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 to represent the the aforementioned idea.

Finally, we apply the damping factor to the matrix. Since the damping factor represents the probability that a suM_{VV_p}, coded as `M[V_p, V]`,rfer 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 vector with the PageRank scores, 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 `pagerank_iteration` function calculates PageRank with 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 Algorithm.** The HITS algorithm 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.

A vertex V has a high authority score if many other vertices point towards it, and a high hub score if it points towards many vertices with high authority scores.

2.4.1. *HITS Matrix Pair Generation.* The function `hits_matrices` generates two matrices `A` and `H` that are the authority and hub matrices, respectively.

We generate these matrices, which resemble adjacency matrices, by iterating through all of the vertices.

We construct the authority matrix `A` as follows. For each vertex V in the graph, for each of the other vertices V_p that V points to (has an outgoing edge towards), we set the matrix entry corresponding to the edge from V to V_p to 1. This entry is coded as `A[V_p, V]`.

We construct the hub matrix `H` as follows. For each vertex V in the graph, for each of the other vertices V_f that point to V, we set the matrix entry corresponding to the edge from V_f to V to 1. This entry is coded as `H[V_f, V]`.

```
function hits_matrices(graph::Graph)
  A = zeros(Float64, (graph.num_vertices, graph.num_vertices))
  H = zeros(Float64, (graph.num_vertices, graph.num_vertices))
  for vertex in graph.vertices
    for index_to in vertex.out_neighbors
      A[index_to, vertex.index] = 1
    end
    for index_from in vertex.in_neighbors
      H[index_from, vertex.index] = 1
    end
  end
  A, H
end
```

2.4.2. *HITS General Algorithm.* The function `hits` first finds the initial authority and hub HITS matrices for the user-provided graph, then applies either the iterative or the epsilon method to calculate two resultant vectors containing authority and hub scores, respectively.

`hits` takes the following parameters:

> `graph`: the user-provided directed graph
>
> `modeparam`: the mode and the parameters for HITS. Designates the calculation method we will use– either `"iter"` for the iterative method or `"epsi"` for the epsilon method.
>
> Iterative: `modeparam = ("iter", num_iterations)`: HITS for a given number of iterations Epsilon: `modeparam = ("epsi", epsilon)`: HITS until convergence of both Hub and Authority vectors with epsilon

2.4.3. *HITS - Updating Authority and Hub Scores.* To update the authority and hub scores, we refer back to the idea of what causes those scores to increase– A vertex's authority score increases if the vertices pointing towards it have higher hub scores; and a vertex's hub score increases if the vertices it points towards have high authority scores.

Thus, we have the concept behind HITS: Multiplying the hub scores vector by the authority matrix on the left gives us a new vector with the updated authority scores. This is because the product gives a vector where each component is the sum of the hub scores of the vertices that point to it.

Similarly, multiplying the authority scores vector by the hub matrix on the left gives us the updated hub scores, because each component of the resultant vector is the sum of the authority scores of the vertices that the vertex in that component points towards.

So to update the authority and hub scores, at every step we set the updated authority scores to the product of the authority matrix and current hub scores (coded as `A*h`), and the updated hub scores to the product of the hub matrix and the current authority scores (coded as `H*a`).

```
function hits_update(A::Matrix{Float64},
  H::Matrix{Float64}, a::Vector{Float64}, h::Vector{Float64})

  normalize(A * h), normalize(H * a)
end
```

2.4.4. *HITS - Iterative Method.* The `hits_iteration` function calculates HITS with the iterative method. Its parameters are the number of vertices `num_vertices`, the authority matrix `A`, the hub matrix `H`, and the number of iterations `num_iterations`.

We start with the authority and hub scores all set to the ones vector, with `num_vertices` components each.

Next, we update the authority and hub vectors, which are `a` and `h` respectively, using the function `hits_update` at every step for `num_iterations` iterations, and return the resulting two vectors of authority and hub scores.

```
function hits_iteration(num_vertices::UInt32, A::Matrix{Float64},
  H::Matrix{Float64}, num_iterations::UInt32)

  a, h = ones(Float64, num_vertices), ones(Float64, num_vertices)
  for _ in 1:num_iterations
    a, h = hits_update(A, H, a, h)
  end
  a, h
end
```

2.4.5. *HITS - Epsilon Method.* The `hits_epsilon` function calculates HITS with the epsilon method. Similar to `hits_iteration`, its parameters are the number of vertices `num_vertices`, the authority matrix `A`, the hub matrix `H`, and the convergence cutoff `epsilon`.

We start with the authority and hub scores all set to the ones vector, with `num_vertices` components each.

Next, we repeatedly update the authority and hub vectors, which are `a` and `h` respectively, using the function `hits_update` at every step until the norm between the previous and updated authority scores is less than epsilon and the norm between the previous and updated hub scores is also less than epsilon.

Finally, the resulting two vectors of authority and hub scores are returned.

```
function hits_epsilon(num_vertices::UInt32, A::Matrix{Float64},
  H::Matrix{Float64}, epsilon::Float64)

  prev_a, prev_h = ones(Float64, num_vertices), ones(Float64, num_vertices)
  curr_a, curr_h = hits_update(A, H, prev_a, prev_h)
```

```
  while norm(prev_a - curr_a) > epsilon || norm(prev_h - curr_h) > epsilon
    prev_a, prev_h, (curr_a, curr_h) = curr_a, curr_h, hits_update(A, H, curr_a, curr_h)
  end
  curr_a, curr_h
end
```

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

The parameters for `print_pagerank` are:

`graph`: the graph

`pg`: the PageRank scores for the graph

`num_lines`: the number of vertices whose information is printed

`params`: the types of information printed in order. Options are as follows:

`index`: one-based index

`0ndex`: zero-based index

`val`: PageRank score, two digits after decimal

`vall`: PageRank score, four digits after decimal

`valll`: PageRank score, six digits after decimal

`in`: number of incoming neighbors

`out`: number of outgoing neighbors

For HITS, the function `hits_print` prints the Vertices with the top HITS authority and hub scores separately to standard output in an organized format.

The parameters for `hits_print` are:

`graph`: the graph

`a`: the Authority scores for the graph

`h`: the Hub scores for the graph

`num_lines`: the number of vertices whose information is printed

`params`: the types of information printed in order. Options are as follows:

`index`: one-based index

`0ndex`: zero-based index

`val`: Authority/Hub score, two digits after decimal

`vall`: Authority/Hub score, four digits after decimal

`valll`: Authority/Hub score, six digits after decimal

`in`: print number of incoming neighbors

`out`: print number of outgoing neighbors

3. Code Appendix

ADD AT END