A definition of scale-independent communities & a flow-free vertex cut algorithm

Alexander Braekevelt, Pieter Leyman & Patrick De Causmaecker

Abstract

Finding groups of connected individuals or communities in graphs has received considerable attention in literature. In an attempt to quantify the intuitive concept of "strongly connected individuals with few outgoing connections", many definitions have been proposed. Most of these, however, provide little insight in the quality and properties of the resulting subgraphs in general graphs. Concrete applications might require more rigorous definitions. We propose to analyze different characteristics from literature, derived from clique relaxations, and determine a better definition of connectedness. An important aspect in this regard is the scalability of the definition, which we explicitly take into account. We subsequently discuss our algorithm, based on the Moody & White approach for finding vertex connected sets and a new flow-free vertex cut algorithm, to detect the defined types of communities given initial values. Hence, the proposed algorithm is able to identify groups of vertices which satisfy predefined properties. Possible applications include social media network analysis (e.g. Facebook, Twitter), bioinformatics and real-world vehicle routing.

1. Introduction

Networks are a useful tool for modeling relations between entities. Because of big data, the world collects more relational data today than ever before. In search for powerful insights, people search for idealized patterns in those relations. One of those patterns is the equivalence relation. In the context of networks, this is more commonly known as communities. Communities are groups of elements in the network that have a high number of connections inside the group and a low number of connections between the groups. Ideally, the network consists of fully connected groups that share no connections. This way, elements in the same group are identical with respect to the relation. In real data, this might correspond to a common function or property.

Real data, however, are seldom perfect. Incomplete information and errors distort the observed connections. Therefore, a definition is needed that determines how strong a group of elements resembles the perfect community. The search for a good definition has been the topic of many research in the past decades. Despite these efforts, there is still no consensus about how to best define and find communities. As Fortunato [1] remarks, "what the field lacks the most is a theoretical framework that defines precisely what clustering algorithms are supposed to do" which caused the field to grow "in a rather chaotic way, without a precise direction or guidelines". This lack of framework makes it very hard to estimate the quality of the results of the existing methods.

To avoid adding yet another definition to an already crowded field of research, the definition in this paper will give strict guaranties on the properties of the produced results. Specifically, the research questions of this paper are:

RQ1 What is a good definition for a community such that it's properties correspond to the intuitive concept of "strongly connected individuals with few outgoing connections"?

RQ2 How does an algorithm for finding this definition scale with respect to the data size?

In section 3, the definition of RQ1 will be derived from the taxonomic framework of clique relaxation models by Pattillo [2]. Section 4 will describe the algorithm of RQ2 for finding communities with this definition, based on the algorithm of Moody & White for finding connectivity sets [3]. A notable contribution of this paper to the algorithm is the invention of a fast, local, flow-free vertex cut algorithm. This contribution will be discussed in more detail in section 5 and 6. Finally, the performance will be analyzed in section 7.

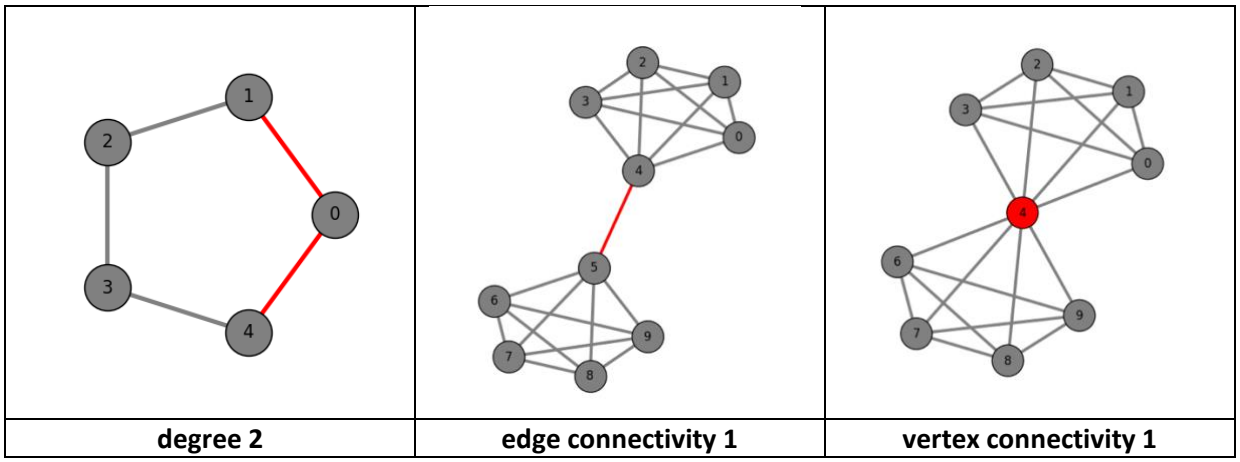
2. Notations, terms and fundamental theorems

Following the usual mathematical notation, a network will be referred to as a graph, called $\mathrm{G}(\mathrm{V}, \mathrm{E})$ or just G, consisting of a set of vertices V and a set of edges E. The size of the sets V and E are referred to as n and m respectively. A vertex v in V, also called a node or an object, is an element of the graph. An edge $e = (v_i, v_j)$ in E, also called an arc or a relation, is a connection between an unordered pair of different vertices. The vertices v_i and v_j in an edge are neighbours.

The number of neighbours of a vertex is called its degree. Every vertex that can be reached by following the edges starting from some vertex is connected to and reachable from that vertex. A graph is connected if all pairs of its vertices are connected. The connectivity can be quantified using either edges or vertices. Two vertices are said to be k-edge-connected if a minimum of k edges must be removed to disconnect them. They are k-vertex-connected if a minimum of k vertices must be removed to disconnect them. The removed set of k edges or vertices is called a minimum edge- or vertex-cut. These definitions extend to the connectivity of a graph if they are satisfied by all pairs of vertices. A k-vertex-connected subgraph is also called a k-vertex-component. A k-edge-connected subgraph is called a k-edge-component. A subgraph with minimum degree k is a k-core. Figure 1 illustrates some of the described properties.

The max-flow min-cut theorem [4] proves that the minimum edge- and vertex-cut of size k is equivalent with k edge- or vertex-independent paths between the vertices. Whitney's theorem [5] states that the minimum degree of a graph is an upper bound to the minimum edge-connectivity, which in turn is an upper bound to the vertex-connectivity. This implies that a k-vertex-connected component is embedded in a k-edge-component, which in turn is embedded in a k-core.

Using this notation, the problem of finding a community is described as finding a set of vertices $S = \{v_1, v_2, \dots, v_n\}$ that satisfies a chosen definition, given an input graph G. The subgraph of G in which only the vertices of S are not removed is referred to as $G[S]$.



degree 2	edge connectivity 1	vertex connectivity 1

Figure 1: *examples to illustrate some properties of graphs.*

3. A scale-independent definition of communities

To find a measure of how strong a group of elements resembles the perfect community, a logical first step is to define the perfect community itself. In graph theory, a perfect community of maximal connected vertices is called a clique. Cliques are well studied graphs with many known properties. By relaxing one (or more) of these properties, many definitions can be constructed. These definitions are called clique-relaxations. The definitions all have a value, say 'x', that determines how much a certain property is preserved or violated. This value may either be set beforehand to search communities or calculated afterward as a measure for a given set of vertices. Pattillio et al. [2] have constructed a framework that classifies the existing clique relaxations and reveals possible new ones. In Table 1, only a simplified and slightly modified version of this framework is given. The details of the definitions can be found in the original paper [2], but they are of lesser importance to the following discussion. The left side of Table 1 lists some of the basic clique properties. The top tells if the value 'x' either ensures a lower bound on a property or an upper bound on its violation. Notice that relative definitions are dependent on the number of vertices of the community and thus are the same for both the upper- and lower bound. The names of already existing definitions are in black, new ones are in grey.

By analyzing the characteristics of the definitions in the framework, it became clear that only some of them really provided good guaranties on the vertex connectivity. The derived bounds are listed in Table 2. This is surprising because communities that can be split by removing a small number of vertices can hardly be considered good communities. This is especially true if you consider that this also corresponds to the number of redundant paths between the vertices, in other words the robustness of the community. This observation left only the domination, degree, edge connectivity and vertex connectivity as feasible options. The domination is however equivalent to the vertex connectivity. Because the remaining definitions are exactly those of the inclusion relation of Whitney's theorem [5], the vertex connectivity is known to be the strongest of the three. As shown by Table 3, his definition also has very good bounds on the other properties. Therefore, this will be the definition of choice in this paper.

Considering the scalability of the vertex connectivity, three versions remain. Their values will be referred to as 'x'. The absolute lower bound version, called the k-block, demands a minimum vertex connectivity of x. The relative version requires a vertex connectivity of $x\ (n\ -\ 1)$ where $0\ \leq\ x\ \leq\ 1$, since the maximum possible connectivity is $(n\ -\ 1)$. The absolute upper bound version, called the s-bundle, needs a vertex connectivity of $n\ -\ 1\ -\ x$ [1]. The relative version seems to be the best because it makes it possible to compare the strength of communities with different sizes. In comparison, the k-block is overly permissive and the s-bundle is overly restrictive for large communities with respect to the number of edges, as illustrated by Figure 2. However, the algorithm in the next section will allow all three versions because the desired scalability may be different depending on the application.

This section has led to the following conclusion for the first research question of section 1:

C1 A good definition for a community is the vertex connectivity. As shown in Table 3, all varieties of this definition require reasonable values for clique-defining properties. Additionally, Table 2 shows that this is not the case for the other clique-relaxations.

[1] In Table 2 the absolute upper bound of the vertex connectivity definition has a vertex connectivity of $n\ -\ x$ as defined by Pattillio et al. [2]. In this paper $n\ -1-\ x$ is preferred because then x can also be zero. However, the constant difference in value does not contribute to any significant difference.

	ensure lower bound		relax upper bound	
	absolute	**relative**	**absolute**	**relative**
Distance	s-clique	def. 3	def. 8	def. 3
Diameter	s-club	def. 4	def. 9	def. 4
Domination	s-plex	def. 5	def. 10	def. 5
Degree	k-core	λ in (λ,γ)-quasi-clique	s-plex	λ in (λ,γ)-quasi-clique
Edge connectivity	def. 1	def. 6	def. 11	def. 6
Vertex connectivity	k-block	def. 7	s-bundle	def. 7
Edge density	def. 2	γ-quasi-clique	s-defective-clique	γ-quasi-clique

Table 1: *simplified and slightly modified version of the taxonomic framework of clique relaxation models by Pattillo et al. [2].*

Definition	**Vertex connectivity k**	
s-clique	$k \geq 0$ *	**absolute lower bound**
s-club	$k \geq 1$ *	
s-plex	$k \geq n - 2x + 2$ *	
k-core	$k \geq 2x + 2 - n$ *	
def. 1	$k \geq 2x + 2 - n$	
k-block	$k \geq x$ *	
def. 2	$k \geq \lceil x\,n / 2 - (n-1)(n-2) / 2 \rceil$ *	
def. 3	$k \geq 0$ *	**relative bound**
def. 4	$k \geq 1$ *	
def. 5	$k \geq n - 2x(n-1) + 2$ *	
λ in (λ,γ)-quasi-clique	$k \geq 2x(n-1) + 2 - n$ *	
def. 6	$k \geq 2x(n-1) + 2 - n$	
def. 7	$k \geq x(n-1)$ *	
γ-quasi-clique	$k \geq \lceil x\,n(n-1) / 2 - (n-1)(n-2) / 2 \rceil$ *	
def. 8	$k \geq 0$ *	**absolute upper bound**
def. 9	$k \geq 1$ *	
def. 10	$k \geq n - 2(n-x) + 2$ *	
s-plex	$k \geq 2(n-x) + 2 - n$ *	
def. 11	$k \geq 2(n-x) + 2 - n$	
s-bundle	$k \geq n - x$ *	
s-defective-clique	$k \geq \lceil (n-x)\,n / 2 - (n-1)(n-2) / 2 \rceil$ *	
clique (= fully connected)	$\boldsymbol{k = n - 1}$	

Table 2: *derived vertex connectivity properties of the definitions in Table 1. 'x' is a value for the definition and 'k' is the derived vertex connectivity. 'n' and 'm' are the number of vertices and edges in the community. Properties marked with * are tight bounds.*

Distance	$dG(v1, v2) \leq \lfloor (n - 2) / x + 1 \rfloor$	**k-block**
Diameter	$diam(S) \leq \lfloor (n - 2) / x + 1 \rfloor$	
Domination	$\lvert D \rvert \leq n - x$	
Degree	$\delta(S) \geq x$	
Edge connectivity	$\lambda(S) \geq x$	
Vertex connectivity	$\kappa(S) \geq x$	
Edge density	$\rho(S) \geq x / (n - 1)$	
Distance	$dG(v1, v2) \leq \lfloor (n - 2) / (x (n - 1)) + 1 \rfloor$	**def. 7**
Diameter	$diam(S) \leq \lfloor (n - 2) / (x (n - 1)) + 1 \rfloor$	
Domination	$\lvert D \rvert \leq n - x (n - 1)$	
Degree	$\delta(S) \geq x (n - 1)$	
Edge connectivity	$\lambda(S) \geq x (n - 1)$	
Vertex connectivity	$\kappa(S) \geq x (n - 1)$	
Edge density	$\rho(S) \geq x$	
Distance	$dG(v1, v2) \leq \lfloor (n - 2) / (n - x) + 1 \rfloor$	**s-bundle**
Diameter	$diam(S) \leq \lfloor (n - 2) / (n - x) + 1 \rfloor$	
Domination	$\lvert D \rvert \leq x$	
Degree	$\delta(S) \geq n - x$	
Edge connectivity	$\lambda(S) \geq n - x$	
Vertex connectivity	$\kappa(S) \geq n - x$	
Edge density	$\rho(S) \geq (n - x) / (n - 1)$	

Table 3: *the defining properties of the vertex connectivity definitions.*

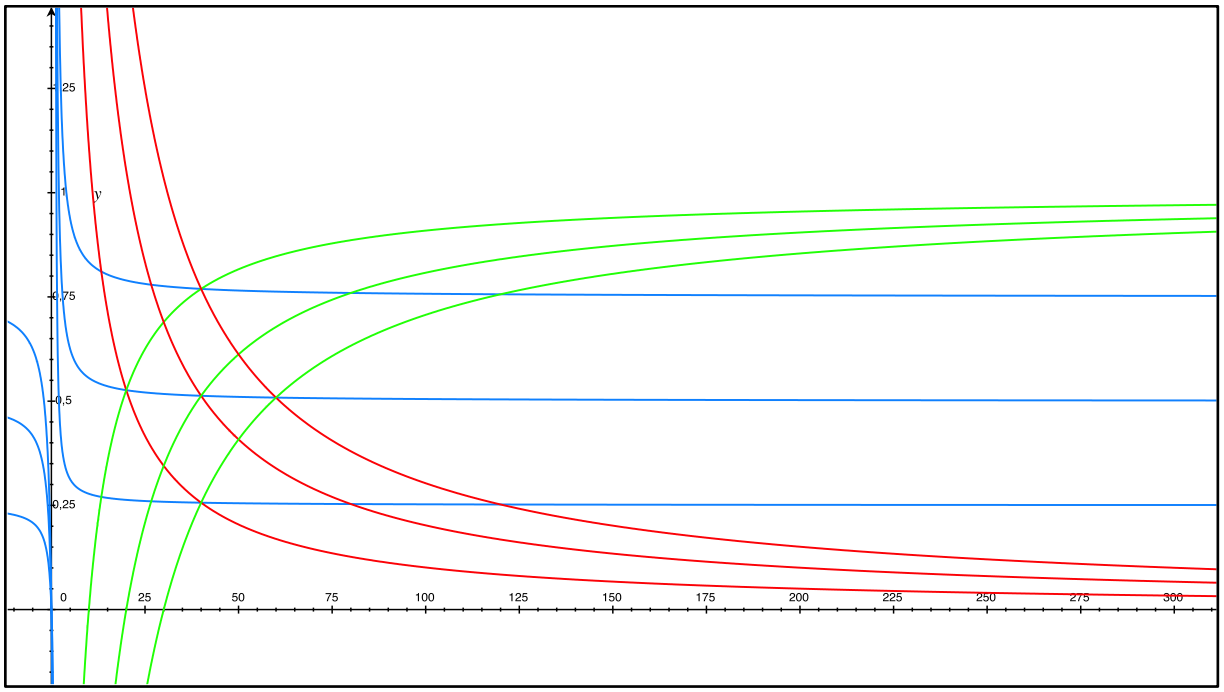


Figure 2: *minimum required percentage of edges in function of number of vertices.*
red: *k-block;* ***green:*** *s-bundle;* ***blue:*** *def. 7*

4. An algorithm for the community decomposition of a network

The algorithm for finding communities in a graph is given in Algorithm 1. It is based on the algorithm for finding vertex connected sets by Moody & White [3]. The input is a graph and the values for the definition of section 3. The output consists of sets of vertices that represent the communities. It works by repeatedly finding a minimal vertex cut in the input graph which splits the graph in two components (9). The vertices in the cut are added to both components. The algorithm is then repeated for the subgraphs defined by these components (12-16). After every cut, the vertex connectivity of the split graph is known, precisely because this is defined as the length of the cut. If this vertex connectivity satisfies the values chosen in the definition, the split graph is a community and thus added to the results (10-11).

Because a vertex-component is always embedded within a vertex-component that has a vertex connectivity that is equal or smaller, the vertex-components form an inclusion hierarchy of increasing vertex connectivity all the way down. Therefore, a component with a lower vertex connectivity than the cut from which it was separated can never be a valid community, because this indicates that another minimal vertex cut in the graph was part of the component. If this would not be the case, this would mean the cut was not minimal. For this reason, the calculated vertex cut does not always need to be minimal, because cuts smaller than previous cuts do not split stronger communities. The pseudocode of this algorithm is given in Algorithm 1 and examples of found communities are shown in Figure 3.

```
   algorithm get_communities is
        input:      graph g, number x_lower, float x_relative, number x_upper
        output:     set of communities (= sets of vertices)

1       results <— {}
2       stack <— [ (vertices of g, -1) ]
3       while stack is not empty do
4           (vertex_set, parent_k) <— last element of stack
5           stack <— stack \ [ (vertex_set, parent_k) ]
6           s <— subgraph of g limited to the nodes in vertex_set
7           v <— number of vertices in s
8           e <— number of edges in s
9           (k, a, b) <— sufficient_vertex_cut(subgraph, parent_k)
10          if k > parent_k and is_valid_community(k, v, x_lower, x_relative, x_upper) do
11              results <— results ∪ {vertex_set}
12          if e ≠ v * (v - 1) / 2 do
13              new_k <— max {k, parent_k}
14              a <— {vertices in a with degree > new_k}
15              b <— {vertices in b with degree > new_k}
16              stack <— stack ∪ [ (a, new_k), (b, new_k) ]
17      return results
18
```

Algorithm 1: *the algorithm for finding communities in a graph.*

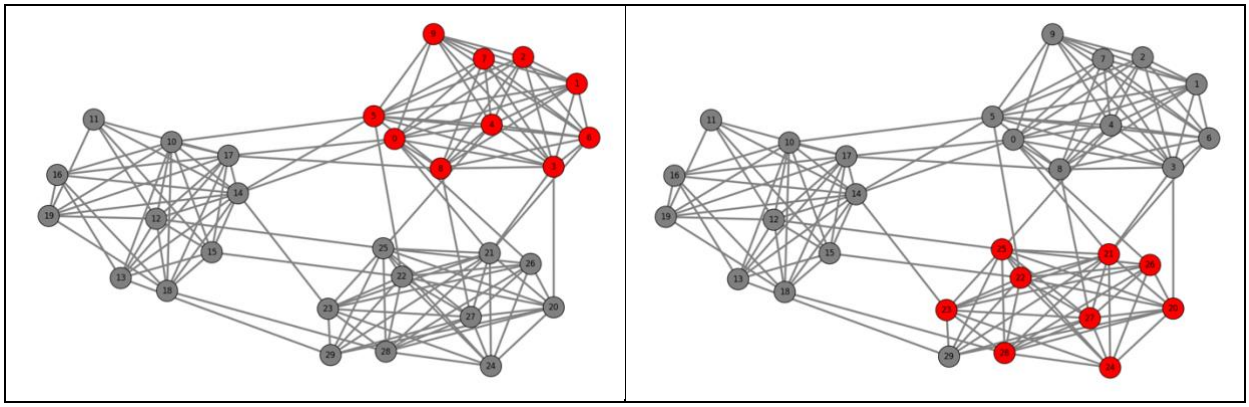


Figure 3: *graphs found with relative vertex connectivity of 0.25.*

```
    algorithm sufficient_vertex_cut is
            input:          graph g, number sufficient_k
            output:         number k, set of vertices a, set of vertices b

1           v <— vertex in g with minimum degree
2           v_neighbours <— neighbours of v in g
3           a <— v_neighbours ∪ {v}
4           non_neighbours <— {vertices of g} \ a
4           b <— non_neighbours ∪ v_neighbours
5           k <— length of v_neighbours
6           if k ≤ sufficient_k do
7                   return (k, a, b)
8
9           for each w in non_neighbours do
10                  this_k, _, this_a, this_b <— flow_free_vertex_cut(g, v, w, sufficient_k)
11                  if k > this_k do
12                          k, a, b <— this_k, this_a, this_b
13                          if k ≤ sufficient_k do
14                                  return (k, a, b)
15          for each x in v_neighbours do
16                  for each y in v_neighbours do
17                          this_k, _, this_a, this_b <— flow_free_vertex_cut(g, x, y, sufficient_k)
18                          if k > this_k do
19                                  k, a, b <— this_k, this_a, this_b
20                                  if k ≤ sufficient_k do
21                                          return (k, a, b)
22          return (k, a, b)
```

Algorithm 2: *the algorithm for finding a global vertex cut in a graph.*

5. A global vertex cut algorithm

The vertex cuts in Algorithm 1 are calculated using Algorithm 2. These vertex cuts are called global cuts because they apply to the whole graph. Vertex cuts between a specific pair of vertices in the graph are called local vertex cuts. The basic idea of Algorithm 2 is to stop the calculation as soon as possible, since calculating a global vertex cut is a very expensive operation. This is done by halting as soon as the local minimal cut found so far is sufficient for splitting the graph (6, 13, 20), hence the name 'sufficient_vertex_cut'. The input is a graph and a number that indicates which size of cut is sufficient. The latter value is equal to the maximum sizes of all global cuts which have previously split the given graph. As mentioned in section 4, this does not alter the results since a sufficient cut does not split communities. If no local cut is found below or equal to the sufficient cut size, the cut will be equal to the exact global minimal vertex cut, because every possible local minimal cut is checked.

Because every vertex in the graph is part of one or both sides of the cut, if is sufficient to take one vertex and calculate the minimal local vertex cuts between this vertex and all other vertices in the graph (9). After that, the minimal local cuts between all non-adjacent neighbours of the vertex have to be checked (15), because it is also possible that the vertex itself belongs to the minimal cut. Note that this approach is way more efficient than checking the local cut between every pair of vertices. This approach was based on algorithm 11 of Esfahanian [6] and its implementation in the NetworkX library. The pseudocode of this algorithm is given in Algorithm 2.

6. A flow-free local vertex cut algorithm

The minimal local vertex cuts of Algorithm 2 are calculated using Algorithm 3. To our knowledge, this is the first time that an algorithm has been found for vertex cuts that does not use a graph transformation to reduce the problem to the problem of finding local minimal edge cuts via flow computations. This new flow-free vertex cut algorithm takes as input a graph, a start vertex s and an end vertex t. In our implementation, a sufficient cut size is added to reduce the computation in the same way as Algorithm 2, but of course this is optional. The output is the vertex connectivity number, the vertex cut and the two resulting components.

The fundamental insight of this new algorithm is that when traversing the graph from vertex s to vertex t (or vice versa) with a moving vertex cut (called a 'wave'), the size of the vertex cut always significantly increases beyond a cut node. If the vertex cut would decrease or stay equal, then a smaller or equivalent vertex cut would exist. Let us clarify this statement. Take as initial vertex cut the neighbours of vertex s. Keep track of the vertices that have already been encountered, initially s and its neighbours. Proceed by taking the vertex in the current cut with the least connections to vertices that have not been encountered. Remove this vertex from the cut and add its neighbours instead. Also add those neighbours to the encountered vertices. Now keep repeating this procedure. By remembering which of the encountered vertex cuts was minimal, the minimal vertex cut can be found.

The only problem left now is when to stop repeating the procedure. If continued long enough, all vertices will be encountered and the cut will just become smaller and smaller, but this does not represent an actual vertex cut anymore. This problem is solved by simultaneously moving two 'waves', one from s and one from t. The selected vertex is then the vertex with the least connections to vertices that have not been encountered by the wave of which it is a member. When the waves have one of more vertices in common, these vertices are taken out of the waves and put into a set of intersection vertices. These vertices can no longer be selected for removal but are still part to the vertex cut of both waves. The pseudocode of the described actions is given in Algorithm 3 and a sample execution is shown in Table 4.

By design, the flow-free vertex cut algorithm will always return a valid cut of the graph. It must however be noted that this cut will not always be minimal. Figure 4 gives an example of such a suboptimal cut. Nevertheless, this example is a hand-crafted edge case. Suboptimal results happen rarely in practice. Not a single generated graph in our tests resulted in suboptimal cuts. The frequency of this error is however still unknown and demands further research in future work.

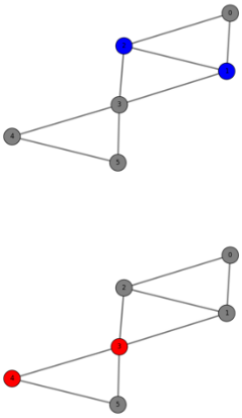
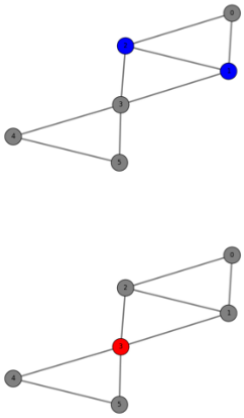
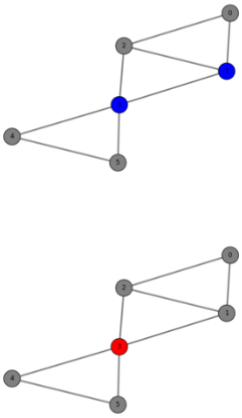
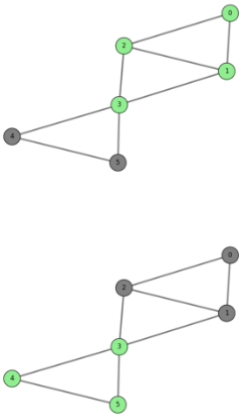
			
step 1	**step 2**	**step 3**	**step 4**

Table 4: *example execution of the flow-free vertex cut algorithm.*

```
algorithm flow_free_vertex_cut is
        input:          graph g, vertex s, vertex t, number sufficient_k
        output:         number k, set of vertices cut, set of vertices a, set of vertices b

1       cut_graph <— copy of g
2       s_wave <— [ neighbours of s in g ]
3       t_wave <— [ neighbours of t in g ]
4       s_a <— {s}
5       t_a <— {t}
6       remove s and t from cut_graph
7       remove edges in cut_graph between vertices in s_wave
8       remove edges in cut_graph between vertices in t_wave
9
10      wave_intersection <— s_wave ∩ t_wave
11      s_wave <— s_wave \ wave_intersection
12      t_wave <— t_wave \ wave_intersection
13
        …
```

```
        …
14
15      if length of s_wave < length of t_wave do
16              cut <— s_wave ∪ wave_intersection
17              a <— copy of s_a
18      else do
19              cut <— t_wave ∪ wave_intersection
20              a <— copy of t_a
21
22      while s_wave is not empty and t_wave is not empty
23                                      and length of cut > sufficient_k do
24              s_vertex <— vertex in s_wave with lowest degree in cut_graph
25              t_vertex <— vertex in t_wave with lowest degree in cut_graph
26
27              if degree of s_vertex in cut_graph ≤ d degree of t_vertex in cut_graph do
28                      new <— {neighbours of s_node in cut_graph} \ s_wave
29                      s_wave <— s_wave \ {s_vertex}
30                      s_wave <— s_wave ∪ new
31                      s_a <— s_a ∪ {s_vertex}
32                      remove s_vertex from cut_graph
33                      remove edges in cut_graph between vertices in new and in s_wave
34              else do
35                      new <— {neighbours of t_node in cut_graph} \ t_wave
36                      t_wave <— t_wave \ {t_vertex}
37                      t_wave <— t_wave ∪ new
38                      t_a <— t_a ∪ {t_vertex}
39                      remove t_vertex from cut_graph
40                      remove edges in cut_graph between vertices in new and in t_wave
41
42              wave_intersection <— s_wave ∩ t_wave
43              s_wave <— s_wave \ wave_intersection
44              t_wave <— t_wave \ wave_intersection
45
46              if length of s_wave + length of wave_intersection < length of cut do
47                      cut <— s_wave ∪ wave_intersection
48                      a <— copy of s_a
49              else if length of t_wave + length of wave_intersection < length of cut do
50                      cut <— t_wave ∪ wave_intersection
51                      a <— copy of t_a
52
53      b <— {vertices of g} \ a
54      a <— a ∪ cut
55      k <— length of cut
56
57      return (k, cut, a, b)
```

Algorithm 3: *the algorithm for finding a local vertex cut in a graph.*

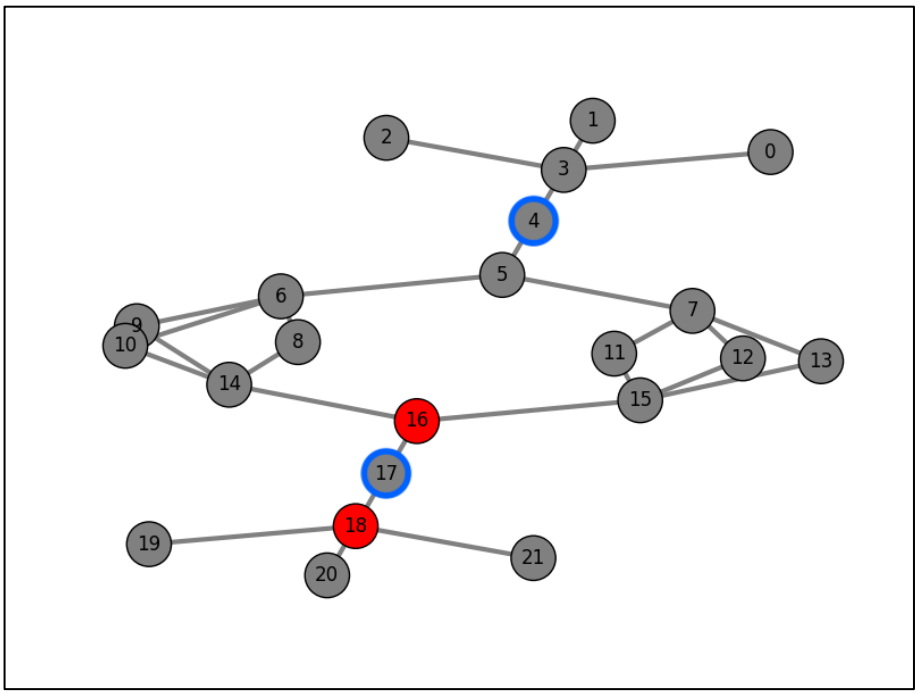


Figure 4: *example of a suboptimal cut by "flow_free_vertex_cut".*
The nodes marked in red are returned as cut between node 4 and node 17.

7. Performance analysis

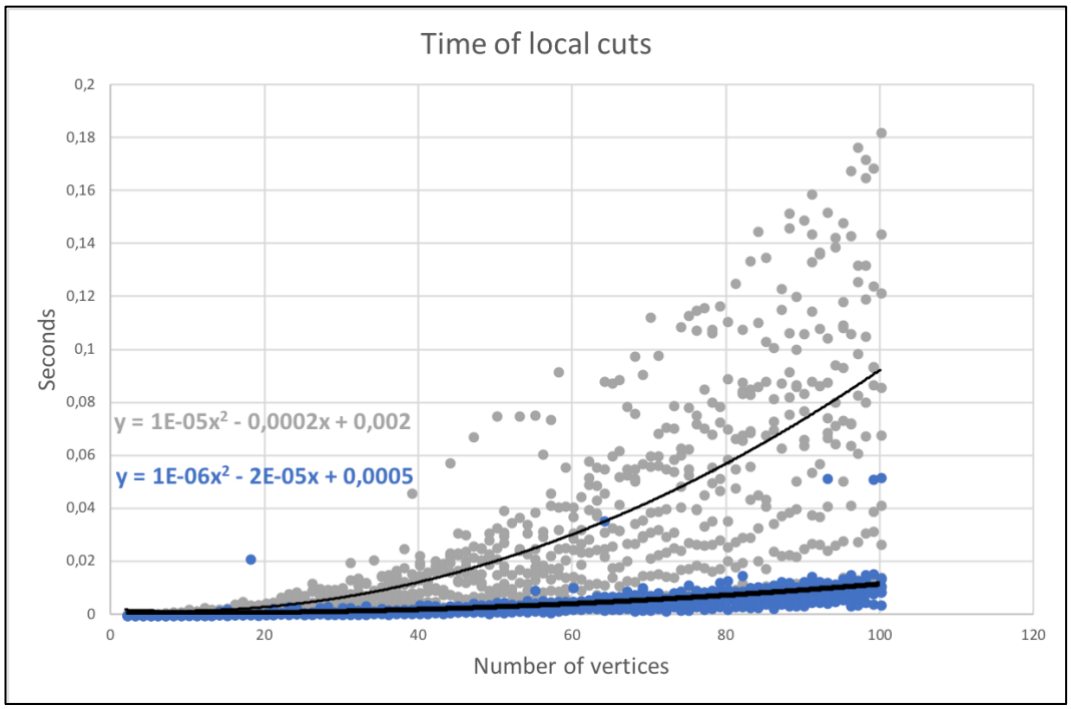


Figure 5: *performance data of "flow_free_vertex_cut" in blue.*
The local cut algorithm of the NetworkX library is shown in grey.

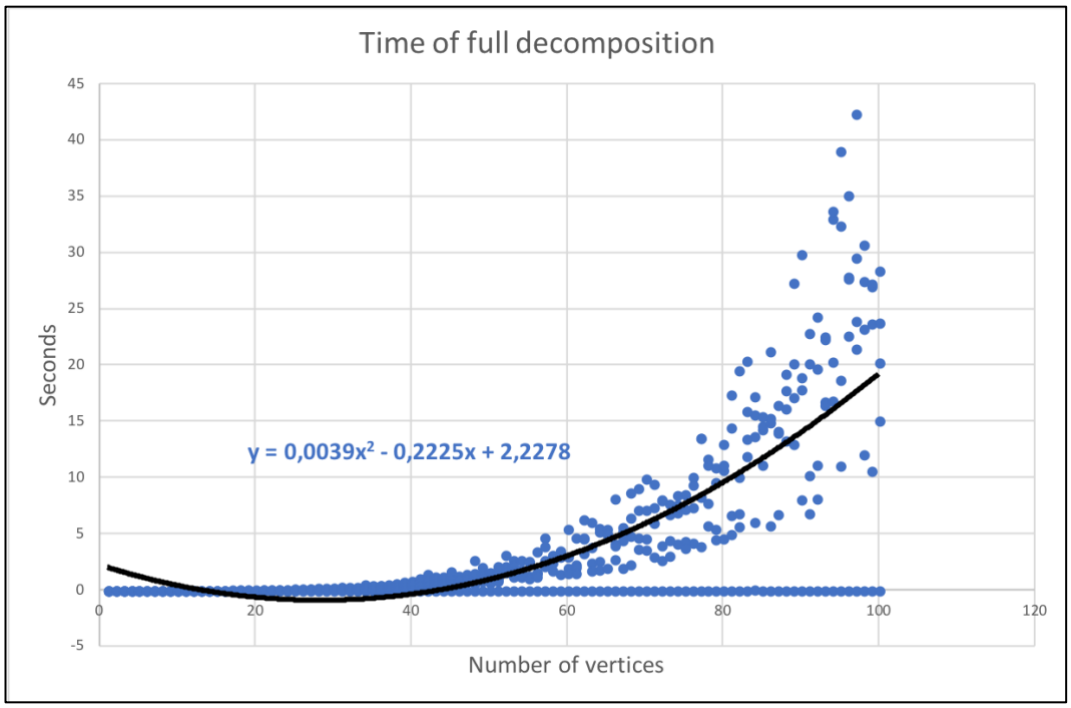


Figure 6: *performance data of "get_communities"*

To test the performance of the algorithms, they were executed on generated graphs of various sizes with a uniform probability of connection. As indicated by Figure 5, the time to calculate the community decomposition with the "get_communities" algorithm scales nearly quadratically with respect to the number of vertices. Figure 6 shows that the execution time of the "flow_free_vertex_cut" algorithm also scales quadratically. Compared to the existing local cut algorithm of the NetworkX library, however, the constant factors of the fitted curve are much lower, which allows the algorithm to process the graphs ten to a hundred times faster.

These observations lead to the second and final conclusion, answering research question 2:

C2 The execution time of the "get_communities" algorithm for searching communities that satisfy the definition, scales quadratically with respect to the number of vertices in the input graph.

Conclusion

In this paper, relative vertex connectivity is proposed as a scale-independent definition of communities. This definition gives the best guaranties of all the analyzed clique relaxations on a number of basic clique properties. An algorithm is given for finding communities with this definition based on the algorithm of Moody & White [3] for finding vertex connected sets. In this algorithm a flow-free vertex cut algorithm is used as a fast heuristic. The speed in the latter algorithm is faster than currently implemented vertex cut algorithms. The execution time of the total algorithm scales quadratically with respect to the number of vertices in the input graph.

Notes

The source code can be found at:
https://github.com/MCMXCVII/scale-independent-communities-flow-free-vertex-cut

References

[1] S. Fortunato, "Community detection in graphs," *Physics reports,* pp. 75-174, 2010.

[2] J. Pattillo, N. Youssef and S. Butenko, "On clique relaxation models in network analysis," *European Journal of Operational Research,* vol. 226, no. 1, pp. 9-18, 2013.

[3] J. Moody and D. R. White, "Structural Cohesion and Embeddedness: A Hierarchical Concept of Social Groups," *American Sociological Review,* vol. 68, no. 1, pp. 103-127, 2003.

[4] L. R. F. Jr. and D. R. Fulkerson, "Maximal flow through a network," *Journal canadien de mathématiques,* vol. 8, no. 0, p. 399–404, 1956.

[5] D. White and F. Harary, "The cohesiveness of blocks in social networks: Node connectivity and conditional density," *Sociological Methodology,* p. 305–359, 2001.