

ASILATA BAPAT

# GAMES, GRAPHS, AND MACHINES

# 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<sup>1</sup>: 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.

The Zermelo–Fraenkel axioms<sup>2</sup> 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  $x, y$ <sup>3</sup>. 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  $\{x \in \mathbb{N} \mid x \text{ is even}\}$ .

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<sup>4</sup> a subset of every set.

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  $T$ .

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

**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

<sup>1</sup> A situation in which exactly one of two possible options is true.

<sup>2</sup> Historical remarks and something about ZFC?

<sup>3</sup> 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.

<sup>4</sup> 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.

### Example 1.

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

### 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.  $\{1, 2\} \times \emptyset = \emptyset$ .

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.

## 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*<sup>5</sup>.

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  $S = T = \mathbb{N}$ . If we want  $R$  to model the relation *is-factor-of*, then we take  $R$  to be the subset of  $\mathbb{N} \times \mathbb{N}$  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)<sup>6</sup> relation on  $S$ .

## Functions

Informally, a function is a rule that can be used to find the output value given a certain input value. This can be formally expressed using relations, as follows. Let  $R \subset S \times T$  be a relation. We say that  $R$  is a function if whenever  $(s, t) \in R$  and  $(s, u) \in R$ , we have  $t = u$ . In other words, any first coordinate has at most one possible second coordinate. In this case, we often write  $t = R(s)$  or often  $t = f(s)$ . We also have the following definitions.

*Domain* The *domain* of this function is the set

$$\{x \in S \mid (x, y) \in R \text{ for some } y \in T\}.$$

*Codomain (or range)* The *codomain* of this function is the set

$$\{y \in T \mid (x, y) \in R \text{ for some } x \in S\}.$$

If  $S'$  is the domain and  $T'$  is the range, we usually say that  $f$  is a function from  $S'$  to  $T'$ , written  $f: S' \rightarrow T'$ .

## Graphs

Graphs provide an extremely useful way to organise information about relations. For the moment we use them as powerful visual

<sup>5</sup> In English, we might read one of these as “3 is a factor of 27”.

<sup>6</sup> 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  $S$ .

### Example 3.

1. The relation  $\{(a, b) \in \mathbb{N} \times \mathbb{N} \mid a + b \text{ is even}\}$  is not a function because, for example,  $(2, 4)$  and  $(2, 0)$  are both in it.
2. The relation  $\{(a, b) \in \mathbb{N} \times \mathbb{N} \mid b = a^2\}$  is a function.

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  $(w, v)$ .

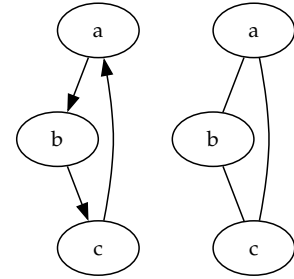


Figure 1: A directed and an undirected graph

### 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  $i$ th row and  $j$ th column is denoted  $M_{ij}$ .

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

**Definition 4.** Suppose  $G = (V, E)$  is a graph<sup>7</sup>. Choose an ordering on the elements of  $V$ , say the ordered tuple  $(v_1, \dots, v_n)$ . The *adjacency matrix* of  $G$  with respect to the chosen ordering is the  $n \times n$  matrix  $A$ , defined by

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

The adjacency matrix is a matrix that only contains the elements 0 and 1. It encodes the entire information contained in the original 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 series of simultaneous swaps of corresponding row and column numbers. For example, the adjacency matrix given by the ordering  $(v_2, v_1, v_3, \dots, v_n)$  can be obtained from  $A$  by swapping rows 1 and 2 and also swapping columns 1 and 2.

<sup>7</sup> For simplicity we usually consider finite sets  $V$  when we construct adjacency matrices but in general  $V$  may be infinite.

**Example 5.** 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  $(c, b, a)$ , the adjacency matrix becomes

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

## 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<sup>8</sup>. 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.

## 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<sup>9</sup>? To make this precise, we give the following definition.

### Example 6.

1. The relation

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

is reflexive, anti-symmetric, and transitive.

2. The relation

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

is symmetric but not reflexive or transitive.

<sup>8</sup> Convince yourself of this from the definitions!

<sup>9</sup> 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 7.** A reflexive (resp. symmetric, transitive) closure of  $R$  is a set  $\bar{R}$  with the following properties.

1.  $R \subset \bar{R} \subset S \times S$ .
2.  $\bar{R}$  is reflexive (resp. symmetric, transitive).
3. If  $T$  is a subset of  $S \times S$  such that  $R \subset T \subsetneq \bar{R}$ , 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<sup>10</sup>. We won't prove this formally, but instead we will just produce a construction of each.

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<sup>11</sup>: whenever  $A_{ij} = 1$ , we also set  $A_{ji} = 1$ .

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.

<sup>10</sup> 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?

<sup>11</sup> This is the same as taking  $\frac{1}{2}(A + A^t)$ . Do you see why?

# 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 8.** A equivalence relation is one that is reflexive, symmetric, and transitive.

**Example 9.** Let  $R$  be the relation on  $\mathbb{Z}$  defined as

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

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*<sup>12</sup>.

**Definition 10.** 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:

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

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

**Proposition 11.** 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<sup>13</sup> the set  $S$  into disjoint subsets whose union is  $S$ .

*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

<sup>12</sup> The idea is that we can treat all elements of one equivalence class as being interchangeable in some sense.

In Example 9,  $a \sim b$  if and only if they have the same parity, so there are two equivalence classes of  $R$  on  $\mathbb{Z}$ , 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  $[1]$ .

<sup>13</sup> If  $S = S_1 \cup \dots \cup S_n$ , we say that it is a *partition* if  $S_i \cap S_j = \emptyset$  for  $i \neq j$ . In this case we write  $S = S_1 \sqcup \dots \sqcup S_n$ , or more concisely,  $S = \bigsqcup_{i=1}^n S_i$ .

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.  $\square$

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.

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

### Modular arithmetic

As an important application of equivalence classes, we briefly study modular arithmetic. First recall the relation from Example 9. 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<sup>14</sup>  $\{[0], [1]\}$ . We will simply define this addition using the four properties above, which can be written more concisely as

$$[a] + [b] := [a + b] \text{ for each } a, b \in \mathbb{Z}.$$

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

More generally, fix a *modulus*  $d \in \mathbb{N}$ . 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  $m \in \mathbb{Z}$  such that  $x - y = md$ .

In this case, we have equivalence classes  $[0], [1], \dots, [d-1]$ . 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

<sup>14</sup> Note that this set is *not* equal to  $\mathbb{Z}$ ! It is also not equal to the set  $\{0, 1\}$ . Instead this is a set with two elements, which are themselves subsets of  $\mathbb{Z}$ .

**Exercise 12.** Check that  $\sim_d$  is an equivalence relation.



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  $[0], [1], \dots, [d-1]$ . Of course these can be represented by different integers. For example,  $[1] = \{\dots, 1-2d, 1-d, 1, 1+d, 1+2d, \dots\}$ , so any of these elements would do as a representative of  $[1]$ . We will write  $\mathbb{Z}/d\mathbb{Z} = \{[0], \dots, [d-1]\}$  to be the set of equivalence classes in this case.

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

$$[a] + [b] := [a + b].$$

We now have to check whether this is *well-defined*<sup>15</sup>. 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.

<sup>15</sup> 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!*  $[a] + [a] + \dots + [a]$  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

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

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

# Graphs

## Overview

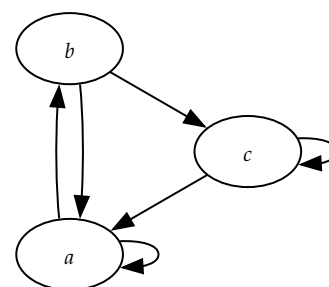
Let us recall the definitions. A (directed) graph consists of a vertex set  $V$  and an edge set  $E \subset V \times V$ . If  $(a, b) \in E$ , we also write  $a \rightarrow b$  as a directed edge. Typically we consider finite vertex sets when we work with concrete examples. An *undirected* graph is 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.

## 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<sup>16</sup>. 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.)

**Example 14.** The drawing of a graph where the vertex set is  $V = \{a, b, c\}$  and the edge relation is  $E = \{(a, a), (a, b), (b, a), (b, c), (c, c), (c, a)\}$ .



<sup>16</sup> 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 4). Given a graph  $(V, E)$ , first we order the set  $V$  into a tuple  $(v_1, \dots, v_n)$ . Then we create an  $n \times n$  matrix  $A$  such that  $A_{ij} = 1$  if  $i \rightarrow j$  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:

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

### Powers of the adjacency matrix

Consider the example directed graph shown in Figure 2. The adjacency matrix and its square are

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

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

**Example 15.** Suppose that

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

Then

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

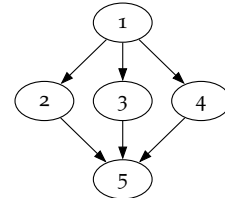


Figure 2: A directed graph

*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  $v_\ell$ , and  $A_{\ell,j}$  is the number of edges from  $v_\ell$  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  $v_\ell$ . Since we add over all possible vertices  $v_\ell$ , 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  $v_j$ .  $\square$

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 17.** *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  $\{0, 1\} \times \{0, 1\} \rightarrow \{0, 1\}$ .

*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<sup>17</sup>:

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

<sup>17</sup> 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):

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

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  $\ell$ th such entry equals 1 if and only if there is an edge from  $i$  to  $\ell$  and also an edge from  $\ell$  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 16 and so we omit it.

**Proposition 18.** *Let  $A$  be the adjacency matrix of a simple directed graph  $(V, E)$ . Suppose that the vertices are ordered as  $(v_1, \dots, v_n)$ . Then the entry in the  $(i, j)$ th position of the  $k$ th 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  $w: E \rightarrow \mathbb{R}$ , 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  $\infty$ <sup>18</sup>.

**Definition 19.** *Let  $G = (V, E)$  be a directed graph with weight function  $w: E \rightarrow \mathbb{R}$ . Suppose that the vertices are ordered as  $(v_1, \dots, v_n)$ . The*

<sup>18</sup> We use the symbol  $\infty$  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  $i \rightarrow \ell \rightarrow j$ , add the edge weights of  $i \rightarrow \ell$  and  $\ell \rightarrow j$ , and then take the minimum. This operation is extremely similar to the standard matrix product, except that instead of multiplying the  $(i, \ell)$ th entry with the  $(\ell, j)$ 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 21.** 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  $\infty$ . The “min-plus” product of  $A$  and  $B$ , denoted  $A \odot B$ , is defined as follows (entry-wise):

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

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 16, and is omitted.

**Proposition 23.** Let  $W$  be the weighted adjacency matrix of a weighted graph with  $n$  vertices.

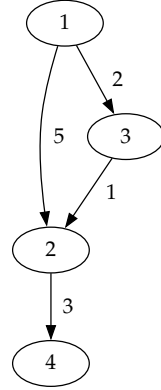
1. The  $(i, j)$ th entry of  $W^{\odot k}$  is the weight of the minimum-weight path from  $i$  to  $j$  that has at most  $k$  edges.
2. If all the edge weights are non-negative, then the  $(i, j)$ th entry of  $W^{\odot(n-1)}$  is the weight of the minimum-weight path (with any number of edges) from  $i$  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 20.** 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 22.** For the graph in Example 20, 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<sup>19</sup>, and then multiply together the corresponding powers of  $A$  to get the final result. Here is an example.

**Example 24.** 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.*

<sup>19</sup> 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,

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

# Partial orders

In this section we return to another important kind of relation, called *partial orders*. These are entirely different in flavour from equivalence relations, and are very useful to formalise. Once we cover the preliminaries, we will also see a few applications of the theory of partial orders.

**Definition 25.** 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.

**Example 26.** 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 28.** 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  $b \preceq a$ .

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

- The usual inequality ordering on  $\mathbb{N}$ ,  $\mathbb{Z}$ ,  $\mathbb{Q}$ , or  $\mathbb{R}$ , 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  $\mathbb{N}$ , 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.

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.

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  $b \preceq a$ .

**Exercise 27.** Find examples to show that the partial order of Example 26 is not usually a total order.

**Exercise 29.** 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 3.

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 already 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 4.

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.

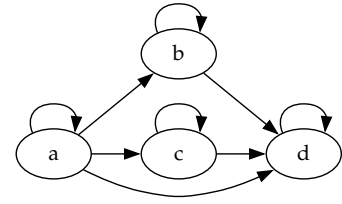


Figure 3: The graph of a partial order relation

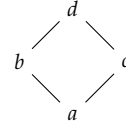


Figure 4: The Hasse diagram of the partial order relation from Figure 3.

## Upper and lower bounds

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

**Definition 30.** 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 31.

**Definition 32.** 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 31. However, if they exist, they are unique.

**Exercise 33.** 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  $P$ .

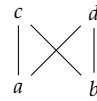


Figure 5: The Hasse diagram of the “bow-tie” poset

**Example 31.** Let  $(P, \preceq)$  be the bow-tie poset shown in Figure 5. 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 34.** 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  $y \in P$ .

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  $P$ .

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 5.

### *Incidence algebra*

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.

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

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

**Definition 35.** The *incidence algebra* of the poset  $P$  is defined as the set of all functions from  $I(P)$  to  $\mathbb{R}$ :

$$\mathcal{A}_P = \{f: I(P) \rightarrow \mathbb{R}\}.$$

**Example 36.** 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  $\delta$  to be the Kronecker delta function:

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

3. Set  $\zeta$  to be the function that sends every closed interval to 1:

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

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

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  $r \in \mathbb{R}$ , and  $f \in \mathcal{A}_P$  we define their scalar product  $rf$  as the following element<sup>20</sup> of  $\mathcal{A}_P$ :

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

<sup>20</sup> 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  $\delta$  is the identity element for the convolution product<sup>21</sup>.

**Proposition 37.** *Let  $(P, \preceq)$  be any finite poset, and let  $f$  be an element of  $\mathcal{A}(P)$ . Then*

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

<sup>21</sup> In particular, since multiplicative identities are unique, the zeta function is *not* the multiplicative identity for the convolution product!

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

### Functions on posets and one-sided convolution

Let  $(P, \preceq)$  be any finite poset. A *function on the poset  $P$*  is simply a function

$$p: P \rightarrow \mathbb{R}.$$

If  $f \in \mathcal{A}_P$  and  $p: P \rightarrow \mathbb{R}$ , then we can define two *one-sided convolutions*,  $f * p$  and  $p * f$ . Both of these will be new functions on  $P$ . The definitions are as follows<sup>22</sup>:

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

<sup>22</sup> Remember that elements of  $\mathcal{A}(P)$  take in intervals as inputs, and functions on posets take elements of the poset as inputs.

### The matrix representation of the incidence algebra

The convolution product on  $\mathcal{A}_P$  does not seem very intuitive at first glance, and it is not clear why it might be useful. To understand the motivation behind this, we look at the *matrix representation* of  $\mathcal{A}_P$ . First, it will be useful to sort the elements of  $P$  in a nice way, in order to be able to write down matrices such as the adjacency matrix. Since  $P$  already has a partial order on it, it is most natural to sort the elements of  $P$  going “bottom to top along its Hasse diagram”. More formally, this means that we should sort the elements so that whenever  $x \preceq y$ , we put  $x$  before  $y$  in our total sorting. This is called a *topological sorting* of  $P$ .

**Definition 39.** Let  $(P, \preceq)$  be a poset. An ordering  $x_1, \dots, x_n$  of the elements of  $P$  is called a *topological sorting* if whenever  $x_i \preceq x_j$ , we have  $i \leq j$ .

Note that  $P$  may have several different valid topological sortings! To write down the matrix representation of  $\mathcal{A}_P$ , fix a sorting  $x_1, \dots, x_n$  on  $P$ . The matrix representation depends on the chosen sorting, and looks particularly nice if we choose a topological sorting, but this is not necessary.

**Definition 41.** Let  $f \in \mathcal{A}_P$ . The matrix corresponding to  $f$  (with respect to the chosen sorting) is defined to be an  $n \times n$  matrix  $M_f$ , with the following entries:

$$(M_f)_{i,j} = \begin{cases} f([x_i, x_j]), & x_i \preceq x_j \\ 0, & \text{otherwise.} \end{cases}$$

Similarly, we have a vector associated to any function on a poset.

**Definition 42.** Let  $p: P \rightarrow \mathbb{R}$ . The vector corresponding to  $p$  in the chosen sorting is defined as follows:

$$v_p = \begin{pmatrix} p(x_1) \\ \vdots \\ p(x_n) \end{pmatrix}.$$

We also often consider the transpose of this vector:

$$v_p^t = (p(x_1) \quad \cdots \quad p(x_n)).$$

It turns out that the three operations we defined on the incidence algebra translate into already-familiar operations on matrices. The next theorem shows that addition and convolution product of elements in the incidence algebra becomes addition and matrix product of the corresponding matrices. Similarly, one-sided convolution turns into a matrix-vector product of the corresponding matrix and vector.

**Theorem 44.** Let  $f, g \in \mathcal{A}(P)$ , and let  $p: P \rightarrow \mathcal{A}(P)$ . Then the following hold.

In a topological sorting  $x_1, \dots, x_n$ , having  $i \leq j$  does not necessarily imply that  $x_i \preceq x_j$ ! However if  $i \leq j$  and  $x_i \not\preceq x_j$ , then it must be the case that  $x_i$  and  $x_j$  are incomparable.

**Exercise 40.** A square matrix is called *upper-triangular* if every entry below the main diagonal is zero. Check that if the chosen sorting  $x_1, \dots, x_n$  of the elements of  $P$  is a topological sorting, then for any  $f \in \mathcal{A}(P)$ , the corresponding matrix  $M_f$  is upper-triangular.

Note that if  $P$  has  $n$  elements, then the matrix associated to any element of the incidence algebra is an  $n \times n$  matrix. The vector  $v_p$  associated to a function  $p$  on the poset is an  $n \times 1$  matrix (or column vector), and the transpose vector  $v_p^t$  is a  $1 \times n$  matrix (or row vector).

**Exercise 43.** Check that the operations described in the theorem make sense in terms of numerics: what will be the sizes of the matrices obtained from the operations on the right hand side of each equation in the theorem?

1.  $M_{f+g} = M_f + M_g$ .
2.  $M_{f*g} = M_f \cdot M_g$ .
3.  $v_{f*p} = M_f \cdot v_p$ .
4.  $v_{p*f}^t = v_p^t \cdot M_f$ .

*Proof.* We prove each of the four assertions.

1. The first one is clear: the  $(i, j)$ th entry of  $M_f$  is  $f([x_i, x_j])$ , while the  $(i, j)$ th entry of  $M_g$  is  $g([x_i, x_j])$ . Their sum is clearly the  $(i, j)$ th entry of  $M_{f+g}$ .
2. Consider the  $(i, j)$ th entry of  $M_f M_g$ , which can be expanded into a matrix product as follows:

$$\sum_{1 \leq k \leq n} (M_f)_{i,k} (M_g)_{k,j}.$$

Note that  $(M_f)_{i,k} = 0$  unless  $x_i \preceq x_k$ , and otherwise it is  $f([x_i, x_k])$ . Similarly,  $(M_g)_{k,j} = 0$  unless  $x_k \preceq x_j$ , and otherwise it is  $g([x_k, x_j])$ . So the sum above reduces to

$$\sum_{x_i \preceq z \preceq x_j} f([x_i, z])g([z, x_j]),$$

which is precisely the formula for  $(f * g)([x_i, x_j])$ . Therefore

$$M_f M_g = M_{f*g}.$$

3. Again the formula for the  $i$ th entry of  $M_f \cdot v_p$  is precisely

$$\sum_j (M_f)_{i,j} (v_p)_j.$$

The  $j$ th term is zero unless  $x_i \preceq x_j$ , and in that case it is  $f([x_i, x_j])p(x_j)$ . Therefore the formula becomes

$$\sum_{x_i \preceq z} f([x_i, z])p(z),$$

which is precisely  $(f * p)(x)$ .

4. This is similar to the previous case.

□

### *Invertibility and the Möbius function*

Recall that the element  $\delta \in \mathcal{A}(P)$  is the multiplicative identity with respect to the convolution product.<sup>23</sup>

**Definition 45.** We say that an element  $f \in \mathcal{A}(P)$  is right invertible if it has a multiplicative inverse with respect to convolution on the right hand side. That is, if there exists some  $g \in \mathcal{A}(P)$  such that

$$f * g = \delta.$$

<sup>23</sup> This means that for every  $f \in \mathcal{A}(P)$ , we have  $f * \delta = f = \delta * f$ .

We say that  $g \in \mathcal{A}(P)$  is left invertible if it has a multiplicative inverse with respect to convolution on the left hand side. That is, if there exists some  $f \in \mathcal{A}(P)$  such that

$$f * g = \delta.$$

In the setting of the definition above, we say that  $f$  is a left inverse of  $g$  and  $g$  is a right inverse of  $f$ .

We have the following theorem, which gives a criterion for checking when some  $f$  is invertible.

**Theorem 46.** Let  $f \in \mathcal{A}(P)$ . The following statements are equivalent.

1.  $f$  has a right inverse  $g$ .
2.  $f$  has a left inverse  $h$ .
3.  $f([x, x]) \neq 0$  for every  $x \in P$ .

Moreover, in this case, we have  $g = h$ , and that  $g = h$  is the unique left or right inverse of  $f$ .

*Proof.* We first check that  $1 \Rightarrow 2$  and  $2 \Rightarrow 3$ . Given a right inverse  $g$ , we have for each  $x \in P$  that

$$(f * g)([x, x]) = f([x, x])g([x, x]) = 1.$$

Therefore  $f([x, x]) \neq 0$ .

Similarly, given a left inverse  $h$ , we have for each  $x \in P$  that

$$(h * f)([x, x]) = h([x, x])f([x, x]) = \delta([x, x]) = 1.$$

Therefore  $f([x, x]) \neq 0$ .

Now let us check that  $3 \Rightarrow 1$ . Suppose that  $f([x, x]) \neq 0$  for every  $x \in P$ . Define an element  $g \in \mathcal{A}(P)$  inductively as follows. Set

$$g([x, x]) = \frac{1}{f([x, x])}$$

for every  $x \in P$ . Now suppose that we have  $x \prec y$  with  $x \neq y$ , and suppose that we have defined  $g([z, y])$  for every  $x \prec z \preceq y$ . Then set<sup>24</sup>

$$g([x, y]) = -\frac{1}{f([x, x])} \sum_{x \prec z \preceq y} f([x, z])g([z, y]).$$

Inductively,  $g$  is well-defined. Now let us check that  $f * g = \delta$ . First,

$$(f * g)([x, x]) = f([x, x]) \cdot \frac{1}{f([x, x])} = 1.$$

Next, let  $x \prec y$ .

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

From the formula above, it is clear that in this case,  $(f * g)([x, y]) = 0 = \delta([x, y])$ .

The proof for  $3 \Rightarrow 1$  is similar, and we obtain the following inductive formulas for a left inverse  $h$  of  $f$ <sup>25</sup>:

<sup>24</sup> Note that we get this formula exactly by back-solving the required equation, namely that  $(f * g) = \delta$ .

<sup>25</sup> Again, we get these formulas by back-solving the required equation, namely that  $(h * f) = \delta$ .

- for any  $x \in P$ , we have  $h([x, x]) = \frac{1}{f([x, x])}$ , and
- if  $x \prec y$ , we have

$$h([x, y]) = -\frac{1}{f([y, y])} \sum_{x \preceq z \prec y} h([x, z])f([z, y]).$$

Then  $h$  is well-defined, and the check that  $h * f = \delta$  is exactly analogous to the previous.

Finally, let us check that  $g = h$ . Consider the equation

$$h * f = \delta,$$

and multiply both sides by  $g$  on the right:

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

Associating the terms on the left hand side differently, we get

$$h * (f * g) = g,$$

but since  $f * g = \delta$ , we have  $h * \delta = h = g$ .

Finally, if  $g'$  is another right inverse of  $f$ , we can use the same argument to see that  $h = g'$ , and similarly if  $h'$  is another left inverse of  $f$  then we can use the same argument to see that  $h' = g$ .  $\square$

In the situation of the above theorem, we will say that  $f$  is *invertible*, and denote by  $f^{-1}$  the unique left and right inverse of  $f$ . Since the matrix representation of the incidence algebra preserves multiplication, it also preserves inverses, as we see in the following proposition.

**Proposition 47.** *If  $f \in \mathcal{A}(P)$  is invertible, then*

$$M_{f^{-1}} = (M_f)^{-1}.$$

*Proof.* We know that  $f * f^{-1} = \delta$ , and we also know that  $M_\delta = I$ , the identity matrix. So we have by Theorem 44 that

$$M_{f^{-1}} \cdot M_f = M_\delta = I.$$

This shows that the matrix  $M_f$  is invertible as a matrix, and by multiplying both sides on the right by  $(M_f)^{-1}$  that

$$M_{f^{-1}} = (M_f)^{-1}.$$

Therefore  $M_f$  is invertible, and its inverse is simply  $M_{f^{-1}}$ .  $\square$

In fact, the other direction is also true! (Although much less obvious.) We need two lemmas from linear algebra, which you can take as black boxes if you are not familiar with linear algebra.

**Lemma 48.** *An upper-triangular square matrix  $M$  is invertible if and only if each diagonal entry is non-zero.*

*Proof.* We won't prove this here; this is standard from linear algebra by either Gaussian elimination or computing the determinant.  $\square$

**Lemma 49.** Suppose that  $f \in \mathcal{A}(P)$ . Then the invertibility of  $M_f$  is independent of the sorting chosen on  $P$ . That is, if  $M_f$  is invertible with respect to one sorting of  $P$ , then it is invertible respect to every sorting of  $P$ .

*Proof.* Changing the ordering of the elements of  $P$  is a (particularly easy) change of basis on the space  $\mathbb{R}^n$ , and it conjugates  $M_f$  by a permutation matrix. Consequently, if  $A$  and  $B$  are matrices of  $f$  with respect to two different sortings, then they are similar matrices. So one is invertible if and only if the other one is.  $\square$

Now we are ready to state and prove the other direction.

**Proposition 50.** Suppose that  $f \in \mathcal{A}(P)$  such that  $M_f$  is an invertible matrix. Then  $f$  is invertible as an element of  $\mathcal{A}(P)$ , and  $M_{f^{-1}} = (M_f)^{-1}$ .

*Proof.* Fix some  $f \in \mathcal{A}(P)$  such that  $M_f$  is invertible as a matrix. For simplicity, we can assume by Lemma 49 that the chosen sorting on  $P$  is a topological sorting, so that  $M_f$  is upper-triangular.

For simplicity, we write  $M = M_f$ . Let  $N = M^{-1}$ . We now have to show the following.

1. We have to show that if  $x \not\preceq y$ , then  $N_{(x,y)} = 0$ .
2. We also have to show that if we define  $g \in \mathcal{A}(P)$  via  $g([x, y]) = N_{(x,y)}$  for every  $x \preceq y$ , then  $g$  is the inverse of  $f$  under the convolution product.

We start by showing the first statement. Suppose (for contradiction) that  $x \not\preceq y$ , and that  $N_{(x,y)} \neq 0$ . Fix  $y$  and by enlarging  $x$  if necessary, suppose that  $x$  is the maximal element with this property. Let us compute  $(M \cdot N)_{(x,y)} = I_{(x,y)} = 0$ .

$$(M \cdot N)_{(x,y)} = \sum_{z \in P} M_{(x,z)} \cdot N_{(z,y)}.$$

The only possible non-zero terms in the sum arise from  $z$  such that  $x \preceq z$ , otherwise  $M_{x,z} = 0$ .

On the other hand,  $x$  is the maximal element such that  $N_{(x,y)} \neq 0$ . So for all values of  $z$  such that  $x \prec z$ , we have  $N_{(z,y)} = 0$ . This shows that

$$(M \cdot N)_{(x,y)} = M_{(x,x)} \cdot N_{(x,y)} = 0.$$

Since we had chosen to represent  $M$  in a topological sorting of  $P$ , the matrix  $M$  is upper-triangular (and invertible). By Lemma 48, we have  $M_{(x,x)} \neq 0$ . This shows that  $N_{(x,y)} \neq 0$ , which is a contradiction.

Now we prove the second statement. Consider the same assumptions as before:  $f \in \mathcal{A}(P)$  such that  $M = M_f$  is invertible, and set  $N = M^{-1}$ . Define a new function  $g \in \mathcal{A}(P)$  as follows:

$$g([x, y]) = N_{(x,y)},$$

for every  $x \preceq y$ . We now check explicitly that  $g$  is the inverse of  $f$  under the convolution product.

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



By the previous proof, we know that if  $z$  does not satisfy  $x \preceq z \preceq y$ , then either  $M_{(x,z)} = 0$  or  $N_{(z,y)} = 0$ , and so the product  $M_{(x,z)}N_{(z,y)}$  is zero. Therefore we have

$$(f * g)([x, y]) = \sum_{z \in P} M_{(x,z)}N_{(z,y)} = (M \cdot N)_{(x,y)} = \delta([x, y]).$$

This calculation proves that  $g = f^{-1}$ . □

# 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 51.** An alphabet  $\Sigma$  is a finite set of symbols, called the letters of  $\Sigma$ . A string or a word is a finite ordered list of elements of  $\Sigma$ , written without spaces or punctuation. The length of a word is the number of letters in the word.<sup>26</sup>

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  $\epsilon$ .

If  $\Sigma$  is a fixed alphabet, then we denote by  $\Sigma^*$  the set of all strings, including  $\epsilon$ .

**Definition 53.** Fix an alphabet  $\Sigma$ . A language  $L$  on  $\Sigma$  is any subset of  $\Sigma^*$ .

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

## Basic constructions with strings

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

*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  $\Sigma$ , denoted by  $L_1 \circ L_2$  and defined as follows.

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

<sup>26</sup> The unique empty word is also allowed, and is denoted  $\epsilon$ . For this reason we usually assume that  $\epsilon$  is not a symbol in  $\Sigma$ .

**Exercise 52.** Check that if  $\Sigma = \emptyset$  then  $\Sigma^* = \{\epsilon\}$ , but otherwise  $\Sigma^*$  is infinite.

*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,

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

### Lexicographic order (dictionary order)

Suppose that we have ordered the elements of  $\Sigma$ . Then  $\Sigma^*$  (and any other language on  $\Sigma$ ) 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

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

where  $a_i, b_i$  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  $\Sigma$ , we say  $v < w$ . Otherwise if  $b_i < a_i$ , we say  $w < v$ .

### 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 56.** Fix an alphabet  $\Sigma$ . A word  $r$  written using the letters of  $\Sigma$ , together with the symbols  $*$  and  $|$ , is a valid regular expression if it satisfies one of the following.<sup>27</sup>

1.  $r = \emptyset$
2.  $r = \epsilon$
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  $s$ .

**Example 54.** 1. If  $L = \emptyset$  then  $L^* = \{\epsilon\}$ .

**Example 55.** Assume we use the order  $(0,1)$  on  $\Sigma = \{0,1\}$ .

1. The word  $\epsilon$  is shorter than every other word, so appears first in the lexicographic order on  $\Sigma^*$ .
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  $00$ .

<sup>27</sup> Additionally, we are also allowed to parenthesise subexpressions to avoid ambiguity. We assume that  $\Sigma$  does not contain any of the symbols “(”, “)”, “|”, “\*”, or “ $\emptyset$ ”.

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

**Definition 57.** Let  $\Sigma$  be an alphabet and let  $r$  be a regular expression on  $\Sigma$ . 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 58.** Fix an alphabet  $\Sigma$ . A deterministic finite automaton for  $\Sigma$  is described by the following pieces of data.

1. A (usually finite) set of states, usually denoted  $Q$ .
2. A start state<sup>28</sup>, usually denoted  $q_0 \in Q$ .
3. A set of accept states  $A \subseteq Q$ .<sup>29</sup>
4. A transition function

$$\delta: Q \times \Sigma \rightarrow Q.$$

The definition is not very illuminating. It is often much clearer to draw the *state diagram* of a finite automaton, as shown in Example 59. In this example, we can decode the formal data of the DFA as follows.

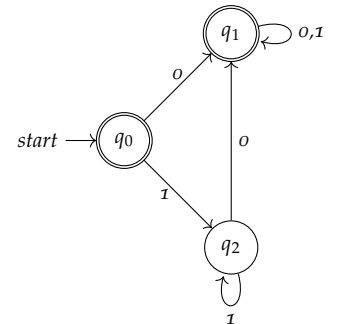
1. The set of states is  $Q = \{q_0, q_1, q_2\}$ .
2. The start state is  $q_0$ .
3. The set of accept states is  $A = \{q_0, q_1\}$ .
4. The transition function can be represented as a table as follows.

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	$q_2$

<sup>28</sup> The start state is always unique.

<sup>29</sup> 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 59.** 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  $\Sigma$ . 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 letter  $a_3$ .
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*  $w$ .

**Definition 61.** 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  $\Sigma$ . We also give ourselves the luxury of allowing arrows labelled by the empty string  $\varepsilon$ . If there is an arrow labelled  $\varepsilon$  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 63.** Fix an alphabet  $\Sigma$ . A nondeterministic finite automaton for  $\Sigma$  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<sup>30</sup>

$$\Delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q).$$

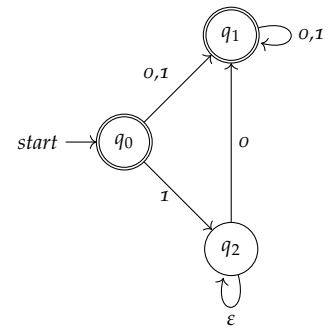
Once again, let us describe the parts of the definition for the example in Example 62. The set of states, the start state, and the set of accept states are exactly as in the previous example (Example 59). The transition function is as follows.

**Example 60.** 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  $w$ .

**Example 62.** Here is an example of a nondeterministic finite automaton.



<sup>30</sup> This transition function also takes in a pair  $(q, a)$  as input, where  $q \in Q$  and  $a$  is either a letter of  $\Sigma$  or the empty string  $\varepsilon$ . 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  $a$ .

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

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.

### **SHELVED** Calculation tree example

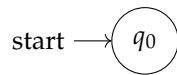
### **TODO** Relationship between NFAs and DFAs

### Regular expressions to finite automata

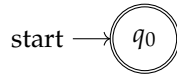
The aim of this section is to try and convert any given regex to an *equivalent*<sup>31</sup> finite automaton (either a DFA or an NFA). We have already seen that given any NFA, one can construct an equivalent (probably much larger) DFA. So to make things simpler for us, we will convert regexes to NFAs.

We do this inductively, constructor-by-constructor.

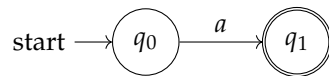
1. If  $r = \emptyset$ , we simply have to find an NFA that rejects every string. The easiest option is as follows.



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



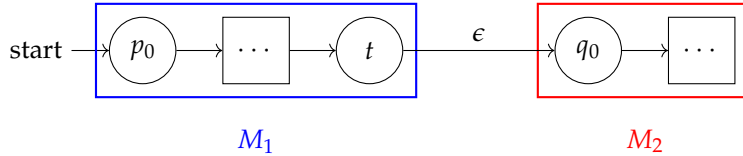
3. If  $r = a$  for some  $a \in \Sigma$ , we construct an NFA that only accepts the string  $a$ . A possible option is as follows.



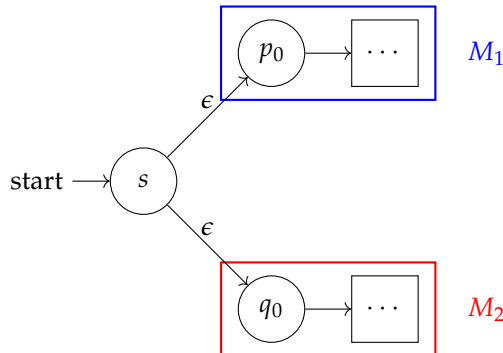
4. If  $r = r_1 r_2$  for two regexes  $r_1$  and  $r_2$ , we construct an equivalent NFA inductively. Assume that  $M_1$  and  $M_2$  are NFAs equivalent to  $r_1$  and  $r_2$  respectively. Assume furthermore for simplicity

<sup>31</sup> We say that a regex  $r$  is equivalent to a finite automaton  $M$  if  $L(r) = L(M)$ .

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  $\epsilon$ -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  $\epsilon$  transition, whose accepting states are simply the accepting states of  $M_2$ . This construction is illustrated below.

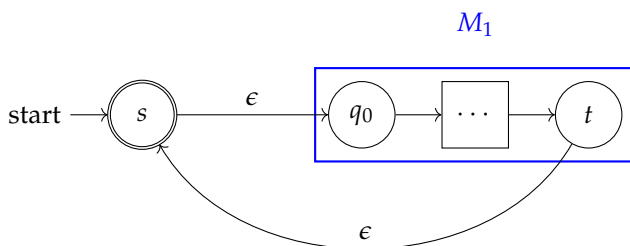


5. If  $r = r_1 \mid r_2$  for two regexes  $r_1$  and  $r_2$ , we construct an equivalent NFA inductively. Assume that  $M_1$  and  $M_2$  are NFAs equivalent to  $r_1$  and  $r_2$  respectively. Let  $p_0$  and  $q_0$  be the start states of  $M_1$  and  $M_2$  respectively. We construct a new automaton with start state  $s$ , which connects to both  $p_0$  and  $q_0$  by  $\epsilon$ -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  $\epsilon$  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

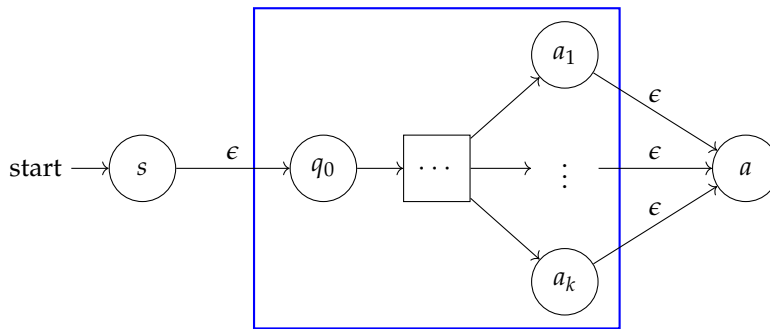
Recall that we say that two automata  $M_1$  and  $M_2$  are *equivalent* if  $L(M_1) = L(M_2)$ . We have already seen that DFAs and NFAs are equivalent in power. That is, for any DFA  $M$  there is an equivalent NFA  $N$ , and for any NFA  $N$  there is an equivalent DFA  $M$ .

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<sup>32</sup>.

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

Here is a visual representation.



The "operating box".

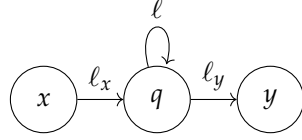
Once we have done this procedure, we get to work deleting each state of the machine inside the “operating box” one by one. We can’t however simply delete a state! We have to compensate for deleting a state by adding extra arrows or labels, so that anything that would have previously gone through that 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.

<sup>32</sup> 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.



2. Let  $\ell$  be the label on any loop from  $q$  to itself. If there is no loop, then we take  $\ell$  to be the empty word  $\epsilon$ .
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  $\ell_x$  and  $\ell_y$  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  $x \rightarrow q \rightarrow y$  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  $x \rightarrow y$  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  $x \rightarrow y$  with label  $\ell_x \ell^* \ell_y$ .
5. Once this has been done for every configuration  $x \rightarrow q \rightarrow y$  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.<sup>33</sup>

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

<sup>33</sup> A detailed proof of this is outside the scope of these notes, but here is a proof sketch. If you fix an alphabet  $\Sigma$ , then the cardinality of the set of all languages on  $\Sigma$  is the cardinality of the power set of  $\Sigma^*$ . The set  $\Sigma^*$  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<sup>34</sup> 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 accept state that  $w$  ends on! We can say the same thing about the words  $xz$ ,  $xyyz$ , 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.<sup>35</sup> Understanding this theorem is an exercise in getting your order of quantifiers correct!

**Theorem 64.** *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.  $|y| \geq 1$ .

<sup>34</sup> Be careful — there are non-regular languages that fool the pumping lemma. That is, it does not pick up all non-regular languages.

<sup>35</sup> 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.

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.  $\square$

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.

# ***TODO 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<sup>36</sup>.
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<sup>37</sup>.
6. There is complete information (the entire game state is known to both players at all times).
7. There are no chance moves.

<sup>36</sup> 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.

<sup>37</sup> This is called *normal play*. In the variant called *misère play*, the first player unable to make a move wins the game.

*Easy examples*

*Strategic and Grundy labelling*

*Nim*

## *Bibliography*