Project 18: Cycle Detection in a Graph

Sarvesh Soni
B.Math, 3rd Year

October 17, 2025

---

1 Problem Statement and Motivation

For this project, the main task was to implement an algorithm for detecting cycles in a graph. I learned that this is an important problem because cycles in a dependency graph can represent a "deadlock," where a set of processes are all waiting on each other and can't proceed. A simple example is in software package management, where if "Package A needs B, B needs C, and C needs A," it would make it impossible to install any of them. My task was to write a program that could find these situations.

2 Approach and Method Used

I decided to use a Depth-First Search (DFS) based algorithm. It felt like the most natural way to explore a graph path by path and check for "back edges," which are the main cause of cycles in a directed graph.

My implementation works by keeping track of the nodes in the current path of the traversal. For this, I used two boolean arrays:

- visited[]: This is just to keep track of all nodes I have ever visited. It helps to avoid doing redundant work on parts of the graph I have already checked and confirmed to be cycle-free.
- recStack[]: This is the important one. It keeps track of only the nodes that are in the current chain of recursive calls. If the DFS process ever lands on a node that is already in the recStack, it means I have followed a path back to an ancestor, which confirms a cycle.

The project specification also asked to compare with other methods. For an undirected graph, a much faster method is using a Disjoint Set (Union-Find) data structure. You can iterate through all edges and if you find an edge '(u,v)' where both 'u' and 'v' are already in the same set, you have found a cycle. This is very efficient but doesn't work for directed graphs, so I stuck with DFS.

2.1 Proof of Correctness

The correctness of the DFS algorithm for cycle detection lies in its systematic tracking of the traversal path. By using a 'recStack' (recursion stack) array, the algorithm maintains a precise record of the current path from the starting ancestor to the current node.

A cycle is proven to exist if and only if the algorithm, from a node 'u', encounters a neighbor 'v' that is already in the 'recStack'. This condition guarantees that 'v' is an ancestor of 'u', and the edge ‘(u, v)‘ is a ”back edge” that completes a loop. Because the algorithm explores every reachable path, if no such back edge is ever found for any traversal, the graph is proven to be acyclic. The implementation correctly backtracks by removing nodes from the ‘recStack‘ as it returns, ensuring the logic holds for all parts of the graph.

3 Complexity Analysis

My analysis of the algorithm’s complexity is as follows:

- Time Complexity: O(V + E). This is the standard complexity for DFS. In the worst-case, my algorithm has to visit every single vertex (V) and cross every single edge (E) one time.
- Space Complexity: O(V). The space is needed for the visited and recStack arrays. Both depend on the number of vertices. Also, the recursion call stack can go as deep as V in the worst case (like a single long chain graph).

4 Experiments and Datasets

To test my implementation on a real-world problem, I used the Bitcoin-Alpha trust network dataset from the Stanford Network Analysis Project (SNAP).

- Source: SNAP Website.
- Description: The dataset represents trust relationships. An edge from node A to B means user A trusts user B.
- Size: It has 3,783 nodes (users) and 24,186 edges (relationships).

I chose this single, large dataset because it represents a highly diverse and complex graph structure. It contains thousands of different paths—some cyclic, some acyclic—which provides a more robust and realistic test than using several smaller, simpler graphs.

A challenge with this dataset was that the node IDs are not small or sequential. They go up to over 7000. So, my program first reads the whole file just to find the highest node ID. This way, I could create my graph structure with the exact size needed, instead of hardcoding a large number and wasting memory. Then, it reads the file a second time to actually add the edges.

5 Results and Discussion

After loading all 24,186 edges from the dataset, I ran the cycle detection algorithm. The program produced a clear result:

RESULT: A cycle was detected in the graph.

The program also printed the first cycle it found to provide proof:

--- Cycle Found! ---
Cycle Path: 1 -> 160 -> 1

This was an interesting finding. It means there are circular trust structures in the network (User 1 trusts User 160, and User 160 trusts User 1 back). The program’s execution was very fast. It analyzed all 3,783 nodes and 24,186 edges and finished in approximately 0.05 seconds on my machine. This confirms the theoretical O(V+E) efficiency of the algorithm.

5.1 Future Work

The current algorithm stops as soon as the first cycle is found. A potential extension to this project would be to modify the algorithm to find and print all elementary cycles in the graph. This is a much harder problem (it's computationally expensive), but it would give a more complete picture of the circular dependencies in a network like the Bitcoin-Alpha dataset.

6 Challenges Faced and Lessons Learned

One of the first challenges was simply setting up the file reading logic in C++. It took some trial and error to correctly parse the CSV and handle potential errors.

The biggest lesson for me was seeing a classic textbook algorithm like DFS be applied to a real, messy dataset. At first, I thought a single 'visited' array would be enough, but I quickly realized that would just tell me if I'd ever seen a node before, not if it was part of my current path. The "aha!" moment was realizing the 'recStack' was the key to tracking the ancestors, which is what makes cycle detection in DFS actually work. This project was a great experience in moving from theory to a practical, working program.