

CSE 101 Homework 2

Kreshiv Chawla, Brian Masse, Taira Sakamoto, Emily Xie, Annabelle Coles

October 14, 2025

1.

1. $V = \{A, B, C, D, E, F\}$

edges: $\{(A, B), (A, C), (A, F), (B, D), (B, C), (C, D), (C, E), (D, E), (E, F), (F, C)\}$

(a)

(A, B)	(B, A) ✓
(A, C)	(C, A) ✓
(A, F)	(F, A) ✓
(B, D)	(D, B) ✓
(B, C)	(C, B) ✓
(C, D)	(D, C) ✓
(C, E)	(E, C) ✓
(D, E)	(E, D) ✓
(E, F)	(F, E) ✓
(F, C)	(C, F) ✓

A: A
B: A
C: A, B, F
D: B, C
E: C, D
F: A, E

(c)

A: B, C, F
B: C, D
C: D, E
D: E
E: F
F: C

descending post number
12, 11, 10, 9, 4, 2
C, F, E, D, B, A

(b)

nodes	pre	post
A	1	2
B	3	4
C	5	12
D	8	9
E	7	10
F	6	11

(d) strongly connected components

- every node in $\{C, D, E, F\}$ can reach every other node.
- no back edges to A or B

$\{C, D, E, F\}, \{B\}, \{A\}$

2. Algorithm.

```

count ← 0;
for each unordered edge  $\{u, v\}$  with  $A[u][v] = 1$  do
     $S \leftarrow \{w \in V : A[u][w] = 1 \text{ and } A[v][w] = 1\}$ ;
     $t \leftarrow 0$ ;
    for each unordered pair  $\{x, y\} \subseteq S$  do
        if  $A[x][y] = 1$  then
             $t \leftarrow t + 1$ ;
        end
    end
    count ← count + t;
end
return count/6;

```

Correctness.

Base Case: If all 6 edges are present, there is exactly 1 4-clique. For each edge (u, v) , the common neighbor set S has the other 2 vertices. Those two vertices are adjacent, so we add 1 for each of the 6 edges. Total count = 6; dividing by 6 gives 1, which is correct.

Inductive Hypothesis: Assume the algorithm correctly counts all 4-cliques in any undirected graph with $n = k$ vertices.

Inductive Step: Consider a graph G with $k + 1$ vertices. Let v_{k+1} be the newly added vertex. We can divide 4-cliques in G into two types:

- (a) 4-cliques not containing v_{k+1} : These exist entirely in the subgraph with k vertices. By the inductive hypothesis, the algorithm counts them correctly.
- (b) 4-cliques containing v_{k+1} : Suppose v_{k+1} is part of some 4-clique $\{v_{k+1}, a, b, c\}$. For each edge among these 4 vertices, the intersection $N(a) \cap N(b)$ includes both v_{k+1} and c , and since they are connected, the inner loop adds 1. Each 4-clique contributes 6 such additions overall (once per edge). Thus, each new 4-clique is included and counted 6 times. Dividing the final sum by 6 again gives the correct total number of 4-cliques.

Therefore, the algorithm correctly counts all 4-cliques in any undirected graph.

Time Complexity.

For each edge $\{u, v\}$, finding $S = N(u) \cap N(v)$ costs $O(|V|)$. Checking all pairs inside S costs $O(|S|^2)$. Summed over all edges, the total time is $O((|V| + |E|)^2)$.

3. Let G be a directed graph that is not strongly connected. We want to make it strongly connected by adding a new vertex u and as few edges as possible from u to vertices in G and from vertices in G to u .

- (a) Can we ever make G strongly connected by adding u and a single edge in or out of u ? Explain your answer.

No. Strong Connected requires that $\forall u \in V$ there is a path to and from u . If we add one edge from u , it becomes a source and there is no way of reaching u . If we add an edge to u , it becomes a sink and there is no way of leaving u . Thus G will not be strongly connected.

- (b) Give an example of a directed graph with more than one strongly connected component where we can make it strongly connected by adding u and two edges in or out of u ?

$$G : (A, B), (B, C), (C, A), (C, D), (D, E), (E, F)$$

Add the vertex u and edges $(D, u), (u, C)$ to make G strongly connected.



- (c) Give a characterization (an if and only if condition) of the minimum number of edges we must add, in terms of the strongly connected components of G .

The minimum number of edges E_m is given by

$$E_m = S + K$$

where K is the number of sink SCCs, and S is the number of source SCCs.

- (d) Describe how to use an algorithm from class to compute this number.
- Run the Tarjan-Koseraju algorithm to find all SCCs within the graph.
 - Check whether each SCC is a sink in G by running a DFS algorithm on it. If a node outside the SCC is reachable, it is not a sink. If the only nodes that are reachable are those in the SCC, it is a sink.
 - Repeat items i, ii for the reverse graph to find its sinks, and thus the regular graph's sources.

(e) How long would this algorithm take to do this? Explain.

The algorithm runs in $O(|V|(|V| + |E|))$

- i. The two Tarjan-Koseraju algorithms run in $O(|V| + |E|)$ time.
- ii. Each DFS call runs in $O(|V| + |E|)$, and is run for every vertex at most twice.
- iii. Thus the total algorithm takes $O(|V|(|V| + |E|))$

4. **Experiment:** Programming language: Python 3.11.3. Packages: Numpy (1.24.4), Python time module, Matplotlib (3.7.1). We ran the code in a Jupyter notebook. To implement the two algorithms, we wrote two functions *find_triangle(n)* and *count_triangle(n)*. Both functions take a single input parameter, n , the number of vertices in the graph. The *find_triangle(n)* returns a boolean, True if a triangle is found, and False if no triangles. The *count_triangle(n)* returns an integer, zero if no triangles, or greater than zero if at least one triangle is found.

Function Summary:

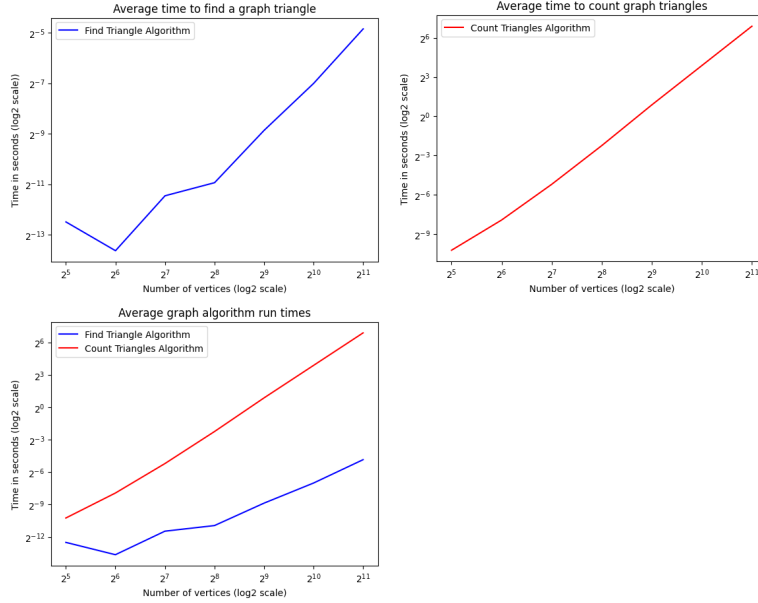
Within each function call, it generates a random graph, represented as an adjacency matrix. We used a Numpy array to store an adjacency matrix with shape (n, n) . We used the Numpy's randomization function to fill the array elements with either 0 or 1. 1s represent edges between vertices and 0s represent no edge. Numpy's randomization function will fill the array randomly so it will not be symmetrical. Since the graph is undirected the adjacency matrix must be symmetric. Therefore to make the matrix array symmetric, we take the top diagonal half, and replace the bottom diagonal half with the transposed top half. Also, we fill the diagonal with 1s as this represents vertex self edges.

In both algorithms, we used a nested for loop to find vertices (represented as array indices) to find rows and columns with 1s. If the element at array index *adjacency_matrix*[u][v] == 1, there exists an edge between (u, v) and (v, u) since the graph is undirected. Only if a (u, v) edge is found, we then used a third for loop to loop through the array indices from positions in *range*($v + 1, n$) (Python range function is non-inclusive of n). We can start the loop at index $v + 1$ because if there exists a w index where $w < v$, this would have been found when $u' < u$. Similarly, we can start the v for loop at index $u + 1$ and we do not consider self-loops. If and only if there exists all three edges (u, v) and (u, w) and (v, w) , we have identified a triangle.

In the *find_triangle(n)* algorithm, a triangle has been identified, the function terminates and returns True. If the outer for loop finishes and no vertices that satisfied the condition were found, the function returns False. Therefore, the function will terminate after the first triangle is found. In the *count_triangle(n)* algorithm, a counter variable, initialized to zero, is used to keep track of the number of triangles in the graph. If a triangle has been identified, the counter is incremented by a value of one. After this, the loops will continue to run until the outer loop ends. The function returns the counter variable – the total number of triangles in the graph.

Experiment summary: To calculate the average run time for each input n , we ran each algorithm 15 times which used a random graph of size n for each run. We used the Python time module to record the run time of the algorithm for every run and then calculated the average run time as the mean of the 15 run times. We repeated this for each n . We did this for both algorithms.

Output graphs: To generate graphs, we used Matplotlib to plot the mean run times as a line plot. The y-axis is the log2 scaled algorithm run time in seconds. The x-axis is the number of vertices, also log2 scaled.



Conclusions:

The theoretical worst-case for the finding triangle algorithm is the worst-case run time of the counting algorithm. The counting algorithm has the same average and worst-case run times. From our experiments, the average run time of the finding algorithm was always shorter than the run time of the counting algorithm. Looking at the finding algorithm (blue line) and the counting algorithm (red line), we can see that both algorithm run times have a positive slope increase as n increases. Additionally, looking at the graph with both algorithm run times plotted, we can see that the counting algorithm has much higher run times (higher values on the y-axis). From the graphs, we can see that the average run times are closest when n is small, and as n increases, the difference between the algorithm run times becomes greater. From our experiments we can conclude that if there exists at least one triangle in the graph, the finding algorithm will always have a shorter run time as it will terminate sooner.