Lecture 13

Variants of Turing machines and encoding objects as strings

In this lecture we will continue to discuss the Turing machine model, beginning with some ways in which the model can be changed without affecting its power. We will also start discussing various *encoding schemes* that allow one to represent mathematical objects as strings of symbols, so that they can be viewed as inputs to Turing machines.

13.1 Variants of Turing machines

There is nothing sacred about the specific definition of DTMs that we covered in the previous lecture. In fact, if you look at two different books on the theory of computation, you're pretty likely to see two definitions of Turing machines that differ in one or more respects.

For example, the definition we discussed specifies that a Turing machine's tape is infinite in both directions, but sometimes people choose to define the model so that the tape is only infinite to the right. Naturally, if there is a left-most tape square on the tape, the definition must clearly specify how the Turing machine is to behave if it tries to move its tape head left from this point. Perhaps the Turing machine immediately rejects if its tape head tries to move off the left edge of the tape, or the tape head might remain on the left-most tape square in this situation.

Another example concerns tape head movements. Our definition states that the tape head must move left or right at every step, while some alternative Turing machine definitions allow the possibility for the tape head to remain stationary. It is also common that Turing machines with multiple tapes are considered, and we will indeed consider this Turing machine variant shortly.

DTMs allowing stationary tape heads

Let us begin with a very simple Turing machine variant already mentioned above, where the tape head is permitted to remain stationary on a given step if the DTM designer wishes. This is an extremely minor change to the Turing machine definition, but because it is our first example of a Turing machine variant we will go through it in detail (and perhaps in more detail than it actually deserves).

If the tape head of a DTM is allowed to remain stationary, we would naturally expect that instead of the transition function taking the form

$$\delta : Q\backslash\{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma \rightarrow Q \times \Gamma \times \{\leftarrow, \rightarrow\}, \tag{13.1}$$

it would instead take the form

$$\delta : Q\backslash\{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma \rightarrow Q \times \Gamma \times \{\leftarrow, \downarrow, \rightarrow\}, \tag{13.2}$$

where the arrow pointing down indicates that the tape head does not move. Specifically, if it is the case that $\delta(p, a) = (q, b, \downarrow)$, then whenever the machine is in the state p and its tape head is positioned over a square that contains the symbol a, it overwrites a with b, changes state to q, and *leaves the position of the tape head unchanged*.

For the sake of clarity let us give this new model a different name, to distinguish it from the ordinary DTM model we already defined in the previous lecture. In particular, we will define a *stationary-head-DTM* to be a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}), \tag{13.3}$$

where each part of this tuple is just like an ordinary DTM except that the transition function δ takes the form (13.2).

Now, if we wanted to give a formal definition of what it means for a stationary-head-DTM to accept or reject, we could of course do that. This would require that we extend the yields relation that we discussed last time for ordinary DTMs to account for the possibility that $\delta(p, a) = (q, b, \downarrow)$ for some choices of $p \in Q$ and $a \in \Gamma$. This is actually quite easy—we simply include the following third rule to the two rules that define the yields relation for ordinary DTMs:

3. For every choice of $p \in Q\backslash\{q_{\text{acc}}, q_{\text{rej}}\}$, $q \in Q$, and $a, b \in \Gamma$ satisfying

   $$\delta(p, a) = (q, b, \downarrow), \tag{13.4}$$

   the yields relation includes these pairs for all $u \in \Gamma^*\backslash\{\sqcup\}\Gamma^*$ and $v \in \Gamma^*\backslash\Gamma^*\{\sqcup\}$:

   $$u(p, a)v \vdash_M u(q, b)v \tag{13.5}$$

As suggested before, allowing the tape head to remain stationary doesn't actually change the computational power of the Turing machine model. The standard way to argue that this is so is through the technique of *simulation*. A standard DTM cannot leave its tape head stationary, so it cannot behave *precisely* like a stationary-head-DTM, but it is straightforward to *simulate* a stationary-head-DTM with an ordinary one—by simply moving the tape head to the left and back to the right (for instance), we can obtain the same outcome as we would have if the tape head had remained stationary. Naturally, this requires that we remember what state we are supposed to be in after moving left and back to the right, but it can be done without difficulty.

To be more precise, if

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}) \tag{13.6}$$

is a stationary-head-DTM, then we can simulate this machine with an ordinary DTM

$$K = (R, \Sigma, \Gamma, \eta, q_0, q_{\text{acc}}, q_{\text{rej}}) \tag{13.7}$$

as follows:

1. For each state $q \in Q$ of M, the state set R of K will include q, as well as a distinct copy of this state that we will denote q'. The intuitive meaning of the state q' is that it indicates that K needs to move its tape head one square to the right and enter the state q.
2. The transition function η of K is defined as

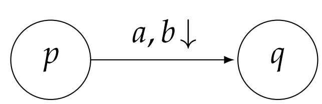
$$\eta(p,a) = \begin{cases} (q, b, \leftarrow) & \text{if } \delta(p,a) = (q,b,\leftarrow) \\ (q, b, \rightarrow) & \text{if } \delta(p,a) = (q,b,\rightarrow) \\ (q', b, \leftarrow) & \text{if } \delta(p,a) = (q,b,\downarrow) \end{cases} \tag{13.8}$$

for each $p \in Q \backslash \{q_{\text{acc}}, q_{\text{rej}}\}$ and $a \in \Gamma$, as well as

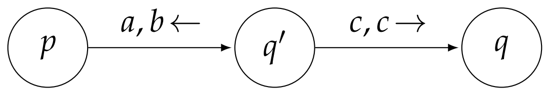
$$\eta(q', c) = (q, c, \rightarrow) \tag{13.9}$$

for each $q \in Q$ and $c \in \Gamma$.

Written in terms of state diagrams, one can describe this simulation as follows. Suppose that the state diagram of a stationary-head-DTM M contains a transition that looks like this:



The state diagram for K replaces this transition as follows:



(Here, the transition from q' to q is to be included for every tape symbol $c \in \Gamma$. The same state q' can safely be used for every stationary tape head transition into q.)

It is not hard to see that the computation of K will directly mimic the computation of M. The DTM K might take longer to run, because it sometimes requires two steps to simulate one step of M, but this does not concern us. The bottom line is that every language that is either recognized or decided by a stationary-head-DTM is also recognized or decided by an ordinary DTM.

The other direction is trivial: a stationary-head-DTM can easily simulate an ordinary DTM by simply not making use of its ability to leave the tape head stationary. Consequently, the two models are equivalent.

Thus, if you were to decide at some point that it would be more convenient to work with the stationary-head-DTM model, you could switch to this model—and by observing the equivalence we just proved, you would be able to conclude interesting facts concerning the original DTM model. In reality, however, the stationary-head-DTM model just discussed is not a significant enough departure from the ordinary DTM model to really be concerned with. We've gone through the equivalence in detail because it is our first example, but in comparison to some of the other variants of Turing machines we will discuss shortly, it's not worthy of too much thought. For this reason there will not likely be a need for us to refer specifically to the stationary-head-DTM model again.

DTMs with multi-track tapes

Another useful variant of the DTM model is one in which the tape has multiple tracks, as suggested by Figure 13.1. More specifically, we may suppose that the tape has k tracks for some positive integer k, and for each tape head position the tape has k separate tape squares that can each store a symbol. It is useful to allow the different tracks to store different symbols, so we may imagine k different tape alphabets $\Gamma_1, \ldots, \Gamma_k$, with Γ_j being the tape alphabet for track number j.

For example, based on the picture in Figure 13.1 it appears as though the first tape track of this DTM stores symbols from the tape alphabet $\Gamma_1 = \{0, 1, \sqcup\}$, the second track stores symbols from the tape alphabet $\Gamma_2 = \{\#, \sqcup\}$, and the third track stores symbols from the tape alphabet $\Gamma_3 = \{\clubsuit, \heartsuit, \diamondsuit, \spadesuit, \sqcup\}$.

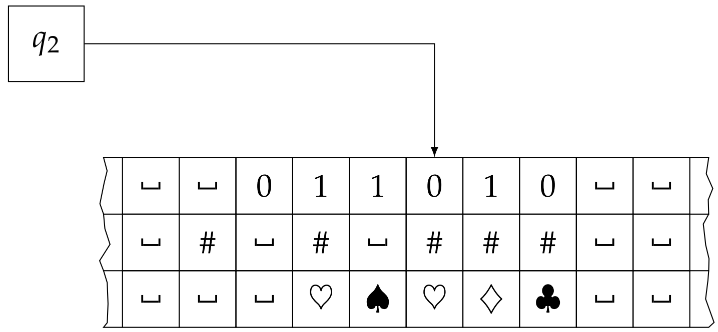


Figure 13.1: A DTM with a three-track tape.

When the tape head scans a particular location on the tape, it can effectively see and modify all of the symbols stored on the tape tracks for this tape head location simultaneously.

It turns out that this isn't really even a variant of the DTM definition at all—it's just an ordinary DTM whose tape alphabet Γ is equal to the Cartesian product

$$\Gamma = \Gamma_1 \times \cdots \times \Gamma_k. \tag{13.10}$$

Given that DTMs are supposed to have tape alphabets that contain all of the possible input symbols, meaning that $\Sigma \subset \Gamma$, and also contain a blank symbol, we should perhaps make clear that we identify each alphabet symbol $\sigma \in \Sigma$ with the tape symbol $(\sigma, \sqcup, \ldots, \sqcup)$, and also that we consider the symbol $(\sqcup, \ldots, \sqcup)$ to be the blank symbol of the multi-track DTM.

DTMs with one-way infinite tapes

DTMs with one-way infinite tapes were mentioned before as a common alternative to DTMs with two-way infinite tapes. Figure 13.2 illustrates such a DTM. Let us say that if the DTM ever tries to move its tape head left when it is on the leftmost tape square, its head simply remains on this square and the computation continues—maybe it makes an unpleasant crunching sound in this situation.

It is easy to simulate a DTM with a one-way infinite tape using an ordinary DTM (with a two-way infinite tape). For instance, we could drop a special symbol, such as ✂, on the two-way infinite tape at the beginning of the computation, to the left of the input. The DTM with the two-way infinite tape will exactly mimic the behavior of the one-way infinite tape, but if the tape head ever scans the special ✂ symbol during the computation, it moves one square right without changing state. This exactly mimics the behavior of the DTM with a one-way infinite tape that was suggested above.

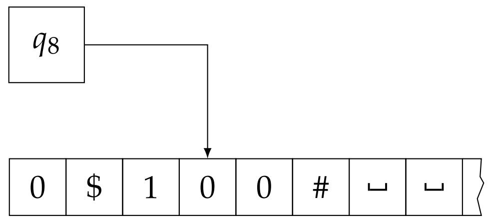


Figure 13.2: A DTM with a one-way infinite tape.

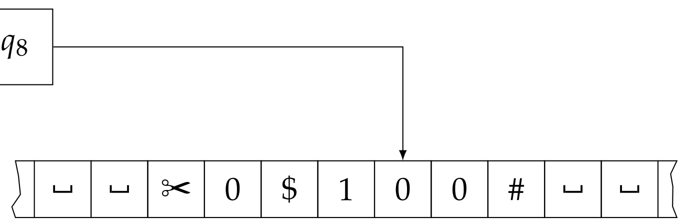


Figure 13.3: A DTM with a two-way infinite tape can easily simulate a DTM with a one-way infinite tape, like the one pictured in Figure 13.2, by writing a special symbol on the tape (in this case the symbol is ✂) that indicates where we should imagine the tape has been cut. When the tape head scans this symbol, the DTM adjusts its behavior accordingly.

Simulating an ordinary DTM having a two-way infinite tape with one having just a one-way infinite tape is slightly more challenging, but not difficult. Two natural ways to do it come to mind.

The first way is suggested by Figure 13.4. In essence, the one-way infinite, two-track tape of the DTM suggested by the figure may be viewed as the tape of the original DTM being simulated, but folded in half. The finite state control keeps track of the state of the DTM being simulated and which track of the tape stores the symbol being scanned. A special tape symbol, such as ➥, could be placed on the first square of the bottom track to assist in the simulation.

The second way to perform the simulation of a two-way infinite tape with a one-way infinite tape does not require two tracks, but will result in a simulation that is somewhat less efficient with respect to the number of steps required. A special symbol could be placed in the left-most square of the one-way infinite tape, and anytime this symbol is scanned the DTM can transition into a subroutine in which every other symbol on the tape is shifted one square to the right in order to "make room" for a new square to the left. This would presumably require that we also use a special symbol marking the right-most non-blank symbol on the tape, so

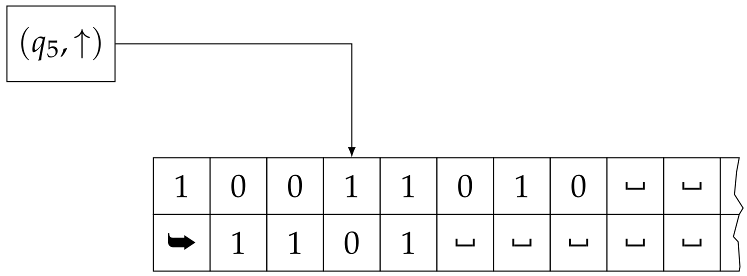


Figure 13.4: A DTM with a one-way infinite tape that simulates an ordinary DTM having a two-way infinite tape. The top track represents the portion of the two-way infinite tape that extends to the right and the bottom track represents the portion extending to the left.

that the shifting subroutine can be completed—for otherwise we might not know when every (non-blank) symbol on the tape had been shifted one square to the right.

Multi-tape DTMs

The last variant of the Turing machine model that we will consider will also be the most useful variant for our purposes. A *multi-tape DTM* works in a similar way to an ordinary, single-tape DTM, except that it has k tape heads that operate independently on k tapes, for some fixed positive integer k. For example, Figure 13.5 illustrates a multi-tap DTM with three tapes.

In general, a k-tape DTM is defined in a similar way to an ordinary DTM, except that the transition function has a slightly more complicated form. In particular, if the tape alphabets of a k-tape DTM are $\Gamma_1, \ldots, \Gamma_k$, then the transition function might take this form:

$$\delta : Q\backslash\{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma_1 \times \cdots \times \Gamma_k \rightarrow Q \times \Gamma_1 \times \cdots \times \Gamma_k \times \{\leftarrow, \downarrow, \rightarrow\}^k. \tag{13.11}$$

If we make the simplifying assumption that the same alphabet is used for each tape (which of course does not restrict the model, as we could always take this single tape alphabet to be the union $\Gamma = \Gamma_1 \cup \cdots \cup \Gamma_k$ of multiple alphabets), the transition function takes the form

$$\delta : Q\backslash\{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{\leftarrow, \downarrow, \rightarrow\}^k. \tag{13.12}$$

(In both of these cases it is evident that the tape heads are allowed to remain stationary. Naturally you could also consider a variant in which every one of the tape heads must move at each step, but we may as well allow for stationary tape heads

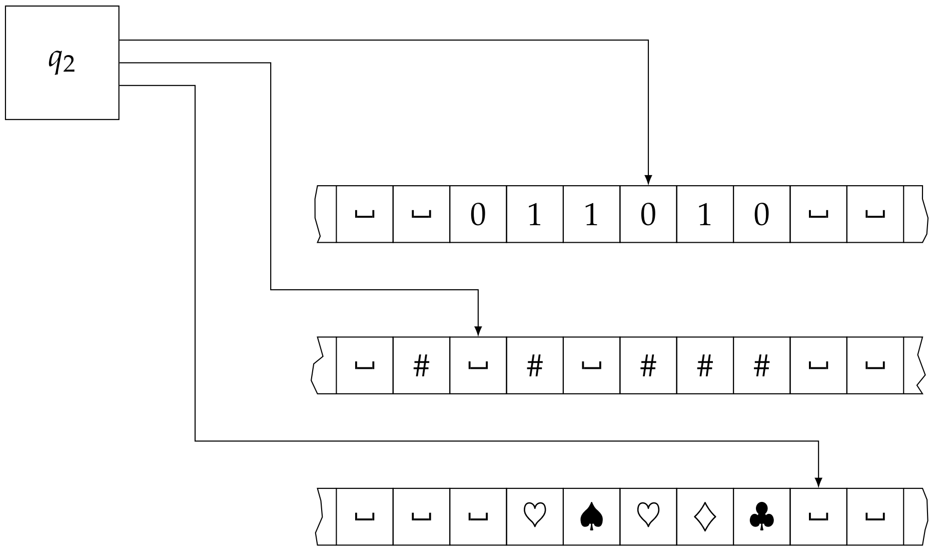


Figure 13.5: A DTM with three tapes.

when considering the multi-tape DTM model—it is meant to be flexible and general, so as to make it easier to perform complex computations.) The interpretation of the transition function taking the form (13.12) is as follows. If it holds that

$$\delta(p, a_1, \ldots, a_k) = (q, b_1, \ldots, b_k, h_1, \ldots, h_k), \tag{13.13}$$

then if the DTM is in the state p and is reading the symbols $a_1, \ldots, a_k$ on its k tapes, then (i) the new state becomes q, (ii) the symbols $b_1, \ldots, b_k$ are written onto the k tapes (overwriting the symbols $a_1, \ldots, a_k$), and (iii) the j-th tape head either moves or remains stationary depending on the value $h_j \in \{\leftarrow, \downarrow, \rightarrow\}$, for each $j = 1, \ldots, k$.

One way to simulate a multi-tape DTM with a single-tape DTM is to store the contents of the k tapes, as well as the positions of the k tape heads, on separate tracks of a single-tape DTM whose tape has multiple tracks. For example, the configuration of the multi-tape DTM pictured in Figure 13.5 could be represented by a single-tape DTM as suggested by Figure 13.6.

Naturally, a simulation of this sort will require many steps of the single-tape DTM to simulate a single step of the multi-tape DTM. Let us refer to the multi-tape DTM as K and the single-tape DTM as M. To simulate one step of K, the DTM M needs many steps: it must first scan through the tape in order to determine which symbols are being scanned by the k tape heads of K, and store these symbols within its finite state control. Once it knows these symbols, it can decide what action K is supposed to take, and then implement this action—which means again scanning through the tape in order to update the symbols stored on the tracks that represent

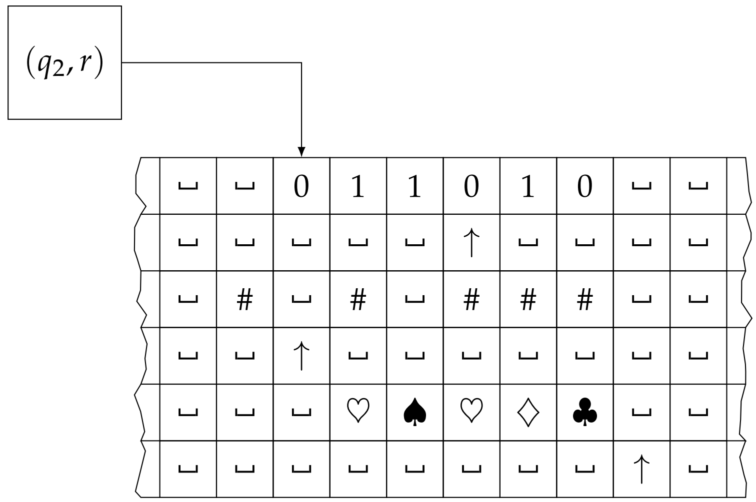


Figure 13.6: A single-tape DTM with a multi-track tape can simulate a multi-tape DTM. Here, the odd numbered tracks represent the contents of the tapes of the DTM illustrated in Figure 13.5, while the even numbered tracks store the locations of the tape heads of the multi-tape DTM. The finite state control of this single-tape DTM stores the state of the multi-tape DTM it simulates, but it would also need to store other information (represented by the component r in the picture) in order to carry out the simulation.

the tape contents of K and the positions of the tape heads, which may have to move. It would be complicated to write this all down carefully, and there are many specific ways in which this general idea could be carried out—but with enough time and motivation it would certainly be possible to give a formal definition for a single-tape DTM M that simulates a given multi-tape DTM K in this way.

13.2 Encoding various objects as strings

As we continue to discuss Turing machines and the languages they recognize or decide, it will be necessary for us to consider ways that interesting mathematical objects can be represented as strings. For example, we may wish to consider a DTM that takes as input a number, a graph, a DFA, a CFG, another DTM (maybe even a description of itself), or a list of such objects of multiple types.

In the remainder of this lecture we will begin this discussion with some simple examples, and the discussion will continue next lecture as we discuss the decidability of various languages connected with concepts we've seen previously in the course.

Encoding strings over one alphabet by strings over another

Suppose we have two alphabets: $\Sigma = \{0,1\}$ and $\Gamma = \{0,\ldots,n-1\}$, for n being some positive integer that could be very large. (Here, we are imagining that each integer between 0 and $n-1$ is a single symbol.)

There are a number of ways that we could encode symbols from the alphabet Γ by strings over the alphabet Σ, by selecting a different binary strings for each symbol in Γ. For example, if $n = 4$, we could choose to use the following simple encoding:

$$0 \rightarrow 00 \qquad 1 \rightarrow 01 \qquad 2 \rightarrow 10 \qquad 3 \rightarrow 11. \tag{13.14}$$

Or, we could choose this encoding:

$$0 \rightarrow 1 \qquad 1 \rightarrow 10 \qquad 2 \rightarrow 100 \qquad 3 \rightarrow 1000. \tag{13.15}$$

Now, if we have a *string* w over the alphabet Γ, we could aim to encode this string by simply concatenating together the strings over the alphabet Σ that encode the individual symbols of w. For example, with respect to the first example of an encoding above, the string $221302 \in \Gamma^*$ would be encoded as

$$221302 \rightarrow 101001110010, \tag{13.16}$$

and with respect to the second example of an encoding we would obtain

$$221302 \rightarrow 1001001010001100. \tag{13.17}$$

This will work so long as we've chosen the encodings of the individual symbols in Γ well.

When we say we've"chosen well," we mean that there won't be any ambiguity in the encodings of strings over the alphabet Γ obtained in this way. For instance, the encoding

$$0 \rightarrow 0 \qquad 1 \rightarrow 1 \qquad 2 \rightarrow 01 \qquad 3 \rightarrow 10 \tag{13.18}$$

is not chosen well: by concatenating these encodings together as suggested above, we would have (for example) that 01 and 2 have the same encoding 01. This is unacceptable, because we should be able to recover the original string from its encoding. If the encoding of each symbol of Γ has the same length, however, but different symbols are encoded by different strings, such as in the first example encoding above, we will never have this sort of ambiguity. More generally, if there is no encoding of a symbol in Γ that happens to be a prefix of an encoding of a different symbol in Γ, no ambiguity will arise.

There are good reasons to prefer encodings for which different symbols in Γ may have encodings of different lengths. For instance, the second example of an encoding suggested above can obviously be generalized for any alphabet $\Gamma = \{0, \ldots, n-1\}$, and it could be handy to use this encoding in a situation in which n is not known beforehand. Another example of a variable-length encoding is this one:

$$0 \to 0 \qquad 1 \to 10 \qquad 2 \to 110 \qquad 3 \to 111. \tag{13.19}$$

This encoding never causes any ambiguity of the sort described above, and it might result in shorter encodings in a situation where 0 is much more likely to appear than the other three symbols for some reason. (It is an example of a so-called *Huffman code*, which is useful for data compression.)

Encoding multiple strings into one

Next, suppose that we wish to represent *two or more* strings over some alphabet Σ by a single string over the same alphabet. Of course you could also consider encoding multiple strings over one alphabet by a single string over another, but it will be evident that this situation could be handled in a similar way.

This is easily done. One (of many) ways to encode multiple strings into a single string is to first introduce a new symbol that is not in our alphabet and use it as a marker to separate distinct strings. For example, if we have strings $x = \sigma_1 \cdots \sigma_n$ and $y = \tau_1 \cdots \tau_m$ over the alphabet $\Sigma = \{0, 1\}$, for instance, then perhaps we could introduce the symbol # to indicate a separation between the strings. The pair (x, y) could then be represented by the string

$$\sigma_1 \cdots \sigma_n \# \tau_1 \cdots \tau_m \tag{13.20}$$

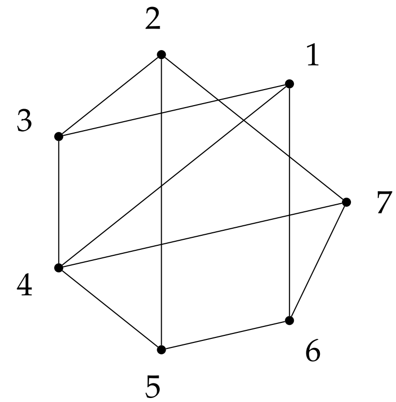
over the alphabet $\{0, 1, \#\}$. More generally, any tuple of strings $(x_1, \ldots, x_m)$ can be encoded as

$$x_1 \# x_2 \# \cdots \# x_m. \tag{13.21}$$

Once we have such a string over the alphabet $\{0, 1, \#\}$, we can then encode this string as a string over Σ as suggested in the previous subsection.

Numbers, vectors, and matrices

It probably goes without saying that natural numbers can be represented by binary strings using binary notation, as can arbitrary integers if we interpret the first bit of the encoding to be a sign bit. Rational numbers can be encoded as pairs of integers (representing the numerator and denominator), by first expressing the individual integers in binary, and then encoding the two strings into one as suggested above. One could also consider floating point representations, which are of course very



$$\begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Figure 13.7: A graph together with its adjacency matrix.

common in practice, but also have the disadvantage that they only represent rational numbers for which the denominator is a power of two.

Of course there are alternatives. For instance, we could use a *unary encoding*, where the nonnegative integer n is represented as 1^n (i.e., a string of length n consisting only of 1s), or you could invent new ways of representing numbers. Representing a nonnegative integer in unary notation may seem inefficient, but there are situations in which we do not care at all about efficiency, and the simplicity of this notation might be appealing for some reason.

With a method for encoding numbers in mind, one can then represent vectors by encoding the entries as strings, then encoding these multiple strings into a single string; and matrices can be encoded by combining the encodings of multiple vectors together. Alternatively, you might decide to encode a matrix by (for instance) introducing one special symbol that separate entries within a row and another special symbol that separates the columns.

Encoding graphs as binary strings

The last example of an encoding for today is for graphs. If we have a graph G with n nodes, one simple way to represent G as a binary string is by specifying the adjacency matrix. An example of a graph and its adjacency matrix appear in Figure 13.7. Because adjacency matrices contain only 0 and 1 entries, we could simply concatenate the entries of this matrix without doing anything further. For instance, the graph in Figure 13.7 would then be encoded by the binary string

$$0011010001010111010001010101010101010001010101010. \tag{13.22}$$

Alternatively, we could just concatenate the bits in the upper-triangular part of the adjacency matrix, as the diagonal entries are all 0 and the matrix is symmetric (for an undirected graph). For the same graph as above, this would yield the encoding

$$011010101011000101101. \tag{13.23}$$

Another alternative is not to use the adjacency matrix at all, but instead to encode a graph by describing a list of its edges, in some arbitrary order. Each edge could be represented as a pairs of integers, with the integers representing names for the vertices. For instance, the graph from Figure 13.7 has the following set of edges:

$$\begin{gathered}\{\{1,3\},\{1,4\},\{1,6\},\{2,3\},\{2,5\},\{2,7\},\\ \{3,4\},\{4,5\},\{4,7\},\{5,6\},\{6,7\}\}.\end{gathered} \tag{13.24}$$

Such a list of edges could be encoded as a string in a variety of straightforward ways.