# condor_tutorial

August 5, 2020

## 1 CONDOR usage example

#### 1.0.1 Author:

Genís Calderer*.
*Kuijjer Lab (NCMM) - genis.calderer@gmail.com

### 1.1 Introduction

The condor method is an implementation of the brim algorithm for the analysis of bipartite networks. The purpose of this algorithm is to find a community structure in bipartite networks that takes into account the bipartite structure of the network as opposed to using the network as if it did not have an extra structure. This algorithm was first described in the paper "Modularity and community detection in bipartite networks" by Michael J. Barber." The python implementation of condor is based on the R version presented in the paper "Bipartite Community Structure of eQTLs" by John Platig , Peter J. Castaldi, Dawn DeMeo, John Quackenbush.

This guide will show how to use CONDOR using a toy network of pollinization between bee species and flower species. It is a small network but as we will see it has a quite well defined modularity structure.

### 1.2 1. Importing CONDOR from netZooPy

In order to use the SAMBAR functions it has to be imported from the netZooPy as follows:

[1]:
```
from netZooPy import condor
```

To check the parameters and information on the main condor functions type this:

[8]:
```
help(condor.condor_object)
help(condor.initial_community)
help(condor.brim)
help(condor.condor)
```

```
Help on function condor_object in module condor.condor:

condor_object(net)
    Initialization of the condor object. The function gets a network in edgelist
format encoded in a pandas dataframe.
    Returns a dictionary with an igraph network, names of the targets and
regulators, list of edges, modularity, and vertex memberships.
```

```
Help on function initial_community in module condor.condor:

initial_community(CO, method='LCS', project=False)
    Computation of the initial community structure based on unipartite methods.
    The implementation using bipartite projection is not yet available, but
project=False performs better modularity-wise (at least with the networks I
worked on).

Help on function brim in module condor.condor:

brim(CO, deltaQmin='def', c=25)
    Implementation of the BRIM algorithm to iteratively maximize bipartite
modularity.
    Note that c is the maximum number of communities. Dynamic choice of c is not
yet implemented.

Help on function condor in module condor.condor:

condor(filename, c=25, deltaQmin='def')
    Default settings run of condor with output of the membership dataframes.
    Reads a network in csv format and index_col=0.
```

## 1.3 2. Loading the network into a CONDOR object

To use the CONDOR method we first have to import a network's edgelist into a pandas dataframe.

[22]:
```
import pandas as pd
network = pd.read_csv("toynetwork.csv",index_col=0)
network.head(5)
```

[22]:
```
        pollinator                     plant  interactions
1  Adela.purpurea            Salix.fragilis            20
2  Adela.purpurea  Chamaedaphne.calyculata             0
3  Adela.purpurea     Nemopanthus.mucronata             0
4  Adela.purpurea    Andromeda.glaucophylla             0
5  Adela.purpurea          Kalmia.polifolia             0
```

We initialize the CONDOR object with the condor_object function:

[10]:
```
condor_object = condor.condor_object(network)
```

```
Weights detected
Condor object built in 0.0059773921966552734
```

The condor object contains several features associated to the network and once initialized is passed to the other condor functions:

[13]:
```
condor_object.keys()
```

[13]:
```
dict_keys(['G', 'tar_names', 'reg_names', 'index_dict', 'edges', 'modularity',
'reg_memb', 'Qcoms'])
```

## 1.4 3. Running CONDOR

The next step is computing the initial community structure. By default we use the Louvain method.

[14]:
```
condor_object = condor.initial_community(condor_object)
```

```
Initial community structure computed in  0.00099945068359375 . Modularity =
0.5253469286550467
```

The condor object now has a community structure associated but it is not specific for bipartite networks. We apply the brim algorithm to find the bipartite community structure.

[18]:
```
condor_object = condor.brim(condor_object,deltaQmin="def")
```

```
Matrices computed in 0.0019989013671875
0.5266669647502602
0.5266669647502602
```

The numbers in the output of this function show the bipartite modularity score for each iteration. The modularity of a bipartite network is a value from 0 to 1 that quantifies how well separated is the network into modules. A score of 0.52 is quite high.

## 1.5 4. Results

The resulting condor_object of the above process has the membership of the target and regulator nodes into the different communities that have been found.

[19]:
```
condor_object.keys()
```

[19]:
```
dict_keys(['G', 'tar_names', 'reg_names', 'index_dict', 'edges', 'modularity',
'reg_memb', 'Qcoms', 'tar_memb'])
```

For example if we want to see the membership of the *reg* nodes we can do it as follows:

[21]:
```
condor_object["reg_memb"].head(5)
```

[21]:
```
                        reg  com
0   Andromeda.glaucophylla    2
1       Aronia.melanocarpa    5
2     Calopogon.pulchellus    1
3  Chamaedaphne.calyculata    6
4      Gaylussacia.baccata    6
```

## 1.6 5. Running CONDOR from filename

We note that the guide above shows how to use the method step by step. There is also the possibility to run automatically the whole process starting only with the filename of the network's edgelist in csv format. This is done using the condor function, and it outputs the target and regulator memberships into csv files. This however allows less control on the parameters of the method.

[23]:
```
condor.condor("toynetwork.csv")
```

```
Weights detected
Condor object built in 0.004999399185180664
Initial community structure computed in  0.0009996891021728516 . Modularity =
0.5253469286550467
Matrices computed in 0.0009989738464355469
0.5266669647502602
0.5266669647502602
```

[23]:
```
'Runtime 0.1844480037689209s'
```

[ ]:
```

```