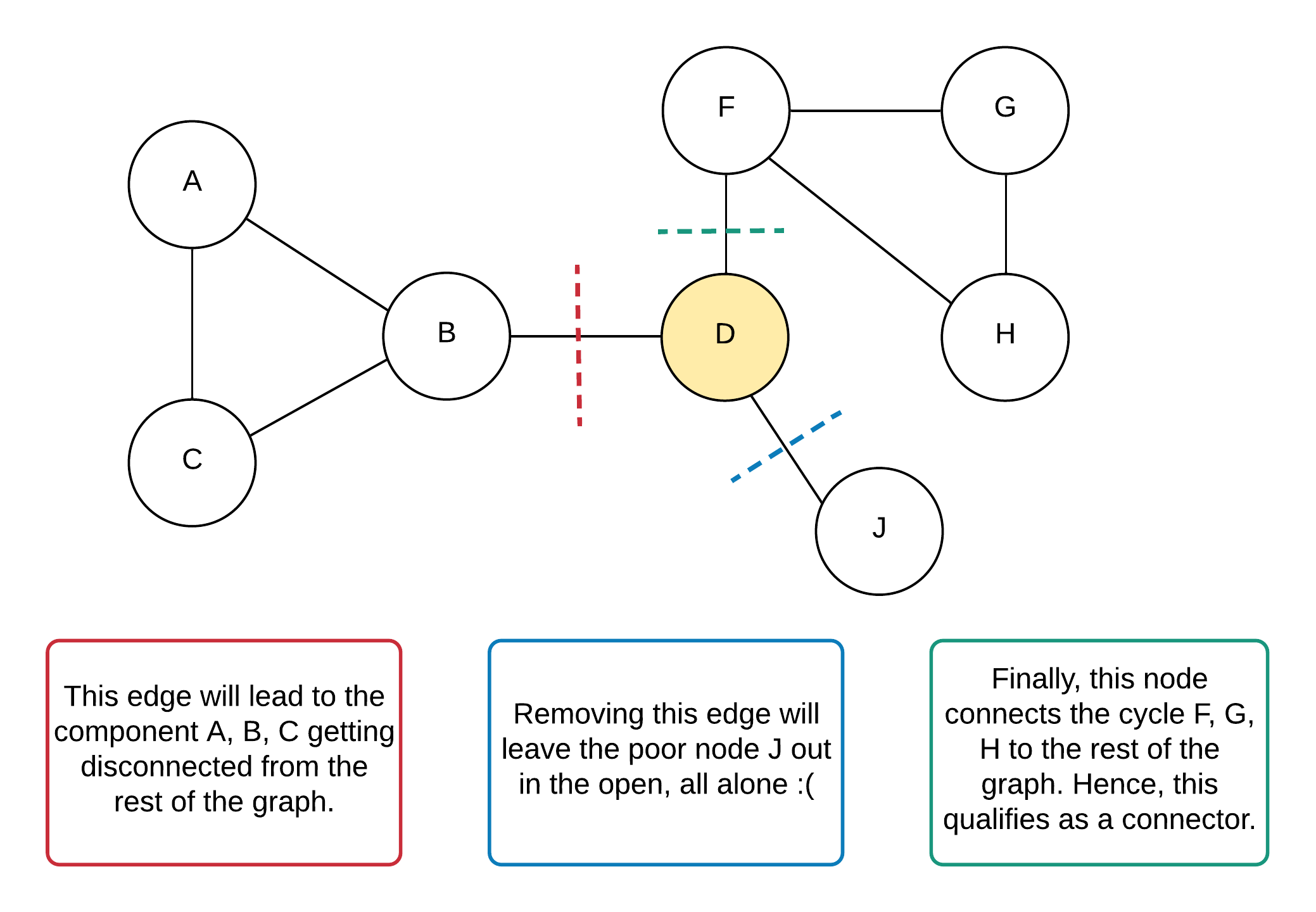
**Solution**

**Overview**

Let's try and rephrase the problem a bit to see what it means in terms of the given graph. We are given an undirected graph and are asked to find the number of *bridges* in the graph. A bridge is essentially an edge whose removal would make the graph disconnected. Let's look at a couple of example graphs where the bridges are marked.



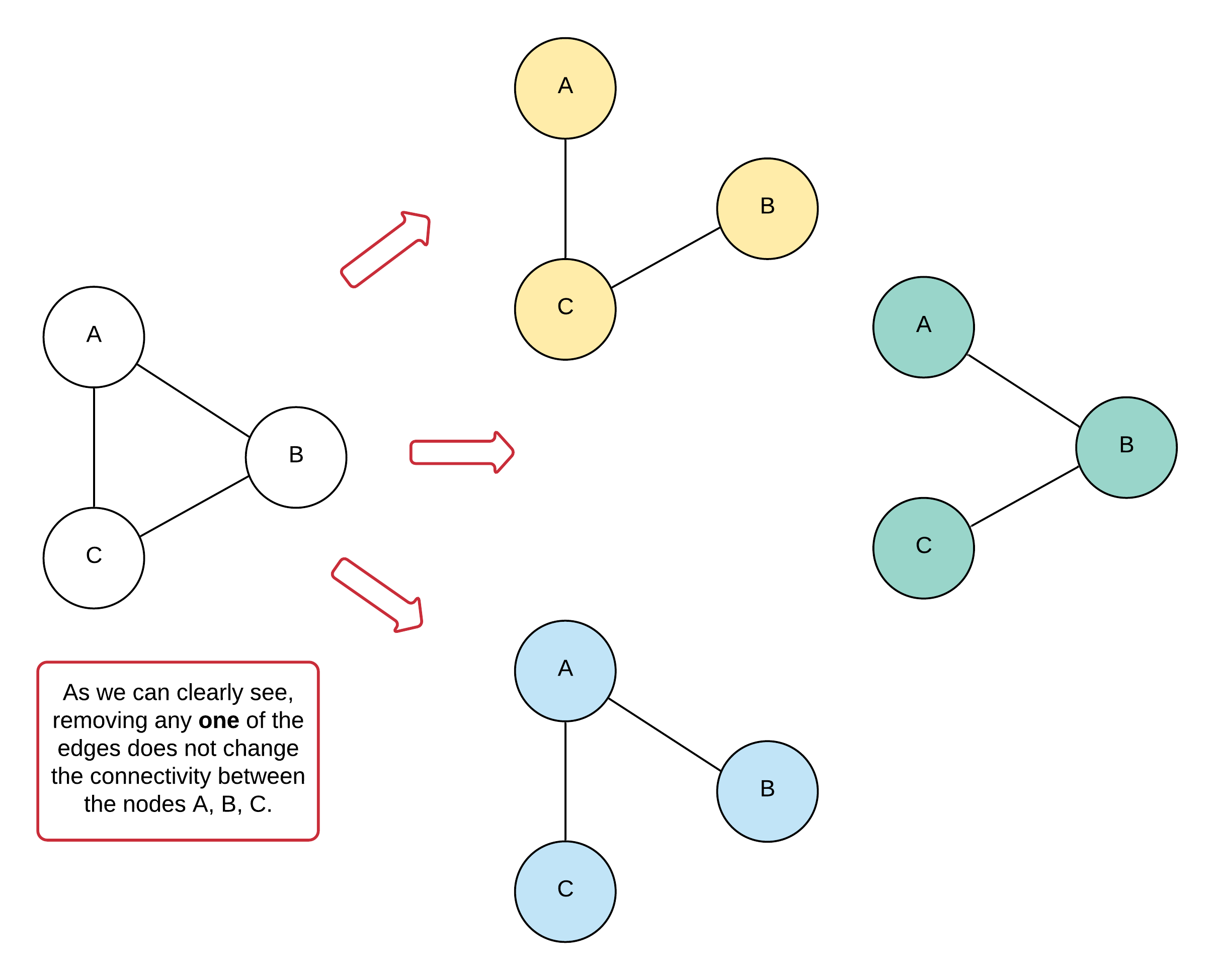
*Figure 1. A graph with the highlighted bridges.*

The node D in the example above is known as a cut. In graph theory, a cut is a partition of the vertices of a graph into two disjoint subsets. Any cut determines a cut-set, the set of edges that have one endpoint in each subset of the partition. In this problem, we are not concerned with finding the cuts in a graph: we are instead tasked with the related problems of finding all of the bridges in the graph.

As a quick aside, there's a standard algorithm out there that does precisely this i.e. find the bridges in an undirected graph. This algorithm is known as the Tarjan's algorithm, and is a standard way of finding the ***articulation points and bridges*** in a graph. [Here](https://en.wikipedia.org/wiki/Bridge_%28graph_theory%29#Tarjan.27s_Bridge-finding_algorithm) is some Wikipedia pseudo-code for the algorithm if you're interested to learn more about it.

We won't be explaining or discussing this approach in our solution because we feel that there are a lot of other sources out there for explaining standard algorithms much better and probably in much more detail. Also, such standard algorithms usually have multiple applications. For example, we use the standard traversal algorithms for solving so many graph problems, don't we? Thus, in the interest of avoiding making the article feel like a textbook, or implying that you need to have memorized Tarjan's algorithm to be able to solve this problem, and also for providing a fresh take on this problem, we will be skipping Tarjan's algorithm altogether here, and instead focussing on how we could apply existing knowledge of DFS to deriving a solution. Being able to solve graph problems using existing DFS knowledge is an essential skill to learn - you don't want to have to learn every graph algorithm anybody has ever come up with, do you?

Let's get back to the figure that depicts the bridges in the graph. There's an interesting property to be noted there (which is also used by Tarjan's algorithm) and which will end up forming the key component of our algorithm as well. Let us look at the edges belonging to a cycle and see if they could possibly be a critical connection or not.



*Figure 2. A graph showing why none of the edges in a cycle can be critical connections.*

From the above figure, we can draw the following conclusion about critical connections in general:

An edge is a critical connection, if and only if it is not in a cycle.

Edges that are not a part of the cycle end up being a single route from getting from one part of the graph to the other. Why, you ask? Well if there were multiple ways, then our edge would be a part of the cycle considering this is an undirected graph. Thus, edges not belonging to any cycle end up being a critical connection for the graph.

Thus, the problem simply boils down to finding all the cycles in the graph and discarding all the edges belonging to such cycles. If we do that, we will only be left with edges that are critical connections in the graph.

**Approach: Depth First Search for Cycle Detection**

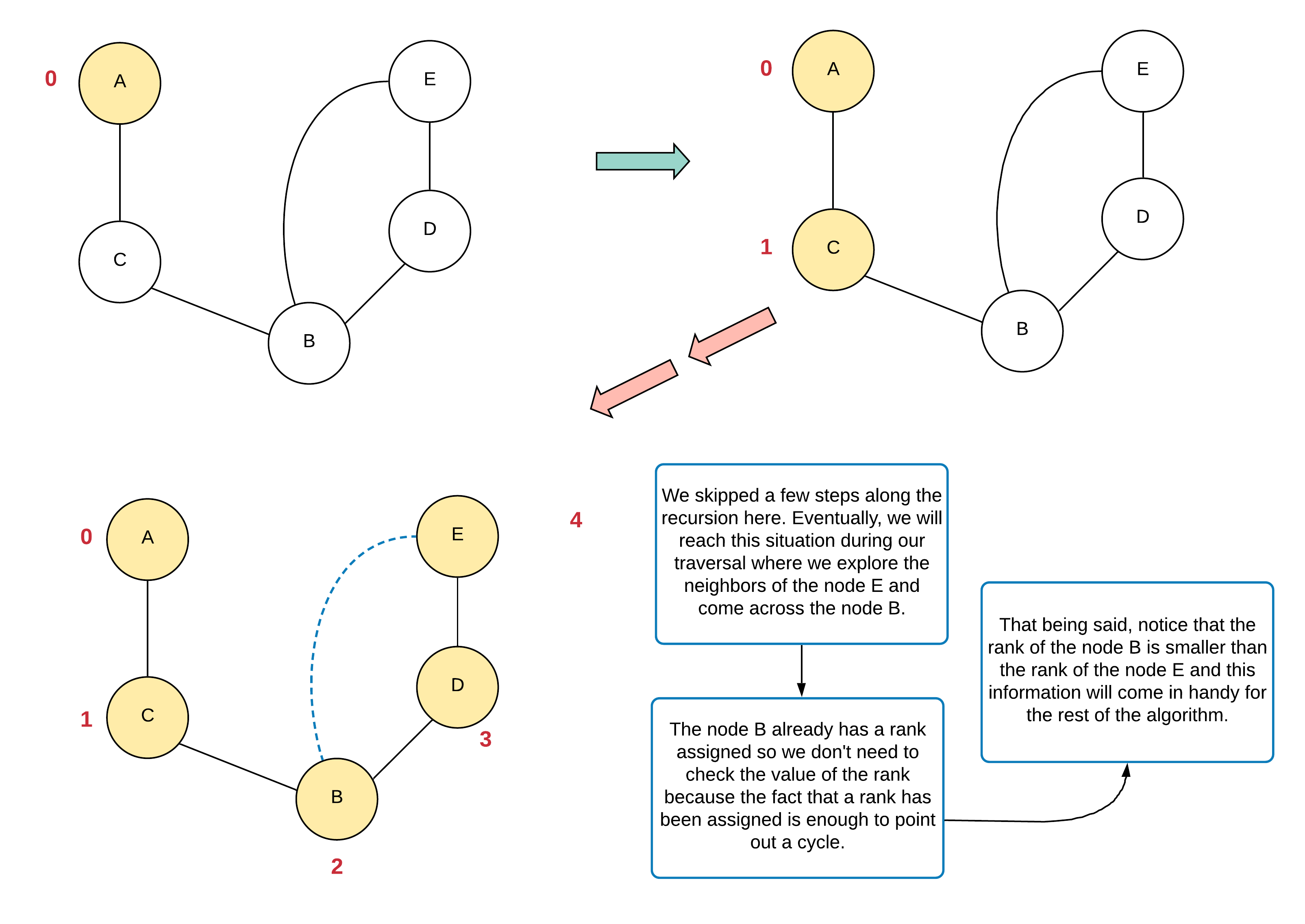
**Intuition**

**How does this fit in?** This problem is not simply about finding ***a*** cycle in an undirected graph. DFS can be easily used to detect the presence of a cycle in the graph. Given that this is an undirected graph, we can simply achieve that by keeping a visited dictionary/array of nodes and simply check for already visited nodes during the depth-first traversal. If we find a node that is already visited, that implies the presence of a cycle. The algorithm is a bit more involved for directed graphs though due to the presence of different kinds of edges.

That being said, we would need to see what variations we can make to the standard depth-first traversal to catch ***all*** cycles in the graph and then remove the corresponding edges along the way. For this, we will look at the concept of a rank of a node. This is very similar to the concept of discovery times in Tarjan's algorithm for those who are familiar with that algorithm. Since this is a graph with no designated concept of a root node like in the case of trees, we can consider any node to be the root node of our graph. Essentially, we need some node to start the traversal from, and that node becomes the root node for all intents and purposes in our algorithm.

The rank of the root node is always 0. If a node has not yet been visited, we'll keep the rank as null/None (you can use any sentinel value here except for -1 - you'll see why in the implementation.)

So how does this rank help us in detecting cycles in the graph? Well, it works exactly like keeping a set of visited nodes would work. At each step of our traversal, we maintain the rank of the nodes we've come across until now on the current path. If at any point, we come across a neighbor that has a rank lower than the current node's rank, we know that the neighbor must have already been visited. In other words, if we started along a path with rank 0 from the root node and are currently at a node with rank m and now we discover a node that already has a rank assigned to it and that value is 0 <= n < m, then it implies the presence of a cycle.



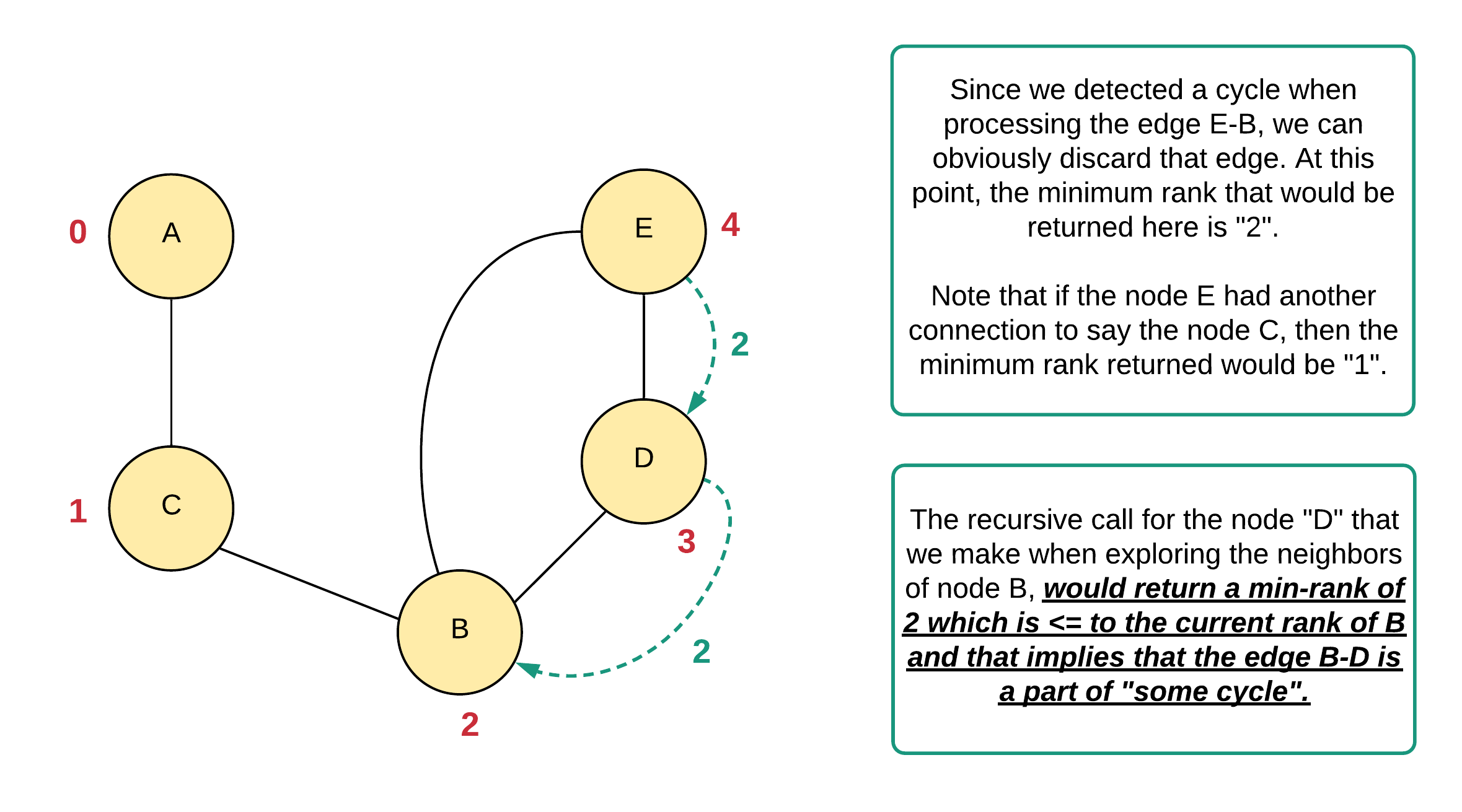
*Figure 3. The use of rank in finding nodes in a cycle.*

**Importance of the rank**

We can detect a cycle simply by checking if a rank has already been assigned to some neighbor or not (except for the parent of course, which we can handle separately). So when we detect a cycle, we can discard the current edge since that edge cannot be a critical connection. In our example graph above, we can discard edge E-B. However, how do we discard edges B-D and D-E? The mere presence of a cycle in a subgraph doesn't guarantee that an ancestral edge/node will also be a part of the cycle.

What we need is the minimum rank that a cycle includes. We need our traversal function to return this information so that the previous callers can use it to identify if an edge has to be discarded or not.

Let's see how this information might be helpful with the help of our previous graph.



*Figure 4. Showcasing the importance of rank.*

We know that only the current level knows of the presence of a cycle. To make the upper levels (of recursion) make aware of this cycle, and also to help them discard necessary edges, we return the minimum rank that our traversal finds. During a step of recursion from node u to its neighbor v, if DFS returns something smaller than or equal to rank of u, then u knows its neighbor v is a part of a cycle spanning back to u or some other node higher up in the recursion tree i.e. an ancestor node. Thus, we can safely discard edge u-v because it is part of a cycle.

**Algorithm**

1. Let's define a function called dfs that takes in the node and the discoveryRank to be assigned to this node.
2. The first step is to build the graph itself. For that, we will be using an adjacency list structure.
3. Since our algorithm involves discarding edges, we need some efficient data structure that will allow us to do this operation in *O*(1) time. We'll convert the list of edges into a dictionary for that.
4. For our algorithm, we'll name our graph as graph and our connections dictionary as connDict. The graph here is a dictionary of lists (i.e. it's an adjacency list).
5. Additionally, we need an array to keep track of the rank of our nodes. That's also something we define in the main function criticalConnections along with all the things explained above.
6. Inside our function dfs:
   1. We check if the node already has a rank assigned; if so, we return that value.
   2. Else, we assign the rank of this node i.e. rank[node] to the discoveryRank.
   3. We iterate over all the neighbors of the node and for each of them, we make a recursive call and we obtain the recursiveRank as the return value and do two things using this value.
      1. If this recursiveRank is *less than* the current discoveryRank, that implies this edge is a part of a cycle and can be discarded.
      2. Second, we record the minimum rank till now from amongst all the neighbors. Let's name this minRank. This is the value we return at the end of the function.

*Note that we don't make a recursive call to the parent node. We detect parent nodes by checking if the rank of the neighboring node is discoveryRank - 1*

1. Return the minRank.
2. We call the dfs function using the node 0 and rank 0 and once our search function completes, we convert the remaining edges from connDict to a list and return that as the result.

**Implementation**

* Time Complexity: *O*(*V*+*E*) where *V* represents the number of vertices and *E* represents the number of edges in the graph. We process each node exactly once thanks to the rank dictionary also acting as a "visited" safeguard at the top of the dfs function. Since the problem statement mentions that the graph is connected, that means *E*>=*V* and hence, the overall time complexity would be dominated by the number of edges i.e. *O*(*E*).
* Space Complexity: *O*(*E*). The overall space complexity is a sum of the space occupied by the connDict dictionary, rank dictionary, and graph data structure. *E*+*V*+(*V*+*E*) = *O*(*E*).