SWARMATHON 3

INTRO TO DETERMINISTIC SEARCH

nasaswarmathon.com

1 SWARM ROBOTS ON MARS

In Swarmathon 1 and 2, we examined biologically-inspired search techniques that employed randomness. In Swarmathon 3, we'll add a new kind of technique to our toolbox: a deterministic search.

1.1 WHAT IS DETERMINISTIC SEARCH?

We can call a search strategy *deterministic* if for some **A**, the strategy searches **A** for a condition or element **x** in

order until **x** is found or the end of **A** is reached (Cormen et al., *Introduction to Algorithms 3rd ed.*). This is very different from the wiggle walk/correlated random walk that the robots in Swarmathon 1 and 2 used.

1.2 REVIEW OF SWARMATHON 1 WALK STRATEGIES

Please review the table on p.14 of [Sw1]. Note that the third type of walk, ballistic motion, is not a random walk. It is a *deterministic* walk. Given a path **A**, the agent searches **A** for resource **x** until the end of the path **A** is reached.

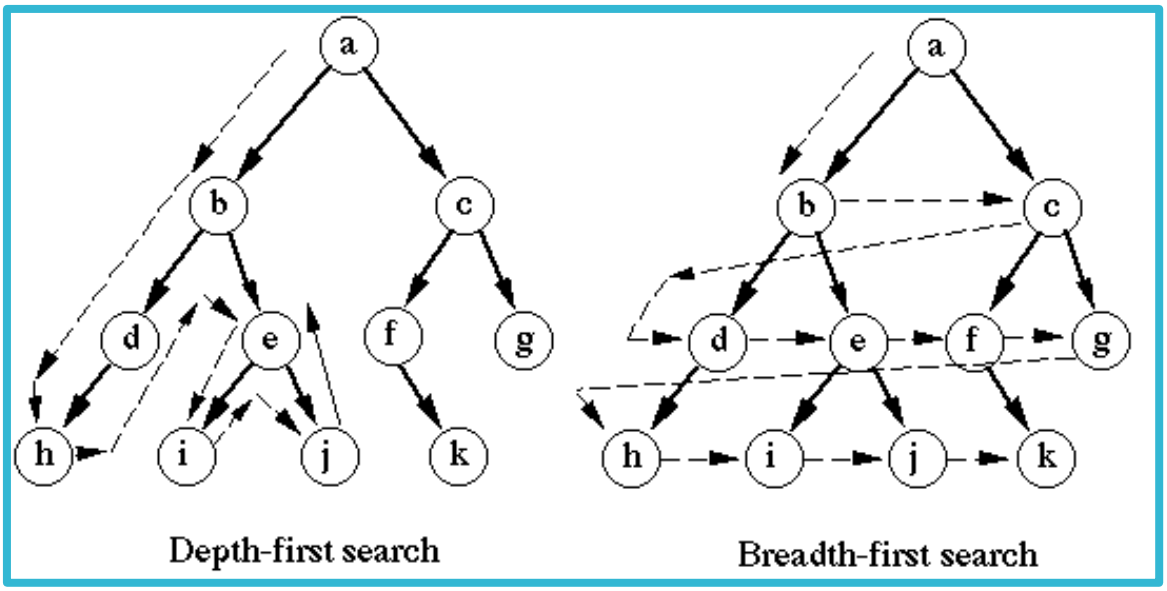
We will implement ballistic motion in the robots in Swarmathon 3 by having them travel in a straight line (**A**), storing the locations of resources they have seen (**x**) in a list, until they reach the edge of the arena (end of the path **A**). The robots will then process the list of **x**'s until all resources seen on that line have been collected.

1.3 RELATIONSHIP TO DEPTH-FIRST SEARCH

The search strategy that we just described has a strong relationship to a standard search algorithm in computer science called Depth-First Search (**DFS**). As the name suggests, the algorithm goes as deeply as it can into a data structure before backtracking.

DFS contrasts with another standard search algorithm, Breadth-First Search (**BFS**). **BFS** explores all local options before expanding in depth. See the picture on the following page to get an idea of what these search algorithms look like in action on a data structure.

While there is some room for debate as to what **DFS** looks like in a 2-D world (such as our simulation), from now on we'll refer to our *deterministic* strategy as **DFS**. We'll take a look at a strategy related to **BFS** in the next module, Swarmathon 4.



GitHub user tinkerpop

2 ROBOT CONFIGURATION

2.1 FILE SETUP

As in Swarmathon 1 and 2, we will be using NetLogo base code. Open the file *[Sw3]IntroDetSearchstudentCode.nlogo* and rename it *yourlastname_Swarmathon3.nlogo*.

2.2 ROBOT PROPERTIES SETUP

What do our robots need to know to do DFS? Recall from Section **1.2** that we want our robots to perform the following behaviors:

- Travel in a straight line until they reach the edge of the arena.
- Store the locations of resources they have seen while traveling on that line in a list.
- After reaching the edge of the arena, process the list of resources until all resources seen on that line have been collected.

Robots can easily travel in a straight line by simply taking the wiggle out of their walk. We should store the angle they are traveling at. But how about storing locations of resources? Let's give each robot its own list. When it encounters a rock while traveling to the edge of the arena, let's have the robot add the coordinates of the rock to its personal list. Then, when it hits the edge, it can go back to the locations on its list one by one and gather the resources.

WHAT DOES A SWARMIE SEE?

Swarmie robots are equipped with a camera (circled) and cannot see long distances. Since Swarmies only know what's right in front of them, they must use local information to gather resources efficiently.

We've identified that each robot needs a personal list, current heading, as well as a target x and y coordinate to head towards when it is processing the list. Next, we should decide what states a robot can be in.

Let's give the Swarmies two special states: processingList? (for the behavior we described above) and returning? (for dropping off rocks at the base). Let's assume that if they are not doing either of these things, they are doing DFS.

Scroll to the top of the code and fill in the robots-own section as in the picture below. Be sure to read the comments.

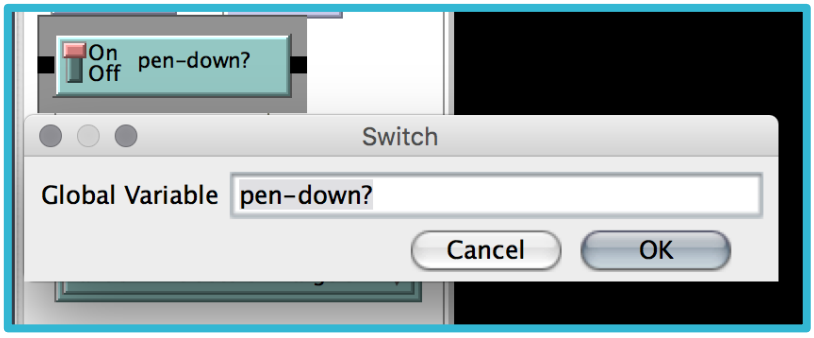
```
;;1) robots need to know:
robots-own [
   ;;are they currently working with a list of rock locations?
   ;;(in the processingList? state)
   processingList?

   ;;are they currently returning to the base?
   ;;(in the returning? state)
   returning?

   ;;store a list of rocks we have seen
   ;;rockLocations is a list of lists: [ [a b] [c d]...[y z] ]
   rockLocations

   ;;target coordinate x
   locX

   ;;target coordinate y
   locY

   ;;what heading (direction they are facing in degrees) they start with
   initialHeading
  ]
```

We'll need to create some robots and set values for the properties we just gave them. Scroll to the setup procedure. Note that the setup procedure contains several subprocedures: `make-robots`, `make-rocks`, and `make-base`. Write the `make-robots` procedure now. The `make-robots` subprocedure is located just beneath the setup procedure.

Use the following picture to guide you and add your own comments. It's good to get in the habit of adding comments now as they will be required for your final Swarmathon submission.

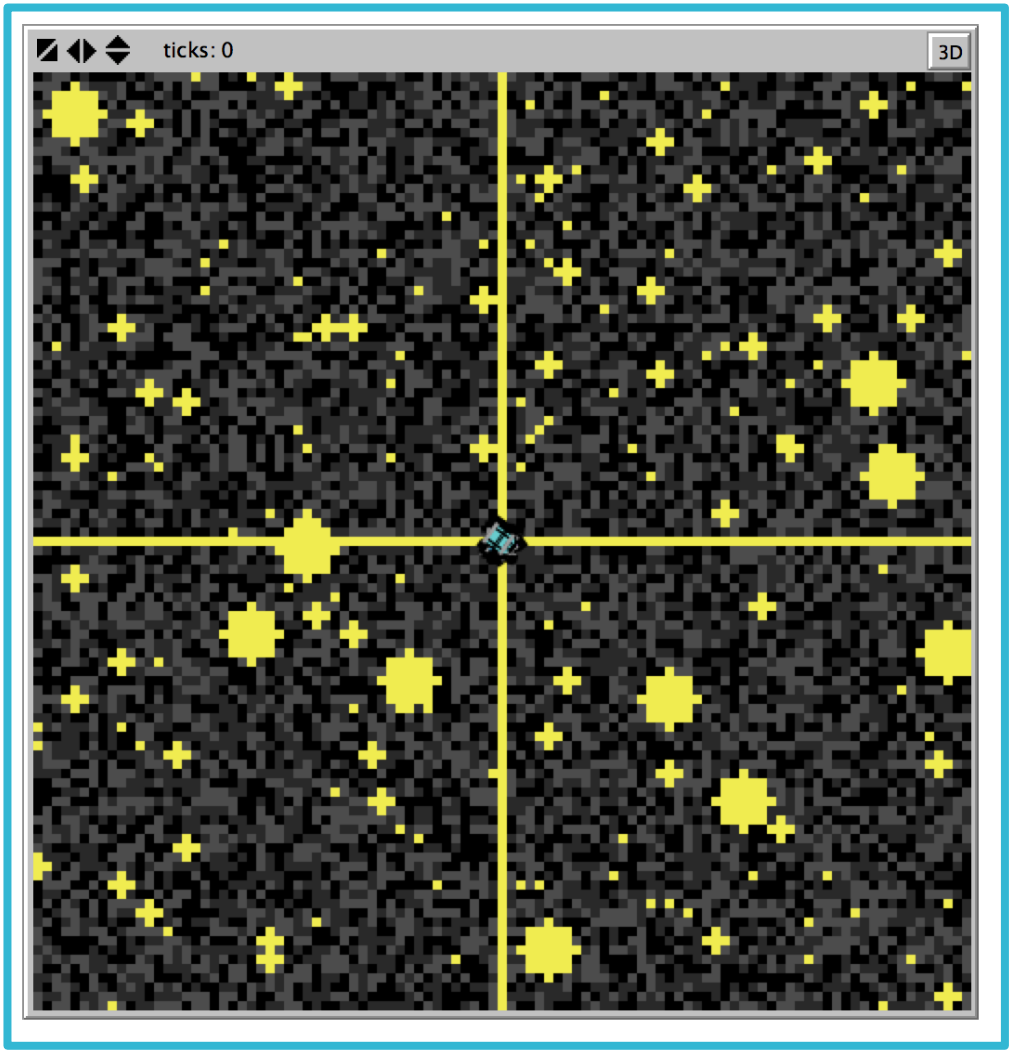
```
;Fill in the make-robots sub-procedure below, then carefully
;read the sub-procedures that follow to understand what the code is doing.
;-------------------------------------------------------------------------------
;;1) Create the number of robots equal to the value of the
;;numberOfRobots slider.
;; Set their properties and their variables that you defined previously.
to make-robots
  create-robots numberOfRobots[
    set size 5
    set shape "robot"
    set processingList? false
    set returning? false
    set rockLocations []
    set locX 0
    set locY 0
    set initialHeading random 360
    set heading initialHeading
    if pen-down? [pen-down]
  ]

end
```

Note that we haven't defined pen-down? yet! Let's do that now by navigating to the Interface tab and adding a new kind of feature to our Interface: a Switch.



Robots have a built-in ability to draw their path with a pen. By flipping this switch, we can now this behavior turn on (pen-down) or off (pen-up).

Click the setup button on the interface to test your code. Your robots should appear at the base and rocks are created. Try changing the rock distribution by selecting a different value from the distribution drop-down menu on the Interface. Also, try changing the slider values that control the number of rocks of each type. There are nearly countless combinations.



In the picture above we have 6 robots and the random + clusters + large clusters + cross distribution.

3 IMPLEMENTING DFS

3.1 DFS PROCEDURE

We will structure our program based on the values of the robot's internal state, as we did in Swarmathon 1 and 2. Our DFS button will activate a main DFS procedure that sets and decides what state a robot should be in, and then calls the appropriate subprocedure. Fill in the DFS procedure as in the picture below. Carefully read the comments to understand what the code is doing.

```
 ;;;;;;;;;;;;;;;;;
 ;;    DFS      ;; : MAIN PROCEDURE
 ;;;;;;;;;;;;;;;;;
 ;-------------------------------------------------------------------------------
;;Write the DFS procedure.
to DFS

  ;;1) Put the exit condition first. Stop when no yellow patches (rocks) remain.
  if count patches with [pcolor = yellow] = 0 [stop]

  ;;2)All sub procedures called after this (set-direction, do-DFS, process-list) are within the ask robots block.
  ;;So, the procedures act as if they are already in ask robots.
  ;;That means that when you write the sub procedures, you don't need to repeat the ask robots command.
  ask robots[

   ;;If the robot can't move, it must've reached a boundary.
   if not can-move? 1[
     ;;Add the last rock to our list if we're standing on it by calling do-DFS.
     do-DFS

     ;;If there's anything in our list, turn on the processingList? status.
     ifelse not empty? rockLocations
     [set processingList? true]

     ;;else go home to reset our search angle.
     [set returning? true]
   ]

   ;;3) Main control of the procedure goes here in an ifelse statement.
   ;;Check if we are in the processing list state and not returning. If we are, then process the list.
   ;;(While we are processing, we'll also sometimes be in the returning? state
   ;;at the same time when we're dropping off rocks.
   ;;Robots should only process the list though when they're not dropping off a rock.
      if processingList? and not returning? [process-list]

   ;;If returning mode is on, the robots should return-to-base.
      if returning? [return-to-base]

   ;;Else, if the robots are not processing a list and not returning, they should do DFS.
      if not processingList? and not returning? [do-DFS]

  ]
  tick ;;tick must be called from observer context--place outside the ask robots block.
end
```

Notice that DFS called 3 other procedures: do-DFS, process-list, and return-to-base. (The last one should look familiar!)

The robots won't do anything until we write those procedures and their subprocedures, so let's get to work!

3.2 PROCESS-LIST PROCEDURE

Scrolling down from DFS, the next procedure to write is process-list, pictured below. Process-list will handle the robots' behavior when they are in the processing-list? state; that is, the state they go into after they have traveled from the base in their chosen direction and hit the edge of the arena (when they have found rocks). Write process-list now.

```
;-------------------------------------------------------------------------------
 ;;;;;;;;;;;;;;;;;;
 ;; process-list ;; : MAIN PROCEDURE
 ;;;;;;;;;;;;;;;;;;
;-------------------------------------------------------------------------------
;;Write the process-list procedure.
to process-list

  ;;1) Control the robots based on the status of their internal list of rocks.
  ;;If the robot's list is not empty:
  ifelse not empty? rockLocations[

  ;;2) ;If locX and locY are set to 0, then we just started or we just dropped off a rock.
    if locX = 0 and locY = 0 [

    ;;If they are, then we need a new destination, so reset our target coordinates, locX and locY.
    ;;We'll write the code for that in a sub procedure, so just call the procedure for now.
      reset-target-coords
    ]

    ;;Now move-to-location of locX locY.
    ;;We'll write the code for that in a sub procedure, so just call the procedure for now.
    move-to-location
  ]

  ;;3) rockLocations is empty. We're done processing the list.
  [set processingList? false]

  ;;Go forward 1 step.
  fd 1

end
```

Note that movement is also handled here with **fd 1**. Robots can move a maximum of one step each tick.

Process-list has **two** subprocedures:

1. reset-target-coords, which examines the robot's list after dropping off a rock and handles its targeting based on the contents; and
2. move-to-location, which controls the robot's behavior as it moves towards the next location in its list.

Complete this section by writing reset-target-coords and move-to-location. These two subprocedures are heavily commented to help you. Look at the following pictures following the name of each subprocedure to write them. Read the comments carefully to understand what the code is doing.

WHY DO WE USE SUBPROCEDURES?

Subprocedures help keep code **organized**.

Subprocedures make code **easier to read**.

Subprocedures are sometimes **used by multiple main procedures**, so code does not have to be repeated.

RESET TARGET COORDS

```
;;Fill in sub procedures.
;-------------------------------------------------------------------------------
;;1) Reset the robot's target coordinates when they are still processing the list but
;;have just dropped off a rock and don't know where to go.
;;Recall that rockLocations is a list of lists: [ [a b] [c d]...[y z] ]
to reset-target-coords

  ;;if rockLocations is not empty
  if not empty? rockLocations[

        ;;Grab the first element of rockLocations, a list of 2 coordinates: [a b]
        let loc first rockLocations

        ;;Now set robots-own x to the first element of this [a _]
        set locX first loc

        ;;and robots-own y to the last. [_ b]
        set locY last loc

        ;;and keep everything but the first list of coords (the ones we just used)
        ;;in rockLocations. --> [ [c d]...[y z] ]
        set rockLocations but-first rockLocations
  ]

end
```

MOVE-TO-LOCATION

```
;-------------------------------------------------------------------------------

;;2) The robot arrived at its locX locY. Pick up the rock and set the robot's mode
;;to returning so it can drop off the rock. Remain in processing state so the robot goes
;;back to processing the list after dropping off the rock.
to move-to-location

  ;;If we've reached our target coordinates locX and locY,
  ifelse (pxcor = locX and pycor = locY)[

        ;; pick up the rock by setting the robot's shape to the one holding the rock,
        set shape "robot with rock"

        ;; and ask the patch-here to return to its base color.
        ask patch-here[ set pcolor baseColor]

        ;; Turn on returning? mode.
        set returning? true
  ]

  ;Else the robot has not arrived yet; face the target location.
  [facexy locX locY]

 end
```

3.3 A FAMILIAR FRIEND—RETURN-TO-BASE

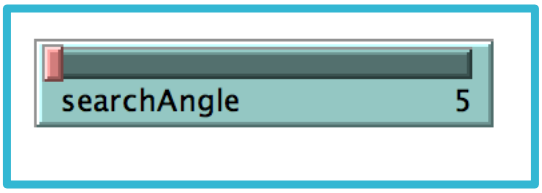
Let's write the return-to-base procedure. As in Swarmathon 1 and 2, a robot running this procedure detects if it has reached the base, drops off the rock if it has and changes it shape back to the one not holding a rock, then switches out of returning? mode. We've added a few extra behaviors for Swarmathon 3:

SET LOCX AND LOCY TO 0

Robots that have reached the base change their destination (locX and locY) to 0. returning? was turned off, but processingList? is still on. That means that on the next tick, the robot will process-list. Now look at the process-list procedure. **Notice that there is a special condition for robots whose target coordinates are set to 0**: the reset-target-coords subprocedure is called to give them a new destination. Through this series of handoffs, we are able to use simple robot states to encode complex behaviors.

CHANGE HEADING IF LIST IS EMPTY

You may have wondered how we would have the robots do more than simply travel in one line. You also may have wondered this slider was for on the interface:



Robots who are dropping off the last rock in their list will increase their heading by the value of searchAngle. **Example**: A robot's current heading is 90 degrees (right). If the value of searchAngle is 5, as in the picture above, the robot will set its new heading to 95 degrees after clearing its list. It will then travel in a line from the origin to the edge of the arena.

Now that you've learned about the new behaviors added to return-to-base, try to write the parts of the procedure that you recognize from Swarmathon 1 and 2 on your own. Then look at the picture to write the new behaviors.

```
;-------------------------------------------------------------------------------
 ;;;;;;;;;;;;;;;;;;;;
 ;; return-to-base ;; : MAIN PROCEDURE
 ;;;;;;;;;;;;;;;;;;;;
;-------------------------------------------------------------------------------
 ;;1) We've used the return-to-base procedure many times.
 ;;This time, we'll make some changes to support list processing.
 to return-to-base

 ;; If we're at the origin, we found the base.
 ifelse pcolor = green[

 ;; Change the robot's shape to the one without the rock.
   set shape "robot"

 ;; We've arrived, so turn off returning? mode.
   set returning? false

 ;; set locX
   set locX 0

 ;; and locY to 0. Robots will return to base if they don't find anything.
   set locY 0

  ;;Use an if statement. A robot can also be here if it has finished processing a list
  ;;of if it didn't find anything at the current angle and was sent back to base.
  ;;If this happened, change its heading so it searches in a different direction.
  ;;It will begin to search +searchAngle degrees from its last heading.
   if not processingList?[
     set initialHeading initialHeading + searchAngle
     set heading initialHeading
   ]
 ]

 ;; Else, we didn't find the origin yet--face the origin.
 [ facexy 0 0 ]

 ;; Go forward 1.
 fd 1

 end
```

3.4 DO-DFS

In Sections **3.2** and **3.3**, we handled the conditions of the robot processing its list and dropping off rocks it found at the list coordinates. To finish up, let's write the DFS behavior itself, which is contained in the do-DFS procedure. This procedure is strikingly simple—it contains just **four lines of code**!

The code, however, contains some new commands that are not easy to understand by reading them. If there's a command that you don't understand, try looking it up in the NetLogo dictionary.

Let's look up the command **fput**, which is used in do-DFS. Follow the steps in the box below.

EXERCISE: FINDING COMMANDS IN THE NETLOGO DICTIONARY

- Go to the NetLogo Dictionary website at http://ccl.northwestern.edu/netlogo/5.0/docs/dictionary.html.
- To search the webpage, press Command-F (Mac) or Control-F (Windows). Type in **fput**.
- The page jumps to the **fput** entry in the list section and is highlighted.

- Click on the **fput** link and read the description.

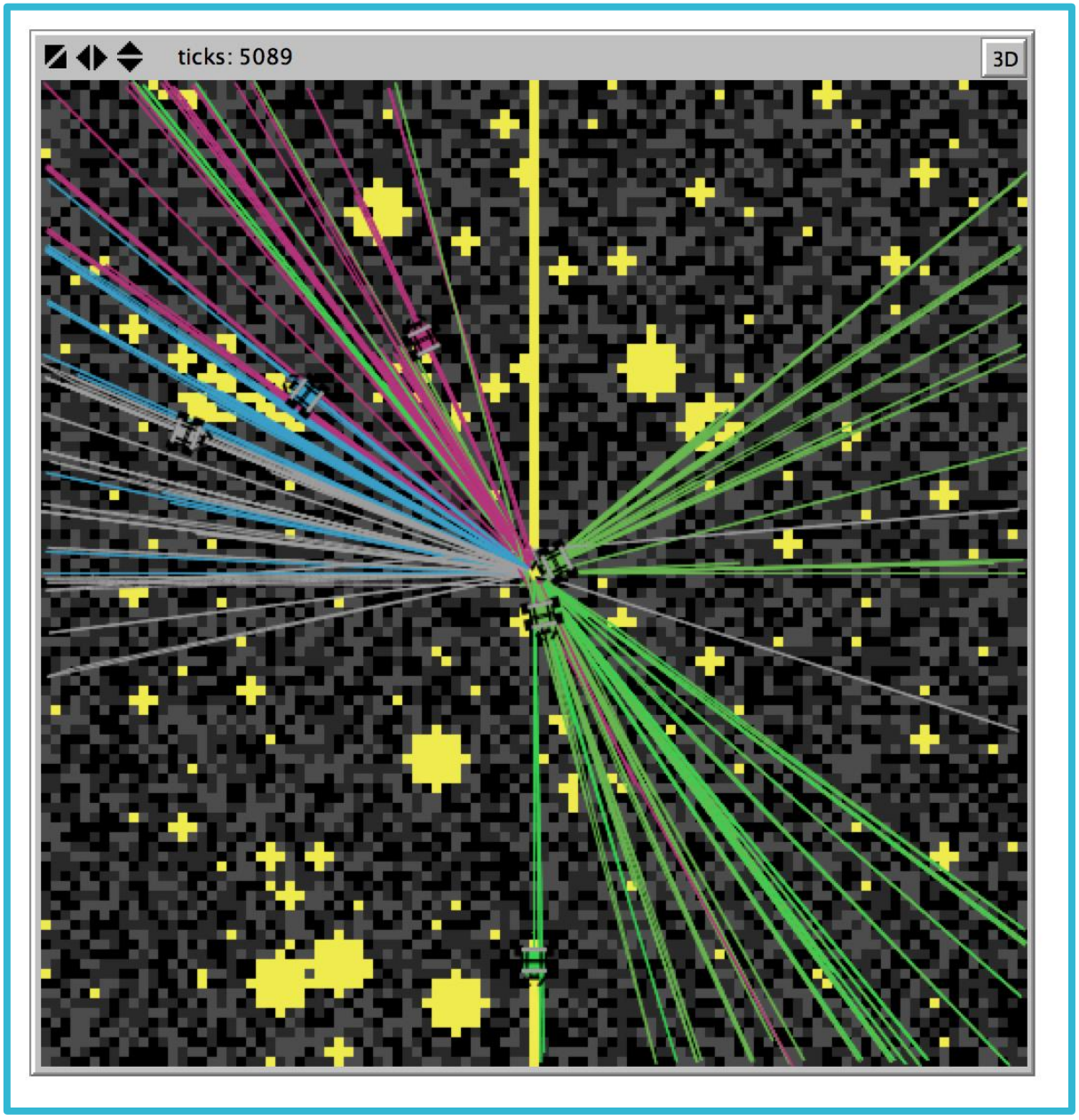
You are encouraged to use the NetLogo dictionary as a reference. You may also get some ideas for your competition submission by looking through it!

Let's finish up Swarmathon 3 by writing the do-DFS procedure. Use the picture below to guide you.

```
;-------------------------------------------------------------------------------
 ;;;;;;;;;;;;;;;;;
 ;; do-DFS      ;; : MAIN PROCEDURE
 ;;;;;;;;;;;;;;;;;
 ;-------------------------------------------------------------------------------
;;Write the do-DFS procedure. do-DFS finds rocks and stores them in a list.
to do-DFS

  ;;1) ask the patch-here
  ask patch-here[

     ;;if its pcolor is yellow,
     if pcolor = yellow[

      ;;make a list of the coords of the rock we're on.
      let location (list pxcor pycor)

          ;;to add those coordinates to the front of their list of rocklocations and remove any duplicates.
         ask myself[ set rockLocations remove-duplicates (fput location rockLocations)]
     ]
  ]

  ;;Go forward 1.
  fd 1

end
```

Test your module by pressing setup and DFS. Ask yourself the following questions:

- Try changing the value of the searchAngle slider. How does changing the value effect the robots?
- What happens when the value of searchAngle is large?
- What are some advantages to DFS vs random search? How about some disadvantages?



GREAT JOB! You completed SWARMATHON 3.

BUG REPORT? FEATURE REQUEST?

Email sherbet@unm.edu with the subject SW3 Report

NEXT UP
SWARMATHON 4: Advanced Deterministic Search