Parallelism and Domain Dependence in Constraint Satisfaction

Paul R. Cooper and Michael J. Swain

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 255
December, 1988

This work was supported by a Canadian NSERC post-graduate scholarship, by the Air Force Systems Command, Rome Air Development Center, Griffis Air Force Base, New York 13441-15700 and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F30602-85-C-0008. The latter contract supports the Northeast Artificial Intelligence Consortium (NAIC).
Abstract

This paper discusses how better arc consistency algorithms for constraint satisfaction can be developed by exploiting parallelism and domain specific problem characteristics. A massively parallel algorithm for arc consistency is given, expressed as a digital circuit. For a constraint satisfaction problem with $n$ variables and $a$ labels, this algorithm has a worst case time complexity of $O(na)$, significantly better than that of the optimal uniprocessor algorithm. An algorithm of intermediate parallelism suitable for implementation on a SIMD machine is also given. Analyses and implementation experiments are shown for both algorithms.

A method for exploiting characteristics of a problem domain to achieve arc consistency algorithms with better time and space complexity is also discussed. A detailed example is developed, and a general technique for expressing domain knowledge and using it to develop optimized arc consistency algorithms is described. The domain specific optimizations can be applied analogously to any of the arc consistency algorithms along the sequential/parallel spectrum.
Contents

1 Introduction 3

2 General Purpose Arc Consistency Algorithms 5
  2.1 Constraint Satisfaction: Problem Definition 5
  2.2 Uniprocessor Solution 5
  2.3 Massively Parallel Solution 6
    2.3.1 Algorithm 6
    2.3.2 Analysis 8
    2.3.3 Application Example: Graph Matching 10
  2.4 SIMD Solution 11
    2.4.1 Algorithm 11
    2.4.2 Analysis 12
    2.4.3 Application Experiment: Connection Machine Implementation 14
  2.5 Relating the Algorithms 14
  2.6 Limits to Parallelism 16
    2.6.1 Intrinsic Sequentiality 16
    2.6.2 Sub-linear-time Algorithms 17

3 Domain Specific Parallel Constraint Satisfaction 18
  3.1 Example Problem: Matching Descriptions of Tinker Toys 18
    3.1.1 Mapping the Problem to Constraint Satisfaction 18
    3.1.2 Domain Specific Meta-Constraints 19
    3.1.3 Application Example: Connectionist Simulation of AC Chip, Specialized for Tinker Toy Recognition 22
  3.2 Domain Specific Performance Improvements for Arbitrary Domains 26
    3.2.1 Predicate Specification 26
    3.2.2 Circuit Optimization 28
    3.2.3 Analysis and Performance 30
    3.2.4 Mapping the Optimizations to AC-4 33

4 Conclusions 34

A Appendix: The Language Algol* 37

B Appendix: A Space-Efficient AC-4 37
C Appendix: Optimized AC-4 Algorithm 39

D Appendix: Fast Arc Consistency on a PRAM 41

E Appendix: Extra Minimizations for the Tinker Toy Domain 44
1 Introduction

Constraint satisfaction is an important technique used in the solution of many artificial intelligence problems. Examples range widely, from planning [Stefik, 1981] through low-level and high-level vision [Waltz, 1975; Davis and Rosenfeld, 1981], to known NP-complete problems such as map coloring. Efficient implementations of constraint satisfaction algorithms are thus important to AI. In this paper, we examine efficient parallel methods for computing constraint satisfaction. We focus in particular upon the parallel computation of arc consistency. Arc consistency is a well-known algorithm in the network consistency class of methods [Waltz, 1975; Montanari, 197; Mackworth, 1977; Freuder, 1978]. These algorithms compute and propagate local consistency or compatibility, eliminating local inconsistencies which can never take part in a global solution.

The optimal single-processor algorithm for doing arc consistency can take up to $O(a^2n^2)$ time steps for a problem with $n$ variables (or nodes) and $a$ labels [Mohr and Henderson, 1986]. In contrast, we describe a parallel algorithm for arc consistency which is virtually instantaneous. In many cases, only a small constant number of time steps are required, and even in the worst case only $O(na)$ time is required.

Substantial gains in the performance of arc consistency algorithms can be gained in at least two ways. First, processors can be traded off for time, and completely general purpose parallel algorithms with significantly improved time complexities can be developed. We show two such algorithms, each of which corresponds to a different choice of spatial parallelism, and each of which is faster than the optimal uniprocessor algorithm. The first algorithm, corresponding to the unit/value choice for parallelism, is specified as massively parallel digital circuit. This algorithm requires $O(a^2n^2)$ processing units. The second algorithm is somewhat less parallelized, and is more suited to implementation on a highly parallel SIMD machine such as the Connection Machine.

But there is also a second effective way to gain performance increases for constraint satisfaction algorithms. In this approach, completely general purpose constraint satisfaction schemes are not used. Instead, the algorithms are optimized to perform in a specific domain. When a problem domain is known in advance, it is possible to exploit characteristics of the classes of problem instances in the domain in order to simplify the algorithm. In AI this approach has often been used to generate unprincipled heuristic solutions whose generality is ill-defined. But more principled approaches to domain constraint exploitation also exist. For example, Mackworth et al [1985] show how hierarchic domains can be exploited to yield a more efficient constraint satisfaction process. In our work, the time complexity of the parallel algorithms is already probably as low as can reasonably be expected. Thus, for parallel algorithms the exploitation of domain dependence takes the form of reducing the amount of space (or number of processing elements) required. In the paper, we describe an example domain of problems, and demonstrate the kinds of improvements that can be made by exploiting the domain characteristics. Then we show how the process works in general, and describe an algorithm which creates optimized arc consistency circuits given any domain description. Significant reductions in space requirements can be achieved in this way, and these optimizations can be translated to time speedups for less parallel algorithms.

The structure of the paper reflects these two basic mechanisms for achieving efficient
constraint satisfaction by arc consistency. In the first part of the paper, general purpose arc consistency algorithms are discussed, beginning with a review of the best sequential algorithms and their development. Both our parallel algorithms are then described, along with their associated analysis and example applications. This part of the paper concludes with a discussion of the limits of parallelism for arc consistency, including a description of a theoretically interesting sub-linear-time algorithm which requires exponentially many processors. The second part of the paper describes the use of domain characteristics to generate optimized parallel arc consistency algorithms.
2 General Purpose Arc Consistency Algorithms

2.1 Constraint Satisfaction: Problem Definition

We first review the constraint satisfaction problem as classically formulated [Mackworth, 1977; Freuder, 1978; Hummel and Zucker, 1983; Mohr and Henderson, 1986]. Formally, a constraint satisfaction problem (CSP) is defined as follows: Given a set of \( n \) variables each with an associated domain and a set of constraining relations each involving a subset of the variables, find all possible \( n \)-tuples such that each \( n \)-tuple is an instantiation of the \( n \) variables satisfying the relations. We consider only those CSPs in which the domains are discrete, finite sets and the relations are unary and binary.

In general, a constraint satisfaction problem may be NP-complete. Many algorithms, such as search with backtracking, have been used in the solution of CSPs. Network consistency methods have been developed as a way to improve the efficiency of these search algorithms. They work by the elimination of locally inconsistent partial solutions, in a preprocessing or filtering phase. Network consistency algorithms are thus not guaranteed to find a unique solution to a CSP, and subsequent processing may still be required. A \( k \)-consistency algorithm removes all inconsistencies involving all subsets of size \( k \) of the \( n \) variables. In particular, node and arc consistency algorithms detect and eliminate inconsistencies involving \( k = 1 \) and 2 variables, respectively. A review of network consistency algorithms and their use in the solution of constraint satisfaction problems can be found in Mackworth [1987].

A typical arc consistency problem consists of a set of variables, a set of possible labels for the variables, a unary predicate, and a binary predicate with an associated constraint graph. For each \( i \) of the \( n \) variables, the unary predicate \( P_i(x) \) defines the list of allowable labels \( x \) taken from the domain of the variables. For each pair of variables \((i, j)\) in the constraint graph, the binary predicate \( Q_{ij}(x, y) \) defines the list of allowable label pairs \((x, y)\). To compute the \( n \)-tuples which satisfy the overall problem requires that the local constraints are propagated among the variables and arcs until no inconsistencies remain.

2.2 Uniprocessor Solution

In this section, we illustrate the computation of node and arc consistency with a discussion of algorithms for a uniprocessor. The most efficient algorithm for achieving arc consistency on a uniprocessor is AC-4 [Mohr and Henderson, 1986]. Predecessors to AC-4 include the original Waltz filtering application [Waltz, 1975] and AC-3 [Mackworth, 1977; Mackworth and Freuder, 1985]. As Mohr and Henderson show [1986], AC-4 is the optimal algorithm for the uniprocessor case.

Achieving node consistency is easy. One simply eliminates all labels \( x \) at a node \( i \) which do not conform to the unary predicate \( P_i(x) \). This can be done with a single pass over the nodes. One can make an analogous pass over the arcs, and eliminate labels which do not take part in consistent label-pairs on that arc (ie. do not conform to \( Q_{ij}(x, y) \)). However, this does not guarantee that the network is arc consistent. This is because eliminating a label from a node may make neighboring nodes inconsistent. Therefore, the process of checking
for arc consistency must be iterated until no more eliminations occur, and the network is arc consistent. This is the 'relaxation' phase of the computation, when the constraints are propagated from node to node.

The simplest idea for achieving arc consistency sequentially is to iterate a process which checks each arc for consistency, until no further changes occur. This process does much unnecessary work. As stated above, elimination of a label from a node can only affect the consistency of labels at neighboring nodes. Therefore, only neighboring nodes need be checked, not every node (or arc) in the network. The algorithm AC-3 exploits this fact. AC-4 exploits an additional observation. That is, the elimination of a particular label at a node often does not make labels at even neighboring nodes inconsistent. This is because as Mohr and Henderson put it, arc consistency is based on the notion of support. To be more precise, a label \( x \) at some node \( i \) is arc consistent if it has some minimum of support at each of its neighboring nodes. The label \( x \) only becomes inconsistent when the last of its support at a neighbor (eg. the last consistent label) is eliminated. AC-4 makes this notion explicit with the use of counters which measure support for a label. Relevant counters are decremented when a label is eliminated, and search is in this way minimized. This strategy yields an optimal algorithm for a uniprocessor, with time complexity \( O(n^3a^2) \) [Mohr and Henderson, 1986]. The space complexity of AC-4 is also \( O(n^3a^2) \).

The notion that a label has lost sufficient support from its neighbors and is now inconsistent can be stated independent of a sequential or parallel control strategy for its implementation. This notion was expressed by Hummel and Zucker [1983] as the label discarding rule: discard a label \( x \) at a node \( i \) if there exists a neighbor \( j \) of \( i \) such that every label \( y \) currently assigned to \( j \) is incompatible with \( x \) at \( i \), that is, \( \neg Q_{ij}(x,y) \) for all \( y \) in the domain of \( j \). In the next section, we will give an algorithm which applies the label discarding rule in a parallel fashion.

We are not the first to study parallel algorithms for arc consistency. Rosenfeld [1975] described a parallel algorithm for a class of machines called web automata. Henderson and Samal [1987] designed shared memory parallel algorithms for arc consistency. We reduce the 'processors' to their simplest form: latches and gates, and show how a single chip can compute arc consistency in a most highly parallel manner.

### 2.3 Massively Parallel Solution

#### 2.3.1 Algorithm

In this section, we describe a fast parallel algorithm for computing arc consistency. The algorithm is specified as a massively parallel digital circuit we call the Arc Consistency (AC) Chip. The design is derived from an earlier purely connectionist algorithm [Cooper, 1988], and is based upon the use of the unit/value principle [Barlow, 1972; Ballard, 1986]. In massively parallel or connectionist designs using the unit/value principle, very simple processing elements are assigned to every single value which may take part in the computation. Only simple messages are passed between the processing elements or units, and it is the pattern of connections between the units that encodes the information necessary to solve the problem. In the case of the AC chip, the processing elements are logic gates (and,
or and not gates) and memory elements, and the messages are digital signals. The result is a digital circuit amenable to implementation in VLSI [Mead, 1988].

In essence, the AC chip consists of two arrays of JK flip-flops and suitable amounts of combinational circuitry. The most important part of the design is the representation for the two constraint predicates $P_i(x)$ and $Q_{ij}(x, y)$. Adopting the unit/value principle, we assign one memory element to represent every possible value of $P_i(x)$ and $Q_{ij}(x, y)$. (As will be seen, JK flip-flops are used as the memory elements because of their convenient reset characteristics). To allow the hardware to compute any arc consistency problem, the two arrays must be able to represent any given $P_i(x)$ and $Q_{ij}(x, y)$ of sizes bounded by $n$ and $a$.

The first (node) array will be designated the labeling array. It consists of $na$ flip-flops we call $u(i, x)$. The array of flip-flops serves two purposes. First, it represents all possible solutions to the problem, and eventually after the constraint satisfaction is completed, the set of of solutions which are arc consistent. That is, the array explicitly represents the nodes of the CSP and all possible labels for each node. The second purpose of the labeling (or node) array is to represent and apply the unary constraint predicate $P_i(x)$. This is accomplished by initializing each flip-flop $u(i, x)$ of the labeling array to the value of the corresponding element of the unary constraint predicate $P_i(x)$. That is, if $x$ is a valid label at node $i$, then the the flip-flop $u(i, x)$ is initialized to on. Thus initially at least one flip-flop which are on all correspond to labelings of a node which are valid considering only the local (unary) constraint at that node. Note that all flip-flops are initialized. The final answer to the computation (which labels are arc consistent at each node) will be contained in this array at the end of the computation.

The second (arc) array consists of $n a^2 (n - 1)$ flip-flops we designate $v(i, j, x, y)$ which are initialized to conform to the arc constraint predicate $Q_{ij}(x, y)$, if $i$ is adjacent to $j$ in the constraint graph, and to 1 (no constraint) otherwise. Note that the arc array is static; it does not change throughout the computation.

The basic structure of the two arrays of flip-flops is shown in Figure 1.

It remains only to develop combinational circuitry that causes the flip-flop representing the label $x$ at node $i$ to be reset to zero if it becomes inconsistent. In other words, we want circuitry which implements the label discarding rule at each node. Arc consistency is then achieved when a limiting label set is obtained.

To implement the arc consistency label discarding rule, the combinational circuitry is designed so that the K (reset) input of the JK-flip-flop $u(i, x)$ receives the value:

$$\text{reset}(u(i, x)) = \bigwedge_{j=1, j \neq i}^a \bigvee_{y=1}^a (u(j, y) \land v(i, j, x, y))$$

The J input of each JK-flip-flop is tied to 0. A partial circuit diagram for this equation is given in Figure 2. This figure show the reset circuitry for one flip-flop in the labeling array $u(i, x)$. In the figure, the entire labeling array is present, but only the part of the arc table $v(i, j, x, y)$ useful for this node is drawn. An analogous circuit for each element of the labeling array completes the whole circuit.

To interpret the equation and circuit, consider first the inner term $u(j, y) \land v(i, j, x, y)$ for a particular case of $u(i, x)$. The fact that $v(i, j, x, y)$ is true tells us that there is an
Figure 1: Unary and Binary Constraint Tables

arc between $i$ and $j$, and $(x,y)$ is a consistent label pair for this arc. We already know that $u(i,x)$ is true; anding with $u(j,y)$ checks that the other end of the arc has a valid label. Point A on the circuit diagram in Figure 2 shows where this term is computed. (In Figure 16, a particular and gate of this type is referred to as $A(i,j,x,y)$).

At this point, as far as node $i$ is concerned, $x$ is a label consistent with node neighbor $j$'s label $y$. The $\forall_{y=1}^{m}$ simply ensures that at least one label $y$ on neighboring node $j$ is consistent. This function has been computed after the or gate at point B in Figure 2. (In Figure 16 these or gates or designated $B(j)$).

Label $x$ on node $i$ is thus consistent with its neighbor $j$. But what about node $i$'s other neighbors? The $\wedge_{j=1, j \neq i}^{n}$ ensures that there is arc consistency among all node $i$'s neighbor's. The and gate at C in Figure 2 ensures this.

If the signal is on at point C, that means that label $x$ is consistent for node $i$ - therefore, the flip-flop need not be reset. Thus the not gate.

To reverse the analysis, if some node $j$ does not have a consistent labeling, then at point B, the signal will be off. The and will fail, so the signal at C will also be 0, and then the not gate will cause flip-flop $u(i,x)$ to be reset.

2.3.2 Analysis

Correctness: To begin with, recall that we are interested in discarding labels, an operation which corresponds to resetting on flip-flops to 0. Furthermore, since the J input of each
Figure 2: Partial Circuit Diagram for the AC Chip
JK-flip-flops in the node array are always tied to zero, the flip-flops can only ever be reset to 0, never set. Once they are off they must stay off, so the whole process is clearly monotonic. Therefore, all we need to show for correctness is to show that the network correctly applies the label discarding rule. If the network discards labels when they should be discarded, and does not discard them when they should be kept, then it implements the label discarding rule correctly.

The label discarding rule can be formally expressed as follows:

\[ \exists j (j \neq i) \forall y [u(j, y) \land v(i, j, x, y) = 0] \]

But this expression is equivalent to

\[ \bigwedge_{j=1, j \neq i}^{n} \bigvee_{y=1}^{a} (u(j, y) \land v(i, j, x, y)) = 0 \]

or

\[ \neg \bigwedge_{j=1, j \neq i}^{n} \bigvee_{y=1}^{a} (u(j, y) \land v(i, j, x, y)) = 1 \]

which is just the condition under which \((i, x)\) is reset. Therefore, the network correctly discards labels when it should. The converse follows from negating the above equations.

**Complexity** The circuit requires \(na\) JK-flip-flops for the labeling array, and \(a^2n(n-1)\) flip-flops for the arc array. From Figure 2 we see that there is an and gate for every flip-flop in the arc array, so \(a^2n(n-1)\) 2-input and gates are required for this purpose. For each of the \(na\) flip-flops in the labeling array there is \(n-1\) or gates required, each taking \(a\) inputs - a total of \(an(n-1)\) or gates. Finally, there are \(na\) and and not gates (nand gates), each taking \(n-1\) inputs. There are also \(O(a^2n^2)\) wires.

The worst case time complexity of the network occurs when only one JK-flip-flop is free to reset at a time. So if propagation through the and and or gates is considered instantaneous, the worst case time complexity is \(na\). If a logarithmic time cost is assigned to the large fan-in and and or gates the worst case time complexity is \(O(a \log(a)n \log(n))\).

Note that if the node and arc arrays must be initialized serially, loading them takes more time \(O(a^2n^2)\) steps) than executing the algorithm. For almost all applications of constraint satisfaction the binary predicate \(Q_{ij}(x, y)\) can be specified with less than \(O(a^2n^2)\) information, and so instead of the arc array a circuit could be built that supplies the correct values to the and gates without needing so many memory elements to fill. An application in which this is true is graph matching, which we describe in the next section.

### 2.3.3 Application Example: Graph Matching

Graph matching can be defined as a constraint satisfaction problem. General graph matching requires k-consistency [Freuder, 1978] (and is NP-complete, in fact). With just arc consistency, a restricted yet still interesting class of graphs may be matched. Furthermore, the effectiveness of matching graphs by constraint satisfaction with only arc consistency can
be enhanced if the graphs are labeled. This kind of restricted matching of labeled graphs is particularly suited to the visual indexing problem [Cooper, 1988]. In this problem, labeled graphs are used to represent structurally composed objects. The constraint satisfaction process is used only to filter recognition candidates, and the few graphs not discriminable with the limited power of arc consistency can be addressed in other ways.

The key to framing any problem as a CSP is to determine what is to be the set of variables or nodes for the problem, and what is to be the set of labels for each node. Then, the unary and binary constraint predicates must be defined. Labeled graph matching can be framed as a constraint satisfaction problem as follows. Suppose we are attempting to match graph A to graph B. First, the nodes or variables in the constraint satisfaction problem are taken to represent the vertices of one of the graphs, say graph A. The labels for the nodes represent the vertices of the other graph, graph B. Then, a unique match between graph A and graph B is represented by a unique labeling in the constraint satisfaction problem. To ensure an arc consistent labeling is computed, we also need to define the constraints. The unary constraint is that the labels of corresponding vertices be the same. The binary (arc) constraint ensures that the connectedness between pairs of corresponding vertices are the same. In other words, if there is an edge between 2 vertices in one graph, the arc constraint requires there to be an edge between the corresponding vertices in the other graph. In this section, we describe without loss of generality the matching of undirected graphs.

So, for the graph matching problem:

\[ P_i(x) = (\text{label}(i) = \text{label}(x)) \]

and

\[ Q_{ij}(x, y) = (\text{adjacent}(i, j) = \text{adjacent}(x, y)) \]

For the graph matching problem the number of possible labels equals the number of vertices so \( a = n \).

2.4 SIMD Solution

Here we present a parallel algorithm for constraint satisfaction that requires \( na \) processors and \( O(na \log(na)) \) time. The algorithm is Single Instruction, Multiple Data (or SIMD), and requires only local memory and modest communication requirements. Experiments have shown that a constraint satisfaction problem of size \( na = 32K \) will run in 16 seconds on the Connection Machine™ multiprocessor.

2.4.1 Algorithm

The algorithm, called ACP, is given in Figure 3. It, like the AC Chip and AC-4 algorithms from which it was derived, relies on the notion of support. In ACP, there is a processor for every member of the labeling array. Each such member of the labeling array will be referred to as a candidate label, and corresponds to one possible label at one node. At each

\(^1\)Connection Machine is a registered trademark of Thinking Machines Corporation.
processor there is a set of counters which denote the support given by each other node to the candidate label.

ACP has four phases:

1. Initialize the processor array according to the values of $P_i(x)$.
2. Accumulate support.
3. Iteratively remove inconsistent candidate labels.
4. Read answer back from the machine.

One candidate label is removed at a time. It broadcasts its identity to all the other processors which, in parallel, check to see if that candidate label supported them. If it did, the appropriate counter is decremented. If a counter becomes zero, the *dying* flag is set, indicating that the processor has become eligible to broadcast its identity as an inconsistent pair.

Unlike Mohr and Henderson's AC-4, there are no support lists, simply because they are unnecessary. (They aren't necessary in AC-4 either, see Appendix B.) The List data structure of AC-4 is also not present in ACP; it simply corresponds to the set of processors for which the variable *dying* is true.

The algorithm is presented in a parallel Pidgin Algol based on C*, a language developed by Thinking Machines Corporation for programming the Connection Machine [Rose and Steele, 1987]. We call the modified Pidgin Algol Algol*. The centerpiece of Algol* is the poly data structure. The language assumes a processor for every element of data structure with the poly attribute. Expressions involving poly data structures are evaluated concurrently in every processor associated with the data structure. Algol* is more fully described in A.

2.4.2 Analysis

Complexity Since there are $na$ candidate labels and one inconsistent candidate label is removed per iteration, the while loop commencing at line 31 will be executed at most $na$ times. If the individual Connection Machine operations are considered to take $O(1)$ the complexity of the algorithm is $O(na)$. In fact, there is a logarithmic component of complexity associated with the broadcast network, but this is fixed for a given machine. Including this complexity term gives a total complexity of $O(an \log an)$.

This algorithm cannot take advantage of a sparse constraint graph (e.g. planar graphs). A modification of the algorithm optimized for sparse constraint graphs is described in [Swain, 1988].

Correctness The proof of correctness for AC-4 given by Mohr and Henderson [1986] works for ACP as well. We repeat the proof here for ease of reference.

Step 1. By induction, each label deleted from $A_i$ is not admissible for any arc consistency solution: The label is removed if one of its counters goes to zero, so it has no more
procedure acp(P, Q, result) begin
  i, y, ia, xa, jd, yd: integer;
candidate.label: poly array[1..n, 1..a] of record
counter: array[1..n] of integer;
  alive, dying: boolean;
i, x: integer;
end;

(* Initialize data structures *)
dying := false;
for ia := 1 to n do
  for xa := 1 to a do
    begin
      candidate.label[ia, xa].alive := P(ia, xa);
      candidate.label[ia, xa].dying := false;
    end

with candidate.label do
  begin
    (* Accumulate support *)
    if alive do
      for j := 1 to n do
        begin
          for y in P do
            if Qr(j, y) then counter[j] := counter[j] + 1;
          if counter[j] = 0 then dying := true;
        end
    (* Remove inconsistent variable-label pairs *)
    while dying do
      begin
        (jd, yd) := one of (i, x);
        if Qr(jd, yd) then
          begin
            counter[jd] := counter[jd] - 1;
            if counter[jd] = 0 then
              if alive then dying := true;
          end
        candidate.label[jd, yd].alive := false;
      end
    (* Find out which processors are still alive *)
    result := false;
    while alive do
      begin
        (ia, xa) := one of (i, x);
        result[ia, xa] := true;
      end
  end
end acp

Figure 3: The algorithm ACP
corresponding labels at one edge: by induction all the previously removed labels could not belong to any solution; so this one cannot belong to any solution.

Step 2. The result is arc consistent: for all \((i, j)\), for all labels \(b\) for \(i\), we have \(\text{Counter}(i, j, b) > 0\) so \(b\) has a corresponding label node \(j\); therefore, ACP builds an arc consistent solution.

Step 3. From Steps 1 and 2 we conclude that the algorithm builds the largest arc consistent solution.

2.4.3 Application Experiment: Connection Machine Implementation

The Connection Machine 2 is ideal for implementing ACP. It has a very large number of processors, 32K (1K = 1024), each with 64K bits of memory. The algorithm was implemented in C.*

The problem that the algorithm was tested on was similar to the example used in [Samal and Henderson, 1987]. For \(i, j = 1, \ldots, n\) and \(x, y = 1, \ldots, a\) we have \(P(i, x) = \text{true}\) and

\[
Q(i, x, j, y) = \neg((i - j)_{\text{n}} = 1 \land x > y).
\]

We assumed the constraint graph to be complete, although for this problem a sparser constraint graph could have been used. We did this to compare the sequential and parallel algorithms on a problem for which the parallel algorithm was well-suited. Problems with truly complete constraint graphs tend to be more complex than the one we used.

The experimental results are shown in Figure 4. The graph is a log-log plot, the slope of which gives the degree of the polynomial expression of complexity. The slope for the Connection Machine results is 0.992, with a coefficient of correlation \(r = 0.999\), showing an almost perfect fit to a straight line of slope 1, indicating a complexity of \(\Theta(na)\) with \(na\) processors.

After 32K processors the Connection Machine we used in these experiments ran out of real processors. One might expect the complexity to grow quadratically after this point, but the slope of the line from \(na = 32K\) to \(na = 64K\) is only 1.3. The explanation for the low growth in complexity lies in the overhead in decoding an instruction from the host. Because a processor simulating more than one virtual processor only needs to decode the instruction once for all virtual processors, it can simulate \(k\) processors in less than \(k\) times the time to simulate one processor. As \(k\) grows, the instruction decoding time becomes less important, but from \(k=1\) to \(k=2\) the effect is significant.

As expected, the slope of the log-log plot of the sequential run-times is 2, indicating a complexity of \(O(a^2n^2)\). The largest problem tried on the Connection Machine, \(na = 64K\), took 16.5 seconds whereas the estimated time on a Sun 3/260 would be 81 hours.

2.5 Relating the Algorithms

The AC-4 uniprocessor algorithm for arc consistency, the AC Chip massively parallel solution, and the ACP SIMD solution are all closely related. The algorithm is most explicitly expressed in the AC Chip solution - each element of the solution is a visible element of the
Figure 4: Parallel vs. sequential arc consistency.
chip. The less parallel versions of the solution all group some elements of the AC Chip solution at a processor or in a variable.

Consider the ACP algorithm of the previous section, compared to the AC Chip specification. In ACP, a processor is assigned to every element of the AC Chip labeling array. The functionality of the tree of gates in Figure 2 for the AC Chip solution is replaced by equivalent code at each node in the labeling array. In effect, the or gate in each row of Figure 2 is replaced by a counter maintained on each processor. When an element of the labeling array goes off, it broadcasts a message. Upon receipt of a message that an element has gone off, each still-active element of the labeling array decrements the appropriate support counter. If some counter at a processor goes to zero, that element of the labeling array has become inconsistent.

AC-4 can also be seen as a uniprocessor simulation of the functionality of the AC Chip representation. It uses support counters for each or gate of Figure 2 in a manner analogous to ACP. If a support counter falls to zero, that candidate label is marked inconsistent by adding it to a list of inconsistent candidate labels. The list is required because AC-4 must deal sequentially with events, and must deal with the consequences of each candidate label becoming inconsistent one at a time. When a particular inconsistent label at a node is being processed, the neighbors of the node must also be visited in sequence, decrementing the appropriate counters if necessary.

2.6 Limits to Parallelism

2.6.1 Intrinsic Sequentiality

So far, we have seen parallel schemes for arc consistency that achieve faster worst case and expected case running times than more sequential algorithms. A natural question arises: how fast can constraint satisfaction be accomplished? It has long been recognized (e.g. [Hinton, 1977]) that there is an intrinsically parallel aspect to relaxation type computations. Consistency can be computed locally and in parallel, as we have seen. But there is also an intrinsically serial aspect to these computations, because a globally consistent answer cannot be found without constraint propagation and constraints may only propagate one arc per time step.

This latter suggestion of inherent sequentiality is borne out by the analysis of Kasif [1986] who showed that constraint satisfaction by arc consistency is log-space complete for P. P is the class of problems that require polynomial time for their solution, given a polynomial number of processors. This suggests that finding an algorithm for arc consistency with significantly sub-linear time complexity is unlikely, as we will explain. A class of problems with faster solutions is NC. NC is the class of problems solvable in logarithmic parallel time with polynomially many processors. Kasif's result shows that unless NC = P, a result thought as unlikely as P = NP [McKenzie and Cook, 1987], a logarithmic-time polynomial-processor algorithm for arc consistency does not exist. This also suggests that a linear lower bound for arc consistency will be extremely difficult to prove - it would be equivalent to proving that NC ≠ P.

Problems in NC are typically those thought of as having fast parallel solution, and those in P are typically regarded as being in some way intrinsically sequential. But it is
important to remember that these results analyze only worst case performance. Consider arc consistency for example. As discussed in the worst-case analysis of the AC Chip in Section 2.3.2, it is possible that only one node/label possibility is eliminated in every time step. It is easy to imagine a case where this is true - all that is required is a linear chain of constraint dependencies. But the reasonable expectation that arc consistency could be achieved in near constant time on the AC Chip is in no way affected. Although expected case analysis is problematic as usual, many experiments support this claim.

In short, although the possible existence of algorithms with time complexities slightly below linear cannot be discounted, it is likely that the AC Chip algorithm has the fastest possible worst case time complexity. Certainly, the AC Chip algorithm can be expected to operate much faster than linear on most problems.

2.6.2 Sub-linear-time Algorithms

If the limitation of polynomially-many processors is abandoned, significantly faster algorithms for arc consistency than the AC Chip can be constructed. These algorithms are interesting primarily from a theoretical point of view. They prove that linear-time algorithms are not the fastest possible, contrary to the claim of [Samal and Henderson, 1987], and offer insights into how arc consistency algorithms might be designed to be faster.

In Appendix D an algorithm for node and arc consistency is described that takes $O(\log na)$ time on a PRAM with $O(n^2a^22^{na})$ processors. It does so by finding the maximum cardinality consistent set of variable-label pairs, which is shown to be equivalent to the result of applying node and arc consistency.
3 Domain Specific Parallel Constraint Satisfaction

3.1 Example Problem: Matching Descriptions of Tinker Toys

We have seen how a network may be made arc consistent very quickly if parallel hardware resources are used. Furthermore, we have seen some examples of how to use the schemes to solve real constraint satisfaction problems. But what if we wanted to solve a non-trivial problem of significant size using parallel constraint satisfaction? Are the schemes we have proposed feasible? Consider the general AC chip solution, for example. This algorithm offers an extremely fast expected run time that is essentially independent of the size of the problem, which makes it a very desirable solution for CSPs. But the AC chip requires $O(a^2n^2)$ space or hardware, and it is easy to see that this becomes an infeasibly large number for relatively small $n$ and $a$. For example, if $n$ and $a$ are both 100, the AC chip requires over 100 million gates (somewhat beyond the capabilities of VLSI, even as it speedily evolving).

Often, however, a completely general purpose solution is not required, especially for large real problems. Instead, a solution to a particular class of problems is desired, and the characteristics of the problem domain can sometimes be used advantageously in providing a special purpose solution. In this section, we provide a concrete example of this process. We outline a non-trivial problem, and show how it can be formulated as a constraint satisfaction problem for which arc consistent solutions are sufficiently good answers. Then, we derive characteristics of the problem domain which allow us to build a special purpose AC chip whose space complexity is much lower. This special purpose solution is still adequate to solve all problem instances which could occur in the domain. Later, we show how to generalize this process for any domain.

3.1.1 Mapping the Problem to Constraint Satisfaction

Consider the problem of matching descriptions of Tinker Toy objects, such as those seen in Figure 5. This problem arises in the recognition of Tinker Toy objects - one description is derived from a picture, and then matched against a stored library of such descriptions. Tinker Toy matching is an instance of the structure recognition problem. Constraint satisfaction or relaxation methods have been shown useful in the solution of such structure recognition problems [Waltz, 1975; Kitchen and Rosenfeld, 1979; Cooper, 1988].

![Figure 5: Example Tinker Toy Objects](image-url)
The most straightforward approach to Tinker Toy matching is to treat the structures as labeled graphs. That is, Tinker Toy disks are regarded as the labeled vertices of graphs, and the rods are the edges in the graph description. In this case, the graph matching solution outlined in Section 2.3.3 provides a scheme whereby arc consistency can be used to match the descriptions [Cooper, 1988]. But this solution can only distinguish between topologically different graphs. Many Tinker Toy objects are topologically similar but differ in their geometric characteristics, such as the two objects in Figure 5.

An alternative formulation of the problem is to map to the nodes of the CSP both the rods and the slots (or holes) on the Tinker Toy disks (see Figure 6). In this formulation, the labels for the nodes in the CSP are the list of possibly corresponding rods and slots from the other Tinker Toy description. The unary and binary constraints are essentially the same as for the graph matching example. That is, the unary constraint is label agreement, and the binary (or arc) constraint is that the connectivity of the structures being matched must be the same. The labeling array that results from this framing of the Tinker Toy matching problem as a CSP is diagramed in Figure 7. This formulation has the advantage of encoding the geometric information pertaining to where the pieces attach to each other. But when the problem is framed this way, real instances of Tinker Toy matching can easily require 100 nodes and labels ($n$ and $a$ are 100). Thus, the general purpose AC Chip is somewhat infeasible for this problem.

### 3.1.2 Domain Specific Meta-Constraints

But consider Figure 7 again. If we are trying to match two Tinker Toys, it is clear that in a correct match, rods must match only rods. Similarly for slots; rods and slots cannot possibly match. But the nodes to be labeled in the CSP can represent both nodes and slots. Obviously, a node in the CSP which represents a rod cannot be labeled by a slot, and vice versa. For a particular problem instance, the constraint predicate $P_i(x)$ is what specifies which labels are valid for which nodes. But we know in advance that for all Tinker Toy matching problems framed in this way, those elements of the Labeling Array (or $P_i(x)$) that correspond to slot/rod or rod/slot matchings must always be zero. Whole sections of the labeling array become 0, as shown in Figure 8. Also, for each element of the labeling array
<table>
<thead>
<tr>
<th>Model</th>
<th>Rods</th>
<th>Disk 0 slots</th>
<th>Disk 1 slots</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rods 1</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Rods 2</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Rods 3</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Rods 4</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Rods 5</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Rods 6</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Rods 7</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Rods 8</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

Figure 7: Labeling Array for Tinker Toy Matching with Geometry
Figure 8: Reduced Domain Specific Labeling Array
that is 0, the entire circuit in Figure 2 does not have to be built for that element.

This is one example of the application of a domain specific \textit{meta-constraint}. In this example, we had meta-knowledge about the unary constraint predicate $P_i(x)$ - we could make statements about which node/label pairs were possible for any problem in the domain. We can make the same kind of observations about the binary (arc) constraints $Q_{ij}(x, y)$ in the Tinker Toy match problem as well. Recall that the simple binary constraint for structure matching is that the connectivity of the pairs agree. Formally, we stated this in section 2.3.3 as:

$$Q_{ij}(x, y) = (\text{adjacent}(i, j) = \text{adjacent}(x, y))$$

But $\text{adjacent}(i, j)$ is never true if both $i$ and $j$ are rods. In this way we can apply domain specific meta-constraints to the binary predicate $Q_{ij}(x, y)$ as well. And again the result is that portions of the AC Chip circuit do not have to be built.

\subsection{3.1.3 Application Example: Connectionist Simulation of AC Chip, Specialized for Tinker Toy Recognition}

The potential for reducing the space requirements of the AC Chip design was discovered during implementation of a connectionist simulation of the AC Chip for recognizing Tinker Toys. In this section, we describe that implementation. This primarily illustrates two things. First, it demonstrates explicitly the resource savings that can be made. Secondly, it provides a detailed example of the use and operation of the AC Chip on a real problem.

The example problem used to illustrate the implementation is given in Figure 9. The two objects are identical. In the Figure 9, the second Tinker Toy object has been rotated 180 degrees. The characteristics of the problem that are interesting include the fact that each disk on an object has a different pattern of rod attachments, there are two lengths of rods being used, and that one of the disks (Disk 0) has a rotationally symmetric rod attachment pattern. Because only local characteristics are considered in the $P$ and $Q$ predicates, the rotational symmetry on the one disk means a unique correspondence of all parts on that disk can only be established if a constraint can be propagated from the other disk's matches.

The state of the labeling arrays before the relaxation phase begins is given in Figure 10. (This and the following figures are screen dumps from a Sun running this application on the Rochester Connectionist Simulator [Goddard et al., 1988]). To achieve this initial state, a variety of computation has already taken place. Domain simplifications to the network (as
Figure 10: Labeling Array Initial State

described in the previous section) have already been built in. As well, a variety of unary or local criteria that could be computed easily without constraint propagation have been applied. In particular, rods only match equal length rods, and slots attached to rods only match slots which are also attached to a rod (likewise for empty slots). But further, the slots on one disk can only match the slots on another disk with the same rod attachment pattern.

The ambiguities remaining in the match problem after the application of these restrictions are graphically illustrated in Figure 10. All the rods but one, for example, are the same length. Therefore, initially at least, one rod has been assigned a unique correspondence and all the other rods have ambiguous correspondences. The geometry of the rod attachment characteristics on Disk 1 is unique - this allows a unique match of Disk 1 to be computed during initialization. The rod attachment patterns between Disk 0 and Disk 1 differ, so no slot matches are possible there. And the rotational symmetry at Disk 0 remains to be resolved.

The progress of the relaxation phase can be followed in Figures 11 through 14. For example, in Figure 11 some of the rod/rod matches have been determined to be inconsistent. The final state of the network, shown in Figure 14, represents a unique match between the Tinker Toy representations.

A naive and completely general purpose mapping of this problem to the AC Chip implemented on the Simulator would have taken about half a million units to simulate. Including domain simplifications, 26,632 units were used. For a larger problem with 3 disks and 5 rods, a general solution would require 1.5 million units, and the simplified simulation required only 41,830 units. Thus, orders of magnitude savings in resource use can be gained by exploiting domain specific information.
Figure 11: Relaxation:1

Figure 12: Relaxation:2
Figure 13: Relaxation:3

Figure 14: Relaxation: Final State
3.2 Domain Specific Performance Improvements for Arbitrary Domains

The gate and wire-saving optimizations that we discussed in the previous section for the Tinker Toy domain can be generalized to arbitrary domains. In this section, we give an algorithm that takes as input the description of a class of constraint satisfaction problems including domain information, and gives as output a specification for an optimized constraint satisfaction circuit for that domain. As in the Tinker Toy example above, characteristics of the domain are exploited to eliminate wires and gates in the circuit. The algorithm Circ_MIN performs a set of straightforward optimizations that may reduce the numbers of gates needed by orders of magnitude. For an arbitrary domain, these optimizations could reduce a circuit that was far too large for feasible hardware implementations to one that could be implemented economically. As well, because of the close relationship between the AC Chip implementation and uniprocessor implementations such as AC-4, the circuit optimizations can be used to reduce the space and time complexity of algorithms such as AC-4.

3.2.1 Predicate Specification

Specifying a CSP to be solved by arc consistency requires defining a set of nodes, a set of labels for the nodes, and the unary and binary predicates P and Q. P and Q are logical predicates consisting of terms whose truth values depend on the problem instance in the world. We have seen an example of this in section 2.3.3.

To develop domain specific constraint satisfaction solutions there is an additional requirement. That is, domain specific meta-constraints on P and Q must be given. Such meta-constraints amount to knowledge about the truth value of the constituent terms of P and Q over the problems in the domain. In particular, the truth value of some terms might be known before runtime, and might be constant for all problems in the domain. This information can be exploited to build an optimized circuit (or algorithm) for the CSPs.

For illustrative purposes, consider again the Tinker Toy matching problem. We saw in the preceding section that there is much domain specific knowledge available concerning this problem. The first step is to incorporate all this information into the expression of P and Q. Consider these definitions for P and Q:

\[ P(i, x) = (\text{rod}(i) \land \text{rod}(x)) \land (\text{length}(i) = \text{length}(x)) \lor \]
\[ (\text{slot}(i) \land \text{slot}(x)) \land (\text{filled}(i) = \text{filled}(x)) \]  

\[ Q(i, x, j, y) = \begin{cases} 
\text{true} & \text{if } i = j \text{ and } x = y \\
\text{false} & \text{if } i = j \text{ or } x = y \text{ but not both} \\
\end{cases} \]

\[ sd(i, j) \land sd(x, y) \land [p(i) - p(j)]_6 = [p(x) - p(y)]_6 \lor \]
\[ sr(i, j) \land sr(x, y) \land \text{connected}(i, j) = \text{connected}(x, y) \lor \]
\[ rr(i, j) \land rr(x, y) \text{ otherwise.} \]  

26
where some of the predicates can be expanded as follows:

\[
p(i) = \text{position}(i)
\]

\[
rr(i, j) = rod(i) \land rod(j),
\]

\[
sd(i, j) = \text{same\_disk}(i, j),
\]

\[
sr(i, j) = slot(i) \land rod(j) \lor rod(i) \land slot(j).
\]

and connected(i, j) is true iff rod i is plugged into slot j or rod j is plugged into slot i. The predicate same\_disk(i, j) is true iff i and j are slots on the same disk, and position(i) is the position of slot i on the disk, a number between 0 and s – 1.

These expressions for P and Q are part of a problem instance. A description of properties invariant across problem instances in the domain is also required. In constraint satisfaction problems, the nodes and labels are the entities that can have properties. In a Tinker Toy problem instance, the node i might represent a rod part, and thus have the property rod(i). If the node i is used to represent a rod part for all problem instances, rod(i) is a property invariant for all problem instances. Note that the mapping between Tinker Toy parts and the nodes of the CSP must maintain the invariant property across problem instances. It is straightforward to arrange that a certain subset of nodes represent rods and a different subset of nodes represent slots for all problem instances.

Once an invariant is established in this way, it can be conveniently described with a three-valued meta-term. A meta-term \( C'(i) \) is defined in terms of an object term describing \( C(i) \) as follows. By defining a domain as a (possibly infinite) set \( \delta \) and designating problem instances as \( \alpha \),

\[
C'(i) = \begin{cases} 
  t & \text{if } C(i) = t \text{ for all } \alpha \in \delta \\
  f & \text{if } C(i) = f \text{ for all } \alpha \in \delta \\
  u & \text{otherwise} \\
  (\text{if } C(i) = t \text{ for some } \alpha \in \delta \text{ and } C(i) = f \text{ for some } \alpha \in \delta) 
\end{cases}
\]

Invariants in a domain are, of course, represented by those meta-terms that evaluate to \( t \) or \( f \). For example, rod'(i) would be \( t \) if rod(i) was true for all problem instances in the domain. Many properties will not be invariant, and their meta-description will evaluate to \( u \).

Once the meta-terms describing the invariants are defined, it is a simple matter to define meta-predicates \( P' \) and \( Q' \). These are exactly the same as the unary and binary constraint predicates \( P \) and \( Q \), with meta-terms used in place of the object terms of \( P \) and \( Q \). The meta-predicates are evaluated with Kleene's three-valued logic [Turner, 1984], as in Figure 15.

Consider for example:

\[
P'(i, x) = (rod'(i) \land rod'(x)) \land (length(i) = length(x))' \lor (slot'(i) \land slot'(x)) \land (filled(i) = filled(x))'
\]

(3)
Domain invariants can be exploited when the meta-predicates $P'$ or $Q'$ evaluate as invariant on some arguments. As a concrete example, consider the meta unary predicate $P'(i, x)$ for some $i$ where $\text{rod}(i)$ is always true ($\text{rod}'(i) = \text{t}$), and for some $\bar{x}$ where $\text{slot}(\bar{x})$ is always true. Because a rod does not correctly correspond to a slot, the predicate $P'(i, \bar{x})$ will evaluate to $\text{f}$ for all problem instances. This fact is known in advance of runtime, and can be exploited in developing the algorithms to solve the CSP in this domain. We demonstrate this optimization in the next section.

### 3.2.2 Circuit Optimization

If the meta-predicates $P'$ or $Q'$ are $\text{t}$ or $\text{f}$ for some arguments, this translates to flip-flops which are always on or off in the AC Chip arc consistency algorithm. Systematically modifying the circuit to exploit the ramifications of these constants constitutes the domain dependent optimization of the algorithm. For example, if for some $i$ and $x$ $P'(i, x)$ is $\text{f}$, the entire circuit of Figure 2 is not required for the $u(i, x)$ element. The algorithm that performs these simplifications in general is given in Figure 16, and is called Circ.Min.

Circ.Min consists of five phases: $P$ elimination, $Q$ elimination, Replace_By_Hierarchical_Gates, Merge_With_Same_Inputs, and Flatten_Hierarchies.

$P$.elimination deletes any flip-flops $u(i, x)$ that represent impossible candidate labels along with the trees of gates that feed into them. $Q$.elimination removes type $A$ and gates that are not necessary and replaces each one by a wire when $Q' = \text{true}$. These two optimizations are often the most important.

The purpose of the remaining three procedures is to factor out common sub-terms in the circuit and eliminate the redundant circuitry that computes them. The procedure Replace_By_Hierarchical_Gates (Figure 17) replaces a single gate by a hierarchy of gates in the pattern set by the template $\text{or.hier}$ (for or gates) or $\text{and.hier}$ (for and gates). These hierarchies are supplied by the user, and represent additional description of the domain. The hierarchy must correspond to natural groupings of the labels for hierarchical gate replacement to enable more wires to be eliminated in the merge phase. The procedure Merge_With_Same_Inputs (Figure 18) replaces redundant gates by one gate (thus also eliminating redundant input wires), and wires the output appropriately to the multiple places it is used. Finally, the procedure Flatten_Hierarchy (Figure 19) replaces hierarchies of gates of the same type with functionally equivalent multiple-input gates. Flatten_Hierarchy thus

---

Figure 15: Truth Tables for Kleene's 3-valued logic

<table>
<thead>
<tr>
<th></th>
<th>$A$</th>
<th>$\sim A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>u</td>
<td>u</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>A $\land$ B</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>t</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>u</td>
<td>f</td>
<td>u</td>
<td>u</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>A $\lor$ B</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>f</td>
<td>t</td>
<td>t</td>
<td>f</td>
</tr>
<tr>
<td>u</td>
<td>u</td>
<td>f</td>
<td>u</td>
</tr>
</tbody>
</table>
procedure Circ_Min
inputs: domain meta-predicates $P'$ and $Q'$
node tree node.hier
label tree label.hier
modifies: circuit, as in Figure 2
begin
(* P elimination *)
forall $(i, x)$ in $P'$ do
  if $P'(i, x) = f$ then
    Recursive_Delete(flip-flop $u(i, x)$)

(* Q elimination *)
forall $(i, j, x, y)$ in $Q'$ do
  if $P'(j, y) = f$ or $Q'(i, j, x, y) = f$ then
    Recursive_Delete(and gate $A(i, j, x, y)$)
  else if $Q'(i, j, x, y) = t$ then
    begin
      wire output of flip-flop $u(j, y)$ to or gate $B(j)$
      Recursive_Delete(and gate $A(i, j, x, y)$)
    end
end
Replace_By_Hierarchical_Gates(circ, label.hier, node.hier);
Merge_With_Same_Inputs(circ);
Flatten_Hierarchies(circ);
end

Figure 16: Procedure Circ_Min

inverts the function of Replace_by_Hierarchical_Gates where it is still possible after common subterms have been eliminated. It is reasonable to apply this function only to minimize the number of gates and wires. If other complexity measures (wire length or wire crossings) are important, the functionality of Flatten_Hierarchy might be undesirable.

The procedure Recursive_Delete deletes the gate that is its argument, and recursively deletes any gates which become redundant by its deletion because they have no inputs or no outputs.

![Figure 17: Procedure Replace_By_Hierarchical_Gates](image-url)
3.2.3 Analysis and Performance

The correctness of the algorithm can be determined by noting that each transformation transforms one or more gates into an equivalent set of gates. Therefore, the function of the AC circuit is preserved throughout the algorithm.

A straightforward analysis shows that the algorithm takes $O(a^2n^2)$ time and space. Since the circuit reduction algorithm is run offline, the exact value of its complexity is not important. However, it should be noted that the algorithm does not need combinatorial resources; it fact it needs time and space linear in the original size of the AC circuit.

Circ.Min may reduce the complexity of an AC circuit by orders of magnitude. For instance, for Tinker Toy domain problems with a number of disks $d$, number of rods $r$, and number of slots per disk $s$, the number of gates is reduced from almost $2(r + ds)^4$ to about $2r^2d^2s^2$. For the Tinker Toy domain with a maximum of 5 rods and 3 disks, each with 8 slots, this is a twenty-fold decrease (see Table 1 and Figure 21). The order of magnitude of decrease is given for an arbitrary Tinker Toy domain in Table 2.

As an example, Figure 20 gives part of the reduced circuit for a Tinker Toy matching problem with at most 3 rods and 3 disks, with 2 slots per disk. The Figure gives the input circuit to a single target candidate labeling - the match of slot 1 on disk 1 of the object, to slot 2 of disk 3 of the model. This candidate labeling is shaded in Figure 20.

In the figure, the square and rectangular boxes represent the array of flip-flops $u(i, x)$ (holding $P(i, x)$) and the array of flip-flops $v(i, j, x, y)$ (holding $Q(i, j, x, y)$) respectively. All four types of optimizations have been made to this circuit. P-elimination has eliminated $u(i, x)$ flip-flops (and their associated circuitry) from the upper left-hand corner and lower right-hand corners of the $u(i, x)$ array. Q-elimination has eliminated the $v(i, j, x, y)$ flip-flops associated with all of the slot-slot matches. Slot-slot matches can be divided into 2 groups - those compatible with the target candidate labeling ($Q' = t$), and those incompatible with...
the target candidate labeling ($Q' = f$). Those that are compatible have an output wire and can affect the consistency of the candidate labeling. The or gates receiving input from slot-slot matches have been broken up into a two-level hierarchy, with slots from the same disk grouped together at the lower level.

We do not claim to be able to generate the optimal (most reduced) circuit for any problem. Domain knowledge not expressed in the meta-predicates might be exploitable [Swain and Cooper, 1988], or subtle optimizations not discovered by Circ.Min might be feasible. Since, in general, circuit optimization is a difficult problem and must be approached heuristically [Brayton et al., 1984], we do not expect there to be a tractable algorithm that finds the optimal circuit.
Table 1: Optimizations for Tinker Toy domain with 5 rods and 3 disks with 8 slots, quantities in thousands

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Gates saved</th>
<th>Gates remaining</th>
<th>Wires saved</th>
<th>Wires remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>(original AC circuit)</td>
<td></td>
<td>1390</td>
<td></td>
<td>2085</td>
</tr>
<tr>
<td>P flip flop elimination</td>
<td>678</td>
<td>712</td>
<td>1026</td>
<td>1059</td>
</tr>
<tr>
<td>Q flip flop elimination</td>
<td>637</td>
<td>75</td>
<td>803</td>
<td>256</td>
</tr>
<tr>
<td>B or gate merging</td>
<td>9</td>
<td>66</td>
<td>147</td>
<td>109</td>
</tr>
<tr>
<td>hierarchical gates</td>
<td></td>
<td>66</td>
<td>9</td>
<td>100</td>
</tr>
</tbody>
</table>

Figure 21: Gates (left) and wires (right) in AC circuit after each optimization, for the Tinker Toy domain with 5 rods and 3 disks with 8 slots. O = original, P = P flip flop elim., Q = Q flip flop elim., B = B or gate merging, H = hierarchical gates. Quantities in millions.
3.2.4 Mapping the Optimizations to AC-4

As discussed in Section 2.5, there is a close relationship between the arc consistency algorithms at various levels of parallelism. The AC Chip to AC-4 mapping can be used to take the optimizations described above for the AC Chip and apply them to the AC-4 algorithm. The resultant algorithm is given in Appendix C.

Since AC-4 only iterates over the candidate labels for which $P(i, x)$ is true and not all $(i, x)$ it already incorporates an optimization similar to $P_{flip-flop elimination}$. Because the optimizations of AC-4 overlap the optimizations achieved by circuit minimization, we do not see as great reductions in the example Tinker Toy match done above. In a typical Tinker Toy experiment, AC-4 took 195,000 steps. The optimized algorithm AC4-Min took 45,000 steps, about one-fifth as many.

The processor and time reductions the optimizations achieve for the AC Chip, ACP and AC-4 (in one experiment) are shown in Figure 23. Processor reductions for the AC Chip are translated into time reductions for AC-4.

---

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Gates saved</th>
<th>Gates remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original AC circuit</td>
<td>$d^3s^3r + d^2s^2r^2 + dsr^3$</td>
<td>$(ds + r)^4$</td>
</tr>
<tr>
<td>P flip flop elimination</td>
<td>$d^4s^4 + r^4$</td>
<td>$d^4s^4 + r^4$</td>
</tr>
<tr>
<td>Q flip flop elimination</td>
<td>$d^3s^3 + r^3$</td>
<td>$d^3s^3 + d^2s^2r^2 + r^3$</td>
</tr>
<tr>
<td>B or gate merging</td>
<td></td>
<td>$d^2s^2r^2$</td>
</tr>
<tr>
<td>hierarchical gates</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Wires saved</th>
<th>Wires remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original AC circuit</td>
<td>$d^3s^3r + d^2s^2r^2 + dsr^3$</td>
<td>$(ds + r)^4$</td>
</tr>
<tr>
<td>P flip flop elimination</td>
<td>$d^4s^4 + r^4$</td>
<td>$d^4s^4 + r^4$</td>
</tr>
<tr>
<td>Q flip flop elimination</td>
<td>$d^3s^3 + r^3$</td>
<td>$d^3s^3 + d^2s^2r^2 + r^3$</td>
</tr>
<tr>
<td>B or gate merging</td>
<td></td>
<td>$d^2s^2r^2 + d^3 + r^3$</td>
</tr>
<tr>
<td>hierarchical gates</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Efficiency of optimizations for Tinker Toy domain
4 Conclusions

The utility of constraint satisfaction methods in the solution of many AI problems suggests that efficient implementations might be widely useful. For arc consistency, a uniprocessor algorithm optimal in time complexity is now known, so further improvements must come through parallelization or specialization for specific problems. In this paper, we have explored both possibilities.

We have shown the AC Chip, a massively parallel algorithm for the arc consistency problem. As might be expected, the highly parallel implementation runs very fast. Although worst case running time is linear in the number of variables and labels, it is more reasonable to expect that the network runs in a small constant number of time steps. A slightly less parallel version of the algorithm has also been specified and tested on the Connection Machine.

We have also shown how to exploit domain specific meta-constraints to optimize arc consistency algorithms for a particular domain. For highly parallel spatially intensive algorithms such as the AC Chip, this specialization amounts to reducing the space requirements (or number of processing elements) for the algorithm. But the close relationship between all the algorithms allows us to port the optimizations from one form of the algorithms to another, such as the AC-4 uniprocessor algorithm.

Consider Figures 22 and 23. These figures show a spectrum of algorithms for the arc consistency problem, from sequential to parallel. In this paper we have added to the spectrum of algorithms. The solid line (Line A) in Figure 22 represents the space-time trade-offs for completely general purpose algorithms. The AC Chip algorithm we have specified in this paper provides a definition for the parallel end of the spectrum of general purpose arc consistency algorithms. The parallelism in this algorithm reflects the unit/value design principle. A less extreme choice of parallelism leads to the SIMD algorithm implemented on the Connection Machine. This algorithm is found somewhere between the sequential and parallel extremes.

Equally important, we have shown how to define a second sequential/parallel trade-off line with better performance. The height of Line A in the figure is intended to represent the optimal achievable algorithm (e.g. AC-4, which takes $O(n^2a^2)$ time and is known to be optimal) for a given space-time trade-off. But by exploiting domain specific problem characteristics, better performance of arc consistency algorithms can be achieved right across the spectrum of sequential-to-parallel algorithms, as diagramed by Line B.

While Figure 22 represents the spectrum of algorithms with their worst-case complexity, Figure 23 represents the performance of the algorithms on a real experiment, and thus could be considered more representative.

With an understanding of the complete spectrum of possible algorithms for the arc consistency problem comes the freedom to choose appropriately. If optimum time performance is required for completely general problems, the AC Chip can be constructed. If an IBM PC is to be used to perform arc consistency in some repetitive vision task in a well understood domain, the time-optimized version of AC-4 can be selected.
Line A: Spectrum of General Purpose Algorithms
Line B: Domain Specific Algorithms
Figure 22: Sequential/Parallel Spectrum of Arc Consistency Algorithms
Figure 23: Optimizations to Arc Consistency Algorithms on One Experiment (Tinker Toy circuit with rods=5, disks=3, and slots/disk=8)
Appendix: The Language Algol*

Algol* is a language for expressing parallel algorithms based on the language C*, a language developed by Thinking Machines Corporation for programming the Connection Machine [Rose and Steele, 1987]. Algol* is simply Pidgin Algol with the addition of the poly data structure, parallel sequencing statements if and while, and reduction operators.

The difference between a poly data structure and a similar data structure in a traditional programming language is that a processor is associated with each element of a data structure declared poly. Non-poly or traditional data structures are said to have the attribute mono: they reside on a distinguished processor called the host. Each element of a poly data structure resides in the memory of the processor associated with it. Parallel operations on a poly data structure take place within a with statement:

```plaintext
with poly data structure do
  statement
```

The semantics of parallel if and while statements in Algol* are identical to those in C*.

```plaintext
if expression then
  statement
```

If `expression` is a poly value then the `statement` is executed in all the processors in which `expression` is true. If `expression` is false in all processors then `statement` is not executed (including mono code).

The while statement can be defined in terms of the if statement as follows:

```plaintext
while expression do
  statement
```

is identical to:

```plaintext
Top:
  if not expression then goto Done;
  statement
  goto Top;
Done: ;
```

Reduction operations reduce a poly variable to a mono value. For example, one of selects an arbitrary member of a poly variable and returns it as a mono value.

In Algol*, variables are mono unless explicitly declared to be poly.

Appendix: A Space-Efficient AC-4

The AC-4 algorithm described in [Mohr and Henderson, 1986] can be modified so that it requires only $O(ea)$ space instead of $O(ea^2)$ space without increasing the order of complexity. This reduction can be made by eliminating the support lists, and instead re-evaluating the predicate. The modified algorithm is given in Figure 24.
Step 1. Construction of the data structures

\[ M := 0; \]
\[ \text{for } (i, j) \text{ in } E \text{ do} \]
\[ \quad \text{for } x \text{ in } A_i \text{ do} \]
\[ \quad \quad \begin{align*}
& \text{Total} := 0; \\
& \quad \text{for } y \text{ in } A_j \text{ do} \\
& \quad \quad \quad \text{if } Q(i, x, j, y) \text{ then} \\
& \quad \quad \quad \quad \text{Total} := \text{Total} + 1; \\
& \quad \quad \quad \text{if } \text{Total} = 0 \text{ then} \\
& \quad \quad \quad \quad \quad \begin{align*}
& \quad \quad \quad \quad M[i, x] := 1; \\
& \quad \quad \quad \quad A_i := A_i - x
\end{align*}
\quad \text{else} \\
& \quad \quad \quad \quad \text{Counter}[(i, j), x] := \text{Total;}
\end{align*} \]
\[ \text{end} \]
\[ \text{Initialize List with } \{(i, x) | M(i, x) = 1\}; \]

Step 2. Pruning the inconsistent labels

\[ \text{while List not Empty do} \]
\[ \quad \text{begin} \]
\[ \quad \quad \text{choose } (j, c) \text{ from List and remove } (j, c) \text{ from List;} \]
\[ \quad \quad \text{for } i \text{ in } E_j \text{ do} \]
\[ \quad \quad \quad \begin{align*}
& \quad \text{for } x \text{ in } A_i \text{ do} \\
& \quad \quad \quad \quad \text{if } Q(i, x, j, y) \text{ then} \\
& \quad \quad \quad \quad \quad \begin{align*}
& \quad \quad \quad \quad \text{Counter}[(i, j), x] := \text{Counter}[(i, j), x] - 1; \\
& \quad \quad \quad \quad \quad \text{if } \text{Counter}[(i, j), x] = 0 \text{ and } M[i, x] = 0 \text{ then} \\
& \quad \quad \quad \quad \quad \quad \begin{align*}
& \quad \quad \quad \quad \quad \text{Append(List,(i,x));} \\
& \quad \quad \quad \quad \quad M[i, x] := 1; \\
& \quad \quad \quad \quad \quad A_i := A_i - x;
\end{align*}
\end{align*}
\quad \text{end}
\end{align*} \]
\[ \quad \text{end} \]
\[ \text{end} \]

Figure 24: Space-efficient AC-4
C Appendix: Optimized AC-4 Algorithm

AC-4 can be modified so that it directly simulates the circuit generated by the circuit minimization algorithm Circ.Min. The modified algorithm is given in Figure 2.5. It differs slightly from AC-4 in that the or gates are put on the List data structure when their output becomes false (g.counter = 0) instead of the candidate labels, which are associated with the flip-flops. If a the output of a gate g connects to the input of only one other gate, that gate is referred to as g.out, whereas if there is more than one, the list of them is referred to as g.outputs. For simplicity AC4_Min simulates a circuit without hierarchical gates.
procedure AC4_Min(circuit circ, set P, set Q):
begin
  for (i,x) in P do
    for g in circ.P.flip_flops(i.x).out do
      if g.type = A_and_gate then
        begin
          if g.and_input.index ∈ Q then
            g.out.counter := g.out.counter + 1;
        end
        else g.counter := g.counter + 1;
    end
  for g in circ.B_or_gates do
    if g.counter = 0 then List.append(g);
    while not List.empty() do
      begin
        g := List.get(); (* remove and delete first element *)
        for out in g.outputs do out.kill()
      end
  end
procedure C_and_gate::kill()
begin
  if alive then
    begin
      for g in outputs do g.kill();
      alive := false
    end
end
procedure P_flip_flops::kill()
begin
  if alive then
    begin
      for g in outputs do
        begin
          if g.type = A_and_gate then
            begin
              if Q(g.and_input.index) then g.out.decrement() end
            else g.decrement();
          alive := false
        end
    end
end
procedure B_or_gate::decrement()
begin
  if counter > 0 then
    begin
      counter := counter - 1;
      if counter = 0 then List.append(this)
    end
end

Figure 25: The Algorithm AC4_Min

40
D Appendix: Fast Arc Consistency on a PRAM

Here we describe an algorithm for arc consistency on a Parallel Random Access Machine (PRAM) that has a very low time complexity, \( O(\log na) \), achieved at the cost of a huge number of processors, \( O(n^2a^{2n^2}) \). The algorithm relies on the equivalence between the answer of constraint satisfaction and the maximum sized consistent set of variable-label pairs. It tries all possible solutions, determines which of them are consistent, and finds the consistent set of maximum size. Verifying whether an answer is consistent is done in constant time, and discovering which of the consistent answers is the one of maximum size is done in \( \log na \) time.

**Theorem 1 (Kitchen and Rosenfeld, 1979)** Let \( S \) be the result of applying node and arc consistency. Let \( T \) be another consistent set of variable-label pairs. Then \( T \subseteq S \).

**Proof.** Let \( D = T - S \). Suppose \( D \neq \emptyset \). Let \( (i, x) \) be the first element from \( D \) that was deleted by the node and arc consistency algorithm. Then \( (i, x) \) is inconsistent in \( S \cup D \), and so \( (i, x) \) is inconsistent in \( U \), since \( T \subseteq S \cup D \) and deleting labels can only make a label inconsistent, not consistent. But by assumption \( U \) is consistent, and we have a contradiction. Therefore \( D = \emptyset \) and so \( T \subseteq S \).

**Corollary.** The result of applying node and arc consistency is the (unique) largest set of consistent labels.
Let $U$ be the universal set of all variable-label pairs.

$$U = \{(i, x) \mid i \in 1..n, x \in 1..a\}.$$ 

Enumerate all subsets $T$ of $U$, and let the integer associated with $T$ be $id(T)$.

The procedure $Fast_{AC}$ computes node and arc consistency, and returns the i.d. of the result (a subset of $U$). Its sub-procedures, $Consistent$ and $High_{Val}$, are described below.

In a PRAM, if two or more processors write to the same variable, it is undefined which of the writes determines the final value of the variable. A large number of processors may write into the array $consistent_{id}$, but because there is a unique largest set of consistent variable-label pairs $T'$ only one processor will write into the array element $consistent_{id}[\text{cardinality}(T')]$. Therefore, the answer to the constraint satisfaction problem will not be overwritten under any ordering of writes into the array.

```plaintext
procedure Fast_{AC} returns integer
    array consistent_{id}[0..na] of integer;
    || for $k := 0$ to $na$ do
        consistent_{id}[k] := VOID;
    || for $T \in U$ do
        if Consistent($T$) then
            consistent_{id}[\text{cardinality}(T)] := id($T$);
        return High_{Val}(consistent)
    end

The notation $||$ for represents a parallel for loop using as many processors as iterations in the corresponding sequential for loop [Samal and Henderson, 1987].
```
The procedure Consistent checks a set $U$ of variable-label pairs for node and arc consistency using $O(n^2a^2)$ processors on a PRAM in constant time. It does this by simulating one iteration of the network in Section 2.3. Note that $P_i(x)$ and $Q_{ij}(x, y)$ denote values of the unary and binary predicate, respectively.

```plaintext
procedure Consistent(pair_set T) returns boolean
  boolean consistent := true;
  array or_gate[1..n, 1..a, 1..n] of boolean := false;
  array and_gate[1..n, 1..a] of boolean := true;

  // for i := 1 to n do
  //   // for x := 1 to a do
  //     if not $P_i(x)$ and $(j, y) \in T$ then
  //       consistent := false;

  // for i := 1 to n do
  //   // for j := 1 to n do
  //     // for x := 1 to a do
  //       // for y := 1 to a do
  //         if $Q_{ij}(x, y)$ and $(j, y) \in T$ then
  //           or_gate[i, x, j] := true;

  // for i := 1 to n do
  //   // for x := 1 to a do
  //     // for j := 1 to a do
  //       if or_gate[i, x, j] = false then
  //         and_gate[i, x] := false;

  // for i := 1 to n do
  //   // for x := 1 to a do
  //     if $(i, x) \in T$ and not and_gate[i, x] then
  //       consistent := false;

  return consistent
end
```
The procedure High_Val finds the consistent set with the highest cardinality with \( \log \alpha \) recursive calls.

\begin{verbatim}
procedure High_Val(array A) returns integer
array B[roundup(size(A)/2)];

if size(A) = 1 then
    return A[1]
else
    begin
        for i := 1 to roundup(size(A)/2) do
            if 2i + 1 > size(A) then
            else if A[2i + 1] ≠ VOID then
            else
                B[i] := A[2i];
        return High_Val(B)
    end

end
\end{verbatim}

E Appendix: Extra Minimizations for the Tinker Toy Domain

Here we give an example of an optimization that requires deeper knowledge of the domain than provided by the meta-predicate described in Section 3.2. The optimization can be made for the Tinker Toy domain when it is described by a slightly weaker predicate than the previous domain description, Equations 1 and 2. The weaker predicate filters fewer incorrect matches, but at the same time reduce the number of edges in the constraint graph. The extra optimization takes advantage of the sparseness of the constraint graph to build fewer wires that can be switched to connect the appropriate flip-flops. Since the meta-predicate corresponding to the modified (weaker) predicate is identical to the original meta-predicate (see below), the optimizations discovered by Circ_Min are the same in both cases.

The modified predicate reads

\begin{align}
P_M(i, x) &= (\text{rod}(i) \land \text{rod}(x)) \land (\text{length}(i) = \text{length}(x)) \lor \\
&\quad (\text{slot}(i) \land \text{slot}(x)) \land (\text{filled}(i) = \text{filled}(x))
\end{align}
\[ Q_M(i, x, y) = \begin{cases} 
  t & \text{if } i = j \text{ and } x = y \\
  f & \text{if } i = j \text{ or } x = y \text{ but not both}
\end{cases} \]

\[ Q_M(i, x, y) = \begin{cases} 
  sd(i, j) \land sd(x, y) \land [p(i) - p(j)] = [p(x) - p(y)] \lor \\
  sr(i, j) \land sr(x, y) \land \text{connected}(i, j) \land \text{connected}(x, y) \lor \\
  sr(i, j) \land sr(x, y) \land \neg\text{connected}(i, j) \lor \\
  rr(i, j) \land rr(x, y) \text{ otherwise.}
\end{cases} \] (5)

Notice that the predicate is now asymmetric (compare Equation 2). There is already a natural asymmetry in the recognition problem. To be able to deal with occlusion it is natural to frame the recognition problem in such a way that variables refer to elements of the scene and that they are labeled by elements of the database. Then an occluded object in the scene can still have a legal labeling, whereas if the variables were the elements of the database, some would have no valid label. Because of this asymmetry, i's and j's refer to image features and x's and y's to database features.

Since \( P_M = P \), the meta-predicate \( P'_M = P' \). The binary predicates \( Q_M \) and \( Q \) differ, but

\[ Q_M = Q' = \begin{cases} 
  t & \text{if } i = j \text{ and } x = y \\
  f & \text{if } i = j \text{ or } x = y \text{ but not both}
\end{cases} \]

\[ Q_M = Q' = \begin{cases} 
  sd(i, j) \land sd(x, y) \land [p(i) - p(j)] = [p(x) - p(y)] \lor \\
  sr(i, j) \land sr(x, y) \land \text{connected}(i, j) \land \text{connected}(x, y) \lor \\
  sr(i, j) \land sr(x, y) \land \neg\text{connected}(i, j) \lor \\
  rr(i, j) \land rr(x, y) \text{ otherwise.}
\end{cases} \]

Therefore, the algorithm \( \text{Circ.Min} \) produces the same circuit for both domains.

In a Tinker Toy, only one slot is only connected to one rod, that is, for a given slot i, \text{connected}(i, j) is true for only one rod j, and so the slot i is adjacent to only one rod j in the constraint graph. Since slot x is connected to only one rod y the slot-slot match \( (i, x) \) is supported by at most one rod-rod match \( (j, y) \). Therefore, for any given problem only one wire is need run from the rod-rod array to each element of the slot-slot array. The connection that must be made depends on the problem, so the Tinker Toy matching chip must allow any rod-rod match to be connected to each slot-slot match.

An architecture that solves this problem using \( O(rds(r + ds)) \) wires is shown in Figure 26, for Tinker Toys with at most three rods and three two-slot disks \( (r=3, d=3, s=2) \). In it, there are \( rds \) junctions labeled \( (i, y) \), wired to every possible match of rod y and slot i \( (r + ds \text{ connections in total}) \). The predicates \text{connected}(i, j) \text{ and connected}(x, y) \text{ select (at most) one wire into and one out of the junction.}

The correctness of the circuit can be determined by noting that the value at point A in Figure 26 is

\[
\begin{align*}
1 & \quad \text{if } \neg\text{filled}(i) \\
0 & \quad \text{if } \text{filled}(i) \text{ and } \neg\text{filled}(x) \\
\nu(j, y) & \quad \text{for } (j, y) \text{ such that } \text{connected}(i, j) \text{ and } \text{connected}(x, y), \text{ otherwise}
\end{align*}
\]
Figure 26: Minimized Tinker Toy recognition circuit; rods=3, disks=3, slots/disk=2.
and that this is the same value as what would be calculated by the original circuit. There are \( rds \) of the type B OR gates (see Figure 26) each with \( r \) inputs and \( ds \) outputs, giving \( O(rds(r + ds)) \) wires in total.
References


