**GraphAnalyzer findCycles() example**

The GraphAnalyzer class provides the public method, **getCycles()**. In this paper we will discuss in depth how **getCycles()** works using a simple example. Before we begin, a brief discussion of the GraphAnalyzer as a whole is needed.

The GraphAnalyzer class has two members that concern **getCycles()**: **ROOT**, which is the starting node for all graph analysis, and **foundCycles**, a List that will contain the found cycles. The actual search for the cycles starts in **findCycles()**, which starts a recursive depth-first-search through the graph and adds any detected cycles to **foundCycles**.

Since the **ROOT** node may never be changed, and the nodes are assumed to be immutable, there will never be any changes to the graph we are analyzing. Because the process of finding the cycles is somewhat computationally intensive, and our results will always be the same for a given GraphAnalyzer, we have opted to save the result of **findCycles()** into **foundCycles**, rather than return the cycles directly.

Whenever **getCycles()** is called, we first check to see if **foundCycles** has been created yet. If it has not, we perform **findCycles()** and then return **foundCycles**. If it has, we simply return **foundCycles**.

The figure to the left shows a simple graph containing a cycle. Let us assume we have initialized a GraphAnalyzer with node A as the root. Upon calling **getCycles()**, we first check **foundCycles**. Since it is currently null, we must make a call to **findCycles()**.

In **findCycles()**, we initialize three collections: an ArrayDeque of Nodes (called **currentPath**) to represent the current path we are on in the graph, a HashSet of Nodes (called **visited**) to store the Nodes visited on the current path, and we initialize **foundCycles** as an ArrayList of Strings. Once this setup is complete, we make our initial call to **cyclesDFS()**, passing in the **ROOT** node, followed by the newly initialized **currentPath**, and **visited**.

**cyclesDFS()** essentially applies the white-path theorem with one minor difference: rather than marking nodes as visited, we instead add visited nodes to a HashSet. Because HashSet’s provide O(1) searches, there is no change in runtime complexity due to our alteration. While there is some slight memory overhead for using the HashSet, it is more or less negligible, and the benefit is that we do not have to require additional methods in our interface (and thus require more complex implementations).

|  |  |  |  |
| --- | --- | --- | --- |
| Step 1 Start | | **current** = A | |
| **currentPath** | **visited** | | **foundCycles** |
|  |  | |  |
| Step 1 End | | **current** = B | |
| **currentPath** | **visited** | | **foundCycles** |
| A | A | |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Step 2 Start | | **current** = B | |
| **currentPath** | **visited** | | **foundCycles** |
| A | A | |  |
| Step 2 End | | **current** = C | |
| **currentPath** | **visited** | | **foundCycles** |
| A | A | |  |
| B | B | |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Step 3 Start | | **current** = C | |
| **currentPath** | **visited** | | **foundCycles** |
| A | A | |  |
| B | B | |  |
| Step 3 End | | **current** = D | |
| **currentPath** | **visited** | | **foundCycles** |
| A | A | |  |
| B | B | |  |
| C | C | |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Step 4 Start | | **current** = D | |
| **currentPath** | **visited** | | **foundCycles** |
| A | A | |  |
| B | B | |  |
| C | C | |  |
| Step 4 End | | **current** = B | |
| **currentPath** | **visited** | | **foundCycles** |
| A | A | |  |
| B | B | |  |
| C | C | |  |
| D | D | |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Step 5 Start | | **current** = B | |
| **currentPath** | **visited** | | **foundCycles** |
| A | A | | A => B => C => D => B |
| B | **B** | |  |
| C | C | |  |
| D | D | |  |
| B |  | |  |
| Step 5 End | | **current** = D | |
| **currentPath** | **visited** | | **foundCycles** |
| A | A | | A => B => C => D => B |
| B | B | |  |
| C | C | |  |
| D | D | |  |

From this point on, we pop up the recursive call stack, moving backwards step-by-step removing the bottom-most element from **currentPath** and **visited** at each recursive call until we reach the end of the initial recursive call, at which point, **findCycles()** concludes.