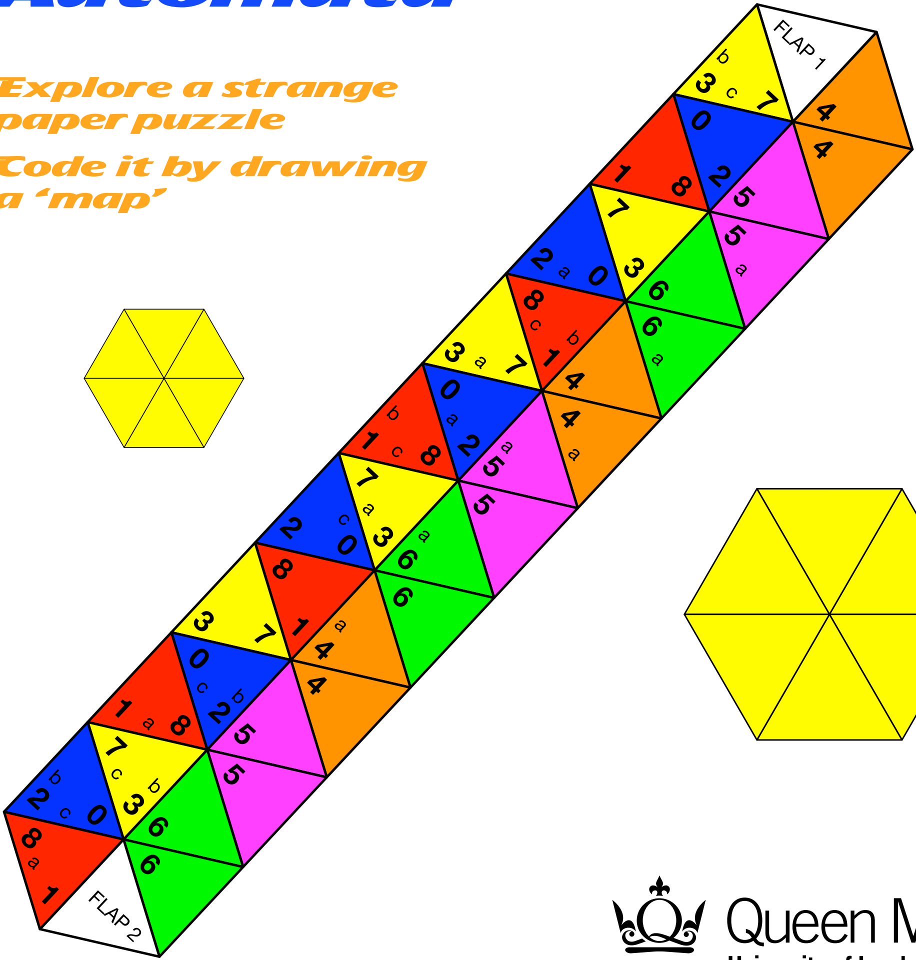
CS4FN

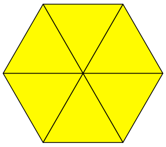
A Computer Science for Fun/Teaching London Computing/CHI+MED Special

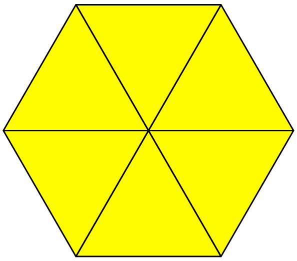
Computational Thinking: HexaHexaFlexagon Automata

Explore a strange paper puzzle

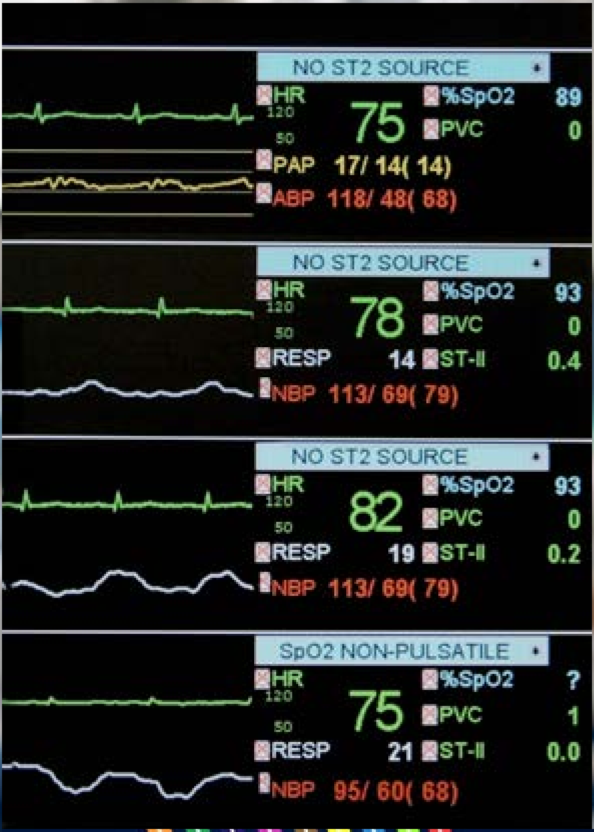
Code it by drawing a 'map'







Queen Mary
University of London


NO ST2 SOURCE
HR 120 50 75
%SpO2 89
PVC 0
PAP 17/ 14(14)
ABP 118/ 48(68)
NO ST2 SOURCE
HR 120 50 78
%SpO2 93
PVC 0
RESP 14
ST-II 0.4
NBP 113/ 69(79)
NO ST2 SOURCE
HR 120 50 82
%SpO2 93
PVC 0
RESP 19
ST-II 0.2
NBP 113/ 69(79)
SpO2 NON-PULSATILE
HR 120 50 75
%SpO2 ?
PVC 1
RESP 21
ST-II 0.0
NBP 95/ 60(68)

From paper puzzles to saving lives

You make a red and yellow hexahexaflexagon by folding and gluing a multicoloured paper strip in a special way. Once made you start to explore it. As you fold and unfold it, you magically reveal new sides as the flexagon changes colour.

You create a map as you explore it. It's like a tube map. Circles are places you find. Lines show the places you can move between by folding and unfolding the flexagon.

This special kind of map is called a finite state machine. Even though the map is just a sketch of circles and lines it is also a kind of program. It describes the computations involved in flexing the flexagon.

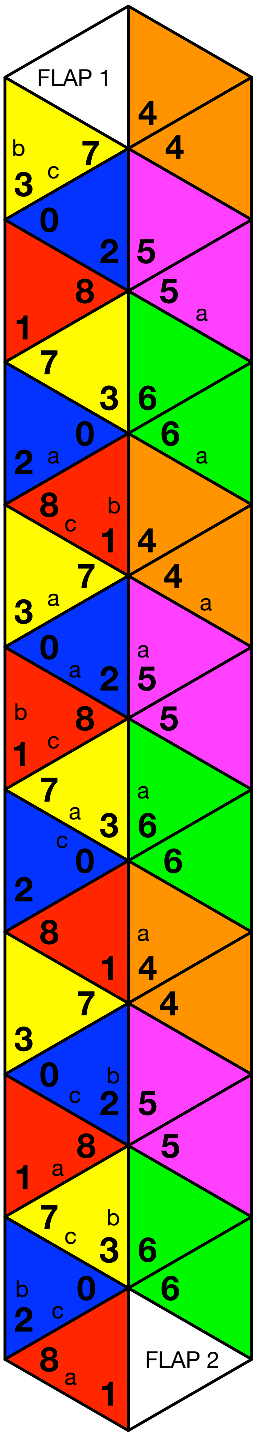
By drawing the map in a special finite state machine simulator, you have created a virtual version of the hexahexaflexagon. You can now explore it using the simulation you have created.

Using the same idea you can quickly create early prototypes of the user interfaces for any program you are developing, to check your ideas work. You can show it to the people who will use it. That way you can check they find it easy to use, don't make mistakes and never get confused about what to do next. It helps you make sure that your design works.

By quickly creating an early finite state machine version of a user interface you can also check other things about it, like whether you can always get back to the main home screen in one button press, or WON'T get stuck with no way out.

This really matters if people's lives depend on the program you are creating - if it is a heart monitor to keep people alive perhaps, or controls a roller coaster, or even if it is the shutdown system of a nuclear power plant.

Finite state machines are powerful tools in the computational thinking toolbox.



A hexahexaflexagon template

Making a Flexagon

Hexahexaflexagons

Hexahexaflexagons are strange folded objects that have hidden sides. At any time only 2 sides are visible, but a hexahexaflexagon actually has six sides. Some sides can appear in more than one state with a different centre.

Let's start by making one. Here is how to do it (or watch the video at www.cs4fn.org/hexahexaflexagon/).

Making a hexahexaflexagon: the algorithm

1. Download a copy of the coloured hexahexaflexagon template from www.cs4fn.org/hexahexaflexagon/

2. Print an enlarged, coloured version on to A3 paper.

3. Cut round the outside of the strip of triangles.

4. Fold carefully down the centre line.

5. Glue the two white sides back-to-back down the full length of the strip:

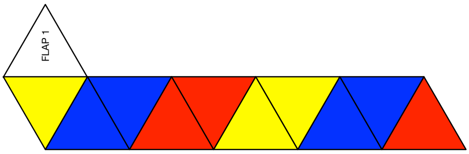
- The back of triangle FLAP 1 should be glued to the back of the end-most ORANGE triangle.

- At the other end the back of triangle FLAP 2 should be stuck to the back of the end RED triangle.

Download a copy of the multicoloured hexahexaflexagon template from www.cs4fn.org/hexahexaflexagon/

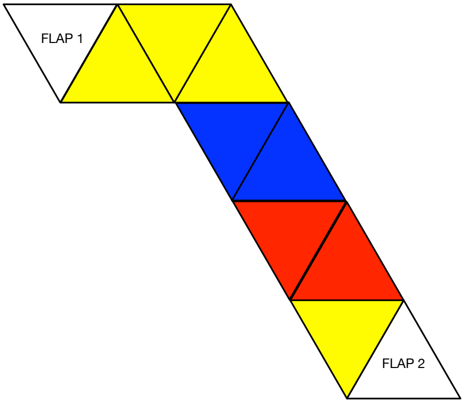
Making a Flexagon

6. 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. Make sure you fold as precisely as you can along the lines, so that triangles fold exactly over each other. You should end up with a shorter folded strip that looks like the one below.



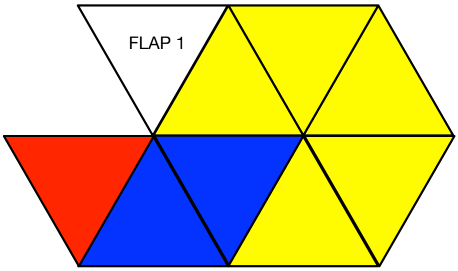
After the first round of folding you have a strip looking like this.

7. Do a similar thing with the result. Starting at the FLAP1 end, fold the first blue triangle against the adjacent blue triangle. This brings three yellow triangles together as below.



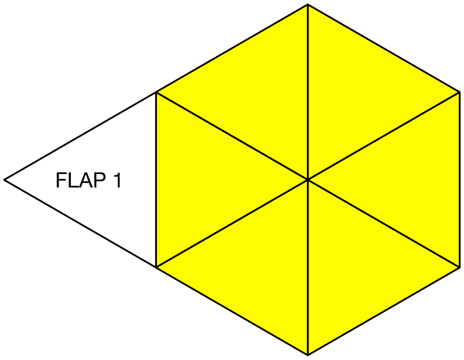
With another fold it starts to make a hexagon of yellow triangles.

8. 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.



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

9. 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.

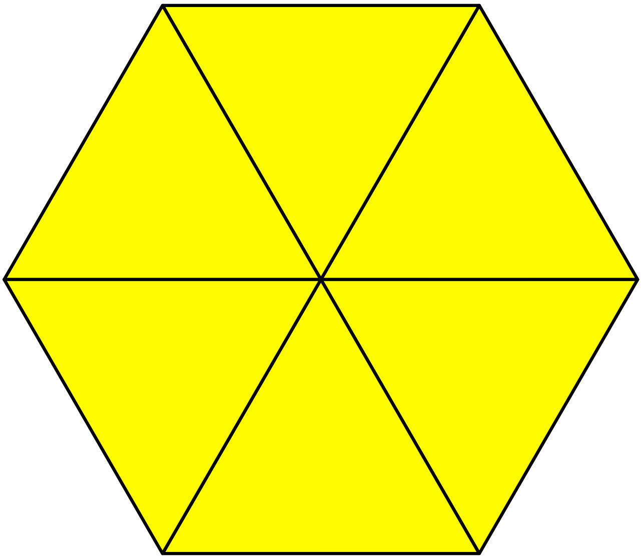


After one more fold, FLAP 2 is tucked under the hexagon.

Making a Flexagon

10. Fold FLAP1 back and glue it to FLAP 2 so the faces with these 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.

The final hexahexaflexagon is yellow on one side and red on the other.



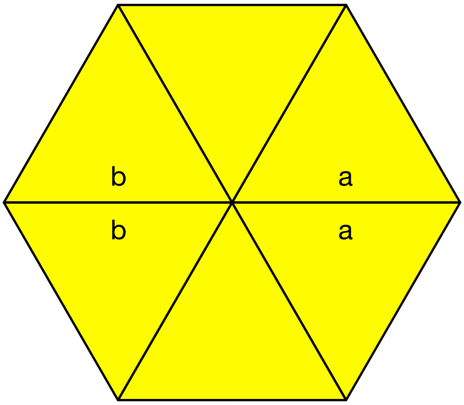
You have now made a hexahexaflexagon… time to flex it.

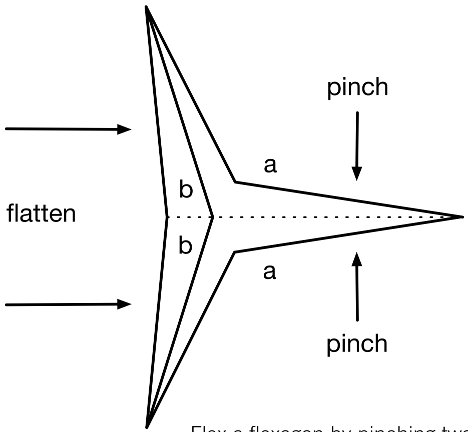
Flexing your hexahexaflexagon

Your flexagon has a yellow side and a red side. You can reveal new, different coloured sides by 'flexing' it: folding it up and unfolding it again. Here's how.

Flexing a hexahexaflexagon: the algorithm

1. Start with the yellow side up. Two of the yellow triangles have a letter 'a' on them. Pinch those two triangles together.
2. Opposite them are two yellow triangles with 'b' marked on them. While still pinching the 'a' s, flatten the 'b' sides towards them so the flexagon makes a Y shape from above as in diagram.
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. Explore!





Flex a flexagon by pinching two sides together between the 2 'a's, then flatten the opposite side, before opening it out from the middle.

Using Graphs to Explore

Exploring your hexahexaflexagon

There are six different coloured sides to find on the flexagon. The colours you reveal will depend on where you flex it – that is, which triangles you pinch together. On some coloured sides it only works if you pinch in some places not others. The flexagon is marked with letters a, b and c. Places to pinch that work have two identical letters together one either side of a fold.

If you started on the yellow side and 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 is marked 3. When you first folded it you ended up with a blue side marked 2. As you explore you will find a second yellow 'side' with different numbers in the middle. Other colours have two versions with different numbers too. 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.

A state is just a unique place you can be in a system. Each state is different in some way from any other state in that system.

Using Graphs to Explore

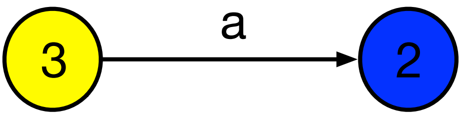
Mapping your hexahexaflexagon

If you've just explored randomly, you probably haven't found all the flexagon's sides. Some are easier to get to than others so you've probably been back to some sides a lot. Sides 7, 8 and 0 are harder to find. Can you work out why?

The great explorers didn't just wander aimlessly around new continents looking at rivers, forests and waterfalls. They drew maps as they went. To explore the flexagon thoroughly, and to make sure you don't forget what you discover, 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 decide to include important features (roads say) while ignoring others (perhaps rivers).

Let's draw a map as we explore the flexagon. What matters are the colours and numbers of the different sides, and how you can get from one to the next. We will abstract away everything else in our map.

Get a large piece of paper and draw a circle to stand for the current face up side of the flexagon. Perhaps you are on yellow side 3, so put a 3 in the circle (like naming stations on the underground map). Now flex the flexagon and see where you end up. Perhaps blue side 2. Draw another circle with a 2 in it nearby and then draw an arrow between them to show you can get from side 3 to side 2. Label the edges with the letter showing where you flexed it. We pinched the flexagon at **a** to get from side 3 to side 2 so we write that above the arrow. If we are ever at side 3 again we will know from our map what to do to get to side 2.



The start of a simple map of how to move around a flexagon

Graphs that are maps

As you flex the flexagon, moving from side to side, draw new circles for each new side. Put arrows between them showing how you got there. If you return to a side you've already seen, then the arrow should point back to the original circle. Don't draw a new one.

Computer scientists have a special name for this kind of map: a **graph**. 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**. The nodes represent places to visit and the edges show how you can move between those nodes. An underground or rail network map is a graph, where the stations are the nodes and the railway lines connecting them the edges. A railway map ignores (abstracts away from) the exact positions and distances between stations to focus on how they are connected.

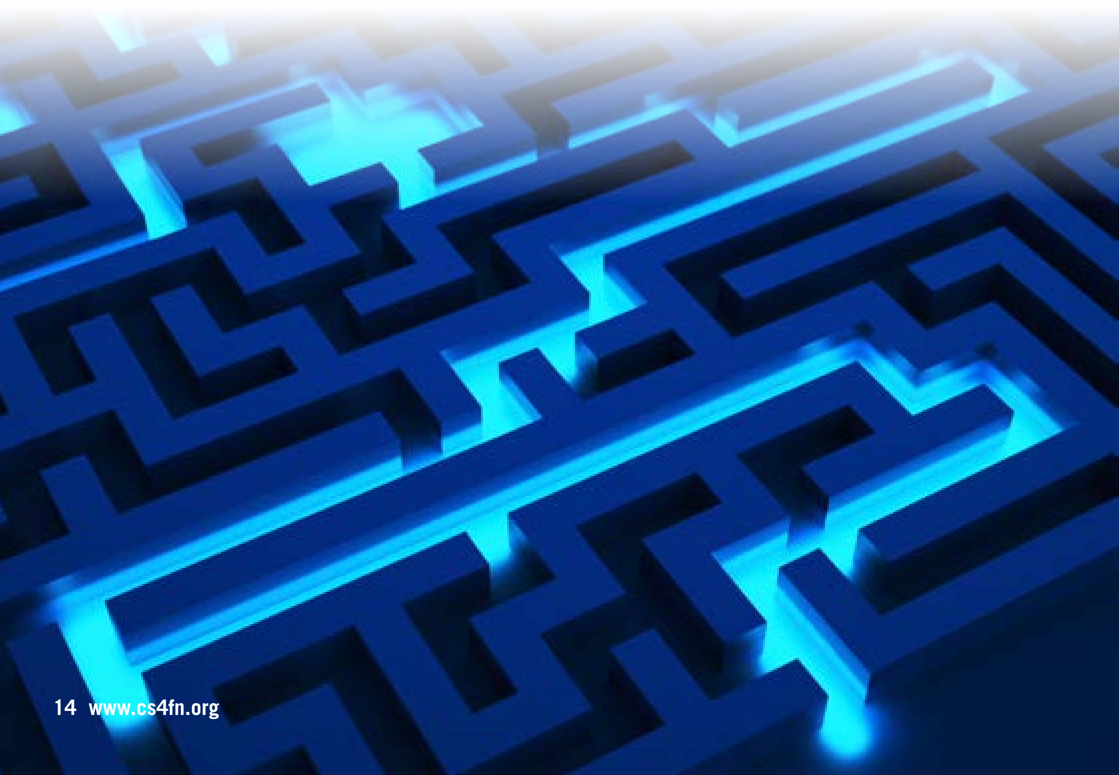
For a flexagon the places you can visit – the nodes - are the different numbered sides of the flexagon. The edges show how you can get from side to side. We've used arrows as edges of the graph to show the direction you can travel along them. This is called a **directed graph**.

Using Graphs to Explore

An algorithm for exploration

Our 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. We are also likely to waste a lot of time going round in circles, a bit like wandering round a maze. You can actually represent a maze as a graph too, 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 turn them in to graphs. We are creating an **abstraction** of them using a graph as **representation**.

So we need a way to make sure we do visit every part of the flexagon and don't waste time going round in circles – we need an algorithm to build the graph. At every node (or side of the flexagon), we need to see where each possible action takes us. That is, we need to explore all the edges from it to see where they lead.



Exploring with a 'To Do' list

We need a way to keep track of which parts of the graph are unexplored. We can do that by keeping a to-do list of nodes with unexplored edges. As we explore the flexagon, adding new nodes and edges to our graph, we can update our list.

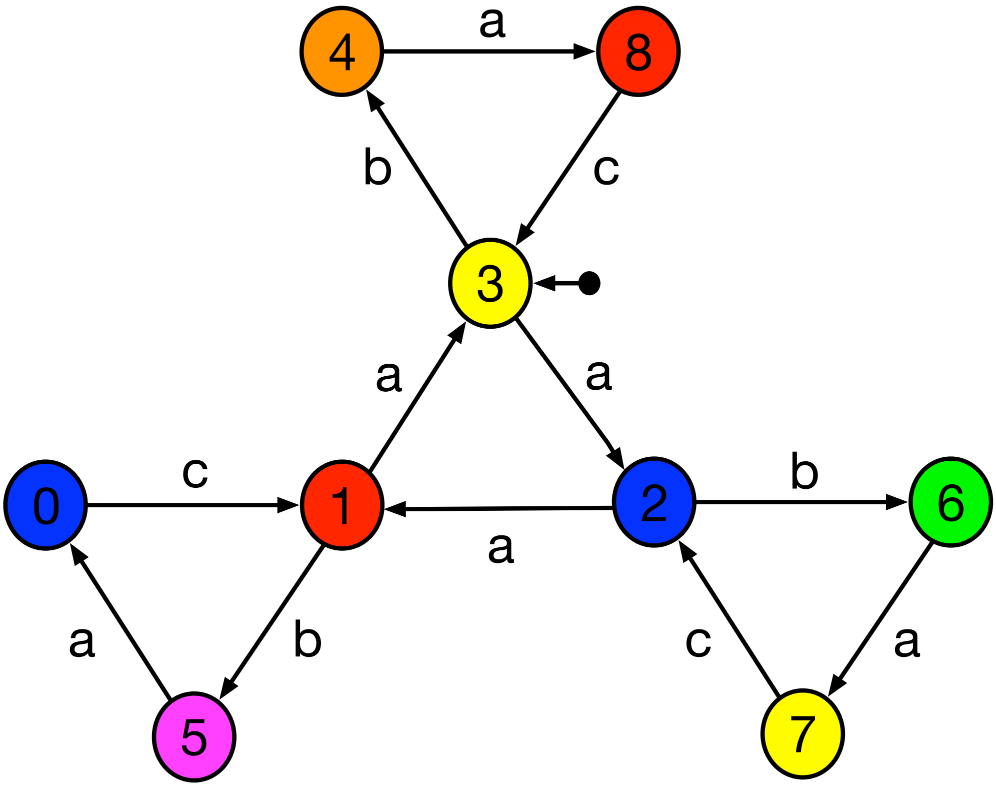
We start at any side of the flexagon, drawing it as the first node of our graph. It has as yet unexplored places to flex so we add that node to our to-do list too along with which actions (flexing at **a**, **b** or **c**) we have yet to explore. We pick any and try it, crossing it off the list for that node. We update the graph with the new edge. We are now at a new side of the flexagon, so we do the same from there adding each node to our list if it isn't already there.

Eventually we will get to a node where we have crossed off all the actions. That means it is fully explored. What then?

There are two possibilities. If our to-do list is empty then we have finished. We have fully explored the flexagon from that start point. The other possibility is that there is something still on our to-do list. We then need to use the graph we've created to navigate to one of those unexplored nodes and carry on.

This will work as long as there are no dead-end sections of the graph 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 flex the flexagon backwards to undo the moves made, or dismantle and re-glue it back to the start. That is the equivalent of being teleported out of a maze back to the entrance to try again!

Once you have drawn the full graph for the flexagon, draw a tidied version of it, so the lines don't cross and the nodes are nicely spread out. A full graph for the flexagon is given overleaf.



4
a
8
b
c
3
a
a
0
c
1
2
b
6
a
a
b
c
a
5
7

Finite State Machines

Drawing a program

Suppose we want to create a virtual version of a hexahexaflexagon. We want a program that displays an image of a flexagon's side. When we touch an edge it should switch to the correct new side as though it were a real flexagon. We could write a program to do this from scratch. Or we could just use the graph we already have as a program!

You can code just by drawing a graph.

Graphs are used to represent 'places' showing how you can move between them. This is a form of computation, and that means graphs can be turned in to a kind of program. We call it a **finite state machine**. They describe a machine that moves from **state** to **state**, where the states are just virtual versions of whatever the nodes represent. For our flexagon the states are the different numbered sides. The states are linked by **transitions**. They are the edges of the graph and tell you how you get from one state to another.

Each transition has a label that tells you what you have to do to take it – the **actions**. They act like the inputs to a program. The possible actions that label the transitions in the graph are called the **alphabet** of the finite state machine. The alphabet for our flexagon is the set of letters {**a**, **b**, **c**} as those are the labels used to mark flex points.

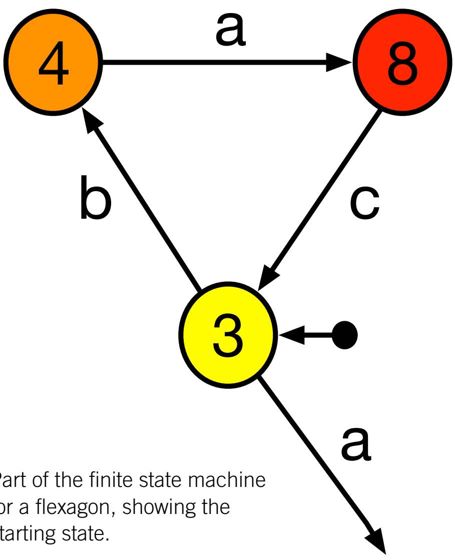
A finite state machine also needs a specific place to start – the **initial state**. We will use a black dot with an arrow to point to it.

Finally, each state of a finite state machine can **output** something. The output for our virtual flexagon could be a picture of the flexagon that is to be displayed when the flexagon is in that state. To keep things simple here, we will just assume the output is the current colour of the flexagon, together with its number. We will colour the states with the colour to be output.

Finite State Machines

An example finite state machine

The fragment of the finite state machine for our flexagon below says that if the machine is in state 3 (displaying a yellow side) and **b** is input then the flexagon will move to state 4 where it will display an orange side. From state 4, if an **a** is input, it will move to state 8, outputting red. If **c** is then input it will return to the original state. An **a** rather than a **b** from that state will take you to a so far unexplored state.



Part of the finite state machine for a flexagon, showing the starting state.

Quickly creating simulations

A finite state machine is a program because it describes actions that can be taken and the computations done as a result. The computations are just changes to the state.

We can write a one-off program that executes finite state machines, called a **finite state machine simulator**. Then if we give it the description of any finite state machine, we have a working simulation of whatever the finite state machine describes.

The simulator first sets the state to the start state. Then as keys are pressed it follows the transitions changing the state to the new state, showing the output on its display. The labels on the transitions correspond to the keys being pressed and show which transition to take.

If the simulator combines the finite state machine with an image of whatever it is describing we can make the simulation's output realistic. For example, we can link pictures of each side of the flexagon to the output of the finite state machine, displaying them as the output. We can also link parts of the image displayed to actions, so that, for example, when we touch the picture of a button the linked transition happens. With a flexagon, we can make touching a pinch point flex the flexagon there. We then have a working simulation of the flexagon.

The finite state machine simulator, pvsioweb, (**www.cs4fn.org/pvsioweb/**) is an example of a research toolset that does this. Created to help design medical devices, it allows finite state machines to be drawn like this and then quickly turns them in to working simulations. It allows much more, as it is linked to powerful mathematical tools too…

Checking Properties

Once we have created the finite state machine simulation of our flexagon, we can ignore the flexagon itself and explore it further using its model. We can do a similar thing with all sorts of phenomena we want to investigate – it is an alternative way to do science. Create a model of the phenomenon (like our finite state machine) then explore it. **Computational models** are a useful way to understand the world, from how our flexagon works to the way cancer spreads through cells.

Using the model, we can answer questions like: what is the quickest sequence of actions from the start state that 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. 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 executing the simulation. For a large machine that may take an impractically long time. A better way is to use some **logical thinking** to check properties of the model. We don't have to do this by hand. We can devise algorithms and write programs that check the properties we are interested in automatically. If we write them to work on finite state machine models and build them in to our finite state machine simulator, then they can be used to check these kind of properties automatically for any machine we create. This allows us to check the properties of the finite state machine in a really rigorous way (as we did to explore and create it in the first place).

For our flexagon, the finite state machine is small. So once we have the complete graph 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.

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

Graphs for real

Prototyping a persuasive game

Finite state machines are powerful tools in the computational thinking toolkit. Designers use them to mock up early versions of a program so they can check it will work how they expect - very much like the way film makers create a storyboard of a film early on.

Imagine you are designing a new 'persuasive game' - a game that helps make people aware of an issue you care about. The people who play it will learn what its like to be a child trying to escape a war torn country, perhaps. You want to get it right. The more people who play it the more who will learn about the problems as they work through the game's story, by making decisions. Each decision leads them to a new screen with a new situation to find themselves in.

Ideally you want a quick way to get a rough working version of the program to try it out and check your ideas, or show to people to get feedback.You don't want to put in lots of work, coding a slick production version, only to then find it has a fundamental flaw. You might sketch ideas for what the screens will look like at different points in the story. But you also need to plan out how the different decisions the players make will move you around the story.

How could you do this? One way would be to create a finite state machine of the game. This time the states are the different situations in the game. Decisions correspond to the transitions. The outputs are your sketches for the different screens. Simulating this finite state machine will quickly give you a working prototype of the game.

Finite state machines are also used as a way to say what a program, gadget, or even set of webpages should do at an early stage – to give a precise mathematical specification of what is required. The programmers creating the production version 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.

After giving the prototype of your persuasive game to different people you might realise some parts of the game don't work. After creating and trialling a whole series of prototypes, you eventually decide on a final version that works well. Now when you start to program that slick, fast version, you use the finite state machine as your guide as to what should happen in the real game.

Graphs for real

Manually set that Alarm

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 wasn't a mode 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.Using a finite state machine version we would be able to check.

Finite state machine descriptions are particularly useful for describing how you use a gadget: what sequences of buttons you must press (the actions) to do different things. This is exactly what you need in an instruction manual. Just as it helped us see how the flexagon works, one could help us understand how our new watch works. Finite state machine diagrams are often given in the instruction manuals of gadgets like digital watches and central heating systems, exactly for that reason. If the programmers used one when designing it, then the instruction manual could be automatically generated from that finite state machine, ensuring that it is right.

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 written instructions. That in itself suggests the watch isn't well designed and neither is the manual! Perhaps the programmers should have used a finite state machine.

With my new watch, even when you have worked out how to do things, it is hard to remember what to do the next time. 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. Not only can I now see why it is so difficult to use, if I keep my map safe then when I need to reset the date again (or do anything else), I can just follow my map.

Life and death

Of course all of this applies to any program we might want to create, and to anything we can represent with a finite state machine. That includes gadgets that people's lives depend on.

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 painkillers or lost blood. You want to be sure it is easy to use. You want the instruction manual to be correct. You want to be sure that certain properties hold. For example, often the machine will be left in a random state during 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 easy and obvious without special training. A designer can check important properties like that at the outset based on the finite state machine model of the device. Similarly, regulators, those charged with making sure new machines are safe, could check the same properties before they allow the machine to be sold in a similar way.

That is exactly how finite state machines are now being used…helping ensure machines are safe, saving lives.

Computational Thinking

Representations and abstraction

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 because it is about moving between 'places'. 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 that wouldn't have worked. We wouldn't have been able to tell the difference between 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**: hiding the right detail. Which features of the flexagon matter and which don'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? That matters if our problem is to make a flexagon, but not to 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. If we use the colours as our abstraction of states then 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.

Computational Thinking

Algorithmic and logical thinking

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 well we need another algorithm. We use algorithmic thinking there too. If we know an algorithm for exploring a maze, 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.

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, a web page, and so on) or how to 'move' between them (flexing, walking, get on a train, pressing a button, clicking a link, …). That is more generalisation.

In coming up with an algorithm we are doing **logical thinking**: thinking clearly through the steps of how to explore the flexagon so we visit it all. 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 true of the graph.

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.

General tools we write for doing things with finite state machines (simulating them, checking properties, and so on) will work for all these situations (generalisation at work again). 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, visiting each state and checking the property held there, automatically. 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.

Finite state machines also give a way to rapidly create early versions of programs so we can check them, trialling lots of ideas and generating the program before we commit to creating the final version. They can even then be used to generate instruction manuals that are guaranteed to exactly match the final program.

More to do

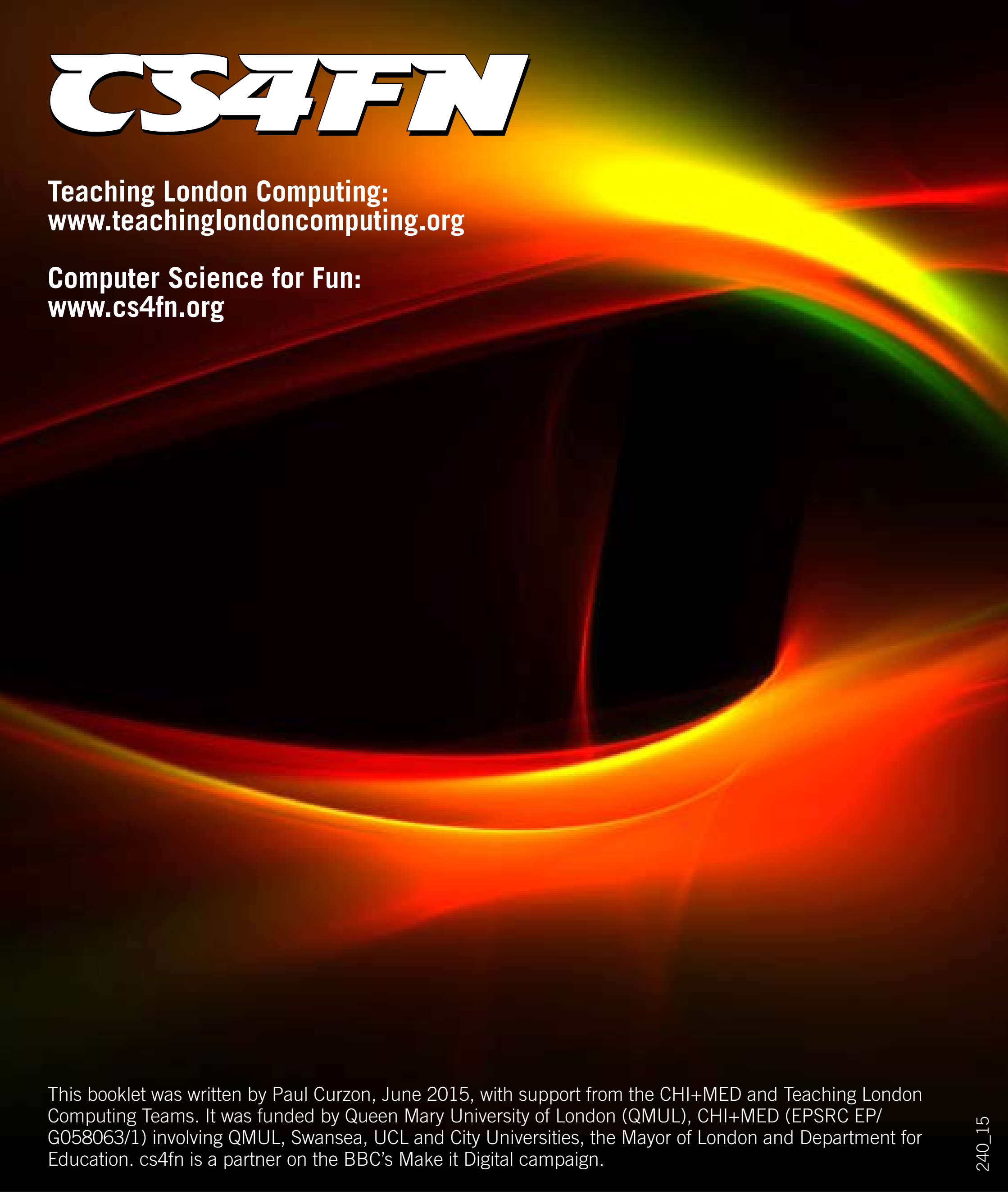
Flexagon Birthday Cards

You can have lots of fun with flexagons if you are creative. Add pictures to the sides and you can create a puzzle that involves finding a hidden picture. You could even create a hexahexaflexagon birthday card with hidden messages to find.

The other side of the flexagon…

We have completely ignored the other side of the flexagon! Turn it over and some sides have 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 overlap with those we already know about? If they do how do we tell them apart? 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 sides without turning the flexagon over? It's time to create a finite state machine for the flexagon as a whole.

All in all, graphs and finite state machines are incredibly useful… there is a lot of computational thinking in a hexahexaflexagon!



CS4FN

Teaching London Computing:
www.teachinglondoncomputing.org

Computer Science for Fun:
www.cs4fn.org

This booklet was written by Paul Curzon, June 2015, with support from the CHI+MED and Teaching London Computing Teams. It was funded by Queen Mary University of London (QMUL), CHI+MED (EPSRC EP/G058063/1) involving QMUL, Swansea, UCL and City Universities, the Mayor of London and Department for Education. cs4fn is a partner on the BBC's Make it Digital campaign.

240_15

Queen Mary University of London

Department for Education

BBC MAKE IT DIGITAL

SUPPORTED BY MAYOR OF LONDON

COMPUTING AT SCHOOL EDUCATE · ENGAGE · ENCOURAGE

EPSRC Engineering and Physical Sciences Research Council

chi+med