Distributed community detection in complex networks using synthetic coordinates

H. Papadakis [1], C. Panagiotakis [2], P. Fragopoulou [1] ‡

1 Department of Informatics Engineering, Technological Educational Institute of Crete, 71004 Heraklion, Crete, Greece
2 Department of Business Administration, Technological Educational Institute of Crete, 72100 Agios Nikolaos, Crete, Greece

E-mail: adanar@ie.teicrete.gr, cpanag@staff.teicrete.gr, fragopou@ics.forth.gr

Abstract. Various applications like finding web communities, detecting the structure of social networks, or even analyzing a graph's structure to uncover Internet attacks are just some of the applications for which community detection is important. In this paper, we propose an algorithm that finds the entire community structure of a network, based on local interactions between neighboring nodes and on an unsupervised distributed hierarchical clustering algorithm. The novelty of the proposed approach, named SCCD (to stand for Synthetic Coordinate Community Detection), is the fact that the algorithm is based on the use of Vivaldi synthetic network coordinates computed by a distributed algorithm. The current paper not only presents an efficient distributed community finding algorithm, but also demonstrates that synthetic network coordinates could be used to derive efficient solutions to a variety of problems. Experimental results and comparisons with other methods from the literature are presented for a variety of benchmark graphs with known community structure, derived by varying a number of graph parameters and real dataset graphs. The experimental results and comparisons to existing methods with similar computation cost on real and synthetic data sets demonstrate the high performance and robustness of the proposed scheme.

Random graphs, networks; Clustering techniques; Critical phenomena of socio-economic systems; Socio-economic networks

1. Introduction

Networks in various application domains present an internal structure, where nodes form groups of tightly connected components which are more loosely connected to the rest of the network. These components are mostly known as *communities*,

‡ P. Fragopoulou is also with the Foundation for Research and Technology-Hellas, Institute of Computer Science, 70013 Heraklion, Crete, Greece.

clusters, *groups*, or *modules*, the first two terms interchangeably used in the rest of this paper. Uncovering the community structure of a network is a fundamental problem in complex networks which presents many variations. With the advent of Web 2.0 technology, came along the emerging need to analyze network structures like web communities, social network relations, and in general user's collective activities. The newly emerging applications came along with a different set of parameters and demands due to the enormous data size, rendering prohibitive the static manipulation of data and raising the demand for flexible solutions.

Several attempts have been made to provide a formal definition to the generally described "community finding" concept, providing different approaches. Some of them aim at detecting the so-called, *strong communities*, groups of nodes for which each node has more edges to nodes of the same community than to nodes outside the community [1]. Others aim at detecting *weak communities*, which is defined as a subgraph in which the sum of all node degrees within the community is larger than the sum of all node degrees towards the rest of the graph [2]. Variations also appear in the method used to identify communities: Some algorithms follow an iterative approach starting by characterizing either the entire network, or each individual node as community, and splitting [3, 4, 5] or merging [2] communities respectively, producing a hierarchical tree of nested communities, called *dendrogram*. Several researchers aim to find the entire hierarchical community dendrogram [3, 4] while others wish to identify only the optimal community partition [1]. More recently used approaches aim to identify the community surrounding one or more seed nodes [6]. Some researchers aim at discovering distinct (non-overlapping) communities, while others allow for overlaps between communities [7].

In this paper we propose SCCD (ta stand for Synthetic Coordinate Community Detection), an algorithm that identifies the entire community structure of a network based on interactions between neighboring nodes. In the core of our proposal lies the spring metaphor which inspired the Vivaldi synthetic network coordinate algorithm [8]. The algorithm comprises two main phases. First, each node selects a "local" set containing mostly nodes of the same community, and a "foreign" set containing mostly nodes of different communities. As the algorithm evolves, and the springs connecting local and foreign nodes are tightened and relaxed, nodes of the same community pull each other close together, while nodes of different communities push each other further away. Given that the initial selection of local and foreign sets is "mostly" correct, nodes of the same community eventually gravitate to the same area in space, while nodes of different communities are placed further away. In other words nodes belonging to the same community will form natural clusters in space. In the second phase of the algorithm, a distributed hierarchical clustering algorithm has been proposed to automatically

identify the natural communities formed in space. Extensive experiments on several benchmark graphs with known community structure indicate that our algorithm is highly accurate in identifying community membership of nodes.

A first version of our algorithm was presented in [9]. The algorithm presented in this paper is a heavily modified and improved version. A new simpler and more accurate algorithm termination mechanism has been introduced. More importantly, the algorithm can now dynamically make an effort to correct the "foreign" and "local" sets as we shall see later on, increasing the obtained accuracy. Finally, we added an optional third phase in the algorithm, which allows for a user defined value on the number of communities requested. As far as the experimental evaluation is concerned, we performed experiments on real world graphs, in addition to the benchmark graphs. In particular, we performed experiments using new benchmark graphs of diverse community sizes and node degrees. Finally, we compared our algorithm based on a new accuracy metric, for a total of two metrics to evaluate performance on benchmark graphs.

The remaining of the paper is organized as follows: Section 2 presents an overview of some of the methods developed over the years for community detection in networks. Our distributed community detection algorithm is presented and analyzed in Section 3. Section 4 describes the experimental framework and comparison results with other known algorithms on a number of benchmark graphs. Finally, we conclude in Section 6 with some directions for future research.

2. Related Work

Below we review some of the known methods for community detection and give insight on the approach they follow. For the interested reader, two comprehensive and relatively recent surveys covering the latest developments in the field can be found in [10, 11]. While the first algorithms for the problem used the agglomerative approach trying to derive an optimal community partition by merging or splitting other communities, recent efforts concentrate on the derivation of algorithms based exclusively on local interaction between nodes. A community surrounding a seed node is identified by progressively adding nodes and expanding a small community.

One of the most known community finding algorithms was developed by Girvan an Newman [3, 4]. This algorithm iteratively removes edges participating in many shortest paths between nodes (indicating bridges), connecting nodes in different communities. By gradually removing edges, the graph is split and its hierarchical community structure is revealed. The algorithm is computationally intensive because following the removal of an edge, the shortest paths between all pairs of nodes have to be recalculated. However, it reveals not only individual communities, but the entire hierarchical community dendrogram of the graph. In

[5], a centralized method for decomposing a social network into an optimal number of hierarchical subgroups has been proposed. With a perfect hierarchical subgroup defined as one in which every member is automorphically equivalent to each other, the method uses the REGGE algorithm to measure the similarities among nodes and applies the k-means method to group the nodes that have congruent profiles of dissimilarities with other nodes into various numbers of hierarchical subgroups. The best number of clusters is determined by minimizing the intra-cluster variance of dissimilarity subject to the constraint that the improvement in going to more clusters is better than a network whose n nodes are maximally dispersed in the n-dimensional space would achieve.

In a different approach, the algorithm presented in [2], named *CiBC*, starts by assuming that each node is a different community, and merges closely connected communities. This algorithm is less intensive computationally since it starts by manipulating individual nodes rather than the entire graph.

The authors of [6] introduce a local methodology for community detection, named *Bridge Bounding*. The algorithm can identify individual communities starting at seed nodes. It initiates community detection from a seed node and progressively expands a community trying to identify *bridges*. An edge is characterized as a bridge by computing a function related to the *edge clustering coefficient*. The edge clustering coefficient is calculated for each edge, looking at the edge's neighborhood, and edges are characterized as bridges depending on wether their clustering coefficient exceeds a threshold. The method is local, has low complexity and allows the flexibility to detect individual communities, albeit less accurately. Additionally, the entire community structure of a network can be uncovered starting the algorithms at various unassigned seed nodes, till all nodes have been assigned to a community.

In [12], a local partitioning algorithm using a variation of PageRank with a specified starting distribution, which allows to find such a cut in time proportional to its size. A PageRank vector is a weighted sum of the probability distributions obtained by taking a sequence of random walk steps starting from a specified initial distribution. The cut can be found by performing a sweep over the PageRank vector, which involves examining the vertices of the graph in an order determined by the PageRank vector, and computing the conductance of each set produced by this order. In [13], three distributed community detection approaches based on Simple, K-Clique, and Modularity metrics, that can approximate their corresponding centralized methods up to 90% accuracy.

Other community finding methods of interest involve [1] in which the problem is regarded as a maximum flow problem and edges of maximum flow are identified to separate communities from the rest of the graph. In clique percolation [14, 15] a complete subgraph of k nodes (k-*clique*) is rolled over the network through other

cliques with $k-1$ common nodes. This way a set of nodes can be reached, which is identified as a community. A method based on voltage drops across networks and the physics kirchhoff equations is presented in [16]. A mathematical Markov stochastic flow formulation method known as *MCL* is presented [17], and a local community finding method in [18], just to mention a few.

We will now describe the four state-of-the-art algorithms that we compare our approach with, in the experimental evaluation section. An exceptionally interesting method for community detection was developed by Lancichinetti et al. and appears in [7]. Although most previous approaches identify distinct (non-overlapping) communities, this algorithm is developed based on the observation that network communities may have overlaps, and thus, algorithms should allow for the identification of overlapping communities. Based on this principle, a local algorithm is devised developing a community from a starting node and expanding around it based on a *fitness* measure. This fitness function depends on the number of inter- and intra-community edges and a tunable parameter α. Starting at a node, at each iteration, the community is either expanded by a neighboring node that increases the community fitness, or shrinks by omitting a previously included node, if this action results in higher fitness for the resulting community. The algorithm stops when the insertion of any neighboring node would lower the fitness of the community. This algorithm is local, and able to identify individual communities. The entire overlapping and hierarchical structure of complex networks can be found by initiating the algorithm at various unassigned nodes.

Another efficient algorithm is the one described by Chen et al. in [19]. The algorithm follows a top down approach where the process starts with the entire graph and sequentially removes inter-community links (bridges) until either the graph is partitioned or its density exceeds a certain desired threshold. If a graph is partitioned, the process is continued recursively on its two parts. In each step, the algorithm removes the link between two nodes with the smallest number of common neighbors. The density of a graph is defined as the number of edges in the graph divided by the number of edges of a complete graph with the same number of nodes.

The algorithm described by Blondel et al. in [20] follows a bottom-up approach. Each node in the graph comprises a singleton community. Two communities are merged into one if the resulting community has larger modularity value [21] than both the initial ones. This is a rapid and accurate algorithm which detects all communities in the graph. In suffers however, in the sense, from the fact that during its execution, it constantly requires the knowledge of some global information of the graph, namely the number of its edges (which changes during the execution since the algorithm modifies the graph), limiting, to a certain extend, its distributed nature.

Finally, we compare our algorithm with the one described in [22], called Infomap. This algorithm transforms the problem of community detection into efficiently compressing the structure of the graph, so that one can recover almost the entire structure from the compressed form. This is achieved by minimizing a function that expresses the tradeoff between compression factor and loss of information (difference between the original graph and the reconstructed graph).

Most of the approaches found it the literature are centralized, heuristic without a global optimality criterion. On the contrary, in this paper, we have proposed a fully distributed method that solves the community detection problem. In addition, another strong point of the proposed method is that according to the experimental results and comparisons to existing methods on real and synthetic data sets, the proposed method clearly outperforms the other methods.

3. SCCD Community Finding

The proposed local community finding algorithm comprises the following steps:

- The position estimation algorithm, which is a distributed algorithm inspired by Vivaldi [8].
- The community detection algorithm using hierarchical clustering.

3.1. Vivaldi synthetic coordinates

Network coordinate systems predict latencies between network nodes, without the need of explicit measurements using probe queries. These algorithms assign synthetic coordinates to nodes, so that the distance between two nodes' coordinates provides an accurate latency prediction between them. This technique provides to applications the ability to predict round trip time with less measurement overhead than probing.

Vivaldi is a fully decentralized, light-weight, adaptive network coordinate algorithm that was initially developed to predict Internet latencies with low error. Vivaldi uses the Euclidian coordinate system (in *n-dimensional* space, where n is a parameter) and the associated distance function. Conceptually, Vivaldi simulates a network of physical springs, placing imaginary springs between pairs of network nodes.

Let $G = (V, E)$ denote the given graph comprising a set V of nodes together with a set E of edges. Each node $x \in V$ participating in Vivaldi maintains its own coordinates $p(x) \in \Re^n$ (the position of node x that is a point in the *n-dimensional* space). The Vivaldi method consists of the following steps:

- Initially, all node coordinates are set at the origin.

- Periodically, each node communicates with another node (randomly selected among a small set nodes of nodes known to it). Each time a node communicates with another node, it measures its latency and learns that node's coordinates. Subsequently, the node allows itself to be moved a little by the corresponding imaginary spring connecting them (i.e the positions change a little so as the Euclidian distance of the nodes to better match the latency distance).
- When Vivaldi converges, any two nodes' Euclidian distance will match their latency distance, even though those nodes may never had any communication.

Unlike other centralized network coordinate approaches, in Vivaldi each node only maintains knowledge for a handful of other nodes, making it completely distributed. Each node computes and continuously adjusts its coordinates based on measured latencies to a handful of other nodes. Finally, Vivaldi does not require any fixed infrastructure as for example landmark nodes.

3.2. The position estimation algorithm

As we mentioned, in the core of our proposal lies the spring metaphor which inspired the Vivaldi algorithm. Vivaldi uses the spring relaxation metaphor to position the nodes in a virtual space (the n-dimensional Euclidean space), so as the Euclidean distance of any two node positions approximates the actual distance between those nodes. In the original application of Vivaldi, the actual distances were the latencies between Internet hosts. Our algorithm is based on the idea that by providing our own, appropriate, definition of distance between nodes, we can use Vivaldi to position the nodes in a way as to reflect community membership, i.e. nodes in the same community will be placed closer in space than nodes of different communities. In other words nodes belonging to the same community will form natural clusters in space.

Let $C(x)$, $C(y)$ denote the communities' sets of two nodes $x, y \in V$, respectively, of a given graph. Since two nodes either belong to the same community ($C(x) = C(y)$) or not, we define the initial node distance between two nodes x and y as $d(x, y)$:

$$d(x,y) = \begin{cases} 0, & C(x) = C(y) \\ 1, & C(x) \neq C(y) \end{cases} \tag{1}$$

When $C(x) \neq C(y)$, we have set $d(x, y) = 1$ in order to normalize the distances in range between 0 and 1. Given this definition of distance, we can employ the core part of the Vivaldi algorithm to position the nodes appropriately in the n-dimensional Euclidian space ($\Re^n$). As one can expect from those dual distances, Vivaldi will position nodes in the same community close-by in space, while place

nodes of different communities away from each other. This is the reason for the dual nature of the distance function, otherwise all nodes, regardless of community membership, would gravitate to the same point in space.

In addition, Vivaldi requires a selection of nodes to probe. Each node calculates a "local" set containing nodes of the same community, and a "foreign" set containing nodes of different communities. The size of the local set as well as the size of the foreign set of a node equals the degree of the node. The perfect construction of these sets depends on the apriori knowledge of node community membership, which is the actual problem we are trying to solve. However, even though we do not know the community each node belongs to, there are two facts we can exploit to make Vivaldi work without this knowledge:

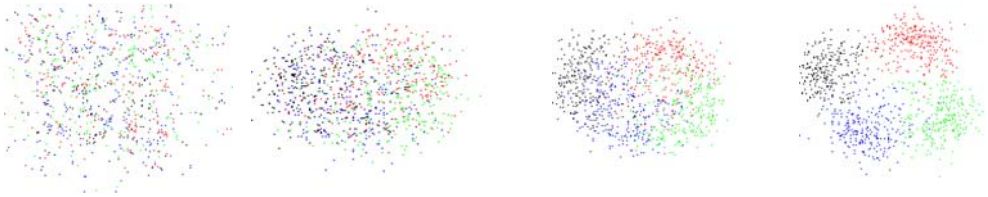
- The first is the fact that, by definition, the number of *intra-community links* of a node exceeds the number of its *inter-community links*. This means that, if we assume that all of a node's neighbors belong to the same community, this assumption will be, *mostly*, correct, which in turn means that even though some times the node may move to the wrong direction, most of the time it will move to the right direction and thus, will eventually acquire an appropriate position in space. Thus, we let the local set $L(x)$ of a node $x \in V$, be its "neighbor set".
$$L(x) = \{y \in V : x \sim y\} \tag{2}$$
The distance from node x to nodes in $L(x)$ is set to 1 according to Equation (1).
- The second fact we exploit concerns the *foreign links*. Since we consider all a node's links as local links, we need to find some nodes which most likely do not belong to the same community as that node, and therefor will be considered as foreign nodes. This can simply be done by randomly selecting a small number of nodes from the entire graph. Assuming that the number of communities in the graph is at least three, the majority of the nodes in this set will belong to a different community than the node itself. These nodes will comprise the "foreign set" $F(x)$ of node $x \in V$:
$$F(x) \subset \{y \in V : x \not\sim y\}. \tag{3}$$
The distance from node x to the nodes in $F(x)$ is set to 1 according to Equation (1).

The pseudo-code of the position estimation algorithm is given in Algorithm 1 and it is described hereafter. The function $getRandomNumber(0, 1)$ returns a random number in $[0, 1]$. Initially, each node is placed at a random position in $\Re^n$. Iteratively, each node $x \in V$ randomly selects a node from either its $L(x)$ or its $F(x)$ set (see line 8,11 of Algorithm 1). It then uses Vivaldi to update its current

input : $L(x), F(x),\ \forall x \in V$.
output: $p(x),\ \forall x \in V$.

foreach $x \in V$ **do**
  $p(x) =$ random position in $\Re^n$
  $ite(x) = 0$
end
repeat
  foreach $x \in V$ **do**
    if $getRandomNumber(0,1) < 0.5$ **then**
      Let v be a random vertex from $L(x)$
      $p(x) = Vivaldi(p(x), p(v), 0)$
    else
      Let v be a random vertex from $F(x)$
      $p(x) = Vivaldi(p(x), p(v), 1)$
    end
    $ite(x) = ite(x) + 1$
    if $ite(x) > 5 \cdot (|L(x)| + |F(x)|)$ **then**
      $ite(x) = 0$
      $maxD = \max_{y \in L(x)}(||p(x) - p(y)||)$
      $minD = \min_{y \in L(x)}(||p(x) - p(y)||)$
      $T_2 = minD + max(\frac{maxD - minD}{3}, \frac{1}{3})$
      $\mu = \frac{minD + maxD}{2}$
      $\sigma_n = \frac{\sqrt{E_{y \in L(x)}[(||p(x) - p(y)|| - \mu)^2]}}{T_2}$
      if $\sigma_n > 0.6$ **then**
        foreach $y \in L(x)$ **do**
          if $||p(x) - p(y)|| > T_2$ **then**
            $L(x) = L(x) - \{y\}$
          end
        end
        foreach $y \in F(x)$ **do**
          if $||p(x) - p(y)|| \leq T_2$ **then**
            $F(x) = F(x) - \{y\}$
          end
        end
      end
    end
  end
until $\forall x \in V\ p(x)$ *is stable*

Algorithm 1: The position estimation algorithm.



(a) (b) (c) (d)

Figure 1. Snapshots of the execution of the first phase of our algorithm for a graph with known community structure. **(a)** initialization, **(b)** after 150 iterations, **(c)** after 400 iterations.

position using the appropriate distance (i.e. 0 or 1) to the selected node (see lines 9, 12 of Algorithm 1).

Each node continues this process until it deems its position to have stabilized as much as possible (see line 36 of Algorithm 1). This is done by calculating the sum of the distances between each two consecutive positions of the node between 40 iterations (corresponding to 40 position updates, experiments showed a larger number only slows down the algorithm without adding to efficiency). Each node also calculates the distance, in a straight line, between the two positions before and after the 40 updates. Should this value be less than one tenth the actual traveled distance (the aforementioned sum) for 40 consecutive times, the node declares itself to have stabilized. This means that even though the node moved 40 times, its movement was around the same position in space instead of moving constantly in the same direction. Each node continues, however, to execute the algorithm until at least 90% of its "foreign" and "local" sets have also stabilized.

As we have already mentioned, both the "local" and the "foreign" sets of a node will initially contain erroneous nodes. One of the most important augments of our algorithm is its ability to dynamically correct those sets (see lines 23-32 of Algorithm 1). This is based on the fact that, as the algorithm progresses, those nodes in the "local" set of a node x which do not actually belong in the same community as x, will be located a long distance away from X. As a result, even though the distances between a node and the nodes in its "local" set will initially be uniformly distributed, after a while we will notice the nodes of the local set to be divided into two groups of smaller and larger distance values. This is a good indication that we can separate the wheat from the chaff, which is implemented in the following fashion:

Let $ite(x)$ denotes the number of updates of node x (see lines 3, 14 of Algorithm 1). After several number of updates ($5 \cdot (|L(x)| + |F(x)|$, where $|L(x)|$, $|F(x)|$ denote the number of elements of the sets $L(x)$, $F(x)$), the node x as will have been updated from most of the nodes of $L(x)$, $F(x)$. This means that x is able to

check its "confidence level" in identifying the erroneous nodes of its local-foreign sets (see line 15 of Algorithm 1).

Let $minD$ and $maxD$ denote the minimum and maximum Euclidean distance values from x to all the nodes in its "local" set, respectively (see lines 17-18 of Algorithm 1). Let μ be the average of $minD$ and $maxD$ (see line 19 of Algorithm 1). Next, we calculate the normalized standard deviation σ_n of its distances to the nodes in its "local" set, normalizing σ_n based on its distance of the closest and furthest away node in the local set (the same formula used in equation 5):

$$\sigma_n = \frac{\sqrt{E_{y \in L(x)}[(||p(x) - p(y)|| - \mu)^2]}}{T_2} \tag{4}$$

In addition, σ_n is computed used the value μ instead of the mean value of distances. This is because our "confidence" should be high when most neighbor distances are located in the extreme ends (close to minD and maxD). Should σ_n exceed a certain threshold $T_1 = 0.6$, the node iterates between the nodes in both its "local" and "foreign" sets, removing any inappropriate nodes (see line 22 of Algorithm 1). In order to identify a node as "local" or "foreign" based on its distance, another threshold is required. This threshold T_2 is calculated as follows:

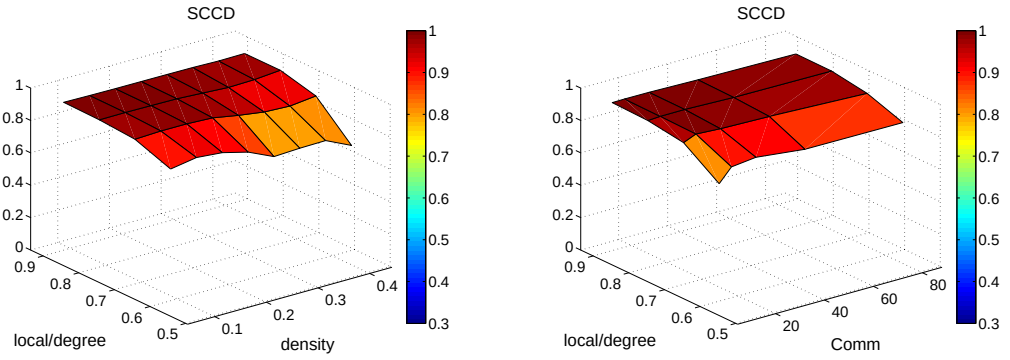
$$T_2 = minD + max(\frac{maxD - minD}{3}, \frac{1}{3}), \tag{5}$$

The idea behind this formula is that the distance of a "foreign" node is both related to the distance values of the rest of the nodes but also has a fixed minimum value. Overall, this dynamic set correction gave us on the benchmark graph tests an increase of about 6% on average.

Fig. 1 shows a small time-line of the execution of our algorithm on a graph with 1024 nodes, degree 20, and a known community structure comprising four communities. We have used different colors for the nodes of each different community. Initially, nodes were randomly placed in $\Re^2$. As we can see, in the beginning all colors are dispersed on the entire space. As the algorithm progresses, we see that nodes of the same color, belonging to the same community, gradually gravitate to the same area, forming distinct clusters in space.

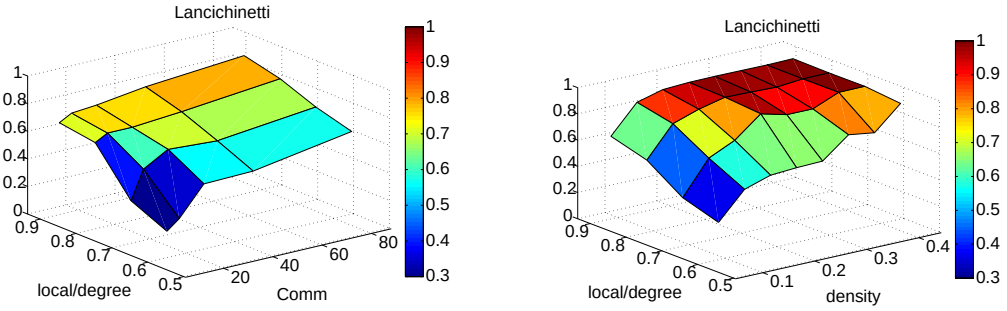
3.3. Hierarchical clustering

After each node has converged to a point in space, we use a hierarchical clustering algorithm to perform the actual grouping of nodes into clusters. The main advantages of the hierarchical clustering algorithms is that the number of clusters need not be specified a priory, and problems due to initialization and local minima do not arise [23]. The pseudo-code of the proposed hierarchical clustering method is given in Algorithm 2 and it is described hereafter. Let $c(x)$ denote the cluster



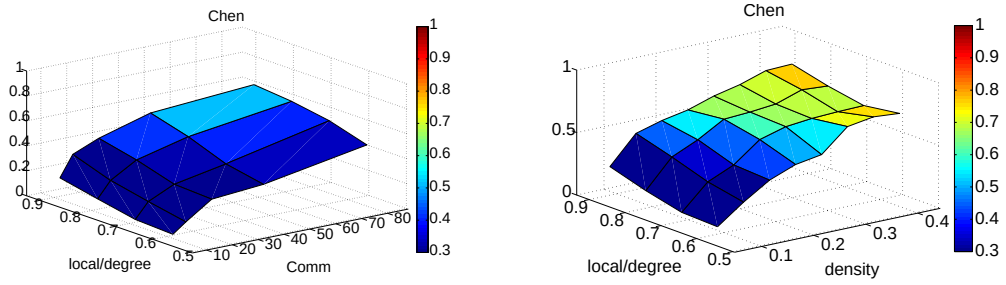
(a) (b)

Figure 2. (a),(b) The mean value of accuracy under **(a)** different ratios of total degree to local links ($local/degree$) and number of communities ($Comm$) and **(b)** total degree to local links ($local/degree$) and densities for our algorithm.



(a) (b)

Figure 3. (a),(b) The mean value of accuracy under **(a)** different ratios of total degree to local links ($local/degree$) and number of communities ($Comm$) and **(b)** total degree to local links ($local/degree$) and densities for the Lancichinetti algorithm.



(a) (b)

Figure 4. (a),(b) The mean value of accuracy under **(a)** different ratios of total degree to local links ($local/degree$) and number of communities ($Comm$) and **(b)** total degree to local links ($local/degree$) and densities for the Chen et al. algorithm.

input : $p(x), \forall x \in V$.
output: $c(x), \forall x \in V$.

1 $i = 1$
2 **foreach** $x \in V$ **do**
3 $\quad c(x) = i$
4 $\quad i = i + 1$
5 **end**
6 **repeat**
7 $\quad S = 0$
8 $\quad$ **foreach** $x \in V$ **do**
9 $\quad\quad y = getClosest(x)$
10 $\quad\quad$ **if** $getClosest(y) = x \wedge |p(x) - p(y)|_2 < \frac{1}{2}$ **then**
11 $\quad\quad\quad S = S + 1$
12 $\quad\quad\quad p(x) = \frac{|x| \cdot p(x) + |y| \cdot p(y)}{|x| + |y|}$
13 $\quad\quad\quad c(y) = c(x)$
14 $\quad\quad\quad x = x \cup y$
15 $\quad\quad\quad V = V - y$
16 $\quad\quad$ **end**
17 $\quad$ **end**
18 **until** $S = 0$

Algorithm 2: The Hierarchical clustering algorithm.

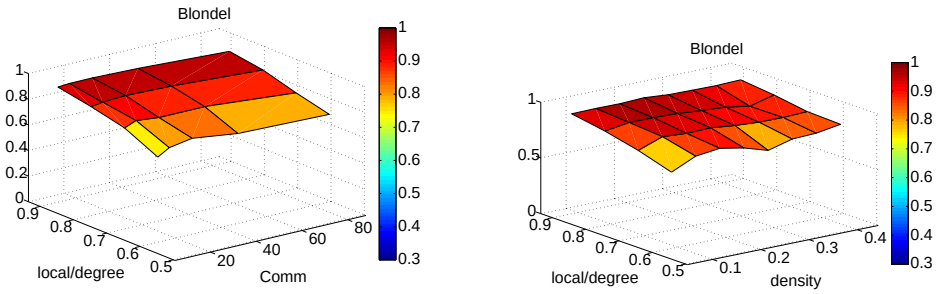
id of node x. The function $getClosest(x)$ returns the closest neighboring cluster of x.

Firstly, each node is considered as a (singleton) cluster (see lines 2-5 of Algorithm 2). In addition, to make the procedure completely distributed, each node-cluster is aware of the location only of its neighboring node-clusters. Then, the following loop is executed repeatedly, until no appropriate pair of clusters can be located: Given a pair of neighboring clusters x and y, if both of them are each other's closest neighbor and the distance between the two clusters is less than a threshold $T_3 = \frac{1}{2}$, then those two clusters are merged in the following fashion (see line 10 of Algorithm 2):

- The merged cluster contains the union of the neighbors of A and B (see lines 14-15 of Algorithm 2).
- Its position is calculated as the weighted based on the population of nodes ($|x|$ and $|y|$) in each cluster x and y average of the positions of x and y, $p(x)$ and $p(y)$ (see line 12 of Algorithm 2):

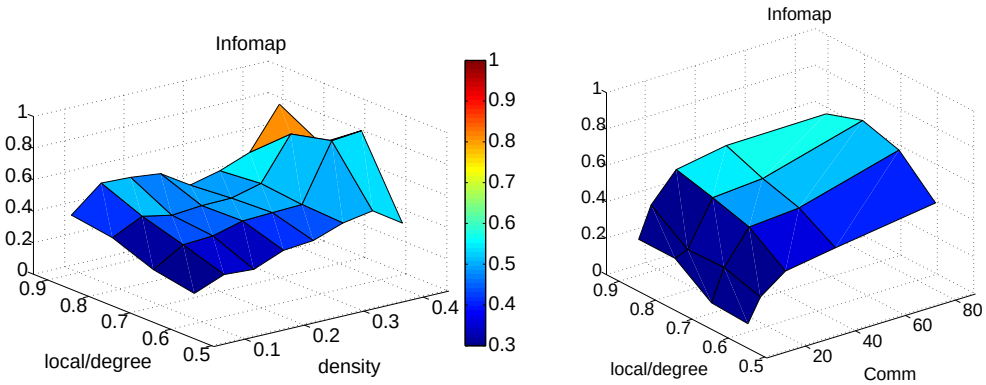
$$p(x) = \frac{|x| \cdot p(x) + |y| \cdot p(y)}{|x| + |y|} \tag{6}$$

Since, $d(x, y)$ is zero and one when x, y belong on the same and different community, respectively, then it holds that the selection of $T_3 = 0.5$ is the most



(a) (b)

Figure 5. (a),(b) The mean value of accuracy under **(a)** different ratios of total degree to local links (*local/degree*) and number of communities (*Comm*) and **(b)** total degree to local links (*local/degree*) and densities for the Blondel algorithm.



(a) (b)

Figure 6. (a),(b) The mean value of accuracy under **(a)** different ratios of total degree to local links (*local/degree*) and number of communities (*Comm*) and **(b)** total degree to local links (*local/degree*) and densities for the Infomap algorithm.

physical selection. Generally, $T_3 \in (0, 1)$. When T_3 gets low values (e.g. close to zero), then the proposed method results high number of small communities (oversegmentation). Otherwise, if T_3 gets high values (e.g. close to one), then we get low number of large communities.

3.4. Communication Load and Computational Complexity

SSCD can be implemented as a fully distributed system, since both of the two main parts of the proposed method are distributed.

- The first part concerns the position estimation algorithm that uses the Vivaldi synthetic network coordinates [8] (see Sections 3.1 and 3.2). This part can be

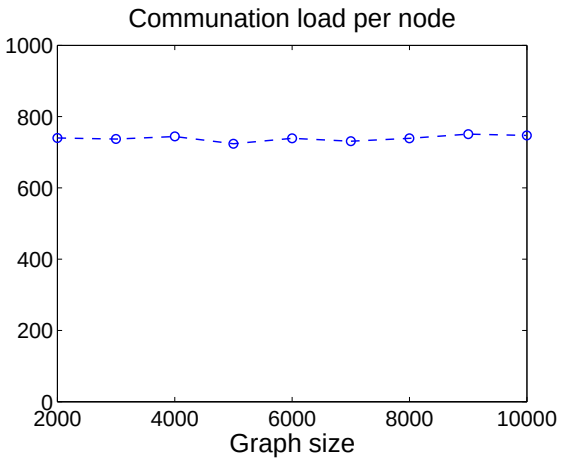


Figure 7. Average number of Vivaldi update messages per node, per graph size.

computed by a distributed algorithm. It holds that each node only maintains knowledge for a handful of other nodes, computing and continuously adjusting its coordinates based on the coordinates of the nodes that belong on its local and foreign sets, making it completely distributed.

- The second part concerns the hierarchical clustering that can be computed by a distributed algorithm. In order to make this procedure distributed, each node-cluster is aware of the location only of its neighboring node-clusters (see Section 3.3).

Hereafter, we provide an analysis of the communication load and computational complexity.

- Concerning the position estimation algorithm, it holds that during the update process each node communicates with a node of its local or its foreign set. So, the communication load depends on the time of convergence. In order to measure the dependance of the number of messages to the size of the graphs, we performed experiments on graphs with identical parameters but varying sizes. Namely, we used 9 graphs with 2000, 3000, ..., 10000 nodes, with 500 nodes per community, 10 degree per node and a 0.75 ratio of local to foreign links per node. Fig. 3.4 shows that the average number of update messages *per node* required by Vivaldi in order to stabilize is approximately the same, regardless of the size of the graph. This means that the convergence time (computational complexity) *per node* is also independent of the size of the graph.
- Concerning the hierarchical clustering algorithm, it holds that during the merging process each node communicates with the nodes of its neighborhood in order to find the closest. In a distributed implementation, the initial

communication load is $O(degree)$ for the first merging. Next, each new cluster sends its updated position to its neighborhood that needs $O(degree)$ messages. In the second level of merging, each new cluster sends its updated position to its neighborhood that have size $O(2^1 \cdot degree)$ (worst case). In the last level of merging ($l = log(\frac{N}{Comm})$), when the hierarchical clustering tree is balanced, each new cluster sends its updated position to its neighborhood that have size $O(2^l \cdot degree) = O(\frac{N}{Comm} \cdot degree)$ (worst case). The total communication load is $O(N \cdot degree + \frac{N}{2} \cdot 2^1 \cdot degree + ... + \frac{N}{2^l} \cdot 2^l \cdot degree) = O(l \cdot N \cdot degree) = O(log(\frac{N}{Comm}) \cdot N \cdot degree)$. The computation cost per node is $O(l \cdot degree) = O(log(\frac{N}{Comm}) \cdot degree)$.

The communication load as well as the computational complexity of the proposed distributed framework make possible the execution of SCCD on graphs of very large scale (e.g. 50 millions of nodes with a billion of links).

The proposed method has been implemented in Java and it is not optimized for speed. Moreover, Vivaldi and hierarchal clustering algorithms are not implemented as a distributed system. Hereafter, we have reported the execution times of our implementation for the 208 benchmark graphs (see Section 4.1) that will be significantly reduced in a fully distributed implementation. For our experiments, we have used an Intel Xeon CPU of 2.67 GHz with 16 GB of memory. The average execution times for the benchmark graphs of 1000, 5000 and 10000 nodes are 28, 178 and 403 secs, respectively.

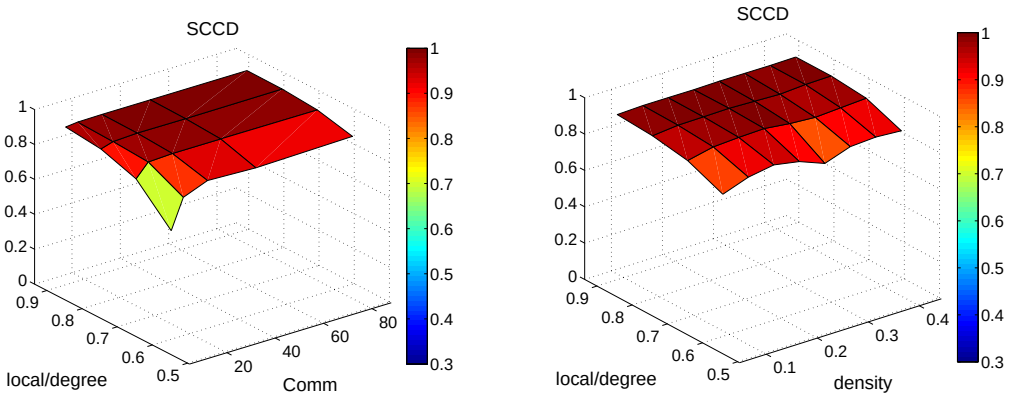
4. Benchmark Graph Experiments

4.1. Benchmark graphs

We have created a variety of benchmark graphs with known community structure to test the accuracy of our algorithm. Benchmark graphs are essential in the testing of a community detection algorithm since there is an apriori knowledge of the structure of the graph and thus one is able to accurately ascertain the accuracy of the algorithm. Since there is no consensus on the definition of a community, using a real-world graph makes it more difficult to assess the accuracy of a community partition.

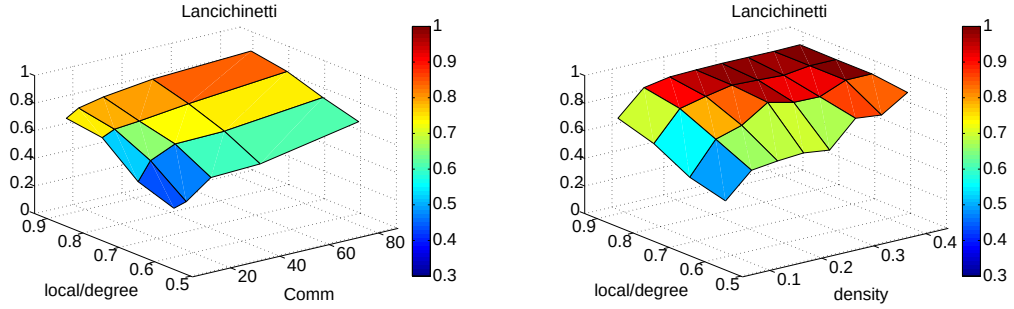
Our benchmark graphs were generated randomly given the following set of parameters:

- The number of nodes N of the graph.
- The number of communities $Comm$ of the graph.
- The ratio of local links to node degree $local/degree$.
- The (average) degree of nodes $degree$.



(a) (b)

Figure 8. (a),(b) The mean value of normalized mutual information under **(a)** different ratios of total degree to local links ($local/degree$) and number of communities ($Comm$) and **(b)** total degree to local links ($local/degree$) and densities for our algorithm.



(a) (b)

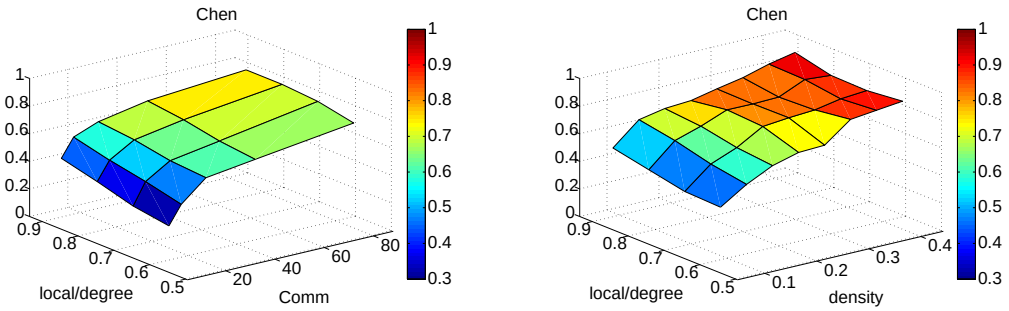
Figure 9. (a),(b) The mean value of normalized mutual information under **(a)** different ratios of total degree to local links ($local/degree$) and number of communities ($Comm$) and **(b)** total degree to local links ($local/degree$) and densities for the Lancichinetti algorithm.

Notice that even though the number of the nodes, the number of the communities and the degree of the nodes are parameters of the construction of the graph, the degree of each node as well as the number of nodes in a single community varies based on a pareto distribution. This enables us to create graphs of community sizes and individual degrees varying up to an order of magnitude.

The parameters used by the algorithm and their corresponding values are shown in Table 1. In total, we created a number of 208 benchmark graphs.

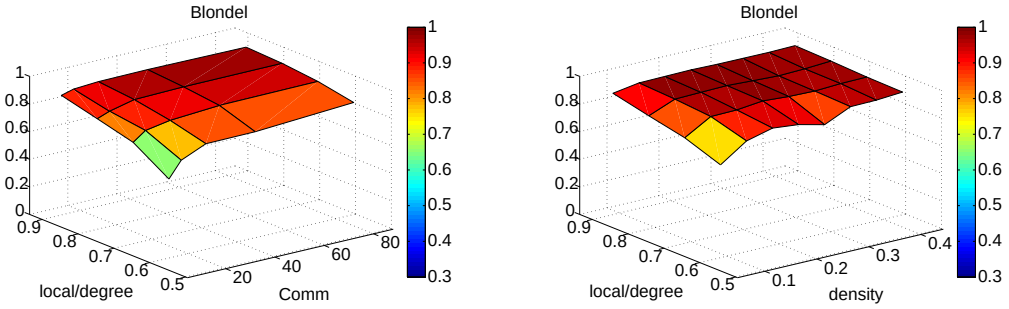
A demonstration of the propose method is given in §, that contains the

§ `http://www.csd.uoc.gr/~cpanag/DEMOS/commDetection.htm`



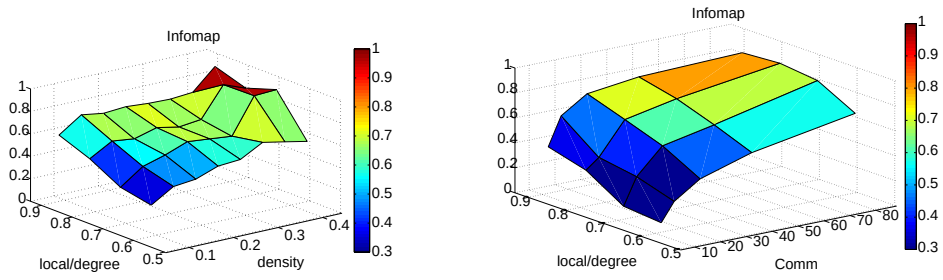
(a) (b)

Figure 10. (a),(b) The mean value of normalized mutual information under **(a)** different ratios of total degree to local links ($local/degree$) and number of communities ($Comm$) and **(b)** total degree to local links ($local/degree$) and densities for the Chen et al. algorithm.



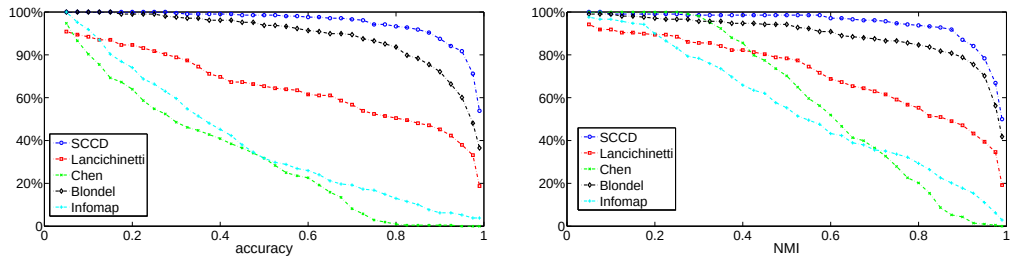
(a) (b)

Figure 11. (a),(b) The mean value of normalized mutual information under **(a)** different ratios of total degree to local links ($local/degree$) and number of communities ($Comm$) and **(b)** total degree to local links ($local/degree$) and densities for the Blondel algorithm.



(a) (b)

Figure 12. (a),(b) The mean value of normalized mutual information under **(a)** different ratios of total degree to local links ($local/degree$) and number of communities ($Comm$) and **(b)** total degree to local links ($local/degree$) and densities for the Infomap algorithm.



(a) (b)

Figure 13. (a),(b) Percentage of "successful" experiments, given an accuracy success threshold (x axis) for **(a)** the accuracy metric and **(b)** the NMI metric.

N	1000, 5000, 10000
$Comm$	5, 10, 20, 40, 80
$local/degree$	0.55, 0.65, 0.75, 0.85
$degree$	10, 20, 30, 40

Table 1. The different values for the used parameters.

benchmark graphs, related articles and an executable of the proposed method.

4.2. Comparison metrics

We compared the five algorithms using two accuracy-related metrics found mostly used in the literature. The first is a simple accuracy metric paired however with the Hungarian algorithm. The simple accuracy is defined as follows: Let $S_i, i \in \{1, ..., Comm\}$ be the estimated and $\hat{S}_i, i \in \{1, ..., Comm\}$ the corresponding actual communities. The accuracy (acc) is given by the average (of all communities) of the number of nodes that belong to the intersection of $S_i \cap \hat{S}_i$ divided by the number of nodes that belongs to the union $S_i \cup \hat{S}_i$.

$$acc = \frac{1}{Comm} \cdot \sum_{i=1}^{Comm} \frac{|S_i \cap \hat{S}_i|}{|S_i \cup \hat{S}_i|} \tag{7}$$

It holds that $acc \in [0, 1]$, the higher the accuracy the better the results. When $acc = 1$ the community detection algorithm gives perfect results. The Hungarian algorithm [24] is used to better match the estimated communities with the actual communities, in order to calculate and average the accuracies over all communities.

We also used the Normalized Mutual Information (NMI) metric to evaluate the correctness of the detected communities [25]. NMI is calculated using the

following formula:

$$NMI = \frac{\sum_{i=1}^{k}\sum_{j=1}^{l} n_{i,j} log(\frac{n \cdot n_{i,j}}{n_i \cdot n_j})}{\sqrt{(\sum_{i=1}^{k} n_i log(\frac{n_i}{n}))(\sum_{j=1}^{l} n_j log(\frac{n_j}{n}))}} \quad (8)$$

where i iterates through the population of the "correct" communities, j iterates through the communities detected by the algorithm, $n_{i,j}$ is the size of the union of the nodes of the i^{th} and j^{th} communities, n_i is the size of the i^{th} community and n is the total number of nodes in the graph. In both cases, a value close to one indicates correct detection of the communities of the graph.

4.3. Results on benchmark graphs

The performance of four different algorithms, SCCD presented in this paper, the Lancichinetti [7], Chen [19], and Blondel [20] algorithms are compared on Benchmark graphs. A brief description of these algorithms has been provided in the Related Work section of the paper.

Each graph shows the performance of a single algorithm using either the first or the second metric, on all 208 benchmark graphs. Since there are four types of parameters which describe the graphs of the experiments, we decided to use the two most important factors which affect the algorithms' performances, in order to plot the accuracies in 3D graphs. The first of these factors is always the local links to node degree value, which dictates how strong the clear the communities in the graph are. The second factor in some cases is the number of communities in the graph, while in other cases is the (average) density of these communities. Thus, we decided to plot, for each algorithm and accuracy metric, two 3D graphs using, in one case, the number of communities as the values on one the axes and the average density on the other. Each accuracy value in the graphs is the average of the accuracies of all the experiments based on the benchmark graphs with the same value on the aforementioned factors.

Fig. 13 shows the percentage of "successful" experiments, given an accuracy threshold to define "success". One can see the better performance of our algorithm, especially in "tougher" cases.

We can see from 3D graphs and from Fig. 13 how our algorithm greatly outperforms almost all other algorithms, algorithms, with the exception of the Blondel algorithm. Compared with Blondel, we see that the performance of SCCD is slightly higher. The average ACC and NMI of SCCD is 95.5% and 94.9%, respectively. The second highest performance method is Blondel algorithm that clearly outperforms the rest algorithms of literature. The average ACC and NMI

Graph	Nodes	Nr of coms		Modularity		Conductance		Coverage	
		SCCD	Blondel	SCCD	Blondel	SCCD	Blondel	SCCD	Blondel
Citations	27771	375	171	0.58	0.59	0.25	0.05	0.7	0.74
Enron email	36693	1590	1247	0.5	0.51	0.09	0.03	0.73	0.73
Amazon	262111	8943	177	0.87	0.89	0.3	0.1	0.88	0.92
stanford.edu	281904	3997	793	0.91	0.91	0.12	0.01	0.96	0.98
nd.edu	325730	6825	475	0.91	0.93	0.2	0.04	0.93	0.96

Table 2. Results on real world graphs without merging.

Graph	Nodes	Nr of coms		Modularity		Conductance		Coverage	
		SCCD	Blondel	SCCD	Blondel	SCCD	Blondel	SCCD	Blondel
Citations	27771	181	171	0.58	0.59	0.07	0.05	0.78	0.74
Enron email	36693	1300	1247	0.5	0.51	0.04	0.03	0.74	0.73
Amazon	262111	594	177	0.88	0.89	0.1	0.1	0.93	0.92
stanford.edu	281904	1122	793	0.93	0.91	0.02	0.01	0.98	0.98
nd.edu	325730	995	475	0.94	0.93	0.08	0.04	0.97	0.96

Table 3. Results on real world graphs with merging.

of Blondel algorithm is 89.4% and 89.7%, respectively. Apart from SCCD and Blondel, only Lancichinetti has produced some respectable results, however the figures show that it fails to work on less dense graphs. The average ACC and NMI of Lancichinetti is 66.1% and 72.3%, respectively. Lancichinetti has the advantage of being able to detect just one community (whereas SCCD and Blondel only produce all communities). In order to do so, however, it relies on the existence of triangles in the graph, which is the case only in more dense graphs, hence the observed results. Chen and Infomap obtained very low performance results.

5. Experiments on Real World Graphs

We also conducted experiments on five real world graphs of diverse sizes. Due to the size of those graphs, the only algorithms capable of analyzing them in reasonable time were our algorithm and the Blondel algorithm. These graphs include a network of scientific papers and their citations [26], an email communication network from Enron, two web graphs (of Stanford.edu and nd.edu) and an Amazon product co-purchasing network, all obtained from [27].

5.1. Comparison metrics

Three different metrics are used for the comparison of results on real world graphs, namely modularity, conductance, and coverage [21]. These are different than those used in the case of benchmark graphs, since the real decomposition of graphs into

communities is not known and as such the resulted community structure cannot be compared against the real one.

One of the most popular validation metrics for topological clustering, *modularity* states that a good cluster should have a bigger than expected number of internal edges and a smaller than expected number of inter-cluster edges when compared to a random graph with similar characteristics. The modularity Q for a clustering given below, where $e \in \Re^{k,k}$ is a symmetric matrix whose element e_{ij} is the fraction of all edges in the network that link vertices in communities i and j, and $Tr(e)$ is the trace of matrix e, i.e., the sum of elements from its main diagonal.

$$Q = Tr(e) - \sum_{i=1}^{k}(\sum_{j=1}^{k} e_{ij})^2 \tag{9}$$

The modularity Q often presents values between 0 and 1, with 1 representing a clustering with very strong community characteristics.

The *conductance* of a cut is a metric that compares the size of a cut (i.e., the number of edges cut) and the number of edges in either of the two subgraphs induced by that cut. The conductance $\phi(G)$ of a graph is the minimum conductance value between all its clusters.

Consider a cut that divides $G = (V, E)$ into k non-overlapping clusters $C_1, C_2, ..., C_k$. The conductance of any given cluster $\phi(C_i)$ is given by the following ratio:

$$\phi(C_i) = \frac{\sum_{(u,v)\in E \wedge u \in C_i \wedge v \notin C_i} 1}{\min\left(\alpha(C_i), \alpha(V \setminus C_i)\right)} \tag{10}$$

where

$$\alpha(C_i) = \sum_{(u,v)\in E \wedge u \in C_i \wedge v \in V} 1 \tag{11}$$

Essentially, $a(C_i)$ is the number of edges with at least one endpoint in C_i. This $\phi(C_i)$ value represents the cost of one cut that bisects G into two vertex sets C_i and $V \setminus C_i$ (the complement of C_i). Since we want to find a number k of clusters, we will need $k-1$ cuts to achieve that number. The conductance for the whole clustering is the average value of those $k-1$ ϕ cuts, as follows:

$$\phi(G) = avg(\phi(C_i)), \forall C_i \subseteq V \tag{12}$$

The final metric used to assess the performance of clustering algorithms on real world graphs is called *Coverage*. The coverage of a clustering C (where $C = C_1, C_2, ..., C_k$) is given as the fraction of the intra-cluster edges (E_C) with respect to all edges (E_G) in the whole graph G, $coverage(C) = \frac{E_C}{E_G}$. Coverage values usually range from 0 to 1. Higher values of coverage mean that there are more edges inside the clusters than edges linking different clusters, which translates to a better clustering.

5.2. Results on real world graphs

In Table 2, we present the results of running SCCD and Blondel [20] on these graphs, using three metrics for comparison, namely modularity, conductance and coverage. An explanatory survey of those metrics can be found in [21]. Both a high modularity and a high coverage indicate a better partitioning of the graph whereas in the case of conductance, a lower value is better. Although we included three metrics to get a better understanding of how the two algorithms behave in real world graphs, it is widely accepted that the modularity is the metric which better captures all the characteristics of a good partitioning of the graph into clear communities.

Two observations are quickly apparent from the results. One is that although the two algorithms locate a completely different number of communities for each graph, in most cases, the respective modularities are very comparable (in the order of 0.01). This shows that a "good" partitioning cannot be achieved in one way only. Coverage values are also comparable. This is not the case for conductance values where Blondel outperforms our algorithm. This is because conductance favors algorithms which "produce" a smaller number of communities. The main reason our algorithm prefers many, denser communities is the fact that it tries to locate strong communities. This stems from the algorithm assumption that the short links *per node* are more in number than the long links.

```
   input  : C = {c(x), ∀x ∈ V}.
   output: c'(x), ∀x ∈ V.
 1 foreach x ∈ V do
 2     c'(x) = c(x)
 3 end
 4 repeat
 5     S = 0
 6     for i = 1 to |C| do
 7         A = {x : c(x) = C_i}
 8         foreach B ~ A do
 9             if getDM(A, B) > 0 then
10                 c'(B) = c(A)
11                 A = A ∪ B
12                 V = V − B
13                 S = S + 1
14             end
15         end
16     end
17 until S = 0
```

Algorithm 3: The Correction Clustering algorithm.

In order to verify this, we modified our algorithm by adding a third phase

which iteratively merges the communities found, if this results in an increase of the modularity. The pseudo-code of this face is given in Algorithm 3 and it is described hereafter. Let $c'(x)$ denotes of updated cluster id of node x. Although the modularity is calculated on the entire graph, the change in the modularity value dm can be computed only using information related to the communities (i,j) which are to be merged, since the substraction eliminates the values in the modularity table of communities not participating in the merge. This is implemented by function $getDM(.,.)$ (see line 9 of algorithm 3).

$$dm = \dot{e}_{ii} - \dot{a}_i^2 - (e_{ii} + e_{jj} - a_i^2 - a_j^2), \tag{13}$$

where $\dot{e}$ is the modularity table before the merge and e is after the merge. Thus e_{ii} and $\dot{e}_{ii}$ are the fraction of all edges in the graph with both ends connected to nodes in community i, before and after the merge, and $\dot{a}_i$ and a_i are the fractions of edges with at least one end vertex in community i before the merge and after the merge, respectively. Note that community i in the merged graph corresponds to the community that results from merging communities i and j before the merge.

The only global information required in this phase is the number of edges in the initial graph, in order to calculate the aforementioned fractions. Although this somehow limits the completely distributed nature of our algorithm, it is an information which is easily obtained and furthermore, in contrast to the Blondel algorithm, does not change as communities are merged. Table 3 shows the new results after we apply the merging step. We can see now that both the number of communities and the conductance value are also comparable in the two algorithms. This shows that there is a range in the number of communities that equally produce partitions of quality (modularity values).

It is also noteworthy that using this final step, our algorithm is able to produce any desired (user-defined) number of communities from the range of quality partitioning (highest modularity value)

5.3. Method Sensitivity to Parameters and Initial Conditions

The proposed method has two main parameters the T_1 and T_3 that have been set 0.6 and 0.5 under all experiments. In this section we have performed experiments to measure the sensitivity of the proposed method to several values of T_1 and T_3. In addition, we have tested the stability of the proposed algorithm, i.e. how much the output is affected by the (random) initial conditions from which it starts.

Fig. 14 depicts the modularity and the number of communities of Citations real graph for different values of T_3. Modularity receives high values for several values of T_3 ($T_3 \in [0.2, 0.6]$), which means that the proposed method is not sensitive on T_3 changes. When the merging process is not executed (see Fig. 14(b)), it holds

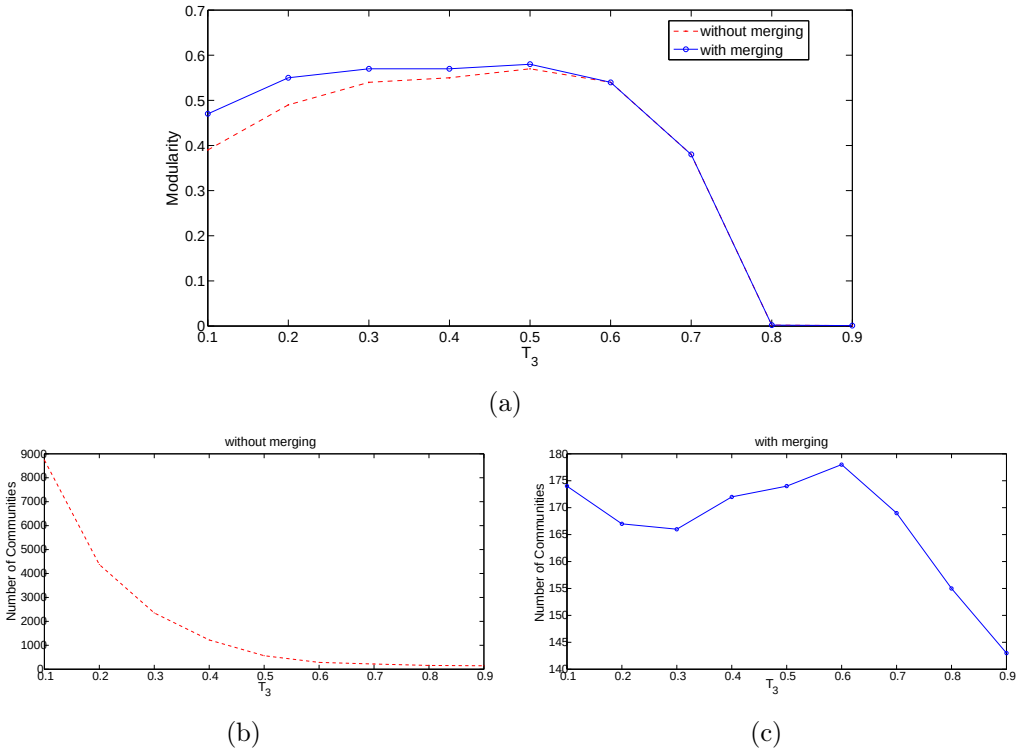


Figure 14. (a),(b) The modularity and the number of communities of Citations real graph for different values of T_3.

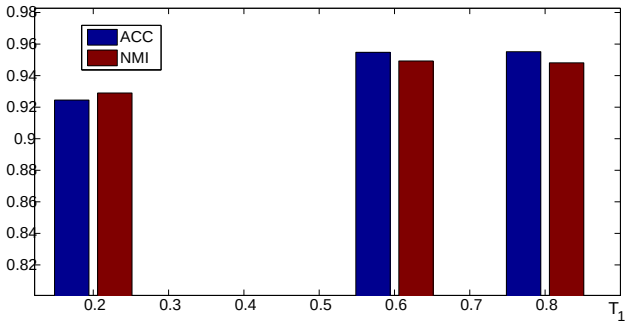


Figure 15. The average value of ACC and NMI for different values of T_1 under the 208 benchmark graphs.

that when T_3 gets low values, then the proposed method results high number of communities. Otherwise, if T_3 gets high values, then we get low number of communities. When the merging process is executed the number of communities is not really affected by T_3, since the merging significantly decreases the number of communities (see Fig. 14(c)).

Fig. 15 the average value of ACC and NMI for different values of T_1 under the 208 benchmark graphs. It holds that the performance of the method is not really affected by changing T_1 ($T_1 \in [0.2, 0.8]$).

In order to examine how much the performance of the proposed method is affected by the (random) initial conditions from which it starts, we have executed it for each benchmark graph ten times and we measure the standard deviation of ACC and NMI getting very low values, 0.0079 and 0.0051 on average, respectively. This means that the initial conditions does not affect the performance of the proposed method.

6. Conclusions and Future Work

We presented a community finding algorithm which is based on a custom-tailored version of the Vivaldi network coordinate system. The proposed algorithm has been tested on a large number of benchmark graphs with known community structure comparing it with several state-of-the-art algorithms, proving its effectiveness against all other algorithms. In addition we performed experiments on large, real world datasets and compared it with the next most efficient algorithm resulting in very comparable effectiveness. Moreover, our algorithm can employ a simple third step to allow it to provide a wider range of similarly optimal results.

We plan to expand the algorithm, in order to enable the detection of also overlapping communities. In addition, another goal is the modification of the algorithm in order to locate only a single community. Both of these problems are of great interest to the field of social networks, since overlaps can be appeared between communities. In addition, it an is important to provide the single community detection (per node) when each node of the social network ask for its community instead of entire community detection. Another possible extension of the proposed scheme is the application in weighted networks that can measure the strength of social relationships in social networks [28]. In iterative process of position estimation algorithm, this extension can be done by setting the probability of edge selection according to the edge weight, so that the strong edges would have higher selection probability corresponding to high-tension springs.

Finally, there is an emerging need to devise community detection algorithms for dynamic graphs, i.e. graphs whose structure evolved over time. Such algorithms would be able to capture for example the dynamic evolution of social networks.

So, we plan to extend the proposed community finding method for dynamic graphs. This extension is possible, since SCCD is a fully distributed system, under the constraint that the dynamic evolution of social network is slower than the adaptation of the Vivaldi synthetic network coordinates and the proposed distributed hierarchial clustering algorithm.

Acknowledgments

This research has been partially co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Programs: ARCHIMEDE III-TEI-Crete-P2PCOORD.

References

[1] Gary William Flake, Steve Lawrence, C. Lee Giles, and Frans M. Coetzee. Self-organization and identification of web communities. *IEEE Computer*, 35:66–71, March 2002.

[2] Dimitrios Katsaros, George Pallis, Konstantinos Stamos, Athena Vakali, Antonis Sidiropoulos, and Yannis Manolopoulos. Cdns content outsourcing via generalized communities. *IEEE Transactions on Knowledge and Data Engineering*, 21:137–151, 2009.

[3] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences of the United States of America*, 99(12):7821–7826, June 2002.

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

[5] Mo-Han Hsieh and Christopher L Magee. A new method for finding hierarchical subgroups from networks. *Social Networks*, 32(3):234–244, 2010.

[6] Symeon Papadopoulos, Andre Skusa, Athena Vakali, Yiannis Kompatsiaris, and Nadine Wagner. Bridge bounding: A local approach for efficient community discovery in complex networks. Technical Report arXiv:0902.0871, Feb 2009.

[7] Andrea Lancichinetti, Santo Fortunato, and János Kertész. Detecting the overlapping and hierarchical community structure in complex networks. *New Journal of Physics*, 11(3):033015+, March 2009.

[8] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of the ACM SIGCOMM '04 Conference*, August 2004.

[9] Harris Papadakis, Paraskevi Fragopoulou, and Costas Panagiotakis. Distributed community detection: Finding neighborhoods in a complex world using synthetic coordinates. In *ISCC'11*, pages 1145–1150, 2011.

[10] Andrea Lancichinetti and Santo Fortunato. Community detection algorithms: a comparative analysis. *Physical Review E*, 80(5 Pt 2):056117, Sep 2009.

[11] Satu Elisa Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27 – 64, 2007.

[12] Reid Andersen, Fan Chung, and Kevin Lang. Local graph partitioning using pagerank vectors. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 475–486. IEEE, 2006.

[13] Pan Hui, Eiko Yoneki, Shu Yan Chan, and Jon Crowcroft. Distributed community detection in delay tolerant networks. In *Proceedings of 2nd ACM/IEEE international workshop on Mobility in the evolving internet architecture*, page 7. ACM, 2007.

[14] Imre Derényi, Gergely Palla, and Tamás Vicsek. Clique Percolation in Random Networks. *Physical Review Letters*, 94(16):160–202, Apr 2005.

[15] Gergely Palla, Imre Derenyi, Illes Farkas, and Tamas Vicsek. Uncovering the overlapping community structure of complex networks in nature and society, June 2005.

[16] F. Wu and B. A. Huberman. Finding communities in linear time: a physics approach. *The European Physical Journal B - Condensed Matter and Complex Systems*, 38(2):331–338, March 2004.

[17] Stijn Van Dongen. Graph clustering via a discrete uncoupling process. *SIAM J. Matrix Anal. Appl.*, 30:121–141, February 2008.

[18] James P. Bagrow and Erik M. Bollt. Local method for detecting communities. *Physical Review E*, 72(4):46–108, Oct 2005.

[19] Jie Chen and Yousef Saad. Dense subgraph extraction with application to community detection. *IEEE Transactions on Knowledge and Data Engineering*, 24:1216–1230, 2012.

[20] V.D. Blondel, J.L. Guillaume, R. Lambiotte, and E.L.J.S. Mech. Fast unfolding of communities in large networks. *J. Stat. Mech*, page P10008, 2008.

[21] Hélio Almeida, Dorgival Guedes, Wagner Meira, and Mohammed J. Zaki. Is there a best quality metric for graph clusters? In *Proceedings of the 2011 European conference on Machine learning and knowledge discovery in databases - Volume Part I*, ECML PKDD'11, pages 44–59, Berlin, Heidelberg, 2011. Springer-Verlag.

[22] M. Rosvall and C.T. Bergstrom. An information-theoretic framework for resolving community structure in complex networks. *Proceedings of the National Academy of Sciences*, 104(18):7327, 2007.

[23] H. Frigui and R. Krishnapuram. A robust competitive clustering algorithm with applications in computer vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(5):450–465, 1999.

[24] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982.

[25] Alexander Strehl, Joydeep Ghosh, and Claire Cardie. Cluster ensembles - a knowledge reuse framework for combining multiple partitions. *Journal of Machine Learning Research*, 3:583–617, 2002.

[26] Cornell kdd cup.

[27] Stanford large network dataset collection.

[28] Tore Opsahl and Pietro Panzarasa. Clustering in weighted networks. *Social networks*, 31(2):155–163, 2009.