Enumerating Communities for a Deeper Understanding of Community Finding

Zachary Kurmas, Hugh McGuire, Jerry Scripps, and Christian Trefftz
Grand Valley State University
Allendale MI
USA
(kurmasz,mcguireh,scrippsj,trefftzc)@gvsu.edu

***Abstract*—**

Often new insights and advancements are made by a detailed study of the problem and the solution space. The area of community finding has had many algorithms proposed recently, but to our knowledge there have not been any detailed studies of the solution space. In this paper, we present two algorithms for enumerating and unranking the possible valid community assignments for a network. To demonstrate the value of our algorithms, we also present some interesting insights gained by examining the solution space of some small networks.

INTRODUCTION

Community finding algorithms take a network as input and return a set of communities. Generally, the goal is a community set with many intra-community links and few inter-community links. However, each algorithm tends to use its own objective function.

A convenient way to represent the resulting set of k communities for a graph of n nodes is with a $0/1$, $n \times k$ matrix C, where C_{ij} is 1 if node i is in community j and 0 otherwise. The number of possible states for C is 2^{nk}. Even though some values of C are not valid community assignments, it will suffice for now as a crude approximation of the number of possible community assignments in a network. Considering a small network such as karate ($n = 34$), the number of community assignments with $k = 2$ is a staggering 2.9×10^{20}!

A number of metrics have been proposed to measure the quality of a community assignment. The solution space using any of these metrics has the familiar landscape of local minimums and maximums. Many approximate solutions have been presented; however, there does not appear to be a clear method to compare them. We have not seen any assurances that an algorithm falls within a restricted bounds proximate to the global optimum. It appears that there is more work to be done in this area.

Often, advances and new insights are revealed from a detailed examination of the problem and its solution space. It is the purpose of this paper to present tools for counting and enumerating community assignment solutions and mapping rank numbers to these solutions. We are not presenting a new community finding algorithm, but rather a set of tools to aid in developing and refining community finding algorithms. Using the tools, developers and researchers can get a better feel for the solution space.

The contributions of this paper include the following. For a given size (n) of network and a desired number (k) of communities,

1) an algorithm for enumerating all of the possible community assignments for a given network size (n) and desired number (k) of communities, and
2) an algorithm to provide an unranking algorithm for generating the jth community assignment in an enumeration.

A ranking function is a mapping (or ranking) of valid assignment to a number. The unranking function we present does the reverse, that is, it returns the community assignment for a particular number. This can be used for not only enumerating all of the valid assignments but it can also be very useful for sampling. For anything larger than a very small network, enumerating all of the possible assignments is not feasible. However, using the unranking function makes it straightforward to randomly generate a large number of communities.

We invite researchers in the area of community finding to use these algorithms to enhance their knowledge of the domain and further the state of the art. We offer some experiments to illustrate their usefulness. First, the community assignment metrics have a well-behaved distribution. Like a normal curve, there are very few good solutions, many in the middle, and few bad ones. Second, unlike the sum squared error (SSE) metric for clustering, some metrics for community sets follow a parabolic-like curve. This suggests that finding the ideal k value can be done computationally using hill-climbing or other such techniques.

We begin by describing the terms and metrics for assignments. The next two sections present the process of community counting and design of the algorithms for enumeration of assignments. After that, we show the results of the experiments followed by conclusions.

NOTATION, METRICS, AND RELATED WORK

A *network* $G = (V, E)$ is a closed system of *nodes* $V = \{v_1, \ldots, v_n\}$ connected by *links* $E \subset V \times V$. Nodes can also be grouped into *communities*, $c_i = \{v_j, \ldots v_m\}$, through a process called community finding. An *assignment* $S = \{G, C\}$ is a tuple where $C = \{c_1, \ldots, c_k\}$ is a collection

978-1-4799-4143-8/14 $31.00 © 2014 IEEE
DOI 10.1109/WI-IAT.2014.14

IEEE computer society
RIGHTSLINK

of k communities. In the next section we will represent an assignment as a group of strings. One can think of it as converting each column (community) of C into a string as seen in Figure 1.

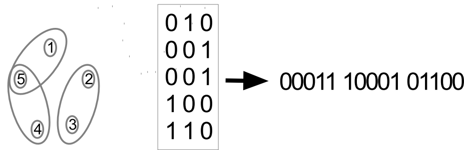


Fig. 1. Conversion of community structure to string format for $n = 5$ and $k = 3$

In related work, a number of community finding algorithms have been developed with a wide range of characteristics. Disjoint algorithms [1], [9], [3] have no overlap but typically optimize a metric such as modularity which tends to minimize edges between communities. There are agglomerative methods that have zero overlap and others that have zero between-community edges [11]. There are also those that find clique-like communities and minimize the non-edges within communities [7]. A variety of overlapping algorithms can be found in [12]; many of them optimize a variant of modularity.

Enumeration of Assignments

As discussed in the introduction, there are 2^{nk} ways of naively assigning nodes to communities. They may be represented bitwise, e.g. as follows for $n = 5$ & $k = 3$:

```
    bit patterns           ( decimals )
    ------------           ------------
00000 00000 00000          (  0  0  0 )
00000 00000 00001          (  0  0  1 )
.  .  .  .  .  .  .  .  .  (  .  .  . )
00110 10101 11000          (  6 21 24 )
.  .  .  .  .  .  .  .  .  (  .  .  . )
11111 11111 11110          ( 31 31 30 )
11111 11111 11111          ( 31 31 31 )
```

(The mapping above shows bit patterns and their decimal values.) Each line represents an assignment of nodes to communities. E.g. with the middle line above, each of the 3 bit patterns "00110", "10101", and "11000" describes a community, where each bit is "1" if the corresponding node is assigned to that community, or "0" otherwise. Naturally, these assignments can be enumerated straightforwardly using k integer counters ranging from 0 to $2^n - 1$, e.g. as indicated above for $n = 5$ & $k = 3$.[1]

However, some of the naive assignments of nodes to communities are unreasonable or redundant. We wish to exclude such undesirable assignments, thereby reducing the amount of assignments we process from 2^{nk} to a lower number. Specifically, we wish to have assignments that are:

1) non-empty (no empty communities)
2) unique (no duplicates)
3) comprehensive (every node is assigned to at least one community)

First, a trivial improvement to the basic enumeration algorithm indicated above is to exclude communities that are empty, e.g. "00000". We satisfy this requirement by starting counters at the bit pattern "...0001" instead of "...0000". This guarantees that all assignments will have no empty communities.

Second, we exclude duplicate assignments, e.g. "00110 10101 11000 (6 21 24)" and "11000 10101 00110 (21 6 24)"; and we also exclude assignments with multiple identical communities, e.g. "01101 01101 10110 (13 13 22)". We avoid such duplication by initializing each counter 1 higher than the counter to its left and limiting each counter to $2^n - 1$ minus the number of counters at its right. E.g., for $n = 5$ & $k = 2$ the three counter variables may be designated as `c2`, `c1`, and `c0` from left to right, and we make `c2` range from 1 to 29, we make `c1` range from `c2 + 1` to 30, and we make `c0` range from `c1 + 1` to 31. Thus, we generate the assignment "00110 10101 11000 (6 21 24)" but none of its permutations, nor any assignments with multiple identical communities.

Third, we guarantee only comprehensive assignments by excluding assignments for which some nodes are not included in any community. For example, suppose $n = 5$ & $k = 3$. With the comprehensive assignment "00110 10101 11000", each node is included in at least one community; by contrast, the assignment "00010 01000 01100" is not comprehensive because the first and last nodes are not included in any community. Observe that an assignment is comprehensive if and only if the bitwise OR of the values is the bit pattern comprising n "1"s, i.e. "111...111"; e.g. `00110 OR 10101 OR 11000` yields `11111`, whereas `00010 OR 01000 OR 01100` yields only `01110`. To achieve comprehensiveness, each time we modify the rightmost counter, we also set any needed bits in it if necessary. E.g. for $n = 5$ & $k = 3$, suppose the current assignment of counters `c2,c1,c0` is "00010 01000 10111 (2 8 23)". Simply incrementing the rightmost counter `c0` from `10111` to `11000` would produce the assignment `00010 01000 1100`, which is not comprehensive. We remedy this by setting additional bits in `c0` as follows:

1) We set a variable named "`need2`" to `11111`.
2) We set "`need1`" to `need2` with `c2`'s bits cleared:
```
need1 :=  need2 & ~c2
      :=  11111 & ~00010  := 11101
```
3) We set "`need0`" to `need1` with `c1`'s bits cleared:
```
need0 :=  need1 & ~c1
      :=  11101 & ~01000  := 10101
```
4) We set "`lack`" to `need0` with `c0`'s bits cleared:
```
lack :=  need0 & ~c0
     :=  10101 & ~11000  := 00101
```

[1]Keeping the counters separate makes this processing comparable to a classic automobile odometer. Each time one counter needs to 'roll around' from "...1111" to "...0000", the counter at its left needs to advance by 1.

RIGHTSLINK
Copyright Clearance Center

5) Since `lack` $\neq$ `00000`, we set `need0`'s bits in `c0`:[2]
```
c0 := need0 | c0
   := 10101 | 11000  := 11101 (29)
```

Thus, we obtain the comprehensive assignment "`00010 01000 11101 (2 8 29)`".

One issue is that using this algorithm to address comprehensiveness whenever `c0` is re-initialized to `c1+1` may skip some assignments. E.g. for $n = 5$ & $k = 3$, suppose the counters `c2,c1,c0` are "`00010 10000 11111 (2 16 31)`". To advance, `c1` increments to `10001 (17)`, `c0` restarts at `c1 + 1` which is `10010 (18)`, and then `c0` needs the bits in `need0 = 01100`, so `c0` might be `11110 (30)` and the resulting assignment might be "`00010 10001 11110 (2 17 30)`". But that would skip the assignments "`00010 10001 11100 (2 17 28)`" and "`00010 10001 11101 (2 17 29)`"! To address this issue, before we set needed bits in `c0`, we clear some low-order bit positions as follows:

1) Considering the value of `lack = 01100`, we set a variable named "`mask`" to its low-order bit positions, `00111`.[3]
2) Then, we actually set `need0`'s bits in `c0` as follows:
```
c0 := need0 | (~mask & c0)
   := 01100 | (~00111 & 10010)
   := 11100 (28)
```

Thus, we obtain the assignment "`00010 10001 11100 (2 17 28)`".

The code for our enumeration algorithm can be found in the Appendix.

As mentioned above, a goal here is to generate fewer than all the 2^{nk} naive assignments. This enumeration generates no more than $(2^k - 1)^n / k!$ assignments. This should be clear because for each of the n nodes there are naively 2^k assignments, but this enumeration excludes the one assignment where the node is not in any community; and this enumeration includes only one of each group of $k!$ assignments that are permutations. An exact formula for the number of assignments enumerated here is in [5].

Unranking

The previous section presents an algorithm for iterating over all community sets in a given community set space. However, in order to generate the ith community set, this algorithm must also generate the first $i - 1$ community sets. We present here an *unranking* algorithm that can directly and efficiently generate the ith community set in a given community set space.

The days of the year can be easily ranked by assigning a number to each day, starting with January 1. The rank for 1 February would be 32. The process to unrank would be to find the date given the number. Unranking 205 (in a non-leap year), for example would mean first finding out which month 205 is in (July). Since the rank of 1 July is 182, day 205 is 23 July, because 205 - 182 = 23. In this example, unranking is non-trivial because of the different number of days in the different months.

Like the date example, we break up the assignments into subranks. We will describe how the assignments are ordered within the subranks, provide methods of counting the assignments in the subranks, and describe how to find a particular assignment within a subrank.

Ordering

We based our choice of subranks on a process for building an assignment. We will show that this process is valid (no empty communities, no orphan nodes) and exhaustive (every assignment can be found from a unique rank). The process of constructing an assignment is iterative: At each step, we construct an assignment for nodes 1 through $n - 1$, then add node n to one or more of those assignments. Our unranking algorithm is used to specify (1) which assignment of nodes 1 through $n - 1$ to choose (the "Base" assignment) and which communities to which n should be added.

One could naively simply add n to a combination of the communities of B. However, that would miss "difference" community pairs. Consider a community $c_i \in B$; there would be many valid assignments that include both c_i and $c_i \cup n$ but the naive approach would miss these.

In our process, we start with an empty assignment C and add communities to it. To build the communities, we select a base assignment B for nodes 1 to $n - 1$. B can have k or fewer communities. Our algorithm uses assignments from $\frac{k}{2}$ to k. From B some of the communities are selected to be difference communities, D. For each $d_i \in D$, we add both d_i and $d_i \cup j$ to C. The remaining communities $B - D$, can be added to C either with or without n.

A couple of artifacts must be addressed. First, this approach does not generate assignments that contain the set $\{n\}$ by itself. This is fixed by splitting up the process into two parts where the assignments without $\{n\}$ are generated and then the ones with $\{n\}$ are generated using the same ordering. Second, assure that we end up with the right number of communities k, we need to select base sets using $k - i$ for $1 \leq i \leq \frac{k}{2}$ followed by selecting i difference sets.

The assignments are ordered as follows:

1) subrank1: first all sets that do not include $\{n\}$ then those that do
2) subrank2: all base sets B for $\frac{k}{2}$ communities, up to k communities
3) subrank3: within subrank2, base sets are enumerated recursively
4) subrank4: for a particular B, the difference sets D are enumerated lexicographically
5) subrank5: the remaining communities, $B - D$ are appended with n according to a high-to-low order permutation

[2] While clearly `lack` indicates whether we need to set bits in `c0`, we use `need0` to actually set the bits because we clear some bits in `c0`, as discussed in the next paragraph.

[3] We do this efficiently by shifting `need0` down one and then repeatedly doubling its bits.

RIGHTSLINK
Copyright Clearance Center

In mathematical terms, an assignment is a k-cover of the set $\{1, ..., n\}$ (also called a "k-cover of n"). For the rest of this section we refer to a community assignment as a k-cover.

Counting

Like with the date example where it is important to know the days in a month, we have to have some expressions that are used to count the subranks. For the rest of this section, we use the notation, $f(n, k)$ to be the number of k-covers of n and $f_-(n, k)$ to be the number of k-covers of n that do not include the set $\{n\}$.

Because each independent choice generates a different k-cover of n (see Theorem 2), the number of k-covers of n that do not contain the set $\{n\}$ is

$$f_-(n, k) = \sum_{i=0}^{\frac{k}{2}} f(n-1, k-1)\binom{k-i}{i}(2^{k-2i} - z) \quad (1)$$

where $z = 1$ when i is 0, and $z = 0$ otherwise. [4]

If we bring the case where $i = 0$ outside the summation, Equation 1 simplifies to

$$f_-(n, k) = f(n-1, k)(2^k-1)+\sum_{i=1}^{\frac{k}{2}} f(n-1, k-1)\binom{k-i}{i}2^{k-2i} \quad (2)$$

where:

- $f(n-1, k-1)$ represents the number of different base sets for a given i.
- $\binom{k-i}{i}$ is the number of choices for D given a specific base set B.
- 2^{k-2i} is the number of different ways to choose which of the $k - 2^i$ sets in $B - D$ receive then additional n.

For the k-covers of n that do include $\{n\}$, let C be a k-cover of n that contains the set $\{$n$\}$. and let C' be C - $\{$n$\}$. If n appears somewhere in C', then C' is a $k - 1$-cover of n. If n does not appear anywhere in C', then C' is a $k - 1$-cover of $n - 1$. Thus,

$$f(n, k) = f_-(n, k) + f(n, k-1) + f_-(n-1, k-1) \quad (3)$$

Unranking algorithm

To "unrank" a k-cover means to calculate which cover is at position $rank$ in the list. We do this by considering each choice in order. If $rank < f_-(n, k)$, we generate a k-cover that does not contain $\{n\}$. Otherwise, we generate a k-cover that does contain $\{n\}$.

[4] This accounts for the fact that when $i = 0$, D is empty and so one of the assignments generated would not have n which would be invalid, and so is removed from the count.

The next step is to choose the number of difference pairs (which we call i). Define $g(n, k, i)$ to be the number of k-covers of n that contain exactly i difference pairs and do not include $\{$n$\}$. Our development of Equation 2 shows that

$$g(n, k, i) = f(n-1, k-1)\binom{k-i}{i}2^{k-2i}$$

when $i >= 1$ and

$$f_+(n-1, k)2^k - 1$$

when $i = 0$.

Because we order covers by the number of difference pairs, we need to determine the value of i for which

$$\sum_{j=0}^{i-1} g(n, k, j) < rank <= \sum_{j=0}^{i} g(n, k, j)$$

The algorithm to compute i is analogous to the algorithm one would use to determine the month in which a given ordinal date occurs.

Once we have determined i, we must determine which of the g covers corresponds to $rank$. Thus, we set $remainder = rank - g(n, k, i)$ and chose our base set B based on $remainder$.

Each base set B can serve as the basis for $\binom{k-i}{i}2^{k-2i}$ covers. Thus, we set $basis_index = \frac{remainder}{\binom{k-i}{i}2^{k-2i}}$ and recursively call $unrank(n-1, k, basis_index)$ to obtain B.

Once we have selected B, $remainder2 = remainder \bmod \binom{k-i}{i}2^{k-2i}$ specifies both D and sets in $B - D$ to which we add $\{n\}$. Given i, B, and D, we can generate 2^{k-2i} different covers. Thus, $d_index = \frac{remainder2}{2^{k-2i}}$ and $sets_with_n = remainder2 \bmod 2^{k-2i}$.

The set D represents one of the $\binom{|B|}{|D|}$ ways to choose $|D|$ sets from B. We use Stolee's unranking algorithm to choose D given d_index [10].

Once we have made all four choices, we build our k-cover (which we call C) as follows:

1) For each $D_x \in D$, adding both D_x and $D_x + \{n\}$ to C.
2) For each $E_x \in B - D$, adding E_x to C if the xth bit of $sets_with_n$ is 1.
3) Adding $\{n\}$ to C, if $rank >= without_n$.

Correctness

The algorithm described above takes a number $rank$ as input and returns a community assignment C. We show here that C is a valid assignment and that each unique x will produce a unique assignment C.

Theorem 1. *The resulting set C is a valid k-cover of n*

Proof: First, we show that $|C| = k$. We generate C by taking the union of D, a "copy" of D with n added to each

RIGHTSLINK
Copyright Clearance Center

set, and a "copy" of $B - D$ with n added some of the sets. Therefore, $|C| = 2|D| + |B - D|$. We defined B and D such that $|D| = i$ and $|B| = k - i$; thus, $|C| = 2|D| + |B| - |D| = 2i + (k - i) - i = k$ as desired.

Now we show that, $\cup_{C_x \in C} = \{1, ..., n\}$. Because B is a valid covering of $n - 1$, we know that each $1 \leq j \leq n - 1$ appears in some $B_x \in B$. For each $B_x \in B$, either B_x or $B_x + \{n\}$ appears in C; thus, C is a valid covering of $n - 1$. Furthermore, we know that n appears in at least one set of C: If $i > 0$, then C contains $D_x + \{n\}$ for some $D_x \in D$. If $i = 0$, then we specifically assure that C contains $E_x + \{n\}$ for some $E_x \in B - D$ Hence C is a valid cover of n. ■

This algorithm to generate C relies on three choices:

1) The choice of B (the base $k - i$-cover of $n - 1$).
2) The choice of which i sets in B comprise D.
3) The choice of sets in $B - D$ to which we add n.

For each different choice of input $0 < rank < max$ where max is the number of possible community assignments, a unique set of choices will be selected. The following proves that for each unique set of choices, a unique k-cover of n will be built. This is sufficient to show that every possible assignment is mapped to a single rank.

Theorem 2. *Each unique set of choices produces a unique k-cover C.*

Proof: Let C_1 be a k-cover of n generated by base cover B_1 and difference sets D_1. Likewise, let C_2 be defined by B_2 and D_2. We will show that if either $B_1 \neq B_2$ or $D_1 \neq D_2$, then $C_1 \neq C_2$. Notice that every set $C_x \in C_1$ is either an element of B_1 or there exists a $B_x \in B_1$ such that $C_x = B_x + \{n\}$ (these are the only sets our algorithm adds to C_1).

If B_1 and B_2 differ, then so do C_1 and C_2. Without loss of generality, assume $|B_2| \geq |B_1|$. This means that there is some set $X \in B_2$ that is not in B_1. The resulting k-cover C_2 will contain either X or $X + \{n\}$. Assume to the contrary that either $X \in C_1$ or $X + \{n\} \in C_1$. We can then assume $X \in B_1$, which contradicts our assumption that $X \in B_2 - B_1$. Thus, we know $C_1 \neq C_2$.

We now show that if $D_1 \neq D_2$, then $C_1 \neq C_2$. We will assume that $B_1 = B_2$. Without loss of generality, assume $|D_2| \geq |D_1|$ and let X be in D_2 but not D_1. Both X and $X + \{n\}$ will appear C_2; however, only one of X or $X + \{n\}$ will appear in C_1. Thus, $C_1 \neq C_2$. ■

EXPERIMENTS

In this section we use the algorithms that were developed in the preceding sections to investigate the characteristics of assignments by exhaustive search. Our experiments uncovered two interesting insights. First, we found that metric scores for assignments appear to have a well-behaved distribution. This is helpful because it reinforces our intuition that good community assignments are very uncommon and difficult to find. Second, the optimum k curve for most graphs appears to have a parabolic shape. This differs from the typical SSE curve used in clustering where the optimum k value has to be selected by visually inspecting the curve for a "knee". The parabolic curve allows analysts to use a automated process such as hill climbing to find the optimum number of communities.

Both sets of experiments used metrics to measure the quality of the assignments. There have been many such metrics proposed. For our experiments we chose two dissimilar metrics. First we used the violations from the community set space [8]. A concept of *home community*, a particular community to which a node is assigned is used. In the experiments, violations refer to the sum of *missing neighbors* (those neighbors of a node that do not appear in the node's home community), *extraneous nodes* (nodes not directly linked to a node within its home community), and *overlap* (the number of communities that a node is placed in besides its home community). The second metric is the overlapping modularity metric proposed by Shen[12]. It is based on Newman's[4] well-known modularity metric but allows us to examine overlapping assignments.

Distribution of Sets.

In a typical community finding algorithm, a specific graph is given, a set of communities are formed and the metric (violations or modularity) is calculated. In this set of experiments we look at the distribution of metrics across the range of possible community assignments.

Rather than doing this for a single graph and then generalizing it to all graphs, we chose to look at all graphs of a certain size and then generate all communities for each one. In this way we can get a broader picture of the space. The challenge is that the numbers get very large very quickly. For $n = 7$, there are 9×10^{10} different assignments and $1,044$ possible graphs. For $n = 8$ there are 4×10^{14} different assignments and $12,346$ possible graphs. The largest graph size feasible for the analysis was $n = 7$.

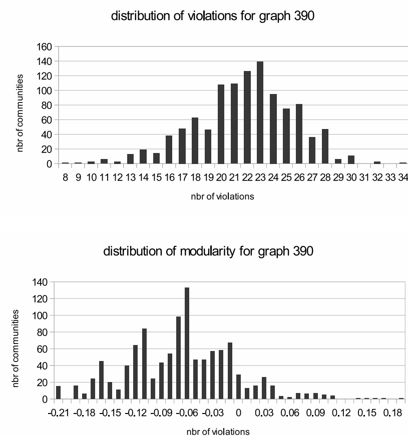


Fig. 2. distribution histograms for graph 390

For each of the 1044 graphs, we calculated the violations for each of the possible assignments (for $k = 2, 3, 4$ and 5) and then summarized the results in the histograms. Figures 2 and 3 show results for two randomly selected graphs. A number of observations can be made. First, the histograms appear to follow a distribution, perhaps normal. Second, typical of such distributions, very few of the assignments have low violations

RIGHTSLINK

TABLE I. PERCENTAGE OF GRAPHS THAT PASSED THE χ^2 TEST AT DIFFERENT VALUES OF k.

k	violations	modularity
2	90	67
3	74	3
4	88	0
5	94	0

– i.e., the assignments that most community finding algorithms are searching for represent a very small percentage of the total possible sets. This reinforces the difficulty of the community finding problem.

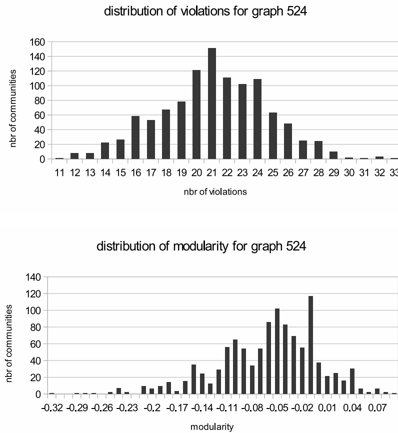
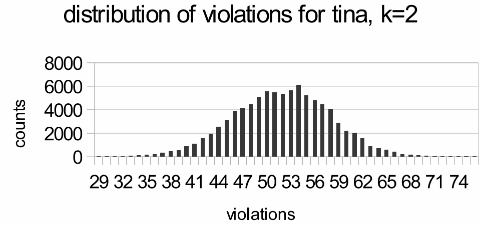


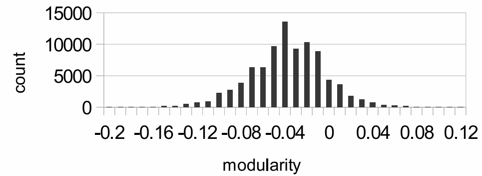
Fig. 3. distribution histograms for graph 524

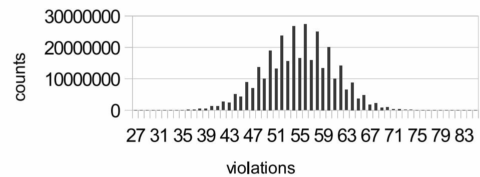
To find out if the violations from all possible assignments follow a normal distribution we applied the χ^2 test. The experiments are summarized in Table I. A large number of these small 7 node networks appear to follow a normal distribution when we use violations. For modularity the percentage is high for $k = 2$ but then drops off dramatically. We can conclude that modularity is not distributed normally but there still seems to be a bell-shaped distribution. The modularity histograms in Figures 2 and 3 are typical of the graphs in general. While the histograms for violations had a more symmetric curve, the modulation histograms were more skewed.

These examples are fairly choppy histograms. To get a perspective of slightly larger networks we also plotted the histogram for the Tina [6] data set. A small network with 11 nodes, it still generates 88572 assignments with $k = 2$ and over 329 million assignments with $k = 3$. The histograms can be seen in Figure 4. For both violations and modularity, for $k = 2$ the curves appear smoother than the 7 node graphs. For $k = 3$, the curves are even smoother (the spikes in the violation histogram is explained by the binning of the floating point modularity scores). The violation histograms are both symmetric while the two modularity histograms are a bit skewed which supports the earlier findings with the 7 node graphs. While these few experiments are not conclusive, they do lend support to our conjecture that the metrics for community quality follow a bell-shaped distribution.

In the χ^2 experiment above we showed evidence to support the claim that the violations for the assignments of most of the







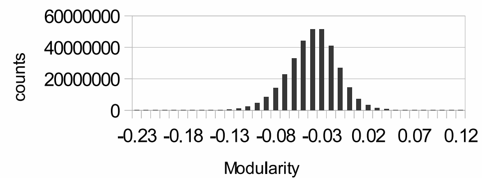


Fig. 4. Distribution of communities for Tina using violations and modularity for both $k = 2$ and $k = 3$

violation graphs followed a normal distribution. We would like now to focus on the graphs that failed the χ^2 test. Rather than look at every non-normal graph, we show two representatives: graphs 342 and 872. Looking at Figure 5, the histograms appear to have a nearly normal, but skewed distribution. Graph 342 has a long tail to the right and 872 has a long tail to the left. Looking at the actual graphs we see that 342 is bipartite and graph 872 is two nearly connected components. Inspection of other non-normal graphs were similar. Recall that with violation histograms, assignments that are plotted to the left have fewer violations. So it appears that graphs that graphs that lend themselves to nearly perfect "good" assignments will have longer left tail distributions, in other words, more good assignments. And those with that lend themselves to nearly perfect "bad" assignments will have more poor assignments.

RIGHTSLINK

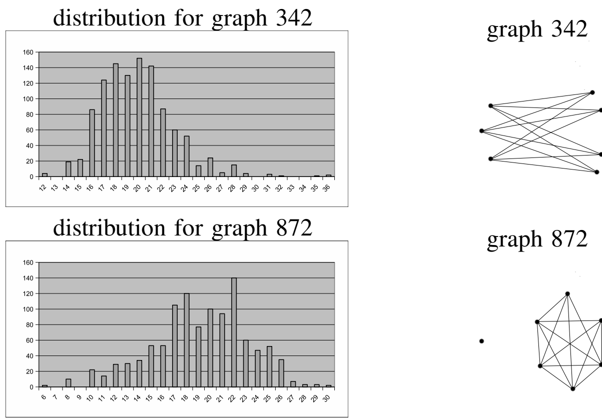


Fig. 5. Selected graphs on non-normal distributions.

Distribution of Minimum Violations for k.

Community finding algorithms can be separated into those where the parameter k is specified and those where it is not. In the latter group, the algorithm determines the best k for the optimum assignment. The algorithms in the former group then must determine the best assignment by exhaustively trying each value of k. To find a more efficient way to find the k with the minimum violations we examined the distribution of violations for each value of k.

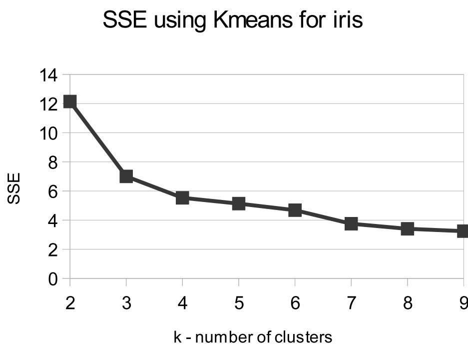


Fig. 6. Typical SSE curve for finding the optimum k value

Clustering has a similar problem. The popular Kmeans clustering algorithm finds a good set of clusters when given a k value. It is based on the sum squared error (SSE) function. When applying Kmeans to a set with multiple values of k, SSE has a curve that is highest when $k = 1$, falls quickly and then tails off until it becomes zero at $k = n$, as seen for the Iris data set in Figure 6. To find the optimum k, analysts can visually inspect the graph of SSE and choose a point where the curve changes dramatically (the knee). In the case of Iris, it is at $k = 3$.

We are interested whether the metrics for community finding behave similarly. Note that the minimum violation curve will be different for different graphs. First, consider a graph that is one large clique. Clearly this graph will have a minimum number of violations with $k = 1$ (one large community) and will get worse as k grows as seen in Figure 7(a). Next, consider a graph with no edges. This graph will have the minimum violations when $k = n$, (all singletons) and grows as k decreases as in Figure 7(b).

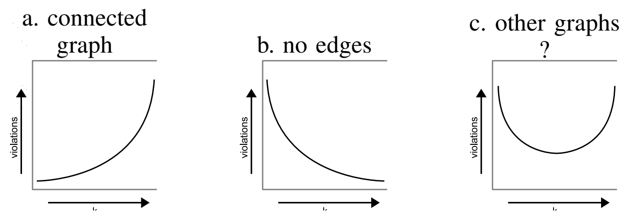


Fig. 7. Hypothesis of minimum violation curve for k for different types of graphs.

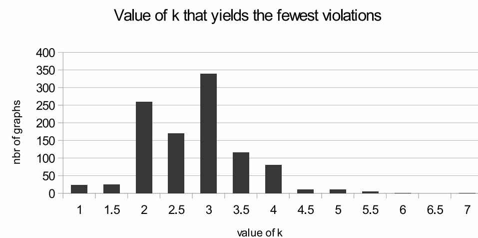


Fig. 8. Finding the best k. This chart shows the totals for each value of k of all the graphs where the assignment that had the lowest violations was equal to k. The half numbers are explained by the fact that some graphs had their lowest assignments for two values of k so the average was taken.

For all the other graphs, that are not cliques and have edges, we hypothesize that the minimum violation curve will look something like the one in Figure 7(c). It may be skewed one way or the other but the absolute minimum will be between 1 and n. To test this hypothesis, we ran tests on the graphs, finding the assignment or sets that had the lowest violations and keeping track of its k value (the number of communities used) In cases of ties between two values of k, an average was taken.

The results of the tests can be seen in Figure 8. Notice that for most of the graphs the number of communities for the lowest violation set was between 1 and n. There were a number of graphs that had minimums at $k = 1$, however these graphs were all highly connected so we suspect that in most real social networks the minimum violation curve will be U shaped with a minimum between 1 and n.

This will make finding the best value of k a bit more efficient for algorithms that require k to be specified. Analysts can use a hill-climbing technique, repeatedly calling the algorithm with k values to quickly and accurately find the optimum value of k.

Conclusions

The purpose of this paper is to further the study of community finding. When one actually counts the number of potential solutions to the community finding problem the

RIGHTSLINK Copyright Clearance Center

number is staggeringly large. However, it seems important to study this solution space even if for small graphs.

Generating an exhaustive list of community assignments is itself a difficult problem. We presented an enumeration algorithm that efficiently lists all community assignments for a given n and k, generating only assignments that are non-empty, unique, and comprehensive. In addition, we also provided an unranking algorithm that researchers can use for sampling the solution space for a particular graph.

As a way to demonstrate the usefulness of these algorithms, we ran experiments that showed that the community quality metrics are distributed in a bell-like curve over the range of possible solutions. We also showed that curve for the optimum metric value across all k is shaped like a parabola, meaning finding the optimum k can be done using a traditional hill-climbing approach.

Appendix

Our code is as follows. The parameter "`LIMIT`" is the bit pattern comprising n "1"s, i.e. `111...111`. For the community-representing counters `...,c2,c1,c0`, we use a k-long array named "`c[]`" of integers which have at least $n+1$ bits.[5] We initialize `c[]` to contain the bit patterns "`{...0001,...0010,...0011,...0100,...}`", i.e. `{1,2,3,4,...}` (from the highest index `k-1` down to the lowest index `0`). Similarly, for the variables `...,need2,need1,need0` we use an array named "`need[]`" which we set as discussed above. Then, we perform repetition as shown below. The variable `e` is available if desired for numbering the assignments we enumerate. The repetition stops when advancing would make `c[k-1]` exceed its limit discussed above, `LIMIT - (k - 1)`.

```
for ( e = 1 ;   ;  e++  ) {
  // if nec. modify c[0] so comprehensive:
  if ( (lack = need[0] & ~c[0]) != 0x0 ) {
    // use mask for unnecessary bits:
    for ( mask = lack >> 1, s = 1 ;
          ((mask + 1) & mask) != 0x0 ;
          s <<= 1 )
      mask |= mask >> s;
    c[0] = need[0] | (~mask & c[0]);
    }

  <OUTPUT OR PROCESS ASSIGNMENT IN c[]>

  // advance/'roll around' counters:
  for ( i = 0 ;  i < k ;  i++ )
    if ( c[i] < LIMIT - i )
      break;
  if ( i == k )
    break;      // all done
  c[i]++;
  while ( --i >= 0 ) {
    c[i] = c[i + 1] + 1;
    need[i] = need[i + 1] & ~c[i + 1];
    }
  }
```

[5] For unlimited values of n, we could use something like `BigInteger` in Java or a custom data structure (possibly based on `bitset`) in C++/C.

References

[1] A. Clauset, M. E. J.Newman, and C. Moore. Finding community structure in very large networks. In *Statistical Mechanics*, 2004.

[2] A. Lancichinetti and S. Fortunato. Community detection algorithms: a comparative analysis. *Physical Review E*, 80, Sep 2009.

[3] M. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69, Feb 2004.

[4] The on-line encyclopedia of integer sequences. http://oeis.org/A055154.

[5] Pajek datasets. http://vlado.fmf.uni-lj.si/pub/networks/pajek/default.htm.

[6] G. Palla, I. Dernyi, I. Farkas, and T. Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435, Jun 2005.

[7] J. Scripps. Exploring the community set space. In *IEEE/WIC/ACM International Conference on Web Intelligence*, 2011.

[8] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions On Pattern Analysis And Machine Intelligence*, 22(8), August 2000.

[9] Derrick Stolee. Combinatorics using computational methods. 2012.

[10] Lei Tang, Xufei Wang, Huan Liu, and Lei Wang. A multi-resolution approach to learning with overlapping communities. In *KDD Workshop on Social Media Analytics*, 2010.

[11] J. Xie, S. Kelly, and B. Szymanski. Overlapping community detection in networks: the state of the art and comparative study. *CoRR*, abs/1110.5813, 2011.

RIGHTSLINK
Copyright Clearance Center