Bioinformatics 3

V 2 – Clusters, Dijkstra, and Graph Layout

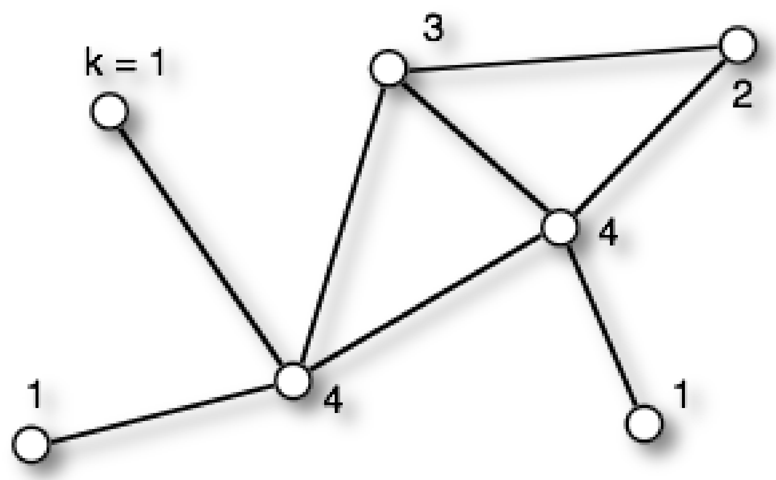
Mon, Oct 26, 2015

Graph Basics

A **graph** *G* is an ordered pair *(V, E)* of a set *V* of **vertices** and a set *E* of **edges**.

Degree distribution *P(k)*

$$P(k) = \frac{n_k}{N}$$



k	0	1	2	3	4
P(k)	0	3/7	1/7	1/7	2/7

Random network:
also called the "Erdös-Renyi model"
start from all nodes, add links randomly
P(k) = "Poisson“ (will show this on the next slides)

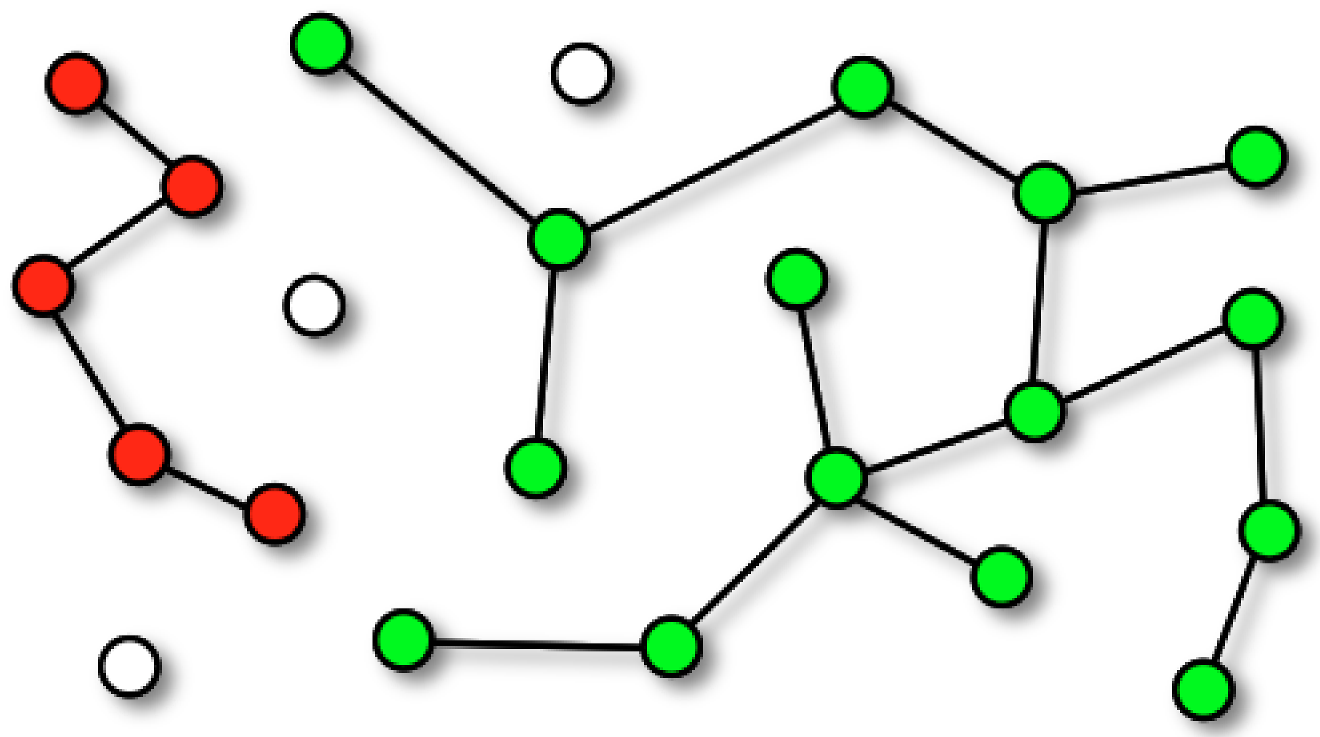
$$P(k) = \frac{\lambda^k}{k!} e^{-\lambda}$$

Scale-free network:
grow with preferential attachment
P(k) = power law (*dt. Potenzgesetz*)

Connected Components

Connected graph <=> there is a path between all pairs of nodes

In large (random) networks: complete $\{V\}$ often not connected
$\rightarrow$ identify connected subsets $\{V_i\}$ with $\{V\} = \text{U} \{V_i\}$
$\rightarrow$ **connected components** (CC)



#CC = 5
N_{max} = 15
N_{min} = 1

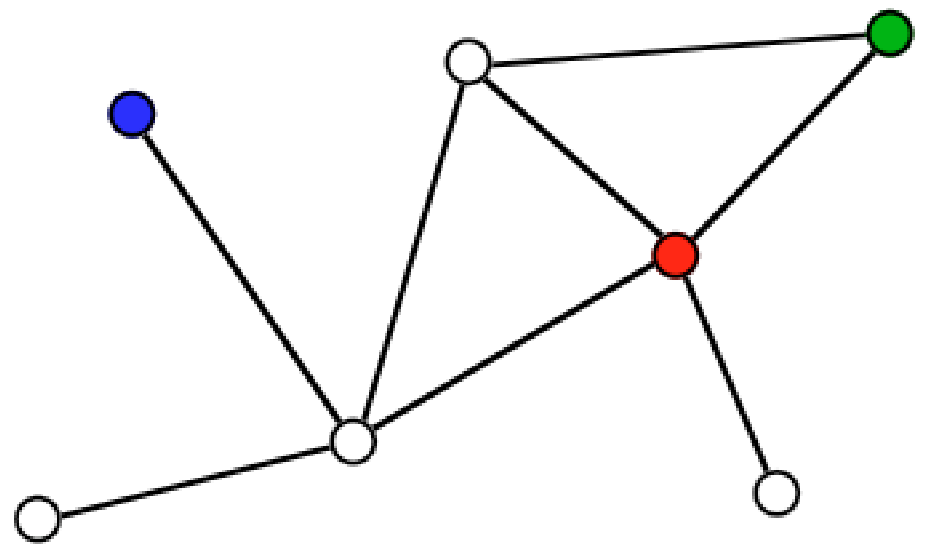
Connectivity of the Neighborhood

How many of the neighboring vertices are themselves neighbors?
=> **clustering coefficient** *C(k)*

Number of possible edges between *k* nodes: $n_{max} = \frac{k(k-1)}{2}$

n_k is the actual number of edges between the neighbor nodes.

Fraction of actual edges ≅ **clustering coefficient** $C(k, n_k) = \frac{2n_k}{k(k-1)}$



green: $k = 2$, $n_k = 1$ $\rightarrow$ $C = 1$

red: $k = 4$, $n_k = 2$ $\rightarrow$ $C = 1/3$

blue: $k = 1$, $n_k = ?$ $\rightarrow$ C is not defined

Note: clustering coeff. sometimes also defined via fraction of possible triangles

Clustering Coefficient of a Graph

Data: C_i for each node i $\rightarrow$ N values

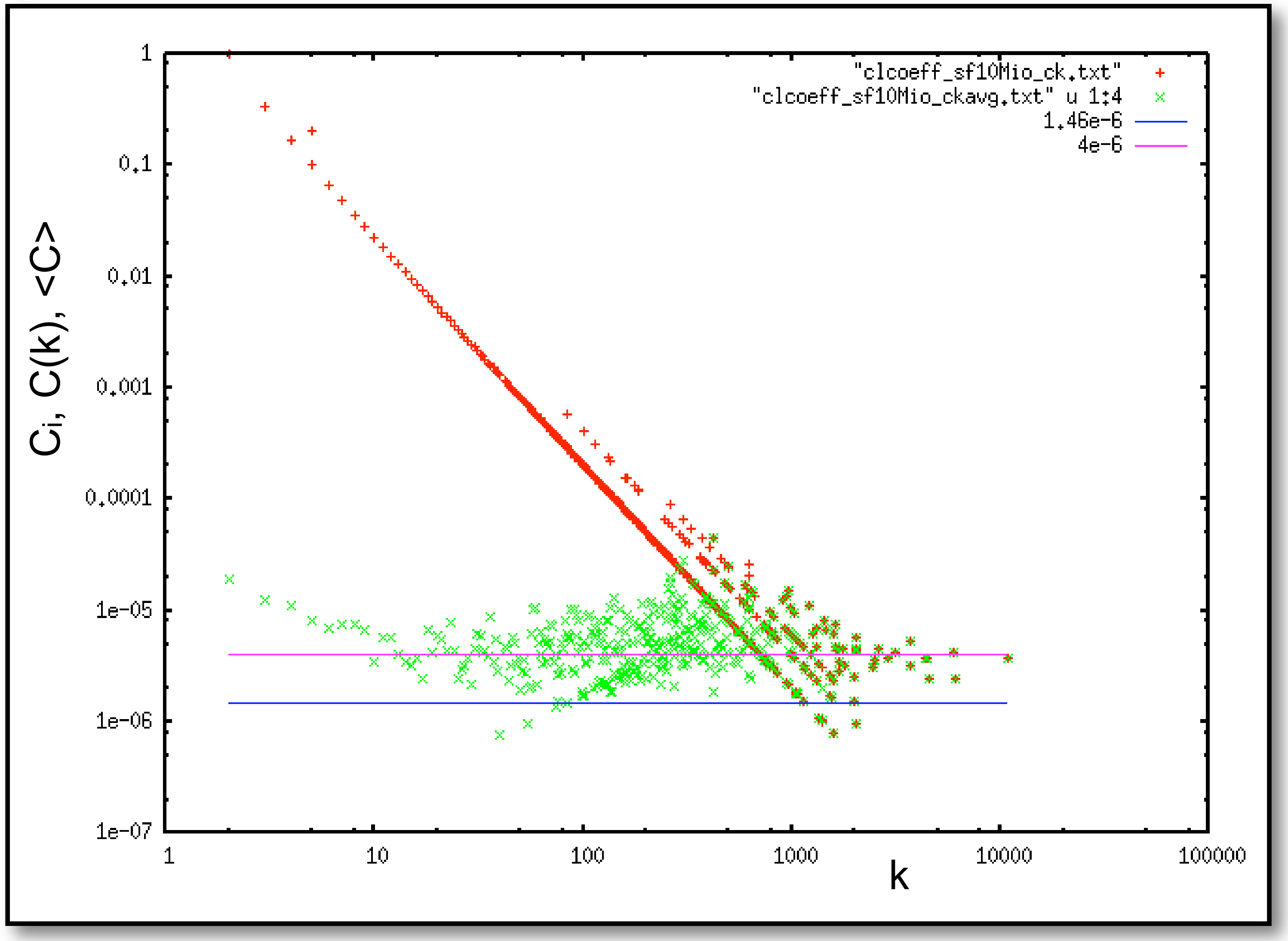
Statistics:

average at **fixed k**

$$\rightarrow \quad C(k) = \frac{1}{n_k} \sum_{k_i = k} C_i$$

average over **all nodes**

$$\rightarrow \quad \langle C \rangle = \frac{1}{N} \sum C_i$$



Note: it is also possible to average the *C(k)*
$\Rightarrow$ This yields a different value for <*C*> !!!
because no weighting is done for different occupancy of k's.

Basic Types: (1) Random Network

Generally: *N* vertices connected by *L* edges

More specific: **distribute** the edges **randomly** between the vertices

Maximal number of links between *N* vertices:

$$L_{max} = \frac{N(N-1)}{2}$$

=> **probability** *p* for an edge between two randomly selected nodes:

$$p = \frac{L}{L_{max}} = \frac{2L}{N(N-1)}$$

=> **average degree** λ

$$\lambda = \frac{2L}{N} = p\,(N-1)$$

path lengths in a random network grow with log(*N*) => small world

Random Network: *P(k)*

Network with *N* vertices, *L* edges
=> probability for a random link:

$$p = \frac{2L}{N(N-1)}$$

Probability that random node has links to *k* other particular nodes:

$$W_k = p^k \, (1-p)^{N-k-1}$$

Probability that random node has links to any *k* other nodes:

$$P(k) = \binom{N-1}{k} W_k = \frac{(N-1)!}{(N-k-1)!\,k!} W_k$$

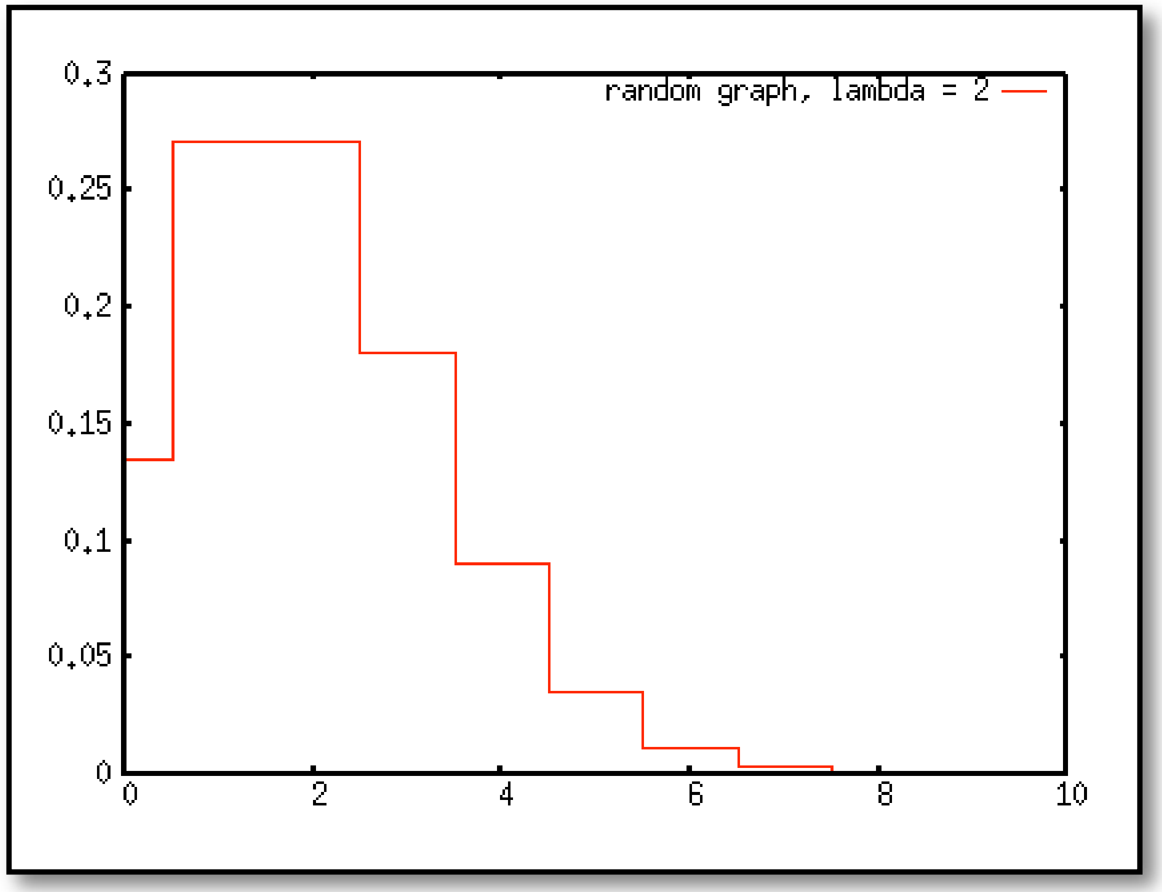
Limit of large graph: $N \rightarrow$ oo, $p = \lambda / N$

$$\begin{aligned}
\lim_{N\to\infty} P(k) &= \lim_{N\to\infty} \frac{N!}{(N-k)!\,k!} p^k \, (1-p)^{N-k} \\
&= \lim_{N\to\infty} \left(\frac{N(N-1)\ldots(N-k+1)}{N^k} \right) \frac{\lambda^k}{k!} \left(1 - \frac{\lambda}{N}\right)^N \left(1 - \frac{\lambda}{N}\right)^{-k} \\
&= \qquad 1 \qquad \frac{\lambda^k}{k!} \qquad e^{-\lambda} \qquad 1 \\
&= \frac{\lambda^k}{k!} e^{-\lambda}
\end{aligned}$$

Random Network: *P(k)*

Many independently placed edges => **Poisson statistics**

$$P(k) = \frac{\lambda^k}{k!} e^{-\lambda}$$



k	*P(k | λ = 2)*
0	0.135335283237
1	0.270670566473
2	0.270670566473
3	0.180447044315
4	0.0902235221577
5	0.0360894088631
6	0.0120298029544
7	0.00343708655839
8	0.000859271639598
9	0.000190949253244
10	3.81898506488e-05

=> Small probability for $k >> \lambda$

Basic Types: (2) Scale-Free

Growing network a la Barabasi and Albert (1999):
- start from a small "nucleus"
- add new node with *n* links
- connect new links to existing nodes with probability proportional to degree *k* (preferential attachment; β(BA) = 1)

$$p_i = \left(\frac{k_i}{\sum k_i}\right)^{\beta}$$

=> "the rich get richer"

Properties:

- this leads to power-law degree distribution:

$$P(k) \propto k^{-\gamma}$$ with γ = 3 for the BA model

- self-similar structure with highly connected hubs (no intrinsic length scale)
  => path lengths grow with log(log(*N*))
  => very small world

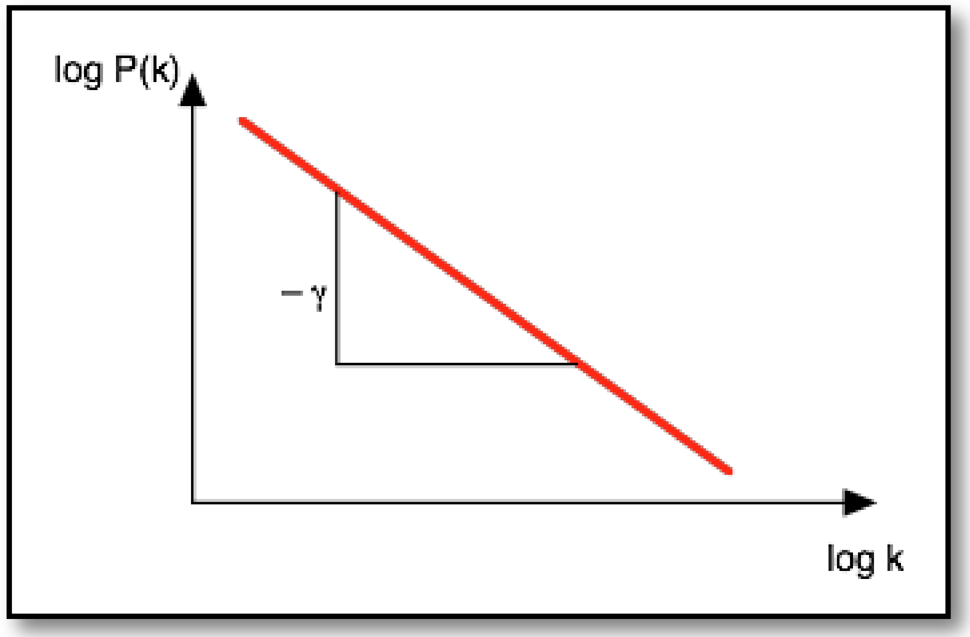
The Power-Law Signature

Power law $P(k) \propto k^{-\gamma}$

Take log on both sides:

$$\log(P(k)) = -\gamma \log(k)$$

Plot log(*P*) vs. log(*k*) => straight line



Note: for fitting γ against experimental data it is often better to use the integrated *P(k)*
=> integral smoothes the data

$$\int_{k_0}^{k} P(k)dk = \left[-\frac{k^{-(\gamma-1)}}{\gamma}\right]_{k_0}^{k}$$

Scale-Free: Examples

The World-Wide-Web:

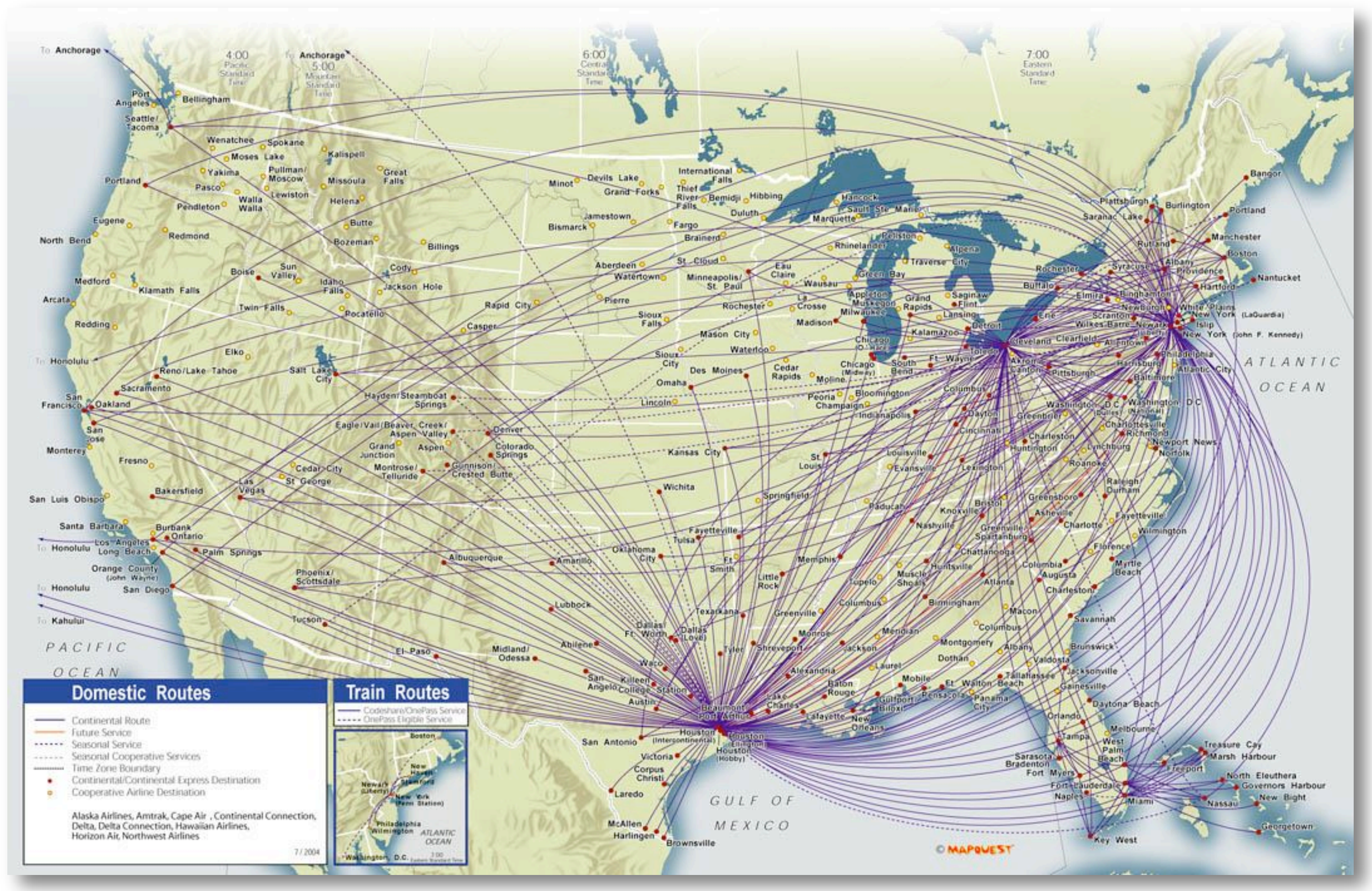
=> growth via links to portal sites

Flight connections between airports

=> large international hubs, small local airports

Protein interaction networks

=> some central, ubiquitous proteins



http://a.parsons.edu/~limam240/blogimages/16_full.jpg

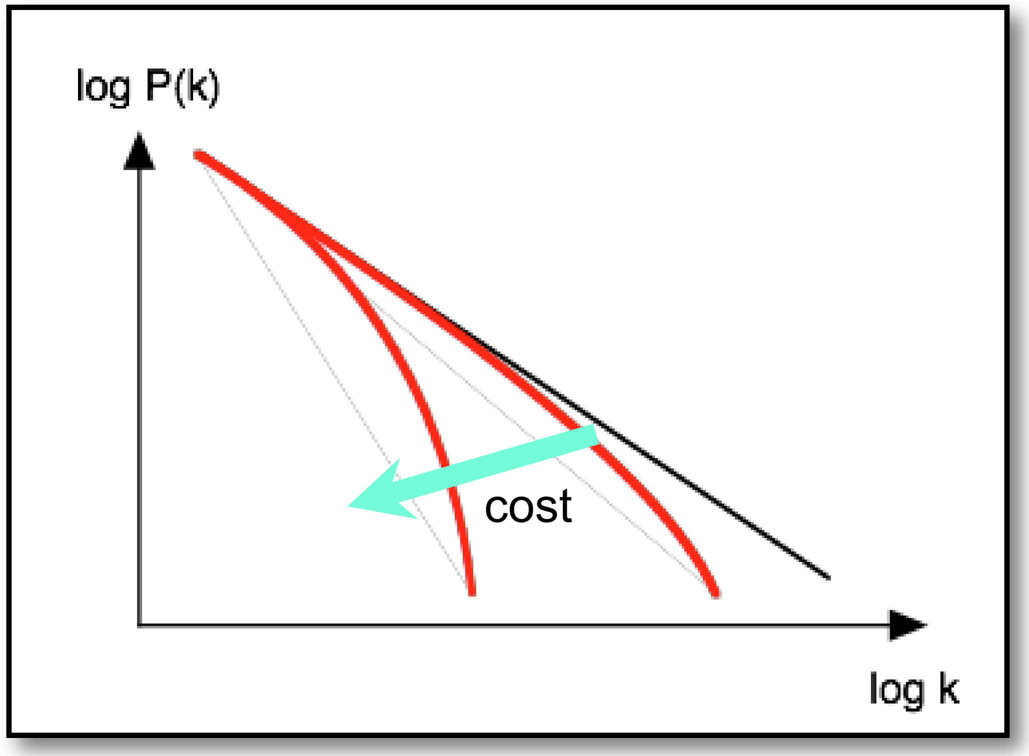
Saturation: Ageing + Costs

Example:	network of movie actors (with how many other actors did an actor appear in a joint movie?)

Each actor makes new acquaintances for ~40 years before retirement
=> limits maximum number of links

Example: building up a physical computer network

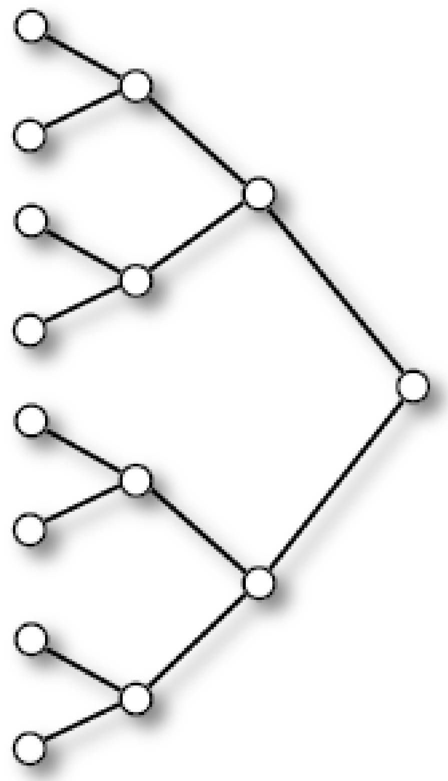
It gets more and more expensive for a network hub to grow further
=> number of links saturates

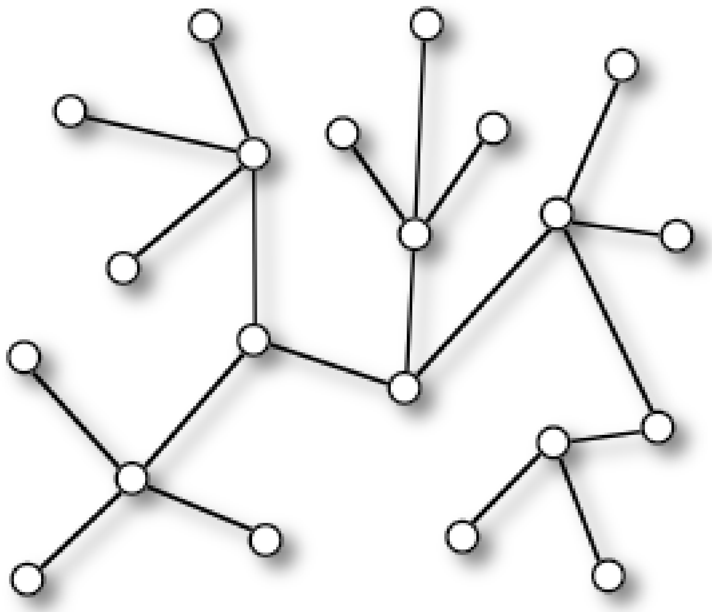


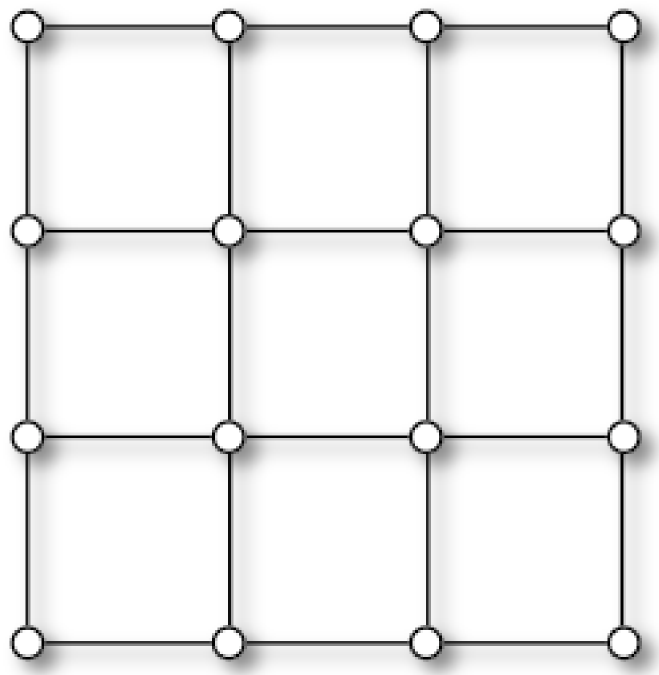
Hierarchical, Regular, Clustered…

Tree-like network with similar degrees
=> like an organigram
=> hierarchic network

All nodes have the same degree and the same local neighborhood
=> regular network







P(k) for these example networks? (finite size!)

Note: most real-world networks are somewhere in between the basic types

C(k) for a Random Network

Clustering coefficient when *m* edges exist between *k* neighbors

$$C(k,m) = \frac{2m}{k(k-1)}$$

Probability to have exactly *m* edges between the *k* neighbors

$$W(m) = \binom{k}{m} p^m (1-p)^{\frac{k(k-1)}{2}-m}$$

In this way, we pick the *m* start nodes for the *m* edges from the *k* nodes.

Average *C(k)* for degree *k*:

$$C(k) = \frac{\sum_{m=0}^{\frac{k(k-1)}{2}} W(m)\, C(k,m)}{\sum_{m=0}^{\frac{k(k-1)}{2}} W(m)} = \ldots = p$$

$\rightarrow$ *C(k)* is independent of *k*
<=> same local connectivity throughout the network

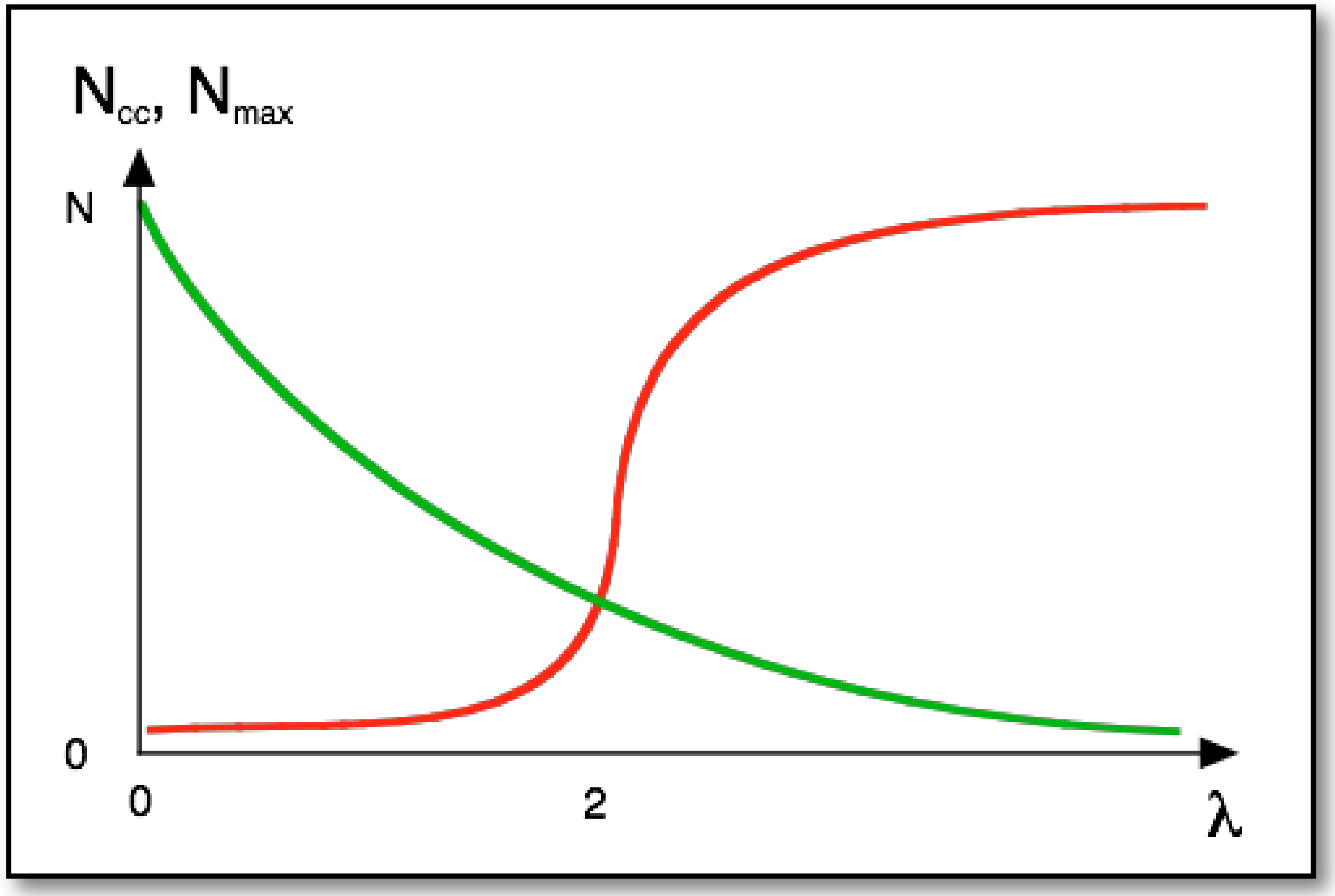
The Percolation Threshold

Connected component = all vertices that are connected by a path

Very few edges
$\Rightarrow$ only CCs
of size 2

$\xleftrightarrow[\text{transition at}]{\text{Percolation}}$
$\lambda = 2$

Many edges
$\rightarrow$ graph is one CC



average degree λ

$$\lambda = \frac{2L}{N} = p(N-1)$$

Identify:

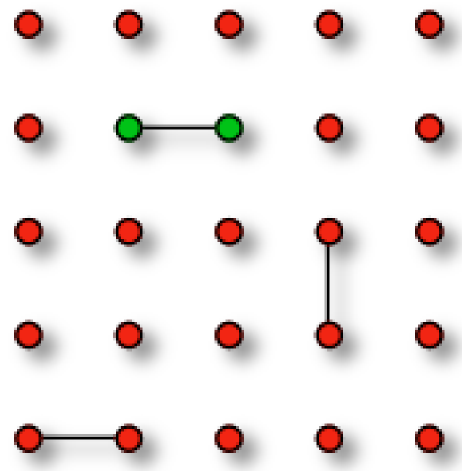
N_{cc} = number of connected components (clusters) (green)

N_{max} = size of the largest cluster (red)

For $\lambda > 2$:
"giant component" exists

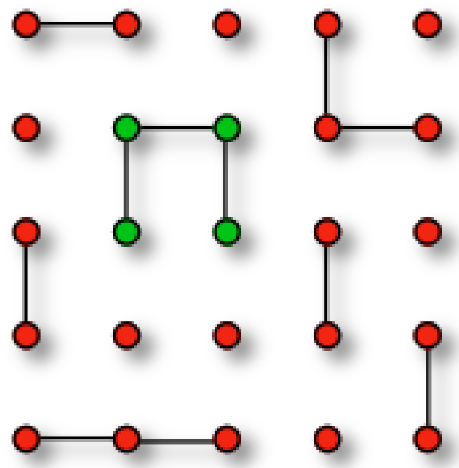
Percolation Transition

Example: regular square lattice, $N = 25$ nodes, $L_{max} = 40$ links between next neighbors



$L = 3$
$\lambda = 0.24$

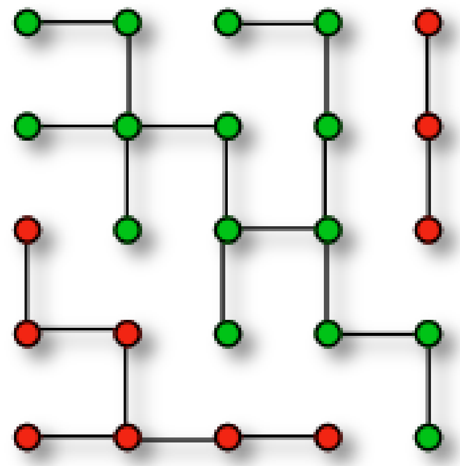
$N_{cc} = 22$
$N_{max} = 2$



$L = 11$
$\lambda = 0.88$

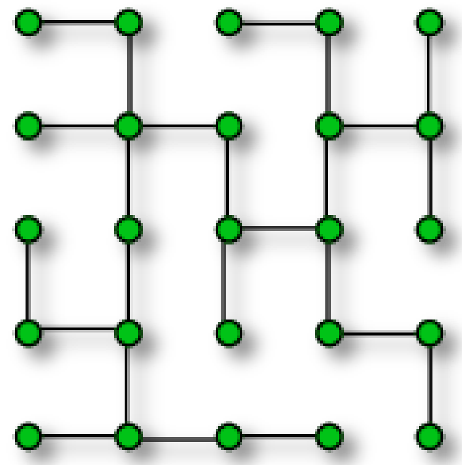
$N_{cc} = 14$
$N_{max} = 4$

"percolation"



$L = 22$
$\lambda = 1.76$

$N_{cc} = 3$
$N_{max} = 15$



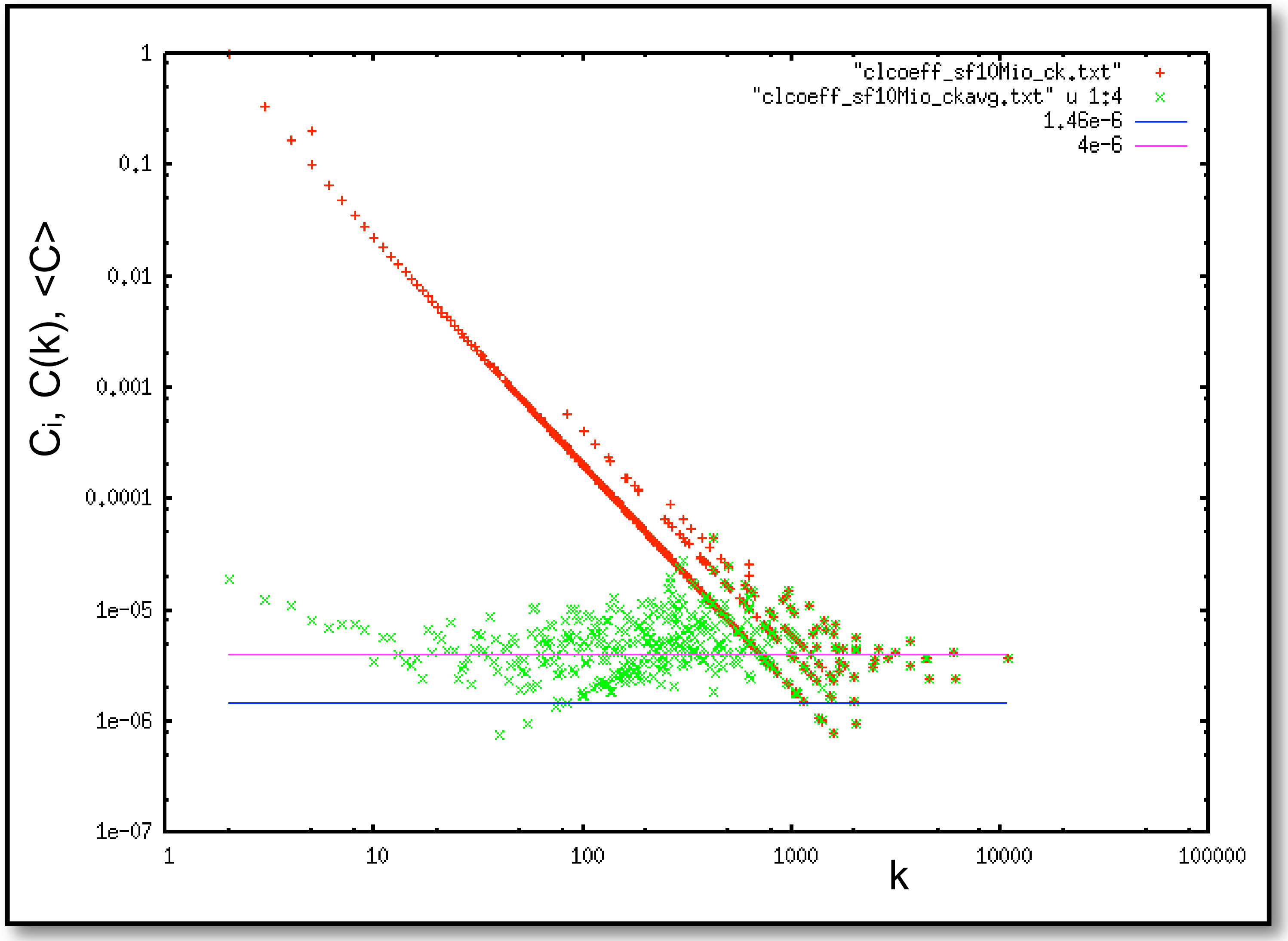
$L = 24$
$\lambda = 1.92$

$N_{cc} = 1$
$N_{max} = 25$

percolation = "spanning structure" emerges (long range connectivity)
for an infinite square lattice: percolation transition at $\lambda = 2$
here: finite size effect <=> fewer possible links at the boundaries

Clusters in scale free graphs

Scale-free network <=> no intrinsic scale
$\rightarrow$ same properties at any *k*-level
$\rightarrow$ same local connectivity
$\rightarrow$ *C(k)* = const.

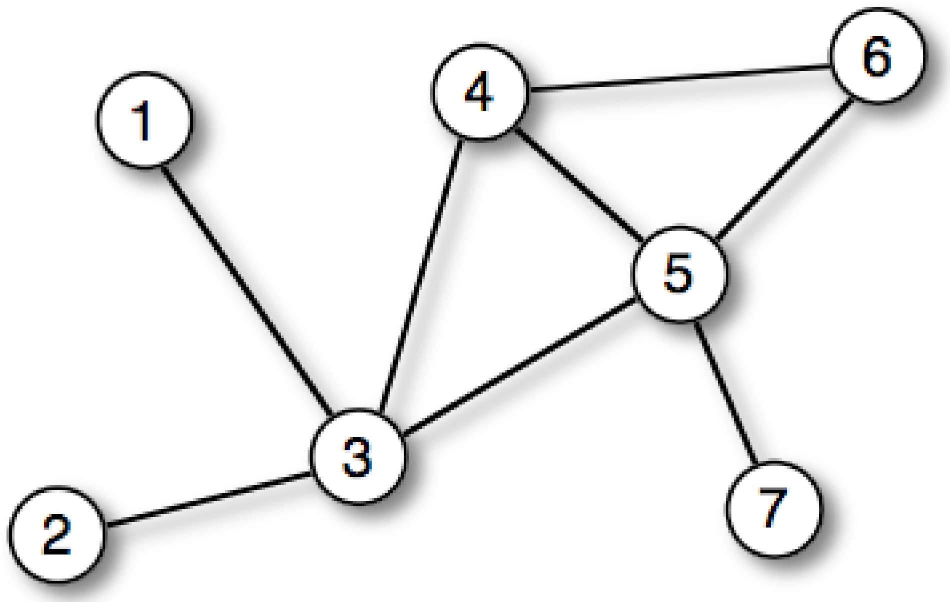


"Real" biological data
$\rightarrow$ missing links
$\rightarrow$ multiple clusters

Is the metabolic network of a cell fully connected?

Algorithms on Graphs

How to **represent** a graph in the **computer**?



1. **Adjacency list**
=> list of neighbors for each node

1:	(3)
2:	(3)
3:	(1, 2, 4, 5)
4:	(3, 5, 6)
5:	(3, 4, 6, 7)
6:	(4, 5)
7:	(5)

+ minimal memory requirement
+ vertices can easily be added or removed

– requires *O(λ)* time to determine whether a certain edge exists

Note: for weighted graphs store pairs of (neighbor label, edge weight)

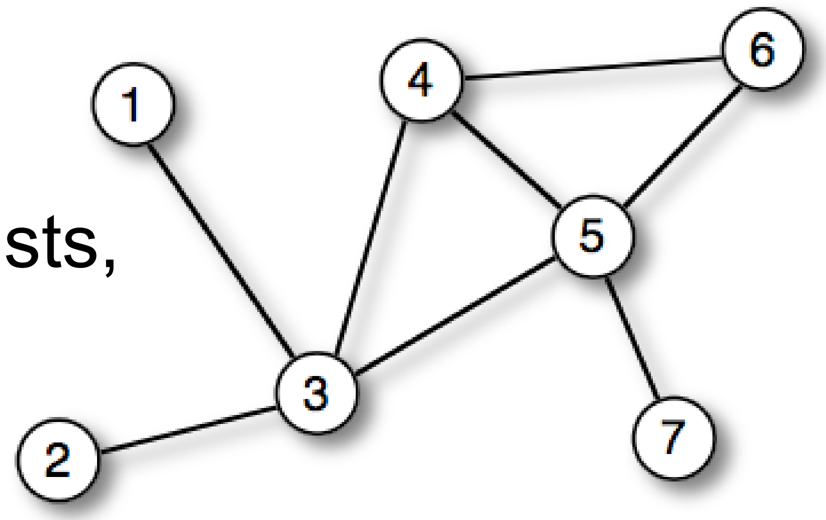
Graph Representation II

2. **Adjacency matrix**

$\rightarrow$ *N* x *N* matrix with entries M_{uv}

M_{uv} = weight when edge between *u* and *v* exists,
0 otherwise

$\rightarrow$ symmetric for undirected graphs



	1	2	3	4	5	6	7
1	–	0	1	0	0	0	0
2	0	–	1	0	0	0	0
3	1	1	–	1	1	0	0
4	0	0	1	–	1	1	0
5	0	0	1	1	–	1	1
6	0	0	0	1	1	–	0
7	0	0	0	0	1	0	–

+ fast *O(1)* lookup of edges

– large memory requirements

– adding or removing nodes is expensive

Note: very convenient in programming languages that support sparse multi-dimensional arrays
=> Perl

Graph Representation III

3. **Incidence matrix**

→ N x M matrix with entries M_{nm}

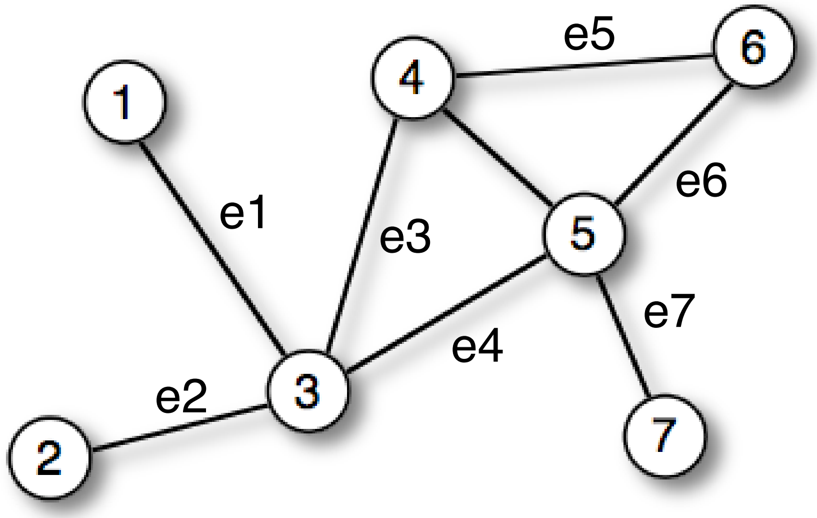
M_{nm} = weight when edge m ends at node n

0 otherwise

→ for a plain graph there are two entries per column

→ directed graph: indicate direction via sign (in/out)

The incidence matrix is a special form of the stoichiometric matrix of reaction networks.



	e1	e2	e3	e4	e5	e6	e7
1	1						
2		1					
3	1	1	1	1			
4			1		1		
5				1		1	1
6					1	1	
7							1

The Shortest Path Problem

Edsger Dijkstra (1930-2002):

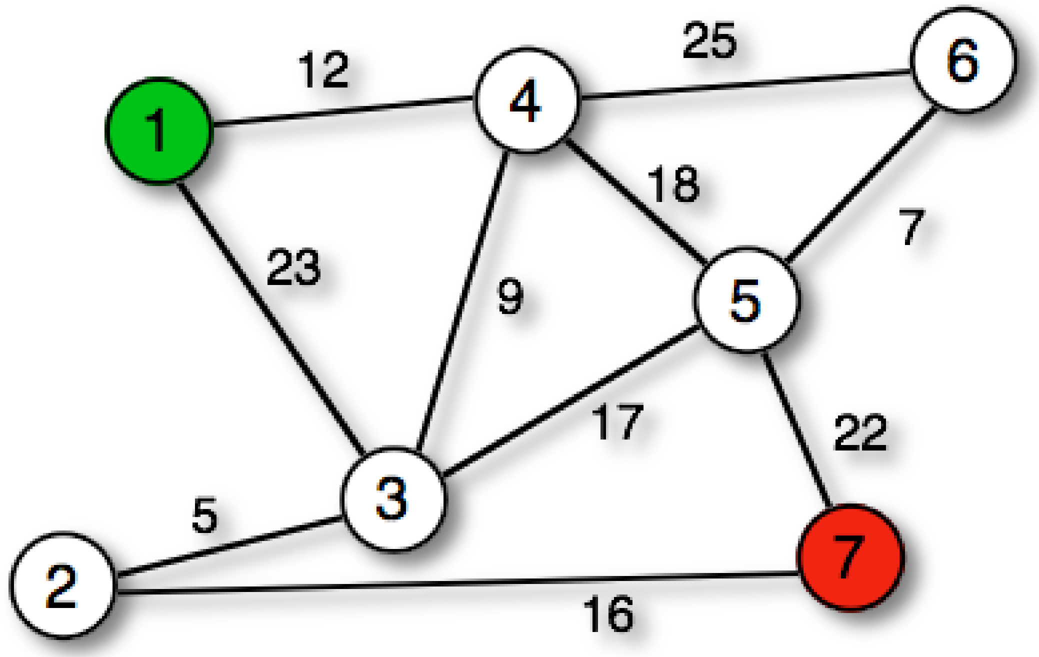
Problem:
Find the shortest path from a given vertex to the other vertices of the graph (Dijkstra 1959).

We need (input):
- weighted graph *G(V, E)*
- start (source) vertex *s* in *G*

We get (output):
- shortest distances *d[v]* between *s* and *v*
- shortest paths from *s* to *v*

Idea: Always proceed with the closest node
$\rightarrow$ greedy algorithm

Real world application:
$\rightarrow$ GPS navigation devices



Dijkstra Algorithm 0

Initialization:

```
for all nodes v in G:
    d[v] = oo
    pred[v] = nil
d[s] = 0
```

distance and path to all other nodes is still unknown

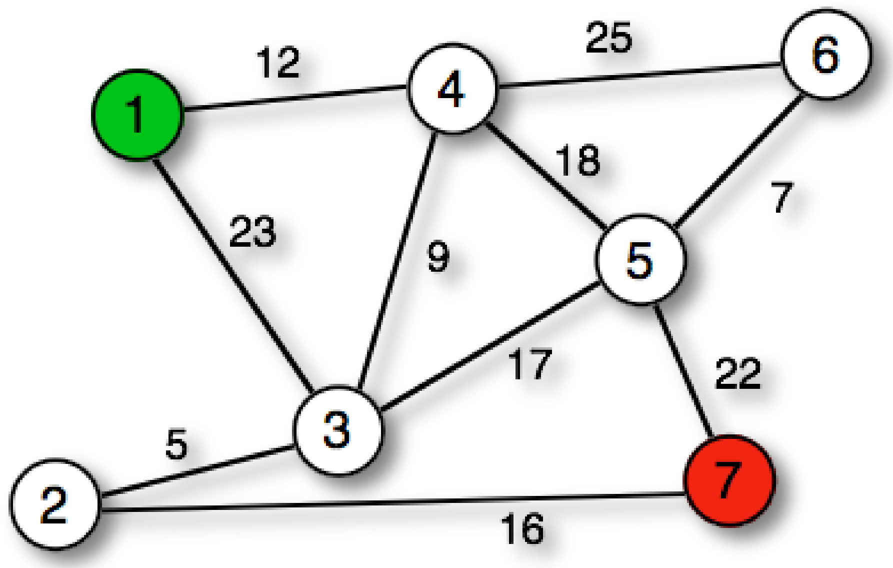
distance from source to source = 0

d[v] = length of path from *s* to *v*

pred[v] = predecessor node on the shortest path

In the example: *s* = 1

node	1	2	3	4	5	6	7
d	0	oo	oo	oo	oo	oo	oo
pred	–	–	–	–	–	–	–



Dijkstra I

Iteration:

```
Q = V
while Q is not empty:
   u = node with minimal d

   if d[u] = oo:
      break

   delete u from Q

   for each neighbor v of u:
      d_temp = d[u] + d(u,v)

      if d_temp < d[v]:
         d[v] = d_temp
         pred[v] = u

return pred[]C
```

Save {*V*} into working copy *Q*

choose node closest to *s*

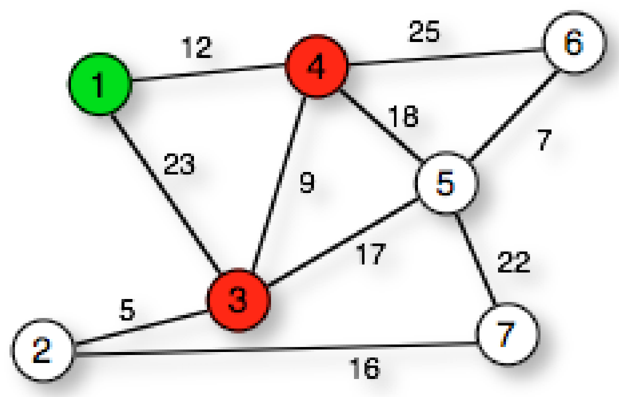
exit if all remaining nodes are inaccessible

calculate distance to *u*'s neighbors

if new path is shorter => update

Dijkstra-Example

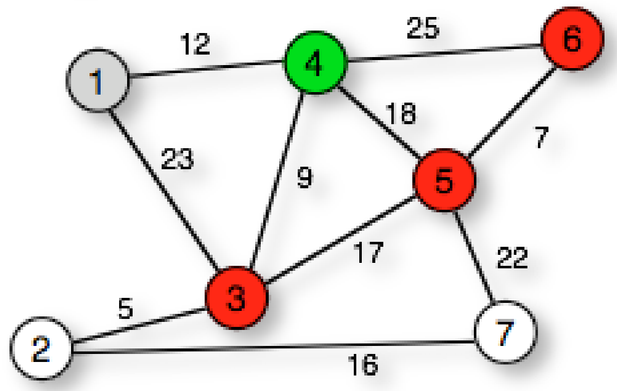
1)



Q = (**1**, 2, 3, 4, 5, 6, 7)

node	**1**	2	3	4	5	6	7
d	**0**	oo	23	12	oo	oo	oo
pred	–	–	1	1	–	–	–

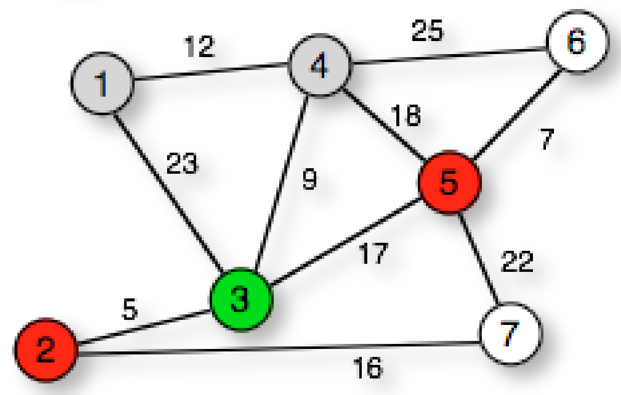
2)



Q = (2, 3, **4**, 5, 6, 7)

node	1	2	3	**4**	5	6	7
d	0	oo	21	**12**	30	37	oo
pred	–	–	4	**1**	4	4	–

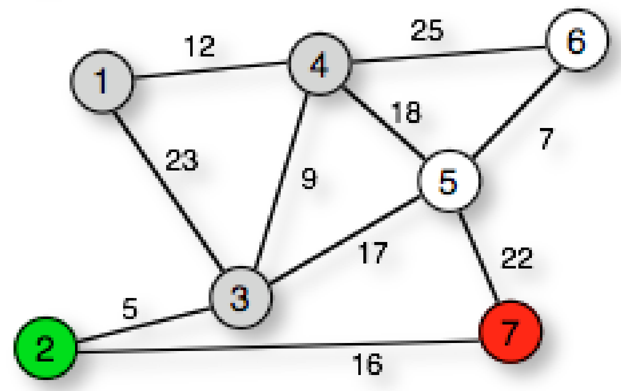
3)



Q = (2, **3**, 5, 6, 7)

node	1	2	**3**	4	5	6	7
d	0	26	**21**	12	30	37	oo
pred	–	3	**4**	1	4	4	–

4)



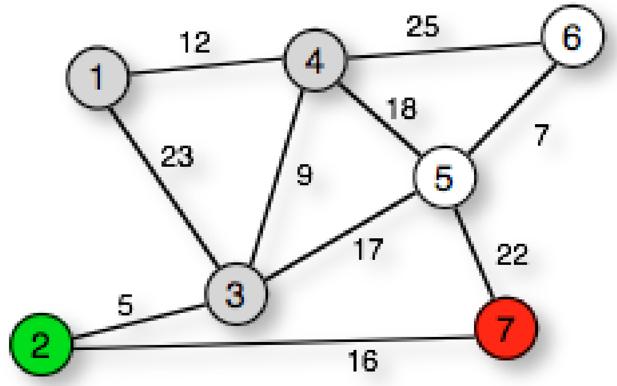
Q = (**2**, 5, 6, 7)

node	1	**2**	3	4	5	6	7
d	0	**26**	21	12	30	37	42
pred	–	**3**	4	1	4	4	2

```
Q = V
while Q is not empty:
    u = node with minimal d

    if d[u] = oo:
        break

    delete u from Q

    for each neighbor v of u:
        d_temp = d[u] + d(u,v)

        if d_temp < d[v]:
            d[v] = d_temp
            pred[v] = u

return pred[]C
```

Example contd.

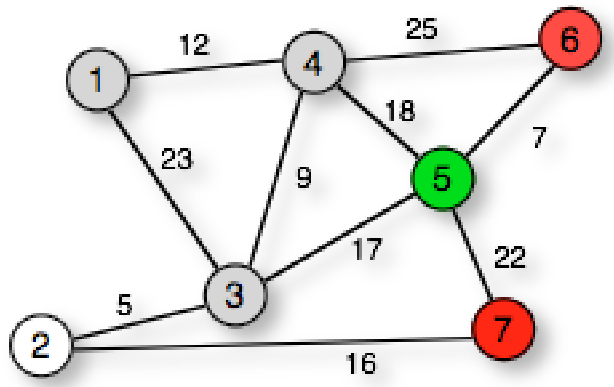
4)



Q = (**2**, 5, 6, 7)

node	1	**2**	3	4	5	6	7
d	0	**26**	21	12	30	37	42
pred	–	**3**	4	1	4	4	2

5)



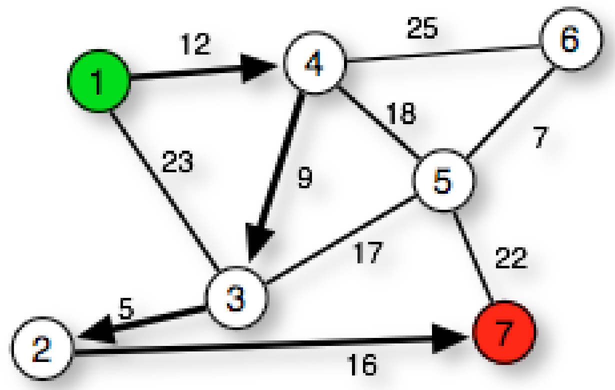
Q = (**5**, 6, 7)

node	1	2	3	4	**5**	6	7
d	0	26	21	12	**30**	37	42
pred	–	3	4	1	**4**	4	2

Q = (**6**, 7)

Q = (**7**)

Final result:



node	1	2	3	4	5	6	7
d	0	26	21	12	30	37	42
pred	–	3	4	1	4	4	2

d(1, 7) = 42 path = (1, 4, 3, 2, 7)

d(1, 6) = 37 path = (1, 4, 6) or (1,4,5,6)

Beyond Dijkstra

Dijkstra works for directed and undirected graphs with
non-negative weights.

Straight-forward implementation: $O(N^2)$

Graphs with positive and negative weights
$\rightarrow$ **Bellman-Ford**-algorithm

If there is a heuristic to estimate weights:
$\rightarrow$ improve efficiency of Dijkstra
$\rightarrow$ **A***-algorithm

Graph Layout

Task: visualize various interaction data:

e.g. **protein interaction data** (undirected):
- nodes – proteins
- edges – interactions

metabolic pathways (directed)
- nodes – substances
- edges – reactions

regulatory networks (directed):
- nodes – transcription factors + regulated proteins
- edges – regulatory interaction

co-localization (undirected)
- nodes – proteins
- edges – co-localization information

homology (undirected/directed)
- nodes – proteins
- edges – sequence similarity (BLAST score)

Graph Layout Algorithms

Graphs encapsulate relationship between objects
→ drawing gives **visual impression** of these relations

Good Graph Layout: **aesthetic**
• minimal edge crossing
• highlight symmetry (when present in the data)
• even spacing between the nodes

Many approaches in literature (and in software tools),
most useful ones usually NP-complete (exponential runtime)

Most popular for **straight-edge-drawing**:
→ **force-directed**: spring model or spring-electrical model
→ **embedding** algorithms like H3 or LGL

Force-Directed Layout

Peter Eades (1984): graph layout heuristic

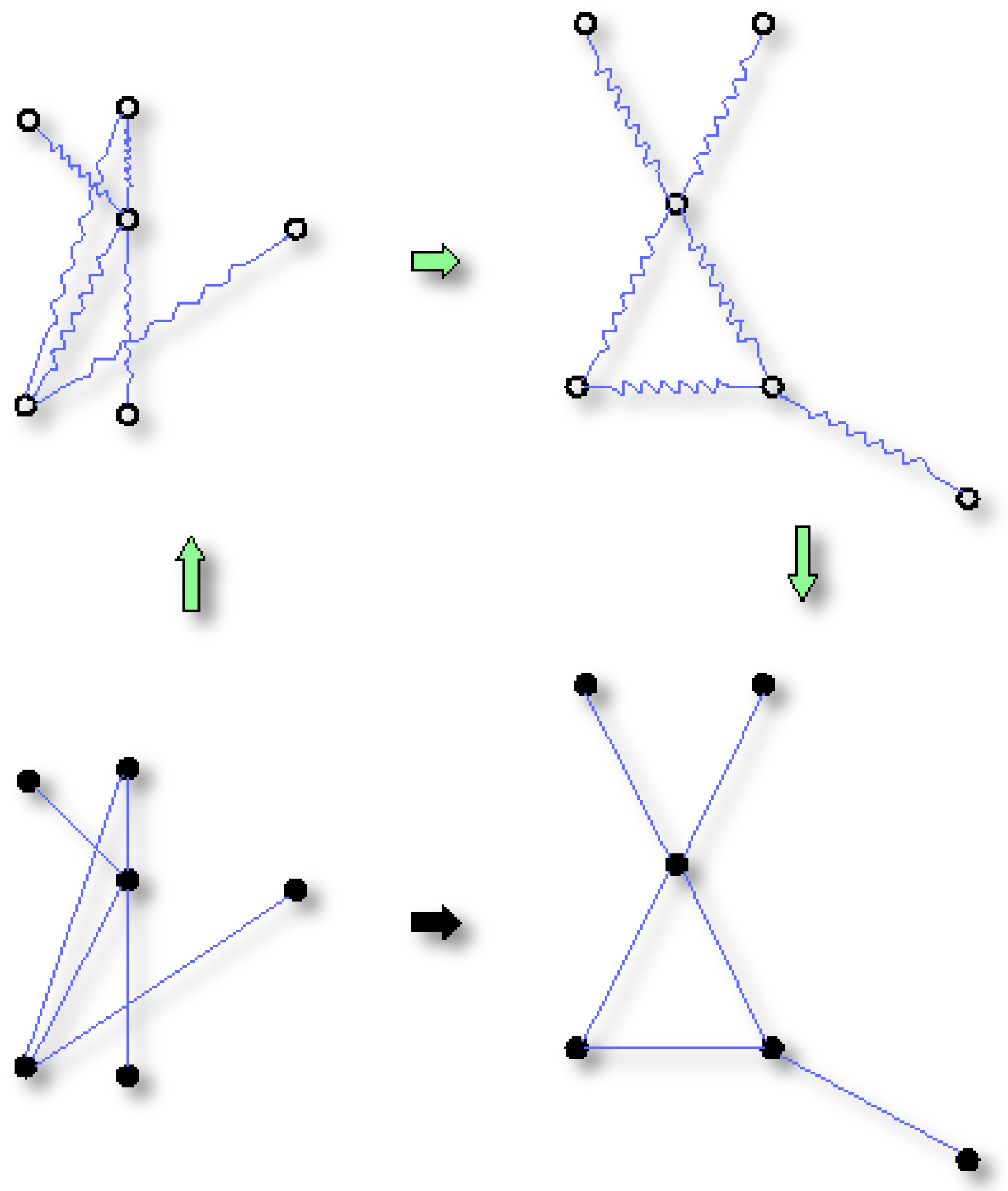
→ "**Spring Embedder**" algorithm.

- edges → springs
  vertices → rings that connect the springs

- Layout by dynamic relaxation

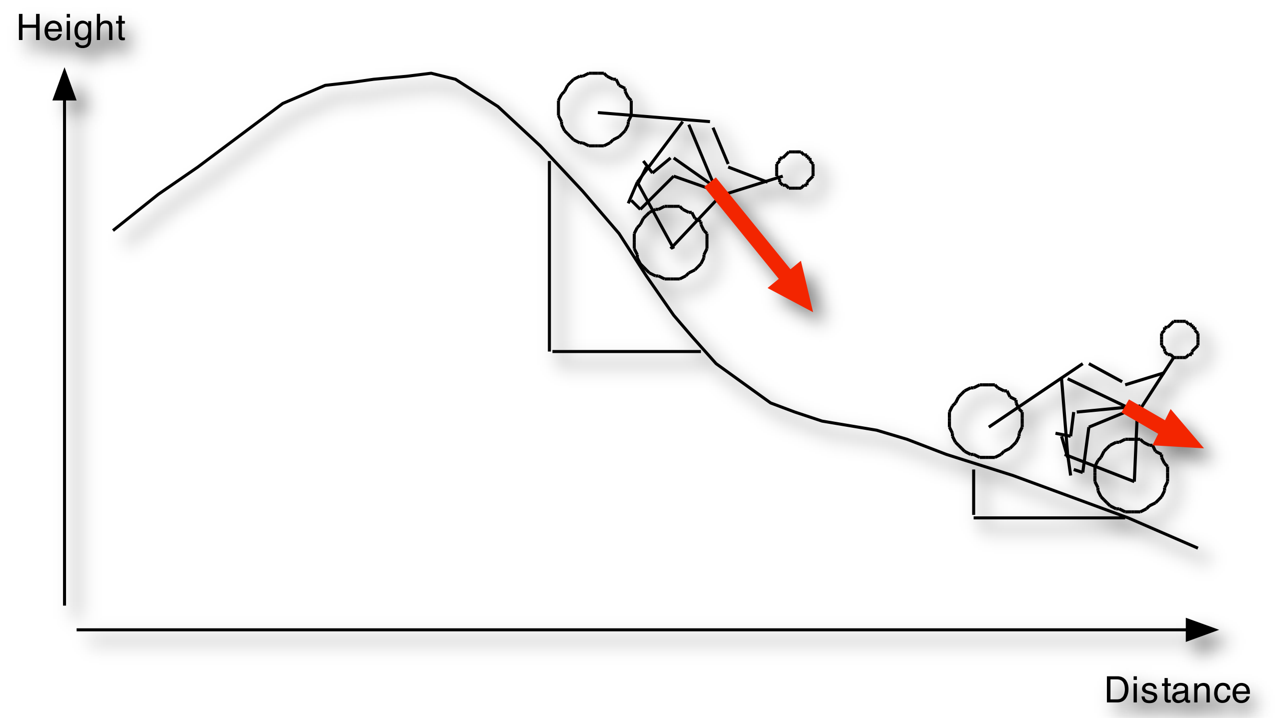
→ lowest-energy conformation

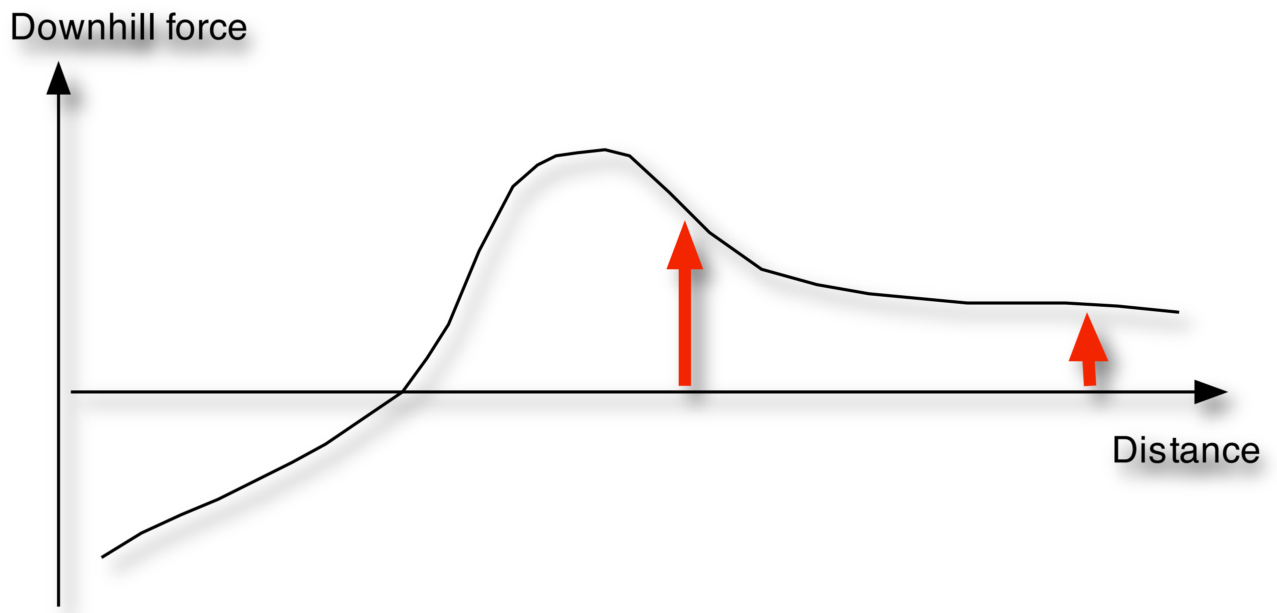
→ "**Force Directed**" algorithm



http://www.hpc.unm.edu/~sunls/research/treelayout/node1.html

Energy and Force





Energy: describes the altitude of the landscape

$$E(x) = mgh(x)$$

Energy increases when you go up the hill

$$\updownarrow$$

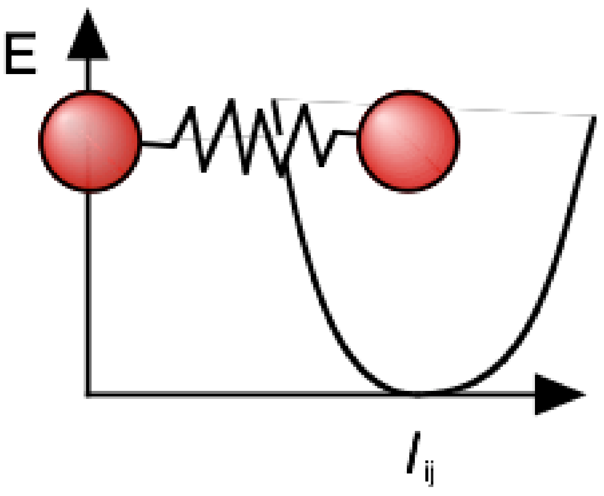
You need more force for a steeper ascent

$$F(x) = -\frac{dE(x)}{dx}$$

Force: describes the change of the altitude, points downwards.

Spring Embedder Layout

Springs regulate the mutual distance between the nodes
- too close $\rightarrow$ repulsive force
- too far $\rightarrow$ attractive force



Spring embedder algorithm:
- add springs for all edges
- add loose springs to all non-adjacent vertex pairs

Total energy of the system:

$$E = \sum_{i=1}^{|V|-1} \sum_{j=i+1}^{|V|} \frac{R}{l_{ij}^2} \left(|x_i - x_j| - l_{ij} \right)^2$$

x_i, x_j = position vectors for nodes *i* and *j*
l_{ij} = rest length of the spring between *i* and *j*
R = spring constant (stiffness)

Problem: l_{ij} have to be determined a priori, e.g., from network distance

Spring Model Layout

Task: find configuration of **minimal energy**

In 2D/3D: force = negative gradient of the energy

$$\vec{F}(\vec{x}) = -\nabla E(\vec{x}) = -\begin{pmatrix} \frac{\partial E}{\partial x} \\ \frac{\partial E}{\partial y} \\ \frac{\partial E}{\partial z} \end{pmatrix}$$

$\rightarrow$ Iteratively **move** nodes "**downhill**" along the gradient of the energy

$\rightarrow$ displace nodes **proportional** to the **force** acting on them

Problems:

• local minima

• a priori knowledge of all spring lengths

$\rightarrow$ works best for regular grids

The Spring-Electrical-Model

More general model than spring embedder model: use two types of forces

1) **attractive harmonic** force between connected nodes (springs)

$$F_{ij}^{h} = -k\,|r_i - r_j|$$

one uses usually the same spring constant *k* for all edges

2) **repulsive Coulomb**-like force between all nodes
"all nodes have like charges" $\rightarrow$ repulsion

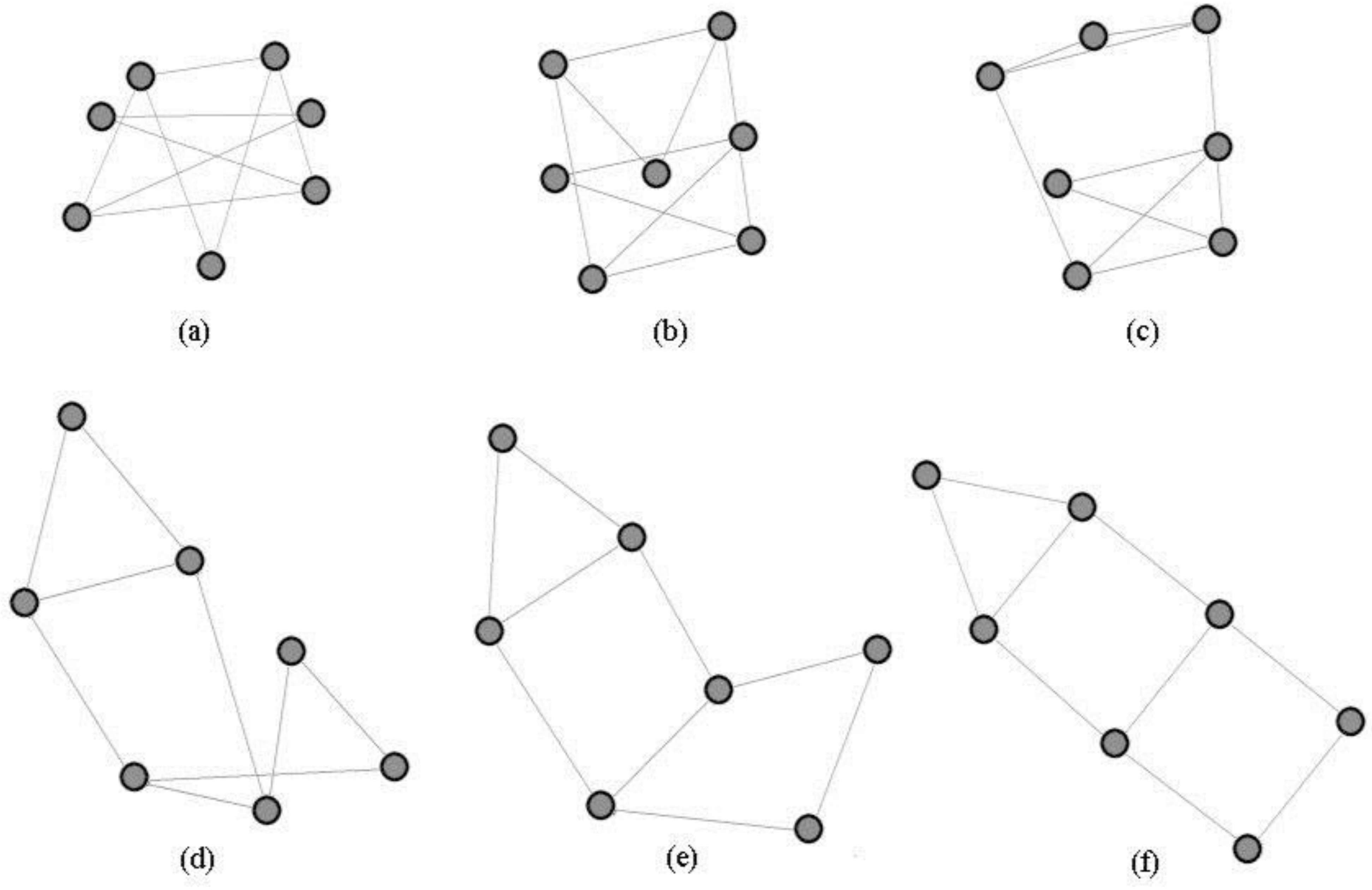
$$F_{ij}^{c} = \frac{Q_{ij}}{|r_i - r_j|^2}$$

either $Q_{ij} = Q$ or, e.g., $Q_{ij} = k_i\, k_j$

Repulsion pushes all nodes apart, springs pull connected nodes together
$\rightarrow$ **workhorse method** for small to medium sized graphs

$\rightarrow$ Do-it-yourself in Assignment 2 <=

Spring-Electrical Example



http://www.it.usyd.edu.au/~aquigley/3dfade/

Force-Directed Layout: Summary

Analogy to a **physical** system
=> force directed layout methods tend to meet various **aesthetic** standards:

- efficient **space filling**,
- **uniform** edge length (with equal weights and repulsions)
- **symmetry**
- smooth **animation** of the layout process (visual continuity)

Force directed graph layout → the "**work horse**" of layout algorithms.

Not so nice: the **initial random placement** of nodes and even very small changes of layout parameters will lead to **different representations**.
(no unique solution)

Side-effect: vertices at the periphery tend to be closer to each other than those in the center…

Runtime Scaling

Force directed layout:

```
loop until convergence:
    calculate forces:
        $L$ springs
        $N(N-1)/2$ charge pairs
    move vertices
    output positions
```

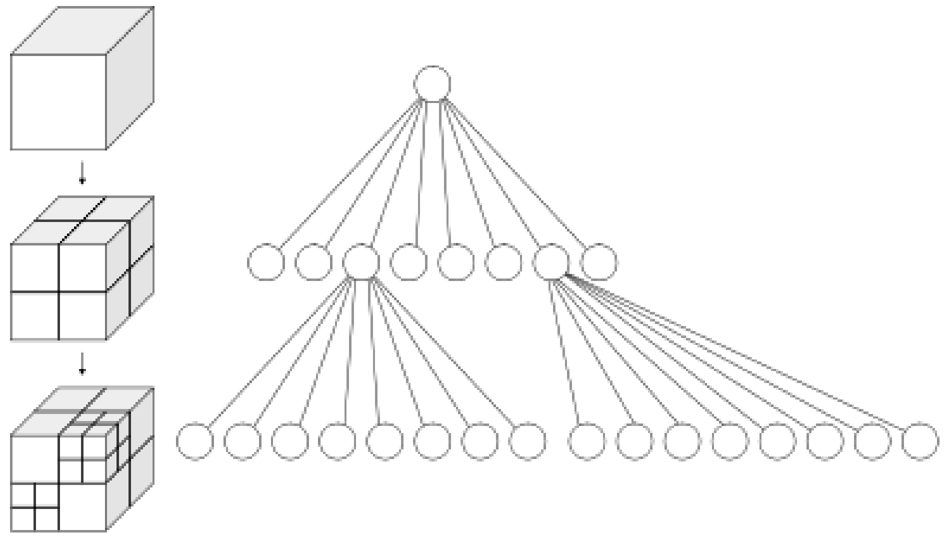
Several possible arrangements!!! (local minima)

$O(N^2)$!!!

$\rightarrow$ force directed layout suitable for small to medium graphs ($\leq$ O(1000) nodes?)

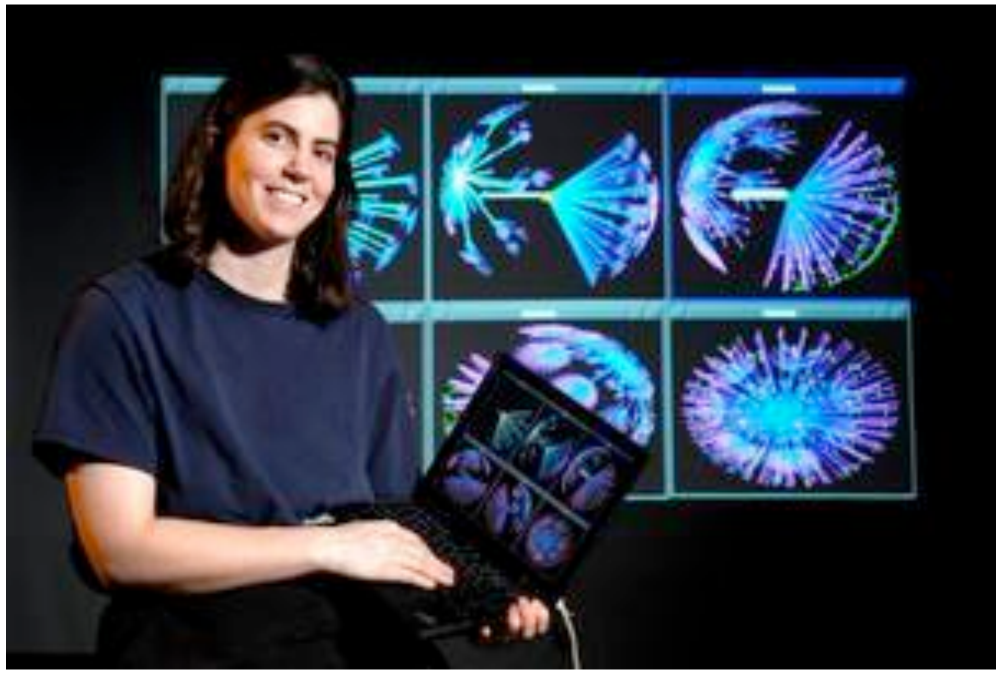
Speed up layout by:

- **multi-level** techniques to overcome local minima
- **clustering** (octree) methods for distant groups of nodes $\rightarrow$ *O(N log N)*



H3 Algorithm

Two problems of force directed layout:
- runtime scaling
- 2D space for drawing the graph



Tamara Munzner (1996-1998): H3 algorithm

$\rightarrow$ **interactively** visualize **large data sets** of $<$100.000 nodes.

- focusses on **quasi-hierarchical** graphs

  $\rightarrow$ use a **spanning tree** as the backbone of a layout algorithm
- graph layout in **exponential space** (projected on 2D for interactive viewing)

Spanning tree: connected acyclic subgraph that contains all the vertices of the original graph, but does not have to include all the links

$\rightarrow$ find a minimum-weight spanning tree through a graph with weighted edges, where **domain-specific information** is used to compute the **weights**

Spanning Tree

Some algorithms work only/better on trees

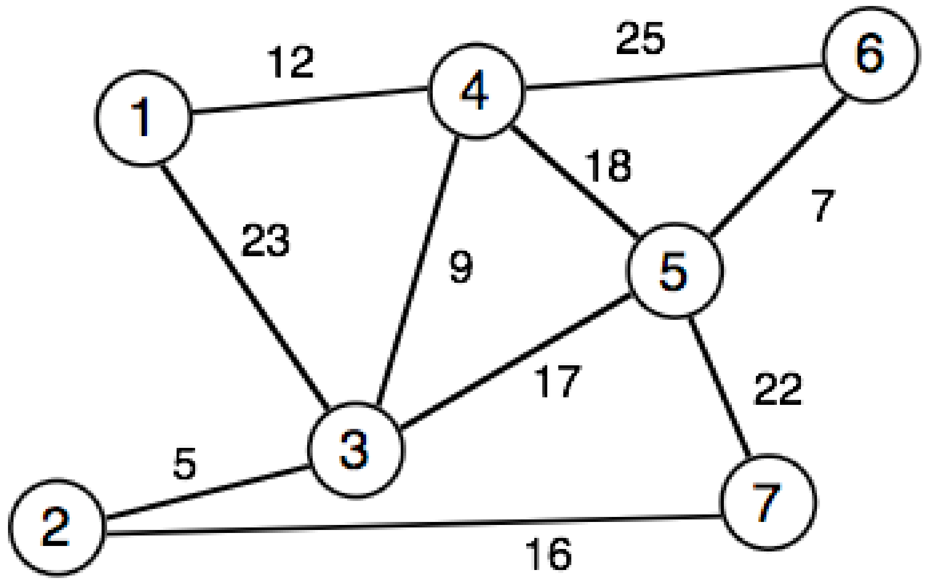
Idea: remove links until graph has tree structure, keep all nodes connected
$\rightarrow$ spanning tree

Minimal spanning tree = spanning tree with the least total weight of the edges

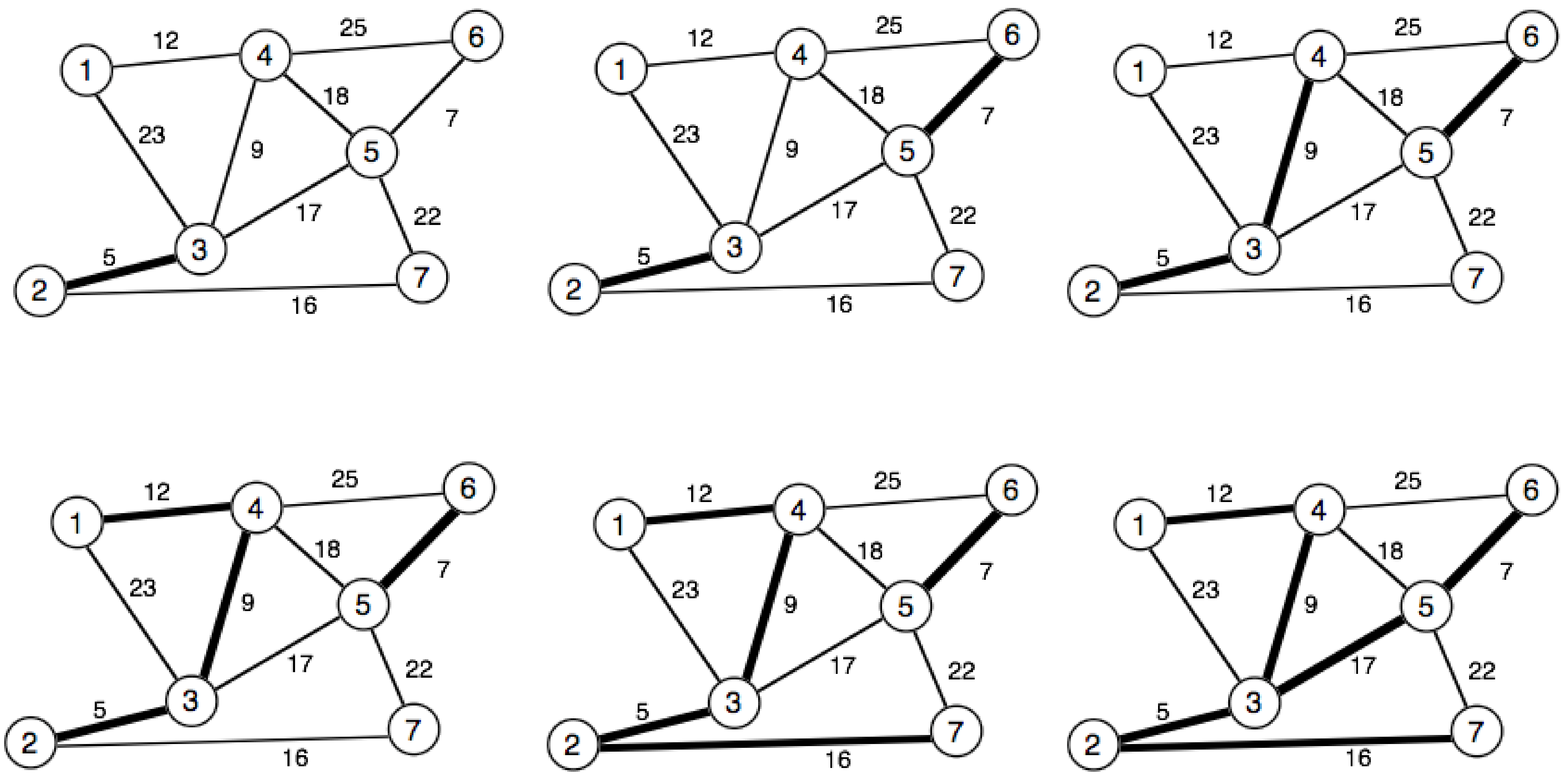
Greedy **Kruskal**-Algorithm:

$\rightarrow$ **iteratively** choose unused edge
with **smallest weight**,
if it does not lead to a **circle**!

greedy <=> base choice on current state,
(locally optimal choice)



Kruskal - Example

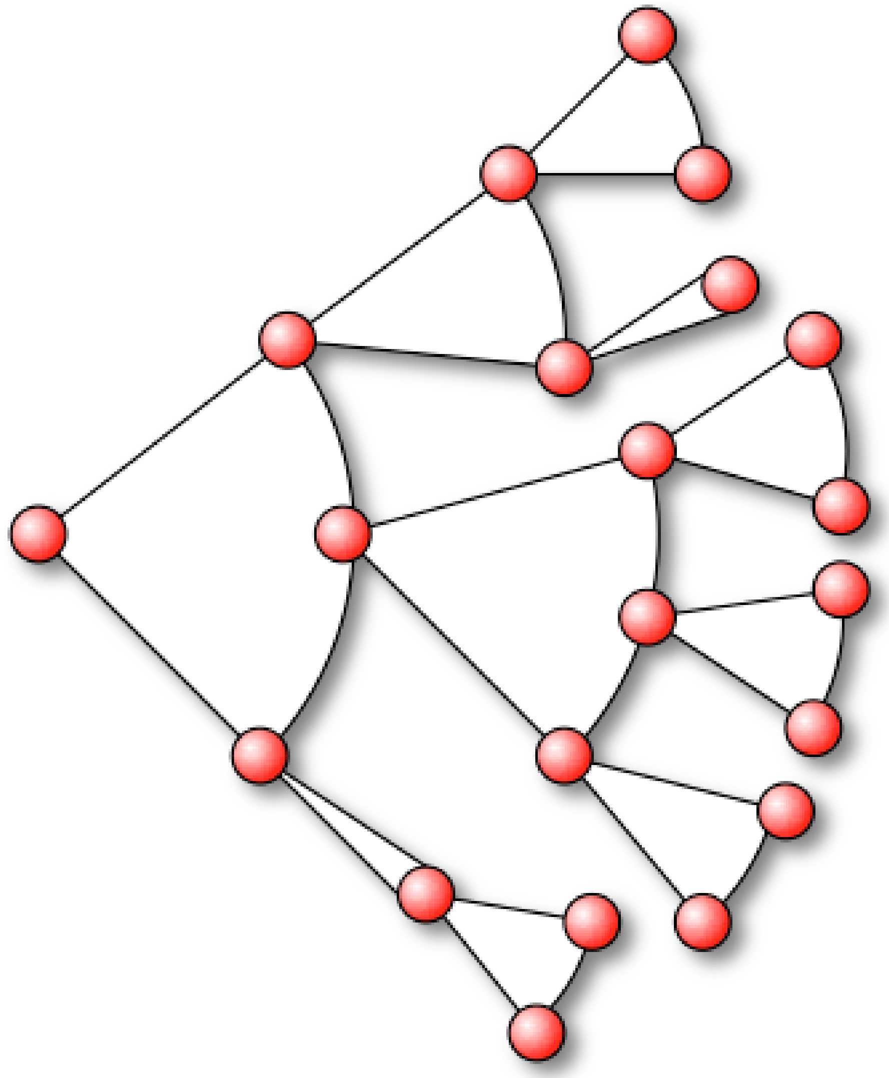


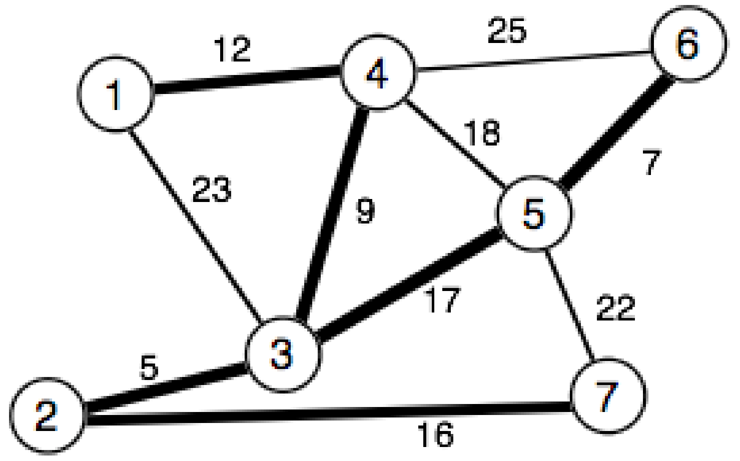
Proof that there is no spanning tree with a **lower** weight?

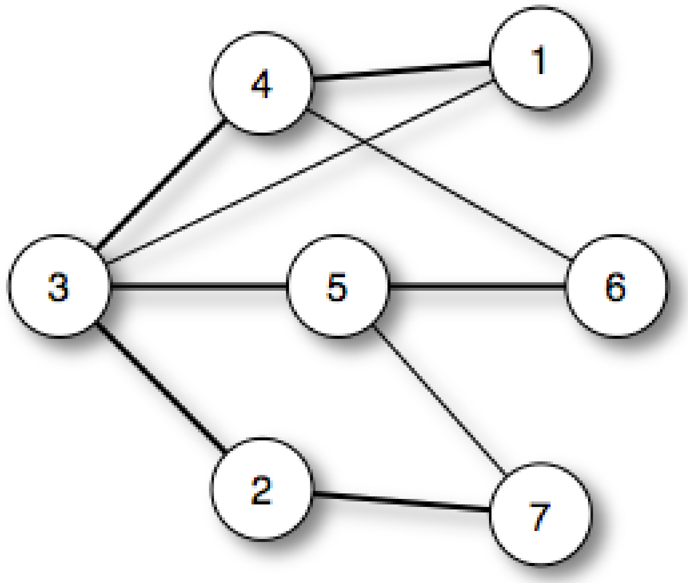
Minimum spanning tree weight = 66

Cone Layout

Place the nodes according to their hierarchy starting from the root node
→ direction indicates lineage







For arbitrary graphs
→ how to get weights?
→ which node is the root?

Exponential Room

In Euklidian space: circumference of a circle grows linear:

$$U = 2\pi r$$

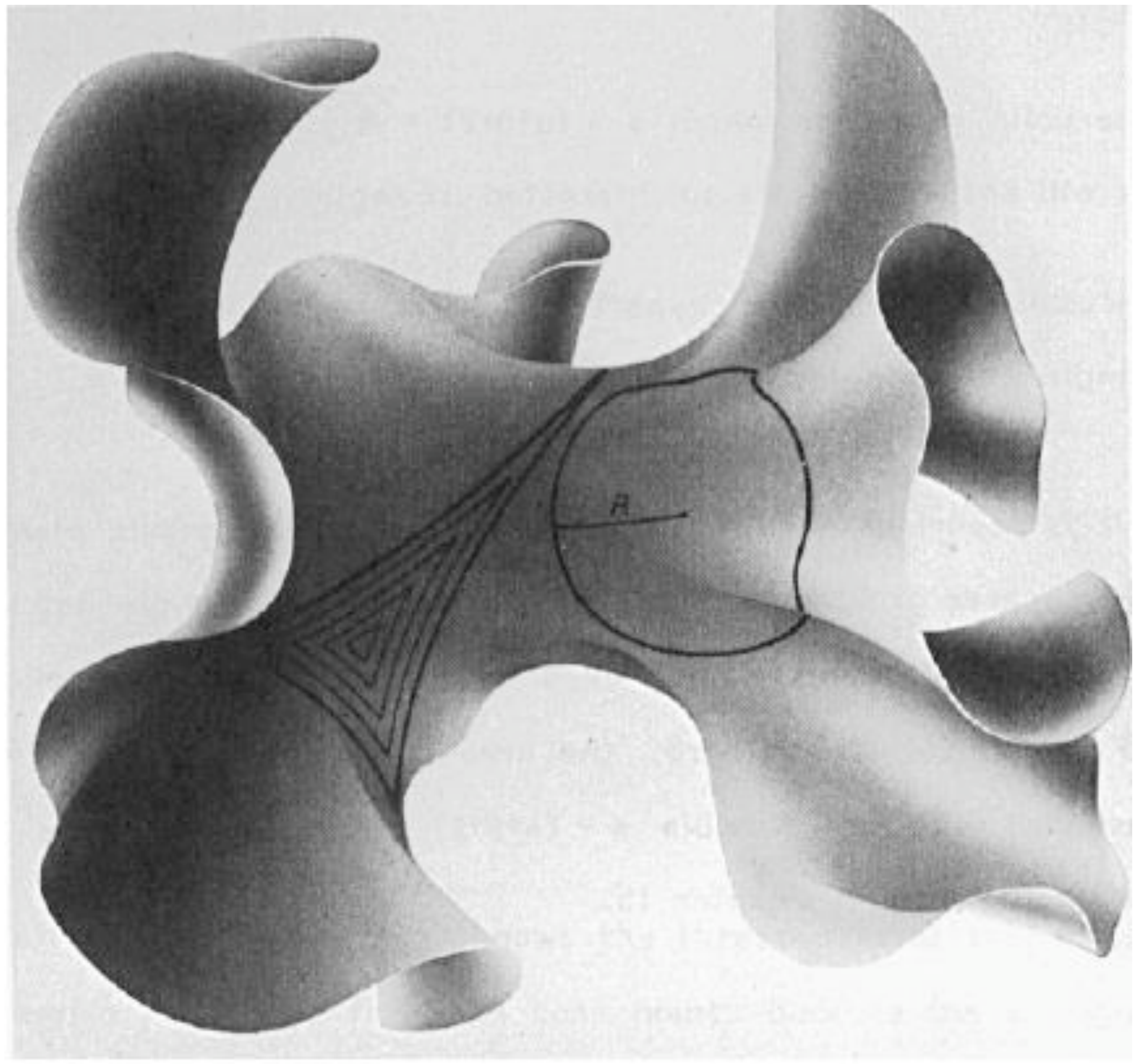
In hyperbolic space:

$$U = 2\pi \sinh r$$

$\rightarrow$ exponentially **growing** space
on the circle

For (cone) graph layout
$\rightarrow$ there is **enough room**
for yet another level

Also: **mappings** of the
complete hyperbolic space
$\rightarrow$ finite volume of Euklidian space



PhD thesis Tamara Munzner, chapter 3

Models of hyperbolic space

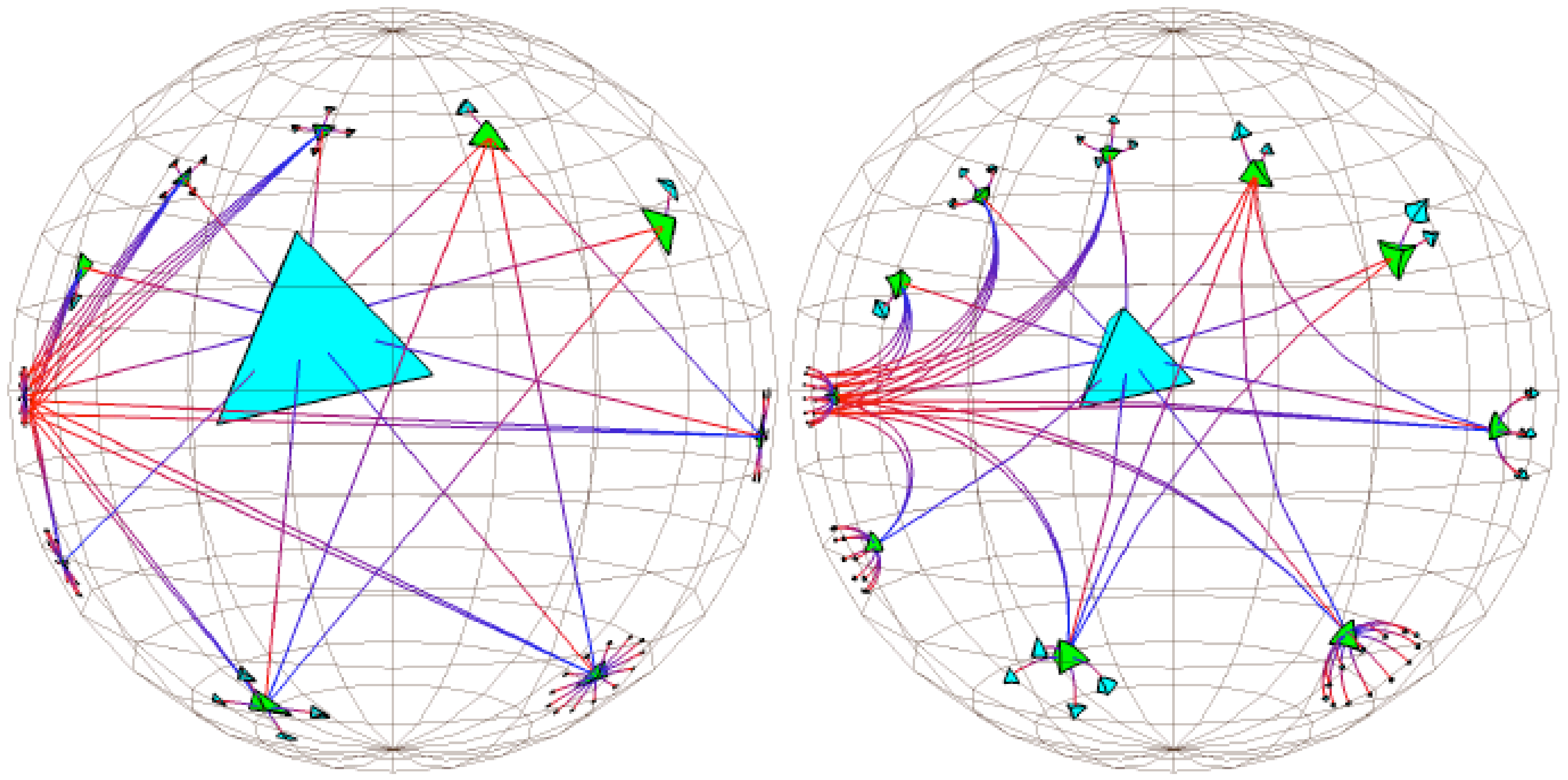
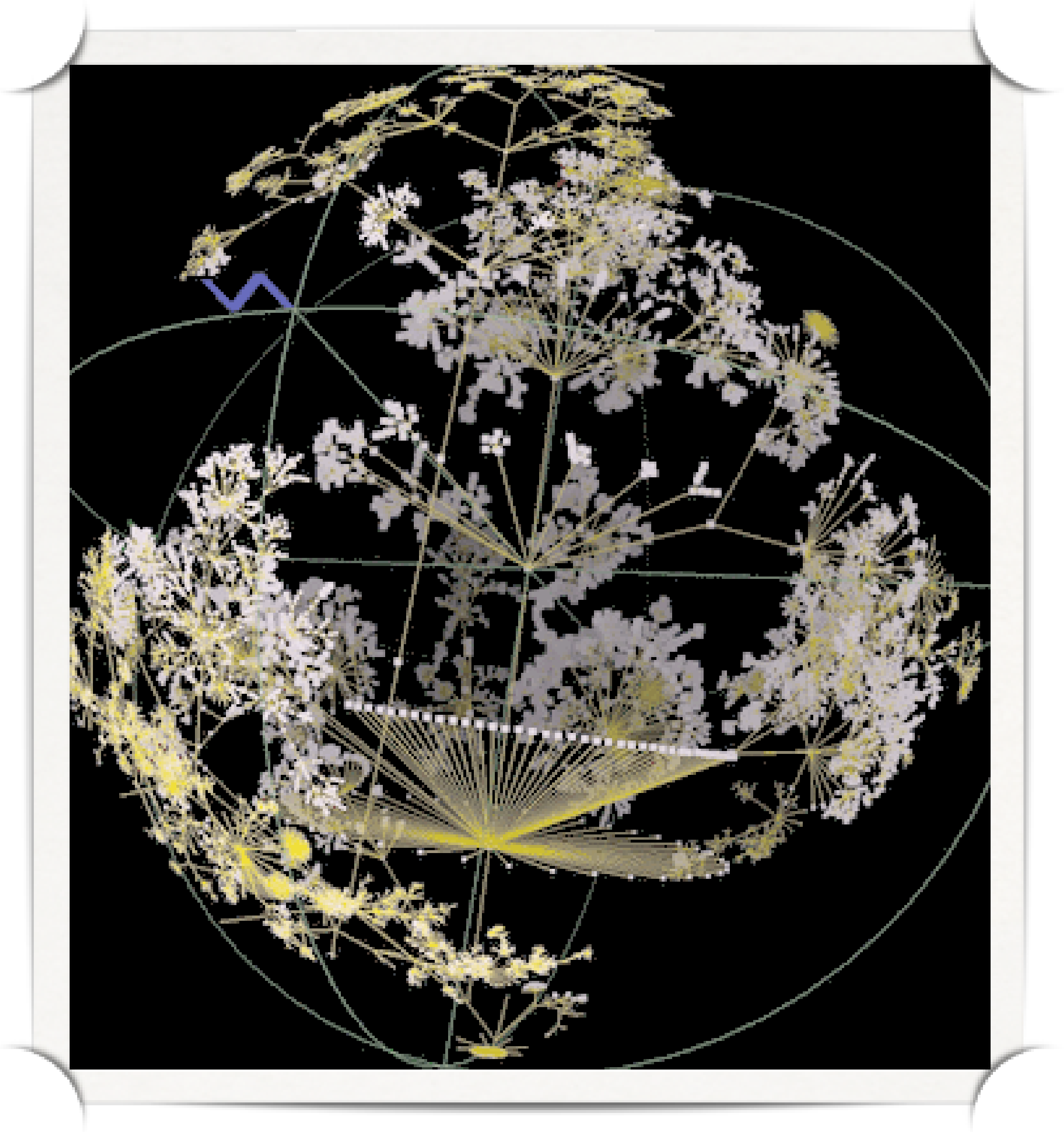
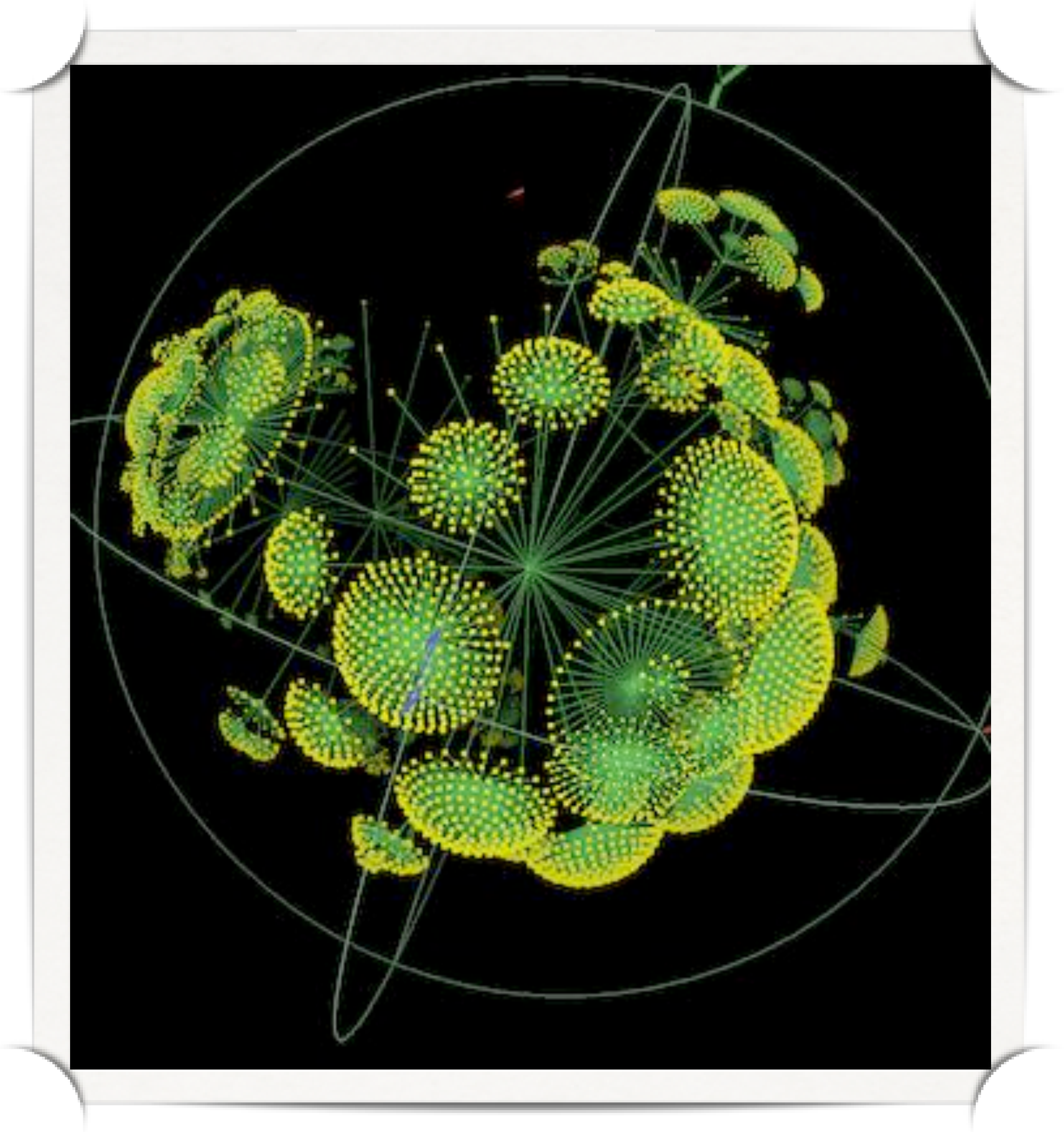


Figure 3.5: **Models of hyperbolic space. Left:** The projective model of hyperbolic space, which keeps lines straight but distorts angles. **Right:** The conformal model of hyperbolic space, which preserves angles but maps straight lines to circular arcs. These images were created with the *webviz* system from the Geometry Center [MB95], a first attempt to extend cone tree layouts to 3D hyperbolic space that had low information density. The cone angle has been widened to 180°, resulting in flat discs that are obvious in the projective view. The arcs visible in conformal view are actually distorted straight lines.

PhD thesis Tamara Munzner, chapter 3

GIFs don't work here…





http://www.caida.org/tools/visualization/walrus/gallery1/

H3: + layout based on MST $\rightarrow$ fast
+ layout in hyperbolic space $\rightarrow$ enough room
– how to get the MST for biological graphs????

Summary

What you learned **today**:

→ Local connectivity: clustering

→ random graphs vs. scale-free graphs

→ shortest path: Dijkstra algorithm

→ graph layout: force-directed and embedding schemes

→ spanning tree: Kruskal algorithm

Next lecture:

→ biological data to build networks from