Reconstructing Pathways in Large Genetic Networks from Genetic Perturbations

Andreas Wagner

SFI WORKING PAPER: 2001-09-050

SANTA FE INSTITUTE

Reconstructing pathways in large genetic networks from genetic perturbations.

Andreas Wagner

University of New Mexico
and
The Santa Fe Institute

University of New Mexico
Department of Biology
167A Castetter Hall
Albuquerque, NM 817131-1091
Phone: +505-277-2021
FAX: +505-277-0304
Email: wagnera@unm.edu

Abstract: I present an algorithm that determines the longest path between every gene pair in an arbitrarily large genetic network from large scale gene perturbation data. As a by-product, the algorithm reconstructs all direct regulatory gene interactions in the network. The algorithm is recursive, and is built around a graph representation of genetic networks. Its computational complexity is $O(nk^2)$, where n is the number of genes in the network, and k is the average number of genes affected by a genetic perturbation. In practice, it can reconstruct all path lengths for a network of more than 6000 genes in less than 30 CPU seconds. It is able to distinguish a large fraction of direct regulatory interactions from indirect interactions, even if the quality of its input data is substantially compromised

Introduction

Perhaps the most fundamental questions about the structure of a large genetic networks are these: what are the pathways connecting genes in the network? And which genes in a network influence the activity of which other genes *directly*? I here present an algorithm that can answer these question for gene networks of an arbitrary number of genes. The algorithm's power to resolve regulatory gene interactions, and thus to resolve the causal structure of a genetic network, comes at a price. It requires different kinds of data – and more of it – than existing "clustering" methods of various flavors (DeRisi et al. 1997; Eisen et al. 1998; Tavazoie et al. 1999). Specifically, it requires perturbation of many genes in a network. The required large-scale data is now becoming available in a variety of model organisms (Hughes et al. 2000; Pennisi 1998; Somerville and Somerville 1999; Spradling et al. 1999).

Before introducing the algorithm, I need to introduce some terminology. First, what is a *genetic network*? For the purpose of this paper, I define a genetic network as a group of genes in which individual genes can change the activity of other genes. What, then, is a change in *gene activity*? Changes in gene activity might include many different things, such as changes in gene expression – on the mRNA or protein level –, alternative splicing, post-transcriptional regulation, or post-translational modification, such as phosphorylation. I will restrict myself here to mRNA expression as measured by micro-arrays (Lockhart and Winzeler 2000), and its change in response to a genetic perturbation. However, the principal idea applies to any notion of gene activity.

Next, what is a *genetic perturbation*? For my purpose, it is an experimental manipulation of gene activity through either the gene itself or its product. Genetic perturbations include point mutations, gene deletions, overexpression, inhibition of translation, for example by using antisense RNA, or any other interference with the activity of the product. Although mutations are not usually thought of as manipulations of gene activity, I choose to view them as such.

When manipulating a gene which affects the activity of other genes one can ask whether this effect is due to a direct or an indirect interaction. For example, when overexpressing a transcription factor X, I might find that the expression levels of genes A and B change. Upon further investigation, I may find that X binds the upstream regulatory region of A and up-regulates its expression. This is what I call a *direct* effect of X on A. However, in the case of B I might find that X induces the expression of a protein phosphatase, which dephosphorylates and thus inactivates a transcriptional repressor of B. This is what I call an *indirect* effect of X on B.

To reconstruct a genetic network is to identify all direct effects of network genes on one another, within the limits of experimental resolution.

I will restrict myself to qualitative information on how genetic perturbations affect gene expression. That is, when manipulating the activity of one gene, what other genes are affected in their expression? The motivation for this restriction lies in the available technology to measure gene expression: it is associated with great amounts of experimental noise. Qualitative information on gene interactions lends itself ideally to a graph representation of genetic networks. A *directed graph* or *digraph* is a mathematical object consisting of *nodes* and *directed edges*. In a graph representation of a genetic network, the nodes of the graph correspond to genes, and two genes, say gene 1 and gene 2, are connected by a directed edge (an arrow, $1 \rightarrow 2$) if gene 1 influences the activity of gene 2 directly. Figure 1A shows a graph representation of a hypothetical genetic network G of 21 genes. Figure 1B shows an alternative representation of the network shown in 1A. For each gene *i*, Figure 1B contains a list of genes whose activity is directly influenced by gene *i*. This list is called the adjacency list *Adj(i)* of *i*. *Adj* is the adjacency list of *G*. One might also call it the list of direct regulatory interactions. It completely defines the structure of the network.

Genetic perturbation experiments can not distinguish direct from indirect interactions. That is, when perturbing a gene in the network shown in Figure 1A, one identifies all genes that this perturbation affects directly or indirectly, as its effects ripple through the network. For example, gene 0 influences, directly or indirectly, the activity of genes 2 and 16. Gene 1 does not influence the activity of any other gene, but its activity is influenced by genes 9 and 10. Gene 4 is an isolated node in this network. Its activity is neither influenced by any gene in the network, nor does it influence the activity of any gene. Starting from a graph representation of the network in Figure 1A, one arrives at the list of direct and indirect causal interactions in Figure 1C by following all paths leaving a gene. This list is also called the accessibility list *Acc* of a graph. because it shows all nodes (genes) that can be accessed (influenced in their activity state) from a given node by following paths of direct interactions. In the context of a genetic network one might also call it *the list of perturbation effects* or the *list of regulatory effects*. For each perturbed gene *i*, *Acc(i)* denotes the list of affected genes.

Generating the list in Figure 1C from 1A or 1B is straightforward. The subject of this paper is the more difficult problem of reconstructing all direct interactions in a network from all indirect ones, that is, to reconstruct 1A (*Adj*) from 1C (*Acc*), and to do that automatically for very large networks comprising thousands of genes. Genome scale perturbation experiments are beginning to provide the raw material for this endeavor (Hughes et al. 2000; Pennisi 1998;

Somerville and Somerville 1999; Spradling et al. 1999). Its significance is best viewed in relation to the main thrust of molecular genetics, which uses genetic perturbations to reconstruct biochemical pathways involving painstaking gene-by-gene analysis. Automating such reconstruction for networks of thousands of genes would not only facilitate this task, but take it to a completely different scale.

Results

Previous results. I briefly review some standard terminology (Harary 1969) as well as some relevant theorems from a recent contribution (Wagner 2001b). For my purpose, graphs are best divided into two classes, those with and without cycles. A cycle is a path starting at a node and leading back to the same node. Graphs without cycles are called *acyclic graphs*. The problem that networks with cycles pose for reconstruction is illustrated in Figure 2A with two simple cyclic networks. The order of direct regulatory interactions in these two networks is different, as reflected in the adjacency lists shown. However, both networks, when perturbed, would generate the accessibility list shown in Figure 2B. Characteristically, perturbation of any gene influences the activity of all other genes in such a cyclic network (Figure 2B). Thus, all possible orders of the five genes in a cycle are consistent with the accessibility list of Figure 2B. Notice that this is not an algorithmic but an experimental limitation. Single gene perturbations can not resolve gene orders in a cycle.

One might think that acyclic graphs must be rather simple objects. This is not so. Consider the two networks of 20 genes shown in Figure 2C and Figure 2D. They differ by only one edge: The network in Figure 2D has a directed edge from gene 13 to gene 4 that is missing in Figure 2C. The network in Figure 2D is cyclic (e.g., it contains the cycle 4→6→9→19→13→4). The network in Figure 2C is acyclic. Thus, the distinction between cyclic and acyclic networks need not be obvious.

Statistical inference from the perturbation response of transcriptional regulatory networks suggests that such networks are sparse, containing on average less than one direct regulatory interaction per gene (Wagner 2001a). And biological networks whose coarse scale structure has been analyzed resemble sparse random networks with a scale-free degree distribution (Fell and Wagner 2000; Jeong 2000; Wagner 2001c). Together, these observations suggest, albeit not conclusively, that gene networks may not be rife with cycles.

How does one treat cycles in network reconstruction? The idea is to first identify all cycles, and then collapse all nodes that are part of a cycle. The remaining network is acyclic and can be reconstructed as described below. More precisely, a *strongly connected component* or *strong component* of a directed graph G is a maximal subset of nodes of G in which every two nodes are mutually accessible. (This implies that there is a cycle through any two nodes of a strong component.) I have recently described an algorithm that identifies all strong components of a graph from its accessibility list, and generates from these components the *condensation* of the graph (Wagner 2001b). The *condensation* G^* of a directed graph G has the strong components of the graph G as its nodes,. Denote the components of G (nodes of G^*) as $S_1, \ldots, S_k$. There is an edge from any S_i to any S_j in G^* if there is an edge in G from at least one node in the component S_i to at least one node in the component S_j. To reconstruct a genetic network from single-gene perturbation experiments, one thus needs to identify all the strong components from the accessibility list generated by a perturbation experiment. The acyclic condensation thus obtained can be reconstructed as outlined below. If the network has no cycles to begin with, this step can be omitted and the original, acyclic network can be reconstructed directly.

An acyclic directed graph uniquely defines its accessibility list, but the converse is not true. In general, if *Acc* is the accessibility list of a graph, there may be many graphs *G* with Acc as their accessibility list. Among all these graphs, however, one has a unique property: it has the fewest edges. I call it the *most parsimonious graph* consistent with *Acc*. More precisely, the following theorem holds (Wagner 2001b).

Theorem 1: Let *Acc* be the accessibility list of an acyclic digraph. Then there exists exactly one graph G_{pars} that has *Acc* as its accessibility list and that has fewer edges than any other graph *G* with *Acc* as its accessibility list.

This graph has the property that it is shortcut-free (Wagner 2001b):

Definition: Consider two nodes *i* and *j* of a digraph that are connected by an edge *e*. The **range** *r* of the edge *e* is the length of the shortest path between *i* and *j* in the absence of *e*. If there is no other path connecting *i* and *j*, then $r := \infty$. An edge e with range $r \geq 2$ but $r \neq \infty$ is called a **shortcut**.

Mathematical foundation of the algorithm. The algorithm I present here rests on the following two propositions.

Proposition 1. For two nodes *u* and *w* of an acyclic, shortcut-free digraph, let *p(u,w)* be the length (number of edges) of a *longest* path connecting *u* and *v* (*p(u,u)=0*, *p(u,w)=*∞ if *w*$\notin$*Acc*(*u*)).

$$\forall w \in Acc(u): \quad p(u,w) = \max_{\{v|v\in Acc(u)\wedge w\in Acc(v)\}} p(u,v) + p(v,w)$$

That is, the longest path *p(u,w)* between *u* and *w*, where *w* is accessible but not adjacent to *u*, is equivalent to the sum over the longest paths *p(u,v)* + *p(v,w)*, maximized over all *v* from which *w* is accessible. The Proposition is a consequence of path length additivity in a graph. If *w* is adjacent to *u*, then *v=u*, and *p(u,w)=p(u,u)+p(u,w)=0+1=1*. No path with *p(u,w)>1* exists in this case, because the graph is shortcut-free.

The following could be viewed as a special case of Proposition 1.

Proposition 2: For two nodes *u* and *v* of an acyclic, shortcut-free digraph, let *p(u,v)* be the length (number of edges) of a *longest* path connecting *u* and *v* (*p(u,u)=0*, *p(u,v)=*∞ if *v*$\notin$*Acc(u)*).

$$\forall w \in Acc(u) \setminus Adj(u): \quad p(u,w) = \max_{\{v|v\in Acc(u)\wedge w\in Acc(v)\}} 1 + p(v,w)$$

Proof: For the purpose of this proof, consider only nodes *v* from the set $\{v|v\in Acc(u)\wedge w\in Acc(v)\}$ Among such nodes *v*, I will distinguish between those adjacent to *u*, and those accessible but not adjacent to *u*. For any *v*$\in$*Adj(u)*, *p(u,v)=1* because the graph is shortcut-free. A longest path (not necessarily unique) from *u* to *w* must pass through one $v_i \in Adj(u)$. The length of this longest path is one plus the length of the longest path from v_i to *w*. Thus, to arrive at a longest path between *u* and *w*, one can take the above maximum over *v*$\in$*Adj(u)*. What remains to be shown is that there exists no v_j accessible <u>but not adjacent</u> to *u* ($v_j\in Acc(u)\setminus Adj(u)$) so that

$$1 + p(v_j, w) > \max_{\{v|v\in Adj(u)\wedge w\in Acc(v)\}} 1 + p(v,w) = p(u,w)$$

Assume there was such a $v_j\in Acc(u)\setminus Adj(u)$. Because v_j is accessible from *u*, there exists a v_h adjacent to *u*, such that the longest path from *u* to v_j is given by $p(u,v_j)=1+ p(v_h,v_j)$. Because of Proposition 1, $p(u,w)\geq p(u,v_j)+p(v_j,w)$, and in sum we have $p(u,w)\geq p(u,v_j)+p(v_j,w)=1+p(v_h,v_j)+p(v_j,w)>1+p(v_j,w)$, which is in contradiction to the assumption.

Proposition 2 does not apply to nodes $w \in Adj(u)$, but for those nodes $p(u,w)=1$. One might surmise that an exactly analogous proposition must hold for minimum path lengths $p_{min}(u,w)$. That is not the case. The reason lies in the second half of the proof. For u,w such that $p_{min}(u,w)>2$, there exists a $v_j \in Acc(u) \backslash Adj(u)$, such that $1+p_{min}(v_j,w)<p_{min}(u,w)$. It is a v_j adjacent to w, for which $p_{min}(v_{j,},w)=1$. (However, minimum path lengths can easily be reconstructed from a breadth first search (Mehlhorn and Naher 1999) on the adjacency list $Acc(u)$, given by all v such that $p(u,v)=1$.)

The algorithm. The recursive algorithm shown as pseudocode in Figure 3 takes advantage of Proposition 2 to build a list of maximum path lengths $p(u,v)$ in a graph. Because it operates on lists, programming languages such as perl or library extensions of other languages permitting list operations will facilitate its implementation. (Appendix A shows a perl implementation of the algorithm, where accessibilities and path lengths are represented by a two-dimensional hashing array.)

The algorithm (Fig. 3) needs an accessibility list for each node u, $Acc(u)$, which would be obtained experimentally from gene perturbation data and subsequent gene activity measurements. Initially all pathlengths $p(u,v)$ should be undefined or assigned some impossibly small value, such as $p(u,v)=-1$. In lines one through four (Figure 3), a master loop cycles over all nodes u and calls the routine MAXPATH for each node u. In the last statement of this routine (line 17) the calling node is declared as visited. Once a node u has been visited, the maximum path lengths $p(u,v)$ to all $v \in Acc(u)$ have been calculated. The conditional statement in the master loop (line 2) skips nodes that have already been visited.

Aside from storing *Acc* and path lengths, the algorithm needs to keep track of all visited nodes. In an actual implementation, *Acc*, path lengths, and any data structure that keeps track of visited nodes must be either global variables or passed into the routine MAXPATH, preferably by reference. In contrast, the calling node u needs to be a local variable because MAXPATH is recursive.

I will now explain the function MAXPATH itself, which is at the algorithm's core. It consists of two loops. The first loop (lines 6-12) cycles over all nodes v accessible from the calling node u. If there exists a node accessible from v, then MAXPATH is called from v. If no node is accessible from v, that is, if $Acc(v) = \emptyset$, then v is declared as visited. Because its accessibility list is empty in this case, there exists no node z for which $p(v, z)$ has a positive integer value. Thus $p(v, z)$ needs no further modification. In sum, MAXPATH calls itself recursively in the first loop until a node is reached whose accessibility list is empty. There always exists such a node, otherwise the graph would not be acyclic. This also means that infinite recursion is not possible for an acyclic graph. Thus, the algorithm always terminates. More precisely, the longest possible chain of nested calls of MAXPATH is $(n-1)$ if G has n nodes. For any node u calling MAXPATH, the number of nested calls is at most equal to the length of the longest path starting at u.

In line 12, the last statement of the first loop, the longest path from u to v is temporarily assigned a length of one, which is the smallest possible maximum path length for any $v \in Acc(u)$. This assignment is the departure point for the iterative path length calculations in the second loop of MAXPATH (lines 13-15), where Proposition 2 is applied to determine $p(u,v)$ for each v accessible from u. This loop only starts once the algorithm has explored all nodes accessible from the calling node u, that is, as the function calls made during the first loop return. The iteration scheme in lines 13 and 14 deserves some explanation. Following Proposition 2, one would first iterate over all nodes $w \in Acc(u)$, and then maximize $1+p(v,w)$ over all v with $w \in Acc(v)$. This would require, for each v, a time consuming search to determine whether $w \in Acc(v)$. Instead, the algorithm uses a modified iteration scheme, which essentially swaps the order of iterations. It iterates first over all nodes $v \in Acc(u)$, and then over all $w \in Acc(v)$ to maximize $1+p(v,w)$ (line 15). By the time line 15 of the second loop is reached, all nodes v accessible from u have been visited in previous calls to MAXPATH. Thus, all $p(v,w)$ are known, which is what the calculation of $p(u,w)$ relies on. In sum, lines 13-15 determine all $p(u,w)$, except for nodes adjacent to u, for which the maximum path lengths is one (established on line 12), and u itself, for which $p(u,u)=0$ (line 16).

A by-product of the algorithm is the adjacency list of each node u, that is, the list of nodes v for which $p(u,v)=1$.From this list, minimum path lengths can easily be reconstructed using a breadth-first search algorithhm (Mehlhorn and Naher 1999).

Computational and storage complexity. Both measures of algorithmic complexity are determined by the average number of entries in a node's accessibility list. Let $k<n-1$ be that number. For all practical purposes, there will be many fewer entries than that, not only because accessibility lists with nearly n entries are not accessibility lists of acyclic digraphs, but also because most real-world graphs are sparse (Fell and Wagner 2000; Jeong 2000; Wagner 2001c)

During execution, each node v accessible from a node u induces one recursive call of MAXPATH, after which the node accessed from u is declared as visited. Thus, each entry of the accessibility list of a node is explored no more than once. However, line 14 of the algorithm

(Figure 3) loops over all nodes *w* accessible from the called node *v*. This leads to overall computational complexity $O(nk^2)$.

For practical matters, large scale experimental gene perturbations in the yeast *Saccharomyces cerevisiae* ($n \approx 6300$) suggests that $k<50$ (Hughes et al. 2000; Wagner 2001a), and thus that $nk^2<n^2$ in that case. In practice, a network of 6,300 nodes (the approximate number of genes in the genome of the yeast *Saccharomyces cerevisiae*) and the same number of edges was reconstructed in less than 30 seconds on a desktop workstation (450MHz Pentium II; RedHat Linux 6.2), using the perl implementation of the algorithm shown in the Appendix. Even for the much larger human genome ($n \approx 30000$), network reconstruction would thus be feasible on a desktop computer.

In terms of memory requirements, the algorithm needs a copy of the accessibility list ($O(nk)$), a list of path lengths($O(nk)$), as well as a list of the nodes that have been visited ($O(n)$). The recursion stack requires additional storage. However, the recursion depth can be no greater than $n-1$ because otherwise the graph would not be acyclic. Thus, overall storage requirements are $O(nk)$.

Messy data. Any gene perturbation experiment will be subject to measurement error. Either genes affected by a perturbation will not be detected as such, or genes unaffected will show spurious changes in activity. This will affect the accessibility list generated by an experiment, which is a problem, because not just any list of the form shown in Fig. 1C is the accessibility list of a graph. Take the accessibility list of the simple graph $1 \rightarrow 2 \rightarrow 3$. Eliminate only one entry, indicated in parentheses

1:	2	(3)
2:	3	
3:		

and the resulting list is not the accessibility list of a graph anymore. You can convince yourself that even for simple graphs, removal or addition of entries can lead to arbitrarily pathological situations, such as structures that look like cycles in the accessibility list but that do not correspond to any possible cycle in a graph. Such pathologies pose challenges for any reconstruction algorithm.

There are two approaches to address this problem. One approach is to use only the most reliable data. For example, in a micro array experiment assessing the effect of a gene deletion on the mRNA expression state of other genes, some genes change expression to a greater extent than others. One could only use those genes whose expression state has changed beyond a pre-specified threshold, according to some suitable statistical criterion. However, being excessively conservative would lead to failure to identify some important interactions.

The second approach regards heuristic modifications to the algorithm. For example, because no cycle in a graph of *n* nodes can be longer than *n* edges, one might set a limit to the recursion depth of the algorithm to prevent infinite recursion in case an accessibility list contains spurious cycles. If that recursion depth is exceeded for at least one node, one can generate the graph's condensation (Wagner 2001b) iteratively, until the reconstruction algorithm (Fig. 3) yields an adjacency list. Even with such heuristic modifications, however, it is almost certain that one can construct arbitrarily pathological "accessibility" lists for which any heuristic modification would fail. If and how the algorithm should be modified depends on the error structure of the empirical data. This error structure, in turn, may depend on the notion of gene activity and also on the kind of perturbation used. As large-scale genetic perturbation data is accumulating, the statistical nature of these errors will become clear. I will thus postpone a more rigorous treatment of this problem.

To provide a crude assessment of algorithmic robustness to defects in an accessibility list, I analyzed robustness of the algorithm in Figure 3 to missing and to added entries this list. I did so by (i) generating a random network of a pre-specified number of nodes and edges along with its accessibility list, and (ii) eliminating or adding a fraction of nodes to the accessibility list at random. I then applied the algorithm from Fig. 3 to the list thus generated and determined the number of edges, corresponding to node pairs u,v, with $p(u,v)=1$, that the algorithm identifies correctly. More specifically, I determined the fraction f_- of edges in G_{pars} that the algorithm does not identify as such (false negative edges), and the fraction of edges f_+ the algorithm finds but that are not part of G_{pars} (false positive edges).

A complementary way of viewing algorithmic robustness is to analyze how much the algorithm "enriches" direct interactions in a reconstructed adjacency list. Let C be the number of entries in the accessibility list of a graph, and A the number of entries in the adjacency list. Then, a fraction A/C of entries in the accessibility list corresponds to direct regulatory interactions in *Acc*. One can think of A/C metaphorically as the "concentration" of direct interactions among all detected interactions. For instance, for the graph in Figure 5 with 610 edges, there are 37889 entries of the accessibility list. Thus, only 610/37889, or about one 62nd of all entries are direct interactions. Perfect network reconstruction would enrich direct interactions by a factor 62. With flawed data, the algorithm may not identify all direct interactions, but the adjacency list it produces may still be enriched for direct interactions. The quantity

$$A(1-f_-)/A(1+f_+)=(1-f_-)/(1+f_+)$$

is the ratio of correctly identified direct interactions, $A(1-f_-)$, to the sum of the total number of entries of *Adj* and the number of false positive interactions, Af_+ . It is at most one, in the case of perfect network reconstruction. As a measure for how much an algorithm enriches for direct interactions, I define

$$E=(C\ (1-f_-))/(A\ (1+f_+))$$

E attains its maximally possible value, *C/A*, for perfect network reconstruction. An *E* of 10 means that the "concentration" of direct interactions in a reconstructed adjacency list is 10 times greater than in the accessibility list of the same network.

Figures 4 and 5 show robustness of the algorithm to deleted and added accessibilities, respectively. The algorithm generates fewer false negative than false positive edges for a given fraction of deleted (Figures 4A vs. 4B) or added (Figures 5A vs. 5B) accessibilities. Second, the algorithm is in general more sensitive to added accessibilities, in that both f_- (4A vs 5A) and f_+ (4B vs. 4C) are greater if the same fraction of edges is added as opposed to deleted. Third, f_+ depends more strongly on the interaction density in the network than f_- . An accessibility list derived from large scale gene perturbations in yeast suggests that transcriptional regulatory networks may be sparsely connected. That is, their edge density may fall between the two more sparsely connected graphs analyzed here (Wagner 2001a). If true in general, this would be reassuring, because the lower the edge density in a graph, the greater the robustness of the algorithm.

The relation of *E* to the fraction of deleted and added entries of an accessibility list is shown in Figures 4C and 5C. Take the above example with 37889 entries of the accessibility list. For perfect network reconstruction (dotted line in Figure 4C and 5C), $E\approx 62$. If the size of the accessibility list is decreased or increased by 10%, the fractions of direct interactions in the reconstructed adjacency list are still 30-fold and 10-fold greater, respectively, than in the original accessibility list (Figure 5C). Thus, even for substantially flawed data, leading to a sizable fraction of false negative and false positive edges, the algorithm may still significantly enrich direct interactions.

Discussion

Limitations. The inability to resolve cycles is a limitation of experimental data rather than of any network reconstruction method. It raises two questions. First, how abundant are such cycles? If genetic networks involving only one kind of gene activity are sparse, as I have suggested recently (Wagner 2001a) cycles may not be as abundant as one might think. However, the question how many transcription factor genes indirectly influence their own transcriptional state, or how many protein kinases indirectly influence their own phosphorylation state can currently not be answered rigorously at this point. (I emphasize indirect feedback, because direct self-regulation can be immediately inferred from a suitably designed experiment.) The second question is how to design experiments to resolve cycles. The obvious but naïve answer is to study the effect of double mutations on a network: one mutations breaks a cycle, and the remaining (linear) path can be analyzed by single mutations. However, this may work only for very sparse network components. It is easy to think of densely connected subnets, where even multiple mutations may not resolve cyclic structures. To identify the limits of genetic perturbation analysis in network reconstruction is a promising area of further research.

A second limitation is that the notion of gene activity constrains the information one can obtain on a network's structure. This is again an experimental limitation that can only be overcome if multiple aspects of gene activity, such as transcription, phosphorylation state, or methylation state, can be measured on a large scale.

A third shortcoming is that the algorithm requires more data than conventional methods using gene expression correlations (Eisen et al. 1998; Tavazoie et al. 1999). Fortunately, large-scale genetic perturbation experiments may provide such data for several organisms (Pennisi 1998; Somerville and Somerville 1999; Spradling et al. 1999; Winzeler et al. 1999). The most effective strategy to reconstruct genetic networks may in fact be a combination of the correlative and perturbative approaches. Available correlation methods can be used to identify groups of genes likely to participate in a particular process of interest. These genes can then be systematically perturbed, and the resulting data can be used to reconstruct regulatory interactions. Conventional biochemical methods can then be applied to study subnets of interest in greater detail.

Fourth, there are many networks consistent with any given list of perturbation effects and the algorithm only reconstructs one of them. It reconstructs the simplest or most parsimonious network, the network that contains the fewest regulatory interactions. There can be no guarantee that this most parsimonious network reflects the actual structure of a genetic network, which might have vastly more interactions. However, it is not likely that a genetic network would maintain vastly more regulatory interactions than necessary to exert its function. The reason is that such unnecessary interactions are likely to disappear rapidly through degenerative mutations.

Fifth, as one would expect, the algorithm's robustness to experimental errors is limited. My analysis of its robustness has two implications. First, the algorithm is generally more sensitive to false positive than to false negative gene activity changes. Thus, it is better to be statistically conservative when deciding which genes have changed their activity state after perturbation. In terms of micro array analyses, this means that it is better to apply conservative thresholds for expression ratios when deciding what genes were affected by a genetic perturbation. Second, when viewed as a tool to enrich direct interactions in an accessibility list, the algorithm is useful even if the quality of gene activity measurements is substantially compromised.

The next steps. Computational complexity of the algorithm is low, and the algorithm is so fast that a genetic network with as many genes as the human genome could be reconstructed in little time on a desktop workstation. While efforts to improve algorithmic efficiency may thus lead to marginal returns, significant improvement in other areas is possible. These include, (i) a rigorous, perhaps analytical quantification of robustness to experimental noise, and heuristic modifications to improve robustness, (ii) inclusion of quantitative information on gene activity changes, once such information becomes available, (iii) integration of different kinds of gene activity (transcription, phosphorylation) to obtain a less fragmented and more comprehensive network picture.

Acknowledgments. I would like to thank the Santa Fe Institute for its continued support of my research program.

Appendix

Below is a perl implementation of the algorithm from Figure 3. When given the accessibility list in Figure 1C, i.e., the list of adjacent node pairs (*p(u,v)=1*), it produces as part of its output the adjacency list of Figure 1B, except for node pairs (3,2), (6,12), and (13,8), which are shortcuts of the graph in Figure 1A. The algorithm's structure follows exactly that of the pseudocode shown in Fig. 3 and explained in the main text, with one difference: only one data structure is used to represent both accessibility list and path lengths between nodes. This structure is a two-dimensional hashing array `acc`. The accessibility list needs to be read into this array (input and output are not shown here) such that after input but before the algorithm is run `$acc{$u}{$v}`=1 if gene `$v` is accessible from gene `$u`. This implies that `$acc{$u}{$v}` is also *defined*. If gene `$v` is not accessible from gene `$u`, then `$acc{$u}{$v}` must be *undefined*. The algorithm tests whether `$v` is accessible from `$u` by testing whether the corresponding entry of `acc` is defined, and it performs the path length maximization on the integer entries of `acc`. Initiating `$acc{$u}{$v}` to one for accessible nodes makes step 12 of Figure 3 unnecessary. After execution, the defined entries of `acc` indicate the maximum path lengths between two nodes. Visited nodes are tracked by a one-dimensional hashing array `%visited` which needs to be initialized as '`%visited=();`' before execution. I do not claim that this is the most efficient or most elegant implementation.

```
foreach $u (sort keys %acc)   {
    if($visited{$u}!=1) {
          MAXPATH($u);
    }
}

sub MAXPATH       {
      # declare calling variable as local
      my $u=@_[0];

      #loop 1 of MAXPATH
      foreach $v (keys %{$acc{$u}}) {
            if($visited{$v}!=1)     {
                  if (scalar(keys %{$acc{$v}}) == 0)  {
                        $visited{$v}=1;
                  }
                  else  {
                        MAXPATH($v);
                  }
            }
      }

      #loop 2 of MAXPATH
      foreach $v (keys %{$acc{$u}}  )     {
            foreach $w (keys %{$acc{$v}}) {
                  $tmp=$acc{$u}{$v} + $acc{$v}{$w};
                  if ($tmp  > $acc{$u}{$w} )    {
                        $acc{$u}{$w}=$tmp;
                  }
            }
      }
      $acc{$u}{$v}=0;
      $visited{$u}=1;
}# end subroutine MAXPATH
```

Figure Captions

Fig. 1: Reconstructing a network from a list of direct and indirect perturbation effects. Panel A shows a genetic network represented as a graph whose nodes correspond to genes numbered from 0 through 20. Two genes are connected by an arrow if they influence each other's activity directly. Panel B shows the adjacency list of this network. It completely defines the network. For each gene *i* (to the left of the colon), it is the list of all genes directly influenced by *i*. Panel C shows the list of direct and indirect perturbation effects for the network in A. When perturbing the activity of a gene *i* in the network, all genes whose activities are directly or indirectly influenced by this gene will change their activity. For each perturbed gene, one gets the list shown in C by following all paths leaving a gene along the arrows.

Fig. 2. Single gene perturbations can not resolve the order of genes in a cycle. Panel A shows two cycles with their respective adjacency list. The order of genes in these cycles is different, yet they generate the same accessibility list, which is shown in B. Panels C and D show a cyclic and an acyclic network, respectively, that differ by only one edge, the edge between nodes 13 and 4.

Fig. 3. A recursive algorithm to reconstruct the maximum path length between two genes. see text for details.

Fig. 4.: Quality of network reconstruction with unidentified perturbation effects. Results are shown for three random graphs of 500 nodes and 250 edges (circles), 500 edges (squares), or 750 edges (diamonds), from which edges are removed until each graph is rendered acyclic (Mehlhorn and Naher 1999). After removal of these edges, the resulting three acyclic graphs have 250, 494 and 624 edges left, respectively. For each of these networks, a pre-specified fraction of entries was then eliminated at random from the accessibility list. The fraction of remaining entries is shown on the abscissa of each panel. The algorithm from Fig.3 was then applied to the changed accessibility list, and the resulting adjacency list was then compared to that of the maximally parsimonious graph of the original accessibility list. For the reconstructed network, **A)** shows the fraction of false negative edges, **B)** shows the fraction of false positive edges, and **C)** shows by how many fold direct interactions are enriched (see text) in the reconstructed adjacency list, relative to the original accessibility list. A logarithmic scale is chosen in C) merely to fit all data

points conveniently on one graph. The dotted horizontal lines indicate the maximally possible enrichment, that is, for perfect network reconstruction.

Fig. 5.: Quality of network reconstruction with spurious perturbation effects. Results are shown for three random graphs of 500 nodes and 250 edges (circles), 500 edges (squares), or 750 edges (diamonds), from which edges are removed until each network is rendered acyclic (Mehlhorn and Naher 1999). After removal of these edges, the resulting three acyclic graphs have 250, 494 and 610 edges left, respectively. For each of these networks, a pre-specified fraction of entries was then added at random to the accessibility list. For each added entry *u*, it was verified whether adding *u* as an edge would create a cycle in the graph. If so, a new *u* was chosen at random, until one was found that did not create a cycle. The fraction of entries thus added is shown on the abscissa of each panel. The algorithm from Fig.3 was then applied to the changed accessibility list, and the resulting adjacency list was then compared to that of the maximally parsimonious graph of the original accessibility list. For the reconstructed network, **A)** shows the fraction of false negative edges, **B)** shows the fraction of false positive edges, and **C)** shows by how many fold direct interactions are enriched (see text) in the reconstructed adjacency list, relative to the original accessibility list. A logarithmic scale is chosen in C) merely to fit all data points conveniently on one graph. The dotted horizontal lines indicate the maximally possible enrichment, that is, for perfect network reconstruction.

References

DeRisi JL, Iyer VR, Brown PO (1997) Exploring the metabolic and genetic control of gene expression on a genomic scale. Science 278:680-686

Eisen MB, Spellman PT, Brown PO, Botstein D (1998) Cluster analysis and display of genome-wide expression patterns. Proceedings of the National Academy of Sciences of the United States of America 95:14863-14868

Fell D, Wagner A (2000) The small world of metabolism. Nature Biotechnology 18:1121-1122

Harary F (1969) Graph theory. Addison-Wesley, Reading, Massachusetts

Hughes TR, Marton MJ, Jones AR, Roberts CJ, Stoughton R, Armour CD, Bennett HA, Coffey E, Dai HY, He YDD, Kidd MJ, King AM, Meyer MR, Slade D, Lum PY, Stepaniants SB, Shoemaker DD, Gachotte D, Chakraburtty K, Simon J, Bard M, Friend SH (2000) Functional discovery via a compendium of expression profiles. Cell ; 102:109-126

Jeong H, Tombor, B. Albert, R. Oltvai, Z.N., Barabasi, A.L. (2000) The large-scale organization of metabolic networks. Nature. 407:651-654

Lockhart D, Winzeler EA (2000) Genomics, gene expression and DNA arrays. Nature 405:827-836

Mehlhorn K, Naher S (1999) LEDA: A platform for combinatorial and geometric computing. Cambridge University Press, Cambridge, UK

Pennisi E (1998) Worming secrets from C-elegans genome. Science ; 282:1972-1974

Somerville C, Somerville S (1999) Plant functional genomics. Science ; 285:380-383

Spradling AC, Stern D, Beaton A, Rhem EJ, Laverty T, Mozden N, Misra S, Rubin GM (1999) The Berkeley Drosophila Genome Project gene disruption project: Single P-element insertions mutating 25% of vital drosophila genes. Genetics ; 153:135-177

Tavazoie S, Hughes JD, Campbell MJ, Cho RJ, Church GM (1999) Systematic determination of genetic network architecture. Nature Genetics 22:281-285

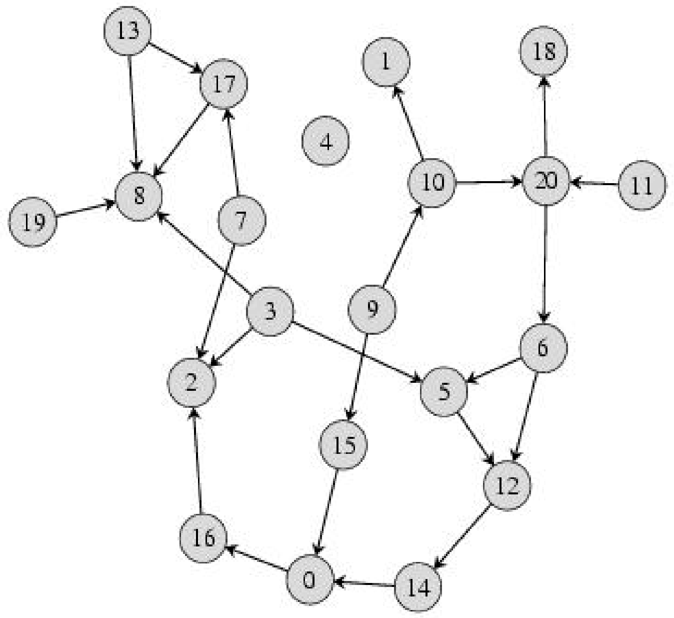
Wagner A (2001a) Genetic networks are sparse: estimates based on a large-scale genetic perturbation experiment. (submitted).

Wagner A (2001b) How to reconstruct a genetic network from n single-gene perturbations in fewer than n^2 easy steps (submitted).

Wagner A (2001c) A rapidly evolving protein interaction network with many duplicate genes. Molecular Biology and Evolution (in press).

Winzeler EA, Shoemaker DD, Astromoff A, Liang H, Anderson K, Andre B, Bangham R, Benito R, Boeke JD, Bussey H, Chu AM, Connelly C, Davis K, Dietrich F, Dow SW, M EL, Foury F, Friend SH, Gentalen E, Giaever G, Hegemann JH, Jones T, Laub M, Liao H, Liebundguth N, Lockhart DJ, LucauDanila A, Lussier M, N MR, Menard P, Mittmann M, Pai C, Rebischung C, Revuelta JL, Riles L, Roberts CJ, RossMacDonald P, Scherens B, Snyder M, SookhaiMahadeo S, Storms RK, Veronneau S, Voet M, Volckaert G, Ward TR, Wysocki R, Yen GS, Yu KX, Zimmermann K, Philippsen P, Johnston M, Davis RW (1999) Functional characterization of the S. cerevisiae genome by gene deletion and parallel analysis. Science 285:901-906

A



B

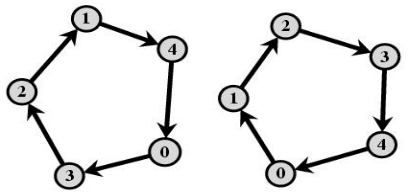
0:	**16**
1:	
2:	
3:	***2* 5 8**
4:	
5:	**12**
6:	**5 *12***
7:	**2 17**
8:	
9:	**10 15**
10:	**1 20**
11:	**20**
12:	**14**
13:	***8* 17**
14:	**0**
15:	**0**
16:	**2**
17:	**8**
18:	
19:	**8**
20:	**6 18**

C

0:	**2 16**
1:	
2:	
3:	**0 2 5 8 12 14 16**
4:	
5:	**0 2 12 14 16**
6:	**0 2 5 12 14 16**
7:	**2 8 17**
8:	
9:	**0 1 2 5 6 10 12 14 15 16 18 20**
10:	**0 1 2 5 6 12 14 16 18 20**
11:	**0 2 5 6 12 14 16 18 20**
12:	**0 2 14 16**
13:	**8 17**
14:	**0 2 16**
15:	**0 2 16**
16:	**2**
17:	**8**
18:	
19:	**8**
20:	**0 2 5 6 12 14 16 18**

Fig. 1

A)

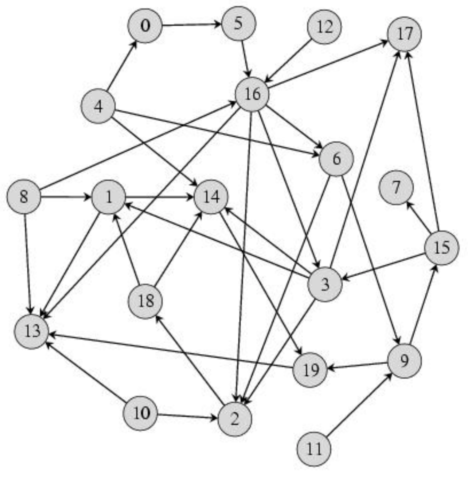


0:	3	0:	1
1:	4	1:	2
2:	1	2:	3
3:	2	3:	4
4:	0	4:	0

B)

0:	1 2 3 4
1:	0 2 3 4
2:	0 1 3 4
3:	0 1 2 4
4:	0 1 2 3

C)



D)

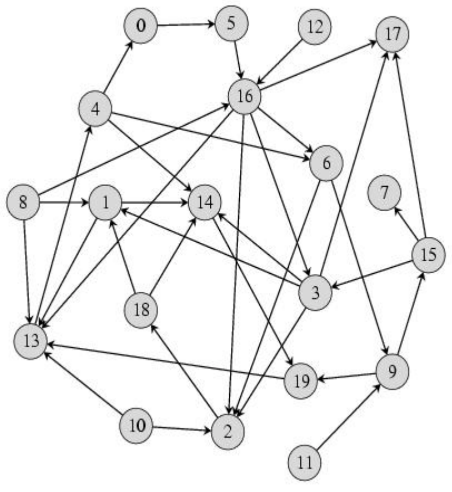
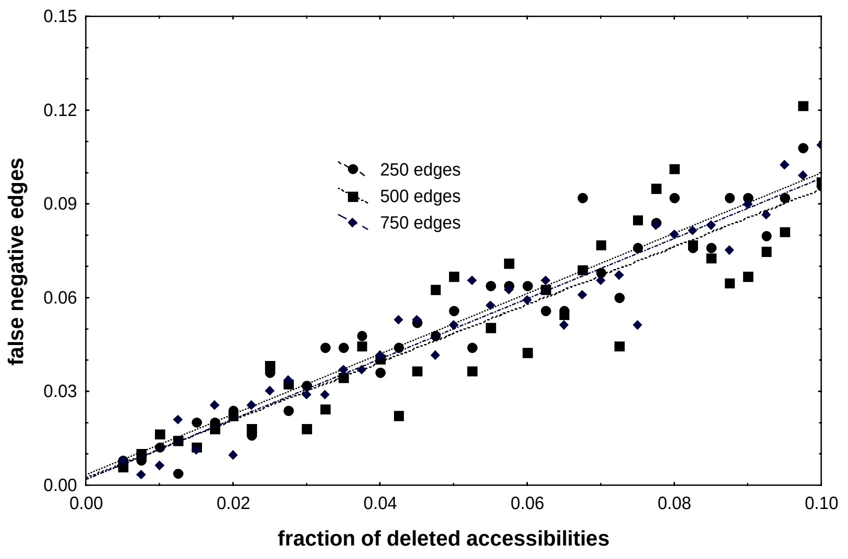


Fig. 2

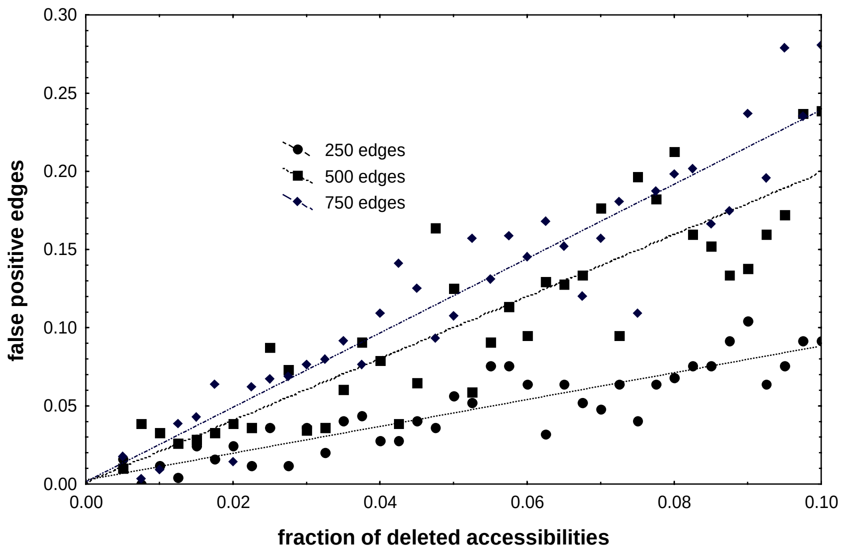
```
for all nodes $u$ of G
    if node $u$ has not been visited
        call MAXPATH($u$)
    end if

MAXPATH($u$)
    for all nodes $v \in Acc(u)$
        if $Acc(v)=\emptyset$
            declare $v$ as visited.
        else
            call MAXPATH($v$)
        end if
        $p(u,v)=1$

    for all nodes $v \in Acc(u)$
        for all nodes $w \in Acc(v)$
              $p(u,w)=max(p(u,w), 1+p(v,w))$
$p(u,u)=0$
declare node $u$ as visited
end MAXPATH(u)
```

Fig. 3

A)


250 edges
500 edges
750 edges
false negative edges
fraction of deleted accessibilities
0.00
0.03
0.06
0.09
0.12
0.15
0.00
0.02
0.04
0.06
0.08
0.10

B)


250 edges
500 edges
750 edges
false positive edges
fraction of deleted accessibilities
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.00
0.02
0.04
0.06
0.08
0.10

C)

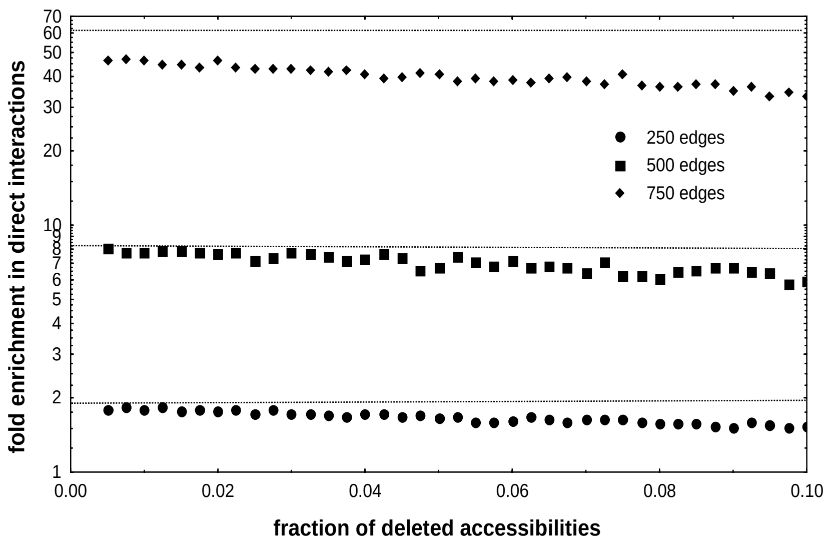
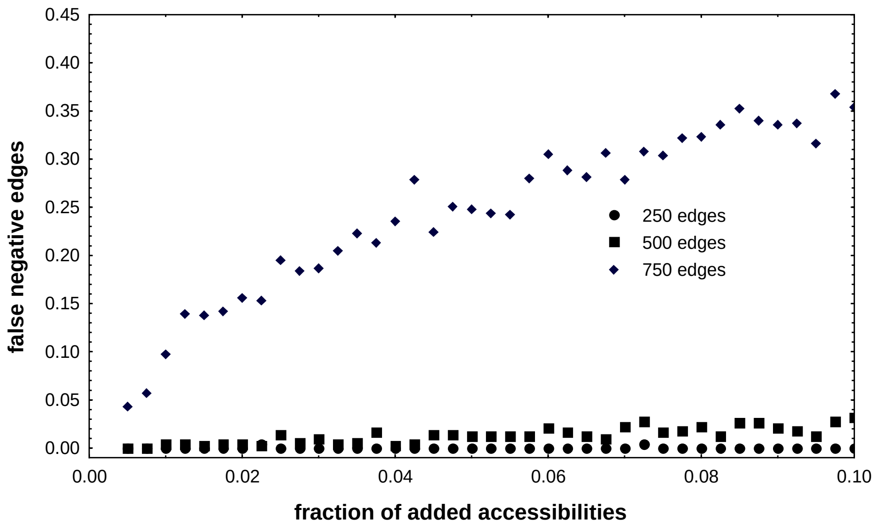

250 edges
500 edges
750 edges
fold enrichment in direct interactions
fraction of deleted accessibilities
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
0.00
0.02
0.04
0.06
0.08
0.10

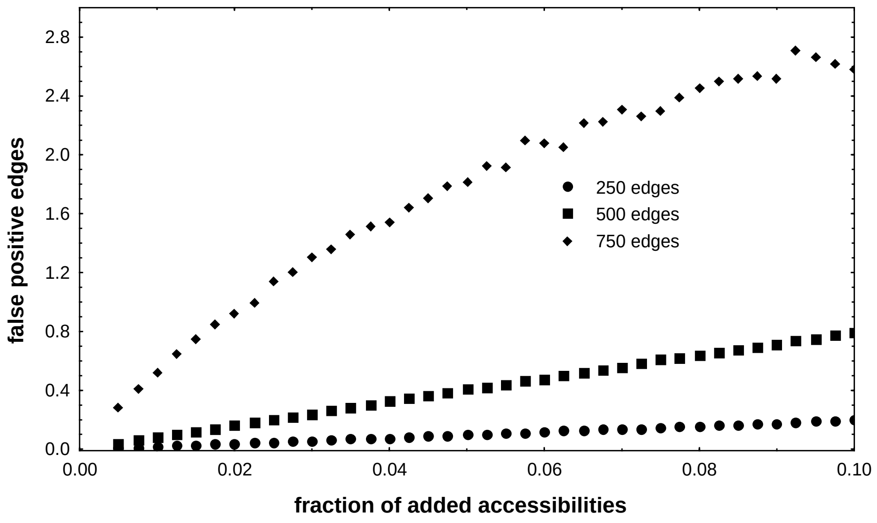
Fig. 4

A)



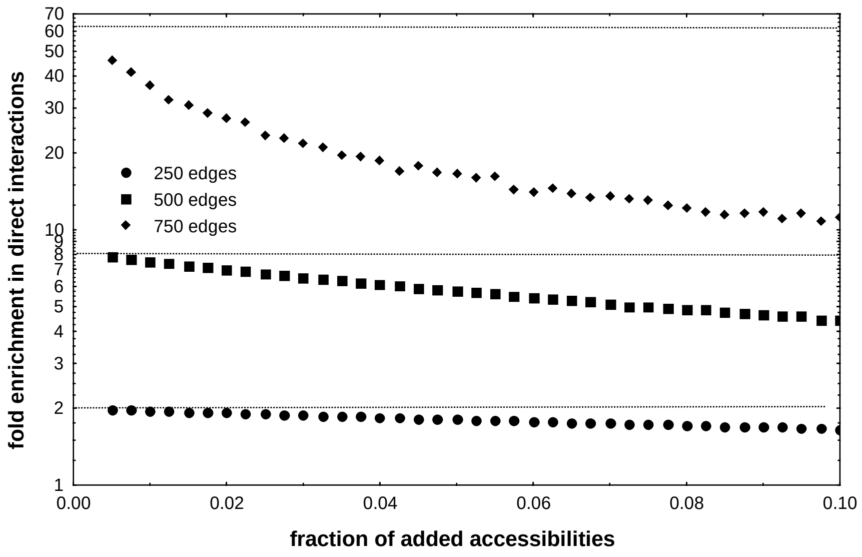
false negative edges
fraction of added accessibilities
● 250 edges
■ 500 edges
◆ 750 edges
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0.45
0.00
0.02
0.04
0.06
0.08
0.10

B)



false positive edges
fraction of added accessibilities
● 250 edges
■ 500 edges
◆ 750 edges
0.0
0.4
0.8
1.2
1.6
2.0
2.4
2.8
0.00
0.02
0.04
0.06
0.08
0.10

C)



fold enrichment in direct interactions
fraction of added accessibilities
● 250 edges
■ 500 edges
◆ 750 edges
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
0.00
0.02
0.04
0.06
0.08
0.10

Fig. 5