IsoRank

This software is meant to be a program that can be used to obtain an approximate answer to the NP-hard graph matching problem. The program written takes two graphs at a time and finds a node to node mapping between the two graphs so that a subgraph isomorphism graph can be found between two graphs. The graph matching algorithm has two steps: first a score is assigned to each node to node mapping and then scores are chosen to create an appropriate mapping. For the first step we implemented the IsoRank Algorithm and for the second step there are 5 possible algorithms one can choose to run: simple greedy, greedy connectivity 1, greedy connectivity 2, greedy connectivity 3 and greedy connectivity 4. The program takes as input a directory where each file represents a graph. Each graph in the directory is compared to every other graph and for each pair of graphs three values are computed: a node to node mapping, a score indicating how appropriate the mapping is, and another value the frobenius norm to indicate the quality of mapping (the higher the score the better, the lower the frobenius norm the better). Created in the summer of 2013 this software was originally written with the intention of being used by the Ferguson group at the Materials Science & Engineering Department at University of Illinois at Urbana-Champaign. This software however, can be used for any purpose that requires an approximate solution to the graph matching problem.

Compilation:

To compile this project you can use the make file included in the directory:

```
    Command line:
        make Makefile
```

Preprocessor flags:

```
Library flags:
    -DARPACK: To use Arpack++ library for eigenvector decomposition.
    -DEIGEN: To use Eigen3 for eigenvector decomposition.
* Only one of the library flags should be set.

Executable flag:
    -DSEQ: to compile the serial version
    -DNODE_PAIR: to compile parallel version using node pair method (see Parallelization)
* The default parallelization method is Broadcast (see Parallelization)
```

The library path must be set for Arpack++ and Eigen depend on the one that you are using. You need Open MPI 32 bit compiler.

Execution:

There are three versions of the graph matching software available, a sequential version and two parallel versions.

To run the sequential version:

```
    ./IsoRank [-dir <directory_name>] [-ext <file_extension>] [-num_files <number_of_files>]
        [-match_alg <matching_algorithm>] [-alg <graph_matching_alg>] [-print] [-debug]
```

To run the parallel versions with mpi:

```
    mpirun -np #number_of_processors ./IsoRank [-dir <directory_name>] [-ext <file_extension>]
    [-num_files <number_of_files>] [-match_alg <matching_algorithm>] [-alg <graph_matching_alg>] [-print] [-debug]
```

Explanation of flags:

```
    [-dir <directory_name>] -dir indicates that the files where the graphs are stored are in the directory called direct
        *Default directory is "IsoRank/Sample input/"

    [-ext <file_extension>] -ext indicates that the extension of the files being read is file_extension:
        *Default for file_extension is .dat

    [-num_files <number_of_files>] -num_files indicates that number_of_files need to be read in to be compared:
        *Default for number_of_files is 2

    [-match_alg <matching_algorithm>] -match_alg indicates that matching_algorithm needs be used to map nodes to nodes:
        matching_algorithm options: greedy, con-enf-1, con-enf-2,con-enf-3,con-enf-4
        *Default value for matching_algorithm is greedy

    [-alg <graph_matching_alg>] -alg indicates that graph_matching_alg is to be used:
        graph_matching_alg options: isorank, gpgm
        Default value for graph_matching_alg is isorank

    [-print] prints out results i.e. frobenius norm, time taken,  etc.
    [-debug] prints out values useful for debugging your program

    -np #number_of_processors indicates that #number_of_processors need to be used to run the program in parallel.
```

Format of Input Files:

Each graph in the graph matching algorithm is represented by an adjacency matrix. The program expects that input files be formatted in a specific way. The first line of the file should have 3 integers: #nodes, #nodes, and number of non-zero values (each separated by a space). Each of the following lines should contain the row and column of the matrix where a non-zero value resides**. For example the input file for the following matrix:

```
    1 0 0
    0 1 1
    0 1 1

    Should look like
    3 3 5
    1 1
    2 2
    2 3
    3 2
    3 3
```

Sample files are included under the "Sample input" folder and you can open them using a text editor. **The program expects files that have row/column indices that are 1-based. So the upper-left most value in a matrix A is A(1,1).

Design Decisions:

Matrices Implemented

Each graph is represented as an adjacency matrix where if matrix(i, j)==1 then there is an edge between node i and node j, and if matrix(i, j)==0 then there is no edge between node i and node j. All matrix classes can be found in the Matrices directory. There are 3 classes that have been implemented: DenseMatrix1D.h, DenseMatrix2D.h, SymMatrix.h. DenseMatrix2D uses a 2-dimensional array to represent the adjacency matrix. DenseMatrix1D uses a 1-d array to represent the adjacency matrix using row major order.

SymMatrix.h uses a 1-dimensional array to represent the adjacency matrix as well but since the matrix is symmetric only half the values are stored.

The default Matrix class we've used in the program is the DenseMatrix1D.h class.

**Note that the SymMatrix class is not complete and only some of the methods are implemented.

Connectivity Algorithms

Recall that the second step of the algorithm requires us to choose the best scores to create a final mapping. There are 5 connectivity algorithms we've implemented. We recommend that that one use either the simple greedy algorithm, greedy connectivity 3 or greedy connectivity 4. Simple greedy is the fastest of the 5 algorithms and gives a fairly good approximate isomorphic graph for graphs. Greedy connectivity 3 and 4 both perform slower than simple greedy but both do a much better job of giving an isomorphic graph for input graphs that are very highly connected. Greedy Connectivity 1 and 2 were both implemented since they contain elements of greedy connectivity 3 and 4, but their performance is not as good as either 3 or 4.

Parallelization

Two parallelization methods have been used.

Node_Pair method:

This method is a simple master-slave architecture. Each worker node is passed a pair of graphs and once a worker node is done performing the computation on the pair of graphs, it returns the result to the master node and requests for another pair of graphs.

Broadcast method:

In the second parallelization method each worker node reads in all the graphs. The master node then assigns each worker node indices indicating which subset of the graphs to run isorank on.

After benchmarking both parallelization methods we have come to the conclusion that the second method is faster than the first method.

Note: It is simple to have each of the processors read all the graphs and then compute their portion of the graphs. This method will save time in the case that the number of inputs are large.

Send/Recv Matrices:

All the matrix classes can send and receive data from each other except full matrices to SymMatrix class while full matrices can receive data from a SymMatrix and they assume that the matrix is symmetric.

Sending matrices is done by calling the the method MPI_Send_Matrix/MPI_Bcast_Send_Matrix. In addition to require parameters for these method there is an optional boolean that if set to true matrices will send sparse data. This method is useful is the matrices are enormous and very sparse.

Receiving matrices is being done by calling the constructor of your desired matrix type, setting the template data type and giving it the right parameters. You do not need to tell it that it's receiving a sparse matrix in case that sender is sending sparse form.

Potential Bugs:

As mentioned previously Greedy Connectivity 1 and 2 were implemented as stepping stones to be able to implement Greedy Connectivity 3 and Greedy Connectivity 4. Although we have tested connectivity 1 and 2, there may be potential bugs in the code.

Sending and receiving sparse matrices on OSX.