

SCOTTYRANK.JL: AN IMPLEMENTATION OF PAGERANK & HITS

SIYUAN CHEN AND MICHAEL ZHOU

ABSTRACT. We present [ScottyRank.jl](#), a Julia implementation of the PageRank algorithm and one of its successors, the HITS algorithm. Famously used by Google, PageRank, as well as its variations, models a network of interest (such as the Internet) as a simple directed graph and produces relative rankings of the vertices that roughly translate to the vertices' levels of relevance and importance.

In this paper, we first introduced the mathematical background for the two algorithms. We then outlined the core algorithmic insights of both PageRank and HITS and expressed them in matrix form. Using code segments, we presented our implementation of the two algorithms. Finally, we ran the two algorithms on the Wikipedia PageRank network and analyzed the outputs. [6] We included the source code in the Appendix.

CONTENTS

1. Background	3
1.1. Linear Algebra	3
1.2. Graph Theory	3
2. Algorithms	4
2.1. The Network Model	4
2.2. PageRank	4
2.3. HITS	5
3. Implementation	5
3.1. Structs	5
3.2. Input	6
3.3. PageRank	6
3.4. HITS	8
3.5. Output	9
4. Example	10
4.1. PageRank	10
4.2. HITS	11
References	12
Appendix A. Links	12
Appendix B. Code	13

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 discrete memoryless transition process between n states, namely the process where

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

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

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

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} ,

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

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

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

and

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

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 probability of a typical surfer visiting that website after an indefinitely long damped random walk. In matrix form,

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

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 referred 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

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

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,

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

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]

```

1  # export Vertex, Graph
2
3  struct Vertex
4      index::UInt32
5      in_neighbors::Vector{UInt32}
6      out_neighbors::Vector{UInt32}

```

```

7  end
8
9  struct Graph
10     num_vertices::UInt32
11     vertices::Vector{Vertex}
12 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:

```

1  [num_nodes] [num_edges]
2  [index_from] [index_to] # repeats [num_edges] times
3  ...                  # in total

```

Adjacency list files follow the following format:

```

1  [num_nodes]
2  [index_to_1] [index_to_2] ... [index_to_m] # repeats [num_nodes] times
3  ...                  # 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.

```

1  function pagerank_matrix(graph::Graph, damping::Float64)
2      M = zeros(Float64, (graph.num_vertices, graph.num_vertices))
3      for vertex in graph.vertices
4          num_out_neighbors = length(vertex.out_neighbors)
5          if num_out_neighbors == 0
6              for index_to in 1:graph.num_vertices
7                  M[index_to, vertex.index] = 1 / (graph.num_vertices - 1)
8              end
9              M[vertex.index, vertex.index] = 0
10         else
11             for index_to in vertex.out_neighbors
12                 M[index_to, vertex.index] = 1 / num_out_neighbors
13             end
14         end

```

```

15     end
16     map(x -> damping * x + (1 - damping) / graph.num_vertices, M)
17 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.

```

1 function pagerank_iteration(num_vertices::UInt32, M::Matrix{Float64},
   ↪ num_iterations::UInt32)
2     M_pwr = Base.power_by_squaring(M, num_iterations)
3     M_pwr * (ones(Float64, num_vertices) / num_vertices)
4 end

```

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

```

1 function pagerank_epsilon(num_vertices::UInt32, M::Matrix{Float64},
   ↪ epsilon::Float64)
2     prev = ones(Float64, num_vertices) / num_vertices
3     curr = M * prev
4     while norm(prev - curr) > epsilon
5         prev, curr = curr, M * curr
6     end
7     curr
8 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.

```

1 # export pagerank_print, pagerank
2
3 function pagerank(graph::Graph;
4     damping::Float64=0.85, modeparam::Tuple{String, Union{Int64, UInt32,
   ↪ Float64}}=("iter", 10))
5     if damping < 0 || damping > 1
6         error("invalid damping")
7     end
8     M = pagerank_matrix(graph, damping)
9     if modeparam[1] == "iter"
10         if !(isinteger(modeparam[2])) || modeparam[2] < 0
11             error("invalid param")
12         end
13         pagerank_iteration(graph.num_vertices, M, UInt32(modeparam[2]))
14     elseif modeparam[1] == "epsi"
15         if modeparam[2] <= 0
16             error("invalid param")
17         end
18     end
19 end

```

```

18     pagerank_epsilon(graph.num_vertices, M, Float64(modeparam[2]))
19 else
20     error("invalid mode")
21 end
22 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.

```

1 function hits_matrix(graph::Graph)
2     A = zeros(Float64, (graph.num_vertices, graph.num_vertices))
3     for vertex in graph.vertices
4         for index_to in vertex.out_neighbors
5             A[index_to, vertex.index] = 1
6         end
7     end
8     A
9 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

```

1 function hits_update(A::Matrix{Float64}, H::Matrix{Float64},
2     ↪ a::Vector{Float64}, h::Vector{Float64})
3     normalize(A * h), normalize(H * a)
4 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.

```

1 function hits_iteration(num_vertices::UInt32, A::Matrix{Float64},
2     ↪ H::Matrix{Float64}, num_iterations::UInt32)
3     a, h = ones(Float64, num_vertices), ones(Float64, num_vertices)
4     for _ in 1:num_iterations
5         a, h = hits_update(A, H, a, h)
6     end
7     a, h
8 end

```


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

```

1 function hits_epsilon(num_vertices::UInt32, A::Matrix{Float64},
  ↪ H::Matrix{Float64}, epsilon::Float64)
2   prev_a, prev_h = ones(Float64, num_vertices), ones(Float64,
  ↪ num_vertices)
3   curr_a, curr_h = hits_update(A, H, prev_a, prev_h)
4   while norm(prev_a - curr_a) > epsilon || norm(prev_h - curr_h) >
  ↪ epsilon
5     prev_a, prev_h, (curr_a, curr_h) = curr_a, curr_h, hits_update(A, H,
  ↪ curr_a, curr_h)
6   end
7   curr_a, curr_h
8 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.

```

1 # export hits_print, hits
2
3 function hits(graph::Graph;
4   modeparam::Tuple{String, Union{Int64, UInt32, Float64}}=("iter", 10))
5   A = hits_matrix(graph)
6   H = copy(transpose(A))
7   if modeparam[1] == "iter"
8     if !(isinteger(modeparam[2])) || modeparam[2] < 0
9       error("invalid param")
10    end
11    hits_iteration(graph.num_vertices, A, H, UInt32(modeparam[2]))
12  elseif modeparam[1] == "epsi"
13    if modeparam[2] <= 0
14      error("invalid param")
15    end
16    hits_epsilon(graph.num_vertices, A, H, Float64(modeparam[2]))
17  else
18    error("invalid mode")
19  end
20 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.

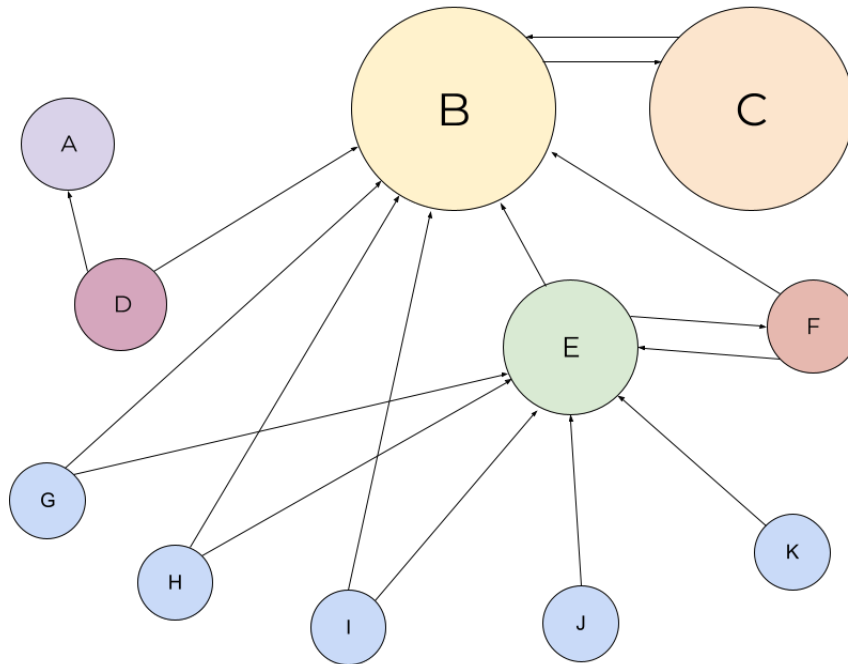
3.5. **Output.** We offer two ways of exporting the `graph` struct: `generate_adjacency_matrix` and `generate_adjacency_list`.

`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` can 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

```

1  """
2      module ScottyRank
3
4  Provides PageRank and HITS analysis functionalities for directed graphs.
5
6  Siyuan Chen & Michael Zhou, November 2021.
7  """
8  module ScottyRank
9
10 using DelimitedFiles
11 using LinearAlgebra
12 using Printf
13
14 export Vertex, Graph
15 export read_graph
16 export pagerank_print, pagerank
17 export hits_print, hits
18 export generate_adjacency_matrix, generate_adjacency_list
19
20 # export Vertex, Graph
21
22 """
23     Vertex
24
25 ScottyRank vertex
26
27 # Fields
28 - `index::UInt32`: stores the 1-based index as an unsigned integer
29 - `in_neighbors::Vector{UInt32}`: stores the indices of incoming
    ↪ neighbors as a list
30 - `out_neighbors::Vector{UInt32}`: stores the indices of outgoing
    ↪ neighbors as a list
31 """
32 struct Vertex
33     index::UInt32
34     in_neighbors::Vector{UInt32}
35     out_neighbors::Vector{UInt32}
36 end
37
38 """
39     Graph
40
41 ScottyRank graph
42
43 # Fields
44 - `num_vertices`: stores the number of vertices as an unsigned integer
45 - `vertices::Vector{Vertex}`: stores the vertices as a sorted list
46 """

```

```

47 struct Graph
48     num_vertices::UInt32
49     vertices::Vector{Vertex}
50 end
51
52 # export read_graph
53
54 """
55     read_graph(filepath::String="data/medium-el.txt");
56     ↪ filetype::String="el", zero_index::Bool=false) -> Graph
57
58 Reads a graph from an edge list/adjacency list file
59
60 # Arguments
61 - `filepath::String="data/medium-el.txt"`: the path to the source file
62     ↪ (default: Wikipedia PageRank graph)
63
64 # Keywords
65 - `filetype::String="el"`: "el" for edge list, "al" for adjacency list
66 - `zero_index::Bool=false`: whether the input file is zero-based
67
68 # Returns
69 - `Graph`: the graph from the source file
70 """
71 function read_graph(filepath::String="data/medium-el.txt";
72     filetype::String="el", zero_index::Bool=false)
73     if filetype == "el"
74         read_edge_list(filepath, zero_index)
75     elseif filetype == "al"
76         read_adjacency_list(filepath, zero_index)
77     else
78         error("invalid filetype")
79     end
80 end
81
82 function read_edge_list(filepath::String, zero_index::Bool)
83     file = open(filepath)
84     num_vertices, num_edges = map(x -> convert(UInt32, x),
85     ↪ readlm(IOBuffer(readline(file))))
86     vertices = Array{Vertex}(undef, num_vertices)
87     for i in 1:num_vertices
88         vertices[i] = Vertex(i, Array{UInt32}(undef, 0), Array{UInt32}(undef,
89     ↪ 0))
90     end
91     for _ in 1:num_edges
92         index_from, index_to = map(x -> convert(UInt32, x),
93     ↪ readlm(IOBuffer(readline(file))))
94         if zero_index
95             push!(vertices[index_from + 1].out_neighbors, index_to + 1)

```

```

91     push!(vertices[index_to + 1].in_neighbors, index_from + 1)
92 else
93     push!(vertices[index_from].out_neighbors, index_to)
94     push!(vertices[index_to].in_neighbors, index_from)
95 end
96 end
97 close(file)
98 Graph(num_vertices, vertices)
99 end
100
101 function read_adjacency_list(filepath::String, zero_index::Bool)
102     file = open(filepath)
103     num_vertices = parse{UInt32, readline(file)}
104     vertices = Array{Vertex}(undef, num_vertices)
105     for i in 1:num_vertices
106         vertices[i] = Vertex(i, Array{UInt32}(undef, 0), Array{UInt32}(undef,
107             ↪ 0))
108     end
109     for index_from in 1:num_vertices, index_to in map(x -> convert{UInt32,
110         ↪ x), readdlm{IOBuffer(readline(file))})
111         if zero_index
112             push!(vertices[index_from].out_neighbors, index_to + 1)
113             push!(vertices[index_to + 1].in_neighbors, index_from)
114         else
115             push!(vertices[index_from].out_neighbors, index_to)
116             push!(vertices[index_to].in_neighbors, index_from)
117         end
118     end
119     close(file)
120     Graph(num_vertices, vertices)
121 end
122
123 # export pagerank_print, pagerank
124
125 """
126     function pagerank_print(graph::Graph, pg::Vector{Float64};
127         num_lines::Union{Int64, UInt32}=10,
128         ↪ params::Vector{String}=String["vall", "index", "in", "out"]
129         ) -> Nothing
130
131     Pretty-prints information about the vertices with top PageRank scores to
132     ↪ stdout
133
134     # Arguments
135     - `graph::Graph`: the graph
136     - `pg::Vector{Float64}`: the PageRank scores for the graph
137
138     # Keywords

```

```

135 - `num_lines::Union{Int64, UInt32}=10`: the number of vertices whose
    ↪ information is printed
136 - `params::Vector{String}=String["vall", "index", "in", "out"]`: the
    ↪ types of information printed in order
137 - `index`: one-based index
138 - `0ndex`: zero-based index
139 - `val`: PageRank score, two digits after decimal
140 - `vall`: PageRank score, four digits after decimal
141 - `valll`: PageRank score, six digits after decimal
142 - `in`: number of incoming neighbors
143 - `out`: number of outgoing neighbors
144
145 # Returns
146 - `Nothing`
147 """
148 function pagerank_print(graph::Graph, pg::Vector{Float64};
149     num_lines::Union{Int64, UInt32}=10,
150     ↪ params::Vector{String}=String["vall", "index", "in", "out"])
151 if num_lines > graph.num_vertices
152     error("invalid num_lines")
153 end
154 perm = sortperm(pg, rev=true)
155 for param in params
156     @printf(" - - %5s |", param)
157 end
158 println()
159 println("---- pagerank ----")
160 for line in 1:num_lines
161     for param in params
162         if param == "index"
163             @printf("%10d |", perm[line])
164         elseif param == "0ndex"
165             @printf("%10d |", perm[line] - 1)
166         elseif param == "val"
167             @printf("%10.2f |", pg[perm[line]])
168         elseif param == "vall"
169             @printf("%10.4f |", pg[perm[line]])
170         elseif param == "valll"
171             @printf("%10.6f |", pg[perm[line]])
172         elseif param == "in"
173             @printf("%10d |",
174                 ↪ length(graph.vertices[perm[line]].in_neighbors))
175         elseif param == "out"
176             @printf("%10d |",
177                 ↪ length(graph.vertices[perm[line]].out_neighbors))
178         else
179             error("invalid param")
180         end
181     end
182 end

```



```

179     println()
180 end
181 end
182
183 """
184     function pagerank(graph::Graph;
185         damping::Float64=0.85, modeparam::Tuple{String, Union{Int64,
186     ↪ UInt32, Float64}}=("iter", 10)
187         ) -> Vector{Float64}
188
189 Computes PageRank scores for the graph
190
191 # Arguments
192 - `graph::Graph`: the graph
193
194 # Keywords
195 - `damping::Float64=0.85`: the damping factor for PageRank
196 - `modeparam::Tuple{String, Union{Int64, UInt32, Float64}}=("iter", 10)`:
197     ↪ the mode and the parameters for PageRank
198 - `("iter", num_iterations::Union{Int64, UInt32})`: PageRank for a
199     ↪ given number of iterations
200 - `("epsi", epsilon::Union{Int64, UInt32, Float64})`: PageRank until
201     ↪ convergence with epsilon
202
203 # Returns
204 - `Vector{Float64}`: the PageRank scores for the graph
205 """
206 function pagerank(graph::Graph;
207     damping::Float64=0.85, modeparam::Tuple{String, Union{Int64, UInt32,
208     ↪ Float64}}=("iter", 10))
209     if damping < 0 || damping > 1
210         error("invalid damping")
211     end
212     M = pagerank_matrix(graph, damping)
213     if modeparam[1] == "iter"
214         if !(isinteger(modeparam[2])) || modeparam[2] < 0
215             error("invalid param")
216         end
217         pagerank_iteration(graph.num_vertices, M, UInt32(modeparam[2]))
218     elseif modeparam[1] == "epsi"
219         if modeparam[2] <= 0
220             error("invalid param")
221         end
222         pagerank_epsilon(graph.num_vertices, M, Float64(modeparam[2]))
223     else
224         error("invalid mode")
225     end
226 end
227 end
228

```

```

223 function pagerank_iteration(num_vertices::UInt32, M::Matrix{Float64},
    ↪ num_iterations::UInt32)
224     M_pwr = Base.power_by_squaring(M, num_iterations)
225     M_pwr * (ones(Float64, num_vertices) / num_vertices)
226 end
227
228 function pagerank_epsilon(num_vertices::UInt32, M::Matrix{Float64},
    ↪ epsilon::Float64)
229     prev = ones(Float64, num_vertices) / num_vertices
230     curr = M * prev
231     while norm(prev - curr) > epsilon
232         prev, curr = curr, M * curr
233     end
234     curr
235 end
236
237 function pagerank_matrix(graph::Graph, damping::Float64)
238     M = zeros(Float64, (graph.num_vertices, graph.num_vertices))
239     for vertex in graph.vertices
240         num_out_neighbors = length(vertex.out_neighbors)
241         if num_out_neighbors == 0
242             for index_to in 1:graph.num_vertices
243                 M[index_to, vertex.index] = 1 / (graph.num_vertices - 1)
244             end
245             M[vertex.index, vertex.index] = 0
246         else
247             for index_to in vertex.out_neighbors
248                 M[index_to, vertex.index] = 1 / num_out_neighbors
249             end
250         end
251     end
252     map(x -> damping * x + (1 - damping) / graph.num_vertices, M)
253 end
254
255 # export hits_print, hits
256
257 """
258     function hits_print(graph::Graph, a::Vector{Float64},
    ↪ h::Vector{Float64};
259         num_lines::Union{Int64, UInt32}=10,
    ↪ params::Vector{String}=String["vall", "index", "in", "out"]
260         ) -> Nothing
261
262 Pretty-prints information about the vertices with top Authority and Hub
    ↪ scores (separately) to stdout
263
264 # Arguments
265 - `graph::Graph`: the graph
266 - `a::Vector{Float64}`: the Authority scores for the graph

```

```

267 - `h::Vector{Float64}`: the Hub scores for the graph
268
269 # Keywords
270 - `num_lines::Union{Int64, UInt32}=10`: the number of vertices whose
    ↪ information is printed
271 - `params::Vector{String}=String["vall", "index", "in", "out"]`: the
    ↪ types of information printed in order
272   - `index`: one-based index
273   - `Ondex`: zero-based index
274   - `val`: Authority/Hub score, two digits after decimal
275   - `vall`: Authority/Hub score, four digits after decimal
276   - `valll`: Authority/Hub score, six digits after decimal
277   - `in`: number of incoming neighbors
278   - `out`: number of outgoing neighbors
279
280 # Returns
281 - `Nothing`
282 ""
283 function hits_print(graph::Graph, a::Vector{Float64}, h::Vector{Float64};
284     num_lines::Union{Int64, UInt32}=10,
    ↪ params::Vector{String}=String["vall", "index", "in", "out"])
285 if num_lines > graph.num_vertices
286     error("invalid num_lines")
287 end
288 perm_a = sortperm(a, rev=true)
289 perm_h = sortperm(h, rev=true)
290 for param in params
291     @printf(" - - %5s |", param)
292 end
293 println()
294 println("--- authority ---")
295 for line in 1:num_lines
296     for param in params
297         if param == "index"
298             @printf("%10d |", perm_a[line])
299         elseif param == "Ondex"
300             @printf("%10d |", perm_a[line] - 1)
301         elseif param == "val"
302             @printf("%10.2f |", a[perm_a[line]])
303         elseif param == "vall"
304             @printf("%10.4f |", a[perm_a[line]])
305         elseif param == "valll"
306             @printf("%10.6f |", a[perm_a[line]])
307         elseif param == "in"
308             @printf("%10d |",
    ↪ length(graph.vertices[perm_a[line]].in_neighbors))
309         elseif param == "out"
310             @printf("%10d |",
    ↪ length(graph.vertices[perm_a[line]].out_neighbors))

```

```

311     else
312         error("invalid param")
313     end
314 end
315 println()
316 end
317 println("--- hub ---")
318 for line in 1:num_lines
319     for param in params
320         if param == "index"
321             @printf("%10d |", perm_h[line])
322         elseif param == "0ndex"
323             @printf("%10d |", perm_h[line] - 1)
324         elseif param == "val"
325             @printf("%10.2f |", h[perm_h[line]])
326         elseif param == "vall"
327             @printf("%10.4f |", h[perm_h[line]])
328         elseif param == "valll"
329             @printf("%10.6f |", h[perm_h[line]])
330         elseif param == "in"
331             @printf("%10d |",
332                 ↪ length(graph.vertices[perm_h[line]].in_neighbors))
333         elseif param == "out"
334             @printf("%10d |",
335                 ↪ length(graph.vertices[perm_h[line]].out_neighbors))
336         else
337             error("invalid param")
338         end
339     end
340 end
341
342 """
343     function hits(graph::Graph;
344         modeparam::Tuple{String, Union{Int64, UInt32, Float64}}=("iter",
345             ↪ 10)
346         ) -> Tuple{Vector{Float64}, Vector{Float64}}
347
348     Computes Hub and Authority scores (HITS) for the graph
349
350     # Arguments
351     - `graph::Graph`: the graph
352
353     # Keywords
354     - `modeparam::Tuple{String, Union{Int64, UInt32, Float64}}=("iter", 10)`:
355         ↪ the mode and the parameters for HITS
356     - `("iter", num_iterations::Union{Int64, UInt32})`: HITS for a given
357         ↪ number of iterations

```

```

355 - `("epsi", epsilon::Union{Int64, UInt32, Float64})`: HITS until
    ↪ convergence with epsilon (both Hub and Authority)
356
357 # Returns
358 - `Tuple{Vector{Float64}, Vector{Float64}}`: the Hub and Authority scores
    ↪ for the graph
359 """
360 function hits(graph::Graph;
361     modeparam::Tuple{String, Union{Int64, UInt32, Float64}}=("iter", 10))
362     A = hits_matrix(graph)
363     H = copy(transpose(A))
364     if modeparam[1] == "iter"
365         if !(isinteger(modeparam[2])) || modeparam[2] < 0
366             error("invalid param")
367         end
368         hits_iteration(graph.num_vertices, A, H, UInt32(modeparam[2]))
369     elseif modeparam[1] == "epsi"
370         if modeparam[2] <= 0
371             error("invalid param")
372         end
373         hits_epsilon(graph.num_vertices, A, H, Float64(modeparam[2]))
374     else
375         error("invalid mode")
376     end
377 end
378
379 function hits_iteration(num_vertices::UInt32, A::Matrix{Float64},
    ↪ H::Matrix{Float64}, num_iterations::UInt32)
380     a, h = ones(Float64, num_vertices), ones(Float64, num_vertices)
381     for _ in 1:num_iterations
382         a, h = hits_update(A, H, a, h)
383     end
384     a, h
385 end
386
387 function hits_epsilon(num_vertices::UInt32, A::Matrix{Float64},
    ↪ H::Matrix{Float64}, epsilon::Float64)
388     prev_a, prev_h = ones(Float64, num_vertices), ones(Float64,
    ↪ num_vertices)
389     curr_a, curr_h = hits_update(A, H, prev_a, prev_h)
390     while norm(prev_a - curr_a) > epsilon || norm(prev_h - curr_h) >
    ↪ epsilon
391         prev_a, prev_h, (curr_a, curr_h) = curr_a, curr_h, hits_update(A, H,
    ↪ curr_a, curr_h)
392     end
393     curr_a, curr_h
394 end
395

```

```

396 function hits_update(A::Matrix{Float64}, H::Matrix{Float64},
    ↪ a::Vector{Float64}, h::Vector{Float64})
397     normalize(A * h), normalize(H * a)
398 end
399
400 function hits_matrix(graph::Graph)
401     A = zeros(Float64, (graph.num_vertices, graph.num_vertices))
402     for vertex in graph.vertices
403         for index_to in vertex.out_neighbors
404             A[index_to, vertex.index] = 1
405         end
406     end
407     A
408 end
409
410 # export generate_adjacency_matrix, generate_adjacency_list
411
412 """
413     function generate_adjacency_matrix(graph::Graph) -> Matrix{Bool}
414
415     Generates the adjacency matrix representation for the graph
416
417     # Arguments
418     - `graph::Graph`: the graph
419
420     # Returns
421     - `Matrix{Bool}`: the adjacency matrix representation for the graph
422     """
423     function generate_adjacency_matrix(graph::Graph)
424         AM = zeros(Bool, (graph.num_vertices, graph.num_vertices))
425         for vertex in graph.vertices, index_to in vertex.out_neighbors
426             AM[vertex.index_to, vertex.index] = true
427         end
428         AM
429     end
430
431 """
432     function generate_adjacency_list(graph::Graph) ->
    ↪ Vector{Vector{UInt32}}
433
434     Generates the adjacency list representation for the graph
435
436     # Arguments
437     - `graph::Graph`: the graph
438
439     # Keywords
440     - `zero_index::Bool=false`: whether the output file is zero-based
441
442     # Returns
443     - `Vector{Vector{Bool}}`: the adjacency list representation for the graph

```

```
444 """
445 function generate_adjacency_list(graph::Graph; zero_index::Bool=false)
446     AL = Array{Vector{UInt32}}(undef, graph.num_vertices)
447     for vertex in graph.vertices
448         AL[vertex.index] = Array{UInt32}(undef, 0)
449         for index_to in vertex.out_neighbors
450             if zero_index
451                 push!(AL[vertex.index], index_to - 1)
452             else
453                 push!(AL[vertex.index], index_to)
454             end
455         end
456     end
457     AL
458 end
459
460 end
```