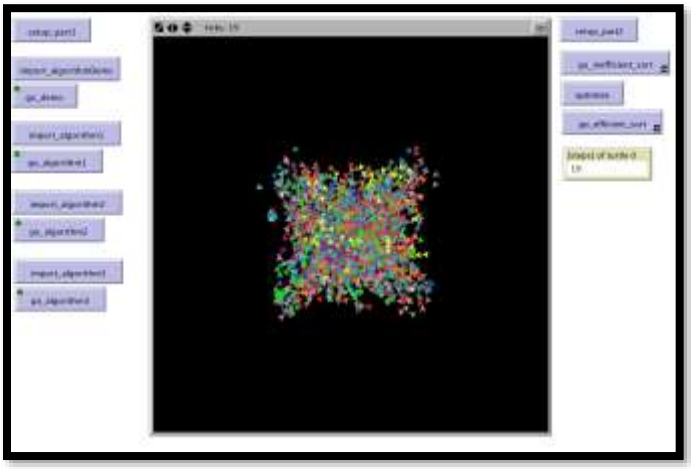
CS108L Computer Science for All
Module 7: Algorithms

Part 1: Patch Destroyer

Part 2: ColorSort



Part 1 Patch Destroyer Model Overview:

Your mission for Part 1 is to get your turtle to destroy the green patches as efficiently as possible. You will download the base model and some image files. The image files create different patterns in green patches in the model. You need to create procedures that move your turtle around the NetLogo world **one patch at a time** destroying the green patches (turning them black) as it goes. The object of the assignment is to do so in the fewest steps possible, while still following the rules listed below.

NOTE: when this model runs, it requires image files (algorithmDemo.png, algorithm1.png, algorithm2.png, algorithm3.png) and the base model. **You must download the images and base program from the link on the class website. The images must be in the same folder as your program for it to work.**

The way you run the code for each image is:

- Hit the setup_part1 button
- Hit the import_algorithm# button for that image
- Hit the go_algorithm# button for that image

Now try the demo to make sure that your base model is working properly:

- Hit setup_part1
- Hit import_algorithmDemo
- Hit go_demo

Your turtle should destroy the straight line of patches. Now it's your turn to write your own algorithms for the images algorithm1.png, algorithm2.png, and algorithm3.png!

The Rules:

1) You will start with the base model (program) provided. **You must download the images and base program from the link on the class website. The images must be in the same folder as your program for it to work.** The base model includes the procedures that import the file images as well as a demo. **DO NOT EDIT THESE PROCEDURES. Please save the base program under a different name (W9.*firstname.lastname*.nlogo)**
2) Your NetLogo world needs to be wrapped with a max-pxcor and a max-pycor of 20 (as already provided in the base model). **Do not change the dimensions**. If you change the dimensions of the NetLogo world, the images may not import correctly!
3) The base model also includes a setup_part1 procedure. You may edit this procedure. However, **the turtle must start at the center of the NetLogo world and be facing up (heading 0) and be a fixed color that is NOT green or black** (otherwise you might not see your turtle!).
4) **Your turtle can only move ONE patch at a time**. You cannot use a different command (such as setxy) to move the turtle to another patch.
5) You must count the total number of steps that your turtle takes to munch the patches. That means that you must:
    a. **Create a turtles-own variable** to keep track of the number of steps that your turtle takes. You can call it whatever you want!
    b. Initialize this variable to 0 (zero) in the setup procedure.
    c. Increment this variable **every time** your turtle moves forward 1 step. So your code would look something like this if your variable was called **steps:**

```
forward 1
set steps steps + 1
```

6) You must create **a monitor** to report the number of steps your turtle takes.

```
[steps] of turtle 0
```

7) You must create a **go procedure** for each of the imported images you solve:
    a. You are trying to develop an algorithm to solve each problem as efficiently as possible.
    b. Each solution must include a repeat or while loop to make it more efficient.
    c. Minimize the steps your turtle takes to munch the patches! **Remember, your turtle can only take one step at a time and you must count each step.**
8) Your program should work like this for each image:
    a. **First: Click** on the setup_part1 procedure to clear your world and setup your turtle to be in the right place, starting in the right direction and be the color you select (but not green or black)
    b. **Second: Click** on the import_algorithm_# procedure to import the image you are solving.
    c. **Third: Click** on the go_algorithm# procedure for that imported image to destroy the patches!

Hints:

1) A random walk or wiggle walk will NOT be an improvement in the number of steps (it is likely to take 1000s of steps).
2) You may have to backtrack over patches that your turtle has already munched to completely finish the munching process – that is OK. You still MUST count all the steps even the ones that do not turn green patches to black patches!
3) **It is easier to program the go procedure if you make it a TURTLE procedure instead of an observer procedure.**

Part 2 ColorSort Model Overview:

We learned about sorting algorithms from the videos this week. We also learned that algorithms can be efficient or inefficient. In Part 2, we will sort turtles 2 ways: an efficient way and an inefficient way. The ticks taken and result will be the same, but you will notice that the efficient way takes much less time to sort!

First, we'll create our turtles and give them some values.

Then, we'll sort inefficiently. On each tick, we ask turtles to move towards another turtle that has the same color. So every time through, we are creating a set of turtles with the same color. (This is a forever button)

Next, we'll optimize how they sort. Let's make that set of turtles with the same color *just once* in our setup. Then we don't have to do it every time!

Last, we'll sort efficiently, using the set of turtles we created at the start. (This is a forever button)

Do the following first:

1) Turn on world-wrapping if you haven't already.

2) You should already have a turtles-own variable from part 1. Now, add one more variables in the turtles-own section:
   - create a variable that will store a list of other turtles with the same color, like **turtlesWithMyColor**

Setup:

Make a setup_part2 button and procedure. setup_part2 is simple: all it does is clear the world, reset ticks, create 2000 turtles, and set those turtles to random locations.

Inefficient Method Procedure:

The first color sorting method you will be trying is the Inefficient Method. The inefficient method is successful in getting all the turtles sorted by color but it does so slowly. The algorithm for the inefficient method is:

a) Each turtle has a color. You should create a **local** variable to keep track of that color for you. (let MyColor color)
b) Each turtle looks at the other 3999 turtles one at a time to see what color it is. If the other turtle has the same color (MyColor) then it is put into a NetLogo **agentset** (a list or set of agents). An **agentset** lets you keep track of a group of agents so you can use that group later. See the Hint below for more help.
c) The turtle then randomly chooses one of the other turtles of the same color (in the created agentset) and names it something, say **myTarget**.
d) The turtle turns to face the turtle chosen in step (c) (set heading towards **myTarget**)
e) The turtle takes one patch size step forward.

How to do Inefficient Method Steps b and c:

The simplest way to do steps b and c of the inefficient method is to chain a few of NetLogo's reporters together: In NetLogo, **commands** and **reporters** tell agents what to do. Rememebr: A **command** is an action for an agent to carry out, resulting in some effect. A **reporter** is instructions for computing a value, which the agent then "reports" to whoever asked it. Here are some of the **reporters** you might use:

one-of *agentset*	Given an agentset as input, **one-of**selects and reports a random agent from that set. If the agentset is empty, the **one-of**reports **nobody**.
other *agentset*	Netlogo's **other** reporter is used to report a new agentset that is the same as the agentset it is given

	except with "this" turtle (the turtle in the current iteration of **ask turtles**) removed. The **other** reporter is used in this model because we want each turtle to move toward a turtle of the same color, BUT NOT to pick itself as the turtle to move toward.
turtles	Reports the agentset consisting of all turtles.
agentset **with[***reporter***]**	The **with**reporter takes two inputs: on the left, an agentset (usually "turtles" or "patches"). On the right, the reporter must be a boolean reporter. Given these inputs, **with** reports a new agentset containing only those agents that reported true -- in other words, the agents satisfying the given condition.

Putting this all together, we can build the powerful NetLogo statement:

let myTarget one-of other turtles with [color = MyColor]

The statement above reads from turtles in the middle to outer edges as:

turtles: the set of all turtles,

turtles with [color = MyColor]: Look for the turtles that have **color** equal to **MyColor** and group them in an agentset. Note that **color** is the color of each turtle in the agentset and **MyColor** is a local variable that needs to be defined as the color of "this" turtle before you use it in this NetLogo statement.

other: Remove "this" turtle (the turtle that is performing the action) from the agentset reported by **with [color = MyColor]**.

other turtles with [color = MyColor]: The agentset containing all the other turtles with color = MyColor

one-of: Pick and Report a random agent form the agentset that was create and reported by **other**.

let myTarget:assign the agent returned by **one-of**: to the local variable **myTarget**.

Optimize Procedure:

In the inefficient method, for **every** tick, **every** turtle builds an agentset of all turtles that share its color, then the turtle picks a random element of that agentset to turn towards.

The key to making the Inefficient program more efficient is noticing that in the Inefficient program the turtle is creating an agent set and randomly picks another turtle **EVERY TICK. BUT,** each turtle's agentset of all turtles with the same color **NEVER CHANGES**! So, for each turtle we can build an agentset of turtles with the same color just ONE TIME. Then each turtle can look at their already made agentset each time it picks a target to move towards. Here's how you can do that:

1) Declare a new **turtles-own** variable (I called it **agentsWithMyColor**) to store each turtle's agentset of like colored turtles.
2) In the **Optimize**procedure, build the agentset **agentsWithMyColor**:
   - Create the *local* variable **myColor** and store the turtle's color in it.
   - Set the agent variable **agentsWithMyColor** to the agentset containing all the **other** turtles with the same color (color = myColor).

Efficient Procedure:

In the Efficient procedure, pick a random member from the agentset.

- Create the *local* variable **myColor** and store the turtle's color in it.
- Pick your target from the agents list **agentsWithMyColor,** which you created in the Optimize procedure.
- The turtle turns to face the turtle chosen.
- The turtle takes one patch size step forward.

Try It Out!

- Click your setup_part2 button.
- Click your inefficient sort button, and watch for a few ticks. Click it again to stop the sort. (This is a forver button.)
- Click your Optimize button.
- Now, click your efficient sort button. The turtles should move much more quickly.

When you get this to work, you will see that it saves lots of time. For one turtle to create its agentset of like colored turtles, that turtle must examine the color of all other turtles. Thus, the more turtles there are, the longer it takes for ONE turtle to build its agentset. Let n be the number of turtles (in this lab, n = 2000). Since EVERY turtle must build its own agentset, the total time to build an agentset is proportional to the time it takes one turtle to build its agentset, O(n) times the number of turtles that build lists, also O(n). The whole process then takes $O(n^2)$ computational time (in this case about 16 million steps each tick)! In the inefficient method, this is done every tick and when you use the Efficient Method with the Optimize procedure, it happens only once.

THE UNIVERSITY of NEW MEXICO

NOTE: Sets versus Lists:

In Netlogo, there are things called **sets** and different things called **lists**. An **agentset** is a set that contains only agents. In casual English, the words set and list are often used interchangeably. In computer science, however, these words have very different meanings.

A **set** is an **unordered collection** where any **repeated elements make no difference** to the set. For example, the sets: {1, 2, 3, 4} and {4, 2, 1, 3} are the same. Also, if 2 is added to the set {1, 2, 3, 4}, then the resulting set is still {1, 2, 3, 4} since 2 was already an element of the set. With a set, it makes no sense to ask "what is the first element" since none of the elements have any particular order.

A **list** is an **ordered collection** where **repeated elements DO make a difference**. Thus, the lists, [1, 2, 3, 4] and [4, 2, 1, 3], are different. If 2 is added to the list, [1, 2, 3, 4], it is important to ask **where** the 2 is added because the lists [2, 1, 2, 3, 4], [1, 2, 2, 3, 4], [1, 2, 3, 4, 2], ... are each different from one another.