ASILATA BAPAT AND ANAND DEOPURKAR

GAMES, GRAPHS, AND MACHINES

Contents

Some foundations

We begin by briefly introducing some language to talk about the objects we will encounter in this course. We will revisit this foundational material several times throughout the course in several contexts.

Sets

Informally, a *set* is an unordered collection of objects with no repetitions. This is the most basic object usually used to discuss almost every construction in mathematics. If T is a set and x is any object, we have the following dichotomy[1]: either x is an element of T, denoted $x \in T$, or x is not an element of T, denoted $x \notin T$. Two sets are equal if and only if they have the same elements. That is, every element of the first set is an element of the second set, and vice versa.

[1] A situation in which exactly one of two possible options is true.

The Zermelo–Fraenkel axioms can be used to develop this theory more formally, but we will not go into the details in this course.

Sets are often denoted by capital letters such as S, T, and potential elements as small letters x, y[2]. If we are listing all the elements of a set, we put them in curly braces, for example $\{1, 2, 3, 4\}$. We can also specify a set by taking all elements of another set that satisfy a particular property, for example $\{x \in \mathbf{N} \mid x \text{ is even}\}$.

[2] This is just a convention. In fact, sets are often elements of other sets, so there is no clear distinction between sets and potential elements.

A set S is a *subset* of a set T, denoted $S \subset T$, if every element of S is also an element of T. A set U is a *superset* of a set T, denoted $U \supset T$, if every element of T is also an element of U. There is a unique set that contains no elements. It is called *the empty set* and is denoted $\emptyset$. The empty set is vacuously[3] a subset of every set.

[3] We say that a statement of type "if ... then ...", or equivalently "for every ... we have ..." is *vacuously true* if nothing satisfies the "if" or "for every" condition.

The *size* or *cardinality* of a set is the number of elements in the set. If the number of elements is infinite, then we say that the set is infinite, and its cardinality is ∞.

Here are some things we can do with sets.

Unions The union of S and T, denoted $S \cup T$, is the set such that each element of $S \cup T$ is either an element of S or of T, or both.

Intersections The intersection of S and T, denoted $S \cap T$, is the set such that each element of $S \cap T$ is both an element of S and an element of T.

Difference The difference denoted $S - T$ is the set such that an element of $S - T$ is an element of S but not an element of T.

Example 1.

1. $\{1, 2\} \cup \{2, 3\} = \{1, 2, 3\}$.
2. $\{1, 2\} \cap \{2, 3\} = \{2\}$.

Cartesian products The Cartesian product of S and T, denoted $S \times T$, is the set whose elements are *ordered pairs* (x, y), where x runs over all the elements of S, and y runs over all the elements of T. Note that if one of the two sets is empty, then the Cartesian product is also empty.

Power set The power set of S, denoted $\mathcal{P}(S)$, is the set whose elements are all the subsets of S.

Example 2.

1. $\mathcal{P}(\{1,2\}) = \{\emptyset, \{1\}, \{2\}, \{1,2\}\}$.
2. $\{1,2\} \times \{2,3\} = \{(1,2), (1,3), (2,2), (2,3)\}$.
3. $\{1,2\} \times \emptyset = \emptyset$.

Functions

A function f from a set S to a set T is a rule that takes elements of S as input and produces elements of T as output. We write $f\colon S \to T$ to say that f is a function from S to T. Usually, after writing $f\colon S \to T$, we specify the rule. For an input element $s \in S$, we denote by $f(s)$ the output (in T) produced by f for the input s. For f to be a valid function, the rule that defines it must be *unambiguous*, that is, for every input $s \in S$, it must produce a unique output $f(s)$. Furthermore, the rule cover all possible input values $s \in S$.

If we have a function $f\colon S \to T$, we say that S is the *source* or *domain* of f and T is the *target* or *co-domain* of f.

Example 3. *1. We have a function $f\colon \mathbf{N} \to \mathbf{N}$ defined by $f(s) = s^2$.*

2. The rule $s \mapsto s/2$ does not *define a function $f\colon \mathbf{N} \to \mathbf{N}$ because for inputs s that are odd numbers, the output $s/2$ is not an element of $\mathbf{N}$. However, it does define a function $f\colon \mathbf{N} \to \mathbf{Q}$.*

Relations

Informally, a relation is a specification that links objects of one set and objects of another set. If x is related to y under a relation R, we say that the ordered pair (x, y) satisfies R. For example, we may consider a relation called `is-factor-of`, on pairs of natural numbers, which specifies that (x, y) satisfies `is-factor-of` if and only if x is a factor of y. In this case, $(1, 3)$, $(3, 27)$, $(4, 24)$ are all examples of ordered pairs that satisfy the relation `is-factor-of`[4].

[4] In English, we might read one of these as "3 is a factor of 27".

To MODEL THIS MATHEMATICALLY, we formally define a relation as a subset $R \subset S \times T$, where S and T are two sets. In this case, the elements of R are precisely the ordered pairs that we think of as satisfying the relation R. In the previous example, we have $S = T = \mathbf{N}$. If we want R to model the relation `is-factor-of`, then we take R to be the subset of $\mathbf{N} \times \mathbf{N}$ consisting of exactly the pairs (x, y) where x is a factor of y.

As in the previous example, we often want S and T to be the same set. In this case, we say that a subset $R \subset S \times S$ is a (binary[5]) relation on S.

[5] This is a binary relation because we are looking at a subset of the product of two copies of S. An n-ary relation on S would just be a subset of the product of n copies of S.

A FUNCTION GIVES RISE TO A RELATION, which we can think of as the `input-output` relation of the function. Given a function $f\colon S \to T$, the *input-output relation* of f is the relation $R \subset S \times T$ defined by $R = \{(s, t) \in S \times T \mid t = f(s)\}$. In other words, we think of s and t as related if t is the output given by f for the input s.

The input-output relation R associated to a function f satisfies an important property. For every $s \in S$, there is a *unique* $t \in T$ such that $(s,t) \in R$. This is another way of saying that for every input, f produces a unique output. If a relation does not satisfy this property, then it cannot be the input-output relation of a function.

Example 4.

1. *The relation* $\{(a,b) \in \mathbf{N} \times \mathbf{N} \mid a+b \text{ is even}\}$ *is not the input-ouput relation of a function because, for example, for the element* $2 \in \mathbf{N}$*, we have two elements* 0 *and* 4 *of* $\mathbf{N}$ *such that* $(2,0)$ *and* $(2,4)$ *are related.*
2. *The relation* $\{(a,b) \in \mathbf{N} \times \mathbf{N} \mid b = a^2\}$ *is the input-output relation of a function. The function is* $f\colon \mathbf{N} \to \mathbf{N}$ *given by* $f(a) = a^2$*.*

Graphs

Graphs provide an extremely useful way to organise information about relations. For the moment we use them as powerful visual aids, but we will see later that graphs also lend themselves well to computational tools.

A *directed graph* consists of a *vertex set* V and an *edge set* E. We require that the edge set E is a relation on V, that is, $E \subset V \times V$. We will write this graph as (V,E). Visually, we draw the vertices as nodes and an edge (v,w) as an arrow from v to w.

We think of *undirected graph* as a directed graph with the extra property that the edge relation E is symmetric. That is, $(v,w) \in E$ if and only if $(w,v) \in E$. In this case, we draw the vertices as nodes, and we draw a single segment joining v and w for every corresponding pair of edges (v,w) and (w,v).

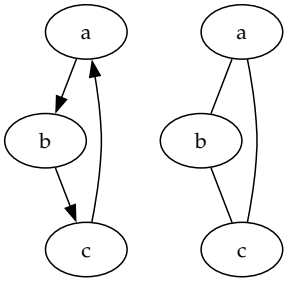


Figure 1: A directed and an undirected graph

Representing a relation on a set as a graph

Note that the definition of a graph is very similar to the definition of a relation on a single set — in fact, a directed graph is just another way of looking at a relation on a set. More precisely, let R be a relation on a set S. Then we can construct a directed graph whose vertex set is S and whose edge set is R. This point of view is useful in certain situations, as we will see later.

The adjacency matrix of a graph

Recall that a *matrix* is a rectangular array, usually filled with numbers. An $m \times n$ matrix M has m rows (numbered 1 through m) and n columns (numbered 1) through n). The entry in the ith row and jth column is denoted M_{ij}.

It is extremely useful to encode the data of a graph into a matrix, called an *adjacency matrix*.

Definition 5. *Suppose* $G = (V,E)$ *is a graph*[6]*. Choose an ordering on the elements of* V*, say the ordered tuple* $(v_1, \ldots, v_n)$*. The adjacency matrix of* G *with respect to the chose ordering is the* $n \times n$ *matrix* A*, defined by*

$$A_{ij} = \begin{cases} 1, & (i,j) \in E, \\ 0, & (i,j) \notin E \end{cases}.$$

[6] For simplicity we usually consider finite sets V when we construct adjacency matrices but in general V may be infinite.

The adjacency matrix is a matrix that only contains the elements 0 and 1. It encodes the entire information contained in the original

Example 6. *Let* (V,E) *be the directed graph shown in Figure 1, with the ordering on the vertices chosen to be* (a,b,c)*. Then the adjacency matrix is*

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}.$$

Now if we reorder the vertices as (c,b,a)*,*

graph, in a way that is highly adapted to calculations — we will see more of this soon.

Note that changing the ordering on the elements of V produces a different-looking adjacency matrix. It is related to the original adjacency matrix by a serious of simultaneous swaps of corresponding row and column numbers. For example, the adjacency matrix given by the ordering $(v_2, v_1, v_3, \ldots, v_n)$ can be obtained from A by swapping rows 1 and 2 and also swapping columns 1 and 2.

Properties of relations

Sometimes, relations (on a single set) satisfy further special properties. Here are some common ones. Remember that a relation R is simply a subset of $S \times S$ for some set S. So the following properties are about R as a whole, as a subset of $S \times S$.

Reflexivity A relation R is *reflexive* if $(x, x) \in R$ for each $x \in S$.

Symmetry A relation R is *symmetric* if whenever we have $(x, y) \in R$, we also have $(y, x) \in R$.

Anti-symmetry A relation R is *anti-symmetric* if having both $(x, y) \in R$ and $(y, x) \in R$ implies that $x = y$.

Transitivity A relation R is *transitive* if whenever $(x, y) \in R$ and $(y, z) \in R$, we also have $(x, z) \in R$.

Note that the properties of being *symmetric* and *anti-symmetric* are almost but not quite complementary to each other: if a relation is both symmetric and anti-symmetric, it means that only pairs of the form (x, x) can be in the relation[7]. However, not all pairs of this form have to satisfy the relation (i.e. the relation need not be reflexive).

The adjacency matrix can be helpful in order to read off properties about the relation. For example, since a reflexive relation has all possible pairs (x, x) in it, all diagonal entries A_{ii} of the adjacency matrix must equal 1, and conversely if $A_{ii} = 1$ for each i, then the relation is reflexive.

Similarly, a relation is symmetric if $A_{ij} = A_{ji}$ for each i, j. That is, if the adjacency matrix is symmetric. A relation is anti-symmetric if whenever $i \neq j$ and $A_{ij} = 1$, we have $A_{ji} = 0$.

What does it mean in terms of the adjacency matrix if a relation is transitive? The answer to this question is slightly more complicated, and we will get back to it later.

Example 7.

1. *The relation*

   $$R = \{(a, b) \in \mathbf{N} \times \mathbf{N} \mid a \text{ divides } b\}$$

   is reflexive, anti-symmetric, and transitive.

2. *The relation*

   $$R = \{(a, b) \in \mathbf{N} \times \mathbf{N} \mid a + b \text{ is odd}\}$$

   is symmetric but not reflexive or transitive.

[7] Convince yourself of this from the definitions!

Closures of relations

If S is any set, then the entire cartesian product $S \times S$ is itself a relation on S. Note that certain properties are true for $S \times S$: for example, of the four properties discussed in the previous section, $S \times S$ has reflexivity, symmetry, and transitivity.

If R is any relation on S, it makes sense to ask about the *reflexive closure* (resp. symmetric or transitive closure) of R. In the following discussion we'll talk about the reflexive closure, but you can use the same definition for symmetric and transitive closures respectively.

Informally, we'd like the reflexive closure of R to be the smallest relation on S that contains R, and which is reflexive. If R is already reflexive, then it is its own reflexive closure. Otherwise, the reflexive closure will contain some more elements. But what does *smallest* mean in the above context[8]? To make this precise, we give the following definition.

[8] If S is a finite set, then we can say that that smallest means the one with the least number of elements, but we give a general definition because we don't want to be restricted to this case.

Definition 8. *A reflexive (resp. symmetric, transitive) closure of R is a set $\overline{R}$ with the following properties.*

1. $R \subset \overline{R} \subset S \times S$.
2. $\overline{R}$ *is reflexive (resp. symmetric, transitive).*
3. *If T is a subset of $S \times S$ such that $R \subset T \subsetneq \overline{R}$, then T is not reflexive (resp. symmetric, transitive).*

It can be shown that reflexive (resp. symmetric, transitive) closures always exist, and that they are unique[9]. We won't prove this formally, but instead we will just produce a construction of each.

[9] Think about when it makes sense to ask for the closure of a relation with respect to a property, and when you can expect it to exist uniquely. For example, it doesn't really make sense to ask for the anti-symmetric closure of a relation. Do you see why?

Let us first tackle the reflexive closure. To make a relation reflexive, we need to add in all pairs of the form $\{(x, x)\}$, where $x \in S$. So you can convince yourself that the reflexive closure is simply the set $R \cup \{(x, x) \mid x \in S\}$: not only is this new relation reflexive, but also if you take away any pair that is not already an element of R, you get something non-reflexive. In terms of adjacency matrices, the reflexive closure is the relation corresponding to the matrix obtained by changing all diagonal entries of the original adjacency matrix to 1.

Similarly, the *symmetric closure* of R is obtained by adding the flipped pair $\{(b, a)\}$ for every pair $(a, b) \in R$. This is the same thing as taking $R \cup \{(a, b) \mid (b, a) \in R\}$. In terms of the adjacency matrix, we obtain this by symmetrising the adjacency matrix[10]: whenever $A_{ij} = 1$, we also set $A_{ji} = 1$.

[10] This is the same as taking $\frac{1}{2}(A + A^t)$. Do you see why?

Once again, it is not so easy to describe how to construct the *transitive closure* of a relation R, but it can be done by developing some techniques for working with adjacency matrices. We will revisit this later once we have those techniques.

Equivalence relations

Recall that a relation R on a set S is just a subset of the product $S \times S$. We take a short tour through the theory of equivalence relations, which are extremely important in constructing all sorts of mathematical structures.

Definition 9. *A* equivalence relation *is a relation that is reflexive, symmetric, and transitive.*

Example 10. *Let R be the relation on $\mathbf{Z}$ defined as*

$$R = \{(a, b) \in \mathbf{Z} \times \mathbf{Z} \mid a - b \text{ is even}\}.$$

Usually, if we have an equivalence relation R on a set S, we say that $x \sim_R y$ if (x, y) is in R. If the context is clear, we will simply say $x \sim y$. The most important application is that having an equivalence relation on a set allows us to treat an object x as "being equivalent" to an object y if $x \sim y$: the equivalence relation gives us a new way of identifying various objects. We will capture this identification with the notion of *equivalence classes*[11].

[11] The idea is that we can treat all elements of one equivalence class as being interchangeable in some sense.

Definition 11. *Let R be an equivalence relation on a set S. For any $x \in S$, the* equivalence class *of x, denoted $[x]$, is the subset of S defined as follows:*

$$[x] = \{y \in S \mid x \sim_R y\}.$$

In Example 10, $a \sim b$ if and only if they have the same parity, so there are two equivalence classes of R on $\mathbf{Z}$, namely $[0]$ and $[1]$. Note that $[0]$ is the same as $[2]$ or $[-6]$, and $[1]$ is the same as $[-55]$ or $[7]$, but it's traditional to use the smallest non-negative values, which are $[0]$ and $[1]$.

The special properties that an equivalence relation satisfies guarantees the following proposition.

Proposition 12. *Let R be an equivalence relation on a set S.*

1. *Every element of S belongs to at least one equivalence class (its own!).*
2. *If $x, y \in S$ such that $y \in [x]$, then $[x] = [y]$. In other words, the set of equivalence classes of an equivalence relation partitions*[12] *the set S into disjoint subsets whose union is S.*

[12] If $S = S_1 \cup \cdots \cup S_n$, we say that it is a *partition* if $S_i \cap S_j = \emptyset$ for $i \neq j$. In this case we write $S = S_1 \sqcup \cdots \sqcup S_n$, or more concisely, $S = \bigsqcup_{i=1}^{n} S_i$.

Proof. Let x be any element of S. First note that $x \in [x]$ by reflexivity, which proves the first statement. To prove the second statement, suppose that $x, y \in S$ such that $y \in [x]$. To show that $[x] = [y]$, we need to show that for every $z \in S$, we have $z \in [x]$ if and only if $z \in [y]$.

Recall that $y \in [x]$ means that $x \sim_R y$. If $z \in [y]$, then we have $z \in [x]$ by transitivity: $x \sim_R y$ and $y \sim_R z$ implies $x \sim_R z$. On the

other hand, since we know that $y \in [x]$, we also have $x \in [y]$ by symmetry, and then by the previous argument we see that if $z \in [x]$ then $z \in [y]$ by transitivity. The proof is now complete. □

If $y \in [x]$, we say that y is a *representative* of $[x]$.

Often we can uncover new structures by working with the set of equivalence classes rather than the original set S, and it can even give rise to new structures. An important example of this technique is modular arithmetic.

Modular arithmetic

As an important application of equivalence classes, we briefly study modular arithmetic. First recall the relation from Example 10. We can observe that in the integers, the sum of two numbers is always even. The sum of an even with an odd is odd, and the sum of two odd numbers is always odd. But the set of even numbers has another name: $[0]$, and the set of odd numbers is also called $[1]$ with respect to this relation.

So we can express the above statements by writing down the following statements instead.

1. Whenever $a \in [0]$ and $b \in [0]$, we have $a + b \in [0]$.
2. Whenever $a \in [0]$ and $b \in [1]$, we have $a + b \in [1]$.
3. Whenever $a \in [1]$ and $b \in [0]$, we have $a + b \in [1]$.
4. Whenever $a \in [1]$ and $b \in [1]$, we have $a + b \in [0]$.

Let us instead express this by defining a *new addition operation* on the set[13] $\{[0], [1]\}$. We will simply define this addition using the four properties above, which can be written more concisely as

$$[a] + [b] \coloneqq [a + b] \text{ for each } a, b \in \mathbf{Z}.$$

[13] Note that this set is *not* equal to $\mathbf{Z}$! It is also not equal to the set $\{0, 1\}$. Instead this is a set with two elements, which are themselves subsets of $\mathbf{Z}$.

Because we know the properties we stated above about even/odd addition, we have effectively proven that it actually doesn't matter whic representative we take for each equivalence class. This is the idea behind modular arithmetic.

More generally, fix a *modulus* $d \in \mathbf{N}$. We say that $x \sim_d y$ if $x - y$ is divisible by d, which is also written as $d \mid x - y$. More traditionally, we write $x \equiv y \pmod{d}$. Note that if $x \sim_d y$, then there is some integer $m \in \mathbf{Z}$ such that $x - y = md$.

Exercise 13. *Check that $\sim_d$ is an equivalence relation.*

In this case, we have equivalence classes $[0], [1], \ldots, [d-1]$. Note that $[d] = [0]$ again. But if $0 \leq e, f < d$, how do we know for sure that $[e] \neq [f]$ when $e \neq f$? We know this by Euclid's algorithm, which guarantees that for every integer n and positive integer d, we can write a *unique* equation

$$n = qd + r, \quad 0 \leq r < d.$$

In our case, suppose that $e \geq f$. Since $0 \leq e - f < d$, the equation for $e - f$ has to be $e - f = 0 \cdot d + (e - f)$. On the other hand

if $[e] = [f]$ then we also have a valid equation that looks like $e - f = m \cdot d + 0$ for some m. Matching up the two, we see that $m = 0$ and $e = f$ is the only possibility.

Having established this, we now know that we have exactly d different equivalence classes, namely $[0], [1], \ldots, [d-1]$. Of course these can be represented by different integers. For example, $[1] = \{\ldots, 1-2d, 1-d, 1, 1+d, 1+2d, \ldots\}$, so any of these elements would do as a representative of $[1]$. We will write $\mathbf{Z}/d\mathbf{Z} = \{[0], \ldots, [d-1]\}$ to be the set of equivalence classes in this case.

Once again we define a *new addition operation*, this time on $\mathbf{Z}/d\mathbf{Z}$. The definition is the same: for any $[a], [b] \in \mathbf{Z}/d\mathbf{Z}$, set

$$[a] + [b] \coloneqq [a + b].$$

We now have to check whether this is *well-defined*[14] Suppose that $[p] = [a]$ and $[q] = [b]$. Then $p - a = md$ and $q - b = nd$ for some integers m, n. Adding these, we see that $(p+q) - (a+b) = (m+n)d$, and so $[p+q] = [a+b]$. Indeed, our operation is well-defined! This is called modular addition.

[14] This means that if $[p] = [a]$ and $[q] = [b]$, do we have $[p+q] = [a+b]$? If not, we don't have a good definition because it depends on the specific representative we had chosen!

Notice that this has properties similar to the addition in the integers, with some key differences. For example, we have the following.

similarity $[0] + [a] = [a] + [0] = [a]$

similarity $[a] + [b] = [b] + [a]$

difference! $[a] + [a] + \cdots + [a]$ can equal $[0]$ even if $[a] \neq 0$. For example, $[1] + [1] + [1] = [0]$ when $d = 3$.

What about multiplication? Can we define a modular multiplication? Let us try. We will attempt to define a multiplication operation by saying that

$$[a] \cdot [b] \text{ should b } [ab].$$

Again, we must check that this is well-defined. Suppose that $[p] = [a]$ and $[q] = [b]$. Then $p - a = md$ and $q - b = nd$ for some integers m, n. Note that $pq - aq = mqd$ and $aq - ab = nad$. Adding these, we see that $pq - ab = (mq + na)d$, so $[pq] = [ab]$, and this multiplication is well-defined! This is called modular multiplication.

Exercise 14. *What are some similarities and differences between modular multiplication and usual integer multiplication?*

Partial orders

In this section we study another important kind of relation, called *partial orders*. These are entirely different in flavour from equivalence relations, and very common.

Definition 15. *A relation R on a set P is a* partial ordering *or* partial order *if it is reflexive, anti-symmetric, and transitive.*

A set equipped with a partial order relation is called a *partially ordered set* or a *poset*. If R is a partial order on P, we usually write $x \preceq y$ if $(x, y) \in R$. We also often say that $(P, \preceq)$ is a poset, to mean that $\preceq$ is a partial order relation on the set P.

Here is an example of a non-numerical partial ordering.

Note that a partially ordered set P need not be a set of numbers, so the curly inequality sign denoting the partial order relation is not necessarily a numerical inequality.

Example 16. *Suppose that S is any set, and let $\mathcal{P}(S)$ be the power set of S, so that the elements of $\mathcal{P}(S)$ are all the subsets of S. We can define a partial ordering on $\mathcal{P}(S)$ by setting $A \preceq B$ whenever $A \subseteq B$. Let us check the three properties.*

1. *This relation is reflexive because any set A is a subset of itself.*

2. *It is anti-symmetric because if $A \subseteq B$ and $B \subseteq A$ both hold, then all elements of A are elements of B and all elements of B are elements of A, and so A and B must be equal.*

3. *It is transitive because whenever $A \subseteq B$ and $B \subseteq C$, we also have $A \subseteq C$.*

Suppose that $\preceq$ is a partial order on some set P.

Definition 18. *We say that two elements $a, b \in P$ are* comparable *if they are related in some order, that is, either $a \preceq b$ or $b \preceq a$.*

Let $\preceq$ be a partial order on a set P. We say that this partial order is *total* if any two elements a, b of P are comparable. That is, we either have $a \preceq b$ or $b \preceq a$.

Here are a couple of other important examples of partial orderings.

- The usual inequality ordering on $\mathbf{N}$, $\mathbf{Z}$, $\mathbf{Q}$, or $\mathbf{R}$, where $a \preceq b$ whenever $a \leq b$ as numbers. This is a total order because any two numbers are comparable.

- The division ordering on $\mathbf{N}$, where $a \preceq b$ whenever $a \mid b$, that is, a is a factor of b. This is not a total order, because (for example) 12 and 15 are incomparable.

Exercise 17. *Find examples to show that the partial order of Example 16 is not usually a total order.*

Exercise 19. *Check that the examples given satisfy the properties of being partial orders, and come up with some more of your own.*

Hasse diagrams

A Hasse diagram is a useful way to visualise a partial order. It is similar to drawing the graph of the partial order, but much less cluttered. Let us consider the example in Figure 2.

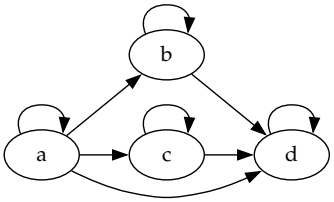


Figure 2: The graph of a partial order relation

This is the graph of the relation, which contains all the information about the relation. But it is also highly redundant: we already know that partial order relations are reflexive, so the self-loops are redundant. Similarly, we alreday know that the relation is transitive, so any "shortcuts", such as the one from the node a to the node d, are redundant.

So to convert the graph of a partial order relation into a Hasse diagram, we do the following:

- remove all self-loops,
- remove all edges implied by transitivity, and
- implicitly order all edges from bottom to top to get rid of the arrowheads.

The result can be seen in Figure 3.

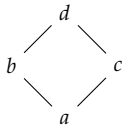


Figure 3: The Hasse diagram of the partial order relation from Figure 2.

Similarly, to convert from a Hasse diagram to the graph of the relation, we do the following:

- add arrowheads going from the bottom to the top,
- add all edges in the transitive closure, and
- add self-loops at each vertex.

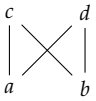


Figure 4: The Hasse diagram of the "bow-tie" poset

Upper and lower bounds

Let $(P, \preceq)$ be a partially ordered set. Let $A \subseteq P$ be a subset.

Definition 20. *We say that $u \in P$ is an* upper bound for A *if for every $a \in A$, we have $a \preceq u$. We say that $l \in P$ is a* lower bound for A *if for every $a \in A$, we have $l \preceq a$.*

Note that upper and lower bounds may not be unique, and they may not even lie in the set A, as can be seen from Example 21.

Definition 22. *Let $(P, \preceq)$ be a poset. We say that u is the* least upper bound *or* lub *for a subset $A \subseteq P$ if it is the smallest among all upper bounds of A. That is, if u' is any upper bound for A, then we have $u \preceq u'$. We say that l is the* greatest lower bound *or* glb *for a subset $A \subseteq P$ if it is the greatest among all lower bounds of A. That is, if l' is any lower bound for A, then we have $l' \preceq l$.*

Note that lubs and glbs need not always exist, again as demonstrated in Example 21. However, if they exist, they are unique.

Exercise 23. *Let A be a subset of a poset $(P, \preceq)$ and suppose that A has a least upper bound $u \in P$. Show that it is the unique least upper bound for A in P. Similarly, if A has a greatest lower bound l, then show that it is the unique greatest lower bound for A in P.*

Example 21. *Let $(P, \preceq)$ be the* bow-tie poset *shown in Figure 4. We have the following.*

1. *The set $\{a, b\}$ has two upper bounds (c and d), but no lub.*
2. *The set $\{a, b\}$ has no lower bound.*
3. *The set $\{a, b, c, d\}$ has no upper or lower bound.*
4. *The set $\{a, b, c\}$ has c as its unique upper bound (and hence its unique lub), and no lower bounds.*

Definition 24. *Let $(P, \preceq)$ be a poset. We have the following definitions.*

1. *An element $x \in P$ is called a* minimum element *of P if $x \preceq y$ for every $y \in P$.*
2. *An element $x \in P$ is called a* minimal element *of P if $y \not\preceq x$ for every $y \in P$.*
3. *An element $x \in P$ is called a* maximum element *of P if $y \preceq x$ for every $y \in P$.*
4. *An element $x \in P$ is called a* maximal element *of P if $x \not\preceq y$ for every $y \in P$.*

Do you understand the difference between minimal and minimum (resp. maximal and maximum) elements of a poset? Test your understanding in the bow-tie poset of Figure 4.

These definitions are quite similar in flavour to those of upper and lower bounds: in particular, an element is maximum (resp. minimum) if and only if it is an upper (resp. lower) bound for the entire set P.

Incidence algebra NOXEPORT

In this section we introduce a useful algebraic tool to work with partial orders. First we introduce some definitions. Let $(P, \preceq)$ be a partially ordered set. For $x \preceq y$ in P, the *interval* (or more specifically, the *closed interval*) $[x, y]$ is defined as follows:

$$[x, y] = \{z \in P \mid x \preceq z \preceq y\}.$$

We can also define open and half open intervals as follows.

$$(x, y) = \{z \in P \mid x \prec z \prec y\},$$
$$(x, y] = \{z \in P \mid x \prec z \preceq y\},$$
$$[x, y) = \{z \in P \mid x \preceq z \prec y\}.$$

Let $I(P)$ be the set of all non-empty closed intervals of P.

Definition 25. *The* incidence algebra *of the poset P is defined as the set of all functions from $I(P)$ to $\mathbf{R}$:*

$$\mathcal{A}_P = \{f \colon I(P) \to \mathbf{R}\}.$$

Note that to specify an element of $\mathcal{A}_P$, we need to specify its value on every closed interval $[x, y]$ of P.

Example 26. *We note the following three examples of elements of $\mathcal{A}_P$.*

1. *Set f_0 to be the function that sends every closed interval to 0:*

$$f_0([x, y]) = 0 \quad \forall x \preceq y.$$

2. *Set δ to be the* Kronecker delta function*:*

$$\delta([x, y]) = \begin{cases} 1, & x = y, \\ 0, & x \neq y \end{cases}.$$

3. *Set ζ to be the function that sends every closed interval to 1:*

$$\zeta([x, y]) = 1 \quad \forall x \preceq y.$$

The incidence algebra may not seem all that interesting as a set. But it has several nice operations on it, which we now explore.

Addition If $f, g \in \mathcal{A}_P$, we define their sum $f + g$ as the following element of $\mathcal{A}_P$:

$$(f+g)([x,y]) = f([x,y]) + g([x,y]).$$

Scalar multiplication If $r \in \mathbf{R}$, and $f \in \mathcal{A}_P$ we define their scalar product rf as the following element[15] of $\mathcal{A}_P$:

$$(rf)([x,y]) = r \cdot f([x,y]).$$

[15] The $\cdot$ symbol represents usual multiplication of real numbers.

Convolution product If $f, g \in \mathcal{A}_P$, we define their *convolution product* $f * g$ as the following element of $\mathcal{A}_P$:

$$(f*g)([x,y]) = \sum_{x \preceq z \preceq y} f([x,z]) \cdot g([z,y]).$$

It is clear that the function f_0 is the identity element for the addition operation. That is, for any other function $f \in \mathcal{A}(P)$, we have

$$f + f_0 = f_0 + f = f.$$

The following proposition shows that the function δ is the identity element for the convolution product[16].

[16] In particular, since multiplicative identities are unique, the zeta function is *not* the multiplicative identity for the convolution product!

Proposition 27. *Let $(P, \preceq)$ be any finite poset, and let f be an element of $\mathcal{A}(P)$. Then*

$$(f * \delta) = (\delta * f) = f.$$

Proof. This can be verified by direct calculation, as follows:

$$(f*\delta)([x,y]) = \sum_{x \preceq z \preceq y} f([x,z])\delta([z,y]).$$

Note that $\delta([z,y]) = 0$ unless $z = y$. So the only term that survives in the summation is the one where $z = y$. So we have

$$(f*\delta)([x,y]) = f([x,y])\delta([y,y]) = f([x,y]).$$

Since this is true no matter what interval $[x,y]$ we chose, we see that $f * \delta = f$.

A similar calculation shows that $\delta * f = f$. □

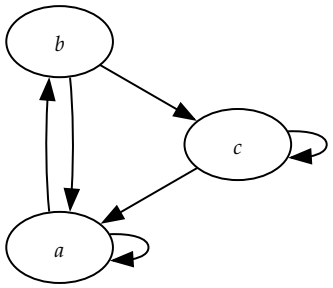
Exercise 28. *Prove that the multiplicative identity for the convolution product is unique. That is, if there is some function δ' such that $f * \delta' = \delta' * f = f$ for every function f, then $\delta = \delta'$.*

Graphs

Overview

Let us recall the definitions. A (directed) graph consists of a vertex set V and an edge set $E \subset V \times V$. If $(a, b) \in E$, we also write $a \to b$ as a directed edge. Typically we consider finite vertex sets when we work with concrete examples. An *undirected* graph is one in which the edge relation is symmetric: $(a, b) \in E$ if and only if $(b, a) \in E$. In this case, we often group the two flipped ordered pairs $\{(a, b), (b, a)\}$ and think of it as a *single* undirected edge $a - b$. Note that in this case if $a = b$, then the set $\{(a, b), (b, a)\}$ just becomes $\{(a, a)\}$, so we don't get a double loop. Usually we consider *simple* graphs, that is, those where we disallow multiple edges and parallel loops.

Example 29. *The drawing of a graph where the vertex set is* $V = \{a, b, c\}$ *and the edge relation is* $E = \{(a, a), (a, b), (b, a), (b, c), (c, c), (c, a)\}$.



Some natural questions

Graphs are a natural tool used to model various kinds of networks. This includes, for example, road/rail/flight networks, electrical/water flow networks, the "Facebook friend" graph, links between webpages, etc. Sometimes, these networks can be enhanced by adding "edge weights", which can be used, for example, to represent the distance between the two corresponding vertices, or in the context of flows, the "capacity" of an edge[17]. There are some very natural questions that one can ask about graphs: either practical ones that come up in many of the above contexts, or more theoretical ones. Here is a sample list, by no means exhaustive.

1. Is there a route from point A to point B?
2. How long is the route, and what is the shortest path?
3. How many routes are there? How long are they?
4. How much water/current/etc can flow through the network when at full capacity?
5. Is there a good way to figure out natural "clusters" in the graph? For example, how does Facebook know whom to suggest to you as a potential friend?
6. Can you find an unbroken path along the edges of the graph that goes through each vertex exactly once? (This is the *Hamiltonian path* problem.)

[17] In a "normal" graph, we usually take each edge to have weight 1.

7. Can you find an unbroken path along the edges of the graph that goes through each edge exactly once? (This is the *Eulerian path* problem.)

8. What is the shortest circuit (path that comes back to the starting point) that visits each vertex exactly once?

9. Is the graph *planar*? That is, can you draw the graph on a plane without crossing any of the edges?

Adjacency matrix

Recall the definition of an *adjacency matrix* of a graph (Definition 5). Given a graph (V, E), first we order the set V into a tuple $(v_1, \ldots, v_n)$. Then we create an $n \times n$ matrix A such that $A_{ij} = 1$ if $i \to j$ in the graph, and $A_{ij} = 0$ otherwise. In this section we will see how studying adjacency matrices of graphs helps us make progress towards some of the questions above.

Matrix products

First we recall matrix products. If A is an $m \times n$ matrix and B is an $n \times p$ matrix, then we can construct a product matrix AB, defined as follows:

$$(AB)_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} + \cdots + A_{in}B_{nj} = \sum_{k=1}^{n} A_{ik}B_{kj}.$$

Example 30. *Suppose that*

$$A = \begin{pmatrix} 1 & 2 \\ 0 & -1 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & 1 & -2 \\ 2 & 3 & 4 \end{pmatrix}$$

Then

$$AB = \begin{pmatrix} 4 & 7 & 6 \\ -2 & -3 & -4 \end{pmatrix}.$$

Powers of the adjacency matrix

Consider the example directed graph shown in Figure 5. The adjacency matrix and its square are

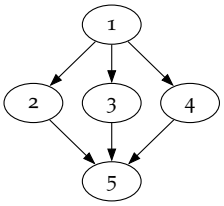


Figure 5: A directed graph

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad A^2 = \begin{pmatrix} 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Note that $A^k = 0$ for all $k > 2$. From the graph and from the matrix, we see that the only nonzero entry in A^2 is the entry at position $(1, 5)$, which equals 3. It arises as the sum $1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1$, which itself records all the possible compositions of two edges such that the composed path goes from 1 to 5. As in the picture, there are exactly three possibilities, and so the answer is 3.

This is a general phenomenon, and we have the following result.

Proposition 31. *Let A be the adjacency matrix of a simple directed graph (V, E). Suppose that the vertices are ordered as $(v_1, \ldots, v_n)$. Then the entry in the (i, j)th position of the kth power A^k of A counts the number of paths of length k from the vertex v_i to the vertex v_j.*

Proof. We proceed by induction. Indeed for $k = 1$, from the definition of the adjacency matrix, the (i, j)th entry equals 1 if and only if there is an edge from i to j in the graph. Now assume that we know the result for some $k > 0$, and we prove it for $k + 1$.

Let $B = A^k$, so that we can write $A^{k+1} = B \cdot A$. We calculate the (i, j)th entry of A^{k+1} as follows.

By the definition of matrix product, we know that this entry is the following sum:

$$(A^{k+1})_{i,j} = B_{i,1} \cdot A_{1,j} + B_{i,2} \cdot A_{2,j} + \cdots + B_{i,n} \cdot A_{n,j}.$$

For each number $1 \leq \ell \leq n$, we know that $B_{i,\ell}$ is the number of paths of length k from v_i to v_ℓ, and $A_{\ell,j}$ is the number of edges from v_ℓ to v_j. All together, the product $B_{i,\ell}A_{\ell,j}$ equals the number of paths of length $k + 1$ from v_i to v_j that travel through the vertex v_ℓ. Since we add over all possible vertices v_ℓ, the result (which is the (i, j)th entry of A^{k+1}) is the total number of paths of length $k + 1$ from v_i to v_j. □

We can also use the adjacency matrix to answer questions about connectedness of graphs. Suppose we want to know whether there is a path (of any length) from a vertex v_i to a vertex v_j. The previous proposition tells us that to find paths of a given length k, we need to look at entries of A^k. So as long as we find a positive entry in the (i, j)th spot of some power of A, we know that we have found a path. In other words, we can look at the (i, j)th entry of a sum $A + A^2 + \cdots$, and stop once we find a positive entry.

But how do we know when to stop adding? To answer this question, let us analyse the shortest possible path from some v_i to some v_j, under the assumption that there is at least one path.

Proposition 32. *If v_i and v_j are vertices in the graph such that there is at least one path from v_i to v_j, then the length of the shortest path from v_i to v_j cannot be more than n. Further, if $v_i \neq v_j$, then the length of the shortest path from v_i to v_j cannot be more than $n - 1$.*

The Boolean product and transitive closures

In this subsection and the next, we study a couple of variant products on the adjacency matrix, that let us compute different things about our graphs. The first variant is the *Boolean product*, which will be used to compute transitive closures.

First we define the following binary operations on the set $\{0, 1\}$. That is, we define the following functions $\{0, 1\} \times \{0, 1\} \to \{0, 1\}$.

Boolean addition This is also known as "OR" or "$\vee$", and is defined as follows:

$$0 \vee 0 = 0, \quad 0 \vee 1 = 1 \vee 0 = 1 \vee 1 = 1.$$

Boolean multiplication This is also known as "AND" or "$\wedge$", and is defined as follows[18]:

$$1 \wedge 1 = 1, \quad 0 \wedge 1 = 1 \wedge 0 = 0 \wedge 0 = 0.$$

[18] Note that Boolean multiplication coincides with the usual multiplication operation restricted to the set $\{0, 1\}$.

The Boolean matrix product is then defined on matrices with entries in the set $\{0,1\}$, and also outputs a matrix with entries in the same set $\{0,1\}$. To define the Boolean matrix product, we use $\vee$ instead of $+$, and $\wedge$ instead of $\times$" respectively, as follows. Let A be an $m \times n$ matrix and B be an $n \times k$ matrix, both with entries in the set $\{0,1\}$. Then the Boolean product $A * B$ is defined as follows (entry-wise):

$$\begin{aligned}(A * B)_{i,j} &= (A_{i1} \wedge B_{1j}) \vee (A_{i2} \wedge B_{2j}) \vee \cdots \vee (A_{in} \wedge B_{nj}) \\ &= \bigvee_{k=1}^{n} A_{ik} \wedge B_{kj}.\end{aligned}$$

Now let A be the adjacency matrix of a graph. Then the (i,j)th entry of the Boolean square of A equals 1 if and only if there exists a path of length two from i to j in the graph. This is because the (i,j)th entry is a Boolean sum ($\vee$) of several entries, and the ℓth such entry equals 1 if and only if there is an edge from i to ℓ and also an edge from ℓ to j. The Boolean sum of all of these equals 1 if and only if at least one of the entries is equal to 1, which is true if and only if there is some path of length two from i to j. Extending this reasoning to a k-fold product, we obtain the following result. The proof is similar to that of Proposition 31 and so we omit it.

Proposition 33. *Let A be the adjacency matrix of a simple directed graph (V,E). Suppose that the vertices are ordered as $(v_1,\ldots,v_n)$. Then the entry in the (i,j)th position of the kth Boolean power A^{*k} of A equals* 1 *if there is a path of length k from the vertex v_i to the vertex v_j, and equals* 0 *otherwise.*

Weighted graphs and weighted adjacency matrices

Now suppose that $G = (V,E)$ is a *weighted graph*. This means that each edge has an associated *weight*, which is usually a non-negative real number. Mathematically, we can write this as a function $w\colon E \to \mathbf{R}$, sending each edge to a real number. In practical applications, graphs often have edge weights, for example the length of a road or the cost of going through a toll bridge, and weighted graphs are models of these situations. We would like to use adjacency matrices to compute the weight of the least-cost (that is, smallest weight) path between any pair of vertices. We can achieve this by writing down a *weighted adjacency matrix*, and by computing a new product on it. The weighted adjacency matrix simply lists the weight of each edge. The diagonal entries are all 0 because one can get from any vertex to itself with zero cost (by not moving). All entries (i,j) where (i,j) is not an edge are set to ∞[19].

Definition 34. *Let $G = (V,E)$ be a directed graph with weight function $w\colon E \to \mathbf{R}$. Suppose that the vertices are ordered as $(v_1,\ldots,v_n)$. The*

[19] We use the symbol ∞ as a placeholder for an extremely large number: for any real number r in our calculations, we will set $r + \infty = \infty$ and $\min\{r,\infty\} = r$.

weighted adjacency matrix of G is an $n \times n$ matrix W, defined as follows:

$$W_{ij} = \begin{cases} 0, & \text{if } i = j, \\ w((i,j)), & \text{if } (i,j) \in E, \\ \infty, & \text{otherwise.} \end{cases}$$

Note that this adjacency matrix is set up in a way such that the (i,j)th entry shows the minimum-cost path of length at most 1 (that is, either one edge or no edge, in the case that $i = j$) from i to j. To find the minimum-cost path of length at most 2 from i to j, we need to iterate over all possible intermediate steps $i \to \ell \to j$, add the edge weights of $i \to \ell$ and $\ell \to k$, and then take the minimum. This operation is extremely similar to the standard matrix product, except that instead of multiplying the (i,ℓ)th entry with the (ℓ,j)th entry we are adding them, and instead of adding over all possibilities we are taking the minimum over all possibilities. We define this "min-plus" matrix product as follows.

Definition 36. *Let A be an $m \times n$ matrix and B be an $n \times k$ matrix, such that the entries of A and B are either real numbers or ∞. The "min-plus" product of A and B, denoted $A \odot B$, is defined as follows (entry-wise):*

$$(A \odot B)_{i,j} = \min\{(A_{i1} + B_{1j}), (A_{i2} + B_{2j}), \ldots, (A_{in} + B_{nj})\}.$$

Now let W be the weighted adjacency matrix of a weighted graph. Note that the (i,j)th entry of $W \odot W$ is precisely the weight of the minimum-weight path from i to j that has at most two edges. Generalising this, we have the following proposition. The proof is similar to that of Proposition 31, and is omitted.

Proposition 38. *Let W be the weighted adjacency matrix of a weighted graph with n vertices.*

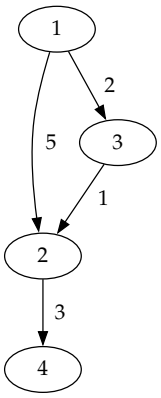
1. *The (i,j)th entry of $W^{\odot k}$ is the weight of the minimum-weight path from i to j that has at most k edges.*
2. *If all the edge weights are non-negative, then the (i,j)th entry of $W^{\odot(n-1)}$ is the weight of the minimum-weight path (with any number of edges) from i to j.*

The technique of repeated squaring

This section is an aside. We discuss the method of *repeated squaring* to quickly find powers of a matrix (or indeed, to quickly find powers in general). This method works for any associative product operation, including the standard matrix product, the Boolean matrix product, and the min-plus matrix product. For concreteness, we discuss it for the standard matrix product.

Let A be a square matrix. The naive method to compute a power of A, for example A^8, would be to multiply A serially with itself 8 times. This consist of 7 matrix product operations. However,

Example 35. *Consider the weighted graph shown below.*



Its weighted adjacency matrix is

$$W = \begin{pmatrix} 0 & 5 & 2 & \infty \\ \infty & 0 & \infty & 3 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix}$$

.

Example 37. *For the graph in Example 35, the second and third min-plus powers of the weighted adjacency matrix are:*

$$W^{\odot 2} = \begin{pmatrix} 0 & 3 & 2 & 8 \\ \infty & 0 & \infty & 3 \\ \infty & 1 & 0 & 4 \\ \infty & \infty & \infty & 0 \end{pmatrix},$$

and

$$W^{\odot 3} = \begin{pmatrix} 0 & 3 & 2 & 6 \\ \infty & 0 & \infty & 3 \\ \infty & 1 & 0 & 4 \\ \infty & \infty & \infty & 0 \end{pmatrix}.$$

Indeed, the entries of the min-plus cube give the minimum weights of possible paths between any pairs of vertices in the graph.

there is a quicker method: if we first find and save A^2, then we can multiply that with itself to obtain and save A^4, and finally multiply that with itself to get A^8. In total, that corresponds to only 3 matrix product operations! This is considerably faster than serial multiplication.

But what if we don't have an even number, or a power of two as the power we need to compute? Suppose we are trying to compute A^n where n is not necessarily a power of two. In this case, we simply square the matrix repeatedly, saving the results, until we reach a power less than or equal to n. Then we write n as a sum of distinct powers of two[20], and then multiply together the corresponding powers of A to get the final result. Here is an example.

Example 39. *Suppose that $n = 19$. In this case, we remember $M_0 = A$, $M_1 = A^2$, $M_2 = M_1^2 = A^4$, $M_3 = A^8$, and $M_4 = A^{16}$. Finally, note that $19 = 16 + 2 + 1 = 2^4 + 2^1 + 2^0$, and so*

$$A^{19} = M_4 \cdot M_1 \cdot M_0.$$

This process corresponds to a total of 6 matrix product operations (four squarings and two multiplications), as opposed to the 18 product operations required for serial multiplication.

[20] Writing a positive integer n as the sum of distinct powers of two is also called *binary writing*. There are several ways to obtain it. For example, we can follow the following recursive algorithm: if n is even, we write it as $2m$, and if n is odd, we write it as $2m + 1$. Repeating the process on the m obtained until we reach 1, we obtain an expression which expands to a sum of distinct powers of two. For example,

$$\begin{aligned} 7 &= 2(3) + 1 = 2(2(1) + 1) + 1 \\ &= 4 + 2 + 1. \end{aligned}$$

Regular expressions and finite automata

In this chapter, we will study regular expressions, regular languages, and finite automata. The aim of the chapter is to build up tools for "pattern-matching" strings over a fixed alphabet, and to isolate subsets of strings that match certain patterns.

Regular expressions

A regular expression is a systematic formula that specifies certain strings of a given alphabet. We first need to define what we mean by alphabet and string, and some basic constructions.

Definition 40. *An alphabet Σ is a finite set of symbols, called the* letters *of Σ. A* string *or a* word *is a finite ordered list of elements of Σ, written without spaces or punctuation. The* length *of a word is the number of letters in the word.*[21]

[21] The unique empty word is also allowed, and is denoted ε. For this reason we usually assume that ε is not a symbol in Σ.

A commonly used alphabet is $\Sigma = \{0, 1\}$. In that case, examples of strings or words in this alphabet are 10, 00, 1110, 0, 1, and ε.

If Σ is a fixed alphabet, then we denote by Σ^* the set of all strings, including ε.

Definition 42. *Fix an alphabet Σ. A* language *L on Σ is any subset of Σ^*.*

Exercise 41. *Check that if $\Sigma = \emptyset$ then $\Sigma^* = \{\varepsilon\}$, but otherwise Σ^* is infinite.*

Unless otherwise specified, we will use the alphabet $\Sigma = \{0, 1\}$ as our default alphabet.

Basic constructions with strings

Fix an alphabet Σ. We begin by listing some basic constructions on languages on Σ and strings in Σ.

Concatenation (on strings) Let $v = a_1 \dots a_k$ and $w = b_1 \dots b_l$ be strings, with $a_i, b_j \in \Sigma$ for every i and j. The *concatenation* of v and w is the string

$$vw = a_1 \dots a_k b_1 \dots b_l.$$

Concatenation (on languages) Let $L_1, L_2 \subseteq \Sigma^*$ be languages. The *concatenation of L_1 and L_2* is a new language on Σ, denoted by $L_1 \circ L_2$ and defined as follows.

$$L_1 \circ L_2 = \{vw \mid v \in L_1, w \in L_2\}.$$

Union (of languages) If $L_1, L_2 \subseteq \Sigma^*$ are languages, then their *union* $L_1 \cup L_2$ is just the set union. So

$$L_1 \cup L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ or } w \in L_2\}.$$

Star (of a language) Let $L \subseteq \Sigma^*$ be a language. Then the *star of* L, denoted L^*, consists of any number of concatenations of words in L. That is,

$$L^* = \{w_1 w_2 \dots w_k \mid k \geq 0 \text{ and } w_i \in L \text{ for each } i\}.$$

Example 43. *1. If $L = \emptyset$ then $L^* = \{\epsilon\}$.*

Lexicographic order (dictionary order)

Suppose that we have ordered the elements of Σ. Then Σ^* (and any other language on Σ) inherits a total order, known as the lexicographic order. In this order, we can compare two words v and w using the following steps.

1. If v and w have unequal lengths, then the shorter word is said to be less than or equal to the longer word.
2. If v and w have the same length n, then we can write them as

$$v = a_1 \cdots a_n \text{ and } w = b_1 \cdots b_n,$$

where a_i, b_j are letters. Then we compare letter by letter starting from 1 to n. If $v \neq w$ then at least one position i must have $a_i \neq b_i$. Let i be the smallest number for which the letters a_i and b_i differ. If $a_i < b_i$ in the order on Σ, we say $v < w$. Otherwise if $b_i < a_i$, we say $w < v$.

Example 44. *Assume we use the order $(0, 1)$ on $\Sigma = \{0, 1\}$.*

1. *The word ϵ is shorter than every other word, so appears first in the lexicographic order on Σ^*.*
2. *The word 11 appears before 011 (or any other word of three or more letters).*
3. *The word 01 appears before 11 but after 00.*

Regular expression syntax and matching

We are now ready to define regular expressions. A regular expression should be thought of as a particular way to specify a pattern, that can "match" zero or more strings in a given language. Regular expressions are built out of three basic patterns and three "operators" that make bigger patterns using smaller ones.

Definition 45. *Fix an alphabet Σ. A word r written using the letters of Σ, together with the symbols $*$ and $|$, is a valid regular expression if it satisfies one of the following.*[22]

1. $r = \emptyset$
2. $r = \varepsilon$
3. $r = a$ *for some* $a \in \Sigma$
4. $r = r_1 r_2$ *for two valid regular expressions* r_1 *and* r_2
5. $r = r_1 | r_2$ *for two valid regular expressions* r_1 *and* r_2
6. $r = s^*$ *for a valid regular expression* s.

[22] Additionally, we are also allowed to parenthesise subexpressions to avoid ambiguity. We assume that Σ does not contain any of the symbols "(", ")", "|", "$*$", or "$\emptyset$".

We now discuss what it means for a string to "match" a regular expression.

Definition 46. *Let Σ be an alphabet and let r be a regular expression on Σ. Let $w \in \Sigma^*$ be any word. We say that w* matches *r if the following hold.*

1. *$r \neq \emptyset$, because no word matches the regular expression $\emptyset$.*
2. *If $r = \epsilon$ or $r = a$ for some $a \in \Sigma$, then $w = r$.*
3. *If $r = r_1 r_2$ then there is at least one way to break up w into $w = v_1 v_2$, such that v_1 matches r_1 and v_2 matches r_2.*
4. *If $r = r_1 | r_2$ then either w matches r_1 or w matches r_2 (or it matches both).*
5. *If $r = s^*$, then w can be broken up as a concatenation of zero or more subwords, $w = v_1 \dots v_k$, such that each v_i matches s.*

Deterministic finite automata

A *finite automaton* is an abstract machine that performs calculations according to certain rules. We will begin by discussing deterministic finite automata, and discuss their relationship to regular expressions.

Definition 47. *Fix an alphabet Σ. A deterministic finite automaton for Σ is described by the following pieces of data.*

1. *A (usually finite) set of* states*, usually denoted Q.*
2. *A* start state[23]*, usually denoted $q_0 \in Q$.*
3. *A set of* accept states *$A \subseteq Q$.*[24]
4. *A transition function*

$$\delta \colon Q \times \Sigma \to Q.$$

The definition is not very illuminating. It is often much clearer to draw the *state diagram* of a finite automaton, as shown in Example 48. In this example, we can decode the formal data of the DFA as follows.

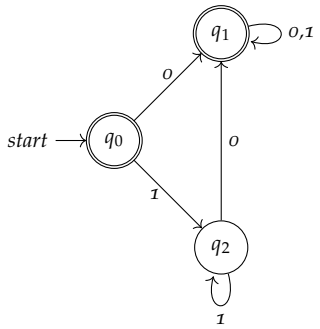
1. The set of states is $Q = \{q_0, q_1, q_2\}$.
2. The start state is q_0.
3. The set of accept states is $A = \{q_0, q_1\}$.
4. The transition function can be represented as a table as follows.

Input state	Letter	Output state
q_0	0	q_1
q_0	1	q_2
q_1	0	q_1
q_1	1	q_1
q_2	0	q_1
q_2	1	q_2

[23] The start state is always unique.

[24] The set of accept states can be *any* subset of Q, including the empty set. Changing the set of accept states while keeping everything else the same typically changes the results of the calculation drastically.

Example 48. *Here is an example of a finite automaton.*



Given a DFA M and a word $w \in \Sigma^*$, we can *run* the machine M on the word w, as follows. Suppose that $w = a_1 a_2 \dots a_k$, where each a_i is a letter of Σ. We then have the following steps.

1. We begin at the start state $p_0 = q_0$ and "read" the letter a_1.
2. We move to the state $p_1 = \delta(p_0, a_1)$. From here, we read the letter a_2.
3. Next, we move to the state $p_2 = \delta(p_1, a_2)$. From here, we read the letetr a_3.
4. Continue in this manner, moving to the state $p_n = \delta(p_{n-1}, a_n)$ by reading the letter a_n.
5. Stop at the state p_k, which is reached after reading the last letter a_k.
6. If p_k is an accepting state of M, we say that M *accepts* w. If p_k is not an accepting state of M, we say that M *rejects* w.

Example 49. *Consider the string $w =$ 1101, running on the machine M from the previous example. It goes through the following steps on M.*

1. *Start at q_0, read 1, move to q_2.*
2. *From q_2 read 1, move to q_2.*
3. *From q_2 read 0, move to q_1.*
4. *From q_1 read 1, move to q_1.*

At the end of this process, we are at q_1, which is an accepting state. Therefore M accepts w.

Definition 50. *Let M be a DFA. The set of all strings accepted by M is called the* language of M*, denoted $L(M)$.*

Nondeterministic finite automata

A *nondeterministic finite automaton* or NFA is a generalisation of a DFA. It is a machine in which, informally, we may have some choices when we try to read letters. More precisely, in an NFA we relax the restriction that there is exactly one outgoing arrow from each state labelled by each letter of Σ. We also give ourselves the luxury of allowing arrows labelled by the empty string ε. If there is an arrow labelled ε from a state a to a state b, then informally it means that we have a choice, when we are at a, to teleport to the state b without reading any letter.

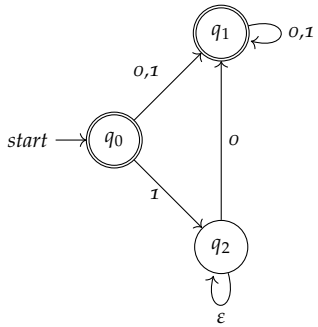
Let us give a formal definition.

Definition 52. *Fix an alphabet Σ. A nondeterministic finite automaton for Σ is described by the following pieces of data.*

1. *A finite set of states, usually denoted Q.*
2. *A start state, usually denoted $q_0 \in Q$.*
3. *A set of accept states, usually denoted $A \subseteq Q$.*
4. *A transition function*[25]

$$\Delta \colon Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q).$$

Example 51. *Here is an example of a nondeterministic finite automaton.*



[25] This transition function also takes in a pair (q, a) as input, where $q \in Q$ and a is either a letter of Σ or the empty string ε. The output is a (possibly empty) set of states of Q. Visually, we should think of having outgoing arrows from q to each element of $\Delta(q, a)$, each of them labelled by a.

Once again, let us describe the parts of the definition for the example in Example 51. The set of states, the start state, and the set of accept states are exactly as in the previous example (Example 48). The transition function is as follows.

Input state	Letter or ε	Set of output states
q_0	0	$\{q_1\}$
q_0	1	$\{q_1, q_2\}$
q_0	ε	$\emptyset$
q_1	0	$\{q_1\}$
q_1	1	$\{q_1\}$
q_1	ε	$\emptyset$
q_2	0	$\{q_1\}$
q_2	1	$\emptyset$
q_2	ε	$\{q_1\}$

As before, we can *run* strings on NFAs. However, the process of calculation may now involve several choices, depending on how many possible output states there are for each input state and letter (or empty string). We represent the calculation as a *calculation tree*, as shown by the following example.

TODO *Calculation tree example*

TODO *Relationship between NFAs and DFAs*

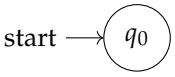
Regular expressions to finite automata

The aim of this section is to try and convert any given regex to an *equivalent*[26] finite automaton (either a DFA or an NFA). We have already seen that given any NFA, one can construct an equivalent (probably much larger) DFA. So to make things simpler for us, we will convert regexes to NFAs.

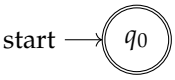
[26] We say that a regex r is equivalent to a finite automaton M if $L(r) = L(M)$.

We do this inductively, constructor-by-constructor.

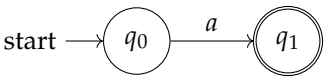
1. If $r = \emptyset$, we simply have to find an NFA that rejects every string. The easiest option is as follows.



2. If $r = \epsilon$, we construct an NFA that only accepts the empty string. A possible option is as follows.

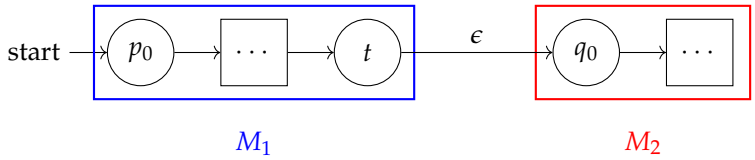


3. If $r = a$ for some $a \in \Sigma$, we construct an NFA that only accepts the string a. A possible option is as follows.

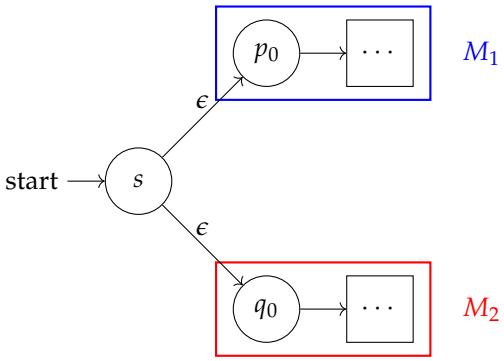


4. If $r = r_1 r_2$ for two regexes r_1 and r_2, we construct an equivalent NFA inductively. Assume that M_1 and M_2 are NFAs equivalent to r_1 and r_2 respectively. Assume furthermore for simplicity

that M_1 has exactly one accept state t (if not, we can add a new accepting state, and redirect all previously accepting states to it by ϵ-transition arrows). Let p_0 and q_0 be the start states of M_1 and M_2 respectively. We can then construct a new automaton that connects M_1 and M_2 by joining t to q_0 by an ϵ transition, whose accepting states are simply the accepting states of M_2. This construction is illustrated below.

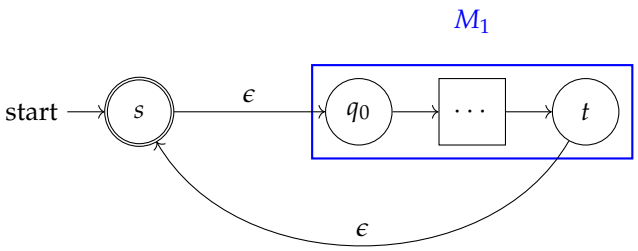


5. If $r = r_1 \mid r_2$ for two regexes r_1 and r_2, we construct an equivalent NFA inductively. Assume that M_1 and M_2 are NFAs equivalent to r_1 and r_2 respectively. Let p_0 and q_0 be the start states of M_1 and M_2 respectively. We construct a new automaton with start state s, which connects to both p_0 and q_0 by ϵ-arrows. The set of accepting states of the new automaton is a union of the sets of accepting states of M_1 and M_2. This construction is illustrated below.



6. If $r = (r_1)^*$ for a regex r_1, we construct an equivalent NFA inductively. Assume that M_1 is an NFA equivalent to r_1. Assume again for simplicity that t is the only accepting state of M_1, and that q_0 is its start state. To construct our new NFA, we add a dummy start state s, make it accepting, and connect t to s via an ϵ arrow. This construction ensures that we accept the empty string, as well as any string that successfully passes through M_1 several times. The construction is illustrated below.

What would happen if we didn't add s, and instead made q_0 accepting instead, connecting t to q_0?



We see at the end of this process that every regex constructor can be "converted" to an equivalent NFA. By chaining together these basic constructions, we can therefore convert every regex to an equivalent automaton!

Converting finite automata to regular expressions

Recall that we say that two automata M_1 and M_2 are *equivalent* if $L(M_1) = L(M_2)$. We have already seen that DFAs and NFAs are equivalent in power. That is, for any DFA M there is an equivalent NFA N, and for any NFA N there is an equivalent DFA M.

In this section we focus on converting a given DFA or NFA to an equivalent regular expression r. We will start with an arbitrary machine M, and perform a series of reductions to delete states, successively overloading the arrow labels, until we hit a machine with only two states and a single label, which will be the required regex.

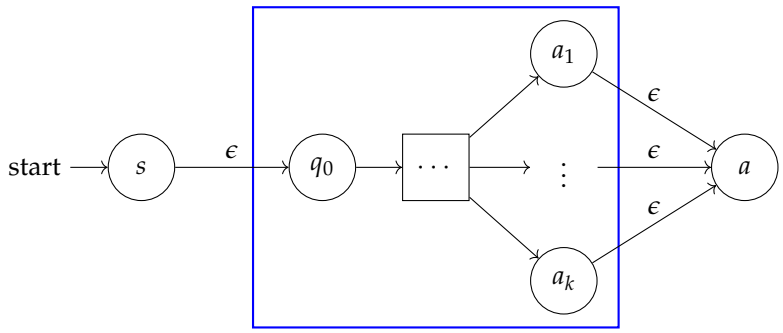
Consider a machine M. First, we *sanitise* or *quarantine* the machine as follows[27].

[27] I use this terminology because you should imagine that you are performing a surgery on your machine, so you want to put it inside a nice sanitised operating box. We then put on our gloves and perform our surgery inside the box, not letting anything in the box interact with anything outside the box, unless strictly necessary.

1. If q_0 was the original start state of M, add a new state s before q_0, connecting it to q_0 by an ϵ arrow. The state s is now our new start state.

2. If $a_1, \ldots, a_k$ were previously the accepting states of M, we add a new accepting state a after $a_1, \ldots, a_k$, with an ϵ-arrow $a_i \xrightarrow{\epsilon} a$ for each $i \in \{1, \ldots, k\}$. We then make $a_1, \ldots, a_k$ non-accepting. Consequently, the new machine has only one accepting state.

   Here is a visual representation.



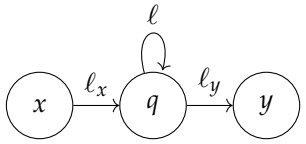
The "operating box".

Once we have done this procedure, we get to work deleting each state of the machine inside the "operating box" one by one. We can't however simply delete a state! We have to compensate for deleting a state by adding extra arrows or labels, so that anything that would have previously gone through that state has an alternate route. The deletion algorithm goes as follows.

1. Choose a state q inside the operating box to delete. If there are no more states left, we are done.

2. Let ℓ be the label on any loop from q to itself. If there is no loop, then we take ℓ to be the empty word ϵ.

3. Consider every possible configuration as follows.



   Here, x and y are states such that $x \neq q$ and $y \neq q$ (but x and y may be equal), and ℓ_x and ℓ_y are the labels on the arrows shown respectively.

4. Each such configuration is a portion that accepts substrings that match the regex $\ell_x \ell^* \ell_y$. In other words, a substring goes through successfully along the portion $x \to q \to y$ if and only if it matches $\ell_x \ell^* \ell_y$. By simply removing q, those substrings would no longer have access to this path. So instead, we add on $\ell_x \ell^* \ell_y$ as a direct label from x to y. Formally, consider any existing arrow $x \to y$ with label r. If there is such an arrow, then change the label on that arrow to $r \mid \ell_x \ell^* \ell_y$. Otherwise, create an arrow $x \to y$ with label $\ell_x \ell^* \ell_y$.

5. Once this has been done for every configuration $x \to q \to y$ for all possible values of x and y, delete q.

6. Note that a string is accepted by the new machine (noting that now an arrow with a label e accepts any substring that matches e) if and only if it is accepted by the old machine.

7. The machine now has one fewer state in the operating box, so go back to step 1.

It is clear that this algorithm terminates with no states left inside the operating box. When there are no more states left, there will be exactly one arrow from s to a, and it will have a regular expression as a label on it. By construction, this regex accepts exactly the strings accepted by the machine we started with, and so it is equivalent to the original machine!

Non-regular languages and the pumping lemma

Recall that a *regular* language is one that is accepted by a deterministic or non-deterministic finite automaton, or equivalently, one that is the language of some regular expression. It turns out that not all languages are regular.[28]

How do we know whether a given language is regular or not? If we can find a machine or regex that recognises precisely that language, then the language is regular. However, if we cannot come up with a regex or machine that recognises that language, how can we *provably* say that the language is not regular? In this section, we

[28] A detailed proof of this is outside the scope of these notes, but here is a proof sketch. If you fix an alphabet Σ, then the cardinality of the set of all languages on Σ is the cardinality of the power set of Σ^*. The set Σ^* is a countable infinite set, and so its power set is uncountable. On the other hand, the set of regular languages is necessarily countable — it is possible to lexicographically enumerate all possible regular expressions, so the set of languages that is recognised by any one of them must also be countable!

discuss a criterion, called the *pumping lemma*, that lets us pin down some[29] non-regular languages.

[29] Be careful — there are non-regular languages that fool the pumping lemma. That is, it does not pick up all non-regular languages.

The idea behind the pumping lemma is relatively simple. Suppose you have a regular language L. This means that there is some DFA M such that $L = L(M)$. This DFA M has finitely many states, say n states.

Every word that M accepts must start at the start state, and pass through (some of) these n states, before ending up at an accepting state. If the language L is infinite, then it must contain words that are longer than n letters. Therefore, as these words travel through M, they must repeat a state. Suppose w is such a word, and note that we can break up w into three pieces $w = xyz$, such that the portion y is non-empty, and starts and ends at the same state. In this situation, because y ends at the same point that it started, the word $xyyz$ must also necessarily end on the same accet state that w ends on! We can say the same thing about the words xz, $xyyyz$, and more generally, xy^iz for any integer $i \geq 0$.

With this background, here is the idea for the pumping lemma. Suppose that there is a language L such that for arbitrarily long words $w \in L$, there is *no way* to split up w into three sections $w = xyz$ such that $xy^iz \in L$ for every L. Then L cannot be regular.

We state this theorem formally first, and will then give an example to see how it can be used.[30] Understanding this theorem is an exercise in getting your order of quantifiers correct!

[30] Typically we use the *contrapositive* of this theorem. That is, we find a language that does not satisfy the consequence of the theorem, and thereby conclude that it is not regular.

Theorem 53. *Suppose L is a regular language. Then there is some positive integer n, called a* pumping length *for L, with the following property. For any $w \in L$ such that $|w| \geq n$, there exists some way to split w as $w = xyz$, such that:*

1. $|y| \geq 1$;
2. $|xy| \leq n$;
3. *the words xy^iz are in L for every integer $i \geq 0$.*

Proof. The proof goes as explained in the previous discussion. Suppose L is regular, and suppose that M is some DFA that recognises L. Suppose also that M has n states. Then we claim that n is a pumping length for L.

Consider any $w \in L$ such that $|w| \geq n$. Now as w travels through M from the start state to the accept state, it will encounter a repeated state q within the first n of its letters. Take x to be the portion of w up until we reach q for the first time. That is, after reading the last letter of x, we are at the state q for the first time. Let y be the portion after x up until we reach q for the second time.

Since we know that q appears at least twice as we travel through the first n letters, we see immediately that

1. $|xy| \leq n$, and
2. $|y| \geq 1$.

Finally, let z be the remainder of w after xy. Recall that after reading the last letter of z, we are at an accepting state of M. Now consider any string of the form xy^iz for i a non-negative integer. As we run xy^iz through M, we reach q after we finish travelling through x. The portion y starts and ends at q, so any power of it will also start and end at q. Finally, the portion z starts at q and ends at an accepting state of M. Therefore it is evident that M accepts xy^iz for each integer $i \geq 0$, and the proof is complete. □

Let us see a simple example of how to use this theorem. Consider the language $L = \{0^k1^k \mid k \geq 0\}$. The words in this language consist of a string of 0s followed by a string of *the same number of* 1s. We will show that L is not regular.

Suppose for contradiction that L is regular, and let n be its pumping length. Then any word in L of size at least n can be split up as in the pumping lemma. Consider the word $w = 0^n1^n$. We have to be able to split up w as $w = xyz$ such that $|xy| \leq n$, $|y| \geq 1$, and $xy^iz \in L$ for every non-negative integer i.

Note however that the first condition ($|xy| \leq n$) guarantees that both x and y only consist of strings of zeroes. Suppose that $x = 0^a$ and $y = 0^b$ for some a, b such that $b \geq 1$ and $a + b \leq n$. Then z is necessarily equal to $0^{n-a-b}1^n$.

Now consider the string $xyyz = xy^2z$. This can be computed to be $0^a0^b0^b0^{n-a-b}1^n = 0^{n+b}1^n$. Clearly, this string is not in L! We have thus managed to violate the pumping lemma, from which we conclude that L could not have been regular in the first place.

Combinatorial games

We begin the course with some games. The theory of games is a rich subject that can be used to model problems in logic, computer science, economics, and social science, depending on the rules you impose on your games. We will focus on *impartial combinatorial games*.

An impartial combinatorial game is usually played with two players and satisfies the following conditions.

1. There is a (usually finite) set of possible *game states*.
2. There are rules that describe the possible moves from a given game state to other game states.
3. The game is *impartial*, which means that the rules to go from one game state to the next do not depend on which player is about to make the move[31].
4. The players alternate making moves to move from one game state to the next.
5. The first player to be unable to make a move loses the game[32].
6. There is complete information (the entire game state is known to both players at all times).
7. There are no chance moves.

[31] Contrast this to a game such as chess, in which one player may only move the white pieces and the other player may only move the black pieces.

[32] This is called *normal play*. In the variant called *misère play*, the first player unable to make a move wins the game.

Here is a basic example of an impartial combinatorial game, namely the *subtraction game*.

Fix a finite set of positive integers, say $S = \{1, 3, 4\}$. In the subtraction game with respect to S, we start with a non-negative integer n. A valid move consists of replacing n by $n - k$ where k is some element of S. In this case, the possible valid moves are $n \mapsto n - 1$, $n \mapsto n - 3$, and $n \mapsto n - 4$. The output must remain a non-negative integer, and the person who cannot make a move loses.

Can the first player win if the starting position is $n = 5$? How about $n = 10$? How can you be sure?

Strategic labelling

A basic tool to study an impartial combinatorial game is the *game graph*. This is a directed graph whose vertices represent possible states of the game (usually all states potentially reachable from the starting state). We draw an edge from state s_1 to state s_2 if there is a single move that takes us from s_1 to s_2.

The finiteness condition on impartial combinatorial games means that there are only finitely many states reachable from any given starting state, so the game graph drawn from any fixed starting position is finite. Moreover, there are no directed cycles in this graph, because each possible sequence of moves terminates at a state from which there are no moves possible.

So if we build the full game graph starting at the starting configuration, we can then analyse whether there is a winning strategy. As an easy example, if there are no possible moves from the starting configuration, then the first player will automatically lose.

Since the possible moves from a given state do not depend on which player is going to play next, we can simply figure out if a given state is a "winning" or a "losing" position. Let s be a game state. We say that s is an N state if the *next* player to play has a winning strategy for the state s. We say that s is a P state if the next player has no winning strategy for the state s; equivalently, if the *previous* player has a winning strategy for the state s no matter what move the next player makes. So N states are next-player wins, and P-states are previous player wins.[33]

[33] Remember that this labelling assumes that everyone plays optimally and makes no mistakes! It is still possible for the next player to lose from an N state if they make the wrong move, but a state gets the label N if it is possible for the next player to win by playing optimally.

We can label states as N and P inductively, building up from the bottommost positions. First, it is clear that if s has no outgoing arrows, then it is a P state — the next player to play automatically loses, and hence the *previous* player has won. We call such states *terminal* states, because the game terminates or ends at these states. So any terminal state is a P state.

Therefore, anything that has at least one arrow to a terminal state is an N state: the next player can simply move to the terminal state, so that the player after the next player (aka the previous player) has no possible moves left. To generalise this, any state that has at least one arrow to a P state is an N state: the next player can simply move to the P state, which is guaranteed to be a losing position for the player after next (aka the previous player). So when is a state a P state? Well, a state is a P state if no matter what the next player does, the previous player has a winning strategy. This means that a state is a P state if and only if all outgoing arrows point to N states.

Definition 54. *The* outcome *of a game G is defined to be P if the game state G is a P-state, and N if G is an N-state.*

TODO *Draw running example*

Nim

Let us discuss *nim*, which is a very important example of an impartial combinatorial game. The game is played as follows. The start state consists of a finite number of piles of stones, each possibly of a different size. For instance, we may have the state $\{2, 3, 5\}$. We will represent states as *multisets*: that is, the order is unimportant, but entries can repeat. The size of each pile must be a non-negative integer. If a pile shrinks to size 0, we optionally omit it from the

representation, so that $\{2,3,5\}$ is the same as $\{0,2,3,5\}$.

A move consists of choosing *one* of the piles, and removing and discarding some of the stones in that pile. At least one stone must be removed, and the player may choose to remove all the stones from the chosen pile. For instance, from the state $\{2,3,5\}$, we can move to the state $\{2,2,5\}$ by removing one stone from the pile that had 3 stones. Or for instance, we could move to the state $\{2,5\}$, by removing all the stones from the pile that had 3 stones. The person who cannot make a move loses; this can only happen if the player is presented with the empty state $\{\}$.

Let us work out some easy examples. First, suppose that the start state consists of a single pile with 0 stones: $\{\}$. This is clearly a P-state. Next, suppose that the start state consists of a single pile with n stones for $n > 0$: $\{n\}$. This is an N-state, because the next player can remove all n stones to reach the terminal state $\{\}$.

Next, suppose that the start state is $\{m,n\}$, with both m and n positive. First suppose that $m = n$. We claim that this state, namely $\{n,n\}$, is a P-position. To see this, proceed by induction: the only possible sequence of moves from $\{1,1\}$ is

$$\{1,1\} \to \{0,1\} \to \{0,0\}.$$

Since $\{0,0\}$ is a P state, we see that $\{0,1\}$ is an N state, and so $\{1,1\}$ is a P state.

Now let $n > 1$, and assume the result for all $1 \leq k < n$. The only possible *type* of move from $\{n,n\}$ is $\{n,n\} \to \{m,n\}$, where $m < n$. This state is an N state, because there is a move $\{m,n\} \to \{m,m\}$, and we know by induction that $\{m,m\}$ is a P state. Therefore, $\{m,n\}$ is an N-state for all m such that $m < n$, and so all arrows out of $\{n,n\}$ point to N-states. Hence $\{n,n\}$ is a P state as well.

This is a more general technique, known as *mirroring*. In some situations, it is possible, by symmetry, to mirror the opponent's move so that the opponent is presented with a smaller version of the same kind of state as before. In this situation, the opponent will always be presented with a P state, by the same sort of inductive argument. Watch out for states in other games that can be deduced to be P states via mirroring!

When there are more than two piles, nim is not as easy to analyse. For example, the state $\{1,2,3\}$ is a P-state, although this is not completely obvious.[34] And therefore, for example, a state of the form $\{1,2,n\}$ for $n > 3$ is always an N-state.

[34] Try to convince yourself by drawing the game graph and using existing knowledge about states with one or two piles.

However, it turns out that any state of nim can be completely analysed, and there is an easy algorithm to figure out if the state is an N state or a P state. The answer, which is beautiful and somewhat mysterious, lies in the binary expansions of the pile sizes.

To get to the answer, we first recall some facts. Recall that the binary expansion of a non-negative integer is obtained by successively subtracting the largest power of 2 from that integer until we reach zero, and recording "1" for each power we subtract, and "0" for each power that we don't. We will usually use a subscript of 2 to indicate a binary representation; for example, $5 = 101_2$.

Moreover, we have the following binary[35] operation on non-negative integers, which we call either the *nim-sum* or XOR.

Definition 56. *The* nim-sum *of two non-negative integers m and n, denoted $m \oplus n$, is the bitwise XOR of their binary representations.*

We explain this definition via an example. Take $m = 5$ and $n = 15$. We have seen that $m = 101_2$, and $n = 1111_2$. To compute

Example 55. *Since* $15 = 1 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 1 \cdot 1$, *its binary representation is* 1111_2. *Since* $16 = 1 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 0 \cdot 1$, *its binary representation is* 10000_2.

[35] *Binary* here refers to the number of inputs to this operation, not the base for the representation!

$m \oplus n$, we line up the binary representations aligned by binary place (that is, right aligned), and in each column, take the XOR of the bits, as follows.[36]

$$\begin{array}{r} 1\,0\,1 \\ \oplus\, 1\,1\,1\,1 \\ \hline 1\,0\,1\,0 \end{array}$$

[36] Writing bitwise XOR also as $\oplus$, recall that $1 \oplus 1 = 0 \oplus 0 = 0$, and that $1 \oplus 0 = 0 \oplus 1 = 1$.

Since $1010_2 = 1 \cdot 8 + 1 \cdot 2 = 10$, we see that $5 \oplus 15 = 10$.

We note some properties of the nim-sum operation.

1. Nim-sum is commutative: $m \oplus n = n \oplus m$. This is clear, because (bitwise) XOR is commutative.
2. Nim-sum is associative: $(m \oplus n) \oplus k = m \oplus (n \oplus k)$. This is clear, because (bitwise) XOR is associative.
3. The number 0 is the identity for nim-sum: $0 \oplus m = m \oplus 0 = m$ for every m.
4. Every number is its own inverse under nim-sum: $m \oplus m = 0$ for every m. This is because bitwise XOR has the same property. As a consequence, if we have $a \oplus b = c$, then by adding b to both sides, we see that $a = b \oplus c$, and similarly, $b = a \oplus c$.
5. Every number has a unique inverse: if $m \oplus n = 0$, then $m = n$. To see this, add m to both sides of the equation above, to get $m \oplus m \oplus n = m$, which means that $n = m$.

It turns out that given a nim state $\{n_1, \ldots, n_k\}$, it is useful and important to keep track of the nim-sum of the elements of the state (for short, the nim-sum of the state), which is $n_1 \oplus \cdots \oplus n_k$. The following lemma makes this precise.

Lemma 57. *Suppose that $\{n_1, \ldots, n_k\}$ is a nim state with each $n_i > 0$ and nim-sum 0. Then every possible move from this state will result in a state with a non-zero nim-sum.*

Proof. Suppose that $n_1 \oplus \cdots \oplus n_k = 0$. Consider any move from this state; without loss of generality, suppose that we change n_1 to n_1' after removing some stones. Let us compute the new nim-sum; it is

$$n_1' \oplus \cdots \oplus n_k = (n_1' \oplus \cdots \oplus n_k) \oplus (n_1 \oplus \cdots \oplus n_k),$$

because we know that $0 = n_1 \oplus \cdots \oplus n_k$. Now we cancel the pairs $n_i \oplus n_i$ for $i \neq 1$, to see that

$$n_1' \oplus \cdots \oplus n_k = n_1' \oplus n_1.$$

We know that $n_1' < n_1$, so these are distinct numbers. Their nim-sum cannot be zero! □

It turns out that the converse direction is true as well.

Lemma 58. *Suppose that $\{n_1, \ldots, n_k\}$ is a nim state with each $n_i > 0$, and nim-sum $s > 0$. Then there is* some *move that results in a state with zero nim-sum.*

Proof. We give the proof with a simultaneous running example. Consider the state $\{3, 6, 7\}$. The binary representations of these numbers are 11_2, 110_2, and 111_2 respectively. In general, consider the binary representations of the numbers n_i.

The nim-sum in the example is $010_2 = 2$, which is non-zero.

$$\begin{array}{r} 011 \\ 110 \\ \oplus\, 111 \\ \hline 010 \end{array}$$

In general, since $s > 0$, we see that s contains at least one "1" in its binary representation. In particular, the left-most 1 in s arises precisely because the column above it has an odd number of "1"s. In the example, the second column from the right has this feature; it contains three "1"s, which XOR to produce the "1" we see in s.

Now choose any of the n_i that contain a "1" in the column corresponding to the leftmost "1" in s. Let us call this column C. For instance, in the example, we could choose n_i to be $n_2 = 6$. Consider $n_i' = n_i \oplus s$. In our example, we get $6 \oplus 2 = 4$. Note that $n_i' \oplus s$ has a 0 in column C, and all columns to the left of column C are unchanged in n_i', because s only has zeroes in any column to the left of C.

This implies that n_i' must be less than n_i. This is because we are flipping a "1" in its binary representation to zero, without changing anything to the left of that "1": regardless of what changes happen to the right of this "1", the resulting value must decrease. So $n_i' < n_i$, and hence it is a valid nim move to change n_i to n_i'.

At the same time, let us compute the new nim-sum. Since $n_i' = n_i \oplus s$, it is

$$n_1 \oplus \cdots \oplus n_i' \oplus \cdots \oplus n_k = n_1 \oplus \cdots \oplus (n_i \oplus s) \oplus \cdots \oplus n_k.$$

Moving s out to the left and recalling that $n_1 \oplus \cdots \oplus n_k = s$, we see that

$$n_1 \oplus \cdots \oplus n_i' \oplus \cdots \oplus n_k = s \oplus s = 0.$$

This completes the proof. □

Note by the previous lemmas that in the game graph,

1. every state with zero nim-sum only points to states with positive nim-sum, and
2. every state with positive nim-sum points to *some* state with zero nim-sum.

Furthermore, the empty state, which is the only terminal position, clearly has zero nim-sum, and is a P-state. By following the algorithm of strategic labelling on game graphs, we have proven the following theorem.[37]

Theorem 59. *A nim state is a P-state if and only if it has zero nim-sum, and an N-state if and only if it has positive nim-sum.*

[37] The technical name for such an argument is *structural induction* on the game graph. Because the game graph is directed acyclic and finite, and because the N/P labelling is defined only in terms of states that come after a given state, we can build up the labelling from the labellings of the terminal states. Then at each step, we compare what we do to give the N/P label with the outputs of the two lemmas.

Grundy labelling

In this section we refine the idea of strategic labelling in order to get more information about games.

We start with some basic observations about *sums of games*.

Definition 60. *Let G and H be combinatorial games. Then $G + H$ is defined to be the game whose game state is a disjoint union of the game states of G and of H. Making a move in the game $G + H$ means that you either make a single move either in G or in H (but not both).*

We will think about the following question: can we deduce the outcome of $G + H$ if we know the outcomes of G and H?

Let us start with some easy cases. Let $\emptyset$ denote the "empty game": this is the game whose game state is the empty set, and there are no possible moves. It is clear that the outcome of the game $\emptyset$ is P. Now it is also clear that the outcome of $G + \emptyset$ equals the outcome of G. This is because the game graph of $G + \emptyset$ is the same as the game graph of G, since there are no possible moves in the $\emptyset$ game.

What about if we add a game G to itself?

Proposition 61. *Let G be any impartial combinatorial game. The outcome of $G + G$ is always P.*

Proof. Informally, this is because one can use a mirroring strategy. If player 1 makes a move $G \to H$ in (say) the first copy of G, player 2 can make the same move $G \to H$ in the second copy of G. Continuing in this manner, player 1 will be the first one to run out of moves.

More formally, we can use structural induction on the game graph of G. For the base case, consider any terminal position of G, which is the same game as $\emptyset$. Now we know from the previous observation that the outcome of $\emptyset + \emptyset$ is the same as the outcome of $\emptyset$, which is P.

Suppose we know that for any position H reachable from G such that $G \neq H$, the outcome of $H + H$ is P. Starting at $G + G$, the possible reachable positions are $G + H$ for any move $G \to H$, or $H + G$ for any move $G \to H$. Let us show that the outcome of $G + H$ (and hence $H + G$) is N for any possible move $G \to H$. Recall that the outcome of $H + H$ is P, and there is a move from $G + H$ to $H + H$, namely, by making the move $G \to H$ in the first coordinate, namely in G. Therefore in the game graph of $G + G$, the position $G + H$ (and hence $H + G$) is labelled N, for every possible move $G \to H$. But the only arrows from the position $G + G$ are to positions of the form $G + H$ or $H + G$ as above. Therefore, $G + G$ is a P position as well. □

Now, what happens if we add two possibly different games together? Let us start with some examples. Consider the nim game $G = \{1, 2\}$ and $H = \{3\}$. These are both N positions. When we add two nim games, we simply get another, bigger nim game. So

the game $G + H$ is just the nim game with state $\{1, 2, 3\}$. However, we have seen that $G + H$ is a P position because its nim sum is $1 \oplus 2 \oplus 3 = 0$. So in this example, we added two N games to obtain a P game.

On the other hand, if we now take $G = \{1, 2\}$ and $H = \{4\}$, then $G + H$ has nim-sum $1 \oplus 2 \oplus 4 = 7 \neq 0$. So in this case, $G + H$ is an N game!

We observe that if G and H are two N games, then the outcome of $G + H$ may be either N or P. However, it is a powerful fact that if we take the sum $G + H$ where H is a P game, then the outcome of $G + H$ is determined by the outcome of G.

Theorem 62. *Let H be a P game and G be any game. Then the outcome of $G + H$ is the same as the outcome of G.*

Proof. We can see this informally as follows. Suppose that G is an N game. Then player 1 has a winning strategy in $G + H$, as follows. Player 1 should start by making an optimal move in G, sending $G \to G'$ where G' is a P game. Now if player 2 makes a move in H as $H \to H'$, we know that H' is an N position, so player 1 can counter it with an optimal move in H'. If player 2 makes a move in G' as $G' \to G''$, we know that G'' is an N position, so player 1 can counter it with an optimal move in G''. After each pair of moves, player 2 is presented with a game state of the form $A + B$ where A and B are both P games. No matter which move player 2 makes, player 1 can counter it to once again give player 2 a state of the form (P, P). The game eventually terminates with player 2 running out of moves.

If G is a P game, then $G + H$ is of the form (P, P). By reversing the previous argument, we see that no matter which move player 1 makes, player 2 can always counter it so that player 1 always has a game state of type (P, P). Thus player 2 has a winning strategy.

More formally, we can use induction on the game graphs again. The base case is that one of the games is terminal (or empty), in which case we know the result. By induction, suppose that for every possible move from $G + H$, which goes a state of type either (x, P) or (P, x), we know that the outcome of the resulting state is x.

First suppose that G is an N position. Then there is a move $G \to G'$ such that G' is a P position, so that $G' + H$ has type (P, P). Therefore by the inductive hypothesis, $G' + H$ has outcome P. Due to the existence of the arrow $G + H \to G' + H$, the position $G + H$ is an N position.

Now suppose that G is a P position. Then a move from $G + H$ either goes to $G' + H$, which is of type (N, P), or $G + H'$, which is of type (P, N). In either case, the outcome of the resulting state is N by the inductive hypothesis. We conclude that $G + H$ is a P position as well. □

Let us introduce a better labelling system on the game graph, which will more accurately capture the outcomes of game addition.

The upshot of all this is that adding a P game to any game preserves the outcome, while adding an N game may change the outcome. The moral of the story is that all P games behave the same, while there are different kinds of N games. The Grundyl labelling is a way to distinguish between these different kinds of N games.

This is called *Grundy labelling*. First we define an operation called *mex*.

Definition 63. *Let $S = \{s_1, \ldots, s_k\}$ be a finite set of non-negative integers. The* minimum excluded *or* mex *of S, denoted* $\mathrm{mex}(X)$*, is the minimum non-negative integer that is not in S.*

Example 64. *If $S = \{0,1,2\}$ then $\mathrm{mex}(S) = 3$ If $S = \{0,2,4\}$, then $\mathrm{mex}(S) = 1$. If $S = \{4, 2000, 50\}$, then $\mathrm{mex}(S) = 0$.*

Definition 65. *The Grundy labelling is a labelling of a game graph, which takes values in positive integers. It is defined inductively as follows.*

1. *All terminal states are labelled by* 0*.*
2. *Consider a state G such that all states G' that G points to have been labelled. Let S be the set of labels of all G' such that there is an arrow $G \to G'$. Then label G by* $\mathrm{mex}(S)$*.*

Proposition 66. *The Grundy labelling enhances the N/P labelling. More precisely, a position is a P position if and only if its Grundy label is zero. More precisely, a position is an N position if and only if its Grundy label is positive.*

Proof. Once again, we use structural induction on the game graph, and compare both labelling methods. For terminal positions, the proposition is clear: they are P positions, and their Grundy label is always zero. Suppose we know the result for all positions reachable from a given game state G.

Suppose G is an N state. G will be labelled N if and only if there is an arrow $G \to G'$ where G' is labelled P. Since we know that the Grundy label of G' is zero, we see that the set S of all labels following G contains zero, and hence its mex must be positive. So the Grundy label of G is positive.

Suppose G is a P state. For every arrow $G \to G'$, the outcome of G' is N, and hence its Grundy label is positive. Since 0 does not appear among the Grundy labels of the possible G', we see that the Grundy label of G must be zero. This completes the proof. □

It turns out that Grundy labels are extremely useful in terms of computing outcomes, because they behave well with respect to game addition!

Theorem 67. *Let G and H be games with Grundy labels g and h respectively. Then the Grundy label of $G + H$ is $g \oplus h$.*

Proof. Let $s = g \oplus h$. Let S be the set of Grundy labels of $G' + H$ and $G + H'$, for arrows $G \to G'$ and $H \to H'$. By the inductive hypothensis, we know that these labels are precisely $g' \oplus h$ and $g \oplus h'$, where g' and h' are the Grundy labels of G and H respectively. Moreover, we know that g is the mex of all possible g', and h is the mex of all possible h'. Let us show that $s = g \oplus h$ is the mex of S.

First, let us show that $s \notin S$. If s were in S, then we would either have $s = g' \oplus h$ for some g', or $s = g \oplus h'$ for some h'. The two cases are symmetric, so we only tackle the first one. In that case we have

$g \oplus h = g' \oplus h$, which means (by adding h to both sides) that $g = g'$. This is not possible.

Next, let us show that if $s' < s$, then $s' \in S$. If $s = 0$, there is nothing to prove, so suppose that $s > 0$. Consider the triple sum $g \oplus h \oplus s'$, which is non-zero. By our arguments in the nim section, we know that there is a possible move in either g, h, or s', such that the resulting nim-sum is zero. Note that because $s' < s$, any nim move that decreases s' takes it to some s'' such that $s'' < s' < s$, and $s'' \oplus g \oplus h = s'' \oplus s$ cannot be zero. So the optimal nim move *does not* decrease s'; instead it either decreases g or h. WLOG suppose it takes g to g'. Then we know that $g' < g$, and that $g' \oplus h \oplus s' = 0$, that is, $s' = g' \oplus h$. Now g' must be one of the labels of the games G' reachable from G, because $g' < g$ and g was the mex of all possible g'. Hence $g' \oplus h$ is the Grundy label of some game $G' + H$, reachable from $G + H$. Therefore $s' \in S$!

Since $s \notin S$ and for every $s' < s$ we have $s' \in S$, we see that $s = \text{mex}(S)$. □

Bibliography