Networks of Relations

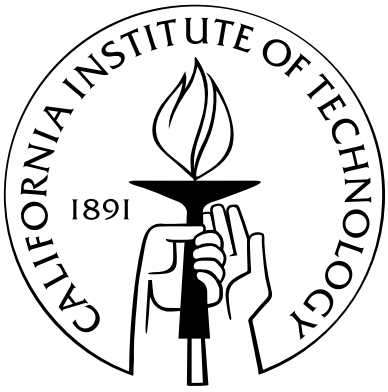
Thesis by

Matthew Cook

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy



California Institute of Technology

Pasadena, California

2005

(Defended May 27, 2005)

Acknowledgements

I would like to thank the ARCS foundation, the National Institute of Health, the Alpha Project, and my advisor Shuki Bruck for supporting me during my studies. I would also like to thank Shuki for being a good advisor and collaborator. I am grateful not only to Shuki but to all the people I have worked with, including Erik Winfree and David Soloveichik, in collaboration with whom the material in section 3.4.2 was produced. My family has supported my adventure of being a student, especially my wife Éva, my children András, Adam, and Emma, my mother Sarah, and my grandfather Howard, and to them I am very grateful.

Abstract

Relations are everywhere. In particular, we think and reason in terms of mathematical and English sentences that state relations. However, we teach our students much more about how to manipulate functions than about how to manipulate relations. Consider functions. We know how to combine functions to make new functions, how to evaluate functions efficiently, and how to think about compositions of functions. Especially in the area of boolean functions, we have become experts in the theory and art of designing combinations of functions to yield what we want, and this expertise has led to techniques that enable us to implement mind-bogglingly large yet efficient networks of such functions in hardware to help us with calculations. If we are to make progress in getting machines to be able to reason as well as they can calculate, we need to similarly develop our understanding of relations, especially their composition, so we can develop techniques to help us bridge between the large and small scales. There has been some important work in this area, ranging from practical applications such as relational databases to extremely theoretical work in universal algebra, and sometimes theory and practice manage to meet, such as in the programming language Prolog, or in the probabilistic reasoning methods of artificial intelligence. However, the real adventure is yet to come, as we learn to develop a better understanding of how relations can efficiently and reliably be composed to get from a low level representation to a high level representation, as this understanding will then allow the development of automated techniques to do this on a grand scale, finally enabling us to build machines that can reason as amazingly as our contemporary machines can calculate.

This thesis explores new ground regarding the composition of relations into larger relational structures. First of all a foundation is laid by examining how networks of relations might be used for automated reasoning. We define *exclusion networks*, which have close connections with the areas of constraint satisfaction problems, belief propagation, and even boolean circuits. The foundation is laid somewhat deeper than usual, taking us inside the

relations and inside the variables to see what is the simplest underlying structure that can satisfactorily represent the relationships contained in a relational network. This leads us to define *zipper networks*, an extremely low-level view in which the names of variables or even their values are no longer necessary, and relations and variables share a common substrate that does not distinguish between the two. A set of simple equivalence operations is found that allows one to transform a zipper network while retaining its solution structure, enabling a relation-variable duality as well as a canonical form on linear segments. Similarly simple operations allow automated deduction to take place, and these operations are simple and uniform enough that they are easy to imagine being implemented by biological neural structures.

The canonical form for linear segments can be represented as a matrix, leading us to *matrix networks*. We study the question of how we can perform a change of basis in matrix networks, which brings us to a new understanding of Valiant's recent *holographic algorithms*, a new source of polynomial time algorithms for counting problems on graphs that would otherwise appear to take exponential time. We show how the holographic transformation can be understood as a collection of changes of basis on individual edges of the graph, thus providing a new level of freedom to the method, as each edge may now independently choose a basis so as to transform the matrices into the required form.

Consideration of zipper networks makes it clear that "fan-out," i.e., the ability to duplicate information (for example allowing a variable to be used in many places), is most naturally itself represented as a relation along with everything else. This is a notable departure from the traditional lack of representation for this ability. This deconstruction of fan-out provides a more general model for combining relations than was provided by previous models, since we can examine both the traditional case where fan-out (the equality relation on three variables) is available and the more interesting case where its availability is subject to the same limitations as the availability of other relations. As we investigate the composition of relations in this model where fan-out is explicit, what we find is very different from what has been found in the past.

First of all we examine the relative expressive power of small relations: For each relation on three boolean variables, we examine which others can be implemented by networks built solely from that relation. (We also find, in each of these cases, the complexity of deciding whether such a network has a solution. We find that solutions can be found in polynomial

time for all but one case, which is NP-complete.) For the question of which relations are able to implement which others, we provide an extensive and complete answer in the form of a hierarchy of relative expressive power for these relations. The hierarchy for relations is more complex than Post's well-known comparable hierarchy for functions, and parts of it are particularly difficult to prove. We find an explanation for this phenomenon by showing that in fact, the question of whether one relation can implement another (and thus should be located above it in the hierarchy) is undecidable. We show this by means of a complicated reduction from the halting problem for register machines. The hierarchy itself has a lot of structure, as it is rarely the case that two ternary boolean relations are equivalent. Often they are comparable, and often they are incomparable—the hierarchy has quite a bit of width as well as depth. Notably, the fan-out relation is particularly difficult to implement; only a very few relations are capable of implementing it. This provides an additional *ex post facto* justification for considering the case where fan-out is absent: If you are not explicitly provided with fan-out, you are unlikely to be able to implement it.

The undecidability of the hierarchy contrasts strongly with the traditional case, where the ubiquitous availability of fan-out causes all implementability questions to collapse into a finite decidable form. Thus we see that for implementability among relations, fan-out leads to undecidability. We then go on to examine whether this result might be taken back to the world of functions to find a similar difference there. As we study the implementability question among functions without fan-out, we are led directly to questions that are independently compelling, as our functional implementability question turns out to be equivalent to asking what can be computed by sets of chemical reactions acting on a finite number of species. In addition to these chemical reaction networks, several other nondeterministic systems are also found to be equivalent in this way to the implementability question, namely, Petri nets, unordered Fractran, vector addition systems, and "broken" register machines (whose decrement instruction may fail even on positive registers). We prove equivalences between these systems.

We find several interesting results in particular for chemical reaction networks, where the standard model has reaction rates that depend on concentration. In this setting, we analyze questions of possibility as well as questions of probability. The question of the possibility of reaching a target state turns out to be equivalent to the reachability question for Petri nets and vector addition systems, which has been well studied. We provide a

new proof that a form of this reachability question can be decided by primitive recursive functions. Ours is the first direct proof of this relationship, avoiding the traditional excursion to Diophantine equations, and thus providing a crisper picture of the relationship between Karp's coverability tree and primitive recursive functions.

In contrast, the question of finding the probability (according to standard chemical kinetics) of reaching a given target state turns out to be undecidable. Another way of saying this is that if we wish to distinguish states with zero probability of occurring from states with positive probability of occurring, we can do so, but if we wish to distinguish low probability states from high probability states, there is no general way to do so. Thus, if we wish to use a chemical reaction network to perform a computation, then if we insist that the network must always get the right answer, we will only be able to use networks with limited computational power, but if we allow just the slightest probability of error, then we can use networks with Turing-universal computational ability. This power of probability is quite surprising, especially when contrasted with the conventional computational complexity belief that $BPP = P$.

Exploring the source of this probabilistic power, we find that the probabilities guiding the network *need* to depend on the concentrations (or perhaps on time)—fixed probabilities aren't enough on their own to achieve this power. In the language of Petri nets, if one first picks a transition at random, and then fires it if it is enabled, then the probability of reaching a particular target state can be calculated to arbitrary precision, but if one first picks a token at random, and then fires an enabled transition that will absorb that token, then the probability of reaching a particular target state cannot in general be calculated to any precision whatsoever.

In short, what started as a simple thorough exploration of the power of composition of relations has led to many decidability and complexity questions that at first appear completely unrelated, but turn out to combine to paint a coherent picture of the relationship between relations and functions, implementability and reachability, possibility and probability, and decidability and undecidability.

Contents

Chapter 1

Exclusion Networks: Points of View

This thesis presents a model of networks of relations. This model represents knowledge as the conjunction of many small relations.

These networks of relations are similar to other models such as constraint satisfaction problems (CSPs) and Bayesian networks (also known as belief propagation networks), and indeed, in the simplest boolean cases, we will see that all three models are equivalent.

This first chapter introduces networks of relations, and exclusion networks in particular, and presents several different ways of using them and thinking about them. It concludes with comparisons of exclusion networks with some well known similar models.

Chapter 2 will look at networks of relations from a logical point of view, presenting many results about networks whose elements are restricted to be of a certain form.

Chapter 3 will examine questions of decidability relating to and inspired by networks of relations.

1.1 Discrete Relations

A relation on n variables is simply a set of n-tuples. As a familiar example, the relation $\geq$ can be thought of as the set of all pairs $\langle a, b\rangle$ such that a is no less than b. In most of this thesis, we will only be considering discrete systems, often just the boolean alphabet $\{0, 1\}$. For variables whose values must be either 0 or 1, the relation $\geq$ is simply the set $\{\langle 0, 0\rangle, \langle 1, 0\rangle, \langle 1, 1\rangle\}$.

Of course, relations can also exist on other numbers of variables. A familiar relation on

3 variables is the notion that $y \in [x, z]$, which is equivalent to $x \leq y \leq z$. On the boolean alphabet, this relation would be the set $\{\langle 0,0,0\rangle, \langle 0,0,1\rangle, \langle 0,1,1\rangle, \langle 1,1,1\rangle\}$. Some familiar relations on 1 variable are notions such as "x is prime" or "x is even." The latter makes sense on the boolean alphabet as well, where it is simply the set $\{\langle 0\rangle\}$.

A relation is essentially the same thing as a predicate in logic. Our networks of relations will use the relations to represent relationships (possibly learned) between different variables. The term "predicate" has more the flavor of something that can be true or false given values of the variables, but here we will typically start out with just the relations and slowly try to figure out what these known relationships can tell us about the values the variables might have. Thus we do not think of our relations as being true or false, but simply as relationships that must be satisfied. When analyzing a situation, we may speak of a relation being satisfied or not, but our networks will never actually contain an unsatisfied relation—the presence of a relation in the network implies that the relation holds.

As a simple exercise, let us find all possible relations on two boolean variables, to see whether they all look familiar or not. There are four possible pairs of values for the two variables, namely $\langle 0,0\rangle$, $\langle 0,1\rangle$, $\langle 1,0\rangle$, and $\langle 1,1\rangle$. Thus there are 2^4 possible relations, as shown in table 1.1. Each of these relations can be expressed by a simple traditional equation or inequality. Note that some of the relations, such as the fourth one ($x = 0$), are only sensitive to the value of one of the two variables. We say that this relation *ignores*, or *doesn't care* about, the value of y. In particular, note that the *unhappy* ☹ relation on two variables, which is never satisfied, and the *happy* ☺ relation on two variables, which is always satisfied, are the two relations which ignore both of their variables.

Once we are comfortable with the relations on two boolean variables, we can take a look at the four relations on one boolean variable, and the two relations on zero variables. The four relations on one boolean variable are $\{\}$, $\{\langle 0\rangle\}$, $\{\langle 1\rangle\}$, and $\{\langle 0\rangle, \langle 1\rangle\}$. We see that these are the unhappy relation, the "$= 0$" relation, the "$= 1$" relation, and the happy relation. We will also sometimes call the happy relation on one variable the *don't care* relation.

For zero variables, we might ask, how can there be a relation on zero variables? Well, there is exactly one tuple of length zero, namely $\langle\rangle$. So there are two possible relations on zero variables, the unhappy relation $\{\}$ and the happy relation $\{\langle\rangle\}$. After all, these are the only two relations that could possibly decide whether they are satisfied or not without consulting any variables at all. Although they seem a bit odd at first, it is sometimes useful

$\langle 1,1\rangle$	$\langle 1,0\rangle$	$\langle 0,1\rangle$	$\langle 0,0\rangle$	*relation*	*equation*
				$\{\}$	$0 = 1$
			$\surd$	$\{\langle 0,0\rangle\}$	$x + y = 0$
		$\surd$		$\{\langle 0,1\rangle\}$	$x < y$
		$\surd$	$\surd$	$\{\langle 0,0\rangle, \langle 0,1\rangle\}$	$x = 0$
	$\surd$			$\{\langle 1,0\rangle\}$	$x > y$
	$\surd$		$\surd$	$\{\langle 0,0\rangle, \langle 1,0\rangle\}$	$y = 0$
	$\surd$	$\surd$		$\{\langle 0,1\rangle, \langle 1,0\rangle\}$	$x \neq y$
	$\surd$	$\surd$	$\surd$	$\{\langle 0,0\rangle, \langle 0,1\rangle, \langle 1,0\rangle\}$	$x \cdot y = 0$
$\surd$				$\{\langle 1,1\rangle\}$	$x \cdot y = 1$
$\surd$			$\surd$	$\{\langle 0,0\rangle, \langle 1,1\rangle\}$	$x = y$
$\surd$		$\surd$		$\{\langle 0,1\rangle, \langle 1,1\rangle\}$	$y = 1$
$\surd$		$\surd$	$\surd$	$\{\langle 0,0\rangle, \langle 0,1\rangle, \langle 1,1\rangle\}$	$x \leq y$
$\surd$	$\surd$			$\{\langle 1,0\rangle, \langle 1,1\rangle\}$	$x = 1$
$\surd$	$\surd$		$\surd$	$\{\langle 0,0\rangle, \langle 1,0\rangle, \langle 1,1\rangle\}$	$x \geq y$
$\surd$	$\surd$	$\surd$		$\{\langle 0,1\rangle, \langle 1,0\rangle, \langle 1,1\rangle\}$	$x + y > 0$
$\surd$	$\surd$	$\surd$	$\surd$	$\{\langle 0,0\rangle, \langle 0,1\rangle, \langle 1,0\rangle, \langle 1,1\rangle\}$	$1 = 1$

Table 1.1: All of the possible relations on two boolean variables. For each relation, an equation (or inequality) is given which holds exactly when the relation, on the variables $\langle x, y\rangle$, is satisfied. Often other equations or inequalities could equally well have been given. Note that the first relation is never satisfied, and the last relation is always satisfied, regardless of the values of x and y. We call the first relation the *unhappy* (☹) relation on two variables, and we call the last one the *happy* (☺) relation on two variables.

to understand things in terms of them, and soon they will seem completely natural.

Of course, we can keep investigating specific relations as long as we like, looking at relations on three boolean variables (which much of chapter 2 is about) or even more variables, or we could expand the variables from being two-valued (boolean) variables to being three-valued variables, or integer-valued variables, or even good old familiar real-valued variables (which section 1.13 will discuss). But let us instead decide that we are comfortable enough with relations now to proceed to the next step: looking at how we build *networks* of relations.

1.2 Networks of Relations

A network of relations is a graph where the edges are variables and the vertices are relations. For example, figure 1.1(a) shows a network of relations. The vertices are drawn as circles with a symbol inside representing the relation. The symbol "=" indicates the relation that all three variables should have the same value. The symbol "$\neq$" indicates that the variables should not all three be the same (since the variables here are boolean, this means two must be the same while one differs). The symbol "$\oplus_e$" indicates that the parity of the three variables must be even (i.e., their sum must be even). The "☺" indicates that any values are acceptable to that relation. The "$\boxed{012}$" symbol indicates that the sum of the three incoming wires should equal 0, 1, or 2. Similarly, the "$\boxed{3}$" symbol indicates that the sum should equal 3, that is, all three variables must equal 1.

There are many ways that values can be assigned to the variables (the edges) so that all the relations are satisfied. One such solution is shown in figure 1.1(b). This network is small enough that one can simply list by hand all the solutions. (If you do, you should find that there are eleven.)

But in larger networks, it may not even be clear whether or not any solution exists at all. Using just "=," "$\neq$," and "$\boxed{123}$" relations, it is straightforward to construct a network that corresponds directly to a given 3-SAT problem, so it is NP-complete to determine whether any solutions exist for a given arbitrary network of relations. However, as we will see in chapter 2, there are many cases where polynomial time is sufficient for finding a solution. Similarly, counting the number of solutions is #P-complete in the general case, but in many cases the number of solutions can be found in polynomial time.

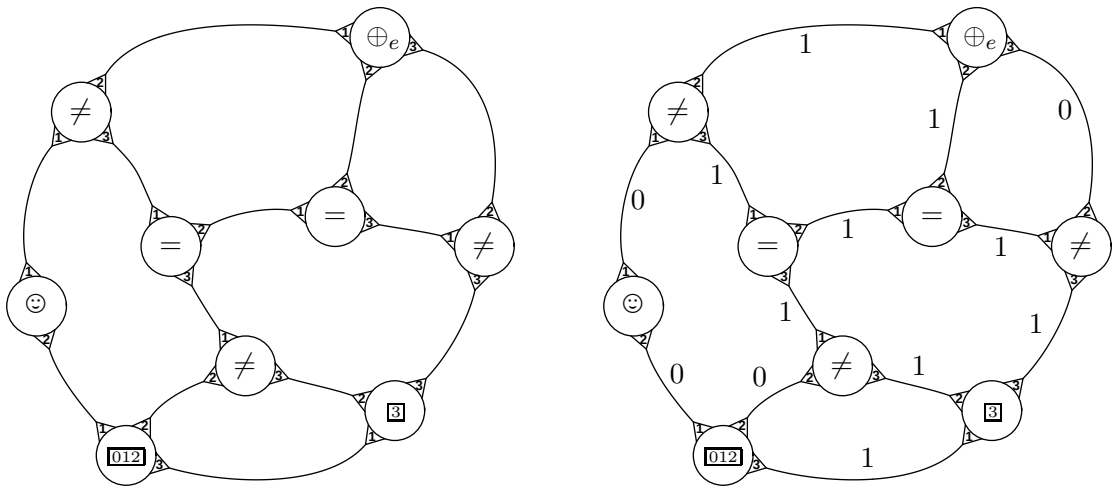


Figure 1.1: An example of (a) a network of relations, and (b) one possible solution for the network. The edges in this network represent boolean variables (variables which must be either 0 or 1), and the vertices represent relations on the incident edges. The little number in the cone where an edge touches a vertex represents how the relation is able to keep track of which wire is which: The k^{th} position in each of the relation's tuples contains values for the wire connected at the cone with label k.

Sometimes we will be interested in the question of whether there is a solution, and sometimes we will be interested in the question of how many solutions there are. We will also often be interested in considering just a portion of a network, since a portion of a network effectively implements one big relation. For example, in figure 1.2, we see that the upper portion has a solution as long as not all of the connecting edges have value 1. So for solvability, the upper portion is equivalent, from the outside, to a single "$\boxed{012}$" relation.

Indeed, the notion of a portion of a network is important enough that we will broaden our definition of a network of relations to allow such "portions." We will allow a network to have *dangling edges* (such as the connecting edges hanging down in figure 1.2), and we will say that such a network *implements* a relation on the dangling edges. These definitions will be given more formally in section 1.6. In chapter 2, we will consider questions of implementability such as "Does there exist a network of "$\neq$" relations that implements the "$=$" relation?"

Another question we will be interested in is, given values for variables in part of a network, what can we deduce about the variables in the rest of the network? This is what the *exclusion process*, discussed in the next section, is designed for.

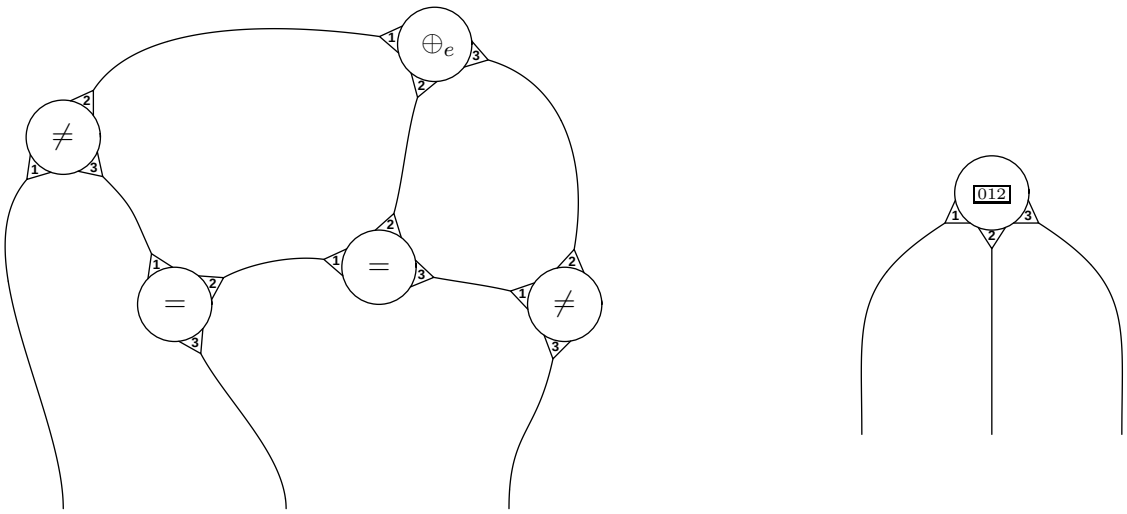


Figure 1.2: On the left is the upper portion of the network of figure 1.1. This network has three dangling edges, and is equivalent (regarding satisfiability) to the single relation on the right. For example, the combination of values $\langle 1, 0, 1\rangle$, for the three dangling wires, is acceptable to the network on the left because all the internal wires can be set so that all the relations in the network are satisfied. However, for the combination of values $\langle 1, 1, 1\rangle$, there is no way to set all the internal wires so that all the relations are satisfied. The relation on the right can be thought of as a simplification of the relational network on the left. We say that the network on the left *implements* the relation on the right.

1.3 Exclusion Networks

In exclusion networks, the word "exclusion" refers to an exclusion process which propagates among the relations in the network. We will now describe this exclusion process.

In an exclusion network, an edge between two relations (which we will also call a *wire*) represents a variable. A wire connecting two relations effectively says that the associated variable must have the same value for both relations. So what is the information that gets transmitted along the wire?

A wire transmits the set of values which are plausible for the given variable.

The plausibility referred to is local plausibility: The wire contains those values which currently seem plausible, given the states of the relations the wire is connected to. If there is some subtle reason, involving remotely located variables, why a wire cannot have a value, the wire might never become aware of this.

Thus, a wire for a boolean variable will always be in one of four states: *(1)* both values are currently considered possible for the variable, *(2)* the variable must be 0, *(3)* the variable

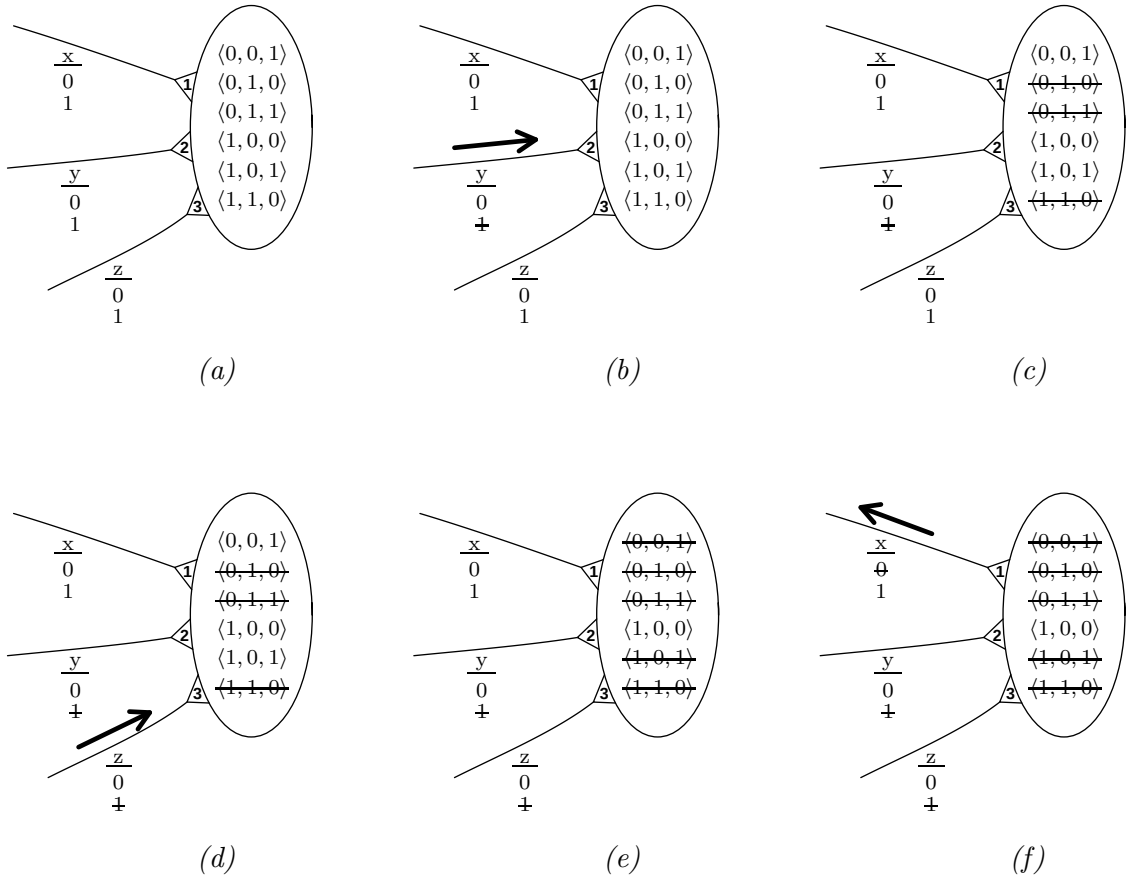


Figure 1.3: The exclusion process. A $\neq$ relation on the variables x, y, and z starts out as shown in *(a)*. The wire for y excludes the value 1 in *(b)*, causing the relation to exclude such tuples (where y is 1) in *(c)*. Then in *(d)* the wire for z excludes the value 1, causing even more tuples to be excluded as shown in *(e)*. This leads the relation to be able to tell the wire for x to exclude the value 0 as shown in *(f)*.

must be 1, or *(4)* there is no value for the variable that is consistent with possible values for the other variables. More concisely, the wire is always in one of the four states: $\{0, 1\}$, $\{0\}$, $\{1\}$, or $\{\}$. (The astute reader will notice that the wire's state is just a unary relation on its variable.)

Wires are generally undirected—either end of the wire may decide to transmit information along the wire. There is only one kind of information that gets transmitted by a wire, namely that some value for the variable is to be *excluded*.

As an example, figure 1.3(a) shows a "$\neq$" relation on the three values x, y, and z. The relation shows all six possible triples that $\langle x, y, z\rangle$ might be. Next to each wire is shown the possible singletons that the variable on that wire might be. Suppose the wire for y excludes

the value 1, as shown in part (b) of the figure. When the relation receives this information, it can exclude all triples where y's value was 1, as shown in (c). Suppose that some time after that, wire z also excludes the value 1, as shown in (d). Then the relation will exclude two more triples, where z's value was 1. At this point, as shown in (e), all triples where x had the value 0 have been excluded from the relation, so it can transmit this exclusion of the value 0 for x as shown in (f).

In this case, the relation narrowed down the possible triples to a single case before transmitting an exclusion for a. However, in general, the transmission of excluded variable values may occur as soon as the relation has excluded all tuples where that value occurred.

So we see how the exclusion process can propagate. When variable values get excluded on wires, relations can exclude tuples from their list of possibilities, which in turn can lead to more variable values being excluded on the wires, and so on.

Readers familiar with constraint satisfaction problems (CSPs) will notice that this algorithm is very similar to *arc consistency*. In the typical formulation of arc consistency, vertices are variables and edges are binary relations (i.e., on two variables), but the exclusion process is the same at a conceptual level: elements exclude possibilities that are not locally possible until convergence is reached.

This process may be initiated externally, by an operator excluding a value on a dangling edge, or it may be initiated by a backtracking search program trying to find solutions to the network (in which case values on internal edges may be excluded as well). The exclusion process then propagates around the network until everything, that can be excluded, has been. What order should the propagating exclusions be processed in? It turns out that it doesn't matter. No matter what order they are processed in, once they have all been processed, there is a unique final state for the exclusion network, independent of the processing order. This is a nice result that frees us from having to specify an update order. We can let the process proceed in a distributed asynchronous fashion, and the result is guaranteed to be unique. We will prove this, along with a couple of other nice convergence theorems, in section 1.7.

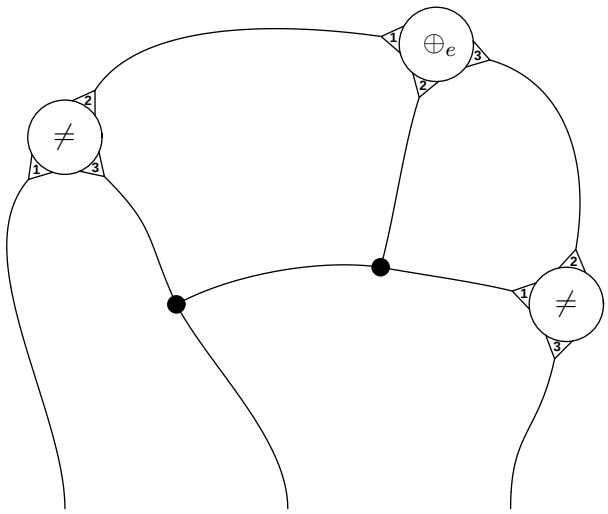


Figure 1.4: The network of figure 1.2 with some of the relations replaced by a single wire that connects several vertices instead of just two. The black dots indicate branching points of the wire.

1.4 Reusable Edges and Fan-Out

Consider the network in figure 1.4. It is the same as the left side of figure 1.2, but now the equality relations are drawn as dots. The idea is to think of the edges connected by dots as one big hyperedge (an edge with more than two ends). In the case shown, the edge provides its variable to three different relations, as well as to whatever the dangling edge might get connected to. When we allow ourselves to provide unlimited access to variables via such edges, duplicating values at will, we say that we have *reusable variables*, or *reusable edges*. We may also describe the situation by saying that "fan-out is available." An equality relation on three or more variables is often called a *fan-out* relation, and if we don't want to use hyperedges, we may think of the dots in figure 1.4 as equality relations—the overall behavior of the network is the same either way.

The vast literature of previous work on combinations of relations, whether for CSPs (e.g., [CKS01, Dec03, Nea05]), questions of implementability (e.g., [Gei68, BKKR69, Pip97]), or relational databases (e.g., [Cod90, Her97, DD00]), has almost always assumed that fan-out (unlimited repeated use of variables) is available. Some of the very few exceptions to this rule include [Fed01], [HS90], [BD97], and [Pap94] (p. 183, prop. 9.3, and p. 207, problem 9.5.4).

Our model, where the equality relation may or may not be available, is a generalization of

the historical model where fan-out is assumed to be available. When considering questions of implementability and solvability for networks of relations where the relations in the network all come from a given class, we are able to model reusable variables simply by saying that the fan-out relation is included in the given class of relations. On the other hand, in the historical analysis of models where fan-out is implicitly available, no matter how thorough the results seem, the results typically provide no information about the situation without fan-out. We stress this point because this is one of the key differences between the work in this thesis and previous work. Here, many questions about the situation without fan-out are addressed for the first time. For example, in chapter 3 we show that questions of implementability are in general undecidable, whereas previously, only the case with fan-out had been studied, in which case such questions are decidable. So we can see that the world where fan-out is optional is a bigger world than the historical reusable world.

In chapter 2 we will see that fan-out is often impossible to implement if it is not provided outright. This provides further motivation for studying the situation where it is not available. Other motivations include the cost of duplicating information, whether in a network of neurons in the brain, or in a quantum calculation where states cannot be duplicated (this was the context in which Valiant's matchgates of [Val02] were developed).[1]

1.5 Multivariate Edges

Everything so far has presented the point of view that edges are variables. However, one can also take the view that variables have their own platonic existence apart from the structure of the graph, and the edges in the graph simply represent communication channels between the vertex relations.

From this point of view, the relations know what variables they are relating, and the edges represent paths of communication between relations that have variables in common.

For example, in the network shown in figure 1.5, there are two different edges that communicate the value of variable x. On the left, there are two relations that share their information about the value of x, and on the right there are two other relations that share their information about the value of x. However, neither of the relations on the left communicates directly with either of the relations on the right about the value of x, so any

[1] Several open questions from [Val02] turn out to be answered in this thesis.

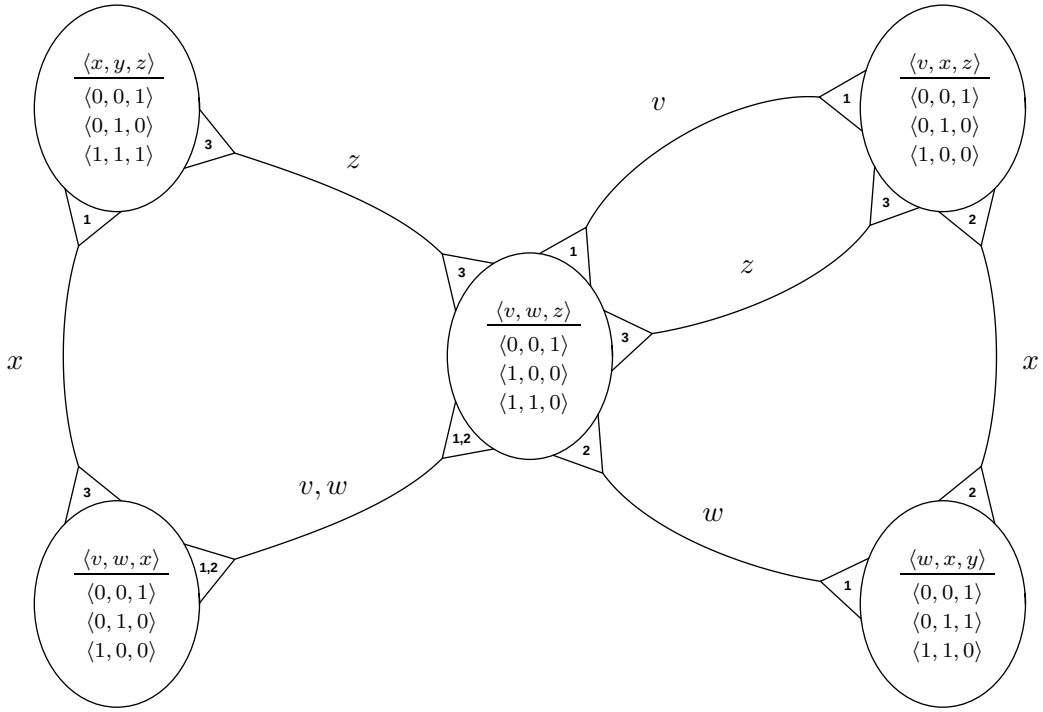


Figure 1.5: A multivariate edge network.

information deduced on the left about the value of x will never be communicated to the relations on the right, and vice versa. (The deductive ability of the network would clearly be improved if an edge were added between the left and right sides, communicating the value of x.) Similarly, the two relations with knowledge about y have no way to communicate their findings regarding y.

Another advantage of viewing edges as communication links is that an edge can communicate information about more than a single variable. For example, the edge between the center vertex and the lower left vertex in figure 1.5 communicates information about the pair of variables v and w. For the relations shown, the edge knows from the lower left vertex that the pair $\langle v, w\rangle$ cannot have the value $\langle 1, 1\rangle$, so this exclusion can be sent to the central vertex, which can then exclude its final tuple. In contrast, although the central relation does not allow v and z to both equal 0, this information cannot get transmitted to the upper right vertex, since the communication between the two is along two individual edges, one for v and one for z. Since the edge for v cannot exclude either value for v, and likewise for the edge for z, nothing gets communicated. We can see that communicating information about joint combinations of variables is more powerful than just communicating

information about individual variables.

We call the lower left edge (carrying information about the pair $\langle v, w \rangle$) a *multivariate edge*. We will call a network such as the one in figure 1.5, where edges represent communication channels rather than distinct variables, and variables are assumed to have a platonic existence and meaning independent of the network structure, a *platonic* network.

Platonic networks allow multiple edges to access the same variable at a vertex, as in the example of figure 1.5 where two different edges share information about z with the central vertex. This amounts to the same thing as fan-out being available since any such connected subgraph of edges for the same variable is equivalent in its effect to a reusable edge connecting the same set of vertices, and vice versa.

The example of figure 1.5 was designed for didactic purposes, and is otherwise a terrible example of network design. A typical network topology for a multivariate edge network might be as follows. To approximate a relation among 40 boolean variables (which cannot reasonably be fully stored in memory with current technology–if technology has improved, just increase 40 to a higher number to motivate the example), we will use a network with $\binom{40}{4}$ relations, each relating a different group of 4 variables. This network will have $\binom{40}{3}$ reusable edges, one for each different group of 3 variables. Each reusable edge is connected to every relation that uses the 3 variables of the edge. Thus the network uses $\binom{40}{4} \times 2^4 \approx 2^{20}$ bits to approximate the original relation, instead of the 2^{40} bits required to represent it directly. This strategy converts the memory requirement from exponential to polynomial, thus avoiding the "curse of dimensionality" that plagues many representation systems. The cost of this conversion is an inability to represent high-order (many-variable) relationships that do not derive from low-order relationships. Experimental evidence [HWP98] supports the view that humans similarly represent knowledge with relations of low order, and cannot directly process high-order relationships.

Networks of the sort discussed in the previous sections, in which each edge represents a unique variable, and each vertex has one incident edge for each variable related by the relation, are called *localized* networks. Localized networks can clearly be viewed as a special case of platonic networks. On the other hand, any platonic network can be converted into an equivalent localized network in two steps: First, we convert any multivariate edge into an edge for a new variable, whose possible values are the possible tuples of the multivariate edge. Then we convert each relation to specify a distinct variable for each edge, expanding

or contracting its set of tuples in the obvious way. For variables represented in disconnected regions of the network, like x or y in figure 1.5, localized networks take the view that it is completely meaningless to say that the two variables are the same.

1.6 Definitions

The time has come to state things more precisely. This will come as a relief to some readers, while others may only have the patience to read a single definition before skipping ahead to the next section.

We will not limit ourselves to defining only a single term under each "Definition" heading. We hope this is all right with the reader.

1.6.1 Relations

Definition 1 *An* n-tuple *is a sequence of length* n*, where* n *is a non-negative integer. A* singleton *is a 1-tuple. A* pair *is a 2-tuple. A* triple *is a 3-tuple. A* tuple *is an* n*-tuple for some unspecified* n*.*

We often write a tuple with angle brackets, as in $\langle x, y, z \rangle$. We occasionally call a tuple a *vector*.

Definition 2 *A* relation *is a set of* n*-tuples, for some* n*. The number* n *is called the* order *of the relation.*

Definition 3 Unary *means order 1.* Binary *means order 2.* Ternary *means order 3.* k-ary *means order* k*.*

According to this definition, the empty relation does not have a well-defined order. One can imagine a slightly more complicated definition, where a relation could be empty but still have an order, but since it won't make any difference for what we do, we will use this simpler definition.

Note that a relation, in itself a set of n-tuples, has no knowledge of which particular variables it is relating. Indeed, the same relation may be used in many places, relating a different set of variables each time. To describe a relation that is relating specific variables, we speak of a relation "on the variables."

Definition 4 *A* relation on variables *(e.g., on x, y, and z) is an ordered pair $\langle H, R\rangle$ where H is an n-tuple of variables (e.g., $\langle x, y, z\rangle$), and R is a relation whose tuples are of length n. H is called the* heading *for the relation. The variables in H are the variables that the relation is on.*

You can think of the heading as the headings for a table of values, as shown in each relation in figure 1.5. Note that a heading might list a variable more than once, as could happen in a localized network with self-loops (defined below).

Definition 5 *Given values for some variables, a relation on those variables is* satisfied *if the heading H, with variables replaced by their values, is one of the tuples of the relation R. The relation is* unsatisfied *(or* dissatisfied*) if the heading (with variables replaced by values) is* not *one of the tuples of R.*

Definition 6 *If the number of possible values for a variable is n, then we say that the variable is* n-valued*. If the number of tuples in a relation is n, then we say that the relation is* n-valued*.* Boolean *means 2-valued.*

1.6.2 Networks of Relations

When we talk about a graph for a network of relations, we are generally talking about a richer structure than the standard graph theory $G = (V, E)$ where E is simply a set of pairs of vertices. For example, we allow *self-loops* (edges with both ends at the same vertex) and *multiple edges* (more than one edge between the same pair of vertices). And furthermore, the relations at the vertices know which edges have which variables and which positions of the relation's tuples correspond to those variables.

Definition 7 *A* network graph *(usually just called a* graph *or* network*) is an ordered pair* $G = (V, E)$ *where* E *is an arbitrary set (whose members are called* edges*) and* V *is a sequence* $v_1, v_2, \ldots, v_n$ *where each vertex* $v_i \in V$ *is a set of ordered pairs* $\langle e_{i,j}, c_{i,j} \rangle$ *called* connections *(index* i *here specifies which vertex, index* j *specifies which of the vertex's connections), where* $e_{i,j}$ *is an element of* E *(i.e.,* $\exists a \in E$ *such that* $e_{i,j} = a$*), and* $c_{i,j}$ *is a tuple of distinct positive integers, called the* indices *of the connection. All of the connections, among all vertices, for a given edge must have the same number of indices at each connection; this number is called the* dimension *of the edge.*

In the figures, vertices are shown as circles or ovals, and the reader has probably noticed pointy cones sticking out where they are connected to edges. Each cone represents a *connection*, and the tiny numbers shown inside the cone are the *indices* of the connection. In almost every example so far, each $c_{i,j}$ has been a singleton, as all edges have been one-dimensional, except for a lone two-dimensional edge in figure 1.5.

Definition 8 *A* network of relations *is a quadruple* $N = (X, G, M, R)$ *where* X *is a set of variables,* G *is a network graph* $G = (V, E)$ *with* n *vertices,* M *is a function from* E *to tuples of variables from* X*, and* R *is a sequence* $R = r_1, r_2, \ldots, r_n$ *of relations on subsets of variables of* X*. For each connection* $\langle e_{i,j}, c_{i,j} \rangle$ *of vertex* v_i*, the tuple of indices* $c_{i,j}$ *must be of the same length as the tuple of variables* $M(e_{i,j})$*, and we call this length the* dimension *of the edge. (This extends the previous definition to cover edges which participate in no connections.) Furthermore, if* $M(e_{i,j}) = \langle x_1, \ldots, x_d \rangle$ *and* $c_{i,j} = \langle b_1, \ldots, b_d \rangle$*, then the heading of relation* r_i*, say* $H = \langle h_1, \ldots, h_p \rangle$*, must satisfy* $b_k \leq p$ *and* $h_{b_k} = x_k$ *for each* $k \in [1, d]$.

The final part specifies how the indices of a connection indicate the correspondence between the variables of the edge and the variables of the relation. In the common case that an edge has only one variable, then $d = 1$ and the condition is simply that the b^{th}

variable of the relation should be the variable of the edge, where b is the index of the connection.

Definition 9 *A* multivariate edge *is an edge of dimension other than 1. A* univariate edge *is an edge of dimension 1.*

Definition 10 *A* platonic *network of relations is a network of relations as defined above. A* localized *network of relations is one in which M is a bijection from E to* $\left\{ \langle x_i \rangle \;\middle|\; x_i \in X \right\}$.

Note that all edges of a localized network must clearly be univariate. In a localized network, edges and variables are in perfect 1-1 correspondence with each other, so given the network graph, one can infer that there must be a unique variable on each edge. Thus the variables are usually omitted from the sketch of a localized network, as in figure 1.1, and we speak of edges as having values.

Definition 11 *The class of* networks of relations with reusable edges *is the class of networks of relations as defined above. The class of* networks of relations without reusable edges *consists of only those networks in which every edge is used in either one or two connections (among all the connections of all the vertices).*

Note that these two classes are not complementary. The class of networks of relations without reusable edges is a subset of the class of networks of relations with reusable edges. This is because, when talking about networks with reusable edges, we do not intend to exclude networks that just happen to use every variable just once or twice.

Definition 12 *When we talk about networks* without fan-out*, that means we are restricting ourselves to the class of networks of relations without reusable edges.*

Definition 13 *In a network without fan-out, a* dangling edge *is an edge that is used in only one connection. An* internal edge *is an edge that is used in two connections.*

Of course, if we are not restricting ourselves to networks without fan-out, then any edge may be used in an additional location if the network is expanded, so in effect all edges are dangling. We may however still wish to treat some edges as *internal* edges, and others as *dangling* edges, as was shown for example in figure 1.4. To allow this perspective, we will define the notion of a network *portion* for the case of reusable edges.

Definition 14 *A network* portion *is an ordered pair $P = (N, D)$ in which N is a network of relations and D is a subset of the edges. The edges in D are called* dangling edges*, and the edges not in D are called* internal edges*.*

Definition 15 *By a* network specification *we will mean a network portion or a network without fan-out. A network specification is* closed *if it has no dangling edges. A network specification is called a* fragment *if it does have dangling edges. Variables in the tuples $M(e), e \in \{$dangling edges$\}$, are called* accessible *variables.*

We can now define the satisfiability of a network specification.

Definition 16 *Given values for the accessible variables of a network specification (whether closed or a fragment), the network specification is said to be* satisfiable *if there exists a way to assign values to the remaining variables such that every relation in the network is satisfied.*

The idea of an *implementation* will be the subject of much of chapter 2.

Definition 17 *A network specification is said to* implement *a relation R if there is a relation $R' = \langle H, R \rangle$ on the accessible variables such that the heading H lists each accessible variable exactly once, and the network is satisfiable for exactly those values of the accessible variables for which R' is satisfied.*

Note that every closed network specification implements either the happy relation on zero variables or the unhappy relation, depending on whether it is satisfiable or not.

1.6.3 The Exclusion Process

Now we can start to define the exclusion process, where relations send messages which are their projections, and mask themselves by messages that they receive.

Definition 18 *Given a relation R on variables $H = \langle x_1, \ldots, x_n \rangle$, its* projection *onto a subset of those variables $H' = \langle x_{s_1}, \ldots, x_{s_k} \rangle$ is the relation R' having a tuple $T' = \langle y_{s_1}, \ldots, y_{s_k} \rangle$ for every tuple $T = \langle y_1, \ldots, y_n \rangle$ of the original relation R.*

Example 1 *If $H' = H$, then the projection R' is the same as R.*

Example 2 *If H' is the empty set, then the projection R' is either the unhappy relation or the happy relation on no variables, depending on whether R was the unhappy relation or not.*

Definition 19 *Given a relation R on variables $H = \langle x_1, \ldots, x_n \rangle$, and a relation R' on a subset of those variables $H' = \langle x_{s_1}, \ldots, x_{s_k} \rangle$, the* masking *of R by R' is the relation R'' on variables H containing just those tuples $T = \langle y_1, \ldots, y_n \rangle$ of R for which the tuple $T' = \langle y_{s_1}, \ldots, y_{s_k} \rangle$ is in R'.*

Example 3 *The masking of a relation R by some projection R' of R results in a relation R'' that is exactly the same as R.*

Definition 20 *The* exclusion process *on a network of relations is a distributed asynchronous process in which each vertex updates its relation by excluding tuples so as to shrink the relation. Every connection spontaneously and repeatedly updates the other vertices connected to its edge. Specifically, the connection projects its vertex's relation onto the variables $M(e)$ of its edge to get a relation r', and then it replaces each neighbor vertex's relation r with the relation r'' obtained by masking r by r'. This process continues until no connection can*

change any of its neighbor vertices (i.e., until all possible updates have no effect). If the process continues forever (as could happen for relations with an infinite number of tuples, on variables that can have any of an infinite number of values), then any connection which can *change a neighbor vertex* must *do so eventually (i.e., in a finite amount of time).*

1.7 Convergence Theorems

Theorem 1 *For discrete variables, the exclusion process always converges to a stable state.*

Proof: Since the variables are discrete, they only have a limited number of values. Each step of the process can only exclude more values. Once a value has been excluded, it remains excluded for the duration of the process. Thus the process must end at a stable state, when no relations can exclude any more values. ■

Theorem 2 *For continuous variables, the exclusion process converges, but not necessarily to a stable state.*

Proof: In this case, convergence must be defined more carefully, as the exclusion process need not end in a finite amount of time. We say that the state of the network converges if the state (excluded vs. not) of each value for each variable has a limit as time $t \to \infty$. But each value necessarily has such a limit, since if it is ever excluded, its state cannot change further. So values that ever get excluded are excluded in the limit, whereas values that are never excluded are not excluded in the limit. This limiting state is what the exclusion process converges to. However, the process might not ever reach the limit for any $t < \infty$, and thus some further exclusion which is warranted by the limit state may never be warranted for $t < \infty$. and thus may never occur, and therefore not be excluded in the limit state. In this way, the limit state need not be a stable state. ■

Example 4 *If we have real-valued variables w, x, y, and z, and three relations R, S, T, and U, where R specifies that $w + 1 < x$, and S specifies that $x + 1 < y$, and T specifies that $y + 1 < w$, and U specifies that $z = sign(x)$, then if we start the exclusion process by*

excluding all negative values for w, then we can see that the limit of the process will be that all values are excluded for w, x, and y, but the value of 1.0 *for z will not be excluded. This limit state is not a stable state, since from this state, the value of* 1.0 *for z could also be excluded.*

Note that if time progresses not just to infinity but through ordinals of higher cardinality than that of the set of possible values for the variables, then the limiting state after such time has passed must indeed be a stable state. (Perhaps most readers will feel that such an abstract statement is of little practical interest, but I include it for the more mathematically-minded readers.)

Theorem 3 *For discrete variables, the exclusion process described above converges faster than you can say what the current state of the process is.*

Proof: If each newly excluded value is considered as one step of progress, then the maximum number of steps that can occur is $\sum_{(\text{vars } v)} |\sigma_v|$, where σ_v is the set of values that variable v might take. To describe the current state of the process, you need to specify which values are plausible for each variable, which takes $\sum_{(\text{vars } v)} |\sigma_v|$ bits. ■

This maximum number of steps only occurs if everything gets excluded (which is indeed a stable state), which only happens if the supplied partial input was already inconsistent with the network's relations.

For continuous variables, we will see in section 1.13 that there is *no* guaranteeable rate of convergence.

Our final theorem is that the limit state (which is the final stable state in the discrete case) is unique, meaning that asynchronous implementations do not need to worry about any subtle effects arising from the asynchrony. The uniqueness of the limit state is due to the monotonicity of the exclusion procedure.

Theorem 4 *The order in which relations update their information has no effect on the final limit state.*

If one is familiar with theorems regarding confluence, this can easily be proved with the help of such theorems, since it is easy to show that if either of two variable values v and w

could be excluded at the next step, then the other of the two could be excluded at the step after that, and the effect of excluding v and then w is exactly the same as excluding w and then v. This is known to be a sufficient condition to imply confluence, which in our case would imply that the limit states are the same. The theorem is also fairly easy to prove in the case that the variables are discrete, since then there are only a finite number of possible states of the system, and any limit state must also be a stable state. Instead, we give a proof that works in the general case, and stands on its own. (Some people feel this theorem to be intuitively almost self-evident, but our proof should convince even a skeptic.)

Proof: If we define a state of the network as a list of all variable values which have not yet been excluded, then we get a partial ordering on states where $s_i \preceq s_j$ means that state s_i is a subset of state s_j. It is straightforward to see that if $s_i \preceq s_j$, then any variable value excludable at the next step by s_j is also excludable or already excluded by s_i. Since the sequence of states $s_1, s_2, s_3, \ldots$ produced by the exclusion process is steadily decreasing in this partial order, it is the case that any variable value that is excludable at some point during the exclusion process will also be excludable (if not already excluded) at any later point in the process. Thus, if we consider any variable value v which could be excluded on the next step of the exclusion process, it is impossible that, were we to choose some other update order, v would at some later time not be excludable. Therefore in *any* limit state, v will be excluded, by the definition of the exclusion procedure.

Now suppose there are two possible limit states, s_1 and s_2, that could be reached from the current state. We have just shown that the next variable value v to be excluded, regardless of whether it should be excluded next if one is trying to reach a particular limit state, is known to be excluded in both s_1 and s_2. But not only will s_1 and s_2 both be $\preceq$ the new state, but we also know that s_1 and s_2 will both still be reachable if we exclude v at the next step, since we can still proceed with whatever sequence of exclusions was previously necessary in order to reach the desired limit state: Anything we could have excluded without excluding v is also excludable with v already excluded, and v itself is known to be excluded in the desired limit state. Thus, for any excludable variable value v, we know that v is excluded in all reachable limit states, and we know that we may exclude it immediately without affecting which limit states are reachable.

Since this is true at each and every step of the exclusion process, we see that we can only ever exclude variables values that are excluded in both s_1 and s_2, and thus any limit

state must be contained in $s_1 \bigcap s_2$. But since s_1 and s_2 are themselves limit states, we have $s_1 \subseteq s_1 \bigcap s_2$ as well as $s_2 \subseteq s_1 \bigcap s_2$, which together imply that $s_1 = s_2$. Thus there cannot be two different reachable limit states. ■

1.8 Zipper Networks

We can take our analysis to a lower level, and develop a structure to represent what is going on inside the edges and inside the vertices. This analysis applies to localized graphs, not platonic graphs.

The structure consists of *zipper lines* (sometimes also called *wires*) that run through *pipes*, as shown in figure 1.6. Each connection is turned into a pipe segment containing zipper lines that connect the tuples of a vertex to the values of an edge (or to the tuples of a multivariate edge). Each place where two zipper lines merge into one is called a *zipper*.

A solution to a zipper network consists of a subset of the zipper lines such that every section of pipe contains one zipper line from the solution, and at an n-way pipe junction, the solution contains one n-way zipper line junction. That is, the structure of the solution must match exactly the structure of the pipes. An example of this is shown in figure 1.7.

1.8.1 Burning

The exclusion process is particularly simple when viewed at the zipper level: A value being excluded for a variable, or a tuple being excluded at a relation, corresponds to a zipper junction burning up. The burning of a zipper wire propagates along the wire. If it comes to a zipper that splits the wire into two wires, then both wires burn. If it comes to a zipper that merges the burning wire with another wire, then the burning comes to an end at the zipper, and the zipper disappears (as only a single wire remains). If it comes to a junction of k pipe segments (and thus of k zipper lines), then the burning continues along the $k - 1$ other zipper wires connected to the burning wire at the junction. The point of these rules, shown in figure 1.8, is that a wire burns if its presence in a solution would directly necessitate the presence of a neighboring wire which has already burned.

In a zipper network in which some wires are burnt, a solution is required to consist entirely of unburnt wires.

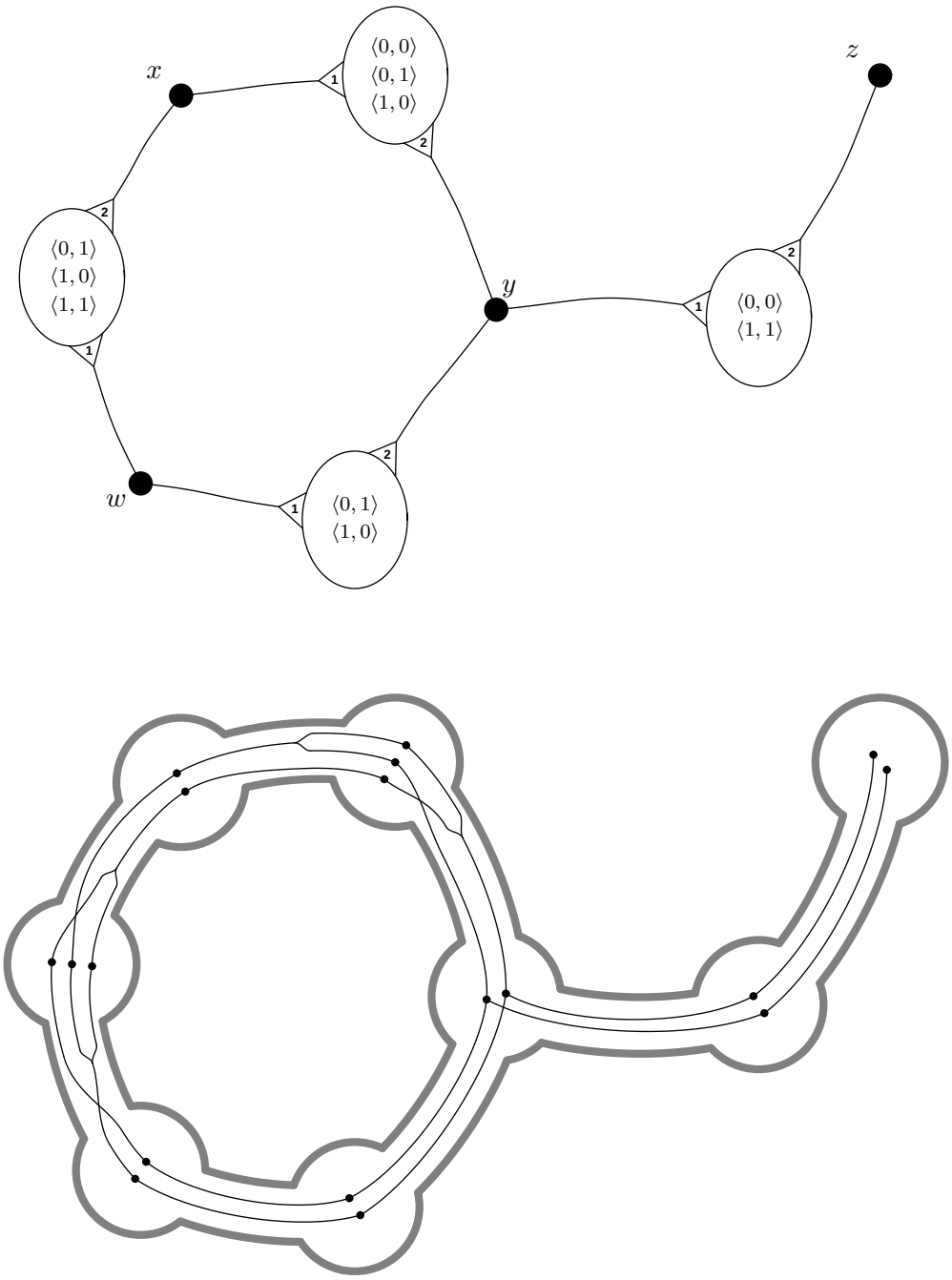


Figure 1.6: The upper diagram shows an exclusion network with reusable variables, and the lower diagram shows the same network converted into zippers and pipes. Each variable or relation turns into a *pipe junction*, and each connection turns into a *pipe segment*, shown in thick gray. Then, within the pipes, the zipper lines show the connectivity between the possible variable values and the tuples in the relations. Each tuple in a relation is turned into a *zipper junction*, shown as a black dot inside the pipe junction, as is each possible value for a variable. Zipper lines connect variable values with those tuples which contain that variable value. Where one line splits into two, we call it a zipper.

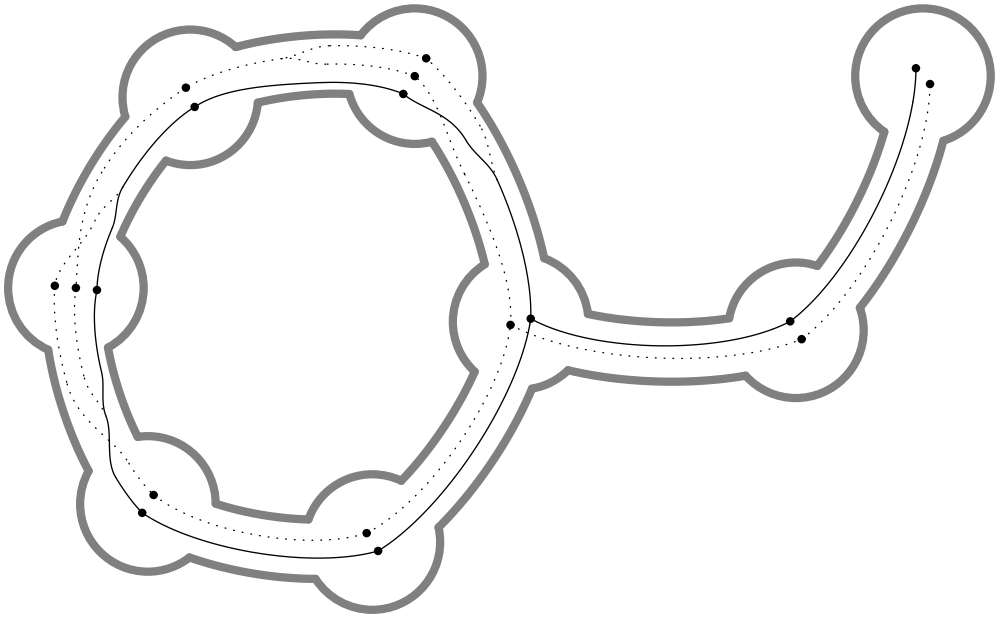


Figure 1.7: A solution to a zipper network is a set of zipper lines having exactly the same structure as the pipes. The solid lines here show one possible solution to the zipper network of figure 1.6. Although the variables and relations are long gone, the solution shown here corresponds to the original solution $\{w = 1, x = 1, y = 0, z = 0\}$ of the network of relations shown in figure 1.6.

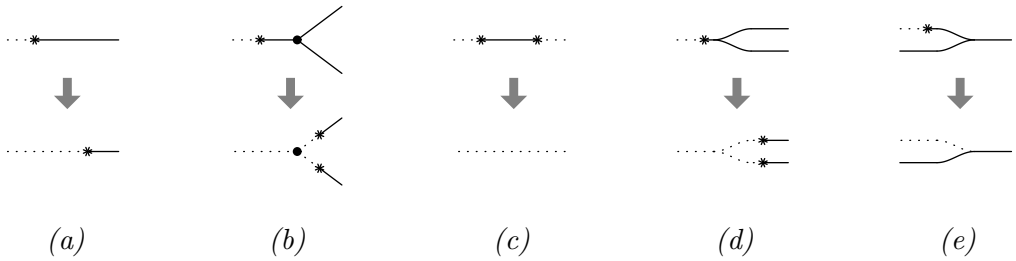


Figure 1.8: How zipper lines burn. A zipper lines can burn up like a fuse. Unburnt lines are shown as solid lines, and burnt lines are shown as dotted lines. A spark is shown where the end is burning. In (a) we see how the spark progresses along a stretch of zipper line, burning it up. In (b) we see how a spark spreads in all directions at a junction. In (c) we see that when two sparks meet, they disappear, leaving the wire fully burnt. In (d) we see how a spark can be split by a zipper. In (e), we see that a spark coming in on just one branch of a zipper does not have the power to burn the full merged line, and the spark dies at the zipper. However, once only a single branch of the zipper is left, as in the lower part of (e), it ceases to be a zipper, and it will thereafter behave as in (a) if a spark approaches from either side.

From the zipper point of view, it is easy to prove that there is a unique result of all this burning. Suppose there were two different possible final sets of burnt wires, A and B. Say A contains one or more wires not in B. Consider the first such wire that burns, on the way to final set A. This wire was ignited based solely on the burning of wires in B, so if all of B burns, then this wire must burn as well, contradicting the assumption that B was a final (maximal) set of burnt wires. Therefore there cannot be two differing final states A and B, but rather just one unique final state.

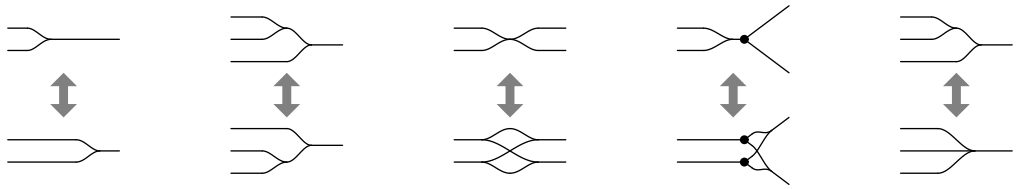
1.8.2 Zipping Up and Unzipping

A zipper may clearly be moved back and forth along a zipper line without affecting the structure of a zipper network. Furthermore, a zipper can be moved past other zippers, and past junctions, according to the zipper rules shown in figure 1.9, without affecting the number of solutions to the zipper network.

We say that a zipper is *facing* the direction in which it has two zipper lines. If we move the zipper towards the side with two lines, we say it is moving in the "zipping-up" direction. If we move the zipper towards the side with just one line, we say it is moving in the "unzipping" direction.

Since the rules always allow an unzipping zipper to progress past other zippers and junctions, unzipping can continue forever if there is a cycle in the pipe structure. If there is no cycle, then we can unzip all the zippers until they fall off the leaves of the tree. At that point, the solutions of the tree network are disjoint identical copies of the pipe structure, with no zippers present. This also shows that in a zipper network whose pipe structure is a tree, every single zipper line is part of one or more solutions. This is not true in general. For example, in figure 1.6, the lower right zipper line in the "tail" of the ring is not part of any solution. We call such a line a *red herring*, denoted by RH^+. A zipper line which *can* be part of a solution to the network is called RH^-. If a zipper network has the property that for any stable state reached by the burning process (including the initial state where nothing is burnt), all RH^+ wires are burnt (where the RH factor of an edge depends on the current burnt state of the network), then we say the zipper network is a *perfect* network. We see that if the pipe structure is a tree, then the zipper network is perfect.

If we try to zip things up as much as possible, in the hopes of finding a concise canonical



(a) *(b)* *(c)* *(d)* *(e)*

Figure 1.9: The zipper rules. In (a) we see that a zipper can move back and forth on a wire, unzipping it or zipping it up. In (b) we see that if two zippers are facing the same way, then one can pass by the other. In (c) we see how two back-to-back zippers can simultaneously unzip each other, resulting in four zippers (since each of the two zippers gets split in two by the other). Of course, if four zippers are facing each other in the right way, the reverse operation is also possible. In (d) we see that a zipper can unzip a junction, continuing along each branch. The reverse operation is possible if zippers are facing each other along all but one of the branches of a junction—in this case, they can zip the junction together and merge into a single zipper. In (e) we see that we can also think of multi-way zippers, and if we do, we should allow smaller zippers to merge into larger zippers, and larger zippers to split up into smaller zippers. The generalization of (a) through (d) for multi-way zippers is straightforward, and if in doubt, multi-way zippers can always be expressed as a sequence of 2-way zippers via (e), and then the multi-way operations correspond to sequences of 2-way operations. Generalizing the junction rule in (d) down to smaller junctions, we can see that 2-way junctions have no effect on the zippers (or on the burning process), so the presence or absence of a 2-way junction in a pipe is completely irrelevant to the zippers. For 1-way junctions (i.e., vertices of degree 1), the junction rule can seem a little peculiar: An unzipping zipper can fall off the end (but a zipper that is zipping up cannot), and a zipper can appear out of nowhere to zip the lines together (but not to zip them apart). These peculiarities are easy to remember either by noting that these are the operations that necessarily preserve the number of solutions, or by imagining that the lines all merge together just past the end, in which case these are the regular multi-way zipper operations. As an example, the "tail" of the zipper network in figure 1.6 could zip its two lines together right up to the 3-way junction.

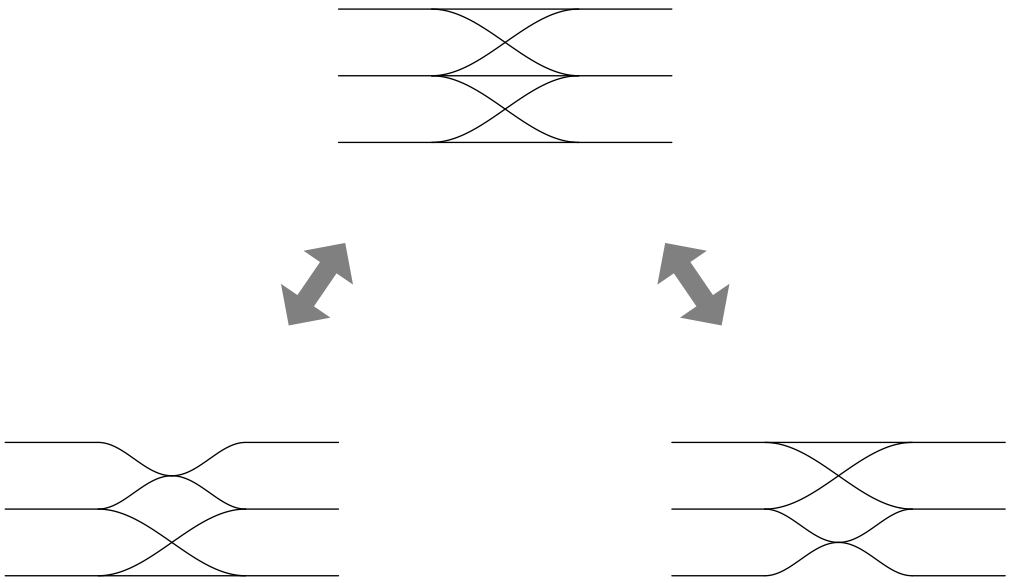


Figure 1.10: An effort to zip things up maximally may not have a unique result. Here the arrangement at the top can be converted to either of the lower arrangements by applying zipper rule (e) to each of the 3-way zippers followed by rule (c) in the collapsing direction. But the two lower arrangements cannot be zipped up any further, thus demonstrating that no particular result can be guaranteed if one simply zips up everything in sight. However, it is not hard to tell whether two arrangements (within a single pipe segment) are equivalent under the zipper operations: One simply needs to *un*zip everything in sight, and use rule (e) to merge zippers into multi-way zippers. This yields a canonical form; in this case it will convert each of the lower two arrangements back into the upper arrangement. This canonical form can be represented as a connectivity matrix with no loss of information; for example the upper arrangement can be represented as the following matrix:

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

form, we will find that there does not seem to be any unique canonical way to maximally zip things up, as in the example in figure 1.10. However, what we *can* do is *unzip* things as much as possible, at least within a single pipe segment. This results in every zipper having its back to a junction at the end of the pipe segment, since zippers never get stuck when they are unzipping. If the zippers with their back to a given zipper line junction are merged into a multi-way zipper, then we see that we have a simple canonical form for the pipe segment, in which the number of zipper lines connecting two particular multi-way zippers at the ends is exactly the number of distinct paths that connected those junctions at the outset. This canonical form will be the basis for the matrix networks of section 1.10.

1.8.3 The String Cheese View

Readers familiar with string cheese can view zippers as similar to the splitting of string cheese. When two zippers meet back to back and unzip each other as in rule (c) of figure 1.9, the splits in the cheese are always assumed to be orthogonal, so that every possible combination appears somewhere. If we think of the cheese as a bundle of infinitely thin fibers which are parallel but *not* stuck together in the typical cheesy way, then since no individual fiber is actually split by a zipper, we can see that the precise location of a zipper no longer has any objective meaning, and all of the zipper operations of figure 1.9 can be seen to be no-ops.

1.8.4 A Recursive Strategy

Zippers do provide a simple way for mechanically finding the total number of solutions of any network, although it might require an impractical number of zipper lines. For this, we perform the following operation: pick any little stretch of pipe in which there are no zippers, and label the zipper lines in it each with a different label. Then have these labels propagate just as a burning wire propagates. Due to the rules for burning, any line receiving a label *implies* the originally labeled line of the same label, where "one wire implies another" means that if one wire is in a solution, then the other wire must be in the solution as well. The propagation of these labels can only get stuck where two or more labels merge at a zipper. In this case, we start unzipping the zipper to allow the labels to propagate further. In this way, the labels can propagate around the network so that every wire in the network winds

up getting a label. However, we must specify what happens when propagating labels arrive to some point from both sides, as will happen if the pipe structure is not a tree. When propagating labels meet each other, we let them go just slightly past each other, so each wire in that cross-section of pipe has a label from each direction. Then, any wire whose two labels do not match is burnt (as it implies two different wires back where the labels were originally applied). It is not too hard to see that this causes any zippers that were being pushed by the labels to disappear, so the propagation of labels at that point stops.

In this way the labels propagate throughout the zipper network until every zipper line has been labeled, and at the end all the zippers that are left have the same label on all of their wires. This means that the zipper line structure now consists of disjoint components, one per label. Each component can then have its label removed and be processed in the same way again, starting in some other pipe section where it has more than one possible wire. We can continue recursively until all the zippers are gone. Why will they eventually be gone? Because the number of zippers *per component* is strictly decreasing—the unzipping of a zipper in this process is always separating a wire (or zipper or junction) into wires (or zippers or junctions) that will be in distinct components.

This recursive process corresponds roughly to performing a traditional recursive search for a solution by setting more and more edge variables to values. The zipper point of view gives us a nice way of visualizing the information flow of such a process.

1.8.5 Shallow Search

There is another operation which is useful for getting rid of RH^+ wires, which amounts to doing a depth-limited search to see if any wires can easily be ruled out. The idea is to pick a zipperless stretch of pipe in an area we suspect of having many RH^+ wires. Then we burn all but one of the wires in that stretch of pipe, and we keep track of what burns. We then pick a different wire in that stretch of pipe and start over, burning all wires except for our new pick, and making a note of what burns. We continue doing this (burning all but one of the wires) until we have done it for each wire in that stretch of pipe. Then we compare notes to see if there are any wires, anywhere in the network, that burned *every time*. Any such wire is clearly an RH^+ wire, since it cannot be part of a solution regardless of which wire in that stretch of pipe is part of the solution. Then these wires that are known to be

RH^+ may of course be burned not only to simplify the network, but also to improve the network's deductive capability.

The previous paragraph described a depth-1 shallow search. We could do a depth-2 shallow search by picking a second zipperless stretch of pipe elsewhere in the network, and then each time we burn all but one of the wires in the first stretch of pipe, we continue by burning all but one of the wires in the second stretch of pipe (again repeating this for each wire in the second stretch of pipe), so that wires are even more likely to burn every time. Again, at the end, any wire that got burned every time is known to be RH^+ and can be burned outright without affecting any solutions to the network. The deeper the shallow search that we do, the better our ability to detect and eliminate RH^+ wires. Of course, if one tries a shallow search of unbounded depth (so that it is not shallow after all), then one is simply performing a depth-first search for all solutions to the network.

Recall that the spreading labels of section 1.8.4 can result in wires getting burnt (when they receive two different labels). The wires that burn from one stage of that process are exactly the same wires that are eliminated in a depth-1 shallow search, assuming you use the same stretch of pipe to initiate both processes. Indeed, the wires that burn from n stages of that recursive process are the same as the wires that burn from a depth-n shallow search. The growth in wires (space) required by the parallel recursive process corresponds to the growth in time required by the sequential shallow search.

1.9 Relation-Variable Duality

In any solution to a network of relations, each relation has effectively chosen one of its many possible tuples. Extending this viewpoint, we can expand the network in the following way: View each relation as a variable whose possible values are the tuples of the relation. Then, each edge must be changed to a relation that enforces agreement between the tuples chosen by the two relations for the variable(s) of the edge.

This expansion is appropriate in the context of reusable variables, since a relation on n variables becomes a variable used in n relations, while a variable used in k relations becomes a relation on k variables.

After we have expanded the relations into variables (and the variables into relations) in this way, we can shrink the network back down by merging wire values according to the

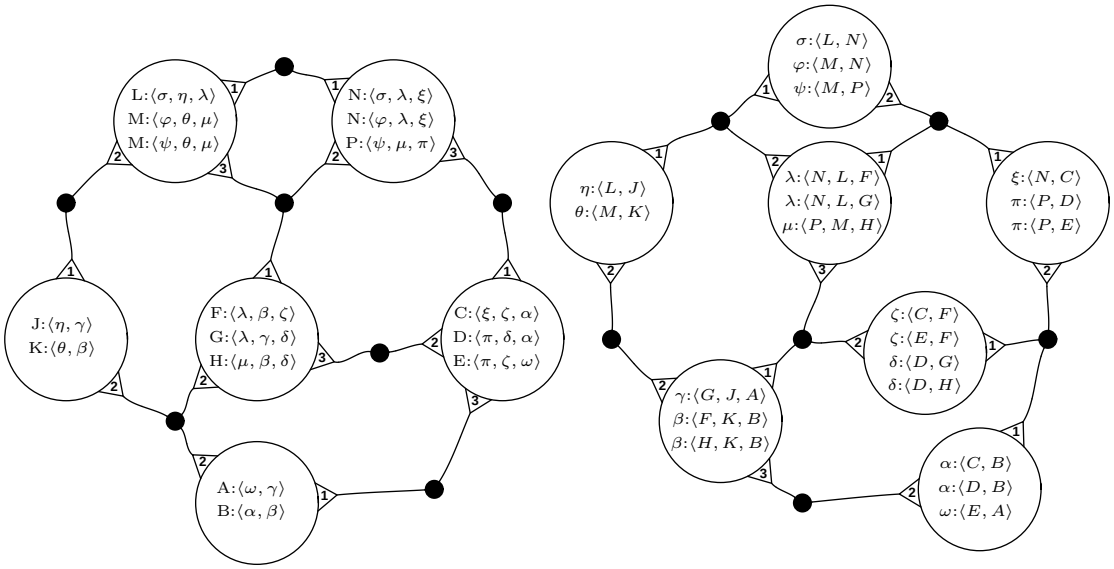


Figure 1.11: A network of relations and its dual. In the dual, every relation has become a variable, and every variable has become a relation. The label shown in front of each tuple is not part of the relation in which it appears, but rather indicates a possible value for the variable at that location in the dual. A single label can appear more than once as a result of values being merged during the shrinking stage. For clarity, disjoint sets are used for the possible values of each variable, rather than giving every variable the same set of possible values (e.g. $\{0, 1\}$ or $\{0, 1, 2\}$). The variables themselves are not shown; only their values are shown. The dual of the dual is the original network.

following shrinking rule: If two (or more) of the values for a wire arose from a set of tuples T of the original relation, where T is the outer product $\{t_1 \otimes t_2 \otimes \ldots t_k\}$ of some variable values t_i for each original variable v_i connected to that relation, and the values in t_i appear only in the tuples T, then those values arising from T should be merged (subject to the technical condition that the merged value appear in as many tuples of a new relation as the corresponding original variable had values in t_i—but this condition, which is needed only to preserve the exact number of solutions, can be ignored if one uses multi-relations as in section 1.12, since then the condition can be satisfied simply by setting the multiplicity appropriately). For example, in figure 1.11, the bottom two tuples in the upper left vertex can be expressed as the outer product $\{\varphi, \psi\} \otimes \{\theta\} \otimes \{\mu\}$, and since none of $\varphi, \psi, \theta, \mu$ are used (in the same position) in any other tuple in that relation, they get mapped to a single variable value, M, in the dual.

The purpose of the shrinking rule is to counteract the expansion operation. We can define the dual of a network to be the result of expanding it and then shrinking it. If we

start with a network that cannot be shrunk, then in general the dual of its dual is the original network. (That's the reason we call it the dual.) It is easy to see that the dual has exactly the same number of solutions as the original network, as both expansion and shrinking preserve the number of solutions.

It turns out that the expanding and shrinking operations are even more easily expressed in the zipper point of view: Given a graph with reusable variables, we consider a zipper graph with a junction for each relation and a junction for each variable. Note that the zipper graph is bipartite, with one part being the relation junctions and the other part being the variable junctions. The zippers start out facing the relation junctions, so that each variable value is connected to the tuples it appears in.

Then, the expansion operation is simply the unzipping of every zipper to just past the next junction. And the shrinking operation is simply the application of the zipper-junction rule in the zipping-up direction along with the facing-zippers rule, to be applied when there is a unique way of zipping together values of the variable so the zippers don't get stuck. The zipper point of view makes it much more intuitive why the dual of a dual would be the original network, assuming the original network was unshrinkable.

1.10 Matrix Networks

In this section we will look at how networks can be represented with matrices, and we will examine the question of how one might transform these matrices by performing a change of basis.

1.10.1 Edge Matrices

If we look at a pipe segment in a zipper network, we see that between the two relations at the ends, the zipper lines provide a number of paths. In fact, for any given tuple in the relation at the left end, and any given tuple in the relation at the right end, the zipper lines provide zero or more distinct paths between those two entries. Furthermore, zipper operations within this pipe segment all preserve this number exactly. We can create a matrix of these numbers, with a row for each tuple of the left relation, and a column for each tuple of the right relation, where each entry gives the number of distinct paths

connecting the row's tuple on the left with the column's tuple on the right. This matrix is unchanged by zipper operations within the segment. In fact, it is easy to see that matrices of the appropriate dimensions having non-negative integer entries can be put in 1-1 correspondence with canonical forms for the zippers. The meaning of the matrix is simply how many solutions that pipe segment has for each possible pair of tuples at the two ends. We can represent every edge in a network of relations by such a matrix, and then the relations can be eliminated from the vertices, losing no information. We call the result a *matrix network*.

Suppose a vertex in a matrix network has degree 2. Then we can simply multiply the matrices on either side together to get a single matrix giving the correct number of paths between any pair of tuples from the vertex's two neighbors. (Depending on how the matrices were constructed, one or both of them may need to be transposed first. Since it is obvious from the meanings of the indices when this sort of transposition needs to be done, we will assume such transpositions will be done as needed in the discussion below.)

If a matrix network has the shape of a cycle, then we can multiply the matrices together all the way around the cycle until we just have a single matrix, which will be a square matrix. We can see that the trace of this matrix (the sum of the entries on its main diagonal) is then the total number of solutions for the original network of relations that was converted to this matrix network.

Can we extend this to general graphs, so that a simple sequence of matrix multiplications can give us the number of solutions to an arbitrary network of relations? Not in any practical way, we can't, because one of the things we could then do this way is solve NP-complete problems such as circuit satisfiability.

How about for trees? For example, zippers on trees can easily find the number of solutions, so we know that finding the number of solutions for a tree of relations is tractable. How would we do it with matrices? Well, a tree must have a leaf, and leaves are of degree 1, so they are particularly easy to zip and unzip using the zipper-junction rule. If we fully unzip it, and then zip all the tuples together into a single tuple with one large zipper, then the matrix on its edge becomes a vector. This vector can then be included in one of the other matrices on an edge touching the parent of the leaf, simply by multiplying each row by the corresponding element of the vector. In this way, the leaf can be pruned. We can repeat this process until there is just one edge left, and at that point the sum of all entries

in the remaining matrix gives the total number of solutions to the network of relations. So trees are efficiently solvable, where solvable means counting the number of solutions.

We can combine the method used for a cycle with the method used for a tree, so if we have a tree of cycles (i.e., a graph in which no two vertices are connected by more than two disjoint paths), then at each step we can either prune a leaf as described, or we can similarly prune a "leaf cycle" (a cycle having only one vertex of degree greater than two), by going around the cycle as described, starting and ending at the high-degree vertex, and then using the trace as the vector to multiply into another matrix as done for the leaf.

Since the trace is a basis-independent quantity, we may wonder if the types of matrix operations we are doing are all basis-independent operations. To understand how this question can even make sense, we have to notice that each vertex is in its own vector space, so the only possible change of basis is at a vertex. How would we change the basis at a vertex? Given a matrix M that takes vectors in the old basis to vectors in the new basis (which need not be of the same length), we would be inclined to multiply M into each of the edge matrices adjacent to the vertex, since this at least gets all the dimensions right. But how can we decide if this was the right thing to do? What would make it right or wrong? To answer this, let us introduce vertex tensors.

At a vertex of degree d, we can represent the number of d-way paths to each possible combination of d tuples (one from each of the d neighbor vertices) in a big d-dimensional matrix, or *tensor*. We will call this the *vertex tensor* for the vertex. Given a matrix network, the vertex tensor for a vertex v can be computed from the matrices on the edges that touch v. Specifically, for each combination of d tuples at the d neighbors, we can collect, from each of the d matrices, the length-n vector corresponding to the chosen tuple at that neighbor, where n is the number of tuples at v. We then simply multiply these vectors together and add the terms of the result (in effect a multi-way dot product) to get the entry for the vertex tensor. (This description assumed there are no self-loops. If there are, either stick an equality relation in them to break the self-loop, or first process the self-loop as described above for eliminating a leaf cycle.)

We can see that any vertex and its edges (with their matrices) can be replaced by one big vertex tensor (and then applying the obvious semantics when using the tensor in the matrix network), and from the point of view of the rest of the network, nothing has changed.

Now we can say what would make a change-of-basis at some vertex a right or wrong

change: If the vertex tensor is changed, then it was a bad change. If the vertex tensor is unchanged, then the change of basis was a fine thing to do.

So how does our simple idea, of multiplying each matrix by the basis-change matrix M, fare? Well, the original vectors that were being collected from the neighbors (in the course of calculating an entry of the vertex tensor) are now multiplied by M before we take their multi-way dot product.

$$\sum_{i=1}^{n'} \prod_{j=1}^{d} \sum_{k=1}^{n} (V_j)_k M_{k,i} \stackrel{?}{=} \sum_{i=1}^{n} \prod_{j=1}^{d} (V_j)_i \ .$$

The first sum and product on each side compute the multi-way dot product. V_j is the vector collected from the j^{th} neighbor. The inner sum on the left handles the multiplication by M. We would like to know what conditions M must satisfy so that this equality holds for any V. Clearly if M is an identity matrix, or even a permutation matrix, then the equality holds. Does this extend at least to orthonormal matrices (matrices which simply convert between differently oriented coordinate systems)? Our lack of familiarity with the multi-way dot product prevents an immediate answer. Let us examine it more closely.

Define the *down product* $\downarrow$ of a matrix to be the sum of the products of the entries in each column. So for example:

$$\downarrow \begin{pmatrix} 1 & 3 & 2 \\ 4 & 4 & 4 \\ 2 & 1 & 0 \end{pmatrix} = 1 \cdot 4 \cdot 2 + 3 \cdot 4 \cdot 1 + 2 \cdot 4 \cdot 0 = 8 + 12 + 0 = 20 \ .$$

This is exactly the multi-way dot product discussed above, and now our question is, for what matrices M does the equation $\downarrow(N) = \downarrow(N \cdot M)$ hold for all matrices N (of appropriate dimension)? That is, what matrices can we multiply by without affecting the down product?

The answer can be found by doing simple algebra and expanding out all the terms on both sides of $\downarrow(N) = \downarrow(N \cdot M)$, and then grouping the terms according to the contributions from N. Then, since the equation should be an identity for all N, we can view each side as a polynomial in the elements of N, and the coefficients on both sides must match. It turns out that this gives us the simple requirement that, if there are k rows in N, then

the rows of M must be vectors m_i such that the down product of any k of them (allowing repetition) must be zero, unless a single row was repeated k times, in which case the down product must equal one. This is a generalization of the notion of an orthonormal matrix, which corresponds to the case $k = 2$. Since k is the number of neighbors of the relation being transformed, we see that only vertices of degree 2 can be safely transformed by an orthonormal transformation. For higher degrees we must leave the down-product criterion in that form.

1.10.2 Vertex Matrices

Let us consider the variables on the edges as having the values we may want to transform. A relation between n variables will be viewed as an n-dimensional lookup table, giving the number of solutions afforded by the relation for any given combination of values.

With this point of view, we see that for any edge, if the values on all the neighboring edges at both ends are fixed, then the number of solutions afforded by that edge and its two relations is a simple dot product of the relevant vectors from each relation's lookup table. Thus, any transformation that preserves dot products may be applied to the values of the edge, without having any effect on the overall strength of the connectivity provided by the edge. The transformations that preserve dot products are exactly the orthonormal linear transformations.

Thus any edge may have its variables remapped to another set of symbols by an orthonormal transformation, and the total number of solutions will be unchanged.

Formally, once we fix the values of all neighboring edges at both ends, we have a vector v_1 at one end, and v_2 at the other, where v_1 gives the number of ways that a given value for the edge can yield a solution in the first relation given the other values at that end, and v_2 is the same for the second relation. Thus the total number of solutions is $v_1 \cdot v_2$. For any orthonormal transformation T, we have $(v_1T) \cdot (v_2T) = v_1 \cdot v_2$, which is why transforming each relation's view of the edge by T preserves the number of solutions.

If we expand the dot product into a matrix multiplication of a row vector by a column vector, then assuming v_1 and v_2 are row vectors, we have

$$(v_1T) \cdot (v_2T)^\top = (v_1T) \cdot (T^\top v_2^\top) = v_1 v_2^\top,$$

where the final step is due to the property (sometimes used as the definition) of orthonormal matrices that $T^{-1} = T^\top$. This suggests that instead of restricting ourselves to a uniform transformation, we could in fact use any *pair* of transformations T_1 and T_2 at the two ends, so long as $T_1 T_2^\top = I$. In this case, the transformations need not even be square matrices; they must simply be reversible. The next section discusses this in more detail.

1.10.3 Valiant Holography

In 2004, Valiant [Val04] introduced a notion of *holographic algorithms*, that is, algorithms that count the number of solutions to a planar network of relations by doing linear transformations (that preserve the number of solutions) on the relations so that they become implementable as planar networks of "=1" relations, and the associated edge matrices are diagonal (but not necessarily real—any field in which calculations are feasible is allowed). The reason for this is that one can then use a known polynomial time algorithm for this particular form of problem [Fis61, Kas61, TF61]. This method only applies to planar networks, since the known polynomial time algorithm only applies to planar networks.

Valiant's transformations correspond to replacing $v_1 \cdot v_2^\top$ with $(v_1 T_1) \cdot (v_2 T_2)^\top$, where $T_1 T_2^\top = I$, which clearly preserves the result.

However, this transformation is different at the two ends of the edge. Valiant addresses this by requiring the network to be bipartite, so that one part can replace all vectors v with $v\,T_1$, while the other part replaces all vectors v with $v\,T_2$.

Given a non-bipartite graph, we can prepare it for this approach by simply adding a vertex in the middle of every edge, so that the overall structure remains the same, but there are twice as many edges, and the graph is bipartite. Then, in this new graph, we can transform the original relations according to $v \rightsquigarrow v\,T_1$, and transform the new edge (equality) relations according to $v \rightsquigarrow v\,T_2$, where $T_1 T_2^\top = I$. (We can also use an *inequality* relation on the new edges; this can be useful for representing problems where we want to count ways of assigning an orientation, rather than a color, to every edge.)

The transformations associated with each new edge vertex can be merged into the transformation on one of the two sides: T_1 and $T_2^\top$ cancel each other, followed by the identity matrix of the new edge vertex, and the T_2 of the other side. So any time the method of placing a new vertex on every edge works, it would also have worked to use an asymmetric

matrix pair on each edge. Here, "works" means that transformations can be found which allow the new relations to be implementable as planar networks of "=1" relations.

We see that the *generators* (relations whose vectors are each transformed by T_1 in every direction) and *recognizers* (relations whose vectors are each transformed by T_2 in every direction) of Valiant's method can in general be replaced with *transducers* (relations where some vectors are transformed by T_1 and the others are transformed by T_2), without any need for the graph to be bipartite, and without any need for the flow induced by the directionality of "inputs" vs. "outputs" to be cycle free.

One limitation of Valiant's method appears to be that most relations cannot be transformed, by any linear transformation, into relations implementable as planar networks of $=1$ relations.

The main useful category which *can* be so transformed is the set of self-dual relations on three or fewer legs. These can all be converted into the proper form with a Walsh transform. Thus all planar networks of self-dual relations on three or fewer legs can have their solutions counted in polynomial time. Most of Valiant's examples fall into this category.

1.10.4 Meanings of Non-Positive Weights

Non-positive weights come up in many contexts. If they come about as the result of a linear transformation, there may not be much of an answer to the question of what the new weights mean. But many formalisms use weighted combinations of values rather than single values, so we will briefly remind ourselves of a couple of these ways.

In a standard network of relations, we can count the total number of solutions by taking the sum, over all possible assignments of values to the edges, of the product of every relation's acceptance (0 or 1, or more for counted values) of those values. These values are typically found from a lookup table for the 0/1 case, and there is not a clear meaning to a weighted combination of values. Changes of basis are occasionally helpful as a computational optimization.

In belief propagation, the full probability space is the sum, over all possible assignments of values to the edges, of the product of every node's conditional probability contribution for those values. Then a weighted combination of values corresponds to uncertainty. Changes of basis are practically unheard of.

In quantum mechanics, the full probability space is the sum, over all possible assignments of values (base states) to variables, of the product of each variable's weight (squared amplitude) to be in that state. Entangled variables have joint contributions instead of individual contributions. A weighted combination of values corresponds to superposition, which can correspond to measurement uncertainty, and changes of basis are commonly performed.

Although there are striking similarities between these topics in the mathematics, the interpretation is quite different in each case.

1.11 Gaseous Relations

One interesting aspect of exclusion networks is that the messages sent along the edges have exactly the same form as the relations at the vertices. In this brief section we look at how this equivalence can be made more explicit.

When two vertices communicate about the value of a variable (or of a combination of variables, for a multivariate edge), they are sharing information about their projections onto the edge's variable(s). To send information, a relation sends its projection. Upon receiving a projection from a neighbor, a relation eliminates tuples as necessary so that its projection will match the incoming projection.

Both input and output of information, therefore, depend not upon who or where the neighbor is, but simply upon what subset of variables is getting its joint possibilities communicated. This means that instead of having the structured edges of a graph, we can imagine the relations to be simply floating around "in solution." When two relations bump into each other, they can exchange information about the intersection of their sets of variables before continuing on their way. Or we can simplify the process even further, separating the information generation process from the information absorption process: Relations can spontaneously generate projections of themselves onto a random subset of variables (thus one relation splits into two), and when two relations meet, if one of the relations is on a subset of the variables of the other relation, then the two can merge to become the masking of one by the other.

Note that there is no distinction in this model between a relation and a message—the messages are relations like any other, capable of sending and receiving information. Although such a "relation gas" does not seem practical to implement, this point of view does

underscore the fact that messages are of exactly the same form as the relations themselves.

1.12 Counted Values

In figure 1.2, the equivalence, between the network shown on the left and the single relation on the right, does not extend to the case where we wish to keep track of *how many* solutions there are. For some combinations of variable values, such as $\langle a, b, c\rangle = \langle 0, 1, 0\rangle$, there are two solutions to the network on the left, while this triple is clearly just a single solution to the relation on the right.

If we would like to represent the network on the left as a single relation while still keeping track of the total number of solutions to the original network, we will need to use a relation that keeps track of a *multiplicity* for each tuple. We will call such a relation a *multi-relation*. This generalizes the notion of a tuple simply being acceptable (multiplicity 1) or unacceptable (multiplicity 0).

This section will explore possible ways to extend the exclusion process to apply to multi-relations. Although each of the possibilities we explore has its advantages and disadvantages, the main purpose in this exploration is simply to get a feel for the range of possibilities.

We will speak as if all multiplicities are positive integers, but what we say can immediately be extended to multiplicities chosen from any abelian semigroup.

If the relations are being constructed on-line, recording those combinations of values that have been observed so far (in the real world, by some external observational system), then multi-relations provide a simple way to store the information: For each observed combination of values, one simply increments the corresponding entry in each multi-relation. We will call this the "histogram method," and each entry is the "tally count" of the corresponding tuple. However, the histogram method is only appropriate in certain contexts, as will become clear.

To understand how to extend the exclusion process to the case of multi-relations, we must examine the meaning of the entries that the exclusion process manipulates. In the boolean case, the meaning of an entry was simply whether the corresponding tuple of values was *plausible* or *implausible*. In the case of multi-relations, we will examine three possible interpretations for the meaning of the numbers, and in each case the meaning will lead to a different updating algorithm.

1.12.1 Min-Sum

One possible interpretation for the values in a multi-relation is that each value represents the maximum number of times that that tuple has occurred, totaled among all observed occasions that are consistent with the current input.

So at the start of the exclusion process, the entries are simply the integers of the multi-relation as produced by the histogram method. Then comes the current input, which may tell us for example that the variable x is known to have the value 1, or in other words, other values for x are excluded. This means that in the multi-relations involving x, we may reduce the tally count for all tuples where x is not 1 to a tally count of zero.

Then, how does information propagate? How should we project a multi-relation onto a subset of its variables? Based on our interpretation of the meanings of the values, the correct way to project is clearly to sum up the tally counts for all the tuples corresponding to the sub-tuple, and assign this sum to be the tally count of the sub-tuple. Then the message relation's values have the same interpretation as the vertex relation's values.

When the message arrives at the neighboring relation, how should the neighbor update itself? How should the incoming tally count of a sub-tuple affect the tally count of a full tuple? Based on the semantics of the values again, we see that the existing tally count of the full tuple must be reduced to the incoming tally count of the sub-tuple, but any further reduction of the existing tally count is unwarranted. If the incoming tally count is larger than the existing tally count, then no change is warranted in the existing tally count.

To summarize, the updated value is the minimum of itself and the incoming value, where the incoming value is the sum of the values of the corresponding tuples in the neighbor. We will refer to this as the "min–sum" algorithm.

We can see that even after the network has converged to a stable state under this algorithm, two neighbors may have different projections onto their common variables. This means that in a sense, the projection has not been fully conveyed from one neighbor to the other. One neighbor has some knowledge about the subset of variables, but the other neighbor is not able to absorb this knowledge into itself in such a way that it then has the same knowledge. In this sense, the min–sum algorithm loses information during propagation.

If there is any noise in the measurements (for example, observations of erratic situations recorded in the histogram method), the summation increases the noise level of individual

values, while the min operation does not decrease it, and the result is that information cannot propagate very far before being swamped by noise (that is, the noise increases the communicated values so that they have no effect on the neighbor).

1.12.2 Min-Max

Another possible interpretation for the values in a multi-relation is that each value represents the maximum number of times (or at least a bound on the number of times) that that tuple has occurred, where the maximum is over all possible assignments of values to all variables, of the number of observed occasions matching all the variable values. That is, if we consider the input to be a partial specification of some unknown total assignment X of values to variables, then the entry for any given tuple represents the maximum number of times that X may have occurred, assuming that X sets the variables of the relation to the values in the given tuple.

In this case, at the start of the exclusion process, the multi-relations do not possess this kind of information, but the existing entries are certainly a bound on this number.

To send a message, a relation projects itself onto a subset of its variables by taking the maximum value of all relevant tuples as the value of the sub-tuple. Then when such a message is received, each entry reduces itself to the incoming value if the incoming value is smaller. We will refer to this as the "min–max" algorithm.

This process may reduce the values significantly even before we give the system any input. When we do give the system some input, we can again do it by zeroing the tally counts for tuples of variable values differing from the input.

After the process converges, the network will have the property that neighbors always have the same projection onto common variables, so information is not being lost as it was in the min–sum scheme.

However, the numbers being manipulated by the network are bounds which even at the beginning are not known to be tight (and probably aren't), so the utility of the final state of the network seems limited in practice.

Another issue is that with this interpretation of the values, the quantity being bounded depends on distributions of values of variables that are far away in the network from the value under consideration. For example, if a new boolean variable is introduced in a remote

part of the network, then the local quantities could conceivably all be divided in half as a result, even if they are uncorrelated with the new variable. However, the bounds calculated by this process may well be unaffected by the remote change. So this interpretation has a global sensitivity to structure, which is not necessarily reflected in the values it computes.

1.12.3 Product-Sum

Another possible interpretation for the values in a multi-relation is that each value represents a local number of modes of existence compatible with that tuple.

The idea is that we are interested in counting the total number of solutions of the network, and each multi-relation contributes some number of solutions for any given tuple. In this interpretation it is perfectly appropriate to replace a subnetwork with a single multi-relation whose entries correspond to the number of solutions previously existing in the subnetwork for any given tuple of connecting values.

The total number of solutions of the network can be expressed as:

$$\sum_{\substack{\text{variable} \\ \text{assignments } X}} \prod_{\text{relations } R} R_{|X} \quad , \tag{1.1}$$

where X maps every variable to a value, and $R_{|X}$ is the entry in relation R for the tuple indicated by X.

In this interpretation, it is not appropriate for the multi-relation to have its values arising as tally counts of observed situations, and indeed, even if one might like to think of the number of matching observed situations as the number of solutions, it is not clear that this representation can do it, since there is not necessarily any local place in the network that can be updated so as to increment the total number of solutions by one. Despite these difficulties with the histogram method, this interpretation has other applications (including counting solutions) which will be discussed in section 1.10.4.

How might an exclusion process proceed in this context? The purpose of the exclusion process would be to reduce the total number of solutions to just those solutions which are consistent with the given input. But this can be done immediately, simply by reducing the values of tuples proscribed by the input to zero. So no exclusion process is necessary for

this purpose.

Another purpose for the exclusion process might be for each multi-relation to try to approximate the total number of solutions consistent with each of its tuples. In other words, if equation 1.1 above gives the quantity of interest, perhaps the exclusion process could calculate it for us.

The exclusion process unfortunately cannot calculate it in general, but a modification of the exclusion process can calculate it for networks that are trees (contain no cycles). Projections are formed by taking the value for a sub-tuple to be the sum of the values of its super-tuples. A message is then the number of solutions that the relation knows about, for each sub-tuple in the message. When such a message is received, the receiver can then multiply each of its own entries by the corresponding sub-tuple entry. This corresponds to a "product–sum" algorithm, but we will have to modify it for it to be useful. The main problem is that this method assumes that the incoming values represent numbers of solutions that the receiving relation does not yet know about—otherwise, the receiving relation will overcount.

We can avoid this problem as follows: Assuming the network is a tree, we can limit the number of messages on each edge to just one in each direction. A message is sent to a neighbor once a message has been received from every other neighbor. (Thus leaf nodes send messages spontaneously, getting the process started.) The message is still the projection, but we must be sure that if a message has already been received from the target neighbor, then the information of that message should *not* be included in the message that is sent. That is, the sent message consists of the product of the multi-relation's entries with the entries of all the received messages *except* for the message received from the target neighbor. This algorithm requires more recordkeeping, as in general it is not possible to store all the necessary information just in the entries of the multi-relation—the contents of each received message must be retained, at least until a message has been sent back.

The reader has probably noticed that this algorithm is then equivalent to the standard belief propagation algorithm [Pea88], which has also been formulated more abstractly as a "generalized distributive law" [AM99].

1.12.4 f–g

The exclusion process as originally presented is clearly an "AND–OR" algorithm, and it is equivalent to any of the three algorithms described above if we map positive values to "plausible," and zero to "implausible."

If we examine the theorems regarding quick convergence and uniqueness of result, we see that they are a consequence of the absorptive law, $f(x, g(x, y)) = x$, which guarantees a form of lattice structure on the elements under f and g. If $f(x, g(x, y)) = x$, then an "f–g" algorithm will converge to a unique result, and if the variables are discrete, it will converge quickly. (If the variables are not discrete, then any rate of convergence is possible.) This absorptive law condition can be understood intuitively as saying that if a message on an edge is bounced back to its sender, who processes it as a received message, then this should not have any affect on the sender's values. Even more intuitively, one can describe this as "hearing your own information should not change your mind."

While min–sum (for non-negative numbers) and min–max satisfy the absorptive law, product–sum does not, and this is why we did not find any use for the standard exclusion process in this context, and why belief propagation algorithms are not guaranteed to converge to a unique result, or even to converge at all, when used on an arbitrary graph.

1.13 Continuous Values

Mathematically it is easy to extend the discussion of previous sections to the case where a variable has a continuous range of possible values.

For example, a relation might give a joint conditional probability distribution over a continuous space of values.

However, the naive extension of the histogram method to the continuous case proves near useless, since an exact value of a continuous random variable will never be seen twice. Instead, regression methods need to be used to infer a continuous distribution from a finite set of samples. Unfortunately, regression techniques lead us away from the exact, provable world and into the heuristic, experimentally testable world, so we will not discuss them here.

Let us look at a couple of simple examples of networks of continuous relations to see

how such things might behave.

It has been mentioned that nothing can be guaranteed about the convergence rate of a network using continuous values. To see why this is, consider a simple network on two variables, x and y, both capable of having any real value from 0 to 1. We can easily construct a network whose only solution is $x = y = 0$, but which converges to this result at any desired rate. Suppose $1 > c_1 > c_2 > \ldots > 0$ are the desired steps of convergence. We use a network with just two relations, a relation $x < y$, and a relation $y < f(x)$, where f is a monotonically increasing function satisfying $f(0) = 0$ and $f(1) = c_1, f(c_1) = c_2, f(c_2) = c_3$, and so on. Given the sequence $c_1, c_2, \ldots$, it is straightforward to construct such a function f, for example by interpolating linearly between the points (c_i, c_{i+1}).

Now when the exclusion process is used on this network, it is easy to see that at the i^{th} step when y has values excluded, the remaining values are $0 \leq y < c_i$ (and the same is true for x). Thus x and y converge at exactly the arbitrary desired rate. This example could also have been constructed using a single relation and an edge from that relation to itself.

Let us consider another simple network: We can construct a simple one-relation, one-variable network whose solution, which it approaches in the limit, is the Cantor set. Let the relation consist of all pairs of the form $\langle x, 3x \rangle$ or $\langle x, 3x - 2 \rangle$, for all $x \in [0, 1]$. Then, with each step of the exclusion process, the set of allowed values gets one step closer to the Cantor set.

If a network of relations is built with piecewise continuous inequality relations, so that at every step the set of allowed values for a variable is the union of a finite set of intervals, then there is no need to use an uncountable infinity of values for the variables. Using a straightforward application of Dedekind cuts, all of the information will be present in a network whose variables take on only rational values (or even reals whose binary expansion is finite), except perhaps information about the endpoints of the intervals. (Depending on the inequalities, initial conditions, and relations, the endpoints may possibly also be known to be excluded or not excluded.)

We see that questions about networks of relations on real-valued variables can quickly lead to many familiar questions from real analysis. Interesting though it is, we will not explore this direction here.

1.14 Comparison with Belief Propagation Networks

Belief propagation [Pea88] is a message passing algorithm designed for calculating probabilities of events based on tables of conditional probabilities. It was originally designed to work for trees, with one message being passed in each direction on each edge, as described in section 1.12.

Later it was found [MWJ99] that essentially the same algorithm often gives reasonable approximations (though not exact results, and not always reasonable results) on graphs with cycles. The algorithm has to be modified to send repeated messages on each edge rather than just a single message, and the numbers eventually converge, not to the correct result, but to a reasonable approximation of the correct result, if all goes well. This modified algorithm is often called "loopy belief propagation." The performance of this algorithm, particularly in the case of turbo code decoding [MMC98], was quite impressive compared to previous methods.

This finding sparked widespread curiousity, leading to the realization that a similar model (the Bethe free energy) was developed in statistical physics, and it was generalized many decades ago to give better approximations (Kikuchi approximations). With some effort, this generalization was transferred back from the lattice graphs used in statistical physics to the arbitrary graphs used for belief propagation [YFW00], and it was found that they were an improvement in the belief propagation domain as well, although the method was still not perfectly reliable or accurate. This improved algorithm is referred to as "generalized belief propagation."

Sections 1.12 and 1.10 have already discussed many of the similarities and differences between exclusion networks, belief propagation, and loopy belief propagation.

Generalized belief propagation corresponds, in the exclusion network point of view, roughly to unzipping zippers (which still makes sense in the multi-relation case) across the diameter of the regions being used in the generalized belief propagation algorithm. Naturally, this improves performance, at a cost of increasing the size of the network, since relationships within regions become "built in" to the network, rather than having to be approximated.

1.15 Comparison with Constraint Satisfaction Networks

Constraint satisfaction problems, often called CSPs, are very similar to networks of relations. There is a vast literature about CSPs (including recent books such as [CKS01, Dec03, Nea05]), and we will only say a few words here—we do not intend to attempt an in-depth comparison.

A CSP is a list of clauses to be satisfied, and the problem is to find values for the variables that satisfy all, or as many as possible, of the clauses.

The form in which all the clauses must be satisfied is pretty much identical to networks of relations with reusable variables, except that there is no edge structure implicit in the CSP. If one applies a simple edge structure, like having one edge for each variable, connected to every vertex which references that variable, then one gets a network of relations on which the exclusion process may be used.

The exclusion process corresponds quite nicely to standard methods for trying to find solutions to CSPs. Typically the exclusion process is combined with a search strategy in which values are tried for variables to see if they work. Various heuristics exist for how to do a backtracking search in a reasonably optimal way, and quite often the same heuristics are applicable to networks of relations as to CSPs.

One simple example of a process that is very similar to the exclusion process is *arc consistency*. Arc consistency gets its name from situations where the relations are binary (they relate two variables), and the variables are reusable. In this case, the graph can be drawn with vertices for the variables and edges (arcs) for the relations, and arc consistency consists of reducing the possibilities for the variables so that all the arcs are consistent, meaning that every remaining variable value is compatible (according to the relation on the arc) with some remaining value of the variable at the other end of the arc.

If we take the dual of a network graph without reusable variables, we get a graph of the form that arc consistency algorithms are designed for.

Arc consistency is sometimes generalized to "K-consistency" essentially by creating a relation for every possible group of K variables. Each relation is the maximal relation whose projections onto two variables are contained in any binary relations present on those two variables. Generalizing in the direction of K-consistency is generally an expensive approach, since it increases the number and size of the relations that need to be manipulated.

1.16 Comparison with Boolean Circuits

Boolean circuits are a well-studied and widely familiar model [Weg87], in which *gates*, such as AND-gates, OR-gates, and NOT-gates, are connected together by wires into a feed-forward circuit. This is one of the standard levels of abstraction currently used in the design of computers.

Any boolean circuit can be converted quite directly into a network of relations. Each gate can be converted directly into a relation on all the input and output wires of the gate. The relation accepts those tuples of values in which the "output" values are the appropriate function of the "input" values, although of course the relation does not make any fundamental distinction between input and output wires. When we convert a gate into a relation in this way, we call the relation the *characteristic relation* of the gate.

Unlike gates, these gate-like relations can be connected in non-traditional ways (e.g., output-to-output, input-to-input, two outputs to one input, and so on) without any inconsistency.

The network of relations model has a stronger representational ability, and at least as powerful computational ability as traditional circuits of gates. The stronger representational ability comes from being able for example to build a circuit and then force the *output* of the circuit to be 1. If the "inputs" to the relation-ized circuit are the dangling edges (for the purposes of considering it as an implementation), then such a circuit implements the relation that accepts those tuples of values that cause the circuit to evaluate to 1.

The idea of implementing a circuit with a network of relations is the key to showing some of the results in chapter 2.

The reverse direction is also possible, that is, it is easy to build a circuit to implement the exclusion process for a given network of relations on discrete variables. However, the circuit will have lots of cycles and feedback, rather than being a traditional feed-forward circuit. Nonetheless, it will be combinational (in the sense of [Rie03]), even if an external input is supplied for every variable value in the network, allowing each to be independently excluded.

It is widely recognized that lower bounds are hard to prove for circuits [RR94]. Any lower bound on the size of a relational network would translate directly into a lower bound for circuit size (although the reverse direction does not follow). Thus although such lower

bounds may require the development of new methods, they would be very valuable if they could be found.

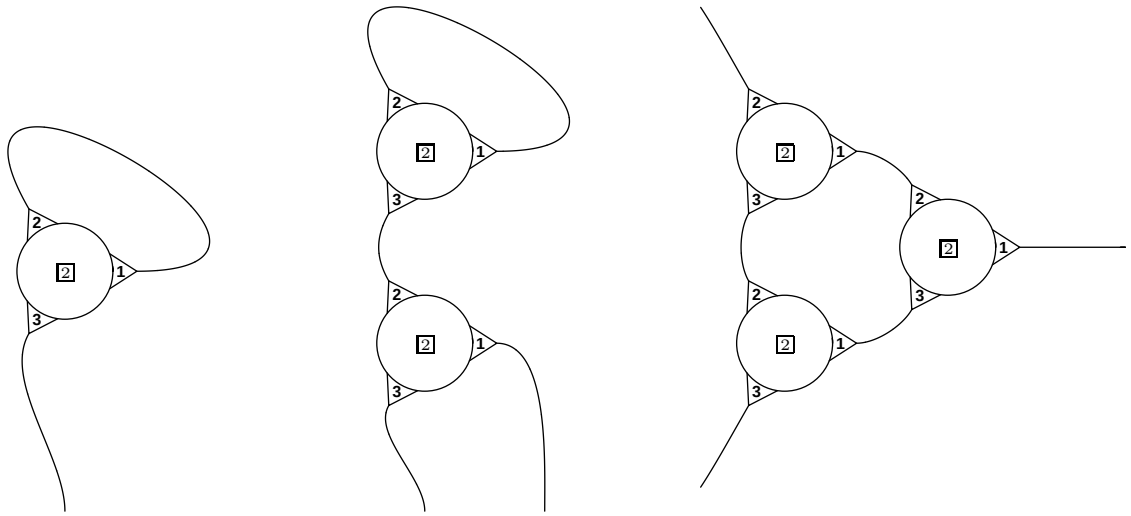
Chapter 2

Relations Emulating Relations

Suppose you are given a box full of little relations. You look inside the box, and you see that all the relations are the same: They each have three wires sticking out, and the relation is that any two of the three wires may have the value 1, while the other has the value 0. That is, the three wires are restricted to the values 0 and 1, and the sum of the three wires must be 2. In the notation of section 1.2, each relation is a "$\boxed{2}$."

You take a handful of relations out of the box and you see that the ends of the wires have little connectors, so any two wire ends can be plugged into each other. For example, if you take one of the relations, and connect two of its three wires together, then the connection forces those two wires to have the same value, so they must be the two 1s, and the third wire takes the value 0, available for use. If you then connect this "0" wire to a wire of another relation, then the two remaining wires on the new relation will be forced to be the "1"s. So you are able to make as many constants as you desire, and this makes you feel good.

Next you try connecting three of them together in a ring as in figure 2.1(c), and you see that you have implemented XOR, the parity relation on three variables. Encouraged, you try an easier one: implementing OR on three variables. It turns out not to be so easy, so first you try for just OR on two variables. If you just consider two of a "$\boxed{2}$" relation's three wires, the relation already enforces OR on two wires, if the third is free to take either value. So you just need to feed the third wire into a circuit that ignores its value. But try as you might, you are unable to make a circuit that takes just a single value and ignores it. You eventually decide to try to prove that it is impossible to do this. This turns out to be easy: In any circuit, the "1" wires in a solution must form chains that can only end at a dangling wire. So each chain of 1s either forms a closed loop or leads to two dangling wires, one at



(a) Forcing a 0 *(b)* Forcing a 1 *(c)* Implementing XOR

Figure 2.1: Small constructions using the "exactly two" relation.

each end. So *any* circuit with only a single dangling wire cannot have that wire being a "1," so in other words it forces that wire to be a "0." Wait, weren't we able to produce 1s as well? Looking again, we see that we made a *pair* of 1s. Now we understand why.

The goal of this chapter is to help us be able to answer questions like this about what can be implemented with what.

We will examine all the relations on three boolean variables, to see which of them can implement which others. Why on three variables? If we just have relations on two variables, and there is no fan-out, then every network can be analyzed simply by multiplying matrices as in section 1.10, and no complexities can arise. However, once we have relations on three variables, the network structure can branch out, and in fact we will show in section 2.4.1 that with networks of relations on three variables, we can implement any relation whatsoever, on any number of variables. Similarly, we use variables that are boolean (two-valued) because that is the smallest kind of variable that leads to arbitrarily complicated behavior.

A relation that can be implemented by a disconnected network (built out of any relations whatsoever), with dangling edges not all on one component, is called a *composite* relation. A relation that is not a composite relation is called a *prime* relation. It is not hard to show that every relation has a unique prime factorization (with the happy relation on no variables being a unit, and the unhappy relation being a zero).

Given a restricted set of building-block relations, sometimes a composite relation can be

implemented even when its prime factors cannot. For example, if the building block relation is the happy relation on two variables, then we cannot use it to build a happy relation on just one variable (because there will always be another dangling edge happily hanging out somewhere). However, if the prime factors can be implemented, then the composite relation can be as well, as a consequence of disconnected networks being permitted.

Even if we limit ourselves to boolean values, there is a countable infinity of possible relations, and an uncountable infinity of possible sets of relations, all of which may be placed into a lattice, so that each set is above those sets which it can implement, and below those sets which can implement it. Each position in the lattice is an equivalence class of sets of relations that can implement each other. At the bottom of the lattice is the happy relation on no variables, implementable by any set of relations simply by forming an empty network of no relations. Also in its equivalence class is the empty set, which is trivially implementable by anything. Clearly nothing else is in this bottom equivalence class. At the top is the equivalence class of all universal sets of relations, which are capable of implementing any arbitrary relation (on boolean variables). Of course, this equivalence class can only be at the top if it is not empty, that is, if there do exist such universal sets of relations. Fortunately, there do. One example of a universal set is the set consisting of just the single relation $\{\langle 1,0,0,0,0\rangle, \langle 0,1,1,0,0\rangle, \langle 0,0,0,1,1\rangle\}$. (We leave it as a nice homework problem to prove that this is universal.) Clearly any superset of a universal set is also universal.

How big is the lattice? Are there a finite, countably infinite, or uncountably infinite number of equivalence classes? The answer turns out to be that it is a big lattice; there are uncountably many equivalence classes. This can be seen by considering just the OR relations on any non-negative number of variables. No matter which ones you have at your disposal, it is not possible to implement any of the others. (Again a good homework exercise.) This means that each subset of this countably infinite set of relations occupies a distinct position in the lattice, so the lattice must have an uncountable infinity of distinct positions. This contrasts with the lattice of functions, which is only countably infinite for boolean functions, although it becomes uncountably infinite for 3-valued functions [YM59].

The work in this chapter can indeed be seen as similar to the project that Post [Pos41] carried out for boolean functions, except that here we carry it out for networks of relations. Of course, since the subject matter and conclusions reached are rather different, the com-

parison is limited. However, the flavor is quite similar, as both projects work to elucidate the lattice of implementability in their respective domains. Indeed, a classic result in universal algebra [Gei68, BKKR69] is that a portion of Post's lattice (the part above the class Post calls R_1) and a portion of our lattice (the part above fan-out, including relations on more variables, and sets of more than one relation, not shown in figure 2.3), are identical to, but upside down from, each other. Whereas this is most of Post's lattice for functions, it is only a countable subportion of our uncountably large lattice for relations.

The part of figure 2.3 that corresponds (in the sense of [Gei68], [BKKR69], and section 1.7 of [Pip97]) to part of Post's lattice of functions (shown on p. 101 of [Pos41]) is the middle peak and three locations below it: $\{26, 44, 46\}$ corresponds to Post's class D_2 (and to the middle peak of figure 2.2), $\{24\}$ corresponds to Post's class D_3 (and to the fan-out node of figure 2.2), $\{131, 137, 139\}$ corresponds to Post's class A_1, and $\{129\}$ corresponds to Post's class C_1.

Figures 2.2 and 2.3 show subsets of the full lattice. These subsets are themselves not lattices, since the meet and join operations are not well defined within these subsets. The figures show the Hasse diagram for these subsets, which include edges that correspond to a sequence of edges in the Hasse diagram for the full lattice. Mathematically speaking, these subsets are simply posets (partially-ordered sets). We will refer to them as *hierarchies*.

Let us remind ourselves of the rules for constructing an implementation. These rules follow from the definitions in section 1.6, but we list them here, in a more rule-like form, for convenient reference.

- Self-loops (edges from a vertex to itself) are allowed.
- Multiple edges (between the same two vertices) are allowed.
- Short circuits (a wire whose two ends are both dangling) are not allowed. If you want to do this, you must implement the equality relation on two variables and use it.
- Ignored wires (a dangling edge which is not considered to be a variable of the implemented relation) are not allowed. If you want to ignore a wire, you must implement the happy relation on one variable (the "don't care" relation).
- Disconnected networks are allowed. For a scenario in which networks are required to be connected, our investigation corresponds to the case when composite relations are

available whenever their factors are available.

- Non-planar networks are allowed. If you want to worry about planarity, then you would need to include (in the definition of relation) information about the order in which the edges occur as you go around the relation, in order to be able to use implementations that use other implementations. For example, being able to implement the "crossover" relation (the relation that $a = c$ and $b = d$, for edges in order (a, b, c, d)) is very valuable (essentially eliminating the constraint of planarity), while being able to implement a similar relation that differs only in variable ordering (such as the relation that $a = b$ and $c = d$) is almost useless. Despite this rule, all of our implementations happen to be planar, and all of our proofs that no implementation exists clearly apply to the planar case as well, so our hierarchies also apply to planarity-conscious applications.

- Constant-valued wires are not available. If you need a constant, you must implement it.

- Fan-out is not available. If you need it, you must implement it.

2.1 Trivalent Boolean Networks With Negation

In this section we will examine networks that are constructed from relations that are all equivalent to each other up to negations of inputs. For example, if one of the relations used is $\textsc{or}(x, y, z)$ then the relations $\textsc{or}(x, y, \overline{z})$, $\textsc{or}(x, \overline{y}, \overline{z})$, and $\textsc{or}(\overline{x}, \overline{y}, \overline{z})$ can also be used, but no others.

This is only slightly different from considering networks consisting of the boolean binary negation relation along with a single boolean ternary relation. For example, if both $\textsc{or}(x, y, z)$ and $\textsc{not}(x, y)$ are available, then we can trivially implement $\textsc{not}(x, y)$, which cannot be implemented just using $\textsc{or}(x, y, z)$ and its negated-variable variants. However, this is the only real difference: $\textsc{not}(x, y)$ and $\textsc{equal}(x, y)$ (formable as a chain of two $\textsc{not}(x, y)$ relations) are the only two prime relations that are implementable one way but not the other (regardless of what the ternary relation is).

The hierarchy representing the results is given in figure 2.2. If we want to think of this hierarchy, as we often will, as showing what individual relations can do when negation is

free (freely available), we simply have to note that just as short-circuit wires are excluded from our implementations, so are short-circuit wires containing negations excluded from implementations where negation is free.

One reason for considering free negation is that in models such as zipper networks, the communication of a variable value from one relation to the next can just as easily swap the two values as keep them unswapped. In these kinds of models, negation really is free: It is just as hard to be sure negation is not present as to be sure it is.

There are $2^{2^3} = 256$ ternary boolean relations, but only 22 of them are distinct when one is allowed to permute and negate legs, so there are only 22 distinct relations that need to be considered for the hierarchy. It turns out that there are only 20 distinct positions in the hierarchy, since there are two positions where instead of a single relation, there are two equivalent relations, each of which can implement the other.

To show that each line in the hierarchy should indeed be there, one can simply give explicit constructions for each case in which one relation can implement another. In every case (including cases that could otherwise be achieved by transitivity), the implementation can be attained using just three instances of the building-block relation, so we will leave the numerous implementations (nearly 100) as easy exercises. (For example, an implementation of XOR by "=1" was already shown, apart from some negations, in figure 2.1.)

All that remains is to show that the missing lines should indeed be absent. The following sections will prove this on a case-by-case basis. It would be nice to have a general purpose theorem to tell us when one relation can or can't implement another, but that question turns out to be undecidable, as we will prove in section 3.1. However, where possible, we will develop theorems and techniques that are applicable to various subsets of the hierarchy.

2.1.1 Relations of Constant Parity

We will start by proving that the relations on the left side of figure 2.2 can only implement other relations as shown in the figure.

Our first theorem is similar in spirit to the discussion of figure 2.1, and its proof will use an inductive "juxtaposition/jumper" technique that is useful in many situations.

Theorem 5 *Any set of relations, each of whose acceptable tuples all have the same parity,*

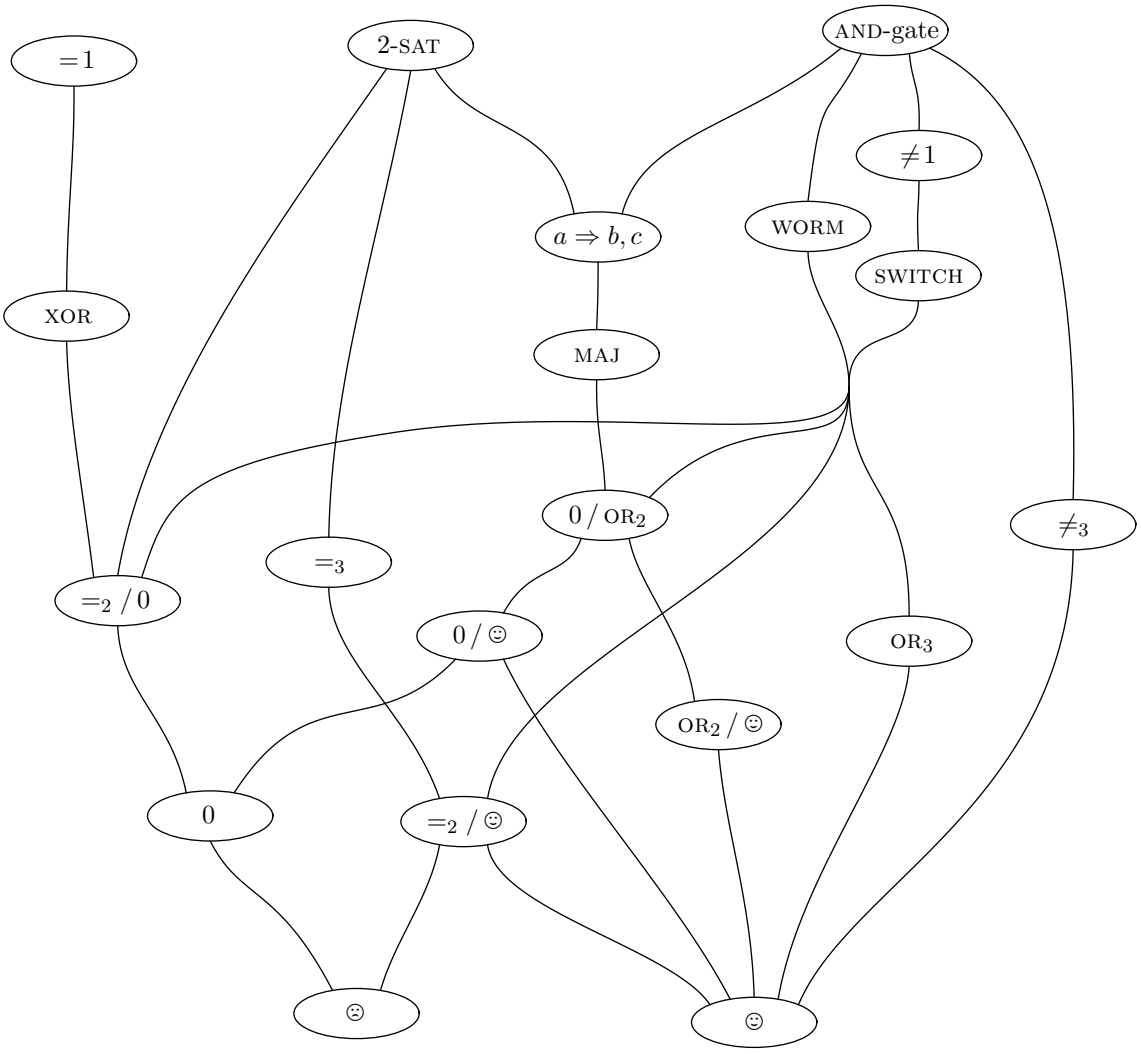


Figure 2.2: Which ternary (three-variable) boolean relations can implement which others when variables can be negated. Where two ovals are connected by a line, the relations represented by the upper oval are able to implement the relations represented by the lower oval. Thus, the relations at the top of this part of the hierarchy are the most powerful, and the ones at the bottom are the weakest. The meaning of the notations in the ovals is as follows: "= 1" accepts triples where $a + b + c = 1$. "XOR" accepts triples where $a + b + c \bmod 2 = 1$. "$=_n$" is the equality relation on n variables. "0" is the *is-zero* relation on a single variable. Where two notations appear in an oval, separated by a / between them, this indicates a relation that contains both of the given relations, operating independently on disjoint subsets of the variables. "☺" is the *don't-care* relation that accepts all tuples. "☹" is the empty relation that is always dissatisfied. "OR_n" is the relation on n variables, that they are not all zero. "$\neq_3$" accepts triples where a, b, and c do not all have the same value. "MAJ" accepts triples where $a + b + c \geq 2$. "$a \Rightarrow b, c$" accepts triples where $a \Rightarrow b$ and $a \Rightarrow c$. "2-SAT" represents two relations, one being $a \leq b \leq c$, and the other being $a = b \leq c$. "WORM" represents the relation that either $a \neq b$ or $a = b = c = 1$. "SWITCH" represents $a \Rightarrow (b = c)$. "$\neq 1$" represents $a + b + c \neq 1$. "AND-gate" represents $a = (b \wedge c)$.

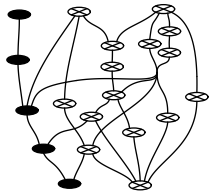
can only implement other such relations.

Proof: We will prove this by induction on the size of the network (defined as the number of relations plus the number of internal edges). Any network can be formed using just two types of growth operations: The first type of growth operation is *juxtaposition*, simply placing two smaller networks side by side, leaving all their dangling edges dangling, not connecting them to each other. The second type of growth operation is to take a smaller network and attach two dangling edges to each other to form an internal edge. This is known as adding a *jumper* wire. For the class of relations which have same-parity tuples, it is easy to see that both of these growth operations will lead to a network that is still within the class. ■

Since negation is a relation whose two tuples both have the same parity, networks containing negations along with other relations of this type, such as a network of "=1" relations with free negations, are subject to the theorem.

Regarding the hierarchy of figure 2.2, this tells us that "=1" (and all the relations below it, those being the ones accepting same-parity tuples) cannot implement "☺" (or any of the relations above it, those being the ones that accept tuples of both parities).

We will represent information such as this with a picture such as the following, in which the relations marked by ×'s cannot be implemented by any of the relations shown in black.



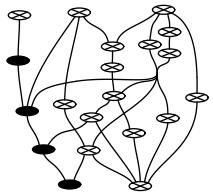
We will also need theorems such as the following, that apply to a very specific part of the hierarchy.

Theorem 6 *Any set of relations, each of which can be factored into relations that accept* all *tuples having the same parity, can only implement other such relations.*

Proof: Again, this property is clearly preserved both for juxtaposition and for a jumper. ■

Our use for this theorem is simply to show that "XOR" (which is prime and accepts all relations of even parity), together with negations (negation is the relation on two variables

accepting all tuples of odd parity), cannot implement "=1" (which is prime but only accepts three of the four relations of odd parity).

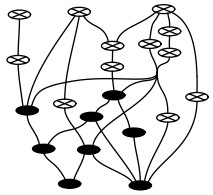


We can examine "XOR" networks more closely and see that an "XOR" network of one component is always equivalent to either the "even-parity" relation or the "odd-parity" relation on all the dangling edges, depending on how many negations are present. To see this consider that negations can be added in pairs without affecting satisfiability of the network. Between any two points on the edges of a network, we may choose a path and flip every value along the path, adding a negation at each point. Any negation or "XOR" relation on the path will still be satisfied (if it was previously), since flipping the two values on the path on the two sides of the negation or "XOR" relation will not affect the parity (recall that negation can also be thought of as the "odd-parity" relation on two variables). Since two adjacent negations cancel each other out, this ability to add negations in pairs also allows us to remove them in pairs, or to move them about from one place in the network to another. It is interesting to note that if a network of "XOR" relations (with values given for any dangling edges) is unsatisfiable, the problem cannot be localized to any part of the network, since adding a negation *anywhere* in the network will make the network satisfiable. The number of solutions to an "XOR" network is also easy to calculate: It is a good homework problem to show that a satisfiable network with e edges, v relations, and no dangling edges, has 2^{e-v+1} solutions, which for a planar network with f faces is 2^f solutions. (If there are no negations, each face can be colored black or white and then edges separating black from white receive a 1.)

Continuing down the left side, next we need to show that "$=_2 / 0$" cannot implement XOR. Recall that the order of a relation is the number of variables it relates (in other words, the number of edges sticking out from it).

Theorem 7 *A set of relations, each of which can be factored into relations of order one or two, can only implement other such relations.*

Proof: This property is clearly preserved both for juxtaposition and for jumpers. ■

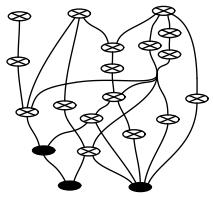


Since "$=_2 / 0$" factors into "$=_2$" and "0," we see that it cannot implement XOR, which is a prime relation of order three.

A similar theorem will help with the next step.

Theorem 8 *A set of relations, each of which can be factored into relations of order one, can only implement other such relations.*

Proof: This property is again clearly preserved both for juxtaposition and for jumpers. ■



This theorem shows why the "0" relation on three variables (which can factor into three "0" relations on one variable each) cannot implement the "$=_2 / 0$" relation, since "$=_2$ is a prime relation of order two.

Note that the unhappy relation "☹" is like the number 0: It can always be factored into any set of relations, so long as at least one of the factors is itself a "☹." It is the only exception to the unique factorization rule.

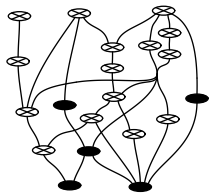
Although it is obvious that the "☹" relation cannot implement the "0" relation, we will give here a more general-purpose theorem of which this is a special case.

Definition 21 *If the set of tuples accepted by a relation is closed under swapping zeros with ones, then we say the relation is* self-dual.

The self-dual relations in figure 2.2 are "☺," "☹," "$=_3$," "$\neq_3$," and "$=_2 /$☺." Negation ("$\neq_2$") is also a self-dual relation. Note that any relation that forces a variable to a constant is not self-dual.

Theorem 9 *A set of self-dual relations can only implement other self-dual relations.*

Proof: Any solution to a network of self-dual relations can have every wire's value flipped from 0 to 1 or vice-versa, and each relation will still be satisfied, since each relation is self-dual. Thus, the dual of any tuple acceptable to the network is also acceptable, so the implemented relation is self-dual as well. ■



This theorem implies that "☹" cannot implement "0."

Thus we are done showing that for all the relations down the left-hand side of figure 2.2 (all the relations under $=1$), each can implement exactly the relations shown in the figure and no others (among relations of order three).

2.1.2 Relations Based on "⇒"

We now move to the relations under the central peak in figure 2.2. This peak is labeled "2-SAT" because networks built from relations at this peak are equivalent to traditional 2-SAT problems: Such networks can be represented as a conjunction of clauses, where each clause is a disjunction of two literals, where each literal is either a variable or a negation of a variable. There are no fan-out limitations in this representation (note that fan-out, "$=_3$," lies under the second peak).

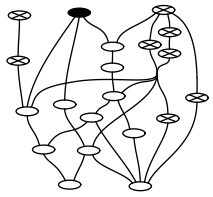
There are two ternary relations that are located at the second peak. One of them, the "$\cdot\leq\cdot\leq\cdot$" relation, specifies that the three variables (say a, b, and c) must satisfy $a \leq b \leq c$. The other specifies that $a = b$ and $b \leq c$; we call it the "$\cdot=\cdot\leq\cdot$" relation. It is not hard to see that each of these two can implement the other, and they can both implement fan-out as well as the ordinary "$\leq$" relation on two variables. Note that the "$\leq$" relation is exactly the same as the "⇒" relation, and negating a variable makes it equivalent to the "OR_2" relation.

We can see that any tuple that is rejected by a network of these relations must be rejected due to some particular pair of variable values. That is, assuming the relation is of order at least two, there must be two positions of the rejected tuple which account for

the rejection in the sense that all tuples matching that tuple in those two positions are also rejected, regardless of the values in the other positions.

Theorem 10 *Relations implementable by "$\cdot\leq\cdot\leq\cdot$" (along with free negation) are exactly those relations having the following property: Any rejected tuple is also rejected in some projection of the relation onto two or fewer variables.*

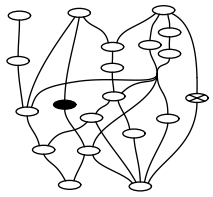
Proof: The forward direction is as discussed above. The reverse direction is also clear, since if all rejections are accounted for by particular variables or pairs of variables, then we can write down a 2-SAT formula representing the relation, and this can be implemented in a network. ■



Theorem 10 explains why "XOR," "OR," and "$\neq_3$" (as well as all the relations above them) are not implementable by either of the "2-SAT" relations.

Theorem 11 *The "$=_3$" relation, with free negation, is unable to implement the "$\neq_3$" relation.*

Proof: In any connected component of a network consisting of "$=_3$" (fan-out) relations and negation relations, there are either zero or two (dual) solutions for that component. The "$\neq_3$" relation must have all dangling edges coming from the same component, since it is a prime relation. But any implementation of it must have at least six solutions, not just two, since it accepts six different triples. ■



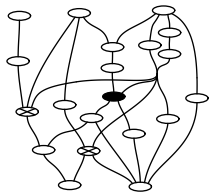
This theorem, together with theorem 9, shows that "$=_3$" can only implement other relations as shown in figure 2.2.

To finish going down from fan-out, the "$=_2$ / ☺" relation can only implement other relations as shown due to theorems 9 and 7, and the "☺" relation cannot implement any other ternary relation due to theorems 9 and 8.

Working our way back up the other relations under 2-SAT, we see that theorem 8 explains why "0 / ☺" can only implement the relations shown.

Theorem 12 *The "0 / OR_2" relation with free negation cannot implement the binary relation $=_2$.*

Proof: The only way to get a prime binary relation is by stringing together a chain of binary relations (since no higher order prime relations are available), but all we have available is "OR_2" and free negation. Recalling that the rules prohibit us from making a short-circuit consisting of free negation alone, we see that the chain will have to include an "OR_2," and thus in any alleged implementation of "$=_2$," starting from a solution to the network, we can flip the values on the wires starting from one end or the other of the chain implementing "$=_2$" so that the flips can stop at an "OR_2" relation in the middle of the chain, demonstrating a solution to the network which is inconsistent with $=_2$. ■

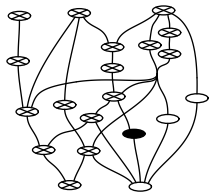


This theorem, combined with theorem 7, explains why the "0 / OR_2" relation can only implement other relations as shown.

Theorem 13 *Every network built from the "OR_2 / ☺" relation with free negation is satisfiable somehow.*

Proof: Factoring the "OR_2 / ☺" relations into "OR_2" relations and unary "☺" relations, we see that the network must consist of loops or chains of relations, with either a unary "☺" relation or a dangling edge at each end of a chain, and with any binary relation in the middle of a chain or loop being either the negation relation or an "OR_2" relation. Furthermore, a loop cannot consist solely of negation relations, according to the rules.

First of all, we note that every loop is satisfiable: Starting at an "OR_2" relation, we can set the wires around the loop alternately to 1 and 0 (starting with 1), and this will satisfy both every negation relation and every "OR_2" relation, including the one we started with. Similarly, every chain is satisfiable (assuming we get to set dangling edges as we like). Therefore, there must be some set of values for the dangling edges which is acceptable to the network. ■

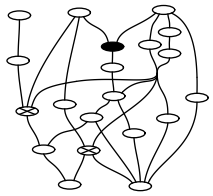


The point of this theorem is to show why the "OR_2 / ☺" relation cannot implement the "☹" relation (or, therefore, anything that can implement the "☹" relation). This theorem, combined with theorem 7, explains why the "OR_2 / ☺" relation can only implement other relations as shown.

We only have a couple of relations to go to finish the 2-SAT region. We only need to show that "$a \Rightarrow b, c$" cannot implement "$=_2$," and that "MAJ" cannot implement "$a \Rightarrow b, c$."

Theorem 14 *The "$a \Rightarrow b, c$" relation with free negation cannot implement "$=_2$."*

Proof: Consider the first of the two dangling edges. After possible negations, it must (by the rules) be attached to an "$a \Rightarrow b, c$" relation as either a, b, or c. For any of these cases, one of the two possible values for this edge cannot be forced by the rest of the network. Thus the network cannot enforce the "$=_2$" relationship. ■

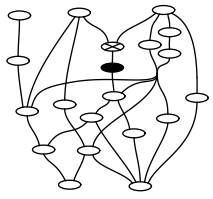


Note that this theorem implies theorem 12, since "$a \Rightarrow b, c$" can implement "$0 / \text{OR}_2$."

The proof of the next theorem will introduce the flipper method, which is often useful when simpler proof techniques are not working.

Theorem 15 *The "MAJ" relation with free negation cannot implement "$a \Rightarrow b, c$."*

Proof: Suppose we have an implementation, using "MAJ" and negation relations, of the "$a \Rightarrow b, c$" relation. Since "$a \Rightarrow b, c$" accepts the triple $\langle 0, 0, 0 \rangle$, there must be a solution to the network corresponding to this triple. Starting from this solution, let us send a flipper into the network on the dangling edge corresponding to a. The flipper will flip edges from 0 to 1 (or from 1 to 0) as it travels through the network. It will do this so as to keep every relation in the network satisfied. In fact, the only place in which the network's state is not satisfactory is at the flipper itself. So the flipper will keep moving until it can disappear somehow. When it gets to negation, it just keeps going, flipping more edges, and the negation will stay satisfied, since both its edges will have been flipped. When it gets to a "MAJ" relation, there are a couple of possibilities: If, after flipping the incoming edge, the "MAJ" relation is still satisfied, then the flipper can disappear and we have arrived at another solution to the network. (However, this solution would contradict the alleged implementation of "$a \Rightarrow b, c$," so this case will not occur.) Otherwise the incoming edge was a 1 getting flipped to a 0, and the other two edges are a 1 and a 0. In this case, the flipper must continue out along the edge that was 0, flipping it to 1. Can the flipper cycle around the network forever? No, because every wire it flips is *forced* to have its new value by the new value for the dangling edge corresponding to a. Furthermore, every time the flipper leaves a "MAJ" relation, all three wires are forced to have the values they have. This means that if the flipper were to return, flipping one of the wires, that would mean that the wire is forced to have two contradictory values, meaning that the network cannot accept the new value for the dangling edge corresponding to a. (But this unaccaptance would contradict the alleged implementation of "$a \Rightarrow b, c$," so this will not occur.) Thus, the flipper must eventually leave at another dangling edge of the network. But this, too, would contradict the alleged implementation of "$a \Rightarrow b, c$," since starting from $\langle 0, 0, 0 \rangle$ and flipping the a edge, we should have to flip both b and c, not just one of them. So the flipper shows us that "$a \Rightarrow b, c$" is not implementable by "MAJ" and negation. ■



We have now seen why each of the relations at or below the first and second peaks can

only implement other relations as shown in figure 2.2.

2.1.3 The Third Peak

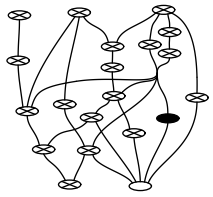
We now move on to the relations at or below the third peak of figure 2.2. We need to show, for each such relation which is not under the first or second peak, that it can only implement other relations as shown in the figure.

This area is where the more elaborate relations lie, and the proofs in this section will be more difficult. To start with the easier proofs, we will work our way up from the bottom.

Theorem 16 *Any network of "OR_3" relations with free negation can only implement relations factorable into "OR_n" for $n \geq 3$ (up to free negation) or "☺."*

Proof: The "OR_3" relation can easily implement the "don't-care" relation ("☺" on one variable), simply by connecting two of its three connections with an edge. Thus we will allow the "don't-care" relation to appear in the network as well, and we will shrink a given network to an equivalent form by replacing parts of it with "don't-care" relations. We will think (and speak) of negations as appearing on an edge rather than distinguishing the two sides of the negation as separate edges. First of all, any internal edge of the network with no negations along it (or an even number of negations along it) may be set to 1 so that both "OR_3" relations at the ends are guaranteed to be satisfied. Thus we may eliminate this edge and replace those two "OR_3" relations with "don't-care" relations attached to each of the other wires previously connected to those relations. Further, any "OR_3" relation with a wire whose other end has a "don't care" can set that wire to 1 so as to guarantee satisfaction of the "OR_3" relation, and again the wire may be removed and the "OR_3" relation may be replaced by "don't care" relations on the other incident edges. In this way, "don't-care" relations eat up an entire component once they appear, leaving nothing but "don't-care" relations on the individual dangling edges. The only way for a component to avoid this is to have one negation on each internal edge. But even then, if there is a cycle in the component, then we can set each edge in the cycle so as to satisfy the following "OR_3" relation, and thus the entire cycle may be eliminated and replaced with "don't-care" relations on the other edges incident to the cycle, which will again cause the component to degenerate into the "☺" relation on all its dangling edges. Thus, to avoid this fate, a component must be a tree

of "OR_3" relations with a negation on each internal edge. This can be seen to implement "OR" on all the dangling edges (assuming they are not negated), of which there are three or more. ■



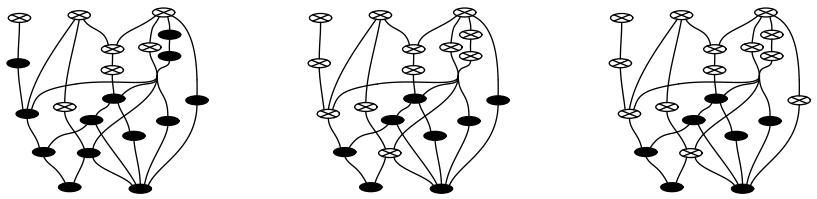
Definition 22 *Define a* k-robust *relation as a relation with the following property: For any tuple that is not acceptable to the relation, all other tuples up to Hamming distance k away* are *acceptable to the relation.*

Note that a k-robust relation is also m-robust for every $m < k$. The only 3-robust relations in figure 2.2 are "OR_3," "OR_2," "0," and "☺." (These are actually k-robust for all k.) ("☹," treated as a relation on 3 variables, is factorable into 3-robust relations.) The only 2-robust relation (apart from the 3-robust relations) in figure 2.2 is "$\neq_3$." The only 1-robust relations (apart from the 2-robust relations) in figure 2.2 are "$=_2$," "XOR," "SWITCH," and "$\neq 1$." Since the negation relation is only 1-robust, it is better to think of negation here not as a separate relation, but as built in to one of the adjacent ternary relations, since negating a variable of a relation does not affect the robustness of that relation.

Theorem 17 *A set of relations factorable into k-robust relations can only implement other such relations.*

Proof: Since all relations are 0-robust, the theorem is trivial for $k = 0$. Assume that $k \geq 1$. Consider a connected network of k-robust relations. We need to show that from any rejected tuple, we can flip from 1 to k inputs (dangling edges), and the network will accept the result. To do this, we will first set the dangling edges to be an arbitrary tuple that is rejected by the network. We then choose a spanning tree of the component (including dangling edges), and arbitrarily set all internal edges not on the spanning tree. Next, given a set of dangling edge values that is not acceptable to the network, we note that for any relation in the network, we can orient the edges of the spanning tree to point away from that

relation, and we can set those edges so that each is responsible for satisfaction of the relation it points to (or consistency with the dangling edge), and thus localize the dissatisfaction to a single relation in the network, of our choosing. Further, this dissatisfaction can be pushed around: To push it from the dissatisfied relation to one of its neighbors, simply flip an edge connecting them. (How do we know that will make the neighbor dissatisfied? Because otherwise the values on the dangling edges would turn out not to be rejected by the network, contrary to assumption.) We will consider pushing it around just on the spanning tree, indeed, just on the subtree of the spanning tree whose leaves are the (up to k) dangling edges that we intend to flip. Note that this subtree has vertices of degree at most k. Note that as we push the dissatisfaction around on this subtree, each relation is only ever dissatisfied by one particular setting of the edges. We can make the subtree extremely dissatisfied by introducing an inconsistency on every internal edge of the subtree so that every relation in the subtree can be simultaneously dissatisfied. We also introduce an inconsistency on each dangling edge of the subtree, flipping those dangling edges as we intended. Now we can arbitrarily pick a relation in the subtree and orient all the edges of the subtree to point towards it. Now, every internal edge can resolve its inconsistency by changing its value at the end with the arrowhead to be consistent with the value at its tail. Since every relation in the subtree has at least one but no more than k arrowheads, the result has every relation satisfied. ■

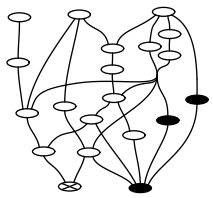


Note that this theorem also provides an alternate proof that "XOR" (which is 1-robust) cannot implement "=1" (which is 0-robust).

Theorem 18 *Any network of relations which are factorable into k-robust relations (of order at least two), for $k \geq 2$, has a satisfying assignment (i.e., does not implement "☹"). Any connected component of the network which has a cycle implements the "☺" relation.*

Proof: For a connected component with a cycle, we can orient the edges of the cycle to go around the cycle, and we can find a spanning forest for the remainder of the network

(without the cycle edges, but including dangling edges) such that each tree of the forest touches the cycle, and orient those edges to point away from the cycle, leaving any other edges unoriented. We can first set the unoriented edges arbitrarily, and then set each oriented edge of the forest, working our way towards the cycle, so as to guarantee the happiness of the relation pointed to by the edge, or for a dangling edge, so as to match the arbitrary input to the network (since we want to show that the component implements the "☺" relation). We then set the edges on the cycle arbitrarily, and if any relation on the cycle is dissatisfied, we can go around the cycle, starting at any dissatisfied relation, flipping edges until we can stop. This process cannot loop more than once around the cycle, due to the 2-robust property. This proves the second claim of the theorem. For the first claim, we now need only consider networks without cycles. Since the relations are of order at least two, there must be at least two dangling edges in any component. Thus, for each component, we may connect two dangling edges to form a cycle, apply arbitrary values to the remaining edges, and we are guaranteed a solution. We then disconnect the two dangling edges to restore the original network, and we have a solution for it. ■



These theorems, combined with theorem 9, show that "$\neq_3$" can only possibly implement what it is shown to implement in figure 2.2.

There are four relations remaining in the hierarchy for which we have yet to prove that they can only implement other relations as shown in figure 2.2. Specifically, we need to show the following:

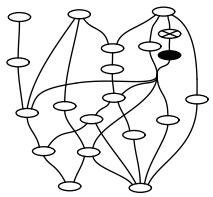
- "$\neq 1$" and "WORM" cannot implement "MAJ" or "$\neq_3$."
- Of "SWITCH," "$\neq 1$," and "WORM," each cannot implement the next (cyclically; three proofs are needed).
- "AND-gate" cannot implement "$=_3$" or "XOR."

These proofs will complete the overall proof that the only possible implementations among the relations are as shown in figure 2.2. (Other cases are covered by the cases above. For

example, "SWITCH" cannot implement "$=1$" because if it could, then "AND-gate" would be able to implement "XOR" via "SWITCH" and "$=1$.") We will spend the rest of this section on these proofs.

Theorem 19 *"SWITCH" cannot implement "$\neq 1$."*

Proof: A "SWITCH" relation can be thought of as a wire with a controller. If the controller is 1, then the wire is connected (must have the same value on both sides of the switch). If the controller is 0, then the wire is disconnected (may have any value on either side of the switch). Consider a network of "SWITCH" relations in which wires may have negations along them. Each wire in such a network is either a path or a cycle. A cycle with an even number of negations may be assigned values so that none of the controller values matter. So an equivalent network can be obtained by deleting the cycle, replacing each "SWITCH" relation in the cycle with a "don't-care" relation on the impinging controller. A cycle with an odd number of negations needs to have at least one controller be a 0, so if there are n "SWITCH" relations in the cycle, the cycle is equivalent to an "OR_n" relation with every wire negated. Each end of a path is either a controller or a dangling edge. (If either end of a path is a "don't-care," then the path may be removed, changing all incident edges to end in "don't-care" relations. Thus, "don't-care" relations eat up everything in sight, converting all dangling edges in the component to "☺" relations. This is inconsistent with the "$\neq 1$" relation.) In a network implementing "$\neq 1$," at least one of the three dangling edges must be on a path with a controller at the other end. Of the two possible values for this dangling edge, one will cause the component to get eaten by "don't-care" relations, meaning that all triples having that value for that edge are acceptable to the network. This is incompatible with implementing "$\neq 1$." ■



Theorem 20 *"WORM" cannot implement "SWITCH," "MAJ," or "$\neq_3$."*

Proof: The discussion in section 2.4.6 explains how a network of "WORM" relations may be viewed in terms of paths, loops, negations, and controllers on negations. At least one of the

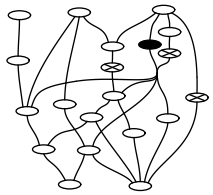
three dangling edges must be on a path with a controller at the far end. Any such dangling edge can only force its variable in one direction—one of the two possible values (0 or 1) cannot be forced, regardless of the rest of the network, since a controller cannot be forced to be 0, and a controlled negation cannot be forced not to negate, so setting the controller to 1 and having the value flip at each negation will lead to a value for the dangling edge that is acceptable if any value is. Thus "$\neq_3$" cannot be implemented, as "$\neq_3$" can force each dangling edge in either direction, based on the other two dangling edges. Similarly, an implementation of "SWITCH" would need to have the two wires which can be forced to be equal to be at two ends of a single path.

Consider a path with a dangling edge at each end. If all controllers are 0, then a perfect correspondence is enforced between the two dangling edges. Grouping controllers so that controllers on odd-numbered negations are in one group while controllers on even-numbered negations are in the other group, if only one group has controllers that are 1, then three out of the four possible pairs of values for the ends are acceptable. If both groups have controllers that are 1, then all four pairs of values are acceptable. In no case are both dangling edges forced, so "MAJ" cannot be implemented using a path with dangling edges at each end. An implementation of "SWITCH" would need its third wire to be able to force all controllers on the dangling-edges path to be 0, where for the other value of the third wire, at least two controllers could be 1.

If we send in a flipper on this third wire with instructions to disappear at the first opportunity, then whenever it needs to continue it can always go straight across negations until it reaches a controller, at which point if it needs to continue it may choose either direction. (If the controller is controlling a negation already traversed by the flipper, then the flipper will not need to continue after reaching this controller.) So we may specify a route of this type for it, and it will never need to leave the route. We may pick an arbitrary route, stopping if we get to the dangling-edges wire. This route will lead to at most a single controller on the dangling-edges wire. Thus it is not possible that changing the value of the third wire could force more than a single controller on the dangling-edges wire to change to 0. So "WORM" cannot implement "SWITCH."

In any network of "WORM" relations, we can find a route for the flipper from any given dangling edge either to another dangling edge or to no dangling edge (if it ends by bumping into itself). This means that for any variable, there is a second variable, such that for

any acceptable tuple of values, if the value of the first variable is changed, then either the new tuple is acceptable, or it can be made acceptable by changing the value of the second variable. This property does not hold for "MAJ," since it is acceptable for the first variable to be 1, the second to be 1, and the third to be 0, but if we flip the first one, we cannot reach an acceptable tuple by flipping the second. ■

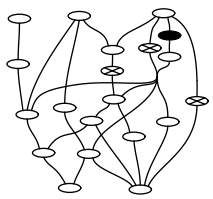


Theorem 21 *"$\neq 1$" cannot implement "*WORM*," "*MAJ*," or "$\neq_3$."*

Proof: A network of "$\neq 1$" relations is easily satisfied. The three target relations we are considering, "WORM," "MAJ," and "$\neq_3$," are prime relations and thus must be implementable (if at all) by a network of just one component. Thus we will consider "$\neq 1$" networks consisting of a single connected component.

Any network of "$\neq 1$" relations is at least as satisfiable as a network of "XOR" relations, in the sense that if a network of "XOR" relations can be satisfied, then the same edge values can be used to satisfy a network that is exactly the same except each "XOR" relation is replaced by a "$\neq 1$" relation, since the tuples accepted by "$\neq 1$" are a superset of the tuples accepted by "XOR."

Theorem 6 (and the discussion following it) tells us that any network of "XOR" relations, consisting of a single component with three dangling edges, always implements another "XOR" relation (up to free negation when available). Thus a single-component "$\neq 1$" network can only implement other relations that are also supersets of "XOR." This condition does not hold for "WORM," "MAJ," or "$\neq_3$," even with free negations. ■



Theorem 22 *A network of "*AND*-gate" relations, with free negation, cannot implement*

fan-out.

Proof: We will prove this by showing that if there is such a network, then it can be reduced to a smaller one. Repeating this would take us down to a network consisting of no gates, giving us the desired contradiction. We will measure size of the network as the number of gates which are not "constant-forcers," where a constant-forcing gate is one which has a self-loop edge between two connections, with a single negation on that edge, so that a constant is forced at the third edge. (The presence or absence of a negation on this third edge determines which constant.)

Suppose we are given a network of "AND-gate" relations that implements the equality relation on at least three variables. We will examine this network in the vicinity of one of the dangling edges, call it d. Say that g is the "AND-gate" relation that d is connected to. We see that g cannot be a constant-forcer. There must be at least one solution to the network where g forces $d = 0$, and at least one where g forces $d = 1$.

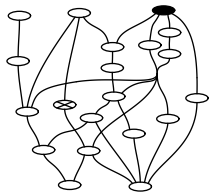
Suppose d is connected to an "input" end of the "AND-gate." Then there cannot be any solution to the network in which the other input is 0, since in this case d could take either value without anything else in the network changing, which is inconsistent with fan-out. Thus we can replace g with a constant-forcing gate that forces the other input to 0, and directly connect d's input to the output wire, thus reducing the size of the implementation. For any solution that existed previously, there is a solution to the new network with the same values for the dangling edges, but there are no solutions to the new network that do not correspond to solutions of the original network, so the new network still implements fan-out.

Now suppose d is connected to the output end of g. We will consider two cases: Either the network has some solution where g's input wires differ, or it does not. If it does not, then in fact both of the inputs to g are already being forced to be perfectly correlated (or anti-correlated) with the other dangling edges, so we can remove g to get a network that implements equality on even more variables than the original network.

If the network does have a solution where g's input wires differ, then we may take the wire with value 1 and force it to always be 1, by replacing its middle with two constant-forcers forcing the ends to always be 1. This may eliminate some solutions to the network, but it will not add any solutions, and there will still be a solution with $d = 0$ and a solution

with $d = 1$, so the new network still implements fan-out. But in this new network, g's function is trivial: since one of its inputs is always 1, it simply forces equality on its other two wires, so we may simply connect those wires together, eliminating g and the constant-forcer feeding into it. Thus we once again have a smaller network that still implements fan-out.

Since in every case we have found that the network could be reduced to a smaller one while still implementing fan-out, we see that there is no smallest network, and therefore no network at all, that implements fan-out. ■



Theorem 23 *A network of "*AND*-gate" relations, with free negation, cannot implement "*XOR*."*

Proof: Assume we have a network implementing "XOR" on dangling edges a, b, and c. Consider a solution S to the network. There must be another solution S' in which a and b are flipped, along with a minimal number of other edges. Let D be the set of edges that have different values in S and S'. Consider what happens if we start with S and send a flipper in a, and the flipper is constrained to stay within D. There are three kinds of vertices the flipper may encounter: "Type 1" vertices have all three wires being 0, "type 2" vertices have two differing "inputs" (and therefore "output" 0), and "type 3" vertices have all three wires being 1. As the flipper travels, some vertices may force a flipper to split into two flippers, but it is clearly never necessary for any flipper to leave D. The flippers are instructed not to split unless necessary.

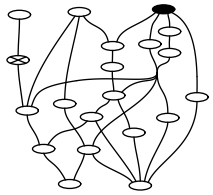
- A flipper arriving at the output of a type 1 vertex will be split into two flippers that continue on both of the inputs, converting it to type 3.
- A flipper arriving at an input of a type 1 vertex can simply stop, converting it to type 2.

- A flipper arriving at the 0 input of a type 2 vertex must continue either on the other input, converting it to type 1, or on the output, converting it to type 3.
- A flipper arriving at the 1 input of a type 2 vertex can simply stop, converting it to type 1.
- A flipper arriving at the output of a type 2 vertex must continue along the 1 input, converting it to type 3.
- A flipper arriving at an input of a type 3 vertex must continue on the output, converting it to type 2.
- A flipper arriving at the output of a type 3 vertex must continue on an input.

These rules must result in all of D being flipped, regardless of whether the flipper starts at a or b. No edge will have a flipper traverse it more than once, since whenever a flipper arrives on an unflipped edge of a previously visited vertex, it may either stop or continue on another unflipped edge, while staying within D. If two flippers meet head-on on an edge, they may both simply stop. If we orient the edges of D according to the direction of travel of the flipper traversing it when we start a flipper at a (and for edges where flippers collided, orient parts of the edge differently), we can see that no vertex will have all three edges oriented towards it, because a flipper can never arrive at the output and just stop. Similarly, no vertex can have all three edges oriented away from it, because flippers do not spontaneously arise. Now, we can start a flipper at b, and we may constrain it and its descendants to travel against the orientations, and they will be able to do so while following the above rules. This means that every vertex with all three edges in D must have split a flipper by the first rule above in one of the two directions, so each three-way junction in D is a type 1 or type 3 vertex, which gets flipped to the other type. For all other vertices along the flipper paths, the unused edge at the vertex must be a 1 input, since otherwise, in one of the two directions, the flipper could simply stop.

Now consider a minimal set of edges D' whose flippage would change solution S into a solution S'' in which a and c are flipped. Clearly D' must satisfy all the same properties that D satisfies. Consider any vertex that has an edge in $D \bigcap \overline{D'}$ *and* an edge in $\overline{D} \bigcap D'$. Since neither D nor D' contain dead ends, the third edge of such a vertex must be in $D \bigcap D'$. Since an unused edge (of either D or D') must be a 1 input, the vertex must be type 3, with

the third edge being the output. This means that in fact we may flip all of $D \bigcup D'$ and the result will be a valid solution to the network, with a, b, and c flipped. This is inconsistent with implementing XOR. ■



This proof can be adapted to give us a somewhat stronger theorem:

Theorem 24 *Any symmetric relation implementable by "*AND*-gate" relations and negations must have a list of acceptable weights (ordered by weight) of one of the following forms, where "$\surd$" indicates an acceptable weight, "$\times$" indicates an unacceptable weight, and an asterisk superscript indicates a sequence of any number of the superscripted symbol.*

- $\times\surd^*\times$
- $\times^*\surd^*$
- $\surd^*\times^*$
- $\surd^* \times \surd^*$

Proof: The proof of theorem 23 actually shows that no symmetric relation on three or more variables can be implemented by the "AND-gate" relation if the list of the symmetric relation's acceptable input weights contains "$\ldots \times \surd \times \ldots$." Extending the proof argument to consider not just D and D', but an arbitrary number of such minimal flipped edge sets (all sharing a single dangling edge), we see that an implementable symmetric relation's list of acceptable weights may not strictly contain "$\ldots \times \surd^+ \times \ldots$," where $\surd^+$ represents one or more $\surd$'s in sequence, although the list may be exactly "$\times\surd^+\times$." We also know that the "AND-gate" relation cannot implement any relation containing "$\ldots \surd \times^{2+} \surd \ldots$," where $\times^{2+}$ represents two or more $\times$'s in sequence, since then some variables could be forced to constants to implement fan-out, contradicting theorem 22. ■

We note that we have only found constructions for the following cases and their reverses:

- $\times\sqrt{}^{*}\times$ (use a tree of "$\neq_3$" with each edge negated)
- $\times^{*}\sqrt{}$ (force each dangling edge to 1)
- $\times^{*}\sqrt{}\sqrt{}$ (use a tree of "MAJ" with each edge negated)
- $\times^{*}$ (force an internal edge to be both 1 and 0)
- $\times\sqrt{}^{*}$ (use a tree of "OR" with each edge negated)
- $\sqrt{}^{*}$ (let each dangling edge be a "don't-care")
- $\sqrt{}\times\sqrt{}^{*}$ (use a tree of "$\neq 1$")

Between theorem 24 and these constructions remain many open problems.

2.2 Trivalent Boolean Networks

In this section we will consider the case where negation is *not* assumed to be available, and we will show that the implementability hierarchy is as shown in figure 2.3

It turns out that our efforts proving the correctness of the hierarchy with free negation will help us greatly with this larger hierarchy. Consider two relations A and B in the hierarchy without negation, where A' and B' are the corresponding elements in the hierarchy with free negation. Suppose A can implement the two-leg negation ("$\neq$") relation. Then A can implement A', and together with B', can implement B. So if A' can implement B' with negation, then A can do likewise to fully implement B. Conversely, if A can implement B without negation, then A' can certainly implement B' with free negation in the same way. We encapsulate the latter observation in the following theorem.

Theorem 25 *If a relation A' cannot implement a relation B' with free negation, then A cannot implement B without negation, where A is the same as A' except for possible negations of variables, and B is similarly similar to B'.*

To make use of the former observation, we would like to be able to decide whether a relation can implement negation based just on the data visible in the hierarchy of ternary boolean relations. The following theorem will help us with this. This is a positive theorem

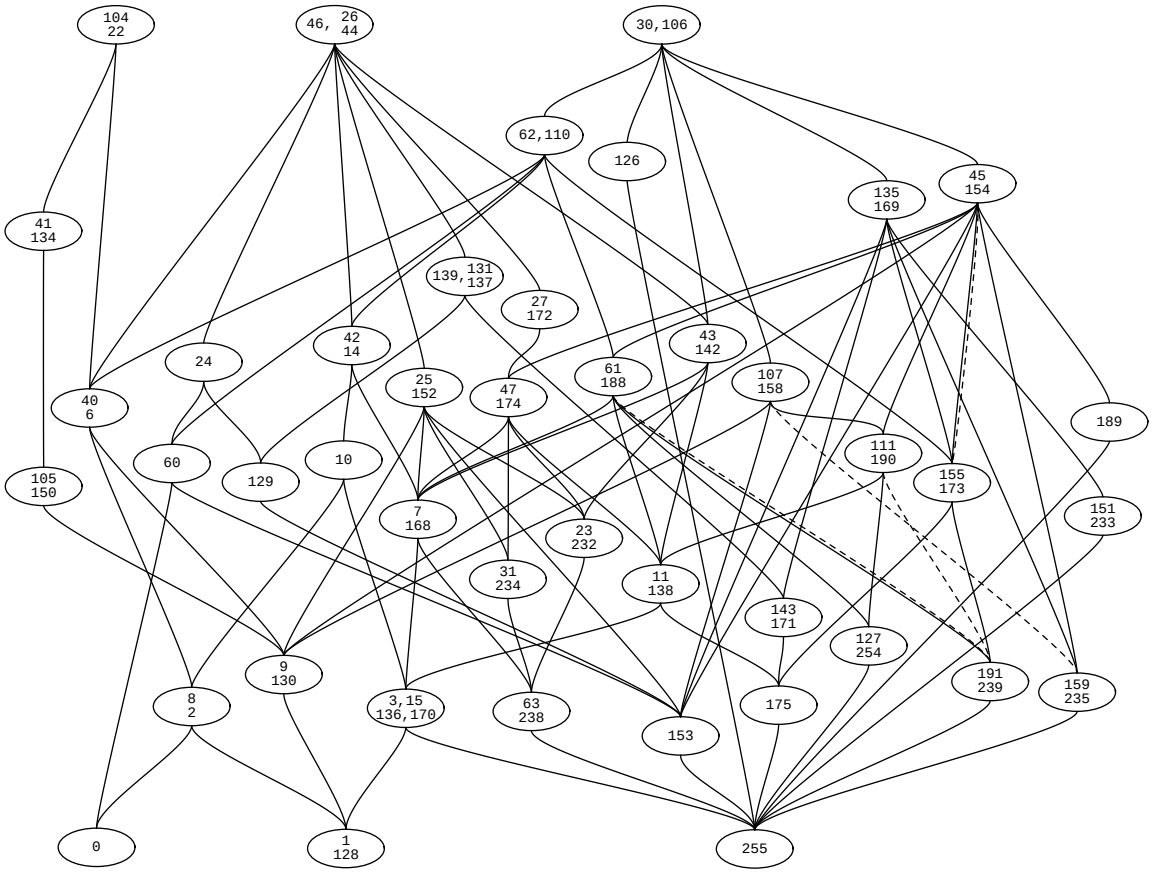


Figure 2.3: Which ternary boolean relations can implement which others. Each number indicates a relation by treating each acceptable triple as a three digit binary number, and then calculating $\sum_{n \in triples} 2^n$. For example, the number 30 (in the top right oval) is $2^4 + 2^3 + 2^2 + 2^1$, so the triples accepted by that relation are $\langle 1,0,0 \rangle$, $\langle 0,1,1 \rangle$, $\langle 0,1,0 \rangle$, and $\langle 0,0,1 \rangle$ (this is the relation $A = \text{NOR}(B,C)$). Where two relations can implement each other, they are shown in the same oval, separated by a comma. A surprising feature of the hierarchy is that this is not very common. If two relations are dual to each other (by flipping 0's and 1's), then they are shown in the same oval, one above the other, in which case the oval really represents two distinct points in the hierarchy, one for each relation. Lines between two such ovals only indicate implementability between the upper relations in each oval, and between the lower relations in each oval. A dotted line indicates crossing to the other side of the duality symmetry, so the upper relation in the upper oval can implement the lower relation in the lower oval, and the lower relation in the upper oval can implement the upper relation in the lower oval (these two implementations being equivalent by duality). Where three relations are listed in an oval, the two atop each other are duals of each other, but each of the three can implement the other two. The oval containing four relations represents two points in the hierarchy: one for the upper two relations, and one for their duals, the lower two relations.

of a sort that although quite common in Post's lattice of functions [Pos41], is quite rare in the lattice of relations without fan-out.

Theorem 26 *Any boolean relation which accepts tuples of both parities can implement the happy relation on two variables. Combined with any non-empty relation (possibly itself) on an odd number of variables, it can implement the happy relation on one variable (the "don't-care" relation).*

Proof: For the first claim, create a network using two copies of the relation. Let t_1 be a tuple of odd parity, and t_2 be a tuple of even parity. If a connection has value v_1 (either 0 or 1) in tuple t_1 and value v_2 in tuple t_2, say it is of type $2v_1 + v_2$. For any connection of type 0 or 3, use a wire to connect that connection of the first relation to that connection of the second relation. Then use wires to pair up connections of type 1 on the first relation, and also on the second relation. Do the same for type 2. This will leave a single connection (either of type 1 or of type 2) unpaired on the first relation, and a single connection (of the same type) unpaired on the second relation. Putting dangling wires on these connections, we get the happy relation on two variables.

For the second claim, given a non-empty relation on $2k+1$ variables, we can attach $k+1$ copies of the happy relation on two variables, which will completely cover the non-empty relation on $2k+1$ variables, leaving one wire of one of the bi-happy relations left over. This yields the happy relation on one variable. ■

For ternary boolean relations, this theorem tells us that if the relation is not parity preserving, then we know that "don't-care" is implementable, so in this case negation plus "don't-care" is implementable exactly if negation is implementable. If a ternary boolean relation *is* parity preserving, then it accepts an even number of either 1s or 0s, and so does any network built with it, so we know it cannot implement plain negation. However, if it can implement negation plus a constant, then the constants can be paired off and at most one extra constant will be left over after arbitrary negation. This means if we adapt A''s implementation of B' to have A implement B, then a constant may be left over when we are done. Whenever a constant is left over, a parity argument already shows that A cannot do B, since B has the wrong parity compared to what A can implement. Thus we have the following theorem.

Theorem 27 *If relation can implement "negation / don't-care" (number 60), then it can implement (without free negation) exactly those relations which correspond to what the version with free negation can do. If an odd-order relation is parity-preserving and can implement "negation / constant" (numbers 6 or 40), then it can implement (without free negation) exactly those relations which correspond to what the version with free negation can do.*

The following two metatheorems, along the lines of theorem 21, are interesting.

Theorem 28 *If property X is preserved under implementation, then the property Y, of being a (non-strict) superset (in terms of acceptable vectors) of some relation having property X, is also preserved under implementation.*

Proof: A network of type-Ys can be interpreted as a network of type-Xs plus extra possibilities, so the result is a type-X plus extra possibilities, which makes it a type-Y. ■

Theorem 29 *If property X is preserved under implementation, then the property Z, of being a (non-strict) subset (in terms of acceptable vectors) of some relation having property X, is also preserved under implementation.*

Proof: Any solution to a type-Z network is also a solution of a corresponding type-X network, so the implemented relation must be a (non-strict) subset of a type-X, which makes it a type-Z. ■

Unfortunately, these theorems are only occasionally useful. Any relation that doesn't preserve parity can, by theorem 26, implement the happy relation, which makes property Z true for all relations and therefore useless. If the unhappy relation has property X then property Y is similarly useless, and if supersets of relations having property X also have property X, then property Y is the same as property X. But sometimes the theorems turn out to be useful, as in theorem 21.

2.2.1 Preserved Properties

Here we prove that many specific properties are preserved under implementation, meaning that if a relation X has the property, and X can implement Y, then Y must have the property as well. For the proof, we will usually use the "juxtaposition / jumper" method of theorem 5. Occasionally, for properties that only apply to prime factors, we will use a similar method but instead of plain juxtaposition, we will consider juxtaposition with a wire connecting the two juxtaposed relations to maintain connectivity.

Theorem 30 (Convexity via the OR) *The following property is preserved under implementation: If a relation accepts A and B, then it also accepts any vector C that (1) is directly between them in Hamming distance and (2) is bitwise $\geq$ either A or B.*

Proof: Juxtaposition: Walk both relations to peak, then walk them both down. Jumper: If A and B are accepted with the same jumper value, then so is C. Otherwise, if A uses jumper=1 and B uses jumper=0, and C is a superset of A, then it is accepted with jumper=1. If C is a superset of B, then it is accepted with either jumper. ■

Theorem 31 (Convexity via the AND) *The following property is preserved under implementation: If a relation accepts A and B, then it also accepts any vector C that (1) is directly between them in Hamming distance and (2) is bitwise $\leq$ either A or B.*

Proof: This can be proved just like the previous theorem. ■

Theorem 32 (Conjunctivity) *The following property is preserved under implementation: If a network accepts A and B, then it also accepts the bitwise conjunction A OR B.*

Proof: Juxtaposition: Easy. Jumper: Easy. ■

Theorem 33 (Disjunctivity) *The following property is preserved under implementation: If a network accepts A and B, then it also accepts the bitwise disjunction A AND B.*

Proof: Juxtaposition: Easy. Jumper: Easy. ■

Theorem 34 (Parity of 1s) *The following property is preserved under implementation: All accepted vectors have even parity.*

Proof: Juxtaposition: Easy. Jumper: Taking away two identical bits always preserves the parity. ■

Theorem 35 (Parity of 0s) *The following property is preserved under implementation: All accepted vectors have an even number of 0s.*

Proof: Juxtaposition: Easy. Jumper: Taking away two identical bits preserves the parity of the number of 0s. ■

Theorem 36 (Superset of parity of 1s) *The following property is preserved under implementation: The relation's prime factors accept all vectors having an even number of 1s.*

Proof: Juxtaposition with wire: This construction still yields a superset of parity. Reducible: If a relation is reducible and is a superset of parity then it accepts everything, and its factors will have the property too, so we can just check prime factors. Jumper: Still superset of parity. ■

Theorem 37 (Superset of parity of 0s) *The following property is preserved under implementation: The relation's prime factors accept all vectors having an even number of 0s.*

Proof: Similar to the previous theorem. ■

Theorem 38 (Low weight) *The following property is preserved under implementation: All vectors with weight $\leq k$ are acceptable.*

Proof: Juxtaposition: Any giant vector with weight $\leq$k has this property on both subparts too. Jumper: For any small vector, the large vector with jumper=0 is ok. ■

Theorem 39 (High weight) *The following property is preserved under implementation: All vectors with at most k zeros are acceptable.*

Proof: Similar to the previous theorem. ■

Theorem 40 (Almost monotone) *The following property is preserved under implementation: From an acceptable tuple of values, any leg may change from* $0 \rightarrow 1$*, requiring at most one other to do so.*

Proof: Juxtaposition: The other leg, if any, will be in the same subpart. Jumper: If the other leg is not in the jumper, we're fine. If the other leg is in the jumper, then we cannot stop there, but must change the other jumper leg too, which may result in yet another leg changing. If that other leg is the first jumper leg, or the original changed leg, then we're having bad luck, but actually those can't happen, as we're only changing from $0 \rightarrow 1$. ■

Theorem 41 (Almost negative-monotone) *The following property is preserved under implementation: From an acceptable tuple of values, any leg may change from* $1 \rightarrow 0$*, requiring at most one other to do so.*

Proof: Similar to the previous theorem. ■

Theorem 42 (Monotone with specific factors) *A monotone (respectively negative-monotone) relation X can only implement relations that are factorable into relations that can be implemented simply by forcing legs to 1 (respectively 0).*

Proof: If a relation is monotone, then the set of vectors satisfying the network is the same as the set of vectors satisfying the netwok when all intenal wires are forced to 1. In the latter case it is clear that inputs connected to separate nodes are independent. So the only

implementable target relations are ones reducible to relations implementable by the original monotone relation with some legs forced to 1. ■

We note that although this theorem applies to many relations, we only need it here for 31 (in which a 1 on the controller forces a 0 on the other two wires), and its dual, 234.

Theorem 43 (Singletons and pairs of 1s) *The following property is preserved under implementation: For any tuple t accepted by the relation, the 1s of that tuple can be grouped into disjoint singletons and pairs such that the zeroing of any single group results in another acceptable tuple.*

Proof: Juxtaposition: Easy. Jumper: For any tuple setting the jumper to 1, group the zero, one, or two partners of the jumper legs. ■

Theorem 44 (Singletons and pairs of 0s) *The following property is preserved under implementation: For any tuple t accepted by the relation, the 0s of that tuple can be grouped into disjoint singletons and pairs such that converting any single group to 1 results in another acceptable tuple.*

Proof: Similar to the previous theorem. ■

2.2.2 The Final Eighteen Proofs

Of the 243 fundamental cases (not implied by any other cases) where we need to prove that one relation cannot implement another, 95 cases follow directly from the hierarchy of figure 2.2 and theorem 25, leaving 148 cases. The proofs above handle 113 of those cases, leaving 35 stubborn cases. But only one of the 35 (namely, $126 \Rrightarrow 189$) is among self-dual relations, so the other 34 can be paired up by duality, and only one member of each pair need be proven. This leaves a total of just 18 distinct stubborn cases that need to be attacked directly. Specifically, we need to show that the following implementations are impossible.

- $25 \Rrightarrow \{175\}$
- $27 \Rrightarrow \{9, 43, 153\}$

- $45 \Rrightarrow \{43, 151, 235\}$
- $61 \Rrightarrow \{9, 153, 155, 173\}$
- $111 \Rrightarrow \{9, 153\}$
- $126 \Rrightarrow \{189\}$
- $151 \Rrightarrow \{159, 191\}$
- $155 \Rrightarrow \{153\}$
- $159 \Rrightarrow \{191\}$

Many of these proofs share a common technique. Assume the network is in a stable state X, and consider another stable state Y. Paint all wires differing in the two states. Flippers can be sent forth, with instructions to stay on those wires. When a flipper gets to a vertex, it chooses some minimal acceptable set of flipped wires which is a subset of the painted wires and a superset of the wires flipped so far (including the incoming one). The number of exiting flippers may be 0, 1, or 2. It is rarely 2. By giving the flippers instructions on how to make the choice, or by wisely choosing X and/or Y in the first place, we can often show something about the structure of the network, or show that flippers going in different directions on the same subset could avoid proper operation of the allegedly implemented relation. A "down flipper" will mean a flipper that only flips 1s to 0s, while an "up flipper" only flips 0s to 1s.

Also, a relation that is at least 1-robust (every bad vector is isolated) is "flammable," meaning that if any leg or wire is "don't-care" (DC), then it "burns," meaning that given the two outgoing burnt wire values (at a ternary relation), one can always find a value for the originating burnt wire that satisfies the relation. Thus all the leaves of a burnt tree (counting places where it burned into itself in mid-edge as leaves) may be set arbitrarily.

$25 \not\Rrightarrow \{175\}$

25 has two legs equal and not all three may be 1. 175 is the $\leq$ relation on two legs. In a 25 net, started with all wires 0, an up flipper will only propagate along an equal-leg route. So if it pops out another leg, the implications all work in reverse too. This is unlike 175.

$27 \not\Rightarrow \{9, 43, 153\}$

27 is $A \leq B \leq \overline{C}$. Only B can be forced to 1. 9 is equality on two legs with the third forced to 0. 43 is MAJ with two legs negated. 153 is equality on two legs. Equality done with a 27 net would need to use Bs for the ends. Suppose the network is in a state corresponding to $1 = 1$. A down flipper coming in a B can only continue out A. Entering anywhere else it always dies. So path cannot be bidirectional like = needs. If 43 has last leg negated, then 011 and 101 are acceptable, and in each case a down flipper sent in the last leg must come out the other 1 leg. But a non-dying down flipper's path (enter B, leave A) is fixed by topology, and cannot depend on current wire values.

$45 \not\Rightarrow \{43, 151, 235\}$

45 is an AND-gate with one negated input. 43 is MAJ with two legs negated. 151 is $\Sigma \neq 2$. 235 is two wire ends whose controller, if 0, forces =. We will refer to the output of 45 as A, the negated input as B, and the plain input as C. Starting from a 45 net with all wires 0, there are only up flippers. An up flipper in a 45 net never splits into two. Entering B, it dies. Entering A, it exits C. Entering C, it can choose A or B. Suppose a net has leg x which if it flips up forces y to flip up too (if nothing else does). Then the path must contain only $A \to C$, $C \to A$, and $C \to B$ transitions at the vertices (but note that the connection type need not be the same at both ends of an edge). To do 235, there is a reversible forcing path from 0s, so it doesn't use B. But there is also some path to the third leg, so it must use a B. This cannot implement 235, since routes to B cannot merge from other two legs. For 43, the contradiction is because routes to negated leg cannot merge from other two legs. For 151, we will need to consider a general state of the network. The 1-valued wires form chains that do not split, though they may end. Any chain may be fully converted from 1 to 0 and the network remains satisfied. Consider a network implementing 151, with every input a 1. At least one of the three legs must start a chain that ends somewhere inside the network. This may be converted to 0, so it's not a 151.

$61 \not\Rightarrow \{9, 153, 155, 173\}$

61 is a negated wire whose controller, if 0, allows 00. 9 is equality on two wires and a forced 0. 153 is equality on two wires. 155 is a wire (B, C) whose controller (A), if 0, allows $B = 0$

and $C = 1$. 173 is a wire (A, C) whose controller (B), if 1, allows $A = 0$ and $C = 1$. In a 61 net, starting from all 0s, an up flipper dies unless it enters C, in which case it can choose where to leave. Since this is a unidirectional process, equality (9, 153) cannot be done. A 61 net doing 155, started with 0s, could take an up flipper at 155's B. This must emerge at 155's C. But in a 61 net, we can instruct the flipper to head straight towards 155's A. Similarly for 173, a flipper starting at A can emerge at B (or nowhere) instead of C.

$111 \not\Rightarrow \{9, 153\}$

111 is two loose wires whose controller, if 1, connects them negatedly. It is flammable. 9 is equality and 0. 153 is equality and DC. Connecting the two loose wires forces the controller to 0. So equality is the only question. A cycle of wires burns if length is even. If odd, then at least one controller must be 0. A 111 net, started from 0s, kills up flippers unless they enter the controller, in which case they may leave where they like. Thus, (proof 1) they may avoid any particular destination (by heading for a different destination), and (proof 2) their route is not reversible (a flipper started from the other end could die immediately).

$126 \not\Rightarrow \{189\}$

126 is $\neq_3$. Any cycle is equivalent to DC on the legs sticking out from the cycle, because given any legs, we can walk around the cycle setting its edges so as to satisfy all the nodes. Any tree is $\neq_k$ with legs negated according to the two-coloring of the tree. Any tree with v vertices has $v + 2$ legs. 189 is $\neq_3$ with one leg negated, which is prime and must be implementable (if at all) with a network of a single component. Since legs are not all negated the same, it must have at least two vertices. But this is impossible: Since it has only three legs, it must be implemented with just one vertex.

$151 \not\Rightarrow \{159, 191\}$

151 is $\Sigma \neq 2$. 159 is two loose ends whose controller, if 1, connects them (forces equality). 191 is OR_3 with two legs negated. Any DC on a 151 burns the whole thing. Any cycle can be set to 0 and is a giant DC. So assume the network is a tree. Any tree with v vertices has $v + 2$ legs. 159 and 191, being prime, cannot be interpreted as disconnected trees. A tree with 3 legs has a single vertex, and can only be 151.

$155 \not\Rightarrow \{153\}$

153 is equality on two wires. 155 is a wire (B, C) whose controller (A), if 0, allows $B = 0$ and $C = 1$. Start with the network of 155s having all wires 1. A nondying flipper's path through 155 must use one of the following transitions at each vertex: $B \to A$, $B \to C$, or $C \to B$. If the path ever enters at B, it has a choice, but the two choices can never remerge, so no particular outcome can be forced. So the path forcing equality must always enter at C and leave at B. But it can't do this both ways.

$159 \not\Rightarrow \{191\}$

159 has a controller leg, which, if 1, forces the other two legs to be equal. 191 is OR with two legs negated.

Two legs on the 191 can be forced to 0 (by 01), and one can be forced to 1 (by 11). An up flipper coming in 159's controller can die or exit as either an up or down flipper. A flipper coming in another 159 wire can die, continue on the other wire, or exit as a down flipper on the controller. If the 159 network has a cycle of the controlled wire, then it becomes a DC on all the controllers. A DC on any 159 leg makes both other legs DC, so the whole component burns up. So all wires end at a leg or a controller. If a wire has controllers at both ends, it can all be set to 0 and the component burns. So all wires have at least one end at a leg. So there are at most three wires in a network implementing a 191. Three or two. Counting wire ends, the number of wires is 3 plus the number of vertices, divided by two. So three wires means three vertices, and two wires means one vertex. These possibilities can be exhaustively verified not to implement 191.

2.3 Comments on the Hierarchies

We initially started studying these ternary boolean relations in the hopes of finding some general principles or overall order. However, this is not what we found. Although these investigations did not take us where we had hoped, they did yield quite a lot of information. In addition to detailed information about what can implement what, inspection of the hierarchies also provides some more general higher-level information:

- There is no general proof method for all unimplementabilities.

- There is no clear overall structure to the hierarchies.
- Implementations within the hierarchies have small sizes.
- Equivalence between relations is rare.
- Comparability of relations is common.
- Incomparability of relations is common.
- Most other relations cannot implement fan-out.
- No ternary boolean relation can implement all others.

We will show in chapter 3 that the first two points are actually provable, as they are consequences of theorem 47. Thus the reason our investigation failed to fulfill our hopes was because our hopes were unfulfillable.

Interestingly, this same theorem shows that the third point is not true in general, since if it were true that implementations always have small sizes, then it would be easy (if tedious) to decide whether one relation implements another just by trying all small networks.

The last item contrasts with the situation for functions, where the NAND function of two variables can implement all other boolean functions. As discussed on page 53, we must go to larger relations to find universal ones.

The difficulty of implementing fan-out is surprising, given how we take it for granted in so many areas, from circuit design to universal algebra. However, quantum circuits are one area in which we know fan-out is not available, and chemical systems (as discussed in section 3.4) generally do not have fan-out either. The hierarchies give us the intuition that if fan-out is not somehow explicitly available, we are very unlikely to be able to implement it. This provides extra motivation for considering the case when it is not available.

2.4 Complexity of Deciding Whether There Is a Solution

A natural question to ask is, if we know that a relational network is built out of some particular building block, does that allow us to quickly analyze the network to check whether there are any solutions?

The categorization of the complexity of this decision problem for all boolean ternary networks with free negation is in fact mentioned as an open problem in [Val02]. (The categorization presented here was completed before I became aware of the connection to Valiant's work.)

The answer to this question turns out to be very simple: Solvability can be decided in polynomial time for each of the boolean ternary relations (with free negation), except for the AND-gate relation, which can create circuits whose solvability question is NP-complete.

The hierarchy aids us greatly in proving this, since we only need to demonstrate polynomial time algorithms for the first two peaks of the hierarchy, along with the nodes directly under the third peak. The rest of the nodes in the hierarchy (except for the third peak) can be implemented by one of those using a simple item-by-item substitution, and so a linear time transformation will allow application of the algorithm for the higher node.

For the third peak (the AND-gate relation), we will give a proof using an idea from Feder [Fed01], and Valiant also mentions in [Val02] that Hunt and Stearns [HS90] also proved this result.

For the relations in the larger hierarchy, where negation is not provided, an algorithm designed for the case where negations may be present will of course still work. So the only thing that needs to be checked in the larger hierarchy is whether any of the relations corresponding to the AND-gate relation become easily analyzable when negation is unavailable. There are three cases to consider.

An AND-gate relation with its output negated is a NAND-gate relation, and this can be used to implement negation, so it is equivalent to the case where negation is available.

An AND-gate relation with zero or one input wires negated accepts the all-zeros tuple, and so any network of these is satisfiable by setting all variables to zero.

An AND-gate relation with two or three of its wires negated is dual to an AND-gate relation with one or zero of its wires negated, so the satisfiability question is equivalent to one of the cases covered above.

In conclusion, only the NAND-gate and NOR-gate relations (which can implement each other, and are thus a single node (the third peak) in the big hierarchy) have NP-complete solvability questions. For all other relations, the solvability question is decidable in polynomial time.

Of course, if multiple types of relations are available, such as $=1$ together with fan-out,

then it is in general much easier for the solvability question to be NP-complete. However, we have not made any attempt to categorize all 2^{80} such cases.

For the case where fan-out is available, things are quite a bit simpler, and the entire infinite lattice above fan-out was analyzed by Schaefer [Sch78].

We will now provide the proofs.

2.4.1 Solvability of Networks of AND-gate Relations is NP-complete

Clearly any solvability question for any particular network of relations is in NP, since it is easy to give a certificate that there is a solution. To show that this problem is complete in NP, we reduce formula satisfiability to it. Given a formula, we will build a network out of AND-gate relations and negation relations that has a solution if and only if the formula is satisfiable.

Given a formula, it is easy to construct a feed-forward circuit of AND, OR, and NOT gates that evaluates the formula. Each literal in the formula is an input to the circuit, so one variable might appear at several inputs. The output of the circuit is 1 if the formula is satisfied by the given inputs, and 0 if not. First we convert the OR gates in this circuit into combinations of AND and NOT gates, since by DeMorgan's laws an OR gate is the same as an AND gate with every wire (both inputs and output) negated. Now that the circuit consists solely of AND and NOT gates, we are ready to convert it into a network of AND-gate relations and negation relations. The reason we reduce to formula satisfiability as opposed to circuit satisfiability is that circuits may have fan-out greater than one, but we do not have fan-out available. Formulas do not have any fan-out, so our lack of available fan-out is no problem for formulas.

The conversion to AND-gate relations and negation relations is direct, but it does not finish off the problem—it leaves all the inputs and output of the circuit as dangling wires, which we need to do something with. The output is simple enough: We want to force it to be 1, so we simply attach a gadget that forces the output to be 1, for example the negated output of an AND-gate relation whose two inputs are connected to each other with a negation.

The inputs are trickier. The problem is, a variable (say x) may appear at several of the inputs, but since fan-out is not available, we have no way to force those inputs to be the

same. The solution to this dilemma is that we do not have to force the inputs to all be the same. A weaker forcing will suffice.

For any given input wire of the circuit, there is a single path from that input to the output, and depending on the parity of the number of negations between the input and the output, we can say that the output is *positive* or *negative* in that particular input. Specifically, if the output is positive in that input, then changing that input from 0 to 1 can only increase the output. Similarly, changing a negative input from 1 to 0 can only increase the output. In light of this, our weaker requirement on the inputs will be the following: Instead of forcing all the inputs for variable x to have the same value, we will simply require that if any of the positive inputs for x are 1 (as they would like to be, to help the output be 1), then all negative inputs for x must be 1 (that is the price you must pay if you want the benefit of x being 1). Likewise, if any of the negative inputs for x are 0 (which they would like to be, to help the output be 1), then all positive inputs for x must be 0 (that being the price that must be paid).

This weaker requirement seems complicated, but it is very easy to implement: Simply attach all the positive inputs for x to the inputs of an OR-gate of appropriate fan-in size, and attach all the negative inputs for x to the inputs of an AND-gate of the appropriate size. (These gates are easily implementable with negation and AND-gates of two inputs, by using DeMorgan's laws and using trees of small gates to make big gates.) Then, attach the outputs of the OR-gate and the AND-gate to each other with a wire that we will call the *indicator* wire for x. (If x is used only positively or only negatively, so that the OR- or AND-gate is missing, then simply attach a "don't-care" gadget to the other side of the indicator wire, for example the output of an AND-gate relation whose inputs are attached to each other.) In this way, we can make an indicator wire for each variable of the formula.

Now, the indicator wire forces the inputs in exactly the way we wanted. If any positive input for x is 1, then the indicator wire for x must be 1, and so all the negative inputs for x must also be 1. Likewise, if any negative input for x is 0, then all positive inputs for x must also be 0. The indicator wire indicates which value the variable will take so as to satisfy the formula.

We now claim that the network is satisfiable if and only if the formula is satisfiable. If the formula is satisfiable, we can clearly set the indicator wire and all the inputs for each variable to the satisfying value of that variable, and the network of relations will be satisfied.

Conversely, if the network of relations is satisfiable, then we can change all the inputs for a variable to match the value of its indicator wire (and change gate outputs accordingly, down into the circuit), and still have a solution to the circuit. This is because the only way an input can differ from the indicator wire is if the indicator wire is 1 but a positive input is 0, or the indicator wire is 0 but a negative input is 1. Either way, changing a positive input from 0 to 1, or changing a negative input from 1 to 0, the output of the circuit will remain a 1. Thus we arrive at a solution to the formula, and we are done with the proof.

Note that this construction can be modified slightly to give a general purpose construction for implementing an arbitrary relation using just the NAND-gate relation and at most one fan-out relation per wire of the relation to be implemented. Simply construct the circuit for the characteristic function of the relation as described above, and then put a fan-out relation in the middle of each indicator wire, so each has a dangling edge. This network clearly implements the desired relation. Such an implementation can even be done with a planar network, using the well-known pattern of twelve NAND-gates that allows wires to cross. (And if one wants to avoid fan-out entirely, one can use dual-rail logic.)

2.4.2 Networks of $=1$ Relations Can Be Solved in Polynomial Time

In a network of $=1$ relations and negation relations, in any solution, each $=1$ relation must touch exactly one 1, and each negation relation must also touch exactly one 1, so we see that the 1 edges must form a perfect matching among the vertices. Thus there is a solution if and only if the graph has a perfect matching (with negations counted as vertices). There are a variety of polynomial time algorithms for finding perfect matchings [BBDL01].

2.4.3 Networks of $\cdot\leq\cdot\leq\cdot$ Relations Can Be Solved in Polynomial Time

The $\cdot\leq\cdot\leq\cdot$ relation (the relation on three variables, that they must be nondecreasing) can implement fan-out as well as the two-variable $<$ relation, which is the same as implication ($\Rightarrow$). Conversely, $\Rightarrow$ and fan-out together can implement $\cdot\leq\cdot\leq\cdot$, so it is possible to convert back and forth between networks of $\cdot\leq\cdot\leq\cdot$ relations and networks of $\Rightarrow$ and fan-out relations.

Networks of $\Rightarrow$ relations, with reusable variables and free negation, correspond exactly to 2-SAT problems, which are well known to be efficiently solvable by depth-first search (using a simple optimization that is equivalent to the burning of zipper wires, as described

in section 1.8.5).

2.4.4 Networks of $\neq_3$ Relations Can Be Solved in Polynomial Time

If there is any cycle in the network, then all non-cycle edges touching a cycle vertex may be set arbitrarily, and the edges in the cycle may be set so as to satisfy every vertex in the cycle, simply by starting at a $\neq_3$ relation, setting a cycle edge to the opposite of the non-cycle edge, and continuing greedily around the cycle.

Treating the cycle as a single root node, we can find a spanning tree of the resulting graph, and orient each edge of the spanning tree away from the root. (We assume without loss of generality that the graph is connected.) We will then say that each edge in the spanning tree is *responsible* for the vertex it points to, whether the vertex is a $\neq_3$ relation or a negation.

Now we may set all edges not in the spanning tree to arbitrary values, and then work our way from the leaves of the spanning tree back towards the root, having each edge choose a value so as to satisfy the vertex it is responsible for, and then finally at the root we choose values on the cycle as described above.

Thus every closed network of $\neq_3$ relations is satisfiable.

2.4.5 Networks of $\neq 1$ Relations Can Be Solved in Polynomial Time

We will assume without loss of generality that the network graph is connected. If the total number of negations is even, then pair up the negations and for each pair draw a path connecting the pair. First we convert the pairings and paths to be non-overlapping: Color each edge red which participates in an odd number of paths. The red edges must consist of paths between negations and closed loops. If the red edges are assigned the value 1, and all other edges are 0, then we have a solution to the network.

Now suppose there are an odd number of negations. Suppose there is a $\neq 1$ relation none of whose three edges is a bridge (a cut edge). Then we can add a temporary negation on one of these edges to make the total number of negations even, and solve as above. Next, we want to tweak the solution so that the edge from the $\neq 1$ relation to the temporary negation is a 0, with the other two edges of the $\neq 1$ are each a 1. If the solution does not already have this property, then we can convert it to this form by finding a cycle passing through

the two edges we'd like to flip (we know we can find such a cycle because neither of those edges is a bridge), and flipping the value (the redness) of every edge on the cycle. This will clearly take us from one solution to another acceptable solution. Once we have done this, then we can flip the edge between the $\neq 1$ relation and the temporary negation to be a 1, and remove the temporary negation, to arrive at a solution to the original network.

Now suppose every $\neq 1$ relation has a bridge for at least one of its edges. Suppose one of the relations has exactly one bridge edge, and it has the property that if we disconnect it from the relation (so the graph becomes disconnected, with the bridge edge dangling), then there are an odd number of negations on the side with the dangling bridge edge. In this case, we can solve the other side and tweak it so the two remaining edges of the relation are 1s, and we can add a temporary negation to the dangling bridge edge, solve the other side, and then reconnect the dangling bridge edge back to the relation and remove the temporary negation as before.

If this is not possible, from either end of the bridge (which may be a long bridge with negations along it), then we see that the bridge must have an odd number of negations, and if we remove the (possibly long) bridge, then the two remaining components each also have an odd number of negations. Thus the form of the graph that we are considering at this point is a tree of cycles, where every cycle has an odd number of negations on it, and every edge of the tree has an odd number of negations. Such a structure can be recursively pruned and seen to be unsatisfiable.

Finally, if we have a relation with three bridge edges, then removing it leaves three components, and we know the total number of negations is odd, so of these three, either one or three must have an odd number of negations. If three do, then we can add a temporary negation to each, solve each, and then put them back together and remove the temporary negations so the three-bridge relation has three 1s. Otherwise, every three-bridge relation joins three components with an odd number of relations each. Thus any bridge between two such relations also has an odd number of negations, and these relations may be processed along with the cycles as above to show that the network is unsatisfiable.

2.4.6 Networks of WORM Relations Can Be Solved in Polynomial Time

As we might guess from its name, the WORM relation is not one that we are used to thinking about. It may be the most unnatural of all the relations in figure 2.2, but like anything else, one can get used to it. We can think of the WORM relation as a negation on a wire, where the end of a third wire, the *controller*, controls the negation. If the controller is 0, then the negation operates normally. But if the controller is 1, then it allows the other two wires to also be 1s, if they don't want to be the negation of each other.

We will consider the wires to be in loops and paths. We will think of a negation as being on a wire, so the wire is thought of as being the same wire on both sides of the negation, so for example we may speak of how many negations are on the wire. The two non-controller wires at a WORM relation will similarly be considered to be on the same loop or path, and the relation will count as a negation. Thus every wire is either a loop, or a path with a controller at each end.

If a loop has an even number of negations, then it may be assigned values that alternate at each negation, so that none of the impinging controllers are relevant. A wire that is irrelevant at one end can be said to "burn," meaning that it can be oriented to point away from the source of burning, and if it is not a controller, it can devote itself to successfully satisfying the relation it points to (whether WORM or negation), thus freeing the other wires at that relation to burn, and so on.

A loop or path containing an even number of negations may burn. (The loop, by alternating values around the loop, and the path, by alternating values so that both ends are 1.) A negation with a burnt controller is a two-way OR relation, which if on a loop allows the loop to burn, and if it is an odd-numbered negation along a path, allows the path to burn. An even-numbered negation along a path cannot make any use of a controller being 1, so any controllers on even-numbered negations along a path may burn.

At this point, all loops and paths have an odd number of negations, and paths only have controllers at odd positions along them. If anything burns now, the whole thing will burn, meaning the original network is satisfiable. If there is a cycle that passes through a controller, we can set the controller to 1 to start things burning, and when the burning comes around the cycle to that controller, that shows that it was ok to set it to 1.

If anything is left at this point, it is a tree (treating loops as points) whose loops and

paths all have an odd number of negations, and only odd-numbered negations on paths have controllers. Now each WORM relation has to choose at least one side: either the controller side (by enforcing the negation on the wire), or the wire side (by setting the controller to 1). Since the structure is a tree (except for loops), some loop or path must get left out, with no relation choosing it. (This loop or path may be found by following the "choice directions" backwards until we get stuck.) That loop or path cannot be assigned values in a satisfactory way, so the network is not satisfiable.

Chapter 3

Decidability

This chapter will discuss issues of decidability. There are two main areas in which decidability results are examined. The first is questions of implementability among relations without fan-out. (Some old and new results on the case when fan-out is available are given as well.) The second is questions of implementability among *functions* without fan-out. This is a topic that has not received as much attention in the literature as one might expect. We show that the question of implementability in this context is equivalent to the question of reachability for Petri nets, chemical reaction networks, and other systems.

For chemical reaction networks, we prove that using standard reaction rate kinetics, Turing machines can be reliably simulated using a probabilistic method. Without probabilities, the power disappears, and the systems become only as powerful as primitive recursive functions. This power of probability is the first example we are aware of where treating a system probabilistically dramatically increases the range of functions it is able to compute, in this case from primitive recursive or lower to general recursive. (Several examples are already known where treating a system probabilistically can dramatically improve the *efficiency* of the task at hand, with respect to some precious resource, for example reducing communication in communication complexity or reducing congestion in network routing.)

For Petri nets, our results show that there is a big difference between two probabilistic models that might seem similar at first. If at each step, a transition is chosen at random, and then it fires if it is enabled, then the probability of reaching any given state can be calculated to any desired precision. However, if at each step, a token is chosen at random, and the token chooses an enabled transition that will absorb it upon firing, then even the approximate probability of reaching a given state can be undecidable.

3.1 Decidability $\Longleftrightarrow$ Fan-Out

The availability of fan-out turns out to be central to several questions of decidability. First we will show that if fan-out is available, then the implementability question for relations is decidable. We show this by giving an explicit decision algorithm. Then we will show that when fan-out is not available, the implementability question for relations is undecidable. We show this by giving an explicit construction for reducing an arbitrary halting problem to this form of implementability question.

3.1.1 Fan-Out $\Longrightarrow$ Decidable

Much previous work exists on the analysis of relations where fan-out is available [BKKR69, BKJ03, Gei68, Pip97]. Here we present a proof due to Pippenger [Pip97], followed by a significant improvement in the algorithmic complexity of the decision procedure.

Theorem 45 *There is a general procedure for deciding the following question: Given a set of relations $\mathcal{X}$ that includes fan-out (or that can implement fan-out), can a graph of relations from $\mathcal{X}$ be built that implements a given desired target relation Y?*

Proof: The main idea is the following: Suppose a graph accepts and rejects certain sets of tuples, $\mathcal{T}_{\text{acc}}$ and $\mathcal{T}_{\text{rej}}$. For each accepted tuple, there is by definition some assignment of values to edges that is acceptable to every relation. We will arbitrarily choose one particular such accepting assignment for each accepted tuple and call it the witness assignment for that tuple. For any set of two or more edges that have the same value in every witness assignment, we will replace those edges with a connected graph of equality relations, effectively connecting those edges together so they are forced to always have the same value. This new graph will still be able to use essentially the same witness assignments to accept every tuple that was previously accepted, and the new restrictions (forcing certain edges to always match in value) will certainly not allow any previously rejected tuples to become acceptable, so the new graph is implementing the same relation that the old graph implemented. If we use just a single variable for each subgraph of edges connected by equality relations (as if it were a hyperedge for a hypergraph version of the problem), and there are s possible values that a variable can have, then we see that there are at most $s^{|\mathcal{T}_{\text{acc}}|}$ variables,

since any edges matching in value on each of the $|\mathcal{T}_{\text{acc}}|$ witnesses have been connected to use a single variable. Since there is a bound on the number of variables, this means there is a bound on the number of ways a relation can be applied to the variables, and thus a bound on the number of possible graphs, and thus the problem is decidable by exhaustive search. ■

To Pippenger's proof we add the observation that actually the search can be narrowed down to simply checking one single graph which will implement the target relation if anything can. This graph has all $s^{|\mathcal{T}_{\text{acc}}|}$ variables, and relates them in all possible ways (compatible with $\mathcal{T}_{\text{acc}}$) using relations from $\mathcal{X}$. The relations are therefore compatible with the values of the variables for each of the $|\mathcal{T}_{\text{acc}}|$ required solution states of the graph, but the graph is maximally restrictive subject to accepting what it needs to. All that then needs to be checked is whether the target relation Y is indeed enforced on the set of variables to which it would be connected if it were in $\mathcal{X}$. This simplification makes the algorithm feasible for small cases, whereas otherwise even the simplest cases cannot be completed in any reasonable amount of time.

When contrasted with the next section, this theorem shows how the presence of fan-out can have a surprising effect on our ability to analyze the power of a given set of relations.

Indeed, when fan-out is present, not only does implementability become decidable, but there are some powerful theorems ([BKKR69, Gei68]) that can convert any question of implementability for relations into an implementability question for functions (which combine to implement other functions using regular function composition), and vice versa, by means of a *Galois connection* between the two (as discussed on page 54). The implementability question for functions has been studied since Post [Pos41], who in fact investigated a more general model than just what the Galois connection applies to. Despite trying, we have not been able to generalize the Galois connection so as to apply to general networks of relations.

3.1.2 No Fan-Out $\Longrightarrow$ Undecidable

This section will prove the following theorem:

Theorem 46 *There is no general procedure for deciding the following question: Given a*

relation X, can a network of X's be built that implements a relation that rejects a given tuple T?

From theorem 46, many corollaries immediately follow, showing that many natural questions of implementability are undecidable:

Theorem 47 *There is no general procedure for deciding the following question: Given a relation X, can a network of X's be built that implements a given desired target relation Y?*

Proof: If there were a procedure for deciding this question, we could use it to decide whether or not X can implement Y for every possible Y relating a given number k of values (there are only a finite number of such Y, since the values must be from the finite alphabet of values that can be accepted by a variable participating in X). This would then tell us whether or not any particular k-tuple T can be rejected by a network of X's, which goes against our main theorem. ■

We can also consider implementability questions involving *sets* of relations rather than just a single relation:

Theorem 48 *There is no general procedure for deciding the following question: Given a* set *of relations $\mathcal{X}$, can a network of relations from $\mathcal{X}$ be built that implements a relation that rejects a given tuple T?*

Proof: Trivial: Consider a singleton set $\mathcal{X}$ and use Theorem 46. ■

Theorem 49 *There is no general procedure for deciding the following question: Given a set of relations $\mathcal{X}$, can a network of relations from $\mathcal{X}$ be built that implements a given desired target relation Y?*

This can be proved just like the previous ones. In some sense this seems like the most natural or general way to phrase the question of implementability. If the question *were* decidable, this is what you would want the decision procedure to be able to do. Recall that in the previous section, we saw that in the traditional setting where the "fan-out" relation (the equality relation on three variables) is in the set $\mathcal{X}$, this question becomes decidable.

Theorem 50 *There exists a fixed relation X for which there is no general procedure for deciding the following question: Can a network of X's reject a given set of tuples $\mathcal{T}$?*

This is not really a corollary, as it does not follow from theorem 46. However, it can be proved along almost identical lines as theorem 46, as described in section 3.1.2.6. Analogous corollaries follow from this theorem as well.

The proof of our main theorem constructs an X with a large alphabet and a small number of variables (three variables). Is the question still undecidable if X has a small alphabet (but more variables)? The answer turns out to be yes (boolean values are sufficient), but new ideas are needed for the construction in this case, as discussed in section 3.1.2.6. Of course, if one limits X to both a small (bounded) alphabet and a small (bounded) number of variables, then there are only a finite number of possible distinct choices for X, and so all such questions about small relations are necessarily decidable. (For example, all such questions about relations on just three boolean variables can be decided by referencing figure 2.3.)

If infinite networks are allowed (which strikes mathematicians as natural, but not too many other people), then the undecidability results are unchanged, although the proofs must be modified slightly.

3.1.2.1 The Line Between Decidability and Undecidability

After The following theorem highlights the different nature of acceptance and rejection.

Theorem 51 *There is a general procedure for deciding the following question: Given a set of relations $\mathcal{X}$, can a network of relations from $\mathcal{X}$ be built that implements a relation that* accepts *a given tuple T?*

Proof: The idea here is that each acceptable tuple in each relation in $\mathcal{X}$ contains either an even or an odd number of instances of each possible variable value. We can think of each of these acceptable tuples as a 0/1 vector of length s, if there are s values appearing among the acceptable tuples. The given tuple T also contains either an even or an odd number of instances of each variable value. If we can find a subset of the 0/1 vectors whose sum (mod 2) matches the tuple T (which is a straightforward linear algebra problem), then we just lay

them all out on the table and start connecting identical values in pairs, and the end result (without T) is the desired network. If we cannot find such a set, then clearly no accepting network can be built. (To avoid connecting a pair of values appearing in T, we may need to add a pair of identical acceptable tuples containing that value (assuming such a tuple exists), which then allows us to connect values in pairs so that the pair of instances of that value in T are connected to relations in the network rather than directly to each other.) ■

Even asking about the acceptance of entire *sets* of tuples leaves us in the decidable realm:

Theorem 52 *There is a general procedure for deciding the following question: Given a set of relations $\mathcal{X}$, can a network of relations from $\mathcal{X}$ be built that implements a relation that accepts a given* set *of tuples $\mathcal{T}$?*

Proof: This problem can be reduced to the previous problem by replacing the set of variable values $\mathcal{S}$ with a larger set of variable values. If there are m tuples in the set of tuples $\mathcal{T}$, then we collapse $\mathcal{T}$ into a single tuple T, each of whose values is a member of the set $\mathcal{S}^m$. The available relations in $\mathcal{X}$ can be similarly converted into relations on the larger alphabet $\mathcal{S}^m$. We can then continue as in the proof of the previous theorem, using this larger alphabet of values. ■

It is at first surprising that the decision problem for acceptance should be so easy when the decision problem for rejection is undecidable. One way to understand this intuitively is that a network accepts a tuple if "$\exists$ an assignment of values to edges such that $\forall$ relations in the network, the relation is satisfied." On the other hand, a network rejects a tuple if "$\forall$ assignments of values to edges, $\exists$ a relation that is not satisfied." Now, we have been considering questions of the form "Does there exist a network that accepts/rejects something?" So for acceptance, we are prepending an existential quantifier to something that already started with an existential quantifier, whereas for rejection, we are prepending an existential quantifier to something that started with a universal quantifier, and the additional level of alternation manages to complicate the problem considerably.

3.1.2.2 An Overview of the Proof

The way we prove Theorem 46 is by showing that for any program P, we can set T to be a simple one-tuple (just a single value to reject on a single dangling edge), and then we can carefully construct a relation X on three variables such that the only way a network of X's can reject T is for the network to have a structure that corresponds to the execution of P, with P halting. So if P halts, then it is possible to build a (unique) network of X's that rejects T, but if P does not halt, then any network of X's will accept T. Since this reduces the halting problem to a question of the form given in Theorem 46, it proves the theorem.

Our construction is based on programs P that are "register machine" programs (also known as "counter machine" or "Minsky machine" programs—Minsky himself called them "program machines." [Min67]) They work like a simplified assembly code, in which the only operations are incrementing and decrementing one of a fixed number of registers, and the decrement operation can branch according to whether the register was zero (undecrementable) or not. An example will be given in figure 3.1.

In constructing the relation X (based on P), we find ourselves doing a very strange kind of programming. Each part of the strange program encoded in the acceptable tuples of X is essentially a detector which checks to see if the network topology does not correspond in the intended way to the execution of program P. That is, the strange program in X is built up of pieces that specify what should *not* happen when program P runs. Each piece is designed so that if the undesired topology can be detected, then the one-tuple T will be acceptable to the network. The strange program in X is complete when all undesired topologies have been excluded, and the only remaining possibility for the network topology is to be a perfect representation of the execution of P. The pieces of the strange program in X only get "run" if the network topology is failing to represent the execution of P in the way being checked for by that piece of the strange program. If the network topology corresponds perfectly to the execution of P, then no part of the strange program in X can be run, and thus T is not accepted by the network.

Thus, the construction here has a rather different flavor from most undecidability constructions, in which one shows how some new and different simple system is capable of performing computation. Here, we take a new and different simple system, and show how we can get it to detect *all* the situations where it is *not* performing computation. Like

sculpting with chisel and stone instead of with glue and wood, instead of making gadgets that slowly build up more calculational power until the desired program can be executed, this construction has gadgets that chip away at miscalculations until all that is left is proper execution of the desired program. In this regard, this proof is similar to the standard proof [Sip97] that it is undecidable whether a context-free grammar generates all strings, although that proof is far more straightforward.

3.1.2.3 Register Machines

Register machines are a simplified, idealized abstraction of how computers work, with a cpu manipulating memory. (Different texts often use slightly different definitions for the details of how they work, but these differences are never of consequence for the results.) Minsky showed in the 60's that register machines are capable of universal computation.

A register machine is a machine that has a fixed number of *registers*, each of which can hold an arbitrary non-negative integer. In addition to the registers, it has a fixed *program* which consists of a set of instructions. Every instruction is either an increment instruction, a decrement instruction, or a halt instruction. The increment and decrement instructions specify which register is to be incremented or decremented, and they also specify which instruction should be executed next, after the increment or decrement. Decrement instructions, however, might not succeed with their intended decrement—if the register is 0, it cannot be decremented. In this case, the decrement instruction is said to *fail*, and each decrement instruction specifies an alternate next instruction to go to in the case that the decrement fails. The current *state* of a register machine is given by the values of the registers, along with which instruction is the next one to execute.

Register machines are nice because of their simplicity, which makes it easy for other systems to simulate them. Not only are they well suited to the present proof, but our proof of theorem 53 will also work by showing how chemical reaction networks can simulate register machines.

As an example of a register machine program in the format we will use, we present one in figure 3.1 that implements the well-known "$3x + 1$" procedure [Lag85]. This procedure manipulates a positive integer value repeatedly in the following way: If the number is even, divide it by two, but if it is odd, then multiply it by three and add one. This process is

	action	next	but if zero
1.	x-	2	4
2.	y+	3	
3.	x-	1	8
4.	y-	5	6
5.	x+	4	
6.	x-	7	5
7.	x-	9	halt
8.	y-	9	3
9.	x+	10	
10.	x+	11	
11.	x+	8	

Figure 3.1: A register machine program for the $3x + 1$ procedure. The "$3x + 1$ conjecture" corresponds to the conjecture that this program will always eventually halt, regardless of the initial values of its registers. As an example of why it is hard to show that this program always halts, observe that if it is started with the value 9 in both x and y, it will toil away for over $100,000$ steps before halting.

repeated until the value is eventually reduced down to 1. The well-known conjecture (which we are not about to solve here) is that the value 1 is indeed eventually reached no matter what positive integer value the procedure starts with.

The program shown in figure 3.1 uses two registers, x and y, and is written in the format "line number, increment, next instruction" for the incrementing instructions, and "line number, decrement, next instruction if decrement succeeded, next instruction if decrement failed (because register is already 0)" for the decrementing instructions.

For our proof, we will need to use programs with a few specific properties. The register machine programs that we use in our proof will need to be programs that start with 0 in all the registers. If a program needs to start with a nonzero value in some register, it can

simply increment the register to the desired value at the beginning of the program. Any "input" to the program is thus treated as part of the program itself. Another feature of the programs used in the proof will be that they should have 0 in all the registers whenever they halt. If a program might halt with nonzero registers, we can simply attach loops to the end of the program to decrement all the registers back down to zero before halting. We will also require the program instructions never to point to themselves as the next instruction. If a program needs to execute a single line repeatedly, we can simply duplicate the line and let the execution go back and forth between the two lines. Thus, although our proof will only address the halting problem for programs of a certain form, we see that any program can easily be converted to the required form, so if we can solve the halting problem for programs of this form, then we can solve the halting problem for any $\langle \text{program}, \text{input} \rangle$ pair.

3.1.2.4 The Graph Corresponding to P's Execution

Here we simply define what the network corresponding to P's execution should look like.

The network will be built using many copies of a single relation on three variables, and the network will have a single dangling edge. (This edge is where it will reject a value.)

The building-block relation, X, will be a relation on three values, which we will call A, B, and C. So three edge-ends will meet at each vertex in the network. The relation at each vertex knows which edge is which—it knows which edge's value will be treated as A, which edge's value will be treated as B, and which edge's value will be treated as C. Of course, it may be the case that at one end, an edge's value will be treated as the A value in one relation, while at the other end, it will be treated as the C value in another relation. It may even be the case that an edge loops from a vertex back to the same vertex, so its value might be used as both the A value and the C value of a single particular relation. Such edges are called *self-loops*. (If for some reason self-loops are unacceptable, parts of the proof become much more complicated, but the nature of the proof does not change.)

Where an edge is treated by a vertex as the A (or B, or C) value for that relation, we will say that that edge is the *A-connection* (or *B-connection*, or *C-connection*) for that relation.

The main feature of the network corresponding to P's execution will be a backbone consisting of an A-C chain of relations. What we mean by this is that the dangling edge will

be the A-connection of the first relation in the backbone, and that relation's C-connection will then be the A-connection of the next relation in the backbone, and so on, until finally the backbone ends with a self-loop from the C-connection back to the B-connection of the final relation. In figure 3.2, the relations along the bottom form the backbone.

Each relation along the backbone will correspond to one step in the execution of P. All that remains to be specified is the B-connections of all the relations in the backbone (except the final one). We refer to these B-connections as the *hairs* on the backbone. The hairs will represent the values of the various registers.

Since all the registers start and end at zero, it must be possible to pair up the increment instructions with the decrement instructions for any given register, so that each increment instruction is paired with a later decrement instruction for the same register. (The decrement instructions which fail due to the register already being 0 will not participate in this pairing.) In the network corresponding to P's execution, this pairing of instructions is represented by simply connecting their hairs. So if the 100th instruction increments register y and is paired with the 200th instruction (which decrements y), then an edge will connect the 100th relation in the backbone to the 200th relation in the backbone, and it will be the B-connection of both of them.

The only thing left to specify is what happens with the hairs for decrement instructions that fail because the register is already 0. For each such instruction, we will add a new vertex that has an A-C self-loop, and we will connect the instruction's vertex (which is in the backbone) to this new vertex (which is not in the backbone) with an edge that is the B-connection of both relations.

This completes our specification of the network corresponding to the execution of P. As a very simple example, figure 3.2 shows the network corresponding to the execution of the register machine program of figure 3.1, with registers started at zero.

The execution of that program, with the registers started at zero, progresses through the following sequence of lines: $(1, 4, 6, 5, 4, 6, 7, \mathit{halt})$. Five of the seven executed instructions are decrement instructions that fail because the registers are already 0. These cases are easily spotted in the network structure due to the "parking meter" relation planted atop each one (named for its visual appearance in this diagram).

If we imagine that all the hairs of backbone relations for instructions manipulating the first register are colored green, then we see that the number of green hairs passing

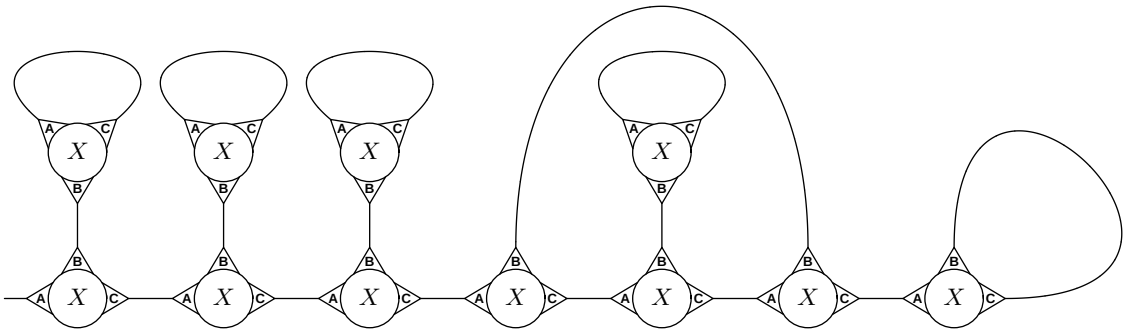


Figure 3.2: The network corresponding to the register machine program of figure 3.1, started with zero in both registers.

over an edge of the backbone gives the value that is in the first register at that stage of running the program. In particular, a green parking meter post cannot occur under a green increment/decrement pair hair, because that would mean that a decrement failed when the register was not in fact 0, which does not happen.

Now that we have a reasonable understanding of the network corresponding to P's execution, let us look at exactly what relation on A, B, and C we can use so that the requirement (of rejecting a particular value at the dangling edge at the start of the backbone) will force the network to be the one corresponding to P's execution.

3.1.2.5 The Construction of the Relation X

The easiest way to explain X's construction is just to give it, and then show why it works. To really understand it, you will probably have to get out your pencil and convince yourself of why it does what we say it does, as we walk through it. The tuples of X are given in figure 3.3. The finite alphabet consists of all the symbols appearing in the table, namely:

$$\{e_1, e_2, e_3, P_2, ..., \#_1, \#_1^x, \#_1^y, \#_2, ..., \bowtie, \triangledown, \blacktriangledown, \frown, ...\} \ .$$

Despite the subscripts, these symbols are not variables; they are values that a variable might have. The subscripts merely serve to help organize them. Only i and k (appearing in figure 3.3 as superscripts and subscripts to the # symbol) need to be substituted for to get actual symbols: i with a register name, and k with a line number of the program.

The one-tuple to be rejected is $\langle \#_1 \rangle$. So the network must have exactly one dangling

A	B	C
e_1	e_1	$\#_1$
e_1	$\#_1$	e_1
e_1	e_1	e_1
$\#_1$	e_2	P_2
P_2	e_2	P_2
P_2	e_2	Q_2
e_2	Q_2	e_2
e_2	e_2	Q_2
P_2	Q_2	P_2
$\#_1$	Q_2	P_2
$\#_1$	e_2	Q_2
e_2	e_2	e_2
$\#_1$	e_3	R_3
R_3	e_3	R_3
R_3	S_3	T_3
$\#_1$	S_3	T_3
T_3	e_3	T_3
T_3	U_3	U_3
e_3	e_3	e_3
S_3	e_3	e_3
e_3	e_3	S_3
e_3	S_3	V_3
e_3	e_3	V_3
e_3	V_3	e_3
V_3	e_3	e_3

A	B	C
$\bowtie$	$\triangledown$	$\bowtie$
$\bowtie$	$\blacktriangledown$	$\bowtie$
$\#_{k:inc}$	$\frown$	$\#_{k.next}$
$\#_{k:dec}$	$\frown$	$\#_{k.next_{\neq 0}}$
$\#_{k:dec}$	$\triangledown$	$\#_{k.next_{=0}}$
$\#^i_{k:inc}$	$\frown$	$\#^i_{k.next}$
$\#^i_{k:dec}$	$\frown$	$\#^i_{k.next_{\neq 0}}$
$\#^i_{k:dec}$	$\triangledown$	$\#^i_{k.next_{=0}}$
$\square$	$\frown$	$\boxdot$
$\boxdot$	$\frown$	$\square$
$\square$	$\triangledown$	$\boxdot$
$\boxdot$	$\triangledown$	$\square$
$\square$	!	$\bigcirc$
$\boxdot$	!	$\bigcirc$
$\bigcirc$	$\frown$	$\odot$
$\odot$	$\frown$	$\bigcirc$
$\bigcirc$	$\triangledown$	$\odot$
$\odot$	$\triangledown$	$\bigcirc$
$\bigcirc$	$\circlearrowleft$	$\circlearrowleft$
$\odot$	$\circlearrowleft$	$\circlearrowleft$
$\#_{k:inc}$	$\blacktriangledown$	$\bigcirc$
$\#_{k:inc/dec}$	$\circlearrowleft$	$\circlearrowleft$
$\#_{k:halt}$	$\frown$	$\bigcirc$
$\#_{k:halt}$	$\triangledown$	$\bigcirc$
$\#_{k:dec}$	!	$\square$
$\#_{k:inc\ r_i}$	!	$\#^i_{k.next}$
$\#^i_{k:inc}$	!	$\bigcirc$
$\#^i_{k:dec\ r_{j\neq i}}$	!	$\bigcirc$
$\#^i_{k:dec\ r_{i=i}}$	$\blacktriangledown$	$\square$

A	B	C
$\#_{k:inc\ r_i}$	!!	$\#^i_{k.next}$
$\#^i_{k:inc\ r_{i=i}}$	!	$\pentagon$
$\pentagon$	$\frown$	$\fivestar$
$\fivestar$	$\frown$	$\pentagon$
$\pentagon$	$\triangledown$	$\fivestar$
$\fivestar$	$\triangledown$	$\pentagon$
$\pentagon$	!!	$\square$
$\fivestar$	!!	$\square$

Figure 3.3: The triples forming relation X, based on a given register machine program.

edge, and we want to know if there is such a network that is unsatisfiable when that edge has the value $\#_1$.

The first three triples shown in figure 3.3 force the edge dangling out of the network to be the A edge of the relation it is dangling from. For if it were not the A edge, the first three triples shown above would allow it to have the value $\#_1$, and every other edge in the network could have the value e_1, and thus $\#_1$ would be acceptable, even though it is supposed to be rejected.

The next nine triples shown above for X force the A-C chain to end with a self-loop from the final C to the B of the same relation. In any other case, $\#_1$ will be acceptable with the edges in the A-C chain having value P_2, the final edge from the C at the end of the chain (which, being the end of the chain, must go to a non-A connection) having value Q_2, and all other edges in the network having value e_2.

The next nine triples shown force the B connections along the backbone to either connect to another relation on the backbone, or to connect to the B connection of a relation not on the backbone.

The final four triples in the first column work together with the previous nine to place further restrictions when a B on the backbone is connected to a B not on the backbone: In this case, the relation not on the backbone must have an A-C self-loop, since otherwise its C connection could have the value V_3 while every other edge (including its A connection) has value e_3.

So the first column forces the network to consist of a backbone with hairs either doubly-connected or leading to parking meters. So the topology looks like *some* program, but we have yet to make sure it is the *right* program.

We now move to the second column of triples.

Here, the first two triples are the only ones, of all the remaining triples, that have the same value for the A and C connections, so one of them must be used on each parking meter in any remaining solution.

The next three triples actually represent many triples, based on the program P. The expression $\#_{k:inc}$ represents a different symbol (namely $\#_k$) for each line k of the program P that is an increment instruction. The "*inc*" is just for our reference in knowing what triples should be created based on this "proto-triple." The expression $\#_{k.next}$ represents the value $\#_j$ for that j which the program indicates is the next instruction to be executed

after the increment instruction at line k. The proto-triples for decrement instructions work similarly. All the triples indicated by these three proto-triples work together to allow an initial segment of the backbone to be filled with $\#_k$ values on each edge, with k progressing exactly as the current instruction progresses during the execution of P.

The next three proto-triples are exactly the same, except that they create even more triples: one not only for each transition in the program, but for each possible value of i, where i identifies one of the registers (but not the register's value). The value of i is independent of the instruction at line k. So if there are two registers, x and y, then these proto-triples would create triples that are superscripted with either the letter x or the letter y, but always the same superscript on both the A connection and the C connection. The purpose of these triples is that they can be used along some stretch of the backbone, thus following the progress of P's execution while "remembering" which of the registers needs to be coordinated between the left and right end of that backbone stretch.

The next six triples allow a square symbol to fill a stretch of the backbone, ending with a single hair with an exclamation mark, at which point a stretch of circle symbols will follow. The dot inside the square has no purpose except to make the A and C values be different for every triple, so as not to interfere with the purpose of the first two triples in the column.

The next six triples allow the stretch of circles to fill the final stretch of the backbone, ending with the C-B self-loop.

Up to here the column has been preparing some infrastructure without forcing any particular structure on the network, but from here on down, the triples use the infrastructure to actively enforce things. At the first error in the network representation of P's execution, they will take advantage of the error to allow $\langle \#_1 \rangle$ to be accepted.

The next proto-triple, for example, prevents increment instructions from being connected to parking meters. For if one is, then $\#_k$ instructions can be used leading up to that position, and circles can be used afterwards, and the network will accept $\langle \#_1 \rangle$. The filled triangle is used because it cannot mistakenly occur on a non-parking meter hair.

The next three proto-triples prevent the backbone from ending before the program does (the first proto-triple), or from ending after the program does (the second or third proto-triple).

The next proto-triple prevents a decrement hair from coming back down to the backbone after the decrement instruction. In other words, decrement hairs must have been produced

prior to the decrement instruction, which also implies they cannot have been produced by a previous decrement instruction, but only by a previous increment instruction.

The final four proto-triples force each increment to be matched to a decrement of the same register, and prevent parking meters from existing when the corresponding register is not zero.

Finally, the eight triples and proto-triples in the short final column of triples are optional, and merely enforce the increments and decrements to be matched like matching parentheses. The purpose of this is to detangle the network's hair into a unique form. If these triples are included, and there is a network that rejects $\langle \#_1 \rangle$, then that network is unique. If these triples are not included, then the network will generally not be unique. The proof will work either way.

It is not hard to see that the various groups of triples cannot combine in unforeseen ways to mistakenly accept $\langle \#_1 \rangle$ when each individual group would have rejected it. For example, the distinct indices on the symbols in the first table of triples prevent the groups of triples from being able to interact with each other (except for the third and fourth groups, which are designed to work together).

In summary, if the network is exactly the network corresponding to P's execution, then $\langle \#_1 \rangle$ will be rejected by the network, but if the network is anything else, then $\langle \#_1 \rangle$ will be accepted.

3.1.2.6 Proof Variants

Using a fixed relation

The main construction creates a different relation X for different programs. If we want to always use a single relation X, the natural idea is that X should be the relation given by the above construction for a fixed register machine program P that is itself universal. Then, the initial values of the registers, when the machine is started, can contain the "real" program for which we would like to know whether it halts or not.

However, getting the initial values of the registers into the network is not trivial. The initial register values need to be incorporated into the network by using many lengthy tuples in $\mathcal{T}$ instead of just a single one-tuple. The idea behind the many tuples is that they encode all the values, for the dangling backbone and hair edges, that could have been used in the

original construction (which would have had a single dangling edge and started with many increment instructions) to detect one of the many possible types of errors. This moves the initial condition from the triples that form X into the tuples of $\mathcal{T}$. So then, using a universal register machine program P completes the job.

The reader will have noticed that we have failed to simplify $\mathcal{T}$ down to a single tuple for this case. It is an open question whether a fixed relation X can be constructed for which the question "Given a tuple T, does there exist a network of X's which rejects T?" is undecidable.

Boolean Alphabet

With a boolean alphabet, if we want to use roughly the same construction idea, we need to have multiple edges between relations. This leads to a host of new ways that the network might fail to accurately represent the running of the program. In particular, the multiple edges might not be properly aligned with each other (regarding their orderings at the vertices), or perhaps might not even all go to the same other vertex. Many gadgets are required to deal with these troubles; we will not list them here. The main idea that allows them to succeed is to use low-weight tuples and high-weight tuples for enforcing multiple edge alignment, while using medium-weight tuples to encode the variable values of the previous construction.

Infinite Graphs

The main difference in the proof, when infinite networks are allowed, is that now the execution of a non-halting program might be accurately modeled by the topology of an infinite network. However, whereas for a finite network, having an accurate topology led to rejection of the proscribed tuple T, in an infinite network, T can be accepted even if the topology has no errors, since all edges can take the values they would take if there were going to be an error farther down the line. Thus, it is still the case that a network rejecting T exists if and only if the program halts.

3.2 A Natural Class of Nondeterministic Machines

In this section we will discuss a class of computing machines that deserves to be better known, due both to its repeated appearance in many forms and to its simplicity and elegance. We will present it in several forms, each elegant on its own.

3.2.1 Vector Addition Systems (VASs)

Vector addition systems (VASs) were developed and studied by Karp and Miller [KM69] for analyzing asynchronous parallel processes. A vector addition system is a nondeterministic walk through an m dimensional integer lattice, where each step must be one of d given vectors $\vec{V}_\alpha \in \mathbb{Z}^m$, and each point in the walk must have no negative coordinates.

Whether it is possible to walk from a point x to a point y (the "reachability question") is known to be decidable [May81]. It is also decidable whether it is possible for a walk to enter a linearly-defined subregion [ST77] – a special case is whether the i^{th} component of the point ever becomes non-zero (the "producibility question"). Although the producibility question and the reachability question are both decidable, the producibility question is in general easier to decide than the reachability question.

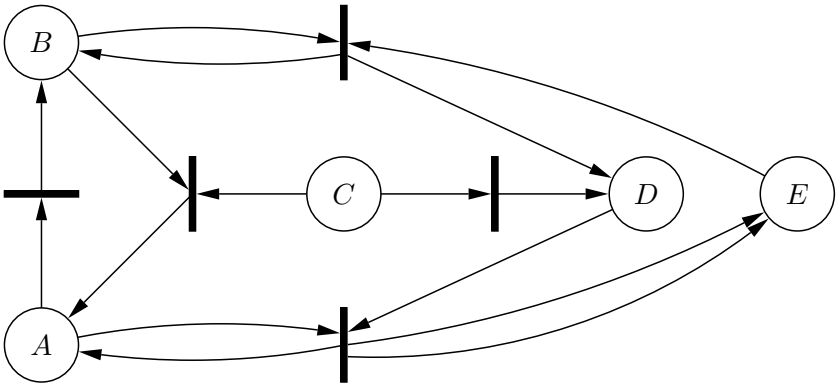
3.2.2 Petri Nets

An example of a Petri net [Pet62], is shown in figure 3.4(b). In this model a network consists of a directed bipartite graph, with the partitions called *places* and *transitions*. The places are shown as circles, and the transitions are shown as black bars with arrows entering and exiting on the sides. There is no difference between the two sides of the bar. The state consists of a non-negative number of *tokens* at each place, and a new state is achieved by the *firing* of a transition. When a transition fires, it consumes one token from the incident place for each incoming edge, and produces one token at the incident place for each outgoing edge. Thus, a transition is *enabled* only if there are enough tokens in the input places. In any given state, there are typically many transitions that could fire. Which one fires first is intentionally left unspecified; the theory of Petri nets address exactly the question of how to analyze asynchronous events.

a)

$$\begin{array}{rcl} A & \rightarrow & B \\ C & \rightarrow & D \\ B+C & \rightarrow & A \end{array} \qquad \begin{array}{rcl} A+D & \rightarrow & A+2E \\ B+E & \rightarrow & B+D \end{array}$$

b)



c)

	A	B	C	D	E	F	G	
$\langle$	-1	1	0	0	0	0	0	$\rangle$
$\langle$	0	0	-1	1	0	0	0	$\rangle$
$\langle$	1	-1	-1	0	0	0	0	$\rangle$
$\langle$	-1	0	0	-1	0	1	0	$\rangle$
$\langle$	1	0	0	0	2	-1	0	$\rangle$
$\langle$	0	-1	0	0	-1	0	1	$\rangle$
$\langle$	0	1	0	1	0	0	-1	$\rangle$

d)

$$\frac{3}{2} \quad \frac{7}{5} \quad \frac{2}{15} \quad \frac{13}{14} \quad \frac{242}{13} \quad \frac{17}{33} \quad \frac{21}{17}$$

Figure 3.4: Four representations of the same program for computing powers of 2. (a) A chemical reaction network. Starting with 1 A and n C's, the maximum number of D's that can be produced is 2^n. (b) A Petri net. Each circle corresponds to a *place* (like a molecular species in (a)), and each black bar corresponds to a *transition* (like a reaction in (a)). (c) A Vector Addition System. Note that dimensions F and G must be added to the Vector Addition System to capture the two reactions that are catalyzed by A and B. (d) A Fractran program. The numerators correspond to the reaction products, and the denominators correspond to the reactants. The first seven prime numbers are used here in correspondence to the letters A through G in the other examples. As in the previous example, F (13) and G (17) must be introduced, here to avoid unreduced fractions for the catalytic reactions.

3.2.3 Unordered Fractran

Another model that turns out to be related is a lesser known model called "Fractran" [Con72], shown by Conway to be Turing universal. A Fractran program consists of an ordered list of rational numbers (as in figure 3.4(d)). Execution is deterministic: starting with a positive integer n as input, we find the first fraction on the list that produces an integer when multiplied by n, and this product becomes the new number n'. This process is iterated forever unless it halts due to no fraction resulting in an integer. Conway showed that any register machine program can be converted directly into a Fractran program: representing every integer in fully factored form, $n = p_1^{a_1} \cdots p_m^{a_m}$, where p_i is the i^{th} prime, the exponents $a_1 \ldots a_k$ store the contents of the k registers, while other distinct primes p_h are each present if and only if the register machine is in state h. The denominator of each Fractran fraction conditions execution on being in state h and – if the operation is to decrement the register – on having a non-empty register. The numerator provides for increments and sets the new state. Since register machines are Turing-universal, it follows that Fractran is also universal. However, it is worth noticing that since these systems only allow increment and decrement operations (thus storing all state in unary), they entail exponential slowdowns compared to Turing machines.

Unordered Fractran is the same, except that there is no ordering among the fractions: At each step, any fraction in the list may be used, so long as the resulting product is an integer. Unordered Fractran has not previously been considered, to our knowledge.

3.2.4 Broken Register Machines

Here we will examine a variant of register machines which we call broken register machines (which, to our knowledge, have not previously been considered). These are the same as the register machines of section 3.1.2.3 except that decrement instructions are allowed to fail (nondeterministically) even if the register is non-zero. (If the register is zero, the instruction is of course forced to fail as before.) We will show in section 3.2.6 that broken register machines turn out to be equivalent to unordered Fractran, Petri nets and VASs (and thus to chemical reaction networks as well), although the equivalence is not quite as direct as for the other systems. The nature of the equivalence between broken register machines and chemical reaction networks, combined with the fact that broken register machines only need

to decide between two options at a time, enables one to show that in fact only two priority levels are necessary for a chemical reaction network or Petri net to be universal.

3.2.5 Chemical Reaction Networks

Chemical reaction networks are a standard model of chemical behavior [Gil77, RML93, ARM98, GP98, GB00], used in situations, such as within a cell, where it makes sense to keep track of the exact number present of each species of molecule.

Chemical reaction networks are among the most fundamental models used in chemistry, biochemistry, and most recently, computational biology. Traditionally, analysis has focused on *mass action* kinetics, where reactions are assumed to involve sufficiently many molecules that the state of the system can be accurately represented by continuous molecular concentrations with the dynamics given by deterministic differential equations. However, analyzing the kinetics of small-scale chemical processes involving a finite number of molecules, such as occurs within cells, requires stochastic dynamics that explicitly track the exact number of each molecular species [Gil77, ARM98, GB00].

3.2.5.1 The Standard Model

A chemical reaction network is defined as a finite set of d reactions acting on a finite number m of species. Each reaction α is defined as a vector of non-negative integers specifying the stoichiometry of the reactants, $(r_{\alpha,1}, \ldots, r_{\alpha,m})$, together with another vector of non-negative integers specifying the stoichiometry of the products, $(p_{\alpha,1}, \ldots, p_{\alpha,m})$. The stoichiometry is the non-negative number of copies of each species required for the reaction to take place, or produced when the reaction does take place. We will use capital letters to refer to various species and we will use standard chemical notation to describe reactions. So for example, the reaction $A + D \rightarrow A + 2E$ consumes 1 molecule of species A and 1 molecule of species D and produces 1 molecule of species A and 2 molecules of species E (see figure 3.4). In this reaction, A acts catalytically because it must be present for the reaction to occur, but it is not itself affected by the reaction. We will use the word *catalyst* to mean a species that occurs on both the left and right side of a single reaction. (In chemistry, catalysis can involve a series of reactions or intermediate states, but we will restrict our meaning to "instantaneous catalysts.")

The state of the network is defined as a vector of non-negative integers specifying the quantities present of each species, $\mathcal{A} = (q_1, \ldots, q_m)$. A reaction α is possible in state $\mathcal{A}$ only if there are enough reactants present, i.e., $\forall i, q_i \geq r_{\alpha,i}$. When reaction α occurs in state $\mathcal{A}$, the reactant molecules are used up and the products are produced. The new state is $\mathcal{B} = \mathcal{A} * \alpha = (q_1 - r_{\alpha,1} + p_{\alpha,1}, \ldots, q_m - r_{\alpha,m} + p_{\alpha,m})$. We write $\mathcal{A} \xrightarrow{C} \mathcal{B}$ if there is some reaction in chemical reaction network C that can change $\mathcal{A}$ to $\mathcal{B}$; we write $\xrightarrow{C*}$ for the reflexive transitive closure of $\xrightarrow{C}$. We write $\Pr[\mathcal{A} \xrightarrow{C} \mathcal{B}]$ to indicate the probability that, given that the state is initially $\mathcal{A}$, the next reaction will transition to the state $\mathcal{B}$. $\Pr[\mathcal{A} \xrightarrow{C*} \mathcal{B}]$ refers to the probability that at *some* time in the future, the system will be in state $\mathcal{B}$.

Every reaction α has an associated rate constant $k_\alpha > 0$. The rate of every reaction α is proportional to the concentrations (number of molecules present) of each reactant, with the constant of proportionality being given by the rate constant k_α. Specifically, given a volume V, for any state $\mathcal{A} = (q_1, \ldots, q_m)$, the rate of reaction α in that state is

$$\rho_\alpha(\mathcal{A}) = k_\alpha V \prod_{i=1}^{m} \frac{(q_i)^{\underline{r_{\alpha,i}}}}{V^{r_{\alpha,i}}} \quad \text{where} \quad q^{\underline{r}} = \frac{q!}{(q-r)!} = \overbrace{q(q-1)\cdots(q-r+1)}^{r \text{ terms}} \ . \tag{3.1}$$

Since the solution is assumed to be well-stirred, the time until a particular reaction α occurs in state $\mathcal{A}$ is an exponentially distributed random variable with the rate parameter $\rho_\alpha(\mathcal{A})$; i.e., the dynamics of a stochastic chemical reaction network is a continuous-time Markov process.

Note that

$$\Pr[\mathcal{A} \xrightarrow{C} \mathcal{B}] = \frac{\rho_{\mathcal{A}\to\mathcal{B}}}{\rho_{\mathcal{A}}^{tot}} \tag{3.2}$$

$$\text{where} \ \ \rho_{\mathcal{A}\to\mathcal{B}} = \sum_{\alpha \text{ s.t. } \mathcal{A}*\alpha=\mathcal{B}} \rho_\alpha(\mathcal{A}) \ \ \text{and} \ \ \rho_{\mathcal{A}}^{tot} = \sum_{\mathcal{B}} \rho_{\mathcal{A}\to\mathcal{B}} \ .$$

The average time for a step $\mathcal{A} \to \mathcal{B}$ to occur is $1/\rho_{\mathcal{A}}^{tot}$, and the average time for a sequence of steps is simply the sum of the average times for each step.

Since this chapter addresses the limits of computability by chemical reaction networks, it behooves us to examine whether the model retains its physical plausibility in the limits we consider. An immediate concern is that, while we will consider chemical reaction networks that produce arbitrarily large numbers of molecules, it is impossible that so many molecules

can fit within a pre-determined volume. Thus we recognize that the reaction volume V must change with the total number of molecules present, which in turn will slow down all reactions involving more than one reactant molecule. (Practical experiments may thus be effectively limited to logspace computations.) Choosing V to scale proportionally with the total number of molecules present (of any form) results in a model appropriate for analysis of reaction times. Note, however, that for any chemical reaction network in which every reaction involves exactly the same number of reactants, the transition probabilities $\Pr[\mathcal{A} \xrightarrow{C} \mathcal{B}]$ are *independent* of the volume. For all the positive results discussed later in this chapter, we can design chemical reaction networks involving exactly two reactants in every reaction, and therefore volume can for the most part be ignored. A remaining concern—which we cannot satisfactorily address—is that the assumption of a well-stirred reaction may become less tenable for large volumes. However, the well-stirred assumption seems to be a geometrical weakness regarding embedding in our 3-D universe, comparable for example to the practical issue in boolean circuits that wires may need to be arbitrarily long without having any chance of transmission error. As such, it seems the model remains useful for theoretical study of the system.

A second immediate concern is that the reactions we consider are of a very general form, including reactions such as $A \to A + B$ that seem to violate the conservation of energy and mass and the intrinsic reversibility of elementary chemical steps. This is true, but reactions such as these are necessary for modeling biochemical circuits within the cell, such as genetic regulatory networks that control the production of mRNA molecules (transcription) and of protein molecules (translation). Thus, our models intrinsically assume that energy and mass are available in the form of chemical fuel (analogous to ATP, activated nucleotides, and amino acids) that is sufficient to drive reactions irreversibly and to allow the creation of new molecules. Thus, our reaction volume can be envisioned as a two-dimensional puddle that grows and shrinks as it adsorbs fuel from and releases waste to a three-dimensional environment. This is very similar in spirit to computational models such as Turing Machines and Stack Machines that add resources (tape or stack space) as they are needed.

3.2.5.2 Other Approaches to Chemical Computation

It is worth noting that several other flavors of chemical system have been shown to be Turing universal. Bennett [Ben82] sketched a set of hypothetical enzymes that will modify a information-bearing polymer (such as DNA) so as to exactly and efficiently simulate a Turing machine. In fact, he even analyzed the amount of energy required per computational step and argued that if the reactions are made chemically reversible and biased only slightly in the favorable direction, an arbitrarily small amount of energy per computational step can be achieved. Since then, there have been many more formal works proving that biochemical reactions that act on polymers can perform Turing-universal computation (e.g., [Pau95]). In all of these studies, unlike the work presented here, there are an infinite number of distinct chemical species (polymers with different lengths and different sequences) and thus, formally, an infinite number of distinct chemical reactions. These reactions, of course, can be represented finitely using an augmented notation (e.g., "cut the polymer in the middle of any ATTGCAAT subsequence"), but they certainly do not qualify as finite chemical reaction networks.

A second common way of achieving Turing universality is through compartmentalization. By having a potentially unbounded number of spatially separate compartments, each compartment can implement a finite state machine and store a fixed amount of information. Communication between compartments can be achieved by diffision of specific species, or by explicit transfer reactions. This is exploited for example in the Chemical Abstract Machine [BB90].

Regarding finite stochastic chemical reaction networks, there have been previous reports in the literature claiming to show that mass action chemical kinetics is Turing universal [HWR91, Mag97], but these actually showed only that individual boolean logic gates can be constructed, and that they can be connected together in a circuit. In those constructions, the number of species is at least linear in the number of gates. This provides for efficient computation but is a non-uniform model, i.e., any individual construction can only handle a finite number of inputs.

Indeed, a natural relation to boolean circuits has led many people to expect similar computational power. For example, given a circuit built from NAND gates, we can construct a corresponding chemical reaction network by replacing each gate $x_k = x_i \text{ NAND } x_j$ with

the four reactions $A_i + A_j \rightarrow A_i + A_j + B_k$, $A_i + B_j \rightarrow A_i + B_j + B_k$, $B_i + A_j \rightarrow B_i + A_j + B_k$, $B_i + B_j \rightarrow B_i + B_j + A_k$. The presence of a single A_i molecule represents that $x_i = 0$, the presence of a single B_i molecule represents that $x_i = 1$, and the presence of neither indicates that x_i has not yet been computed. If the circuit has only feed-forward dependencies, it is easy to see that if one starts with a single A or B molecule for each input variable, then with probability 1 the correct species will eventually be produced for each output variable. In this sense, a chemical reaction network can deterministically compute the same function as the boolean circuit, despite the uncontrollable order in which reactions occur. Note that in this particular network, the specific rate constants can affect the speed with which the computation occurs, but have no effect on the final state.

In contrast with the limited (finite state) computational power of boolean circuits, chemical reaction networks do not have finite state spaces: there may potentially be an unbounded number of molecules of any given species. As even minimal finite-state machinery coupled with unbounded memory tends to allow for Turing-universal computation, one might speculate that chemical reaction networks should be universal. We show that this is the case in section 3.4.2, which means that predicting the long-term behavior of a given chemical reaction network is undecidable. Our results are the first to demonstrate the possibility of uniform, Turing universal computation for chemical reaction networks.

3.2.5.3 Intuition for Proving Universality

If it were possible to prioritize the reactions in a chemical reaction network, then by analogy to the ordered fractions in Fractran, this would establish the Turing-universality of chemical reaction networks. (This result is also well known in the field of Petri nets, and our analysis of register machines shows that in fact only two distinct priority levels are necessary.)

By giving higher-priority reactions vastly faster rate constants k_α, we can approximate a priority list: almost surely, of all reactions for which all reactants are present in sufficient number, a reaction with a much faster rate will occur first. However, "almost surely" turns out not to be good enough for a couple of reasons. First, there is a non-zero probability of the slow reaction happening at each step, and thus probability of correct prioritization falls exponentially with the number of steps. Second, the number of molecules of a given species can potentially exceed any bound, so the ordering of actual rates $\rho_\alpha(\mathcal{A})$ may eventually be

different from the specified ordering of rate constants k_α. Especially in light of the decidability results mentioned above, it is not surprising that this naive approach to achieving Turing universality with chemical reaction networks fails.

If there were some way to increase rate constants over time, this could solve these problems, but of course, rate constants cannot change. Another way to promote one reaction over another would be to give the preferred reaction some extra time to occur before the competing reaction has a chance to occur. This approach turns out to be workable, and it is not too hard to set up some reactions that produce a signal after some delay, where the delay depends on a particular concentration. We refer to such a set of reactions as a *clock*. An important technical point is that since the entire computation will consist of an unknown number of steps, the probability of error at any given step must be decreasing so that the sum of all the probabilities can remain small regardless of how long the computation winds up taking. To address this issue, the clock can at each step increase the concentration that controls its delay, so that the delays are progressively longer, and thus the probabilities of error are progressively smaller. Fortunately, it turns out that a simple polynomial slowdown in overall computation time is all that is required for making the total probability of error (over the entire course of the arbitrarily long computation) be small.

In section 3.4.2 we give a construction along these lines for simulating register machines with chemical reaction networks with only quadratic slowdown, and we prove that successful output will occur with fixed probability $1-\epsilon$ independent of the input and computation time. An initial number of "precision molecules" can be added to achieve any desired ϵ. Thus, tolerating a fixed, but arbitrarily low, probability that computation will result in an error, chemical reaction networks become Turing universal. In consequence, the probabilistic variants of the reachability and producibility questions are undecidable. (The reachability question asks if a particular target state will be reached. The producibility question asks if the target state or any superset of it will be reached.)

The simulation given in section 3.4.2 is relatively simple to understand, but its performance is limited by the fact that it is simulating a register machine, which is exponentially slower than a Turing Machine (in the space used by the Turing Machine), due to its unary representation of information. Can chemical reaction networks do better than this? It seems likely, and we list this among our open questions in section 3.4.2.3.

3.2.6 Equivalences

The correspondence between vector addition systems, chemical reaction networks, and Petri nets is direct. First consider chemical reactions in which no species occurs both on the left side (as a reactant) and on the right side (as a product) – i.e., reactions that have no *instantaneous catalysts*. When such a reaction α occurs, the state of the chemical reaction network, represented as a vector, changes by addition of the vector $\vec{p}_\alpha - \vec{r}_\alpha$. Thus the trajectory of states is a walk through $\mathbb{Z}^m$ wherein each step is any of d given vectors, subject to the inequalities requiring that the number of molecules of each species remain non-negative, thus restricting the walk to the non-negative orthant. Karp and Miller's decidability results for VASs [KM69] directly imply that the reachability and producibility questions for catalyst-free chemical reaction networks are both decidable. As a consequence, any chemical reaction network computation that is guaranteed to yield a unique final state cannot be Turing universal, since questions such as whether the YES output molecule or the NO output molecule will be produced are decidable. The restriction to catalyst-free reactions can easily be dispensed with: each catalytic reaction can be replaced by two new reactions involving a new molecular species (an "intermediate state" such as F or G in figure 3.4(c)), and then all reachability and producibility questions (not involving the new species) are identical for the catalyst-free and the catalyst-containing networks.

If a Petri net uses rate constants as in equation 3.1 for each transition (in which case the model is a type of stochastic Petri net), the model is formally identical to stochastic chemical reaction networks: each place corresponds to a molecular species (the number of tokens is the number of molecules) and each transition corresponds to a reaction [GP98].

To show that a broken register machine can simulate unordered Fractran, we write a program to do so. The program has a main loop where it tries multiplying by each fraction in turn. For every prime appearing as a factor in some numerator or denominator, the register machine has a register to keep track of that prime's exponent in the current integer state. To multiply by a fraction, therefore, we simply decrement registers according to the denominator, and increment registers according to the numerator. If any decrement fails, control is passed to a restoration section that increments any registers already decremented for this fraction (which we have decided not to use after all) back to their original value before trying the next fraction.

To show the other direction, that unordered Fractran can simulate a broken register machine, we simply use a distinct prime for each line of the broken register machine program, and a distinct prime for each register. Then every increment instruction can be converted directly into a fraction, and every decrement instruction can be converted into two fractions. (This is the same construction that allows ordered Fractran to simulate a non-broken register machine.)

Petri nets, chemical reaction networks, and VASs have historically been conceptually grouped with non-uniform models such as boolean circuits. However, in section 3.4.2 we will show that these models, when provided with rate constants, are in fact capable of uniform computation as well.

3.3 Functions: Decidable Either Way

After what we have seen with relations, it may occur to us that functions (gates) are almost always considered in a context where fan-out is available. (Quantum computation is a notable exception.) It has been known since the time of Post [Pos41] that, given a set of functions of boolean values, only a finite number of tests need to be done to know whether a particular target function can or cannot be implemented. It is natural to wonder, if fan-out is *not* available, might the implementability question become *un*decidable, as it did for relations?

First of all, we have to be clear about what we mean by "without fan-out" in the case of functions. As was the case with relations, we would like fan-out to be optionally available. From a feed-forward point of view, a fan-out node in a circuit is a device with one input and two outputs, and both outputs equal the input. So, we will be generous and expand the definition of "function" to allow multiple outputs. (If we do not do this, then all circuits must be trees, and it becomes difficult to implement anything at all, since in contrast with formulas, inputs cannot be used at more than one leaf of the tree.) We will sometimes call these functions "gates." We will define the outputs of a feed-forward circuit to be all of the output wires which have not been fed into some other gate, and the inputs are of course all the input wires which are not produced as the output of another gate. This means the capability of ignoring a value will be optionally available, as it was for relations: To ignore a wire, you need to be able to build a circuit that ignores it (perhaps by using a gate provided

expressly for this purpose). Of course one can also consider questions in the context of it being ok to ignore wires, which can help simplify some proofs.

This gives us an implementability question for feed-forward circuits that is comparable to the implementability question for relations. As with relations, the availability of fan-out makes the question easily decidable: Simply iteratively expand the set of implementable functions, starting with the inputs and the given functions. However, without fan-out available, the situation is not so clear. After the results of section 3.1.2, we wonder whether this question may be undecidable.

The answer to this is far from obvious, but after some pondering, I was finally able to prove that the question remains decidable even when fan-out is not present. But, as it turned out, someone else beat me to the proof—it was proven by Karp [KM69] (for vector addition systems), shortly before I was born. Now it is a classic result within the field of Petri nets [EN94].

In this section we will show that the implementability question for functions is equivalent to the models of the previous section. The relationship of those models to primitive recursive functions (which demonstrates that these questions are decidable) will be proven in section 3.4.1.

We will now show that the implementability question can be reduced to a reachability question for a chemical reaction network. The first idea is that we consider all possible inputs to the circuit simultaneously. Since we know what we are trying to implement, we know how many inputs there are, and what the possible values for each input are, and thus we know exactly how many distinct possible states the entire circuit can be in. For example, if there are five boolean inputs, then there are $2^5 = 32$ possible states for the circuit, and every wire in the circuit can have its behavior described by a vector of length 32, giving the value of that wire in each of the 32 possible situations the circuit might be in. In this example, the five inputs to the circuit would be described by the following vectors:

$$\langle 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0\rangle$$
$$\langle 1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0\rangle$$
$$\langle 1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0\rangle$$
$$\langle 1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0\rangle$$
$$\langle 1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0\rangle$$

The vector describing an output of a function is easily calculated from the vectors for

the inputs.

The corresponding chemical reaction network will be designed to have one species for each possible vector. (In the example above, there would be 2^{32} species.) Then, each function available in the implementability question is converted into a long list of chemical reactions: For each possible combination of input vectors to the function, we provide a chemical reaction which takes those species as reactants and produces the appropriate species (those corresponding to the correct outputs of the function for these inputs) as products. The starting state for the chemical reaction network is one molecule of each of the species used as inputs (in the example above, recalling that each vector is a species, the starting state would be the five listed vectors), and the target state for the reachability question is simply the corresponding set of output vector species for the target function in the implementability question. It is clear from the design that the target state is reachable in the chemical reaction network if and only if the target function is implementable in the implementability question.

Now we will show the other direction, that any reachability question for a chemical reaction network can be reduced to an implementability question for functions without fan-out.

The idea for this direction is do design some functions that can only be usefully combined by following exactly the reactions of the network. The alphabet of values used by the functions will consist of one symbol for each of the chemical species, plus an extra symbol "ϵ," which we will think of as an error symbol. There will be one function per reaction, plus one extra function. Each reaction will be converted into a function with as many inputs as reactants and as many outputs as products. For example, the reaction $A + 2B \to C + D$ would become a function with 3 inputs and 2 outputs, and the computation performed by the function is almost trivial: It outputs ϵ on every output, *unless* its inputs are $\langle A, B, B\rangle$, in which case it outputs $\langle C, D\rangle$. Other reactions are similarly converted. We also provide an extra function with two inputs and two outputs, which is the identity function, except that if *either* input is ϵ, then *both* outputs are ϵ. Otherwise the first output matches the first input, and the second output matches the second input. The purpose of this function is to allow the error symbol ϵ to spread, as we will see shortly.

The initial state and target state for the reachability question then become the inputs and outputs of the target function, and again every other possible input should lead to all

outputs being ϵ.

Any satisfactory solution to this implementability question clearly corresponds to a partially ordered sequence of reactions that demonstrates a positive answer to the reachability question. Conversely, any sequence of reactions reaching the target state of the reachability question can be directly converted into a circuit of functions that is *almost* guaranteed to implement the target function. The only potential problem is that if the input given to the circuit differs just slightly from the intended input, then some of the functions will still be getting exactly the inputs that were intended, and for some circuits, it may not be the case that *all* outputs are ϵ, but rather just some subset of them. It is for this reason that we supplied the extra "error propagating" function. If necessary, this function can be used many times at the end of a circuit ($2n$-3 times for a circuit with n outputs) to ensure that if any outputs are ϵ, then all outputs must be ϵ. Clearly the availability of this function will not otherwise affect the ability to simulate the sequence of reactions. Thus, the answer to the functional implementability question will match exactly the answer to the chemical reachability question.

3.4 Chemical Reaction Networks: Decidability $\Longleftrightarrow$ No Probabilities

The results in this section paint an interesting picture of the relationship between decidability and probability in chemical reaction networks. Given a chemical reaction network in some initial state, every state in the full state space has some probability (perhaps 0) of ever being entered. Questions about what the system might do correspond to questions about these probabilities.

In this section we will show that the question of whether a given target state has a positive probability of occurring (vs. 0 probability) is decidable, while the question of whether a given state has high probability of occurring (vs. a low probability) is undecidable. Thus, the price of Turing-universal computational power is that one must be willing to accept a minute chance that the answer might be wrong. We believe this to be the first result of this nature.

3.4.1 No Probabilities $\Longrightarrow$ Primitive Recursive

This section will present a primitive recursive bound on the depth of the tree of reachable states for a chemical reaction network. The "degree" of the primitive recursive bound (what Ackermann [Ack28] called the "type") is on the order of the number of species in the chemical reaction network.

My original proof was almost identical to Karp's, merely showing finiteness of the tree, but then I expanded it to analyze the running time more precisely. Where Karp invokes König's Infinity Lemma to show finiteness, we will construct an explicit bound that shows not only finiteness, but primitive recursiveness. I believe the proof presented here is the first direct proof of this relationship (it had previously been proven by way of theorems regarding Diophantine equations).

It has long been known that certain questions about whether a Petri net "might do X" are decidable, where typical values of X are, in the language of chemical reaction networks, "keep having reactions forever" or "grow without bound" or "reach a certain state" or "produce at least some given quantities of given species" [KM69, May81, EN94]. These results carry over directly to chemical reaction networks so long as the question does not ask about the *probability* of X happening, but only about the *possibility* of it happening (i.e., only about whether the probability of X is zero vs. non-zero).

As mentioned in section 3.2.6, any chemical reaction network computation that is guaranteed to yield a unique final state can only implement decidable decision problems. Thus, for questions about the output of a chemical reaction network (given by some final quantity of the output species) to have any hope of being undecidable, the output *must* be probabilistic in nature. Section 3.4.2 addresses exactly such probabilistic questions.

Although the questions of possibility listed above are known to be decidable, their complexity is sometimes not so clear. The complexity of the problem for X="grow without bound" is known to be doubly exponential [RY86], but the complexity of the problem for X="reach a certain state" has been an open problem for decades [EN94].

Even though double exponential complexity sounds quite complex, the complexity of these types of problems can in fact be far greater. Some suspect that the reachability problem (i.e., X="reach a certain state") may have complexity comparable to primitive recursive functions, which are so powerful that few natural non-primitive recursive functions

are known.

Here we present examples of "might do X" problems whose complexity does exactly match the power of primitive recursive functions. Specifically, if X = "have a molecule of S_1 present when attaining the maximum possible amount of S_2," or X = "have a molecule of S_1 present after taking the longest possible sequence (over all possible sequences) of reactions." These questions are equivalent in power to primitive recursively defined predicates, where the number of primitive recursive functions used to recursively build up the predicate is on the order of the number of molecular species in the chemical reaction network, and the input to the predicate corresponds to the initial state of the chemical reaction network.

To show that such questions are no more powerful than primitive recursive functions, we show that for any chemical reaction network, it is possible to define a primitive recursive function which can return the amount of S_1 that is produced by whichever sequence of reactions leads to the largest possible amount of S_2. Our proof, while far from straightforward, is much simpler than previous similar proofs (which used results on bounds for solutions to bounded versions of Hilbert's tenth problem), since it gives an explicitly primitive recursive formula bounding the size of the tree of all possible runs of the chemical reaction network. The bulk of the proof lies in defining this bounding function and proving that it indeed bounds the depth of the tree. This bound enables the definition of a primitive recursive function which analyzes the entire tree, explicitly finding the run with the largest amount of S_2 and returning the corresponding amount of S_1.

The other direction is much simpler. Here we show by construction that given any primitive recursive function f, a chemical reaction network can be designed so that the state with the maximal amount of S_2 will have exactly $f(n)$ molecules of S_1, where n is given as input by being the number of molecules of an input species S_3 (along with a fixed number of molecules of other species) when the system is started. The chemical reaction network is designed to first compute an upper bound B on the running time needed to compute f with a register machine. Such a bound can be computed by simply calculating a primitive recursive function known to majorize the running time of f, and it is well known [Ack28] how such a function can be constructed based on a simple structural examination of f. The chemical reaction network then simulates a broken register machine (that is, a register machine whose decrement instructions may fail nondeterministically even when the register is not empty) for B steps, which we know is more than enough time for the register

machine program to finish. After *each* of the B steps (with the `halt` instruction changed to a `nop` (no operation) instruction so that B steps can indeed occur), the chemical reaction network passes control to a "subroutine" which doubles the amount of S_2 (actually, all it can do is allow the amount of S_2 to at most double, but that is good enough). In addition, every successful decrement of a register produces an extra molecule of S_2. Thus, S_2 winds up being a large integer whose binary digits are a record of the times at which decrement instructions successfully decremented a register. This means that any run with the largest possible amount of S_2 must have always succeeded at decrementing whenever possible. In other words, it emulated the register machine in the correct, non-broken way. Thus we can be sure that in this run, S_1 has been computed correctly. Since the bulk of the time is consumed by doubling S_2, the correct run is also the longest possible sequence of reactions for the chemical reaction network, and the same remains true if we append a "clean up" routine to the end of the computation, that clears away the large quantity of S_2.

Thus primitive recursive functions are in perfect correspondence with questions of the form "How many molecules of S_1 will there be if a chemical reaction network produces the maximal amount of S_2?" or "How many molecules of S_1 will there be if the chemical reaction network takes the longest possible sequence of reactions?" So although questions of possibility in chemical reaction networks are decidable, we have shown here that in some ways they have the full power of primitive recursive functions.

3.4.1.1 The Algorithm

In this section we will present an algorithm for finding which species can be produced and which cannot. That is, it will find out whether any reachable states have non-zero levels of any species of interest. In fact, it will do slightly more: For any given set of molecule quantities (such as $(10A, 3B, \ldots)$), the algorithm can find out whether or not it is possible to reach any state that has at least these levels of these species.

The algorithm is simply to search through the full tree of all possible reaction sequences, using a couple of simple tricks to try to avoid getting stuck in infinite loops.

If state $\mathcal{B}$ has at least as many molecules of each species as state $\mathcal{A}$ does, then we will say that $\mathcal{B} \geq \mathcal{A}$. On the other hand, if $\mathcal{B}$ has more of some species and less of others than $\mathcal{A}$ has, we say that $\mathcal{B}$ and $\mathcal{A}$ are incomparable: $\mathcal{A} \not\geq \mathcal{B}$ and $\mathcal{B} \not\geq \mathcal{A}$.

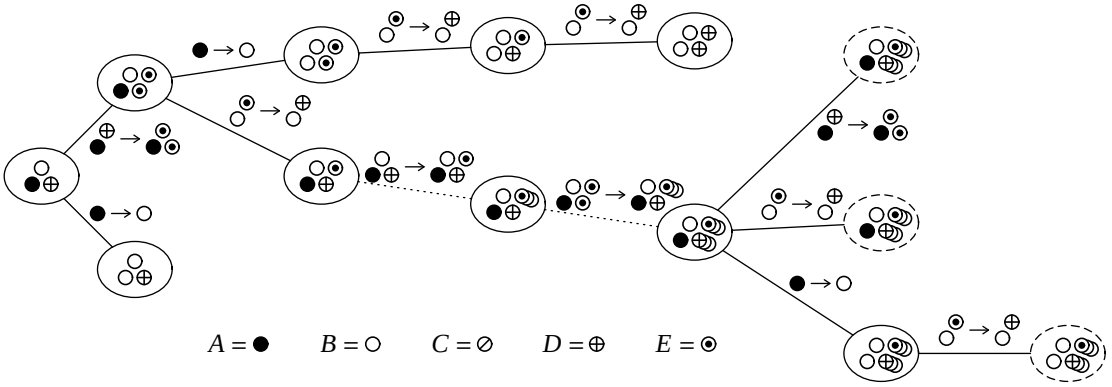


Figure 3.5: The search tree for the system of figure 3.4, starting on the left with state (A, B, D). Solid lines represent single reactions, while dotted lines represent any number of further repetitions of a completed cycle that purely increases a molecular quantity, leading to the attainability of arbitrarily large quantities of that species, shown for example as ◎. The dashed circles are repeats of previous states and thus do not require further exploration even if further reactions are possible.

In this example, the search tree is finite. Must this always be the case? If so, then there are no undecidable questions among questions which can be answered by scanning the full search tree. This section shows that the search tree is not only finite, but its size is boundable by a primitive recursive function.

A fundamental observation is that if the system is in state $\mathcal{A}$ at some point, and then later it is in state $\mathcal{B}$, and $\mathcal{B} \geq \mathcal{A}$, then the sequence of reactions that led from $\mathcal{A}$ to $\mathcal{B}$ may be repeated arbitrarily many times before continuing. This would appear to be a serious obstacle to exhaustively searching the space of reachable states, but in fact it will be the key to bounding the search. When this happens, we can consider two cases: $\mathcal{B} = \mathcal{A}$ or $\mathcal{B} > \mathcal{A}$.

If $\mathcal{B} = \mathcal{A}$, then this sequence of reactions leading from $\mathcal{A}$ to $\mathcal{B}$ had no effect, and may be omitted entirely. In particular, it is clear that the shortest sequence of reactions leading from the initial state of the system to any particular final state will not visit any state more than once. Thus, no possibilities will be missed if the search tree is simply pruned at any point where a previous state is repeated.

On the other hand, if $\mathcal{B} > \mathcal{A}$, that is, if $\mathcal{B}$ has strictly more of some species than the earlier state $\mathcal{A}$ had, then by repeating this sequence of reactions, an arbitrarily large amount of those species may be produced. We will call such species *freely generatable* after the sequence of reactions from $\mathcal{A}$ to $\mathcal{B}$ has occurred. If at any later point in the calculation,

some potential reaction is not possible because one of the freely generatable species has run out, we can simply retroactively assume that more repeats of the sequence from $\mathcal{A}$ to $\mathcal{B}$ were performed back at the time when that species became freely generatable, and this will allow the potential reaction to proceed after all. For this reason, when a species becomes freely generatable, it may effectively be removed from the problem statement, reducing the problem to a simpler problem. So although the search tree cannot be pruned when $\mathcal{B}$ is reached, the subtree beyond that point corresponds to searching the space of a simpler problem, in which a further repetition of the reaction sequence leading from $\mathcal{A}$ to $\mathcal{B}$ would indeed lead to pruning, since states $\mathcal{A}$ and $\mathcal{B}$ are equal in the reduced problem. The algorithm therefore specifies the quantity of a freely generatable species as ∞, a value which is considered larger than any other value, and which is unchanged by the addition or removal of molecules.

3.4.1.2 The Data Structure

Now we will define a data structure whose purpose will be to help us define the bound in the next section.

At each point in the search tree, there is a possibly infinite set $\mathbb{S}$ of all states $\mathcal{S}$ satisfying $\mathcal{S} \not\geq \mathcal{A}$ for every $\mathcal{A}$ which is an ancestor of that point in the search tree. We will call this set of states $\mathbb{S}$ the *remaining states* for that point in the search tree. Our proof will examine this set of states and use the structure of this set to provide a bound on how much deeper the search tree can be.

For any given point in the search tree, we represent the set of remaining states by lists $\mathbb{L}_i$, with each entry in list $\mathbb{L}_i$ representing an i-dimensional region of remaining states, specified by $n - i$ integers (specifying quantities of $n - i$ of the n species, the other i species being allowed to have any quantity). The union of all regions from all lists exactly yields the set of remaining states for the given point in the search tree.

When a reaction takes the system to a new state (taking the search to a new point in the search tree), the lists are modified by eliminating each list entry which represents a region containing any state greater than or equal to the new state. Each eliminated entry is replaced by new entries in the list of next lower index. The new entries are found by considering all regions of dimension one less than the old region, lying within the old region,

$\mathbb{L}_4$		$\mathbb{L}_3$
		$\vdots$
		$(2,0,\cdot,5,\cdot,3,\cdot)$
$\vdots$		$(2,1,\cdot,5,\cdot,3,\cdot)$
		$(2,2,\cdot,5,\cdot,3,\cdot)$
		$(2,3,\cdot,5,\cdot,3,\cdot)$
$(2,\cdot,\cdot,5,\cdot,3,\cdot)$	$\rightsquigarrow$	$(2,\cdot,0,5,\cdot,3,\cdot)$
		$(2,\cdot,\cdot,5,0,3,\cdot)$
		$(2,\cdot,\cdot,5,1,3,\cdot)$
$\vdots$		$(2,\cdot,\cdot,5,2,3,\cdot)$
		$\vdots$

Figure 3.6: An example of a possible entry in list $\mathbb{L}_4$, for a system with 7 species, and all the 8 entries that will replace it in list $\mathbb{L}_3$ if the system arrives at state $(2,4,1,3,3,3,0)$. The union of the new 3-dimensional regions is precisely that portion of the old 4-dimensional region which is $\not\geq$ the new state.

with a previously unspecified coordinate now specified as some particular integer k, with $0 \leq k < m$, where m is the number of molecules present, in the new state, of the species corresponding to the dimension being specified. An example is shown in figure 3.6.

The lists for the initial state of the system are created similarly, with the "old" region taken to be the full n-dimensional space, just a single entry in list $\mathbb{L}_n$. Thus, a system started in state $(q_1, q_2, ..., q_n)$, where q_i is the quantity of the i^{th} species, will start with $\sum_i q_i$ entries in list $\mathbb{L}_{n-1}$. Similarly, whenever an entry in list $\mathbb{L}_i$ is replaced by new entries in list $\mathbb{L}_{i-1}$ due to a new state $(q_1, q_2, ..., q_n)$, the number of new entries will be $\sum_{i \in P} q_i$, where P is the set of species whose quantity is unspecified in the old entry. The entries in the lists are not guaranteed to represent disjoint regions, or even unique regions. All that matters is that their union is the set of remaining states.

If the i^{th} species becomes freely generated, all list entries in all lists will have their i^{th} component changed to be specified as ∞, which may move some of them to the list of next lower index: Since ∞ is treated by the lists as a specified quantity, any list entry which previously did not specify the quantity of the i^{th} species will now have one fewer unspecified quantities, and will thus move to the list of next lower index.

3.4.1.3 The Bound

To each point in the search tree, with its state and its lists, we assign a positive integer as described below. We will see that regardless of which reaction is performed at the next step, the positive integer assigned to the ensuing point in the search tree will always be less than the positive integer assigned to the current point. Since the positive integer strictly decreases with depth, it is in fact a bound on the depth.

The integer for a given state $\mathcal{A}$ and lists $\mathbb{L}_i$ is defined for a system with n species in the following non-trivial way:

$$\mathfrak{B}(\mathcal{A},\mathbb{L}) = f_{n-1}^{|\mathbb{L}_{n-1}|}(f_{n-2}^{|\mathbb{L}_{n-2}|}(...(f_1^{|\mathbb{L}_1|}(f_0^{|\mathbb{L}_0|+m\cdot r+q_{\max}}(0)))...)) \ ,$$

where r is the number of non-freely generatable species, $q_{\max}$ is the largest number of molecules present of any of those r species, and m, a constant, is one more than the maximum coefficient appearing on the right-hand side of any reaction.

The functions f_i are defined as follows:

$$f_i(x) = f_{i-1}^{i\cdot x+m}(x)$$
$$f_0(x) = x+1 \ .$$

These definitions are not meant to capture intuitive notions of any meaningful functions, but rather are meant to (a) be explicitly primitive recursive, and (b) be of a form that enables the necessary proof steps below to work.

In these definitions, the exponents on the functions denote multiple applications of the function, so for example $f_8^3(x) = f_8(f_8(f_8(x)))$. Each f_i, as well as $\mathfrak{B}$, is a Primitive Recursive Function, since it is easy to define repeated application of a function: Given a function $g(x)$, we can define $h(n,x) = g^n(x)$ using the primitive recursive definition $h(0,x) = x,\ \ h(m+1,x) = g(h(m,x))$.

It is straightforward to show that the functions $f_i(x)$ are strictly increasing in x, and that $f_{i+1}(x) > f_i(x)$. Thus, if the exponents appearing within the definition of $\mathfrak{B}$ are in any way reduced or shifted to the right, $\mathfrak{B}$ will decrease.

This can be used to show that regardless of whether a reaction leads to a remaining

state or leads to a new freely generatable species, $\mathfrak{B}$ will always decrease.

If a reaction results in one or more new freely generatable species, then some parts of the exponents may shift to the right (due to components becoming ∞), and r will decrease. In the exponent of f_0, the decrease of r will more than make up for any increase in $q_{\max}$ (by the definition of m), so $\mathfrak{B}$ will decrease as promised.

If a reaction leads to a remaining state, then one or more list entries will be replaced by other entries. Each i-dimensional entry to be removed will be replaced by $\sum_{j \in P} q_j$ entries that are $(i-1)$-dimensional. This number of new entries is no more than $i \cdot q_{\max}$, since P, the set of species of unspecified quantity, is of size i. So the exponent of f_i is reduced by 1 while the exponent of f_{i-1} increases by at most $i \cdot q_{\max}$. In the formula for $\mathfrak{B}$, then, the innermost f_i gets replaced with $f_{i-1}^{i \cdot q_{\max}}$, and then this exponent is possibly reduced. But the original f_i was equivalent (by definition) to $f_{i-1}^{i \cdot x + m}$, where x is the full argument (which must be at least $q_{\max}$, since $q_{\max}$ appears in the exponent of f_0), so even just the $f_{i-1}^{i \cdot x}$ portion is bigger than the replacement, and the disappearing f_{i-1}^{m} portion more than compensates for any increase in the exponent of f_0 due to any change in $q_{\max}$. The total effect is therefore again a decrease in $\mathfrak{B}$.

3.4.2 Probabilities $\Longrightarrow$ Undecidable

In this section[1] we examine the computational power of stochastic chemical reaction networks.

Here we show that questions of *probability* for chemical reaction networks are undecidable. This result derives from showing that chemical reaction networks can efficiently simulate register machines within a known error bound that is independent of the unknown number of steps that will occur prior to halting.

In particular, we show that stochastic chemical reaction networks with reaction rates can compute any computable function with probability of error less than ϵ for any $\epsilon > 0$. (This was previously known to be impossible for $\epsilon = 0$, and was an open question for $\epsilon \neq 0$.)

Theorem 53 *For all $0 \leq \epsilon < 1/2$, the following problem is undecidable: given a chemical reaction network C, a species S, and a starting state $\mathcal{A}$, determine, to within ϵ, the*

[1] This section is joint work with David Soloveichik and Erik Winfree.

probability that C starting from $\mathcal{A}$ will produce at least one molecule of S.

What is interesting here is that ϵ can be almost $1/2$. That is, even if the system is practically guaranteed to produce a molecule of S, or practically guaranteed not to, it is impossible to distinguish between these situations. That is, the difficulty in determining what will happen does not lie in a technical difficulty of getting enough precision to decide which side of a fine line the probability lies on, but rather even if the answer is guaranteed to be far from the line, we still cannot figure out which side it is on. For example, not only is it undecidable whether the probability of entering a given state is over or under 50%, but it remains undecidable even if we are told that in fact the probability is either over 99.9% or under 0.1%. In other words, we really can't predict the probability in the slightest.

In section 3.4.2.2 we will show that stochastic chemical reaction networks in which each reaction's probability of occuring depends only on what reactions are possible (but not on the concentrations) are not capable of universal computation. This corresponds to a similarly provable result for Petri nets, that if the next transition is chosen randomly without regard to tokens (although of course it only fires if it is enabled), then the probability of entering a target state can be computed to arbitrary precision, but if the next transition is chosen by first randomly choosing a token and then firing (if possible) a transition that will absorb that token, then the probability of entering a target state cannot be predicted at all in general.

3.4.2.1 Simulating a Register Machine

In this section we will show how probabilistic chemical reaction networks are capable of simulating register machines very precisely. First, we define the correspondence between instantaneous descriptions of register machines and states of chemical reaction networks that our construction attains. Then, we show that determining whether a register machine ever reaches a particular instantaneous description is equivalent to ascertaining whether our chemical reaction network enters a set of states with sufficiently high probability.

Definition 23 *An* instantaneous description *ID of a register machine M with t registers is a vector $(a, c_1, \ldots, c_t)$ where a is a line of M's program (the* state *of the register machine), and $c_i \in \mathbb{N}$ represents the value of register i.*

Definition 24 *The reachability relation* $ID \overset{M*}{\rightarrow} ID'$ *is defined naturally. Namely, it is satisfied if* M *eventually reaches* ID' *starting from* ID*.*

Instantaneous descriptions of a register machine map to sets of states of our chemical reaction network as follows:

Definition 25 *For an instantaneous description* $ID = (a, c_1, \ldots, c_t)$ *of a register machine* M *let* $\Xi(ID, n)$ *be the state of a chemical reaction network that contains exactly:*

- n *molecules of species* A*,*
- c_i *molecules of* $R_i \quad \forall i \in [1, t]$*,*
- 1 *molecule of* S_a*,*
- *and* 1 *molecule of* T*,* B*,* B' *and* B'' *each.*

Definition 26 *Our chemical reaction networks will be said to* ϵ-follow *a register machine* M *if there is some* n_0 *such that for all instantaneous descriptions* ID *and* ID' *of* M *we have:*

(a) $ID \overset{M*}{\rightarrow} ID' \iff \Pr[\Xi(ID, n_0) \overset{C*}{\rightarrow} \Xi(ID', n) \text{ for some } n] > 1 - \epsilon$

(b) $ID \overset{M*}{\not\rightarrow} ID' \iff \Pr[\Xi(ID, n_0) \overset{C*}{\rightarrow} \Xi(ID', n) \text{ for some } n] < \epsilon$

Theorem 54 *For any register machine* M*, and any* $\epsilon > 0$*, there is a chemical reaction network* C *that* ϵ*-follows* M*.*

Corollary 55 *For all* $0 \leq \epsilon < 1/2$*, the following problem is undecidable: given a chemical reaction network* C *and a starting state* $\mathcal{A}$*, determine, to within* ϵ*, the probability that* C *starting from* $\mathcal{A}$ *will produce at least one molecule of* S*.*

In fact, slight modifications of our construction can show that all questions about whether a chemical reaction network "might do X" mentioned in section 3.4.1 become uncomputable in the probabilistic setting ("does X with probability $> \epsilon$").

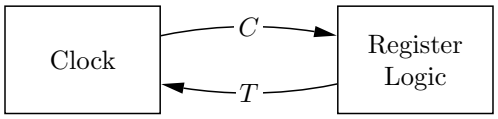
We now give the proof of theorem 54.

Proof: We construct a chemical reaction network to simulate the register machine, consisting of two components: a clock module and a register logic module (shown in figure 3.7). The communication between the modules is established through two species, T and C, of which at most a single molecule is present. Whenever the clock releases the C, the register logic module can complete a step of the register machine (with the exception of the actual decrement of a decrement instruction), converting the C into a T in the process. The clock module then takes the T and, after a delay, releases another C to repeat the process. The delay imposed by the clock module makes it exceedingly likely that any decrement waiting to happen will occur before the next C is released. This effectively enforces the reaction order that is necessary for correct computation.

The register logic module has a single molecule of species S_a for every state a of the register machine. The number of molecules of species R_i stores the value of the register i. If the current register machine state a is an increment state, once the clock module releases the C then the reaction $S_a + C \rightarrow S_b + R_i + T$ increments the ith register and transitions to the next state b. If the current state is a decrement state and the register i being read is empty, then the reaction $S_a + R_i \rightarrow S'_a$ is not possible, and once the clock module releases the C, the reaction $S_a + C \rightarrow S_c + T$ takes place and transitions to the state c indicating that the register is empty. If the register i is not empty (i.e., there is at least one molecule of R_i in solution), then the intent is that the reaction $S_a + R_i \rightarrow S'_a$ should decrement the register and capture S_a before the clock module next releases a C. (Otherwise, the reaction $S_a + C \rightarrow S_c + T$ could occur first, erroneously sending the register logic module into the state c, which is only supposed to happen if the register is empty.)

Thus, the only possible error that can occur in the register logic module is if $S_a + C \rightarrow S_c + T$ occurs before $S_a + R_i \rightarrow S'_a$ in a decrement step, when register i is not empty. By delaying the release of the C, the clock module ensures that the probability of this happening is low. The delay increases from step to step sufficiently to guarantee that the union bound, taken over all steps, of the probability of error does not exceed ϵ.

(a) The Clock and Register Logic Modules



(b) Clock

$$
\begin{aligned}
&T + B \to T' + B \\
&T' + A \to T + A \\
&T' + B' \to T'' + B' \\
&T'' + A \to T' + A \\
&T'' + B'' \to C + B'' + A
\end{aligned}
$$

(c) Register Logic

In state a increment register i and go to state b	$\Rightarrow$	$S_a + C$	$\to$	$S_b + R_i + T$
In state a decrement register i and go to state b, or if the register is empty go to state c	$\Rightarrow$	$S_a + R_i \to S'_a$ $S'_a + C \to S_b + T$ $S_a + C \to S_c + T$		

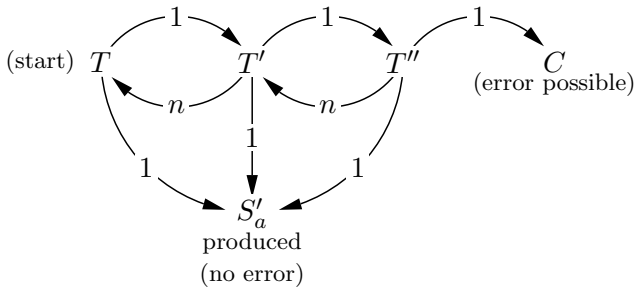
Figure 3.7: Simulating a register machine. (a) The communication between the clock and the register logic modules is through single molecules of species C and T. (b) The clock module is responsible for producing a C molecule once every so often. The clock module is designed so that the length of time between receiving a T and producing a C slowly increases throughout the computation, thus slowing down the register logic module to help it avoid error. Specifically, the more A's there are, the longer the delay. The clock starts out with n_0 A's and one each of B, B', B'', and T. Every clock cycle not only produces a C, but increases the number of A's by one. Thus at the beginning of the k^{th} cycle, there are $n = n_0 + k - 1$ molecules of A. The clock's operation is further analyzed in figure 3.8. (c) The register logic module simulates the register machine state transitions. The register logic module starts out with quantities of molecules of R_i indicating the starting value of register i, and a single molecule of species S_a where a is the start state of the register machine. Note that at all times the entire system contains at most a single molecule of any species other than the A and R_i species. All rate constants are 1. (The construction will work with any rate constants.)

Let us analyze the probability of error quantitatively. Suppose the current step is a decrement step and that the decremented register has value 1. This is the worst case scenario since if the register holds value greater than 1, the rate of the reaction $S_a + R_i \to S'_a$ is correspondingly faster, and if the step is an increment step or the register is zero, then no error can occur. Figure 3.8 illustrates the state diagram of the relevant process. All of the reactions in our chemical reaction network have the same rate constant of 1. (This construction will in fact work *regardless* of the rate constants!) Thus we can scale time (according to volume) so that all reactions with exactly one molecule of each reactant species in solution have the same reaction rate of 1. There are two reactions for which this single molecule condition is not true: $T' + A \to T + A$ and $T'' + A \to T' + A$, since there are many A's in solution. If there are n A's in solution, each of these two reactions has rate n. Now, we'll bound the probability that the clock produces the C before the $S_a + R_i \to S'_a$ reaction occurs, which is a bound on the probability of error. The top 4 states in the diagram (figure 3.8) represent the 4 possible states of the clock: we either have a T, T', T'', or a C. A new cycle starts when the register logic module produces the T and this is the start state of the diagram. No matter what state the clock is in, the reaction $S_a + R_i \to S'_a$ can occur at rate 1 in the register logic module. Once this happens, no error is possible for the current step. On the diagram this is indicated by the bottom state (no error) which is a sink. On the other hand, if a C is produced first then an error is possible. This is indicated by the sink state C (error possible).

We compute the absorption probability of the error-possible state by solving the corresponding flow problem. Solving the system of differential equations in figure 3.8 for $\frac{dp}{dt}$ under the condition that $\frac{ds}{dt} = -1$, $\frac{ds'}{dt} = \frac{ds''}{dt} = 0$, we find that the absorption probability of the error-possible state is $p = \frac{1}{(n+2)^2+4}$. Thus the probability of error for a step with n A's is bounded by $p = \frac{1}{(n+2)^2+4}$. In order to be sure that the probability that no error occurs during *any* point in the computation is larger than $1 - \epsilon$, recall that n increases by one at each step, so we need

$$\sum_{n=n_0}^{\infty} \frac{1}{(n+2)^2+4} < \epsilon.$$

The terms in the above inequality are inversely quadratic in n, so if $n_0 = 1$ then the sum is finite (in fact, it is roughly 0.3354). This means that for any ϵ, we can choose an appropriate n_0, the initial number of A's, to make the above inequality true. ■



$$\frac{d}{dt}\begin{bmatrix} s \\ s' \\ s'' \\ p \\ q \end{bmatrix} = \begin{bmatrix} -2 & n & 0 & 0 & 0 \\ 1 & -2-n & n & 0 & 0 \\ 0 & 1 & -2-n & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} s \\ s' \\ s'' \\ p \\ q \end{bmatrix}$$

Figure 3.8: The state diagram for a single decrement operation when there are n A's and the register to be decremented holds the value 1, and the corresponding system of differential equations governing the instantaneous probabilities of being in a given state. The numbers on the arrows are the transition rates. The instantaneous probability of being in state T is s, in state T' is s', and in state T'' is s''. The instantaneous probability of being in the error-possible state is p and the probability of being in the no-error state is q.

It is straightforward to see that if one has a different clock design that exhibits even slightly super-linear slowdown – e.g., $O(\frac{1}{n^{1+\Delta}})$ – then this would also result in constant probability of output error. That is to say, the cost of reliable, efficient simulation of register machines could be made negligible with such a clock.

3.4.2.2 Universality $\Longrightarrow$ Probability Is Concentration Dependent

If the rates of the possible reactions do not depend on the number of molecules then it can be shown that the system is incapable of universal computation. In particular, it will be predictable in the sense that the probability that at least one molecule of a given species is eventually produced can be computed to arbitrary precision. This result implies that all stochastic chemical reaction networks using an indicator species whose production with high or low probability indicates the outcome of the computation (or any other method of output that can be converted to this form) cannot be universal.

Specifically, the model we are considering here is the following: Suppose we are given a chemical reaction network with given constant rates for all the reactions, and an initial set of molecules. Then at each step, based solely on the reaction rates (and not on concentrations), a reaction is chosen. This reaction then occurs if the reactants for it are present. Such steps continue indefinitely.

Theorem 56 *Suppose for all reactions α and states $\mathcal{A}$, $\rho_\alpha(\mathcal{A}) = k_\alpha$ if all the reactants of α are present in $\mathcal{A}$ and 0 otherwise. Then there is an algorithm that given $0 < \epsilon$ and any starting state $\mathcal{A}$ and any decidable set of states $\mathbb{S}$, computes $\Pr[\mathcal{A} \stackrel{C*}{\rightarrow} \mathcal{B}$ for some $\mathcal{B} \in \mathbb{S}]$ within ϵ.*

The difference between this model and the standard stochastic one is that in the standard model, the reaction rate is obtained by combining a rate constant with the current concentrations as described in section 3.2.5.1 (eqn. 3.1), while here for all reactions α and states $\mathcal{A}$, $\rho_\alpha(\mathcal{A}) = k_\alpha$ if all the reactants of α are present in $\mathcal{A}$ and 0 otherwise.

To see why this difference has such an effect, let $\mathbb{S}$ be the infinite set of all states with at least one molecule of the indicator species present. Let $\mathbb{Q}$ be the (probably infinite) set of states from which no state in $\mathbb{S}$ is reachable, and let $\mathbb{R}$ be the set of states outside $\mathbb{S}$ from

which it is possible to reach $\mathbb{S}$. (Note that given any state, the question of whether it is possible to reach some state in $\mathbb{S}$ is computable, as discussed in section 3.4.1.) Note also that there is a bound b such that for any state $\mathcal{A} \in \mathbb{R}$, the length of the shortest sequence of reactions leading from $\mathcal{A}$ into $\mathbb{S}$ is at most b. This means that there is some constant p_0 such that for any state $r \in \mathbb{R}$, the probability of entering $\mathbb{S}$ within b steps is at least p_0. Thus, the probability of remaining in $\mathbb{R}$ must decay at least exponentially.

This implies that the probability that the system will eventually enter $\mathbb{S}$ or $\mathbb{Q}$ is 1, and so simply by computing the probabilities of the state tree for $\mathbb{R}$ far enough, one can compute the probability of entering $\mathbb{S}$ to arbitrary precision.

3.4.2.3 Open Questions

Here we list some questions, along the lines of the results we have given, which we believe may be within reach and which (regardless of whether they are proven or disproven) would contribute to our understanding of what allows chemical reaction networks to be universal.

- Are continuous chemical reaction networks (using mass action kinetics) universal?
- Can one have a universal chemical reaction network which has constant probabilities (that don't depend on concentrations) for all reactions except one, with the remaining reaction having a decaying probability that depends on time (but not on concentrations)?
- Can chemical reaction networks with reversible reactions be universal?
- Can the time and space requirements for stochastic chemical reaction networks, compared to a Turing Machine, be a simple polynomial slowdown in time, and an exponential increase in space?

The last question is almost easy: Our definition of a register machine can be augmented to to allow an instruction to multiply a register by 2 or to divide it by 2. This allows efficient simulation of a Turing machine, assuming the multiplication and division are efficient. Multiplication can be efficiently performed by a chemical reaction network using a self-catalyzing doubling reaction (taking time logarithmic in the size of the register), but it turns out that dividing by 2 is the stumbling block. Simple approaches, like using a reaction of

the form $R_i + R_i \longrightarrow R'_i$, are very slow to finish, as the last few molecules being divided by 2 (i.e., being merged) have trouble finding each other. If an efficient way could be found to divide a given species by 2 (and note the remainder), the rest of the construction is comparatively easy. We have found this to be a very intriguing open problem.

Bibliography

[Ack28] Wilhelm Ackermann. Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematische Annalen*, 99:118–133, 1928.

[AM99] Srinivas M. Aji and Robert J. McEliece. The generalized distributive law. September 23 1999.

[ARM98] Adam P. Arkin, John Ross, and Harley H. McAdams. Stochastic kinetic analysis of a developmental pathway bifurcation in phage-l escherichia coli. *Genetics*, 149(4):1633–1648, 1998.

[BB90] Gérard Berry and Gérard Boudol. The chemical abstract machine. In *Proceedings of the 17th ACM SIGPLAN-SIGACT annual symposium on principles of programming languages*, pages 81–94, 1990.

[BBDL01] Therese C. Biedl, Prosenjit Bose, Erik D. Demaine, and Anna Lubiw. Efficient algorithms for petersen's matching theorem. *Journal of Algorithms*, 38:110–134, 2001.

[BD97] R. Bubley and M. Dyer. Graph orientations with no sink and an approximation for a hard case of #SAT. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 248–257, 1997.

[Ben82] Charles H. Bennett. The thermodynamics of computation—a review. *International Journal of Theoretical Physics*, 21:905–940, 1982.

[BKJ03] Andrei A. Bulatov, Andrei Krokhin, and Peter Jeavons. The complexity of maximal constraint languages. Technical report, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, 2003.

[BKKR69] V. G. Bodnarchuk, L. A. Kaluzhnin, V. N. Kotov, and B. A. Romov. Galois theory for Post algebras. *Kibernetika*, 5(3):1–10, May-June 1969.

[CKS01] Nadia Creignou, Sanjeev Khanna, and Madhu Sudan. *Complexity Classifications of Boolean Constraint Satisfaction Problems*. Society for Industrial and Applied Mathematics, 2001.

[Cod90] Edgar F. Codd. *The Relational Model for Database Management: Version 2*. Addison Wesley, April 1 1990.

[Con72] John Horton Conway. Unpredictable iterations. In *Proceedings of the 1972 Number Theory Conference*, pages 49–52. University of Colorado, Boulder, 1972.

[DD00] C. J. Date and Hugh Darwen. *Foundation for Future Database Systems: The Third Manifesto, 2nd Edition*. Addison Wesley Professional, 2000.

[Dec03] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[EN94] Javier Esparza and Mogens Nielsen. Decidability issues for petri nets - A survey. *Bulletin of the European Association for Theoretical Computer Science*, 52:245–262, 1994.

[Fed01] Tomás Feder. Fanout limitations on constraint systems. *Theoretical Computer Science*, 255(1–2):281–293, March 2001.

[Fis61] M. E. Fisher. Statistical mechanics of dimers on a plane lattice. *Physical Review*, 124:1664–1672, 1961.

[GB00] Michael Gibson and Jehoshua Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *Journal of Physical Chemistry A*, 104(9):1876–1889, 2000.

[Gei68] D. Geiger. Closed systems of functions and predicates. *Pacific Journal of Mathematics*, 27:95–100, 1968.

[Gil77] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, 81:2340–2361, 1977.

[GP98] Peter J. E. Goss and Jean Peccoud. Quantitative modeling of stochastic systems in molecular biology by using stochastic petri nets. *Proceedings of the National Academy of Sciences USA*, 95:6750–6755, 1998.

[Her97] Michael J. Hernandez. *Database Design for Mere Mortals: A Hands-On Guide to Relational Database Design*. Addison-Wesley Developers Press, 1997.

[HS90] Harry B. Hunt III and R. E. Stearns. The complexity of very simple boolean formulas with applications. *SIAM Journal on Computing*, 19:44–70, 1990.

[HWP98] Graeme Halford, William H. Wilson, and Steven Phillips. Processing capacity defined by relational complexity: Implications for comparative, developmental, and cognitive psychology. *Behavioral and Brain Sciences*, 21:803–831, 1998.

[HWR91] Allen Hjelmfelt, Edward D. Weinberger, and John Ross. Chemical implementation of neural networks and turing machines. *Proceedings of the National Academy of Sciences USA*, 88:10983–10987, 1991.

[Kas61] P. W. Kasteleyn. The statistics of dimers on a lattice. *Physica*, 27:1209–1225, 1961.

[KM69] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3:147–195, 1969.

[Lag85] Jeff Lagarias. The $3x+1$ problem and its generalizations. *American Mathematical Monthly*, 92:3–23, 1985.

[Mag97] Marcelo O. Magnasco. Chemical kinetics is turing universal. *Physical Review Letters*, 78(6):1190–1193, 1997.

[May81] E. W. Mayr. Persistence of vector replacement systems is decidable. *Acta Informatica*, 15:309–318, 1981.

[Min67] Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice Hall, 1967.

[MMC98] Robert J. McEliece, David MacKay, and Jung-Fu Cheng. Turbo decoding as an instance of Pearl's 'belief propagation' algorithm. *IEEE Journal on Selected Areas in Communication*, 16(2):140–152, 1998.

[MWJ99] Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. Loopy belief propagation for approximate inference: an empirical study. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 467–475. Morgan Kaufmann, 1999.

[Nea05] Nicoleta Neagu. *Constraint Satisfaction Techniques for Agent-Based Reasoning*. Birkhäuser, 2005.

[Pap94] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Pau95] Gheorghe Paun. On the power of the splicing operation. *International Journal of Computer Mathematics*, 59:27–35, 1995.

[Pea88] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[Pet62] Carl Adam Petri. Kommunikation mit Automaten. Schriften des IIM 2, Institut für Instrumentelle Mathematik, 1962.

[Pip97] Nicholas Pippenger. *Theories of Computability*. Cambridge University Press, 1997.

[Pos41] Emil L. Post. *On The Two-Valued Iterative Systems of Mathematical Logic*. Princeton University Press, 1941.

[Rie03] Marc D. Riedel. *Cyclic Combinational Circuits*. PhD thesis, California Institute of Technology, November 17 2003.

[RML93] V. N. Reddy, M. L. Mavrovouniotis, and M. N. Liebman. Petri net representations in metabolic pathways. In *Proceedings of the First International Conference on Intelligent Systems for Molecular Biology*, volume 1, pages 328–336, 1993.

[RR94] Alexander A. Razborov and Steven Rudich. Natural proofs. In *26th ACM Symposium on Theory of Computing*, pages 204–213, 1994.

[RY86] Louis E. Rosier and Hsu-Chun Yen. A multiparameter analysis of the boundedness problem for vector addition systems. *Journal of Computer and System Sciences*, 32:105–135, 1986.

[Sch78] T. J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 216–226. Association for Computing Machinery, 1978.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing (Boston), 1997.

[ST77] George S. Sacerdote and Richard L. Tenney. The decidability of the reachability problem for vector addition systems (preliminary version). In *9th Annual Symposium on Theory of Computing, Boulder*, pages 61–76, 1977.

[TF61] H. N. V. Temperley and M. E. Fisher. Dimer problems in statistical mechanics – an exact result. *Philosophical Magazine*, 6:1061–1063, 1961.

[Val02] Leslie G. Valiant. Quantum circuits that can be simulated classically in polynomial time. *SIAM Journal of Computing*, 31(4):1229–1254, April 2002.

[Val04] Leslie G. Valiant. Holographic algorithms. In *FOCS Proceedings*, 2004.

[Weg87] Ingo Wegener. *The complexity of Boolean functions*. Wiley-Teubner, 1987.

[YFW00] Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. Generalized belief propagation. Technical Report TR-2000-26, Mitsubishi Electric Research Laboratory, June 2000.

[YM59] Yu. I. Yanov and A. A. Muchnik. On existence of k-valued closed classes without a finite basis. *Dokl. Akad. Nauk USSR*, 127(1):44–46, 1959.