Modularity Network Clustering Using the Louvain Algorithm with the Gravity Null Models

MMSC Special Topic Report for Python for Scientific Computing

Candidate Number: 1048531

1 Introduction

Classical community detection methods and network measures may return trivial results when applied to spatial networks [5]. Other network metrics such as the clustering coefficient and degree distribution are also found to produce uninformative results due to the effect of space on clustering and the additional costs associated with long distances links [5]. An example of this spatial bias is shown for a global cargo ship network in Figure 1. This has obvious ramifications for any researchers who wish to study networks in which space plays a role in link formation as useful information contained in the network may be masked by this spatial bias. Being able to identify communities of functionally related nodes in space has important practical implications. For example, it can be used to guide the construction of administrative and economic boundaries, based on empirical evidence of passenger and trade flows [5] or assess a node's integration into the network, which may have economic and cultural implications for it, in the case where nodes represent ports or other forms of regional infrastructure [16, 8, 7].

For this project, we will focus on the classical community detection measure known as modularity. Modularity is extremely effective at finding assortative subgroups of nodes in networks, that is subgroups of nodes that are densely connected while the rest of the network is sparse [5]. It relies on the optimisation of a simple quality function that compares the observed network to an expected network based on assumptions of how the network may have been constructed. It is also possible to include what is known as a resolution parameter which weights the positive and negative terms in the quality function and allows users to 'zoom' in and out of different resolution partitions, identifying fewer, larger partitions or more smaller ones [9]. Here, we will limit our analysis to undirected networks, though it has been shown that symmetrising directed networks does not affect the outcomes of community detection [10] so technically this extends to directed networks too. This project aims to modify an existing Python package which implements the Louvain method for modularity optimisation, and extend its functionality to spatial modularity.

Various methods of modularity optimisation have been studied before. In 2011, Expert et al. [5] produced a paper that modified the classical Newman-Girvan modularity for spatial effects. They defined a data-derived distance cost function which

they appended to the null model. Many variations of this function have been studied [14, 15, 3, 12]. The question of how to validate any of these methods once developed has also been addressed in the literature. Generally, what are known as benchmarking models, synthetic graphs with known partitions, have been developed and these are used for validation by comparing predicted and known partitions using similarity measures such as the Jaccard index or normalised mutual information (NMI).

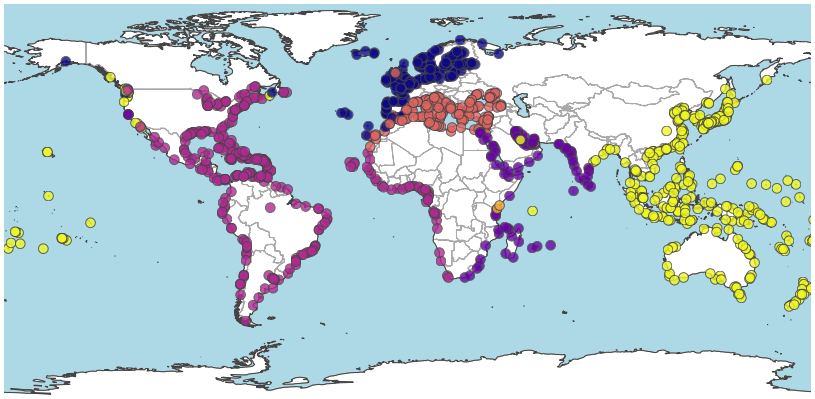


Figure 1: World cargo shipping network from AIS ship tracking data[1]. Node colours correspond to communities assigned by the `python-louvain` package by Thomas Aynaud[2]. Communities show clear bias towards geographic location. If there is any other community structure in the network, it is masked by this.

The report will be organised as follows: first, we will overview the main mathematical theory which underpins this work in Section 2. Here, we will discuss the formulation of the modularity score and methods for its optimisation. We will follow this by introducing spatial modularity and how it would be implemented computationally. Finally, we discuss the construction of synthetic benchmarking networks for validating any results. In Section 3 we descibe how we went about implementing this model in Python and integrated it into an existing community detection package. Finally, we give a very brief overview of some preliminary results before we discuss issues with the project and future directions for improvement in the Conclusion in Section 4.

[1] https://github.com/jasperverschuur/Port_Disruption_database
[2] https://github.com/taynaud/python-louvain

2 Mathematical Theory

We will first layout some of the notation that will be used throughout this report. An undirected network or graph is defined as a tuple $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_N\}$ is the set of vertices or nodes and $E = \{e_{11}, e_{12}, ..., e_{NN}\} = \{(v_1, v_1), (v_1, v_2), ..., (v_N, v_N)\}$ is the set of links or edges between them. We define the cardinalities of these sets as $N = |V|$ and $L = |E|$. The edges have assigned weights w_{ij} for each e_{ij}. The inherent meaning of these weights is generally arbitrary, and can be defined to suit many different purposes. Additionally, G undirected requires that the existence of e_{ij} implies the existence of an identical e_{ji}. We define the set of neighbours of a node v_i by $\eta_i = \{v_j | \exists (v_i, v_j) \in E\}$. The degree of node v_i is given by k_i where $k_i = \sum_{v_j \in \eta_i} w_{ij} = |\eta_i|$ and self-loops are counted twice. For an undirected network, summing the degrees of each node v_i yields $\sum_i k_i = 2M$, and we define M to be the total weight of the network. This sum is known as the handshaking lemma [9].

A graph is usually viewed in the form of an adjacency matrix, where each edge is represented by assigning $A_{ij} = A_{ji} = w_{ij}$. In the context of spatial networks, this may also be called a flow matrix. In this particular case, the weights are then understood to explicitly represent flow volumes between different spatial locations.

For a given graph, we define a partition $\mathcal{P} = \{C_1, C_2, ..., C_K\}$ of its vertices. For this form of community detection, communities do not overlap and the union of all communities is V. We assign names to two types of partition; firstly when all nodes are assigned to one community $C = V$ we denote this the *trivial partition* and secondly when each node is assigned its own individual community, $v_1 \in C_1, v_2 \in C_2, ..., v_N \in C_n$ we denote this the *singleton partition*.

A relevant and slightly confusing subtlety that is worth highlighting is that, by convention, a single self-loop edge of weight one will add two to a node's degree. For this reason, when calculating measures such as modularity by hand, self-links of weight w_{ii} will often be represented on the diagonal as $2w_{ii}$. In NetworkX, the package that we use for the majority of this work, this is not done and the weights of self-loops along the diagonal of the graph's adjacency matrix are not doubled. However the effect of the self-loop on a node's degree is accounted for, and self-loops add two to a node's degree. For this reason, in this report we will work with adjacency matrices as they

are represented in NetworkX but bearing in mind that $\sum_{ij} A_{ij} = \sum_{i \neq j} A_{ij} + 2\sum_i A_{ii}$. This also means that calling a node's degree using the NetworkX built-in function `graph.degree()[node]` is no longer equivalent to summing the rows or columns of the adjacency matrix. This convention is only for adjacency matrices and any other matrices used in this report may be summed as normal.

2.1 Optimisation of Classical Modularity

The classic modularity optimisation problem aims to find groups of nodes such that the nodes within the groups are most densely connected to eachother and the connections between different groups are sparse [9, 13]. The modularity statistic, Q, sums the intra-group weights in a network and values their significance versus some null model. Optimising the modularity is equivalent to finding a community arrangement that maximises the in-group density versus what would be expected according to this null model. The classic modularity problem uses the Newman-Girvan null model [6] in which the probability of a link existing between two nodes in a network is proportional to the product of their weights, balanced by the total weight of all edges in the network. We focus exclusively on the undirected case for this report but explicit extensions to the directed case are covered in the literature [11]. For two nodes, v_i and v_j, with respective degrees k_i and k_j, the expected number of edges between them is $\frac{k_i k_j}{2M}$ where $M = \sum_i k_{ij} = \sum_j k_{ij}$ is the total weight or flow of the network [9]. Considering a partition $\mathcal{P}$ of our network, we thus form the classic modularity equation

$$Q = \frac{1}{2M} \sum_{C \in \mathcal{P}} \sum_{i,\, j \in C} \left(A_{ij} - \frac{k_i k_j}{1M} \right).$$

A key feature of this modularity measure is that the null model preserves the total flow of the original network. Given the trivial partition of all nodes in one large community then $Q = 0$. There is also the option to include a resolution parameter, γ in the form

$$Q = \frac{1}{2M} \sum_{C \in \mathcal{P}} \sum_{i,\, j \in C} \left(\gamma A_{ij} - \frac{k_i k_j}{1M} \right). \tag{1}$$

Increasing this parameter will prefer fewer, larger communities and vice versa. The modularity measure can now be generalised by considering different null models. We

denote a general null model representing the expected weight of the links between nodes as P_{ij}. From this, we can write a generalised modularity measure as

$$Q = \frac{1}{2M} \sum_{C \in \mathcal{P}} \sum_{i,\, j \in C} \left(\gamma A_{ij} - P_{ij} \right). \tag{2}$$

Optimising the modularity function has been proven to be NP-hard [9] and many algorithms for finding approximate solutions have been developed, and shown to be sufficient. The predominant two methods are spectral optimisation, which uses eigenvector approximations to divide the network into successively smaller partitions and the Louvain method (named after the university its authors attended at the time) which is a greedy agglomerative method [9, 1]. The development of Louvain algorithm allowed modularity optimisation to be performed on networks of up to 118 million nodes and 1 billion links [1]. The method "unfolds" a hierarchical community structure for networks in the form of a dendogram, where the last entry of the dendogram corresponds to the largest communities. It is this algorithm that we will focus on for the rest of this project.

2.2 The Louvain Algorithm

The Louvain algorithm consists of two phases and operates as follows:

2.2.1 Phase 1

Each node in the network is initially assigned to its own community, this is called the singleton partition $\mathcal{P}_0$. The modularity of this partition is calculated according to Equation 1. Then each node in the network is (randomly) selected, and the modularity change for reassigning it to a neighbouring community is calculated. If moving the node to a different community leads to a strict modularity increase then the node is moved to that community, otherwise, it remains where it is. An iteration is considered to be complete once every node in the network has been considered at least once. The algorithm will generally perform multiple iterations until the modularity stops increasing, or its increase is below a certain threshold. If the algorithm returns the singleton partition, then we consider it finished, otherwise we begin the second phase [9].

2.2.2 Phase 2

The second phase of the Louvain algorithm agglomerates each community of nodes into what we now call a meta-node. Meta-links between meta-nodes become the weighted sum of all edges between nodes in those communities and the meta-nodes have self-loops which are the sum of all intra-community links. This meta-network is then passed back to phase 1 to perform more community detection.

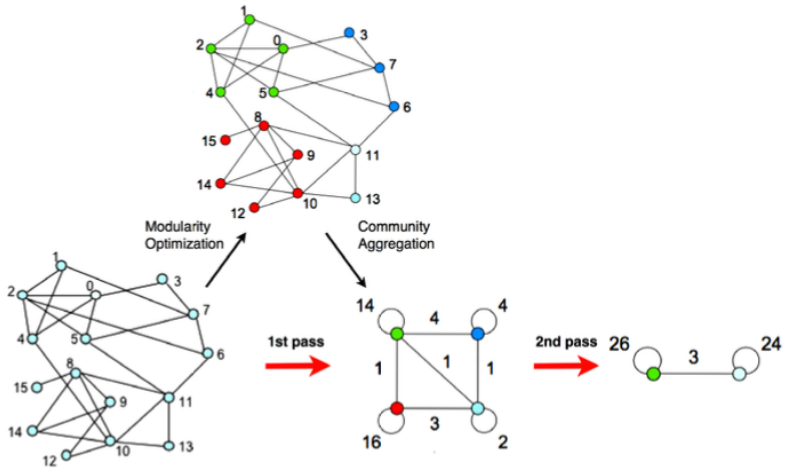


Figure 2: Diagram showing the two phases, or one pass, of the Louvain algorithm, taken from Blondel et al.[1]. Phase 1 shows the nodes being assigned to communities (colours) and phase 2 shows all communities being agglomerated into meta-nodes. (Here, the authors use 'pass' to describe what we call a 'phase'.)

2.2.3 Computation

When computing the change in modularity in phase 1, it is unnecessary to compute the full modularity sum for each possible node re-assignment. Instead, the cost of removing or inserting a node from a community C can be given by [1]

$$\Delta Q^{-} = \frac{1}{2M}\left(-2k_{i,\text{in}} + 2k_i\frac{\Sigma_{\text{total}}}{2M}\right) \tag{3}$$

and

$$\Delta Q^{+} = \frac{1}{2M}\left(2k_{i,\text{in}} - 2k_i\frac{\Sigma_{\text{total}}}{2M}\right). \tag{4}$$

Here, $k_{i,\text{in}}$ is the sum of all weights linking node v_i to community C and Σ_{total} is the sum of the degrees of all nodes in community C, $\sum_{i,\,j\in C} k_{ij}$, including v_i. Using Equations (3) and (4) the algorithm can efficiently cycle through all possible community assignments for a node and calculate the modularity changes for each assignment.

In the second phase, when communities are gathered into meta-nodes a new adjacency matrix $\tilde{A}$ may be calculated by summing (bearing in mind the weighting of self-loops) the entries of the adjacency matrix corresponding to intra-community links. Deriving the new null matrix from this $\tilde{A}$ should result in the same matrix as would be obtained by summation (without worrying about self-loops) of the relevant entries in the null model. The modularity of the graph with partititon $\mathcal{P}$ should be equal to the modularity of the graph post-agglomeration. This is a useful sanity check that will be used later to verify that phase 2 is being implemented correctly.

Each time both phases of the Louvain algorithm are completed this is known as a pass. The algorithm usually continues for multiple passes until the increase in modularity between subsequent passes falls below some minimum threshold. The result is a dendogram corresponding to a hierarchical community structure which contains a partition of the network for each pass on each level. The final partition contains the largest communities and is then chosen as the final partition of the network.

2.3 Spatially-Corrected Modularity

We now direct our attention to the main focus of this project: spatial networks. In this context, convention renames the adjacency matrix to a flow matrix, and edge weights no longer have an arbitrary meaning but are understood to measure the magnitude of, or the number of flows between any two vertices. The classical modularity community detection method works extremely well in networks where nodes do not have locations in space or, if they do, the spatial locations have no bearing on link formation [3]. For many spatial networks, however, this is not true and this may render the results of community detection algorithms that don't account for space as trivial [5]. In particular, algorithms may isolate groups of spatially close nodes and thus mask any underlying community structure related to non-spatial node attributes [5]. The development of spatially-corrected community detection methods is an active field of research. In 2011, a paper co-authored by two of the original authors of the Louvain algorithm developed a variation of the modularity function based on the gravity model [5]. In its simplest form, the expected flow between two vertices v_i and v_j is given by the equation

$$P_{ij} = k_i k_j f(d_{ij}). \tag{5}$$

Here, the node degrees k_i and k_j are used as estimates of a node's importance in the network, generalisations of this to other measures of importance such as population potential, trade volume have previously been considered. Replacing the null model P_{ij} in Equation 2. with this gravity model we arrive at the spatial modularity function

$$Q^{\text{gravity}} = \frac{1}{2M} \sum_{C \in \mathcal{P}} \sum_{i,\, j \in C} \left(\gamma A_{ij} - k_i k_j f(d_{ij})\right).$$

In order for this model to produce results that are on par with the Newman-Girvan modularity it's important that the total flow in the null model is the same as that in the spatial network. This can be ensured by introducing a weighting K such that $\sum_{i,j} K P_{ij} = \sum_{i,j} A_{ij} = 2M$, i.e. $K = 2M / \sum_{i,j} P_{ij}$. Using this, we finally arrive at the final form of the gravity modularity measure

$$Q^{\text{gravity}} = \frac{1}{2M} \sum_{C \in \mathcal{P}} \sum_{i,\, j \in C} \left(\gamma A_{ij} - K k_i k_j f(d_{ij})\right) \tag{6}$$

where $K = 2M / \sum_{i,j} P_{ij}$.

Various forms for the distance decay function $f(d)$ have been discussed. Expert et al. [5] computes a weighted average for the probability that a link exists based on the data. This avoids needing to estimate arbitrary parameters but adds an additional layer of computational complexity: a binning procedure must be carried out to discretise the set of distances. Additionally, if space and other relevant attributes are completely uncorrelated, this algorithm is optimal, but as noted by Cerina et al. [3], this does not account for the possibility of there being some unknown level of correlation between space and other relevant attributes. Cerina et al. choose an exponential distance decay function of

$$f(d) = \mathrm{e}^{-d/\ell} \tag{7}$$

where ℓ is the average distance in the network which they claim is sufficient to capture the spatial signature of the network [3]. Another popular choice is the simple inverse power function

$$f(d) = \mathrm{e}^{-d\ell}. \tag{8}$$

The paper [14] explores a variety of values of ℓ for both Equations (7) and (8). Both [14, 15] assessed their methods' performance based on the average modularity score

obtained versus that of the Newman-Girvan modularity. In general, using $\ell = 2$ with Equation (8) and $\ell = 1$ with Equation (7) resulted in the highest modularity scores [14]. This is not, however, necessarily a good way of measuring a model's performance, as is noted by Expert et al. in [5]. The value of modularity is rather meaningless and decreases when the null model better describes the data. It is better to validate these methods through the construction of synthetic spatial benchmark networks with a known community structure. The predicted and true partitions may then be compared using various similarity measures such as the Jaccard index or the normalised mutual information score (NMI) as in [5, 3, 12]. The construction of these networks will be described in Section 2.5.

The cost of removing or inserting a node from a community, as given in Equations 3-4 now becomes

$$\Delta Q^{\text{gravity},-} = \frac{1}{2M}\left(-2k_{i,\text{in}} + 2Kk_i \sum_j k_j f(d_{ij})\right) \tag{9}$$

and

$$\Delta Q^{\text{gravity},+} = \frac{1}{2M}\left(2k_{i,\text{in}} - 2Kk_i \sum_j k_j f(d_{ij})\right). \tag{10}$$

For simplicity, we refer to the $\sum_j k_j f(d_{ij})$ term for each node v_i as its $k_{j,\text{dists}}$ for the rest of this report and in our code, implicitly associating it with node v_i without letting the notation become convoluted. Calculating the $k_{j,\text{dists}}$ vector for a node v_i needs a little bit of explanation. Each community must have a contribution from $k_i f(d_{ii})$ because v_i will contribute that to any community it moves to. For all other nodes v_j we have a contribution to the null from $k_i k_j f(d_{ij})$ and $k_j k_i f(d_{ji})$ which are identical by symmetry. To separate k_i from $k_{j,\text{dists}}$ we just add $2 * k_j f(d_{ij})$ for each $j \neq i$.

2.3.1 Computational Considerations

The distances between each pair of nodes are introduced in the form of a pairwise distance matrix. Computing distance decay functions adds an extra layer of complexity to the algorithm and having to store the dense distance in memory places a bound on the size of the network this algorithm can be used for. In general, it will not be possible to use this algorithm for networks of more than 10^5 nodes [15]. One of the advantages of the Louvain algorithm over spectral methods was its ability to

deal with large networks. Estimating the eigenvectors in spectral methods becomes unfeasible for networks of over 10^4 nodes [9] so for spatial modularity measures the Louvain algorithm loses this advantage over spectral methods.

A second layer of complexity that is introduced occurs in phase 2 of the algorithm. Each time the meta-nodes are constructed the meta-distances must also now be calculated. This was done by [15] by calculating the distances between the centroids of each community. While this makes sense in the physical world, it is not ideal theoretically or computationally. The agglomeration step is nontrivial as the modularity of the original network with its given partition must be equal to the modularity of the singleton partition on the agglomerated network, we don't want to be changing anything about our network when we carry out this step. Additionally, calculating the centroids of each cluster requires information about node locations to be supplied instead of just the distance matrix. Not all datasets will have node location data and inferring node locations from pairwise distances alone is not possible. If node distances are not based on straightforward distance measures in Euclidean space we encounter another issue and calculating the distances between the centroids is also nontrivial and potentially expensive. We, therefore, want this algorithm to only require a distance matrix and no additional information. Instead, we propose calculating new distances in a simple manner that guarantees the modularity is preserved.

2.3.2 Agglomerating the Distance Matrix

At each agglomeration, we denote the new null model's entry for two meta-nodes C_1 and C_2 as $\tilde{P}_{C_1C_2}$. Its entries can be derived from the fact that the modularity for the union of these communities, $\frac{1}{2m}\sum_{i,j\in C_1\cup C_2}(A_{ij}-P_{ij})$ is

$$\frac{1}{2m}\sum_{i,\,j\in C_1}(A_{ij}-P_{ij})+\frac{2}{2m}\sum_{i\in C_1,\,j\cup C_2}(A_{ij}-P_{ij})+\frac{1}{2m}\sum_{i,\,j\in C_2}(A_{ij}-P_{ij})\,.$$

Thus since C_1 and C_2 represent communities in the partition that we are basing the agglomeration on, we know that for $C_1 \neq C_2$ then $C_1 \cup C_2 = \emptyset$. Then with a bit of rearranging we can see that

$$A_{C_1C_1} = \sum_{i,\,j\in C_1} A_{ij}$$

and

$$P_{C_1C_1} = \sum_{i,\,j\in C_1} P_{ij}.$$

The entries $P_{C_1C_2|C_1\neq C_2}$ correspond to the negative entries in the modularity sum if we were to combine groups C_1 and C_2 so the entries

$$\tilde{P}_{C_1C_2|C_1\neq C_2} = \sum_{i\in C_1,\, j\in C_2} P_{ij}.$$

This can be used to define how we will construct the meta-distances. If

$$\tilde{P}_{C_1C_2} = \sum_{i\in C_1,\, j\in C_2} P_{ij} = \sum_{i\in C_1,\, j\in C_2} K k_i k_j f(d_{ij})$$

then

$$K\tilde{k}_{C_1}\tilde{k}_{C_2} f\left(\tilde{d}_{C_1C_2}\right) = \tilde{P}_{C_1C_2}.$$

If we calculate $\tilde{P}$ entries directly from $\sum_{i\in C_1,\, j\in C_2} P_{ij}$ then distances become

$$\tilde{d}_{C_1C_2} = f^{-1}\left(\frac{\tilde{P}_{C_1C_2}}{K\tilde{k}_{C_1}\tilde{k}_{C_2}}\right).$$

In fact, its clear from this rather circular argument that we don't even need to recalculate the distances. We can proceed in a very general manner that won't distort the data by simply summing the original null model's entries as described to obtain the new null model with entries

$$\tilde{P}_{C_1C_1} = \sum_{i\in C_1,\, j\in C_2} P_{ij}. \tag{11}$$

2.4 Validation: Benchmarking

In order to validate these methods, some authors [14, 15] observed the average modularity score. As noted in [5], this is not a very accurate measure of performance. It is best to construct some synthetic networks with known partitions that influence link formation and to use these for validation [3, 5]. This form of validation also allows us to explore the parameter space in great detail and understand what networks this algorithm performs well for.

2.5 Benchmarking: Synthetic Spatial Networks

In order to easily construct a variety of spatial , we follow a similar routine for constructing synthetic graphs to those described in [12, 3, 5]. The most important parameters in determining the structure of these networks are λ and ρ, which control

the assortativity and density of the network. First, N nodes are randomly given some spatial location and assigned a binary attribute value of either 0 or 1. For any two nodes v_i and v_j, their community assignment is denoted by $g_i \in \{0, 1\}$. The expected number of flows between them is calculated according to

$$P_{ij} = \frac{1}{Z} \frac{\lambda_{g_i g_j}}{d^\ell} \tag{12}$$

where Z is a normalisation constant such that $\sum_{i \leq j} P_{ij} = 1$ and [4]. The Λ matrix takes the form

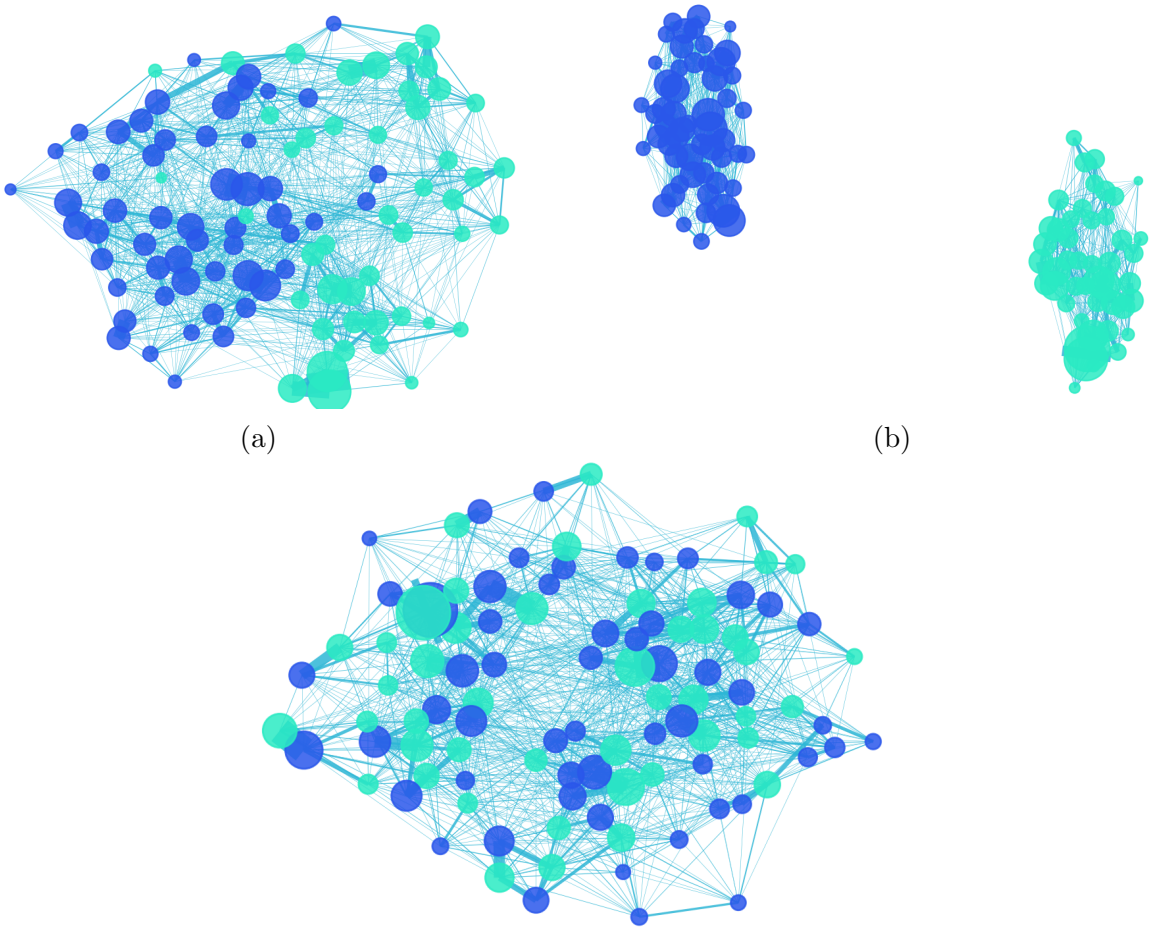
$$\Lambda = \begin{pmatrix} \lambda_{00} & \lambda_{01} \\ \lambda_{10} & \lambda 11 \end{pmatrix} = \begin{pmatrix} 1 & \lambda \\ \lambda & 1 \end{pmatrix}. \tag{13}$$

In this way, varying λ provides a simple way to control graph assortativity. For $\lambda < 1$ the graph is assortative, with disconnected communities for $\lambda = 0$. For all $\lambda > 1$ the graph has a primarily disassortative community structure. The parameter ρ controls the total number of flows in the network, which will be given by $L = \frac{1}{2}N(N-1)$ in our undirected case. Networks resulting from three different values of λ are shown in Figure 3., Figure 3b. shows a network where there are no connections between communities corresponding to $\lambda = 0$, Figure 3b. shows a predominantly assortative network corresponding to $\lambda = 0.1$ and Figure 3c. shows a network with a highly disassortative structure corresponding to $\lambda = 20$.

3 Python Implementation

For this project, we implement a spatially-corrected modularity optimisation algorithm by leveraging the work done in the `python-louvain` package by Thomas Aynaud[1]. This package is designed to work with the NetworkX package for network analysis in Python and implements the Louvain algorithm described in Section 2.2. It can be installed using pip with the commands `python3 -m pip install python-louvain` and it is generally advised to import its `community` package under an alias using `import community as community_louvain`. The package has been modified for spatial modularity in similar work by [15] but the code for this was not publically available, the validation method wasn't very comprehensive and the method of calculating meta-distances made generalisation to new data difficult. Additionally, this work used the distance decay function proposed by Expert et al. [5]

[1] https://github.com/taynaud/python-louvain



(a) (b)

(c)

Figure 3: Three networks of 100 nodes placed in a 10×10 square with $\ell = 2$. The nodes are not plotted in their spatial locations but according to NetworkX's `spring_layout` to exhibit the effects of assortativity. Community is denoted by colour and node degree is represented by size. Fig. 3a. shows a highly assortative graph produced when $\lambda = 0.1$. Nodes are predominantly connected to nodes within the same community. Fig. 3c. shows a primarily disassortative graph produced for $\lambda = 20$. There are more strong connections between nodes in different communities compared to the network in Fig. 3a. Finally, Fig. 3b. shows network of disconnected communities produced when $\lambda = 0$.

and for the reasons discussed in Section 2.3, this function is not implemented. Instead, combinations of decay functions and the parameter ℓ as in Table 1 will be used and compared.

arg	command	decay function $f(d)$
`decay`	`'invpow'`	$d^{-\ell}$
`decay`	`'invexp'`	$\mathrm{e}^{-d\ell}$

arg	command	ℓ-value
`ell`	`float`	float value
`ell`	`'mean'`	$(\text{mean distance})^{-1}$
`ell`	`'max'`	$(\text{max distance})^{-1}$

Table 1: Table of possibilities for the distance decay function

The package's author advised that any modifications should be kept close to the original implementation. That would mean as little change to the code as possible and that the usage interface will not change, hence it should default to the classical Newman-Girvan modularity optimisation unless otherwise specificied. In general, we have attempted to do this. A number of auxilliary modifications have been made to the code, including the renaming of a few variables, functions and parameters to facilitate readability for the user who is not already familiar with the `python-louvain` package. Any renamed variables were chosen to align with the mathematical notation in this report. Additionally, a new class inheriting from the NetworkX `Graph` object was defined to streamline usage for our specific purposes. These modifications are optional, however, and if this code were to be contributed to the `python-louvain` package, they could be easily removed. Our modifications of `python-louvain` are stored in a package called `spatial-louvain` which contains two subpackages `spatial_graphs` which defines the new spatial graph class and `community` which contains modifications to the original `community` package in `python-louvain`.

Even though the majority of the network science theory covered here is presented in matrix form, the implementation used in `python-louvain` uses very few matrix operations. The majority of the work is done by creating dictionaries where each key represents a node or community and values represent node or community attributes or else summands. For example the the i^{th} entry of the `nulls` dictionary usually contains the null model's total contribution to the modularity sum $\sum_{i,j\in C_i} P_{ij}$ for community C_i. This saves time and memory by reducing unnecessary calculations.

3.1 A New NetworkX Class

To streamline the usage of this package we began by defining a class to contain instances of spatial graphs called `SpatialGraph` which inherits from NetworkX's `Graph`

class. We inherit its standard constructor method from the parent class but place empty holders for attributes to store its flow matrix `fmat`, distance matrix `dists`, community partition `part` and node locations `locs`. We also define two new constructor class methods. The first, `from_numpy_array(self)`, can be passed a `numpy` array for the flow matrix as well as optional keyword arguments that allow users to assign any arbitrary attribute. This feature is useful for development but would probably be removed if this package were to be used publically.

The second constructor method allows the user to instantiate a random `SpatialGraph` object with a binary community structure by generating a random synthetic network from two parameter arguments `lamb` and `rho`, which correspond to assortativity λ, and flow density ρ, as described in Section 2.5. The distance decay function to be used is controlled by the optional parameters `decay` and `ell` which default to `'invpow'` and `ell=1.0`. Additional parameters included in this class method are `N`: the number of nodes, `len`: controls the size of the `len`×`len` square the nodes will be placed in, and `seed` which sets the random seed for generating nodes, allowing for reproducible results. The definition of this class is not strictly necessary but allows us to streamline our workflow by making benchmark construction straightforward and making frequently used graph attributes such as the adjacency matrix and distance matrix more quickly accessible. Functionality for sparse matrices and directed graphs has not been included but is a possibility for further extending the class.

3.2 Modifying the `python-louvain` Package

The original `python-louvain` package is renamed to `spatial-louvain` for this project. This is to enable users to have both packages installed in their virtual environments to allow for easy comparison. The package was modified in such a way as to leave the original workflow unchanged. Previous users should not have to change anything about how they call functions in order to use the package as they previously did. The original community detection module was not changed internally but it was renamed from `community_louvain.py` to `newman_girvan.py`. All functions in this module are loaded using the `__init__.py` file so that calling `community_louvain.best_partition()` automatically uses the classical Newman-Girvan algorithm. The `gravity` module was not loaded in this way so if users wish to use the gravity model-based modularity func-

tions they now can call
`community_louvain.gravity.best_partition()`.

3.2.1 The `status.py` Module

Both the modules `gravity.py` and `newman-girvan.py` use the `Status` class in the `status.py` module to store variables for passing between functions while running the `best_partition()` optimisation procedure. The original `Status` class contained two methods: a constructor which initialised the class and its empty attributes and a method called `init()` which accepts the graph, a weight key and an optional partition as parameters. It then calculates the summands for the internal weight and null model contributions to the modularity with respect to a partition and stores them as class attributes. If no partition is supplied it uses the singleton partition for this. To modify `Status` for our implementation we added the single argument `method` with default=`'newman_girvan'` to its constuctor which and this string becomes a class attribute. When the `Status.init()` method is called the `Status.method` attribute will determine whether to call the original method `Status.init_newman_girvan()` or our newly-defined one `Status.init_gravity()`. We rename the `init()` function to `init_newman_girvan()`. Then when the `init()` function is called it will by default implement the `init_newman_girvan()` method. If `method='gravity'`, however, a new method, `init_gravity()` will be implemented which calculates a different set of variables and attributes for use with the gravity modularity optimisation instead. In this way, we preserve the original functionality of the package and no change is made to how users would implement classic modularity clustering. The new `Status.init_gravity()` plays the same role as the original function but now calculates the null summands according to the gravity null model. It also accepts some additional optional parameters that correspond to the flexibility of the new `gravity.best_partition()` method which allows users to specify which specific decay function they wish to use and whether to normalise the null model to preserve the original flow.

3.2.2 The `newman_girvan.py` and `gravity.py` Modules

The majority of the algorithm is contained in the `gravity.py` module which is a modification of the `newman_girvan.py` module. The main function of this module is

the `best_partition()` function. On a high-level, this function initialises the Louvain algorithm and a dendogram is created. Each layer of the dendogram corresponds to a pass of the Louvain algorithm and successive layers contain coarser partitions. The last layer of the dendogram contains the largest communities [1] and this is the partition that `best_partition()` selects and returns. In general, efforts were made to preserve the programming style of the original package. In a few cases, however, variable names that were a little arbitrary were renamed to be more intuitive. Variable names that were renamed were named to correspond to their naming in the mathematical background theory given in this report. The degree of the node under consideration, for example, is labeled as k_i in this report, so would be referred to as `k_i` in the code. This renaming has been done to facilitate ease of understanding for readers unfamiliar with the code but if this code were to be made publically available the variables names would be renamed to prioritise consistency with pre-existing variable names in the package. A few private methods have also had their names changed so that they are accessible to facilitate testing.

3.2.3 Finding the Optimal Partition

The most important functions in the `gravity.py` module for generating one layer of the dendogram (one pass) are `modularity()`, `induced_graph()`, `_one_level()`, `get_ki_in()`, `get_kj_dists()`, `_modularity`, `_remove` and `_insert`. Functions with `_` in front of their names were originally private functions. The function `modularity()` is a standalone function which calculates the modularity of a graph and partition according to a specified decay function. The function `_one_level()` uses `get_ki_in()`, `get_kj_dists()`, `_modularity`, `_remove` and `_insert` to carry out phase 1 of the Louvain algorithm: determining the best partition and `induced_graph()` returns the induced graph for phase 2 of the Louvain algorithm according to a supplied partition. The `generate_dendogram()` method cycles between these to construct a dendogram of partitions for each phase.

The `get_ki_in()` and `get_kj_dists()` calculate the $k_{i,\text{in}}$ and $k_{j,\text{dists}}$ values to be used in calculating the cost of removing and inserting a node from each community as in Equations 9-10. The functions `_remove` and `_insert` then modify the `Status` object by removing and inserting nodes depending on which community assignment is calculated to best improve the modularity.

3.2.4 Additional Notes

In the case of zero distances in the distance matrix, which is also the case of self-loops the distance decay function will obviously blow up. As a temporary fix for this, we have placed code to replace all zero entries in the distance matrix at the beginning of the `best_partition()` function. This is a somewhat quick fix and better ways of doing this should definitely be explored. In the case of calculating meta-distances we do not allow zero distance values but we do allow meta-distances of infinity for the reason that these will go to zero once the distance decay function is reapplied.

It is also possible that meta-nodes will have no out-links after agglomeration, in this case we assign these nodes to their own community for all future phases of the algorithm.

3.3 Results

Unfortunately, the results when performing spatially-corrected modularity optimisation on the spatial benchmarking networks were quite underwhelming. For most parameter combinations and distance decay functions, the algorithm failed to produce average NMIs of more than a few decimal points higher than those calculated from the Newman-Girvan algorithm. There is a high chance that this is due to some persisting bugs in the code. Since this exact combination of features has not been attempted before it is difficult to diagnose the issue. For small, test networks, when ℓ was set to zero we observed almost identical behaviour to the Newman-Girvan null model. More discrepancies arose during parameter searches over multiple larger networks. This implies that there are still remaining bugs in the code which need to be uncovered before we can confidently make any statements about the suitability of this particular combination of distance decay functions and the Louvain algorithm. Some heatmaps of the averaged normalised mutual information score over ρ and λ in $[0.01, 100]$ and $[0, 2]$ are shown in Figures 4 and 5. The score for each parameter is the average of five runs which create new random synthetic graphs and perform both classical and spatial clustering on that graph using a random seed each time. In Figure 4 we expect to see almost identical behaviour but we find that changing to $\ell = 2$ in Figure 5 does not make the results drastically more similar and for this particular run actually decreased the NMI. Additionally, the algorithm occasionally

failed to converge during a pass. The reasons for this were not easily discernible but would be worth investigating. For these reasons we will not spend too much time on the results in this report.

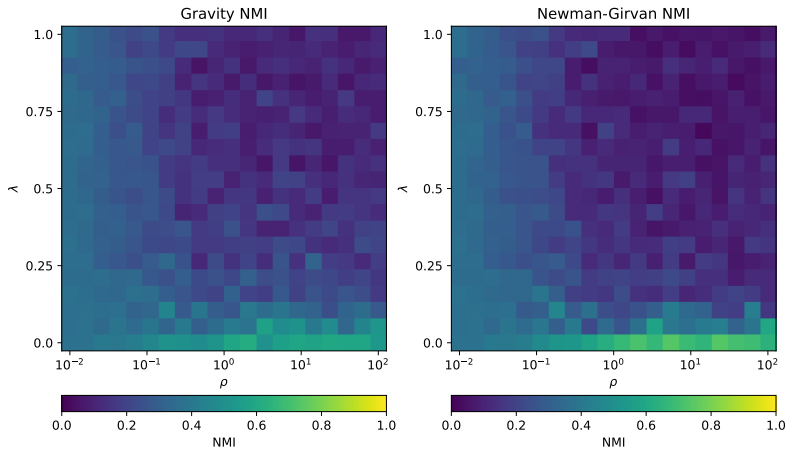


Figure 4: Average NMI score calculated for partitions predicted by the gravity and Newman-Girvan null models with $\ell = 0$ and resolution-0. The models behave extremely similarly as is expected because the gravity model should reduce to the Newman-Girvan null model when $\ell = 0$. Synthetic networks have `N=20` nodes, and are placed in a square of length `len=20`. They were built using `ell=2.` in the distance decay function $e^{-d\ell}$. Average NMI for the gravity model was 0.2292 with std. 0.1192 vs. 0.2068 with std. 0.141 for the Newman-Girvan model.

4 Conclusion

In this report, we have described our implementation of a spatially-corrected version of the popular Louvain modularity optimisation algorithm. The aim of this project was to create a software package that could seamlessly integrate with the existing `python-louvain` package while also retaining a degree of flexibility by allowing the user to specify their own distance decay functions for the modularity's null model. In general, we believe we have achieved this with our package. The remaining bugs should be fixable but would require a few more hours of investigation. Once those are fixed the `spatial-louvain` package should provide a useful tool that will allow users to explore different variations of spatially-corrected modularity clustering on their network.

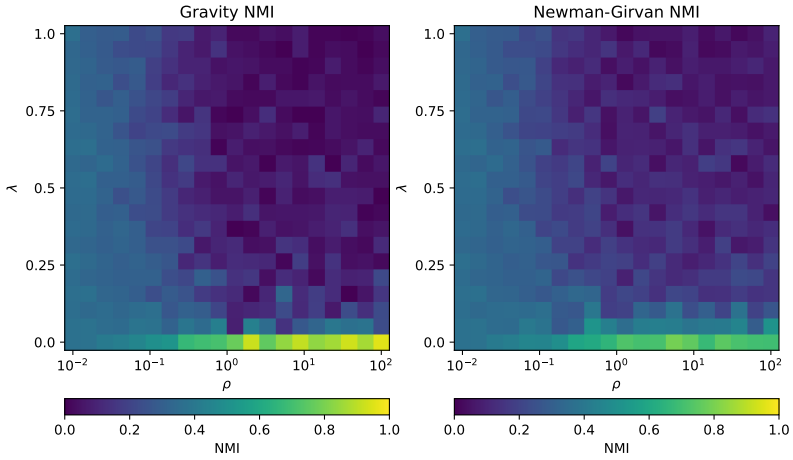


Figure 5: Average NMI score calculated for partitions predicted by the gravity and Newman-Girvan null models with $\ell = 2$ and resolution-0. Again, networks have `N=20` nodes, and are placed in a square of length `len=20`. They were built using `ell=2.` in the distance decay function $e^{-d\ell}$. Average NMI for the gravity model was 0.1886 with std. 0.1708 vs. 0.2118 with std. 0.1435 for the Newman-Girvan model.

The Louvain spatial optimisation algorithm itself, however, takes on average twice as long to run as the original Newman-Girvan one. As mentioned previously, the algorithm is also limited by the necessity of storing the large dense pairwise distance matrix, which limits its implementation to networks of under 10^5 nodes [15]. Unless the results were seen to dramatically improve, this additional computational expense does not seem justified and it's likely that other optimisation methods such as spectral methods would be more suitable. Additionally, all modularity-based clustering only identifies assortative structures. For this reason, it might be preferable to investigate clustering algorithms more suitable to different types of structures such as stochastic block models.

Acknowledgements

I would like to thank Renaud Lambiotte, Rodrigo Leal Cervantes and Thomas Aynaud for their helpful advice and interesting discussions which facilitated the completion of this project as well as Jim Hall and Jasper Verschuur for supplying and explaining the dataset which inspired this project's topic.

References

[1] Vincent D Blondel et al. "Fast unfolding of communities in large networks". In: *Journal of statistical mechanics: theory and experiment* 2008.10 (2008), P10008.

[2] Remy Cazabet, Pierre Borgnat, and Pablo Jensen. "Enhancing space-aware community detection using degree constrained spatial null model". In: *International Workshop on Complex Networks*. Springer. 2017, pp. 47–55.

[3] Federica Cerina et al. "Spatial correlations in attribute communities". In: *PLoS One* 7.5 (2012), e37507.

[4] Rodrigo Leal Cervantes. *Community Detection and Retail Networks, Confirmation Thesis for the degree of Doctor of Philosophy*. 2011.

[5] Paul Expert et al. "Uncovering space-independent communities in spatial networks". In: *Proceedings of the National Academy of Sciences* 108.19 (2011), pp. 7663–7668.

[6] Michelle Girvan and Mark EJ Newman. "Community structure in social and biological networks". In: *Proceedings of the national academy of sciences* 99.12 (2002), pp. 7821–7826.

[7] Sadamori Kojaku et al. "Multiscale core-periphery structure in a global liner shipping network". In: *Scientific reports* 9.1 (2019), pp. 1–15.

[8] Paul Krugman and Anthony J Venables. "Globalization and the Inequality of Nations". In: *The quarterly journal of economics* 110.4 (1995), pp. 857–880.

[9] Renaud Lambiotte. *C5.4 Networks Lecture Notes*. 2021.

[10] Renaud Lambiotte, J-C Delvenne, and Mauricio Barahona. "Laplacian dynamics and multiscale modular structure in networks". In: *arXiv preprint arXiv:0812.1770* (2008).

[11] Elizabeth A Leicht and Mark EJ Newman. "Community structure in directed networks". In: *Physical review letters* 100.11 (2008), p. 118703.

[12] Marta Sarzynska et al. "Null models for community detection in spatially embedded, temporal networks". In: *Journal of Complex Networks* 4.3 (2016), pp. 363–406.

[13] Michael T Schaub et al. "The many facets of community detection in complex networks". In: *Applied network science* 2.1 (2017), p. 4.

[14] Sebastijan Sekulić, JA Long, and U Demšar. "The effect of geographical distance on community detection in flow networks". In: *Proceedings of the AGILE 2018 conference, Lund, Sweden*. 2018, pp. 12–5.

[15] Paulo Shakarian et al. "Mining for geographically disperse communities in social networks by leveraging distance modularity". In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2013, pp. 1402–1409.

[16] Wei-Xing Zhou et al. "Predicting highway freight transportation networks using radiation models". In: *Physical Review E* 102.5 (2020), p. 052314.

A Appendix

A.1 Package Structure

The `spatial-louvain` package contains two subpackages. The *spatial_graphs* package which contains our definition of the new `SpatialGraph` class, and the *community* package, where the original Newman-Girvan modularity optimisation module is contained in `newman_girvan.py` along with the new `gravity.py` and modified `status.py` `modules`. In addition to this there is a folder named *scripts* which contains scripts for running parameter searches and generating visualisations of the synthetic networks. Finally, the *tests* folder contains tests for verifying our algorithm is behaving as it should. Not all the tests have been included in this writeup but a description of the tests in the script `test_final.py` is given below in Section A.2

A.2 Testing

A.2.1 Testing the modularity scores

To ensure our programme is giving accurate modularity results we test its performance on some simple graphs where we know what the results should be. We construct `pytest` tests using these known scores. We use of hand calculations and some of the simple graphs and modularity calculations covered in a Kaggle Tutorial[2] that covers some of the more subtle cases such as self-loops. First, we look at the simple graph with one self-loop given by the adjacency matrix

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

By convention, since self-loops are considered twice, we rewrite this for hand calculation as

$$A = \begin{pmatrix} 2 & 1 \\ 1 & 0 \end{pmatrix}.$$

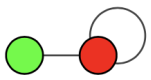


Figure 6: Diagram of first simple network to be used for testing. Image taken from Kaggle Tutorial[3].

[2]https://www.kaggle.com/alexandervc/hw1-part3-modularity-and-louvain-by-ac

Here, $2M = \sum_j k_{ij} = \sum_{ij} A_{ij} = 4$ and the degrees are $k = (3 \quad 1)$. For the trivial partition we know the modularity should be zero and for the singleton partition the modularity sum is

$$\begin{aligned} Q &= \frac{1}{2M}\left(A_{00} - \frac{k_1 k_1}{2M} + A_{11} - \frac{k_2 k_2}{2M}\right) \\ &= \frac{1}{4}\left(0 - \frac{1}{4} + 2 - \frac{9}{4}\right) \\ &= -\frac{1}{8}. \end{aligned}$$

We also compute for the same graph with two self-loops.

$$A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}.$$

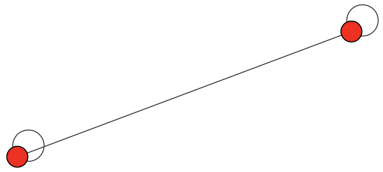


Figure 7: Diagram of second simple network to be used for testing. Image taken from Kaggle Tutorial[4].

For the trivial partition we know this should again be zero and for the singleton partition the modularity should be $\frac{1}{6}$.

The third sample graph we use the weighted adjacency matrix,

$$\begin{pmatrix} 1 & 4 & 1 \\ 4 & 0 & 2 \\ 1 & 2 & 0 \end{pmatrix}.$$

which will take the form

$$\begin{pmatrix} 2 & 4 & 1 \\ 4 & 0 & 2 \\ 1 & 2 & 0 \end{pmatrix}$$

for hand calculations.

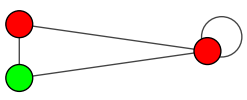


Figure 8: Diagram of third simple network to be used for testing. Plot generated using `iGraph`

The Newman-Girvan null model for the third graph is

$$P = \begin{pmatrix} 49 & 42 & 21 \\ 42 & 36 & 18 \\ 21 & 18 & 9 \end{pmatrix}.$$

The degrees are $k = (7 \quad 6 \quad 3)$ and $2M = 16$. The modularity for the singleton partition here is

$$Q = \frac{1}{16}\left(\left(2 - \frac{49}{16}\right) + \left(-\frac{36}{16}\right) + \left(-\frac{9}{16}\right)\right) = -0.2421875.$$

We construct tests to make sure both the modularity functions `modularity()` and `_modularity()` in `gravity.py` equally reproduce these values for modularity when `ell=0.`, at which point we recover the Newman-Girvan modularity function. We also test that both partitions generated are equal.

A.2.2 Testing the removal and insertion of nodes

We also want to test that removing and inserting nodes is working as expected. We do this by calculating the effect of moving node 0 from community 0 to community 1 for each of the three graphs. Here, the blue represents matrix entries contributing to the community 0 summands and red to the communities contributing to the community 1 summands as node 0 is first removed from community 0 and then placed in community 1.

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix} \to \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix} \to \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}$$
$$P = \begin{bmatrix} 9 & 3 \\ 3 & 1 \end{bmatrix} \to \begin{bmatrix} 9 & 3 \\ 3 & 1 \end{bmatrix} \to \begin{bmatrix} 9 & 3 \\ 3 & 1 \end{bmatrix}$$

From this we can see that with each removing/inserting operation the internals summands for $\{C_0, C_1\}$ should be $\{2, 0\} \to \{0, 0\} \to \{0, 4\}$ and the nulls summands for $\{C_0, C_1\}$ should be $\{9, 1\} \to \{0, 1\} \to \{0, 16\}$. For node 0, its $k_{i,\text{in}}$ vector is $(1 \quad 2)$ and its $k_{j,\text{dists}}$ vector is derived from $\frac{1}{3}P$ and is $(3 \quad 1)$. Note that in the implementation we only consider flows in one direction so the entries in k are halved and the $k_{j,\text{dists}}$ vector does not consider the effect of P_{ii} but rather the additional entries that are added to the summand for community when v_i is added.

Now we look at the second graph.

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \to \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \to \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$
$$P = \begin{bmatrix} 9 & 9 \\ 9 & 9 \end{bmatrix} \to \begin{bmatrix} 9 & 9 \\ 9 & 9 \end{bmatrix} \to \begin{bmatrix} 9 & 9 \\ 9 & 9 \end{bmatrix}$$

From this we can see the internals summands for $\{C_0, C_1\}$ should be $\{2, 0\} \to \{0, 2\} \to \{0, 6\}$ and the nulls summands for $\{C_0, C_1\}$ should be $\{9, 9\} \to \{0, 9\} \to \{0, 36\}$. For node 0, its $k_{i,\text{in}}$ vector is $(1 \quad 4)$ and its $k_{j,\text{dists}}$ vector is $(3 \quad 3)$.

Finally, we look at the third graph.

$$A = \begin{bmatrix} 2 & 4 & 1 \\ 4 & 0 & 2 \\ 1 & 2 & 0 \end{bmatrix} \to \begin{bmatrix} 2 & 4 & 1 \\ 4 & 0 & 2 \\ 1 & 2 & 0 \end{bmatrix} \to \begin{bmatrix} 2 & 4 & 1 \\ 4 & 0 & 2 \\ 1 & 2 & 0 \end{bmatrix} \text{ or } \begin{bmatrix} 2 & 4 & 1 \\ 4 & 0 & 2 \\ 1 & 2 & 0 \end{bmatrix} \to$$
$$P = \begin{bmatrix} 49 & 42 & 21 \\ 42 & 36 & 18 \\ 21 & 18 & 9 \end{bmatrix} \to \begin{bmatrix} 49 & 42 & 21 \\ 42 & 36 & 18 \\ 21 & 18 & 9 \end{bmatrix} \to \begin{bmatrix} 49 & 42 & 21 \\ 42 & 36 & 18 \\ 21 & 18 & 9 \end{bmatrix} \text{ or } \begin{bmatrix} 49 & 42 & 21 \\ 42 & 36 & 18 \\ 21 & 18 & 9 \end{bmatrix}$$

We see the internals summands for $\{C_0, C_1, C_3\}$ should be $\{2, 0, 0\} \to \{0, 0, 0\}$ when node 0 is removed from group 0. Inserting it into group 1 makes the internals summands $\{0, 10, 0\}$ while inserting it into group 2 makes the internals summands $\{0, 0, 4\}$. The nulls summands for $\{C_0, C_1, C_2\}$ should be $\{49, 36, 9\} \to \{0, 36, 9\}$ when node 0 is removed from group 0. Inserting it into group 1 makes the null summands $\{0, 169, 9\}$ and inserting it into group 2 makes the null summands $\{0, 36, 100\}$. Remembering that the node degrees are given by $k = (7 \quad 6 \quad 3)$ we finally derive $k_{i,\text{in}}$ and $k_{j,\text{dists}}$ for node v_i as $k_{i,\text{in}} = (2 \quad 10 \quad 4)$ and $k_{j,\text{dists}} = (7 \quad 6 \quad 3)$.