Train Sets

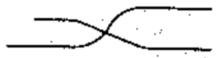
Adam Chalcraft and Michael Greene

Suppose we have a train set—large stocks of straight and curved track, bridge-building materials, different kinds of sets of points, and a single engine, say clockwork for the sake of nostalgia. What can we do?

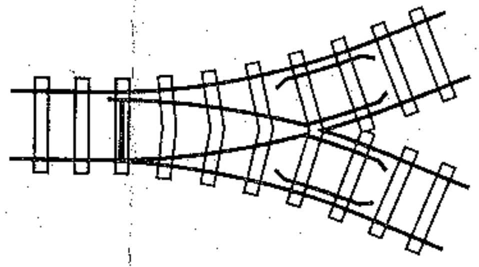
This depends, of course, on what kinds of point we have (for simplicity, a *set of points* or *pair of points* will be called a *point*). We use the following general notation:

Track will be drawn as a single line:

Bridges will be drawn simply as crossovers:

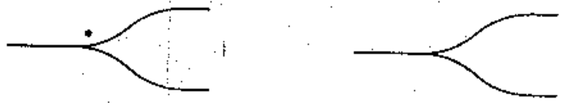


Lazy points



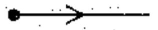
Lazy points will be taken to behave in the way you might expect from studying the physical piece of track above. With the point in the state shown, a train entering from F will leave by S_1, and vice versa, and a train from S_2 will change the point to its other state (with the central A-shaped piece up rather than down) and leave by the feeder. If the state has changed, the behaviour is exactly the same with S_1 and S_2 swapped.

When it is important which siding is *live*, i.e., which siding a train entering the feeder will leave by, a dot will be added to that siding; the other siding will be called *dead*. The lazy point above will therefore be represented by one of these:

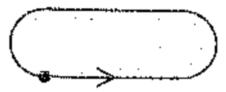


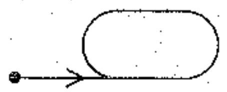
Suppose we have a finite *layout* (a connected arrangement of *sections* of track between feeders and sidings, with no ends, or 'buffers') with lazy points. Define the *state* of the system to be the setting of each of the lazy points, the section of track which the train T is on, and the direction in which T is travelling; then there are only finitely many states. We allow the train to run until the states cycle, and then remove any unused pieces of track from the layout and replace partially-used points by pieces of track. We would like to classify all possibilities for the *stabilised* layout which remains.

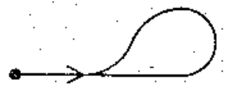
Follow the train from the middle of some section of track.



The layout is finite. Consider the first time T reaches track it has already traversed—then we have one of the following:

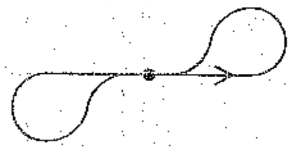

(a)

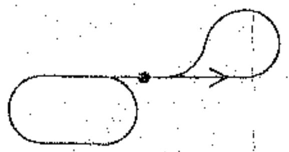

(b)

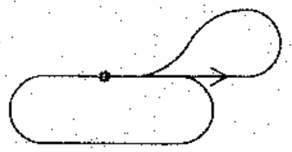

(c)

(Note that, although T may have crossed bridges on its route so far, these are the only essentially different connections.) Case (a) is a simple loop and is certainly a solution. In (b), T will continue around the loop and so had not settled down to a cycle of states; this case therefore does not occur.

In the last case, we need to follow the train round further. Again there are three essentially different cases:


(d)


(e)

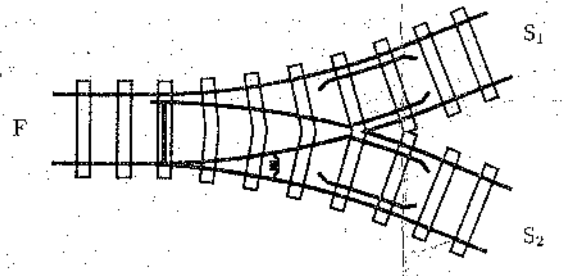

(f)

Both (e) and (f) fall into a smaller cycle and simplify to case (a). (d), however, is a solution, and is traversed in an unexpectedly complex way (and its behaviour shows that these are all the cases).

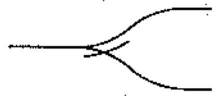
This classifies all finite stabilised lazy point layouts. □

We now introduce another natural kind of point.

Sprung points



Suppose we take a lazy point and attach a spring under tension as shown. Here, although a train from S$_2$ will push the central piece up as it runs over the point, the piece will not stay there, and any subsequent train from F will leave by S$_1$. This means that the live siding never changes. We call this a *sprung* point and use the notation



to represent it; the extra arc should be thought of as guiding a train from the feeder along the correct siding.

Suppose we have a finite layout with lazy and sprung points. As before, allow the train to run until the layout stabilises; then every point which was only going to change state finitely often will not change again.

We simplify the layout in three stages, without altering the route of the train.

Stage 1 We first simplify the network by removing any point and section of track which is unused, and replacing a point by track if one of its sidings is unused.

Now every section of track is either used infinitely often in both directions, or infinitely often in one direction and not at all in the other. The former sort of track is called *two-way*, and the latter *one-way*.

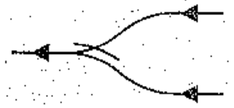
Stage 2 For the next stage, we regard sprung points as simpler than lazy points, and simplify the network by replacing every lazy point with a sprung point if this can be done without altering the route of the train.

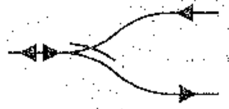
If a lazy point has a siding through which the train never leaves, then the point may be replaced by the sprung point which does not allow the train to leave through that siding.

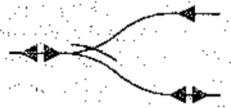
If a lazy point has a siding through which the train never enters, then the point will only be able to change state once. It will therefore not change state at all, and may as well be a sprung point.

Having done this, both sidings on every lazy point are two-way, and so therefore is the feeder.

Stage 3 Now consider the sprung points. The dead siding must be one-way. There are three remaining possibilities, shown in cases (g)–(i).


(g)


(h)

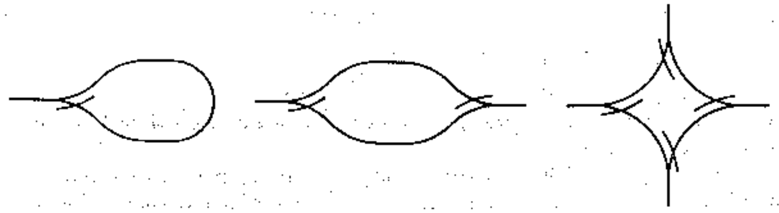

(i)

If we count the number of inputs and outputs in each of these cases, we find that (g) and (i) have one more input than output, whereas (h) has equal numbers of each. By Stage 2, all the lazy points have equal numbers of inputs and outputs. Across the entire layout, the number of inputs must equal the number of outputs, and so cases (g) and (i) cannot occur.

The sprung points must therefore all be as in case (h), and be joined to each other in circles, live siding joined to dead siding. These circles are called *roundabouts*, and the following figure (overleaf) shows roundabouts with 1, 2, and 4 sprung points. A train entering a roundabout must turn left, and then leave by the next exit.

The layout now consists of roundabouts and lazy points. Every section of track is two-way except the track inside a roundabout. This completes the classification of stabilised layouts made from lazy and sprung points. □

There is one more observation to be made, however. Suppose Stage 1 of the simplification has just been done, and we are about to embark on Stage 2. The only thing that



Stage 2 does is to turn some of the lazy points into sprung points. When this is over, all the sprung points look like case (h). Therefore any lazy point which was turned into a sprung point in Stage 2 had a one-way and two-way diagram as in case (h), but this is impossible for a lazy point.

The conclusion is that Stage 2 did not do anything. After Stage 1 was complete, all the lazy points were already two-way on both sidings and the sprung points were all arranged into roundabouts.

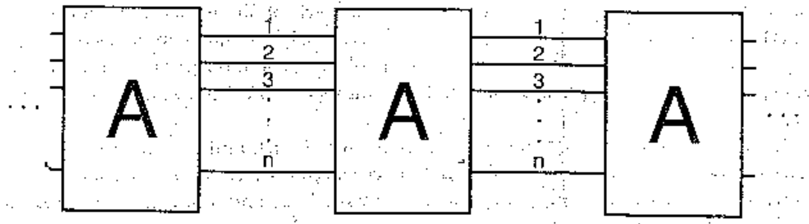
Can we build a computer?

We would like to know how complicated the route of the train can get. At one extreme, we might be able to simulate a general computer. The usual formulation of this is a *Turing machine*, which works as follows. Imagine a box of hardware which is able to move along an infinite 1-dimensional tape on which is written an infinite discrete string of 0s and 1s. The box can be in any of a finite number of *states* at any point on the track. It has a list of instructions as to what to do for any state looking at either digit. Each of these instructions is of the form

> leave unaltered or change the current digit,
> move one place to the right or left along the tape, and
> enter a particular state before performing the next instruction.

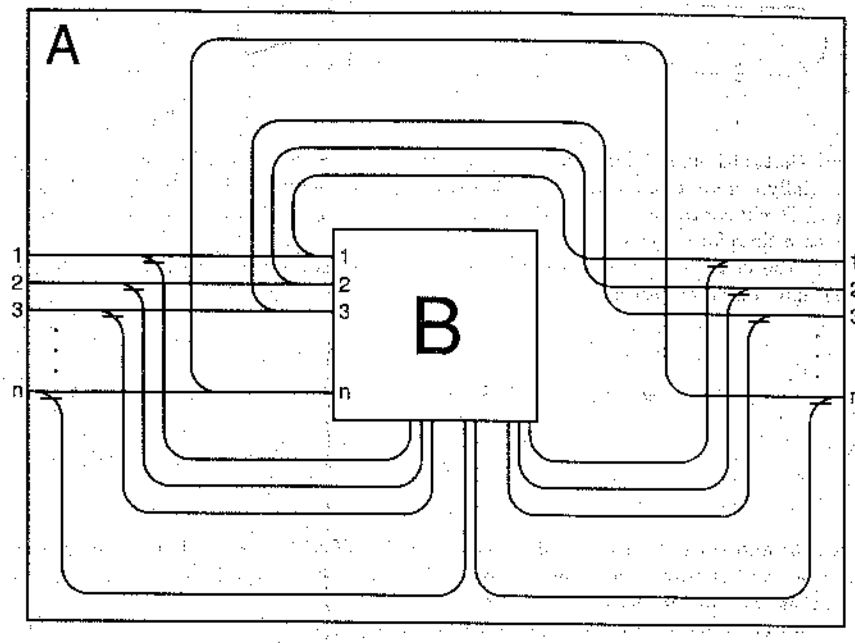
Now the process repeats. It is 'well-known' that any algorithm which can be run on an ordinary computer can be encoded to run on such a device.

In the search for such a model, we shall try to find a cell A which simulates the behaviour of the box at one point on the tape, and then plug these together to represent the whole tape:



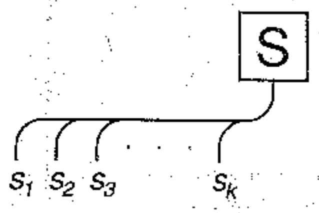
If T is in a particular copy of A, this will correspond to the box being at the equivalent position on the tape. If T travels from one copy of A to an adjacent copy along a piece of track labelled k, this will correspond to the box moving one step in the same direction along the tape, about to start the next action in state k. Note that this means there are two fundamentally different kinds of states which A can be in, representing either a 0 or a 1 at the corresponding point on the tape.

Now examine A. Around the outside, the tracks are used as both input and output, and the Turing machine does not remember which direction it came from to execute its next action. This track can therefore be used for the outer shell:



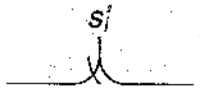
The B shown has n inputs on the left side and a selection of outputs coming out of the bottom. The outer shell sends the train out left or right on the appropriate line, as required. The sprung points ensure that the uses of the lines for output and input are split cleanly apart.

It will be useful to introduce the concept of a *subroutine*. This is a section of track with one input, which is also used as the output. When a train is sent in, it wanders around inside and eventually comes out again on the same section. We can give a subroutine more than one input:



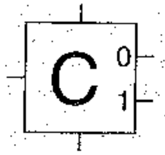
(*)

A train entering this diagram at s_i will set the lazy points to the route it took, enter and leave S, and then return along the same route—in particular it will return to s_i. We can *call* S by connecting s_i' (overleaf) to s_i:



A train travelling from left to right in this diagram will call S and continue.

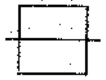
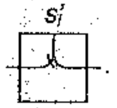
Suppose we have built a subroutine S, with entry points $s_1, s_2, \ldots, s_k$ as above, which changes the digit represented by the current copy of A from 0 to 1 or vice versa. Suppose further that we have a piece of layout which looks like



and behaves as follows:

i) a train from the left comes out on either 0 or 1 depending on the digit at the current point on the tape, and
ii) a train from the top changes the state recorded and comes out the bottom.

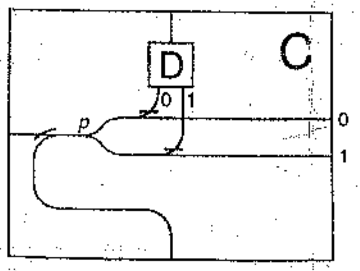
We now connect these up (see opposite).

Each is either a  or a .

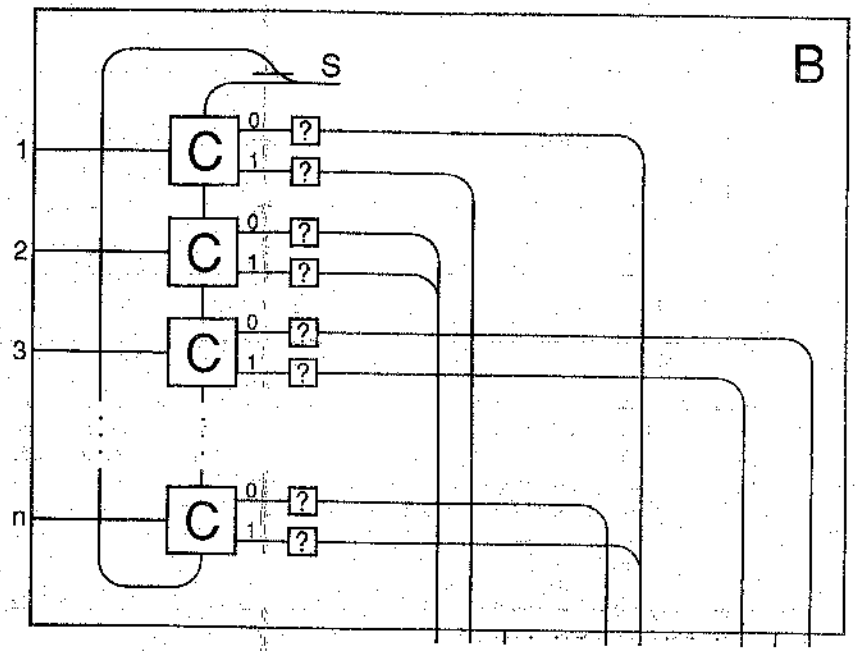
If T enters B on track k and passes through one of the Cs, we know exactly what to do to simulate the original Turing machine: we know whether to change the digit recorded in this position on the tape (that is, whether to call S), and we then know which way to leave this cell and on which track. Thus this diagram encodes the program. All we need to do is give S multiple entry points (as in (*)) and connect s_i to s_i' for each i, and the track will behave as wanted. (For neatness, we also should remove the unused track in A.)

Now to construct C. Let us suppose the existence of a *distributor* D, which has two outputs, 0 and 1, and one input. Trains sent in the input come out of 0 and 1 alternately. The reader is strongly advised to try constructing one of these—it makes an infuriating problem.

Suppose for the moment that such a piece of track exists. Take C to be this:



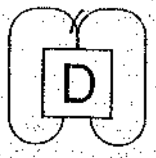
When we store our program's input data on the tape, we must set the (lazy) point p to



direct trains from the left out of the appropriate exit on the right, and make sure that the first train to enter by the top will come out of D on the correct side to flip p. From then on, C will behave as we want it to, and we nearly have a Turing machine—all we need is a distributor.

Distributors are impossible

Suppose D is a distributor made from a finite number of lazy points and sprung points, with one input and two outputs as shown in the first figure below. We can add one extra sprung point and some extra track to get the network shown in the second figure.

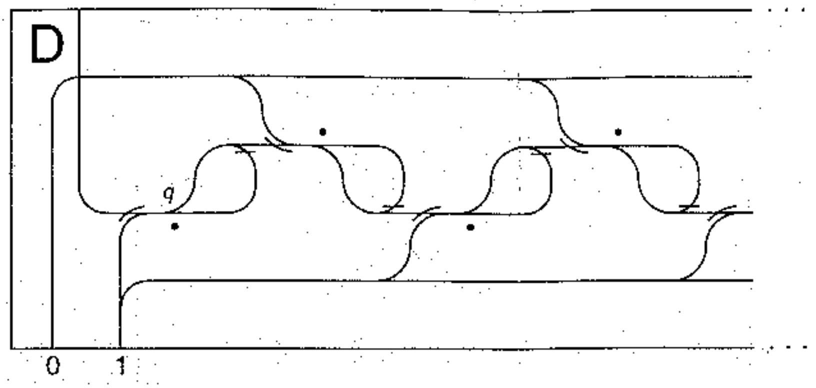


Now run this network until it stabilises. Since D will work as a distributor for ever, the stable form of D is still a distributor, so the network stabilises to a form in which the extra sprung point is being used like case (g) earlier.

Unfortunately, we showed there that case (g) could not happen, so distributors are impossible. □

A distributor

Now we shall exhibit a distributor. The construction is rather naive:

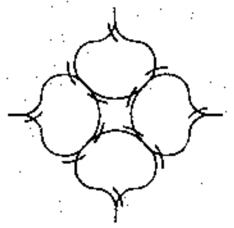


All lazy points are initially set to send trains across horizontally, except perhaps the one labelled q, which should be set the other way exactly when D is in a copy of A representing a 1 on the tape. Of course, D can be bent round to point upwards to avoid too many bridges or arbitrarily short sections of track.

This completes the construction of a Turing machine simulator. □

One further result

The reader may object to the impurity of using bridges. If we have sprung points, it is easy to simulate a bridge in the plane:



Conjectures

1. The Turing machine given here suffers from complexity problems: an algorithm which uses bits of the tape over and over again will get slower as more distant parts of the distributors are used (judging time by counting the number of points T crosses). We conjecture that no Turing-powerful layout has the same complexity for every algorithm as a conventional Turing machine.
2. Suppose we have a *random* point, which sends trains in a siding out of the feeder and trains in the feeder out of one of the sidings with equal probability, independent of previous behaviour. It seems likely that even with lazy, sprung, and random points we cannot build a finite distributor. (We now only require a box which always sends the train out of the correct exit if it sends it out at all, and does so in finite time with probability 1.)
3. A weaker version of 2: given lazy, sprung, and random points, we have been unable to construct a *partial* distributor, which never sends T out of the wrong exit, has for any n a positive probability of working at least n times in a row, but may have a chance of trapping the train inside after some time.