Parameterized Streaming Algorithms for Matching and Covering

Graham Cormode

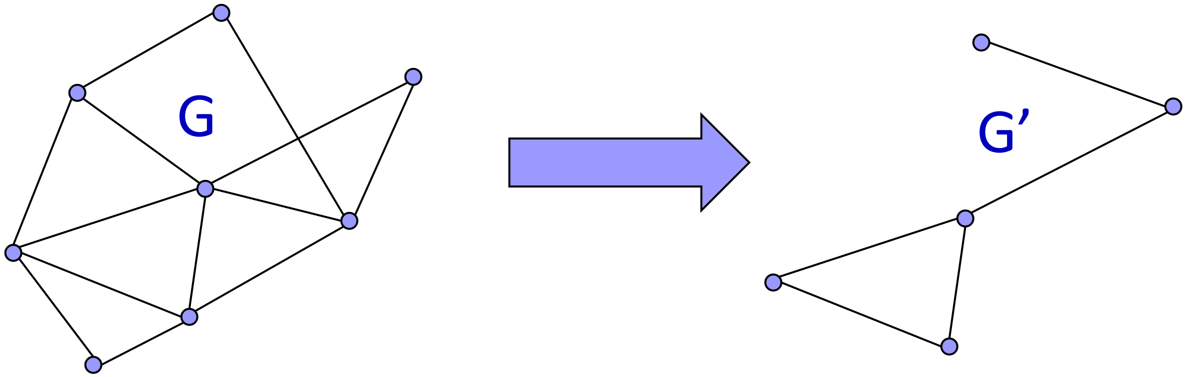
g.cormode@warwick.ac.uk

Joint work with

Rajesh Chitnis (UMD)

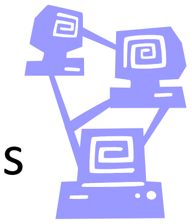
MohammadTaghi Hajiaghayi (UMD)

Morteza Monemizadeh (Frankfurt)

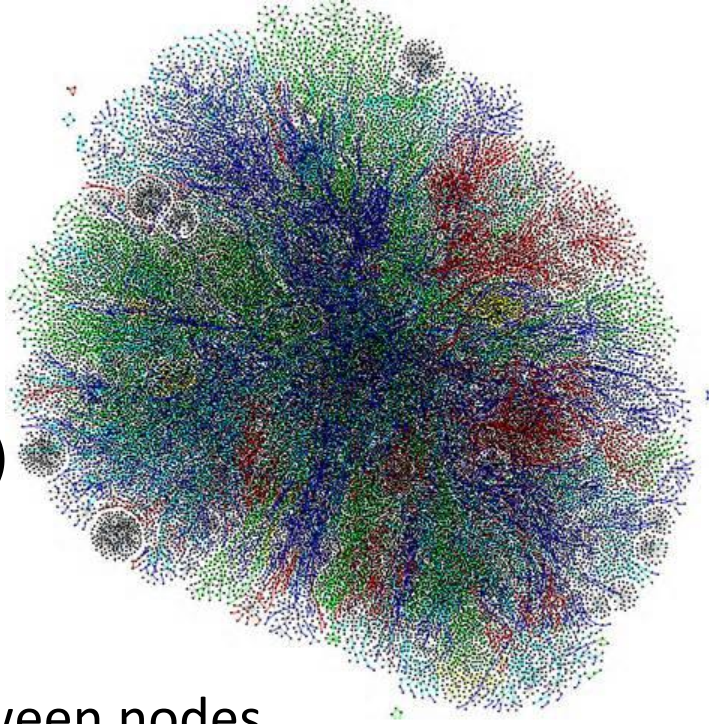


A tale of three graphs

- The telephone call-graph
  - Each edge denotes a call between two phones
  - 2-3 $\times$ 10^9 calls made each day in US, maybe 0.5 $\times$ 10^9 phones
  - Can store this information (for billing etc.)
- The social graph
  - Each edge denotes a link from one person to another
  - > 10^9 people, > 10^{11} links
  - Store people (nodes) in memory, but maybe not all links
- The IP graph
  - Each edge denotes communication between IP addresses
  - 10^9 packets/hour/router in a large ISP, 2^{32} possible addresses
  - Not feasible to store nodes or edges

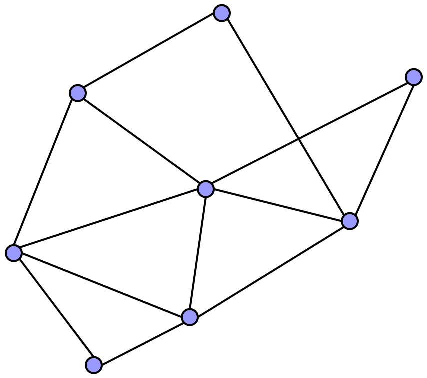


Big Graphs



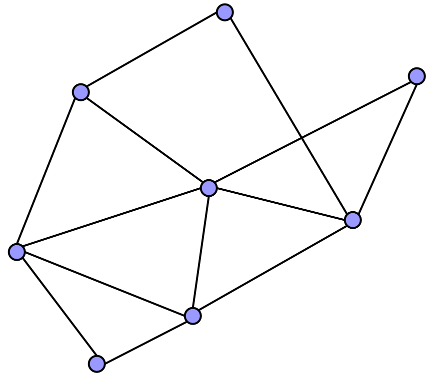
- Increasingly many "big" graphs:
  - Internet/web graph (2^{64} possible edges)
  - Online social networks (10^{11} edges)
- Many natural problems on big graphs:
  - Connectivity/reachability/distance between nodes
  - Summarization/sparsification
  - Traditional optimization goals: vertex cover, maximal matching
- Various models for handling big graphs:
  - Parallel (BSP/MapReduce): store and process the whole graph
  - Sampling: try to capture a subset of nodes/edges
  - Streaming (this talk): seek a compact summary of the graph

Streaming graph model



- The "you get one chance" model:
  - See each edge only once
  - Space used must be sublinear in the size of the input
  - Analyze costs (time to process each edge, accuracy of answer)
- Variations within the model:
  - See each exactly once or at least once?
    - Assume exactly once, this assumption can be removed
  - Insertions only, or edges added and deleted?
  - How sublinear is the space?
    - Semi-streaming: linear in n (nodes) but sublinear in m (edges)
    - "Strictly streaming": sublinear in n, polynomial or logarithmic

Streaming is hard!

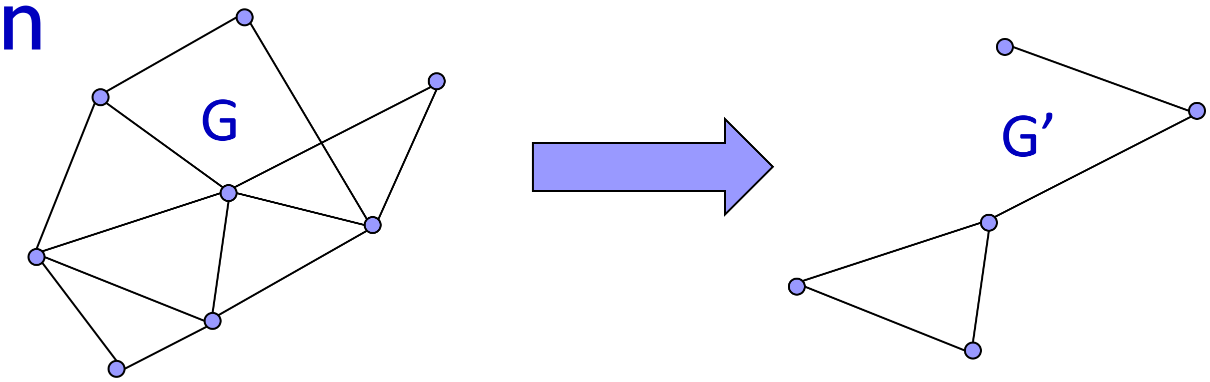


- With sublinear in n (nodes) space, life is difficult
  - Cannot remember whether or not a given edge was seen
  - Therefore, cannot determine (e.g.) whether graph is connected
  - Standard relaxations, specifically randomization, do not help
  - Formal hardness proved via communication complexity
- Different relaxations are needed to make any progress
  - Relax space: allow linear in n space – semi-streaming model
  - Make assumptions about input – parameterized streaming model

Parameterized Streaming

- For many “reasonable” graphs we can make assumptions
  - About edge density (many real massive graphs are not dense)
  - About cost/size of the solution
- Draw inspiration from fixed parameter-tractablility (FPT)
  - For (NP) Hard problems: assume solution has size k
  - Naïve solutions have cost exp(n)
  - Seek solutions with cost poly(n)exp(k) – reasonable for small k
  - Report “no” if solution size is greater than k

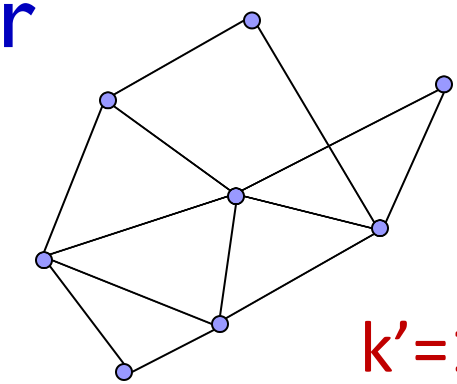
Kernelization



- A key technique is kernelization
  - Reduce input (graph) G to a smaller (graph) instance G'
  - Such that solution on G' corresponds to solution on G
  - Size of G' is poly(k)
  - So naïve (exponential) algorithm on G' is FPT
- Kernelization is a powerful technique
  - Any problem that is FPT has a kernelization solution

Kernelization for Vertex Cover

Vertex cover: find a set of vertices S so every edge has at least one vertex in S



k'=1

- Set k'=k, desired size of vertex cover
- Repeat till neither of the following can be applied
  - There is a vertex v in G with degree > k'. v must be in any cover. Remove v and all edges incident on v from G, decrease k' by one.
  - There is an isolated vertex v in G. Remove v from G.
- If neither rule can be applied, but $m>k'^2$ then G does not have a vertex cover of size at most k'.
- Else, G' is a kernel with at most $2k'^2$ nodes and k'^2 edges
  - Can run exponential time algorithm on G' to test for vertex cover

J. F. Buss and J. Goldsmith. Nondeterminism within P, 1993

Kernelization on Graph Streams

- A simple algorithm for insertions only
  - Maintain a matching M (greedily) on the graph seen so far
  - For any v in the matching, keep up to k edges incident on v as G_M
  - If $|M|>k$, quit: any vertex cover must have more than k nodes
  - At any time, run kernelization algorithm on the stored edges G_M

- Key insight: size of M is a lower bound on size of vertex cover
- Proof outline: argue that kernelization on G_M mimics that on G
  - Every step on G_M can be applied to G correspondingly
  - We keep "enough" edges on a node to test if it is high-degree
- Guarantees $O(k^2)$ space: at most k edges on $2k$ nodes
  - Lower bound of $\Omega(k^2)$ in the streaming model for Vertex Cover

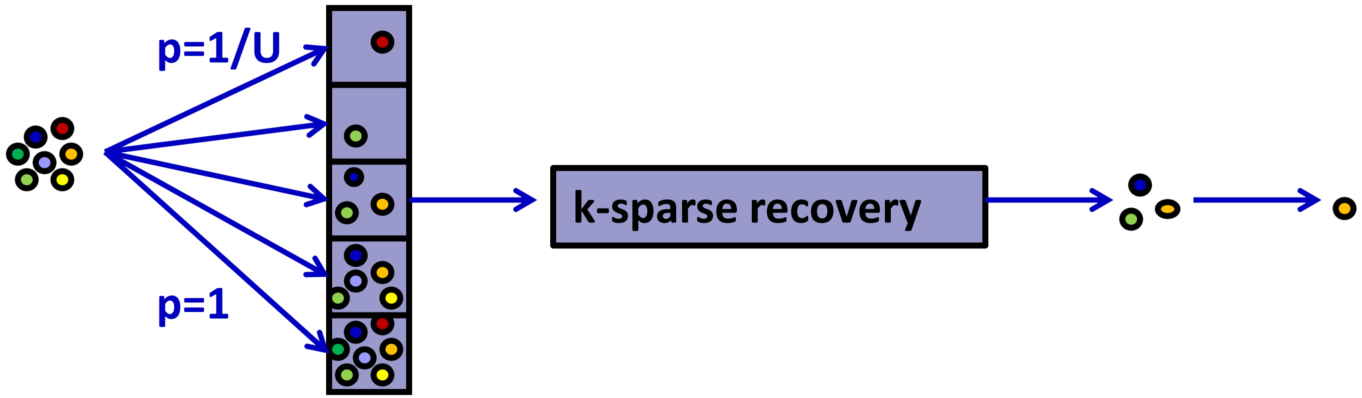
Kernelization on Dynamic Graph Streams

- More challenging case: dynamic graph streams
  - Edges are inserted and deleted
- Previous algorithm breaks: deleting a matched edge means we no longer have a maximal matching
- Study promise problem that max matching always at most size k
  - Open problem: remove the need for this promise
- Need some additional technology: l_0 sampling
  - Allows us to deal with high degree nodes
  - A sketch algorithm: maintains linear transform of input
    - Allows inserts and deletes to be analyzed easily

L_0 Sampling

- Goal: sample (near) uniformly from items with non-zero frequency
- General approach: [Frahling, Indyk, Sohler 05, C., Muthu, Rozenbaum 05]
  - Consider input to define a vector of frequencies
  - Sub-sample all items (present or not) with probability p
  - Generate a sub-sampled vector of frequencies f_p
  - Feed f_p to a *k-sparse recovery* data structure
    - Allows reconstruction of f_p if number of non-zero entries $< k$
  - If vector f_p is k-sparse, sample from reconstructed vector
  - Repeat in parallel for exponentially shrinking values of p

Sampling Process



- Exponential set of probabilities, $p=1, \frac{1}{2}, \frac{1}{4}, 1/8, 1/16 \ldots 1/U$
  - Let $N = F_0 = |\{ i : f_i \neq 0\}|$
  - Want there to be a level where k-sparse recovery will succeed
  - At level p, expected number of items selected S is Np
  - Pick level p so that $k/3 < Np \leq 2k/3$
- Chernoff bound: with probability exponential in k, $1 \leq S \leq k$
  - Pick $k = O(\log 1/\delta)$ to get $1-\delta$ probability

k-Sparse Recovery

- Given vector x with at most k non-zeros, recover x via sketching
  - A core problem in compressed sensing/compressive sampling
- Randomized construction: hash elements to $O(k)$ buckets
  - Elements are probably isolated in each bucket
  - Keep count of items and sum of item identifiers in each cell
  - Sum/count will reveal item id
  - Avoid false positives: keep fingerprint of items in each cell
- Can keep a sketch of size $O(k \log U)$ to recover up to k items

Sum, $\sum_{i : h(i)=j} i$
Count, $\sum_{i : h(i)=j} x_i$
Fingerprint, $\sum_{i : h(i)=j} x_i r^i$

Sampling and recovery of neighbourhoods

- Back to maximal matchings and vertex cover
  - Algorithm outline: maintain a maximal matching under updates
- Can have large neighbourhoods of matched nodes
  - E.g. high degree node (degree n)
- If edge from matching is deleted, we want to replace it
  - There are many possible candidates, can't store them all
  - Some are incident on other matched nodes, so can't be used
  - Insight: there are at most $2k$ matched nodes (from promise)
  - So if we can recover more than $2k$, should find some to match
  - Or, there are no edges to add to matching, so it is maximal
- Keep an l_0 sampling sketch for each matched node

Algorithm Outline

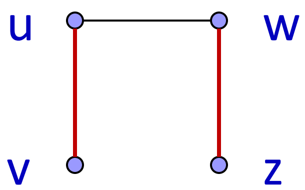
- **Goal**: keep information on only $O(k)$ matched nodes at a time
  - Keep $O(k$ poly-log$)$ size sketch per node to recover $2k$ neighbours
  - Guarantee $O(k^2$ poly-log$)$ space, and fast time to update
- Insertion of edge (u,v):
  - If u and v unmatched, add edge to matching and create sketches
  - If u (repectively v) matched, add edge to sketch of u (resp. v)
  - If u and v both matched (to other nodes), add edge to both sketches
- Deletion of edge (u,v):
  - If u and v unmatched – error! Matching was not maximal!
  - If only 1 of u, v matched, delete edge from corresponding sketch*
  - If (u, v) in M, delete from M and sketches. Attempt to rematch!*
  - If (u,v) matched but not to each other, delete (u,v) from sketches

Rematching nodes

- Setting: (u,v) was in M but got deleted
  - Want to see if we can rematch u (resp. v) from current edges

- Depends on degree of u:
  - u is low-degree (< k poly-log):
    Can recover the full neighbourhood of u, and see if any available
  - u is high-degree
    Can’t recover the full neighbourhood of u
    But there can only be 2k matched neighbours
    If we use sketch to sample neighbours, the odds are in our favour
    Even over the course of the stream (assumed fixed in advance)
    Formally: analyze probability of successful rematching (Chernoff)

*Bookkeeping



- Challenge: sketches may not contain all edge information
- E.g: edge (u,v) arrives, add to matching, and sketch(u), sketch(v)
  edge (u,w) arrives: add to sketch(u)
  edge (w,z) arrives: add to matching M,
  add to sketch(w) and sketch(z)
  delete(u,v): u is unmatched, cannot rematch
  delete(w,z): w is unmatched.
  Then (u,w) is available but is not stored in sketch(w)
  We wouldn't know to look in sketch(u)!

- Solution: keep more information about arrival time of edges and extract neighbourhood information from low-degree nodes

Data Structure and Timestamps

- Need to keep more information on edges
  - To avoid deleting edges from sketches that don't contain them
- Keep a structure T containing subset of edges incident on M
  - At most k matched nodes, so T contains $O(k^2)$ edges
- Assign "timestamps" to each event
  - Via a counter, or a clock
  - t_u of vertex u is time when u was most recently matched

Invariants

- Maintain the following set of invariants over the structures. For every "live" edge (u, v) at time t:

1. (u,v) is encoded in at least one of sketch(u) and sketch(v) [so no missing edges]
2. If u and v both in M: $(u,v) \notin$ sketch(v) iff $t_u < t_v$ and $(u, v) \notin T$
3. If u and v both in M: $(u,v) \in$ sketch(v), $\in$ sketch(u) iff $(u, v) \in T$

- Invariants ensure we know where to look for edges
  - Can implement updates that maintain all invariants
  - Means that all unmatched neighbours of a matched node are encoded in its sketch

Summary of Matching Algorithm

- Keep $O(k \text{ poly-log})$ space for $O(k)$ nodes in current matching
- Move edges between sketches so only k sketches are kept
- Patch up the matching online to keep it maximal
- Matching also allows vertex cover kernelization at any time
  - Takes time $O(2^{2k^2})$ to look for a vertex cover

Concluding Remarks

- Use of l_0 sketches has arisen in several recent graph algorithms
  - Streaming graph connectivity in $O(n \text{ polylog})$ space [Ahn, Guha, McGregor 12]
  - Dynamic graph connectivity in polylogarithmic worst-case time [Kapron, King, Mountjoy 13]
- Prompts several natural questions:
  - Can other streaming ideas inspire new graph algorithms?
  - Can streaming (bounded space) lead to dynamic (fast updates)?
  - Can the primitives (l_0 sampling) be engineered for practical use?
  - Can assumptions (promise on input) be removed?

Thank you!