Mapping Influential Articles in Local Wikipedia Networks via Personalized PageRank and Hyperlink-Induced Topic Search

Soham Jain and Saye Karthikeyan
School of Computer Science, Carnegie Mellon University
Matrices and Linear Transformations (21-241)
Pavel Kovalev
December 5, 2025

1 Mathematical Background

1.1 Markov Chains

A probability vector is a vector

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^n$$

whose entries are all nonnegative and sum to one. In other words,

$$x_1, x_2, \ldots, x_n \geq 0 \quad \text{and} \quad x_1 + x_2 + \cdots + x_n = 1.$$

Some example of probability vectors are

$$\begin{bmatrix} 0.3 \\ 0.2 \\ 0.5 \end{bmatrix}, \begin{bmatrix} 0.25 \\ 0.75 \\ 0 \end{bmatrix}, \text{ and } \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}.$$

A Markov matrix is a matrix $M \in M_{n\times n}(\mathbb{R})$ whose columns are all probability vectors. Thus,

$$M = [\mathbf{v}_1 \ \ldots \ \mathbf{v}_n] \text{ where } \mathbf{v}_1, \ldots \mathbf{v}_n \in \mathbb{R}^n \text{ are probability vectors.}$$

Some examples of Markov matrices are

$$\begin{bmatrix} 0.5 & 0.3 & 0.2 \\ 0.3 & 0.4 & 0.3 \\ 0.2 & 0.3 & 0.5 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & 0.2 \\ 0 & 0.8 \end{bmatrix}.$$

Let $M \in M_{n\times n}(\mathbb{R})$ be a Markov matrix and let $\mathbf{x}_0 \in \mathbb{R}^n$ be an initial probability vector. Then, the following sequence of probability vectors is called a Markov chain:

$$\mathbf{x}_0,\ \mathbf{x}_1,\ \mathbf{x}_2, \ldots \text{ where each } \mathbf{x}_k = M^k\mathbf{x}_0$$

Consider the Markov matrix $M = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix}$. We will attempt to show that 1 is an eigenvalue of M via showing $\det(M - I_n) = 0$. We can now observe:

$$= \det \begin{bmatrix} a_{1,1} - 1 & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} - 1 \end{bmatrix} \qquad \text{(Subtraction by } I_n\text{)}$$

$$= \det \begin{bmatrix} a_{1,1} + \cdots + a_{m,1} - 1 & \cdots & a_{1,n} + \cdots + a_{m,n} - 1 \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} - 1 \end{bmatrix}$$

(From addition of columns to the first row)

$$= \det \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} - 1 \end{bmatrix} \qquad \text{(From definition of stochastic matrix: } M^T 1 = 1\text{)}$$

$$= 0 \qquad \text{(From definition of determinant of a matrix with a zero row)}$$

Therefore, 1 is an eigenvalue of any Markov matrix M. This eigenvector associated to $\lambda = 1$ will be of interest when considering the topic of convergence to a unique steady state.

1.2 Random Walks

In the context of Markov chains, a random walk is a process where each state corresponds to a node in a graph, and the probabilities of moving from one state to another are given by a Markov matrix $M \in M_{n\times n}(\mathbb{R})$. In particular, each M_{ij} represents the probability of moving from state j to state i. Starting at an initial probability vector $\mathbf{x}_0$, the probability distribution after k steps is $\mathbf{x}_k = M^k\mathbf{x}_0$.

Random walks can be used to model the way that a user navigates through links in a website. At each step, the user can choose the next page to visit according to the probabilities in M. If a page j has d_j outgoing links, one simple example of a Markov matrix representation is

$$M_{ij} = \begin{cases} \dfrac{1}{d_j} & \text{if page } j \text{ links to page } i \\ 0 & \text{if page } j \text{ does not link to page } i \end{cases}$$

The goal of a random walk is often to use an algorithm like PageRank to find a steady state vector $\mathbf{w}$ where the probabilities converge (i.e. $\mathbf{x}_k \to \mathbf{w}$ as $k \to \infty$). The steady-state distribution can be used to indicate which pages are the most "important" or frequently visited in the long run [6].

1.3 PageRank

PageRank refines on this idea of a random walk by capturing real user behavior in two parts.

A specific PageRank score of a page i can be computed by the sum of all the pages w that link to i, divided by the number of outbound links on our selected page w. We can represent this through $P(v) = \sum_{i\in Q} \frac{P(i)}{L(i)}$, where Q is the set of all pages linked to v, and $L(i)$ is the amount of links on each page i. This gives way to PageRank being defined recursively: if page v links to an important page w, then v must be of relative importance as well.

One issue that we have yet to address is the problem surrounding dangling nodes. Dangling nodes are those that do not have pages linking to them, which would make our algorithm come to a halt. We address this through the damping factor d, which represents the probability that a user will stop following links and jump to a random page in our network. For the purposes of this paper, we will use $d = 0.85$ for our damping factor. We can account for dangling nodes with adding a $\frac{(1-d)}{N}$ term to our PageRank score, which represents the probability that our user chooses any of the n webpages at random after a dangling node.

Let $M \in M_{n\times n}(\mathbb{R})$ be a Markov matrix that represents a network of n pages, where $M_{ij} > 0$ if page j links page i and $M_{ij} = 0$ otherwise. M is called an importance matrix. Let $M' \in M_{n\times n}(\mathbb{R})$ be the same as the matrix M, but with any zero columns replaced by $\frac{1}{n}\mathbf{1}$. Still, the matrix M' may have zero entries because it may contain columns where some entries are zero and some are nonzero. Thus, we cannot necessarily say that M' is a positive Markov matrix (a Markov matrix with all positive entries).

In order to ensure a positive Markov matrix representation, PageRank uses this damping factor for dangling nodes. Let $B = \frac{1}{n}\mathbf{1}\mathbf{1}^T \in M_{n\times n}(\mathbb{R})$. Then, we can let the Google matrix $G = dM' + (1-d)B \in M_{n\times n}(\mathbb{R})$. Because the values in M' are originally nonnegative and we are adding a positive $(1-p)\times\frac{1}{n}$ to each entry in the matrix, G is a positive Markov matrix. By the Perron-Frobenius Theorem, there exists a positive steady-state vector $\mathbf{w}$ that satisfies $\mathbf{x}_k \to \mathbf{w}$ with $\lambda = 1$ as $k \to \infty$. We can use $\mathbf{w}$ to identify the most important pages in the network (namely, the greatest values in the normalized steady-state vector). A PageRank score P_i represents the long-run probability that a random-walker will land on page i in their walk.

1.4 Power Iteration

We will attempt to compute our PageRank vector P_r through power iteration on our stochastic matrix $M := (K^{-1}A)^T$, where $A \in M_{n\times n}(\mathbb{R})$ is our constructed adjacency matrix from graph G, where $A_{ij} = \begin{cases} 1 \text{ if page i links to j} \\ 0 \text{ otherwise} \end{cases}$.

First, let $K = diag(k_1, \ldots, k_n)$, where k_i is the number of edges in G s.t. $(i, w) \in E, \exists w \in V$, where G is our subgraph, consisting of vertices in V and edges in E. We can then show that $K = diag(\frac{1}{k_1}, \ldots, \frac{1}{k_n})$. We can now show the following:

$$
\begin{aligned}
K^{-1}A & \\
&= \begin{bmatrix} \frac{1}{k_1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{k_n} \end{bmatrix} \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix} && \text{(Expansion)} \\
&= \begin{bmatrix} \frac{a_{1,1}}{k_1} & \cdots & \frac{a_{n,1}}{k_1} \\ \vdots & \ddots & \vdots \\ \frac{a_{m,1}}{k_n} & \cdots & \frac{a_{m,1}}{k_n} \end{bmatrix}
\end{aligned}
$$

We know that for some row i, $\sum_{k=1}^{n} a_{ik} = k_i$ by the way we defined matrix K. Therefore, $(K^{-1}A)$ is row-stochastic, and $(K^{-1}A)^T$ is column stochastic, thus a Markov matrix.

We can now apply our iteration operation, where

$$
T(v+1) = dMT(v) + \frac{1-d}{n}1, 1 := \begin{bmatrix} 1 \\ \vdots \\ 1 \\ 1 \end{bmatrix}
$$

We will check on each iteration whether $|T(v+1) - T(v)| < \epsilon, \epsilon = 10^{-6}$, on which case we will consider the steady state to have been stabilized.

2 Project Background

The goal of this project is to investigate how PageRank behaves on a subgraph of Wikipedia constructed from a small set of seed articles. By treating Wikipedia as a directed graph, where articles

serve as vertices and hyperlinks act as directed edges, we aim to understand how importance acts in this subgraph. Our objective is to determine whether a seed-focused PageRank procedure can reveal meaningful structures and relations, which could serve useful for further application. We seek to also compare these results with Hyperlink-Induced Topic Search (HITS) to observe the differences that each algorithm gives in importance.

To accomplish this, we first extract a subset of Wikipedia centered around three to four seed articles, recursively gathering linked pages to form a well-defined induced subgraph. We then construct the corresponding Markov transition matrix and apply personalized PageRank, biasing the random-walk distribution toward the chosen seeds. This personalization allows us to emphasize specific topics in our subgraph, effectively influencing steady-state probabilities within the subgraph. Finally, we visualize the resulting directed graph, emphasizing according to the computed PageRank scores to provide an interpretable presentation of article importance.

3 Methodology

3.1 Extension: Personalized PageRank

While the PageRank algorithm is efficient in producing consistent results, a problem arises when we want to tailor it towards personalized preferences. Consider a subset $S \subseteq V$ of our total articles, which we consider to be the seed from where we begin our surfing. Then, some of the articles in our graph are not of much relevance to us.

We can remedy this by making a small change to our PageRank algorithm. We know that in our original algorithm, we compute the chance that our web-surfer makes a random jump to an article in our graph. However, we desire articles closely linked to S to be of more relevance. Personalized PageRank's impact comes from modifying this idea of a random jump, giving preference to articles that match our interests [2].

In the standard PageRank formula, the teleportation probability distributes uniformly across all nodes. In Personalized PageRank, we replace this uniform distribution with a personalized teleportation vector v, where $v_i = \begin{cases} \frac{1}{|S|} & \text{if page } i \in S \\ 0 & \text{otherwise} \end{cases}$, which concentrates probability mass on the seed pages [3]. We can apply power iteration on this matrix M until we find a steady state vector that our system converges to [1]. This vector p represents the long-run probabilities that a web-surfer will land on page i, which is in other words, the relative importance of a page i.

We will attempt to make a Column-stochastic Matrix M from our graph G, then we will iterate on a PageRank vector p which will be initialized to $\begin{bmatrix} \frac{1}{n} \\ \frac{1}{n} \\ \vdots \\ \frac{1}{n} \end{bmatrix}$, which will terminate once the differences between iterations are within range of our tolerance ϵ.

The Google Matrix was not explicitly used, but can be shown to be equivalent to applying power

iteration on our stochastic matrix M.

3.2 Hyperlink-Induced Topic Search

Rather than modeling a random walk, another way to measure importance in a network is by looking at how pages reinforce each other through two roles. One role is that a page can act as a hub if it points to many useful sources. Another role is that a page may act as an authority if many strong hubs point to it. These two features feed into each other through an iterative process, as strong hubs boost authority for the pages they link to, and vice versa. HITS utilizes this approach in order to identify important pages in a network.

Suppose that our directed graph with n pages has adjacency matrix $A \in M_{n\times n}(\mathbb{R})$, where

$$A_{ij} = \begin{cases} 1 & \text{if page } i \text{ links to page } j \\ 0 & \text{if otherwise} \end{cases}$$

Let $\mathbf{a} \in \mathbb{R}^n$ be the authority vector and let $\mathbf{h} \in \mathbb{R}^n$ be the hub vector. At each iteration k, we update these vectors:

$$\mathbf{a}_{k+1} = A^T\mathbf{h}_k, \;\; \mathbf{h}_{k+1} = A\mathbf{a}_{k+1}$$

.
Applying L2 normalization at each iteration ensures that these vectors remain bounded:

$$\mathbf{a}_{k+1} = \frac{\mathbf{a}_{k+1}}{||\mathbf{a}_{k+1}||_2}, \;\; \mathbf{h}_{k+1} = \frac{\mathbf{h}_{k+1}}{||\mathbf{h}_{k+1}||_2}$$

These updates repeat until both vectors stabilize. Substituting the update for $\mathbf{a}_{k+1}$ into the update for $\mathbf{h}_{k+1}$, and vice versa, we obtain:

$$\mathbf{h}_{k+1} = AA^T\mathbf{h}_k, \;\; \mathbf{a}_{k+1} = A^TA\mathbf{a}_k$$

Thus, the iterative process of HITS is equivalent to computing the eigenvectors of the matrices A^TA and AA^T, which are symmetric and positive definite [4]. Therefore, the algorithm converges when the hub and authority vectors converge to the eigenvectors for AA^T and A^TA, respectively. This mathematical interpretation makes it clear how authorities and hubs complement one another. Because authority scores are computed from the hubs that point to them, a page's importance as an authority grows when it is referenced by strong hubs. Similarly, hub scores are computed from the authorities they point to, so a page becomes a stronger hub if it links to important authorities.

After normalization, the largest entries in **a** correspond to the strongest authorities and the largest entries in **h** correspond to the strongest hubs. Thus, we receive two sets of articles from a given seed of articles with HITS, compared to just one set with PageRank [5].

3.3 Code: Setup and Graph Initialization

```
    import wikipediaapi
    import networkx as nx
    import matplotlib.pyplot as plt
    import numpy as np
    from collections import deque
    import time
    import requests
    import json
    import random
    # Included all necessary libraries. NetworkX used for graph creation. WikiAPI
    # used for scraping necessary data. Matplotlib and Numpy for
        calculation/visualization

    # Initialize Wikipedia API
    wiki_wiki = wikipediaapi.Wikipedia(
        user_agent='WikipediaGraphAnalysis/1.0 (https://example.com)',
        language='en',
        extract_format=wikipediaapi.ExtractFormat.WIKI
    )
```

First, we imported all necessary libraries. Wikipedia-API is the primary API we use to load the dataset of Wikipedia articles. We also imported NetworkX, a module for modeling relations between objects and working with graphs, to construct the graph for visualizing the network of Wikipedia articles. In addition, we imported libraries like NumPy and Matplotlib for mathematical calculations and visualizing our results.

```
    def search_wikipedia_first_result(search_term):
        '''
          Searched for the very first result for the search term.
          Doing a search instead of direct lookup since syntactically
          whatever we enter as a seed might not be the exact page name,
          so this helps us circumvent that.
        '''
        try:
            url = "https://en.wikipedia.org/w/api.php"
            params = {
                "action": "query",
                "list": "search",
                "srsearch": search_term,
                "srlimit": 1,
                "format": "json",
                "srnamespace": 0
            }

            headers = {
                'User-Agent': 'WikipediaGraphAnalysis/1.0 (https://example.com;
                    contact@example.com)'
            }

            response = requests.get(url, params=params, timeout=15, headers=headers)

            # Check response status
            if response.status_code != 200:
                print(f"HTTP {response.status_code} for '{search_term}'")
                return None

```

```
            # Parsing JSON
            try:
                data = response.json()
            except ValueError as e:
                print(f"Response preview")
                return None

            if "query" in data and "search" in data["query"] and
                len(data["query"]["search"]) > 0:
                title = data["query"]["search"][0]["title"]
                page = wiki_wiki.page(title)
                if page.exists():
                    return title
            return None
        except Exception as e:
            print(f"Error searching")
            return None

        return None

    def get_wiki_links(page_title, max_links=50):
        '''
            Get internal Wiki links from a page.
            Based on parameters that we tuned, especially
            the maximum number of links, which is discussed
            more thoroughly in the report. Filtering for actual
            links and not files or categories.

            Randomized since we initially found that
            our links were returned in alphabetical order
            in the event of a tie between pagerank scores.
        '''
        try:
            page = wiki_wiki.page(page_title)

            # Collect all valid links
            all_links = []
            for link_title in page.links.keys():
                if link_title and not link_title.startswith('Category:') and not
                    link_title.startswith('File:') and not
                    link_title.startswith('Template:'):
                    all_links.append(link_title)

            # Randomize to avoid uniformity, then returning up until the maximum
                amount of links
            random.shuffle(all_links)
            return all_links[:max_links]
        except Exception as e:
            print(f"Error getting links")
            return []
```

These were our methods for data collection. Instead of a direct match with a Wikipedia article, we used the search feature of the Wikipedia API to gather results. Then, we scraped the most similar result from the platform. This process was done in order to circumvent the need to match our seeds to the exact naming convention that the articles use. Furthermore, we filtered out pages starting with “Category:”, “File:”, or “Template:”, since these are metadata or auxiliary pages. We also used Wikipedia API’s “links” feature to extract all the outgoing Wiki links from the article.

```python
    def build_wikipedia_graph(seed_terms, max_depth=2, max_links_per_page=20,
        max_total_pages=200):
        '''
            Build a directed graph from Wikipedia pages starting with seed terms. (u,
                w) and (w, u) valid edges in this graph.
            - seed_terms: List of search terms
            - max_depth: How many levels of links to follow
            - max_links_per_page: Maximum links to extract per page
            - max_total_pages: Maximum total pages to include in graph
        '''
        G = nx.DiGraph()
        # BFS Mark set to check membership easily
        visited = set()
        seed_pages = []

        for term in seed_terms:
            page_title = search_wikipedia_first_result(term)
            if page_title:
                print(f"Found: '{term}' -> '{page_title}'")
                seed_pages.append(page_title)
                visited.add(page_title)
                G.add_node(page_title, is_seed=True)
            else:
                print(f"No result found for '{term}'")

        # Running BFS
        queue = deque([(page, 0) for page in seed_pages])  # (page_title, depth)

        while queue and len(visited) < max_total_pages:
            current_page, depth = queue.popleft()

            if depth >= max_depth:
                continue

            # Stop if we have reached the maximum number of pages
            if len(visited) >= max_total_pages:
                break

            # Get links from current page - use max_links_per_page parameter
            links = get_wiki_links(current_page, max_links=max_links_per_page)

            for link in links:
                # Stop adding nodes if we've reached the limit
                if len(visited) >= max_total_pages:
                    break

                if link != current_page:
                    if link not in visited and len(visited) < max_total_pages:
                        # Can add new node if the link is not already in visited
                        visited.add(link)
                        G.add_node(link, is_seed=False)
                        G.add_edge(current_page, link)

                        # Add to queue if we haven't reached max depth
                        if depth + 1 < max_depth:
                            queue.append((link, depth + 1))

            time.sleep(0.1) # To avoid timeouts from excessive API calls
```

```
        return G
```

We used the NetworkX module to construct our directed graphs of Wikipedia articles. Our graph is represented as $G = (V, E)$, where V is the set of vertices (Wikipedia articles) and E is the set of edges (v, w), which represents article v linking to article w. Note that $(v, w) \neq (w, v)$, since one Wikipedia article does not necessarily have to link to its source. We run Breadth First Search, starting with our seed pages, to populate our graph for PageRank and HITS.

3.4 Code and Algorithm Design: Personalized PageRank

```
    def personalized_pagerank(G, seed_pages, damping=0.85, max_iter=100, tol=1e-6):
        '''
        Runs Personalized PageRank

        Using power iteration to calculate the Pagerank vector, within a tolerance of
            1e-6.

        Only addition to Power iteration is that we restart at seed pages, making
            algorithm focus on nodes relevant to the seed topics.
        '''

        nodes = list(G.nodes())
        n = len(nodes)

        if n == 0:
            return {}

        # Create node-to-index mapping, since networkX doesn't enumerate nodes
        node_to_idx = {node: i for i, node in enumerate(nodes)}
        idx_to_node = {i: node for i, node in enumerate(nodes)}

        # Build adjacency matrix A (for directed graph)
        # A[i,j] = 1 if there's a directed edge FROM node i TO node j
        A = np.zeros((n, n))
        for i, node in enumerate(nodes):
            # For directed graph: check outgoing edges (successors)
            for neighbor in G.successors(node):
                j = node_to_idx[neighbor]
                A[i, j] = 1.0  # Edge from node i to node j

        # Making markov matrix out of adjacency matrix A
        # Compute column sums (out-degrees)
        col_sums = np.sum(A, axis=0)

        # Build transition matrix M: handle dangling nodes (columns with sum 0)
        # For columns with sum > 0: M[i,j] = A[i,j] / col_sum[j]
        # For columns with sum = 0 (dangling nodes): M[i,j] = 1/n (uniform
            distribution)
        M = np.zeros((n, n))
        for j in range(n):
            if col_sums[j] > 0:
                M[:, j] = A[:, j] / col_sums[j]
            else:
                # Dangling node: replace with uniform distribution 1/n
                M[:, j] = 1.0 / n

```

```
        # Create personalization vector v (teleportation vector)
        v = np.zeros(n)
        seed_weight = 1.0 / len(seed_pages)
        for node in nodes:
            if node in seed_pages:
                # Giving equal weight to seed pages, nonzero only for seeds
                v[node_to_idx[node]] = seed_weight
            # Normalize to ensure it sums to exactly 1
        v = v / np.sum(v) if np.sum(v) > 0 else np.ones(n) / n

        # Initialize PageRank vector
        pr = np.ones(n) / n

        # Power method
        for iteration in range(max_iter):
            # Compute new PageRank
            pr_new = (1 - damping) * v + damping * (M @ pr)

            # Check if it is within tolerance
            diff = np.sum(np.abs(pr_new - pr))
            pr = pr_new


            if diff < tol:
                break

        # Normalize entries
        pr = pr / np.sum(pr) if np.sum(pr) > 0 else pr

        # Convert back to dictionary
        pagerank_scores = {idx_to_node[i]: float(pr[i]) for i in range(n)}

        return pagerank_scores
```

For calculation of the PageRank scores for each article, we first need to find the transition matrix M. This was done by creating an adjacency matrix A from our directed graph, and then normalizing by the sum of each column such that $M_{ij} = \frac{A_{ij}}{\sum_{j=0} A_{ij}}$. We also deal with the problem of dangling nodes by replacing them with $\frac{1}{n}$ to ensure a random jump from the side of the web-surfer when faced with a Wiki article with no outgoing links, which ensures that we can find a unique steady state vector w_q when applying our PageRank algorithm.

We are interested in articles related to our seed articles, therefore, we replace $\mathbf{1}$, which is represented as $\begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \in \mathbb{R}^n$, with a personalization vector v, where $v = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$ such that $x_i = 0$ if x_i does not represent the long run probability for one of our selected seed-pages. We assign a non-zero probability for our seed-pages, which we encoded as $\frac{1}{n}$.

As for performing the Personalized PageRank on this directed graph, we used Power Iteration, which was a choice we made due to the scalability of the algorithm. Let p^{t+1} be the $t+1^{\text{th}}$ iteration of our process to find the steady-state. Then, $p^{t+1} = (1-d)v + dMp^t$, where d is the damping factor. We set our tolerance $\epsilon = 10^{-6}$ to be the degree at which we consider our PageRank scores

to have stabilized [1]. A corresponding probability from this vector p will then be the PageRank score for its corresponding webpage.

```
    def visualize_graph(G, pagerank_scores, seed_pages, top_n=10):
        '''
            Visualize the graph with node sizes and colors based on PageRank scores.
        '''
        if G.number_of_nodes() == 0:
            print("Graph is empty, cannot visualize.")
            return

        plt.figure(figsize=(16, 12))

        # Get top influential nodes - sort by score (descending)
        def sort_key(item):
            node, score = item
            # tie breaker for same scores
            return (-score, hash(node) % 1000000)

        sorted_nodes = sorted(pagerank_scores.items(), key=sort_key)
        top_nodes = [node for node, _ in sorted_nodes[:top_n]]

        node_sizes = []
        node_colors = []
        node_labels = {}

        for node in G.nodes():
            # Node size based on PageRank score
            size = pagerank_scores.get(node, 0) * 10000
            node_sizes.append(max(size, 50))  # Minimum size

            # Color: red for seeds, orange for top influential, blue for others
            if node in seed_pages:
                node_colors.append('red')
            elif node in top_nodes:
                node_colors.append('orange')
            else:
                node_colors.append('lightblue')

            # Only label seed pages and top influential nodes
            if node in seed_pages or node in top_nodes:
                # Truncate long titles
                label = node[:30] + '...' if len(node) > 30 else node
                node_labels[node] = label

        # Use spring layout for better visualization
        pos = nx.spring_layout(G, k=1, iterations=50, seed=42)

        # Draw the graph
        nx.draw_networkx_nodes(G, pos, node_size=node_sizes, node_color=node_colors,
            alpha=0.7)
        nx.draw_networkx_edges(G, pos, alpha=0.2, width=0.5, arrows=True,
            arrowsize=10, arrowstyle='->')
        nx.draw_networkx_labels(G, pos, node_labels, font_size=8, font_weight='bold')

        plt.title('Wikipedia Graph with Personalized PageRank\n(Red=Seed, Orange=Top
            Influential, Blue=Others)',
                  fontsize=14, fontweight='bold')
        plt.axis('off')
```

```
        plt.tight_layout()
        plt.show()

        # Print top influential nodes
        print(f"\nTop {top_n} Most Influential Pages (by Personalized PageRank):")
        print("-" * 70)
        prev_score = None
        for i, (node, score) in enumerate(sorted_nodes[:top_n], 1):
            # Show if score is same as previous (tied)
            print(f"{i:2d}. {node[:50]:50s} (Score: {score:.8f})")
            prev_score = score
```

We proceeded with visualizing the graph in terms of influence. Seed pages are given red-coloring, pages with significant PageRank scores are given orange coloring, and every other page is given a blue coloring.

3.5 Code and Algorithm Design: Hyperlink-Induced Topic Search

```
    def hits(G, max_iter=100, tol=1e-6):
        '''
        Runs Hyperlink-Induced Topic Search (HITS)
        Updates authority and hub vectors to convergence until tolerance of 1e-6 is
            reached:
        a_{k+1} = A^T * h_k
        h_{k+1} = A * a_{k+1}
        a_{k+1} = a_{k+1} / ||a_{k+1}||_2
        h_{k+1} = h_{k+1} / ||h_{k+1}||_2
        '''
        nodes = list(G.nodes())
        n = len(nodes)

        if n == 0:
            return {}, {}

        # Create node-to-index mapping, since networkX doesn't enumerate nodes
        node_to_idx = {node: i for i, node in enumerate(nodes)}
        idx_to_node = {i: node for i, node in enumerate(nodes)}

        # Build adjacency matrix A (for directed graph)
        # A[i,j] = 1 if there's a directed edge FROM node i TO node j
        A = np.zeros((n, n))
        for i, node in enumerate(nodes):
            # For directed graph: check outgoing edges (successors)
            for neighbor in G.neighbors(node):
                j = node_to_idx[neighbor]
                A[i, j] = 1.0 # Edge from node i to node j

        # Initialize authority vector and hub vector
        hub = np.random.rand(n) * 1e-6
        auth = np.random.rand(n) * 1e-6

        # Iterative updates of a and h
        for iteration in range(max_iter):
            # Compute new authority vector: a_{k+1} = A^T * h_k
            new_auth = A.T @ hub
            # Compute new hub vector: h_{k+1} = A * a_{k+1}
            new_hub = A @ new_auth
```

```
            # Normalize authority and hub vectors
            a_norm = np.linalg.norm(new_auth, 2)
            h_norm = np.linalg.norm(new_hub, 2)
            if a_norm == 0 or h_norm == 0:
              break
            new_auth = new_auth / a_norm
            new_hub = new_hub / h_norm

            # Check if it is within the tolerance
            diff_a = np.linalg.norm(new_auth - auth, 2)
            diff_h = np.linalg.norm(new_hub - hub, 2)
            auth = new_auth
            hub = new_hub
            if diff_a < tol and diff_h < tol:
              break

        # Convert back to dictionary
        authority_scores = {idx_to_node[i]: float(auth[i]) for i in range(n)}
        hub_scores = {idx_to_node[i]: float(hub[i]) for i in range(n)}

        return authority_scores, hub_scores
```

For each article, we calculated the authority and hub scores through an iterative process. We did so by first constructing an adjacency matrix A from the directed graph using the following formula:

$$A_{ij} = \begin{cases} 1 & \text{if page } i \text{ links to page } j \\ 0 & \text{otherwise} \end{cases}$$

Once the matrix is transformed, we initialized the authority and hub vectors. During each update, the authority vector is reassigned to $\mathbf{a}_{k+1} = A^T\mathbf{h}_k$ and the hub vector is reassigned to $\mathbf{h}_{k+1} = A\mathbf{a}_{k+1}$ at step k. Then, we L2 normalize $\mathbf{a}_{k+1}$ and $\mathbf{h}_{k+1}$, which involves scaling each vector so that the sum of squares of all entries adds to 1. Note that this means the sum of all authority scores and the sum of all hub scores may be greater than 1, since squaring a value between 0 and 1 may be less than the value itself. The normalization inside each loop is important, since without it the values in both vectors would grow without bound. The loop continues until the difference between consecutive updates becomes smaller than the tolerance $\epsilon = 10^{-6}$.

After convergence, the greatest entries of the authority vector identify pages supported by reliable hubs, and the greatest entries of the hub vector identify pages that direct users to strong authorities. The output gives us two rankings tied to the seed set of articles, allowing us to compare how the Wikipedia network's structure differs from the long-run behavior captured by Personalized PageRank.

```
    def visualize_hits(G, authority_scores, hub_scores, seed_pages, top_n=10):
        '''
        Visualize the graph with node sizes and colors based on HITS authority and
            hub scores.
        '''
        if G.number_of_nodes() == 0:
            return

        # Get top influential nodes - sort by score (descending)
        def sort_key(item):
```

```
            node, score = item
            # tie breaker for same scores
            return (-score, hash(node) % 1000000)

        sorted_auth = sorted(authority_scores.items(), key=sort_key)
        sorted_hub = sorted(hub_scores.items(), key=sort_key)
        top_auth = [node for node, _ in sorted_auth[:top_n]]
        top_hub = [node for node, _ in sorted_hub[:top_n]]

        # Use spring layout for better visualization
        pos = nx.spring_layout(G, k=1, iterations=50, seed=42)

        # Function to draw the graph for each type of score
        def draw_graph(title, score_map, top_nodes):
            plt.figure(figsize=(16, 12))

            node_sizes = []
            node_colors = []
            node_labels = {}

            for node in G.nodes():
                # Node size based on HITS authority or hub score
                size = score_map.get(node, 0) * 10000
                node_sizes.append(max(size, 50)) # Minimum size

                # Color: red for seeds, orange for top influential, blue for others
                if node in seed_pages:
                    node_colors.append('red')
                elif node in top_nodes:
                    node_colors.append('orange')
                else:
                    node_colors.append('lightblue')

                # Only label seed pages and top influential nodes
                if node in seed_pages or node in top_nodes:
                    # Truncate long titles
                    lbl = node[:30] + '...' if len(node) > 30 else node
                    node_labels[node] = lbl

            # Draw the graph
            nx.draw_networkx_nodes(G, pos, node_size=node_sizes,
                node_color=node_colors, alpha=0.7)
            nx.draw_networkx_edges(G, pos, alpha=0.2, width=0.5, arrows=True,
                arrowsize=10, arrowstyle='->')
            nx.draw_networkx_labels(G, pos, node_labels, font_size=8,
                font_weight='bold')
            plt.title(title, fontsize=14, fontweight='bold')
            plt.axis('off')
            plt.tight_layout()
            plt.show()

        # Draws graph for authority and hub scores
        draw_graph("Wikipedia Graph with HITS Authority Scores\n(Red=Seed, Orange=Top
            Influential, Blue=Others)", authority_scores, top_auth)
        draw_graph("Wikipedia Graph with HITS Hub Scores\n(Red=Seed, Orange=Top
            Influential, Blue=Others)", hub_scores, top_hub)

        # Print top influential nodes for authorities
        print(f"\nTop {top_n} Most Influential Pages (by HITS Authority Scores):")
```

```
        print("-" * 70)
        prev_score = None
        for i, (node, score) in enumerate(sorted_auth[:top_n], 1):
          # Show if score is same as previous (tied)
          print(f"{i:2d}. {node[:50]:50s} (Score: {score:.8f})")
          prev_score = score

        # Print top influential nodes for hubs
        print(f"\nTop {top_n} Most Influential Pages (by HITS Hub Scores):")
        print("-" * 70)
        prev_score = None
        for i, (node, score) in enumerate(sorted_hub[:top_n], 1):
          # Show if score is same as previous (tied)
          print(f"{i:2d}. {node[:50]:50s} (Score: {score:.8f})")
          prev_score = score
```

Similar to the graph visualization for PageRank, we visualized the graphs for HITS in terms of influence for both authority scores and hub scores, but separately. Like before, seed pages are given red-coloring, pages with significant scores are given orange coloring, and every other page is given a blue coloring.

4 Results and Discussion

We tested our implementation on few local Wikipedia networks with varying sizes and compared results across Personalized PageRank and HITS.

4.1 Small Wikipedia Network

```
# Input params
seed_terms = [
    "planets",
    "supernova",
    "hubble"
]

# Build the graph
G = build_wikipedia_graph(
    seed_terms,
    max_depth=2,
    max_links_per_page=2,
    max_total_pages=200
)

# Initialize pages for seed articles
seed_pages = [node for node, attrs in G.nodes(data=True) if attrs.get('is_seed',
    False)]
```

First, we tested our Personalized PageRank and HITS algorithms with the set of seed terms $S = \{$“planets”, “supernova”, “hubble”$\}$, while setting `max_links` to 2 and `max_links_per_page` to 2 in order to observe algorithmic behavior closely. The `build_wikipedia_graph` function identified the top Wikipedia page search results, matching the seed terms to the pages “Star,” “Planet,” “Supernova,” and “Hubble Space Telescope,” respectively. Then, we ran the algorithms using the functions `personalized_pagerank` and `hits`.

The vectors representing Personalized PageRank scores, HITS authority scores, and HITS hub scores, respectively, are shown below:

$$\begin{bmatrix} 0.16930952 \\ 0.16930952 \\ 0.13846833 \\ 0.05762713 \\ 0.05762713 \\ 0.05762713 \\ 0.05762713 \\ 0.05762713 \\ 0.02134336 \\ 0.02134336 \\ 0.02134336 \\ 0.02134336 \\ 0.02134336 \\ 0.02134336 \\ 0.02134336 \\ 0.02134336 \\ 0.02134336 \\ 0.02134336 \\ 0.02134336 \end{bmatrix}, \begin{bmatrix} 0.36873486 \\ 0.36873486 \\ 0.35883998 \\ 0.35883998 \\ 0.34209901 \\ 0.34209901 \\ 0.20683462 \\ 0.20683462 \\ 0.20355417 \\ 0.20355417 \\ 0.13336998 \\ 0.13336998 \\ 0.12545139 \\ 0.12545139 \\ 0.02227737 \\ 0.02227737 \\ 0.00000000 \\ 0.00000000 \\ 0.00000000 \end{bmatrix}, \begin{bmatrix} 0.52146983 \\ 0.50747636 \\ 0.48380106 \\ 0.29250832 \\ 0.28786906 \\ 0.18861364 \\ 0.17741506 \\ 0.03150495 \\ 0.00000000 \\ 0.00000000 \\ 0.00000000 \\ 0.00000000 \\ 0.00000000 \\ 0.00000000 \\ 0.00000000 \\ 0.00000000 \\ 0.00000000 \\ 0.00000000 \\ 0.00000000 \end{bmatrix}$$

Furthermore, the raw outputs below show the scores corresponding with article names in the network.

Personalized PageRank:

```
1. Hubble Space Telescope (Score: 0.16930952)
2. Planet (Score: 0.16930952)
3. Supernova (Score: 0.13846833)
4. Super Jupiter (Score: 0.05762713)
5. Kepler's laws of planetary motion (Score: 0.05762713)
6. Space Flyer Unit (Score: 0.05762713)
7. Broad Band X ray Telescope (Score: 0.05762713)
8. Guest star (astronomy) (Score: 0.05762713)
9. List of most massive stars (Score: 0.02134336)
10. Kosmos 2312 (Score: 0.02134336)
11. Vis viva equation (Score: 0.02134336)
12. Newton's laws of motion (Score: 0.02134336)
13. Solar Orbiter (Score: 0.02134336)
14. TENKO 100 (Score: 0.02134336)
15. List of proper names of exoplanets (Score: 0.02134336)
16. JAXA (Score: 0.02134336)
17. SPHEREx (Score: 0.02134336)
18. History of supernova observation (Score: 0.02134336)
19. Planetesimal (Score: 0.02134336)
```

HITS authority scores:

```
1. Space Flyer Unit (Score: 0.36873486)
2. Broad Band X ray Telescope (Score: 0.36873486)
3. Super Jupiter (Score: 0.35883998)
4. Kepler's laws of planetary motion (Score: 0.35883998)
5. TENKO 100 (Score: 0.34209901)
6. Guest star (astronomy) (Score: 0.34209901)
7. Vis viva equation (Score: 0.20683462)
8. Newton's laws of motion (Score: 0.20683462)
9. List of proper names of exoplanets (Score: 0.20355417)
10. Planetesimal (Score: 0.20355417)
11. Kosmos 2312 (Score: 0.13336998)
12. JAXA (Score: 0.13336998)
13. Solar Orbiter (Score: 0.12545139)
14. SPHEREx (Score: 0.12545139)
15. List of most massive stars (Score: 0.02227737)
16. History of supernova observation (Score: 0.02227737)
17. Hubble Space Telescope (Score: 0.00000000)
18. Supernova (Score: 0.00000000)
19. Planet (Score: 0.00000000)
```

HITS hub scores:

```
1. Hubble Space Telescope (Score: 0.52146983)
2. Planet (Score: 0.50747636)
3. Supernova (Score: 0.48380106)
4. Kepler's laws of planetary motion (Score: 0.29250832)
5. Super Jupiter (Score: 0.28786906)
6. Space Flyer Unit (Score: 0.18861364)
7. Broad Band X ray Telescope (Score: 0.17741506)
8. Guest star (astronomy) (Score: 0.03150495)
9. List of most massive stars (Score: 0.00000000)
10. Kosmos 2312 (Score: 0.00000000)
11. Vis viva equation (Score: 0.00000000)
12. Newton's laws of motion (Score: 0.00000000)
13. Solar Orbiter (Score: 0.00000000)
14. TENKO 100 (Score: 0.00000000)
15. List of proper names of exoplanets (Score: 0.00000000)
16. JAXA (Score: 0.00000000)
17. SPHEREx (Score: 0.00000000)
18. History of supernova observation (Score: 0.00000000)
19. Planetesimal (Score: 0.00000000)
```

We found that Personalized PageRank emphasizes pages directly related to the seed terms, with "Hubble Space Telescope" (0.16930952), "Planet" (0.16930952), and "Supernova" (0.13846833) at the top due to their centrality across multiple related pages. This aligns with the mathematical intuition that PageRank, especially Personalized PageRank, amplifies nodes with strong link connections back to the seeds.

On the other hand, HITS authority scores lacked seeds as the most influential pages since many Wikipedia articles do not link bidirectionally. In fact, all three of the seed terms had an authority score of 0.00000000, emphasizing that there were no articles that linked back to them. There were a few tied authority scores, which makes sense because they likely had the same amount of articles linking to them. For example, if the page "Supernova" linked to "History of supernova observation," but nothing else linked to this page, it received a score of 0.02227737. This same logic could be applied to "List of most massive stars," and several other pages too. Furthermore, some pages like "Space Flyer Unit" (0.36873486) were highly ranked because many hub pages linked to it.

Hub scores revealed a different pattern. Seed terms like "Hubble Space Telescope", "Planet," and "Supernova" appeared as some of the most influential, because naturally, these pages point to the rest of the articles in the network. Thus, they have high hub scores. In this way, we noticed similarities between HITS hub scores and Personalized PageRank, because both emphasized the seed articles as the most important in the network. However, Personalized PageRank and HITS authority scores differed, as authority showed articles that were linked to the most in the network. Notably, some hub scores had a value of 0.00000000, because these were dangling nodes, lacking outgoing links to authoritative nodes. Since we set our `max_depth` to 2, it makes sense that some articles did not link to others.

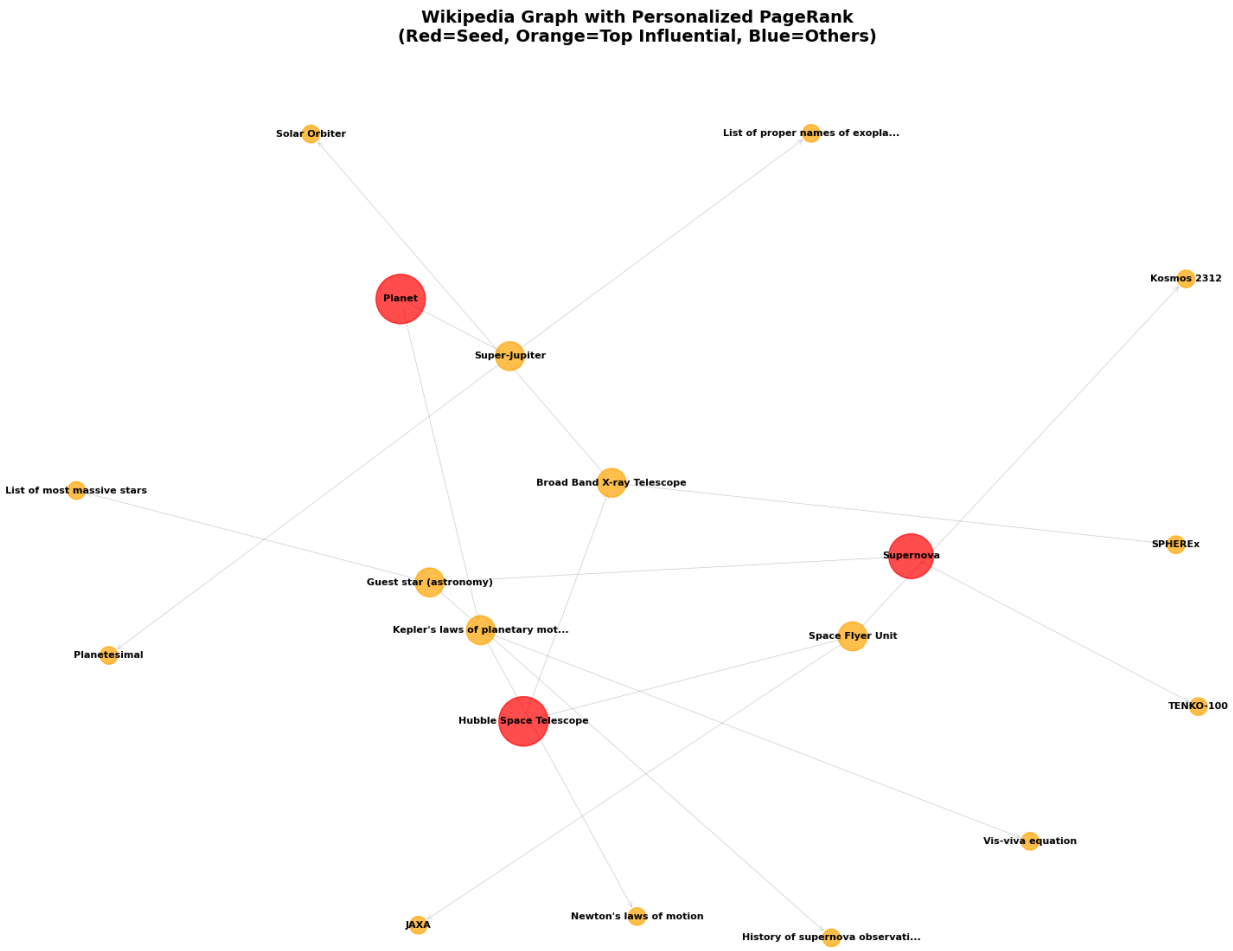

Figure 1: Personalized PageRank visualization for small graph

Figure 1 shows the Wikipedia graph network using Personalized PageRank, with red circles indicating the seed articles, orange indicating top influential, and blue indicating other articles. The proximity of top influential articles to the seeds highlights that Personalized PageRank effectively prioritizes pages that are both strongly linked to the seeds and widely referenced by other nodes. The numerous blue nodes scattered around the edges correspond to pages with low influence scores. Their peripheral placement emphasizes the steep decay in PageRank values, where a few highly central pages dominate the network, while the majority have minimal influence. Thus, the graph concurs with our mathematical results.

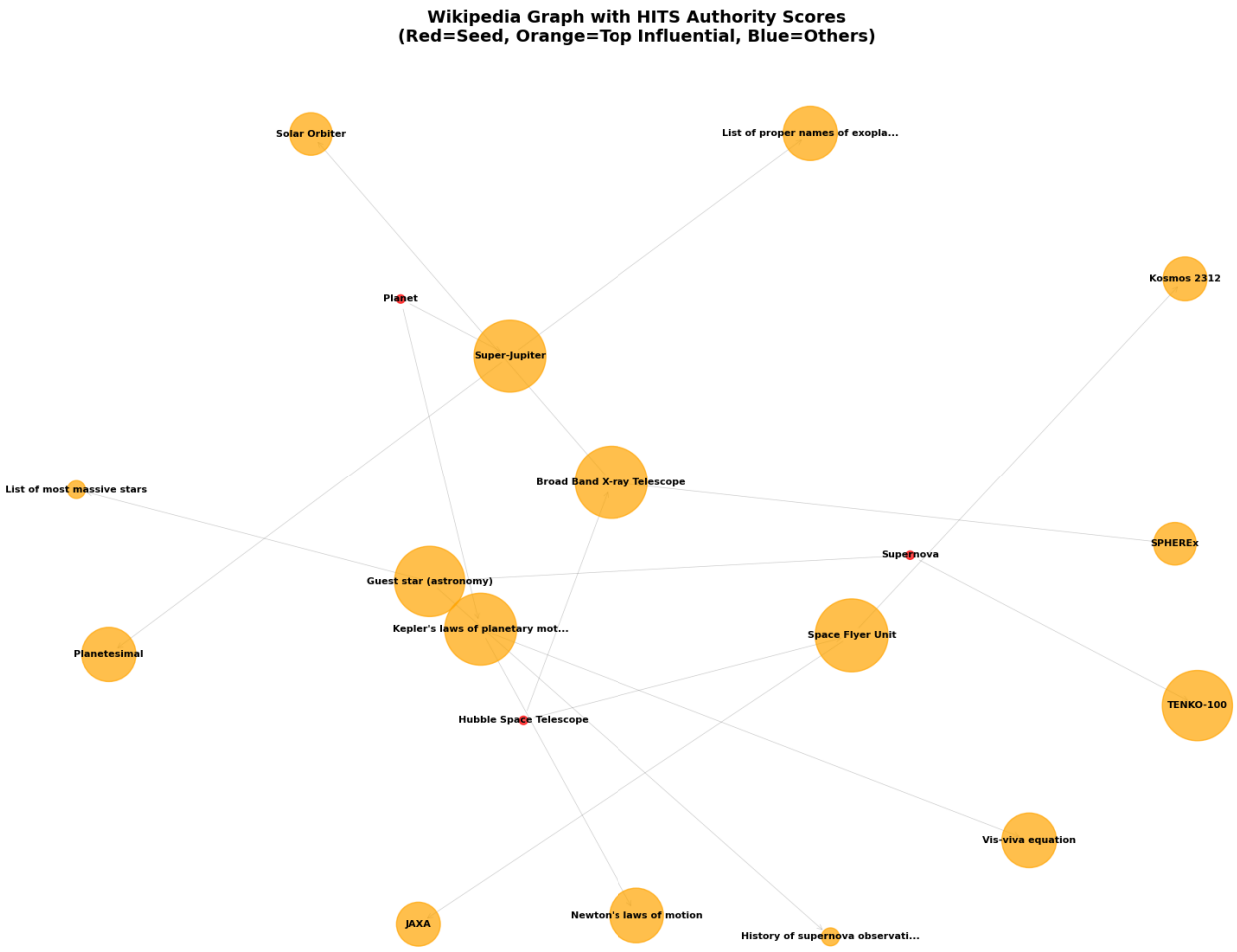

Figure 2: HITS authorities visualization for small graph

Figure 2 illustrates the HITS authority scores for the small Wikipedia network. The seed nodes, represented in red, were significantly smaller than the nodes surrounding them, indicating a lower authority score. As noted before, this makes sense because the seed nodes act as the base, linking to other articles, but since many Wikipedia pages are not bidirectional, they are not necessarily high authority. On the other hand, several peripheral nodes showed high authority since many Wikipedia pages linked to them.

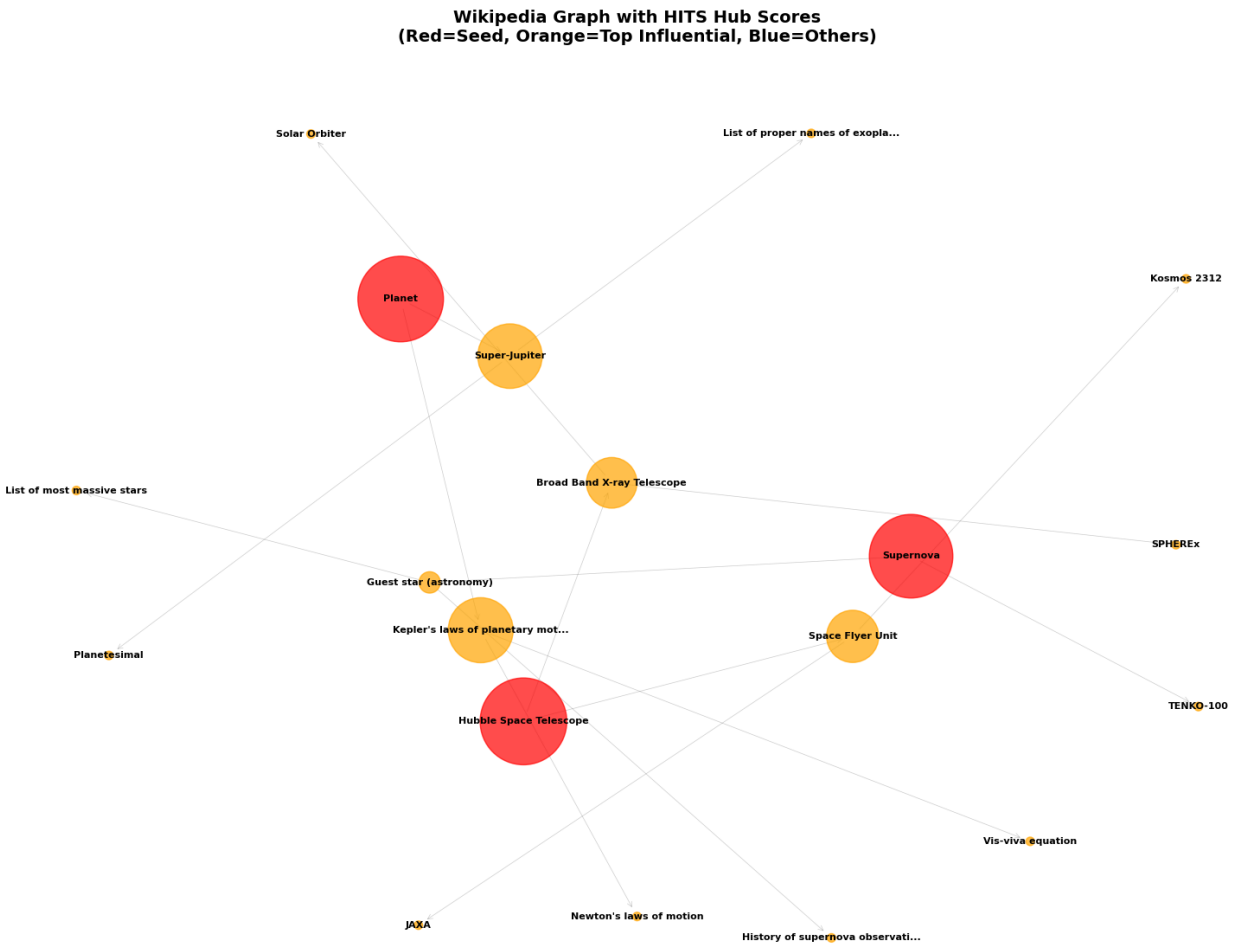

Figure 3: HITS hubs visualization for small graph

Figure 3 shows the HITS hub scores. In contrast to the authority scores, seeds had the highest hub scores because they linked to the most authority pages. This result makes sense because seeds served as the baseline for constructing the graph network. Similar to Personalized PageRank, the peripheral nodes had low scores, because they did not link to many other pages in the graph.

Overall, Personalized PageRank produced sharper concentration around the seed pages, with all three seeds dominating the ranking due to the sparse and highly interconnected structure. HITS produced more varied authority and hub lists, driven by sensitivity of the algorithm to a small network.

4.2 Extension: Medium-Sized Wikipedia Network

```
    # Input params
    seed_terms = [
        "linear algebra",
        "eigenvalues",
        "dot product"
```

```
    ]

    # Build the graph
    G = build_wikipedia_graph(
        seed_terms,
        max_depth=2,
        max_links_per_page=5,
        max_total_pages=200
    )

    # Initialize pages for seed articles
    seed_pages = [node for node, attrs in G.nodes(data=True) if attrs.get('is_seed',
        False)]
```

Next, we ran our implementation on a medium-sized network, with a Wikipedia structure containing 5 `max_links` and 5 `max_links_per_page` as parameters. Here, we used the set of seed terms $S = \{$"linear algebra", "eigenvalues", "dot product"$\}$.

Personalized PageRank found the top 30 most influential pages to be:

```
1. Dot product (Score: 0.13858620)
2. Linear algebra (Score: 0.13858620)
3. Eigenvalues and eigenvectors (Score: 0.13858620)
4. Sparse matrix (Score: 0.01994972)
5. Van der Waerden notation (Score: 0.01994972)
                                    ⋮
28. Moon (Score: 0.00379994)
29. Absolutely convex set (Score: 0.00379994)
30. Finite set (Score: 0.00379994)
```

Also, HITS found these 30 articles to have the highest authority scores:

```
1. Bar (unit) (Score: 0.18171734)
2. Angular momentum (Score: 0.18171734)
3. Multilinear algebra (Score: 0.18171734)
4. Symmetry (Score: 0.18171734)
5. Vector (geometry) (Score: 0.18171734)
                                    ⋮
28. Algebraic variety (Score: 0.13018781)
29. Parallelogram (Score: 0.13018781)
30. Quantum mechanics (Score: 0.13018781)
```

Then, HITS found these 30 articles to have the highest hub scores:

```
1. Cauchy stress tensor (Score: 0.40633233)
2. Arthur Cayley (Score: 0.40071487)
3. Graph theory (Score: 0.34656665)
4. Eigenvalues and eigenvectors (Score: 0.31498772)
5. Hecke eigensheaf (Score: 0.30715994)
```

⋮

28. Moon (Score: 0.00000000)
29. Absolutely convex set (Score: 0.00000000)
30. Finite set (Score: 0.00000000)

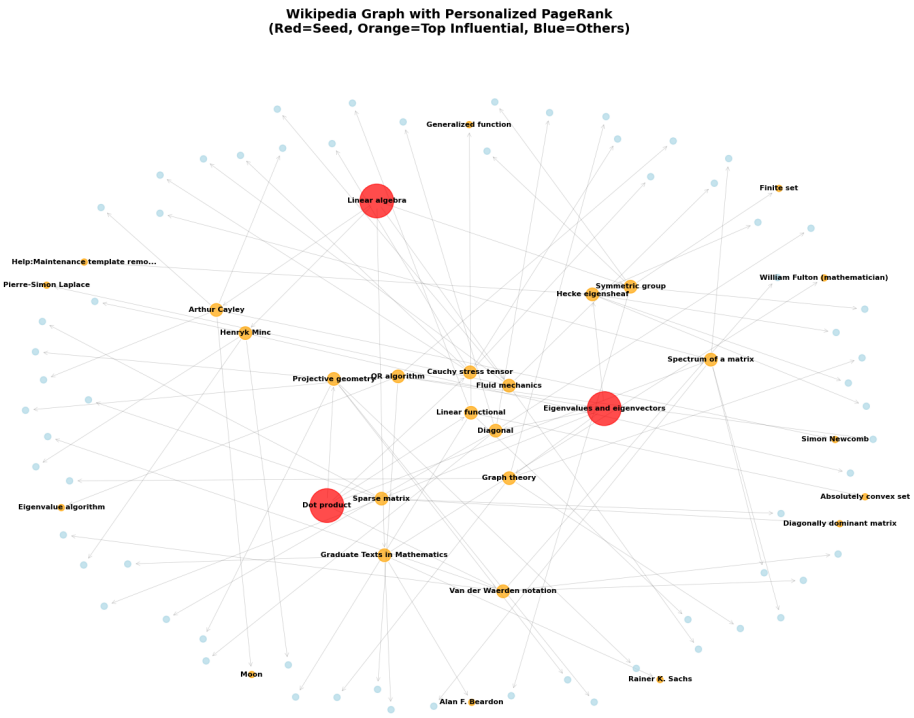

Figure 4: Personalized PageRank visualization for medium-sized graph

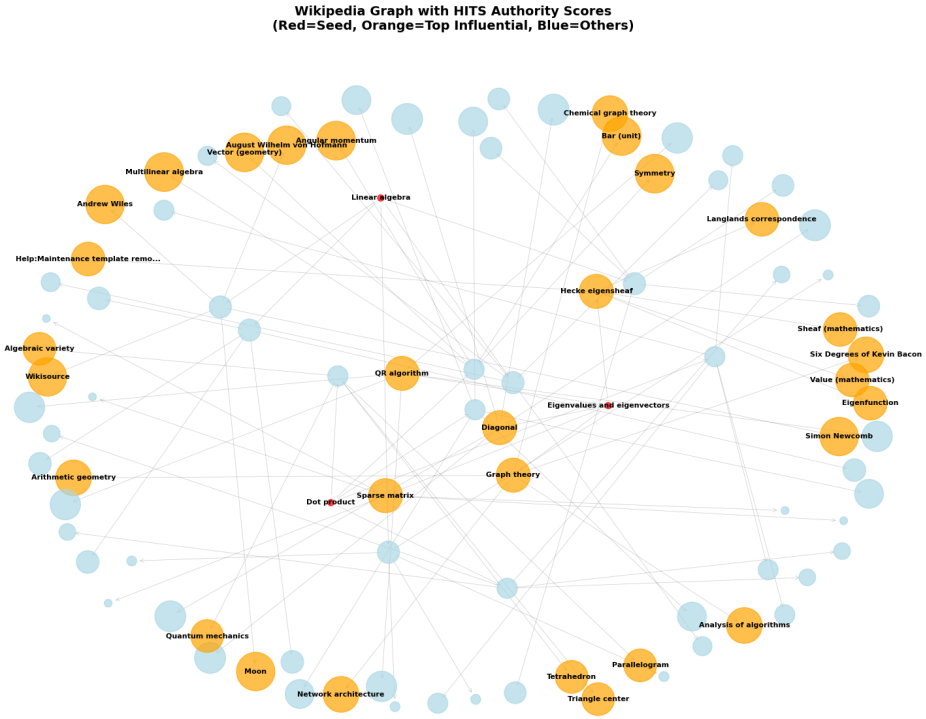

Figure 5: HITS authority visualization for medium-sized graph

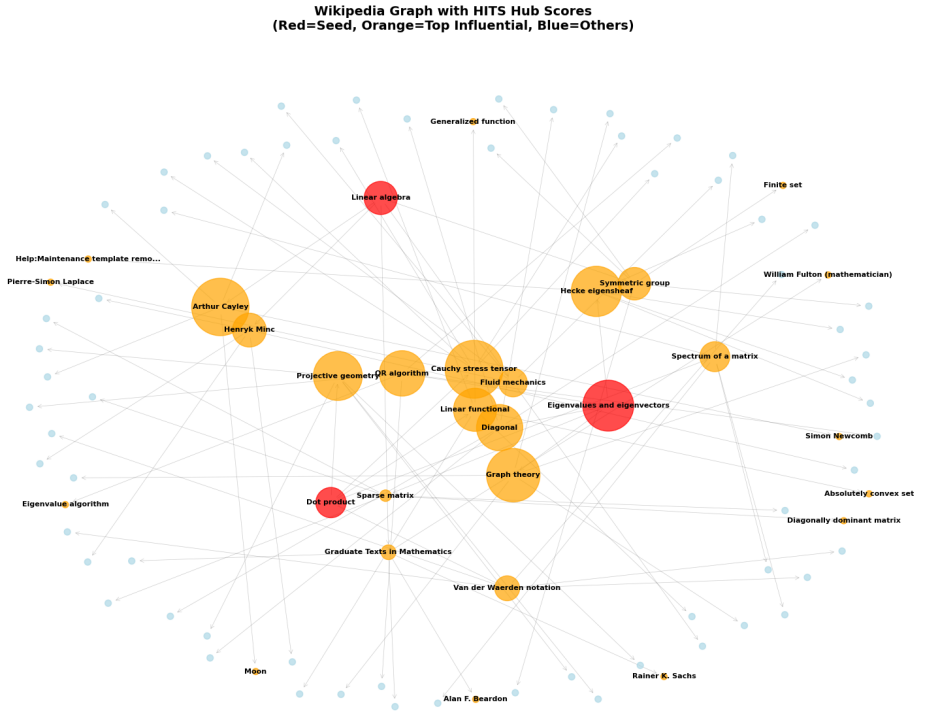

Figure 6: HITS hub visualization for medium-sized graph

For the medium-sized graph, Personalized PageRank continued to prioritize the seeds "Linear algebra," "Eigenvalues and eigenvectors," and "Dot product," localizing strongly around seed-adjacent neighborhoods. Unlike the small network, the effect of larger connectivity becomes more apparent here. Specifically, the decay of PageRank values is sharper for nodes further from the seeds, reflecting the algorithm's tendency to favor nodes within dense clusters, as shown in Figure 4. Even pages like "Sparse matrix" and "Van der Waerden notation" appear with moderate scores, showing how Personalized PageRank can propagate influence. Figure 4 confirms this pattern, with high-scoring nodes tightly clustered near the seeds, and peripheral nodes being loosely connected and less influential.

HITS surfaced with different patterns, however. Authority scores, as seen in Figure 5, prioritize pages that receive links from many other nodes, regardless of proximity to the seeds. This explains why some high-authority pages like "Bar (unit)" and "Angular momentum" appear prominently despite not being seed-adjacent. The medium-sized graph allowed these distinctions to emerge more clearly than in the small network, where limited connectivity often produced ties in scores.

Hub scores, visualized in Figure 6, highlight articles that point to influential authorities, including both seed-related pages and other hubs like "Cauchy stress tensor" or "Graph theory." The variation in hub scores demonstrates how the algorithm captures influence across the network.

4.3 Extension: Large Wikipedia Network

```
    # Input params
    seed_terms = [
        "computer science",
        "artificial intelligence",
        "robotics"
    ]

    # Build the graph
    G = build_wikipedia_graph(
        seed_terms,
        max_depth=5,
        max_links_per_page=8,
        max_total_pages=200
    )

    # Initialize pages for seed articles
    seed_pages = [node for node, attrs in G.nodes(data=True) if attrs.get('is_seed',
        False)]
```

Finally, we ran our implementation on a large network with `max_links` set to 8 and `max_links_per_page` set to 8 as well. Furthermore, we set `max_depth` to 5. We used the set of seed terms $S = \{$"computer science", "artificial intelligence", "robotics"$\}$.

For Personalized PageRank, the 30 most influential articles were:

```
1. Computer science (Score: 0.14327787)
2. Artificial intelligence (Score: 0.14203077)
3. Robotics (Score: 0.12083008)
4. Snake (Score: 0.01346351)
```

```
5. Wireless sensor network (Score: 0.01346351)
                                    ⋮
28. NarrowBand IOT (Score: 0.00172609)
29. Windows 1.0 (Score: 0.00172609)
30. Curt Weldon (Score: 0.00172609)
```

HITS authority scores were the greatest for these 30 articles:

```
1. Empirical observation (Score: 0.12191533)
2. Completely integrable (Score: 0.12191533)
3. Orbital mechanics (Score: 0.12191533)
4. Electric field (Score: 0.12191533)
5. Lunar space elevator (Score: 0.12191533)
                                    ⋮
28. Chimera (Barth novel) (Score: 0.11055698)
29. Invisible Man (Score: 0.11055698)
30. John Casey (novelist) (Score: 0.11055698)
```

Also, HITS hub scores were the greatest for the following 30 articles:

```
1. Glossary of aerospace engineering (Score: 0.34482863)
2. Computer science (Score: 0.32345325)
3. History of the graphical user interface (Score: 0.31970106)
4. Jonathan Franzen (Score: 0.31270236)
5. Atanasoff–Berry computer (Score: 0.28817081)
                                    ⋮
28. NarrowBand IOT (Score: 0.00000000)
29. Windows 1.0 (Score: 0.00000000)
30. Curt Weldon (Score: 0.00000000)
```

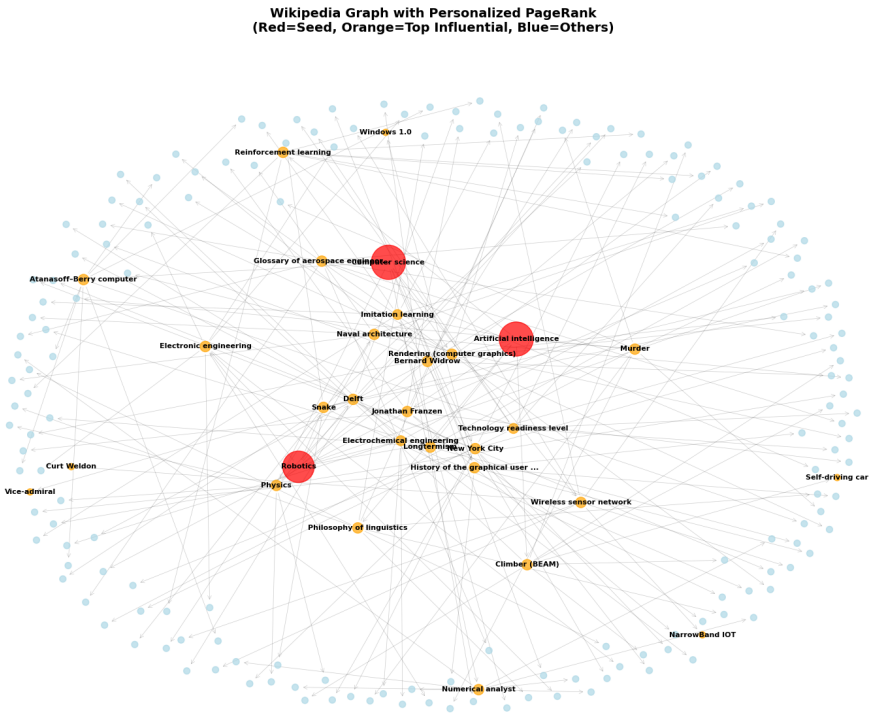

Figure 7: Personalized PageRank visualization for large graph

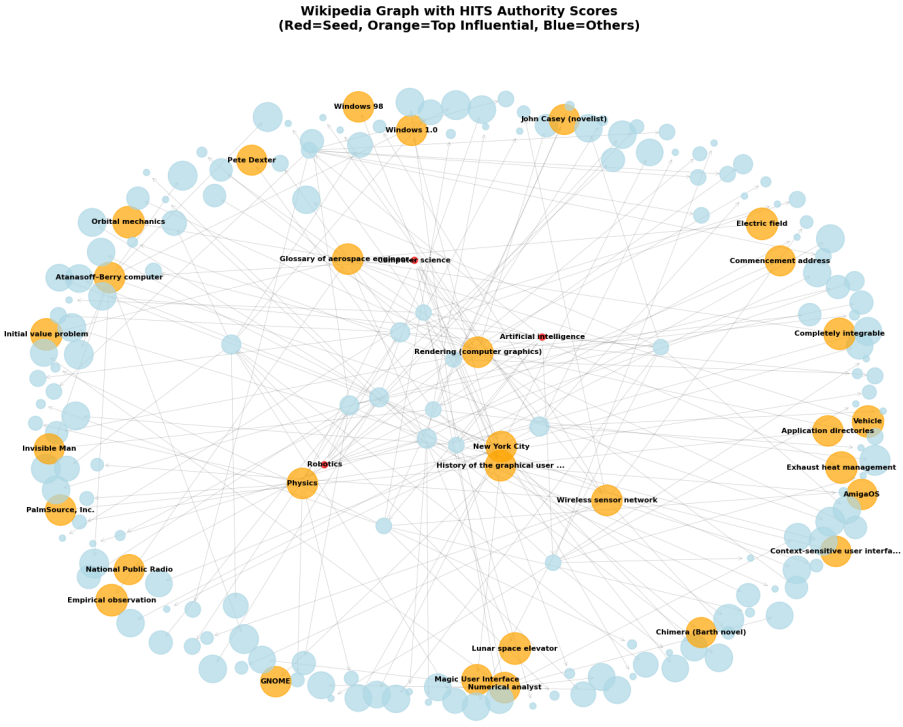

Figure 8: HITS authority visualization for large graph

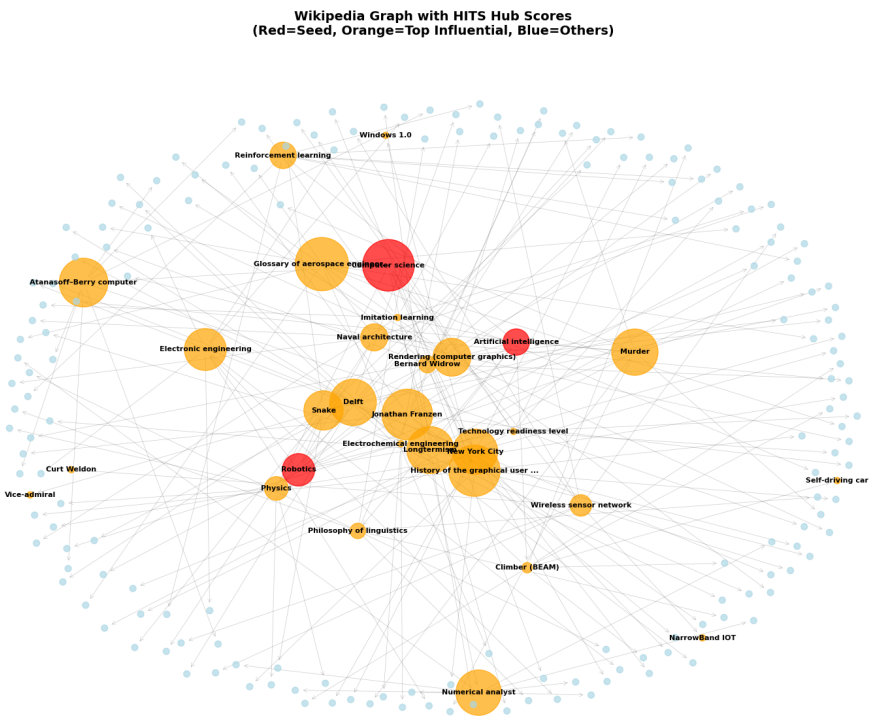

Figure 9: HITS hub visualization for large graph

In the large Wikipedia network, Personalized PageRank continued to prioritize the seed articles. While the seeds still dominated, peripheral nodes such as “Snake” and “Wireless sensor network” received moderate scores, demonstrating that PageRank can propagate influence, as we saw in the medium-sized graph. For the large graph specifically, though, increasing `max_depth` to 5 shows that not all the influential articles directly link to the seeds. Figure 7 confirms this pattern, as the red nodes are central, but there are several clusters of orange nodes that indirectly link to the seeds. However, there were several hubs in the interior of the graph that repeatedly linked to these articles.

HITS authority scores in the large network, shown in Figure 8, reveal more diversity in influential articles. Many high-authority pages like “Empirical observation” and “Orbital mechanics” are not directly connected to the seeds, emphasizing that authority is determined purely by in-links rather than proximity. As the network grows, the separation between authorities and seed-adjacent nodes becomes more pronounced.

Hub scores, depicted in Figure 9, similarly show that seed articles remain highly ranked hubs due to their extensive linking to authority pages.

Overall, we can identify a clear correlation between PageRank and its emphasis on seed proximity, while HITS highlights asymmetric behavior within the subgraph.

5 Conclusion

5.1 What We Learned

Through this project, we learned how different link-analysis algorithms (Personalized PageRank and HITS) capture importance within real-world Wikipedia networks. Personalized PageRank emphasized pages closely tied to the selected seed topics, demonstrating how biasing in teleportation affects random-walk behavior, while HITS distinguished authoritative pages from strong hubs, often identifying influential non-seed nodes from the asymmetry of Wiki links. We understood how graph and seed selection, and algorithmic design shape the ranking outcomes in large networks like this one.

5.2 Limitations

We found that a potential limitation could be the inconsistency of Wikipedia hyperlink structure. Some important articles might be poorly linked, having many outbound links while others have close to none, which could bias the PageRank and HITS scores away from desired influential pages. In particular, PageRank may inflate the importance of seed pages that simply sit in denser local neighborhoods, while HITS may assign low authority scores to pages that are well-known but receive few incoming links. When the seed set S is small, PageRank will overweight the seed i with the largest local neighborhood.

Our set of vertices might also underrepresent the true PageRank score of many of these articles since relevant articles might get omitted.

5.3 Future Work

We will consider performing optimizations on the PageRank algorithm in order to accomodate for large and dense graphs. Considering that PageRank's power iteration method requires repeated matrix multiplication of the Markov matrix $M \in M_{n\times n}(\mathbb{R})$, this process will be computationally expensive when considering large graphs and networks.

A possible extension to this project will be the exploration of low-rank approximations of our stochastic matrix and a smaller graph G' to iterate on, and using these results to approximate the PageRank vector for the original graph G. A potential avenue would be exploring the CUR-trans method, expressing the stochastic matrix M as $M = CUR$ [7], pooling together the most "informative" rows and columns in order to approximate PageRank.

References

[1] Artificial Intelligence - All in One. Lecture 8: Pagerank power iteration — stanford university. YouTube video, 2016. Published April 12.

[2] P. Chau. Pagerank algorithm explained with examples. Medium article, 2023. Published November 14.

[3] P. Lofgren. Efficient algorithms for personalized pagerank, 2015.

[4] Pi Math Cornell. Hits algorithm: Hubs and authorities on the internet, n.d.

[5] Wikipedia Contributors. Hits algorithm, 2019. Page updated May 26.

[6] Wikipedia Contributors. Pagerank, 2019. Page updated June 18.

[7] S. Wu, D. Wu, J. Quan, T. N. Chan, and K. Lu. Efficient and accurate pagerank approximation on large graphs. *Proceedings of the ACM on Management of Data*, 2(4):1–26, 2024.

Appendix I. Code

```
    import wikipediaapi
    import networkx as nx
    import matplotlib.pyplot as plt
    import numpy as np
    from collections import deque
    import time
    import requests
    import json
    import random
    # Included all necessary libraries. NetworkX used for graph creation. WikiAPI
    # used for scraping necessary data. Matplotlib and Numpy for
        calculation/visualization

    # Initialize Wikipedia API
    wiki_wiki = wikipediaapi.Wikipedia(
        user_agent='WikipediaGraphAnalysis/1.0 (https://example.com)',
        language='en',
        extract_format=wikipediaapi.ExtractFormat.WIKI
    )

    def search_wikipedia_first_result(search_term):
        '''
          Searched for the very first result for the search term.
          Doing a search instead of direct lookup since syntactically
          whatever we enter as a seed might not be the exact page name,
          so this helps us circumvent that.
        '''
        try:
            url = "https://en.wikipedia.org/w/api.php"
            params = {
                "action": "query",
                "list": "search",
                "srsearch": search_term,
                "srlimit": 1,
                "format": "json",
                "srnamespace": 0
            }

            headers = {
                'User-Agent': 'WikipediaGraphAnalysis/1.0 (https://example.com;
                    contact@example.com)'
            }

            response = requests.get(url, params=params, timeout=15, headers=headers)

            # Check response status
            if response.status_code != 200:
                print(f"HTTP {response.status_code} for '{search_term}'")
                return None

            # Parsing JSON
            try:
                data = response.json()
            except ValueError as e:
                print(f"Response preview")
                return None
```

```
            if "query" in data and "search" in data["query"] and
                len(data["query"]["search"]) > 0:
                title = data["query"]["search"][0]["title"]
                page = wiki_wiki.page(title)
                if page.exists():
                    return title
            return None
        except Exception as e:
            print(f"Error searching")
            return None

        return None

    def get_wiki_links(page_title, max_links=50):
        '''
            Get internal Wiki links from a page.
            Based on parameters that we tuned, especially
            the maximum number of links, which is discussed
            more thoroughly in the report. Filtering for actual
            links and not files or categories.

            Randomized since we initially found that
            our links were returned in alphabetical order
            in the event of a tie between pagerank scores. Randomized to introduce
                variation in topics chosen.
        '''
        try:
            page = wiki_wiki.page(page_title)

            # Collect all valid links
            all_links = []
            for link_title in page.links.keys():
                if link_title and not link_title.startswith('Category:') and not
                    link_title.startswith('File:') and not
                    link_title.startswith('Template:'):
                    all_links.append(link_title)

            # Randomize to avoid uniformity, then returning up until the maximum
                amount of links
            random.shuffle(all_links)
            return all_links[:max_links]
        except Exception as e:
            print(f"Error getting links")
            return []

    def build_wikipedia_graph(seed_terms, max_depth=2, max_links_per_page=20,
        max_total_pages=200):
        '''
            Build a directed graph from Wikipedia pages starting with seed terms. (u,
                w) and (w, u) valid edges in this graph.
            - seed_terms: List of search terms
            - max_depth: How many levels of links to follow
            - max_links_per_page: Maximum links to extract per page
            - max_total_pages: Maximum total pages to include in graph
        '''
        G = nx.DiGraph()
        # BFS Mark set to check membership easily
        visited = set()
```

```
        seed_pages = []

        for term in seed_terms:
            page_title = search_wikipedia_first_result(term)
            if page_title:
                print(f"Found: '{term}' -> '{page_title}'")
                seed_pages.append(page_title)
                visited.add(page_title)
                G.add_node(page_title, is_seed=True)
            else:
                print(f"No result found for '{term}'")

        # Running BFS
        queue = deque([(page, 0) for page in seed_pages])  # (page_title, depth)

        while queue and len(visited) < max_total_pages:
            current_page, depth = queue.popleft()

            if depth >= max_depth:
                continue

            # Stop if we have reached the maximum number of pages
            if len(visited) >= max_total_pages:
                break

            # Get links from current page - use max_links_per_page parameter
            links = get_wiki_links(current_page, max_links=max_links_per_page)

            for link in links:
                # Stop adding nodes if we've reached the limit
                if len(visited) >= max_total_pages:
                    break

                if link != current_page:
                    if link not in visited and len(visited) < max_total_pages:
                        # Can add new node if the link is not already in visited
                        visited.add(link)
                        G.add_node(link, is_seed=False)
                        G.add_edge(current_page, link)

                        # Add to queue if we haven't reached max depth
                        if depth + 1 < max_depth:
                            queue.append((link, depth + 1))

            time.sleep(0.1) # To avoid timeouts from excessive API calls

        return G

    def personalized_pagerank(G, seed_pages, damping=0.85, max_iter=100, tol=1e-6):
        '''
        Runs Personalized PageRank

        Using power iteration to calculate the Pagerank vector, within a tolerance of
            1e-6.

        Only addition to Power iteration is that we restart at seed pages, making
            algorithm focus on nodes relevant to the seed topics.
        '''

```

```
        nodes = list(G.nodes())
        n = len(nodes)

        if n == 0:
            return {}

        # Create node-to-index mapping, since networkX doesn't enumerate nodes
        node_to_idx = {node: i for i, node in enumerate(nodes)}
        idx_to_node = {i: node for i, node in enumerate(nodes)}

        # Build adjacency matrix A (for directed graph)
        # A[i,j] = 1 if there's a directed edge FROM node i TO node j
        A = np.zeros((n, n))
        for i, node in enumerate(nodes):
            # For directed graph: check outgoing edges (successors)
            for neighbor in G.successors(node):
                j = node_to_idx[neighbor]
                A[i, j] = 1.0  # Edge from node i to node j

        # Making markov matrix out of adjacency matrix A
        # Compute column sums (out-degrees)
        col_sums = np.sum(A, axis=0)

        # Build transition matrix M: handle dangling nodes (columns with sum 0)
        # For columns with sum > 0: M[i,j] = A[i,j] / col_sum[j]
        # For columns with sum = 0 (dangling nodes): M[i,j] = 1/n (uniform
            distribution)
        M = np.zeros((n, n))
        for j in range(n):
            if col_sums[j] > 0:
                M[:, j] = A[:, j] / col_sums[j]
            else:
                # Dangling node: replace with uniform distribution 1/n
                M[:, j] = 1.0 / n

        # Create personalization vector v (teleportation vector)
        v = np.zeros(n)
        seed_weight = 1.0 / len(seed_pages)
        for node in nodes:
            if node in seed_pages:
                # Giving equal weight to seed pages, nonzero only for seeds
                v[node_to_idx[node]] = seed_weight
            # Normalize to ensure it sums to exactly 1
        v = v / np.sum(v) if np.sum(v) > 0 else np.ones(n) / n

        # Initialize PageRank vector
        pr = np.ones(n) / n

        # Power method
        for iteration in range(max_iter):
            # Compute new PageRank
            pr_new = (1 - damping) * v + damping * (M @ pr)

            # Check if it is within tolerance
            diff = np.sum(np.abs(pr_new - pr))
            pr = pr_new


            if diff < tol:
```

```
                break

        # Normalize entries
        pr = pr / np.sum(pr) if np.sum(pr) > 0 else pr

        # Convert back to dictionary
        pagerank_scores = {idx_to_node[i]: float(pr[i]) for i in range(n)}

        return pagerank_scores

    def visualize_graph(G, pagerank_scores, seed_pages, top_n=10):
        '''
            Visualize the graph with node sizes and colors based on PageRank scores.
        '''
        if G.number_of_nodes() == 0:
            print("Graph is empty, cannot visualize.")
            return

        plt.figure(figsize=(16, 12))

        # Get top influential nodes - sort by score (descending)
        def sort_key(item):
            node, score = item
            # tie breaker for same scores
            return (-score, hash(node) % 1000000)

        sorted_nodes = sorted(pagerank_scores.items(), key=sort_key)
        top_nodes = [node for node, _ in sorted_nodes[:top_n]]

        node_sizes = []
        node_colors = []
        node_labels = {}

        for node in G.nodes():
            # Node size based on PageRank score
            size = pagerank_scores.get(node, 0) * 10000
            node_sizes.append(max(size, 50))  # Minimum size

            # Color: red for seeds, orange for top influential, blue for others
            if node in seed_pages:
                node_colors.append('red')
            elif node in top_nodes:
                node_colors.append('orange')
            else:
                node_colors.append('lightblue')

            # Only label seed pages and top influential nodes
            if node in seed_pages or node in top_nodes:
                # Truncate long titles
                label = node[:30] + '...' if len(node) > 30 else node
                node_labels[node] = label

        # Use spring layout for better visualization
        pos = nx.spring_layout(G, k=1, iterations=50, seed=42)

        # Draw the graph
        nx.draw_networkx_nodes(G, pos, node_size=node_sizes, node_color=node_colors,
            alpha=0.7)
```

```
        nx.draw_networkx_edges(G, pos, alpha=0.2, width=0.5, arrows=True,
            arrowsize=10, arrowstyle='->')
        nx.draw_networkx_labels(G, pos, node_labels, font_size=8, font_weight='bold')

        plt.title('Wikipedia Graph with Personalized PageRank\n(Red=Seed, Orange=Top
            Influential, Blue=Others)',
                  fontsize=14, fontweight='bold')
        plt.axis('off')
        plt.tight_layout()
        plt.show()

        # Print top influential nodes
        print(f"\nTop {top_n} Most Influential Pages (by Personalized PageRank):")
        print("-" * 70)
        prev_score = None
        for i, (node, score) in enumerate(sorted_nodes[:top_n], 1):
            # Show if score is same as previous (tied)
            print(f"{i:2d}. {node[:50]:50s} (Score: {score:.8f})")
            prev_score = score

    def hits(G, max_iter=100, tol=1e-6):
        '''
        Runs Hyperlink-Induced Topic Search (HITS)
        Updates authority and hub vectors to convergence until tolerance of 1e-6 is
            reached:
        a_{k+1} = A^T * h_k
        h_{k+1} = A * a_{k+1}
        a_{k+1} = a_{k+1} / ||a_{k+1}||_2
        h_{k+1} = h_{k+1} / ||h_{k+1}||_2
        '''
        nodes = list(G.nodes())
        n = len(nodes)

        if n == 0:
            return {}, {}

        # Create node-to-index mapping, since networkX doesn't enumerate nodes
        node_to_idx = {node: i for i, node in enumerate(nodes)}
        idx_to_node = {i: node for i, node in enumerate(nodes)}

        # Build adjacency matrix A (for directed graph)
        # A[i,j] = 1 if there's a directed edge FROM node i TO node j
        A = np.zeros((n, n))
        for i, node in enumerate(nodes):
            # For directed graph: check outgoing edges (successors)
            for neighbor in G.neighbors(node):
                j = node_to_idx[neighbor]
                A[i, j] = 1.0 # Edge from node i to node j

        # Initialize authority vector and hub vector
        hub = np.random.rand(n) * 1e-6
        auth = np.random.rand(n) * 1e-6

        # Iterative updates of a and h
        for iteration in range(max_iter):
            # Compute new authority vector: a_{k+1} = A^T * h_k
            new_auth = A.T @ hub
            # Compute new hub vector: h_{k+1} = A * a_{k+1}
            new_hub = A @ new_auth
```

```
            # Normalize authority and hub vectors
            a_norm = np.linalg.norm(new_auth, 2)
            h_norm = np.linalg.norm(new_hub, 2)
            if a_norm == 0 or h_norm == 0:
              break
            new_auth = new_auth / a_norm
            new_hub = new_hub / h_norm

            # Check if it is within the tolerance
            diff_a = np.linalg.norm(new_auth - auth, 2)
            diff_h = np.linalg.norm(new_hub - hub, 2)
            auth = new_auth
            hub = new_hub
            if diff_a < tol and diff_h < tol:
              break

        # Convert back to dictionary
        authority_scores = {idx_to_node[i]: float(auth[i]) for i in range(n)}
        hub_scores = {idx_to_node[i]: float(hub[i]) for i in range(n)}

        return authority_scores, hub_scores

    def visualize_hits(G, authority_scores, hub_scores, seed_pages, top_n=10):
        '''
        Visualize the graph with node sizes and colors based on HITS authority and
            hub scores.
        '''
        if G.number_of_nodes() == 0:
            return

        # Get top influential nodes - sort by score (descending)
        def sort_key(item):
            node, score = item
            # tie breaker for same scores
            return (-score, hash(node) % 1000000)

        sorted_auth = sorted(authority_scores.items(), key=sort_key)
        sorted_hub = sorted(hub_scores.items(), key=sort_key)
        top_auth = [node for node, _ in sorted_auth[:top_n]]
        top_hub = [node for node, _ in sorted_hub[:top_n]]

        # Use spring layout for better visualization
        pos = nx.spring_layout(G, k=1, iterations=50, seed=42)

        # Function to draw the graph for each type of score
        def draw_graph(title, score_map, top_nodes):
            plt.figure(figsize=(16, 12))

            node_sizes = []
            node_colors = []
            node_labels = {}

            for node in G.nodes():
                # Node size based on HITS authority or hub score
                size = score_map.get(node, 0) * 10000
                node_sizes.append(max(size, 50)) # Minimum size

                # Color: red for seeds, orange for top influential, blue for others
```

```
                if node in seed_pages:
                    node_colors.append('red')
                elif node in top_nodes:
                    node_colors.append('orange')
                else:
                    node_colors.append('lightblue')

                # Only label seed pages and top influential nodes
                if node in seed_pages or node in top_nodes:
                    # Truncate long titles
                    lbl = node[:30] + '...' if len(node) > 30 else node
                    node_labels[node] = lbl

            # Draw the graph
            nx.draw_networkx_nodes(G, pos, node_size=node_sizes,
                node_color=node_colors, alpha=0.7)
            nx.draw_networkx_edges(G, pos, alpha=0.2, width=0.5, arrows=True,
                arrowsize=10, arrowstyle='->')
            nx.draw_networkx_labels(G, pos, node_labels, font_size=8,
                font_weight='bold')
            plt.title(title, fontsize=14, fontweight='bold')
            plt.axis('off')
            plt.tight_layout()
            plt.show()

        # Draws graph for authority and hub scores
        draw_graph("Wikipedia Graph with HITS Authority Scores\n(Red=Seed, Orange=Top
            Influential, Blue=Others)", authority_scores, top_auth)
        draw_graph("Wikipedia Graph with HITS Hub Scores\n(Red=Seed, Orange=Top
            Influential, Blue=Others)", hub_scores, top_hub)

        # Print top influential nodes for authorities
        print(f"\nTop {top_n} Most Influential Pages (by HITS Authority Scores):")
        print("-" * 70)
        prev_score = None
        for i, (node, score) in enumerate(sorted_auth[:top_n], 1):
          # Show if score is same as previous (tied)
          print(f"{i:2d}. {node[:50]:50s} (Score: {score:.8f})")
          prev_score = score

        # Print top influential nodes for hubs
        print(f"\nTop {top_n} Most Influential Pages (by HITS Hub Scores):")
        print("-" * 70)
        prev_score = None
        for i, (node, score) in enumerate(sorted_hub[:top_n], 1):
          # Show if score is same as previous (tied)
          print(f"{i:2d}. {node[:50]:50s} (Score: {score:.8f})")
          prev_score = score

    # Input params
    seed_terms = [
        "planets",
        "supernova",
        "hubble"
    ]

    # Build the graph
    G = build_wikipedia_graph(
        seed_terms,
```

```
        max_depth=2,
        max_links_per_page=2,
        max_total_pages=200
    )

    # Initialize pages for seed articles
    seed_pages = [node for node, attrs in G.nodes(data=True) if attrs.get('is_seed',
        False)]

    # Calculate PageRank scores
    pagerank_scores = personalized_pagerank(G, seed_pages, damping=0.85)
    # Visualize the graph
    visualize_graph(G, pagerank_scores, seed_pages, top_n=30)

    # Calculate HITS authority and hub scores
    authority_scores, hub_scores = hits(G)
    # Visualize the graphs
    visualize_hits(G, authority_scores, hub_scores, seed_pages, top_n=30)
```