Computational Linear Algebra: Homework 3

Academic Year: 2025/2026

Team Members:

1. Indiano, Giovanni (357942);
2. Stradiotti, Fabio (359415).

0. Import and Helper Functions

In [65]:
```
import numpy as np
from scipy import sparse
from scipy.sparse import linalg as splinalg
import matplotlib.pyplot as plt
from IPython.display import Image, display
```

In [66]:
```
def read_dat(file_name):
    labels = {}
    row_indices = [] # Lists to store sparse matrix coordinates
    col_indices = []

    try:
        with open(file_name, 'r') as file:
            first_line = file.readline().strip()
            if not first_line:
                 return None, None
            parts = first_line.split()
            num_nodes = int(parts[0])
            num_edges = int(parts[1])

            # COO for construction (efficient for appending)
            # Later convert to CSC for calculation

            for _ in range(num_nodes):
                line = file.readline().strip()
                if line:
                    parts = line.split(maxsplit=1)
                    node_id = int(parts[0])
                    node_name = parts[1]
                    labels[node_id] = node_name

            for _ in range(num_edges):
                line = file.readline().strip()
                if line:
                    parts = line.split()
                    source = int(parts[0])
                    target = int(parts[1])
                    # Store coordinates instead of filling
                    # dense matrix directly
                    row_indices.append(target - 1)
                    col_indices.append(source - 1)

            # Create sparse matrix with 1s at specific coordinates
            data = np.ones(len(row_indices))
            A = sparse.coo_matrix(
                (data, (row_indices, col_indices)),
                shape=(num_nodes, num_nodes)).tocsc()

            # Efficient column normalization for sparse matrix
            # Calculate sum of each column
            col_sums = np.array(A.sum(axis=0)).flatten()

            # If sum is 0, scaling factor is 0.
            # This leaves the column as a zero vector in A,
            # which is correct because we handle the dangling
            # mass explicitly in the power_iteration function
            with np.errstate(divide='ignore', invalid='ignore'):
                scale_factors = np.where(
                    col_sums != 0, 1.0 / col_sums, 0)
```

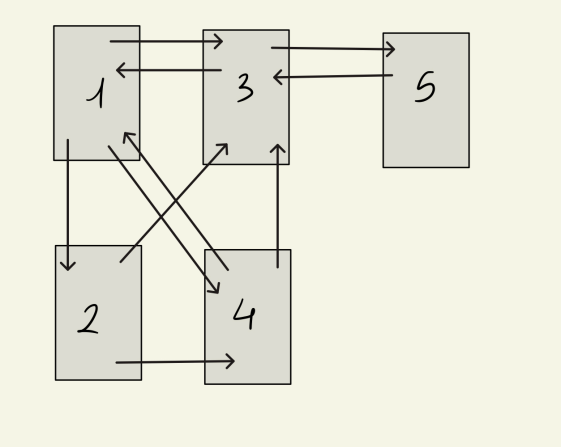
```
            # Multiply A by D_inv from the right
            D_inv = sparse.diags(scale_factors)
            A = A @ D_inv

    except FileNotFoundError:
        print(f"Error: File '{file_name}' not found.")
        return None, None
    except Exception as e:
        print(f"Error during the analysis of the file: {e}")
        return None, None

    return A.toarray(), labels

def power_iteration_with_A(A, tolerance=1e-6, max_iterations=1000):
    n = A.shape[0]
    x = np.ones(n) / n # initial vector (normalized)

    for iteration in range(max_iterations):
        # 1. Compute contribution from existing links
        # Mass from dangling nodes is lost here
        # because their columns are 0
        x_new = A @ x

        # Check convergence (L1 norm)
        if np.linalg.norm(x_new - x, 1) < tolerance:
            print(f"  Converged in {iteration + 1} iterations")
            break
        x = x_new
    else:
        print(f"  Warning: Max iter ({max_iterations}) reached")

    return x

def power_iteration_with_M(A, s, m, tolerance=1e-6,
                           max_iterations=1000):
    n = A.shape[0]
    x = np.ones(n) / n # initial vector (normalized)

    # Identify dangling nodes indices (columns that sum to zero)
    col_sums = np.array(A.sum(axis=0)).flatten()
    dangling_indices = np.where(col_sums == 0)[0]

    for iteration in range(max_iterations):
        # 1. Compute contribution from existing links
        # Mass from dangling nodes is lost
        # here because their columns are 0
        Ax = A @ x

        # Compute contribution from dangling nodes
        dangling_contribution = np.sum(x[dangling_indices])

        # Apply the Google PageRank formula with
        # dangling node mass redistribution
        x_new = (1-m)*Ax + (1-m)*dangling_contribution*s+m*s

        # Check convergence (L1 norm)
        if np.linalg.norm(x_new - x, 1) < tolerance:
            print(f"  Converged in {iteration + 1} iterations")
            break
        x = x_new
    else:
        print(f"  Warning: Max iter ({max_iterations}) reached")

    return x

def get_eigenpairs(A, k=None):
    # Helper function: Use Scipy for large
    # matrices, Numpy for small ones.
    # Scipy eigs requires k < N-1, which
    # fails on very small graphs (e.g., 4 nodes).
    n = A.shape[0]
    if k is None: k = min(n - 2, 6)
    if k < 1: k = 1

    if n < 15 or k >= n-1:
        # Fallback to dense for small graphs
```

```
        # or when many eigenvalues are needed
        vals, vecs = np.linalg.eig(A)
    else:
        try:
            vals, vecs = splinalg.eigs(A, k=k, which='LM')
        except:
            vals, vecs = np.linalg.eig(A)
    return vals, vecs

def analyze_graph(filename, google_matrix=True,
                  m=0.15, print_top_k=None):
    print(f"Analyzing {filename} ...")
    A, labels = read_dat(filename)
    if A is None: return None, None

    n = A.shape[0]
    s = np.ones(n) / n # Uniform personalization vector

    # 1. Check Dangling Nodes
    col_sums = np.array(A.sum(axis=0)).flatten()
    dangling_indices = np.where(col_sums == 0)[0]
    if len(dangling_indices) > 0:
        len_d = len(dangling_indices)
        print(f"  - Warning: Found {len_d} dangling node(s).")
    else:
        print(f"  - No dangling nodes detected.")

    # 2. Compute PageRank
    if google_matrix:
        x = power_iteration_with_M(A, s, m)
    else:
        x = power_iteration_with_A(A)

    # 3. Display Results
    sorted_indices = np.argsort(x)[::-1]

    print(f"  PageRank scores (Top results):")
    print(f"  {'-'*40}")

    limit = print_top_k if print_top_k else n
    for rank, idx in enumerate(sorted_indices[:limit], 1):
        node_label = labels[idx + 1]
        score = x[idx]
        print(f"  {rank}. {node_label:20s}: {score:.6f}")

    if print_top_k and n > print_top_k:
        print(f"  ... (and {n - print_top_k} more)")

    print("\n" + "="*60 + "\n")
    return
```

Exercise 1: Graph 1 Analysis

Problem statement: Suppose the people who own page 3 in the web of Figure 1 are infuriated by the fact that its importance score, computed using formula (2.1), is lower than the score of page 1. In an attempt to boost page 3's score, they create a page 5 that links to page 3; page 3 also links to page 5. Does this boost page 3's score above that of page 1?

In [67]:
```
display(Image(filename="Graphs/Graph1_modified.png", width=350))
```



In [68]:
```
# Original Graph 1
print("Original Graph 1:")
filename="Graphs/graph1.dat"
print(f"Ranking using Matrix A (Raw Link Structure)")
analyze_graph(filename, google_matrix=False, m=0.15)

# Graph 1 with Node 5
print("Graph 1 with Node 5:")
filename="Graphs/exercise1_graph.dat"
print(f"Ranking using Matrix A (Raw Link Structure)")
analyze_graph(filename, google_matrix=False, m=0.15)
```

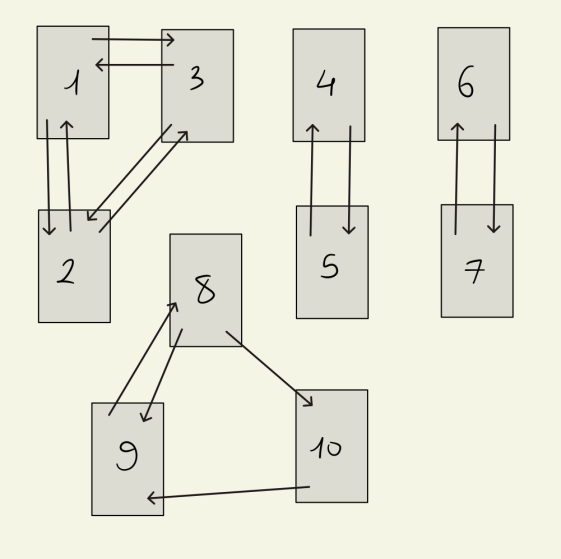
```
Original Graph 1:
Ranking using Matrix A (Raw Link Structure)
Analyzing Graphs/graph1.dat ...
  - No dangling nodes detected.
  Converged in 23 iterations
  PageRank scores (Top results):
  ----------------------------------------
  1. Node1               : 0.387097
  2. Node3               : 0.290323
  3. Node4               : 0.193548
  4. Node2               : 0.129032

============================================================

Graph 1 with Node 5:
Ranking using Matrix A (Raw Link Structure)
Analyzing Graphs/exercise1_graph.dat ...
  - No dangling nodes detected.
  Converged in 42 iterations
  PageRank scores (Top results):
  ----------------------------------------
  1. Node3               : 0.367347
  2. Node1               : 0.244898
  3. Node5               : 0.183673
  4. Node4               : 0.122449
  5. Node2               : 0.081633

============================================================
```

We can see that the addition of Page 5 created a self-reinforcing feedback loop that allowed Page 3 to successfully manipulate the ranking system and overtake Page 1.

Exercise 2

Problem statement: Construct a web consisting of three or more subwebs and verify that $\dim(V_1(\mathbf{A}))$ equals (or exceeds) the number of the components in the web.

In [69]:
```
display(Image(filename="Graphs/exercise2_graph.png", width=350))
```



In [70]:
```
filename="Graphs/exercise2_graph.dat"
A, labels = read_dat(filename)

# Use dense fallback for accurate counting
# of multiplicity on small graphs
eigenvalues, eigenvectors = get_eigenpairs(A)
dimension = np.sum(np.isclose(eigenvalues, 1))
print(f"The dimension of the eigenspace associated", end=" ")
print(f"with the eigenvalue 1 is: {dimension}.")
```

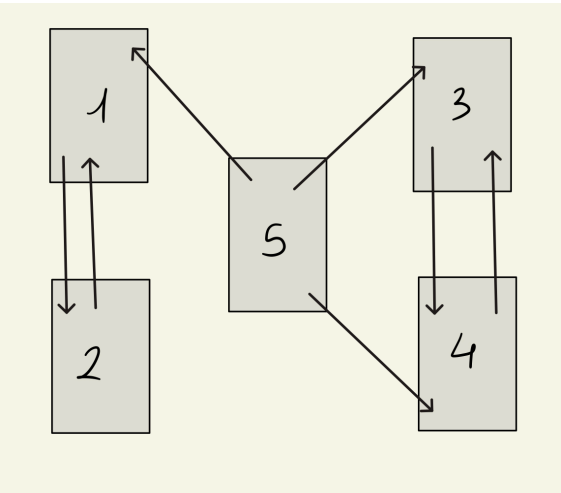
```
The dimension of the eigenspace associated with the eigenvalue 1 is: 4.
```

$\dim(V_1(\mathbf{A}))$ = 4 >= of the number of the components in the web graph(4).

Exercise 3

Problem statement: Add a link from page 5 to page 1 in the web of Figure 2. The resulting web, considered as an undirected graph, is connected. What is the $\dim(V_1(\mathbf{A}))$?

In [71]:
```
display(Image(filename="Graphs/exercise3_graph.png", width=350))
```



In [72]:
```
filename="Graphs/exercise3_graph.dat"
A, labels = read_dat(filename)
eigenvalues, eigenvectors = get_eigenpairs(A)
dimension = np.sum(np.isclose(eigenvalues, 1))
print(f"The dimension of the eigenspace associated", end=" ")
print(f"with the eigenvalue 1 is: {dimension}.")
```

```
The dimension of the eigenspace associated with the eigenvalue 1 is: 2.
```

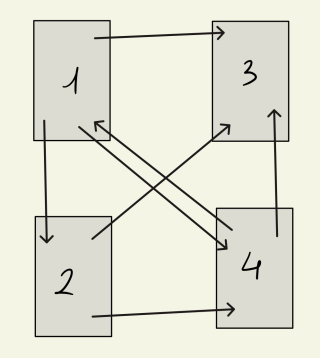
The dimension of the eigenspace associated with the eigenvalue 1 is: 2 because the web contains two closed strongly connected components, {1,2} and {3,4}. Indeed from the node group {1,2} we can't reach the node group {3,4} and from the node group {3,4} we

can't reach the node group {1,2}. Node 5 is not considered since, without the implementation of the Google Matrix, its importance score is null.

Exercise 4: Perron Eigenvector

Problem statement: In the web of Figure 2.1, remove the link from page 3 to page 1. In the resulting web page 3 is now a dangling node. Set up the corresponding substochastic matrix and find its largest positive (Perron) eigenvalue. Find a non-negative Perron eigenvector for this eigenvalue, and scale the vector so that components sum to one. Does the resulting ranking seem reasonable?

In [73]:
```
display(Image(filename="Graphs/exercise4_graph.png", width=200))
```



In [74]:
```
filename = "Graphs/graph1_modified.dat"
A, labels = read_dat(filename)
vals, vecs = get_eigenpairs(A)
# Sort by magnitude
idx = np.argsort(np.abs(vals))[::-1]
perron_val = np.real(vals[idx[0]])
perron_vec = np.real(vecs[:, idx[0]])

# Ensure non-negative and normalized
if np.sum(perron_vec) < 0: perron_vec = -perron_vec
perron_vec = perron_vec / np.sum(perron_vec)

print(f"Perron Eigenvalue: {perron_val:.6f}")
print("Perron Eigenvector (Ranking):")
sorted_idx = np.argsort(perron_vec)[::-1]
for i, idx in enumerate(sorted_idx, 1):
    print(f"  {i}. {labels[idx+1]}: {perron_vec[idx]:.4f}")
```

```
Perron Eigenvalue: 0.561353
Perron Eigenvector (Ranking):
  1. Node3: 0.4386
  2. Node4: 0.2320
  3. Node1: 0.2066
  4. Node2: 0.1227
```

Does the resulting ranking seem reasonable?

Mathematically: The result is correct. Page 3 acts as a "sink" (or dangling node): it receives links but does not provide any. Consequently, it accumulates all the "vote" entering the system and does not redistribute it, making it appear as the most important page.

Practically (for a search engine): No, it is not reasonable. It makes no sense for a page to be considered the "best" on the web (almost twice as important as Page 4) simply because it contains no outgoing links. This behavior rewards "dead ends" and discourages the creation of an interconnected web.

Exercise 5: No Backlinks nodes

Problem statement: Prove that in any web the importance score of a page with no backlinks is zero.

Proof:

The importance score is defined as:

$$x_k = \sum_{j \to k} \frac{x_j}{d_j}$$

where $j \to k$ represents the nodes pointing to k, and d_j is the number of outbound links from j. If a node k has no backlinks (the set of nodes pointing to it is empty), the sum is over an empty set

Exercise 6: Permutations and Invariance

Problem statement:

Implicit in our analysis up to this point is the assertion that the manner in which the pages of a web W are indexed has no effect on the importance score assigned to any given page.

Prove this, as follows: Let W contains n pages, each page assigned an index 1 through n, and let A be the resulting link matrix. Suppose we then transpose the indices of pages i and j (so page i is now page j and vice-versa). Let $\tilde{A}$ be the link matrix for the relabelled web.

- Argue that $\tilde{A} = PAP^T$, where P is the elementary matrix obtained by transposing rows i and j of the $n \times n$ identity matrix. Note that the operation $A \to PA$ has the effect of swapping rows i and j of A, while $A \to AP$ swaps columns i and j. Also, $P^2 = I$, the identity matrix.
- Suppose that x is an eigenvector for A, so $Ax = \lambda x$ for some λ. Show that $y = Px$ is an eigenvector for $\tilde{A}$ with eigenvalue λ.
- Explain why this shows that transposing the indices of any two pages leaves the importance scores unchanged, and use this result to argue that any permutation of the page indices leaves the importance scores unchanged.

In [75]:
```
def swap_node_indices(A, i, j):
    idx_i = i - 1
    idx_j = j - 1
    A_swapped = A.copy()
    A_swapped[[idx_i, idx_j], :] = A_swapped[[idx_j, idx_i], :]
    A_swapped[:, [idx_i, idx_j]] = A_swapped[:, [idx_j, idx_i]]

    return A_swapped

def exercise_6(filename, i, j):
    A, _ = read_dat(filename)
    x=power_iteration_with_A(A)

    n = A.shape[0]
    P = np.eye(n)
    P[i-1, i-1] = 0; P[j-1, j-1] = 0
    P[i-1, j-1] = 1; P[j-1, i-1] = 1

    # P is simmetric, so P^T = P
    A2_theoretical = P @ A @ P

    A2 = swap_node_indices(A, i, j)

    print(f"1) Matrix match: {np.allclose(A2, A2_theoretical)}")

    y_found = power_iteration_with_A(A2)
    y_theoretical = P @ x

    y_found = y_found / np.linalg.norm(y_found)
    y_theoretical = y_theoretical / np.linalg.norm(y_theoretical)

    is_proven = (
        np.allclose(y_found, y_theoretical) or
        np.allclose(y_found, -y_theoretical)
    )
    print(f"2) Relationship y = Px verified: {is_proven}")

exercise_6("Graphs/Graph1.dat", 2, 3)
```

```
 Converged in 23 iterations
1) Matrix match: True
 Converged in 23 iterations
2) Relationship y = Px verified: True
```

Mathematically: We propose that the new eigenvector for the relabeled web is $y = Px$. We must show that y satisfies the eigenvector equation for $\tilde{A}$ with the same eigenvalue λ.

Start with the product $\tilde{A}v$:

$$\tilde{A}y = (PAP)(Px)$$

Using the associative property of matrix multiplication:

$$\tilde{A}y = PA(P \cdot P)x$$

Since $P^2 = I$:

$$\tilde{A}y = PA(I)x = PAx$$

From the original definition, we know that $Ax = \lambda x$. Substituting this into the equation:

$$\tilde{A}y = P(\lambda x)$$

Since λ is a scalar, we can factor it out:

$$\tilde{A}y = \lambda(Px)$$

Recalling that we defined $v = Px$:

$$\tilde{A}y = \lambda y$$

Conclusion

We have proven that $y = Px$ is an eigenvector of the permuted matrix $\tilde{A}$ with eigenvalue λ. The relation $y = Px$ implies that the components of the new vector y are identical to those of x, except that the i-th and j-th components have swapped positions. Therefore, the numerical importance score associated with a specific page remains unchanged regardless of the indexing scheme used.

Exercise 7: Stochastic Matrix

Problem statement:

Prove that if A is an $n \times n$ column-stochastic matrix and $0 \leq m \leq 1$, then $M = (1-m)A + mS$ is also a column-stochastic matrix.

Proof:

1. A is column-stochastic: $\sum_i A_{ij} = 1$ for all j.
2. S is column-stochastic: $S_{ij} = 1/n$, so $\sum_i S_{ij} = 1$.

Sum of the j-th column of M:

$$\begin{aligned}\sum_i M_{ij} &= \sum_i [(1-m)A_{ij} + mS_{ij}] \\ &= (1-m)\sum_i A_{ij} + m\sum_i S_{ij} \\ &= (1-m)(1) + m(1) = 1\end{aligned}$$

Conclusion: Since the sum of every column of M is 1, M is column-stochastic.

Exercise 8

Problem statement:

Show that the product of two column-stochastic matrices is also column-stochastic

Proof: Let $\mathbf{C} = \mathbf{AB}$. The element $C_{ij} = \sum_k A_{ik}B_{kj}$. The sum of the j-th column of C is:

$$\sum_i C_{ij} = \sum_i \sum_k A_{ik}B_{kj} = \sum_k B_{kj} \left(\sum_i A_{ik} \right)$$

Since A is stochastic, $\sum_i A_{ik} = 1$.

$$= \sum_k B_{kj} \cdot 1 = 1 \quad (\text{since } \mathbf{B} \text{ is stochastic})$$

Conclusion: $\mathbf{C}$ is column-stochastic.

Exercise 9

Problem statement:

Show that a page with no backlinks is given importance score m/n by formula (3.2).

Proof:

The PageRank importance score is defined by the Google matrix equation (3.2):

$$\mathbf{x} = (1 - m)\mathbf{A}\mathbf{x} + m\mathbf{s}$$

Where:

- $\mathbf{x}$ is the importance score vector.
- m is the teleportation parameter ($0 \leq m \leq 1$).
- $\mathbf{A}$ is the link matrix.
- $\mathbf{s}$ is the vector with all components equal to $1/n$.

For a node i with no backlinks, the i-th row of the link matrix $\mathbf{A}$ is zero (as there are no incoming links to page i).

Consequently, the i-th component of the product $(1 - m)\mathbf{A}\mathbf{x}$ is:

$$(1 - m)(\mathbf{A}\mathbf{x})_i = 0$$

The only remaining term in the equation for the i-th component is the "teleportation" term:

$$x_i = m \cdot s_i$$

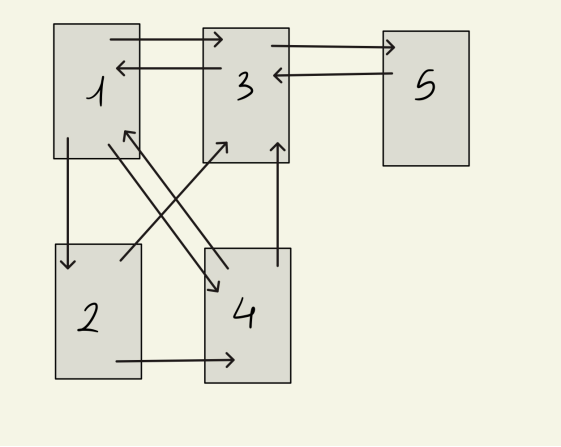
Since $\mathbf{s}$ is defined as a vector where all values are equal to $1/n$, we obtain:

$$x_i = m \cdot \frac{1}{n} = \frac{m}{n}$$

Exercise 11

Problem statement: Consider again the web in Figure 2.1, with the addition of a page 5 that links to page 3, where page 3 also links to page 5. Calculate the new ranking by finding the eigenvector of M (corresponding to λ = 1) that has positive components summing to one. Use m = 0.15

In [76]:
```
display(Image(filename="Graphs/Graph1_modified.png", width=350))
```



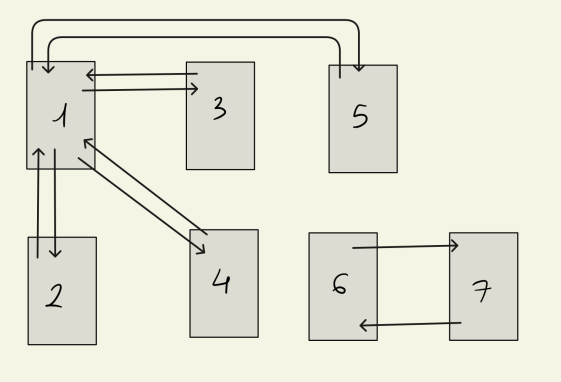
In [77]:
```
analyze_graph("Graphs/exercise11_graph.dat", m=0.15)
```

```
Analyzing Graphs/exercise11_graph.dat ...
  - No dangling nodes detected.
  Converged in 29 iterations
  PageRank scores (Top results):
  ---------------------------------------
  1. Node3               : 0.348894
  2. Node1               : 0.237141
  3. Node5               : 0.178280
  4. Node4               : 0.138495
  5. Node2               : 0.097190

============================================================
```

Exercise 13

Problem statement: Construct a web consisting of two or more subwebs and determine the ranking given by formula (3.1)

In [78]:
```
display(Image(filename="Graphs/exercise13_graph.png", width=350))
```



In [79]:
```
analyze_graph("Graphs/exercise13_graph.dat", m=0.15)
```

```
Analyzing Graphs/exercise13_graph.dat ...
  - No dangling nodes detected.
  Converged in 85 iterations
  PageRank scores (Top results):
  ---------------------------------------
  1. Node1               : 0.339768
  2. Node7               : 0.142857
  3. Node6               : 0.142857
  4. Node5               : 0.093629
  5. Node4               : 0.093629
  6. Node3               : 0.093629
  7. Node2               : 0.093629

============================================================
```

Comment:

The analysis using matrix M shows that the isolated pair (Nodes 6-7) outranks the peripheral nodes of the larger cluster (Nodes 2-5). This demonstrates that out-degree dilution significantly weakens the authority transferred by Node 1, the central hub.

Exercise 14: Convergence speed

Problem statement: For the web in Exercise 11, compute the values of

$$||M^k x_0 - q||_1$$

and

$$\frac{||M^k x_0 - q||_1}{||M^{k-1} x_0 - q||_1}$$

for $k = 1, 5, 10, 50$, using an initial guess x_0 not too close to the actual eigenvector q (so that you can watch the convergence).

Determine

$$c = \max_{1 \le j \le n} \left| 1 - 2 \min_{1 \le i \le n} M_{ij} \right|$$

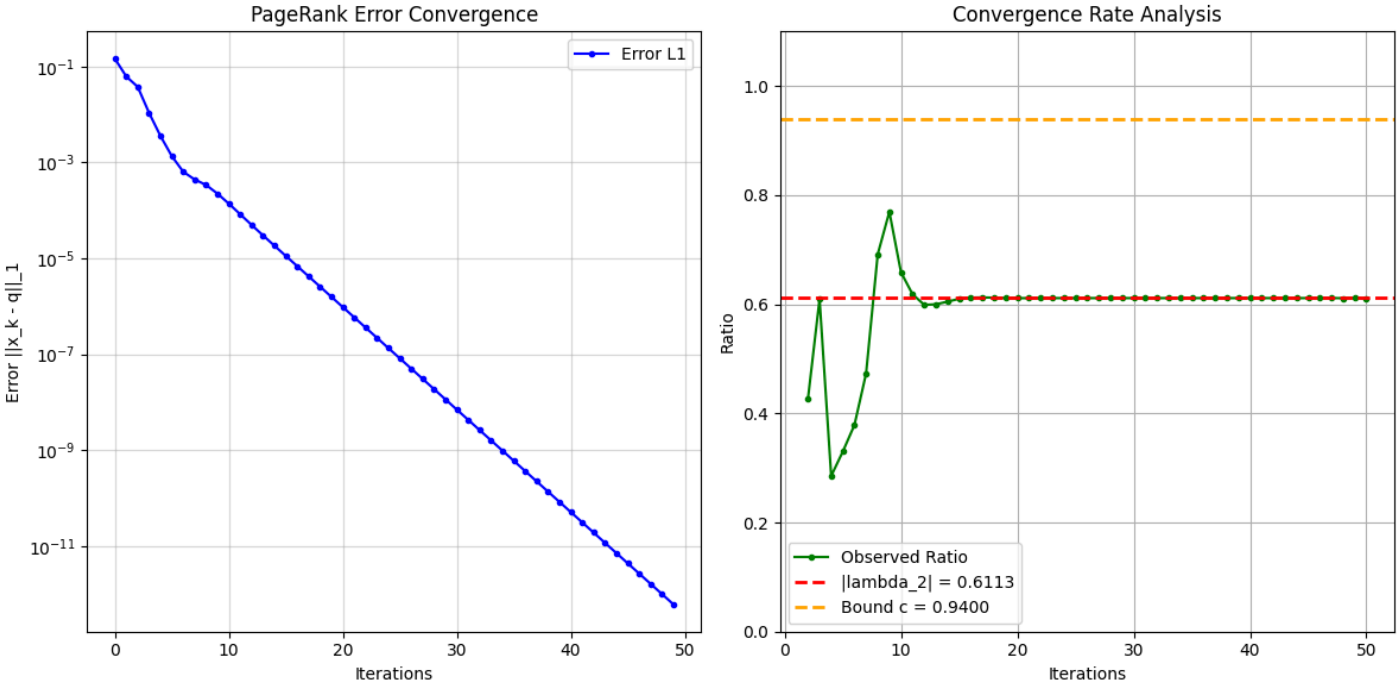
and the absolute value of the second largest eigenvalue of M.

Note: Computing eigenvalues explicitly via np.linalg.eig implies constructing the dense matrix M. This is feasible only for small datasets like the ones in these exercises, not for web-scale graphs

In [80]:
```
A,_ = read_dat("Graphs/exercise11_graph.dat")
n=A.shape[0]
m = 0.15
S = np.ones((n, n)) / n
M = (1 - m) * A + m * S

vals, vecs = get_eigenpairs(M)
idx = np.argsort(np.abs(vals))[::-1]
vals = vals[idx]
vecs = vecs[:, idx]

# q is the dominant eigenvector
q = np.real(vecs[:, 0])
if np.sum(q) < 0: q = -q
q = q / np.sum(q)

lambda_2 = np.abs(vals[1])
min_element = np.min(M)
c_bound = 1 - 2 * min_element

print(f"Theoretical Parameters:")
print(f" -> Second Eigenvalue |lambda_2|: {lambda_2:.6f}")
print(f" -> Theoretical Bound c         : {c_bound:.6f}")
print("-" * 65)

np.random.seed(42)
x_k = np.random.rand(n)
x_k = x_k / np.sum(x_k)
s_vec = np.ones(n) / n

errors = []
ratios = []
prev_error = None

print(f"{'k':<5} | {'Error ||x_k - q||_1':<22}   ", end="")
print(f"| {'Ratio (err_k/err_k-1)':<22}")
print("-" * 65)

for k in range(1, 51):
    x_new = (1 - m) * (A @ x_k) + m * s_vec
    x_new = x_new / np.sum(x_new)

    error = np.linalg.norm(x_new - q, 1)
    errors.append(error)

    if prev_error is not None and prev_error > 1e-15:
        ratio = error / prev_error
        ratios.append(ratio)
    else:
        ratio = 0.0
        ratios.append(0.0)

    if k in [1, 5, 10, 50]:
        print(f"{k:<5} | {error:.6e}             | {ratio:.6f}")

    x_k = x_new
    prev_error = error

# 5. Visualizzazione Grafica
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.semilogy(errors, 'b.-', label='Error L1')
plt.title('PageRank Error Convergence')
plt.xlabel('Iterations')
plt.ylabel('Error ||x_k - q||_1')
plt.grid(True, which="both", ls="-", alpha=0.5)
```

```
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(range(2, 51), ratios[1:], 'g.-', label='Observed Ratio')
plt.axhline(y=lambda_2, color='r', linestyle='--',
            linewidth=2, label=f'|lambda_2| = {lambda_2:.4f}')
plt.axhline(y=c_bound, color='orange', linestyle='--',
            linewidth=2, label=f'Bound c = {c_bound:.4f}')

plt.title('Convergence Rate Analysis')
plt.xlabel('Iterations')
plt.ylabel('Ratio')
plt.ylim(0.0, 1.1)
plt.legend()
plt.grid(True)

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

```
Theoretical Parameters:
 -> Second Eigenvalue |lambda_2|: 0.611269
 -> Theoretical Bound c         : 0.940000
------------------------------------------------------------------
k     | Error ||x_k - q||_1      | Ratio (err_k/err_k-1)
------------------------------------------------------------------
1     | 1.462376e-01             | 0.000000
5     | 3.573691e-03             | 0.329961
10    | 2.244275e-04             | 0.658540
50    | 6.087214e-13             | 0.610727
```



The results confirm that the PageRank algorithm converges much faster than the pessimistic theoretical bound suggested in the paper, effectively stabilizing at a rate determined by the second largest eigenvalue is equal to 0.61, which is well below the upper limit of c =0.94.

Exercise 16: Non-Diagonalizability

Problem statement: Consider the link matrix

$$A = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} \\ 1 & \frac{1}{2} & 0 \end{bmatrix}.$$

Show that

$$M = (1-m)A + mS \qquad (\text{all } S_{ij} = \tfrac{1}{3})$$

is not diagonalizable for $0 \leq m < 1$.

In [81]:
```
A = np.array([[0, 0.5, 0.5], [0, 0, 0.5], [1, 0.5, 0]])
eigenvalues, eigenvectors = get_eigenpairs(A)
print(eigenvalues)
```

```
[-0.5+1.06051605e-08j -0.5-1.06051605e-08j  1. +0.00000000e+00j]
```

We analyze the specific case of the provided matrix.

1. **Eigenvalues of A:** $\lambda_A \in \{1, -1/2, -1/2\}$.
2. **Eigenvalues of M:** $\lambda_1 = 1$ and $\lambda^* = -(1-m)/2$ (with algebraic multiplicity 2).
3. **Geometric Multiplicity:** For M to be diagonalizable, the geometric multiplicity (m.g.) of λ^* must be 2. Calculating $rank(\mathbf{M} - \lambda^*\mathbf{I})$, we find the rank is 2, thus m.g. is $3 - 2 = 1$. Since $m.g.(1) < m.a.(2)$, the matrix is **NOT diagonalizable**.

Exercise 17: Choice of m=0.15

Problem statement: How should the value of m be chosen? How does this choice affect the rankings and the computation time?

The choice of $m = 0.15$ is a trade-off:

1. **Speed:** $|\lambda_2| \leq 1 - m$. If m is large, convergence is very fast. If $m \to 0$, convergence slows down.
2. **Accuracy:** If m is too large, the matrix becomes pure noise ($\mathbf{S}$) and the link structure is lost. $m = 0.15$ ensures fast convergence (factor approx 0.85) while maintaining the relevance of the hypertextual structure.