SCOTTYRANK.JL: AN IMPLEMENTATION OF PAGERANK & 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.

Contents

1. Background

1.1. Linear Algebra.

1.1.1. *Definitions.* Positive matrices are defined as matrices with positive entries.

Markov matrices are defined as square matrices with nonnegative entries and column sum 1 across all of its columns. Note that for a $n \times n$ matrix M, the latter condition is equivalent to $M^T\vec{1} = \vec{1}$, where $\vec{1} \in \mathbb{R}^n$ has all ones as components.

Positive Markov matrices are defined as, well, positive Markov matrices.

1.1.2. *Facts.* (Perron-Frobenius theorem) Let A be a positive square matrix. Let λ_1 be A's maximum eigenvalue in terms of absolute values. Then λ_1 is positive and has algebraic (and subsequently geometric) multiplicity 1. [7]

Let M be a Markov matrix. Let λ_1 be M's maximum eigenvalue in terms of absolute values. Then $\lambda_1 = 1$. [4]

Let M' be a positive Markov matrix. Let λ_1 be M''s maximum eigenvalue in terms of absolute values. Then $\lambda_1 = 1$ and has algebraic (and subsequently geometric) multiplicity 1.

1.1.3. *Usage.* Let M be a $n \times n$ Markov matrix. Then M specifies a dicrete memoryless transition process between n states, namely the process where

$$(\forall(t,i,j) \in \mathbb{N} \times [n] \times [n]) \left[\Pr(\text{state } i \text{ at time } t+1 \mid \text{state } j \text{ at time } t) = M_{ij}\right].$$

Let $\vec{v} \in \mathbb{R}^n$ such that $\vec{v}$ has nonnegative components and $\vec{v}^T\vec{1} = 1$ (a stochastic vector). Then $\vec{v}$ specifies an (initial) discrete probability distribution over the n states, namely the distribution where

$$(\forall i \in [n])[\Pr(\text{state } i \text{ at time } 0) = \vec{v}_i].$$

Then the probability distribution over the n states after t steps of the transition process specified by M is precisely $M^t\vec{v}$, or equivalently

$$(\forall(t,i) \in \mathbb{N} \times [n]) \left[\Pr(\text{state } i \text{ at time } t) = \left(M^t\vec{v}\right)_i\right].$$

1.2. Graph Theory.

1.2.1. *Definitions.* A simple directed graph is defined as an unweighted directed graph without self-referential edges or multiple edges between the same origin destination pair.

For a simple directed graph with n vertices, the adjacency matrix $\mathcal{A}$ is defined to be the $n \times n$ matrix where

$$(\forall(i,j) \in [n] \times [n]) \left(A_{ij} = \begin{cases} 1 & \text{there is an edge to } i \text{ from } j \\ 0 & \text{otherwise} \end{cases}\right).$$

1.2.2. *Facts.* For a simple directed graph with n vertices and its adjacency matrix $\mathcal{A}$,

$$(\forall j \in [n]) \left[\text{number of outgoing neighbors from vertex } j = \text{out}(j) = (\mathcal{A}_{*j})^T\vec{1}\right]$$
$$(\forall i \in [n]) \left[\text{number of incoming neighbors to vertex } i = \text{in}(i) = (\mathcal{A}_{i*})^T\vec{1}\right].$$

2. Algorithms

2.1. The Network Model. Both algorithms, PageRank and HITS, model the network of interest as a simple directed graph with websites as vertices and links as edges. This implies that there will be no self-referential links, no duplicate links between the same origin and destination pair, and no priority difference between links.

2.2. PageRank.

2.2.1. *The random walk.* PageRank models the behavior of a typical web surfer as a damped random walk. [6]

(1) The surfer starts out by visiting a random site out of all sites with equal probability.

(2) At every step, the surfer has a probability λ of continuing surfing and a complementary $1-\lambda$ probability of losing interest, for a predetermined λ.

   (a) If the surfer continues ...

      (i) ... and there are links exiting the current site, the surfer clicks on a random link (and visits the site it points to) out of those links with equal probability.

      (ii) ... and there aren't any links exiting the current site, the surfer simply visits a random site out of all other sites with equal probability.

   (b) If the surfer loses interest, they simply visits a random site out of all sites with equal probability.

To best model a typical surfer's probability of continuing surfing, λ, also known as the damping factor, is empirically determined to be around 0.85.

2.2.2. *Matrix representation.* Let n be the number of websites in the network of interest. Let $\mathcal{A}$ be the adjacency matrix for the network of interest. Let $\langle \vec{v}_t \rangle_{t\in\mathbb{N}}$ be the probability distributions describing the website the surfer is visiting at time t. Let M be the transition matrix for the random walk process.

Then $\vec{v}_0 = \vec{1}/n$, M is the $n \times n$ matrix where

$$(\forall (i,j) \in [n] \times [n]) \left[M_{ij} = \begin{cases} \frac{\lambda}{\mathrm{out}(j)} + \frac{1-\lambda}{n} & \mathcal{A}_{ij} = 1 \\ \frac{1-\lambda}{n} & \mathcal{A}_{ij} = 0 \wedge \mathrm{out}(j) > 0 \\ \frac{\lambda}{n-1} + \frac{1-\lambda}{n} & i \neq j \wedge \mathrm{out}(j) = 0 \\ \frac{1-\lambda}{n} & i = j \wedge \mathrm{out}(j) = 0 \end{cases} \right],$$

and

$$(\forall t \in \mathbb{N}) \left(\vec{v}_t = M^t \vec{v}_0 \right).$$

Note that in this case M is a positive Markov matrix, assuming reasonable λ.

2.2.3. *Definition.* The PageRank score for a given website in the network of interest is defined as the probabilty of a typical surfer visiting that website after an indefinitely long damped random walk. In matrix form,

$$(\forall i \in [n]) \left[\mathrm{PageRank}(i) = \lim_{t\to\infty} (\vec{v}_t)_i = \lim_{t\to\infty} \left(M^t \vec{v}_0 \right)_i \right].$$

Note that the limits exist: convergence is guaranteed as M has a unique maximal eigenvalue of 1 and thus an steady attracting state.

2.3. **HITS.**

2.3.1. *Authorities and hubs.* Due to PageRank's algorithmic design, a given website's PageRank score determined mostly by the scores of its incoming neighbors. Consequently, PageRank tends to underestimate the importance of websites similar to "web directories", i.e., those with few significant incoming neighbors yet many significant outgoing neighbors.

To address this issue, HITS (Hyperlink-Induced Topic Search) introduces Authority and Hub scores, which measure a given website's tendencies to be refered to by others and to refer to others, respectively. Note that the two metrics are not "mutually exclusive"; a website like Wikipedia can have both a high Authority score and a high Hub score.

Specifically, Authority and Hub scores are recursively defined: a website's Authority score is determined by the Hub scores of its incoming neighbors and its Hub score is determined by the Authority scores of its outgoing neighbors. [5] [3]

2.3.2. *Matrix representation.* Let n be the number of websites in the network of interest. Let $\mathcal{A}$ be the adjacency matrix for the network of interest. Let $\langle \vec{a}_t \rangle_{t\in\mathbb{N}}$ and $\langle \vec{h}_t \rangle_{t\in\mathbb{N}}$ be the (pre-normalization) Authority and Hub scores for the n websites at time t.

Then $\vec{a}_0 = \vec{h}_0 = \vec{1}$ and

$$(\forall t \in \mathbb{N}) \left[\left(\vec{a}_{t+1}, \vec{h}_{t+1} \right) = \left(\mathcal{A}\vec{h}_t, \mathcal{A}^T \vec{a}_t \right) \right].$$

2.3.3. *Definition.* The Authority and Hub scores for a given website in the network of interest is defined as the respective scores after indefinitely many iterations. In matrix form,

$$(\forall i \in [n]) \left[(\text{Authority}(i), \text{Hub}(i)) = \lim_{t\to\infty} \left((\vec{a}_t)_i, (\vec{h}_t)_i \right) \right].$$

To guarantee convergence, the Authority and Hub scores are normalized. Our implementation performs normalization after every iteration. This means

$$(\forall t \in \mathbb{N}) \left[\|(\vec{a}_t)'\| = \|(\vec{h}_t)'\| = 1 \right]$$

where

$$(\forall t \in \mathbb{N}) \left[(\vec{a}_t)' = \frac{\vec{a}_t}{\|\vec{a}_t\|} \wedge (\vec{h}_t)' = \frac{\vec{h}_t}{\|\vec{h}_t\|} \right].$$

3. Implementation

3.1. **Structs.** We define two structs, `Vertex` and `Graph`, to represent the vertices and the graph itself in our simple directed graph model for the network of interest.

Note that to align with Julia conventions, we use 1-based indexing. [1]

```
# export Vertex, Graph

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

```
end

struct Graph
  num_vertices::UInt32
  vertices::Vector{Vertex}
end
```

3.2. **Input.** We define three functions, `read_graph`, `read_edge_list`, and `read_adjacency_list`, to read and construct graphs from text files. We expose `read_graph` to the client with the option to specify the type of the input file and whether or not the input file uses 0-based indexing.

Edge list files follow the following format:

```
[num_nodes] [num_edges]
[index_from] [index_to] # repeats [num_edges] times
...                     # in total
```

Adjacency list files follow the following format:

```
[num_nodes]
[index_to_1] [index_to_2] ... [index_to_m] # repeats [num_nodes] times
...                                        # in total
```

The code for `read_graph`, `read_edge_list`, and `read_adjacency_list` can be found in the Appendix.

3.3. **PageRank.** We divide the PageRank algorithm into three steps:

(1) Generating the transition matrix: `pagerank_matrix`.

(2) Running the transition process: `pagerank_iteration`, `pagerank_epsilon`.

(3) Returning the desired output: `pagerank_print`, `pagerank`.

3.3.1. *Generating the transition matrix.* The function `pagerank_matrix` generates a Markov matrix M that specifies the transition probabilities of the PageRank transition process.

We first compute the entries in M prior to damping, casing on whether the origin vertex is a "sink" (no outgoing neighbors), and then apply the damping at the end.

```
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
```

```julia
  end
  map(x -> damping * x + (1 - damping) / graph.num_vertices, M)
end
```

3.3.2. *Running the transition process.* The functions `pagerank_iteration` and `pagerank_epsilon` both generate an initial stochastic vector and then carry out the transition process using the transition matrix.

`pagerank_iteration` runs the process for a given number of iterations.

```julia
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
```

`pagerank_epsilon` runs the process until the norm of the difference vector is smaller than a given threshold, or until $\left\|\vec{v}_{k+1} - \vec{k}\right\| < \epsilon$.

```julia
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
```

3.3.3. *Returning the desired output.* We expose two functions to the client: `pagerank` and `pagerank_print`.

`pagerank` calculates the PageRank scores for the input `graph`, with the option to specify the damping factor and the transition mode.

```julia
# export pagerank_print, pagerank

function pagerank(graph::Graph;
    damping::Float64=0.85, modeparam::Tuple{String, Union{Int64, UInt32, Float64}}=("iter", 10))
  if damping < 0 || damping > 1
    error("invalid damping")
  end
  M = pagerank_matrix(graph, damping)
  if modeparam[1] == "iter"
    if !(isinteger(modeparam[2])) || modeparam[2] < 0
      error("invalid param")
    end
    pagerank_iteration(graph.num_vertices, M, UInt32(modeparam[2]))
  elseif modeparam[1] == "epsi"
    if modeparam[2] <= 0
      error("invalid param")
    end
```

```
    pagerank_epsilon(graph.num_vertices, M, Float64(modeparam[2]))
  else
    error("invalid mode")
  end
end
```

`pagerank_print` pretty-prints the PageRank scores along with relevant information about the top vertices for the input `graph` and scores `pg`.

The code for `pagerank_print` can be found in the Appendix.

3.4. **HITS.** Similarly, we divide the HITS algorithm into three steps:

(1) Generating the transition matrix: `hits_matrix`.

(2) Running the transition process: `hits_update`, `hits_iteration`, `hits_epsilon`.

(3) Returning the desired output: `hits_print`, `hits`.

3.4.1. *Generating the transition matrix.* The transition matrices for the hits algorithm are simply the adjacency matrix and its transpose.

```
function hits_matrix(graph::Graph)
  A = 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
  end
  A
end
```

3.4.2. *Running the transition process.* The function `hits_update` computes the normalized new Authority and Hub scores from the previous Authority and Hub scores and the two transition matrices

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

The functions `hits_iteration` and `hits_epsilon` both generate the initial Authority and Hub scores and then carry out the transition process using the update function.

`hits_iteration` runs the process for a given number of iterations.

```
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
```

`hits_epsilon` runs the process until the norms of both difference vectors are smaller than the given threshold, or until $\|\vec{a}_{k+1} - \vec{a}_k\| < \epsilon \wedge \left\|\vec{h}_{k+1} - \vec{h}_k\right\| < \epsilon$.

```
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
```

3.4.3. *Returning the desired output.* We expose two functions to the client: `hits` and `hits_print`

`hits` calculates the Authority and Hub scores for the input `graph`, with the option to specify the transition mode.

```
# export hits_print, hits

function hits(graph::Graph;
    modeparam::Tuple{String, Union{Int64, UInt32, Float64}}=("iter", 10))
  A = hits_matrix(graph)
  H = copy(transpose(A))
  if modeparam[1] == "iter"
    if !(isinteger(modeparam[2])) || modeparam[2] < 0
      error("invalid param")
    end
    hits_iteration(graph.num_vertices, A, H, UInt32(modeparam[2]))
  elseif modeparam[1] == "epsi"
    if modeparam[2] <= 0
      error("invalid param")
    end
    hits_epsilon(graph.num_vertices, A, H, Float64(modeparam[2]))
  else
    error("invalid mode")
  end
end
```

`hits_print` pretty-prints the Authority and Hub scores along with relevant information about the top vertices for the input `graph` and scores `a` and `h`.

The code for `hits_print` can be found in the Appendix.

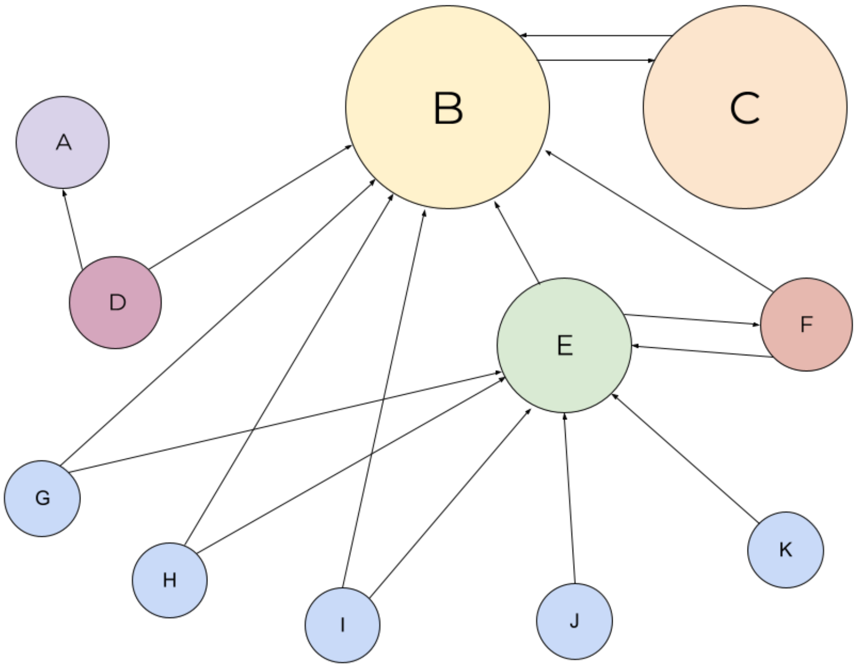
3.5. **Output.** We offer two ways of exporting the `graph` struct: `generate_adjacency_matrix` and `generate_adjacency_list`.

`generate_adjacency_matrix`, well, generates the `graph`'s adjacency matrix, with the option to specify whether the desired output should use 0-based indexing.

The code for `generate_adjacency_matrix` and `generate_adjacency_list` and be found in the Appendix.

4. Example

Shown below is a recreation of the medium-sized network with 11 websites displayed in the Wikipedia article on PageRank. [6][2] We will run both PageRank and HITS on this network.



4.1. **PageRank.**

4.1.1. *Expectations.* We expect websites B and C to have the highest PageRank scores because virtually every website links to it, either directly or through other websites such as E.

We also expect E to have a fairly high PageRank score because there are many websites linking to it.

Finally, we expect websites G, H, I, J, and K to have the lowest PageRank scores because they only have outgoing links.

4.1.2. *Results.* We run the following commands to apply PageRank on the network.

```
$ julia
julia> using ScottyRank
julia> G = read_graph("data/medium-el.txt", filetype = "el")
julia> pg = pagerank(G, modeparam=("epsi", 0.01))
julia> pagerank_print(G, pg, num_lines=11)
 - -  vall | - - index | - -    in | - -   out |
--- pagerank ---
    0.3824 |         2 |         7 |         1 | # B
    0.3467 |         3 |         1 |         1 | # C
```

```
    0.0811 |          5 |          6 |          3 | # E
    0.0392 |          4 |          1 |          2 | # D
    0.0392 |          6 |          1 |          2 | # F
    0.0303 |          1 |          1 |          0 | # A
    0.0162 |          7 |          0 |          2 | # G
    0.0162 |          8 |          0 |          2 | # H
    0.0162 |          9 |          0 |          2 | # I
    0.0162 |         10 |          0 |          1 | # J
    0.0162 |         11 |          0 |          1 | # K
```

The first column lists the calculated PageRank score and the second lists the index of the website with that score. The "in" column displays the number of incoming links to the corresponding website. Similarly, the "out" column displays the number of outgoing edges from the corresponding website.

We see that, as expected, websites B and C have the highest PageRank scores by far, and E has the highest score of the remaining websites. Additionally, we see the websites G, H, I, J, and K share the same lowest PageRank score.

4.2. **HITS.**

4.2.1. *Expectations.* Recall the core concept of HITS: websites linked by websites with high Hub scores will have high Authority scores; websites linking to websites with high Authority scores will have high Hub scores.

With this in mind, we expect that websites B and E to have the highest Authority scores simply because they have the greatest number of websites linking to them.

However, since B does not link towards many other Authorities, its Hub score should be low. But E links to B, which has a high Authority score, so we expect the Hub score of E to be above average.

Similarly, the Hub score of C should be relatively high since it links to B, but the Authority score of C should be low because B does not have a high Hub score.

Finally, we expect the Authority scores of G, H, I, J, and K all to be very low because no websites link towards them. However, their Hub scores should be relatively high because they point to one or both of the highest Authority websites, namely B and E.

4.2.2. *Results.* We run the following commands to apply HITS on the network.

```
$ julia
julia> using ScottyRank
julia> G = read_graph("data/medium-el.txt", filetype = "el")
julia> a, h = hits(G, modeparam=("epsi", 0.01))
julia> hits_print(G, a, h, num_lines=11)
 - -  vall | - - index | - -    in | - -   out |
--- authority ---
    0.7567 |          2 |          7 |          1 | # B
    0.6370 |          5 |          6 |          3 | # E
    0.0880 |          4 |          1 |          2 | # D
    0.0880 |          6 |          1 |          2 | # F
    0.0784 |          1 |          1 |          0 | # A
    0.0000 |          3 |          1 |          1 | # C
    0.0000 |          7 |          0 |          2 | # G
```

```
    0.0000 |         8 |         0 |         2 | # H
    0.0000 |         9 |         0 |         2 | # I
    0.0000 |        10 |         0 |         1 | # J
    0.0000 |        11 |         0 |         1 | # K
--- hub ---
    0.4259 |         6 |         1 |         2 | # F
    0.4259 |         7 |         0 |         2 | # G
    0.4259 |         8 |         0 |         2 | # H
    0.4259 |         9 |         0 |         2 | # I
    0.2836 |         5 |         6 |         3 | # E
    0.2544 |         4 |         1 |         2 | # D
    0.2306 |         3 |         1 |         1 | # C
    0.1952 |        10 |         0 |         1 | # J
    0.1952 |        11 |         0 |         1 | # K
    0.0000 |         2 |         7 |         1 | # B
    0.0000 |         1 |         1 |         0 | # A
```

Similar to `pagerank_print`, the first column of the first table lists the Authority score and the first column of the second table lists the Hub scores. The "in" column displays the number of incoming links to the corresponding website, and the "out" column displays the number of outgoing edges from the corresponding website.

As expected, the Authority scores of B and E are the highest, with C having a low Authority score. The vertices G, H, I, J, and K have Authority scores of 0 because no vertices point to them.

Furthermore, we see that the Hub scores are also calculated as predicted: the Hub score of F is the highest because F links towards both B and E, the two vertices with the highest authority scores. G, H, I also have very high Hub scores. However, J and K have lower Hub scores because they only link to one of the main Authorities.

References

[1] Julia contributors. Linear algebra. https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/.

[2] Leskovec, J., and Krevl, A. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[3] Radu, R., and Tanase, R. Lecture 4: Hits algorithm - hubs and authorities on the internet. http://pi.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture4/lecture4.html.

[4] Strang, G. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, 2016.

[5] Wikipedia contributors. Hits algorithm — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=HITS_algorithm&oldid=1008455356, 2021. [Online; accessed 3-December-2021].

[6] Wikipedia contributors. Pagerank — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=PageRank&oldid=1058274873, 2021. [Online; accessed 3-December-2021].

[7] Wikipedia contributors. Perron–frobenius theorem — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Perron%E2%80%93Frobenius_theorem&oldid=1058136598, 2021. [Online; accessed 3-December-2021].

Appendix A. Links

The project repository can be found at https://github.com/mzhou08/ScottyRank.jl.

Appendix B. Code

```julia
"""
    module ScottyRank

Provides PageRank and HITS analysis functionalities for directed graphs.

Siyuan Chen & Michael Zhou, November 2021.
"""
module ScottyRank

using DelimitedFiles
using LinearAlgebra
using Printf

export Vertex, Graph
export read_graph
export pagerank_print, pagerank
export hits_print, hits
export generate_adjacency_matrix, generate_adjacency_list

# export Vertex, Graph

"""
    Vertex

ScottyRank vertex

# Fields
- `index::UInt32`: stores the 1-based index as an unsigned integer
- `in_neighbors::Vector{UInt32}`: stores the indices of incoming neighbors as a list
- `out_neighbors::Vector{UInt32}`: stores the indices of outgoing neighbors as a list
"""
struct Vertex
  index::UInt32
  in_neighbors::Vector{UInt32}
  out_neighbors::Vector{UInt32}
end

"""
    Graph

ScottyRank graph

# Fields
- `num_vertices`: stores the number of vertices as an unsigned integer
- `vertices::Vector{Vertex}`: stores the vertices as a sorted list
"""
```

```julia
struct Graph
  num_vertices::UInt32
  vertices::Vector{Vertex}
end

# export read_graph

"""
    read_graph(filepath::String="data/medium-el.txt"); filetype::String="el", zero_index::Bool=false) -> Graph

Reads a graph from an edge list/adjacency list file

# Arguments
- `filepath::String="data/medium-el.txt"`: the path to the source file (default: Wikipedia PageRank graph)

# Keywords
- `filetype::String="el"`: "el" for edge list, "al" for adjacency list
- `zero_index::Bool=false`: whether the input file is zero-based

# Returns
- `Graph`: the graph from the source file
"""
function read_graph(filepath::String="data/medium-el.txt";
    filetype::String="el", zero_index::Bool=false)
  if filetype == "el"
    read_edge_list(filepath, zero_index)
  elseif filetype == "al"
    read_adjacency_list(filepath, zero_index)
  else
    error("invalid filetype")
  end
end

function read_edge_list(filepath::String, zero_index::Bool)
  file = open(filepath)
  num_vertices, num_edges = map(x -> convert(UInt32, x), readdlm(IOBuffer(readline(file))))
  vertices = Array{Vertex}(undef, num_vertices)
  for i in 1:num_vertices
    vertices[i] = Vertex(i, Array{UInt32}(undef, 0), Array{UInt32}(undef, 0))
  end
  for _ in 1:num_edges
    index_from, index_to = map(x -> convert(UInt32, x), readdlm(IOBuffer(readline(file))))
    if zero_index
      push!(vertices[index_from + 1].out_neighbors, index_to + 1)
```

```
      push!(vertices[index_to + 1].in_neighbors, index_from + 1)
    else
      push!(vertices[index_from].out_neighbors, index_to)
      push!(vertices[index_to].in_neighbors, index_from)
    end
  end
  close(file)
  Graph(num_vertices, vertices)
end

function read_adjacency_list(filepath::String, zero_index::Bool)
  file = open(filepath)
  num_vertices = parse(UInt32, readline(file))
  vertices = Array{Vertex}(undef, num_vertices)
  for i in 1:num_vertices
    vertices[i] = Vertex(i, Array{UInt32}(undef, 0), Array{UInt32}(undef, 0))
  end
  for index_from in 1:num_vertices, index_to in map(x -> convert(UInt32, x), readdlm(IOBuffer(readline(file))))
    if zero_index
      push!(vertices[index_from].out_neighbors, index_to + 1)
      push!(vertices[index_to + 1].in_neighbors, index_from)
    else
      push!(vertices[index_from].out_neighbors, index_to)
      push!(vertices[index_to].in_neighbors, index_from)
    end
  end
  close(file)
  Graph(num_vertices, vertices)
end

#  export pagerank_print, pagerank

"""
    function pagerank_print(graph::Graph, pg::Vector{Float64};
        num_lines::Union{Int64, UInt32}=10, params::Vector{String}=String["vall", "index", "in", "out"]
      ) -> Nothing

Pretty-prints information about the vertices with top PageRank scores to stdout

# Arguments
- `graph::Graph`: the graph
- `pg::Vector{Float64}`: the PageRank scores for the graph

# Keywords
```

```
- `num_lines::Union{Int64, UInt32}=10`: the number of vertices whose information is printed
- `params::Vector{String}=String["vall", "index", "in", "out"]`: the types of information printed in order
  - `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

# Returns
- `Nothing`
"""
function pagerank_print(graph::Graph, pg::Vector{Float64};
    num_lines::Union{Int64, UInt32}=10, params::Vector{String}=String["vall", "index", "in", "out"])
  if num_lines > graph.num_vertices
    error("invalid num_lines")
  end
  perm = sortperm(pg, rev=true)
  for param in params
    @printf(" - - %5s |", param)
  end
  println()
  println("--- pagerank ---")
  for line in 1:num_lines
    for param in params
      if param == "index"
        @printf("%10d |", perm[line])
      elseif param == "0ndex"
        @printf("%10d |", perm[line] - 1)
      elseif param == "val"
        @printf("%10.2f |", pg[perm[line]])
      elseif param == "vall"
        @printf("%10.4f |", pg[perm[line]])
      elseif param == "valll"
        @printf("%10.6f |", pg[perm[line]])
      elseif param == "in"
        @printf("%10d |", length(graph.vertices[perm[line]].in_neighbors))
      elseif param == "out"
        @printf("%10d |", length(graph.vertices[perm[line]].out_neighbors))
      else
        error("invalid param")
      end
    end
```

```julia
    println()
  end
end

"""
    function pagerank(graph::Graph;
        damping::Float64=0.85, modeparam::Tuple{String, Union{Int64, UInt32, Float64}}=("iter", 10)
      ) -> Vector{Float64}

Computes PageRank scores for the graph

# Arguments
- `graph::Graph`: the graph

# Keywords
- `damping::Float64=0.85`: the damping factor for PageRank
- `modeparam::Tuple{String, Union{Int64, UInt32, Float64}}=("iter", 10)`: the mode and the parameters for PageRank
  - `("iter", num_iterations::Union{Int64, UInt32})`: PageRank for a given number of iterations
  - `("epsi", epsilon::Union{Int64, UInt32, Float64})`: PageRank until convergence with epsilon

# Returns
- `Vector{Float64}`: the PageRank scores for the graph
"""
function pagerank(graph::Graph;
    damping::Float64=0.85, modeparam::Tuple{String, Union{Int64, UInt32, Float64}}=("iter", 10))
  if damping < 0 || damping > 1
    error("invalid damping")
  end
  M = pagerank_matrix(graph, damping)
  if modeparam[1] == "iter"
    if !(isinteger(modeparam[2])) || modeparam[2] < 0
      error("invalid param")
    end
    pagerank_iteration(graph.num_vertices, M, UInt32(modeparam[2]))
  elseif modeparam[1] == "epsi"
    if modeparam[2] <= 0
      error("invalid param")
    end
    pagerank_epsilon(graph.num_vertices, M, Float64(modeparam[2]))
  else
    error("invalid mode")
  end
end
```

```
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

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

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

# export hits_print, hits

"""
    function hits_print(graph::Graph, a::Vector{Float64}, h::Vector{Float64};
        num_lines::Union{Int64, UInt32}=10, params::Vector{String}=String["vall", "index", "in", "out"]
      ) -> Nothing

Pretty-prints information about the vertices with top Authority and Hub scores (separately) to stdout

# Arguments
- `graph::Graph`: the graph
- `a::Vector{Float64}`: the Authority scores for the graph
```

```
- `h::Vector{Float64}`: the Hub scores for the graph

# Keywords
- `num_lines::Union{Int64, UInt32}=10`: the number of vertices whose information is printed
- `params::Vector{String}=String["vall", "index", "in", "out"]`: the types of information printed in order
  - `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`: number of incoming neighbors
  - `out`: number of outgoing neighbors

# Returns
- `Nothing`
"""
function hits_print(graph::Graph, a::Vector{Float64}, h::Vector{Float64};
    num_lines::Union{Int64, UInt32}=10, params::Vector{String}=String["vall", "index", "in", "out"])
  if num_lines > graph.num_vertices
    error("invalid num_lines")
  end
  perm_a = sortperm(a, rev=true)
  perm_h = sortperm(h, rev=true)
  for param in params
    @printf(" - - %5s |", param)
  end
  println()
  println("--- authority ---")
  for line in 1:num_lines
    for param in params
      if param == "index"
        @printf("%10d |", perm_a[line])
      elseif param == "0ndex"
        @printf("%10d |", perm_a[line] - 1)
      elseif param == "val"
        @printf("%10.2f |", a[perm_a[line]])
      elseif param == "vall"
        @printf("%10.4f |", a[perm_a[line]])
      elseif param == "valll"
        @printf("%10.6f |", a[perm_a[line]])
      elseif param == "in"
        @printf("%10d |", length(graph.vertices[perm_a[line]].in_neighbors))
      elseif param == "out"
        @printf("%10d |", length(graph.vertices[perm_a[line]].out_neighbors))
```

```julia
      else
        error("invalid param")
      end
    end
    println()
  end
  println("--- hub ---")
  for line in 1:num_lines
    for param in params
      if param == "index"
        @printf("%10d |", perm_h[line])
      elseif param == "0ndex"
        @printf("%10d |", perm_h[line] - 1)
      elseif param == "val"
        @printf("%10.2f |", h[perm_h[line]])
      elseif param == "vall"
        @printf("%10.4f |", h[perm_h[line]])
      elseif param == "valll"
        @printf("%10.6f |", h[perm_h[line]])
      elseif param == "in"
        @printf("%10d |", length(graph.vertices[perm_h[line]].in_neighbors))
      elseif param == "out"
        @printf("%10d |", length(graph.vertices[perm_h[line]].out_neighbors))
      else
        error("invalid param")
      end
    end
    println()
  end
end

"""
    function hits(graph::Graph;
        modeparam::Tuple{String, Union{Int64, UInt32, Float64}}=("iter", 10)
      ) -> Tuple{Vector{Float64}, Vector{Float64}}

Computes Hub and Authority scores (HITS) for the graph

# Arguments
- `graph::Graph`: the graph

# Keywords
- `modeparam::Tuple{String, Union{Int64, UInt32, Float64}}=("iter", 10)`: the mode and the parameters for HITS
  - `("iter", num_iterations::Union{Int64, UInt32})`: HITS for a given number of iterations
```

```julia
  - `("epsi", epsilon::Union{Int64, UInt32, Float64})`: HITS until convergence with epsilon (both Hub and Authority)

# Returns
- `Tuple{Vector{Float64}, Vector{Float64}}`: the Hub and Authoriy scores for the graph
"""
function hits(graph::Graph;
    modeparam::Tuple{String, Union{Int64, UInt32, Float64}}=("iter", 10))
  A = hits_matrix(graph)
  H = copy(transpose(A))
  if modeparam[1] == "iter"
    if !(isinteger(modeparam[2])) || modeparam[2] < 0
      error("invalid param")
    end
    hits_iteration(graph.num_vertices, A, H, UInt32(modeparam[2]))
  elseif modeparam[1] == "epsi"
    if modeparam[2] <= 0
      error("invalid param")
    end
    hits_epsilon(graph.num_vertices, A, H, Float64(modeparam[2]))
  else
    error("invalid mode")
  end
end

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

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
```

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

function hits_matrix(graph::Graph)
  A = 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
  end
  A
end

# export generate_adjacency_matrix, generate_adjacency_list

"""
    function generate_adjacency_matrix(graph::Graph) -> Matrix{Bool}

Generates the adjacency matrix representation for the graph

# Arguments
- `graph::Graph`: the graph

# Returns
- `Matrix{Bool}`: the adjacency matrix representation for the graph
"""
function generate_adjacency_matrix(graph::Graph)
  AM = zeros(Bool, (graph.num_vertices, graph.num_vertices))
  for vertex in graph.vertices, index_to in vertex.out_neighbors
    AM[vertex.index_to, vertex.index] = true
  end
  AM
end

"""
    function generate_adjacency_list(graph::Graph) -> Vector{Vector{UInt32}}

Generates the adjacency list representation for the graph

# Arguments
- `graph::Graph`: the graph

# Keywords
- `zero_index::Bool=false`: whether the output file is zero-based

# Returns
- `Vector{Vector{Bool}}`: the adjacency list representation for the graph
```

```
"""
function generate_adjacency_list(graph::Graph; zero_index::Bool=false)
  AL = Array{Vector{UInt32}}(undef, graph.num_vertices)
  for vertex in graph.vertices
    AL[vertex.index] = Array{UInt32}(undef, 0)
    for index_to in vertex.out_neighbors
      if zero_index
        push!(AL[vertex.index], index_to - 1)
      else
        push!(AL[vertex.index], index_to)
      end
    end
  end
  AL
end

end
```