**Explain the Algorithm**

**1. Graph Construction:**

We create a directed graph based on the job dependencies. Each job is represented as a node, and an edge from job u to job v means that job v depends on job u. Additionally, we maintain an in\_degree array, where each element tracks how many jobs need to be completed before a given job.

graph = defaultdict(list)

in\_degree = [0] \* num\_jobs

for u, v in dependencies:

    graph[*u*].append(v)

    in\_degree[*v*] += 1

### 2. ****Initialize Queue with Jobs with No Dependencies****:

Kahn’s Algorithm starts by adding all jobs that have no dependencies (eg, jobs with an in\_degree of 0) to a queue.

queue = deque([*i* *for* *i* *in* *range(num\_jobs)* *if* *in\_degree*[*i*] *==* 0])

### 3. ****Process Jobs and Reduce In-degree****:

We process each job in the queue:

* Remove a job from the queue and add it to the top\_order list (the topological order).
* For each dependent job (neighbor), we reduce its in-degree (i.e., one dependency has been completed). If a dependent job’s in-degree becomes 0, it is added to the queue.
* top\_order = []
* while queue:
* job = queue.popleft()
* top\_order.append(job)
* for neighbor in graph[*job*]:
* in\_degree[*neighbor*] -= 1
* if in\_degree[*neighbor*] == 0:
* queue.append(neighbor)

### 4. ****Cycle Detection****:

If there are still jobs left with non-zero in-degree after processing all jobs, it means a cycle exists in the dependency graph. This is checked by comparing the length of the top\_order with num\_jobs. If they don’t match, a cycle is detected, and an error is raised.

if len(top\_order) == num\_jobs:

    return top\_order

else:

    raise ValueError("Cycle detected in the dependency graph.")

### ****Integration with Scheduling****:

Once the jobs are sorted topologically, the order is passed to the schedule\_jobs function, which schedules the jobs on available machines