4. Conclusions

The results presented here have shown that the storing of two key values in the nodes of a static binary search tree can significantly speed searches of that tree by decoupling the potentially conflicting functions of frequency ordering and subtree structuring. Median split trees have been shown to be good estimates for the corresponding optimal split trees, to have $\log n$ bounded execution time, require less storage than FOBS trees, and to be stable around a performance which is optimal for any method based on key comparisons.

It is hoped that they will find application in any searching problem characterized by a static set of keys with a highly skewed distribution of frequencies of occurrence. One such application, dictionary lookup for English, has been discussed in detail and shown to be well suited to this technique. This success provides encouragement for the investigation of other applications of this type.

Received February 1977; revised January 1978

References

1. Blum, M., Floyd, R., Pratt, V., Rivest, R., and Tarjan, R. Time bounds for selection. *JCSS 7*, 4 (Aug. 1973), 448–461.
2. Kay, M., and Kaplan, R. Word recognition. Xerox Palo Alto Res. Ctr., Palo Alto, Calif., 1976.
3. Knuth, D. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1968.
4. Knuth, D. Optimum binary search trees. *Acta Informatica 1*, 1 (1971), 14–25.
5. Knuth, D. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.
6. Kucera, H., and Francis, W. *Computational Analysis of Present-day American English.* Brown U. Press, Providence, R.I., 1967.
7. Maly, K. Compressed tries. *Comm. ACM 19*, 7 (July 1976), 409–415.
8. Sprugnoli, R. Perfect hashing functions: A single probe retrieving method for static sets. *Comm. ACM 20*, 11 (Nov. 1977), 841–849.
9. Zipf, G. *Human Behavior and the Principle of Least Effort.* Addison-Wesley, Reading, Mass., 1949.

Corrigendum. Programming Languages
C.A.R. Hoare, "Communicating Sequential Processes," *Comm. ACM 21*, 8 (August 1978), 666-677.

The technical department for this paper was incorrectly listed as Programming Techniques, S.L. Graham and R.L. Rivest, Editors. The correct department is Programming Languages, J.J. Horning, Editor, with Ben Wegbreit having completed the editorial consideration.

Programming Techniques
S.L. Graham, R.L. Rivest Editors

Synthesizing Constraint Expressions

Eugene C. Freuder
University of New Hampshire

A constraint network representation is presented for a combinatorial search problem: finding values for a set of variables subject to a set of constraints. A theory of consistency levels in such networks is formulated, which is related to problems of backtrack tree search efficiency. An algorithm is developed that can achieve any level of consistency desired, in order to preprocess the problem for subsequent backtrack search, or to function as an alternative to backtrack search by explicitly determining all solutions.

Key Words and Phrases: backtrack, combinatorial algorithms, constraint networks, constraint satisfaction, graph coloring, network consistency, relaxation, scene labeling, search

CR Categories: 3.63, 3.64, 5.25, 5.30, 5.32

1. Satisfying Simultaneous Constraints: Problem and Applications

We are given a set of variables $X_1, \ldots, X_n$ and constraints on subsets of these variables limiting the values they can take on. These constraints taken together constitute a global constraint which specifies which sets of values $a_1, \ldots, a_n$ for $X_1, \ldots, X_n$ can simultaneously satisfy all the given constraints. In other words, the constraints define an n-ary relation. Our problem is to synthesize this relation, i.e. to determine those sets of values which simultaneously satisfy the set of constraints.

Author's address: Department of Mathematics and Computer Science, University of New Hampshire, Durham, NH 03824.

Research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

The simultaneous satisfaction of several constraints—call them properties, relationships, predicates, features, or attributes—is a very general problem, with more applications than I can fully survey here. The essential technique I apply, iterated reduction of possibilities through constraint propagation, has analogs in many areas of computer science and mathematics. Many of these applications and analogs are described in [10, 14, 24]. Applications range from database retrieval (find all x, y, and z such that x is a part and y is a part, and z is a supplier, x must be installed before y, and z supplies both x and y) [7, 10, 11] to scene analysis (segment the scene into regions such that sky regions are blue, grass regions are green, and cars are not totally surrounded by either grass or sky) [18]. Of particular note is the work of J.R. Ullman, who has used constraint propagation methods in a variety of contexts, ranging from pattern recognition [19] to graph isomorphism [21].

Often we are only given, or choose to use, "local" constraints, i.e. constraints on small subsets of the variables, from which we must synthesize the global constraint. (For fundamental results on the complementary problem, analysis of a global constraint into local ones, see [13].)

2. Previous Results: Partial Consistency

Constraints represented in network form may be propagated through (potentially) parallel algorithms which cut down the solution search space by ruling out inconsistent combinations of values.

The obvious brute force approach of testing every possible combination of values faces an equally obvious combinatorial explosion. Backtrack search techniques cut down the search space but often exhibit costly "thrashing" behavior [2, 16]. Mackworth [10] has interpreted previous work by Fikes [5], Waltz [23], and Montanari [14] as cutting down the search space and avoiding classes of thrashing behavior by eliminating combinations of values which could not appear together in any set satisfying the global constraint. These combinations are eliminated by algorithms which can be viewed as removing inconsistencies in a constraint network representation of the problem. Mackworth distinguishes three levels of inconsistency in constraint networks.

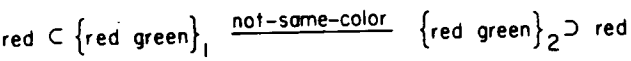
This section introduces these levels of inconsistency, and a network representation of the constraint satisfaction problem. The representation and the concept of inconsistency will be generalized in subsequent sections. The classical graph coloring problem will be used for illustration: the problem of coloring the vertices of a graph using a given set of colors, such that if two vertices are connected by an edge they do not have the same color. (The famous "four color problem" can be represented in these terms.)

Montanari and Mackworth represent variables and constraints in the form of a network, where each variable is represented by a node and each constraint by a link or arc. (Constraints are restricted to unary and binary constraints, i.e. predicates on one or two variables.)

For example, consider the problem of coloring a two-vertex complete graph using one color, say red. A complete graph is one in which all possible edges between pairs of vertices are present. The two-vertex complete graph contains two vertices, v_1 and v_2, and an edge between them. The coloring problem can be represented as a constraint satisfaction problem where variables X_1 and X_2 represent the colors of v_1 and v_2. There is a binary constraint, specifying that X_1 and X_2 are not the same color, and a unary constraint on both X_1 and X_2, requiring them to be red. Let us suppose we have two colors available in general, red and green; these form the initial domain of possible values for X_1 and X_2.

The problem can be represented as a constraint network as in Figure 1.

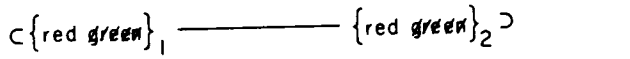
Fig. 1.



The nodes $\{\text{red green}\}_1$ and $\{\text{red green}\}_2$ contain the possible values for X_1 and X_2. The loop at each node corresponds to the unary constraints, and the link between the nodes corresponds to the binary constraint.

The first and most obvious level of inconsistency in constraint networks is node inconsistency. In this example, the potential domain of values for X_1 and X_2 is given as red and green, but the unary constraints specify red. We can immediately eliminate green from both nodes as in Figure 2.

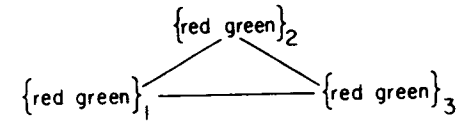
Fig. 2.



The next level of inconsistency is arc inconsistency. The arc from X_1 to X_2 is inconsistent because for a value in X_1, namely "red," there does not exist any value a_2 in X_2 such that red and a_2 together satisfy the relation "red is not the same color as a_2." To remedy this inconsistency, remove red from X_1 and similarly from X_2. This cuts down the search space all right: unfortunately, in this case it reflects the fact that the problem is impossible. There is no global solution, i.e. the network is what I call "unsatisfiable."

It is entirely possible for a network to have no arc inconsistencies, and still be unsatisfiable. Consider the problem of coloring a complete three-vertex graph with two colors, represented in Figure 3.

Fig. 3.



Assume the set of possible values for each variable is {red green} and the binary predicate between each pair again specifies "is not the same color as."

This network is arc consistent, e.g. given a value "red" for X_1, we can choose "green" for X_2: red is not the same color as green. Yet obviously there is no way of choosing single values a_1, a_2, a_3, for X_1, X_2, and X_3, such that all three binary constraints are satisfied simultaneously. If we choose red, for X_1, for example, we are forced to choose green for X_2 to satisfy the constraint between X_1 and X_2. This forces a choice of red for X_3, which forces a choice of green for X_1, already picked to be red.

Nevertheless, it may be helpful to remove arc inconsistencies from a network. This involves comparing nodes with their neighbors as we did above. Each node must be so compared; however, comparisons can cause changes (deletions) in the network and so the comparisons must be iterated until a stable network is reached. These iterations can propagate constraints some distance through the network. The comparisons at each node can theoretically be performed in parallel and this parallel pass iterated.

Thus removing arc inconsistencies involves several distinct ideas: local constraints are globally propagated through iteration of parallel local operations. It remains to be seen which aspects of this process are most significant to its application. The parallel possibilities may prove to be particularly important; however, at the moment serial implementations are used in practice.

Waltz's "filtering" algorithm for scene labeling [23] is the paradigm example of an arc consistency algorithm. Waltz wishes to attach labels to the lines in a line drawing indicating their semantic interpretations as convex, concave or occluding three-dimensional edges. The line drawing itself functions as the constraint network. Vertices function as network nodes. An individual vertex value consists of a label for each of the lines incident to the vertex; the set of possible values is initially constrained according to realizable three-dimensional interpretations for the various types of vertices. The lines are the arcs of the network and each represents the relation "the labelings of the adjacent vertices must agree along this line."

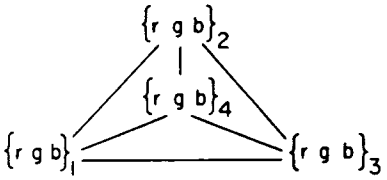
Waltz's filtering algorithm (especially when further constrained by specifying initial labels for edges on the background) generally results in an amazing combinatorial reduction: thousands of possibilities are often reduced to a state where all nodes have a single value remaining, thus totally solving the problem of obtaining the global solution. Of course the algorithm does not always terminate with a unique value at each node. Generally, in this case, most nodes will still have a unique value, while a few nodes will have a small set of values remaining. Normally this final state indicates that several ambiguous interpretations are possible; alternative sets of values that simultaneously satisfy all constraints can be quickly found with tree search.

It is perhaps not as well appreciated that this final state may also be reached for a figure which in fact admits no consistent labeling [6]. This is to be suspected, however, given that the filtering algorithm only achieves arc consistency. Motivated in part by Waltz's success, there has been a recent groundswell of constraint analysis applications in scene analysis, e.g. [1, 8, 12, 15, 18, 24].

Montanari [14] has developed a more powerful notion of inconsistency which Mackworth calls path inconsistency. A network is path inconsistent if there are two nodes X_1 and X_2 such that a satisfies X_1, b satisfies X_2, a and b together satisfy the binary constraint between them, yet there is some other path through the network from X_1 to X_2, such that there is no set of values, one for each node along the path, which includes a and b, and can simultaneously satisfy all constraints along the path. For example, the network in Figure 3 is path inconsistent: red satisfies X_1, green X_3, red is not the same color as green: however, there is no value for X_2 which will satisfy the constraints between X_1 and X_2, and between X_2 and X_3, while X_1 is red, X_3 is green.

Montanari gives an algorithm that essentially removes path inconsistencies from a network. However, path consistency does not necessarily insure satisfiability either, as powerful as it sounds. Consider the problem of coloring the complete four-vertex graph with three colors (Figure 4). Each node contains red, green, and blue, and each arc again represents the constraint "is not the same color as." In particular, path consistency does not fully determine the set of values satisfying the global constraint, which in this inconsistent case is the empty set.

Fig. 4.



In summary, arc and path consistency algorithms may reduce the search space, but do not in general fully synthesize the global constraint. When there are multiple solutions, additional search will be required to specify the several acceptable combinations of values. Even a unique solution may require further search to determine, and the consistency algorithms may even fail to reveal that no solutions at all exist.

3. An Extended Theory

As the coloring problem suggests, the general problem of synthesizing the global constraint is *NP*-complete [4], and thus unlikely to have an efficient (polynomial time) solution. On the other hand the experimental results of Waltz, and the theoretical studies of Montanari, suggest that in specific applications it may be possible to greatly facilitate the search for solutions. I will present

an algorithm for synthesizing the n-ary constraint defined by a set of constraints on subsets of n variables. It may be of substantial benefit in applications where pruning of arc and path inconsistencies still leaves many possibilities to be searched.

There are two key observations that motivated the algorithm.

1. Node, arc, and path consistency in a constraint network for n variables can be generalized to a concept of k-consistency for any $k \leq n$.

2. The given constraints can be represented by nodes, as opposed to links, in a constraint network; we can add nodes representing k-ary constraints to a constraint network for all $k \leq n$ (whether or not a corresponding k-ary constraint is given); and we can then propagate these constraints in this augmented net to obtain higher levels of consistency.

By successively adding higher level nodes to the network and propagating constraints in the augmented net, we can achieve k-ary consistency for all k. We do not need to restrict the given constraints to binary relations. Ruling out lower order inconsistencies in stages progressively reins in the combinatorial explosion. The final result is a network where the n-ary node specifies explicitly the n-ary constraint we seek to synthesize. No further search is required. The rest of this paper will present the algorithm, along with a sufficient theoretical base to justify its operation.

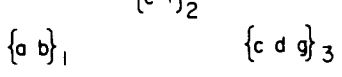
4. A Preliminary Example of the Synthesis Algorithm

I will give a crude example of the synthesis algorithm in operation, by way of motivation for the formal description which follows. The presentation in this section is intentionally sketchy.

Suppose we are given the following constraints on variables X_1, X_2, X_3: The unary constraint C_1 specifies that X_1 must be either a or b, i.e. $C_1 = \{a\ b\}$. Similarly $C_2 = \{e\ f\}$ and $C_3 = \{c\ d\ g\}$. The binary constraint on X_1 and X_2 specifies that either X_1 is b and X_2 is e, or X_1 is b and X_2 is f: $C_{12} = \{be\ bf\}$. Likewise $C_{13} = \{bc\ bd\ bg\}$ and $C_{23} = \{ed\ fg\}$.

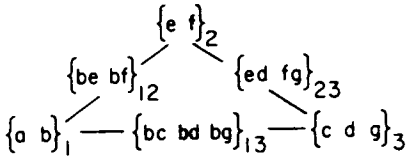
We wish to determine what choices for X_1, X_2, X_3, if any, can simultaneously satisfy all these constraints. We begin building the constraint network with three nodes representing the unary constraints on the three variables, as shown in Figure 5.

Fig. 5.



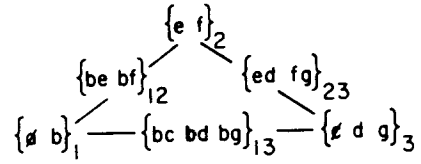
Next we add *nodes* representing the binary constraints, and link them to the unary constraints as shown in Figure 6 (e.g. $\{be\ bf\}_{12}$ represents C_{12}).

Fig. 6.



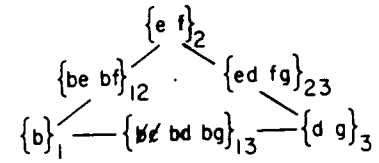
After we add and link node C_{12} we look at node C_1 and find that element a does not occur in any member of C_{12}. We delete a from C_1. Similarly, we delete c from C_3 after adding C_{23}. The constraint network now appears as in Figure 7.

Fig. 7.



Now from C_3 we look at C_{13} and find that there is an element bc in C_{13} which requires c as a value for X_3, while c is no longer in C_3. We remove bc from C_{13}, as in Figure 8.

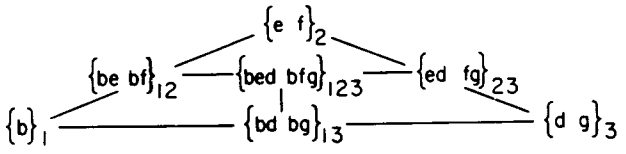
Fig. 8.



So far we have merely achieved a sort of "arc consistency" (though we indicate the restriction of the pair bc, as well as the elements a and c).

Next, we add a node for the *ternary* constraint. No order-three constraint was given originally, so we could assume initially the "nonconstraint," all possible triples. However, we will take advantage of the restrictions available from the binary and unary predicates to construct a more limited set of possibilities. C_1 and C_{23} together allow only the following set of triples: $\{bed\ bfg\}$. We use this as the ternary node and link it to the binary nodes as shown in Figure 9. (Choosing C_2 and C_{13} or C_3 and C_{12} would yield a larger ternary node.)

Fig. 9.



We look at the new node from its neighbors and vice versa, as we did earlier, to insure consistency of the sort we obtained earlier between neighboring nodes. C_{13} is consistent with the new node: bd is part of bed, bg part

of *bfg*. Similarly C_{12} and C_{23} are consistent with the ternary node. If necessary, we could propagate deletions around until local consistency is achieved on this augmented network. However, in this case, the network is already stable; no further changes are required.

The ternary node represents the synthesis of the given constraints. There are two ways to simultaneously satisfy the given constraints: $X_1 = b$, $X_2 = e$, $X_3 = d$ or $X_1 = b$, $X_2 = f$, $X_3 = g$.

5. Basic Definitions

This section presents several definitions needed to state the problem and its solution precisely.

We are given a set of variables $X_1, \ldots, X_n$ which may take on values from a set of universes $U_1, \ldots, U_n$, respectively. We will assume the U_i to be discrete, finite domains. Let $I = \{1\ 2 \ldots n\}$. Many of our definitions will be made for any subset $J \subseteq I$. We denote by X_J the indexed set of variables $\{X_j\}_{j \in J}$. A value a_i in U_i will be called an *instantiation* of X_i. An instantiation of a set of variables X_J, denoted by a_J, is an indexed set of values $\{a_j\}_{j \in J}$.

A *constraint* on X_J, denoted C_J, is a set of instantiations of X_J. The "indexed set" notation implies that there is a function, a, from J onto the instantiation a_J, which serves to indicate which member of a_J instantiates which variable: the value of a at j, denoted a_j, is the instantiation of X_j. We could also represent a_J as an ordered set or m-tuple, where m is the number of elements in the set J (called the cardinality of J and denoted $|J|$): $a_J = (a_{j_1}, \ldots, a_{j_m})$, a_{j_i} in U_{j_i}, $j_i < j_k$ for $i < k$, $i, k = 1, \ldots, m$. Thus C_J may be thought of as an m-ary relation. I have found it useful, however, to use set notation rather than refer to cross products or predicates in the presentation which follows. Given a_J, "$a_j \in a_J$" will denote the instantiation of X_j contained in a_J.

A *constraint expression* of order n is a conjunction of constraints $C = \bigwedge_{J \in 2^I} C_J$, one constraint for each subset J of I (except the empty subset). This represents the logical conjunction of the relations expressed by the C_J. Normally we will not be explicitly given constraints for all $J \subseteq I$; however, we can assume they exist, with no loss of generality, as the "nonconstraint" for X_J can always be specified: the set of all combinations of elements from the domains of the variables in X_J.

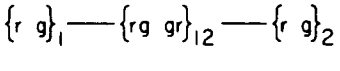
We say that an instantiation a_J *satisfies* a constraint C_J if $a_J \in C_J$. The instantiation a_J satisfies a constraint C_H, $H \subseteq J$, if the set $\{a_j \in a_J\}_{j \in H}$, which we call a_J restricted to H, is a member of C_H. An instantiation a_J, where $|J| = k$, *k-satisfies* a constraint expression of order $n \geq k$ if a_J satisfies the constraints C_H for all $H \subseteq J$. If an instantiation a_I n-satisfies the constraint expression of order n, we say that a_I *satisfies* the expression. A constraint expression C is *k-satisfiable* if for all cardinality k subsets J of I, there exists an a_J such that a_J k-satisfies C. If C of order n is n-satisfiable it is said to be *satisfiable*.

A constraint expression defines another constraint: the set of all instantiations a_I which satisfy the constraint expression. Our central problem is to synthesize the order n constraint, i.e. the n-ary relation, on X_I defined by the constraint expression, i.e. to determine explicitly the set of instantiations a_I which simultaneously satisfy all the given constraints. An instantiation a_I which satisfies C is called a solution of the constraint expression.

A *constraint network* of order k in n variables, $k \leq n$, is a set of constraints called *nodes*, N_J, for each $J \subseteq I$, $|J| \leq k$, where a *link* is said to exist between N_J and N_H if $H \subseteq J$ and $|H| = |J| - 1$. Linked nodes are called neighbors. A constraint network of order n in n variables will be called a *full* constraint network. A node N_J is said to *correspond* to a given constraint C_J if $N_J = C_J$, i.e. each instantiation of one is a member of the other. A full constraint network in n variables corresponds to a constraint expression of order n if each node N_J in the network corresponds to the constraint C_J in the expression. The order of a node N_J, or a constraint C_J, is the cardinality of J.

For example, the network in Figure 10 corresponds to the constraint expression $C = \bigwedge_{J \in 2^I} C_J$, where: $I = \{1\ 2\}$, $C_1 = \{r\ g\}$, $C_2 = \{r\ g\}$, $C_{12} = \{rg\ gr\}$. (I avoid set notation in the subscripts for simplicity.)

Fig. 10.



This is obviously a representation of the problem of coloring a two-node graph with two colors. Note that the constraint "not the same color" is represented by a "higher order" node, not a link as in Section 2.

As nodes are constraints we are able to restate all the above definitions involving satisfiability in terms of nodes and networks, rather than constraints and constraint expressions. In particular, we can speak of an instantiation a_J satisfying a node N_H for $H \subseteq J$. We also will want to talk about a_J satisfying N_H for $H \supset J$. We will say that a_J satisfies N_H, $H \supset J$, if there exists an a_H in N_H such that $\{a_j \in a_H\}_{j \in J} = a_J$, i.e. there is an instantiation which satisfies N_H whose restriction to J is a_J.

6. Constraint Propagation

We can now define the basic constraint propagation mechanism. To *locally propagate* the constraint N_J to a neighboring constraint N_H, remove from N_H all a_H which do not satisfy N_J. Global propagation is defined recursively. To *globally propagate* a constraint N_J through a neighboring constraint N_H: first locally propagate N_J to N_H; then, if anything was removed from N_H by the local propagation, globally propagate N_H through all its neighbors except N_J. To *propagate* a constraint N_J, globally

propagate N_J through all its neighbors. The propagation procedure is similar to an arc consistency algorithm. Mackworth discusses efficient serial algorithms for arc consistency [10]. Of course, a parallel implementation is possible.

A constraint network is said to be *relaxed* if we can propagate every constraint N_J in the network without causing any change (deletions from nodes) in the net. The *relaxation* of a constraint network is the network obtained by propagating all nodes of the network. (The propagation obviously terminates in a relaxed network.)

7. Synthesis Algorithm

We are now ready to state the synthesis algorithm. The claim, to be proven in Section 10, is that this algorithm, given a constraint expression, produces a constraint network whose order n node corresponds to the order n constraint defined by the constraint expression.

Algorithm:

Given $C = \wedge_{J \in 2^I} C_J$. We define the algorithm inductively:

Step 1. Construct a constraint network with nodes N_J corresponding to constraints C_J in the given constraint expression, for all $J \subseteq I$ of cardinality one.

Step $k + 1$. For all $J \subseteq I$ of cardinality $k + 1$:

Add the node N_J to the network corresponding to the given constraint C_J. Link N_J to all N_H such that H is a cardinality k subset of J.

Locally propagate to N_J from each of its neighbors. Propagate N_J.

For a constraint expression of order n, the algorithm is run for n steps. The result is a full constraint network, where N_I corresponds to C.

The next section will present several examples of the algorithm in operation. First a few general observations. The network produced by this algorithm is the relaxation of the network corresponding to C. We could have obtained it simply by building the corresponding order n network and propagating each node. By proceeding in stages we take advantage of the elimination of possibilities that may occur at each stage to mitigate combinatorial explosion. We take this principle further and propagate each node as it is added, before adding another. A good heuristic would be to add earlier those nodes which exert a heavy constraint, e.g. where C_J is small. The propagation of these constraints may eliminate elements from nodes used in constructing later constraints. If C_J is the nonconstraint we can construct N_J initially from some N_H and N_{J-H}, where H is a cardinality k subset of J, preferably the one for which $|N_J| \times |N_{J-H}|$ is a minimum. (Add to each member of N_H each member of N_{J-H}.)

Other refinements are clearly possible. Provision should be made for early termination, e.g. as soon as one node becomes empty. Propagation can be simplified, e.g. by noting nonconstraints, or using complements of nodes. Additional links could permit direct propagation between a node N_J and the nodes for all subsets of J.

It is generally redundant to require all nonconstraint nodes; basically we only need one "path" up to the n-ary node for every "real" constraint. Consider a constraint expression on four variables where only the binary constraints are really specified (the others are nonconstraints). Only the binary constraints can really have any effect on the global solution. Three ternary nodes are sufficient for the network constructed by the algorithm. If the fourth ternary node rules out any element of the order-four node, it is only reflecting a binary constraint, which is reflected in one of the other ternary nodes. On the other hand we may be interested in the effects on nonconstraints of the propagation process. In general the pruning process of the algorithm progressively makes explicit at N_J restrictions on instantiations of X_J that are not originally given by C_J, but rather implied by the other constraints. In the final network produced by the algorithm every member of every N_J is part of some solution of the constraint expression. (In particular, we have derived the "minimal" network, Montanari's "central problem" [14].)

8. Further Examples

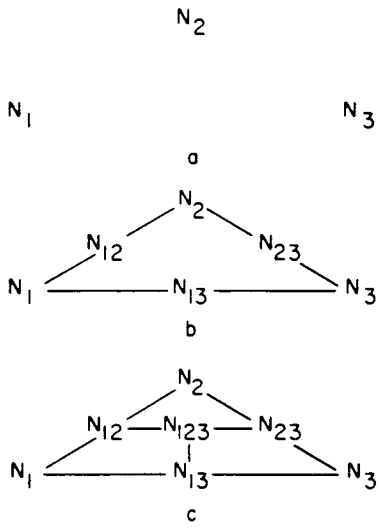
As the synthesis problem is such a general one, the synthesis algorithm has many potential applications. Graph problems, of course, lend themselves particularly to a constraint network formulation. I present in this section two applications which will serve to illustrate the algorithm, and are of some independent interest as well.

As we would expect from the discussion in Section 2, the graph coloring problem can easily be represented as a constraint network. Given a graph G, and a set of colors, we construct a constraint network from G as follows: each vertex of G is replaced by the unary constraint node representing the set of colors. If there is an edge between vertices in G, we replace it by a binary constraint linked to the unary nodes which represents "is not the same color as." If there is no link between vertices in G, we add the nonconstraint between the nodes.

Let us consider two examples. First consider the problem of coloring a complete three-vertex graph with three colors. Figures 11(a), 11(b), and 11(c) show the constraint network after steps one, two, and three of the algorithm, where the nodes N_1, N_2, N_3 are all the set $\{r\ g\ b\}$, N_{12}, N_{13}, and N_{23} all equal $\{rg\ rb\ gr\ gb\ br\ bg\}$ and $N_{123} = \{rgb\ rbg\ brg\ bgr\ grb\ gbr\}$, the six possible colorings. We could construct a network of the sort we used in Section 2 for this problem. However the network would be path consistent; arc and path consistency algorithms would not rule out any elements at the nodes.

Consider now the problem of coloring a complete four-vertex graph with three colors, which we used in Section 2 to illustrate that path consistency is not a

Fig. 11.



sufficient condition for satisfiability. After the third step of the algorithm we would have four ternary nodes, each equal to the ternary node in the previous example.

At the beginning of the fourth step we use N_{123} and N_4 to construct an order-four node: N_{1234} = {*rgbr rgbg rgbb rbgr rbgg rbgb brgr brgg brgb bgrr bgrg bgrb grbr grbg grbb gbrr gbrg gbrb*}. Local propagation from the other ternary nodes quickly reduces the order-four node to the empty set (and this constraint propagates back down to remove all elements from all the nodes). No instantiation of the order-four node will simultaneously satisfy the four ternary nodes. Unsatisfiability is demonstrated.

These examples are rather perverse cases, of course, though they do illustrate points with respect to the discussion in Section 2. Applications in the scene labeling domain generally involve more propagation than occurs in these coloring problems. The synthesis algorithm does function as a test for whether scenes can be labeled in the manner described in Section 2. It also finds all the interpretations in an ambiguous scene.

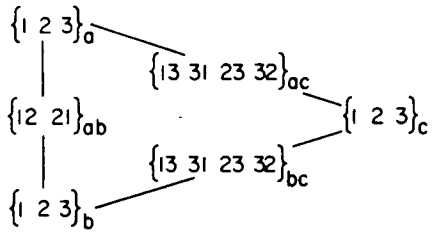
For another example, we take graph isomorphism. Given two graphs G and H, which we wish to test for isomorphism, construct a constraint network from G as follows. (If H has more vertices than G the algorithm will seek isomorphic mappings of G onto subgraphs of H.) Replace each vertex of G with a unary constraint node containing all the vertices of H. (If we allow loops, edges from a vertex to itself, the unary constraint for a vertex in G with a loop will be "has a loop in H," for a vertex in G without a loop, "has no loop in H." We could also incorporate additional unary constraints such as the degree of the vertex [22].) Replace each edge between vertices a and b in G with a binary constraint node, linked to the unary nodes for a and b. This binary node will represent the constraint "these two (distinct) vertices share an edge in H." In other words, the binary constraint will contain a pair xy if and only if there is an edge between x and y in H. Between two vertices which do not share an edge in G we also place a binary node, linked to them, but representing the constraint "these two (distinct) vertices do not share an edge in H."

For example: given the graphs G and H in Figure 12,

Fig. 12.

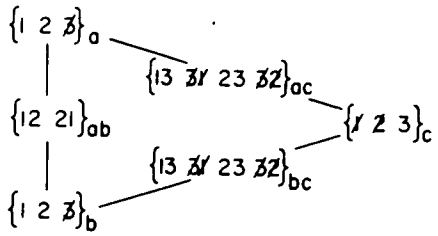
we produce the constraint network in Figure 13.

Fig. 13.



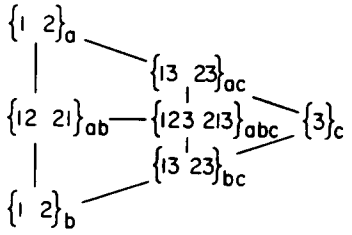
Propagating constraints, we obtain the network in Figure 14.

Fig. 14.



Now adding the ternary node we obtain Figure 15.

Fig. 15.



This network is relaxed. The ternary node represents the two possible isomorphisms: $a \rightarrow 1$, $b \rightarrow 2$, $c \rightarrow 3$ and $a \rightarrow 2$, $b \rightarrow 1$, $c \rightarrow 3$. (The algorithm also finds isomorphic subgraphs along the way.)

In the above applications, the desired global state is defined in terms of local constraints. Often one first faces an analysis problem: choosing, or learning, a set of local constraints that specify or approximate the desired global state [20]. (An important concern will be the choice of a

"good" constraint expression, i.e. one that can be synthesized efficiently.) As various applications are explored, it will, of course, be equally important to develop theoretical methods for analyzing the performance of the algorithm in a given domain.

9. Compatibility, Completeness, and Consistency

The synthesis algorithm operates by removing higher and higher level inconsistencies. In this section, I define this sequence of consistency states, and also define concepts of compatibility and completeness which I will want to apply to constraint networks.

A node N_J of order k is *k-compatible* with a constraint expression C if all members of N_J k-satisfy C. A constraint network of order k or greater is k-compatible with C if all nodes of order k are k-compatible with C. If a full constraint network of order n is n-compatible with a constraint expression C of order n we say that it is *compatible* with C.

A node N_J of order k is *k-complete* for C if any instantiation a_J which k-satisfies C is a member of N_J. A network N is k-complete for C if every node of order k is k-complete. An n-complete full constraint network of order n is said to be *complete.*

A constraint network of order k or greater, in n variables, is *k-consistent* if for any set X_H of $k-1$ variables, any instantiation a_H of X_H which $(k-1)$-satisfies N_H, and any choice of a kth variable, X_i, there exists an instantiation of X_i which combines with a_H to k-satisfy N_J, where J is the union of H and $\{i\}$.

K-consistency generalizes the notions of consistency introduced in Section 2, in particular 1-consistency corresponds to node consistency, 2-consistency to arc consistency, and 3-consistency to path consistency. K steps of the synthesis algorithm produce a network that is j-consistent for all j less than or equal to k. After k steps of the algorithm, therefore, we can do backtrack tree search on the remaining values knowing that backup will not be initiated in the first k levels.

K steps of the algorithm also achieve k-compatibility and k-completeness (Corollary 3 in the following section), so we can choose an order k node and use its members as the alternative paths through the first k levels of a search tree, only really doing tree search on the remaining $n-k$ nodes. Of course, if we have carried the synthesis algorithm to completion, the members of the order n node are the solutions and no further search is required. (The discussion of k-consistency in [6] is faulty. For another recent approach to constraining backtrack search see [17].)

10. Synthesis Theorem

We are now ready to state the theorem which justifies the synthesis algorithm.

THEOREM. *The relaxation of the network corresponding to a constraint expression* $C = \wedge_{J \in 2^I} C_J$ *is compatible and complete with respect to C.*

The proof will be by induction. Compatibility and completeness of order one are obvious. Our induction hypothesis is that the network is k-compatible and k-complete; we wish to prove $k+1$-compatibility and $k+1$-completeness.

Consistency. We want to show that all N_J, for J any cardinality $k+1$ subset of I, are $k+1$-compatible. N_J, before relaxation, corresponded to C_J, so included nothing which did not satisfy C_J; relaxation does not add any elements to a node. Suppose there exists an a_J in N_J such that a_J does not satisfy C_H, for some proper subset H of J, i.e. a_J restricted to H is not a member of C_H. Pick a set G of cardinality k such that $H \subseteq G \subset J$. Because of the local propagation during the relaxation process, we know that a_J satisfies N_G. Thus a_J restricted to G, a_G, is a member of N_G. As the network is k-compatible a_G restricted to H is a member of C_H. But a_G restricted to H is a_J restricted to H: contradiction.

Completeness. Consider any a_J not in N_J, for J any cardinality $k+1$ subset of I. There are two possibilities. If a_J was not in N_J before relaxation, then a_J does not satisfy C_J. If a_J was removed during the relaxation process, then a_J does not satisfy N_H for some cardinality k subset H of J; by the induction hypothesis a_J restricted to H does not k-satisfy C. In either case, a_J does not $k+1$-satisfy C.

There are several immediate corollaries.

COROLLARY 1. *N_I corresponds to the order n constraint defined by the constraint expression C.*

COROLLARY 2. *C is satisfiable if and only if N_I is not the empty set.*

COROLLARY 3. *The constraint network constructed by the synthesis algorithm operating on a constraint expression C is k-compatible with and k-complete for C after step k. The network constructed by the algorithm is compatible with and complete for C, and N_I corresponds to C.*

Thus the algorithm can be used to find explicitly all the solutions to a constraint satisfaction problem. Alternatively, the algorithm can be run for k steps as a preprocessor to simplify subsequent backtrack search.

Received August 1976; revised March 1978

References

1. Barrow, H.G., and Tenenbaum, J.M. MSYS: A system for reasoning about scenes. Tech. Note 121, A.I. Ctr., Stanford Research Inst., Menlo Park, Calif., April 1976.
2. Bobrow, D.G., and Raphael, B. New programming languages for artificial intelligence research. *Computing Surveys 6*, 3(1974), 153–174.
3. Clowes, M.B. On seeing things. *Artif. Intell. 2* (1971), 79–116.
4. Cook, S.A. The complexity of theorem-proving procedures. Conf. Rec. 3rd Annual ACM Symp. Theory of Computing, 1971, pp. 151–158.
5. Fikes, R.E. REF-ARF: A system for solving problems stated as procedures. *Artif. Intell. 1*, 1(1970), 27–120.
6. Freuder, E.C. Synthesizing constraint expressions. A.I. Memo 370, A.I. Lab., M.I.T., Cambridge, Mass., 1976.
7. Grossman, R.W. Some data base applications of constraint expressions. TR 158, Lab. for Computer Science, M.I.T., Cambridge, Mass., 1976.

8. Horn, B.K.P. Determining lightness from an image. *Comptr. Graphics and Image Processing 3*, 4(Dec. 1974), 277–299.
9. Huffman, D.A. Impossible objects as nonsense sentences. In *Machine Intelligence 6*, B. Meltzer and D. Michie, Eds., Edinburgh U. Press, Edinburgh, 1971, pp. 295–323.
10. Mackworth, A.K. Consistency in networks of relations. *Artif. Intell. 8*, 1(1977), 99–118.
11. Marr, D. Simple memory: A theory for archicortex. *Philos. Trans. Roy. Soc. B. 252* (1971), 23–81.
12. Marr, D., and Poggio, T. Cooperative computation of stereo disparity. A.I. Memo 364, A.I. Lab., M.I.T., Cambridge, Mass., 1976.
13. Minsky, M., and Papert, S. *Perceptrons.* M.I.T. Press, Cambridge, Mass., 1968.
14. Montanari, U. Networks of constraints: Fundamental properties and applications to picture processing. *Inform. Sci. 7*, 2(April 1974), 95–132.
15. Rosenfeld, A., Hummel, R., and Zucker, S.W. Scene labelling by relaxation operations. *IEEE Trans. Systems, Man, and Cybernetics SMC-6*, 6(June 1976), 420–433.
16. Sussman, G.J., and McDermott, D.V. From PLANNER to CONNIVER—a genetic approach. Proc. AFIPS 1972 FJCC, Vol. 41, AFIPS Press, Montvale, N.J., pp. 1171–1179.
17. Sussman, G.J., and Stallman, R.M. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. A.I. Memo 380, A.I. Lab., M.I.T., Cambridge, Mass., 1976.
18. Tenenbaum, J.M., and Barrow, H.G. IGS: A paradigm for integrating image segmentation and interpretation. In *Pattern Recognition and Artificial Intelligence*, C. H. Chen, Ed., Academic Press, New York, 1976, pp. 472–507.
19. Ullman, J.R. Associating parts of patterns. *Inform. and Control 9*, 6(Dec. 1966), 583–601.
20. Ullman, J. R. *Pattern Recognition Techniques.* Crane Russak, New York, 1973.
21. Ullman, J.R. An algorithm for subgraph isomorphism. *J.ACM 23*, 1(Jan. 1976), 31–42.
22. Unger, S.H. GIT—a heuristic program for testing pairs of directed line graphs for isomorphism. *Comm. ACM 7*, 1(Jan. 1964), 26–34.
23. Waltz, D.L. Generating semantic descriptions from drawings of scenes with shadows. AI-TR-271, A.I. Lab., M.I.T., Cambridge, Mass., 1972.
24. Zucker, S.W. Relaxation labelling, local ambiguity, and low-level vision. In *Pattern Recognition and Artificial Intelligence*, C. H. Chen, Ed., Academic Press, New York, 1976, pp. 593–616.

Operating Systems

R.S. Gaines
Editor

On-the-Fly Garbage Collection: An Exercise in Cooperation

Edsger W. Dijkstra
Burroughs Corporation

Leslie Lamport
SRI International

A.J. Martin, C.S. Scholten, and
E.F.M. Steffens
Philips Research Laboratories

As an example of cooperation between sequential processes with very little mutual interference despite frequent manipulations of a large shared data space, a technique is developed which allows nearly all of the activity needed for garbage detection and collection to be performed by an additional processor operating concurrently with the processor devoted to the computation proper. Exclusion and synchronization constraints have been kept as weak as could be achieved; the severe complexities engendered by doing so are illustrated.

Key Words and Phrases: multiprocessing, fine-grained interleaving, cooperation between sequential processes with minimized mutual exclusion, program correctness for multiprogramming tasks, garbage collection

CR Categories: 4.32, 4.34, 4.35, 4.39, 5.24

1. Introduction

In any large-scale computer installation today, a considerable amount of time of the (general purpose)

Authors' addresses: E.W. Dijkstra, Burroughs, Plataanstraat 5, 5671 Al Nuenen, The Netherlands; L. Lamport, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025; A.J. Martin, C.S. Scholten, and E.F.M. Steffens, Philips Research Laboratories, 5656 AA Eindhoven, The Netherlands.

For reference purposes a glossary of names has been added as an Appendix.