CODERS' CORNER

CENTRE FOR THE STUDY OF AFRICAN ECONOMIES

RANDOM DYADS IN SPARSE NETWORKS

How do you randomly generate dyads in sparse networks? Imagine you want to randomly group people together in pairs of two (e.g. job mentors with a mentee, tutors with a student, or workers with one of their friends), while considering certain conditions in your matching process (e.g. dyads with same interests, age groups or grades). Imagine you also want to randomly assign these pairs to treatment. Then you might want to check out the code below.

Let's say we have a pool of workers and information about up to two of their friends (in the same pool). Our overarching goal is to generate as many pairs as possible within our given network. We show possible ways to pair workers for two sub-goals:

GOAL #A: Generate dyads of friends

GOAL #B: Generate dyads of non-friends

We also show how to include additional conditions into the matching process. Finally, we explain how to randomly assign dyads to treatment groups.

This exercise is based on three assumptions:

1) Every worker has at least one friend, i.e. workers without friends are excluded from our sample.
2) Every worker can mention only up to two friends.
3) Friendships are symmetric, i.e. if A is mentioned as a friend of B, then B is also classified as a friend of A (even if A did not mention B as a friend).

We have the following basic variables in our sample:

ID:	*Unique identifier per person*
nbfriends:	*Number of friends of a person (either one or two)*
friendID1:	*Unique identifier of first friend*
friendID2:	*Unique identifier of second friend*
popular:	*Number of times a person is mentioned as a friend by someone else*

GOAL #A: GENERATE DYADS OF FRIENDS

In order to generate as many dyads of friends as possible, we first pair workers with only one friend and then workers with two friends (variable ‘nbfriends’ equals to one resp. two). Within those groups, we start with workers who have been mentioned least by someone else as a friend, i.e. we first pair the most unpopular workers and in the end pair the most popular workers (variable ‘popular’). As the number of possible matches for unpopular workers is lower than for popular workers, this approach allows us to come up with the highest number of possible matches in a sparse network. Depending on the density of the network, some workers will not be able to be matched.

Before we start, we generate the following new variables with all observations set to missing, which will be completed as part of the matching process.

```
gen D1          = .     // ID1 of dyad
gen D2          = .     // ID2 of dyad
gen done        = .     // Indicator equals to one if worker was successfully paired
gen F1done      = .     // Indicator equals to one if friend1 was successfully paired
gen F2done      = .     // Indicator equals to one if friend2 was successfully paired
gen dyad        = .     // Dyad ID
```

To determine the order in which workers will be randomly paired during the matching process, we generate random numbers as follows:

```
bys nbfriends popular: gen          rand = runiform()
bys nbfriends popular: egen         rank = rank(rand)
```

As described above, we start generating the first set of dyads for workers who have mentioned only one friend (nbfriends==1), starting with the least popular workers in the following loop:

```
su popular          if nbfriends == 1
forval i = `r(min)'/`r(max)' {
su popular          if nbfriends == 1 & popular == `i'

// i. Within each set of equally popular workers, we pursue the matching process in the randomly generated
order as determined by the variable 'rank'.
forval k = 1/`r(N)' {
su done             if rank==`k'

// ii. We only continue the loop if the current worker (with rank `k') has not been paired already as indicated
by the variable 'done'.
if r(mean)!=1 {

// iii. We replace D1 with the respective worker's ID and D2 with the friend's ID.
replace D1          = workerID          if rank==`k' & nbfriends==1 & popular==`i' & D1==. & F1done!=1 & done!=1
replace D2          = friendID1         if rank==`k' & nbfriends==1 & popular==`i' & D2==. & F1done!=1 & done!=1

// iv. We replace the variable 'done' with one to indicate that the respective worker has been successfully
paired with a friend.
replace done=1                          if rank==`k' & nbfriends==1 & popular==`i' & D1!=. & D2!=. & done!=1

// v. We replace D1 and D2 for the friend's observation as well.
gen D1_             = D1                if rank==`k' & nbfriends==1 & popular==`i' & done==1
egen maxD1          = max(D1_)
gen D2_             = D2                if rank==`k' & nbfriends==1 & popular==`i' & done==1
egen maxD2          = max(D2_)
replace D1          = maxD2             if maxD2==workerID
replace D2          = maxD1             if maxD2==workerID

//  vi. We need to ensure that the both D1 and D2 are not used any more as potential partners during the
matching process.
replace done        = 1                 if maxD2==workerID
replace F1done      = 1                 if maxD2==friendID1 | maxD1==friendID1
replace F2done      = 1                 if maxD2==friendID2 | maxD1==friendID2

// vii. We drop variables which will be continuously generated in the loop.
drop D1_ D2_ maxD1 maxD2
}
}
}
```

The above loop needs to be repeated for workers who have mentioned two friends (nbfriends==2). As we are now dealing with two possible friends to be paired with, we just need to modify step iii. of the loop as follows:

```
// iii. We first try to pair respective workers with their first friend and if that's not possible (because the first
friend has already paired with someone else, i.e. F1done==1) we pair respective workers with their second
friend.
replace D1          = workerID          if rank==`k' & nbfriends==1 & popular==`i' & D1==. & (F1done!=1  F2done!=1)
& done!=1
replace D2          = friendID1         if rank==`k' & nbfriends==1 & popular==`i' & D2==. & F1done!=1 & done!=1
```

```
=2 & popular==`i' & D2==. & F2done!=1 & done!=1
```

GOAL #B: GENERATE DYADS OF NON-FRIENDS

Imagine we now want to pair workers with co-workers who are not friends. We also want to pair them with someone they have not been paired with in the past (and not with themselves obviously).

We further add two general additional conditions:
- Workers must be paired only if they work in business unit one (cdt_one)
- Workers must be paired with co-workers with the same grade (cdt_two)

For clarity, we generate new variables for both conditions:

```
gen cdt_one       = business_unit
gen cdt_two       = grade
```

Before we start, we also generate the following new variables with all observations set to missing, which will be completed as part of the matching process.

```
gen D1            = .       // ID1 of dyad
gen D2            = .       // ID2 of dyad
gen done          = .       // Indicator equals to one if worker was successfully paired
gen dyad          = .       // Dyad ID
```

To determine the order in which workers will be randomly paired during the matching process (taking into account the conditions above), we generate random numbers as follows:

```
bys cdt_one cdt_two: gen random           = runiform()
bys cdt_one cdt_two: egen rank   = rank(random)
```

We first summarize the general conditions, and run the loop for each group of workers fulfilling these conditions.

```
su cdt_two                    if cdt_one==1
forval j=`r(min)'/`r(max)' {

// i. Within the set of equally graded workers in business unit one, we pursue the matching process in the
randomly generated order as determined by the variable 'rank'.
su rank                              if cdt_one==1 & cdt_two==`j'
forval i=1/`r(N)' {

// ii. We only continue the loop if the current worker (with rank `k') has not been paired already as indicated
by the variable 'done'.
su done                              if cdt_one==1 & cdt_two==`j' & rank==`i'
if r(mean)!=1 {

// iii. We generate variables for all IDs which the respective worker cannot be paired with (ID of first and
second friend, own ID, and ID of old partner)
gen noID_one_             = friendID1       if cdt_one==1 & cdt_two==`j' & rank==`i'
egen noID_one             = max(noID_one_)
gen noID_two_             = friendID2       if cdt_one==1 & cdt_two==`j' & rank==`i'
egen noID_two             = max(noID_two_)
gen noID_three_           = workerID        if cdt_one==1 & cdt_two==`j' & rank==`i'
egen noID_three           = max(noID_three_)
gen noID_four_            = partner_old     if cdt_one==1 & cdt_two==`j' & rank==`i'
egen noID_four            = max(noID_four_)

// iv. We generate a new rank for all possible partners, excluding IDs from step iii.
bys cdt_one cdt_two: gen random_partner          = runiform()       if workerID!=noID_one &
workerID!=noID_two & workerID!=noID_three & workerID!=noID_four & done!=1
bys cdt_one cdt_two: egen rank_partner  = rank(random_partner)
replace rank_partner                    = .                 if cdt_one!=1 | cdt_two!=`j'  |  workerID==noID_one
| workerID==noID_two | workerID==noID_three | workerID==noID_four | done!=1

// v. We select the partner with a rank of one. We replace D1 with the respective worker's ID and D2 with the
partner's ID.
gen D2_         = workerID      if rank_partner==1
egen maxD2      = max(D2_)
replace D1      = workerID      if cdt_one==1 & cdt_two==`j' & rank==`i'
replace D2      = maxD2         if cdt_one==1 & cdt_two==`j' & rank==`i'

// vi. We replace D1 and D2 for the partner's observation as well.
replace D1      = maxD2         if rank_partner==1
replace D2      = noID_three    if rank_partner==1

// vii. We need to ensure that the both D1 and D2 are not used any more as potential partners during the
matching process.
replace done=1  if rank==`i' & cdt_one==1 & cdt_two==`j' & D2!=.
replace done=1  if rank_partner==1 & D2!=.

// viii. We drop variables which will be continuously generated in the loop.
drop noID_one_ noID_one noID_two_ noID_two  noID_three_        noID_three  noID_four_ noID_four
random_partner rank_partner D2_ maxD2
}
}
}
```

Once the loop has been completed, we can finally generate a unique identifier per pair, for example:

```
replace dyad= 100*(D1 + D2) + min(D1, D2)
```

This code can be easily extended by additional conditions.

TREATMENT ASSIGNMENT

If we want to randomly assign dyads to treatment and control groups, we first need to generate a variable with the total number of all dyads. With strata variables, the size would have to be adjusted accordingly, e.g. bys strata: gen size=_N.

```
gen size        = _N
replace size    =  size/2

bys dyad: gen n  =_n == 1
sort dyad
```

We then generate a rank by dyad.

```
gen random_             = runiform()
replace random_         =. if n==1
egen rank_              =rank(random_)
bys dyad: egen random   = max(random_)
bys dyad: egen rank     =max(rank_)
```

If the size is even, we assign the first half to treatment, and the second half to control.

```
replace treat   = 1             if mod(size,2)==0 & rank<=size/2
replace treat   = 0             if mod(size,2)==0 & rank>size/2
```

If the size is odd, we assign the first half to treatment and the second half to control. The rank in the middle is randomly assigned to either treatment or control.

```
replace treat     = 1              if mod(size,2)==1 & rank<=int(size/2)
replace treat     = 0              if mod(size,2)==1 & rank>round(size/2)
gen randnb_       = runiform()     if mod(size,2)==1 & rank==int(size/2)+1
replace randnb_ = 0                if n==0
bys dyad: egen randnb=max(randnb_)

replace treat     = 1              if randnb>.5 & mod(size/2)==1 & rank==int(size/2)+1
replace treat     = 0              if randnb<.5 & mod(size/2)==1 & rank==int(size/2)+1
```

To be able to conduct randomization inference, it is important to re-randomize both the generation of dyads as well as the selection of treatment groups, and save re-randomization results, i.e. the composition of pairs and treatment assignments.

Vanessa Schreiber, DPhil Candidate in Economics, Oxford
03 March 2020