Analysis of Biological Networks

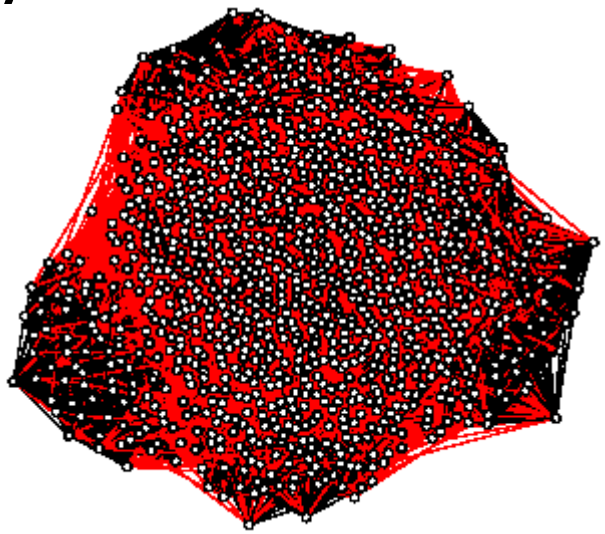
1. Clustering
2. Random Walks
3. Finding paths

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

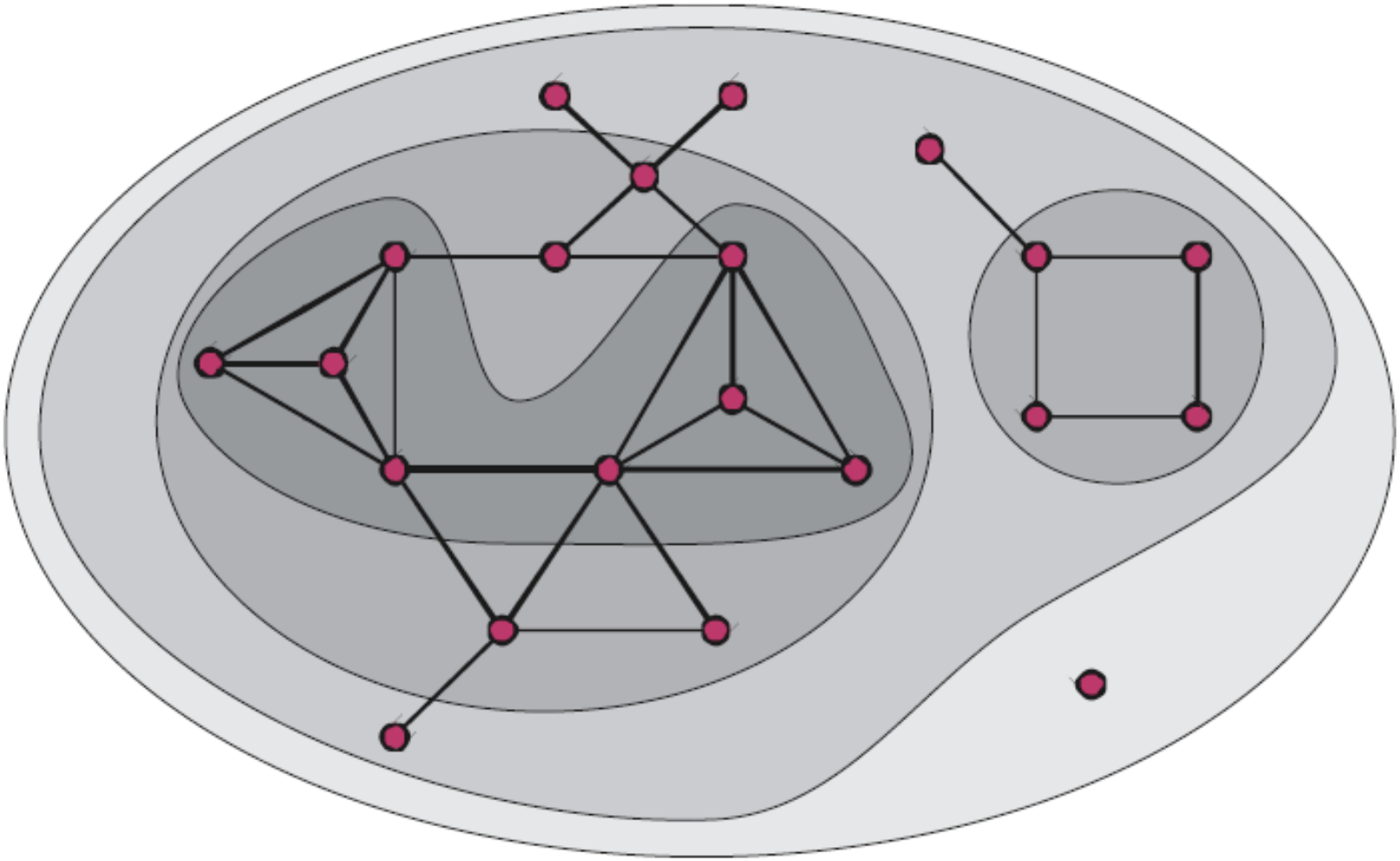
$$CC_i = \frac{2e_i}{d_i(d_i - 1)}$$

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

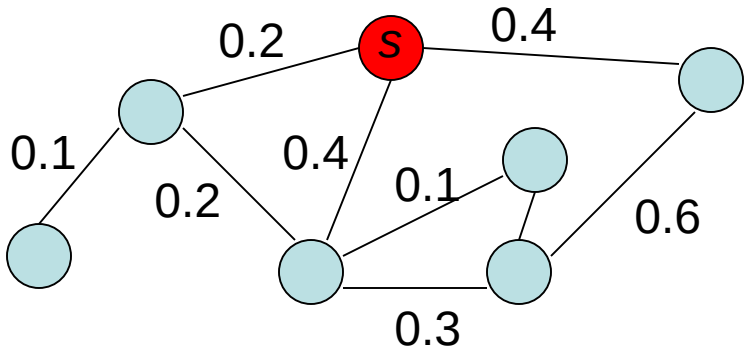
➔**PageRank** [BP '98]

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 $\boldsymbol{c}$.



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).

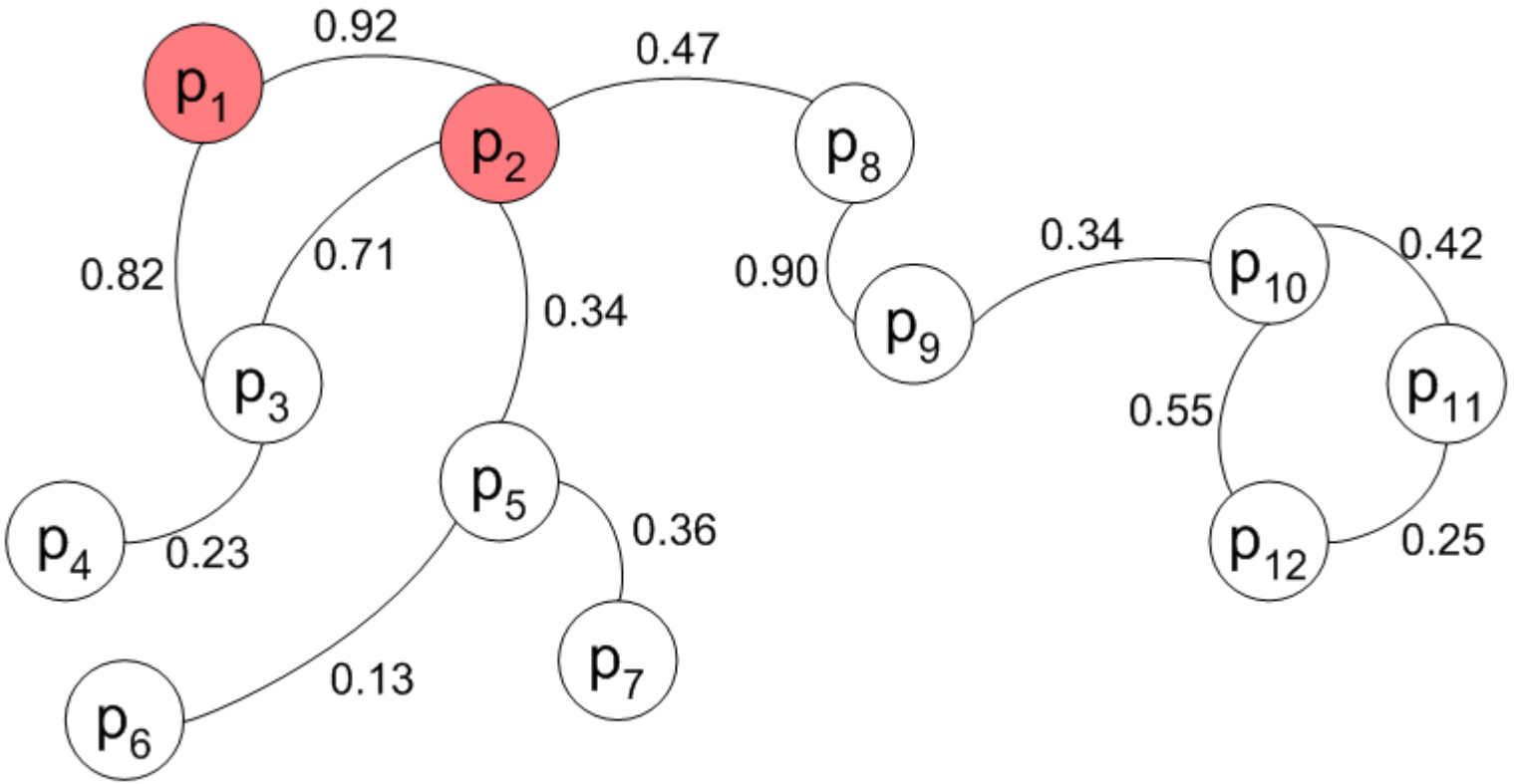
- Compute the following until $\mathbf{p}$ converges:

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

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

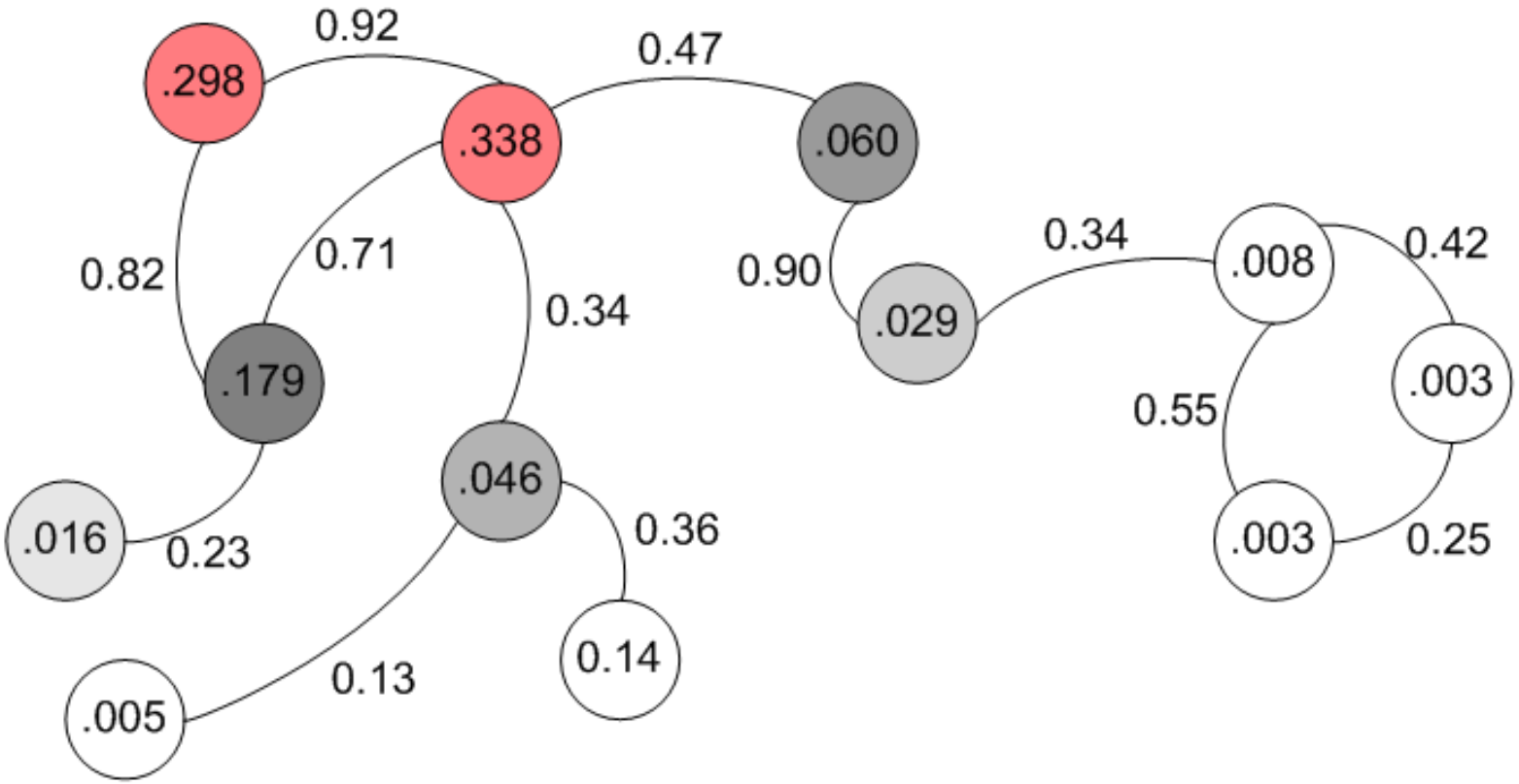
Same example

- Start nodes: p_1 and p_2



Random walk results

- Restart probability, c = 0.3



Problem 3: Finding paths

- Find the best simple path of length *k* starting from a given node in the graph
- Applications
  - The biological network is probabilistic (e.g. predicted network)
  - Signaling pathways of known size

Problem definition

- Given a set I of start vertices, what is the best simple path of length k?
- Simple path:
  - each vertex is visited once, no cycles
- *Best* simple path
  - That is the most probable path
    - i.e., if edge weights show probabilities, the probability of a path can be computed by:

$$\prod_{\text{for every edge } e \in path} w(e_i)$$

Additive edge weights

- For an easier formulation of the problem (similar to shortest paths) it is better to work with additive edge weights rather than multiplicative ones. So convert each edge probability to:
  - new_weight (e) = -log weight(e)
  - probabilities between 0 and 1 → new weights positive values between 0 and Infinity
  - smaller probabilities will have larger weights and higher probabilities will have smaller weights → best path is the *shortest* path

Formal definition

- *Weight* of a path is the sum of the weights of its edges, and the *length* of a path is the number of vertices it contains.
- Given an undirected weighted graph $G=(V,E,w)$ with $|V|=n$, $|E|=m$ and a set I of start vertices, we wish to find, for each vertex v, a minimum-weight simple path of length k that starts with I and ends at v. If no such simple path exists, the algorithm should report this fact.
- Simple-path restriction makes the problem a difficult one.
  - without simple path restriction we can get the a shortest path of desired length by looping at smallest edges back and forth.

Dynamic programming

- The best simple-path of length k problem can be solved by dynamic programming.
- Define $W(v, S)$ as the minimum weight of a simple path of length $|S|$ which starts at some vertex in I, visits each vertex in S, and ends at v. Starting at smaller sets we can use the following recurrence function to fill in a table of $W(v, S)$ for all v and S.

$$W(v, S) = \min_{u \in S - \{v\}} W(u, S - \{v\}) + w(u, v), |S| > 1$$

$$W(v, \{v\}) = 0 \text{ if } v \in I \text{ and } \infty \text{ otherwise}$$

- Complexity: $O(kn^k)$

Color coding

- Idea: Instead of using vertex ids (resulting in n^k possible subsets of length k), let's assign random colors (out of k possible colors) to the vertices.
- Instead of searching for paths with distinct vertices, search for paths with distinct colors (*colorful* paths)
- This reduces the possible sets to look for to (2^k)

Color-Coding

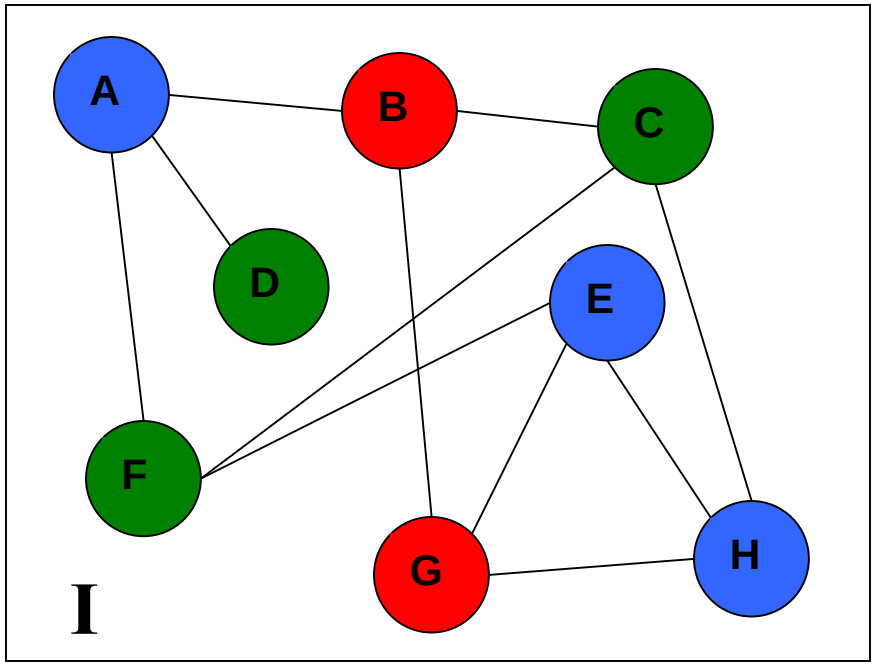
- Colorful paths can be found with dynamic programming
- Key point: a colorful path of length *k* contains a colorful path of length *k*-1.
- Store path information at each node for each subset of *k* colors
  - Only 2^k color subsets, rather than $O(n^k)$ node subsets

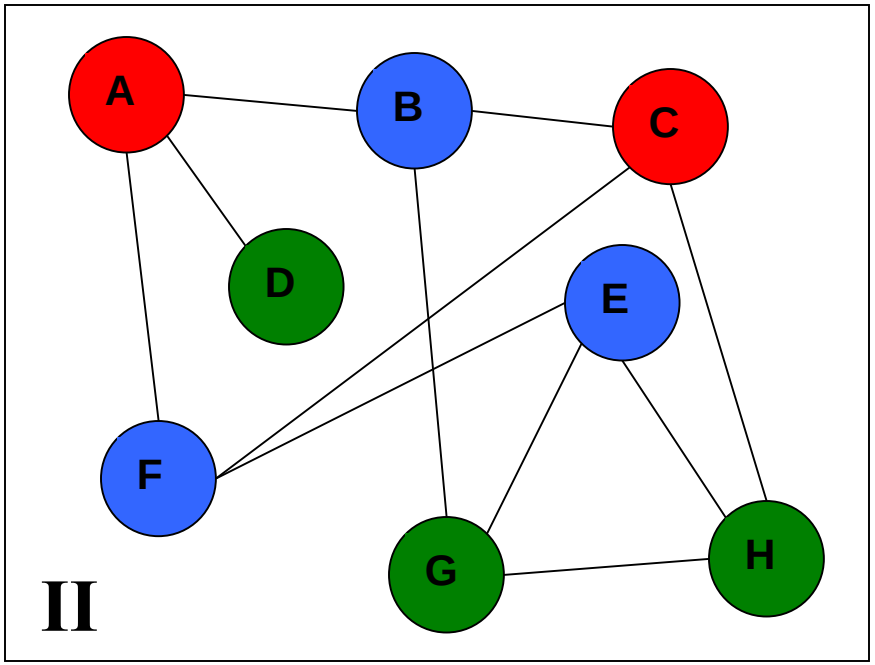
$$W(v, S) = \min_{u:c(u)\in(S-\{c(v)\})} W(u, S - \{c(v)\}) + w(u, v), |S| > 1$$

$$W(v, \{c(v)\}) = 0 \text{ if } v \in I \text{ and } \infty \text{ otherwise}$$

- Runtime is $O(2^k km)$ << $O(kn^k)$ brute force
- Space is $O(2^k n)$ << $O(kn^k)$ brute force

Coloring Example





- Two different colorings on toy graph, *k*=3
- In coloring **I**, W(A,RGB) is built C->BC->ABC
- In coloring **II**, W(A,RGB) is built G->BG->ABG
- ABC is not colorful in coloring **II**