Readings

- [Lecture Notes Chapter 16: Graph Traversals: DFS]

Review: DFS

In DFS, we explore “deep” before exploring “wide.” We begin at some node \( v \) and examine its neighbors. When we encounter a neighbor that has not been visited yet, we visit it. Once we arrive at a node whose neighbors have all been visited, we “backtrack,” via returning from recursive calls, until we reach a node that still has unvisited neighbors. Because of this, DFS can be implemented recursively as seen in this pseudocode, but we can also implement it iteratively by using a stack since it is LIFO. The iterative pseudocode for DFS with a stack is the BFS pseudocode, except instead of a queue we use a stack, instead of dequeue we use pop, and instead of enqueue we use push. We know that DFS will visit every node in the graph, so the output of DFS is a DFS forest. Note that DFS also introduces the concept of colors and finishing times, which makes it useful for other graph applications later on.

The running time analysis for DFS is similar to that of BFS; for each vertex \( v \), we iterate through its neighbors to yield a runtime of \( O(n + m) \).

Problems

Problem 1

The CIS Department wants to assign prerequisites to their \( n \) classes. It has a list of \( m \) prerequisite pairings, such as CIS 110 being a prerequisite for CIS 120. Given the list of prerequisite pairings, design an algorithm that determines if this list of pairings is a valid list of prerequisites. For example, (CIS 110, CIS 120), (CIS 120, CIS 121), (CIS 121, CIS 110) is not a valid list of prerequisites. What is the runtime of your algorithm?

Solution

Algorithm: Construct a graph \( G \), where each of the vertices is one of the \( n \) classes in the CIS department and there is a directed edge \((u, v)\) if \( u \) is a prerequisite for \( v \). Run DFS on \( G \). If the DFS traversal finds a back edge during its execution, return that this list is invalid. Otherwise, return that this is list is valid.

Proof of Correctness: If \( u \) is a prerequisite for \( v \), then by definition, we know that students must take course \( u \) before taking course \( v \). Directing an edge from \( u \) to \( v \) thus translates this exact information and nothing more, so \( G \) sufficiently translates all of the information given to us. Observe that a list of prerequisites is valid iff \( G \) does not contain a cycle. Now, we want to show that \( G \) has a cycle iff a DFS traversal finds a back edge:

\((\Rightarrow)\) If \( G \) has some cycle \( C \), then let \( v \) be the first vertex on this cycle discovered by DFS and let \((u, v)\) be the edge directly preceding \( v \) on \( C \). When \( v \) is discovered, we know that all nodes in \( C \) are still undiscovered and colored white. By definition, \( v \) is an ancestor of \( u \) in the DFS forest, so we know \((u, v)\) is a back edge.

\(\leftarrow\) If there is a back edge \((u, v)\), then by definition, we know that there is a path from \( v \) to \( u \) made of tree edges. Concatenating this path from \( v \) to \( u \) with the back edge \((u, v)\) creates a cycle.
Therefore, since $G$ has a cycle iff a DFS traversal finds a back edge, we know that our algorithm is correct.

**Runtime Analysis:** Constructing $G$ as an adjacency list takes $O(n + m)$ time. Running DFS on $G$ and determining if there are any back edges takes $O(n + m)$ time as well, so this algorithm runs in $O(n + m)$ time.