ProbGraph: High-Performance and High-Accuracy Graph Mining with Probabilistic Set Representations

Authors: Maciej Besta, Cesare Miglioli, Paolo Sylos Labini, Jakub Tˇetek, Patrick Iff,
Raghavendra ́Kanakagiri, Saleh Ashkboos, Kacper Janda, Michał Podstawski,
Grzegorz Kwa ́sniewski, Niels Gleinig, Flavio Vella, Onur Mutlu, Torsten Hoefler

Presented By: Collin Warner

Motivation

- Graph mining is slow
    - Hard to parallelize since there exists little locality and irregularities in some graphs
- Useful to many problems in modern graphs
    - Examples: Triangle Counting, Clique Counting, Vertex Similarity, Graph Clustering

Contributions

- Provides an approximate algorithm trading accuracy for speed
- Helps general class of graph problems requiring set intersections in their routines.
- Approximation is tunable, and claims up to 50x speedups with up to 90% accuracy

Data Review

- Claims appear to lack support in their data, there is high variance, and not a clear link on how they get 98% or 90% accuracy claims.

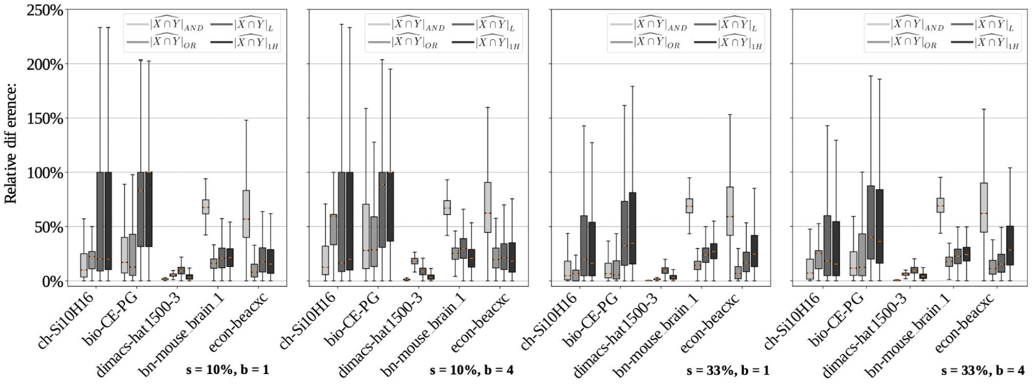
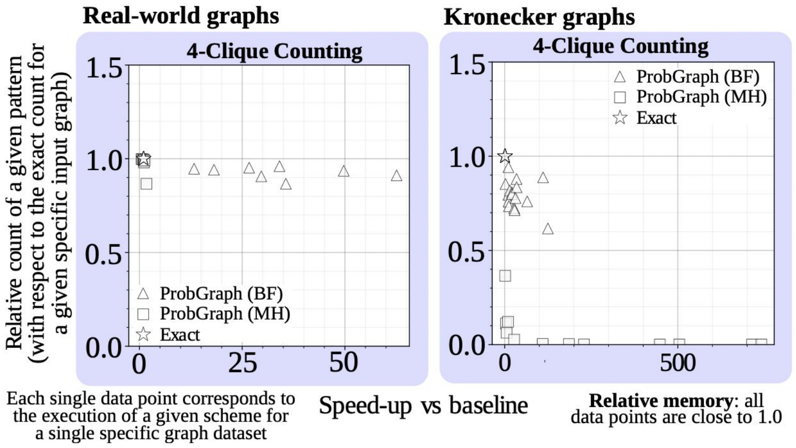
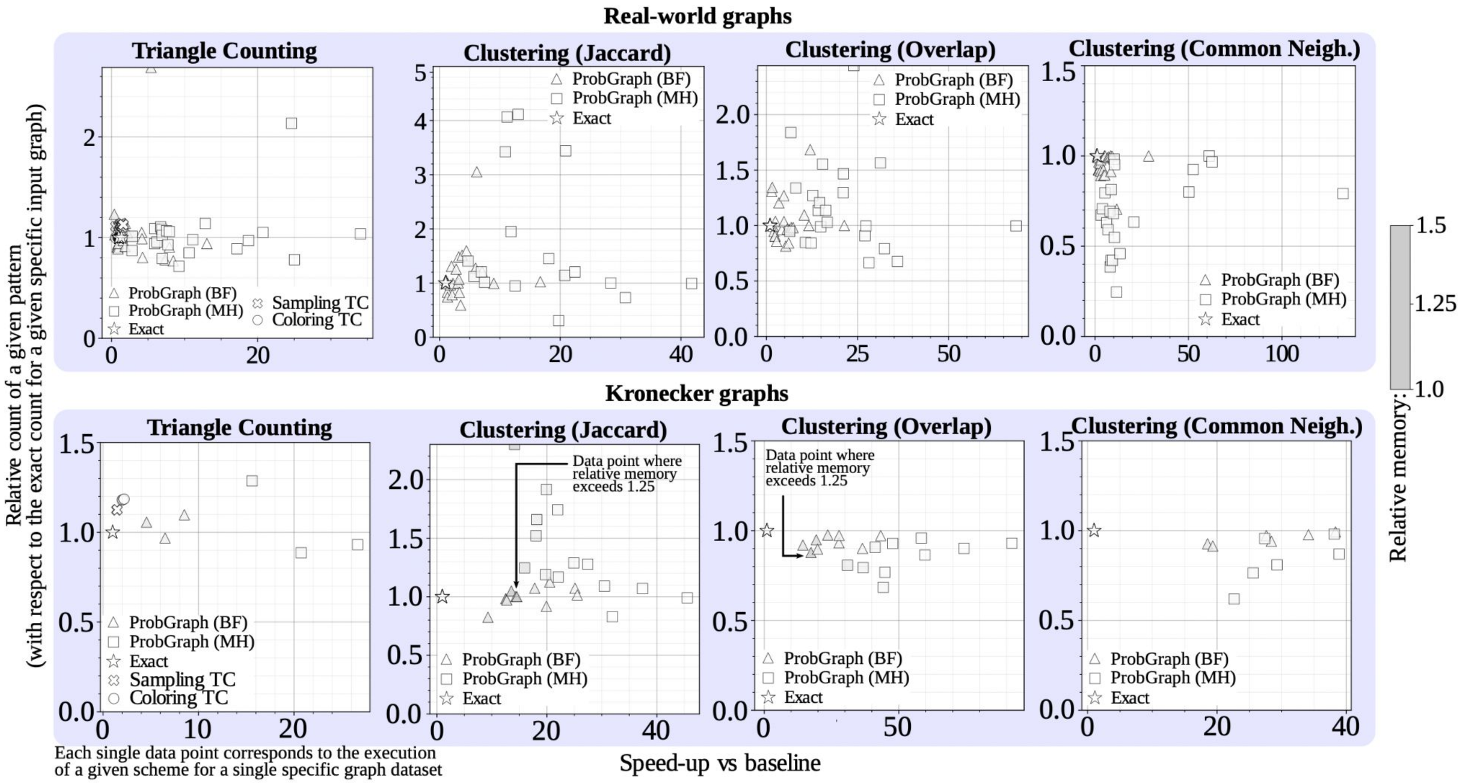


Fig. 3: Analysis of the accuracy of PG estimators of $|X \cap Y|$.



Additional Data Review

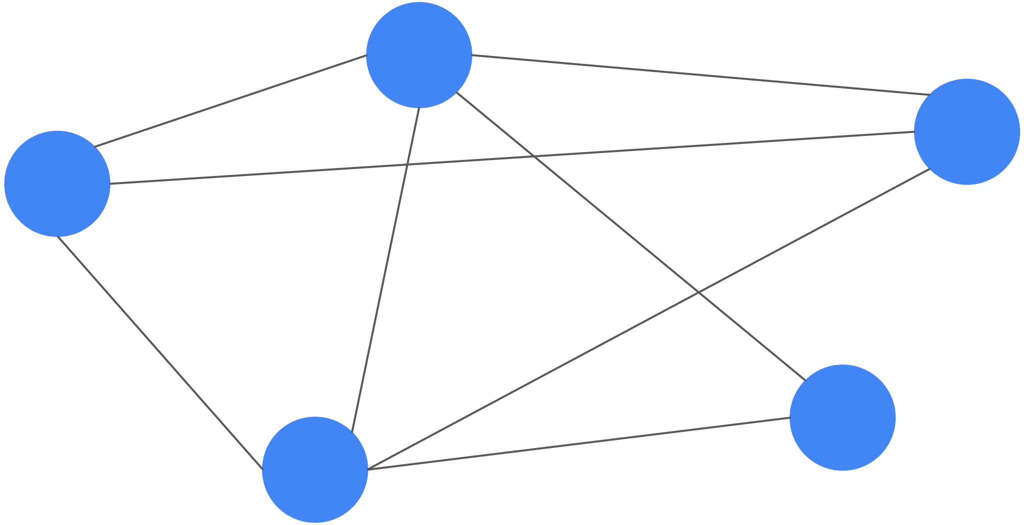


Overview

- Provide background on triangle counting to use as a motivating example
- Recognize a common subroutine in computation is set intersections
- Delve into Bloom Filters and MinHash approximation algorithms
- Show approximation algorithm given in ProbGraph

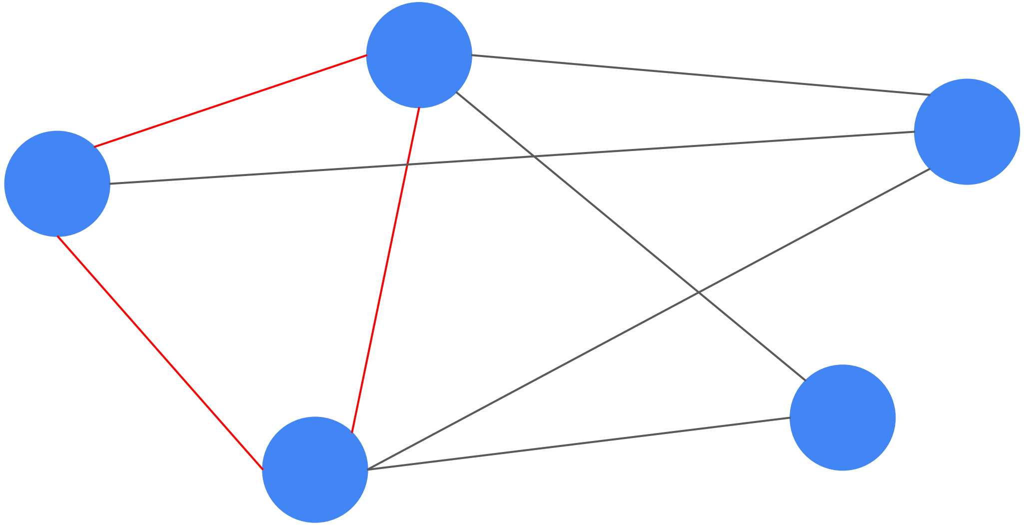
Triangle Counting

- Find all unique triples such that each pair of vertices shares and edge.
- Used to analyze real world graphs: cluster coefficient, spam filtering, find structure
- There is an n^3 algorithm: enumerate all triples and check



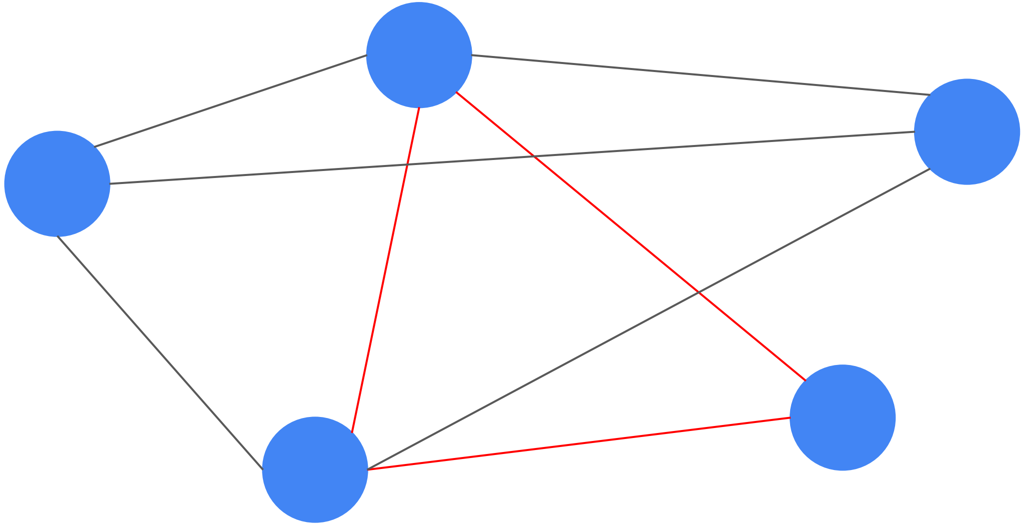
Triangle Counting

- Find all unique triples such that each pair of vertices shares and edge.
- Used in real-world graphs to figure out connectedness of a graph.
- There is an n^3 algorithm: enumerate all triples and check



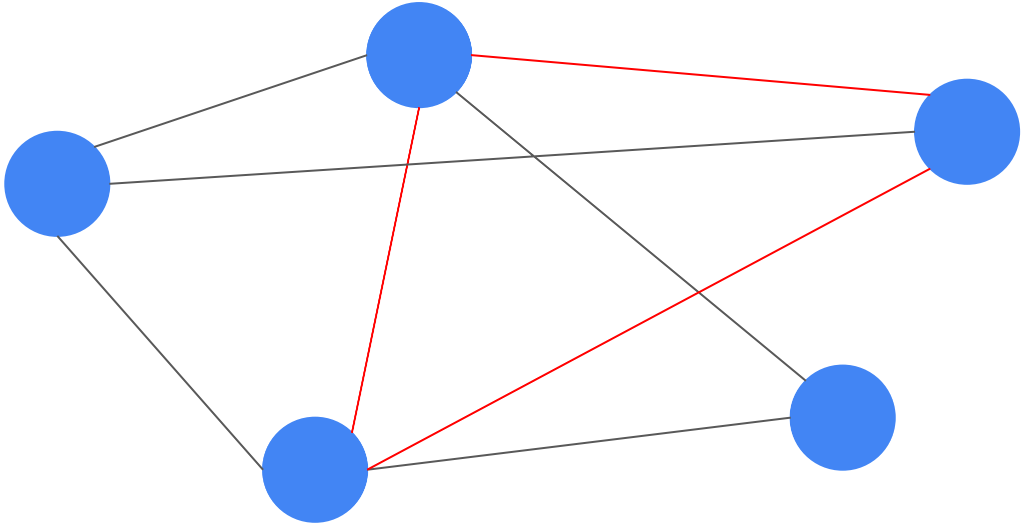
Triangle Counting

- Find all unique triples such that each pair of vertices shares and edge.
- Used in real-world graphs to figure out connectedness of a graph.
- There is an n^3 algorithm: enumerate all triples and check



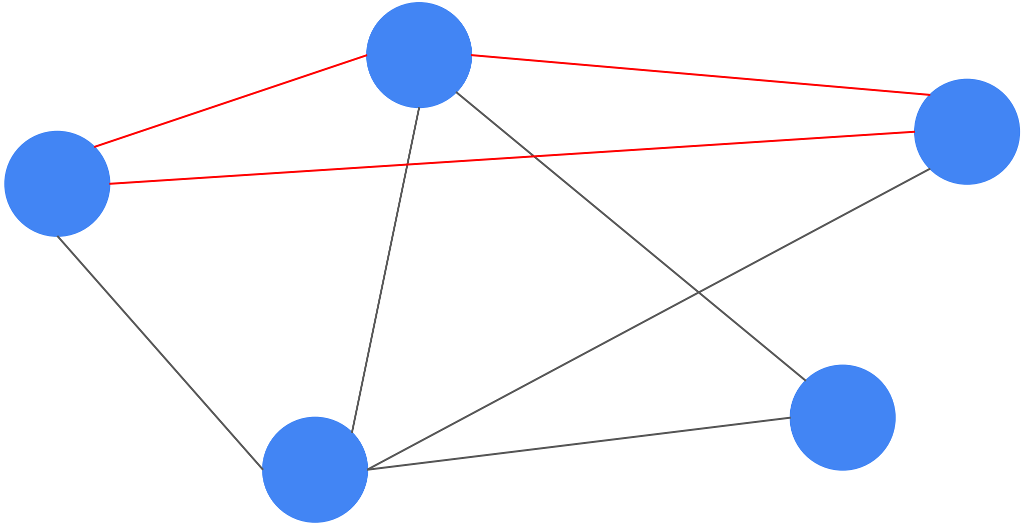
Triangle Counting

- Find all unique triples such that each pair of vertices shares and edge.
- Used in real-world graphs to figure out connectedness of a graph.
- There is an n^3 algorithm: enumerate all triples and check



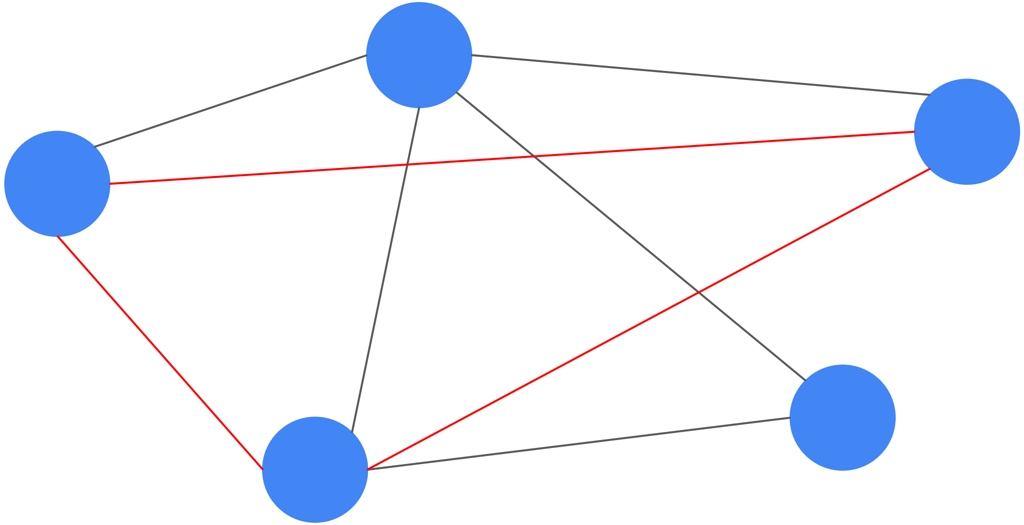
Triangle Counting

- Find all unique triples such that each pair of vertices shares and edge.
- Used in real-world graphs to figure out connectedness of a graph.
- There is an n^3 algorithm: enumerate all triples and check



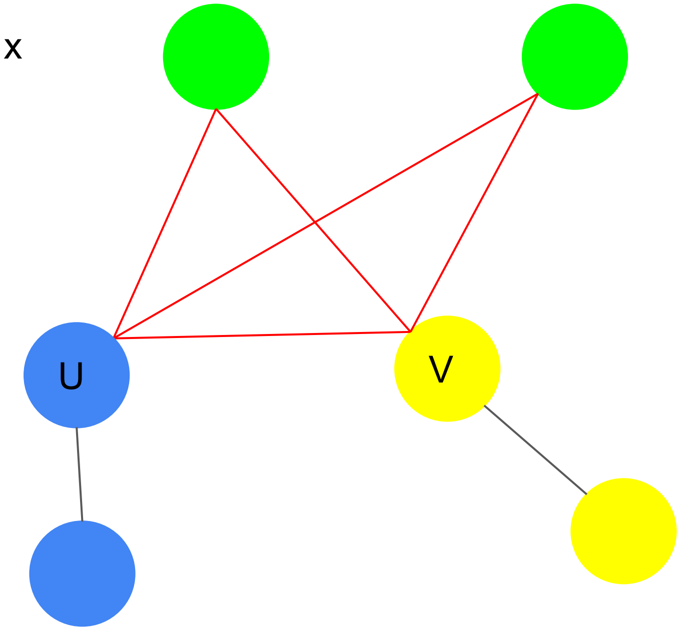
Triangle Counting

- Find all unique triples such that each pair of vertices shares and edge.
- Used in real-world graphs to figure out connectedness of a graph.
- There is an n^3 algorithm: enumerate all triples and check



Triangle Counting Faster Approach

- Let U, V be neighboring vertices and N_x be the neighbors of x
- Then $N_U \cap N_V$ / {U, V} are triangles

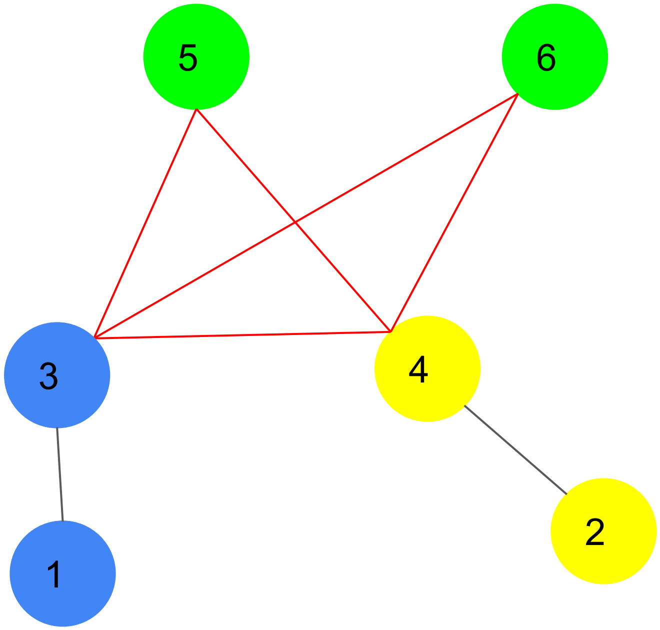


Triangle Counting Algorithm

// Derive a vertex order R s.t if $R(v) < R(u)$ then $d_v \le d_u$
for $v \in V$ do: $N_v^+ = \{u \in N_v \mid R(v) < R(u)\}$

tc = 0
for $v \in V$ do:
    for $u \in N_v^+$ do: tc += $|N_v^+ \cap N_v^+|$

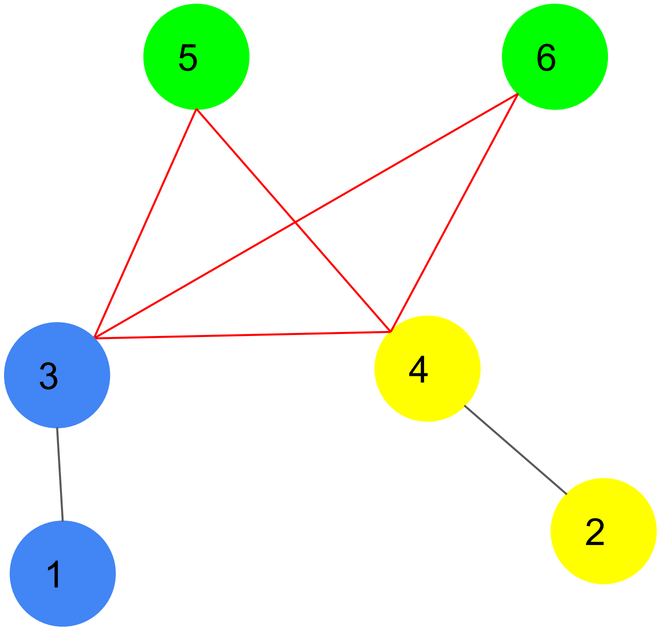
- Let d be max degree and n be number of nodes
- Initial loop takes O(nd)
- Main loop takes $O(nd^2)$



Triangle Counting Parallel Algorithm

// Derive a vertex order R s.t if $R(v) < R(u)$ then $d_v \leq d_u$
for $v \in V$ [in par] do: $N_v^+ = \{u \in N_v \mid R(v) < R(u)\}$

tc = 0
for $v \in V$ [in par] do:
    for $u \in N_v^+$ [in par] do: tc += $|N_v^+ \cap N_v^+|$



Other examples

- Clique Counting
- Vertex Similarity
- Graph Clustering

```
/* Input: A graph G. Output: Number of 4-cliques ck ∈ ℕ. */
/Derive a vertex order R s.t. if R(v) < R(u) then d_v ≤ d_u:
for v ∈ V [in par] do: N_v^+ = {u ∈ N_v | R(v) < R(u)}
ck = 0;
for u ∈ V [in par] do:
  for v ∈ N_u^+ [in par] do:
    C_3 = N_u^+ ∩ N_v^+ //Find 3-cliques
    for w ∈ C_3 do: //For each 3-clique...
      ck += |N_w^+ ∩ C_3| //Find 4-cliques
```

Listing 2: Reformulated 4-Clique Counting.


```
/* Input: A graph G. Output: Number of 4-cliques ck ∈ ℕ. */
/Derive a vertex order R s.t. if R(v) < R(u) then d_v ≤ d_u:
for v ∈ V [in par] do: N_v^+ = {u ∈ N_v | R(v) < R(u)}
ck = 0;
for u ∈ V [in par] do:
  for v ∈ N_u^+ [in par] do:
    C_3 = N_u^+ ∩ N_v^+ //Find 3-cliques
    for w ∈ C_3 do: //For each 3-clique...
      ck += |N_w^+ ∩ C_3| //Find 4-cliques
```

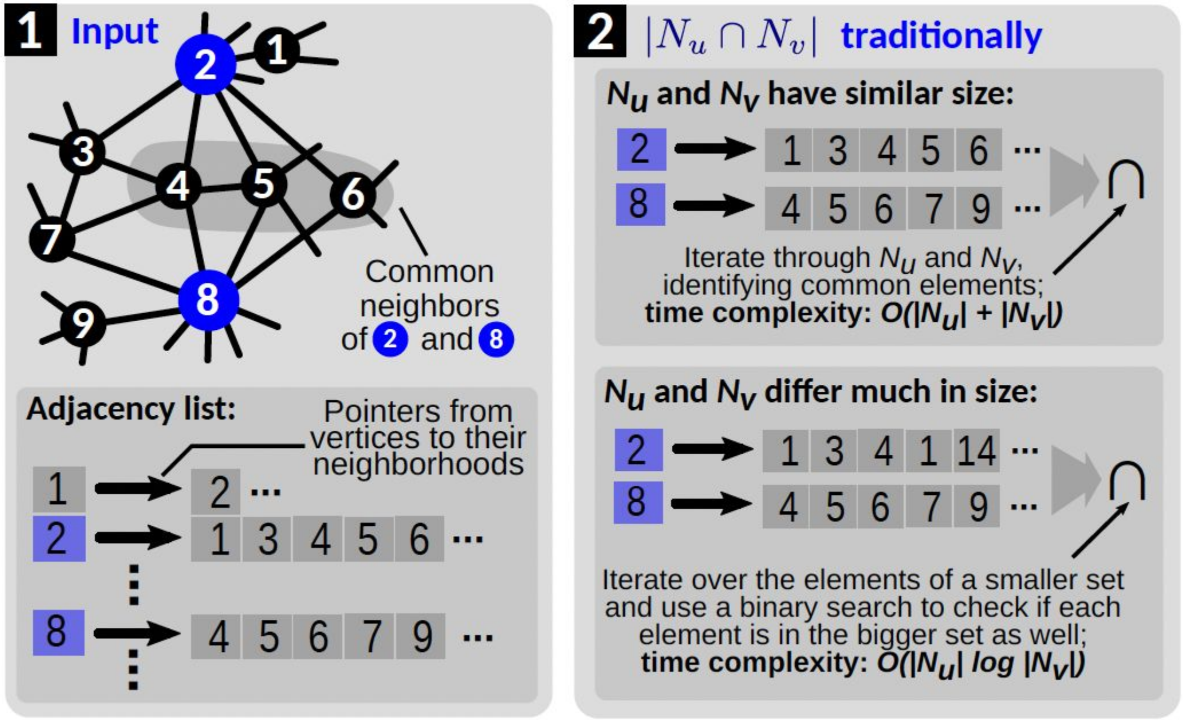
Listing 2: Reformulated 4-Clique Counting.


```
/* Input: A graph G = (V, E). Output: Clustering C ⊆ E
 * of a given prediction scheme. */
//Use a similarity S_C(v, u) = |N_v ∩ N_u| (see Listing 3).
for e = (v, u) ∈ E [in par] do: //τ is a user-defined threshold
  if |N_v ∩ N_u| > τ: C ∪= {e}
//Other clustering schemes use other similarity measures.
```

Listing 4: Jarvis-Patrick clustering.

Bottleneck

- |X ∩ Y| is slow



How to make $|X \cap Y|$ faster?

- Trading some accuracy for speed
- Use of Bloom Filters and MinHash sets to approximate these intersections

Bloom Filter

- Want space efficient/fast answering to membership queries
- False positives
- Bloom filter has L element bit vector
  - Set of hashes, $\{h_i\}$, computes an integer in [1, L]
- Add Element
  - Compute each hash, set corresponding bit to 1
- Retrieve Element
  - Compute each hash, if all 1, return True

2 hashes, $L \in [1, 3]$

Insert a -> {1, 1}
BF = 100
Insert b -> {3}
BF = 101

Now c -> {1, 3} would be “contained” although not inserted

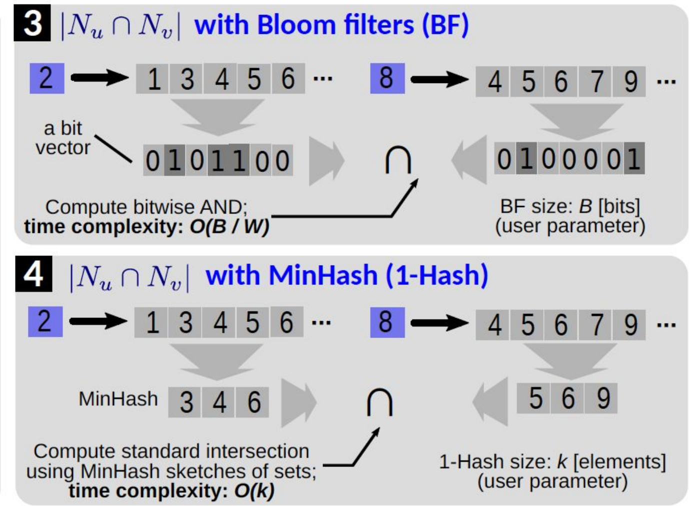
MinHash

- Take k hashes, h_1, h_2 ,.... h_k
- Compute hash for each element
- Keep values that produce the smallest per hash values
- Variant (1-Hash): keep k smallest hash values using 1 hash function

$\{\min \{h_1\}, \min \{h_2\}, \ldots, \min \{h_k\}\}$ or $\{\min \{h\}, \min \{h\} / \min \{h\}, \ldots \}$

Approximating Intersections

- Two Options:
  - Take bitwise and of Bloom Filter and compute popcount
  - Find intersection of smaller MinHash sets



Size of bloom filter (B), cache word size (W), size of MinHash set (k)

ProbGraph Implementation

- Set a storage limit as a percentage of the graph size
- Now Bloom Filter and MinHash representations exist for the neighborsets of every node with parameters chosen not to exceed this size limit.
- Choose what approximate algorithm you would like to use.
- Very fast to compute as both approximations are much smaller than original neighbor sets.
- Additionally BF is easily vectorized.

```
//Input: Graph G, two vertices u and v
//Create a standard CSR graph with G as the input graph
CSRGraph g = CSRGraph(G);
//Create a ProbGraph representation of G based on Bloom filters
ProbGraph pg = ProbGraph(g, BF, 0.25); //Use the 25% storage budget

//Derive the exact intersection cardinality |N_u ∩ N_v|
int interEX = pg.int_card(g.N(u), g.N(v));
//Derive the estimator |N_u ∩ N_v|^_AND
int interBF = pg.int_BF_AND(pg.N(u), pg.N(v));

//Compute the exact Jaccard coefficient between u and v
double jacEX = interEX / (g.N(u).size() + g.N(v).size() - interEX)
//Compute the approximate Jaccard coefficient based on BF
double jacBF = interBF / (g.N(u).size() + g.N(v).size() - interPG)
```

Listing 5: Obtaining exact and approximate Jaccard (see Listing 3)

Questions ?