def is\_DAG(graph: TripleDictGraph):  
 *"""* ***:param*** *graph: a TripleDictGraph, containing as nodes the activities, and as costs the duration of the every activity* ***:return****: True, if the actual graph is a directed acyclic graph (DAG) and False otherwise  
 """* # the is\_DAG function is used to check whether a given graph is a Directed Acyclic Graph (DAG). It performs a depth-first search (DFS) to detect cycles in the graph visited\_vertices = []  
 visited\_vertices = []  
 stack = []  
  
 # in the visited vertices I store the nodes that have been verified already for not taking part in a cycle  
  
 def has\_cycle(activity\_node):  
 if activity\_node in visited\_vertices:  
 return False  
 # this means that we have already verified this vertex  
 visited\_vertices.append(activity\_node)  
 stack.append(activity\_node)  
  
 for predecessor in graph.dictionary\_in[activity\_node]:  
 if predecessor in stack and predecessor != activity\_node or has\_cycle(predecessor):  
 return True  
  
 stack.remove(activity\_node)  
 return False  
  
 for node in graph.dictionary\_in:  
 if has\_cycle(node):  
 return False  
  
 return True  
  
  
def topological\_sort(graph: TripleDictGraph):  
 if is\_DAG(graph) is False:  
 return None  
 else:  
 sorted\_activities = []  
 queue\_activities = deque()  
 count\_of\_predecessors = {}  
  
 for vertex in graph.dictionary\_in:  
 count\_of\_predecessors[vertex] = graph.in\_degree(vertex)  
 if count\_of\_predecessors[vertex] == 0:  
 queue\_activities.append(vertex)  
  
 while queue\_activities:  
 current\_activity = queue\_activities.popleft()  
 sorted\_activities.append(current\_activity)  
  
 for successor in graph.dictionary\_out[current\_activity]:  
 count\_of\_predecessors[successor] -= 1  
 if count\_of\_predecessors[successor] == 0:  
 queue\_activities.append(successor)  
  
 return sorted\_activities  
  
  
def calculate\_earliest\_starting\_time(graph: TripleDictGraph):  
 # if the activities can be done simultaneously  
 sorted\_activities = topological\_sort(graph)  
  
 if sorted\_activities is None:  
 return None  
  
 costs = {}  
 for vertex in sorted\_activities:  
 costs[vertex] = 0  
  
 earliest\_starts = {}  
 for vertex in sorted\_activities:  
 earliest\_starts[vertex] = 0  
  
 cost\_until = 0  
  
 processed\_vertices = []  
 for vertex in sorted\_activities:  
 if vertex not in processed\_vertices:  
 processed\_vertices.append(vertex)  
 for successor in graph.dictionary\_out[vertex]:  
 if successor not in processed\_vertices:  
 processed\_vertices.append(successor)  
 cost = graph.dictionary\_cost[(vertex, successor)]  
 for predecessor in graph.dictionary\_in[successor]:  
 if predecessor in processed\_vertices:  
 cost = min(cost, graph.dictionary\_cost[(predecessor, successor)])  
 costs[successor] = max(costs[successor], costs[vertex] + cost)  
 cost\_until += costs[successor]  
 earliest\_starts[successor] = cost\_until  
  
 return earliest\_starts  
  
  
def calculate\_latest\_starting\_time(graph: TripleDictGraph):  
 sorted\_activities = topological\_sort(graph)  
  
 if sorted\_activities is None:  
 return None  
  
 costs = {}  
 for vertex in sorted\_activities:  
 costs[vertex] = 0  
  
 earliest\_starts = calculate\_earliest\_starting\_time(graph)  
  
 latest\_starts = {}  
 for vertex in sorted\_activities:  
 latest\_starts[vertex] = float('inf') # Initialize latest starting times as infinity  
  
 cost\_until = 0  
  
 processed\_vertices = []  
 for vertex in reversed(sorted\_activities): # Reverse the order of iteration  
 if vertex not in processed\_vertices:  
 processed\_vertices.append(vertex)  
 for predecessor in graph.dictionary\_in[vertex]: # Iterate over predecessors instead of successors  
 if predecessor not in processed\_vertices:  
 processed\_vertices.append(predecessor)  
 cost = graph.dictionary\_cost[(predecessor, vertex)] # Use the predecessor-vertex cost  
 for successor in graph.dictionary\_out[predecessor]:  
 if successor in processed\_vertices:  
 cost = min(cost, graph.dictionary\_cost[(predecessor, successor)])  
 costs[predecessor] = max(costs[predecessor], costs[vertex] + cost)  
 cost\_until += costs[predecessor]  
 latest\_starts[predecessor] = cost\_until # Store the latest starting time  
  
 for vertex in latest\_starts:  
 if latest\_starts[vertex] == float('inf'):  
 latest\_starts[vertex] = 0  
  
 maximum\_late\_starting\_time = max(latest\_starts.values())  
 for vertex in latest\_starts:  
 latest\_starts[vertex] = max(maximum\_late\_starting\_time - latest\_starts[vertex], earliest\_starts[vertex])  
  
 return latest\_starts  
  
  
def find\_critical\_activities(graph):  
 sorted\_activities = topological\_sort(graph)  
 if sorted\_activities is None:  
 return None  
  
 critical\_activities = []  
 earliest\_starts = calculate\_earliest\_starting\_time(graph)  
 latest\_start = reversed(calculate\_latest\_starting\_time(graph))  
  
 for activity in sorted\_activities:  
 if earliest\_starts[activity] == latest\_start[activity]:  
 critical\_activities.append(activity)  
  
 if len(critical\_activities) == 0:  
 return None  
  
 return critical\_activities