward the front of the area if the places are already filled by larger numbers. After having moved past the larger entries, it settles on an index to place the number. The numbers ahead of this position are shifted to the left to make room. In this way, the CRS arrays are sorted after the algorithm is finished. The last code block above repeats the process with `to` and `from` switched (the matrix is symmetric).

It could have been better to start from the beginning of the intervals and move forward through them. The Facebook dataset is (atleast almost) sorted from lowest to highest such that the numbers would not be moved

around as much if I traversed the interval in the other direction. However, there are a few small practical challenges to this if you want a sorted array that caused me to go for the strategy I have described.

D. create_SNN_graph1

Here, we want to calculate the number of shared neares neighbors from the two-dimensional array of connections made by the function discussed in section I B. The strategy is simple: if nodes `i` and `j` are connected, that is if `table2D` is 1, then the corresponding SNN-value is given by the dot product between these two rows (`table2D[i]` and `table2D[j]`). Since the SNN-matrix is symmetric, we can manage by looping over only the upper or lower triangular part of the matrix as follows

```
for (int i=0; i<N; i++) {
  for (int j=i+1; j<N; j++) {
    if (table2D[i][j]) {
      dot_product = 0;
      for (int k=0; k<N; k++)
        dot_product += table2D[i][k]*table2D[j][k];
      SNN_table[i][j] = dot_product;
      SNN_table[j][i] = dot_product;
    }
  }
}
```

In order to parallelize this code I have simply instructed that the outermost for-loop be run in parallel. Since the size of the second for-loop decreases with `i`, it is desirable to distribute the iterations between the threads such that iteration 0 goes to thread 0, iteration 1 to thread 1, etc. This is achieved by adding `schedule(static, 1)` to the parallelization instruction. Thus the total computation time is evened out between the threads without the extra overhead of dynamic loop scheduling.

E. create_SNN_graph2