Homework 2 of CS7316

METHOD

To compute the page rank value of each node in the graph, we firstly try using following power iteration.

$$r^N = M^{N\times N} r^N$$

$$M_{ji} = \begin{cases} \frac{1}{d_i} & \text{if graph[i,j] exists} \\ 0 & \text{otherwise} \end{cases}$$

Where N is the node number and d_i is the out degree of node i. To avoid spider traps problem and dead ends problem, we can use the teleports, which means we randomly jump to a page with probability $1-\beta$. Then we has the new power iteration

$$r = Ar$$

$$A = \beta M + (1-\beta)[\frac{1}{N}]^{N\times N}$$

Where β is the random probability parameter, typically set to 0.8 or 0.9. However, computing the above requires significant memory when N is large. For instance, in this homework, with $N = 281873$, the memory required for A is approximately 296 GB (assuming each data point requires 4 bytes, $281873^2 * 4/1024^3 \approx 296$). Allocating 296 GB of memory is not feasible in most cases. Therefore, it's necessary to reduce the memory usage. Since M is a sparse matrix, it can be stored using much less memory. Subsequently, we can compute page rank values using the following equations.

$$\begin{aligned} r_j &= \sum_{i=1}^{N} A_{ji} * r_i \\ &= \sum_{i=1}^{N} [\beta M_{ji} + \frac{1-\beta}{N}] * r_i \\ &= \sum_{i=1}^{N} \beta M_{ji} * r_i + \frac{1-\beta}{N} \sum_{i=1}^{N} r_i \\ &= \sum_{i=1}^{N} \beta M_{ji} * r_i + \frac{1-\beta}{N} \end{aligned}$$

To store M, we can utilize the format illustrated in Figure 1. This format allows us to store M with the graph in this homework using only about 11 MB of memory, calculated as $(281873 + 281873 + 2300000) * 4/1024/1024 \approx 11$ MB. Subsequently, we employ Algorithm 1 to compute page rank values. In Line 15 of the algorithm, we address the dead ends problem by forcing that $\sum r = 1$.

Conference'17, July 2017, Washington, DC, USA
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

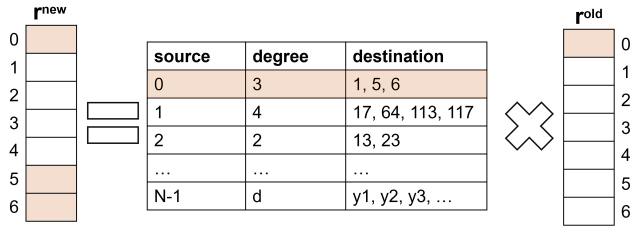

Figure 1: Store format of M

Algorithm 1 PageRank Computing

1: **Input:** graph G, node number N ,random probability β
2: **Output:** PageRank Values
3: out degree $D \leftarrow$ zeros(N)
4: sparse matrix $M \leftarrow$ dict()
5: for edge e in G
6: $D[e[0]] += 1$
7: $M[e[0]]$ add $e[1]$
8: page rank $r \leftarrow [\frac{1}{N}]^N$
9: set iteration with T
10: for t from 0 to T
11: new page rank $r^{new} \leftarrow [\frac{1-\beta}{N}]^N$
12: for each node i:
13: for each destination j in $M[i]$:
14: $r^{new}[j] += \beta r[i]/D[i]$
15: $r^{new} += (1 - \sum r^{new})/N$ //deal with dead ends
16: $r \leftarrow r^{new}$
17: save page rank r

EVALUATION

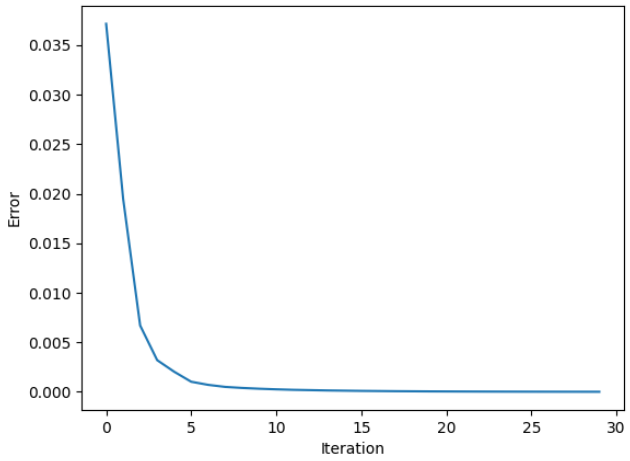

Figure 2: Iteration process of Algorithm 1. Error is the norm of $(r^{new} - r)$.

After implementing Algorithm 1 in Python, with $\beta = 0.85$, on a server equipped with 64 CPU cores (Intel(R) Xeon(R) Platinum 8336C CPU @ 2.30GHz) and 256 GB of memory, we observed the iteration process depicted in Figure 2. The figure illustrates Algorithm 1 converging rapidly, typically within about 30 iterations. We utilized the Python libraries `time` and `resource` to measure the time and memory costs, respectively. In our experiments, the time cost was approximately 43.1 seconds, while the memory cost was around 361.3 MB. The memory requirement is negligible for most devices, so storing the matrix M and the page rank vector r to disk is unnecessary. However, the time cost is somewhat high due to the inefficiency of the two-level loop structure in the algorithm. To achieve faster computation, we can utilize the `csr_matrix` format from the Python `scipy.sparse` module to store the matrix M and leverage its sparse matrix-vector multiplication operator for computing the page rank values. With this approach, we can complete the process in only 3.7 seconds, compared to the previous 43.1 seconds, while maintaining the same memory consumption. Thus, for efficient computation, leveraging powerful libraries is advisable. The final top 1000 nodes and their page rank values are saved in the `test_prediction.csv`.