Algorithms for Massive, Expensive, and Otherwise Inconvenient Graphs

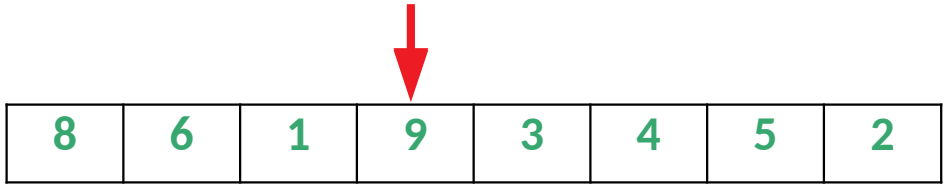
DAVID TENCH

UNIVERSITY OF MASSACHUSETTS AMHERST

Convenient Inputs

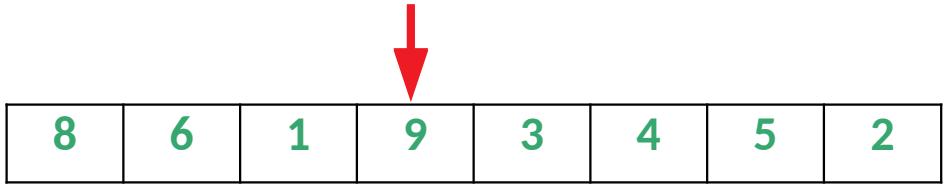
Convenient Inputs

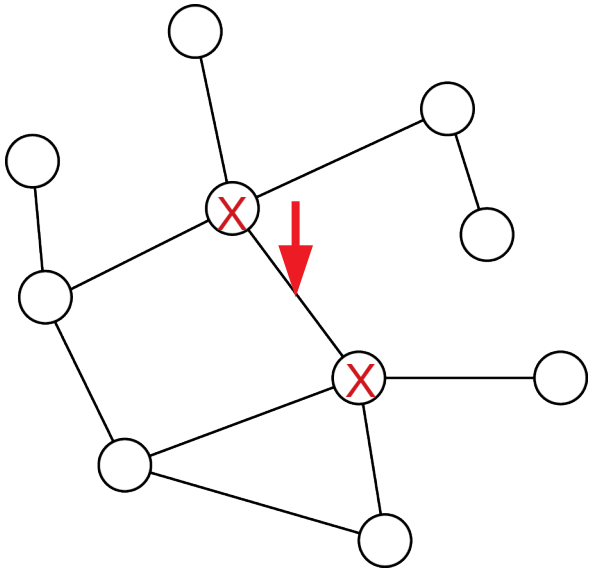
- What is the 4th element of this list?



Convenient Inputs

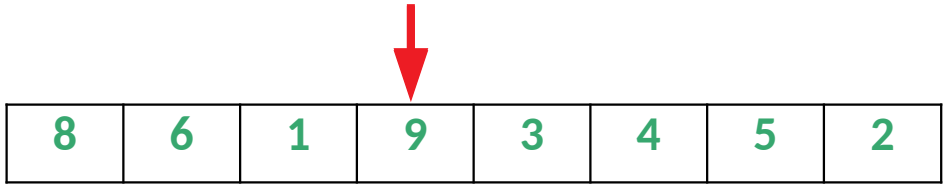
- What is the 4th element of this list?
- Is there an edge between some pair of nodes in this graph?

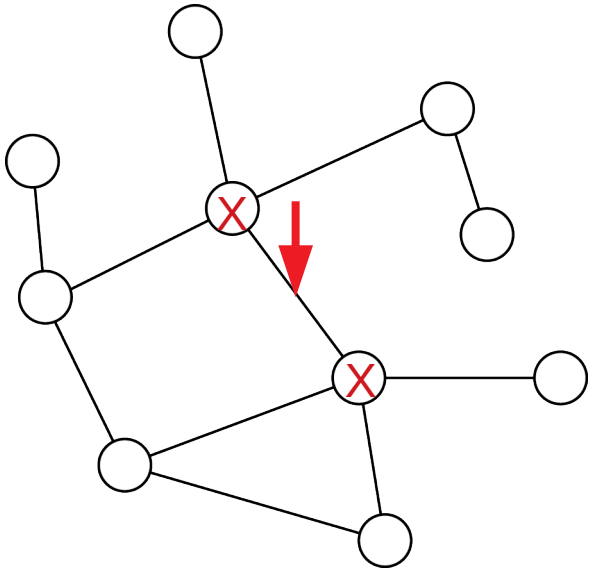




Convenient Inputs

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

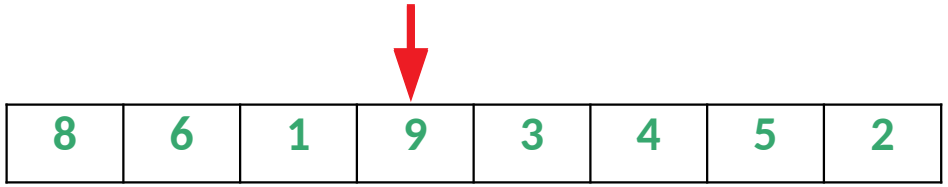


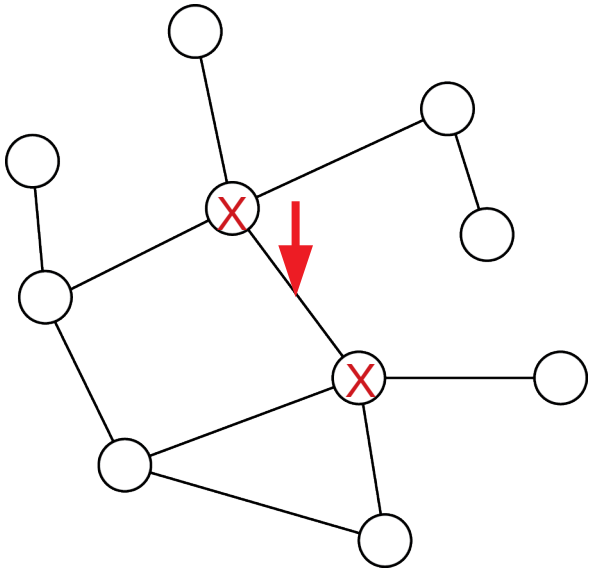


An algorithm can access **any part** of its input at **any time** at **unit cost**.

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

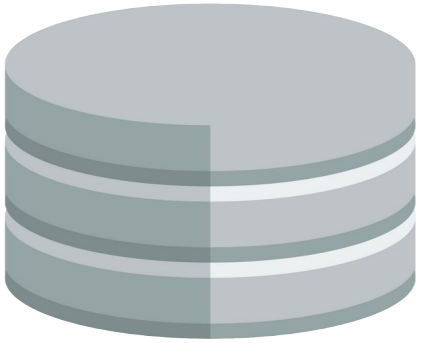
When inputs are too large to fit in memory

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



Previously Covered Work

Streaming graph algorithms: Approximating vertex connectivity in small space

Previously Covered Work

Streaming graph algorithms: Approximating vertex connectivity in small space

I presented algorithms that, given a stream of edges that define massive graph G, compute the smallest set of nodes whose removal would disconnect G.

(this is hard because input is **large**)

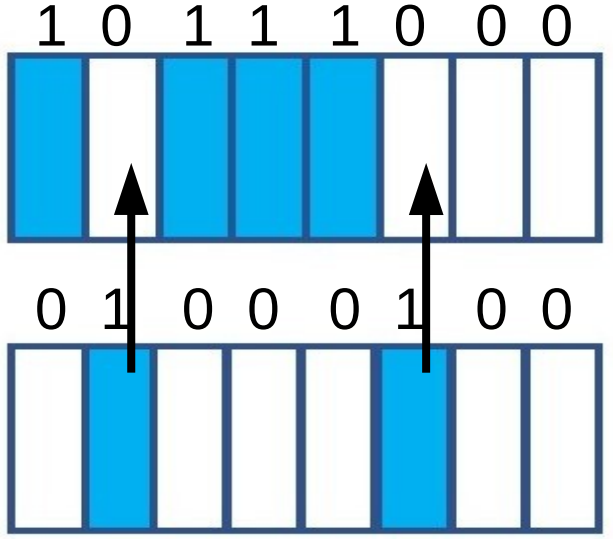
Previously Covered Work

I also presented Mesh, a system that performs memory compaction that was thought to be impossible.

Previously Covered Work

I also presented Mesh, a system that performs memory compaction that was thought to be impossible.

Mesh: “Impossible” C and C++ Memory Compaction via Random Graph Algorithms



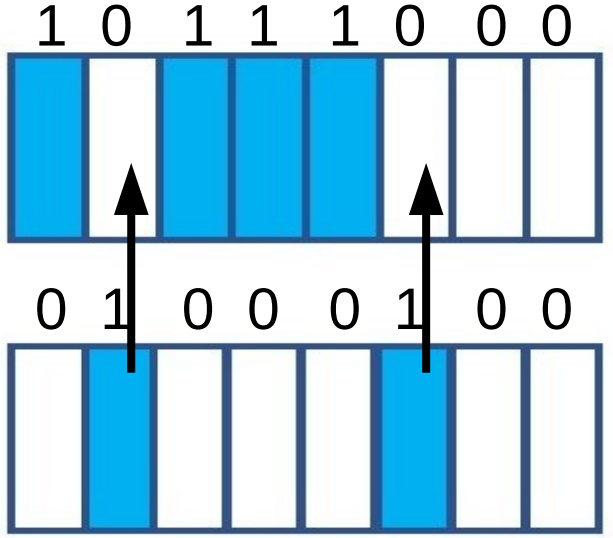
Previously Covered Work

I also presented Mesh, a system that performs memory compaction that was thought to be impossible.

Mesh does this by finding an approximately optimal maximum matching on a graph whose edges are only accessible via costly queries.

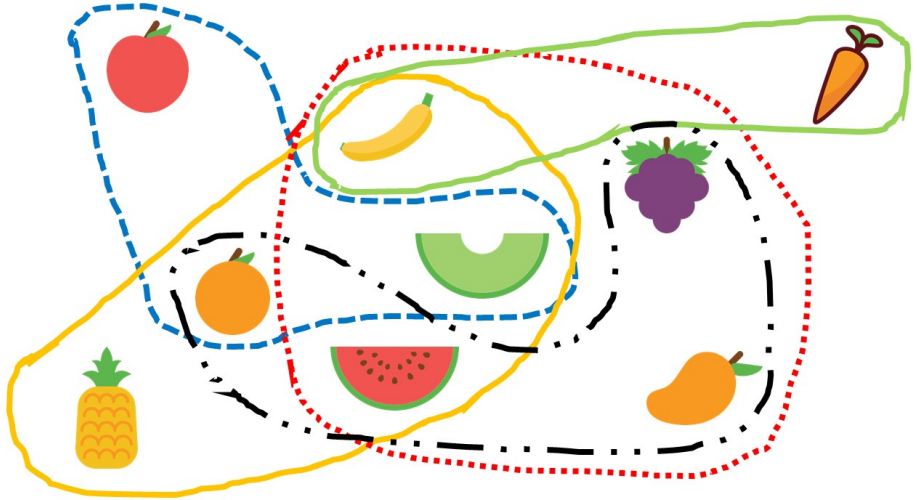
(this is hard because input is **costly**)

Mesh: “Impossible” C and C++ Memory Compaction via Random Graph Algorithms



Overview of this talk

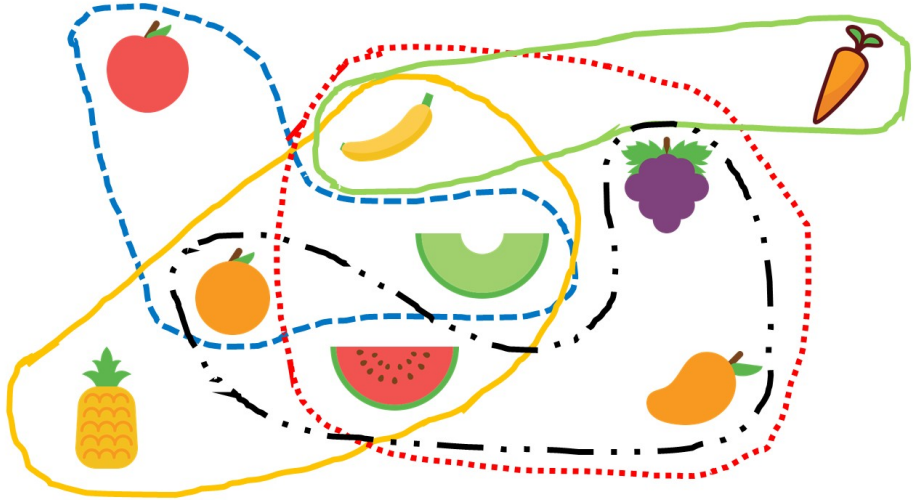
Streaming hypergraph algorithms: Computing maximum (unique) coverage



(this is hard because input is **large**)

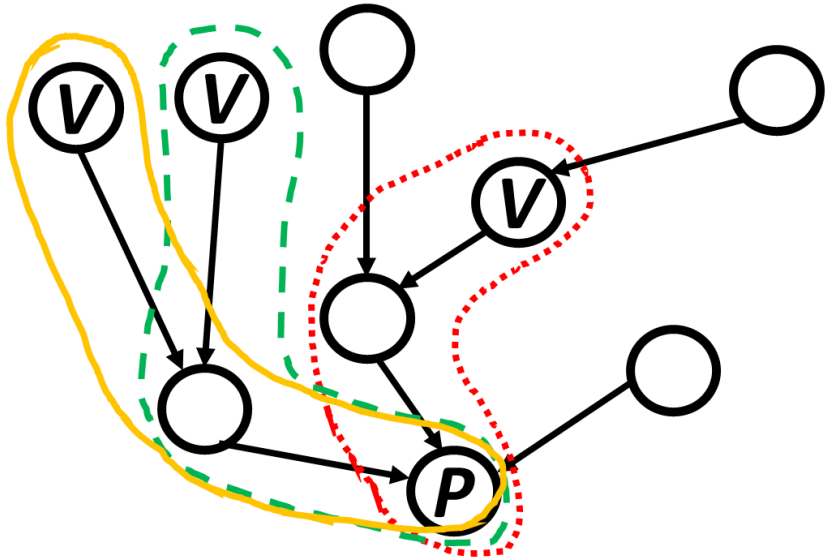
Overview of this talk

Streaming hypergraph algorithms: Computing maximum (unique) coverage



(this is hard because input is **large**)

PathCache: predicting Internet paths via costly measurements

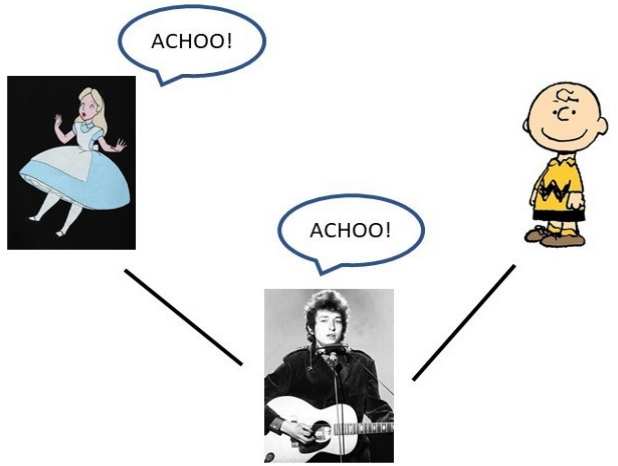


(this is hard because input is **costly**)

Overview of this talk

- Streaming hypergraph algorithms for coverage
- PathCache: Internet mapping

If we have time:
Temporal graph streams & applications to infection tracking



(this is hard because input is **large**)

Overview of this talk

- Streaming hypergraph algorithms for coverage
- PathCache: Internet mapping
- Temporal graph streams (?)

Future Research Directions

Overview of this talk

- Streaming hypergraph algorithms for coverage
- PathCache: Internet mapping
- Temporal graph streams (?)
- Future Research Directions

Questions before we begin?

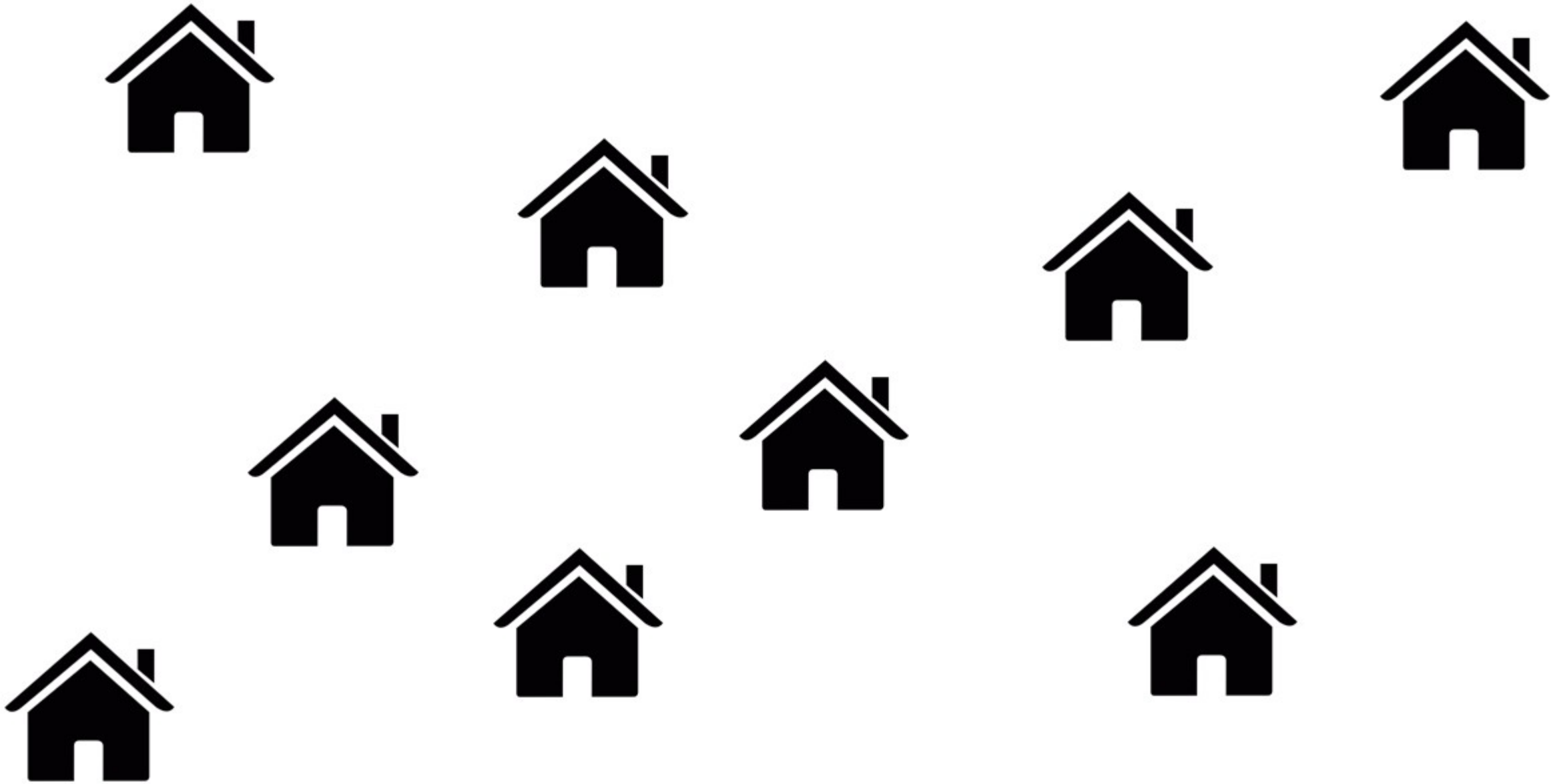
Coverage Problems in Streams

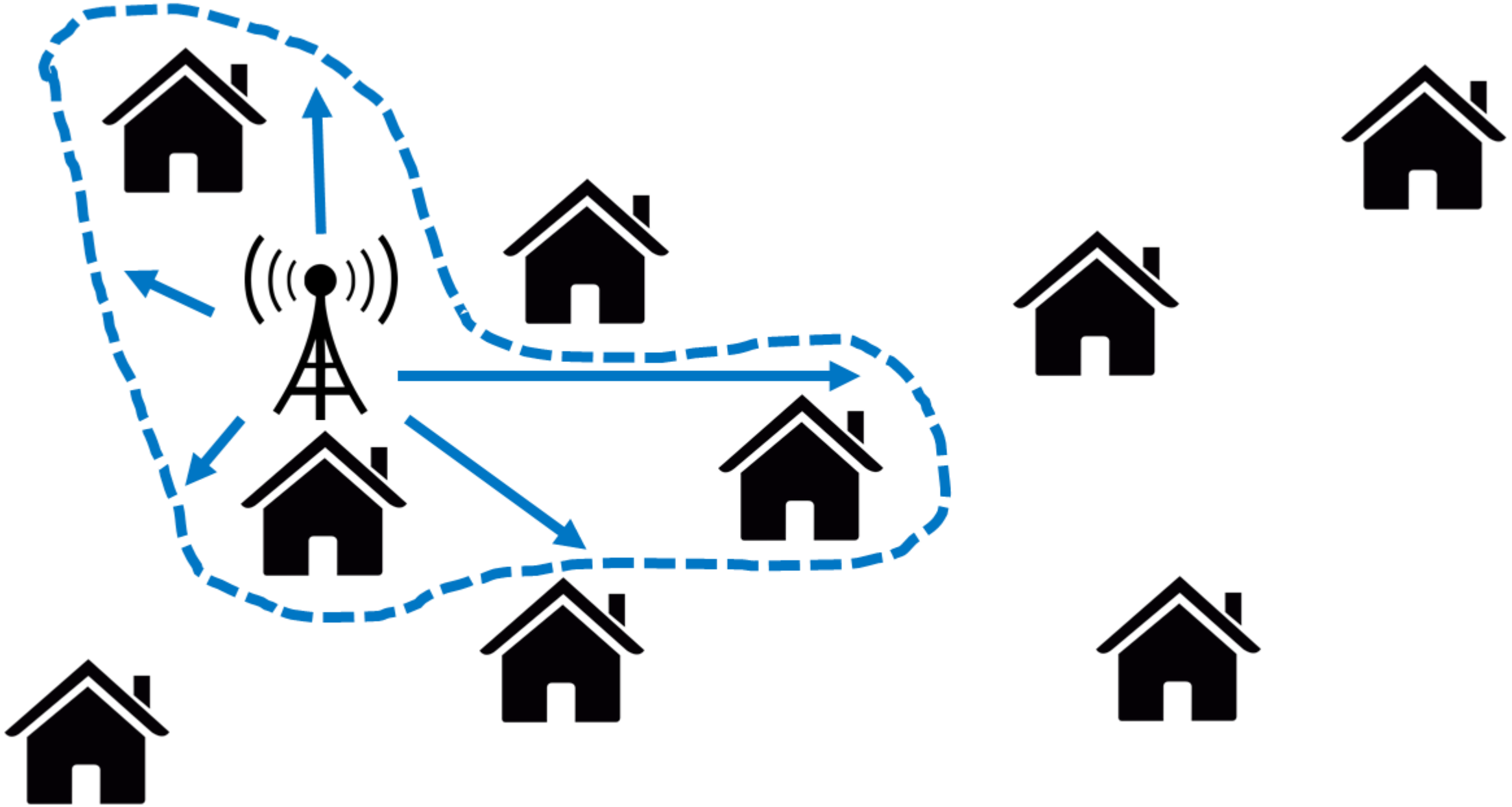
COMPUTING WITH INCREDIBLY LARGE INPUTS

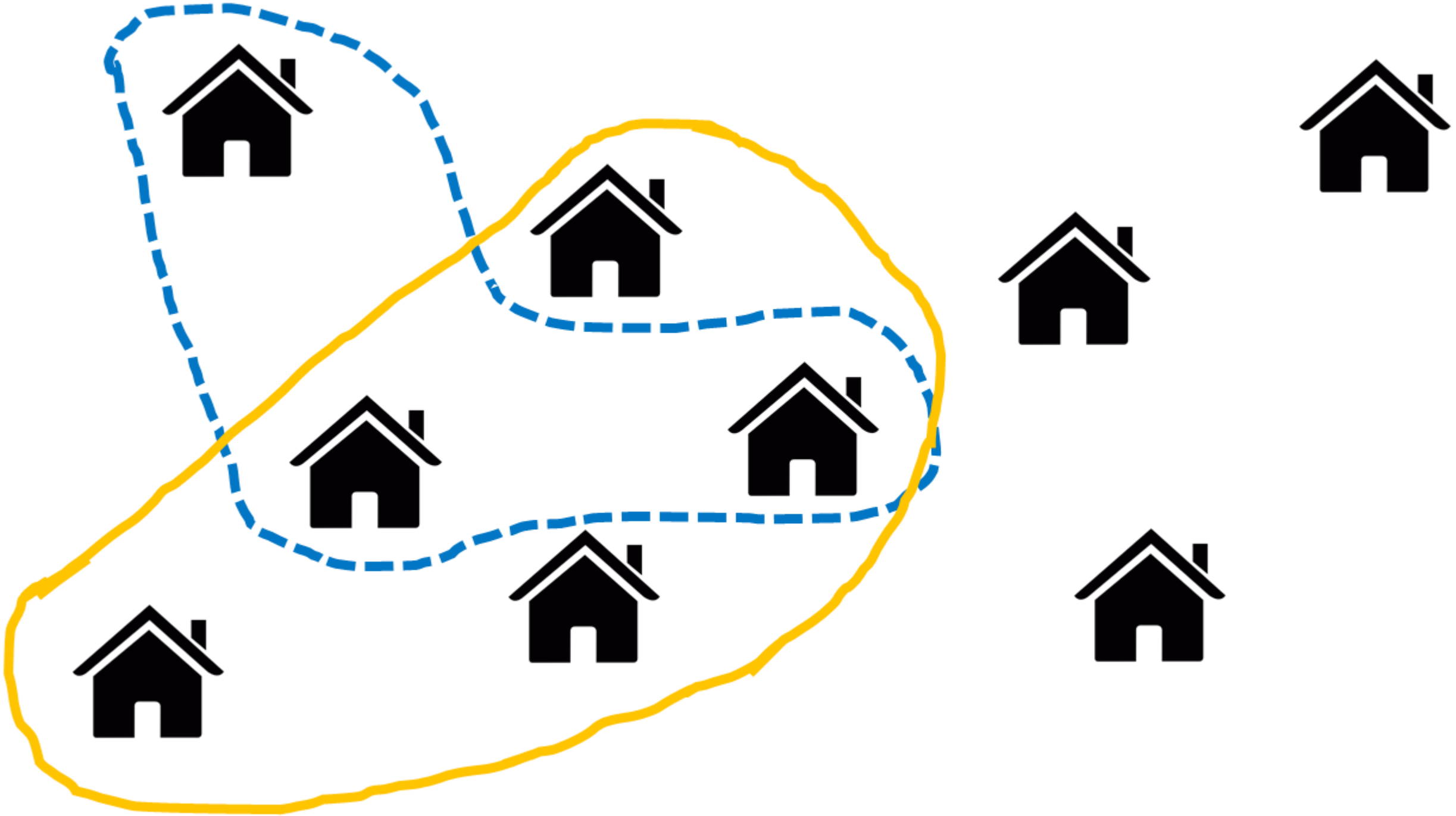
Joint work with:

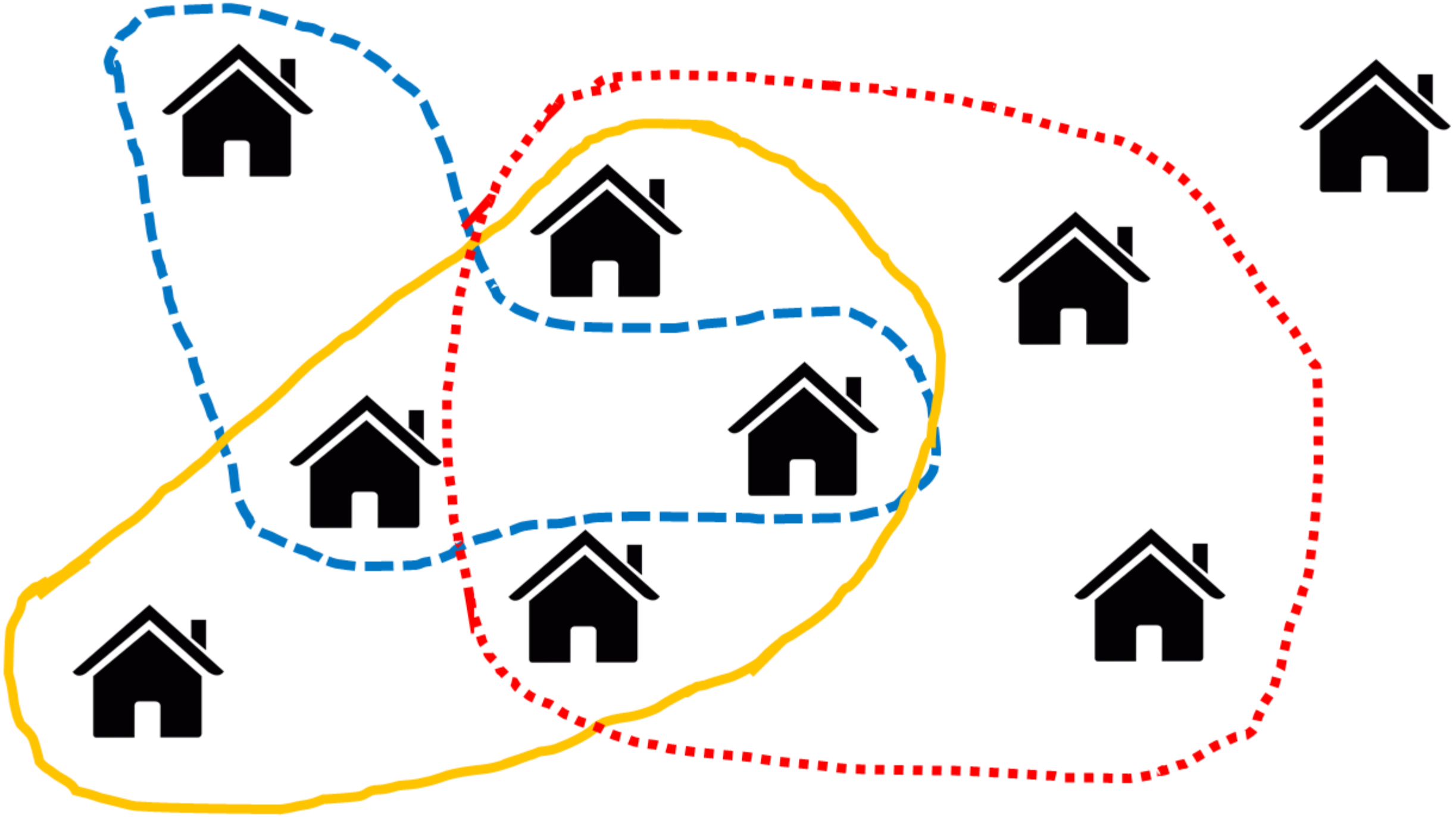
Andrew McGregor

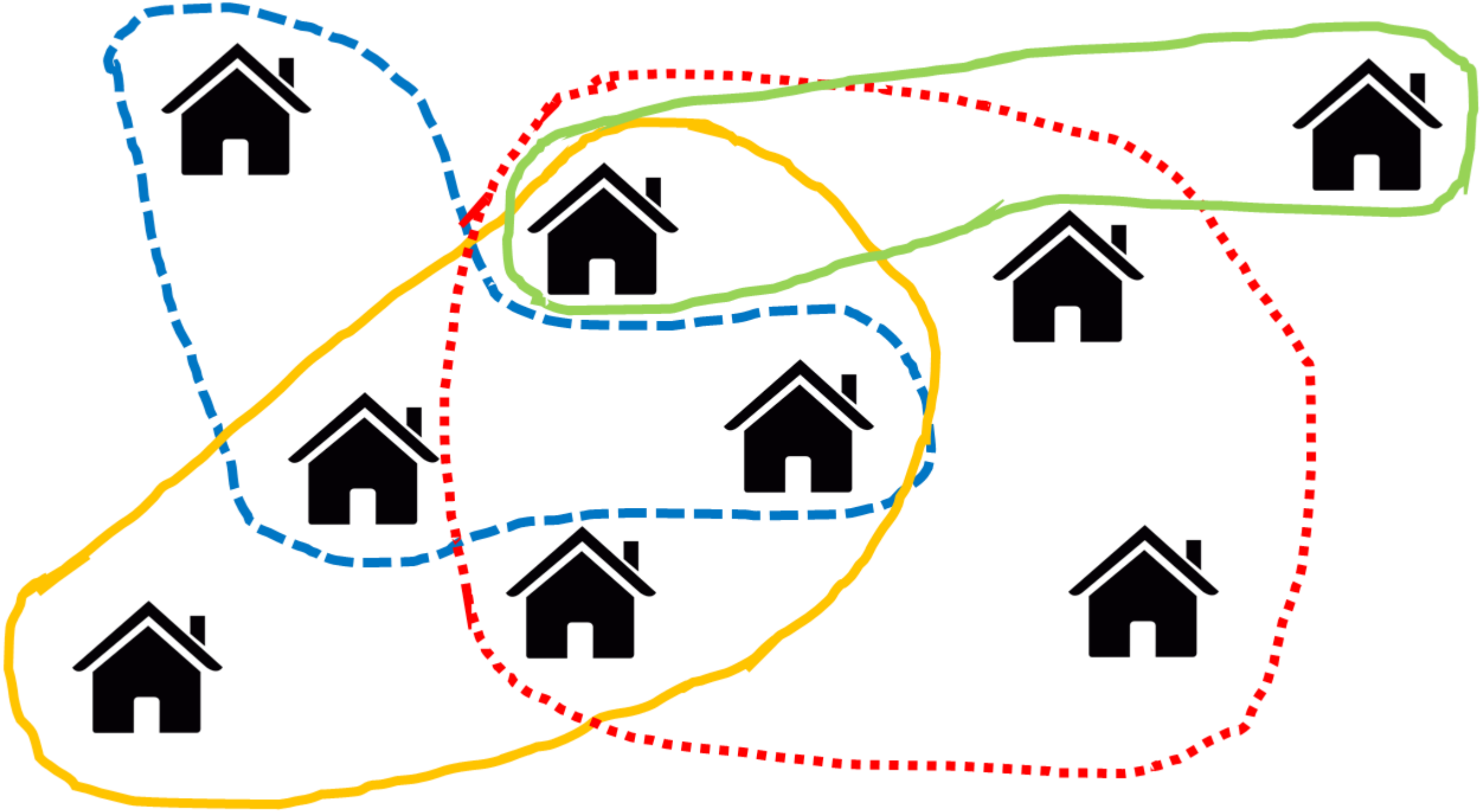
Hoa T. Vu

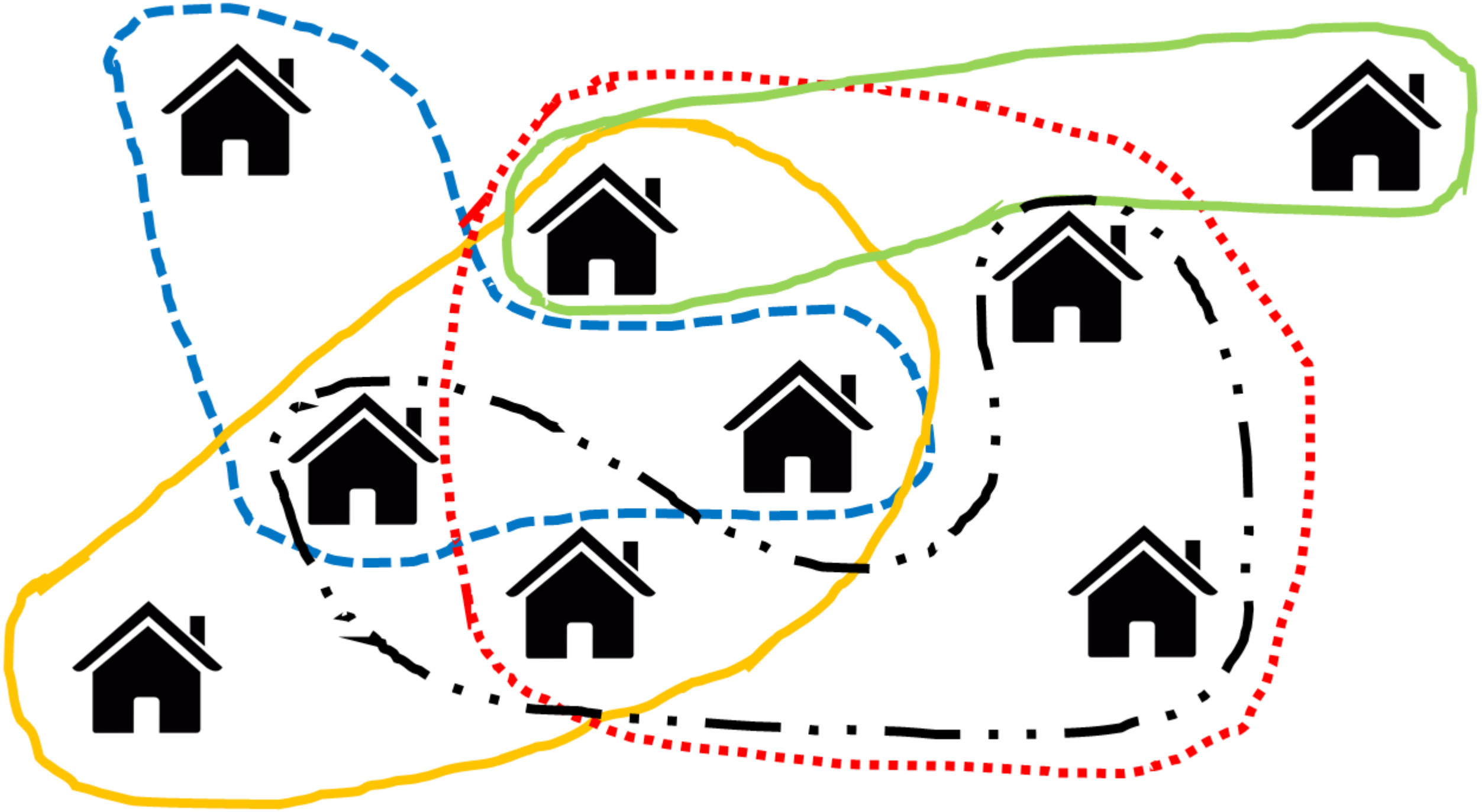


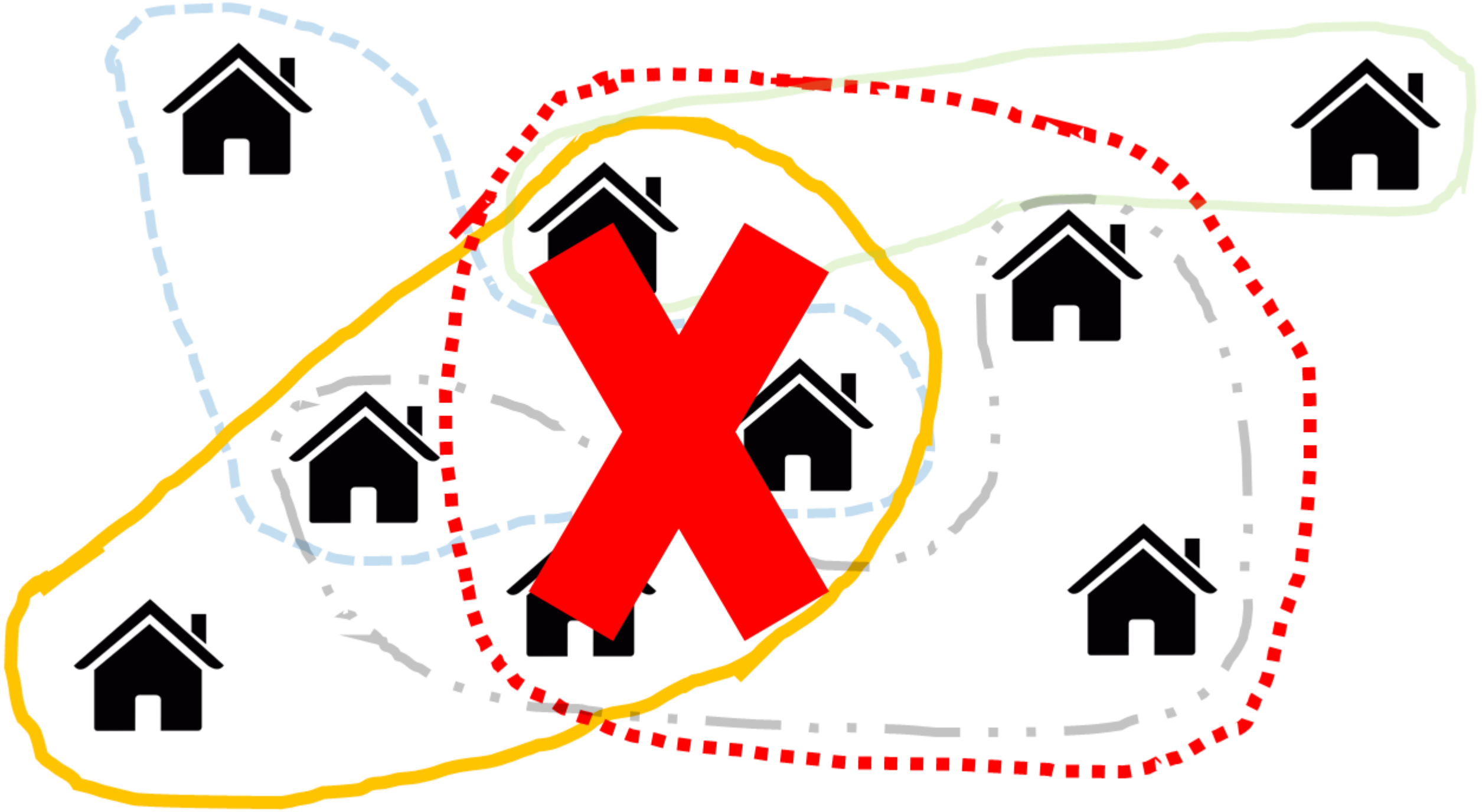


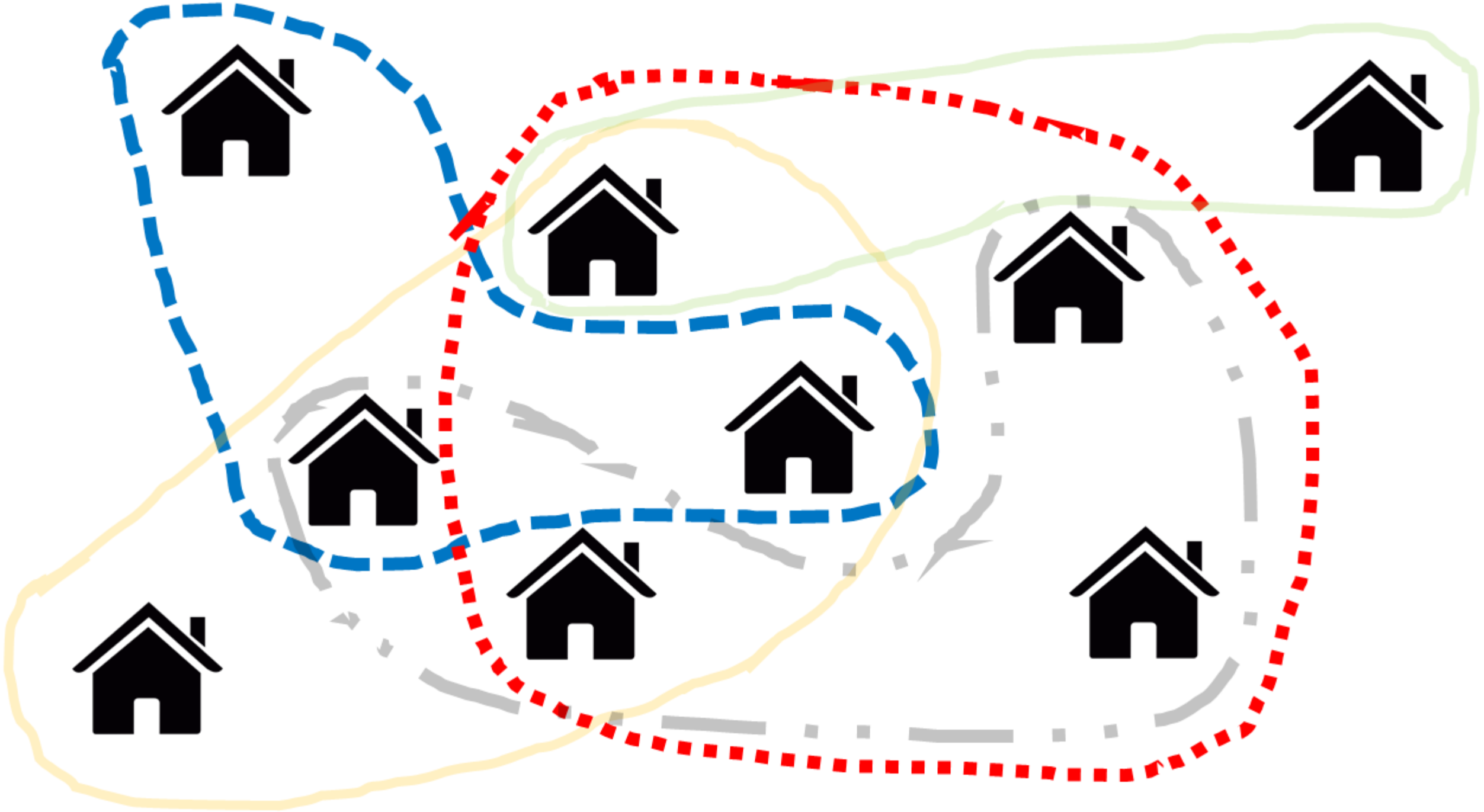


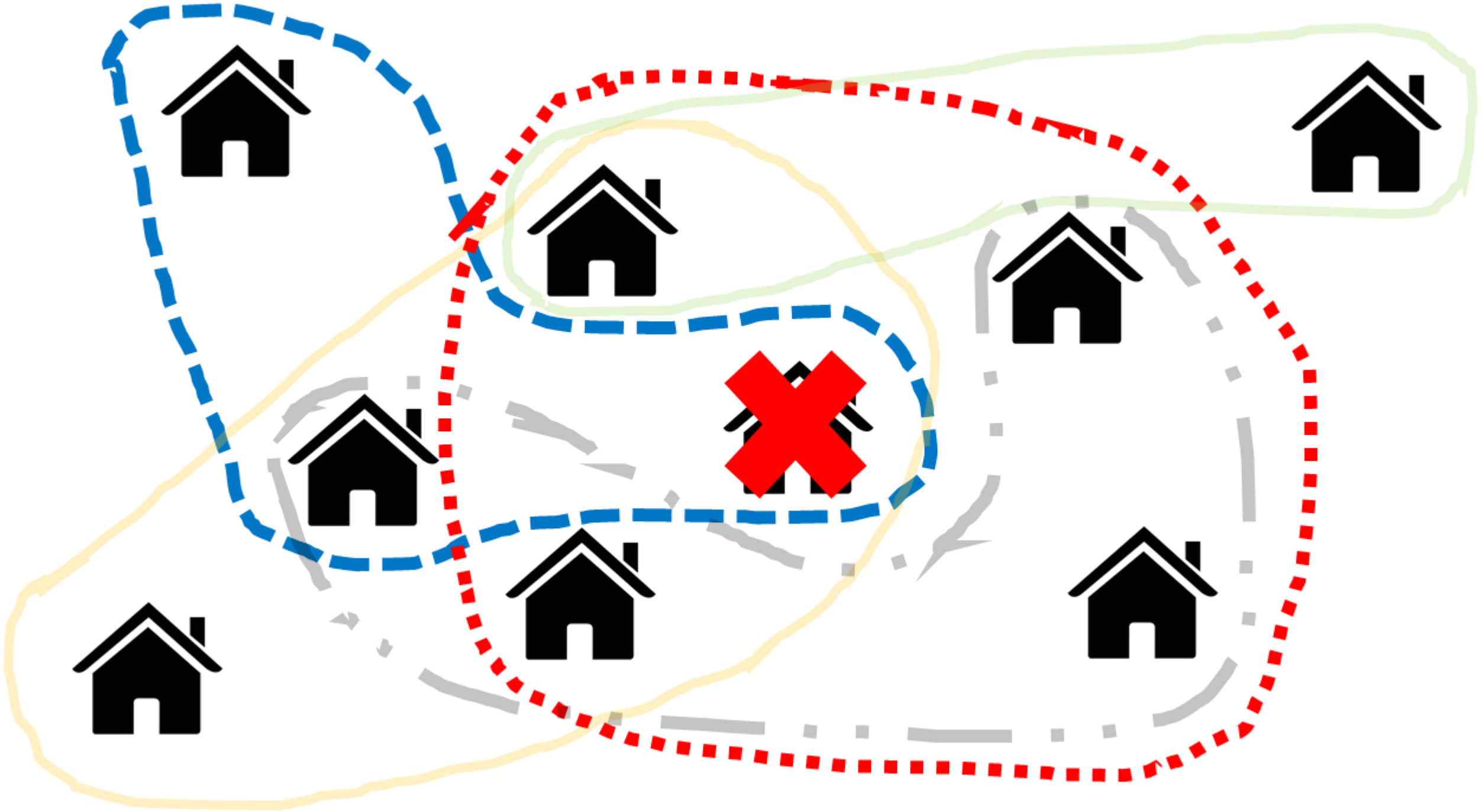






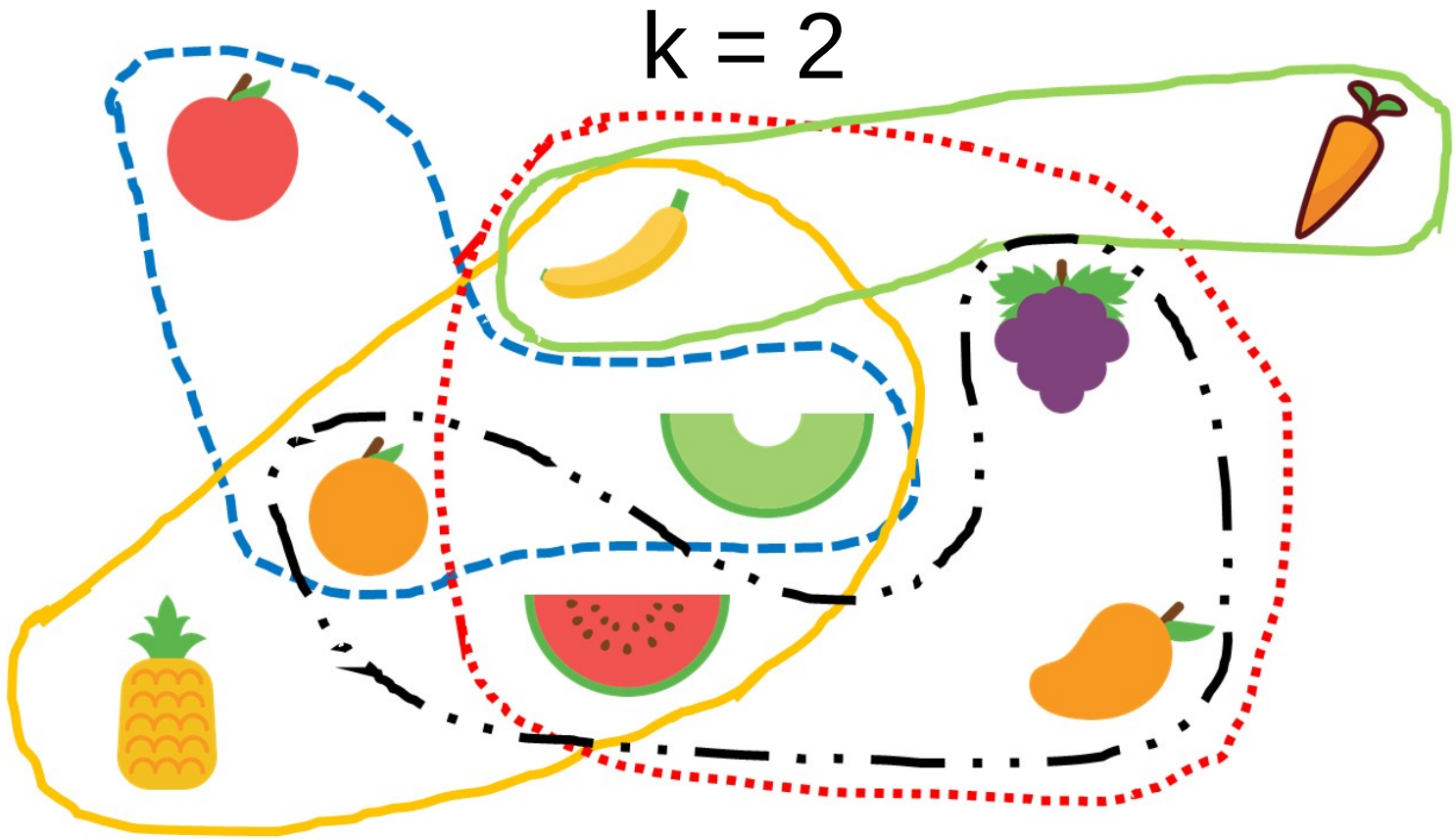






Maximum Coverage

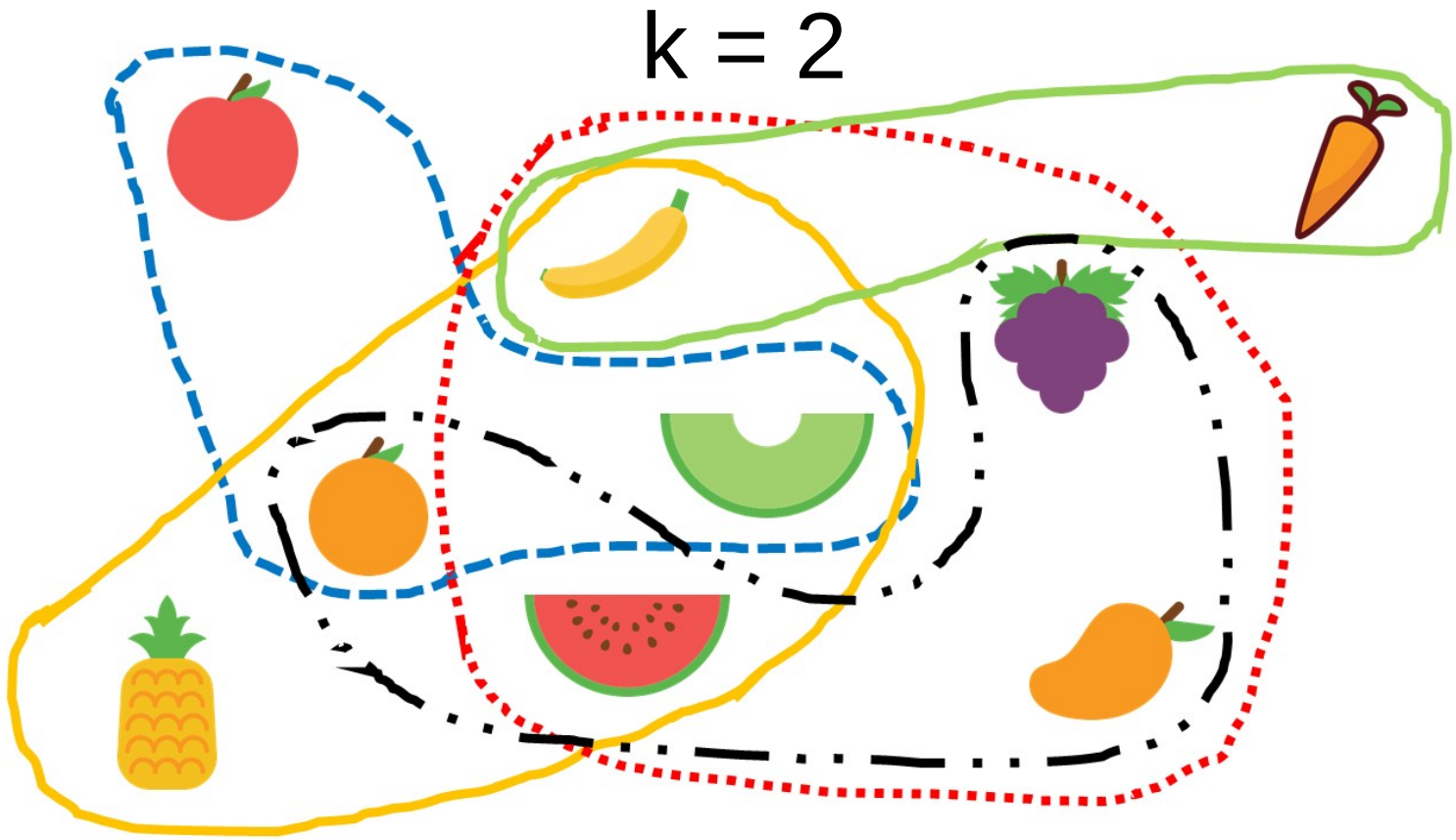
These are examples of *coverage* problems.



Maximum Coverage

These are examples of *coverage* problems.

Given a universe U of n objects and m subsets of U, choose k subsets that maximize:

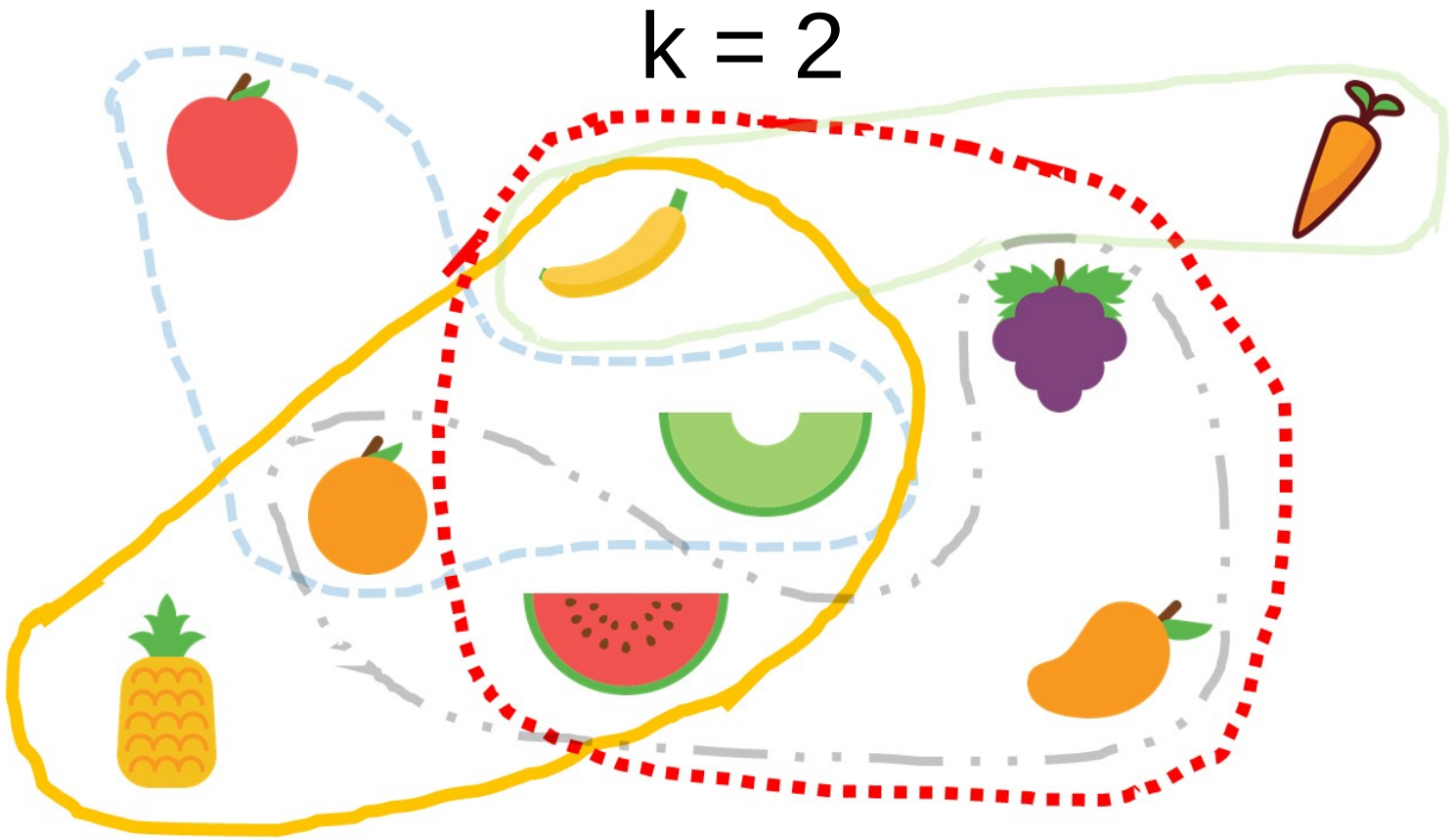


Maximum Coverage

These are examples of *coverage* problems.

Given a universe U of n objects and m subsets of U, choose k subsets that maximize:

- # of objects covered by at least 1 subset (Max-k Coverage)

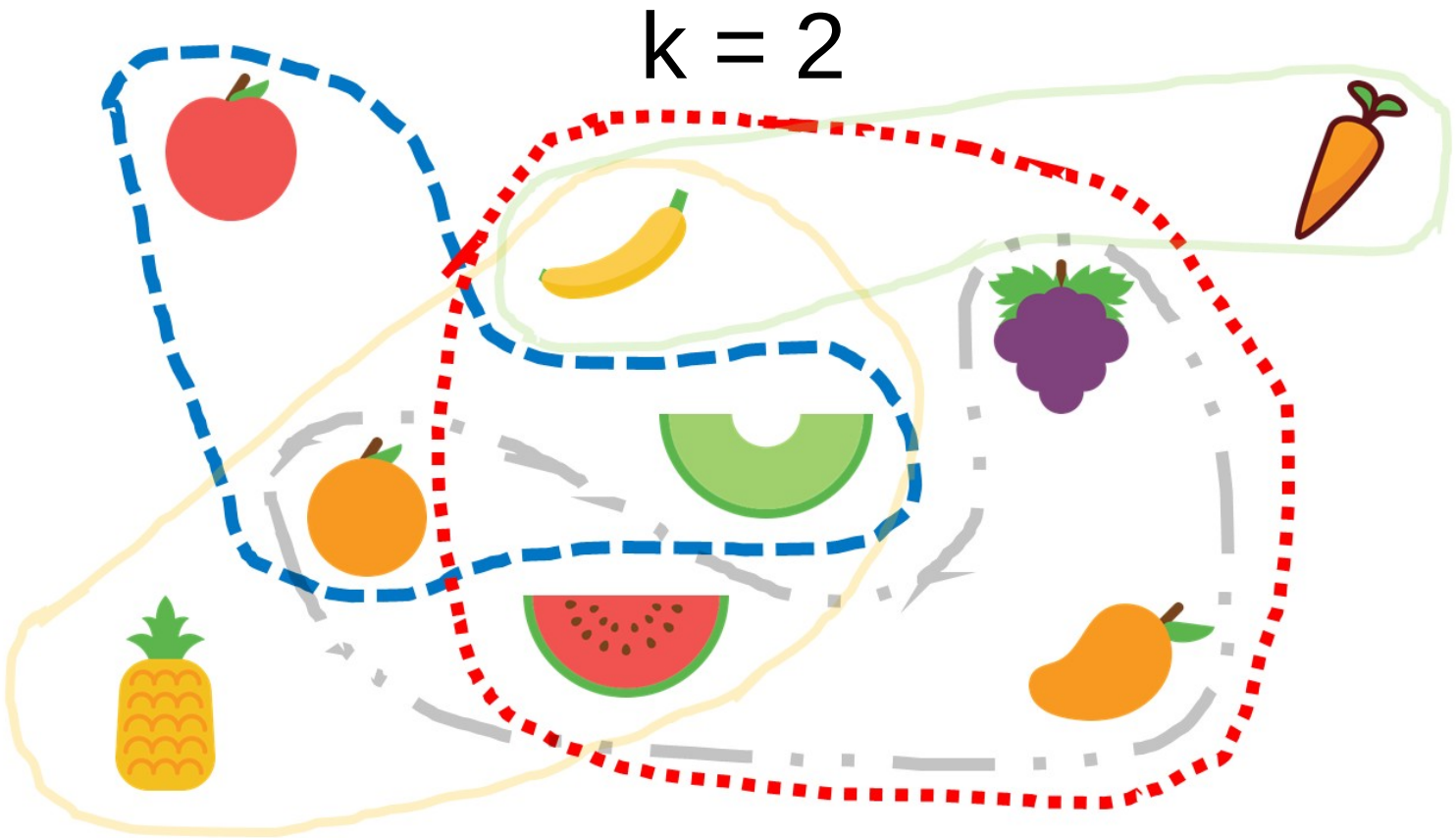


Maximum Coverage

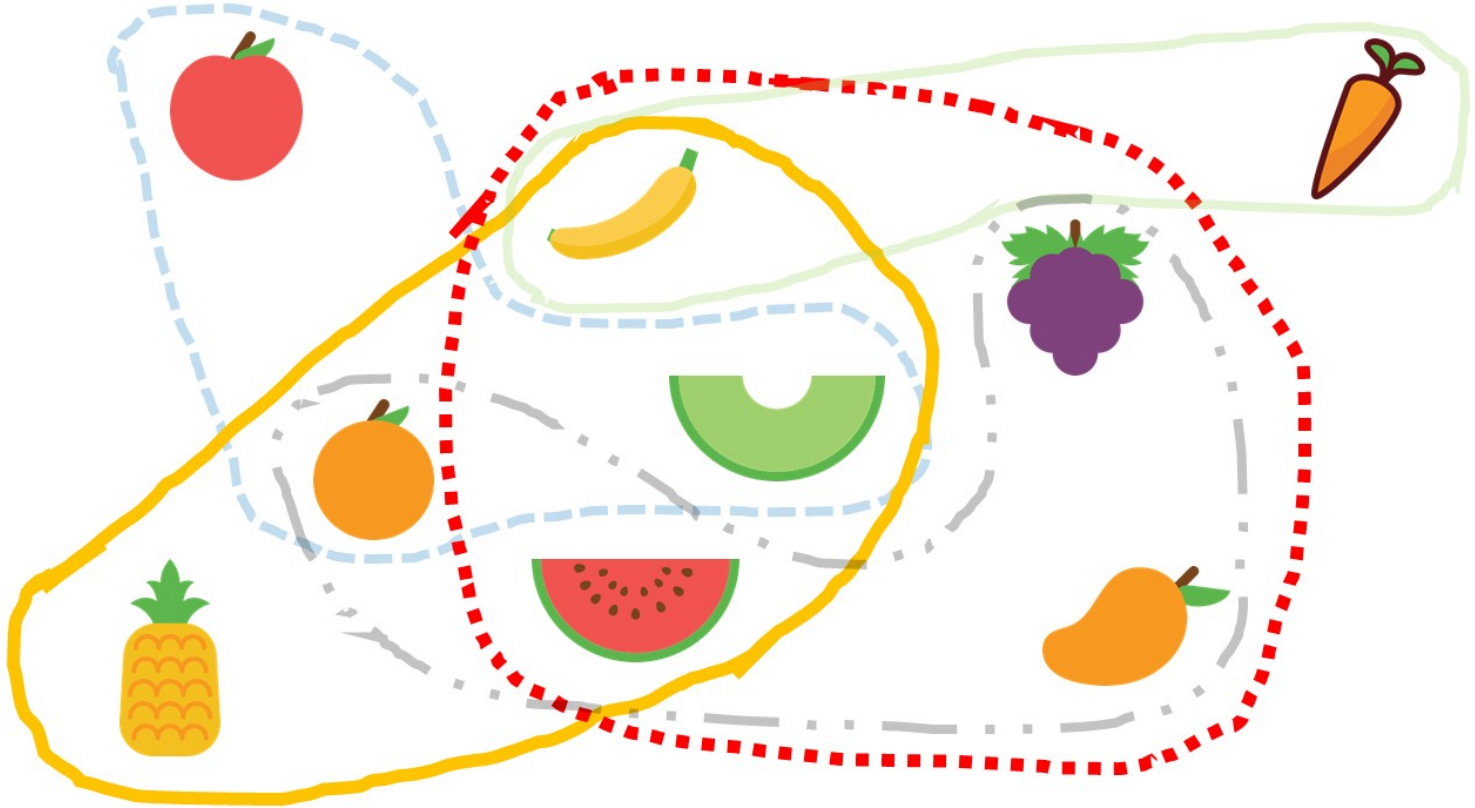
These are examples of *coverage* problems.

Given a universe U of n objects and m subsets of U, choose k subsets that maximize:

- # of objects covered by at least 1 subset (Max-k Coverage)
- # of objects covered by exactly 1 subset (Unique-k Coverage)

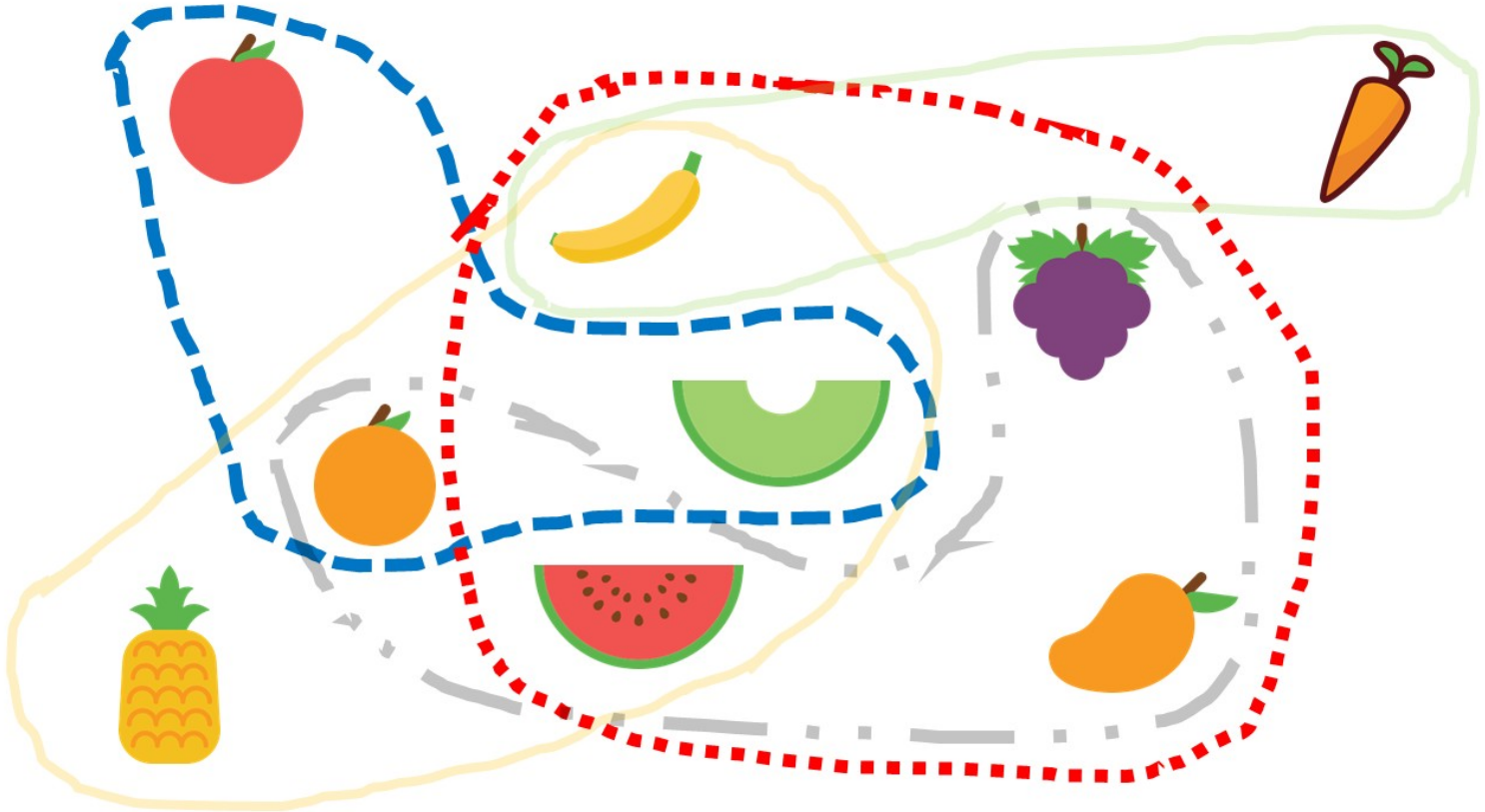


Max-k Cover



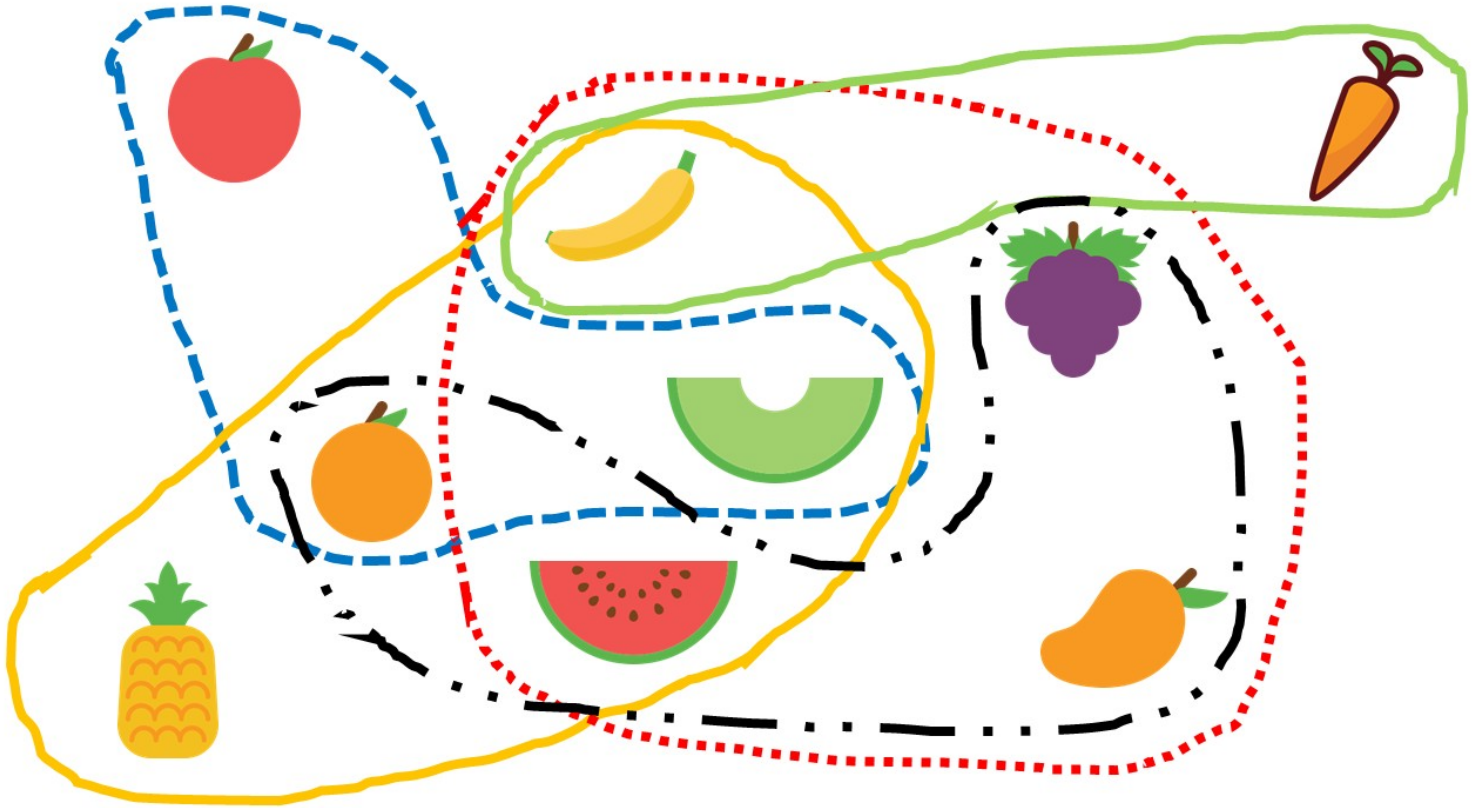
- NP-Hard
- Greedy (e/(e-1)) - approximation is the best possible

Unique-k Cover



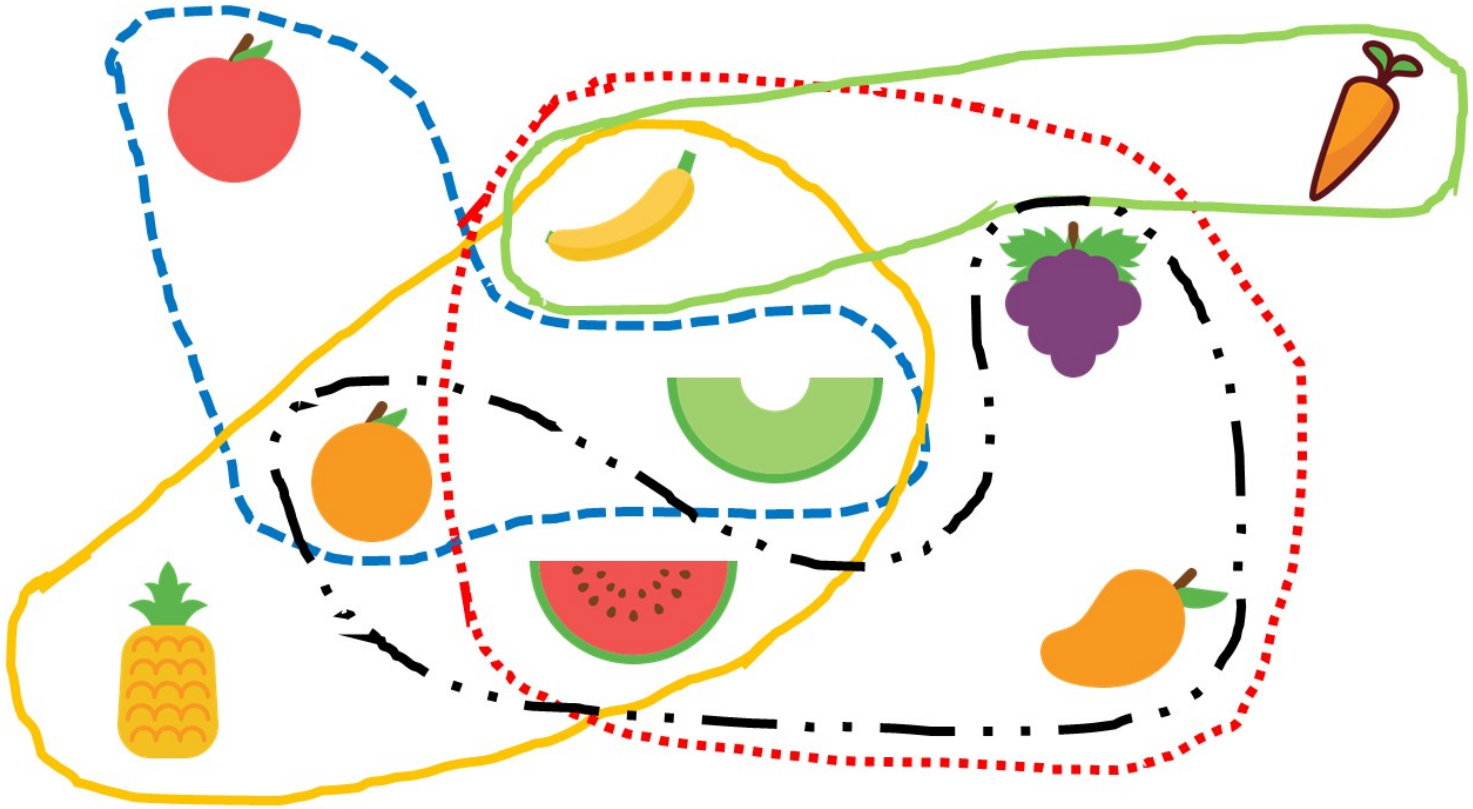
- NP-Hard
- Probably hard to O(polylog(n))-approximate

Another Perspective



These are also hypergraphs!

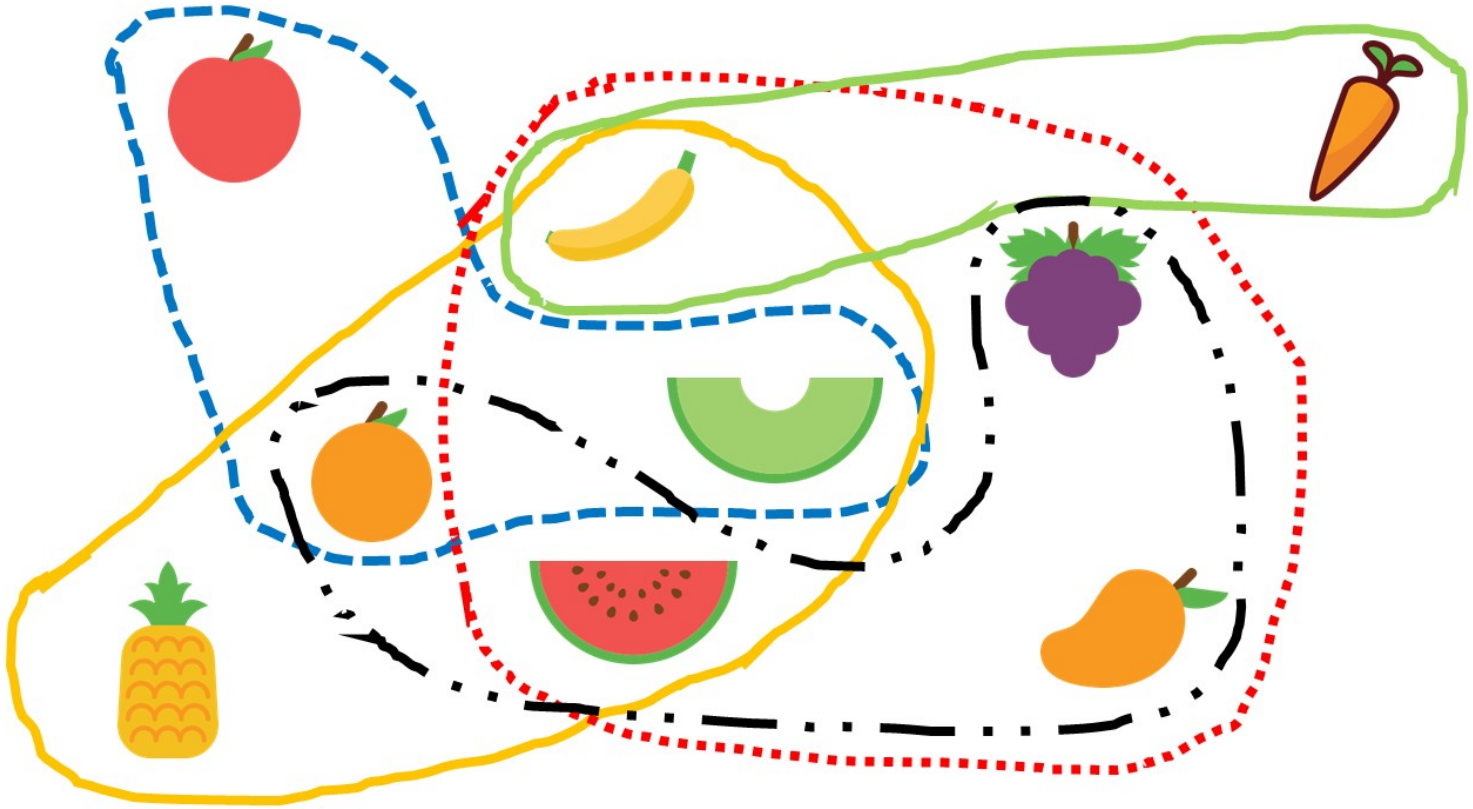
Another Perspective



These are also hypergraphs!

Like graphs, but edges (colored lines) can have more than 2 endpoints.

Another Perspective



These are also hypergraphs!

Like graphs, but edges (colored lines) can have more than 2 endpoints.

I'll use "node"/ "object" and "set"/"edge" interchangeably.

Our version is harder!

We assume there are more subsets than we can fit into our computer's memory.

Our version is harder!

We assume there are more subsets than we can fit into our computer's memory.

Our version is harder!

We assume there are more subsets than we can fit into our computer's memory.

We are told about these subsets one at a time, and we have o(m) space to work with.

Our version is harder!

We assume there are more subsets than we can fit into our computer's memory.

We are told about these subsets one at a time, and we have o(m) space to work with.

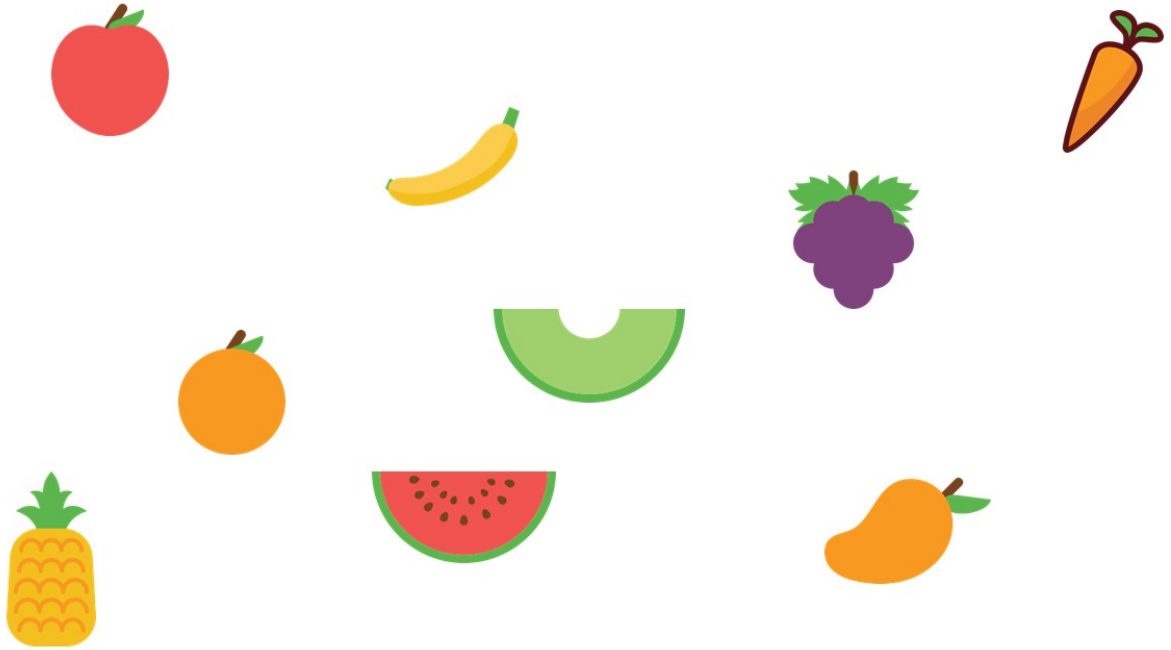
We can't remember every set we're shown.

Our version is harder!

We assume there are more subsets than we can fit into our computer's memory.

We are told about these subsets one at a time, and we have o(m) space to work with.

We can't remember every set we're shown.

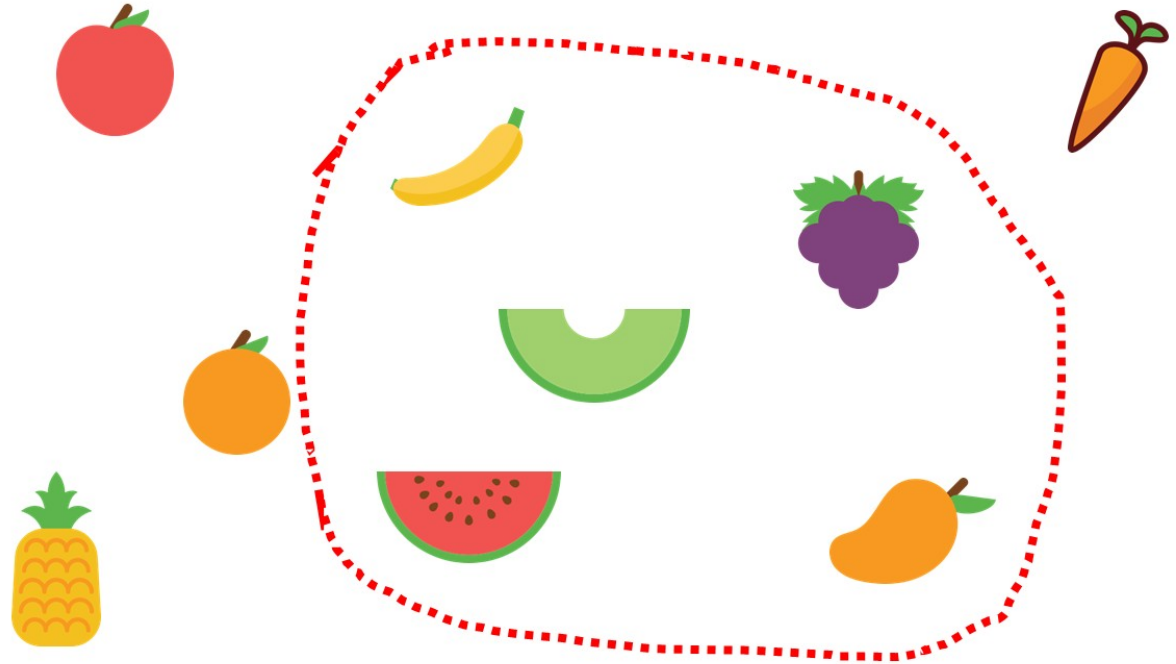


Our version is harder!

We assume there are more subsets than we can fit into our computer's memory.

We are told about these subsets one at a time, and we have o(m) space to work with.

We can't remember every set we're shown.

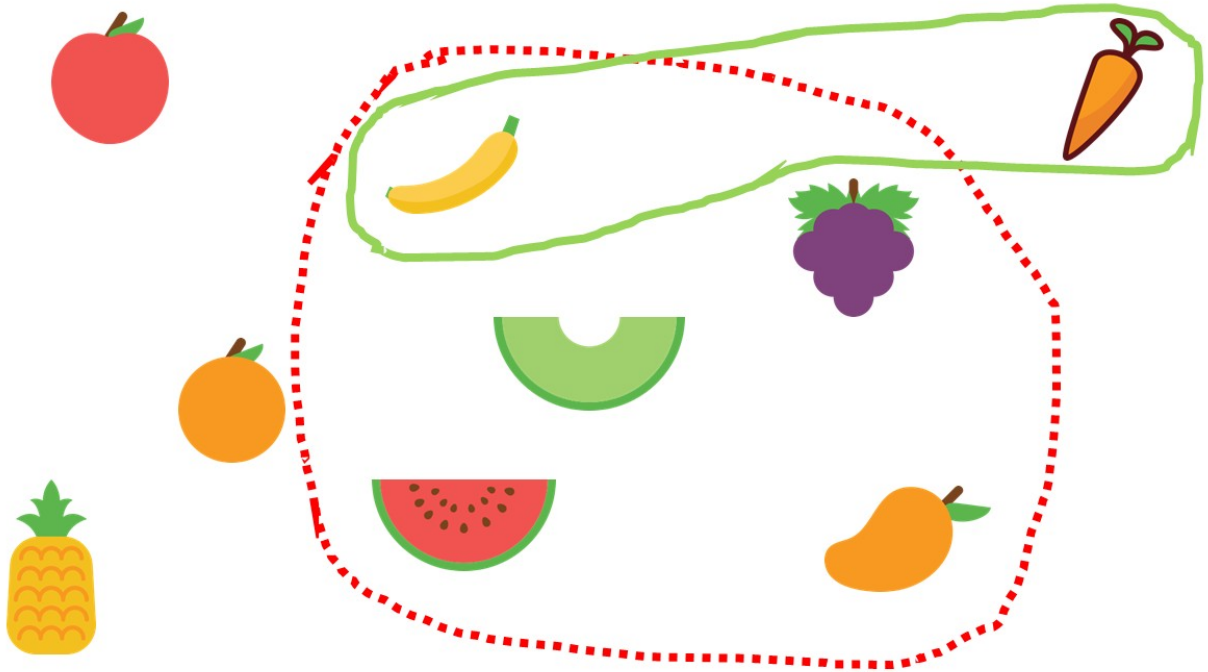


Our version is harder!

We assume there are more subsets than we can fit into our computer's memory.

We are told about these subsets one at a time, and we have o(m) space to work with.

We can't remember every set we're shown.



Our version is harder!

We assume there are more subsets than we can fit into our computer's memory.

We are told about these subsets one at a time, and we have o(m) space to work with.

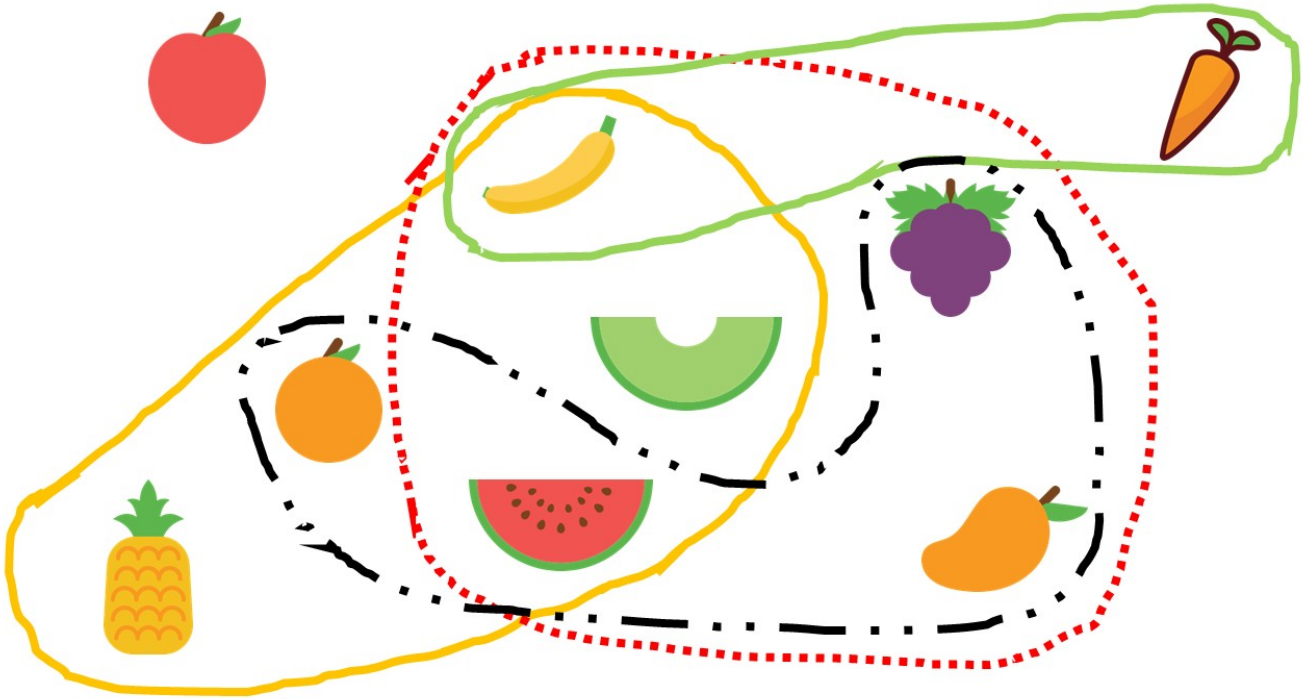
We can't remember every set we're shown.

Our version is harder!

We assume there are more subsets than we can fit into our computer's memory.

We are told about these subsets one at a time, and we have o(m) space to work with.

We can't remember every set we're shown.

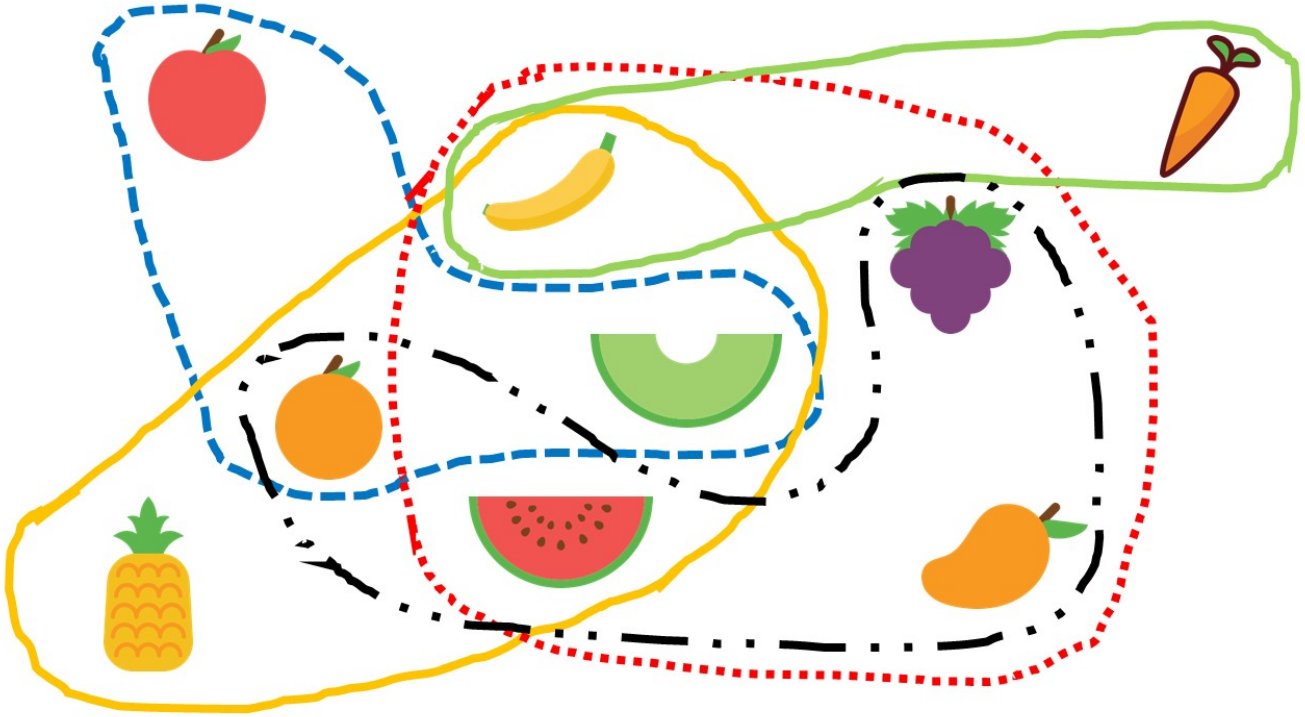


Our version is harder!

We assume there are more subsets than we can fit into our computer's memory.

We are told about these subsets one at a time, and we have o(m) space to work with.

We can't remember every set we're shown.

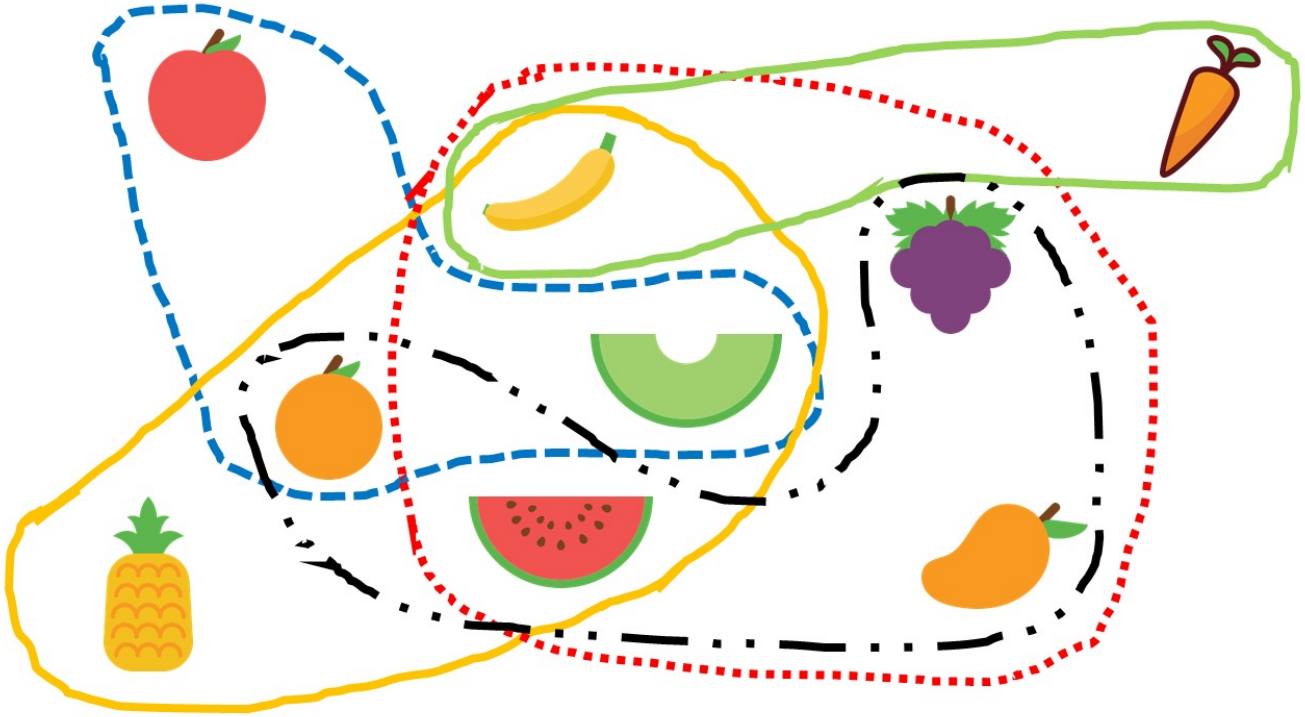


Our version is harder!

We assume there are more subsets than we can fit into our computer's memory.

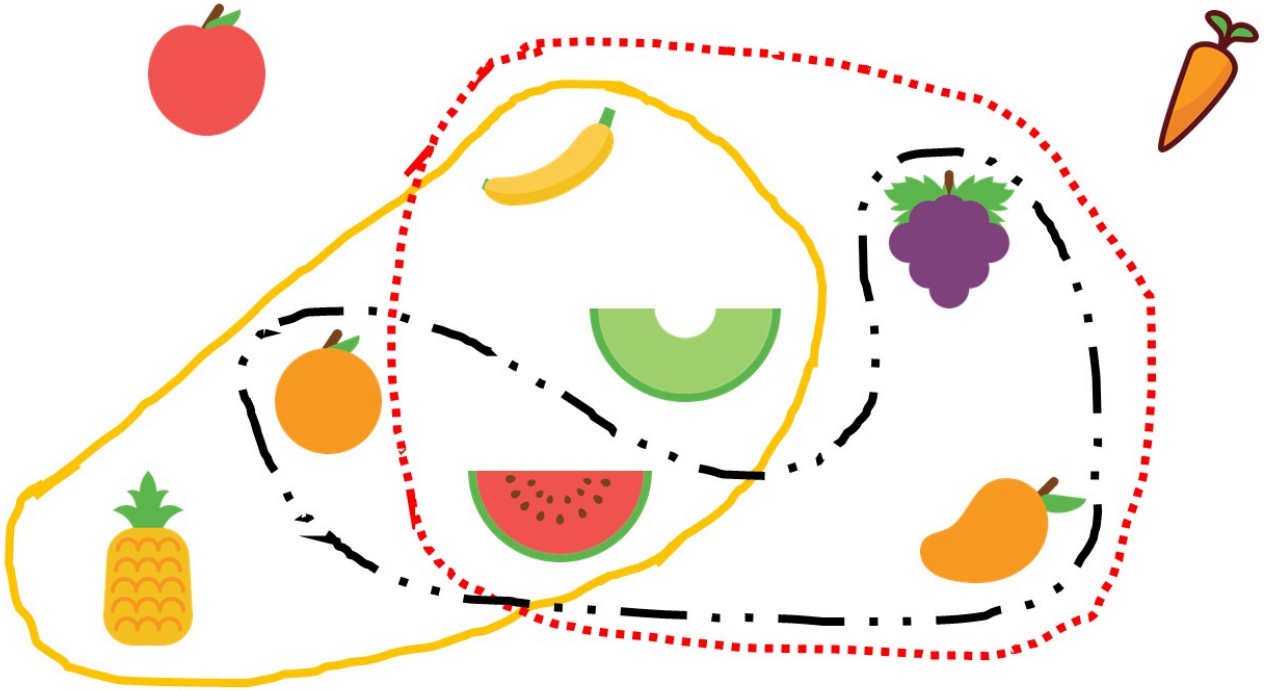
We are told about these subsets one at a time, and we have o(m) space to work with.

This is a generalization of the graph streaming setting.



Our version is harder!

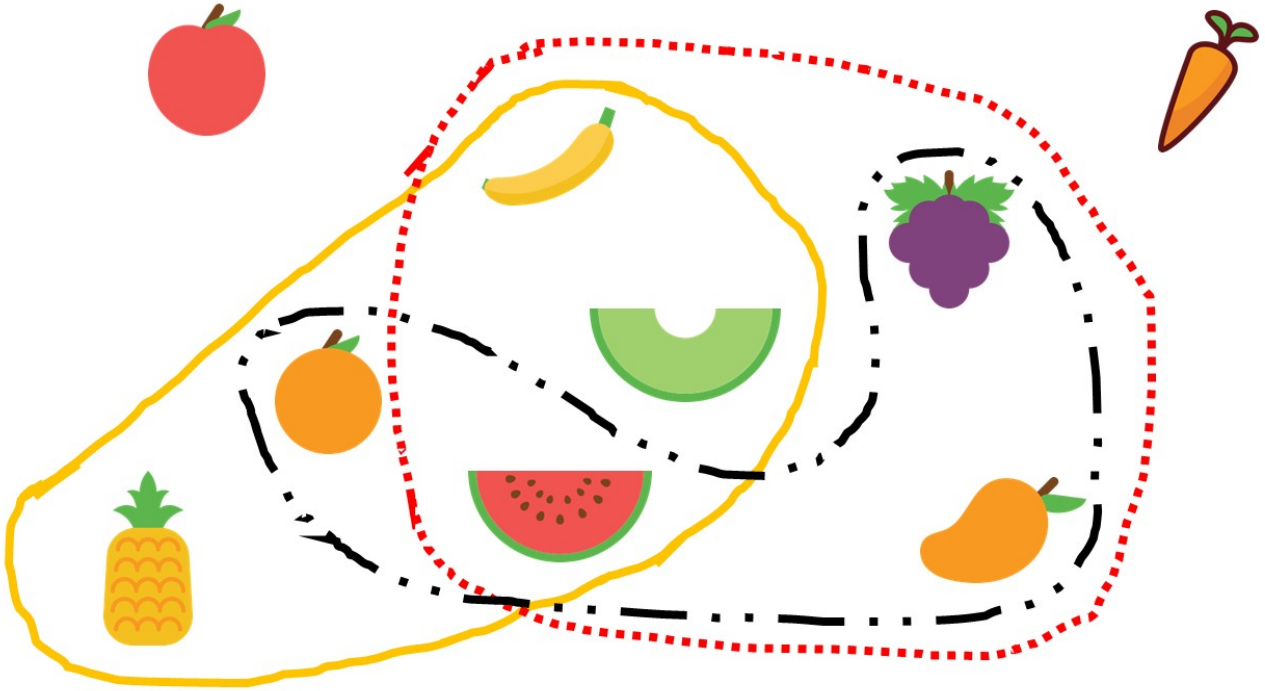
Our streaming algorithms produce *kernels*: a collection C of some (but not all) of the sets we were shown.



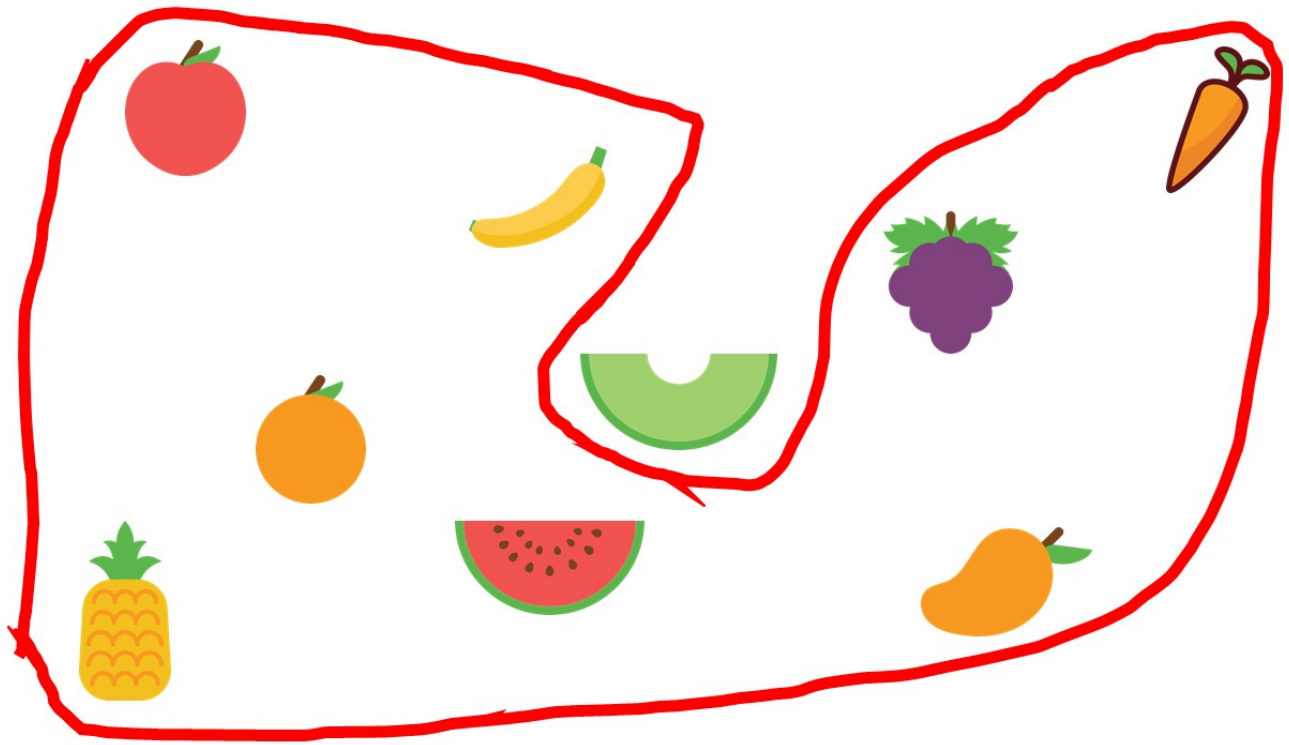
Our version is harder!

Our streaming algorithms produce *kernels*: a collection C of some (but not all) of the sets we were shown.

C has solutions to Max-k Cover and Unique-k Cover that are just as good (or almost as good) as the entire input.

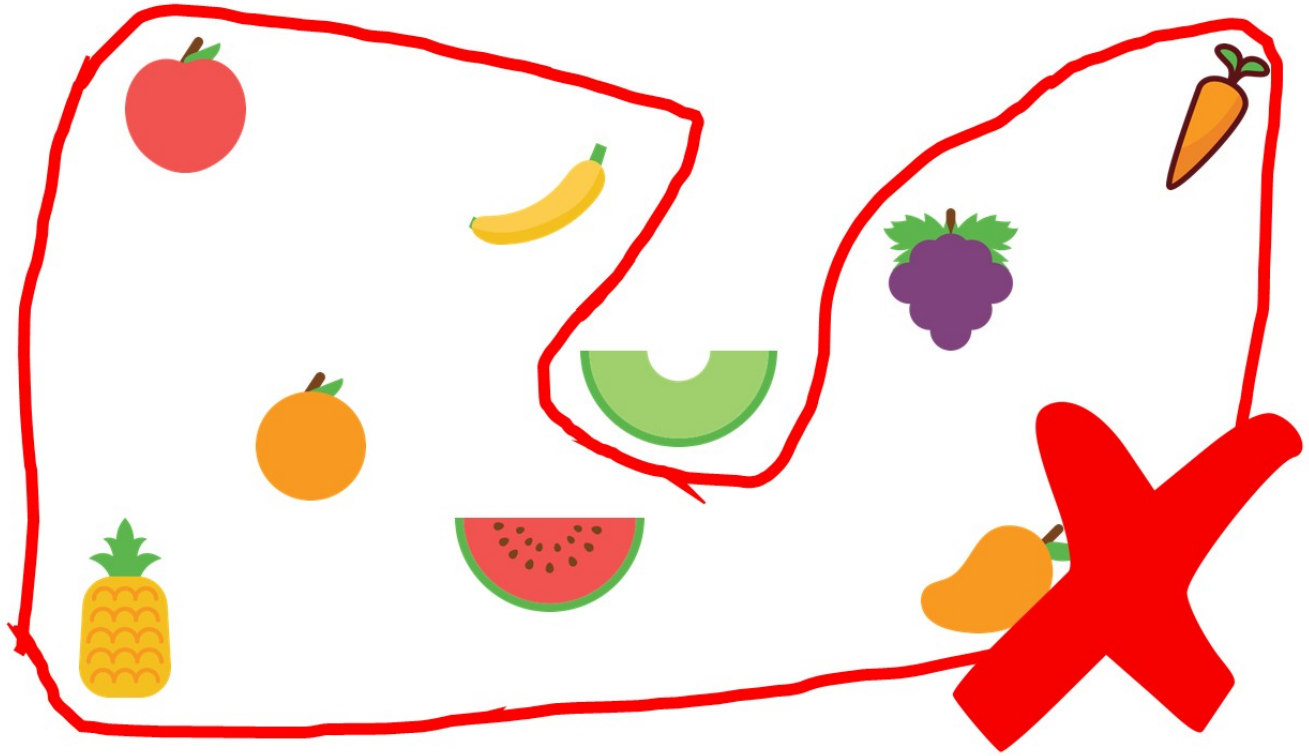


Some structural assumptions



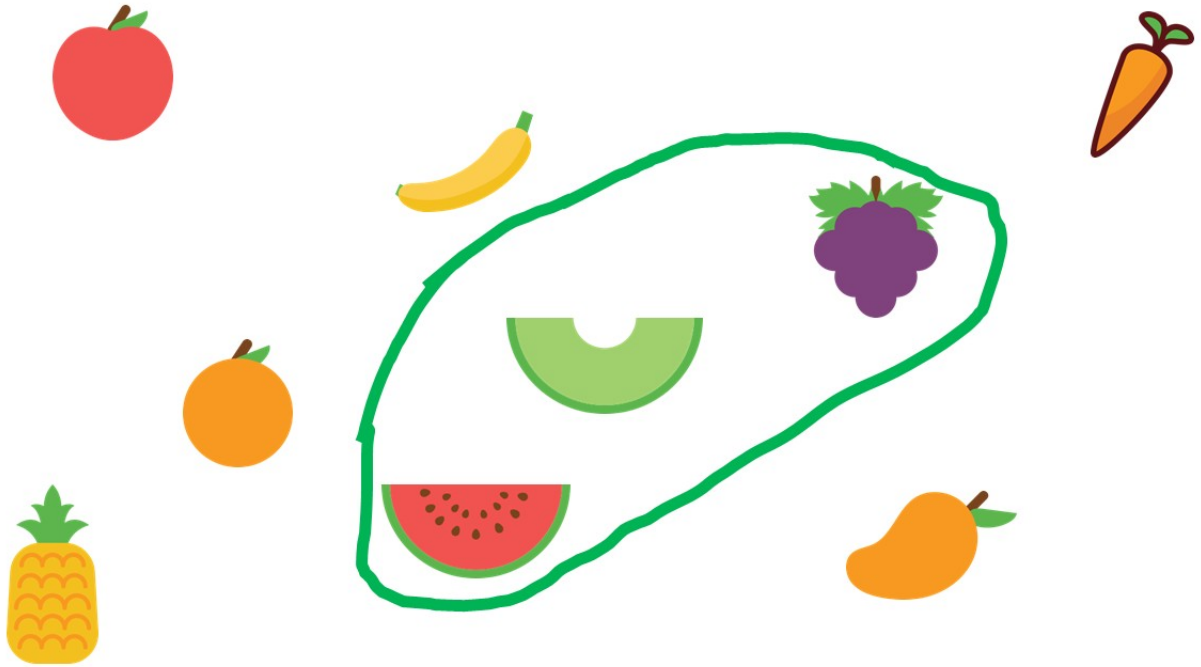
We sometimes assume bounded set size d

Some structural assumptions



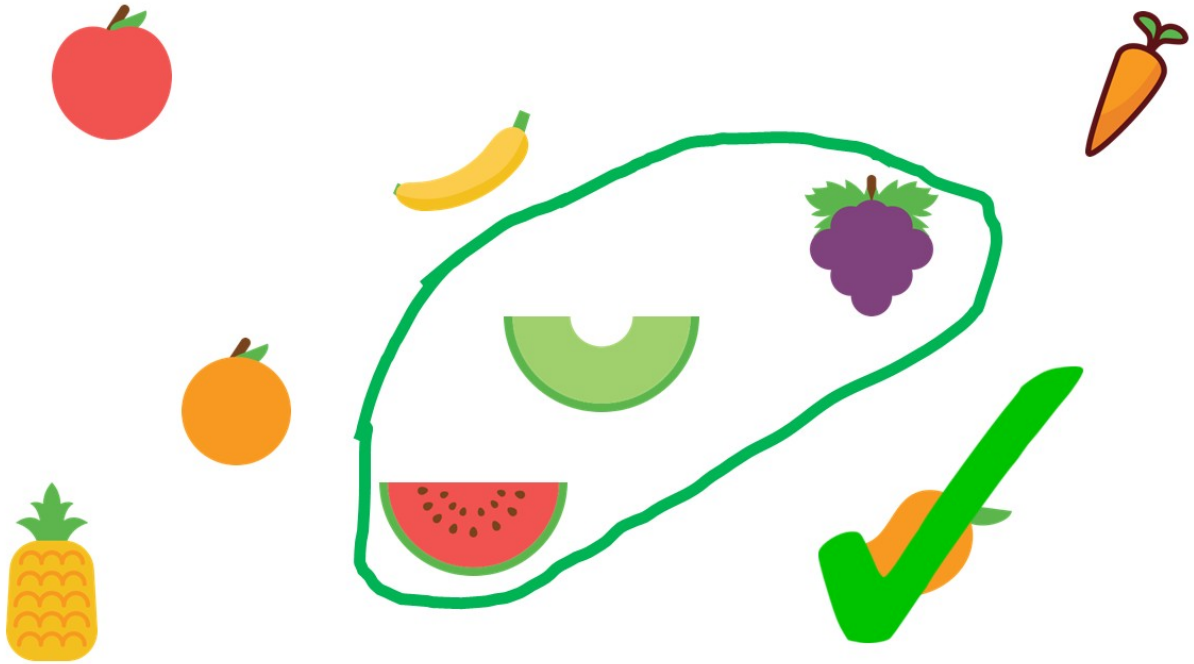
We sometimes assume bounded set size d

Some structural assumptions



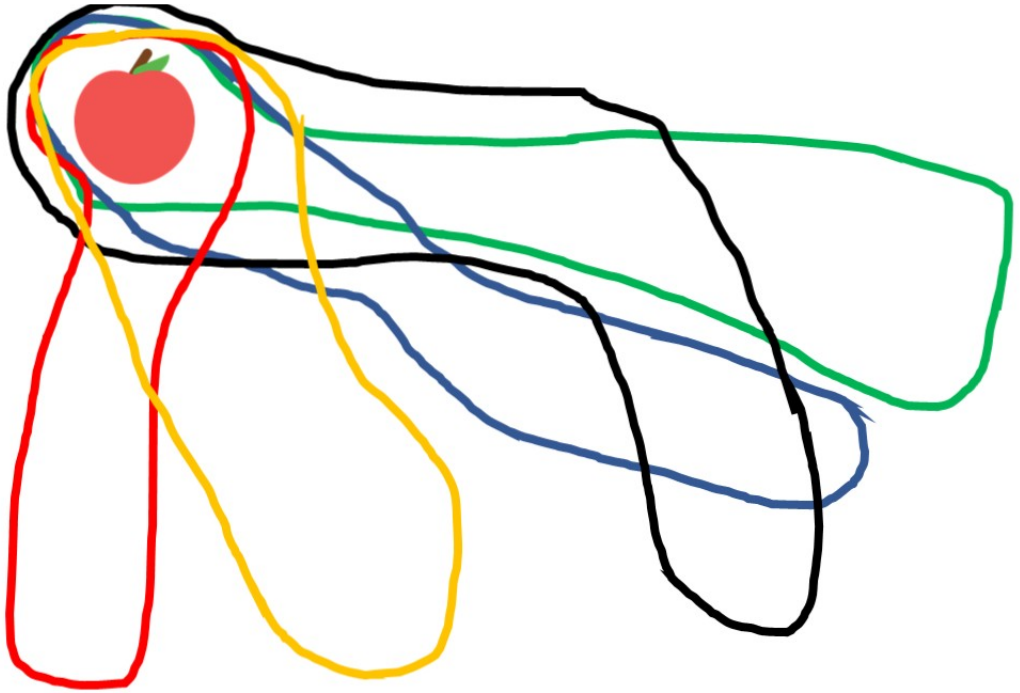
We sometimes assume bounded edge size d

Some structural assumptions



We sometimes assume bounded edge size d

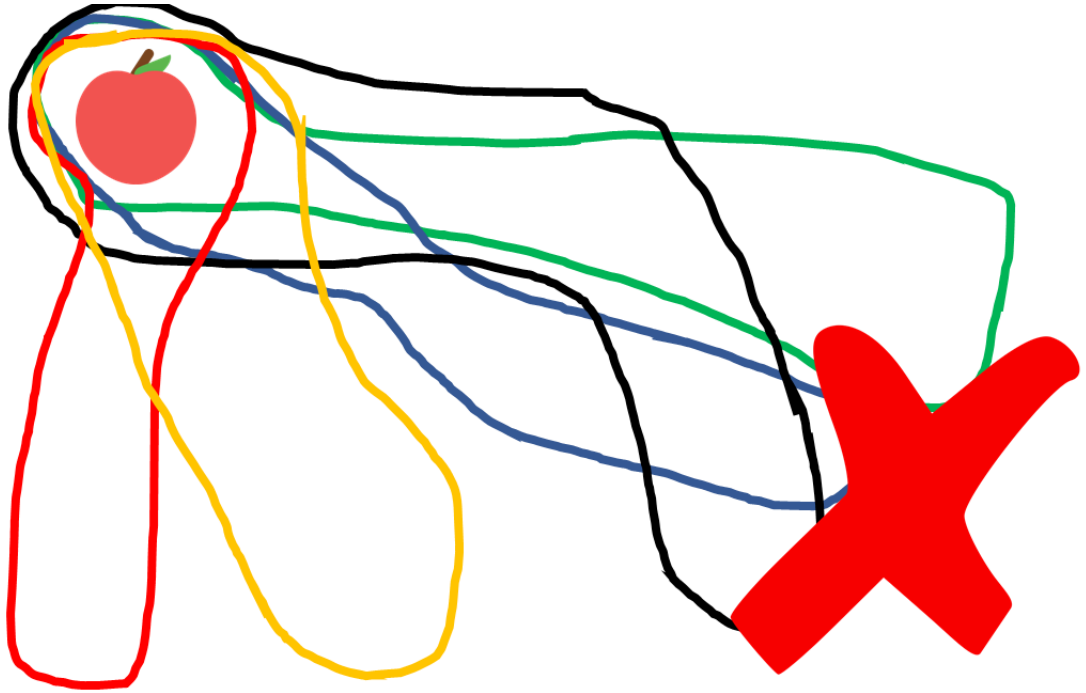
Some structural assumptions



We sometimes assume bounded edge size d

or that any node is in at most r edges

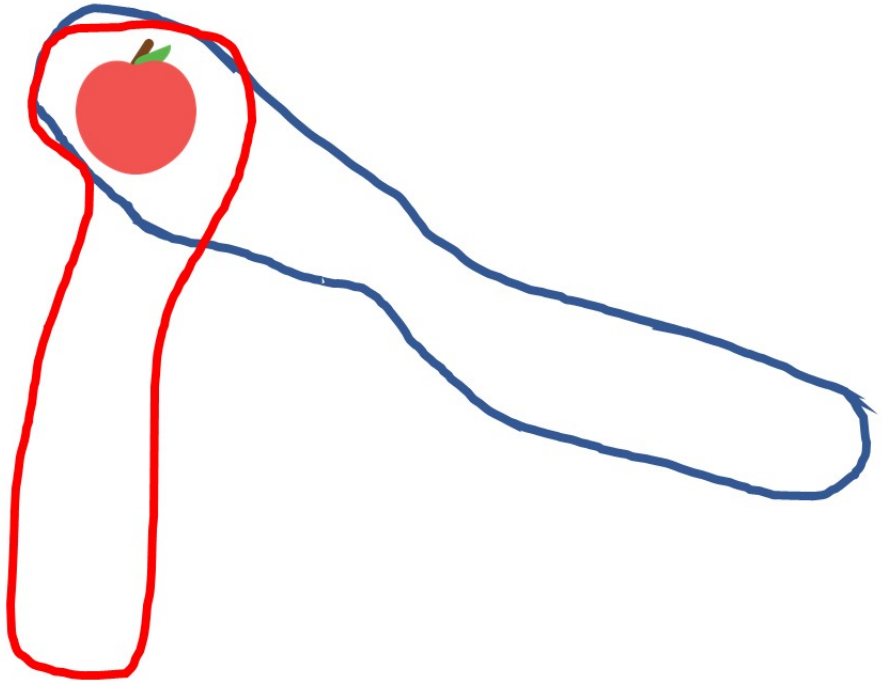
Some structural assumptions



We sometimes assume bounded edge size d

or that any node is in at most r edges

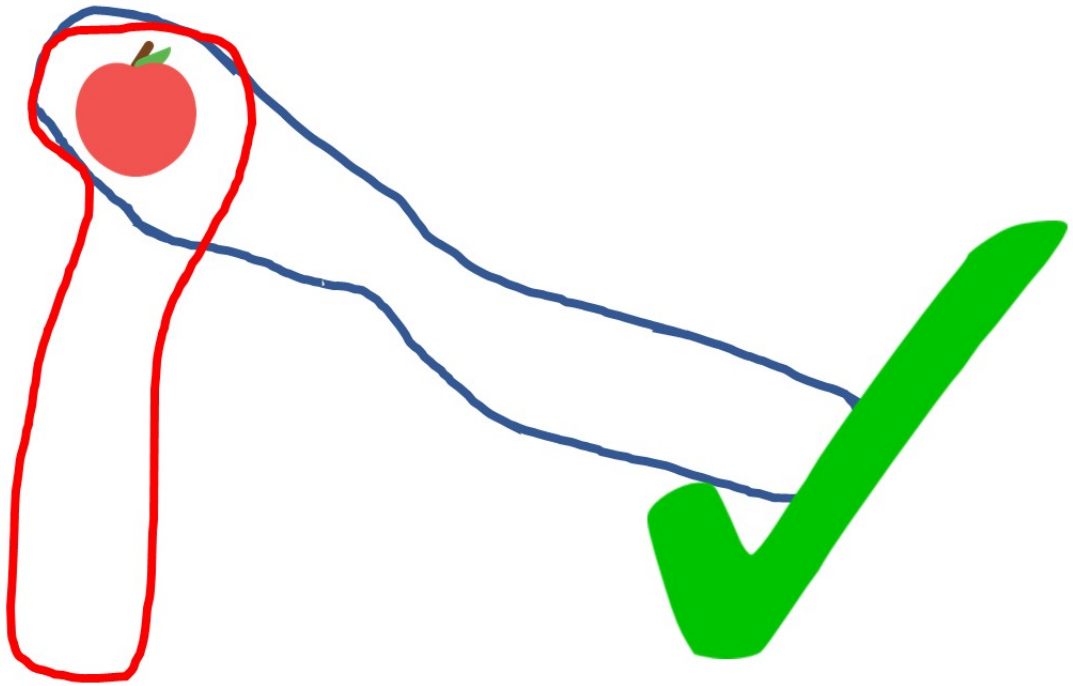
Some structural assumptions



We sometimes assume bounded edge size d

or that any node is in at most r edges

Some structural assumptions



We sometimes assume bounded edge size d

or that any node is in at most r edges

Our Results: Algorithms

When d is bounded we:

- Solve Max-k Cover and Unique-k Cover exactly using $\tilde{O}(d^{d+1}k^d)$ space (nearly optimal)

Our Results: Algorithms

When d is bounded we:

- Solve Max-k Cover and Unique-k Cover exactly using $\tilde{O}(d^{d+1}k^d)$ space (nearly optimal)

When r is bounded we:

- $(2+\epsilon)$-approx. Unique-k Cover using $\tilde{O}(\epsilon^{-3} k^2 r)$ space

Our Results: Algorithms

When d is bounded we:

- Solve Max-k Cover and Unique-k Cover exactly using $\tilde{O}(d^{d+1}k^d)$ space (nearly optimal)

When r is bounded we:

- $(2+\epsilon)$-approx. Unique-k Cover using $\tilde{O}(\epsilon^{-3} k^2 r)$ space
- $(1+\epsilon)$-approx. Unique-k Cover using $\tilde{O}(\epsilon^{-4} k^3 r)$ space

Our Results: Algorithms

When d is bounded we:

- Solve Max-k Cover and Unique-k Cover exactly using $\tilde{O}(d^{d+1}k^d)$ space (nearly optimal)

When r is bounded we:

- $(2+\epsilon)$-approx. Unique-k Cover using $\tilde{O}(\epsilon^{-3} k^2 r)$ space
- $(1+\epsilon)$-approx. Unique-k Cover using $\tilde{O}(\epsilon^{-4} k^3 r)$ space
- $(1+\epsilon)$-appx. Max-k Cover using $\tilde{O}(\epsilon^{-3} k^2 r)$ space

Our Results: Algorithms

When d is bounded we:

- Solve Max-k Cover and Unique-k Cover exactly using $\tilde{O}(d^{d+1}k^d)$ space (nearly optimal)

With no d or r assumptions we:

- O(log min(k,r)) approx. Unique-k Cover using $\tilde{O}(k^2)$ space

When r is bounded we:

- $(2+\epsilon)$-approx. Unique-k Cover using $\tilde{O}(\epsilon^{-3} k^2 r)$ space
- $(1+\epsilon)$-approx. Unique-k Cover using $\tilde{O}(\epsilon^{-4} k^3 r)$ space
- $(1+\epsilon)$-appx. Max-k Cover using $\tilde{O}(\epsilon^{-3} k^2 r)$ space

Our Results: Algorithms

When d is bounded we:

- Solve Max-k Cover and Unique-k Cover exactly using $\tilde{O}(d^{d+1}k^d)$ space (nearly optimal)

With no d or r assumptions we:

- O(log min(k,r)) approx. Unique-k Cover using $\tilde{O}(k^2)$ space

When r is bounded we:

- (2+ϵ)-approx. Unique-k Cover using $\tilde{O}(\epsilon^{-3} k^2 r)$ space
- (1+ϵ)-approx. Unique-k Cover using $\tilde{O}(\epsilon^{-4} k^3 r)$ space
- (1+ϵ)-appx. Max-k Cover using $\tilde{O}(\epsilon^{-3} k^2 r)$ space

Our Results: Lower Bounds

When d is bounded:

- Solving either problem exactly requires $\Omega(k^d)$ space

Our Results: Lower Bounds

When d is bounded:

- Solving either problem exactly requires $\Omega(k^d)$ space

With no d or r assumptions:

- $(1+\epsilon)$-approx. either problem requires $\Omega(\epsilon^{-2} m)$ space, even with constant passes over the stream

Our Results: Lower Bounds

When d is bounded:

- Solving either problem exactly requires $\Omega(k^d)$ space

With no d or r assumptions:

- $(1+\epsilon)$-approx. either problem requires $\Omega(\epsilon^{-2} m)$ space, even with constant passes over the stream
- Any approx. better than $e^{1-1/k}$ requires $\Omega(k^{-2} m)$ space, even with constant passes over the stream

Our Results: Algorithms

When d is bounded we:

- Solve Max-k Cover and Unique-k Cover exactly using $\tilde{O}(d^{d+1}k^d)$ space (nearly optimal)

With no d or r assumptions we:

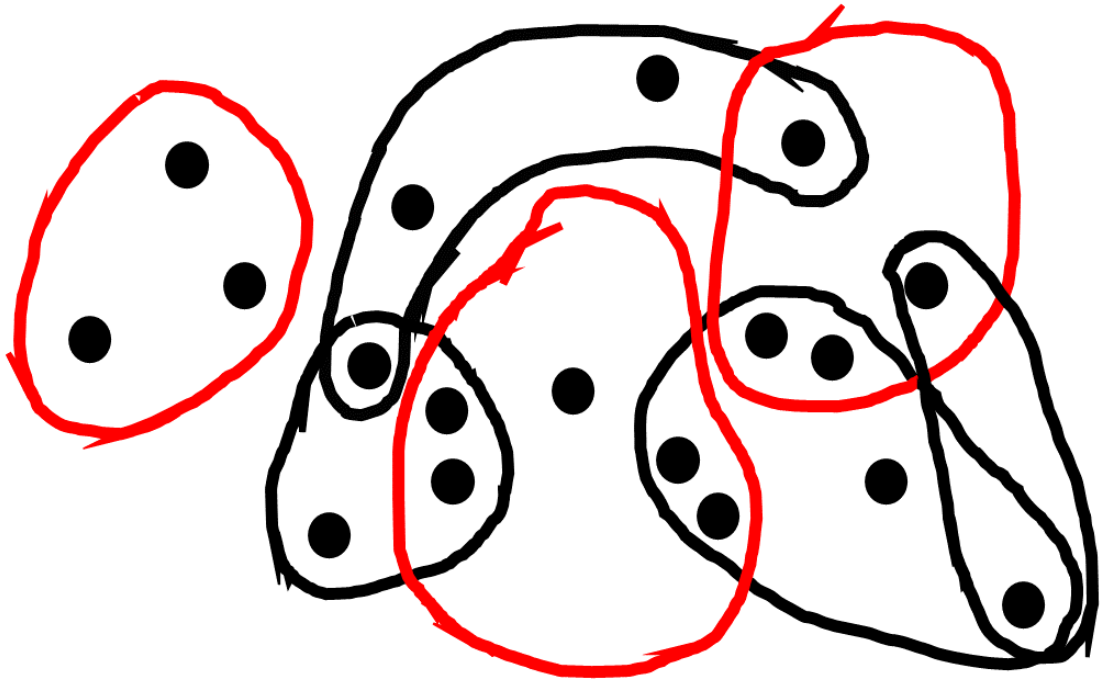
- O(log min(k,r)) approx. Unique-k Cover using $\tilde{O}(k^2)$ space

When r is bounded we:

- (2+ϵ)-approx. Unique-k Cover using $\tilde{O}(\epsilon^{-3} k^2 r)$ space
- (1+ϵ)-approx. Unique-k Cover using $\tilde{O}(\epsilon^{-4} k^3 r)$ space
- (1+ϵ)-appx. Max-k Cover using $\tilde{O}(\epsilon^{-3} k^2 r)$ space

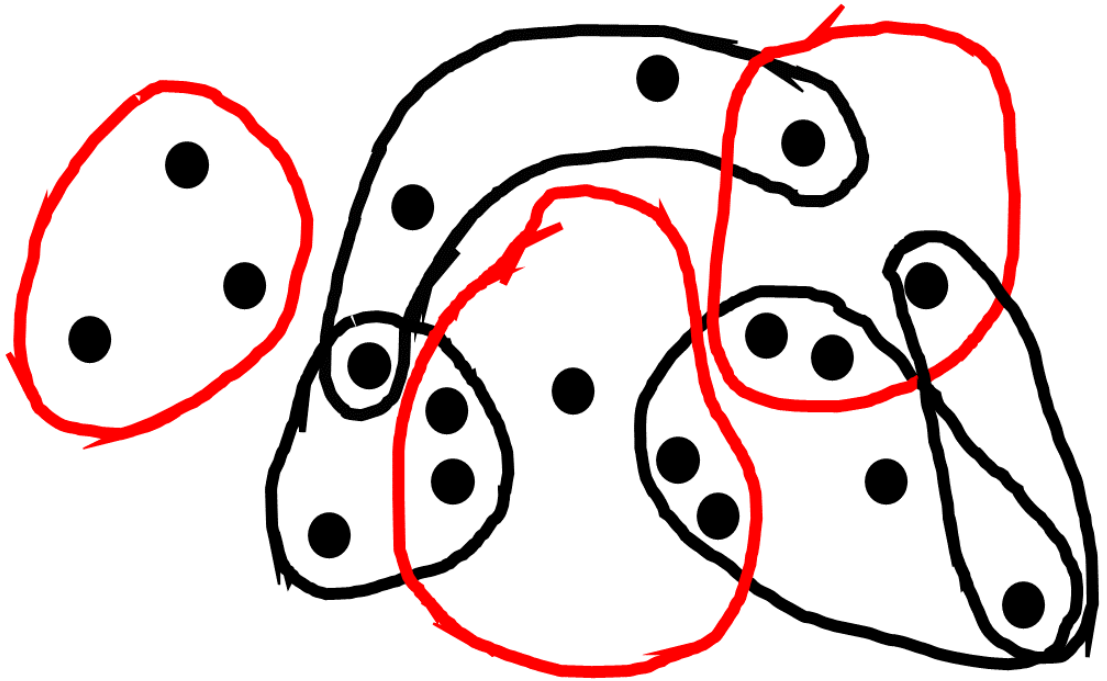
Solving Max-k Cover Exactly

Fix some maximal matching M.



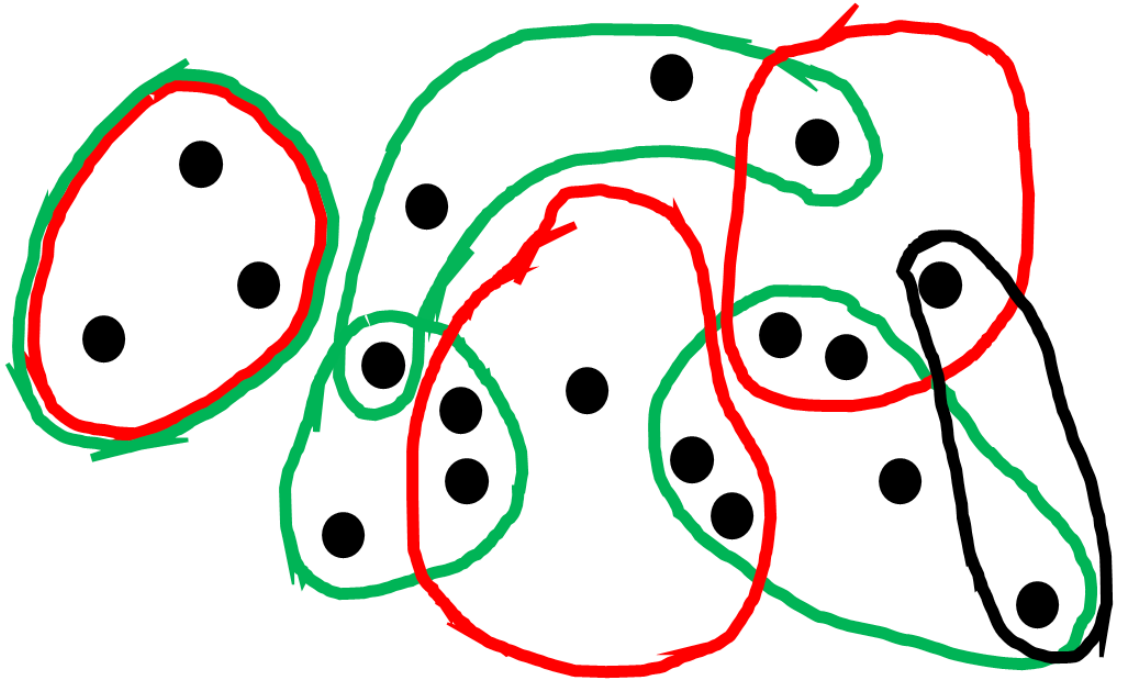
Solving Max-k Cover Exactly

Fix some maximal matching M.
Every set in an optimal solution OPT for Max-k Cover intersects with a set in M.



Solving Max-k Cover Exactly

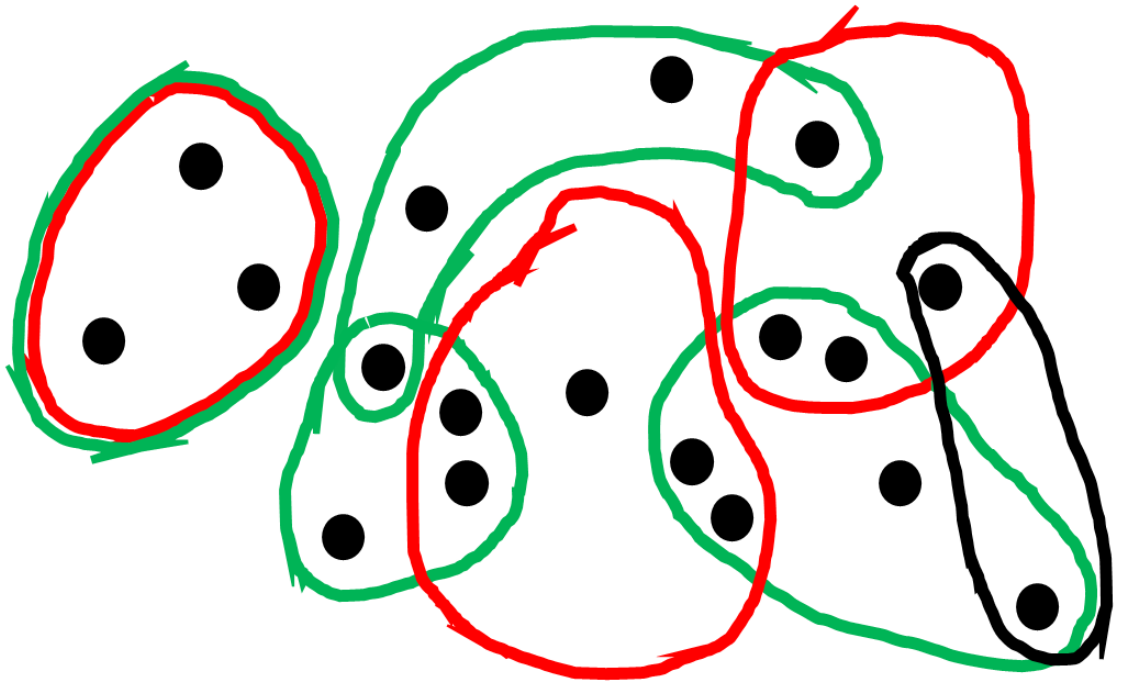
Fix some maximal matching M.
Every set in an optimal solution OPT for Max-k Cover intersects with a set in M.



Solving Max-k Cover Exactly

Fix some maximal matching M. Every set in an optimal solution OPT for Max-k Cover intersects with a set in M.

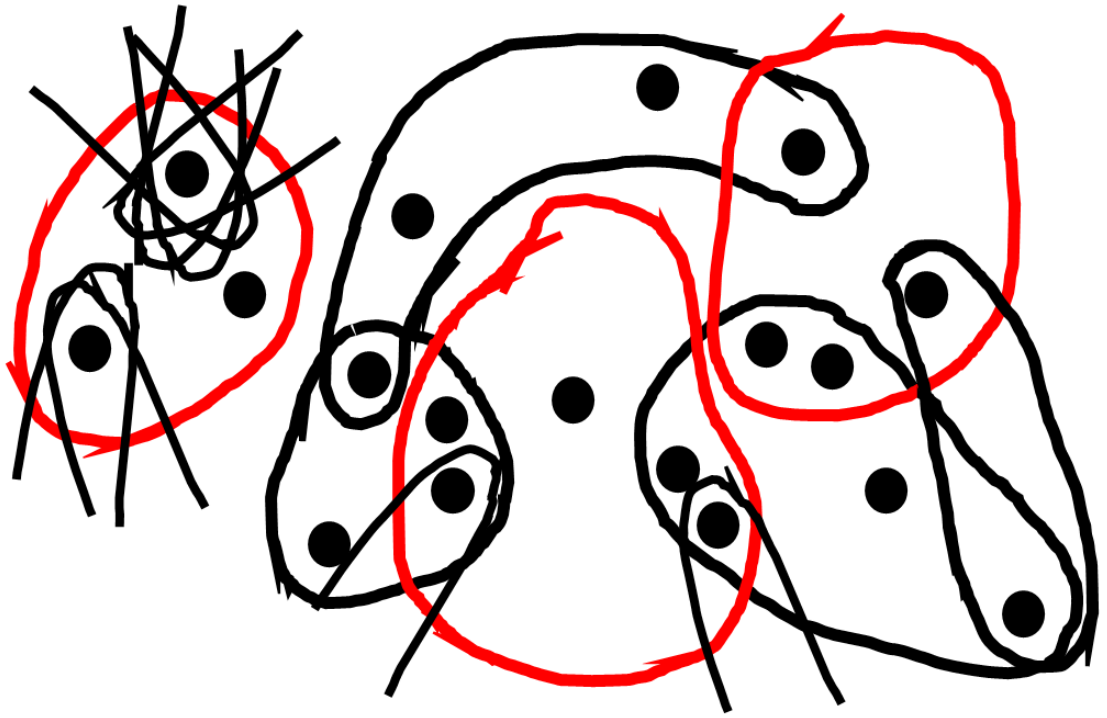
If we build M from the stream, and keep everything which overlaps M, we preserve OPT.



Solving Max-k Cover Exactly

Fix some maximal matching M. Every set in an optimal solution OPT for Max-k Cover intersects with a set in M.

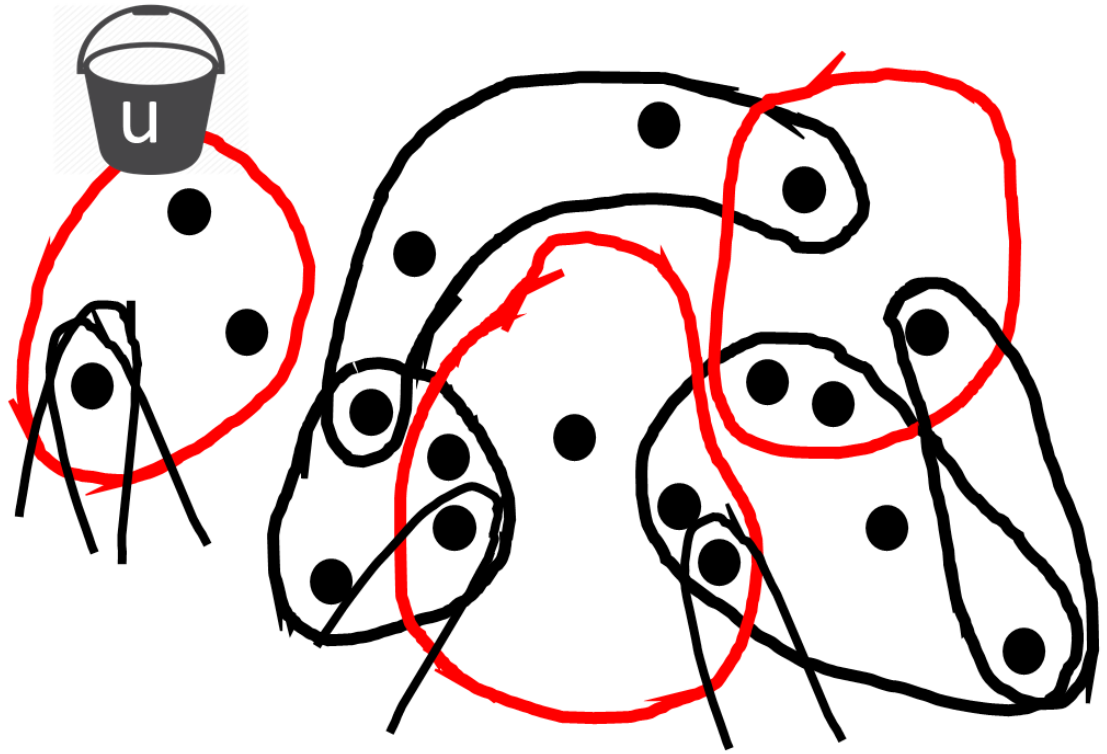
But since we assume no bound on r, Ω(m) sets can intersect with M. Too many to store!



Solving Max-k Cover Exactly

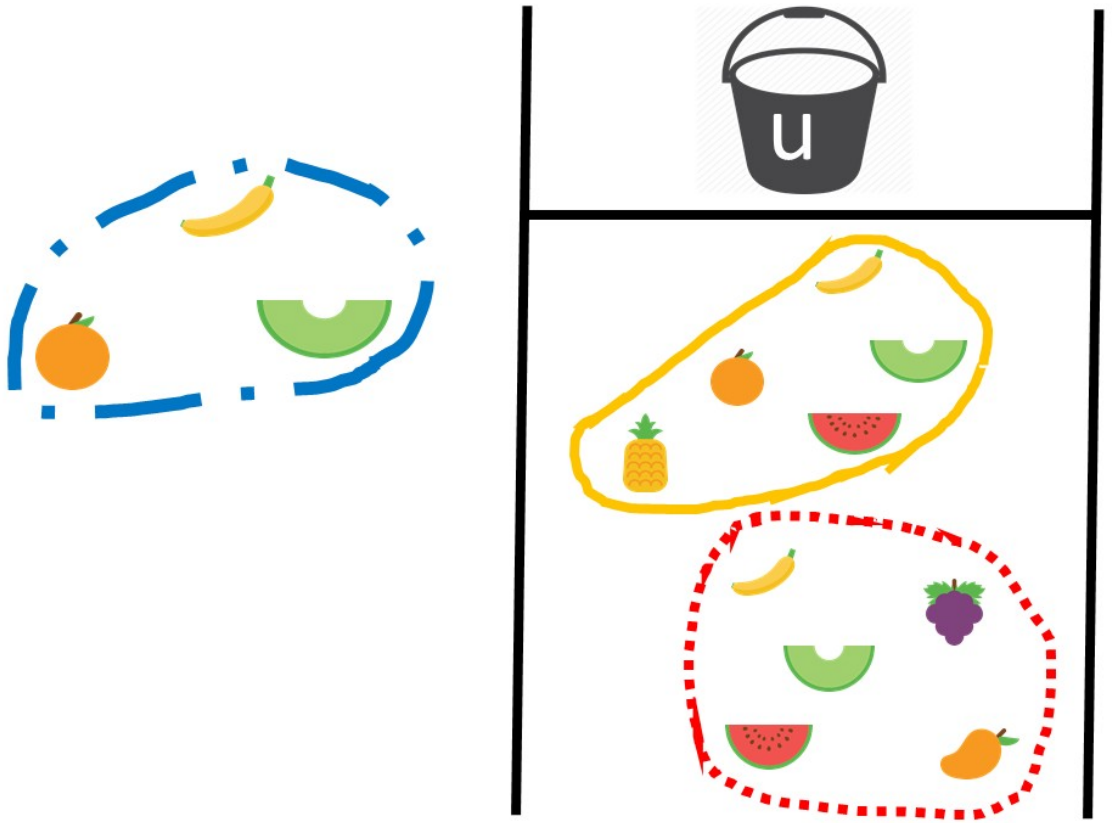
M + intersecting sets too big.

Solution: store M and a "bucket" b_u for each object u in M. When a set intersects with M at node u, we keep it unless it is too similar to the contents of b_u.



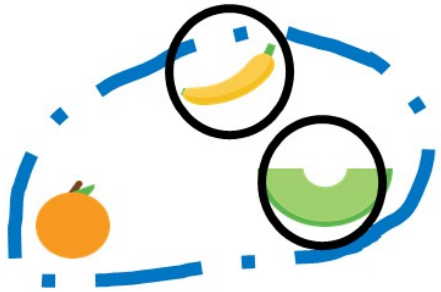
Solving Max-k Cover Exactly

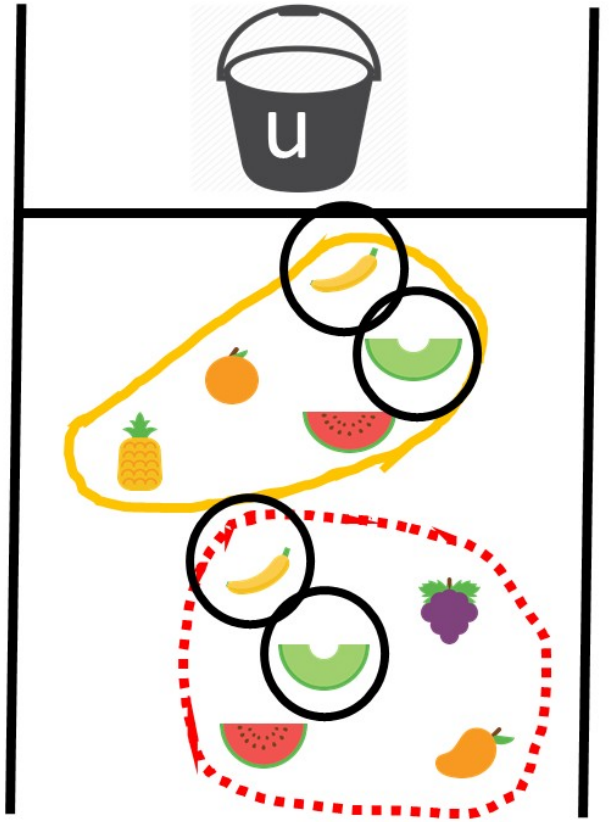
Solution: store M and a "bucket" b_u for each object u in M. When a set intersects with M at node u, we keep it unless it is too similar to the contents of b_u.



Solving Max-k Cover Exactly

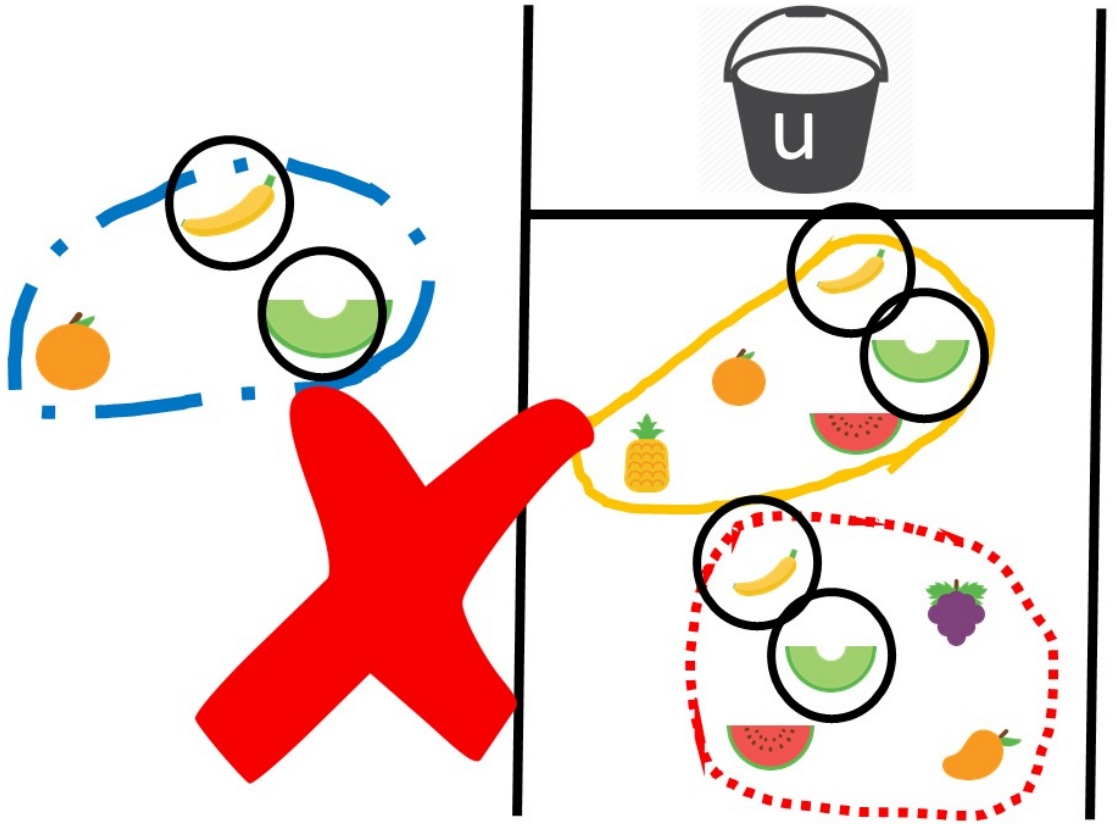
Solution: store M and a "bucket" b_u for each object u in M. When a set intersects with M at node u, we keep it unless it is too similar to the contents of b_u.





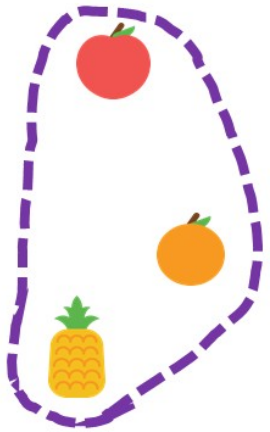
Solving Max-k Cover Exactly

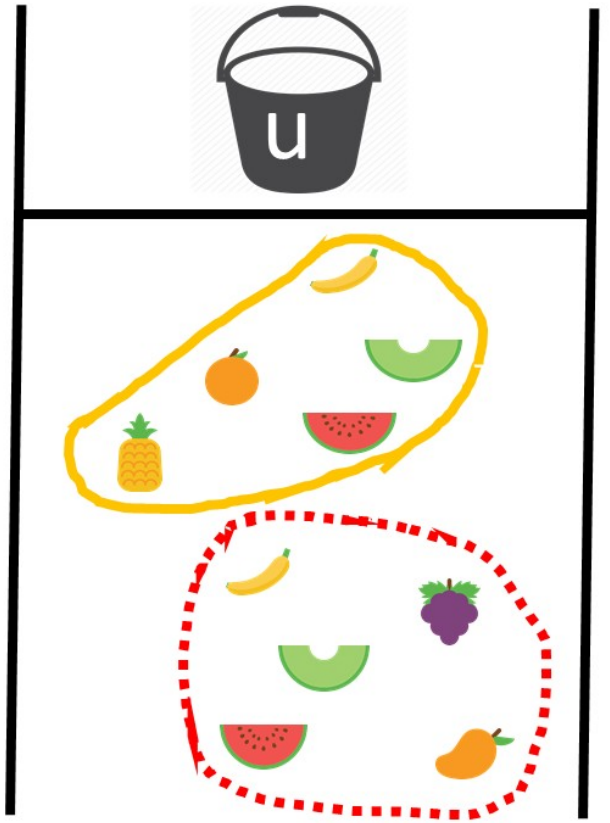
Solution: store M and a "bucket" b_u for each object u in M. When a set intersects with M at node u, we keep it unless it is too similar to the contents of b_u.



Solving Max-k Cover Exactly

Solution: store M and a "bucket" b_u for each object u in M. When a set intersects with M at node u, we keep it unless it is too similar to the contents of b_u.

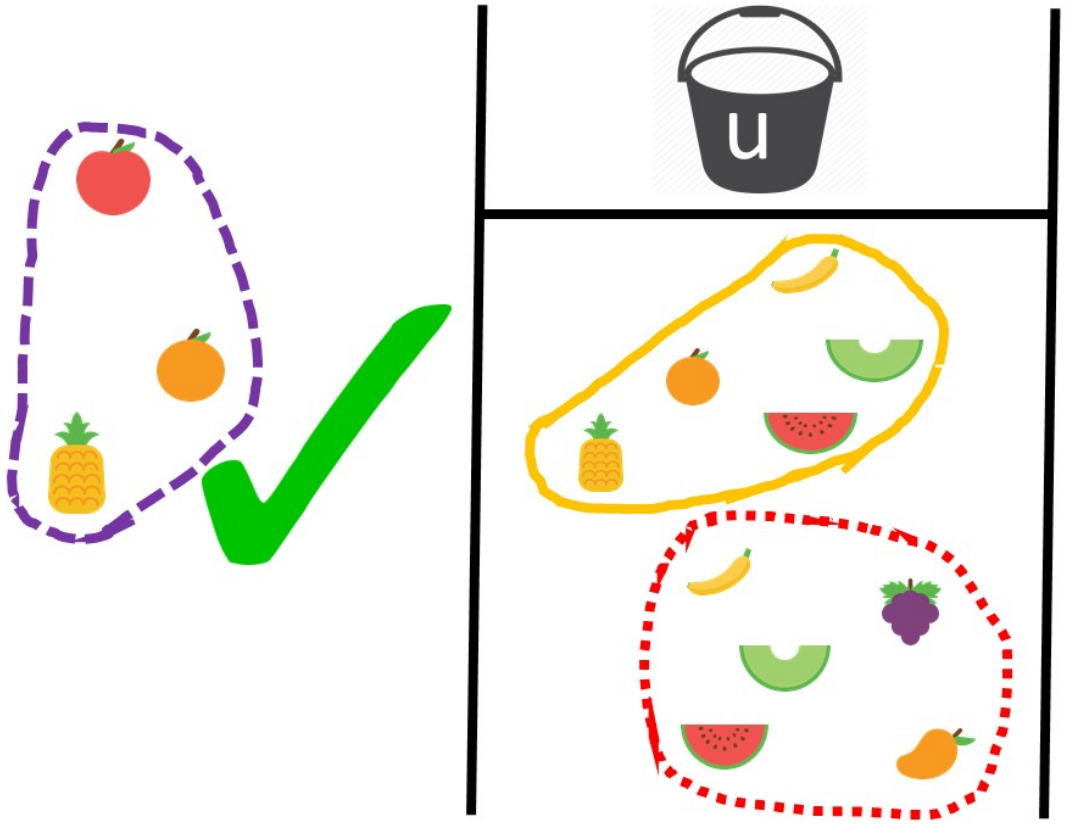




Solving Max-k Cover Exactly

Solution: store M and a "bucket" b_u for each object u in M. When a set intersects with M at node u, we keep it unless it is too similar to the contents of b_u.

Intuition: we only discard sets when there's a similar set in the bucket that can substitute for it.



Solving Max-k Cover Exactly

Set S arrives in stream.

Solving Max-k Cover Exactly

Set S arrives in stream.

- If S is disjoint from M, M += S.

Solving Max-k Cover Exactly

Set S arrives in stream.

- If S is disjoint from M, M += S.
- If it intersects with M at node u:

Solving Max-k Cover Exactly

Set S arrives in stream.

- If S is disjoint from M, M += S.
- If it intersects with M at node u:
- If there is $T \subset S \backslash \{u\}$ that already appears in $\ell_{|T|}$ sets in b_u, discard S. b_u has similar sets already.
  - Otherwise, add it.

Solving Max-k Cover Exactly

Set S arrives in stream.

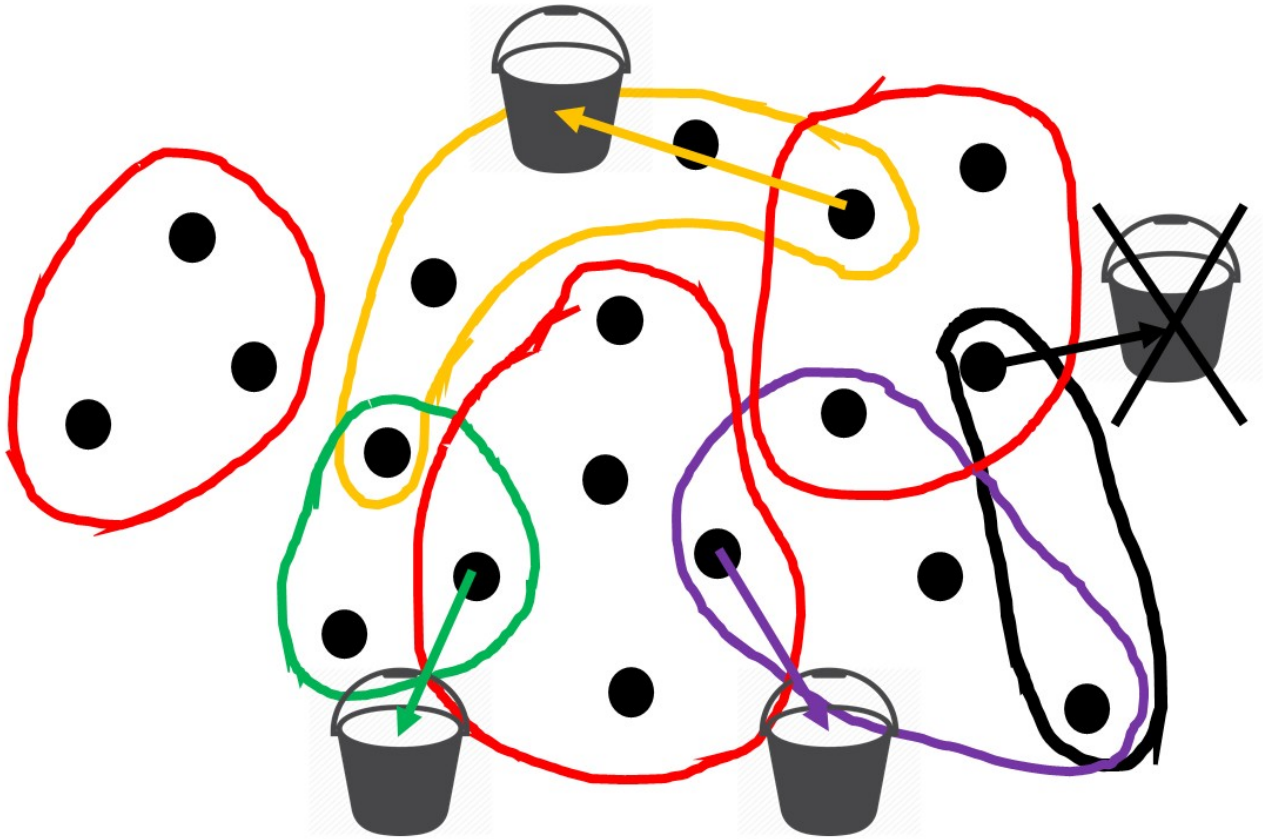
- If S is disjoint from M, M += S.
- If it intersects with M at node u:
- If there is $T \subset S \backslash \{u\}$ that already appears in $\ell_{|T|}$ sets in b_u, discard S. b_u has similar sets already.
  - Otherwise, add it.

Solving Max-k Cover Exactly

Set S arrives in stream.

- If S is disjoint from M, M += S.
- If it intersects with M at node u:

- If there is $T \subset S \backslash \{u\}$ that already appears in $\ell_{|T|}$ sets in b_u, discard S. b_u has similar sets already.
  - Otherwise, add it.

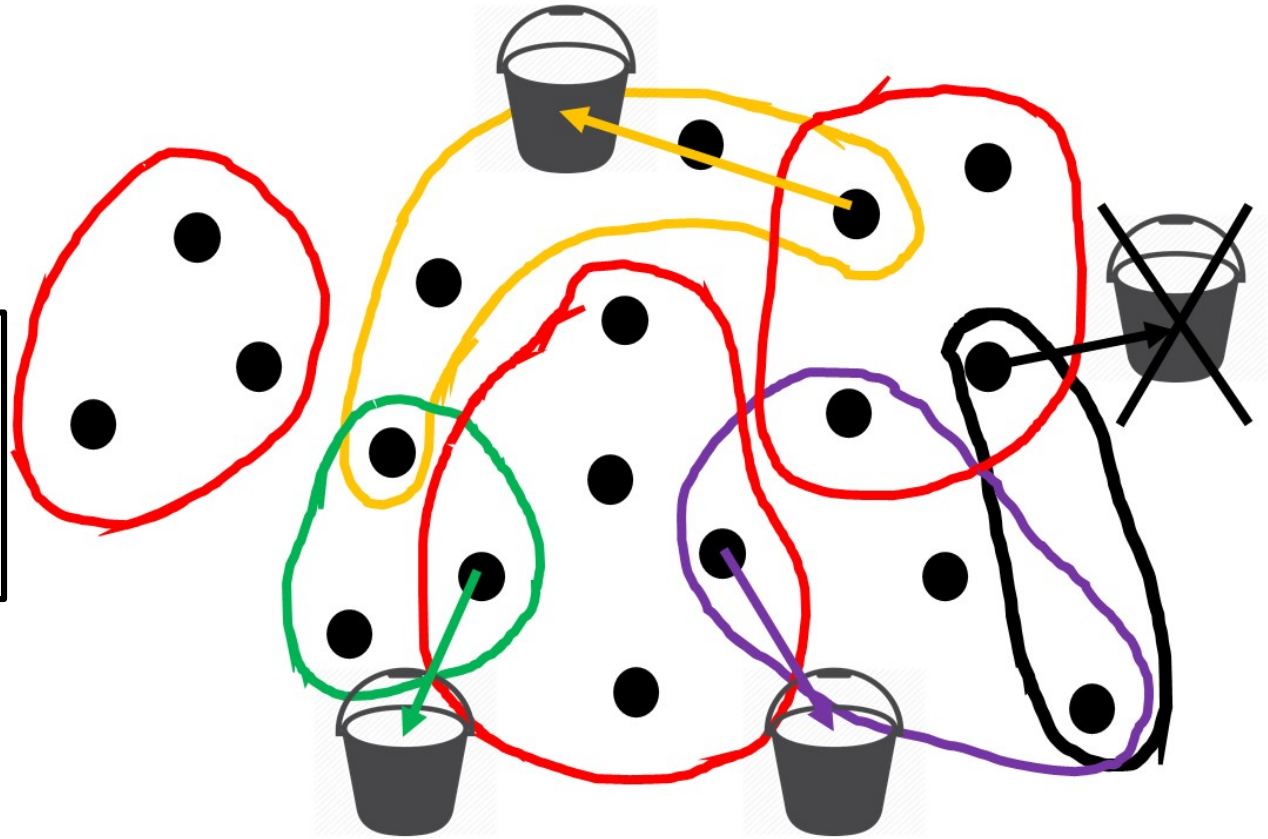


Solving Max-k Cover Exactly

Set S arrives in stream.

- If S is disjoint from M, M += S.
- If it intersects with M at node u:

- If there is $T \subset S \backslash \{u\}$ that already appears in $\ell_{|T|}$ sets in b_u, discard S. b_u has similar sets already.
  - Otherwise, add it.



If there is $T \subset S \backslash \{u\}$ that already appears in ℓ_T sets in b_u, discard S. b_u has similar sets already.

If there is $T \subset S \backslash \{u\}$ that already appears in ℓ_T sets in b_u, discard S. b_u has similar sets already.

Lemma: If we have $X = \{S_1, S_2, \ldots\}$ where $|S_i| = d - 1$, and some set T^* appearing ℓ_T times in X, then for any collection B of at most $d(k-1)$ other nodes, X has a set S' containing T^* and no nodes in B.

If there is $T \subset S \backslash \{u\}$ that already appears in ℓ_T sets in b_u, discard S. b_u has similar sets already.

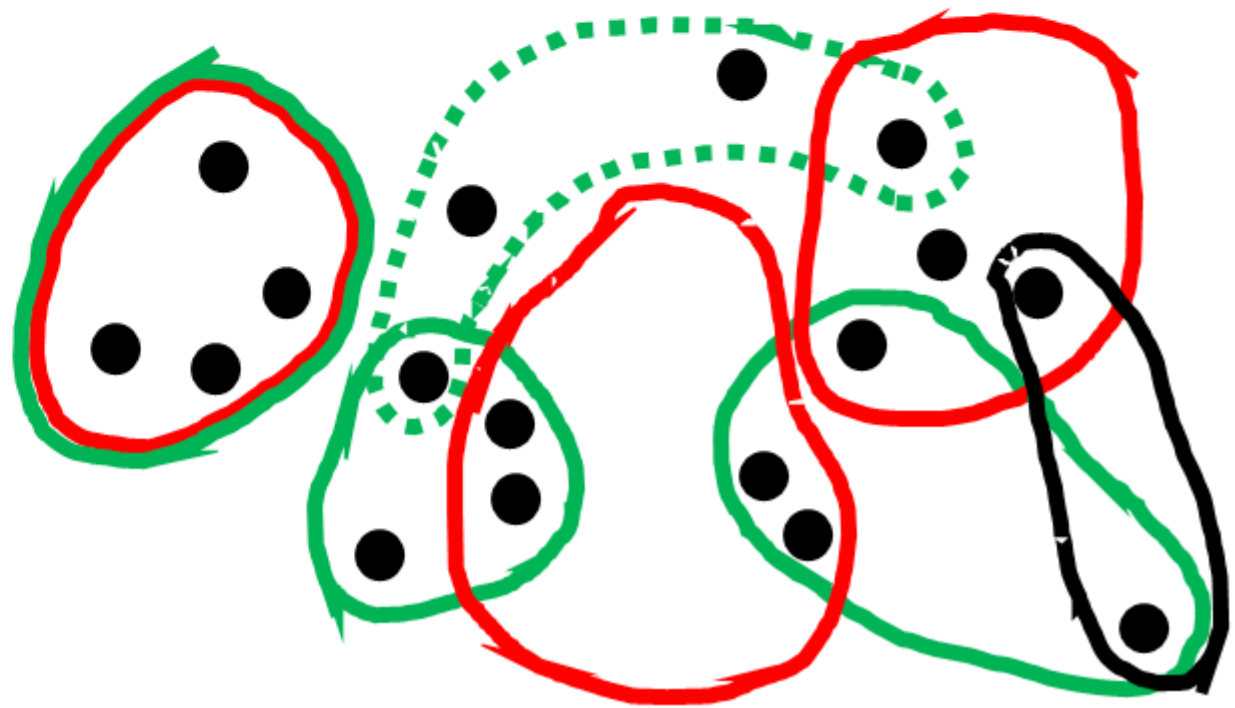
Lemma: If we have $X = \{S_1, S_2, \ldots\}$ where $|S_i| = d - 1$, and some set T^* appearing ℓ_T times in X, then for any collection B of at most $d(k-1)$ other nodes, X has a set S' containing T^* and no nodes in B.

Note: assume that all sets have size exactly d.

If there is $T \subset S \backslash \{u\}$ that already appears in ℓ_T sets in b_u, discard S. b_u has similar sets already.

Lemma: If we have $X = \{S_1, S_2, \ldots\}$ where $|S_i| = d - 1$, and some set T^* appearing ℓ_T times in X, then for any collection B of at most $d(k-1)$ other nodes, X has a set S' containing T^* and no nodes in B.

Set S_i discarded in stream.



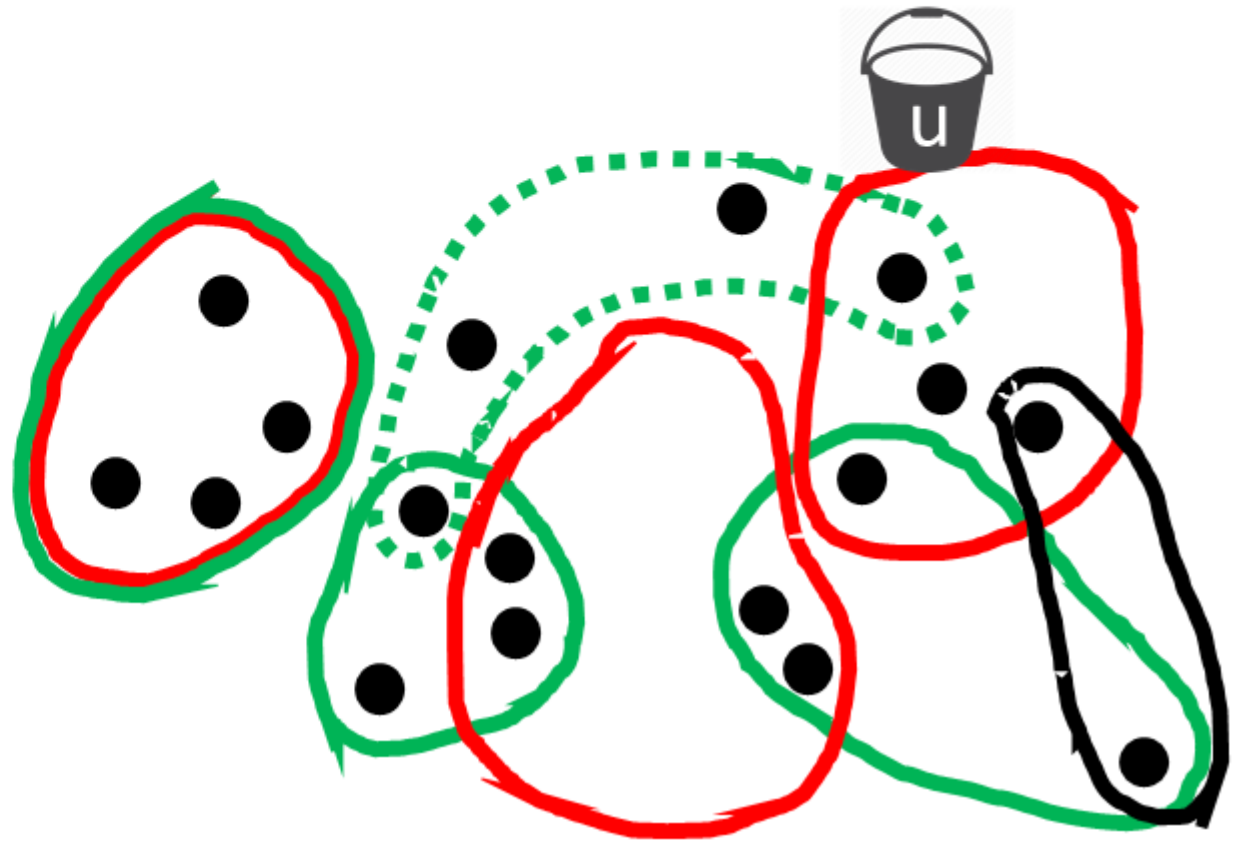
Matching in red

OPT in green

If there is $T \subset S \backslash \{u\}$ that already appears in ℓ_T sets in b_u, discard S. b_u has similar sets already.

Lemma: If we have $X = \{S_1, S_2, \ldots\}$ where $|S_i| = d - 1$, and some set T^* appearing ℓ_T times in X, then for any collection B of at most $d(k-1)$ other nodes, X has a set S' containing T^* and no nodes in B.

Set S_i discarded in stream. S_i overlaps M at node u.



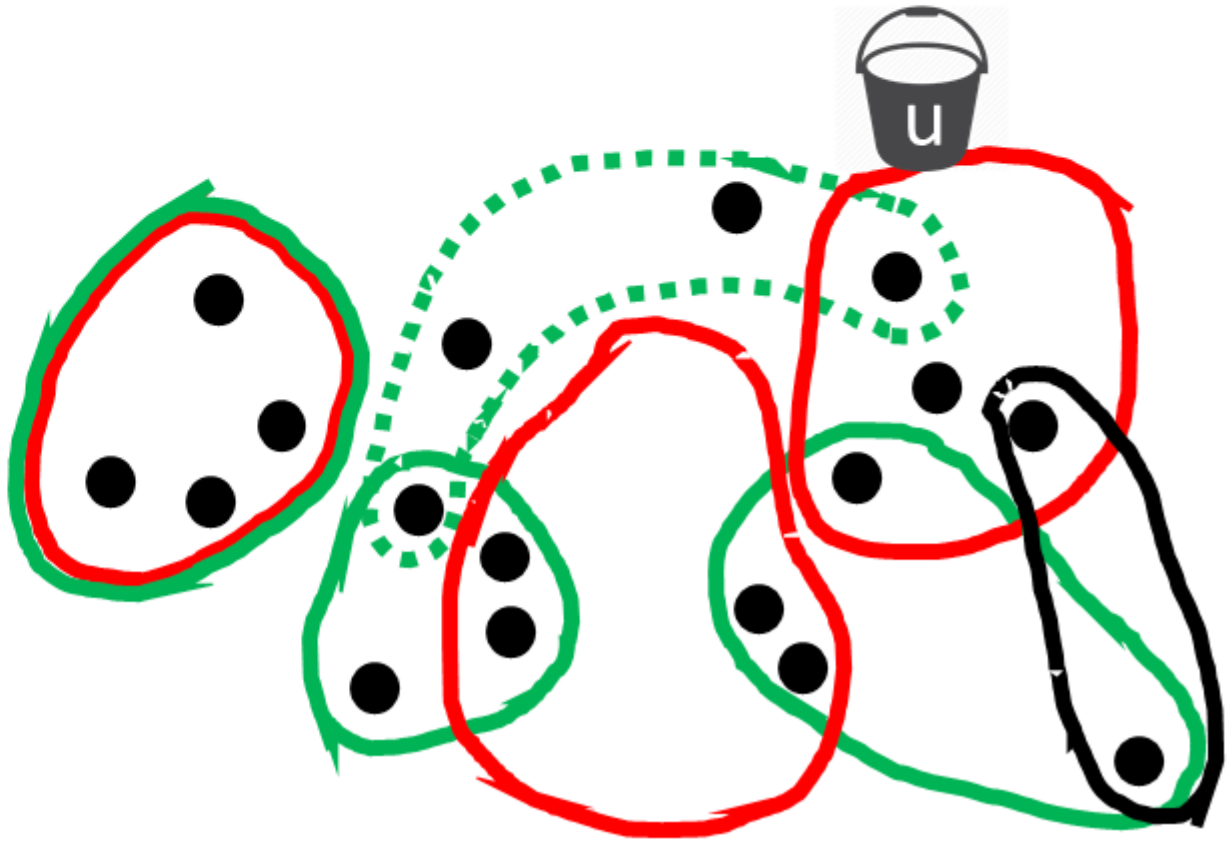
Matching in red

OPT in green

If there is T $\subset S\backslash\{u\}$ that already appears in ℓ_T sets in b_u, discard S. b_u has similar sets already.

Lemma: If we have $X = \{S_1, S_2, \ldots\}$ where $|S_i| = d - 1$, and some set T^* appearing ℓ_T times in X, then for any collection B of at most $d(k-1)$ other nodes, X has a set S' containing T^* and no nodes in B.

Set S_i discarded in stream. S_i overlaps M at node u. ∃ some $T^* \subset S_i$ which already appears ℓ_{T^*} times in b_u.



Matching in red

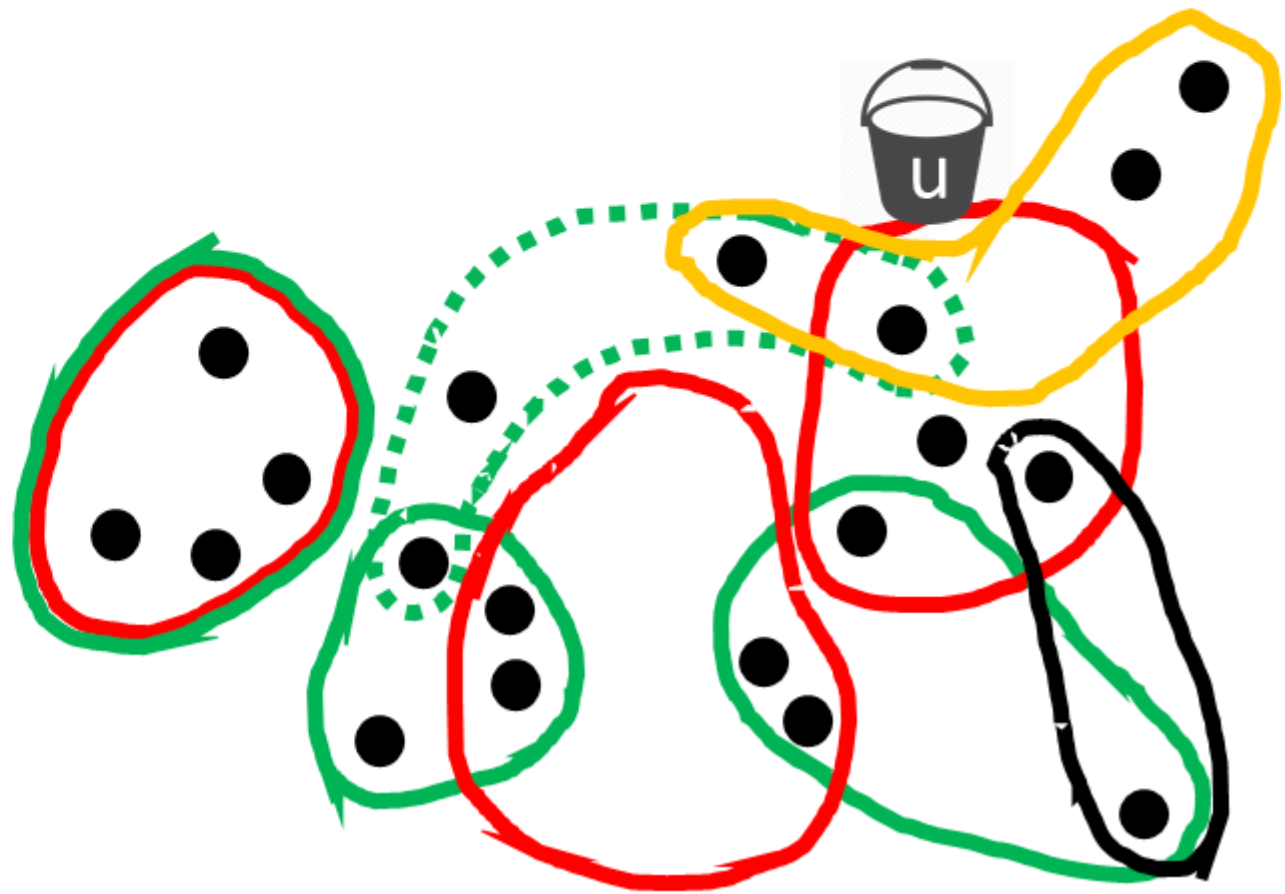
OPT in green

If there is T $\subset S\backslash\{u\}$ that already appears in ℓ_T sets in b_u, discard S. b_u has similar sets already.

Lemma: If we have $X = \{S_1, S_2, \ldots\}$ where $|S_i| = d - 1$, and some set T^* appearing ℓ_T times in X, then for any collection B of at most $d(k-1)$ other nodes, X has a set S' containing T^* and no nodes in B.

Set S_i discarded in stream. S_i overlaps M at node u. ∃ some $T^* \subset S_i$ which already appears ℓ_{T^*} times in b_u.

Lemma: $X = b_u, B = OPT \backslash\{S_i\}$, then b_u has replacement S': $|S' \cap B| = 0$.



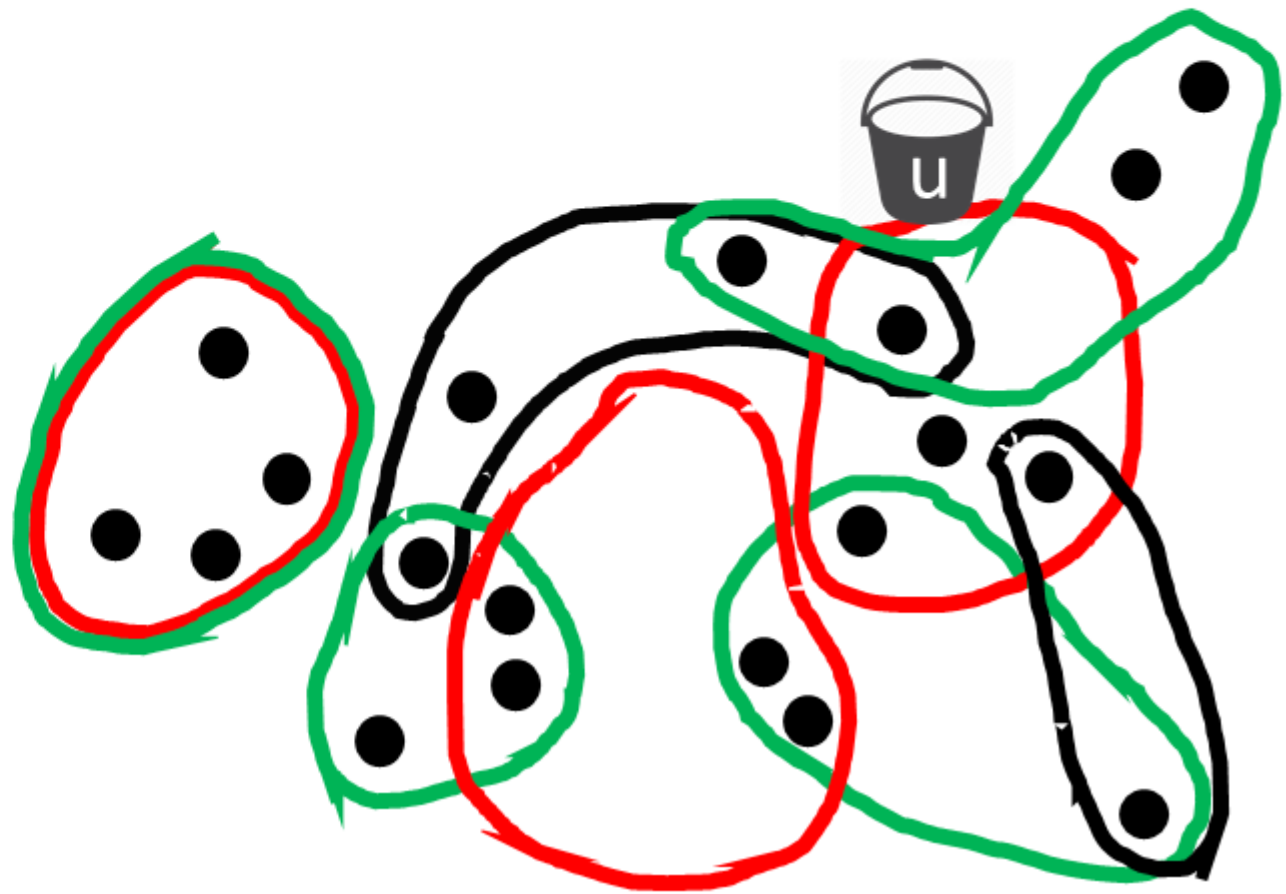
Matching in red

OPT in green

If there is $T \subset S\backslash\{u\}$ that already appears in ℓ_T sets in b_u, discard S. b_u has similar sets already.

Lemma: If we have $X = \{S_1, S_2, \ldots\}$ where $|S_i| = d - 1$, and some set T^* appearing ℓ_T times in X, then for any collection B of at most $d(k-1)$ other nodes, X has a set S' containing T^* and no nodes in B.

Since $|S' \cap B| = 0$, we can replace OPT with $(OPT\backslash\{S_i\}) \cup \{S'\}$. This new solution is just as good.

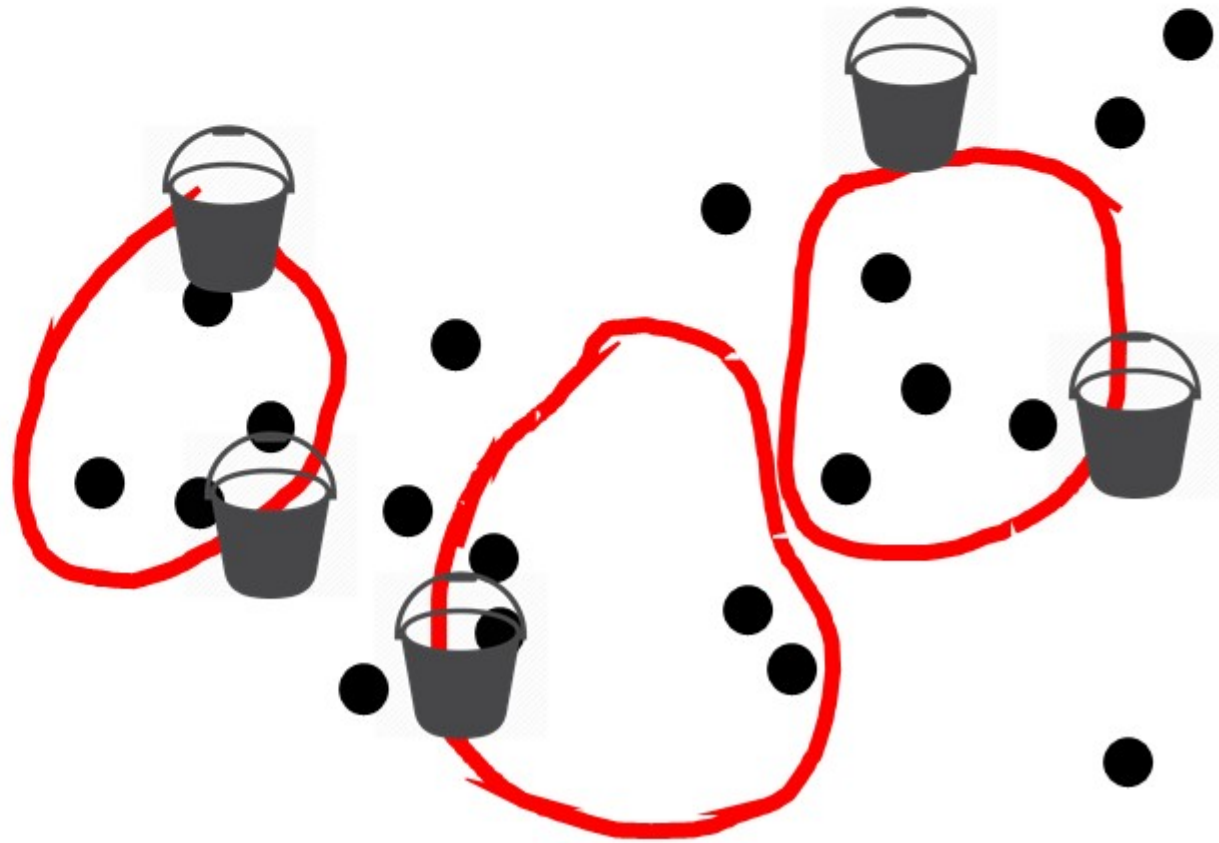


Matching in red

OPT in green

If there is T $\subset S\backslash\{u\}$ that already appears in ℓ_T sets in b_u, discard S. b_u has similar sets already.

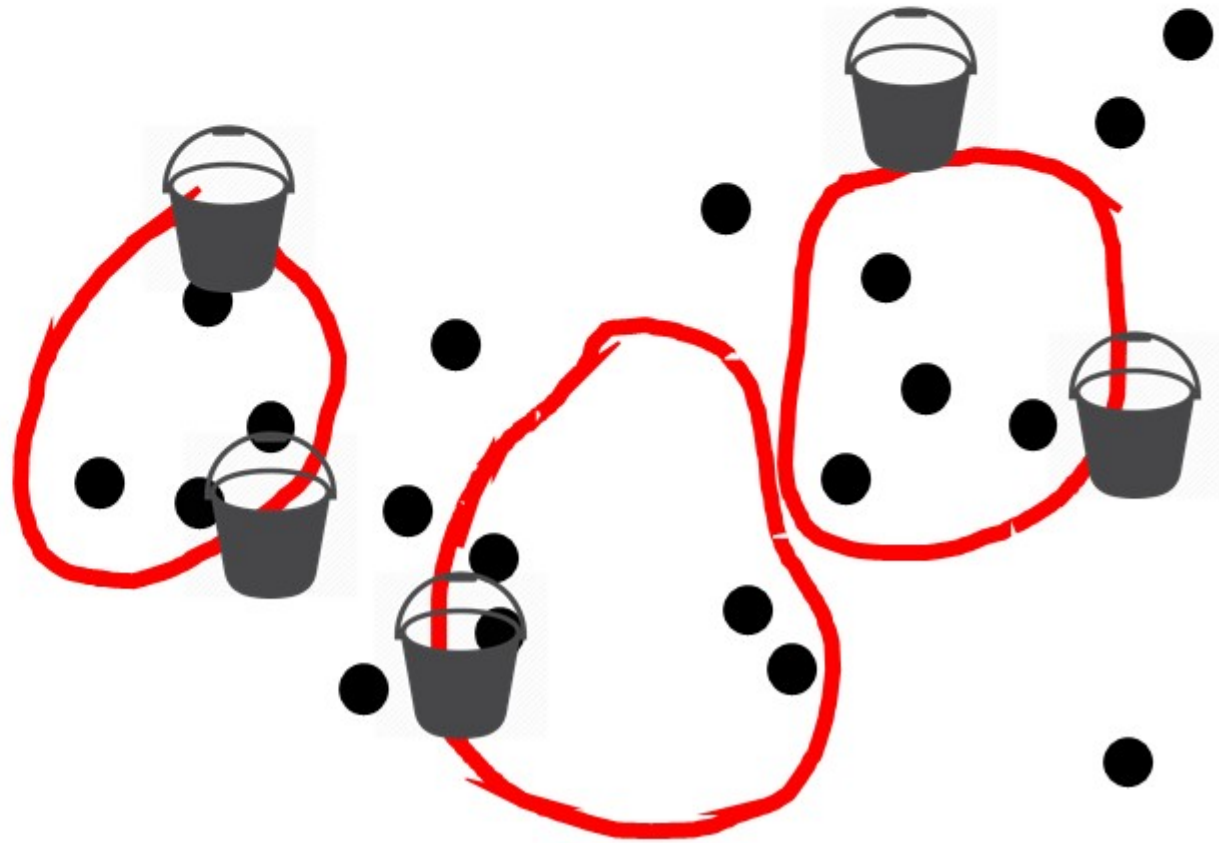
We kept matching M and some number of buckets. How many edges in total?



If there is T $\subset S\backslash\{u\}$ that already appears in ℓ_T sets in b_u, discard S. b_u has similar sets already.

$$\ell_T := (d(k-1)+1)^{d-1-|T|}$$

We kept matching M and some number of buckets. How many edges in total?



If there is T $\subset S\backslash\{u\}$ that already appears in ℓ_T sets in b_u, discard S. b_u has similar sets already.

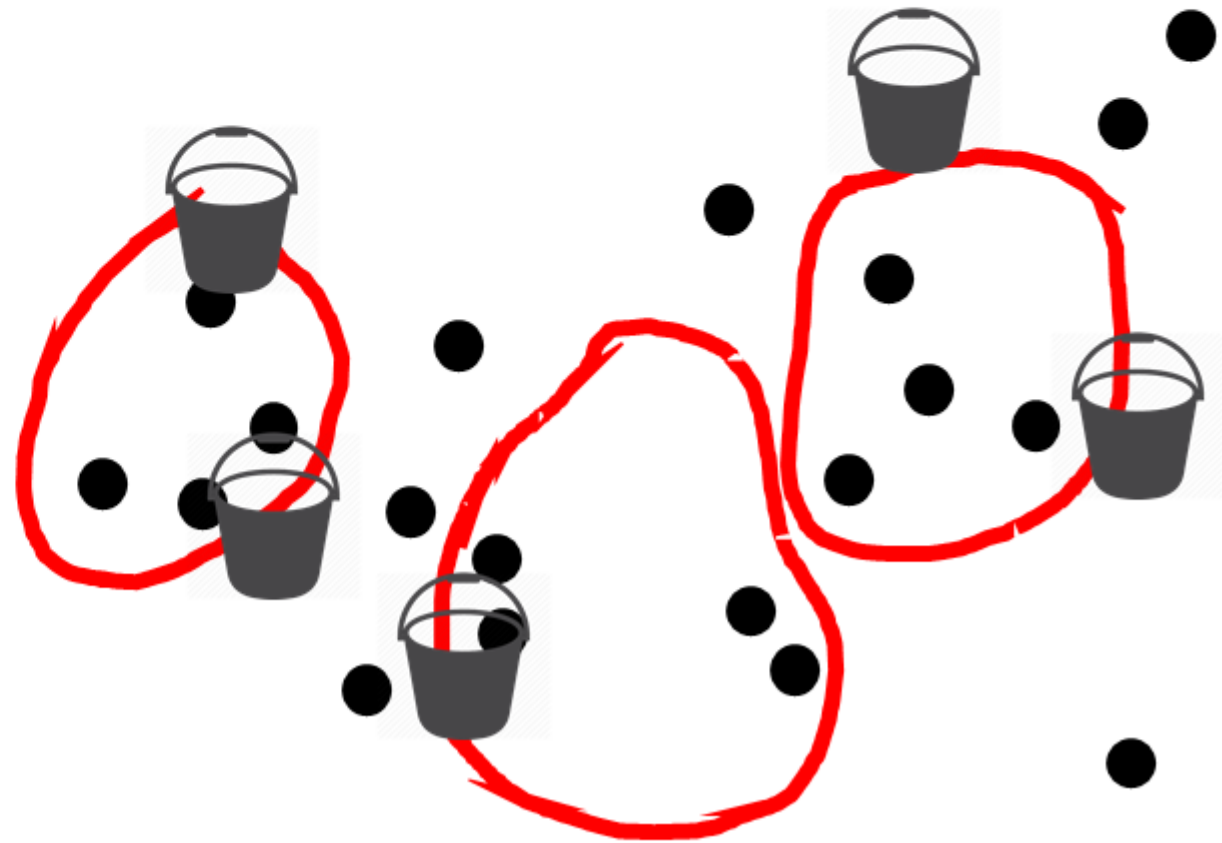
$\ell_T := (d(k-1)+1)^{d-1-|T|}$

Each bucket can have at most

$\ell_0 = (d(k-1)+1)^{d-1-0}$

$= O((dk)^{d-1})$ edges because every edge has the subset {}.

We kept matching M and some number of buckets. How many edges in total?



If there is T $\subset S\backslash\{u\}$ that already appears in ℓ_T sets in b_u, discard S. b_u has similar sets already.

$\ell_T := (d(k-1)+1)^{d-1-|T|}$

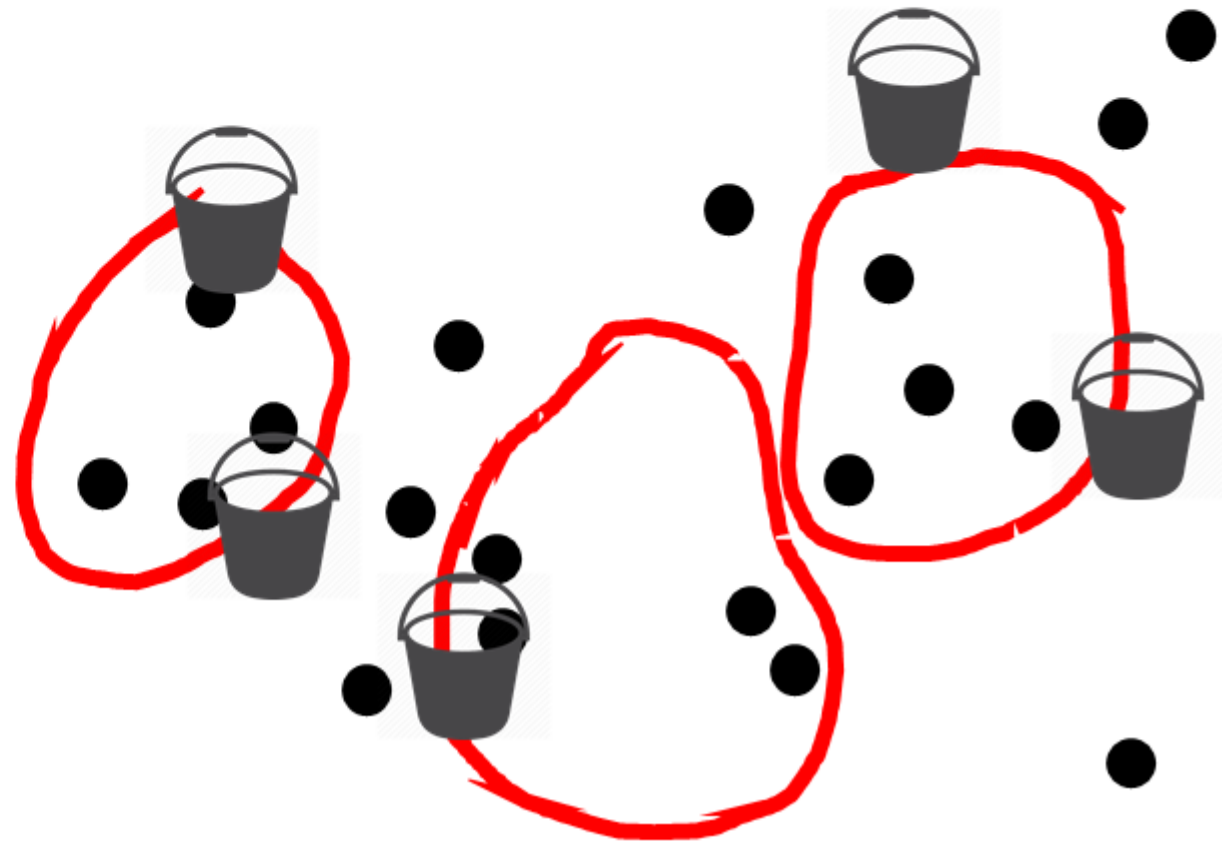
Each bucket can have at most

$\ell_0 = (d(k-1)+1)^{d-1-0}$

$= O((dk)^{d-1})$ edges because every edge has the subset {}.

$|M| < k$ and so there are $< dk$ buckets.

We kept matching M and some number of buckets. How many edges in total?



If there is T $\subset S\backslash\{u\}$ that already appears in ℓ_T sets in b_u, discard S. b_u has similar sets already.

$\ell_T := (d(k-1)+1)^{d-1-|T|}$

Each bucket can have at most

$\ell_0 = (d(k-1)+1)^{d-1-0}$

$= O((dk)^{d-1})$ edges because every edge has the subset {}.

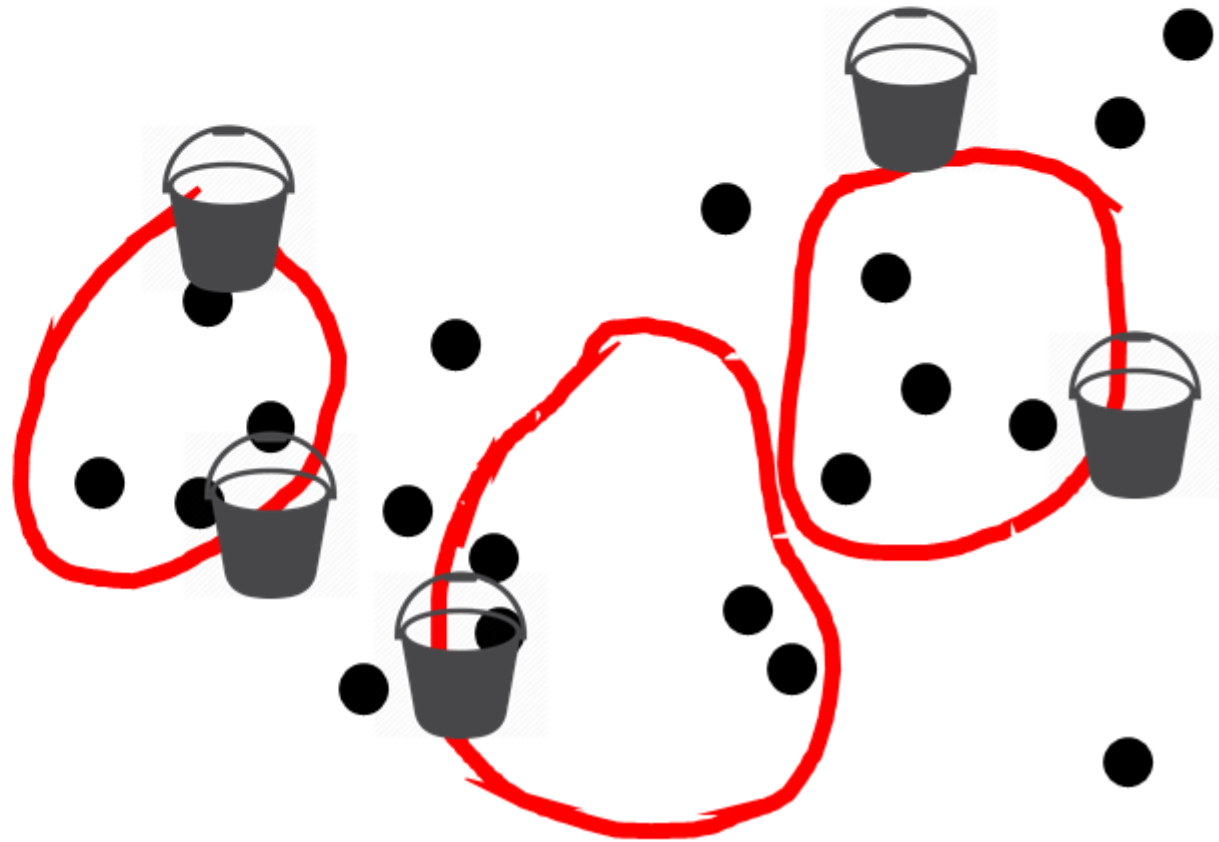
$|M| < k$ and so there are $< dk$ buckets.

So we keep

$|M| + dk\, O\left((dk)^{d-1}\right) = O((dk)^d)$

edges.

d nodes per edge means total space is $O(d^{d+1}k^d)$.

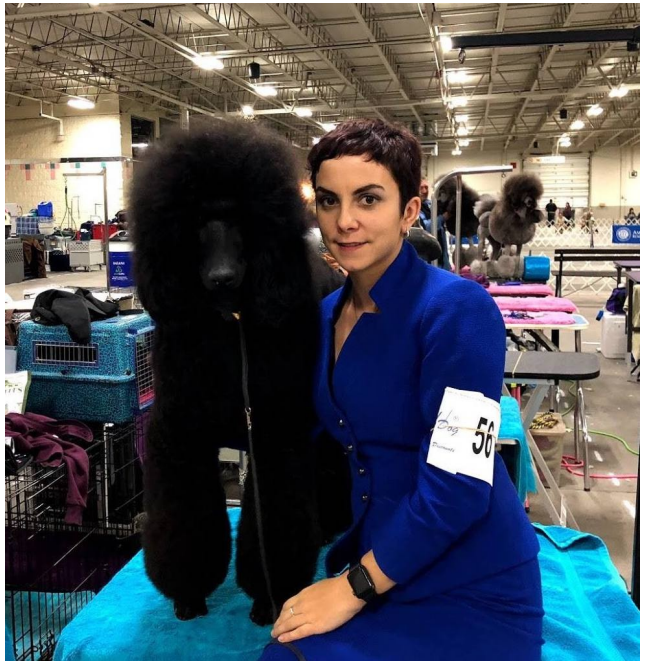


PathCache

EFFICIENTLY MAPPING THE INTERNET

Joint work with:

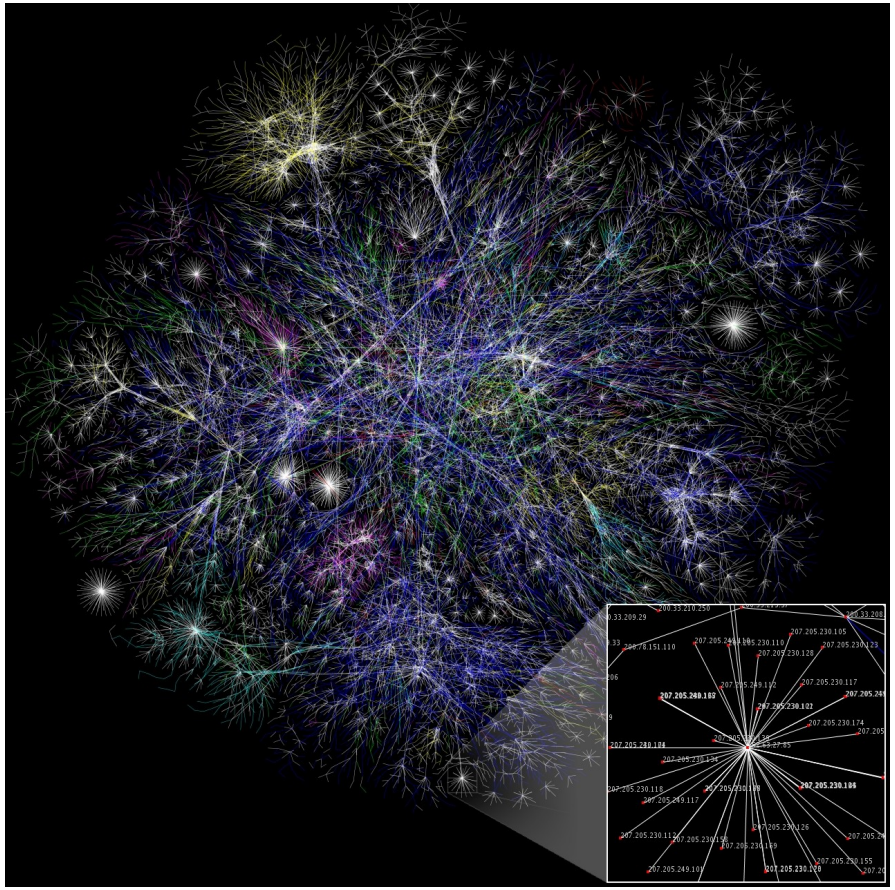
Andrew McGregor



Phillipa Gill

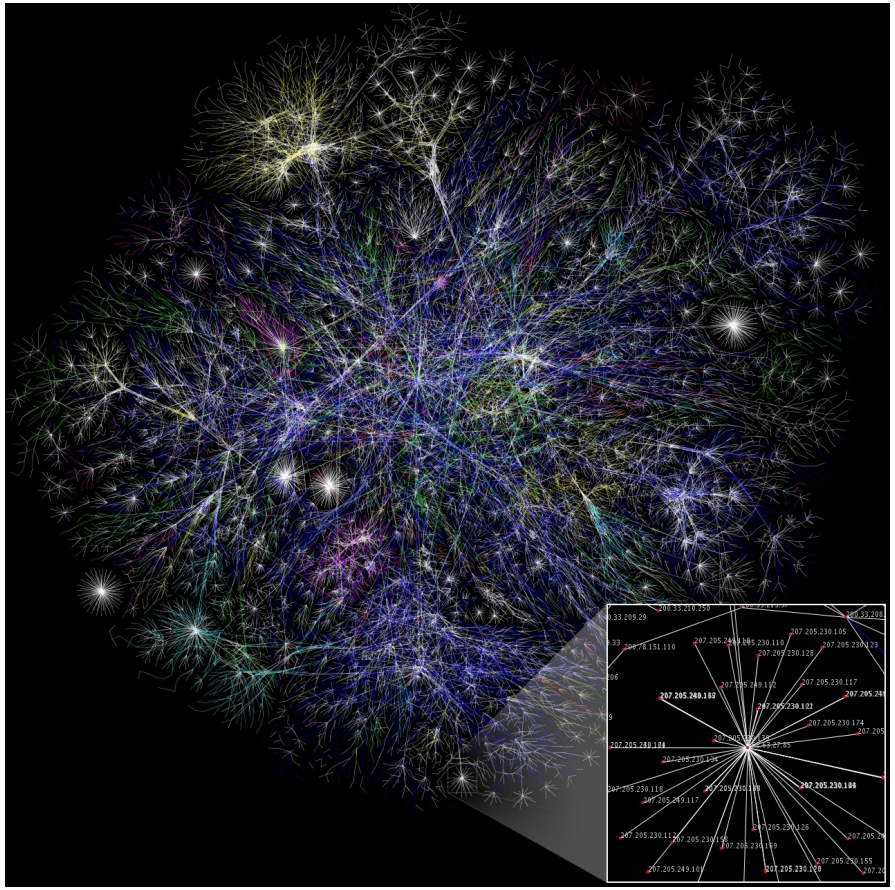
Rachee Singh

Understanding Internet Paths



Want to map the Internet to predict paths. Useful for:

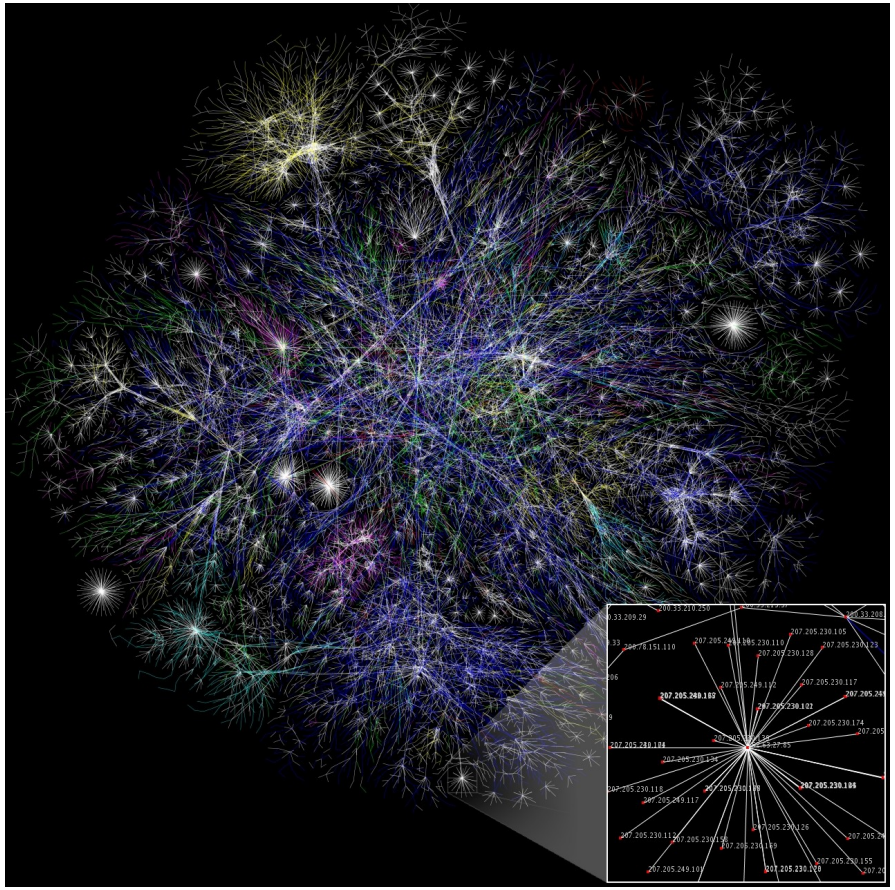
Understanding Internet Paths



Want to map the Internet to predict paths. Useful for:

- Network administrators

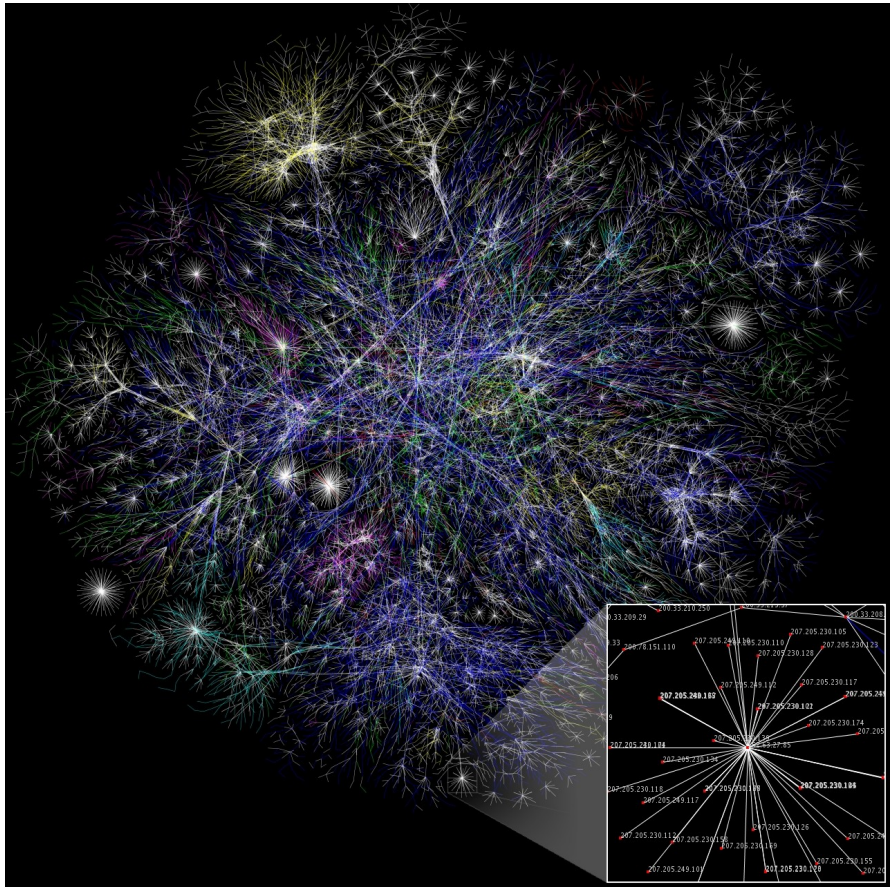
Understanding Internet Paths



Want to map the Internet to predict paths. Useful for:

- Network administrators
- Censorship Researchers

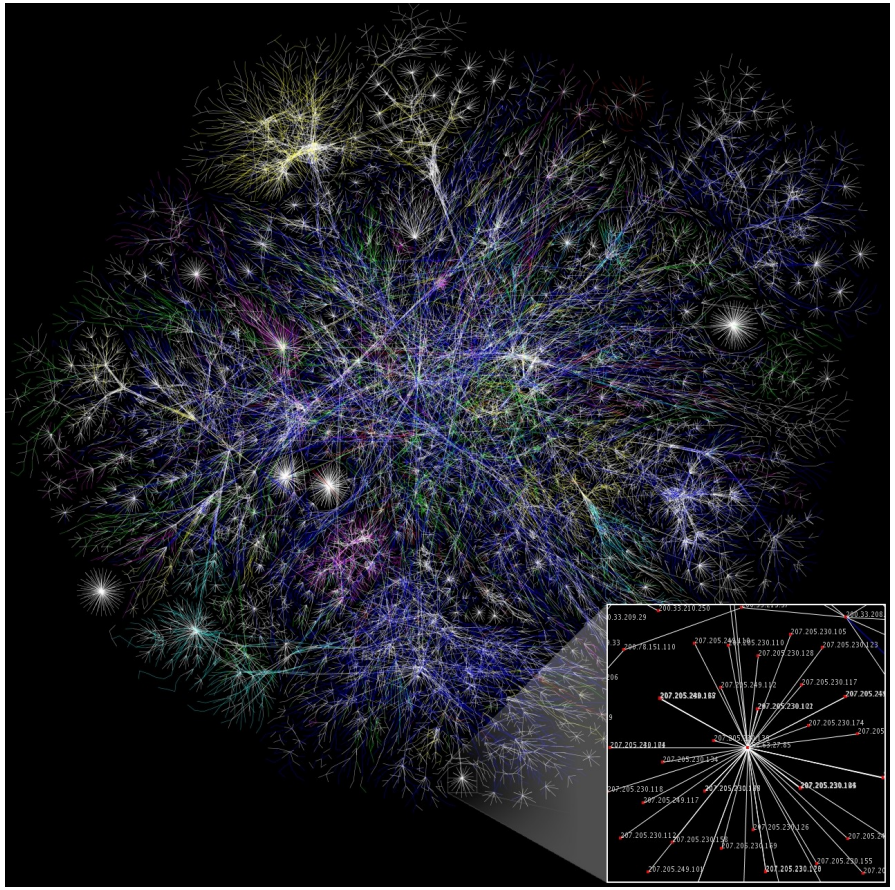
Understanding Internet Paths



Want to map the Internet to predict paths. Useful for:

- Network administrators
- Censorship Researchers
- Cloud service providers

Understanding Internet Paths



Want to map the Internet to predict paths. Useful for:

- Network administrators
- Censorship Researchers
- Cloud service providers

Introducing PathCache, a system for predicting Internet paths accurately & efficiently.

Internet Basics

172.16.254.1

IP address: numerical label assigned to connected device

Internet Basics

172.16.254.1

172.16.254.0/24

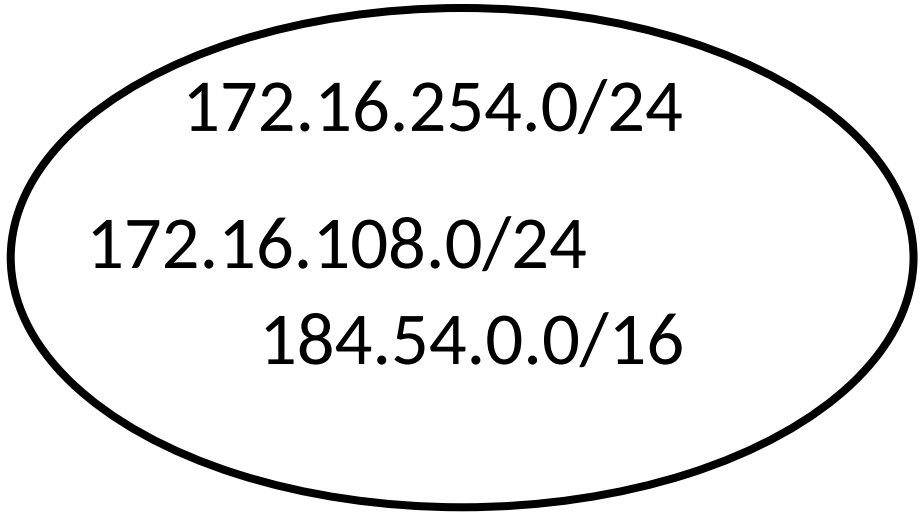
IP address: numerical label assigned to connected device

IP prefix: range of sequential IP addresses with same prefix

Internet Basics

172.16.254.1

172.16.254.0/24



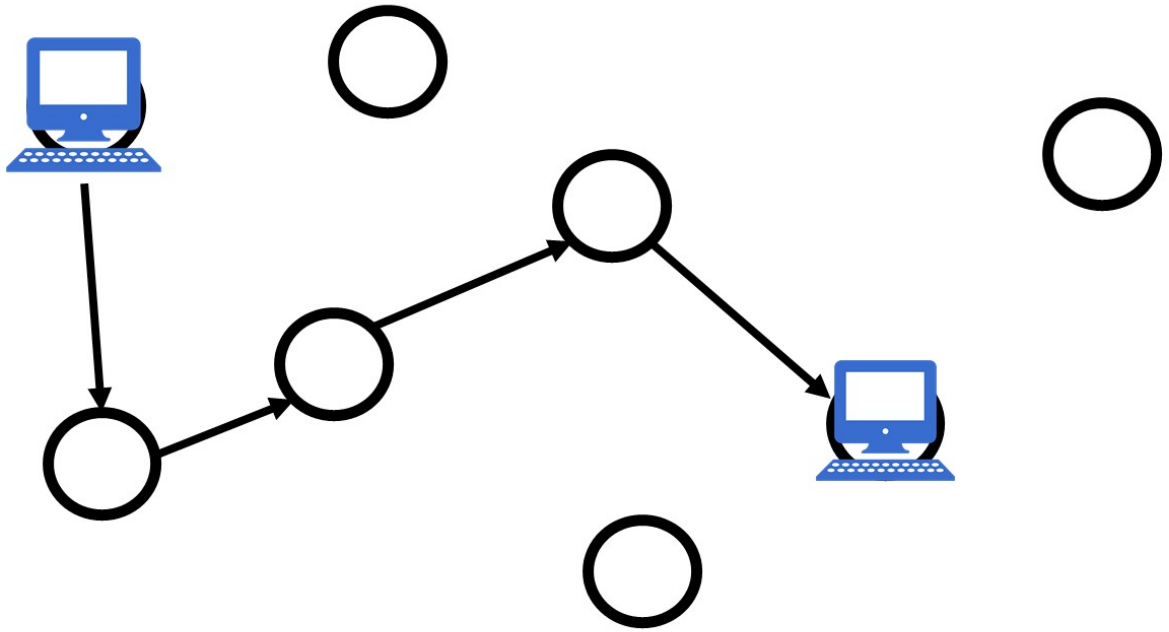
IP address: numerical label assigned to connected device

IP prefix: range of sequential IP addresses with same prefix

Autonomous System (AS): collection of IP prefixes owned by network w/ 1 routing policy

At a high level, the Internet is made up of ASes. Routing handled by BGP.

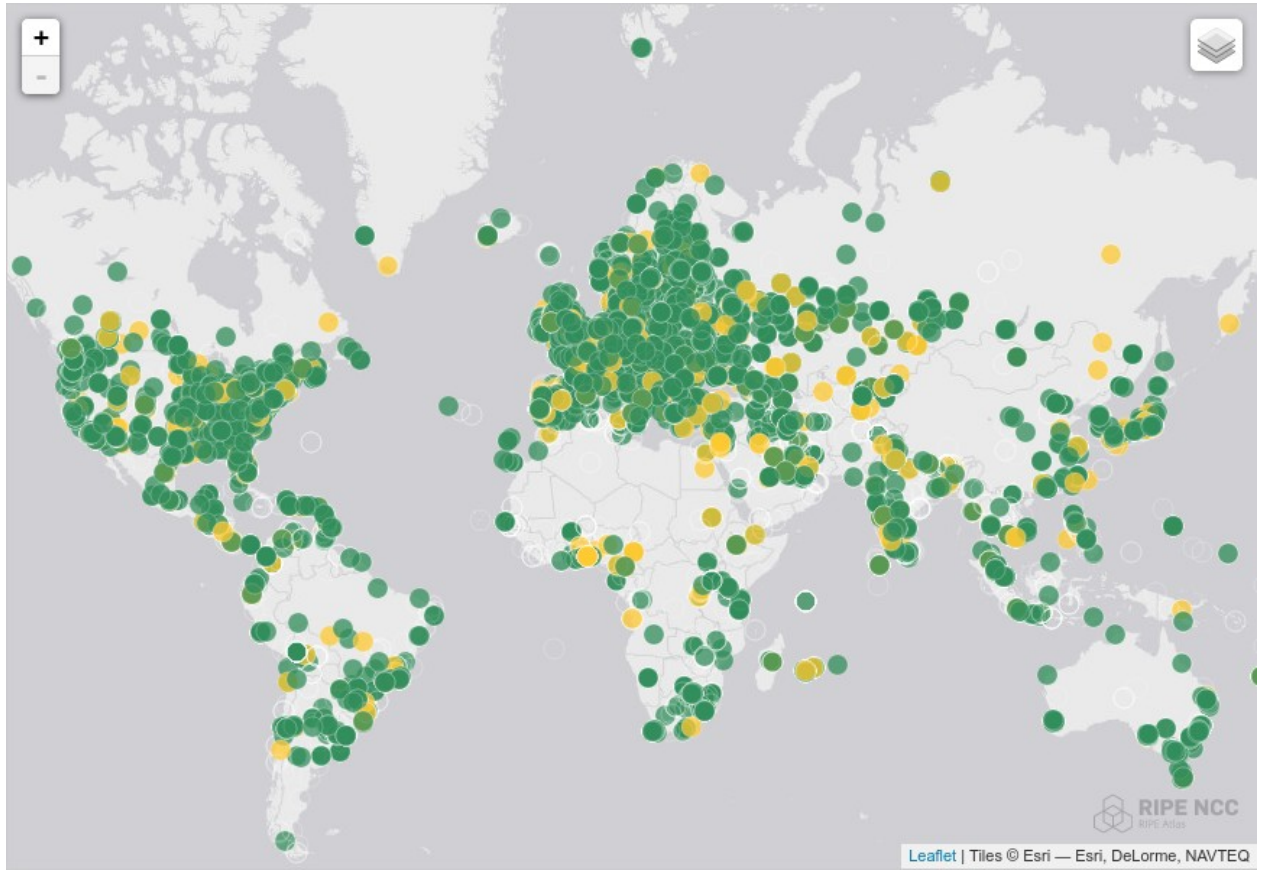
Internet Basics



Traceroute: path measurement between host and other device

You need to control a host to issue a traceroute from it.

Internet Basics

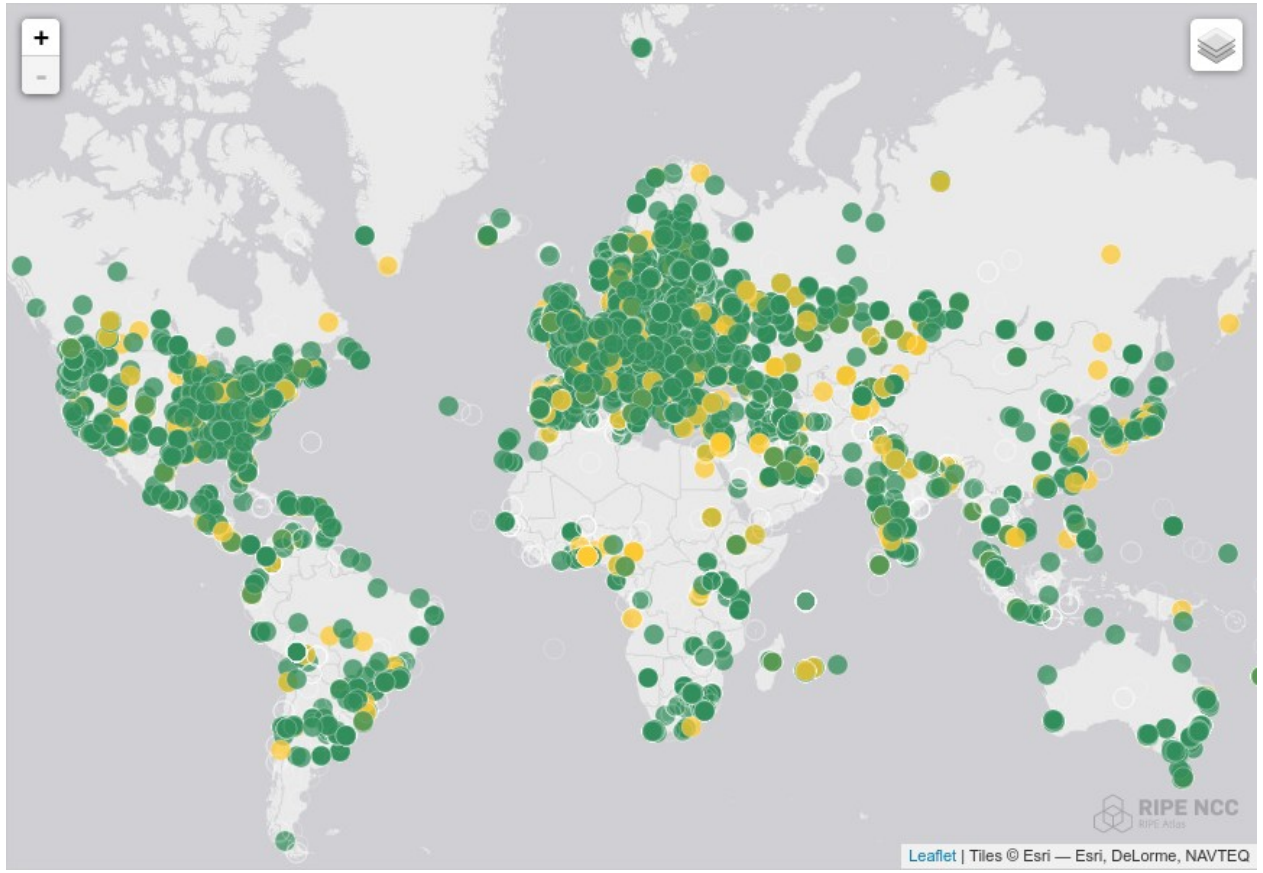


Traceroute: path measurement between host and other device

You need to control a host to issue a traceroute from it.

Vantage points: devices which do traceroutes for researchers. VPs can't be used too often.

Internet Basics



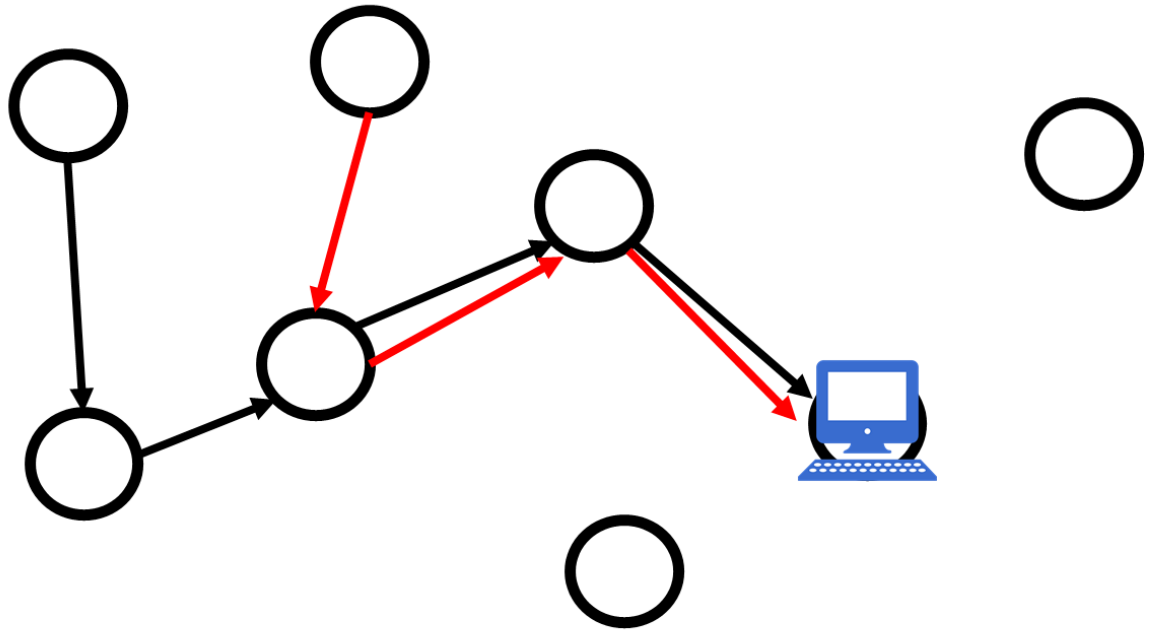
Traceroute: path measurement between host and other device

You need to control a host to issue a traceroute from it.

Vantage points: devices which do traceroutes for researchers. VPs can't be used too often.

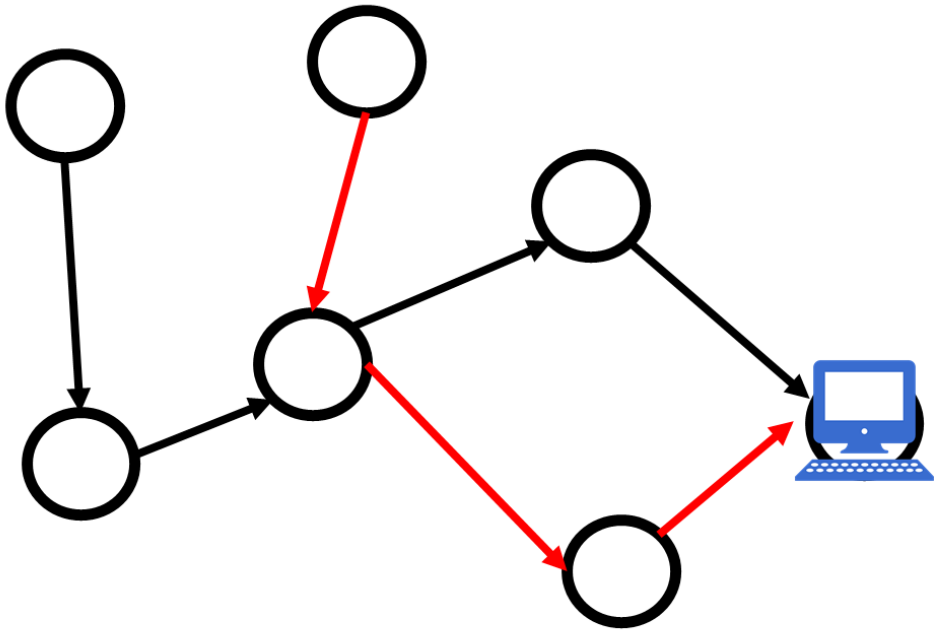
Goal: make few measurements & predict paths from results.

Internet Basics



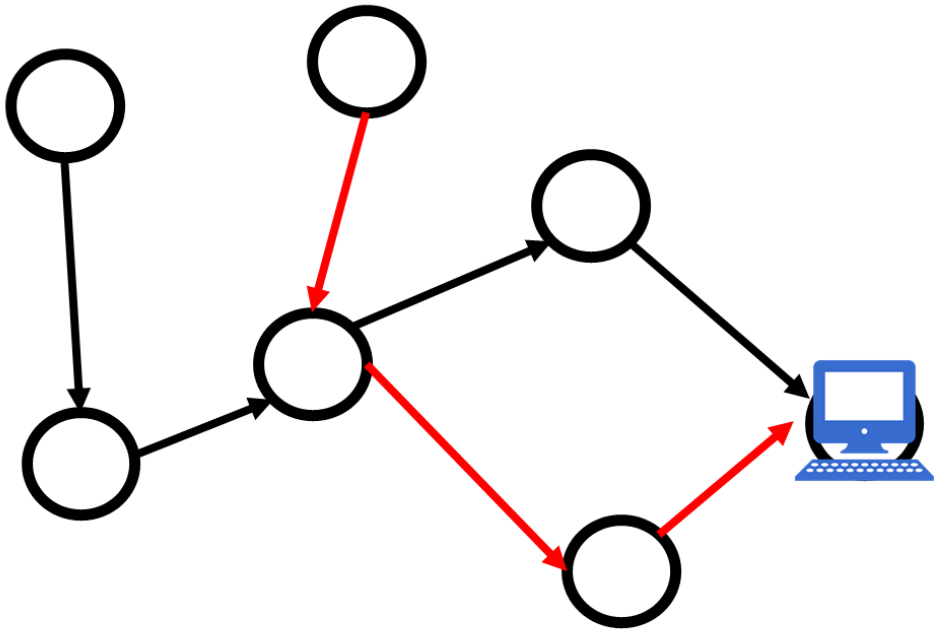
<u>Destination-based routing (DBR):</u>
Messages to same destination take consistent paths thru AS graph.

Internet Basics



<u>Destination-based routing (DBR):</u>
Messages to same destination take consistent paths thru AS graph.

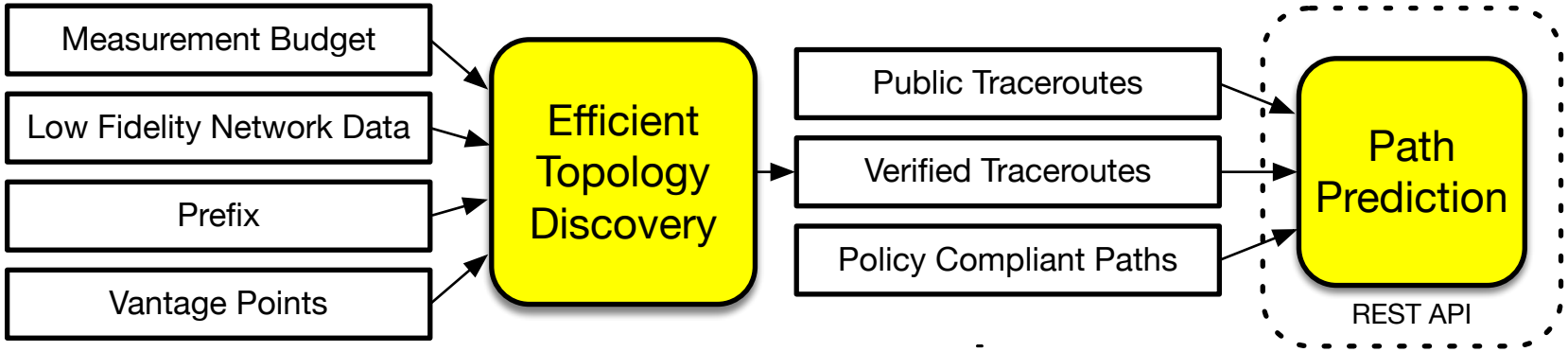
Internet Basics



<u>Destination-based routing (DBR):</u>
Messages to same destination take consistent paths thru AS graph.

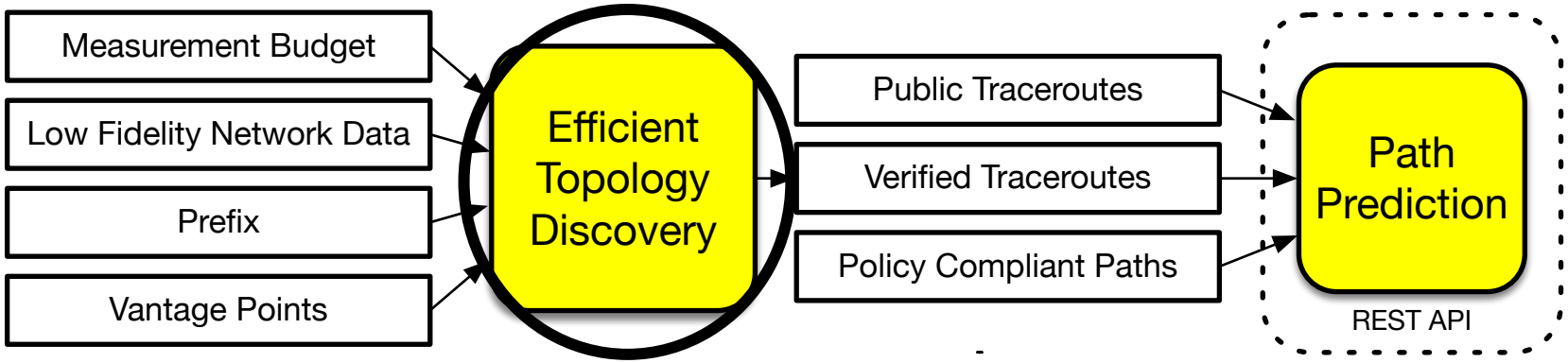
PathCache insight: DBR consistency allows organizing measurements & path predictions by destination

PathCache System overview



PathCache starts with untrusted routing data towards a prefix *P*. It:

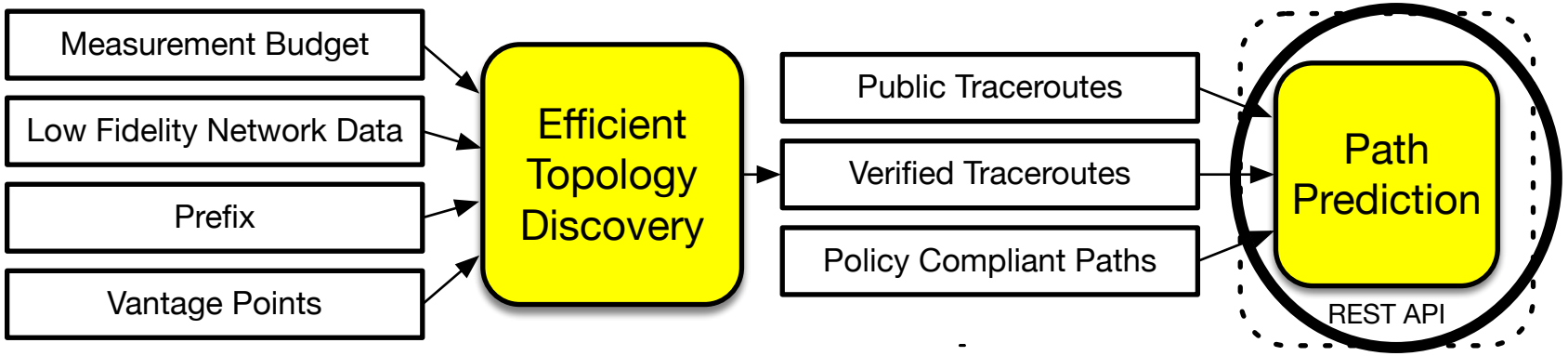
PathCache System overview



PathCache starts with untrusted routing data towards a prefix *P*. It:

1. Uses this data to make traceroutes to *P* from VPs via a cover algorithm.

PathCache System overview

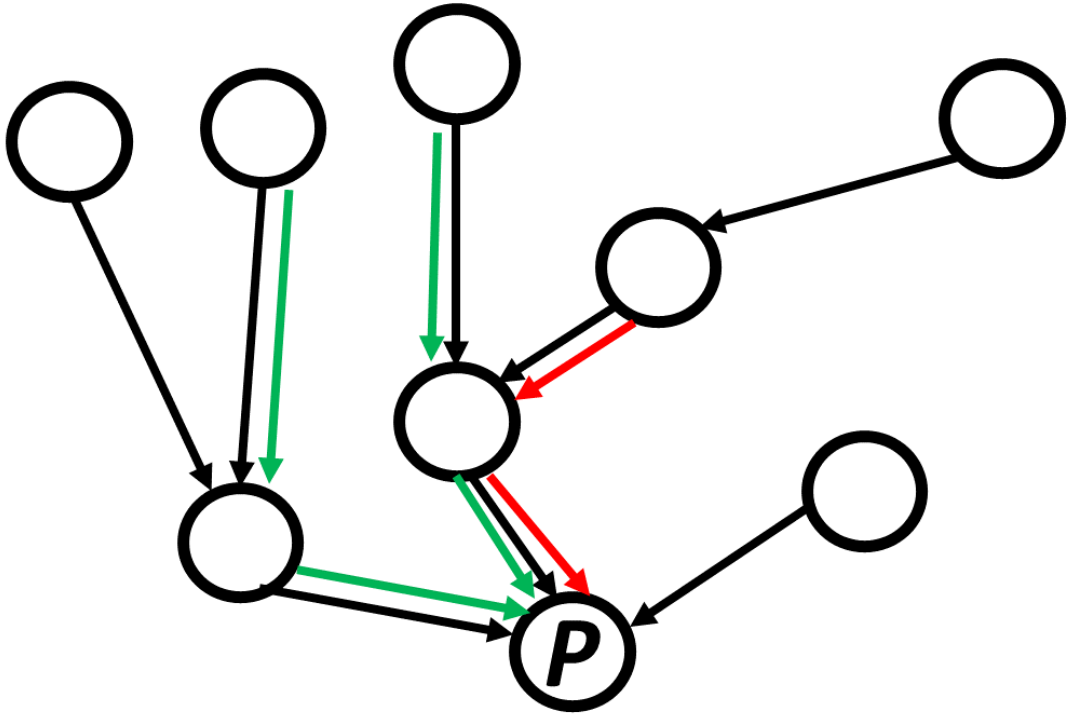


PathCache starts with untrusted routing data towards a prefix *P*. It:

1. Uses this data to make traceroutes to *P* from VPs via a cover algorithm.

2. Combines traceroute results to make a directed graph which it uses to predict path from <u>any</u> starting location to prefix *P*.

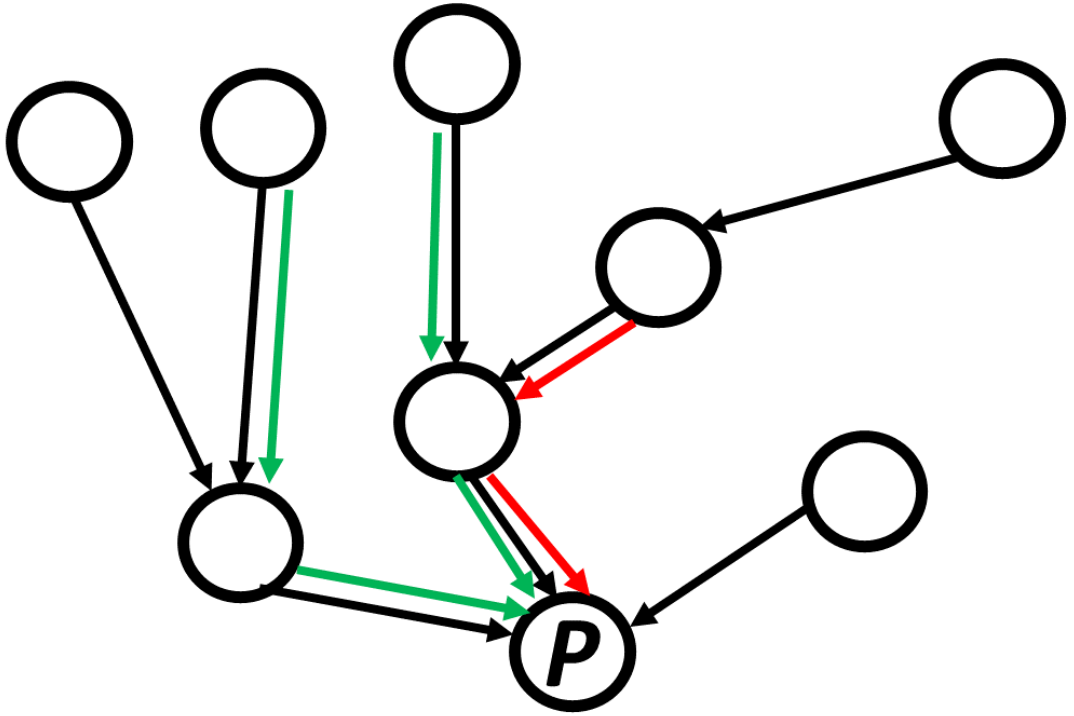
Efficient Topology Discovery



Input is untrusted path data

- Stale traceroutes (out of date?)
- BGP routing info (maybe wrong)

Efficient Topology Discovery

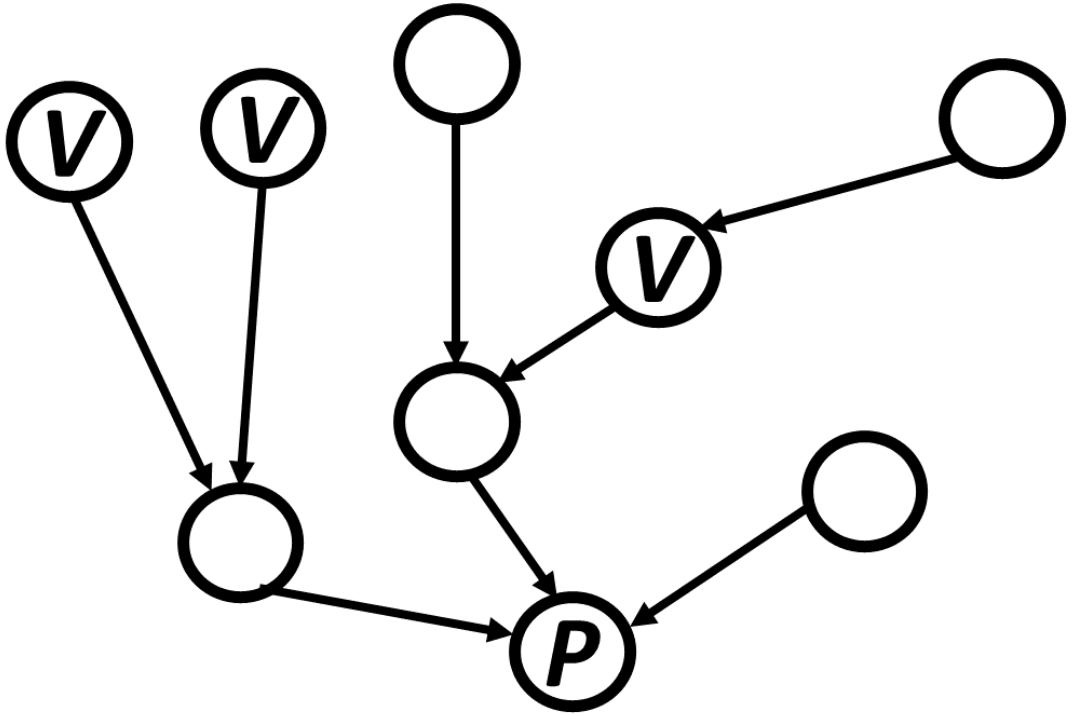


Input is untrusted path data

- Stale traceroutes (out of date?)
- BGP routing info (maybe wrong)

We expect the paths in this data to form a directed tree rooted at *P*.

Efficient Topology Discovery



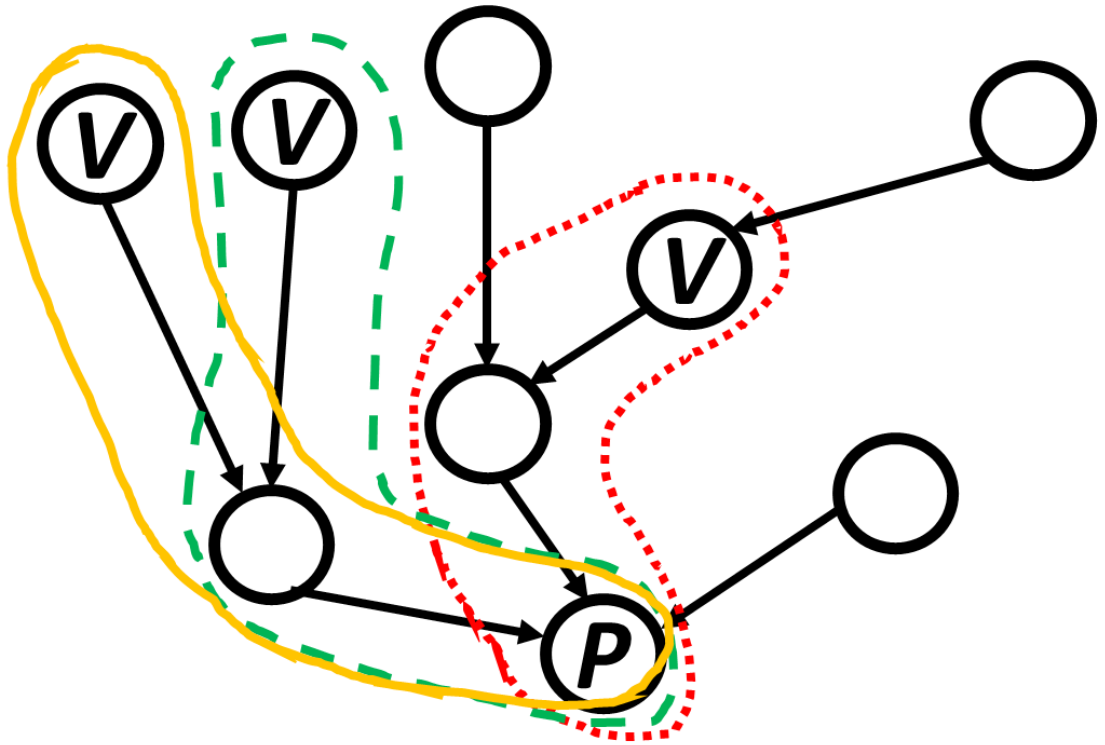
Input is untrusted path data

- Stale traceroutes (out of date?)
- BGP routing info (maybe wrong)

We expect the paths in this data to form a directed tree rooted at *P*.

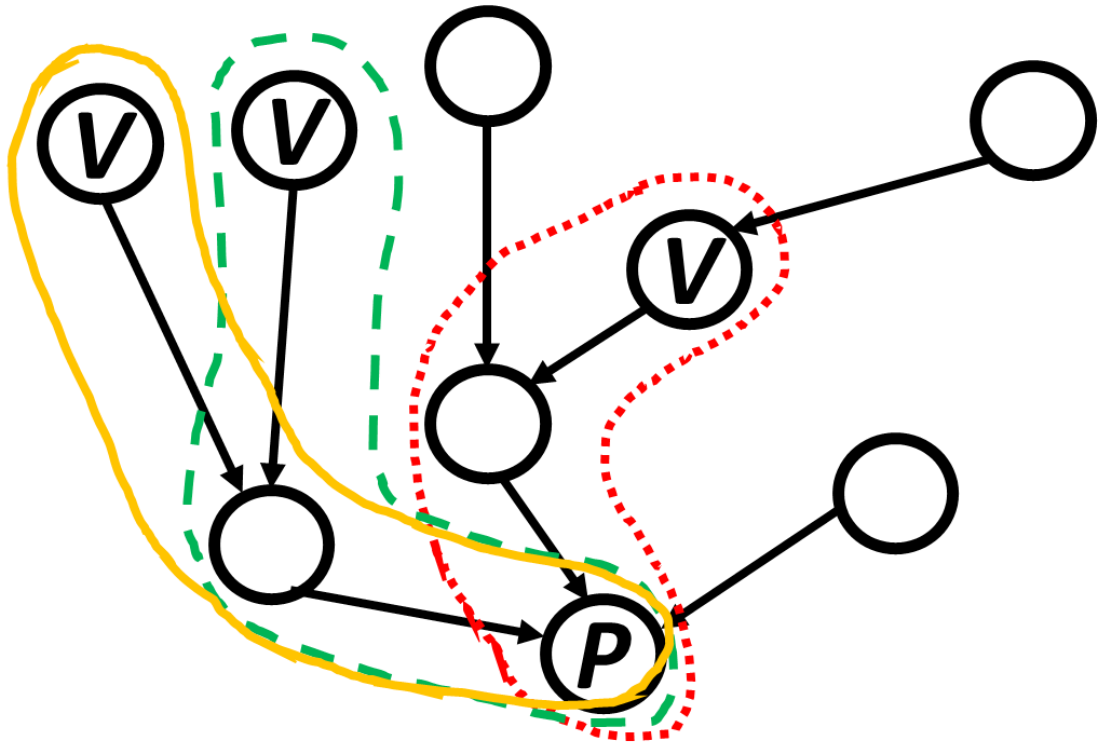
Output: k VPs whose traceroutes will maximize edges discovered.

Efficient Topology Discovery



We can think of each VP as a set: the path from VP to prefix *P*. Pick k sets that maximize edge coverage. Sound familiar?

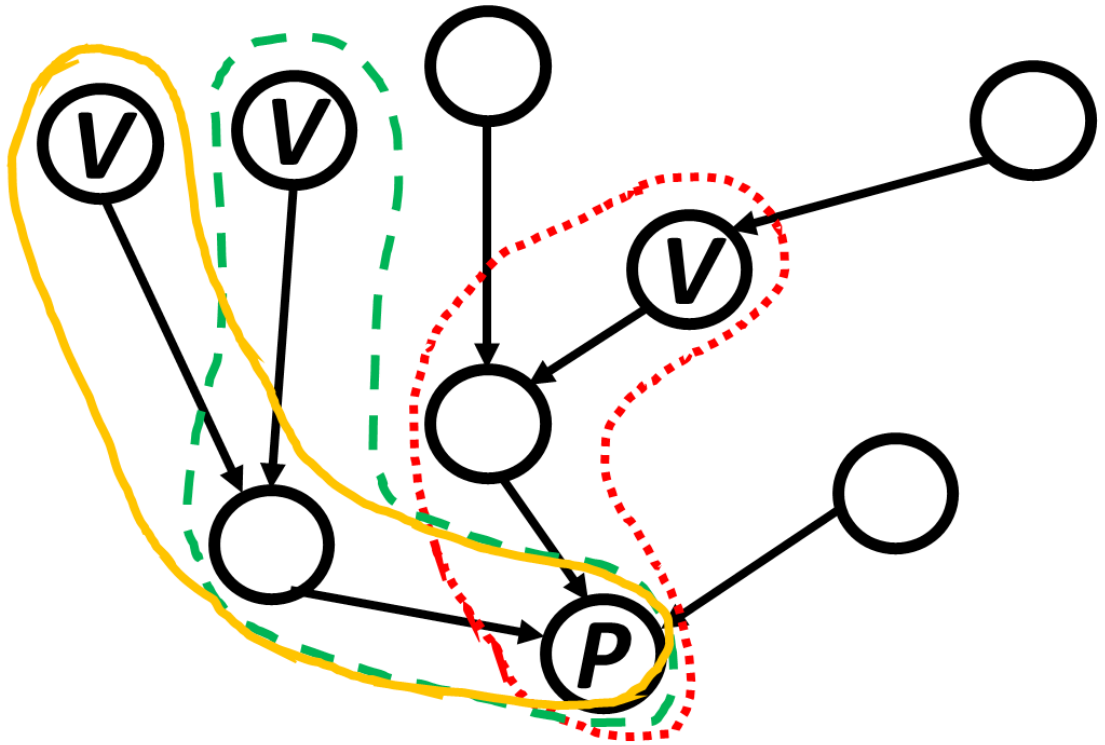
Efficient Topology Discovery



We can think of each VP as a set: the path from VP to prefix *P*. Pick k sets that maximize edge coverage. Sound familiar?

It's a special case of Max-k Cover!

Efficient Topology Discovery

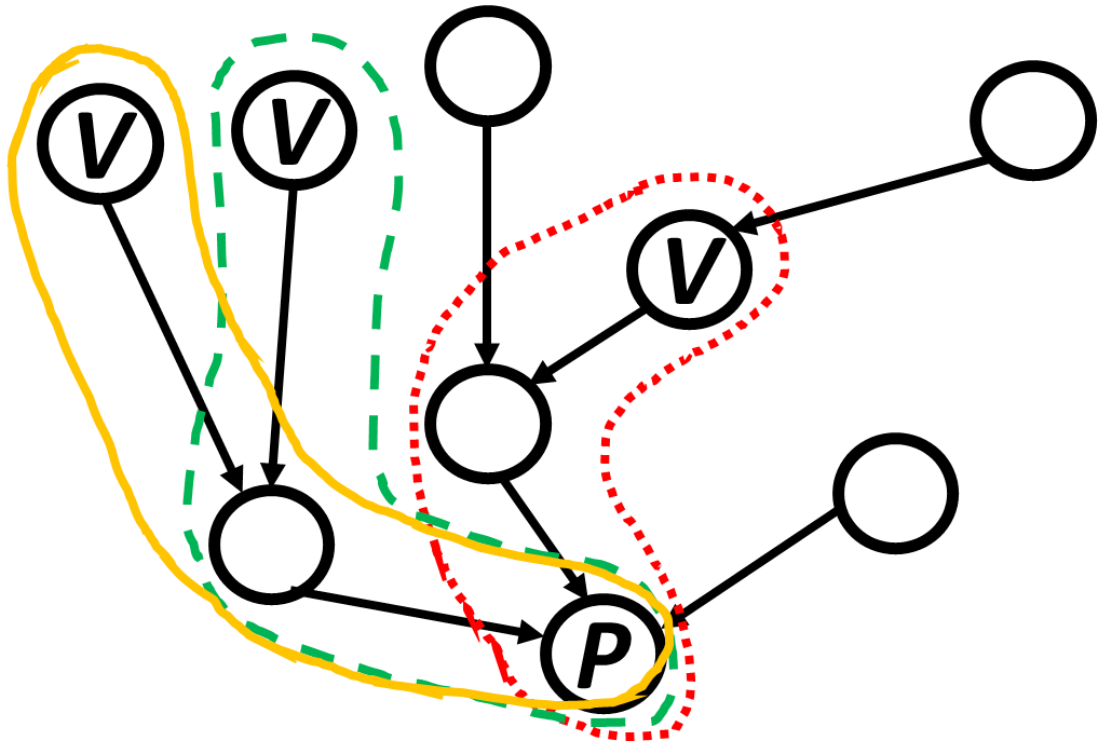


We can think of each VP as a set: the path from VP to prefix *P*. Pick k sets that maximize edge coverage. Sound familiar?

It's a special case of Max-k Cover!

Recall: NP-Hard, greedy algorithm gives (1-1/e)-approximation

Efficient Topology Discovery

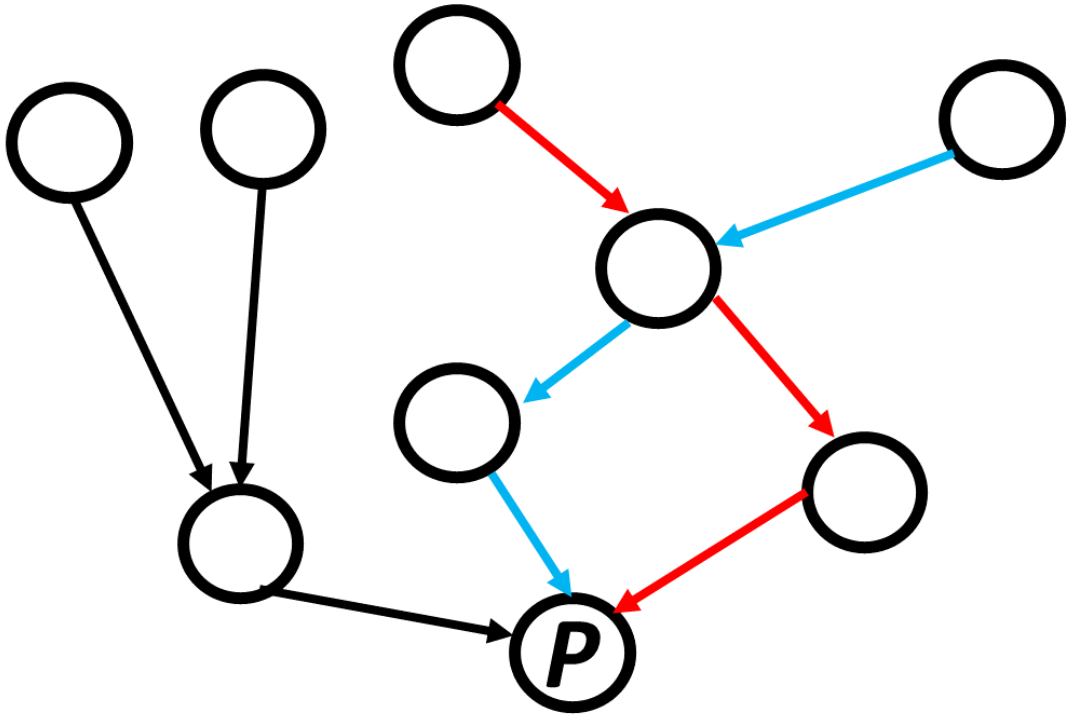


We can think of each VP as a set: the path from VP to prefix *P*. Pick k sets that maximize edge coverage. Sound familiar?

It's a special case of Max-k Cover!

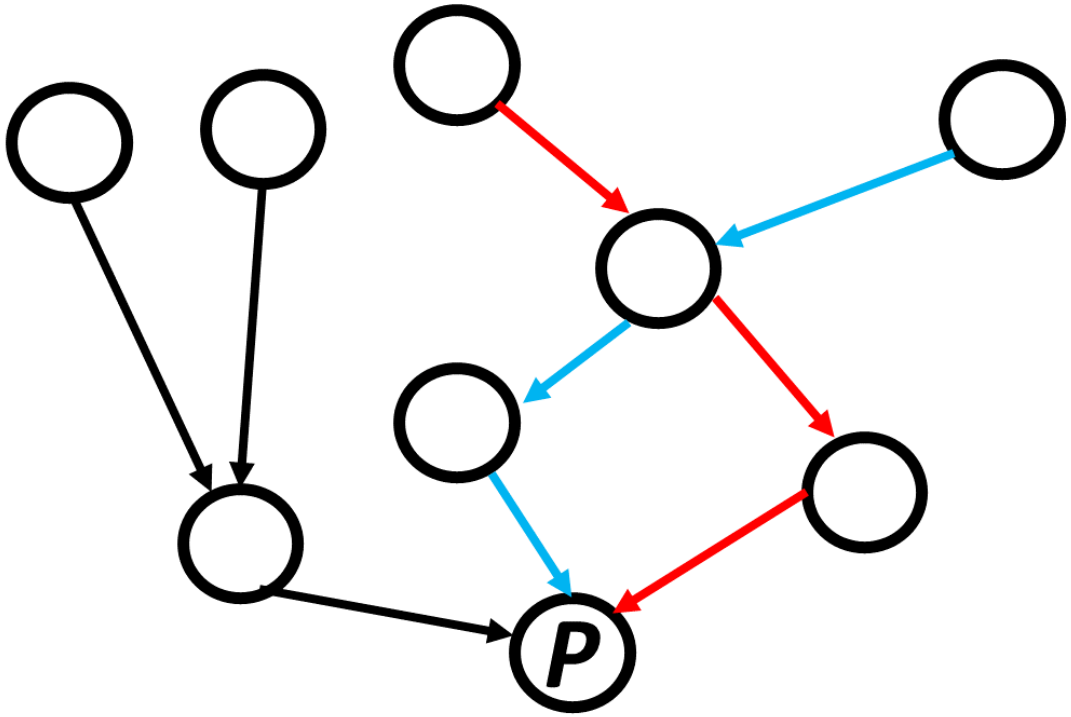
In this special case: greedy algorithm gives optimal coverage.

Efficient Topology Discovery



Sometimes our untrusted input data violates DBR, and isn't a tree.

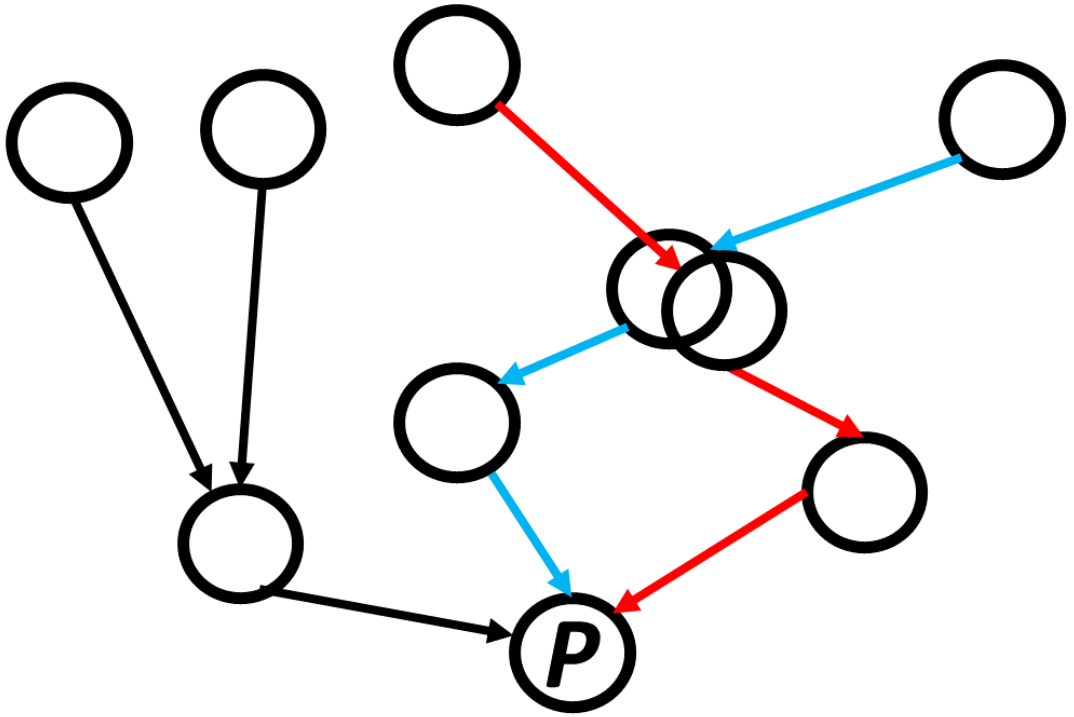
Efficient Topology Discovery



Sometimes our untrusted input data violates DBR, and isn't a tree.

If routing is consistent given prior hop: split violating nodes.

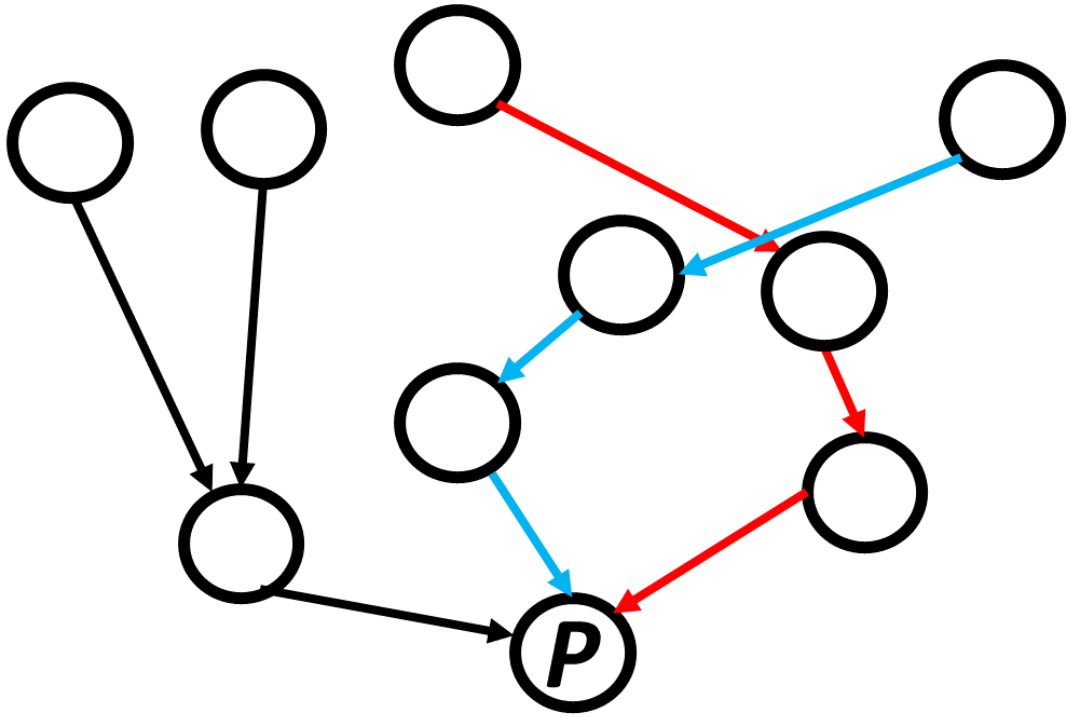
Efficient Topology Discovery



Sometimes our untrusted input data violates DBR, and isn't a tree.

If routing is consistent given prior hop: split violating nodes.

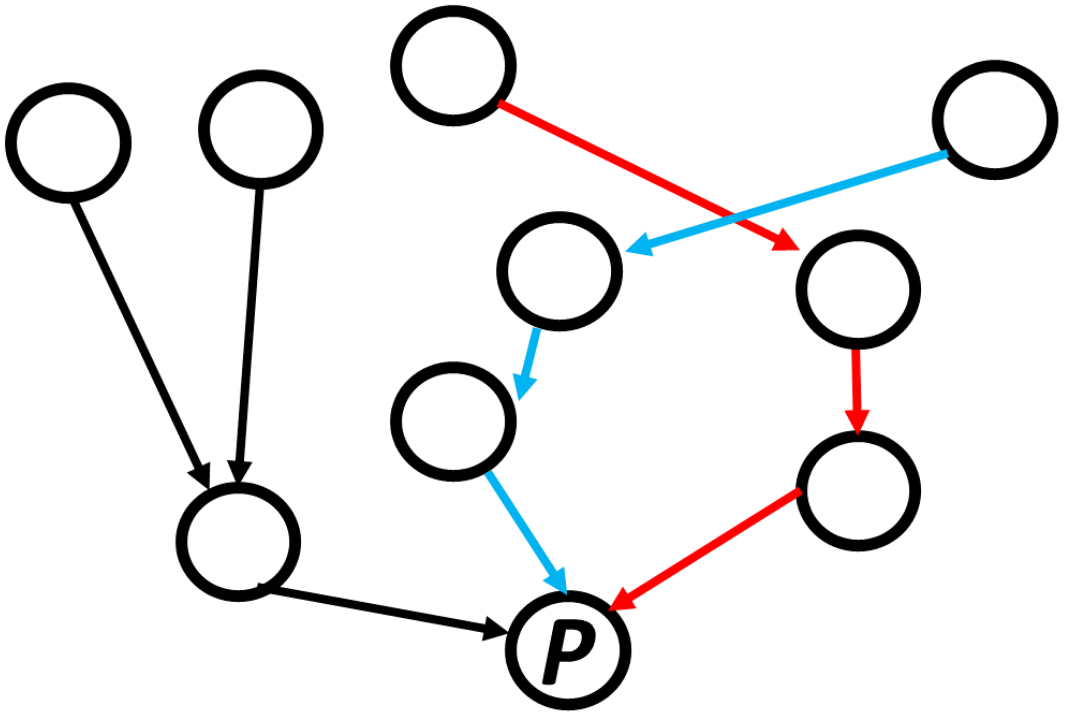
Efficient Topology Discovery



Sometimes our untrusted input data violates DBR, and isn't a tree.

If routing is consistent given prior hop: split violating nodes.

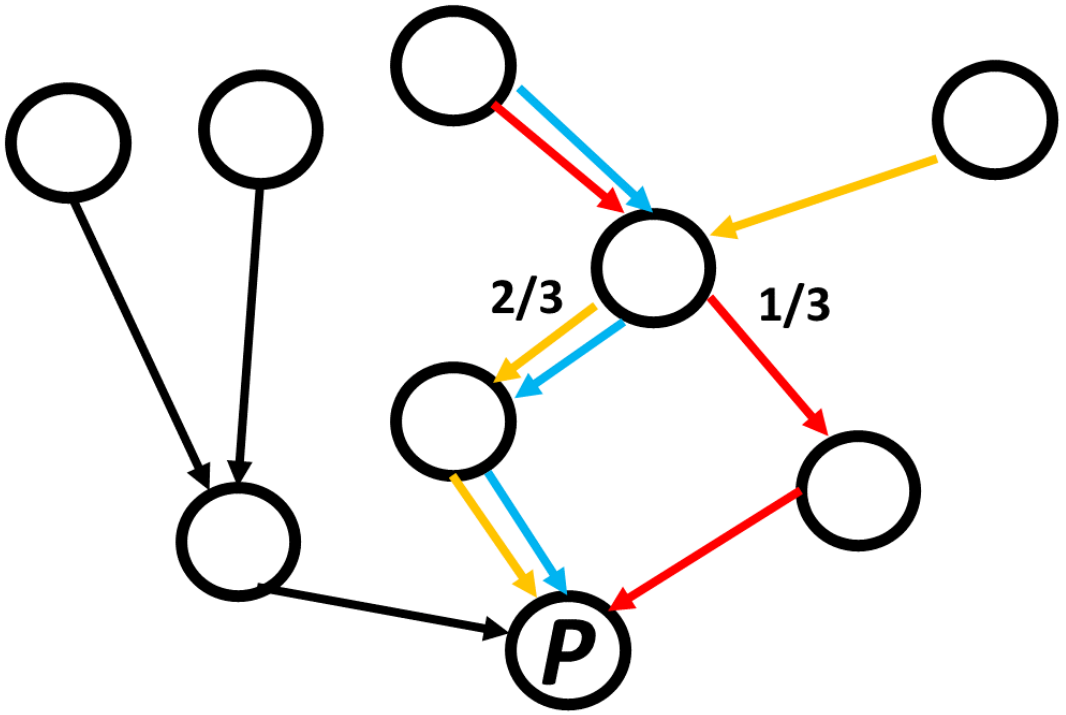
Efficient Topology Discovery



Sometimes our untrusted input data violates DBR, and isn't a tree.

If routing is consistent given prior hop: split violating nodes.

Efficient Topology Discovery

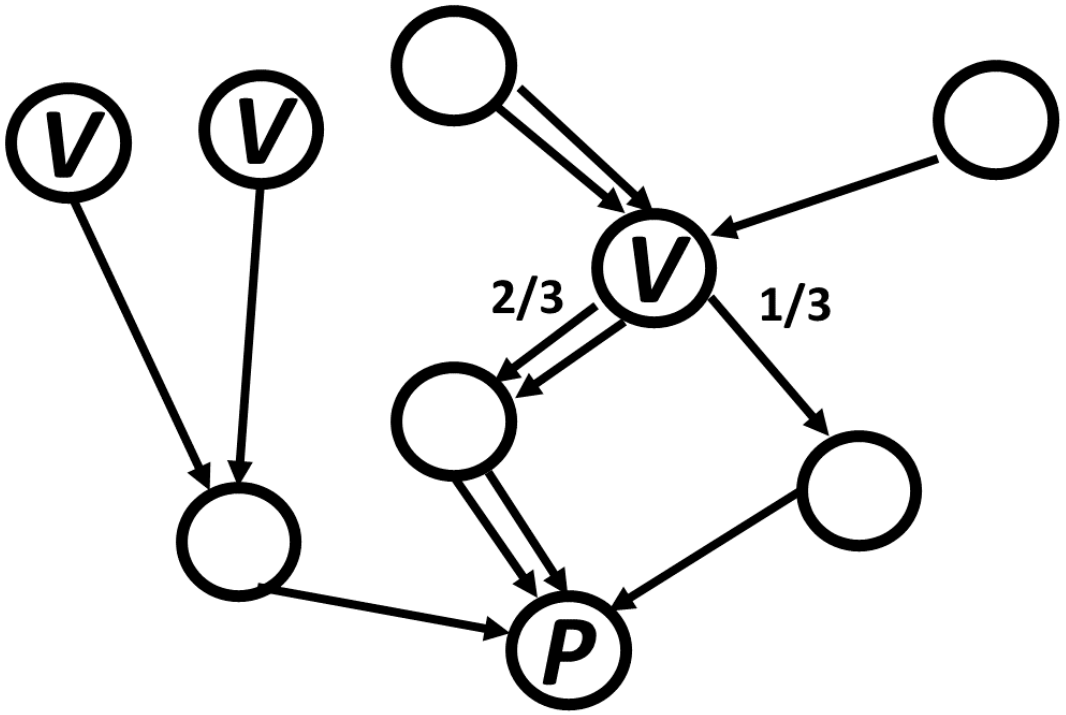


Sometimes our untrusted input data violates DBR, and isn't a tree.

If routing is consistent given prior hop: split violating nodes.

If it isn't, assume each violating node chooses outgoing link randomly, weighted by frequency.

Efficient Topology Discovery

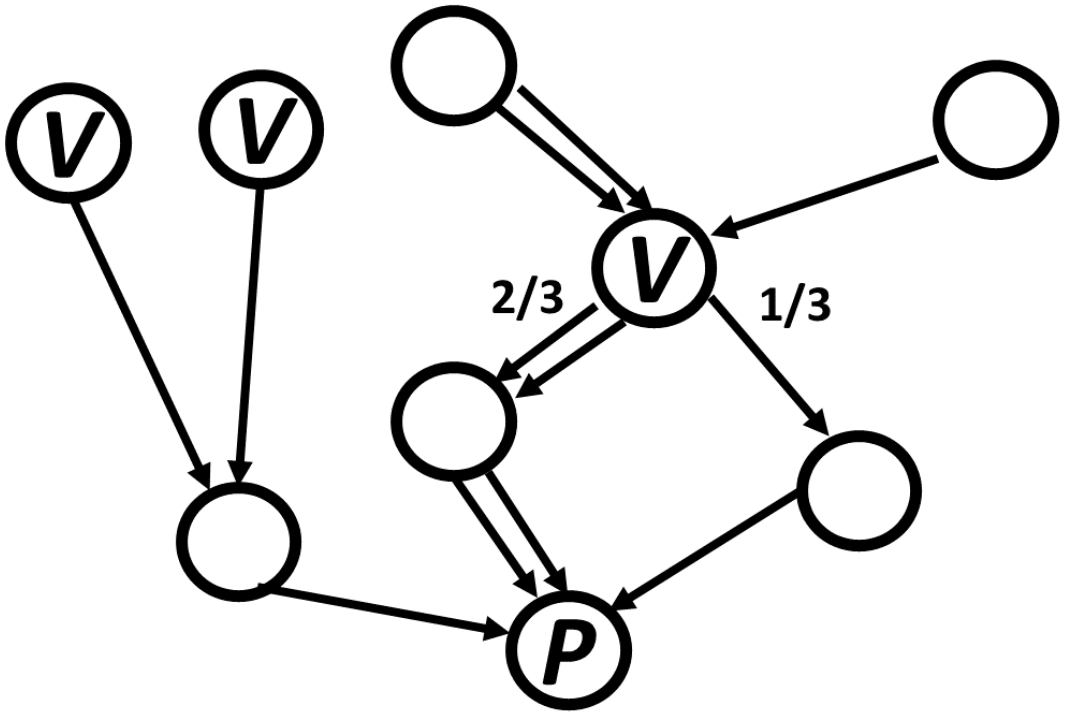


Sometimes our untrusted input data violates DBR, and isn't a tree.

If routing is consistent given prior hop: split violating nodes.

If it isn't, assume each violating node chooses outgoing link randomly, weighted by frequency.

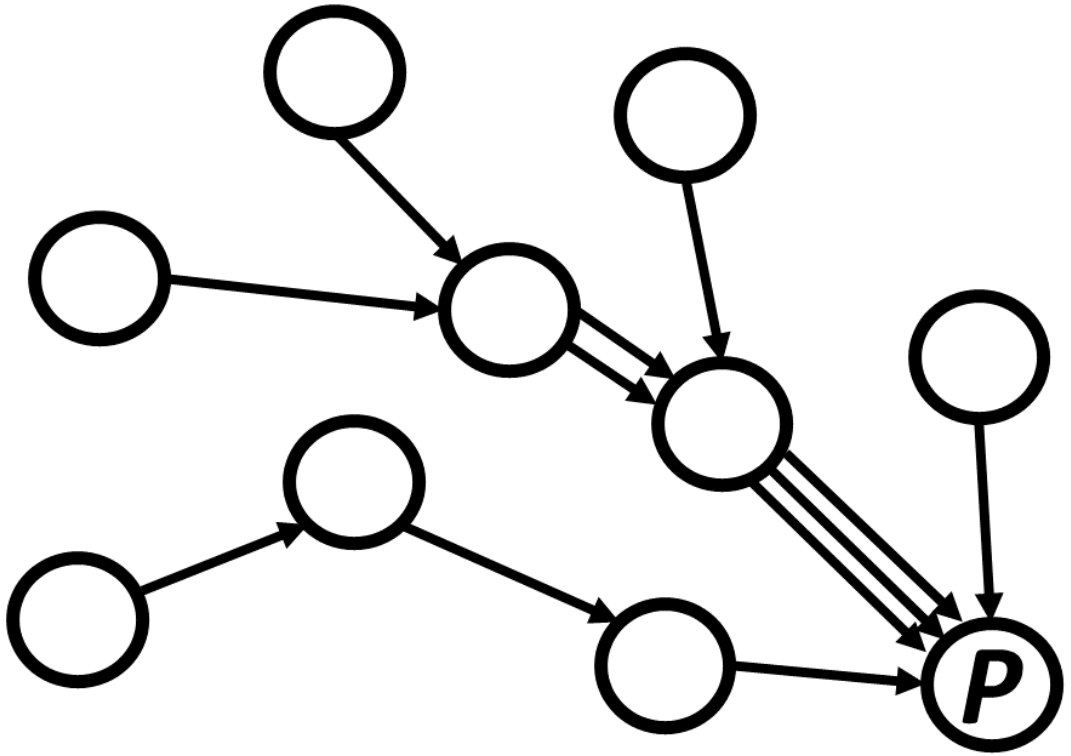
Efficient Topology Discovery



Sometimes our untrusted input data violates DBR, and isn't a tree.

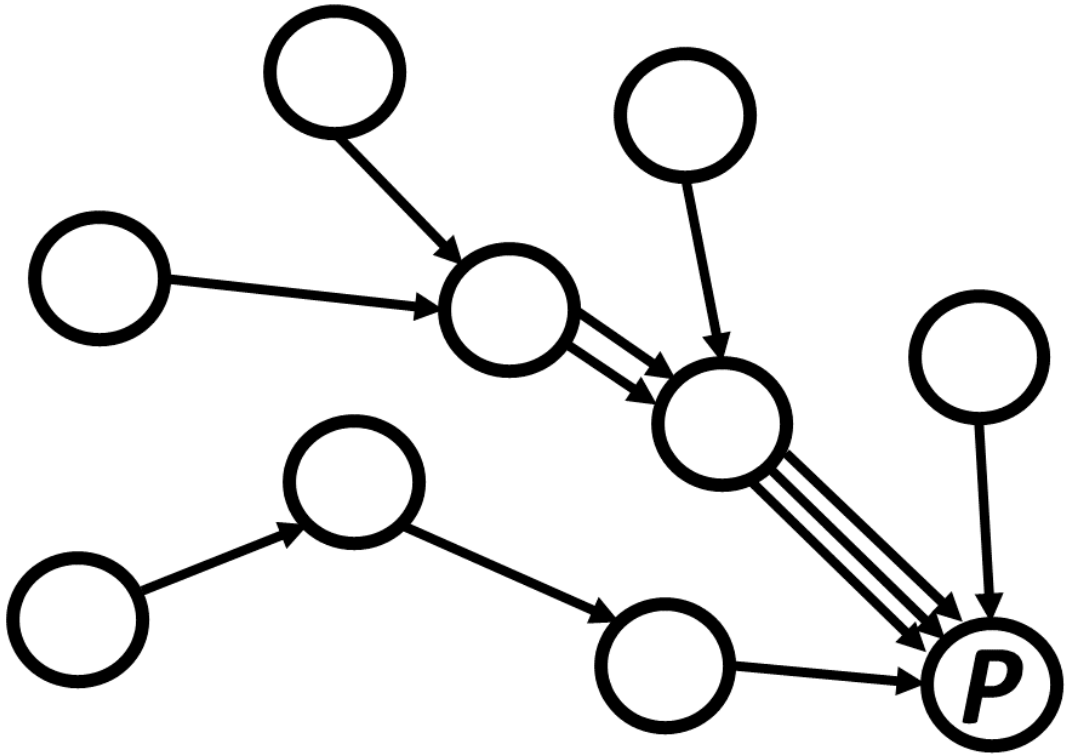
In this random routing case, a generalized greedy algorithm chooses VP with max expected coverage and gets $(e/(e-1))^2$ appx.

Path Prediction



We run traceroutes from each VP in our chosen set to *P*. Now we have a big collection of paths.

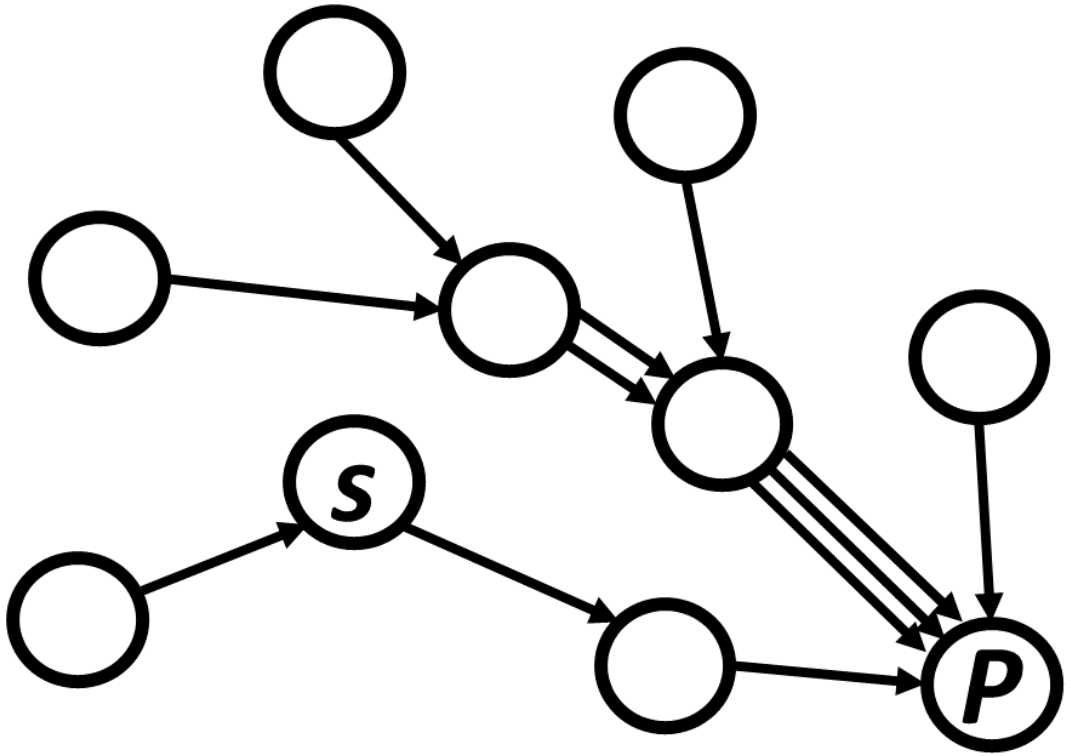
Path Prediction



We run traceroutes from each VP in our chosen set to *P*. Now we have a big collection of paths.

Merge all these paths together into *D*, a directed acyclic graph → *P*.

Path Prediction

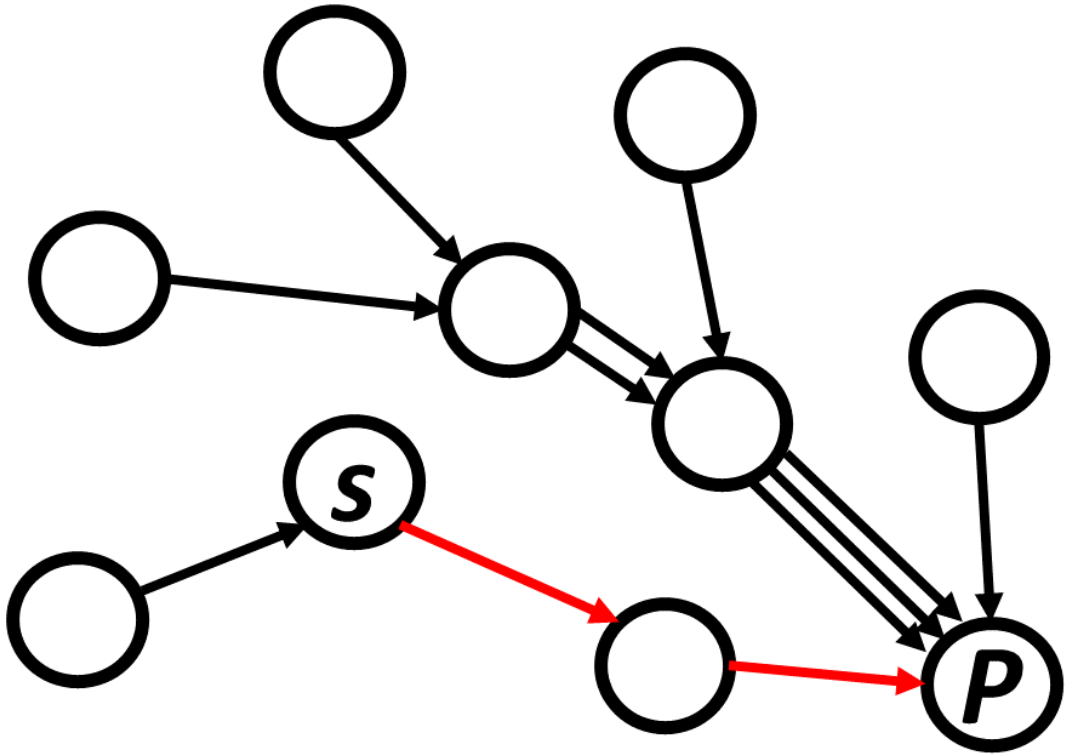


We run traceroutes from each VP in our chosen set to *P*. Now we have a big collection of paths.

Merge all these paths together into *D*, a directed acyclic graph → *P*.

If *D* is an in-tree rooted at *P*, path prediction from node s is easy: Return the only path from s to *P*.

Path Prediction

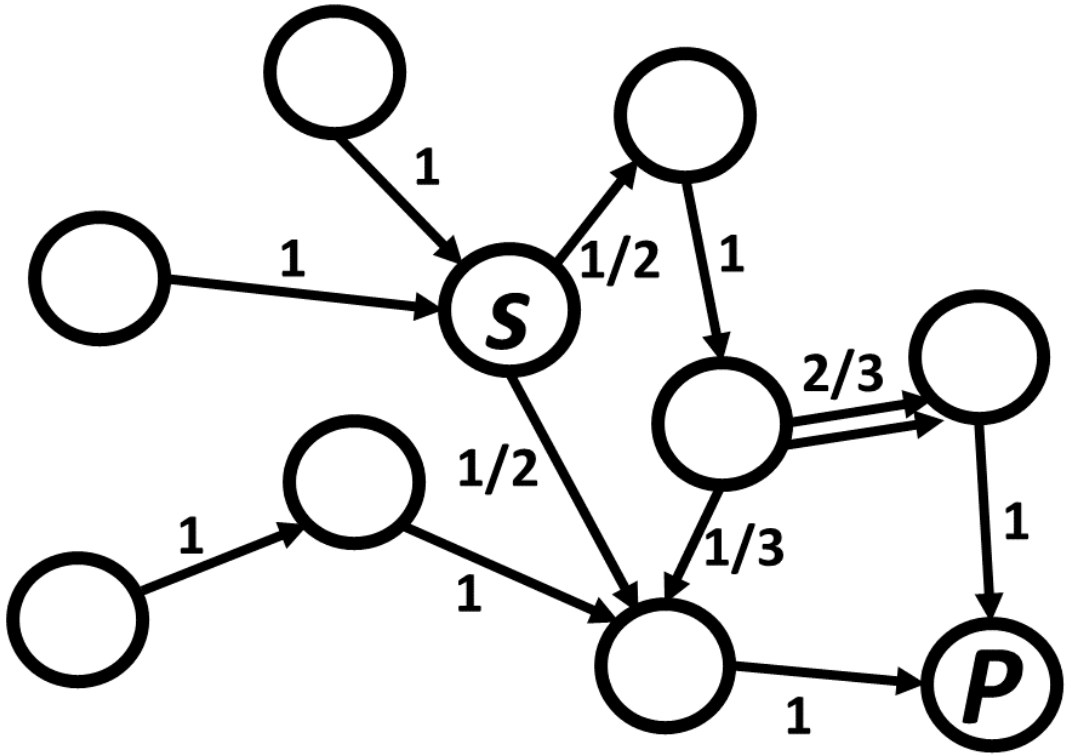


We run traceroutes from each VP in our chosen set to *P*. Now we have a big collection of paths.

Merge all these paths together into *D*, a directed acyclic graph → *P*.

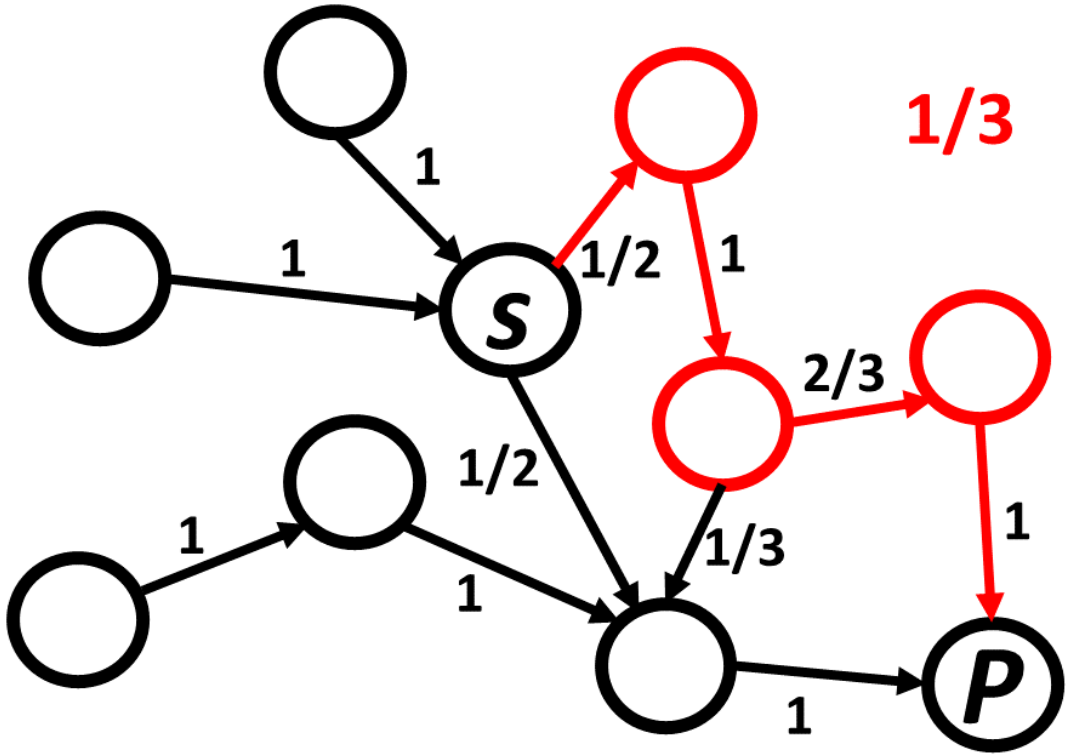
If *D* is an in-tree rooted at *P*, path prediction from node s is easy: Return the only path from s to *P*.

Path Prediction



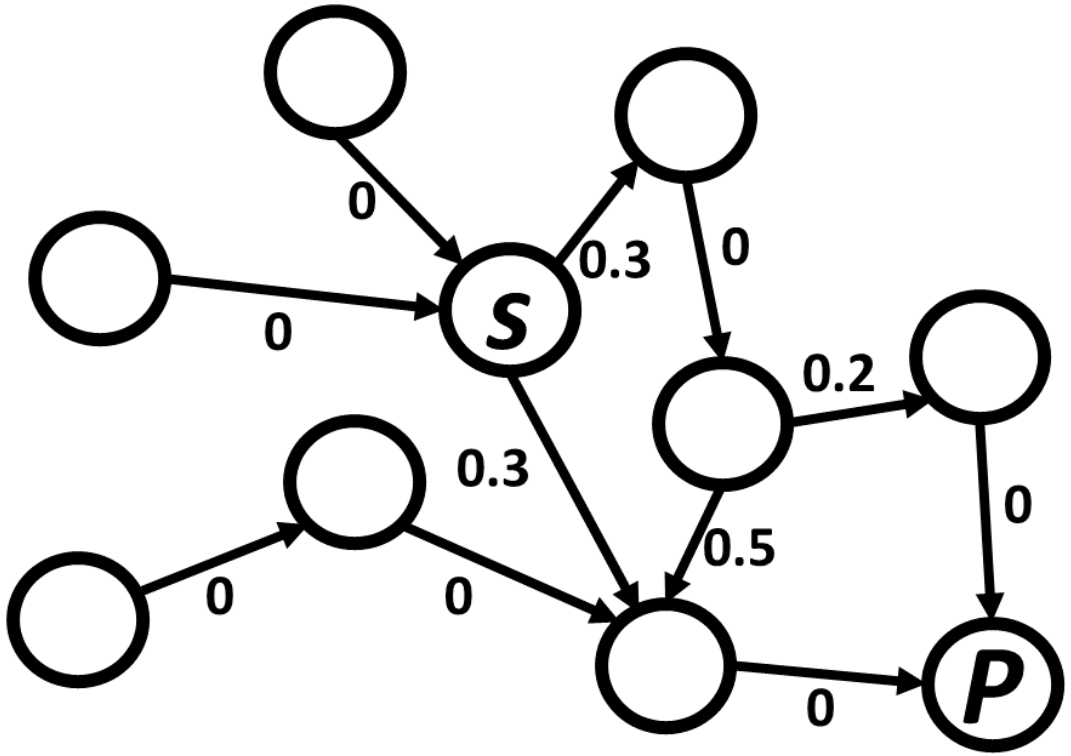
But *D* is basically never an in-tree, it turns out. So we model each node as routing randomly like before. *D* encodes a Markov chain.

Path Prediction



But *D* is basically never an in-tree, it turns out. So we model each node as routing randomly like before. *D* encodes a Markov chain.

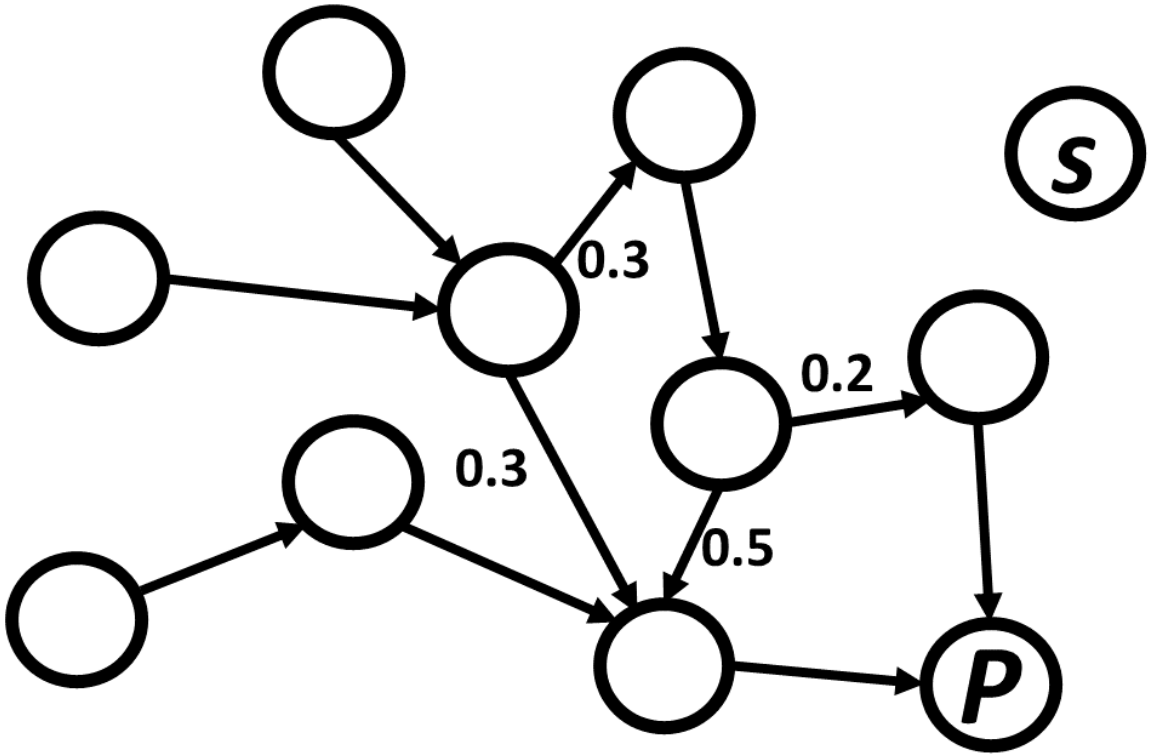
Path Prediction



But *D* is basically never an in-tree, it turns out. So we model each node as routing randomly like before. *D* encodes a Markov chain.

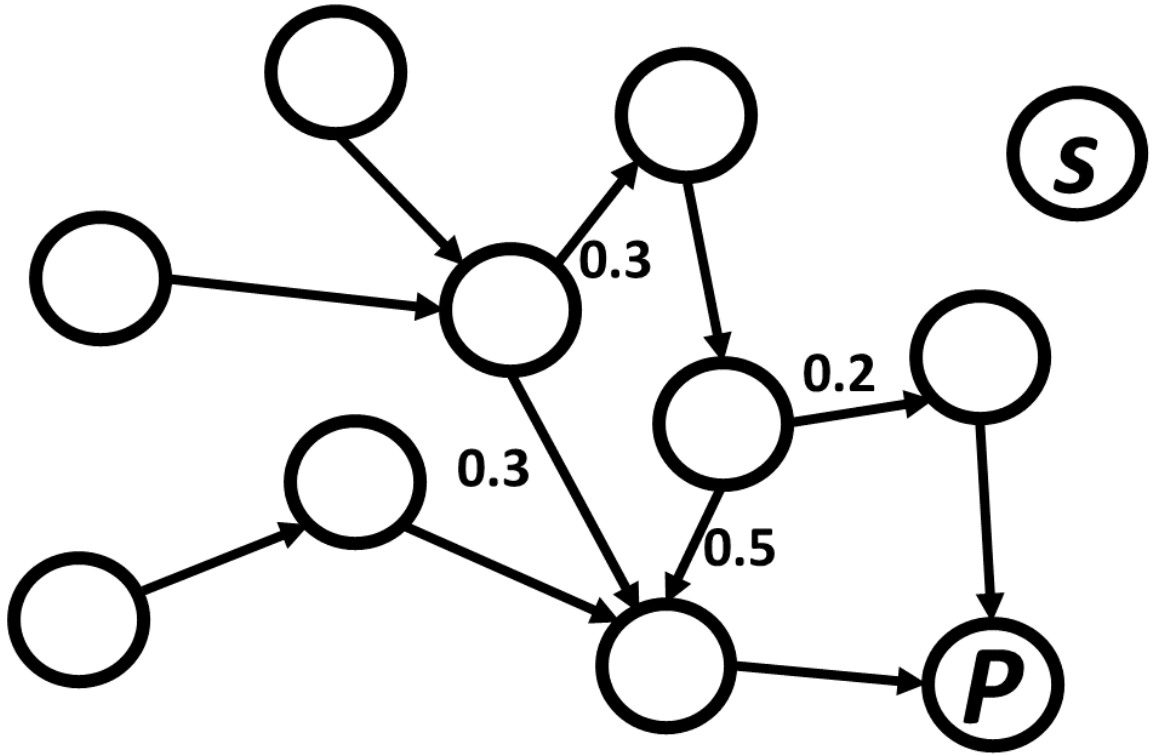
Given a query source *s*, we can efficiently predict the most likely p paths from *s* to *P* by taking the –log of each edge probability and using Yen's p-shortest paths algorithm.

Path Prediction



D doesn’t contain every AS node in the Internet.

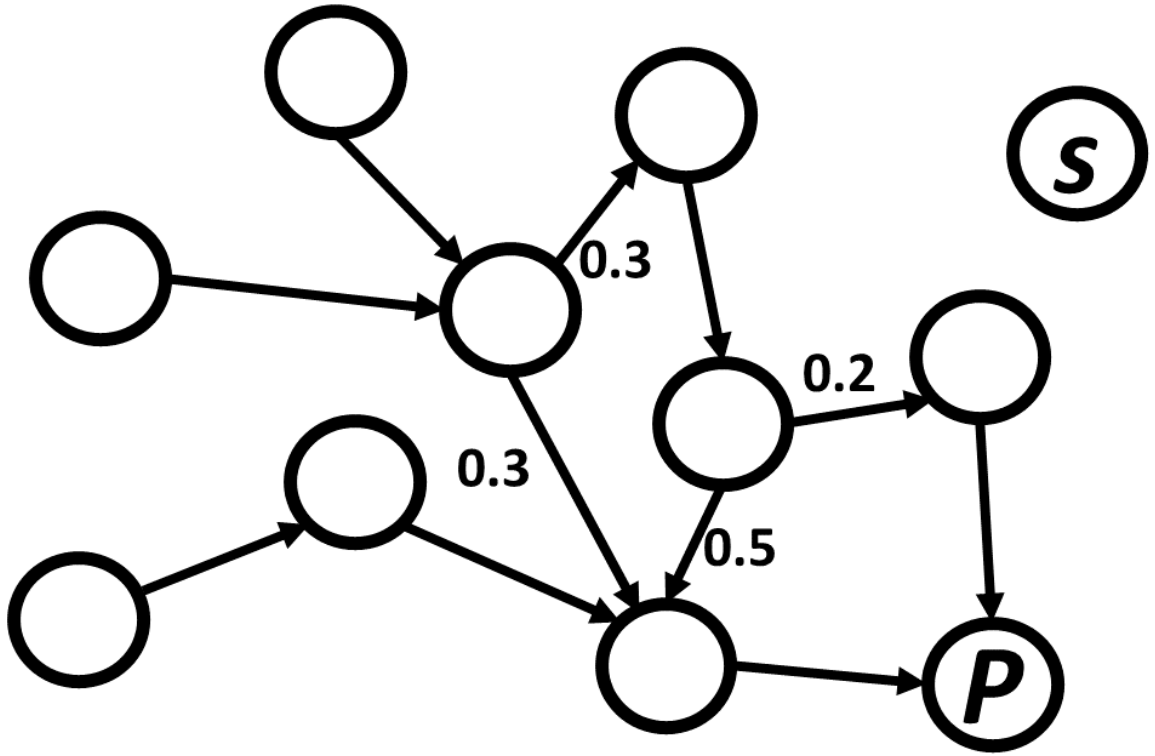
Path Prediction



D doesn't contain every AS node in the Internet.

We may be asked to predict a path from some disconnected node *s*.

Path Prediction

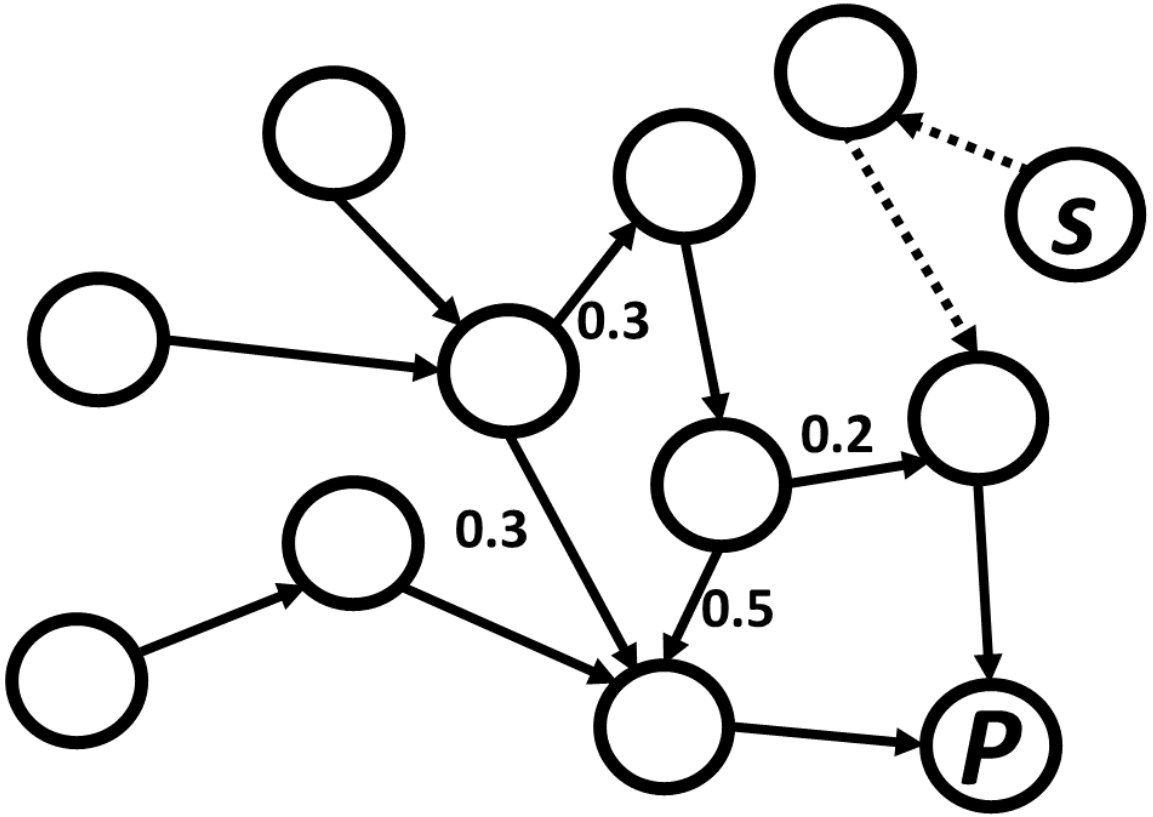


D doesn't contain every AS node in the Internet.

We may be asked to predict a path from some disconnected node *s*.

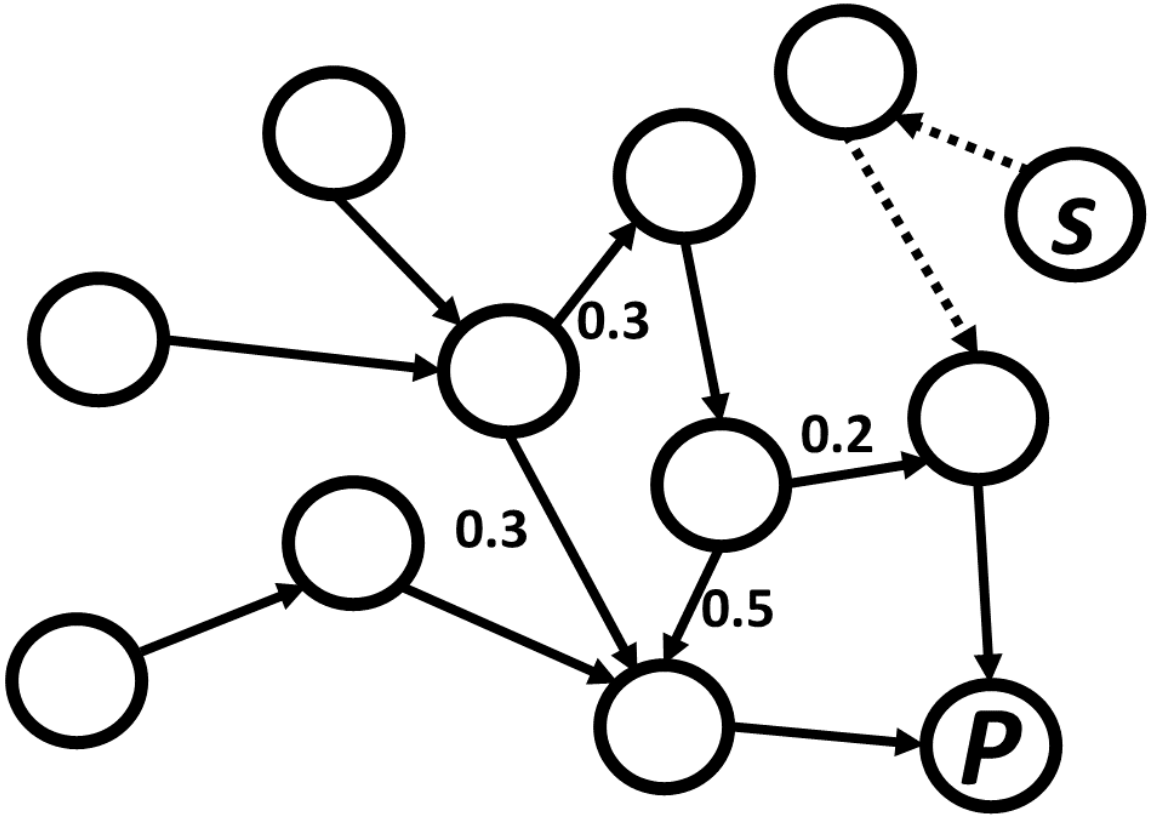
How can we predict paths from *s*?

Path Prediction



We use a system called BGPSim which uses public BGP data to guess the path from *s* to *P*.

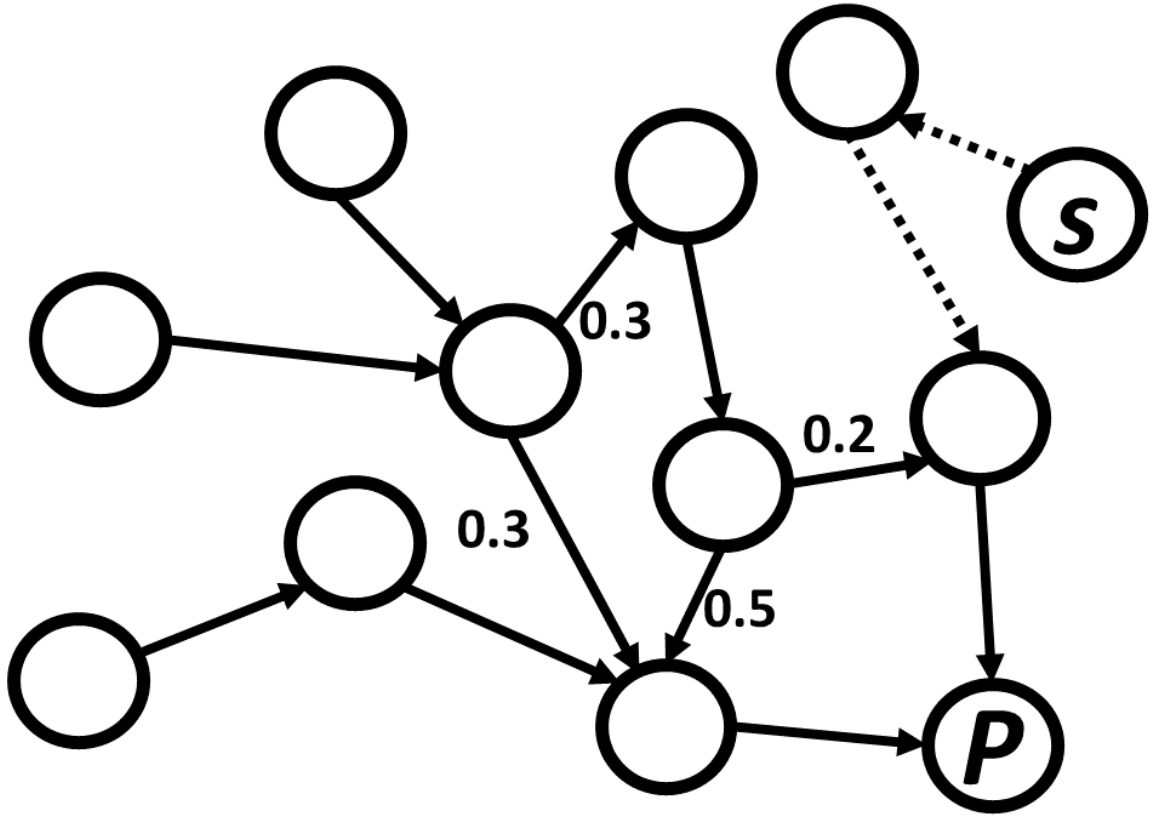
Path Prediction



We use a system called BGPSim which uses public BGP data to guess the path from *s* to *P*.

BGPSim always returns a path, but it might not be totally accurate.

Path Prediction



We use a system called BGPSim which uses public BGP data to guess the path from *s* to *P*.

BGPSim always returns a path, but it might not be totally accurate.

We follow the BGPSim path until we get to *D*, then predict from *D*.

PathCache’s Performance

- 75% of PathCache’s predicted paths err on at most 1 edge.

PathCache’s Performance

- 75% of PathCache’s predicted paths err on at most 1 edge.
- We discover 4x more network topology than the state of the art

PathCache's Performance

- 75% of PathCache's predicted paths err on at most 1 edge.
- We discover 4x more network topology than the state of the art
- By path splicing, PathCache can respond to 100% of path queries

Temporal Graph Streaming

AND APPLICATIONS TO DISEASE TRACKING

Joint work with:

Andrew McGregor

Cameron Musco

Time-Dependent Graph Streams

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

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?

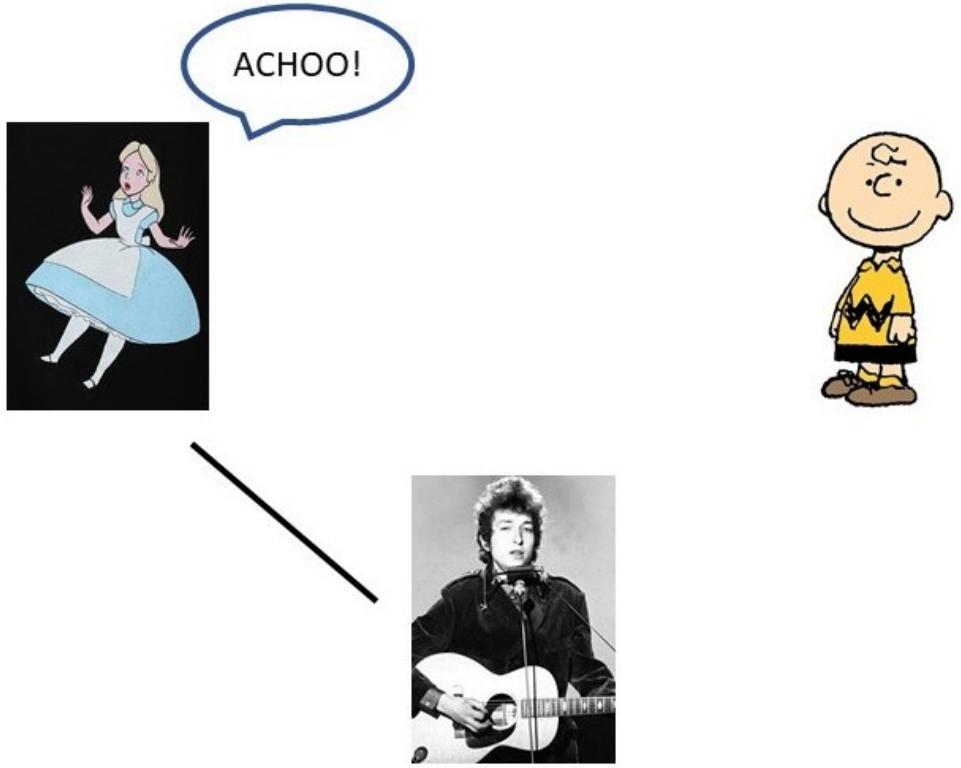
Time-Dependent Graph Streams

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

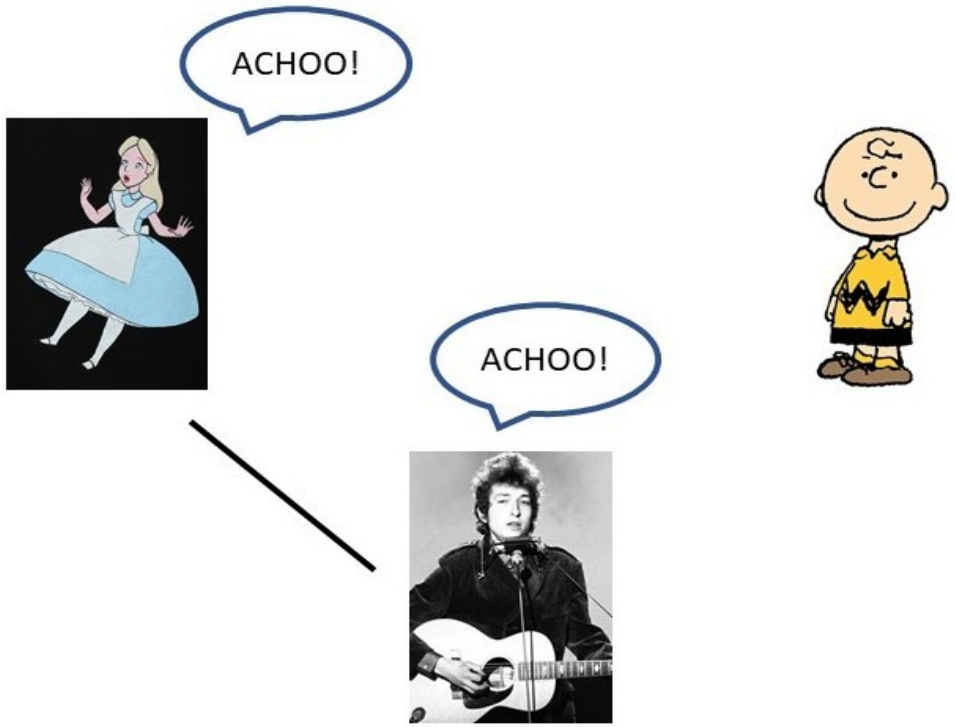


In the typical streaming model, graph is the same **regardless of the order edges appear in the stream.**

What if the order mattered?

Ex: disease spreading

Time-Dependent Graph Streams

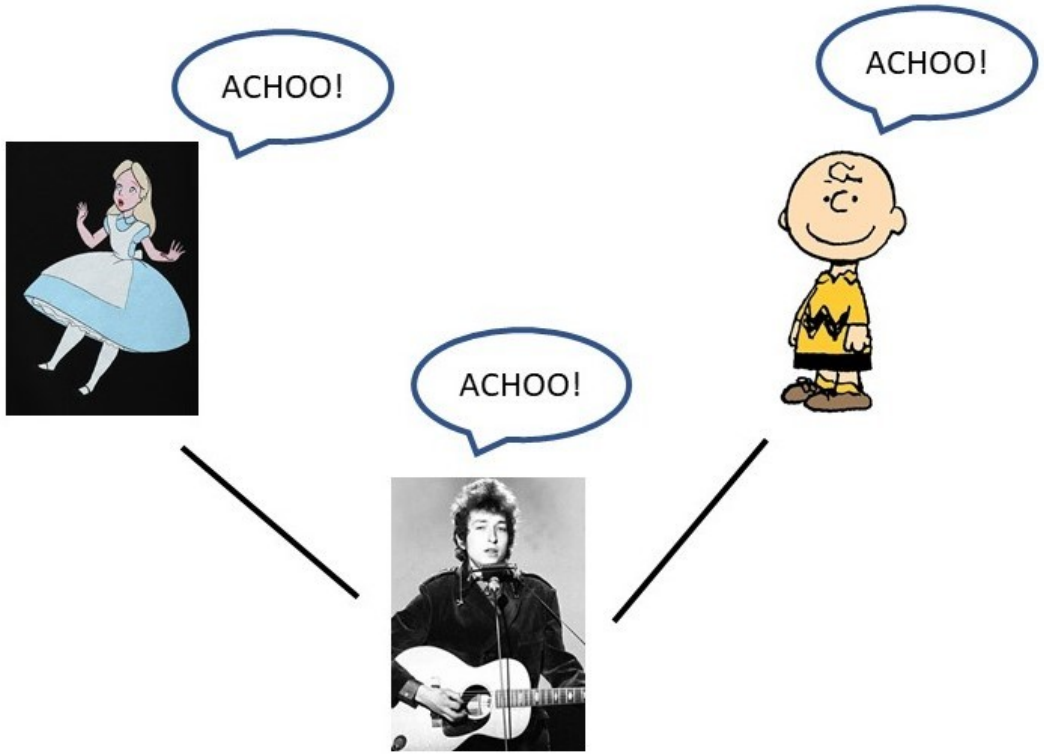


In the typical streaming model, graph is the same **regardless of the order edges appear in the stream.**

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

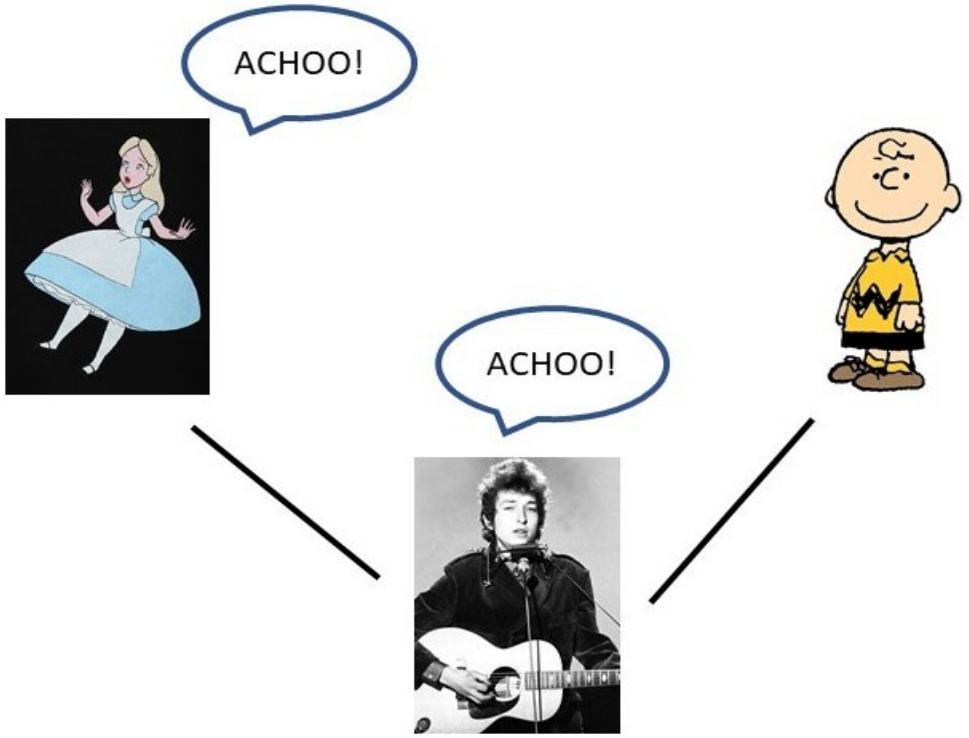
Time-Dependent Graph Streams

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



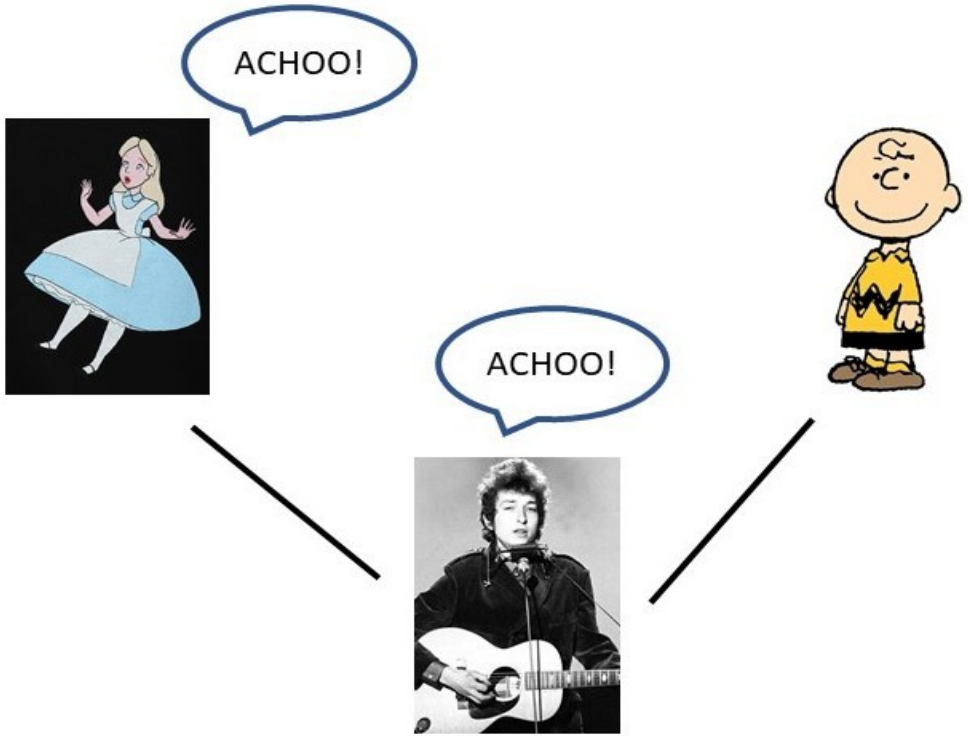
In the typical streaming model, graph is the same **regardless of the order edges appear in the stream.**

What if the order mattered?

Ex: disease spreading

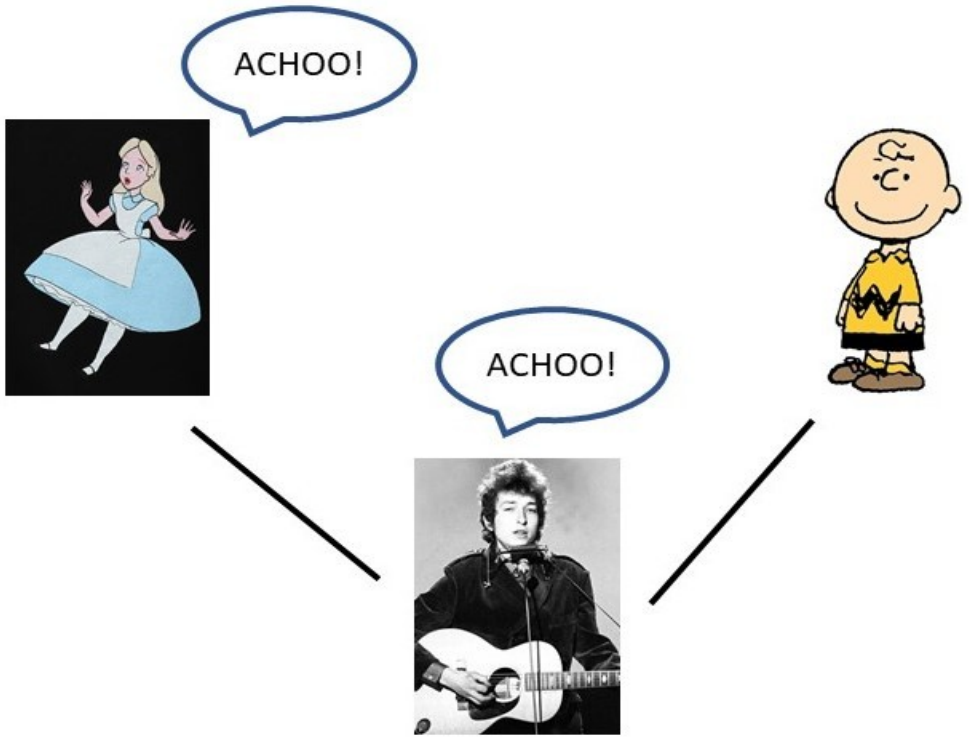
Temporal graphs are just beginning to be investigated. No streaming work.

Time-Dependent Graph Streams



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

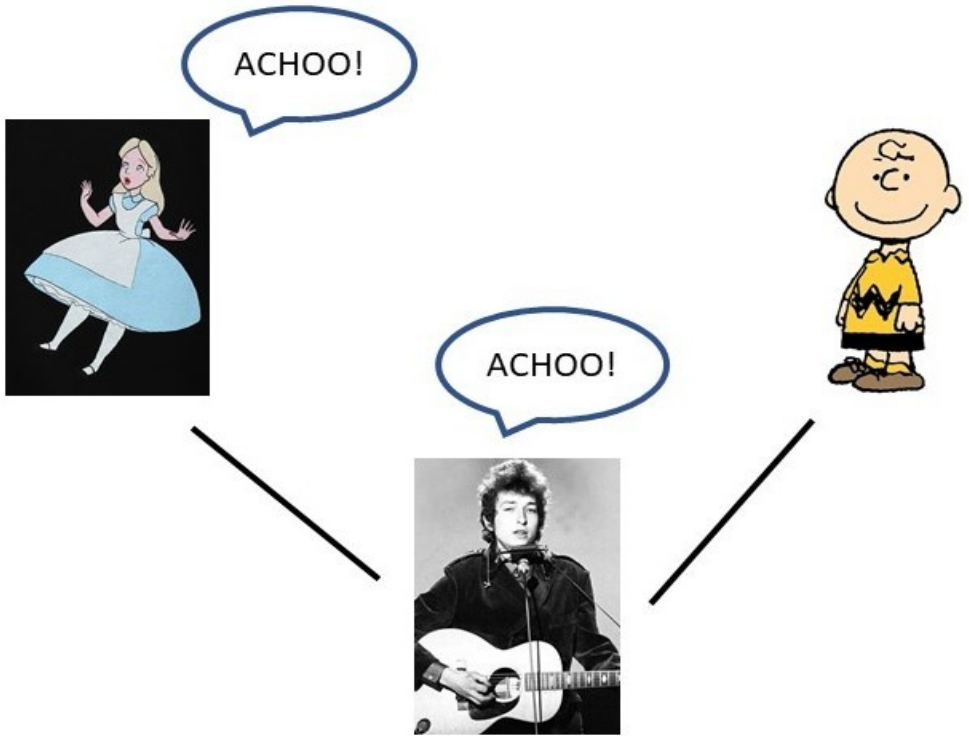
Time-Dependent Graph Streams



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

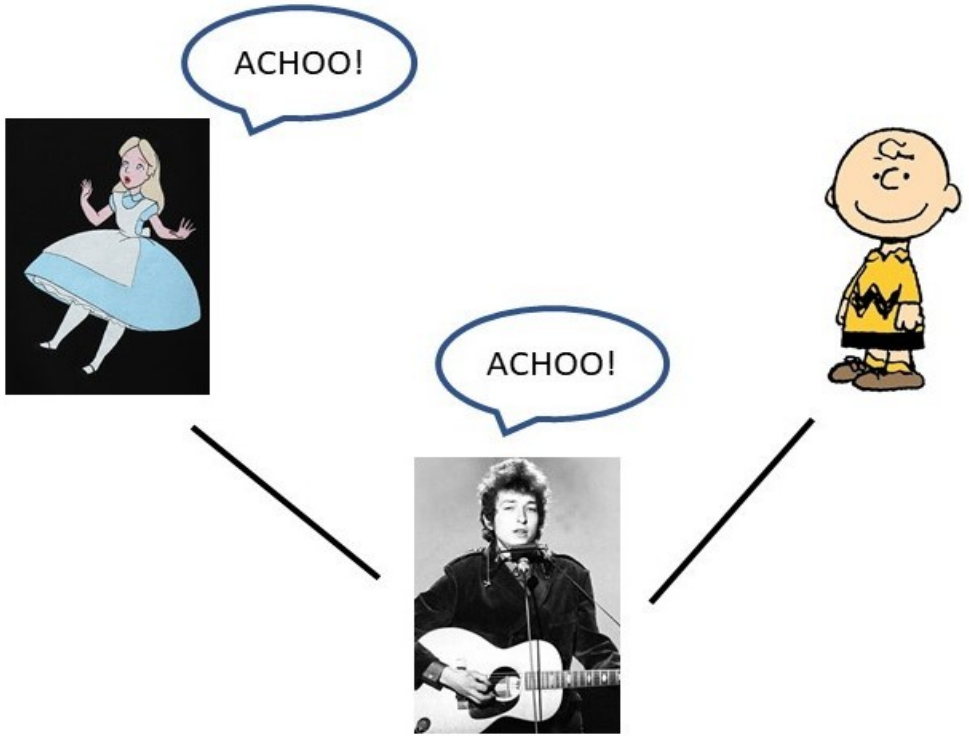


Imagine we receive a massive stream of handshakes between many people. Later, we learn one of those people was sick.

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

If the edges appear “out of order” in the stream, NO.

Time-Dependent Graph Streams

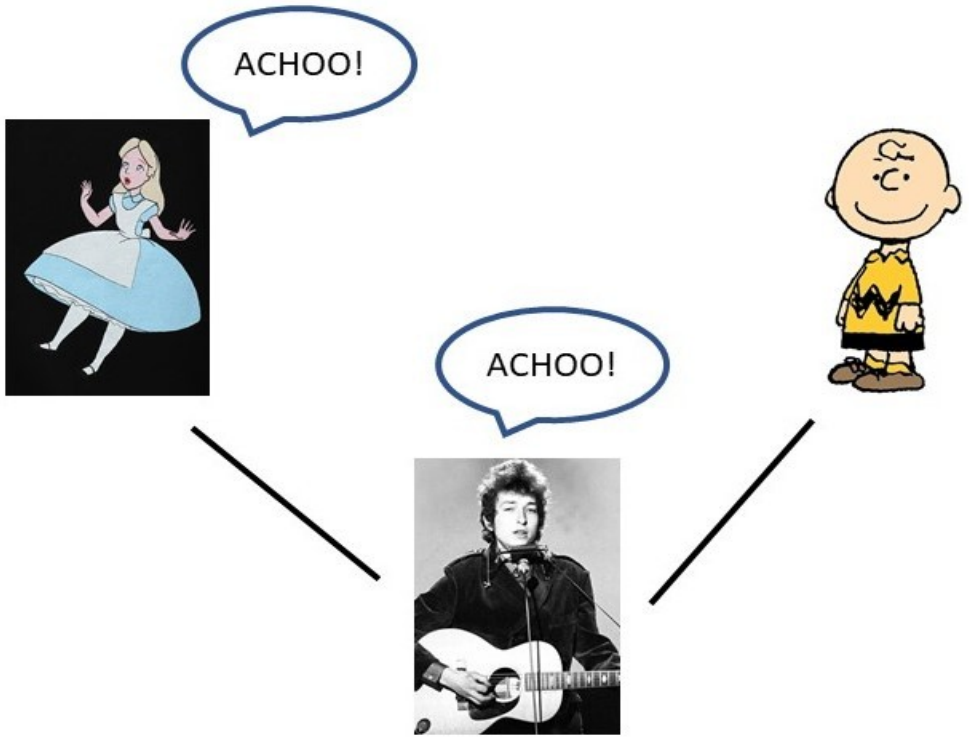


Imagine we receive a massive stream of handshakes between many people. Later, we learn one of those people was sick.

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

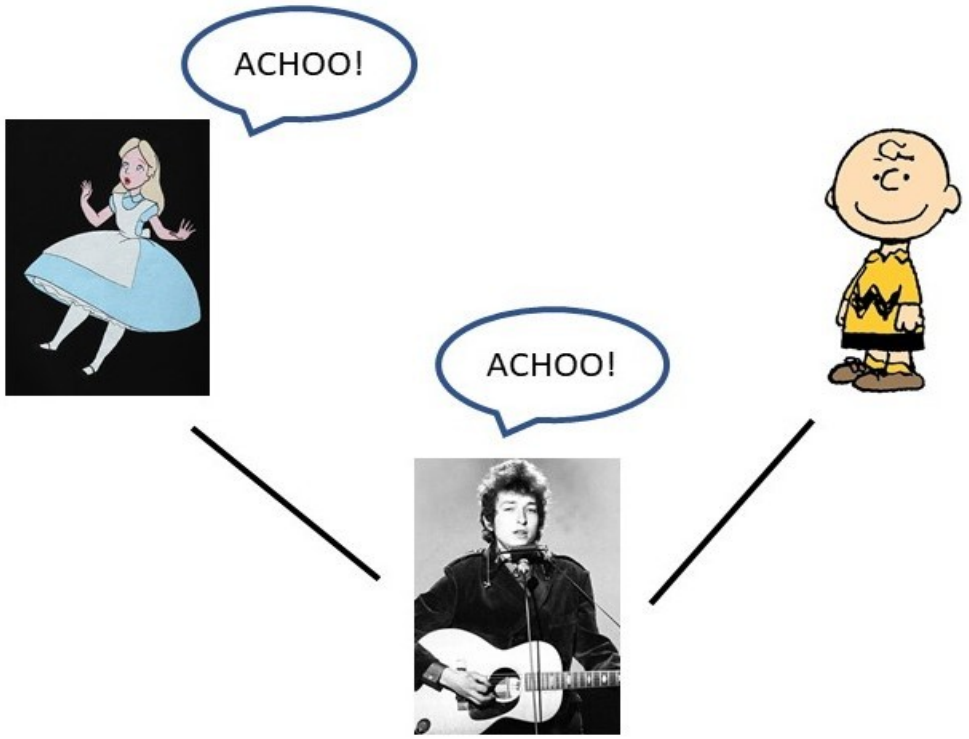
However, if the edges appear in order we can solve some similar problems.

Time-Dependent Graph Streams



We can determine the “susceptibility” of a particular person to infection – estimate the number of people who, if sick, would indirectly infect them.

Time-Dependent Graph Streams



We can determine the “susceptibility” of a particular person to infection – estimate the number of people who, if sick, would indirectly infect them.

We can also randomly sample from the set of potential infectors.

Future Research Directions

External Memory Algorithms

- Disk latency is decreasing. Disk as resource for algs?

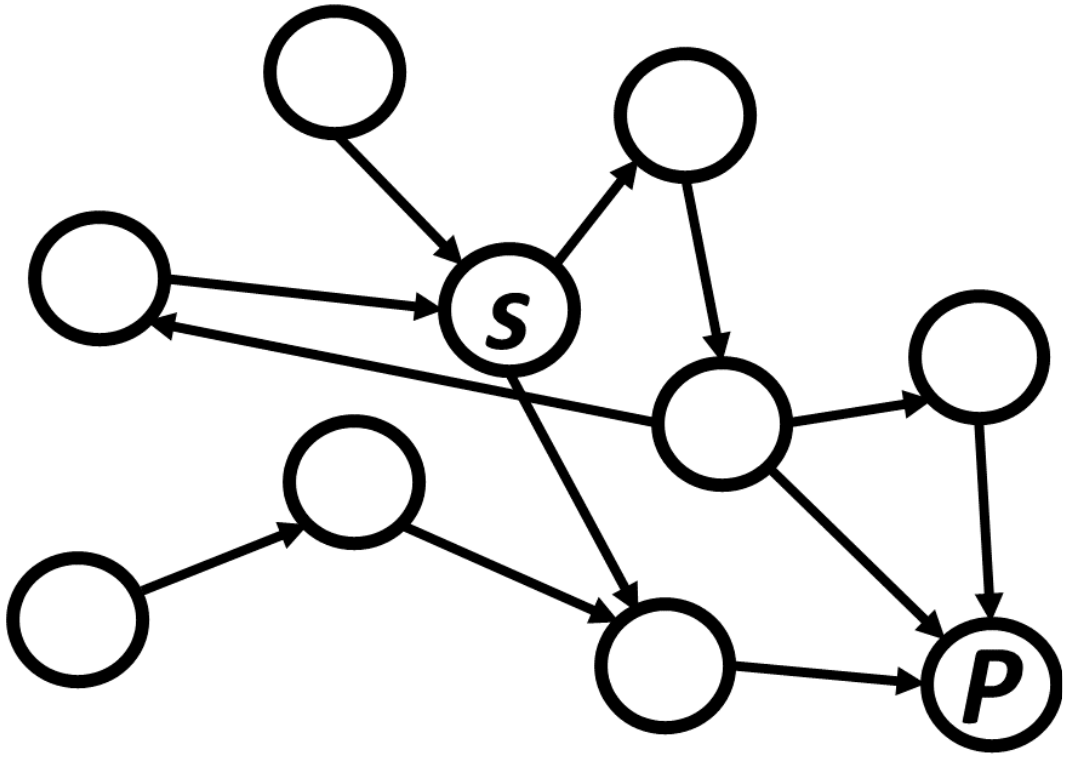
External Memory Algorithms

- Disk latency is decreasing. Disk as resource for algs?
- Random I/O bandwith in RAM comparable to sequential I/O in NVMe SSDs.

External Memory Algorithms

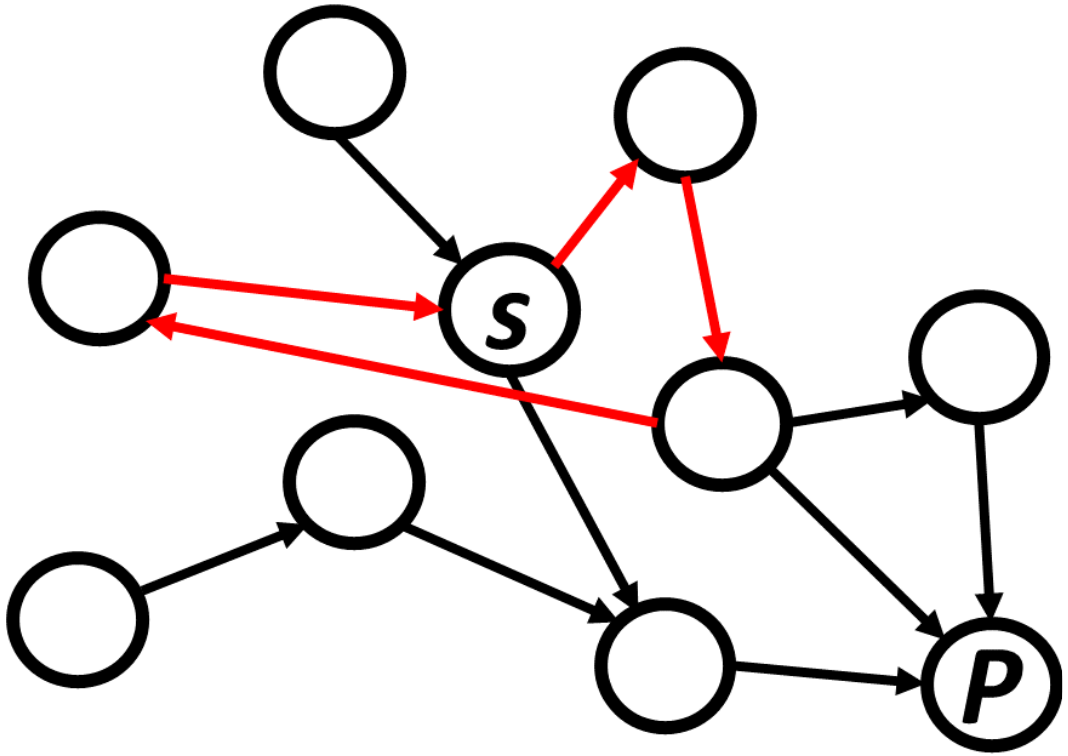
- Disk latency is decreasing. Disk as resource for algs?
- Random I/O bandwith in RAM comparable to sequential I/O in NVMe SSDs.
- Can techniques from streaming help utilize large sequentially-accessible working memory to design new data structures?

Cycles in PathCache “DAGs”



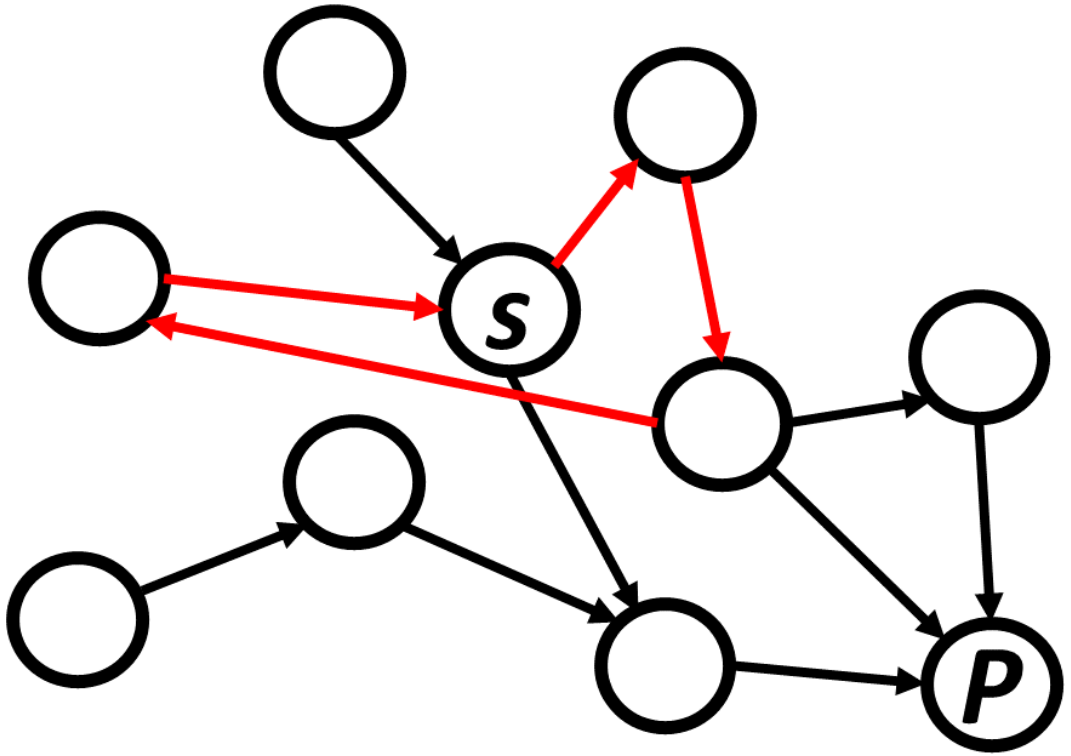
Some of our path prediction DAGs were not actually DAGs – they contained cycles made up of different traceroutes!

Cycles in PathCache “DAGs”



Some of our path prediction DAGs were not actually DAGs – they contained cycles made up of different traceroutes!

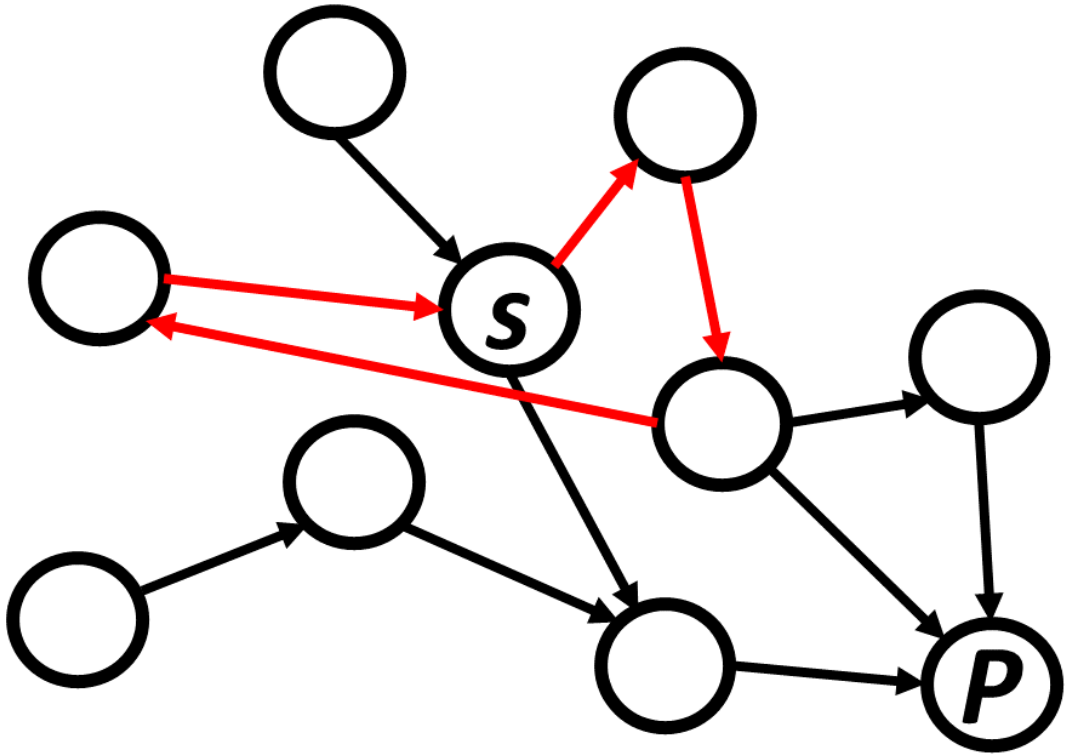
Cycles in PathCache “DAGs”



Some of our path prediction DAGs were not actually DAGs – they contained cycles made up of different traceroutes!

Very unexpected violation of destination-based routing.

Cycles in PathCache “DAGs”

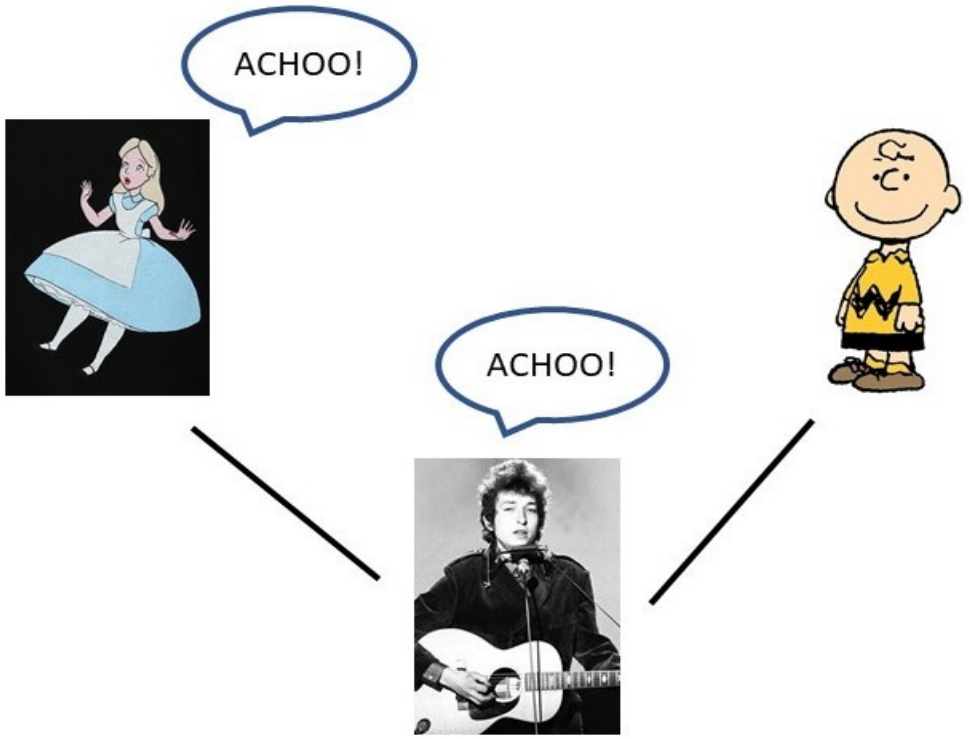


Some of our path prediction DAGs were not actually DAGs – they contained cycles made up of different traceroutes!

Very unexpected violation of destination-based routing.

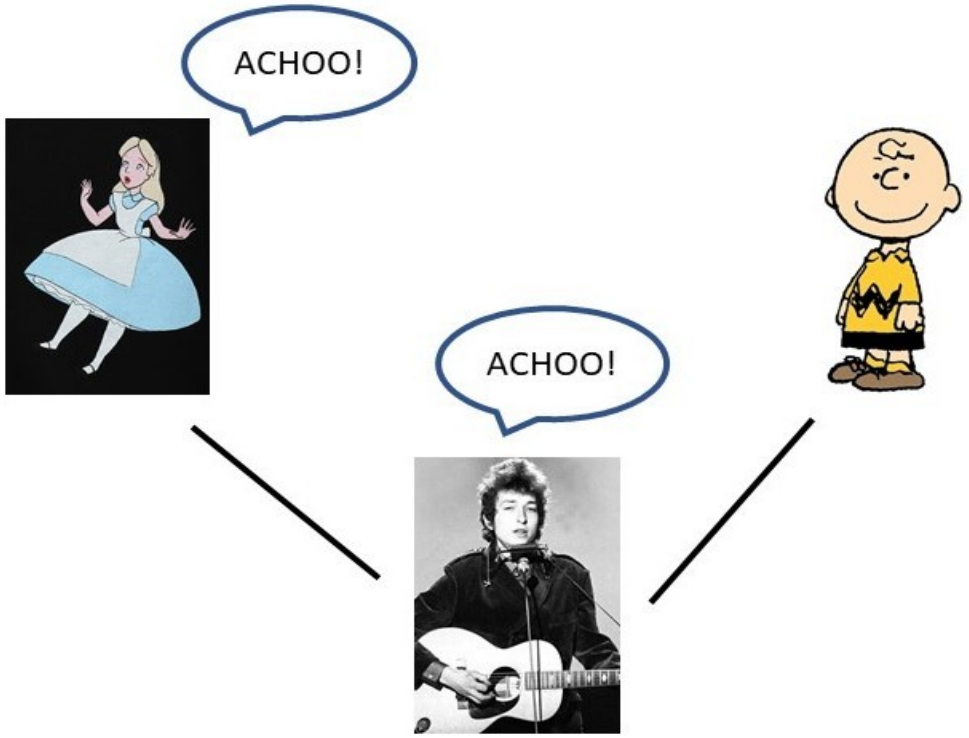
Studying this could reveal new perspectives on Internet routing!

Temporal Graph Streaming



Temporal graphs are a relatively new area, and no one else has studied them in the streaming setting.

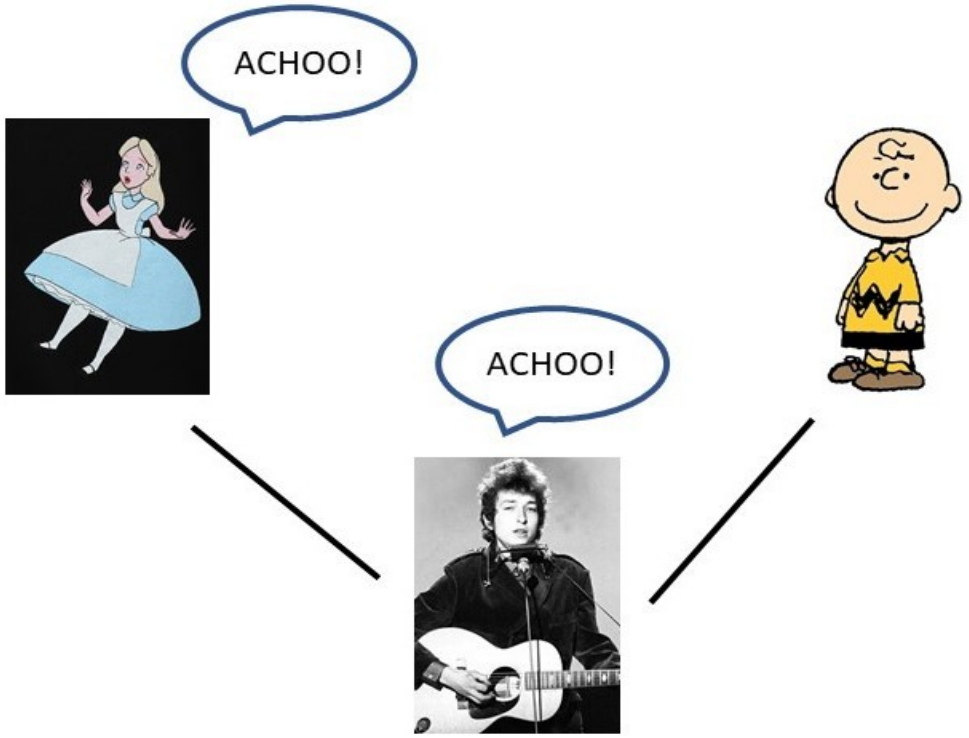
Temporal Graph Streaming



Temporal graphs are a relatively new area, and no one else has studied them in the streaming setting.

Can we fingerprint temporal graph streams?

Temporal Graph Streaming

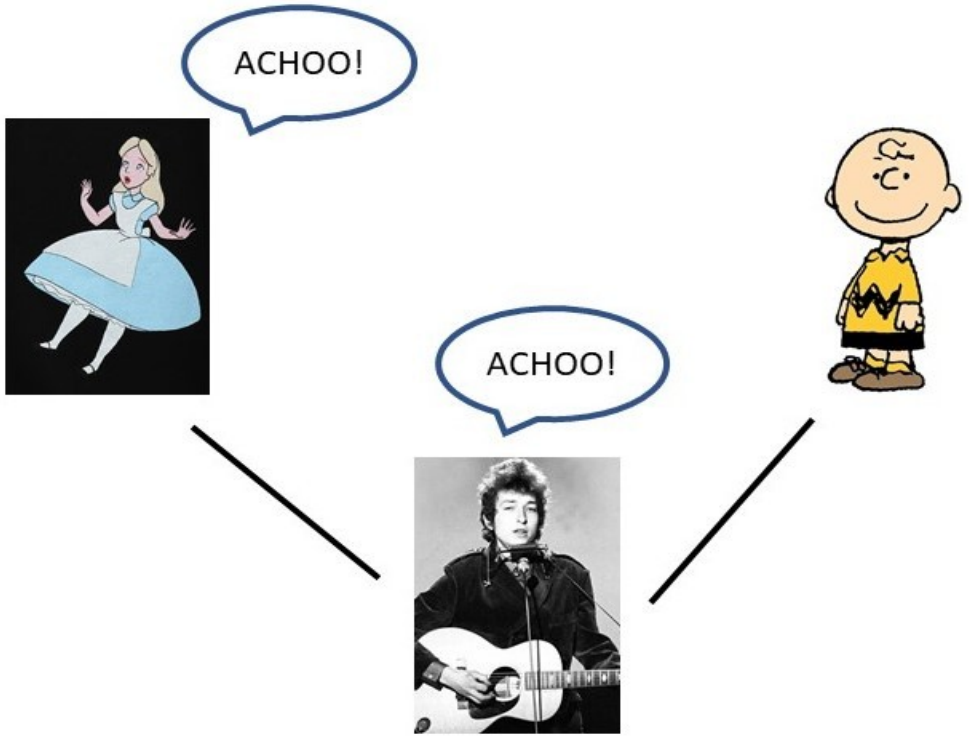


Temporal graphs are a relatively new area, and no one else has studied them in the streaming setting.

Can we fingerprint temporal graph streams?

Can we compute temporal tours?

Temporal Graph Streaming



Temporal graphs are a relatively new area, and no one else has studied them in the streaming setting.

Can we fingerprint temporal graph streams?

Can we compute temporal tours?

Can we sparsify temporal graphs while retaining reachability?

Acknowledgements

My committee members:
Phillipa Gill
Markos Katsoulakis
Cameron Musco

My collaborators, peers,
family, and friends

Acknowledgements

My committee members:
Phillipa Gill
Markos Katsoulakis
Cameron Musco

My collaborators, peers, family, and friends

My advisor:
Andrew McGregor

Acknowledgements

My committee members:
Phillipa Gill
Markos Katsoulakis
Cameron Musco

My collaborators, peers, family, and friends

My advisor:
Andrew McGregor

Thanks for listening!