Appendix - A note on matricial ways to compute structural holes

October 28, 2022

1 A note on matricial ways to compute Burt's structural holes

This note is the supplementary material for the article "A note on matricial ways to compute Burt's structural holes", by Alessio Muscillo.

[ ]:
```
import networkx as nx
import numpy as np
import time
import matplotlib
import matplotlib.pyplot as plt
from scipy import sparse
```

[ ]:
```
nx.__version__
```

[ ]:
```
'2.6.3'
```

2 Algorithm for Effective Size

Notice: this algorithm only works for undirected, binary networks with no self-loops.

First, compute Borgatti's **redundancy** for each node i, that is:

$$\frac{2\ t_i}{d_i},$$

where d_i is i's degree and t_i is "the number of ties in the network (not including ties to ego)".

According to this paper's algorithm, the vector containing all nodes' redundancies is given by:

$$\mathbf{r} = (A^2 \odot A)\mathbf{1} \oslash Diag(A^2),$$

where A is the adjacency matrix, A^2 is its squared, $\mathbf{1}$ is the vector of all 1s, $Diag(A^2)$ is the vector of A^2 diagonal elements and $\odot$ and $\oslash$ are respectively the element-wise multiplication and division between vectors or matrices.

Let us define a function according to this paper's algorithm:

[ ]:
```
def eff_size(g):
    A = nx.to_scipy_sparse_matrix(g)
    n = nx.number_of_nodes(g)
```

```
    A = A - sparse.dia_matrix((A.diagonal(), [0]), shape=(n, n)) # eliminate␣
↪self-loops (if present)

    A_sq = A * A
    diag = A_sq.diagonal()
    n = nx.number_of_nodes(g)

    r = A_sq.multiply(A) * np.ones(n) / diag

    d = [d for _,d in g.degree]

    return dict(zip([name for name in g.nodes], d-r))
```

2.1 Compare results

[ ]:
```
g = nx.barabasi_albert_graph(100, 5)
```

[ ]:
```
dict_1 = eff_size(g)
dict_2 = nx.effective_size(g)
dict_1 == dict_2
```

[ ]: True

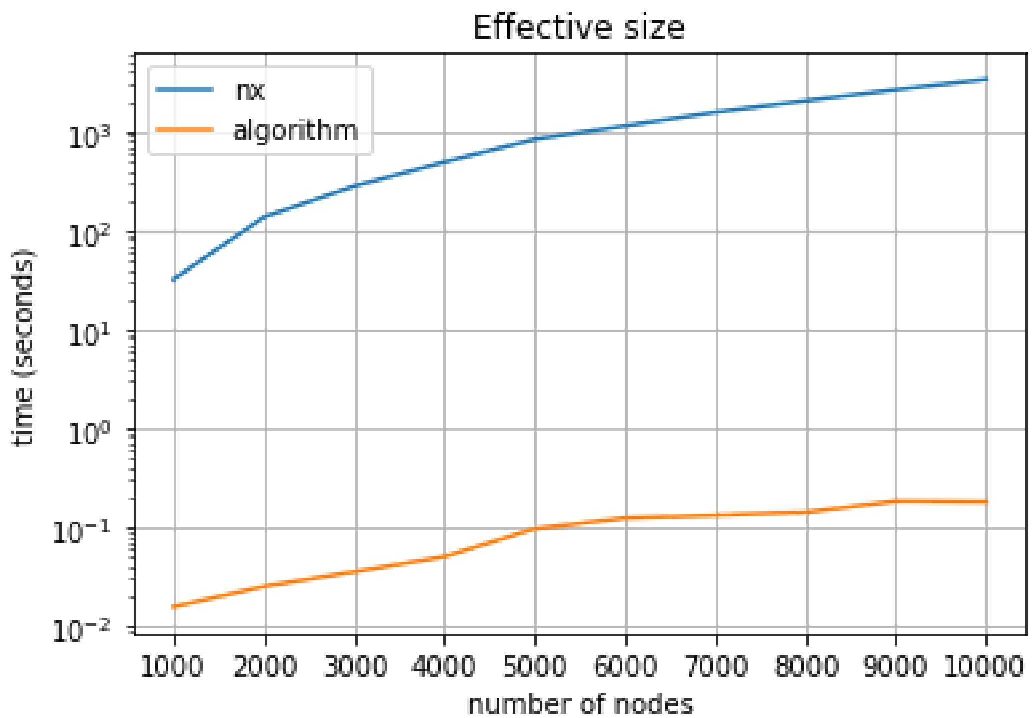
2.2 Compare computational speed

[ ]:
```
def compare_times_effective_size(g):
    start_time = time.time()
    eff_size(g)
    end_time = time.time()
    time_algorithm = end_time - start_time

    start_time = time.time()
    nx.effective_size(g)
    end_time = time.time()
    time_nx = end_time - start_time

    return time_algorithm, time_nx
```

[ ]:
```
list_n_times = []
for n in range(1000,11000,1000):
    g = nx.barabasi_albert_graph(n, 5)
    times = compare_times_effective_size(g)
    list_n_times.append([n, times])
```

[ ]:
```
plt.plot([n for [n,_] in list_n_times], [time_nx for [_,[_,time_nx]] in␣
↪list_n_times], label='nx')
```

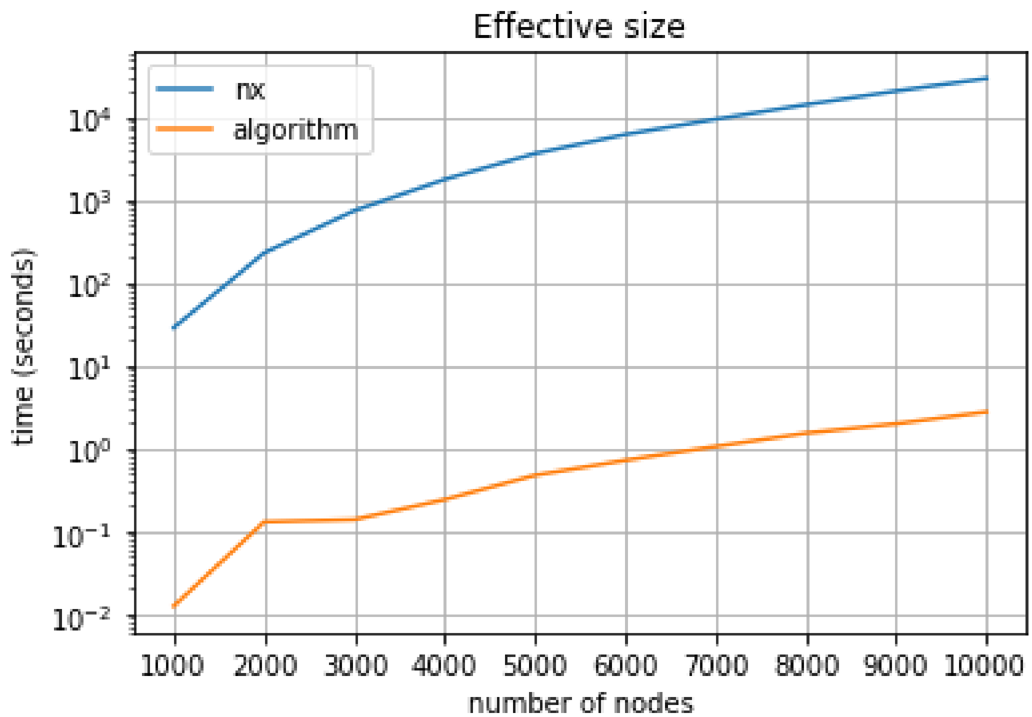
```
plt.plot([n for [n,_] in list_n_times], [time_algorithm for␣
↪[_,[time_algorithm,_]] in list_n_times], label='algorithm')
plt.grid()
plt.ylabel('time (seconds)')
plt.xlabel('number of nodes')
plt.title('Effective size')
plt.legend()
plt.xticks(range(1000,11000,1000))
plt.yscale('log')
plt.savefig('./../img/speed_effective_size_BA_networks.pdf',␣
↪bbox_inches='tight')
```



[ ]:
```
list_n_times = []
for n in range(1000,11000,1000):
    g = nx.erdos_renyi_graph(n, .01)
    times = compare_times_effective_size(g)
    list_n_times.append([n, times])
```

[ ]:
```
plt.plot([n for [n,_] in list_n_times], [time_nx for [_,[_,time_nx]] in␣
↪list_n_times], label='nx')
plt.plot([n for [n,_] in list_n_times], [time_algorithm for␣
↪[_,[time_algorithm,_]] in list_n_times], label='algorithm')
plt.grid()
```

```
plt.ylabel('time (seconds)')
plt.xlabel('number of nodes')
plt.title('Effective size')
plt.legend()
plt.xticks(range(1000,11000,1000))
plt.yscale('log')
plt.savefig('./../img/speed_effective_size_ER_networks.pdf',␣
 ↪bbox_inches='tight')
```



3 Algorithm for Local Constraint

The *local constraint* on i with respect to j, denoted ℓ_{ij}, is defined by

$$\ell_{ij} = \left(p_{ij} + \sum_{k \in N(i) \setminus \{j\}} p_{ik}\, p_{kj} \right)^2,$$

where $N(j)$ is the set of neighbors of j and p_{ij} is the *normalized mutual weight* of the edges joining i and j defined by

$$p_{ij} = \frac{a_{ij} + a_{ji}}{\sum_k (a_{ik} + a_{ki})}.$$

The algorithm to compute $L = (\ell_{ij})_{i,j}$ in matricial form is the following: 1. First, compute $\mathbf{x} = (A + A^T)\mathbf{1}$; 1. Then, invert every element of $\mathbf{x}$ computing $\mathbf{y} = \mathbf{1} \oslash \mathbf{x}$. 1. Then, compute $P = Diag(\mathbf{y}) \cdot (A + A^T)$ 1. Lastly, compute $L = [P + P(P \odot A)] \odot [P + P(P \odot A)]$.

```
[ ]: def local_constraint(g):
    A = nx.to_scipy_sparse_matrix(g)
    n = nx.number_of_nodes(g)
    A = A - sparse.dia_matrix((A.diagonal(), [0]), shape=(n, n)) # eliminate␣
↪self-loops (if present)

    ones = np.ones(n)

    x = (A + A.T) * ones
    y = ones / x

    diag_y = sparse.dia_matrix((y, [0]), shape=(n, n)) # diagonal matrix with␣
↪vector y on diagonal

    P = diag_y * (A + A.T)
    L_temp = P + P * P.multiply(A)

    return L_temp.multiply(L_temp)
```

3.1 Compare results with NetworkX's local constraint

Consider a random graph.

```
[ ]: g = nx.erdos_renyi_graph(50,.3)
```

Compute the local constraint using NetworkX routine. Make it a matrix where ℓ_{ij} contains the local constraint on node i with respect to node j.

```
[ ]: %%time
# create the matrix with local constraint from the package NetworkX
LC_nx = [[nx.local_constraint(g, i, j) for j in g.nodes] for i in g.nodes]
LC_nx = np.matrix(LC_nx)
#LC_nx
```

```
Wall time: 3 s
```

Now, compute the local constraint using the paper's algorithm. (The output is already given as a matrix.)

```
[ ]: %%time
LC = local_constraint(g)
LC = LC.todense()
#LC
```

```
Wall time: 2.97 ms
```

Compare the two results, to check that they both give the same results.

As an additional feature, it is worth noticing that the computational times are already quite different (the "new" algorithm being much faster).

[ ]:
```
# it must be False
# to check whether the two matrices are the same: if any of the local␣
↪constraints differed, it would return True
np.any(LC.round(decimals=5) != LC_nx.round(decimals=5))
```

[ ]:
```
False
```

4 Algorithm for Constraint

According to Everett, Borgatti 2020, the **constraint** for node i is

$$c_i = \sum_{j \in N(i)} \ell_{ij},$$

where ℓ_{ij} is the *local constraint* on i with respect to node j (as computed above). Notice that in our notation $N(i)$ does not include i itself. To be even more clear, one could then write:

$$c_i = \sum_{j \in N(i) \setminus \{i\}} \ell_{ij}.$$

One can re-write this as follows:

$$c_i = \sum_j \ell_{ij} a_{ij}.$$

(In case the network is directed, depending on whether one is considering the successors as neighbors or the predecessors, one has to use a_{ij} or a_{ji}. In case the network is weighted, then here the matrix A is the binary version of the weighted adjacency matrix W.)

So, the vector $\mathbf{c} = (c_i)_i$ containing the constraints of the network is obtained by summing the rows of the matrix $L \odot A$:

$$\mathbf{c} = [\mathbf{1}^T (L \odot A)]^T.$$

(Remember that in our notation vectors are always considered as columns.)

[ ]:
```
def constraint(g):
    A = nx.to_scipy_sparse_matrix(g)
    n = nx.number_of_nodes(g)
    A = A - sparse.dia_matrix((A.diagonal(), [0]), shape=(n, n)) # eliminate␣
↪self-loops (if present)

    L = local_constraint(g)

    C = (L.multiply(A)).sum(axis=1)
    C = np.squeeze(np.asarray(C)) # convert to type 'numpy array', for␣
↪readability

    return dict(zip([name for name in g.nodes], C)) # return a dict, as␣
↪NetworkX package does
```

4.1 Compare results with NetworkX's constraint

Compare the results (taking into account the minimal numerical differences due to computations).

[ ]:
```
g = nx.barabasi_albert_graph(100,3)
```

Chech whether the two results coincide:

[ ]:
```
decimals = 10
{key:value.round(decimals) for key,value in constraint(g).items()} == \
{key:round(value,decimals) for key,value in nx.constraint(g).items()}
```

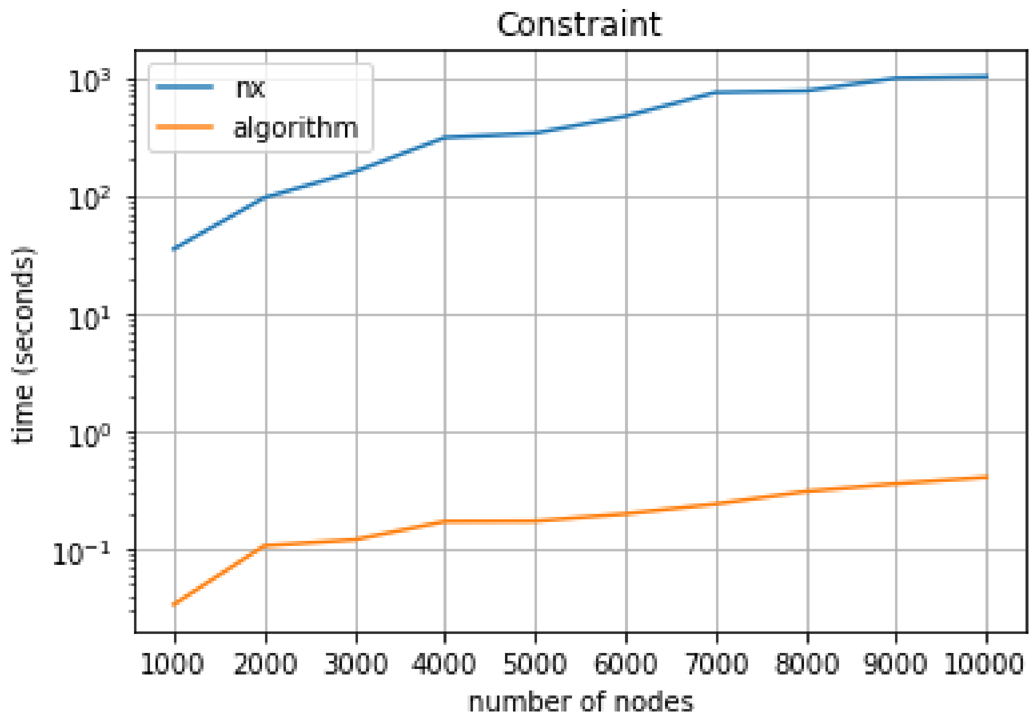
[ ]:
```
True
```

4.2 Compare computational speed

[ ]:
```
def compare_times_constraint(g):
    start_time = time.time()
    constraint(g)
    end_time = time.time()
    time_algorithm = end_time - start_time
    start_time = time.time()
    nx.constraint(g)
    end_time = time.time()
    time_nx = end_time - start_time
    return time_algorithm, time_nx
```

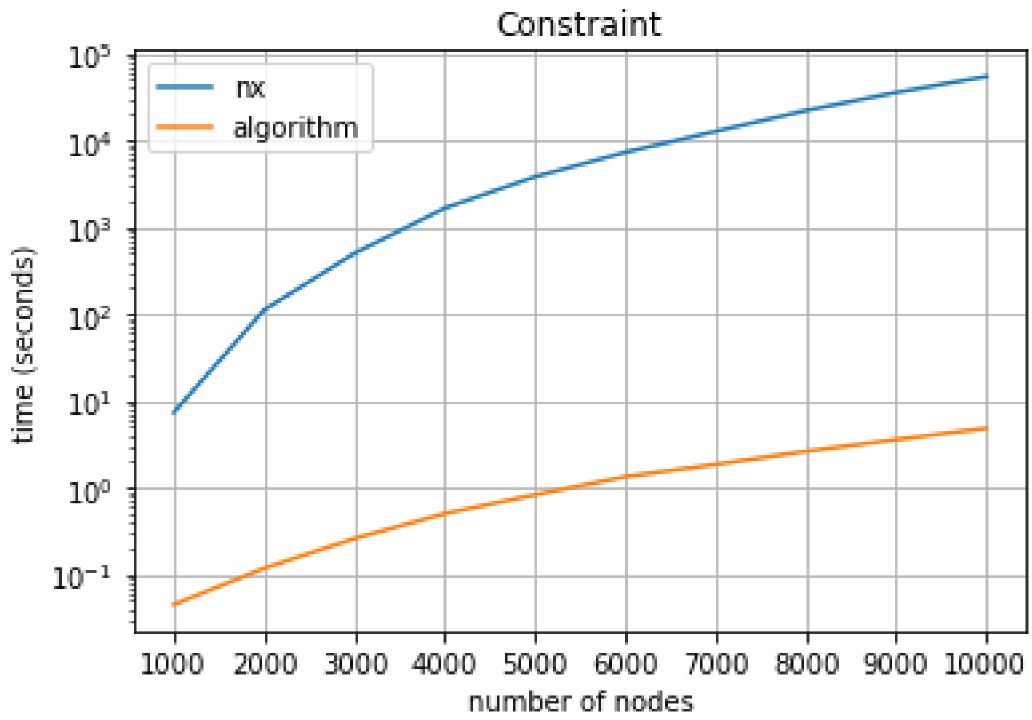
[ ]:
```
list_n_times_constraint = []
for n in range(1000,11000,1000):
    g = nx.barabasi_albert_graph(n, 5)
    times = compare_times_constraint(g)
    list_n_times_constraint.append([n, times])
```

[ ]:
```
plt.plot([n for [n,_] in list_n_times_constraint], [time_nx for [_,[_,time_nx]]␣
 ↪in list_n_times_constraint], label='nx')
plt.plot([n for [n,_] in list_n_times_constraint], [time_algorithm for␣
 ↪[_,[time_algorithm,_]] in list_n_times_constraint], label='algorithm')
plt.grid()
plt.ylabel('time (seconds)')
plt.xlabel('number of nodes')
#plt.title('Difference in computational speed for constraint\n(Barabasi-Albert␣
 ↪networks)')
plt.title('Constraint')
plt.legend()
plt.xticks(range(1000,11000,1000))
plt.yscale('log')
plt.savefig('./../img/speed_constraint_BA_networks.pdf', bbox_inches='tight')
```




```
[ ]: list_n_times_constraint = []
     for n in range(1000,11000,1000):
         g = nx.erdos_renyi_graph(n, .01)
         times = compare_times_constraint(g)
         list_n_times_constraint.append([n, times])
```

```
[ ]: plt.plot([n for [n,_] in list_n_times_constraint], [time_nx for [_,[_,time_nx]]␣
      ↪in list_n_times_constraint], label='nx')
     plt.plot([n for [n,_] in list_n_times_constraint], [time_algorithm for␣
      ↪[_,[time_algorithm,_]] in list_n_times_constraint], label='algorithm')
     plt.grid()
     plt.ylabel('time (seconds)')
     plt.xlabel('number of nodes')
     #plt.title('Difference in computational speed for constraint\n(Barabasi-Albert␣
      ↪networks)')
     plt.title('Constraint')
     plt.legend()
     plt.xticks(range(1000,11000,1000))
     plt.yscale('log')
     plt.savefig('./../img/speed_constraint_ER_networks.pdf', bbox_inches='tight')
```



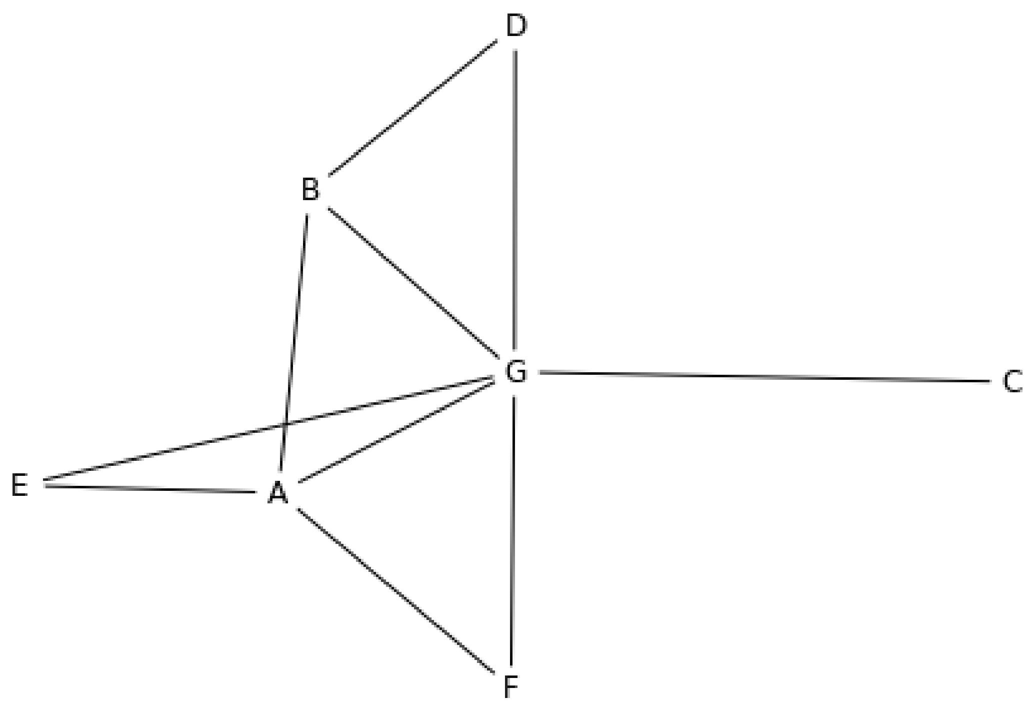
5 An example

Create Burt's and Borgatti's network.

[ ]:
```
g = nx.Graph()
g.add_nodes_from(['A','B','C','D','E','F','G'])
g.add_edges_from([('A','B'),('A','E'),('A','G'),('A','F')])
g.add_edges_from([('B','D'),('B','G')])
g.add_edges_from([('C','G')])
g.add_edges_from([('D','G')])
g.add_edges_from([('E','G')])
g.add_edges_from([('F','G')])
```

Let us draw this network:

[ ]:
```
nx.draw(g, with_labels=True, node_color='w')
```



Compute effective size with NetworkX algorithm:

```
[ ]: nx.effective_size(g)
```

```
[ ]: {'A': 2.5,
      'B': 1.6666666666666667,
      'C': 1.0,
      'D': 1.0,
      'E': 1.0,
      'F': 1.0,
      'G': 4.666666666666667}
```

Compute effective size with this paper's algorithm.

```
[ ]: eff_size(g)
```

```
[ ]: {'A': 2.5,
      'B': 1.6666666666666667,
      'C': 1.0,
      'D': 1.0,
      'E': 1.0,
      'F': 1.0,
      'G': 4.666666666666667}
```

Compute constraint with NetworkX algorithm:

```
[ ]: nx.constraint(g)
```

```
[ ]: {'A': 0.595486111111111,
      'B': 0.6427469135802467,
      'C': 1.0,
      'D': 0.7847222222222223,
      'E': 0.7309027777777779,
      'F': 0.7309027777777779,
      'G': 0.40027006172839497}
```

Compute constraint with this paper's algorithm:

```
[ ]: constraint(g)
```

```
[ ]: {'A': 0.595486111111111,
      'B': 0.6427469135802467,
      'C': 1.0,
      'D': 0.7847222222222223,
      'E': 0.7309027777777779,
      'F': 0.7309027777777779,
      'G': 0.400270061728395}
```

6 Alternative version of (Local) Constraint

The definition of local constraint, that is,

$$\ell_{ij} = \left(p_{ij} + \sum_{k \in N(i) \setminus \{j\}} p_{ik} p_{kj} \right)^2 ,$$

is what one finds in Everett, Borgatti (2020) and also in R's package igraph.

Notice that this is slightly different from NetworkX's definition of local constraint, which instead is:

$$\ell_{ij} = \left(p_{ij} + \sum_{k \in N(j)} p_{ik} p_{kj} \right)^2 .$$

This requires a small modification to what done above. In particular, in the paper one has to change equation

$$\sum_{k \in N(i)} p_{ik} p_{kj} = \sum_k a_{ik} p_{ik} p_{kj}$$

with

$$\sum_{k \in N(j)} p_{ik} p_{kj} = \sum_k p_{ik} p_{kj} a_{kj} ,$$

so that now the matricial form obtained is

$$P(P \odot A)$$

instead of $(A \odot P)P$.

In matricial form, this is a slight modification of what we have written above, according to the following algorithm: 1. First, compute $\mathbf{x} = (A + A^T)\mathbf{1}$; 1. Then, invert every element of $\mathbf{x}$ computing $\mathbf{y} = \mathbf{1} \oslash \mathbf{x}$. 1. Then, compute $P = Diag(\mathbf{y}) \cdot (A + A^T)$ 1. Lastly, compute $L = [P + (A \odot P)P] \odot [P + (A \odot P)P]$.

[ ]:
```
def local_constraint_alternative(g):
    n = nx.number_of_nodes(g)
    A = nx.to_scipy_sparse_matrix(g)
    ones = np.ones(n)

    x = (A + A.T) * ones
    y = ones / x

    diag_y = sparse.dia_matrix((y, [0]), shape=(n, n))

    P = diag_y * (A + A.T)
    L_temp = P + (A.multiply(P) * P)

    return L_temp.multiply(L_temp)
```

The computation of *constraint* remains unaltered.

[ ]:
```
def constraint_alternative(g):
    A = nx.to_scipy_sparse_matrix(g)
    L = local_constraint_alternative(g)

    C = (L.multiply(A)).sum(axis=1)
    C = np.squeeze(np.asarray(C)) # convert to type 'numpy array', for␣
↪readability

    return dict(zip([name for name in g.nodes], C)) # return a dict, as␣
↪NetworkX package does
```

Applying it to the network of the Example gives:

[ ]:
```
constraint_alternative(g)
```

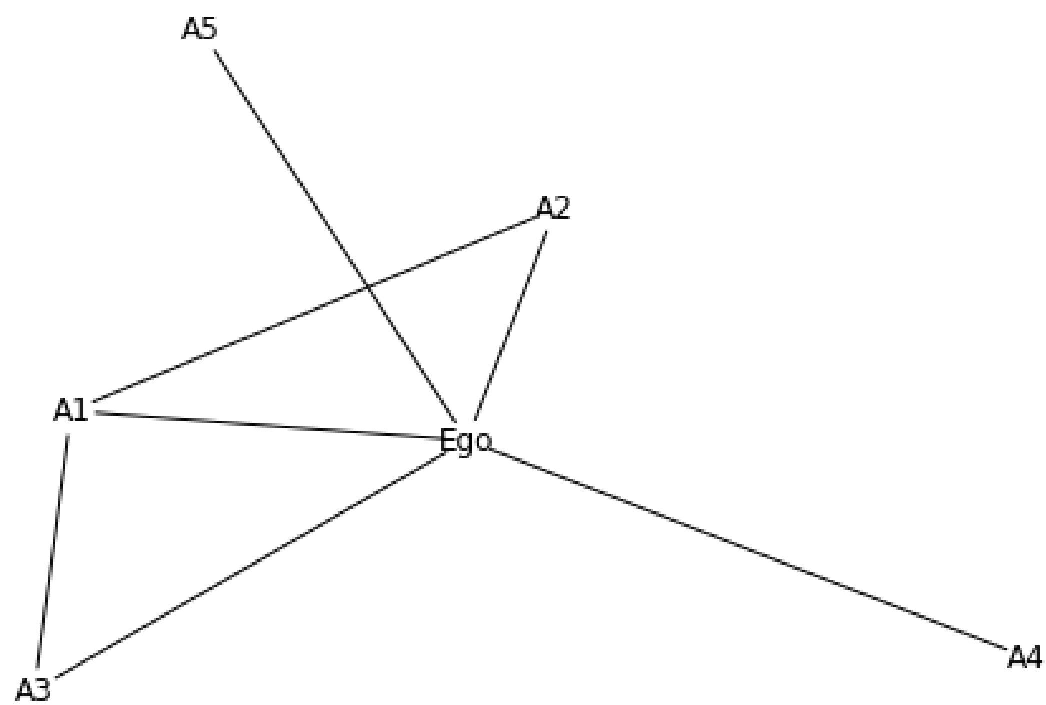
[ ]:
```
{'A': 0.5954861111111111,
 'B': 0.6427469135802467,
 'C': 1.0,
 'D': 0.7847222222222223,
 'E': 0.7309027777777779,
 'F': 0.7309027777777779,
 'G': 0.40027006172839497}
```

7 Example in Everett, Borgatti (2020)

Consider the networks in Figure 1 of Everett, Borgatti (2020) and also in R's package igraph

[ ]:
```
g = nx.Graph()
g.add_nodes_from(['Ego','A1','A2','A3','A4','A5'])
g.
 ↪add_edges_from([('Ego','A1'),('Ego','A2'),('Ego','A3'),('Ego','A4'),('Ego','A5')])
g.add_edges_from([('A1','A2'),('A1','A3')])
```

[ ]:
```
nx.draw(g, with_labels=True, node_color='w')
```



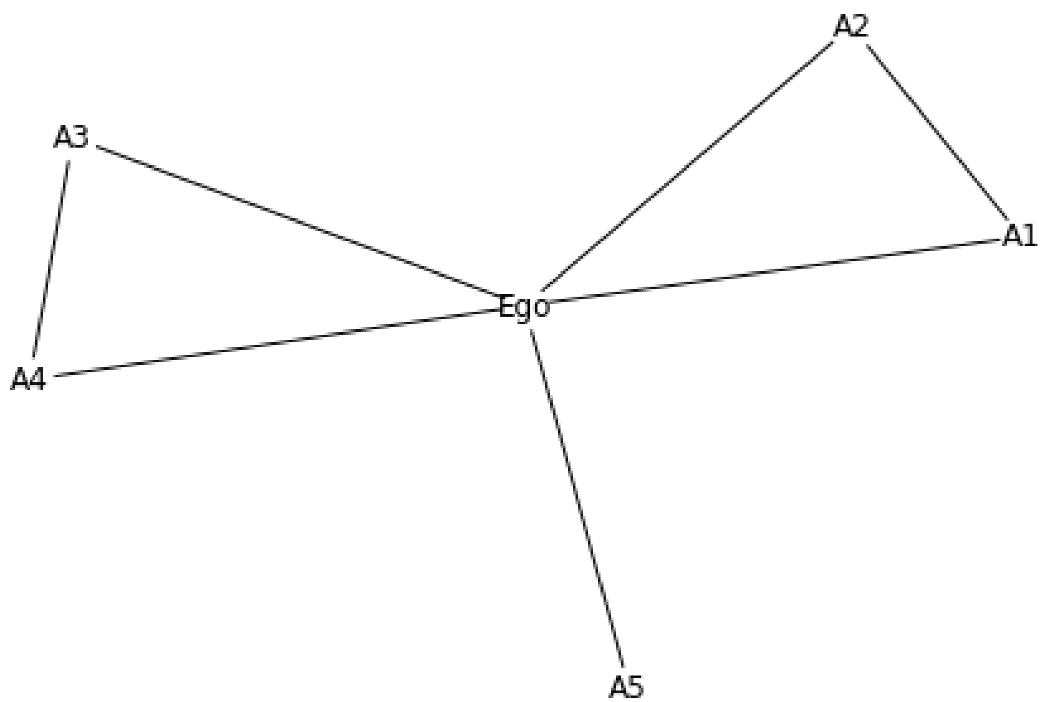
[ ]:
```
constraint(g)
```

[ ]:
```
{'Ego': 0.38222222222222224,
 'A1': 0.7644444444444444,
 'A2': 0.8044444444444444,
 'A3': 0.8044444444444444,
 'A4': 1.0,
 'A5': 1.0}
```

[ ]:
```
g = nx.Graph()
g.add_nodes_from(['Ego','A1','A2','A3','A4','A5'])
```

```
g.
↪add_edges_from([('Ego','A1'),('Ego','A2'),('Ego','A3'),('Ego','A4'),('Ego','A5')])
g.add_edges_from([('A1','A2')])
g.add_edges_from([('A3','A4')])
```

[ ]:
```
nx.draw(g, with_labels=True, node_color='w')
```



[ ]:
```
constraint(g)
```

[ ]:
```
{'Ego': 0.4000000000000001,
 'A1': 0.9225,
 'A2': 0.9225,
 'A3': 0.9225,
 'A4': 0.9225,
 'A5': 1.0}
```

8 Improved Structural Holes

Here we compute in matricial form the Improved Structural Hole measure proposed in Yu et al. 2017.

Maintaining a notation consistent with the one used above, let $A = (a_{ij})_{i,j} \in \mathbb{R}^{n \times n}$ be the adjacency matrix of an undirected unweighted graph.

Let us define

$$W = A \odot (A\mathbf{1}\mathbf{1}^\top + \mathbf{1}\mathbf{1}^\top A^\top)$$

Then, the matrices

$$P = W \oslash (W1)$$

and also

$$B = P + A \odot P^2$$

and, lastly, the vector of the ISH:

$$\mathbf{k} = (B \odot B)\mathbf{1}.$$

[ ]:
```
def improved_structural_holes(g):
    A = nx.to_scipy_sparse_matrix(g)
    n = nx.number_of_nodes(g)
    A = A - sparse.dia_matrix((A.diagonal(), [0]), shape=(n, n)) # eliminate␣
↪self-loops (if present)

    ones_v = np.ones((n,1))
    ones_m = np.ones((n,n))

    W = A.multiply(A * ones_v + ones_v.T * A) # short for: A * ones_v * ones_v.
↪T + ones_v * ones_v.T * A
    P = W / (W * ones_m)
    B = P + A.multiply(P*P)
    c = np.multiply(B,B) * ones_v
    return dict(zip([name for name in g.nodes], np.array(c).flatten())) #␣
↪return a dict, as NetworkX package does
    # return c
```

[ ]:
```
# Example Everett, Borgatti (2020)
g = nx.Graph()
g.add_nodes_from(['A','B','C','D','E','F','G'])
g.add_edges_from([('A','B'),('A','E'),('A','G'),('A','F')])
g.add_edges_from([('B','D'),('B','G')])
g.add_edges_from([('C','G')])
g.add_edges_from([('D','G')])
g.add_edges_from([('E','G')])
g.add_edges_from([('F','G')])
```

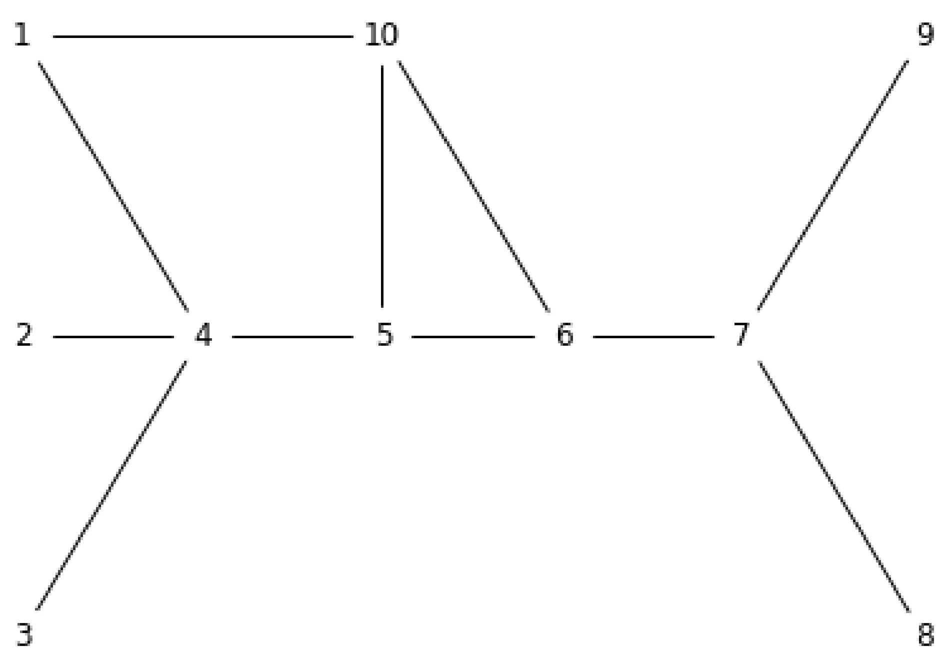
[ ]:
```
improved_structural_holes(g)
```

[ ]:
```
{'A': 0.6982950326385013,
 'B': 0.7457924159185401,
 'C': 1.0,
 'D': 0.8541488226059655,
 'E': 0.8119595234050814,
 'F': 0.8119595234050814,
 'G': 0.3835730648257023}
```

8.1 Example of Yu et al. 2017

[ ]:
```
g = nx.Graph()
g.add_node('1', pos=(1,3))
g.add_node('2', pos=(1,2))
g.add_node('3', pos=(1,1))
g.add_node('4', pos=(2,2))
g.add_node('5', pos=(3,2))
g.add_node('6', pos=(4,2))
g.add_node('7', pos=(5,2))
g.add_node('8', pos=(6,1))
g.add_node('9', pos=(6,3))
g.add_node('10', pos=(3,3))
g.add_edges_from([('1','4'),('1','10')])
g.add_edges_from([('2','4')])
g.add_edges_from([('3','4')])
g.add_edges_from([('4','5')])
g.add_edges_from([('5','6'),('5','10')])
g.add_edges_from([('6','7'),('6','10')])
g.add_edges_from([('7','8'),('7','9')])
```

[ ]:
```
pos = nx.get_node_attributes(g,'pos')
nx.draw(g, pos, with_labels=True, node_color='w')
```



[ ]:
```
constraint(g)
```

[ ]:
```
{'1': 0.5,
 '2': 1.0,
 '3': 1.0,
 '4': 0.25,
 '5': 0.5061728395061729,
 '6': 0.5061728395061729,
 '7': 0.3333333333333333,
 '8': 1.0,
 '9': 1.0,
 '10': 0.5061728395061729}
```

[ ]:
```
improved_structural_holes(g)
```

[ ]:
```
{'1': 0.5041322314049586,
 '2': 1.0,
 '3': 1.0,
 '4': 0.2551984877126654,
 '5': 0.4955573234671088,
 '6': 0.5068613073386433,
 '7': 0.346938775510204,
 '8': 1.0,
 '9': 1.0,
 '10': 0.5236223868722982}
```

[ ]:
```

```