Solving Constraint Satisfaction Problems Using Finite State Automata

Nageshwara Rao Vempaty,
Department of Computer Sciences,
University of Central Florida,
Orlando, FL - 32816.
E-mail: vnrao@teja.cs.ucf.edu

Abstract

In this paper, we explore the idea of representing CSPs using techniques from formal language theory. The solution set of a CSP can be expressed as a regular language; we propose the minimized deterministic finite state automaton (MDFA) recognizing this language as a canonical representation for the CSP. This representation has a number of advantages. Explicit (enumerated) constraints can be stored in lesser space than traditional techniques. Implicit constraints and networks of constraints can be composed from explicit ones by using a complete algebra of boolean operators like AND, OR, NOT, etc., applied in an arbitrary manner. Such constraints are stored in the same way as explicit constraints — by using MDFAs. This capability allows our technique to construct networks of constraints incrementally. After constructing this representation, answering queries like satisfiability, validity, equivalence, etc., becomes trivial as this representation is canonical. Thus, MDFAs serve as a means to represent constraints as well as to reason with them. While this technique is not a panacea for solving CSPs, experiments demonstrate that it is much better than previously known techniques on certain types of problems.

Overview

The finite domain constraint satisfaction problem (CSP)[1] is well known in Artificial Intelligence. It has been investigated in the past by a number of researchers in different contexts.(For a recent survey, refer to [Kumar, 1992].) A CSP involves a (finite) set of variables, a (finite) set of values for the variables and a set of constraints, each of which is a relation on the values of some of the variables. Constraint Satisfaction involves identifying value assignments to variables that satisfy all the constraints. The problem is formally defined in the next section.

As might be expected, many sequential algorithms have been developed for solving CSPs; these include backtracking, arc consistency [Waltz, 1975; Mackworth, 1977], path consistency [Freuder, 1988], linear programming techniques [Rivin and Zabih, 1989], etc.. The representation often used for expressing a constraint and storing it, is the list of tuples form, in which the tuples of values satisfying the constraint are enumerated. Since such lists may become cumbersome as the number of variables in the constraint increases, it became customary to restrict the constraints to be unary or binary. A CSP that satisfies this restriction is called a Binary CSP (BCSP). A general CSP can be converted into an equivalent BCSP, but this process usually entails the introduction of new variables and constraints, and hence an increase in problem size. Constraint propagation (discrete relaxation) algorithms construct new lists of tuples of more restrictive constraints that can be extended to solutions of the CSP[Montanari, 1974; Dechter and Meiri, 1989; Rossi and Monatanari, 1990]. Backtracking algorithms enumerate the entire universe of tuples possible, to construct a list of tuples satisfying all the constraints[Nadel, 1988]. However the list of tuples representation of a constraint has two important drawbacks (i) many types of constraints require an exponential space. For example, consider a CSP on n variables $v_1, v_2, \cdots, v_n$, with m values for each variable $a_1, a_2, \cdots, a_m$. Consider a k-ary constraint that requires at least one of the variables to take a value of a_1. The number of tuples in this constraint is $O((m-1)^k)$ and the size of a list representing this grows exponentially in k. Now consider the problem of finding all solutions to such a constraint (referred to as VALidity query later on in this paper). Clearly, an enumeration of all solutions requires exponential space and time. Is there a better technique? (ii) some operations on constraints that lead to new constraints require an exponential amount of computation time. For example, the typical operation of negating a constraint, requires exponential time. In the past, this problem was circumvented by specifying simple constraints explicitly as tuple lists and a CSP is specified as an 'and' of the simple constraints. This type of representation is difficult to manipulate and hence a one time algorithm is used to solve the CSP after it has been completely specified. Implicit constraints were specified by 'and'ing explicit constraints; it is difficult to extract the solution set from such a representation. In this paper, we develop a representation that is more compact in many cases and permits us to use arbitrary logical operators like 'and', 'or', 'not', etc.. It is some times advantageous to construct CSPs and new implicit

[1] Through out this paper, we use the acronym CSP to refer to the finite domain constraint satisfaction problem.

constraints incrementally[Dechter and Dechter, 1988; Mittal and Falkenheiner, 1990]. Further, the list of tuples representation is not canonical; hence problems such as determining the equivalence of two CSPs require enormous computation. Our representation is canonical and permits incremental construction of CSPs.

In this paper, we propose the minimal deterministic finite state machine (MDFA) accepting (exactly) the satisfying tuples of a constraint as a means to handle CSPs. It is well known that MDFAs are canonical up to a renaming of states [Hopcroft and Ullman, 1979]. Thus if we use a MDFA to represent a CSP, we have a compact and a canonical representation. New constraints and networks of constraints can be constructed from known ones using arbitrary operators like intersection (**and**), union (**or**), difference (**diff**), etc.. Thus CSPs can be specified with a complete algebra of operators, instead of just using **and** as in previous literature. A detailed discussion of this representation is presented in the next section. To solve the classical network of constraints problem with this representation, explicit constraints are first converted into MDFAs. MDFAs to represent implicit constraints and networks of constraints are constructed directly from these automata by applying the corresponding operations. Algorithms for constructing such a representation and processing queries are discussed in the third section.

The computational difficulty of the new approach lies in constructing MDFAs corresponding to implicit constraints. Once the MDFAs are constructed, answering queries is easy, since our representation is canonical. The strengths and weaknesses of this technique are analyzed and a comparison to past work is made in the fourth section. The new representation is much more effective on certain problems; experiments illustrating this are presented. Conclusions and scope for future research are discussed in the last section.

Representation of CSPs using MDFAs

Finite State Automata(FSA) were used to represent and manipulate boolean functions by Clarke et. al. in [Kimura and Clarke, 1990; Clarke *et al.*, 1991]. These are related to Binary Decision Diagrams (BDDs) developed by Akers[Akers, 1982], Bryant[Bryant, 1986] and McCarthy. We extend this representation to deal with constraints and CSPs. We first generalize the definition of a CSP and then give a canonical representation using MDFAs.

Terminology

A system to represent and manipulate CSPs has a set of n variables $\mathcal{V} = \{v_1, v_2, \cdots, v_n\}$ and a domain of m values for the variables, $\mathcal{D} = \{a_1, a_2, \cdots, a_m\}$,[2] with which constraints may be expressed. A k-ary constraint C over the variables $\{v_{i1}, v_{i2}, \cdots, v_{ik}\}$ is a subset of $\mathcal{D}^k$ and it specifies permissible combinations of values for the variables $\{v_{i1}, v_{i2}, \cdots, v_{ik}\}$. It is a k-ary relation. An n-tuple of values (also called a value assignment) is a mapping from variables $\mathcal{V}$ to values $\mathcal{D}$; it specifies a value for each variable. A value assignment satisfies a constraint if it gives a combination of values to variables that is permitted by the constraint; otherwise it falsifies it. Traditionally, CSPs were specified as a set of constraints that need to be satisfied together. For example, the N-queens problem involves placing N queens on an NxN chess board such that (i) queen i is placed in row i and (ii) no two queens attack each other, by a column, major diagonal or minor diagonal. The 3-queens problem is graphically illustrated in Figure 1.

A k-ary constraint C can be extended to an equivalent n-ary constraint, by adding in all the remaining variables in $\mathcal{V}$ and allowing them to take any combination of values from $\mathcal{D}$. By using a special wild card value, $\star$, to mean any, every k-tuple of C is now made into an n-tuple. An ordering of the variables $v_1, v_2, \cdots v_n$ is a permutation of these variables. A fixed ordering of variables is used by our system to list out the components of tuples for all the constraints. Henceforth, every constraint is considered to be n-ary and uses the ordering $\{v_1, v_2, \cdots, v_n\}$, to list its set of satisfying tuples $\mathcal{S}$. *Semantically speaking, this list of tuples defines the constraint.* Note that such a list of ordered tuples forms a canonical representation if we sort the lists using a lexicographic ordering. But this representation appears to be cumbersome. (For ease of presentation, it is assumed that the set of variables we are working with is fixed and it is denoted by $\mathcal{V} = \{v_1, v_2 \cdots v_n\}$. The set of values is also assumed to be fixed and is denoted by $\mathcal{D} = \{a_1, a_2 \cdots a_m\}$. We use C_1, C_2, $\cdots$ to denote constraints.)

We now define the syntax of an algebra for expressing constraints. New constraints can be constructed from a given set of constraints by using using logical operators **and**, **or** and **diff**. The semantics of these operators are given by the intersection, union and difference respectively, of the associated sets of satisfying n-tuples. A Constraint Satisfaction Problem (CSP) specifies some explicit constraints by enumerating their tuples and implicit constraints by using logical operators over the explicit constraints. An algebra for specifying constraints is formally defined below.

Definition 1 *Syntactic definition of a constraint :-*

1. *the empty set Φ is a constraint. It is the only unsatisfiable constraint.*
2. *a set containing a single ordered n-tuple of values is a constraint. Those with more tuples can be constructed using the* **or** *operator defined below.*
3. *If C_1 and C_2 are constraints, then so are $(C_1 \bigwedge C_2)$, $(C_1 \bigvee C_2)$ and $(C_1 - C_2)$. $(C_1 \bigwedge C_2)$ is the constraint that lists all tuples that belong to both C_1 and C_2. $(C_1 \bigvee C_2)$ is the constraint that lists all tuples that belong to C_1 or C_2. $(C_1 - C_2)$ is the constraint whose tuples are present in C_1, but not in C_2. Other operators like complementation, exclusive-or, etc., are analogously defined.*

A system handling CSPs needs to represent constraints defined this way and solve queries on them.

[2] Having a different domain of values for each variable is equivalent and can be handled similarly

Queries to the CSP system include the following types:[3]

1. SATisfiability : Is $C = \Phi$? An answer to this query is expected to return a satisfying tuple if one exists.
2. VALidity : What is the set of tuples satisfying a constraint C ?
3. EQUivalence : Are C_1 and C_2 satisfied by the same set of tuples ?

MDFA representation

An ordered list of values of an n-tuple denotes a string of length n over the alphabet $\mathcal{D}$. Recall that the semantic definition of a constraint says that it is a finite set of n-tuples. In formal language theory, viewing the tuples as strings, a constraint is a finite set of strings of length n. Since all finite sets are regular, a constraint is a regular language and hence can be recognized by using finite state machines. A deterministic finite state automaton (DFA) is a five tuple $(Q, \mathcal{D}, \delta, S, F)$ [Hopcroft and Ullman, 1979], where (i) Q is a (finite) set of states (ii) δ is a transition function defined from $Q \times \mathcal{D}$ to Q (iii) S denotes the distinguished start state and (iv) F is the set of accepting states of the machine. A DFA is said to accept a string w over the alphabet $\mathcal{D}$ if and only if starting the machine in state S with w on the tape leads to an accepting state. A DFA can be represented as a graph, using the nodes to represent states and the edges labeled by alphabet, to represent the transition function. The language accepted by a DFA is the set of strings accepted by it and it is always a regular language. A minimal deterministic finite state automaton (MDFA) for a language L is the deterministic finite state machine with the least number of states among all DFAs accepting L. For example, the MDFA in Figure 2 accepts all strings of length n on $\mathcal{D}$. This MDFA liberally accepts the entire universe $\mathcal{D}^n$. Compare this representation with the list of tuples representation for an n-ary constraint that permits any n-tuple as a solution.

The MDFA corresponding to a language is canonical up to a renaming of states; refer to [Hopcroft and Ullman, 1979] for a proof of this and more details on finite state machines. In terms of graph theory, the minimized finite state automaton (MDFA) is canonical up to an isomorphism. So we can use them as a canonical representation of constraints. Formally —

Definition 2 *The MDFA corresponding to a constraint is the minimized deterministic finite state automaton that accepts (all and only) the tuples satisfying the constraint.*

For example, the MDFA in Figure 2 represents the constraint C_u satisfied by all tuples of $\mathcal{D}^n$. Even this trivial example illustrates that the MDFA representation is compact. We require only n nodes and mn arcs (see Figure 2). The list of tuples representation requires a space of $\mathcal{D}^n$. In this representation, we use MDFAs to represent explicit constraints as well as to represent a network of constraints which when 'and'ed together specify a CSP in the traditional sense. Since all the constraints in our system are expressed over the same variable set $\mathcal{V}$, equivalent constraints are represented by identical MDFAs. The MDFA representation of a constraint is minimal over all deterministic finite state machines representing the constraint. Recall that the list of tuples representation itself defines a DFA. Hence, the MDFA representation is smaller than the list of tuples representation as well.

[3] Other queries like IMPlication, SUBsumption, etc., can be handled with the algebraic operators in a similar fashion

Algorithms to handle MDFAs

In this section, we present algorithms for constructing the MDFA representation of constraints and CSPs, and algorithms for answering queries.

Algorithms for constructing the MDFAs

Constructing the minimal canonical form involves two steps (i) first construct a DFA that recognizes the constraint and (ii) minimize this machine to get the canonical MDFA representation.

Constructing a finite state machine Given a constraint defined by 1, we can construct a DFA by the following steps. We are in fact replaying the syntactic definition of the constraint in a bottom up fashion.

1. The empty constraint is accepted by the empty DFA. This DFA has no states at all and hence has no satisfying tuple.
2. The constraint which has a single satisfying tuple $\{(a_1, a_2, \cdots, a_n)\}$ has the DFA in Figure 3. It accepts only the string $a_1 a_2 \cdots a_n$. Constraints with more tuples can be constructed using the **or** operator.
3. Given two constraints C_1 and C_2, with DFAs M_1 and M_2 representing them, we can construct DFAs for $C_1 \bigvee C_2$, $C_1 \bigwedge C_2$, $C_1 - C_2$, etc., as follows. Construct a new machine M_p, traditionally called the product machine [Hopcroft and Ullman, 1979; Kimura and Clarke, 1990], that simulates the working of both the machines M_1 and M_2. The state of M_p is a pair, (q_1, q_2), where q_1 is used to simulate the working of M_1 and q_2 that of M_2. On reading alphabet a in the state (q_1, q_2), this new machine transits to state $(\delta_1(q_1, a), \delta_2(q_2, a))$. It thus traces the working of M_1 and M_2 together. We can deal with various operators, by defining the accepting states of M_p, as follows :-
   - **Or**:- To construct the union of two constraints, the state (q_1, q_2) of the product machine is made an accepting state iff q_1 is an accepting state of M_1 **or** q_2 is an accepting state of M_2.
   - **And**:- To construct the intersection of two constraints, the state (q_1, q_2) of the product machine is made an accepting state iff q_1 is an accepting state of M_1 **and** q_2 is an accepting state of M_2.
   - **Diff**:- To construct the difference of two constraints, the product machine is made to accept a tuple in a state iff M_1 reaches an accepting state and M_2 reaches a rejecting state.

We denote the number of states of M_i by $|M_i|$. The product DFA of M_1 and M_2 has at most $|M_1| \star |M_2|$ states in the worst case. This implies that the worst

case complexity of the product construction algorithm is $|\mathcal{D}| \star |M_1| \star |M_2|$. In the best case, its complexity is $O(|M_1| + |M_2|)$.

Minimizing a DFA to the MDFA The DFA constructed in the previous step should be minimized to get our canonical MDFA representation. This can be done using the standard algorithm in page 70 of [Hopcroft and Ullman, 1979]. If $|P|$ is the number of states of the product DFA we start with, the time complexity of this algorithm is $O(|\mathcal{D}||P|^2)$. Since our DFAs are directed acyclic graphs, we can employ the linear time minimization algorithm in [Hopcroft and Ullman, 1979] for better performance.

Answering Queries with MDFA representation

Since our MDFA representation is canonical and minimal, answering queries can be performed efficiently once MDFAs have been constructed. In the following discussion, it is assumed that MDFAs for C_1, C_2, $\cdots$ have been constructed and they are named M_1, M_2, $\cdots$. Each of the C_i's may have been specified as an explicit constraint, implicit constraints or a network of constraints.

- SATisfiability: C_1 is satisfiable iff M_1 has an accepting state. A satisfying tuple for C_1 is obtained by tracing any path from source to sink in M_1. This takes $O(|\mathcal{V}|)$ time. Since our representation is canonical, only the empty constraint C_Φ is unsatisfiable and it has a unique representative M_Φ.
- VALidity: The set of all satisfying tuples of C_1 can be enumerated by a depth-first traversal of M_1. Every visit to the accepting state gives us a new satisfying tuple. M_1 itself is a compact representation of the satisfying set of C_1 and can be used as a response to a VALidity query.
- EQUivalence: To check if C_1 and C_2 have the same satisfying tuples, we can simply check if the corresponding MDFAs M_1 and M_2 are identical. Since our MDFAs are directed acyclic graphs, this can be done in $O(|\mathcal{D}|(|M_1| + |M_2|))$ time.

Solving CSPs with MDFAs

In order to solve a CSP with the above techniques, we execute the following steps: (i) Construct MDFAs for all explicitly enumerated constraints. (ii) For implicit constraints, networks of constraints and CSPs algebraically constructed from explicit constraints, build MDFAs using product and minimization algorithms. (iii) Solve queries like SATisfiability, VALidity, etc., using query processing algorithms. A few general comments are worth mentioning, to aid the reader in correlating with previous techniques for solving a network of constraints. The MDFA constructed has a structural correspondence to the decision tree of top down backtrack search algorithms for solving CSPs[Nadel, 1988; Grimson, 1990]. The MDFA is a graph with equivalent states of the tree reduced to a single representative and the failure nodes of the tree being completely discarded. MDFA construction is a bottom up approach, where constraints are logically 'and'ed to define a network of constraints. There is also an interesting difference between the way two constraints C_1 and C_2 are 'and'ed with MDFAs and with discrete relaxation. Bottom up approaches like discrete relaxation and its isomorphic relative, resolution[DeKleer, 1989], add a new restrictive constraint that is logically implied by C_1 and C_2. The MDFA approach presented here replaces C_1 and C_2, by their exact logical 'and' — $C_1 \bigwedge C_2$.

Example of MDFA construction

To illustrate the technique of solving CSPs using MDFAs, consider the (toy) problem of 3-queens, graphically shown in Figure 1. We follow the convention that queen i is placed in row i. There are three unary constraints C1, C2 and C3, each specifying that a queen can be in any of the three columns. The three binary constraints C12, C23 and C31 specify that no pair of queens attack each other. Figure 5 shows the important steps involved in solving this CSP with MDFAs. First, we convert the explicitly specified constraints C1, C2, C3, C12, C23 and C31 into MDFAs. Then we **and** these constraints in this order to get the constraint that specifies the 3-queens problem depicted by the network of Figure 1. As the final MDFA is empty, 3-queens problem does not have a solution.

Even with this toy example, we can observe that with a MDFA representation, we deal with explicit constraints, implicit constraints and networks of constraints (CSPs) in the same manner. One can reuse the MDFAs built for C1,C23, etc., to build other constraints and other CSPs and also to answer queries. In an application where constraint networks once specified are reused again and again, the MDFA representation saves recomputation time. For example, to solve another queens problem in which queen1 is allowed to attack queen2, we simply ignore C12 and **and** the rest of the constraints we built for 3-queens. We can also infer that C1, C2 and C3 can be ignored. This type of information is lost in the traditional CSP algorithms, as they are one time algorithms.

Overall Complexity of the New Strategy

Constructing MDFAs for explicit constraints, minimization and answering queries can be performed in near linear time. When an implicit constraint C is composed from two constraints C_1 and C_2 with a binary operator, we need to construct the product machine from M_1 and M_2 and then minimize it. The complexity of the product-construction algorithm varies. It depends on the (i) size of the MDFAs involved and (ii) type of the satisfying sets for the constraints involved. The time consumed by the product construction algorithm, given two DFAs M_1 and M_2, is $\Theta(M_1 + M_2)$ in the best case and $O(M_1 \star M_2)$ in the worst case. In the first case, the product DFA is small. The second case can lead to blow up in size. Since the construction of an MDFA to solve a network of constraints uses the product-construction algorithm repeatedly, its run time is some times linear, some times quadratic and it may even be exponential in the worst case. The time complexity also depends on the ordering used to enumerate variables and tuples. Heuristics

for variable ordering devised for previous CSP algorithms [Purdom *et al.*, 1981; Dechter and Meiri, 1989; Zabih and McAllester, 1988] seem to work very well in practice. The algorithms to handle query processing, after MDFAs have been constructed, are quite efficient. Thus, once the MDFA representation has been constructed, it can be manipulated in almost linear time. This is not the case with the list of tuples representation.

Experimental Comparison with Previous Methods

We performed some experiments to compare the MDFA based techniques with other known algorithms to solve CSPs.

The forward checking algorithm (FC) for solving CSPs combines backtracking with forward checking. It is also called backtracking combined with directed arc consistency (DAC) in [Dechter and Meiri, 1989]. In [Dechter and Meiri, 1989], this algorithm was shown to be superior to path consistency and adaptive consistency algorithms on randomly generated CSP instances. We compared the algorithms presented in this paper with FC enhanced with dependency directed backtracking[Stallman and Sussman, 1977] to solve VALidity query on randomly generated CSP instances. On hard instances, the MDFA approach runs faster by a factor of 10 – 100, both in terms of time and number of states generated. We also compared the MDFA approach with forward checking and hyperresolution on the pigeon hole clauses. (Hyperresolution is equivalent to discrete relaxation for this problem by a result of De Kleer[DeKleer, 1989].) Pigeon hole clauses express the pigeon hole principle (Ramseys theorem) — if $n+1$ pigeons are placed in n holes, then some hole will have at least two pigeons. Stephen Cook proposed that for any given n, the clause form of the pigeon hole principle is a hard problem for automated reasoning systems in his classic paper [Cook, 1971]. We compared MDFA approach with FC and Hyperresolution on pigeon hole problem for different values of n and found that as n increases, the MDFA approach is much superior in terms of time as well as number of states generated. Table 1 summarizes our experimental results. Hyperresolution is unable to solve any of the problems in the table in a time limit of 1 hour and is hence not listed. On the n-queens problem[Nadel, 1988], FC was found to be better than MDFAs.

Conclusions and Future Research

We presented a canonical representation for CSPs using MDFAs and discussed algorithms for manipulating CSPs with it. Well known theoretical results on regular languages can be used to solve the finite domain constraint satisfaction problem by viewing it as a problem on regular languages. The key property exploited is the fact that the domains $\mathcal{V}$ and $\mathcal{D}$ being dealt with are finite. Our representation, being compact and canonical has a number of advantages. It enables us to construct implicit constraints using many different types of operators. Being a canonical form of solution sets, MDFAs serve as a representation tool as well as a reasoning tool. They allow for incremental construction and manipulation of constraints. All these results indicate that the MDFA approach is suitable for constructing knowledge bases for storing constraints and reasoning with them. The property of minimality guarantees a compact representation and the property of being canonical eliminates duplicate copies. Since an MDFA compactly represents the set of satisfying tuples, it allows fast query processing. Also, costly recomputation and backtracking are eliminated for repeated queries.

A number of open problems arise in the context of our discussion: (i) On what type of problems is the bottom up MDFA construction more effective than the classical top down backtrack search algorithms like forward checking and constraint propagation algorithms like k-consistency? On what type of problems are they worse? (ii) What is the order in which a network of constraints is best solved? When a series of constraints need to be composed, they can be composed in any order. Even though the final MDFA is the same, the sizes of intermediate MDFAs can be different for different orderings.

The discussion of this paper assumes that the number of variables and the number of values is fixed; it also treats all constraints as n-ary constraints. The proof of the canonical nature of MDFAs in [Hopcroft and Ullman, 1979] suggests that we can relax these assumptions and devise more compact canonical representations for constraints. We propose to investigate this further in our future work.

Acknowledgements:- The author would like to thank the following people for discussions and support, at various stages of this work: Ed Clarke and Randy Bryant of CMU, Vipin Kumar of the University of Minnesota, Bob Boyer of the University of Texas at Austin and Carl Pixley of MCC.

References

Akers, S. B. 1982. Functional testing with binary decision diagrams. In *Proceedings of the 8th Annual IEEE Conference on Fault-Tolerant Computing.* 75–82.

Bryant, Randall E. 1986. Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computers* C-35(8):667–691.

Clarke, Edmund M.; Kimura, S.; Long, David E.; Michaylov, Spirov; Schwab, Stephen A.; and Vidal, J. P. 1991. Parallel symbolic computation on a shared memory multiprocessor. In *Proceedings of the International Symposium on Shared Memory Multiprocessors.*

Cook, Stephen A. 1971. On the complexity of theorem proving procedures. In *Proc. of the Third Annual ACM Symposium on Theory of Computing.* 151–158.

Dechter, Rina and Dechter, Avi 1988. Belief maintenance in dynamic constraint networks. In *Proceedings of AAAI-88.*

Dechter, Rina and Meiri, Itay 1989. Experimental evaluaton of preprocessing techniques in constraint satisfaction problems. In *Proceedings of IJCAI-89.* 271–277.

DeKleer, Johann 1989. A comparison of atms and csp techniques. In *Proceedings of IJCAI-89.* 290–296.

Freuder, E. 1988. Backtrack-free and backtrack-bounded search. In Kanal, Laveen and Kumar, Vipin, editors 1988, *Search in Artificial Intelligence.* Springer-Verlag, New York.

Grimson, W. E. L. 1990. The combinatorics of object recognition in cluttered environments using constrained search. *Artificial Intelligence* 44:121–165.

Hopcroft, John E. and Ullman, Jeffrey D. 1979. *Introduction to Automata Theory, Formal Languages and Computation.* Addison-Wesley.

Kimura, S. and Clarke, Edmund M. 1990. A parallel algorithm for constructing binary decision diagrams. In *Proceedings of the International Symposium on Computer Design(ICCD).*

Kumar, Vipin 1992. Algorithms for constraint satisfaction problems: A survey. *AI Magazine* 13(1):32–44.

Mackworth, A. K. 1977. Consistency in networks of relations. *Artificial Intelligence* 8 (1):99–118.

Mittal, S. and Falkenheiner, B. 1990. Dynamic constraint satisfaction problems. In *Proceeding of AAAI-90.* 25–32.

Montanari, Ugo 1974. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science* 7,95-132.

Nadel, Bernard 1988. Constraint satisfaction algorithms. In Kanal, Laveen and Kumar, Vipin, editors 1988, *Search in Artificial Intelligence.* Springer-Verlag, New York.

Purdom, P.; Brown, C.; and Robertson, E. L. 1981. Backtracking with multi-level dynamic search rearrangement. *Acta Informatica* 15:99–113.

Rivin, Igor and Zabih, Ramin 1989. An algebraic approach to constraint satisfaction problems. In *Proceedings of IJCAI-89.* 284–289.

Rossi, Francesca and Monatanari, Ugo 1990. Exact solution of networks of constraints using perfect relaxation. In *Proceedings of First International Conference on Knowledge Representation.*

Stallman, R. and Sussman, G.J. 1977. Forward reasoning and dependency directed backtracking. *Artificial Intelligence* 9 (2):135–196.

Waltz, D. 1975. Understanding line drawings of scenes with shadows. In Winston, P. H., editor 1975, *The Psychology of Computer Vision.* McGraw Hill, Cambridge, MA.

Zabih, Ramin and McAllester, David 1988. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the 1988 National Conference on Artificial Intelligence.* 155–160.

Num. of Pigeons	FC Time in secs	MDFA Time in secs	Num. Nodes visited by FC	No. of MDFA states
6	0	2	3093	286
7	1	6	27271	2064
8	13	22	279467	4458
9	159	89	3272499	11856
10	2184	276	43095315	41708
11	*> 1hr*	906	NA	104221

Table 1: Experimental results on pigeon hole example.

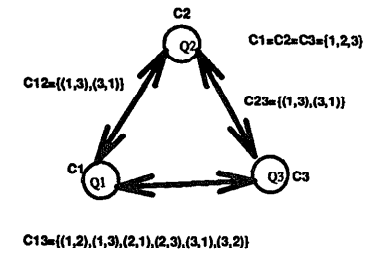

Fig 1. The 3-Queens Problem expressed as a CSP.

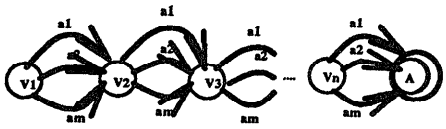

Fig 2. Example of an MDFA; this one accepts all n-tuples of values.

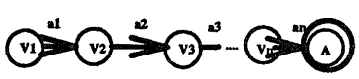

Fig 3. An MDFA representing the constraint with the single tuple (a1 ,a2, ...,an).

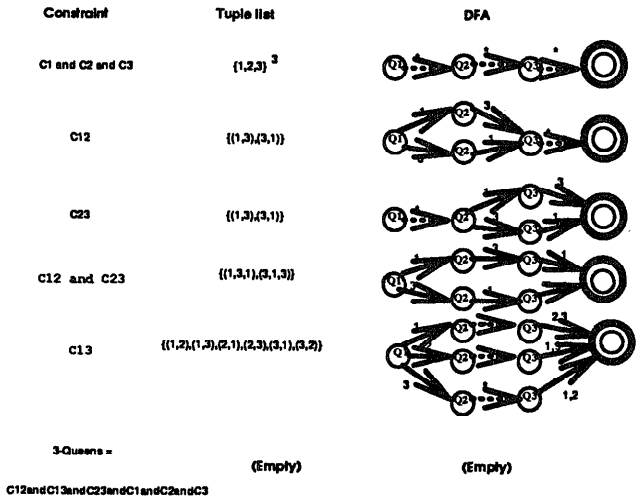

Fig 4. Solving the 3-queens CSP using MDFAs.

** denotes a wild card meaning any.*