Efficient Q-matrix computation for the visualization of complex networks

Roldan Pozo
National Institute of Standards and Technology (NIST)
Gaithersburg, MD USA
Email: pozo@nist.gov

***Abstract*—The Q-matrix representation of a large network graph is a transformation yielding meaningful visualizations which can reveal internal structure, help classify networks according to their application area, and in some cases distinguish and identify real networks from synthetically generated data. However, the issue remains whether computing the Q-matrix is prohibitively expensive for problem sizes encountered in practice. Given an undirected network graph, we describe an efficient near-linear time algorithm for computing its Q-matrix. Besides providing theoretical bounds, we present experimental results showing that application networks with millions of edges and vertices can be processed in seconds on desktop computers.**

I. Introduction

The Q-matrix of a network is a sparse matrix transformation of its underlying graph which provides useful visualizations in the classification of complex systems [11]. Figure 1 illustrates Q-matrix visualizations for various application networks used in the research literature. These figures are generalizations of degree-distribution plots which reveal such things as the number of connected components, the formation and growth of the giant component, and the effect of node-removal (i.e. site percolation) [3] on the connectivity of the remaining subgraphs –useful, for example, in the simulations of network reliability[12] and the spread of infectious diseases[5].

One interesting characteristic of these Q-matrix visualizations is that they serve as useful network *portraits*. That is, networks from different application area yield different portraits (Fig. 1) yet networks from the *same* application area share visual similarities [11]. Furthermore, unlike conventional graph layout algorithms [6] the visual representations of a Q-matrix is unique. That is, given a network graph, there is only one Q-matrix representation. The Q-matrix visualizations, like as those in Fig. 1, are just a three-dimensional view obtained by mapping the nonzero values of this sparse matrix to the z-axis, and can be easily rendered within scientific packages like MATLAB [10] or Mathematica [9]. In contrast, conventional graph drawing packages [4], [8], [13] used for visualizing networks offer a multitude of layout algorithms and heuristics (e.g. circular, hierarchical, random, force-based methods) and within each method there are often several parameters which need to be further specified in order to obtain a visually meaningful result. Thus, a single input graph may produce hundreds or even thousands of different visual configurations, and hence are of limited use as a canonical representation for identification purposes.

To be clear, the Q-matrix is not the topology of a graph in matrix adjacency format, but rather the connected component size distribution of its successive degree-limited subgraphs, defined as follows. Given an undirected graph $G = (\mathbf{V}, \mathbf{E})$, the **degree-limited subgraph** G_i is the induced graph created from vertices of G which have degree less than or equal to i. That is, $G_i \equiv (\mathbf{V}_i, \mathbf{E}_i)$ where

$$\mathbf{V}_i = \{v \in \mathbf{V} \mid \mathbf{degree}(v, G) \leq i\} \tag{1}$$

$$\mathbf{E}_i =: \{(u, v) \in \mathbf{E} \,|u, v \in \mathbf{V}_i\} \tag{2}$$

and $\mathbf{degree}(v, G)$ denotes the degree of vertex v in graph G. Furthermore, if $K_j(A)$ is the number of connected components of graph A which have j vertices, the Q-matrix is defined as

$$Q_{i,j} \equiv K_j(G_i) \tag{3}$$

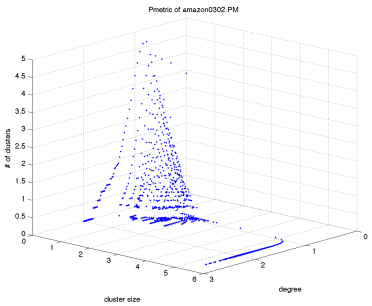
In other words, Q_{ij} is the number of connected components of size j in G_i. The Q-matrix transformation, $G \rightarrow Q(G)$, therefore, maps an undirected graph with $V = |\mathbf{V}|$ vertices and $E = |\mathbf{E}|$ edges into a (sparse) matrix indexed by $[0, 1, \ldots, D(G)] \times [1, 2, \ldots, T(G)]$, where $D(G)$ is the maximum node degree of G, and $T(G)$ is the size of the largest connected component of G. As an example, a small graph and its Q-matrix are illustrated in Fig.2. Note that these matrices are usually rather sparse (as not all component sizes are represented in the sugraphs G_i) and that some rows are repeated (if there are no vertices with degree i in G, then $G_i = G_{i-1}$, hence row i and row $i - 1$ are identical in Q). In practice, we represent Q as a sparse matrix and remove the repeated rows, as this information can be easily re-constructed according to the definitions of Q. This compact representation requires less memory, but is mathematically equivalent.

II. Computing the Q-matrix

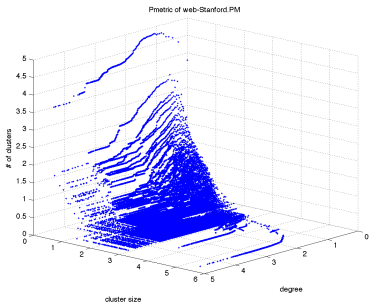
We now turn our attention to the matter of calculating the Q-matrix in practice. While creating a Q-matrix of a network has important uses, it is of little value if it is prohibitively expensive to compute. Because the network graphs we are interested in are often have millions of vertices and edges, providing a practical algorithm with reasonable complexity

978-0-7695-4911-8 2012

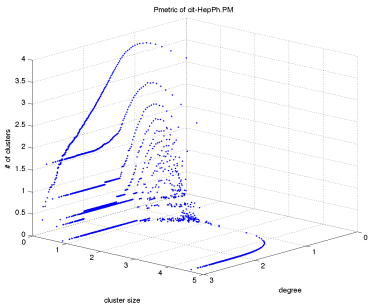
DOI 10.1109/SITIS.2012.115



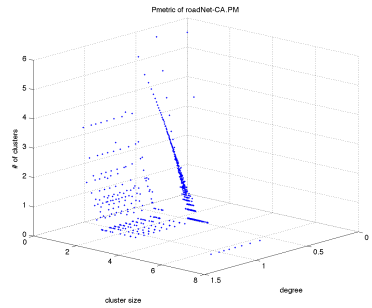
(a) co-purchasing network



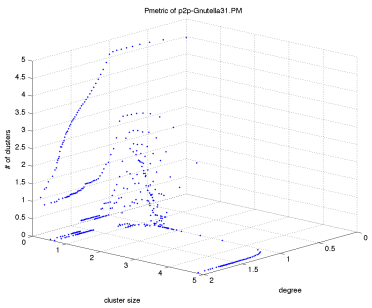
(b) webgraph



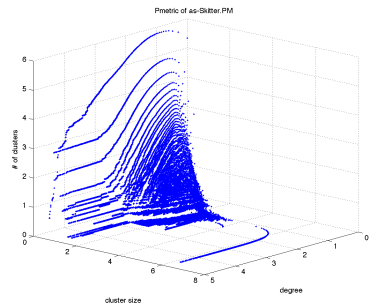
(c) citation graph



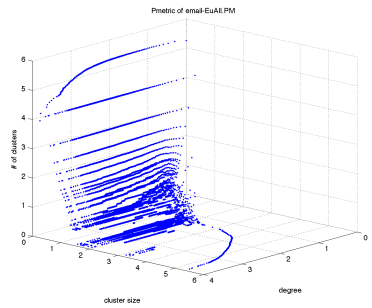
(d) road network



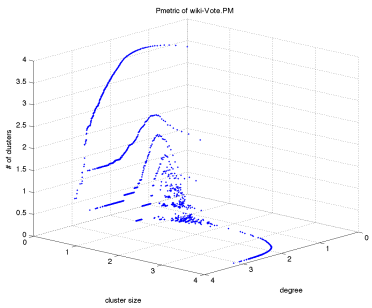
(e) peer-to-peer (p2p) network



(f) autonomous network

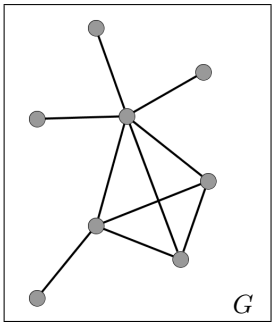


(g) email network



(h) Wikipedia network

Figure 1: Examples of Q-matrix visualizations of application networks from the *Stanford Large Network Dataset Collection* [7] : (a) **amazon0302** (262K vertices, 1.2 M edges), (b) **web-Stanford** (325K vertices, 2.3M edges) , (c) **cit-HepPh** (34.5K vertices, 421K edges), (d) **roadNet-CA** (1.9M vertices, 5.5M edges), (e) **p2p-Gnutella31** (62.6K verties, 147K edges), (f) **as-Skitter** (1.7M vertices, 11.1M edges), (g) **email-EuAll** (265K vertices, 420K edges), (h) **wiki-Vote** (76K vertices, 508K edges).



$$Q(G) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{4} & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{3} & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\ \mathbf{3} & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \end{bmatrix} \quad (4)$$

Figure 2: A small graph G and its Q-matrix representation

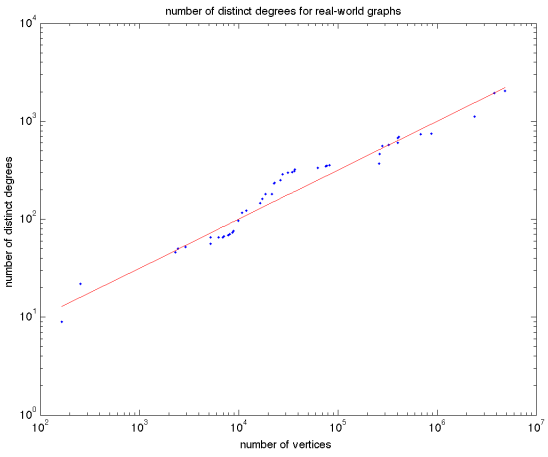


Figure 3: The number of distinct degrees of real-world networks grow at about the square root of the the number of vertices, O($\sqrt{(V)}$).

bounds and an efficient implementation is essential. At first glance, the computational cost may seem to be $O(n^2)$ in terms of the size of the graph (number of edges and vertices) yielding an impractical solution. However, we will examine various optimizations that result in a near linear time bound, $O(n)$, allowing for large network graphs to be efficiently processed.

A. *Naive algorithm:* $O(n^2)$

Since the ith row of the Q-matrix represents the component size distribution for the degree-limited subgraph G_i, computing the entire matrix would seem to require the creation of subgraphs $G_0, G_1, \ldots G_D$ and finding their subsequent connected components. Computing each G_i separately requires $O(V + E)$ operations to pick out the participating vertices and their corresponding edges from the original graph. Once we have G_i, its connected components can be computed in $O(V_i + E_i)$ operations using either a breadth-first or depth-first search. Since $V_i \leq V$, and $E_i \leq E$, their combined operation count is still $O(V + E)$ and we repeat this for $i = 0, 1, 2, \ldots D$ or a total of $D+1$ times. Furthermore, since $D+1 \leq V$, the worst-case complexity is $O(V(V+E))$ operations, or $O(V^2)$ when $V \approx E$ for this naive approach.

B. *First optimization:* $O(n^{1.5})$

Let $\vec{d} \equiv \{d_0, d_1, \ldots, d_D\}$ be the degree distribution of G. That is, d_i is the number of vertices in G with degree equal to i. We can reduce the complexity by realizing that the number of distinct degree-limited subgraphs G_i is equal to the number of non zeros in the degree distribution of G. Thus, if $z \equiv [z_1, z_2, \ldots, z_k]$ denotes the nonzero indices of the degree sequence $\vec{d}$, then we need only to compute k subgraphs, $G_{z_1}, G_{z_2}, \ldots G_{z_k}$, rather than $D + 1$ of them. Fig. 3 shows that in practice, the number of distinct degrees in networks graphs from various applications grows approximately as $O(\sqrt{V})$, reducing the overall complexity to $O(\sqrt{V}(V + E))$, or $O(V^{1.5})$ when $V \approx E$.

C. *Second optimization:* $O(n)$

The second, and more important, optimization is to compute the connected components incrementally and avoid forming the G_i subgraphs explicitly. Having established that we only need degree values which have a nonzero count in the frequency distribution, we consider degree values $\{z_i \mid i = 1, 2 \ldots k\}$.

We can use the connected components of the subgraph at degree z_i compute the connected components for the next distinct subgraph at degree z_{i+1} by noting that $G_{z_i} \subset G_{z_{i+1}}$ and that the connected component relationship forms an equivalence class among the participating vertices. The key idea is to first presort the vertices of the original graph, G, according to their degree. Then, by examining vertices in this order, one can determine the connected component distributions of each G_i without computing each subgraph explicitly. By using an equivalence class structure based on disjoint sets, we can amortize the cost of inserting an edge or vertex to $O(\alpha)$, where α is the inverse Ackerman function, which essentially provides a constant cost. (See details in next section.) This insertion is performed for every vertex and edge, so the total cost is $O(\alpha(V + E))$ or $O(V)$ when $V \approx E$, effectively yielding a linear time algorithm.

Note that even though high-degree nodes are excluded in the early stages of the algorithm, their connections to low-degree nodes are later recovered due to the bi-connectedness (undirected edges) of the graph topology, as detailed in the next section.

Data: Graph G

Result: Q-matrix

```
1  (* initialization *);
2  create D | D[i] = set of vertices in G with degree i ;
3  Q ← 0;
4  L ← ∅;
5  (* compute Q, row by row *);
6  foreach i | D[i] ≠ ∅ do
7      foreach vertex v ∈ D[i] do
8          make v a separate equivalence class in L;
9          Q(i, 1) ← Q(i, 1) + 1;
10     foreach vertex v ∈ D[i] do
11         foreach neighbor u of v in G do
12             if degree(u) ≤ i then
13                 if u and v are in different classes of L then
14                     join equivalence classes [u] and [v] in L;
15                     Q(i, |[u]|) ← Q(i, |[u]|) − 1;
16                     Q(i, |[v]|) ← Q(i, |[v]|) − 1;
17                     Q(i, |[u]| + |[v]|) ← Q(i, |[u]| + |[v]|) + 1;
```

Figure 4: optimized algorithm for incremental Q-matrix computation

III. Algorithmic description

An overview of the optimized incremental algorithm is shown in Fig. 4. We begin by pre-sorting the vertices into a vertex set array D such that the element $D[i]$ denotes the set of vertices with degree i (line 2). The largest index is the maximum degree of G, which is at most $V - 1$. Hence, the cost for forming D is $O(V)$.

Next we create an empty Q-matrix, usually implemented as a sparse row matrix, and an empty equivalence class L, which will hold the vertex sets forming the individual connected components during the incremental computations. (That is, if vertices u and v are in the same equivalence class, then they are in the same connected component.)

Beginning with an empty equivalence relation, L, and a Q-matrix with all zeros, the algorithm proceeds by building Q, row by row, starting from the top down. We continue increasing i until $D[i]$ is non-empty (line 6). At this point, $D[i]$ contains the set vertices with degree i, and we insert each of these vertices as a separate single-element equivalence class in L (line 8). In essence, Q will maintain a running tally of the equivalence class sizes in L, so for each new vertices we added, we need to increment the number of single-element components (line 9). We now go back and examine each new vertex separately. For each v in $D[i]$, we examine its neighbors; if neighbor u has degree i or less, then that vertex is already represented in L. We therefore join the equivalence class of v with the equivalence class of u (line 14). If $[u]$ denotes the equivalence class of u, and $|[u]|$ denotes the size of this equivalence class, then this join operation effective removes a class of size $|[u]|$ and one of size $|[v]|$ and replaces it with one of size $|[u]| + |[v]|$ (lines 15-17).

In the early stages of the algorithm, where the degree value is low, large-degree nodes (hubs) are excluded from being counted in the connected components (line 12). This may seem like this relationship is lost. However, when the hubs are visited later in the algorithm, the connections to low-order nodes are recovered, because they still appear as neighbors (line 11) and qualify for inclusion (line 12).

Implementations of the equivalence relation L can be based on forest disjoint-set methods [2], which provide an amortized complexity time of $O(\alpha)$ for inserting new elements (line 8), and joining two distinct subsets (line 11), where α is the inverse Ackerman function [1]. This function grows very slowly (much slower than $\log(n)$), and for practical purposes, can be treated as a constant. Indeed, $\alpha(2^{2^{10^{19,729}}}) < 5$, so this is essentially $O(1)$ for any graph of practical size.

The Q-matrix complexity cost is broken down as follows: $O(V)$ for the initialization of D (line 2); then, for each vertex, $O(\alpha)$ for creating a new single-element equivalence class (line 8), $O(1)$ and for

updating the Q-matrix (line 9); for each edge, $O(1)$ for the three Q-matrix updates (lines 13-15), and $O(\alpha)$ for joining two equivalence classes (line 16). The total complexity is $O(V) + O(\alpha)V + E(O(\alpha) + O(1))$ or simply $O(V + E)$, as $O(\alpha)$ can be bounded by $O(1)$ for any graph of practical interest.

IV. Experimental Results

Aside from the theoretical analysis of this optimized algorithm, it is important to understand how it is implemented in practice. Algorithms with an impressive theoretical complexity may still fail to translate into practical solutions, as the hidden constants in the $O()$ notation can sometimes be overwhelming. In this section, we present timing results for a C++ implementation for Q-matrix calculations on application networks varying from several thousand to several million edges and vertices.

The application networks are taken from the Stanford Large Network Dataset Collection (SNAP) [7], many of which are commonly used in network science research. They include collaboration networks, web graphs, citation networks, co-purchasing networks, email networks, and online social networks. A summary appears in Table I. The networks sizes range over five orders of magnitude, from several thousand, to nearly 70 million.

The fifth column of this table illustrates the execution times of computing the Q-matrix for each network, using g++ 4.6.3 on a 3.4GHz Intel i7 desktop with 16GB RAM, running Linux 3.2.0. The data shows that most networks (those containing up to a few million vertices and edges) require only a few seconds to compute their respective Q-matrices. The largest network in this group, the Live-Journal social network graph (soc-LiveJournal1) with nearly 5 million vertices and 70 million edges required 122 seconds.

For a better illustration, we plot the graph size for these networks (V+E) and their execution times on a log-log scale and compare this to the theoretical $O(V + E)$ slope, showing fairly good agreement. Of course, practical considerations must be taken into account, such as the physical size of memory on the computer used, the particular compiler optimizations employed, and other platform-specific details. Nevertheless, the data at least suggests that the running time and theoretical algorithmic complexity share similar characteristics for problem sizes spanning several orders of magnitude.

V. Conclusion

The Q-matrix representation of a network graph allows for useful visualizations and has been shown to be an effective tool for network analysis. Nevertheless, its usefulness is diminished if the computational costs limit it to tiny networks. Dataset sizes continue to grow in the area of network science, and while

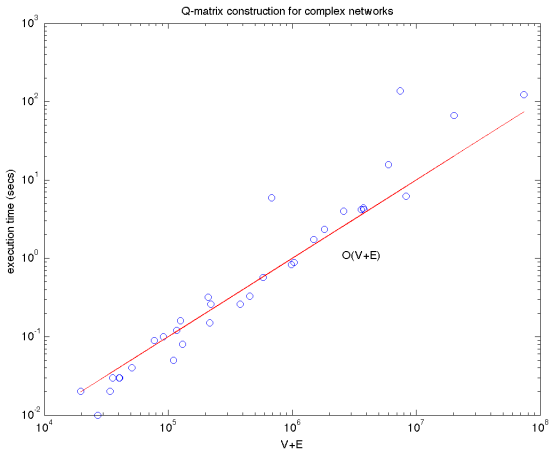


Figure 5: Execution times for Q-matrix and, for comparison, the theoretical $O(V + E)$ time bound.

networks of several thousand elements seemed adequate for analysis in the recent past, researchers are now facing networks consisting of millions or more elements on a regular basis.

We have presented an efficient algorithm for computing Q-matrices of large network graphs. Starting with a prohibitively expensive $O(n^2)$ algorithm, we have shown how successive optimizations can reduce this to a near linear $O(n)$ complexity. The basic idea is to compute the Q matrix in one pass of the graph and not have to form explicit degree-limited subgraphs to compute components. This requires abandoning the classic linear-time component algorithm employing a depth-first or breadth-first search with a slightly less efficient version using equivalence classes (disjoint sets) that allows for incremental component calculation.

To ground these theoretical bounds with practical results, we have presented experimental data for Q-matrix calculations for over forty network graphs, from various application areas (collaboration networks, web graphs, citation networks, co-purchasing networks, email networks, and online social networks) ranging from sizes of several thousands to several millions of edges and vertices. These calculations were performed on an ordinary desktop computer and demonstrate that even networks containing millions of elements can be process in just a few seconds, with our largest network in this study (5 million edges, 70 million vertices) requiring just a little over two minutes to compute.

In short, Q-matrices can be efficiently computed using the algorithms presented here. This makes Q-matrix analysis accessible to nearly everyone and allows this approach to remain an important tool for the study of complex networks.

	NETWORK	NODES	EDGES	TIME (secs)	REFERENCE[7]
Collaboration Networks	Astro Physics	18,772	396,160	0.15	ca-AstroPh
	Condensed Matter	23,133	186,936	0.12	ca-CondMat
	High Energy Physics	12,008	237,010	0.08	ca-HepPh
	High Energy Physics Theory	9,877	51,971	0.03	ca-HepTh
Web graphs	Google	875,713	5,105,039	15.7	web-Google
	Notre Dame	325,729	1,497,134	2.35	web-NotreDame
	Stanford	281,903	2,312,497	4.02	web-Stanford
	Berkeley-Stanford	685,230	7,600,595	6.19	web-BerkStan
Citation Networks	High Energy Physics	34,546	421,578	0.33	cit-HepPh
	High Energy Theoretical Physics	27,770	352,807	0.26	cit-HepTh
	US Patents	3,774,768	16,518,948	66.42	cit-Patents
Co-purchasing networks	March 2	262,111	1,234,877	1.75	amazon0302
	March 12	400,727	3,200,440	4.18	amazon0312
	May 5	410,236	3,356,824	4.37	amazon0505
	June 1	403,394	3,387,388	4.22	amazon0601
Email networks	Enron	36,692	183, 831	0.26	email-Enron
	European University	265,214	420,045	5.90	email-EuAll
Online social	Epinons	5,879	508,837	0.57	soc-Epinions1
	LiveJournal	4,847,571	68,993,773	122.37	soc-LiveJournal1
	Slashdot (11-2008)	77,360	905,468	0.83	soc-Slashdot0811
	Slashdot (02-2009)	82,168	948,464	0.89	soc-Slashdot0922

Table I: Application networks from the Stanford Large Network Collection [7], together with their graph size and execution times (secs) for computing their Q-martrix.

REFERENCES

[1] W. Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, 99:118–133, 1928.

[2] J. J. Fischer B. A. Galler. An improved equivalence algorithm. *Comm. ACM*, 7:301–303, 1964.

[3] A. Aharony D. Stauffer. *Introduction to Percolation Theory*. Taylor and Francis, London, 1992.

[4] S. C. North E. R. Gansner. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[5] H. W. Hethcote. The mathematics of infectious diseases. *Siam Rev.*, 42:599–563, 2000.

[6] M. S. Marshall I. Herman, G. Melancon. Graph visualization and navigation in information visualization: A survey. *IEEE Trans. Visualization and Computer Graphics.*, 6:24–43, 2000.

[7] J J. Leskovec. Stanford large network dataset collection. `http://snap.stanford.edu/data`.

[8] M. Jacomy M. Bastian, S. Heymann. Gephi: an open source software exploring and manupulating networks. *Int. AAAI Conf. Weblogs and Social Media*, 2009.

[9] Mathematica. *version 7.0*. Wolfram Research, Inc., Champaign, Illinois, 2008.

[10] MATLAB. *version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts, 2010.

[11] R. Pozo. Q-matrix: an algebraic formulation for the analysis and visualizaiton of network graphs. preprint (2012), available at `http://math.nist.gov/pozo`.

[12] A. Barabási R. Albert, H. Jeong. Attack and error tolerance in complex networks. *Nature.*, 406:378–382, 1999.

[13] yEd. *version 3.9.2*. yWorks GmbH., Tübingen, Germany, 2012.