CS 330 Algorithms in the Real World: HW4

Due 3:30 pm on Tuesday, October 12

Submission. You will submit materials for homework number n on Gradescope under two separate assignments: HWn Report and HWn Code. You should turn in a pdf containing all of your written solutions for the report, and a source code file containing all of the code you wrote for the assignment. When you submit your code, the interface may mention an autograder; you can ignore this, as the assignment will not be autograded. Your grade will be recorded on the report; the source code is to show your work and for reference during grading.

Report. The pdf you submit for the report should be typed, and solutions to individual questions should be clearly labeled. Show your work or explain your reasoning for your answers. You should use LaTeX (which is free) or some other editor of your choice (Microsoft Word, Google Docs, etc.) to prepare your reports. If you use an editor like Microsoft Word, make sure to convert the final document to a pdf, confirm that the symbolic math from the equation editor is properly formatted, and submit the pdf.

Collaboration. You may complete this assignment independently or in a group of two students. If you work with another student you should submit a single report and a single code file with both of your names, and should use the group feature when submitting on gradescope to indicate that you worked as a group. Do not split up the assignment, and each only complete half of the problems. Instead, complete each portion of the assignment by working together synchronously (for example, by pair programming with screen sharing over zoom or some other similar service) or by working independently and then coming together to merge solutions and check one another?s work.

Allowed Materials. You can use any standard library functions and data structures in your programming language of choice (Java or Python 3 are recommended). You may also use any slides or notes from class or reference materials posted on the course website. You may search the internet for basic definitions, terminology, and language documentation (for example, checking the syntax for array slicing in python), but you may not use anyone else's code (either another student's or from the internet), nor may you search the internet for solutions or descriptions of solutions to the homework problems.

Problem 1 (PageRank and the Twitter Graph of the United States Congress, 46 points). Recall that pagerank is the stationary distribution π^* of a Markov chain (a stochastic process) over a directed graph $G = (V, E)$ with $|V| = n$ vertices/nodes and E the set of directed edges. We initialize a vector π^0 to be the uniform distribution, i.e., a vector of $1/n$ repeated n times.[1] Let d_i be the *outdegree* of node i (that is, the number of outgoing edges from node i). We iteratively update the values by the formula (note that $(i, j) \in E$ means that there is a directed edge from node i to node j):

$$\pi_j^{t+1} = \sum_{i \in V} \pi_i^t P_{ij} \text{ where } P_{ij} = \begin{cases} \frac{0.85}{d_i} + \frac{0.15}{n} & (i,j) \in E \\ \frac{0.15}{n} & (i,j) \notin E \text{ and } d_i > 0 \\ \frac{1}{n} & d_i = 0 \end{cases} \quad (1)$$

To interpret equation 1, note that the probability of being at node j in round $t + 1$ is just the sum over all nodes i of the probability of being at i in round t times the probability of transitioning from i to j according to the pagerank random walk. The pagerank random walk dictates the transition probabilities P_{ij}. Recall that with 85% probability, a node follows one of it's outgoing edges chosen uniformly at random, and with

[1] The superscript here on π simply refers to the distribution at a given round of the Markov chain, *not* to exponentiation.

the remaining 15% probability it jumps to a random node in the graph. The first case above corresponds to this when there is an edge from i to j. The second case corresponds to this when there is not an edge from i to j, but there are some outgoing edges from i. The final case corresponds to "sink" nodes, i.e., vertices with no outgoing edges, which simply jump to a uniform random node in the graph.[2]

The pagerank is defined as $\pi^* = \lim_{t\to\infty} \pi^t$ according to the above updates. In practice, we compute a close approximation to this value by performing iterations until the distribution does not change very much (no more than some tolerance that is given as input). We measure the difference between two distributions as the total variation distance (TVD), so that we continue iterating until $TVD(\pi^t, \pi^{t-1}) = \frac{1}{2}\sum_{i=1}^{n} |\pi_i^t - \pi_i^{t-1}| < \epsilon$ for some small tolerance threshold ϵ. We say that the algorithm has converged when this happens, and we simply report the last distribution π^t as the pagerank vector, where the pagerank of a particular node is just its corresponding entry in the vector.

(a) Implement an algorithm to compute pagerank up to a tolerance threshold as described above. While debugging, you may wish to add a `max_iterations` value/parameter to ensure that your code stops iterating eventually even if the code does not converge to a solution. 100 is a reasonable value to use for this while debugging. Another good debugging check is to ensure that $\sum_{i=1}^{n} \pi^t \approx 1$ at each iteration t (note that π is a probability distribution, so it should sum to 1 except for very small rounding errors). A common error to watch out for: Make sure you calculate *all* of the round $t+1$ pagerank scores before you update any of them (i.e., make sure you only use round t scores in computing round $t+1$ scores).

You can test your implementation with the simple `test_nodes.txt` and `test_edges.txt` files. Each line of `test_nodes.txt` gives the name of a particular node in a graph, and each line of `test_edges.txt` gives an integer x followed by a space followed by another integer y. Each such line represents a directed edge from the line x node to the line y node (all 0-indexed) in test_nodes. So for example, if the first two lines of "test_nodes" were to read `Alice` and `Bob`, and the first line of "test_edges" reads `0 1`, that denotes that there is an edge from Alice to Bob, whereas `1 0` would denote an edge from Bob to Alice. This is the same format that will be used for the subsequent data files for this assignment.

Testing on this file using a tolerance threshold ϵ of 0.001, you should find the following values (up to small rounding / floating point error). It should take about 31 iterations to converge. **Nothing to report for this part, just your code.**

```
Pagerank of  A :  0.05257
Pagerank of  B :  0.41544
Pagerank of  C :  0.38474
Pagerank of  D :  0.04734
Pagerank of  E :  0.05257
Pagerank of  F :  0.04734
```

(b) The files `members.txt` and `follows.txt` provide a somewhat more interesting example of a directed graph. In particular, members contains a list of the members of the United States Congress (house and senate) with official Twitter accounts in summer 2020. `follows.txt` has a directed edge from member x to member y if member x follows member y on Twitter.[3] Compute the pagerank scores for the members of congress according to their Twitter graph. Use a tolerance threshold ϵ of 0.001. **Report how many iterations it takes to converge as well as the names of the 5 members of congress with the highest pagerank scores in order, along with their pagerank scores.**

(c) **Report the 5 members of congress with the highest pagerank scores after *just one* iteration, again in order and with their scores.** You should notice some differences from the converged results. **How would you explain those differences?** Your answer only needs to be a couple of sentences

[2] It is mathematically equivalent to simply connect sink nodes to every node in the graph (including a self-loop edge to itself) and then ignore the third and now-irrelevant case.

[3] Credit for pulling this data from the Twitter API to Charles Lyu as part of a research project on graph centrality measures.

explaining the intuition, not a formal proof. [hint: look at the indegrees (that is, the number of edges pointing to) the nodes that are different and think about the first step of the pagerank updates in equation 1.]

Problem 2 (Scaling PageRank and Search in a Citation Network, 40 points). In this problem, you will work with a citation network of high energy physics papers from arXiv from 1993 to 2003,[4] consisting of over 20,000 research articles (nodes in the graph). Most papers cite a number of other papers in the set, and each such citation is considered an edge in the graph from the paper citing to the paper being cited. Each line of the file `papers.txt` contains all of the information (title, authors, abstract, etc) of a given paper (all in lower case), and each line of `cite.txt` denotes a citation edge as in the edge files from problem 1.

(a) Recall that n is the number of nodes in the graph. In this data set, n is much larger than in problem 1. The running time of a single iteration in computing the pagerank (i.e., computing π^{t+1} given π^t) is $O(n^2)$ in a simple implementation using equation 1 directly. In a *sparse* graph (one where the number of edges is $O(n)$), it is possible to compute a single iteration in $O(n)$ time. **Explain how to do this and implement the optimization in your code.** Be sure to double check that you get the same results with or without the runtime optimization on the data from problem 1.

You will need to do some pre-processing to determine, for each node, which other nodes point to it, and what their outdegrees are. When doing this, you should only scan through the edges, not a nested $O(n^2)$ for loop over the nodes. Dictionaries/HashMaps might be useful for this. Then think about how you can simplify your calculation of equation 1. It may be helpful to think separately about adding the contributions to scores from edges (the $0.85/d_i$ terms) and from the random teleportation. If you get stuck on this question, you can still try the subsequent questions with a more straightforward implementation.

(b) Using your efficient implementation, compute the pagerank scores of the articles. Use a tolerance threshold of $\epsilon = 0.00001$ (note this is smaller than in problem 1). **Report the titles[5] of the five articles with the highest overall pagerank scores, along with their scores.**[6]

(c) Now you will build a simple search engine over the articles. To do so, we will first find the articles that are relevant to a search query, and then rank them according to their pagerank importance. First implement a simple function that checks which articles contain a search query. For simplicity, treat a search query as a string, and return just those articles that contain *exactly* that string somewhere in the information about them stored in the `papers.txt` file. This should just be a simple loop over the articles and a `.contains` check in Java or `in` check in Python. Next, rather than simply returning the articles unordered, *sort* the articles *by their pagerank scores* that you calculated in part (b). Return/print the first five (or however many results there are, if less than 5) articles in order.

As an example, below we show the results for the query "quantum". **Report your results for the query "neutrino".** Congratulations, you have a working search engine![7]

1. dirichlet-branes and ramond-ramond charges 0.0041329
2. exact results on the space of vacua of four dimensional susy gauge theories 0.0037514
3. unity of superstring dualities 0.0032304
4. monopoles, duality and chiral symmetry breaking in n=2 supersymmetric. qcd 0.0030672
5. m theory as a matrix model: a conjecture 0.0026161

[4] This data is originally taken from the Stanford SNAP network data repository: `https://snap.stanford.edu/data/cit-HepTh.html`. It has been reformatted for your convenience.

[5] It is fine to just copy paste the titles out of all of the paper data rather than manipulating the strings in code, unless you want to practice regular expressions

[6] If you were unable to solve part (a) of this problem, you can still use your previous $O(n^2)$ solution and get credit on parts b and c. The only problem is that your code may be slow to run, expect something on the order of 1-5 minutes per iteration of pagerank. If it is too slow to run on the entire dataset, you can just count the in-degrees (i.e., the number of citations each paper receives) and use those instead of pageranks to answer parts b and c for partial credit.

[7] Did you enjoy this assignment? Want to play around more with scaling up even further to a really interesting use case? A snapshot of the entire Wikipedia graph (where nodes are pages and edges are hyperlinks) as of 2013 is publicly available here: `https://snap.stanford.edu/data/enwiki-2013.html` if you are interested, with over 4 million articles.