Part I

Proofs

Part II

Mathematical Data Types

Chapter 5

Sets and Relations

5.1 Sets

Informally, a *set* is a bunch of objects, which are called the *elements* of the set. The elements of a set can be just about anything: numbers, points in space, or even other sets. The conventional way to write down a set is to list the elements inside curly-braces. For example, here are some sets:

$$\begin{array}{lcll} A & = & \{\text{Alex}, \text{Tippy}, \text{Shells}, \text{Shadow}\} & \text{dead pets} \\ B & = & \{\text{red}, \text{blue}, \text{yellow}\} & \text{primary colors} \\ C & = & \{\{a, b\}, \{a, c\}, \{b, c\}\} & \text{a set of sets} \end{array}$$

This works fine for small finite sets. Other sets might be defined by indicating how to generate a list of them:

$$D = \{1, 2, 4, 8, 16, \ldots\} \qquad \text{the powers of 2}$$

The order of elements is not significant, so $\{x, y\}$ and $\{y, x\}$ are the same set written two different ways. Also, any object is, or is not, an element of a given set —there is no notion of an element appearing more than once in a set.[1] So writing $\{x, x\}$ is just indicating the same thing twice, namely, that x is in the set. In particular, $\{x, x\} = \{x\}$.

The expression $e \in S$ asserts that e is an element of set S. For example, $32 \in D$ and blue $\in B$, but Tailspin $\notin A$ —yet.

Sets are simple, flexible, and everywhere. You'll find some set mentioned in nearly every section of this text.

5.1.1 Some Popular Sets

Mathematicians have devised special symbols to represent some common sets.

[1] It's not hard to develop a notion of *multisets* in which elements can occur more than once, but multisets are not ordinary sets.

symbol	set	elements
$\emptyset$	the empty set	none
$\mathbb{N}$	nonnegative integers	$\{0, 1, 2, 3, \ldots\}$
$\mathbb{Z}$	integers	$\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$
$\mathbb{Q}$	rational numbers	$\frac{1}{2},\ -\frac{5}{3},\ 16,$ etc.
$\mathbb{R}$	real numbers	$\pi,\ e,\ -9,\ \sqrt{2},$ etc.
$\mathbb{C}$	complex numbers	$i,\ \frac{19}{2},\ \sqrt{2} - 2i,$ etc.

A superscript "$^+$" restricts a set to its positive elements; for example, $\mathbb{R}^+$ denotes the set of positive real numbers. Similarly, $\mathbb{R}^-$ denotes the set of negative reals.

5.1.2 Comparing and Combining Sets

The expression $S \subseteq T$ indicates that set S is a *subset* of set T, which means that every element of S is also an element of T (it could be that $S = T$). For example, $\mathbb{N} \subseteq \mathbb{Z}$ and $\mathbb{Q} \subseteq \mathbb{R}$ (every rational number is a real number), but $\mathbb{C} \not\subseteq \mathbb{Z}$ (not every complex number is an integer).

As a memory trick, notice that the $\subseteq$ points to the smaller set, just like a $\leq$ sign points to the smaller number. Actually, this connection goes a little further: there is a symbol $\subset$ analogous to $<$. Thus, $S \subset T$ means that S is a subset of T, but the two are *not* equal. So $A \subseteq A$, but $A \not\subset A$, for every set A.

There are several ways to combine sets. Let's define a couple of sets for use in examples:

$$X ::= \{1, 2, 3\}$$
$$Y ::= \{2, 3, 4\}$$

- The *union* of sets X and Y (denoted $X \cup Y$) contains all elements appearing in X or Y or both. Thus, $X \cup Y = \{1, 2, 3, 4\}$.
- The *intersection* of X and Y (denoted $X \cap Y$) consists of all elements that appear in *both* X and Y. So $X \cap Y = \{2, 3\}$.
- The *set difference* of X and Y (denoted $X - Y$) consists of all elements that are in X, but not in Y. Therefore, $X - Y = \{1\}$ and $Y - X = \{4\}$.

5.1.3 Complement of a Set

Sometimes we are focused on a particular domain, D. Then for any subset, A, of D, we define $\overline{A}$ to be the set of all elements of D *not* in A. That is, $\overline{A} ::= D - A$. The set $\overline{A}$ is called the *complement* of A.

For example, when the domain we're working with is the real numbers, the complement of the positive real numbers is the set of negative real numbers together with zero. That is,

$$\overline{\mathbb{R}^+} = \mathbb{R}^- \cup \{0\}\,.$$

It can be helpful to rephrase properties of sets using complements. For example, two sets, A and B, are said to be *disjoint* iff they have no elements in common, that is, $A \cap B = \emptyset$. This is the same as saying that A is a subset of the complement of B, that is, $A \subseteq \overline{B}$.

5.1.4 Power Set

The set of all the subsets of a set, A, is called the *power set*, $\mathcal{P}(A)$, of A. So $B \in \mathcal{P}(A)$ iff $B \subseteq A$. For example, the elements of $\mathcal{P}(\{1,2\})$ are $\emptyset, \{1\}, \{2\}$ and $\{1,2\}$.

More generally, if A has n elements, then there are 2^n sets in $\mathcal{P}(A)$. For this reason, some authors use the notation 2^A instead of $\mathcal{P}(A)$.

5.1.5 Set Builder Notation

An important use of predicates is in *set builder notation*. We'll often want to talk about sets that cannot be described very well by listing the elements explicitly or by taking unions, intersections, etc., of easily-described sets. Set builder notation often comes to the rescue. The idea is to define a *set* using a *predicate*; in particular, the set consists of all values that make the predicate true. Here are some examples of set builder notation:

$$\begin{aligned} A &::= \{n \in \mathbb{N} \mid n \text{ is a prime and } n = 4k+1 \text{ for some integer } k\} \\ B &::= \left\{x \in \mathbb{R} \mid x^3 - 3x + 1 > 0\right\} \\ C &::= \left\{a + bi \in \mathbb{C} \mid a^2 + 2b^2 \leq 1\right\} \end{aligned}$$

The set A consists of all nonnegative integers n for which the predicate

$$\text{“}n \text{ is a prime and } n = 4k+1 \text{ for some integer } k\text{”}$$

is true. Thus, the smallest elements of A are:

$$5, 13, 17, 29, 37, 41, 53, 57, 61, 73, \ldots.$$

Trying to indicate the set A by listing these first few elements wouldn't work very well; even after ten terms, the pattern is not obvious! Similarly, the set B consists of all real numbers x for which the predicate

$$x^3 - 3x + 1 > 0$$

is true. In this case, an explicit description of the set B in terms of intervals would require solving a cubic equation. Finally, set C consists of all complex numbers $a + bi$ such that:

$$a^2 + 2b^2 \leq 1$$

This is an oval-shaped region around the origin in the complex plane.

5.1.6 Proving Set Equalities

Two sets are defined to be equal if they contain the same elements. That is, $X = Y$ means that $z \in X$ if and only if $z \in Y$, for all elements, z. (This is actually the first of the ZFC axioms.) So set equalities can be formulated and proved as "iff" theorems. For example:

Theorem 5.1.1 (*Distributive Law* for Sets)**.** *Let A, B, and C be sets. Then:*

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C) \tag{5.1}$$

Proof. The equality (5.1) is equivalent to the assertion that

$$z \in A \cap (B \cup C) \quad \text{iff} \quad z \in (A \cap B) \cup (A \cap C) \tag{5.2}$$

for all z. Now we'll prove (5.2) by a chain of iff's.

First we need a rule for distributing a propositional AND operation over an OR operation. It's easy to verify by truth-table that

Lemma 5.1.2. *The propositional formulas*

$$P \text{ AND } (Q \text{ OR } R)$$

and

$$(P \text{ AND } Q) \text{ OR } (P \text{ AND } R)$$

are equivalent.

Now we have

$$\begin{array}{lll}
z \in A \cap (B \cup C) & & \\
\quad \text{iff} & (z \in A) \text{ AND } (z \in B \cup C) & (\text{def of } \cap) \\
\quad \text{iff} & (z \in A) \text{ AND } (z \in B \text{ OR } z \in C) & (\text{def of } \cup) \\
\quad \text{iff} & (z \in A \text{ AND } z \in B) \text{ OR } (z \in A \text{ AND } z \in C) & (\text{Lemma 5.1.2}) \\
\quad \text{iff} & (z \in A \cap B) \text{ OR } (z \in A \cap C) & (\text{def of } \cap) \\
\quad \text{iff} & z \in (A \cap B) \cup (A \cap C) & (\text{def of } \cup)
\end{array}$$

■

5.1.7 Glossary of Symbols

symbol	meaning
$::=$	is defined to be
$\wedge$	and
$\vee$	or
$\longrightarrow$	implies
$\neg$	not
$\neg P$	not P
$\overline{P}$	not P
$\longleftrightarrow$	iff, equivalent
$\oplus$	xor
$\exists$	exists
$\forall$	for all
$\in$	is a member of, belongs to
$\subseteq$	is a subset of, is contained by
$\subset$	is a proper subset of, is properly contained by
$\cup$	set union
$\cap$	set intersection
$\overline{A}$	complement of the set A
$\mathcal{P}(A)$	powerset of the set A
$\emptyset$	the empty set, $\{\}$
$\mathbb{N}$	nonnegative integers
$\mathbb{Z}$	integers
$\mathbb{Z}^+$	positive integers
$\mathbb{Z}^-$	negative integers
$\mathbb{Q}$	rational numbers
$\mathbb{R}$	real numbers
$\mathbb{C}$	complex numbers

5.1.8 Problems

Homework Problems

Problem 5.1.
Let A, B, and C be sets. Prove that:

$$A \cup B \cup C = (A - B) \cup (B - C) \cup (C - A) \cup (A \cap B \cap C). \tag{5.3}$$

Hint: P OR Q OR R is equivalent to

$$(P \text{ AND } \overline{Q}) \text{ OR } (Q \text{ AND } \overline{R}) \text{ OR } (R \text{ AND } \overline{P}) \text{ OR } (P \text{ AND } Q \text{ AND } R).$$

5.2 The Logic of Sets

5.2.1 Russell's Paradox

Reasoning naively about sets turns out to be risky. In fact, one of the earliest attempts to come up with precise axioms for sets by a late nineteenth century logican named Gotlob *Frege* was shot down by a three line argument known as *Russell's Paradox*:[2] This was an astonishing blow to efforts to provide an axiomatic foundation for mathematics.

> Let S be a variable ranging over all sets, and define
>
> $$W ::= \{S \mid S \notin S\}.$$
>
> So by definition,
>
> $$S \in W \text{ iff } S \notin S,$$
>
> for every set S. In particular, we can let S be W, and obtain the contradictory result that
>
> $$W \in W \text{ iff } W \notin W.$$

A way out of the paradox was clear to Russell and others at the time: *it's unjustified to assume that W is a set*. So the step in the proof where we let S be W has no justification, because S ranges over sets, and W may not be a set. In fact, the paradox implies that W had better not be a set!

But denying that W is a set means we must *reject* the very natural axiom that every mathematically well-defined collection of elements is actually a set. So the problem faced by Frege, Russell and their colleagues was how to specify *which* well-defined collections are sets. Russell and his fellow Cambridge University colleague Whitehead immediately went to work on this problem. They spent a dozen years developing a huge new axiom system in an even huger monograph called *Principia Mathematica*.

5.2.2 The ZFC Axioms for Sets

It's generally agreed that, using some simple logical deduction rules, essentially all of mathematics can be derived from some axioms about sets called the Axioms of Zermelo-Frankel Set Theory with Choice (ZFC).

We're *not* going to be working with these axioms in this course, but we thought

[2] Bertrand *Russell* was a mathematician/logician at Cambridge University at the turn of the Twentieth Century. He reported that when he felt too old to do mathematics, he began to study and write about philosophy, and when he was no longer smart enough to do philosophy, he began writing about politics. He was jailed as a conscientious objector during World War I. For his extensive philosophical and political writing, he won a Nobel Prize for Literature.

you might like to see them –and while you're at it, get some practice reading quantified formulas:

Extensionality. Two sets are equal if they have the same members. In formal logical notation, this would be stated as:

$$(\forall z.\ (z \in x \text{ IFF } z \in y)) \text{ IMPLIES } x = y.$$

Pairing. For any two sets x and y, there is a set, $\{x, y\}$, with x and y as its only elements:

$$\forall x, y.\ \exists u.\ \forall z.\ [z \in u \text{ IFF } (z = x \text{ OR } z = y)]$$

Union. The union, u, of a collection, z, of sets is also a set:

$$\forall z.\ \exists u \forall x.\ (\exists y.\ x \in y \text{ AND } y \in z) \text{ IFF } x \in u.$$

Infinity. There is an infinite set. Specifically, there is a nonempty set, x, such that for any set $y \in x$, the set $\{y\}$ is also a member of x.

Power Set. All the subsets of a set form another set:

$$\forall x.\ \exists p.\ \forall u.\ u \subseteq x \text{ IFF } u \in p.$$

Replacement. Suppose a formula, ϕ, of set theory defines the graph of a function, that is,

$$\forall x, y, z.\ [\phi(x, y) \text{ AND } \phi(x, z)] \text{ IMPLIES } y = z.$$

Then the image of any set, s, under that function is also a set, t. Namely,

$$\forall s\ \exists t\ \forall y.\ [\exists x.\ \phi(x, y) \text{ IFF } y \in t].$$

Foundation. There cannot be an infinite sequence

$$\cdots \in x_n \in \cdots \in x_1 \in x_0$$

of sets each of which is a member of the previous one. This is equivalent to saying every nonempty set has a "member-minimal" element. Namely, define

$$\text{member-minimal}(m, x) ::= [m \in x \text{ AND } \forall y \in x.\ y \notin m].$$

Then the Foundation axiom is

$$\forall x.\ x \neq \emptyset \text{ IMPLIES } \exists m.\ \text{member-minimal}(m, x).$$

Choice. Given a set, s, whose members are nonempty sets no two of which have any element in common, then there is a set, c, consisting of exactly one element from each set in s.

5.2.3 Avoiding Russell's Paradox

These modern ZFC axioms for set theory are much simpler than the system Russell and Whitehead first came up with to avoid paradox. In fact, the ZFC axioms are as simple and intuitive as Frege's original axioms, with one technical addition: the Foundation axiom. Foundation captures the intuitive idea that sets must be built up from "simpler" sets in certain standard ways. And in particular, Foundation implies that no set is ever a member of itself. So the modern resolution of Russell's paradox goes as follows: since $S \notin S$ for all sets S, it follows that W, defined above, contains every set. This means W can't be a set —or it would be a member of itself.

5.2.4 Does All This Really Work?

So this is where mainstream mathematics stands today: there is a handful of ZFC axioms from which virtually everything else in mathematics can be logically derived. This sounds like a rosy situation, but there are several dark clouds, suggesting that the essence of truth in mathematics is not completely resolved.

- The ZFC axioms weren't etched in stone by God. Instead, they were mostly made up by some guy named Zermelo. Probably some days he forgot his house keys.

  So maybe Zermelo, just like Frege, didn't get his axioms right and will be shot down by some successor to Russell who will use Zermelo's axioms to prove a proposition P and its negation NOT P. Then math would be broken. This sounds crazy, but after all, it has happened before.

  In fact, while there is broad agreement that the ZFC axioms are capable of proving all of standard mathematics, the axioms have some further consequences that sound paradoxical. For example, the Banach-Tarski Theorem says that, as a consequence of the Axiom of Choice, a solid ball can be divided into six pieces and then the pieces can be rigidly rearranged to give *two* solid balls, each the same size as the original!

- Georg *Cantor* was a contemporary of Frege and Russell who first developed the theory of infinite sizes (because he thought he needed it in his study of Fourier series). Cantor raised the question whether there is a set whose size is strictly between the "smallest[3]" infinite set, $\mathbb{N}$, and $\mathcal{P}(\mathbb{N})$; he guessed not:

  Cantor's *Continuum Hypothesis*: There is no set, A, such that $\mathcal{P}(\mathbb{N})$ is strictly bigger than A and A is strictly bigger than $\mathbb{N}$.

  The Continuum Hypothesis remains an open problem a century later. Its difficulty arises from one of the deepest results in modern Set Theory — discovered in part by Gödel in the 1930's and Paul Cohen in the 1960's — namely, the ZFC axioms are not sufficient to settle the Continuum Hypothesis: there are two collections of sets, each obeying the laws of ZFC, and in

[3]See Problem 5.3

one collection the Continuum Hypothesis is true, and in the other it is false. So settling the Continuum Hypothesis requires a new understanding of what Sets should be to arrive at persuasive new axioms that extend ZFC and are strong enough to determine the truth of the Continuum Hypothesis one way or the other.

- But even if we use more or different axioms about sets, there are some unavoidable problems. In the 1930's, Gödel proved that, assuming that an axiom system like ZFC is consistent —meaning you can't prove both P and NOT(P) for any proposition, P —then the very proposition that the system is consistent (which is not too hard to express as a logical formula) cannot be proved in the system. In other words, no consistent system is strong enough to verify itself.

5.3 Sequences

Sets provide one way to group a collection of objects. Another way is in a *sequence*, which is a list of objects called *terms* or *components*. Short sequences are commonly described by listing the elements between parentheses; for example, (a, b, c) is a sequence with three terms.

While both sets and sequences perform a gathering role, there are several differences.

- The elements of a set are required to be distinct, but terms in a sequence can be the same. Thus, (a, b, a) is a valid sequence of length three, but $\{a, b, a\}$ is a set with two elements —not three.
- The terms in a sequence have a specified order, but the elements of a set do not. For example, (a, b, c) and (a, c, b) are different sequences, but $\{a, b, c\}$ and $\{a, c, b\}$ are the same set.
- Texts differ on notation for the *empty sequence*; we use λ for the empty sequence.

The product operation is one link between sets and sequences. A *product of sets*, $S_1 \times S_2 \times \cdots \times S_n$, is a new set consisting of all sequences where the first component is drawn from S_1, the second from S_2, and so forth. For example, $\mathbb{N} \times \{a, b\}$ is the set of all pairs whose first element is a nonnegative integer and whose second element is an a or a b:

$$\mathbb{N} \times \{a, b\} = \{(0, a), (0, b), (1, a), (1, b), (2, a), (2, b), \dots\}$$

A product of n copies of a set S is denoted S^n. For example, $\{0, 1\}^3$ is the set of all 3-bit sequences:

$$\{0, 1\}^3 = \{(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)\}$$

5.4 Functions

A *function* assigns an element of one set, called the *domain*, to elements of another set, called the *codomain*. The notation

$$f : A \to B$$

indicates that f is a function with domain, A, and codomain, B. The familiar notation "$f(a) = b$" indicates that f assigns the element $b \in B$ to a. Here b would be called the *value* of f at *argument* a.

Functions are often defined by formulas as in:

$$f_1(x) ::= \frac{1}{x^2}$$

where x is a real-valued variable, or

$$f_2(y, z) ::= y10yz$$

where y and z range over binary strings, or

$$f_3(x, n) ::= \text{ the pair } (n, x)$$

where n ranges over the nonnegative integers.

A function with a finite domain could be specified by a table that shows the value of the function at each element of the domain. For example, a function $f_4(P, Q)$ where P and Q are propositional variables is specified by:

P	Q	$f_4(P, Q)$
T	**T**	**T**
T	**F**	**F**
F	**T**	**T**
F	**F**	**T**

Notice that f_4 could also have been described by a formula:

$$f_4(P, Q) ::= [P \text{ IMPLIES } Q].$$

A function might also be defined by a procedure for computing its value at any element of its domain, or by some other kind of specification. For example, define $f_5(y)$ to be the length of a left to right search of the bits in the binary string y until a 1 appears, so

$$\begin{aligned} f_5(0010) &= 3, \\ f_5(100) &= 1, \\ f_5(0000) &\text{ is undefined.} \end{aligned}$$

Notice that f_5 does not assign a value to any string of just 0's. This illustrates an important fact about functions: they need not assign a value to every element in

the domain. In fact this came up in our first example $f_1(x) = 1/x^2$, which does not assign a value to 0. So in general, functions may be *partial functions*, meaning that there may be domain elements for which the function is not defined. If a function is defined on every element of its domain, it is called a *total function*.

It's often useful to find the set of values a function takes when applied to the elements in *a set* of arguments. So if $f : A \to B$, and S is a subset of A, we define $f(S)$ to be the set of all the values that f takes when it is applied to elements of S. That is,

$$f(S) ::= \{b \in B \mid f(s) = b \text{ for some } s \in S\}.$$

For example, if we let $[r, s]$ denote the interval from r to s on the real line, then $f_1([1, 2]) = [1/4, 1]$.

For another example, let's take the "search for a `1`" function, f_5. If we let X be the set of binary words which start with an even number of `0`'s followed by a `1`, then $f_5(X)$ would be the odd nonnegative integers.

Applying f to a set, S, of arguments is referred to as "applying f pointwise to S", and the set $f(S)$ is referred to as the *image* of S under f.[4] The set of values that arise from applying f to all possible arguments is called the *range* of f. That is,

$$\text{range}(f) ::= f(\text{domain}(f)).$$

Some authors refer to the codomain as the range of a function, but they shouldn't. The distinction between the range and codomain will be important in Sections 5.5.4 and 5.6 when we relate sizes of sets to properties of functions between them.

5.4.1 Function Composition

Doing things step by step is a universal idea. Taking a walk is a literal example, but so is cooking from a recipe, executing a computer program, evaluating a formula, and recovering from substance abuse.

Abstractly, taking a step amounts to applying a function, and going step by step corresponds to applying functions one after the other. This is captured by the operation of *composing* functions. Composing the functions f and g means that first f applied is to some argument, x, to produce $f(x)$, and then g is applied to that result to produce $g(f(x))$.

Definition 5.4.1. For functions $f : A \to B$ and $g : B \to C$, the *composition*, $g \circ f$, of g with f is defined to be the function $h : A \to C$ defined by the rule:

$$h(x) ::= (g \circ f)(x) ::= g(f(x)),$$

for all $x \in A$.

Function composition is familiar as a basic concept from elementary calculus, and it plays an equally basic role in discrete mathematics.

[4] There is a picky distinction between the function f which applies to elements of A and the function which applies f pointwise to subsets of A, because the domain of f is A, while the domain of pointwise-f is $\mathcal{P}(A)$. It is usually clear from context whether f or pointwise-f is meant, so there is no harm in overloading the symbol f in this way.

5.5 Relations

Relations are another fundamental mathematical data type. Equality and "less-than" are very familiar examples of mathematical relations. These are called *binary relations* because they apply to a pair (a, b) of objects; the equality relation holds for the pair when $a = b$, and less-than holds when a and b are real numbers and $a < b$.

In this section we'll define some basic vocabulary and properties of binary relations.

5.5.1 Binary Relations and Functions

Binary relations are far more general than equality or less-than. Here's the official definition:

Definition 5.5.1. A *binary relation*, R, consists of a set, A, called the *domain* of R, a set, B, called the *codomain* of R, and a subset of $A \times B$ called the *graph of* R.

Notice that Definition 5.5.1 is exactly the same as the definition in Section 5.4 of a *function*, except that it doesn't require the functional condition that, for each domain element, a, there is *at most* one pair in the graph whose first coordinate is a. So a function is a special case of a binary relation.

A relation whose domain is A and codomain is B is said to be "between A and B", or "from A to B." When the domain and codomain are the same set, A, we simply say the relation is "on A." It's common to use infix notation "$a \ R \ b$" to mean that the pair (a, b) is in the graph of R.

For example, we can define an "in-charge of" relation, T, for MIT in Spring '10 to have domain equal to the set, F, of names of the faculty and codomain equal to all the set, N, of subject numbers in the current catalogue. The graph of T contains precisely the pairs of the form

$$(\langle\text{instructor-name}\rangle, \langle\text{subject-num}\rangle)$$

such that the faculty member named ⟨instructor-name⟩ is in charge of the subject with number ⟨subject-num⟩ in Spring '10. So graph (T) contains pairs like

```
(A. R. Meyer,  6.042),
(A. R. Meyer,  18.062),
(A. R. Meyer,  6.844),
(T. Leighton,  6.042),
(T. Leighton,  18.062),
(G, Freeman,   6.011),
(G, Freeman,   6.UAT),
(G. Freeman,   6.881)
(G. Freeman,   6.882)
(G. Freeman,   6.UAT)
(T. Eng,       6.UAT)
(J. Guttag,    6.00)
   ⋮
```

This is a surprisingly complicated relation: Meyer is in charge of subjects with three numbers. Leighton is also in charge of subjects with two of these three numbers —because the same subject, Mathematics for Computer Science, has two numbers: 6.042 and 18.062, and Meyer and Leighton are co-in-charge of the subject. Freeman is in-charge of even more subjects numbers (around 20), since as Department Education Officer, he is in charge of whole blocks of special subject numbers. Some subjects, like 6.844 and 6.00 have only one person in-charge. Some faculty, like Guttag, are in charge of only one subject number, and no one else is co-in-charge of his subject, 6.00.

Some subjects in the codomain, N, do not appear in the list —that is, they are not an element of any of the pairs in the graph of T; these are the Fall term only subjects. Similarly, there are faculty in the domain, F, who do not appear in the list because all their in-charge subjects are Fall term only.

5.5.2 Relational Images

The idea of the image of a set under a function extends directly to relations.

Definition 5.5.2. The *image* of a set, Y, under a relation, R, written $R(Y)$, is the set of elements of the codomain, B, of R that are related to some element in Y, namely,

$$R(Y) ::= \{b \in B \mid yRb \text{ for some } y \in Y\}.$$

For example, to find the subject numbers that Meyer is in charge of in Spring '09, we can look for all the pairs of the form

$$(\text{A. Meyer}, \langle \text{subject-number} \rangle)$$

in the graph of the teaching relation, T, and then just list the right hand sides of these pairs. These righthand sides are exactly the image $T(\text{A. Meyer})$, which happens to be $\{6.042, 18.062, 6.844\}$. Similarly, to find the subject numbers that

either Freeman or Eng are in charge of, we can collect all the pairs in T of the form

$$(\text{G. Freeman}, \langle\text{subject-number}\rangle)$$

or

$$(\text{T. Eng}, \langle\text{subject-number}\rangle);$$

and list their right hand sides. These right hand sides are exactly the image $T(\{\text{G. Freeman, T. Eng}\}$. So the partial list of pairs in T given above implies that

$$\{\texttt{6.011, 6.881, 6.882, 6.UAT}\} \subseteq T(\{\text{G. Freeman, T. Eng}\}.$$

Finally, since the domain, F, is the set of all in-charge faculty, $T(F)$ is exactly the set of *all* Spring '09 subjects being taught.

5.5.3 Inverse Relations and Images

Definition 5.5.3. The *inverse*, R^{-1} of a relation $R : A \to B$ is the relation from B to A defined by the rule

$$bR^{-1}a \text{ IFF } a\ R\ B.$$

The image of a set under the relation, R^{-1}, is called the *inverse image* of the set. That is, the inverse image of a set, X, under the relation, R, is $R^{-1}X$.

Continuing with the in-charge example above, we can find the faculty in charge of 6.UAT in Spring '10 can be found by taking the pairs of the form

$$(\langle\text{instructor-name}\rangle, 6.UAT)$$

in the graph of the teaching relation, T, and then just listing the left hand sides of these pairs; these turn out to be just Eng and Freeman. These left hand sides are exactly the inverse image of {6.UAT} under T.

Now let D be the set of introductory course 6 subject numbers. These are the subject numbers that start with *6.0*. Now we can likewise find out all the instructors who were in-charge of introductory course 6 subjects in Spring '09, by taking all the pairs of the form $(\langle\text{instructor-name}\rangle, 6.0\ldots)$ and list the left hand sides of these pairs. These left hand sides are exactly the inverse image of of D under T. From the part of the graph of T shown above, we can see that

$$\{\text{Meyer, Leighton, Freeman, Guttag}\} \subseteq T^{-1}(D).$$

That is, Meyer, Leighton, Freeman, and Guttag were among the instructors in charge of introductory subjects in Spring '10. Finally, the inverse image under T of the set, N, of all subject numbers is the set of all instructors who were in charge of a Spring '09 subject.

It gets interesting when we write composite expressions mixing images, inverse images and set operations. For example, $T(T^{-1}(D))$ is the set of Spring '09 subjects that have an instructor in charge who also is in in charge of an introductory subject. So $T(T^{-1}(D)) - D$ are the advanced subjects with someone in-charge who is also in-charge of an introductory subject. Similarly, $T^{-1}(D) \cap T^{-1}(N - D)$ is the set of faculty in charge of both an introductory *and* an advanced subject in Spring '09.

5.5.4 Surjective and Injective Relations

There are a few properties of relations that will be useful when we take up the topic of counting because they imply certain relations between the *sizes* of domains and codomains. We say a binary relation $R : A \to B$ is:

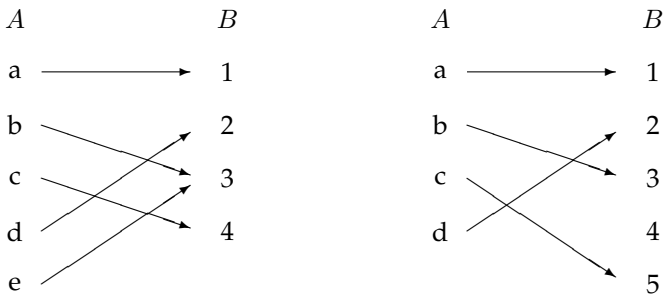
- *total* when every element of A is assigned to some element of B; more concisely, R is total iff $A = RB$.
- *surjective* when every element of B is mapped to *at least once*; more concisely, R is surjective iff $R(A) = B$.
- *total* when every element of A is assigned to some element of B; more concisely, R is total iff $A = R^{-1}(B)$.
- *injective* if every element of B is mapped to *at most once*, and
- *bijective* if R is total, surjective, and injective *function*.[5]

Note that this definition of R being total agrees with the definition in Section 5.4 when R is a function.

If R is a binary relation from A to B, we define $R(A)$ to to be the *range* of R. So a relation is surjective iff its range equals its codomain. Again, in the case that R is a function, these definitions of "range" and "total" agree with the definitions in Section 5.4.

5.5.5 Relation Diagrams

We can explain all these properties of a relation $R : A \to B$ in terms of a diagram where all the elements of the domain, A, appear in one column (a very long one if A is infinite) and all the elements of the codomain, B, appear in another column, and we draw an arrow from a point a in the first column to a point b in the second column when a is related to b by R. For example, here are diagrams for two functions:



[5]These words "surjective," "injective," and "bijective" are not very memorable. Some authors use the possibly more memorable phrases *onto* for surjective, *one-to-one* for injective, and *exact correspondence* for bijective.

Here is what the definitions say about such pictures:

- "R is a function" means that every point in the domain column, A, has *at most one arrow out of it*.
- "R is total" means that *every* point in the A column has *at least one arrow out of it*. So if R is a function, being total really means every point in the A column has *exactly one arrow out of it*.
- "R is surjective" means that *every* point in the codomain column, B, has *at least one arrow into it*.
- "R is injective" means that every point in the codomain column, B, has *at most one arrow into it*.
- "R is bijective" means that *every* point in the A column has exactly one arrow out of it, and *every* point in the B column has exactly one arrow into it.

So in the diagrams above, the relation on the left is a total, surjective function (every element in the A column has exactly one arrow out, and every element in the B column has at least one arrow in), but not injective (element 3 has two arrows going into it). The relation on the right is a total, injective function (every element in the A column has exactly one arrow out, and every element in the B column has at most one arrow in), but not surjective (element 4 has no arrow going into it).

Notice that the arrows in a diagram for R precisely correspond to the pairs in the graph of R. But graph (R) does not determine by itself whether R is total or surjective; we also need to know what the domain is to determine if R is total, and we need to know the codomain to tell if it's surjective.

Example 5.5.4. The function defined by the formula $1/x^2$ is total if its domain is $\mathbb{R}^+$ but partial if its domain is some set of real numbers including 0. It is bijective if its domain and codomain are both $\mathbb{R}^+$, but neither injective nor surjective if its domain and codomain are both $\mathbb{R}$.

5.6 Cardinality

5.6.1 Mappings and Cardinality

The relational properties in Section 5.5 are useful in figuring out the relative sizes of domains and codomains.

If A is a finite set, we let $|A|$ be the number of elements in A. A finite set may have no elements (the empty set), or one element, or two elements,... or any non-negative integer number of elements.

Now suppose $R : A \to B$ is a function. Then every arrow in the diagram for R comes from exactly one element of A, so the number of arrows is at most the number of elements in A. That is, if R is a function, then

$$|A| \geq \#\text{arrows}.$$

Similarly, if R is surjective, then every element of B has an arrow into it, so there must be at least as many arrows in the diagram as the size of B. That is,

$$\#\text{arrows} \geq |B|\,.$$

Combining these inequalities implies that if R is a surjective function, then $|A| \geq |B|$. In short, if we write $A \text{ surj } B$ to mean that there is a surjective function from A to B, then we've just proved a lemma: if $A \text{ surj } B$, then $|A| \geq |B|$. The following definition and lemma lists this statement and three similar rules relating domain and codomain size to relational properties.

Definition 5.6.1. Let A, B be (not necessarily finite) sets. Then

1. $A \text{ surj } B$ iff there is a surjective *function* from A to B.
2. $A \text{ inj } B$ iff there is a total injective *relation* from A to B.
3. $A \text{ bij } B$ iff there is a bijection from A to B.
4. $A \text{ strict } B$ iff $A \text{ surj } B$, but not $B \text{ surj } A$.

Lemma 5.6.2. *[Mapping Rules] Let A and B be finite sets.*

1. *If $A \text{ surj } B$, then $|A| \geq |B|$.*
2. *If $A \text{ inj } B$, then $|A| \leq |B|$.*
3. *If $R \text{ bij } B$, then $|A| = |B|$.*
4. *If $R \text{ strict } B$, then $|A| > |B|$.*

Mapping rule 2. can be explained by the same kind of "arrow reasoning" we used for rule 1. Rules 3. and 4. are immediate consequences of these first two mapping rules.

5.6.2 The sizes of infinite sets

Mapping Rule 1 has a converse: if the size of a finite set, A, is greater than or equal to the size of another finite set, B, then it's always possible to define a surjective function from A to B. In fact, the surjection can be a total function. To see how this works, suppose for example that

$$A = \{a_0, a_1, a_2, a_3, a_4, a_5\}$$
$$B = \{b_0, b_1, b_2, b_3\}\,.$$

Then define a total function $f : A \to B$ by the rules

$$f(a_0) ::= b_0,\ f(a_1) ::= b_1,\ f(a_2) ::= b_2,\ f(a_3) = f(a_4) = f(a_5) ::= b_3.$$

In fact, if A and B are finite sets of the same size, then we could also define a bijection from A to B by this method.

In short, we have figured out if A and B are finite sets, then $|A| \geq |B|$ *if and only if* $A \text{ surj } B$, and similar iff's hold for all the other Mapping Rules:

Lemma 5.6.3. *For* finite *sets*, A, B,

$$\begin{aligned} |A| \geq |B| \quad &\textit{iff} \quad A \text{ surj } B, \\ |A| \leq |B| \quad &\textit{iff} \quad A \text{ inj } B, \\ |A| = |B| \quad &\textit{iff} \quad A \text{ bij } B, \\ |A| > |B| \quad &\textit{iff} \quad A \text{ strict } B. \end{aligned}$$

This lemma suggests a way to generalize size comparisons to infinite sets, namely, we can think of the relation surj as an "*at least as big as*" relation between sets, even if they are infinite. Similarly, the relation bij can be regarded as a "*same size*" relation between (possibly infinite) sets, and strict can be thought of as a "*strictly bigger* than" relation between sets.

Warning: We haven't, and won't, define what the "size" of an infinite is. The definition of infinite "sizes" is cumbersome and technical, and we can get by just fine without it. All we need are the "as big as" and "same size" relations, surj and bij, between sets.

But there's something else to watch out for. We've referred to surj as an "as big as" relation and bij as a "same size" relation on sets. Of course most of the "as big as" and "same size" properties of surj and bij on finite sets do carry over to infinite sets, but *some important ones don't* —as we're about to show. So you have to be careful: don't assume that surj has any particular "as big as" property on *infinite* sets until it's been proved.

Let's begin with some familiar properties of the "as big as" and "same size" relations on finite sets that do carry over exactly to infinite sets:

Lemma 5.6.4. *For any sets*, A, B, C,

1. A surj B *and* B surj C, *implies* A surj C.
2. A bij B *and* B bij C, *implies* A bij C.
3. A bij B *implies* B bij A.

Lemma 5.6.4.1 and 5.6.4.2 follow immediately from the fact that compositions of surjections are surjections, and likewise for bijections, and Lemma 5.6.4.3 follows from the fact that the inverse of a bijection is a bijection. We'll leave a proof of these facts to Problem 5.2.

Another familiar property of finite sets carries over to infinite sets, but this time it's not so obvious:

Theorem 5.6.5 (Schröder-Bernstein)**.** *For any sets* A, B, *if* A surj B *and* B surj A, *then* A bij B.

That is, the Schröder-Bernstein Theorem says that if A is at least as big as B and conversely, B is at least as big as A, then A is the same size as B. Phrased this way, you might be tempted to take this theorem for granted, but that would be a mistake. For infinite sets A and B, the Schröder-Bernstein Theorem is actually

pretty technical. Just because there is a surjective function $f : A \to B$ —which need not be a bijection —and a surjective function $g : B \to A$ —which also need not be a bijection —it's not at all clear that there must be a bijection $e : A \to B$. The idea is to construct e from parts of both f and g. We'll leave the actual construction to Problem 5.7.

5.6.3 Infinity is different

A basic property of finite sets that does *not* carry over to infinite sets is that adding something new makes a set bigger. That is, if A is a finite set and $b \notin A$, then $|A \cup \{b\}| = |A| + 1$, and so A and $A \cup \{b\}$ are not the same size. But if A is infinite, then these two sets *are* the same size!

Lemma 5.6.6. *Let A be a set and $b \notin A$. Then A is infinite iff A bij $A \cup \{b\}$.*

Proof. Since A is *not* the same size as $A \cup \{b\}$ when A is finite, we only have to show that $A \cup \{b\}$ *is* the same size as A when A is infinite.

That is, we have to find a bijection between $A \cup \{b\}$ and A when A is infinite. Here's how: since A is infinite, it certainly has at least one element; call it a_0. But since A is infinite, it has at least two elements, and one of them must not be equal to a_0; call this new element a_1. But since A is infinite, it has at least three elements, one of which must not equal a_0 or a_1; call this new element a_2. Continuing in the way, we conclude that there is an infinite sequence $a_0, a_1, a_2, \ldots, a_n, \ldots$ of different elements of A. Now it's easy to define a bijection $e : A \cup \{b\} \to A$:

$$\begin{aligned} e(b) &::= a_0, \\ e(a_n) &::= a_{n+1} && \text{for } n \in \mathbb{N}, \\ e(a) &::= a && \text{for } a \in A - \{b, a_0, a_1, \ldots\}. \end{aligned}$$

■

A set, C, is *countable* iff its elements can be listed in order, that is, the distinct elements is A are precisely

$$c_0, c_1, \ldots, c_n, \ldots.$$

This means that if we defined a function, f, on the nonnegative integers by the rule that $f(i) ::= c_i$, then f would be a bijection from $\mathbb{N}$ to C. More formally,

Definition 5.6.7. A set, C, is *countably infinite* iff $\mathbb{N}$ bij C. A set is *countable* iff it is finite or countably infinite.

A small modification[6] of the proof of Lemma 5.6.6 shows that countably infinite sets are the "smallest" infinite sets, namely, if A is a countably infinite set, then A surj $\mathbb{N}$.

Since adding one new element to an infinite set doesn't change its size, it's obvious that neither will adding any *finite* number of elements. It's a common

[6]See Problem 5.3

mistake to think that this proves that you can throw in countably infinitely many new elements. But just because it's ok to do something any finite number of times doesn't make it OK to do an infinite number of times. For example, starting from 3, you can add 1 any finite number of times and the result will be some integer greater than or equal to 3. But if you add add 1 a countably infinite number of times, you don't get an integer at all.

It turns out you really can add a countably infinite number of new elements to a countable set and still wind up with just a countably infinite set, but another argument is needed to prove this:

Lemma 5.6.8. *If A and B are countable sets, then so is $A \cup B$.*

Proof. Suppose the list of distinct elements of A is $a_0, a_1, \ldots$ and the list of B is $b_0, b_1, \ldots$. Then a list of all the elements in $A \cup B$ is just

$$a_0, b_0, a_1, b_1, \ldots a_n, b_n, \ldots. \tag{5.4}$$

Of course this list will contain duplicates if A and B have elements in common, but then deleting all but the first occurrences of each element in list (5.4) leaves a list of all the distinct elements of A and B. ■

5.6.4 Power sets are strictly bigger

It turns out that the ideas behind Russell's Paradox, which caused so much trouble for the early efforts to formulate Set Theory, also lead to a correct and astonishing fact discovered by Georg Cantor in the late nineteenth century: infinite sets are *not all the same size*.

In particular,

Theorem 5.6.9. *For any set, A, the power set, $\mathcal{P}(A)$, is strictly bigger than A.*

Proof. First of all, $\mathcal{P}(A)$ is as big as A: for example, the partial function $f : \mathcal{P}(A) \to A$, where $f(\{a\}) ::= a$ for $a \in A$ and f is only defined on one-element sets, is a surjection.

To show that $\mathcal{P}(A)$ is strictly bigger than A, we have to show that if g is a function from A to $\mathcal{P}(A)$, then g is not a surjection. So, mimicking Russell's Paradox, define

$$A_g ::= \{a \in A \mid a \notin g(a)\}.$$

Now A_g is a well-defined subset of A, which means it is a member of $\mathcal{P}(A)$. But A_g can't be in the range of g, because if it were, we would have

$$A_g = g(a_0)$$

for some $a_0 \in A$, so by definition of A_g,

$$a \in g(a_0) \quad \text{iff} \quad a \in A_g \quad \text{iff} \quad a \notin g(a)$$

for all $a \in A$. Now letting $a = a_0$ yields the contradiction

$$a_0 \in g(a_0) \quad \text{iff} \quad a_0 \notin g(a_0).$$

So g is not a surjection, because there is an element in the power set of A, namely the set A_g, that is not in the range of g. ■

Larger Infinities

There are lots of different sizes of infinite sets. For example, starting with the infinite set, $\mathbb{N}$, of nonnegative integers, we can build the infinite sequence of sets

$$\mathbb{N},\ \mathcal{P}(\mathbb{N}),\ \mathcal{P}(\mathcal{P}(\mathbb{N})),\ \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{N}))),\ \ldots.$$

By Theorem 5.6.9, each of these sets is strictly bigger than all the preceding ones. But that's not all: the union of all the sets in the sequence is strictly bigger than each set in the sequence (see Problem 5.8). In this way you can keep going, building still bigger infinities.

So there is an endless variety of different size infinities.

5.7 Infinities in Computer Science

We've run into a lot of computer science students who wonder why they should care about infinite sets. They point out that any data set in a computer memory is limited by the size of memory, and there is a finite limit on the possible size of computer memory for the simple reason that the universe is (or at least appears to be) finite.

The problem with this argument is that universe-size bounds on data items are so big and uncertain (the universe seems to be getting bigger all the time), that it's simply not helpful to make use of such bounds. For example, by this argument the physical sciences shouldn't assume that measurements might yield arbitrary real numbers, because there can only be a finite number of finite measurements in a universe with a finite lifetime. What do you think scientific theories would look like without using the infinite set of real numbers?

Similarly, in computer science it simply isn't plausible that writing a program to add nonnegative integers with up to as many digits as, say, the stars in the sky (billions of galaxies each with billions of stars), would be any different than writing a program that would add any two integers no matter how many digits they had.

That's why basic programming data types like integers or strings, for example, can be defined without imposing any bound on the sizes of data items. Each datum of type `string` has only a finite number of letters, but there are an infinite number of data items of type `string`. When we then consider string procedures of type `string-->string`, not only are there an infinite number of such procedures, but each procedure generally behaves differently on different inputs, so that a single `string-->string` procedure may embody an infinite number of behaviors. In

short, an educated computer scientist can't get around having to cope with infinite sets.

On the other hand, the more exotic theory of different size infinities and continuum hypotheses rarely comes up in mainstream mathematics, and it hardly comes up at all in computer science, where the focus is mainly on finite sets, and occasionallly on countable sets. In practice, only logicians and set theorists have to worry about collections that are too big to be sets. In fact, at the end of the 19th century, the general mathematical community doubted the relevance of what they called "Cantor's paradise" of unfamiliar sets of arbitrary infinite size. So if the romance of really big infinities doesn't appeal to you, be assured that not knowing about them won't lower your professional abilities as a computer scientist.

Yet the idea behind Russell's paradox and Cantor's proof embodies the simplest form of what is known as a "diagonal argument." Diagonal arguments are used to prove many fundamental results about the limitations of computation, such as the undecidability of the Halting Problem for programs (see Problem 5.9) and the inherent, unavoidable, inefficiency (exponential time or worse) of procedures for other computational problems. So computer scientists do need to study diagonal arguments in order to understand the logical limits of computation.

5.7.1 Problems

Class Problems

Problem 5.2.
Define a *surjection relation*, surj, on sets by the rule

Definition. $A \text{ surj } B$ iff there is a surjective **function** from A to B.

Define the *injection relation*, inj, on sets by the rule

Definition. $A \text{ inj } B$ iff there is a total injective *relation* from A to B.

(a) Prove that if $A \text{ surj } B$ and $B \text{ surj } C$, then $A \text{ surj } C$.

(b) Explain why $A \text{ surj } B$ iff $B \text{ inj } A$.

(c) Conclude from (a) and (b) that if $A \text{ inj } B$ and $B \text{ inj } C$, then $A \text{ inj } C$.

Problem 5.3. (a) Several students felt the proof of Lemma 5.6.6 was worrisome, if not circular. What do you think?

(b) Use the proof of Lemma 5.6.6 to show that if A is an infinite set, then there is surjective function from A to $\mathbb{N}$, that is, every infinite set is "as big as" the set of nonnegative integers.

Problem 5.4.
Let $R : A \to B$ be a binary relation. Use an arrow counting argument to prove the following generalization of the Mapping Rule:

Lemma. *If R is a function, and $X \subseteq A$, then*

$$|X| \geq |XR|.$$

Problem 5.5.
Let $A = \{a_0, a_1, \ldots, a_{n-1}\}$ be a set of size n, and $B = \{b_0, b_1, \ldots, b_{m-1}\}$ a set of size m. Prove that $|A \times B| = mn$ by defining a simple bijection from $A \times B$ to the nonnegative integers from 0 to $mn - 1$.

Problem 5.6.
The rational numbers fill in all the spaces between the integers, so a first thought is that there must be more of them than the integers, but it's not true. In this problem you'll show that there are the same number of nonnegative rational as nonnegative integers. In short, the nonnegative rationals are countable.

(a) Describe a bijection between all the integers, $\mathbb{Z}$, and the nonnegative integers, $\mathbb{N}$.

(b) Define a bijection between the nonnegative integers and the set, $\mathbb{N} \times \mathbb{N}$, of all the ordered pairs of nonnegative integers:

$$\begin{array}{l}(0,0),(0,1),(0,2),(0,3),(0,4),\ldots\\(1,0),(1,1),(1,2),(1,3),(1,4),\ldots\\(2,0),(2,1),(2,2),(2,3),(2,4),\ldots\\(3.0),(3,1),(3,2),(3,3),(3,4),\ldots\\\quad\vdots\end{array}$$

(c) Conclude that $\mathbb{N}$ is the same size as the set, $\mathbb{Q}$, of all nonnegative rational numbers.

Problem 5.7.
Suppose sets A and B have no elements in common, and

- A is as small as B because there is a total injective function $f : A \to B$, and
- B is as small as A because there is a total injective function $g : B \to A$.

Picturing the diagrams for f and g, there is *exactly one* arrow *out* of each element —a left-to-right f-arrow if the element in A and a right-to-left g-arrow if the

element in B. This is because f and g are total functions. Also, there is *at most one* arrow *into* any element, because f and g are injections.

So starting at any element, there is a unique, and unending path of arrows going forwards. There is also a unique path of arrows going backwards, which might be unending, or might end at an element that has no arrow into it. These paths are completely separate: if two ran into each other, there would be two arrows into the element where they ran together.

This divides all the elements into separate paths of four kinds:

i. paths that are infinite in both directions,

ii. paths that are infinite going forwards starting from some element of A.

iii. paths that are infinite going forwards starting from some element of B.

iv. paths that are unending but finite.

(a) What do the paths of the last type (iv) look like?

(b) Show that for each type of path, either

- the f-arrows define a bijection between the A and B elements on the path, or
- the g-arrows define a bijection between B and A elements on the path, or
- both sets of arrows define bijections.

For which kinds of paths do both sets of arrows define bijections?

(c) Explain how to piece these bijections together to prove that A and B are the same size.

Problem 5.8.

There are lots of different sizes of infinite sets. For example, starting with the infinite set, $\mathbb{N}$, of nonnegative integers, we can build the infinite sequence of sets

$$\mathbb{N},\ \mathcal{P}(\mathbb{N}),\ \mathcal{P}(\mathcal{P}(\mathbb{N})),\ \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{N}))),\ \ldots.$$

By Theorem 5.6.9 from the Notes, each of these sets is *strictly bigger*[7] than all the preceding ones. But that's not all: if we let U be the union of the sequence of sets above, then U is strictly bigger than every set in the sequence! Prove this:

Lemma. *Let $\mathcal{P}^n(\mathbb{N})$ be the nth set in the sequence, and*

$$U ::= \bigcup_{n=0}^{\infty} \mathcal{P}^n(\mathbb{N}).$$

Then

[7]Reminder: set A is *strictly bigger* than set B just means that A surj B, but NOT(B surj A).

1. U surj $\mathcal{P}^n(\mathbb{N})$ *for every* $n \in \mathbb{N}$, *but*

2. *there is no* $n \in \mathbb{N}$ *for which* $\mathcal{P}^n(\mathbb{N})$ surj U.

Now of course, we could take $U, \mathcal{P}(U), \mathcal{P}(\mathcal{P}(U)), \ldots$ and can keep on indefinitely building still bigger infinities.

Problem 5.9.
Let's refer to a programming procedure (written in your favorite programming language —C++, or Java, or Python, ...) as a *string procedure* when it is applicable to data of type `string` and only returns values of type `boolean`. When a string procedure, P, applied to a `string`, s, returns **True**, we'll say that P *recognizes* s. If $\mathcal{R}$ is the set of strings that P recognizes, we'll call P a *recognizer* for $\mathcal{R}$.

(a) Describe how a recognizer would work for the set of strings containing only lower case Roman letter —`a`, `b`, ..., `z` —such that each letter occurs twice in a row. For example, `aaccaabbzz`, is such a string, but `abb`, `00bb`, `AAbb`, and `a` are not. (Even better, actually write a recognizer procedure in your favorite programming language).

A set of `string`s is called *recognizable* if there is a recognizer procedure for it.

When you actually program a procedure, you have to type the program text into a computer system. This means that every procedure is described by some `string` of typed characters. If a `string`, s, is actually the typed description of some string procedure, let's refer to that procedure as P_s. You can think of P_s as the result of compiling s.[8]

In fact, it will be helpful to associate every string, s, with a procedure, P_s; we can do this by defining P_s to be some fixed string procedure —it doesn't matter which one —whenever s is not the typed description of an actual procedure that can be applied to `string`s. The result of this is that we have now defined a total function, f, mapping every `string`, s, to the set, $f(s)$, of `string`s recognized by P_s. That is we have a total function,

$$f : \texttt{string} \to \mathcal{P}(\texttt{string}). \tag{5.5}$$

(b) Explain why the actual range of f is the set of all recognizable sets of strings.

This is exactly the set up we need to apply the reasoning behind Russell's Paradox to define a set that is not in the range of f, that is, a set of strings, $\mathcal{N}$, that is *not* recognizable.

(c) Let

$$\mathcal{N} ::= \{s \in \texttt{string} \mid s \notin f(s)\}.$$

[8]The string, s, and the procedure, P_s, have to be distinguished to avoid a type error: you can't apply a string to string. For example, let s be the string that you wrote as your program to answer part (a). Applying s to a string argument, say `oorrmm`, should throw a type exception; what you need to do is apply the procedure P_s to `oorrmm`. This should result in a returned value **True**, since `oorrmm` consists of three pairs of lowercase roman letters

Prove that $\mathcal{N}$ is not recognizable.

Hint: Similar to Russell's paradox or the proof of Theorem 5.6.9.

(d) Discuss what the conclusion of part (c) implies about the possibility of writing "program analyzers" that take programs as inputs and analyze their behavior.

Problem 5.10.

Though it was a serious challenge for set theorists to overcome Russells' Paradox, the idea behind the paradox led to some important (and correct :-)) results in Logic and Computer Science.

To show how the idea applies, let's recall the formulas from Problem 1.13 that made assertions about binary strings. For example, one of the formulas in that problem was

$$\text{NOT}[\exists y\, \exists z. s = y1z] \qquad \textbf{(all-0s)}$$

This formula defines a property of a binary string, s, namely that s has no occurrence of a 1. In other words, s is a string of (zero or more) 0's. So we can say that this formula *describes* the set of strings of 0's.

More generally, when G is any formula that defines a string property, let ok-strings(G) be the set of all the strings that have this property. A set of binary strings that equals ok-strings(G) for some G is called a *describable* set of strings. So, for example, the set of all strings of 0's is describable because it equals ok-strings(**all-0s**).

Now let's shift gears for a moment and think about the fact that formula **all-0s** appears above. This happens because instructions for formatting the formula were generated by a computer text processor (for this text, we used the LaTeX text processing system), and then an image suitable for printing or display was constructed according to these instructions. Since everybody knows that data is stored in computer memory as binary strings, this means there must have been some binary string in computer memory —call it $t_{\textbf{all-0s}}$ —that enabled a computer to display formula **all-0s** once $t_{\textbf{all-0s}}$ was retrieved from memory.

In fact, it's not hard to find ways to represent *any* formula, G, by a corresponding binary word, t_G, that would allow a computer to reconstruct G from t_G. We needn't be concerned with how this reconstruction process works; all that matters for our purposes is that every formula, G, has a representation as binary string, t_G.

Now let

$$V ::= \{t_G \mid G \text{ defines a property of strings and } t_G \notin \text{ok-strings}(G)\}.$$

Use reasoning similar to Russell's paradox to show that V is not describable.

Homework Problems

Problem 5.11.

Let $f : A \to B$ and $g : B \to C$ be functions and $h : A \to C$ be their composition, namely, $h(a) ::= g(f(a))$ for all $a \in A$.

(a) Prove that if f and g are surjections, then so is h.

(b) Prove that if f and g are bijections, then so is h.

(c) If f is a bijection, then define $f' : B \to A$ so that

$$f'(b) ::= \text{ the unique } a \in A \text{ such that } f(a) = b.$$

Prove that f' is a bijection. (The function f' is called the *inverse* of f. The notation f^{-1} is often used for the inverse of f.)

Problem 5.12.
In this problem you will prove a fact that may surprise you —or make you even more convinced that set theory is nonsense: the half-open unit interval is actually the *same size* as the nonnegative quadrant of the real plane![9] Namely, there is a bijection from $(0, 1]$ to $[0, \infty)^2$.

(a) Describe a bijection from $(0, 1]$ to $[0, \infty)$.

Hint: $1/x$ almost works.

(b) An infinite sequence of the decimal digits $\{0, 1, \ldots, 9\}$ will be called *long* if it has infinitely many occurrences of some digit other than 0. Let L be the set of all such long sequences. Describe a bijection from L to the half-open real interval $(0, 1]$.

Hint: Put a decimal point at the beginning of the sequence.

(c) Describe a surjective function from L to L^2 that involves alternating digits from two long sequences. a *Hint:* The surjection need not be total.

(d) Prove the following lemma and use it to conclude that there is a bijection from L^2 to $(0, 1]^2$.

Lemma 5.7.1. *Let A and B be nonempty sets. If there is a bijection from A to B, then there is also a bijection from $A \times A$ to $B \times B$.*

(e) Conclude from the previous parts that there is a surjection from $(0, 1]$ and $(0, 1]^2$. Then appeal to the Schröder-Bernstein Theorem to show that there is actually a bijection from $(0, 1]$ and $(0, 1]^2$.

(f) Complete the proof that there is a bijection from $(0, 1]$ to $[0, \infty)^2$.

Problem 5.13.
Let $[\mathbb{N} \to \{1, 2, 3\}]$ be the set of all sequences containing only the numbers 1, 2, and

[9] The half open unit interval, $(0, 1]$, is $\{r \in \mathbb{R} \mid 0 < r \leq 1\}$. Similarly, $[0, \infty) ::= \{r \in \mathbb{R} \mid r \geq 0\}$.

3, for example,

$$(1, 1, 1, 1...),$$
$$(2, 2, 2, 2...),$$
$$(3, 2, 1, 3...).$$

For any sequence, s, let $s[m]$ be its mth element.

Prove that $[\mathbb{N} \to \{1, 2, 3\}]$ is uncountable.

Hint: Suppose there was a list

$$\mathcal{L} = \text{sequence}_0, \text{sequence}_1, \text{sequence}_2, \ldots$$

of sequences in $[\mathbb{N} \to \{1, 2, 3\}]$ and show that there is a "diagonal" sequence diag $\in [\mathbb{N} \to \{1, 2, 3\}]$ that does not appear in the list. Namely,

$$\text{diag} ::= r(\text{sequence}_0[0]), r(\text{sequence}_1[1]), r(\text{sequence}_2[2]), \ldots,$$

where $r : \{1, 2, 3\} \to \{1, 2, 3\}$ is some function such that $r(i) \neq i$ for $i = 1, 2, 3$.

Problem 5.14.

For any sets, A, and B, let $[A \to B]$ be the set of total functions from A to B. Prove that if A is not empty and B has more than one element, then NOT$(A \text{ surj } [A \to B])$.

Hint: Suppose there is a function, σ, that maps each element $a \in A$ to a function $\sigma_a : A \to B$. Pick any two elements of B; call them 0 and 1. Then define

$$\text{diag}(a) ::= \begin{cases} 0 \text{ if } \sigma_a(a) = 1, \\ 1 \text{ otherwise.} \end{cases}$$

5.8 Glossary of Symbols

symbol	meaning
$\in$	is a member of
$\subseteq$	is a subset of
$\subset$	is a proper subset of
$\cup$	set union
$\cap$	set intersection
$\overline{A}$	complement of a set, A
$\mathcal{P}(A)$	powerset of a set, A
$\emptyset$	the empty set, $\{\}$
$\mathbb{N}$	nonnegative integers
$\mathbb{Z}$	integers
$\mathbb{Z}^+$	positive integers
$\mathbb{Z}^-$	negative integers
$\mathbb{Q}$	rational numbers
$\mathbb{R}$	real numbers
$\mathbb{C}$	complex numbers
λ	the empty string/list

Chapter 6

Partial Orders and Equivalence Relations

Partial orders are a kind of binary relation that come up a lot. The familiar $\leq$ order on numbers is a partial order, but so is the containment relation on sets and the divisibility relation on integers.

Partial orders have particular importance in computer science because they capture key concepts used, for example, in solving task scheduling problems, analyzing concurrency control, and proving program termination.

6.1 Axioms for Partial Orders

The prerequisite structure among MIT subjects provides a nice illustration of partial orders. Here is a table indicating some of the prerequisites of subjects in the the Course 6 program of Spring '07:

Direct Prerequisites	Subject
18.01	6.042
18.01	18.02
18.01	18.03
8.01	8.02
6.001	6.034
6.042	6.046
18.03, 8.02	6.002
6.001, 6.002	6.004
6.001, 6.002	6.003
6.004	6.033
6.033	6.857
6.046	6.840

Since 18.01 is a direct prerequisite for 6.042, a student must take 18.01 before 6.042. Also, 6.042 is a direct prerequisite for 6.046, so in fact, a student has to take *both* 18.01 and 6.042 before taking 6.046. So 18.01 is also really a prerequisite for 6.046, though an implicit or indirect one; we'll indicate this by writing

$$18.01 \to 6.046.$$

This prerequisite relation has a basic property known as *transitivity*: if subject a is an indirect prerequisite of subject b, and b is an indirect prerequisite of subject c, then a is also an indirect prerequisite of c.

In this table, a longest sequence of prerequisites is

$$18.01 \to 18.03 \to 6.002 \to 6.004 \to 6.033 \to 6.857$$

so a student would need at least six terms to work through this sequence of subjects. But it would take a lot longer to complete a Course 6 major if the direct prerequisites led to a situation[1] where two subjects turned out to be prerequisites of *each other*! So another crucial property of the prerequisite relation is that if $a \to b$, then it is not the case that $b \to a$. This property is called *asymmetry*.

Another basic example of a partial order is the subset relation, $\subseteq$, on sets. In fact, we'll see that every partial order can be represented by the subset relation.

Definition 6.1.1. A binary relation, R, on a set A is:

- *transitive* iff $[a\ R\ b \text{ and } b\ R\ c]$ IMPLIES $a\ R\ c$ for every $a, b, c \in A$,
- *asymmetric* iff $a\ R\ b$ IMPLIES NOT$(b\ R\ a)$ for all $a, b \in A$,
- a *strict partial order* iff it is transitive and asymmetric.

So the prerequisite relation, $\to$, on subjects in the MIT catalogue is a strict partial order. More familiar examples of strict partial orders are the relation, $<$, on real numbers, and the proper subset relation, $\subset$, on sets.

The subset relation, $\subseteq$, on sets and $\leq$ relation on numbers are examples of *reflexive* relations in which each element is related to itself. Reflexive partial orders are called *weak* partial orders. Since asymmetry is incompatible with reflexivity, the asymmetry property in weak partial orders is relaxed so it applies only to two different elements. This relaxation of the asymmetry is called antisymmetry:

Definition 6.1.2. A binary relation, R, on a set A, is

- *reflexive* iff $a\ R\ a$ for all $a \in A$,
- *antisymmetric* iff $a\ R\ b$ IMPLIES NOT$(b\ R\ a)$ for all $a \neq b \in A$,
- a *weak partial order* iff it is transitive, reflexive and antisymmetric.

[1] MIT's Committee on Curricula has the responsibility of watching out for such bugs that might creep into departmental requirements.

Some authors define partial orders to be what we call weak partial orders, but we'll use the phrase "partial order" to mean either a weak or strict one.

For weak partial orders in general, we often write an ordering-style symbol like $\preceq$ or $\sqsubseteq$ instead of a letter symbol like R. (General relations are usually denoted by a letter like R instead of a cryptic squiggly symbol, so $\preceq$ is kind of like the musical performer/composer Prince, who redefined the spelling of his name to be his own squiggly symbol. A few years ago he gave up and went back to the spelling "Prince.") Likewise, we generally use $\prec$ or $\sqsubset$ to indicate a strict partial order.

Two more examples of partial orders are worth mentioning:

Example 6.1.3. Let A be some family of sets and define $a \; R \; b$ iff $a \supset b$. Then R is a strict partial order.

For integers, m, n we write $m \mid n$ to mean that m *divides* n, namely, there is an integer, k, such that $n = km$.

Example 6.1.4. The divides relation is a weak partial order on the nonnegative integers.

6.2 Representing Partial Orders by Set Containment

Axioms can be a great way to abstract and reason about important properties of objects, but it helps to have a clear picture of the things that satisfy the axioms. We'll show that every partial order can be pictured as a collection of sets related by containment. That is, every partial order has the "same shape" as such a collection. The technical word for "same shape" is "isomorphic."

Definition 6.2.1. A binary relation, R, on a set, A, is *isomorphic* to a relation, S, on a set D iff there is a relation-preserving bijection from A to D. That is, there is bijection $f : A \to D$, such that for all $a, a' \in A$,

$$a \; R \; a' \quad \text{iff} \quad f(a) \; S \; f(a').$$

Theorem 6.2.2. *Every weak partial order, $\preceq$, is isomorphic to the subset relation, on a collection of sets.*

To picture a partial order, $\preceq$, on a set, A, as a collection of sets, we simply represent each element A by the set of elements that are $\preceq$ to that element, that is,

$$a \longleftrightarrow \{b \in A \mid b \preceq a\}.$$

For example, if $\preceq$ is the divisibility relation on the set of integers, $\{1, 3, 4, 6, 8, 12\}$, then we represent each of these integers by the set of integers in A that divides it.

So

$$\begin{aligned} 1 &\longleftrightarrow \{1\} \\ 3 &\longleftrightarrow \{1,3\} \\ 4 &\longleftrightarrow \{1,4\} \\ 6 &\longleftrightarrow \{1,3,6\} \\ 8 &\longleftrightarrow \{1,4,8\} \\ 12 &\longleftrightarrow \{1,3,4,6,12\} \end{aligned}$$

So, the fact that $3 \mid 12$ corresponds to the fact that $\{1,3\} \subseteq \{1,3,4,6,12\}$.

In this way we have completely captured the weak partial order $\preceq$ by the subset relation on the corresponding sets. Formally, we have

Lemma 6.2.3. *Let $\preceq$ be a weak partial order on a set, A. Then $\preceq$ is isomorphic to the subset relation on the collection of inverse images of elements $a \in A$ under the $\preceq$ relation.*

We leave the proof to Problem 6.3. Essentially the same construction shows that strict partial orders can be represented by set under the proper subset relation, $\subset$.

6.2.1 Problems

Class Problems

Problem 6.1.

Direct Prerequisites	Subject
18.01	6.042
18.01	18.02
18.01	18.03
8.01	8.02
8.01	6.01
6.042	6.046
18.02, 18.03, 8.02, 6.01	6.02
6.01, 6.042	6.006
6.01	6.034
6.02	6.004

(a) For the above table of MIT subject prerequisites, draw a diagram showing the subject numbers with a line going down to every subject from each of its (direct) prerequisites.

(b) Give an example of a collection of sets partially ordered by the proper subset relation, $\subset$, that is isomorphic to ("same shape as") the prerequisite relation among MIT subjects from part (a).

(c) Explain why the empty relation is a strict partial order and describe a collection of sets partially ordered by the proper subset relation that is isomorphic to the empty relation on five elements —that is, the relation under which none of the five elements is related to anything.

(d) Describe a *simple* collection of sets partially ordered by the proper subset relation that is isomorphic to the "properly contains" relation, $\supset$, on $\mathcal{P}\{1,2,3,4\}$.

Problem 6.2.
Consider the proper subset partial order, $\subset$, on the power set $\mathcal{P}\{1,2,\ldots,6\}$.

(a) What is the size of a maximal chain in this partial order? Describe one.

(b) Describe the largest antichain you can find in this partial order.

(c) What are the maximal and minimal elements? Are they maximum and minimum?

(d) Answer the previous part for the $\subset$ partial order on the set $\mathcal{P}\{1,2,\ldots,6\} - \emptyset$.

Homework Problems

Problem 6.3.
This problem asks for a proof of Lemma 6.2.3 showing that every weak partial order can be represented by (is isomorphic to) a collection of sets partially ordered under set inclusion ($\subseteq$). Namely,

Lemma. *Let $\preceq$ be a weak partial order on a set, A. For any element $a \in A$, let*

$$L(a) ::= \{b \in A \mid b \preceq a\},$$
$$\mathcal{L} ::= \{L(a) \mid a \in A\}.$$

Then the function $L : A \to \mathcal{L}$ is an isomorphism from the $\preceq$ relation on A, to the subset relation on $\mathcal{L}$.

(a) Prove that the function $\mathrm{L} : A \to \mathcal{L}$ is a bijection.

(b) Complete the proof by showing that

$$a \preceq b \quad \text{iff} \quad \mathrm{L}(a) \subseteq \mathrm{L}(b) \tag{6.1}$$

for all $a, b \in A$.

6.3 Total Orders

The familiar order relations on numbers have an important additional property: given two different numbers, one will be bigger than the other. Partial orders with

this property are said to be *total*[2] *orders*.

Definition 6.3.1. Let R be a binary relation on a set, A, and let a, b be elements of A. Then a and b are *comparable* with respect to R iff $[a \ R \ b \ \text{ OR } \ b \ R \ a]$. A partial order for which every two different elements are comparable is called a *total order*.

So $<$ and $\leq$ are total orders on $\mathbb{R}$. On the other hand, the subset relation is *not* total, since, for example, any two different finite sets of the same size will be incomparable under $\subseteq$. The prerequisite relation on Course 6 required subjects is also not total because, for example, neither 8.01 nor 6.001 is a prerequisite of the other.

6.3.1 Problems

Practice Problems

Problem 6.4.
For each of the binary relations below, state whether it is a strict partial order, a weak partial order, or neither. If it is not a partial order, indicate which of the axioms for partial order it violates. If it is a partial order, state whether it is a total order and identify its maximal and minimal elements, if any.

(a) The superset relation, $\supseteq$ on the power set $\mathcal{P}\{1, 2, 3, 4, 5\}$.

(b) The relation between any two nonegative integers, a, b that the remainder of a divided by 8 equals the remainder of b divided by 8.

(c) The relation between propositional formulas, G, H, that G IMPLIES H is valid.

(d) The relation 'beats' on Rock, Paper and Scissor (for those who don't know the game Rock, Paper, Scissors, Rock beats Scissors, Scissors beats Paper and Paper beats Rock).

(e) The empty relation on the set of real numbers.

(f) The identity relation on the set of integers.

(g) The divisibility relation on the integers, $\mathbb{Z}$.

Class Problems

Problem 6.5. (a) Verify that the divisibility relation on the set of nonnegative integers is a weak partial order.

(b) What about the divisibility relation on the set of integers?

[2] "Total" is an overloaded term when talking about partial orders: being a total order is a much stronger condition than being a partial order that is a total relation. For example, any weak partial order such as $\subseteq$ is a total relation.

Problem 6.6.
Consider the nonnegative numbers partially ordered by divisibility.

(a) Show that this partial order has a unique minimal element.

(b) Show that this partial order has a unique maximal element.

(c) Prove that this partial order has an infinite chain.

(d) An *antichain* in a partial order is a set of elements such that any two elements in the set are incomparable. Prove that this partial order has an infinite antichain. *Hint:* The primes.

(e) What are the minimal elements of divisibility on the integers greater than 1? What are the maximal elements?

Problem 6.7.
How many binary relations are there on the set $\{0, 1\}$?

How many are there that are transitive?, ... asymmetric?, ... reflexive?, ... irreflexive?, ... strict partial orders?, ... weak partial orders?

Hint: There are easier ways to find these numbers than listing all the relations and checking which properties each one has.

Problem 6.8.
A binary relation, R, on a set, A, is *irreflexive* iff $\text{NOT}(a\ R\ a)$ for all $a \in A$. Prove that if a binary relation on a set is transitive and irreflexive, then it is strict partial order.

Problem 6.9.
Prove that if R is a partial order, then so is R^{-1}

Homework Problems

Problem 6.10.
Let R and S be binary relations on the same set, A.

Definition 6.3.2. The *composition*, $S \circ R$, of R and S is the binary relation on A defined by the rule:[3]

$$a\ (S \circ R)\ c \quad \text{iff} \quad \exists b\, [a\ R\ b\ \text{ AND }\ b\ S\ c].$$

[3]Note the reversal in the order of R and S. This is so that relational composition generalizes function composition, Composing the functions f and g means that f is applied first, and then g is applied to the result. That is, the value of the composition of f and g applied to an argument, x, is $g(f(x))$. To reflect this, the notation $g \circ f$ is commonly used for the composition of f and g. Some texts do define $g \circ f$ the other way around.

Suppose both R and S are transitive. Which of the following new relations must also be transitive? For each part, justify your answer with a brief argument if the new relation is transitive and a counterexample if it is not.

(a) R^{-1}

(b) $R \cap S$

(c) $R \circ R$

(d) $R \circ S$

Exam Problems

Problem 6.11.

(a) For each row in the following table, indicate whether the binary relation, R, on the set, A, is a weak partial order or a total order by filling in the appropriate entries with either Y = YES or N = NO. In addition, list the minimal and maximal elements for each relation.

A	a R b	weak partial order	total order	minimal(s)	maximal(s)
$\mathbb{R} - \mathbb{R}^+$	$a \mid b$				
$\mathcal{P}(\{1,2,3\})$	$a \subseteq b$				
$\mathbb{N} \cup \{i\}$	$a > b$				

(b) What is the longest *chain* on the subset relation, $\subseteq$, on $P(\{1,2,3\})$? (If there is more than one, provide ONE of them.)

(c) What is the longest *antichain* on the subset relation, $\subseteq$, on $P(\{1,2,3\})$? (If there is more than one, provide *one* of them.)

6.4 Product Orders

Taking the product of two relations is a useful way to construct new relations from old ones.

Definition 6.4.1. The product, $R_1 \times R_2$, of relations R_1 and R_2 is defined to be the relation with

$$\begin{array}{rcl} \text{domain}\,(R_1 \times R_2) & ::= & \text{domain}\,(R_1) \times \text{domain}\,(R_2)\,, \\ \text{codomain}\,(R_1 \times R_2) & ::= & \text{codomain}\,(R_1) \times \text{codomain}\,(R_2)\,, \\ (a_1, a_2)\,(R_1 \times R_2)\,(b_1, b_2) & \text{iff} & [a_1\, R_1\, b_1 \text{ and } a_2\, R_2\, b_2]. \end{array}$$

Example 6.4.2. Define a relation, Y, on age-height pairs of being younger *and* shorter. This is the relation on the set of pairs (y, h) where y is a nonnegative integer ≤ 2400 which we interpret as an age in months, and h is a nonnegative integer ≤ 120 describing height in inches. We define Y by the rule

$$(y_1, h_1)\, Y\, (y_2, h_2) \quad \text{iff} \quad y_1 \leq y_2 \ \text{ AND } \ h_1 \leq h_2.$$

That is, Y is the product of the $\leq$-relation on ages and the $\leq$-relation on heights.

It follows directly from the definitions that products preserve the properties of transitivity, reflexivity, irreflexivity, and antisymmetry, as shown in Problem 6.12. That is, if R_1 and R_2 both have one of these properties, then so does $R_1 \times R_2$. This implies that if R_1 and R_2 are both partial orders, then so is $R_1 \times R_2$.

On the other hand, the property of being a total order is not preserved. For example, the age-height relation Y is the product of two total orders, but it is not total: the age 240 months, height 68 inches pair, (240,68), and the pair (228,72) are incomparable under Y.

6.4.1 Problems

Class Problems

Problem 6.12.
Let R_1, R_2 be binary relations on the same set, A. A relational property is preserved under product, if $R_1 \times R_2$ has the property whenever both R_1 and R_2 have the property.

(a) Verify that each of the following properties are preserved under product.

1. reflexivity,
2. antisymmetry,
3. transitivity.

(b) Verify that if *either* of R_1 or R_2 is irreflexive, then so is $R_1 \times R_2$.

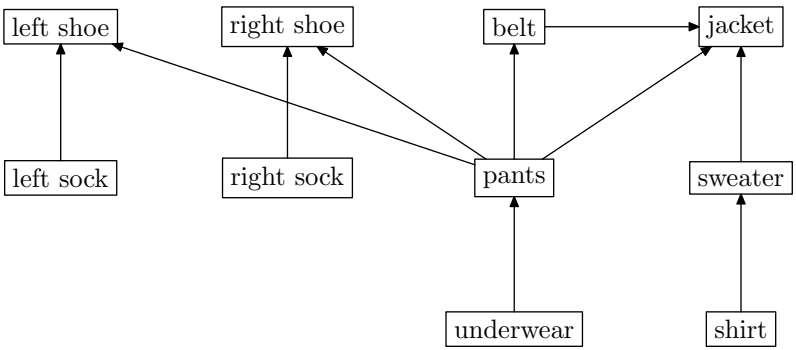
Note that it now follows immediately that if if R_1 and R_2 are partial orders and at least one of them is strict, then $R_1 \times R_2$ is a strict partial order.

6.5 Scheduling

6.5.1 Scheduling with Constraints

Scheduling problems are a common source of partial orders: there is a set, A, of tasks and a set of constraints specifying that starting a certain task depends on other tasks being completed beforehand. We can picture the constraints by drawing labelled boxes corresponding to different tasks, with an arrow from one box to another if the first box corresponds to a task that must be completed before starting the second one.

Example 6.5.1. Here is a drawing describing the order in which you could put on clothes. The tasks are the clothes to be put on, and the arrows indicate what should be put on directly before what.



When we have a partial order of tasks to be performed, it can be useful to have an order in which to perform all the tasks, one at a time, while respecting the dependency constraints. This amounts to finding a total order that is consistent with the partial order. This task of finding a total ordering that is consistent with a partial order is known as *topological sorting*.

Definition 6.5.2. A *topological sort* of a partial order, $\prec$, on a set, A, is a total ordering, $\sqsubset$, on A such that

$$a \prec b \text{ IMPLIES } a \sqsubset b.$$

For example,

$$\text{shirt} \sqsubset \text{sweater} \sqsubset \text{underwear} \sqsubset \text{leftsock} \sqsubset \text{rightsock} \sqsubset \text{pants} \sqsubset \text{leftshoe} \sqsubset \text{rightshoe} \sqsubset \text{belt} \sqsubset \text{jacket},$$

is one topological sort of the partial order of dressing tasks given by Example 6.5.1; there are several other possible sorts as well.

Topological sorts for partial orders on finite sets are easy to construct by starting from *minimal* elements:

Definition 6.5.3. Let $\preceq$ be a partial order on a set, A. An element $a_0 \in A$ is *minimum* iff it is $\preceq$ every other element of A, that is, $a_0 \preceq b$ for all $b \neq a_0$.

The element a_0 is *minimal* iff no other element is $\preceq a_0$, that is, $\text{NOT}(b \preceq a_0)$ for all $b \neq a_0$.

There are corresponding definitions for *maximum* and *maximal*. Alternatively, a maximum(al) element for a relation, R, could be defined to be as a minimum(al) element for R^{-1}.

In a total order, minimum and minimal elements are the same thing. But a partial order may have no minimum element but lots of minimal elements. There are four minimal elements in the clothes example: leftsock, rightsock, underwear, and shirt.

To construct a total ordering for getting dressed, we pick one of these minimal elements, say shirt. Next we pick a minimal element among the remaining ones. For example, once we have removed shirt, sweater becomes minimal. We continue in this way removing successive minimal elements until all elements have been picked. The sequence of elements in the order they were picked will be a topological sort. This is how the topological sort above for getting dressed was constructed.

So our construction shows:

Theorem 6.5.4. *Every partial order on a finite set has a topological sort.*

There are many other ways of constructing topological sorts. For example, instead of starting "from the bottom" with minimal elements, we could build a total starting *anywhere* and simply keep putting additional elements into the total order wherever they will fit. In fact, the domain of the partial order need not even be finite: we won't prove it, but *all* partial orders, even infinite ones, have topological sorts.

6.5.2 Parallel Task Scheduling

For a partial order of task dependencies, topological sorting provides a way to execute tasks one after another while respecting the dependencies. But what if we have the ability to execute more than one task at the same time? For example, say tasks are programs, the partial order indicates data dependence, and we have a parallel machine with lots of processors instead of a sequential machine with only one. How should we schedule the tasks? Our goal should be to minimize the total *time* to complete all the tasks. For simplicity, let's say all the tasks take one unit of time, and all the processors are identical.

So, given a finite partially ordered set of tasks, how long does it take to do them all, in an optimal parallel schedule? We can also use partial order concepts to analyze this problem.

In the clothes example, we could do all the minimal elements first (leftsock, rightsock, underwear, shirt), remove them and repeat. We'd need lots of hands, or maybe dressing servants. We can do pants and sweater next, and then leftshoe, rightshoe, and belt, and finally jacket.

In general, a *schedule* for performing tasks specifies which tasks to do at successive steps. Every task, a, has be scheduled at some step, and all the tasks that have

to be completed before task a must be scheduled for an earlier step.

Definition 6.5.5. A *parallel schedule* for a strict partial order, $\prec$, on a set, A, is a partition[4] of A into sets $A_0, A_1, \ldots$, such that for all $a, b \in A$, $k \in \mathbb{N}$,

$$[a \in A_k \text{ AND } b \prec a] \quad \text{IMPLIES} \quad b \in A_j \text{ for some } j < k.$$

The set A_k is called the set of elements *scheduled at step* k, and the *length* of the schedule is the number of sets A_k in the partition. The maximum number of elements scheduled at any step is called the *number of processors* required by the schedule.

So the schedule we chose above for clothes has four steps

$$\begin{aligned} A_0 &= \{\text{leftsock, rightsock, underwear, shirt}\}, \\ A_1 &= \{\text{pants, sweater}\}, \\ A_2 &= \{\text{leftshoe, rightshoe, belt}\}, \\ A_3 &= \{\text{jacket}\}. \end{aligned}$$

and requires four processors (to complete the first step).

Because you have to put on your underwear before your pants, your pants before your belt, and your belt before your jacket, at least four steps are needed in *every* schedule for getting dressed —if we used fewer than four steps, two of these tasks would have to be scheduled at the same time. A set of tasks that must be done in sequence like this is called a *chain*.

Definition 6.5.6. A *chain* in a partial order is a set of elements such that any two different elements in the set are comparable. A chain is said to *end at* an its maximum element.

In general, the earliest step at which a task, a, can ever be scheduled must be at least as large as any chain that ends at a. A *largest* chain ending at a is called a *critical path* to a, and the size of the critical path is called the *depth* of a. So in any possible parallel schedule, it takes at least depth (a) steps to complete task a.

There is a very simple schedule that completes every task in this minimum number of steps. Just use a "greedy" strategy of performing tasks as soon as possible. Namely, schedule all the elements of depth k at step k. That's how we found the schedule for getting dressed given above.

Theorem 6.5.7. *Let $\prec$ be a strict partial order on a set, A. A minimum length schedule for $\prec$ consists of the sets $A_0, A_1, \ldots$, where*

$$A_k ::= \{a \mid \text{depth}(a) = k\}.$$

[4]Partitioning a set, A, means "cutting it up" into non-overlapping, nonempty pieces. The pieces are called the blocks of the partition. More precisely, a *partition* of A is a set $\mathcal{B}$ whose elements are nonempty subsets of A such that

- if $B, B' \in \mathcal{B}$ are different sets, then $B \cap B' = \emptyset$, and
- $\bigcup_{B \in \mathcal{B}} B = A$.

We'll leave to Problem 6.19 the proof that the sets A_k are a parallel schedule according to Definition 6.5.5.

The minimum number of steps needed to schedule a partial order, $\prec$, is called the *parallel time* required by $\prec$, and a largest possible chain in $\prec$ is called a *critical path* for $\prec$. So we can summarize the story above by this way: with an unlimited number of processors, the minimum parallel time to complete all tasks is simply the size of a critical path:

Corollary 6.5.8. *Parallel time = length of critical path.*

6.6 Dilworth's Lemma

Definition 6.6.1. An *antichain* in a partial order is a set of elements such that any two elements in the set are incomparable.

For example, it's easy to verify that each set A_k is an antichain (see Problem 6.19). So our conclusions about scheduling also tell us something about antichains.

Corollary 6.6.2. *If the largest chain in a partial order on a set, A, is of size t, then A can be partitioned into t antichains.*

Proof. Let the antichains be the sets $A_2, A_2, \ldots, A_t$. ■

Corollary 6.6.2 implies a famous result[5] about partially ordered sets:

Lemma 6.6.3 (Dilworth)**.** *For all $t > 0$, every partially ordered set with n elements must have either a chain of size greater than t or an antichain of size at least n/t.*

Proof. Suppose the largest chain is of size $\leq t$. Then by Corollary 6.6.2, the n elements can be partitioned into at most t antichains. Let ℓ be the size of the largest antichain. Since every element belongs to exactly one antichain, and there are at most t antichains, there can't be more than ℓt elements, namely, $\ell t \geq n$. So there is an antichain with at least $\ell \geq n/t$ elements. ■

Corollary 6.6.4. *Every partially ordered set with n elements has a chain of size greater than $\sqrt{n}$ or an antichain of size at least $\sqrt{n}$.*

Proof. Set $t = \sqrt{n}$ in Lemma 6.6.3. ■

Example 6.6.5. In the dressing partially ordered set, $n = 10$.

Try $t = 3$. There is a chain of size 4.

Try $t = 4$. There is no chain of size 5, but there is an antichain of size $4 \geq 10/4$.

[5] Lemma 6.6.3 also follows from a more general result known as Dilworth's Theorem which we will not discuss.

Example 6.6.6. Suppose we have a class of 101 students. Then using the product partial order, Y, from Example 6.4.2, we can apply Dilworth's Lemma to conclude that there is a chain of 11 students who get taller as they get older, or an antichain of 11 students who get taller as they get younger, which makes for an amusing in-class demo.

6.6.1 Problems

Practice Problems

Problem 6.13.
What is the size of the longest chain that is guaranteed to exist in any partially ordered set of n elements? What about the largest antichain?

Problem 6.14.
Describe a sequence consisting of the integers from 1 to 10,000 in some order so that there is no increasing or decreasing subsequence of size 101.

Problem 6.15.
What is the smallest number of partially ordered tasks for which there can be more than one minimum time schedule? Explain.

Class Problems

Problem 6.16.
The table below lists some prerequisite information for some subjects in the MIT Computer Science program (in 2006). This defines an indirect prerequisite relation, $\prec$, that is a strict partial order on these subjects.

$$
\begin{array}{rr}
18.01 \to 6.042 & 18.01 \to 18.02 \\
18.01 \to 18.03 & 6.046 \to 6.840 \\
8.01 \to 8.02 & 6.001 \to 6.034 \\
6.042 \to 6.046 & 18.03, 8.02 \to 6.002 \\
6.001, 6.002 \to 6.003 & 6.001, 6.002 \to 6.004 \\
6.004 \to 6.033 & 6.033 \to 6.857
\end{array}
$$

(a) Explain why exactly six terms are required to finish all these subjects, if you can take as many subjects as you want per term. Using a *greedy* subject selection strategy, you should take as many subjects as possible each term. Exhibit your complete class schedule each term using a greedy strategy.

(b) In the second term of the greedy schedule, you took five subjects including 18.03. Identify a set of five subjects not including 18.03 such that it would be possi-

ble to take them in any one term (using some nongreedy schedule). Can you figure out how many such sets there are?

(c) Exhibit a schedule for taking all the courses —but only one per term.

(d) Suppose that you want to take all of the subjects, but can handle only two per term. Exactly how many terms are required to graduate? Explain why.

(e) What if you could take three subjects per term?

Problem 6.17.
A pair of Math for Computer Science Teaching Assistants, Jay and Oscar, have decided to devote some of their spare time this term to establishing dominion over the entire galaxy. Recognizing this as an ambitious project, they worked out the following table of tasks on the back of Oscar's copy of the lecture notes.

1. **Devise a logo** and cool imperial theme music - 8 days.
2. **Build a fleet** of Hyperwarp Stardestroyers out of eating paraphernalia swiped from Lobdell - 18 days.
3. **Seize control** of the United Nations - 9 days, after task #1.
4. **Get shots** for Jay's cat, Tailspin - 11 days, after task #1.
5. **Open a Starbucks chain** for the army to get their caffeine - 10 days, after task #3.
6. **Train an army** of elite interstellar warriors by dragging people to see *The Phantom Menace* dozens of times - 4 days, after tasks #3, #4, and #5.
7. **Launch the fleet** of Stardestroyers, crush all sentient alien species, and establish a Galactic Empire - 6 days, after tasks #2 and #6.
8. **Defeat Microsoft** - 8 days, after tasks #2 and #6.

We picture this information in Figure 6.1 below by drawing a point for each task, and labelling it with the name and weight of the task. An edge between two points indicates that the task for the higher point must be completed before beginning the task for the lower one.

(a) Give some valid order in which the tasks might be completed.

Jay and Oscar want to complete all these tasks in the shortest possible time. However, they have agreed on some constraining work rules.

- Only one person can be assigned to a particular task; they can not work together on a single task.

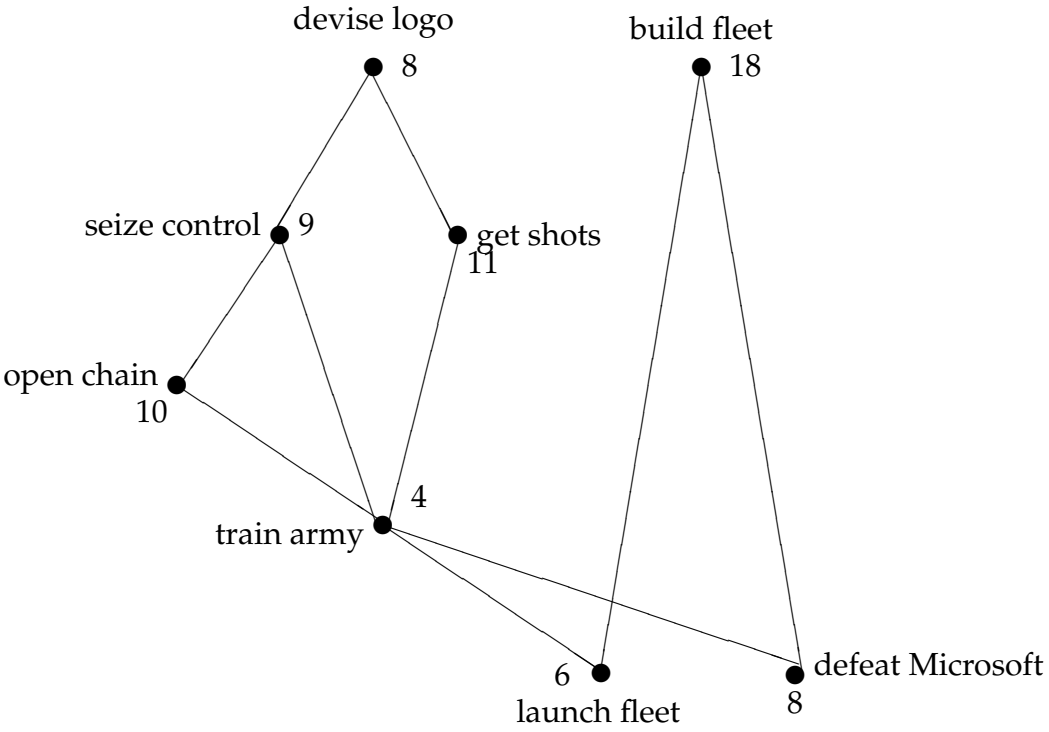


Figure 6.1: Graph representing the task precedence constraints.

- Once a person is assigned to a task, that person must work exclusively on the assignment until it is completed. So, for example, Jay cannot work on building a fleet for a few days, run to get shots for Tailspin, and then return to building the fleet.

(b) Jay and Oscar want to know how long conquering the galaxy will take. Oscar suggests dividing the total number of days of work by the number of workers, which is two. What lower bound on the time to conquer the galaxy does this give, and why might the actual time required be greater?

(c) Jay proposes a different method for determining the duration of their project. He suggests looking at the duration of the "critical path", the most time-consuming sequence of tasks such that each depends on the one before. What lower bound does this give, and why might it also be too low?

(d) What is the minimum number of days that Jay and Oscar need to conquer the galaxy? No proof is required.

Problem 6.18. **(a)** What are the maxim*al* and minim*al* elements, if any, of the power set $\mathcal{P}(\{1,\ldots,n\})$, where n is a positive integer, under the *empty relation*?

(b) What are the maxim*al* and minim*al* elements, if any, of the set, $\mathbb{N}$, of all non-negative integers under divisibility? Is there a minim*um* or maxim*um* element?

(c) What are the minimal and maximal elements, if any, of the set of integers greater than 1 under divisibility?

(d) Describe a partially ordered set that has no minimal or maximal elements.

(e) Describe a partially ordered set that has a *unique minimal* element, but no minimum element. *Hint:* It will have to be infinite.

Homework Problems

Problem 6.19.
Let $\prec$ be a partial order on a set, A, and let

$$A_k ::= \{a \mid \text{depth}\,(a) = k\}$$

where $k \in \mathbb{N}$.

(a) Prove that $A_0, A_1, \ldots$ is a parallel schedule for $\prec$ according to Definition 6.5.5.

(b) Prove that A_k is an antichain.

Problem 6.20.
Let S be a sequence of n different numbers. A *subsequence* of S is a sequence that can be obtained by deleting elements of S.

For example, if

$$S = (6, 4, 7, 9, 1, 2, 5, 3, 8)$$

Then 647 and 7253 are both subsequences of S (for readability, we have dropped the parentheses and commas in sequences, so 647 abbreviates $(6, 4, 7)$, for example).

An *increasing subsequence* of S is a subsequence of whose successive elements get larger. For example, 1238 is an increasing subsequence of S. Decreasing subsequences are defined similarly; 641 is a decreasing subsequence of S.

(a) List all the maximum length increasing subsequences of S, and all the maximum length decreasing subsequences.

Now let A be the *set* of numbers in S. (So $A = \{1, 2, 3, \ldots, 9\}$ for the example above.) There are two straightforward ways to totally order A. The first is to order its elements numerically, that is, to order A with the $<$ relation. The second is to order the elements by which comes first in S; call this order $<_S$. So for the example above, we would have

$$6 <_S 4 <_S 7 <_S 9 <_S 1 <_S 2 <_S 5 <_S 3 <_S 8$$

Next, define the partial order $\prec$ on A defined by the rule

$$a \prec a' \quad ::= \quad a < a' \text{ and } a <_S a'.$$

(It's not hard to prove that $\prec$ is strict partial order, but you may assume it.)

(b) Draw a diagram of the partial order, $\prec$, on A. What are the maximal elements,... the minimal elements?

(c) Explain the connection between increasing and decreasing subsequences of S, and chains and anti-chains under $\prec$.

(d) Prove that every sequence, S, of length n has an increasing subsequence of length greater than $\sqrt{n}$ or a decreasing subsequence of length at least $\sqrt{n}$.

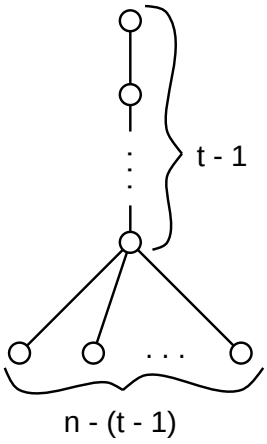
(e) (Optional, tricky) Devise an efficient procedure for finding the longest increasing and the longest decreasing subsequence in any given sequence of integers. (There is a nice one.)

Problem 6.21.

We want to schedule n partially ordered tasks.

(a) Explain why any schedule that requires only p processors must take time at least $\lceil n/p \rceil$.

(b) Let $D_{n,t}$ be the strict partial order with n elements that consists of a chain of $t - 1$ elements, with the bottom element in the chain being a prerequisite of all the remaining elements as in the following figure:



What is the minimum time schedule for $D_{n,t}$? Explain why it is unique. How many processors does it require?

(c) Write a simple formula, $M(n,t,p)$, for the minimum time of a p-processor schedule to complete $D_{n,t}$.

(d) Show that *every* partial order with n vertices and maximum chain size, t, has a p-processor schedule that runs in time $M(n,t,p)$.

Hint: Induction on t.

Chapter 7

Directed graphs

7.1 Digraphs

A *directed graph* (*digraph* for short) is formally the same as a binary relation, R, on a set, A —that is, a relation whose domain and codomain are the same set, A. But we describe digraphs as though they were diagrams, with elements of A pictured as points on the plane and arrows drawn between related points. The elements of A are referred to as the *vertices* of the digraph, and the pairs $(a, b) \in \text{graph}(R)$ are *directed edges*. Writing $a \to b$ is a more suggestive alternative for the pair (a, b). Directed edges are also called *arrows*.

For example, the divisibility relation on $\{1, 2, \ldots, 12\}$ is could be pictured by the digraph:

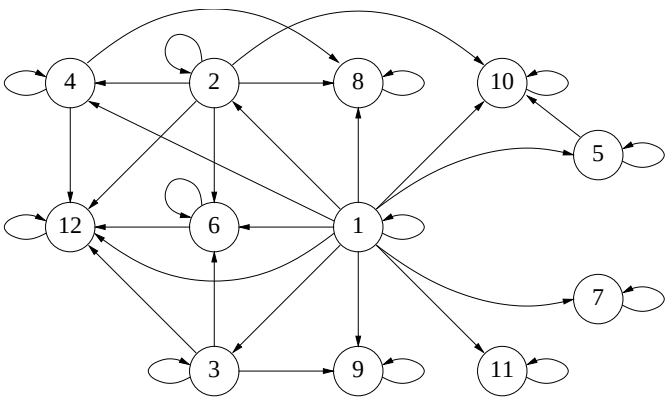


Figure 7.1: The Digraph for Divisibility on $\{1, 2, \ldots, 12\}$.

7.1.1 Paths in Digraphs

Picturing digraphs with points and arrows makes it natural to talk about following a *path* of successive edges through the graph. For example, in the digraph of Figure 7.1, a path might start at vertex 1, successively follow the edges from vertex 1 to vertex 2, from 2 to 4, from 4 to 12, and then from 12 to 12 twice (or as many times as you like). We can represent the path with the sequence of sucessive vertices it went through, in this case:

$$1, 2, 4, 12, 12, 12.$$

So a path is just a sequence of vertices, with consecutive vertices on the path connected by directed edges. Here is a formal definition:

Definition 7.1.1. A *path in a digraph* is a sequence of vertices $a_0, \ldots, a_k$ with $k \geq 0$ such that $a_i \to a_{i+1}$ is an edge of the digraph for $i = 0, 1, \ldots, k-1$. The path is said to *start* at a_0, to *end* at a_k, and the *length* of the path is defined to be k. The path is *simple* iff all the a_i's are different, that is, if $i \neq j$, then $a_i \neq a_j$.

Note that a single vertex counts as length zero path that begins and ends at itself.

It's pretty natural to talk about the edges in a path, but technically, paths only have points, not edges. So to instead, we'll say a path *traverses* an edge $a \to b$ when a and b are consecutive vertices in the path.

For any digraph, R, we can define some new relations on vertices based on paths, namely, the *path relation*, R^*, and the *positive-length path relation*, R^+:

$$a\ R^*\ b ::= \text{there is a path in } R \text{ from } a \text{ to } b,$$
$$a\ R^+\ b ::= \text{there is a positive length path in } R \text{ from } a \text{ to } b.$$

By the definition of path, both R^* and R^+ are transitive. Since edges count as length one paths, the edges of R^+ include all the edges of R. The edges of R^* in turn include all the edges of R^+ and, in addition include an edge (self-loop) from each vertex to itself. The self-loops get included in R^* because of the a length zero paths in R. So R^* is reflexive. [1]

7.2 Picturing Relational Properties

Many of the relational properties we've discussed have natural descriptions in terms of paths. For example:

Reflexivity: All vertices have self-loops (a *self-loop* at a vertex is an arrow going from the vertex back to itself).

Irreflexivity: No vertices have self-loops.

Antisymmetry: At most one (directed) edge between different vertices.

[1] In many texts, R^+ is called the *transitive closure* and R^* is called the *reflexive transitive closure* of R.

Asymmetry: No self-loops and at most one (directed) edge between different vertices.

Transitivity: Short-circuits—for any path through the graph, there is an arrow from the first vertex to the last vertex on the path.

Symmetry: A binary relation R is *symmetric* iff aRb implies bRa for all a, b in the domain of R. That is, if there is an edge from a to b, there is also one in the reverse direction.

7.3 Composition of Relations

There is a simple way to extend composition of functions to composition of relations, and this gives another way to talk about paths in digraphs.

Let $R : B \to C$ and $S : A \to B$ be relations. Then the composition of R with S is the binary relation $(R \circ S) : A \to C$ defined by the rule

$$a\ (R \circ S)\ c ::= \exists b \in B.\ (b\ R\ c) \text{ AND } (a\ S\ b).$$

This agrees with the Definition 5.4.1 of composition in the special case when R and S are functions.

Now when R is a digraph, it makes sense to compose R with itself. Then if we let R^n denote the composition of R with itself n times, it's easy to check that R^n is the length-n path relation:

$$a\ R^n\ b \quad \text{iff} \quad \text{there is a length } n \text{ path in } R \text{ from } a \text{ to } b.$$

This even works for $n = 0$, if we adopt the convention that R^0 is the identity relation Id_A on the set, A, of vertices. That is, $(a\ \mathrm{Id}_A\ b)$ iff $a = b$.

7.4 Directed Acyclic Graphs

Definition 7.4.1. A *cycle* in a digraph is defined by a path that begins and ends at the same vertex. This includes the cycle of length zero that begins and ends at the vertex. A *directed acyclic graph (DAG)* is a directed graph with no *positive* length cycles.

A *simple cycle* in a digraph is a cycle whose vertices are distinct except for the beginning and end vertices.

DAG's can be an economical way to represent partial orders. For example, in Section 6.1 the *direct prerequisite* relation between MIT subjects was used to determine the partial order of indirect prerequisites on subjects. This indirect prerequisite partial order is precisely the positive length path relation of the direct prerequisites.

Lemma 7.4.2. *If D is a DAG, then D^+ is a strict partial order.*

Proof. We know that D^+ is transitive. Also, a positive length path from a vertex to itself would be a cycle, so there are no such paths. This means D^+ is irreflexive, which implies it is a strict partial order (see problem 6.8). ■

It's easy to check that conversely, the graph of any strict partial order is a DAG.

The divisibility partial order can also be more economically represented by the path relation in a DAG. A DAG whose *path* relation is divisibility on $\{1, 2, \ldots, 12\}$ is shown in Figure 7.2; the arrowheads are omitted in the Figure, and edges are understood to point upwards.

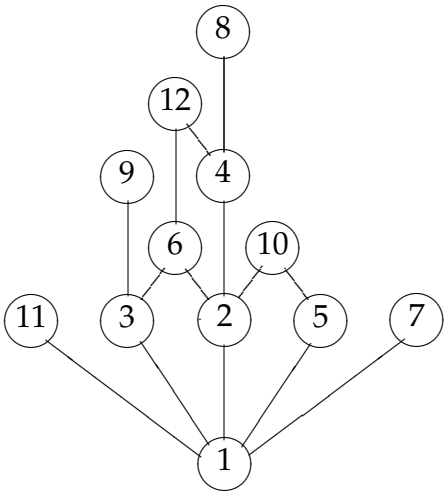


Figure 7.2: A DAG whose Path Relation is Divisibility on $\{1, 2, \ldots, 12\}$.

If we're using a DAG to represent a partial order —so all we care about is the the path relation of the DAG —we could replace the DAG with any other DAG with the same path relation. This raises the question of finding a DAG with the same path relation but the *smallest* number of edges. This DAG turns out to be unique and easy to find (see Problem 7.2).

7.4.1 Problems

Practice Problems

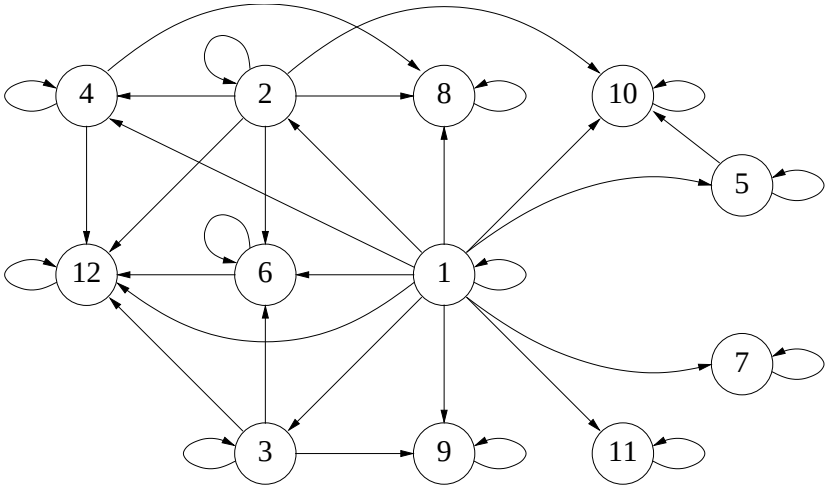
Problem 7.1.
Why is every strict partial order a DAG?

Class Problems

Problem 7.2.
If a and b are distinct nodes of a digraph, then a is said to *cover* b if there is an edge

from a to b and every path from a to b traverses this edge. If a covers b, the edge from a to b is called a *covering edge*.

(a) What are the covering edges in the following DAG?



(b) Let covering(D) be the subgraph of D consisting of only the covering edges. Suppose D is a finite DAG. Explain why covering(D) has the same positive path relation as D.

Hint: Consider *longest* paths between a pair of vertices.

(c) Show that if two *DAG*'s have the same positive path relation, then they have the same set of covering edges.

(d) Conclude that covering(D) is the *unique* DAG with the smallest number of edges among all digraphs with the same positive path relation as D.

The following examples show that the above results don't work in general for digraphs with cycles.

(e) Describe two graphs with vertices $\{1, 2\}$ which have the same set of covering edges, but not the same positive path relation (*Hint:* Self-loops.)

(f) (i) The *complete digraph* without self-loops on vertices $1, 2, 3$ has edges between every two distinct vertices. What are its covering edges?

(ii) What are the covering edges of the graph with vertices $1, 2, 3$ and edges $1 \to 2, 2 \to 3, 3 \to 1$?

(iii) What about their positive path relations?

Homework Problems

Problem 7.3.

Let R be a binary relation on a set A. Then R^n denotes the composition of R with

itself n times. Let G_R be the digraph associated with R. That is, A is the set of vertices ofG_R and R is the set of directed edges. Let $R^{(n)}$ denote the length n path relation G_R, that is,

$$a\ R^{(n)}\ b ::= \text{there is a length } n \text{ path from } a \text{ to } b \text{ in } G_R.$$

Prove that

$$R^n = R^{(n)} \tag{7.1}$$

for all $n \in \mathbb{N}$.

Problem 7.4. (a) Prove that if R is a relation on a finite set, A, then

$$a\ (R \cup I_A)^n\ b \quad \text{iff} \quad \text{there is a path in } R \text{ of length length} \leq n \text{ from } a \text{ to } b.$$

(b) Conclude that if A is a finite set, then

$$R^* = (R \cup I_A)^{|A|-1}. \tag{7.2}$$

Chapter 8

State Machines

State machines are an abstract model of step-by-step processes, and accordingly, they come up in many areas of computer science. You may already have seen them in a digital logic course, a compiler course, or a probability course.

8.1 Basic definitions

A state machine is really nothing more than a binary relation on a set, except that the elements of the set are called "states," the relation is called the *transition relation*, and a pair (p, q) in the graph of the transition relation is called a *transition*. The transition from state p to state q will be written $p \longrightarrow q$. The transition relation is also called the *state graph* of the machine. A state machine also comes equipped with a designated *start state*.

State machines used in digital logic and compilers usually have only a finite number of states, but machines that model continuing computations typically have an infinite number of states. In many applications, the states, and/or the transitions have labels indicating input or output values, costs, capacities, or probabilities, but for our purposes, unlabelled states and transitions are all we need.[1]

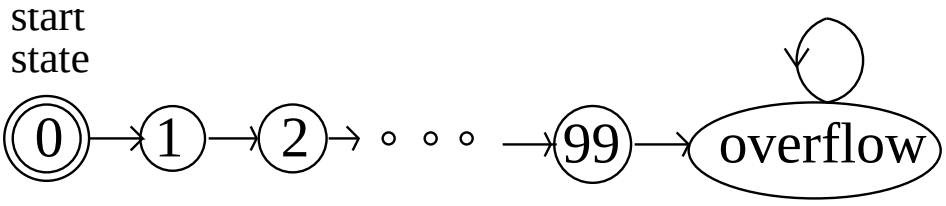


Figure 8.1: *State transitions for the 99-bounded counter.*

[1] We do name states, as in Figure 8.1, so we can talk about them, but the names aren't part of the state machine.

Example 8.1.1. A bounded counter, which counts from 0 to 99 and overflows at 100. The transitions are pictured in Figure 8.1, with start state zero. This machine isn't much use once it overflows, since it has no way to get out of its overflow state.

Example 8.1.2. An unbounded counter is similar, but has an infinite state set. This is harder to draw :-) .

Example 8.1.3. In the movie *Die Hard 3: With a Vengeance*, the characters played by Samuel L. Jackson and Bruce Willis have to disarm a bomb planted by the diabolical Simon Gruber:

> **Simon:** On the fountain, there should be 2 jugs, do you see them? A 5-gallon and a 3-gallon. Fill one of the jugs with exactly 4 gallons of water and place it on the scale and the timer will stop. You must be precise; one ounce more or less will result in detonation. If you're still alive in 5 minutes, we'll speak.
>
> **Bruce:** Wait, wait a second. I don't get it. Do you get it?
>
> **Samuel:** No.
>
> **Bruce:** Get the jugs. Obviously, we can't fill the 3-gallon jug with 4 gallons of water.
>
> **Samuel:** Obviously.
>
> **Bruce:** All right. I know, here we go. We fill the 3-gallon jug exactly to the top, right?
>
> **Samuel:** Uh-huh.
>
> **Bruce:** Okay, now we pour this 3 gallons into the 5-gallon jug, giving us exactly 3 gallons in the 5-gallon jug, right?
>
> **Samuel:** Right, then what?
>
> **Bruce:** All right. We take the 3-gallon jug and fill it a third of the way...
>
> **Samuel:** No! He said, "Be precise." Exactly 4 gallons.
>
> **Bruce:** Sh - -. Every cop within 50 miles is running his a - - off and I'm out here playing kids games in the park.
>
> **Samuel:** Hey, you want to focus on the problem at hand?

Fortunately, they find a solution in the nick of time. We'll let the reader work out how.

The *Die Hard* series is getting tired, so we propose a final *Die Hard Once and For All*. Here Simon's brother returns to avenge him, and he poses the same challenge, but with the 5 gallon jug replaced by a 9 gallon one.

We can model jug-filling scenarios with a state machine. In the scenario with a 3 and a 5 gallon water jug, the states will be pairs, (b, l) of real numbers such that

$0 \leq b \leq 5, 0 \leq l \leq 3$. We let b and l be arbitrary real numbers. (We can prove that the values of b and l will only be nonnegative integers, but we won't assume this.) The start state is $(0, 0)$, since both jugs start empty.

Since the amount of water in the jug must be known exactly, we will only consider moves in which a jug gets completely filled or completely emptied. There are several kinds of transitions:

1. Fill the little jug: $(b, l) \longrightarrow (b, 3)$ for $l < 3$.
2. Fill the big jug: $(b, l) \longrightarrow (5, l)$ for $b < 5$.
3. Empty the little jug: $(b, l) \longrightarrow (b, 0)$ for $l > 0$.
4. Empty the big jug: $(b, l) \longrightarrow (0, l)$ for $b > 0$.
5. Pour from the little jug into the big jug: for $l > 0$,

$$(b, l) \longrightarrow \begin{cases} (b + l, 0) & \text{if } b + l \leq 5, \\ (5, l - (5 - b)) & \text{otherwise.} \end{cases}$$

6. Pour from big jug into little jug: for $b > 0$,

$$(b, l) \longrightarrow \begin{cases} (0, b + l) & \text{if } b + l \leq 3, \\ (b - (3 - l), 3) & \text{otherwise.} \end{cases}$$

Note that in contrast to the 99-counter state machine, there is more than one possible transition out of states in the Die Hard machine. Machines like the 99-counter with at most one transition out of each state are called *deterministic*. The Die Hard machine is *nondeterministic* because some states have transitions to several different states.

Quick exercise: Which states of the Die Hard 3 machine have direct transitions to exactly two states?

8.2 Reachability and Preserved Invariants

The Die Hard 3 machine models every possible way of pouring water among the jugs according to the rules. Die Hard properties that we want to verify can now be expressed and proved using the state machine model. For example, Bruce's character will disarm the bomb if he can get to some state of the form $(4, l)$.

A (possibly infinite) path through the state graph beginning at the start state corresponds to a possible system behavior; such a path is called an *execution* of the state machine. A state is called *reachable* if it appears in some execution. The bomb in Die Hard 3 gets disarmed successfully because the state (4,3) is reachable.

A useful approach in analyzing state machine is to identify properties of states that are preserved by transitions.

Definition 8.2.1. A *preserved invariant* of a state machine is a predicate, P, on states, such that whenever $P(q)$ is true of a state, q, and $q \longrightarrow r$ for some state, r, then $P(r)$ holds.

> The Invariant Principle
>
> If a preserved invariant of a state machine is true for the start state, then it is true for all reachable states.

The Invariant Principle is nothing more than the Induction Principle reformulated in a convenient form for state machines. Showing that a predicate is true in the start state is the base case of the induction, and showing that a predicate is a preserved invariant is the inductive step.[2]

8.2.1 Die Hard Once and For All

Now back to Die Hard Once and For All. This time there is a 9 gallon jug instead of the 5 gallon jug. We can model this with a state machine whose states and transitions are specified the same way as for the Die Hard 3 machine, with all occurrences of "5" replaced by "9."

Now reaching any state of the form $(4, l)$ is impossible. We prove this using the Invariant Principle. Namely, we define the preserved invariant predicate, $P(b, l)$, to be that b and l are nonnegative integer multiples of 3. So P obviously holds for the state state $(0, 0)$.

To prove that P is a preserved invariant, we assume $P(b, l)$ holds for some state (b, l) and show that if $(b, l) \longrightarrow (b', l')$, then $P(b', l')$. The proof divides into cases, according to which transition rule is used. For example, suppose the transition followed from the "fill the little jug" rule. This means $(b, l) \longrightarrow (b, 3)$. But $P(b, l)$ implies that b is an integer multiple of 3, and of course 3 is an integer multiple of 3, so P still holds for the new state $(b, 3)$. Another example is when the transition rule used is "pour from big jug into little jug" for the subcase that $b + l > 3$. Then state is $(b, l) \longrightarrow (b - (3 - l), 3)$. But since b and l are integer multiples of 3, so is $b - (3 - l)$. So in this case too, P holds after the transition.

We won't bother to crank out the remaining cases, which can all be checked just as easily. Now by the Invariant Principle, we conclude that every reachable

[2]Preserved invariants are commonly just called "invariants" in the literature on program correctness, but we decided to throw in the extra adjective to avoid confusion with other definitions. For example, another subject at MIT uses "invariant" to mean "predicate true of all reachable states." Let's call this definition "invariant-2." Now invariant-2 seems like a reasonable definition, since unreachable states by definition don't matter, and all we want to show is that a desired property is invariant-2. But this confuses the *objective* of demonstrating that a property is invariant-2 with the *method* for showing that it is. After all, if we already knew that a property was invariant-2, we'd have no need for an Invariant Principle to demonstrate it.

state satisifies P. But since no state of the form $(4, l)$ satisifies P, we have proved rigorously that Bruce dies once and for all!

By the way, notice that the state (1,0), which satisfies NOT(P), has a transition to (0,0), which satisfies P. So it's wrong to assume that the complement of a preserved invariant is also a preserved invariant.

8.2.2 A Robot on a Grid

There is a robot. It walks around on a grid, and at every step it moves diagonally in a way that changes its position by one unit up or down *and* one unit left or right. The robot starts at position $(0, 0)$. Can the robot reach position $(1, 0)$?

To get some intuition, we can simulate some robot moves. For example, starting at (0,0) the robot could move northeast to (1,1), then southeast to (2,0), then southwest to (1,-1), then southwest again to (0,-2).

Let's model the problem as a state machine and then find a suitable invariant. A state will be a pair of integers corresponding to the coordinates of the robot's position. State (i, j) has transitions to four different states: $(i \pm 1, j \pm 1)$.

The problem is now to choose an appropriate preserved invariant, P, that is true for the start state $(0, 0)$ and false for $(1, 0)$. The Invariant Theorem then will imply that the robot can never reach $(1, 0)$. A direct attempt for a preserved invariant is the predicate $P(q)$ that $q \neq (1, 0)$.

Unfortunately, this is not going to work. Consider the state $(2, 1)$. Clearly $P(2, 1)$ holds because $(2, 1) \neq (1, 0)$. And of course $P(1, 0)$ does not hold. But $(2, 1) \longrightarrow (1, 0)$, so this choice of P will not yield a preserved invariant.

We need a stronger predicate. Looking at our example execution you might be able to guess a proper one, namely, that the sum of the coordinates is even! If we can prove that this is a preserved invariant, then we have proven that the robot never reaches $(1, 0)$ —because the sum $1 + 0$ of its coordinates is odd, while the sum $0 + 0$ of the coordinates of the start state is even.

Theorem 8.2.2. *The sum of the robot's coordinates is always even.*

Proof. The proof uses the Invariant Principle.

Let $P(i, j)$ be the predicate that $i + j$ is even.

First, we must show that the predicate holds for the start state $(0, 0)$. Clearly, $P(0, 0)$ is true because $0 + 0$ is even.

Next, we must show that P is a preserved invariant. That is, we must show that for each transition $(i, j) \longrightarrow (i', j')$, if $i + j$ is even, then $i' + j'$ is even. But $i' = i \pm 1$ and $j' = j \pm 1$ by definition of the transitions. Therefore, $i' + j'$ is equal to $i + j$ or $i + j \pm 2$, all of which are even. ■

Corollary 8.2.3. *The robot cannot reach* $(1, 0)$.

Robert W. Floyd

The Invariant Principle was formulated by Robert Floyd at Carnegie Tech[a] in 1967. Floyd was already famous for work on formal grammars which transformed the field of programming language parsing; that was how he got to be a professor even though he never got a Ph.D. (He was admitted to a PhD program as a teenage prodigy, but flunked out and never went back.)

In that same year, Albert R. Meyer was appointed Assistant Professor in the Carnegie Tech Computer Science Department where he first met Floyd. Floyd and Meyer were the only theoreticians in the department, and they were both delighted to talk about their shared interests. After just a few conversations, Floyd's new junior colleague decided that Floyd was the smartest person he had ever met.

Naturally, one of the first things Floyd wanted to tell Meyer about was his new, as yet unpublished, Invariant Principle. Floyd explained the result to Meyer, and Meyer wondered (privately) how someone as brilliant as Floyd could be excited by such a trivial observation. Floyd had to show Meyer a bunch of examples before Meyer understood Floyd's excitement —not at the truth of the utterly obvious Invariant Principle, but rather at the insight that such a simple theorem could be so widely and easily applied in verifying programs.

Floyd left for Stanford the following year. He won the Turing award —the "Nobel prize" of computer science— in the late 1970's, in recognition both of his work on grammars and on the foundations of program verification. He remained at Stanford from 1968 until his death in September, 2001. You can learn more about Floyd's life and work by reading the eulogy written by his closest colleague, Don Knuth.

[a] The following year, Carnegie Tech was renamed Carnegie-Mellon Univ.

8.3 Sequential algorithm examples

8.3.1 Proving Correctness

Robert Floyd, who pioneered modern approaches to program verification, distinguished two aspects of state machine or process correctness:

1. The property that the final results, if any, of the process satisfy system requirements. This is called *partial correctness*.

   You might suppose that if a result was only partially correct, then it might also be partially incorrect, but that's not what he meant. The word "partial" comes from viewing a process that might not terminate as computing a *partial function*. So partial correctness means that when there is a result, it is correct,

but the process might not always produce a result, perhaps because it gets stuck in a loop.

2. The property that the process always finishes, or is guaranteed to produce some legitimate final output. This is called *termination*.

Partial correctness can commonly be proved using the Invariant Principle. Termination can commonly be proved using the Well Ordering Principle. We'll illustrate Floyd's ideas by verifying the Euclidean Greatest Common Divisor (GCD) Algorithm.

8.3.2 The Euclidean Algorithm

The *Euclidean algorithm* is a three-thousand-year-old procedure to compute the greatest common divisor, $\gcd(a, b)$ of integers a and b. We can represent this algorithm as a state machine. A state will be a pair of integers (x, y) which we can think of as integer registers in a register program. The state transitions are defined by the rule

$$(x, y) \longrightarrow (y, \text{remainder}(x, y))$$

for $y \neq 0$. The algorithm terminates when no further transition is possible, namely when $y = 0$. The final answer is in x.

We want to prove:

1. Starting from the state with $x = a$ and $y = b > 0$, if we ever finish, then we have the right answer. That is, at termination, $x = \gcd(a, b)$. This is a *partial correctness* claim.

2. We do actually finish. This is a process *termination* claim.

Partial Correctness of GCD First let's prove that if GCD gives an answer, it is a correct answer. Specifically, let $d ::= \gcd(a, b)$. We want to prove that *if* the procedure finishes in a state (x, y), then $x = d$.

Proof. Define the state predicate

$$P(x, y) ::= \quad [\gcd(x, y) = d \text{ and } (x > 0 \text{ or } y > 0)].$$

P holds for the start state (a, b), by definition of d and the requirement that b is positive. Also, the preserved invariance of P follows immediately from

Lemma 8.3.1. *For all $m, n \in \mathbb{N}$ such that $n \neq 0$,*

$$\gcd(m, n) = \gcd(n, \text{remainder}(m, n)). \tag{8.1}$$

Lemma 8.3.1 is easy to prove: let q be the quotient and r be the remainder of m divided by n. Then $m = qn + r$ by definition. So any factor of both r and n will be a factor of m, and similarly any factor of both m and n will be a factor of r. So r, n

and m, n have the same common factors and therefore the same gcd. Now by the Invariant Principle, P holds for all reachable states.

Since the only rule for termination is that $y = 0$, it follows that if (x, y) is a terminal state, then $y = 0$. If this terminal state is reachable, then the preserved invariant holds for (x, y). This implies that $\gcd(x, 0) = d$ and that $x > 0$. We conclude that $x = \gcd(x, 0) = d$. ■

Termination of GCD Now we turn to the second property, that the procedure must terminate. To prove this, notice that y gets strictly smaller after any one transition. That's because the value of y after the transition is the remainder of x divided by y, and this remainder is smaller than y by definition. But the value of y is always a nonnegative integer, so by the Well Ordering Principle, it reaches a minimum value among all its values at reachable states. But there can't be a transition from a state where y has its minimum value, because the transition would decrease y still further. So the reachable state where y has its minimum value is a state at which no further step is possible, that is, at which the procedure terminates.

Note that this argument does not prove that the minimum value of y is zero, only that the minimum value occurs at termination. But we already noted that the only rule for termination is that $y = 0$, so it follows that the minimum value of y must indeed be zero.

We've already observed that a preserved invariant is really just an induction hypothesis. As with induction, finding the right hypothesis is usually the hard part. We repeat:

> **Given the right preserved invariant, it can be easy to verify a program even if you don't understand it.**

We expect that the Extended Euclidean Algorithm presented above illustrates this point.

8.4 Derived Variables

The preceding termination proofs involved finding a nonnegative integer-valued measure to assign to states. We might call this measure the "size" of the state. We then showed that the size of a state decreased with every state transition. By the Well Ordering Principle, the size can't decrease indefinitely, so when a minimum size state is reached, there can't be any transitions possible: the process has terminated.

More generally, the technique of assigning values to states —not necessarily nonnegative integers and not necessarily decreasing under transitions— is often useful in the analysis of algorithms. *Potential functions* play a similar role in physics. In the context of computational processes, such value assignments for states are called *derived variables*.

For example, for the Die Hard machines we could have introduced a derived variable, f : states $\to \mathbb{R}$, for the amount of water in both buckets, by setting

$f((a,b)) ::= a+b$. Similarly, in the robot problem, the position of the robot along the x-axis would be given by the derived variable x-coord, where $x\text{-coord}((i,j)) ::= i$.

We can formulate our general termination method as follows:

Definition 8.4.1. Let $\prec$ be a strict partial order on a set, A. A derived variable f : states $\to A$ is *strictly decreasing* iff

$$q \longrightarrow q' \text{ implies } f(q') \prec f(q).$$

We confirmed termination of the GCD and Extended GCD procedures by finding derived variables, y and $\mathtt{Y}$, respectively, that were nonnegative integer-valued and strictly decreasing. We can summarize this approach to proving termination as follows:

Theorem 8.4.2. *If f is a strictly decreasing $\mathbb{N}$-valued derived variable of a state machine, then the length of any execution starting at state q is at most $f(q)$.*

Of course we could prove Theorem 8.4.2 by induction on the value of $f(q)$, but think about what it says: "If you start counting down at some nonnegative integer $f(q)$, then you can't count down more than $f(q)$ times." Put this way, it's obvious.

8.4.1 Weakly Decreasing Variables

In addition being strictly decreasing, it will be useful to have derived variables with some other, related properties.

Definition 8.4.3. Let $\preceq$ be a weak partial order on a set, A. A derived variable $f : Q \to A$ is *weakly decreasing* iff

$$q \longrightarrow q' \text{ implies } f(q') \preceq f(q).$$

Strictly increasing and *weakly increasing* derived variables are defined similarly.[3]

8.4.2 Problems

Homework Problems

Problem 8.1.
You are given two buckets, A and B, a water hose, a receptacle, and a drain. The buckets and receptacle are initially empty. The buckets are labeled with their respectively capacities, positive integers a and b. The receptacle can be used to store an unlimited amount of water, but has no measurement markings. Excess water can be dumped into the drain. Among the possible moves are:

1. fill a bucket from the hose,

[3] Weakly increasing variables are often also called *nondecreasing*. We will avoid this terminology to prevent confusion between nondecreasing variables and variables with the much weaker property of *not* being a decreasing variable.

2. pour from the receptacle to a bucket until the bucket is full or the receptacle is empty, whichever happens first,

3. empty a bucket to the drain,

4. empty a bucket to the receptacle,

5. pour from A to B until either A is empty or B is full, whichever happens first,

6. pour from B to A until either B is empty or A is full, whichever happens first.

(a) Model this scenario with a state machine. (What are the states? How does a state change in response to a move?)

(b) Prove that we can put $k \in \mathbb{N}$ gallons of water into the receptacle using the above operations if and only if $\gcd(a, b) \mid k$. *Hint:* Use the fact that if a, b are positive integers then there exist integers s, t such that $\gcd(a, b) = sa + tb$ from Section 4.2.4.

Problem 8.2.
Here is a *very, very fun* game. We start with two distinct, positive integers written on a blackboard. Call them a and b. You and I now take turns. (I'll let you decide who goes first.) On each player's turn, he or she must write a new positive integer on the board that is the difference of two numbers that are already there. If a player can not play, then he or she loses.

For example, suppose that 12 and 15 are on the board initially. Your first play must be 3, which is $15 - 12$. Then I might play 9, which is $12 - 3$. Then you might play 6, which is $15 - 9$. Then I can not play, so I lose.

(a) Show that every number on the board at the end of the game is a multiple of $\gcd(a, b)$.

(b) Show that every positive multiple of $\gcd(a, b)$ up to $\max(a, b)$ is on the board at the end of the game.

(c) Describe a strategy that lets you win this game every time.

Problem 8.3.
In the late 1960s, the military junta that ousted the government of the small republic of Nerdia completely outlawed built-in multiplication operations, and also forbade division by any number other than 3. Fortunately, a young dissident found a way to help the population multiply any two nonnegative integers without risking persecution by the junta. The procedure he taught people is:

procedure *multiply*(x, y: nonnegative integers)
$r := x$;
$s := y$;
$a := 0$;
while $s \neq 0$ **do**
    if $3 \mid s$ **then**
        $r := r + r + r$;
        $s := s/3$;
    else if $3 \mid (s-1)$ **then**
        $a := a + r$;
        $r := r + r + r$;
        $s := (s-1)/3$;
    else
        $a := a + r + r$;
        $r := r + r + r$;
        $s := (s-2)/3$;
return a;

We can model the algorithm as a state machine whose states are triples of nonnegative integers (r, s, a). The initial state is $(x, y, 0)$. The transitions are given by the rule that for $s > 0$:

$$(r, s, a) \to \begin{cases} (3r, s/3, a) & \text{if } 3 \mid s \\ (3r, (s-1)/3, a + r) & \text{if } 3 \mid (s-1) \\ (3r, (s-2)/3, a + 2r) & \text{otherwise.} \end{cases}$$

(a) List the sequence of steps that appears in the execution of the algorithm for inputs $x = 5$ and $y = 10$.

(b) Use the Invariant Method to prove that the algorithm is partially correct —that is, if $s = 0$, then $a = xy$.

(c) Prove that the algorithm terminates after at most $1 + \log_3 y$ executions of the body of the `do` statement.

Problem 8.4.
A robot named Wall-E wanders around a two-dimensional grid. He starts out at $(0, 0)$ and is allowed to take four different types of step:

1. $(+2, -1)$
2. $(+1, -2)$
3. $(+1, +1)$

4. $(-3, 0)$

Thus, for example, Wall-E might walk as follows. The types of his steps are listed above the arrows.

$$(0,0) \xrightarrow{1} (2,-1) \xrightarrow{3} (3,0) \xrightarrow{2} (4,-2) \xrightarrow{4} (1,-2) \to \ldots$$

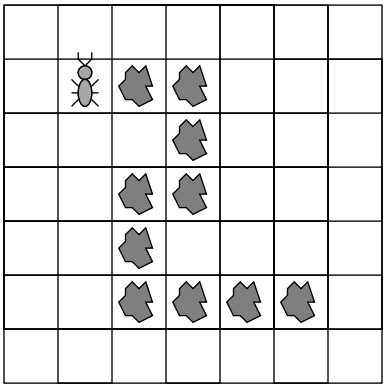
Wall-E's true love, the fashionable and high-powered robot, Eve, awaits at $(0, 2)$.

(a) Describe a state machine model of this problem.

(b) Will Wall-E ever find his true love? Either find a path from Wall-E to Eve or use the Invariant Principle to prove that no such path exists.

Problem 8.5.

A hungry ant is placed on an unbounded grid. Each square of the grid either contains a crumb or is empty. The squares containing crumbs form a path in which, except at the ends, every crumb is adjacent to exactly two other crumbs. The ant is placed at one end of the path and on a square containing a crumb. For example, the figure below shows a situation in which the ant faces North, and there is a trail of food leading approximately Southeast. The ant has already eaten the crumb upon which it was initially placed.



The ant can only smell food directly in front of it. The ant can only remember a small number of things, and what it remembers after any move only depends on what it remembered and smelled immediately before the move. Based on smell and memory, the ant may choose to move forward one square, or it may turn right or left. It eats a crumb when it lands on it.

The above scenario can be nicely modelled as a state machine in which each state is a pair consisting of the "ant's memory" and "everything else" —for example, information about where things are on the grid. Work out the details of such a

model state machine; design the ant-memory part of the state machine so the ant will eat all the crumbs on *any* finite path at which it starts and then signal when it is done. Be sure to clearly describe the possible states, transitions, and inputs and outputs (if any) in your model. Briefly explain why your ant will eat all the crumbs.

Note that the last transition is a self-loop; the ant signals done for eternity. One could also add another end state so that the ant signals done only once.

Problem 8.6.
Suppose that you have a regular deck of cards arranged as follows, from top to bottom:

$$A\heartsuit\ 2\heartsuit \ldots K\heartsuit\ A\spadesuit\ 2\spadesuit \ldots K\spadesuit\ A\clubsuit\ 2\clubsuit \ldots K\clubsuit\ A\diamondsuit\ 2\diamondsuit \ldots K\diamondsuit$$

Only two operations on the deck are allowed: *inshuffling* and *outshuffling*. In both, you begin by cutting the deck exactly in half, taking the top half into your right hand and the bottom into your left. Then you shuffle the two halves together so that the cards are perfectly interlaced; that is, the shuffled deck consists of one card from the left, one from the right, one from the left, one from the right, etc. The top card in the shuffled deck comes from the right hand in an outshuffle and from the left hand in an inshuffle.

(a) Model this problem as a state machine.

(b) Use the Invariant Principle to prove that you can not make the entire first half of the deck black through a sequence of inshuffles and outshuffles.

Note: Discovering a suitable invariant can be difficult! The standard approach is to identify a bunch of reachable states and then look for a pattern, some feature that they all share.[4]

Problem 8.7.
The following procedure can be applied to any digraph, G:

1. Delete an edge that is traversed by a directed cycle.
2. Delete edge $u \to v$ if there is a directed path from vertex u to vertex v that does not traverse $u \to v$.
3. Add edge $u \to v$ if there is no directed path in either direction between vertex u and vertex v.

Repeat these operations until none of them are applicable.

This procedure can be modeled as a state machine. The start state is G, and the states are all possible digraphs with the same vertices as G.

[4]If this does not work, consider twitching and drooling until someone takes the problem away.

(a) Let G be the graph with vertices $\{1, 2, 3, 4\}$ and edges

$$\{1 \to 2, 2 \to 3, 3 \to 4, 3 \to 2, 1 \to 4\}$$

What are the possible final states reachable from G?

A *line graph* is a graph whose edges can all be traversed by a directed simple path. All the final graphs in part (a) are line graphs.

(b) Prove that if the procedure terminates with a digraph, H, then H is a line graph with the same vertices as G.

Hint: Show that if H is *not* a line graph, then some operation must be applicable.

(c) Prove that being a DAG is a preserved invariant of the procedure.

(d) Prove that if G is a DAG and the procedure terminates, then the path relation of the final line graph is a topological sort of G.

Hint: Verify that the predicate

$$P(u, v) ::= \text{there is a directed path from } u \text{ to } v$$

is a preserved invariant of the procedure, for any two vertices u, v of a DAG.

(e) Prove that if G is finite, then the procedure terminates.

Hint: Let s be the number of simple cycles, e be the number of edges, and p be the number of pairs of vertices with a directed path (in either direction) between them. Note that $p \leq n^2$ where n is the number of vertices of G. Find coefficients a, b, c such that $as + bp + e + c$ is a strictly decreasing, $\mathbb{N}$-valued variable.

Class Problems

Problem 8.8.

In this problem you will establish a basic property of a puzzle toy called the *Fifteen Puzzle* using the method of invariants. The Fifteen Puzzle consists of sliding square tiles numbered $1, \ldots, 15$ held in a 4×4 frame with one empty square. Any tile adjacent to the empty square can slide into it.

The standard initial position is

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

We would like to reach the target position (known in my youth as "the impossible" — ARM):

15	14	13	12
11	10	9	8
7	6	5	4
3	2	1	

A state machine model of the puzzle has states consisting of a 4×4 matrix with 16 entries consisting of the integers $1, \ldots, 15$ as well as one "empty" entry—like each of the two arrays above.

The state transitions correspond to exchanging the empty square and an adjacent numbered tile. For example, an empty at position $(2, 2)$ can exchange position with tile above it, namely, at position $(1, 2)$:

n_1	n_2	n_3	n_4
n_5		n_6	n_7
n_8	n_9	n_{10}	n_{11}
n_{12}	n_{13}	n_{14}	n_{15}

$\longrightarrow$

n_1		n_3	n_4
n_5	n_2	n_6	n_7
n_8	n_9	n_{10}	n_{11}
n_{12}	n_{13}	n_{14}	n_{15}

We will use the invariant method to prove that there is no way to reach the target state starting from the initial state.

We begin by noting that a state can also be represented as a pair consisting of two things:

1. a list of the numbers $1, \ldots, 15$ in the order in which they appear—reading rows left-to-right from the top row down, ignoring the empty square, and

2. the coordinates of the empty square—where the upper left square has coordinates $(1, 1)$, the lower right $(4, 4)$.

(a) Write out the "list" representation of the start state and the "impossible" state.

Let L be a list of the numbers $1, \ldots, 15$ in some order. A pair of integers is an *out-of-order pair* in L when the first element of the pair both comes *earlier* in the list and *is larger*, than the second element of the pair. For example, the list $1, 2, 4, 5, 3$ has two out-of-order pairs: (4,3) and (5,3). The increasing list $1, 2 \ldots n$ has no out-of-order pairs.

Let a state, S, be a pair $(L, (i, j))$ described above. We define the *parity* of S to be the mod 2 sum of the number, $p(L)$, of out-of-order pairs in L and the row-number of the empty square, that is the parity of S is $p(L) + i \pmod 2$.

(b) Verify that the parity of the start state and the target state are different.

(c) Show that the parity of a state is preserved under transitions. Conclude that "the impossible" is impossible to reach.

By the way, if two states have the same parity, then in fact there *is* a way to get from one to the other. If you like puzzles, you'll enjoy working this out on your own.

Problem 8.9.

The most straightforward way to compute the bth power of a number, a, is to multiply a by itself b times. This of course requires $b - 1$ multiplications. There is another way to do it using considerably fewer multiplications. This algorithm is called *fast exponentiation*:

Given inputs $a \in \mathbb{R}, b \in \mathbb{N}$, initialize registers x, y, z to $a, 1, b$ respectively, and repeat the following sequence of steps until termination:

- if $z = 0$ **return** y and terminate
- $r :=$ remainder$(z, 2)$
- $z := \text{quotient}(z, 2)$
- if $r = 1$, then $y := xy$
- $x := x^2$

We claim this algorithm always terminates and leaves $y = a^b$.

(a) Model this algorithm with a state machine, carefully defining the states and transitions.

(b) Verify that the predicate $P((x, y, z)) ::= [yx^z = a^b]$ is a preserved invariant.

(c) Prove that the algorithm is partially correct: if it halts, it does so with $y = a^b$.

(d) Prove that the algorithm terminates.

(e) In fact, prove that it requires at most $2\lceil \log_2(b+1) \rceil$ multiplications for the Fast Exponentiation algorithm to compute a^b for $b > 1$.

Problem 8.10.

A robot moves on the two-dimensional integer grid. It starts out at $(0, 0)$, and is allowed to move in any of these four ways:

1. (+2,-1) Right 2, down 1
2. (-2,+1) Left 2, up 1
3. (+1,+3)
4. (-1,-3)

Prove that this robot can never reach (1,1).

Problem 8.11.

The Massachusetts Turnpike Authority is concerned about the integrity of the new Zakim bridge. Their consulting architect has warned that the bridge may collapse if more than 1000 cars are on it at the same time. The Authority has also been warned by their traffic consultants that the rate of accidents from cars speeding across bridges has been increasing.

Both to lighten traffic and to discourage speeding, the Authority has decided to make the bridge *one-way* and to put tolls at *both* ends of the bridge (don't laugh, this is Massachusetts). So cars will pay tolls both on entering and exiting the bridge,

but the tolls will be different. In particular, a car will pay \$3 to enter onto the bridge and will pay \$2 to exit. To be sure that there are never too many cars on the bridge, the Authority will let a car onto the bridge only if the difference between the amount of money currently at the entry toll booth minus the amount at the exit toll booth is strictly less than a certain threshold amount of $\$T_0$.

The consultants have decided to model this scenario with a state machine whose states are triples of natural numbers, (A, B, C), where

- A is an amount of money at the entry booth,
- B is an amount of money at the exit booth, and
- C is a number of cars on the bridge.

Any state with $C > 1000$ is called a *collapsed* state, which the Authority dearly hopes to avoid. There will be no transition out of a collapsed state.

Since the toll booth collectors may need to start off with some amount of money in order to make change, and there may also be some number of "official" cars already on the bridge when it is opened to the public, the consultants must be ready to analyze the system started at *any* uncollapsed state. So let A_0 be the initial number of dollars at the entrance toll booth, B_0 the initial number of dollars at the exit toll booth, and $C_0 \leq 1000$ the number of official cars on the bridge when it is opened. You should assume that even official cars pay tolls on exiting or entering the bridge after the bridge is opened.

(a) Give a mathematical model of the Authority's system for letting cars on and off the bridge by specifying a transition relation between states of the form (A, B, C) above.

(b) Characterize each of the following derived variables

$$A, B, A + B, A - B, 3C - A, 2A - 3B, B + 3C, 2A - 3B - 6C, 2A - 2B - 3C$$

as one of the following

constant	C
strictly increasing	SI
strictly decreasing	SD
weakly increasing but not constant	WI
weakly decreasing but not constant	WD
none of the above	N

and briefly explain your reasoning.

The Authority has asked their engineering consultants to determine T and to verify that this policy will keep the number of cars from exceeding 1000.

The consultants reason that if C_0 is the number of official cars on the bridge when it is opened, then an additional $1000 - C_0$ cars can be allowed on the bridge. So as long as $A - B$ has not increased by $3(1000 - C_0)$, there shouldn't more than 1000 cars on the bridge. So they recommend defining

$$T_0 ::= 3(1000 - C_0) + (A_0 - B_0), \tag{8.2}$$

where A_0 is the initial number of dollars at the entrance toll booth, B_0 is the initial number of dollars at the exit toll booth.

(c) Use the results of part (b) to define a simple predicate, P, on states of the transition system which is satisfied by the start state, that is $P(A_0, B_0, C_0)$ holds, is not satisfied by any collapsed state, and is a preserved invariant of the system. Explain why your P has these properties.

(d) A clever MIT intern working for the Turnpike Authority agrees that the Turnpike's bridge management policy will be *safe*: the bridge will not collapse. But she warns her boss that the policy will lead to *deadlock*— a situation where traffic can't move on the bridge even though the bridge has not collapsed.

Explain more precisely in terms of system transitions what the intern means, and briefly, but clearly, justify her claim.

Problem 8.12.

Start with 102 coins on a table, 98 showing heads and 4 showing tails. There are two ways to change the coins:

(i) flip over any ten coins, or

(ii) let n be the number of heads showing. Place $n+1$ additional coins, all showing tails, on the table.

For example, you might begin by flipping nine heads and one tail, yielding 90 heads and 12 tails, then add 91 tails, yielding 90 heads and 103 tails.

(a) Model this situation as a state machine, carefully defining the set of states, the start state, and the possible state transitions.

(b) Explain how to reach a state with exactly one tail showing.

(c) Define the following derived variables:

C	$::=$	the number of coins on the table,	H	$::=$	the number of heads,
T	$::=$	the number of tails,	C_2	$::=$	$\text{remainder}(C/2)$,
H_2	$::=$	$\text{remainder}(H/2)$,	T_2	$::=$	$\text{remainder}(T/2)$.

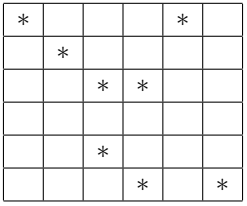
Which of these variables is

1. strictly increasing
2. weakly increasing
3. strictly decreasing
4. weakly decreasing
5. constant

(d) Prove that it is not possible to reach a state in which there is exactly one head showing.

Problem 8.13.
A classroom is designed so students sit in a square arrangement. An outbreak of beaver flu sometimes infects students in the class; beaver flu is a rare variant of bird flu that lasts forever, with symptoms including a yearning for more quizzes and the thrill of late night problem set sessions.

Here is an illustration of a 6×6-seat classroom with seats represented by squares. The locations of infected students are marked with an asterisk.

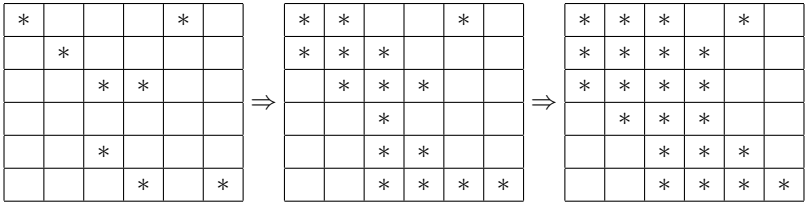


Outbreaks of infection spread rapidly step by step. A student is infected after a step if either

- the student was infected at the previous step (since beaver flu lasts forever), or
- the student was adjacent to *at least two* already-infected students at the previous step.

Here *adjacent* means the students' individual squares share an edge (front, back, left or right); they are not adjacent if they only share a corner point. So each student is adjacent to 2, 3 or 4 others.

In the example, the infection spreads as shown below.



In this example, over the next few time-steps, all the students in class become infected.

Theorem. *If fewer than n students among those in an $n \times n$ arrangment are initially infected in a flu outbreak, then there will be at least one student who never gets infected in this outbreak, even if students attend all the lectures.*

Prove this theorem.

Hint: Think of the state of an outbreak as an $n \times n$ square above, with asterisks indicating infection. The rules for the spread of infection then define the transitions of a state machine. Try to derive a weakly decreasing state variable that leads to a proof of this theorem.

8.5 The Alternating Bit Protocol

The Alternating Bit Protocol is a well-known two-process communication protocol that achieves reliable FIFO communication over unreliable channels. The unreliable channels may lose or duplicate messages, but are assumed not to reorder them. We'll use the Invariant Method to verify that the Protocol

The Protocol allows a **Sender** process to send a sequence of messages from a message alphabet, M, to a **Receiver** process. It works as follows.

Sender repeatedly sends the rightmost message in its **outgoing-queue** of messages, tagged with a **tagbit** that is initially 1. When **Receiver** receives this tagged message, it sets its **ackbit** to be the message tag 1, and adds the message to the lefthand end of its **received-msgs** list. Then as an acknowledgement, **Receiver** sends back **ackbit** 1 repeatedly. When **Sender** gets this acknowledgement bit, it deletes the rightmost outgoing message in its queue, sets its **tagbit** to 0, and begins sending the new rightmost outgoing message, tagged with **tagbit**.

Receiver, having already accepted the message tagged with **ackbit** 1, ignores subsequent messages with tag 1, and waits until it sees the first message with tag 0; it adds this message to the lefthand side of its **received-msgs** list, sets **ackbit** to 0 and acknowledges repeatedly with with **ackbit** 0. **Sender** now waits till it gets acknowledgement bit 0, then goes on to send the next outgoing message with tag 1. In this way, it alternates use of the tags 1 and 0 for successive messages.

We claim that this causes **Sender** to receive *suffix* original **outgoing-msgs** queue. That is, at any stage in the process when the the **outgoing-msgs**

(The fact that **Sender** actually outputs the entire outgoing queuee is a *liveness* claim —liveness properties are a generalization of termination properties. We'll ignore this issue for now.)

We formalize the description above as a state whose states consist of:

outgoing-msgs, a finite sequence of M, whose initial value is called **all-msgs**
tagbit $\in \{0, 1\}$, initially 1

received-msgs, a finite sequence of M, initially empty
ackbit$(\in \{0, 1\}$, initially 0

msg-channel, a finite sequence of $M \times \{0, 1\}$, initially empty,
ack-channel, a finite sequence of $\{0, 1\}$, initially empty

The transitions are:

SEND: (a) **action**: **send-msg**(m, b)
precondition: $m =$ **rightend**(**outgoing-msgs**) AND $b =$ **tagbit**
effect: add (m, b) to the lefthand end of **msg-channel**, any number ≥ 0 of times

(b) **action**: **send-ack**(b)
precondition: $b =$ **ackbit**
effect: add b to the righthand end of **ack-channel**, any number ≥ 0 of times

RECEIVE: (a) **action**: **receive-msg**(m, b)
precondition: $(m, b) =$ **rightend**(**msg-channel**)
effect: remove **rightend** of **msg-channel**;
if $b \neq$ **ackbit**, then [add m to the lefthand end of **receive-msgs**; **ackbit** $:=$ b.]

(b) **action**: **receive-ack**(b)
precondition: $b =$ **leftend**(**ack-channel**())
effect: remove **leftend** of **ack-channel**.
if $b =$ **tagbit**, then [remove **rightend** of **outgoing-msgs** (if nonempty); **tagbit** $:=$ $\overline{\textbf{tagbit}}$]

Our goal is to show that when **tagbit** $\neq$ **ackbit**, then

$$\textbf{outgoing-queue} \cdot \textbf{received-msgs} = \textbf{all-msgs}. \tag{8.3}$$

This requires three auxiliary invariants. For the first of these, we need a definition.

Let **tag-sequence** be the sequence consisting of bits in **ack-channel**, in right-to-left order, followed by **tagbit**, followed by the tag components of the elements of **msg-channel**, in left-to-right order, followed by **ackbit**.

Property 2: tag-sequence consists of one of the following:

1. All 0's.
2. All 1's.
3. A positive number of 0's followed by a positive number of 1's.
4. A positive number of 1's followed by a positive number of 0's.

What is being ruled out by these four cases is the situation where the sequence contains more than one switch of tag value.

The fact that Property 2 is an invariant can be proved easily by induction. We also need:

Property 3: If $(m,$ **tag**$)$ is in **msg-channel** then $m =$ **rightend**(**outgoing-queue**).

Proof. (That Property 3 is an invariant)

By induction, using Property 2.

Base: Obvious, since no message is in the channel initially.

Inductive step: It is easy to see that the property is preserved by $\textbf{send}_{m,b}$, which adds new messages to $\textbf{channel}_{1,2}$. The only other case that could cause a problem is $\textbf{receive}(b)_{2,1}$, which could cause $\textbf{tag}_1$ to change when there is another message already in $\textbf{channel}_{1,2}$ with the same tag. But this can't happen, by Property 2 applied before the step – since the incoming tag g must be equal to $\textbf{tag}_1$ in this case, all the tags in **tag-sequence** must be the same. ■

Finally, we need that the following counterpart to (8.3): when **tagbit** = **ackbit**, then

$$\textbf{lefttail}(\textbf{outgoing-queue}) \cdot \textbf{received-msgs} = \textbf{all-msgs}, \tag{8.4}$$

where **lefttail**(**outgoing-queue**) all but the rightmost message, if any, in **outgoing-queue**.

Property 4, part 2, easily implies the goal Property 1. It also implies that $\textbf{work-buf}_2$ is always nonempty when $\textbf{receive}(b)_{2,1}$ occurs with equal tags; therefore, the parenthetical check in the code always works out to be true.

Proof. (That Property 4 is an invariant)

By induction. Base: In an initial state, the tags are unequal, $\textbf{work-buf}_1 = \textbf{buf}_1$ and $\textbf{buf}_2$ is empty. This suffices to show part 1. part 2 is vacuous.

Inductive step: When a **send** occurs, the tags and buffers are unchanged, so the truth of the invariants must be preserved. It remains to consider **receive** events.

$\textbf{receive}(m, b)_{1,2}$:

If $b = \textbf{tag}_2$, nothing happens, so the invariants are preserved. So suppose that $b \neq \textbf{tag}_2$. Then Property 2 implies that $b = \textbf{tag}_1$, and then Property 3 implies that m is the first message on $\textbf{work-buf}_1$. The effect of the transition is to change $\textbf{tag}_2$ to make it equal to $\textbf{tag}_1$, and to replicate the first element of $\textbf{work-buf}22_1$ at the end of $\textbf{buf}_2$.

The inductive hypothesis implies that, before the step, $\textbf{buf}_2 \cdot \textbf{work-buf}_1 = \textbf{buf}_1$. The changes caused by the step imply that, after the step, $\textbf{tag}_1 = \textbf{tag}_2$, $\textbf{work-buf}_1$ and $\textbf{buf}_2$ are nonempty, $\textbf{head}(\textbf{work-buf}_1) = \textbf{last}(\textbf{buf}_2)$, and $\textbf{buf}_2 \cdot \textbf{tail}(\textbf{work-buf}_1) = \textbf{buf}_1$. This is as needed.

$\textbf{receive}(b)_{2,1}$:

The argument is similar to the one for $\textbf{receive}(m, b)_{1,2}$. If $b \neq \textbf{tag}_1$, nothing happens so the invariants are preserved. So suppose that $b = \textbf{tag}_1$. Then Property 2 implies that $b = \textbf{tag}_2$, and the step changes $\textbf{tag}_1$ to make it unequal to $\textbf{tag}44_2$. The step also removes the first element of $\textbf{work-buf}_1$. The inductive hypothesis implies that, before the step, $\textbf{work-buf}_1$ and $\textbf{buf}_2$ are nonempty, $\textbf{head}(\textbf{work-buf}_1) = \textbf{last}(\textbf{buf}_2)$, and $\textbf{buf}_2 \cdot \textbf{tail}(\textbf{work-buf}_1) = \textbf{buf}_1$. The changes caused by the step imply that, after the step, $\textbf{tag}_1 \neq \textbf{tag}_2$ and $\textbf{buf}_2 \cdot \textbf{work-buf}_1 = \textbf{buf}_1$. This is as needed. ■

8.5.1 Applications

Not surprisingly, a stable matching procedure is used by at least one large dating agency. But although "boy-girl-marriage" terminology is traditional and makes some of the definitions easier to remember (we hope without offending anyone), solutions to the Stable Marriage Problem are widely useful.

The Mating Ritual was first announced in a paper by D. Gale and L.S. Shapley in 1962, but ten years before the Gale-Shapley paper was appeared, and unknown by them, the Ritual was being used to assign residents to hospitals by the National

Resident Matching Program (NRMP). The NRMP has, since the turn of the twentieth century, assigned each year's pool of medical school graduates to hospital residencies (formerly called "internships") with hospitals and graduates playing the roles of boys and girls. (In this case there may be multiple boys married to one girl, but there's an easy way to use the Ritual in this situation (see Problem 9.12). Before the Ritual was adopted, there were chronic disruptions and awkward countermeasures taken to preserve assignments of graduates to residencies. The Ritual resolved these problems so successfully, that it was used essentially without change at least through 1989.[5]

MIT Math Prof. Tom Leighton, who regularly teaches 6.042 and also founded the internet infrastructure company, Akamai, reports another application. Akamai uses a variation of the Gale-Shapley procedure to assign web traffic to servers. In the early days, Akamai used other combinatorial optimization algorithms that got to be too slow as the number of servers and traffic increased. Akamai switched to Gale-Shapley since it is fast and can be run in a distributed manner. In this case, the web traffic corresponds to the boys and the web servers to the girls. The servers have preferences based on latency and packet loss; the traffic has preferences based on the cost of bandwidth.

8.5.2 Problems

Practice Problems

Problem 8.14.
Four Students want separate assignments to four VI-A Companies. Here are their preference rankings:

Student	Companies
Albert:	HP, Bellcore, AT&T, Draper
Rich:	AT&T, Bellcore, Draper, HP
Megumi:	HP, Draper, AT&T, Bellcore
Justin:	Draper, AT&T, Bellcore, HP

Company	Students
AT&T:	Justin, Albert, Megumi, Rich
Bellcore:	Megumi, Rich, Albert, Justin
HP:	Justin, Megumi, Albert, Rich
Draper:	Rich, Justin, Megumi, Albert

(a) Use the Mating Ritual to find *two* stable assignments of Students to Companies.

(b) Describe a simple procedure to determine whether any given stable marriage problem has a unique solution, that is, only one possible stable matching.

[5]Much more about the Stable Marriage Problem can be found in the very readable mathematical monograph by Dan Gusfield and Robert W. Irving, The Stable Marriage Problem: Structure and Algorithms, MIT Press, Cambridge, Massachusetts, 1989, 240 pp.

Problem 8.15.
Suppose that Harry is one of the boys and Alice is one of the girls in the *Mating Ritual*. Which of the properties below are preserved invariants? Why?

a. Alice is the only girl on Harry's list.

b. There is a girl who does not have any boys serenading her.

c. If Alice is not on Harry's list, then Alice has a suitor that she prefers to Harry.

d. Alice is crossed off Harry's list and Harry prefers Alice to anyone he is serenading.

e. If Alice is on Harry's list, then she prefers to Harry to any suitor she has.

Class Problems

Problem 8.16.
A preserved invariant of the Mating ritual is:

> For every girl, G, and every boy, B, if G is crossed off B's list, then G has a favorite suitor and she prefers him over B.

Use the invariant to prove that the Mating Algorithm produces stable marriages. (Don't look up the proof in the Notes or slides.)

Problem 8.17.
Consider a stable marriage problem with 4 boys and 4 girls and the following partial information about their preferences:

B1:	G1	G2	–	–
B2:	G2	G1	–	–
B3:	–	–	G4	G3
B4:	–	–	G3	G4
G1:	B2	B1	–	–
G2:	B1	B2	–	–
G3:	–	–	B3	B4
G4:	–	–	B4	B3

(a) Verify that

$$(B1, G1), (B2, G2), (B3, G3), (B4, G4)$$

will be a stable matching whatever the unspecified preferences may be.

(b) Explain why the stable matching above is neither boy-optimal nor boy-pessimal and so will not be an outcome of the Mating Ritual.

(c) Describe how to define a set of marriage preferences among n boys and n girls which have at least $2^{n/2}$ stable assignments.

Hint: Arrange the boys into a list of $n/2$ pairs, and likewise arrange the girls into a list of $n/2$ pairs of girls. Choose preferences so that the kth pair of boys ranks the kth pair of girls just below the previous pairs of girls, and likewise for the kth pair of girls. Within the kth pairs, make sure each boy's first choice girl in the pair prefers the other boy in the pair.

Homework Problems

Problem 8.18.
The most famous application of stable matching was in assigning graduating medical students to hospital residencies. Each hospital has a preference ranking of students and each student has a preference order of hospitals, but unlike the setup in the notes where there are an equal number of boys and girls and monogamous marriages, hospitals generally have differing numbers of available residencies, and the total number of residencies may not equal the number of graduating students. Modify the definition of stable matching so it applies in this situation, and explain how to modify the Mating Ritual so it yields stable assignments of students to residencies. No proof is required.

Problem 8.19.
Give an example of a stable matching between 3 boys and 3 girls where no person gets their first choice. Briefly explain why your matching is stable.

Problem 8.20.
In a stable matching between n boys and girls produced by the Mating Ritual, call a person *lucky* if they are matched up with one of their $\lceil n/2 \rceil$ top choices. We will prove:

Theorem. *There must be at least one lucky person.*

To prove this, define the following derived variables for the Mating Ritual:

$q(B) = j$, where j is the rank of the girl that boy B is courting. That is to say, boy B is always courting the jth girl on his list.

$r(G)$ is the number of boys that girl G has rejected.

(a) Let

$$S ::= \sum_{B \in \text{Boys}} q(B) - \sum_{G \in \text{Girls}} r(G). \tag{8.5}$$

Show that S remains the same from one day to the next in the Mating Ritual.

(b) Prove the Theorem above. (You may assume for simplicity that n is even.)
Hint: A girl is sure to be lucky if she has rejected half the boys.

8.6 Reasoning About While Programs

Real programs and programming languages are often huge and complicated, making them hard to model and even harder to reason about. Still, making programs "reasonable" is a crucial aspect of software engineering. In this section we'll illustrate what it means to have a clean mathematical model of a simple programming language and reasoning principles that go with it —if only real programming languages allowed for such simple, accurate modeling.

8.6.1 While Programs

The programs we'll study are called "***while** programs*." We can define them as a recursive data type:

Definition 8.6.1.
base cases:

- $x := e$ is a **while** program, called an *assignment statement*, where x is a variable and e is an expression.
- ***Done*** is a **while** program.

constructor cases: If C and D are **while** programs, and T is a test, then the following are also **while** programs:

- $C;D$ —called the *sequencing* of C and D,
- **if** T **then** C **else** D —called a *conditional* with *test*, T, and *branches*, C and D,
- **while** T **do** C **od** —called a *while loop* with *test*, T, and *body*, C.

For simplicity we'll stick to **while** programs operating on integers. So by expressions we'll mean any of the familiar integer valued expressions involving integer constants and operations such as addition, multiplication, exponentiation, quotient or remainder. As *tests*, we'll allow propositional formulas built from basic formulas of the form $e \leq f$ where e and f are expressions. For example, here is the Euclidean algorithm for $\gcd(a, b)$ expressed as a **while** program.

$\mathtt{x} := a$**;**
$\mathtt{y} := b$**;**
while $\mathtt{y} \neq 0$ **do**
    $\mathtt{t} := \mathtt{y}$**;**
    $\mathtt{y} := \mathrm{rem}(\mathtt{x}, \mathtt{y})$**;**
    $\mathtt{x} := \mathtt{t}$ **od**

8.6.2 The While Program State Machine

A **while** program acts as a pure command: it is run solely for its side effects on stored data and it doesn't return a value. The data consists of integers stored as the values of variables, namely environments:

Definition 8.6.2. An *environment* is a total function from variables to integers. Let Env be set of all environments.

So if ρ is an environment and x is a variable, then $\rho(x)$ is an integer. More generally, the environment determines the integer value of each expression, e, and the truth value of each test, T. We can think of an expression, e as defining a function $[\![e]\!] : \text{Env} \to \mathbb{Z}$, and refer to this function, $[\![e]\!]$ as the *meaning* of e, and likewise for tests.

It's standard in programming language theory to write $[\![e]\!]\rho$ as shorthand for $[\![e]\!](\rho)$, that is, applying the *meaning*, $[\![e]\!]$, of e to ρ. For example, if $\rho(\mathtt{x}) = 4$, and $\rho(\mathtt{y}) = -2$, then

$$[\![\mathtt{x}^2 + \mathtt{y} - 3]\!]\rho = \rho(\mathtt{x})^2 + \rho(y) - 3 = 11. \tag{8.6}$$

Executing a program causes a succession of changes to the environment[6] which may continue until the program halts. Actually the only command which immediately alters the environment is an assignment command. Namely, the effect of the command

$$x := e$$

on an environment is that the value assigned to the variable x is changed to the value of e in the original environment. We can say this precisely and concisely using the following notation: $f[a \leftarrow b]$ is a function that is the same as the function, f, except that when applied to element a its value is b. Namely,

Definition 8.6.3. If $f : A \to B$ is a function and a, b are arbitrary elements, define

$$f[a \leftarrow b]$$

to be the function g such that

$$g(u) = \begin{cases} b & \text{if } u = a. \\ f(u) & \text{otherwise.} \end{cases}$$

Now we can specify the step-by-step execution of a **while** program as a state machine, where the states of the machine consist of a **while** program paired with an environment. The transitions of this state machine are defined recursively on the definition of **while** programs.

Definition 8.6.4. The transitions $\langle C, \rho\rangle \longrightarrow \langle D, \rho'\rangle$ of the **while** program state machine are defined as follows:

[6] More sophisticated programming models distinguish the environment from a *store* which is affected by commands, but this distinction is unnecessary for our purposes.

base cases:

$$\langle x := e,\ \rho\rangle \longrightarrow \langle \textbf{Done},\ \rho[x \leftarrow [\![e]\!]\rho]\rangle$$

constructor cases: If C and D are **while** programs, and T is a test, then:

- if $\langle C,\ \rho\rangle \longrightarrow \langle C',\ \rho'\rangle$, then

  $$\langle C;D,\ \rho\rangle \longrightarrow \langle C';D,\ \rho'\rangle\,.$$

  Also,

  $$\langle \textbf{Done};D,\ \rho\rangle \longrightarrow \langle D,\ \rho\rangle\,.$$

- if $[\![T]\!]\rho = \mathbf{T}$, then

  $$\langle \textbf{if } T \textbf{ then } C \textbf{ else } D,\ \rho\rangle \longrightarrow \langle C,\ \rho\rangle\,,$$

  or if $[\![T]\!]\rho = \mathbf{F}$, then

  $$\langle \textbf{if } T \textbf{ then } C \textbf{ else } D,\ \rho\rangle \longrightarrow \langle D,\ \rho\rangle\,.$$

- if $[\![T]\!]\rho = \mathbf{T}$, then

  $$\langle \textbf{while } T \textbf{ do } C \textbf{ od},\ \rho\rangle \longrightarrow \langle C;\textbf{while } T \textbf{ do } C \textbf{ od},\ \rho\rangle$$

  or if $[\![T]\!]\rho = \mathbf{F}$, then

  $$\langle \textbf{while } T \textbf{ do } C \textbf{ od},\ \rho\rangle \longrightarrow \langle \textbf{Done},\ \rho\rangle\,.$$

Now **while** programs are probably going to be the simplest kind of programs you will ever see, but being condescending about them would be a mistake. It turns that *every function on nonnegative integers that can be computed by any program* on any machine whatsoever can also be computed by **while** programs (maybe more slowly). We can't take the time to explain how such a sweeping claim can be justified, but you can find out by taking a course in computability theory such as 6.045 or 6.840.

8.6.3 Denotational Semantics

The net effect of starting a **while** program in some environment is reflected in the final environment when the program halts. So we can think of a **while** program, C, aas defining a function, $[\![C]\!] : \text{Env} \to \text{Env}$, from initial environments to environments at halting. The function $[\![C]\!]$ is called the *meaning* of C.

$[\![C]\!]$ of a **while** program, C to be a partial function from Env to Env mapping an initial environment to the final halting environment.

We'll need one bit of notation first. For any function $f : S \to S$, let $f^{(n)}$ be the composition of f with itself n times where $n \in \mathbb{N}$. Namely,

$$f^{(0)} ::= \mathrm{Id}_S$$
$$f^{(n+1)} ::= f \circ f^{(n)},$$

where "$\circ$" denotes functional composition.

The recursive definition of the meaning of a program follows the definition of the **while** program recursive data type.

Definition 8.6.5. base cases:

- $[\![x := e]\!]$ is the function from Env to Env defined by the rule:

$$[\![x := e]\!]\rho ::= \rho[x \leftarrow [\![e]\!]\rho].$$

- $$[\![\textbf{Done}]\!] ::= \mathrm{Id}_{\mathrm{Env}}$$

  where $\mathrm{Id}_{\mathrm{Env}}$ is the identity function on Env. In other words, $[\![\textbf{Done}]\!]\rho ::= \rho$.

constructor cases: If C and D are **while** programs, and T is a test, then:

- $$[\![C;D]\!] ::= [\![D]\!] \circ [\![C]\!]$$

  That is,

$$[\![C;D]\!]\rho ::= [\![D]\!]([\![C]\!]\rho).$$

- $$[\![\textbf{if } T \textbf{ then } C \textbf{ else } D]\!]\rho ::= \begin{cases} [\![C]\!]\rho & \text{if } [\![T]\!]\rho = \mathbf{T} \\ [\![D]\!]\rho & \text{if } [\![T]\!]\rho = \mathbf{F}. \end{cases}$$

- $$[\![\textbf{while } T \textbf{ do } C \textbf{ od}]\!]\rho ::= [\![C]\!]^{(n)}\rho$$

  where n is the least nonnegative integer such that $[\![T]\!]([\![C]\!]^{(n)}\rho) = \mathbf{F}$. (If there is no such n, then $[\![\textbf{while } T \textbf{ do } C \textbf{ od}]\!]\rho$ is undefined.)

We can use the denotational semantics of **while** programs to reason about **while** programs using structural induction on programs, and this is often much simpler than reasoning about them using induction on the number of steps in an execution. This is OK as long as the denotational semantics accurately captures the state machine behavior. In particular, using the notation $\longrightarrow^*$ for the transitive closure of the transition relation:

Theorem 8.6.6.

$$\langle C, \rho\rangle \longrightarrow^* \langle \textit{Done}, \rho'\rangle \quad \textit{iff} \quad [\![C]\!]\rho = \rho'$$

Theorem 8.6.6 can be proved easily by induction; it appear in Problem8.21.

8.6.4 Problems

Homework Problems

Problem 8.21.
Prove

Theorem 8.6.7.

$$\langle C,\, \rho\rangle \longrightarrow^* \langle \textbf{\textit{Done}},\, \rho'\rangle \quad \textit{iff} \quad [\![C]\!]\rho = \rho'$$

Hint: Prove the left to right direction by induction on the number of steps C needs to halt starting in environment ρ. Prove the right to left direction by structural induction on the definition of **while** programs. Both proofs follow almost mechanically from the definitions.

8.6.5 Logic of Programs

A typical program specification describes the kind of inputs and environments the program should handle, and then describes what should result from an execution. The specification of the inputs or initial environment is called the *precondition* for program execution, and the prescription of what the result of execution should be is called the *postcondition*. So if P is a logical formula expressing the precondition for a program, C, and likewise Q expresses the postcondition, the specification requires that

If P holds when C is started, then Q will hold if and when C halts.

We'll express this requirement as a formula

$$P\ \{C\}\ Q$$

called a *partial correctness assertion*.

For example, if E is **while** program above for the Euclidean algorithm, then the partial correctness of E can be expressed as

$$(a, b \in \mathbb{N} \text{ AND } \mathtt{x} \neq 0)\ \{C\}\ (\mathtt{x} = \gcd(a, b))\,. \tag{8.7}$$

More precisely, notice that just as the value of an expression in an environment is an integer, the value of a logical formula in an environment is a truth value. For example, if $\rho(\mathtt{x}) = 4$, and $\rho(\mathtt{y}) = -2$, then by (8.6), $[\![\mathtt{x}^2 + \mathtt{y} - 3]\!]\rho = 11$, so

$$[\![\exists \mathtt{z}.\, \mathtt{z} > 4 \text{ AND } \mathtt{x}^2 + \mathtt{y} - 3 = \mathtt{z}]\!]\rho = \mathbf{T},$$
$$[\![\exists \mathtt{z}.\, \mathtt{z} > 13 \text{ AND } \mathtt{x}^2 + \mathtt{y} - 3 = \mathtt{z}]\!]\rho = \mathbf{F}.$$

Definition 8.6.8. For logical formulas P and Q, and **while** program, C, the partial correctness assertion

$$P\ \{C\}\ Q$$

is true proving that for all environments, rho, if $[\![P]\!]\rho$ is true, and $\langle C,\, \rho\rangle \longrightarrow^* \langle \textbf{Done},\, \rho'\rangle$ for some ρ', then $[\![Q]\!]\rho'$ is true.

In the 1970', Univ. Dublin formulated a set of inference rules for proving partial correctness formulas. These rules are known as *Hoare Logic*.

The first rule captures the fact that strengthening the preconditions and weakening the postconditions makes a partial correctness specification easier to satisfy:

$$\frac{P \text{ IMPLIES } R, \quad R\ \{C\}\ S, \quad S \text{ IMPLIES } Q}{P\ \{C\}\ Q}$$

The rest of the logical rules follow the recursive definition of **while** programs. There are axioms for the base case commands:

$$P(x)\ \{x := e\}\ P(e)$$
$$P\ \{\textbf{Done}\}\ P,$$

and proof rules for the constructor cases:

- $$\frac{P\ \{C\}\ Q \text{ AND } Q\ \{D\}\ R}{P\ \{C;D\}\ R}$$
- $$\frac{P \text{ AND } T\ \{C\}\ Q}{P \text{ AND } T\ \{\textbf{if } T \textbf{ then } C \textbf{ else } D\}\ Q \text{ AND } T}$$
- $$\frac{P \text{ AND } T\ \{C\}\ P}{P\ \{\textbf{while } T \textbf{ do } C \textbf{ od}\}\ P \text{ AND NOT}(T)}$$

Example 8.6.9. **Proof of partial correctness** (8.7) **for the Euclidean algorithm.**

TBA - Brief discussion of "relative completeness".

Chapter 9

Simple Graphs

Graphs in which edges are *not* directed are called *simple graphs*. They come up in all sorts of applications, including scheduling, optimization, communications, and the design and analysis of algorithms. Two Stanford students even used graph theory to become multibillionaires!

But we'll start with an application designed to get your attention: we are going to make a professional inquiry into sexual behavior. Namely, we'll look at some data about who, on average, has more opposite-gender partners, men or women.

Sexual demographics have been the subject of many studies. In one of the largest, researchers from the University of Chicago interviewed a random sample of 2500 people over several years to try to get an answer to this question. Their study, published in 1994, and entitled *The Social Organization of Sexuality* found that on average men have 74% more opposite-gender partners than women.

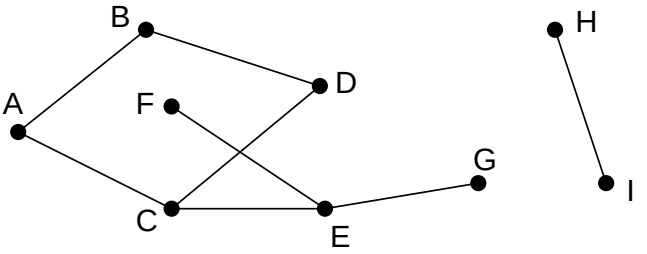
Other studies have found that the disparity is even larger. In particular, ABC News claimed that the average man has 20 partners over his lifetime, and the average woman has 6, for a percentage disparity of 233%. The ABC News study, aired on Primetime Live in 2004, purported to be one of the most scientific ever done, with only a 2.5% margin of error. It was called "American Sex Survey: A peek between the sheets," —which raises some question about the seriousness of their reporting.

Yet again, in August, 2007, the N.Y. Times reported on a study by the National Center for Health Statistics of the U.S. government showing that men had seven partners while women had four. Anyway, whose numbers do you think are more accurate, the University of Chicago, ABC News, or the National Center? —don't answer; this is a setup question like "When did you stop beating your wife?" Using a little graph theory, we'll explain why none of these findings can be anywhere near the truth.

9.1 Degrees & Isomorphism

9.1.1 Definition of Simple Graph

Informally, a graph is a bunch of dots with lines connecting some of them. Here is an example:



For many mathematical purposes, we don't really care how the points and lines are laid out —only which points are connected by lines. The definition of *simple graphs* aims to capture just this connection data.

Definition 9.1.1. A *simple graph*, G, consists of a nonempty set, V, called the *vertices* of G, and a collection, E, of two-element subsets of V. The members of E are called the *edges* of G.

The vertices correspond to the dots in the picture, and the edges correspond to the lines. For example, the connection data given in the diagram above can also be given by listing the vertices and edges according to the official definition of simple graph:

$$V = \{A, B, C, D, E, F, G, H, I\}$$
$$E = \{\{A, B\}, \{A, C\}, \{B, D\}, \{C, D\}, \{C, E\}, \{E, F\}, \{E, G\}, \{H, I\}\}.$$

It will be helpful to use the notation A—B for the edge $\{A, B\}$. Note that A—B and B—A are different descriptions of the same edge, since sets are unordered.

So the definition of simple graphs is the same as for directed graphs, except that instead of a directed edge $v \to w$ which starts at vertex v and ends at vertex w, a simple graph only has an undirected edge, v—w, that connects v and w.

Definition 9.1.2. Two vertices in a simple graph are said to be *adjacent* if they are joined by an edge, and an edge is said to be *incident* to the vertices it joins. The number of edges incident to a vertex is called the *degree* of the vertex; equivalently, the degree of a vertex is equals the number of vertices adjacent to it.

For example, in the simple graph above, A is adjacent to B and B is adjacent to D, and the edge A—C is incident to vertices A and C. Vertex H has degree 1, D has degree 2, and E has degree 3.

Graph Synonyms

A synonym for "vertices" is "*nodes*," and we'll use these words interchangeably. Simple graphs are sometimes called *networks*, edges are sometimes called *arcs*. We mention this as a "heads up" in case you look at other graph theory literature; we won't use these words.

Some technical consequences of Definition 9.1.1 are worth noting right from the start:

1. Simple graphs do *not* have self-loops ($\{a, a\}$ is not an undirected edge because an undirected edge is defined to be a set of *two* vertices.)

2. There is at most one edge between two vertices of a simple graph.

3. Simple graphs have at least one vertex, though they might not have any edges.

There's no harm in relaxing these conditions, and some authors do, but we don't need self-loops, multiple edges between the same two vertices, or graphs with no vertices, and it's simpler not to have them around.

For the rest of this Chapter we'll only be considering simple graphs, so we'll just call them "graphs" from now on.

9.1.2 Sex in America

Let's model the question of heterosexual partners in graph theoretic terms. To do this, we'll let G be the graph whose vertices, V, are all the people in America. Then we split V into two separate subsets: M, which contains all the males, and F, which contains all the females.[1] We'll put an edge between a male and a female iff they have been sexual partners. This graph is pictured in Figure 9.1 with males on the left and females on the right.

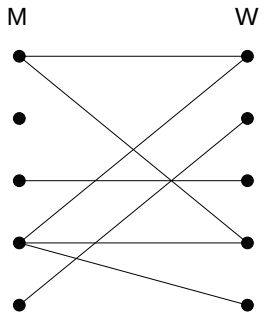


Figure 9.1: The sex partners graph

[1]For simplicity, we'll ignore the possibility of someone being both, or neither, a man and a woman.

Actually, this is a pretty hard graph to figure out, let alone draw. The graph is *enormous*: the US population is about 300 million, so $|V| \approx 300M$. Of these, approximately 50.8% are female and 49.2% are male, so $|M| \approx 147.6M$, and $|F| \approx 152.4M$. And we don't even have trustworthy estimates of how many edges there are, let alone exactly which couples are adjacent. But it turns out that we don't need to know any of this —we just need to figure out the relationship between the average number of partners per male and partners per female. To do this, we note that every edge is incident to exactly one M vertex (remember, we're only considering male-female relationships); so the sum of the degrees of the M vertices equals the number of edges. For the same reason, the sum of the degrees of the F vertices equals the number of edges. So these sums are equal:

$$\sum_{x \in M} \deg(x) = \sum_{y \in F} \deg(y).$$

Now suppose we divide both sides of this equation by the product of the sizes of the two sets, $|M| \cdot |F|$:

$$\left(\frac{\sum_{x \in M} \deg(x)}{|M|}\right) \cdot \frac{1}{|F|} = \left(\frac{\sum_{y \in F} \deg(y)}{|F|}\right) \cdot \frac{1}{|M|}$$

The terms above in parentheses are the *average degree of an M vertex* and the *average degree of a F* vertex. So we know:

$$\text{Avg. deg in } M = \frac{|F|}{|M|} \cdot \text{Avg. deg in } F$$

In other words, we've proved that the average number of female partners of males in the population compared to the average number of males per female is *determined solely by the relative number of males and females in the population*.

Now the Census Bureau reports that there are slightly more females than males in America; in particular $|F| / |M|$ is about 1.035. So we know that on average, males have 3.5% more opposite-gender partners than females, and this tells us nothing about any sex's promiscuity or selectivity. Rather, it just has to do with the relative number of males and females. Collectively, males and females have the same number of opposite gender partners, since it takes one of each set for every partnership, but there are fewer males, so they have a higher ratio. This means that the University of Chicago, ABC, and the Federal government studies are way off. After a huge effort, they gave a totally wrong answer.

There's no definite explanation for why such surveys are consistently wrong. One hypothesis is that males exaggerate their number of partners —or maybe females downplay theirs —but these explanations are speculative. Interestingly, the principal author of the National Center for Health Statistics study reported that she knew the results had to be wrong, but that was the data collected, and her job was to report it.

The same underlying issue has led to serious misinterpretations of other survey data. For example, a couple of years ago, the Boston Globe ran a story on a survey

of the study habits of students on Boston area campuses. Their survey showed that on average, minority students tended to study with non-minority students more than the other way around. They went on at great length to explain why this "remarkable phenomenon" might be true. But it's not remarkable at all —using our graph theory formulation, we can see that all it says is that there are fewer minority students than non-minority students, which is, of course what "minority" means.

9.1.3 Handshaking Lemma

The previous argument hinged on the connection between a sum of degrees and the number edges. There is a simple connection between these in any graph:

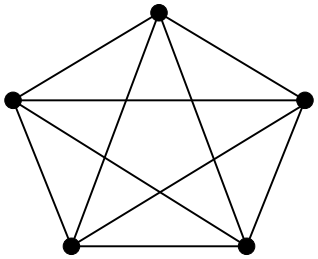
Lemma 9.1.3. *The sum of the degrees of the vertices in a graph equals twice the number of edges.*

Proof. Every edge contributes two to the sum of the degrees, one for each of its endpoints. ■

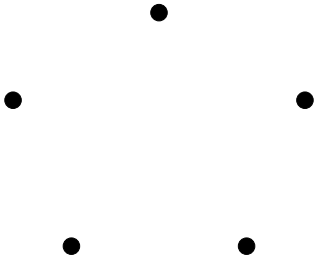
Lemma 9.1.3 is sometimes called the *Handshake Lemma*: if we total up the number of people each person at a party shakes hands with, the total will be twice the number of handshakes that occurred.

9.1.4 Some Common Graphs

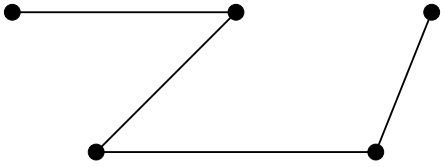
Some graphs come up so frequently that they have names. The *complete graph* on n vertices, also called K_n, has an edge between every two vertices. Here is K_5:



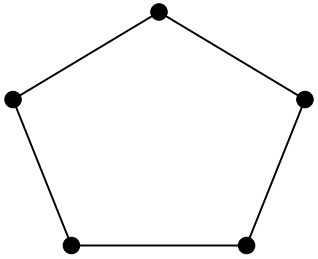
The *empty graph* has no edges at all. Here is the empty graph on 5 vertices:



Another 5 vertex graph is L_4, the *line graph* of length four:

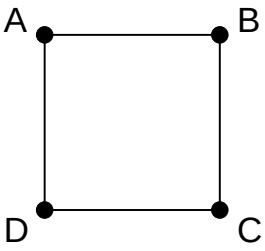


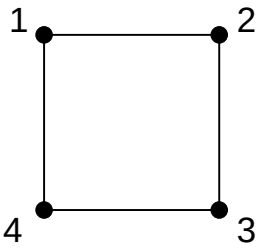
And here is C_5, a *simple cycle* with 5 vertices:



9.1.5 Isomorphism

Two graphs that look the same might actually be different in a formal sense. For example, the two graphs below are both simple cycles with 4 vertices:





But one graph has vertex set $\{A, B, C, D\}$ while the other has vertex set $\{1, 2, 3, 4\}$. If so, then the graphs are different mathematical objects, strictly speaking. But this is a frustrating distinction; the graphs *look the same*!

Fortunately, we can neatly capture the idea of "looks the same" by adapting Definition 6.2.1 of isomorphism of digraphs to handle simple graphs.

Definition 9.1.4. If G_1 is a graph with vertices, V_1, and edges, E_1, and likewise for G_2, then G_1 is *isomorphic* to G_2 iff there exists a **bijection**, $f : V_1 \to V_2$, such that for every pair of vertices $u, v \in V_1$:

$$u\text{—}v \in E_1 \quad \text{iff} \quad f(u)\text{—}f(v) \in E_2.$$

The function f is called an *isomorphism* between G_1 and G_2.

For example, here is an isomorphism between vertices in the two graphs above:

A corresponds to 1	B corresponds to 2
D corresponds to 4	C corresponds to 3.

You can check that there is an edge between two vertices in the graph on the left if and only if there is an edge between the two corresponding vertices in the graph on the right.

Two isomorphic graphs may be drawn very differently. For example, here are two different ways of drawing C_5:

Isomorphism preserves the connection properties of a graph, abstracting out what the vertices are called, what they are made out of, or where they appear in a drawing of the graph. More precisely, a property of a graph is said to be *preserved under isomorphism* if whenever G has that property, every graph isomorphic to G also has that property. For example, since an isomorphism is a bijection between sets of vertices, isomorphic graphs must have the same number of vertices. What's more, if f is a graph isomorphism that maps a vertex, v, of one graph to the vertex, $f(v)$, of an isomorphic graph, then by definition of isomorphism, every vertex adjacent to v in the first graph will be mapped by f to a vertex adjacent to $f(v)$ in the isomorphic graph. That is, v and $f(v)$ will have the same degree. So if one graph has a vertex of degree 4 and another does not, then they can't be isomorphic.

In fact, they can't be isomorphic if the number of degree 4 vertices in each of the graphs is not the same.

Looking for preserved properties can make it easy to determine that two graphs are not isomorphic, or to actually find an isomorphism between them, if there is one. In practice, it's frequently easy to decide whether two graphs are isomorphic. However, no one has yet found a *general* procedure for determining whether two graphs are isomorphic that is *guaranteed* to run much faster than an exhaustive (and exhausting) search through all possible bijections between their vertices.

Having an efficient procedure to detect isomorphic graphs would, for example, make it easy to search for a particular molecule in a database given the molecular bonds. On the other hand, knowing there is no such efficient procedure would also be valuable: secure protocols for encryption and remote authentication can be built on the hypothesis that graph isomorphism is computationally exhausting.

9.1.6 Problems

Class Problems

Problem 9.1. (a) Prove that in every graph, there are an even number of vertices of odd degree.

Hint: The Handshaking Lemma 9.1.3.

(b) Conclude that at a party where some people shake hands, the number of people who shake hands an odd number of times is an even number.

(c) Call a sequence of two or more different people at the party a *handshake sequence* if, except for the last person, each person in the sequence has shaken hands with the next person in the sequence.

Suppose George was at the party and has shaken hands with an odd number of people. Explain why, starting with George, there must be a handshake sequence ending with a different person who has shaken an odd number of hands.

Hint: Just look at the people at the ends of handshake sequences that start with George.

Problem 9.2.
For each of the following pairs of graphs, either define an isomorphism between them, or prove that there is none. (We write ab as shorthand for a—b.)

(a)

$$G_1 \text{ with } V_1 = \{1,2,3,4,5,6\},\ E_1 = \{12, 23, 34, 14, 15, 35, 45\}$$
$$G_2 \text{ with } V_2 = \{1,2,3,4,5,6\},\ E_2 = \{12, 23, 34, 45, 51, 24, 25\}$$

(b)

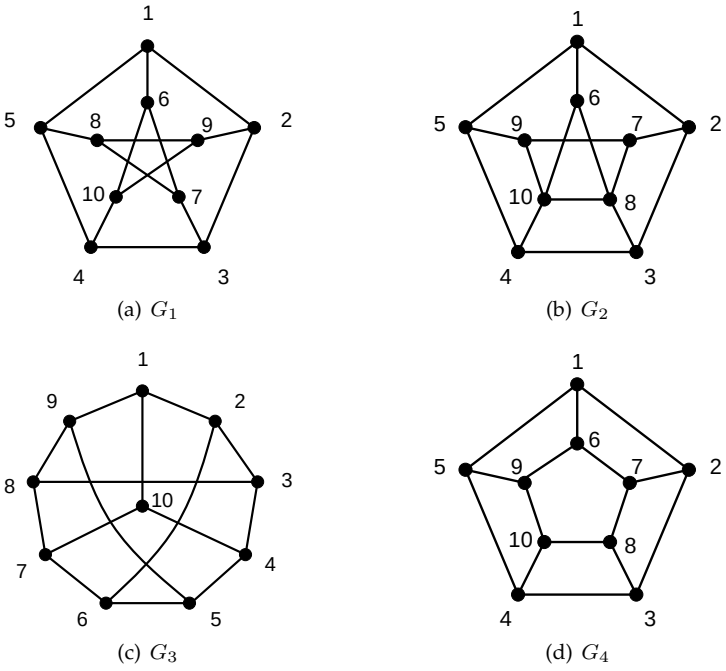
$$G_3 \text{ with } V_3 = \{1,2,3,4,5,6\}\,,\ E_3 = \{12,23,34,14,45,56,26\}$$
$$G_4 \text{ with } V_4 = \{a,b,c,d,e,f\}\,,\ E_4 = \{ab,bc,cd,de,ae,ef,cf\}$$

(c)

$$G_5 \text{ with } V_5 = \{a,b,c,d,e,f,g,h\}\,,\ E_5 = \{ab,bc,cd,ad,ef,fg,gh,he,dh,bf\}$$
$$G_6 \text{ with } V_6 = \{s,t,u,v,w,x,y,z\}\,,\ E_6 = \{st,tu,uv,sv,wx,xy,yz,wz,sw,vz\}$$

Homework Problems

Problem 9.3.
Determine which among the four graphs pictured in the Figures are isomorphic. If two of these graphs are isomorphic, describe an isomorphism between them. If they are not, give a property that is preserved under isomorphism such that one graph has the property, but the other does not. For at least one of the properties you choose, *prove* that it is indeed preserved under isomorphism (you only need prove one of them).



(a) G_1 (b) G_2

(c) G_3 (d) G_4

Figure 9.2: Which graphs are isomorphic?

Problem 9.4. (a) For any vertex, v, in a graph, let $N(v)$ be the set of *neighbors* of v, namely, the vertices adjacent to v:

$$N(v) ::= \{u \mid u\text{—}v \text{ is an edge of the graph}\}.$$

Suppose f is an isomorphism from graph G to graph H. Prove that $f(N(v)) = N(f(v))$.

Your proof should follow by simple reasoning using the definitions of isomorphism and neighbors —no pictures or handwaving.

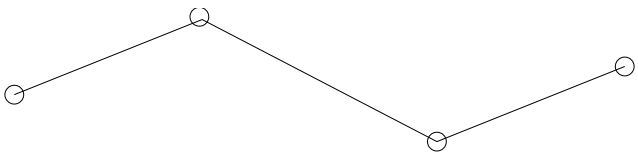
Hint: Prove by a chain of iff's that

$$h \in N(f(v)) \quad \text{iff} \quad h \in f(N(v))$$

for every $h \in V_H$. Use the fact that $h = f(u)$ for some $u \in V_G$.

(b) Conclude that if G and H are isomorphic graphs, then for each $k \in \mathbb{N}$, they have the same number of degree k vertices.

Problem 9.5.
Let's say that a graph has "two ends" if it has exactly two vertices of degree 1 and all its other vertices have degree 2. For example, here is one such graph:



(a) A *line graph* is a graph whose vertices can be listed in a sequence with edges between consecutive vertices only. So the two-ended graph above is also a line graph of length 4.

Prove that the following theorem is false by drawing a counterexample.
False Theorem. *Every two-ended graph is a line graph.*

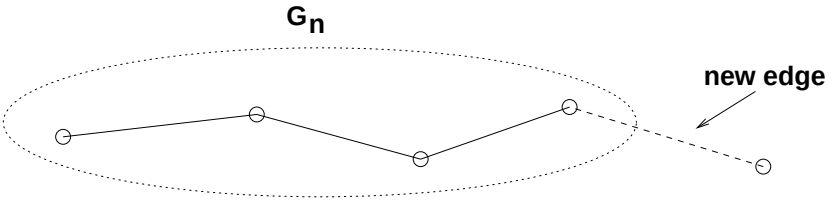
(b) Point out the first erroneous statement in the following alleged proof of the false theorem. Describe the error as best you can.

False proof. We use induction. The induction hypothesis is that every two-ended graph with n edges is a path.

Base case ($n = 1$): The only two-ended graph with a single edge consists of two vertices joined by an edge:

Sure enough, this is a line graph.

Inductive case: We assume that the induction hypothesis holds for some $n \geq 1$ and prove that it holds for $n + 1$. Let G_n be any two-ended graph with n edges. By the induction assumption, G_n is a line graph. Now suppose that we create a two-ended graph G_{n+1} by adding one more edge to G_n. This can be done in only one way: the new edge must join an endpoint of G_n to a new vertex; otherwise, G_{n+1} would not be two-ended.

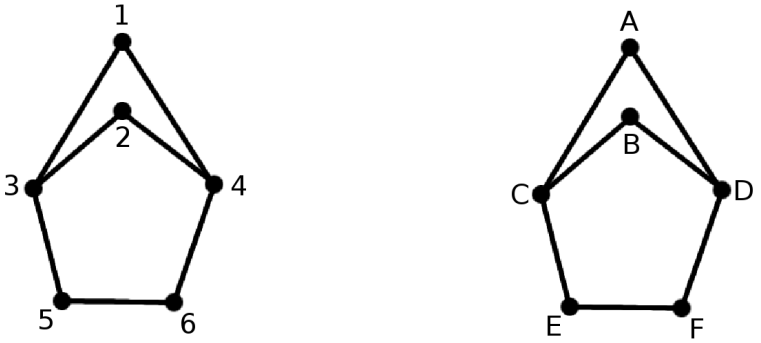


Clearly, G_{n+1} is also a line graph. Therefore, the induction hypothesis holds for all graphs with $n + 1$ edges, which completes the proof by induction.

■

Exam Problems

Problem 9.6.
There are four isomorphisms between these two graphs. List them.



Problem 9.7.
A researcher analyzing data on heterosexual sexual behavior in a group of m males and f females found that within the group, the male average number of female partners was 10% larger that the female average number of male partners.

(a) Circle all of the assertions below that are implied by the above information on average numbers of partners:

(i) males exaggerate their number of female partners

(ii) $m = (9/10)f$

(iii) $m = (10/11)f$

(iv) $m = (11/10)f$

(v) there cannot be a perfect matching with each male matched to one of his female partners

(vi) there cannot be a perfect matching with each female matched to one of her male partners

(b) The data shows that approximately 20% of the females were virgins, while only 5% of the males were. The researcher wonders how excluding virgins from the population would change the averages. If he knew graph theory, the researcher would realize that the nonvirgin male average number of partners will be $x(f/m)$ times the nonvirgin female average number of partners. What is x?

9.2 The Stable Marriage Problem

Okay, frequent public reference to derived variables may not help your mating prospects. But they can help with the analysis!

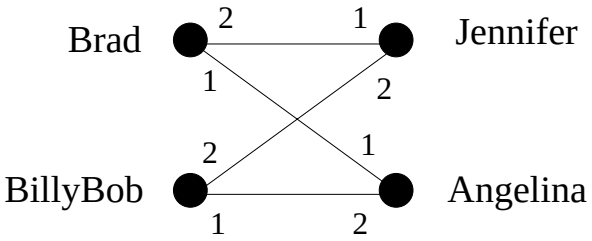
9.2.1 The Problem

Suppose there are a bunch of boys and an equal number of girls that we want to marry off. Each boy has his personal preferences about the girls —in fact, we assume he has a complete list of all the girls ranked according to his preferences, with no ties. Likewise, each girl has a ranked list of all of the boys.

The preferences don't have to be symmetric. That is, Jennifer might like Brad best, but Brad doesn't necessarily like Jennifer best. The goal is to marry off boys and girls: every boy must marry exactly one girl and vice-versa—no polygamy. In mathematical terms, we want the mapping from boys to their wives to be a bijection or *perfect matching*. We'll just call this a "matching," for short.

Here's the difficulty: suppose *every* boy likes Angelina best, and every girl likes Brad best, but Brad and Angelina are married to other people, say Jennifer and Billy Bob. Now *Brad and Angelina prefer each other to their spouses*, which puts their marriages at risk: pretty soon, they're likely to start spending late nights is study sessions together :-) .

This situation is illustrated in the following diagram where the digits "1" and "2" near a boy shows which of the two girls he ranks first and which second, and similarly for the girls:



More generally, in any matching, a boy and girl who are not married to each other and who like each other better than their spouses, is called a *rogue couple*. In the situation above, Brad and Angelina would be a rogue couple.

Having a rogue couple is not a good thing, since it threatens the stability of the marriages. On the other hand, if there are no rogue couples, then for any boy and girl who are not married to each other, at least one likes their spouse better than the other, and so won't be tempted to start an affair.

Definition 9.2.1. A *stable matching* is a matching with no rogue couples.

The question is, given everybody's preferences, how do you find a stable set of marriages? In the example consisting solely of the four people above, we could let Brad and Angelina both have their first choices by marrying each other. Now neither Brad nor Angelina prefers anybody else to their spouse, so neither will be in a rogue couple. This leaves Jen not-so-happily married to Billy Bob, but neither Jen nor Billy Bob can entice somebody else to marry them.

It is something of a surprise that there always is a stable matching among a group of boys and girls, but there is, and we'll shortly explain why. The surprise springs in part from considering the apparently similar "buddy" matching problem. That is, if people can be paired off as buddies, regardless of gender, then a stable matching *may not* be possible. For example, Figure 9.3 shows a situation with a love triangle and a fourth person who is everyone's last choice. In this figure Mergatoid's preferences aren't shown because they don't even matter.

Let's see why there is no stable matching:

Lemma. *There is no stable buddy matching among the four people in Figure 9.3.*

Proof. We'll prove this by contradiction.

Assume, for the purposes of contradiction, that there is a stable matching. Then there are two members of the love triangle that are matched. Since preferences in the triangle are symmetric, we may assume in particular, that Robin and Alex are matched. Then the other pair must be Bobby-Joe matched with Mergatoid.

But then there is a rogue couple: Alex likes Bobby-Joe best, and Bobby-Joe prefers Alex to his buddy Mergatoid. That is, Alex and Bobby-Joe are a rogue couple, contradicting the assumed stability of the matching. ■

So getting a stable *buddy* matching may not only be hard, it may be impossible. But when boys are only allowed to marry girls, and vice versa, then it turns out that a stable matching is not hard to find.

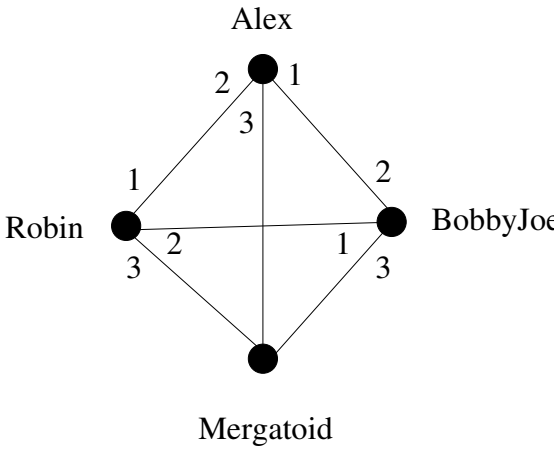


Figure 9.3: Some preferences with no stable buddy matching.

9.2.2 The Mating Ritual

The procedure for finding a stable matching involves a *Mating Ritual* that takes place over several days. The following events happen each day:

Morning: Each girl stands on her balcony. Each boy stands under the balcony of his favorite among the girls on his list, and he serenades her. If a boy has no girls left on his list, he stays home and does his 6.042 homework.

Afternoon: Each girl who has one or more suitors serenading her, says to her favorite among them, "We might get engaged. Come back tomorrow." To the other suitors, she says, "No. I will never marry you! Take a hike!"

Evening: Any boy who is told by a girl to take a hike, crosses that girl off his list.

Termination condition: When every girl has at most one suitor, the ritual ends with each girl marrying her suitor, if she has one.

There are a number of facts about this Mating Ritual that we would like to prove:

- The Ritual has a last day.
- Everybody ends up married.
- The resulting marriages are stable.

9.2.3 A State Machine Model

Before we can prove anything, we should have clear mathematical definitions of what we're talking about. In this section we sketch how to define a rigorous state machine model of the Marriage Problem.

So let's begin by formally defining the problem.

Definition 9.2.2. A *Marriage Problem* consists of two disjoint sets of the same finite size, called the-Boys and the-Girls. The members of the-Boys are called *boys*, and members of the-Girls are called *girls*. For each boy, B, there is a strict total order, $<_B$, on the-Girls, and for each girl, G, there is a strict total order, $<_G$, on the-Boys. If $G_1 <_B G_2$ we say B *prefers* girl G_2 to girl G_1. Similarly, if $B_1 <_G B_2$ we say G *prefers* boy B_2 to boy B_1.

A *marriage assignment* or *perfect matching* is a bijection, w : the-Boys $\to$ the-Girls. If $B \in$ the-Boys, then $w(B)$ is called B's *wife* in the assignment, and if $G \in$ the-Girls, then $w^{-1}(G)$ is called G's *husband*. A *rogue couple* is a boy, B, and a girl, G, such that B prefers G to his wife, and G prefers B to her husband. An assignment is *stable* if it has no rogue couples. A *solution* to a marriage problem is a stable perfect matching.

To model the Mating Ritual with a state machine, we make a key observation: to determine what happens on any day of the Ritual, all we need to know is which girls are still on which boys' lists on the morning of that day. So we define a state to be some mathematical data structure providing this information. For example, we could define a state to be the "still-has-on-his-list" relation, R, between boys and girls, where $B\ R\ G$ means girl G is still on boy B's list.

We start the Mating Ritual with no girls crossed off. That is, the start state is the *complete bipartite* relation in which every boy is related to every girl.

According to the Mating Ritual, on any given morning, a boy will *serenade* the girl he most prefers among those he has not as yet crossed out. Mathematically, the girl he is serenading is just the maximum among the girls on B's list, ordered by $<_B$. (If the list is empty, he's not serenading anybody.) A girl's *favorite* is just the maximum, under her preference ordering, among the boys serenading her.

Continuing in this way, we could mathematically specify a precise Mating Ritual state machine, but we won't bother. The intended behavior of the Mating Ritual is clear enough that we don't gain much by giving a formal state machine, so we stick to a more memorable description in terms of boys, girls, and their preferences. The point is, though, that it's not hard to define everything using basic mathematical data structures like sets, functions, and relations, if need be.

9.2.4 There is a Marriage Day

It's easy to see why the Mating Ritual has a terminal day when people finally get married. Every day on which the ritual hasn't terminated, at least one boy crosses a girl off his list. (If the ritual hasn't terminated, there must be some girl serenaded by at least two boys, and at least one of them will have to cross her off his list). So starting with n boys and n girls, each of the n boys' lists initially has n girls on it, for a total of n^2 list entries. Since no girl ever gets added to a list, the total number of entries on the lists decreases every day that the Ritual continues, and so the Ritual can continue for at most n^2 days.

9.2.5 They All Live Happily Every After...

We still have to prove that the Mating Ritual leaves everyone in a stable marriage. To do this, we note one very useful fact about the Ritual: if a girl has a favorite boy suitor on some morning of the Ritual, then that favorite suitor will still be serenading her the next morning —because his list won't have changed. So she is sure to have today's favorite boy among her suitors tomorrow. That means she will be able to choose a favorite suitor tomorrow who is at least as desirable to her as today's favorite. So day by day, her favorite suitor can stay the same or get better, never worse. In others words, a girl's favorite is a weakly increasing variable with respect to her preference order on the boys.

Now we can verify the Mating Ritual using a simple invariant predicate, P, that captures what's going on:

> For every girl, G, and every boy, B, if G is crossed off B's list, then G has a suitor whom she prefers over B.

Why is P invariant? Well, we know that G's favorite tomorrow will be at least as desirable to her as her favorite today, and since her favorite today is more desirable than B, tomorrow's favorite will be too.

Notice that P also holds on the first day, since every girl is on every list. So by the Invariant Theorem, we know that P holds on every day that the Mating Ritual runs. Knowing the invariant holds when the Mating Ritual terminates will let us complete the proofs.

Theorem 9.2.3. *Everyone is married by the Mating Ritual.*

Proof. Suppose, for the sake of contradiction, that it is the last day of the Mating Ritual and some boy does not get married. Then he can't be serenading anybody, and so his list must be empty. So by invariant P, every girl has a favorite boy whom she prefers to that boy. In particular, every girl has a favorite boy whom she marries on the last day. So all the girls are married. What's more there is no bigamy: a boy only serenades one girl, so no two girls have the same favorite.

But there are the same number of girls as boys, so all the boys must be married too. ■

Theorem 9.2.4. *The Mating Ritual produces a stable matching.*

Proof. Let Brad be some boy and Jen be any girl that he is *not* married to on the last day of the Mating Ritual. We claim that Brad and Jen are not a rogue couple. Since Brad is an arbitrary boy, it follows that no boy is part of a rogue couple. Hence the marriages on the last day are stable.

To prove the claim, we consider two cases:

Case 1. Jen is not on Brad's list. Then by invariant P, we know that Jen prefers her husband to Brad. So she's not going to run off with Brad: the claim holds in this case.

Case 2. Otherwise, Jen is on Brad's list. But since Brad is not married to Jen, he must be choosing to serenade his wife instead of Jen, so he must prefer his wife. So he's not going to run off with Jen: the claim also holds in this case. ■

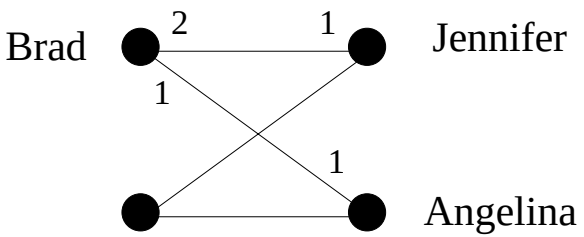
9.2.6 ...Especially the Boys

Who is favored by the Mating Ritual, the boys or the girls? The girls seem to have all the power: they stand on their balconies choosing the finest among their suitors and spurning the rest. What's more, we know their suitors can only change for the better as the Ritual progresses. Similarly, a boy keeps serenading the girl he most prefers among those on his list until he must cross her off, at which point he serenades the next most preferred girl on his list. So from the boy's point of view, the girl he is serenading can only change for the worse. Sounds like a good deal for the girls.

But it's not! The fact is that from the beginning, the boys are serenading their first choice girl, and the desirability of the girl being serenaded decreases only enough to give the boy his most desirable possible spouse. The mating algorithm actually does as well as possible for all the boys and does the worst possible job for the girls.

To explain all this we need some definitions. Let's begin by observing that while the mating algorithm produces one stable matching, there may be other stable matchings among the same set of boys and girls. For example, reversing the roles of boys and girls will often yield a different stable matching among them.

But some spouses might be out of the question in all possible stable matchings. For example, Brad is just not in the realm of possibility for Jennifer, since if you ever pair them, Brad and Angelina will form a rogue couple; here's a picture:



Definition 9.2.5. Given any marriage problem, one person is in another person's *realm of possible spouses* if there is a stable matching in which the two people are married. A person's *optimal spouse* is their most preferred person within their realm of possibility. A person's *pessimal spouse* is their least preferred person in their realm of possibility.

Everybody has an optimal and a pessimal spouse, since we know there is at least one stable matching, namely the one produced by the Mating Ritual. Now here is the shocking truth about the Mating Ritual:

Theorem 9.2.6. *The Mating Ritual marries every boy to his optimal spouse.*

Proof. Assume for the purpose of contradiction that some boy does not get his optimal girl. There must have been a day when he crossed off his optimal girl —otherwise he would still be serenading her or some even more desirable girl.

By the Well Ordering Principle, there must be a *first* day when a boy, call him "Keith," crosses off his optimal girl, Nicole.

According to the rules of the Ritual, Keith crosses off Nicole because Nicole has a favorite suitor, Tom, and

Nicole prefers Tom to Keith (*)

(remember, this is a proof by contradiction :-)).

Now since this is the first day an optimal girl gets crossed off, we know Tom hasn't crossed off his optimal girl. So

Tom ranks Nicole at least as high as his optimal girl. (**)

By the definition of an optimal girl, there must be some stable set of marriages in which Keith gets his optimal girl, Nicole. But then the preferences given in (*) and (**) imply that Nicole and Tom are a rogue couple within this supposedly stable set of marriages (think about it). This is a contradiction. ■

Theorem 9.2.7. *The Mating Ritual marries every girl to her pessimal spouse.*

Proof. Say Nicole and Keith marry each other as a result of the Mating Ritual. By the previous Theorem 9.2.6, Nicole is Keith's optimal spouse, and so in any stable set of marriages,

Keith rates Nicole at least as high as his spouse. (+)

Now suppose for the purpose of contradiction that there is another stable set of marriages where Nicole does worse than Keith. That is, Nicole is married to Tom, and

Nicole prefers Keith to Tom (++)

Then in this stable set of marriages where Nicole is married to Tom, (+) and (++) imply that Nicole and Keith are a rogue couple, contradicting stability. We conclude that Nicole cannot do worse than Keith. ■

9.2.7 Applications

Not surprisingly, a stable matching procedure is used by at least one large dating agency. But although "boy-girl-marriage" terminology is traditional and makes some of the definitions easier to remember (we hope without offending anyone), solutions to the Stable Marriage Problem are widely useful.

The Mating Ritual was first announced in a paper by D. Gale and L.S. Shapley in 1962, but ten years before the Gale-Shapley paper was appeared, and unknown by them, the Ritual was being used to assign residents to hospitals by the National Resident Matching Program (NRMP). The NRMP has, since the turn of the twentieth century, assigned each year's pool of medical school graduates to hospital residencies (formerly called "internships") with hospitals and graduates playing the roles of boys and girls. (In this case there may be multiple boys married to one

girl, but there's an easy way to use the Ritual in this situation (see Problem 9.12). Before the Ritual was adopted, there were chronic disruptions and awkward countermeasures taken to preserve assignments of graduates to residencies. The Ritual resolved these problems so successfully, that it was used essentially without change at least through 1989.[2]

The internet infrastructure company, Akamai, also uses a variation of the Gale-Shapley procedure to assign web traffic to servers. In the early days, Akamai used other combinatorial optimization algorithms that got to be too slow as the number of servers (over 20,000 in 2010) reference needed and traffic increased. Akamai switched to Gale-Shapley since it is fast and can be run in a distributed manner. In this case, the web traffic corresponds to the boys and the web servers to the girls. The servers have preferences based on latency and packet loss; the traffic has preferences based on the cost of bandwidth.

9.2.8 Problems

Practice Problems

Problem 9.8.
Four Students want separate assignments to four VI-A Companies. Here are their preference rankings:

Student	Companies
Albert:	HP, Bellcore, AT&T, Draper
Rich:	AT&T, Bellcore, Draper, HP
Megumi:	HP, Draper, AT&T, Bellcore
Justin:	Draper, AT&T, Bellcore, HP

Company	Students
AT&T:	Justin, Albert, Megumi, Rich
Bellcore:	Megumi, Rich, Albert, Justin
HP:	Justin, Megumi, Albert, Rich
Draper:	Rich, Justin, Megumi, Albert

(a) Use the Mating Ritual to find *two* stable assignments of Students to Companies.

(b) Describe a simple procedure to determine whether any given stable marriage problem has a unique solution, that is, only one possible stable matching.

Problem 9.9.
Suppose that Harry is one of the boys and Alice is one of the girls in the *Mating Ritual*. Which of the properties below are preserved invariants? Why?

[2]Much more about the Stable Marriage Problem can be found in the very readable mathematical monograph by Dan Gusfield and Robert W. Irving, The Stable Marriage Problem: Structure and Algorithms, MIT Press, Cambridge, Massachusetts, 1989, 240 pp.

a. Alice is the only girl on Harry's list.

b. There is a girl who does not have any boys serenading her.

c. If Alice is not on Harry's list, then Alice has a suitor that she prefers to Harry.

d. Alice is crossed off Harry's list and Harry prefers Alice to anyone he is serenading.

e. If Alice is on Harry's list, then she prefers to Harry to any suitor she has.

Class Problems

Problem 9.10.
A preserved invariant of the Mating ritual is:

> For every girl, G, and every boy, B, if G is crossed off B's list, then G has a favorite suitor and she prefers him over B.

Use the invariant to prove that the Mating Algorithm produces stable marriages. (Don't look up the proof in the Notes or slides.)

Problem 9.11.
Consider a stable marriage problem with 4 boys and 4 girls and the following partial information about their preferences:

B1:	G1	G2	–	–
B2:	G2	G1	–	–
B3:	–	–	G4	G3
B4:	–	–	G3	G4
G1:	B2	B1	–	–
G2:	B1	B2	–	–
G3:	–	–	B3	B4
G4:	–	–	B4	B3

(a) Verify that

$$(B1, G1), (B2, G2), (B3, G3), (B4, G4)$$

will be a stable matching whatever the unspecified preferences may be.

(b) Explain why the stable matching above is neither boy-optimal nor boy-pessimal and so will not be an outcome of the Mating Ritual.

(c) Describe how to define a set of marriage preferences among n boys and n girls which have at least $2^{n/2}$ stable assignments.

Hint: Arrange the boys into a list of $n/2$ pairs, and likewise arrange the girls into a list of $n/2$ pairs of girls. Choose preferences so that the kth pair of boys ranks

the kth pair of girls just below the previous pairs of girls, and likewise for the kth pair of girls. Within the kth pairs, make sure each boy's first choice girl in the pair prefers the other boy in the pair.

Homework Problems

Problem 9.12.
The most famous application of stable matching was in assigning graduating medical students to hospital residencies. Each hospital has a preference ranking of students and each student has a preference order of hospitals, but unlike the setup in the notes where there are an equal number of boys and girls and monogamous marriages, hospitals generally have differing numbers of available residencies, and the total number of residencies may not equal the number of graduating students. Modify the definition of stable matching so it applies in this situation, and explain how to modify the Mating Ritual so it yields stable assignments of students to residencies. No proof is required.

Problem 9.13.
Give an example of a stable matching between 3 boys and 3 girls where no person gets their first choice. Briefly explain why your matching is stable.

Problem 9.14.
In a stable matching between n boys and girls produced by the Mating Ritual, call a person *lucky* if they are matched up with one of their $\lceil n/2 \rceil$ top choices. We will prove:

Theorem. *There must be at least one lucky person.*

To prove this, define the following derived variables for the Mating Ritual:

$q(B) = j$, where j is the rank of the girl that boy B is courting. That is to say, boy B is always courting the jth girl on his list.

$r(G)$ is the number of boys that girl G has rejected.

(a) Let

$$S ::= \sum_{B \in \text{the-Boys}} q(B) - \sum_{G \in \text{the-Girls}} r(G). \tag{9.1}$$

Show that S remains the same from one day to the next in the Mating Ritual.

(b) Prove the Theorem above. (You may assume for simplicity that n is even.)

Hint: A girl is sure to be lucky if she has rejected half the boys.

9.3 Connectedness

9.3.1 Paths and Simple Cycles

Paths in simple graphs are esentially the same as paths in digraphs. We just modify the digraph definitions using undirected edges instead of directed ones. For example, the formal definition of a path in a simple graph is a virtually that same as Definition 7.1.1 of paths in digraphs:

Definition 9.3.1. A *path* in a graph, G, is a sequence of $k \geq 0$ vertices

$$v_0, \ldots, v_k$$

such that v_i—v_{i+1} is an edge of G for all i where $0 \leq i < k$. The path is said to *start* at v_0, to *end* at v_k, and the *length* of the path is defined to be k.

An edge, u—v, is *traversed* n times by the path if there are n different values of i such that v_i—$v_{i+1} = u$—v. The path is *simple*[3] iff all the v_i's are different, that is, if $i \neq j$ implies $v_i \neq v_j$.

For example, the graph in Figure 9.4 has a length 6 simple path A,B,C,D,E,F,G. This is the longest simple path in the graph.

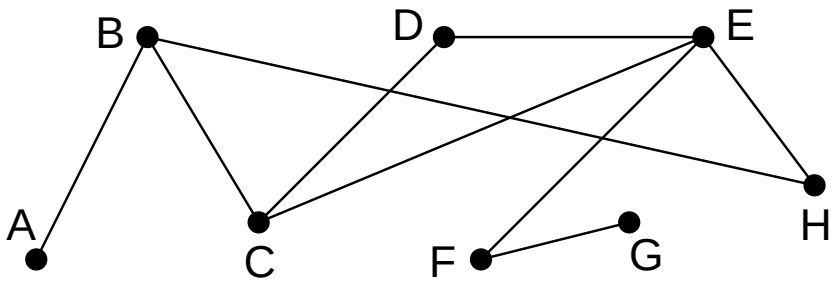


Figure 9.4: *A graph with 3 simple cycles.*

As in digraphs, the length of a path is the total number of times it traverses edges, which is *one less* than its length as a sequence of vertices. For example, the length 6 path A,B,C,D,E,F,G is actually a sequence of seven vertices.

A *cycle* can be described by a path that begins and ends with the same vertex. For example, B,C,D,E,C,B is a cycle in the graph in Figure 9.4. This path suggests that the cycle begins and ends at vertex B, but a cycle isn't intended to have a

[3]Heads up: what we call "paths" are commonly referred to in graph theory texts as "walks," and simple paths are referred to as just "paths". Likewise, what we will call *cycles* and *simple cycles* are commonly called "closed walks" and just "cycles".

beginning and end, and can be described by *any* of the paths that go around it. For example, D,E,C,B,C,D describes this same cycle as though it started and ended at D, and D,C,B,C,E,D describes the same cycle as though it started and ended at D but went in the opposite direction. (By convention, a single vertex is a length 0 cycle beginning and ending at the vertex.)

All the paths that describe the same cycle have the same length which is defined to be the *length of the cycle*. (Note that this implies that going around the same cycle twice is considered to be different than going around it once.)

A *simple* cycle is a cycle that doesn't cross or backtrack on itself. For example, the graph in Figure 9.4 has three simple cycles B,H,E,C,B and C,D,E,C and B,C,D,E,H,B. More precisely, a simple cycle is a cycle that can be described by a path of length at least three whose vertices are all different except for the beginning and end vertices. So in contrast to simple *paths*, the length of a simple *cycle* is the *same* as the number of distinct vertices that appear in it.

From now on we'll stop being picky about distinguishing a cycle from a path that describes it, and we'll just refer to the path as a cycle. [4]

Simple cycles are especially important, so we will give a proper definition of them. Namely, we'll define a simple cycle in G to be a *subgraph* of G that looks like a cycle that doesn't cross itself. Formally:

Definition 9.3.2. A *subgraph*, G', of a graph, G, is a graph whose vertices, V', are a subset of the vertices of G and whose edges are a subset of the edges of G.

Notice that since a subgraph is itself a graph, the endpoints of every edge of G' must be vertices in V'.

Definition 9.3.3. For $n \geq 3$, let C_n be the graph with vertices $1, \ldots, n$ and edges

$$1\text{—}2, \quad 2\text{—}3, \quad \ldots, \quad (n-1)\text{—}n, \quad n\text{—}1.$$

A graph is a *simple cycle* of length n iff it is isomorphic to C_n for some $n \geq 3$. A *simple cycle of a graph*, G, is a subgraph of G that is a simple cycle.

This definition formally captures the idea that simple cycles don't have direction or beginnings or ends.

9.3.2 Connected Components

Definition 9.3.4. Two vertices in a graph are said to be *connected* when there is a path that begins at one and ends at the other. By convention, every vertex is considered to be connected to itself by a path of length zero.

The diagram in Figure 9.5 looks like a picture of three graphs, but is intended to be a picture of *one* graph. This graph consists of three pieces (subgraphs). Each piece by itself is connected, but there are no paths between vertices in different pieces.

[4] Technically speaking, we haven't ever defined what a cycle *is*, only how to describe it with paths. But we won't need an abstract definition of cycle, since all that matters about a cycle is which paths describe it.

Figure 9.5: *One graph with 3 connected components.*

Definition 9.3.5. A graph is said to be *connected* when every pair of vertices are connected.

These connected pieces of a graph are called its *connected components*. A rigorous definition is easy: a connected component is the set of all the vertices connected to some single vertex. So a graph is connected iff it has exactly one connected component. The empty graph on n vertices has n connected components.

9.3.3 How Well Connected?

If we think of a graph as modelling cables in a telephone network, or oil pipelines, or electrical power lines, then we not only want connectivity, but we want connectivity that survives component failure. A graph is called *k-edge connected* if it takes at least k "edge-failures" to disconnect it. More precisely:

Definition 9.3.6. Two vertices in a graph are *k-edge connected* if they remain connected in every subgraph obtained by deleting $k-1$ edges. A graph with at least two vertices is k-edge connected[5] if every two of its vertices are k-edge connected.

So 1-edge connected is the same as connected for both vertices and graphs. Another way to say that a graph is k-edge connected is that every subgraph obtained from it by deleting at most $k-1$ edges is connected. For example, in the graph in Figure 9.4, vertices B and E are 2-edge connected, G and E are 1-edge connected, and no vertices are 3-edge connected. The graph as a whole is only 1-edge connected. More generally, any simple cycle is 2-edge connected, and the complete graph, K_n, is $(n-1)$-edge connected.

If two vertices are connected by k edge-disjoint paths (that is, no two paths traverse the same edge), then they are obviously k-edge connected. A fundamental

[5]The corresponding definition of connectedness based on deleting vertices rather than edges is common in Graph Theory texts and is usually simply called "k-connected" rather than "k-vertex connected."

fact, whose ingenious proof we omit, is Menger's theorem which confirms that the converse is also true: if two vertices are k-edge connected, then there are k edge-disjoint paths connecting them. It even takes some ingenuity to prove this for the case $k = 2$.

9.3.4 Connection by Simple Path

Where there's a path, there's a simple path. This is sort of obvious, but it's easy enough to prove rigorously using the Well Ordering Principle.

Lemma 9.3.7. *If vertex u is connected to vertex v in a graph, then there is a simple path from u to v.*

Proof. Since there is a path from u to v, there must, by the Well-ordering Principle, be a minimum length path from u to v. If the minimum length is zero or one, this minimum length path is itself a simple path from u to v. Otherwise, there is a minimum length path

$$v_0, v_1, \ldots, v_k$$

from $u = v_0$ to $v = v_k$ where $k \geq 2$. We claim this path must be simple. To prove the claim, suppose to the contrary that the path is not simple, that is, some vertex on the path occurs twice. This means that there are integers i, j such that $0 \leq i < j \leq k$ with $v_i = v_j$. Then deleting the subsequence

$$v_{i+1}, \ldots v_j$$

yields a strictly shorter path

$$v_0, v_1, \ldots, v_i, v_{j+1}, v_{j+2}, \ldots, v_k$$

from u to v, contradicting the minimality of the given path. ■

Actually, we proved something stronger:

Corollary 9.3.8. *For any path of length k in a graph, there is a simple path of length* at most *k with the same endpoints.*

9.3.5 The Minimum Number of Edges in a Connected Graph

The following theorem says that a graph with few edges must have many connected components.

Theorem 9.3.9. *Every graph with v vertices and e edges has at least $v - e$ connected components.*

Of course for Theorem 9.3.9 to be of any use, there must be fewer edges than vertices.

Proof. We use induction on the number of edges, e. Let $P(e)$ be the proposition that

for every v, every graph with v vertices and e edges has at least $v - e$ connected components.

Base case:($e = 0$). In a graph with 0 edges and v vertices, each vertex is itself a connected component, and so there are exactly $v = v - 0$ connected components. So $P(e)$ holds.

Inductive step: Now we assume that the induction hypothesis holds for every e-edge graph in order to prove that it holds for every $(e + 1)$-edge graph, where $e \geq 0$. Consider a graph, G, with $e + 1$ edges and k vertices. We want to prove that G has at least $v - (e + 1)$ connected components. To do this, remove an arbitrary edge a—b and call the resulting graph G'. By the induction assumption, G' has at least $v - e$ connected components. Now add back the edge a—b to obtain the original graph G. If a and b were in the same connected component of G', then G has the same connected components as G', so G has at least $v - e > v - (e + 1)$ components. Otherwise, if a and b were in different connected components of G', then these two components are merged into one in G, but all other components remain unchanged, reducing the number of components by 1. Therefore, G has at least $(v-e)-1 = v-(e+1)$ connected components. So in either case, $P(e+1)$ holds. This completes the Inductive step. The theorem now follows by induction. ■

Corollary 9.3.10. *Every connected graph with v vertices has at least $v - 1$ edges.*

A couple of points about the proof of Theorem 9.3.9 are worth noticing. First, we used induction on the number of edges in the graph. This is very common in proofs involving graphs, and so is induction on the number of vertices. When you're presented with a graph problem, these two approaches should be among the first you consider. The second point is more subtle. Notice that in the inductive step, we took an arbitrary $(n + 1)$-edge graph, threw out an edge so that we could apply the induction assumption, and then put the edge back. You'll see this shrink-down, grow-back process very often in the inductive steps of proofs related to graphs. This might seem like needless effort; why not start with an n-edge graph and add one more to get an $(n + 1)$-edge graph? That would work fine in this case, but opens the door to a nasty logical error called *buildup error*, illustrated in Problems 9.5 and 9.18. Always use shrink-down, grow-back arguments, and you'll never fall into this trap.

9.3.6 Problems

Class Problems

Problem 9.15.
The n-dimensional hypercube, H_n, is a graph whose vertices are the binary strings of length n. Two vertices are adjacent if and only if they differ in exactly 1 bit. For example, in H_3, vertices `111` and `011` are adjacent because they differ only in the first bit, while vertices `101` and `011` are not adjacent because they differ at both the first and second bits.

(a) Prove that it is impossible to find two spanning trees of H_3 that do not share some edge.

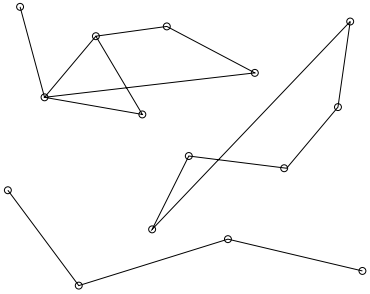
(b) Verify that for any two vertices $x \neq y$ of H_3, there are 3 paths from x to y in H_3, such that, besides x and y, no two of those paths have a vertex in common.

(c) Conclude that the connectivity of H_3 is 3.

(d) Try extending your reasoning to H_4. (In fact, the connectivity of H_n is n for all $n \geq 1$. A proof appears in the problem solution.)

Problem 9.16.
A set, M, of vertices of a graph is a *maximal connected* set if every pair of vertices in the set are connected, and any set of vertices properly containing M will contain two vertices that are not connected.

(a) What are the maximal connected subsets of the following (unconnected) graph?



(b) Explain the connection between maximal connected sets and connected components. Prove it.

Problem 9.17. (a) Prove that K_n is $(n-1)$-edge connected for $n > 1$.

Let M_n be a graph defined as follows: begin by taking n graphs with non-overlapping sets of vertices, where each of the n graphs is $(n-1)$-edge connected (they could be disjoint copies of K_n, for example). These will be subgraphs of M_n. Then pick n vertices, one from each subgraph, and add enough edges between pairs of picked vertices that the subgraph of the n picked vertices is also $(n-1)$-edge connected.

(b) Draw a picture of M_4.

(c) Explain why M_n is $(n-1)$-edge connected.

Problem 9.18.
Definition 9.3.5. A graph is *connected* iff there is a path between every pair of its vertices.

False Claim. *If every vertex in a graph has positive degree, then the graph is connected.*

(a) Prove that this Claim is indeed false by providing a counterexample.

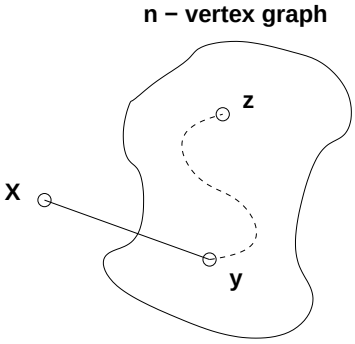
(b) Since the Claim is false, there must be an logical mistake in the following bogus proof. Pinpoint the *first* logical mistake (unjustified step) in the proof.

Bogus proof. We prove the Claim above by induction. Let $P(n)$ be the proposition that if every vertex in an n-vertex graph has positive degree, then the graph is connected.

Base cases: ($n \leq 2$). In a graph with 1 vertex, that vertex cannot have positive degree, so $P(1)$ holds vacuously.

$P(2)$ holds because there is only one graph with two vertices of positive degree, namely, the graph with an edge between the vertices, and this graph is connected.

Inductive step: We must show that $P(n)$ implies $P(n+1)$ for all $n \geq 2$. Consider an n-vertex graph in which every vertex has positive degree. By the assumption $P(n)$, this graph is connected; that is, there is a path between every pair of vertices. Now we add one more vertex x to obtain an $(n+1)$-vertex graph:



All that remains is to check that there is a path from x to every other vertex z. Since x has positive degree, there is an edge from x to some other vertex, y. Thus, we can obtain a path from x to z by going from x to y and then following the path from y to z. This proves $P(n+1)$.

By the principle of induction, $P(n)$ is true for all $n \geq 0$, which proves the Claim.

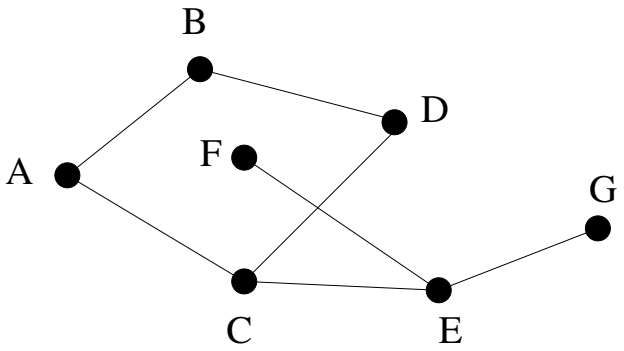
■

Homework Problems

Problem 9.19.
In this problem we'll consider some special cycles in graphs called *Euler circuits*, named after the famous mathematician Leonhard Euler. (Same Euler as for the constant $e \approx 2.718$ —he did a lot of stuff.)

Definition 9.3.11. An Euler circuit of a graph is a cycle which traverses every edge exactly once.

Does the graph in the following figure contain an Euler circuit?



Well, if it did, the edge (E, F) would need to be included. If the path does not start at F then at some point it traverses edge (E, F), and now it is stuck at F since F has no other edges incident to it and an Euler circuit can't traverse (E, F) twice. But then the path could not be a circuit. On the other hand, if the path starts at F, it must then go to E along (E, F), but now it cannot return to F. It again cannot be a circuit. This argument generalizes to show that if a graph has a vertex of degree 1, it cannot contain an Euler circuit.

So how do you tell in general whether a graph has an Euler circuit? At first glance this may seem like a daunting problem (the similar sounding problem of finding a cycle that touches every vertex exactly once is one of those million dollar NP-complete problems known as the *Traveling Salesman Problem*) —but it turns out to be easy.

(a) Show that if a graph has an Euler circuit, then the degree of each of its vertices is even.

In the remaining parts, we'll work out the converse: if the degree of every vertex of a connected finite graph is even, then it has an Euler circuit. To do this, let's define an Euler *path* to be a path that traverses each edge *at most* once.

(b) Suppose that an Euler path in a connected graph does not traverse every edge. Explain why there must be an untraversed edge that is incident to a vertex on the path.

In the remaining parts, let W be the *longest* Euler path in some finite, connected graph.

(c) Show that if W is a cycle, then it must be an Euler circuit.

Hint: part (b)

(d) Explain why all the edges incident to the end of W must already have been traversed by W.

(e) Show that if the end of W was not equal to the start of W, then the degree of the end would be odd.

Hint: part (d)

(f) Conclude that if every vertex of a finite, connected graph has even degree, then it has an Euler circuit.

Homework Problems

Problem 9.20.
An edge is said to *leave* a set of vertices if one end of the edge is in the set and the other end is not.

(a) An n-node graph is said to be *mangled* if there is an edge leaving every set of $\lfloor n/2 \rfloor$ or fewer vertices. Prove the following claim.

Claim. *Every mangled graph is connected.*

An n-node graph is said to be *tangled* if there is an edge leaving every set of $\lceil n/3 \rceil$ or fewer vertices.

(b) Draw a tangled graph that is not connected.

(c) Find the error in the proof of the following

False Claim. *Every tangled graph is connected.*

False proof. The proof is by strong induction on the number of vertices in the graph. Let $P(n)$ be the proposition that if an n-node graph is tangled, then it is connected. In the base case, $P(1)$ is true because the graph consisting of a single node is trivially connected.

For the inductive case, assume $n \geq 1$ and $P(1), \ldots, P(n)$ hold. We must prove $P(n+1)$, namely, that if an $(n+1)$-node graph is tangled, then it is connected.

So let G be a tangled, $(n+1)$-node graph. Choose $\lceil n/3 \rceil$ of the vertices and let G_1 be the tangled subgraph of G with these vertices and G_2 be the tangled subgraph with the rest of the vertices. Note that since $n \geq 1$, the graph G has a least two vertices, and so both G_1 and G_2 contain at least one vertex. Since G_1 and G_2 are tangled, we may assume by strong induction that both are connected. Also, since G is tangled, there is an edge leaving the vertices of G_1 which necessarily connects to a vertex of G_2. This means there is a path between any two vertices of G: a path within one subgraph if both vertices are in the same subgraph, and a path traversing the connecting edge if the vertices are in separate subgraphs. Therefore, the entire graph, G, is connected. This completes the proof of the inductive case, and the Claim follows by strong induction.

■

Problem 9.21.
Let G be the graph formed from C_{2n}, the simple cycle of length $2n$, by connecting every pair of vertices at maximum distance from each other in C_{2n} by an edge in G.

(a) Given two vertices of G find their distance in G.

(b) What is the *diameter* of G, that is, the largest distance between two vertices?

(c) Prove that the graph is not 4-connected.

(d) Prove that the graph is 3-connected.

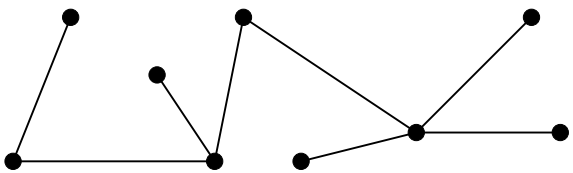
9.4 Trees

Trees are a fundamental data structure in computer science, and there are many kinds, such as rooted, ordered, and binary trees. In this section we focus on the purest kind of tree. Namely, we use the term *tree* to mean a connected graph without simple cycles.

A graph with no simple cycles is called *acyclic*; so trees are acyclic connected graphs.

9.4.1 Tree Properties

Here is an example of a tree:



A vertex of degree at most one is called a *leaf*. In this example, there are 5 leaves. Note that the only case where a tree can have a vertex of degree zero is a graph with a single vertex.

The graph shown above would no longer be a tree if any edge were removed, because it would no longer be connected. The graph would also not remain a tree if any edge were added between two of its vertices, because then it would contain

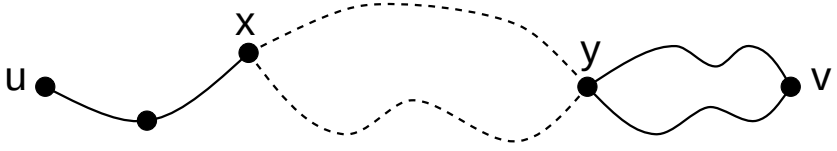
a simple cycle. Furthermore, note that there is a unique path between every pair of vertices. These features of the example tree are actually common to all trees.

Theorem 9.4.1. *Every tree has the following properties:*

1. *Any connected subgraph is a tree.*
2. *There is a unique simple path between every pair of vertices.*
3. *Adding an edge between two vertices creates a cycle.*
4. *Removing any edge disconnects the graph.*
5. *If it has at least two vertices, then it has at least two leaves.*
6. *The number of vertices is one larger than the number of edges.*

Proof. 1. A simple cycle in a subgraph is also a simple cycle in the whole graph, so any subgraph of an acyclic graph must also be acyclic. If the subgraph is also connected, then by definition, it is a tree.

2. There is at least one path, and hence one simple path, between every pair of vertices, because the graph is connected. Suppose that there are two different simple paths between vertices u and v. Beginning at u, let x be the first vertex where the paths diverge, and let y be the next vertex they share. Then there are two simple paths from x to y with no common edges, which defines a simple cycle. This is a contradiction, since trees are acyclic. Therefore, there is exactly one simple path between every pair of vertices.



3. An additional edge u—v together with the unique path between u and v forms a simple cycle.

4. Suppose that we remove edge u—v. Since the tree contained a unique path between u and v, that path must have been u—v. Therefore, when that edge is removed, no path remains, and so the graph is not connected.

5. Let $v_1, \ldots, v_m$ be the sequence of vertices on a longest simple path in the tree. Then $m \geq 2$, since a tree with two vertices must contain at least one edge. There cannot be an edge v_1—v_i for $2 < i \leq m$; otherwise, vertices $v_1, \ldots, v_i$ would from a simple cycle. Furthermore, there cannot be an edge u—v_1 where u is not on the path; otherwise, we could make the path longer. Therefore, the only edge incident to v_1 is v_1—v_2, which means that v_1 is a leaf. By a symmetric argument, v_m is a second leaf.

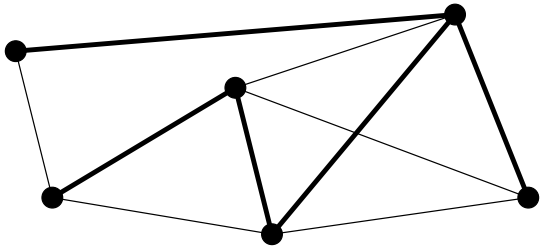
6. We use induction on the number of vertices. For a tree with a single vertex, the claim holds since it has no edges and $0+1=1$ vertex. Now suppose that the claim holds for all n-vertex trees and consider an $(n+1)$-vertex tree, T. Let v be a leaf of the tree. You can verify that deleting a vertex of degree 1 (and its incident edge) from any connected graph leaves a connected subgraph. So by 1., deleting v and its incident edge gives a smaller tree, and this smaller tree has one more vertex than edge by induction. If we re-attach the vertex, v, and its incident edge, then the equation still holds because the number of vertices and number of edges both increase by 1. Thus, the claim holds for T and, by induction, for all trees.

■

Various subsets of these properties provide alternative characterizations of trees, though we won't prove this. For example, a *connected* graph with a number of vertices one larger than the number of edges is necessarily a tree. Also, a graph with unique paths between every pair of vertices is necessarily a tree.

9.4.2 Spanning Trees

Trees are everywhere. In fact, every connected graph contains a subgraph that is a tree with the same vertices as the graph. This is a called a *spanning tree* for the graph. For example, here is a connected graph with a spanning tree highlighted.



Theorem 9.4.2. *Every connected graph contains a spanning tree.*

Proof. Let T be a connected subgraph of G, with the same vertices as G, and with the smallest number of edges possible for such a subgraph. We show that T is acyclic by contradiction. So suppose that T has a cycle with the following edges:

$$v_0\text{—}v_1, v_1\text{—}v_2, \ldots, v_n\text{—}v_0$$

Suppose that we remove the last edge, v_n—v_0. If a pair of vertices x and y was joined by a path not containing v_n—v_0, then they remain joined by that path. On the other hand, if x and y were joined by a path containing v_n—v_0, then they remain joined by a path containing the remainder of the cycle. So all the vertices of

G are still connected after we remove an edge from T. This is a contradiction, since T was defined to be a minimum size connected subgraph with all the vertices of G. So T must be acyclic. ■

9.4.3 Problems

Class Problems

Problem 9.22.
Procedure $Mark$ starts with a connected, simple graph with all edges unmarked and then marks some edges. At any point in the procedure a path that traverses only marked edges is called a *fully marked* path, and an edge that has no fully marked path between its endpoints is called *eligible*.

Procedure $Mark$ simply keeps marking eligible edges, and terminates when there are none.

Prove that $Mark$ terminates, and that when it does, the set of marked edges forms a spanning tree of the original graph.

Problem 9.23.

Procedure **create-spanning-tree**

Given a simple graph G, keep applying the following operations to the graph until no operation applies:

1. If an edge u—v of G is on a simple cycle, then delete u—v.
2. If vertices u and v of G are not connected, then add the edge u—v.

Assume the vertices of G are the integers $1, 2, \ldots, n$ for some $n \geq 2$. Procedure **create-spanning-tree** can be modeled as a state machine whose states are all possible simple graphs with vertices $1, 2, \ldots, n$. The start state is G, and the final states are the graphs on which no operation is possible.

(a) Let G be the graph with vertices $\{1, 2, 3, 4\}$ and edges

$$\{1\text{—}2, 3\text{—}4\}$$

What are the possible final states reachable from start state G? Draw them.

(b) Prove that any final state of must be a tree on the vertices.

(c) For any state, G', let e be the number of edges in G', c be the number of connected components it has, and s be the number of simple cycles. For each of the derived variables below, indicate the *strongest* of the properties that it is guaranteed to satisfy, no matter what the starting graph G is and be prepared to briefly explain your answer.

The choices for properties are: *constant*, *strictly increasing*, *strictly decreasing*, *weakly increasing*, *weakly decreasing*, *none of these*. The derived variables are

(i) e

(ii) c

(iii) s

(iv) $e - s$

(v) $c + e$

(vi) $3c + 2e$

(vii) $c + s$

(viii) (c, e), partially ordered coordinatewise (the *product* partial order, Ch. 6.4).

(d) Prove that procedure **create-spanning-tree** terminates. (If your proof depends on one of the answers to part (c), you must prove that answer is correct.)

Problem 9.24.
Prove that a graph is a tree iff it has a unique simple path between any two vertices.

Homework Problems

Problem 9.25. (a) Prove that the average degree of a tree is less than 2.

(b) Suppose every vertex in a graph has degree at least k. Explain why the graph has a simple path of length k.

Hint: Consider a longest simple path.

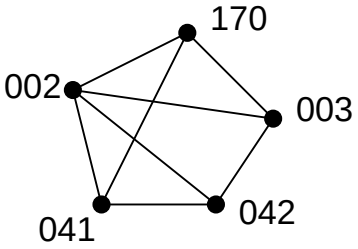
9.5 Coloring Graphs

In section 9.1.2, we used edges to indicate an affinity between two nodes, but having an edge represent a *conflict* between two nodes also turns out to be really useful.

9.6 Modelling Scheduling Conflicts

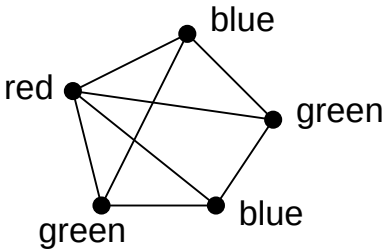
Each term the MIT Schedules Office must assign a time slot for each final exam. This is not easy, because some students are taking several classes with finals, and a student can take only one test during a particular time slot. The Schedules Office wants to avoid all conflicts. Of course, you can make such a schedule by having every exam in a different slot, but then you would need hundreds of slots for the hundreds of courses, and exam period would run all year! So, the Schedules Office would also like to keep exam period short. The Schedules Office's problem is easy

to describe as a graph. There will be a vertex for each course with a final exam, and two vertices will be adjacent exactly when some student is taking both courses. For example, suppose we need to schedule exams for 6.041, 6.042, 6.002, 6.003 and 6.170. The scheduling graph might look like this:



6.002 and 6.042 cannot have an exam at the same time since there are students in both courses, so there is an edge between their nodes. On the other hand, 6.042 and 6.170 can have an exam at the same time if they're taught at the same time (which they sometimes are), since no student can be enrolled in both (that is, no student *should* be enrolled in both when they have a timing conflict). Next, identify each time slot with a color. For example, Monday morning is red, Monday afternoon is blue, Tuesday morning is green, etc.

Assigning an exam to a time slot is now equivalent to coloring the corresponding vertex. The main constraint is that *adjacent vertices must get different colors* — otherwise, some student has two exams at the same time. Furthermore, in order to keep the exam period short, we should try to color all the vertices using as *few different colors as possible*. For our example graph, three colors suffice:



This coloring corresponds to giving one final on Monday morning (red), two Monday afternoon (blue), and two Tuesday morning (green). Can we use fewer than three colors? No! We can't use only two colors since there is a triangle in the graph, and three vertices in a triangle must all have different colors.

This is an example of what is a called a *graph coloring problem*: given a graph G, assign colors to each node such that adjacent nodes have different colors. A color

assignment with this property is called a *valid coloring* of the graph —a "*coloring*," for short. A graph G is *k-colorable* if it has a coloring that uses at most k colors.

Definition 9.6.1. The minimum value of k for which a graph, G, has a valid coloring is called its *chromatic number*, $\chi(G)$.

In general, trying to figure out if you can color a graph with a fixed number of colors can take a long time. It's a classic example of a problem for which no fast algorithms are known. In fact, it is easy to check if a coloring works, but it seems really hard to find it (if you figure out how, then you can get a $1 million Clay prize).

9.6.1 Degree-bounded Coloring

There are some simple graph properties that give useful upper bounds on colorings. For example, if we have a bound on the degrees of all the vertices in a graph, then we can easily find a coloring with only one more color than the degree bound.

Theorem 9.6.2. *A graph with maximum degree at most k is $(k+1)$-colorable.*

Unfortunately, if you try induction on k, it will lead to disaster. It is not that it is impossible, just that it is extremely painful and would ruin you if you tried it on an exam. Another option, especially with graphs, is to change what you are inducting on. In graphs, some good choices are n, the number of nodes, or e, the number of edges.

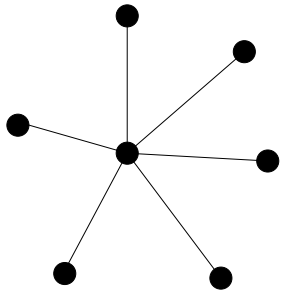
Proof. We use induction on the number of vertices in the graph, which we denote by n. Let $P(n)$ be the proposition that an n-vertex graph with maximum degree at most k is $(k+1)$-colorable.

Base case: $(n = 1)$ A 1-vertex graph has maximum degree 0 and is 1-colorable, so $P(1)$ is true.

Inductive step: Now assume that $P(n)$ is true, and let G be an $(n+1)$-vertex graph with maximum degree at most k. Remove a vertex v (and all edges incident to it), leaving an n-vertex subgraph, H. The maximum degree of H is at most k, and so H is $(k+1)$-colorable by our assumption $P(n)$. Now add back vertex v. We can assign v a color different from all its adjacent vertices, since there are at most k adjacent vertices and $k+1$ colors are available. Therefore, G is $(k+1)$-colorable. This completes the inductive step, and the theorem follows by induction. ■

Sometimes $k+1$ colors is the best you can do. For example, in the complete graph, K_n, every one of its n vertices is adjacent to all the others, so all n must be assigned different colors. Of course n colors is also enough, so $\chi(K_n) = n$. So K_{k+1} is an example where Theorem 9.6.2 gives the best possible bound. This means that Theorem 9.6.2 also gives the best possible bound for *any* graph with degree bounded by k that has K_{k+1} as a subgraph.

But sometimes $k+1$ colors is far from the best that you can do. Here's an example of an n-node star graph for $n = 7$:



In the n-node star graph, the maximum degree is $n-1$, but the star only needs 2 colors!

9.6.2 Why coloring?

One reason coloring problems come all the time is because scheduling conflicts are so common. For example, at Akamai, a new version of software is deployed over each of 20,000 servers every few days. The updates cannot be done at the same time since the servers need to be taken down in order to deploy the software. Also, the servers cannot be handled one at a time, since it would take forever to update them all (each one takes about an hour). Moreover, certain pairs of servers cannot be taken down at the same time since they have common critical functions. This problem was eventually solved by making a 20,000 node conflict graph and coloring it with 8 colors – so only 8 waves of install are needed! Another example comes from the need to assign frequencies to radio stations. If two stations have an overlap in their broadcast area, they can't be given the same frequency. Frequencies are precious and expensive, so you want to minimize the number handed out. This amounts to finding the minimum coloring for a graph whose vertices are the stations and whose edges are between stations with overlapping areas.

Coloring also comes up in allocating registers for program variables. While a variable is in use, its value needs to be saved in a register, but registers can often be reused for different variables. But two variables need different registers if they are referenced during overlapping intervals of program execution. So register allocation is the coloring problem for a graph whose vertices are the variables; vertices are adjacent if their intervals overlap, and the colors are registers.

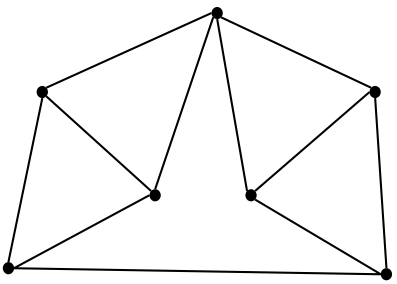
Finally, there's the famous map coloring problem stated in Propostion 1.5.4. The question is how many colors are needed to color a map so that adjacent territories get different colors? This is the same as the number of colors needed to color a graph that can be drawn in the plane without edges crossing. A proof that four colors are enough for the *planar* graphs was acclaimed when it was discovered about thirty years ago. Implicit in that proof was a 4-coloring procedure that takes time proportional to the number of vertices in the graph (countries in the map). On the other hand, it's another of those million dollar prize questions to find an

efficient procedure to tell if a planar graph really *needs* four colors or if three will actually do the job. But it's always easy to tell if an *arbitrary* graph is 2-colorable, as we show in Section 9.7. Later in Chapter 11, we'll develop enough planar graph theory to present an easy proof at least that planar graphs are 5-colorable.

9.6.3 Problems

Class Problems

Problem 9.26.
Let G be the graph below[6]. Carefully explain why $\chi(G) = 4$.



Homework Problems

Problem 9.27.
6.042 is often taught using recitations. Suppose it happened that 8 recitations were needed, with two or three staff members running each recitation. The assignment of staff to recitation sections is as follows:

- R1: Eli, Megumi, Rich
- R2: Eli, Stephanie, David
- R3: Megumi, Stav
- R4: Liz, Stephanie, Oscar

[6]From *Discrete Mathematics*, Lovász, Pelikan, and Vesztergombi. Springer, 2003. Exercise 13.3.1

- R5: Liz, Tom, David
- R6: Tom, Stav
- R7: Tom, Stephanie
- R8: Megumi, Stav, David

Two recitations can not be held in the same 90-minute time slot if some staff member is assigned to both recitations. The problem is to determine the minimum number of time slots required to complete all the recitations.

(a) Recast this problem as a question about coloring the vertices of a particular graph. Draw the graph and explain what the vertices, edges, and colors represent.

(b) Show a coloring of this graph using the fewest possible colors. What schedule of recitations does this imply?

Problem 9.28.
This problem generalizes the result proved Theorem 9.6.2 that any graph with maximum degree at most w is $(w+1)$-colorable.

A simple graph, G, is said to have *width*, w, iff its vertices can be arranged in a sequence such that each vertex is adjacent to at most w vertices that precede it in the sequence. If the degree of every vertex is at most w, then the graph obviously has width at most w —just list the vertices in any order.

(a) Describe an example of a graph with 100 vertices, width 3, but *average* degree more than 5. *Hint:* Don't get stuck on this; if you don't see it after five minutes, ask for a hint.

(b) Prove that every graph with width at most w is $(w+1)$-colorable.

(c) Prove that the average degree of a graph of width w is at most $2w$.

Exam Problems

Problem 9.29.
Recall that a *coloring* of a graph is an assignment of a color to each vertex such that no two adjacent vertices have the same color. A *k-coloring* is a coloring that uses at most k colors.

False Claim. *Let G be a graph whose vertex degrees are all $\leq k$. If G has a vertex of degree strictly less than k, then G is k-colorable.*

(a) Give a counterexample to the False Claim when $k = 2$.

(b) Underline the exact sentence or part of a sentence that is the first unjustified step in the following "proof" of the False Claim.

False proof. Proof by induction on the number n of vertices:

Induction hypothesis:

$P(n)$::= "Let G be an n-vertex graph whose vertex degrees are all $\leq k$. If G also has a vertex of degree strictly less than k, then G is k-colorable."

Base case: ($n = 1$) G has one vertex, the degree of which is 0. Since G is 1-colorable, $P(1)$ holds.

Inductive step:

We may assume $P(n)$. To prove $P(n+1)$, let G_{n+1} be a graph with $n+1$ vertices whose vertex degrees are all k or less. Also, suppose G_{n+1} has a vertex, v, of degree strictly less than k. Now we only need to prove that G_{n+1} is k-colorable.

To do this, first remove the vertex v to produce a graph, G_n, with n vertices. Let u be a vertex that is adjacent to v in G_{n+1}. Removing v reduces the degree of u by 1. So in G_n, vertex u has degree strictly less than k. Since no edges were added, the vertex degrees of G_n remain $\leq k$. So G_n satisfies the conditions of the induction hypothesis, $P(n)$, and so we conclude that G_n is k-colorable.

Now a k-coloring of G_n gives a coloring of all the vertices of G_{n+1}, except for v. Since v has degree less than k, there will be fewer than k colors assigned to the nodes adjacent to v. So among the k possible colors, there will be a color not used to color these adjacent nodes, and this color can be assigned to v to form a k-coloring of G_{n+1}. ■

(c) With a slightly strengthened condition, the preceding proof of the False Claim could be revised into a sound proof of the following Claim:
Claim. *Let G be a graph whose vertex degrees are all $\leq k$. If* ⟨statement inserted from below⟩ *has a vertex of degree strictly less than k, then G is k-colorable.*

Circle each of the statements below that could be inserted to make the Claim true.

- G is connected and
- G has no vertex of degree zero and
- G does not contain a complete graph on k vertices and
- every connected component of G
- some connected component of G

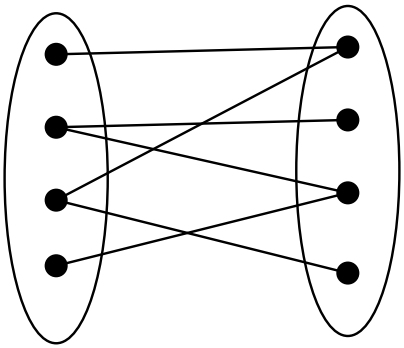
9.7 Bipartite Matchings

9.7.1 Bipartite Graphs

There were two kinds of vertices in the "Sex in America" graph —males and females, and edges only went between the two kinds. Graphs like this come up so frequently they have earned a special name —they are called *bipartite graphs*.

Definition 9.7.1. A *bipartite graph* is a graph together with a partition of its vertices into two sets, L and R, such that every edge is incident to a vertex in L and to a vertex in R.

So every bipartite graph looks something like this:



Now we can immediately see how to color a bipartite graph using only two colors: let all the L vertices be black and all the R vertices be white. Conversely, if a graph is 2-colorable, then it is bipartite with L being the vertices of one color and R the vertices of the other color. In other words,

"bipartite" is a synonym for "*2-colorable*."

The following Lemma gives another useful characterization of bipartite graphs.

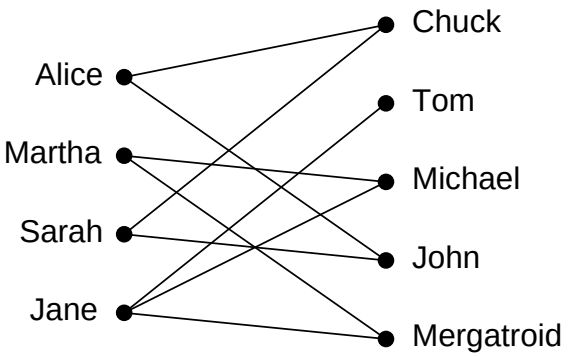
Theorem 9.7.2. *A graph is bipartite iff it has no odd-length cycle.*

The proof of Theorem 9.7.2 is left to Problem 9.33.

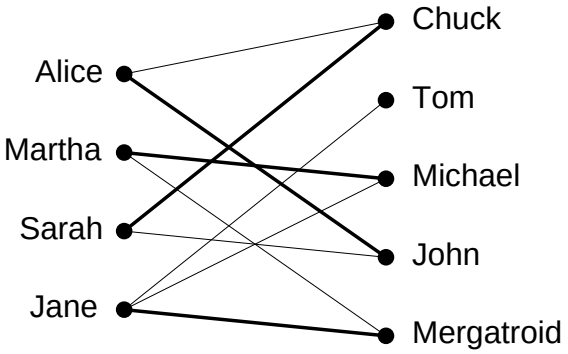
9.7.2 Bipartite Matchings

The *bipartite matching* problem resembles the stable Marriage Problem in that it concerns a set of girls and a set of at least as many boys. There are no preference lists, but each girl does have some boys she likes and others she does not like. In the bipartite matching problem, we ask whether every girl can be paired up with a boy that she likes. Any particular matching problem can be specified by a bipartite

graph with a vertex for each girl, a vertex for each boy, and an edge between a boy and a girl iff the girl likes the boy. For example, we might obtain the following graph:



Now a *matching* will mean a way of assigning every girl to a boy so that different girls are assigned to different boys, and a girl is always assigned to a boy she likes. For example, here is one possible matching for the girls:



Hall's Matching Theorem states necessary and sufficient conditions for the existence of a matching in a bipartite graph. It turns out to be a remarkably useful mathematical tool.

9.7.3 The Matching Condition

We'll state and prove Hall's Theorem using girl-likes-boy terminology. Define *the set of boys liked by a given set of girls* to consist of all boys liked by at least one of those girls. For example, the set of boys liked by Martha and Jane consists of Tom, Michael, and Mergatroid. For us to have any chance at all of matching up the girls, the following *matching condition* must hold:

Every subset of girls likes at least as large a set of boys.

For example, we can not find a matching if some 4 girls like only 3 boys. Hall's Theorem says that this necessary condition is actually sufficient; if the matching condition holds, then a matching exists.

Theorem 9.7.3. *A matching for a set of girls G with a set of boys B can be found if and only if the matching condition holds.*

Proof. First, let's suppose that a matching exists and show that the matching condition holds. Consider an arbitrary subset of girls. Each girl likes at least the boy she is matched with. Therefore, every subset of girls likes at least as large a set of boys. Thus, the matching condition holds.

Next, let's suppose that the matching condition holds and show that a matching exists. We use strong induction on $|G|$, the number of girls.

Base Case: ($|G| = 1$) If $|G| = 1$, then the matching condition implies that the lone girl likes at least one boy, and so a matching exists.

Inductive Step: Now suppose that $|G| \geq 2$. There are two cases:

Case 1: Every proper subset of girls likes a *strictly larger* set of boys. In this case, we have some latitude: we pair an arbitrary girl with a boy she likes and send them both away. The matching condition still holds for the remaining boys and girls, so we can match the rest of the girls by induction.

Case 2: Some proper subset of girls $X \subset G$ likes an *equal-size* set of boys $Y \subset B$. We match the girls in X with the boys in Y by induction and send them all away. We can also match the rest of the girls by induction if we show that the matching condition holds for the remaining boys and girls. To check the matching condition for the remaining people, consider an arbitrary subset of the remaining girls $X' \subseteq (G - X)$, and let Y' be the set of remaining boys that they like. We must show that $|X'| \leq |Y'|$. Originally, the combined set of girls $X \cup X'$ liked the set of boys $Y \cup Y'$. So, by the matching condition, we know:

$$|X \cup X'| \leq |Y \cup Y'|$$

We sent away $|X|$ girls from the set on the left (leaving X') and sent away an equal number of boys from the set on the right (leaving Y'). Therefore, it must be that $|X'| \leq |Y'|$ as claimed.

So there is in any case a matching for the girls, which completes the proof of the Inductive step. The theorem follows by induction. ■

The proof of this theorem gives an algorithm for finding a matching in a bipartite graph, albeit not a very efficient one. However, efficient algorithms for finding a matching in a bipartite graph do exist. Thus, if a problem can be reduced to finding a matching, the problem is essentially solved from a computational perspective.

9.7.4 A Formal Statement

Let's restate Hall's Theorem in abstract terms so that you'll not always be condemned to saying, "Now this group of little girls likes at least as many little boys..."

A *matching* in a graph, G, is a set of edges such that no two edges in the set share a vertex. A matching is said to *cover* a set, L, of vertices iff each vertex in L has an edge of the matching incident to it. In any graph, the set $N(S)$, of *neighbors*[7] of some set, S, of vertices is the set of all vertices adjacent to some vertex in S. That is,

$$N(S) ::= \{r \mid s\text{—}r \text{ is an edge for some } s \in S\}.$$

S is called a *bottleneck* if

$$|S| > |N(S)|.$$

Theorem 9.7.4 (Hall's Theorem)**.** *Let G be a bipartite graph with vertex partition L, R. There is matching in G that covers L iff no subset of L is a bottleneck.*

An Easy Matching Condition

The bipartite matching condition requires that *every* subset of girls has a certain property. In general, verifying that every subset has some property, even if it's easy to check any particular subset for the property, quickly becomes overwhelming because the number of subsets of even relatively small sets is enormous —over a billion subsets for a set of size 30. However, there is a simple property of vertex degrees in a bipartite graph that guarantees a match and is very easy to check. Namely, call a bipartite graph *degree-constrained* if vertex degrees on the left are at least as large as those on the right. More precisely,

Definition 9.7.5. A bipartite graph G with vertex partition L, R is *degree-constrained* if $\deg(l) \geq \deg(r)$ for every $l \in L$ and $r \in R$.

Now we can always find a matching in a degree-constrained bipartite graph.

Lemma 9.7.6. *Every degree-constrained bipartite graph satisifies the matching condition.*

Proof. Let S be any set of vertices in L. The number of edges incident to vertices in S is exactly the sum of the degrees of the vertices in S. Each of these edges is incident to a vertex in $N(S)$ by definition of $N(S)$. So the sum of the degrees of the vertices in $N(S)$ is at least as large as the sum for S. But since the degree of every vertex in $N(S)$ is at most as large as the degree of every vertex in S, there would have to be at least as many terms in the sum for $N(S)$ as in the sum for S. So there have to be at least as many vertices in $N(S)$ as in S, proving that S is not a bottleneck. So there are no bottlenecks, proving that the degree-constrained graph satisifies the matching condition. ■

Of course being degree-constrained is a very strong property, and lots of graphs that aren't degree-constrained have matchings. But we'll see examples of degree-constrained graphs come up naturally in some later applications.

[7] An equivalent definition of $N(S)$ uses relational notation: $N(S)$ is simply the image, SR, of S under the adjacency relation, R, on vertices of the graph.

9.7.5 Problems

Class Problems

Problem 9.30.
A certain Institute of Technology has a lot of student clubs; these are loosely overseen by the Student Association. Each eligible club would like to delegate one of its members to appeal to the Dean for funding, but the Dean will not allow a student to be the delegate of more than one club. Fortunately, the Association VP took Math for Computer Science and recognizes a matching problem when she sees one.

(a) Explain how to model the delegate selection problem as a bipartite matching problem.

(b) The VP's records show that no student is a member of more than 9 clubs. The VP also knows that to be eligible for support from the Dean's office, a club must have at least 13 members. That's enough for her to guarantee there is a proper delegate selection. Explain. (If only the VP had taken an *Algorithms*, she could even have found a delegate selection without much effort.)

Problem 9.31.
A *Latin square* is $n \times n$ array whose entries are the number $1, \ldots, n$. These entries satisfy two constraints: every row contains all n integers in some order, and also every column contains all n integers in some order. Latin squares come up frequently in the design of scientific experiments for reasons illustrated by a little story in a footnote[8]

[8] At Guinness brewery in the eary 1900's, W. S. Gosset (a chemist) and E. S. Beavan (a "maltster") were trying to improve the barley used to make the brew. The brewery used different varieties of barley according to price and availability, and their agricultural consultants suggested a different fertilizer mix and best planting month for each variety.

Somewhat sceptical about paying high prices for customized fertilizer, Gosset and Beavan planned a season long test of the influence of fertilizer and planting month on barley yields. For as many months as there were varieties of barley, they would plant one sample of each variety using a different one of the fertilizers. So every month, they would have all the barley varieties planted and all the fertilizers used, which would give them a way to judge the overall quality of that planting month. But they also wanted to judge the fertilizers, so they wanted each fertilizer to be used on each variety during the course of the season. Now they had a little mathematical problem, which we can abstract as follows.

Suppose there are n barley varieties and an equal number of recommended fertilizers. Form an $n \times n$ array with a column for each fertilizer and a row for each planting month. We want to fill in the entries of this array with the integers 1,...,n numbering the barley varieties, so that every row contains all n integers in some order (so every month each variety is planted and each fertilizer is used), and also every column contains all n integers (so each fertilizer is used on all the varieties over the course of the growing season).

For example, here is a 4×4 Latin square:

1	2	3	4
3	4	2	1
2	1	4	3
4	3	1	2

(a) Here are three rows of what could be part of a 5×5 Latin square:

2	4	5	3	1
4	1	3	2	5
3	2	1	5	4

Fill in the last two rows to extend this "Latin rectangle" to a complete Latin square.

(b) Show that filling in the next row of an $n \times n$ Latin rectangle is equivalent to finding a matching in some $2n$-vertex bipartite graph.

(c) Prove that a matching must exist in this bipartite graph and, consequently, a Latin rectangle can always be extended to a Latin square.

Exam Problems

Problem 9.32.
Overworked and over-caffeinated, the TAs decide to oust Albert and teach their own recitations. They will run a recitation session at 4 different times in the same room. There are exactly 20 chairs to which a student can be assigned in each recitation. Each student has provided the TAs with a list of the recitation sessions her schedule allows and no student's schedule conflicts with all 4 sessions. The TAs must assign each student to a chair during recitation at a time she can attend, if such an assignment is possible.

(a) Describe how to model this situation as a matching problem. Be sure to specify what the vertices/edges should be and briefly describe how a matching would determine seat assignments for each student in a recitation that does not conflict with his schedule. (This is a *modeling problem*; we aren't looking for a description of an algorithm to solve the problem.)

(b) Suppose there are 65 students. Given the information provided above, is a matching guaranteed? Briefly explain.

Homework Problems

Problem 9.33.
In this problem you will prove:

Theorem. *A graph G is 2-colorable iff it contains no odd length cycle.*

As usual with "iff" assertions, the proof splits into two proofs: part (a) asks you to prove that the left side of the "iff" implies the right side. The other problem parts prove that the right side implies the left.

(a) Assume the left side and prove the right side. Three to five sentences should suffice.

(b) Now assume the right side. As a first step toward proving the left side, explain why we can focus on a single connected component H within G.

(c) As a second step, explain how to 2-color any tree.

(d) Choose any 2-coloring of a spanning tree, T, of H. Prove that H is 2-colorable by showing that any edge *not* in T must also connect different-colored vertices.

Problem 9.34.
Take a regular deck of 52 cards. Each card has a suit and a value. The suit is one of four possibilities: heart, diamond, club, spade. The value is one of 13 possibilities, $A, 2, 3, \ldots, 10, J, Q, K$. There is exactly one card for each of the 4×13 possible combinations of suit and value.

Ask your friend to lay the cards out into a grid with 4 rows and 13 columns. They can fill the cards in any way they'd like. In this problem you will show that you can always pick out 13 cards, one from each column of the grid, so that you wind up with cards of all 13 possible values.

(a) Explain how to model this trick as a bipartite matching problem between the 13 column vertices and the 13 value vertices. Is the graph necessarily degree constrained?

(b) Show that any n columns must contain at least n different values and prove that a matching must exist.

Problem 9.35.
Scholars through the ages have identified *twenty* fundamental human virtues: honesty, generosity, loyalty, prudence, completing the weekly course reading-response, etc. At the beginning of the term, every student in 6.042 possessed exactly *eight* of these virtues. Furthermore, every student was unique; that is, no two students possessed exactly the same set of virtues. The 6.042 course staff must select *one* additional virtue to impart to each student by the end of the term. Prove that there is

a way to select an additional virtue for each student so that every student is unique at the end of the term as well.

Suggestion: Use Hall's theorem. Try various interpretations for the vertices on the left and right sides of your bipartite graph.

Chapter 10

Recursive Data Types

Recursive data types play a central role in programming. From a mathematical point of view, recursive data types are what induction is about. Recursive data types are specified by *recursive definitions* that say how to build something from its parts. These definitions have two parts:

- **Base case(s)** that don't depend on anything else.
- **Constructor case(s)** that depend on previous cases.

10.1 Strings of Brackets

Let `brkts` be the set of all strings of square brackets. For example, the following two strings are in `brkts`:

$$]][[[[]] \quad \text{and} \quad [[]][]][\tag{10.1}$$

Since we're just starting to study recursive data, just for practice we'll formulate `brkts` as a recursive data type,

Definition 10.1.1. The data type, `brkts`, of strings of brackets is defined recursively:

- **Base case:** The *empty string*, λ, is in `brkts`.
- **Constructor case:** If $s \in$ `brkts`, then $s]$ and $s[$ are in `brkts`.

Here we're writing $s]$ to indicate the string that is the sequence of brackets (if any) in the string s, followed by a right bracket; similarly for $s[$.

A string, $s \in$ `brkts`, is called a *matched string* if its brackets "match up" in the usual way. For example, the left hand string above is not matched because its second right bracket does not have a matching left bracket. The string on the right is matched.

We're going to examine several different ways to define and prove properties of matched strings using recursively defined sets and functions. These properties are pretty straighforward, and you might wonder whether they have any particular relevance in computer scientist —other than as a nonnumerical example of recursion. The honest answer is "not much relevance, *any more*." The reason for this is one of the great successes of computer science.

Expression Parsing

During the early development of computer science in the 1950's and 60's, creation of effective programming language compilers was a central concern. A key aspect in processing a program for compilation was expression parsing. The problem was to take in an expression like

$$x + y * z^2 \div y + 7$$

and *put in* the brackets that determined how it should be evaluated —should it be

$$[[x + y] * z^2 \div y] + 7, \text{ or,}$$
$$x + [y * z^2 \div [y + 7]], \text{ or,}$$
$$[x + [y * z^2]] \div [y + 7],$$

or . . . ?

The Turing award (the "Nobel Prize" of computer science) was ultimately bestowed on Robert Floyd, for, among other things, being discoverer of a simple program that would insert the brackets properly.

In the 70's and 80's, this parsing technology was packaged into high-level compiler-compilers that automatically generated parsers from expression grammars. This automation of parsing was so effective that the subject needed no longer demanded attention. It largely disappeared from the computer science curriculum by the 1990's.

One precise way to determine if a string is matched is to start with 0 and read the string from left to right, adding 1 to the count for each left bracket and subtracting 1 from the count for each right bracket. For example, here are the counts

for the two strings above

	[	]	]	[	[	[	[	]	]	]	]
0	1	0	−1	0	1	2	3	4	3	2	1	0

	[	[	[	]	]	[	]	]	[	]
0	1	2	3	2	1	2	1	0	1	0

A string has a *good count* if its running count never goes negative and ends with 0. So the second string above has a good count, but the first one does not because its count went negative at the third step.

Definition 10.1.2. Let

$$\text{GoodCount} ::= \{s \in \texttt{brkts} \mid s \text{ has a good count}\}.$$

The matched strings can now be characterized precisely as this set of strings with good counts. But it turns out to be really useful to characterize the matched strings in another way as well, namely, as a recursive data type:

Definition 10.1.3. Recursively define the set, RecMatch, of strings as follows:

- **Base case:** $\lambda \in \text{RecMatch}$.
- **Constructor case:** If $s, t \in \text{RecMatch}$, then

$$[\, s \,]\, t \in \text{RecMatch}.$$

Here we're writing $[\, s \,]\, t$ to indicate the string that starts with a left bracket, followed by the sequence of brackets (if any) in the string s, followed by a right bracket, and ending with the sequence of brackets in the string t.

Using this definition, we can see that $\lambda \in \text{RecMatch}$ by the Base case, so

$$[\,\lambda\,]\,\lambda = [\,\,] \in \text{RecMatch}$$

by the Constructor case. So now,

$$\begin{aligned}
[\,\lambda\,][\,\,] &= [\,\,][\,\,] \in \text{RecMatch} && \text{(letting } s = \lambda, t = [\,\,]\text{)}\\
[\,[\,\,]\,]\,\lambda &= [\,[\,\,]\,] \in \text{RecMatch} && \text{(letting } s = [\,\,], t = \lambda\text{)}\\
[\,[\,\,]\,][\,\,] &\in \text{RecMatch} && \text{(letting } s = [\,\,], t = [\,\,]\text{)}
\end{aligned}$$

are also strings in RecMatch by repeated applications of the Constructor case. If you haven't seen this kind of definition before, you should try continuing this example to verify that $[\,[\,[\,\,]\,][\,]\,][\,\,] \in \text{RecMatch}$

Given the way this section is set up, you might guess that RecMatch = GoodCount, and you'd be right, but it's not completely obvious. The proof is worked out in Problem 10.6.

10.2 Arithmetic Expressions

Expression evaluation is a key feature of programming languages, and recognition of expressions as a recursive data type is a key to understanding how they can be processed.

To illustrate this approach we'll work with a toy example: arithmetic expressions like $3x^2 + 2x + 1$ involving only one variable, "x." We'll refer to the data type of such expressions as Aexp. Here is its definition:

Definition 10.2.1. • **Base cases:**

1. The variable, x, is in Aexp.
2. The arabic numeral, `k`, for any nonnegative integer, k, is in Aexp.

- **Constructor cases:** If $e, f \in$ Aexp, then

3. $(e + f) \in$ Aexp. The expression $(e + f)$ is called a *sum*. The Aexp's e and f are called the *components* of the sum; they're also called the *summands*.
4. $(e * f) \in$ Aexp. The expression $(e * f)$ is called a *product*. The Aexp's e and f are called the *components* of the product; they're also called the *multiplier* and *multiplicand*.
5. $-(e) \in$ Aexp. The expression $-(e)$ is called a *negative*.

Notice that Aexp's are fully parenthesized, and exponents aren't allowed. So the Aexp version of the polynomial expression $3x^2 + 2x + 1$ would officially be written as

$$((\mathtt{3} * (x * x)) + ((\mathtt{2} * x) + \mathtt{1})). \tag{10.2}$$

These parentheses and $*$'s clutter up examples, so we'll often use simpler expressions like "$3x^2 + 2x + 1$" instead of (10.2). But it's important to recognize that $3x^2 + 2x + 1$ is not an Aexp; it's an *abbreviation* for an Aexp.

10.3 Structural Induction on Recursive Data Types

Structural induction is a method for proving some property, P, of all the elements of a recursively-defined data type. The proof consists of two steps:

- Prove P for the base cases of the definition.
- Prove P for the constructor cases of the definition, assuming that it is true for the component data items.

A very simple application of structural induction proves that the recursively defined matched strings always have an equal number of left and right brackets. To do this, define a predicate, P, on strings $s \in$ `brkts`:

$$P(s) ::= \quad s \text{ has an equal number of left and right brackets.}$$

Proof. We'll prove that $P(s)$ holds for all $s \in$ RecMatch by structural induction on the definition that $s \in$ RecMatch, using $P(s)$ as the induction hypothesis.

Base case: $P(\lambda)$ holds because the empty string has zero left and zero right brackets.

Constructor case: For $r = [\, s \,]\, t$, we must show that $P(r)$ holds, given that $P(s)$ and $P(t)$ holds. So let n_s, n_t be, respectively, the number of left brackets in s and t. So the number of left brackets in r is $1 + n_s + n_t$.

Now from the respective hypotheses $P(s)$ and $P(t)$, we know that the number of right brackets in s is n_s, and likewise, the number of right brackets in t is n_t. So the number of right brackets in r is $1 + n_s + n_t$, which is the same as the number of left brackets. This proves $P(r)$. We conclude by structural induction that $P(s)$ holds for all $s \in$ RecMatch. ■

10.3.1 Functions on Recursively-defined Data Types

Functions on recursively-defined data types can be defined recursively using the same cases as the data type definition. Namely, to define a function, f, on a recursive data type, define the value of f for the base cases of the data type definition, and then define the value of f in each constructor case in terms of the values of f on the component data items.

For example, from the recursive definition of the set, RecMatch, of strings of matched brackets, we define:

Definition 10.3.1. The *depth*, $d(s)$, of a string, $s \in$ RecMatch, is defined recursively by the rules:

- $d(\lambda) ::= 0$.
- $d([\, s \,]\, t) ::= \max\{d(s) + 1, d(t)\}$

Warning: When a recursive definition of a data type allows the same element to be constructed in more than one way, the definition is said to be *ambiguous*. A function defined recursively from an ambiguous definition of a data type will not be well-defined unless the values specified for the different ways of constructing the element agree.

We were careful to choose an *un*ambiguous definition of RecMatch to ensure that functions defined recursively on the definition would always be well-defined. As an example of the trouble an ambiguous definition can cause, let's consider yet another definition of the matched strings.

Example 10.3.2. Define the set, $M \subseteq \texttt{brkts}$ recursively as follows:

- **Base case:** $\lambda \in M$,
- **Constructor cases:** if $s, t \in M$, then the strings $[\, s \,]$ and st are also in M.

Quick Exercise: Give an easy proof by structural induction that $M = \text{RecMatch}$.

Since $M = \text{RecMatch}$, and the definition of M seems more straightforward, why didn't we use it? Because the definition of M is ambiguous, while the trickier definition of RecMatch is unambiguous. Does this ambiguity matter? Yes it does. For suppose we defined

$$\begin{aligned} f(\lambda) &::= 1, \\ f([\,s\,]) &::= 1 + f(s), \\ f(st) &::= (f(s)+1)\cdot(f(t)+1) \qquad \text{for } st \neq \lambda. \end{aligned}$$

Let a be the string $[[]] \in M$ built by two successive applications of the first M constructor starting with λ. Next let $b ::= aa$ and $c ::= bb$, each built by successive applications of the second M constructor starting with a.

Alternatively, we can build ba from the second constructor with $s = b$ and $t = a$, and then get to c using the second constructor with $s = ba$ and $t = a$.

Now by these rules, $f(a) = 2$, and $f(b) = (2+1)(2+1) = 9$. This means that $f(c) = f(bb) = (9+1)(9+1) = 100$.

But also $f(ba) = (9+1)(2+1) = 27$, so that $f(c) = f(ba\,a) = (27+1)(2+1) = 84$.

The outcome is that $f(c)$ is defined to be both 100 and 84, which shows that the rules defining f are inconsistent.

On the other hand, structural induction remains a sound proof method even for ambiguous recursive definitions, which is why it was easy to prove that $M = \text{RecMatch}$.

10.3.2 Recursive Functions on Nonnegative Integers

The nonnegative integers can be understood as a recursive data type.

Definition 10.3.3. The set, $\mathbb{N}$, is a data type defined recursivly as:

- $0 \in \mathbb{N}$.
- If $n \in \mathbb{N}$, then the *successor*, $n+1$, of n is in $\mathbb{N}$.

This of course makes it clear that ordinary induction is simply the special case of structural induction on the recursive Definition 10.3.3, This also justifies the familiar recursive definitions of functions on the nonnegative integers. Here are some examples.

The Factorial function. This function is often written "$n!$." You will see a lot of it later in the term. Here we'll use the notation $\text{fac}(n)$:

- $\text{fac}(0) ::= 1$.
- $\text{fac}(n+1) ::= (n+1)\cdot\text{fac}(n)$ for $n \geq 0$.

The Fibonacci numbers. Fibonacci numbers arose out of an effort 800 years ago to model population growth. They have a continuing fan club of people captivated by their extraordinary properties. The nth Fibonacci number, fib, can be defined recursively by:

$$\begin{aligned} \text{fib}(0) &::= 0, \\ \text{fib}(1) &::= 1, \\ \text{fib}(n) &::= \text{fib}(n-1) + \text{fib}(n-2) \qquad \text{for } n \geq 2. \end{aligned}$$

Here the recursive step starts at $n = 2$ with base cases for 0 and 1. This is needed since the recursion relies on two previous values.

What is $\text{fib}(4)$? Well, $\text{fib}(2) = \text{fib}(1) + \text{fib}(0) = 1$, $\text{fib}(3) = \text{fib}(2) + \text{fib}(1) = 2$, so $\text{fib}(4) = 3$. The sequence starts out $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$.

Sum-notation. Let "$S(n)$" abbreviate the expression "$\sum_{i=1}^{n} f(i)$." We can recursively define $S(n)$ with the rules

- $S(0) ::= 0$.
- $S(n+1) ::= f(n+1) + S(n)$ for $n \geq 0$.

Ill-formed Function Definitions

There are some blunders to watch out for when defining functions recursively. Below are some function specifications that resemble good definitions of functions on the nonnegative integers, but they aren't.

$$f_1(n) ::= 2 + f_1(n-1). \tag{10.3}$$

This "definition" has no base case. If some function, f_1, satisfied (10.3), so would a function obtained by adding a constant to the value of f_1. So equation (10.3) does not uniquely define an f_1.

$$f_2(n) ::= \begin{cases} 0, & \text{if } n = 0, \\ f_2(n+1) & \text{otherwise.} \end{cases} \tag{10.4}$$

This "definition" has a base case, but still doesn't uniquely determine f_2. Any function that is 0 at 0 and constant everywhere else would satisfy the specification, so (10.4) also does not uniquely define anything.

In a typical programming language, evaluation of $f_2(1)$ would begin with a recursive call of $f_2(2)$, which would lead to a recursive call of $f_2(3)$, ... with recursive calls continuing without end. This "operational" approach interprets (10.4) as defining a *partial* function, f_2, that is undefined everywhere but 0.

$$f_3(n) ::= \begin{cases} 0, & \text{if } n \text{ is divisible by 2,} \\ 1, & \text{if } n \text{ is divisible by 3,} \\ 2, & \text{otherwise.} \end{cases} \tag{10.5}$$

This "definition" is inconsistent: it requires $f_3(6) = 0$ and $f_3(6) = 1$, so (10.5) doesn't define anything.

A Mysterious Function

Mathematicians have been wondering about this function specification for a while:

$$f_4(n) ::= \begin{cases} 1, & \text{if } n \leq 1, \\ f_4(n/2) & \text{if } n > 1 \text{ is even,} \\ f_4(3n+1) & \text{if } n > 1 \text{ is odd.} \end{cases} \tag{10.6}$$

For example, $f_4(3) = 1$ because

$$f_4(3) ::= f_4(10) ::= f_4(5) ::= f_4(16) ::= f_4(8) ::= f_4(4) ::= f_4(2) ::= f_4(1) ::= 1.$$

The constant function equal to 1 will satisfy (10.6), but it's not known if another function does too. The problem is that the third case specifies $f_4(n)$ in terms of f_4 at arguments larger than n, and so cannot be justified by induction on $\mathbb{N}$. It's known that any f_4 satisfying (10.6) equals 1 for all n up to over a billion.

Quick exercise: Why does the constant function 1 satisfy (10.6)?

10.3.3 Evaluation and Substitution with Aexp's

Evaluating Aexp's

Since the only variable in an Aexp is x, the value of an Aexp is determined by the value of x. For example, if the value of x is 3, then the value of $3x^2 + 2x + 1$ is obviously 34. In general, given any Aexp, e, and an integer value, n, for the variable, x, we can evaluate e to finds its value, $\text{eval}(e, n)$. It's easy, and useful, to specify this evaluation process with a recursive definition.

Definition 10.3.4. The *evaluation function*, $\text{eval} : \text{Aexp} \times \mathbb{Z} \to \mathbb{Z}$, is defined recursively on expressions, $e \in \text{Aexp}$, as follows. Let n be any integer.

- **Base cases:**

  1. Case[e is x]

     $$\text{eval}(x, n) ::= n.$$

     (The value of the variable, x, is given to be n.)

2. Case[e is k]

$$\text{eval}(\mathtt{k}, n) ::= k.$$

(The value of the numeral $\mathtt{k}$ is the integer k, no matter what value x has.)

- **Constructor cases:**

3. Case[e is $(e_1 + e_2)$]

$$\text{eval}((e_1 + e_2), n) ::= \text{eval}(e_1, n) + \text{eval}(e_2, n).$$

4. Case[e is $(e_1 * e_2)$]

$$\text{eval}((e_1 * e_2), n) ::= \text{eval}(e_1, n) \cdot \text{eval}(e_2, n).$$

5. Case[e is $-(e_1)$]

$$\text{eval}(-(e_1), n) ::= -\,\text{eval}(e_1, n).$$

For example, here's how the recursive definition of eval would arrive at the value of $3 + x^2$ when x is 2:

$$\begin{aligned}
\text{eval}((\mathtt{3} + (x * x)), 2) &= \text{eval}(\mathtt{3}, 2) + \text{eval}((x * x), 2) && \text{(by Def 10.3.4.3)}\\
&= 3 + \text{eval}((x * x), 2) && \text{(by Def 10.3.4.2)}\\
&= 3 + (\text{eval}(x, 2) \cdot \text{eval}(x, 2)) && \text{(by Def 10.3.4.4)}\\
&= 3 + (2 \cdot 2) && \text{(by Def 10.3.4.1)}\\
&= 3 + 4 = 7.
\end{aligned}$$

Substituting into Aexp's

Substituting expressions for variables is a standard, important operation. For example the result of substituting the expression $3x$ for x in the $(x(x-1))$ would be $(3x(3x-1)$. We'll use the general notation $\text{subst}(f, e)$ for the result of substituting an Aexp, f, for each of the x's in an Aexp, e. For instance,

$$\text{subst}(3x, x(x-1)) = 3x(3x-1).$$

This substitution function has a simple recursive definition:

Definition 10.3.5. The *substitution function* from Aexp $\times$ Aexp to Aexp is defined recursively on expressions, $e \in$ Aexp, as follows. Let f be any Aexp.

- **Base cases:**

1. Case[e is x]

$$\text{subst}(f, x) ::= f.$$

(The result of substituting f for the variable, x, is just f.)

2. Case[e is $\mathtt{k}$]

$$\text{subst}(f, \mathtt{k}) ::= \mathtt{k}.$$

(The numeral, $\mathtt{k}$, has no x's in it to substitute for.)

- **Constructor cases:**

3. Case[e is $(e_1 + e_2)$]

$$\text{subst}(f, (e_1 + e_2))) ::= (\text{subst}(f, e_1) + \text{subst}(f, e_2)).$$

4. Case[e is $(e_1 * e_2)$]

$$\text{subst}(f, (e_1 * e_2))) ::= (\text{subst}(f, e_1) * \text{subst}(f, e_2)).$$

5. Case[e is $-(e_1)$]

$$\text{subst}(f, -(e_1)) ::= -(\text{subst}(f, e_1)).$$

Here's how the recursive definition of the substitution function would find the result of substituting $3x$ for x in the $x(x-1)$:

$$\begin{aligned}
\text{subst}(3x, (x(x-1))) &= \text{subst}(3x, (x * (x + -(1)))) && \text{(unabbreviating)}\\
&= (\text{subst}(3x, x) * \text{subst}(3x, (x + -(1)))) && \text{(by Def 10.3.5 4)}\\
&= (3x * \text{subst}(3x, (x + -(1)))) && \text{(by Def 10.3.5 1)}\\
&= (3x * (\text{subst}(3x, x) + \text{subst}(3x, -(1)))) && \text{(by Def 10.3.5 3)}\\
&= (3x * (3x + -(\text{subst}(3x, 1)))) && \text{(by Def 10.3.5 1 \& 5)}\\
&= (3x * (3x + -(1))) && \text{(by Def 10.3.5 2)}\\
&= 3x(3x - 1) && \text{(abbreviation)}
\end{aligned}$$

Now suppose we have to find the value of $\text{subst}(3x, (x(x-1)))$ when $x = 2$. There are two approaches.

First, we could actually do the substitution above to get $3x(3x-1)$, and then we could evaluate $3x(3x-1)$ when $x = 2$, that is, we could recursively calculate $\text{eval}(3x(3x-1), 2)$ to get the final value 30. In programming jargon, this would be called evaluation using the *Substitution Model*. Tracing through the steps in the evaluation, we find that the Substitution Model requires two substitutions for occurrences of x and 5 integer operations: 3 integer multiplications, 1 integer addition, and 1 integer negative operation. Note that in this Substitution Model the multiplication $3 \cdot 2$ was performed twice to get the value of 6 for each of the two occurrences of $3x$.

The other approach is called evaluation using the *Environment Model*. Namely, we evaluate $3x$ when $x = 2$ using just 1 multiplication to get the value 6. Then we evaluate $x(x-1)$ when x has this value 6 to arrive at the value $6 \cdot 5 = 30$. So the Environment Model requires 2 variable lookups and only 4 integer operations: 1

multiplication to find the value of $3x$, another multiplication to find the value $6 \cdot 5$, along with 1 integer addition and 1 integer negative operation.

So the Environment Model approach of calculating

$$\text{eval}(x(x-1), \text{eval}(3x, 2))$$

instead of the Substitution Model approach of calculating

$$\text{eval}(\text{subst}(3x, x(x-1)), 2)$$

is faster. But how do we know that these final values reached by these two approaches always agree? We can prove this easily by structural induction on the definitions of the two approaches. More precisely, what we want to prove is

Theorem 10.3.6. *For all expressions $e, f \in$ Aexp and $n \in \mathbb{Z}$,*

$$\text{eval}(\text{subst}(f, e), n) = \text{eval}(e, \text{eval}(f, n)). \tag{10.7}$$

Proof. The proof is by structural induction on e.[1]

Base cases:

- Case[e is x]

  The left hand side of equation (10.7) equals $\text{eval}(f, n)$ by this base case in Definition 10.3.5 of the substitution function, and the right hand side also equals $\text{eval}(f, n)$ by this base case in Definition 10.3.4 of eval.

- Case[e is `k`].

  The left hand side of equation (10.7) equals `k` by this base case in Definitions 10.3.5 and 10.3.4 of the substitution and evaluation functions. Likewise, the right hand side equals `k` by two applications of this base case in the Definition 10.3.4 of eval.

Constructor cases:

- Case[e is $(e_1 + e_2)$]

  By the structural induction hypothesis (10.7), we may assume that for all $f \in$ Aexp and $n \in \mathbb{Z}$,

  $$\text{eval}(\text{subst}(f, e_i), n) = \text{eval}(e_i, \text{eval}(f, n)) \tag{10.8}$$

  for $i = 1, 2$. We wish to prove that

  $$\text{eval}(\text{subst}(f, (e_1 + e_2)), n) = \text{eval}((e_1 + e_2), \text{eval}(f, n)) \tag{10.9}$$

[1] This is an example of why it's useful to notify the reader what the induction variable is—in this case it isn't n.

But the left hand side of (10.9) equals

$$\text{eval}(\ (\text{subst}(f, e_1) + \text{subst}(f, e_2)),\ n)$$

by Definition 10.3.5.3 of substitution into a sum expression. But this equals

$$\text{eval}(\text{subst}(f, e_1), n) + \text{eval}(\text{subst}(f, e_2), n)$$

by Definition 10.3.4.3 of eval for a sum expression. By induction hypothesis (10.8), this in turn equals

$$\text{eval}(e_1, \text{eval}(f, n)) + \text{eval}(e_2, \text{eval}(f, n)).$$

Finally, this last expression equals the right hand side of (10.9) by Definition 10.3.4.3 of eval for a sum expression. This proves (10.9) in this case.

- e is $(e_1 * e_2)$. Similar.
- e is $-(e_1)$. Even easier.

This covers all the constructor cases, and so completes the proof by structural induction.

■

10.3.4 Problems

Practice Problems

Problem 10.1.

Definition. Consider a new recursive definition, MB_0, of the same set of "matching" brackets strings as MB (definition of MB is provided in the Appendix):

- **Base case:** $\lambda \in \text{MB}_0$.
- **Constructor cases:**

  (i) If s is in MB_0, then $[s]$ is in MB_0.

  (ii) If $s, t \in \text{MB}_0$, $s \neq \lambda$, and $t \neq \lambda$, then st is in MB_0.

(a) Suppose structural induction was being used to prove that $\text{MB}_0 \subseteq \text{MB}$. Circle the one predicate below that would fit the format for a structural induction hypothesis in such a proof.

- $P_0(n) ::= |s| \leq n$ IMPLIES $s \in \text{MB}$.
- $P_1(n) ::= |s| \leq n$ IMPLIES $s \in \text{MB}_0$.
- $P_2(s) ::= s \in \text{MB}$.

- $P_3(s) ::= s \in \text{MB}_0$.
- $P_4(s) ::= (s \in \text{MB IMPLIES } s \in \text{MB}_0)$.

(b) The recursive definition MB_0 is *ambiguous*. Verify this by giving two different derivations for the string "[ ][ ][ ]" according to MB_0.

Class Problems

Problem 10.2.
The Elementary 18.01 Functions (F18's) are the set of functions of one real variable defined recursively as follows:

Base cases:

- The identity function, $\text{id}(x) ::= x$ is an F18,
- any constant function is an F18,
- the sine function is an F18,

Constructor cases:
If f, g are F18's, then so are

1. $f + g$, fg, e^g (the constant e),
2. the inverse function $f^{(-1)}$,
3. the composition $f \circ g$.

(a) Prove that the function $1/x$ is an F18.

Warning: Don't confuse $1/x = x^{-1}$ with the inverse, $\text{id}^{(-1)}$ of the identity function $\text{id}(x)$. The inverse $\text{id}^{(-1)}$ is equal to id.

(b) Prove by Structural Induction on this definition that the Elementary 18.01 Functions are *closed under taking derivatives*. That is, show that if $f(x)$ is an F18, then so is $f' ::= df/dx$. (Just work out 2 or 3 of the most interesting constructor cases; you may skip the less interesting ones.)

Problem 10.3.
Here is a simple recursive definition of the set, E, of even integers:

Definition. Base case: $0 \in E$.
Constructor cases: If $n \in E$, then so are $n + 2$ and $-n$.

Provide similar simple recursive definitions of the following sets:

(a) The set $S ::= \left\{2^k 3^m 5^n \mid k, m, n \in \mathbb{N}\right\}$.

(b) The set $T ::= \left\{2^k 3^{2k+m} 5^{m+n} \mid k, m, n \in \mathbb{N}\right\}$.

(c) The set $L ::= \{(a, b) \in \mathbb{Z}^2 \mid 3 \mid (a - b)\}$.

Let L' be the set defined by the recursive definition you gave for L in the previous part. Now if you did it right, then $L' = L$, but maybe you made a mistake. So let's check that you got the definition right.

(d) Prove by structural induction on your definition of L' that

$$L' \subseteq L.$$

(e) Confirm that you got the definition right by proving that

$$L \subseteq L'.$$

(f) See if you can give an *unambiguous* recursive definition of L.

Problem 10.4.

Let p be the string [ ]. A string of brackets is said to be *erasable* iff it can be reduced to the empty string by repeatedly erasing occurrences of p. For example, here's how to erase the string [[[ ]] [ ]] [ ]:

$$[\,[\,[\,]\,]\,[\,]\,]\,[\,] \to [\,[\,]\,]\,[\,] \to [\,]\,[\,] \to \lambda.$$

On the other hand the string]] [[[[ ]] is not erasable because when we try to erase, we get stuck:

$$]\,]\,[\,[\,[\,[\,]\,]\,] \to]\,[\,[\,[\,]\,] \to]\,[\,[\,[\not\to$$

Let Erasable be the set of erasable strings of brackets. Let RecMatch be the recursive data type of strings of *matched* brackets given in Definition 10.3.7.

(a) Use structural induction to prove that

$$\text{RecMatch} \subseteq \text{Erasable}.$$

(b) Supply the missing parts of the following proof that

$$\text{Erasable} \subseteq \text{RecMatch}.$$

Proof. We prove by induction on the length, n, of strings, x, that if $x \in$ Erasable, then $x \in$ RecMatch. The induction predicate is

$$P(n) ::= \forall x \in \text{Erasable}.\ (|x| \leq n \ \text{IMPLIES}\ x \in \text{RecMatch})$$

Base case:

What is the base case? Prove that P is true in this case.

Inductive step: To prove $P(n+1)$, suppose $|x| \leq n+1$ and $x \in$ Erasable. We need only show that $x \in$ RecMatch. Now if $|x| < n+1$, then the induction hypothesis,

$P(n)$, implies that $x \in$ RecMatch, so we only have to deal with x of length exactly $n+1$.

Let's say that a string y is an *erase* of a string z iff y is the result of erasing a single occurrence of p in z.

Since $x \in$ Erasable and has positive length, there must be an erase, $y \in$ Erasable, of x. So $|y| = n-1$, and since $y \in$ Erasable, we may assume by induction hypothesis that $y \in$ RecMatch.

Now we argue by cases:

Case (y is the empty string).

Prove that $x \in$ RecMatch in this case.

Case ($y = [\, s\,]\, t$ for some strings $s, t \in$ RecMatch.) Now we argue by subcases.

- **Subcase** (x is of the form $[\, s'\,]\, t$ where s is an erase of s').
  Since $s \in$ RecMatch, it is erasable by part (b), which implies that $s' \in$ Erasable. But $|s'| < |x|$, so by induction hypothesis, we may assume that $s' \in$ RecMatch. This shows that x is the result of the constructor step of RecMatch, and therefore $x \in$ RecMatch.
- **Subcase** (x is of the form $[\, s\,]\, t'$ where t is an erase of t').
  Prove that $x \in$ RecMatch in this subcase.
- **Subcase**($x = p[\, s\,]\, t$).
  Prove that $x \in$ RecMatch in this subcase.

The proofs of the remaining subcases are just like this last one. **List these remaining subcases.**

This completes the proof by induction on n, so we conclude that $P(n)$ holds for all $n \in \mathbb{N}$. Therefore $x \in$ RecMatch for every string $x \in$ Erasable. That is,

$$\text{Erasable} \subseteq \text{RecMatch and hence Erasable} = \text{RecMatch}.$$

■

Problem 10.5.

Definition. The recursive data type, binary-2PTG, of *binary trees* with leaf labels, L, is defined recursively as follows:

- **Base case:** $\langle \texttt{leaf}, l \rangle \in$ binary-2PTG, for all labels $l \in L$.
- **Constructor case:** If $G_1, G_2 \in$ binary-2PTG, then

$$\langle \texttt{bintree}, G_1, G_2 \rangle \in \text{binary-2PTG}.$$

The *size*, $|G|$, of $G \in$ binary-2PTG is defined recursively on this definition by:

- **Base case:**
$$|\langle \texttt{leaf}, l\rangle| ::= 1, \quad \text{for all } l \in L.$$
- **Constructor case:**
$$|\langle \texttt{bintree}, G_1, G_2\rangle| ::= |G_1| + |G_2| + 1.$$

For example, for the size of the binary-2PTG, G, pictured in Figure 10.1, is 7.

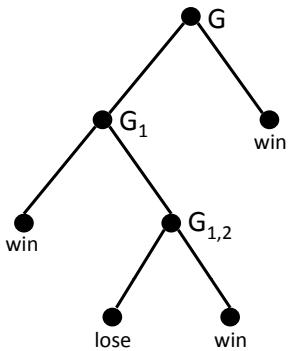


Figure 10.1: *A picture of a binary tree* w.

(a) Write out (using angle brackets and labels `bintree`, `leaf`, etc.) the binary-2PTG, G, pictured in Figure 10.1.

The value of flatten(G) for $G \in$ binary-2PTG is the sequence of labels in L of the leaves of G. For example, for the binary-2PTG, G, pictured in Figure 10.1,

$$\text{flatten}(G) = (\texttt{win}, \texttt{lose}, \texttt{win}, \texttt{win}).$$

(b) Give a recursive definition of flatten. (You may use the operation of *concatenation* (append) of two sequences.)

(c) Prove by structural induction on the definitions of flatten and size that

$$2 \cdot \text{length}(\text{flatten}(G)) = |G| + 1. \tag{10.10}$$

Homework Problems

Problem 10.6.

Definition 10.3.7. The set, RecMatch, of strings of matching brackets, is defined recursively as follows:

- **Base case:** $\lambda \in$ RecMatch.
- **Constructor case:** If $s, t \in$ RecMatch, then

$$[\, s \,]\, t \in \text{RecMatch}.$$

There is a simple test to determine whether a string of brackets is in RecMatch: starting with zero, read the string from left to right adding one for each left bracket and -1 for each right bracket. A string has a *good count* when the count never goes negative and is back to zero by the end of the string. Let GoodCount be the bracket strings with good counts.

(a) Prove that GoodCount contains RecMatch by structural induction on the definition of RecMatch.

(b) Conversely, prove that RecMatch contains GoodCount.

Problem 10.7.
Fractals are example of a mathematical object that can be defined recursively. In this problem, we consider the Koch snowflake. Any Koch snowflake can be constructed by the following recursive definition.

- Base Case: An equilateral triangle with a positive integer side length is a Koch snowflake.
- Recursive case: Let K be a Koch snowflake, and let l be a line segment on the snowflake. Remove the middle third of l, and replace it with two line segments of the same length as is done below:

The resulting figure is also a Koch snowflake.

Prove by structural induction that the area inside any Koch snowflake is of the form $q\sqrt{3}$, where q is a rational number.

10.4 Games as a Recursive Data Type

Chess, Checkers, and Tic-Tac-Toe are examples of *two-person terminating games of perfect information*, —2PTG's for short. These are games in which two players alternate moves that depend only on the visible board position or state of the game. "Perfect information" means that the players know the complete state of the game at each move. (Most card games are *not* games of perfect information because neither player can see the other's hand.) "Terminating" means that play cannot go on forever —it must end after a finite number of moves.[2]

We will define 2PTG's as a recursive data type. To see how this will work, let's use the game of Tic-Tac-Toe as an example.

10.4.1 Tic-Tac-Toe

Tic-Tac-Toe is a game for young children. There are two players who alternately write the letters "X" and "O" in the empty boxes of a 3×3 grid. Three copies of the same letter filling a row, column, or diagonal of the grid is called a *tic-tac-toe*, and the first player who gets a tic-tac-toe of their letter wins the game.

We're now going give a precise mathematical definition of the Tic-Tac-Toe *game tree* as a recursive data type.

Here's the idea behind the definition: at any point in the game, the "board position" is the pattern of X's and O's on the 3×3 grid. From any such Tic-Tac-Toe pattern, there are a number of next patterns that might result from a move. For example, from the initial empty grid, there are nine possible next patterns, each with a single X in some grid cell and the other eight cells empty. From any of these patterns, there are eight possible next patterns gotten by placing an O in an empty cell. These move possibilities are given by the game tree for Tic-Tac-Toe indicated in Figure 10.2.

Definition 10.4.1. A Tic-Tac-Toe *pattern* is a 3×3 grid each of whose 9 cells contains either the single letter, X, the single letter, O, or is empty.

A pattern, Q, is a *possible next pattern after* P, providing P has no tic-tac-toes and

- if P has an equal number of X's and O's, and Q is the same as P except that a cell that was empty in P has an X in Q, or
- if P has one more X than O's, and Q is the same as P except that a cell that was empty in P has an O in Q.

If P is a Tic-Tac-Toe pattern, and P has no next patterns, then the *terminated Tic-Tac-Toe game trees* at P are

- $\langle P, \langle \texttt{win} \rangle \rangle$, if P has a tic-tac-toe of X's.

[2] Since board positions can repeat in chess and checkers, termination is enforced by rules that prevent any position from being repeated more than a fixed number of times. So the "state" of these games is the board position *plus* a record of how many times positions have been reached.

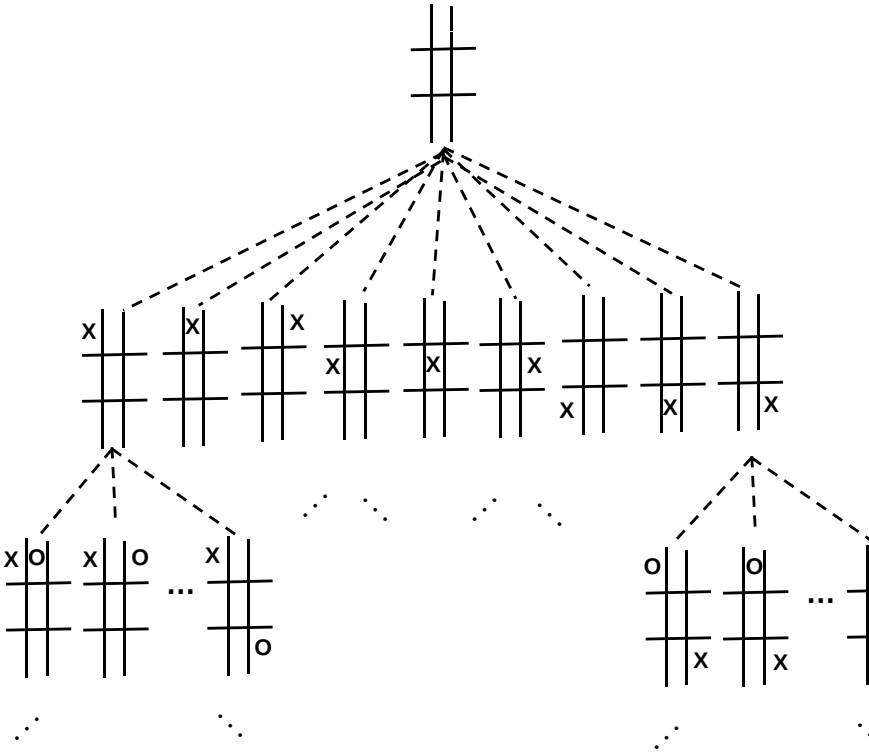


Figure 10.2: The Top of the Game Tree for Tic-Tac-Toe.

- $\langle P, \langle \texttt{lose} \rangle \rangle$, if P has a tic-tac-toe of O's.
- $\langle P, \langle \texttt{tie} \rangle \rangle$, otherwise.

The *Tic-Tac-Toe game trees starting at P* are defined recursively:

Base Case: A terminated Tic-Tac-Toe game tree at P is a Tic-Tac-Toe game tree starting at P.

Constructor case: If P is a non-terminated Tic-Tac-Toe pattern, then the Tic-Tac-Toe game tree starting at P consists of P and the set of all game trees starting at possible next patterns after P.

For example, if

$$P_0 = \begin{array}{|c|c|c|}\hline O & X & O \\ \hline X & O & X \\ \hline X & & \\ \hline \end{array}$$

$$Q_1 = \begin{array}{|c|c|c|}\hline O & X & O \\ \hline X & O & X \\ \hline X & & O \\ \hline \end{array}$$

$$Q_2 = \begin{array}{|c|c|c|}\hline O & X & O \\ \hline X & O & X \\ \hline X & O & \\ \hline \end{array}$$

$$R = \begin{array}{|c|c|c|}\hline O & X & O \\ \hline X & O & X \\ \hline X & O & X \\ \hline \end{array}$$

the game tree starting at P_0 is pictured in Figure 10.3.

Game trees are usually pictured in this way with the starting pattern (referred to as the "root" of the tree) at the top and lines connecting the root to the game trees that start at each possible next pattern. The "leaves" at the bottom of the tree (trees grow upside down in computer science) correspond to terminated games. A path from the root to a leaf describes a complete *play* of the game. (In English, "game" can be used in two senses: first we can say that Chess is a game, and second we can play a game of Chess. The first usage refers to the data type of Chess game trees, and the second usage refers to a "play.")

10.4.2 Infinite Tic-Tac-Toe Games

At any point in a Tic-Tac-Toe game, there are at most nine possible next patterns, and no play can continue for more than nine moves. But we can expand Tic-Tac-Toe into a larger game by running a 5-game tournament: play Tic-Tac-Toe five times and the tournament winner is the player who wins the most individual games. A 5-game tournament can run for as many as 45 moves.

It's not much of generalization to have an *n-game Tic-Tac-Toe tournament*. But then comes a generalization that sounds simple but can be mind-boggling: consolidate all these different size tournaments into a single game we can call *Tournament-Tic-Tac-Toe (T^4)*. The first player in a game of T^4 chooses any integer $n > 0$. Then

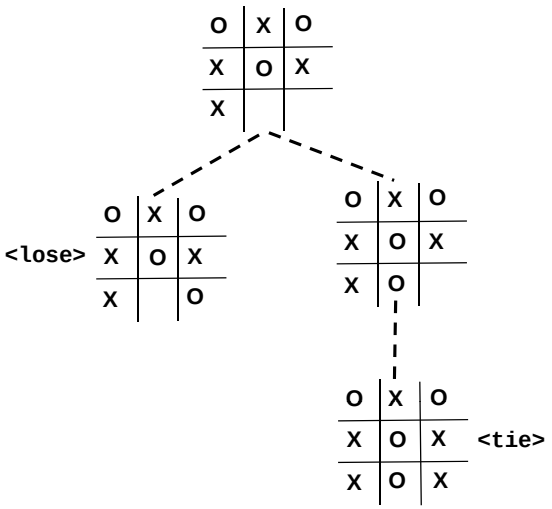


Figure 10.3: Game Tree for the Tic-Tac-Toe game starting at P_0.

the players play an n-game tournament. Now we can no longer say how long a T^4 play can take. In fact, there are T^4 plays that last as long as you might like: if you want a game that has a play with, say, nine billion moves, just have the first player choose n equal to one billion. This should make it clear the game tree for T^4 is infinite.

But still, it's obvious that every possible T^4 play will stop. That's because after the first player chooses a value for n, the game can't continue for more than $9n$ moves. So it's not possible to keep playing forever even though the game tree is infinite.

This isn't very hard to understand, but there is an important difference between any given n-game tournament and T^4: even though every play of T^4 must come to an end, there is no longer any initial bound on how many moves it might be before the game ends —a play might end after 9 moves, or $9(2001)$ moves, or $9(10^{10}+1)$ moves. It just can't continue forever.

Now that we recognize T^4 as a 2PTG, we can go on to a *meta-*T^4 game, where the first player chooses a number, $m > 0$, of T^4 games to play, and then the second player gets the first move in each of the individual T^4 games to be played.

Then, of course, there's meta-meta-T^4....

10.4.3 Two Person Terminating Games

Familiar games like Tic-Tac-Toe, Checkers, and Chess can all end in ties, but for simplicity we'll only consider win/lose games —no "everybody wins"-type games at MIT. :-) But everything we show about win/lose games will extend easily to games with ties, and more generally to games with outcomes that have different payoffs.

Like Tic-Tac-Toe, or Tournament-Tic-Tac-Toe, the idea behind the definition of 2PTG's as a recursive data type is that making a move in a 2PTG leads to the start of a subgame. In other words, given any set of games, we can make a new game whose first move is to pick a game to play from the set.

So what defines a game? For Tic-Tac-Toe, we used the patterns and the rules of Tic-Tac-Toe to determine the next patterns. But once we have a complete game tree, we don't really need the pattern labels: the root of a game tree itself can play the role of a "board position" with its possible "next positions" determined by the roots of its subtrees. So any game is defined by its game tree. This leads to the following very simple —perhaps deceptively simple —general definition.

Definition 10.4.2. The 2PTG, *game trees for two-person terminating games of perfect information* are defined recursively as follows:

- **Base cases:**
$$\begin{array}{ll} \langle \texttt{leaf}, \texttt{win} \rangle & \in \text{2PTG, and} \\ \langle \texttt{leaf}, \texttt{lose} \rangle & \in \text{2PTG.} \end{array}$$

- **Constructor case:** If $\mathcal{G}$ is a nonempty set of 2PTG's, then G is a 2PTG, where
$$G ::= \langle \texttt{tree}, \mathcal{G} \rangle .$$

The game trees in $\mathcal{G}$ are called the possible *next moves* from G.

These games are called "terminating" because, even though a 2PTG may be a (very) infinite datum like Tournament^2-Tic-Tac-Toe, every play of a 2PTG must terminate. This is something we can now prove, after we give a precise definition of "play":

Definition 10.4.3. A *play* of a 2PTG, G, is a (potentially infinite) sequence of 2PTG's starting with G and such that if G_1 and G_2 are consecutive 2PTG's in the play, then G_2 is a possible next move of G_1.

If a 2PTG has no infinite play, it is called a *terminating* game.

Theorem 10.4.4. *Every 2PTG is terminating.*

Proof. By structural induction on the definition of a 2PTG, G, with induction hypothesis

$$G \text{ is terminating.}$$

Base case: If $G = \langle \texttt{leaf}, \texttt{win} \rangle$ or $G = \langle \texttt{leaf}, \texttt{lose} \rangle$ then the only possible play of G is the length one sequence consisting of G. Hence G terminates.

Constructor case: For $G = \langle \texttt{tree}, \mathcal{G} \rangle$, we must show that G is terminating, given the Induction Hypothesis that *every* $G' \in \mathcal{G}$ is terminating.

But any play of G is, by definition, a sequence starting with G and followed by a play starting with some $G_0 \in \mathcal{G}$. But G_0 is terminating, so the play starting at G_0 is finite, and hence so is the play starting at G.

This completes the structural induction, proving that every 2PTG, G, is terminating. ■

10.4.4 Game Strategies

A key question about a game is whether a player has a winning strategy. A *strategy* for a player in a game specifies which move the player should make at any point in the game. A *winning* strategy ensures that the player will win no matter what moves the other player makes.

In Tic-Tac-Toe for example, most elementary school children figure out strategies for both players that each ensure that the game ends with no tic-tac-toes, that is, it ends in a tie. Of course the first player can win if his opponent plays childishly, but not if the second player follows the proper strategy. In more complicated games like Checkers or Chess, it's not immediately clear that anyone has a winning strategy, even if we agreed to count ties as wins for the second player.

But structural induction makes it easy to prove that in any 2PTG, *somebody* has the winning strategy!

Theorem 10.4.5. ***Fundamental Theorem for Two-Person Games:*** *For every two-person terminating game of perfect information, there is a winning strategy for one of the players.*

Proof. The proof is by structural induction on the definition of a 2PTG, G. The induction hypothesis is that there is a winning strategy for G.

Base cases:

1. $G = \langle \texttt{leaf}, \texttt{win} \rangle$. Then the first player has the winning strategy: "make the winning move."

2. $G = \langle \texttt{leaf}, \texttt{lose} \rangle$. Then the second player has a winning strategy: "Let the first player make the losing move."

Constructor case: Suppose $G = \langle \texttt{tree}, \mathcal{G} \rangle$. By structural induction, we may assume that some player has a winning strategy for each $G' \in \mathcal{G}$. There are two cases to consider:

- some $G_0 \in \mathcal{G}$ has a winning strategy for its second player. Then the first player in G has a winning strategy: make the move to G_0 and then follow the second player's winning strategy in G_0.

- every $G' \in \mathcal{G}$ has a winning strategy for its first player. Then the second player in G has a winning strategy: if the first player's move in G is to $G_0 \in \mathcal{G}$, then follow the winning strategy for the first player in G_0.

So in any case, one of the players has a winning strategy for G, which completes the proof of the constructor case.

It follows by structural induction that there is a winning strategy for every 2PTG, G. ■

Notice that although Theorem 10.4.5 guarantees a winning strategy, its proof gives no clue which player has it. For most familiar 2PTG's like Chess, Go, ..., no one knows which player has a winning strategy.[3]

10.4.5 Problems

Homework Problems

Problem 10.8.
Define *2-person 50-point games of perfect information*50-PG's, recursively as follows:

Base case: An integer, k, is a 50-PG for $-50 \leq k \leq 50$. This 50-PG called the *terminated game with payoff* k. A *play* of this 50-PG is the length one integer sequence, k.

Constructor case: If $G_0, \ldots, G_n$ is a finite sequence of 50-PG's for some $n \in \mathbb{N}$, then the following game, G, is a 50-PG: the possible first moves in G are the choice of an integer i between 0 and n, the possible second moves in G are the possible first moves in G_i, and the rest of the game G proceeds as in G_i.

[3]Checkers used to be in this list, but there has been a recent announcement that each player has a strategy that forces a tie. (reference TBA)

A *play* of the 50-PG, G, is a sequence of nonnegative integers starting with a possible move, i, of G, followed by a play of G_i. If the play ends at the game terminated game, k, then k is called the *payoff* of the play.

There are two players in a 50-PG who make moves alternately. The objective of one player (call him the *max*-player) is to have the play end with as high a payoff as possible, and the other player (called the *min*-player) aims to have play end with as low a payoff as possible.

Given which of the players moves first in a game, a strategy for the max-player is said to *ensure* the payoff, k, if play ends with a payoff of at least k, no matter what moves the min-player makes. Likewise, a strategy for the min-player is said to *hold down* the payoff to k, if play ends with a payoff of at most k, no matter what moves the max-player makes.

A 50-PG is said to have *max value*, k, if the max-player has a strategy that ensures payoff k, and the min-player has a strategy that holds down the payoff to k, when the *max-player moves first*. Likewise, the 50-PG has *min value*, k, if the max-player has a strategy that ensures k, and the min-player has a strategy that holds down the payoff to k, when the *min-player moves first*.

The *Fundamental Theorem* for 2-person 50-point games of perfect information is that is that every game has both a max value and a min value. (Note: the two values are usually different.)

What this means is that there's no point in playing a game: if the max player gets the first move, the min-player should just pay the max-player the max value of the game without bothering to play (a negative payment means the max-player is paying the min-player). Likewise, if the min-player gets the first move, the min-player should just pay the max-player the min value of the game.

(a) Prove this Fundamental Theorem for 50-valued 50-PG's by structural induction.

(b) A meta-50-PG game has as possible first moves the choice of *any* 50-PG to play. Meta-50-PG games aren't any harder to understand than 50-PG's, but there is one notable difference, they have an infinite number of possible first moves. We could also define meta-meta-50-PG's in which the first move was a choice of any 50-PG *or* the meta-50-PG game to play. In meta-meta-50-PG's there are an infinite number of possible first *and* second moves. And then there's meta$^3-$50-PG ....

To model such infinite games, we could have modified the recursive definition of 50-PG's to allow first moves that choose any one of an infinite sequence

$$G_0, G_1, \ldots, G_n, G_{n+1}, \ldots$$

of 50-PG's. Now a 50-PG can be a mind-bendingly infinite datum instead of a finite one.

Do these infinite 50-PG's still have max and min values? In particular, do you think it would be correct to use structural induction as in part (a) to prove a Fundamental Theorem for such infinite 50-PG's? Offer an answer to this question, and briefly indicate why you believe in it.

10.5 Induction in Computer Science

Induction is a powerful and widely applicable proof technique, which is why we've devoted two entire chapters to it. Strong induction and its special case of ordinary induction are applicable to any kind of thing with nonnegative integer sizes –which is a awful lot of things, including all step-by-step computational processes.

Structural induction then goes beyond natural number counting by offering a simple, natural approach to proving things about recursive computation and recursive data types. This makes it a technique every computer scientist should embrace.

Chapter 11

Planar Graphs

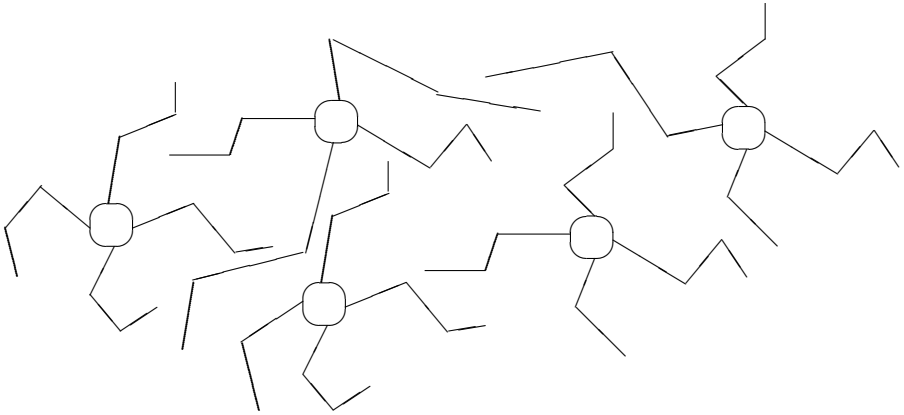
11.1 Drawing Graphs in the Plane

Here are three dogs and three houses.

Dog Dog Dog

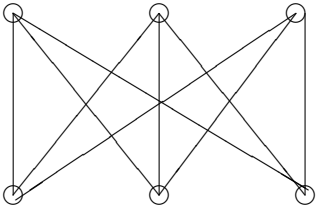
Can you find a path from each dog to each house such that no two paths intersect?

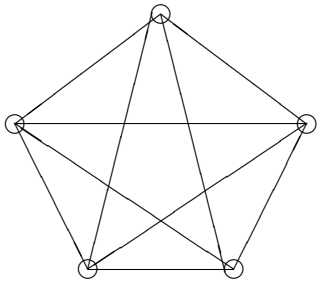
A *quadapus* is a little-known animal similar to an octopus, but with four arms. Here are five quadapi resting on the seafloor:



Can each quadapus simultaneously shake hands with every other in such a way that no arms cross?

Informally, a *planar graph* is a graph that can be drawn in the plane so that no edges cross, as in a map of showing the borders of countries or states. Thus, these two puzzles are asking whether the graphs below are planar; that is, whether they can be redrawn so that no edges cross. The first graph is called the *complete bipartite graph*, $K_{3,3}$, and the second is K_5.





In each case, the answer is, "No— but almost!" In fact, each drawing *would* be possible if any single edge were removed.

Planar graphs have applications in circuit layout and are helpful in displaying graphical data, for example, program flow charts, organizational charts, and scheduling conflicts. We will treat them as a recursive data type and use structural induction to establish their basic properties. Then we'll be able to describe a simple recursive procedure to color any planar graph with *five* colors, and also prove that there is no uniform way to place n satellites around the globe unless $n = 4, 6, 8, 12$, or 20.

When wires are arranged on a surface, like a circuit board or microchip, crossings require troublesome three-dimensional structures. When Steve Wozniak designed the disk drive for the early Apple II computer, he struggled mightly to achieve a nearly planar design:

> For two weeks, he worked late each night to make a satisfactory design. When he was finished, he found that if he moved a connector he could cut down on feedthroughs, making the board more reliable. To make that move, however, he had to start over in his design. This time it only took twenty hours. He then saw another feedthrough that could be eliminated, and again started over on his design. "The final design was generally recognized by computer engineers as brilliant and was by engineering aesthetics beautiful. Woz later said, 'It's something you can only do if you're the engineer and the PC board layout person yourself. That was an artistic layout. The board has virtually no feedthroughs.'"[a]

[a] From apple2history.org which in turn quotes *Fire in the Valley* by Freiberger and Swaine.

11.2 Continuous & Discrete Faces

Planar graphs are graphs that can be drawn in the plane —like familiar maps of countries or states. "Drawing" the graph means that each vertex of the graph corresponds to a distinct point in the plane, and if two vertices are adjacent, their vertices are connected by a smooth, non-self-intersecting curve. None of the curves may "cross" —the only points that may appear on more than one curve are the vertex points. These curves are the boundaries of connected regions of the plane called the *continuous faces* of the drawing.

For example, the drawing in Figure 11.1 has four continuous faces. Face IV, which extends off to infinity in all directions, is called the *outside face*.

This definition of planar graphs is perfectly precise, but completely unsatisfying: it invokes smooth curves and continuous regions of the plane to define a property of a discrete data type. So the first thing we'd like to find is a discrete data type that represents planar drawings.

The clue to how to do this is to notice that the vertices along the boundary of each of the faces in Figure 11.1 form a simple cycle. For example, labeling the vertices as in Figure 11.2, the simple cycles for the face boundaries are

`abca` `abda` `bcdb` `acda`.

Since every edge in the drawing appears on the boundaries of exactly two continuous faces, every edge of the simple graph appears on exactly two of the simple cycles.

Vertices around the boundaries of states and countries in an ordinary map are

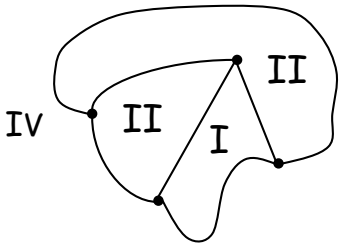


Figure 11.1: A Planar Drawing with Four Faces.

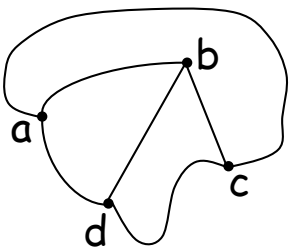


Figure 11.2: The Drawing with Labelled Vertices.

always simple cycles, but oceans are slightly messier. The ocean boundary is the set of all boundaries of islands and continents in the ocean; it is a *set* of simple cycles (this can happen for countries too —like Bangladesh). But this happens because islands (and the two parts of Bangladesh) are not connected to each other. So we can dispose of this complication by treating each connected component separately.

But general planar graphs, even when they are connected, may be a bit more complicated than maps. For example a planar graph may have a "bridge," as in Figure 11.3. Now the cycle around the outer face is

`abcefgecda`.

This is not a simple cycle, since it has to traverse the bridge c—e twice.

Planar graphs may also have "dongles," as in Figure 11.4. Now the cycle around the inner face is

`rstvxyxvwvtur`,

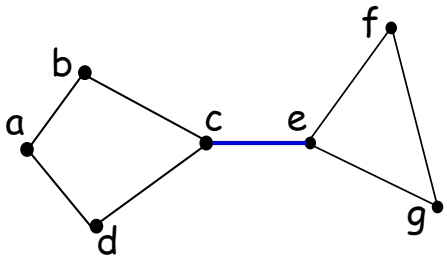


Figure 11.3: A Planar Drawing with a *Bridge*.

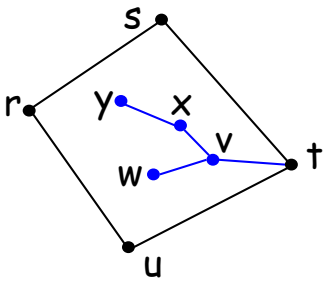


Figure 11.4: A Planar Drawing with a *Dongle*.

because it has to traverse *every* edge of the dongle twice —once "coming" and once "going."

But bridges and dongles are really the only complications, which leads us to the discrete data type of *planar embeddings* that we can use in place of continuous planar drawings. Namely, we'll define a planar embedding recursively to be the set of boundary-tracing cycles we could get drawing one edge after another.

11.3 Planar Embeddings

By thinking of the process of drawing a planar graph edge by edge, we can give a useful recursive definition of planar embeddings.

Definition 11.3.1. A *planar embedding* of a *connected* graph consists of a nonempty set of cycles of the graph called the *discrete faces* of the embedding. Planar embed-

dings are defined recursively as follows:

- **Base case:** If G is a graph consisting of a single vertex, v, then a planar embedding of G has one discrete face, namely the length zero cycle, v.
- **Constructor Case:** (split a face) Suppose G is a connected graph with a planar embedding, and suppose a and b are distinct, nonadjacent vertices of G that appear on some discrete face, γ, of the planar embedding. That is, γ is a cycle of the form

  $$a \ldots b \cdots a.$$

  Then the graph obtained by adding the edge a—b to the edges of G has a planar embedding with the same discrete faces as G, except that face γ is replaced by the two discrete faces[1]

  $$a \ldots ba \quad \text{and} \quad ab \cdots a,$$

  as illustrated in Figure 11.5.

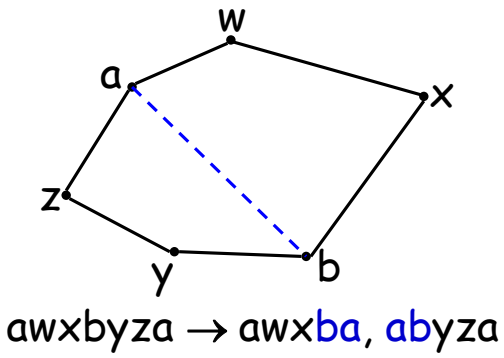


Figure 11.5: The Split a Face Case.

- **Constructor Case:** (add a bridge) Suppose G and H are connected graphs with planar embeddings and disjoint sets of vertices. Let a be a vertex on a discrete face, γ, in the embedding of G. That is, γ is of the form

  $$a \ldots a.$$

[1] There is one exception to this rule. If G is a line graph beginning with a and ending with b, then the cycles into which γ splits are actually the same. That's because adding edge a—b creates a simple cycle graph, C_n, that divides the plane into an "inner" and an "outer" region with the same border. In order to maintain the correspondence between continuous faces and discrete faces, we have to allow two "copies" of this same cycle to count as discrete faces. But since this is the only situation in which two faces are actually the same cycle, this exception is better explained in a footnote than mentioned explicitly in the definition.

Similarly, let b be a vertex on a discrete face, δ, in the embedding of H, so δ is of the form

$$b \cdots b.$$

Then the graph obtained by connecting G and H with a new edge, a—b, has a planar embedding whose discrete faces are the union of the discrete faces of G and H, except that faces γ and δ are replaced by one new face

$$a \ldots ab \cdots ba.$$

This is illustrated in Figure 11.6, where the faces of G and H are:

$$G : \{\texttt{axyza}, \texttt{axya}, \texttt{ayza}\} \qquad H : \{\texttt{btuvwb}, \texttt{btvwb}, \texttt{tuvt}\},$$

and after adding the bridge a—b, there is a single connected graph with faces

$$\{\texttt{axyzabtuvwba}, \texttt{axya}, \texttt{ayza}, \texttt{btvwb}, \texttt{tuvt}\}.$$

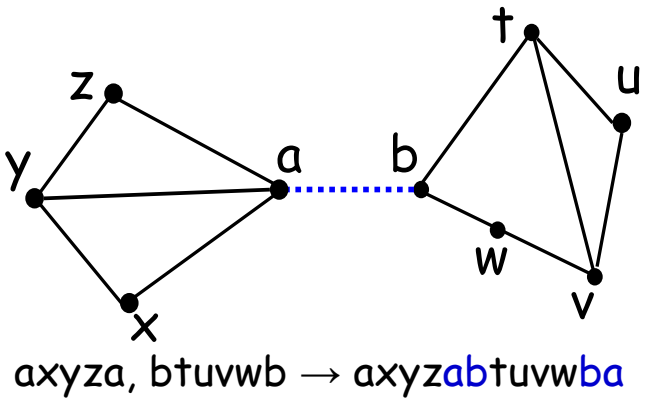


Figure 11.6: The Add Bridge Case.

An arbitrary graph is *planar* iff each of its connected components has a planar embedding.

11.4 What outer face?

Notice that the definition of planar embedding does not distinguish an "outer" face. There really isn't any need to distinguish one.

In fact, a planar embedding could be drawn with any given face on the outside. An intuitive explanation of this is to think of drawing the embedding on a *sphere* instead of the plane. Then any face can be made the outside face by "puncturing" that face of the sphere, stretching the puncture hole to a circle around the rest of the faces, and flattening the circular drawing onto the plane.

So pictures that show different "outside" boundaries may actually be illustrations of the same planar embedding.

This is what justifies the "add bridge" case in a planar embedding: whatever face is chosen in the embeddings of each of the disjoint planar graphs, we can draw a bridge between them without needing to cross any other edges in the drawing, because we can assume the bridge connects two "outer" faces.

11.5 Euler's Formula

The value of the recursive definition is that it provides a powerful technique for proving properties of planar graphs, namely, structural induction.

One of the most basic properties of a connected planar graph is that its number of vertices and edges determines the number of faces in every possible planar embedding:

Theorem 11.5.1 (Euler's Formula)**.** *If a connected graph has a planar embedding, then*

$$v - e + f = 2$$

where v is the number of vertices, e is the number of edges, and f is the number of faces.

For example, in Figure 11.1, $|V| = 4$, $|E| = 6$, and $f = 4$. Sure enough, $4-6+4 = 2$, as Euler's Formula claims.

Proof. The proof is by structural induction on the definition of planar embeddings. Let $P(\mathcal{E})$ be the proposition that $v - e + f = 2$ for an embedding, $\mathcal{E}$.

Base case: ($\mathcal{E}$ is the one vertex planar embedding). By definition, $v = 1$, $e = 0$, and $f = 1$, so $P(\mathcal{E})$ indeed holds.

Constructor case: (split a face) Suppose G is a connected graph with a planar embedding, and suppose a and b are distinct, nonadjacent vertices of G that appear on some discrete face, $\gamma = a \ldots b \cdots a$, of the planar embedding.

Then the graph obtained by adding the edge a—b to the edges of G has a planar embedding with one more face and one more edge than G. So the quantity $v - e + f$ will remain the same for both graphs, and since by structural induction this quantity is 2 for G's embedding, it's also 2 for the embedding of G with the added edge. So P holds for the constructed embedding.

Constructor case: (add bridge) Suppose G and H are connected graphs with planar embeddings and disjoint sets of vertices. Then connecting these two graphs with a bridge merges the two bridged faces into a single face, and leaves all other faces unchanged. So the bridge operation yields a planar embedding of a connected graph with $v_G + v_H$ vertices, $e_G + e_H + 1$ edges, and $f_G + f_H - 1$ faces. But

$$\begin{aligned}
&(v_G + v_H) - (e_G + e_H + 1) + (f_G + f_H - 1) \\
&= (v_G - e_G + f_G) + (v_H - e_H + f_H) - 2 \\
&= (2) + (2) - 2 && \text{(by structural induction hypothesis)} \\
&= 2.
\end{aligned}$$

So $v - e + f$ remains equal to 2 for the constructed embedding. That is, P also holds in this case.

This completes the proof of the constructor cases, and the theorem follows by structural induction. ■

11.6 Number of Edges versus Vertices

Like Euler's formula, the following lemmas follow by structural induction directly from the definition of planar embedding.

Lemma 11.6.1. *In a planar embedding of a connected graph, each edge is traversed once by each of two different faces, or is traversed exactly twice by one face.*

Lemma 11.6.2. *In a planar embedding of a connected graph with at least three vertices, each face is of length at least three.*

Corollary 11.6.3. *Suppose a connected planar graph has $v \geq 3$ vertices and e edges. Then*

$$e \leq 3v - 6.$$

Proof. By definition, a connected graph is planar iff it has a planar embedding. So suppose a connected graph with v vertices and e edges has a planar embedding with f faces. By Lemma 11.6.1, every edge is traversed exactly twice by the face boundaries. So the sum of the lengths of the face boundaries is exactly $2e$. Also by Lemma 11.6.2, when $v \geq 3$, each face boundary is of length at least three, so this sum is at least $3f$. This implies that

$$3f \leq 2e. \tag{11.1}$$

But $f = e - v + 2$ by Euler's formula, and substituting into (11.1) gives

$$\begin{aligned}
3(e - v + 2) &\leq 2e \\
e - 3v + 6 &\leq 0 \\
e &\leq 3v - 6
\end{aligned}$$

■

Corollary 11.6.3 lets us prove that the quadapi can't all shake hands without crossing. Representing quadapi by vertices and the necessary handshakes by edges, we get the complete graph, K_5. Shaking hands without crossing amounts to showing that K_5 is planar. But K_5 is connected, has 5 vertices and 10 edges, and $10 > 3 \cdot 5 - 6$. This violates the condition of Corollary 11.6.3 required for K_5 to be planar, which proves

Lemma 11.6.4. *K_5 is not planar.*

Another consequence is

Lemma 11.6.5. *Every planar graph has a vertex of degree at most five.*

Proof. If every vertex had degree at least 6, then the sum of the vertex degrees is at least $6v$, but since the sum equals $2e$, we have $e \geq 3v$ contradicting the fact that $e \leq 3v - 6 < 3v$ by Corollary 11.6.3. ■

11.7 Planar Subgraphs

If you draw a graph in the plane by repeatedly adding edges that don't cross, you clearly could add the edges in any other order and still wind up with the same drawing. This is so basic that we might presume that our recursively defined planar embeddings have this property. But that wouldn't be fair: we really need to prove it. After all, the recursive definition of planar embedding was pretty technical —maybe we got it a little bit wrong, with the result that our embeddings don't have this basic draw-in-any-order property.

Now any ordering of edges can be obtained just by repeatedly switching the order of successive edges, and if you think about the recursive definition of embedding for a minute, you should realize that you can switch *any* pair of successive edges if you can just switch the last two. So it all comes down to the following lemma.

Lemma 11.7.1. *Suppose that, starting from some embeddings of planar graphs with disjoint sets of vertices, it is possible by two successive applications of constructor operations to add edges e and then f to obtain a planar embedding, $\mathcal{F}$. Then starting from the same embeddings, it is also possible to obtain $\mathcal{F}$ by adding f and then e with two successive applications of constructor operations.*

We'll leave the proof of Lemma 11.7.1 to Problem 11.6.

Corollary 11.7.2. *Suppose that, starting from some embeddings of planar graphs with disjoint sets of vertices, it is possible to add a sequence of edges $e_0, e_1, \ldots, e_n$ by successive applications of constructor operations to obtain a planar embedding, $\mathcal{F}$. Then starting from the same embeddings, it is also possible to obtain $\mathcal{F}$ by applications of constructor operations that successively add any permutation[2] of the edges $e_0, e_1, \ldots, e_n$.*

[2]If $\pi : \{0, 1, \ldots, n\} \to \{0, 1, \ldots, n\}$ is a bijection, then the sequence $e_{\pi(0)}, e_{\pi(1)}, \ldots, e_{\pi(n)}$ is called a *permutation* of the sequence $e_0, e_1, \ldots, e_n$.

Corollary 11.7.3. *Deleting an edge from a planar graph leaves a planar graph.*

Proof. By Corollary 11.7.2, we may assume the deleted edge was the last one added in constructing an embedding of the graph. So the embedding to which this last edge was added must be an embedding of the graph without that edge. ■

Since we can delete a vertex by deleting all its incident edges, Corollary 11.7.3 immediately implies

Corollary 11.7.4. *Deleting a vertex from a planar graph, along with all its incident edges of course, leaves another planar graph.*

A *subgraph* of a graph, G, is any graph whose set of vertices is a subset of the vertices of G and whose set of edges is a subset of the set of edges of G. So we can summarize Corollaries 11.7.3 and 11.7.4 and their consequences in a Theorem.

Theorem 11.7.5. *Any subgraph of a planar graph is planar.*

11.8 Planar 5-Colorability

We need to know one more property of planar graphs in order to prove that planar graphs are 5-colorable.

Lemma 11.8.1. *Merging two adjacent vertices of a planar graph leaves another planar graph.*

Here merging two adjacent vertices, n_1 and n_2 of a graph means deleting the two vertices and then replacing them by a new "merged" vertex, m, adjacent to all the vertices that were adjacent to either of n_1 or n_2, as illustrated in Figure 11.7.

Lemma 11.8.1 can be proved by structural induction, but the proof is kind of boring, and we hope you'll be relieved that we're going to omit it. (If you insist, we can add it to the next problem set.)

Now we've got all the simple facts we need to prove 5-colorability.

Theorem 11.8.2. *Every planar graph is five-colorable.*

Proof. The proof will be by strong induction on the number, v, of vertices, with induction hypothesis:

> Every planar graph with v vertices is five-colorable.

Base cases ($v \leq 5$): immediate.

Inductive case: Suppose G is a planar graph with $v + 1$ vertices. We will describe a five-coloring of G.

First, choose a vertex, g, of G with degree at most 5; Lemma 11.6.5 guarantees there will be such a vertex.

Case 1 ($\deg(g) < 5$): Deleting g from G leaves a graph, H, that is planar by Lemma 11.7.4, and, since H has v vertices, it is five-colorable by induction hypothesis. Now define a five coloring of G as follows: use the five-coloring of H for all

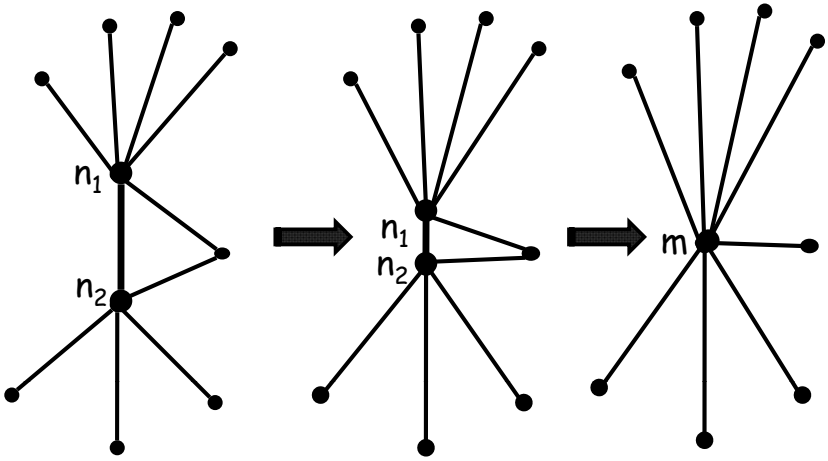


Figure 11.7: Merging adjacent vertices n_1 and n_2 into new vertex, m.

the vertices besides g, and assign one of the five colors to g that is not the same as the color assigned to any of its neighbors. Since there are fewer than 5 neighbors, there will always be such a color available for g.

Case 2 ($\deg(g) = 5$): If the five neighbors of g in G were all adjacent to each other, then these five vertices would form a nonplanar subgraph isomorphic to K_5, contradicting Theorem 11.7.5. So there must be two neighbors, n_1 and n_2, of g that are not adjacent. Now merge n_1 and g into a new vertex, m, as in Figure 11.7. In this new graph, n_2 is adjacent to m, and the graph is planar by Lemma 11.8.1. So we can then merge m and n_2 into a another new vertex, m', resulting in a new graph, G', which by Lemma 11.8.1 is also planar. Now G' has $v - 1$ vertices and so is five-colorable by the induction hypothesis.

Now define a five coloring of G as follows: use the five-coloring of G' for all the vertices besides g, n_1 and n_2. Next assign the color of m' in G' to be the color of the neighbors n_1 and n_2. Since n_1 and n_2 are not adjacent in G, this defines a proper five-coloring of G except for vertex g. But since these two neighbors of g have the same color, the neighbors of g have been colored using fewer than five colors altogether. So complete the five-coloring of G by assigning one of the five colors to g that is not the same as any of the colors assigned to its neighbors.

■

A graph obtained from a graph, G, be repeatedly deleting vertices, deleting edges, and merging adjacent vertices is called a *minor* of G. Since K_5 and $K_{3,3}$ are not planar, Lemmas 11.7.3, 11.7.4, and 11.8.1 immediately imply:

Corollary 11.8.3. *A graph which has K_5 or $K_{3,3}$ as a minor is not planar.*

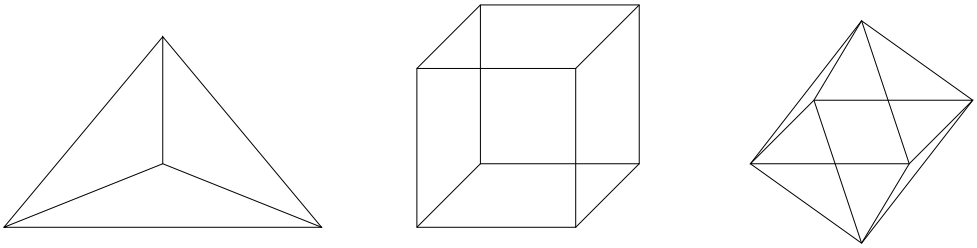
We don't have time to prove it, but the converse of Corollary 11.8.3 is also true. This gives the following famous, very elegant, and purely discrete characterization of planar graphs:

Theorem 11.8.4 (Kuratowksi)**.** *A graph is not planar iff it has K_5 or $K_{3,3}$ as a minor.*

11.9 Classifying Polyhedra

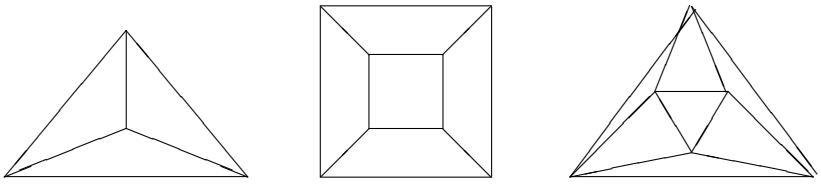
The Pythagoreans had two great mathematical secrets, the irrationality of $\sqrt{2}$ and a geometric construct that we're about to rediscover!

A *polyhedron* is a convex, three-dimensional region bounded by a finite number of polygonal faces. If the faces are identical regular polygons and an equal number of polygons meet at each corner, then the polyhedron is *regular*. Three examples of regular polyhedra are shown below: the tetrahedron, the cube, and the octahedron.



We can determine how many more regular polyhedra there are by thinking about planarity. Suppose we took *any* polyhedron and placed a sphere inside it. Then we could project the polyhedron face boundaries onto the sphere, which would give an image that was a planar graph embedded on the sphere, with the images of the corners of the polyhedron corresponding to vertices of the graph. But we've already observed that embeddings on a sphere are the same as embeddings on the plane, so Euler's formula for planar graphs can help guide our search for regular polyhedra.

For example, planar embeddings of the three polyhedra above look like this:



Let m be the number of faces that meet at each corner of a polyhedron, and let n be the number of sides on each face. In the corresponding planar graph, there are m edges incident to each of the v vertices. Since each edge is incident to two vertices, we know:

$$mv = 2e$$

Also, each face is bounded by n edges. Since each edge is on the boundary of two faces, we have:

$$nf = 2e$$

Solving for v and f in these equations and then substituting into Euler's formula gives:

$$\frac{2e}{m} - e + \frac{2e}{n} = 2$$

which simplifies to

$$\frac{1}{m} + \frac{1}{n} = \frac{1}{e} + \frac{1}{2} \tag{11.2}$$

This last equation (11.2) places strong restrictions on the structure of a polyhedron. Every nondegenerate polygon has at least 3 sides, so $n \geq 3$. And at least 3 polygons must meet to form a corner, so $m \geq 3$. On the other hand, if either n or m were 6 or more, then the left side of the equation could be at most $1/3 + 1/6 = 1/2$, which

is less than the right side. Checking the finitely-many cases that remain turns up only five solutions. For each valid combination of n and m, we can compute the associated number of vertices v, edges e, and faces f. And polyhedra with these properties do actually exist:

n	m	v	e	f	polyhedron
3	3	4	6	4	tetrahedron
4	3	8	12	6	cube
3	4	6	12	8	octahedron
3	5	12	30	20	icosahedron
5	3	20	30	12	dodecahedron

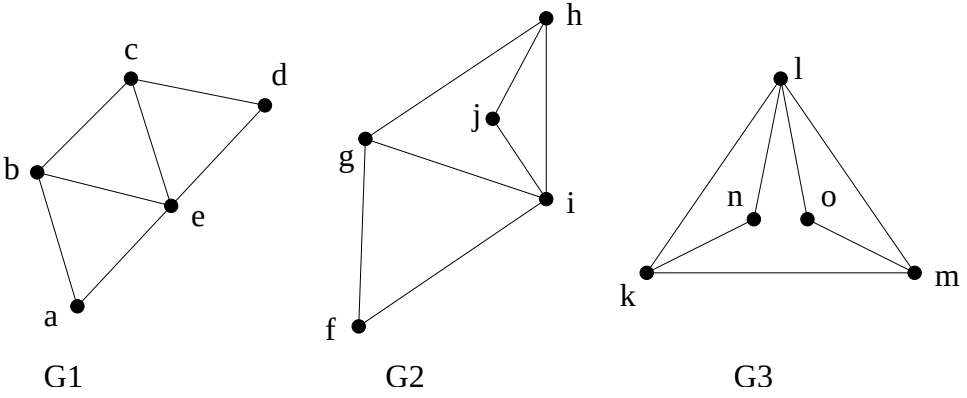
The last polyhedron in this list, the dodecahedron, was the other great mathematical secret of the Pythagorean sect. These five, then, are the only possible regular polyhedra.

So if you want to put more than 20 geocentric satellites in orbit so that they *uniformly* blanket the globe —tough luck!

11.9.1 Problems

Exam Problems

Problem 11.1.



(a) Describe an isomorphism between graphs G_1 and G_2, and another isomorphism between G_2 and G_3.

(b) Why does part (a) imply that there is an isomorphism between graphs G_1 and G_3?

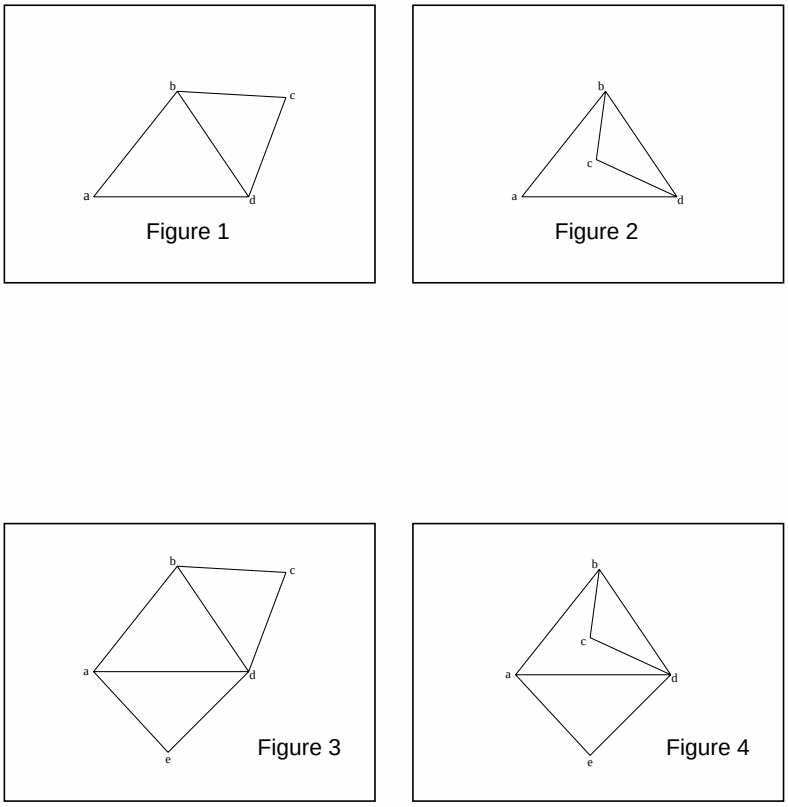
Let G and H be planar graphs. An embedding E_G of G is isomorphic to an embedding E_H of H iff there is an isomorphism from G to H that also maps each face of E_G to a face of E_H.

(c) One of the embeddings pictured above is not isomorphic to either of the others. Which one? Briefly explain why.

(d) Explain why all embeddings of two isomorphic planar graphs must have the same number of faces.

Class Problems

Problem 11.2.
Figures 1–4 show different pictures of planar graphs.



(a) For each picture, describe its discrete faces (simple cycles that define the region borders).

(b) Which of the pictured graphs are isomorphic? Which pictures represent the same *planar embedding*? – that is, they have the same discrete faces.

(c) Describe a way to construct the embedding in Figure 4 according to the recursive Definition 11.3.1 of planar embedding. For each application of a constructor rule, be sure to indicate the faces (cycles) to which the rule was applied and the cycles which result from the application.

Problem 11.3. (a) Show that if a connected planar graph with more than two vertices is bipartite, then

$$e \leq 2v - 4. \tag{11.3}$$

Hint: Similar to the proof of Corollary 11.6.3 that for planar graphs $e \leq 3v - 6$.

(b) Conclude that that $K_{3,3}$ is not planar. ($K_{3,3}$ is the graph with six vertices and an edge from each of the first three vertices to each of the last three.)

Problem 11.4.
Prove the following assertions by structural induction on the definition of planar embedding.

(a) In a planar embedding of a graph, each edge is traversed a total of two times by the faces of the embedding.

(b) In a planar embedding of a connected graph with at least three vertices, each face is of length at least three.

Homework Problems

Problem 11.5.
A simple graph is *triangle-free* when it has no simple cycle of length three.

(a) Prove for any connected triangle-free planar graph with $v > 2$ vertices and e edges, $e \leq 2v - 4$.

Hint: Similar to the proof that $e \leq 3v - 6$. Use Problem 11.4.

(b) Show that any connected triangle-free planar graph has at least one vertex of degree three or less.

(c) Prove by induction on the number of vertices that any connected triangle-free planar graph is 4-colorable.

Hint: use part (b).

Problem 11.6. (a) Prove Lemma 11.7.1. *Hint:* There are four cases to analyze, depending on which two constructor operations are applied to add e and then f. Structural induction is not needed.

(b) Prove Corollary 11.7.2.

Hint: By induction on the number of switches of adjacent elements needed to convert the sequence 0,1,. . . ,n into a permutation $\pi(0), \pi(1), \ldots, \pi(n)$.

Chapter 12

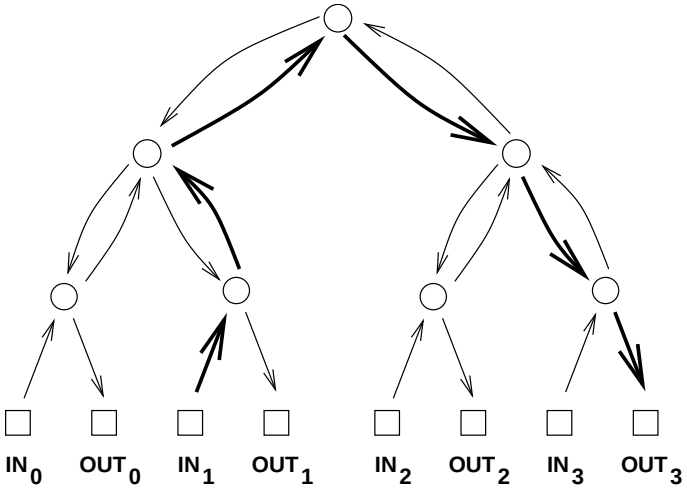
Communication Networks

12.1 Communication Networks

Modeling communication networks is an important application of digraphs in computer science. In this such models, vertices represent computers, processors, and switches; edges will represent wires, fiber, or other transmission lines through which data flows. For some communication networks, like the internet, the corresponding graph is enormous and largely chaotic. Highly structured networks, by contrast, find application in telephone switching systems and the communication hardware inside parallel computers. In this chapter, we'll look at some of the nicest and most commonly used structured networks.

12.2 Complete Binary Tree

Let's start with a *complete binary tree*. Here is an example with 4 inputs and 4 outputs.



The kinds of communication networks we consider aim to transmit packets of data between computers, processors, telephones, or other devices. The term *packet* refers to some roughly fixed-size quantity of data— 256 bytes or 4096 bytes or whatever. In this diagram and many that follow, the squares represent *terminals*, sources and destinations for packets of data. The circles represent *switches*, which direct packets through the network. A switch receives packets on incoming edges and relays them forward along the outgoing edges. Thus, you can imagine a data packet hopping through the network from an input terminal, through a sequence of switches joined by directed edges, to an output terminal.

Recall that there is a unique simple path between every pair of vertices in a tree. So the natural way to route a packet of data from an input terminal to an output in the complete binary tree is along the corresponding directed path. For example, the route of a packet traveling from input 1 to output 3 is shown in bold.

12.3 Routing Problems

Communication networks are supposed to get packets from inputs to outputs, with each packet entering the network at its own input switch and arriving at its own output switch. We're going to consider several different communication network designs, where each network has N inputs and N outputs; for convenience, we'll assume N is a power of two.

Which input is supposed to go where is specified by a permutation of $\{0, 1, \ldots, N-1\}$. So a permutation, π, defines a *routing problem*: get a packet that starts at input i to output $\pi(i)$. A *routing*, P, that *solves* a routing problem, π, is a set of paths from each input to its specified output. That is, P is a set of n paths, P_i, for $i = 0 \ldots, N-1$, where P_i goes from input i to output $\pi(i)$.

12.4 Network Diameter

The delay between the time that a packets arrives at an input and arrives at its designated output is a critical issue in communication networks. Generally this delay is proportional to the length of the path a packet follows. Assuming it takes one time unit to travel across a wire, the delay of a packet will be the number of wires it crosses going from input to output.

Generally packets are routed to go from input to output by the shortest path possible. With a shortest path routing, the worst case delay is the distance between the input and output that are farthest apart. This is called the *diameter* of the network. In other words, the diameter of a network[1] is the maximum length of any shortest path between an input and an output. For example, in the complete binary tree above, the distance from input 1 to output 3 is six. No input and output are farther apart than this, so the diameter of this tree is also six.

More generally, the diameter of a complete binary tree with N inputs and outputs is $2\log N+2$. (All logarithms in this lecture— and in most of computer science —are base 2.) This is quite good, because the logarithm function grows very slowly. We could connect up $2^{10} = 1024$ inputs and outputs using a complete binary tree and the worst input-output delay for any packet would be this diameter, namely, $2\log(2^{10}) + 2 = 22$.

12.4.1 Switch Size

One way to reduce the diameter of a network is to use larger switches. For example, in the complete binary tree, most of the switches have three incoming edges and three outgoing edges, which makes them 3×3 switches. If we had 4×4 switches, then we could construct a complete *ternary* tree with an even smaller diameter. In principle, we could even connect up all the inputs and outputs via a single monster $N \times N$ switch.

This isn't very productive, however, since we've just concealed the original network design problem inside this abstract switch. Eventually, we'll have to design the internals of the monster switch using simpler components, and then we're right back where we started. So the challenge in designing a communication network is figuring out how to get the functionality of an $N \times N$ switch using fixed size, elementary devices, like 3×3 switches.

12.5 Switch Count

Another goal in designing a communication network is to use as few switches as possible. The number of switches in a complete binary tree is $1+2+4+8+\cdots+N$, since there is 1 switch at the top (the "root switch"), 2 below it, 4 below those, and

[1]The usual definition of *diameter* for a general *graph* (simple or directed) is the largest distance between *any* two vertices, but in the context of a communication network we're only interested in the distance between inputs and outputs, not between arbitrary pairs of vertices.

so forth. By the formula (3.16) for geometric sums, the total number of switches is $2N - 1$, which is nearly the best possible with 3×3 switches.

12.6 Network Latency

We'll sometimes be choosing routings through a network that optimize some quantity besides delay. For example, in the next section we'll be trying to minimize packet congestion. When we're not minimizing delay, shortest routings are not always the best, and in general, the delay of a packet will depend on how it is routed. For any routing, the most delayed packet will be the one that follows the longest path in the routing. The length of the longest path in a routing is called its *latency*.

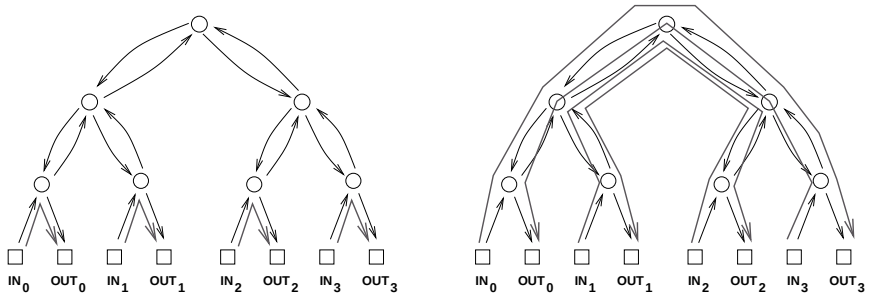
The latency of a *network* depends on what's being optimized. It is measured by assuming that optimal routings are always chosen in getting inputs to their specified outputs. That is, for each routing problem, π, we choose an optimal routing that solves π. Then *network latency* is defined to be the largest routing latency among these optimal routings. Network latency will equal network diameter if routings are always chosen to optimize delay, but it may be significantly larger if routings are chosen to optimize something else.

For the networks we consider below, paths from input to output are uniquely determined (in the case of the tree) or all paths are the same length, so network latency will always equal network diameter.

12.7 Congestion

The complete binary tree has a fatal drawback: the root switch is a bottleneck. At best, this switch must handle an enormous amount of traffic: every packet traveling from the left side of the network to the right or vice-versa. Passing all these packets through a single switch could take a long time. At worst, if this switch fails, the network is broken into two equal-sized pieces.

For example, if the routing problem is given by the identity permutation, $\mathrm{Id}(i) ::= i$, then there is an easy routing, P, that solves the problem: let P_i be the path from input i up through one switch and back down to output i. On the other hand, if the problem was given by $\pi(i) ::= (N-1) - i$, then in *any* solution, Q, for π, each path Q_i beginning at input i must eventually loop all the way up through the root switch and then travel back down to output $(N-1) - i$. These two situations are illustrated below.



We can distinguish between a "good" set of paths and a "bad" set based on congestion. The *congestion* of a routing, P, is equal to the largest number of paths in P that pass through a single switch. For example, the congestion of the routing on the left is 1, since at most 1 path passes through each switch. However, the congestion of the routing on the right is 4, since 4 paths pass through the root switch (and the two switches directly below the root). Generally, lower congestion is better since packets can be delayed at an overloaded switch.

By extending the notion of congestion to networks, we can also distinguish between "good" and "bad" networks with respect to bottleneck problems. For each routing problem, π, for the network, we assume a routing is chosen that optimizes congestion, that is, that has the minimum congestion among all routings that solve π. Then the largest congestion that will ever be suffered by a switch will be the maximum congestion among these optimal routings. This "maximin" congestion is called the *congestion of the network*.

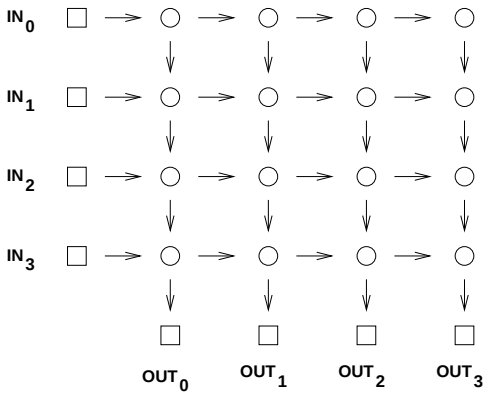
So for the complete binary tree, the worst permutation would be $\pi(i) ::= (N - 1) - i$. Then in every possible solution for π, *every* packet, would have to follow a path passing through the root switch. Thus, the max congestion of the complete binary tree is N —which is horrible!

Let's tally the results of our analysis so far:

network	diameter	switch size	# switches	congestion
complete binary tree	$2\log N + 2$	3×3	$2N - 1$	N

12.8 2-D Array

Let's look at an another communication network. This one is called a *2-dimensional array* or *grid*.



Here there are four inputs and four outputs, so $N = 4$.

The diameter in this example is 8, which is the number of edges between input 0 and output 3. More generally, the diameter of an array with N inputs and outputs is $2N$, which is much worse than the diameter of $2 \log N + 2$ in the complete binary tree. On the other hand, replacing a complete binary tree with an array almost eliminates congestion.

Theorem 12.8.1. *The congestion of an N-input array is 2.*

Proof. First, we show that the congestion is at most 2. Let π be any permutation. Define a solution, P, for π to be the set of paths, P_i, where P_i goes to the right from input i to column $\pi(i)$ and then goes down to output $\pi(i)$. Thus, the switch in row i and column j transmits at most two packets: the packet originating at input i and the packet destined for output j.

Next, we show that the congestion is at least 2. This follows because in any routing problem, π, where $\pi(0) = 0$ and $\pi(N-1) = N-1$, two packets must pass through the lower left switch. ■

As with the tree, the network latency when minimizing congestion is the same as the diameter. That's because all the paths between a given input and output are the same length.

Now we can record the characteristics of the 2-D array.

network	diameter	switch size	# switches	congestion
complete binary tree	$2 \log N + 2$	3×3	$2N - 1$	N
2-D array	$2N$	2×2	N^2	2

The crucial entry here is the number of switches, which is N^2. This is a major defect of the 2-D array; a network of size $N = 1000$ would require a *million* 2×2 switches! Still, for applications where N is small, the simplicity and low congestion of the array make it an attractive choice.

12.9 Butterfly

The Holy Grail of switching networks would combine the best properties of the complete binary tree (low diameter, few switches) and of the array (low congestion). The *butterfly* is a widely-used compromise between the two.

A good way to understand butterfly networks is as a recursive data type. The recursive definition works better if we define just the switches and their connections, omitting the terminals. So we recursively define F_n to be the switches and connections of the butterfly net with $N ::= 2^n$ input and output switches.

The base case is F_1 with 2 input switches and 2 output switches connected as in Figure 12.1.

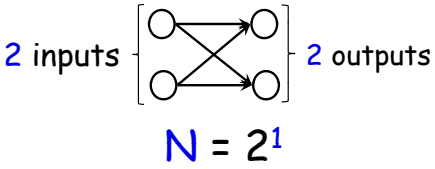


Figure 12.1: F_1, the Butterfly Net switches with $N = 2^1$.

In the constructor step, we construct F_{n+1} with 2^{n+1} inputs and outputs out of two F_n nets connected to a new set of 2^{n+1} input switches, as shown in as in Figure 12.2. That is, the ith and $2^n + i$th new input switches are each connected to the same two switches, namely, to the ith input switches of each of two F_n components for $i = 1, \ldots, 2^n$. The output switches of F_{n+1} are simply the output switches of each of the F_n copies.

So F_{n+1} is laid out in columns of height 2^{n+1} by adding one more column of switches to the columns in F_n. Since the construction starts with two columns when $n = 1$, the F_{n+1} switches are arrayed in $n + 1$ columns. The total number of switches is the height of the columns times the number of columns, namely, $2^{n+1}(n+1)$. Remembering that $n = \log N$, we conclude that the Butterfly Net with

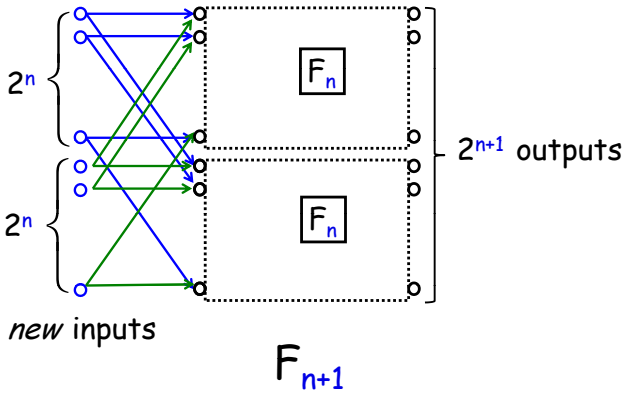


Figure 12.2: F_{n+1}, the Butterfly Net switches with 2^{n+1} inputs and outputs.

N inputs has $N(\log N + 1)$ switches.

Since every path in F_{n+1} from an input switch to an output is the same length, namely, $n+1$, the diameter of the Butterfly net with 2^{n+1} inputs is this length plus two because of the two edges connecting to the terminals (square boxes) —one edge from input terminal to input switch (circle) and one from output switch to output terminal.

There is an easy recursive procedure to route a packet through the Butterfly Net. In the base case, there is obviously only one way to route a packet from one of the two inputs to one of the two outputs. Now suppose we want to route a packet from an input switch to an output switch in F_{n+1}. If the output switch is in the "top" copy of F_n, then the first step in the route must be from the input switch to the unique switch it is connected to in the top copy; the rest of the route is determined by recursively routing the rest of the way in the top copy of F_n. Likewise, if the output switch is in the "bottom" copy of F_n, then the first step in the route must be to the switch in the bottom copy, and the rest of the route is determined by recursively routing in the bottom copy of F_n. In fact, this argument shows that the routing is *unique*: there is exactly one path in the Butterfly Net from each input to each output, which implies that the network latency when minimizing congestion is the same as the diameter.

The congestion of the butterfly network is about $\sqrt{N}$, more precisely, the congestion is $\sqrt{N}$ if N is an even power of 2 and $\sqrt{N/2}$ if N is an odd power of 2. A simple proof of this appears in Problem12.8.

Let's add the butterfly data to our comparison table:

network	diameter	switch size	# switches	congestion
complete binary tree	$2\log N + 2$	3×3	$2N - 1$	N
2-D array	$2N$	2×2	N^2	2
butterfly	$\log N + 2$	2×2	$N(\log(N) + 1)$	$\sqrt{N}$ or $\sqrt{N/2}$

The butterfly has lower congestion than the complete binary tree. And it uses fewer switches and has lower diameter than the array. However, the butterfly does not capture the best qualities of each network, but rather is a compromise somewhere between the two. So our quest for the Holy Grail of routing networks goes on.

12.10 Beneš Network

In the 1960's, a researcher at Bell Labs named Beneš had a remarkable idea. He obtained a marvelous communication network with congestion 1 by placing *two* butterflies back-to-back. This amounts to recursively growing *Beneš nets* by adding both inputs and outputs at each stage. Now we recursively define B_n to be the switches and connections (without the terminals) of the Beneš net with $N ::= 2^n$ input and output switches.

The base case, B_1, with 2 input switches and 2 output switches is exactly the same as F_1 in Figure 12.1.

In the constructor step, we construct B_{n+1} out of two B_n nets connected to a new set of 2^{n+1} input switches *and also* a new set of 2^{n+1} output switches. This is illustrated in Figure 12.3.

Namely, the ith and $2^n + i$th new input switches are each connected to the same two switches, namely, to the ith input switches of each of two B_n components for $i = 1, \ldots, 2^n$, exactly as in the Butterfly net. In addition, the ith and $2^n + i$th new *output* switches are connected to the same two switches, namely, to the ith output switches of each of two B_n components.

Now B_{n+1} is laid out in columns of height 2^{n+1} by adding two more columns of switches to the columns in B_n. So the B_{n+1} switches are arrayed in $2(n + 1)$ columns. The total number of switches is the number of columns times the height of the columns, namely, $2(n + 1)2^{n+1}$.

All paths in B_{n+1} from an input switch to an output are the same length, namely, $2(n + 1) - 1$, and the diameter of the Beneš net with 2^{n+1} inputs is this length plus two because of the two edges connecting to the terminals.

So Beneš has doubled the number of switches and the diameter, of course, but completely eliminates congestion problems! The proof of this fact relies on a clever induction argument that we'll come to in a moment. Let's first see how the Beneš

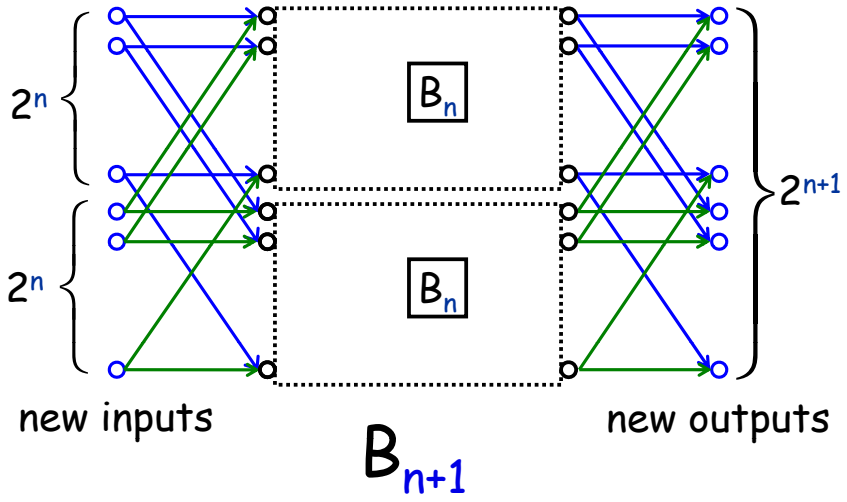


Figure 12.3: B_{n+1}, the Beneš Net switches with 2^{n+1} inputs and outputs.

network stacks up:

network	diameter	switch size	# switches	congestion
complete binary tree	$2\log N + 2$	3×3	$2N - 1$	N
2-D array	$2N$	2×2	N^2	2
butterfly	$\log N + 2$	2×2	$N(\log(N) + 1)$	$\sqrt{N}$ or $\sqrt{N/2}$
Beneš	$2\log N + 1$	2×2	$2N \log N$	1

The Beneš network has small size and diameter, and completely eliminates congestion. The Holy Grail of routing networks is in hand!

Theorem 12.10.1. *The congestion of the N-input Beneš network is 1.*

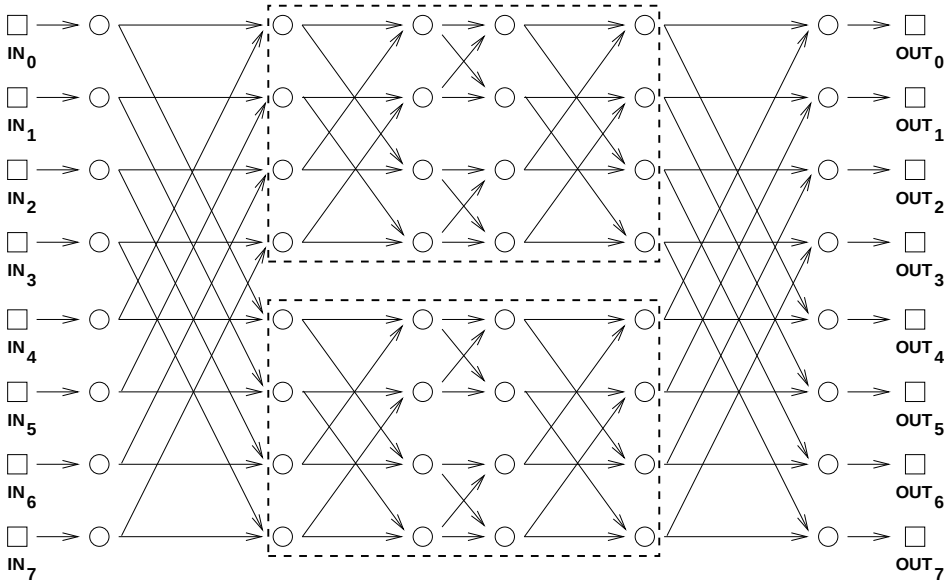
Proof. By induction on n where $N = 2^n$. So the induction hypothesis is

$$P(n) ::= \text{the congestion of } B_n \text{ is 1.}$$

Base case ($n = 1$): $B_1 = F_1$ and the unique routings in F_1 have congestion 1.

Inductive step: We assume that the congestion of an $N = 2^n$-input Beneš network is 1 and prove that the congestion of a $2N$-input Beneš network is also 1.

Digression. Time out! Let's work through an example, develop some intuition, and then complete the proof. In the Beneš network shown below with $N = 8$ inputs and outputs, the two 4-input/output subnetworks are in dashed boxes.

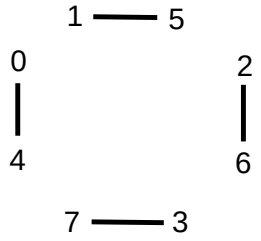


By the inductive assumption, the subnetworks can each route an arbitrary permutation with congestion 1. So if we can guide packets safely through just the first and last levels, then we can rely on induction for the rest! Let's see how this works in an example. Consider the following permutation routing problem:

$$\begin{aligned} \pi(0) &= 1 & \pi(4) &= 3 \\ \pi(1) &= 5 & \pi(5) &= 6 \\ \pi(2) &= 4 & \pi(6) &= 0 \\ \pi(3) &= 7 & \pi(7) &= 2 \end{aligned}$$

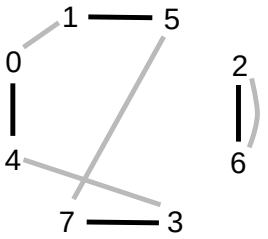
We can route each packet to its destination through either the upper subnetwork or the lower subnetwork. However, the choice for one packet may constrain the choice for another. For example, we can not route both packet 0 *and* packet 4 through the same network since that would cause two packets to collide at a single switch, resulting in congestion. So one packet must go through the upper network and the other through the lower network. Similarly, packets 1 and 5, 2 and 6, and 3

and 7 must be routed through different networks. Let's record these constraints in a graph. The vertices are the 8 packets. If two packets must pass through different networks, then there is an edge between them. Thus, our constraint graph looks like this:



Notice that at most one edge is incident to each vertex.

The output side of the network imposes some further constraints. For example, the packet destined for output 0 (which is packet 6) and the packet destined for output 4 (which is packet 2) can not both pass through the same network; that would require both packets to arrive from the same switch. Similarly, the packets destined for outputs 1 and 5, 2 and 6, and 3 and 7 must also pass through different switches. We can record these additional constraints in our graph with gray edges:



Notice that at most one new edge is incident to each vertex. The two lines drawn between vertices 2 and 6 reflect the two different reasons why these packets must be routed through different networks. However, we intend this to be a simple graph; the two lines still signify a single edge.

Now here's the key insight: *a 2-coloring of the graph corresponds to a solution to the routing problem*. In particular, suppose that we could color each vertex either red or blue so that adjacent vertices are colored differently. Then all constraints are satisfied if we send the red packets through the upper network and the blue packets through the lower network.

The only remaining question is whether the constraint graph is 2-colorable, which is easy to verify:

Lemma 12.10.2. *Prove that if the edges of a graph can be grouped into two sets such that every vertex has at most 1 edge from each set incident to it, then the graph is 2-colorable.*

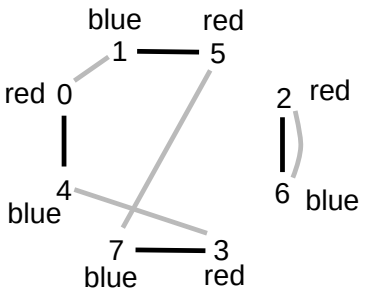
Proof. Since the two sets of edges may overlap, let's call an edge that is in both sets a *doubled edge*.

We know from Theorem 9.7.2 that all we have to do is show that every cycle has even length. There are two cases:

Case 1: [The cycle contains a doubled edge.] No other edge can be incident to either of the endpoints of a doubled edge, since that endpoint would then be incident to two edges from the same set. So a cycle traversing a doubled edge has nowhere to go but back and forth along the edge an even number of times.

Case 2: [No edge on the cycle is doubled.] Since each vertex is incident to at most one edge from each set, any path with no doubled edges must traverse successive edges that alternate from one set to the other. In particular, a cycle must traverse a path of alternating edges that begins and ends with edges from different sets. This means the cycle has to be of even length. ■

For example, here is a 2-coloring of the constraint graph:



The solution to this graph-coloring problem provides a start on the packet routing problem:

We can complete the routing in the two smaller Beneš networks by induction! Back to the proof. **End of Digression.**

Let π be an arbitrary permutation of $\{0, 1, \ldots, N-1\}$. Let G be the graph whose vertices are packet numbers $0, 1, \ldots, N-1$ and whose edges come from the union of these two sets:

$$E_1 ::= \{u — v \mid |u - v| = N/2\}, \text{ and}$$
$$E_2 ::= \{u — w \mid |\pi(u) - \pi(w)| = N/2\}.$$

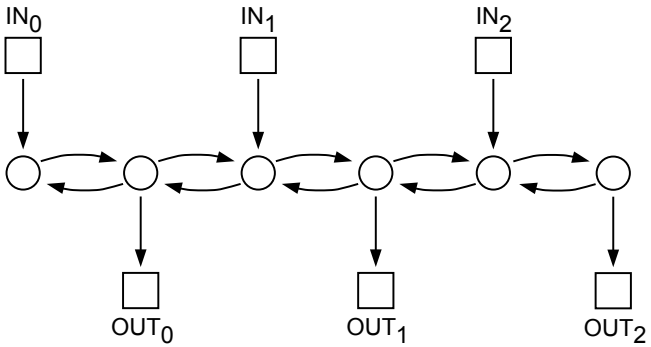
Now any vertex, u, is incident to at most two edges: a unique edge $u — v \in E_1$ and a unique edge $u — w \in E_2$. So according to Lemma 12.10.2, there is a 2-coloring for the vertices of G. Now route packets of one color through the upper subnetwork and packets of the other color through the lower subnetwork. Since for each

edge in E_1, one vertex goes to the upper subnetwork and the other to the lower subnetwork, there will not be any conflicts in the first level. Since for each edge in E_2, one vertex comes from the upper subnetwork and the other from the lower subnetwork, there will not be any conflicts in the last level. We can complete the routing within each subnetwork by the induction hypothesis $P(n)$. ■

12.10.1 Problems

Exam Problems

Problem 12.1.
Consider the following communication network:



(a) What is the max congestion? _____

(b) Give an input/output permutation, π_0, that forces maximum congestion:

$\pi_0(0) =$ _____ $\pi_0(1) =$ _____ $\pi_0(2) =$ _____

(c) Give an input/output permutation, π_1, that allows *minimum* congestion:

$\pi_1(0) =$ _____ $\pi_1(1) =$ _____ $\pi_1(2) =$ _____

(d) What is the latency for the permutation π_1? (If you could not find π_1, just choose a permutation and find its latency.) _____

Class Problems

Problem 12.2.
The Beneš network has a max congestion of 1; that is, every permutation can be routed in such a way that a single packet passes through each switch. Let's work through an example. A Beneš network of size $N = 8$ is attached.

(a) Within the Beneš network of size $N = 8$, there are two subnetworks of size $N = 4$. Put boxes around these. Hereafter, we'll refer to these as the *upper* and *lower* subnetworks.

(b) Now consider the following permutation routing problem:

$$\begin{aligned} \pi(0) &= 3 & \qquad \pi(4) &= 2 \\ \pi(1) &= 1 & \qquad \pi(5) &= 0 \\ \pi(2) &= 6 & \qquad \pi(6) &= 7 \\ \pi(3) &= 5 & \qquad \pi(7) &= 4 \end{aligned}$$

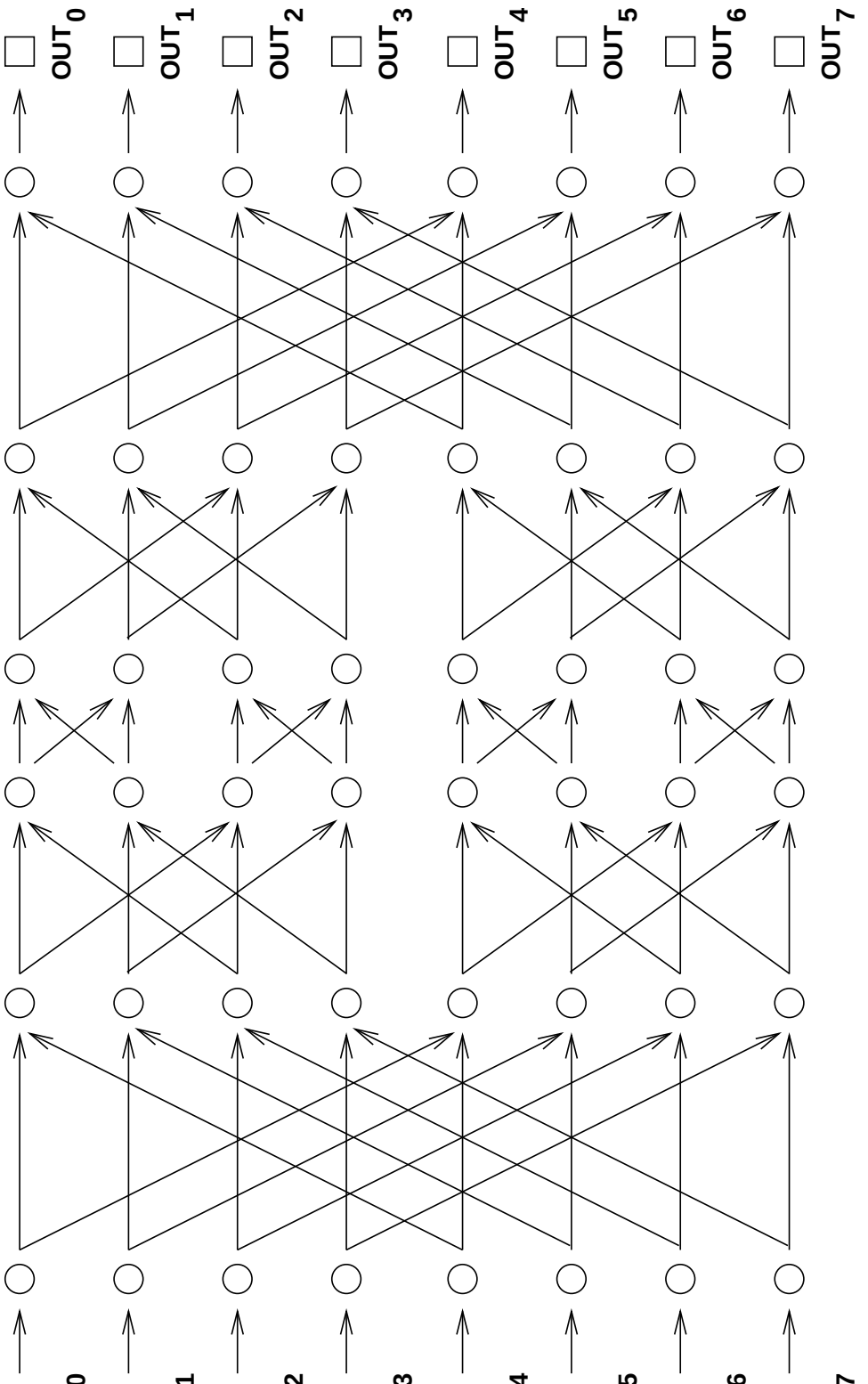
Each packet must be routed through either the upper subnetwork or the lower subnetwork. Construct a graph with vertices 0, 1, ..., 7 and draw a *dashed* edge between each pair of packets that can not go through the same subnetwork because a collision would occur in the second column of switches.

(c) Add a *solid* edge in your graph between each pair of packets that can not go through the same subnetwork because a collision would occur in the next-to-last column of switches.

(d) Color the vertices of your graph red and blue so that adjacent vertices get different colors. Why must this be possible, regardless of the permutation π?

(e) Suppose that red vertices correspond to packets routed through the upper subnetwork and blue vertices correspond to packets routed through the lower subnetwork. On the attached copy of the Beneš network, highlight the first and last edge traversed by each packet.

(f) All that remains is to route packets through the upper and lower subnetworks. One way to do this is by applying the procedure described above recursively on each subnetwork. However, since the remaining problems are small, see if you can complete all the paths on your own.


OUT0
OUT1
OUT2
OUT3
OUT4
OUT5
OUT6
OUT7

Problem 12.3.
A *multiple binary-tree network* has n inputs and n outputs, where n is a power of 2. Each input is connected to the root of a binary tree with $n/2$ leaves and with edges pointing away from the root. Likewise, each output is connected to the root of a binary tree with $n/2$ leaves and with edges pointing toward the root.

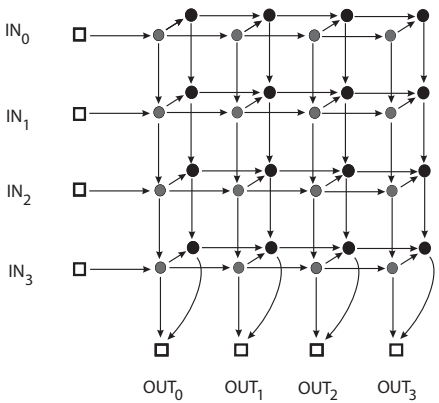
Two edges point from each leaf of an input tree, and each of these edges points to a leaf of an output tree. The matching of leaf edges is arranged so that for every input and output tree, there is an edge from a leaf of the input tree to a leaf of the output tree, and every output tree leaf has exactly two edges pointing to it.

(a) Draw such a multiple binary-tree net for $n = 4$.

(b) Fill in the table, and explain your entries.

# switches	switch size	diameter	max congestion

Problem 12.4.
The n-input *2-D Array* network was shown to have congestion 2. An n-input *2-Layer Array* consisting of two n-input 2-D Arrays connected as pictured below for $n = 4$.



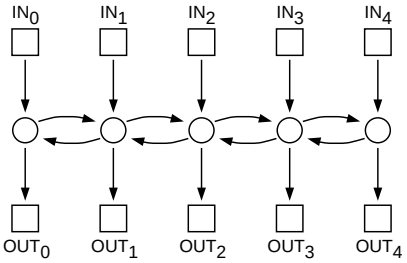
In general, an n-input 2-Layer Array has two layers of switches, with each layer connected like an n-input 2-D Array. There is also an edge from each switch in the first layer to the corresponding switch in the second layer. The inputs of the 2-Layer Array enter the left side of the first layer, and the n outputs leave from the bottom row of either layer.

(a) For any given input-output permutation, there is a way to route packets that achieves congestion 1. Describe how to route the packets in this way.

(b) What is the latency of a routing designed to minimize latency?

(c) Explain why the congestion of any minimum latency (CML) routing of packets through this network is greater than the network's congestion.

Problem 12.5.
A *5-path* communication network is shown below. From this, it's easy to see what an n-path network would be. Fill in the table of properties below, and be prepared to justify your answers.

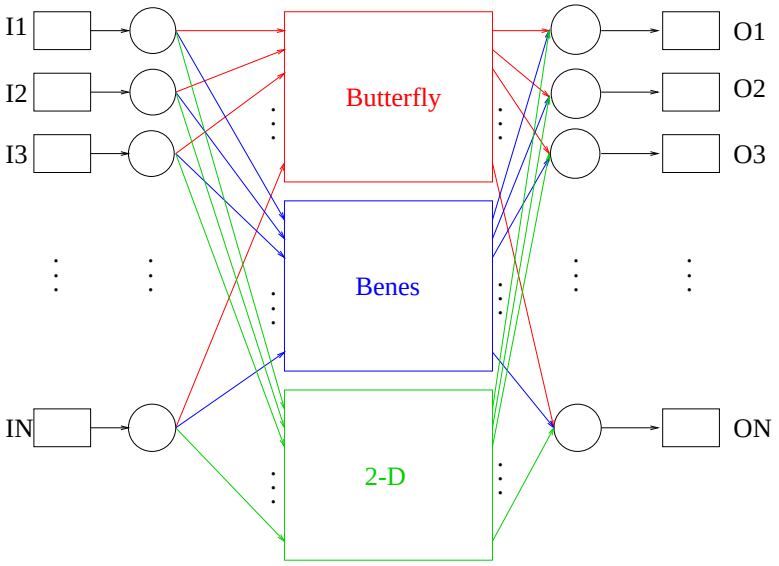


5-Path

network	# switches	switch size	diameter	max congestion
5-path				
n-path				

Problem 12.6.
Tired of being a TA, Megumi has decided to become famous by coming up with a new, better communication network design. Her network has the following specifications: every input node will be sent to a Butterfly network, a Benes network and a 2D Grid network. At the end, the outputs of all three networks will converge on the new output.

In the Megumi-net a minimum latency routing does not have minimum congestion. The *latency for min-congestion* (*LMC*) of a net is the best bound on latency achievable using routings that minimize congestion. Likewise, the *congestion for min-latency* (*CML*) is the best bound on congestion achievable using routings that minimize latency.



Fill in the following chart for Megumi's new net and explain your answers.

network	diameter	# switches	congestion	LMC	CML
Megumi's net					

Homework Problems

Problem 12.7.
Louis Reasoner figures that, wonderful as the Beneš network may be, the butterfly network has a few advantages, namely: fewer switches, smaller diameter, and an easy way to route packets through it. So Louis designs an N-input/output network he modestly calls a *Reasoner-net* with the aim of combining the best features of both the butterfly and Beneš nets:

> The ith input switch in a Reasoner-net connects to two switches, a_i and b_i, and likewise, the jth output switch has two switches, y_j and z_j, connected to it. Then the Reasoner-net has an N-input Beneš network connected using the a_i switches as input switches and the y_j switches as its output switches. The Reasoner-net also has an N-input butterfly net connected using the b_i switches as inputs and¡ the z_j switches as outputs.

In the Reasoner-net a minimum latency routing does not have minimum congestion. The *latency for min-congestion* (*LMC*) of a net is the best bound on latency achievable using routings that minimize congestion. Likewise, the *congestion for min-latency* (*CML*) is the best bound on congestion achievable using routings that minimize latency.

Fill in the following chart for the Reasoner-net and briefly explain your answers.

diameter	switch size(s)	# switches	congestion	LMC	CML

Problem 12.8.
Show that the congestion of the butterfly net, F_n, is exactly $\sqrt{N}$ when n is even.
Hint:

- There is a unique path from each input to each output, so the congestion is the maximum number of messages passing through a vertex for any routing problem.
- If v is a vertex in column i of the butterfly network, there is a path from exactly 2^i input vertices to v and a path from v to exactly 2^{n-i} output vertices.
- At which column of the butterfly network must the congestion be worst? What is the congestion of the topmost switch in that column of the network?

Part III

Counting

Part IV

Probability