Watts & Strogatz model

duw

10/8/2018

The goal of this assignment is to program the Watts-Strogatz model (WS) and to validate their findings presented in Figure 2 of Watts and Strogatz (1998).

Overview

The WS model consists of 4 steps:

1. Find edges - Edges of the regular neighborhood lattice are identified and stored in an edge-list.
2. Order edges - Edges are ordered according to their distances and positions in the ring.
3. Fill network - Empty matrix representing the network is filled with edges.
4. Re-wiring - Move through the ring of nodes and rewire edges with probability p.

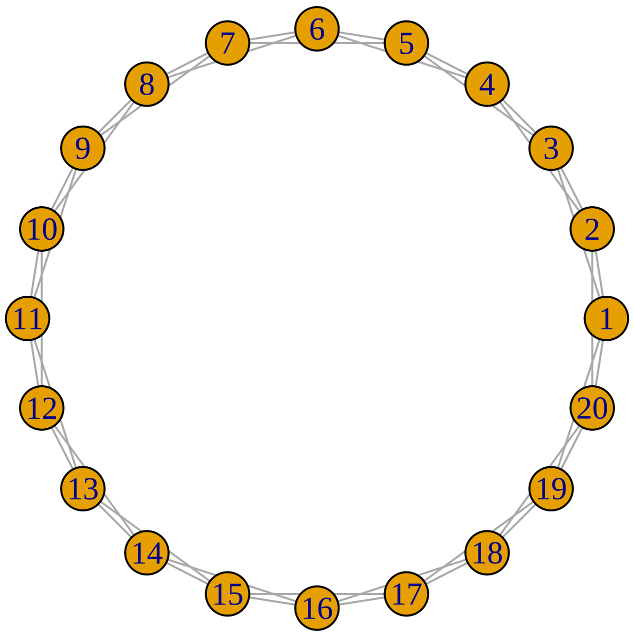
Here is how Watts and Strogatz (1998) describe their model:

Random rewiring procedure for interpolating between a regular ring lattice and a random network, without altering the number of vertices or edges in the graph. We start with a ring of n vertices, each connected to its k nearest neighbours by undirected edges. (For clarity, n = 20 and k = 4 in the schematic examples shown here, but much larger n and k are used in the rest of this Letter.) We choose a vertex and the edge that connects it to its nearest neighbour in a clockwise sense. With probability p, we reconnect this edge to a vertex chosen uniformly at random over the entire ring, with duplicate edges forbidden; other- wise we leave the edge in place. We repeat this process by moving clockwise around the ring, considering each vertex in turn until one lap is completed. Next, we consider the edges that connect vertices to their second-nearest neighbours clockwise. As before, we randomly rewire each of these edges with probability p, and continue this process, circulating around the ring and proceeding outward to more distant neighbours after each lap, until each edge in the original lattice has been considered once. (As there are nk/2 edges in the entire graph, the rewiring process stops after k/2 laps.) Three realizations of this process are shown, for different values of p. For p = 0, the original ring is unchanged; as p increases, the graph becomes increasingly disordered until for p = 1, all edges are rewired randomly. One of our main results is that for intermediate values of p, the graph is a small-world network: highly clustered like a regular graph, yet with small characteristic path length, like a random graph. (p. 441)

Step I - Find edges

1. This is the hardest step. It requires that we iterate (loop) twice over the nodes to determine their distances and to then include edges for nodes whose distance is smaller than a certain neighbor size. Let's begin by considering how many edges we need to expect. Let's assume we have, as in Watts and Strogatz description, 20 nodes and we want to connect each node to its 4 closest neighbors, i.e., those 1 step apart and those 2 steps apart. This means that we want to include edges that connect, for instance, node 1 to node 2, 3, 20, and 19, or node 11 to node 12, 13, 10, and 9. Given this information it is straightforward to determine the number of edges. 20 nodes each connected to 4 neighbors means `20 * 4` connections and `(20 * 4)/2` edges because each edge is counted twice. That is, in an undirected network, the edge between 1 and 2 is the same edge as the one between 2 and 1. Alright, now that we know the number of edges let's create a matrix with as many rows as there are edges and exactly 3 columns. We include 3 columns because we represent edges by two numbers, the two numbers representing the two nodes that the edge binds together, and additionally include a third

column to store the distance of the two connected nodes. Remember empty matrices can be created using `matrix()` with only the `nrow` and `nrow` arguments. Create the matrix and call it `edges`.



2. Next, we need iterate through the matrix using loops. Remember that loops take the form `for(i in sequence){}` with the `sequence` being some iterable object, such as a vector or a list. In our case this is a vector containing every number of nodes. As nodes are numbered `1` to the size of the network, we can easily create such a vector using the colon operator, i.e., `1:size`. Since we want to iterate over pairs of nodes rather than single nodes, we need two loops, one outer loop and one inner loop, such as shown below:

```
for(i in 1:(size-1)){
  for(j in (i + 1):size){

    CODE HERE

    }
  }
```

The example code may first look somewhat surprising. Why iterate through `1:(size-1)` in the first loop and through `(i+1):size` in the second loop? Think about it! What would happen if we would iterate through `1:size` in both loops? Assume size was only 3, then this would mean we would create the following pairs of `i` and `j`: `(1, 1) (1, 2) (1, 3) (2, 1) (2, 2) (2, 3) (3, 1) (3, 2) (3, 3)`. Thus, we would not only include every pair twice, but we would also include edges of nodes with themselves, i.e., (1, 1) (2, 2) (3, 3).

3. Ok, now let's think about what we want to achieve inside our loops, that is, in place of "CODE HERE". Our goal is to identify edges that connect nodes which - with respect to the circle of nodes - are no more than a certain number of steps apart. How to determine distance? That's easy! The nodes are numbered from `1:size` and placed in order along the circular layout, such that node `1` and node `2` are one node away, node `1` and node `3` are two nodes away, and node `3` and node `7` are 4 nodes away. Thus, by taking the difference between node numbers we can determine their distance. The only problem is that the node with number `size` lies adjacent to node `1`, i.e., one step away, but their numeric

difference is `size - 1`. In order to fix this, we have to consider not only the difference `j - i` but also the difference `(j - size) - i`. Furthermore, we have to take the absolute of the differences, so that we are only observing positive distances. Thereby, we are taking account for every pair of nodes that their distance can be determined clock-wise and counter-clockwise. The true distance is then the minimum of the two, thus `distance = min(clock-wise_distance, counter-clockwise_distance)`.

4. Once we determined the distance between the two nodes, which essentially means determining the distance between `i` and `j`, we can figure out whether the distance is smaller or equal to the maximum distance `max_distance` allowed. Let's test this via an if-statment. Remember if-statements take the form `if(logical_comparison){}` with `logical_comparison` encompassing a comparison of two values, e.g., the observed distance and `max_distance`. Given that the observed distance should be smaller than `max_distance` or equal, we use `distance <= max_distance`.

5. If a distance is indeed smaller than `max_distance`, we need to include it in our `edges` matrix. We do this by indexing the matrix at the next open slot for an edge, i.e., `matrix[index, ]` and assign to it a vector containing `i`, `j`, and `distance`. This leaves us only to define `index`. A simple way to do this is to have R increment the index every time a new edge was stored in the `edges` object. That is, before we begin the loops we define an index with value 1 and the everytime we found an edge we add 1 to it, i.e., `index = index + 1`. See the code example.

```
index = 1
for(i in 1:(size-1)){
  for(j in (i + 1):size){

    CODE HERE

    if( CODE HERE ){

      CODE HERE

      index = index + 1
      }
    }
  }
```

6. All that's left is to put everything together and run the code.

Step II - Order edges

1. The next step is to order the `edges` according to the distance, with short distances first, and then the numbers of the nodes. I.e., we want to order by `ord = order(edges[, 3], edges[, 1], edges[, 1])`, which is a list of indexes that we can use to bring the `edges` object in the right order. To do this place `ord` in the row entry of the matrix index, i.e., the first position of `edges[, ]` and reassign the object to itself. Step complete.

Step III - Fill network

1. Now we want to fill a network. To do this we first create an empty matrix with `data = 0`, `nrow = size`, and `ncol = size` and call it `network`. Every position in this matrix represent the presence or absence of an edge between the row's node and the column's node. In the beginning, all values are zero indicating that there are no nodes. The challenge is now to add 1s whenever two nodes are connected with an edge. The easiest way to do this is by making use of the fact that one can use matrices to

index matrices, i.e., `matrix[matrix]`. Note that there is no comma. This means we can use the first two columns of `edges`, i.e., `edges[,1:2]` to index the empty `network` and assign to the selection the value one. Do this also for `edges[, 2:1]` in order to fill the other half of the matrix.

Step IV - Re-wiring

1. The final step is a bit more challenging again. The goal of this step is that we iterate through the ordered `edges`, for every edge decide whether it needs to be rewired or not, and if it has to be rewired, rewire it. To achieve this, we begin with a loop that iterates through `1:nrow(edges)`, i.e., from 1 to the number of edges.
2. Next we have to decide whether a particular edge needs to be rewired by sampling a random number between 0 and 1 and evaluating whether it is smaller than our predefined probability `p_rewire` of rewiring. We can draw such a random number using `runif(1, 0, 1)`, which draws one value from a uniform distribution defined by endpoints 0 and 1. Using the random draw, we then decide whether it is smaller than `p_rewire` using an if-statement, i.e., `if(random < p_rewire){}`.
3. To rewire, we first need to figure out where-to we need to rewire. To select a new end node, we draw a random number using `sample(1:size, 1)`, which draws one random number from the sequence between `1:size`. Then we test - in an if-statement - whether this random number is different from the first node in the current row in `edges`, i.e., `edges[i, 1]`. If it is then we discontinue because we don't want nodes to be connected to themselves. Then, we need to test whether the new node is not already connected to the current start node `edges[i, 1]`. Again, if it is, we disontinue. That is before we actually rewire, we need to perform two logical tests and we want both to be negative. To test both at the same time, the two tests can be combined using a logical AND, in R `&`, i.e., `if(test_1 & test_2){}`.
4. Finally, if all hurdles are passed, we change the orginal edge in the network to 0 and the new edge to 1. Thus, both, `network[edges[i,1], edges[i,2]]` and `network[edges[i,2], edges[i,1]]`, need to be set to 0 and both, `network[edges[i,1], new_end]` and `network[new_end, edges[i,1]]`, need to be set to 1. Rewiring done.

Last step - Network plot

1. To check out whether everything works, plot your networks for different values of `p_rewire`. To plot the network, we need to get the network first into an `igraph` object. `igraph` is a powerful network science R package that provides a lot of functionality including creating decent looking network plots. To do this, install the package and then apply `igraph::graph_from_adjacency_matrix(network, mode = "undirected")` and store the result in graph.
2. Now you can use `igraph::plot.igraph(graph, layout = igraph::layout_in_circle(graph))` to create plots.