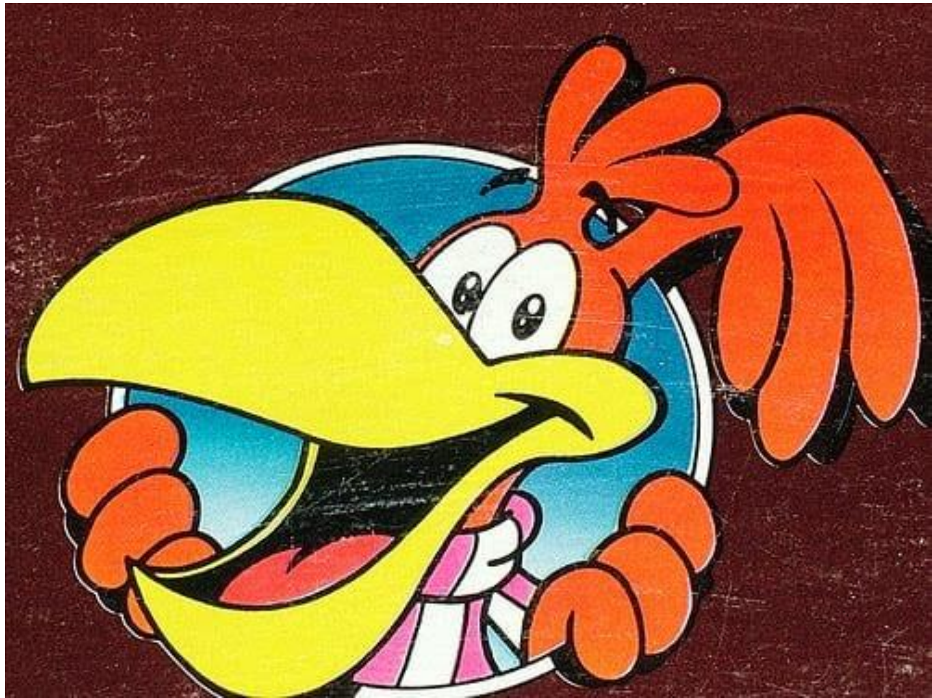
CSEP 521: Applied Algorithms Lecture 8 Cuckoo Hashing

Richard Anderson

January 28, 2021



Announcements

-

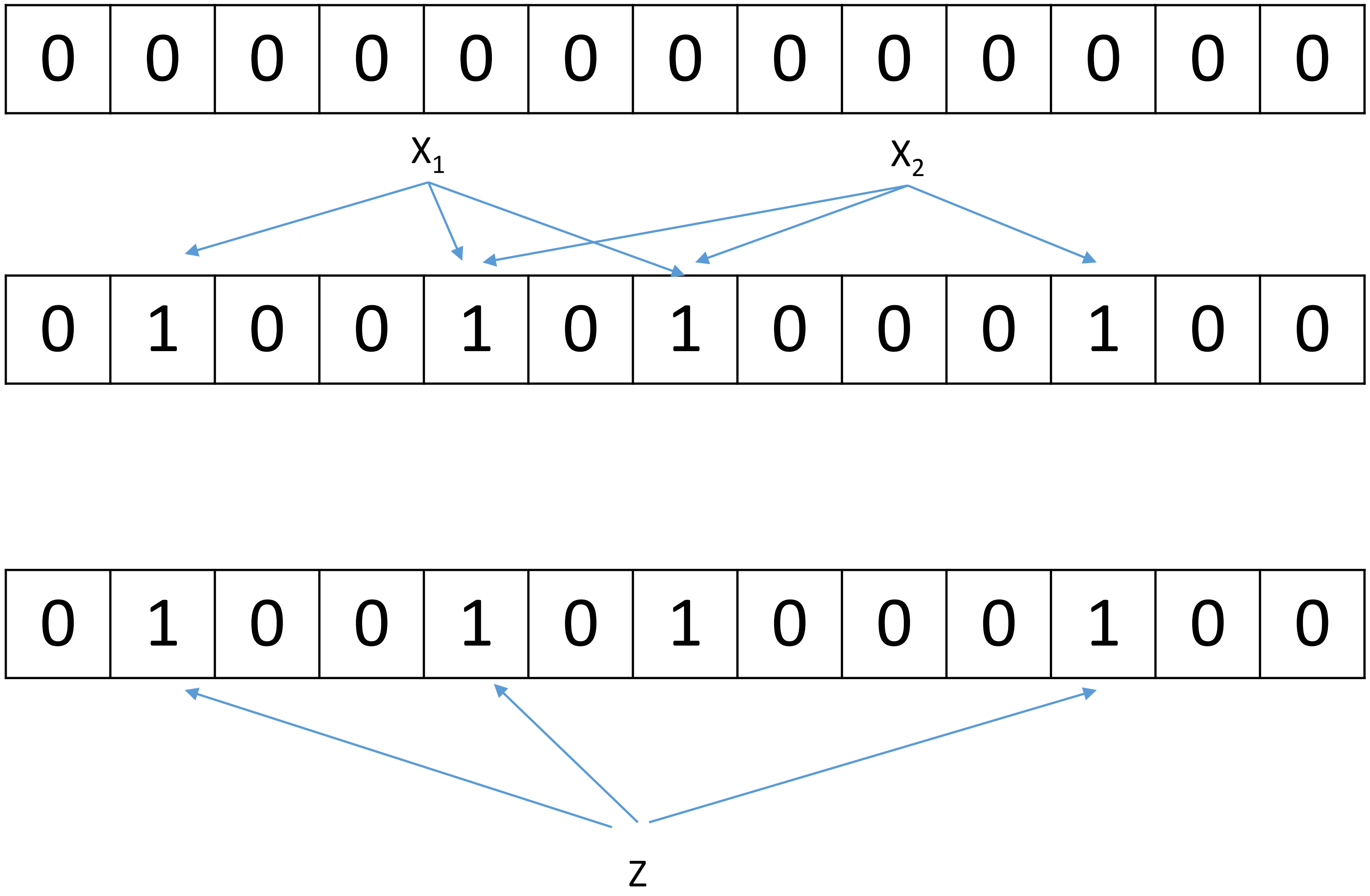
Finishing up Bloom Filters

- Basic idea – k-hash functions

- Bits are set at $h_1(x)$, $h_2(x)$, . . ., $h_k(x)$
- Lookup is done by reading $h_1(x)$, $h_2(x)$, . . ., $h_k(x)$

- False positives are possible, false negatives are not
- Goal is to have a small number of bits per value
- Example: set of malicious URLs

Bloom Filter Example (k = 3)



Some Bloom Filter Math

0	0	0	0	0	0	0	0	0	0	0	0	0

- Table size m, data items n
- After n members of S have been hashed, compute the probability of specific element being zero

$$(1-\frac{1}{m})^{kn} \simeq e^{-kn/m} = p$$

- False probability rate

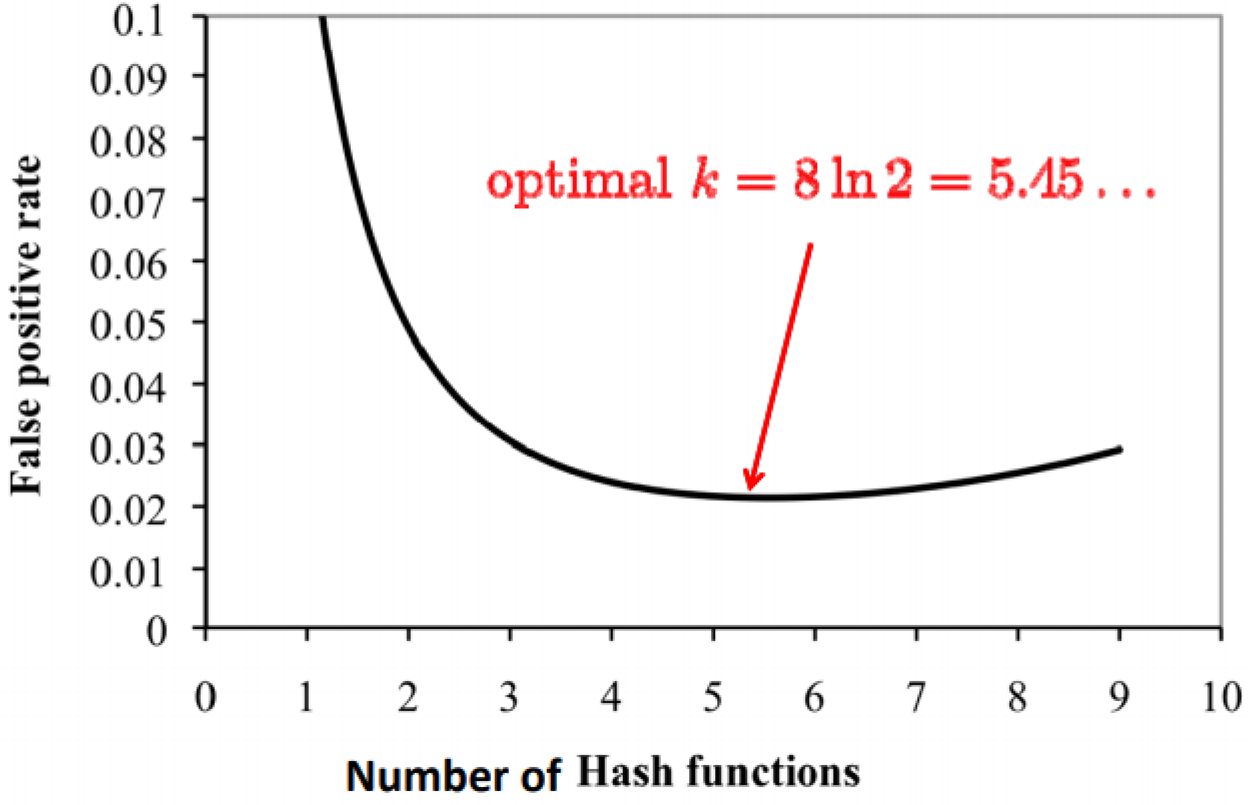
$$(1 - p)^{k}$$

False positive rate vs. k
Find optimal with calculus

$$f' = (1 - p')^k = (1 - (1 - \frac{1}{m})^{kn})^k$$

$$f = (1 - p)^k = (1 - e^{-kn/m})^k$$

Number of bits per member $\frac{m}{n} = 8$



Rewrite f as

$f = \exp(\ln(1 - e^{-kn/m})^k) = \exp(k \ln(1 - e^{-kn/m}))$

Let $\boxed{g = k \ln(1 - e^{-kn/m})}$

Minimizing g will minimize $f = \exp(g)$

$$\frac{\partial g}{\partial k} = \ln(1 - e^{-kn/m}) + \frac{k}{1 - e^{-kn/m}} \frac{\partial(1 - e^{-kn/m})}{\partial k}$$

$$= \ln(1 - e^{-kn/m}) + \frac{k}{1 - e^{-kn/m}} \frac{n}{m} e^{-kn/m} = -\ln(2) + \ln(2) = 0$$

if we plug $\boxed{k = (m / n) \ln 2}$ which is optimal

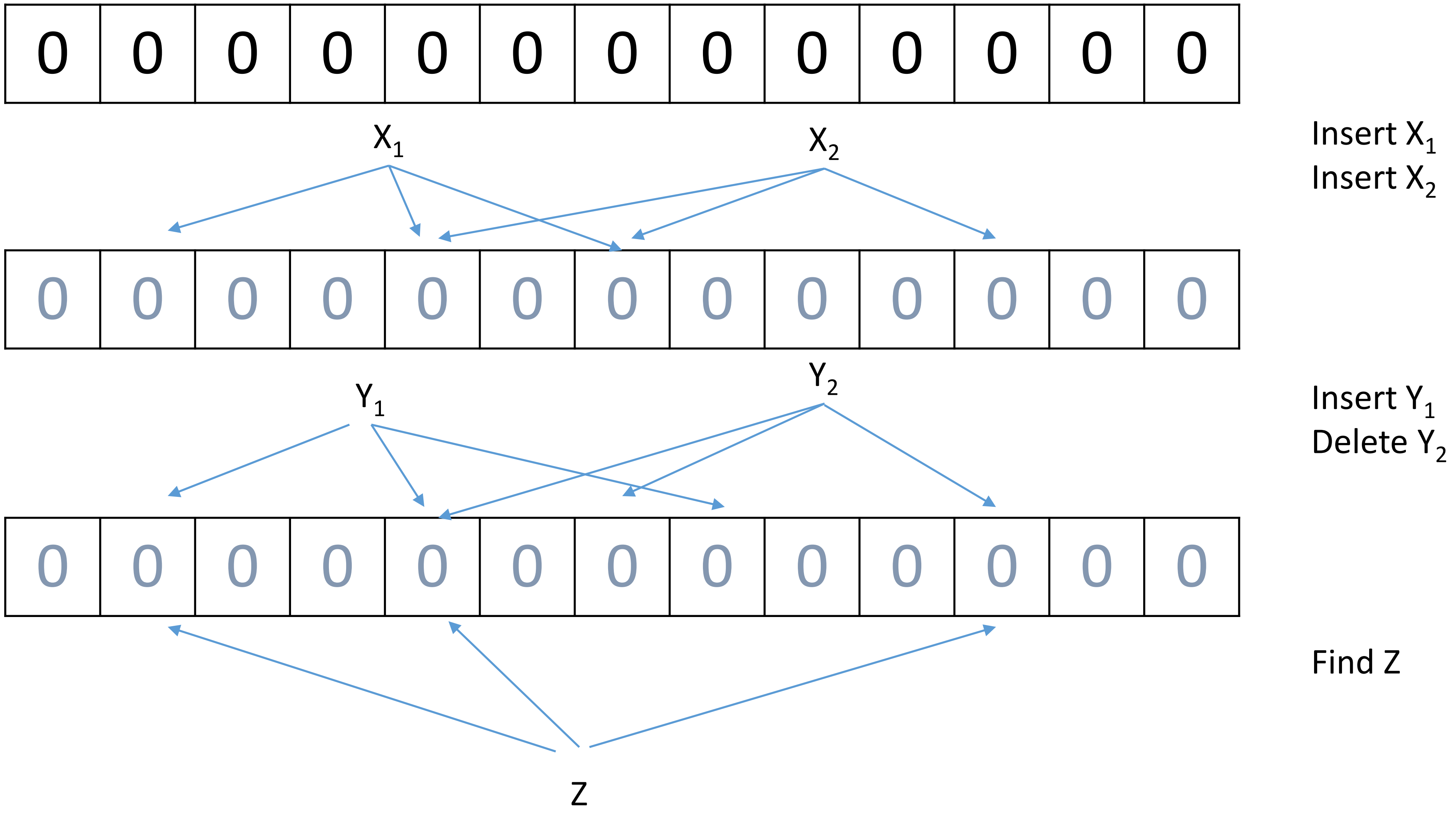
(It is in fact a global optimum)

Bloom filter deletes

- Why do Bloom filters fail for deletes?
- Counting Bloom Filters
- Each cell is a counter (4 bits considered sufficient)
- Insert, add one to each target cell
- Delete, subtract one from each target cell
- Find, test if target cells non-zero

- On overflow, leave counter at maximum value

Bloom Filter Deletes (k = 3)



Cuckoo hashing

A table based hashing scheme that is competitive with other table based hashing schemes and has guaranteed constant find and delete runtimes

Cuckoo Hashing claims

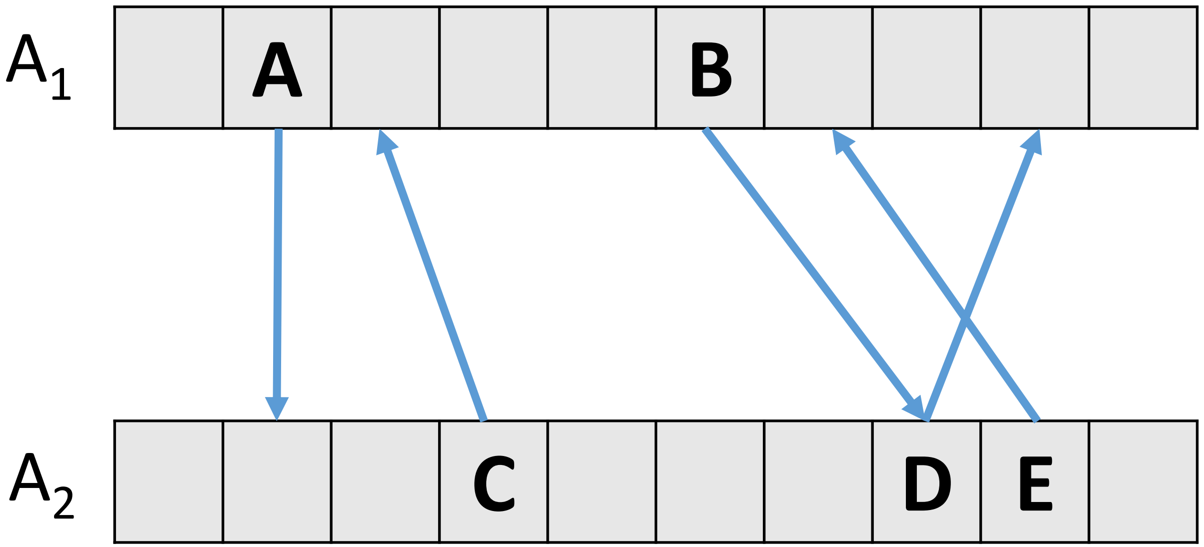
- Simple hashing without errors
- Lookups are worst case O(1) time
- Deletions are worst case O(1) time
- Insertions are **expected** O(1) time
- Insertion time is O(1) with **good probability**
- Plausibly suitable for real world application
  - Storage requirements and implementation complexity in line with conventional hash tables
- How do these compare with conventional hashing?

Cuckoos

- Brood parasitism
  - Laying eggs in other birds nests
- Cuckoo hashing
  - On collision, move to another cell
  - Generate a series of moves of data items

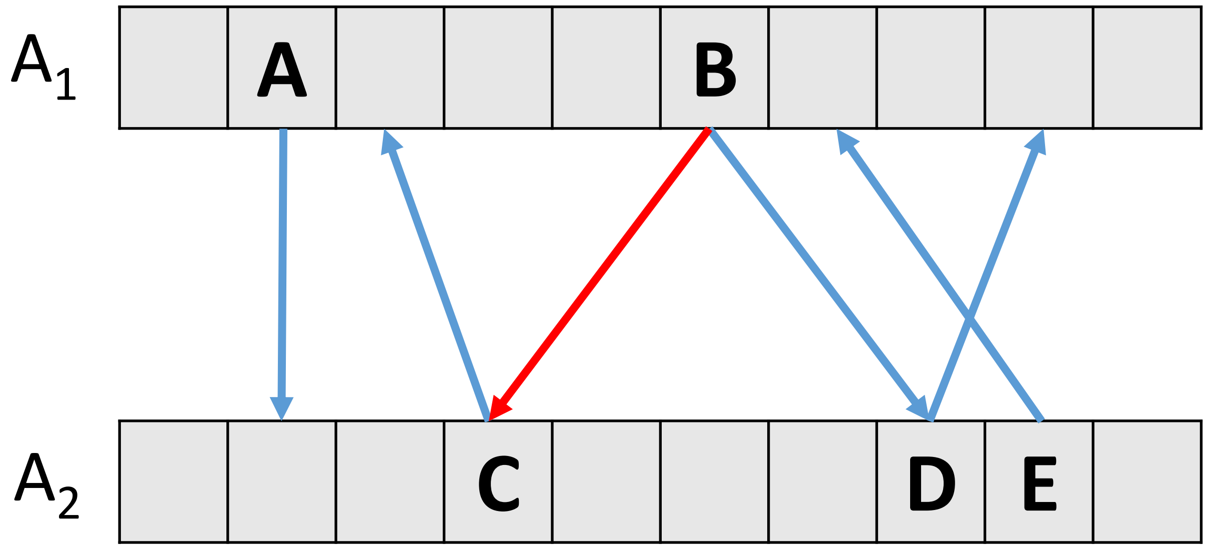
Data structure

- Two tables A_1 and A_2 of size m
- Two hash functions $h_1, h_2 : U \to [1..m]$
- When an element x is inserted, if either $A_1[h_1(x)]$ or $A_2[h_2(x)]$ is empty, store x there.



Bumping

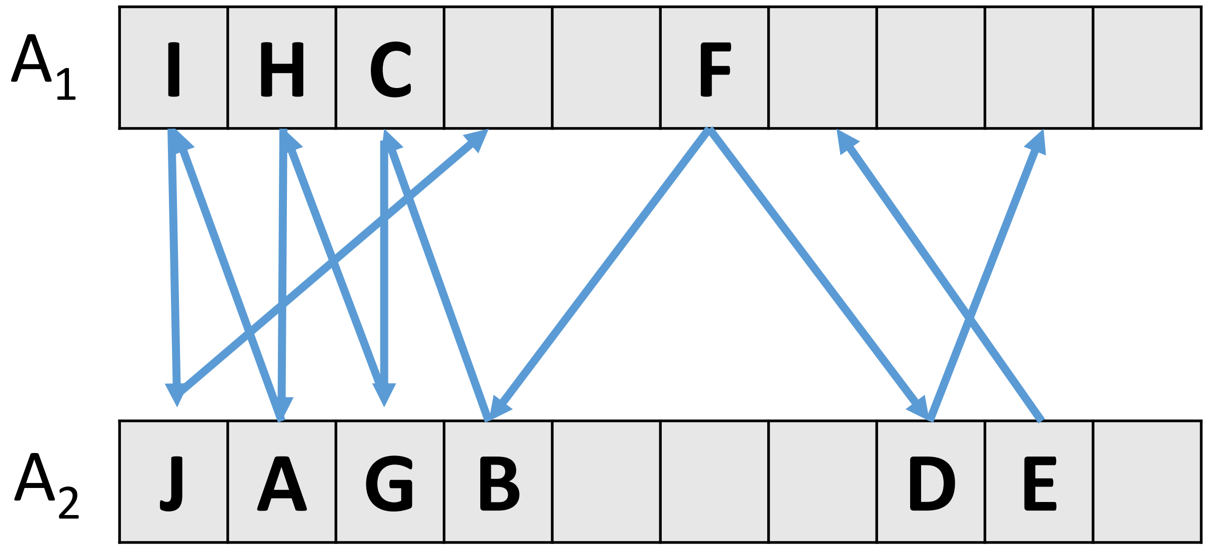
- When an element x is inserted, if either $A_1[h_1(x)]$ or $A_2[h_2(x)]$ is empty, store x there.
- If both locations are occupied, then place x in $A_1[h_1(x)]$ and bump the current occupant.
- When an element z is bumped
  - from $A_1[h_1(z)]$ store it in $A_2[h_2(z)]$
  - from $A_2[h_2(z)]$ store it in $A_1[h_1(z)]$



$h_1(F) = 6, \quad h_2(F) = 4$

Time outs

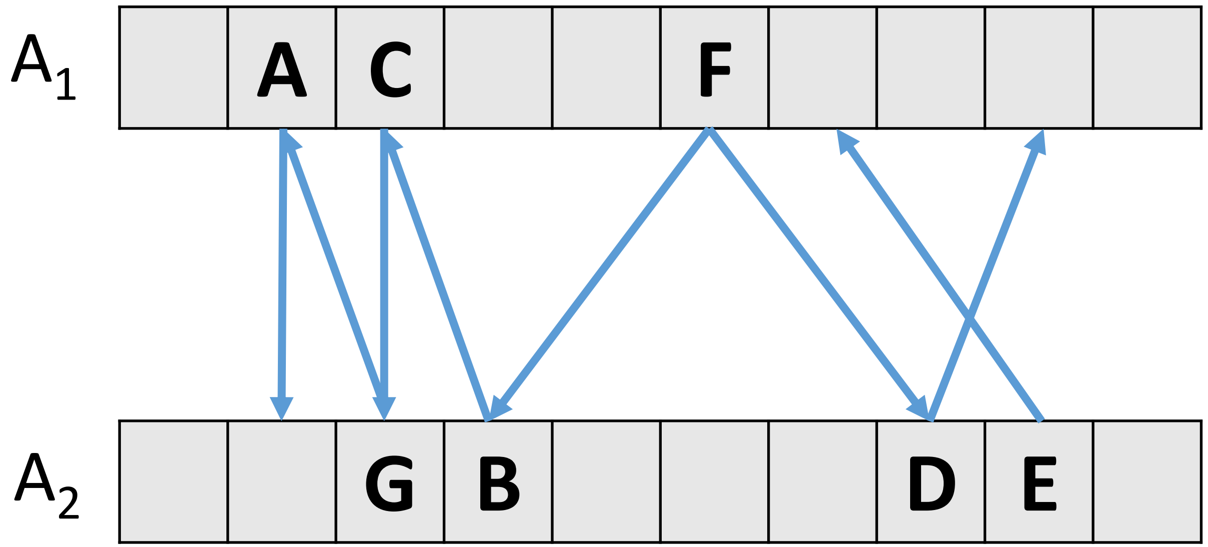
- After 6 log N consecutive bumps, stop the process and build a fresh hash table using new random hash functions h'_1 and h'_2
  - Might be a good time to grow the table as well



$h_1(K) = 6, \quad h_2(K) = 2$

Cycles

- Creating a cycle is okay
- Adding an edge to a cycle is not okay
- The problem with a cycle of length k with a cross edge is that we are trying to put k+1 items in k cells
- Detecting cycles can be done by the timeout



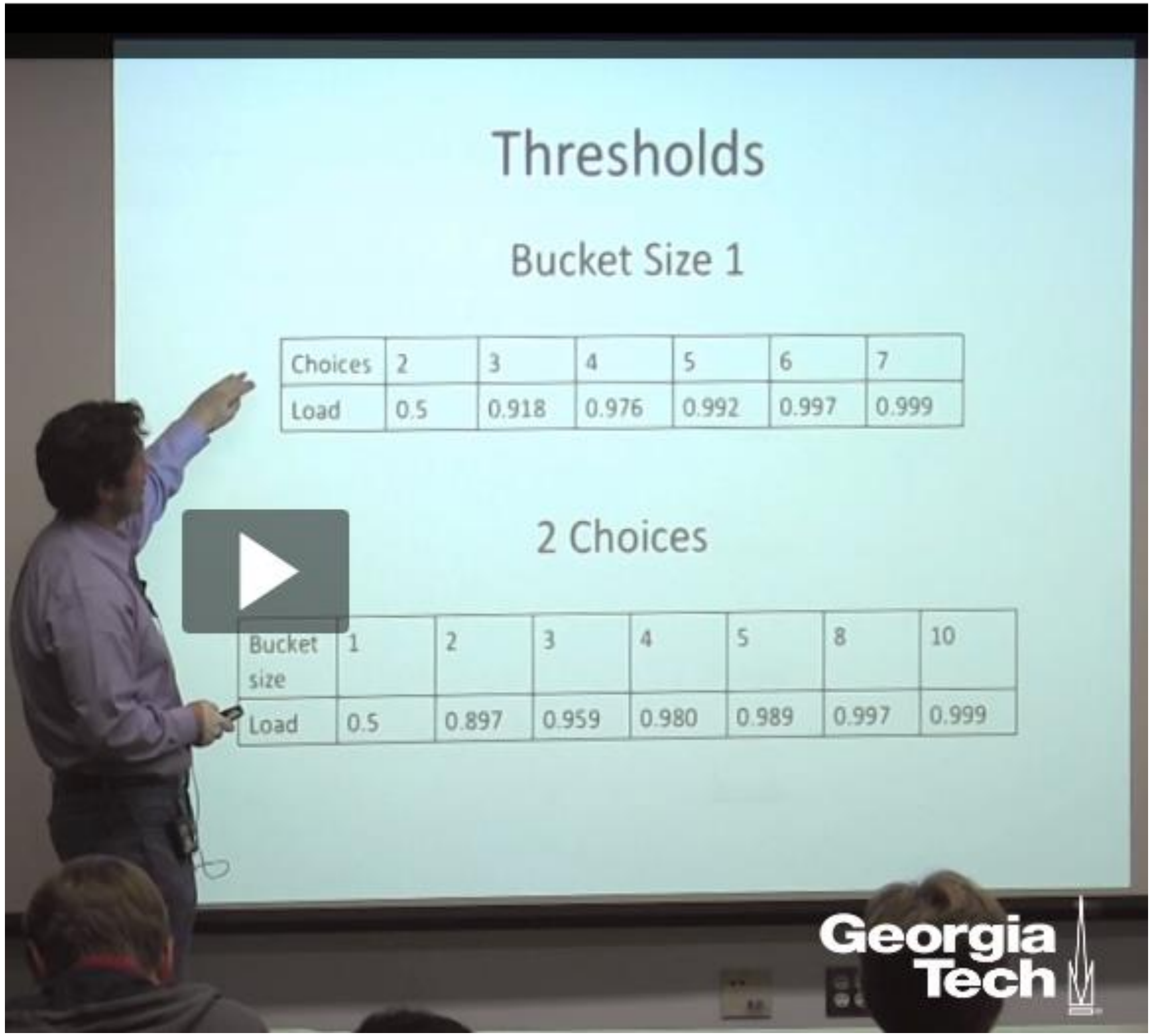
$h_1(H) = 6, \quad h_2(H) = 2$

$h_1(I) = 2, \quad h_2(I) = 4$

Theorem
Expected time for insert is O(1) if $m \geq 2n$

- Okay, 25% memory utilization, but can actually get about 50%

- Engineering optimizations
    - 3 hash functions instead of two
    - Multiple items stored in a bucket
    - These get 90% utilization in simulation, but tight analysis is open

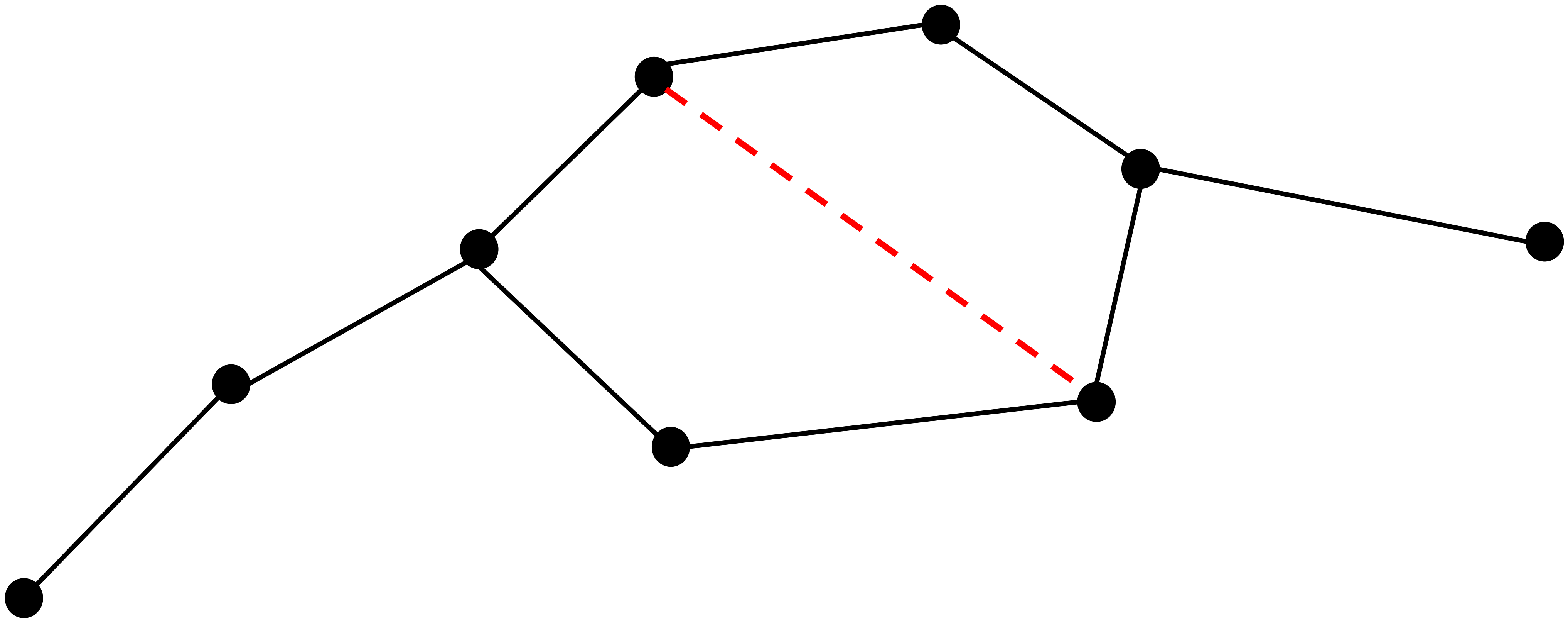


To show O(1) expected time

- We need to show
  - Expected traversal time is O(1) when we don’t time out
  - Probability of timing out is O(1/n)
- This is done with the theory for random graphs and is really, really hairy

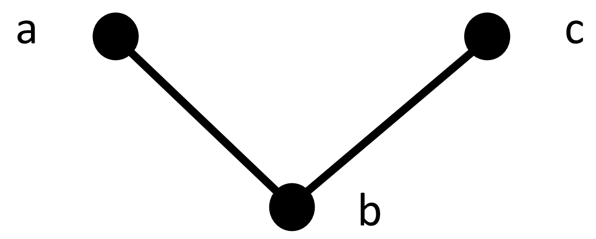
- In practice, failing items can be set aside in a ``stash’’ instead of rehashing
  - Experiments show that the number of elements stashed is tiny

Bad Case



Show this is really rare

Undirected Graphs

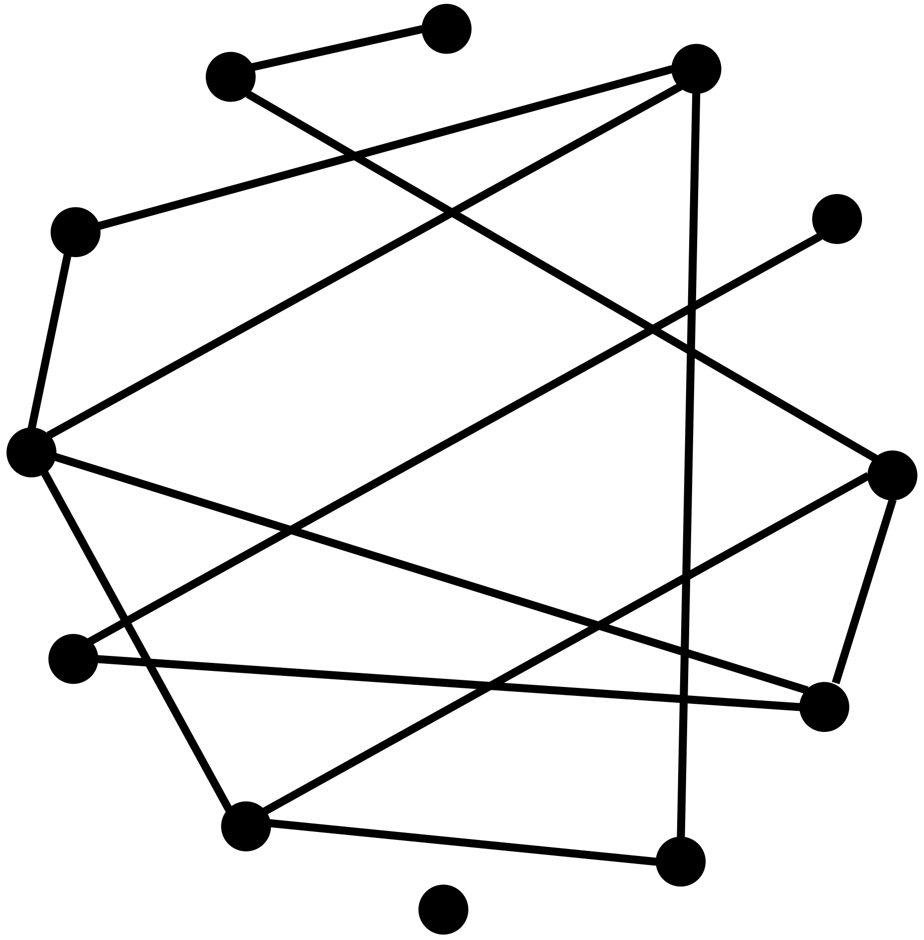


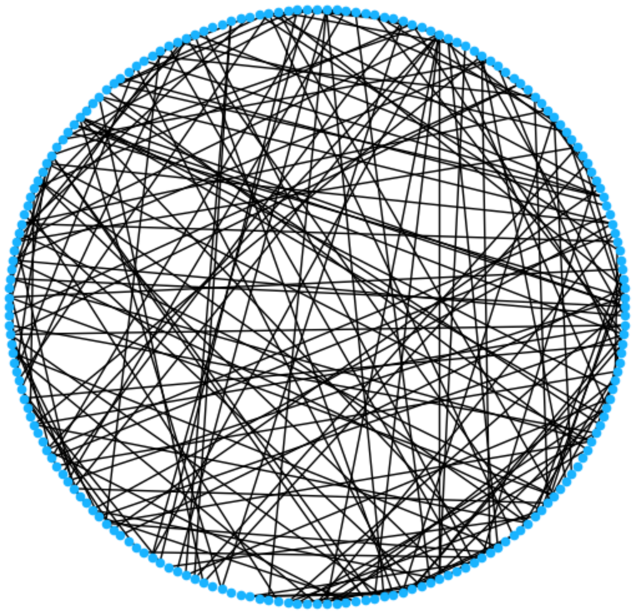
- Vertices V, |V| = n, V = {a, b, c}}
- Edges E, |E| = m, E ={{a,b}, {b,c}}
- $m \leq n(n-1) / 2$

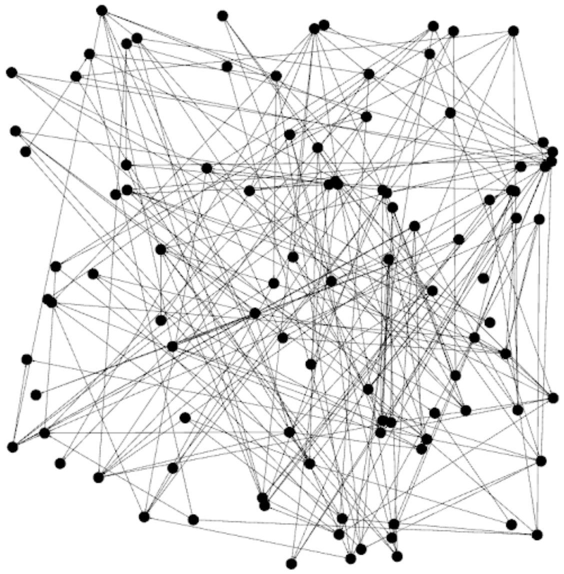
- Key concepts
  - Vertex degree
  - Isolated vertex
  - Path
  - Connectivity
  - Connected components

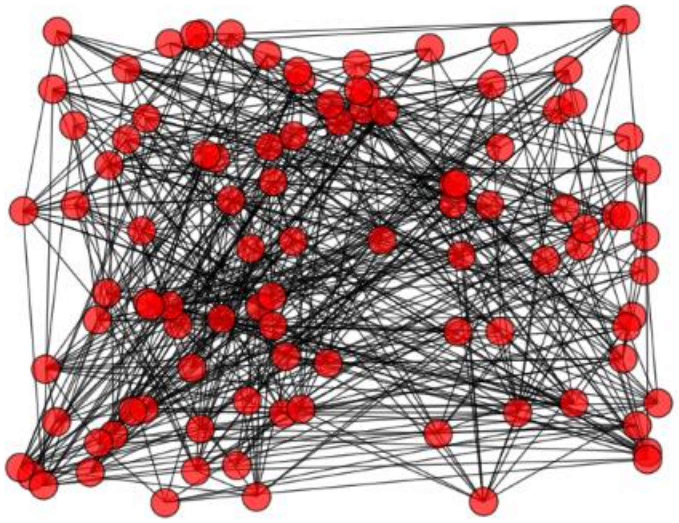
Random Graphs

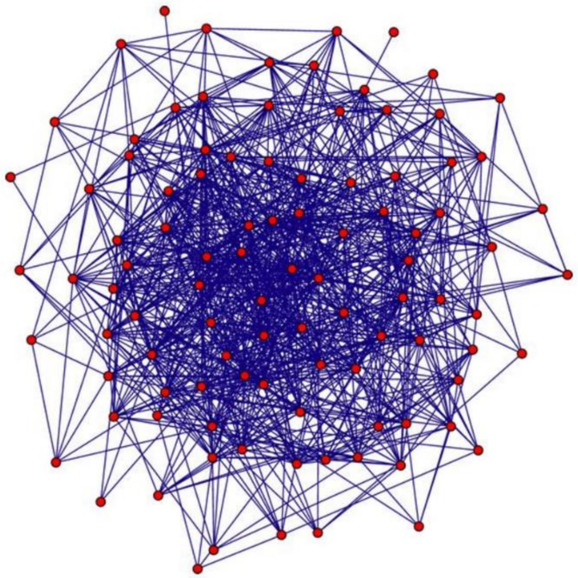
- $G^{n,m}$: Set of all undirected graphs with n vertices and m edges
- Select a graph uniformly from $G^{n,m}$
- Average degree is 2m/n

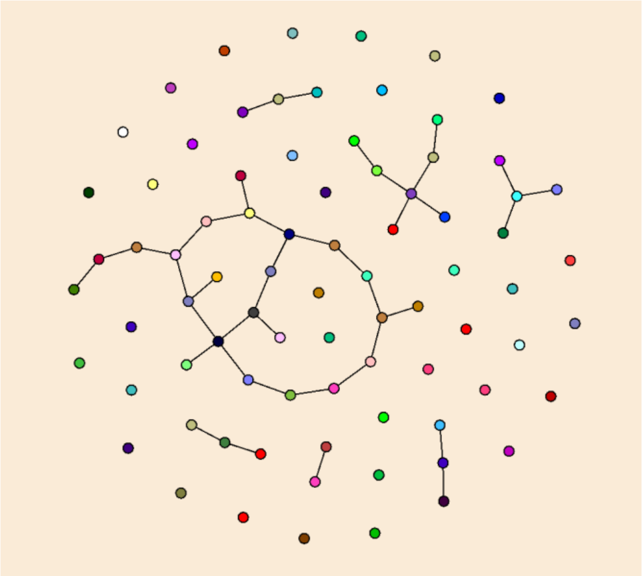
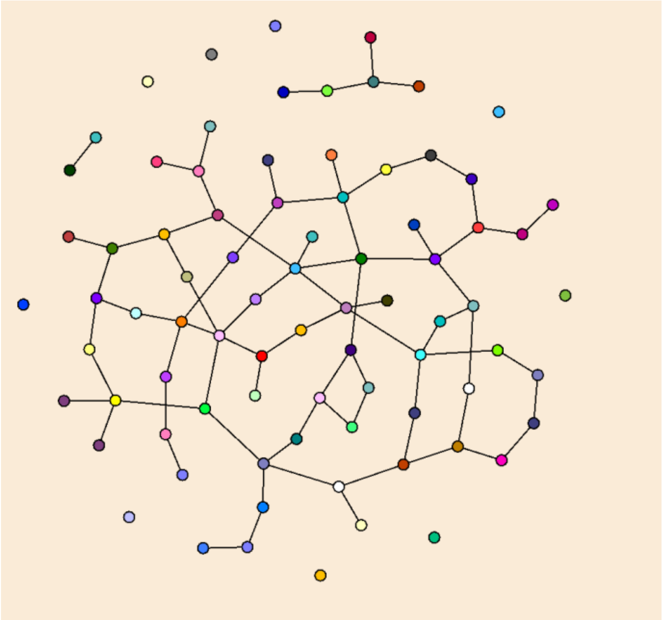
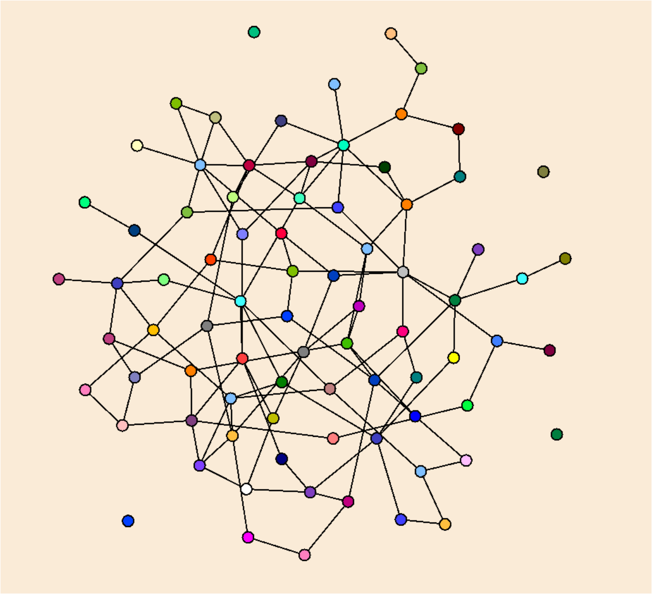
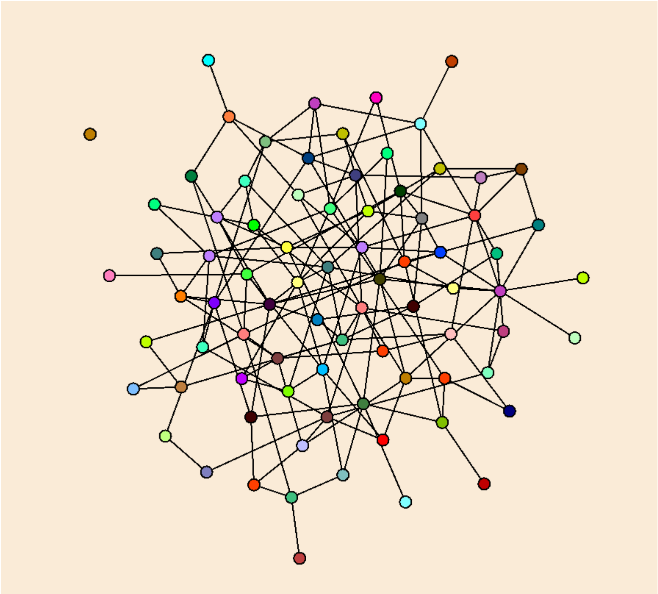










Edge Density

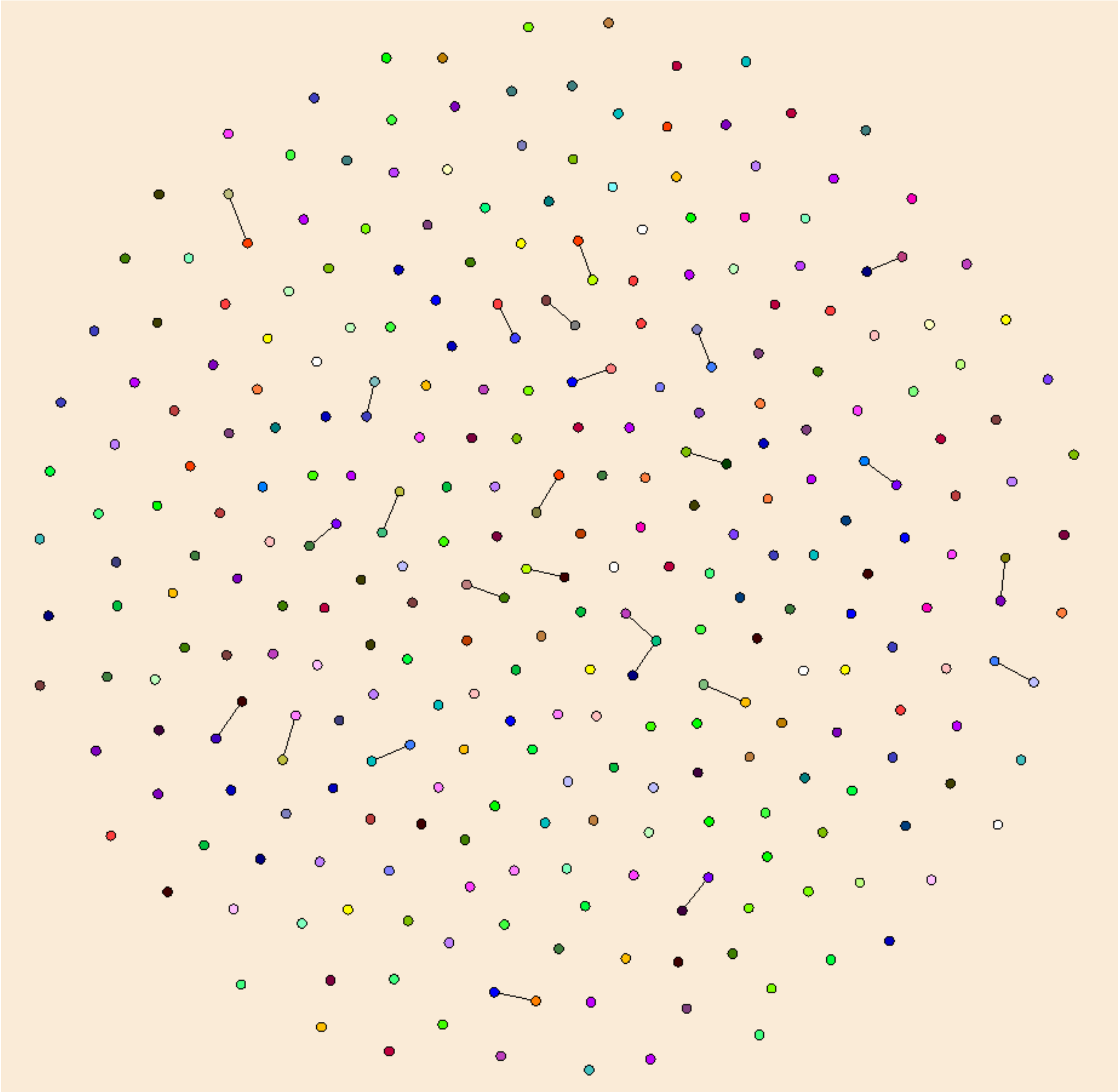
- Consider the number of edges as a function of the number of vertices
- Low density graphs, $m = O(n)$, degree $O(1)$
- Medium density, $m = O(n \log n)$, degree $O(\log n)$
- High density, $m = O(n^{1+\varepsilon})$, degree $O(n^{\varepsilon})$

Properties of Random Graphs

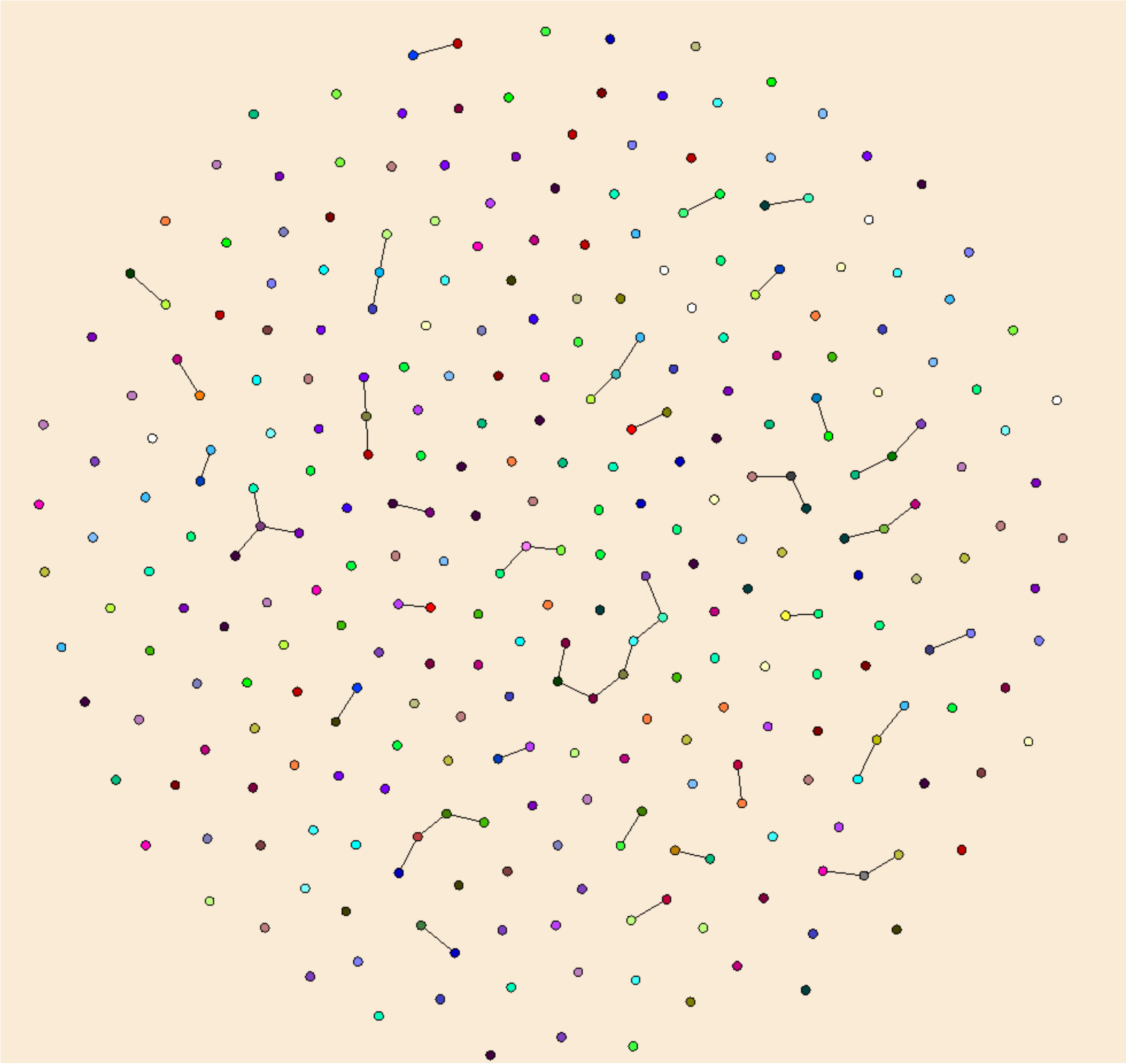
- Random graphs are surprisingly regular
- Edge degree is close to the average degree
- Dense graphs will be connected and have a Hamiltonian circuit (WHP)
- Low diameter

Evolution of Random Graphs

- Consider the process of building a random graph one edge at a time
- Look at how structures of the graph evolve
- Vertices start being connected, then start to form small components
- At a certain point, the components start to coalesce into a ``giant'' component with most of the vertices
- Finally, all of the vertices become connected

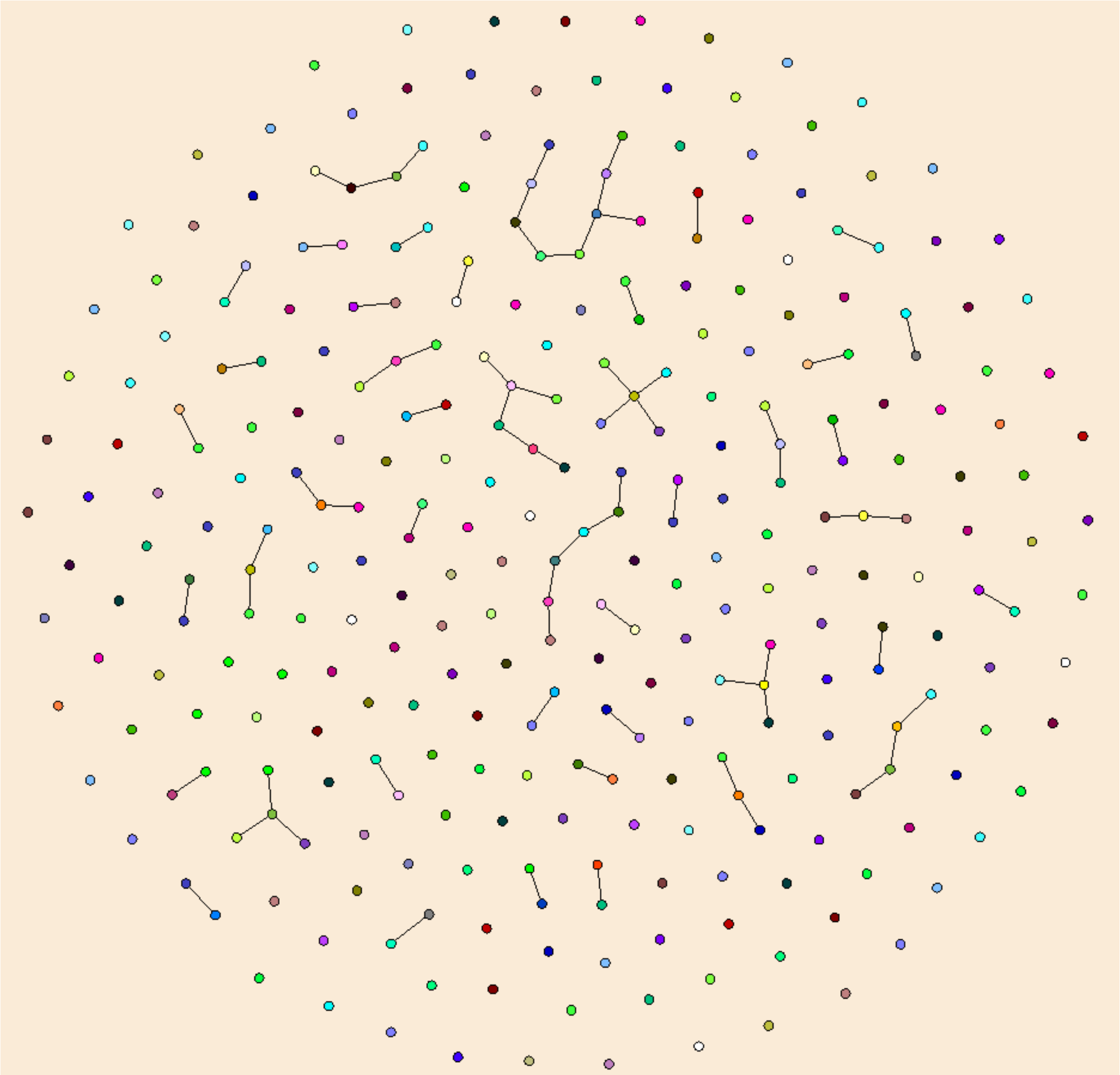


N = 300, M = 25

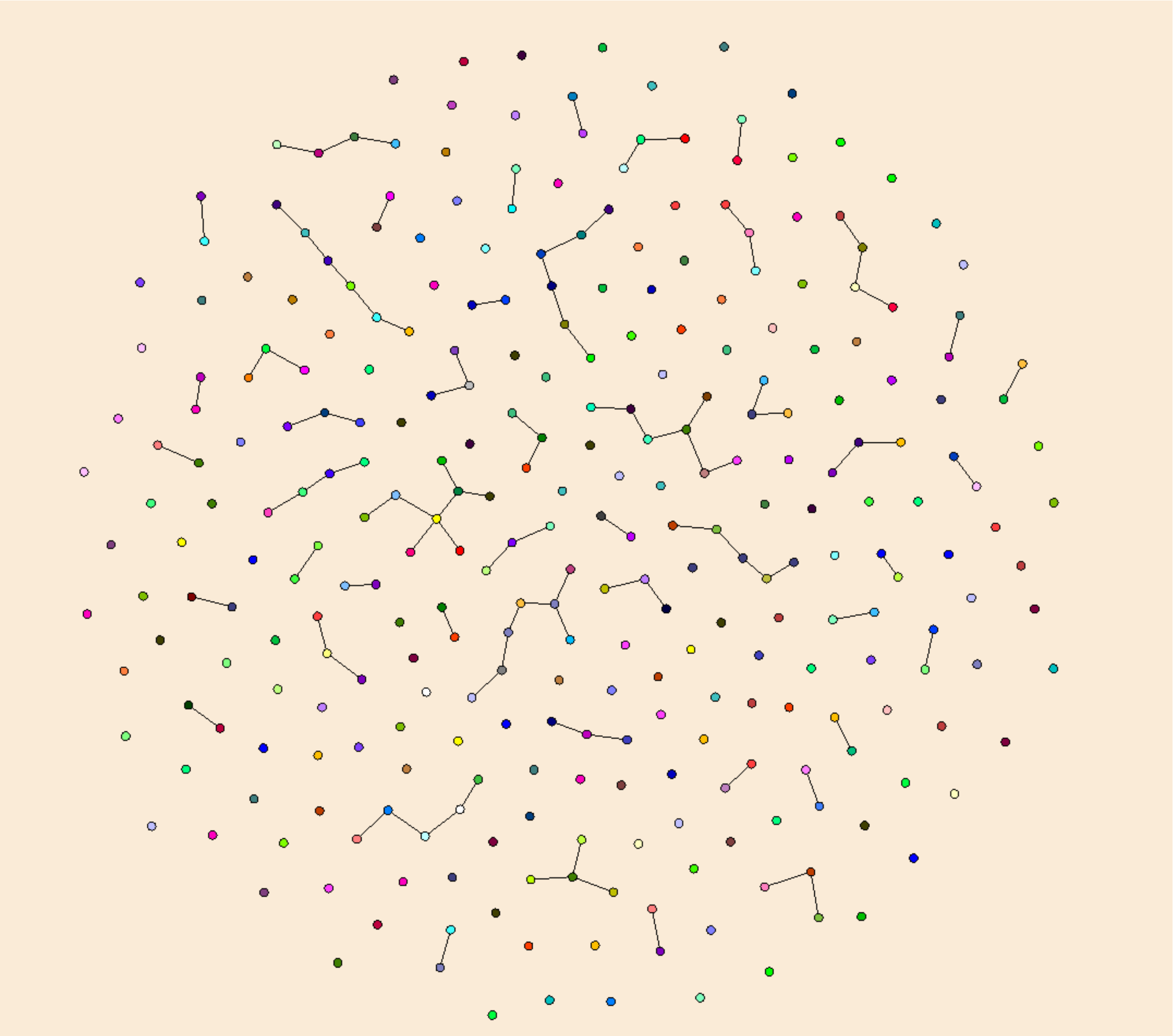


N = 300, M = 50

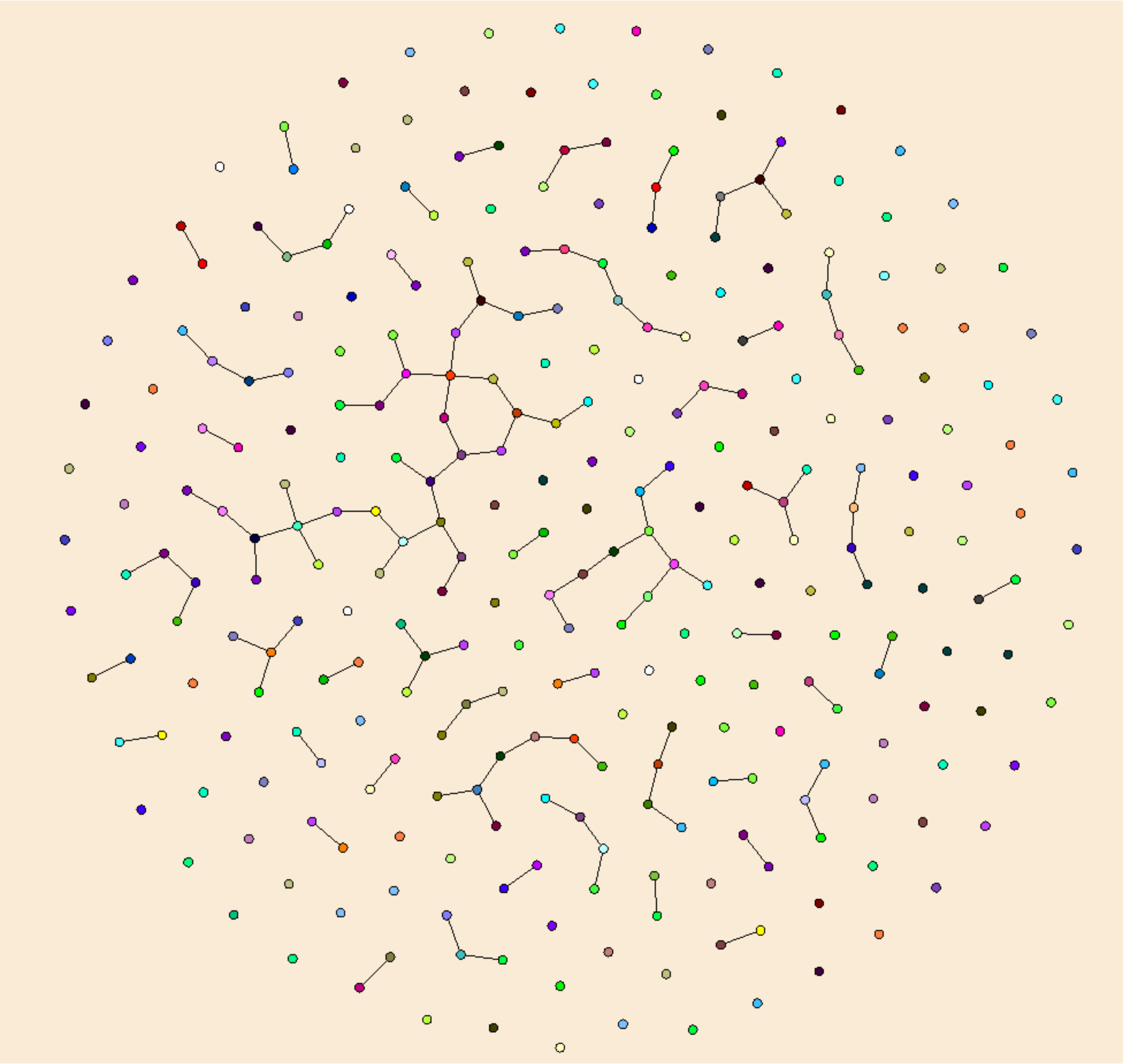
Random bipartite graphs with N=300



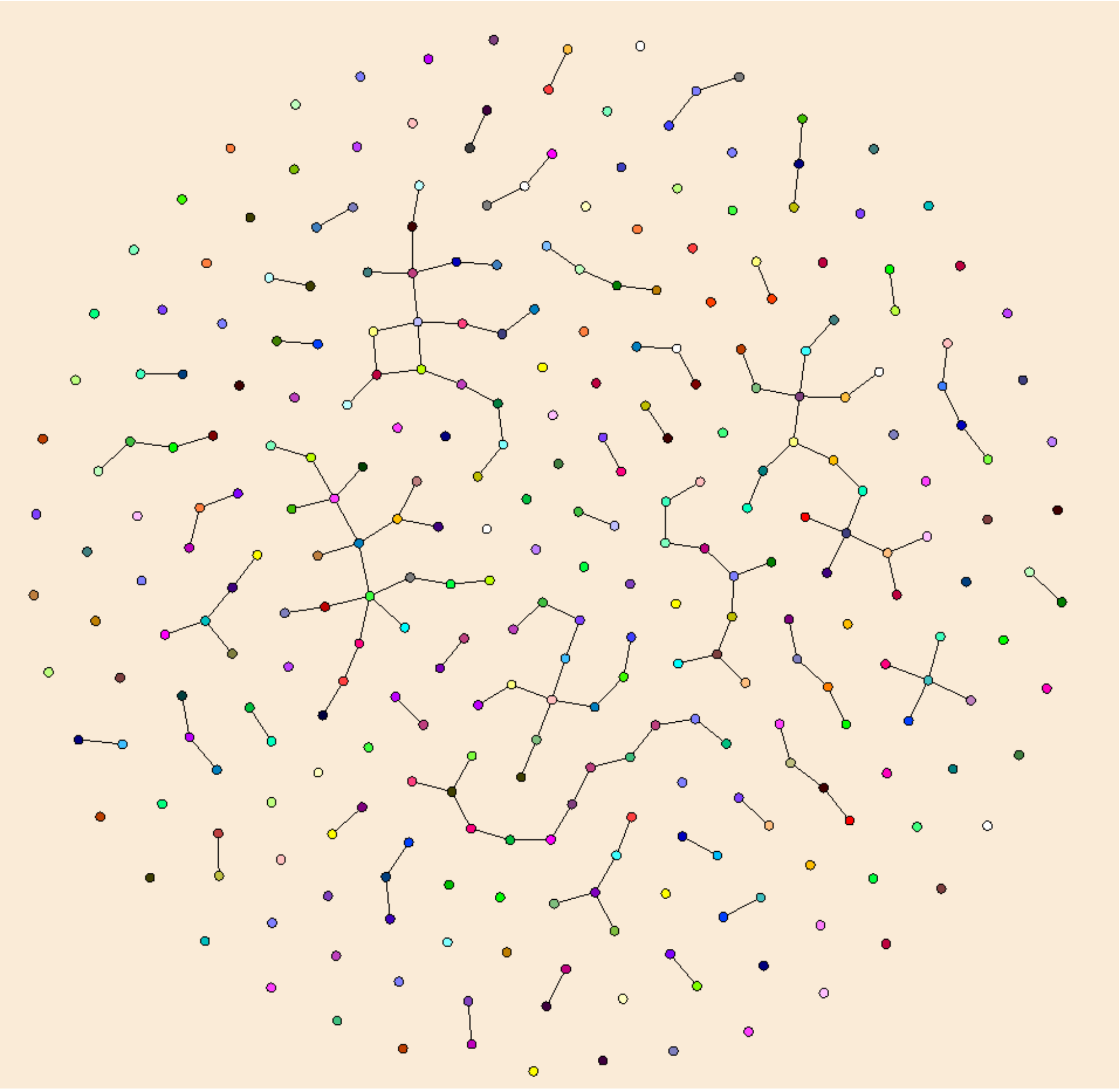
N = 300, M = 75



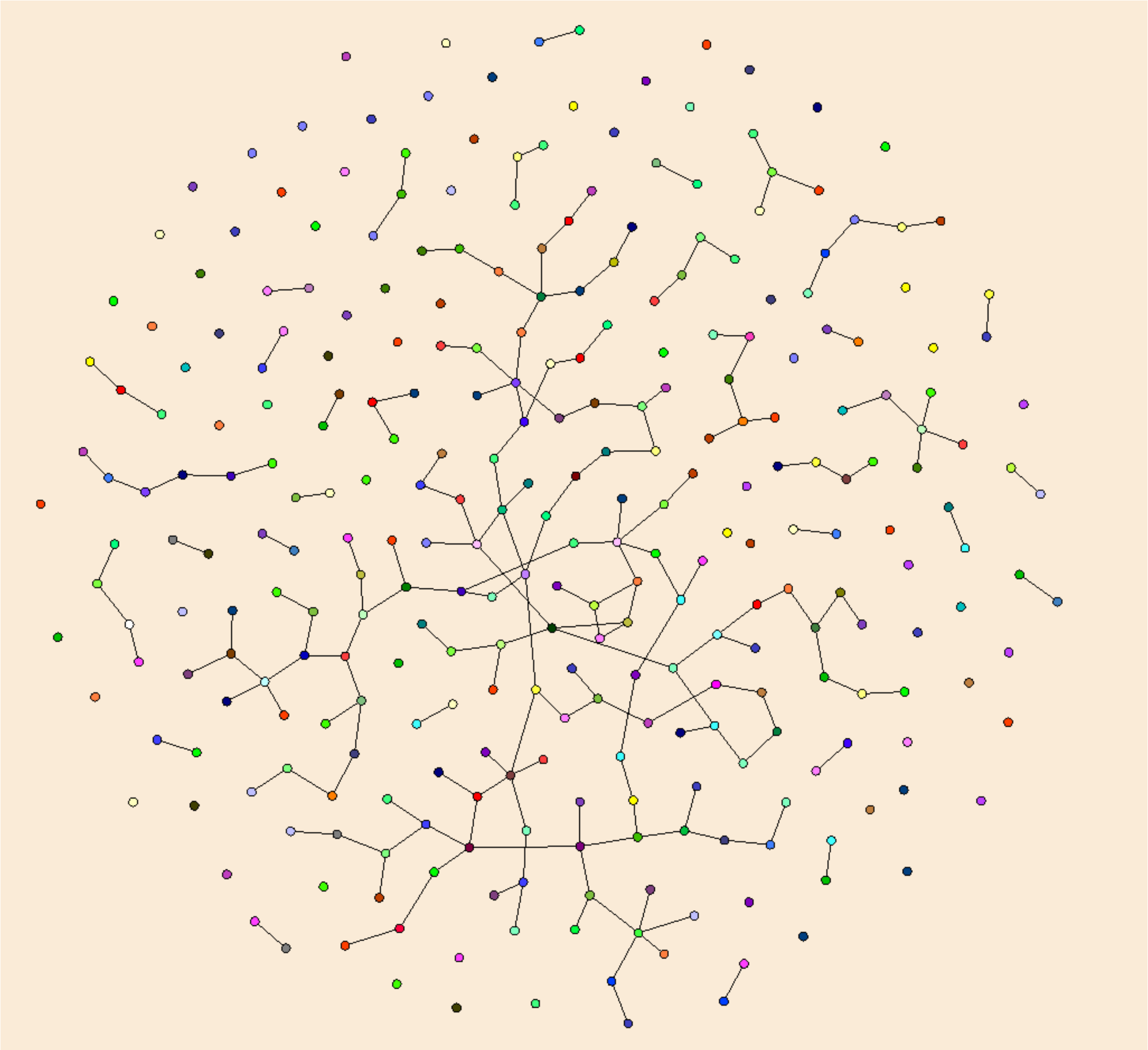
N = 300, M = 100



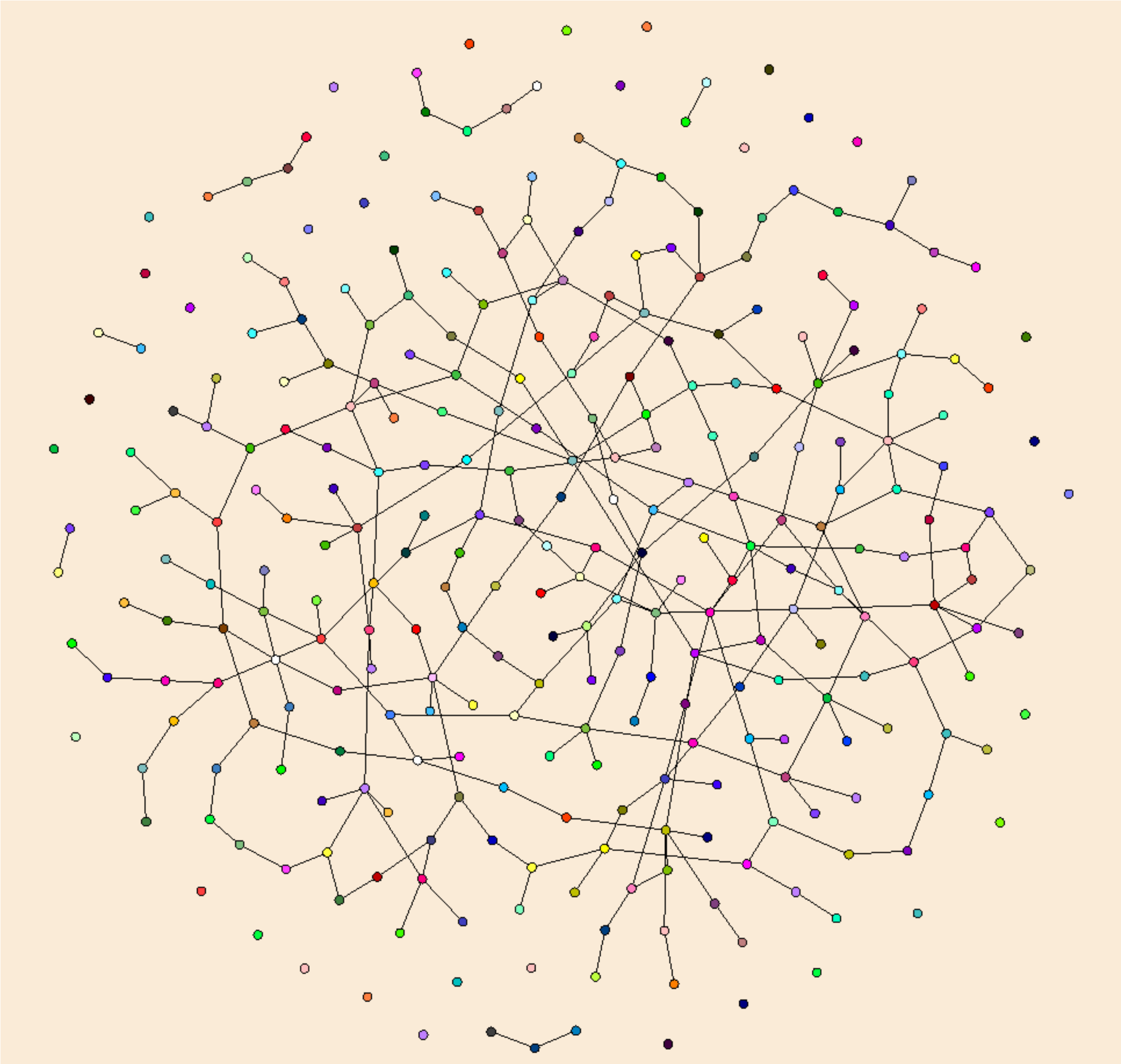
N = 300, M = 125



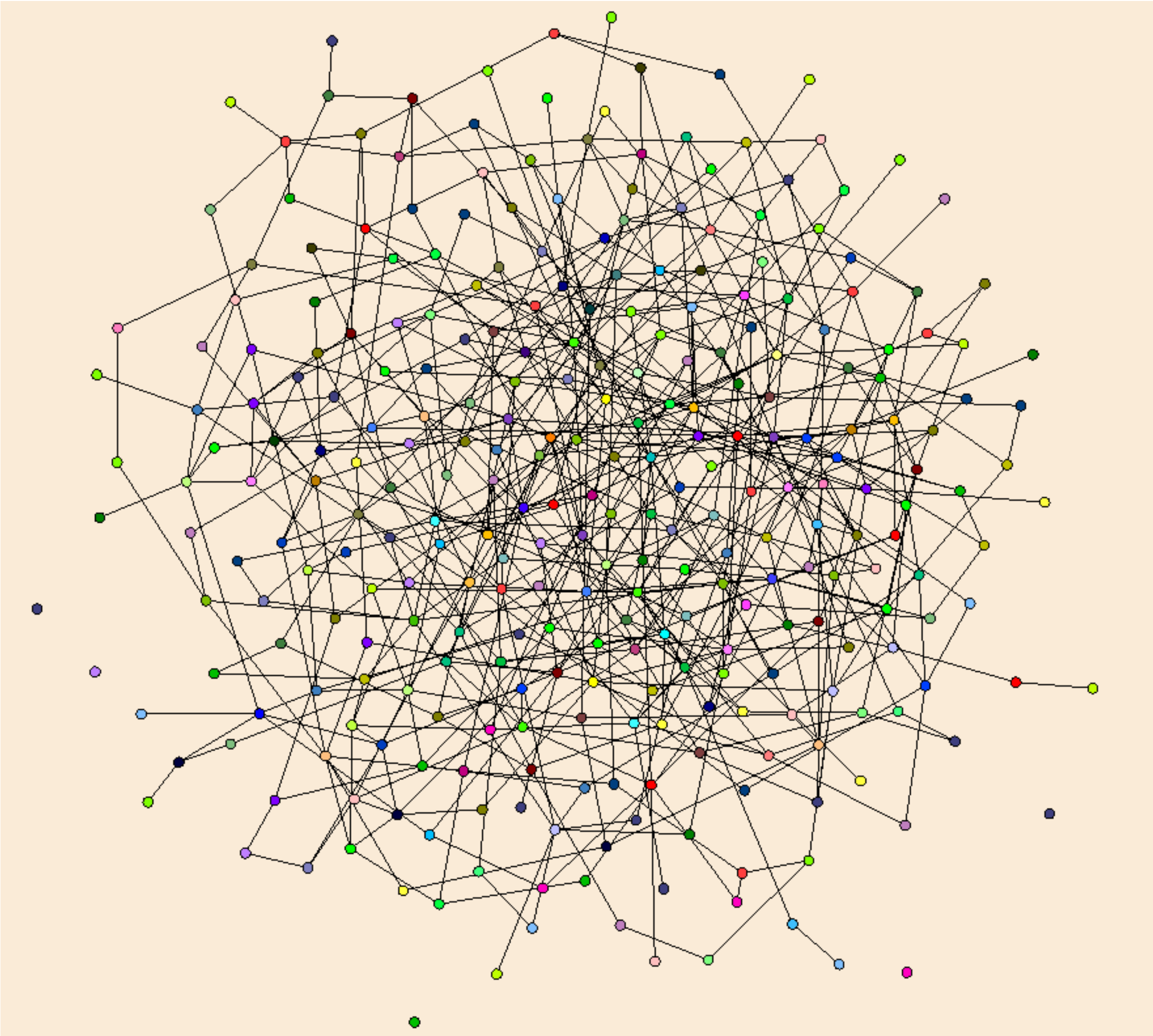
N = 300, M = 150



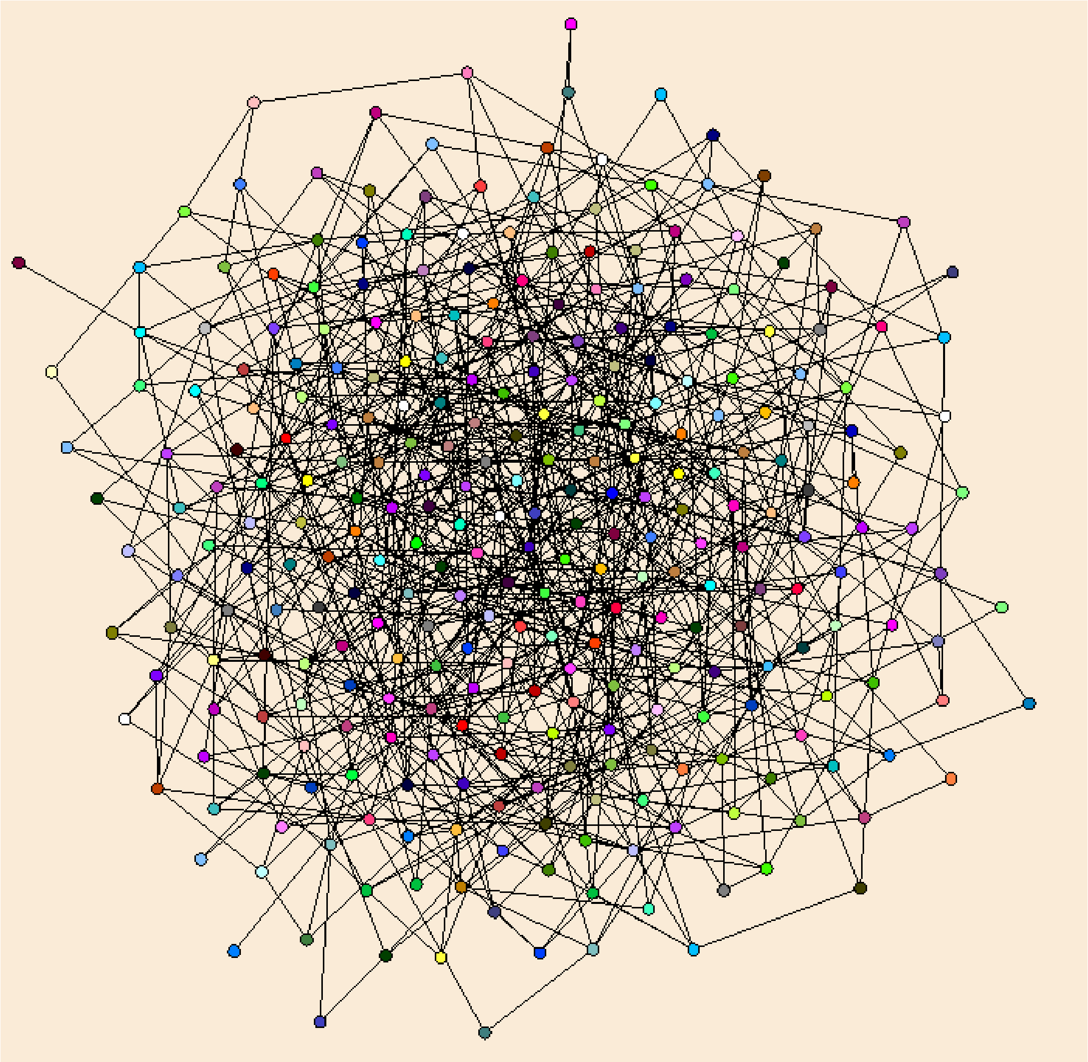
N = 300, M = 200



N = 300, M = 300



N = 300, M = 600



N = 300, M = 900

Threshold properties

- Point at which properties are very likely to hold
  - Giant component
  - All vertices have degree at least one
  - Graph connected
  - Graph has a Hamiltonian path

- Giant component forms at $m = n/2$
- Connectivity occurs at $m = (1/2)n\log n$

Results for Cuckoo Hashing
Low edge density $m \le n/2$

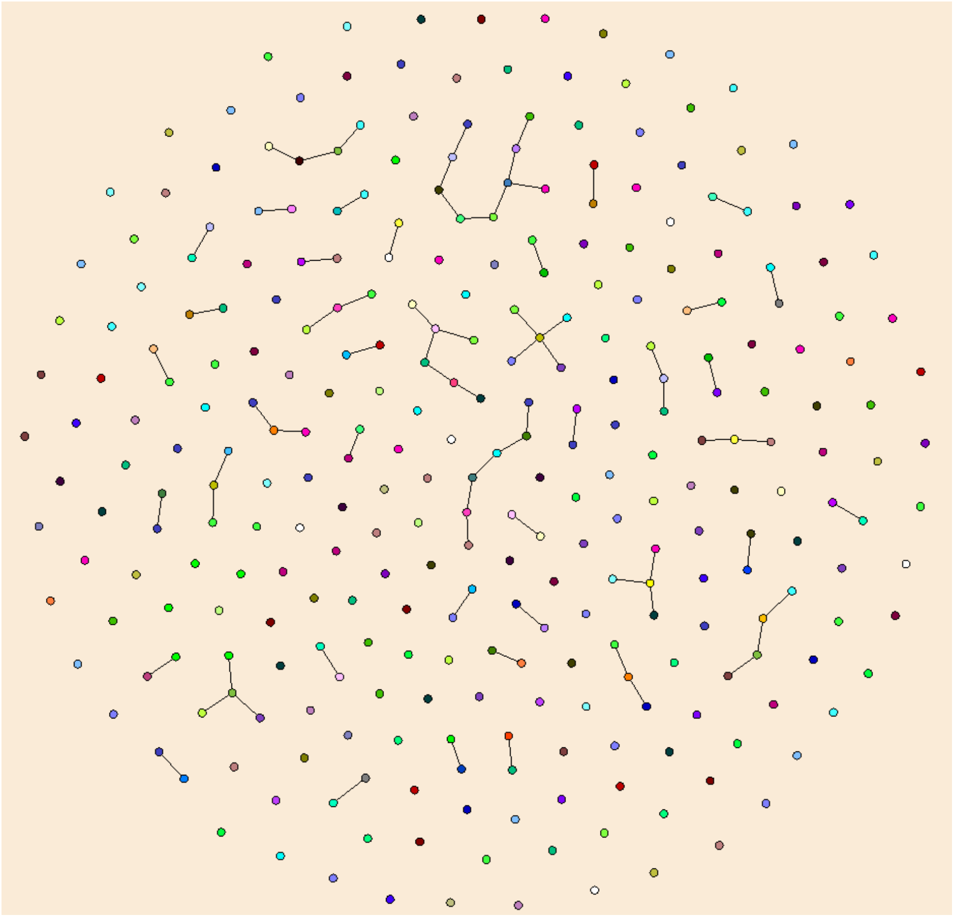
- Let $|S| = K$, use two tables of size 2K each
- Construct a graph where the edges are $(h_1(x_i), h_2(x_i))$ for $i = 1..K$
- This is a random graph on 4K vertices with K edges

If $m \leq cn$, for $c < ½$ the components are trees of size $O(\log n)$

- Mathematical proof based on probabilities of groups of vertices being connected
  - If graph density is sparse, the probability of having enough edges in a set of vertices of size $\alpha \log n$ for it to be connected is small
  - Cycles are unlikely to form
- Intuition is that in early stage of growth, most vertices are isolated, so random edges are unlikely to connect components

If components are small, then there is little chance of creating a bad cycle for hashing

- Bad structure is a cycle with an additional edge coming into the cycle
- Random pairs of vertices (from hash pairs) are unlikely to form this structure until at least ¼ cells are used



Cuckoo Hashing summary

- Table based hashing using two hash functions
- Collision resolution done at insert time with cascades of swaps
  - Timeout at O(log n) steps
  - Expected O(1) time insert
- Finds are O(1) worst case
- Delete are O(1) worst case and easy to do
- Relatively low hash table utilization
- Practical improvements of utilization based on three has functions
- Theoretical analysis based on theory of random graphs