Social Networks in Biology

File: SocNetRandom1000.nb

To accompany
"Getting the 'Edge' on the Next Flu Pandemic: We Should'a 'Node' Better"
By Angela B.Shiflet and George W.Shiflet
Wofford College, Spartanburg, South Carolina

- **This file deals with 1000 people selected at random from "activities-portland-1-v1.dat" at http://ndssl.vbi.vt.edu/opendata/download.php and uses all their activities.**

Based on
Eubank, S., V.S. Anil Kumar, M. Marathe, A. Srinivasan and N. Wang. 2004. “Structural and Algorithmic Aspects of Large Social Networks.” Proc. 15th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 711-720.

Data downloaded from
http://ndssl.vbi.vt.edu/opendata/download.php

NDSSL (Network Dynamics and Simulation Science Laboratory, Virginia Polytechnic Institute and State University). 2009. "NDSSL Proto-Entities" http://ndssl.vbi.vt.edu/opendata/ Accessed 8/27/9.
_____. 2009. Synthetic Data Products for Societal Infrastructures and Proto-Populations: Data Set 1.0. ndssl.vbi.vt.edu/Publications/ndssl-tr-06-006.pdf
_____. 2009. Synthetic Data Products for Societal Infrastructures and Proto-Populations: Data Set 2.0. ndssl.vbi.vt.edu/Publications/ndssl-tr-07-003.pdf
_____. 2009. Synthetic Data Products for Societal Infrastructures and Proto-Populations: Data Set 3.0. ndssl.vbi.vt.edu/Publications/ndssl-tr-07-010.pdf

"NDSSL has produced several synthetic data sets that are being released to the larger academic community for research. The data sets are based on detailed microscopic simulation-based modeling and integration techniques. The data set provided represent a synthetic population of the city of Portland."

Connection matrix

- **read file**
  For this *Mathematica* file, we just used the activities from the first data set.
  For this program, we ignore the times the people were at locations. Thus, if two people went to the same location in a day, even if at different times, we assume they are adjacent in a people-to-people graph.

```
activitiesAll = ReadList["activities-portland-1-v1.dat",
   {Number, Number, Number, Number, Number, Number, Number}];

numActivities = Length[activitiesAll]

8 922 359

maxPersonId = activitiesAll[[-1, 2]]

1 615 860

numPeople = 1000;
```

function to return a sorted list of *numPeople* number of random integers
This will be our *personIdLst*.

```
Clear[getPeople];
getPeople[maxPersonId_, numPeople_] := Module[{lstWithDuplicates, personIdLst},
  personIdLst = {};
  While[Length[personIdLst] < numPeople,
   lstWithDuplicates = Table[RandomInteger[{1, maxPersonId}], {numPeople}];
   personIdLst = DeleteDuplicates[lstWithDuplicates]
  ];
  Sort[personIdLst]
 ]

(* get exactly numPeople=1000 numbers between 1 and maxPersonId,1615860 *)
personIdLst = getPeople[maxPersonId, numPeople];
```

- **Get list of activities of personIdLst people**

```
(* this stops when we know there are no more elements *)Clear[getActivities];
getActivities[activitiesAll_, personIdLst_] :=
 Module[{numPeople, activity, activities, person, i},
  numPeople = Length[personIdLst];
  (* no need to start before this element *)
  activity = personIdLst[[1]];
  activities = {};
  Do[
   person = personIdLst[[i]];
   While[activitiesAll[[activity, 2]] < person,
    activity++
   ];
   While[activitiesAll[[activity, 2]] == person,
    AppendTo[activities, activitiesAll[[activity]]];
    activity++
   ],
   {i, numPeople}
  ];
  activities
 ]

activities = getActivities[activitiesAll, personIdLst];

Length[activities]

5495
```

- **get list of locations**

```
Clear[genLocIDLst];
genLocIDLst[activities_] := Module[{locs},
  locs = Transpose[activities][[7]]; DeleteDuplicates[locs]
 ]

locationIDLst = genLocIDLst[activities];

Length[locationIDLst]

3447
```

- **function to return index of location in *locationIDLst***

```
Clear[locationIndex];
locationIndex[loc_, locationIDLst_] := Flatten[Position[locationIDLst, loc]][[1]]
```

function to return index of person in *personIDLst*

```
Clear[personIndex];
personIndex[person_, personIDLst_] := Flatten[Position[personIDLst, person]][[1]]
```

- **function to generate people-to-location connection matrix for graph**

```
Clear[genPeopleLocConnMat];
genPeopleLocConnMat[people_, locs_, activities_] := Module[{connMat},
  connMat = Table[0, {Length[people]}, {Length[locs]}];
  Do[connMat[[personIndex[activities[[i, 2]], people],
      locationIndex[activities[[i, 7]] , locs]]] = 1, {i, Length[activities]}];
  connMat
 ]
```

```
connMat = genPeopleLocConnMat[personIdLst, locationIDLst, activities]
```

- **function to return the degree of a person node in person-to-location graph**

```
Clear[degPerson];
degPerson[i_] := Count[connMat[[i]], 1]
```

- **function to return the degree of a location node in person-to-location graph**

```
Clear[degLocation];
degLocation[j_, connMat_] := Count[Transpose[connMat][[j]], 1]
```

- **list of ordered pairs of location index & corresponding degree**

```
locDegPairLst = Table[{j, degLocation[j, connMat]}, {j, Length[Transpose[connMat]]}]
```

Minimum dominating set problem

- **function to return *lst* sorted by the second members of the ordered pairs**

```
Clear[sortSecond];
sortSecond[lst_] := Sort[lst, #1[[2]] > #2[[2]] &]
```

```
(* test*)
sortSecond[locDegPairLst]
```

- **Function to return list of personIDs adjacent to location loc**

```
Clear[adjacentPeopleLst];
adjacentPeopleLst[loc_, personIdLst_] :=
 personIdLst[[Flatten[Position[Transpose[connMat][[loc]], 1]]]]
```

- **function to return partial minimum dominating set to cover *percent* fraction of the people using FastGreedy Algorithm**

```
Clear[minDominating];
minDominating[personIdLst_, locationIDLst_, connMat_, percentPeople_] :=
 Module[{people, locations, locDegPairLst, sortedLocDegPairLst,
   locDegPair, locIndex, locDeg, loc, percentLength},
  If[percentPeople < 0 || percentPeople > 1, percentPeople = 1];
  people = {};
  (* next 2 statements added *)
  locDegPairLst = Table[{j, degLocation[j, connMat]}, {j, Length[Transpose[connMat]]}];
  sortedLocDegPairLst = sortSecond[locDegPairLst];
  locations = {};
locDegPair = 1;
  percentLength = percentPeople * Length[personIdLst];
While[Length[people ] < percentLength,
 {locIndex, locDeg} = sortedLocDegPairLst[[locDegPair]];
 loc = locationIDLst[[locIndex]];
 locations = Union[locations, {loc}];
 people = Union[people, adjacentPeopleLst[locIndex, personIdLst]];
 locDegPair ++
   (*;
 Print[locIndex, " ",locDeg, " ", locations,  " ",people]*)
 ];
  {people, locations}
]
```

```
(* test *)
{people, locations} = minDominating[personIdLst, locationIDLst, connMat, 1]
```

```
Length[locations]
```

```
(* test with timing *)
start = TimeUsed[];
{people, locations} = minDominating[personIdLst, locationIDLst, connMat, 0.5]
finish = TimeUsed[];
finish - start
```

```
Length[locations]
```

```
(* test *)
{people, locations} = minDominating[personIdLst, locationIDLst, connMat, 0.75]
```

```
Length[locations]
```

People-to-people graph

- **function to generate connection matrix for a people-to-people graph**

```
Clear[personToPerson];
personToPerson[connMat_] := Module[{maxPersonID, connPeopleMat, i, loc, j},
  maxPersonId = Length[connMat];
  connPeopleMat = Table[0, {maxPersonId}, {maxPersonId}];
  (* go through every column of connMat *)
  Do[
   (* go down loc column looking for 1's *)
   Do[
    If[connMat[[i, loc]] == 1,
     (* for every 1, look through rest of loc column looking for 1's *)
     (* These people are adjacent *)
     Do[
      If[connMat[[j, loc]] == 1, connPeopleMat[[i, j]] = connPeopleMat[[j, i]] = 1],
      {j, i + 1, maxPersonId}]],
    {i, maxPersonId}],
   {loc, Length[Transpose[connMat]]}
  ];
  connPeopleMat
 ]

connPeopleMat = personToPerson[connMat]
```

- **degree distribution of people-to-people graph**

- **function to return the degree of a person node in people-to-people graph**

```
Clear[degPersonPPG];
degPersonPPG[connPeopleMat_, i_] := Count[connPeopleMat[[i]], 1]
```

- **list, *distribLst*, of degrees of each vertex**

```
distribLst = Table[degPersonPPG[connPeopleMat, i], {i, numPeople}]
```

- **counts of number of nodes with each degree and then plot of distribution**

```
tbl = Table[Count[distribLst, i], {i, Max[distribLst]}]

lp = ListPlot[tbl, PlotStyle → PointSize[0.02], PlotRange → {0, 250}]
```

- **fit function to data in tbl**

```
lp2 = ListPlot[tbl ^ - .4, PlotStyle → PointSize[0.02]]

ft = Fit[tbl ^ - .4, {x}, x]

pl = Plot[ft, {x, 0, 13}]

Show[lp2, pl]

Clear[f];
f[x_] := (0.0819161 x) ^ - (10.0 / 4)
```

```
pl2 = Plot[f[x], {x, 0, 13}, PlotRange → {0, 375}]
```

```
Show[pl2, lp]
```

- **average degree in people-to-people graph**

```
Mean[distribLst] // N
```

Clustering coeff in people-to-people graph

- **Function to return list of person indicess adjacent to person v in person-to-person graph**

```
Clear[adjacentPeople];
adjacentPeople[connPeopleMat_, v_] :=
 Flatten[Position[Transpose[connPeopleMat][[v]], 1]]
```

- **Function to return *True* if nodes are adjacent in person-to-person graph**

```
Clear[adjacentPeopleQ];
adjacentPeopleQ[connPeopleMat_, u_, v_] := connPeopleMat[[u, v]] == 1
```

- **Function to return the number of edges in a subgraph of person-to-person graph**

```
Clear[numPeopleEdges];
numPeopleEdges[connPeopleMat_, vertices_] := Module[{subMat, trans},
  subMat = connPeopleMat[[vertices]];
  trans = Transpose[subMat][[vertices]];
  Count[trans, 1, 2] / 2
 ]
```

- **function to return the clustering coefficient for a node**
  For a node with 0 or 1 adjacent nodes, return 0

```
Clearn[clusteringCoeff];
clusteringCoeff[connPeopleMat_, v_] := Module[{deg, conn, numerator, denominator},
  deg = degPersonPPG[connPeopleMat, v];
  If[deg < 2, 0,
   conn = adjacentPeople[connPeopleMat, v];
   numerator = numPeopleEdges[connPeopleMat, conn];
   denominator = deg! / (2.0 * (deg - 2)!);  (* floating point number of combinations *)
   numerator / denominator
  ]
 ]
```

- **average clustering coefficient**

```
Mean[Table[clusteringCoeff[connPeopleMat, v], {v, numPeople}]]
```