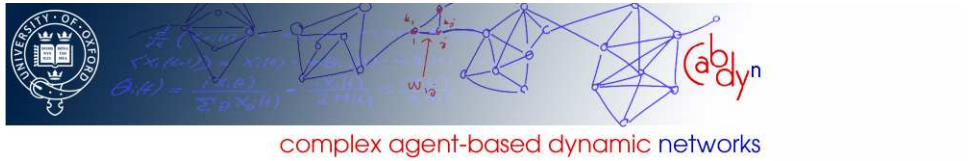
ALGORITHMS IN NETWORKS:

Computational tools for network research

Eduardo López
CABDyN Complexity Centre
University of Oxford

January 2011: University of Warwick



Some Basic Notation and Functions

- Pseudocode:
  - Loops: for (either counter or set element)
  - C or Python like
  - **Approximation of true code (details missing)**
- Auxiliary functions:
  - rand: uniform random number between 0 and 1
  - intrand(a,b): uniform random integer between a and b
  - min/max($\{Q\}$): element of minimum value from set Q
  - min/max($f(\{Q\})$): element of minimum value f from set Q
- Symbols:
  - $\varnothing$: Empty set
  - $\{x\}$: Set element x
  - (i,j): Link between i and j

Something about data structure

- This is a practical choice: Usual trade-off

Higher memory use & Lower CPU time	vs	Lower memory use & Higher CPU time

- Another trade-off

Complex data structure & Better performance	vs	Simple data structure & Worse performance

- Some basic possibilities:
  - Use adjacency matrices (thus use matrix data structure)
  - Use adjacency lists (effectively combination of 1-d arrays)
  - Ex: arrays in C, lists and dictionaries in Python.
  - Always consider network packages (e.g. networkx, igraph) BUT TEST!

Writing your code

- Analyze your problem in detail and decide on algorithms needed

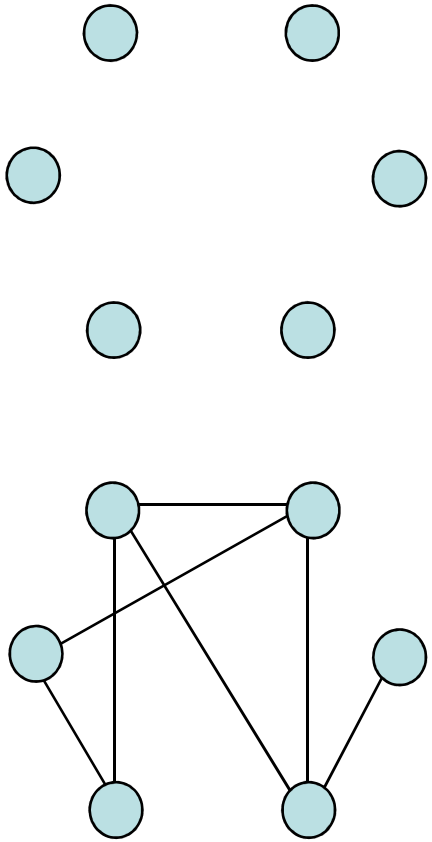
- Consider combination of own + downloaded code.

- **Test your code with every conceivable case within reason!**
  - Look for cases that can be also be solved by hand
  - Check quantities that should be preserved
  - Test for memory and performance
  - Check results against intuition

- Consider making your code modular for future use.

- Document your code, also for future use, or checking of results.

Network construction I
Subroutine ER vers. 1



- Create network data structure
- Scan through all possible links
- Add a link if rand<ϕ

Pseudocode:

ER1(n,ϕ) ->G=(N,Λ):
Input:
- n: Number of nodes
- ϕ: Link density wrt $n(n-1)/2$
Output:
- G: Network
Procedure:
- node set N={$1,\ldots,n$}
- link set $\Lambda=\emptyset$
- for $i=1,n$ {
- for $j=i+1,n$ {
- if (rand<ϕ): Λ<-$\Lambda\cup(i,j)$ }}
- return(G=(N,Λ))

Network construction I
Subroutine ER vers. 2

Use knowledge of ER to optimize:

- For node i, draw number k from: $p_k = e^{-z} z^k / k! \; ; \; z \equiv n\phi$
- For node i draw k random integers $j \neq i$ between 1 and n.
- Each random number is used to create link (i,j).
- After running over all nodes, eliminate repeated links.

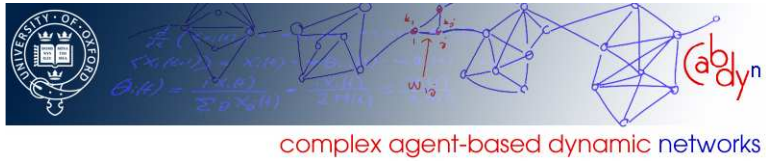
This algorithm works because

$$\text{Prob}[\text{draw } (i,j) \,\&\, (j,i)] \sim \left(\frac{z}{n\text{-}1} \right)^2$$

But number of steps ~ $n\,z$

Pseudocode:

ER2(n,ϕ)->G=(N,Λ):
Input:
- n: Number of nodes
- ϕ: Link density wrt $n(n-1)/2$
Output:
- G: Network
Procedure:
- node set N={$1,\ldots,n$}
- link set $\Lambda=\emptyset$
- $z=(n-1)\,\phi$
- for $i=1,n$ {
- k<-p_k
- for $u_j=1,k$\{
- Λ<-$\Lambda\cup(i,j)$ }}
- return(G=(N,Λ))

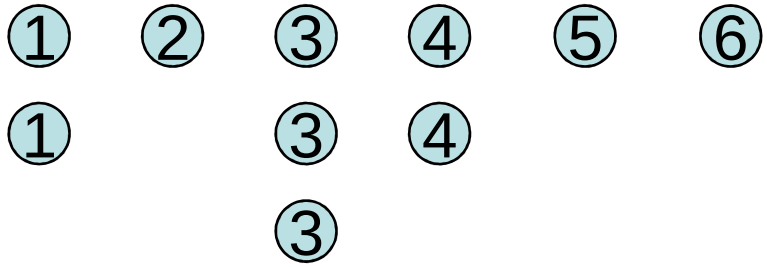


Network Construction: Configuration model

1) Degree sequence:

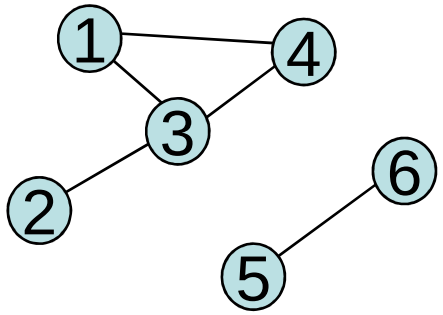
$$\{k_1 = 2, k_2 = 1, k_3 = 3, k_4 = 2, k_5 = 1, k_6 = 1\}$$

2) Create copies according to degree sequence:

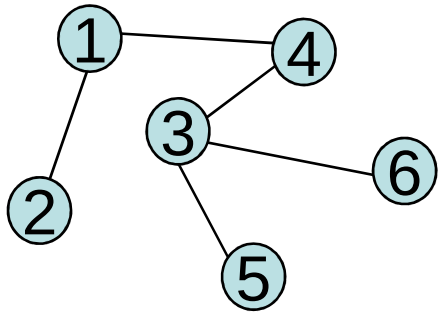


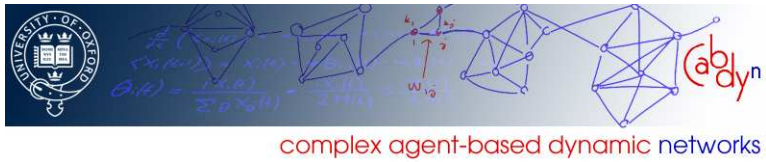
3) Interconnect copies randomly with no self- or repeated connections

Example 1:



Example 2:





- Check degree sequence can create *G*

Network Construction: Configuration model

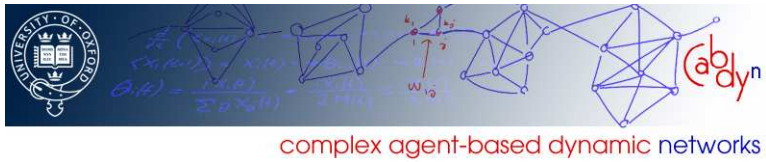
Pseudocode:

CM($\{k_i\}$)->G=(N,Λ):
Input:
- $\{k_i\}_{i=1,\ldots,n}$: Degree sequence
Output:
- G: Network
Procedure:
- $K= \emptyset$
- link set $\Lambda=\emptyset$
- for $i=1,n$ {
- for $j=1,k_i$ {
- K<-$K\cup\{i\}$ }}
- Λ<-Mix(K)
- return(G=(N,Λ))

NOTE: $\{k_i\}$ Must be checked for consistency

Pseudocode:

Mix(K_0)->Λ:
Input:
- K_0: Element List
Output:
- Λ: Link set
Procedure:
- $K = K_0$
- while $K\neq \emptyset$ {
- $K=K_0$
- $\Lambda=\emptyset$
- for x in K {
- y=intrand(K-{x})
- while [(x,y) $\in \Lambda$] OR [$x=y$]{
- K' <- K'(<-K) - {y}
- y=intrand(K')
- if ($K'=\emptyset$): restart Mix}
- $\Lambda \leftarrow \Lambda \cup \{(x,y)\}$
- K<-K-{x,y} }}
- return(Λ)



- Conserves $P(k)$
- Does not guarantee connected network

Randomization of real networks

- Hypothesis:
  - Real networks typically display unique features compared to random

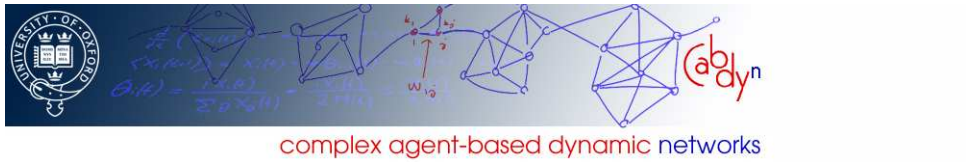
- How to test this statement?
  - Key: “compared to random.” What does this mean?

- Usually, we seek networks displaying atypical features.
  These features are signature of special behavior in network.

- Main difficulty: choose network ensemble with which to compare network of interest.

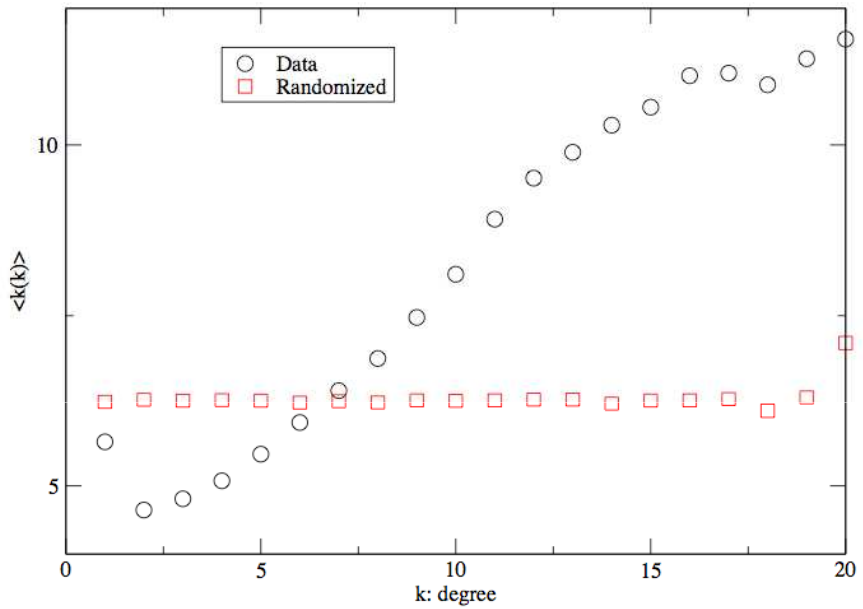
- Some possibilities: From original network “turn off” (or on) characteristics one at a time.

- Many network studies use *P(k)* conserved



Randomization of real networks (cont)

Episims data, large contact users

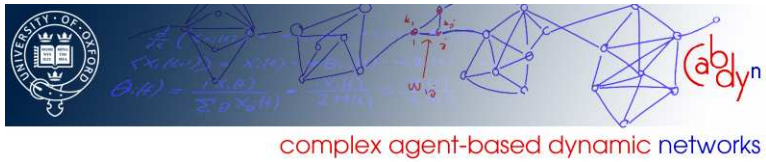


Episims: high resolution agent based simulation of Portland, OR, USA

- Example: Assortativity of social networks- What is my friend's average degree?
- In general, social networks tend to be assortative, technogical and biological disassortative
- Assortativity is measured as:

$$\left\langle k_{neigh}(k) \right\rangle = \sum_{k'} \frac{k' P_{\text{neigh}}(k' \mid k)}{k}$$

- Assortativity reflects how nodes link together, independent of $P(k)$

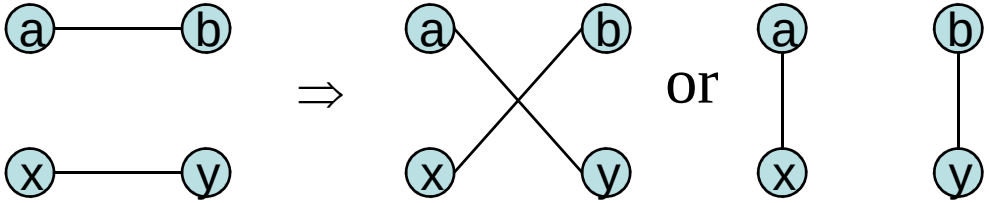


Randomization of real networks (cont)

Pseudocode:

RN(G) -> H
Input:
- G: Original network
Output:
- H: Randomized network
Procedure:
- H <- G
- for *i=1,s x L* {
- do {
- e_1,e_2=randint(*L*),randint(*L-1*)
- *ne1* <- ($e_{1,o}$,$e_{2,f}$)
- *ne2* <- ($e_{2,o}$,$e_{1,f}$)
- until CNTC(H)=True }}
- return(H)

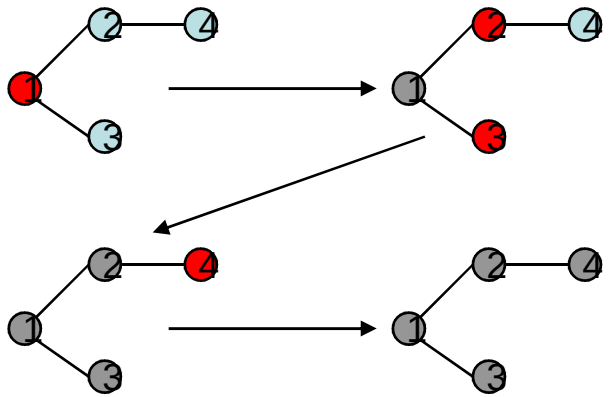
- Link exchange randomization



- Conserves *P(k)*
- Does *not* guarantee connectivity
- Costly due to connectivity check (using routine CNTC (True or False)
- To randomize, choose *s* so expected # of times choosing last link is ≥1. This generally implies *s* ~ 2 or 3 (estimate *s* using negative binomial distribution).

Connectivity Check

- Basic routine. Can be done directly, with Dijkstra, other ways.
- One option is burning algorithm: from any starting node, visit all neighbors of visited nodes at each time step.



- By-product: link-count path length
- AKA: Snowball sampling in population statistical studies

Pseudocode:

CNTC(G,*s*) -> *(True,False)*
Input:
- G: Network
- *s*: Source node

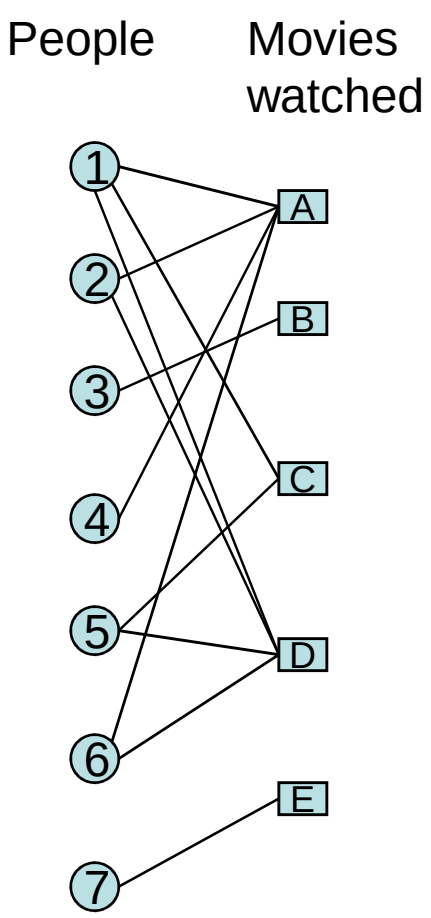
Output:
- *True,False: G* connected or not

Procedure:
- *BF* <- {*s*}
- *V* <- {*s*}
- while $BF \neq \emptyset$ {
- $NBF = \emptyset$
- for *u* in *BF* {
- for *v* in neighbors(*u*) {
- if *v* not in *V* {
- $V \leftarrow V \cup \{v\}$
- $NBF \leftarrow NBF \cup \{v\}$ }}}
- *BF* <- *NBF* }
- return(True if *V*={*1,..,n*}, else False)

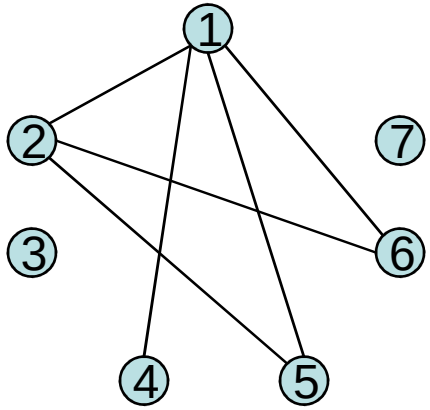
- Trivial change returns component CNTComp

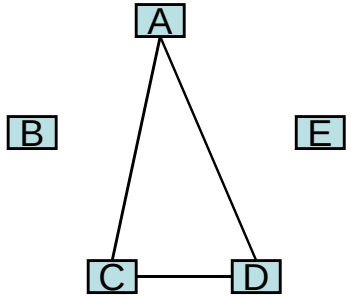
Bipartite Networks and Randomization

- Some networks are naturally (or can be related to) bipartite structures



- Projected networks can be created:





- Objects do not connect directly, but through particular feature⇔Important for randomization

- Examples: Recommendation networks, collaboration networks, genetic diseases, etc.

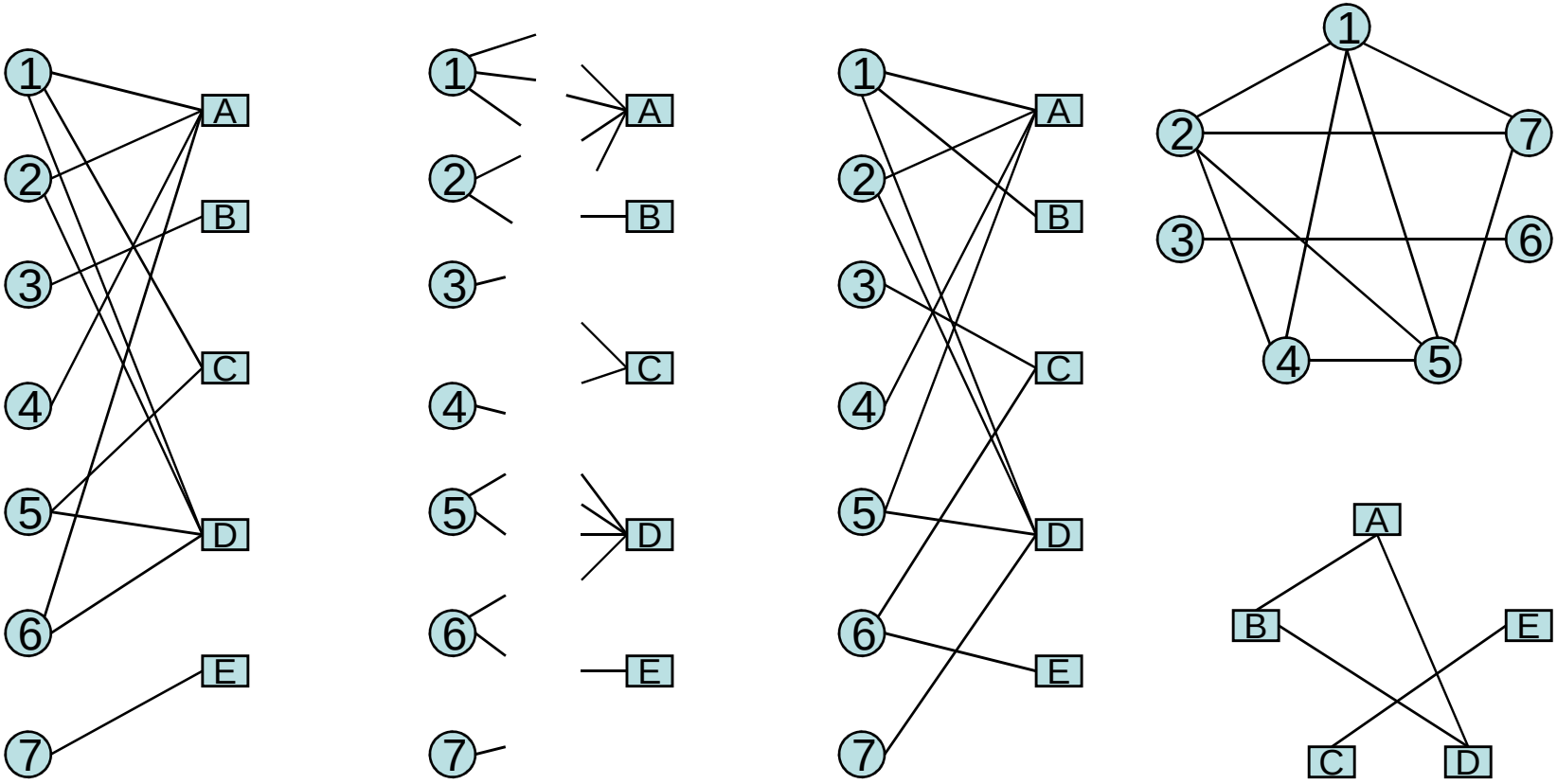
Bipartite Networks and Randomization (cont)

- Algorithm logic⇔preserves degree of people and movies

Original network

Sever connections

Rewire randomly



Bipartite Networks and Randomization (cont)

Pseudocode

BR(G) -> H
Input:
- G: Original bipartite network
Output:
- H: Randomized bipartite network
Procedure:
- *Bipartite degree seqs.* $\{k_i\},\{m_j\}$ <- G
- $K = \varnothing$
- $M = \varnothing$
- for $i=1,n_a$ {
- for $q=1,k_i$ {
- K <- $\{i\}$ }}
- for $j=1,n_b$ {
- for $q=1,m_j$ {
- M <- $\{j\}$ }}
- Λ <- MixB(K,M)
- return(H(Λ))

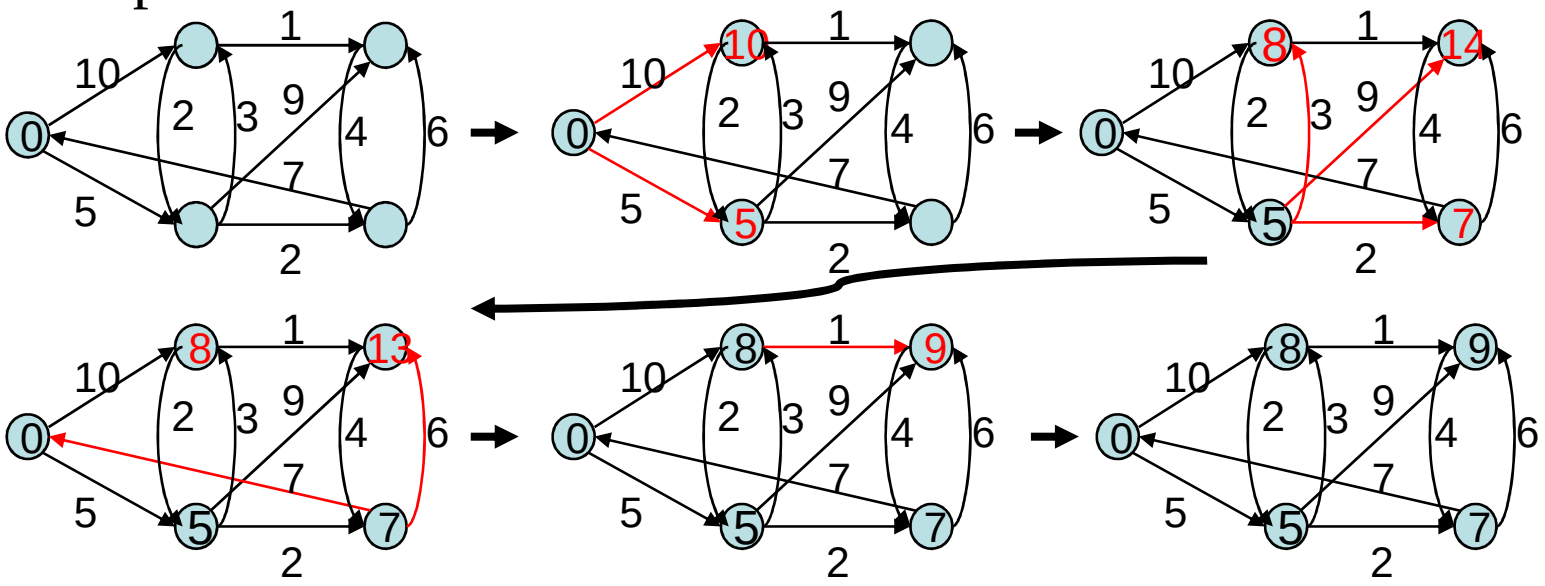
Pseudocode:

MixB(K_0,M_0) -> Λ:
Input:
- K_0,M_0: Degree sequences
Output:
- Λ: Randomized links
Procedure:
- while $K \neq \emptyset$ {
- $K,M=K_0,M_0$
- $\Lambda=\emptyset$
- for x in K {
- y=intrand(M)
- while [(x,y) $\in \Lambda$]{
- M' <- M'(<-M) - $\{y\}$
- y=intrand(M')
- if ($M'=\emptyset$): restart MixB}
- $\Lambda \leftarrow \Lambda \cup \{(x,y)\}$
- K <- K - $\{x\}$,M<- M - $\{y\}$ }}
- return(Λ)

- Bipartite graph: n_a nodes of class a, n_b of b
- Conserves degree distributions $P_a(k_i)$ & $P_b(m_j)$
- Does not guarantee connected network

Path length and Dijkstra's algorithm

- Algorithm based on breath-first search strategy:
  - Start at source node [optimal: end at destination node]; cost to reach=0
  - Visit all source node neighbors
  - Each node is given updated cost to be reached (undefined before)
  - Pick lowest cost node and visit its neighbors. Mark node as 'solved'
  - Continue until making all nodes (or destination node) as 'solved'.
- Example:



Path length and Dijkstra's algorithm

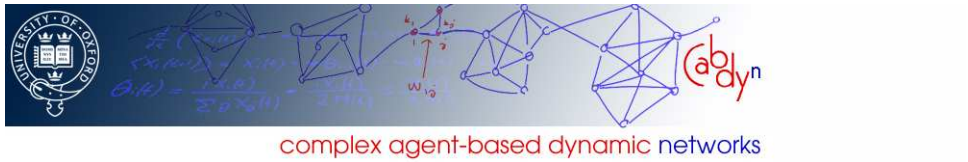
Pseudocode:

Dijkstra(G,*s*,[*d*])->*w*({*N*})
Input:
- G: Network
- *s*: source node
- *d*: destination node

Output:
- *w*({*N*}): cost/distance to node set *N*

Procedure:
- *Q* <- {*s*}
- *w*(*s*) = *0*
- while $Q \neq \emptyset$ {
- *u* <- min(*w*({*Q*}))
- *Q* <- *Q* - {*u*}
- for *v* in neighbors(*u*) {
- if (w(u)+w(u,v)<w(v)) {
- *Q* <- $Q \cup \{v\}$
- *w*(*v*) = *w*(*u*)+*w*(*u*,*v*) }}
- return(*w*({*N*}))

- Notes:
  - For simple link count distance take each link weight *w*(*u*,*v*)=1
  - If only interested in *s* to *d* distance/cost: i) introduce exit clause after finding *u*=*d*, ii) adjust return to desired output.
  - By definition of $\cup$, *Q* <- $Q \cup \{v\}$ does not lead to element duplication in *Q*.
  - Care necessary in function min(w({*Q*})) to achieve optimal performance.



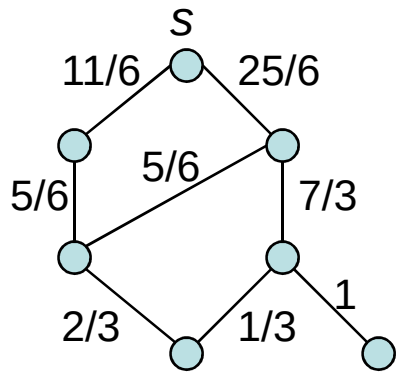
Betweenness: Definitions

- Notion of Betweenness: for a set of paths, measure counting number visiting to a node or link, i.e., way to measure node/link relevance in communication mediated by path set.

- Some specific definitions:
  - Shortest path betweenness: Single shortest path, all shortest paths, count/ignore end nodes.
  - Optimal path betweenness: On weighted network, use optimal paths to calculate betweenness. Same options as before.

- Several algorithms depending on definition. Newman link/node common choice in un-weighted networks, optimal path in weighted networks.

- Example: Newman link (shortest path betweenness due to paths to node *s*)



To calculate total betweenness, one must loop over all nodes as sources and add results.

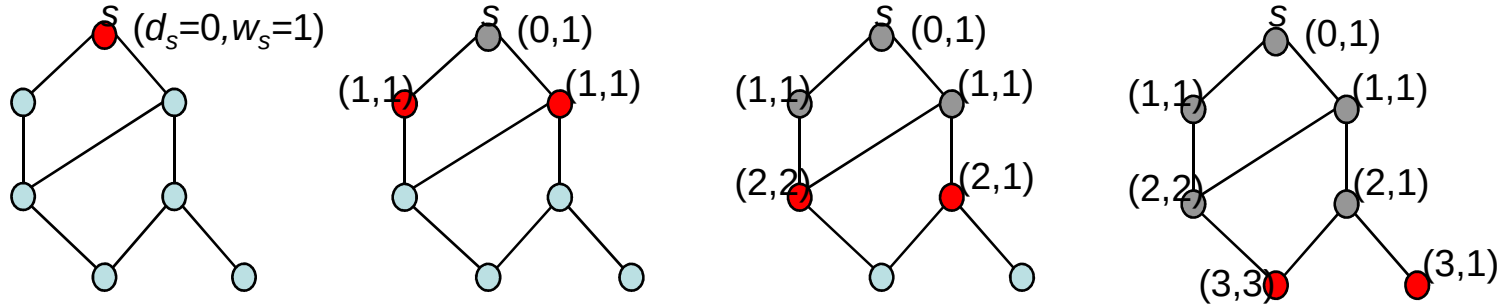
complex agent-based dynamic networks

Betweenness: Newman link algorithm

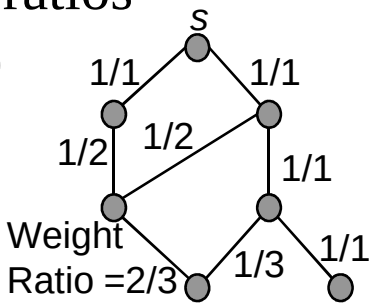
- Based on Breath First Search strategy.
- Looping over all sources *s*, find betweenness due to paths to *s*
- First find all paths reaching a node, then backtrack to find betweenness.
- Example:

-Outbound trip: define variables *d* (distance) & *w* (weight) on all nodes.
w represents # of distinct paths⇒gives relative importance of path.

-Per step:
d=*d*+1
w=# of dist. paths

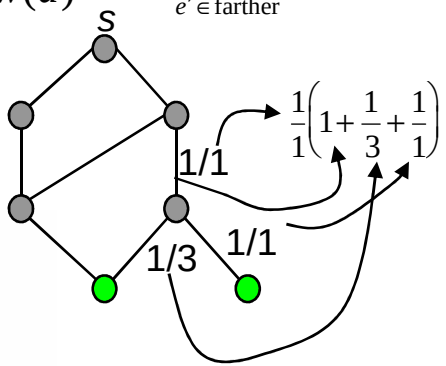


-Find weight ratios w(d-1)/w(d)

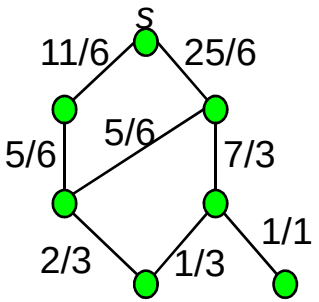


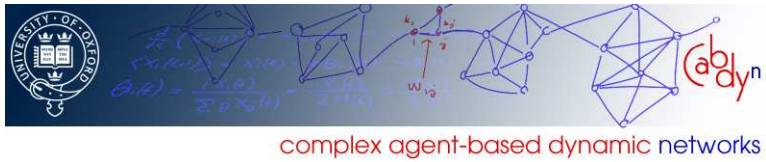
-Backtrack from *d* larger to *s*

$$b_e = \frac{w(d-1)}{w(d)}\left(1+\sum_{e' \in \text{farther}} b_{e'}\right)$$



-For source s, betweenness are





Betweenness: Newman link algorithm (cont)

- Focus on single source code

Pseudocode:

NLBs(G,*s*) -> $\{b_e\}_s$
Input:
- G: Network
- *s*: Source node

Output:
- $\{b_e\}_s$: Link betweennesses from *s*

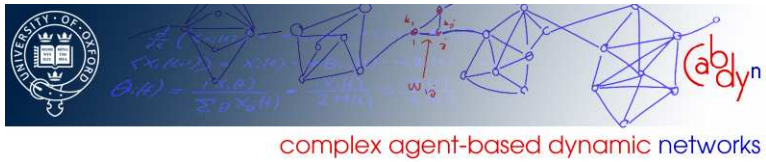
Procedure:
- *W,D* <- FirstRun(G,*s*)
- for *e* in Λ { $\{b_e=w_i/w_j\}_s$ } [$D(j)=D(i)+1$]
- $r_i=\sum_j b_{e(=(i,j))}$ [$D(j)=D_{max}$ (leaves)]
- for *i* in D-D_{max}-1 { (*D* ordered from far to s) {
- for *j* in neighbor(*i*) and $D(j)=D(i)+1$ {
- $\{b_e=b_e \times (1+r_j)\}_s$ }
- $r_i=\sum_j b_{e(=(i,j))}$ }
- return($\{b_e\}_s$)

Pseudocode:

FirstRun(G,*s*) -> *W,D*
Input:
- G: Network
- *s*: Source node

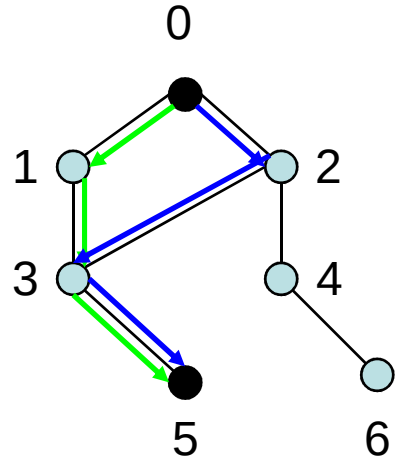
Output:
- *W*: Node weights
- *D*: Node distance from *s*

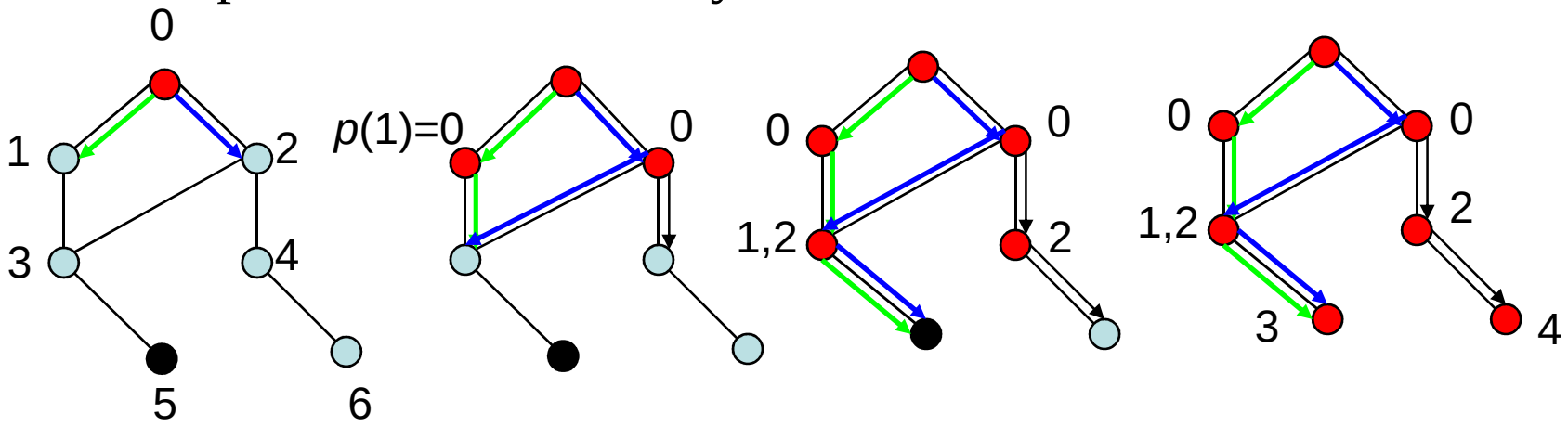
Procedure:
- $D(s)=0, W(s)=1$
- *BF* <- {*s*}
- while $BF \neq \emptyset$ {
- $NBF=\emptyset$
- for *u* in *BF* {
- for *i* in neighbors*(u)* {
- if *D*(*i*) undefined {
- $D(i)=D(u)+1$
- $W(i)=W(u)$
- NBF <- $NBF \cup \{i\}$
- else if $D(i)=D(u)+1$ {
- $W(i)=W(i)+W(u)$ }}}
- *BF* <- *NBF*
- return(*W,D*)



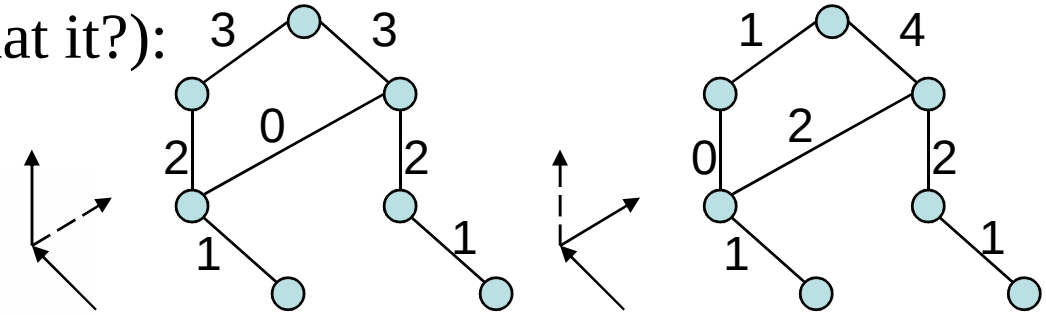
Betweenness in single path case: Algorithm

- Consider case of single paths
- Most networks have multiple paths, e.g. 0 to 5
- Consider notion of predecessors, i.e., node/link visited on way forward. Define $p[V]$ of node set
- At 3, how to choose? Approach: If paths equivalent, choose predecessor randomly. In this case: 1 or 2.





- By returning via predecessors, we obtain (another) non-normalized betweenness. Two results (Is that it?):



Betweenness (cont)

- Focus on single source code (no weights in this case):

Pseudocode:

LBSPs(G,*s*) -> $\{b_e\}_s$
Input:
- G: Network
- *s*: Source node

Output:
- $\{b_e\}_s$: Link betweennesses from *s*

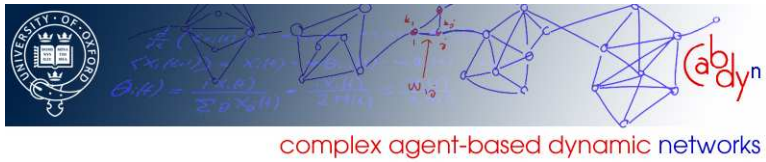
Procedure:
- *P,D* <- FirstRunPre(G,*s*)
- $\{b_e\}_s = 0$
- $\{r_i\}=0$
- for *i* in *D* { (*i* ordered from D_{max} to s) {
- j < - intrand($|P(i)|$)
- $b_{e=(i,j)} = b_{e=(i,j)} + 1 + r_i$
- $r_j = r_j + b_{e=(i,j)}$ }
- return($\{b_e\}_s$)

Pseudocode:

FirstRunPre(G,*s*) -> *P,D*
Input:
- G: Network
- *s*: Source node

Output:
- *P*: Predecessor sets
- *D*: Node distance from *s*

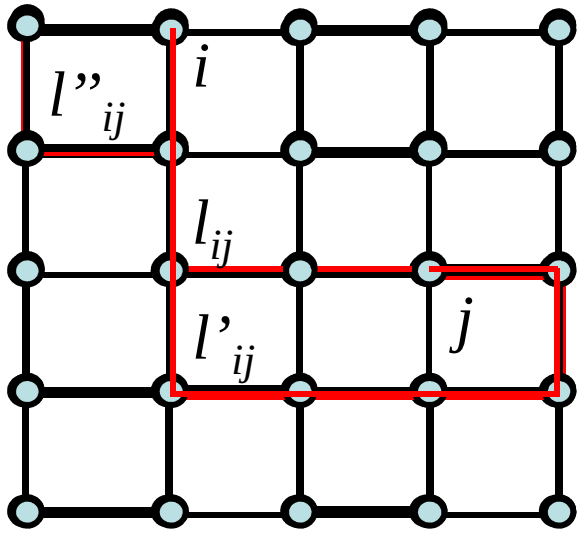
Procedure:
- $D(s)=0, P(s)=\emptyset$
- *BF* <- {*s*}
- while $BF \neq \emptyset$ {
- $NBF=\emptyset$
- for *u* in *BF* {
- for *i* in neighbors*(u)* {
- if *D*(*i*) undefined {
- $D(i)=D(u)+1$
- $P(i) \leftarrow P(i) \cup \{u\}$
- $NBF \leftarrow NBF \cup \{i\}$
- else if $D(i)=D(u)+1$ {
- $P(i) \leftarrow P(i) \cup \{u\}$ } }}
- *BF* <- *NBF*
- return(*P,D*)



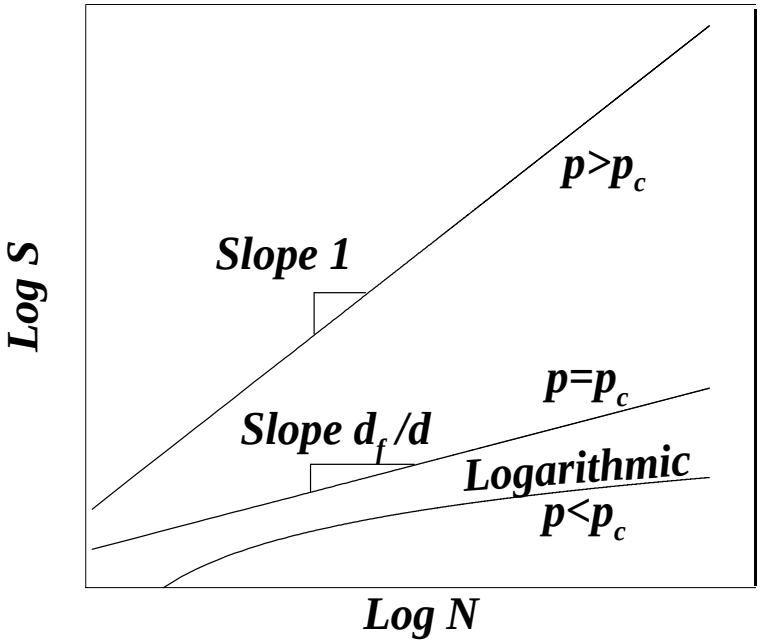
Percolation: Basic theory of network connectivity

- What is minimum condition for network to allow traffic? =To be connected
- How much degradation can network accept before connection is lost? = percolation threshold

p	$S(p)$
1	N
<1	$P_\infty N$
$=p_c$	$N^{df/d}$
$<p_c$	$\log N$



p: occupied fraction of links

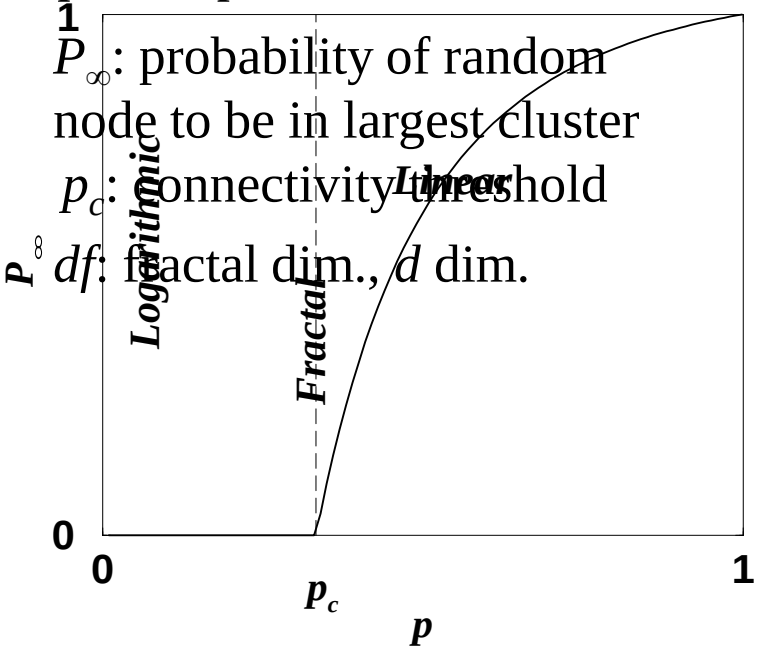


Transition:

connected

↓

disconnected



P_∞: probability of random node to be in largest cluster

p_c: connectivity threshold

df: fractal dim., d dim.

Percolation: Some algorithms

- Most relevant percolation algorithms related to previous slides
- Some important Percolation quantities: i) percolation threshold p_c, ii) dist. of connected cluster sizes, iii) Size $S(p)$ of largest cluster vs. p

Pseudocode:

LPercolate(G, p_f) -> {H_i}
Input:
- G: Original network
- p_f: Final link density

Output:
- {H_i}: Connected clusters set

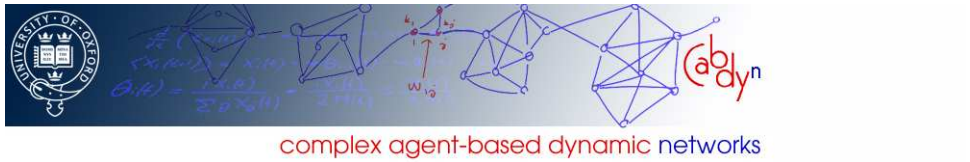
Procedure:
- for l=1,nlr [= L x (1-p_f)] {
- e=rand(Λ)
- Λ <- Λ - {e}
- {H_i (l)} <- FindClusts(G(l)) }
- return({H_i} [full history or final])

Pseudocode:

FindClusts(G) -> {G_i}
Input:
- G: Network

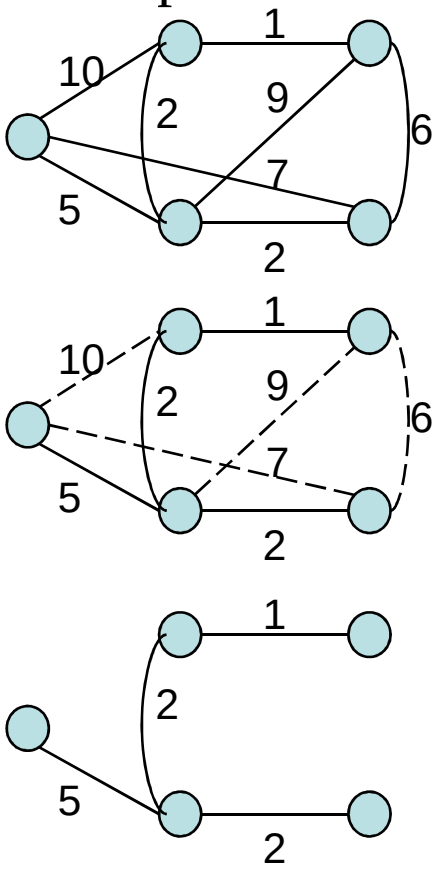
Output:
- {G_i}: Connected clusters

Procedure:
- V <- ∅
- G_{set} <- ∅
- while $V \neq N$ {
- v=rand(N)
- G_{set} <- G_{set} ∪ CNTComp(G,v)
- V <- V - CNTComp(G,v) }
- return(G_{set})



Minimum Spanning Tree

- Weighted networks can be simplified to minimal connected (spanning) tree
- MST is generalization of "all pairs shortest path tree" to weighted networks
- Determined through Prim's or Kruskal's algorithms
- Example:



Pseudocode:

Prim(G,*s*) -> MST
Input:
- G: Network
- *s*: source node

Output:
- MST

Procedure:
- V <- {s}
- while $V \neq N$ {
- i_{new} <- MintoNewN(G,V)
- V <- $V \cup \{i_{new}\}$ }
- return(V)

Pseudocode:

MintoNewN(G,V) -> i
Input:
- G: Network
- V: nodes already visited

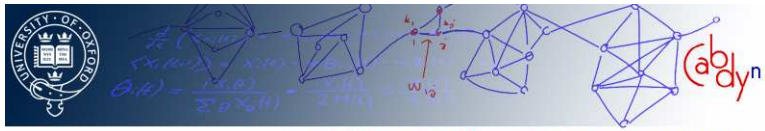
Ouput:
- i: New node through min link

Procedure:
- Λ_{new} <- LinksNew(G,V)
- e_{new} [=(i_{old},i_{new})] <- min(Λ_{new})
- return(i_{new})

- Function LinksNew compares G with nodes found (V) to extract only unsued links of nodes in V.

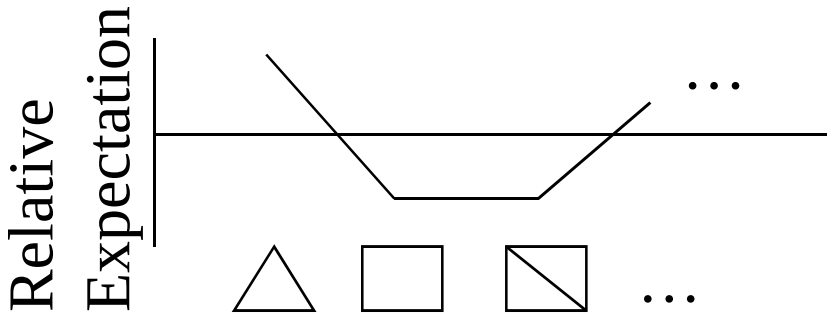
Motifs in Networks

- Question: How many triangles should a network have?
  - Well … it depends on the network.
- This illustrates motifs. Real networks **may** have special structure (e.g. lots of sub-graphs like triangles, long loops, etc.) but this is a relative statement
- In order to determine statistical significance of special features we must
  - Choose and identify the feature
  - Define a base case (null/random model) against which to compare
- Definition of Motifs: Structural network features appearing far more (less) than expected compared to a chosen random network model.
- Algorithmic problem:
  - Identify desired structural feature in real network (including re-weigh of multiple identification of same structure/double counting)
  - Generate random networks to compare with.
  - Identify same features in the random model and determine statistical significance of features in real network.



Motifs in Networks

- Significance profile: i) choose finite feature set (e.g. △ □ ▨ …)
  ii) determine their statistical significance, iii) plot the ratios of the two



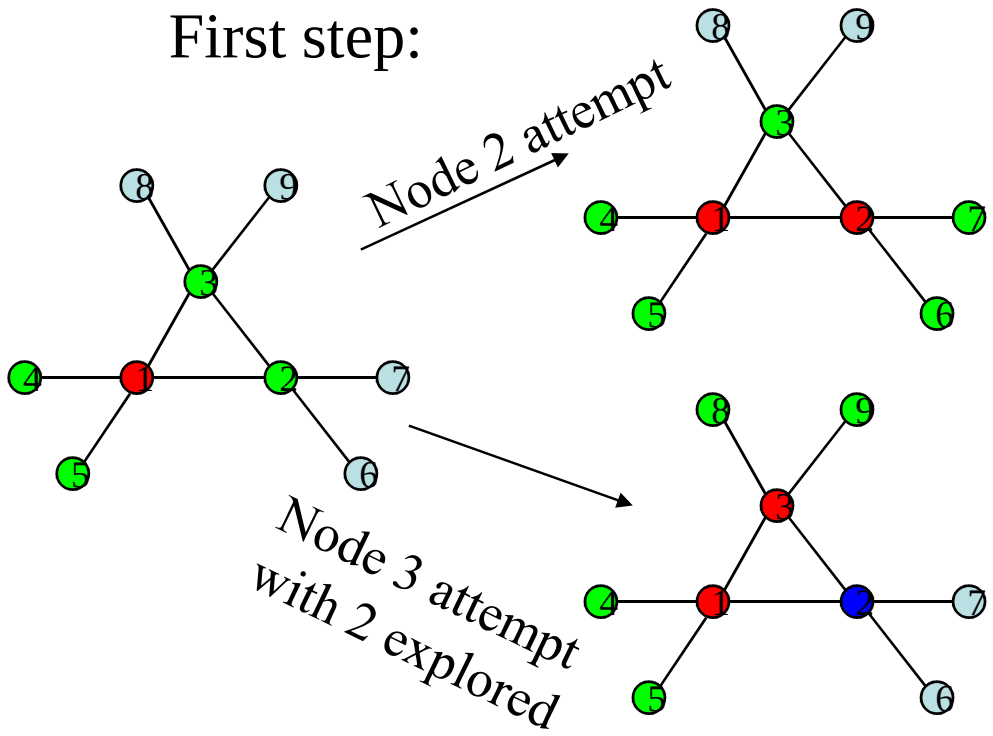
Random flatline=
no stat. significance

- Choice of random model is quite relevant. If choice too random, any feature in significance profile seems special. If choice too specific, possibly no feature is significant.
  - Sensible approach: add one property at a time to random model.
  - Each feature for which significance disappears is likely explained by newly added property to random model.
  - Beware: Large motifs costly to detect due to size.
  - Beware 2: Biased motif sampling leads to wrong significance profile.

Motifs in Networks (Wernicke algorithm)

- Enumerate nodes to avoid multiple visits
- Find subgraphs of desired motif size *m*.

Example of motif count (*m*=3)



- Two more steps: identify isomorphic graphs (*nauty* alg.), and calculate signif.

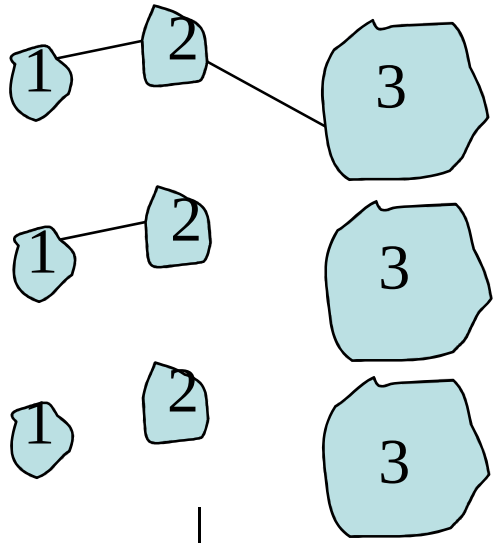
Pseudocode:

CMm(G,*m*) -> {M*m*}
Input:
- G: Network
- *m*: Size of motifs
Output:
- {M*m*}: Subgraphs of size *m*
Procedure:
- for *i=1,n* (nodes need to be labelled) {
- V_g <- {neighbors(i)}
- {M*m*} <- Extend({*i*},Vg,*i*) }
- return({M*m*})

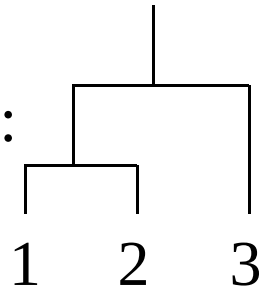
Extend(V_r,V_g,*i*) -> {M*m*}$_j$ (Mm subset)
- if $|V_r| = m$: return(V_r graph)
- while $V_g \neq \emptyset$ {
- V_g <- V_g – {*u* <- intrand(*Vg*)}
- V_g <- $V_g \cup$ {light-blue neighbors(*u*)}
- {M*m*}$_j$ <- Extend($V_r \cup \{u\}$,*Vg*,*i*) }
- return({Mm}j)

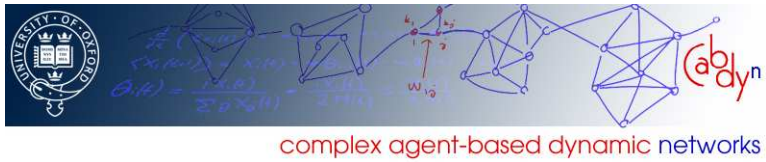
Community Detection: Divisive Clustering

- Notion of Communities: Imagine set of nodes in network with more connections between each other than with rest of network.
- In such cases, one expect betweenness to be larger on links between communities than links inside communities.
- With this motivation, one can set up community detection algorithms based on betweenness. This is core idea of Divisive Clustering.
- Algorithm:
  - Calculate all link betweennesses
  - Remove link with largest betweenness
  - Recalculate all betweenness
  - Keep track of cluster splits
  - Repeat until all links are eliminated
- Outcome: Dendrogram (binary tree) of network community structure



Dendrogram:





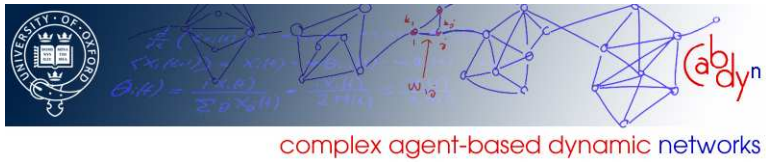
Community Detection: Divisive Clustering

- Algorithm:

Pseudocode:

DC(G) -> T
Input:
- G: Network
Output:
- T: Dendrogram
Procedure:
- while $\Lambda \neq \emptyset$ {
- {*b*} <- NLB(G)
- Λ' <- Λ – Max({b})
- if NC(G')=NC(G)+1 {
- T <- T ∪ {network split}
- Λ <- Λ'
- return(T)

- NC(G) counts number of clusters.
- Keeping track of dendrogram formation requires units to measure significance of split.
- One can use the density of links wrt original network as scale
- Word of caution: Important with any community detection algorithm to check the communities produced. This algorithm is top down ⇒ once lots of network removed, small scale is resolved with little information.



Community Detection: Modularity Maximization

- Modularity $Q = \frac{1}{2L}\sum_{ij}\left(A_{ij} - \frac{k_i k_j}{2L}\right)\delta(c_i, c_j)$
- Goal: Find communities such that Q is maximized.
- Full enumeration costly $\Rightarrow$ Approximate methods needed, e.g., simulated annealing, genetic algorithms, spectral methods, etc.
- In matrix form, modularity $Q = \frac{1}{4L}\mathbf{s}^T\mathbf{B}\mathbf{s}$, $B_{ij} = A_{ij} - \frac{k_i k_j}{2L}$, where **s** is indicator in bipartitioning of network into communities: s_i=1 if node i belongs to community, and s_i=-1 if it belongs to another.
- By solving eigenvalue problem, with $\{\mathbf{u}_j\}$ set of eigenvectors of **B**, and $\{\beta_j\}$ the eigenvalues, $Q = \frac{1}{4L}\sum_{i=1}^{n}(\mathbf{u}_i^T \bullet \mathbf{s})^2 \beta_i$
- Bipartitioning network involves maximizing Q through choosing **s** to maximize via largest eigenvalue. One first partition is found, subsequent partitions are found in similar way.

Community Detection: Modularity Maximization

- Goal: Find communities such that Q is maximized.

$$Q = \frac{1}{4L}\sum_{i=1}^{n}(\mathbf{u}_i^T \bullet \mathbf{s})^2 \beta_i \qquad \mathbf{u}_i = (u_{i1}, u_{i2}, \ldots, u_{in}) \qquad \mathbf{s} = (s_1, \ldots, s_v, \ldots, s_n);\ s_v = \pm 1$$

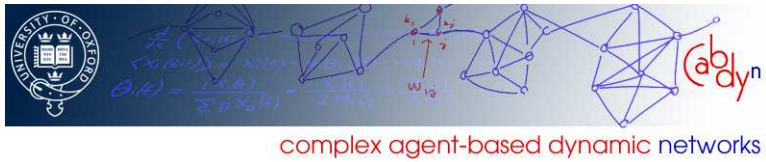
- $u_{iv} > 0$ or < 0. For i=1 such that β_1 is largest eigenvalue, choose s_v so that product $u_{1v}\, s_v > 0$ always for all v.
- Once first division done, each piece may divide again, provided it adds modularity.
- Generally maximize (g: community)

$$\frac{1}{4L}\mathbf{s}^T\mathbf{B}^{(g)}\mathbf{s};\ B_{ij}^{(g)} = B_{ij} - \delta_{ij}\sum_{k\in g} B_{ik}$$

- Remaining task: find largest eigenvalue
- Use Power method, iterate: $\mathbf{x}_q = \dfrac{\mathbf{B}\mathbf{x}_{q-1}}{\left\|\mathbf{B}\mathbf{x}_{q-1}\right\|}$
- Convergence ratio $|\beta_2/\beta_1|$

Pseudocode:

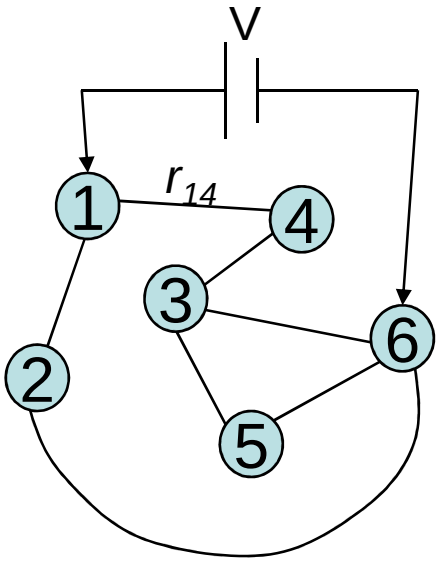
Qmax(G,g,$\mathbf{s}_0$=(1,…,)) -> **s(*g*)**
Input:
- G: Network, *g*: community
Output:
- **s**: Community selection
Procedure:
- **B** <- $\mathbf{B}^{(g)}$
- for {
  - $\mathbf{u}_1, \beta_1$ <- Power(**B**)
  - for v=1,n {s_v=sign($\mathbf{u}_{1v}$)}
  - **s**(1) <- Qmax(G,1,**s**(g))
  - **s**(2) <- Qmax(G,2,**s** (g))
  - recalculate Q
- until $\beta_1 \le 0$ of $\Delta Q \le 0$ }
- return(**s**)



Flow in Networks: Circuit Analysis

- Networks typically carry flows (information, passengers, viruses, electrical power, etc.)
- Network structures may evolve as optimized solution to flow
- Flow details can vary, e.g. DC circuits, laminar hydraulic circuits

Example: DC circuits-- Find conductance distribution



-Circuits can be homogeneous ($r = 1 \ \forall$ links) and heterogeneous (different r for each link)

-Source(s)/sink(d) choice based on problem

-Apply constant voltage or current

-Find R_{sd} and/or distribution $P(R_{sd})$

-With change to AC circuit, results relevant to power grid

Circuit Analysis algorithm

- Solve node potential equations from current conservation

$$\sum_{j\in Neigh(i)} I_{ij} = \sum_{j\in Neigh(i)} \frac{1}{r_{ij}}(V_i - V_j) = V_i \sum_{j\in Neigh(i)} \frac{1}{r_{ij}} - \sum_{j\in Neigh(i)} \frac{V_j}{r_{ij}} = I_i \;;\; \forall i = 1,\ldots,n \;\; (r_{ij} = r_{ji})$$

- Define Laplacian matrix $\vec{\lambda}$ of network from previous equation

$$\begin{pmatrix} \sum_{j\in Neigh(1)} \frac{1}{r_{1j}} & -\frac{1}{r_{12}} & \cdots & -\frac{1}{r_{1n}} \\ -\frac{1}{r_{21}} & \sum_{j\in Neigh(2)} \frac{1}{r_{2j}} & \cdots & -\frac{1}{r_{2n}} \\ \vdots & \vdots & \cdots & \vdots \\ -\frac{1}{r_{n1}} & -\frac{1}{r_{n2}} & \cdots & \sum_{j\in Neigh(2)} \frac{1}{r_{nj}} \end{pmatrix} \begin{pmatrix} V_1 \\ V_2 \\ \vdots \\ V_n \end{pmatrix} = \begin{pmatrix} I_1 \\ I_2 \\ \vdots \\ I_n \end{pmatrix} \Rightarrow \vec{\lambda}\vec{\mathrm{V}} = \vec{\mathrm{I}}$$

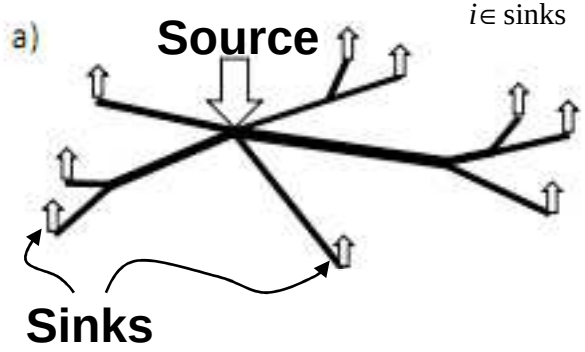
- Application of boundary conditions requires conservation of total current input/output, reducing matrix & specifying some V_i, I_i values

- For all r =1, Laplacian matrix = diagonal degree - adjacency matrix

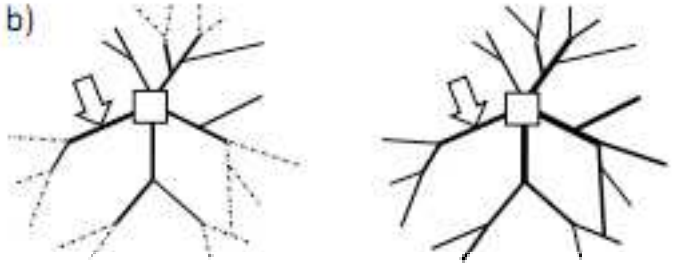
Circuit Analysis applied to Fungal Networks

- Example: Fungal Networks
  - BC: i) One source node, q sink nodes, ii) Growing network dictates in/out current.

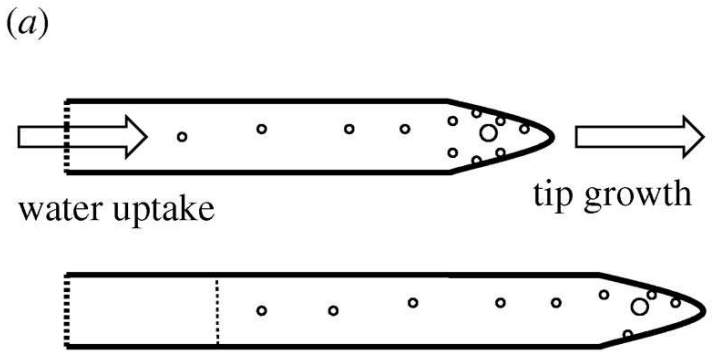
i) Current conservation: $I_s = -\sum_{i \in \text{sinks}} I_i$

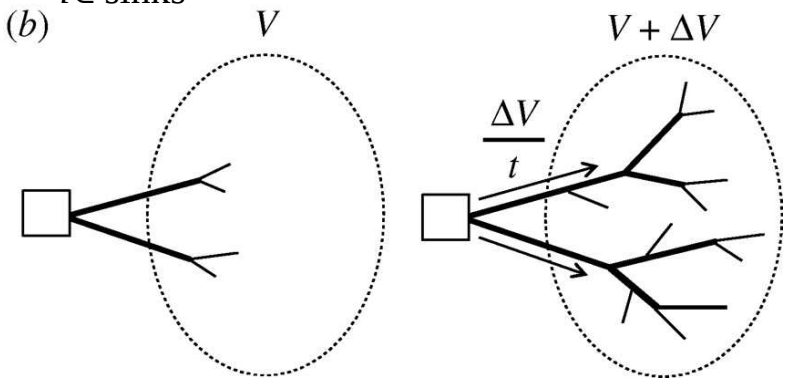


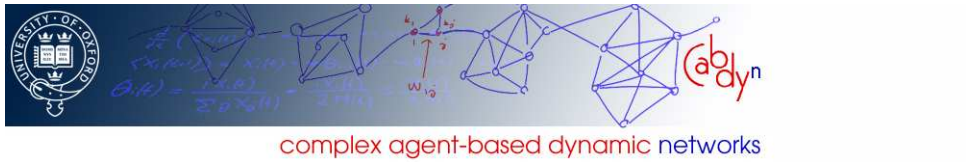
Network growth



ii) Incompresibility of fluid+ growth: $I_{i \in \text{sinks}} = \Delta Volume / t$







Circuit Analysis applied to Fungal Networks (cont)

- Example: Fungal Networks

Pseudocode

FunFlow(G(*t*)) -> $\vec{I}(t)$
Input:
- G(*t*): Time evolving network
Output:
- $\vec{I}(t)$: Link currents over time
Procedure:
- for *t=1,…,T-1* (*T*: final time) {
- for *e*(*t*) in $\Lambda(t)$ {
- $I_{e_o}(t)$, $I_{e_f}(t)$ = *g*(*e*(*t*),*e*(*t+1*))
- $I_s(t) = -\sum_i I_i(t)$ }
- $\vec{\Lambda}(t)$ <- $\Lambda(t)$
- $\vec{V}(t)$ <- CircSolve($\vec{\Lambda}(t)$,{$I_s(t), I_i(t)$})
- $\vec{I}(t) = F(\vec{V}(t))$ }
- return($\vec{I}(t)$)

- Function *g* determines current BC based on link changes. *g may* be global as opposed to local. If so, loop over edges not necessary but *g* more complex.

- Function *F* is Ohm's law

- Routine CircSolve inverts eq.

$$\vec{L}\vec{V} = \vec{I} \Rightarrow \vec{V} = \vec{L}^{-1}\vec{I}$$

- To invert matrices, *always* use numerical package (do not try this at home!)

Conclusions

- In writing network algorithms, consider the entire problem and plan ahead.

- Choose your tools according to the problem and consider different possibilities before starting to code.

- Test code extensively, either custom made or from a package.

- Writing code is, in general, a practical task and not the goal itself. Remember this and always be practical: balance the desire/need to write good code with the time spent on it. Use your best judgement.