Analysis of Biological Networks

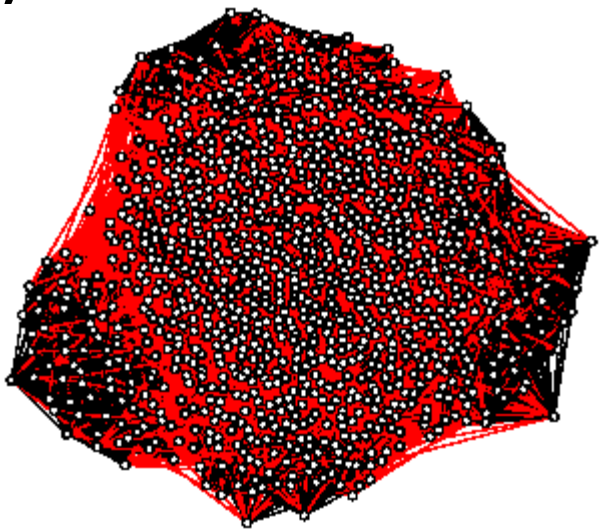
1. Clustering
2. Random Walks

Problem 1: Graph Clustering

- Finding dense subgraphs
- Applications
  - Identification of novel pathways, complexes, other modules?
- Example algorithm: MCODE

The Problem

- Given a protein interaction network find strongly connected components (clusters) with the network that may correspond to biological functional modules (complexes or pathways)



Some Algorithms

- MCL
  - Markov CLustering
- RNSC
  - Restricted Neighborhood Search Clustering
- SPC
  - Super Paramagnetic Clustering
- MCODE
  - Molecular COmplex DEtection

Markov Cluster Algorithm

- Simulates a flow on the graph.
- Calculates successive powers of the adjacency matrix
- Parameters
  - One parameter: *inflation parameter*
- The process partitions the graph (i.e., no overlapping clusters)
- The inflation parameter influence the number of clusters generated

Restricted Neighborhood Search Clustering

- Starts with an initial random clustering
- Tries to minimize a cost function by iteratively moving vertices between neighboring clusters.
- Parameters:
  - Number of iterations
  - Diversification frequency
  - …. and 5 other parameters

Super Paramagnetic Clustering

- Hierarchical algorithm inspired from an analogy with the physical properties of a ferromagnetic model subject to fluctuation at nonzero temperature.
- Parameters:
  - Number of nearest neighbors
  - Temperature

MCODE

- Weight each vertex by its local neighborhood density (using a modified version of clustering coefficient)
- Starting from the top weighted vertex, include neighborhood vertices with similar weights to the cluster
- Remove the vertices from the clusters
- Continue with the next highest weight vertex in the network
- May provide overlapping clusters

Vertex weighting

- Clustering coefficient

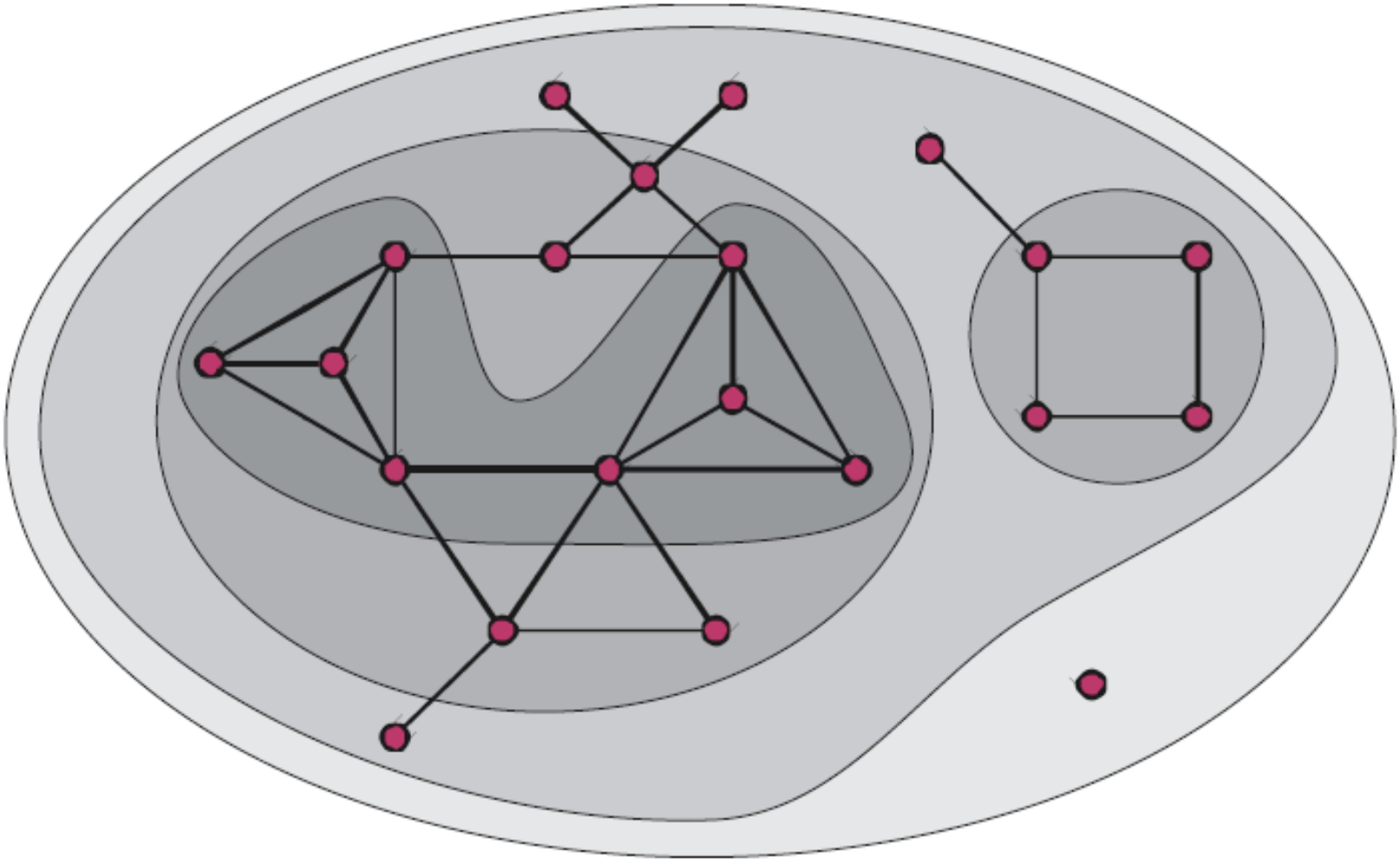
$$CC_i = \frac{2e_i}{d_i(d_i - 1)}$$

where e_i is the number of edges between the neighbors of node i and d_i is the number of neighbors of node i.

k-core

- A part of a graph where every node is connected to other nodes with at least k edges (k=0,1,2,3...)
- Finding a k-core in a graph proceeds by progressively removing vertices of degree < k until all remaining vertices are connected to each other by degree k or more. Complexity: $O(n^2)$. The highest k-core is found by trying to find k-cores from one up until the highest degree in the neighborhood graph. Overall complexity: $O(n^3)$

k-core example



Core-clustering Coefficient

- Product of the clustering coefficient of the highest k-core in the neighborhood of a vertex and *k*.

Problem 2: Finding relationships

- Random Walks on Graphs
  - Finding important nodes (Google's PageRank)
  - Function prediction
  - Adding new members to known pathways, complexes
  - Finding relationships of genes/diseases in gene-disease networks

Google's PageRank

- Assumption: A **link** from page A to page B is a **recommendation** of page B by the author of A (we say B is *successor* of A)

➔Quality of a page is related to its in-degree

- Recursion: Quality of a page is related to
  - its in-degree, and to
  - the *quality* of pages linking to it

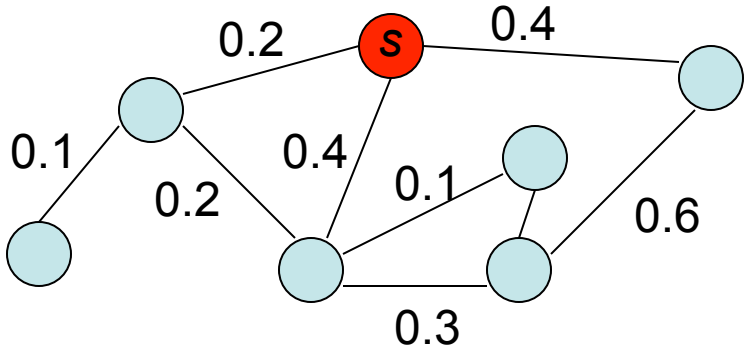
➔**PageRank** [BP '98]

Definition of PageRank

- Consider the following infinite random walk (surf):
  - Initially the surfer is at a random page
  - At each step, the surfer proceeds
    - to a randomly chosen web page with probability d
    - to a randomly chosen successor of the current page with probability 1-d
- **The PageRank of a page p is *the fraction of steps the surfer spends at p in the limit.***

Random walks **with restarts** on interaction networks

- Consider a random walker that starts on a source node, s. At every time tick, the walker chooses randomly among the available edges (based on edge weights), or goes back to node s with probability c.



Random walks on graphs

- The probability $p_s(v)^{(t)}$, is defined as the probability of finding the random walker at node v at time t.

- The steady state probability $p_s(v)$ gives a measure of affinity to node s, and can be computed efficiently using iterative matrix operations.

Computing the steady state $\mathbf{p}$ vector

- Let $\mathbf{s}$ be the vector that represents the source nodes (i.e., $\mathbf{s}_i = 1/n$ if node i is one the n source nodes, and 0 otherwise).

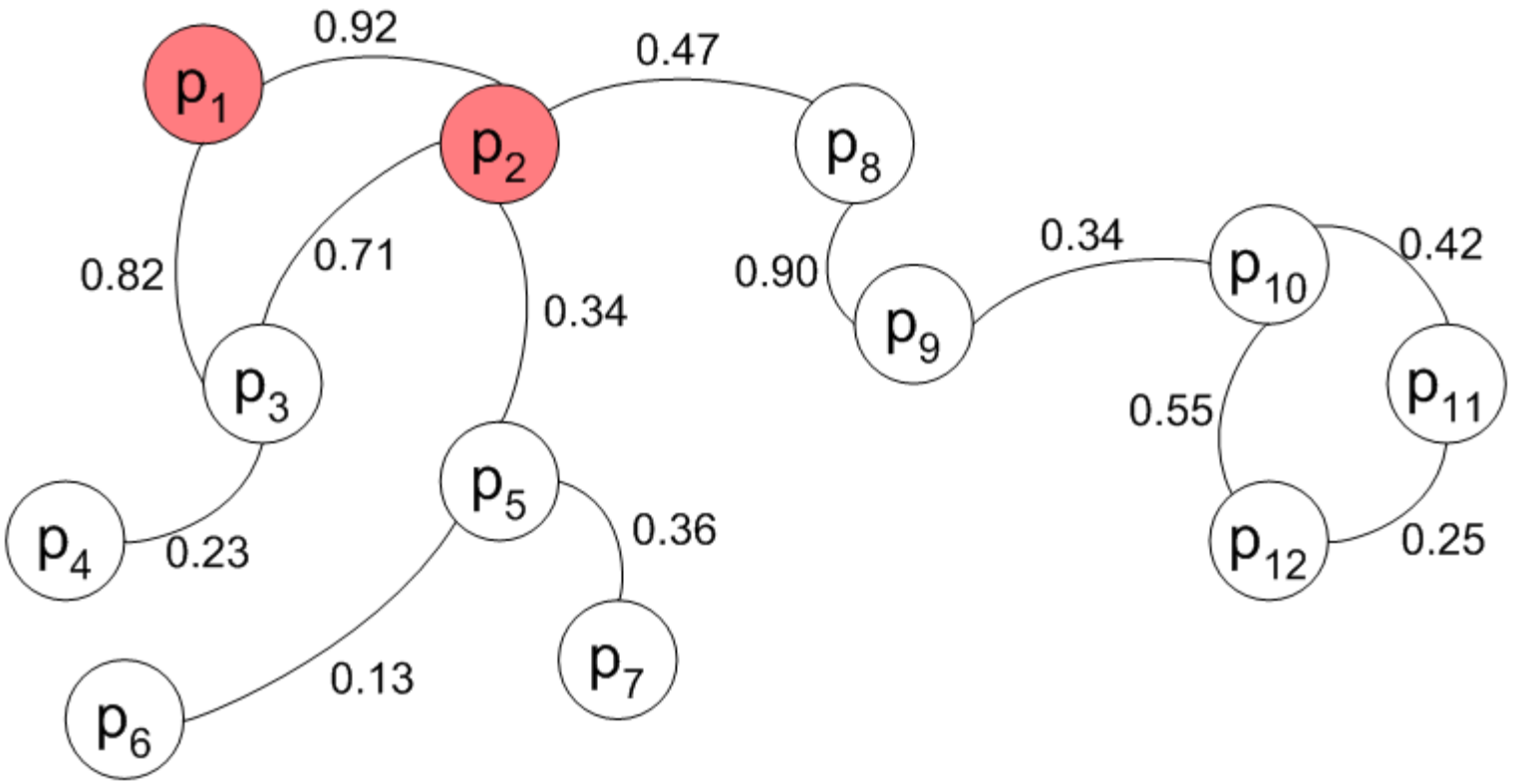
- Compute the following until $\mathbf{p}$ converges:

$$\mathbf{p} = (1\text{-}c)\mathbf{A}^{\mathbf{T}}\mathbf{p} + c\mathbf{s}$$

where $\mathbf{A}$ **is the row normalized adjacency matrix** and c is the restart probability.

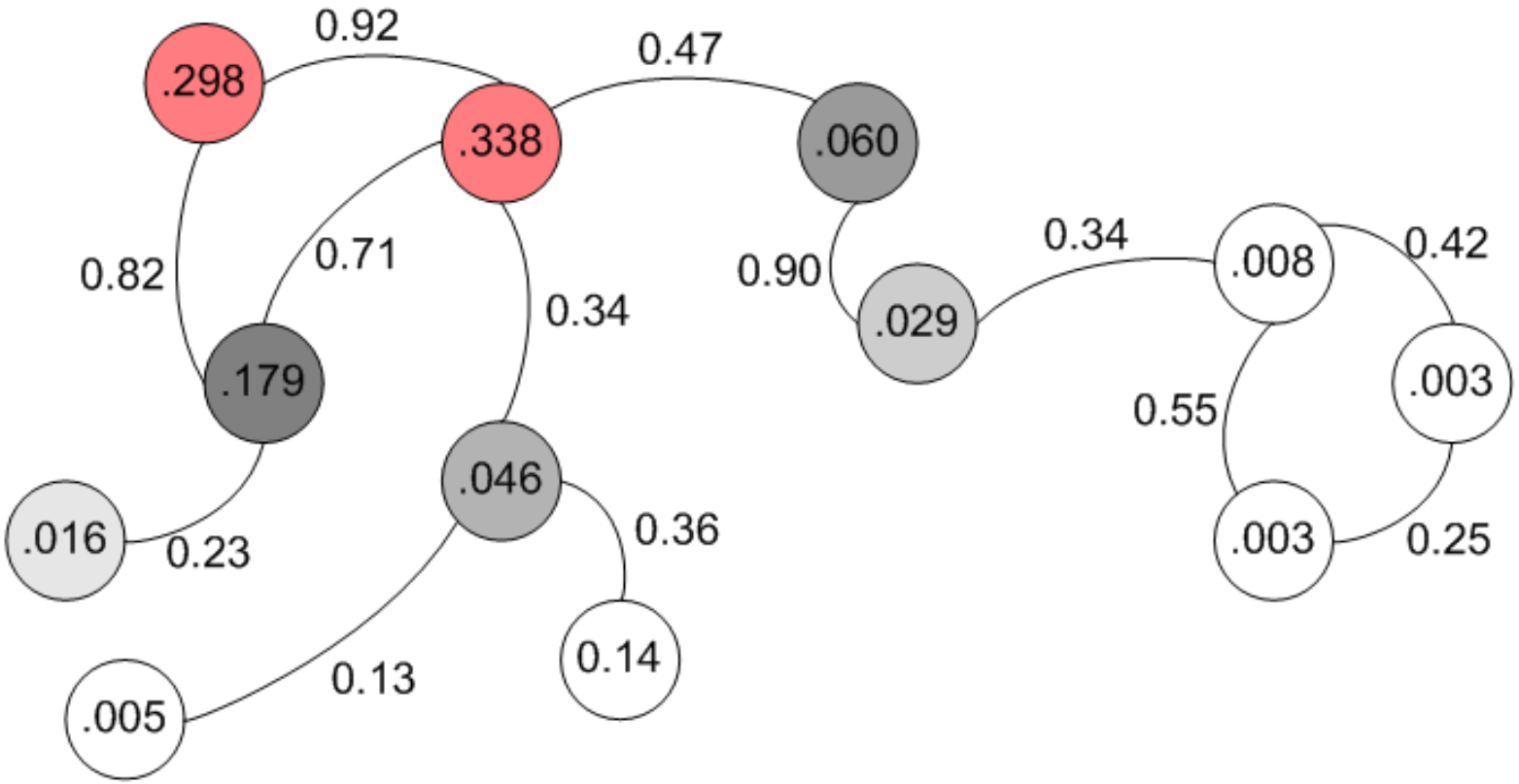
Same example

- Start nodes: p_1 and p_2



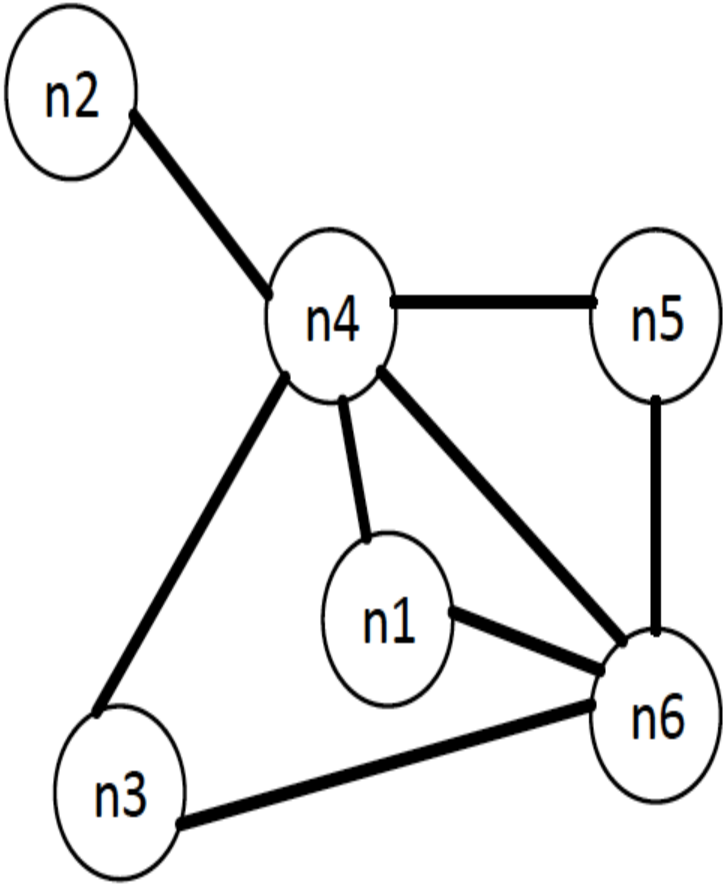
Random walk results

- Restart probability, c = 0.3

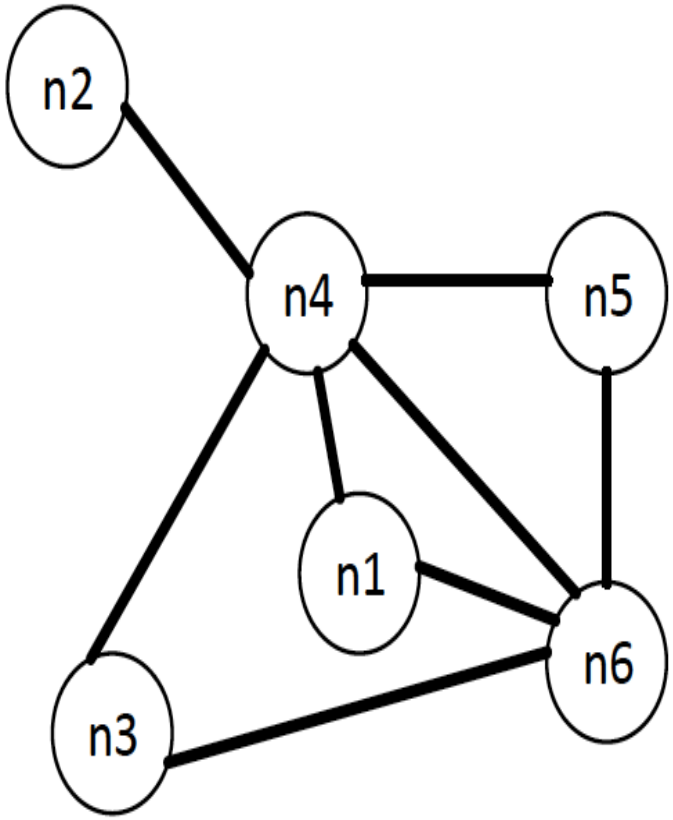


A small example

- Let n_5 and n_6 be the restart nodes



Adjacency matrix and the restart vector



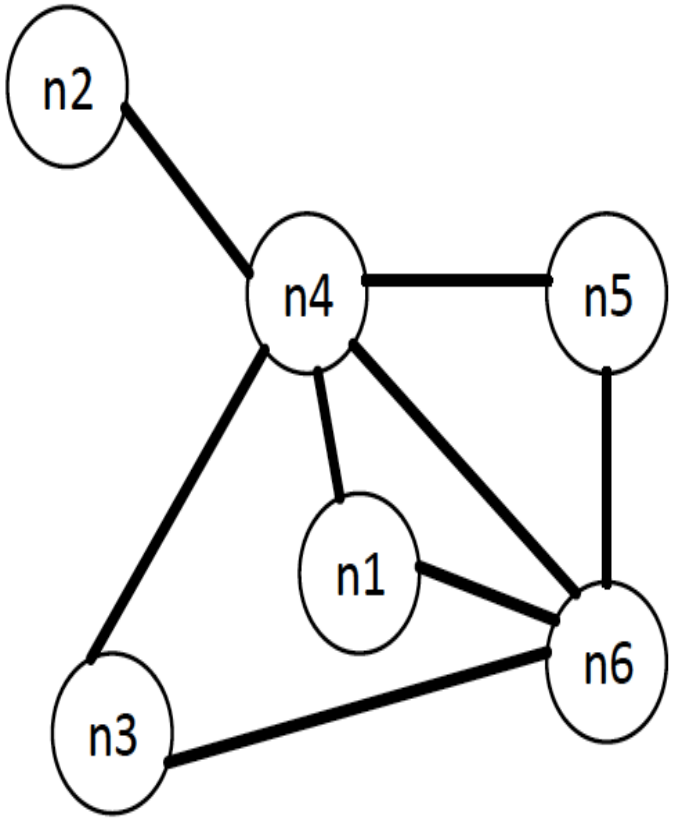
A:

	n1	n2	n3	n4	n5	n6
n1	0	0	0	1	0	1
n2	0	0	0	1	0	0
n3	0	0	0	1	0	1
n4	1	1	1	0	1	1
n5	0	0	0	1	0	1
n6	1	0	1	1	1	0

$s = p_0$

n1	0
n2	0
n3	0
n4	0
n5	0.5
n6	0.5

Normalized adjanceny matrix



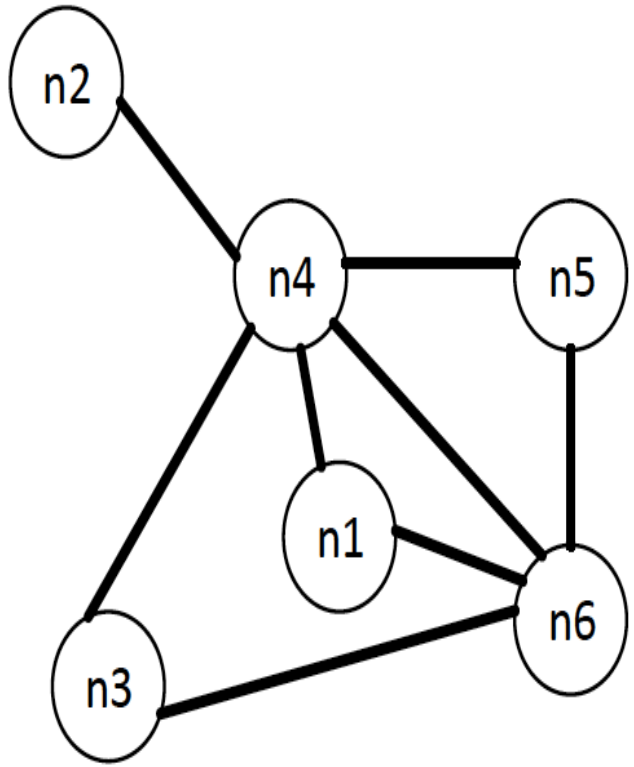
A:

	n1	n2	n3	n4	n5	n6
n1	0	0	0	.5	0	.5
n2	0	0	0	1	0	0
n3	0	0	0	.5	0	.5
n4	.2	.2	.2	0	.2	.2
n5	0	0	0	.5	0	.5
n6	.25	0	.25	.25	.25	0

$s = p_0$

n1	0
n2	0
n3	0
n4	0
n5	0.5
n6	0.5

Computing p_1



Let c = 0.3

$p_1 = 0.7$

A^T:

	n1	n2	n3	n4	n5	n6
n1	0	0	0	.2	0	.25
n2	0	0	0	.2	0	0
n3	0	0	0	.2	0	.25
n4	.5	1	.5	0	.5	.25
n5	0	0	0	.2	0	.25
n6	.5	0	.5	.2	.5	0

x

p_0:

n1	0
n2	0
n3	0
n4	0
n5	0.5
n6	0.5

+ 0.3

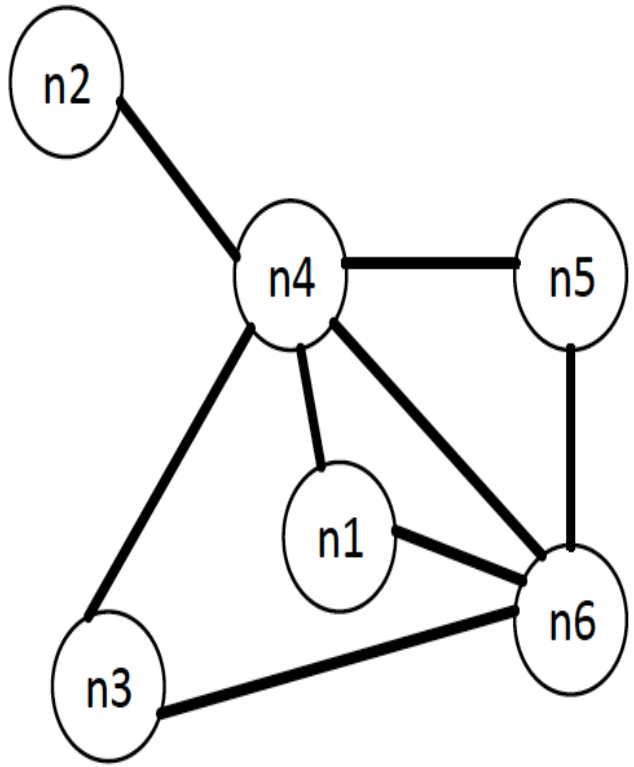
s:

n1	0
n2	0
n3	0
n4	0
n5	0.5
n6	0.5

p_1 =

n1	0.087
n2	0.0
n3	0.087
n4	0.262
n5	0.238
n6	0.325

Computing p_2



$p_2 = 0.7$

A^T:

	n1	n2	n3	n4	n5	n6
n1	0	0	0	.2	0	.25
n2	0	0	0	.2	0	0
n3	0	0	0	.2	0	.25
n4	.5	1	.5	0	.5	.25
n5	0	0	0	.2	0	.25
n6	.5	0	.5	.2	.5	0

x p_1:

n1	0.087
n2	0.0
n3	0.087
n4	0.262
n5	0.238
n6	0.325

+ 0.3 s

n1	0
n2	0
n3	0
n4	0
n5	0.5
n6	0.5

p_2 =

n1	0.094
n2	0.037
n3	0.094
n4	0.201
n5	0.244
n6	0.331

$p_{21} = p_{22}$

n1 0.089
n2 0.032
n3 0.089
n4 0.225
n5 0.239
n6 0.327