CS 2051: Honors Discrete Mathematics
Spring 2023 Homework 5 Supplement

Sarthak Mohanty

Overview

Note: This document and the autograder will be continuously updated until Monday morning, but the fundamentals remain the same.

Traditionally, computer software has been written for serial computation: instructions corresponding to a tasks are executed on one processing unit at a time. However, nowadays many computational tasks consist of many elementary operations, some of which had to be computed sequentially, while others could be computed in parallel.

In a distributed or high performance computing course, one of the first things you may learn is **Amdahl's law**, which gives a quantitative measure of the speedup S – i.e., how much faster the task can be performed by using n parallel processors:

$$S = \frac{1}{1 - p + \frac{p}{n}}.$$

Here, p is the proportion of operations that can be performed in parallel.

However, it is not typically the case that we can classify operations simply as "parallel" or "sequential." Instead, a task might consist of several sub-tasks, some of which need to be completed before others are started. Optimally scheduling these tasks is much trickier, and is currently an area of active research. However, with the right mathematical foundations, we can at least start to understand the complexity of task scheduling, and begin to develop efficient strategies for tackling "easy" versions of the problem.

In this supplement, you learn about a generalized form of functions known as **relations**, and explore the connection between functions and relations. You'll then learn about two important types of relations, including **equivalence relations** and **partial orders**. We'll explore some of the applications of these concepts, and finally return to our original problem of scheduling with multiple processors.

Part 1: Generalizing Functions with Relations

This week, you learned about functions. We (informally) defined them as a well-defined mapping between two sets. However, what about mappings that are not well-defined? Is there no way to represent these? As a matter of fact, there is!

Definition. a *relation* $\mathcal{R}$ over sets A, B is a subset of $A \times B$. The notation $a\mathcal{R}b$ or $a \sim b$ is often used to denote that $(a, b) \in \mathcal{R}$.

Easy Examples

Let $\mathcal{R}_1, \ldots, \mathcal{R}_4$ be a relation on $A = \{1, 2, 3, 4\}$.

- $\mathcal{R}_1 = \{(a, b) \mid a \leq b)\}$
- $\mathcal{R}_2 = \{(a, b) \mid a = b)\}$

- $\mathcal{R}_3 = \{(a, b) \mid a + b \leq 2022)\}$
- $\mathcal{R}_4 = \{(a, b) \mid a \text{ divides } b\}$

Another Example: Functions

Functions are also an example of relations. Specifically, a function is any relation $\mathcal{R} : A \to B$ with the property that for any x there is exactly one y such that $x\mathcal{R}y$.

Properties of A Relation $\mathcal{R}\colon A \to A$

Reflexitivty $\mathcal{R}$ is *reflexive* if

$(\forall a \in A)(a\mathcal{R}a)$
"Everyone has slept with themselves"

Symmetry $\mathcal{R}$ is *symmetric* if

$(\forall a, b \in A)(a\mathcal{R}b \iff b\mathcal{R}a)$
"If a slept with b, then b slept with a"

Antisymmetry $\mathcal{R}$ is *symmetric* if

$(\forall a, b \in A)(a\mathcal{R}b \wedge b\mathcal{R}a \to a = b)$
"No pair of distinct people have slept with each other"

Transitivity $\mathcal{R}$ is *transitive* if

$(\forall a, b, c \in A)(a\mathcal{R}b \wedge b\mathcal{R}c \to a\mathcal{R}c)$
"If a slept with b and b slept with c, then a slept with c too."

When a relation is reflexive, antisymmetric, and transitive, we call it a *partial order*. When a relation is reflexive, symmetric, and transitive, we call it a *equivalence relation*.

In this part, you'll implement the functions `isPartialOrder(elements, relation)` and `isEquivalenceRelation(elements, relation)`. These functions takes in a relation (represented as a list of tuples) and returns whether or not the relation (taken over the set of elements) is a valid partial order or equivalence relation, respectively. You must use the following helper methods:

- `isReflexive(elements, relation)`
- `isSymmetric(elements, relation)`
- `isAntisymmetric(elements, relation)`
- `isTransitive(elements, relation)`

All methods (and all helper methods) must be implemented in one line.
Tip: Use the `all()` method

Part 2: Partitioning with Equivalence Relations

In the world of computer science, there are two main applications for relations: partitioning and scheduling. In this part, we'll cover partitioning, which is essentially just a reframing of our knowledge about equivalence relations. If any of the concepts introduced in this section feel rather hand-wavy, feel free to consult the textbook, which has rigorous proofs for the theorems.

Definition: Given some relation $\mathcal{R}$ over the set A, the *equivalence class* of an element $x \in A$ is

$$[x] = \{y : x\mathcal{R}y\}$$

There is a very powerful theorem related to this concept:

Important Theorem: The equivalence classes of an equivalence relation on a set A *partition* A into a collection of disjoint, nonempty subsets $A_1, A_2, \ldots, A_n$ such that (and this is the important part) $\bigcup_{i=1}^{n} = A$.

Ok, so this fact is cool and all, but what's the point? Well, if we have say a million elements in a set, but only three equivalence classes, we need only consider those three classes to understand the properties of the entire set. This is where the concept of *canonical forms* comes in, which refers to the standard representation of a set in terms of its equivalence classes, making it easier to analyze and manipulate.

Example 1: Congruence Relations

Our first example delves into *number theory*, a field you will become more intimate with in the next few supplements. Informally, define the relation $a\mathcal{R}_nb$ over $\mathbb{Z} \times \mathbb{Z}$ if a and b have the same remainder when divided by some number n. We usually denote this using

$$a \equiv b \pmod{n}.$$

For example, -10 and 15 are related under $\mathcal{R}_5$,

$$-10 \equiv 15 \pmod{5},$$

since $-10 - 15 = -25$ is a multiple of 5, or equivalently since both -10 and 15 have the same remainder 0 when divided by 5.

All such relations $\mathcal{R}_n$ are equivalence relations, and partition the set of integers into n equivalence classes. For example, the relation $\mathcal{R}_3$ partitions the integers like so

$$\{\ldots, -7, -4, -1, 2, 5, 8, \ldots\}$$
$$\{\ldots, -8, -5, -2, 1, 4, 7, \ldots\}$$
$$\{\ldots, -6, -3, 0, 3, 6, 9, \ldots\}$$

The canonical forms for each of these sets are 2, 1, and 3 respectively (try to spot why!)

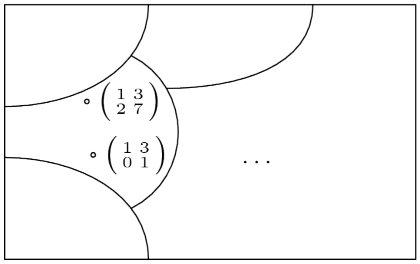
Example 2: Vector Spaces

One of the most prominent uses of equivalence classes is in linear algebra. After all, it's very difficult to determine when two vector spaces or matrices are functionally "identical".

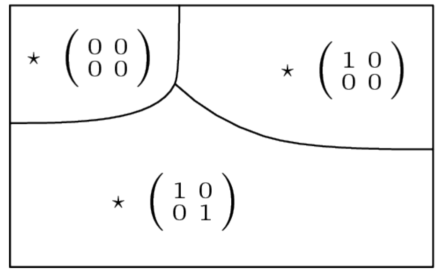
If you've taken an introductory linear algebra course, you learned about Gauss's Method to solve a system of equations. In essence, it worked by starting with a matrix and deriving a sequence of other matrices, each *row equivalent* to each other. It turns out that row equivalence is an equivalence relation, and partitions the set of all matrices into corresponding classes, as shown in Figure (1).

We can generalize this one step further with a relation known as *matrix equivalence*. Matrix equivalent matrices represent the same map, with respect to appropriate pairs of bases. Matrix equivalence classes are also characterized by rank: two same-sized matrices are matrix equivalent if and only if they have the same rank. In fact, matrix similarity is a special case of matrix equivalence!

The canonical forms for row equivalence are the Reduced Echelon form matrices, which you may already be familiar with. Meanwhile, the canonical forms for matrix similarity are Jordan Normal forms. (the canonical forms for matrix equivalence are a bit more complicated.)



(a) Two matrices with the same row space.



(b) The canonical forms for the set of 2×2 matrices.

Figure 1: Row equivalence as an equivalence relation.

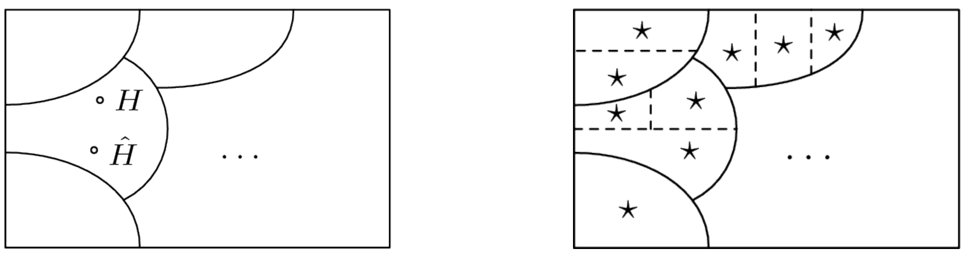


Figure 2: Matrix similarity is a finer partition than matrix equivalence

In this part, you'll implement the following function:

- `partition(elements, relation)`: This function takes a equivalence relation (this time represented as a boolean function), and returns a partition of the elements into equivalence classes. For example, given the congruence relation described above, the function should return

```
>>> partition([i for i in range(-8, 8)], lambda x, y: (x - y) % 3 == 0)
[{-6, -3, 0, 3, 6}, {-8, -5, -2, 1, 4, 7}, {-7, -4, -1, 2, 5, 8}]
```

Tip: iterating over sets is difficult. Try casting.

Part 3: Single Processor Task Scheduling

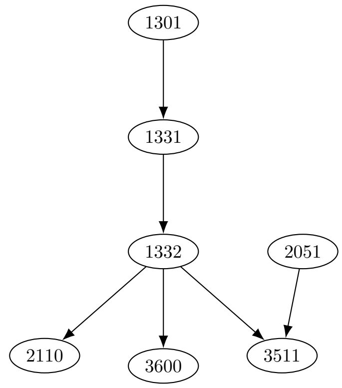
We now delve into partial orderings, or "posets". We'll cover a nice *graph*ical representation of posets, and then cover their applications in the context of task scheduling.

Dependency Graphs

To start, let's consider the following story.

You're a recently admitted CS major at Georgia Tech. To save money, you're trying to graduate as early as possible. You have a lot of course that you can take, but you're not sure which ones to take when. Furthermore, many of the courses are dependent upon completion of other courses (for example, CS 2110 requires CS 1331, which in turn requires CS 1301). How long will it take for you to graduate?

We can represent this problem as a poset $\mathcal{R}$ over the set of courses A, where $a\mathcal{R}b$ iff a is a prerequisite to take b:



This is known as a *dependency graph*.

Task Scheduling

Using the formalism of posets, we can also define the problem of computing an optimal schedule.

Assume that some task T can be decomposed into sub-tasks $T = \{T_1, T_2, \ldots, T_n\}$. In general, we can encode the relationships between the various T_i as a poset, where $T_i \prec T_j$ if (sub)task T_i needs to be performed before T_j. **Assume each task T_i takes 1 unit of time to complete.** Given k processors P_1, P_2, ..., P_k a **schedule** for T assigns each sub-task T_i a processor P_j as well as a time t_i at which P_j should start task T_i. Observe that if a processor starts T_i at times t_i, then the task will complete at time $t_i + 1$. A schedule is **feasible** if:

1 no single processor is performing multiple tasks at the same time (i.e., if a processor is assigned T_i and T_j at times i, j, then $i \neq j$) and

2 for every pair of tasks T_i and T_j with $T_i \prec T_j$, task T_j is scheduled to start some time after (or at the same time) T_i completes. (i.e. $T_i \prec T_j \Rightarrow t_j > t_i$)

Topological Sorting

If we have only one processor, then the answer is fairly simple, just create a topological sorting.
The proof for this statement will be added in a future update.

Definition: A topological sorting is a total ordering of a partially ordered set.

We present a common algorithm for finding the topological sort below, known as Kahn's algorithm. (Note: the pseudocode refers to *minimal elements*. These are simply all elements v such that there is no other element u such that $(u, v) \in \mathcal{R}$)

Algorithm 1 KahnsAlgorithm(*elements*, *poset*)

$T = \emptyset$
$S =$ all minimal elements in *poset*
while $S \neq \emptyset$ **do**
  $u =$ an element in S
  $S = S - \{u\}$
  $T = T \cup \{u\}$
  $uv\_dependencies = \{(u, v) : v \in elements \text{ and } (u, v) \in poset\}$
  for each (u, v) in $uv\_dependencies$ **do**
    if v is minimal **then** insert v in S
return T

In this part, you'll implement the following function:
`topological_sort(poset)`: This method takes in a partially ordered set defined on `elements` (in the form of a dependency list) and returns a valid topological sort. We can use it to solve our course scheduling dilemma above:

```
>>> topological_sort({'1301', '1331', '1332', '2110', '2051', '3600', '3511'},
        {('1301', '1331'), ('1331', '1332'), ('1331', '2110'),\
         ('2051', '3511'), ('1332', '3511'), ('1332', '3600')})
    ['2051', '1301', '1331', '2110', '1332', '3511', '3600']
```

Part 4: Multi-Processor Task Scheduling

Definition A *chain* is such that every distinct pair of elements is comparable in $\mathcal{P}$.
Note that the time it takes to schedule tasks, even with an unlimited number of processors, is at least the length of the longest chain. Indeed, if we used less time, then two items from a longest chain would have to be done at the same time, which contradicts the precedence constraints.

The nice thing about posets is that this is always possible! In other words, for any poset, there is a legal paralle schedule that runs in t steps, where t is the length of the longest chain. This is known as Mirsky's theorem:
The proof for this statement will be added in a future update.

The case where there are a limited number of processors, however, is more useful in practice, as we'll see below.

In this part, you'll implement the following function:

- `generate_schedule(elements, poset, num_processors)`: This method will solve the original problem defined in "Task Scheduling". It should return a valid schedule in the form of a list of lists, where the i-th element in the list represents the jobs we should schedule at time $t = i$.

  If, for instance, we could take two courses a semester, we could generate an optimal schedule as follows:

```
>>> generate_schedule({'1301', '1331', '1332', '2110', '2051', '3600', '3511'},
        {('1301', '1331'), ('1331', '1332'), ('1331', '2110'),\
         ('2051', '3511'), ('1332', '3511'), ('1332', '3600')}, 2)
    ['1301', '2051'], ['1331'], ['1332'], ['2110', '3600'], ['3511']]]
```

(Optional) Processor Optimization

What if we wanted to optimize the number of processors? In other words, what is the minimum number of processors such that increasing the number of processors does not decrease our latency? In the course scheduling example, it was easy to see that the optimal number of processors was 3 by just looking at the dependency graph above. If we wanted to do this in general, one way would be to run `generate_schedule` for all `num_processors` from 1 to `len(elements)`, and then return the minimum.

However, note that this algorithm is not efficient, since there is the possibility that we have to run `generate_schedule` many times. It seems natural to assume there is an efficient solution to this problem, and in fact there is! However, the details of the algorithm are far outside the scope of the course. If you really want to know, the details can be found in the MATH 3012 textbook.

This processor optimization problem is also known as the *dual* problem of Part 3. Duality is a fundamental concept in computer science, and you'll encounter it again in your Algorithms courses.

Submission Instructions (10 pts)

After you fill the appropriate functions, submit the following files to Gradescope and make sure you pass all test cases:

- `relations.py`
- `scheduler.py`

Notes

- The autograder may not reflect your final grade on the assignment. We reserve the right to run additional tests during grading.
- Do not import additional packages, as your submission may not pass the test cases or manual review.