Graph Algorithms on A transpose A.

Benjamin Chang
John Gilbert, Advisor

June 2, 2016

Abstract

There are strong correspondences between matrices and graphs. Of importance to this paper are adjacency matrices and incidence matrices. Multiplying such a matrix by its transpose has many applications in multiple domains including machine learning, quantum chemistry, text similarity, databases, numerical linear algebra, and graph clustering.

The purpose of this paper is to present, compare and analyze efficient original algorithms that compute properties of A^TA from A while avoiding the expensive storage and computation of the matrix A^TA and provide resources for further reading. These algorithms, designed for sparse matrices, include triangle counting, finding connected components, distance between vertices, vertex degrees, and maximally independent sets.

Contents

Chapter 1

Introduction and Motivations

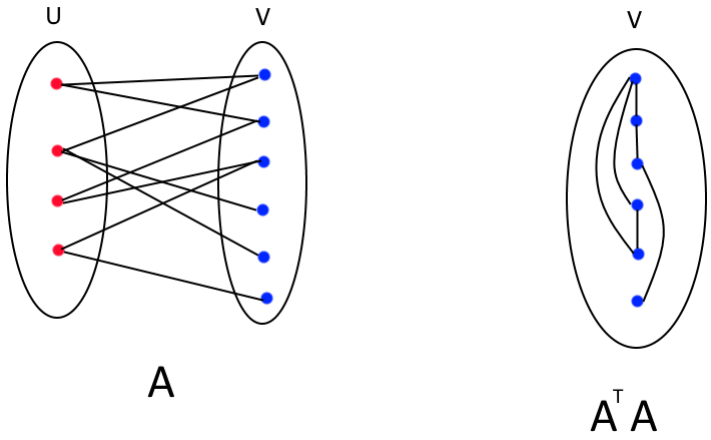
1.1 Definitions

Definition 1. *A bipartite graph $G = (U, V, E)$ has vertex sets U and V and edges $E \subseteq U \times V$. Define $m = |U|$ and $n = |V|$. The adjacency matrix $A \in M_{m \times n}$ has rows representing vertices of U and columns representing vertices of V and is defined by*

$$A_{i,j} = \begin{cases} 1, & i \in U \text{ adjacent to } j \in V \\ 0, & \text{otherwise} \end{cases}$$

We deal only with finite sets of vertices and edges. Each vertex and edge is assumed to be numbered, and a vertex or edge's number is used interchangeably with the vertex or edge itself. The vertices in U are numbered $1, \ldots, m$ and vertices in V are numbered $1, \ldots, n$. Since a graph is uniquely identified by its adjacency matrix, we will use the two interchangeably.

In this work, we are interested in algorithms concerning $A^T A$ where A is the adjacency matrix of some sparse bipartite graph. Since $A^T A$ is square and symmetric, we can represent $A^T A$ as a undirected weighted graph.



1.2 Applications

Matrices and graphs of the form A^TA appear in many contexts.

Definition 2. *If every edge e is between two vertices i and j such that $i < j$, then an oriented incidence matrix, M, of an undirected graph $G = (V, E)$ has dimension $|E| \times |V|$ with rows representing edges and columns representing vertices and is defined by*

$$M_{e,v} = \begin{cases} 1, & v = i \\ -1, & v = j \\ 0, & \textit{otherwise} \end{cases}$$

The matrix M^TM, where M is the oriented incidence matrix of a graph G, is the Laplacian of G. The Laplacian has many practical purposes including computing the number of spanning trees, approximating max flow, and image processing.

Furthermore, if incidence matrices are a way to store sparse matrices and M^TM, the Laplacian, is strongly related to the adjacency matrix, another way to store sparse matrices. Computing properties about M^TM can allows us to discover properties about the adjacency matrix from the incidence matrix.

Definition 3. *The Gram matrix, R, of a set of vectors $v_1, \ldots, v_n$ is an $n \times n$ matrix where*

$$R_{i,j} = v_i \cdot v_j$$

Equivalently, if V is the matrix with column vectors $v_1, \ldots, v_n$, then

$$R = V^T V$$

Gram matrices have applications in machine learning, quantum chemistry, and text similarity.

In addition to Laplacian and Gram matrices, the matrix $A^T A$ appears in triangle counting, finding Erdős numbers, databases, and more.

1.3 Compressed Column/Row Storage

In the algorithms presented in this paper, we will use Compressed Column Storage (CCS) or Compressed Row Storage (CRS) to store the sparse matrix A. CCS consists of three arrays, an array of indices of size $|V| + 1$ called the RI, an array of row values of size $|nnz(A)|$ called R, and an array of element values of size $|nnz(A)|$ called V. To find all the elements in column c, you iterate over all $RI(c) \leq i < RI(c+1)$ Then there is a nonzero row $R(i)$ in column c where $A_{R(i),c} = V(i)$. CRS works in the same fashion except compressing by rows instead of columns.

CCS allows fast access of A by its columns and CRS allows fast access of A by its rows. In CRS we can efficiently find the neighbors of vertices in U and in CCS we can efficiently find neighbors of vertices in V. More information can be found in [11].

1.4 Goals

This paper presents several algorithms computing information about $A^T A$ from the matrix A that requires less time or space than first computing $A^T A$. Calculating the matrix $A^T A$ exactly requires $O(\sum_{i=1}^{n} nnz(A(i,:))^2)$ time. The amount of additional space required is $O(nnz(A^T A))$. Even if A is very sparse, $A^T A$ can be very dense and even storing the matrix can become an issue.

Chapter 2

Connected Components and Distance

In this chapter we explore the concepts of adjacency, connectedness and distance in the graph A^TA and how they relate to the graph A.

2.1 Adjacency in A^TA

Here we introduce a criterion in Theorem 1 for adjacency in A^TA which will be useful throughout every chapter.

Theorem 1. *If A is an adjacency matrix, then vertices $v_i, v_j \in V$ are adjacent in A^TA if and only if they share a common neighbor in A.*

Proof. By definition of matrix multiplication,

$$(A^TA)_{i,j} = \sum_{k=1}^{m} A_{i,k}A_{k,j}$$

Because A is an adjacency matrix, each element is either 0 or 1.

$$(A^TA)_{i,j} = 0 \iff A_{i,k}A_{k,j} = 0 \text{ for } \textbf{all } k$$

.

$$(A^TA)_(i,j) \neq 0 \iff A_{i,k}A_{k,j} \neq 0 \text{ for } \textbf{some } k$$

v_i and v_j are adjacent in $A^TA \iff (A^TA)_{i,j} \neq 0$. Furthermore, they share a common neighbor if and only if $A_{i,k}Ak, j$ is nonzero for some k. So, v_i and v_j are adjacent if and only if they share a common neighbor in A. □

2.2 Connected Components

2.2.1 Connectedness in A^TA

Lemma 1. *There is a path between two vertices in A^TA if and only if there is path between them in A*

Proof. □

Theorem 2. *Two vertices $v_i, v_j \in V$ are connected in A^TA if and only if they are connected in A.*

Proof. $\implies$) Assume v_i and v_j are connected in A^TA. Then there is a walk from v_i to v_j in A^TA. Each consecutive pair of vertices in the path are adjacent in A^TA so they share a mutual neighbor in A. So each pair of vertices are connected in A since there is a walk of length 2 between them. Since connectedness is transitive, v_i and v_j are connected.

$\impliedby$) Assume v_i and v_j are connected in A. Then there is a walk from v_i to v_j in A^TA. Since A is bipartite, the vertices alternate between vertices in V and U. Consecutive vertices in V share a mutual neighbor in U so they must be adjacent in A^TA. Therefore consecutive vertices in V are connected in A^TA. Since connectedness is transitive, all the vertices in V are connected in A^TA. □

2.2.2 Algorithm Description

Since vertices are connected in A^TA if and only if they are connected in A, we can apply traditional methods of finding connected components such as a breadth first search. However, can we do any better? Here we introduce a different algorithm and compare it to doing a breadth first search on A to find the connected components.

We know that vertices in A^TA are adjacent if they share a neighbor in V. This shared neighbor must be a vertex in U. Therefore, for every $u \in U$, the

neighbors of u must all be adjacent. We can use this fact to create a more efficient algorithm. In this algorithm we use a union find data structure.

Algorithm 1 Connected Components - Union Find

Input: Graph $G = (U, V, E)$

Output: Every connected component gets assigned a representative. Each vertex in U is labeled with the representative of its connected component.

```
1: procedure findConnComps Union Find(Graph G = (U, V, E))
2:     for all v ∈ V do
3:         make_set(v)
4:     for all u ∈ U do
5:         if u has at least one neighbor then
6:             Let v_u be a neighbor of u.
7:             for all v adjacent to u do
8:                 link(v, v_u)
9:     Create a set of representatives for each vertex.
10:    for all v ∈ V do
11:        representative(v) = find(v)
```

Now we want to prove that this algorithm is correct.

Proof. Since this algorithm links all vertices that are adjacent, all adjacent vertices have the same representative.

First we show that vertices in the same connected component have the same representative. If a connected component consists of a single vertex, then it will never be linked to anything and it will be its own representative. If two vertices are in the same connected component in A^TA, then there exists a walk between them. Since consecutive vertices in the walk are adjacent, they must have the same representative. Therefore every vertex in the walk has the same representative, in particular the start and end. Every two vertices in the same connected component have the same representative. So all vertices in the same connected component have the same representative.

Now we show that vertices in different connected components have different representatives. Since only adjacent vertices are linked, if two vertices have the same representative, there must be a walk between them. Therefore

they must be in the same connected component.

Vertices are in the same connected component if and only if they have the same representative. □

2.3 Results and Comparisons

2.3.1 Algorithm Complexity

The runtime of the Union Find algorithm is approximately $O(|V|+|E|+|U|)$. To be precise, the runtime is $O(|V| \cdot \alpha(|V|) + |U|)$ where

$$\alpha^{-1}(x) = A(x, x)$$

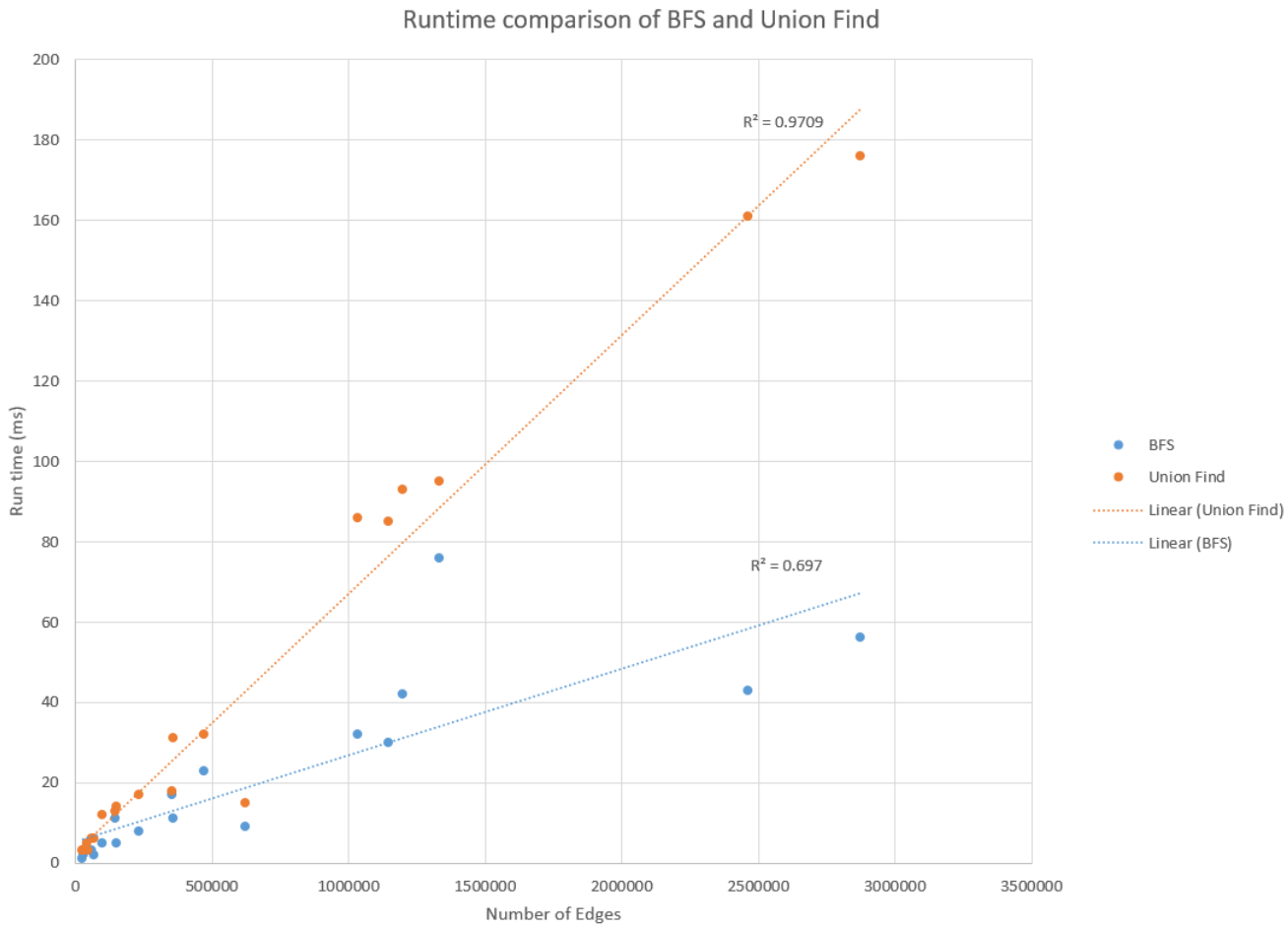
and A is the Ackermann function. The function link() is called once for every element in every column except one element in each column. So link() is called $O(|E|)$ times. Then the second loop takes $O(|E|+|U|)$ time. The find() function takes amortized $\alpha(|V|)$ time so the final loop takes $O(|V|\alpha(|V|))$ time. Together they take $O(|V| \cdot \alpha(|V|) + |E| + |U|)$ time. The algorithm takes addition space $O(|V|)$ since the UnionFind and the solution both take linear space.

In comparison, a standard breadth first search requires $O(|V|+|E|+|U|)$ and $O(|V|)$ space to store the answer. So they have similar asymptotic runtime.

2.3.2 Advantages, Disadvantages, and Results

Some disadvantages of the union-find algorithm is that it technically has nonlinear runtime and the matrix A must be stored in a format which can quickly access all elements in a row such as CRS. Storage formats like CCS would cause the algorithm to run significantly slower. However, a standard breadth first search over A requires the matrix to be easily accessed across rows and columns to be efficient. This is an even stricter requirement. Therefore, while the union-find algorithm is less restrictive on the type of data structure required, as we will see below, it is slower in every test case.

Run times were computed by averaging over 100 trials.



2.4 Distance

Definition 4. *The **distance** between two vertices is the length of the shortest path between them or ∞ otherwise. Let $d_{A^TA}(v_1, v_2)$ denote the distance between two vertices in A^TA. Let $d_A(w_1, w_2)$ denote the distance between two vertices in A.*

Theorem 3. *The distance between two connected vertices v_i and v_j in A^TA*

is exactly half the distance between the vertices in A. That is,

$$d_{A^TA}(v_i, v_j) = \frac{1}{2} d_A(v_i, v_j)$$

Proof. Note that since A is bipartite and $v_i, v_j \in V$, then the distance between them in A must be even. Since the vertices are connected, they are connected in both A and A^TA. So both distances must be finite.

First we want to show that $d_{A^TA}(v_i, v_j) \leq \frac{1}{2} d_A(v_i, v_j)$.There must exist a path of length $d_A(v_i, v_j)$ between v_i and v_j in A. Since this graph is bipartite, it alternates between vertices in V and vertices in U. Each consecutive pair of vertices in V must share a neighbor in A. Therefore, by theorem 1, they are adjacent in A^TA. So they form a path of length $\frac{1}{2} d_A(v_i, v_j)$ in A^TA since half the vertices have been removed. Therefore,

$$d_{A^TA}(v_i, v_j) \leq \frac{1}{2} d_A(v_i, v_j)$$

Next we want to show that $d_{A^TA}(v_i, v_j) \geq \frac{1}{2} d_A(v_i, v+j)$. There must be a path in A^TA between v_i and v_j of length $d_{A^TA}(v_i, v_j)$. Call this path $v_1, \ldots v_k$ where $v_1 = v_i$ and $v_k = v_j$. Each consecutive pair of vertices must be adjacent in A^TA by definition of path. Since they are adjacent they must share mutual neighbors in A by theorem 1. So there exists some $u_1 \ldots u_{k-1}$ such that $v_1 u_1 v_2 u_2 \ldots v_{k-1} u_{k-1} v_k$ is a walk in A. This walk is length $2d_{A^TA}(v_i, v_j)$. Therefore

$$2d_{A^TA}(v_i, v_j) \geq d_A(v_i, v_j)$$

$$d_{A^TA}(v_i, v_j) \geq \frac{1}{2} d_A(v_i, v_j)$$

Therefore,

$$d_{A^TA}(v_i, v_j) = \frac{1}{2} d_A(v_i, v_j)$$

□

This means that traditional methods for finding distance in A like a breadth first search can be applied to find distance in A^TA. This requires $O(|V| + |U|)$ space and $O(|V| + |U|)$ time, significantly less than the time required to compute A^TA alone.

Chapter 3

Independent Sets

Definition 5. *An **independent set** in a graph is a set of vertices such that no two vertices are adjacent. The set is **maximally independent** if it is not the strict subset of any other independent set.*

In this chapter we will modify the standard greedy maximally independent set algorithm to work for independent sets of A^TA. A maximally independent set can be found greedily by iterating over all vertices and adding vertices that maintain the independence of the independent set. To verify independence, you can check to see if the new vertex is adjacent to any vertex in the independent set. While this method can be used to find an independent set in A^TA, it requires checking adjacency in A^TA which is expensive using only A. In this chapter, we derive an alternative criterion for independence in A^TA which is more easily verified and be used to implement a greedy algorithm.

3.1 Maximally Independent Set Criterion

Theorem 4. *A set $S \subset V$ is independent in A^TA if and only if no two vertices in S share a neighbor in A.*

Proof. A set S is independent in A^TA, by definition, if and only if no two vertices are adjacent. Two vertices are adjacent precisely when they share a neighbor in A. So $S \subset V$ is independent in A^TA if and only if no two vertices in S share a neighbor in A. □

Corollary 4.1. *A set $S \subset V$ is independent in A^TA if and only if each vertex in U is adjacent to at most one vertex in S.*

Proof. $\implies$) Let $S \subset V$ be independent. Then from Theorem 4, no two vertices in S share a neighbor in A. Since no two vertices in S share a neighbor in A, no vertex in U can be adjacent to two vertices in S otherwise the two vertices in S would share a neighbor. So each vertex in U is adjacent to at most one vertex in S.

$\impliedby$) Assume $S \subset V$ such that each vertex in U is adjacent to at most one vertex in V. We know that each vertex in S can only have neighbors in U since A is bipartite and S is a subset of V. Then vertices in S have no shared neighbors because no vertex in U is adjacent to two vertices in S. Therefore by Theorem 4, S is independent. □

3.2 Maximally Independent Set Algorithm

Using Corollary 4.1, Algorithm 2 finds maximally independent sets by iteratively adding vertices to S while maintaining the constraint that no two vertices in U can be adjacent to the same vertex in S.

Algorithm 2 Maximally independent set

Input: Graph $A = (U, V, E)$
Output: Set $S \subset V$ a maximally independent set.

```
1: procedure MAXIMALLYINDEPENDENTSET(Graph A = (U, V, E))
2:     Let S = ∅.
3:     for v ∈ V do
4:         if v is unmarked and has no marked neighbors then
5:             Mark v and all its neighbors.
6:             Add v to S.
7:     return S
```

In this algorithm, a vertex $v \in V$ is marked if is in S. A vertex $u \in U$ becomes marked if one of its neighbors is in S. Before we add a vertex v to S, we check if any of its neighbors is marked. If any of its neighbors are marked, then we cannot add v to S because then that neighbor would be adjacent to two vertices in S. So we only add v to S if all its neighbors

are unmarked. This maintains the independence criterion of Corollary 4.1. at each step. Furthermore, after the algorithm finishes running, the set S is maximally independent, because if any vertex not in S would violate the independence of S if added.

This algorithm takes $O(|E| + |V|)$ time since we are looping over vertices in V and for each vertex we are traversing over its edges to check its neighbors. The algorithm requires $O(|V| + |U|)$ additional space to store the output S and to mark the vertices.

Chapter 4

Triangle Counting

Triangle counting in graphs is used as a subroutine for computing clustering coefficients or measure the likeness that neighbors are connected. In this chapter we explore ways to extend to idea of triangle counting for A^TA to be faster to compute but potentially provide similar utility or function.

Recall that we defined $m = |U|$ and $n = |V|$. Furthermore, each vertex is assigned a number from 1 to m or 1 to n.

Definition 6. *A **triangle** in the graph A^TA is a cycle in A^TA of length 3.*

Instead of counting the number of triangles, we compute a weighted count of the triangles. There are two types of weights presented here which we can compute efficiently. The two types are product-weight and sum-weight defined below. Both weights are based on the edge weights of the triangles in A^TA. The product-weight weights each triangle by the product of its edge weights. The sum-weight weights each triangle by the sum of its edge weights.

Definition 7. *Given a triangle in A^TA with vertices $a, b, c \in V$, the **product-weight** of the triangle is the product of its edges, $(A^TA)_{a,b} \cdot (A^TA)_{b,c} \cdot (A^TA)_{c,a}$. The **sum-weight** of the triangle is the sum of its edges, $(A^TA)_{a,b} + (A^TA)_{b,c} + (A^TA)_{c,a}$.*

In order to count triangles efficiently using their edge weights, we have to first understand what the edge weights mean.

Lemma 2. *Given two vertices $a, b \in V$. $(A^TA)_{a,b}$ is the number of mutual neighbors between a and b in A.*

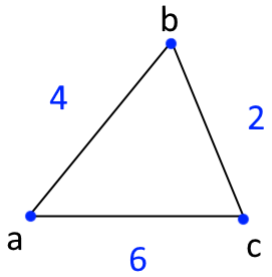
Proof. By definition of matrix multiplication and transpose,

$$(A^T A)_{a,b} = \sum_{i=1}^{m} A_{a,i} \cdot A_{b,i}$$

The product $A_{a,i} \cdot A_{b,i}$ is equal to 1 if i is adjacent to both a and b. Otherwise the product is 0. So $(A^T A)_{a,b}$ counts the number of mutual neighbors to a and b. □

4.1 Counting Triangles with Product-Weight

Consider the following triangle a, b, c in $A^T A$ with edge weights as shown.



This triangle should contribute $4 \times 6 \times 2 = 48$ to the total triangle count with product-weight. From Lemma 2, we know that there are exactly two vertices u_1 and u_2 in U that are mutual neighbors to both b and c in A.

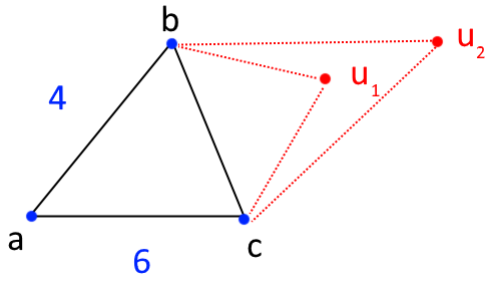


Figure 4.1: Black lines are edges in $A^T A$ and red dotted lines are edges in A.

This gives us an equivalent way to contribute 48 to the total sum. For both u_1 and u_2 we add 4×6 to the total sum. Then to get the total triangle count with multiplicity we can count shapes of the following form.

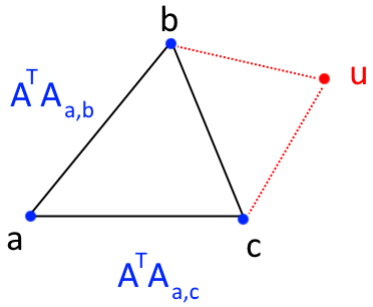


Figure 4.2: Black lines are edges in A^TA and red dotted lines are edges in A.

Each of these shapes in Figure 4.2 contributes $(A^TA)_{a,b} \cdot (A^TA)_{a,c}$ to the total triangle count.

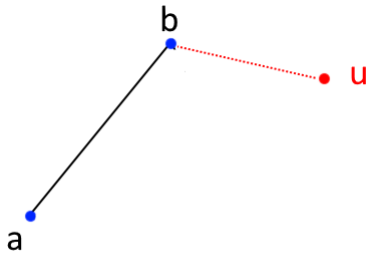


Figure 4.3: Black lines are edges in A^TA and red dotted lines are edges in A.

Fix $a \in V$ and $u \in U$. Then consider all $b \in V$ such that $b \neq a$ and there are edges from a to b in A^TA and from b to u in A as shown in the left of Figure 4.3. We can call these vertices $\{b_1, \ldots, b_n\}$.

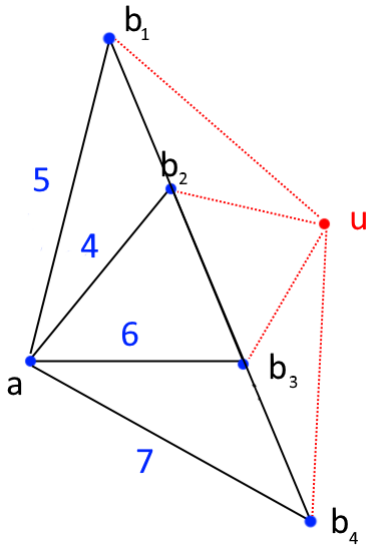


Figure 4.4: Black lines are edges in A^TA and red dotted lines are edges in A.

Then any a, b_i, b_j where $i \neq j$ forms a triangle in A^TA . This is because there is an edge in A^TA between any b_i and b_j because they both share at least one mutual neighbor in A, namely u. To compute the total amount that these $\{b_i\}$ contribute to the total sum with a and u fixed, we add the product of any two edges from a to the $\{b_i\}$ because any two of the $\{b_i\}$ will form one of the shapes from Figure 4.2 with a. This is equivalent to adding

$$\sum_{1 \leq i < j \leq n} (A^TA)_{a,b_i} \cdot (A^TA)_{a,b_j}$$

In the Figure 4.4 this, would be adding $5(4)+5(6)+5(7)+4(6)+4(7)+6(7)$ to the total count. Another way to add the same amount is to add $\frac{1}{2}((5+4+6+7)^2-(5^2+4^2+6^2+7^2))$. When we distribute $(5+4+6+7)^2$ we get all pairs of products of the addends. But we want to remove the cases where we multiply the same two numbers. So we subtract the square of each term. Then we divide by two because $5(4)$ and $4(5)$, which represent the same pair, are both included. So this gives us

$$\frac{1}{2}\left[\left(\sum_{i=1}^{n}(A^TA)_{a,b_i}\right)^2 - \sum_{i=1}^{n}(A^TA)_{a,b_i}^2\right]$$

If we define the matrix B to be A^TA except with the diaganol equal to 0. Then

$$\left(\sum_{i=1}^{n}(A^TA)_{a,b_i}\right)^2 = (AB)^{*2}_{u,a}$$

and

$$\sum_{i=1}^{n}(A^TA)^2_{a,b_i} = (A^{*2}B^{*2})_{u,a} = (AB)^{*2}_{u,a}$$

where M^{*2} denotes element-wise squaring of entries in a matrix M. Multiplying by A on the left of B elects the b_i. So the total contribution of Figure 4.4 is given by

$$\frac{1}{2}\left[(AB)^{*2}_{u,a} - (AB^{*2})_{u,a}\right]$$

Summing over all choices of $a \in V$ and $u \in U$ gives us 3 times the number of triangles with product-weight, because for any triangle there are three vertices which can be chosen as a so the triangle is included 3 times in the count. If we change B further to consist of only the lower triangular elements, then there are only edges from lower indexed vertices to higher indexed vertices. This eliminates the triple counting and gives us Theorem 5.

Theorem 5. *Define $B \in M_{n\times n}$ to be the strictly lower triangular part of A^TA,*

$$B_{i,j} = \begin{cases} 0, & \text{for } i \leq j \\ (A^TA)_{i,j}, & \text{for } i > j \end{cases}$$

The total number of triangles with product-weight is given by

$$\textit{sum with product-weight} = \frac{1}{2}\ \textit{sum}((AB)^{*2}) - \frac{1}{2}\ \textit{sum}(AB^{*2})$$

where M^{*2} denotes squaring every element in the matrix M.

Proof. Now we provide a more rigorous algebraic proof. Because B is the

strictly lower triangular, $B_{b,a} \cdot B_{c,a} \cdot B_{c,b}$ can only be non-zero when $a < b < c$.

$$\text{sum with product-weight} = \sum_{a<b<c} (A^T A)_{a,b} \cdot (A^T A)_{c,a} \cdot (A^T A)_{c,b} = \sum_{a,b,c \in V} B_{b,a} \cdot B_{c,a} \cdot B_{c,b}$$

$$= \sum_{b,c} B_{c,b} \sum_{a \in V} B_{b,a} B_{c,a}$$

We know that $B_{c,b}$ is 0 if $c \leq b$ and $(A^T A)_{c,b}$ otherwise.

$$= \sum_{\substack{b,c \in V \\ b<c}} (A^T A)_{c,b} \sum_{a \in V} B_{b,a} B_{c,a}$$

Instead of iterating over $b < c$, we can iterate over all $b \neq c$ and divide by 2.

$$= \frac{1}{2} \sum_{\substack{b,c \in V \\ b \neq c}} (A^T A)_{c,b} \sum_{a \in V} B_{b,a} B_{c,a}$$

By definition $(A^T A)_{c,b}$ is $\sum_{u \in U} A_{u,b} A_{u,c}$.

$$= \frac{1}{2} \sum_{\substack{b,c \in V \\ b \neq c}} \left(\sum_{u \in U} A_{u,b} A_{u,c} \right) \sum_{a \in V} B_{b,a} B_{c,a}$$

$$= \frac{1}{2} \sum_{\substack{a,b,c \in V \\ u \in U \\ b \neq c}} B_{b,a} B_{c,a} A_{u,b} A_{u,c}$$

$$= \frac{1}{2} \sum_{\substack{u \in U \\ a \in V}} \left[\sum_{b \in V} A_{u,b} B_{b,a} \sum_{\substack{c \in V \\ c \neq b}} A_{u,c} B_{c,a} \right]$$

In the third sum we sum over $c \in V, c \neq b$. We can instead sum over all $c \in V$ and subtract the case where $c = b$.

$$= \frac{1}{2} \sum_{\substack{u \in U \\ a \in V}} \sum_{b \in V} \left[A_{u,b} B_{b,a} \left(\sum_{c \in V} A_{u,c} B_{c,a} - A_{u,b} B_{b,a} \right) \right]$$

Then we can distribute the sum inside the braces.

$$= \frac{1}{2} \sum_{\substack{u \in U \\ a \in V}} \sum_{b \in V} \left[A_{u,b} B_{b,a} \left(\sum_{c \in V} A_{u,c} B_{c,a} \right) - (A_{u,b} B_{b,a})^2 \right]$$

$$= \frac{1}{2} \sum_{\substack{u \in U \\ a \in V}} \left(\left[\sum_{b \in V} A_{u,b} B_{b,a} \left(\sum_{c \in V} A_{u,c} B_{c,a} \right) \right] - \sum_{b \in V} (A_{u,b} B_{b,a})^2 \right)$$

The term $\sum_{b \in V} (A_{u,b} B_{b,a})^2$ is $(A^{*2} B^{*2})_{u,a}$. Since A is an adjacency matrix of 0s and 1s, A^{*2} is exactly A. Similarly, $\sum_{c \in V} A_{u,c} B_{c,a} = (AB)_{u,a}$.

$$= \frac{1}{2} \sum_{\substack{u \in U \\ a \in V}} \left[\sum_{b \in v} A_{u,b} B_{b,a} (AB)_{u,a} - (AB^{*2})_{u,a} \right]$$

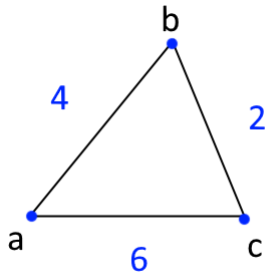
$$= \frac{1}{2} \sum_{\substack{u \in U \\ a \in V}} \left[(AB)^2_{u,a} - (AB^{*2})_{u,a} \right]$$

$$= \frac{1}{2} \operatorname{sum}((AB)^{*2}) - \frac{1}{2} \operatorname{sum}(AB^{*2})$$

□

4.2 Counting Triangles with Sum-Weight

This time we want to count the triangles where each triangle is weighted by the sum of its edge weights. Lets consider the triangle with vertices a, b, c and edge weights $4, 2, 6$.



The contribution of this triangle to the total triangle count with sum-weights is $4 + 2 + 6 = 12$. There are 3 choices of a for each triangle. If for every choice of a, we add the edge weight of the edge opposite of a.

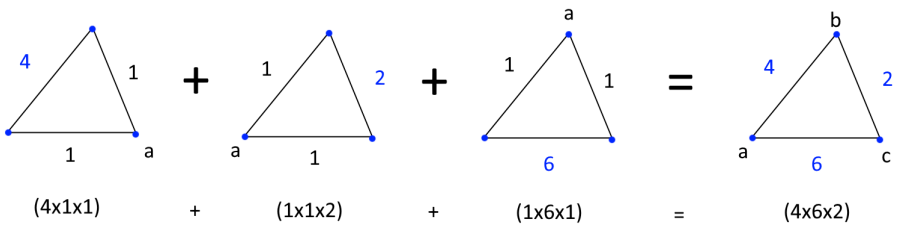


Figure 4.5: Adding edge-weights opposite of each vertex equates to adding all edge-weights

Lets look further into adding the edge with weight 2. From Lemma 2, we know that there are $u_1, u_2 \in U$ that are mutual neighbors to b and c.

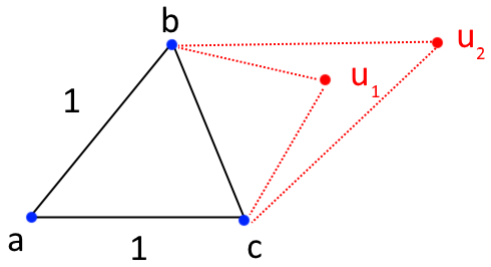


Figure 4.6: Black lines are edges in $A^T A$ and red dotted lines are edges in A.

The for each u_i we add 1 to the total triangle count with sum-weight. So we are actually just counting shapes of this type

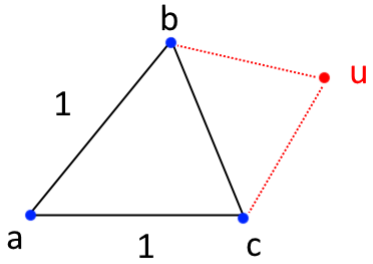


Figure 4.7: Black lines are edges in A^TA and red dotted lines are edges in A.

From here we can see that counting triangles with sum-weight is very similar to counting triangles with product-weight.

Theorem 6. *Define $B \in M_{n\times n}$ to be A^TA where all the non-zero elements are 1 and with the diaganol removed,*

$$B_{i,j} = \begin{cases} 0, & \text{for } i = j \text{ or } A^TA_{i,j} = 0 \\ 1, & \text{for } i = j \text{ and } A^TA_{i,j} \neq 0 \end{cases}$$

The total number of triangles with sum-weight is given by

$$\text{count with sum-weight} = \frac{1}{2}\ sum((AB)^{*2}) - \frac{1}{2}\ sum(AB)$$

where M^{*2} denotes squaring every element in the matrix M.

Proof. For each triangle we want to count the sum of its edge weights. Notice that for vertices $a, b, c \in V$, $B_{b,a}B_{c,b}B_{b,c}$ is 1 if and only if a, b, c forms a triangle. Therefore the following gives the total triangle count with sum-weight.

$$\sum_{a<b<c} ((A^TA)_{b,a} + (A^TA)_{c,b} + (A^TA)_{c,a}) \cdot (B_{b,a}B_{c,b}B_{c,a})$$

Instead of iterating over $a < b < c$ we can iterate over all a, b, c and divide by 6 because there are 3! permutations of a, b, c and each permutation contributes the same amount to the total sum.

$$= \frac{1}{6} \sum_{a,b,c \in V} ((A^T A)_{b,a} + (A^T A)_{c,b} + (A^T A)_{c,a}) \cdot (B_{b,a} B_{c,b} B_{c,a})$$

$$= \frac{1}{6} \left[\sum_{a,b,c \in V} (A^T A)_{b,a} (B_{b,a} B_{c,b} B_{c,a}) + \sum_{a,b,c \in V} (A^T A)_{c,b} (B_{b,a} B_{c,b} B_{c,a}) + \sum_{a,b,c \in V} (A^T A)_{c,a} (B_{b,a} B_{c,b} B_{c,a}) \right]$$

Each of the three summations are the same with variable changes.

$$= \frac{1}{2} \sum_{a,b,c \in V} (A^T A)_{c,b} (B_{b,a} B_{c,b} B_{c,a})$$

By construction of B, we know that $(A^T A)_{c,b} B_{c,b} = (A^T A)_{c,b}$ whenever $c \neq b$. When $c = b$, $B_{c,b} = 0$ so we can just remove that case from the summation.

$$= \frac{1}{2} \sum_{\substack{c,b \in V \\ b \neq c}} (A^T A)_{b,c} \sum_{a \in V} B_{b,a} B_{c,a}$$

From here, the algebra is exactly the same as in Theorem 5.

$$= \frac{1}{2} \operatorname{sum}((AB)^{*2}) - \frac{1}{2} \operatorname{sum}(AB^{*2})$$

In this case B is a matrix of only 1s and 0s. Therefore $B^{*2} = B$.

$$= \frac{1}{2} \operatorname{sum}((AB)^{*2}) - \frac{1}{2} \operatorname{sum}(AB)$$

□

4.3 Sum of a Matrix Product

To compute the number of triangles with product-weight or sum-weight, we need to compute $\operatorname{sum}(AB^{*2})$. In this section we provide an efficient $O(|E| + n)$ time and $O(1)$ space algorithm for computing the sum of any matrix product. Furthermore, $A^T A$ is a matrix product where each element is the weight of an edge. So calculating $sum(A^T A)$ gives us the total edge weights which can be used to get the average edge weight.

Theorem 7. *Let $A \in M_{m\times n}$ and $B \in M_{m\times p}$. Then*

$$sum(AB) = \sum_{k=1}^{n}\left[\left(\sum_{i=1}^{m} A_{i,k}\right)\cdot\left(\sum_{i=1}^{p} B_{k,j}\right)\right]$$

Proof. By definition of *sum* and matrix multiplication,

$$sum(AB) = \sum_{i=1}^{m}\sum_{j=1}^{p}(AB)_{i,j} = \sum_{i=1}^{m}\sum_{j=1}^{p}\sum_{k=1}^{n} A_{i,k}B_{k,j} = \sum_{k=1}^{n}\sum_{j=1}^{p}\sum_{i=1}^{m} A_{i,k}B_{k,j}$$

Since $B_{k,j}$ is independent of i, we can bring it out of the summation over i.

$$sum(AB) = \sum_{k=1}^{n}\left[\sum_{j=1}^{p} B_{k,j}\left(\sum_{i=1}^{m} A_{i,k}\right)\right]$$

Since $\sum_{i=1}^{m} A_{i,k}$ is independent of j, we can bring it out of the summation over j.

$$sum(AB) = \sum_{k=1}^{n}\left[\left(\sum_{i=1}^{m} A_{i,k}\right)\cdot\left(\sum_{i=1}^{p} B_{k,j}\right)\right]$$

□

A basic implementation of this theorem requires $O(|E|+|V|)$ time because each edge is traversed exactly once and the outer loop iterates over $n = |V|$.

4.4 Weighted Triangle Count Algorithms

In these algorithms we use Matlab notation denote columns and rows of a matrix. $A(i,:)$ denotes the ith row of A and $A(:,j)$ denotes the jth column.

Algorithm 3 Count triangles with product-weight

Input: Adjacency Matrix A

Output: Sum of all triangles' product-weights.

1: **procedure** COUNT TRIANGLES PROD-WEIGHT(Adjacency Matrix A)
2: Let B = strictly lower triangular part of $A^T A$.
3: Let $count = 0$
4: **for** $1 \leq j \leq n$ **do**
5: Let $\vec{b} = B(:, j)$.
6: $count = count + ||A\vec{b}||^2$
7: **for** $1 \leq k \leq m$ **do**
8: $count = count - sum(A(:, k) \cdot sum(B^{*2}(k, :))$
return $count/2$.

From Theorem 5, we know that the number of triangles with product weight is $\frac{1}{2}sum((AB)^{*2}) - \frac{1}{2}sum(AB)$. The second loop compute $sum(AB)$ using theorem 7. The first loop computes $sum(AB)^{*2}$ without storing the matrix $(AB)^{*2}$ by computing one column of AB at a time.

The runtime of this algorithm is dependent on the density of B. However, in the worse case when B is completely dense, the runtime is $O(|E||V| + |U||V|)$ because multiplying A and B takes worst case $O(|E||V|)$ time and the second loop requires $O(|U||V|)$ time in the worst case.

To count triangles with sum-weight, we can make nice optimizations and simplifications to decrease the storage requirements and runtime.

Algorithm 4 Count triangles with sum-weight

Input: Adjacency Matrix A

Output: Sum of all triangles' sum-weights.

1: **procedure** COUNT TRIANGLES SUM-WEIGHT(Adjacency Matrix A)
2: Let $count = 0$
3: **for** $1 \leq i \leq n$ **do**
4: Let $\vec{b} = B(:, i)$.
5: Let $\vec{c} = A\vec{b}$.
6: **for** nonzero $x \in \vec{c}$ **do**
7: $count = count + x(x - 1)$
return $count/2$.

Theorem 6 tells use how to count triangles by sum-weight. For each element x in AB we want to add $\frac{1}{2}(x^2 - x) = \frac{1}{2}x(x-1)$. Algorithm 4 does this by computing the columns of AB one at a time and adding $x(x-1)$ for each x in the column. Notice here that we only access B by its columns, and only one column at a time. This means that in Algorithm 4, we only need to store and compute one column of B at a time. This algorithm has the same runtime as the previous algorithm, Algorithm 4. However, because we only need each column of B once and only one at a time, the additional space required is only $O(|V|)$. So counting triangles with sum-weights requires less space than computing $A^T A$.

4.5 Approximating Triangle Count

We can use the triangle count with sum-weight and the average edge weight to approximate the actual number of triangles in $A^T A$.

$$\text{triangle count} \times 3 \times \text{avg edge weight} \approx \text{triangle count with sum-weight}$$

By definition, the edge weights of $A^T A$ are the elements of $A^T A$. Since $A^T A$ is a matrix product between A^T and A, we can apply Theorem 7 to get the total sum. Subtracting out the diaganol gives us all edges excluding loops. Finally, dividing by the number of non-zeros in $A^T A$ without the diaganol gives us the average edge weight. Algorithms such as those by Cohen [12] predict the structure and number of non-zeros in a matrix product to approximate the average edge-weight without computing $A^T A$. However, this method is not used here because we can compute the exact number without much more effort.

Algorithm 5 Approximating Triangle Count

Input: Adjacency Matrix A

Output: Approximate number of triangles in A^TA.

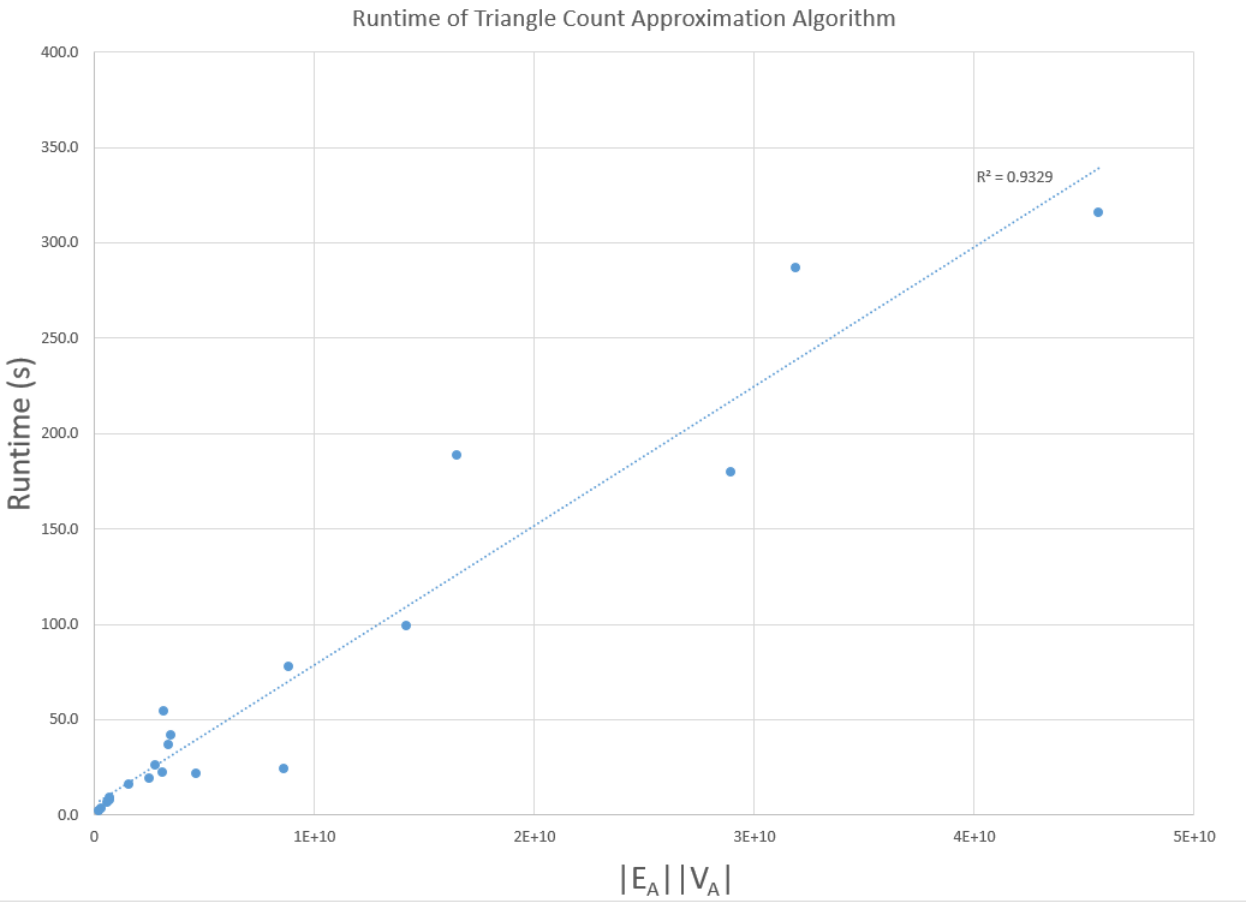
procedure APPROX TRIANGLE COUNT(Adjacency Matrix A)
    Let $count = 0$
    Let $nnzB = 0$.
    for $1 \leq i \leq n$ **do**
        Let $\vec{b} = B(:, i)$.
        $nnzB = nnzB + nnz(\vec{b})$
        Let $\vec{c} = A\vec{b}$.
        for nonzero $x \in \vec{c}$ **do**
            $count = count + x(x - 1)$
    Let $sumWeights = 0$
    for $u \in U$ **do**
        $sumWeights = sumWeights + [deg_A(u)]^2$
    for $v \in V$ **do**
        $sumWeights = sumWeights - deg_A(v)$
    return $count/(2 * sumWeights/nnzB)$.

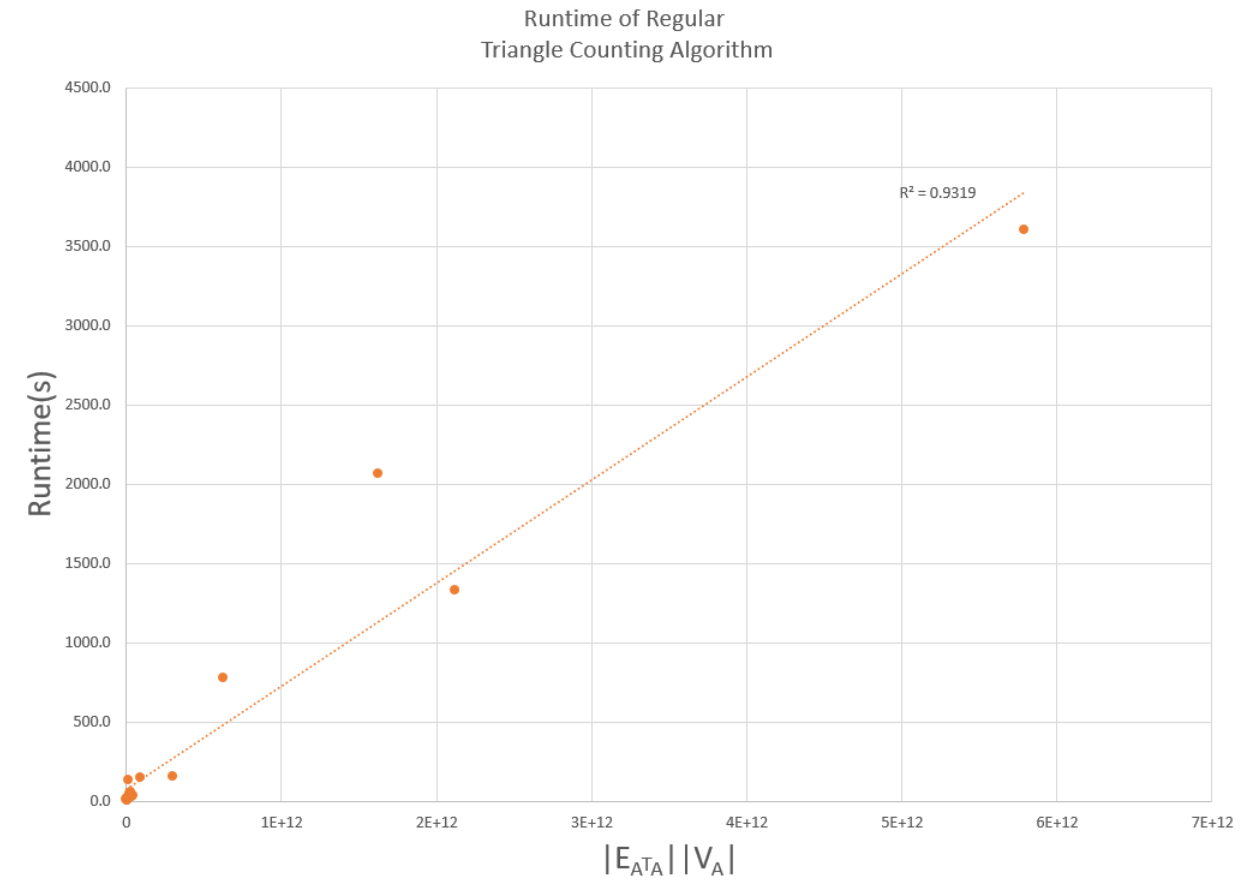
Algorithm 5 first computes the sum-weighted triangle count as in algorithm 4. Then it computes the average edge weight by first computing the total edge weight by calculating $sum(A^TA)$ using Theorem 7. In this case, it is equivalent to adding the sum of the degrees of $u \in U$ squared. We subtract the diaganol because we are not interested in loops and divide by the number of non-zeros in B to get the average edge weight.

4.5.1 Results

Since A^TA can be dense even though A is sparse, we store A^TA and its variations as a dense matrix in our implementation. The code is optimized for worst case behavior that gives the estimation algorithm the most benefit, when A^TA is dense. When A^TA is sparse, the estimation algorithm is not as beneficial and computing the exact number of triangles is feasible. Where the estimation algorithm is useful is when A^TA is not sparse and computing the number of triangles is infeasible or storing the matrix A^TA is too costly.

As a comparison, we ran the triangle estimation algorithm, Algorithm 5, against a regular triangle counting algorithm in $A^T A$. The regular algorithm computes the upper triangular portion of $[U(A^T A) \cdot L(A^T A)].*A^T A$ using a masked matrix multiplication similar to the algorithm described by Azad et al [1], except $A^T A$ is stored as a dense matrix. Here .* denotes element wise matrix multiplication.





Below is a table of all the data and results gathered.

$\|U\|$	$\|V\|$	$\|E\|$	$\|E_{A^TA}\|$	Estimate(s)	Regular(s)	Error %
4.41E+03	1.13E+04	2.86E+04	2.06E+05	3.4	4.0	9.6
5.88E+04	1.18E+04	2.35E+05	7.06E+05	25.6	20.1	0
1.08E+04	3.37E+04	1.01E+05	8.84E+06	36.5	154.5	Overflow
4.40E+03	1.68E+04	1.50E+05	2.63E+06	19.0	32.8	Overflow
1.44E+04	2.77E+04	5.83E+04	2.44E+05	15.8	18.6	1019
1.57E+04	1.57E+04	4.70E+04	2.98E+05	7.8	8.1	0.2
5.20E+04	1.39E+04	6.24E+05	8.32E+05	24.4	19.1	0
1.01E+04	1.64E+04	4.48E+04	1.76E+05	9.0	8.1	43.6
4.56E+03	5.76E+03	2.46E+06	1.64E+07	99.5	147.8	Overflow
3.46E+05	1.23E+04	1.33E+06	1.05E+06	188.4	131.3	6.7
7.20E+04	2.70E+03	1.15E+06	1.18E+05	22.6	11.5	18.8
4.73E+04	8.90E+03	3.56E+05	2.11E+06	54.3	49.2	4.9
1.00E+03	8.81E+03	2.78E+04	5.61E+05	2.4	4.9	6.5
8.25E+02	8.63E+03	7.08E+04	4.36E+06	6.6	23.6	Overflow
1.21E+05	2.37E+04	1.47E+05	2.57E+05	41.9	21.5	1.2
3.16E+03	1.59E+04	2.87E+06	1.02E+08	315.7	2063.6	Overflow
1.18E+05	1.88E+04	4.70E+05	1.41E+06	77.6	60.7	0
4.68E+04	2.66E+04	1.20E+06	2.18E+08	286.5	3599.8	Overflow
1.20E+02	1.29E+04	3.60E+05	1.65E+08	21.9	1329.5	Overflow
3.02E+04	2.79E+04	1.04E+06	2.25E+07	180.0	775.5	Overflow

4.5.2 Advantages and Disadvantages

We can see that the estimation time was usually faster or around the same amount of time as computing the exact number of triangles. Since the run-time of the estimation algorithm is dependent on $|E|$ and $|V|$ in A, the run-time is more predictable. The regular algorithm's runtime is dependent on $|E_A^T A|$. The edges of $A^T A$ are heavily dependent on the structure of A and cannot be determined by $|U|, |V|$, or $|E|$ or A. Therefore, the regular algorithm sometimes runs over 10 times longer than the estimation algorithm. Furthermore, the regular algorithm requires significantly more memory since the $A^T A$ is stored but in the estimation algorithm, only one column is stored at a time.

However, the triangle estimation algorithm does not have any bounds on its error. While most of the data tested had an error of below 50% when overflow did not occur, there was one test case where the error was over

1000%. There is no upper bound to the error as the algorithm is currently written other than the largest weight of any edge in $A^T A$.

While the triangle estimation algorithm require significantly less memory, $O(|V|)$ to run and have more consistent run times, the unbounded error makes it not particularly useful. However, the sum-weight and product-weight algorithms which have similar run time may have practical use.

References and Further Reading

[1] Ariful Azad, Aydin Buluc, and John Gilbert, *Parallel Triangle Counting and Enumeration using Matrix Algebra*, Workshop on Graph Algorithms Building Blocks, 2015.

[2] Alan George and Micahel T. Heath, *Solution of sparse linear least squares problems using givens rotations* , Linear Algebra and its Applications, Volume 34, pp.69-83, December 1980.

[3] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen, *What Color is Your Jacboian? Graph Coloring for Computing Derivatives*, SIAM Rev, pp.627-705, August 2006

[4] C. Seshadhri, Ali Pinar, and Tamara G. Kolda, *Wedge Sampling for Computing Clustering Coefficients and Triangle Counts on Large Graphs*, Statustics Analysis and Data Mining, Vol. 7, No. 4, pp.294-307, August 2014.

[5] Edith Cohen, *Structure Prediction and Computation of Spares Matrix Products*, Journal of Combinatorial Optimization 2, 307-332, 1999.

[6] Tim Davis, John R. Gilbert, Stefan I Larimore, and Esmond G. Ng, *A column appropriate minimum degree ordering algorithm*, ACM Transactions on Mathematical Software (TOMS), Volume 30 Issue 3, pp.353-376, September 2004.

[7] Gene H. Golub, and Chen Greif *Techniques For Solving General KKT Systems*, 2000.

[8] Grey Ballard, Ali Pinar, Tamara G. Kolda, and C. Seshadhri, *Diamond Sampling for Approximate Maximum All-pairs Dot-product (MAD) Search*, arXiv:1506.03872

[9] J. R. Gilbert, X. S. Li, E. G. Ng, B. W. Peyton, *Computing Row and Column Counts for Sparse QR and LU Factorization*, BIT Numerical Mathematics, Volume 41 Issue 4, pp 693-710, September 2001.

[10] Mark Ortmann and Ulrik Brandes, *Triangle Listing Algorithms: Back from the Diversion*, 2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments.

[11] Jeremy Kepner, and John Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2011. Print.

[12] Jonathan Cohen, *Graph Twiddling in a MapReduce World*, Journal Computing in Science and Engineering, Vol. 11, No. 4, pp 29-41, July 2009.