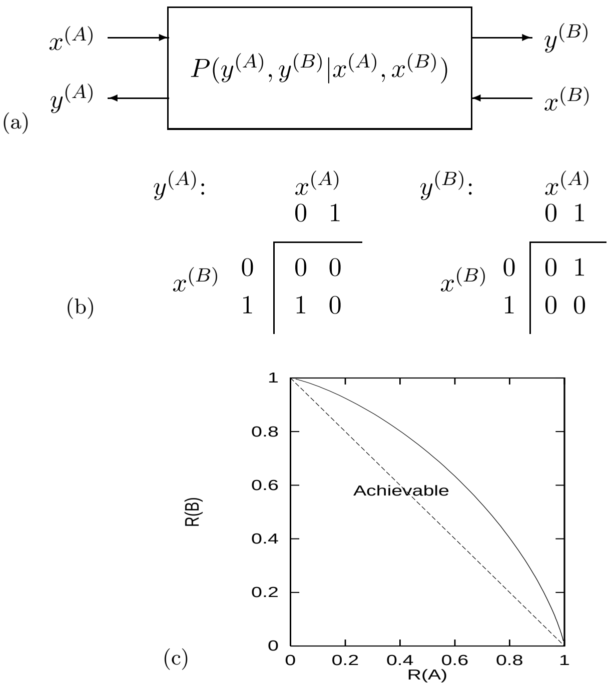


$y^{(A)}$:

		$x^{(A)}=0$	$x^{(A)}=1$
$x^{(B)}$	0	0	0
	1	1	0

$y^{(B)}$:

		$x^{(A)}=0$	$x^{(A)}=1$
$x^{(B)}$	0	0	1
	1	0	0

Figure 15.9. (a) A general two-way channel. (b) The rules for a binary two-way channel. The two tables show the outputs $y^{(A)}$ and $y^{(B)}$ that result for each state of the inputs. (c) Achievable region for the two-way binary channel. Rates below the solid line are achievable. The dotted line shows the 'obviously achievable' region which can be attained by simple time-sharing.

16

Message Passing

One of the themes of this book is the idea of doing complicated calculations using simple distributed hardware. It turns out that quite a few interesting problems can be solved by *message-passing* algorithms, in which simple messages are passed locally among simple processors whose operations lead, after some time, to the solution of a global problem.

▶ 16.1 Counting

As an example, consider a line of soldiers walking in the mist. The commander wishes to perform the complex calculation of counting the number of soldiers in the line. This problem could be solved in two ways.

First there is a solution that uses expensive hardware: the loud booming voices of the commander and his men. The commander could shout 'all soldiers report back to me within one minute!', then he could listen carefully as the men respond 'Molesworth here sir!', 'Fotherington–Thomas here sir!', and so on. This solution relies on several expensive pieces of hardware: there must be a reliable communication channel to and from every soldier; the commander must be able to listen to all the incoming messages – even when there are hundreds of soldiers – and must be able to count; and all the soldiers must be well-fed if they are to be able to shout back across the possibly-large distance separating them from the commander.

The second way of finding this global function, the number of soldiers, does not require global communication hardware, high IQ, or good food; we simply require that each soldier can communicate single integers with the two adjacent soldiers in the line, and that the soldiers are capable of adding one to a number. Each soldier follows these rules:

1. If you are the front soldier in the line, say the number 'one' to the soldier behind you.
2. If you are the rearmost soldier in the line, say the number 'one' to the soldier in front of you.
3. If a soldier ahead of or behind you says a number to you, add one to it, and say the new number to the soldier on the other side.

Algorithm 16.1. Message-passing rule-set A.

If the clever commander can not only add one to a number, but also add two numbers together, then he can find the global number of soldiers by simply adding together:

	the number said to him by the soldier in front of him,	(which equals the total number of soldiers in front)
+	the number said to the commander by the soldier behind him,	(which is the number behind)
+	one	(to count the commander himself).

This solution requires only local communication hardware and simple computations (storage and addition of integers).

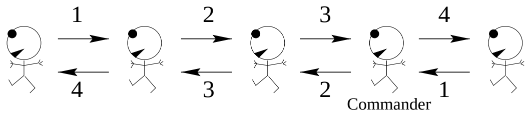


Figure 16.2. A line of soldiers counting themselves using message-passing rule-set A. The commander can add '3' from the soldier in front, '1' from the soldier behind, and '1' for himself, and deduce that there are 5 soldiers in total.

Separation

This clever trick makes use of a profound property of the total number of soldiers: that it can be written as the sum of the number of soldiers *in front of* a point and the number *behind* that point, two quantities which can be computed *separately*, because the two groups are separated by the commander.

If the soldiers were not arranged in a line but were travelling in a swarm, then it would not be easy to separate them into two groups in this way. The guerillas in figure 16.3 could not be counted using the above message-passing rule-set A, because, while the guerillas do have neighbours (shown by lines), it is not clear who is 'in front' and who is 'behind'; furthermore, since the *graph* of connections between the guerillas contains cycles, it is not possible for a guerilla in a cycle (such as 'Jim') to *separate* the group into two groups, 'those in front', and 'those behind'.

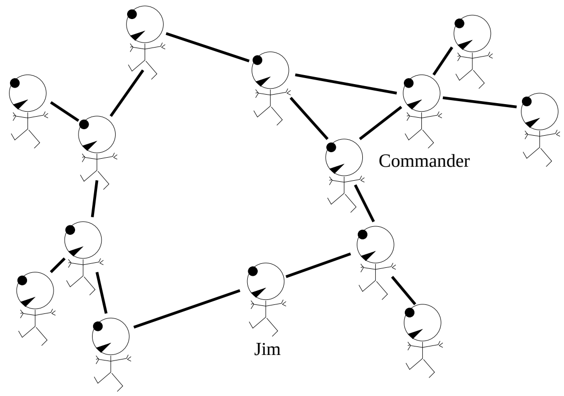


Figure 16.3. A swarm of guerillas.

A swarm of guerillas *can* be counted by a modified message-passing algorithm *if they are arranged in a graph that contains no cycles.*

Rule-set B is a message-passing algorithm for counting a swarm of guerillas whose connections form a *cycle-free graph*, also known as a *tree*, as illustrated in figure 16.4. Any guerilla can deduce the total in the tree from the messages that they receive.

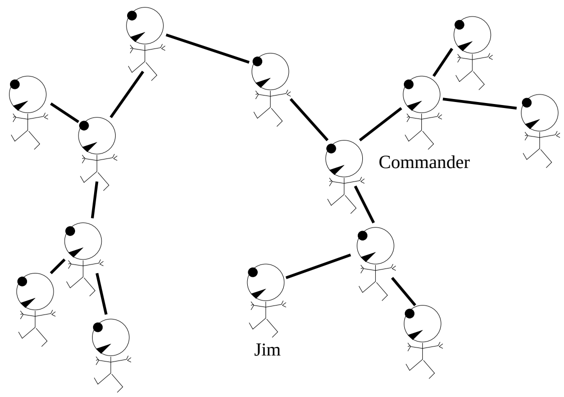


Figure 16.4. A swarm of guerillas whose connections form a tree.

1. Count your number of neighbours, N.
2. Keep count of the number of messages you have received from your neighbours, m, and of the values v_1, v_2, $\ldots, v_N$ of each of those messages. Let V be the running total of the messages you have received.
3. If the number of messages you have received, m, is equal to $N-1$, then identify the neighbour who has not sent you a message and tell them the number $V+1$.
4. If the number of messages you have received is equal to N, then:
   (a) the number $V+1$ is the required total.
   (b) for each neighbour n {
   say to neighbour n the number $V+1-v_n$.
   }

Algorithm 16.5. Message-passing rule-set B.

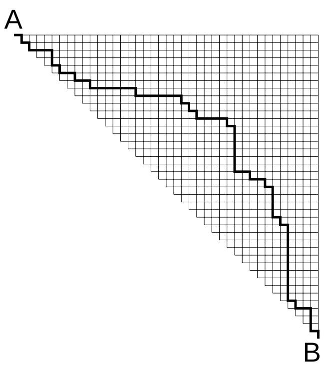


Figure 16.6. A triangular 41×41 grid. How many paths are there from A to B? One path is shown.

▶ 16.2 Path-counting

A more profound task than counting squaddies is the task of counting the number of paths through a grid, and finding how many paths pass through any given point in the grid.

Figure 16.6 shows a rectangular grid, and a path through the grid, connecting points A and B. A valid path is one that starts from A and proceeds to B by rightward and downward moves. Our questions are:

1. How many such paths are there from A to B?

2. If a random path from A to B is selected, what is the probability that it passes through a particular node in the grid? [When we say 'random', we mean that all *paths* have exactly the same probability of being selected.]

3. How can a random path from A to B be selected?

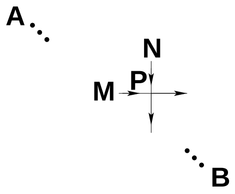


Figure 16.7. Every path from A to P enters P through an upstream neighbour of P, either M or N; so we can find the number of paths from A to P by adding the number of paths from A to M to the number from A to N.

Counting all the paths from A to B doesn't seem straightforward. The number of paths is expected to be pretty big – even if the permitted grid were a diagonal strip only three nodes wide, there would still be about $2^{N/2}$ possible paths.

The computational breakthrough is to realize that to find the *number* of paths, we do not have to enumerate all the paths explicitly. Pick a point P in the grid and consider the number of paths from A to P. Every path from A to P must come in to P through one of its upstream neighbours ('upstream' meaning above or to the left). So the number of paths from A to P can be found by adding up the number of paths from A to each of those neighbours.

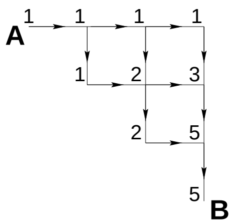


Figure 16.8. Messages sent in the forward pass.

This message-passing algorithm is illustrated in figure 16.8 for a simple grid with ten vertices connected by twelve directed edges. We start by sending the '1' message from A. When any node has received messages from all its upstream neighbours, it sends the *sum* of them on to its downstream neighbours. At B, the number 5 emerges: we have counted the number of paths from A to B without enumerating them all. As a sanity-check, figure 16.9 shows the five distinct paths from A to B.

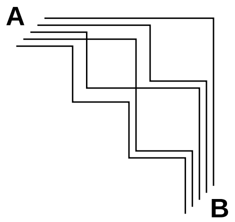


Figure 16.9. The five paths.

Having counted all paths, we can now move on to more challenging problems: computing the probability that a random path goes through a given vertex, and creating a random path.

Probability of passing through a node

By making a backward pass as well as the forward pass, we can deduce how many of the paths go through each node; and if we divide that by the total number of paths, we obtain the probability that a randomly selected path passes through that node. Figure 16.10 shows the backward-passing messages in the lower-right corners of the tables, and the original forward-passing messages in the upper-left corners. By multiplying these two numbers at a given vertex, we find the total number of paths passing through that vertex. For example, four paths pass through the central vertex.

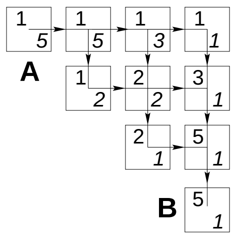


Figure 16.10. Messages sent in the forward and backward passes.

Figure 16.11 shows the result of this computation for the triangular 41 × 41 grid. The area of each blob is proportional to the probability of passing through the corresponding node.

Random path sampling

Exercise 16.1.[*1*, p.247] If one creates a 'random' path from A to B by flipping a fair coin at every junction where there is a choice of two directions, is

the resulting path a uniform random sample from the set of all paths? [Hint: imagine trying it for the grid of figure 16.8.]

There is a neat insight to be had here, and I'd like you to have the satisfaction of figuring it out.

Exercise 16.2.[*2*, p.247] Having run the forward and backward algorithms between points A and B on a grid, how can one draw one path from A to B *uniformly* at random? (Figure 16.11.)

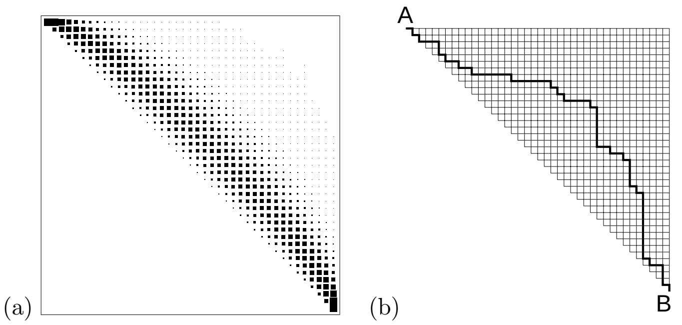


Figure 16.11. (a) The probability of passing through each node, and (b) a randomly chosen path.

The message-passing algorithm we used to count the paths to B is an example of the *sum–product algorithm*. The 'sum' takes place at each node when it adds together the messages coming from its predecessors; the 'product' was not mentioned, but you can think of the sum as a weighted sum in which all the summed terms happened to have weight 1.

16.3 Finding the lowest-cost path

Imagine you wish to travel as quickly as possible from Ambridge (A) to Bognor (B). The various possible routes are shown in figure 16.12, along with the cost in hours of traversing each edge in the graph. For example, the route A–I–L–N–B has a cost of 8 hours. We would like to find the lowest-cost path without explicitly evaluating the cost of all paths. We can do this efficiently by finding for each node what the cost of the lowest-cost path to that node from A is. These quantities can be computed by message-passing, starting from node A. The message-passing algorithm is called the *min–sum algorithm* or *Viterbi algorithm*.

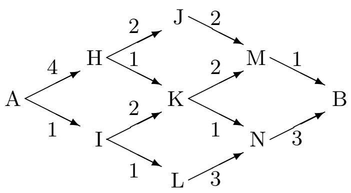


Figure 16.12. Route diagram from Ambridge to Bognor, showing the costs associated with the edges.

For brevity, we'll call the cost of the lowest-cost path from node A to node x 'the cost of x'. Each node can broadcast its cost to its descendants once it knows the costs of all its possible predecessors. Let's step through the algorithm by hand. The cost of A is zero. We pass this news on to H and I. As the message passes along each edge in the graph, the cost of that edge is *added*. We find the costs of H and I are 4 and 1 respectively (figure 16.13a). Similarly then, the costs of J and L are found to be 6 and 2 respectively, but what about K? Out of the edge H–K comes the message that a path of cost 5 exists from A to K via H; and from edge I–K we learn of an alternative path of cost 3 (figure 16.13b). The min–sum algorithm sets the cost of K equal to the minimum of these (the 'min'), and records which was the smallest-cost route into K by retaining only the edge I–K and pruning away the other edges leading to K (figure 16.13c). Figures 16.13d and e show the remaining two iterations of the algorithm which reveal that there is a path from A to B with cost 6. [If the min–sum algorithm encounters a tie, where the minimum-cost path to a node is achieved by more than one route to it, then the algorithm can pick any of those routes at random.]

We can recover this lowest-cost path by backtracking from B, following the trail of surviving edges back to A. We deduce that the lowest-cost path is A–I–K–M–B.

Other applications of the min–sum algorithm

Imagine that you manage the production of a product from raw materials via a large set of operations. You wish to identify the *critical path* in your process, that is, the subset of operations that are holding up production. If any operations on the critical path were carried out a little faster then the time to get from raw materials to product would be reduced.

The critical path of a set of operations can be found using the min–sum algorithm.

In Chapter 25 the min–sum algorithm will be used in the decoding of error-correcting codes.

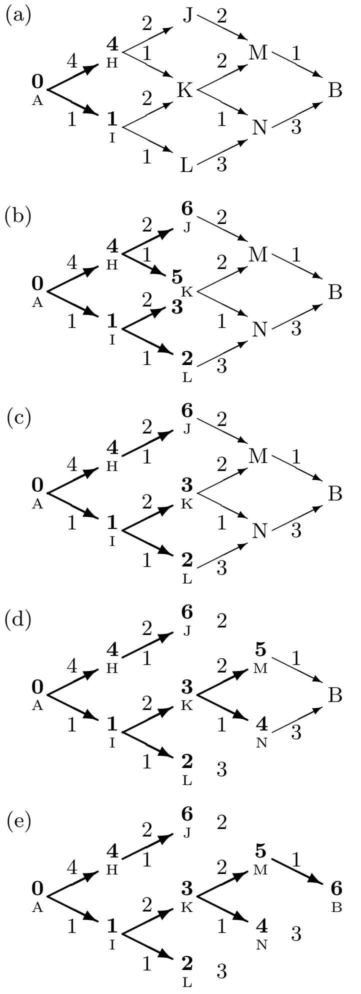


Figure 16.13. Min–sum message-passing algorithm to find the cost of getting to each node, and thence the lowest cost route from A to B.

▶ 16.4 Summary and related ideas

Some global functions have a separability property. For example, the number of paths from A to P separates into the sum of the number of paths from A to M (the point to P's left) and the number of paths from A to N (the point above P). Such functions can be computed efficiently by message-passing. Other functions do not have such separability properties, for example

1. the number of pairs of soldiers in a troop who share the same birthday;
2. the size of the largest group of soldiers who share a common height (rounded to the nearest centimetre);
3. the length of the shortest tour that a travelling salesman could take that visits every soldier in a troop.

One of the challenges of machine learning is to find low-cost solutions to problems like these. The problem of finding a large subset of variables that are approximately equal can be solved with a neural network approach (Hopfield and Brody, 2000; Hopfield and Brody, 2001). A neural approach to the travelling salesman problem will be discussed in section 42.9.

▶ 16.5 Further exercises

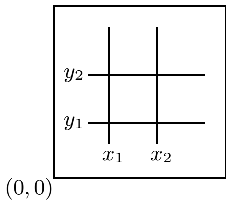
▷ Exercise 16.3.[2] Describe the asymptotic properties of the probabilities depicted in figure 16.11a, for a grid in a triangle of width and height N.

▷ Exercise 16.4.[2] In image processing, the *integral image* $I(x, y)$ obtained from an image $f(x, y)$ (where x and y are pixel coordinates) is defined by

$$I(x, y) \equiv \sum_{u=0}^{x} \sum_{v=0}^{y} f(u, v). \tag{16.1}$$

Show that the integral image $I(x, y)$ can be efficiently computed by message passing.

Show that, from the integral image, some simple functions of the image can be obtained. For example, give an expression for the sum of the image intensities $f(x, y)$ for all (x, y) in a rectangular region extending from (x_1, y_1) to (x_2, y_2).



▶ 16.6 Solutions

Solution to exercise 16.1 (p.244). Since there are five paths through the grid of figure 16.8, they must all have probability 1/5. But a strategy based on fair coin-flips will produce paths whose probabilities are powers of 1/2.

Solution to exercise 16.2 (p.245). To make a uniform random walk, each forward step of the walk should be chosen using a different biased coin at each junction, with the biases chosen in proportion to the *backward* messages emanating from the two options. For example, at the first choice after leaving A, there is a '3' message coming from the East, and a '2' coming from South, so one should go East with probability 3/5 and South with probability 2/5. This is how the path in figure 16.11b was generated.