

Bioinformatics 3

V 2 – Clusters, Dijkstra, and Graph Layout

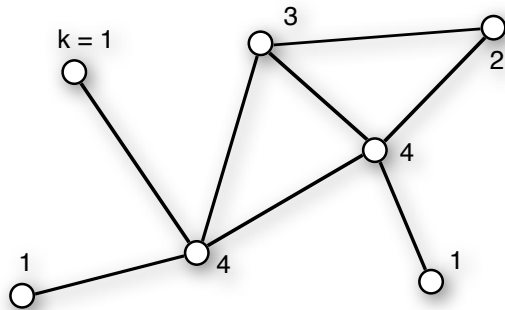
Fri, Oct 21, 2011

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 "Erdős-Renyi model"

start from all nodes, add links randomly

$P(k)$ = "Poisson"

Scale-free network:

grow with preferential attachment

$P(k)$ = power law

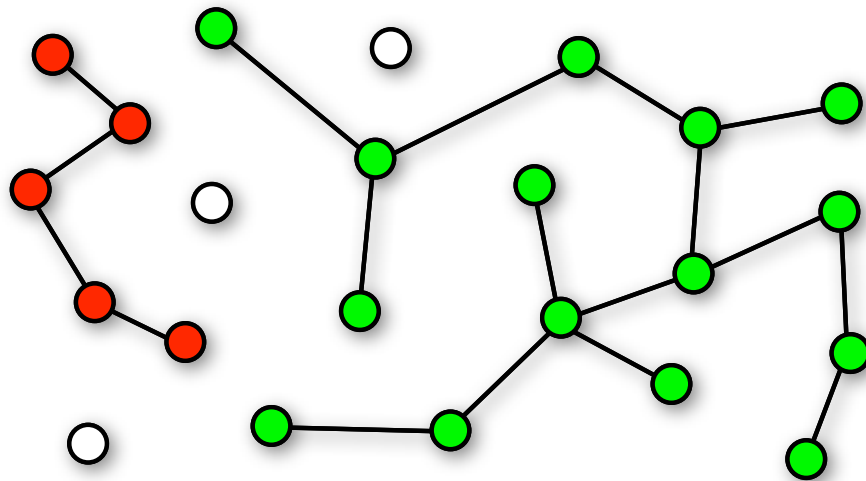
Connected Components

Connected graph \Leftrightarrow there is a path between all pairs of nodes

Large (random) networks: complete $\{V\}$ often not connected

\Rightarrow identify connected sets $\{V_i\}$ with $\{V\} = \cup \{V_i\}$

\Rightarrow **connected components**



#CC = 5

$N_{\max} = 15$

$N_{\min} = 1$

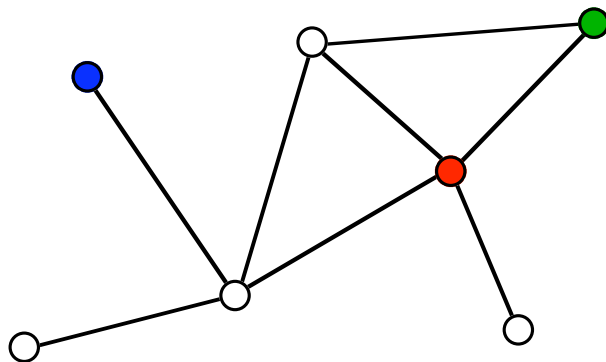
Connectivity of the Neighborhood

How many of the neighboring vertices are themselves neighbors?

=> clustering coefficient $C(k)$

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

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: $k = 4, n_k = 2 \Rightarrow C = 2/6$

blue: $k = 1, n_k = ? \Rightarrow C$ is not def.

Note: sometimes also defined via fraction of possible triangles

Cluster Coefficient of a Graph

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

Statistics:

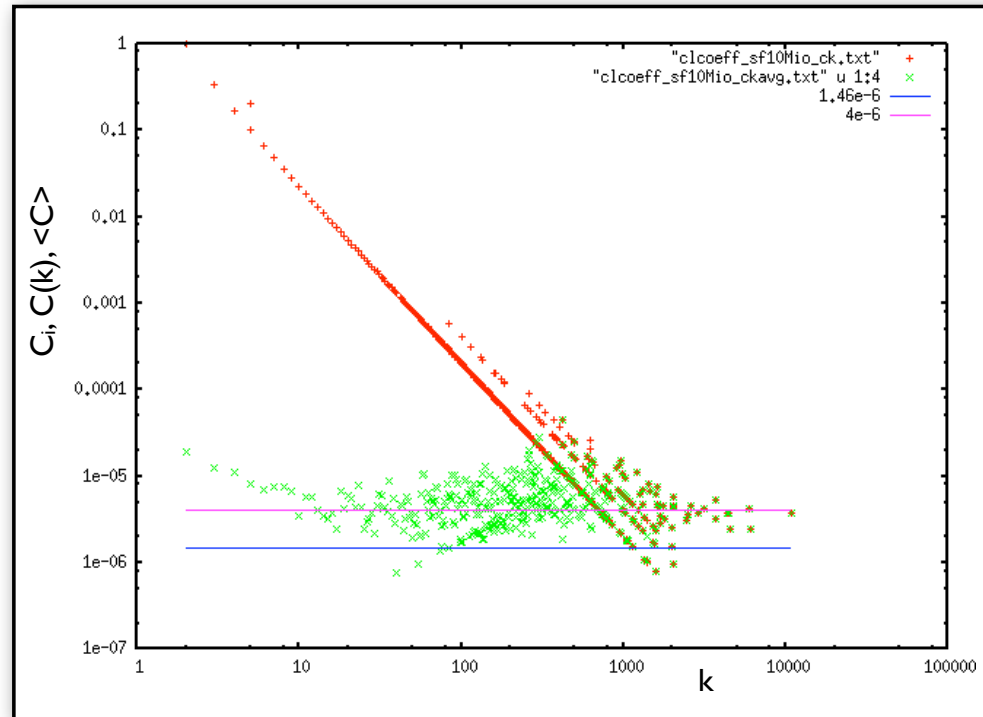
average at **fixed k**

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

average over **all nodes**

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

Note: it is also possible to
average the $C(k)$
 \Rightarrow different value for $\langle C \rangle$!!!



Can you explain the "structure" in the red data points?

C(k) for a Random Network

Clustercoefficient with m edges between k neighbors

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

Probability to have exactly m edges between the k neighbors

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

Average C(k) for degree k:

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

=> C(k) is independent of k

<=> same local connectivity throughout the network

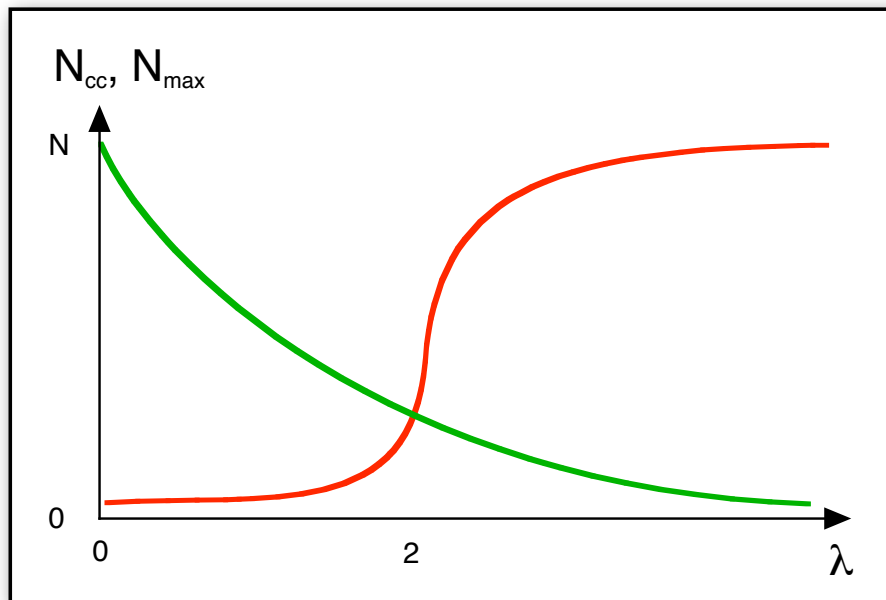
The Percolation Threshold

Connected component = all vertices that are connected by a path

Very few edges
=> only c.c. of size 2

← Percolation
transition at
 $\lambda = 2$ →

Very many edges
=> graph is one c.c.



Identify:

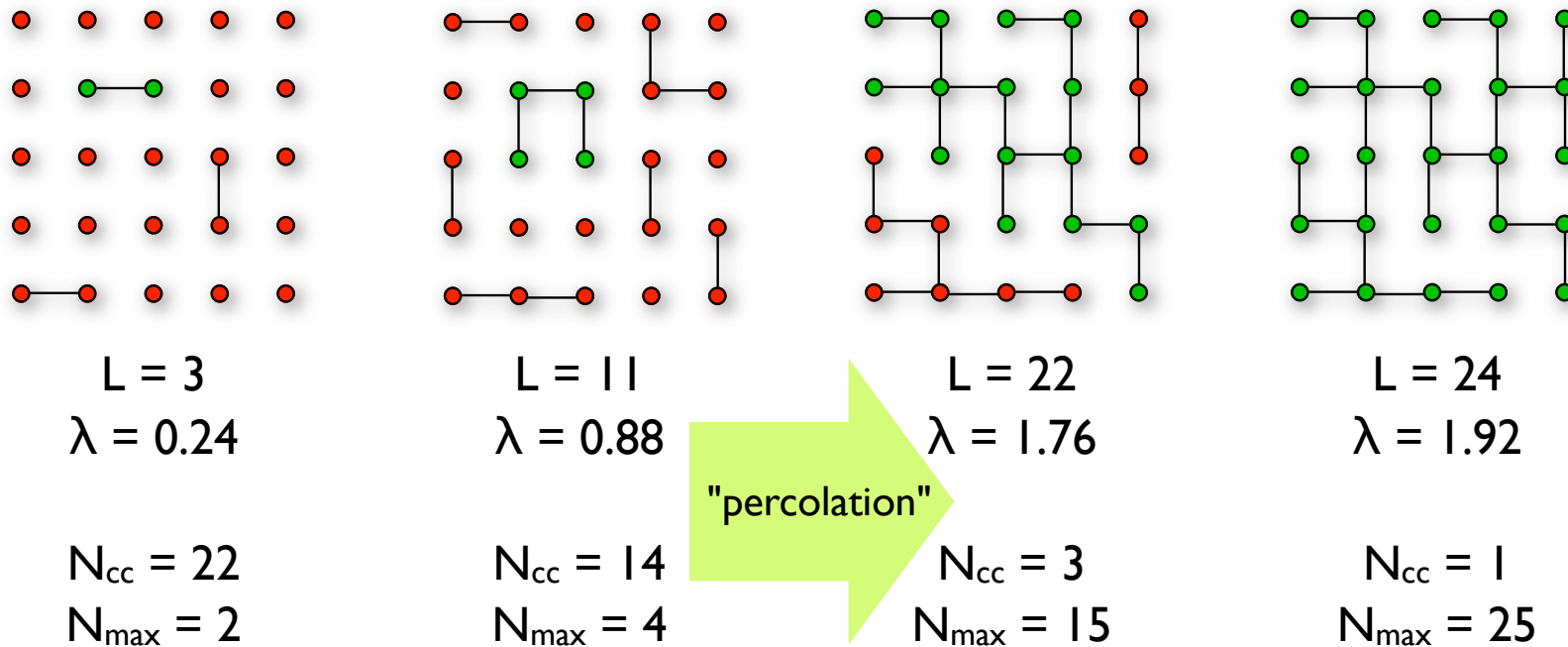
N_{cc} = number of connected components (clusters)

N_{max} = size of the largest cluster

For $\lambda > 2$: giant component

Percolation Transition

Example: regular square lattice, $N = 25$, $L_{\max} = 40$



percolation = "spanning structure" emerges (long range connectivity)

for an infinite square lattice: percolation transition at $\lambda = 2$

here: finite size effect \Leftrightarrow fewer possible links at the boundaries

Clusters in SF Graphs

Scale-free network \Leftrightarrow no intrinsic scale

\Rightarrow same properties at any k-level

\Rightarrow same local connectivity

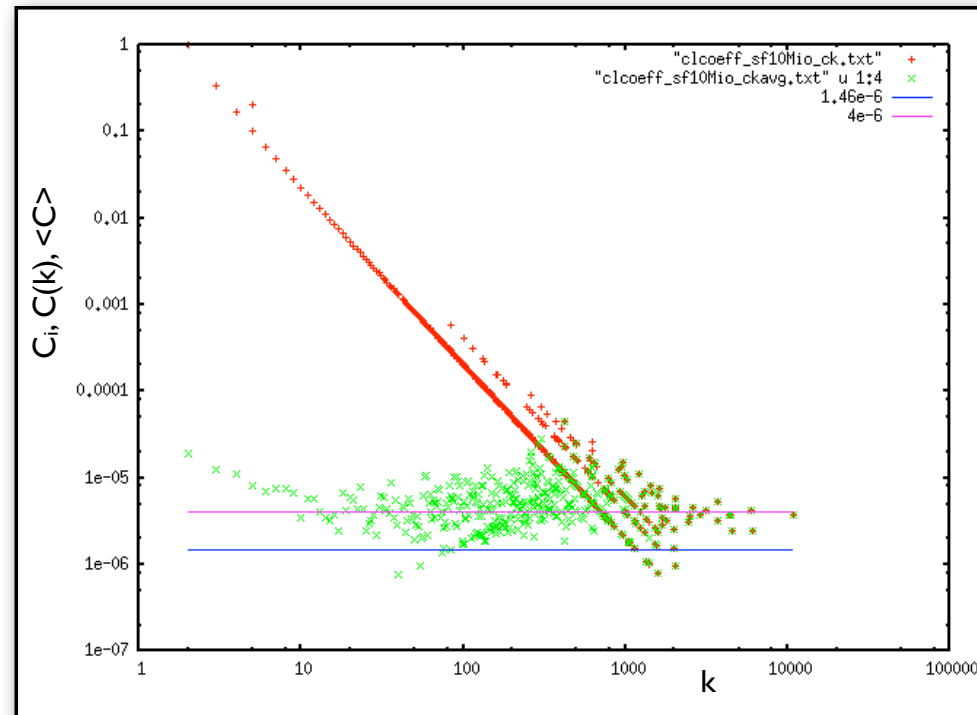
$\Rightarrow C(k) = \text{const.}$

"Real" biological data

\Rightarrow missing links

\Rightarrow multiple clusters

Is the metabolic
network of a cell
fully connected?



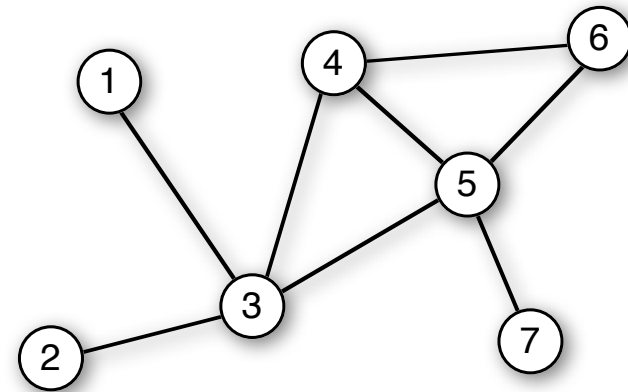
Algorithms on Graphs

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 $O(\lambda)$ 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

=> $N \times N$ matrix with entries M_{uv}

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

=> symmetric for undirected graphs

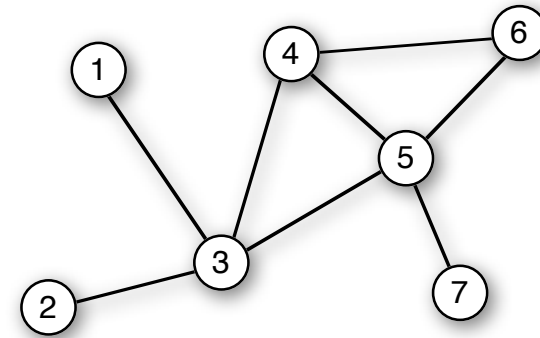
+ fast $O(1)$ lookup of edges

– large memory requirements

– adding or removing nodes is expensive

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

=> Perl



	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	–

Graph Representation III

3. Incidence matrix

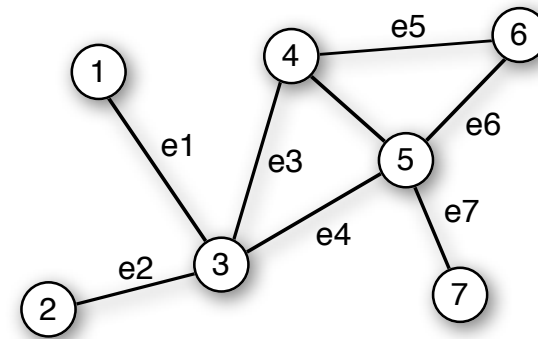
=> $N \times M$ matrix with entries M_{nm}

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: Find the shortest path from a chosen vertex to the other vertices of the graph.

We need (input):

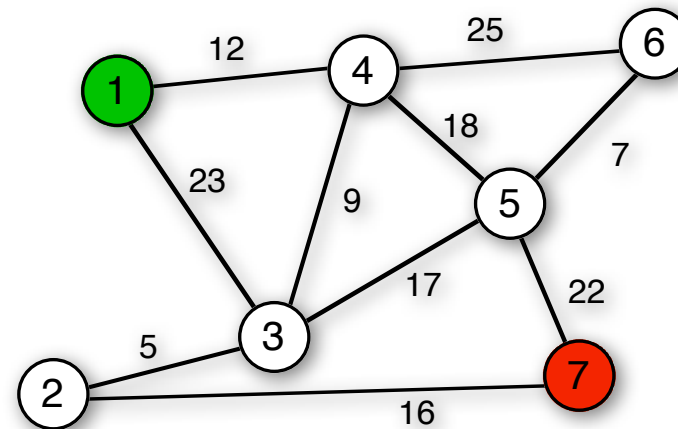
- weighted graph $G(V, E)$
- start (source) vertex s in G

We get (output):

- shortest distances $d[v]$ between s and v
- shortest paths from s to v

Idea: Always proceed with the closest node
=> greedy algorithm

Real world application:
=> GPS navigation devices



Dijkstra Algorithm 0

Initialization:

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

distance and path to all other nodes is still unknown

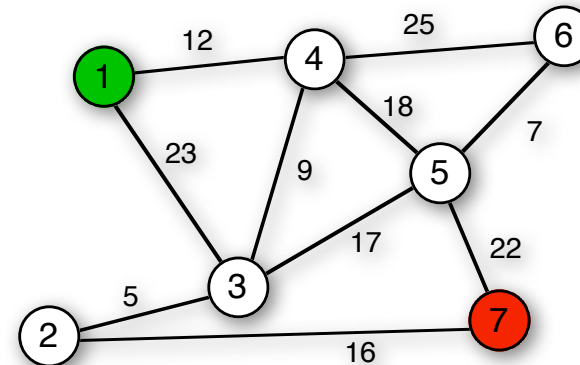
distance from source to source = 0

$d[v]$ = 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	∞	∞	∞	∞	∞	∞
pred	–	–	–	–	–	–	–



Dijkstra I

Iteration:

```
Q = V
while Q is not empty:
    u = node with minimal d
    if d[u] =  $\infty$ :
        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[]
```

Save {V} into working copy Q

choose node closest to s

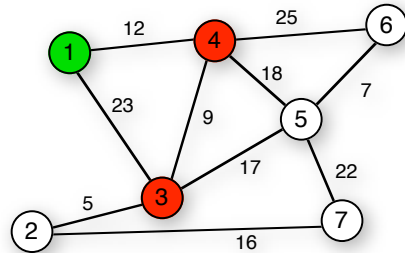
exit if all remaining nodes
are inaccessible

calculate distance to u's
neighbors

if new path is shorter
=> update

Dijkstra-Example

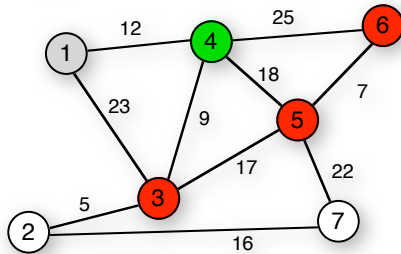
1)



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

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

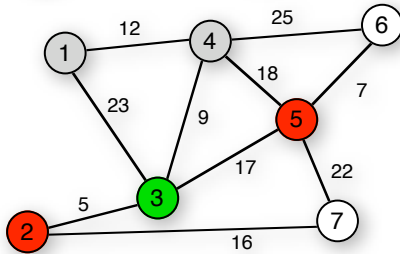
2)



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

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

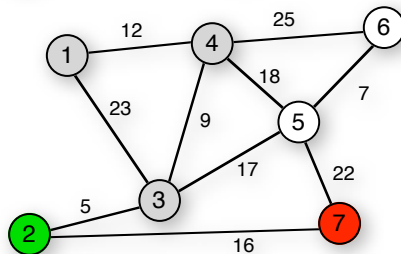
3)



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

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

4)

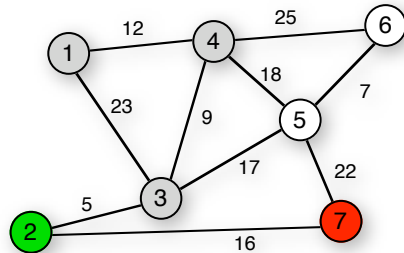


$Q = (2, 5, 6, 7)$

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

Example contd.

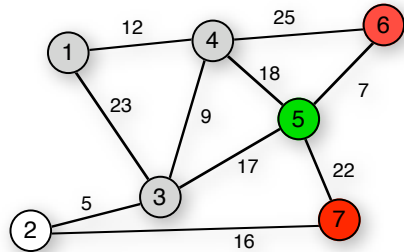
4)



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

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

5)



$Q = (5, 6, 7)$

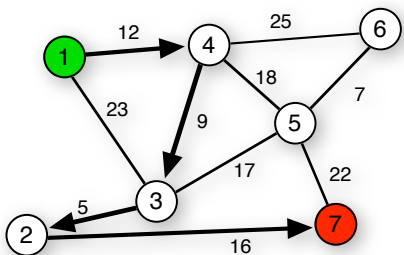
node	1	2	3	4	5	6	7
d	0	26	21	12	30	37	42
pred	–	3	4	1	4	4	2

$Q = (6, 7)$

$Q = (7)$

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

Final result:



$d(1, 7) = 42$

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
=> **Bellman-Ford**-algorithm

If there is a heuristic to estimate weights:
=> improve efficiency of Dijkstra
=> **A***-algorithm

Graph Layout

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** algorithm like H3 or LGL

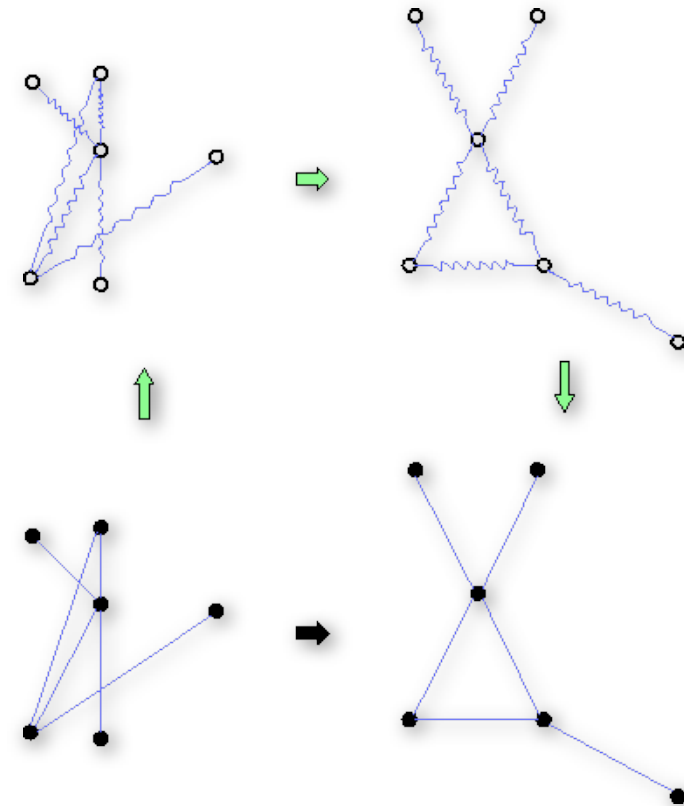
Force-Directed Layout

Peter Eades (1984): graph layout heuristic
=> "**Spring Embedder**" algorithm.

- edges => springs
vertices => rings that connect the springs

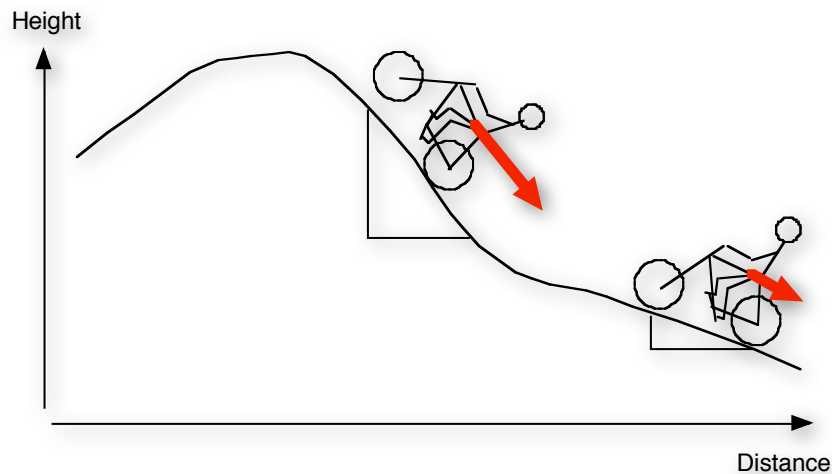
- Layout by dynamic relaxation
=> lowest-energy conformation

=> "**Force Directed**" algorithm



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

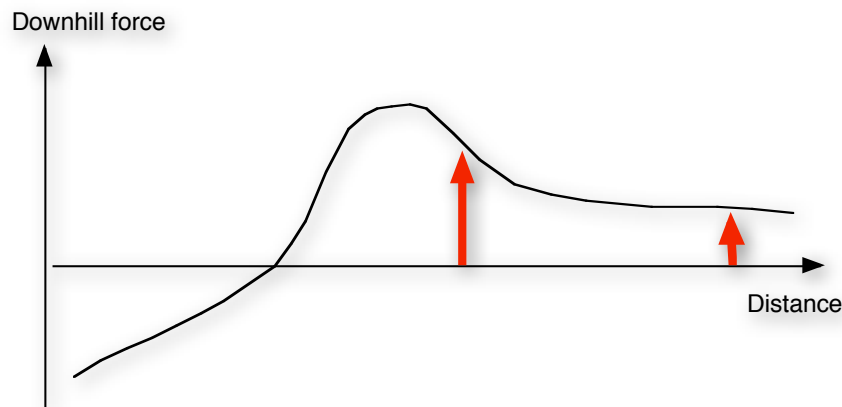
Energy and Force



Energy increases when you go up the hill

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

Energy: describes the altitude of the landscape



You need more force for a steeper ascent

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

Force: describes the change of the altitude, pointing downwards

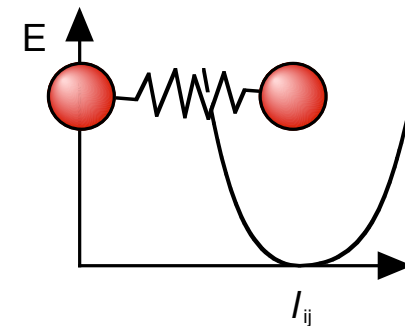
Spring Embedder Layout

Springs regulate the mutual distance between the nodes

- too close => repulsive force
- too far => attractive force

Spring embedder algorithm:

- add springs for all edges
- add loose springs to all non-adjacent vertex pairs



Total energy of the system:

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

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

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

=> Iteratively **move** nodes "**downhill**" along the gradient of the energy

=> displace nodes **proportional** to the **force** acting on them

Problems:

- local minima
- a priori knowledge of all spring lengths
 - => works best for regular grids

The Spring-Electrical-Model

More general than spring model: use two types of forces

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

$$F_{ij}^h = -k |r_i - r_j| \quad \text{usually the same spring constant } k \text{ for all edges}$$

2) **repulsive Coulomb**-like force between all nodes

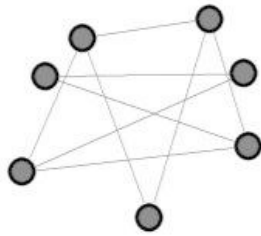
"all nodes have like charges" => repulsion

$$F_{ij}^c = \frac{Q_{ij}}{|r_i - r_j|^2} \quad \text{either } Q_{ij} = Q \text{ or, e.g., } Q_{ij} = k_i k_j$$

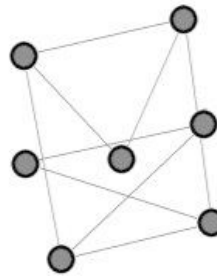
Repulsion pulls nodes apart, springs keep connected nodes together
=> **workhorse method** for small to medium sized graphs

=> Do-it-yourself in Assignment 2 <=

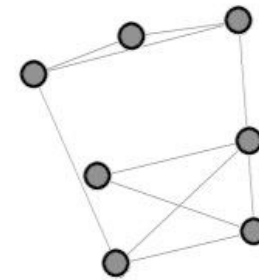
Spring-Electrical Example



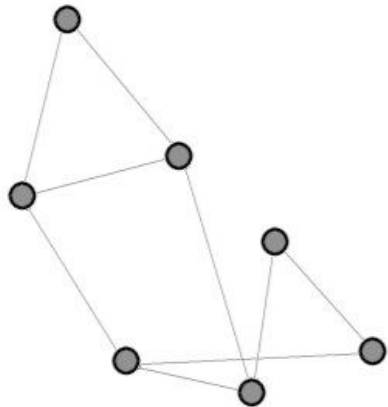
(a)



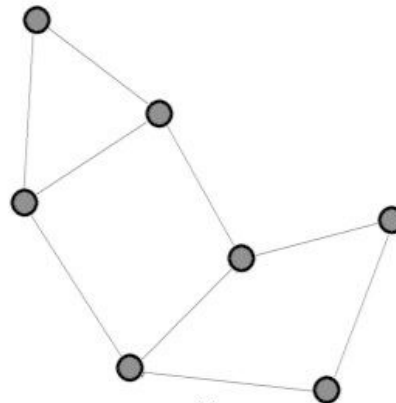
(b)



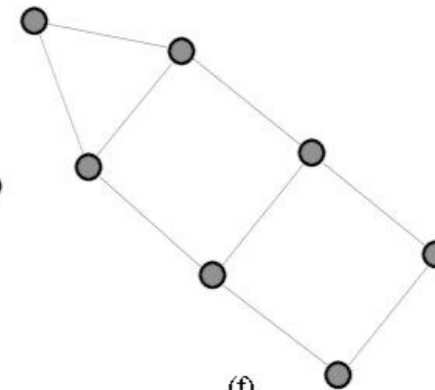
(c)



(d)



(e)



(f)

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

Force-Directed: 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
```

more possible
arrangements!!!
(local minima)

$O(N^2)$!!!

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

Speed up layout by:

- **multi-level** techniques to overcome local minima
- **clustering** (octree) methods for far apart groups of nodes => $O(N \log N)$

Spatial Scaling

random net: small world

$$\langle d \rangle \propto \log N$$

\Leftrightarrow

available area in 2D:

$$A \propto d^2$$

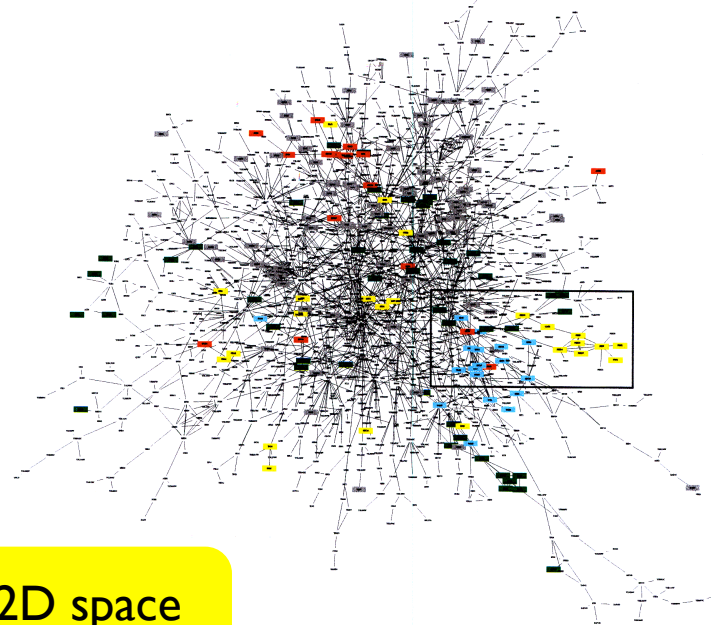
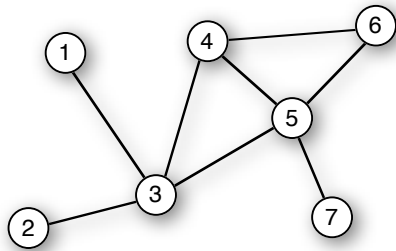
vertex density:

$$\rho = \frac{N}{A} \propto \frac{N}{2 \log N}$$

size of the layout-ed graph grows slower than the number of nodes

=> layout too cramped

(even worse for scale-free!)



=> mapping from network to 2D space

H3 Algorithm

Two problems of force directed layout:

- runtime scaling
- 2D space for drawing the graph



Tamara Munzner (1996-1998): H3 algorithm

=> **interactively** visualize **large data sets** of ~100.000 nodes.

- focusses on **quasi-hierarchical** graphs
=> 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

=> 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

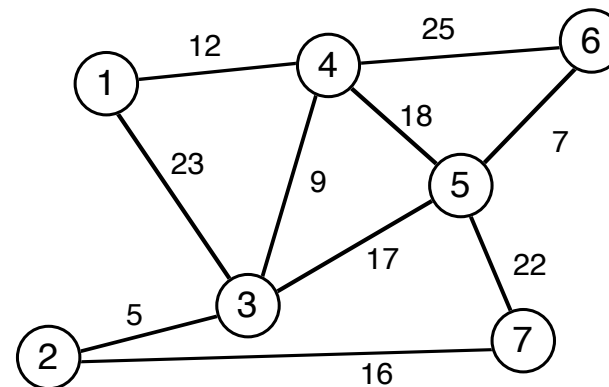
Idea: remove links until graph has tree structure, keep all nodes connected
=> spanning tree

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

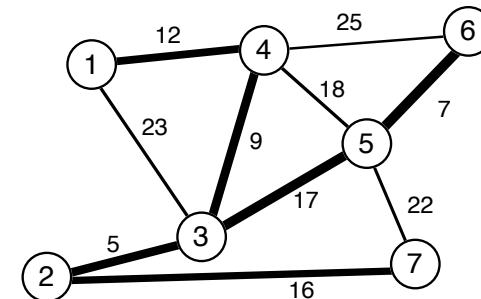
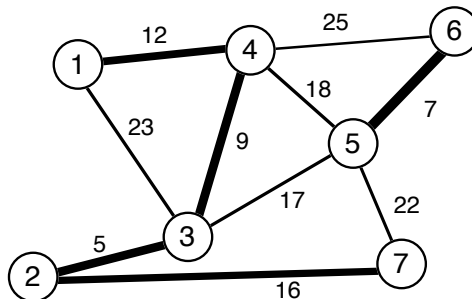
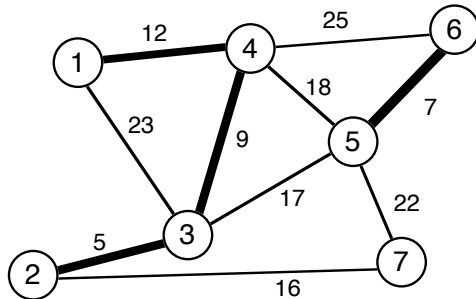
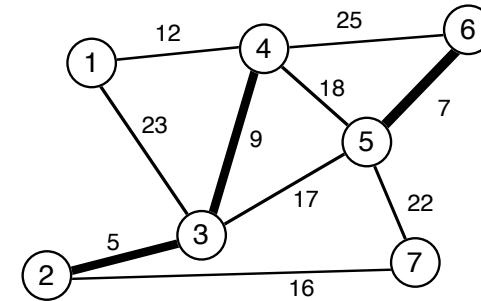
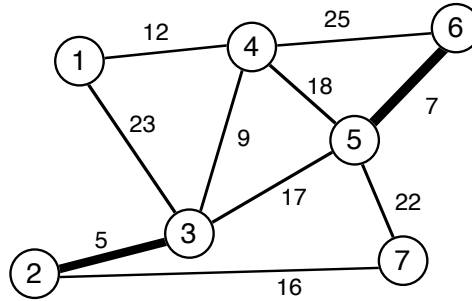
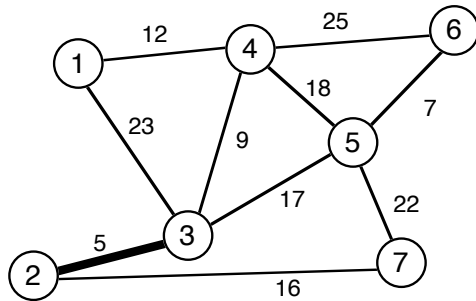
Greedy **Kruskal**-Algorithm:

=> **iteratively** choose unused edge
with **least weight**,
if it does not lead to a **circle**!

greedy \Leftrightarrow base choice on current state,
(locally optimal choice)



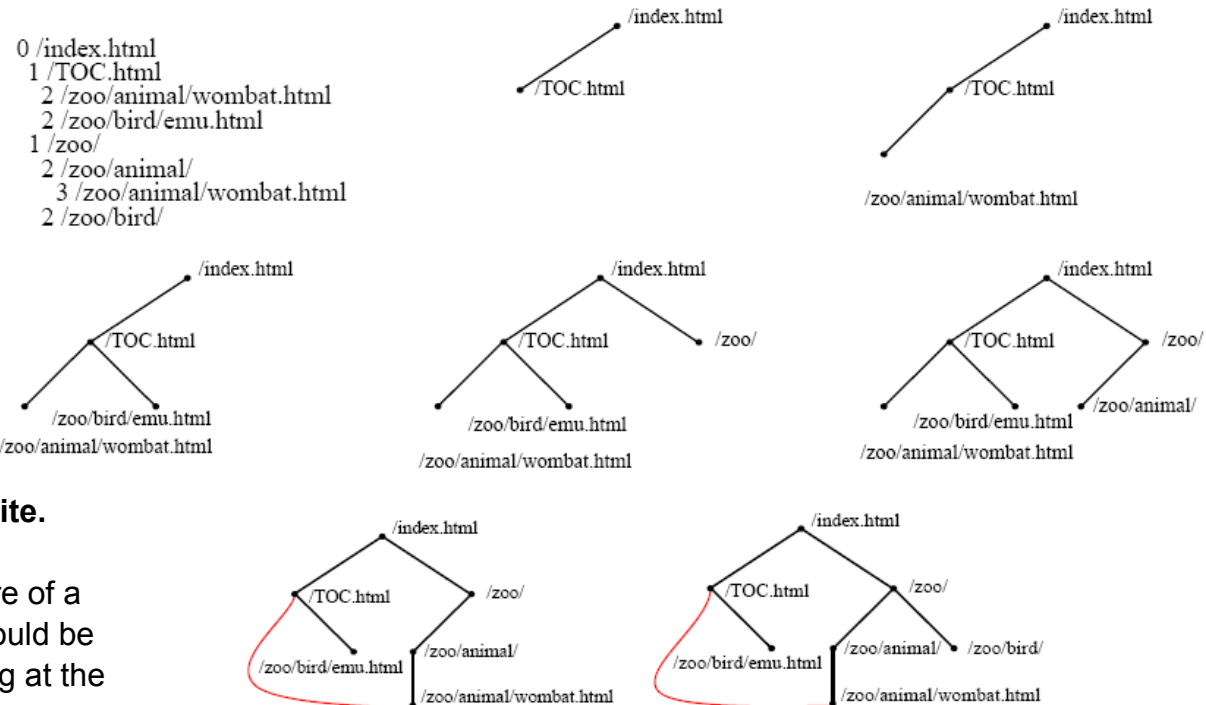
Kruskal - Example



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

Minimum spanning tree weight = 66

Another Spanning Tree



Constructing a spanning tree for a quasi-hierarchical web site.

Top Left: The hyperlink structure of a simple hypothetical site, as it would be reported by a web spider starting at the top page. Nodes represent web pages, and links represent hyperlinks. Although the graph structure itself is determined by hyperlinks, additional information about hierarchical directory structure of the site's files is encoded in the URLs.

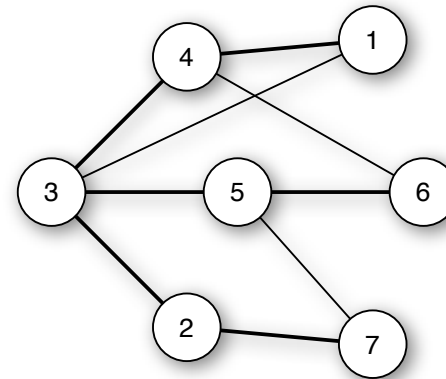
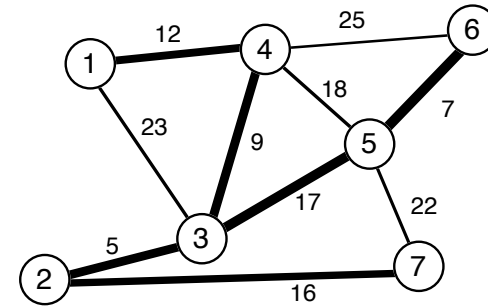
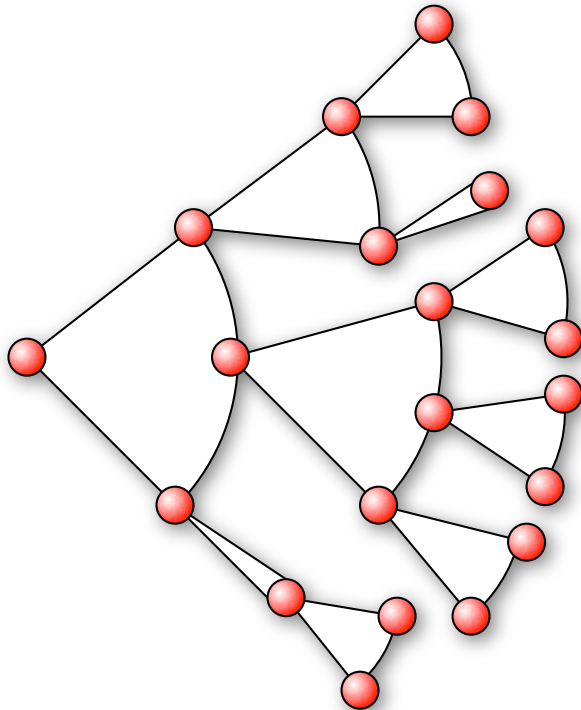
Top Row: We build up the graph incrementally, one link at a time.

Middle Row: We continue adding nodes without moving any of the old ones around.

Bottom Row: When the animal/wombat.html page is added, the label matching test shows that animal is a more appropriate parent than /TOC.html, so the node moves and the link between animal/wombat.html and /TOC.html becomes a non-tree link. In the final stage, note that bird/emu.html does not move when the bird is added, even though the labels match, because there is no hyperlink between them.

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:

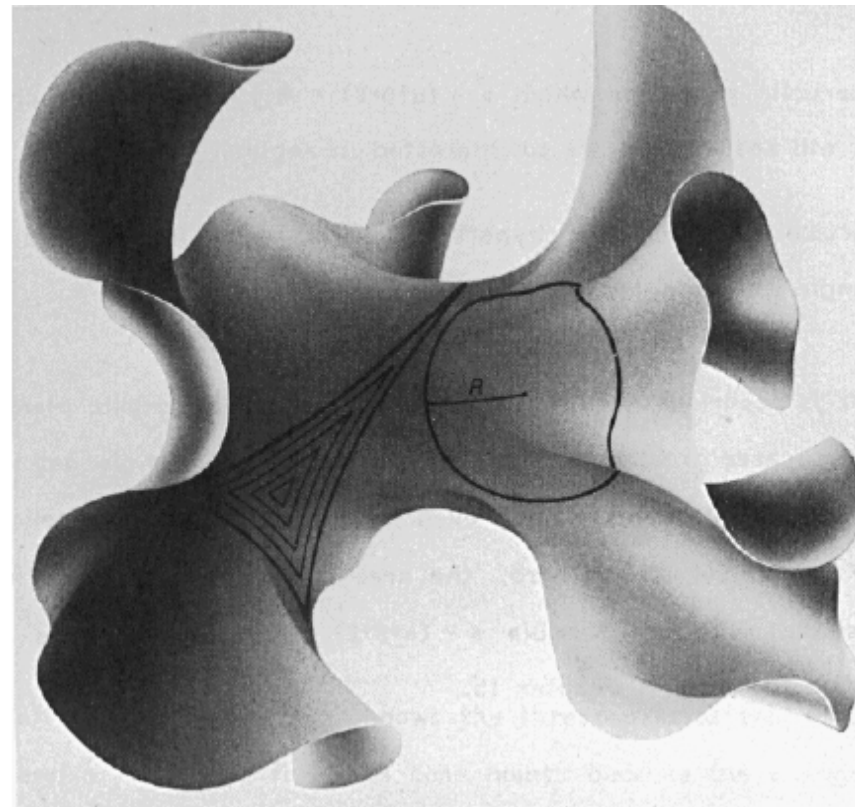
$$U = 2\pi \sinh r$$

=> exponentially **growing** space
on the circle

For (cone) graph layout

=> there is **enough room**
for yet another level

Also: **mappings** of the
complete hyperbolic space
=> finite volume of Euklidian space



Models of hyperbolic space

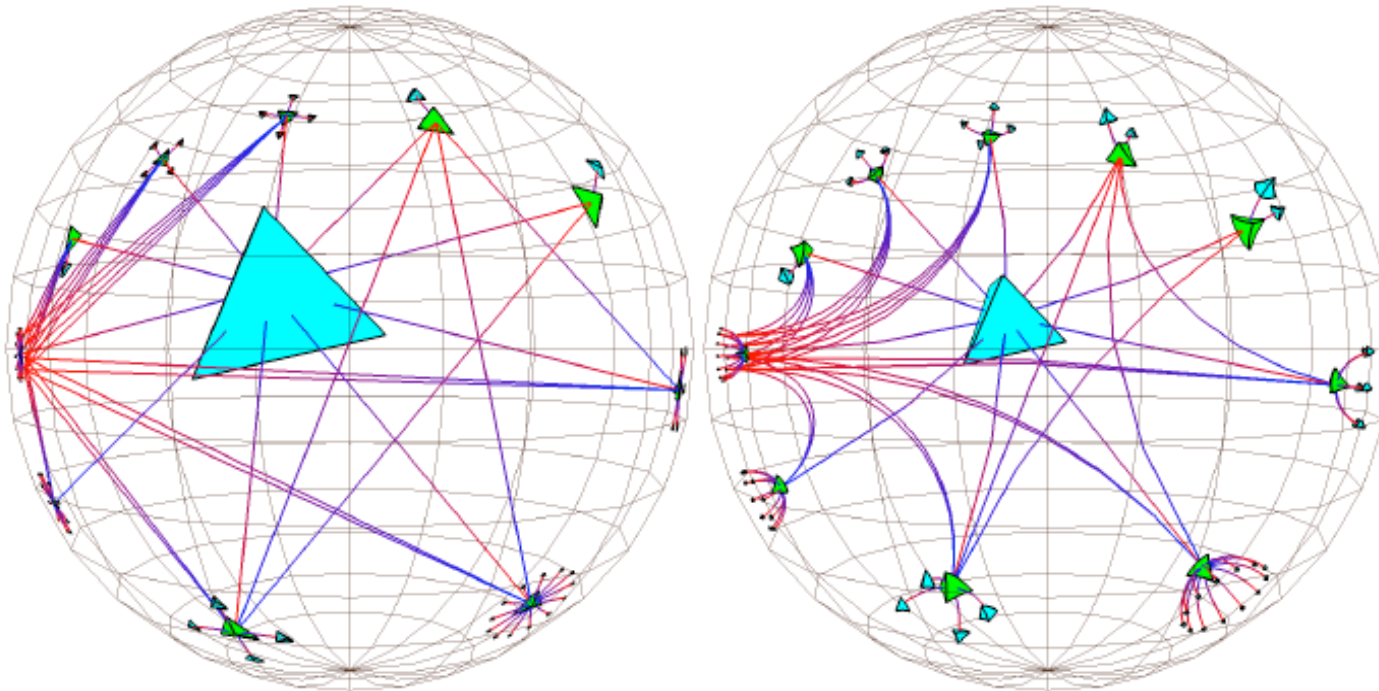


Figure 3.5: **Models of hyperbolic space.** **Left:** The projective model of hyperbolic space, which keeps lines straight but distorts angles. **Right:** The conformal model of hyperbolic space, which preserves angles but maps straight lines to circular arcs. These images were created with the *webviz* 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

Visualization with H3

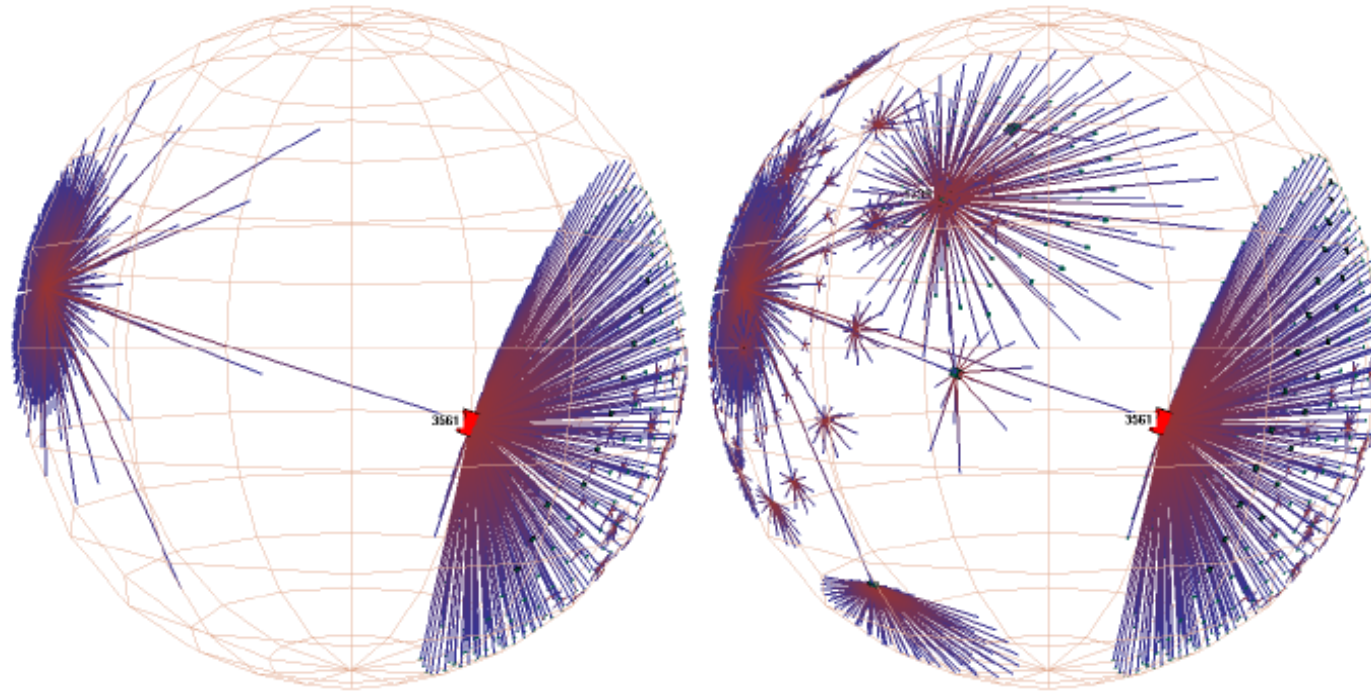


Figure 3.13: **Active vs. idle frames, obvious case.** The H3Viewer guaranteed frame rate mechanism ensures interactive response for large graphs, even on slow machines. **Left:** A frame drawn in 1/20th of a second during user interaction. **Right:** A frame filled in by the idle callbacks for a total of 2 seconds after user activity stopped. The graph shows the peering relationships between Autonomous Systems, which constitute the backbone of the Internet.⁸ The 3000 routers shown here are connected by over 10,000 edges in the full graph.

PhD thesis Tamara Munzner, chapter 3

Visualization with H3

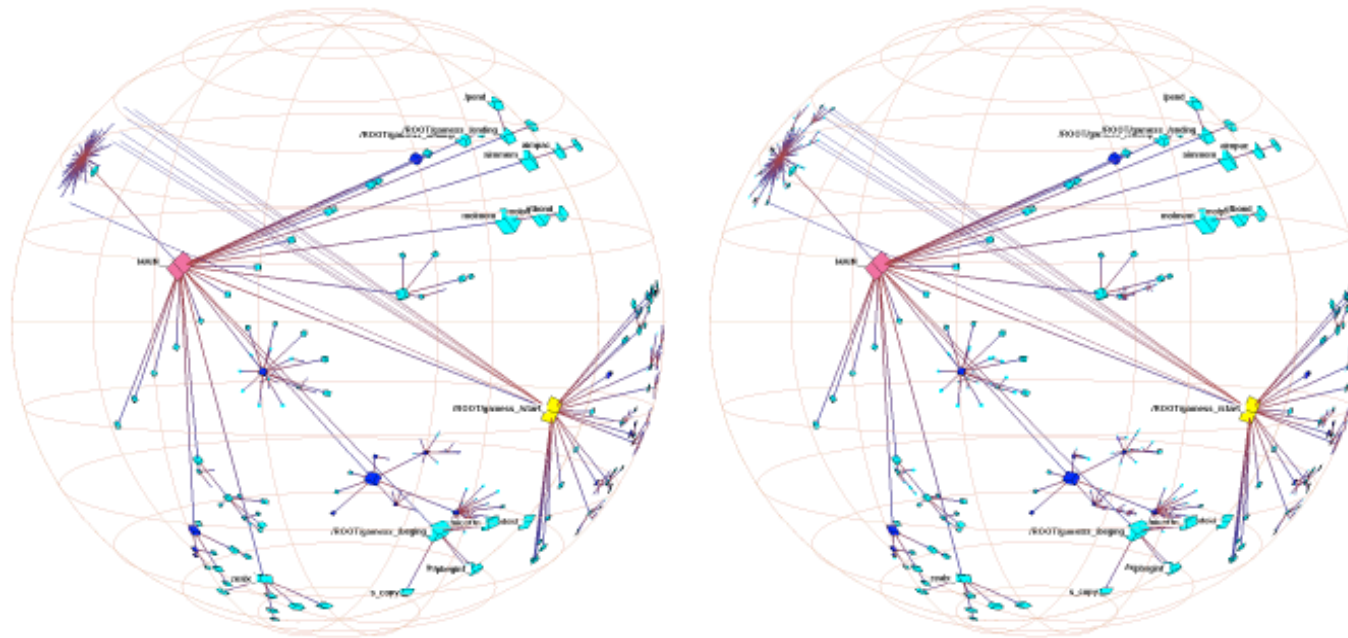
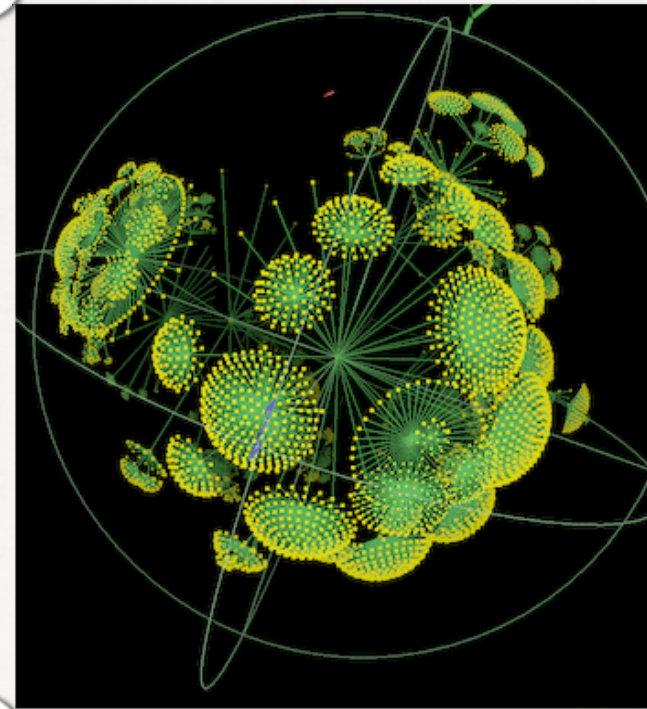
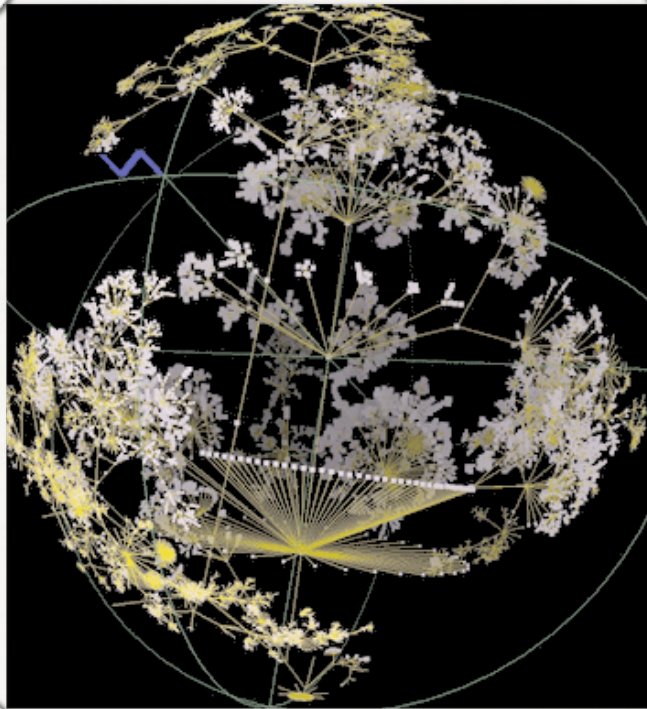


Figure 3.14: **Active vs. idle frames, subtle case.** A function call graph of a small FORTRAN benchmark with 1000 nodes and 4000 edges. Non-tree links from one of the functions are drawn. **Left:** a single frame has been drawn in 1/20th of a second. Note that the non-tree links to the distant fringe are visible even though their terminating nodes were small enough that the active drawing loop terminated before they could be drawn. Our drawing algorithm thus hints at the presence of potentially interesting places that the user might wish to drag toward the center to see in more detail. **Right:** The entire graph is drawn after the user has stopped moving the mouse. The graph is small enough that the difference between the two frames is more subtle than in Figure 3.13, and is visible only in the fringe in the upper left corner.

PhD thesis Tamara Munzner, chapter 3

GIFs don't work here...



<http://www.caida.org/tools/visualization/walrus/gallery/>

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

Summary

What you learned **today**:

- => Local connectivity: clustering
- => shortest path: Dijkstra algorithm
- => graph layout: force-directed and embedding schemes
- => spanning tree: Kruskal algorithm

Next lecture:

- => biological data to build networks from