Edgy Activity - Creating Networks in Edgy

CS4S Maths Workshop @ the University of Newcastle

# Introduction

You will learn about the following *Networks* concepts in this activity:

* *Graphs and Subgraphs*
* *Complete Simple Graphs*
* *Random Graphs*
* *Cyclic Graphs*
* *Connected Graphs*

This activity will also involve applying some of the concepts that you learned about in the *Coding in Snap! Activity*, including:

* *Sequencing*
* *Repetition*
* *Variables*
* *Functions*
* *Branching*
* *Lists*

You will also learn about *Stacks* and *Dictionaries*, which are *Collections* of items that are like *Lists*.

# The Project

The project that you make in this activity will involve creating *Random Graphs* and running *algorithms* on those *Graphs*. There are two main *algorithms* that you will implement in this activity, which are *algorithms* for finding whether a *Sub-Graph* with *edges* of a certain colour:

1. is *connected*
2. contains a *cycle*

These *algorithms* will also be used in the activity where you write other *algorithms* in *Edgy* to find a *Graph's Minimum Spanning Tree*. The main aim of this activity is to familiarise you with the *Edgy* interface and show you some different *algorithms* that can be applied to *Graphs* in *Edgy*.

While we will only look at fairly small *Graphs* of 26 *nodes* or less, it is possible to create graphs that have more than 100 *nodes* in *Edgy*. The default limit on the number of *nodes* is 150, but this can be increased up to 1,000 *nodes* by using the *maximum visible nodes* option. The real power of *Graph algorithms* is the ability for analysing and solving problems that involve large networks that have many *nodes* and *edges*. We will look at some examples of applying Networks to real-world problems in the workshop and also show you some example datasets, such as social network graphs and large road networks, that can be downloaded from websites like: [The Koblenz Network Collection](http://konect.uni-koblenz.de/networks/) and the [Network Repository](http://networkrepository.com/). It is possible to use these datasets in *Edgy*, provided that they are in the correct format.

# Edgy vs Snap!

[*Edgy*](http://www.snap-apps.org/edgy.html) is a modification of the *Snap!* programming language that was created by researchers at the *University of Melbourne* and *Monash University*. *Edgy* has been used in the new VCE (Victorian Certificate of Education) *Algorithmics* subject, which involves the teaching of *algorithm design* and *analysis*. We have included some links to websites that explain *Edgy* in more detail on this session's page on the workshop website.

You may notice that *Edgy*, unlike *Snap!*, does not have the following categories of blocks:

* *Motion*
* *Sensing*

Instead, Edgy has four categories of blocks that Snap! does not have:

* *Network*
* *Nodes*
* *Edges*
* *Collections*

*Edgy* does not really have *Sprites*, unlike *Snap!*. Instead of *Sprites*, one *Graph* is displayed on the *Stage* and this *Graph* is manipulated through the blocks in the *Scripts Area*. Currently, it is not possible to create and interact with multiple *Graphs* in *Edgy*.

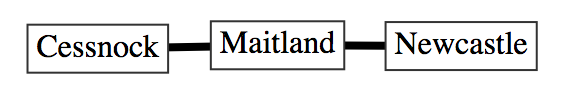
Before moving onto the next section of the activity, you may want to explore some of the blocks in the different sections and see how they affect the graph that is created in *Edgy*. If anything in *Edgy* does not work as you would expect when clicking the different blocks, please let us know.

# Creating Graphs on the Stage

In *Edgy*, you can create graphs by using *Code* blocks, but you can also add *edges* and *nodes* by using the *Stage* menu. Right-click on the *Stage* area and a menu will appear. You can create a *Graph* by using the *add node* and *add edge* options. For example, you could add the following *nodes* and *edges* to create a simple road network:

* node*: Newcastle*
* node*: Maitland*
* node: *Cessnock*
* edge with Start node*: Newcastle* andEnd node*: Maitland*
* edge with Start node*: Maitland* and End node*: Cessnock*

The resulting *Graph* will look like the *Graph* in the image shown below:



Note that, if you add an *edge* where the *nodes* are not in the *Graph* - for example: an *edge* that connects the node *Swansea* to a node called *Sydney* - those *nodes* will be added to the *Graph*. You can also adjust the colours, scales and shapes of *nodes* and *edges*, or even delete *nodes* and *edges*, by right clicking on them and using the menu.

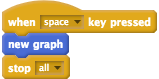
You are also able to import files, that are in certain formats (such as the DOT format), to create *Graphs* in *Edgy*. To see how this works, please download the *Example Food Web Network DOT File* from this session's page on the workshop website and save it somewhere easy to find (such as *the Desktop*). If you would like to see how the DOT file is structured, the file can be opened in text-editor programs like *Notepad* or *TextEdit*. Next, right click on the *Stage*, select the *import graph from file* option and open the DOT file that you downloaded from the workshop website. A *Graph*, which is a food web, will appear on the *Stage*. The layout of the *Graph* is automatic and may also be hard to read because the *nodes* and *edges* bunch together, but you can adjust the layout by selecting the *use manual layout* and dragging the nodes around the *Stage*. When an animal/plant has an edge pointing to another animal, this means that the animal/plant is consumed by that other animal. For example, the Frog *node* has an *edge* pointing to the Python *node*, because Pythons eat Frogs.

This feature for importing *Graphs* can be very useful for analysing interesting networks (such as the social networks mentioned in the *The Project* section above) into *Edgy*. In this activity, we will focus on creating *Graphs* using blocks instead, which we will begin to look at in the next section.

# Sequencing in Edgy

## Adding Reset Code

Before you create *Graphs*, you should create the following stack of blocks:



These blocks will allow you to press the space bar to erase the *Graph* that is currently on the *Stage* and stop all blocks from running.

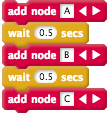
## Adding Nodes

Now, we will create a *Graph* by using blocks. Add the following blocks to the *Scripts* area in *Edgy*:



Once you click the stack of blocks, three *nodes* (named *A*, *B* and *C*) will appear on the *Stage*. You may notice that these *nodes* all appear almost instantly. This is because we are not telling the *program* to wait between the commands. This is similar to how a *Sprite* in *Snap!*, that is told to move with multiple *move* blocks and no *wait* blocks, will only make one movement.

We could add *wait* blocks to introduce slight pauses between adding each of the nodes. Clear the *Graph* by pressing the space bar and change the stack of blocks to look like those below:



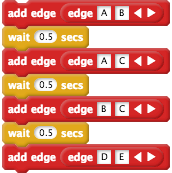
## Adding Edges

This activity will be focused on undirected *Graphs*, which are *Graphs* that have *edges* with no direction.

Now, add the following blocks to your program after the blocks that add the *nodes*:



Like the example with the *nodes* above, these *edges* appear almost instantly. You may want to introduce some pauses between the adding of *edges* as well. Clear the *Graph* again by pressing the space bar and change your blocks to look those shown in the image below:



You may notice that when the last *edge* (*D* connected to *E*) is added to the *Graph*, the *D* and *E* *nodes* are added to the *Graph* as well.

## Attributes of Nodes and Edges

*Edgy* allows you to change different attributes of *nodes* and *edges*, such as: *nodes'* shapes or *edges'* colours. You can use blocks like the *set attribute of node* and *set attribute of edge* blocks to change *nodes'* and *edges'* attributes.

For example, add the following block after your stack of blocks, clear the *Graph* and click the stack of blocks.



After running the stack of blocks, the *A node* should now be shaped like a circle. Next, add the following block, clear the *Graph* and click the stack of blocks.



The *edge* that connects *D* and *E* should now be coloured red. In the next section, we will use *Repetition* to create larger *Graphs* with a smaller amount of blocks.

## Check Your Understanding

Now that we are the end of the *Sequencing in Edgy* section, try and answer the questions below. If you need any clarification about these questions, please let us to know. The solutions to these questions are also available on the workshop website.

### Sequencing: Exercise 1

A student creates the following stack of blocks:



The student expect that, after clicking all of the blocks, all of the *nodes* will be shaped like circles, but only two of them are. Why are only two of the *nodes* shaped like circles and how could the students change the stack of blocks above so that all the *nodes* are shaped like circles?

### Sequencing: Exercise 2

Read the blocks below and, without using *Edgy*, draw what the resulting *Graph* would look like.



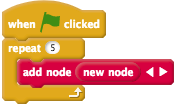
Is the *Graph* created by the blocks above *connected* and/or *cyclic*?

# Repetition in Edgy

## Adding Nodes in a Loop

In this section, you will learn how to create *Graphs* in *Edgy* with *Repetition*. You can move the blocks that you used in the last section away from the other blocks or delete them, as we will be building a different stack of blocks for the rest of the activity.

Create the following stack of blocks, which will create a *Graph* with 5 *nodes* and no *edges*.



The *new node* will add a *node* that is labelled with the number that is 1 more than the largest label in the *Graph*. So, for example, after running the stack of blocks above and then clicking the *new node* block in the *Nodes* section, a *node* with the label *6* will appear in the *Graph*.

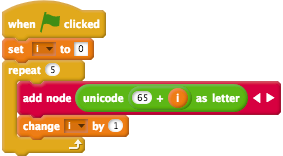
We have not included *wait* blocks in the rest of the activity's examples, to keep these examples concise. However, you can add these if you would like to have pauses between adding *nodes* and/or *edges*.

Instead of using numbers for our *nodes'* labels, we are going to use letters. The *Unicode* format can be used to represent letters with numbers. Each number in the *Unicode* format corresponds to a letter, for example, the number *65* represents the upper-case *A* and the number *97* represents the lower-case *a*. We will create *nodes* with a loop using the range of *Unicode* numbers *65* - *90*, which are the 26 capital letters of the alphabet.

We will need to use a *variable* to keep track of the *Unicode* number in our loop for creating these *nodes*, which we will call *i*. To create the *i* variable, follow these steps:

1. Go to the *Variables* blocks section of the *Blocks Palette*.
2. Click the *Make a Variable* button
3. Name the variable *i*
4. Tick the *For this sprite only* checkbox
5. Click the *OK button*

Next, change your stack of blocks so that they look like the image below:

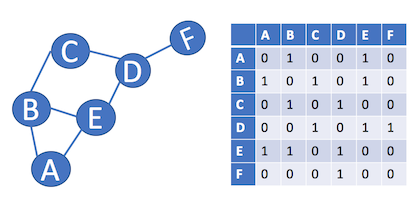


In the blocks above we create a graph with 5 *nodes* with labels, by using *Unicode* codes (which are in the brackets following the letters): *A* (65), *B* (66), *C* (67), *D* (68) and *E* (69).

## Adding Edges in a Loop

Next, we will use *nested repetition* to create *edges* in our *Graph*. In a previous workshop activity you looked at *adjacency matrices*, which can be used to represent *Graphs*.

An example *adjacency matrix* is shown in the image below.

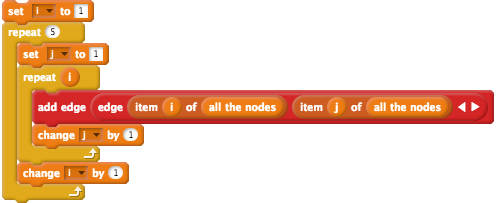


We are going to make a stack of blocks that will connect all of the *nodes* to every other *node* with an *edge*. To do this, we will have to repeat the *add edge* command for every column and row of the *adjacency matrix*.

We will use two *variables* in the *nested loop* for adding the *edges*. We already have a *variable* called *i*, but you will need to create another *variable* called *j* by following these steps:

1. Go to the *Variables* blocks section of the *Blocks Palette*.
2. Click the *Make a Variable* button
3. Name the variable *j*
4. Tick the *For this sprite only* checkbox
5. Click the *OK button*

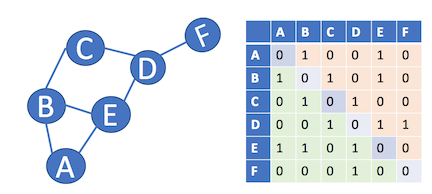
Next, add the following blocks to the bottom of your stack of blocks that add the *nodes*.



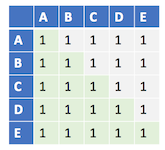
Note that the *i* and *j* variables both start at *1*. The *item of all the nodes* block will give us the appropriate *node* for the given number - item 1 of *all the nodes* is *A*, item 2 of *all the nodes* is *B* and so on. If we created the *adjacency matrix* for the *Graph* created by these blocks above, it would look like the matrix below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **A** | **B** | **C** | **D** | **E** |
| **A** | 1 | 1 | 1 | 1 | 1 |
| **B** | 1 | 1 | 1 | 1 | 1 |
| **C** | 1 | 1 | 1 | 1 | 1 |
| **D** | 1 | 1 | 1 | 1 | 1 |
| **E** | 1 | 1 | 1 | 1 | 1 |

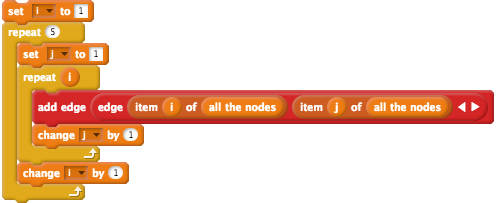
You may recall that *adjacency matrices* for undirected *Graphs* are symmetrical. For example, if we revisit the *Graph* and *adjacency matrix* shown before, you can see that the green and orange cells in the *matrix* are the same, just mirrored.



So, if we have the *adjacency matrix* for our *Graph* in *Edgy*, which is shown below, then we can just add edges for the cells in green and we will still end up with the same resulting *Graph*.

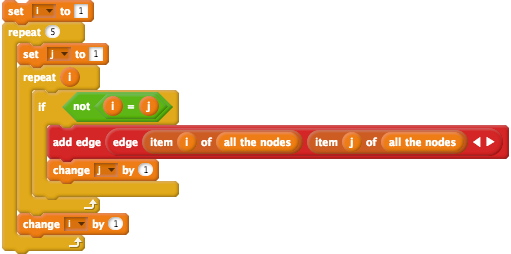


Consequently, we can change the *5* in the inner *repeat* block to the *variable i*, as shown in the image below, and the resulting *Graph* will be the same as before:



## Making a Simple Complete Graph

*Graphs* that have no *loop edges* (*edges* that connect a *node* to itself) and that don't have more than one *edge* between each pair of *nodes* are called *simple Graphs*. The *Graph* that we have created is not a *simple Graph* because it does contain *loop edges*. We can use *branching* to stop the blocks from drawing the loop *edges*, by not adding an *edge* to a *Graph* when the *i* and *j* variables are the same. So, for example, when *i* and *j* are both 3, our *Code* will make sure that an *edge* that connects the *node C* to itself will not be added to the *Graph*.



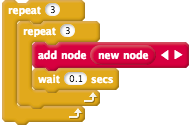
Not only is the resulting *Graph* a *simple Graph*, it is also a *complete Graph*. A *complete graph* is a *Graph* where there is an *edge* between every *node*, which means every possible *edge* in the *Graph* is present. *Complete graphs* are usually denoted by Kn, where *n* is the number of *nodes* in the *Graph*. We have created the K5 *Graph* with the blocks above, but we could adjust the number of *nodes*. For example, you could create K9 by changing the number *5* in the 2 *repeat* blocks to *9* and recreating the *Graph*.

## Check Your Understanding

Now that we are the end of the *Repetition in Edgy* section, try and answer the questions below. If you need any clarification about these questions, please let us to know. The solutions to these questions are also available on the workshop website.

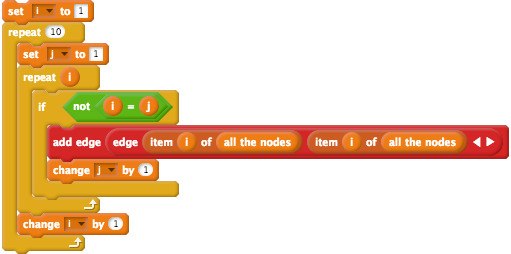
### Repetition: Exercise 1

After the stack of blocks below are clicked, how many nodes will be in the graph?



### Repetition: Exercise 2

A student has added 10 *nodes* to a *Graph*, has the following stack of blocks and is hoping that the blocks will draw a complete *simple Graph K10*.



However, after running the blocks the resulting graph is not complete. The graph has 10 nodes but they are not all connected. The student has made a mistake in their *Code*, what is it?

# Randomisation in Edgy

## Creating Random Graphs

Now, we are going to modify the *program* to create *random Graphs*. There are a few different ways of creating *random Graphs* but the approach we will use in this activity involves using the Erdős–Rényi model of *random Graphs*. We have included some links which explain *random Graphs* in more detail, and gives some examples of where they are applied, on the workshop website.

*Random Graphs* in the Erdős–Rényi model are denoted by G(n,p), where *n* is the number of *nodes* in the *Graph* and where each possible *edge* in the *Graph* has *p* probability of being present in the *Graph*. So, for example, if there is a *random graph* with two *nodes*: *A* and *B* and *p* of 25%, then there will be a 25% chance that there will be an *edge* between *A* and *B*.

Before we add *randomisation* to the blocks, you will need to create a *number of nodes* variable by following these steps.

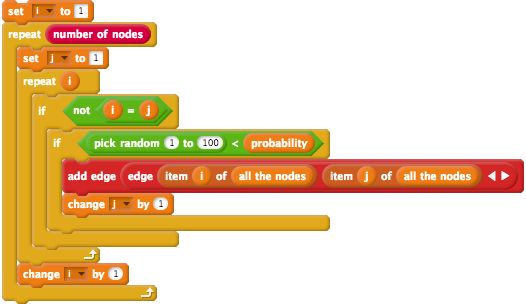
1. Go to the *Variables* blocks section of the *Blocks Palette*.
2. Click the *Make a Variable* button
3. Name the variable *number of nodes*
4. Tick the *For this sprite only* checkbox
5. Click the *OK button*

After creating that *variable*, replace the *5*s in the *repeat* blocks with the *number of nodes* variable. Next, repeat the same steps above for another *variable* called *probability*.

You should also add *set variable* blocks, just after the *when green flag clicked* block, that set the value of the *number of nodes variable* to a number (such as 5) and the *probability variable* to a number (such as 25).

Next, we will use *nested branching* and *randomisation* to create *random graphs*. The *p* (the probability of the *edges* being in the *Graph*) will be a number between 1 and 100, which will represent the percentage of likelihood that an *edge* is added to the *Graph*.

Now, we will change the loop that creates all the *edges* to look like the blocks below.



So for example, for when the *probability variable* is 25 (a probability of 25%), a number between 1 and 100 is chosen randomly. If the random number is less than the probability (for example: if the random number is 22), then the appropriate *edge* will be added to the *Graph*.

You could try changing the values of the *probability* and *number of nodes* variable and see the resulting *Graphs*. What happens when the *probability* variable is 0 and what happens when the *probability* variable is 100? Why is that? Are the *Graphs* that generated *cyclic* and/or *connected*? Are the *Graphs* created with a higher probability more likely to be *connected* and/or *cyclic*?

## Setting Random Edge Weights

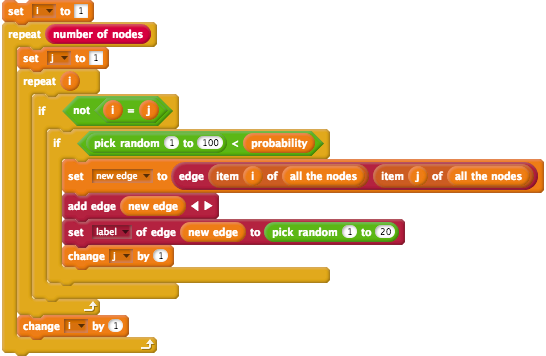
Next, we will add random weights to all of the edges in the *random Graphs* we create.

*Edge weights* usually represent a *cost*. For example, in a road network the *edge weigts* could be the distance between two cities/landmarks. In *Edgy* we can assign *edge weights* by setting the *label attribute* of an *edge*.

Firstly, create another *variable* called *new edge* by following these next steps.

1. Go to the *Variables* blocks section of the *Blocks Palette*.
2. Click the *Make a Variable* button
3. Name the variable *new edge*
4. Tick the *For this sprite only* checkbox
5. Click the *OK button*

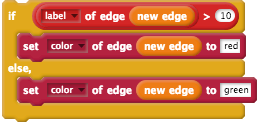
You will use the *new edge variable* when setting the *edges' weights*. To set the *edges weights* to random numbers, you can change your blocks to look like the following stack of blocks.



## Colouring Edges

Now, we will change the colour of the *edges*. If the *edge* has a weight larger than 10, we will make it red. If the *edge* has a weight less than 10, we will make it green.

To do this, add the following stack of blocks after the *set label of edge* block (where a random weight is added to an *edge*), in the inner *if* block.

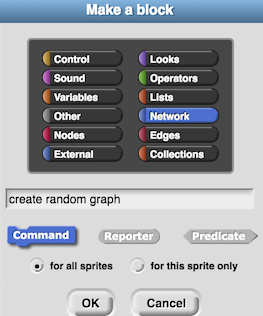


Now, when you create a new *random Graph*, you will notice that the *edges* are now coloured red or green, depending on their weight.

## Creating a Random Graph Block

In the next section, you will learn how to implement some *algorithms* for finding whether a *Graph* is *cyclic* and/or *connected*. Before moving onto the next section, however, the blocks that you have created should be placed into a custom block called *create random graph*.

To create the *create random graph* block, right-click anywhere on the *Scripts Area* (except for the stacks of blocks) and click the *make a block...* option in the menu. After selecting the *make a block...* option from the menu, a popup window titled *Make a block* will appear. We want the *create random graph* block to appear in the *Networks* section of the *Blocks Palette*, so you should select the *Networks* option in the *Make a block* window. You will also have to enter the block name (*create random graph*) and make sure that the *Command* block is selected. The form in the *Make a block* popup should look like the image below:

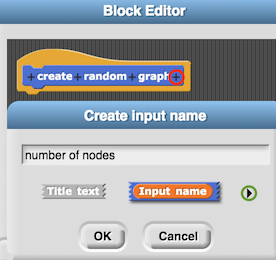


After clicking the *OK* button, the *Block Editor* will open. The *Block Editor* window is where you will add *Code* blocks to define the actions that your new block will perform. You can re-open this window at any time by right-clicking on one of the blocks you have made and clicking the *edit...* option.

The *create random graph* block will have two *inputs*, which will both be *Numbers*, called: *number of nodes* and *probability*.

To add an *input* to the block:

1. Click the + next to *graph* in the block name (as highlighted by the red circle in the image below)
2. When the *Create input name* window appears, put the name of the input (which we will call *number of nodes*) in the textbox, as shown in the image below.

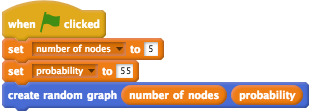


Before you click *OK*, click the right arrow (highlighted by the green circle in the image above) and you will see options appear for choosing the type of *variable* that the *input* is (such as: *Number*, *Boolean* and *List*). We want the *number of nodes* to be a *Number*, so select the *Number* option from these options and click *OK*. The *number of nodes input* should appear in an orange oval (like a *variable*) in the block name and there should also be a # added to the end (to signify that the *input* is a number).

Repeat the same steps above for creating an *input* to the *create random graph* block called *probability*.

The blocks from the *set i to 0* block at the top of the stack, to the bottom of the stack, should be dragged under the *create random graph* block in the *Block Editor*.

After creating this block, you will want to place the *create random graph* after the *set probability to* block. Next, you should use the *number of nodes* and *probability variables* as *input* to the *create random graph* block. Now, your stack of blocks should now look like the stack of blocks shown below.



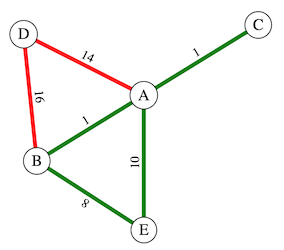
You now have a *Function* for creating *random Graphs*, that you will be able to use for testing the *algorithms* you implement in the next sections. Before moving on to the next section, you may want to try creating *random Graphs* with different values for the *number of nodes* and *probability inputs* and see the different *Graphs* that are created.

## Check Your Understanding

Now that we are the end of the *Randomisation in Edgy* section, try and answer the questions below. If you need any clarification about these questions, please let us to know. The solutions to these questions are also available on the workshop website.

### Random Graphs: Exercise 1

You create a *random Graph* and the result is pictured below.



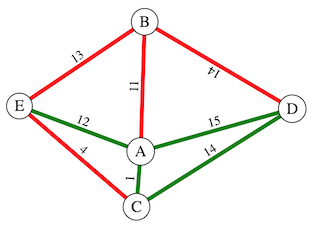
This *Graph* contains two *Sub-Graphs*: one which contains all the *nodes* and only the red-coloured *edges* (GR), and one that also contains all the *nodes* and only the green-coloured edges (GG).

Is GR *connected* and/or *cyclic*?

Is GG *connected* and/or *cyclic*?

### Random Graphs: Exercise 2

You create another *random Graph* and the result is pictured below.



This *Graph* contains two *Sub-Graphs*: one which contains all the *nodes* and only the red-coloured *edges* (GR), and one that also contains all the *nodes* and only the green-coloured edges (GG).

Is GR *connected* and/or *cyclic*?

Is GG *connected* and/or *cyclic*?

# Connected Graphs

## Finding a Connected Graph

In the previous *Check Your Understanding* exercises, you could tell whether a *Graph* was *connected* by inspection. However, would you be able to tell if a *Graph* with hundreds of *nodes* and *edges* was *connected*, just by looking at it? For example, the police might investigate whether a group of 200 criminals are connected in some way. The police could analyse the criminals' phone records to create *edges* in the *Graph* where two criminals (represented by *nodes*) had spoken on the phone to each other. Rather than inspecting the *Graph* to tell whether the network of criminals is *connected*, the police could use an *algorithm* to find this out. If the police find that the *Network* of criminals is *connected*, they could conclude that it is likely that the criminals all belong to the same criminal organisation.

You will learn how to implement an *algorithm* for finding whether or not a *Graph* is *connected* in this activity. *Edgy* already has a block for finding the *connectedness* of a *Graph* called *is connected*. However, the block that we make in this activity will allow us to find whether a *Sub-Graph* with *edges* of a certain colour is *connected*, which the *is connected* block in *Edgy* does not do.

You may recall that a *Graph* is *connected* if there is at least one path from every *node* to every other *node* in the *Graph*. So, if we start at some *node* in a *connected Graph*, follow its *edges* to its neighbours, then follow the neighbours' *edges* to their neighbours, and so on, we should visit every *node* in the *Graph*. However, if we follow all the *edges* from any *node* in the *Graph* and find that, after following all of the *edges*, some of the other *nodes* are not visited - then the *Graph* is *not connected*.

## Depth-First Search

A common approach for finding whether a *Graph* is *connected* is using a *depth-first search*. A *depth-first search* generally involves the following steps:

1. Choose a start *node*
2. Look at all the *node's neighbours*
3. Mark all the *neighbours* that have not been visited yet as *to visit*
4. If there are some *nodes* left *to visit*, take the last *node* marked as *to visit* and go to Step 2. If there are no *nodes* left *to visit*, you are finished the search.

To implement a *depth-first search* in *Edgy* we need to use a *Collection* called a *Stack*. A *Stack* is similar to a *List*, but, unlike a *List*, a *Stack* only has three main ways of accessing and modifying its items. You can imagine a *Stack* to be like a deck of cards, that you can only perform the following actions on:

* *push:* put an item on the front of the *Collection* - you could think of this as placing a card on the top of a deck.
* *peek:* look at the first item in the *Collection* (this is called the *top of stack* block in *Edgy*) - you could think of this as having a look at the top card on a deck and then putting it back on the top of the deck.
* *pop:* removing the first item in the *Collection* - you could think of this as taking the top card on a deck and then discarding the card.

For example, part of the *depth-first search algorithm* will involve pushing a *node's* neighbours to a *Stack*. The *Stack* will be a *variable* that we will call: *to visit*. To create the *to visit variable*, you can follow the steps described below.

1. Go to the *Variables* blocks section of the *Blocks Palette*.
2. Click the *Make a Variable* button
3. Name the variable *to visit*
4. Tick the *For this sprite only* checkbox
5. Click the *OK button*

Now, we will add some blocks to add the neighbours of our first *node* *A* to the *to visit* *Stack*, which will go after the *create random graph* block you created in the previous section. In this activity, we will use some blocks that are not initially visible in *Snap!* and *Edgy*. To show these blocks, you will have to click the menu at the top of the screen that looks like a blank piece of paper (the *File* menu) and then click the *Import tools* option. After clicking the *Import tools* option, you may notice that there are some new blocks in the *Block Palette*. For example, in the *Variables* section you may notice that there is now an *empty?* block.

When a depth-first search is performed on a *Graph*, a start *node* node is usually chosen randomly and marked as *to visit*. In the depth-first search we create in *Edgy*, we will push the first of our *nodes* (*A*) onto a *Stack* (the *to visit* variable). These steps can be seen in the first two blocks in the stack of blocks below.



The first block above sets the *to visit* variable to an empty *Stack* and the second block pushes the first *node* in the *Graph* onto *to visit*, which places that *node* (*A*) on the top of the *Stack*. You can rename *item* by double-clicking on the orange *item* bubble and typing in *neighbor* instead. After running your *program*, the *to visit* Stage monitor should have the *neighours* of the *node* *A* in it.

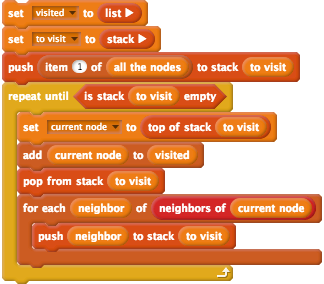
In the *depth-first search*, we will use another *variable*, called *visited*, which will be a *List* of nodes. When the search *visits* a *node* that has not been visited by the search before, that *node* will be added to the *visited* list. Consequently, the *visited List* will be empty to begin with, as the search will not have visited any *nodes*. To create the *visited variable*, follow the steps above for creating the *to visit variable*, but use *variable* name: visited\* instead.

To demonstrate how we will add *nodes* to the *visited* *List*, change your stack of blocks so that they look those in the image shown below. Note that the stack of blocks below will be modified extensively for the *depth-first search*, these are mainly used to demonstrate how we will add items to, and remove items from, the *to visit* \*Stack.

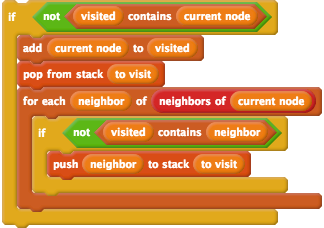


After running the stack of blocks above, you may notice that the items in the *visited* *List* are now in reverse order to what they were in the *to visit* *Stack*. In the *repeat until* block, the top *node* on the *to visit* *Stack* is taken off the top, put at the end of the *visited* *List* and then put the *node* back on the top of the *Stack* Then, the *pop* action is performed on the *to visit Stack*, which removes the top *node* from the *Stack*. These two actions are repeated until all of the items in the *to visit* *Stack* are removed (when it is *empty*).

You will have to rearrange the blocks above, so that all of the *connected* *nodes* are added to the *to visit* *Stack* and the *visited* *List*. But first, you will need to create another *variable* named *current node*, which you can do by following the steps for creating the *to visit variable*, but using the *variable name visited* instead. This *variable* will be used to keep track of the *node* that is currently being *visited*, when we implement the *depth-first search algorithm*. Note that you will not want to click on these blocks yet, we need to add some extra blocks in the next step first.



In the blocks above, the *for each* block from the previous example has been placed inside the *repeat until* block. The *current node* *variable* has also been added to the blocks, to keep track of the *node* that is currently at the top of the *to visit Stack*. However, the blocks above will not work correctly - as they do not include any blocks to check whether the *current node* or *neighbor nodes* have already been visited. If a *node* has already been visited, the depth-first search does not need to search it again and consequently, you will need to use *Branching* to check whether a *node* has already been visited. To do this, you will need to create two *if* blocks, which are shown in the image below. The first *if block* will "wrap around" the last four blocks (*add*,*pop from stack*,*for each* and *push to stack*) and will check if the *current node* has already been visited. The second *if block* will be placed inside the *for each* block and will check if the *current node's neighbours* have already been visited. Consequently, only unvisited neighbours will be added to the *to visit Stack*.



The *if* block above will check if the *visited List* does not contain the *node* from the top of the *to visit* *Stack*, and if so, the *algorithm* will add that *node* to the *visited List* and push all of that *node's* unvivisted *neighbours* to the *to visit* *Stack*.

After adding the *if* blocks, you may want to create some different *random Graphs* and then run the blocks above. What do you notice about the items in the *visited List* for the *Graphs* that are *connected*, compared to those that are not *connected*?

## Checking for Connectedness

You may have noticed that when *Graphs* are *connected*, the *visited List* will contain all of the labels of the *nodes* in the *Graph*. Therefore, we will be able to tell when the *Graph* is *connected*, when the length of the *visited List* is the same as the number of *nodes* in the *Graph*.

In the next step, we are going to make our blocks *report* when the *Graph* is *connected*. Add the following blocks to the bottom of your stack of blocks, after the *repeat until* block:



Now, if you click the stack of blocks, a speech bubble will appear with *true* (if the *Graph* is *connected*) or *false* (if the *Graph* is not *connected*). Try some different *random Graphs*, of larger size (for example: 20+ nodes), and see if the reported result (*true* or *false*) matches what you expect.

Currently, the blocks we have *report* whether the *Graph* is *connected*, regardless of the colour of the *edges*. In the next steps, you will add blocks to *report* whether *Sub-Graphs* with *edges* of certain colours are *connected*.

## Subgraphs with Coloured Edges

You may recall that, as explained in the *Colouring Edges* sub-section of the *Randomisation in Edgy* section, there are blocks in *Edgy* that *report* attributes of *edges*. As we are implementing a depth-first search for *Sub-Graphs* with *edges* of a certain colour, this means that we only want to visit neighbours of the *current node* when the *edge* between them is that certain colour. To do this, you need to change the *for each* part of the stack of blocks to look like the blocks in the image below, which includes a check for the appropriate colour (which is *green* in the example below).

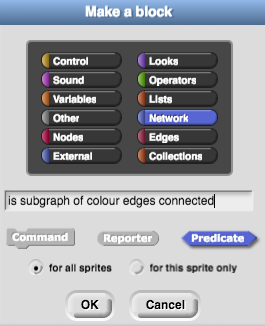


At the moment, the blocks above only work for *edges* coloured green but in the next steps we will put these blocks into a *Function*, which will take any colour name as *input*.

## Creating a Custom Block

Now, we are going to create a block for checking whether a *Sub-Graph* with *edges* of a certain colour is *connected* and name it: *is subgraph of colour edges connected*. Before making this block, duplicate all of the blocks from after the *create random graph* by right clicking on the stack of blocks and clicking *duplicate*. Place this duplicated stack away from, and seperate to, your other blocks. You can keep the duplicated and separate stack as it is for now, we will modify that stack in the next section of the activity.

To create the *is subgraph of colour edges connected* block, right-click anywhere on the *Scripts Area* (except for the stacks of blocks) and click the *make a block...* option in the menu. After selecting the *make a block...* option from the menu, a popup window titled *Make a block* will appear. We want the *is subgraph of colour edges connected* block to appear in the *Networks* section of the *Blocks Palette*, so you should select the *Networks* option in the *Make a block* window. You will also have to enter the block name (*is subgraph of colour edges connected*). The block that we make in this step will *report* one of two *Boolean* values (*true* or *false*), so it is appropriate to make it a *Predicate Function* by selecting the *Predicate* option. The form in the *Make a block* popup should look like the image below:

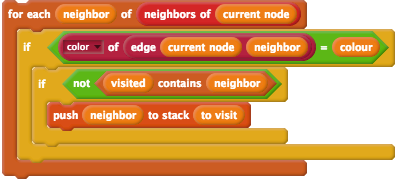


After clicking the *OK* button, the *Block Editor* will open. You can delete the *report* block that is after the *is subgraph of colour edges connected* *hat block*, because we will add our own *report* block in the *Block Editor*. Take all the blocks after the *create random graph* and drag them under the *is subgraph of colour edges connected* *hat block* in the *Block Editor*. The *is subgraph of colour edges connected* block will have one *input*, which will be called *colour*. The name *colour* is already in the name of the block but we can make this an *input* by following these steps in the *Block Editor*:

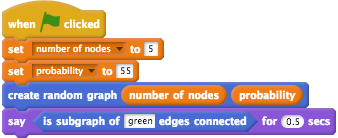
1. Click the *colour* text in the block name, as highlighted by the red circle in the image below
2. Select the *input name* option, as highlighted by the green circle in the image below



Next, we need to use the *colour input variable* in our stack of blocks when checking the colour of the *edge* between the *current node* and the *neighbour*. To do this, replace the text *green* with the *input variable* in the blocks that are in the *is subgraph of colour edges connected* block. That part of the stack should look like the image below:



Now, you can add the *is subgraph of coloured edges connected* after the *create random graph* block. You can put your custom block inside a *say* block, so that the result of the block is shown on the *Stage*. After adding these two blocks to your stack, the stack following the *when green flag clicked* block should look like the below image:

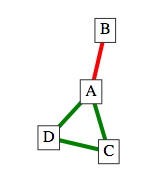


You have now successfully created the block for checking whether a *Sub-Graph* of certain coloured *edges* is *connected*. In the next section, you will extend the depth-first search *algorithm*, to check whether a *Sub-Graph* is *cyclic*.

## Check Your Understanding

Now that we are the end of the *Connected Graphs* section, try and answer the question below. If you need any clarification about this question, please let us to know. The solutions to this question is also available on the workshop website.

### Connected Graphs: Exercise 1



When we use our *is subgraph of coloured edges connected* block for finding whether the *Sub-Graph* of green *edges* and all *nodes* is *connected*, does it matter which *node* the block starts from? For example, will the result of the block be false if it starts from *node B*?

# Cyclic Graphs

## Finding a Cyclic Graph

In the *Graphs* we have looked at so far, it can be easy to tell whether a *Graph* is *cyclic* by simply looking at it. But, for *Graphs* of larger sizes, *programs* have to be written to find whether a *Graph* is *cyclic*.

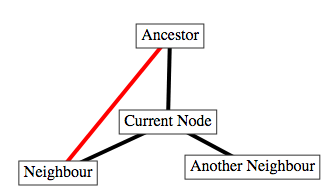
You may recall that *Trees* are acyclic and consequently *algorithms* for finding *cycles* are used in cases where *Networks* have to be *Trees*. Consequently, *algorithms* for finding *Trees* in a *Graph*, such as the *Kruskal's* algorithm you will learn how to implement later, often include steps that check whether a *Graph* is *cyclic*.

One method of checking for *cycles* in a *Graph* is explained in the next steps.

## Back Edges

One way to find whether a *Graph* is *cyclic*, is to find out whether there is a type of *edge*, called a *back edge*, in the *Graph*. In this activity, you will extend the depth-first search *algorithm* to check whether the *Graph* contains a *back edge*.

When we are performing the *depth-first search* and *visiting* the *current node*, we check for *back edges*. A *back edge* is an *edge* that connects one of the *current node's neighours* to one of the *current node's ancestors*. An example of a *back edge*, which is coloured red, is shown in the image below.



In the example above, there are three *edges* that form a *cycle* in the *Graph*: (*Ancestor*, *Current Node*), (*Current Node*, *Neighbour*) and (*Neigbour*, *Ancestor*). The *back edge* that has been found is the *edge* that connects the *Neighbour node* (one of *Current Node's neighbours*) and the *Ancestor node* (one of *Current Node's ancestors*).

A description of the steps for finding this *back edge*, that starts at the *Ancestor node*, are as follows:

1. We would visit the *Ancestor node* first and mark that its *neighbours* (*Current Node* and *Neighbour*) need to be visited
2. We would record that *Ancestor node* is the *parent node* of both *Neighbour* and *Current Node*
3. We would then visit the *Neighbour*, as it would be the last *Node* to be marked as "to visit"
4. We would look at each of the *Neigbour node's neighbours* and:
   1. The neighbour that has not been visited, *Current Node*, will be marked as "to visit" and we would record that the *Neighbour node* is the *parent node* of *Current Node*
   2. For the neighbour that has been visited, *Ancestor*, we check if it been recorded as the *parent* of *Neighbour*. That is the case, so this tells us the *edge* between *Ancestor* and *Neighbour* is not a *back edge*
5. We would then visit the *Current Node* as it would be the next *Node* marked as "to visit"
6. We would then look at each of the *Current Node's neighbours*, both of which (*Ancestor* and *Neighbour*) have been visited:
   1. We check if the first *neighbour*, *Ancestor*, has been recorded as the *parent* of *Current Node*. In Step 4.a, we marked the *parent* of *Current Node* as *Neighbour* and consequently *Neighbour* is the *parent* of the *Current Node*, not *Ancestor*. This tells us that there is a *back edge* between *Neighbour* and *Ancestor* and therefore the *Graph* is *cyclic*

In the next steps of the activity, we will implement the *algorithm* described above in *Edgy*.

## Parent Nodes

To keep track of the *parents* of the *nodes*, you will use a type of *Collection* called a *Dictionary*. A *Dictionary* is similar in some ways to a *List* but, instead of using numbers (such as an *index*) to access and modify items, you use text (*keys*). An example of a *Dictionary* that contains three different items is shown in the table below.

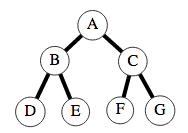
|  |  |
| --- | --- |
| **Key** | **Value** |
| Orange | A fruit that is orange |
| Apple | A fruit that can be a variety of colours |
| Banana | A fruit that is yellow |

The *Dictionary* in the table above has three pairs of *Keys* and *Values*. For example, the *Dictionary* item for *Apple* is "A fruit that can be a variety of colours".

There are three ways of accessing and modifying items in a *Dictionary* in *Edgy* that we will look at in this activity:

* *get key in dict*: this *Function* gives us the value in *Dictionary* for the given *key*. For example, if we use the *Orange* *key* to access the *Dictionary*, the result will be the value: *A fruit that is orange*.
* *set key in dict to value*: this *Function* adds a new *key* and *value* pair to the *Dictionary* or if there is already an item with that *key* in the *Dictionary*, the paired value will be changed. For example, if we used the *set key in dict to value Function* with the *key:* *Chicken* and the *value: A type of bird*, then this would be added to the *Dictionary* as a new item. On the other hand, if we used the *set key in dict to value Function* with the *key: Orange* and *value: A fruit that is usually orange*, the existing value (*A fruit that is orange*) will be replaced with the new value.
* *dict contains key*: this *reports* *true* or *false*, depending on whether there is an entry for the given *key*. For example, if we use the *dict contains key* *Function* with the *key:* *Orange*, the *result* will be *true*.

You will use a *Dictionary* in the next steps of the activity, in order to keep track of each visited *node's parent*, which we will name *parent*. For example, you could have a *Graph* that looks like the one shown below:



The *parent Dictionary* for the *Graph* above would have the following items:

|  |  |
| --- | --- |
| **Key** | **Value** |
| B | A |
| D | B |
| E | B |
| C | A |
| F | C |
| G | C |

The *parent* of *B* is *A*, the *parent* of *D* is *B*, and so on. Note that there is no entry for the *A key*, as it is the *root node* (it does not have any *ancestor nodes*).

## Checking for Back Edges

We will now modify the stack of blocks you duplicated earlier to check for *back edges*. This duplicated stack of blocks can now be placed after the *is subgraph of colour edges connected* block.

First, create a variable called *parent* by following the steps you completed in previous parts of this activity. This *variable* will be used for the *Dictionary* that keeps track of *nodes' parents*.

Add the following block after the *set to visit to* block, which will set the *parent variable* to an empty *Dictionary*:



Next, we want to add a block that adds an item to the *parent Dictionary*, to keep track of each of the visited *node's parents*. We have to do this at the same time as we push the *current node's neighbours* onto the *to visit Stack*. We want to add a block that adds this item within the *if* block that checks if the *neighbour* has been visited, as shown in the image below:



For example, if the *current node* is *A* and that *node* is neighbours with the *node B*, then an item with the *key: B* and *value: A* will be added to the *parent Dictionary*. After clicking the *green flag*, you will see that the *nodes* connected by red *edges* will be *keys* in the *parent variable's Stage Monitor*.

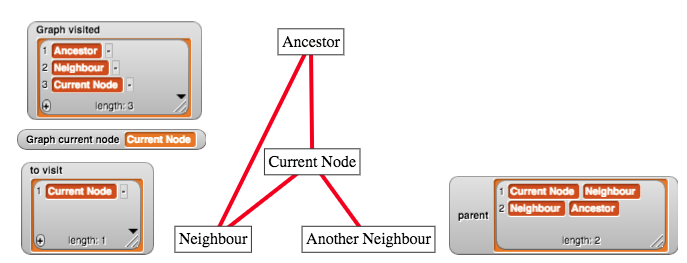
Now, we are going to add blocks to check for a *back edge*. Firstly, right click the *if* block that you just modified, click *relabel...* and select the first option (which changes the *if* block to an *if else* block). In the *else* section of the *if else* block we are going to add another *if* block that looks at the neighbours that the depth-first search has already visited and see if their *parent* is one of the *current node's ancestors*. The following *if* block goes inside the *else* section of the *if else* block, as shown below. Note that you can find the *report* block in the *Control* section and the *true* block in the *Operators* section.



There are some important parts about the blocks above to highlight. Firstly, there is a check in the *if* block that the *parent Dictionary* contains an item for the *neighbour*. This check is there because a *neighbour* that has been visited and is not in the *parent Dictionary*, is the *root node*. The *root node* will have no *ancestors* and consequently we do not need to check what its *parent* is. As both of these checks in the *if* block are in an *and* block, the second check will only be followed if *neighbour* is not the *root node*.

If the *neighbour* is not the *root node*, then we check if the *neighbour* has been marked as the *Current Node's parent*. If the *Current Node* already has another *parent* recorded in the *parent Dictionary* and it is not the *neigbour* then we know that there is a *back edge*.

For example, if we had the *Graph* and *variables* shown below, the blocks above would find the *back edge* from the *Neighbour node* to the *Ancestor node*.



Note that in the image above, the *Current Node node* is currently being visited, and that the *Neighbour node* is in the *visited List*. Additionally, the value of the item with the *key Current Node* in the *parent Dictionary* is not *Ancestor*, it is *Neighbour*. Consequently, the *report true* block would be followed, indicating that the *Graph* is *cyclic*.

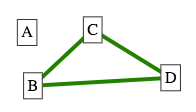
It is also important to note that when a *report* block is followed, it will report a *value* and then stop following the rest of the blocks in that stack. Therefore, if we reach the end of the stack of blocks without the *report true* block being followed, then we know there is no *cycle* in the *Graph*. To indicate that there is no *cycle* in the *Graph*, you can replace the *report* block you previously had at the bottom of the stack of blocks with the following block:



Note that the *false* block comes from the *Operators* section of the *Blocks Palette*. To make the *false* block, click the grey circle next to the *true* block, as highlighted in blue in the image below:



Lastly, we have to consider the case where the *node* we start from (*A*) is not *connected* to the other *nodes* in the *Graph*. For example, if we created the *Graph* shown below, our blocks would not *report* that it has found a *cycle* because *A* has no *neighbours* and would not visit any of the other *nodes* in the *Graph*.



Consequently, we have to change our *algorithm* to make sure every *node* is visited. To do this, you will have to take a *for each item* block and wrap it around the stack of blocks, starting from after the *set visited to* block and ending before the second *report* block. Next, click the *item* name in the *for each item* block and change it to *node* instead. After doing that, you will also have to change the block that pushes the first *node* onto the *to visit* stack to this instead:



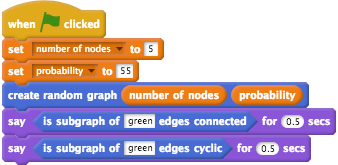
The final stack of blocks that implements this features is available on the solutions page of the workshop website, for you to compare your own stack of blocks against.

Now when you *click the green flag*, you should see a speech bubble appear with *true* or *false*, depending on whether the *Sub-Graph* with red *edges* contains a *cycle* or not.

## Creating a Custom Block

You should now move the blocks that you have created for checking whether a *Sub-Graph* of certain coloured edges into a *custom block*. Follow the same steps that you followed to create the *is subgraph of colour edges connected* block, but name the block *is subgraph of colour edges cyclic* instead. When you have the *Block Editor* open, drag the blocks that you created for checking for a *back edge* in a *Graph* underneath the *is subgraph of colour edges cyclic hat block*. This block will also have an *input* called colour, which should be used in the *if block* that checks the colour of the *edge*.

Next, place your newly created *is subgraph of colour edges cyclic* block in a *say* block and place it at the end of your main stack of blocks, after the *is subgraph of colour edges connected* block. After creating that custom block and placing it in a *say* block in your main stack of blocks, your blocks should look similar to those in the image below:



Now, if you click on the stack of blocks different *random Graphs* will be created and a speech bubble will appear. The first message will indicate whether the *Sub-Graph* of green *edges* is *connected* and the second message will indicate whether the *Sub-Graph* of green *edges* is *cyclic*.

# Conclusion

You have finished the *Creating Networks in Edgy Activity*. Good work!

The blocks that you have made in this activity (the *is subgraph with edges coloured?* and *is subgraph with edges coloured cylic?* blocks), will be used in the next activity, where you create algorithms in *Edgy* to find a *Graph's Minimum Spanning Tree*.

You have learned about the following *Networks* concepts in this activity:

* *Graphs and Subgraphs*
* *Complete Simple Graphs*
* *Random Graphs*
* *Connected Graphs*
* *Cyclic Graph*

And you also applied some *Coding* concepts that you learned about in the *Coding in Snap! Activity*, including:

* *Sequencing*
* *Repetition*
* *Variables*
* *Functions*
* *Branching*
* *Lists*

If you are interested in learning more about *Edgy*, we have included some links to some resources for learning *Edgy* on the workshop website, on the page for this session (*Creating Networks in Edgy*).