Nick Monell, Nick Nguyen, Kersten Ebdane, Khoa Le, Ouyang Yingxuan

CSE101: Design and Analysis of Algorithms (CSE, UCSD, Spring-2022)
Homework-02
Due Thursday April 17, 11:59 PM

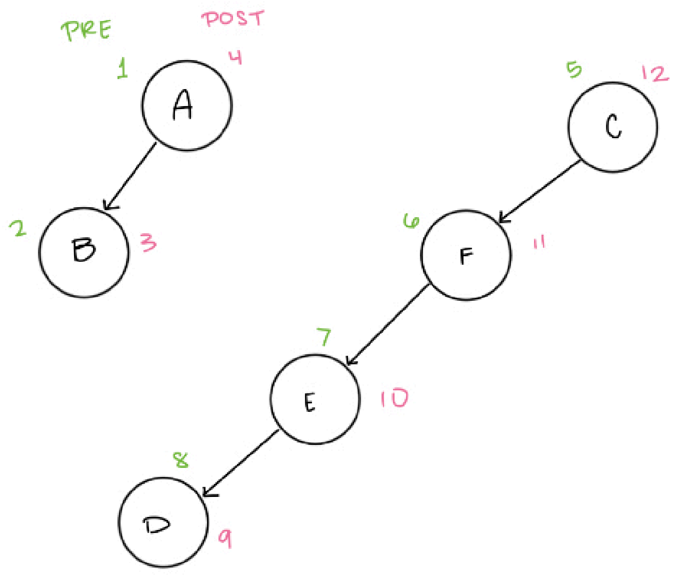
Graded problems

1. For this problem, you can draw your answer rather than use a word processing tool. This is a short answer question, so no proofs are required. Consider a directed graph with vertices $V = \{A, B, C, D, E, F\}$ and edges $\{(A, C),(A, E),(A, F),(B, A),(B, C),(C, D),(C, E),(D, E),(E, F),(F, C)\}$. The goal is to illustrate Tarjan's strongly connected components algorithm on this graph.

   a. Give the adjacency list for the reverse graph. (5 points)

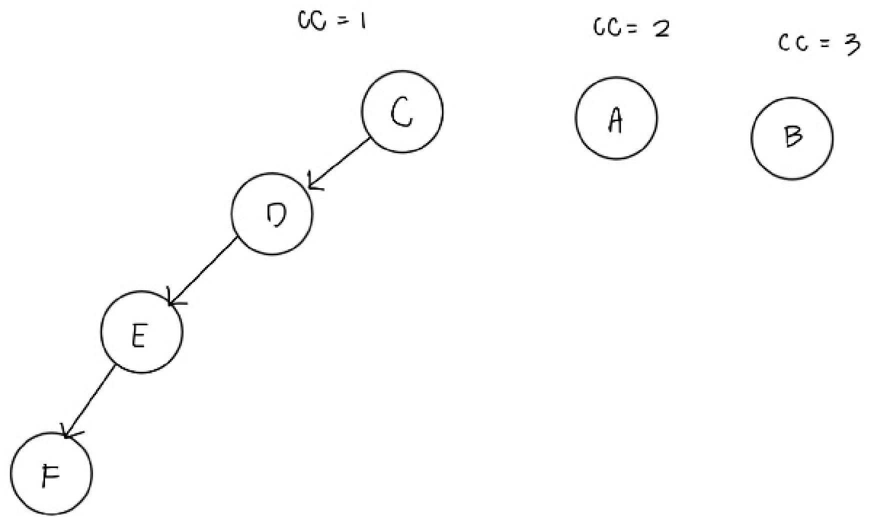
Vertex	
A	→B
B	—
C	→A, B, F
D	→C
E	→A, C, D
F	→A, E

   b. Show a DFS tree for the reverse graph, with pre and post numbers. (5 points)

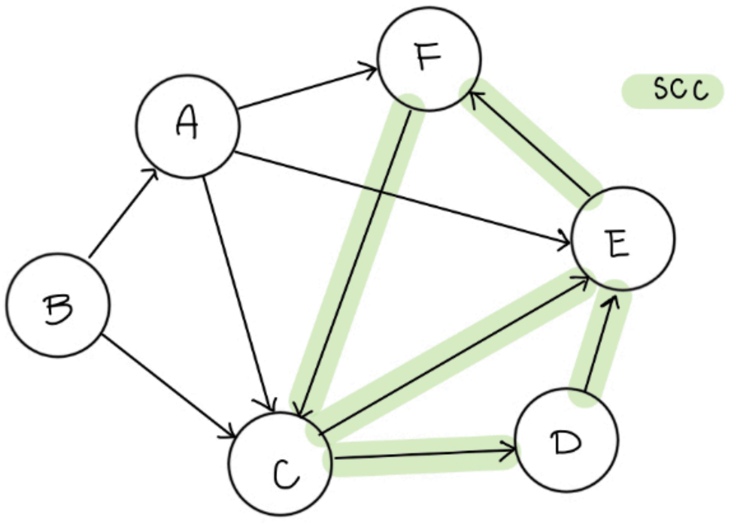


Nick Monell, Nick Nguyen, Kersten Ebdane, Khoa Le, Ouyang Yingxuan

c. Show a DFS tree for the original graph, ordered by descending post number above. (5 points)



d. Use the above trees to give the strongly connected components of the original graph. (5 points)



$SCC_1 = \{C, D, E, F\}$

$SCC_2 = \{A\}$

$SCC_3 = \{B\}$

2. Give an $O(|V| + |E|)$ time algorithm that, given an undirected graph G in adjacency list format, computes the number of paths of length 2 in G. Here a path of length 2 involves two edges *{u, v}*, *{v, w}* where *u != w*. We view $u --> v --> w$ as the same path as $w --> v --> u$. (20 points. 5 for clear algorithm description, 5 for correctness proof, 5 for efficiency, 5 for time analysis.)

Correctness proof:

Loop invariant:

At the beginning of each iteration, variable total holds all the valid paths of length 2 (w,v,u) where v is the middle vertex and w and u are distinct outgoing edges of v:{w,v},{v,u}. And the paths are not double-counted where (w,v,u) and (u,v,w) are the same.

Base:
Total variable holds the number of path that has length 2 in G. When the loop hasn't started, it is vicariously true that the total valid path in G is 0.

Maintenance:
Assume after t iterations, total holds the correct number of paths of length 2 in G.
At t+1 iteration, we are at vertex v, and v has d outgoing edges. If we pick 2 distinct edges, u and w, from any of the v outgoing edges, then we have a valid path (w,v,u) where v is in the middle. Following this logic, we have d choose 2 valid paths when we are at v. Next, we add the combinatorics result to the total. After this step, the total reflects the true number of valid paths in G after t+1 vertices.

Termination: When we finish all iterations, total holds the correct number of paths of length 2 in G

```
Def numPathWith2Edges(G):
total=0
For u in adjacency_list:
        d= len(u.neighbors)
        total+=d choose 2
```

Time complexity:
Traverse all vertices O(V), then each vertex has a list of neighbors, and by summarising all neighbors, we will have 2E. Therefore, run time is O(V+2E) = O(V+E)

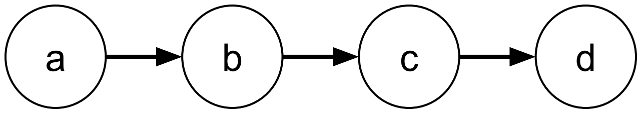
3. You can handwrite answers for this question. Some parts require short proofs, but the examples do not. Say that for a directed graph *G reachability is symmetric* if, whenever there is a path from *u* to *v*, there is a path from *v* to *u*

   a. Are there any undirected graphs where reachability is not symmetric? Give an explanation. (3 points)

      There are no undirected graphs where reachability is not symmetric. Take for example, the undirected graph a-b-c-a, where the three vertices form a strongly connected component, or even a-b-c where c is not connected to a via another edge. The edge {a, b} can be traversed in reverse as {b, a} as it is an undirected edge. Similarly, {b, c} can be traversed {c, b}, meaning any path {a, c} can be followed in reverse to find a path {c, a}. Because of the symmetry of each individual edge, we can draw symmetry between each individual path, meaning reachability from vertex a to any vertex can similarly be seen from any vertex to vertex a.

   b. Give an example of a directed graph where reachability is not symmetric. (3 points)

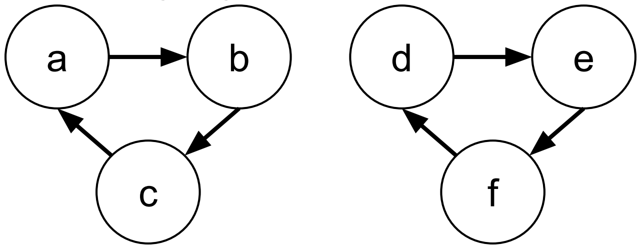
      A directed graph where reachability is not symmetric would be a->b->c->d, shown in the following diagram:



We can follow edge {a, b}, {b, c}, and {c,d} to subsequently find a directed path from a to d, however d has no outgoing edges. Because vertex d is a sink, there is no possible way we can find a path from vertex d to vertex a, or any vertex for that matter. Because there exists a path {a, d} but no path {d, a}, reachability in this directed graph is not symmetric.

c. Give an example of a directed graph with more than one strongly connected component where reachability is symmetric (3 points).

A directed graph with more than one strongly connected component where reachability is symmetric could be a graph with two components and no edges connecting them a->b->c->a, d->e->f->d, shown in the following diagram:

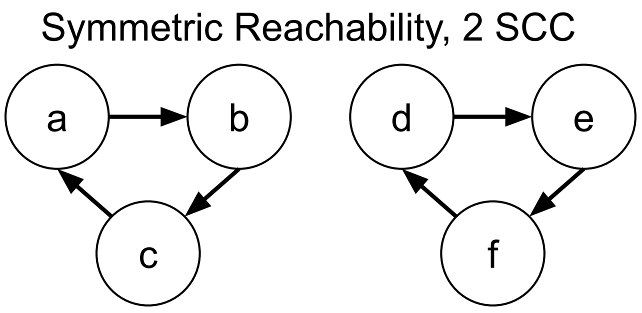


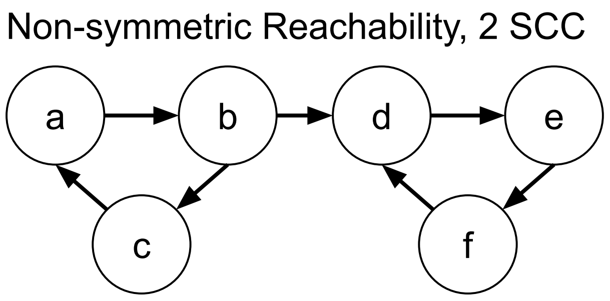
Here, we can see that there is an existing path from every vertex in the SCC {a, b, c} to every other vertex in the {a, b, c} component, in both forward and reverse directions. There exists a path from a to c a->b->c much like there exists a path from c to a, c->a, therefore graph reachability is symmetric considering the first SCC. Looking at SCC {d, e, f}, we can see that it has the exact same structure as {a, b, c} with different vertices, so it must have similar symmetric reachability between its 3 comprising vertices. Therefore, we can see that there is symmetric reachability within each of the two strongly connected components. There are no edges between the two SCC, therefore there is no reachability between any vertex {a, b, c} and any vertex {d, e, f}. Because we cannot reach vertices from one SCC to the other SCC, we still follow symmetric reachability rules as there is no possible path {u, v} that does not also have a path {v, u}.

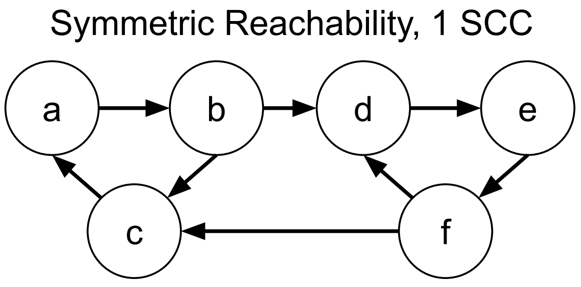
d. Give a characterization (an if and only if condition) of when reachability is symmetric in terms of the strongly connected components of *G*. (4 points)

Reachability is symmetric in G if and only if there exists no edges between strongly connected components and any other vertex of G. Consider some strongly connected component C_1 that consists of vertices connected by directed edges. Because a strongly connected component obeys the property that if any path {u, v} exists, {v, u} also exists, we know that reachability is symmetric within any and all SCC. If we introduce any edge connecting this SCC to another vertex, or another SCC, we introduce additional reachability between nodes.
Consider our previous example (3c). If we introduced an additional directed edge {b, d}, there would exist a path from any {a, b, c} to any {d, e, f}, but no such path from any {d, e, f} to any {a, b, c}, eliminating symmetry in reachability. If we then added an additional edge {f, c}, we would then have the ability to find any path from {a, b, c} to any {d, e, f} and any {d, e, f} to any {a, b, c}, restoring symmetric reachability. However, though we have now added an edge to both of our SCC, we have now

created a symmetric path between each and every node in the graph such that with any path {u, v,} exists a path {v, u}. Therefore, the entire graph has become a single SCC, and our property that no edges exist between strongly connected components and any vertex of G is satisfied once again. If u is reachable from v and v is reachable from u, they must be within the same SCC.
Because symmetric reachability was only restored once our characterization was satisfied, and was lost once we added an additional edge to strongly connected components, we can say that the characterization is correct. The following graphs illustrate the above explanation:







e. Describe how to use an algorithm from class to decide whether reachability in *G* is symmetric. (4 points)

An algorithm that we can use to decide whether reachability in G is symmetric is a strongly connected components algorithm which makes use of two DFS passes.
We have established that reachability in G is symmetric if and only if there exists no edges connecting any SCC in G to another vertex which is not part of the SCC. Therefore, to determine whether or not reachability is symmetric, all we must do is identify the SCC and then check if there is a path from any given node in each SCC to any given node in another SCC. If there is, we know that reachability is not symmetric (if it was, there would be a path from the second SCC to the first, thus closing the loop and making them part of the same SCC which would be identified in the initial algorithm to locate all SCC).

To find all SCC, we employ the SCC algorithm:

- Run DFS on the reverse of G, G^R, keep track of post-numbers where each vertex is visited, these are the source vertices of each SCC in G.
- Run DFS on G, ordering vertices based on the decreasing order of post-numbers from the previous step. Because we have generated the post-numbers of the reversed graph, the largest post-number should be visited first. This allows us to start at the source of each SCC, meaning we should capture each vertex in the SCC before exiting with DFS.
    - We will periodically increase the component counter after starting at a new vertex, indicating that we have located a new SCC which we keep track of.
- Generate a meta-graph of all identified SCC, meaning each SCC becomes its own node. For example, if we have nodes {a, b, c} forming a SCC, we will treat them as a single vertex abc. Maintain any edges in the meta-graph that connect SCC if they exist (if any edge from a vertex

in SCC_1 connects to any vertex in SCC_2, create an edge $\{SCC_1 , SCC_2\}$.
- If there are any edges in our meta-graph, between any pair of SCC, we do not have symmetric reachability, otherwise, we do.

The key is the last bullet point. We know that the graph does not have symmetric reachability if there exists an edge between any two SCC as determined by the characterization in part (3d) as it leads to a path that can only be traversed one way (or else they would not be two separate SCC).

f. How long would this algorithm take to do this? Explain. (3 points)

This algorithm can be performed in linear time complexity, $O(|V| + |E|)$.
The reason for this is because each individual step has at most linear time complexity:
- DFS on G^R can be done in $O(|V| + |E|)$, assuming our data is stored in an adjacency list
- DFS on G can be done in $O(|V| + |E|)$, same as G^R
- Creating a meta-graph (decomposition) can be done in linear time
- Checking if there are any edges in the meta-graph can be done in linear time (one check per SCC vertex)

Adding all time complexities gives us a maximum of linear time $O(|V| + |E|)$.

4. Give an algorithm to decide, given a directed graph G in adjacency list format, whether there is a vertex v from which all vertices in the graph can be reached. Your algorithm should be considerably faster than $O(|V|(|V| + |E|)$. You can use algorithms from class without explanation. (20 points. 5 for clear algorithm description, 5 for correctness proof, 5 for efficiency, 5 for time analysis.)
Algorithm:

```
G = adjacency list of vertices
visited = visited array initialized with false
pre = empty array containing pre-number
post = empty array containing post-number
count = 1, countVertices = 1
sourceVertex = empty

dfsWithPrePost(G, v) => O(V + E)
    visited[v] = true
    pre[v] = count++
    for u in adjacency_list[V]
        if !visited[u]
            dfsWithPrePost(G,u)
    post[v] = count++
    sourceVertex = v

dfsCheckSource(G, sourceVertex) => O(V + E)
    visited[sourceVertex] = true
    for u in adjacency_list[sourceVertex]
        if !visited[u]
            dfsCheckSource(G, u)
            countVertices++
reachAllVertices()

    For v in adjacency_list => O(V + E)
        If !visited[v] => O(1)
            dfsWithPrePost(G,v) => O(V+E)

    visited = array re-initialized with false => O(1)

    dfsCheckSource(G, sourceVertex) => O(V + E)

    if countVertices == |V| => O(1)
        return true => O(1)
    else
        return false => O(1)
```

Algorithm Description:

The algorithm iterates through the adjacency list and performs a DFS to set the pre- and post-numbers for vertices, then it runs another DFS through all the neighbors of the source vertex. After setting the pre- and post-number, the algorithm resets the visited array to perform another DFS using the visited array and finds the source vertex that has the highest post-number. The second DFS explores the vertices reached by the source vertex and counts them. Lastly, the algorithm checks if the source vertex can reach all other vertices by comparing the countVertices found by the second DFS with the total number of vertices. If not equal, meaning there's a vertex that the source vertex cannot reach, return false. Otherwise, return true.

Proof Correctness:

Case 1: If there's a vertex that can reach all the other vertices, the algorithm returns true.
Proof by Induction:
Base case: Let v be the only vertex in the graph. The vertex v will have the highest post-number and be the source vertex. Then, explore the list of vertex v; the counter of vertices reachable from v increments by 1. countVertices = 1 = the number of vertices in the graph, the algorithm returns true.
Induction:

- Induction Hypothesis: Assume there are n vertices from v_0 to v_n reachable from v, given that the algorithm returns true. Let v_(n+1) be added to any vertices from v_0 to v_n. Want to show that v_(n+1) is also reachable from v.
- Inductive Step: As v_(n+1) is connected by any vertex from v_0 to v_n, v_(n+1) is a neighbor of any vertex from v_0 to v_n and reachable by any vertex that it's connected to. Because any vertex from v_0 to v_n is reachable from v, and v_(n+1) is reachable from the vertex it's connected to, v_(n+1) is also reachable from v. As v_(n+1) is reachable, the countVertices increments by one which equals to the total number of vertices after adding v_(n+1) into the graph.

Conclusion: Because countVertices equals the total number of vertices, the algorithm returns true.

Case 2: If there's no vertex that can reach all the other vertices, the algorithm returns false.
Direct Proof:
Let v be the vertex that can reach all the other current vertices from v_0 to v_n. Add vertex u as the vertex that cannot be reached from v in the graph.
Going through DFS with pre/post, u or v can have the largest post-number.
If u is the source vertex, when using the DFS to check for the reachability of u to other vertices, countVertices only equals 1 when the DFS finishes, which is smaller than the total number of vertices. Thus, the algorithm returns false.
If v is the source vertex, when using the DFS to check for the reachability of v to other vertices, n vertices can be reached, except u, as the countVertices = n, which is smaller than the total number of vertices. Thus, the algorithm returns false.
To conclude, if there's no vertex that can reach all the other vertices, the algorithm returns false.

Time Complexity:
The first loop, including DFS, runs O(V + E)
The second DFS counting the number of reachable vertices runs O(V + E)
The remaining statements are O(1)
Overall, the time complexity is O(V + E)

5. Last homework, you came up with an $O(|V|(|V| + |E|)$ algorithm to decide whether an undirected graph had a triangle. In this homework, you will run an experiment to see what the difference between this *worst-case* time and the *average-case* time of your algorithm are. Implement the algorithm in any programming language. Then first generate a large number of random graphs G where for each pair $\{u, v\}$, there is an edge between u and v with probability 1/2, independently for each possible edge. Try this for a number of vertices n a different number of powers of 2,e.g, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384 Run your algorithm on random graphs of each size, and plot the logarithm of the average time your algorithm takes vs. the logarithm of n (5, 6, 7, 8, ...14 for the example above.) Then generate random *bipartite* graphs with the same number of vertices, where there are two equal sized halves, random edges between halves, and no edges within the same half, and plot the same data. What is your conclusion? How do these two plots compare to the theoretical worst-case analysis? What characterizes instances where the algorithm is faster than its worst-case? (5 points for clear description of the experiment you ran, with information about PL used, programming environment, test graph generation and so on. 10 points for clear display of data from the experiment, including clear labelling of axes in graphs. 5 points for discussing conclusions to be drawn from the experiment.)

Here is the experiment description:
PL: Python 3.11
Programming environment:
Lappad with Intel Core, 16GB RAM, Windows system.
Algorithm:
There is a method called identify_Triangle to check if there is any triangle in the graph in the form of an adjacency matrix.. It iterates pairs of vertices v and w to see if they're connected (A[v][w] = 1). For each connected pair, it checks every other vertex u to see if u is connected to both v and w (A[u][v] = 1 and A[u][w] = 1). If it finds such a u, it returns True. Triangle exists. Else, it returns False.
Test random graph generation:
There is a method called random_graph_Generate to create a random graph with n vertices, as an adjacency matrix. There is a number n as input and then the method makes an n by n matrix A

filled with 0s. For each possible pair of vertices i and j (i < j), first it generates a random number between 0 and 1, if less than 0.5, it puts a 1 in A[i][j] and A[j][i]. The result is a random graph where about half the possible edges exist, which will be returned.

There is another method called bipartite_graph_Generate, same with n vertices, stored as an adjacency matrix. After generating matrix A of 0s. It splits the vertices into two equal groups, then only adds edges between these groups, not within them. For each pair of vertices where one is in the first group and the other in the second, it generates a random number from 0 to 1, if it's less than 0.5, it sets A[i][j] = 1 and A[j][i] = 1 for an edge. This ensures no triangles are possible, as edges only connect the two groups.

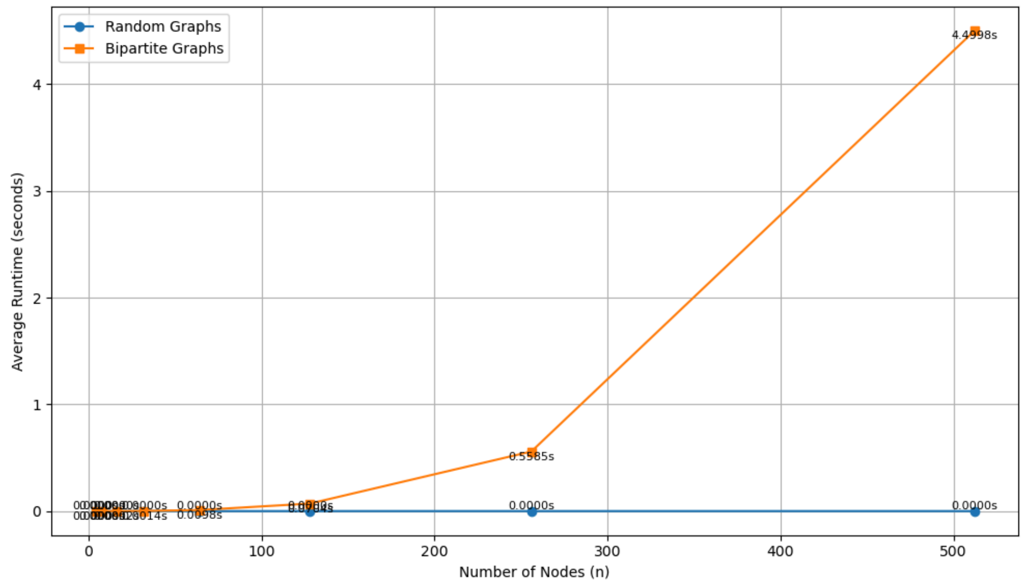
Experiment running:

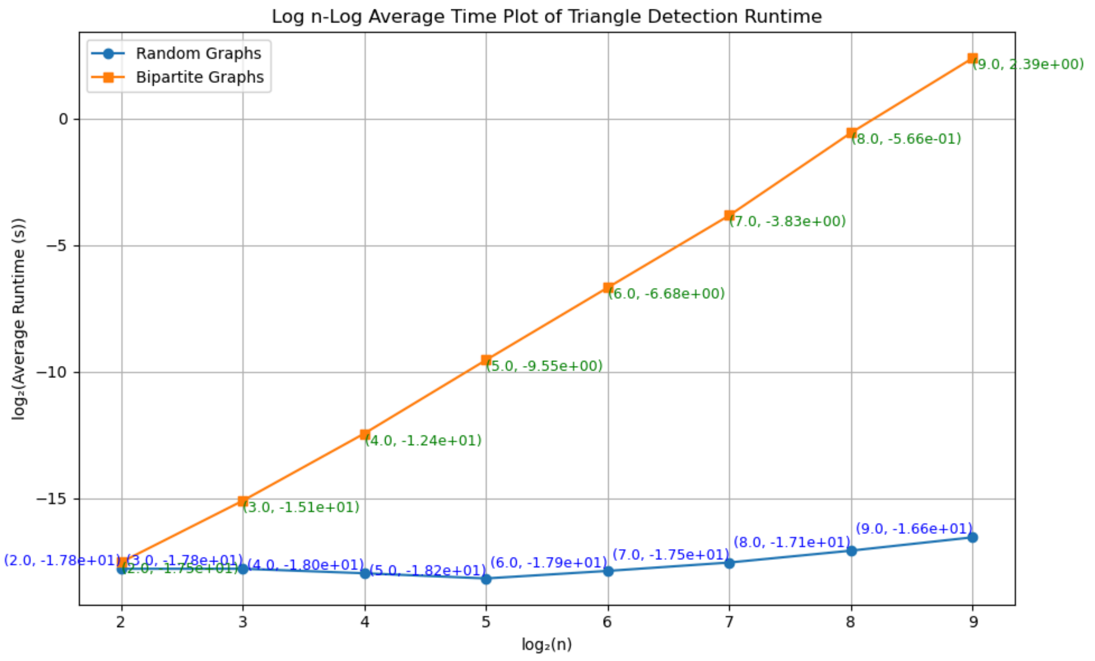
After setting the parameters like n for size and trail times, we use a core loop to test the runtime. The loop iterates over a list of graph sizes, use the methods above to generate multiple graphs of each size to measure the average runtime of the method for identify_Triangle algorithm. The runtime of algorithm is measured for each graph using time.perf_counter() to record the start and end times, and these times are stored in random_trial_times and bipartite_trial_times lists. After completing the trials, the average runtime will be calculated using np.mean and stored.

Plot Generating:

Finally, it converts the graph sizes and average runtimes to logarithmic scale (base 2) using list comprehensions, storing them in log_n, log_random_times, and log_bipartite_times for plotting a log-log graph. Also generating a graph without a log process.

Display of data, including clear labelling of axes in graphs.





Conclusions:

The algorithm's worst-case complexity is $O(|V||E| + |V|^2)$.

For a random graph with edge probability 0.5, by linearity of expectation, the expected number of edges is n choose 2 multiply by ½, $|E| \approx \frac{1}{4}|V|^2$, and $|V||E| = \frac{1}{4}V^3$. dominating the $|V|^2$ term. Thus, the worst-case complexity is $O(V^3)$, which means that $log_2(runtime) \approx 3log_2(|V|)$.

For the bipartite graph, it is a near-worst-case scenario, as the absence of triangles forces the algorithm to perform nearly all checks, aligning closely with the $O(V^3)$ bound.

For the random graph, the average runtime grows slower than the worst case with time complexity of $O(V^3)$. Because random graphs are dense with triangles, causing the algorithm to terminate early after finding one, reducing the running time.

The algorithm is faster when a triangle is found early. In random graphs, triangles are abundant (the probability that three vertices form a triangle is $(0.5)^3 = 0.125$, so the algorithm often stops after checking a small fraction of vertex pairs.