A Fast Agglomerative Algorithm for Edge Clustering

Alexandre Hollocou[1], Quentin Lutz[2], and Thomas Bonald[2]

[1]INRIA, Paris France
[2]Telecom Paristech, Paris France

2018

Abstract

Whereas most of existing graph clustering algorithms focus on finding partitions of nodes, it is often more interesting in practice to partition the edges of a graph. For this purpose, the classic modularity function, that is traditionally used to score node partitions, can be extended to edge partitions. This extension naturally emerges from an interpretation of the modularity in terms of random walks. In this paper, we propose a novel agglomerative algorithm to efficiently maximize this alternative *edge* modularity function. This algorithm is based on an important conservation result that guarantees that modularity is invariant under the aggregation operation that we consider. We show in our experiments that our method outperforms existing algorithms, mainly because, unlike other methods, our approach does not require the construction of a memory-expensive line graph.

1 Introduction

Graph clustering [8] is a central problem in graph analysis. Most of existing approaches focus on finding partitions of the graph nodes [5, 2, 14]. Yet, recovering node partitions is often too restrictive in practice as a node often belongs to several clusters in real-life graphs. For instance, in a social network, if we consider the graph where nodes correspond to users and edges to friendships between these users, we see that a node can potentially belong to multiple social circles (friends, colleagues, family etc.). For this reason, it can be more interesting to cluster the graph edges instead of the nodes [7]. In our social network example, this allows us to make a distinction between the different types of relationships (friendships, business relationships, family ties etc.) by putting the corresponding edges in different clusters.

In this paper, we study a generalization of the classic modularity measure [11], which is originally a score function defined on node partitions, to score partitions of edges. We refer to this alternative objective function as the *edge* modularity. Whereas the standard modularity function can be interpreted using a random walk on graph nodes [6, 10], the edge modularity arises by considering the sequence of edges crossed by this very same random walk. We propose a novel algorithm to maximize the edge modularity using an agglomerative strategy similar to the one used by the popular Louvain algorithm [2] that enables it to scale to large graphs. Our method is based on a key conservation result that states that the edge modularity is invariant under the particular aggregation scheme performed by our algorithm. This aggregation strategy relies on a distinction between *border* nodes that are linked to edges from different clusters and *internal* nodes that are only connected to edges from one cluster.

We show on both synthetic and real-life datasets that our algorithm significantly outperforms, both in terms of execution time and memory consumption, the existing approaches that rely on the construction of a weighted line graph [7] whose size is much larger than the original graph. Another important benefit of our method is the aggregated graph produced by the algorithm, as it is easy to visualize and interpret, and can be used to obtain better insights on large datasets.

The rest of the paper is organized as follows. We present the related work in section 2. In section 3, we define the edge modularity that we consider and study some of its properties. In section 4, we present our aggregation scheme, state the edge modularity conservation result, and define our edge modularity optimization algorithm. We present experimental results on synthetic and real-world data in section 5. Section 6 concludes the paper. Complete proofs of the results presented in this paper can be found in the appendices.

2 Related work

Numerous algorithms have been proposed to cluster graph nodes, with techniques ranging from spectral analysis [14] to matrix factorization [15]. Modularity maximization introduced by Newman and Girvan [11] is certainly the most widely used method to find node partitions. It cannot be solved exactly in reasonable time as it has been proven to be NP-hard [3], but many algorithms have been proposed to find good approximations [8]. The classic Louvain algorithm [2] is one of the fastest algorithms in practice, and consists in two steps that are repeated iteratively: a greedy maximization step and an agglomerative phase. The first step iterates over nodes and moves each node to the neighboring cluster that results in the largest modularity increase, and the second step builds a new graph where nodes are the clusters found during the first step. The key result on which the Louvain algorithm is built is that the modularity is invariant under this aggregation step. Thanks to this result, the algorithm is able to drastically reduce the size of the graph between two iteration steps, which enables it to handle very large graphs [12]. Another important benefit of this aggregation phase is that the aggregated graph is easy to interpret and can be used for synthetic visualizations, as the number of nodes in this graph is small. Indeed, the nodes of the aggregated graph correspond to clusters of nodes in the original graph and the edge weights in this new graph represent interactions between these node clusters.

Evans and Lambiotte have studied extensions of modularity to edge clusters in [7]. In this work, they define three scoring functions on edge partitions based on interpretations of the modularity function in terms of random walks [6, 10]. Evans and Lambiotte do not propose any specific algorithm to maximize these objective functions. They rely on the construction of three variants of the line graph (graphs whose nodes are the edges of the original graph) and the use of classic modularity optimization techniques on these graphs. In practice, the size of these line graphs is prohibitive, as the number of edges in these new graphs is in $O(\sum_u d_u^2)$ where d_u is the degree of node u, and their method cannot be applied to large real-life graphs (see Section 5). Note that other edge clustering algorithms have been proposed [13, 16], but their efficiency has either not been tested on real-world data, or only on particularly small graphs.

In this paper, we define a novel modularity function for edge partitions that is distinct from the scoring functions introduced by Evans and Lambiotte. Unlike their approach, our modularity function is directly related to the canonical random walk on graph nodes. Besides, we propose an efficient algorithm to maximize this function that does not require the construction of the line graph, and is thus able to scale to large datasets.

3 A modularity function for edge partitions

Let us consider an undirected and weighted graph $G = (V, E)$. We use $\boldsymbol{A}$ to denote the adjacency matrix of G (A_{uv} is the weight of edge $\{u, v\}$ if $\{u, v\} \in E$, and $A_{uv} = 0$ otherwise). The modularity of a given partition $\mathcal{C}$ of V is defined as

$$Q(\mathcal{C}) = \sum_{C \in \mathcal{C}} \sum_{u,v \in C} \left(\frac{A_{uv}}{w} - \frac{w_u w_v}{w^2} \right), \tag{1}$$

where w_u denotes the weighted degree of node u, $w_u = \sum_v A_{uv}$, and w denotes the sum of node weights, $w = \sum_u w_u$. The modularity can be interpreted as the difference between the probability to sample an edge between two nodes of the same cluster, and the probability for two independently sampled nodes to belong to the same cluster (where nodes and edges are sampled with probabilities proportional to their weights).

The modularity function can also be interpreted in terms of random walk on the nodes of G [6, 10]. Let us consider the classic random walk $(X_t)_{t\geq 0}$ on the nodes of G where the walk moves from a node u to one of its neighbors v with probability A_{uv}/w_u. $(X_t)_{t\geq 0}$ is a Markov chain. If G is connected and not bipartite, $(X_t)_{t\geq 0}$ has a unique stationary distribution π, defined as $\pi(u) = \frac{w_u}{w}$, and we have for all $u \in V$, $\lim_{t\to\infty} \mathbb{P}[X_t = u] = \pi(u)$ [4]. Then, the modularity of a partition $\mathcal{C}$ can be written as

$$Q(\mathcal{C}) = \sum_{C\in\mathcal{C}} \left(\mathbb{P}_\pi[X_0 \in C, X_1 \in C] - \mathbb{P}_\pi[X_0 \in C, X_\infty \in C]\right), \tag{2}$$

where $\mathbb{P}_\pi[X_0 \in C, X_1 \in C]$ is the probability for the walk to be in C at initial time (with X_0 initialized with the distribution π) and after one step, and $\mathbb{P}_\pi[X_0 \in C, X_\infty \in C]$ is the probability for the walk to be in C at initial time and in the stationary regime.

From the random walk $(X_t)_{t\geq 0}$ on the nodes of G, we can naturally define a Markov chain $(Y_t)_{t\geq 0}$ on the edges E of G, with, for all $t \geq 0$, $Y_t = \{X_t, X_{t+1}\} \in E$. For $e = \{u, v\} \in E$, we have:

$$\mathbb{P}[Y_t = e] = w_e \sum_{u'\in e} \frac{\mathbb{P}[X_t = u']}{w_{u'}},$$

where w_e denotes the weight of the edge e, i.e. $w_e = A_{uv}$.

Building on the random walk interpretation of the modularity, we can define the modularity for a partition of the edges E using $(Y_t)_{t\geq 0}$. Given a partition $\mathcal{C}$ of the edges of G, we define the *edge* modularity of $\mathcal{C}$ as

$$\tilde{Q}(\mathcal{C}) = \sum_{C\in\mathcal{C}} \left(\mathbb{P}_\pi[Y_0 \in C, Y_1 \in C] - \mathbb{P}_\pi[Y_0 \in C, Y_\infty \in C]\right), \tag{3}$$

where $\mathbb{P}_\pi[Y_0 \in C, Y_1 \in C]$ is the probability for the first two edges crossed by the walk to be in C (with X_0 initialized with the distribution π), and $\mathbb{P}_\pi[Y_0 \in C, Y_\infty \in C]$ is the probability for an edge crossed by the walk to be in C for the first jump and in the stationary regime.

The following proposition gives explicit expressions of the *edge* modularity.

Proposition 3.1. *The* edge *modularity of a partition $\mathcal{C}$ of E can be written as*

$$\tilde{Q}(\mathcal{C}) = \sum_{C\in\mathcal{C}} \sum_{e,f\in C} \left(\frac{w_e w_f}{w} \sum_{u\in e\cap f} \frac{1}{w_u} - \frac{w_e w_f}{(w/2)^2} \right) \tag{4}$$

and

$$\tilde{Q}(\mathcal{C}) = \sum_{C\in\mathcal{C}} \left[\sum_{u\in V} \frac{(w_u(C))^2}{w w_u} - \left(\frac{w(C)}{w(E)}\right)^2 \right] \tag{5}$$

where $w_u(C) = \sum_{e\in C} \mathbb{1}_{\{u\in e\}} w_e$ and $w(C) = \sum_{e\in C} w_e$.

Proof. We prove these results in Appendix B. □

The *edge* modularity $\tilde{Q}(\mathcal{C})$ of an edge partition $\mathcal{C}$ is actually equal to the standard modularity of $\mathcal{C}$ in a graph H whose nodes correspond to the edges of G. This result is stated in the following proposition.

Proposition 3.2. *Let us define the graph $H = (V^H, G^H)$ with $V^H = E$, $E^H = \{\{e, f\} \in E \times E : \exists u \in V, u \in e \cap f\}$, and the adjacency matrix $A^H_{ef} = \frac{w_e w_f}{2} \sum_{u\in e\cap f} \frac{1}{w_u}$.*

For all partitions $\mathcal{C}$ of E, we have $\tilde{Q}(\mathcal{C}) = Q^H(\mathcal{C})$. In other words, the edge modularity *of $\mathcal{C}$ in G is equal to the standard modularity of $\mathcal{C}$ in H.*

Proof. This result follows from (4) as proven in Appendix A. □

Note that Evans and Lambiotte [7] uses a matrix $\boldsymbol{E}$ that corresponds to $\boldsymbol{A}^H$ in order to define an adjacency matrix $\boldsymbol{E}_1 = \boldsymbol{E}\boldsymbol{E} - \boldsymbol{E}$ to which they apply the classic modularity maximization algorithms. Hence, they do not directly consider the graph H, but $\boldsymbol{A}^H$ appears in their work as an intermediary result.

Proposition 3.2 has important consequences. First, it follows from the work of Brandes et al. [3] that the problem of maximizing $\tilde{Q}(\mathcal{C})$ is NP-hard and cannot be solved exactly in reasonable time. Then, we see that the standard modularity maximization heuristics as the Louvain algorithm can be used on the graph H to find an approximation of the solution of $\max_{\mathcal{C}} \tilde{Q}(\mathcal{C})$. However, we will see that this approach does not scale to large graphs.

4 An agglomerative algorithm for maximizing edge modularity

We have seen above that the edge modularity can be expressed as the standard modularity in a weighted line graph H. Therefore, a natural approach for maximizing the edge modularity consists in building the graph H corresponding to G using Proposition 3.2 and applying the Louvain algorithm to this new graph. However there is an important limit to this naive approach. Indeed, building the graph H is prohibitive in practice for large graphs G as the number of edges in H is $\left(\sum_{u\in V}(d_u)^2 - m\right)/2$, where d_u denotes the unweighted degree of a node u in G and m denotes the number of edges in G. This number typically explodes as the sizes of the considered graphs increase, because node degrees follow scale-free power-law distributions in most of real-life graphs [1].

In this section, we define an algorithm that does not require the construction of the weighted line graph H. This algorithm is based on an aggregation scheme that preserves the edge modularity (which is the central result of this section) and that can be easily interpreted.

4.1 Agglomerative step

In order to define our aggregation scheme, we define two sets of nodes for each edge cluster: the internal nodes and the border nodes.

Definition 4.1. *Given a partition of the edges of G, $\mathcal{C} = \{C_1, \dots, C_K\}$, we define the internal nodes for cluster C_k as*

$$\mathcal{I}(k) = \{u \in V : \forall e \in E, u \in e \Rightarrow e \in C_k\}$$

and the border nodes for cluster C_k as

$$\mathcal{B}(k) = \{u \in V : \exists e \in C_k, \exists f \notin C_k, u \in e \cap f\}.$$

In other words, the internal nodes correspond to nodes that are bound to edges that all belong to the same cluster, whereas border nodes are bound to edges that belong to distinct clusters. These sets of nodes follow the following properties, for all k:

$$\text{(a) } \mathcal{B}(k) \cap \mathcal{I}(k) = \emptyset \quad \text{(b) } \forall u \in \mathcal{B}(k), \exists l \neq k, u \in \mathcal{B}(l) \quad \text{(c) } \forall l \neq k, I(k) \cap I(l) = \emptyset.$$

In the following definition, we consider an aggregation operation that consists in aggregating the internal nodes for each cluster C_k into a single node v_k with a self-loop.

Definition 4.2. *Given a graph G and an edge partition $\mathcal{C} = \{C_1, \dots, C_K\}$, we define the* aggregated graph relative to G and $\mathcal{C}$ *as $G' = (V', E')$ with the set of node*

$$V' = \left(\bigcup_k \mathcal{B}(k)\right) \cup \{v_k : \mathcal{I}(k) \neq \emptyset\},$$

where $v_k = \mathcal{I}(k)$. The weighted edges E' of G' are defined by the adjacency matrix $\boldsymbol{A'}$

$$\forall u, v \in V', \quad A'_{uv} = \begin{cases} A_{uv} & \text{if } u, v \in \mathcal{B}(k) \\ \sum_{u',v' \in \mathcal{I}(k)} A_{u'v'} & \text{if } u = v = v_k \\ \sum_{v' \in \mathcal{I}(k)} A_{uv'} & \text{if } u \in \mathcal{B}(k),\ v = v_k \\ \sum_{u' \in \mathcal{I}(k)} A_{u'v} & \text{if } u = v_k,\ v \in \mathcal{B}(k) \\ 0 & \text{otherwise.} \end{cases}$$

We also define the corresponding aggregated partition $\mathcal{C}' = \{C'_1, \dots, C'_K\}$ as

$$\begin{aligned} C'_k = &\{\{u,v\} \in C_k : u, v \in \mathcal{B}(k)\} \cup \{\{u, v_k\} : u \in \mathcal{B}(k), \exists v \in \mathcal{I}(k), \{u,v\} \in C_k\} \\ &\cup \{\{v_k, v_k\} : A'_{v_k v_k} > 0\}. \end{aligned}$$

In other words, this aggregation operation can be decomposed into two steps. (i) For each cluster C_k, all the internal nodes are aggregated into a single node v_k bearing a self-loop of weight $\frac{1}{2}\sum_{u,v \in \mathcal{I}(k)} A_{uv}$. (ii) For each border node $u \in \mathcal{B}(k)$, an edge between u and v_k is created if $\exists v \in \mathcal{I}(k)$ s.t. $\{u, v\} \in E$. The weight of this edge is then $\sum_{v \in \mathcal{I}(k)} A_{uv}$. Note that the result of this aggregation operation is easy to interpret. Indeed, the nodes in the new graph G' correspond to single nodes or groups of nodes in the original graph, and the edge weights in G' correspond to the interactions between these groups of nodes (or to internal interactions for self-loops). We will illustrate this point on real-life data in Section 5.

The following theorem states that the edge modularity is invariant under this aggregation operation.

Theorem 4.3. *The edge modularity $\tilde{Q}(\mathcal{C})$ of the edge partition $\mathcal{C}$ in G is equal to the edge modularity $\tilde{Q}(\mathcal{C}')$ of the edge partition $\mathcal{C}'$ in the aggregated graph G'.*

Proof. We use the formulation (4) to prove this result. On the one hand, it is easy to verify that we have $w'(C'_k) = w(C_k)$ and $w'(E') = w(E)$ using Definition 4.2, where w' denotes the weights in the new graph G'. On the other hand, similar calculations lead to $w'_u(C'_k) = w_u(C_k)$ and $w'_u = w_u$ for all nodes $u \in \mathcal{B}(k)$ for all k. Therefore, we only need to prove that

$$\sum_k \sum_{u \in \mathcal{I}(k)} \frac{w_u(C_k)^2}{w_u} = \sum_k \frac{w'_{v_k}(C'_k)^2}{w'_{v_k}}. \tag{6}$$

We remark that, for all k, for all $u \in \mathcal{I}(k)$, $w_u(C_k) = w_u$ and $w'_{v_k}(C'_k) = w'_{v_k}$. Thus (6) is equivalent to $\sum_{u \in \mathcal{I}(k)} w_u = w'_{v_k}$, which follows from Definition 4.2. □

Thanks to this key conservation result, we can use the same strategy as the Louvain algorithm to optimize the edge modularity, i.e. alternate greedy maximization of $\tilde{Q}$ and the agglomerative operation described above.

4.2 Greedy maximization step

The simplest greedy method to maximize the edge modularity consists in cycling over the graph edges and, for each edge, move it to the neighboring cluster that leads to the highest increase in edge modularity.

However, this greedy maximization strategy is only applicable to the first step of the algorithm when the graph has not been aggregated. Indeed, the aggregation operation generally yields to aggregated clusters C'_k that contain more than one edge. The edges in these clusters must then be moved all together in order for the modularity conservation result from Theorem 4.3 to apply. To handle this issue, we consider a method that consists in cycling through the cluster C'_k returned by the aggregation operation, and moves the edges of each of these clusters in groups. We move all the edges of such a group to the neighboring cluster that results to the largest increase in edge modularity.

The variation of edge modularity when a group of edges $M \subset E$ is moved from one cluster C_k to another cluster C_l is given by

$$\Delta\tilde{Q}(M : C_k \to C_l) = \sum_{u : w_u(M) > 0} \frac{w_u(M)}{w(E) w_u}(w_u(M) + w_u(C_l) - w_u(C_k)) - \frac{2w(M)(w(M) + w(C_l) - w(C_k)}{w(E)^2}.$$

4.3 Algorithm

In Algorithm 1, we give the pseudo-code of the algorithm that iteratively repeats the greedy maximization step and the aggregation operation described above. At initial time, the algorithm puts each edge in its own cluster. Like the Louvain algorithm, the algorithm takes a unique parameter $\epsilon > 0$. The greedy maximization step and the aggregation step are repeated until the modularity increase is lower than this sensibility threshold ϵ.

Algorithm 1 Agglomerative edge modularity optimization algorithm

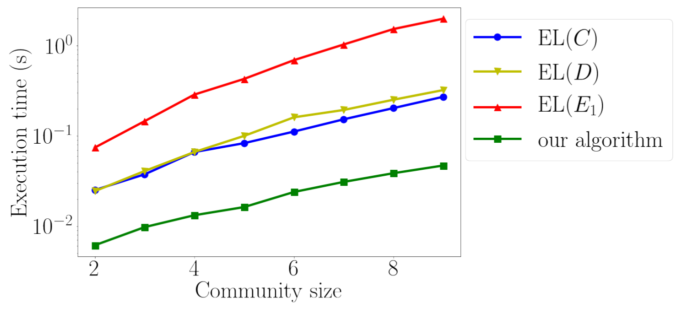
```
Require: Graph $G$ and sensibility threshold $\epsilon \geq 0$.
  Initialization: $\mathcal{C} \leftarrow \{\{e\}, e \in E\}$ (edge partition) and $\mathcal{M} \leftarrow \mathcal{C}$ (groups of edges to consider)
  $\tilde{Q}_{\text{old}} \leftarrow -\infty$, $\tilde{Q}_{\text{new}} \leftarrow \tilde{Q}(\mathcal{C})$
  while $\tilde{Q}_{\text{new}} - \tilde{Q}_{\text{old}} > \epsilon$ do
      Greedy maximization step
      increase ← true
      while increase do
          increase ← false
          for $M \in \mathcal{M}$ do
              $C_k \leftarrow$ current cluster of $M$ in $\mathcal{C}$
              Compute $\Delta\tilde{Q}(M : C_k \to C_l)$ for neighboring clusters $C_l$ in $\mathcal{C}$
              $l^* \leftarrow \arg\max_l \Delta\tilde{Q}(M : C_k \to C_l)$
              if $\Delta\tilde{Q}(M : C_k \to C_{l^*}) > \epsilon$ then
                  increase ← true
                  Modifiy $\mathcal{C}$ by moving $M$ from cluster $C_k$ to $C_{l^*}$
              end if
          end for
      end while
      Aggregation step
      $G, \mathcal{C} \leftarrow$ aggregated graph and corresponding partition given by Definition 4.2
      $\mathcal{M} \leftarrow \mathcal{C}$, $\tilde{Q}_{\text{old}} \leftarrow \tilde{Q}_{\text{new}}$, $\tilde{Q}_{\text{new}} \leftarrow \tilde{Q}(\mathcal{C})$
  end while
```

In order to compute the increase $\Delta\tilde{Q}(M : C_k \to C_l)$ and to perform the aggregation operation in an efficient manner, the algorithm can store the weights $w(C_k)$, $w_u(C_k)$, and the sets $\mathcal{B}(C_k)$ and $\mathcal{I}(C_k)$ for each cluster C_k.

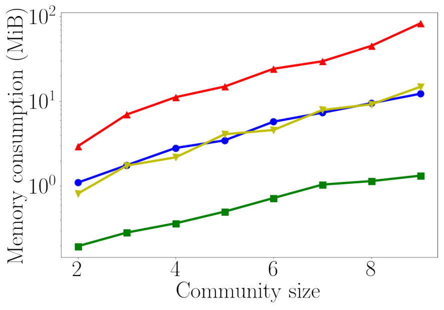
5 Experimental results

5.1 Synthetic data

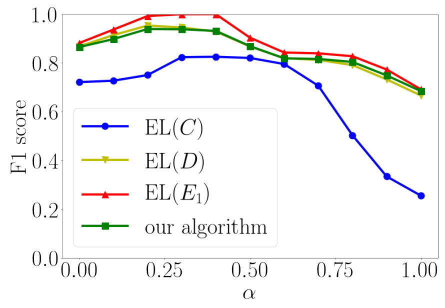
We use a stochastic block model (SBM) [9] with overlapping blocks to evaluate the performance of our algorithm on synthetic data. Given a set $\{B_1, \dots, B_I\}$ of node blocks $B_i \subset V$, we consider the random graph model where the edge $\{u, v\}$ appears in the graph with probability p_{in} if u and v belongs to the same block



(a) Execution time in seconds



(b) Memory consumption in MiB



(c) F1 score

Figure 1: **Overlapping SBM** - Execution times, memory consumption and F1 scores

B_i, and with probability p_{out} otherwise. The probabilities p_{in} and p_{out} are parameters of our model. The blocks that we consider have the same size $c > 0$ ($\forall i, |B_i| = c$) and overlap as follows: for all i, $|B_i \cap B_{i+1}| = o$ and $B_i \cap B_j = \emptyset$ if $j > i + 1$.

We compare our algorithm to the algorithms that consist in building the line graphs defined by Evans and Lambiotte [7], C, D and E_1, and applying the Louvain algorithm to these graphs. We refer to the algorithm that corresponds to the graph C as EL(C) (and respectively we call EL(D) and EL(E_1) the algorithms that correspond to the graphs D and E_1). All these algorithms have been implemented in Python and will be made available online.

First, we compare the execution time and the memory consumption of these algorithms on multiple instances of our random graph model for different numbers of blocks I. The other parameters of the model are set to $p_{\text{in}} = 0.95$, $p_{\text{out}} = 0.05$, $c = 10$ and $o = 2$. We display the results in Figure 1. We see that our algorithm runs on average more than 5 times faster than EL(C), EL(D) and EL(E_1). Besides, our algorithm consumes on average 10 times less memory than EL(C) and EL(D) and almost 100 times less memory than EL(E_1). This can be easily explained by the fact that the size of the line graphs considered by the approach of Evans and Lambiotte is much larger than the size of the original graph G.

In order to compare the quality of the results of the different algorithms, we define the notion of estimated blocks $\hat{B}_k$ for the different algorithms. If $C_1, \ldots, C_K$ are the edge clusters returned by one algorithm, we define the corresponding estimated blocks for this algorithm as

$$\forall k, \quad \hat{B}_k = \left\{ u \in V : \frac{w_u(C_k)}{w_u} \geq \alpha \right\}$$

where $\alpha \in [0, 1]$ is a given parameter. Then, we use the classic F1 score, defined as the harmonic mean of the precision and the recall, to compare the original blocks (B_i) with the estimated blocks ($\hat{B}_k$) for the different algorithms. The F1 score is equal to 1 if and only if the blocks ($\hat{B}_k$) exactly correspond to the planted blocks (B_i).

Cluster 1	Frankfurt am Main, Charles de Gaulle, Amsterdam Schiphol, Munich, Barcelona
Cluster 2	Hartsfield Jackson Atlanta, Chicago O'Hare, Dallas Fort Worth, Houston, Denver
Cluster 3	Atatürk, Dubai, Suvarnabhumi, Kuala Lumpur, King Abdulaziz
Cluster 4	Domodedovo, Sheremetyevo, Pulkovo, Ben Gurion, Vnukovo
Cluster 5	Mohammed V, OR Tambo, Jomo Kenyatta, Murtala Muhammed, Quatro de Fevereiro

Figure 2: **OpenFlights** - Edge clusters around London Heathrow

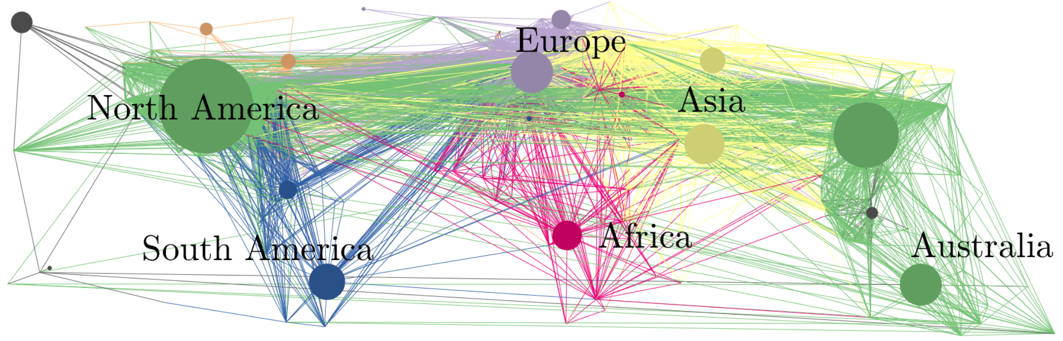


Figure 3: **OpenFlights** - Aggregated graph

In Figure 1, we plot the average F1 scores for values of α ranging from 0 to 1. We see that the F1 score for our algorithm is on average 50% higher than the score observed for EL(C) and is very comparable to the score obtained by EL(D) (within 1% on average). This score is slightly lower (3% on average) than the score achieved by EL(E_1). However, we see that the benefit of EL(E_1) is marginal knowing that the memory consumption of this algorithm is almost 100 times higher than the one measured for our algorithm.

5.2 Real-life datasets

5.2.1 OpenFlights

We first use the OpenFlights Airport Database[1] for our benchmarks on real-life data. We consider the graph whose nodes correspond to airports and whose edges correspond to airline routes between these airports. We run our algorithm and EL(C), EL(D) and EL(E_1) on this graph that contains 3,097 nodes and 18,193 edges. Our algorithm runs in 16.9 seconds and uses 15 MiB of memory, whereas EL(C) and EL(D) run respectively in 95 and $7.2 \cdot 10^3$ seconds and use 817 and 815 MiB of memory. The execution EL(E_1) raises a out-of-memory error on a computer with 16GB of RAM.

To illustrate the results of our algorithm on this dataset, we first focus on the node corresponding to the London Heathrow Airport and consider its adjacent edges. We are interested in the clusters in which these adjacent edges have been categorized by our algorithm. In Figure 2, we list the end-nodes of these edges (i.e. the reachable airports from London Heathrow) and categorize them by edge clusters. We only display the first 5 clusters in terms of number of edges and we only list the top 5 airports in term of degree in the graph for each edge cluster. We see that the different clusters correspond to flights to different regions of the world (Europe for cluster 1, North America for cluster 2, Middle East and South Asia for cluster 3 etc.).

In addition to the clusters returned by our algorithm, the aggregated graph itself is very useful to have a synthetic view of the world airline traffic. This graph only contains 426 nodes, i.e. almost 10 times less nodes than the original graph, and 6,940 edges. We display this graph in Figure 3. We use edges of different colors to indicate the different edge clusters. We use colored circles to represent aggregated nodes, with a size proportional to the number of nodes that have been aggregated and a color corresponding to the edge

[1]`https://openflights.org`

cluster to which the aggregated nodes are connected. We position the aggregated nodes at the centroid of the airports that have been merged. We see that these aggregated nodes correspond to important regions of the world (North America, Central America, South America, Central Europe, North Europe etc.). Besides, we observe that the airports that have not been aggregated correspond to interfaces between these different regions of the world.

5.2.2 Wikipedia subset

In order to compare the performance of the algorithms on a larger graph, we run them on a subgraph of the Wikipedia graph that contain 4,589 nodes and 106,534 edges. In this graph, nodes correspond to Wikipedia articles and edges to hyperlinks between those articles. Whereas our algorithm runs in 281 seconds and use 229 MiB of memory on this graph, the three algorithms EL(C), EL(D) and EL(E_1) raise out-of-memory errors on a computer with 16 GB of RAM.

6 Conclusion

In this paper, we proposed a novel agglomerative algorithm for the edge clustering problem that maximizes a variant of Newman's modularity that emerges by considering a random walk on the graph edges instead of the graph nodes. This algorithm does not require the construction of a weighted line graph unlike the existing approaches. It is based on a special aggregation operation that conserves the edge modularity as proven in Theorem 4.3. Thanks to this aggregation step, the size of the graph decreases at each iteration step of the algorithm which enables it to handle large graphs. We showed on both synthetic and real-world datasets that our algorithm significantly outperforms Evans and Lambiotte's methods in terms of memory consumption and time execution. Besides, we illustrated on the OpenFlights dataset that the aggregated graph produced by our algorithm can be used as a powerful tool for visualization.

In future work, we would like to extend our algorithm to the edge streaming framework in order to handle common situations where graph edges are given in an online fashion.

A Proof of Proposition 3.1

The first equality follows from

$$\begin{aligned}
\mathbb{P}_\pi[Y_0 \in C, Y_1 \in C] &= \sum_{e\in C} \mathbb{P}_\pi[Y_0 \in C, Y_1 = e] = \sum_{e\in C} w_e \sum_{u\in e} \frac{\mathbb{P}_\pi[X_1 = u, Y_0 \in C]}{w_u} \\
&= \sum_{e\in C} w_e \sum_{u\in e} \frac{1}{w_u} \sum_{v\in V} \underbrace{\mathbb{P}_\pi[X_0 = v]}_{=w_v/w} \frac{A_{vu}}{w_v} \mathbb{1}_{\{\{v,u\}\in C\}} \\
&= \frac{1}{w} \sum_{e\in C} w_e \sum_{u\in e} \frac{1}{w_u} \sum_{f\in C} \mathbb{1}_{\{u\in f\}} w_f = \sum_{e,f\in C} \frac{w_e w_f}{w} \sum_{u\in e\cap f} \frac{1}{w_u} \\
\mathbb{P}_\pi[Y_0 \in C, Y_\infty \in C] &= \sum_{e,f\in C} \frac{4 w_e w_f}{(w)^2}.
\end{aligned}$$

where we used the fact that, for all $e \in E$,

$$\lim_{t\to\infty} \mathbb{P}[Y_t = e] = w_e \sum_{u\in e} \frac{1}{w_u} \underbrace{\lim_{t\to\infty} \mathbb{P}[X_t = u]}_{w_u/w} = \frac{2w_e}{w}.$$

For the second equality, we remark that

$$\sum_{e,f\in C} \frac{w_e w_f}{w} \sum_{u\in e\cap f} \frac{1}{w_u} = \sum_{u\in V} \frac{1}{w w_u} \left(\sum_{e\in C} \mathbb{1}_{\{u\in e\}} w_e \right)^2$$

and

$$\sum_{e,f\in C} \frac{w_e w_f}{(w/2)^2} = \frac{\left(\sum_{e\in C} w_e\right)^2}{\left(\sum_{e\in E} w_e\right)^2}.$$

B Proof of Proposition 3.2

First, we have for all $e \in V^H$.

$$\begin{aligned} w_e^H &= \sum_{f\in V^H} A_{ef}^H = \sum_{f\in E} \frac{w_e w_f}{2} \sum_{u\in e\cap f} \frac{1}{w_u} \\ &= \frac{w_e}{2} \sum_{u\in e} \frac{1}{w_u} \sum_{f\in E} \mathbb{1}_{\{u\in f\}} w_f \\ &= \frac{w_e}{2} \sum_{u\in e} \frac{w_u}{w_u} = w_e. \end{aligned}$$

Then, observe that

$$w^H = \sum_{e\in V^H} w_e^H = \sum_{e\in E} w_e = w/2.$$

This gives us

$$\sum_{C\in\mathcal{C}} \sum_{e,f\in C} \frac{A_{ef}^H}{w^H} = \sum_{C\in\mathcal{C}} \sum_{e,f\in C} \frac{w_e w_f}{w} \sum_{u\in e\cap f} \frac{1}{w_u}$$

and

$$\sum_{C\in\mathcal{C}} \sum_{e,f\in C} \frac{w_e^H w_f^H}{(w^H)^2} = \sum_{C\in\mathcal{C}} \sum_{e,f\in C} \frac{w_e w_f}{(w/2)^2}$$

Using (4), we can conclude the proof.

References

[1] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[2] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[3] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner. On finding graph clusterings with maximum modularity. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 121–132. Springer, 2007.

[4] F. R. Chung. *Spectral graph theory.* Number 92. American Mathematical Soc., 1997.

[5] A. Clauset, M. E. Newman, and C. Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

[6] J.-C. Delvenne, S. N. Yaliraki, and M. Barahona. Stability of graph communities across time scales. *Proceedings of the national academy of sciences*, 107(29):12755–12760, 2010.

[7] T. Evans and R. Lambiotte. Line graphs, link partitions, and overlapping communities. *Physical Review E*, 80(1):016105, 2009.

[8] S. Fortunato and C. Castellano. Community structure in graphs. In *Computational Complexity*, pages 490–512. Springer, 2012.

[9] P. W. Holland, K. B. Laskey, and S. Leinhardt. Stochastic blockmodels: First steps. *Social networks*, 5(2):109–137, 1983.

[10] R. Lambiotte, J.-C. Delvenne, and M. Barahona. Laplacian dynamics and multiscale modular structure in networks. *arXiv preprint arXiv:0812.1770*, 2008.

[11] M. E. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[12] A. Prat-Pérez, D. Dominguez-Sal, and J.-L. Larriba-Pey. High quality, scalable and parallel community detection for large real graphs. In *Proceedings of the 23rd international conference on World wide web*, pages 225–236. ACM, 2014.

[13] C. Shi, Y. Cai, D. Fu, Y. Dong, and B. Wu. A link clustering based overlapping community detection algorithm. *Data & Knowledge Engineering*, 87:394–404, 2013.

[14] U. Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.

[15] J. Yang and J. Leskovec. Overlapping community detection at scale: a nonnegative matrix factorization approach. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 587–596. ACM, 2013.

[16] H. Zhou, X. Yuan, W. Cui, H. Qu, and B. Chen. Energy-based hierarchical edge clustering of graphs. In *Visualization Symposium, 2008. PacificVIS'08. IEEE Pacific*, pages 55–61. IEEE, 2008.