SS: Urban Computing

Tyler Chang
April 15, 2018

Network Science

One of the main tools of urban computing is network science. Basic types of graphs include:

- Unipartite/bipartite graphs (one or two groups)
- Planar graphs (no edges cross)
- Trees (no loops in the graph)

One of the most useful tricks for analyzing a graph is to construct its *adjacency matrix.* Each row/column in an adjacency matrix represents a node. If there is a (directed) edge from node i to j then $A_{ij} = 1$; and if not, then $A_{ij} = 0$. For example, consider a three node graph G with a single edge from node 1 to node 2. The adjacency matrix is:

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

A few important notes:

- Unless G has edges that "loop back" to the same node, A's entire diagonal will be 0.
- If G is undirected, A will be symmetric.
- The outdegree of node i is given by $\sum_{j=1}^{n} A_{ij}$.
- The indegree of node j is given by $\sum_{i=1}^{n} A_{ij}$.

Sometimes, when the data is large and edges are sparse, the *adjacency list* can be easier to work with, where we simply list off the destinations for the outgoing edges from each node. For large bipartite graphs, another way we can save space (at the cost of some loss of information) is by the *projection* operation. The projection is given by taking only one mode of the graph, and connecting every pair of nodes that share a mutual link in the second mode.

With these storage formats and operations in mind, the following quesions are generally of interest:

- Graph *connectivity.*
  - How many components does it have, and how big are they?

  - Is there a giant component, and how big is it?

- Distance between nodes.
  - What is the shortest path between two nodes?
  - What is the *diameter* of the graph?

- Network *density.*
  - Consider the total number of nodes that could exist: $e_{max} = n(n-1)$ for a directed graph, $e_{max} = n(n-1)/2$ for an undirected graph.
  - Consider the total number of nodes that do exist: e.
  - density $= e/e_{max}$.

- Node *centrality*
  - (In/out) degree centrality
  - Betweenness centrality
  - Closeness centrality

- Communities (sets of nodes between which connections are common)
  - k-cliques are sets of nodes that are all mutual neighbors up to k degrees of separation.
  - If a group of nodes is λ-dense, i.e., density $> \lambda$.
  - Node similarity: construct the adjacency vector for each node, then two nodes with vectors A and B can be compared by how much these vectors align:
    * Cosine Similarity: $\cos(\theta) = \frac{A\dot{B}}{\|A\|\|B\|}$.
    * Jaccard Similarity: $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$.
    * K-means clustering over vectorized representations for all nodes.

- Isomorphisms: Two graphs are isomorphic if one can be transformed into the other by a structure preserving mapping. I.e., if we can obtain the second by simply renaming elements.

A final powerful tool for graph analytics is the modelling of distributions. A standard technique is to fit a Gaussian (normal) distribution to some quantifiable aspect of the nodes or edges in the graph (such as their connectivity). If the distribution can't be fit with a Gaussian distribution, one can often apply a log or power transform and see if it becomes normal. When fitting the PDF doesn't work, one can try fitting the CDF, or one of the other common parametric distributions:

- Pareto: $P[X > x] \approx x^{-k}$.

- Poisson: $P(k \text{ event in interval}) \approx e^{-C}\frac{C^k}{k!}$.

Applications of the network analysis in urban computing include traffic flow modelling, power grid modelling, social network analysis, economics (supply chains and operations), and computational epidemiology.

The Python package for network anlaysis is `NetworkX`.

Page Rank

Page rank is the search algorithm behind Google, and one of the best demonstrations of the power of networks. The idea is simple, imagine a random web surfer on the internet clicking links. The probability that this surfer ends up at a page X should correlate with page X's importance.

To model this, a simple Markov model is used: Set up a weighted graph where each node corresponds to a webpage and each edge corresponds to the probability that the surfer will head to that next web page (I.e., each edge weight out of page Y should be $\frac{1}{n}$ where n is the total number of links out of Y). At each time step, the model update corresponds to a multiplication of the current state by the (weighted) adjacency matrix. The stability point of this iterative update is the largest eigenvalue of the update matrix, and the normalize eigenvector tells us the probability our random surfer ends up at each web page.

Spatio Temporal Data Mining

Sometimes there is no meaningful way to define nodes and edges in a continuous space, but we still want to do space or time based data mining. One trivial solution is to discretize the space into a grid, or some other meaningful mesh and correlate neighbors. Essentially, this amounts to assigning each element of the mesh a node, connecting each neighboring element with an edge, then performing standard network science.

Moran's I is another discretization based approach. After creating a grid of data points weighted by some matrix w_{ij} with response values y_i with mean value $\bar{y}$, we compute the spatial correlation by:

$$I = \frac{n}{\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}} \frac{\sum_{i=1}^{n}\sum j = 1^{n}(y_i - \bar{y})(y_j - \bar{y})}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

A positive value above indicates that space is strongly correlated with the response value y and a negative value indicates that y is nearly uniformly distributed throughout space.

Another option for the same problem is by fitting a linear model using the spatial autoregressor, then interpreting a steep slope as a strong correlation:

$$y = \rho W y + X\beta + \varepsilon.$$

The drawback of the above two methods is that they can only detect convex (Moran's I) or linear (autocorrelation) relationships. Density based methods such as Gaussian processes and DBSCAN are able to mine nonconvex clusters of data.

By collecting spatial trajectories (as sequences of locations through time), we can also predict future locations by matching with previously observed trajectories. For example, if the trajectory of locations given by $\{1, 5, 7\}$ is common and we observe an actor at location 1 then 5, we can infer that this actor may be headed toward location 7 next.

The Python package for spatial temporal anlaysis is `PySAL`.