Stat 133, Fall 16
Ad Hoc Network Simulation
Due: Tuesday, 25 Oct

Introduction

Wireless networks are all around us. Cell phones communicate with a base-station to send and receive calls. Calls are relayed from base-station to base-station as the cell phone moves away from one and closer to another. A new idea of organizing networks is to avoid the need for a central base-station that coordinates communications. Instead, messages are relayed by "hopping" from one node to the next to the next until it reaches its destination. In other words, one can send a message by using other devices in the network to relay the message to the next device, and so on. These are called *ad hoc* networks because there is no centralized node or fixed structure or topology for the network. Instead, devices can move over time, and dynamically enter and exit the network. And so the route a message takes from one device to another depends on the other nodes.

Ad hoc networks are very promising and becoming important. At their most immediately practical, ad hoc networks can allow nodes outside of a regular network to communicate by piggy-backing off of nodes within the network. Think of driving along and being between base-stations and so your cell phone call would be dropped. But because of ad hoc networks, your data is relayed through cell phones in other cars closer to the base stations. More ambitously, ad hoc networks might be used in controlling traffic on highways by allowing cars communicate with each other.

A very basic aspect of ad hoc networks that people need to understand is how the communication and complete connectivity changes with respect to the broadcasting power. Increased power levels allow one to send a message over a larger distance.

Tasks

This project consists of 3 main tasks.

1. Start by writing a function to generate nodes in an ad hoc network. The function should meet the following conditions:

   (a) The name of the functions is `genNodes()`

   (b) Input argument: `n`, the number of nodes to generate (required).

   (c) Return value: an `n` by 2 matrix with the first column representing the x coordinates and the second columns the y coordinates of the nodes.

   Use *acceptance-rejection sampling* to generate the points at random in the network. The nodes should be generated at random on a 2-dimensional grid shown in Figure **??**, where the node density is proportional to the function shown in Figure **??**. A density function called, `nodeDensity()`, is supplied to help you. This function takes two inputs: x and y, two numeric vectors of the same length. The function returns a numeric vector of values that are proportional to node density at the (x, y) pairs. This function is available on the Web and can be sourced into your R session with

   ```
   source("http://www.stat.berkeley.edu/users/nolan/data/nodeDensity.R")
   ```

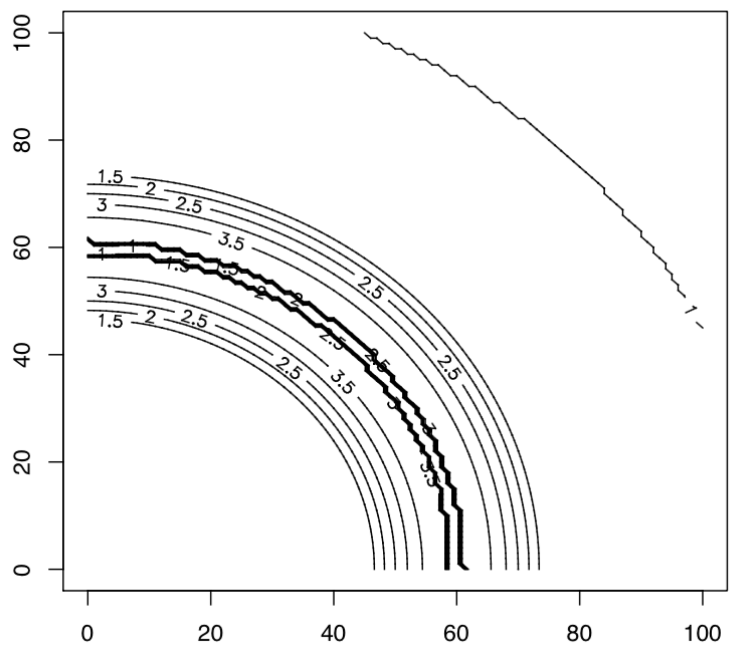

Figure 1: Contour plot of the region of interest. The contours are proportional to the density of nodes.

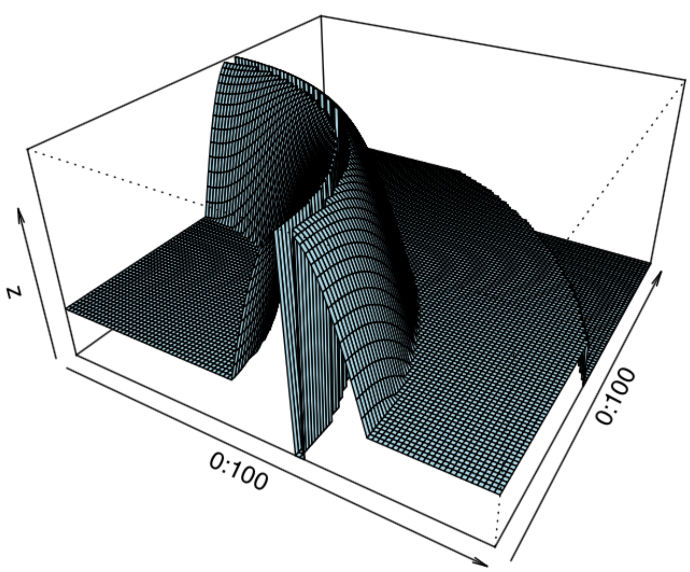

Figure 2: A three-dimensional perspective plot of the region of interest. Note that the river curves through the center of the region and no nodes are located near the river.

2. For a given configuration of nodes, find the smallest radius, R_c, such that the network is completely connected. Write a function to find R_c for a given collection of nodes. This function has the following properties:

   (a) The function is called findRc()

   (b) The first input parameter is `nodes`. It is required. This input is a 2-column matrix of the x and y locations of the nodes

   (c) The second input parameter is `tol`. It has a default value of 0.05, which is the tolerance level for how close you need to get to the true value of R_c for the provided configuration.

   (d) The return value is a numeric vector of length 1, that holds the value of R_c (or a value close to it).

   We are interested in finding the smallest power level that leads to a connected network. We assume here that power level is proportional to the radius, R, of a circle centered on the node, and two nodes are in direct communication if the distance between them is less than R.

   The task of finding R_c relies on a probability argument, some mathematics, and a search algorithm. These are described in the background section. There you will be asked to write a couple of helper functions for findRc().

3. Select a range of sample sizes for n. Then for each n generate 1000 networks and find the value for R_c for each of these 1000 networks. Examine the distribution of these R_c values. Some questions for you to consider are the following:

   (a) How does R_c, the smallest radius such that the network is connected, change with different node configurations?

   (b) Explore the distribution of R_c. Is it symmetric, skewed, long tailed, multimodal?

   (c) Plot the network of connected points for four of your 1,000 n-node configurations corresponding *roughly* to the min, median, mean, and maximum values of R_c.

   Note that in order to plot a network, you will want to be able to keep track of the n-node configuration that correspond to the min, median, mean, and max values of R_c. For example, if you keep track of the value of the seed at the start of each simulation, then, when you find that say the m^{th} simulation corresponds to the median value for R_c, you can set the seed to the m^{th} seed that you have saved and recreate the nodes.

Background

For a set of nodes to be connected means that it is possible for any two nodes to communicate by having a message travel along nodes that are within broadcasting distance of each other. If the power level is very high then all nodes will be able to broadcast directly to each other. On the other hand, if the power level is too low then a node may not be able to connect to any other node, or the nodes may form two disjoint connected subsets. Suppose, our network has n nodes in it, and we have a matrix of all of the pairwise distances between two nodes in the network. That is, the matrix is $n \times n$ where the entry in the i^{th} row and the j^{th} column is the distance between nodes i and j. This matrix is symmetric and has 0s on the diagonal.

Random Walk on the Nodes

A probability model for messages moving on the network can help us determine if the network is completely connected. For a particular power level R, suppose a node has 3 neighbors within R of it. Then in our random walk, a message located at this node will hop at random to one of these three nodes or will stay at its current node, and these four possible scenarios are equally likely. In this way, a message hops around on the network in a random fashion.

We can describe this random "walk" via a transition matrix. That is, for a message located at node i, $i = 1, \ldots, n$, the chance the message moves to node j is 0 if these two nodes are further than R away from each other. Otherwise, it is $1/k_i$ where k_i is the number of nodes within R of node i (including node i itself). The $n\ timesn$ matrix of these transition probabilities is called P (note that it depends on R).

If v_m is a $n \times 1$ row vector of probabilities that a message is at any one of the n nodes at one "instant", then $v_m P = v_{m+1}$ is the distribution of locations of the message at the next instant. And $v_{m+1} P = v_{m+2}$ is the distribution for the next instant. (We can also think of v_m as the distribution of many messages across the network.)

Mathematical properties of transition matrices tell us many things. Namely,

1. the distribution of the locations of the message settles down, i.e., there is some v where $vP = v$.

2. This equation indicates that the steady state (i.e. v) is the eigenvector of the transition matrix associated with the eigenvalue of 1.

3. The eigenvalues of P all have magnitude at most 1.

4. If the network of nodes is fully connected, then there is one unique steady-state solution. In this case, the largest eigenvalue is real, has value 1, and all the other eigenvalues have magnitude less than 1.

The above properties imply that the size of the second largest eigenvalue of P is key to determining if the network is connected.

Write a helper function called findTranMat() to find the transition matrix based on a distance matrix and a value for R. That is, this function takes as an input a distance matrix called `mat` and a value for R, called `R`. Both of these are required arguments. The function returns the transition matrix P for these inputs.

Write a second helper function called getEigen2() which returns the magnitude of the second largest eigenvalue of a matrix. The input to this function has one argument, which is required. The parameter is a matrix, called `mat`.

Range of possible values

What is the smallest possible value that R_c can be? Since each node must be connected to at least one other node, then R_c must be at least as large as the greatest row-wise minimum (ignoring the diagonal element).

Similarly, what is the largest value that we need to consider for R_c? If r_c is greater than the maximum distance in a row, then the corresponding node will be connected to all of the other nodes, i.e., the network will be connected. So We know that R_c is no greater than the smallest row-wise maximum.

These observations give us a lower and upper bound to search within to find R_c.

Write a function called findRange(), which finds the range of Rs to search over based on the above observations. This function has one input: the distance matrix called `mat`. It is required. The function returns a numeric vector of length 2, with the minimum and maximum values of R to search over.

Searching for the smallest value

Our function findRange() gives us an interval to search in for R_c. We know that if the network is completely connected for a particular value of R^*, then it is connected for all larger values, $R > R^*$. We want to find the smallest R (within a particular tolerance) that gives us a completely connected network.

One way to do this is to start with the middle of the interval returned from findRange(), call this midpoint R_0. If R_0 gives a completely connected network, then check the midpoint between the minimum and R_0. Call this point R_1. If this value does not give us a completely connected network, then we know the minimizer is larger than R_1 and smaller than R_0. Pick the midpoint between these 2 points, call it R_2. If R_2 gives a completely connected network, we next try the midpoint between R_2 and R_1; otherwise, we try the midpoint between R_2 and R_0. Continue splitting the intervals, moving left or right depending on whether the midpoint is completely connected (move left) or not (move right). This split search will zero-in on the minimizing value quite quickly. However, you will want to discretize this search, i.e., if you have set the tolerance to 0.001 then you need only examine the values in the range of potential Rs at these locations. Additionally, if the minimizing R_c occurs at the right end point of the range, then when your search gets within 0.001 of the right end point, it finds the minimizing value.

Tips

You are to turn in BOTH an Rmd file and a knitted HTML/Word/PDF document of your simulation study. Be sure to document your code and use the function names, arguments, and return values described above.

Below are some functions that you may find helpful:

- .Random.seed() - used to set and save the seed for the random number generator.
- runif() - a pseudo-random number generator that takes vector arguments.
- dist() - computes all pairwise distances between rows in a matrix or data frame.
- eigen() - computes eigenvalues and eigenvectors. Be sure to read about all of the arguments.
- eigs() - in the `RSpectra` package. The eigs() function lets you calculate the largest k eigenvalues (you can also specify omitting the eigenvectors through `opts`).
- which()-can be very handy for figuring out which elements in a vector have a certain property.
- expand.grid() - a very useful function when creating a grid from vectors, if you need to search over a grid.
- Mod() - calculates the magnitude of complex numbers

Tests for some of the functions that you have been asked to write are available in tests-ForAdHocProject.R in bcourses in the Project folder. Also there is an example of how to use `.Random.seed` to keep track of and reset the random number generator.

Depending on your computer's capabilities, you may choose to run your program one the Statistical Computing Facility machines. If you want to do this, please start early so that you can get an account and figure out how to use the machines.