Computational Thinking: HexaHexaFlexagon Automata

Paul Curzon
Queen Mary University of London

Make a red and yellow hexahexaflexagon by folding and gluing a multicoloured paper strip, following the algorithm. Once made you start to explore it. As you fold it up and unfold it, you magically reveal new sides as the flexagon changes colour. To explore it fully, you need a map. A graph seems a good representation, which you create as you explore. It is like a tube map, with circles (nodes) for places revealed and lines between them (edges) showing which circles you can move between by folding and unfolding the flexagon. It is a special kind of graph that can be thought of as a machine - a 'finite state machine'. The nodes of the graph are different states the flexagon can be in and the edges show what actions that can be taken to move between states. It describes the computations involved in flexing the flexagon. A finite state machines is a very useful tools in the computational thinking toolbox. They are an important way for describing what computer systems do.

Hexahexaflexagon's are strange folded objects that have hidden sides. At any time only 2 sides are visible, but a hexahexaflexagon actually has six sides. They can appear in many different states each with a different centre.

Let's start by making one. Here is the algorithm:

Making a hexahexaflexagon

1. Print a coloured version of the sheet containing the two triangle strips (see Figure 1), enlarging them eg by printing them on to A3 paper. The larger they are the easier the flexagon will be to use, so alternatively make a large one out of individual coloured card triangles taped together.

2. Cut round the outside of both strips of triangles.

3. Glue them back-to-back down the full length of the strips:
   - The back of triangle FLAP 1 should be glued to the back of the end-most ORANGE triangle.
   - At the other end the back of triangle FLAP 2 should be stuck to the back of the end RED triangle.

4. Make creases, both backward and forward, across every edge of the strip so that the strip will fold easily everywhere.

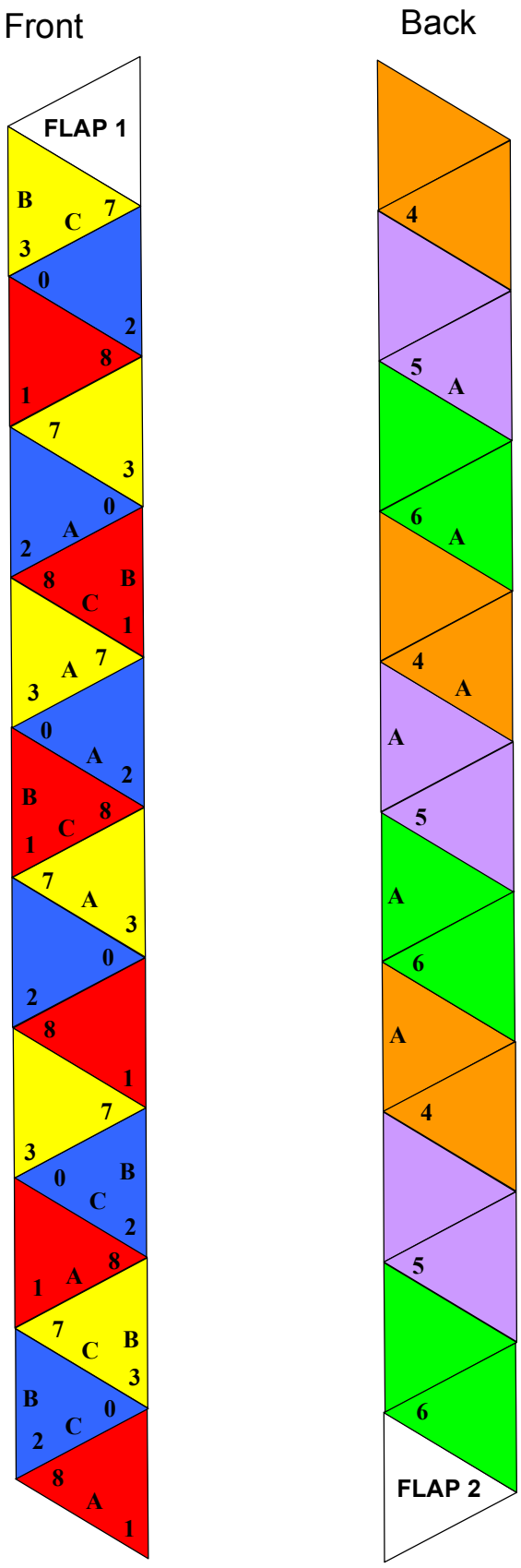

Figure 1. Print a coloured version of these strips. Cut out the strips and glue them back-to-back with the flaps at opposite ends to give a single strip.

5. Starting from the FLAP 1 end, fold the first purple triangle against its adjacent purple one, then green against green, orange against orange, and so on down the strip. You should end up with a shorter folded up strip that looks like the one in Figure 2:

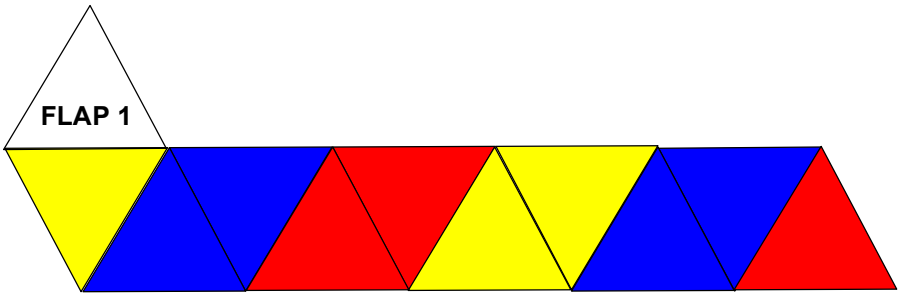

Figure 2. After the first round of folding you have a strip looking like this.

6. Do a similar thing with the new folded strip. Starting at the FLAP1 end, fold the first blue triangle against the adjacent blue triangle. This brings three yellow triangles together as in Figure 3

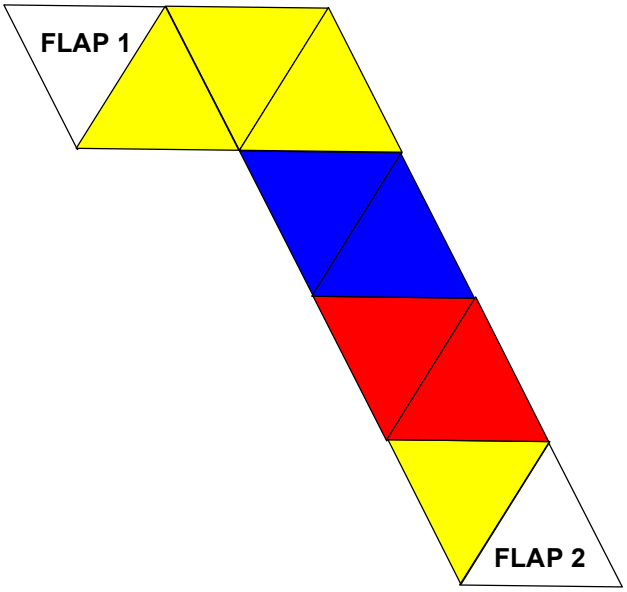

Figure 3. With another fold it starts to make a hexagon of yellow triangles.

7. Now fold the next two blue triangles together, back-to-back in the same way. You should now have five yellow triangles together in a hexagon shape with the next blue triangle.

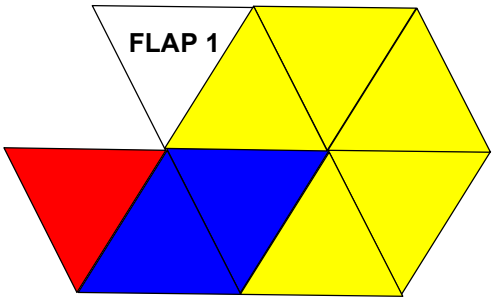

Figure 4. Continue folding the same colours back-to-back to fill out the yellow hexagon.

8. Fold the final two blue triangles together, tucking FLAP 2 in, so it is under the hexagon you have made, leaving a hexagon with FLAP 1 sticking out.

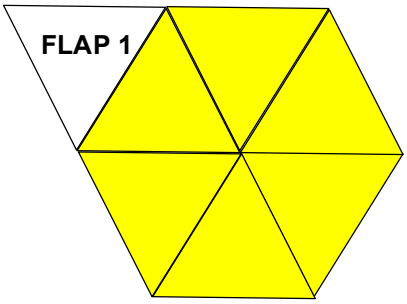

Figure 5. After one more fold, FLAP 2 is tucked under the hexagon.

9. Fold FLAP1 back and glue it to FLAP 2 so the faces with the words are now glued together. You should be left with a yellow hexagon of triangles on top. Turn it over and you should have a similar red hexagon of triangles.

You have now made a hexahexaflexagon…time to flex it.

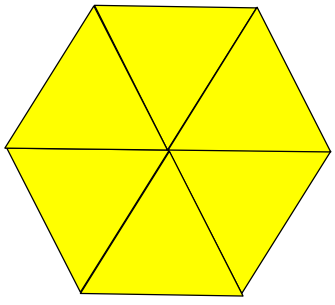

Figure 6. The final glued flexagon.

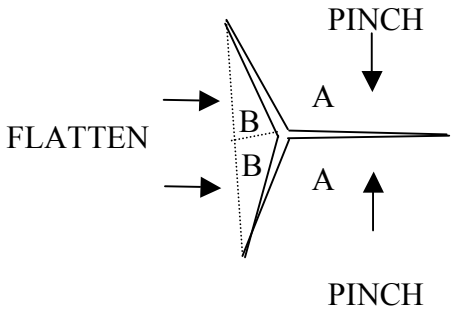

Figure 7. Flex a flexagon by pinching two sides together then flattening the opposite side, before opening it out from the middle.

Flexing your hexahexaflexagon

Your flexagon has a yellow side and a red side. You can reveal new, different coloured sides by folding it up and unfolding it again in a special way ('flexing' it) as described below (see Figure 7).

1. Start with the yellow side up. Two of the yellow triangles have a letter A on them (see Figure 8). Pinch those two triangles together.
2. Opposite them are two yellow triangles with B marked on them. While still pinching the As, flatten the B sides towards them so the flexagon makes a Y shape from above as in Figure 7.
3. Open the flexagon up from the middle of the Y, like a flower bud opening. A new blue side will be revealed. The yellow side has flipped to the back and the original red side has disappeared.
4. Keep doing this – pinching two sides together, flattening the opposite side and opening up from the middle and you will reveal more coloured sides or return to the original colours. Explore!

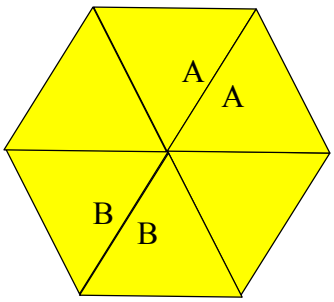

Figure 8. Flex by pinching either at edge between the two As (or between the two Bs), flattening the opposite side.

Exploring your hexahexaflexagon

There are six different coloured sides to find. The colours you reveal will depend on where you flex it – i.e., which sides you pinch together. On some sides it only works if you pinch in certain places. We have marked the flexagon with letters A, B and C at places to pinch that work – they are identified by folds where two identical letters come together.

If you don't turn the flexagon over, all the sides are marked by sets of 6 numbers round the centre of the hexagon. The original yellow side was marked 3. When you first folded it you ended up with a blue face marked 2 in the middle. As you explore you will find a second yellow 'side' with different numbers in the middle and similarly for other colours. All together there are 6 different colours, but 9 sets of numbers (0-8) that can appear in the middle, so 9 different states the top of the flexagon can appear in.

Mapping your hexahexaflexagon

If you've just explored randomly, it's likely you haven't found all of the flexagon's sides. Some are easier than others so you've probably been back to those first sides marked 1, 2 and 3 repeatedly. 7,8 and 0 are harder to find. Can you work out why?

The great explorers didn't just wander around new continents looking at rivers, waterfalls and mountains. They drew maps as they went. To explore the flexagon system more thoroughly, and to make sure you don't forget what you learn, you need a map too. A map is just an **abstraction** of the world. It is a simplified representation of something of interest like rivers and waterfalls, railways and stations or mountains and valleys. When choosing an abstraction you chose to include important features (roads say) while ignoring others (perhaps rivers).

The best representation to help us explore a flexagon is called a **graph** by computer scientists. It's a different thing to the graphs mathematicians draw. To a computer scientist a graph is just a series of circles, called **nodes**, and a series of lines connecting them called **edges** (see Figure 9). The nodes represent places to visit and the edges show how you can move between those places. An underground map is an example of a graph, where the stations are the nodes and the tube lines connecting them the edges. It ignores (abstracts away from) the actual positions and distances between stations to focus on how they are connected.

For a flexagon the places you can visit are the different numbered sides of the flexagon. We can write those numbers in the nodes to label them to show which are which (like naming stations on the underground map). For example, we now know that we can get from node 3 to node 2 by flexing at point A. We can show this with the following very simple 2-node graph.

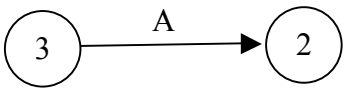

Figure 9. A simple graph showing one possible step on a flexagon.

We have used arrows for the edges of the graph to show the direction you can travel along them. This is called a **directed graph** as the edges now have directions.

Build up the graph – your map – as you explore the flexagon, drawing in new nodes when you find a new state (a new numbered side of the flexagon), and adding new edges each time you flex at a new place.

An algorithm for exploration

The map helps us explore as we can see where we have been, but if we explore at random then we still might not find all the sides, and we are likely to waste a lot of time going round in circles, a bit like wandering round a maze. Mazes are something that you can also represent as a graph, with nodes for junctions and edges the paths between them. Exploring a maze and our flexagon **generalise** to the same kind of problem once we abstract them using a graph as representation.

Perhaps you can come up with a way to explore the flexagon based on what you know about exploring mazes. Rather than get to a maze centre assume you want to map it all. You need a way that will make sure you do visit every part and don't waste time going round in circles – you need an algorithm.

One way to make sure you don't go round in circles in a maze is to mark each junction showing which way you go as you leave. Then, if you ever arrive back at the same point you choose a different direction. Now one of two things might happen. Eventually you will get to a junction where there are no alternatives left. You've been down them all. It might even be a dead end with no way out but the way you went in. What then? Well, if you know that you haven't left any path unexplored from a junction you've passed through, then you've explored all the maze. If you did leave some routes unexplored along the way, you can **backtrack**. You go back exactly the way you came until you get back to a junction that you haven't fully explored. Now explore that, and backtrack again when you run out of options again.

We could use that algorithm to explore the flexagon. It involves being able to undo a move made, though. That is possible but it's easier to always go forward with a flexagon. To make always going forwards work we can keep a list of nodes with unexplored edges, and then use the graph we've created to navigate forward to them from any node with no new options. That will work as long as there are no dead-end sections where the only way to return to the start is to go back the way you came. Is that true for the graph of a flexagon? If so you would need to either backtrack or dismantle the flexagon and re-glue it back to the start and carry on exploring from there – the equivalent of being teleported out of a maze back to the start to try again!

Try exploring the flexagon like this, always going forward, and see if you do ever get stuck.

Once you have drawn the graph, draw a tidied version of it, so the lines don't cross and the nodes are nicely spread out. (A solution graph is given at the end.)

Finite State Machines

Graphs like this, that are used to represent 'places' and how you can move between them, describe a machine that does computation. We call them **finite state machines** or **finite state automata**. The nodes of a finite state machine are **states**, and the edges **transitions**. The set of possible actions that are used to label the transitions are called the **alphabet** of the finite state machine. The alphabet for our flexagon is the set {A,B,C} as those are the labels used to mark flex points. A finite state machine also needs a specific place to start – the **initial state**. We will show it as a double circle. Our initial fragment of the finite state machine for the flexagon would then be drawn as follows:

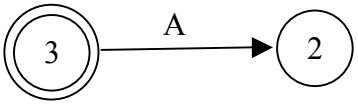

Figure 10. The start of the finite state machine for a flexagon, showing the starting state (double circled).

A finite state machine is like a program. It describes actions that can be taken and the computations done as a result. The machine starts in the start state. Then the machine takes transitions to new states when the corresponding actions happen as indicated by the labels on the arrows.

Finite state machines are powerful tools in the computational thinking toolkit. They are commonly used by designers to say precisely what a program, gadget or system is required to do early on in the design – to give a **specification** of what it should do mathematically. The programmers can then work from that description to ensure they program the right thing. Finite state machine descriptions can be given for just specific parts or the whole system.

They are particularly useful for specifying how you use a gadget: what sequences of buttons you must press (the possible actions) to do different things. They are often given in the user manuals of gadgets like digital watches and central heating systems.

A finite state machine version of a system is a **computational model** of it. Once we have created the finite state machine of our flexagon, for example, we can ignore the flexagon itself and explore it further using its finite state machine model. We can simulate the flexagon on a computer using the model, checking it does as expected. We can answer questions such as: what sequence of actions from the start state will get us to state 4, or how many steps does it take to get from state 5 to state 1? We can also check more general properties like whether there are any dead ends where we would get stuck, or whether we can always get back to the start from any state.

We already saw how important the last property is if we are trying to visit every state of the finite state machine. We can check these and other properties by simulating the computational model. We can also use special

mathematical algorithms that explore the graph in a really rigorous way (as we did to create it in the first place) that answer these kinds of questions automatically.

Once we have the complete graph of the finite state machine we can easily see why we kept returning to sides 1, 2 and 3, for example. Those states form a central triangle with the other states branching off but always returning. Whichever way you go, you always end up at one of those three sides within at most three flexes. Sides 7, 8 and 0 on the other hand are harder to get to. You have to go to the right state out of 1, 2 or 3.Then at that point choose the single correct branch, flexing twice before you get there.

By creating the finite state machine version – the computational model – we are able to better understand the flexagon.

From setting the date to life and death

Of course all of this doesn't just apply to a flexagon. It applies to anything we can represent with a finite state machine, and that includes gadgets. Suppose you have designed a digital watch with lots of different features like multiple timers, alarms, stop-watches, lap times and so on. It would be useful to be sure there are no modes the owner could get it into that there was no way out of. We wouldn't want to find that just because we decided to look at the date it was impossible to ever see the time again.

I just bought a digital watch. The sales assistant in the shop set the time but then couldn't work out how to set the date, even with the instruction manual. That in itself suggests that watch isn't well designed! Even when you have worked it out it turns out it is hard to remember how the next time you need to do it. We need some computational thinking! As I explore the watch pressing buttons to work out how to set the date, I draw the finite state machine. Once I have fully explored it I have a map. Keep that safe and any time in the future I need to reset the date (or do anything else), I can just follow my finite state machine map.

To take a more life-critical situation, suppose you were designing a machine for accident and emergency staff to use when patients arrive at the hospital – a resuscitator perhaps or a machine to quickly give them pain killers or lost blood. Sometimes the machine will be left in a random state at the end of an emergency. It would be good to know, when the next patient arrives, that whatever state it is in you can always get back to the start state quickly. Ideally this should involve doing exactly the same thing, whatever state the machine has been left in. It should be easily done with no thought and shouldn't take special training. A designer can check important properties like that at the outset based on the finite state machine model of the gadget. Similarly, regulators, those charged with making sure new machines are safe, could check that kind of property before they allow the machine to be sold by examining a finite state machine version.

Computational Thinking

Choosing a good **representation** of a problem makes a big difference to how easy it is to solve. We chose a graph representation for the flexagon. There were more decisions to make though – what should the nodes and edges be? We chose as nodes the different numbers in the centre of the sides and edges showing how we can move from one of these 'sides' to another. We could have chosen the six different colours as our nodes, but ultimately that wouldn't have worked. We then wouldn't have been able to tell the difference between several pairs of different states of the flexagon that have the same colours. Choosing the right representation for a problem matters.

Choosing a representation is actually all about **abstraction**. What about the flexagon matters and what doesn't for our problem at hand? The fact that it is made of paper? That it is made of triangles in the shape of hexagons? All that matters if our problem is to make a flexagon, but not if the problem is to understand what happens when we explore it. We can abstract the material and the shapes away. All that we need to worry about are the different states and the way we can move between them. Choosing the right abstraction for the states matters as we saw. Use the colours as our abstraction of states and we have lost too much information. What matters is not just which triangles are face up, but what is in the centre of the hexagon.

Making a flexagon and flexing one are algorithms. Writing instructions of how to make one for others involves **algorithmic thinking**. We could explore the flexagon at random, but to do it efficiently we need an algorithm and so algorithmic thinking there too. If we have worked out an algorithm for exploring a maze in the past, then we can use it to explore a flexagon too, if they are both represented as graphs. We are using **pattern matching** and **generalisation** to do that. We also saw how we can generalise the idea of a graph where the nodes are places visited in to a finite state machine. Any situation where we are exploring something by moving from place to place can be represented as a finite state machine. That holds whatever we mean by a 'place' (a flexagon's side, a junction in a maze, a tube station, a mode of a gadget, etc) or how to 'move' between them (flexing, walking, get on a train, pressing a button, etc).

In coming up with the algorithm we were doing **logical thinking** too: thinking logically through the steps of how to explore a maze so we visit it all, and working out how to overcome problems like not going backwards. We needed more logical thinking to decide how to draw the graph – what to use as nodes for example – and in thinking through whether properties hold of the graph.

Thinking of the graph as a finite state machine gives us a new set of tools to use. A finite state machine is a kind of **computational model**. We can simulate the actual system using it. We can then explore the flexagon without touching an actual flexagon, just using our finite state machine model. The same applies whether that system is a flexagon, a digital watch or even the London Underground. Once we have a computational model we can do more than just simulate it. We can check properties of it like finding the shortest

distances between two points, checking if there are any dead ends we will get stuck in and so on. Our flexagon is small enough that we can answer those questions just looking at the graph. For more complicated systems, like the autopilot of a plane we would need algorithms to check the properties automatically, visiting each state and checking the property held there for example. If we create a general tool for simulating or checking properties of finite state machines we can use it for anything we model as one.

All in all, there is a lot of computational thinking in a hexahexaflexagon!

More to do

Make a finite state machine of a digital watch
Interactive gadgets like digital watches, phones and so on all move through a series of hidden modes, or screens, in a similar way to our flexagon moving between states. Use the same approach to create a finite state machine map of a digital watch or other similar gadget. Each state will represent a mode (like the time mode, date mode, stopwatch mode, and so on). The possible actions – the alphabet – will be the buttons that can be pressed.

The other side of the flexagon!
So far we have completely ignored the other side of the flexagon! Turn it over when it is in the start state, state 3, and there are no numbers in the centre! There is a whole new world to explore by flexing it with that side up. Do any of the states of that side of the flexagon overlap with those we already know about? If they do how do we differentiate between them? Perhaps we will need a different abstraction once we start drawing the graph? If so what to use? Is it possible to pass through all the states without turning the flexagon over? Create a finite state machine for the flexagon as a whole.

More to read
There are many sources of information on Flexagons (and many more kinds of flexagon). One of the best is in Martin Gardner's, Mathematical Puzzles and Diversions, Penguin.

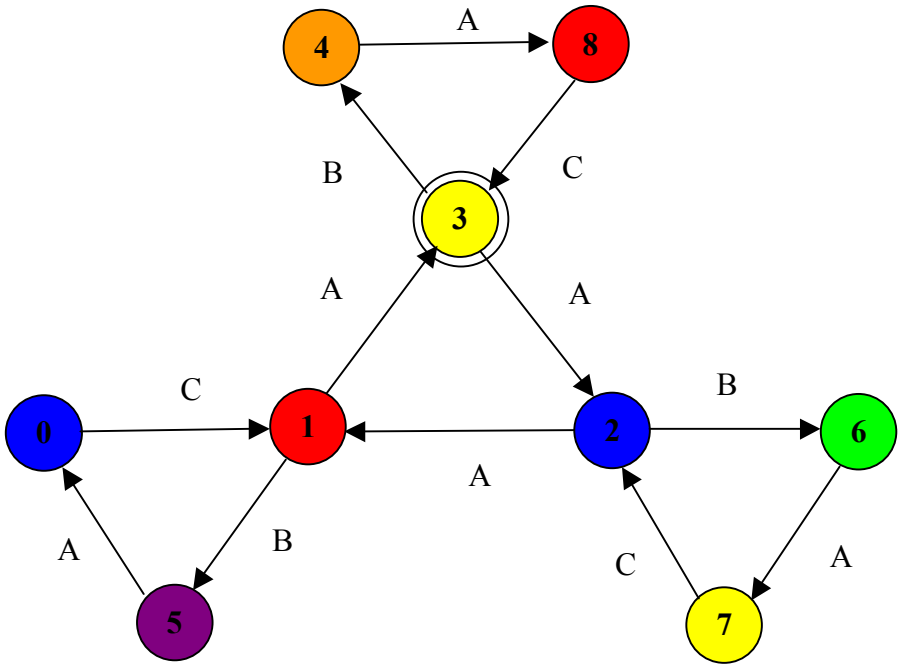

Figure 11. A solution state machine for the hexahexaflexagon

Use of this material

This booklet was written with support from CHI+MED funded by EPSRC (EP/G059063/1) and distributed with support from the Mayor of London and Department for Education.

chi+med
making medical devices safer

EPSRC
Engineering and Physical Sciences Research Council

Department for Education

SUPPORTED BY
MAYOR OF LONDON

COMPUTING AT SCHOOL
EDUCATE · ENGAGE · ENCOURAGE