Improved algorithm and data structures for modularity analysis of large networks

Alexandre P. Francisco **Arlindo L. Oliveira**

INESC-ID / IST, Technical University of Lisbon

{aplf,aml}@inesc-id.pt

Abstract

Graph clustering is an important problem in the analysis of computer networks, social networks, biological networks and many other natural and artificial networks. These networks are in general very large and, thus, finding hidden structures and functional modules is a very hard task. In this paper we propose new data structures and make available a new implementation of a well known agglomerative greedy algorithm to find community structure in large networks. The experimental results show that the improved data structures speedup the method by a large factor, for very large networks.

1 Introduction

In the study of many networks, such as computer networks, social networks, or biological networks, it is frequently useful to identify communities. Informally, communities may be defined as unexpected densely connected sets of vertices. By identifying community structure we are able to capture common properties among vertices, e.g., common function in biological networks or common semantic attributes in social and web networks. Community structure identification can also help the study of huge networks, for which a layered analysis may be useful if we want to study different levels of relations. For a detailed discussion of motivations, see for instance [1].

Given the importance of community structure identification, many work has been done in computer science, physics, biology, economics, and sociology [1, 2, 3]. Main research directions include global, local, overlapping and non-overlapping community identification. An approach to community finding is graph partitioning or clustering, where each vertex belongs to exactly one set. Many approaches have been developed to tackle the problem of finding partitions in graphs [1, 3, 4, 5, 6, 7]. These approaches may differ in many aspects, such as hierarchical and non-hierarchical methods and whether *a priori* knowledge of the size or number of communities is required.

A well known and natural approach was recently proposed by Newman and Grivan [4], which consists of finding communities by maximizing the *modularity* score. The modularity of a given network partition is given as the difference between the number of edges inside partitions, and the expected number of such edges if randomly placed while respecting the vertices degrees. Although the problem of modularity maximization is NP-hard [8], many heuristic algorithms have been developed and recent empiric results show that this approach identifies interesting community structure in real networks [1, 2, 4, 5, 6]. However, we should be careful with modularity maximization results, Fortunato and Barthélemy [9] and Kumpula *et al.* [10] recently noted that modularity suffers some problems, such as resolution limit. Thus, depending on the analysis, we may consider alternative scores such as the similarity-based modularity [11]. Later we will discuss how the algorithm described in this paper can be extended to different measures.

The results reported in this paper are motivated by the emergent interest in finding communities in very large networks with thousands of vertices. Regarding large networks and modularity maximization, Newman [5] has proposed an algorithm based on the greedy optimization of the modularity. It runs in $O((n+m)n)$, or $O(n^2)$ for sparse graphs, where n is the number of vertices and m is the number of edges. The running time of this algorithm can be improved by exploiting some properties of the optimization problem and using more sophisticated data structures. Specifically, Clauset *et al.* [6] proposed a greedy algorithm which runs in $O(md \log n)$, where d is the depth of the "dendrogram" which describes the community structure. On sparse graphs with a hierarchical community structure their algorithm runs on average in $O(n \log^2 n)$ time. In what follows, we refer to this algorithm as the *CNM (Clauset-Newman-Moore) algorithm*.

In this paper we propose and make available[1] a new implementation of the CNM algorithm, using improved data structures. Both analytical analysis and experimental results show that the improved data structures speedup the method by at least a factor of two. The algorithm is an agglomerative greedy algorithm with $O(n^2 \log n)$ running time and $O(n+m)$ required space for networks with n vertices and m edges. We also show that its running time is $O(n \log^2 n)$ for networks that are both sparse and hierarchical.

2 Graph modularity

The concept of modularity is central to this problem [4]. Modularity is a property of the graph and of a specific division of the graph into communities. It measures the quality of the division by evaluating the number of edges within communities and the number of edges that connect vertices in different communities. Let $G = (V, E)$ be an undirected graph. Suppose the vertices are divided into k communities and let $1 \leq c_u \leq k$ denote the community where vertex $u \in V$ belongs. The *adjacency matrix* A of G is such that

$$A_{uv} = \begin{cases} 1 & \text{if } (u,v) \in E, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

The *degree* d_u of a vertex $u \in V$ is defined as the number of edges incident upon it, i.e.,

$$d_u = \sum_{v \in V} A_{uv}. \tag{2}$$

We define the *modularity* Q of G with respect to the given partition as

$$Q = \frac{1}{2m} \sum_{u,v \in V} \left[A_{uv} - \frac{d_u d_v}{2m} \right] \delta(c_u, c_v), \tag{3}$$

where $m = |E|$ and the δ-function is such that $\delta(i,j) = 1$ if $i = j$ and $\delta(i,j) = 0$ otherwise. We note that the above sum runs over all possible pairs of vertices. Therefore, each edge is summed twice. If we split the sum in two terms, the first term

$$\frac{1}{2m} \sum_{u,v \in V} A_{uv} \delta(c_u, c_v) \tag{4}$$

is the fraction of edges that fall within the communities, and the second term

$$\frac{1}{2m} \sum_{u,v \in V} \frac{d_u d_v}{2m} \delta(c_u, c_v) \tag{5}$$

is the expected fraction of edges within the communities, if the edges were randomly distributed while respecting the vertices degrees. In particular, if the edges were randomly placed as mentioned, $d_u d_v / m$ is the probability of the existence of an edge between vertices $u, v \in V$.

Thus, modularity measures the fraction of edges that connect vertices in the same component minus the expected value of the same quantity in a graph with the same components but random connections between the vertices [4]. Values near 1, the maximum value of Q, indicate strong community

[1] Available at `http://kdbio.inesc-id.pt/software/gcf/`.

```
struct adj_node {
    int id;
    int u;
    int v;
    struct adj_node *u_nxt;
    struct adj_node *u_prv;
    struct adj_node *v_prv;
    struct adj_node *v_nxt;
};
```

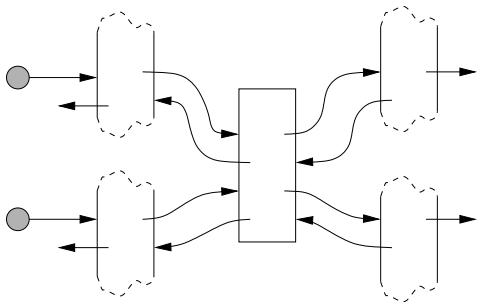

Figure 1: Cross-linked adjacency list data structure. List nodes are defined by the C structure on the left and are linked as depicted on the right, where a cross-link is shown.

structure. Typically, values for graphs with known community structure are in the range from 0.3 to 0.7 [4, 12].

The modularity Q_G of a graph G is defined as the maximum value that can be obtained for expression 3, over all possible graph partitions. Thus, although expression 3 can take negative values, the maximum modularity of a graph takes values between 0 and 1. For any graph, the value 0 is obtained for the trivial partition where all nodes belong to the same community. Given a graph G, finding the partition with maximum modularity Q_G is NP-hard [8].

3 Algorithm and data structures

The proposed algorithm starts with each vertex being the sole member of its community. Its operation consists in finding the pair of communities for which the merge will produce the maximum positive change in the modularity score Q. This is the same principle employed by the Newman's greedy algorithm and by the CNM algorithm.

Given two communities i and j, ΔQ_{ij} denotes the change in Q produced by the merging of i and j. Since calculating the ΔQ_{ij} for each pair i, j becomes time-consuming, we store these values for each pair and only update them when needed. Given a community i, let

$$A_i = \frac{1}{2m} \sum_{u \in V} d_u \delta(c_u, i). \tag{6}$$

If we are merging community i into community j, ΔQ_{jk} for each community k adjacent to j and A_j are updated as follows [6]:

$$\Delta Q_{jk} = \begin{cases} \Delta Q_{ik} + \Delta Q_{jk} & \text{if } k \text{ is connected to } i \text{ and } j, \\ \Delta Q_{ik} - 2A_jA_k & \text{if } k \text{ is connected to } i \text{ but not to } j, \\ \Delta Q_{jk} - 2A_iA_k & \text{if } k \text{ is connected to } j \text{ but not to } i; \end{cases} \tag{7}$$

$$A_j = A_j + A_i. \tag{8}$$

Communities are considered connected if there exists at least one edge between them. Note that merging two communities for which there is no connecting edge does not increase Q. Therefore, we do not care about such pairs and we will not store their ΔQ. Thus, we will have at most m values to store and we will store them in a heap data structure [13]. Since we have to perform both decrease and increase operations over keys in the heap, we use a simple binary heap data structure. Therefore, get minimum operation takes constant time in the worst case and insert, delete and update operations take $O(\log m)$ time in the worst case.

We also maintain the community adjacency list. This data structure is a common graph adjacency list [13], the adjacency list for each community being stored in a double-linked list. In order to improve the algorithm running time, we maintain cross references among adjacency lists, being able to solve side effects in constant time when merging two adjacency lists. In fact we share the adjacency list nodes as depicted in Figure 1.

As described above, the algorithm starts with each vertex $u \in V$ being the sole member of a community c_u. Thus, for each community i and for each edge (i, j) in the graph, we initially set

$$A_i = \frac{d_i}{2m} \qquad \text{and} \qquad \Delta Q_{ij} = \frac{1}{m} - 2A_iA_j. \tag{9}$$

The initial value of Q is set to

$$Q = -\sum_{u \in V} A_{c_u} A_{c_u}. \tag{10}$$

The algorithm proceeds as follows:

1. extract the edge (i, j) with maximum ΔQ in the heap;
2. if the edge (i, j) was removed from the community adjacency data structure, ignore it and go to step 1;
3. remove the edge (i, j) from the community adjacency data structure;
4. merge adjacency lists of communities i and j;
5. update ΔQ for each adjacency pair accordingly to equation 7;
6. update A_j accordingly to equation 8, assuming without loss of generality that j becomes the new community;
7. update modularity Q by adding ΔQ_{ij};
8. repeat from step 1 until one community remains.

Although similar, this algorithm differs from the CNM algorithm since it updates only one heap and, by using cross-linked adjacency lists, the merge step is simplified.

4 Complexity analysis

Let $n = |V|$ and $m = |E|$. The space requirement of the algorithm is simply $O(n + m)$. The community adjacency data structure stores at most m connections and we need to store the connections for each community. Thus, since there are n communities at most, we need $O(n)$ plus $O(m)$ space. The heap stores m elements, and therefore, requires $O(m)$ space.

Let us now examine the running time of the algorithm. We know that updating an element in the heap takes $c_u \log m$ time and that extracting an element from the heap takes $c_r \log m$, where m is the maximum size of the heap and $c_u, c_r > 0$ are constants. Thus, the extraction in step 1 takes $c_r \log m$ time at most and, since there are m elements in the heap, this step is repeated m times. Because we get a direct reference to the pair from step 1 and we are dealing with double-linked lists, removing the edge (i, j) from the community adjacency data structure in step 3 takes constant time. Step 4 takes $3c_l n$ time in the worst case, with $c_l > 0$ being a constant. Note that there are at most n adjacent communities to i and to j and that we can solve side effect changes in constant time because of the above mentioned cross references among adjacency lists. If a community k appears twice in the result, we only keep it once. To achieve linear time with unsorted lists, without loss of generality, we must process the adjacency list of i and build a bit array of size n at most. Then we process the adjacency of j, checking whenever a community k occurs in both adjacencies and updating the bit array. Finally we reprocess the adjacency of i in order to find the communities k which were not in the adjacency of j. Therefore, processing adjacencies takes at most $3c_l n$. Step 5 can and should be done along with step 4 and each update takes $c_u \log m$ time at most, thus step 5 takes less than $c_u n \log m$ time. Steps 6 and 7 are trivial and take constant time.

Although there exist m elements in the heap, steps 3-7 are executed at most $n - 1$ times because there are at most $n - 1$ merges. Therefore, the running time of the algorithm is at most $c_r m \log m + 3c_l n^2 + c_u n^2 \log m$, i.e., $O(n^2 \log n)$ time in the worst case assuming as usual that $m = O(n^2)$.

For sparse and hierarchical graphs we can provide a better upper bound. A graph $G = (V, E)$ is sparse if $m = O(n)$ and G is hierarchical if the resulting dendrogram for the community merging is balanced. In this case, the sum of the communities degrees at a given depth d is at most $2m$. Therefore the running time is $c_r m \log m + 6c_l dm + 2c_u dm \log m$, i.e., $O(md \log n)$, where d is the

depth of the dendrogram. Then for sparse and hierarchical graphs, $m = O(n)$ and $d = O(\log n)$, the algorithm running time becomes $O(n \log^2 n)$.

Although the runtime asymptotic bounds are equal for our algorithm and for the CNM algorithm, we note that there is an improvement of at least a factor of two. As before, the CNM algorithm employs the same greedy algorithm but it maintains different data structures. Specifically, it stores the ΔQ values in a sparse matrix with each row being stored both as a balanced binary tree and as a max-heap. It also maintains a max-heap containing the largest element of each row. Before discussing the running time, we note that we consider the same max-heap implementation as before. Therefore, updating an element takes $c_u \log n$ and extracting an element takes $c_r \log n$, where n is the maximum size of the heaps in this case. Thus, extraction in step 1 takes $c_r \log n$ time. Removing the selected pair from the community adjacency data structure in step 3 takes $2c_t \log n$ to update the binary trees plus $2c_u \log n$ to update the heaps, where $c_t > 0$ is a constant. Steps 4 and 5 take at most $2n(c_t + 2c_u) \log n + c_u n$ time, since we must update the trees, the k-heap and the main heap for each k in the adjacency lists being merged. The heap associated with the resulting adjacency list can be updated in $c_u n$ time [13]. As above, steps 6 and 7 are trivial and take constant time. Since steps 1-6 are executed at most $n - 1$ times, the running time of the CNM algorithm is at most $(c_r + 2c_t)n \log n + 2(c_t + 2c_u)n^2 \log n + c_u n^2$, i.e., $O(n^2 \log n))$. Taking as above d and m, the running time is $(c_r + 2c_t)n \log n + 4(c_t + 2c_u)dm \log n + 2c_u dm$, i.e., $O(md \log n))$. Thus, for sparse and hierarchical graphs the running time becomes $O(n \log^2 n)$.

Thus, although $\log n \leq \log m \leq 2 \log n$, by comparing the above running time expressions we can state that our implementation should achieve an improvement of at least a factor of two. We will confirm this fact within the experimental evaluation.

5 Experimental evaluation

In this section we consider four implementations of the above method in C/C++. We have the original implementation of the CNM algorithm as provided by the authors. We also implemented it using optimized data structures to ensure fairness in the comparison. And we implemented two versions of the algorithm using the new data structures, one using binary heaps and another one using relaxed heaps [14]. Relaxed heaps should be faster than binary heaps, but are much more complex. We used the relaxed heaps implementation of the Boost graph library [14].

We also included in our implementation a randomized comparison function. As noted by Brandes *et al.* [8], the algorithm may perform badly if pairs with equal ΔQ are chosen in some crafted order. Although we can not avoid undesired behavior by ordering these pairs randomly, we expect that it will not happen every time. With respect to such fluctuations of modularity, we must mention that even small fluctuations may correspond to very different node partitionings [15]. Thus, several runs may be required to evaluate a given partitioning.

In order to evaluate the performance of the algorithm, we generated artificial networks from the partial duplication model [16, 17]. We choose this model because we are interested in knowing our algorithm efficiency on biological networks. Although the abstraction of real networks captured by the partial duplication model, and other generalizations, is rather simple, the global statistical properties of the biological networks and their topologies can be well represented by this kind of model. Note that the described model exhibits exponents of the degree distribution in the proper range and it also exhibits cluster coefficients like those seen for biological networks [18]. Note also that we do not ensure any community structure on the generated networks. Let $G_0 = (V_0, E_0)$ be an undirected and unweighted graph. Given $0 \leq p \leq 1$, the partial duplication model builds a graph $G = (V, E)$ by partial duplication as follows: start with $G = G_0$ at time $t = 1$ and, at time $t > 1$, perform a duplication step:

1. uniformly select a random vertex u of G;
2. add a new vertex v and an edge (u, v);
3. for each neighbor w of u, add an edge (v, w) with probability p.

The edges added in step 2 make the graph always connected. For simplicity, we start the duplication time at $t_0 = |V_0|$. Thus, at any time $t \geq t_0$, G has exactly t vertices. Usually, G_0 is taken to be the graph formed by one vertex.

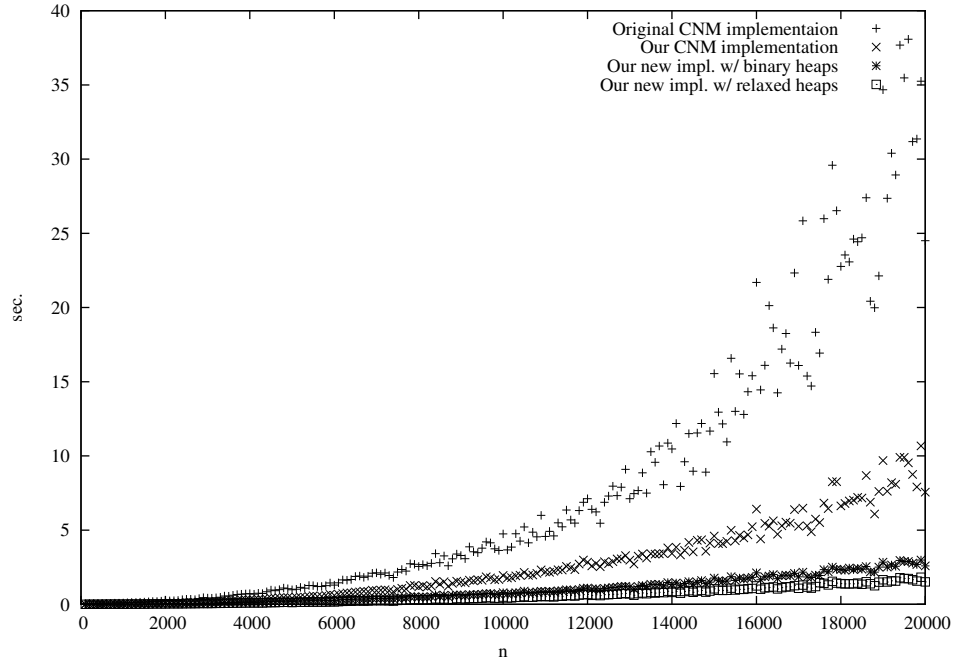

Figure 2: Average running time for duplication model graphs obtained with $p = 0.5$.

Table 1: Time and memory requirements for an artificial network with 1 million vertices and more than 13 million edges.

Implementation	**Time**	**Memory (MB)**	**Q**	**# Communities**
Our w/ relaxed heaps	8h42:24.27	990	0.679	2770
Our w/ binary heaps	14h46:02.49	918	0.679	2828
Our CNM	40h13:25.23	1,796	0.644	-
Original CNM	-	2,487	-	-

Chung *et al.* [16] shown that, with probability p approaching 1 and the number of vertices becoming infinitely large, the partial duplication model generates power-law graphs with the exponent β satisfying

$$p(\beta - 1) = 1 - p^{\beta - 1}. \tag{11}$$

In particular, if $\frac{1}{2} < p < 1$, then $\beta < 2$. However, the range of significant interest of β, between 1 and 2, is produced by only a relatively small range of selection probabilities, i.e., $0.5 < p < 0.56714329...$ In the next section we will consider selection probabilities in this range.

We generate several graphs with different number of vertices and with selection probability $p = 0.5$. Specifically, for each given number of vertices, we generate 10 random graphs. The number of edges for those graphs is approximately 10 times the number of vertices. Figure 2 compares the running time of our implementations versus the running time of the CNM algorithm. It is clear that our implementation shows an improvement over CNM implementation, as we discussed in the previous section. Consider our implementation of CNM algorithm and the new implementation with binary heaps, both use the same implementation of binary heaps and differ only on the community adjacency data structure. By analyzing Figure 2, we see and improvement of at least a factor of two, e.g., for a graph with 20,000 vertices the new implementation takes less than half of the time. The implementation with relaxed heaps performs even better. We run some tests with very large networks and, in Table 1, we have the time and memory requirements to process a network with 1 million vertices and more than 13 millions edges. For this type of networks, the implementation with relaxed heaps is clearly the best one. Note also the improvement in memory requirements.

Although we did not ensured any community structure on the generated networks, it is interesting to note that the values of modularity are most of the times higher than 0.5. This may be an interesting property of the duplication model and it deserves a better understanding. We leave this study as future work questioning if, for this class of networks, the approximation ratio can be lower than 2 since modularity is always smaller than 1. Note that this is in general false, Brandes *et al.* [8] shown that there are networks for which the approximation ratio is unbounded.

Another important fact is that we have obtained different values of modularity for each implementation. As we mentioned before, this is related with the selection order of pairs with equal ΔQ values. We implemented a randomized comparison function that picks randomly a pair whenever two pairs have the same ΔQ value. Table 2 has the ranges of the modularity values for 3 real networks. These ranges were computed by running 1,000 times one of the implementations. Note that with a randomized comparison function we can run several instances and evaluate the stability of a network partition, since modularity fluctuations may correspond to very different graph partitions.

Table 2: Maximum and minimum modularity for 3 real networks after 1,000 runs.

Network	**Max Q**	**Min Q**
Zachary's karate club [19]	0.373	0.395
C. elegans metabolic network [2]	0.388	0.423
Protein interaction network [20]	0.808	0.846

Let us give some more details about the implementation and evaluation. The above running times include the tracking of community membership. For that we use the disjoint sets data structure [13] and, therefore, the running time cost is negligible. All implementations were compiled with GNU C/C++ compiler and optimization flag `-O3`. The experimental evaluation was performed in a 2.33 GHz quad core processor with 16 GB of memory and running a GNU/Linux distribution.

6 Final remarks

In this paper we proposed a new implementation of a well known heuristic algorithm - CNM algorithm - for the problem of finding communities in graphs, based on the greedy maximization of the modularity score. The algorithm has a running time of $O(m \log n + n^2 \log n)$ and a space requirement of $O(n + m)$, in the worst case. Although the method has the same asymptotic complexity as the original CNM method, we obtained an improvement of at least a factor of 2 over previous implementations. To achieve this improvement we used cross-linked adjacency lists, binary heaps and disjoint sets to track community membership. Furthermore, by using relaxed heaps, we were able to achieve an improvement of at least a factor of 4. We were also able to reduce memory requirements by at least a factor of 2.

For sparse graphs the algorithm runs in $O(nd \log n)$, where d the depth of the dendrogram. If the graph has a hierarchical structure, $d = O(\log n)$, the running time becomes $O(n \log^2 n)$. This bound can be enforced for the majority of sparse graphs, even for those without hierarchical structure, by employing recently proposed heuristics [21]. These heuristics combine the ΔQ_{ij} values with the difference of size of communities i and j, balancing the dendrogram.

We also presented results concerning the selection order of the community pairs with equal ΔQ. In recent works as been discussed that the selection order could be important for the greedy optimization [8] and that small modularity fluctuations may correspond to very different partitions []. We implemented a randomized comparison function and we presented results concerning modularity fluctuation for known networks. Such implementation can be useful to evaluate partition stability.

As mentioned before, modularity suffers some problems [9, 10]. Thus, alternative scores are important and the described algorithm may be adapted. It is sufficient to reformulate equations 7 and 8, see for instance [11] where this algorithm was used with the similarity-based modularity.

Acknowledgments

We would like to thank Aaron Clauset for providing the source code of the original implementation. We also thank the anonymous reviewers for helpful feedback on a previous version of the paper. This work was supported by FCT, project ARN (PTDC/EIA/67722/2006).

References

[1] M. E. J. Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical Review E*, 74:036104, 2006.

[2] J. Duch and A. Arenas. Community identification using extremal optimization. *Physical Review E*, 72:027104, 2005.

[3] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99:7821, 2002.

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

[5] M. E. J. Newman. Fast algorithm for detecting community structure in networks. *Physical Review E*, 69:066133, 2004.

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

[7] P. Pons and M. Latapy. Computing communities in large networks using random walks. In *Proceedings on the 20th International Symposium on Computer and Information Sciences*, volume 3733 of *Lecture Notes in Computer Science*, pages 284–293. Springer, 2005.

[8] U. Brandes, D. Delling, M. Gaertler, R. Grke, M. Hoefer, Z. Nikoloski, and D. Wagner. On finding graph clusterings with maximum modularity. In *Proceedings on the 33rd International Workshop Graph-Theoretic Concepts in Computer Science*, 2007. to appear.

[9] S. Fortunato and M. Barthélemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36, 2007.

[10] J. M. Kumpula, J. Saramäki, K. Kaski, and J. Kertész. Limited resolution in complex network community detection with Potts model approach. *The European Physical Journal B-Condensed Matter and Complex Systems*, 56(1):41–45, 2007.

[11] Z. Feng, X. Xu, N. Yuruk, and T. A. J. Schweiger. A novel similarity-based modularity function for graph partitioning. In *DaWaK*, volume 4654 of *Lecture Notes in Computer Science*, pages 385–396. Springer, 2007.

[12] R. Guimerà, M. Sales-Pardo, and L. A. N. Amaral. Modularity from fluctuations in random graphs and complex networks. *Physical Review E*, 70(2):025101, 2004.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[14] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[15] G. Agarwal and D. Kempe. Modularity-maximizing communities via mathematical programming. *arXiv.org*, arXiv:0710.2533v3 [physics.data-an], 2008.

[16] F. Chung, L. Lu, T. G. Dewey, and D. J. Galas. Duplication models for biological networks. *Journal of Computational Biology*, 10(5):677–687, 2003.

[17] F. Chung and L. Lu. *Complex Graphs and Networks*. Number 107 in CBMS - Regional Conference Series in Mathematics. AMS, 2006.

[18] A. Bhan, D. J. Galas, and T. G. Dewey. A duplication growth model of gene expression networks. *Bioinformatics*, 18(11):1486–1493, 2002.

[19] W. W. Zachary. An information flow model for conflict and fission in small groups. *Journal of Anthropological Research*, 33:452–473, 1977.

[20] H. Jeong, S. Mason, A.-L. Barabási, and Z. N. Oltvai. Centrality and lethality of protein networks. *Nature*, 411(41), 2001.

[21] K. Wakita and T. Tsurumi. Finding community structure in mega-scale social networks. In *WWW*, pages 1275–1276. ACM, 2007.