3 OPTION 2: HITS and PageRank implementations (134 points)

Implement Kleinberg's HITS Algorithm, and Google's PageRank algorithm in Java, C, or C++ as explained.

(A) Implement the HITS algorithm as explained in class/Subject notes adhering to the guidelines of Handout 2. Pay attention to the sequence of update operations and the scaling. For an example of the Subject notes, you have output for various initialization vectors. You need to implement class or function `hits` (e.g. `hitsWXYZ`. We expect that no two students will have the same last 4 digits, and no student will use WXYZ (as in characters W, X, Y, Z) either! For an explanation of the arguments see the discussion on PageRank to follow.

```
% java hits iterations initialvalue filename
% ./hits    iterations initialvalue filename
```

(B) Implement Google's PageRank algorithm as explained below adhering also to the guidelines of Handout 2. The input for this (and the previous) problem would be a file containing a graph represented through an adjacency list representation. The command-line interface is as follows. First we have the class/binary file (eg pgrk). Next we have an argument that denotes the number of `iterations` if it is a positive integer or an `errorrate` for a negative or zero integer value. The next argument `initialvalue` indicates the common initial values of the vector(s) used. The final argument is a string indicating the `filename` that stores the input graph.

```
% ./pgrk     iterations initialvalue filename  // in fact pgrkWXYZ
% java pgrk  iterations initialvalue filename  // in fact pgrkWXYZ
```

The two algorithms are iterative. In particular, at iteration t all pagerank values are computed using results from iteration $t-1$. The `initialvalue` helps us to set-up the initial values of iteration 0 as needed. Moreover, in PageRank, parameter d would be set to 0.85. The PageRank of vertex A depends on the PageRanks of vertices $T_1,\ldots,T_m$ incident to A, i.e. pointing to A. The contribution of T_i to the PageRank of A would be the PageRank of T_i i.e. PR(T_i) divided by $C(T_i))$, where $C(T_i)$ is the out-degree of vertex T_i.

$$PR(A) = (1-d)/n + d\,(PR(T_1)/C(T_1) + \ldots + PR(T_m)/C(T_m))$$

The pageranks at iteration t use the pageranks of iteration $t-1$ (synchronous update). Thus PR(A) on the left is for iteration t, but all PR (T_i) values are from the previous iteration $t-1$. (In an asynchronous update, we have only one vector!) Be careful and synchronize!

In order to run the 'algorithm' we either run it for a fixed number of iterations and `iterations` determines that, or for a fixed `errorrate` (an alias for `iterations`); an `iterations` equal to 0 corresponds to a default errorrate of 10^{-5}. A `-1, -2, etc , -6` for `iterations` becomes an errorrate of $10^{-1}, 10^{-2}, \ldots, 10^{-6}$ respectively. At iteration t when all authority/hub/PageRank values have been computed (and auth/hub values scaled) we compare for every vertex the current and the previous iteration values. If the difference is less than **errorrate** for EVERY VERTEX, then and only then can we stop at iteration t.

Argument `initialvalue` sets the initial vector values. If it is `0` they are initialized to 0, if it is `1` they are initialized to 1. If it is `-1` they are initialized to $1/N$, where N is the number of web-pages (vertices of the graph). If it is `-2` they are initialized to $1/\sqrt{N}$. `filename` first.)

Argument `filename` describes the input (directed) graph and it has the following form. The first line contains two numbers: the number of vertices followed by the number of edges which is also the number of remaining lines. All vertices are labeled $0,\ldots,N-1$. Expect N to be less than 1,000,000. In each line an edge (i,j) is represented by `i j`. Thus our graph has (directed) edges $(0,2),(0,3),(1,0),(2,1)$. Vector values are printed to 7 decimal digits. If the graph has N GREATER than 10, then the values for `iterations, initialvalue` are automatically set to `0` and `-1` respectively. In such a case the hub/authority/pageranks at the stopping iteration (i.e t) are ONLY shown, one per line. The graph below will be referred to as `samplegraph.txt`

```
4 4
0 2
0 3
1 0
2 1
```

The following invocations relate to `samplegraph.txt`, with a fixed number of iterations and the fixed error rate that determines how many iterations will run. Your code should compute for this graph the same rank values (intermediate and final). A sample of the output for the case of $N > 10$ is shown (output truncated to first 4 lines of it).

```
% ./pgrk  15 -1 samplegraph.txt
Base   :  0 :P[ 0]=0.2500000 P[ 1]=0.2500000 P[ 2]=0.2500000 P[ 3]=0.2500000
Iter   :  1 :P[ 0]=0.2500000 P[ 1]=0.2500000 P[ 2]=0.1437500 P[ 3]=0.1437500
Iter   :  2 :P[ 0]=0.2500000 P[ 1]=0.1596875 P[ 2]=0.1437500 P[ 3]=0.1437500
Iter   :  3 :P[ 0]=0.1732344 P[ 1]=0.1596875 P[ 2]=0.1437500 P[ 3]=0.1437500
Iter   :  4 :P[ 0]=0.1732344 P[ 1]=0.1596875 P[ 2]=0.1111246 P[ 3]=0.1111246
Iter   :  5 :P[ 0]=0.1732344 P[ 1]=0.1319559 P[ 2]=0.1111246 P[ 3]=0.1111246
Iter   :  6 :P[ 0]=0.1496625 P[ 1]=0.1319559 P[ 2]=0.1111246 P[ 3]=0.1111246
Iter   :  7 :P[ 0]=0.1496625 P[ 1]=0.1319559 P[ 2]=0.1011066 P[ 3]=0.1011066
Iter   :  8 :P[ 0]=0.1496625 P[ 1]=0.1234406 P[ 2]=0.1011066 P[ 3]=0.1011066
Iter   :  9 :P[ 0]=0.1424245 P[ 1]=0.1234406 P[ 2]=0.1011066 P[ 3]=0.1011066
Iter   : 10 :P[ 0]=0.1424245 P[ 1]=0.1234406 P[ 2]=0.0980304 P[ 3]=0.0980304
Iter   : 11 :P[ 0]=0.1424245 P[ 1]=0.1208259 P[ 2]=0.0980304 P[ 3]=0.0980304
Iter   : 12 :P[ 0]=0.1402020 P[ 1]=0.1208259 P[ 2]=0.0980304 P[ 3]=0.0980304
Iter   : 13 :P[ 0]=0.1402020 P[ 1]=0.1208259 P[ 2]=0.0970858 P[ 3]=0.0970858
Iter   : 14 :P[ 0]=0.1402020 P[ 1]=0.1200230 P[ 2]=0.0970858 P[ 3]=0.0970858
Iter   : 15 :P[ 0]=0.1395195 P[ 1]=0.1200230 P[ 2]=0.0970858 P[ 3]=0.0970858
```

```
% ./pgrk -3 -1 samplegraph.txt
Base   :  0 :P[ 0]=0.2500000 P[ 1]=0.2500000 P[ 2]=0.2500000 P[ 3]=0.2500000
Iter   :  1 :P[ 0]=0.2500000 P[ 1]=0.2500000 P[ 2]=0.1437500 P[ 3]=0.1437500
Iter   :  2 :P[ 0]=0.2500000 P[ 1]=0.1596875 P[ 2]=0.1437500 P[ 3]=0.1437500
Iter   :  3 :P[ 0]=0.1732344 P[ 1]=0.1596875 P[ 2]=0.1437500 P[ 3]=0.1437500
Iter   :  4 :P[ 0]=0.1732344 P[ 1]=0.1596875 P[ 2]=0.1111246 P[ 3]=0.1111246
Iter   :  5 :P[ 0]=0.1732344 P[ 1]=0.1319559 P[ 2]=0.1111246 P[ 3]=0.1111246
Iter   :  6 :P[ 0]=0.1496625 P[ 1]=0.1319559 P[ 2]=0.1111246 P[ 3]=0.1111246
Iter   :  7 :P[ 0]=0.1496625 P[ 1]=0.1319559 P[ 2]=0.1011066 P[ 3]=0.1011066
Iter   :  8 :P[ 0]=0.1496625 P[ 1]=0.1234406 P[ 2]=0.1011066 P[ 3]=0.1011066
Iter   :  9 :P[ 0]=0.1424245 P[ 1]=0.1234406 P[ 2]=0.1011066 P[ 3]=0.1011066
Iter   : 10 :P[ 0]=0.1424245 P[ 1]=0.1234406 P[ 2]=0.0980304 P[ 3]=0.0980304
Iter   : 11 :P[ 0]=0.1424245 P[ 1]=0.1208259 P[ 2]=0.0980304 P[ 3]=0.0980304
Iter   : 12 :P[ 0]=0.1402020 P[ 1]=0.1208259 P[ 2]=0.0980304 P[ 3]=0.0980304
Iter   : 13 :P[ 0]=0.1402020 P[ 1]=0.1208259 P[ 2]=0.0970858 P[ 3]=0.0970858
```

```
% ./pgrk  0  -1 verylargegraph.txt
Iter   : 4
P[ 0]=0.0136364
P[ 1]=0.0194318
P[ 2]=0.0310227
   ... other vertices omitted
```

For the HITS algorithm, you need to print two values not one. Follow the convention of the Subject notes

```
Base   :  0 :A/H[ 0]=0.3333333/0.3333333 A/H[ 1]=0.3333333/0.3333333 A/H[ 2]=0.3333333/0.3333333
Iter   :  1 :A/H[ 0]=0.0000000/0.8320503 A/H[ 1]=0.4472136/0.5547002 A/H[ 2]=0.8944272/0.0000000
```

or for large graphs

```
Iter   : 37
A/H[ 0]=0.0000000/0.0000002
A/H[ 1]=0.0000001/0.0000238
A/H[ 2]=0.0000002/1.0000000
A/H[ 3]=0.0000159/0.0000000
 ...
```

Deliverables. Include source code of all implemented functions or classes in an archive per Handout 2 guidelines. Document bugs; no bug report no partial points.