REPORT - Case B

Team Atlantean

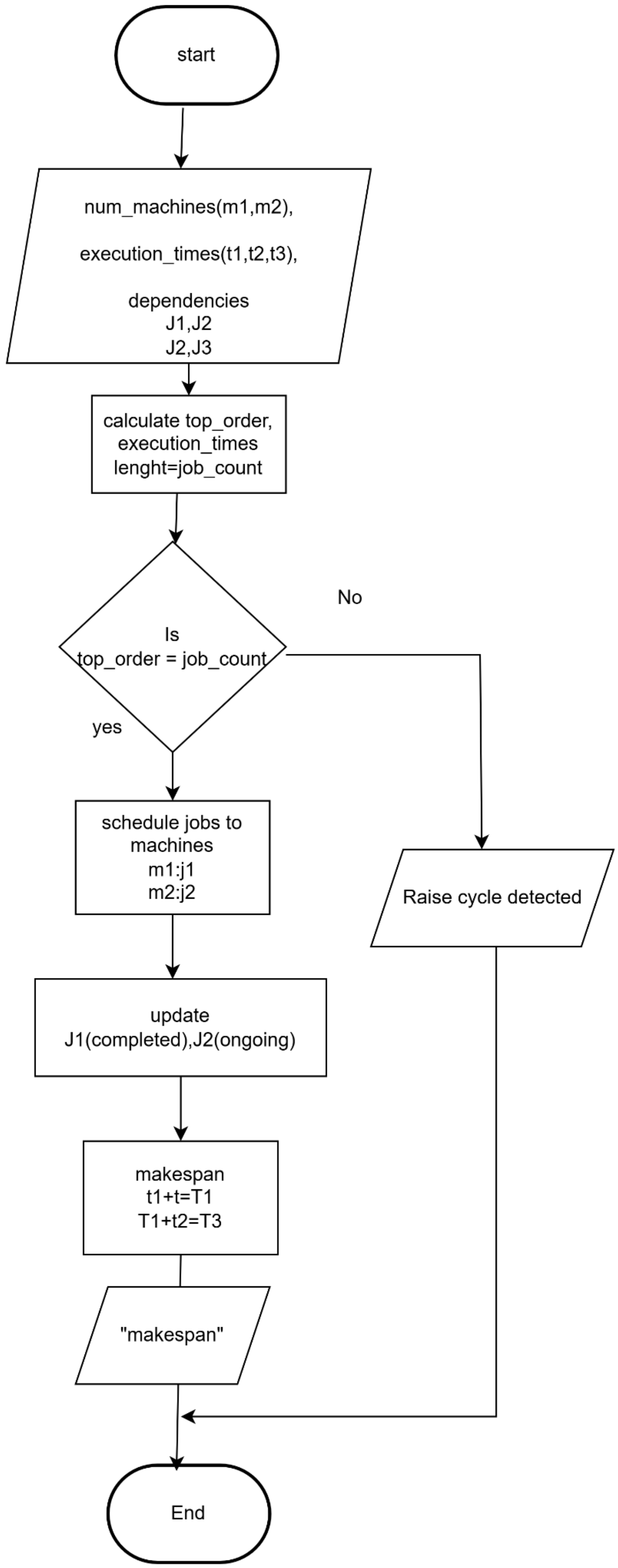
—

Sabaragamuwa University Of
Sri Lanka

—

- K.H.A. Niranga Nayanajith.
- H.A. Ashingshana Ishan.
- C.H.H.K. Gayathri Isurika.

Flow Chart for Case B



Python file for Case B , By using Kahn's Algorithm theory.

algorithm.py

```python
import json
from collections import deque, defaultdict
import heapq

def topological_sort(job_count, dependencies): # Sorting using Kahn's Algorithm
    graph = defaultdict(list)   # Deploy graph and in-degrees
    in_degree = [0] * job_count
    for u, v in dependencies:
        graph[u].append(v)
        in_degree[v] += 1
    queue = deque([i for i in range(job_count) if in_degree[i] == 0])   # Queue for jobs with no dependencies
    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)
    if len(top_order) == job_count:
        return top_order
    else:
        raise ValueError("Cycle detected in the dependency graph.")

def schedule_jobs(job_count, execution_times, top_order, num_machines):    # Schedule Jobs on Machines
    machine_heap = [(0, i) for i in range(num_machines)]  # Min-heap to track next available time of each machine # (time, machine_id)
    heapq.heapify(machine_heap)
    job_completion_time = [0] * job_count # Track each job completion time
    for job in top_order:
        earliest_time, machine_id = heapq.heappop(machine_heap)
        start_time = earliest_time
        finish_time = start_time + execution_times[job]
        job_completion_time[job] = finish_time
        heapq.heappush(machine_heap, (finish_time, machine_id))  # check machine is free or not
    makespan = max(job_completion_time)   # Makespan is the maximum finish time of all jobs
    return makespan, job_completion_time

def minimize_makespan(job_count, execution_times, dependencies, num_machines):   # Main Function
    top_order = topological_sort(job_count, dependencies) # Perform topological sorting
    makespan, job_completion_time = schedule_jobs(job_count, execution_times, top_order, num_machines) # Schedule jobs on machines
    return makespan, job_completion_time

if __name__ == "__main__":

    input_file = "input.json"  # get inputs from JSON file
    try:
        with open(input_file, "r") as file:
            data = json.load(file)

        execution_times = data["execution_times"]  # Extract data from JSON
        dependencies = data["dependencies"]
        num_machines = data["num_machines"]
        job_count = len(execution_times)

        makespan, job_completion_time = minimize_makespan(job_count, execution_times, dependencies, num_machines)  # Run the algorithm

        print("Makespan:", makespan) # Output the results # Times for complete each job
    except FileNotFoundError:       # Error handling
        print(f"Error: File '{input_file}' not found.")
    except json.JSONDecodeError:
        print(f"Error: Failed to parse JSON from '{input_file}'.")
    except ValueError as e:
        print(e)
```

Json file format.


```json
{
    "execution_times": [10, 20, 30, 40,4,5,8,5,2,5,54,4,8,2,4,2,4,6,2,4,65,2,85,2,5,1,4,512,5,5,4,5,7,1,12,4,2,5,2,5,44,84,872,4,3,6,9,5,2,
                        5,7,425,5,6,74,5,2,5,2,56,3,54,6,4,58,55,2,5],
    "dependencies": [[0, 1], [1, 2],[10, 1], [11, 2], [18, 2], [17, 2],[30, 21], [21, 7],[10, 17], [19, 22],[10, 21],
                    [12, 27],[9, 1], [14, 25],[12, 26],[8, 11], [7, 13],[18, 15]],
    "num_machines": 2
}
```


Github Repositorie.
https://github.com/nngeek195/Algothon/blob/main/algorithm.py

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

Why we say this code is efficient

Because in this code we used Kahn's algorithm to complete our task. Kahn's Algorithm, which has a time complexity of O(V + E), where V is the number of jobs and E is the number of dependencies Constructing the graph and processing each node and edge happens linearly with respect to the size of the input.
Also, the priority queue will schedule jobs to ensure that the allocation of jobs to machines is done efficiently. The heap operations are logarithmic, which ensures that the scheduling process scales well even with a large number of jobs and machines.
Combined, the code efficiently resolves both dependency resolution and job scheduling, hence suitable for use cases involving a large number of jobs and dependencies.