CS/Math 240: Introduction to Discrete Mathematics

Reading 1 : Introduction

Author: Dieter van Melkebeek (updates by Beck Hasti and Gautam Prakriya)

Welcome to CS 240, an introduction to discrete mathematics. This reading provides an overview of the course and of the specifics for this particular offering, which is geared towards prospective Computer Science and Electrical and Computer Engineering majors. For the administrative arrangements, you are referred to a separate handout.

1.1 Course Overview

Discrete mathematics is the mathematical study of discrete structures. The word *discrete* here means *opposite of continuous*. In the continuous setting, we consider objects like real numbers that have the property of varying smoothly. In contrast, the objects studied in discrete mathematics do not vary smoothly in this way, but have distinct, separated values. Examples include the integers but also all finite sets. In fact, discrete structures can be characterized as those structures whose elements can be enumerated by integers, that is, we can say what the first element is, the second, and so on.

In this course we focus on discrete structures that are ubiquitous in computer science: integers, bits, strings, and graphs. The goal of the course is two-fold:

- making you familiar with those structures and related notions that are relevant to computer science, and
- developing your skills to reason *rigorously* about those structures and notions, especially in an algorithmic context.

1.1.1 Part I: Logic and Proofs

Rigorous means that all statements need to be precise. The language of mathematics is perfect for this purpose because every concept in mathematics is clearly and unambiguously defined by means of definitions, and nontrivial statements are given as propositions. In the first part of the course, we will cover the basics of *propositional and predicate logic* and of *set theory*, which will allow us to formulate precise statements in the rest of the course.

The other aspect of rigor is the use of *sound reasoning* to justify the truth of propositions. Such a justification is called a *proof*. The goal of a proof is to convince yourself or someone else without any doubt that a particular statement is true. Computer scientists are particularly interested in proofs that their programs work correctly on all cases (rather than just on a few test cases). In the first part of the course we will cover various common *proof paradigms*. You will have ample opportunity to practice writing proofs throughout the semester.

1.1.2 Part II: Induction and Recursion

In the second part of the course we will focus on the proof paradigm known as *induction* – arguably the single most important topic in this course. Mathematical induction is a general technique to show that some property holds for all objects of a discrete structure, and consists of two steps: the base case and the induction step.

1. In the base case we show that the property holds for the first element.

2. In the induction step we prove that, for every positive integer k, if the property holds for the kth element, then it also holds for the $(k+1)$st element.

At first, it may seem a bit convoluted a strategy, but it works and shows up naturally in many discrete settings. Of particular importance to us is its use in *program correctness*. The correctness of iterative programs is typically established via *loop invariants*, which are statements that hold each time the computer starts executing a loop iteration. In this setting, the base case establishes that the property holds the first time the start of the loop is reached, and the induction step shows that the loop maintains the property, i.e., if the property holds at the beginning of an iteration then it also holds at the end.

Induction is closely related to the notion of *recursion*, a term that can refer to both definitions and programs. A *recursive definition* of a concept consists of two parts:

1. A foundation rule which says what are the simplest instances of the concept being defined.

2. A constructor rule which says how to combine simpler instances of the concept being defined into a more complex instance.

Establishing properties of recursively defined concepts naturally goes by induction. The base case shows that the foundation rule produces instances that have the property, and the induction step shows that the constructor rule maintains the property. Such an application of induction is referred to as *structural induction*.

A *recursive program* breaks up a given instance of a problem into (possibly multiple) simpler instances of the same problem, and stipulates how to compose the solution to the given instance out of the solution to the simpler instances. To solve the simpler instances, the program calls itself, continuing the process of breaking up instances down into even simpler instances, up to the point where it reaches very simple instances, for which the program explicitly describes the solution. Proving the *correctness* of recursive programs naturally flows by induction on the number of levels of recursion. The base case corresponds to the very simple instances for which the solution is explicitly described in the program. The induction step shows that the composition procedure produces correct results assuming the recursive calls do. Combined with a *termination* argument that the recursion always bottoms out to the very simple instances, this completes the correctness proof of the recursive program.

We will cover induction and various algorithmic uses in depth. Apart from program correctness, we will also discuss *program analysis* , i.e., determining how many steps a given program takes. In the case of recursive programs, such analysis leads to recurrence relations. In order to keep the analysis manageable, we often focus on the *asymptotic behavior* for large inputs. We will introduce the customary notation for doing so, which some of you may already know from an introductory course on data structures (such as CS 300).

1.1.3 Part III: Graphs and Relations

The third part of the course deals with *graphs and relations*. Graphs are discrete structures consisting of nodes and edges, where each edge consists of a pair of nodes. We use graphs to represent relations among objects, for example, friendships among people. The nodes represent the objects (people), and the edges the pairs of objects that are related (pairs of people that are friends).

Graphs and relations are used all over in computer science. Of particular interest are *trees*, which represent hierarchical relations. A special type are *order relations*, where the hierarchy is

a linear order. Another category of relations that are of special interest are *equivalence relations*, which are intimately connected to partitions of the underlying universe of objects.

We will cover basic notions of graph theory that are relevant in computer science, such as connectivity, isomorphism, and planarity. We will also study *finite state automata*, which can be viewed as a special type of graphs that capture a simple model of computation. The model exactly captures the power of regular expressions, another ubiquitous concept in computer science.

1.1.4 Part IV: Combinatorics

The final part of the course is an introduction to *combinatorics*, the branch of discrete mathematics that counts the number of structures with a given property. Classical examples include *permutations*, *combinations*, and other ways to select a certain number of elements of a finite set under certain restrictions.

In computer science counting comes up in the analysis of the running time of programs, although those counting problems often have a different flavor than the above ones that are typically studied in combinatorics. A better computer science motivation for the latter is the analysis of randomized algorithms, which involves discrete probability theory. Probabilities with respect to the uniform distribution are equivalent to counting. As such, the latter part of the course forms a good introduction to and transition into a course on discrete probability theory (such as Math 331), which you may want to take next.

1.2 Relevance

CS/Math 240 is a foundational course that provides a sound basis for many more advanced courses in Computer Science. Here is a non-exhaustive list of areas and specific topics that are relevant in those areas.

1. **Algorithm Design** - Almost everything from this course is used in algorithm design.
2. **Programming Languages** - Needs set theory and logic. Finite automata and regular expressions are also used a lot.
3. **Databases** - Set theory, logic, and notions of relations are necessary
4. **Artificial Intelligence** - Logic is necessary, and so is set theory, and recursion.
5. **Systems** - Finite state machines are used to model processes.
6. **Networking** - Graphs are important because that's how we model networks.

1.3 Specifics of This Offering

Based on input of faculty from the Computer Sciences Department, three topics receive more emphasis than in standard courses in Discrete Mathematics using existing textbooks (some of which may not cover those topics at all).

- **Program Correctness**

  Whereas some universities cover program correctness in their programming sequence, for organizational reasons this is not the case at the University of Wisconsin. Several faculty

members strongly feel that the topic should be taught in one of the required introductory courses, so that all CS majors are exposed to it early on. Discrete Math seems like the next natural course to cover the topic, after the programming sequence. The topic fits in nicely as an interesting and – for many students – revealing application of induction.

As a consequence, some prior programming experience is expected from the students, although it does not have to be a formal course like CS 200.

- **Recursion**

  Some faculty members felt that CS majors needed more exposure to recursive programs, beyond the training they receive during the programming sequence. Thanks to its close ties to induction, the topic blends in well with induction. Reasoning about the correctness of recursive programs seems like a good way to become more comfortable with them.

- **Regular Expressions and Finite Automata**

  Given the ubiquity of regular expressions and the use of finite automata as a modeling tool in many branches of computer science, several faculty members felt that all CS majors should be exposed to those topics, and relatively early on. These topics are traditionally covered in the Introduction to Theory of Computing (CS 520). However, that course is not required, and the students who take it, typically do so late in their undergraduate career.

1.4 About These Readings

These readings started as scribe notes taken by Dalibor Zelený when Dieter van Melkebeek taught the course in Spring 2011. They were subsequently reviewed by Dieter van Melkebeek and later updated by Beck Hasti and Gautam Prakriya for the Fall 2015 offering of the course.

1.5 Example - Stable Matching Problem

We now look at an application of some of the topics listed above. In this section, we describe the stable matching problem and present an algorithm as a solution. However we defer the proof of correctness of this algorithm.

The computer sciences department in some university has n graduate students and n professors. Each professor wants to hire one student, and each student is looking for an advisor. We play the role of the graduate program co-ordinator and are tasked with assigning students to professors based on preference lists that the students and professors submit. Each student submits a list ranking professors in order of preference and each professor submits a similar list ranking the students. We assume that these lists have no ties.

Our goal is to find a *stable matching* of students and advisors. An assignment of students to professors is called a stable matching if the following conditions are met:

1. The assignment is a *matching*, that is we require that in the assignment every student has exactly one advisor and every professor has exactly one student.

2. The matching of students and professors is *stable*. We say a matching is stable if it doesn't contain disruptive pairs. (See definition below.)

Definition 1.1. *Disruptive pair: In any matching, a student s and a professor p form disruptive pair if and only if*

1. *s and p are not paired together in the matching.*
2. *s prefers p to their current advisor (the professor they are paired with) and p prefers s to the student they are paired with. In other words, both s and p prefer each other to the individuals they are paired with in the matching.*

Notice that a matching in which everyone is assigned their top choice is in fact a stable matching. Unfortunately, pairing everyone with their top choice doesn't always result in a matching. For instance, this is the case when multiple students have the same professor as their first choice. On the other hand, regardless of what the preference lists are, there is always a stable matching. This statement requires a proof. We prove this statement by providing an algorithm which always returns a stable matching. For the proof to be complete, we will need to prove that the algorithm always returns a stable matching, but as we mentioned earlier the proof of correctness of the algorithm is deferred.

We now describe the matching algorithm. While this algorithm can be implemented using any programming language, we will use a metaphor to describe the algorithm.

For each day that some student is rejected do:

- Morning: Each student applies to their top remaining choice.
- Afternoon: Each professor says "Maybe, come back tomorrow" to their top choice applicant and says "NO" to the rest.
- Evening: Each of the rejected student crosses of the professor from their list.

Each student is paired with the professor who last said "Maybe" to them.

We can break down the proof of correctness of the algorithm into the following theorems:

Theorem 1.2. *If the algorithm tertminates, it will return a stable matching.*

Theorem 1.3. *The algorithm always terminates.*

As mentioned earlier, we will prove these statements once we have built our arsenal of proof techniques. Note that there could potentially be many stable matchings; the algorithm outputs one of them. We now investigate some additional properties of the stable matching constructed by the algorithm. We will need a few definitions.

Definition 1.4. *We say that a professor p is **optimal** for a student s if p is the highest ranked professor for whom there is some stable matching that pairs s and p.*

Definition 1.5. *A professor p is **pessimal** for a student s, if p is the lowest ranked professor for whom there is some stable matching pairing s and p.*

We can similarly define what it means for student to be optimal/pessimal for a professor.

Definition 1.6. *We say that a stable matching is **student optimal** if every student is paired with their* optimal *professor. We say a stable matching is **professor pessimal** if every professor is paired with their* pessimal *student.*

Theorem 1.7. *The matching algorithm described above outputs a stable matching that is both student optimal and professor pessimal.*

Variants of the problem described here appear in a wide range of real world situations including assigning medical residents to their first hospital, matching students to schools, and the assignment of human organs for transplant to recepients. The 2012 Nobel Prize in Economics was awarded to Lloyd S. Shapley and Alvin E. Roth "for the theory of stable allocations and the practice of market design."