def topological\_sort(self):

in\_degree = [0] \* self.\_\_number\_of\_vertices

sorted\_order = []

queue = deque()

for vertex in self.\_\_vertices:

for successor in self.get\_successors()[vertex]:

in\_degree[successor] += 1

for vertex in self.\_\_vertices:

if in\_degree[vertex] == 0:

queue.append(vertex)

while queue:

vertex = queue.popleft()

sorted\_order.append(vertex)

for successor in self.\_\_list\_of\_successors[vertex]:

in\_degree[successor] -= 1

if in\_degree[successor] == 0:

queue.append(successor)

if len(sorted\_order) != self.\_\_number\_of\_vertices:

return None

return sorted\_order

def scheduling\_problem(self):

order = self.topological\_sort()

if len(order) != self.get\_number\_of\_vertices():

print("Not a DAG!")

return

total\_time = sum(self.duration.values())

earliest = [0] \* self.get\_number\_of\_vertices()

latest = [total\_time] \* self.get\_number\_of\_vertices()

duration = [self.duration[i] for i in range(self.get\_number\_of\_vertices())]

for x in order:

for in\_vertex in self.service\_inbound\_edges(x):

earliest[x] = max(earliest[x], earliest[in\_vertex] + duration[in\_vertex])

for x in order[::-1]:

if earliest[x]==0:

latest[x]=0

else:

latest[x]=total\_time-earliest[x];

for out\_vertex in self.service\_outbound\_edges(x):

latest[x] = min(latest[x], latest[out\_vertex] - duration[x])

for i in range(self.get\_number\_of\_vertices()):

print(f"For activity {i}, the earliest starting time is {earliest[i]} and the latest is {latest[i]}.")

print(f"The total time to finish the project is {total\_time}.")

critical\_activities = [i for i in range(self.get\_number\_of\_vertices()) if earliest[i] == latest[i]]

print("The critical activities are:", critical\_activities)

def add\_activity(self, activity, duration, dependencies):

vertex = self.activities.get(activity)

if vertex is None:

vertex = len(self.\_\_vertices)

self.add\_vertex(vertex)

self.activities[activity] = vertex

self.duration[vertex] = duration

self.dependencies[vertex] = []

dep\_vertices = []

for dep in dependencies:

dep\_vertex = self.activities.get(dep)

if dep\_vertex is None:

dep\_vertex = len(self.\_\_vertices)

self.add\_vertex(dep\_vertex)

self.activities[dep] = dep\_vertex

self.duration[dep\_vertex] = 0

self.dependencies[dep\_vertex] = []

dep\_vertices.append(dep\_vertex)

self.dependencies[vertex].extend(dep\_vertices)

for dep\_vertex in dep\_vertices:

self.\_\_list\_of\_successors[vertex].append(dep\_vertex)

self.\_\_list\_of\_predecessors[dep\_vertex].append(vertex)

def read\_activities(self, file\_path):

self.\_\_number\_of\_vertices=0

self.\_\_vertices.clear()

self.\_\_list\_of\_predecessors.clear()

self.\_\_list\_of\_successors.clear()

self.duration.clear()

self.activities.clear()

with open(file\_path, 'r') as file:

line\_number = 0

for line in file:

line\_number += 1

line = line.strip()

if line:

parts = line.split(":")

activity = parts[0]

duration = float(parts[1])

dependencies = parts[2:]

self.add\_activity(activity, duration, dependencies)

topological\_sort:

Input: None

Output: A list representing the topologically sorted order of vertices

None if the graph is not a directed acyclic graph (DAG).

Steps:

Create an in\_degree list of size number\_of\_vertices initialized with zeros.

Create an empty sorted\_order list to store the sorted vertices.

Create an empty queue using deque() from the collections module.

Calculate the in-degree of each vertex by iterating over the vertices and their successors using get\_successors() function.

For each vertex, if its in-degree is zero, enqueue it.

While the queue is not empty, do the following:

* Dequeue a vertex from the queue.
* Append the vertex to the sorted\_order list.
* For each successor of the dequeued vertex, reduce its in-degree by 1.
* If the in-degree of a successor becomes zero, enqueue it.

If the length of the sorted\_order list is not equal to number\_of\_vertices, return None indicating that the graph is not a DAG.

Otherwise, return the sorted\_order list.

scheduling\_problem:

Input: None

Output: Prints the earliest and latest starting times for each activity,

The total time to finish the project, and the critical activities.

Steps:

Perform a topological sort of the graph using the topological\_sort function and store the sorted order in the order variable.

If the length of the order list is not equal to the number of vertices in the graph, print "Not a DAG!" and return.

Calculate the total time by summing the durations of all activities.

Create an earliest list of size equal to the number of vertices, initialized with zeros.

Create a latest list of size equal to the number of vertices, initialized with the total time.

Create a duration list by iterating over the vertices and their durations.

Calculate the earliest starting time for each activity in the order using a nested loop:

* For each activity in the order, iterate over its inbound edges using the service\_inbound\_edges function.
* Update the earliest starting time of the current activity by taking the maximum of its current earliest time and the earliest time of the inbound edge activity plus its duration.

Calculate the latest starting time for each activity in the order using a reverse loop over the order list:

* + If the earliest starting time of the current activity is zero, set its latest starting time to zero.
  + Otherwise, set its latest starting time to the total time minus its earliest starting time.
  + For each outbound edge of the current activity, iterate over them using the service\_outbound\_edges function.
  + Update the latest starting time of the current activity by taking the minimum of its current latest time and the latest time of the outbound edge activity minus its duration.

Print the earliest and latest starting times for each activity, print the total time to finish the project.

Find the critical activities by creating a list of activities whose earliest and latest starting times are equal and print them.

add\_activity:

Input: activity: The name of the activity (string).

duration: The duration of the activity (float).

dependencies: A list of dependencies (strings) for the activity.

Output: None

read\_activities:

Input: file\_path - The path to a file containing activity details.

Output: None