Knowledge and Information Systems
https://doi.org/10.1007/s10115-021-01623-y

REGULAR PAPER

Adapting *k*-means for graph clustering

Sami Sieranoja[1] · **Pasi Fränti**[1]

Received: 13 January 2021 / Revised: 1 November 2021 / Accepted: 7 November 2021

Abstract
We propose two new algorithms for clustering graphs and networks. The first, called *K-algorithm*, is derived directly from the *k*-means algorithm. It applies similar iterative local optimization but without the need to calculate the means. It inherits the properties of *k*-means clustering in terms of both good local optimization capability and the tendency to get stuck at a local optimum. The second algorithm, called the *M-algorithm*, gradually improves on the results of the *K*-algorithm to find new and potentially better local optima. It repeatedly merges and splits random clusters and tunes the results with the *K*-algorithm. Both algorithms are general in the sense that they can be used with different cost functions. We consider the conductance cost function and also introduce two new cost functions, called *inverse internal weight* and *mean internal weight*. According to our experiments, the *M*-algorithm outperforms eight other state-of-the-art methods. We also perform a case study by analyzing clustering results of a disease co-occurrence network, which demonstrate the usefulness of the algorithms in an important real-life application.

Keywords Graph mining · Graph clustering · Community detection · Cluster analysis · *k*-means

1 Introduction

Graph clustering is an important problem in several fields, including physics [1, 2], engineering [3], image processing [4], and the medical [5] and social sciences [6]. A cluster in a graph is a set of nodes that has more connections within the set than outside the set [7].

Clustering can be useful for understanding large networks where the numbers of nodes and edges are too large for a human analyst to examine individually. Dividing the network into separate clusters and examining the content of those clusters and their relations can be more useful for the practitioners than examining the whole network.

✉ Sami Sieranoja
sami.sieranoja@uef.fi

Pasi Fränti
pasi.franti@uef.fi

[1] Machine Learning Group, School of Computing, University of Eastern Finland, P.O. Box 111, 80101 Joensuu, Finland

The term "graph clustering" is somewhat ambiguous and can refer to very different types of clustering problems, including *network community detection* [7–13], *graph clustering* [14], *graph partitioning* [4, 15, 16], *graph set clustering* [17] or *graph based clustering*. These terms are somewhat ambiguous and often used interchangeably, which can cause confusion. They can refer to any one of the following cases:

- clustering any kind of data by first converting the whole dataset into a graph [14];
- grouping the nodes of a single graph into distinct clusters when the number of clusters is controlled by the user [4];
- finding clusters in the graph without any control on the number of clusters [7–13];
- clustering a set of graphs where each data object is a graph of its own [18, 19]; and
- separating the graph into clusters while constraining the size of the clusters [15].

In this work, we focus on producing a disjoint clustering of the nodes of a weighted undirected graph. The number of clusters is controlled by the user.

Algorithms use different strategies for the clustering. Divisive or cut-based methods [4, 20, 21] split a graph recursively into sub-networks until some stopping condition is met. Agglomerative methods [9, 22, 23] start by placing each node in its own cluster and then merging the clusters. Iterative algorithms start from some initial clustering, which is then improved via small changes, such as switching an individual node from one cluster to another. Graph growing or seed expansion [11, 24] selects a seed node and then gradually grows a cluster using (for example) best-first-search.

Many algorithms also use a cost function to guide the clustering process. For example, a cost function can be used to select the optimal split operation [4] or best partition for a given node [3]. An algorithm can include several different strategies. The Louvain algorithm [9], for instance, applies both an agglomerative approach and iterative optimization.

Two of the most popular cost functions are *modularity* [1, 2, 23, 25] and *conductance* [4, 16]. Modularity has the deficiency that it cannot be directly optimized for a specific number of clusters. It also has a tendency to produce very large clusters, which is known as *resolution limit* [26]. While conductance can be optimized for a specific number of clusters, it is sensitive to outliers and can produce unbalanced clustering with tiny clusters.

To overcome these problems, we propose two new cost functions called *inverse internal weight* (IIW) and *mean internal weight* (MIW). IIW provides more balanced clustering than the alternatives, while MIW can detect dense clusters and is good at disregarding noise and outliers.

In this paper, we also introduce two algorithms to optimize these cost functions. The first, called the K-algorithm, is a direct derivation of the classical k-means algorithm. It applies similar iterative local optimization but without the need to calculate the means. It inherits the properties of k-means clustering in terms of both good local optimization ability and the tendency to get stuck at a local optimum. The second algorithm, called the M-algorithm, gradually improves on the results of the K-algorithm. It finds new local optima, which often provide a better solution. It works by repeatedly merging and splitting random clusters and tuning the results using the K-algorithm. Both algorithms work on all of the discussed cost functions. They can also be applied for any other cost function that satisfies certain criteria (for further discussion of this point, see Sect. 4.1).

We also introduce new graph benchmark datasets that make it easy to visualize our results. We compare the proposed methods against existing state-of-the-art algorithms and cost functions on these datasets.

As a case study, we analyze connections between diseases in a *disease co-occurrence network* [27], that is, a graph where correlating diseases are connected. We constructed this

type of graph based on electronic health care records in the North Karelia region of Finland. We built two separate graphs, one for ICD10 blocks (with 188 nodes) and one for individual ICD10 disease codes (with 644 nodes). These are too large for manual investigation, but the clustering results are suitable for expert analysis. Here, we used clustering to find the most relevant connections from the network. This provides a good overview of the contents and structure of the disease network.

This type of clustering-based analysis tool is currently lacking in both the scientific literature and healthcare practices. Most community detection methods lack a proper way to control the size of the clusters, which often tend to become too large for manual investigation. This is a serious deficiency of the current algorithms and cost functions. Many algorithms are also missing a mechanism to control the number of clusters. In addition, the cost function used in the optimization may lead to very unbalanced clustering containing overly large clusters that are impossible to investigate manually.

Many papers consider it a drawback if the number of clusters needs to be specified by the user [8, 13, 28]. This is undoubtedly true in many scenarios. However, the hidden assumption of this view is that the data have some specific number of clusters and the task of the algorithm is to detect it. This scenario is often not true in real-world datasets. It may be possible to cluster one dataset in several meaningful ways, and the proper number of clusters may depend on the specific needs of the user.

In the disease co-occurrence network analysis, it is desirable to have clusters of roughly the same size, and one way of controlling this is to specify the number of clusters. If the clustering algorithm and cost function produces somewhat balanced clusters, and the goal is to split N nodes into clusters of roughly size n, then the number of clusters can be chosen simply as $k = N/n$. For example, if a network of 188 diseases is split into 10 parts, then each part will contain roughly 19 diseases—small enough that the contents of the cluster can be investigated manually (e.g., in the form of a similarity or correlation matrix).

The proposed clustering algorithms are able to work in this scenario. The clustering results reveal many interesting properties of the network. For example, we can see that mental health diagnoses are so connected to diagnoses related to alcohol and drug use that they form one cluster.

In summary, our work has the following contributions:

- We propose two new graph clustering algorithms. The algorithms work with several different cost functions and allow running time to be adjusted in a flexible manner to ensure an acceptable compromise between speed and quality.
- We propose two new cost functions: IIW, which provides more balanced clustering, and MIW, which detects dense parts as clusters and disregards noise and outliers.
- We demonstrate the usefulness of the proposed algorithms via a case study of disease co-occurrence analysis.

2 Clustering cost functions

Many methods have been proposed for estimating clustering quality in the absence of ground truth knowledge (e.g., correct partition labels). Clustering quality functions typically favor good separation of clusters (low E_i) and high internal weight (high W_i), but there are differences in how these factors are emphasized and scaled.

In this section, we review the three cost functions implemented in the proposed algorithm. We introduce two new cost functions: MIW and IIW. One of the cost functions (*conductance*)

Table 1 Clustering cost functions

Name	Type	Formula	Range	Studied in
Conductance (CND)	Minimize (↓)	$\frac{1}{k}\sum_{i=1}^{k}\frac{E_i}{T_i}$	[0, 1]	[4, 8, 12, 31]
Mean internal weight (MIW)	Maximize (↑)	$\frac{1}{k}\sum_{i=1}^{k}\frac{W_i}{n_i}$	[0, ∞]	Proposed
Inverse internal weight (IIW)	Minimize (↓)	$\frac{M}{k^2}\sum_{i=1}^{k}\frac{1}{W_i}$	[1, ∞]	Proposed

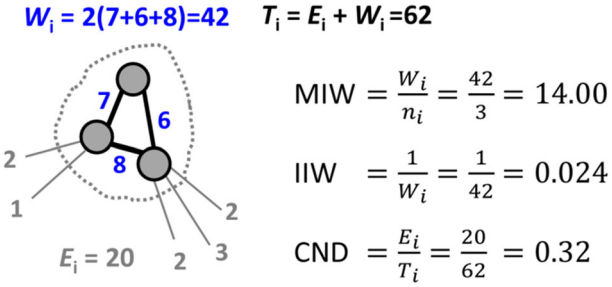


Fig. 1 Single cluster's contribution to the cost function. Most cost functions are based on internal (W_i), external (E_i), and total (T_i) weights of the clusters. The studied cost functions are additive, which means that the quality of clustering is given by summing up the contributions of individual clusters (and possibly scaling the result with some constant)

has previously been studied. All of these cost functions are additive, which means that the quality of clustering is given by the sum of qualities for individual clusters [7]. The studied cost functions are summarized in Table 1 and an example is shown in Fig. 1.

Notations

N Number of nodes
k Number of clusters
n_i Size of cluster i
w_{ij} Weight between node j and i
W_i Sum of internal weights in cluster i
W_{ij} Sum of weights from node j to nodes within cluster i
M_i Total weight (mass) of node i
M Total weight (mass) of whole graph
E_i Sum of external weights from cluster i
E_{ij} $= M_i - W_{ij}$, external weights from node j to clusters other than i
T_i $= E_i + W_{i,}$, total weight of edges connecting to nodes in cluster i

2.1 Conductance

The term *conductance* has been used in slightly different ways in several studies [7, 12, 29, 30]. In this work, we use a formulation of conductance based on a definition by Leskovec et al. [30]. We define the conductance (in Table 1) of a cluster as the weight of all external edges divided by the total weight of the nodes in the cluster. The sum of the values of individual clusters is normalized by dividing it by the number of clusters.

Minimizing conductance leads to clusters with good separation from the rest of the network (low E_i) and high internal weight (high W_i). Conductance also avoids creating overly small clusters. This can be understood by considering a case wherein a cluster consists of just a single node. Then, $E_\mathrm{i} = T_\mathrm{i}$ and conductance is 1.0 (worst) for that cluster. In the case of an empty cluster, it would be undefined (0/0), which we interpret in this case as 1.0.

2.2 Mean internal weight

The MIW cost function (Table 1) scales the internal weights W_i by dividing by the cluster size n_i. Larger values are considered better. While the unweighted version of the cost function has previously been considered by Yang and Leskovec [31], to the best of our knowledge, it has not yet been used as a target function for optimization.

The cost function favors small dense clusters because cluster size is the denominator. However, if the graph contains large and strongly connected subgraphs (i.e., almost complete graphs), it may favor large clusters instead. As an example of the second case, consider a complete graph of four nodes where all six edge weights have value 1. Splitting this graph into two clusters of sizes 2 and 2 would yield a cost function value of $2(1)/2 + 2(1)/2 = 2$, whereas keeping it as one cluster would yield the value of $2(1 + 1 + 1 + 1 + 1 + 1)/4 = 3$. In other words, having one large cluster and one empty cluster has almost the same value as equally splitting into two clusters. As a result, the cost function may sometimes produce empty clusters.

2.3 Inverse internal weight

The IIW cost function (Table 1) calculates the sum of inverse weights inside each cluster. Smaller values are considered better. The inverse weights are scaled to the range $[1,\infty]$ by multiplying them by the mean weight of a perfectly balanced cluster (M/k). In the case of optimal clustering of k completely separated and balanced clusters, all W_i would equal M/k and clustering would take the value 1.0.

There are two reasons for using the inverse weight $1/W_\mathrm{i}$ rather than the mean weight $W_\mathrm{i}/n_\mathrm{i}$. First, doing so ensures that all nodes will be assigned to a cluster to which they are connected. As an example, consider a case wherein a node is assigned to a small cluster A to which it has no connection, but there exists another large cluster B to which it does have a connection. If the node changes from cluster A to cluster B, W_A will remain unchanged, but W_B will increase, which will provide a smaller (better) cost value as expected. Mean weight, on the other hand, may do the opposite by moving the node from B to A, even if it has no connection there. This happens when the node has only a weak link to cluster B, but the penalty of the increased cluster size outweighs the effect of the node weight.

Second, inverse weighting favors more balanced clustering. If a node has an equally strong connection c to two clusters A and B, which have weights W_B and W_A so that $W_\mathrm{B} > W_\mathrm{A}$,

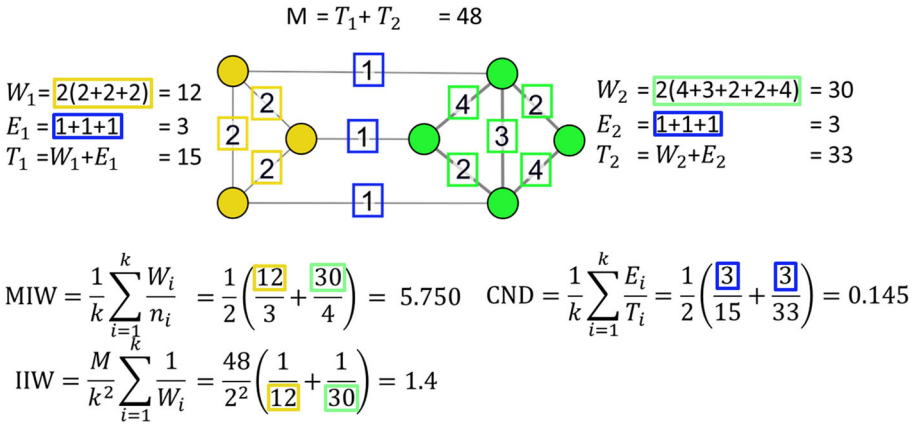


Fig. 2 Example of cost calculation of three different cost functions: mean internal weight (MIW), inverse internal weight (IIW), and conductance (CND). Every edge is counted twice, once for each node it connects

then assigning the node to cluster A would provide a more optimized result. This is because the derivative of $f = 1/x$ is $f' = -1/x^2$ and thus $f'(W_A) < f'(W_B)$.

The cost function is also guaranteed to provide exactly k clusters for the optimal cost. This can be understood by considering a case wherein one cluster is empty. Since the weight of an empty cluster is $W_i = 0$, its inverse weight would be infinite. Thus, as long as there exist sufficient data for all k clusters and there exists a clustering with finite cost, an optimal algorithm would lead to a clustering containing k non-empty clusters.

Examples of CND, MIW and IIW are provided in Fig. 2. While CND is based on the external links (E_i), the two proposed cost functions rely merely on the within-cluster statistics. This more closely follows the standard k-means cost function (sum of squared errors), which completely ignores between-cluster relations.

3 Existing algorithms

In this section, we briefly review the most relevant of the existing algorithms. For more extensive reviews, see [7, 8, 32–34].

3.1 Hierarchical algorithms

Hierarchical algorithms work in either a bottom up or top-down manner. Top-down algorithms are often referred to as *cut-based methods* [4, 20, 21]. They recursively split the graph into sub-networks until some stopping condition is met. Bottom-up methods, also known as *agglomerative clustering*, [9, 22, 23] start by placing each node in its own cluster and then merging the clusters until a similar condition is met.

The *Walktrap* [22] is an agglomerative method based on the observation that random walks in graphs often become trapped within a single cluster. The probability that two nodes will appear in the same random walk is higher if they are in the same cluster. Walktrap is based on *Ward's method*, which minimizes squared distances inside the cluster.

The *Louvain algorithm* [9] is also an agglomerative method but its goal is to minimize modularity. It starts with each node in its own cluster and then sequentially assigns nodes to the cluster that minimizes the total modularity. This process is iterated until a local optimum is reached (i.e., there is no single move of a node to another cluster that would improve the cost value). After local optimization, it reduces clusters to single nodes and starts iterating the optimization again.

The Vieclus method [35] optimizes the modularity cost function using a combination of genetic algorithm and local search. It uses clusterings from the Louvain algorithm to initialize the population. It is the current state-of-the-art for optimizing modularity. However, the Louvain algorithm has the benefit of being able to control the number of clusters.

The *NCut* method [4] minimizes conductance by formulating graph clustering as an eigenvalue problem on the similarity matrix of the graph nodes. Tabatabaei et al. [16] proposed a faster $O(N \log^2 N)$ algorithm called GANC to optimize the same cost function.

The *Sbm_dl* method [36] fits the graph into a stochastic block model that aims to find the most likely model parameters that generates the observed network. It uses greedy agglomerative heuristic which also tries to detect the correct number of clusters. The implementation also enables the minimum and maximum bounds to be given for the number of clusters it returns.

3.2 Iterative algorithms

Iterative algorithms start with an initial clustering that is improved by small changes. The changes are typically made on the cluster partitions by moving nodes (individual or small groups of nodes) from one partition to another, aiming to optimize some criterion for clustering fitness. The process stops when no change improves the solution.

An early example of this approach is the *Kernighan–Lin algorithm* [3], which aims to find an optimal way to cut a graph into two sub-graphs. The two arbitrary initial cluster partitions are improved by finding the two nodes to swap between partitions that lead to the largest improvement in the minimum cut criterion. This is continued until no further improvement is possible.

The *Gemsec* method embeds graphs into vector space and performs centroid based clustering in the vector space [37]. It combines node embedding cost and sum of squared error clustering cost into the same cost function and then uses gradient based iterative optimization for this cost function.

3.3 Balanced clustering

Several methods [38–40] aim to achieve a balanced clustering by minimizing the total cut on the graph (E_i) while constraining the maximum cluster size to $(1 + \varepsilon)$ times the average cluster size. The constraint parameter is typically a small number (≤ 0.05). Complete balance ($\varepsilon = 0$) has also been considered [40]. *KaffpaE* is an evolutionary algorithm that optimizes this cost function. It uses a novel combine operator that guarantees that the fitness of the new offspring will be at least as good as that of the best of the parents.

The *FluidC* algorithm [41] is based on the idea of fluids (communities) interacting by expanding and pushing each other in an environment (graph) until a stable state is reached. It utilizes an update rule that maximizes the number of connections within a cluster scaled by the inverse of the number of vertices composing a cluster. This guarantees k non-empty clusters.

3.4 Graph growing and seed expansion

Clusters can also be formed by gradually growing them from a seed node, typically by using breadth-first search. Karypis and Kumar [24] aimed to achieve an equally sized split of the graph by starting from a random node and then expanding the cluster until half of the vertices are included. This is repeated 10 times. The result with the smallest edge cut is selected as an intermediate result which is improved using the Kernighan–Lin algorithm.

Whang et al. [11] aimed to find overlapping clusters using a process which they called seed expansion. They used several heuristics to select good seeds. One of these heuristics selects nodes in the order of node degree, while disregarding the neighbors of the node in subsequent selections. Clusters are then grown from the seeds by greedily optimizing conductance in each step.

4 *K*-algorithm and *M*-algorithm

Although many iterative and hierarchical algorithms for graph clustering exist, a k-means based solution is still missing. The main reason for this is that k-means requires calculating the mean of a cluster, which is seemingly not possible for graphs. However, we implement a sequential variant of k-means which assigns the nodes to the closest cluster without calculating distance to the mean. This is done by using a *delta approach*, which calculates the change in the cost function value before and after the assignment. The proposed algorithm is called the *K-algorithm*; as it resembles the k-means algorithm but without using means. In Sect. 4.4, we also introduce a merge-and-split based *M-algorithm* which improves on the results of the *K*-algorithm.

4.1 *K*-algorithm: greedy local optimization

The K-algorithm (Algorithm 1) starts from an initial solution and then iteratively tunes it toward a better solution. It can be used to improve any initial clustering. Nonetheless, the quality of the results depends heavily on the initial clustering. We, therefore, introduce a density-based initialization method (Sect. 4.3) that is more effective than a simple random partitioning. The effect of the initialization is studied in Sect. 6.3.

The *K*-algorithm iteratively improves the initial solution by processing the nodes sequentially, in random order (line 5). For each node, the method considers all clusters (line 14). It changes the cluster of the node if this improves the cost function (line 18). The delta calculation (line 15) for the selected cost functions requires recalculating W_{xi}, the sum of weights from node i to all possible clusters x. This is done by looping through all edges adjacent to node i (line 10) and adding to the weight sum of their cluster.

After all nodes have been processed, the algorithm starts another iteration. The iterations continue until no changes occur and the cost function can no longer be improved by changing the partition of a single node (line 22).

```
Algorithm 1: K-algorithm (graph,k,cluster)
INPUT:
   graph (with N nodes)
   k = number of clusters
   cluster = initial clustering (optional)
 1 IF cluster == NULL
 2   cluster = InitialPartition(graph,k)
 3 DO
 4   changed = 0
 5   FOR i=SHUFFLE(1:N) // Process values 1..N in random order
 6     old = cluster[i]
 7     newpart = 1
 8     bestdelta = INF
 9     W_xi = 0 for all x
10     FOR j=1:SIZE(graph[i]) // Loop all connections of i
11       (nodeId,weight) = graph[i][j]
12        x = cluster[nodeId]
13        W_xi += 2*weight
14     FOR y=1:k
15       d = Δf(i,cluster[i],j) // Depends on W_xi (see Section 4.2)
16       IF d < bestdelta
17         bestdelta = d
18         newpart = j
19     IF newpart != old
20       changed += 1
21       cluster[i] = new
22 WHILE changed > 0
23 RETURN cluster
```

In theory, the *K*-algorithm can work with any cost function. In practice, there are a few constraints. First, the delta of a given node should be calculated quickly without looping through the entire dataset. Second, the optimal value for a cost function should produce k non-empty clusters when k is a parameter of the algorithm.

4.2 Cost function delta calculation

A key part of the *K*-algorithm is finding the partition with the best cost function for a given node (lines 14–18). Finding this partition requires calculating the delta value of moving the node in question from its current partition x to another partition y for all possible partitions.

The delta value can be calculated by creating a new clustering ($C \longrightarrow C'$) where the node j is moved from its old cluster X to a new cluster Y. The delta is then calculated as the difference between these two clusterings:

$$\Delta f(j, X, Y) = f\left(C'\right) - f(C) \tag{1}$$

However, in practice, this is very slow, since calculating the cost function for an entire dataset requires looping through all edges of the graph. It is therefore better to directly calculate the delta value. Since the cost functions are additive (see Figs. 1, 2), only changes in cluster X, from which the node is removed, and cluster Y, to which it is added, affect the delta:

$$\Delta f(j, X, Y) = \Delta f(C_X) + \Delta f(C_Y) \tag{2}$$

These can be further split into cost before change (C_X, C_Y) and cost after change (C_X', C_Y'):

$$\Delta f(j, X, Y) = f\left(C_X'\right) + f\left(C_Y'\right) - f(C_X) - f(C_Y) \tag{3}$$

These components of the delta are calculated differently for each cost function. In the case of mean weight, they are calculated as follows:

$$f\left(C_x'\right) = \frac{W_x - W_{xj}}{n_x - 1} \quad f\left(C_Y'\right) = \frac{W_y + W_{yj}}{n_y + 1} \quad f(C_x) = \frac{W_x}{n_x} \quad f(C_Y) = \frac{W_y}{n_y} \tag{4}$$

The full delta function for mean weight (MIW) is therefore:

$$\Delta\mathrm{MIW}(j, X, Y) = \frac{\left(W_x - W_{xj}\right)}{n_x - 1} + \frac{\left(W_y + W_{yj}\right)}{n_y + 1} - \frac{W_x}{n_x} - \frac{W_y}{n_y} \tag{5}$$

Applying a similar derivation in the case of IIW yields the following:

$$\Delta\mathrm{IIW(j,X,Y)} = \frac{1}{\left(W_x - W_{xj}\right)} + \frac{1}{\left(W_y + W_{yj}\right)} - \frac{1}{W_x} - \frac{1}{W_y} \tag{6}$$

For conductance (CND), where $T_x - W_x = E_x$, we get:

$$\Delta\mathrm{CND}(j, X, Y) = \frac{\left(T_x - M_j\right) - \left(W_x - W_{xj}\right)}{T_x - M_j} + \frac{\left(T_y + M_j\right) - \left(W_y + W_{yj}\right)}{T_y + M_j} - \frac{T_x - W_x}{T_x} - \frac{T_y - W_y}{T_y} \tag{7}$$

4.3 Density based initialization

The initialization method (Algorithm 2) grows new clusters into dense parts of the graph. The nodes are sorted based on density, and the densest (central) node is chosen as the seed node for the first cluster. A new cluster is then grown to this position by expanding from the seed node (line 10). The next cluster is grown from the second densest node that has not yet been assigned to another cluster.

Algorithm 2: InitialPartition(graph,k)

```
 1 FOR i=1:N
 2   cluster[i] = NULL
 3 graph=SORT(graph,density)
 4 seed=cluId=1
 5 WHILE cluId <= k AND seed <= N
 6   // Grow a new cluster from the highest density node
 7   // that is not already assigned to a cluster
 8   WHILE cluster[seed] != NULL
 9     seed += 1
10   cluster = GrowCluster(graph,cluster,cluId,seed,0.8*(N/k))
11   cluId += 1
12 FOR i=1:N
13   IF cluster[i] == NULL
14     cluster[i] = RAND(1,k)
15 RETURN cluster
```

Cluster growing (Algorithm 3) is used both in the density-based initialization and later in the merge-and-split algorithm (Sect. 4.4). It starts by creating a new cluster C_i that consists of a single seed node. New nodes are then merged to it in a best-first search manner. That is, the nodes are sequentially added so that the algorithm always picks the node j that has the highest sum of edge weights (W_{ij}) to the cluster i (variable nodeToCluWeight) and is not yet assigned to another cluster. This continues until a specific cluster size is reached (line 3), or there are no more nodes with $W_{ij} > 0$ (line 6).

In density-based initialization, the size of the grown cluster is selected as 80% of the average cluster size ($0.8(N/k)$). This means that 20% of the nodes are left without a cluster label. For these, the labels are chosen randomly (Algorithm 2, lines 12–14). The *K*algorithm later fine-tunes these to more suitable partitions.

It is sufficient to cover 80% of the dataset with the cluster growing and leave the rest for the K-algorithm to optimize. If the cluster growing was to cover 100% of the dataset, the last clusters would sometimes be too fragmented for the K-algorithm to optimize.

Algorithm 3: GrowCluster(graph,cluster,cluId,seedId,growSize)

```
 1 nodeToCluWeight = [0,0...0]
 2 nodeToCluWeight[seed]=1
 3 FOR i=1:growSize
 4   // External node with highest weight to current cluster
 5   nodeToAdd = arg max (nodeToCluWeight[x])
 6   IF nodeToCluWeight[nodeToAdd] <= 0
 7     RETURN cluster
 8   cluster[nodeToAdd] = cluId
 9   nodeToCluWeight[nodeId] = 0
10   // Loop all connections
11   FOR j=1:SIZE(graph[nodeToAdd])
12     (nodeId,weight) = graph[nodeToAdd][j]
13     IF nodeId not already in cluster
14       nodeToCluWeight[nodeId] += weight
15 RETURN cluster
```

The initialization method depends on the estimation of node density. This is calculated using Eq. (8) by looping all connected nodes and multiplying the neighbor node's total weight (M_j) by the edge weight (w_{ij}) connecting to that neighbor. The idea is that a node can be considered central if it has strong ties to other nodes that are themselves central.

$$dens(i) = \sum_{j \in G(i)} w_{ij} M_j \tag{8}$$

Here, $G(i)$ represents the set of nodes to which node i is connected ($w_{ij} > 0$). In our data, the size of $G(i)$ is only a small fraction of the size of the data, N. If $G(i)$ were in the order of $O(N)$ (e.g., in case of complete graph), it would increase the time complexity of the initialization to $O(N^2)$. In this case, it would make sense to limit $G(i)$ to only the most significant edges.

4.4 *M*-algorithm

In most cases, the *K*-algorithm clusters the dataset fast and produces reasonable results. However, it always converges to a local maximum wherein no single change in a node partition can improve the cost function. This can sometimes be the correct clustering, but

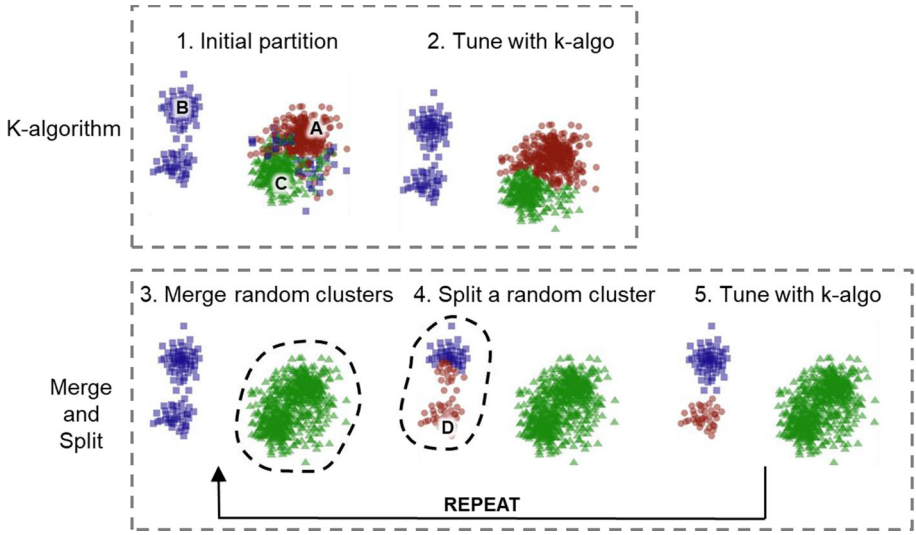


Fig. 3 The *K*-algorithm needs an initial clustering as a starting point. We use density-based initialization (part 1) which grows new clusters from dense parts of the graph (nodes A, B and C). The local optimization of the K-algorithm (part 2) can then fine-tune this initial clustering. The mergeandsplit algorithm can further improve the solution by first merging a random pair of clusters (part 3) and then splitting one cluster (part 4) by growing a new cluster from random node D. The results are fine-tuned with the *K*-algorithm (part 5). The algorithm usually needs to be repeated several times for successful clustering

often there is room for further improvement. For this reason, we will next introduce the M-algorithm which is based on a merge-and-split strategy.

The *M*-algorithm (Algorithm 4) repeats the *K*-algorithm multiple times. It disrupts the stable state produced by the *K*-algorithm by first merging two random clusters (line 5) and then splitting one random cluster (lines 8–12). This effectively reallocates one cluster into a different part of the data space. The solution is then fine-tuned to a new local optimum using the K-algorithm. If this new solution improves on the previous local optimum (line 15), it is kept as the current solution; otherwise, it is discarded. The merge-and-split process (Fig. 3) is repeated multiple times (line 2). The number of repeats is a user-given parameter that enables a flexible tradeoff between clustering quality and processing time.

It is usually good to merge clusters only if there are some links between them. Therefore, we choose the two clusters according to a probability defined by the strength of the connections between them. If the clusters have a strong connection, the merge probability is higher. If there are no connections between them, the merge probability is zero. For a pair of clusters (A, B), the merge probability is defined as:

$$p(A, B) = \frac{C_{AB}}{E} \tag{9}$$

Here, the variable E is the total sum of weights between all clusters ($E = \sum E_i$), and C_{AB} is the total sum of weights between clusters A and B. Therefore, summing $p(A,B)$ for all possible pairs yields the value 1.0.

We perform the split by selecting a random cluster and then growing a new cluster from a random node inside that cluster. The size of the new cluster is selected randomly between

5 and 95% of the size of the original cluster. This sometimes produces unequal splits and therefore offers a better chance to also solve datasets that have clusters of unequal sizes.

```
Algorithm 4: MergeAndSplit(graph,k,R)
INPUT:
   graph (with N nodes)
   k = number of clusters
   R = number of repeats
 1 K-algorithm(graph,k,NULL)
 2 FOR i=1:R
 3   newClu = cluster
 4   (A,B) = choose randomly, according to Equation 9
 5   newClu = MERGE(newClu, A,B)
 6
 7   // Perform a split
 8   cluId = RAND(1,k)
 9   unbalanceFactor = RAND(0.05,0.95)
10   growSize = unbalanceFactor*SIZE(cluId)
11   seedId = random node from cluster cluid
12   newClu = GrowCluster(graph,newClu,cluId,seedId,growSize)
13
14   newClu = K-algorithm(graph,k,newClu)
15   IF cost(graph,newClu) > cost(graph,cluster) // improvement
16      cluster = newClu
17 RETURN cluster
```

The merge-and-split process (Fig. 3) is repeated multiple times (line 2). The number of repeats R is a user-given parameter that enables a flexible tradeoff between clustering quality and processing time.

4.5 Time complexity

The time complexity of the K-algorithm (Algorithm 1) can be calculated by analyzing the number of times a line is executed. The first for-loop (line 5) processes all N points. Inside this loop, the recalculation of W_{xi} loops all connections in $O(|E|/N)$ steps and the calculation of the delta value for all clusters takes k steps. This makes the first for loop have a total of $O(N(k+|E|/N))$ steps. Other variables needed by the delta calculation (W_{i}, E_{i}), also need to be updated after changing the cluster of a node but these can be updated in $O(1)$ time.

The entire process (lines 3–22) is repeated for I iterations, until the algorithm converges. This makes the total time complexity of the K-algorithm $O(IN(k+|E|/N))$. I is a small number that ranged from 4 to 78 in our experiments. It increases slightly with dataset size and other properties (see Table 5).

The initialization of the K-algorithm (Algorithm 2) is not a bottleneck in the algorithm. The time complexity of the initialization is k times (line 5) that of growing a new cluster. The time complexity of cluster growing (Algorithm 3) depends on the size of the cluster (line 3), which is $O(N/k)$, and on the average number of edges (line 11) which is $O(|E|/N)$. This makes the time complexity of growing one cluster $O(|E|/k)$ and the initialization as a whole $O(|E|)$.

The time complexity of the M-algorithm (Algorithm 4) is determined by the number of times the K-algorithm is repeated (line 14) and is $O(RIN(k+|E|/N))$ in total where R is the

number of repeats. Other parts of the algorithm, the merge and split operations are minor compared to the $O(IN(k+|E|/N))$ complexity of the K-algorithm. The merge operation (line 5) is trivial and takes $O(N/k)$ time. The splitting is done by cluster growing (line 12) which is $O(|E|/k)$.

The time complexity of both the K- and M-algorithms is therefore almost linear $O(N)$ with respect to the number of nodes. This can be verified in Fig. 5 in the experimental results.

5 Experimental setup

5.1 Datasets

We performed our experiments using three types of datasets (see Table 2):

- a k-nearest neighbors (kNN) graph of numerical datasets;
- artificial graphs; and
- disease comorbidity networks.

To enable visual inspection of clustering results, we generated kNN graphs with parameter $k = 30$ from selected numerical 2D datasets taken from the clustering basic benchmark [42]. We used the datasets $S1$-$S4$ and *Unbalance*, which we converted to kNN graphs using the method in [43]. The resulting graphs were used as input for the clustering algorithm.

The distances in the graph are converted to weights (similarities) as follows:

$$w_{xy} = \frac{\max(d) - d(x, y)}{\max(d)} \tag{10}$$

where $\max(d)$ is the maximum distance in the entire graph.

We also generated three additional 2D sets (G3A, G3B, UNI) to illustrate the properties of the cost functions. The sets G3A and G3B contain three separate Gaussian clusters, each of which contains both a very dense and very sparse (low- and high variance) area. The clusters

Table 2 Datasets

Dataset	Graph type	Nodes	Avg. degree	Clusters	μ_t
s1	kNN	5000	30	15	–
s2	kNN	5000	30	15	–
s3	kNN	5000	30	15	–
s4	kNN	5000	30	15	–
Unbalance	kNN	6500	30	8	–
G3A	kNN	900	30	3	–
G3B	kNN	1800	30	3	–
UNI	kNN	900	30	–	–
icdA	Disease comorbidity	644	53	19	–
icdB	Disease comorbidity	188	50	19	–
varN	Artifical	1 k...1024 k	30	30	0.63
varDeg	Artifical	5000	4...100	30	0.65
varMu	Artifical	5000	30	30	0.60...0.80

in G3B are more overlapping (less separated) than in G3A. The UNI dataset contains random noise uniformly distributed in the 2D plane.

The icdA and icdB datasets are *disease comorbidity networks* [27] where selected diseases represent nodes in the graph. The diseases have been recorded using the International Classification of Diseases (ICD10) [44]. In the ICD10, diagnoses are organized in a hierarchy where the first letter of a code refers to the category of the diagnosis. For example, in code F31, F indicates that the diagnosis relates to *mental and behavioral disorders*. F31 specifies a diagnosis of *bipolar affective disorder*. Codes are further grouped to form blocks such as F30-F39 for *mood [affective] disorders*, and F20-F29 for *schizophrenia, schizotypal and delusional disorders*. We constructed two variants of the network, each corresponding to different abstraction levels. The first network, named *icdA*, consists of individual diseases (e.g., F31) as the nodes. The second, named *icdB*, consists of the blocks (e.g., F20-F29).

Two diseases are linked if they co-occur in the same patients with sufficient frequency that they are correlated. We use *relative risk* [27] to measure the strength of the connection of the diseases. These networks were constructed using data for 51,909 patients. Data for this study were obtained from a regional electronic patient database. We use the first letter of the ICD10 coding scheme as the ground truth for the cluster labels. In the ICD data, most relative risk values are between 0.5 and 5.0 but they can also be over 100. These outliers would dominate the cost function optimization, and for this reason, we normalize them to the range of [0,1] as follows:

$$w'_{\mathrm{xy}} = \frac{w_{\mathrm{xy}}}{1 + w_{\mathrm{xy}}} \tag{11}$$

We also generated three series of artificial graph datasets using the weighted network benchmark generation tool proposed by Lancichinetti and Fortunato [10]. One key parameter in this tool is the mixing parameter μ_t, which controls the proportion of links inside versus between clusters. An μ_t value of 0 indicates that the clusters are completely separated; 0.5 means that 50% of the links are within the clusters; and 1 indicates that the clusters are completely overlapping. We generated the varMu series by varying μ_t between 0.60 and 0.80. In the varDeg series, the average degree was varied, and in the varN series, the size of the dataset was varied. For details, see Table 2.

5.2 Measuring clustering quality

To evaluate clustering quality with respect to the ground truth, we used two methods: *normalized mutual information* (NMI) [45] and *centroid index* (CI) [46]. The NMI values are within the range [0,1], where 1.0 indicates correct clustering. The CI value represents the number of incorrectly detected clusters in the clustering results so that a value of 0 indicates a correct clustering. The CI gives an intuitive cluster-level overview of the correctness of the clustering, whereas NMI provides a more fine-grained, data-level measure.

5.3 Compared methods

Table 3 summarizes the algorithms used in our experiments (see Sect. 3). We compare the proposed methods (*K*-algorithm and M-algorithm) against eight other algorithms. We selected algorithms that (1) took the number of clusters k as a parameter, (2) could cluster weighted networks, (3) were fast enough to run on a dataset of size 6500, and (4) had a publicly available implementation. We found five algorithms (in Table 3) that matched these

Table 3 Compared graph clustering methods

Algorithm	Cost function	Abbreviation	Language	References
Fluidc	Density	Fluidc	Python	[41]
Gemsec	SSE + embedding	Gemsec	Python	[37]
KaffpaE	Balance constrained total cut	KaffpaE	C + +	[39]
Louvain	Modularity	Louvain	C	[9]
Normalized Cut	Conductance	ncut	Matlab	[4]
Sbm_dl	Maximum likelihood	sbmdl	Python	[36]
Vieclus	Modularity	Vieclus	C + +	[35]
Walktrap	Sum of squared distances	Walktrap	C	[22]
K-algorithm	Inverse internal weight	K-IIW	C	Proposed
M-algorithm	Conductance	M-cnd	C	Proposed
M-algorithm	Inverse internal weight	M-IIW	C	Proposed
M-algorithm	Mean internal weight	M-MIW	C	Proposed

criteria. We also included the Vieclus and Louvain algorithms, which do not take the number of clusters as a parameter but fit the other criteria. FluidC was targeted for unweighted graphs. We also ran the standard k-means algorithm for datasets with numerical representations. We used the implementations for the methods sbm_dl, gemsec, and fluidc from the CDlib package [47] and Louvain and Walktrap from the iGraph library.[1]

We mainly used the default parameters for the algorithms. For KaffpaE, we set the balance constraint $\varepsilon = 20\%$ to match that of the artificial graphs. The Vieclus algorithm required a time limit (seconds) as a parameter which we set according to the formula $N/100$ (10 s for dataset of size 1000).

Running time was measured as run on a single core. As the NCut implementation uses parallel processing, its running times are not comparable to others. The experiments were run on a Ubuntu 20.04 server with 20 processing cores (Intel® Xeon® W-2255 CPU @ 3.70 GHz).

6 Results

In this section, we examine the experimental results in five different parts. We first visually compare the three cost functions using kNN graph datasets. We then show the numerical evaluation results of the benchmark data with respect to the ground truth. We also study the effect of the initialization on the K-algorithm's performance. Next, we study how the dataset parameters affect the number of iterations. Finally, we demonstrate the usefulness of the method using disease comorbidity analysis as a case study.

6.1 Visual cost function comparison

We visually compared the cost functions to identify characteristics that simple clustering validity functions like NMI may hide. The usefulness of the cost function also depends on the clustering application. Therefore, the choice of a cost function should be left up to the

[1] https://igraph.org/

application specialist. Our goal is to provide information that will be useful in making such decisions.

We performed the visual analysis based on kNN graphs of numerical 2D datasets. The graph is first clustered using the proposed method. To ensure that the results were as close to optimal as possible, we ran the M-algorithm for 100,000 splits and merges. The resulting partitions are visualized using the original 2D dataset.

The results in Fig. 4 show that in the two extreme cases—that is, uniformly distributed random data and three clearly separate clusters (with the correct k parameter)—there are no notable differences between cost functions. The differences become visible in cases between the extremes, when there exists (a) overlap between clusters, (b) random noise in the clusters, or (c) a different parameter k compared with the ground truth.

The MIW cost function tends to form small compact clusters composed of nodes in the dense area. These clusters contribute most to the cost function value. Optimizing the cost function also forms one garbage cluster containing all the nodes in sparse areas. This can be perceived either as an error or as a type of outlier detection that filters out less relevant information. When $k = 5$, the other cost functions lead to splitting the denser areas instead of the sparse areas.

Optimizing IIW leads to more balanced clustering. This can be seen especially in the case of Gaussian clusters with $k = 2$, when conductance and MIW both split the three clusters into one part consisting of two clusters and one part containing only one cluster. IIW, on the other hand, splits the dataset into two equally sized parts containing 1.5 clusters. Otherwise, the results are similar to those for conductance.

In the case of completely uniform data, we would expect clusters of equal size, since the dataset does not have any visible clustering structure. However, MIW results in unbalanced cluster sizes, where the largest partition is double the size of the smallest partition. Conductance also provides noticeably unbalanced results. IIW results in the most balanced clustering.

In summary, we recommend that practitioners select their cost function depending on the application:

- IIW is preferable when more balanced clustering is desired.
- MIW is preferable for detecting dense clusters and disregarding noise and outliers. It tends to form one large garbage cluster from the noise points. Therefore, the number of clusters should be set to $k + 1$ if k true clusters are required.
- Conductance generally worked well in many of the tested cases, but it may lead to very unbalanced clustering if the data have isolated tiny clusters with external weight E_i close to 0.

6.2 Algorithm benchmark

In this section, we present a clustering quality comparison of the proposed method (K-algorithm and M-algorithm) with eight other algorithms (Table 3). We ran the M-algorithm for $R = 100$ merge-and-split iterations. We ran each of the tests 10 times and present the mean values of the results.

The results are reported in Fig. 5 and Table 4. The prefixes K- and M- refer to the K- and M-algorithms. Conductance, IIW, and MIW are referred to by the suffixes -CND, -INV, and -MIW, respectively.

In the varN tests (Fig. 5), where data size is varied, the K-algorithm, M-algorithm, and Gemsec clearly stood out from the rest. However, Gemsec's running time was an order of

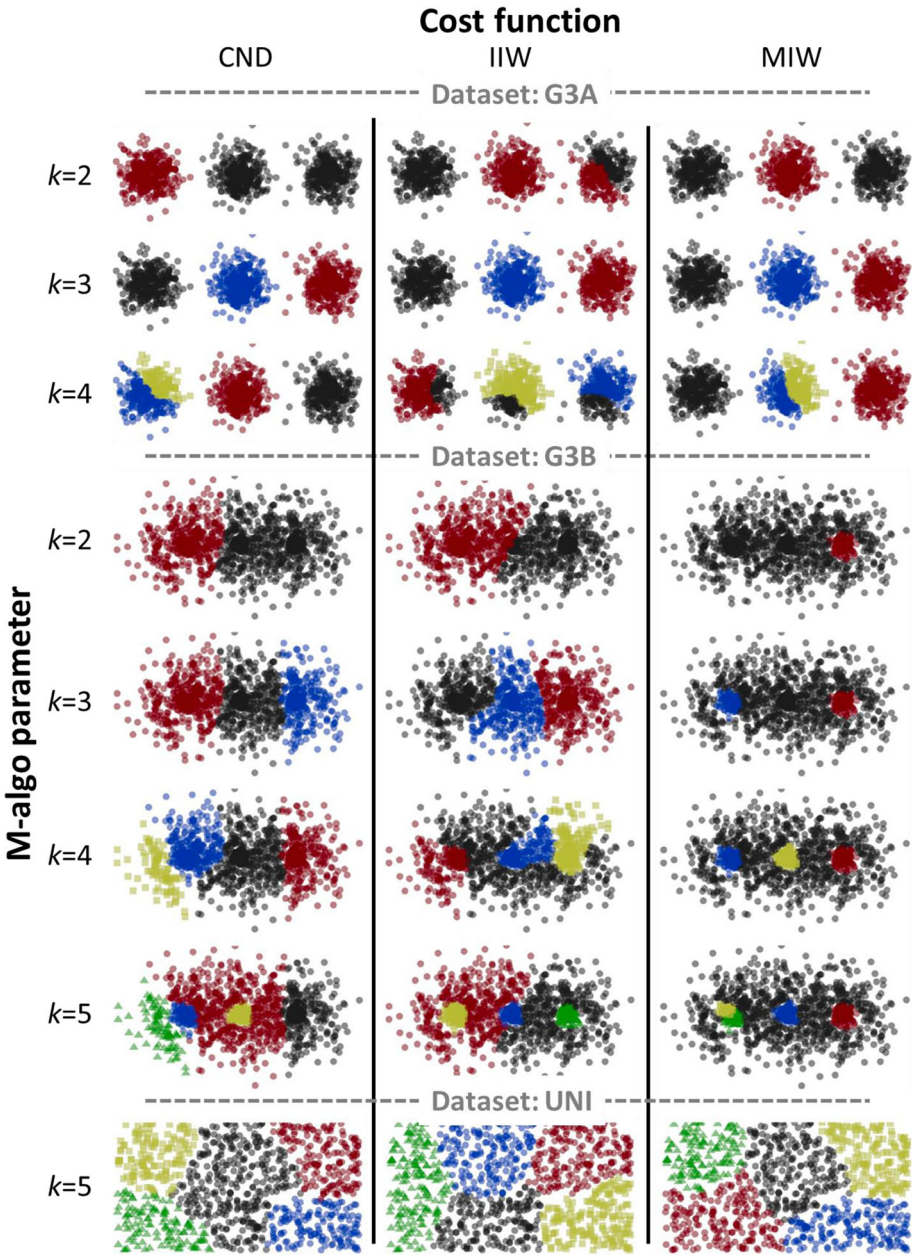


Fig. 4 Comparison of the three cost functions: conductance (CND), inverse internal weight (IIW), and mean internal weight (MIW). Three different datasets are used: three separate clusters (G3A), three sparse clusters with a dense middle area (G3B), and uniformly distributed random noise (UNI). We vary the algorithm parameter k (the number of clusters) to show how cost functions behave when the parameter is different from ground truth. Colors are determined by the size of the cluster. Black indicates the largest cluster, followed by red, blue, yellow, and green (in descending order)

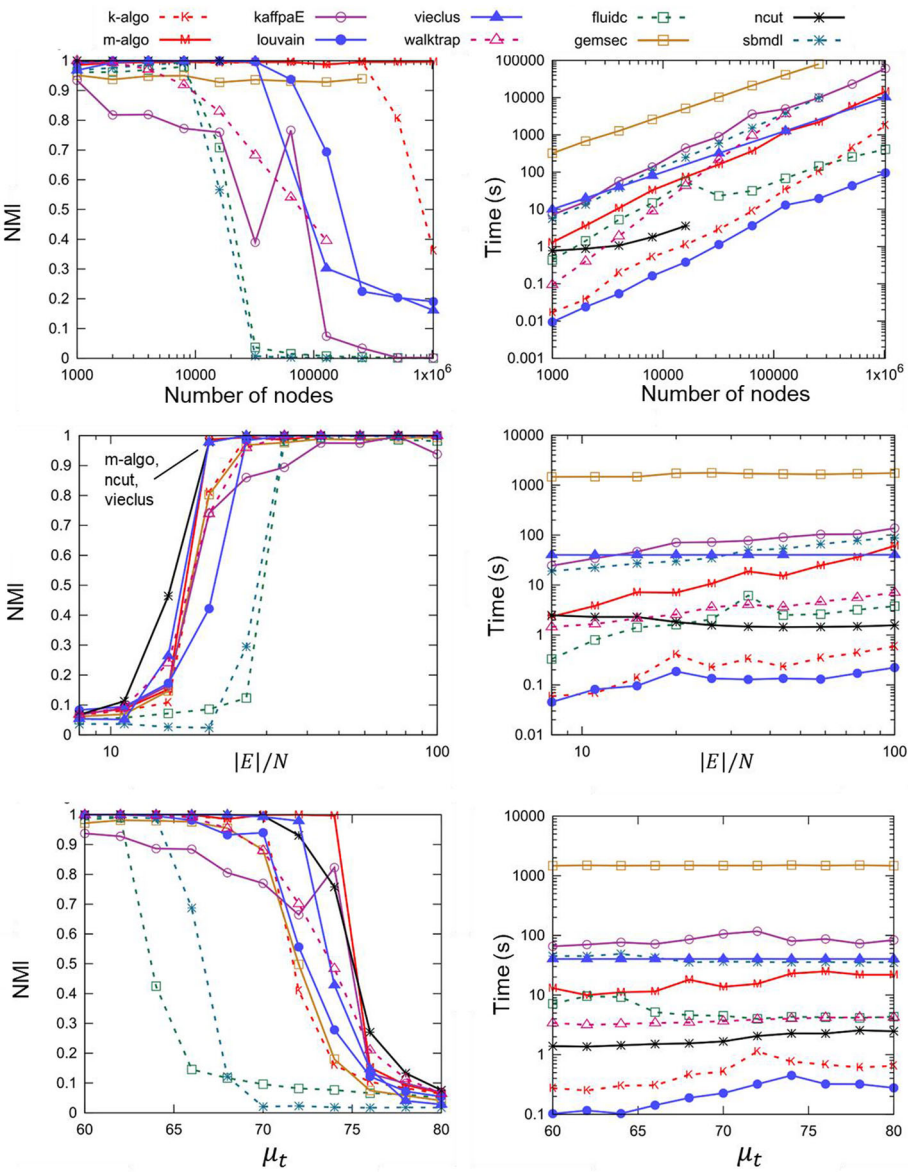


Fig. 5 Results for varying number of nodes N (varN) are shown on top, average degree $|E|/N$ (varDeg) in the middle, and mixing parameter μ_t (varMu) on the bottom. Measurement of quality (NMI) is on the left, and time (seconds) is on the right. In the case of the K- and M-algorithms, we used the IIW cost function. The NCut runtimes are not comparable, as the implementation used parallel processing

Table 4 Summary of results: *k*NN datasets on the left, comorbidity network in the middle, and artificial graphs on the right

Method	s1	s2	s3	s4	unb	G3	imA	imB	varMu $\mu = 74\%$	varDeg $\|E\|/N = 20$	varN $N = 128$ k	Mean
NMI												
K-algo-inv	0.98	0.95	0.80	0.69	0.57	0.64	0.31	0.50	0.16	0.81	0.99	0.67
M-algo-cond	0.98	0.94	0.78	0.68	0.86	0.70	0.28	0.48	0.81	0.99	0.98	0.77
M-algo-inv	0.99	0.95	0.80	0.71	0.63	0.71	0.35	0.51	1.00	0.99	0.99	**0.78**
M-algo-meanw	0.96	0.92	0.78	0.63	0.73	0.51	0.30	0.52	0.62	0.97	0.40	0.67
k-means	0.94	0.91	0.77	0.71	0.80	0.58	–	–	–	–	–	–
fluidc	0.89	0.87	0.69	0.64	0.72	0.56	0.32	0.49	0.08	0.09	0.01	0.49
gemsec	0.99	0.96	0.63	0.66	0.79	0.22	0.28	0.33	0.18	0.80	0.93	0.62
kaffpaE	0.99	0.95	0.80	0.72	0.71	0.71	0.27	0.45	0.82	0.74	0.07	0.66
louvain	0.99	0.94	0.79	0.72	0.65	0.47	0.17	0.28	0.28	0.42	0.69	0.58
ncut	0.99	0.95	0.80	0.72	1.00	0.73	0.24	0.37	0.76	0.98	–	0.75
sbmdl	0.89	0.87	0.78	0.71	0.74	0.46	0.24	0.44	0.02	0.02	0.00	0.47
vieclus	0.99	0.95	0.80	0.72	0.64	0.48	0.13	0.24	0.43	0.98	0.30	0.61
walktrap	0.99	0.94	0.78	0.69	1.00	0.58	0.06	0.27	0.48	0.74	0.40	0.63
	s1	s2	s3	s4	unb	G3	imA	imB	varMu $\mu = 74\%$	varDeg $\|E\|/N = 20$	varN $N = 128$ k	Mean
CI												
K-algo-inv	0.0	0.0	0.0	0.3	5.0	0.0	7.7	5.1	9.2	2.3	0.0	2.7
M-algo-cond	0.0	0.0	0.0	1.0	1.4	0.0	10.4	7.6	7.0	1.2	1.4	2.7
M-algo-inv	0.0	0.0	0.0	0.0	5.0	0.0	7.1	6.6	0.0	0.0	0.0	**1.7**

Table 4 (continued)

	s1	s2	s3	s4	unb	G3	imA	imB	varMu $\mu = 74\%$	varDeg $\|E\|/N = 20$	varN $N = 128$ k	Mean
M-algo-meanw	0.0	0.0	0.8	2.4	2.2	0.0	15.8	7.8	11.2	2.0	29.0	6.5
k-means	1.8	1.4	1.3	0.9	3.2	0.6	–	–	–	–	–	–
fluidc	2.6	1.9	2.4	2.5	4.5	0.1	9.1	8.5	10.6	9.4	15.9	6.1
gemsec	0.0	0.0	4.0	1.0	1.0	2.0	8.0	11.0	8.0	0.0	0.0	3.2
kaffpaE	0.0	0.0	0.0	0.0	4.6	0.0	8.1	6.0	5.0	4.8	15.0	4.0
louvain	0.0	0.0	0.0	1.5	13.1	9.1	12.0	16.0	17.0	11.5	23.0	9.4
ncut	0.0	0.0	0.0	1.0	0.0	0.0	12.0	14.0	0.0	0.0	–	2.7
sbmdl	2.0	1.3	0.6	0.1	4.1	0.4	12.2	9.2	30.0	27.4	29.0	10.6
vieclus	0.0	0.0	0.0	1.0	13.2	8.0	15.0	16.0	16.5	2.1	794.4	78.7
walktrap	0.0	0.0	0.0	0.0	0.0	0.0	18.0	16.0	0.0	0.0	4.0	3.5

Results for varMu, varDeg, and varN are selected from the experiments shown in Fig. 5

Table 5 Effect of dataset properties on number of K-algorithm iterations by varying the parameters N, μ_t, and $|E|/N$ (Table 1)

N	Iterations	μ_t	Iterations	$\|E\|/N$	Iterations
1	6	50	4	4	9
2	6	52	4	5	9
4	9	54	5	6	10
8	8	56	5	8	12
16	8	58	8	11	14
32	9	60	8	15	20
64	13	62	8	20	53
128	21	64	8	26	13
256	22	66	10	34	11
512	42	68	16	44	4
1024	78	70	18	58	5
		72	50	76	3
		74	31	100	2
		76	27		
		78	25		
		80	26		

magnitude greater than the other methods, and it could not be run for the largest datasets. The M-algorithm was the only algorithm that could solve the largest dataset (1,024,000 nodes). The running times for most of the algorithms seem close to linear.

When varying the number of edges (varDeg; Fig. 5), the M-algorithm, NCut, and Vieclus clearly perform better than the others. A smaller number of edges also decreases the number of K-algorithm iterations (Table 5), which likely contributes to the K-algorithm's poor performance compared with the varN tests. In most algorithms, running times increase slightly with larger |E|/N, but in the M- and K-algorithms, this increase is greater.

In the case of the disease co-occurrence network datasets (icdA and icdB), the M-algorithm with IIW clearly performed better than the other methods. However, the ground truth here is the existing human-made disease-level categorization. The algorithms might find different (possibly better) clustering than the human categorization. A CI = 0 result is therefore not expected with the imA and imB datasets.

No algorithm performed best on all datasets. However, the M-algorithm had the best overall performance in terms of quality, as shown in Table 4 (CI = 1.7, NMI = 0.78). NCut (CI = 2.7, NMI = 0.75) was second and the K-algorithm (CI = 2.7, NMI = 0.67) third. KaffpaE (CI = 4.0, NMI = 0.66) and Walktrap (CI = 3.5, NMI = 0.63) performed well on the kNN-based datasets (S1–G3) but had worse results for the Lancichinetti benchmark (Fig. 5). Vieclus (CI = 78.7, NMI = 0.61), which also determined the number of clusters by optimizing modularity, performed well on the Lancichinetti benchmark but not as well for the kNN graphs. It obtained the right number of clusters for many datasets but failed in other cases. It output 796 clusters (30 in the ground truth) for the dataset of size 1,024,000. This result highlights the need to be able to control the number of clusters.

The main weakness of both the M-algorithm and the IIW cost function is data with large cluster-size unbalance, as in unb, which has 20:1 unbalanced and well-separated clusters. For this type of data, the hierarchical approach performs better, and NCut and Walktrap achieved the best results. However, in additional tests, we were also able to solve this dataset with the M-algorithm and the conductance cost function if the number of repeats was increased from 100 to 1000.

The running times in Fig. 5 show that the K-algorithm was the second fastest (after the Louvain algorithm) in most cases. The quality of its results was also good. Table 4 shows that the *K*-algorithm was second best (tied with NCut) according to CI or third best according to NMI. The M-algorithm also improved on the *K*-algorithm's results for almost all datasets except the easiest sets in the varN, varDeg, and varMu series and s1–s3, where the *K*-algorithm's results are also good (NMI within 0.01, CI close to 0.0).

6.3 Effect of initialization

In this section, we study the impact of the initialization method on the *K*-algorithm and *M*-algorithm. We study two different initialization methods: density based initialization described in Algorithm 2; and random partition initialization which assigns a random cluster label to each node. For both these cases, we record the NMI value for (1) initialization, (2) after K-algo convergence, and (3) after repeating M-algo for 100 times. The results are summarized in Fig. 6.

The results show that the initialization method does not affect the results of the *M*-algorithm nearly at all, as the corresponding results with both initializations are virtually

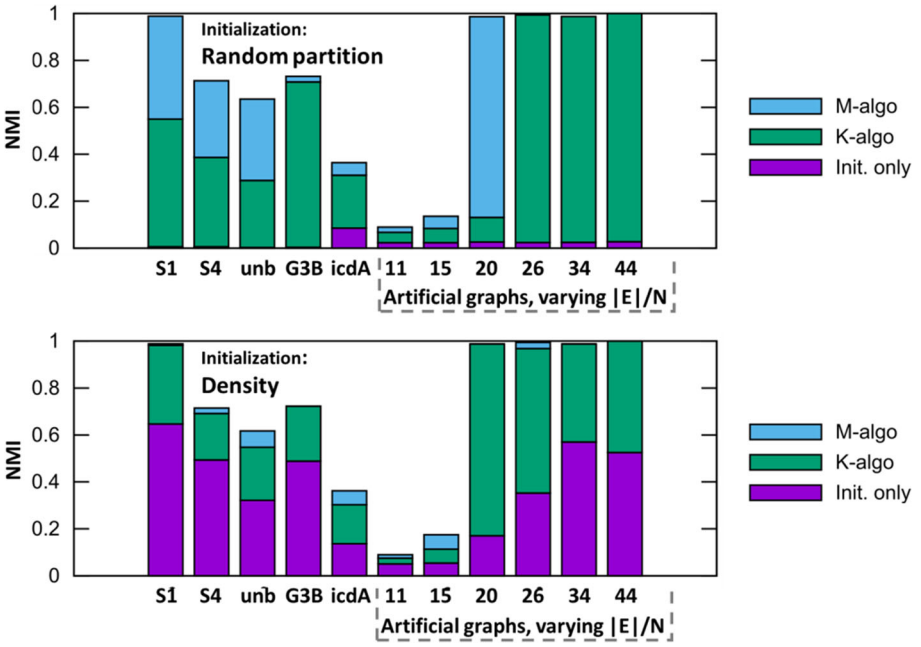


Fig. 6 Impact of initialization on algorithm result. The area of the *M*-algorithm signifies the improvement in NMI over the *K*-algorithm (same with the *K*-algorithm and initialization)

the same. However, in the case of the K-algorithm, the results are much worse for several datasets (S1, S4, unb, 20) in the case of random partition initialization. The average degree on the graph seems to be one affecting factor. The K-algorithm works better even with bad initialization if the number of neighbors in the graph is large. This was the case with dataset icdA ($|E|/N = 53$) and artificial graphs with $|E|/N \geq 26$.

The above observations are in line with the previous results for k-means initialization for numerical data, which indicates that k-means is highly sensitive to initialization [48]. There are two ways to address this issue. The first is to use better initialization [48], and the second is to change to a better algorithm—such as random swap [49]—that does not depend on the initialization. In the case of graph data, we used density-based initialization to implement the first approach and a split-and-merge strategy in the M-algorithm to implement the second.

6.4 K-algorithm iterations

The number of iterations I until convergence was the only part of the time complexity analysis (in Sect. 4.5) that could not be directly calculated by analyzing the algorithm pseudocode. In Table 5, we present experimental results for the number of iterations.

Both the size and difficulty of the dataset caused the K-algorithm to need more iterations until convergence. As observed in Fig. 5, increasing N and μ_t makes the dataset more difficult; the same occurs when decreasing $|E|/N$. In all of these cases, the difficulty also increases the number of iterations. However, when the dataset is too difficult to be solved by any of the algorithms ($|E|/N \leq 15$ and $\mu_t \geq 74$), the number of iterations starts to decrease. The dataset size also has an effect. The highest number of iterations occurred for the dataset of size 1,024,000, where it was 47% higher than the highest value for the varDeg series (where $N = 5000$).

6.5 Disease co-occurrence network

In the previous chapter, the clustering of the disease network was evaluated by how closely it matched the predefined categorization. However, this does not necessarily correspond to the reality, as disease co-occurrences appear beyond category boundaries. In fact, the real goal of clustering is to uncover new information, not to study how well the clustering models existing predefined structures.

We next analyze what new information we found in the clustering results that might have relevance in medical practice. We used the icdB dataset, which has 188 nodes, but removed all symptom codes (starting with R), as these confused the analysis. This left 170 diagnoses. We selected the M-algorithm with IIW cost function as this combination produced accurate clustering for most datasets. Although the size of the dataset is seemingly small, the number of possible connections between nodes is still very large (14,365), which would make manual examination exhausting. Instead, using clustering (with $k = 15$) we can obtain groups that are small enough for realistic human analysis. A proof of concept of such analysis is shown in Fig. 7. A more extensive analysis will be found in a follow-up paper.

Most of the clustering results are as expected. For example, cluster 4 shows the connection between mental health diagnoses and diagnoses related to alcohol and drug use. Furthermore, alcohol use and minor injuries like limb fractures ended up in cluster 9. However, some results are more unexpected. For example, cluster 1 consists of diseases of eye, ear and viral infections wherein the clinical association of different conditions can be recognized but it would likely not constitute a cluster of diseases if defined based only on clinical considerations.

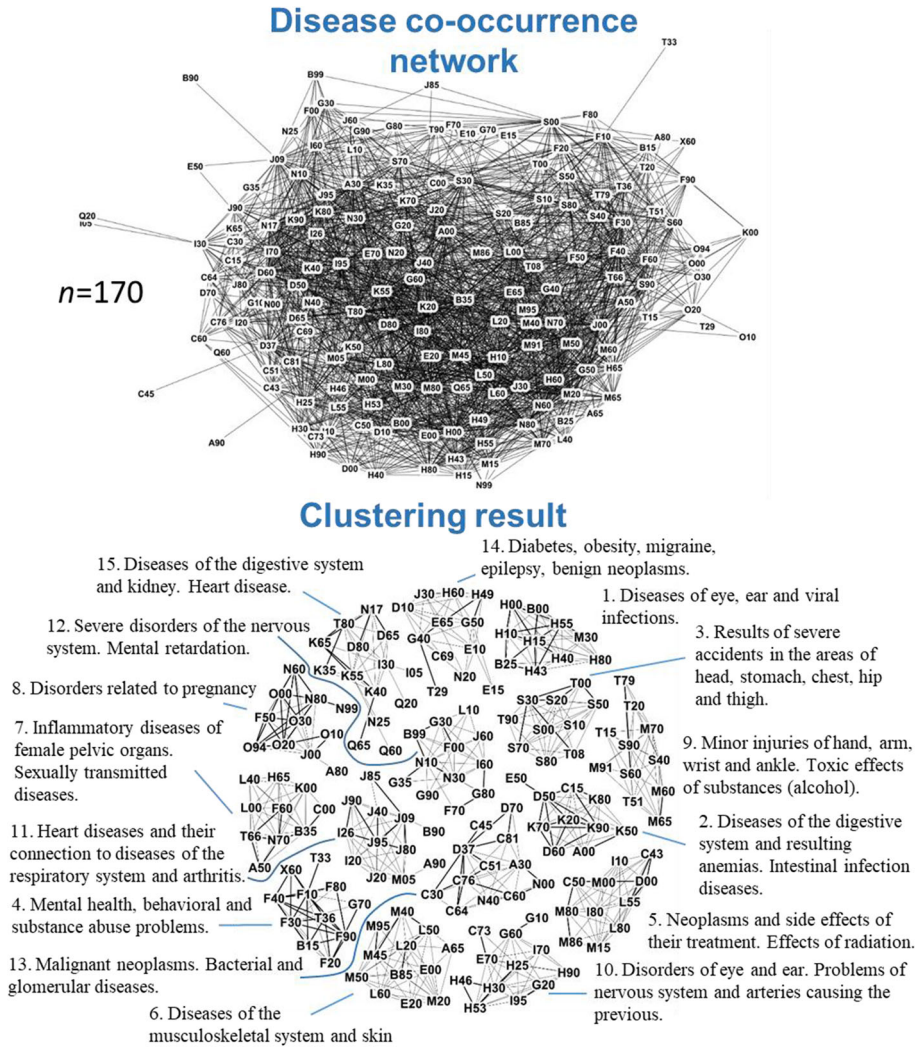


Fig. 7 Clustering result of the icdB dataset. For each cluster, we show a description based on medical expert analysis. Visualization is performed using Gephi. Edges with RR > 1.75 are drawn with a thicker line. The first diagnose code of a range represents the whole range (e.g., T36 represents range T36-T50)

7 Conclusions

We introduced two new cost functions for graph clustering: MIW and IIW. We also studied a third one called conductance. The IIW resulted in the best overall results in the experimental tests. It works especially well in the cases where somewhat balanced clustering is required.

Nevertheless, the choice of a cost function depends on the dataset and the purpose of clustering. All of the studied cost functions are useful in some situations, and the choice of a cost function should ultimately be left up to the analyst. In Sect. 6.1, we provided information to aid in making this decision.

We also introduced two new algorithms K-algorithm and M-algorithm that optimize these cost functions. They are adaptations of the k-means algorithm for graph clustering context. We compared the algorithms against eight previous methods. In the experimental tests, the M-algorithm clearly outperformed existing state-of-the-art. Average clustering error was CI $= 1.7$, whereas best existing (NCut) had CI $= 2.7$.

Acknowledgements This project was funded by the Strategic Research Council (SRC) at the Academy of Finland (grant number 312760-1). We thank Tiina Laatikainen and Katja Wikström for providing the medical expert analysis. More extensive analysis will be provided in a follow-up paper.

Funding Open access funding provided by University of Eastern Finland (UEF) including Kuopio University Hospital. The project was funded by the Strategic Research Council (SRC) at the Academy of Finland (grant number 312760-1).

Data availability The graph datasets documented in Table 2 are published in http://cs.uef.fi/ml/article/graphclu/

Code availability The algorithms' C + + source code is available in: https://github.com/uef-machine-learning/gclu.

Declarations

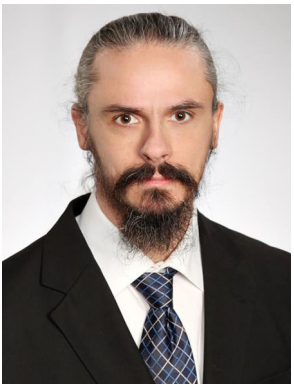
Conflict of interest The authors declared that they have no conflict of interest.

References

1. Newman ME (2006) Modularity and community structure in networks. Proc Natl Acad Sci 103(23):8577–8582
2. Newman ME (2004) Analysis of weighted networks. Phys Rev E 70(5):056131
3. Kernighan BW, Lin S (1970) An efficient heuristic procedure for partitioning graphs. Bell Syst Tech J 49(2):291–307
4. Shi J, Malik J (2000) Normalized cuts and image segmentation. IEEE Trans Pattern Anal Mach Intell 22(8):888–905
5. Divo MJ, Casanova C, Marin JM et al (2015) Chronic obstructive pulmonary disease comorbidities network. Eur Respir J 22:113–118
6. Hromic H, Prangnawarat N, Hulpuş I, Karnstedt M, Hayes C (2015) Graph-based methods for clustering topics of interest in twitter. In: International conference on web engineering, pp 701–704
7. Fortunato S (2010) Community detection in graphs. Phys Rep 486(3–5):75–174
8. Fortunato S, Hric D (2016) Community detection in networks: a user guide. Phys Rep 659:1–44
9. Blondel VD, Guillaume J-L, Lambiotte R, Lefebvre E (2008) Fast unfolding of communities in large networks. J Stat Mech Theory Exp 2008(10):10008
10. Lancichinetti A, Fortunato S (2009) Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities. Phys Rev E 80(1):016118
11. Whang JJ, Gleich DF, Dhillon IS (2016) Overlapping community detection using neighborhood-inflated seed expansion. IEEE Trans Knowl Data Eng 28(5):1272–1284
12. Lu Z, Wen Y, Cao G (2013) Community detection in weighted networks: Algorithms and applications. In: 2013 IEEE international conference on pervasive computing and communications (PerCom), pp 179–184

13. Lancichinetti A, Fortunato S (2009) Community detection algorithms: a comparative analysis. Phys Rev E 80(5):056117
14. Zhang W, Wang X, Zhao D, Tang X (2012) Graph degree linkage: agglomerative clustering on a directed graph. In: European conference on computer vision, pp 428–441
15. LaSalle D, Karypis G (2016) A parallel hill-climbing refinement algorithm for graph partitioning. In: 45th International conference on parallel processing (ICPP), pp 236–241
16. Tabatabaei SS, Coates M, Rabbat M (2012) Ganc: greedy agglomerative normalized cut for graph clustering. Pattern Recogn 45(2):831–843
17. Schäfer T, Mutzel P (2017) Struclus: scalable structural graph set clustering with representative sampling. In: International conference on advanced data mining and applications, pp 343–359
18. Riesen K, Bunke H (2008) Kernel k-means clustering applied to vector space embeddings of graphs. In: IAPR workshop on artificial neural networks in pattern recognition, pp 24–35
19. Ferrer M, Valveny E, Serratosa F, Bardají I, Bunke H (2009) Graph-based k-means clustering: a comparison of the set median versus the generalized median graph. In: International conference on computer analysis of images and patterns, pp 342–350
20. Girvan M, Newman ME (2002) Community structure in social and biological networks. Proc Natl Acad Sci 99(12):7821–7826
21. Kannan R, Vempala S, Vetta A (2004) On clusterings: good, bad and spectral. J ACM 51(3):497–515
22. Pons P, Latapy M (2005) Computing communities in large networks using random walks. In: International symposium on computer and information sciences, pp 284–293
23. Clauset A, Newman ME, Moore C (2004) Finding community structure in very large networks. Phys Rev E 70(6):066111
24. Karypis G, Kumar V (1998) A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Sci Comput 20(1):359–392
25. Newman ME, Girvan M (2004) Finding and evaluating community structure in networks. Phys Rev E 69(2):026113
26. Fortunato S, Barthelemy M (2007) Resolution limit in community detection. Proc Natl Acad Sci 104(1):36–41
27. Hidalgo CA, Blumm N, Barabási A, Christakis NA (2009) A dynamic network approach for the study of human phenotypes. PLoS Comput Biol 5(4):e1000353
28. Okuda M, Satoh S, Sato Y, Kidawara Y (2019) Community detection using restrained random-walk similarity. IEEE Trans Pattern Anal Mach Intell
29. Sinclair A, Jerrum M (1989) Approximate counting, uniform generation and rapidly mixing markov chains. Inf Comput 82(1):93–133
30. Leskovec J, Lang KJ, Mahoney M (2010) Empirical comparison of algorithms for network community detection. In: Proceedings of the 19th international conference on world wide web, pp 631–640
31. Yang J, Leskovec J (2015) Defining and evaluating network communities based on ground-truth. Knowl Inf Syst 42(1):181–213
32. Schaeffer SE (2007) Graph clustering. Comput Sci Rev 1(1):27–64
33. Chakraborty T, Dalmia A, Mukherjee A, Ganguly N (2017) Metrics for community analysis: a survey. ACM Comput Surv 50(4):1–37
34. Javed MA, Younis MS, Latif S, Qadir J, Baig A (2018) Community detection in networks: a multidisciplinary review. J Netw Comput Appl 108:87–111
35. Biedermann S, Henzinger M, Schulz C, Schuster B (2018) Memetic graph clustering. In: Proceedings of the 17th international symposium on experimental algorithms (SEA'18), LIPIcs. Dagstuhl. Technical report arXiv: 1802.07034
36. Peixoto TP (2014) Efficient Monte Carlo and greedy heuristic for the inference of stochastic block models. Phys Rev E 89(1):012804
37. Rozemberczki B, Davies R, Sarkar R, Sutton C (2019) Gemsec: graph embedding with self clustering. In: Proceedings of the 2019 IEEE/ACM international conference on advances in social networks analysis and mining, pp 65–72
38. Buluç A, Meyerhenke H, Safro I, Sanders P, Schulz C (2016) Recent advances in graph partitioning. In: Algorithm engineering, pp 117–158
39. Sanders P, Schulz C (2012) Distributed evolutionary graph partitioning. In: 2012 Proceedings of the fourteenth workshop on algorithm engineering and experiments (ALENEX), pp 16–29
40. Benlic U, Hao JK (2010) An effective multilevel memetic algorithm for balanced graph partitioning. In: 2010 22nd IEEE international conference on tools with artificial intelligence, vol 1, pp 121–128
41. Parés F, Gasulla DG, Vilalta A, Moreno J, Ayguadé E, Labarta J, Cortés U, Suzumura T (2017) Fluid communities: a competitive, scalable and diverse community detection algorithm. In: International conference on complex networks and their applications, pp 229–240

42. Fränti P, Sieranoja S (2018) K-means properties on six clustering benchmark datasets. Appl Intell 48(12):4743–4759
43. Sieranoja S, Fränti P (2018) Fast random pair divisive construction of knn graph using generic distance measures. In: Proceedings of the 2018 international conference on big data and computing, pp 95–98
44. WHO (2016) International statistical classification of diseases and related health problems, 10th revision. World Health Organization, Geneva
45. Danon L, Diaz-Guilera A, Duch J, Arenas A (2005) Comparing community structure identification. J Stat Mech Theory Exp 2005(09):P09008
46. Fränti P, Rezaei M, Zhao Q (2014) Centroid index: cluster level similarity measure. Pattern Recogn 47(9):3034–3045
47. Rossetti G, Milli L, Cazabet R (2019) Cdlib: a python library to extract, compare and evaluate communities from complex networks. Appl Netw Sci 4(1):1–26
48. Fränti P, Sieranoja S (2019) How much can k-means be improved by using better initialization and repeats? Pattern Recogn 93:95–112
49. Fränti P (2018) Efficiency of random swap clustering. J Big Data 5(1):13



Sami Sieranoja received his M.Sc. and Ph.D. degrees from the University of Eastern Finland, 2015 and 2020. Currently he is a postdoctoral researcher at the University of Eastern Finland. His work focuses on analyzing medical data using machine learning methods. His other research interests include neighborhood graphs and data clustering.

Pasi Fränti received his M.Sc. and Ph.D. degrees from the University of Turku, 1991 and 1994 in Science. Since 2000, he has been a professor of Computer Science at the University of Eastern Finland. He has published 99 journals and 175 peer review conference papers. Pasi Fränti is the head of the Machine Learning research group. His current research interests include clustering algorithms, location-based services, machine learning, web and text mining, and optimization of health care services. He has supervised 30 Ph.D. graduates and is currently supervising nine more.