Scheduling Simulator in Python

1 The Goal

Our goal is to create a scheduling simulator in Python. The simulator will take as input:

- A set of tasks, each with a unique identifier and a specified runtime.
- The dependencies between tasks, forming a Directed Acyclic Graph (DAG).
- The number of processors available for executing the tasks.

The scheduler's objective is to execute all tasks while respecting their dependencies, minimizing the total time until completion (makespan). The scheduler will use a specified scheduling algorithm to decide the order in which tasks are assigned to processors.

2 The Algorithm

The algorithm works as follows:

1. Initialize the list of ready tasks with all tasks that have indegree 0.
2. Initialize the current time to 0.
3. Initialize the list of running tasks and an array to track processor usage. All processors are initially free.
4. Use the scheduling algorithm to assign ready tasks to free processors.
5. Enter the main loop, which continues until there are no more running tasks:
   (a) Pop the task with the minimal end time from the list of running tasks.
   (b) Update the current time to this task's end time, mark the processor as free, and update the indegree of all tasks dependent on this completed task. Add tasks whose indegree becomes 0 to the list of ready tasks.

   (c) If there are ready tasks, use the scheduling algorithm to assign them to free processors. Update their end times based on the current time and their runtimes.

6. Once the simulator is done, collect and save statistics, including the total runtime (ie the makespan) and other performance measures.

3 Programming Considerations

To implement the scheduling simulator, we will use an object-oriented approach. The main classes and their interactions are described below.

3.1 Class Descriptions

3.1.1 Task

Represents a single task in the scheduling simulator.

- **Attributes**:
  - `task_id`: Unique identifier for the task.
  - `runtime`: Duration the task takes to complete.
  - `dependencies`: List of task IDs that this task depends on.
  - `end_time`: The time when the task is expected to finish.
  - `processor`: The processor assigned to this task.

3.1.2 SchedulingAlgorithm

An abstract base class for scheduling algorithms.

- **Methods**:
  - `choose_next_task(ready_tasks)`:
    Selects the next task to be scheduled from the list of ready tasks.

We can have three scheduling algorithms: A simple queue, an algorithm that chooses the task with the minimal runtime and one that chooses the task with the maximal outdegree.

3.1.3 SchedulerSimulator

Encapsulates the entire scheduling simulation logic.

- **Attributes**:
  - `tasks`: Dictionary of task IDs to `Task` objects.
  - `num_processors`: Number of available processors.

– `scheduling_algorithm`: Instance of `SchedulingAlgorithm`.
– `ready_tasks`: Queue of tasks ready to be executed.
– `running_tasks`: List of currently running tasks.
– `indegree`: Dictionary of task IDs to their current indegree.
– `current_time`: The current time in the simulation.
– `processors`: List indicating the busy/free status of processors.

- **Methods**:

  – `initialize_indegree_and_ready_tasks()`: Initializes the indegree and ready tasks list.
  – `run()`: Runs the simulation.
  – `assign_tasks()`: Assigns ready tasks to free processors.
  – `process_next_completion()`: Processes the next task completion.
  – `save_statistics()`: Collects and saves the simulation statistics.

4 Metrics of Success

The primary metric of success for the scheduler is the makespan, which is the total time until all tasks are completed. Additionally, other metrics can be considered to evaluate the performance of the scheduling algorithm, such as:

- **Processor Utilization**: The percentage of time each processor is busy.
- **Throughput**: The number of tasks completed per unit of time.
- **Average Waiting Time**: The average time tasks spend waiting to be scheduled.
- **Average Turnaround Time**: The average time from task submission to task completion.

At the end of the simulation, these metrics can be collected and saved to a file for further analysis.

4.1 Saving Statistics

The statistics can be saved to a file as follows:

Listing 1: Saving Statistics in Python

```
def save_statistics(self):
    makespan = self.current_time
    processor_utilization = sum([1 for p in self.processors if p]) / len(self.pr
    # Assuming throughput and waiting/turnaround times are calculated during the
    with open("statistics.txt", "w") as file:
        file.write(f"Makespan: {makespan}\n")
        file.write(f"Processor Utilization: {processor_utilization:.2f}\n")
        file.write(f"Throughput: {throughput}\n")
        file.write(f"Average Waiting Time: {avg_waiting_time}\n")
        file.write(f"Average Turnaround Time: {avg_turnaround_time}\n")
```