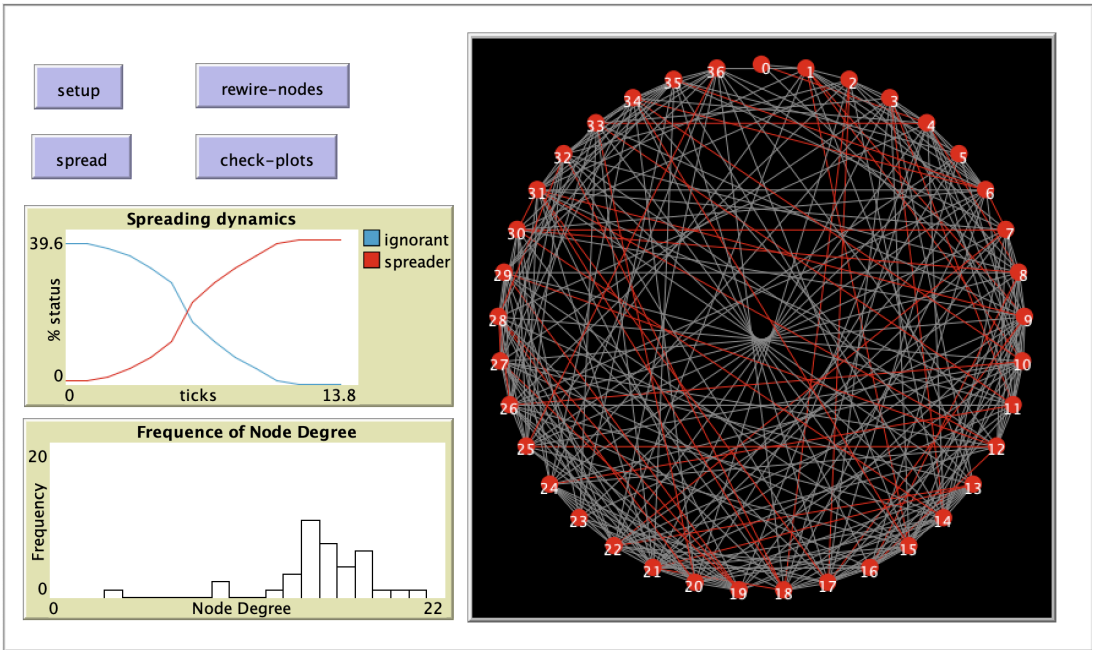
CS108L Computer Science for All
Module 8: Guide Networks Milestone 2
How Rumors Spread



In this module we will be creating a small world network that shows how a rumor might spread amongst individuals of the network. To do this we will formalize our understanding of arguments to a procedure, how to pass variables to that method, what a reporter is and how to open and read from text files, and NetLogo's behavior space which simplifies running some experiments in NetLogo.

Small World Networks

Have you ever had the power go out in your home? Chances are, a lot of your neighbors had their power go out too. Why does this happen?

A power grid is a small-world network!

Recall from the previous lesson that *in a small world network, the path from any given node any other node will be relatively short.* That means that if your neighbor's power line is down, yours is likely to go down too. We can look at a power outage as a type of information. We can look at

a power outage as a type of information spreading on a network: in this case, the information takes the form of failure.

We can also model these phenomena (and many more!) as information spreading:

- a computer virus
- the brain's response to stimuli
- a disease epidemic

How to do it the NetLogo way.

Procedure Arguments in NetLogo.

Arguments are local variables to a procedure. However, unlike using the 'let' command the argument is not set to a value within the procedure, rather the values are assigned when the procedure is called. This is called passing by value (note there is another way to pass called pass by reference, but NetLogo does not use this method, it is quite common in other languages though). In this way you may use the same procedure throughout your program with different input values.

You might want to change the color of a specific turtle, but it might not always be the same turtle. To do that, you need an input or argument to a procedure that changes the turtles color. Something like this;

```
to changeTurtleColor [id]
  ask turtle id [
    set color red
  ]
end
```

The argument is 'id' which is a variable and thus contains a value. If 'id' is assigned the value of 1, then turtle 1's color will become red, if it is 23 then turtle 23's color will turn red. Naming of the argument is the same as naming of any other variable, pick something representative of what it is.

To call the procedure, you need to supply both the name and the value that will be assigned to the argument. Following are examples above the code would look like this;

```
changeTurtleColor 1
changeTurtleColor 23
```

Note that if you call a procedure with an inappropriate value for the argument, you will get an error just as if you had written it directly into the procedure.

A procedure may have as many arguments as you need. With the above procedure you might also want to make it work for any color. In that case just add another argument like so;

```
to changeTurtleColor [id newColor]
  ask turtle id [
    set color newColor
  ]
end
```

You must specify the value of all arguments when you call a procedure or you will get an error. With the above procedure if I wanted turtle 1 to be red and turtle 23 to be green it would look like this;

```
changeTurtleColor 1 red
changeTurtleColor 23 green
```

Reporters in NetLogo.

A reporter is a procedure that returns a value. In all languages you need to specify that a procedure is going to return a value. NetLogo does this by using the 'to-report' to mark the procedure (instead of 'to'). You also need to tell the procedure what the value you are returning is. You do this by using the keyword 'report'. Any value after the report is what is returned. So if we wanted to calculate the area of a square and return that value we could use this procedure;

```
to-report areaOfSquare [side]

  report side * side

end
```

Since the reporter is returning a value, anywhere you see a procedure call using the reporter you can imagine it being replaced with the value it reported. This means you may make a call to the reporter in another equation, to assign the reported value to a variable or just print what you got. In all ways it is treated as that value.

An example of this using the procedure above might be;

```
let sqArea (areaOfSquare 2)

let totalArea (areaOfSquare 2 + areaOfSquare 10 + areaOfSquare 5)

print (areaOfSquare 2 + areaOfSquare 10 + areaOfSquare 5)
```

The first line assigns the local variable sqArea to 4 (2 * 2). The second line assigns the local variable totalArea to 4 + 100 + 25 or 129. And the last line just prints the number 129. Note that while this is done with local variables and the let command, this works when using the set command as well so it works with existing local variables, global variables and agent variables.

Using Files in NetLogo.

It is often convenient to store information that a program uses in a file. This is exactly what happens when you are using Word or a similar program. You write what you want on screen, then hit 'save' and a file is created (if it doesn't exist) or appended (if it does exist). When you open that file again, the program reads the file and loads the information contained in it to the Word program which shows you what is in the file.

We will only be working with reading a file and loading it to our NetLogo program in this section but NetLogo contains all the commands necessary to create and print to files as well.

When program reads a file, it doesn't just read it's file name but also the file extension. The file extension is the letters after the last '.' in the file name and tell what type of file it is. The file extension is generally hidden from our view (all operating systems have a method of showing file extensions, just google how to see your file extensions with the operating system you are using if you need to see them). We will be working with a text file, so the extension is '.txt'. So when working with a text file called 'test' the program will need to know the file is called "test.txt" which is written as a string (so use quotes).

Another important item in finding a file is the file path. This is the path from the top folder (or directory) to the folder that contains the file. So the file path 'C:\\Afolder\\Folder2\\test.txt' (note

in some languages you do not need two '\' only 1) has the program look on the C drive in Afolder then the subfolder Folder2 for the file test.txt. If the file is not there it returns a 'file not found' error. There are convenient ways to navigate between folders and we will use the most convenient. By storing a file in the same folder as the NetLogo program the only filepath you need is the name of the file (in this case 'test.txt'). So make sure you store your file in the same folder as the NetLogo program!

You may check if your program can find the file by using the file-exist? Boolean reporter. This just returns true if it finds the file and false otherwise. It is common create an ifelse where the true portion works on the file if it exists and the false portion reports you did not find a file.

```
ifelse file-exists? filename
[
  ;;work with file
]
[
  print "File not found"
]
```

This prevents you from relying on the default error message (in some languages these are not helpful) and allows the program to not stop if the file is not found.

--File open/close

To work with a file you must get access to the file. Files have certain protocols for accessing them but in NetLogo we can just use the file-open command.

```
let filename "test.txt"
file-open filename
```

The first line assigns the local variable filename to the name of our file 'test.txt' then we open the file with the file-open command and the name of the file contained in the variable filename (note that using a variable instead of just putting file-open "test.txt" is the standard way, if you used file many times and needed to change the filename, you would only change it once at the variable rather than everywhere).

It is important to understand that once a file has been opened by a program that file may be locked (no other program can use it). To release the lock you must tell your program you are finished using it. You do this with the file-close command or file-close-all (if working with multiple files).

```
file-close
```

Note: You may open another file while still having a file open using file-open. When doing this, if you use file-close you will close the last file you opened first. Your program should always close all file prior to stopping.

Note too: That once you open a file, NetLogo keeps track of where you are in the file, so if you read line 1 in procedure A then move to procedure B, and read a line again, you will read line 2. You can only restart if you use file-close then open the file again.

--Reading from a file
Now that we can open and close files we will learn to read from them.

The first and very important thing to note when reading from a file is the file structure. Generally, a simple text file will be just filled with numbers, letters and delimiters (things like commas, spaces, semicolons, etc..). It will not tell you what any of the letters or numbers mean (this is because the file is designed for a computer not a human to read). So you will need to find out. For this class, the assignment will contain the information necessary to understand the file structure.

There are many ways to read a file in the NetLogo dictionary, but we will focus on just one, file-read-line. This reads all characters (numbers, letters, spaces, delimiters, etc..) on a line in one swoop and puts them in a string. Keep in mind a string is considered a word not a number so "1" is not 1. If we are working with something other than a string, we need to convert to that. In NetLogo we can use the command read-from-string. This command allows NetLogo to make an 'educated' guess at what the string is and convert it. So if the string is "10" it will convert it to the number 10. Doing this allows us to use the value just like any number.

```
let number read-from-string file-read-line
create-turtles read-from-string file-read-line
```

The first line sets the variable number to the value of the string it read from a line of the open file. The second line creates the number of turtles that it read from a line of the open file. If there is only one thing on a line this works well. However, if you have several things divided by a space, you need to do more work.

One way of doing this is to read each word (remember in a text file everything is a word even a number) in the text file on a line into a list of words. In the case where the items in the text are separated by a space the way of doing this is straight forward.


```
word "[" file-read-line "]")
```

In the above the command 'word' means concatenate (combine) 2 or more strings together. The strings in this instance are "[", file-read-line, and "]". For example, say you have the values 1 2 on a line in a text file. Then what happens is "[" is added to 1 2 to produce "[1 2" then "]" is added to that producing "[1 2]".

At this point the 1 and 2 are still a string so we need to use the command read-from-string to change them into a number and we can assign the results to a variable and now that variable contains a list of the numbers! Combining everything gives us the below code.

```
let items read-from-string (word "[" file-read-line "]")
```

Again, if the text line we are reading is 1 2 then the above is the same as;

```
let items [1 2]
```

Now that you have a list, you can access the items in a list by using the command 'item'. Like turtles, the first item in a list is item 0.

So if we wanted to use the above list to reference different turtle ids and ask them to do something I could say;

```
ask turtle (item 0 items) [
  ;have the turtle do something
]
```

The code above pulls the first item in a list (item 0) from the list items, which in this case is 1. So the above code asks turtle 1 to do some activity.

Now we know how to pull information of a line on the text file, but what we also want to automatically read all the lines of the that file. We already have a way to loop, we can use a while loop. But to do this we need to tell the while loop when to stop. Fortunately there is a handy NetLogo command just for something like this called file-at-end?. This command is true if you are at the end of the file and false otherwise. Now we want to keep reading while we are not at the end of the file so our stopping command becomes [not file-at-end?] which is true until we are at the end of the file (remember a while loop runs until the condition is false). This is shown in the code on the next page;

```
while [not file-at-end?] [
  ;...read next line
  ;...do something with the line
]
```

When we combine all the above with the while loop we can read each line of the file without setting up a line of code to read each line (if there were 1000 lines in a text file this is a great saving in time). Keep in mind, you want to do this when all the lines are the same, if your first line is something else (like number of turtles) you will want to handle that first then do all the links with a while loop.

Plotting Histograms in NetLogo

A histogram is a plot that tells the frequency of events broken into bins. If you roll a die, you may get a number from 1 to 6. In a histogram, each time you roll a 1 you but a increase the value of the bin by 1 (starting from 0). So if you rolled a 1, 4 times count in that bin would be 4. You do this for each number, so if you rolled the die 20 times you might have 4 1s, 3 2s, 7 3s, 2 4s, 3 5s and 1 6s. The y-axis becomes the number (or frequency) you rolled a particular number and the x-axis is the bin, which is the number you could roll on the die. The resulting graph is a histogram (note, that the bin size or range of values that fit in one bin matters for the shape of the graph, we could have had bins of 1-2, 3-4 and 5-6 for instance).

To do a histogram in NetLogo instead of typing plot, you type histogram.

```
histogram [count link-neighbors] of nodes
```

Sometimes you want to set the plots axes or even have it change. To do this, in the plot update commands section of the graph (edit the graph and click on it). You can type;

```
set-plot-x-range 0 (max [count link-neighbors] of nodes + 2)
set-plot-y-range 0 20
```

The first line sets the minimum of the axis to 0 and the max to the highest degree of nodes + 2. The +2 just keeps the graph from bumping up against the right edge of the plot. This changes if the max number of degrees changes. Note that the plot automatically sets up the bins. The second line sets the minimum y-axis to 0 and max to 20.

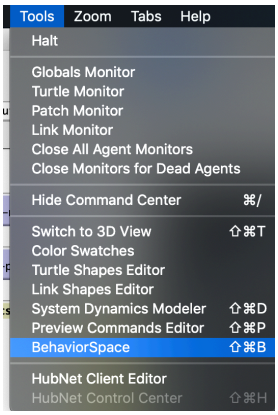
You may update a plot while running your program by using the update-plots command.

```
update-plots
```

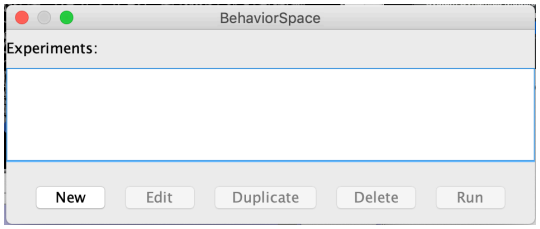
BehaviorSpace

BehaviorSpace is a convenient NetLogo built-in tool for running experiments. It allows you to change the values of variable without actually changing them in code and run the program again automatically. Then outputs a .csv (common delimited) file that a spreadsheet program like excel can read.

To use BehaviorSpace you need to setup it up first. Go to the Tools bar scroll down to 'BehaviorSpace' and select it.

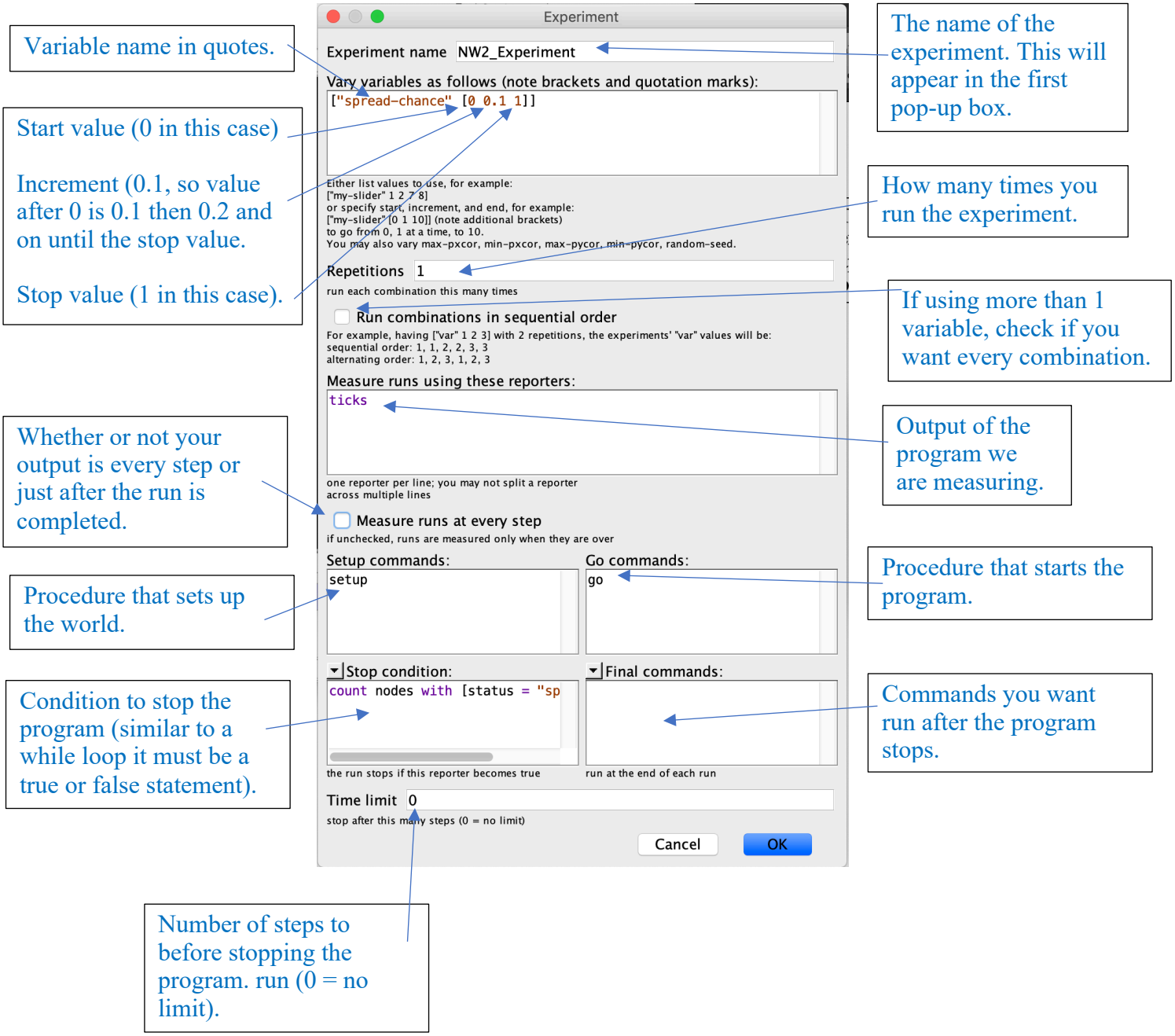


This brings up a pop-up window. Choose 'New' if you want to create a new BehaviorSpace or select 'Edit' is you want to modify an existing one.



This will open another pop-up window for setting up BehaviorSpace . Press Okay when done. An image of this pop-up window is on the next page with a more detail description of how to fill it out. If you need more details on BehaviorSpace, it can be found on the NetLogo website here: (https://ccl.northwestern.edu/netlogo/docs/behaviorspace.html)

To run BehaviorSpace, once you have it set up, simply select the one you want from the first pop-up above and press run then select your output file (.csv is a good one) and file name you are saving it under.


Experiment
Experiment name NW2_Experiment
Vary variables as follows (note brackets and quotation marks):
["spread-chance" [0 0.1 1]]
Either list values to use, for example:
["my-slider" 1 2 7 8]
or specify start, increment, and end, for example:
["my-slider" [0 1 10]] (note additional brackets)
to go from 0, 1 at a time, to 10.
You may also vary max-pxcor, min-pxcor, max-pycor, min-pycor, random-seed.
Repetitions 1
run each combination this many times
Run combinations in sequential order
For example, having ["var" 1 2 3] with 2 repetitions, the experiments' "var" values will be:
sequential order: 1, 1, 2, 2, 3, 3
alternating order: 1, 2, 3, 1, 2, 3
Measure runs using these reporters:
ticks
one reporter per line; you may not split a reporter
across multiple lines
Measure runs at every step
if unchecked, runs are measured only when they are over
Setup commands:
setup
Go commands:
go
Stop condition:
count nodes with [status = "sp
the run stops if this reporter becomes true
Final commands:
run at the end of each run
Time limit 0
stop after this many steps (0 = no limit)
Cancel
OK
Variable name in quotes.
Start value (0 in this case)
Increment (0.1, so value after 0 is 0.1 then 0.2 and on until the stop value.
Stop value (1 in this case).
Whether or not your output is every step or just after the run is completed.
Procedure that sets up the world.
Condition to stop the program (similar to a while loop it must be a true or false statement).
Number of steps to before stopping the program. run (0 = no limit).
The name of the experiment. This will appear in the first pop-up box.
How many times you run the experiment.
If using more than 1 variable, check if you want every combination.
Output of the program we are measuring.
Procedure that starts the program.
Commands you want run after the program stops.

The output of the file for the above settings and for the program run (note this might not be the same program as the one you are working on) looks like this (in excel);

BehaviorSpace results (NetLogo 6.0.1)											
NW2_v1.nlogo											
NW2_Experiment											
04/18/2019 16:36:43:379 -0600											
min-pxcor	max-pxcor	min-pycor	max-pycor								
-16	16	-16	16								
[run number]	1	2	3	4	5	6	7	8	9	10	11
spread-chance	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
[steps]	0	0	0	0	0	0	0	0	0	0	0
[initial & final values]	ticks	ticks	ticks	ticks	ticks	ticks	ticks	ticks	ticks	ticks	ticks
	9	16	8	11	9	13	9	8	18	8	10

Most items are self-explanatory but '**run number**' is how many different variable settings there were. If you had more than 1 repetitions run number is the # of repetitions times the # of different variable settings, so the columns will keep going.

'**spread-chance**' is the variable we are examining and the values to the right are the settings the program used for this variable.

'**ticks**' are our output value and the number below is the number of ticks it takes to get to our stopping criteria. Depending on your experiments you could have 1 or more different outputs (for instance if you turtle count is changing you could have 'count turtles').