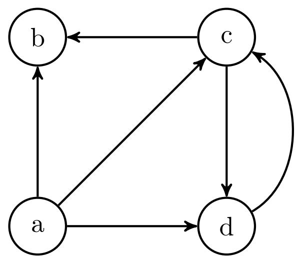
1 The PageRank Algorithm

Lab Objective: *Many real-world systems—the internet, transportation grids, social media, and so on—can be represented as graphs (networks). The PageRank algorithm is one way of ranking the nodes in a graph by importance. Though it is a relatively simple algorithm, the idea gave birth to the Google search engine in 1998 and has shaped much of the information age since then. In this lab we implement the PageRank algorithm with a few different approaches, then use it to rank the nodes of a few different networks.*

The PageRank Model

The internet is a collection of webpages, each of which may have a hyperlink to any other page. One possible model for a set of n webpages is a directed graph, where each node represents a page and node j points to node i if page j links to page i. The corresponding *adjacency matrix* A satisfies $A_{ij} = 1$ if node j links to node i and $A_{ij} = 0$ otherwise.

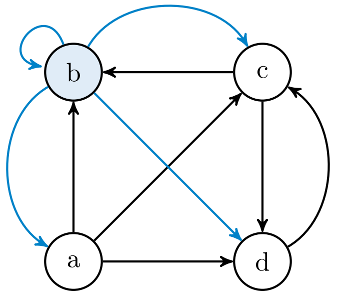


$$A = \begin{array}{c} \\ \text{a} \\ \text{b} \\ \text{c} \\ \text{d} \end{array} \begin{array}{c} \begin{array}{cccc} \text{a} & \text{b} & \text{c} & \text{d} \end{array} \\ \left[\begin{array}{cccc} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{array}\right] \end{array}$$

Figure 1.1: A directed unweighted graph with four nodes, together with its adjacency matrix. Note that the column for node b is all zeros, indicating that b is a *sink*—a node that doesn't point to any other node.

If n users start on random pages in the network and click on a link every 5 minutes, which page in the network will have the most views after an hour? Which will have the fewest? The goal of the PageRank algorithm is to solve this problem in general, therefore determining how "important" each webpage is.

Before diving into the mathematics, there is a potential problem with the model. What happens if a webpage doesn't have any outgoing links, like node b in Figure 1.1? Eventually, all of the users will end up on page b and be stuck there forever. To obtain a more realistic model, modify each sink in the graph by adding edges from the sink to every node in the graph. This means users on a page with no links can start over by selecting a random webpage.



$$\widetilde{A} = \begin{array}{c} \\ \text{a} \\ \text{b} \\ \text{c} \\ \text{d} \end{array} \begin{array}{c} \begin{array}{cccc} \text{a} & \text{b} & \text{c} & \text{d} \end{array} \\ \left[\begin{array}{cccc} 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{array}\right] \end{array}$$

Figure 1.2: Here the graph in Figure 1.1 has been modified to guarantee that node b is no longer a sink (the added links are blue). We denote the modified adjacency matrix by $\widetilde{A}$.

Now let $p_k(t)$ be the likelihood that a particular internet user is surfing webpage k at time t. Suppose at time $t+1$, the user clicks on a link to page i. Then $p_i(t+1)$ can be computed by counting the number of links pointing to page i, weighted by the total number of outgoing links for each node.

As an example, consider the graph in Figure 1.2. To get to page a at time $t+1$, the user had to be on page b at time t. Since there are four outgoing links from page b, assuming links are chosen with equal likelihood,

$$p_{\text{a}}(t+1) = \frac{1}{4}p_{\text{b}}(t).$$

Similarly, to get to page b at time $t+1$, the user had to have been on page a, b, or c at time t. Since a has 3 outgoing edges, b has 4 outgoing edges, and c has 2 outgoing edges,

$$p_{\text{b}}(t+1) = \frac{1}{3}p_{\text{a}}(t) + \frac{1}{4}p_{\text{b}}(t) + \frac{1}{2}p_{\text{c}}(t).$$

The previous equations can be written in a way that hints at a more general linear form:

$$\begin{aligned} p_{\text{a}}(t+1) &= 0p_{\text{a}}(t) + \frac{1}{4}p_{\text{b}}(t) + 0p_{\text{c}}(t) + 0p_{\text{d}}(t), \\ p_{\text{b}}(t+1) &= \frac{1}{3}p_{\text{a}}(t) + \frac{1}{4}p_{\text{b}}(t) + \frac{1}{2}p_{\text{c}}(t) + 0p_{\text{d}}(t). \end{aligned}$$

The coefficients of the terms on the right hand side are precisely the entries of the ith row of the modified adjacency matrix $\widetilde{A}$, divided by the jth column sum. In general, $p_i(t+1)$ satisfies

$$p_i(t+1) = \sum_{j=1}^{n} \widetilde{A}_{ij} \frac{p_j(t)}{\sum_{k=1}^{n} \widetilde{A}_{kj}}. \tag{1.1}$$

Note that the column sum $\sum_{k=1}^{n} \widetilde{A}_{kj}$ in the denominator can never be zero since, after the fix in Figure 1.2, none of the nodes in the graph are sinks.

Accounting for Boredom

The model in (1.1) assumes that the user can only click on links from their current page. It is more realistic to assume that the user sometimes gets bored and randomly picks a new starting page. Let

$0 \le \varepsilon \le 1$, called the *damping factor*, be the probability that a user stays interested at step t. Then the probability that the user gets bored at any time (and then chooses a new random page) is $1-\varepsilon$, and (1.1) becomes

$$p_i(t+1) = \underbrace{\varepsilon \sum_{j=1}^{n} \left(\widetilde{A}_{ij} \frac{p_j(t)}{\sum_{k=1}^{n} \widetilde{A}_{kj}} \right)}_{\substack{\text{User stayed interested and} \\ \text{clicked a link on the current page}}} + \underbrace{\frac{1-\varepsilon}{n}}_{\substack{\text{User got bored and} \\ \text{chose a random page}}}. \tag{1.2}$$

Note that (1.2) can be rewritten as the matrix equation

$$\mathbf{p}(t+1) = \varepsilon \widehat{A}\mathbf{p}(t) + \frac{1-\varepsilon}{n}\mathbf{1}, \tag{1.3}$$

where $\mathbf{p}(t) = [p_1(t), p_2(t), \ldots, p_n(t)]^\mathsf{T}$, $\mathbf{1}$ is a vector of n ones, and $\widehat{A}$ is the $n \times n$ matrix with entries

$$\widehat{A}_{ij} = \frac{\widetilde{A}_{ij}}{\sum_{k=1} \widetilde{A}_{kj}}. \tag{1.4}$$

In other words, $\widehat{A}$ is $\widetilde{A}$ normalized so that the columns each sum to 1. For the graph in Figure 1.2, the matrix $\widehat{A}$ is given by

$$\widehat{A} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \left[\begin{array}{cccc} 0 & 1/4 & 0 & 0 \\ 1/3 & 1/4 & 1/2 & 0 \\ 1/3 & 1/4 & 0 & 1 \\ 1/3 & 1/4 & 1/2 & 0 \end{array} \right] \end{array}. \tag{1.5}$$

Problem 1. Write a class for representing directed graphs via their adjacency matrices. The constructor should accept an $n \times n$ adjacency matrix A and a list of node labels (such as [a, b, c, d]) defaulting to `None`. Modify A as in Figure 1.2 so that there are no sinks in the corresponding graph, then calculate the $\widehat{A}$ from (1.4). Save $\widehat{A}$ and the list of labels as attributes. Use $[0, 1, \ldots, n-1]$ as the labels if none are provided. Finally, raise a `ValueError` if the number of labels is not equal to the number of nodes in the graph.
(Hint: use array broadcasting to compute $\widehat{A}$ efficiently.)

For the graph in Figure 1.1, check that your $\widehat{A}$ matches (1.5).

Computing the Rankings

In the model (1.2), define the *rank* of node i as the limit

$$p_i = \lim_{t\to\infty} p_i(t).$$

There are several ways to solve for $\mathbf{p} = \lim_{t\to\infty} \mathbf{p}(t)$.

Linear System

If $\mathbf{p}$ exists, then taking the limit as $t \to \infty$ to both sides of (1.3) gives the following.

$$\begin{aligned} \lim_{t\to\infty} \mathbf{p}(t+1) &= \lim_{t\to\infty} \left[\varepsilon \widehat{A}\mathbf{p}(t) + \frac{1-\varepsilon}{n}\mathbf{1}\right] \\ \mathbf{p} &= \varepsilon\widehat{A}\mathbf{p} + \frac{1-\varepsilon}{n}\mathbf{1} \\ \left(I - \varepsilon\widehat{A}\right)\mathbf{p} &= \frac{1-\varepsilon}{n}\mathbf{1} \end{aligned} \tag{1.6}$$

This linear system is easy to solve as long as the number of nodes in the graph isn't too large.

Eigenvalue Problem

Let E be an $n \times n$ matrix of ones. Then $E\mathbf{p}(t) = \mathbf{1}$ since $\sum_{i=1} p_i(t) = 1$. Substituting into (1.3),

$$\mathbf{p}(t+1) = \varepsilon\widehat{A}\mathbf{p}(t) + \frac{1-\varepsilon}{n}E\mathbf{p}(t) = \left(\varepsilon\widehat{A} + \frac{1-\varepsilon}{n}E\right)\mathbf{p}(t) = B\mathbf{p}(t), \tag{1.7}$$

where $B = \varepsilon\widehat{A} + \frac{1-\varepsilon}{n}E$. Now taking the limit at $t \to \infty$ of both sides of (1.7),

$$B\mathbf{p} = \mathbf{p}.$$

That is, $\mathbf{p}$ is an eigenvector of B corresponding to the eigenvalue $\lambda = 1$. In fact, since the columns of B sum to 1, and because the entries of B are strictly positive (because the entries of E are all positive), Perron's theorem guarantees that $\lambda = 1$ is the unique eigenvalue of B of largest magnitude, and that the corresponding eigenvector $\mathbf{p}$ is unique up to scaling. Furthermore, $\mathbf{p}$ can be scaled so that each of its entires are positive, meaning $\mathbf{p}/\|\mathbf{p}\|_1$ is the desired PageRank vector.

Note

A *Markov chain* is a weighted directed graph where each node represents a *state* of a discrete system. The weight of the edge from node j to node i is the probability of transitioning from state j to state i, and the adjacency matrix of a Markov chain is called a *transition matrix*.

Since B from (1.7) contains nonnegative entries and its columns all sum to 1, it can be viewed as the transition matrix of a Markov chain. In that context, the limit vector $\mathbf{p}$ is called the *steady state* of the Markov chain.

Iterative Method

Solving (1.6) or (1.7) is feasible for small networks, but they are not efficient strategies for very large systems. The remaining option is to use an iterative technique. Starting with an initial guess $\mathbf{p}(0)$, use (1.3) to compute $\mathbf{p}(1), \mathbf{p}(2), \ldots$ until $\|\mathbf{p}(t) - \mathbf{p}(t-1)\|$ is sufficiently small. From (1.7), we can see that this is just the power method[1] for finding the eigenvector corresponding to the dominant eigenvalue of B.

[1]See the Least Squares and Computing Eigenvalues lab for details on the power method.

Problem 2. Add the following methods to your class from Problem 1. Each should accept a damping factor ε (defaulting to 0.85), compute the PageRank vector $\mathbf{p}$, and return a dictionary mapping label i to its PageRank value p_i.

1. `linsolve()`: solve for $\mathbf{p}$ in (1.6).
2. `eigensolve()`: solve for $\mathbf{p}$ using (1.7). Normalize the resulting eigenvector so its entries sum to 1.
3. `itersolve()`: in addition to ε, accept an integer `maxiter` and a float `tol`. Iterate on (1.3) until $\|\mathbf{p}(t) - \mathbf{p}(t-1)\|_1 <$ `tol` or $t >$ `maxiter`. Use $\mathbf{p}(0) = [\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n}]^\mathsf{T}$ as the initial vector (any positive vector that sums to 1 will do, but this assumes equal starting probabilities).

Check that each method yields the same results. For the graph in Figure 1.1 with $\varepsilon = 0.85$, you should get the following dictionary mapping labels to PageRank values.

```
{'a': 0.095758635, 'b': 0.274158285, 'c': 0.355924792, 'd': 0.274158285}
```

Problem 3. Write a function that accepts a dictionary mapping labels to PageRank values, like the outputs in Problem 2. Return a list of labels sorted **from highest to lowest** rank. (Hint: if `d` is a dictionary, use `list(d.keys())` and `list(d.values())` to get the list of keys and values in the dictionary, respectively.)

For the graph in Figure 1.1 with $\varepsilon = 0.85$, this is the list [c, b, d, a] (or [c, d, b, a], since b and d have the same PageRank value).

Problem 4. The file `web_stanford.txt` contains information on Stanford University webpages[a] and the hyperlinks between them, gathered in 2002.[b] Each line of the file is formatted as `a/b/c/d/e/f...`, meaning the webpage with ID `a` has hyperlinks to webpages with IDs `b`, `c`, `d`, and so on.

Write a function that accepts a damping factor ε defaulting to 0.85. Read the data and get a list of the n unique page IDs in the file (the labels). Construct the $n \times n$ adjacency matrix of the graph where node j points to node i if webpage j has a hyperlink to webpage i. Use your class from Problem 1 and its `itersolve()` method from Problem 2 to compute the PageRank values of the webpages, then rank them with your function from Problem 3. In the case where two webpages have the same rank, resolve ties by listing the webpage with the larger ID number first. (Hint: Sorting the list of unique webpage IDs before ranking will order the site IDs from smallest to largest.) Return the ranked list of webpage IDs.
(Hint: after constructing the list of webpage IDs, make a dictionary that maps a webpage ID to its index in the list. For Figure 1.1, this would be `{'a': 0, 'b': 1, 'c': 2, 'd': 3}`. The values are the row/column indices in the adjacency matrix for each label.)

With $\varepsilon = 0.85$, the top three ranked webpage IDs are 98595, 32791, and 28392.

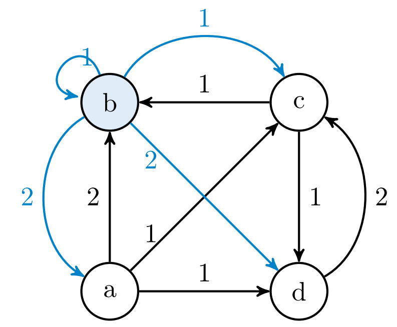
[a] `http://www.stanford.edu/`

[b] See `http://snap.stanford.edu/data/web-Stanford.html` for the original (larger) dataset.

PageRank on Weighted Graphs

Nothing in the formulation of the PageRank model (1.3) requires that the edges of the graph are unweighted. If A_{ij} is the weight of the edge from node j to node i (weight 0 meaning there is no edge from j to i), then the columns of $\widehat{A}$ still sum to 1. Thus $B = \varepsilon\widehat{A} + \frac{1-\varepsilon}{n}E$ is still positive definite, so we can expect a unique PageRank vector $\mathbf{p}$ to exist.

Adding weights to the edges can improve the fidelity of the model and produce a slightly more realistic PageRank ordering. On a given webpage, for example, if hyperlinks to page a are clicked on more frequently hyperlinks to page b, the edge from node a should be given more weight than the edge to node b.



$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \left[\begin{array}{cccc} 0 & 0 & 0 & 0 \\ 2 & 0 & 1 & 0 \\ 1 & 0 & 0 & 2 \\ 1 & 0 & 2 & 0 \end{array}\right] \end{array}$$

$$\widehat{A} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \left[\begin{array}{cccc} 0 & 1/4 & 0 & 0 \\ 1/2 & 1/4 & 1/3 & 0 \\ 1/4 & 1/4 & 0 & 1 \\ 1/4 & 1/4 & 2/3 & 0 \end{array}\right] \end{array}$$

Figure 1.3: A directed weighted graph with four nodes, together with its adjacency matrix and the corresponding PageRank transition matrix. Edges that are added to fix sinks have weight 1, so the computation of $\widetilde{A}$ and $\widehat{A}$ are exactly the same as in Figure 1.2 and (1.4), respectively.

Problem 5. The files `ncaa2010.csv`, `ncaa2011.csv`, ..., `ncaa2017.csv` each contain data for men's college basketball for a given school year.[a] Each line (except the very first line, which is a header) represents a different basketball game, formatted `winning_team,losing_team`.

Write a function that accepts a filename and a damping factor ε defaulting to 0.85. Read the specified file (skipping the first line) and get a list of the n unique teams in the file. Construct the $n \times n$ adjacency matrix of the graph where node j points to node i with weight w if team j was defeated by team i in w games. That is, **edges point from losers to winners**. For instance, the graph in Figure 1.3 would indicate that team c lost to team b once and to team d twice, team b was undefeated, and team a never won a game. Use your class from Problem 1 and its `itersolve()` method from Problem 2 to compute the PageRank values of the teams, then rank them with your function from Problem 3. Return the ranked list of team names.

Using `ncaa2010.csv` with $\varepsilon = 0.85$, the top three ranked teams (of the 607 total teams) should be UConn, Kentucky, and Louisville, in that order. That season, UConn won the championship, Kentucky was a semifinalist, and Louisville lost in the first tournament round (a surprising upset).

[a] `ncaa2010.csv` has data for the 2010–2011 season, `ncaa2011.csv` for the 2011–2012 season, and so on.

Note

In Problem 5, the damping factor ε acts as an "upset" factor: a larger ε puts more emphasis on win history; a smaller ε allows more randomness in the system, giving underdog teams a higher probability of defeating a team with a better record.

It is also worth noting that the sink-fixing procedure is still reasonable for this model because it gives every other team **equal** likelihood of beating an undefeated team. That is, the additional edges don't provide an extra advantage to any one team.

PageRank with NetworkX

NetworkX, usually imported as `nx`, is a third-party package for working with networks. It represents graphs internally with dictionaries, thus taking full advantage of the sparsity in a graph. The base class for directed graphs is called `nx.DiGraph`. Nodes and edges are usually added or removed incrementally with the following methods.

Method	Description
`add_node()`	Add a single node.
`add_nodes_from()`	Add a list of nodes.
`add_edge()`	Add an edge between two nodes, adding the nodes if needed.
`add_edges_from()`	Add multiple edges (and corresponding nodes as needed).
`remove_edge()`	Remove a single edge (no nodes are removed).
`remove_edges_from()`	Remove multiple edges (no nodes are removed).
`remove_node()`	Remove a single node and all adjacent edges.
`remove_nodes_from()`	Remove multiple nodes and all adjacent edges.

Table 1.1: Methods of the `nx.DiGraph` class for inserting or removing nodes and edges.

For example, the weighted graph in Figure 1.3 can be constructed with the following code.

```
>>> import networkx as nx

# Initialize an empty directed graph.
>>> DG = nx.DiGraph()

# Add the directed edges (nodes are added automatically).
>>> DG.add_edge('a', 'b', weight=2)     # a --> b (adds nodes a and b)
>>> DG.add_edge('a', 'c', weight=1)     # a --> c (adds node c)
>>> DG.add_edge('a', 'd', weight=1)     # a --> d (adds node d)
>>> DG.add_edge('c', 'b', weight=1)     # c --> b
>>> DG.add_edge('c', 'd', weight=2)     # c --> d
>>> DG.add_edge('d', 'c', weight=2)     # d --> c
```

Once constructed, an `nx.Digrah` object can be queried for information about the nodes and edges. It also supports dictionary-like indexing to access node and edge attributes, such as the weight of an edge.

Method	Description
`has_node(A)`	Return `True` if `A` is a node in the graph.
`has_edge(A,B)`	Return `True` if there is an edge from `A` to `B`.
`edges()`	Iterate through the edges.
`nodes()`	Iterate through the nodes.
`number_of_nodes()`	Return the number of nodes.
`number_of_edges()`	Return the number of edges.

Table 1.2: Methods of the `nx.DiGraph` class for accessing nodes and edges.

```
# Check the nodes and edges.
>>> DG.has_node('a')
True
>>> DG.has_edge('b', 'a')
False
>>> list(DG.nodes())
['a', 'b', 'c', 'd']
>>> list(DG.edges())
[('a', 'b'), ('a', 'c'), ('a', 'd'), ('c', 'b'), ('c', 'd'), ('d', 'c')]

# Change the weight of the edge (a, b) to 3.
>>> DG['a']['b']["weight"] += 1
>>> DG['a']['b']["weight"]
3
```

NetworkX efficiently implements several graph algorithms. The function `nx.pagerank()` computes the PageRank values of each node iteratively with sparse matrix operations. This function returns a dictionary mapping nodes to PageRank values, like the methods in Problem 2.

```
# Calculate the PageRank values of the graph.
>>> nx.pagerank(DG, alpha=0.85)     # alpha is the damping factor (epsilon).
{'a': 0.08767781186947843,
 'b': 0.23613138394239835,
 'c': 0.3661321209576019,
 'd': 0.31005868323052127}
```

Achtung!

NetworkX also has a class, `nx.Graph`, for *undirected graphs*. The edges in an undirected graph are bidirectional, so the corresponding adjacency matrix is symmetric.

The PageRank algorithm is not very useful for undirected graphs. In fact, the PageRank value for node is close to its *degree*—the number of edges it connects to—divided by the total number of edges. In Problem 5, that would mean the team who simply played the most games would be ranked the highest. Always use `nx.DiGraph`, not `nx.Graph`, for PageRank and other algorithms that rely on directed edges.

Problem 6. The file `top250movies.txt` contains data from the 250 top-rated movies according to IMDb.[a] Each line in the file lists a movie title and its cast as `title/actor1/actor2/...`, with the actors listed mostly in billing order (stars first), though some casts are listed alphabetically or in order of appearance.

Create a `nx.DiGraph` object with a node for each actor in the file. The weight from actor a to actor b should be the number of times that actor a and b were in a movie together but actor b was listed first. That is, **edges point to higher-billed actors** (see Figure 1.4). Compute the PageRank values of the actors and use your function from Problem 3 to rank them. Return the list of ranked actors.
(Hint: Consider using `itertools.combinations()` while constructing the graph. Also, use `encoding="utf-8"` as an argument to `open()` to read the file, since several actors and actresses have nonstandard characters in their names such as ø and æ.)

With $\varepsilon = 0.7$, the top three actors should be Leonardo DiCaprio, Robert De Niro, and Tom Hanks, in that order.

[a]`https://www.imdb.com/search/title?groups=top_250&sort=user_rating`

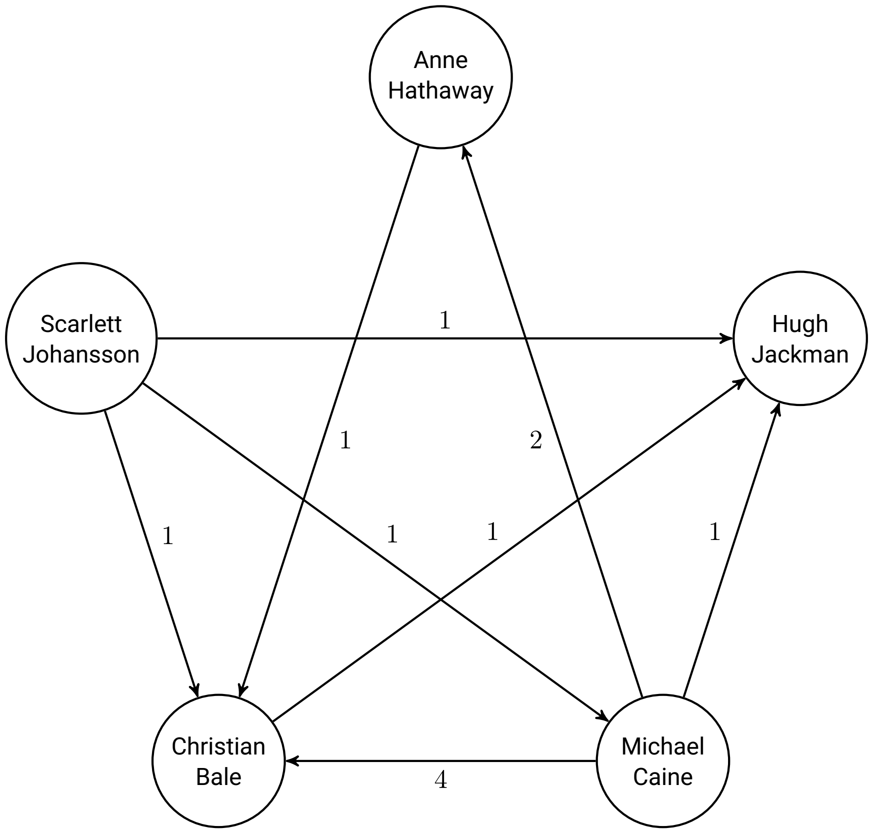


Figure 1.4: A portion of the graph from Problem 6. Michael Caine was in four movies with Christian Bale where Christian Bale was listed first in the cast.

Additional Material

Sparsity

On very large networks, the PageRank algorithm becomes computationally difficult because of the size of the adjacency matrix A. Fortunately, most adjacency matrices are highly sparse, meaning the number of edges is much lower than the number of entries in the matrix. Consider adding functionality to your class from Problem 1 so that it stores $\widehat{A}$ as a sparse matrix and performs sparse linear algebra operations in the methods from Problem 2 (use `scipy.sparse.linalg`).

PageRank as a Predictive Model

The data files in Problem 5 include tournament games for their respective seasons, so the resulting rankings naturally align with the outcome of the championship. However, it is also useful to use PageRank as a predictive model: given data for all regular season games, can the outcomes of the tournament games be predicted? Over 40 million Americans fill out 60–100 million March Madness brackets each year and bet over $9 billion on the tournament, so being able to predict the outcomes of the games is a big deal. See `http://games.espn.com/tournament-challenge-bracket` for more details.

Given regular season data, PageRank can be used to predict tournament results as in Problem 5. There are some pitfalls though; for example, how should ε be chosen? Using $\varepsilon = .5$ with `ncaa2010.csv` minus tournament data (all but the last 63 games in the file) puts UConn—the actual winner that year—in seventh place, while $\varepsilon = .9$ puts UConn in fourth. Both values for ε also rank BYU as number one, but BYU lost in the Sweet Sixteen that year. In practice, Google uses .85 as the damping factor, but there is no rigorous reasoning behind that particular choice.

Other Centrality Measures

In network theory, the *centrality* of a node refers to its importance. Since there are lots of ways to measure importance, there are several different centrality measures.

- *Degree centrality* uses the *degree* of a node, meaning the number of edges adjacent to it (independent of edge direction), for ranking. An academic paper that has been cited many times has a high degree and is considered more important than a paper that has only been cited once.

- *Eigenvector centrality* is an extension of degree centrality. Instead of each neighbor contributing equally to the centrality, nodes that are important are given a higher weight. Thus a node connected to lots of unimportant nodes can have the same measure as a node connected to a few, important nodes. Eigenvector centrality is measured by the eigenvector associated with the largest eigenvalue of the adjacency matrix of the network.

- *Katz centrality* is a modification to eigenvalue centrality for directed networks. Outgoing nodes contribute centrality to their neighbors, so an important node makes its neighbors more important.

- PageRank adapts Katz centrality by averaging out the centrality that a node can pass to its neighbors. For example, if Google—a website that should have high centrality—points to a million websites, then it shouldn't pass on that high centrality to all of million of its neighbors, so each neighbor gets one millionth of Google's centrality.

For more information on these centralities, as well as other ways to measure node importance, see [?].