CS 2051: Honors Discrete Mathematics
Spring 2023 `generate_schedule` Solution

Sarthak Mohanty

The `generate_schedule` function in this supplement seemed to pose quite the challenge! In this short document, we cover some initial approaches, the challenges/issues encountered, and finally the correct approach.

Initial Approach

A common first implementation by many students was iteratively adding the minimal elements, as follows:

Incorrect Approach

```python
def generate_schedule(elements: set, poset: set, num_processors: int) -> list[list]:

    scheduled_tasks = set()
    schedule = list()

    while(len(scheduled_tasks) != len(elements)):
        tasks = []

        # Look at all elements in the poset
        # and check if they are not in scheduled_tasks
        # and if they are not dependent on any elements in scheduled_tasks
        tasks = [v for v in elements if v not in scheduled_tasks\
                 and all(u in scheduled_tasks for u, _v in poset if _v == v)]

        # Take the first num_processors elements
        tasks = tasks[:num_processors]

        # Add the elements in tasks to scheduled_tasks
        scheduled_tasks = scheduled_tasks.union(tasks)

        schedule += [tasks]

    return schedule
```

This approach works fine for the example given in the docstrings, as shown below:

```
>>> generate_schedule({'1301', '1331', '1332', '4641', '2051', '3600', '3511'},
        {('1301', '1331'), ('1331', '1332'), ('1332', '4641'),\
         ('2051', '3511'), ('1332', '3511'), ('1332', '3600')}, 2)
    [['1301', '2051'], ['1331'], ['1332'], ['4641', '3600'], ['3511']]
```

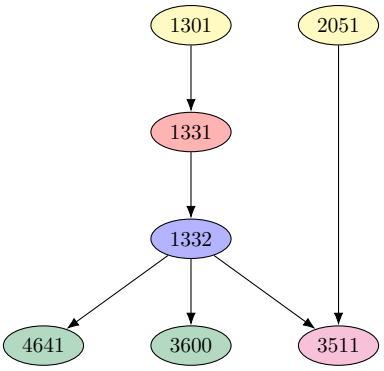


Figure 1: Dependency graph of the sample poset, where the elements in row i represents tasks that are ready to be scheduled at time i

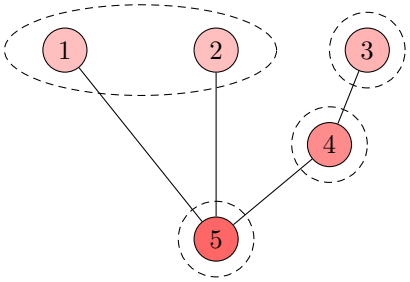
The Issue

However, the above implementation does not always find the optimal schedule. Consider the following poset:

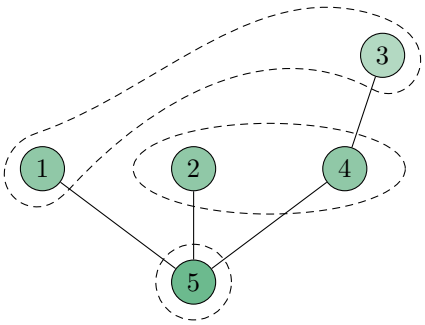
```
>>> generate_schedule({1, 2, 3, 4, 5}, {{(1, 5), (2, 5), (3, 4), (4, 5)}}, 2)
    [[1, 2], [3], [4], [5]]
```

In this case, the poset should schedule Task 3 first, but instead schedules Tasks 1 and 2 first, leading to an unoptimal schedule.

Intuitively, this is because Task 3 has a further distance, or *height* to the final Task 5 in the dependency graph, so it should be scheduled first, as shown in Figure 2b. However, our initial algorithm does not make this differentiation, and instead treats the poset as in Figure 2a.



(a) Dependency graph demonstrating the initial approach.



(b) Dependency graph where elements are ordered by height.

Figure 2: Two schedules, each illustrating a different approach to the problem.

Revised Approach

After understanding why our initial algorithm fails, it is easy to see how to correct it.

1. Before scheduling any tasks, we should calculate the height of every element by finding the maximum distance to any *maximal* elements (elements which do not have any dependents).
2. After obtaining tasks which are ready to be scheduled, we should "rank" them based on their calculated heights in the dependency graph.

The updated approach is shown below:

Correct Approach

```
def generate_schedule(elements: set, poset: set, num_processors: int) -> list[list]:

    scheduled_tasks = set()
    schedule = list()

    # Compute the height of each element
    heights = { x: 0 for x in elements }
    for x in topological_sort(elements, poset)[::-1]:
        dependents = [ v for _u, v in poset if _u == x ]
        if dependents:
            # height of x is 1 + max height of children
            heights[x] = 1 + max(heights[y] for y in dependents)

    while(len(scheduled_tasks) != len(elements)):
        tasks = []

        # Look at all elements in the poset
        # and check if they are not in scheduled_tasks
        # and if they are not dependent on any elements in scheduled_tasks
        tasks = [v for v in elements if v not in scheduled_tasks\
                 and all(u in scheduled_tasks for u, _v in poset if _v == v)]

        # Sort the elements by their depth in the poset
        tasks = sorted(tasks, key=lambda x: heights[x], reverse=True)

        # Take the first num_processors elements
        tasks = tasks[:num_processors]

        # Add the elements in tasks to scheduled_tasks
        scheduled_tasks = scheduled_tasks.union(tasks)

        schedule += [tasks]

    return schedule
```

```
>>> generate_schedule({1, 2, 3, 4, 5}, {{(1, 5), (2, 5), (3, 4), (4, 5)}}, 2)
    [[1, 3], [2, 4], [5]]
```

Time Complexity

The current implementation of this algorithm yields a time complexity of $\mathcal{O}(\texttt{len(elements)}^3)$, where n is the number of elements in the poset. This is because (1) we sort each of the elements, which takes time $\mathcal{O}(\texttt{len(elements)}\log(\texttt{len(elements)}))$ at each iteration, and (2) computing each of the minimal elements takes $\mathcal{O}(\texttt{len(elements)}^2)$ time at each iteration.

However, by using more efficient representations for our data, as well as external data structures such as priority queues, we can reduce the time complexity to

$$\mathcal{O}(\texttt{len(elements)} \times (\texttt{len(elements)} + \texttt{num\_processors} \times \log(\texttt{len(elements)})))[1].$$

It's a good exercise to determine why this is the case, as well as create such an implementation.

[1] I haven't analyzed it too deeply, so it might be able to be improved even further.