BIOINFORMATICS

Vol. 17 no. 12 2001
Pages 1183–1197

How to reconstruct a large genetic network from n gene perturbations in fewer than n^2 easy steps

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, USA

Received on February 2, 2001; revised on April 30, 2001; accepted on May 28, 2001

ABSTRACT

Motivation: The reconstruction of genetic networks is the holy grail of functional genomics. Its core task is to identify the causal structure of a gene network, that is, to distinguish direct from indirect regulatory interactions among gene products. In other words, to reconstruct a genetic network is to identify, for each network gene, which other genes and their activity the gene influences directly. Crucial to this task are perturbations of gene activity. Genomic technology permits large-scale experiments perturbing the activity of many genes and assessing the effect of each perturbation on all other genes in a genome. However, such experiments cannot distinguish between direct and indirect effects of a genetic perturbation.

Results: I present an algorithm to reconstruct direct regulatory interactions in gene networks from the results of gene perturbation experiments. The algorithm is based on a graph representation of genetic networks and applies to networks of arbitrary size and complexity. Algorithmic complexity in both storage and time is low, less than $O(n^2)$. In practice, the algorithm can reconstruct networks of several thousand genes in mere CPU seconds on a desktop workstation.

Availability: A perl implementation of the algorithm is given in the Appendix.

Contact: wagnera@unm.edu

INTRODUCTION

Most techniques to analyze genetic networks monitor changes in the expression of many genes under changing environmental conditions, in different physiological stages, or in different genetic backgrounds. They then identify genes with similar expression in these different situations, and cluster them according to this similarity (DeRisi *et al.*, 1997; Eisen *et al.*, 1998; Tavazoie *et al.*, 1999). The underlying assumption is that genes with similar expression patterns participate in similar biological processes. While not unreasonable, this approach suffers from the problem that correlated expression can only point to regulatory interactions between genes. It cannot be used to infer such interactions. By a regulatory interaction between two genes I mean that one gene directly influences the expression level of the other gene. And if we are to understand the structure and function of genetic networks, this is perhaps the most fundamental question to answer: which genes in a network influence the activity of which other genes directly. I here present an algorithm that can answer this question for arbitrarily large networks. Not surprisingly, its ability 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 correlative methods. Specifically, it requires perturbation of many genes in the network. Large-scale perturbation data of this sort is now becoming available (Bouche and Bouchez, 2001; Fraser *et al.*, 2000; Gonczy *et al.*, 2000; Hughes *et al.*, 2000; Spradling *et al.*, 1999).

Concepts

To begin with, a few definitions are in order. 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 influence the activity of other genes. What, then, is *gene activity*? For my purpose, gene activity can include many different things. Most definitions revolve around gene expression, whether a gene is expressed or not, as mRNA or as protein. At a higher level of resolution, the amount of mRNA or protein expressed might be important as well. However, there is more to gene activity than just expression. For instance, one might consider differences in post-transcriptional regulation, such as differential splicing, or post-translational modification, such as phosphorylation. It is well known that the activity of many gene products is regulated by phosphorylation, and gene products that phosphorylate or dephosphorylate other gene products are key regulators inside any living cell. Another aspect of gene activity is the methylation state of genes, which is related to gene silencing. And there are gene products that are involved in changing the methylation state of other genes. These few examples show that there are endlessly many possible ways of defining gene activity. In principal the approach proposed here applies to all notions of gene activity, as long as they are used consistently and defined clearly within an experimental context.

Next, what is a *genetic perturbation*? For my purpose, it is an experimental manipulation of gene activity by manipulating either a gene itself or its product. Such 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. While mutations are not usually thought of as manipulations of gene activity, I choose to view them as such. Clearly, if I mutate a gene so that the cell cannot produce the gene product, I have affected the activity state of the gene.

Network reconstruction: direct and indirect effects

When manipulating a gene and finding that this manipulation affects the activity of other genes, the question often arises as to whether this is caused by a direct or indirect interaction. For example, when overexpressing a transcription factor X, I might find that the expression level of genes A and B changes. 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. For the purpose of this paper, I will define the task of network reconstruction as follows. To reconstruct a genetic network is to identify all direct effects of network genes on one another's activity within the limits of experimental resolution.

The important issue of experimental resolution is best illustrated by an example. Consider the hypothetical example of a biochemical pathway shown in Figure 1a. A constitutively expressed transcription factor produced by gene 1 induces expression of gene 2, whose product is a protein kinase. This protein kinase phosphorylates a protein phosphatase, the product of gene 3, an event that activates the phosphatase. The phosphatase in turn dephosphorylates a transcription factor, the product of gene 4. Dephosphorylation activates the transcription factor, which binds to and induces expression of gene 5, whose function is unspecified.

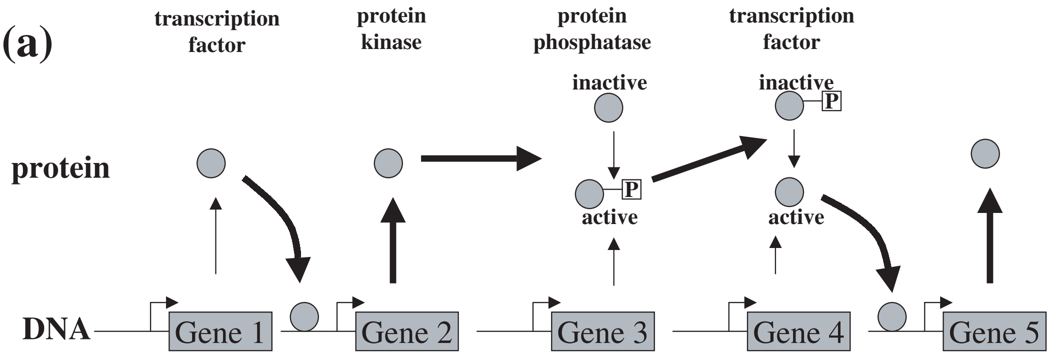
Now consider a hypothetical series of five different experiments, deleting each of the five genes involved in this pathway. For each of these five perturbations, we measure changes in gene activity. The notion of gene activity I choose to use here is that of the mRNA expression level. The results of these five hypothetical experiments are shown in Figure 1b in a format that I will use throughout. Each line contains the results of one genetic perturbation. The leftmost symbol stands for the gene whose activity was manipulated, followed by a colon. To the right-hand side of the colon, the names of genes appear whose activity was influenced by that particular manipulation. In the experiment whose result is shown in the first line of Figure 1b, the activities (mRNA level) of genes 2 and 5 were affected by deleting gene 1. Gene 2 was affected because the product of gene 1 is required for transcription of gene 2. Gene 5 was affected, because, indirectly, the expression of gene 2 influences the phosphorylation state and the activity of the transcription factor produced by gene 4. When the product of gene 2 is absent, then the product of gene 4 will be inactive, and gene 5 will not be expressed. As opposed to its effect on the activities of genes 2 and 5, deletion of gene 1 does not affect the activity of genes 3 and 4. Deletion of gene 1 will only affect the phosphorylation state of genes 3 and 4, but not their mRNA expression, because they are constitutively expressed.

Figure 1c shows a situation where the genetic perturbation is the same, but where the phosphorylation state is used as a measure of gene activity. Manipulation of gene 1 now does not affect the measured activity of genes 2 and 5, but it does affect the activities of genes 3 and 4.

What can we expect a network reconstruction algorithm to achieve? Clearly, for the result of the thought experiment shown in Figure 1b, we would expect that the algorithm identifies the order in which the genes influence each others expression state as G1 → G2 → G5. For Figure 1c, we would expect that the algorithm identifies the order in which the genes influence each other's phosphorylation states as G1 → G3 → G4.

No algorithm can be expected to say anything beyond that. This is a limitation of the data itself and not of any algorithm. Thus, one has to be very clear about the measure of gene activity and the genetic perturbation used, and what interpretations one can extract from them. It is possible to superimpose results such as those shown in Figure 1b and c, but this will not be my focus here. My focus will be to reconstruct networks vastly more complex and reticulate than the pathway shown in Figure 1a, from data like that shown in Figure 1b or c.

In sum, an algorithm to reconstruct a genetic network from perturbation data should be able to distinguish direct from indirect regulatory effects. This is by no means a weak statement, although it may seem so at first glance. Consider a series of experiments in which the activity of every single gene in an organism is manipulated. (For instance, non-essential genes can be deleted, and for essential genes one might construct conditional mutants.) The effect on mRNA expression of all other genes is measured separately for each mutant. The result is a list similar to that shown in Figure1b, but for thousands of genes. An algorithm reconstructing this network would



(b)
Aspect of gene activity: mRNA expression
Genetic perturbation: gene deletion

G1:	**G2, G5**
G2:	**G5**
G3:	**G5**
G4:	**G5**
G5:	

(c)
Aspect of gene activity: phosphorylation state
Genetic perturbation: gene deletion

G1:	**G3, G4**
G2:	**G3, G4**
G3:	**G4**
G4:	
G5:	

Fig. 1. The importance of specifiying gene activity when reconstructing genetic networks. (a) A hypothetical biochemical pathway involving two transcription factors, a protein kinase, and a protein phosphatase, as well as the genes encoding them. See text for details. (b) Shown is a list of perturbation effects for each of the five genes in (a), when perturbing individual genes by deleting them, and when using mRNA expression level as an indicator of gene activity. The left-most symbol in each line stands for the perturbed gene. To the right of each colon is a list of genes whose activity is affected by the perturbation. (c) Analogous to (b) but for a different notion of gene activity (phosphorylation state).

be able to identify all genes regulated directly by any transcription factor encoded in this organism's genome. The significance of extracting such information cannot be overestimated, given the importance of transcriptional regulation in any biological process.

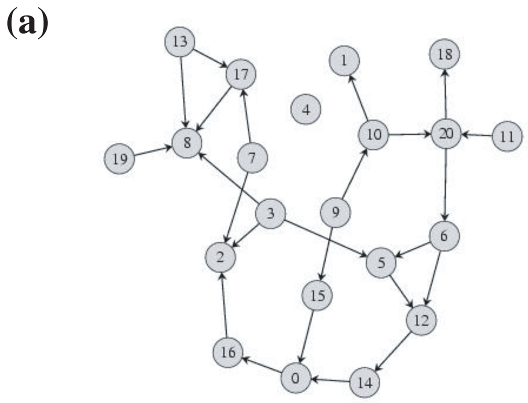
RESULTS

Graph theoretical framework

As the previous examples indicated, I will be largely concerned with qualitative information on gene interaction. That is, when I manipulate the activity of one gene, what other genes are influenced in their activity? Quantitative information, such as whether an interaction is activating or repressing, or the strength of the interaction, can be incorporated into this approach, but I will leave that to a future contribution.

The qualitative information I am considering here 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 the graph representation of a genetic network that I will use here, the nodes of the graph correspond to genes, and two genes, say genes 1 and 2, are connected by a directed edge (an arrow, $1 \rightarrow 2$) if gene 1 influences the activity of gene 2 directly. Figure 2a shows a graph representation of a hypothetical genetic network of 21 genes. For brevity, I will simply label genes by numbers throughout the paper, as in this figure. Figure 2b shows an alternative representation of the network shown in Figure 2a. For each gene i, it simply shows which genes' activity state the gene influences directly. In graph theory, a list like that shown in Figure 2b is called the *adjacency list* of the graph. I will denote it as $\mathrm{Adj}(G)$, and will refer to $\mathrm{Adj}(i)$ as the set of nodes (genes) adjacent to (directly influenced by) node i. One might also call it the list of nearest neighbors in the gene network, or the list of direct regulatory interactions. Importantly, the adjacency list completely defines the structure of a gene network.

When perturbing each gene in the network shown in Figure 2a, one would get the list of influences on the activities of other genes shown in Figure 2c. 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



(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. 2. Reconstructing a network from a list of direct and indirect perturbation effects. (a) shows a genetic network represented as a graph whose nodes correspond to genes numbered from 0 through 19. Two genes are connected by an arrow if they influence each other's activity directly. (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. (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. In this context, the task of network reconstruction is to generate a list such as that shown in (a) from a list of perturbation effects shown in (c).

a graph representation of the network in Figure 2a, one arrives at the list of direct and indirect causal interactions in Figure 2c by following all paths leaving a gene. That is, one follows all arrows emanating from the gene until one can go no further. In graph theory, the resulting list Acc(G) is called the *accessibility list* of the graph G, because it shows all nodes (genes) that can be accessed (influenced in their activity state) from a given gene 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*. I define Acc(i) as the set of nodes that can be reached from node i by following all paths of directed edges leaving i. Acc(G) then simply consists of the accessibility list for all nodes i, as shown in Figure 2c. (Notice that not every list like that shown in Figure 2c is the accessibility list of a graph. For example, there is no graph with the accessibility list $1:2$; $2:1$; $3:2$.)

Generating Figure 2c from Figure 2a or b is straightforward, although time consuming for a large network. The subject of this paper is the more difficult problem of reconstructing the network in Figure 2a from nothing but the list given in Figure 2c, and to do that automatically for very large networks of perhaps thousands of genes.

I will be using two additional mathematical representations of gene networks. The first of these is the *adjacency matrix* of a graph G, $A(G) = (a_{ij})$. $A(G)$ is an $n \times n$ square matrix, where n is the number of nodes (genes) in the graph. An element a_{ij} of this matrix is equal to one if and only if a directed edge exists from node i to node j. All other elements of the adjacency matrix are zero. Second, the *accessibility matrix* $P(G) = p_{ij}$ is also an $n \times n$ square matrix. An element p_{ij} is equal to one if and only if a path following directed edges exists from node i to node j. p_{ij} equals zero otherwise. Adjacency and accessibility matrices are the matrix equivalents of adjacency and accessibility lists.

Most real world graphs are sparse. That is, they have few outgoing edges per node. For such graphs, it is often more efficient to use lists rather than matrices for numerical operations. However, to state or prove theorems, matrix representations are sometimes more convenient.

In the following sections I will develop in two steps an algorithm to reconstruct a network from its accessibility list, as well as its mathematical foundation. First, I will restrict myself to graphs without cycles, where cycles are paths starting at a node and leading back to the same node. Graphs without cycles are called *acyclic graphs*. In the second step I will generalize to graphs with cycles. Before beginning, however, I have to address a problem germane to analyzing any kind of experimental data. It is the problem that there are usually many possible and internally consistent reconstructions of a study system from experimental data. The preferred one is usually a simple or parsimonious one, by some suitable definition. Genetic network reconstruction is no exception to this rule.

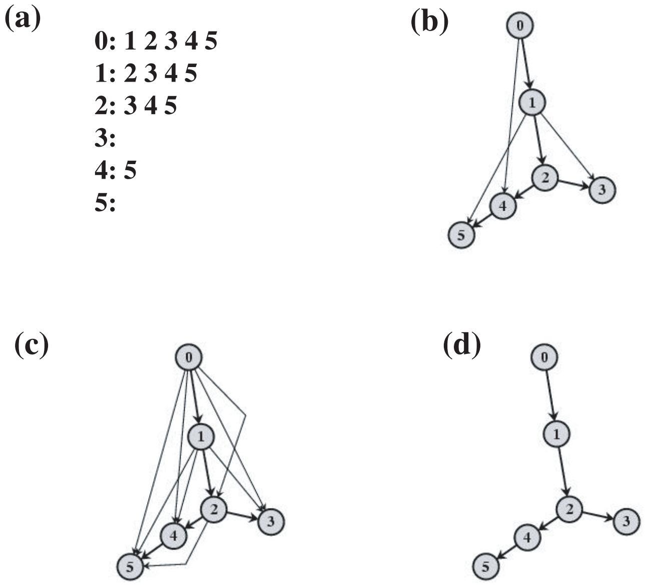


Fig. 3. The most parsimonious graph is the graph with the fewest edges consistent with a given accessibility list. (a) Shows the accessibility Acc of an acyclic graph. (b)–(d) Show graphs that have this accessibility list. The graph shown in (d) is the most parsimonious graph of Acc.

Occam's razor in network reconstruction

An acyclic directed graph defines its accessibility list, but the converse is not true. In general, if Acc is the accessibility list of a graph, there is more than one graph G with the same accessibility list. Figure 3 illustrates this with an example of three graphs (Figures 3b–d) with the same accessibility list Acc (Figure 3a). For an example that simple it is obvious that there is one graph (Figure 3d) that has Acc as its accessibility list and is simpler than all other graphs, in the sense that it has fewer edges. For more complicated graphs, it may not be so clear that there is always such a graph. In this section, I will thus prove that there exists always exactly one such graph. In terms of reconstructing a genetic network, this means that for any list of perturbation effects there exists exactly one genetic network G with fewer edges than any other network with the same list of perturbation effects.

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.*

I will call G_{pars} the *most parsimonious network* compatible with Acc. Before starting the proof, I need to introduce some terminology.

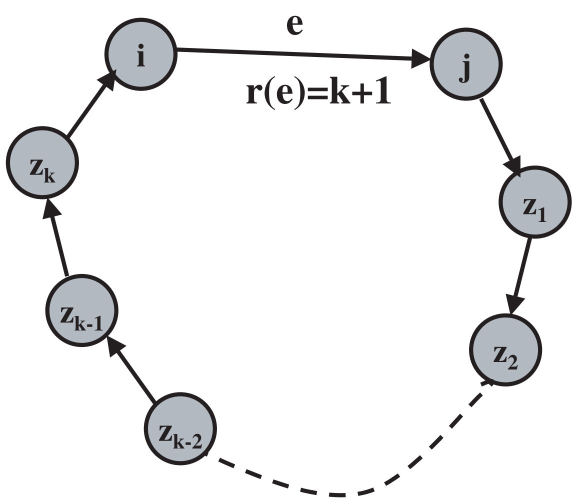


Fig. 4. A shortcut is an edge e connecting two nodes, i and j, that are also connected via a longer path of edges. The shortcut e shown here is a shortcut of range $k+1$. That is, when eliminating e, i and j are still connected by a path of length $k+1$.

DEFINITION. An accessibility list Acc and a digraph G are *compatible* if G has Acc as its accessibility list. Acc is the accessibility list *induced* by G.

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 $r := \infty$.

DEFINITION. An edge e with range $r \geqslant 2$ but $r \neq \infty$ is called a *shortcut*.

The last two definitions, in slightly different form and for undirected graphs, are due to Watts (1997). A shortcut of range $k+1$ is illustrated in Figure 4. A shortcut provides a shortest route between two nodes which are also connected by a longer path. Equipped with these definitions, I am now ready to prove Theorem 1.

LEMMA 1. *For any accessibility list* Acc *of a digraph, there exists a compatible graph* G_{pars} *that is free of shortcuts.*

PROOF. Assume there is no such graph G_{pars}. Without loss of generality, let G be a digraph inducing Acc. By assumption, G has a finite number of node pairs, $(x_1, y_1), \ldots, (x_n, y_n)$, each of which has the following property: There exists a directed edge e_i from x_i to y_i, as well as a path P_i from x_i to y_i of length greater than 1. Take, without loss of generality, the first pair of such nodes, (x_1, y_1), and generate from G a graph G^*

from which the edge e_1 is deleted. This graph has the same accessibility list as G, because x_1 and y_1 are still connected via P_1. By repeating this procedure with the remaining $(n-1)$ node pairs, you arrive at a graph $G^{(n*)}$ with the same accessibility list as G but without shortcuts. This is a contradiction to the assumption made in the beginning of the proof.

LEMMA 2. *Assume that* Acc *is the accessibility list of a digraph G. For each node x, the adjacency list* Adj(x) *of a shortcut-free graph G_{pars} compatible with* Acc *is a subset of the adjacency list* Adj(x) *of* any *graph compatible with* Acc.

PROOF. Assume that Lemma 2 is false. Consider then, without loss of generality, a shortcut-free graph G_{pars} that induces Acc, and some other graph G that also induces Acc. By assumption, G_{pars} contains at least one node x for which the following holds. Adj(x) of G_{pars} contains at least one node y that is not in the adjacency list Adj(x) of G. But because G and G_{pars} have the same accessibility list Acc, there must exist some path $x \to z_1 \to z_2 \to z_3 \to \cdots \to z_k \to y$ from x to y in G. Furthermore, because G and G_{pars} have the same accessibility list, z_1 must be accessible from x in G_{pars}, z_2 from z_1 in $G_{\text{pars}} \ldots$ and z_k from z_{k-1} in G_{pars}. That means that we can find a path in G_{pars}, however indirect, that runs from x to z_1, from z_1 to z_2, from z_2 to $z_3, \ldots$ from z_{k-1} to z_k, and from z_k to y. But now we have two paths in G_{pars}, one of length one, the edge e between x and y, and one involving the nodes z_1 through z_k. Thus, the edge $e(x \to y)$ is a shortcut, in contradiction to the assumption that G_{pars} is shortcut-free.

COROLLARY 1. *The shortcut-free graph G_{pars} compatible with* Acc *is a unique graph with the fewest edges among all graphs G compatible with* Acc.

The corollary follows immediately from Lemma 2. A complementary way of showing that G_{pars} is a minimal graph is to examine, first, the consequences of adding a node y to the adjacency list Adj(x) of some node x in G_{pars}. If $y \notin \text{Acc}(x)$ before the addition, now $y \in \text{Acc}(x)$, the accessibility list has changed, and the altered graph is no longer compatible with the original accessibility list. If, on the other hand, $y \in \text{Acc}(x)$ before the addition, then the addition has created a shortcut, so the graph is no longer shortcut-free. Second, what happens if you eliminate a node y from the adjacency list of any node x in G_{pars}? Then, y is no longer accessible from x, you have altered the accessibility list, and the resulting graph is no longer compatible with Acc. Assume that this was not so, that is, that y was still accessible from x. Then a path from x to y must have existed before the elimination of y from Adj(x). In that case, the edge from x to y was a shortcut, in contradiction to the assumption that G_{pars} is shortcut-free.

The algorithm

The network reconstruction algorithm takes the accessibility list Acc of an acyclic directed graph, and generates from it the adjacency list of the most parsimonious graph of Acc. It relies on two basic relations between accessibility and adjacency lists. The first is that for all nodes i of a graph, $\text{Adj}(i) \subseteq \text{Acc}(i)$. The second is formulated in the following theorem.

THEOREM 2. *Let* Acc(G) *be the accessibility list of an acyclic directed graph, G_{pars} its most parsimonious graph, and $V(G_{\text{pars}})$ the set of all nodes of G_{pars}. Then the following identity holds*

$$\forall i \in V(G_{\text{pars}})\, \text{Adj}\,(i) = \text{Acc}\,(i) \setminus \bigcup_{j \in \text{Acc}\,(i)} \text{Acc}\,(j). \quad (1)$$

In words, for each node i the adjacency list Adj(i) of the most parsimonious genetic network is equal to the accessibility list Acc(i) after removal of all nodes that are accessible from any node in Acc(i).

PROOF. I will first prove that every node in Adj(i) is also contained in the set defined by the right-hand side of equation (1). Let x be a node in Adj(i). This node is also in Acc(i). Now take, without loss of generality any node $j \in \text{Acc}(i)$. Could x be in Acc(j)? If x could be in Acc(j) then we could construct a path from i to j to x. But because x is also in Adj(i), there is also an edge from i to x. This is a contradiction to G_{pars} being shortcut-free. Thus, for no $j \in \text{Acc}(i)$ can x be in Acc(j). x is therefore also not an element of the union of all Acc(j) shown on the right-hand side of equation (1). Thus, subtracting this union from Acc(i) will not lead to the difference operator in equation (1) eliminating x from Acc(i). Thus x is contained in the set defined by the right-hand side of equation (1).

Next I will prove that every node in the set of the right-hand side of equation (1) is also in Adj(i). Let x be a node in the set of the right-hand side of equation (1). Because x is in the right-hand side of equation (1), x must *a fortiori* also be in Acc(i). That is, x is accessible from i. But x can not be accessible from any j that is accessible from i. For if it were, then x would also be in the union of all Acc(j). Then taking the complement of Acc(i) and this union would eliminate x from the set in the right-hand side of equation (1). In sum, x is accessible from i but not from any j accessible from i. Thus x must be adjacent to i.

The algorithm itself will use the following corollary to Theorem 2.

COROLLARY 2. *Let i, j, and k be any three pairwise different nodes of an acyclic directed shortcut-free graph G. If j is accessible from i, then no node k accessible from j is adjacent to i.*

```
1    for all nodes $i$ of G
2        $Adj(i)=Acc(i)$

3    for all nodes $i$ of G
4        if node $i$ has not been visited
5            call PRUNE_ACC($i$)
6        end if

7    PRUNE_ACC($i$)
8        for all nodes $j \in Acc(i)$
9            if $Acc(j)=\emptyset$
10               declare $j$ as visited.
11           else
12               call PRUNE_ACC($j$)
13           end if

14       for all nodes $j \in Acc(i)$
15           for all nodes $k \in Adj(j)$
16               if $k \in Acc(i)$
17                   delete $k$ from $Adj(i)$
18               end if
19   declare node $i$ as visited
20   end PRUNE_ACC(i)
```

Fig. 5. A recursive pruning algorithm to reconstruct the most parsimonious graph from an accessibility list. See text for details.

PROOF. Let j be a node accessible from node i. Assume that there is a node k accessible from j, such that k is adjacent to i. That is, $j \in \text{Acc}(i)$, $k \in \text{Acc}(j)$ and $k \in \text{Adj}(i)$. That k is accessible from j implies that there is a path of length at least one from j to k. For the same reason, there exists a path of length at least one connecting i to j. In sum, there must exist a path of length at least two from i to k. However, by assumption, there also exists a directed edge from i to k. Thus, the graph G can not be short-cut-free.

The algorithm itself takes the accessibility list of a graph and eliminates entries inconsistent with Theorem 2 and Corollary 2. It does so recursively until only the adjacency list of the shortcut-free graph is left. The algorithm is shown as pseudocode in Figure 5. 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 accessibility and adjacency list are represented by a two-dimensional hashing array.)

The algorithm (Figure 5) needs an accessibility list for each node i, $\text{Acc}(i)$, which would be obtained from gene perturbation data and subsequent gene activity measurements for a genetic network. In lines one and two (Figure 5), for each node i the adjacency list $\text{Adj}(i)$ is initialized as equal to the accessibility list. The algorithm will delete elements from this $\text{Adj}(i)$ until the adjacency list of the most parsimonious network of $\text{Acc}(G)$ is obtained.

The master loop in lines 3–6 cycles over all nodes of G, and calls the routine PRUNE_ACC for each node i. In the last statement of this routine (line 19) the calling node is declared as visited. I define a visited node as a node whose adjacency list $\text{Adj}(i)$ needs not be modified any further. This is the purpose of the conditional statement in the master loop (line 4), which skips over nodes that have already been visited.

Aside from storing Acc and Adj, the algorithm thus also needs to keep track of all visited nodes. In an actual implementation, Acc, Adj, and any data structure that keeps track of visited nodes would need to be either global variables or passed into the routine PRUNE_ACC, preferably by reference. In contrast, the calling node i needs to be a local variable because of the recursivity of PRUNE_ACC.

I will now explain the function PRUNE_ACC itself, which is the algorithm's core. It contains two loops. The first loop (lines 8–13) cycles over all nodes j accessible from the calling node i. If there exists a node accessible from j, then PRUNE_ACC is called from j. If no node is accessible from j, that is, if $\text{Acc}(j) = \emptyset$, then j is declared as visited. Because its accessibility list is empty, its adjacency list must be empty as well ($\text{Adj}(i) \subseteq \text{Acc}(i)$), and needs no further modification. Thus, through the first loop PRUNE_ACC calls itself recursively 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 PRUNE_ACC is $(n - 1)$ if G has n nodes. For any node i calling PRUNE_ACC, the number of nested calls is at most equal to the length of the longest path starting at i.

The second loop of PRUNE_ACC (lines 14–18) only starts once the algorithm has explored all nodes accessible from the calling node i, that is, as the function calls made during the first loop return. In the second loop the principle of Corollary 2 is applied. Specifically, the second loop cycles over all nodes j accessible from i in line 14. In a slight deviation from what Corollary 2 suggests, line 15 cycles not over all nodes $k \in \text{Acc}(j)$, but only over $k \in \text{Adj}(j)$. All nodes $k \in \text{Adj}(j)$ are deleted from $\text{Adj}(i)$ in lines 16–18. Cycling only over $k \in \text{Adj}(j)$ saves time, but does not compromise the requirement that all nodes $k \notin \text{Adj}(i)$ be removed, because line 14 covers all nodes j accessible from i. Because of the equality proven in Theorem 2, once this has been done, the adjacency list need not be modified further. This is why upon leaving this routine, the calling node is declared as visited. Notice also that if a node j with $\text{Acc}(j) = \emptyset$ is encountered, the loop in line 15 is not executed.

Computational and storage complexity

Both measures of algorithmic complexity are influenced 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 *et al.*, 2000; Wagner, 2001b).

During execution, each node accessible from a node j induces one recursive call of PRUNE_ACC, after which the node accessed from j is declared as visited. Thus, each entry of the accessibility list of a node is explored no more than once. However, line 15 of the algorithm (Figure 3) loops over all nodes adjacent to j. If $a = |\,\mathrm{Adj}(j)|$, on average, then overall computational complexity becomes $O(nka)$.

For practical matters, large scale experimental gene perturbations in the yeast *Saccharomyces cerevisiae* ($n \approx 6300$) suggests that $k < 50$ (Hughes *et al.*, 2000), $a \leqslant 1$ (Wagner, 2001a), and thus that $nka \ll n^2$ in that case. In practice, a network of 6300 nodes (the approximate number of genes in the genome of the yeast *S.cerevisiae*) and the same number of edges was reconstructed in approximately 15s on a desktop workstation (450 MHz Pentium II; RedHat Linux 6.2), using the perl implementation of the algorithm shown in the Appendix. Even for the much larger human genome ($n \approx 30\,000$), network reconstruction would thus be feasible on a desktop computer.

The algorithm stores two copies of the accessibility list, as well as a list of the nodes that have been visited. 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 $O(nk)$.

Cycles in genetic networks

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

There are two different kinds of cycles. First, an edge leaving a node might be directed onto the node itself. In graph theory such edges are called loops. In genetic networks they correspond to genes autoregulating their activity. Only certain perturbation techniques can detect such loops. For example, perturbing gene activity by gene deletion can not detect autoregulation, in contrast to overexpression of an extrachromosomal gene copy, while the activity of a chromosomal copy is measured. Autoregulation is immediately detected from a suitable perturbation experiment as an entry of the gene itself in its accessibility list. Because autoregulation does not pose any algorithmic problems, I will here discuss only loopfree graphs, corresponding to networks without autoregulated genes.

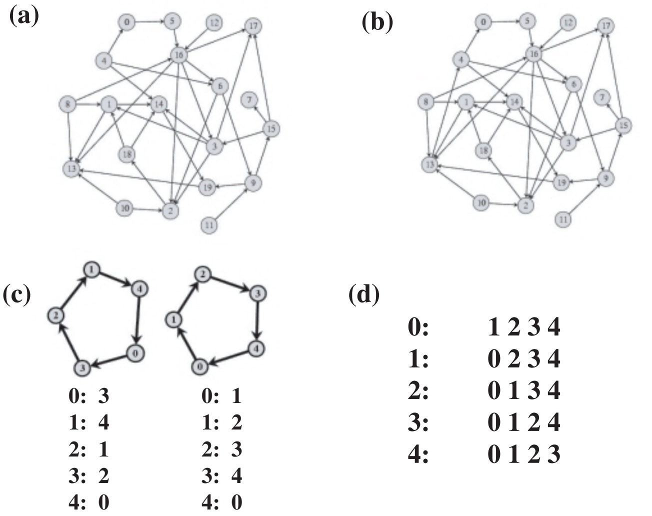


Fig. 6. Single gene perturbations can not resolve the order of genes in a cycle. (a) and (b) show a cyclic and an acyclic network, respectively, that differ by only one edge, the edge between nodes 13 and 4. (c) shows two cycles with their respective adjacency list. The order of genes in these cycles is different, yet they generate the same accessibility list, which is shown in (d).

The second type of cycles involves more than one gene. What is the problem with these cycles? Consider the two simple cyclic networks shown in Figure 6c. Notice that the order of direct regulatory interactions in these two networks is different, as reflected in the adjacency lists written underneath each network. However, both networks, when perturbed, would generate the accessibility list shown in Figure 6c. Characteristically, perturbation of any gene influences the activity of all other genes in the network. Thus, from single gene perturbations one can not uniquely reconstruct the structure of a cycle such as that in Figure 6c. In fact, all possible orders of the five genes in the network are consistent with the list of Figure 6d.

Notice that this is not an algorithmic but an experimental limitation. Elsewhere I will introduce an algorithm able to reconstruct the structure of *any* cyclic network with suitable experimental data. In this contribution, however, I will stay within the limits of single-gene perturbations. As is illustrated in Figure 6, the order of genes that are part of a cycle can not be resolved. They are thus collapsed into a single group of nodes with indistinguishable order of regulatory interactions. The general idea of what follows is to identify all cycles in a network and for each cycle, collapse all nodes that are part of it. The remaining network is acyclic and can be reconstructed with the algorithm for acyclic graphs.

I state some definitions and, without proof, some theorems from the theory of directed graphs. All of them can be found in Harary (1969). 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. That means, for any two nodes i and j, there is a path from i to j, and vice versa. This implies that there is a cycle through any two nodes of a strong component. Conversely, any two nodes through which there is a cycle are part of the same strong component. Strictly speaking, I will thus not only be concerned with all cycles but with all strong components of a digraph. A generalization of the principle above is that single gene perturbations can not resolve the adjacency list for any node in a strong component.

Each node of a directed graph lies in exactly one strong component. This holds also for acyclic graphs, if one defines that a graph (or the subgraph of G) with only one node is a strong component of G. 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. The relationship of a graph and its condensation is illustrated in Figure 7. Panel A shows a cyclic graph of 16 nodes. Upon close examination one finds one component with 5 genes (1, 3, 4, 5, 15; diamond-shaped nodes), another component with three genes (3, 6, 9; square nodes), and eight remaining single-gene components (round nodes). Panel B shows the condensation of the graph in A, where the two non-trivial components are now collapsed into single nodes. The condensation is an acyclic graph and can be reconstructed from the accessibility list.

To reconstruct a genetic network from single-gene perturbation experiments, one thus needs to identify all the strong components from experimental results, that is, from the accessibility list. The following theorem, due to Harary, is very useful for doing that.

THEOREM 3 (HARARY, 1969). *Let P be the accessibility matrix of a digraph G with n nodes, $x_1, \ldots, x_n$. The strong component containing x_i is determined by the unit entries of the ith row in the matrix $P \times P^{\mathrm{T}}$. (The superscripted 'T' denotes the matrix transpose of P, and the product '×' is the elementwise or Hadamard product of the two matrices.)*

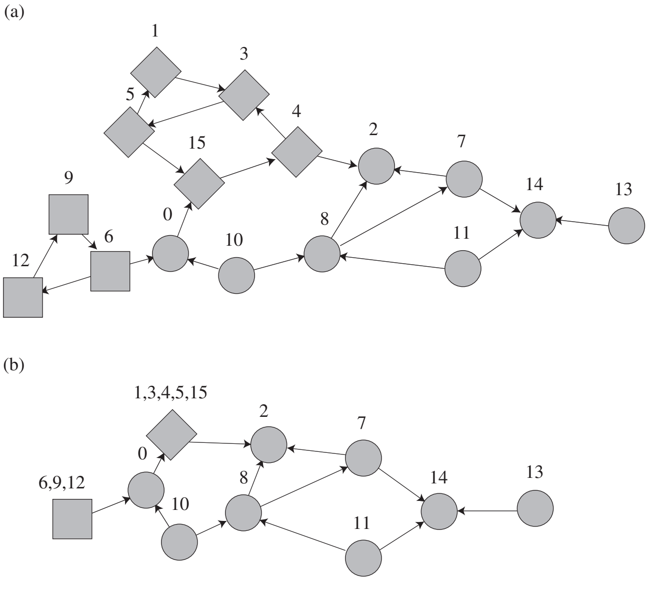


Fig. 7. Graphs and condensations. The graph shown in (a) contains two nontrivial strong components, comprising nodes 1, 3, 4, 5, 15 (diamonds), and comprising nodes 6, 9 and 12 (squares). In the condensation of this graph, shown in (b), the strong components are collapsed onto a single node.

Because I will be working with accessibility lists, not matrices, I will use the following corollary.

COROLLARY 3. *Let i and $j (i \neq j)$ be two nodes of a digraph G. i and j are in the same component iff $i \in \mathrm{Acc}(j)$ and $j \in \mathrm{Acc}(i)$.*

An algorithm applying this corollary to identify the strong components of a graph from the accessibility list is shown in Figure 8 as pseudocode. Not only does it identify the strong components, it also generates a new graph G^*, the condensation of G. To this end, it uses a data structure `component[`i`]` which is an array indexed by the nodes i of G and pointing to a node of G^* which corresponds to the component in which i resides. (In an actual implementation of the algorithm, a hashing array might be a convenient representation of such a data structure).

Before the algorithm starts, `component[`i`]` is undefined for all nodes i of G. The algorithm itself has two parts. In the first part (lines 1–9 in Figure 8), it cycles through all nodes of G. If a node i is found that has not been mapped onto a component (line 2), that is, the component which i belongs to has not yet been defined, then a new node of G^* is created (line 3), and i is mapped onto that node (line 4). Then, a loop (line 5) cycles over all nodes in $\mathrm{Acc}(i)$ and applies the above corollary to identify nodes in the same component as i. These nodes are then also mapped onto

```
1   for all nodes $i$ of $G$
2       if $component[i]$ has not been defined
3           create new node $x$ of $G^*$
4           $component[i]=x$
5           for all nodes $j \in Acc(i)$
6               if $i \in Acc(j)$
7                   $component[j]=x$
8               end if
9       end if

10  for all nodes $i$ of $G^*$
11      $Acc_{G^*}(i)=\varnothing$
12  for all nodes $i$ of $G$
13      for all nodes $j \in Acc(i)$
14          if $component[i] \neq component[j]$
15              if $component[j] \notin Acc_{G^*}(component[i])$
16                  add $component[j]$ to $Acc_{G^*}(component[i])$
17              end if
18          end if
```

Fig. 8. An algorithm to calculate the condensation of a cyclic network from perturbation data. The first part of the algorithm (lines 1–9) generates the nodes of the condensation, as well as a map from the nodes of the graph into the condensation. The second part (lines 10–18) generates edges between the nodes of the condensation.

`component`$[i]$ (lines 6–8). Notice that the conditional statement in line 2 saves potentially much execution time if the graph has few components. This is because it prevents scanning the accessibility list of i (lines 5–8) if `component`$[i]$ has been defined previously during the master loop (line 1). For instance, in the extreme case of a graph with only one component, the statements in the interior of the loop would only be executed for the first node i of the graph.

The second part of the algorithm then generates the accessibility list of G^* from that of G. It first initializes this list to the empty list for each node i of G^* (lines 10, 11). It then cycles through all nodes i of G (line 12), and through each node accessible from i, that is, through all $j \in \mathrm{Acc}(i)$. If i and j are in different components, that is, if they map to different nodes of G^*, the node in G^* represented by `component`$[j]$ must also be in the accessibility list of `component`$[i]$. If it has not been added to that list (line 15), it is added in line 16.

Because the graph G^* has at most the same number of nodes and accessibilities as G, and because the algorithm generates only one copy of G^* and its accessibility list, both storage and time complexity scale as $O(k)$ where k is the number of entries of the accessibility list ($k < n^2$).

Missing genes and messy data

The algorithm presented here can be used to reconstruct both large and small genetic circuits. It can be used to reconstruct a genetic network for an entire organism from perturbation data of all genes. At the time of this writing, the availability of such data is not utopian. For instance, more than 90% of all genes of the yeast *S.cerevisiae* have been perturbed by targeted gene deletion (Winzeler *et al.*, 1999). Similarly large-scale genetic perturbation projects are under way in the fruit fly *Drosophila melanogaster*, the nematode *Caenorhabditis elegans*, as well as in plants (Bouche and Bouchez, 2001; Fraser *et al.*, 2000; Gonczy *et al.*, 2000; Spradling *et al.*, 1999). In such experiments some genes are difficult to perturb, because they are essential to the organism. It is then also difficult to assess how their activity affects the activity of other genes. Sometimes a different kind of perturbation provides a solution to this problem, such as overexpression or conditional expression instead of gene deletion. Even so, it is likely that some genes remain impossible to perturb, or that one can not measure their perturbation effect. In the reconstruction of smaller genetic networks one encounters similar problems. For example, one might be interested in the regulatory interactions of all genes required for sporulation, or for chromosome segregation, or for the repair of radiation damage. Through earlier experiments, one might have an idea about what these genes are. For instance, one might have carried out a saturation mutagenesis experiment, or a large-scale gene expression study monitoring all genes whose expression changed during the process in question. However, some genes involved in the process of interest may not have been detected by this approach.

Thus, for one reason or another, when reconstructing a genetic network one is faced with the problem of missing information, genes for which no perturbation data is available. How does the algorithm perform in the face of such missing information? I will restrict myself here to the case of acyclic networks. The reason is that eliminating expression information from a cyclic network may change the number of cycles observed, and thus the number of nodes in the network's condensation. How condensations with different numbers of nodes are best compared to the original condensation is nontrivial and beyond the scope of this contribution.

To assess robustness of the algorithm from Figure 5 to missing genes, I first use methods described in Mehlhorn and Naher (1999) to generate a random graph of a pre-specified number of nodes and edges, which is then rendered acyclic by removal of suitably chosen edges. I use random networks purely for reasons of computational convenience. However, notice that recent analyses of the structure of large-scale metabolic and genetic networks suggest that they share important features with random networks (Jeong *et al.*, 2000; Wagner, 2000; Wagner and Fell, 2001). For a network thus generated, I then eliminate information on a pre-specified fraction of its nodes from its accessibility list in the following way. For each of the nodes X, I eliminate all entries of $\mathrm{Acc}(X)$ as well as all entries of X found in the accessibility lists of other

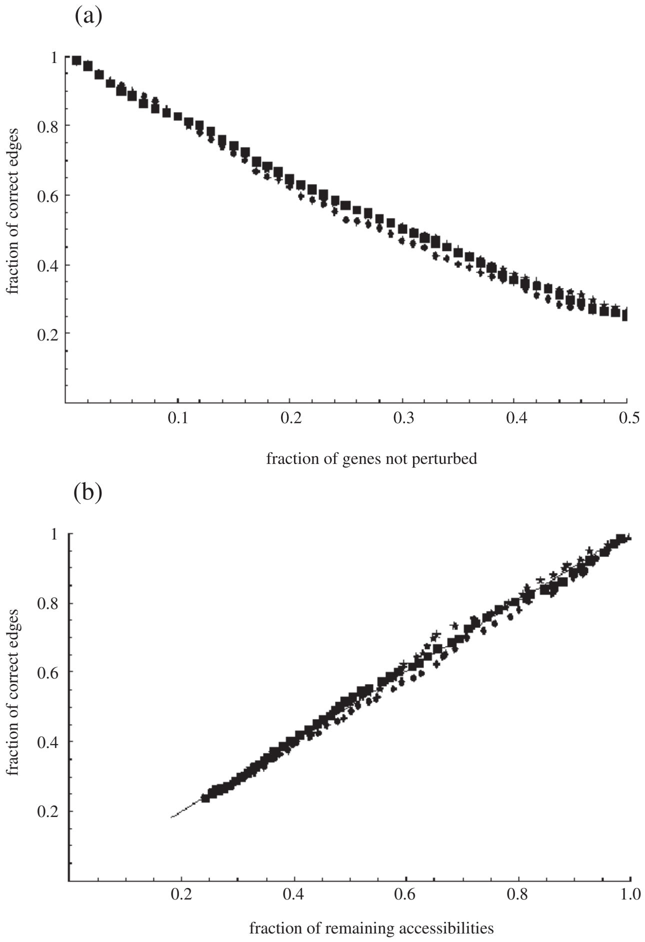


Fig. 9. Quality of network reconstruction with missing genes. Results are shown for three random graphs of 500 nodes and 250 edges (diamonds), 500 edges (stars), or 750 edges (squares), from which edges are removed until each network is rendered acyclic (Mehlhorn and Naher, 1999). After removal of these edges, the resulting three acyclic graphs have 250, 492, and 646 edges, respectively. The pruning algorithm from Figure 5 is then applied to the accessibility list of each of these networks, as well as to the same accessibility list after information on a pre-specified number of nodes is removed, as explained in the main text. This reduced accessibility list emulates a situation where a number of genes have not been perturbed. (a) Shows on the abscissa a measure of the number of these genes, that is, the fraction of genes on which information is missing. Plotted against it is the fraction of correctly reconstructed edges. More precisely, it is the fraction of edges that the network reconstructed with missing perturbation information has in common with the network with complete information. (b) Shows the same measure of reconstruction quality. The only difference to (a) is that the abscissa shows the fraction of remaining entries of the accessibility list, and not the fraction of missing genes. The value of one on the abscissa refers to the number of entries of the accessibility list for a network where all genes were perturbed. As the fraction of missing genes increases from 0 to 0.5 (as shown in (a)), the fraction of remaining entries of the accessibility list decreases from one to approximately 0.25 for all three networks shown. Both panels show that the quality of network reconstruction is not very sensitive to the number of edges in the network. The most direct predictor of this quality is the number of remaining entries of the accessibility list. This is indicated by the slope of the regression line shown in (b) through the data points pooled for all three networks. It is nearly identical to one ($y = 1.01x - 0.002$; Pearson $r^2 = 0.994$).

genes. I then reconstruct a network from this modified accessibility list using the algorithm of Figure 5, and determine what fraction of edges between the remaining nodes the algorithm has identified correctly. The results are shown in Figure 9 for three random networks of 500 nodes and 250, 500 and 750 edges.

Quality of network reconstruction is not sensitive to the number of edges, but decays linearly with the number of genes on which information is missing (Figure 9a). The best predictor of network reconstruction quality is the fraction of entries of the accessibility list remaining after removal of a certain fraction of nodes. Its relation to the fraction of correctly reconstructed direct interactions is practically one to one, as indicated by the slope of the regression line in Figure 9b. This is not surprising, as one can think of each accessibility as a bit of information used in reconstructing the network. It would in fact be very surprising if the reconstruction algorithm could do any better than shown in Figure 9b, that is, if the slope of the regression line could be much less than one. This would mean that for any entry removed from an accessibility list, one would lose less than one accurately reconstructed direct interaction. Conversely, a slope much greater than one would indicate poor performance, in the sense that the algorithm does not use all information contained in an accessibility list.

Another problem is flawed data. By flawed data I mean spurious or missing entries of an accessibility list. Such data is the result of errors in measuring gene activities. The reason why flawed data is a problem is that not just any list of the form shown in Figure 2c is the accessibility list of a graph. Take the accessibility of the simple network 1 → 2 → 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 may pose challenges for any reconstruction algorithm.

There are two ways to address this problem. One way is to use only the most reliable data. For example, in a microarray 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 way 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, the algorithm (Figure 8) generating the condensation is applied repeatedly, until the pruning algorithm (Figure 5) 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 algorithmic modification would yield little or no useful information on the network. 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 at least a crude assessment of algorithmic robustness to defects in the accessibility list, I will focus on one aspect of robustness, robustness to missing entries of the accessibility list. Current techniques to measure the effects of gene perturbations on gene activity, such as transcriptional activity measurements provided by microarrays, are very noisy. It has thus become common practice to call only those genes affected by a perturbation whose expression level changes by more than some pre-specified factor. This factor is chosen in a statistically conservative way in order to avoid false positive results. Statistical conservatism leads to the usual problem that some genes actually affected by the perturbation are not identified as such. With this in mind I will address the question of how the network reconstruction algorithm behaves when a fraction of perturbation effects (entries of the accessibility list) are not identified in an experiment.

I restrict myself to acyclic networks, for reasons discussed above. I generate a random network of a pre-specified number of nodes and edges along with its accessibility list, and eliminate a fraction of the entries of its accessibility list at random. I then apply the algorithm from Figure 5 to the list thus generated, and assess the fraction of edges that the algorithm identifies correctly, that is, the fraction of edges that are in both the actual network and the network reconstructed from the changed accessibility list. Figure 10 shows the results of this analysis. Very similar to what has been shown in Figure 9, and for the same reasons, the quality of network reconstruction shows a statistical one-to-one relationship with the number of remaining entries of the accessibility list.

DISCUSSION

The algorithm presented here proceeds in two steps. First, it renders a genetic network acyclic by collapsing all cycles onto single nodes, and it then reconstructs the regulatory interactions in the remaining acyclic network.

Limitations

The inability to resolve cycles may seem like a limitation of the algorithm but it really is a limitation of the data. No single gene perturbation experiment can resolve cycles, as illustrated in Figure 6. The question then arises how

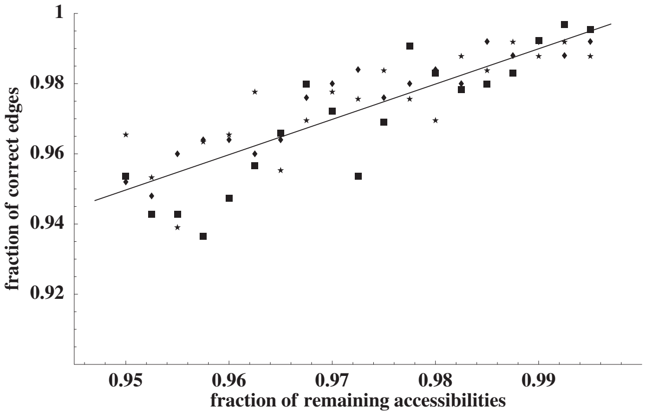


Fig. 10. Quality of network reconstruction with unidentified perturbation effects. Results are shown for three random graphs of 500 nodes and 250 edges (diamonds), 500 edges (stars), or 750 edges (squares), from which edges are removed until each network is rendered acyclic (Mehlhorn and Naher, 1999). After removal of these edges, the resulting three acyclic graphs have 250, 492, and 646 edges left, respectively. For each of these networks, a pre-specified fraction of entries is then eliminated at random from the accessibility list. The fraction of remaining entries is shown on the abscissa. The pruning algorithm from Figure 5 is applied to the changed accessibility list, and the network it reconstructs is then compared to the actual graph. More precisely, the fraction of correctly identified edges in the reconstructed network with missing accessibilities is determined. This fraction is shown on the ordinate axis. There is a statistical one-to-one relation between the number of remaining entries and the fraction of correctly reconstructed interactions ($y = 1.006x - 0.006$; Pearson $r^2 = 0.75$).

important this limitation is in network reconstruction. While it is safe to assume that any genetic circuit contains feedback controls, and thus cycles, it is much less clear how frequent cyclic interactions are when measuring only one aspect of gene activity. In other words, how many transcription factor genes indirectly influence their own transcriptional state, and how many protein kinases indirectly influence their own phosphorylation state? There are examples of such control loops, but they may be less frequent than the more general feedback controls involving two or more different kinds of gene activity. (I am emphasizing indirectness in feedback, because direct self-regulation by a gene product can be immediately inferred from a suitably designed experiment without requiring any sophisticated algorithm.)

A second limitation is that the notion of gene activity limits the information one can obtain on network structure (Figure 1). 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 at the scale required here. Currently, only mRNA expression can be measured at that scale. However, because most eukaryotic genes are regulated transcriptionally, the reconstruction of transcriptional regulation networks will provide a backbone into which other measurements of gene activity can be easily integrated, once they are available.

A third shortcoming is that the algorithm requires more data than conventional methods using gene expression correlations. This is the price to pay for resolving causal interactions. It illustrates an informational trade-off involved in reconstructing genetic networks. Fortunately, genome-scale experiments are in the process of providing the required genetic perturbations for several model organisms (Bouche and Bouchez, 2001; Fraser *et al.*, 2000; Gonczy *et al.*, 2000; Hughes *et al.*, 2000; Spradling *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 then 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. The algorithm only reconstructs one of them, 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. In a related vein, gene perturbation data may resolve the influence of redundant genes on other genes to a limited extent. Gene redundancy has often been postulated when knockout mutations of a gene show weak or no detectable phenotypic effect. However, on a genome-wide scale such redundancy may be less abundant than commonly assumed (Wagner, 2000). For example, a recent large scale analysis of the effect of several hundred knockout mutations on growth phenotypes and mRNA expression patterns in the yeast *S.cerevisiae* reported that the vast majority of mutations with weak phenotypic effects showed detectable alterations in gene expression patterns (Hughes *et al.*, 2000).

The next steps

Computational complexity of the algorithm is low, and it is sufficiently fast that a genetic network with as many genes as the human genome could be reconstructed on a desktop workstation. While efforts to improve algorithmic efficiency may thus lead to marginal returns, significant improvement in other areas is possible.

Because the data currently available to apply the algorithm is notoriously noisy (DeRisi *et al.*, 1997), I have restricted myself to assessing regulatory interactions qualitatively. That is, I do not ask to what extent a gene infuences another gene's activity, or whether this influence is activating or repressing. However, once the network is reconstructed, this information is easily read from the experimental data and superimposed onto the network's edges. Doing that poses no algorithmic problem. Second, although I crudely assessed robustness of reconstruction quality to missing genes and flawed data, a more rigorous evaluation is clearly possible. It is, however, best postponed until we know more about the statistical structure of errors in large-scale gene activity measurements. Third, integrating different kinds of genomic data may provide additional useful information. For instance, superimposing functional annotation for network genes onto the structure of a reconstructed network may help distinguish between direct and indirect interactions beyond the resolution of the perturbation experiment itself.

Conclusions

Genetics is concerned with identifying gene interactions and their biological significance. Functional genomics takes this concern to the next level, that of identifying gene interactions among thousands of genes in a genome. Thus, a tool to identify such interactions, and to distinguish direct from indirect interactions, applies to virtually any area in these two fields.

The algorithm may help answer a multitude of questions about the genetic architecture of organisms. What is the structure of genetic networks? How do patterns of gene interactions change in different developmental stages, in different physiological states, in different environmental conditions, or in different cell types? Are there few or many genes that do not affect the activity of other genes. What about so-called master regulators, genes that drive large parts of a physiological or developmental program? Do they have a characteristic profile of regulatory interactions? These are all coarse-scale questions about genetic networks. In addition, by distinguishing a gene's direct and indirect regulation targets, the algorithm can help sift through a large amount of genomic information to identify candidate genes for targeted biochemical investigation.

Insofar as our understanding of intact organisms helps us understand the nature of disease, a tool to identify direct gene interactions has broad applications in basic and applied biomedical research. To give but two examples, it may be useful to identify targets for conventional therapeutic agents or for gene therapy. Second, there may be variation in genetic network structure within human populations. If so, the tool can be used to identify the nature of this variation, and thus provide information useful to pharmacogeneticists. There are also countless applications to organisms other than humans. One example is agricultural biotechnology, where the design of effective pesticides may depend on our understanding of gene interactions involved in host defense, pest survival, reproduction, or virulence. In sum, a tool to reconstruct genetic network structure from gene perturbation data is useful wherever regulatory gene interactions are important for our understanding of how organisms—be they humans, animals or plants—function, or how disease comes about.

ACKNOWLEDGEMENTS

I would like to thank the Santa Fe Institute for its continued support of my research program, and two reviewers for their constructive comments.

APPENDIX

Below is a perl implementation of the algorithm to reconstruct acyclic genetic networks by pruning accessibility lists. Its structure follows exactly that of the pseudocode shown in Figure 5 and explained in the main text, with one difference. Only one data structure is used to represent both accessibility list and adjacency list. 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 `$acc{$i}{$j}` = 1 if gene `$j` is accessible from gene `$i`. This implies that `$acc{$i}{$j}` is also *defined*. If gene `$j` is not accessible from gene `$i`, then `$acc{$i}{$j}` must be *undefined*. The algorithm tests whether `$j` is accessible from `$i` by testing whether the corresponding entry of `acc` is defined, but it prunes `acc` by setting an entry to zero. After execution, all entries of `acc` that are still equal to one are entries of `adj` and can be read out that way. Visited nodes are kept track of 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.

```
# master loop
foreach $i(sort keys %acc)   {
      if($visited{$i}!=1)       {
          PRUNE_ACC($i);
      }
}

sub PRUNE_ACC    {
      # declare calling variable as local
      my $i=@_[0];
      # loop one of PRUNE_ACC
      foreach $j  (keys %{$acc{$i}})   {
            if($visited{$j}!=1)           {
                 if (scalar(keys %{$acc{$j}})==0) {
                       $visited{$j}=1;
                 }
                 else  {
                       PRUNE_ACC($j);
                 }
            }
      }

      #loop two of PRUNE_ACC
      foreach $j (keys %{$acc{$i}} )    {
            foreach $k (keys %{$acc{$j}})     {
                 if($acc{$j} {$k}==1)     {
                       if ($acc{$i} {$k}==1)     {
                             $acc{$i} {$k}=0;
                       }
                 }
            }
      }
$visited{$i}=1;
}
```

REFERENCES

Bouche,N. and Bouchez,D. (2001) *Arabidopsis* gene knockout: phenotypes wanted. *Curr. Opin. Plant Biol.*, **4**, 111–117.

DeRisi,J.L., Iyer,V.R. and Brown,P.O. (1997) Exploring the metabolic and genetic control of gene expression on a genomic scale. *Science*, **278**, 680–686.

Eisen,M.B., Spellman,P.T., Brown,P.O. and Botstein,D. (1998) Cluster analysis and display of genome-wide expression patterns. *Proc. Natl Acad. Sci. USA*, **95**, 14 863–14 868.

Fell,D. and Wagner,A. (2000) The small world of metabolism. *Nature Biotechnol.*, **18**, 1121–1122.

Fraser,A.G., Kamath,R.S., Zipperlen,P., MartinezCampos,M., Sohrmann,M. *et al.* (2000) Functional genomic analysis of *C.elegans* chromosome I by systematic RNA interference. *Nature*, **408**, 325–330.

Gonczy,P., Echeverri,C., Oegema,K., Coulson,A., Jones,S.J.M. *et al.* (2000) Functional genomic analysis of cell division in *C.elegans* using RNAi of genes on chromosome III. *Nature*, **408**, 331–336.

Harary,F. (1969) *Graph Theory*. Addison-Wesley, Reading, MA.

Hughes,T.R., Marton,M.J., Jones,A.R., Roberts,C.J., Stoughton,R. *et al.* (2000) Functional discovery via a compendium of expression profiles. *Cell*, **102**, 109–126.

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

Mehlhorn,K. and Naher,S. (1999) *LEDA: a Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge.

Spradling,A.C., Stern,D., Beaton,A., Rhem,E.J., Laverty,T. *et al.* (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,J.D., Campbell,M.J., Cho,R.J. and Church,G.M. (1999) Systematic determination of genetic network architecture. *Nature Genet.*, **22**, 281–285.

Wagner,A. (2000) Mutational robustness in genetic networks of yeast. *Nature Genet.*, **24**, 355–361.

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

Wagner,A. (2001b) The yeast protein interaction network evolves rapidly and contains few redundant duplicate genes. *Mol. Bio. Evol.*, **18**, 1283–1292.

Wagner,A. and Fell,D. (2001) The small world inside large metabolic networks. *Proceedings of the Royal Society of London Series B*, **268**, 1803–1810.

Watts,D.J. (1997) *The Structure and Dynamics of Small World Networks*, PhD Dissertation, Cornell University.

Winzeler,E.A., Shoemaker,D.D., Astromoff,A., Liang,H., Anderson,K. *et al.* (1999) Functional characterization of the *S.cerevisiae* genome by gene deletion and parallel analysis. *Science*, **285**, 901–906.