LISTING TRIANGLES

11/19/2025

A. Bjorklund, R. Pagh, V. Vassilevska Williams, and U. Zwick – Presented by Saba Molaei,
Cheriton School of Computer Science

UNIVERSITY OF WATERLOO

Problem definition

- **Input:**

  A graph $G = (V, E)$ with n vertices and m edges.

  Optional parameter t: number of triangles.

- **Output:**

  List of all triangles in G, or

  Any t triangles when full enumeration is unnecessary.

- **Triangle:**

  A triple (u, v, w) with edges (u,v), (v, w), (w, u).

UNIVERSITY OF WATERLOO

Introduction

Why Triangles?

Triangles are fundamental in:

- The study of social processes
- Community detection
- Dense subgraph mining

Triangle listing is a core primitive, but extremely computationally heavy in dense graphs.

Previous Work

Reference	Time bounds	If $\omega = 2$
Itai and Rodeh [13]	$\tilde{\mathcal{O}}\left(n^\omega + \min(n^3, nt, t^{3/2})\right)$ $\mathcal{O}\left(m^{3/2}\right)$	$\tilde{\mathcal{O}}\left(n^2 + \min(nt, t^{3/2})\right)$
Williams and Williams [28]	$\tilde{\mathcal{O}}\left(n^\omega t^{1-\omega/3}\right)$	$\tilde{\mathcal{O}}\left(n^2 t^{1/3}\right)$
Pătraşcu [23]	$\tilde{\Omega}(\min(m^{4/3}, n^2, t^{4/3}))$ *	
This paper	$\tilde{\mathcal{O}}\left(n^\omega + n^{\frac{3(\omega-1)}{5-\omega}} t^{\frac{2(3-\omega)}{5-\omega}}\right)$ $\tilde{\mathcal{O}}\left(m^{\frac{2\omega}{\omega+1}} + m^{\frac{3(\omega-1)}{\omega+1}} t^{\frac{3-\omega}{\omega+1}}\right)$	$\tilde{\mathcal{O}}\left(n^2 + nt^{2/3}\right)$ $\tilde{\mathcal{O}}\left(m^{4/3} + mt^{1/3}\right)$

Preliminaries

Preliminaries

We use $\tilde{O}(.)$ notation to suppress multiplicative factors of size $n^{o(1)}$.

Square matrix products take $\tilde{O}(n^{\omega})$ time. ω≈2.373.

For rectangular matrices, we consider multiplying an n× n^{α} matrix by an $n^{\alpha} \times n$ matrix for some $0 < \alpha < 1$. The product can be computed in $\tilde{O}\left(n^{\omega(1,\alpha,1)}\right)$ time.

When α > 0.303, $\omega(1, \alpha, 1) = 2$.

Preliminaries

- A : adjacency matrix of the graph.
- $\bar{\mathrm{A}}$: the matrix obtained by replacing 1s in the k-th column with k.
- For a set $S \subseteq V$:

  $A[*, \mathrm{S}]$:matrix with columns indexed by S.

  $A[\mathrm{S},*]$:matrix with rows indexed by S.

- The Boolean product $A[*, S].\, A[S.*]$ reveals whether a 2-path via a vertex in S exists.
- If there is only one such 2-path, then the (i, j)-th entry of the product $\bar{A}$[∗,S].A[S,∗] identifies the k for which (i, k), (k, j) ∈ E.

Light and Heavy

For an edge (i, j) ,let

$$T_{i,j} = \{k \in V \mid (i, k), (k, j) \in E\}$$

be the set of midpoints of triangles through (i,j).

The edge is Λ-light if $|T_{i,j}| \leq \Lambda$.

The edge is Λ-heavy otherwise.

A triangle is Λ-light if at least one of the edges participating in it is light, otherwise it is Λ-heavy.

Listing light triangles algorithm

Listing all Λ-light triangles – Key idea

Choose random subsets S of size n/Λ.

For a light edge (i, j) ,the probability that $|S \cap T_{i,j}| = 1$ is at least:

$$\frac{1}{\Lambda}\left(1 - \frac{1}{\Lambda}\right)^{\Lambda-1} \geq \frac{1}{e\Lambda}.$$

Repeating with

$$O(\Lambda \log n)$$

subsets identifies all Λ-light triangles with high probability.

Listing all Λ-light triangles (Monte Carlo)

LIST_LIGHT_TRIANGLES(G, Λ):

```
A  ← adjacency-matrix(G)                          # n × n, entries in {0,1}
Ā ← labeled-adjacency-matrix(G)                   # n × n, column k has label k's instead of 1's
T ← empty set of triangles
R ← c * Λ * log(n)                                # number of random samples (c is a constant)
for r in 1 .. R:                                  # Repeat for O(Λ log n) different subsets.
    S ← RANDOM_SUBSET(V, size = n/Λ)              # Step 1: pick a random vertex subset S of size ≈ n/Λ
    B ← BOOLEAN_PRODUCT(A[*, S], A[S, *])         # Step 3: compute rectangular matrix products
    C ← LABELED_PRODUCT(Ā [*, S], A[S, *])        # to detect 2-paths with midpoint in S
    for each edge (i, j) in E:                    # Step 4: scan all edges (i,j) and report triangles
        if B[i][j] == 1:                          # There is one 2-path via S
            k ← C[i][j]                           # C[i][j] identifies the midpoint if unique
            T.add( sorted-tuple(i, j, k) )
return UNIQUE(T)
```

Las Vegas Version

To convert this pseudocode into a *Las Vegas* version:

1. After collecting T, verify:
   - every reported triangle is valid
   - for every edge (i, j), the number of discovered midpoints equals $(A^2)_{ij}$
2. If any mismatch:

   repeat whole process

Time complexity

Theorem 4 *Let* $G = (V, E)$ *be a graph on* n *vertices and let* $1 \leq \Lambda \leq n$*. Then, all* Λ*-light triangles in* G *can be found in* $\tilde{O}\left(n^{\omega}\Lambda^{3-\omega}\right)$ *time, with high probability.*

Listing all triangles

Listing all triangles

We assume that these algorithms receive an upper bound t on the number of triangles in the input graph.

This upper bound can be computed before calling our algorithms, either in $\mathcal{O}(n^{\omega})$ time, or in $\mathcal{O}(m^{2\omega/(\omega+1)})$ time -> [N. Alon et.al. Color-coding, 1995].

UNIVERSITY OF WATERLOO

Sparse(m, t) algorithm

```
SPARSE_LIST(G, t):
    m ← number-of-edges(G)
    Δ ← choose-parameter-Delta(m, t)                # Choose Δ depending on m and t
    T ← empty set
    for each vertex v in G:                         # Step 1: List triangles containing any low-degree vertex
        if degree(v) ≤ Δ:
            for each pair (u, w) in neighbors(v):   # enumerate triangles through v
                if edge(u, w) exists:
                    T.add(sorted-tuple(u, v, w))
    G-high ← induced-high-degree-subgraph(G, Δ)     # Step 2: Remove edges incident to low-degree vertices
    if G-high has no edges:
        return UNIQUE(T)
    return UNIQUE( T ∪ DENSE_LIST(G-high, t) )      # Step 3: Call Dense on high-degree subgraph, vertices count is ≤ 2m/Δ
```

Dense(2m/Δ, t) algorithm

```
DENSE_LIST(G, t):
    n ← number-of-vertices(G)
    Λ ← choose-parameter-Lambda(n, t)                          # Choose Λ depending on n and t
    T ← empty set
    If n < 3
        Return T
    LightTriangles ← LIST_LIGHT_TRIANGLES(G, Λ):               # Step 1: Find all Λ-light triangles
    T ← T ∪ LightTriangles
    G-heavy ← remove-light-edges (G, LightTriangles, Λ)        # Step 2: Remove Λ-light edges, remaining graph has ≤ 3t/Λ edges.
    if G-heavy has no edges:
        return UNIQUE(T)
    return UNIQUE( T ∪ SPARSE_LIST(G-heavy, t) )               # Step 3: Remaining triangles are Λ-heavy; call Sparse
```

Time complexity

We use D(n, t) to denote the running time of Dense(n, t), and S(m, t) to denote the running time of Sparse(m, t).

In Sparse(m,t) finding all triangles that contain a low degree vertex can be easily done in O (mΔ) time by examining for every edge incident on a low degree vertex x, the length 2-paths formed by taking another edge out of x.

$$S(m, t) \leq m\Delta + D(2m/\Delta, t)$$

In Dense(n,t) we find all Λ-light triangles in $\tilde{O}(n^{\omega}\Lambda^{3-\omega})$ time.

$$D(n, t) \leq n^{\omega}\Lambda^{3-\omega} + S(3t/\Lambda, t)$$

UNIVERSITY OF WATERLOO

Time complexity

- $\Lambda = \lceil \max(3, 6n^{-(\omega+1)/(5-\omega)} t^{2/(5-\omega)}) \rceil$
- $\Delta = \lceil 2 \max(m^{(\omega-1)/(\omega+1)}, m^{2(\omega-2)/(\omega+1)} t^{(3-\omega)/(\omega+1)}) \rceil$

Time complexity

$$S(m,t) \in \begin{cases} S(m,t) \in O(m^{3(\omega-1)/(\omega+1)} t^{(3-\omega)/(\omega+1)}) & t \geq m \\ S(m,t) \in O(m^{2\omega/(\omega+1)}) & t < m \end{cases}$$

$$D(n,t) \in \begin{cases} O(n^{\omega}) & t \leq n^{(\omega+1)/2} \\ O(n^{3(\omega-1)/(5-\omega)} t^{2(3-\omega)/(5-\omega)}) & t > n^{(\omega+1)/2} \end{cases}$$

Deterministic algorithm

Listing all Λ-light triangles (Deterministic)

- Randomization was only used by the algorithm for listing light triangles.

- For each light edge (a, b) ,define:

$$x \in \{0, 1\}^n \; x_k = 1 \iff (\mathrm{a}, \mathrm{k}) \in E \; and \; (\mathrm{k}, \mathrm{b}) \in E.,$$

This is the set of midpoints of triangles on edge (a,b).

- Let P_Λ denote the set of such vectors that we would like to compute.

Listing all Λ-light triangles (Deterministic)

We want a matrix T such that:

T has only *few rows:*

$$d = \Theta\big(\Lambda \cdot f(n)\big), f(n) = n^{o(1)}.$$

Each row of T has very few non-zero entries.
Total non-zeros:

$$\leq nf(n).$$

For every Λ-sparse vector x, we can recover x from the sketch Tx in

$$O\big(\Lambda f(n)\big)\ time.$$

Thus, T acts as a *deterministic* substitute for random sampling.

UNIVERSITY OF WATERLOO

Listing all Λ-light triangles (Deterministic)

```
LIST_LIGHT_TRIANGLES_DETERMINISTIC(G, Λ):
    A ← adjacency-matrix(G)
    T ← BUILD-RECOVERY-MATRIX(n, f(n), Λ)          # Step 1: Build sparse-recovery matrix T
    d ← number-of-rows(T)
    for i in 1..d:
        D-i← diagonal-matrix-from-row(T[i, *])      # Compute diagonal matrices
        M[i] ← MATRIX-PRODUCT(A, D-i, A)            # Step 2: Computes A * D-i* A
    Tx ← BUILD_VECTOR_TX (M)                        # Step 3: Build Tx, (Tx)_i = (A D_i A)_{a,b}
    Triangles ← empty set
    for each edge (a, b) in G:
        x[a,b]← SPARSE-RECOVERY (Tx[a,b])           # Step 4: Recover x from vector Tx
        for k in 1..n:
            if x[a,b][k] == 1:                      # x[k] = 1 when k is midpoint of a triangle (a, k, b)
                Triangles.add( sorted-tuple(a, b, k) )
    return UNIQUE(Triangles)
```


UNIVERSITY OF WATERLOO

Time complexity

We can recover each $x \in P_\Lambda$ in time $O(\Lambda f(n))$.

Theorem 1 *There exists a deterministic algorithm that lists all t triangles in a graph of n vertices in time* $\tilde{O}\left(n^{\omega} + n^{3(\omega-1)/(5-\omega)} t^{2(3-\omega)/(5-\omega)}\right)$.

With the bound $\omega < 2.373$ we get a time bound of $\mathcal{O}\left(n^{2.373} + n^{1.568}\, t^{0.478}\right)$.

Theorem 2 *There exists a deterministic algorithm that lists all t triangles in a graph of m edges in time* $\tilde{O}\left(m^{2\omega/(\omega+1)} + m^{3(\omega-1)/(\omega+1)} t^{(3-\omega)/(\omega+1)}\right)$.

Listing some triangles

List some triangles(G, t) algorithm

- Make the graph tripartite (three copies of each vertex v in Part A,B,C).
- Recursively partition each of the 3 sides in half ($A_1, A_2, B_1, B_2, C_1, C_2$).
- Count triangles in each of the 8 sub-instances.
- Recurse only on the subgraph with **most** triangles.
- Stop when the the subgraph with the most triangles contains **< t** triangles.

▪ Run the full **triangle-listing** algorithm only once on G' (have at least t triangles, but each of the 8 triples of subgraphs of G' have < t).

Time complexity

We get a running time of:

$$\tilde{\mathcal{O}}\left(n^{\omega} + \left(\left(\frac{t}{T}\right)^{1/3} n\right)^{3(\omega-1)/(5-\omega)} t^{2(3-\omega)/(5-\omega)}\right)$$

Hardness motivation

Quadratic Equation Systems (QES)

Let F be a finite field and |F| its number of elements.

A quadratic equation system over F^{ℓ} is a set of k quadratic equations in ℓ variables over F.

It is stated in the paper that:

"It is easy to see that QES is NP-complete."

Hardness Motivation

Unless QES has faster algorithms, our two runtime bounds are tight, even for graphs that contain more triangles.

Theorem 3 *Suppose that for some $\epsilon_1 \geq 0$, $\epsilon_2 \geq 0$ with $\epsilon_1 + \epsilon_2 > 0$, there exists an algorithm that lists all t triangles in an m-edge graph in $O(m^{1-\epsilon_1} t^{(1-\epsilon_2)/3})$ time or in an n-vertex graph in $O(n^{1-\epsilon_1} t^{(1-\epsilon_2)2/3})$ time. Then, for any finite field F, there exists an $|F|^{(1-\delta)l}\,\mathrm{poly}(l,k)$ time algorithm for $\delta > 0$ that solves l-variate quadratic equation systems with k equations over F^l.*

QES Reduction Outline

1. **Start with a QES instance**:
   Equations of the form $x'Q_i x + E_i x + S_i = 0$.

2. **Hash the equations** by forming random linear combinations using random vectors R_i.

3. **Property of hashing:**

   - Every solution of the original system remains a solution of the hashed system.
   - A non-solution becomes a solution with probability q^{-h}.
   - With good probability, the hashed system has very few false solutions.

4. **Build a graph G whose triangles correspond to solutions of the hashed system.**

5. **List triangles in** G using the assumed faster triangle-listing algorithm.

6. **Check each found triangle** to see whether it corresponds to a solution of the original system.

Thank you!