SCOTTYRANK.JL: A JULIA IMPLEMENTATION OF PAGERANK AND HITS

SIYUAN CHEN AND MICHAEL ZHOU

Abstract. PageRank is an algorithm famously used by Google to determine the relative importance of different websites for search results. More generally, PageRank and variations of the algorithm can be applied to any directed graph of objects, where one wishes to find the most "important" nodes, as determined by a combination of the number of nodes pointing to it and the number of nodes that it points to. In our implementation of PageRank, Markov Matrices simulating a random walk along the edges of a directed graph were used to determine each node's relative importance. At every step, the PageRank score of a given node would be distributed among the nodes that can be reached through a directed edge outward from the starting node. Our implementation of the HITS variation of the PageRank algorithm added "hub" and "authority" scores, which distinguish between nodes pointing to many other nodes (hubs) and nodes with many other nodes pointing to itself (authorities).

In our project, we implemented both the PageRank and HITS algorithms using Julia to better understand the linear algebra insights behind the two algorithms, and tested them on datasets of varying sizes and densities.

E-mail addresses: siyuanc2@andrew.cmu.edu, mhzhou@andrew.cmu.edu.
Date: November 2021.

1. Mathematical Background

Markov Matrices and Random Walks

Markov Matrices and their applications to Random Walks are the foundational linear algebra concepts behind the PageRank algorithm.

Markov Matrices are square matrices with strictly non-negative entries where the sum of entries in every column is equal to 1. Markov Matrices have several key properties that make them especially useful for iterative computing– for a given Markov Matrix M, the following are true:

In our PageRank implementation, our Markov Matrices are not positive (all entries positive) because we are doing random walks, which assume that at every step, there is a 0 probability that our marker stays at the current node.

2. Algorithms and Computations

2.1. **Custom Structs.** We defined the structs Vertex and Graph to be used in our PageRank algorithms. Vertices were defined as structs with an unsigned integer index, a list of indices of vertices that have directed edges pointing towards V, and a list of indices of vertices that V has directed edges pointing towards, as shown in the code segment below.

For our purposes, we defined a Graph as a struct with the number of vertices and a list of the vertices in the graph sorted by their index.

```
export Vertex, Graph

struct Vertex
  index::UInt32
  in_neighbors::Vector{UInt32}
  out_neighbors::Vector{UInt32}
end

struct Graph
  num_vertices::UInt32
  vertices::Vector{Vertex} # sorted by index
end
```

2.2. **Reading Graphs from Files.** The functions `read_graph`, `read_edge_list`, and `read_adjacency_list` are used to read and construct graphs from text files.

The format for an edge list input file is as follows:

```
<num_vertices> <num_edges>
# Directed edge from Vertex with index 1 to Vertex with index 2.
1 2
# More generally,
<Index from> <Index to>
...
```

The format for an adjacency list input file is as follows:

```
<num_vertices>
# Vertices that the first Vertex has directed edges towards
<Index> <Index> < Index > ...
...
```

2.3. **PageRank.** The function `pagerank` generates a vector containing the PageRank scores of the vertices calculated from the user-provided graph.

It takes the parameters `graph`, which is the directed graph of vertices we will apply PageRank to, and `damping`, which the the damping factor– the assumed probability that a surfer will stop traveling on any given move.

2.3.1. *PageRank Matrix Generation.* The function `pagerank_matrix` generates a Markov Matrix M that will be used in the PageRank algorithm.

To generate this matrix M, we iterate through all vertices of the graph. If the vertex V has no outgoing neighbors (no outgoing edges), the surfer should continue surfing at a new random vertex W. So, there should be an equal probability that the surfer ends up

at any other vertex W in the graph. Thus, we add the value $\frac{1}{\texttt{num\_vertices - 1}}$ to the matrix entries that represents the probability of the surfer ending up at any one of the remaining vertices W from V, which are M_{VW}, coded as `M[W, V]`.

Otherwise, there should be an equal probability that our surfer goes to any one of the vertices $V_p$that the current vertex V has directed edges towards. So for every one of the vertices V_p that the current vertex V points to, we add the reciprocal of the number of outgoing neighbors of the current vertex ($\frac{1}{\texttt{num\_out\_neighbors}}$) to the matrix entry representing the probability that we end up at V_p from V, which is M_{VV_p}, coded as `M[V_p, V]`, to represent the the aforementioned idea.

Finally, we apply the damping factor to the matrix. Since the damping factor represents the probability that a surfer will continue surfing at any given vertex, we multiply the PageRank value of every entry in the matrix by the damping factor. Then, we must add the value $\frac{\texttt{1-damping}}{\texttt{num\_vertices}-1}$ to each entry in the matrix. We add the previous fraction because if the surfer stops surfing at a given vertex V, we want to continue calculating PageRank, so we assume the surfer picks back up at any vertex. Since there are `num_vertices` vertices, and the probability a surfer stops surfing at V is `1-damping`, we must add $\frac{\texttt{1-damping}}{\texttt{num\_vertices}-1}$ to each entry in the matrix.

```
function pagerank_matrix(graph::Graph, damping::Float64)
  M = zeros(Float64, (graph.num_vertices, graph.num_vertices))
  for vertex in graph.vertices
    num_out_neighbors = length(vertex.out_neighbors)
    if num_out_neighbors == 0
      for index_to in 1:graph.num_vertices
        M[index_to, vertex.index] = 1 / (graph.num_vertices - 1)
      end
      M[vertex.index, vertex.index] = 0
    else
      for index_to in vertex.out_neighbors
        M[index_to, vertex.index] = 1 / num_out_neighbors
      end
    end
  end
  map(x -> damping * x + (1 - damping) / graph.num_vertices, M)
end
```

2.3.2. *PageRank General Algorithm.* In our general PageRank algorithm `pagerank`, we first generate a PageRank Markov Matrix from the user-provided graph, then we apply either the iterative or epsilon method to calculate a resultant matrix simulating the results of a continued random walk.

`pagerank` takes in the following parameters:

> `graph`: the user-defined graph
>
> `damping`: the damping factor for PageRank. This is the assumed probability that a surfer will stop traveling on any given move. Defaulted to 0.85

`modeparam`: the mode and the parameters for PageRank. Designates the calculation method we will use– either `"iter"` for the iterative method or `"epsi"` for the epsilon method.

Iterative: `modeparam = ("iter", num_iterations)`: PageRank for a given number of iterations Epsilon: `modeparam = ("epsi", epsilon)`: PageRank until convergence with epsilon

2.3.3. *PageRank - Iterative Method.* The first method that we can use to calculate PageRank is the iterative method. We start with the matrix M and the vector $\frac{1}{n}\mathbf{1} = (\frac{1}{n}, \frac{1}{n}, \cdots \frac{1}{n})$, where M is our PageRank matrix generated from the graph, and n is the number of vertices (passed in as `num_vertices`). The ones vector has n components, and similarly the Markov matrix M is $n \times n$.

Our result of the iterative method is the vector $M^k(\frac{1}{n}\mathbf{1})$, where k is the value passed in as `num_iterations`.

```
function pagerank_iteration(num_vertices::UInt32,
  M::Matrix{Float64}, num_iterations::UInt32)

  M_pwr = Base.power_by_squaring(M, num_iterations)
  M_pwr * (ones(Float64, num_vertices) / num_vertices)
end
```

2.3.4. *Pagerank - Epsilon Method.* In the epsilon method of our PageRank calculations (`pagerank_epsilon`), similar to the Iterative method, we start with the expression $M(\frac{1}{n}\mathbf{1}) = M(\frac{1}{n}, \frac{1}{n}, \cdots \frac{1}{n})$, where M is our PageRank matrix generated from the graph, and n is the number of vertices (passed in as `num_vertices`). The ones vector has n components, and similarly the Markov matrix M is $n \times n$.

Next, instead of multiplying $(\frac{1}{n}\mathbf{1})$ by M a fixed amount of times, we continue multiplying the vector by M on the left until the norm of the difference between the vector $M^{k+1}(1/n, 1/n, ...1/n)$ and the matrix $M^k(\frac{1}{n}, \frac{1}{n}, ...\frac{1}{n})$ is less than the caller-specified limit of epsilon.

```
function pagerank_epsilon(num_vertices::UInt32,
  M::Matrix{Float64}, epsilon::Float64)

  prev = ones(Float64, num_vertices) / num_vertices
  curr = M * prev
  while norm(prev - curr) > epsilon
    prev, curr = curr, M * curr
  end
  curr
end
```

2.4. **HITS General Algorithm.** The HITS algorithm is another algorithm that serves a similar purpose as PageRank, but provides more insight into the relationships between vertices in the directed graph. HITS assigns "hub" and "authority" scores to each of the vertices in the graph.

2.5. **Output.** For PageRank, the function `print_pagerank` prints the Vertices with the top PageRank scores to standard output.