Edgy Activity - Finding MSTs in Edgy

CS4S Maths - Networks Workshop

# Introduction

In this activity, you will learn how to find *Mininum Spanning Trees* in *Graphs* using these algorithms:

* *Kruskal's*
* *Prim's*

This activity will involve combining all of the *Networks* and *Coding* concepts that you have learned about in the workshop's activities. You will also learn how to sort *edges* in a *Graph* and about *Priority Queues*, which are a *Collection* that are similar to *Lists* and *Stacks*.

# The Project

The project that you create today will be a *program* that will find a *Minimum Spanning Tree* in any *Graph*.

We will start with a *Graph* that looks like the one on the left in the image below. Note that all of the *edges* in that *Graph* are green. Then, after running the *program*, the *edges* that are part of the *Minimum Spanning Tree* will be coloured red, as shown on the right in the image below.



The finished program is available to [download as an XML file from this link](https://cs4s.github.io/math-2017/day2/msts_in_edgy/Finished%20MST%20Project.xml). Note, that *Edgy* does not currently allow you to sign in with a *Cloud Account*, so you can save the project you create today by exporting the final project as an XML file and sending it to yourself via email or putting it on a USB drive.

# Finding Minimum Spanning Trees

In this section, we will briefly remind you of what *Trees* and *Minimum Spanning Trees* are.

## Trees

*Trees* are *Graphs* that are *acyclic* (contain no *cycles*) and that are *connected* (there is at least one path from every *node* to every other *node* in the *Graph*). *Trees* have *n*-1 edges, where *n* is the number of *nodes* in the *Graph* and are *undirected* (the *edges* have no direction). We will look at *Sub-Graphs*, within a given *Graph*, that are *Trees*.

## Minimum Spanning Trees

*Minimum Spanning Trees* (MSTs) are special types of *Trees* that for a *Graph G*:

* Connect all of the *nodes* in that *Graph G*
* Contain the *edges* which produce the lowest possible total of *edge weights*, while still being a *Tree*

*Minimum Spanning Trees* are mentioned in the [MS-N1 Networks and Paths Course Content for the Mathematics Standard Year 12 Syllabus](http://syllabus.nesa.nsw.edu.au/mathematics-standard-stage6/content/1280/). The Course Content section *N1.2: Shortest paths* mentions that students should be able to:

* determine the *Minimum Spanning Tree* of a given *Network* with weighted *edges* (*ACMGM101*, *ACMGM102*)
	+ determine the *Minimum Spanning Tree* by using *Kruskal's* or *Prim's algorithms* or by inspection
	+ determine the definition of a *Tree* and a *Minimum Spanning Tree* for a given *Network*

We will implement both *algorithms* (*Kruskal's* and *Prim's*) today in *Edgy* and demonstrate how a computer can quickly find an *MST* on larger *Graphs*, where it may not be possible (or very difficult) to find an *MST* by inspection.

# Coding the MST Algorithms in Edgy

We have provided the following information for *Kruskal's* and *Prim's algorithms*:

* A high-level *Text Description* of how the *algorithm works*, which does not include any *Code*
* A *Pseudo-Code Algorithm*, which explains the *algorithm* but does not have all of the *commands* that *Edgy* has
* A *Parsons Puzzle*, where you are given the blocks for the *algorithm* and asked to rearrange them in the right order
* The completed stack of blocks as *Block Images*, which are available on the *Solutions* page for this session, on the workshop website

It is up to you decide which approach you would like to take to complete the activity. If you already fairly familar with *Coding* and *Networks* already, we would recommend that you take the *Text Description* or *Pseudo-Code Algorithm* and implement the *algorithms* in *Edgy* Otherwise, we recommend that you try the *Parsons Puzzle*, where you can Whichever option you choose, you can always compare your blocks against the *Block Images* that are on the website, or ask us for any clarifications.

You can run the *MST algorithms* against the *Graphs* created by your *create random graph* block, but we would recommend using the *build muddy city* block to create a *Graph*. That block will draw a *Graph* in *Edgy* that is similar to the one that you found the *MST* for in the *MSTS Activity* on the first day of the workshop. The *build muddy city* block has to be imported first, however. To import the block, go to this session's (*Finding MSTs in Edgy*) page and download the *Build Muddy City Block XML File* file by right clicking on the *Build Muddy City Block XML File* link, selecting *Save Link As..* and saving it somewhere easy to remember (for example: the *Desktop*). Next, go to the File menu (which looks like a blank piece of paper) in *Edgy*, click *Import...* and select the XML file you just downloaded for import. After doing this, you should have a new block in the *Network* section called *build muddy city*. When you click this block, it will create a *Graph* with 10 *nodes* and 18 *edges* (which are all coloured green).

In the following sections, we will assume that you have imported the *build muddy city* block successfully and that you will be using *Kruskal's* and *Prim's algorithms* to find a *Minimum Spanning Tree* on the *Muddy City Graph*.

# Kruskal's Algorithm

*Kruskal's Algorithm* is a method for finding an *MST* that involves choosing *edges* in a *Graph* one-by-one to add to the *MST*. The *edges* are added to the *MST* by starting from the edges with the *smallest weight*.

Before moving onto the next section, create a *variable* named *edges in mst* by clicking the *Make a variable* button in the *Variables* section. This *variable* will be used to record how many *edges* have been added to the *MST* in *Kruskal's algorithm*.

## Sorting Edges

You will need to sort the *Graph's edges* for *Kruskal's* algorithm, so you will learn about how to sort *edges* before creating the *Kruskal's algorithm*.

Firstly, create a variable named *sorted edges* through the *Make a variable* button in the *Variables* section. Next, add the following blocks to your *Snap! program*:



These blocks above set the *sorted edges* variable to a *List* of the *edges* that are sorted by their label (which is the *edge weight*). As the order is *descending* the *edges* with the larger *edge weights* will be before the *edges* with the smaller *edge weights*.

After clicking the above blocks with the *Muddy City Graph*, the *Stage Monitor* for the *sorted edges variable* should look like the image below:



Note that the *edge* (*A*,*F*) is first in the *sorted edges List* because it is one of the *edges* that has the largest *edge weight* (6).

As *Kruskal's algorithm* involves choosing the *edges* with the smallest *edge weights* first, we want to change the order in the above block to *ascending* by selecting that option from the dropdown. After changing the option to *ascending* and clicking the blocks again, the *Stage monitor* for the *sorted edges variable* should look like the image below:



Note that the *edge* (*B*,*E*) is first in the *sorted edges List* because it is one of the *edges* that has the smallest *edge weight* (2).

In the next steps, you will use the sorting of *edges* as part of the *Kruskal's algorithm*.

## Text Description

A text description of *Kruskal's algorithm* is as follows:

**Step 1:** Choose the edge of least weight.

**Step 2:** Choose from those edges remaining the edge of least weight which does not form a cycle with already chosen edges. (If there are several such edges, choose one arbitrarily.)

**Step 3:** Repeat Step 2 until n-1 edges have been chosen (where n is the number of vertices in the graph)

You are welcome to take the steps described above and create *Kruskal's algorithm* in *Edgy*. The next section explains how the *algorithm* works in *pseudo-code*.

## Pseudo-Code

In the *pseudo-code* below, the *select\_edge* adds the *edge* into the *Tree* that will become the *Minimum Spanning Tree*. For example, selecting the *edge* in the code below is the same as changing the colour of the *edge* in *Edgy* to red.

set edges\_in\_mst to 0
set sorted\_edges to sorted(edges, ascending)
for each edge in sorted\_edges {
 if (edges\_in\_mst != (number\_of\_nodes - 1))
 select\_edge(edge)
 edges\_in\_mst = edges\_in\_mst + 1
 if graph with selected\_edges is cyclic {
 unselect\_edge(edge)
 edges\_in\_mst = edges\_in\_mst - 1
 }
 }
}

## Parson's Problem



## Full Blocks

A picture that shows the blocks for *Kruskal's algorithm* arranged in the correct order is available on the *solutions* page for this workshop session. You may want to compare your final result with this picture, to check if you have arranged the blocks in the right order.

# Prim's Algorithm

Before moving onto the next section, create a *variable* named *nodes in mst* by clicking the *Make a variable* button in the *Variables* section. This *variable* will be used to record how many *edges* have been added to the *MST* in *Kruskal's algorithm*.

## Priority Queues

*Priority queues* are *Collections* of items that are similar to *Stacks*. Like *Stacks*, *Priority queues* have a *Function* that gives us the first item in that *Collection*, but they work a bit differently. There are two types of *Priority queues*: *maximum* and *minimum*. We will focus on *minimum priority queues* in this activity.

*Minimum priority queues* have three main ways of accessing and modifying items:

* *enqueue item with priority*: we put the item into the *queue* and give it a *priority* value. For example: in *Prim's* algorithm, we will put *edges* in a *priority queue* and each *edge's weight* will be the *priority*. In *Edgy*, the block that does this is called *enqueue item to pqueue with priority*.
* *head of priority queue*: we take the item with the *priority* of lowest value and have a look at what it is. In *Edgy*, the block that does this is called *head of pqueue*.
* *dequeue from priority queue*: we remove the item with the *priority* of lowest value from the queue and discard it. In *Edgy*, the block that does this is called *dequeue from pqueue*.

To demonstrate this, use the following blocks below to create a *Graph* with 3 *nodes* and two *edges*:



Note that the shortest *edge* is the one between *A* and *C*.

Next, create a variable called *candidate edges*, which we will make a *minimum priority queue* with the *min pqueue* block.

Use the following blocks to make the *candidate edges variable* a *minimum pqueue* and *enqueue* the *edges*.



After these steps, the *candidate edges variable* would look similar to the table shown below:

|  |  |
| --- | --- |
| **Edge** | **Priority** |
| (A,C) | 5 |
| (A,B) | 10 |

Note that because the *edge* between *A* and *C* has the smallest *priority*, it is the first item in the queue.

You could then use the following block to *dequeue* from the *candidate edges*:



The *candidate edges variable* would then look similar to the table shown below:

|  |  |
| --- | --- |
| **Edge** | **Priority** |
| (A,B) | 10 |

In *Prim's algorithm*, we look for the shortest *edge* that is not currently in the *MST* when adding *eges*. Consequently, using a *minimum priority queue* allows us to select the shortest available edge and add it to the *MST*.

## Text Description

A text description of *Prim's algorithm* is as follows. Note that the *T* described in the steps below is the *Sub-Graph* that contains the *MST*. When the *algorithm* is completed, *T* will contain all the *edges* and *nodes* of the *MST*.

**Step 1:** Select a *node* to be the first *node* of *T*.

**Step 2:** Consider the *edges* which connect *nodes* in *T* to *nodes* outside T. Pick the one with minimum weight. Add this *edge* and the extra *node* to *T*. (If there are two or more *edges* of minimum weight, choose any one of them.)

**Step 3:** Repeat Step 2 until *T* contains every *node* of the *Graph*.

## Pseudo-Code

In the pseudo-code below, *mst\_nodes* will be similar to *T* in the *Text Description* for *Prim's algorithm*. The *mst\_nodes variable* will be a *List* of the *nodes* that are in the *MST*.

set mst\_nodes to empty list
set candidate\_edges to empty minimum priority queue
add node 1 to mst\_nodes
for each edge of (edges of node1) {
 enqueue edge to candidate\_edges with priority of (weight of (edge))
}
repeat until (length of mst\_nodes = number of nodes) {
 set shortest\_candidate\_edge to (head\_of\_pqueue())
 dequeue from candidate\_edges
 set start\_node to start\_node\_of(shortest\_candidate\_edge)
 set end\_node to end\_node\_of
 if (mst\_nodes.contains(start\_node) and !(mst\_nodes.contains(end\_node)) {
 add end\_node to mst\_nodes
 select\_edge(shortest\_candidate\_edge)
 for each new\_candidate\_edge of (edges of end\_node) {
 new\_candidate\_edge to candidate\_edges with priority of (weight of (new\_candidate\_edge))
 }
 }

}

## Parsons' Problem

We have separated this section into two sub-sections.

The first sub-section are the blocks that will be arranged to perform the initial steps of *Prim's algorithm* (in the *Pseudo-Code* above, this is the steps from the start of the *pseudo-code* until the *repeat until* command). The initial steps of the *Prim's algorithm* involve adding the first *node* in the *Graph* to a *List* and then adding all of that *node's neighbours* to the *priority queue* called *candidate edges* with their *edge weight* as the priority.

The second sub-section are the blocks that add the *nodes* to the *mst nodes List*, as we select new *edges* to be part of the *MST*. In the *Pseudo-Code* above, these are the steps from the *repeat until* command until the end of the *pseudo-code*.

### Initial Steps

These are the blocks to be used for the initial steps of *Prim's algorithm*.



### Adding Nodes and Edges

These are the blocks to be used for selecting the *nodes* and *edges* in the *MST* during *Prim's algorithm*. These will be added after the blocks you created for the initial steps of the *Prim's algorithm*.

Note that you will need to create a *variable* called *shortest candidate edge* for this section.



## Full Blocks

A picture that shows the blocks for *Prim's algorithm* arranged in the correct order is available on the *solutions* page for this workshop session. You may want to compare your final result with this picture, to check if you have arranged the blocks in the right order.

# Extension Activity: Reverse Deletion

If you complete the *Prim's* and *Kruskal's algorithms* in *Edgy* before the end of the session and would like to learn about other approaches for finding *MSTs*, we recommend that you try to implement the *Reverse Deletion algorithm* in *Edgy*.

*Reverse Deletion* is an approach for finding an *MST* in a *Graph* where all of the *edges* are included in the *MST* to begin with. Edges are removed from the *MST* one-by-one in order from the *edges* with *largest edge weights* to the *edges* with the *smallest edge weights*. Each time an *edge* is removed from the *MST*, the *algorithm* checks whether removing the *edge* causes the *MST* to become *disconnected*, and if so, the *edge* is added back in to the *MST*. This approach is almost an opposite of the *Kruskal's algorithm*, as we are removing *edges* to the *MST*, instead of adding *edges*.

An image with the stack of blocks that implements the *Reverse Deletion algorithm* is also available on the *Solutions* page for this workshop session.

# Conclusion

In this activity, you have learned how to find *Mininum Spanning Trees* in *Graphs* using these algorithms:

* *Kruskal's*
* *Prim's*

This activity combined all of the *Networks* and *Coding* concepts that you have learned about in the workshop's activities. You also learned how to sort *edges* in a *Graph* and about *Priority Queues*, which are a *Collection* that are similar to *Lists* and *Stacks*.

The finished program is available to [download as an XML file from this link](https://cs4s.github.io/math-2017/day2/msts_in_edgy/Finished%20MST%20Project.xml) for you to refer to after the workshop.