Algorithms for Massive, Expensive, and Otherwise Inconvenient Graphs

DAVID TENCH

UNIVERSITY OF MASSACHUSETTS AMHERST

Ask questions if you’re confused!

Ask questions if you’re confused!

Much of this presentation requires basic CS knowledge only.

Ask questions if you’re confused!

Much of this presentation requires basic CS knowledge only.

I’ll warn you before the tricky parts.

Algorithms for Massive, Expensive, and Otherwise Inconvenient Graphs

DAVID TENCH

UNIVERSITY OF MASSACHUSETTS AMHERST

Convenient Inputs

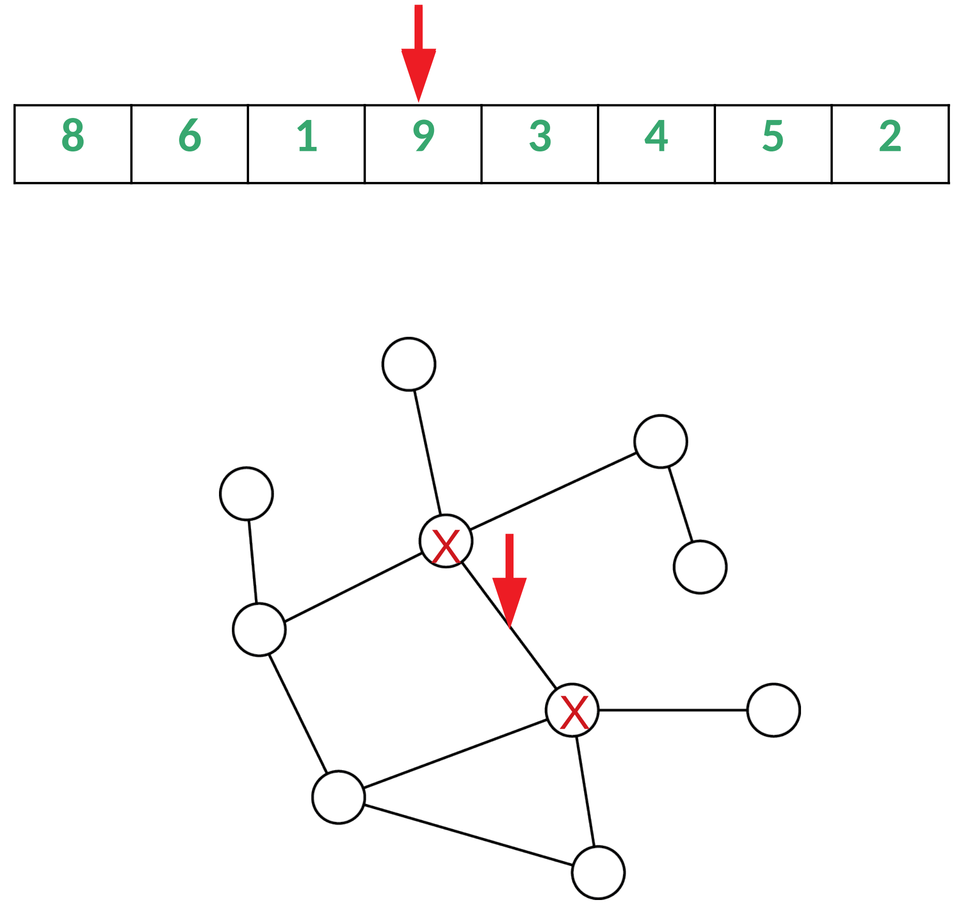
Convenient Inputs

- What is the 4th element of this list?

8	6	1	9	3	4	5	2

Convenient Inputs

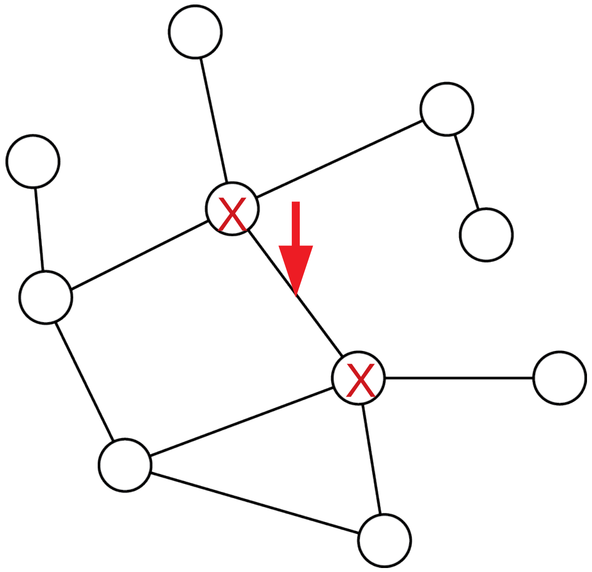
- What is the 4th element of this list?
- Is there an edge between some pair of nodes in this graph?



Convenient Inputs

- What is the 4th element of this list?
- Is there an edge between some pair of nodes in this graph?

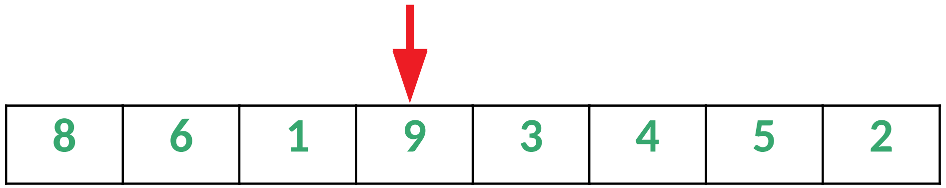
8	6	1	9	3	4	5	2

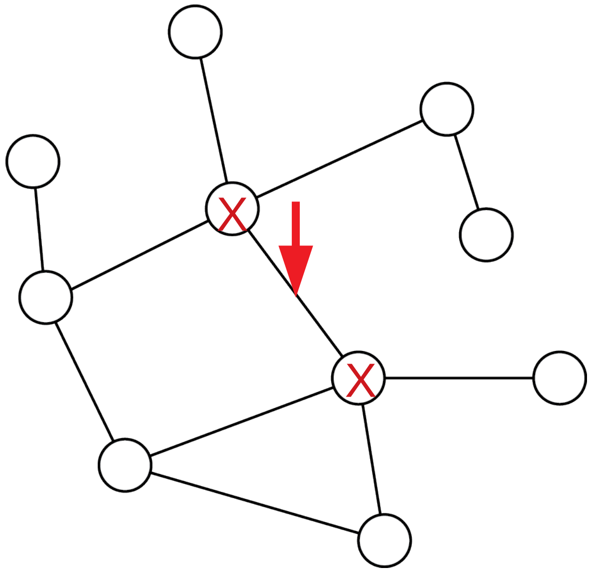


An algorithm can access **any part** of its input at **any time** at **unit cost**.

Convenient Inputs

- What is the 4th element of this list?
- Is there an edge between some pair of nodes in this graph?





An algorithm can access **any part** of its input at **any time** at **unit cost**.

This is the *random access property*.

When Random Access Fails

When Random Access Fails

When inputs are too large
to fit in memory

When Random Access Fails

When inputs are too large
to fit in memory

When Random Access Fails

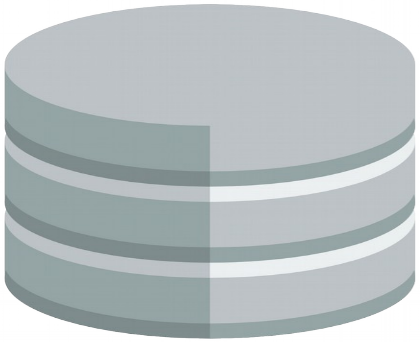
When inputs are too large to fit in memory

When parts of inputs are costly to discover

When Random Access Fails

When inputs are too large to fit in memory

When parts of inputs are costly to discover



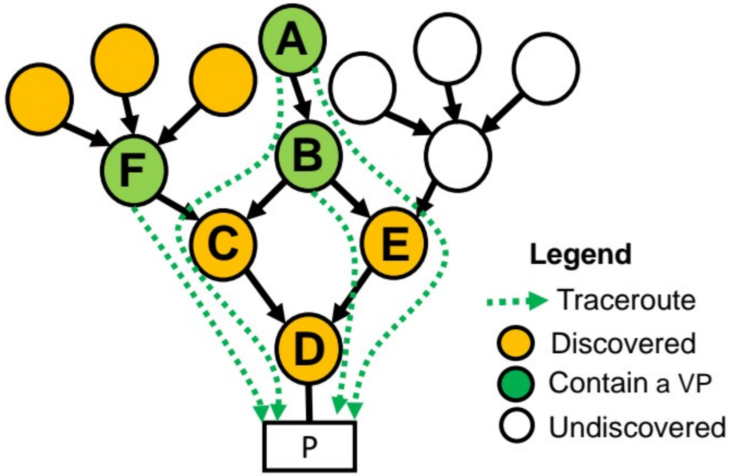
Overview of this talk

Streaming graph algorithms:
Coping with graphs too large to fit in memory

Overview of this talk

Streaming graph algorithms: Coping with graphs too large to fit in memory

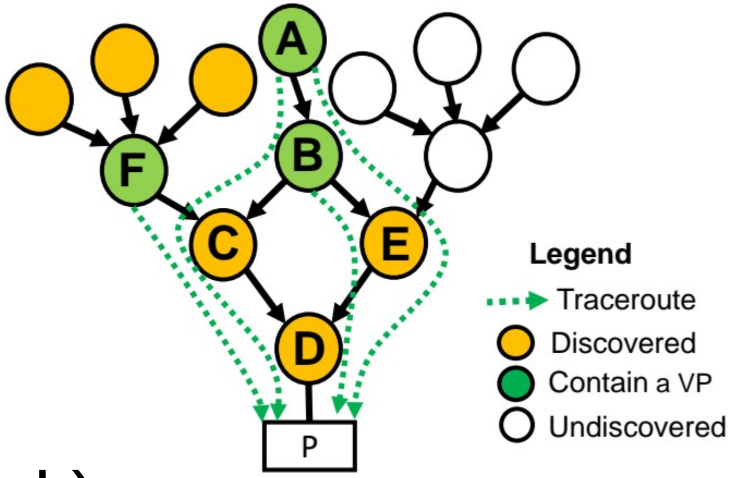
Collaborations with practitioners: Graph algorithms subject to expensive edge queries



Overview of this talk

Streaming graph algorithms: Coping with graphs too large to fit in memory

Collaborations with practitioners: Graph algorithms subject to expensive edge queries



(Also future work)

Streaming

COMPUTING WITH INCREDIBLY LARGE INPUTS

When Graphs Are Too Large

Can’t store graph in memory

When Graphs Are Too Large

Can’t store graph in memory

Receive *stream* of edges

When Graphs Are Too Large

Can't store graph in memory

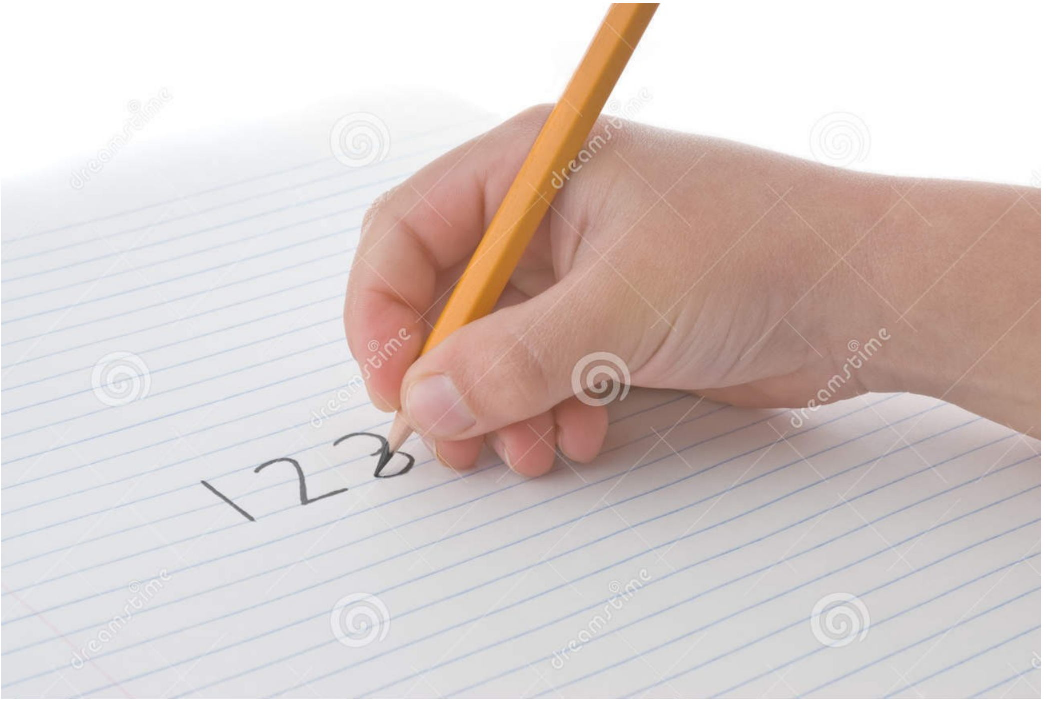
Receive *stream* of edges

- Vertex Connectivity (PODS 2015)
- Densest Subgraph (MFCS 2015)
- Unique Cover (current work)

Warm-up: Missing Number

I read you an unordered list of all the integers from 1 to 5 million – except I leave one of them out. After, I ask you which is missing.

You only have a single piece of paper and a pencil. How do you find the missing number?

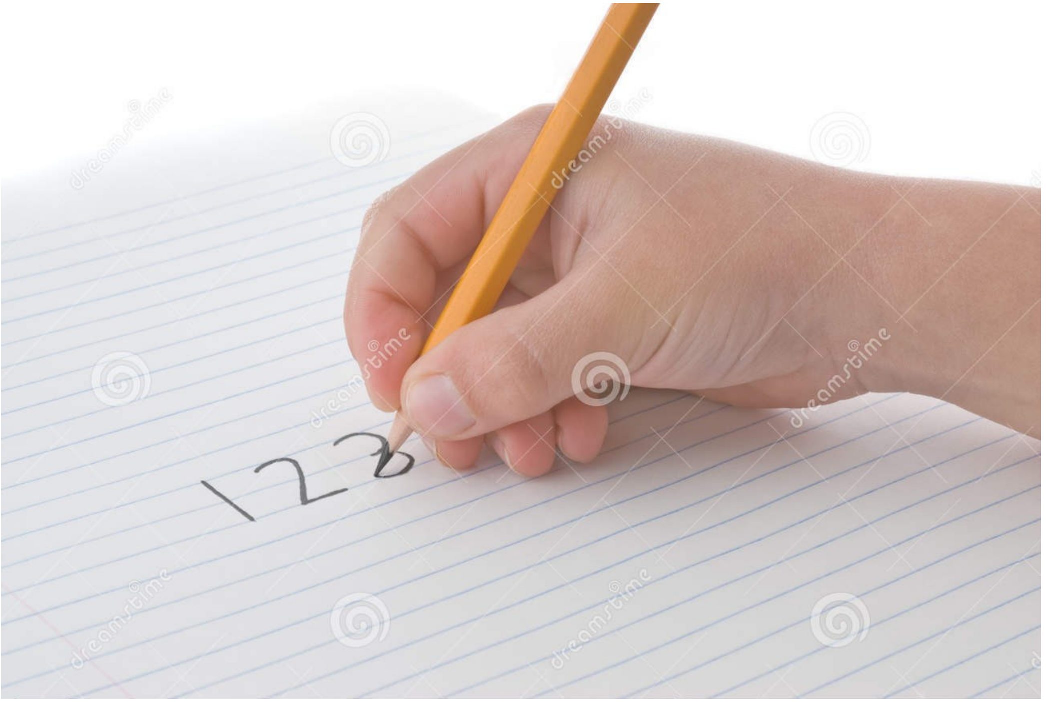


Warm-up: Missing Number

I read you an unordered list of all the integers from 1 to 5 million – except I leave one of them out. After, I ask you which is missing.

You only have a single piece of paper and a pencil. How do you find the missing number?

Answer: keep a running sum of all the numbers, then subtract from 1 + 2 + … + 5 million.



Graph Streaming

Graph Streaming

Facebook has almost 2 billion users

Graph Streaming

Facebook has almost 2 billion users

Nodes = users, edges = friend relationships

Graph Streaming

Facebook has almost 2 billion users

Nodes = users, edges = friend relationships

This graph could have almost
(2 billion)^2 = 4 quintillion edges

Graph Streaming

Facebook has almost 2 billion users

Nodes = users, edges = friend relationships

This graph could have almost $(2 \text{ billion})^2$ = 4 quintillion edges

For modern computers, storing 2 billion objects is maybe reasonable but 4 quintillion is not

Graph Streaming

It’s possible to store some of the edges, but not all

Graph Streaming

It's possible to store some of the edges, but not all

We have roughly n space, where n = # of nodes

Graph Streaming

It’s possible to store some of the edges, but not all

We have roughly n space, where n = # of nodes

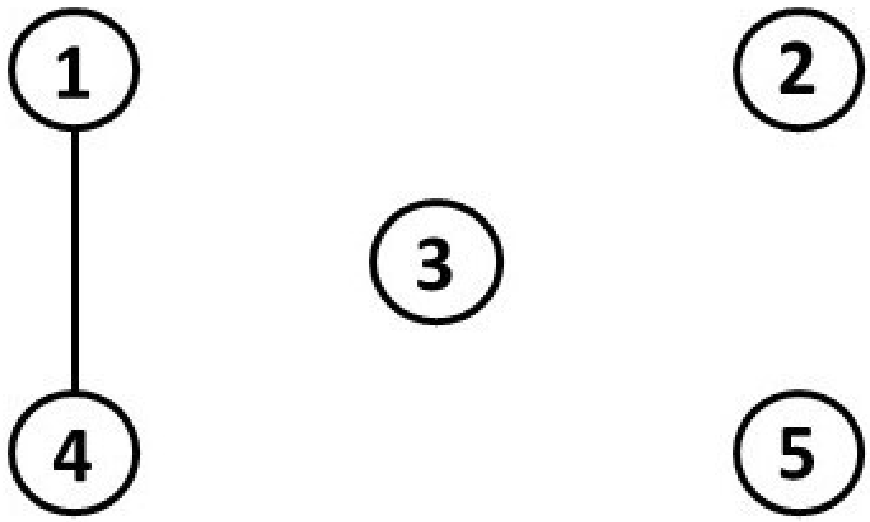
Graph edges are given as a *stream*

Defining a graph via a stream

Defining a graph via a stream

Add edge (1,4)

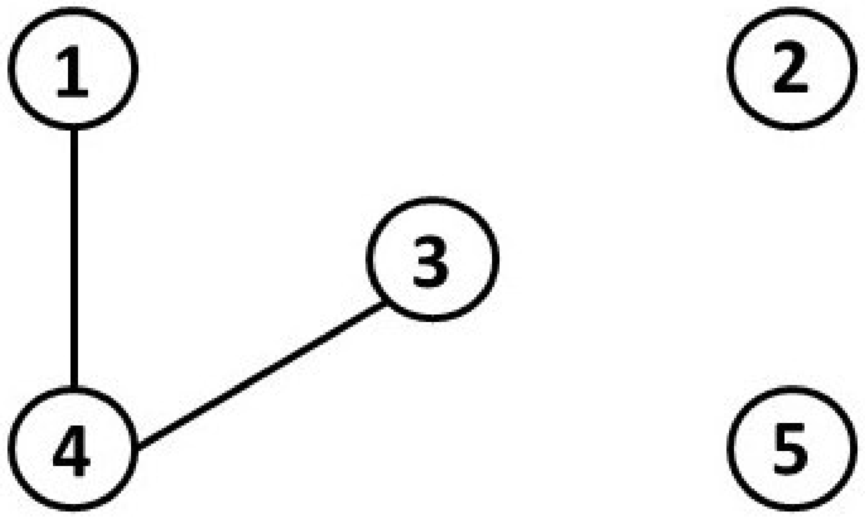
Defining a graph via a stream



Defining a graph via a stream

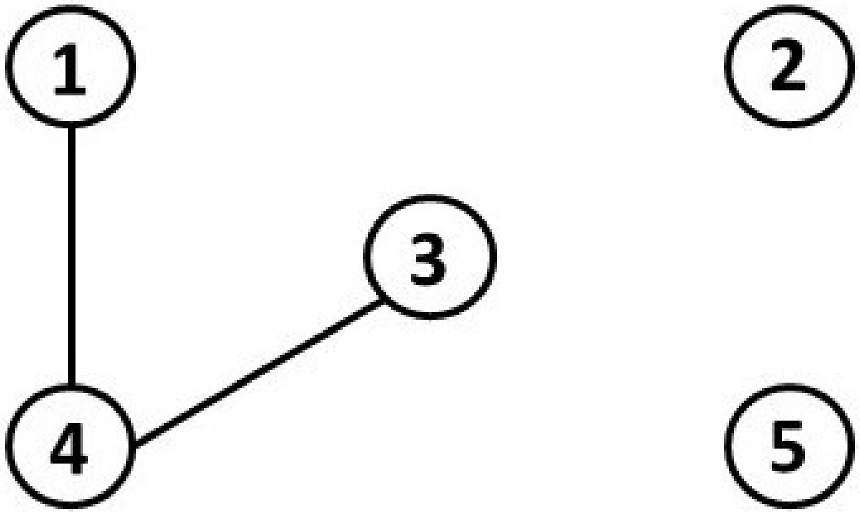
Add edge (3,4)

Defining a graph via a stream

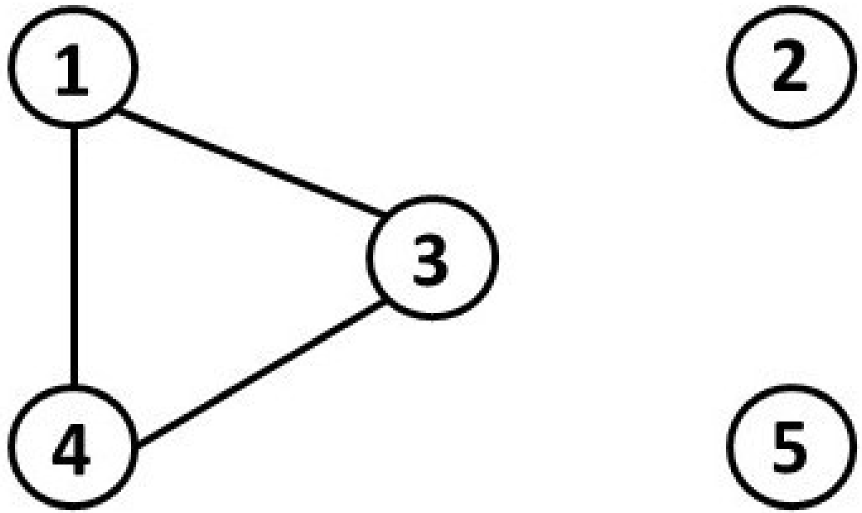


Defining a graph via a stream

Add edge (1,3)

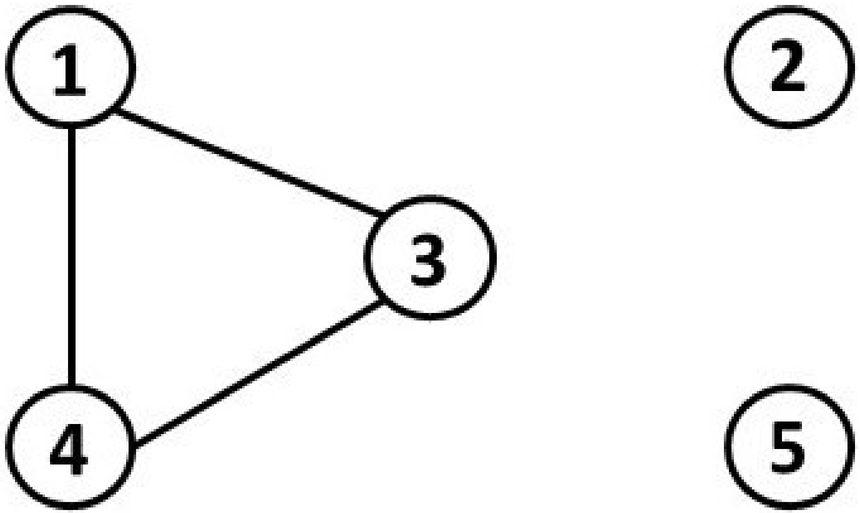


Defining a graph via a stream

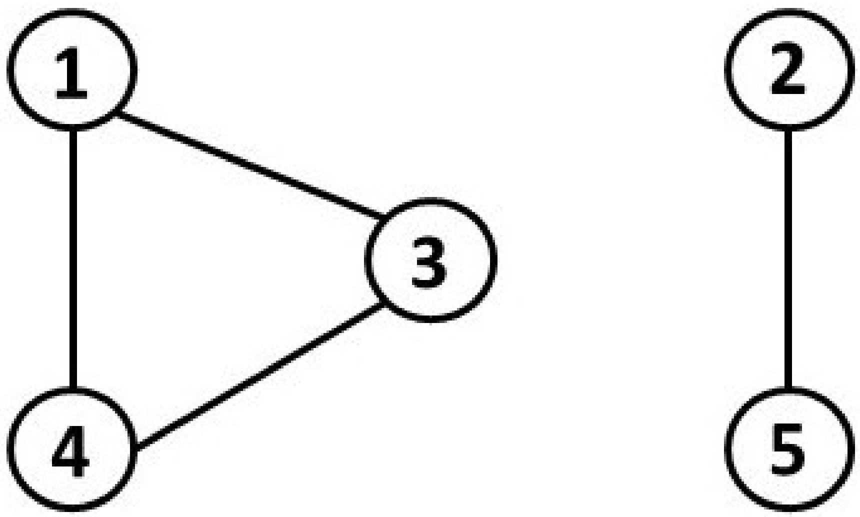


Defining a graph via a stream

Add edge (2,5)



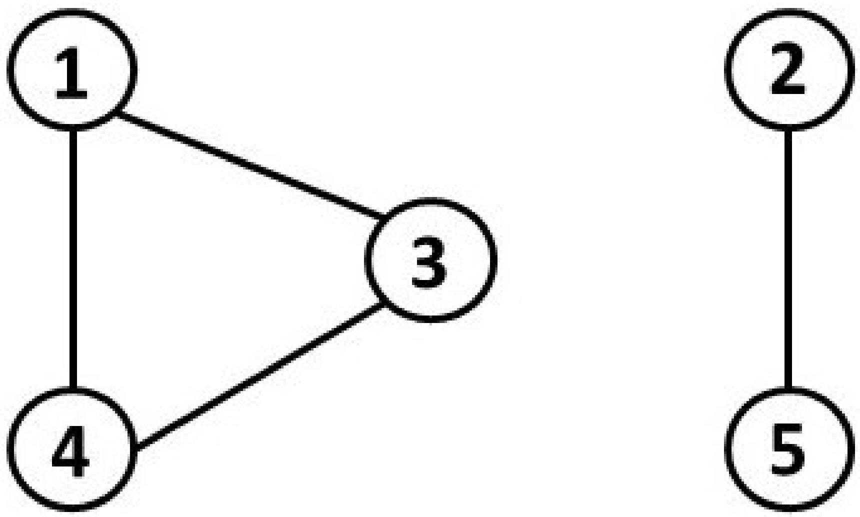
Defining a graph via a stream



Defining a graph via a stream

Stream:
Add edge (1,4)
Add edge (3,4)
Add edge (1,3)
Add edge (2,5)

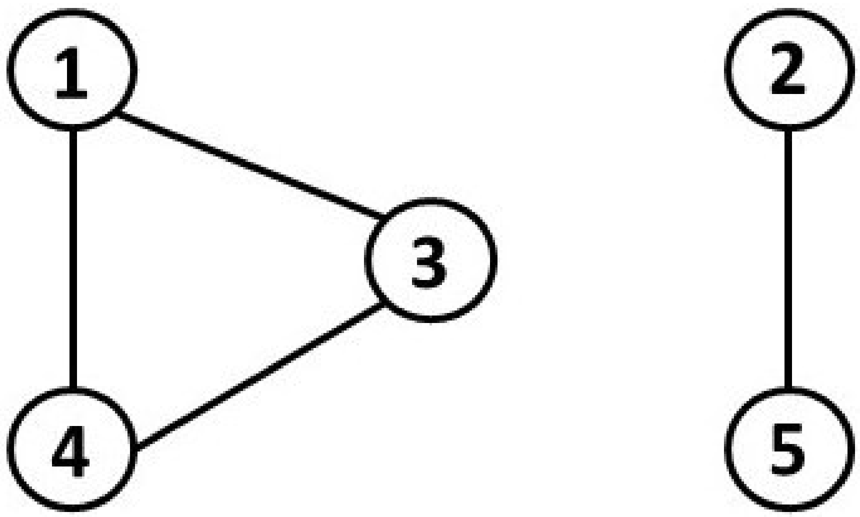
Resulting Graph:



Defining a graph via a stream

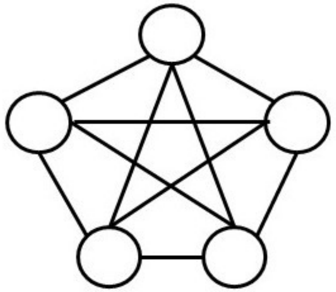
Stream:
Add edge (1,4)
Add edge (3,4)
Add edge (1,3)
Add edge (2,5)

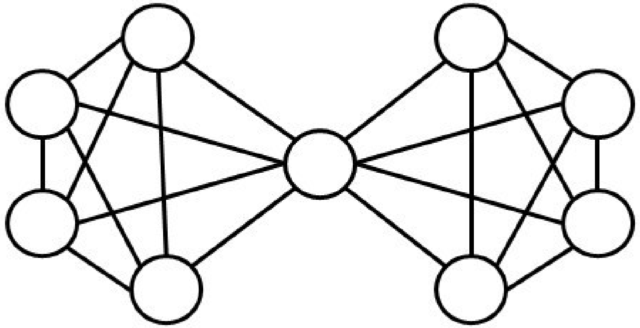
Resulting Graph:



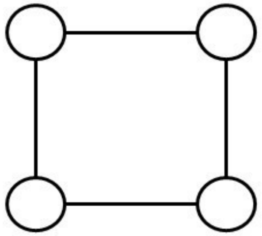
Edge deletions are also possible, but we’re ignoring them today.

Problem: Vertex Connectivity

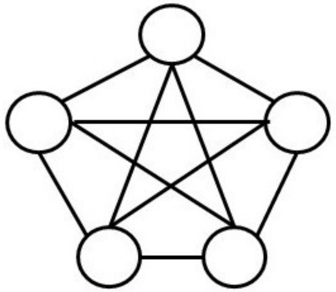


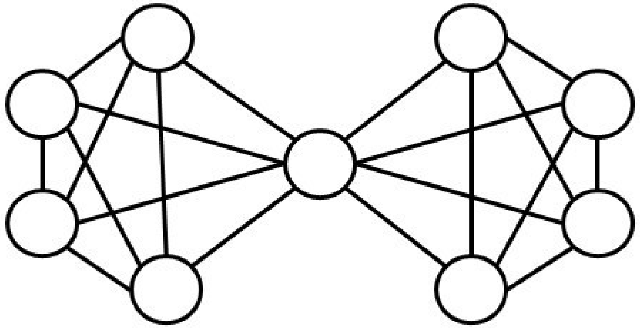


What is the minimum number of nodes we can remove to disconnect a graph G?

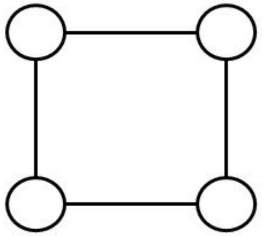


Problem: Vertex Connectivity



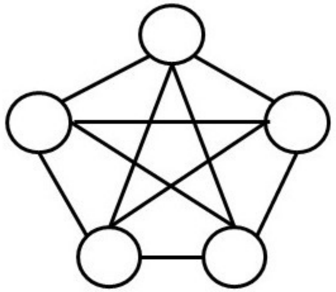


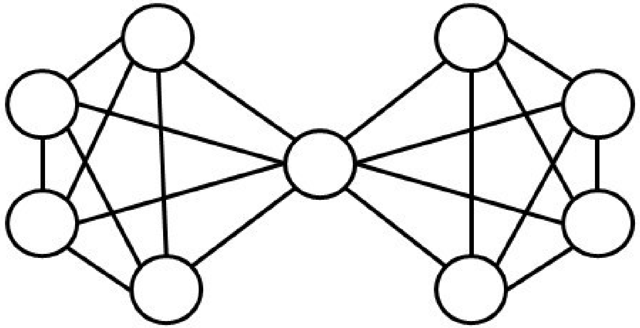
1



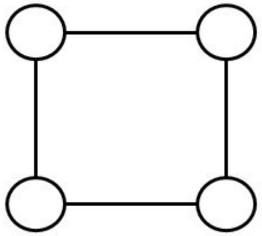
What is the minimum number of nodes we can remove to disconnect a graph G?

Problem: Vertex Connectivity





1

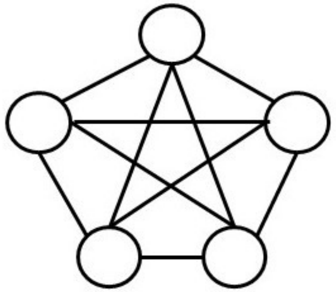


2

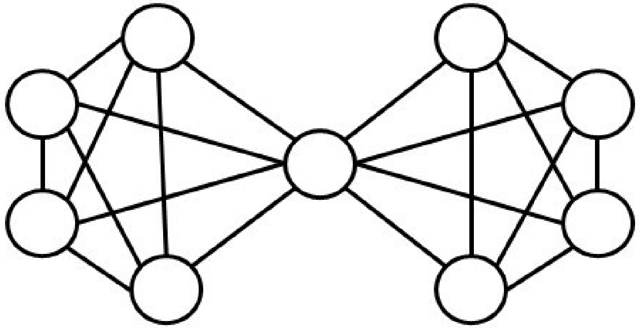
What is the minimum number of nodes we can remove to disconnect a graph G?

Problem: Vertex Connectivity

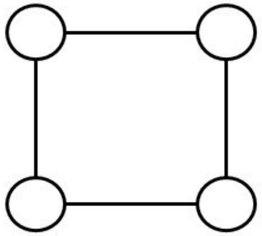
What is the minimum number of nodes we can remove to disconnect a graph G?



5

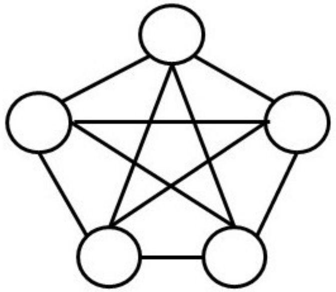


1

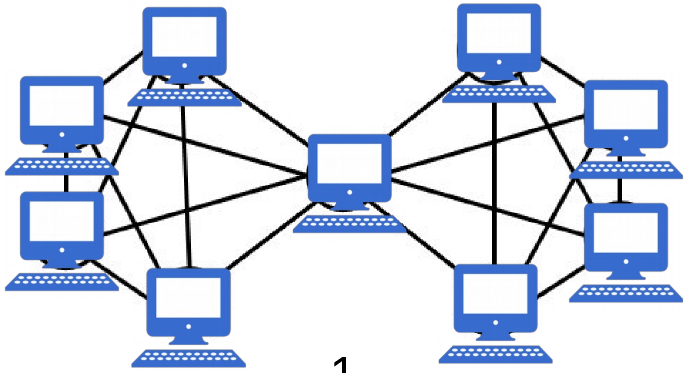


2

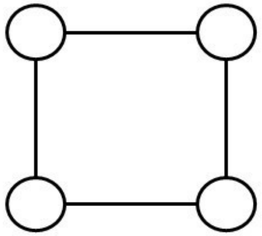
Problem: Vertex Connectivity



5



1

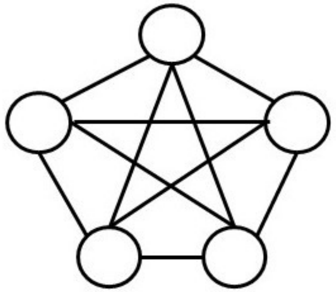


2

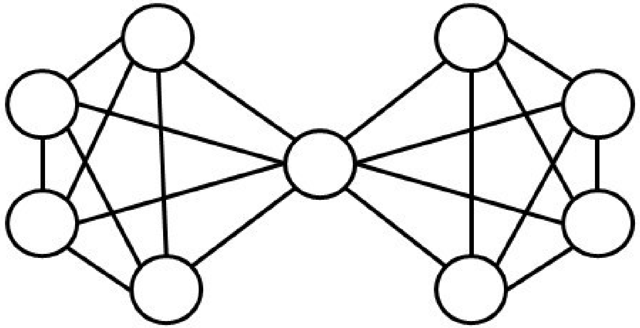
What is the minimum number of nodes we can remove to disconnect a graph G?

How many computers in a network can fail before the remaining network is disconnected?

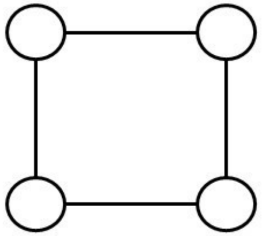
Problem: Vertex Connectivity



5



1

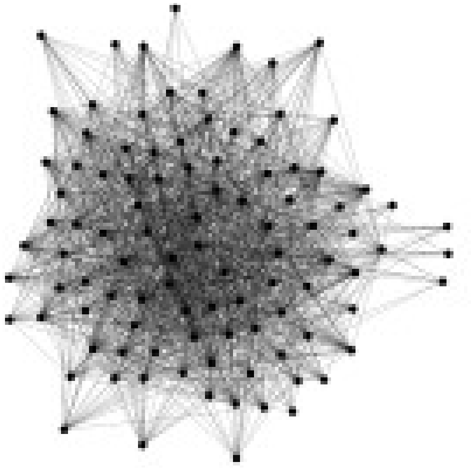


2

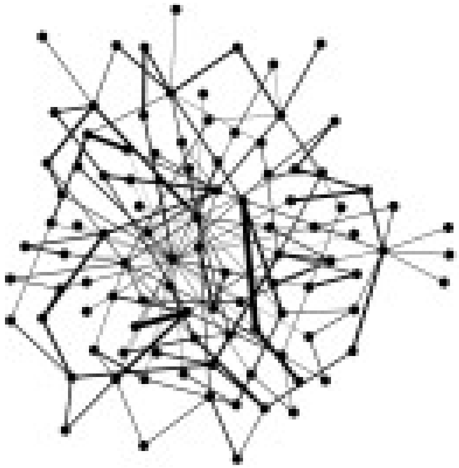
What is the minimum number of nodes we can remove to disconnect a graph G?

Easily solvable using max flow algorithm, but this requires random access to the graph.

Problem: Vertex Connectivity



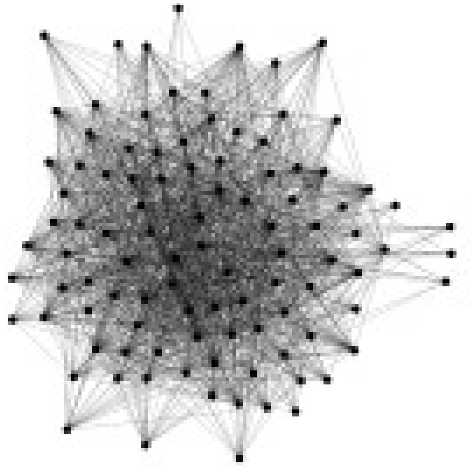
G



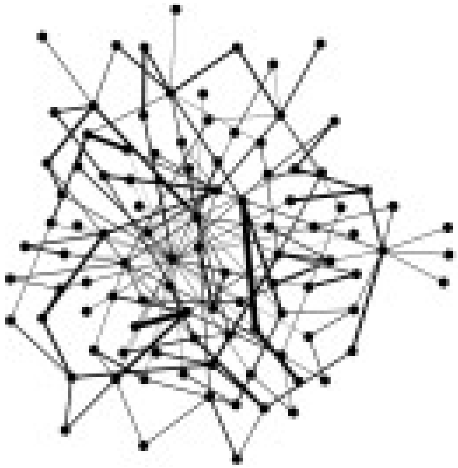
H

We show how to create a *certificate* graph H that matches G’s vertex connectivity up to constant k, but has only roughly kn edges.

Problem: Vertex Connectivity



G

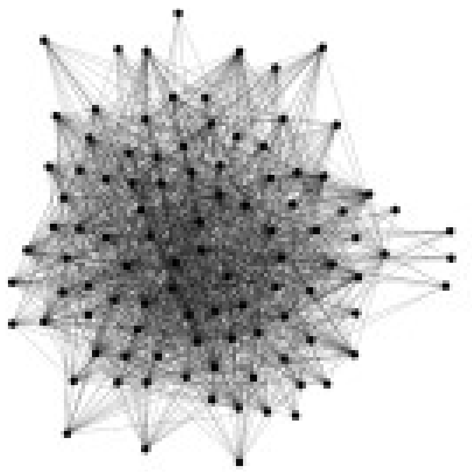


H

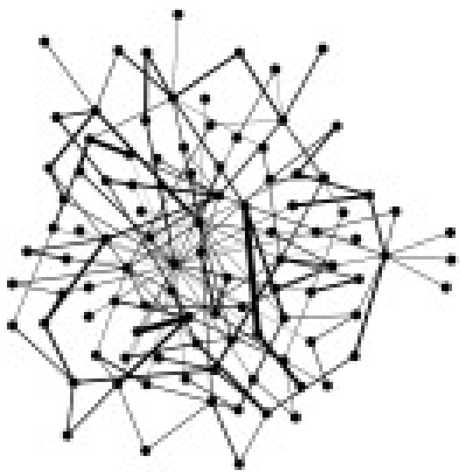
We show how to create a *certificate* graph H that matches G's vertex connectivity up to constant k, but has only kn edges.

We also show how to (1+ϵ)-approximate vertex connectivity while only storing ϵ^{-1} kn edges.

Problem: Vertex Connectivity



G



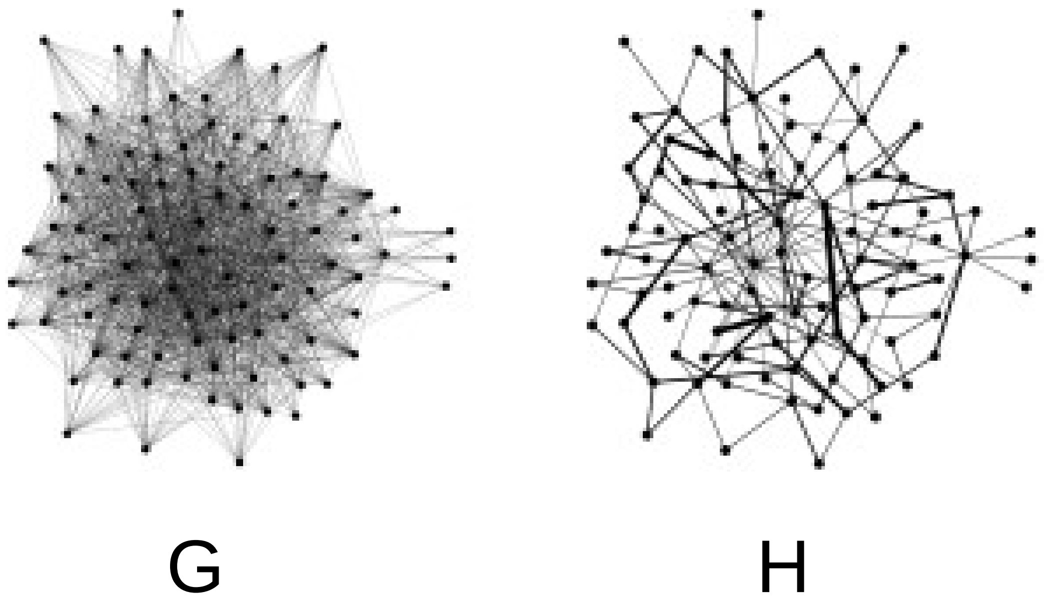
H

We show how to create a *certificate* graph H that matches G's vertex connectivity up to constant k, but has only kn edges.

We also show how to (1+ϵ)-approximate vertex connectivity while only storing ϵ^{-1} kn edges.

Finally, we show how to construct hypergraph sparsifiers in roughly linear space.

Problem: Vertex Connectivity



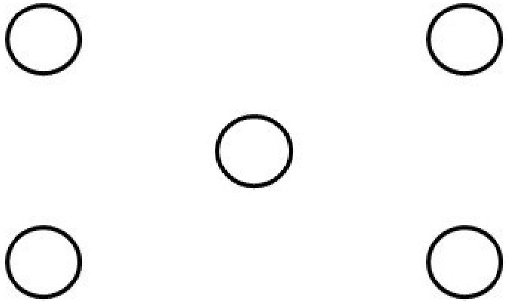
We show how to create a *certificate* graph H that matches G's vertex connectivity up to constant k, but has only kn edges.

We also show how to $(1+\epsilon)$-approximate vertex connectivity while only storing ϵ^{-1} kn edges.

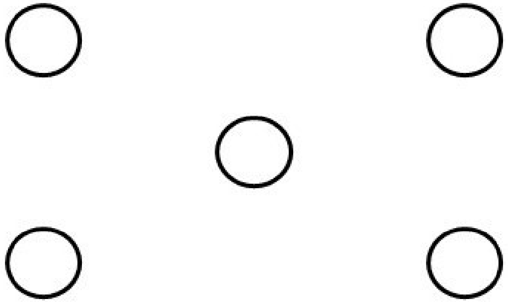
Finally, we show how to construct hypergraph sparsifiers in roughly linear space.

Warm-up: Is the Graph Connected?

Original graph

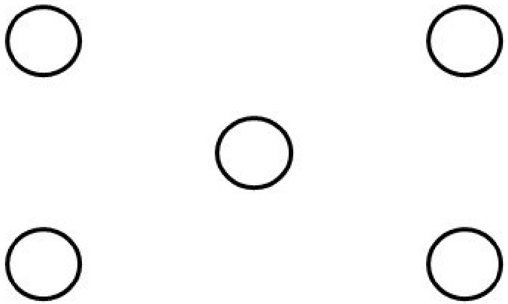


Summary

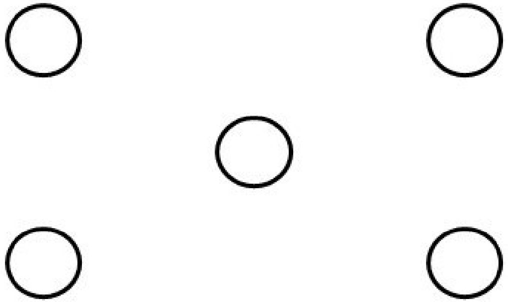


Warm-up: Is the Graph Connected?

Original graph



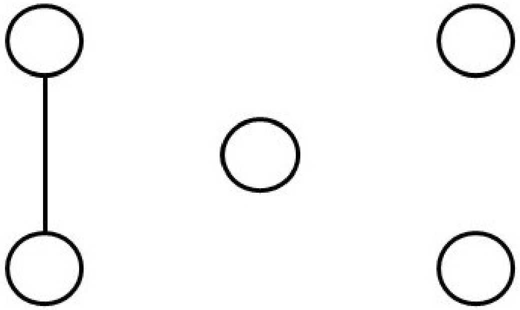
Summary



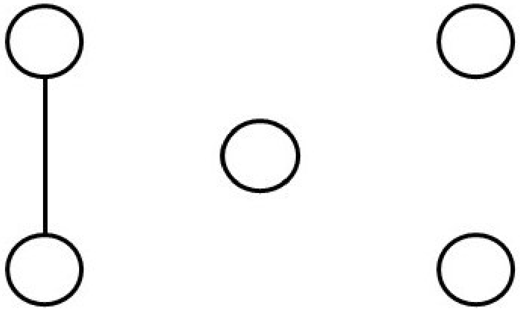
As each edge arrives in the stream, we keep it only if its endpoints were not already connected.

Warm-up: Is the Graph Connected?

Original graph



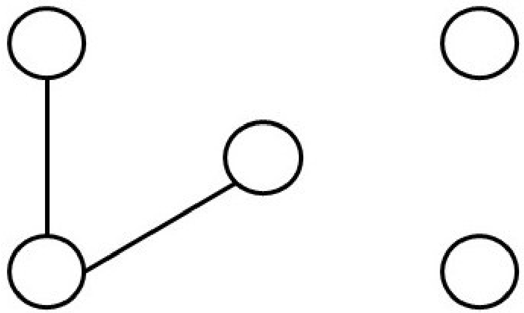
Summary



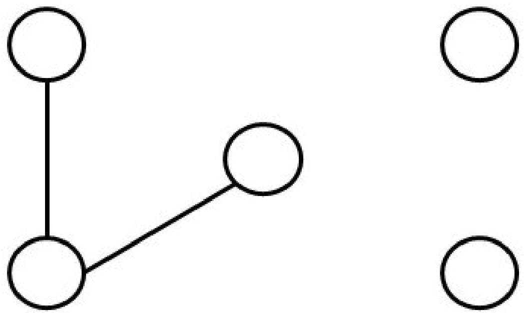
As each edge arrives in the stream, we keep it only if its endpoints were not already connected.

Warm-up: Is the Graph Connected?

Original graph



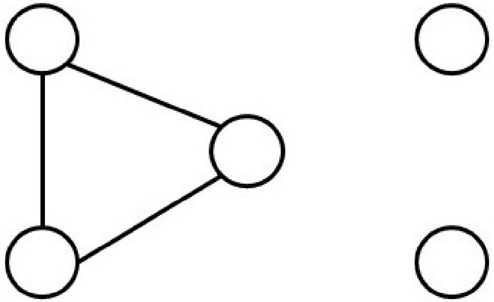
Summary



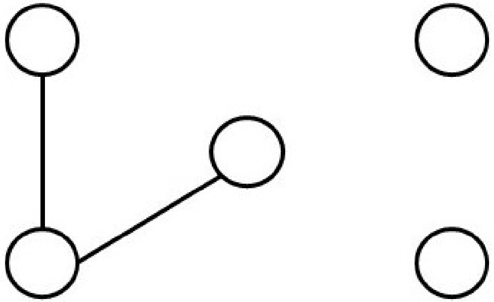
As each edge arrives in the stream, we keep it only if its endpoints were not already connected.

Warm-up: Is the Graph Connected?

Original graph

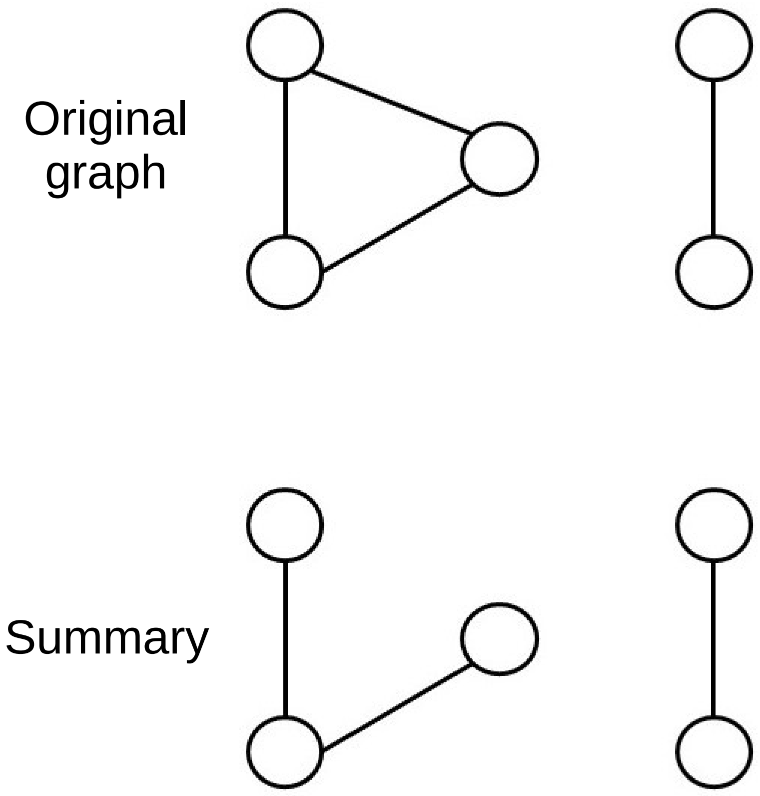


Summary



As each edge arrives in the stream, we keep it only if its endpoints were not already connected.

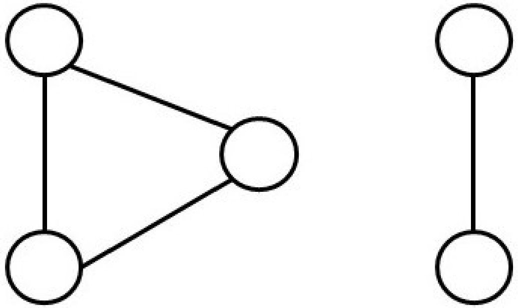
Warm-up: Is the Graph Connected?



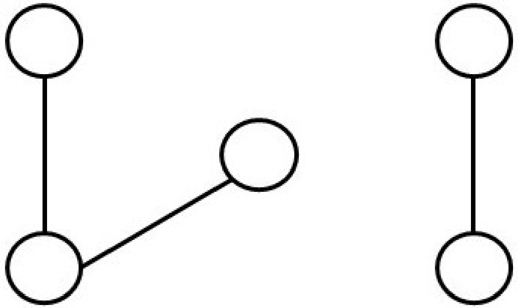
As each edge arrives in the stream, we keep it only if its endpoints were not already connected.

Warm-up: Is the Graph Connected?

Original graph



Spanning Forest

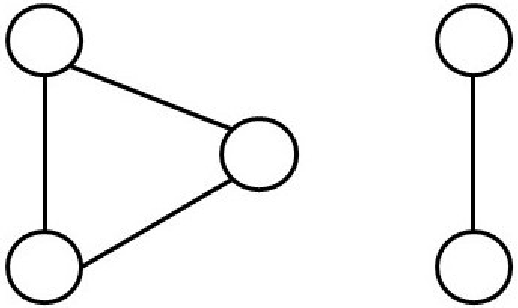


As each edge arrives in the stream, we keep it only if its endpoints were not already connected.

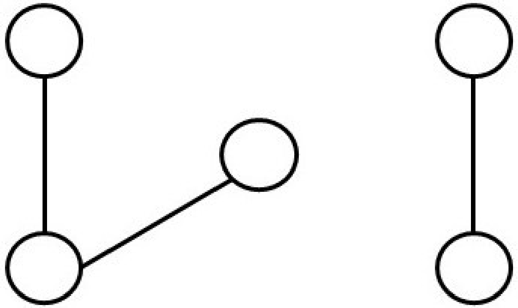
This is a **spanning forest** and has at most n-1 edges.

Warm-up: Is the Graph Connected?

Original graph



Spanning Forest

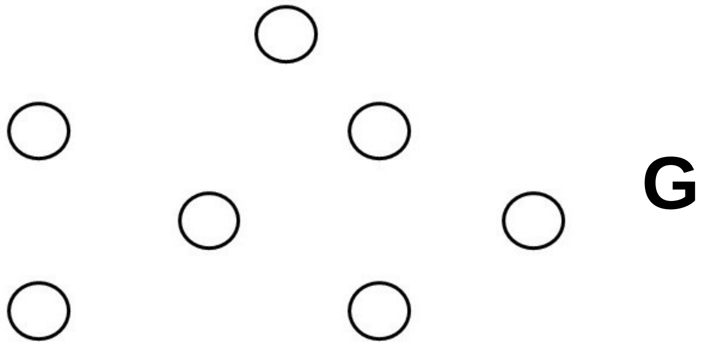


As each edge arrives in the stream, we keep it only if its endpoints were not already connected.

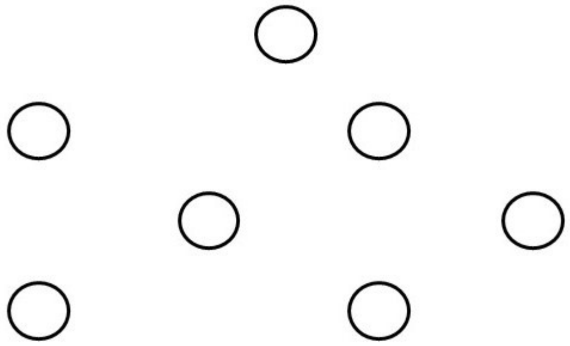
This is a **spanning forest** and has at most n-1 edges.

If the spanning forest is connected, we know the original graph was as well.

Back to Vertex Connectivity



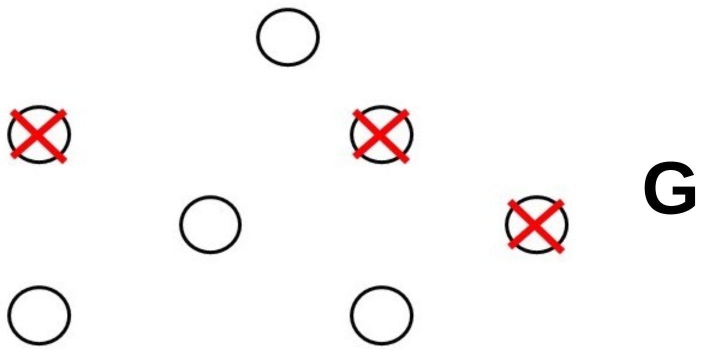
Back to Vertex Connectivity



G

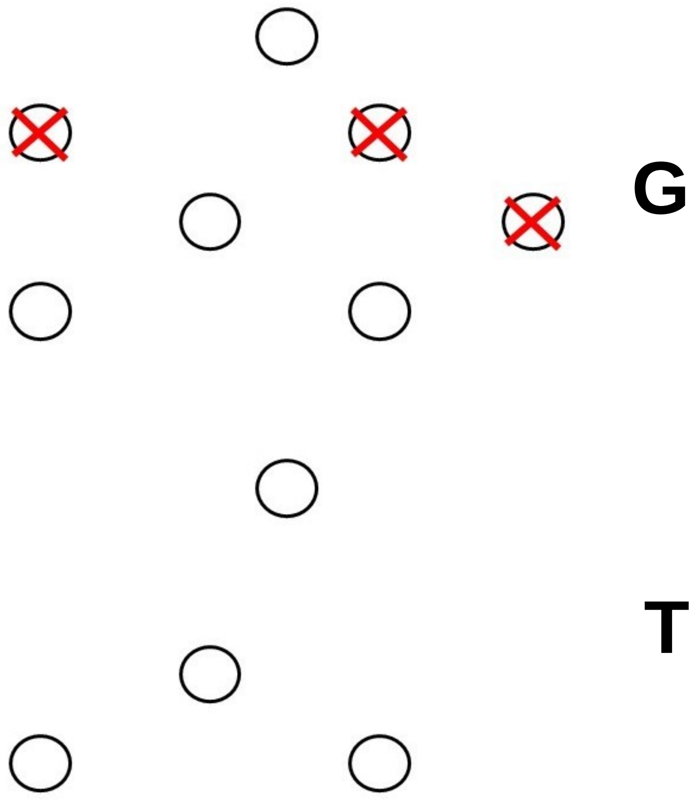
First we delete each node in G with probability 1-1/k.

Back to Vertex Connectivity



First we delete each node in G with probability 1-1/k.

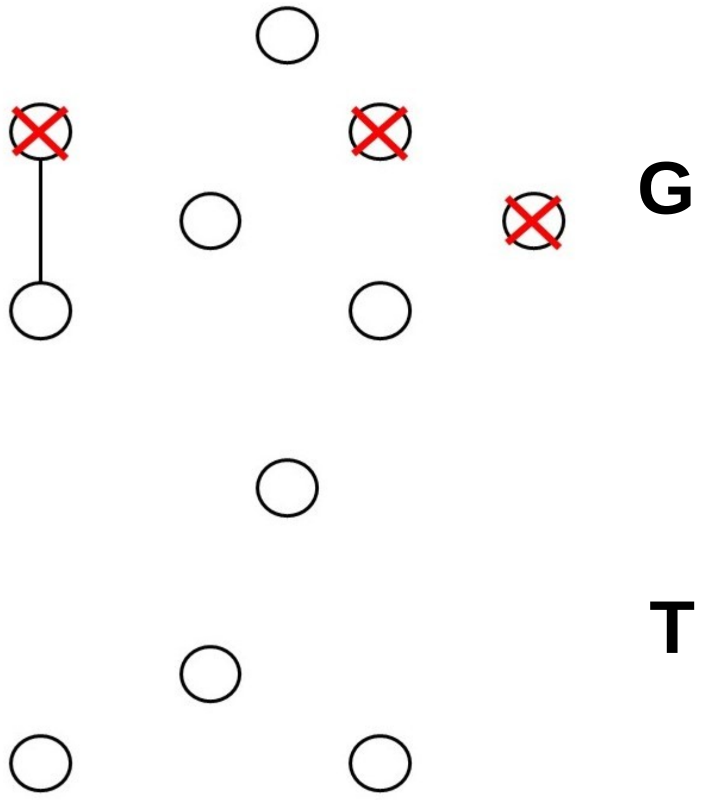
Back to Vertex Connectivity



First we delete each node in G with probability 1-1/k.

During the stream we maintain a spanning forest T on the remaining nodes.

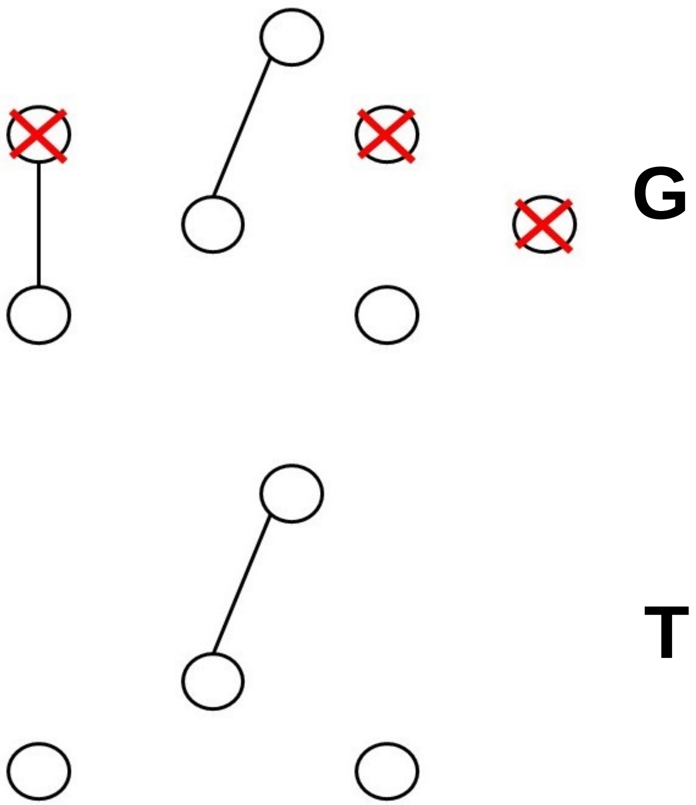
Back to Vertex Connectivity



First we delete each node in G with probability 1-1/k.

During the stream we maintain a spanning forest T on the remaining nodes.

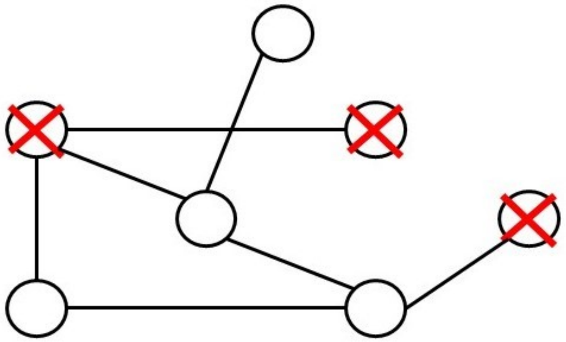
Back to Vertex Connectivity



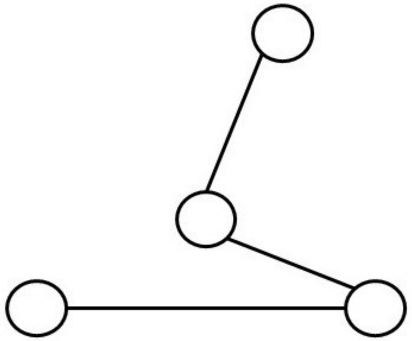
First we delete each node in G with probability 1-1/k.

During the stream we maintain a spanning forest T on the remaining nodes.

Back to Vertex Connectivity



G

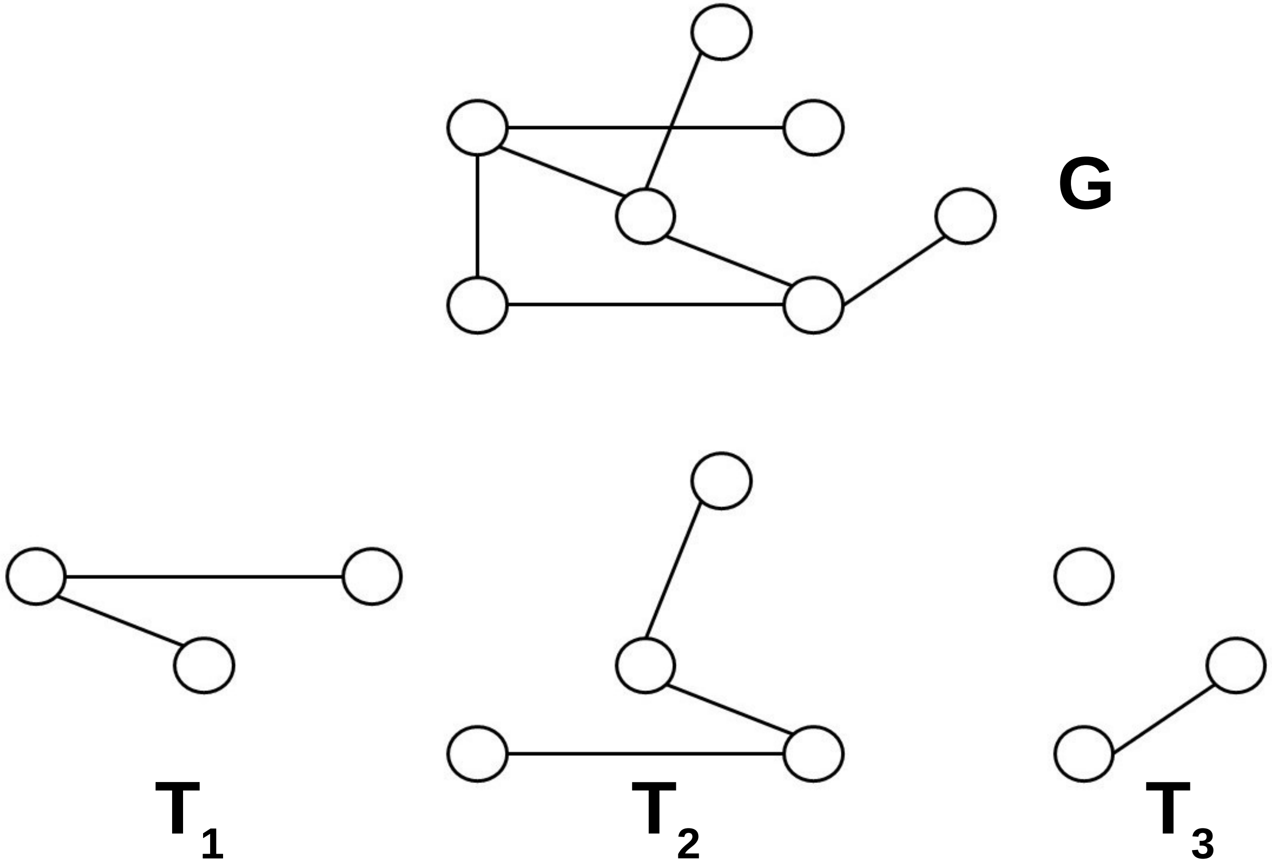


T

First we delete each node in G with probability 1-1/k.

During the stream we maintain a spanning forest T on the remaining nodes.

Back to Vertex Connectivity

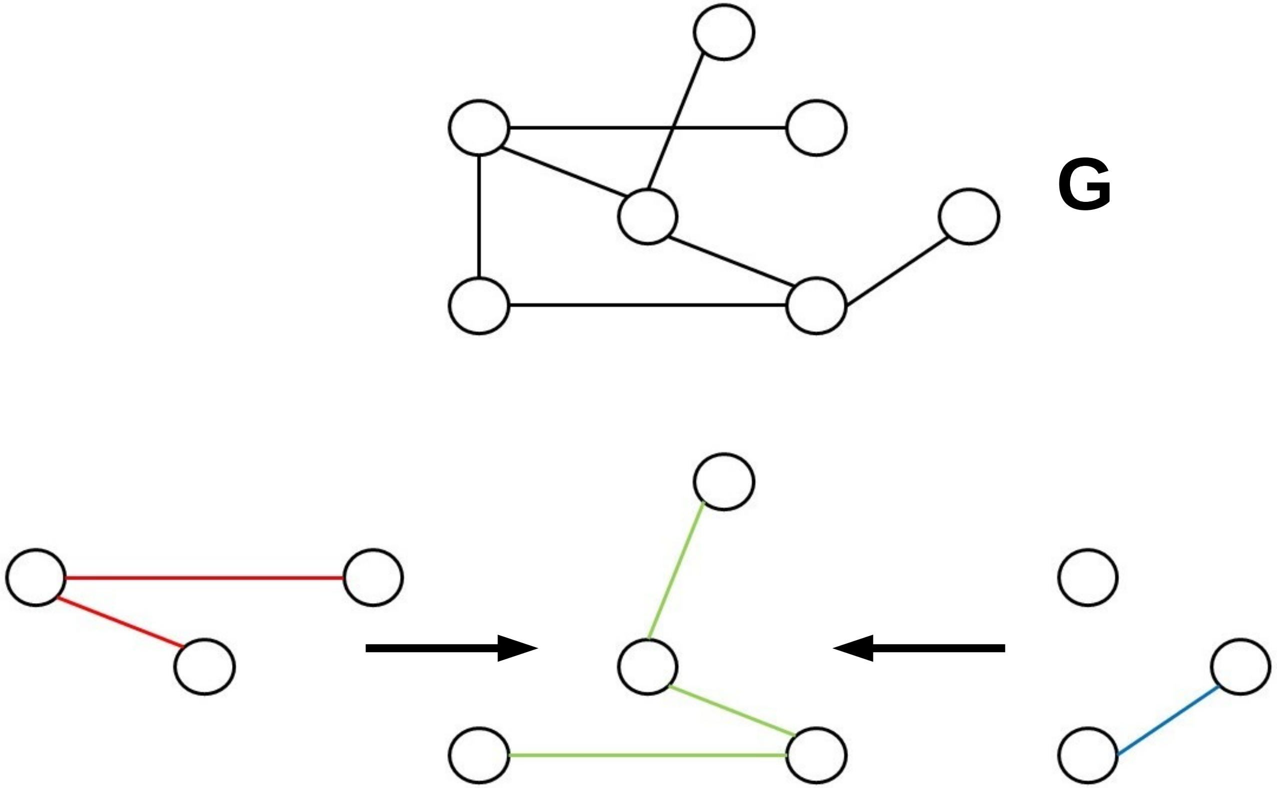


First we delete each node in G with probability 1-1/k.

During the stream we maintain a spanning forest T on the remaining nodes.

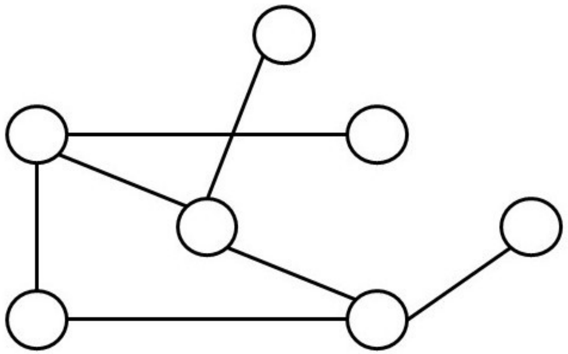
Repeat roughly k^2 times in parallel* to make T_1, T_2, …

Back to Vertex Connectivity

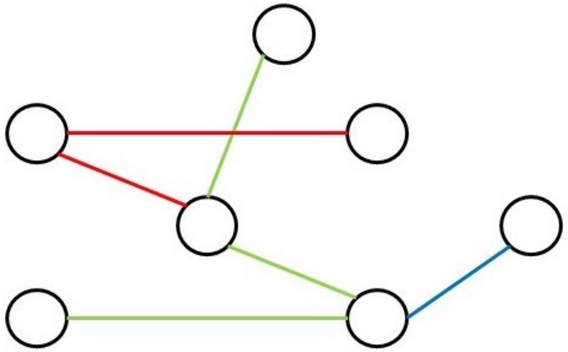


After the stream, merge all T_i to form H, our vertex connectivity certificate.

Back to Vertex Connectivity



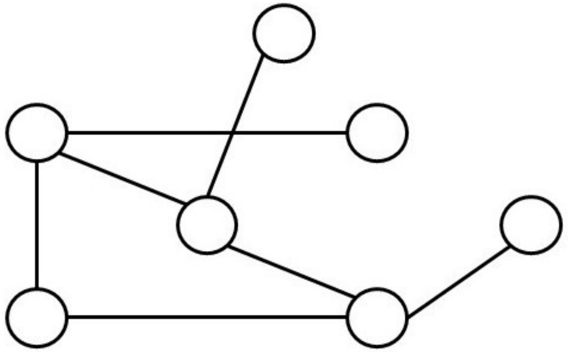
G



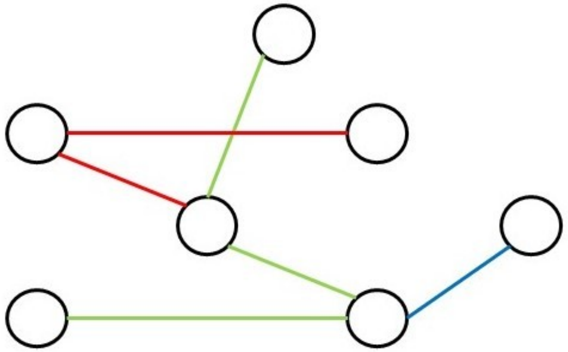
H

After the stream, merge all T_i to form H, our vertex connectivity certificate.

Back to Vertex Connectivity



G

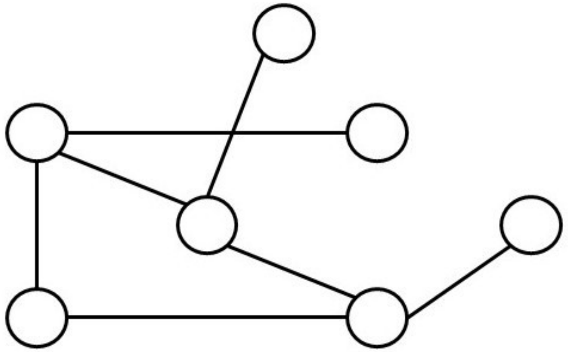


H

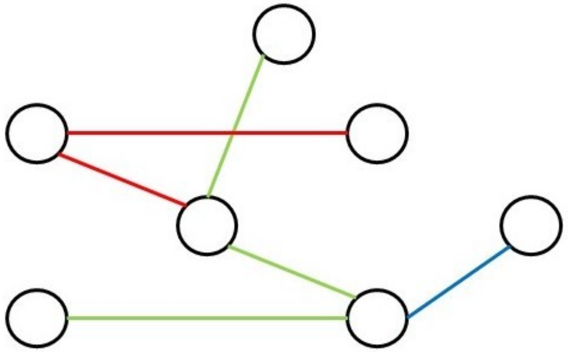
After the stream, merge all T_i to form H, our vertex connectivity certificate.

Each spanning forest has n/k edges in expectation and there are k^2 of them so H has kn edges in expectation.

Back to Vertex Connectivity



G



H

After the stream, merge all T_i to form H, our vertex connectivity certificate.

Theorem: For some arbitrary set S of at most k nodes, G\S is disconnected iff H\S is disconnected.

Proof Sketch of Theorem

Theorem: For some arbitrary set S of at most k nodes, G\S is disconnected iff H\S is disconnected.

Proof Sketch of Theorem

To prove: When G\S is disconnected, H\S must be disconnected

To prove: When G\S is connected, H\S must be connected

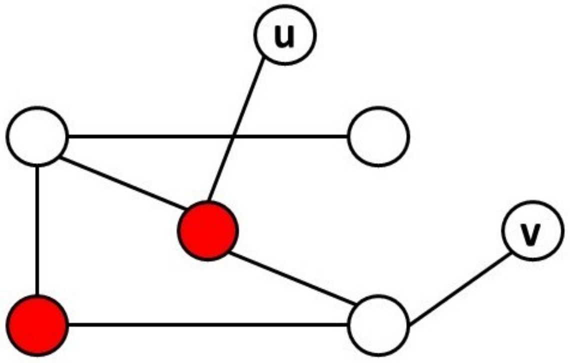
Proof Sketch of Theorem

To prove: When G\S is disconnected, H\S must be disconnected

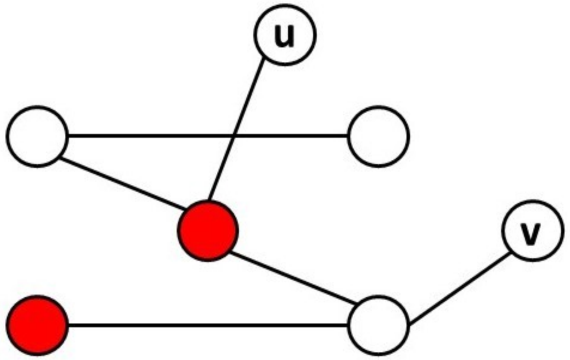
To prove: When G\S is connected, H\S must be connected

Proof Sketch of Theorem

To prove: When G\S is disconnected, H\S must be disconnected



G

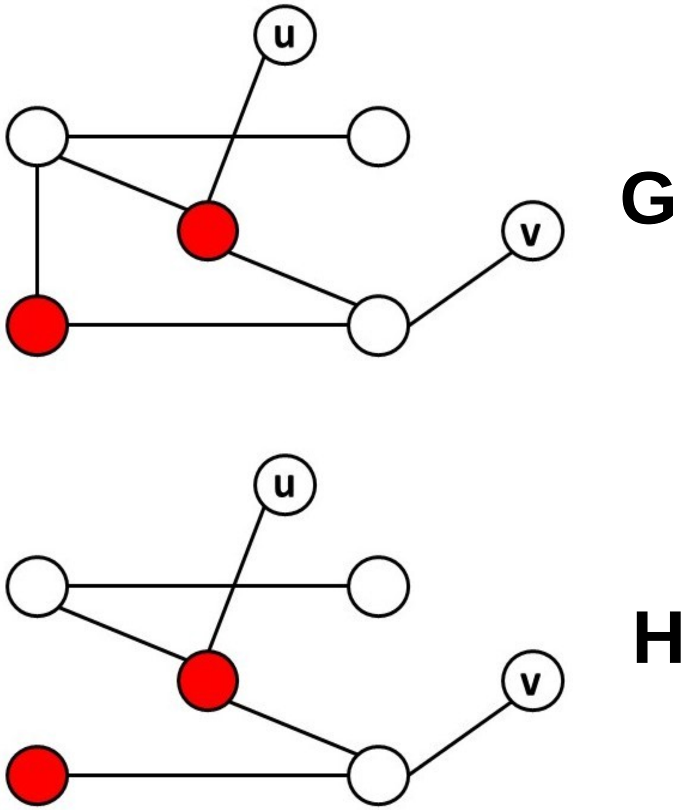


H

Proof Sketch of Theorem

To prove: When G\S is disconnected, H\S must be disconnected

H is a subgraph of G, so if G\S lacks a path between nodes u and v, so does H\S.

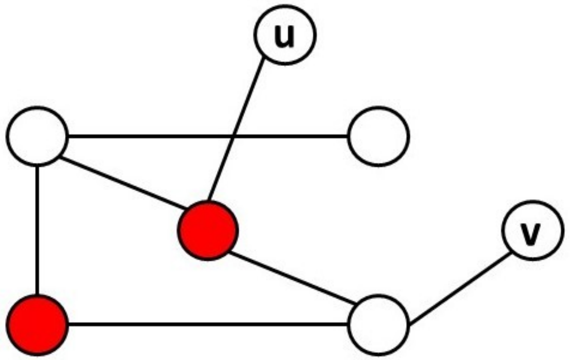


Proof Sketch of Theorem

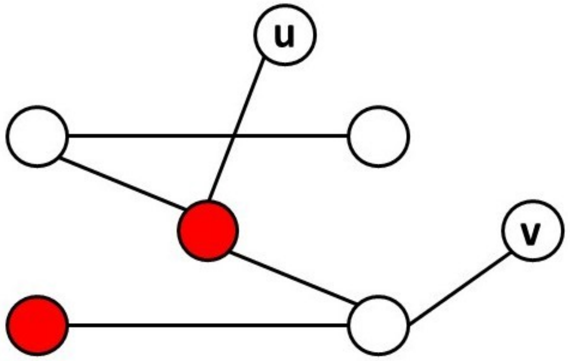
To prove: When G\S is disconnected, H\S must be disconnected

H is a subgraph of G, so if G\S lacks a path between nodes u and v, so does H\S.

With high probability, H has all of G's nodes. //



G



H

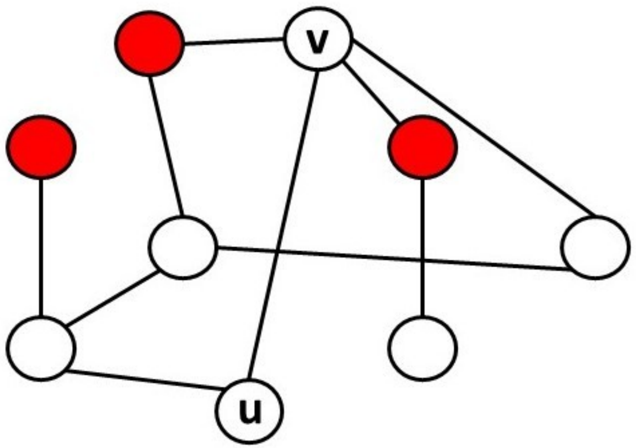
Proof Sketch of Theorem

To prove: When G\S is connected, H\S must be connected

Proof Sketch of Theorem

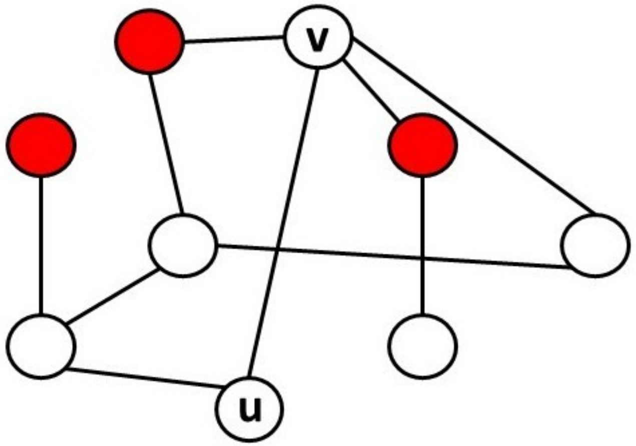
To prove: ~~When G\S is connected, H\S must be connected~~
To prove: If some edge (u,v) exists in G\S, then there is a path between u and v in H\S.

Proof Sketch of Theorem



To prove: If edge (u,v) exists in G\S, u and v are connected in H\S.

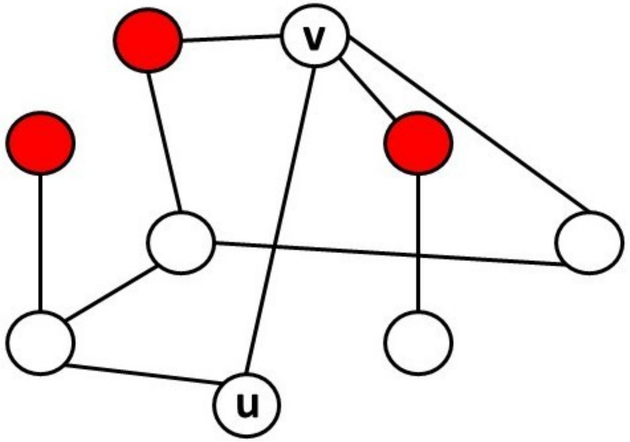
Proof Sketch of Theorem



To prove: If edge (u,v) exists in G\S, u and v are connected in H\S.

If there's some T_i that contains both u and v...

Proof Sketch of Theorem



G

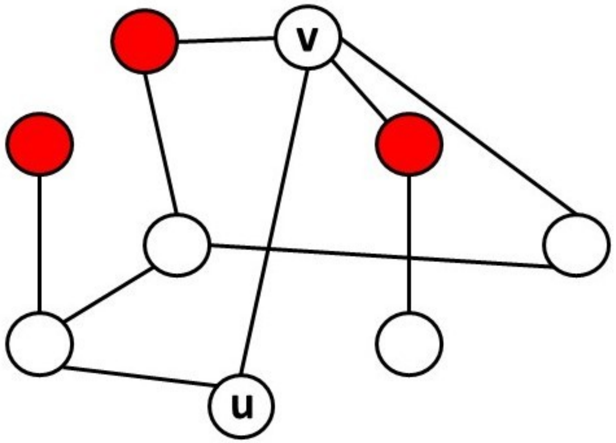
$\mathbf{T_i}$

To prove: If edge (u,v) exists in G\S, u and v are connected in H\S.

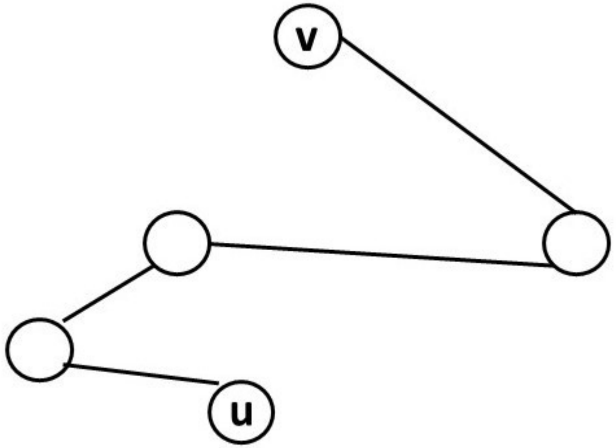
If there's some T_i that contains both u and v…

Either edge (u,v) is in T_i or

Proof Sketch of Theorem



G



T_i

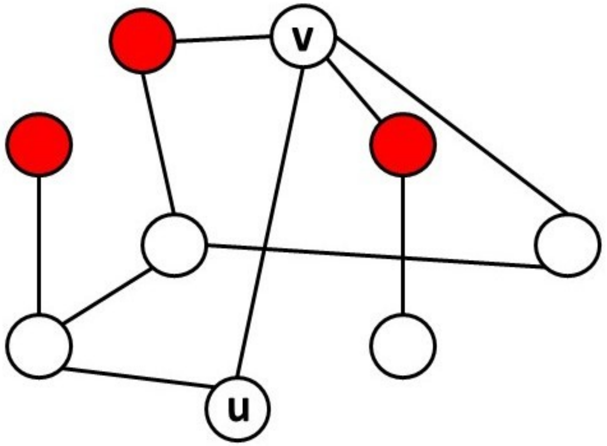
To prove: If edge (u,v) exists in G\S, u and v are connected in H\S.

If there's some T_i that contains both u and v…

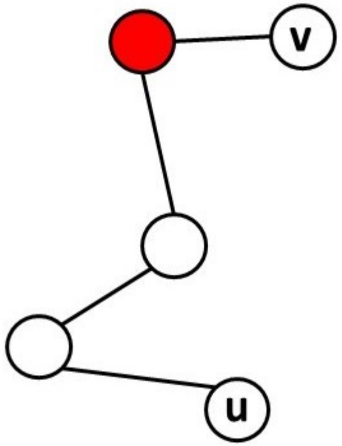
Either edge (u,v) is in T_i or

Some other path between u and v is in T_i

Proof Sketch of Theorem



G



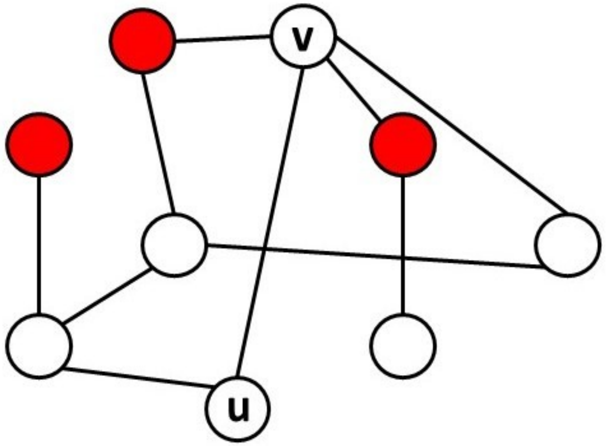
T_i

To prove: If edge (u,v) exists in G\S, u and v are connected in H\S.

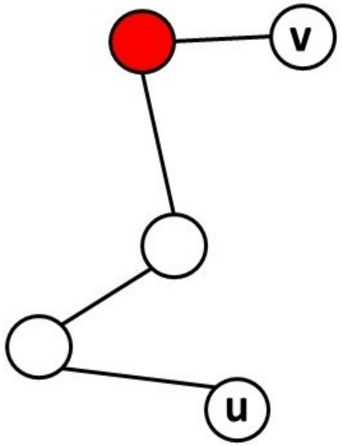
If there’s some T_i that contains both u and v…

The only potential problem is that this alternate path may contain a node that’s in S (red node).

Proof Sketch of Theorem



G



T_i

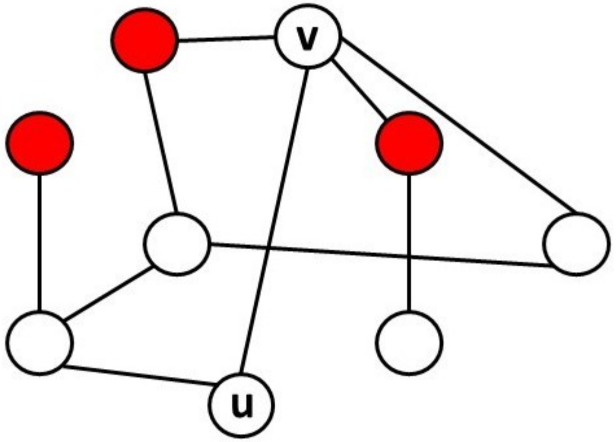
To prove: If edge (u,v) exists in G\S, u and v are connected in H\S.

If there's some T_i that contains both u and v…

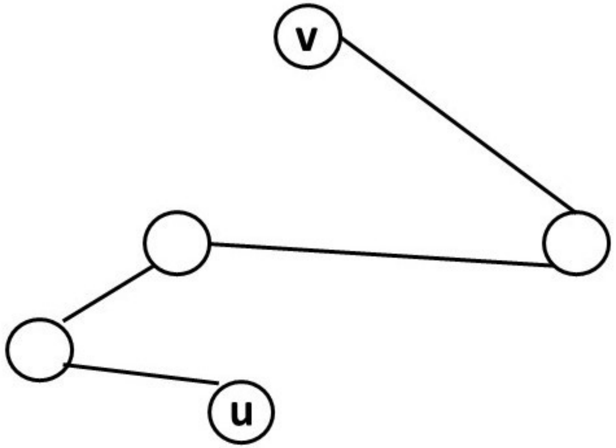
The only potential problem is that this alternate path may contain a node that's in S (red node).

So we need T_i to not contain any nodes in S.

Proof Sketch of Theorem



G



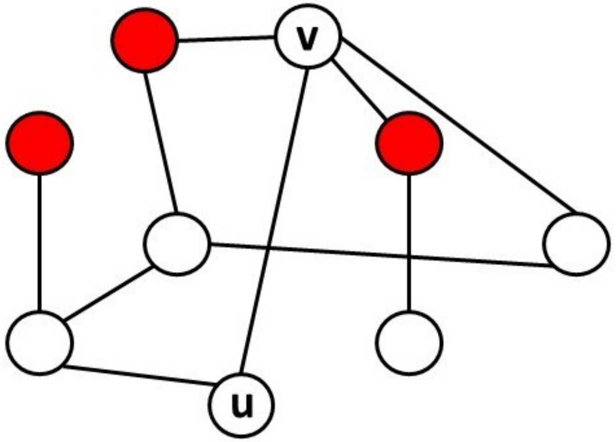
T_i

To prove: If edge (u,v) exists in G\S, u and v are connected in H\S.

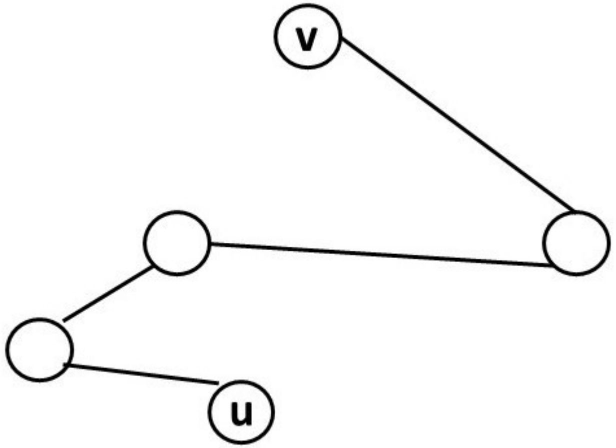
We need T_i to contain u and v, and no red S nodes.

$$P(u \text{ and } v \text{ connected in } T_i \backslash S) = \frac{1}{k^2}\left(1 - \frac{1}{k}\right)^k$$

Proof Sketch of Theorem



G



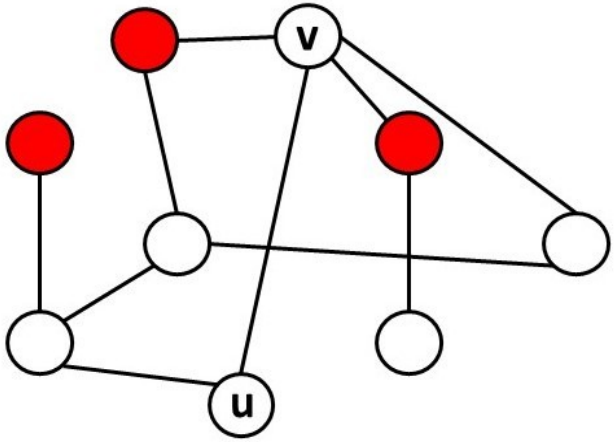
$\mathbf{T_i}$

To prove: If edge (u,v) exists in G\S, u and v are connected in H\S.

We need T_i to contain u and v, and no red S nodes.

$$P(u \text{ and } v \text{ disconnected in } T_i \backslash S) = 1 - \frac{1}{k^2}\left(1 - \frac{1}{k}\right)^k$$

Proof Sketch of Theorem

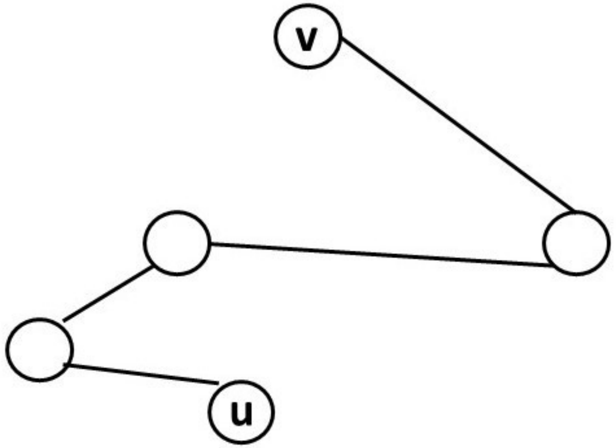


G

To prove: If edge (u,v) exists in G\S, u and v are connected in H\S.

We need T_i to contain u and v, and no red S nodes.

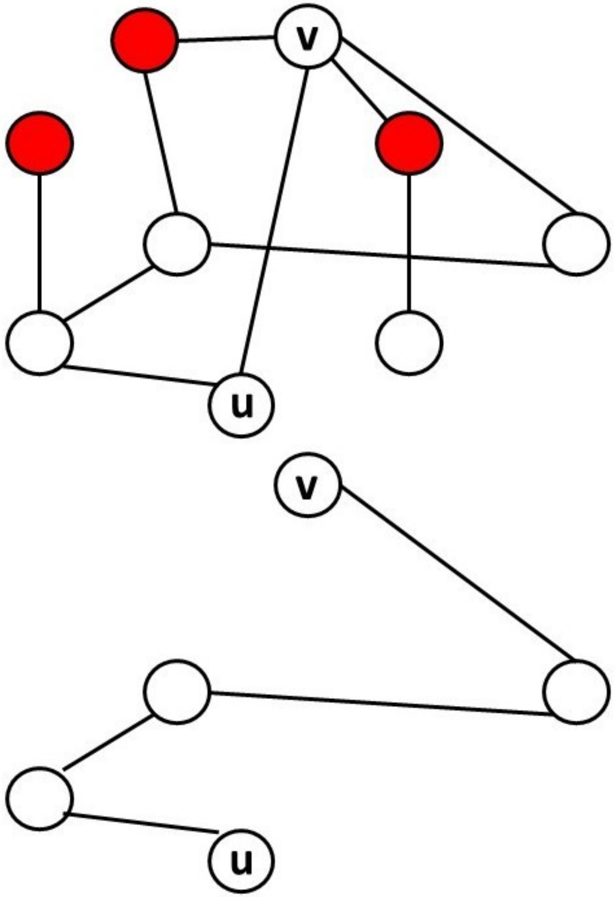
$$P(u \text{ and } v \text{ disconnected in } T_i \backslash S) = 1 - \frac{1}{k^2}\left(1 - \frac{1}{k}\right)^k$$



T_i

$$P(u \text{ and } v \text{ disconnected in } H \backslash S) = \left(1 - \frac{1}{k^2}\left(1 - \frac{1}{k}\right)^k\right)^{O(k^2 log(n))} \leq \frac{1}{n^4}$$

Proof Sketch of Theorem



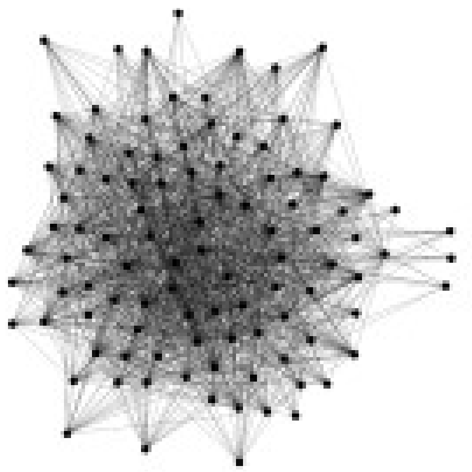
To prove: If edge (u,v) exists in G\S, u and v are connected in H\S.

We need T_i to contain u and v, and no red S nodes.

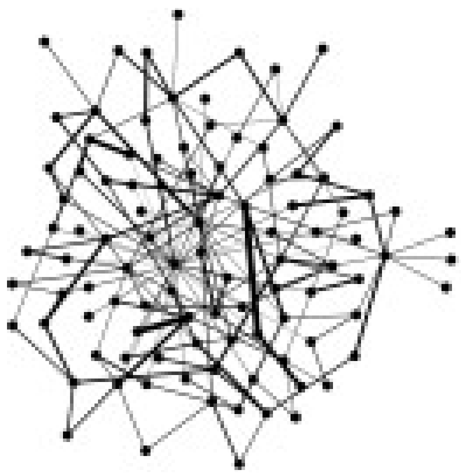
The probability that no T_i meets this requirement is at most $1/n^4$.

So u and v are connected in H\S with high probability. //

We did it!



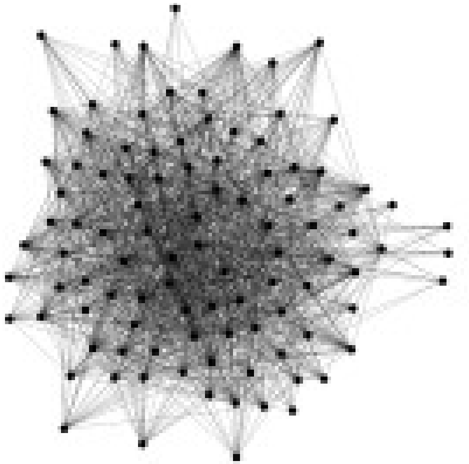
G



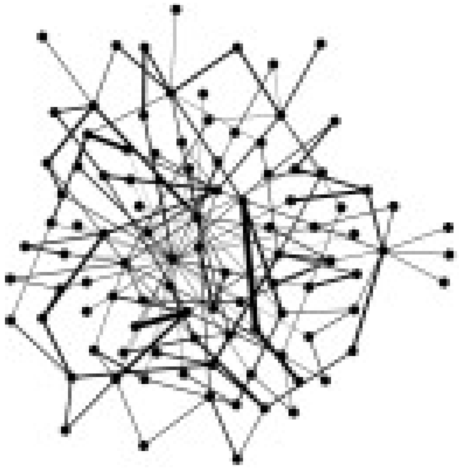
H

We showed how to create a *certificate* graph H that matches G’s vertex connectivity up to constant k, but has only roughly kn edges.

We did it!



G



H

We showed how to create a *certificate* graph H that matches G’s vertex connectivity up to constant k, but has only roughly kn edges.

You can process a massive stream of the edges in G to create H, which is much smaller. Then you can run a traditional vertex connectivity algorithm on H to get your answer.

Query-Based Algorithms

WHEN DISCOVERING GRAPH EDGES IS COSTLY

When Graph Edges Are Costly

Can check existence of any edge at any time, but pay a significant cost

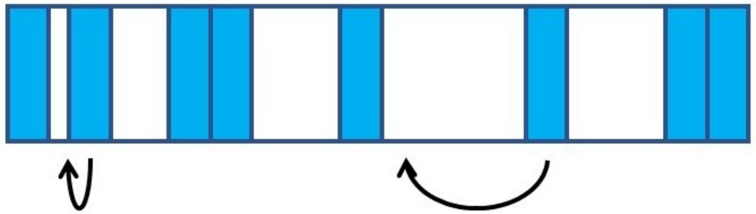
Want to minimize # of queries

When Graph Edges Are Costly

Can check existence of any edge at any time, but pay a significant cost

Want to minimize # of queries

Mesh memory manager (PLDI 2019)

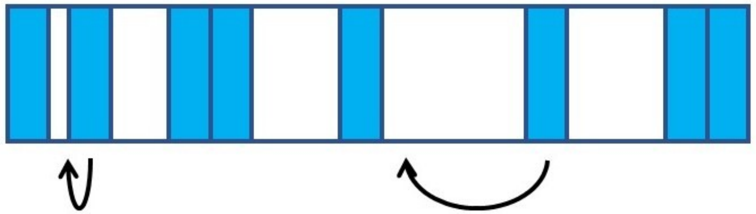


When Graph Edges Are Costly

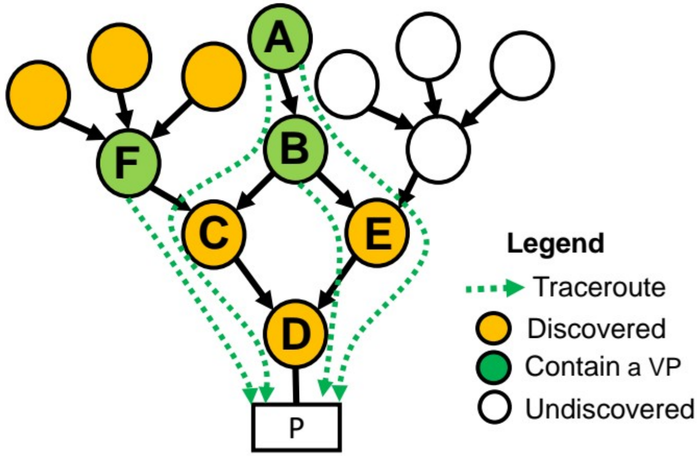
Can check existence of any edge at any time, but pay a significant cost

Want to minimize # of queries

Mesh memory manager (PLDI 2019)



PathCache network path predictor
(to be submitted SIGCOMM 2020)

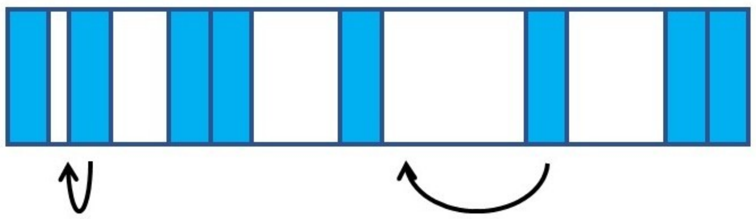


When Graph Edges Are Costly

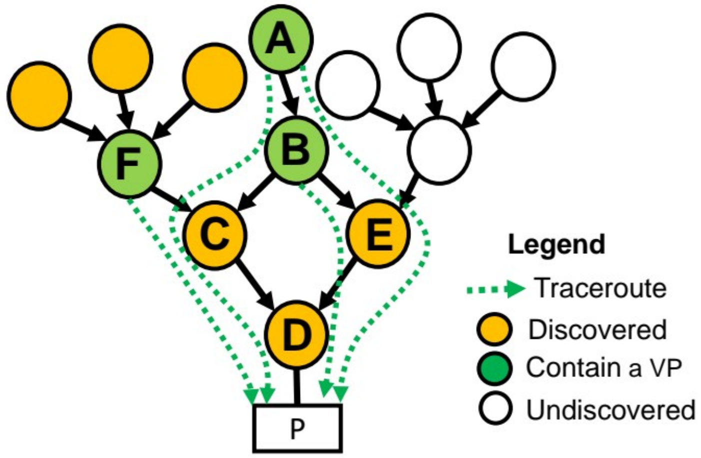
Can check existence of any edge at any time, but pay a significant cost

Want to minimize # of queries

Mesh memory manager (PLDI 2019)



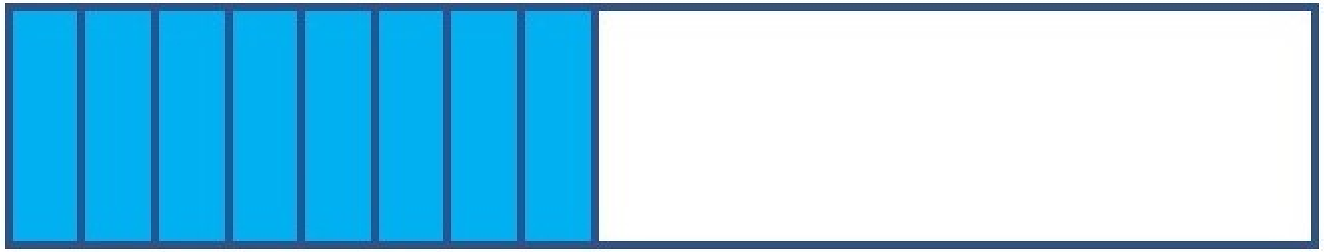
PathCache network path predictor (to be submitted SIGCOMM 2020)



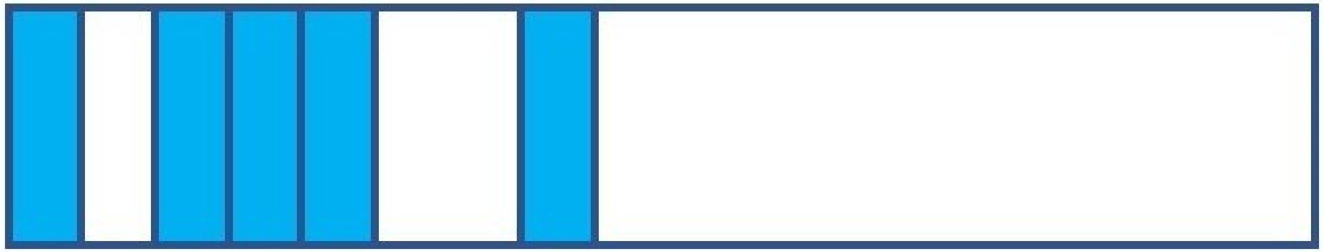
Memory Fragmentation

Memory Fragmentation

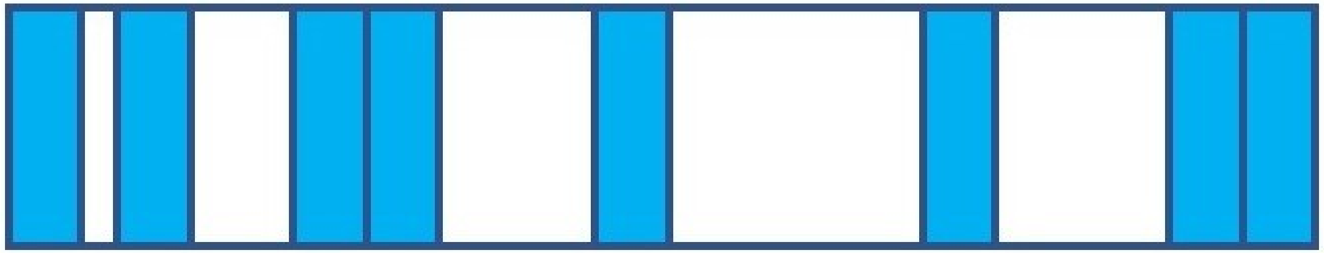
Memory Fragmentation



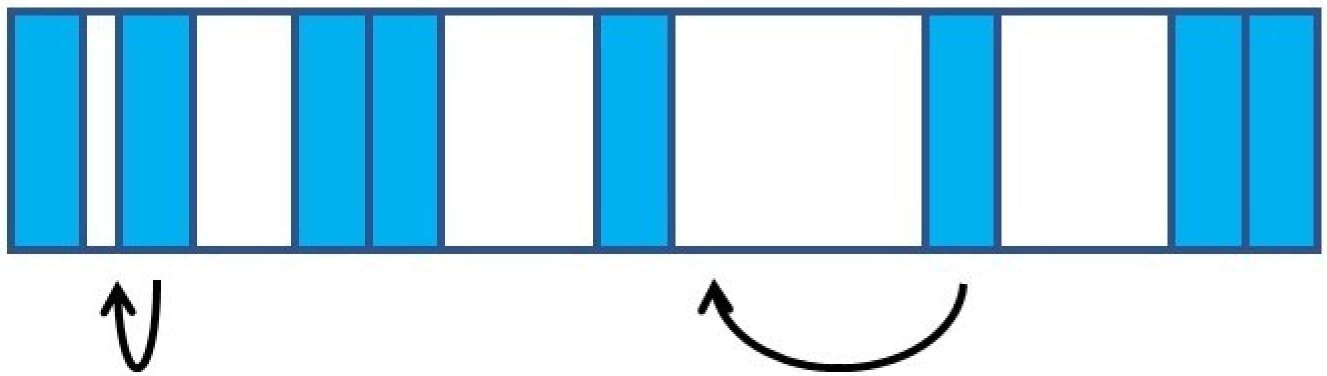
Memory Fragmentation



Memory Fragmentation

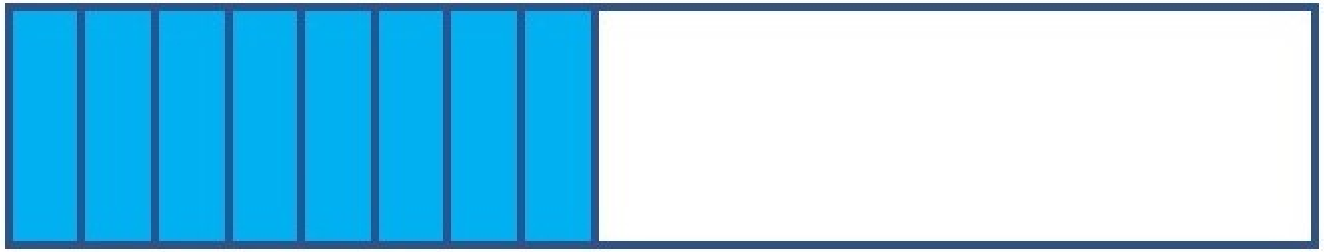


Memory Fragmentation



We want to reorganize or “compact” the allocated regions of memory to be contiguous.

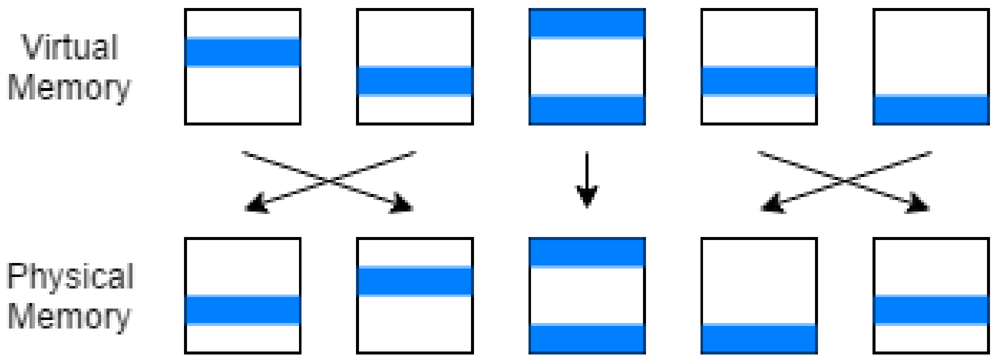
Memory Fragmentation



We want to reorganize or “compact” the allocated regions of memory to be contiguous.

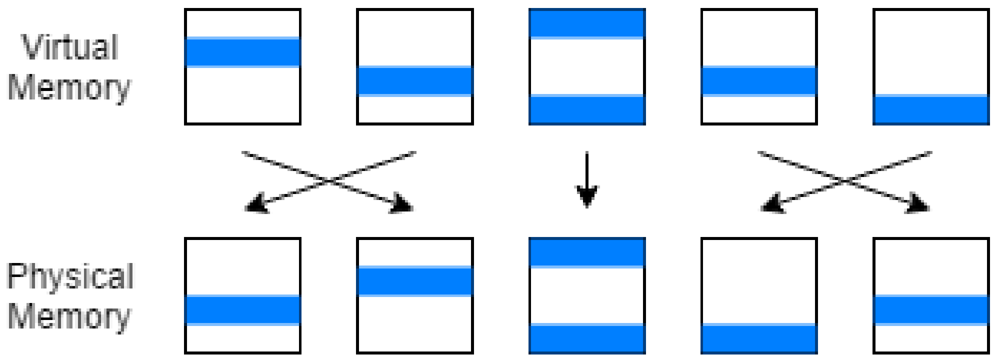
Virtual Memory – A Quick Primer

Modern operating systems maintain a mapping between virtual and physical memory.



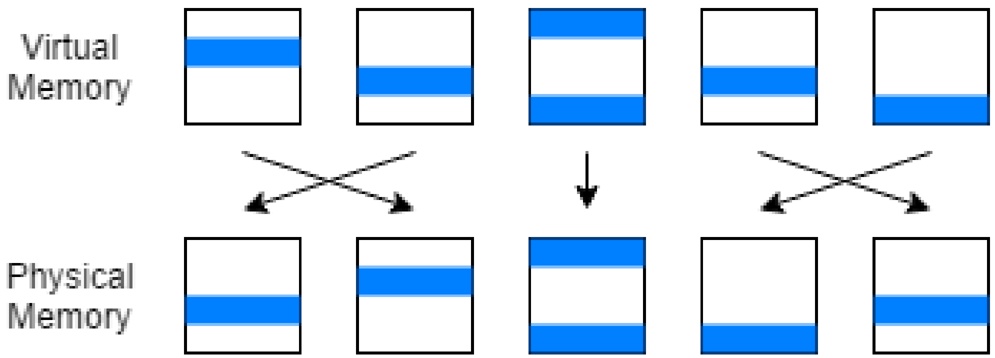
Virtual Memory – A Quick Primer

If we relocate objects in physical memory, we have to update their virtual addresses as well.



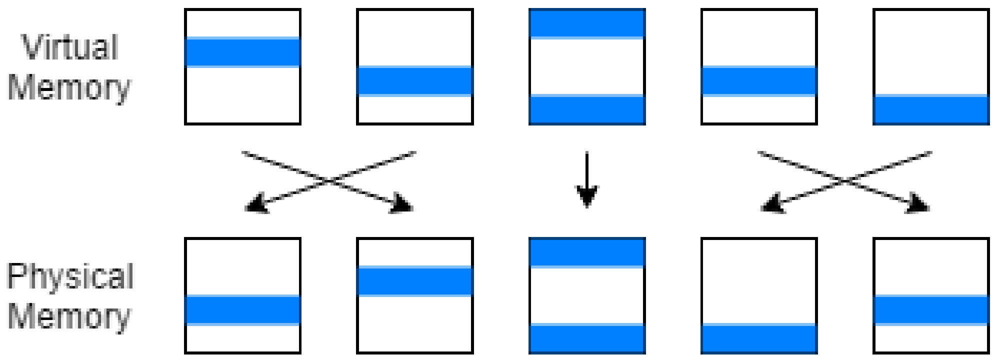
Virtual Memory – A Quick Primer

But in C and C++, we can’t alter virtual addresses safely.



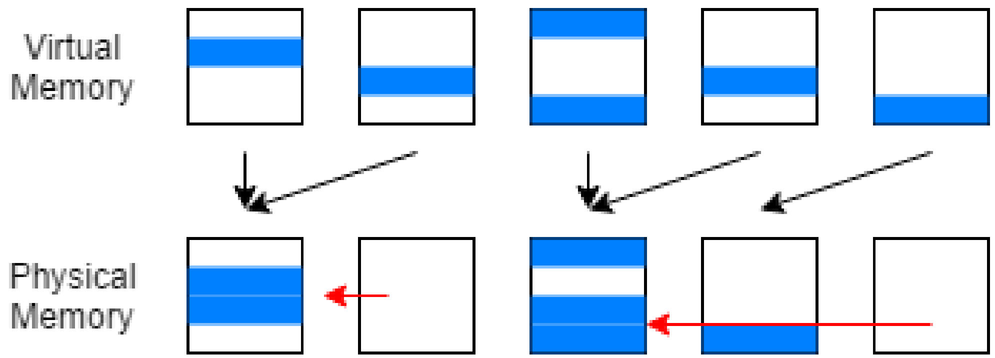
Virtual Memory – A Quick Primer

How can we relocate objects without changing their virtual addresses?



Virtual Memory – A Quick Primer

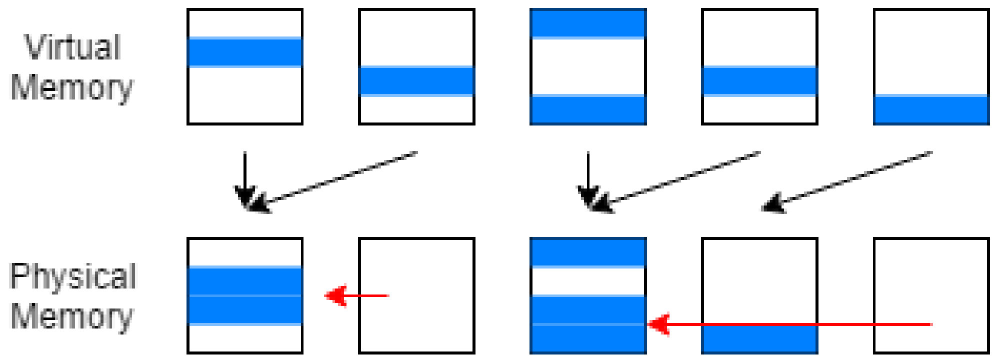
We can remap two virtual pages onto the same physical page in memory*, and discard one of the physical pages.

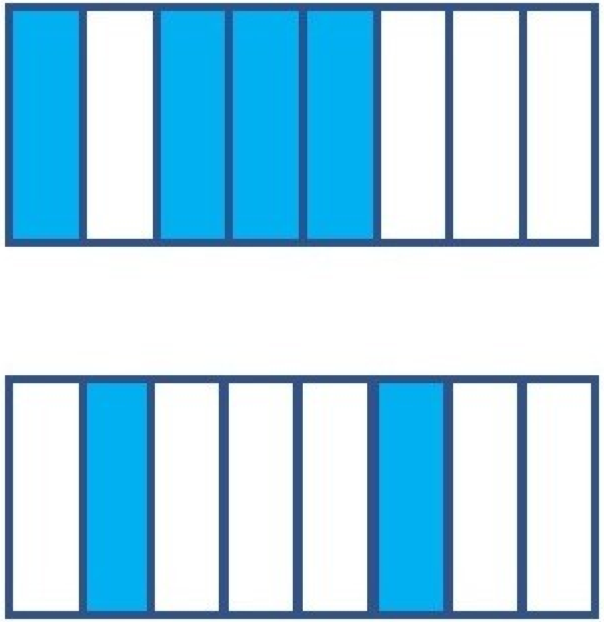


Virtual Memory – A Quick Primer

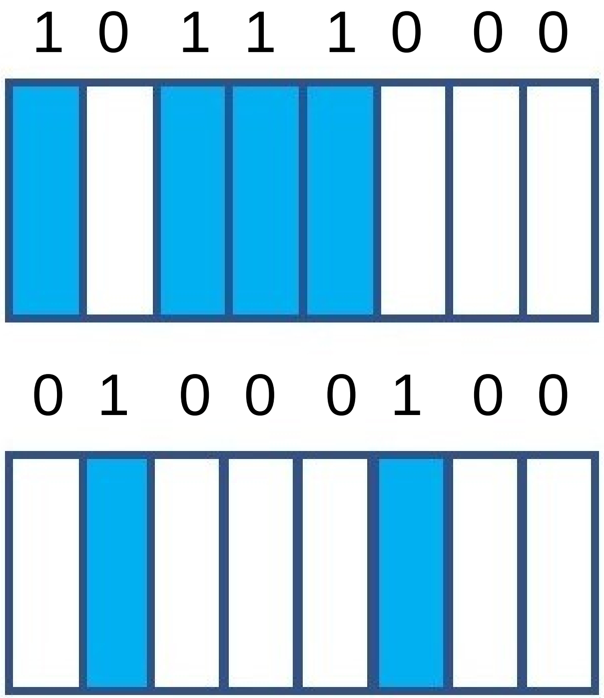
We can remap two virtual pages onto the same physical page in memory*, and discard one of the physical pages.

* provided there are no collisions between objects on the two pages.



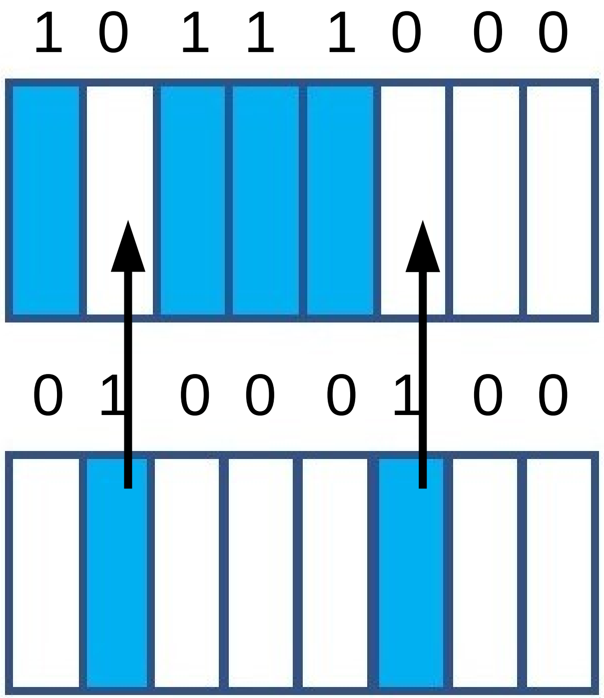


- Memory organized into pages
- Each page holds same # of objects (8)
- Objects are placed on page uniformly at random



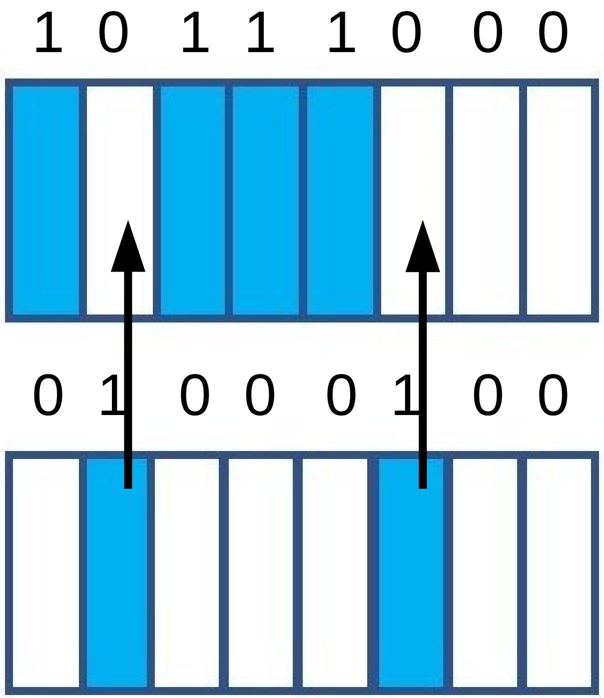
We can represent each page as a bitstring, where 0 indicates a free slot and 1 indicates an occupied slot.

We can *mesh* two pages together if they don't have 1s in the same position.

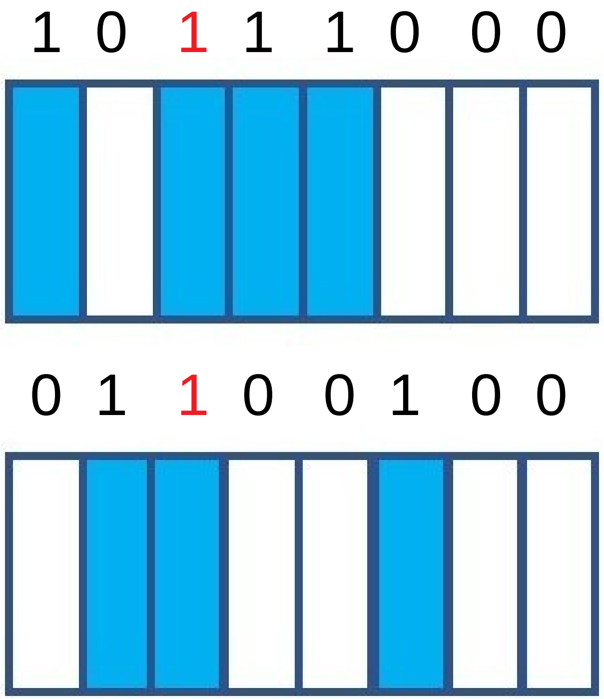


We can represent each page as a bitstring, where 0 indicates a free slot and 1 indicates an occupied slot.

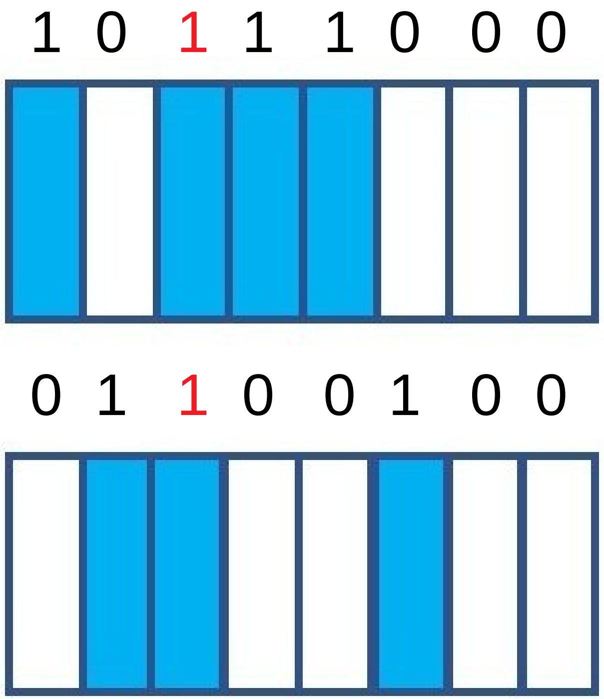
We can *mesh* two pages together if they don’t have 1s in the same position.



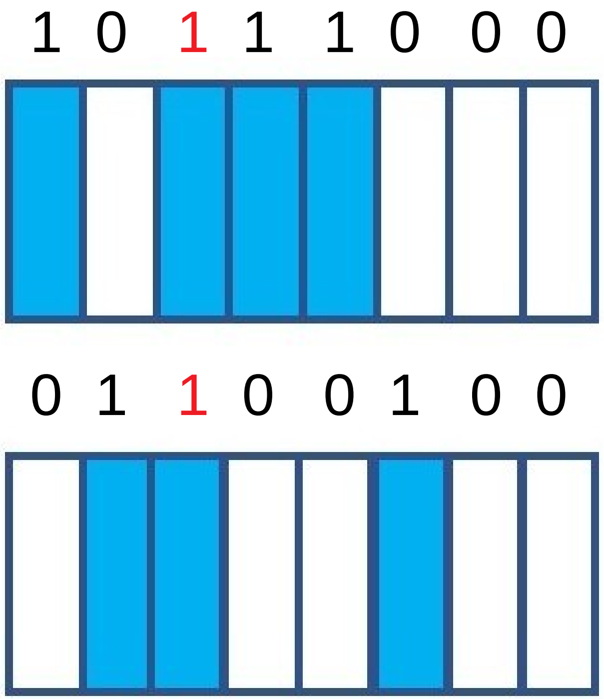
THEY MESH!



If both bitstrings have a 1 in some position, we can’t mesh the strings together.



NO MESH



Now we can forget about the details of memory, and think about our problem in terms of finding meshable pairs of bitstrings.

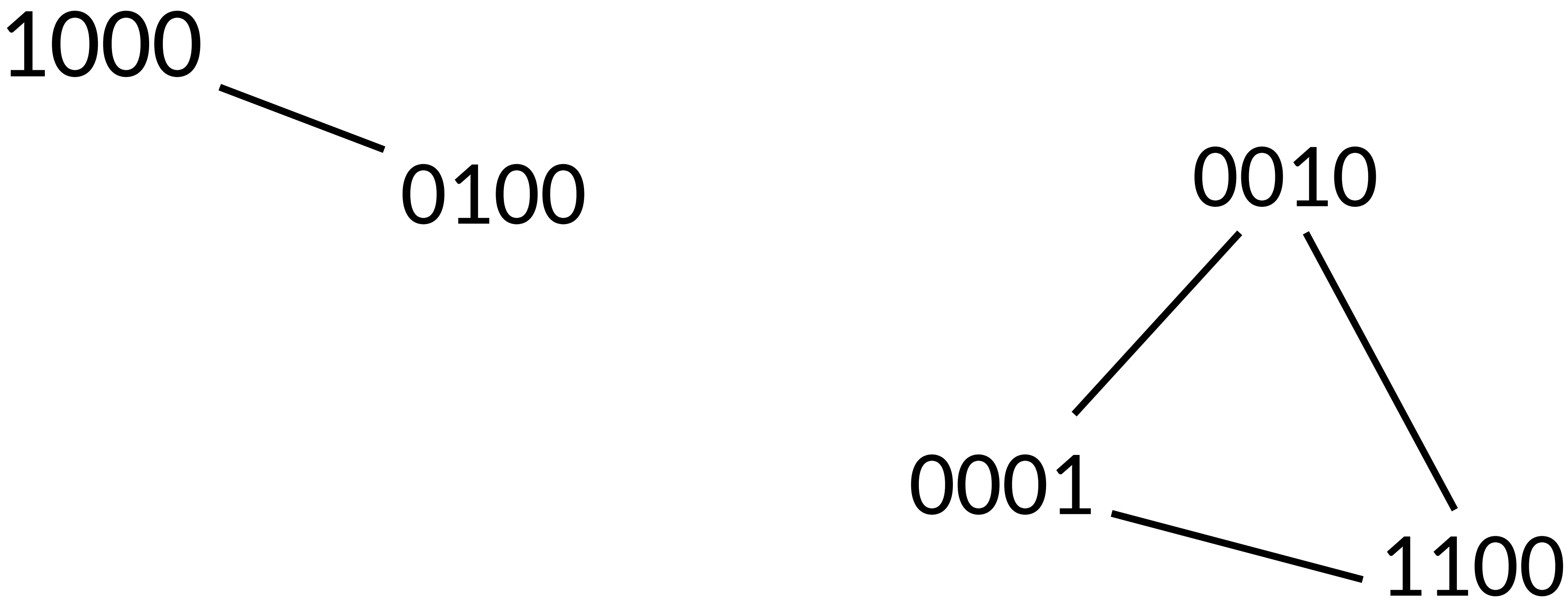
We want to mesh as many pairs of strings as possible.

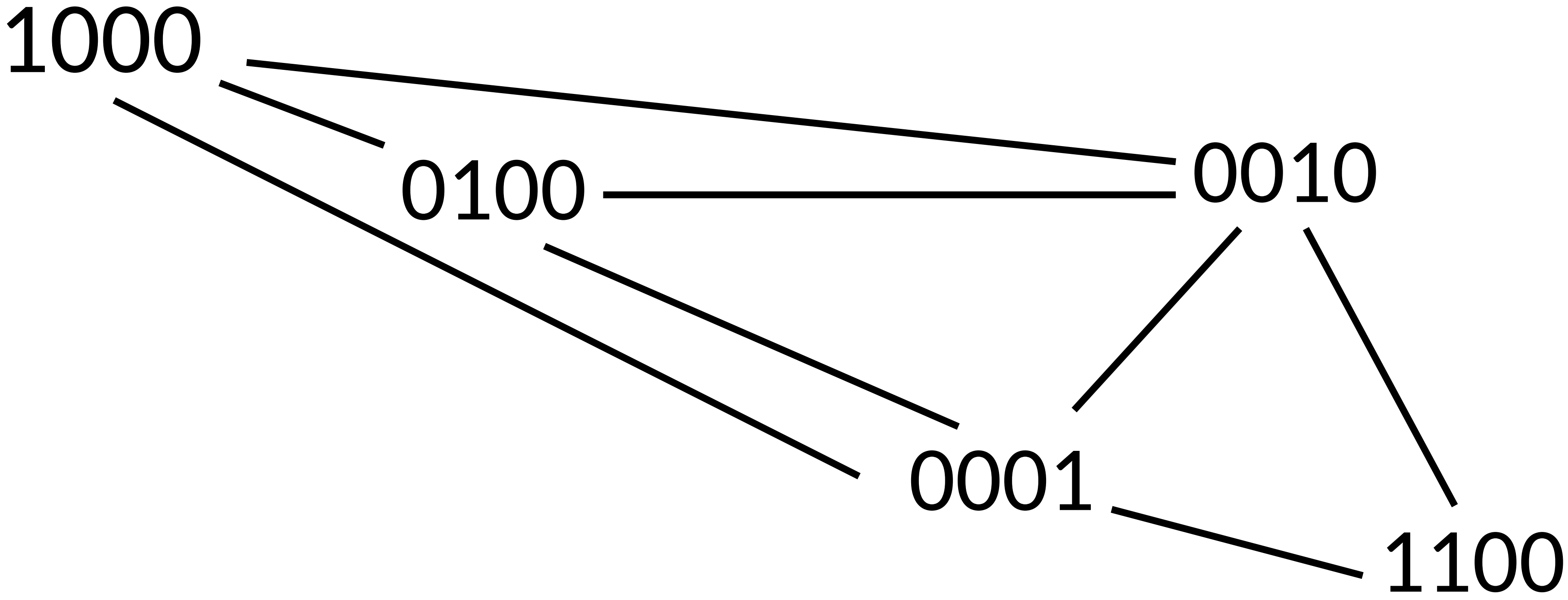
1000

0100

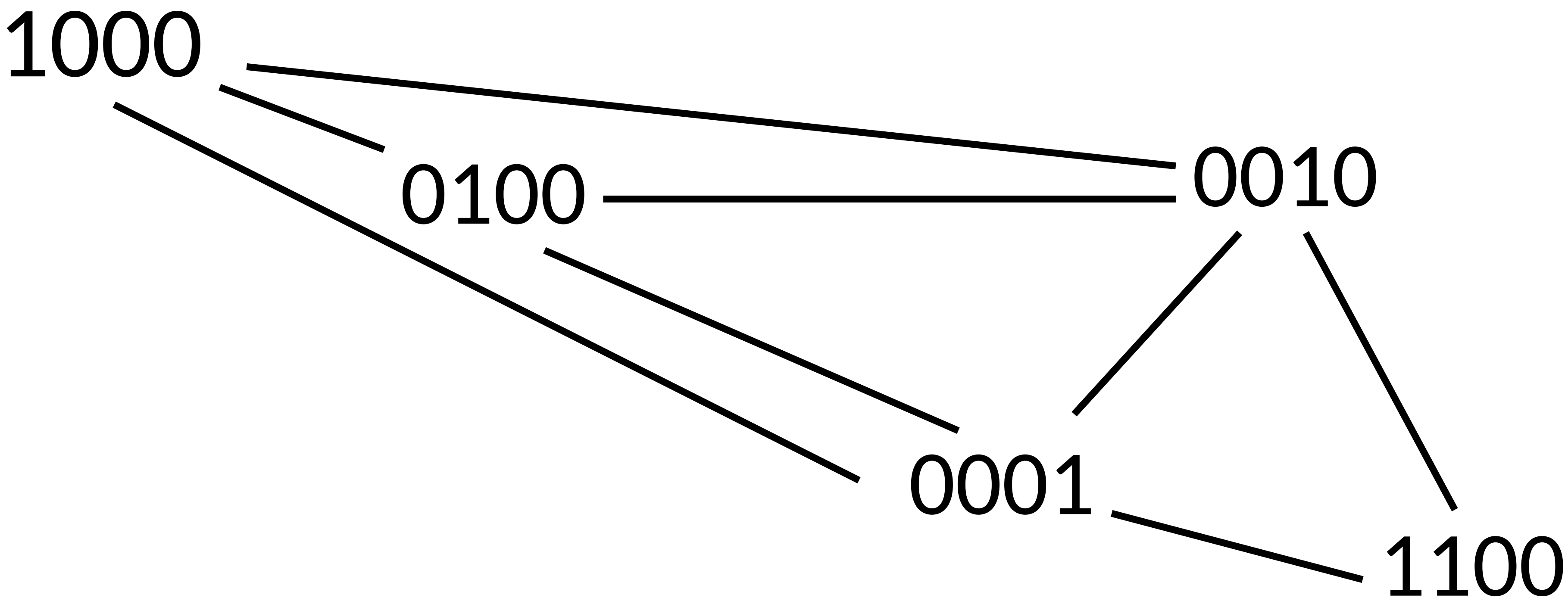
1000

0100

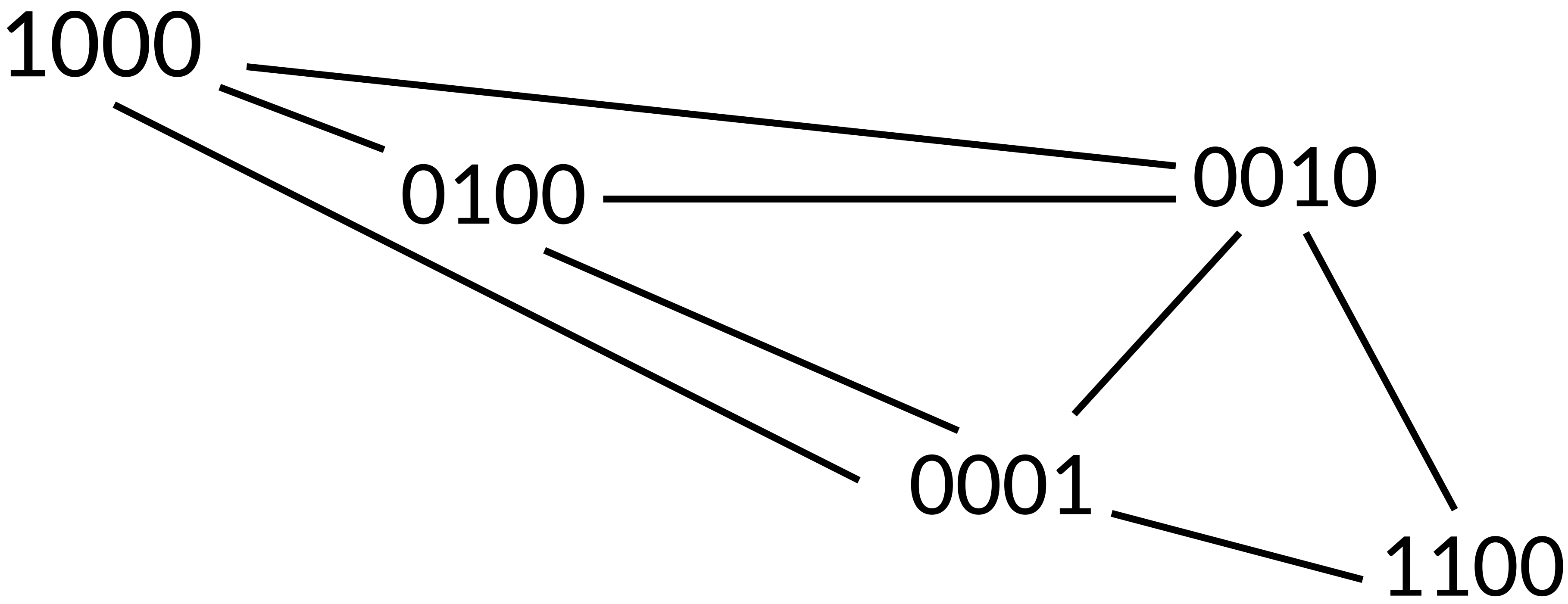

1000
0100
0010
0001
1100


1000
0100
0010
0001
1100

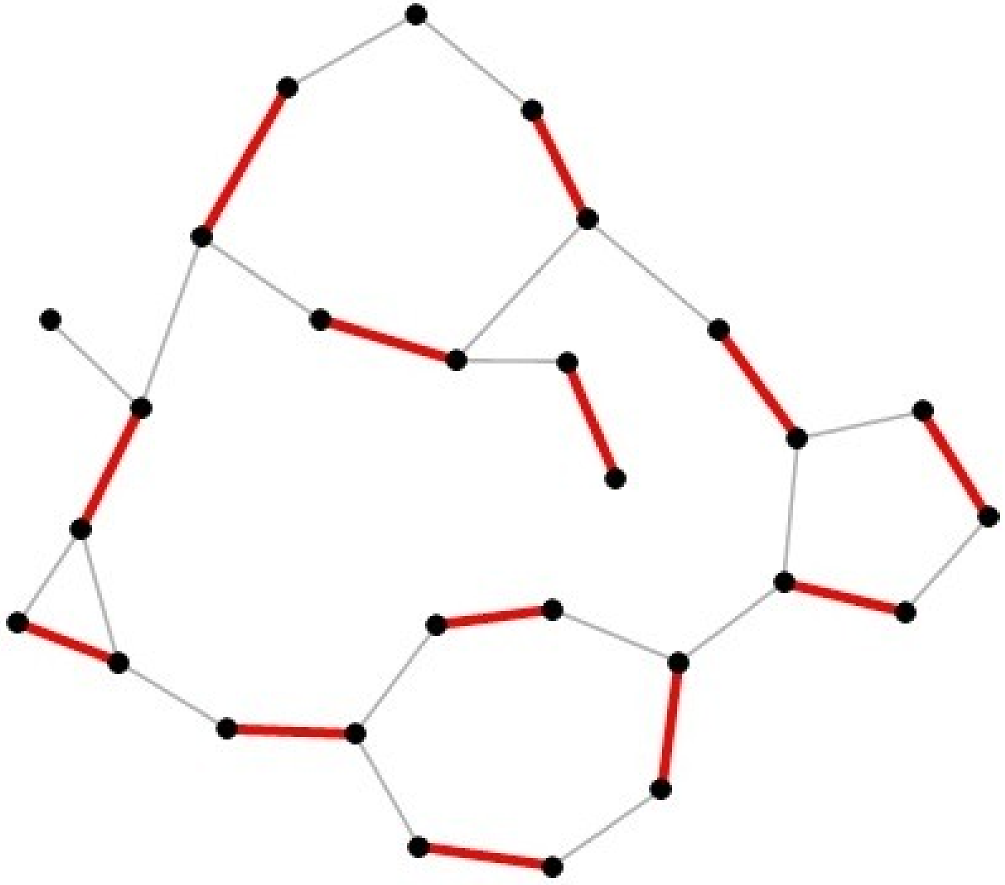
We can think of this as a graph problem!



We can think of this as a graph problem!
Mesh as many pairs as possible.

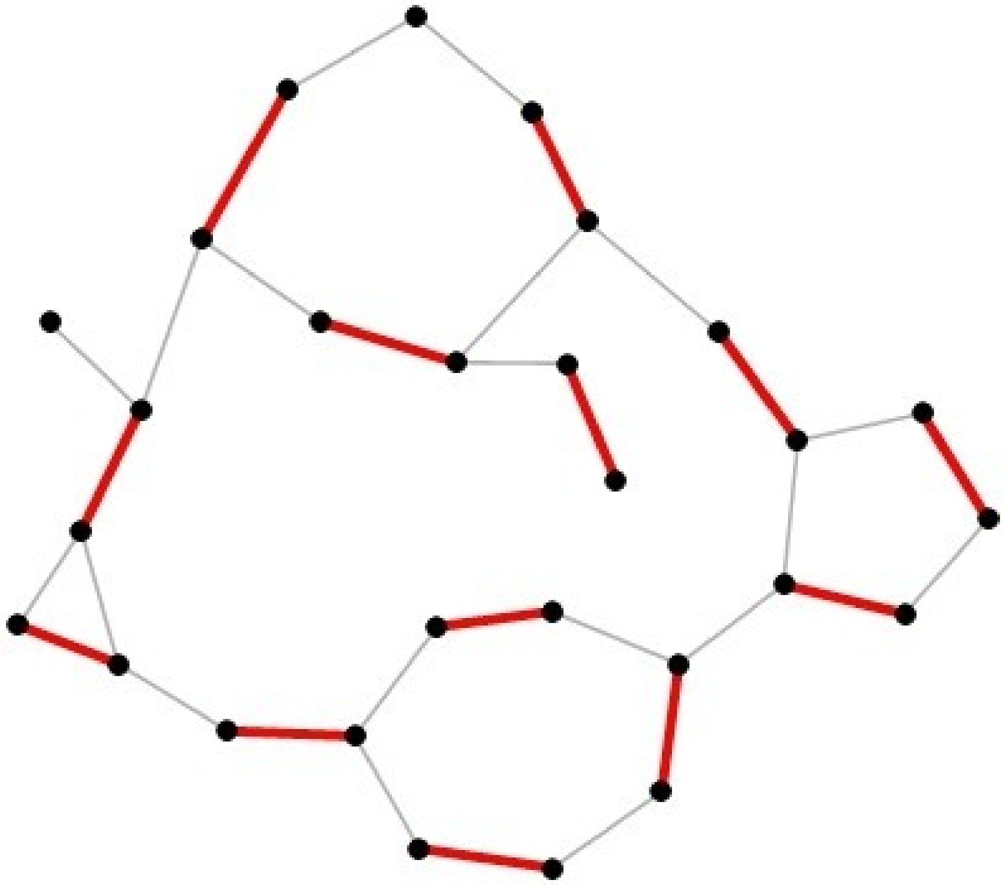


Maximum Matching



Maximum Matching

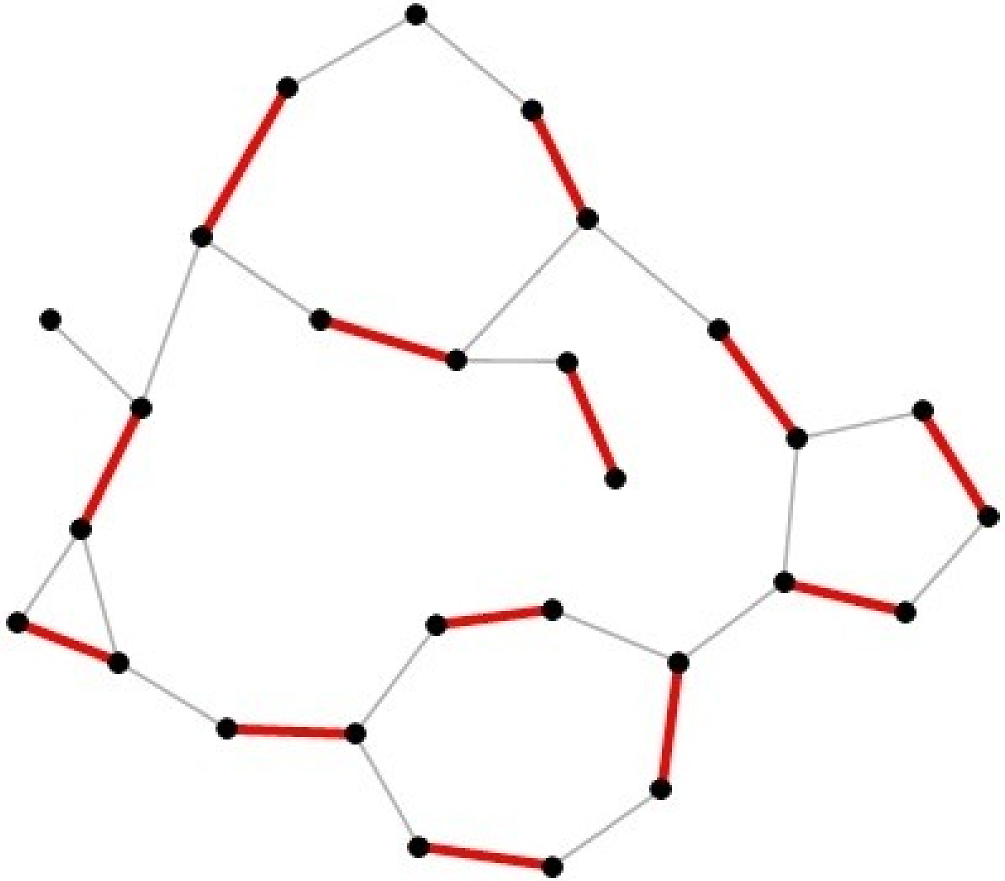
Well-known polynomial time algorithm



Maximum Matching

Well-known polynomial time algorithm

Requires random access to graph

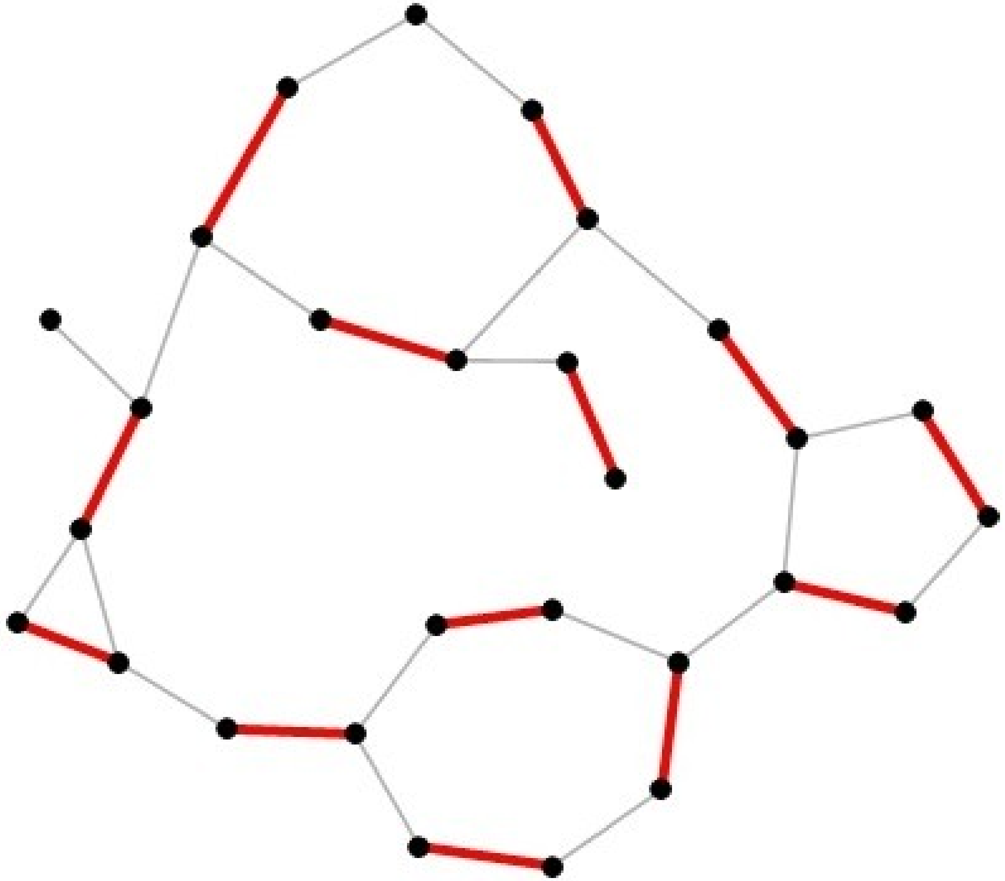


Maximum Matching

Well-known polynomial time algorithm

Requires random access to graph

Do we have random access?
<u>No</u>.



Not Enough Time!

We mesh **during program execution.**

Not Enough Time!

We mesh **during program execution.**

We must “pause” the program to mesh.

Not Enough Time!

We mesh **during program execution.**

We must “pause” the program to mesh.

Only safe to “pause” for a very short time.

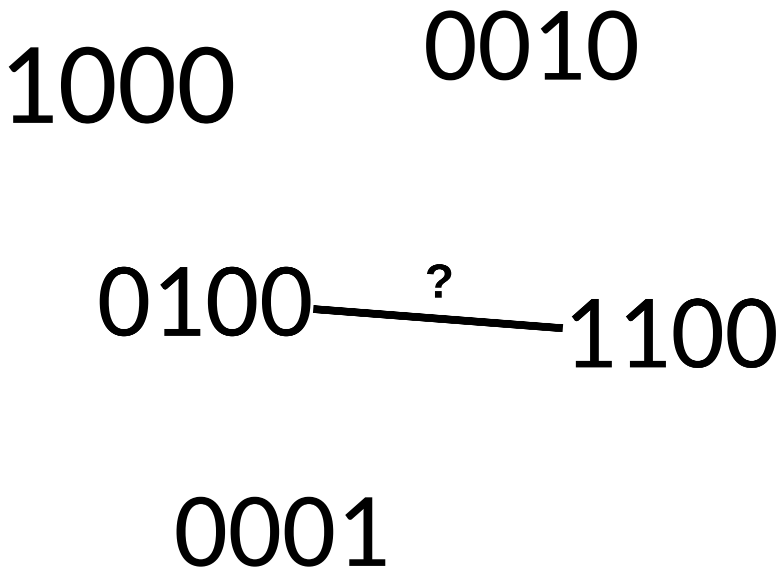
Not Enough Time!

We mesh **during program execution.**

We must “pause” the program to mesh.

Only safe to “pause” for a very short time.

Graph doesn’t exist yet! We have to do computation to test whether any edge exists, costing valuable time.



Not Enough Time!

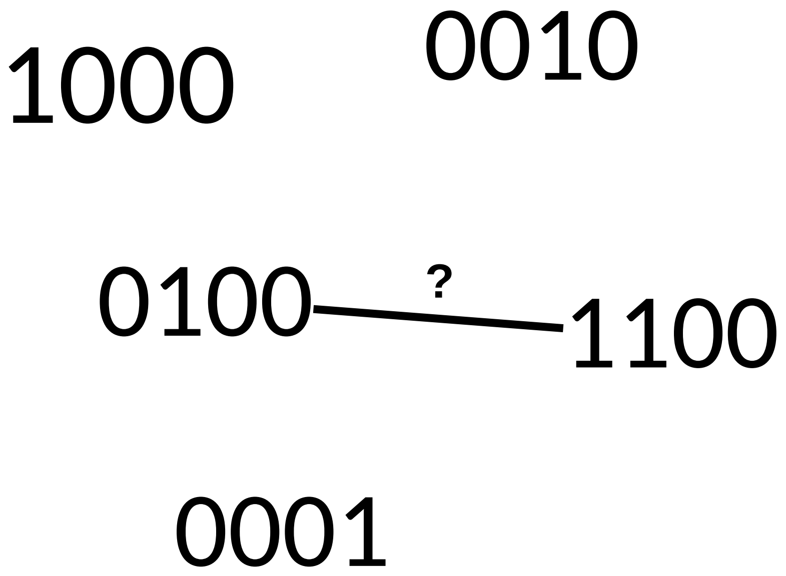
We mesh **during program execution.**

We must “pause” the program to mesh.

Only safe to “pause” for a very short time.

Graph doesn’t exist yet! We have to do computation to test whether any edge exists, costing valuable time.

Checking an edge is a costly query.



Random Graphs

Recall that the 1s in the bitstrings are distributed randomly.

0000100000011000

Random Graphs

Recall that the 1s in the bitstrings are distributed randomly.

0000100000011000

Say we have a length b bitstring that's 10% full.

Random Graphs

Recall that the 1s in the bitstrings are distributed randomly.

We can write the probability that it meshes with another 10% full string as

00001000000011000

Say we have a length b bitstring that's 10% full.

Random Graphs

Recall that the 1s in the bitstrings are distributed randomly.

We can write the probability that it meshes with another 10% full string as

$$q = \binom{b - b/10}{b/10} \Big/ \binom{b}{b/10}$$

0000100000011000

Say we have a length b bitstring that's 10% full.

Random Graphs

Recall that the 1s in the bitstrings are distributed randomly.

We can write the probability that it meshes with another 10% full string as

$$q = \binom{b - b/10}{b/10} \bigg/ \binom{b}{b/10}$$

In general, if we know how many 1s are in each string, we can get the probability of any two strings meshing.

0000100000011000

Say we have a length b bitstring that's 10% full.

Random Graphs

Recall that the 1s in the bitstrings are distributed randomly.

We can write the probability that it meshes with another 10% full string as

$$q = \binom{b - b/10}{b/10} \Big/ \binom{b}{b/10}$$

In general, if we know how many 1s are in each string, we can get the probability of any two strings meshing.

0000100000011000

We can expect to find a mesh for a bitstring using 1/q queries.

Random Graphs

Recall that the 1s in the bitstrings are distributed randomly.

We can write the probability that it meshes with another 10% full string as

$$q = \binom{b - b/10}{b/10} \Big/ \binom{b}{b/10}$$

In general, if we know how many 1s are in each string, we can get the probability of any two strings meshing.

00001000000011000

We can expect to find a mesh for a bitstring using 1/q queries.

This forms the basis for a greedy query algorithm.

Random Graphs

Recall that the 1s in the bitstrings are distributed randomly.

We can write the probability that it meshes with another 10% full string as

$$q = \binom{b - b/10}{b/10} \Big/ \binom{b}{b/10}$$

In general, if we know how many 1s are in each string, we can get the probability of any two strings meshing.

00001000000011000

We can expect to find a mesh for a bitstring using 1/q queries.

This forms the basis for a greedy query algorithm.

But the situation isn't so simple.

Random Graphs

There is a rich body of work on algorithms for random graphs (including matching).

0000100000011000

We can expect to find a mesh for a bitstring using 1/q queries.

This forms the basis for a greedy query algorithm.

But the situation isn't so simple.

Random Graphs

There is a rich body of work on algorithms for random graphs (including matching).

However, this work assumes that the graph's edges are independent.

0000100000011000

We can expect to find a mesh for a bitstring using 1/q queries.

This forms the basis for a greedy query algorithm.

But the situation isn't so simple.

Random Graphs

There is a rich body of work on algorithms for random graphs (including matching).

However, this work assumes that the graph's edges are independent.

That's not the case for meshing graphs!

00001000000011000

We can expect to find a mesh for a bitstring using 1/q queries.

This forms the basis for a greedy query algorithm.

But the situation isn't so simple.

Edge Dependence Example

Say we know all of our bitstrings are half full.

???? (50% full)

????(50% full)

???? (50% full)

Edge Dependence Example

Say we know all of our bitstrings are half full.

Then if two bitstrings mesh, they must be each other's complement.

???? ✓ ????(50% full)

(50% full)

???? (50% full)

Edge Dependence Example

Say we know all of our bitstrings are half full.

Then if two bitstrings mesh, they must be each other's complement.

0110 ✓ 1001

???? (50% full)

Edge Dependence Example

Say we know all of our bitstrings are half full.

Then if two bitstrings mesh, they must be each other's complement.

So in this case triangles are impossible. If two of the edges exist, the third must not exist.

0110 —✓— 1001

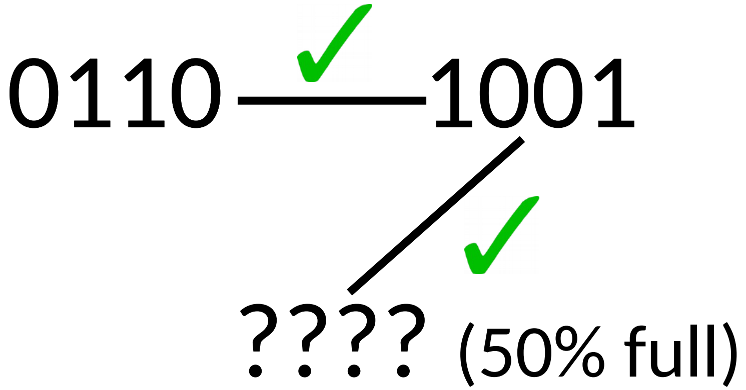
???? (50% full)

Edge Dependence Example

Say we know all of our bitstrings are half full.

Then if two bitstrings mesh, they must be each other’s complement.

So in this case triangles are impossible. If two of the edges exist, the third must not exist.

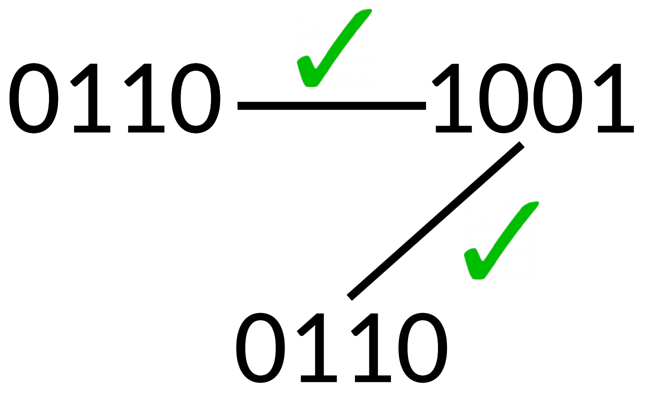


Edge Dependence Example

Say we know all of our bitstrings are half full.

Then if two bitstrings mesh, they must be each other’s complement.

So in this case triangles are impossible. If two of the edges exist, the third must not exist.

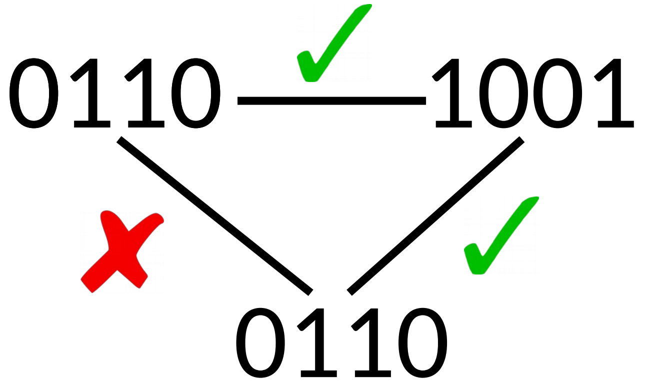


Edge Dependence Example

Say we know all of our bitstrings are half full.

Then if two bitstrings mesh, they must be each other’s complement.

So in this case triangles are impossible. If two of the edges exist, the third must not exist.

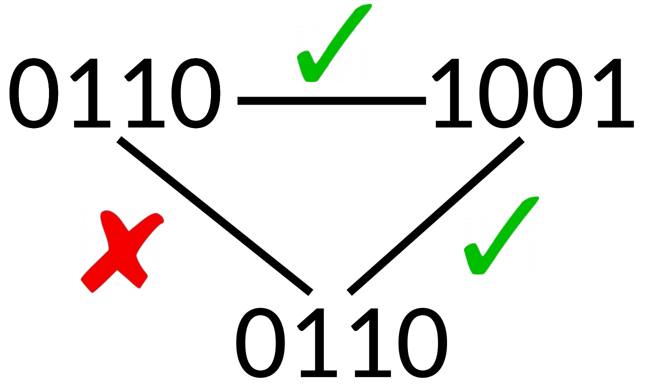


Edge Dependence Example

Say we know all of our bitstrings are half full.

Then if two bitstrings mesh, they must be each other's complement.

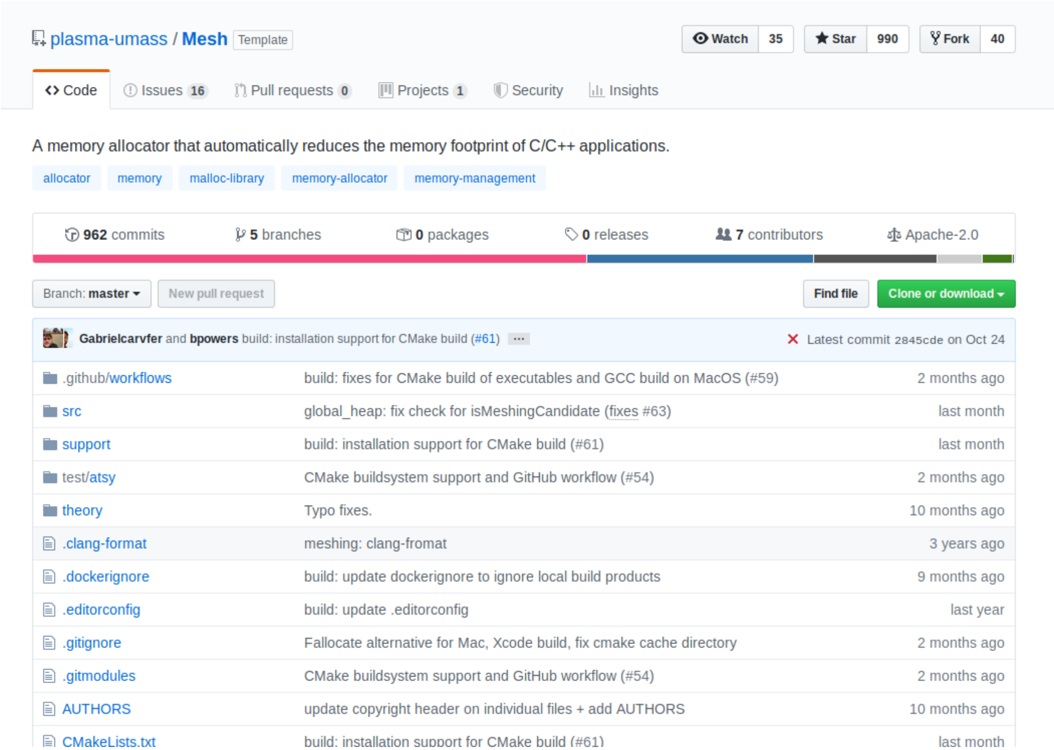
So in this case triangles are impossible. If two of the edges exist, the third must not exist.



This is a novel and interesting mathematical structure!

Mesh

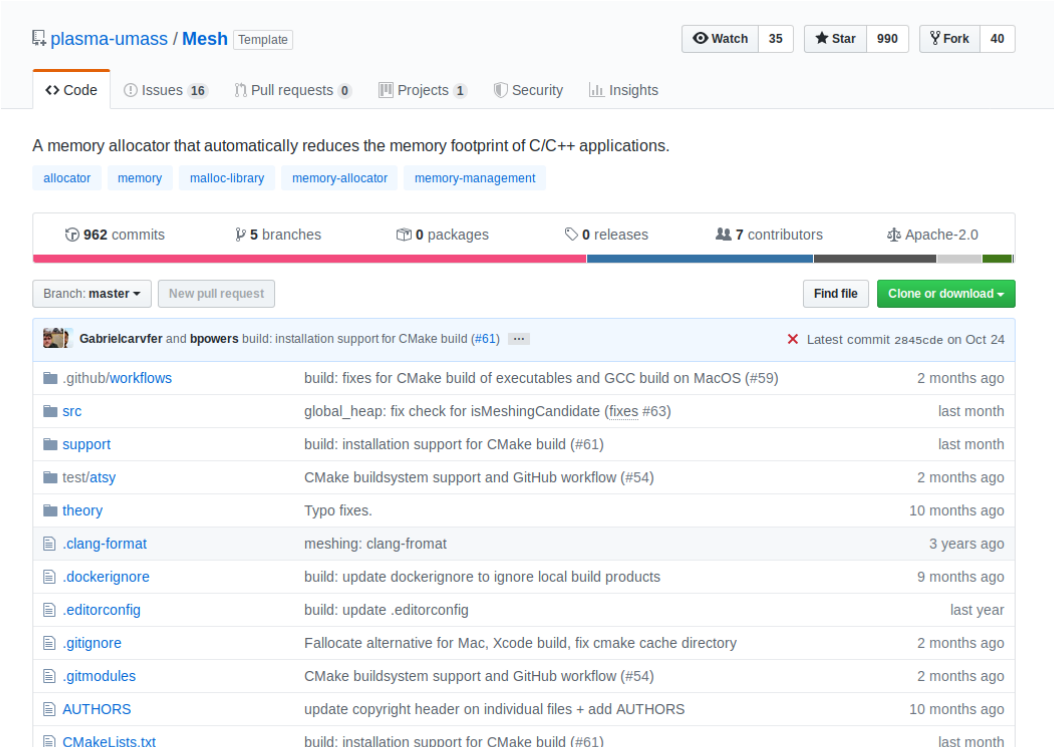
We built Mesh, a memory manager powered by a query-limited matching algorithm that can perform memory compaction in C and C++.



Mesh

We built Mesh, a memory manager powered by a query-limited matching algorithm that can perform memory compaction in C and C++.

It reduces memory consumption of Firefox by 16% and Redis by 39%.

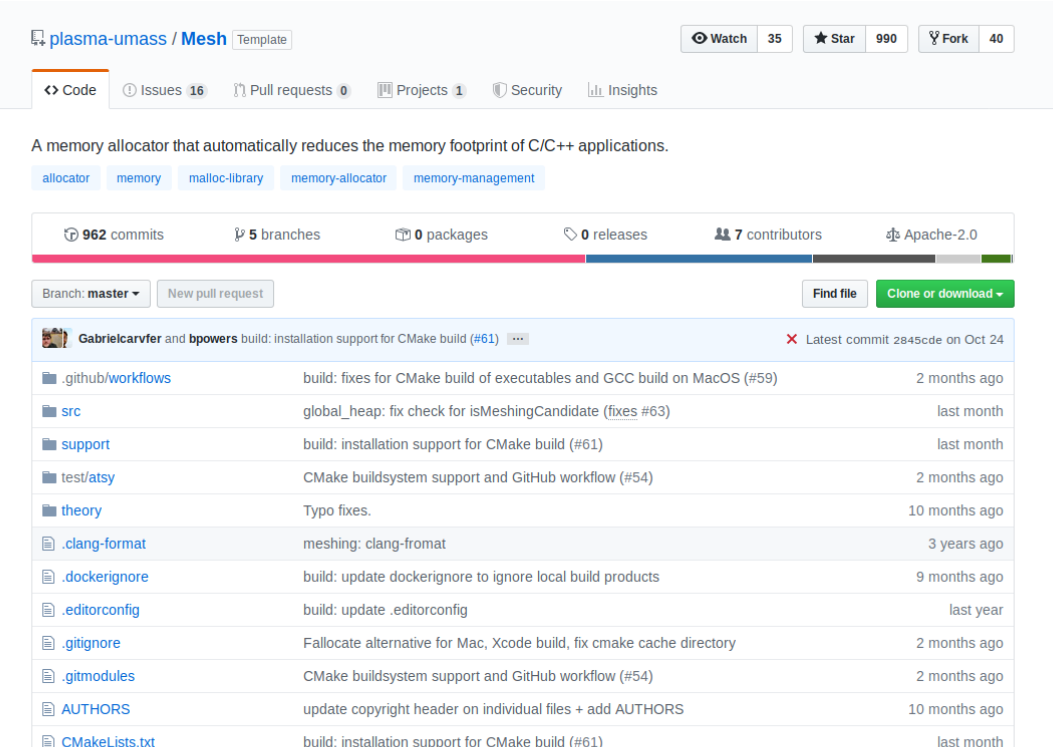


Mesh

We built Mesh, a memory manager powered by a query-limited matching algorithm that can perform memory compaction in C and C++.

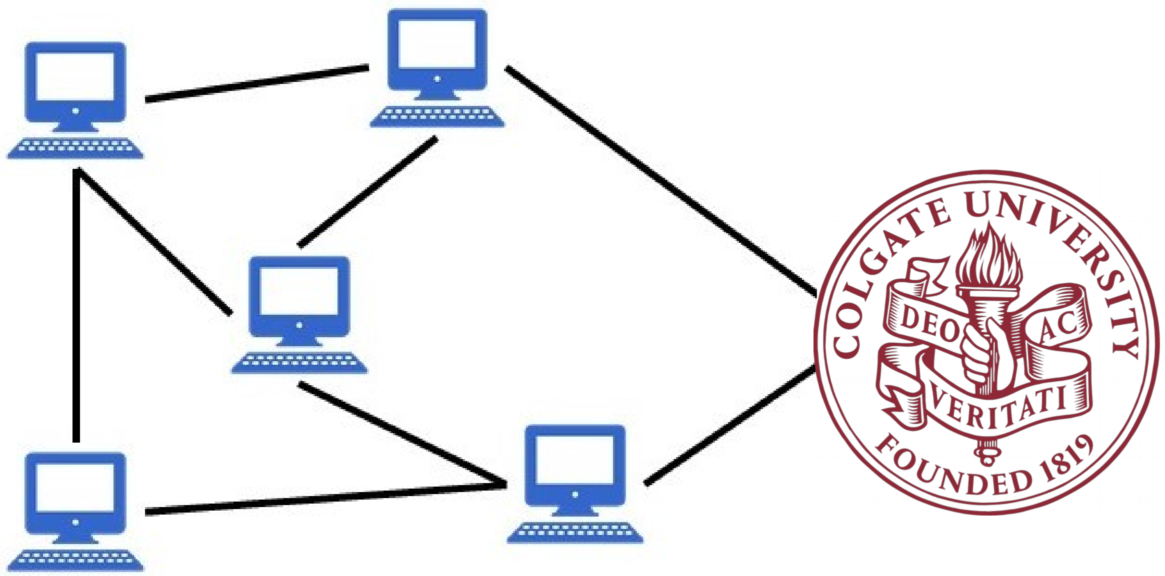
It reduces memory consumption of Firefox by 16% and Redis by 39%.

It is freely available on Github.



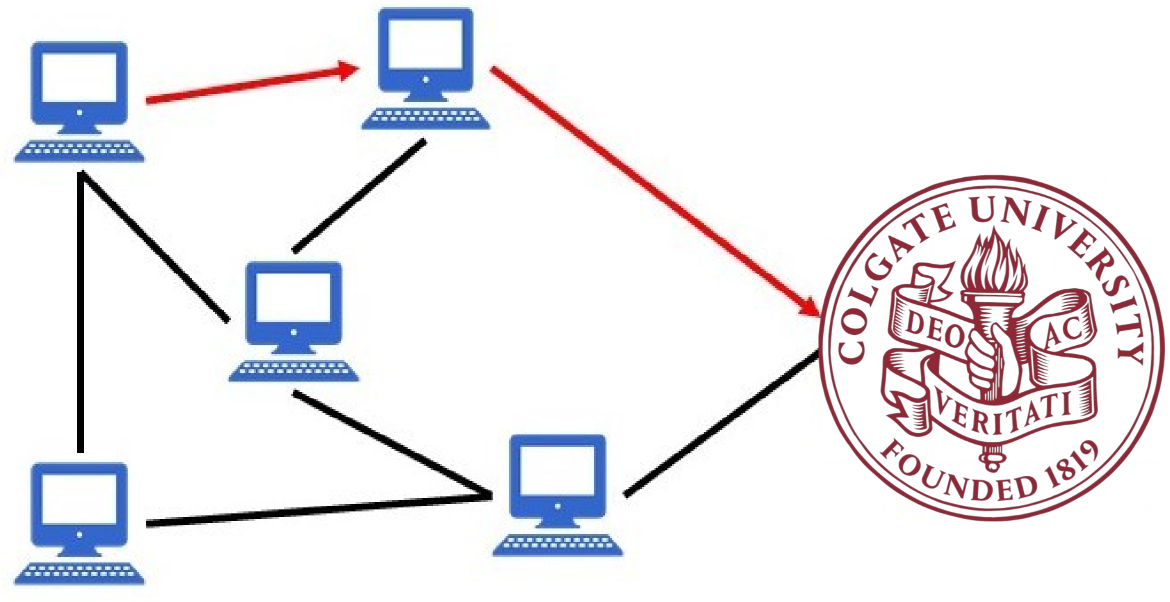
Finding the Shape of the Internet

I want to visit www.colgate.edu. What path through the internet will my HTTP request take?



Finding the Shape of the Internet

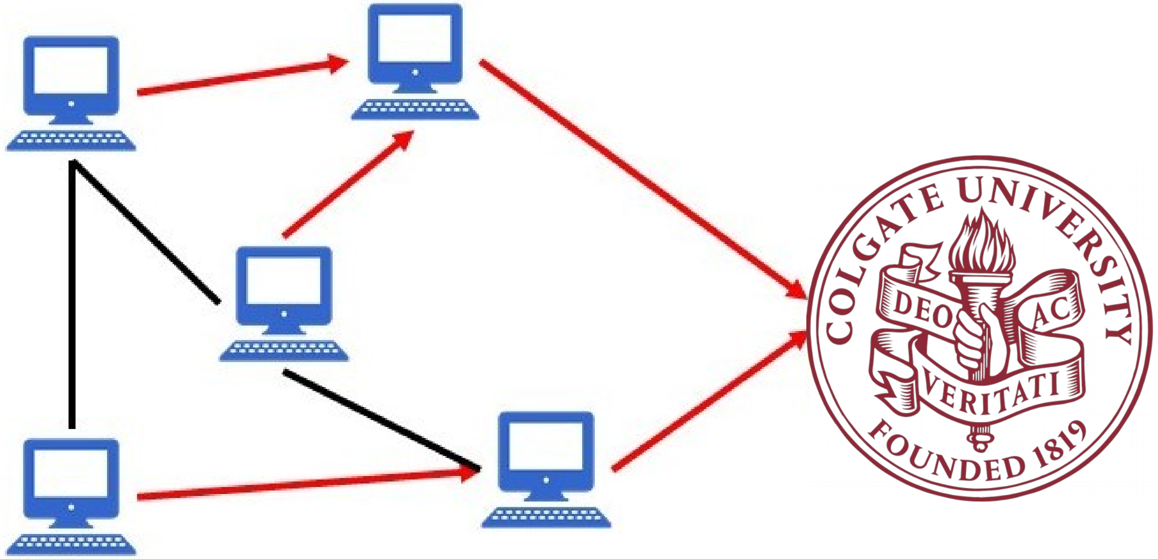
I want to visit www.colgate.edu. What path through the internet will my HTTP request take?



Finding the Shape of the Internet

I want to visit www.colgate.edu. What path through the internet will my HTTP request take?

How can I predict paths from *any* starting location to colgate.edu?

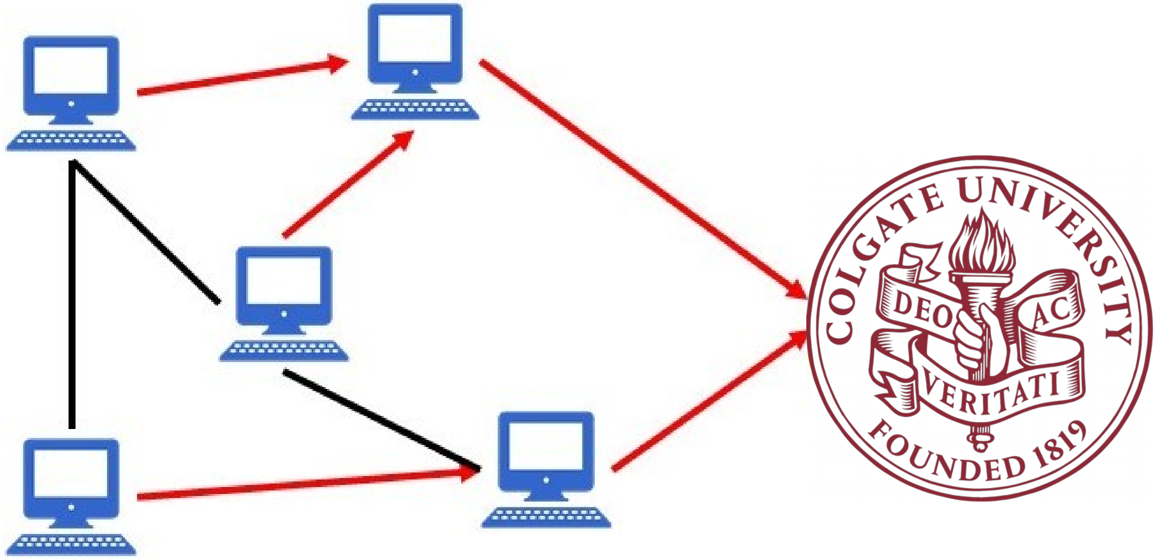


Finding the Shape of the Internet

I want to visit www.colgate.edu. What path through the internet will my HTTP request take?

How can I predict paths from *any* starting location to colgate.edu?

Useful in network measurement.



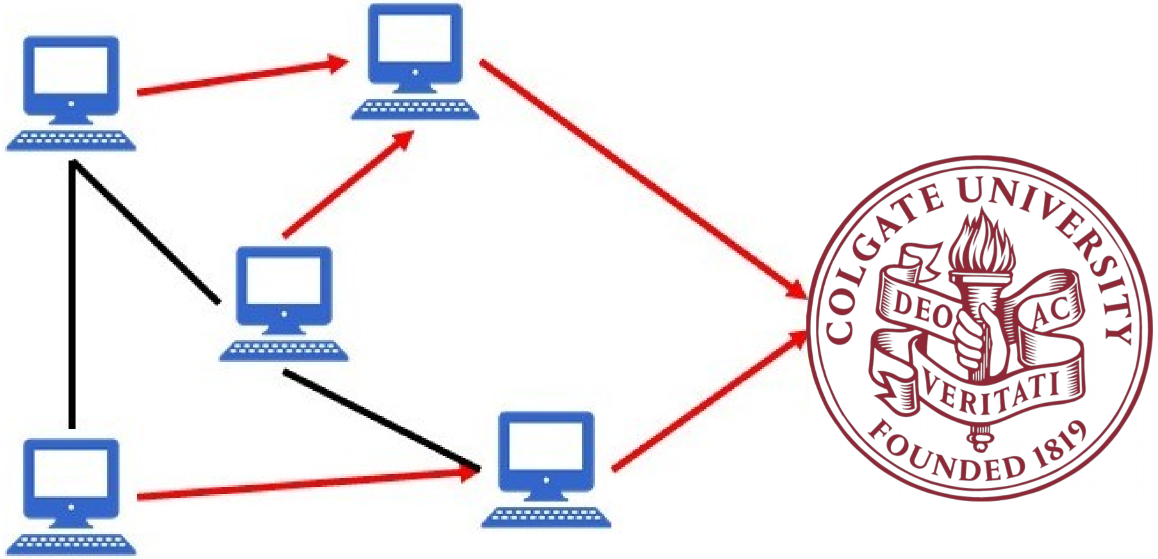
Finding the Shape of the Internet

I want to visit www.colgate.edu. What path through the internet will my HTTP request take?

How can I predict paths from *any* starting location to colgate.edu?

Useful in network measurement.

Ex: Can help identify censorship.



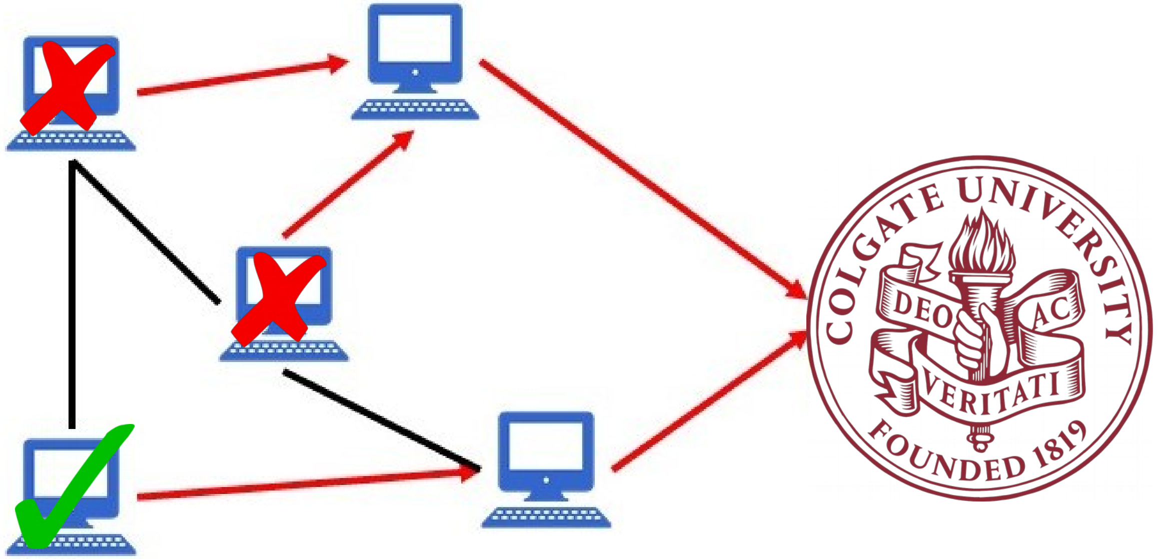
Finding the Shape of the Internet

I want to visit www.colgate.edu. What path through the internet will my HTTP request take?

How can I predict paths from *any* starting location to colgate.edu?

Useful in network measurement.

Ex: Can help identify censorship.



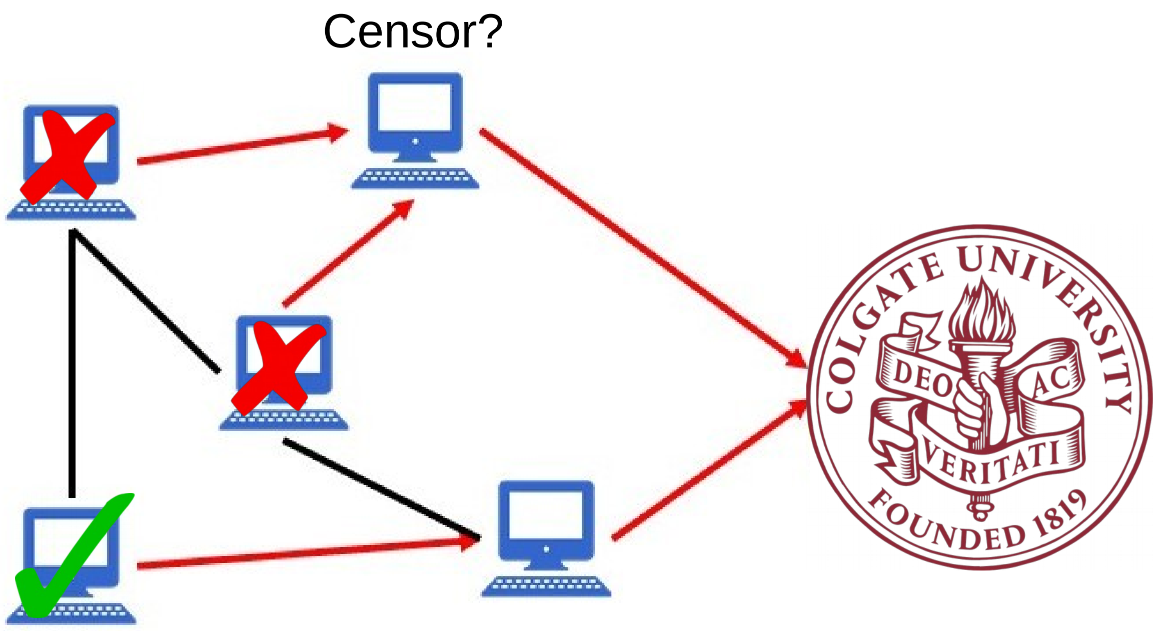
Finding the Shape of the Internet

I want to visit www.colgate.edu. What path through the internet will my HTTP request take?

How can I predict paths from *any* starting location to colgate.edu?

Useful in network measurement.

Ex: Can help identify censorship.

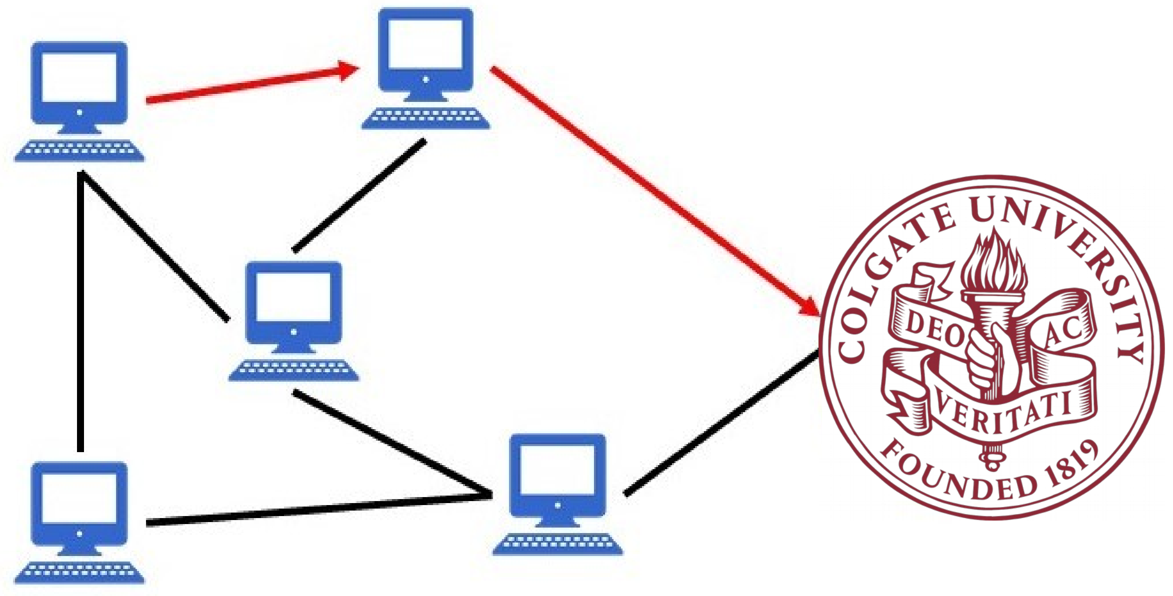


Finding the Shape of the Internet

I want to visit www.colgate.edu. What path through the internet will my HTTP request take?

How can I predict paths from *any* starting location to colgate.edu?

Measuring a path is a costly query called a *traceroute*. We want to discover as much of the Internet as possible using minimal traceroutes.

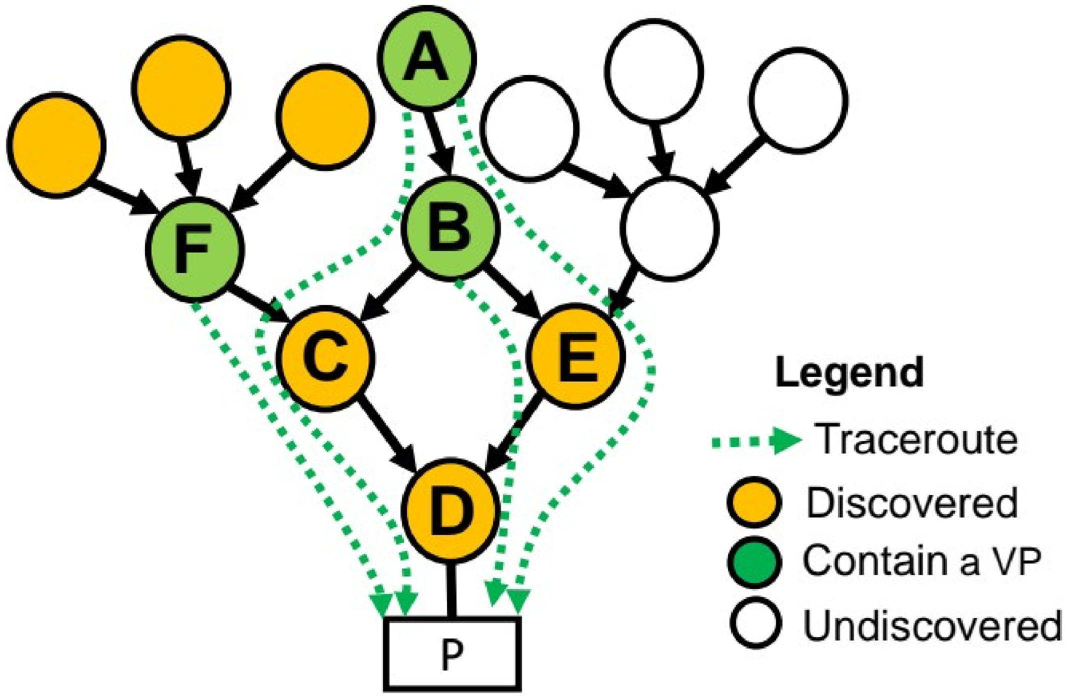


PathCache

A system for efficiently using limited VP measurements to predict paths towards Internet destinations.

Current version discovers 4 times more connections than pre-existing measurement strategies with comparable traceroute budget.

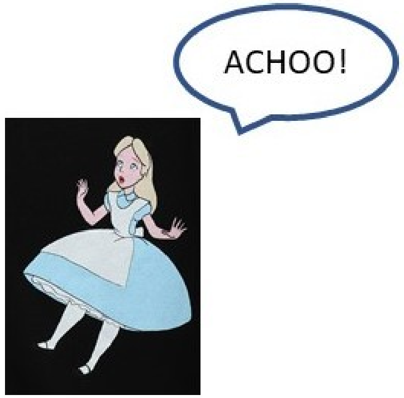
Predicts correct or nearly-correct paths 75% of the time.



Future Work

GRAPHS THAT ARE EVEN MORE INCONVENIENT

Time-Dependent Graph Streams

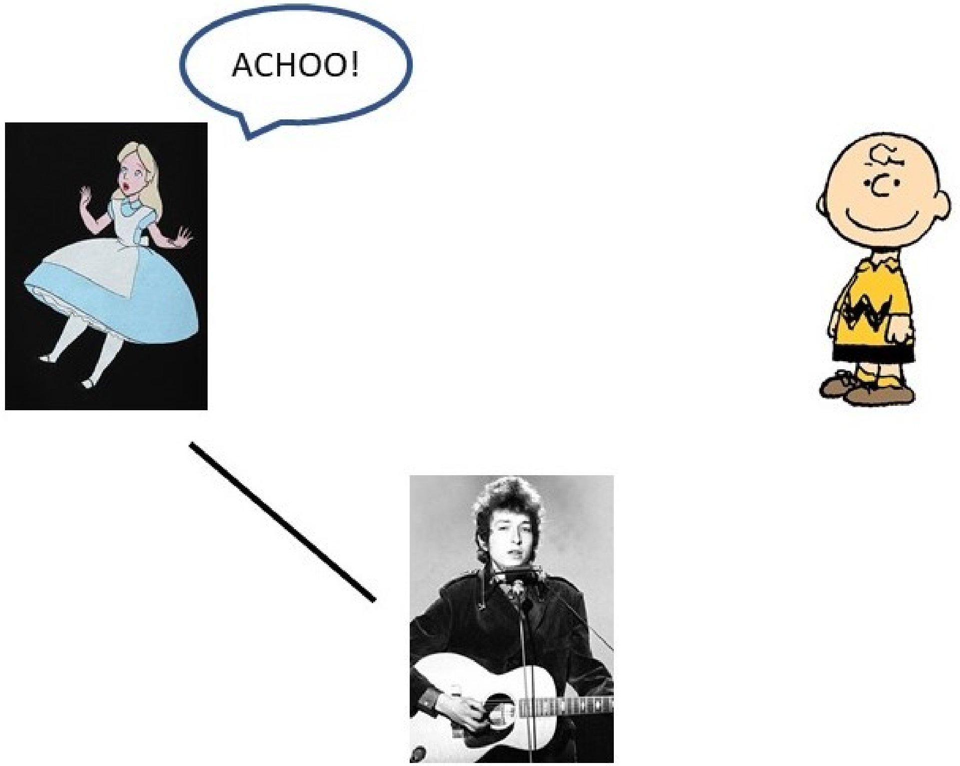


In the typical streaming model, graph is the same **regardless of the order edges appear in the stream.**

What if the order mattered?

Ex: disease spreading

Time-Dependent Graph Streams

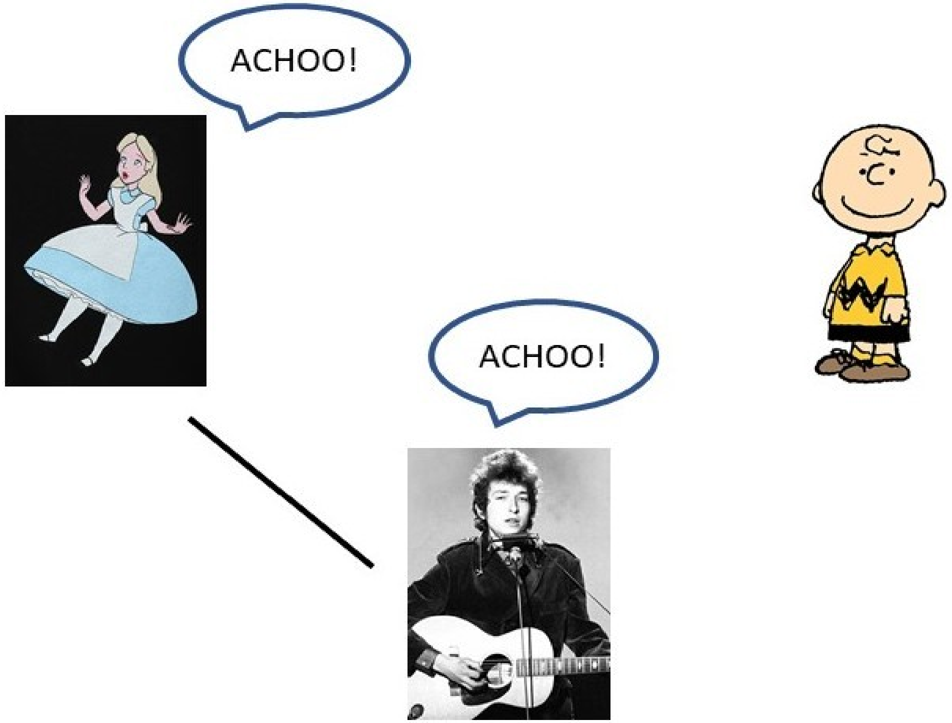


In the typical streaming model, graph is the same **regardless of the order edges appear in the stream.**

What if the order mattered?

Ex: disease spreading

Time-Dependent Graph Streams

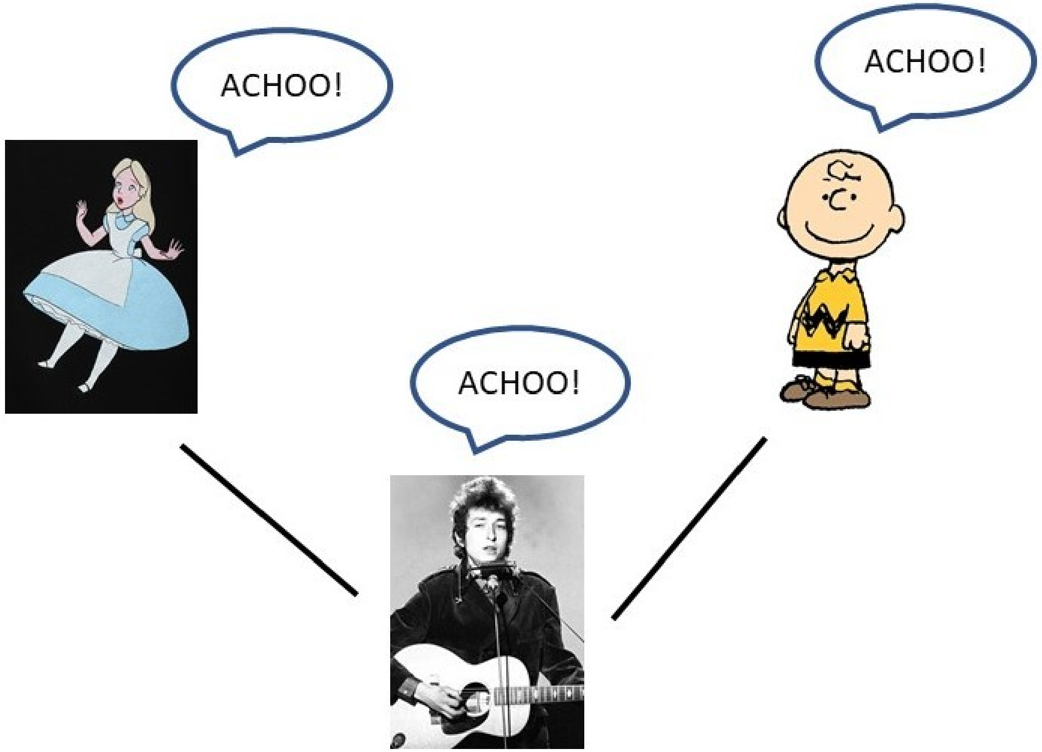


In the typical streaming model, graph is the same **regardless of the order edges appear in the stream.**

What if the order mattered?

Ex: disease spreading

Time-Dependent Graph Streams

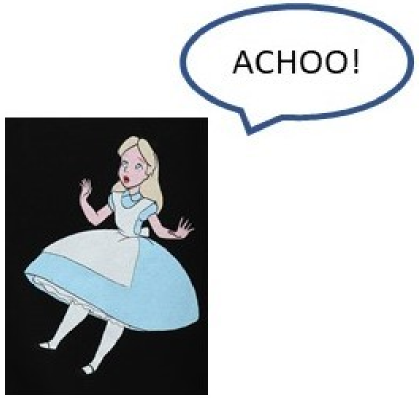


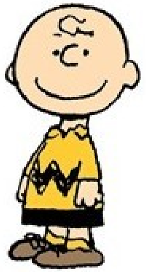
In the typical streaming model, graph is the same **regardless of the order edges appear in the stream.**

What if the order mattered?

Ex: disease spreading

Time-Dependent Graph Streams





In the typical streaming model, graph is the same **regardless of the order edges appear in the stream.**

What if the order mattered?

Ex: disease spreading

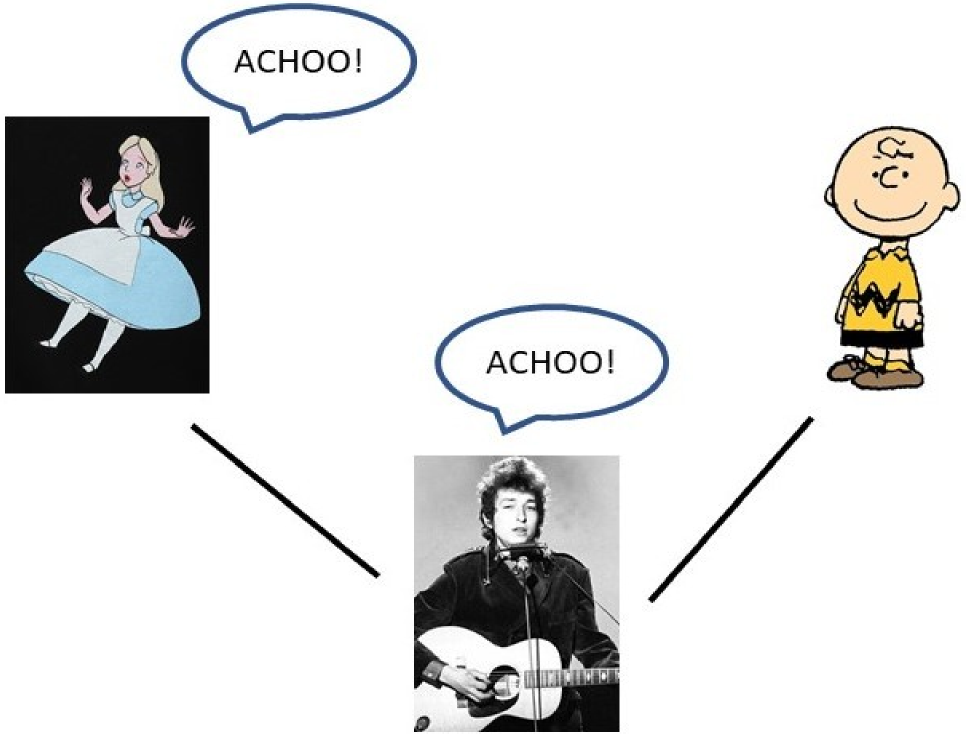
Time-Dependent Graph Streams

In the typical streaming model, graph is the same **regardless of the order edges appear in the stream.**

What if the order mattered?

Ex: disease spreading

Time-Dependent Graph Streams



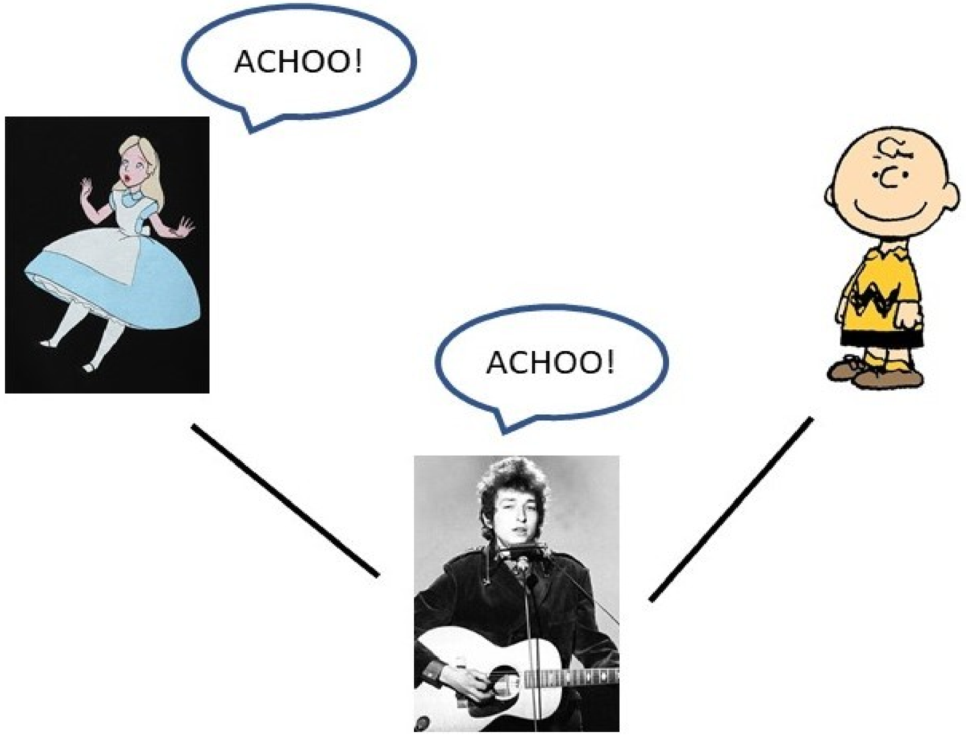
In the typical streaming model, graph is the same **regardless of the order edges appear in the stream.**

What if the order mattered?

Ex: disease spreading

Temporal graphs are just beginning to be investigated. No streaming work.

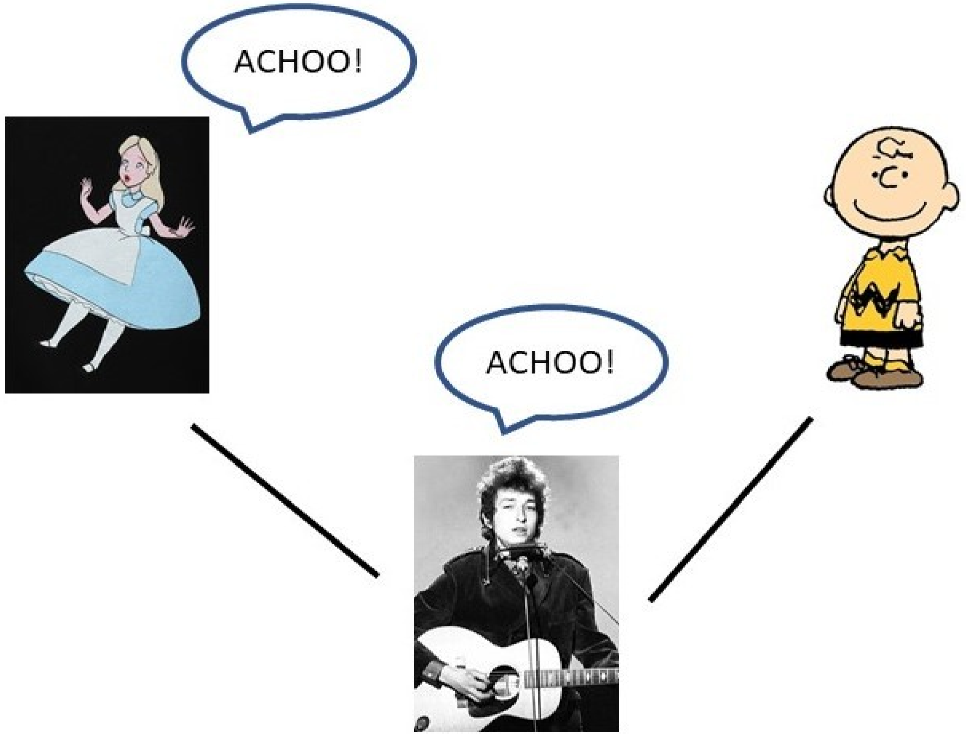
Time-Dependent Graph Streams



Imagine we receive a massive stream of handshakes between many people. Later, we learn one of those people was sick.

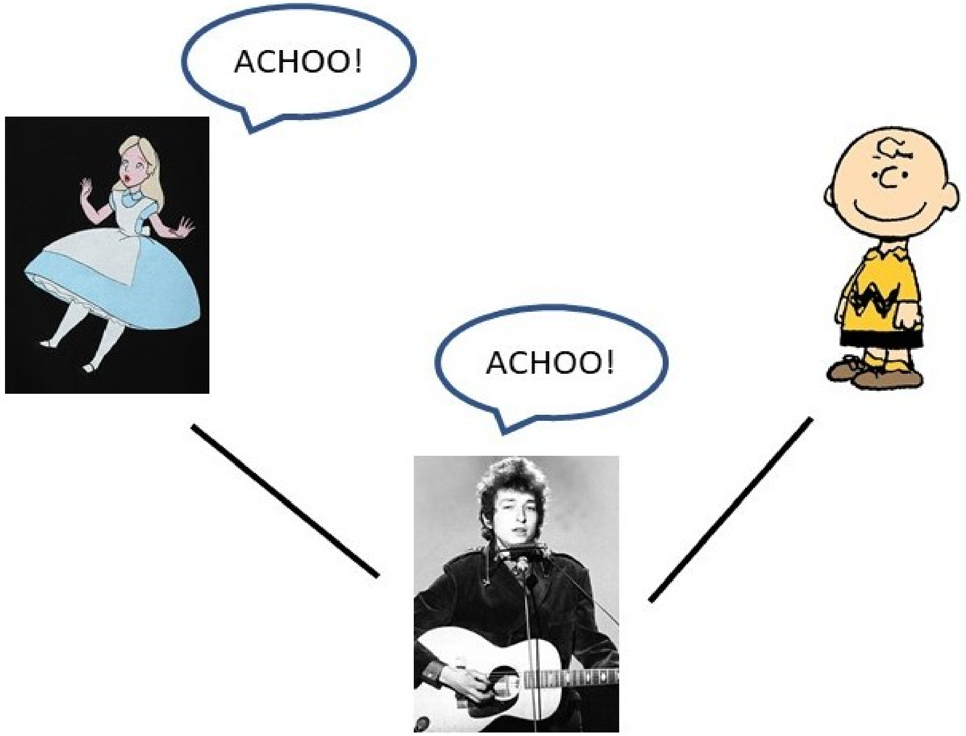
Can we determine who is infected without storing the entire stream?

Time-Dependent Graph Streams



This problem is about connectivity or reachability on temporal graphs.

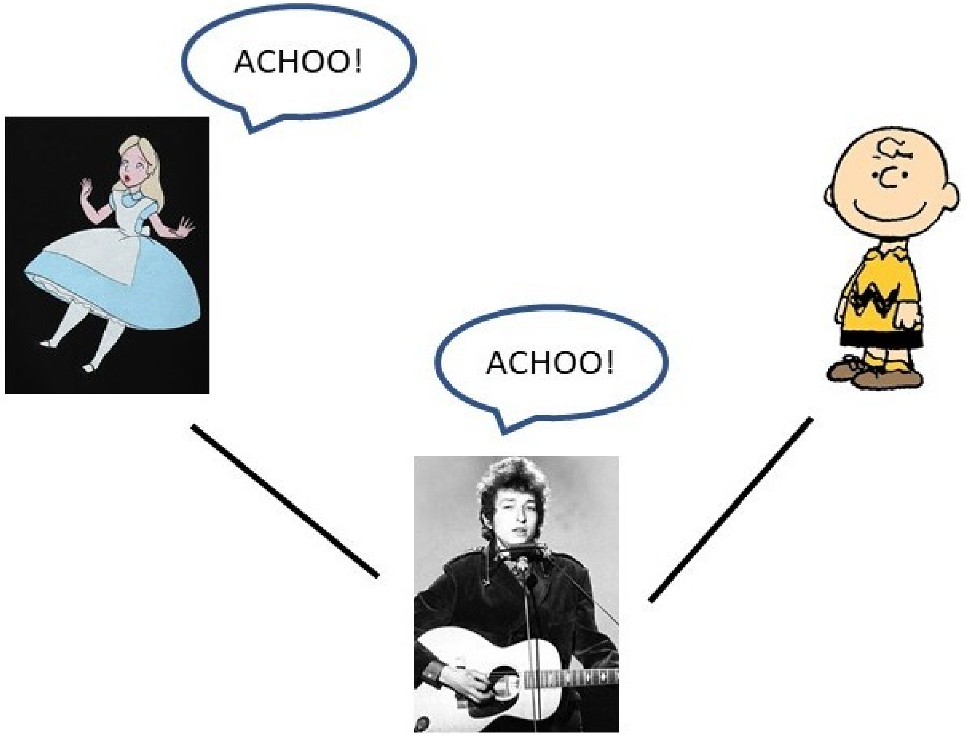
Time-Dependent Graph Streams



This problem is about connectivity or reachability on temporal graphs.

Other potential problems:

Time-Dependent Graph Streams

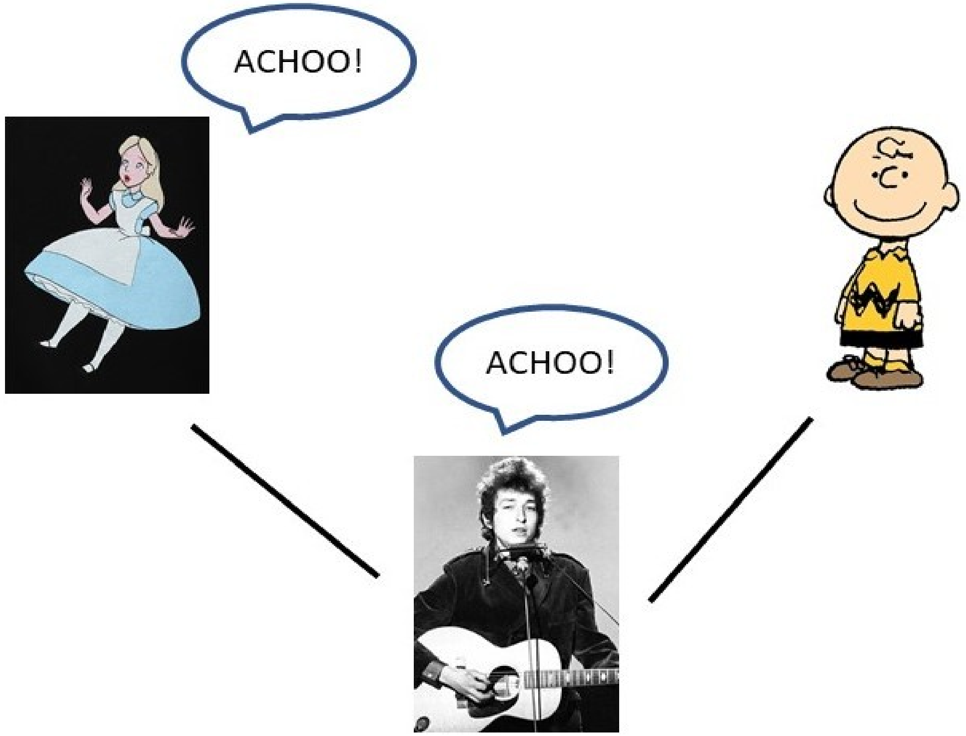


This problem is about connectivity or reachability on temporal graphs.

Other potential problems:

Determine how long it takes for a disease (or information, or goods) to reach every part of a network.

Time-Dependent Graph Streams



This problem is about connectivity or reachability on temporal graphs.

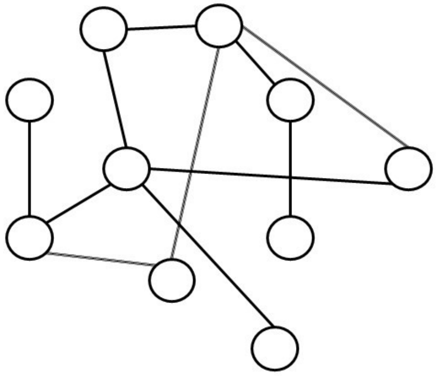
Other potential problems:

Determine how long it takes for a disease (or information, or goods) to reach every part of a network.

Estimate how many different Patient Zeros could infect a particular person.

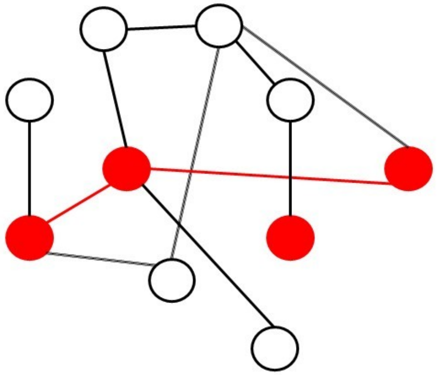
Reconstructing Graphs

We want to reconstruct a graph G.
We can make a query which returns a random, unlabeled induced subgraph of G. How many queries are needed?



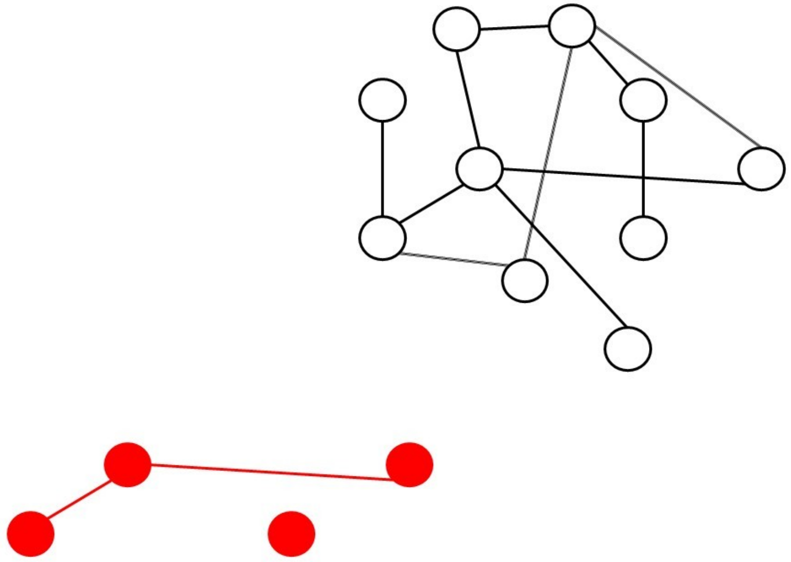
Reconstructing Graphs

We want to reconstruct a graph G.
We can make a query which returns a random, unlabeled induced subgraph of G. How many queries are needed?



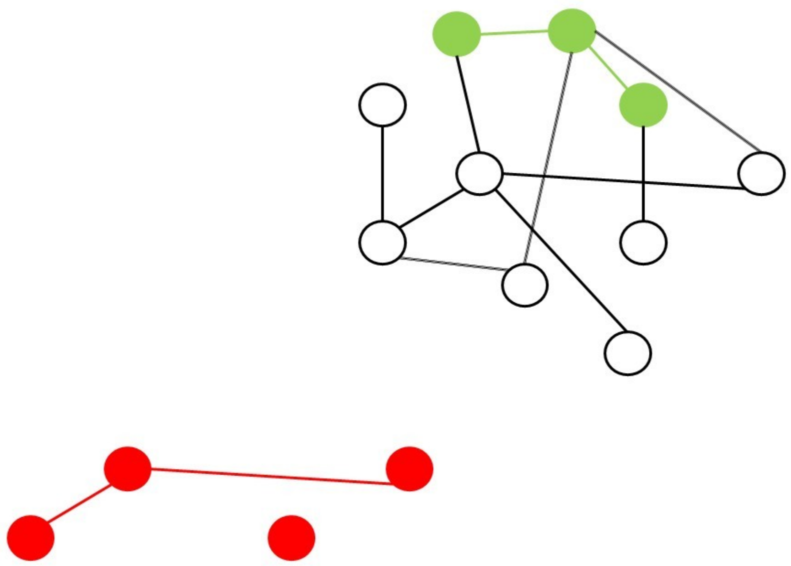
Reconstructing Graphs

We want to reconstruct a graph G.
We can make a query which returns a random, unlabeled induced subgraph of G. How many queries are needed?



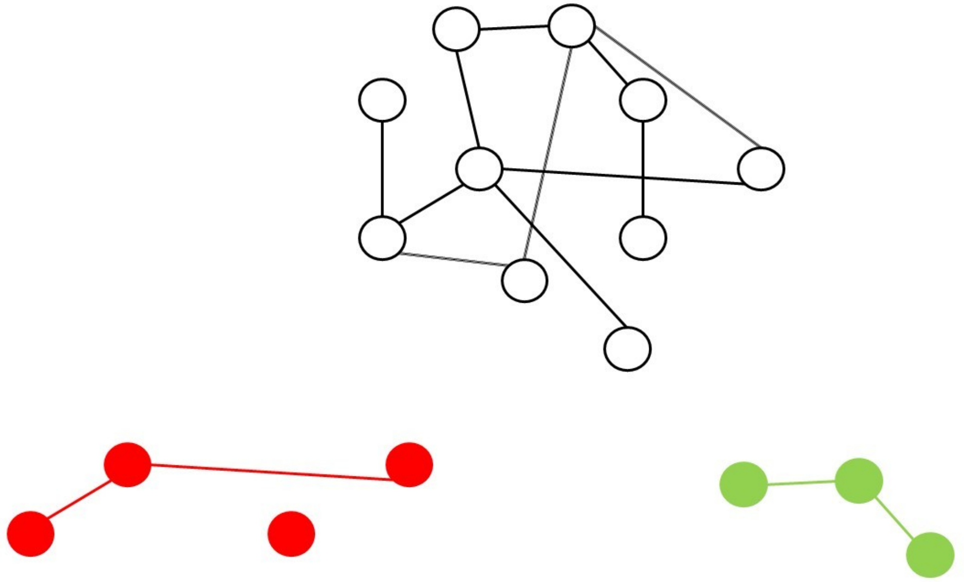
Reconstructing Graphs

We want to reconstruct a graph G.
We can make a query which returns a random, unlabeled induced subgraph of G. How many queries are needed?



Reconstructing Graphs

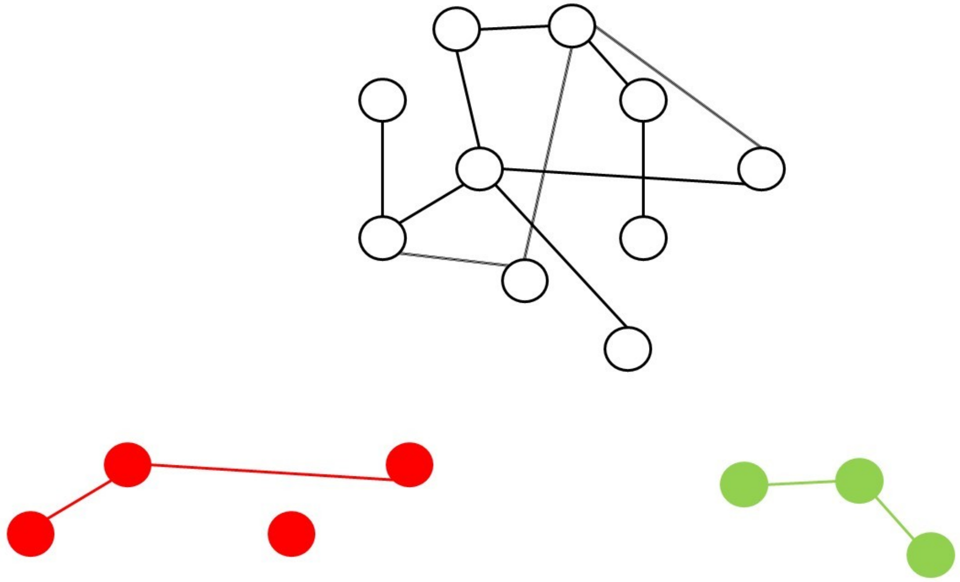
We want to reconstruct a graph G.
We can make a query which returns a random, unlabeled induced subgraph of G. How many queries are needed?



Reconstructing Graphs

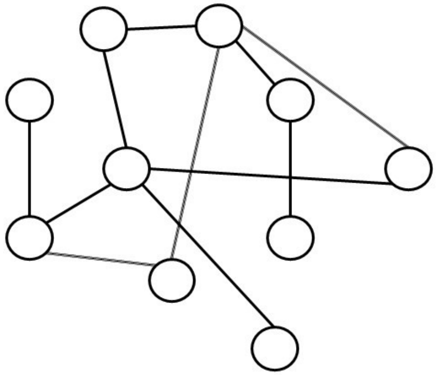
We want to reconstruct a graph G.
We can make a query which returns a random, unlabeled induced subgraph of G. How many queries are needed?

More generally, we must reconstruct a matrix M and can make queries which return a submatrix where rows and columns are deleted randomly. How many queries are needed?

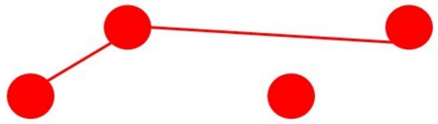


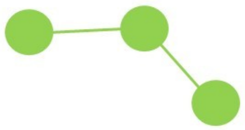
Reconstructing Graphs

We want to reconstruct a graph G.
We can make a query which returns a random, unlabeled induced subgraph of G. How many queries are needed?



More generally, we must reconstruct a matrix M and can make queries which return a submatrix where rows and columns are deleted randomly.
How many queries are needed?





Reconstruction from other queries?

Thanks for listening!