Incorrect Lower Bounds for Path Consistency and More

T. K. Satish Kumar*
Department of Computer Science
University of Southern California
tkskwork@gmail.com

Liron Cohen
Department of Computer Science
University of Southern California
lironcoh@usc.edu

Sven Koenig
Department of Computer Science
University of Southern California
skoenig@usc.edu

In this paper, we present an efficient algorithm for verifying path-consistency on a binary constraint network. The complexities of our algorithm beat the previous conjectures on the lower bounds for verifying path-consistency. We therefore defeat the proofs for several published results that incorrectly rely on these conjectures. Our algorithm is motivated by the idea of reformulating path-consistency verification as fast matrix multiplication. Further, for a computational model that counts arithmetic operations (rather than bit operations), a clever use of the properties of prime numbers allows us to design an even faster variant of the algorithm. Based on our algorithm, we hope to inspire a new class of techniques for verifying and even establishing varying levels of local-consistency on given constraint networks.

Introduction and Background

Constraints constitute a very natural and general means for formulating regularities in the real world. Developments in constraint reasoning bear immediate and important implications on how fast we can solve computational problems that arise in several other areas of research: including computer vision, spatial and temporal reasoning, model-based diagnosis, planning, language understanding, etc.

A *constraint satisfaction problem* (CSP) can be defined using a triplet $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where $\mathcal{X} = \{X_1, X_2 \ldots X_N\}$ is a set of *variables*, and $\mathcal{C} = \{C_1, C_2 \ldots C_M\}$ is a set of *constraints* between subsets of them. Each variable X_i is associated with a discrete-valued *domain* $D_i \in \mathcal{D}$, and each constraint C_i is a pair $\langle S_i, R_i \rangle$ defined on a subset of variables $S_i \subseteq \mathcal{X}$, called the *scope* of C_i. $R_i \subseteq D_{S_i}$ (where $D_{S_i} = \times_{X_j \in S_i} D_j$) denotes all compatible tuples of D_{S_i} allowed by the constraint. The cardinality of S_i is referred to as the *arity* of that constraint. The size of the largest domain is denoted by D. A *solution* to a CSP is an assignment of values to all the variables from their respective domains such that all the constraints are satisfied. In a *binary* CSP, each constraint is restricted to have an arity of at most two. It is well known that binary CSPs are representationally as powerful as general CSPs. The task of finding a solution to a CSP or a binary CSP is NP-hard in general (Dechter 2003).

Although solving a given CSP is hard in general, certain kinds of polynomial-time algorithms can be employed to make quick inferences about inconsistent assignments to subsets of the variables. These algorithms are referred to as *local-consistency* algorithms, and they transform a given CSP into a more explicit one by deducing additional constraints and adding them to the problem. The spectrum of local-consistency algorithms is characterized by the varying cardinality of the subsets of variables that we examine locally in each iteration. For example, in *arc-consistency*, we examine only two variables - X_i and X_j - at a time, and make sure that each domain value of X_i has at least one consistent match in the domain of X_j, and vice-versa. In general, we can examine k variables at a time, and add $(k-1)$-ary constraints to the given CSP. This is referred to as establishing k-consistency. With increasing k, enforcing k-consistency makes deeper inferences but the procedure is exponential in k.

Path-Consistency (PC) refers to local-consistency with $k = 3$. PC ensures that any consistent assignment to any two variables is extensible to any other third variable. Formally, given a CSP $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, two variables (X_i, X_j) are said to be *path-consistent* relative to X_k if and only if for every consistent assignment $(X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})$, there exists $d_{kw} \in D_k$ such that the assignments $(X_i \leftarrow d_{iu}, X_k \leftarrow d_{kw})$ and $(X_j \leftarrow d_{jv}, X_k \leftarrow d_{kw})$ are consistent. We say that a CSP is *path-consistent* if and only if for every $i \neq k, j \neq k, i \neq j$, the two variables (X_i, X_j) are path-consistent relative to X_k. PC has the special property that, on a binary CSP, any newly inferred constraint is also binary. Hence, the result of establishing PC on a binary CSP is also a binary CSP. Furthermore, PC is of deep theoretical interest for analyzing the tractability of various classes of constraints. For example, 2-SAT constraints, Connected Row-Convex (CRC) constraints (Deville et al. 1997), Tree-Convex constraints (Zhang and Freuder 2004), Simple Temporal Problems, Restricted Disjunctive Temporal Problems (Kumar 2005), and certain kinds of geometric CSPs, can all be solved in polynomial time by establishing PC.

Unfortunately, algorithms that establish PC have wrongfully used the conjectured $\Omega(N^3 D^3)$ lower bound from (Mohr and Henderson 1986). The optimality proofs for many of these algorithms use the following common logical structure in their arguments: (1) The proposed algorithm

*Alias: Satish Kumar Thittamaranahalli

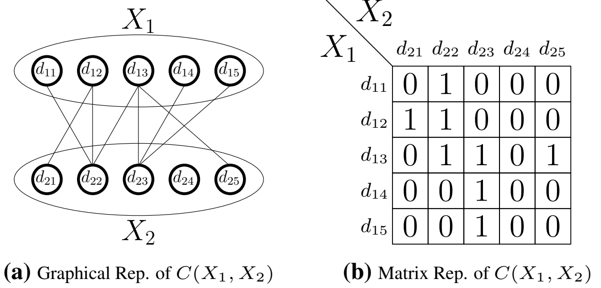
Copyright © 2013, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

runs in cubic time; (2) Establishing PC is harder than verifying it; and (3) Verifying PC has the same cubic lower bound. Improving the lower bound for verifying PC defeats the above proof-structure for the many results that rely on it. Some examples of published papers that rely on these incorrect lower bounds are: (a) Incorrectly remarking about the generic lower bound for PC (Kumar and Russell 2006, page 2): "*This algorithm is optimal, since even verifying path-consistency has the same lower bound.*"; (b) Incorrectly analyzing Singleton Arc-Consistency (Bessiere and Debruyne 2004, page 2): "*Verifying if it is path consistent (PC) is in $O(end^3)$. For each of the variables (i, j), we must check whether the triangles i, k, j with $C_{kj} \in C$ are path consistent. (Checking the other triangles is useless.)*"; and (c) Incorrectly claiming the optimality of PC-4 in a standard text book (Dechter 2003, page 66): "*It is an optimal algorithm, since even verifying path-consistency has that lower bound; namely, it is $\Omega(n^3k^3)$*" (here k refers to the size of the largest domain D).

In this paper, we present an efficient algorithm for verifying PC on a binary constraint network in time $O(N^3D^{2.38})$. This result holds for a computational model that counts the number of bit operations or the number of arithmetic operations. We also present a variant of this algorithm that has a time complexity of $O(N^{2.38}D^{2.38})$ when we measure the number of arithmetic operations. Our algorithms are motivated by the idea of reformulating path-consistency verification as fast matrix multiplication, and a clever use of the properties of prime numbers.

Numerical matrix multiplication - sometimes simply referred to as "matrix multiplication" - finds its use in various applications such as analyzing electrical circuits, solving systems of linear and/or differential equations, computing fast Fourier transforms, etc. At first glance, numerical matrix multiplication of two $N \times N$ matrices seems to require a total of $O(N^3)$ operations. This is because the resulting matrix is also $N \times N$, and each entry in this matrix requires computing the dot product of two N-dimensional vectors. However, Strassen's algorithm (Strassen 1969) provided the first breakthrough in reducing this complexity to $O(N^{2.81})$. The basic ideas in Strassen's algorithm are to: (a) recursively split the matrices into "blocks"; and (b) recognize that the multiplication of blocks is amenable to the exploitation of certain common factors. Over the years, the complexity of matrix multiplication has been reduced even further using various other mathematical manipulations. The best known algorithm for numerical matrix multiplication runs in time $O(N^{2.38})$ (Coppersmith and Winograd 1987). Moreover, there is a continuing effort in the Theory community to reduce this complexity more and more.

The time complexities of the algorithms that we present in this paper for verifying PC match that of multiplying two matrices. This beats the previously conjectured lower bound of $\Omega(N^3D^3)$ and therefore defeats the arguments used in the proofs of various results published throughout the years. We expect our techniques to be generalizable to the task of verifying higher levels of local-consistency. We also hope to inspire a new class of algorithms for *establishing* PC as well as higher levels of local-consistency with lower time



(a) Graphical Rep. of $C(X_1, X_2)$ **(b)** Matrix Rep. of $C(X_1, X_2)$

Figure 1: The graphical representation of a binary constraint $C(X_1, X_2)$ is a set of edges representing consistent combinations of values to the participating variables. The small dark circles indicate the domain values, and the larger ovals represent entire domains for the variables. The matrix representation requires an ordering for the domain values for each of the variables.

complexities. This in turn has crucial implications on the efficiency of solving CSPs in general.

Representing Binary CSPs as Matrices

For a given binary CSP, we can build a matrix representation for it using a simple mechanism. First, we assume that the domain values for each variable are ordered in some way. (We can simply use the order in which the domain values for each of the variables are specified.) Under such an ordering, we can represent each binary constraint as a 2-dimensional matrix with all its entries set to either 1 or 0 based on whether the corresponding combination of values to the participating variables is allowed or not by that constraint. Figure 1 shows the $(0, 1)$-matrix representation of a binary constraint between two variables X_1 and X_2 with domain sizes of 5 each. The combination of values $(X_1 \leftarrow d_{12}, X_2 \leftarrow d_{21})$ is an allowed combination, and the corresponding entry in the matrix is therefore set to 1. However, the combination of values $(X_1 \leftarrow d_{14}, X_2 \leftarrow d_{22})$ is a disallowed combination, and the corresponding entry is therefore set to 0. It is also easy to see that such a matrix representation can be extended to non-binary constraints as well using higher dimensional $(0, 1)$-matrices. However, for the purposes of this paper, and because binary CSPs are representationally as powerful as general CSPs, we choose to restrict our discussion to the binary case alone.

The matrix representation of an entire binary CSP can be constructed simply by stacking up the matrix representations for the individual constraints into a bigger "block" matrix. Figure 2 illustrates how a binary CSP on 3 variables X_1, X_2 and X_3 can be represented as a "mega-matrix" with 3 sets of rows and 3 sets of columns. Each block-entry inside this mega-matrix is the matrix representation of the direct constraint between the corresponding row and column variables. In essence, therefore, the matrix representation of an entire binary CSP has $\sum_{i=1}^{N} |D_i|$ rows and $\sum_{i=1}^{N} |D_i|$ columns. In the rest of this paper, we will commonly denote the matrix representation of an entire given binary CSP by $\mathcal{A}$.

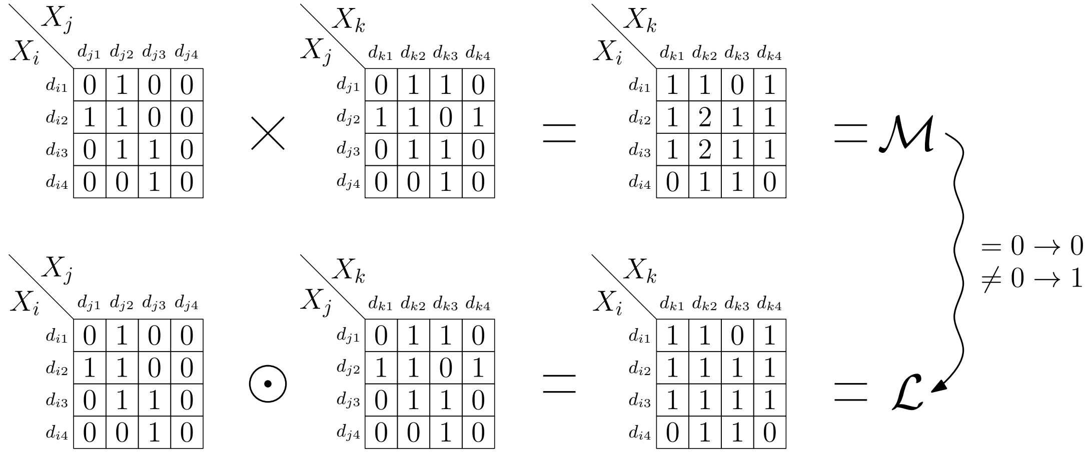


Figure 3: Shows the numerical and logical multiplications of two constraint matrices. It also illustrates that the logical multiplication of two constraint matrices $C(X_i, X_j) \odot C(X_j, X_k)$ can be simulated using the numerical multiplication $C(X_i, X_j) \times C(X_j, X_k)$ with a simple post-processing technique of changing all non-zero entries in the resulting matrix to 1. It is important to note that this emulation works only for operations on individual constraints and not on entire CSPs.

		X_1			X_2			X_3		
		d_{11}	d_{12}	d_{13}	d_{21}	d_{22}	d_{23}	d_{31}	d_{32}	d_{33}
X_1	d_{11}	1	0	0	1	1	1	0	0	1
	d_{12}	0	1	0	0	0	1	1	1	1
	d_{13}	0	0	1	1	1	1	1	0	0
X_2	d_{21}	1	0	1	1	0	0	1	1	1
	d_{22}	1	0	1	0	1	0	1	0	1
	d_{23}	1	1	1	0	0	1	1	0	1
X_3	d_{31}	0	1	1	1	1	1	1	0	0
	d_{32}	0	1	0	1	0	0	0	1	0
	d_{33}	1	1	0	1	1	1	0	0	1

Figure 2: The matrix representation of an entire binary CSP consists of a 2-dimensional array of blocks, each of which represents the direct binary constraint between the corresponding row and column variables.

Logical and Numerical Operations on CSPs and Matrices

In this section, we will define *logical operations* on CSPs that are relevant to PC, and we will try to capture them using *numerical operations* on their matrix representations.

The *logical multiplication* of two matrices representing the constraints $C(X_i, X_j)$ and $C(X_j, X_k)$ is defined to be the matrix representation of a constraint $C(X_i, X_k)$ such that any combination of values to X_i and X_k from their respective domains is set to be allowed if and only if there is some value of X_j that is consistent with both of them. Figure 3 shows the numerical as well as the logical multiplication between the matrix representations of two constraints (denoted by the operators $\times$ and $\odot$, respectively).

Figure 3 also illustrates that the logical multiplication of the matrix representations of two constraints can be simulated using a regular numerical multiplication of them. Although the numerical multiplication yields a matrix with entries that could be greater than 1, simply converting all non-zero entries to 1 results in an equivalence to the logical multiplication. The logical multiplication of constraints is the fundamental operation used repeatedly in establishing PC.

Let $\mathcal{A}$ be the matrix representation of a binary CSP. As noted previously, $\mathcal{A}$ can be viewed as a block-matrix with block $\mathcal{A}_{ij}$ being the matrix representation of the constraint between X_i and X_j. We can now define the *logical multiplication* $\mathcal{L} = \mathcal{A} \odot \mathcal{A}$ as follows. $\mathcal{L}$ is also a block-matrix with the block $\mathcal{L}_{ij}$ representing the *cumulative* effect of the logical multiplications of the constraints $C(X_i, X_k)$ and $C(X_k, X_j)$ over all k. The cumulative effect is in essence just the logical $\wedge$-ing of the N constraint matrices corresponding to every possible value of k.[1]

The task of verifying PC can now be viewed as the task of checking whether $\mathcal{A} \odot \mathcal{A} = \mathcal{A}$. This is because the left-hand side $\mathcal{A} \odot \mathcal{A}$ represents an attempt to tighten the constraints between pairs of variables using every possible third variable as required in PC. The equality check to the right-hand side represents a convergence to a path-consistent network.

Since $\mathcal{A}$ can be viewed as an $N \times N$ block-matrix, $\mathcal{A} \odot \mathcal{A}$ can be computed in N^3 operations on individual blocks.[2] Individual multiplications of blocks are essentially logical multiplications of constraints, and they can be emulated using numerical multiplications as explained above. This means that the total running time for verifying PC can be improved from $\Omega(N^3 D^3)$ to $O(N^3 D^{2.38})$ in a relatively straightforward way.[3] This already defeats the arguments used in various published papers concerning the lower

[1] $\wedge$-ing two constraint matrices is equivalent to simply $\wedge$-ing each of the corresponding entries in the matrices.

[2] The cubic number of operations on blocks cannot be obviated right away because the required multiplication is logical and not numerical.

[3] The bit complexity is also the same because all entries in the matrix remain bounded by D.

bounds for verifying PC.

We can also attempt to reduce N^3 to $N^{2.38}$ using a similar line of thought. Figure 5 shows the result of such an attempt by naively casting the logical multiplication $\mathcal{A} \odot \mathcal{A}$ as a regular numerical multiplication $\mathcal{A} \times \mathcal{A}$. Of course, computing $\mathcal{A} \times \mathcal{A}$ requires us to view $\mathcal{A}$ as a "flattened-out" matrix without a block-structure, but with $\sum_{i=1}^{N} |D_i|$ rows and $\sum_{i=1}^{N} |D_i|$ columns. Figure 5 also illustrates the failure of this naive transformation. In particular, $(X_1 \leftarrow d_{11}, X_2 \leftarrow d_{22})$ is not ruled out by the numerical multiplication $\mathcal{A} \times \mathcal{A}$ while it is ruled out by the logical multiplication $\mathcal{A} \odot \mathcal{A}$.

A closer look to analyze the failure of the naive method leads us to define the concept of a *support*. The assignment $X_k \leftarrow d_{kw}$ is said to be a *support* for $(X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})$ in X_k's domain if and only if $X_k \leftarrow d_{kw}$ is consistent with both $X_i \leftarrow d_{iu}$ and $X_j \leftarrow d_{jv}$. The *support set* of $(X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})$ in X_k's domain is defined to be the set of all domain values of X_k that qualify as supports for $(X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})$. Now, we observe that the numerical multiplication $\mathcal{A} \times \mathcal{A}$ yields a matrix $\mathcal{M}$ with the following properties. First, $\mathcal{M}$ has $\sum_{i=1}^{N} |D_i|$ rows and $\sum_{i=1}^{N} |D_i|$ columns with one row and one column corresponding to each possible variable-value combination. Second, $\mathcal{M}_{[X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv}]}$ computes the *total number* of supports for $(X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})$ over all possible X_k. The true requirement, however, is to compute whether $(X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})$ has *at least one* support in *every* other variable's domain independently.

Properties of Prime Numbers

In the previous section, we noticed that casting logical operations on constraints as numerical operations on regular matrices does not always yield the desired result for entire CSPs. In order to bridge the gap, we make use of the properties of prime numbers in a novel way. A *prime number* is a whole number greater than 1 that is divisible only by itself and 1. The *prime factorization* of any whole number is unique. Moreover, there are an infinite number of primes greater than any given number, and roughly $N/\log N$ primes $\leq N$ (Newman 1980). There is also an efficient algorithm for finding all primes in a specified range (Gries and Misra 1978). For the purposes of this paper, however, it suffices to assume the existence of a large table of primes that is available a priori. Before we apply prime numbers to the task of verifying PC, we illustrate their relevance through the following Lemma.

Lemma 1. *Suppose we have K positive rational numbers $\frac{s_1}{p_1}, \frac{s_2}{p_2} \ldots \frac{s_K}{p_K}$ in their* irreducible forms *s.t.* $\forall i \in [1, 2 \ldots K]$, *$p_i$ is a prime number, s_i is a non-negative integer, and $p_i > s_i$.*[4] *The irreducible form $\frac{A}{B}$ of the sum $\sum_{i=1}^{K} \frac{s_i}{p_i}$ has $B = \prod_{i=1}^{K} p_i$ if and only if none of the s_i are zero.*

[4] The *irreducible form* of a rational number is a fraction $\frac{P}{Q}$ such that $GCD(P, Q) = 1$.

$$(I) \quad \frac{1}{3} + \frac{2}{7} + \frac{11}{13} + \frac{8}{17} = \frac{1 \cdot 7 \cdot 13 \cdot 17 + 2 \cdot 3 \cdot 13 \cdot 17 + 11 \cdot 3 \cdot 7 \cdot 17 + 8 \cdot 3 \cdot 7 \cdot 13}{3 \cdot 7 \cdot 13 \cdot 17} = \frac{8984}{4641}$$

$$(II) \quad \frac{1}{3} + \frac{0}{7} + \frac{11}{13} + \frac{8}{17} = \frac{1 \cdot 13 \cdot 17 + 11 \cdot 3 \cdot 17 + 8 \cdot 3 \cdot 13}{3 \cdot 13 \cdot 17} = \frac{1094}{663}$$

Figure 4: Illustrates the truth of Lemma 1. The first example shows that if none of the terms is zero, then all the primes appear in the denominator of the summation. The second example shows that in case one of the terms is zero, then the denominator of the irreducible form of the summation has a missing prime number.

Proof. Suppose that $\forall i \in [1, 2 \ldots K]$, $s_i > 0$. Then, we know that

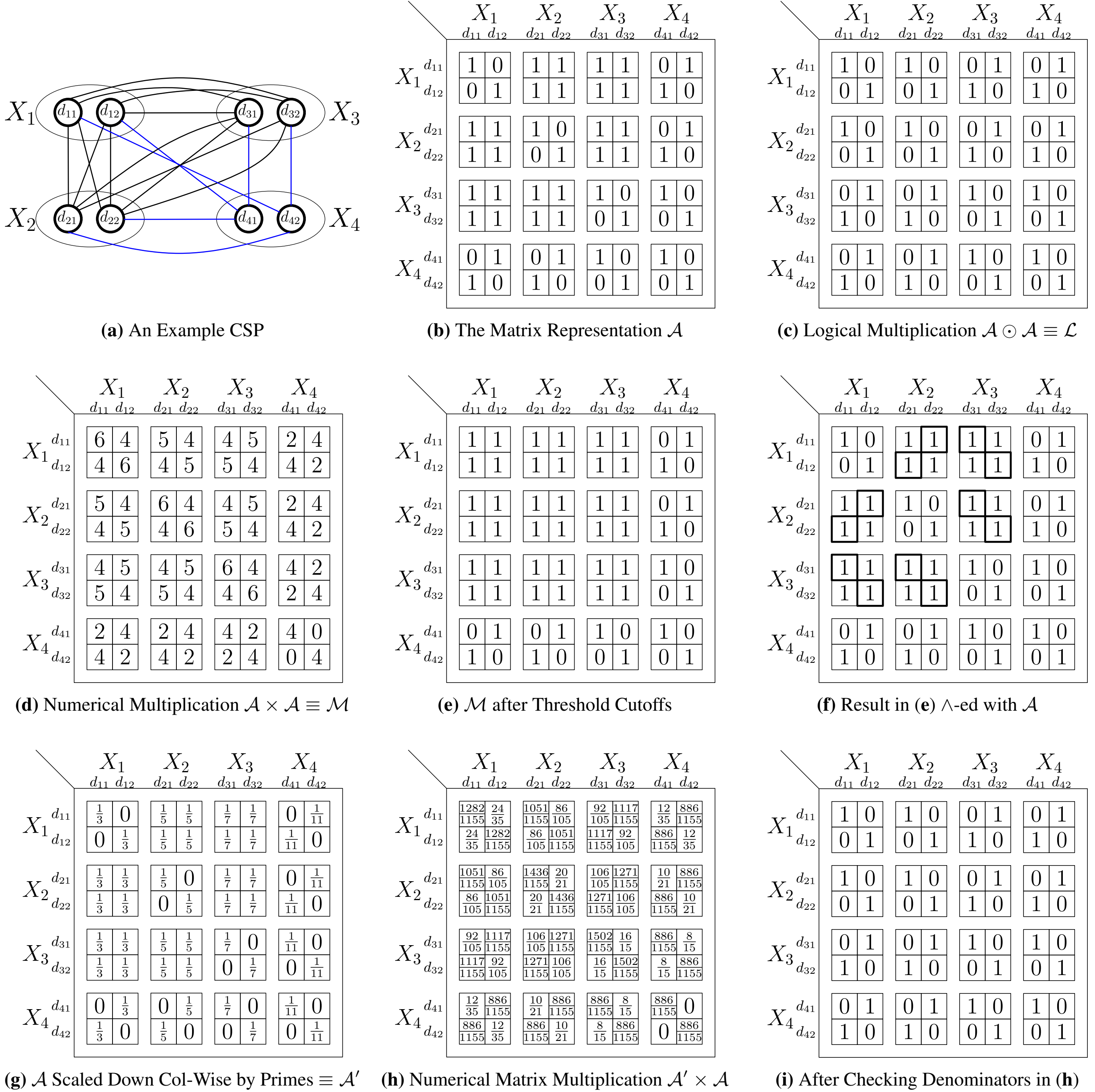
$$\sum_{i=1}^{K} \frac{s_i}{p_i} = \frac{\sum_{i=1}^{K} s_i \prod_{j \neq i} p_j}{\prod_{i=1}^{K} p_i}$$

This quantity is already in its irreducible form. This conclusion can be made based on the following arguments. All of the factors $p_1, p_2 \ldots p_K$ in the denominator are primes. So, they are the only potential common factors with the numerator. But any p_i appears in all but one of the terms in the numerator. Specifically, this term is $s_i \prod_{j \neq i} p_j$. Since p_i appears in all other terms, it divides them. So, p_i would divide the entire numerator if and only if it divides the term $s_i \prod_{j \neq i} p_j$. However, since p_i is also a prime, it clearly does not share any factors with any of the other primes not equal to it. The only remaining possibility for a common factor is if s_i is a multiple of p_i. This too cannot be the case because $s_i < p_i$. Put together, this proves that the irreducible form of the summation has the denominator equal to the product of all the primes. It is also easy to observe that, when one or more of the s_i are zero, then the denominator of the irreducible form of the summation will not contain the corresponding p_i in its factorization. □

Using Prime Numbers to Bridge the Gap

The result presented in the previous section can be used to bridge the gap between logical operations on constraints and numerical operations on regular matrices for entire CSPs. Let $\mathcal{A}$ be the matrix representation of a given binary CSP. We observe that the regular numerical multiplication $\mathcal{A} \times \mathcal{A}$ yields a matrix $\mathcal{M}$ with the entry $\mathcal{M}_{[X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv}]}$ being equal to the total number of supports for $(X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})$ over the domains of all other variables. Here, $1 \leq u \leq |D_i|$ and $1 \leq v \leq |D_j|$. Let us denote the support set of $(X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})$ in X_k's domain by $S_k(X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})$, and use $s_k(X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})$ to denote the cardinality $|S_k(X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})|$. We know that $\mathcal{M}_{[X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv}]} = \sum_{k=1}^{N} s_k(X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})$.

In PC, the logical multiplication $\mathcal{A} \odot \mathcal{A}$ is required to yield a matrix $\mathcal{L}$ with $\mathcal{L}_{[X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv}]}$ equal to 1 whenever $(X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})$ has at least one support in every other variable X_k's domain; and 0 otherwise. This means that $\mathcal{L}_{[X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv}]}$ requires $s_k(X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})$ to be non-zero for all k, whereas $\mathcal{M}_{[X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv}]}$ simply sums over $s_k(X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})$ for all k. Lemma 1 can be used to simulate the requirements of $\mathcal{L}$ using regular numerical multiplication if the individual $s_k(X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})$



(a) An Example CSP **(b)** The Matrix Representation $\mathcal{A}$ **(c)** Logical Multiplication $\mathcal{A} \odot \mathcal{A} \equiv \mathcal{L}$

(d) Numerical Multiplication $\mathcal{A} \times \mathcal{A} \equiv \mathcal{M}$ **(e)** $\mathcal{M}$ after Threshold Cutoffs **(f)** Result in **(e)** $\wedge$-ed with $\mathcal{A}$

(g) $\mathcal{A}$ Scaled Down Col-Wise by Primes $\equiv \mathcal{A}'$ **(h)** Numerical Matrix Multiplication $\mathcal{A}' \times \mathcal{A}$ **(i)** After Checking Denominators in **(h)**

Figure 5: Illustrates the working of our algorithm. The example in (**a**) shows 4 variables with domain sizes of 2 each. The edges indicate consistent combinations of values. For clarity, X_4's edges are in blue. (**b**) shows the matrix representation $\mathcal{A}$. (**c**) shows the logical multiplication $\mathcal{A} \odot \mathcal{A}$ as required by PC. (**d**) shows the regular numerical multiplication $\mathcal{A} \times \mathcal{A}$. Note that $\mathcal{M}_{[X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv}]}$ is the total number of supports for that combination across every other variable's domain, *including* X_i and X_j. Since $X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})$ requires at least one support in *every* variable's domain, the entries in $\mathcal{M}$ that are less than N can be immediately converted to 0 in an attempt to retrieve a (still incorrect) logical representation as shown in (**e**). The entries in (**e**) do not match the intended result in (**c**). Further, even $\wedge$-ing the outcome in (**e**) with $\mathcal{A}$ in order to tighten the constraints as shown in (**f**) does not yield the intended result. The dark squares in (**f**) emphasize the incorrect entries produced by our first attempt in comparison to (**c**). (**g**) shows the column-wise scaled-down version of $\mathcal{A}$ using the primes $3, 5, 7$ and 11 - all greater than the domain size 2 - associated with the variables X_1, X_2, X_3 and X_4, respectively. (**h**) shows the outcome of the regular numerical multiplication $\mathcal{A}' \times \mathcal{A}$. (**i**) shows the result of examining the denominators in each of the entries of (**h**). If the denominator equals the product of the primes $3 \times 5 \times 7 \times 11 = 1155$, then the corresponding entry is set to 1; and set to 0 otherwise. The outcome of doing this matches the intended result in (**c**).

for all k are scaled down by an appropriately chosen unique prime number $p_k > D$.[5]

Overall, the idea is to associate a unique prime number p_i with each of the variables X_i and use them to scale down the entries of the matrix $\mathcal{A}$ appropriately. In checking whether $\mathcal{A} \odot \mathcal{A} = \mathcal{A}$, we can choose to scale down the columns of $\mathcal{A}$ on the left-hand side of the $\odot$, or the rows of $\mathcal{A}$ on the right-hand side of the $\odot$. Figure 5 shows how the entries of a matrix $\mathcal{A}$ representing a CSP are scaled down column-wise using prime numbers corresponding to the column variables. Let us denote the scaled-down version of $\mathcal{A}$ by $\mathcal{A}'$. $\mathcal{L}$ can be obtained by computing $\mathcal{A}' \times \mathcal{A}$, and examining the denominator of each entry in the resulting matrix. Again, Figure 5 shows the result of multiplying $\mathcal{A}' \times \mathcal{A}$ and converting it into the logical matrix $\mathcal{L}$.

We also note that the matrix multiplications can be carried out using fractional forms of the numbers. These fractions will always be in their irreducible forms because of the use of primes.

Algorithm 1: Verify-Path-Consistency

The algorithm for verifying PC. Here, N is the total number of variables, and D is the size of the largest domain. The arithmetic time complexity of the steps in our algorithm is dominated by that of Step (4) which in turn is $O(N^{2.38}D^{2.38})$.

Input: A binary CSP $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$
Output: Report whether $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ is path-consistent

1. Build $\mathcal{A}$, the matrix representation of $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$
2. Choose N prime numbers greater than D
3. Build $\mathcal{A}'$ by scaling down each entry in $\mathcal{A}$ by the unique prime p_i associated with the column variable X_i
4. Compute $\mathcal{A}' \times \mathcal{A}$
5. Set each entry in $\mathcal{A}' \times \mathcal{A}$ with denominator equal to the product of all the primes to 1; set others to 0
6. Report $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ as being path-consistent if and only if the outcome of the previous step matches $\mathcal{A}$

Lemma 2. *Algorithm 1 correctly verifies PC and uses $O(N^{2.38}D^{2.38})$ arithmetic operations.*

Proof. We know that $\mathcal{M} = \mathcal{A} \times \mathcal{A}$, and that any specific entry $\mathcal{M}_{[X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv}]} = \sum_{k=1}^{N} s_k(X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})$. By Lemma 1, if we have primes $p_1, p_2 \ldots p_N$ with $p_k > s_k(X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})$, then $\sum_{k=1}^{N} \frac{s_k(X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})}{p_k}$ will have a denominator indicative of whether all the individual terms are non-zero. This is exactly what is required for verifying PC. Further, Step (3) of Algorithm 1 achieves this scaling by associating a unique prime number with each variable, hence completing the argument for correctness. As for the arithmetic complexity of Algorithm 1, Steps (3), (5) and (6) run in $O(N^2D^2)$, and the total running time is dominated by the multiplication in Step (4) which in turn is $O(N^{2.38}D^{2.38})$. □

[5] p_k is just required to be greater than $s_k(X_i \leftarrow d_{iu}, X_j \leftarrow d_{jv})$; but since the latter term is guaranteed to be at most D, p_k chosen to be greater than D satisfies the requirement.

Arithmetic vs Bit Complexity

In the above algorithm, all primes are greater than D, and we use N such primes. Since there are $K/\log K$ primes $\leq K$ (Newman 1980), and we require N primes greater than D, the size of the largest prime is approximately $O((N + \frac{D}{\log D}) \log(N + \frac{D}{\log D}))$. This requires a bit representation of size $O(\log(N + D))$. Now, the numerators and denominators can grow up to the product of all these primes and might therefore require bit representations of size $O(N \log(N + D))$. Thus, if we count the number of bit operations, we get a total complexity exceeding $O(N^3D^{2.38})$. Nonetheless, if we count the number of machine cycles on an architecture that can accommodate the required bit representations (which is reasonable in many practical problems), then multiplications can be carried out within a few constant machine cycles (Patterson and Hennessy 2008) (p.235), in which case, the arithmetic complexity of $O(N^{2.38}D^{2.38})$ would be more relevant.

Discussions

The results presented in this paper defeat the arguments made in many previous articles and standard literature on PC. It might be possible to generalize our techniques for verifying higher levels of local-consistency. In order to verify whether a given CSP is $(k+1)$-consistent, we need to reason about k-dimensional matrices (because $(k+1)$-consistency introduces k-ary constraints). However, since numerical matrix multiplication of k-dimensional matrices also benefits from the optimization techniques used in fast matrix multiplication algorithms, a generalization of our approach is likely feasible.

Some (hitherto failed) attempts can also be made to reduce the time complexity of verifying PC even further. One such way might be to employ *fingerprinting* techniques from the Randomized Algorithms community. Checking whether the product of two given $N \times N$ matrices equals a third given $N \times N$ matrix can be done in $O(N^2 \log N)$ time using randomized algorithms (Freivalds 1977). However, this approach is not directly applicable for our purposes because what we are really interested in is the logical product $\mathcal{A} \odot \mathcal{A}$ instead of the numerical product $\mathcal{A} \times \mathcal{A}$. Furthermore, this approach cannot even be used directly in conjunction with the prime numbers scaling technique, as we are interested in examining only the denominators of the resulting matrix while the randomized algorithm is tailored to identity checking. Nonetheless, a connection to fingerprinting techniques used in the Randomized Algorithms community might result in important implications for improving the lower bounds for verifying PC and higher levels of local-consistency.

We also hope to inspire a new class of algorithms for actually *establishing* PC and/or higher levels of local-consistency for a given CSP. We surmise that such a new class of algorithms will conceivably make use of the methodologies discussed in this paper. We would also like to inspire a revisit of some well known and important complexity results about PC, Singleton-Arc-Consistency or local-consistency, that are currently taken for granted.

Conclusions

In this paper, we presented an efficient algorithm for verifying PC that runs in time $O(N^3 D^{2.38})$. We also presented an algorithm with $O(N^{2.38} D^{2.38})$ arithmetic time complexity. Here, N is the number of variables, and D is the size of the largest domain in a given binary CSP. The previous conjecture on the lower bound for verifying PC was $\Omega(N^3 D^3)$. The reduction from D^3 to $D^{2.38}$ was based on a straightforward application of fast matrix multiplication algorithms well known in the Theory community. The reduction of the arithmetic complexity from N^3 to $N^{2.38}$ was also inspired by fast matrix multiplication algorithms, but was not directly amenable to those techniques. We used the properties of primes to bridge the gap, and cast logical operations on the matrix representations of constraints as numerical operations on regular matrices.

This paper therefore defeats the arguments used in the proofs of various results published throughout the years. We expect our techniques to be generalizable to the task of verifying higher levels of local-consistency. We also hope to inspire a new class of algorithms for establishing PC as well as higher levels of local-consistency with lower time complexities. This in turn will have crucial implications on the efficiency of solving CSPs in general.

Acknowledgments

This paper is based upon research supported by a MURI under contract/grant number N00014-09-1-1031. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies or the U.S. government.

References

Bessiere, C., and Debruyne, R. 2004. Theoretical analysis of singleton arc consistency. In *Proceedings of the 16th European Conference on Artificial Intelligence*, 20–29.

Coppersmith, D., and Winograd, S. 1987. Matrix multiplication via arithmetic progressions. In *Proceedings of the 19th annual ACM Symposium on Theory of Computing*, 1–6. New York, NY, USA: ACM.

Dechter, R. 2003. *Constraint Processing*. The Morgan Kaufmann Series in Artificial Intelligence. Elsevier Science.

Deville, Y.; Barette, O.; Barbe, P. S.; and Hentenryck, P. V. 1997. Constraint satisfaction over connected row convex constraints. In *Artificial Intelligence*, 405–411.

Freivalds, R. 1977. Probabilistic machines can use less running time. In *International Federation for Information Processing*, 839–842.

Gries, D., and Misra, J. 1978. A linear sieve algorithm for finding prime numbers. *Commun. ACM* 21(12):999–1003.

Kumar, T. K. S., and Russell, S. J. 2006. On some tractable cases of logical filtering. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling*, 83–92.

Kumar, T. K. S. 2005. On the tractability of restricted disjunctive temporal problems. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling*, 110–119.

Mohr, R., and Henderson, T. C. 1986. Arc and path consistence revisited. *Artificial Intelligence* 28(2):225–233.

Newman, D. J. 1980. Simple analytic proof of the prime number theorem. *Amer. Math. Monthly* 87(9):693–696.

Patterson, D. A., and Hennessy, J. L. 2008. *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Strassen, V. 1969. Gaussian elimination is not optimal. *Numerische Mathematik* 14(3):354–356.

Zhang, Y., and Freuder, E. C. 2004. Tractable tree convex constraint networks. In *Proceedings of the 19th American Association for Artificial Intelligence*, 197–202. AAAI Press.