AN OBJECT-ORIENTED CONSTRAINT SATISFACTION SYSTEM APPLIED TO MUSIC COMPOSITION

by

Russell David Ovans

B. Sc. University of Victoria 1988

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
August 1990

APPROVAL

Name: Russell David Ovans

Degree: Master of Science

Title of thesis: An Object-Oriented Constraint Satisfaction System Applied to Music Composition

Examining Committee:

Dr. Veronica Dahl
Chair

Dr. Nick Cercone
Senior Supervisor

Dr. Robert Hadley
Supervisor

Dr. Bill Havens
Supervisor

Dr. Hans Müller
External Examiner

Date Approved: August 7, 1990

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

An Object-Oriented Constraint Satisfaction System Applied to Music Composition.

Author:

(signature)

Russell David Ovans

(name)

August 9, 1990

(date)

Abstract

Constraint satisfaction problems (CSPs) are important and ubiquitous in artificial intelligence and generally require some form of backtracking search to solve. This thesis provides a methodology for solving CSPs using object-oriented programming and describes how music composition can be formulated as a CSP. Firstly, we have built, using an object-oriented programming language, a constraint satisfaction system that is applicable to any CSP. By defining a methodology for the conversion of an abstract model of a CSP, namely the constraint graph, into a network of co-operating objects, we have isolated some useful abstractions of constraint programming. This system can solve CSPs involving constraints of any arity, and frees the programmer from the details of tree-search and constraint propagation.

The second contribution of this thesis is an observation that music composition can be formulated strictly as a CSP. Recent attempts to develop expert systems for music composition have centred mainly around the rule-based approach, which we argue is frequently inefficient due to its reliance on chronological backtracking as the control method. Conversely, when music composition is viewed as a CSP, the complexity of the problem, the usefulness of the musical constraints, and the relationships among notes all become manifest in the form of a constraint graph. Most importantly, consistency techniques can be exploited in an effort to reduce backtracking and thus provide a more efficient procedure for the generation of compositions.

The synthesis is the generation of contrapuntal music by modeling first species counterpoint as a CSP and implementing its solution in our constraint satisfaction system. Using this approach, we have undertaken an analysis of the rules of first species counterpoint by

measuring their individual effect on constraining the number of compositions belonging to the genre.

Acknowledgements

For their time, insight, and encouragement I wish to thank Nick Cercone, Bill Havens, Bob Hadley, Hausi Müller, Dan Fass, Sanjeev Mahajan, Gary Hall, and Keith Hamel. For their financial support, I thank the Natural Sciences and Engineering Research Council of Canada.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The discovery that constraint satisfaction problems (CSPs) occur in artificial intelligence was a result of early work in general problem solving and computational vision. Informally, the problem (which usually requires a backtracking search to solve) is to assign values to a set of variables, each of which range over a finite domain, so that a set of constraints are simultaneously satisfied. Exploiting the effect the constraints have on limiting the search space can result in significantly improved search efficiency, which has fueled continued research into how a better exploitation of problem constraints can lead to more efficient algorithms for their solution.

Exploring constraints and constraint propagation is a good idea. In everyday life humans quite effortlessly and effectively exploit the natural constraints of their environment, employing them as a tool of intelligence. Or as Patrick Winston has said:

> For us to make a computer do something, it is often necessary to understand something about the world's constraints and regularities. These constraints and regularities make it possible for individuals to be intelligent, be they computer or human [Winston84, p. 58].

As such, artificial intelligence stands to benefit from a better understanding of constraints and their propagation, particularly in their role as search-space reducers.

1.1 The Problem

Two questions relating to CSP research are explored in this thesis. *Can we build, using an existing object-oriented programming language, a constraint satisfaction system that is applicable to any CSP?* Specifically, can a transformation be found that converts an abstract model of a CSP, namely the constraint graph, into the object-oriented paradigm of classes, objects, and messages? Instead of creating yet another programming language, we propose to isolate the useful abstractions of constraint programming and implement them in an object-oriented programming language. Hopefully this methodology is as beneficial as existing constraint programming languages where the control of constraint propagation is separate from the declaration of the constraints. This declarative feature relieves the programmer from the details of tree search and constraint propagation, providing a reusable framework for solving CSPs. For a general-purpose CSP-solving system to be truly useful, it should be applicable to not only unary and binary constraints (respectively, constraints involving one and two variables), but also to constraints of greater arity.

A suitable application for testing such a system is any toy CSP, like 8-queens: place eight queens on a chess board so that no two are positioned to attack each other. However, since the 8-queens problem can be formulated as a complete binary CSP where the same binary constraint is applied to every pair of variables, its solution as a CSP would tell us little of the methodology's overall strengths and weaknesses.

Alternatively, music composition provides a richer domain for testing constraint satisfaction systems. Non-trivial problems in music composition involve complex constraint relationships. An example is counterpoint: the simultaneous combination of several independent voices (melodic lines) into a coherent whole, the art of which flourished at the beginning of the 17th century. The rules of counterpoint, which limit allowable note combinations, act as the constraints whereas the notes are the variables to be assigned values.

Thus, counterpoint as CSP is useful for testing the effectiveness of the object-oriented constraint satisfaction system described in this thesis.

Examining music composition within the context of CSPs raises the second question: *What are the benefits of viewing music composition strictly as a constraint satisfaction problem?* Investigation of this question should provide insight on two fronts: the complexity of automated composition in so far as the tractability of the problem such systems attempt to solve; and, the syntax of a given compositional style by individual analysis of the effects of each rule on reducing the number of acceptable compositions.

1.2 Thesis Overview

Empirical answers to these two questions are provided in this thesis. In Chapter 2, the topic of constraint satisfaction problems is introduced with emphasis on their representation, algorithms for their solution, and their potential attractiveness as a programming paradigm.

In the third chapter a method for transforming a constraint graph into an object-oriented program is outlined. Since the constraint graph is a well understood knowledge representation formalism for CSPs, this means that any CSP can be transformed to an equivalent object-oriented representation where it can thus be solved by an object-oriented programming language. The underlying implementation of tree search and constraint propagation is self-contained in a small set of classes, thus providing a foundation for repeated and abstract use.

The application of this object-oriented constraint propagation system to counterpoint, which reveals the usefulness of the methodology as well as the appropriateness of viewing music composition strictly as a CSP, is described in Chapter 4.

In the fifth chapter the results are discussed and evaluated. Chapter 6 contains a summarization of what was achieved and details of future extensions.

Chapter 2

Constraint Satisfaction Problems

A ubiquitous problem class in computer science is that of the constraint satisfaction problem (CSP), also known as the consistent labeling problem. The task can be succinctly stated as follows: given a finite set of variables $X = \{x_1, x_2, \ldots, x_n\}$ whose elements range respectively over the finite (and not necessarily numeric) domains $D_1, D_2, \ldots, D_n$, assign a value to each variable such that the finite set of constraints $C = \{c_1, c_2, \ldots, c_m\}$ is satisfied. The role of the constraints is to reduce the cartesian solution space $D = D_1 \times D_2 \times \ldots \times D_n$. Each $c_i, 1 < i < m$, is a relation on a subset of X that states which values are consistent with each other. CSPs are solved by finding all points in the finite discrete space D that simultaneously satisfy the constraints. Sometimes a single solution is all that is required.

As an example, consider the following toy problem. Instantiate the set of variables $X = \{x_1, x_2, x_3, x_4\}$ whose elements range over the domain $\{1, 2, 3\}$. Let C, the set of constraints, require that: $x_2 < x_1$, $x_4 < x_3$, $x_1 \neq x_3$, and $x_2 \neq x_4$. This CSP has two solutions:

$$x_1 = 2, x_2 = 1, x_3 = 3, x_4 = 2$$

$$x_1 = 3, x_2 = 2, x_3 = 2, x_4 = 1$$

CSPs occur both within artificial intelligence (AI) and operations research. Scene labeling [Waltz75], truth maintenance [Doyle79][1], and logic puzzles [VanHentenryck89] are some of the applications in AI that require solving a CSP. Expert systems have been shown to benefit from an incorporation of constraint-based reasoning (for example, [Hayes-Roth86]). The relationship of CSPs to operations research is a result of the observation that graph colouring is a CSP. Since graph colouring arises in connection with many of the scheduling and partitioning problems found in operations research [Garey79], the class of CSPs as a whole is of interest to the field.

This class of problems has received significant attention and a large body of knowledge has accumulated. A complete overview of the topic of CSPs is beyond the limits of this thesis and is indeed unnecessary given the goals of this research, but a good survey is given in [Mackworth87]. As well, the topic of numeric constraints over both continuous and discrete domains, and their solution via linear programming, simplex algorithms, etc. for continuous domains and integer programming for discrete domains, is ignored to concentrate on symbolic constraints over a discrete space. The methodological advantages of the symbolic model to the object-oriented programming paradigm are shown. More importantly, not every CSP of interest can be easily stated as a set of equations, and in the case of continuous domains, a set of equations is unsuitable for the inference methods to be introduced.

Because of the inherent finiteness of CSPs, the problems are decidable. A backtracking [Golomb65] procedure can be used to systematically instantiate the variables and check if the constraints are satisfied. Since the domains are finite, as is the number of variables and constraints, the procedure terminates. However, efficiency is the issue, not decidability. Graph colouring is known to be **NP**-complete [Garey79], thus a general purpose algorithm for the entire class of CSPs requires (in the worst case) exponential time (unless $\mathbf{P} = \mathbf{NP}$). In fact, despite the elimination of subspaces from the solution space D with each failed

[1] Recently de Kleer has shown that every CSP can be expressed as an assumption-based truth maintenance system problem, but not vice-versa [deKleer89]

instantiation, backtracking is exponential in the number of variables in both the worst case [Mackworth85] and average case.[2] Therefore, research has focused on reducing the time required for solving CSPs, mainly through the exploitation of the constraints and their effect of reducing the search space. A summary of some of the resulting algorithms appears later in this chapter.

2.1 Representing a Constraint Satisfaction Problem

A useful way of representing a CSP is as a network of constraints [Montanari74], or equivalently as a constraint graph [Mackworth77]. In a constraint graph, the nodes are the variables to be assigned values and are usually labeled with the name of the variable they represent. It is sometimes also convenient to explicitly state the corresponding variable's domain at each node in the graph. The arcs represent the constraints: adjacent nodes participate in the constraint denoted by the arc connecting them. Unary constraints are expressed with loops. To represent n-ary constraints, $n > 2$, hyperarcs (arcs that connect more than two nodes) are required. We use the term *constraint graph* to refer to any graph, hyper or normal, which represents a CSP.

The arcs are usually labeled with the name of the constraint to be satisfied. In the case of binary constraints, directed arcs are useful for denoting the position of the variables in the predicate that defines the constraint. An undirected arc denoting the binary constraint P is equivalently two directed arcs: the predicate P and its transpose P^T. (For expediency, Mackworth omits the implicit transpose arcs.) There is no need for more than one arc to connect adjacent nodes in a constraint graph since multiple constraints can always be conjoined together.

The constraint graph for a corresponding graph colouring problem is in fact the graph

[2]The worst case is the pathological condition of a CSP where no solution exists solely because the only inconsistent assignment is caused by the last variable instantiated. The full tree is explored before the procedure terminates. One can expect that in the average case, only half of the complete tree is explored, but this is still exponential in the number of variables. Rigorous proof is perhaps impossible given the dependence on specific problem constraints.

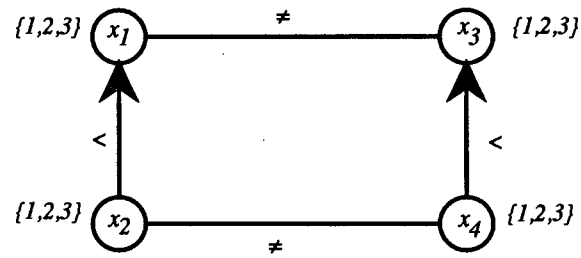


Figure 2.1: A constraint graph for the sample problem.

to be coloured. A constraint graph for the sample problem of the previous section is given in Figure 2.1.

The constraint graph is not only a useful representation device but as a mathematical model reveals limitations on the efficiency of searching for a solution to the problem it represents. For example, in [Freuder82] a sufficient condition for backtrack-free search is given as a function of the graph's connective structure.[3]

2.2 Algorithms for Solution

Backtracking is a general but often inefficient algorithm for solving CSPs. Attempts at improving the performance of CSP-solving algorithms led to the incorporation of consistency techniques. Consistency techniques prune the search space before failure occurs, thus improving the efficiency of tree search by reducing the number of backtrack points. Extra time is invested looking ahead at each node in the tree in the hope that costly failures can be prevented.

Even before any variable has been instantiated we can reason about the domains of the variables insofar as to eliminate values that we know can never participate in a solution to the problem. In our sample CSP we have that x_2 must be less than x_1. We can therefore

[3] A backtracking search is considered backtrack-free if an instantiation never needs to be undone as a result of failure at a lower-level in the tree.

eliminate 1 from D_1 (and 3 from D_2) since $x_1 \leftarrow 1$ (and $x_2 \leftarrow 3$) can never satisfy that constraint. Similarly, 1 and 3 can be removed from D_3 and D_4 respectively.

Once a variable has been instantiated, further reductions can occur. If $x_1 \leftarrow 2$, we can remove 2 from D_3 since the constraints require that x_3 have a different value than x_1. In a likewise fashion, it can be deduced that x_2 must take the value 1 in order that it satisfy the lower-than constraint with x_1. Since x_4 must be different from x_2 and lower-than x_3, it has to be 2. The problem has been solved, without backup, as a result of instantiating a single variable.

This is an illustration of pruning the search space by means of domain reduction; values from a variable's domain that cannot possibly participate in any solution to the problem are eliminated, thus avoiding a possibly costly failure later on in the search tree. This is the key idea behind the consistency techniques examined in this chapter.

2.2.1 Consistency Techniques

Consistency techniques are an alternative to dependency-directed backtracking [Stallman77], which aims to backtrack intelligently by re-instantiating the variable most likely to have been the cause of failure.[4] In discussing the difference between consistency techniques and backtracking, Van Hentenryck notes that "...dependency-directed backtracking...is more a remedy to a symptom of the malady and not to the malady itself. Indeed, it is better to prevent failures than to react intelligently to them" [VanHentenryck87].

Consistency techniques are weak inference methods: they always work, but will not always solve a CSP. They obtain their power through the propagation of constraints. This propagation is used in two different ways: to filter domains (as in [Waltz75]) or to generate new values from known ones through networks resembling electronic circuit diagrams [Sussman80]. Procedures of the former type are most often associated with symbolic domains

[4]Both intelligent and dependency-directed backtracking abandon a purely chronological method for variable reinstantiation.

```
procedure NC (i)
    D_i ← D_i ∩ {x | P_i(x)}
```

Figure 2.2: The node consistency procedure.

and are called *labeling procedures.* Those of the latter type are said to employ *value propagation.* In either case, global consistency is achieved through local computation [Winston84]; the difference between the two methods is in the control of inference. This thesis adopts the labeling procedure approach, particularly because of its intuitive analogue to the constraint graph representation. A potentially useful distinction in the CSP literature would be to reserve the term constraint *network* for a structure that embodies the value propagation of numeric equations. Conversely, the symbolic relaxation of constraints by means of a labeling procedure is best represented with a constraint *graph.*

The fundamental consistency technique is to achieve node consistency [Mackworth77], which essentially ensures that each node in the constraint graph meets any unary constraint that may be acting upon it. (The node's initial domain can be seen as a unary constraint.) This consistency is attained through a procedure (NC) that filters out any inconsistent values in the node's domain (Figure 2.2). Constraints are represented by a predicate that is true if the constraint is satisfied.

The next level of consistency is to achieve arc consistency ([Fikes70, Waltz75], but formalized in [Mackworth77]), named so because the idea is to ensure that connected nodes in the constraint graph are consistent with each other. This consistency is attained by a filtering function Mackworth calls REVISE. In the case of binary constraints, the function takes two nodes (i and j) as parameters. The domain D_i is filtered such that each of its members is supported (i.e., can satisfy the binary predicate P_{ij}) by at least one member of D_j.

The version of REVISE in Figure 2.3 is designed for the enforcement of binary constraints only. Different versions are required for constraints of greater arity. Without loss

```
function REVISE (i, j): boolean
   change ← FALSE
   ∀x ∈ D_i do
   if ¬∃y ∈ D_j such that P_ij(x, y)
   then begin
           remove x from D_i
           change ← TRUE
        end
   return (change)
```

Figure 2.3: The REVISE procedure.

of generality, the remainder of this chapter considers constraint graphs solely comprised of unary and binary constraints.

2.2.2 Arc Consistency Algorithms

A constraint graph is said to be arc consistent if and only if all adjacent nodes in the graph are consistent with each other. A graph becomes arc consistent when it is subjected to an arc consistency algorithm, which in any of its forms essentially controls the propagation of constraints by placing an ordering on calls to REVISE. In [Mackworth77], three different arc consistency algorithms are discussed: AC1, AC2, and AC3.

The simplest of the arc consistency algorithms is AC1 [Mackworth77] given in Figure 2.4. After ensuring node consistency, the procedure iteratively checks every binary constraint (with a call to REVISE) until quiescence.

The efficiency of AC1 has been analyzed in [Mackworth85]. It is shown to be $O(a^3ne)$ where a is the size of the domains (assuming each D_i is the same size), n is the number of variables, and e is the number of edges in the constraint graph G.

```
procedure AC1 (G: constraint graph)
   for i ← 1...n do NC(i);
   Q ← {(i,j) | (i,j) ∈ arcs(G), i ≠ j }
   repeat
       change ← FALSE
       ∀(i,j)∈ Q
            change ← REVISE(i,j) ∨ change
   until ¬change
```

Figure 2.4: Procedure AC1.

Clearly AC1 can be made more efficient.[5] The *repeat* loop in AC1 unnecessarily reconsiders every arc in G. If REVISE(i,j) returns *true* (i.e., D_i is reduced), it is only necessary to reconsider those arcs that lead to node x_i (with the notable exception of (j,i)), for it is only those arcs that can be made inconsistent as a result of the change to D_i. This is the insight behind AC3, an improved arc consistency algorithm given in Figure 2.5.

It is not immediately intuitive why the arc (j,i) does not need to be considered if REVISE(i,j) is *true*. Nadel claims we do not add (j,i) to Q because it is either already there or it is already consistent [Nadel88]. To see that this is true, consider a constraint graph with two variables, A and B with respective domains D_A and D_B, connected by the binary constraint P. In AC3, Q is initialized to $\{(A,B),(B,A)\}$. Assume that arc (A,B) is removed from Q first. After the call to REVISE(A,B) the following holds:

$$D'_A = \{a \in D_A \mid \exists b \in D_B, P(a,b)\}$$

$$(\forall a \in D'_A)(\exists b \in D_B)(P(a,b))$$

Adding (B,A) to Q is unnecessary since it is already there. Next, a call to REVISE(B,A) is made because (B,A) is the remaining element in Q. After this call the following holds:

$$D'_B = \{b \in D_B \mid \exists a \in D'_A, P(a,b)\}$$

[5] Both AC2 and AC3 are more efficient algorithms when implemented on a single processor, but AC1 has more inherent parallelism [Mackworth87].

```
procedure AC3 (G: constraint graph)
   for i ← 1...n do NC(i);
   Q ← {(i,j) | (i,j) ∈ arcs(G), i ≠ j}
   while Q ≠ ∅
   begin
       select and delete any arc (k,m) ∈ Q
       if REVISE(k,m)
       then Q ← Q ∪ {(i,k) | (i,k) ∈ arcs(G), i ≠ k, i ≠ m}
   end
```

Figure 2.5: Procedure AC3.

$$(\forall b \in D'_B)(\exists a \in D'_A)(P(a,b))$$

AC3 does not add arc (A,B) to Q and the procedure terminates. It would be necessary to reconsider arc (A,B) only if

$$(\exists a \in D'_A)(\forall b \in D'_B)(\neg P(a,b))$$

That is, if D'_B does not support each member of D'_A. This could only happen if an element of D'_A has its support solely in $D_B - D'_B$. However, this is not possible since

$$D_B - D'_B = \{b \in D_B \mid \forall a \in D'_A, \neg P(a,b)\}$$

That is, the elements discarded from D_B were precisely those that supported *not one* element of D'_A. Therefore (A,B) is not added to Q because it is already consistent.

In AC3, Q is treated as a set, but in practise it is either a queue or a stack: new elements (the arcs to reconsider) are either added to the rear or to the front. In certain problems it may be more appropriate to design Q one way rather than another. The question of which order to re-visit the arcs "...is the analog for AC3 of the constraint check order issue that arises for most, if not all, constraint satisfaction algorithms"[Nadel88, p. 325].

AC3 is shown to be $O(a^3e)$ in [Mackworth85]; it is linear in the number of edges (i.e., the number of binary constraints).

```
procedure AC2 (G: constraint graph)
   for i ← 1...n do
   begin
       NC(i)
       Q ← {(i,j) | (i,j) ∈ arcs(G), j < i}
       Q' ← {(j,i) | (j,i) ∈ arcs(G), j < i}
       while Q ≠ ∅
       begin
          while Q ≠ ∅
          begin
             pop(Q,(k,m))
             if REVISE(k,m)
             then Q' ← Q' ∪ {(p,k) | (p,k) ∈ arcs(G),p ≤ i,p ≠ m}
          end
       Q ← Q'
       Q' ← ∅
       end
   end
```

Figure 2.6: Procedure AC2.

The final arc consistency algorithm introduced by Mackworth is AC2 (figure 2.6), an equivalent algorithm to Waltz's procedure [Waltz75] and a special case of AC3. AC2 makes one pass through the nodes of the graph, introducing each new node as it is reached in the iteration. A node is introduced by first making it node consistent and then considering each arc involving it and any previously introduced nodes.

2.2.3 Beyond Arc Consistency

Waltz's procedure is the embodiment of a school of thought that Nadel calls *arc consistency primary* [Nadel88]. The belief of this school is that attaining arc consistency is sufficient for solving a CSP, and indeed for certain problems (e.g., scene labeling, the application Waltz's procedure was designed for) this is often the case. However, arc consistency has two serious shortcomings. If a constraint graph is arc consistent this does not necessarily imply

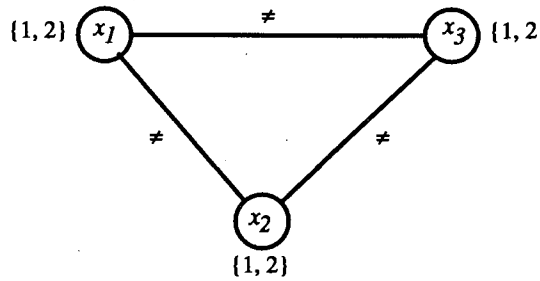


Figure 2.7: A constraint graph that is both arc consistent and unsatisfiable.

a search to solve it is backtrack-free. Arc consistency only implies backtrack-free search if the constraint graph is in fact a tree [Freuder82]. Thus arc consistency alone does not guarantee the eradication of exponential time requirements.[6]

A second deficiency is that a constraint graph can be both arc consistent and unsatisfiable. For example, given certain scenes as input, Waltz's procedure terminates without a unique solution (that is, some nodes have more than one interpretation), yet the figure has no consistent labeling [Freuder78]. Figure 2.7 is a simple constraint graph that is arc consistent but has no solution.

Freuder explains that these deficiencies in arc consistency are a result of its inability to satisfy the global constraint of the problem. Arc consistency may remove all local inconsistencies, but it is ineffectual when faced with incompatibilities that arise in paths through the graph. Path inconsistency [Montanari74] is the following problem as illustrated by Figure 2.7. Nodes x_1 and x_3 are arc consistent, allowing two pairs of values: $(1,2)$ and $(2,1)$. However, neither pair of values is allowed by the indirect path from x_1 to x_3. Each pair of nodes in Figure 2.7 is similarly path inconsistent.

Algorithms to ensure path consistency were first developed in [Montanari74, Mackworth77].

[6] If it did, we would have a polynomial time algorithm for the solution of the general class of CSPs, which includes graph colouring, and thus **P** would equal **NP**.

These polynomial-time procedures require a constraint representation that defines intersection and composition operators (e.g., a matrix). Procedural representations, in the form of Boolean functions, are therefore unsuitable.

Path consistency is generalized in [Freuder78] to be a special case of k-consistency, defined as follows. A graph is k-consistent if and only if every instantiation of any $k-1$ variables that satisfies all the applicable constraints amongst those $k-1$ variables also permits the instantiation of any kth remaining variable such that the k variables together are mutually consistent. Given this definition, node-, arc-, and path-consistency are k-consistency for $k = 1, 2, 3$ respectively.[7]

Freuder submits an algorithm for obtaining k-consistency in [Freuder78], but its usefulness is dubious given its inefficiency: the algorithm is exponential in k [Mackworth87]. Freuder later defined j-consistency for all $j \leq k$ to be *strong* k-consistency [Freuder82]. All of the ACi algorithms are strong 2-consistency algorithms.

2.2.4 Tree Search

In contrast to the arc consistency school is the *tree search primary* school [Nadel88], where it is believed that CSPs need good tree searching more than they need arc consistency. The claim is that for the majority of CSPs, consistency techniques alone will not settle on a single consistent solution (although one could obtain k-consistency with $k = n$, a $O(n^n)$ operation).

Nadel's dichotomy is an interesting one and is exemplified by differences in vertex labeling a line drawing (Waltz's problem) and the 8-queens problem. Line drawings are so highly constrained that propagating these constraints often results in a single consistent solution without resorting to tree search [Waltz75]. In other words, arc consistency is usually sufficienct for the consistent labeling of junctions in a line drawing. Conversely, the constraints in the 8-queens problem are much looser and permit 92 different solutions. In this case, tree

[7] Assuming we restrict our definition of path-consistency, as most do, to paths of length two.

search is necessary to solve the problem.

Tree search is an important method for solving underconstrained CSPs (i.e., CSPs with more than one solution). A unified survey of many of the labeling procedure algorithms in terms of tree search and arc consistency is given in [Nadel88]. Nadel's paper combines the work of Mackworth [Mackworth77] and Haralick and Elliot [Haralick80] into a single class of algorithms that incorporate tree search with varying degrees of arc consistency, espousing that "...in a search for greater efficiency, quite a few algorithms arose by development of successively less complete forms of full arc consistency algorithms coupled with tree search" [Nadel88, p. 297]. The following discussion adopts Nadel's nomenclature. At level k of a tree search, variables $x_1 \ldots x_{k-1}$ have been instantiated, x_k is the current variable to be instantiated, and $x_{k+1} \ldots x_n$, the uninstantiated variables, are referred to as *future* variables.

Even the potentially exponential backtrack search can be seen to employ a measure of arc consistency. At level k in the search tree, variable x_k is instantiated to a value from domain D_k. However, D_k is first filtered against all past instantiations of variables $x_1 \ldots x_{k-1}$. In this sense, filtered means that the variable x_k is checked against every constraint that involves it and the variables $x_1 \ldots x_{k-1}$. However, these variables $(x_1 \ldots x_{k-1})$ are all instantiated: their domains are of size one. This can result in a drastically reduced search tree compared to simple depth-first search (see for example Knuth's exploration of the *instant insanity* problem and his considerations (symmetry, etc.) that lead to the dramatic reduction in the search tree [Knuth74]).

Forward Checking [Haralick80] improves upon backtracking by not only filtering D_k, but also filtering all future variables' domains $(D_{k+1} \ldots D_n)$. However, this filtering of $D_k \ldots D_n$ is only done with respect to the instantiated variable x_{k-1}. Forward Checking is more efficient than backtracking because it can detect dead-ends sooner; by checking all future variables it can find an empty set for a future domain and know not to continue with the current branch. However, Nadel cautions that "...unrepaid effort can be the downfall of many an otherwise 'intelligent' algorithm...", and "...it is the correct balance that is

so elusive and that really must be adjusted for each problem individually..." [Nadel88, p. 307].

Partial Lookahead [Haralick80] augments Forward Checking by filtering the current and future variables' domains with each other, not just the most recently instantiated variable's (singleton) domain. This filtering is done only with respect to "more future" variables' domains, i.e., variable x_j is filtered with respect to $x_{j+1} \ldots x_n$ for $k \leq j \leq n$ (thus *Partial* Lookahead). This algorithm must utilize some form of Mackworth's REVISE procedure since it filters domain against domain, both of which may be larger than size one. At shallow levels of the tree, the domains are large and supportive of each other, thus the extra consistency checks can be very expensive. An intelligent search would know at what level to turn Partial Lookahead on [Nadel88].

Full Lookahead [Haralick80] filters the current variable's domain and all future variables' domains against all other future variables' domains. However, only one pass through the variables is made, which means that some inconsistencies are possibly missed (cf. AC1, which iterates through the arcs until no change occurs).

The next level of lookahead would lead to an incorporation of full arc consistency, for which Nadel develops nine full arc consistency hybrid search algorithms [Nadel89]. The result of each procedure is that full arc consistency is maintained for the entire graph at each search tree node. The algorithms differ in the choice of arc consistency algorithm (AC1, 2, or 3) and the subgraph of G made arc consistent at each level k of the search tree. For example, the algorithm Nadel calls TSAC3 works as follows. At level k in the search tree, AC3 is applied to the subgraph of G induced by $\{x_{k-1}, \ldots, x_n\}$ and x_k is assigned one of its remaining values from D_k. In order to facilitate backtracking, the domains $D_k \ldots D_n$ must be saved before applying the arc consistency algorithm.

In summary, Nadel prefers avoiding full arc consistency, and his test results warrant this claim. Forward Checking is the clear winner in efficiency (measured by the number of constraint checks performed) when it comes to finding all solutions to the n-queens problem,

$n \leq 10$. For some problems, "...it pays not to apply an arc consistency algorithm until some variable has been instantiated" [Nadel88, p. 335]. This intuitively makes sense for underconstrained applications where the original domains are highly supportive of each other. However, it should be noted that Forward Checking produces a larger search tree than any full arc consistency search procedure.

2.3 The Attractiveness of Constraint Satisfaction

2.3.1 Constraints as Knowledge Representation

In the general sense, constraints are rules about acceptable and/or probable states of the world, like the fact that traffic lights at an intersection don't conflict. Constraints aid in the communication of natural language, in understanding what we see, in planning, and in solving problems. In short, they help us act intelligently by restricting the number of interpretations we apply to the world in any given situation. Within the domain of CSPs, a constraint is more rigidly defined as a relation on a non-empty subset of the problem variables. Still, this can have a very wide interpretation (see [Freuder78] for some useful synonyms).

Within artificial intelligence, constraints are evolving as their own useful knowledge representation formalism. Constraints have been included in ambitious projects like the expert system shell KEE [Filman88] and the hybrid knowledge representation system Babylon [Guesgen87]. In KEE, a constraint is a deduction rule whose consequent is always false, thus rendering an inferred world inconsistent. In Babylon, constraints are used in two ways: as their own knowledge representation formalism, and as silent watch-dogs over other formalisms. In both systems, like in the real world, the constraints function to reduce a search space.

CSPs contain three declarative entities: variables, domains, and constraints. The manner in which each of these is represented, particularly the constraints, is not important. The

method of inference, the *control* of these entities, is the important component. Stating a problem as a CSP enables the utilization of consistency techniques as an efficient method of inference. This is how the CSP approach to problem modeling can be advantageous compared to other knowledge representation schemes and their corresponding inference procedures.

2.3.2 Constraint Programming Languages

The emergence of constraints as a knowledge representation formalism and constraint propagation as a useful control mechanism in AI has lead to the recent introduction of constraint programming languages. Constraint programming is a declarative task that separates the control (method of constraint propagation) from the relationships amongst objects (the constraints) [Leler88]. Ideally the programmer only has to concern himself with the constraints and not the chore of tree-search and constraint propagation. This is the goal of many recent attempts to create constraint programming languages.

Constraint programming languages are distinct from their imperative counterparts in the following ways. Since they have a declarative semantics, the programmer concentrates on stating relationships (*what*), not on control (*how*) [Mackworth87]. Pure constraint programming languages are devoid of assignment, procedures, and control flow. Further to the absence of assignment, equality is a relational operator only. The control flow of a constraint language is managed by the *constraint satisfier*; the primary computation mechanism is some form of constraint propagation (usually value propagation).

Some examples of constraint programming languages are Bertrand [Leler88], CONSTRAINTS [Sussman80], and the constraint logic programming languages: CLP(R) [Cohen90], Prolog III [Colmerauer90], and CHIP [VanHentenryck89].[8] Of this group, CHIP is the most versatile and promising.

[8]It should be noted that Van Hentenryck considers CHIP to be the first declarative language based on consistency techniques, and dismisses Bertrand and CONSTRAINTS as something other than true constraint programming languages.

CHIP (an acronym for Constraint Handling in Prolog) is the result of recent attention to the idea of embedding consistency techniques in logic programming. It has been shown that every constraint graph can be converted to an equivalent logic program that elegantly states the corresponding CSP [Rossi88]. Logic programming is thus a convenient method for solving CSPs. However, since these problems are often best stated as *generate and test* programs and are solved by an underlying backtracking algorithm, the resulting programs are highly inefficient [VanHentenryck89]. Thus Van Hentenryck *et al* have worked on embedding consistency techniques in Prolog, and on replacing resolution with constraint-solving. The resulting paradigm is to *constrain and generate.*

Central to this goal is the incorporation of *domains.* Van Hentenryck claims that the distinguishing feature of CSPs is the finiteness of the domains [VanHentenryck89]. CHIP permits the clear expression of a variable's domain, and incorporates Forward Checking and Lookahead as inference rules. Van Hentenryck's results in many operations research applications are encouraging; a testament to the usefulness of consistency techniques and constraint propagation.

Chapter 3

An Object-Oriented Constraint Satisfaction System

An object-oriented programming (OOP) language is an appropriate implementation vehicle for constraint propagation systems: not only have some successful projects been completed using OOP, but it facilitates a natural expression of constraint propagation not available with other programming paradigms.

Borning utilized OOP in the design of ThingLab [Borning81], a constraint-oriented simulation laboratory written in the object-oriented language Smalltalk. Steels has proposed a new mechanism for constraint languages that is implementable in any frame-based object-oriented language [Steels85]. However, these systems have only employed value propagation as the control mechanism. Instead, our proposal is to have objects encapsulate the nodes of a constraint graph, and, by passing messages through the graph, enforce arc consistency amongst the nodes. The problem is thus to establish a methodology for the construction of such a constraint graph and a protocol for the communication among objects so that the arc consistency of the graph is maintained before and as variables become instantiated.

The main objective of this proposal is to provide a convenient and flexible methodology for the specification and solution of CSPs, but without burdening the programmer with tree search, constraint propagation, and learning a new programming language. Though

the first two objectives are also the goals of many constraint programming languages, these specialty languages often fail to provide any benefit to the computing public at large due to their lack of availability (e.g., Bertrand and CHIP), domain specificity (e.g., Bertrand and CONSTRAINTS can only represent numeric constraints), and/or their lack of acceptance by conventional programming shops (an obstacle CHIP, as a logic language, may never overcome). By examining how OOP can be used to implement a computation mechanism based on label inference, we provide the general framework for the creation of a declarative CSP-solving system.

The central idea of the methodology proposed in this chapter is that a constraint graph can be implemented as a set of inter-connected, co-operating objects that attain arc consistency amongst themselves by means of message passing. The end result is that a full arc consistency algorithm (a variant of AC3) is a by-product of this message passing. Once a small set of classes are defined, solving a different CSP is simply a matter of re-defining the object instances that comprise the graph and/or the constraints that it enforces. Because these reusable classes comprise a software-IC [Cox86] for the solution of constraint satisfaction problems, we have named this system *CSP-IC*.

The following discussion assumes prior knowledge of object-oriented programming and its nomenclature. Good overviews are found in [Cox86] and [Stefik86]. The ideas introduced are not particular to any one OOP language, however, the language used here is Objective-C ([Cox86], specifically [NeXT89]). A different implementation language would likely result in a slightly different class hierarchy and/or programming approach.

3.1 The Required Object Classes

Before giving precise details of the structure of the object-oriented constraint graph, it is first necessary to describe the object classes required for its implementation. The methodology requires the definition of four new subclasses of the root class *Object* (see Figure 3.1). The

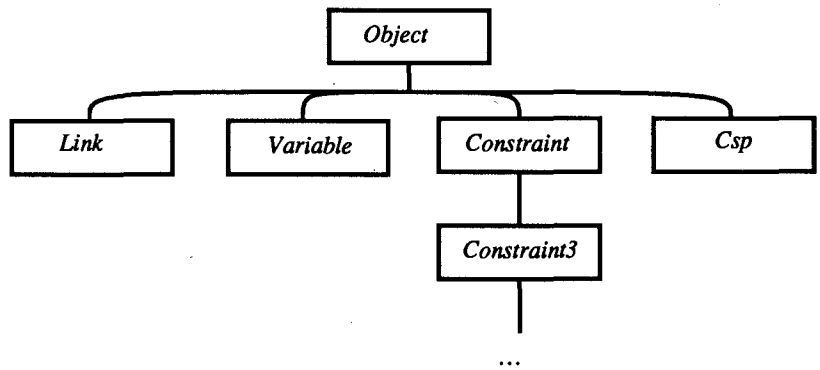


Figure 3.1: The hierarchy for the required classes.

```
Instance Variables:  id node
                     SEL label
Methods:             - getNode
                     - getLabel
                     - setNode:  anObject
                     - setLabel:  aMethod
```

Figure 3.2: The instance variables and methods of class Link.

relative flatness of this design can be in part attributed to the suggestion in [Taenzer89] that the construction approach to reuse is preferable to subclassing. Full implementation details for each class are found in Appendix A, but a general discussion of each follows.

3.1.1 The Class Link

The structure of the graph is represented by the class *Link* (Figure 3.2). A Link is a couple <*node, label*> and corresponds to a labeled, directed arc in a constraint graph. The object *node* is the vertex the arc is incident to, and *label* is the label on the arc.

```
Instance Variables:  List *domain
                     List *neighbours
                     STR name
Methods:             + new:  varName
                     - addToDomain:  anObject
                     - addToNeighbours:  aLink
                     - printDomain
                     - alertChanged
                     - instantiate:  anObject
```

Figure 3.3: The instance variables and some methods of class Variable.

3.1.2 The Class Variable

Objects of class *Variable*, which represent variables in a CSP and thus nodes in a constraint graph, have a *name*, a *domain*, and a collection of *neighbours* (Figure 3.3). In this respect, Variable objects have the same attributes as their counterparts in a CSP.

The domain is a list of objects.[1] The type of these domain objects is problem specific and thus defined by the user in a class separate from the four discussed here. More is said about this aspect of the design in Section 3.2.3. A Variable object is considered instantiated when its domain is of size one. If the domain ever becomes empty, a globally unsatisfiable instantiation has occurred somewhere in the graph. The objects in *domain* are referenced only and shared amongst other Variable objects: space for these objects is neither allocated nor freed by objects of this class.

A Variable object's position in a constraint graph is defined by its neighbours. The *neighbours* instance variable is a collection of objects of the class Link; there is one Link object for each of the arcs incident from the variable in the constraint graph. The number of elements in *neighbours* is thus equal to the out-degree of the vertex representing the variable. When a Variable object experiences a domain reduction, it notifies its neighbours in the

[1] In this implementation, all collections are of class *List*, but it could be an unordered type.

```
i = 0;
result = YES;
while (result && i < [neighbours count])
      {
      link = [neighbours objectAt:  i]; // get the ith neighbour
      result = [[link getNode] revise:self using:[link getLabel]];
      i++;
      }
```

Figure 3.4: Alerting change in a Variable object's domain.

graph so that they may check for arc consistency given the new domain. This notification is carried out by sending the message *revise:using:* to the object *node* for each instance of Link in the *neighbours* collection (see Figure 3.4).

The call to *revise:using:* returns *true* if this change has propagated safely, i.e., has not resulted in a variable's domain being reduced to size zero. The *revise:* argument is the object that has changed and the *using:* argument is the selector for a method to be used as the predicate with which to check consistency. The code fragment in Figure 3.4 is essentially the method *alertChanged.*

3.1.3 The Class Constraint

The recipients of the *revise:using:* messages are objects of class *Constraint* (or one of its descendant subclasses). The class Constraint embodies objects that are responsible for filtering the domains of objects of class Variable (Figure 3.5).

An object of class Constraint constrains an (one) object of class Variable; namely, the object defined as an instance variable in its class definition. In other words, it enforces and propagates binary constraints. When the Constraint object receives a *revise:using:* message, it checks the consistency (with regards to the constraint specified by the *using:* argument) of *variable* with that of the Variable object it received the message from. Because the

```
Instance Variables:  Variable *variable
Methods:             - setVariable:  aVariable
                     - revise:  aVariable using:  aMethod
```

Figure 3.5: The instance variables and methods of class Constraint.

constraint itself is a parameterized variable, Constraint objects can enforce *any* particular binary constraint.

Subclasses of Constraint add an additional object to the instance variables (corresponding to the additional Variable object to constrain) and change the implementation of the *revise:using:* method to match the arity of the constraints it enforces. For example, *Constraint3* defines objects that can enforce any ternary constraint. The exact number of necessary subclasses is problem dependant; specifically, upon the maximum arity of the occuring constraints. If binary constraints are all that appear in the CSP to be solved, then no subclasses of Constraint are required. Unary constraints are not explicitly encapsulated by a class. Instead, it is assumed that each Variable object's domain will be appropriately initialized.

If a Constraint object succeeds in reducing the domain of its *variable*, it must propagate these changes through the graph. It accomplishes this indirectly by sending the message *alertChanged* to the Variable object it constrains. If this propagation is unsuccessful *alertChanged* returns *false* and the Variable object's domain is reinstated to its original unfiltered state.

3.1.4 The Class Csp

Class *Csp* is simply a list of objects of class Variable, and represents a specific CSP (Figure 3.6). Since Variable objects define their position in a graph with their instance variable *neighbours*, a Csp object thus encapsulates an entire CSP's constraint graph. In this case,

```
Instance Variables:  List *variables
Methods:             - addToVariables:  aVariable
                     - findSolutions
                     - findASolution
                     - makeArcConsistent
```

Figure 3.6: The instance variables and methods of class Csp.

the distinction between CSP and constraint graph has been blurred completely: a CSP *is* a constraint graph, and vice-versa. This class provides methods for making the graph arc consistent (*makeArcConsistent*) and finding one (*findASolution*) or all solutions to the represented problem (*findSolutions*).

3.2 Converting a Constraint Graph to the Defined Object Classes

A transformation can be performed on an arbitrary constraint graph that results in a new graph realizable in the classes defined in the previous section. This new graph is called the *modified constraint graph* (MCG).

3.2.1 The Modified Constraint Graph

As an example, this transformation is applied to the simple constraint graph given in Figure 3.7. For the sake of clarity, this first example involves binary constraints only. The conversion comprises four steps.

1. Remove all loops representing unary constraints.

2. Make all implicit transpose arcs explicit by converting each undirected arc to two directed, labeled arcs as in Figure 3.8.

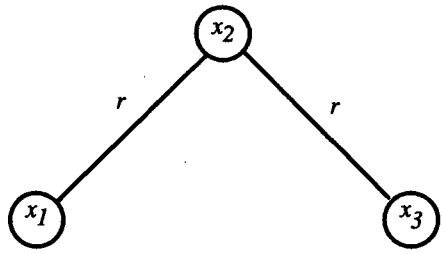


Figure 3.7: The binary constraint graph to be transformed.

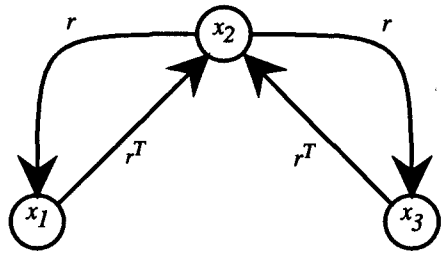


Figure 3.8: All undirected arcs become two directed arcs.

3. Instantiate each constraint as an unlabeled node and insert it between its adjacent nodes while maintaining the directionality of the arc it splits (Figure 3.9). It is only necessary to maintain the label on the arcs incident to the new unlabeled node.

4. Combine those unlabeled nodes whose incident (labeled) nodes are the same. Figure 3.10 shows the completed graph for this example. This merging of unlabeled nodes is also done for distinct constraints; that is, even when the arcs incident to an unlabeled node have different labels. Figure 3.11 is an example of how three nodes can be merged into one because they share a common incident node.

We now repeat the steps on a constraint graph comprised of three nodes and one ternary constraint (Figure 3.12). In general, step 2 causes the conversion of a hyperarc representing an n-ary constraint into n directed hyperarcs (Figure 3.13). This is perhaps a slightly unnatural representation for a hyperarc, but is necessary to maintain the semantics of the

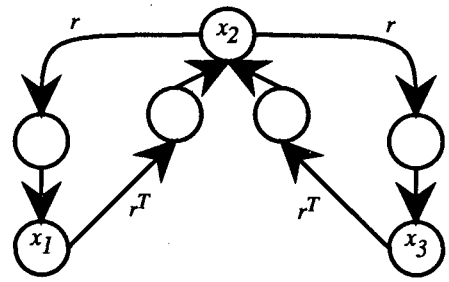


Figure 3.9: The constraints instantiated as nodes.

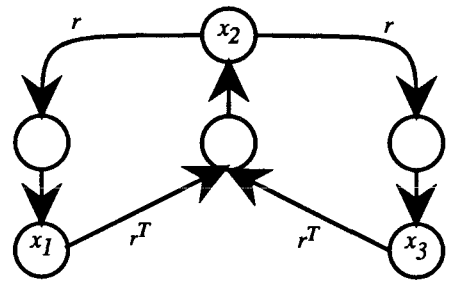


Figure 3.10: The completed modified constraint graph.

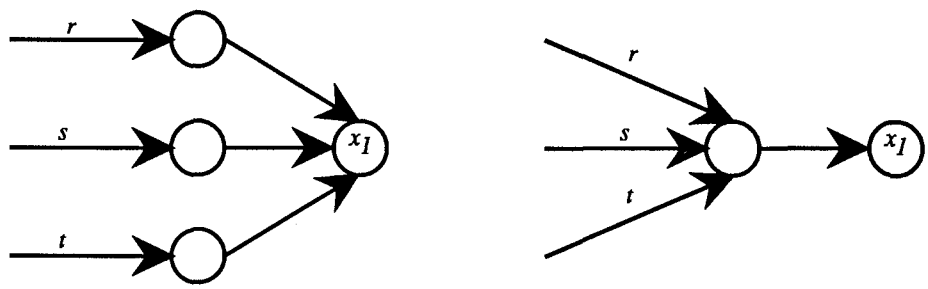


Figure 3.11: Merging three constraint nodes into one.

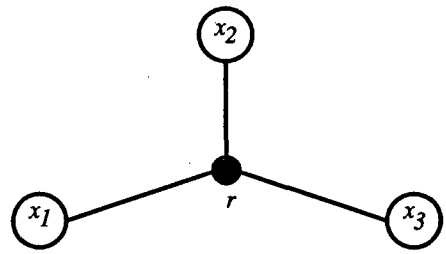


Figure 3.12: A constraint graph with a ternary constraint.

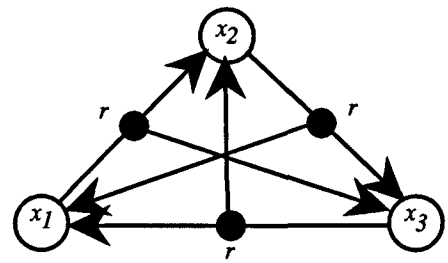


Figure 3.13: Hyperarcs become directed hyperarcs.

original constraint hypergraph.

When the constraints are instantiated as nodes, each hyperarc is split so that the new node has out-degree $n-1$ and in-degree 1, where n is the arity of the constraint the hyperarc represents (Figure 3.14).

The merging of n-ary constraint nodes is somewhat complex: all unlabeled nodes whose $n-1$ incident-from arcs connect the same $n-1$ labeled nodes should be merged into one. Since none of the constraint nodes in Figure 3.14 have common arc destinations, no reduction is possible. Figure 3.14 is thus the completed MCG for this example. Figure 3.15 illustrates an instance where a merging of ternary constraint nodes is possible.

3.2.2 From Modified Constraint Graph to Objects

Once the constraint graph has been transformed into its corresponding MCG, it is a simple matter to construct an equivalent object-oriented representation using the classes of the

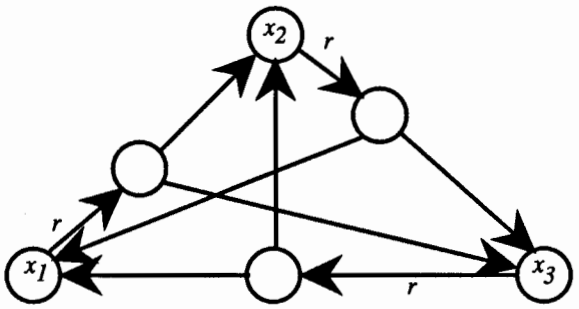


Figure 3.14: The n-ary constraints instantiated as nodes.

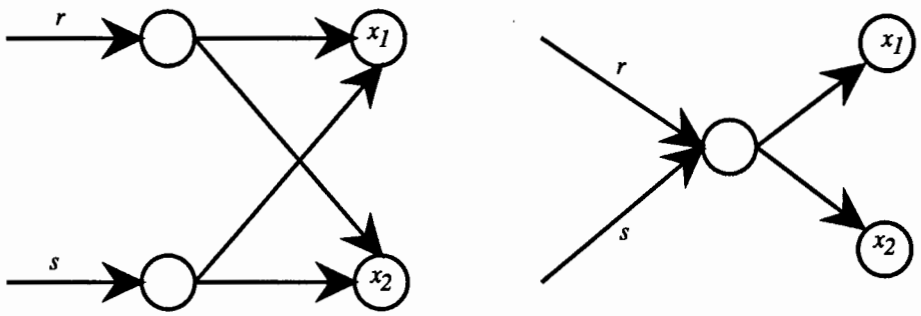


Figure 3.15: Merging two ternary constraint nodes into one.

previous section.

Each node in the MCG is an instance of an object. The labeled nodes are instances of class Variable whereas the unlabeled nodes (as hinted at previously) are instances of the class Constraint (or rather, one of its subclasses depending on the arity of the constraint it is enforcing, denoted by its out-degree).

Each labeled arc is an instance of class Link. The incident-to Constraint object is the *node*, and the label on the arc is the *label*. Each *label* is a selector for a method that the objects comprising the Variable domains respond to. These methods define the constraints. Unlabeled arcs denote instance variable initializations for Constraint objects.

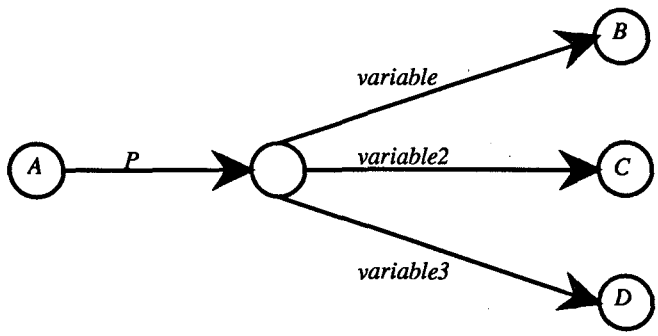
The instances of class Variable collected together form an instance of class Csp. A particular CSP is solved by creating and initializing the objects that comprise the corresponding MCG and sending the message *findSolutions* (perhaps after first sending the message *makeArcConsistent*) to the Csp object.

3.2.3 Representing Constraints

Constraints are enforced by Boolean methods that return *true* if the constraint is satisfied.[2] In CSP-IC a constraint is comprised of a declarative component (the predicate implemented as a Boolean method) and a procedural component (the Constraint object that knows how to use the predicate to filter domains). There is no direct mapping from the set of constraints C in a CSP to objects in this system. When we say *constraint* we are generally referring to the Boolean method. A single Boolean method often can be used to enforce every constraint in a particular CSP. For example, n-queens and graph colouring are two CSPs where each constraint is the same predicate applied to different subsets of the variables.

Since constraints are relations on the objects collected in the Variable objects' domains, they are defined within the class to which these objects belong. By allowing the user to

[2] By choosing a procedural representation for the constraints we have ruled out the incorporation of any of the path consistency algorithms. This is not a serious malady since "...past experiments show them to be not cost-effective in general" [Nadel89, p. 189].



Constraint objects order parameters for the constraint predicates as follows: (sender, variable, variable2, variable3, ...). For the above MCG, consistency would be checked by calling P(a,b,c,d) for objects a, b, c, and d belonging to the domains of variables A, B, C, and D respectively.

Figure 3.16: The relationship between arcs and predicate parameters.

define the constraints within the implementation language, we have provided computational completeness: constraints of arbitrary complexity can be defined amongst objects of arbitrary complexity. In particular constraints of a type Leler calls *higher-order* can be enforced [Leler88]. These are constraints that depend on other constraints and require a conditional operator for their implementation.

In general, n different methods need to be defined to enforce an n-ary constraint, each corresponding to the filtering of $n-1$ variables given a change to a single variable in the relation. This is due to the shuffling of parameters in the predicates, and prompted the transpose arcs in the MCG. In Figure 3.16, the relationship between arcs and predicate parameters is clarified. Instances of subclasses of Constraint must be initialized carefully in order to ensure the proper placement of predicate parameters.

Of course, some relations are fortuitously symmetric and require only the implementation of one method. Some contain a symmetric component and require the definition of some number in between 1 and n. Figure 3.17 lists the two methods that would be required to enforce the ternary constraint $A = B + C$ on objects of an ordered class that return their

```
// When A has changed.
- (BOOL) p1: b and: c
{
   return (value == [b getValue] + [c getValue]);
}

// When B or C has changed.
- (BOOL) p2: a and: bOrc
{
   return (value == [a getValue] - [bOrc getValue]);
}
```

Figure 3.17: The methods required to enforce the constraint A=B+C.

value when sent the message *getValue*. Method *p2:and:* can be used to check consistency when either object B or C has experienced a domain reduction since $B = A - C$ and $C = A - B$, i.e., the first and third parameters can be interchanged.

3.2.4 Arc Consistency within the Graph

As a result of the messaging that occurs within the MCG whenever a Variable object experiences a change to its domain, a full arc consistency algorithm transpires. Figure 3.18 details the arc consistency algorithm that results when a variable in the graph is either instantiated or has its domain reduced. To facilitate comparison with the arc consistency algorithms dicussed in Chapter 2, it is assumed the constraint graph is comprised of binary constraints only. Here G refers to the original constraint graph, not its corresponding MCG.

Alternatively, arc consistency can be enforced without variable instantiation by simply sending the message *makeArcConsistent* to an instance of Csp. The resulting algorithm is a variant of Mackworth's AC3. Figure 3.19 gives a comparative algorithm of the arc consistency that occurs as a result of this message. The algorithms of Figure 3.18 and Figure 3.19 are identical but for the initializations of Q.

There are three notable discrepancies in *makeArcConsistent* compared to AC3. Firstly,

```
Q ← {(j,x) | (j,x) ∈ arcs(G), j ≠ x}
while Q ≠ ∅
   begin
      pop (Q,(k,m))
      if REVISE(k,m)
      then push (Q,{(i,k) | (i,k) ∈ arcs(G), i ≠ k})
   end
```

Figure 3.18: The arc consistency procedure that results from instantiating node x.

```
Q ← ∅
for i = n..1 do
    push(Q,{(j,i) | (j,i) ∈ arcs(G), i ≠ j})
while Q ≠ ∅
    begin
       pop (Q,(k,m))
       if REVISE(k,m)
       then push (Q,{(i,k) | (i,k) ∈ arcs(G), i ≠ k})
    end
```

Figure 3.19: The *makeArcConsistent* algorithm.

node consistency is not explicitly enforced. The assumption is made that the variables' domains have all been initialized appropriately. Second, Q is a stack: arcs to reconsider are added to the front of Q. Finally, REVISE(i, j) returning *true* causes the unnecessary addition of arc (j, i) to Q. This is an unwanted and unavoidable consequence of the communication protocol within the graph.

As a result, *makeArcConsistent* is susceptible to redundant arc considerations. A formal comparison of the relative efficiency of *makeArcConsistent* with respect to AC3 can be done by applying the analysis in [Mackworth85]. Let e be the number of edges in the nondirected constraint graph, and let d_i be the edge degree of vertex i. Two restrictions apply: consider only binary constraints and assume that $\mid D_i \mid = a, 1 \leq i \leq n$. As mentioned in section 2.3.2, with these restrictions AC3 is $O(a^3e)$.

It is now shown that *makeArcConsistent* is also a $O(a^3e)$ algorithm. Initially, the length of Q is $2e$; each arc and its transpose must be made consistent. The *for* loop carries out this initialization of Q. During iterations of the *while* loop, the stack Q expands and contracts until it is finally empty. In the worst case, each time REVISE(k, m) is called, only one element is removed from k's domain. Since this removal from the domain results in REVISE returning *true*, all d_k arcs incident to k are pushed onto Q. Therefore, d_k arcs are added to Q at most a times for each node in the graph. The total number of entries to Q is:

$$\sum_{k=1}^{n} ad_k = a \sum_{k=1}^{n} d_k = 2ea$$

Each iteration removes one arc from Q leaving the total number of iterations at $2e+2ea$ i.e., the initial size of Q plus the number of additions. Each iteration of the *while* loop makes one call to REVISE, which requires at most a^2 evaluations of a binary predicate. The total running time for the algorithm is thus at most $a^2(2e + 2ea)$ and *makeArcConsistent* is $O(a^3e)$.

By comparison, AC3's running time is at most $a^2(2e + a(2e - n))$. In the worst-case, *makeArcConsistent* requires a^3n more time than AC3.

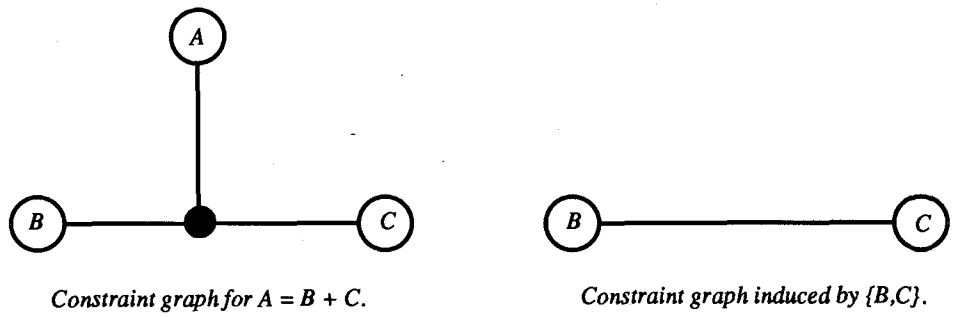


Figure 3.20: Inducing a subhypergraph of a constraint graph.

3.2.5 Tree Searching the Graph

The method *findSolutions* (and *findASolution*) performs a backtrack search augmented with full arc consistency. At level k in the search tree, variable x_k is instantiated and the constraints propagated. A similar search, called TSAC3, is described in [Nadel89]. However, *findSolutions* is a slightly different implementation: it does not invoke a full arc consistency procedure, like AC3, that checks every arc in the graph at least once. Instead, nodes arc only visited as necessary (see Figure 3.18 for the consistency checks performed when a variable is instantiated). Moreover, because Nadel restricts his discussion to binary constraint graphs, at level k in the search tree TSAC3 performs full arc consistency only on the subgraph of G induced by $\{x_{k-1}, \ldots, x_n\}$.

However, if G is a hypergraph, pruning by means of inducing a subhypergraph over the uninstantiated variables becomes a complex operation. Consider the constraint graph of Figure 3.20 representing the ternary constraint $A = B + C$, and the subhypergraph induced by $\{B, C\}$. As illustrated, inducing a subhypergraph actually requires changing the constraint from a ternary relation on $\{A, B, C\}$ to a binary relation on $\{B, C\}$. If after instantiating variable A the ternary relation is simply discarded, further search will yield additional and incorrect solution paths.

There are two remedies to this problem when searching a hypergraph. One is to actually reconstruct the constraints over the induced subhypergraph. This is possible if the constraints are represented in a declarative form (e.g., a table). For example, let R be a relation denoting tuples acceptable to an n-ary constraint on $\{x_1, \ldots, x_n\}$. If $x_1 \leftarrow a$, then the induced relation R' on $\{x_2, \ldots, x_n\}$ is, in relational algebra:

$$R' = \pi_{2,\ldots,n}(\theta_{1=a}(R))$$

where π and θ are the projection and selection operators respectively.

A less complicated method is to simply forget about restricting the node set that arc consistency is applied to, i.e., do not induce a subhypergraph. Since the instantiated variables still have a role to play in constraining the future variables, they should stay within the graph. This latter method is employed in *findSolutions* since the cost of restructuring a hypergraph to avoid the unnecessary consistency checks with the instantiated variables was seen as too high for any expected return in efficiency.

Therefore in *findSolutions*, at level k in the search tree, variable x_k is instantiated and arc consistency is propagated over all neighbouring nodes of the graph, not just nodes x_{k-1} to x_n. As such, in finding all 92 solutions to the (complete binary) 8-queens problem, *findSolutions* performs 54,613 constraint checks compared to 51,188 for Nadel's TSAC3.[3] However, if TSAC3 is to work on constraint hypergraphs, it must adopt one of the two described remedies: in its current form, it will report extra solutions.

Interestingly, only 602 consistency checks are required to determine that the 8-queens graph is initially arc consistent. Thus, not preprocessing the graph with a call to *makeArc-Consistent* did not result in a significant savings for *findSolutions*. Though espoused by others as a potentially important use for AC3 [Mackworth85], preprocessing the constraint graph before tree search begins in no way helps to reduce the search space for the 8-queens

[3]Some of these extra consistency checks in *findSolutions* may be attributed to its implementation of Q as a stack rather than, as in TSAC3, as a queue, as well as redundant arc considerations inherent in the algorithm.

problem. This is, of course, not true in general for CSPs as reaffirmed in Chapter 4.

In [Freuder82] the vertical order of a search tree (the order in which variables are instantiated) is shown to be of potential importance in minimizing the number of backtrack points. By making the instance variable *variables* in the class Csp an ordered collection, the vertical order for the tree search is defined by the sequence of *addToVariables* messages. Similarly, the horizontal order (the order in which values are assigned to a variable) is determined by the sequence of *addToDomain* messages used to initialize a Variable object's domain.

3.2.6 Complexity of the Modified Constraint Graph

The modified constraint graph is more complex than the standard constraint graph from which it is derived, but as demonstrated, its advantages include a direct correspondence to objects in CSP-IC. The complexity of each step in the transformation can be analyzed by noting how many additional edges and nodes are added to the original constraint graph.

The removal of loops from the constraint graph (step 1) cannot be seen as a reduction in complexity as they are generally omitted anyway. The variable's domain can always be initialized in such a way as to reflect any unary constraint.

The introduction of all implicit transpose arcs (step 2) can potentially double the number of arcs in the constraint graph to $2e$ arcs total (that is if the original constraint graph is nondirected). This is a necessary step if any constraint graph is to be subjected to an arc consistency procedure since they must be treated differently by the algorithm.

In step 3, a node is created for each arc in the graph that resulted from step 2. Thus, up to $2e$ additional nodes are added to the graph. However, in step 4, many of these extra nodes are collapsed. The exact number of constraint nodes that remain after step 4 is dependant on the constraints of the original graph. Consider first the case of a graph with binary constraints only. In general, if the graph is connected (i.e., each node participates in at least one binary constraint), the completed MCG has n binary constraint nodes: one incident to each variable. Figure 3.21 is the MCG for the 4-queens problem and illustrates

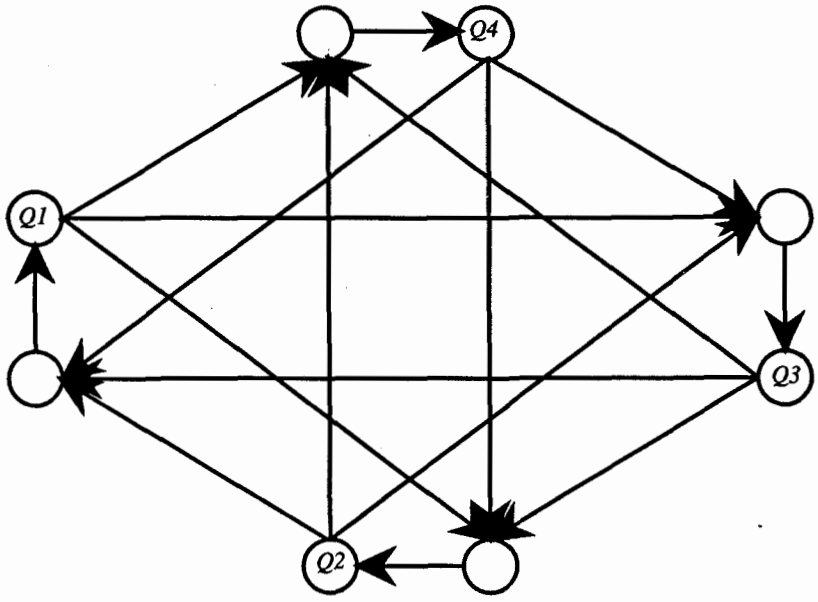


Figure 3.21: The MCG for the 4-queens problem.

this point. A complete binary constraint graph, like n-queens, has n nodes and $n(n-1)/2$ arcs. The corresponding MCG has n variable nodes, n constraint nodes, and $n(n-1)$ arcs, and thus the OOP implementation would require n Variable objects, n Constraint objects, $n(n-1)/2$ Link objects (remembering that the unlabeled arcs are only references to objects), and one instance of class Csp.[4]

Analysing the effect of n-ary constraints is more difficult and is too dependent on the original constraint graph to attempt a formal analysis. Each hyperarc connecting n nodes in the original constraint graph results in n additional (constraint) nodes and $n+n(n-1)$ arcs in the MCG. Node collapses necessarily occur if the CSP is complete with respect to any n-ary hyperarc. Regardless, hyperarcs in the original constraint graph have the potential to seriously increase the complexity of the MCG.

[4]Domain objects are not an intrinsic part of CSP-IC. For n-queens there would be n^2 domain objects.

3.3 A Sample Problem

Appendix B contains the source code for, and small discussion of, the solution to the n-queens problem, $n \leq 10$, using CSP-IC. The MCG for the 4-queens problem, given in Figure 3.21, is helpful for the comprehension of this implementation, particularly the *main.m* file where the MCG is constructed.

3.4 The Accomplishment

Leler says that constraint programming is declarative, such that the control is at the discretion of the system [Leler88]. The proposed object-oriented system of this chapter is consistent with this desire for declarativeness. Once the underlying structure (the object classes detailed in section 3.2) is in place, there is no algorithm that the programmer need be concerned with. To solve a CSP, the programmer need only worry about defining the modified constraint graph and the methods (predicates) that enforce the constraints.

If liberating the programmer from tree search and constraint propagation is the most important reason for the development of constraint programming languages, then this methodology has achieved this result but without introducing yet another programming language. By exploiting both the abstraction facilities of OOP (i.e., classes) and its message passing mechanism it is possible to construct a framework for the general solution of the class of CSPs that incorporates a full arc consistency labeling procedure. This framework is reusable, independent of any specific problem instance, and applicable to any CSP that can be represented by a constraint (hyper-) graph.

Chapter 4

Music Composition as a CSP

A true test for CSP-IC and its accompanying methodology is found in the domain of music composition. Music composition can be readily viewed as a CSP: composers select the attributes of the notes (pitch, duration, etc.), from finite domains, that satisfy the constraints they feel produce the most aesthetically pleasing results. Composition is also quite appropriately suited to further our investigation because music can involve complex constraint relationships that render a CSP anything but a toy. In this chapter we apply our methodology to the problem of producing music compositions in the style of 17th century counterpoint.

Marvin Minsky has said that "...the problem of making a good piece of music is a problem of finding a structure that satisfies a lot of different constraints" [Minsky80], and Curtis Roads concurs that as a knowledge representation formalism for music, constraints show promise [Roads85]. This chapter provides insight into the relevance of abandoning all other forms of knowledge representation (rules, logic, frames, etc.) in favour of viewing music composition strictly as a CSP.

There are two expected benefits to this approach. If a composition exercise has a natural formulation as a CSP, and, in particular, as a constraint graph, then an important discovery about its tractability can be made: the constraint graph representation of the problem

can suggest any expected limitations if backtracking is the control method used to solve the CSP[Freuder82]. The second benefit is the ability to isolate each rule of syntax and analyze its effect on constraining the number of allowable compositions considered a part of a genre. This would allow one to subjectively rank the aesthetic importance of each rule in determining the characteristics of a composition style.

The remainder of this chapter assumes a basic understanding of the fundamentals of music. Counterpoint is a style of music composition that pre-dates our modern notion of harmony. Whereas harmony is concerned mainly with vertical relationships among notes, chords in succession, and supporting a dominant melodic theme, counterpoint is *polyphonic* composition: the combination of several independent and equally interesting melodies into a coherant whole. The rules of counterpoint, which constrain the allowable note combinations occuring in a composition, were codified in 1725 by J. J. Fux. A translation of this work appears in [Mann65]. The historical development of polyphonic music in general is detailed in [Swindale62].

The normal manner of composition in counterpoint is to craft additional melodic lines (called *counterpoints*) to be sung, or played, along with a previously composed *cantus firmus*.[1] The new voices are in counterpoint to the original melody.

4.1 Related Work in Computer Composition

This brief overview of some earlier research into the problem of automated composition and harmonization is not meant to be an exhaustic survey. Instead, we concentrate on those systems that seek to work from a declarative musical knowledge base as oppossed to algorithmic composition systems. In essence, we are concerned only with attempts to apply ideas developed in artificial intelligence research to the domain of music composition. A good survey of the field in general is in [Roads85].

[1] A melody, usually taken from a book of chorales. In this chapter, the terms *melody* and *cantus firmus* are used interchangeably.

Hiller and Isaacson were the first to experiment with using a digital computer for music composition [Hiller59]. Their system permitted the writing of simple melodies, and up to four-part writing in first species counterpoint incorporating a fairly complete set of rules. They employed a random *generate and test* method that moved sequentially through each bar of the composition. After generating 50 unsuccessful candidates for the next required note, their program gave up and started over: no backtracking was employed.

Hiller and Isaacson fully understood the importance of constraints as knowledge representation:

> ...the process of musical composition requires the selection of musical materials out of a random environment. This is accomplished by a process of elimination. The extent of order imposed depends upon the nature of the restrictions imposed during the process of selection [Hiller59, p. 22].

In other words, order depends on constraint.

Recent attempts to develop expert systems for music composition often employ a rule-based knowledge representation scheme ([Thomas85, Cope87, Ebcioglu86], among others); this is understandable given the popularity and effectiveness of rule-based expert systems in many domains outside of music. However, rule-based systems are frequently inefficient due to their reliance on chronological backtracking as the control method. Those aware of this dilemma resort to tactics such as intelligent backtracking (like Ebcioglu, who was effectively trying to improve the search efficiency of logic languages at the same time as Van Hentenryck succeeded in doing just that), but as of yet, no one has incorporated consistency techniques in an effort to improve the search efficiency of automated composition. Since many of these composition systems are attempting to satisfy multiple constraints across several variables (i.e., solve a CSP) while utilizing backtracking as the control method, they are susceptible to an exponential explosion.

However, Levitt has successfully utilized constraints in a system for Jazz improvisation [Levitt84]. Levitt proposes to model some aspects of composition as a psuedo-electronic circuit that, rather than filter domains, computes new note attributes from known ones (i.e.,

value propagation). But like all systems built upon this paradigm it is restricted to equality relations and does not reduce its search space to the fullest capability [VanHentenryck89].

4.2 Counterpoint as a CSP

The remainder of this chapter describes how the problem of generating contrapuntal music can be formulated as a constraint satisfaction problem. Specifically, compositions of first species counterpoint that adhere to a fairly complete set of rules taken from [Mann65] and [Swindale62] are modeled. The first species is counterpoint of the simplest form: two or more voices comprised of notes of equal length. This restriction eliminates the myriad of representation problems concerned with rhythm and time. The only attribute of a note that requires representation is its pitch, which can be done with an integer from 1 to 127 corresponding to an ordering of the semi-tones with middle-C equal to 60. This is in accordance with the Musical Instrument Digital Interface (MIDI) standard [IMA83]. A numeric representation such as this facilitates easy computation of intervals; it merely requires subtraction.[2]

The task to be performed is that of adding a counterpoint to a given melody, which can be modeled as a CSP; the variables are the notes of the counterpoint, and the constraints are the rules regarding allowable intervals, motion, etc., which effectively filter the notes that may appear together in a composition. A first species counterpoint composition in two voices n bars in length is thus comprised of n counterpoint variables $\{c_1, \ldots, c_n\}$ and n melody constants $\{m_1, \ldots, m_n\}$. It is convenient to think of the notes of the melody as instantiated variables rather than constants. Using this notation, the ith bar of a composition is comprised of note m_i in the melody and c_i in the counterpoint. The task is thus to assign values to $\{c_1, \ldots, c_n\}$ that satisfy the rules of first species counterpoint given an instantiation for each of $\{m_1, \ldots, m_n\}$.

[2] Any representation scheme that facilitates the definition of constraint relationships on pitch values would be acceptable.

4.2.1 The Rules as Constraints

In this section we show how each rule of first species counterpoint can be converted to a local constraint representation. As is shown, the enforcement of a single rule of counterpoint often requires that many instances of the same predicate be applied to many different subsets of the variables. In other words, a rule is comprised of many constraints. We therefore define a *rule* to be a non-empty set of constraints. Each rule is referred to by the name given to the predicate used by the constraint(s) that enforces the rule.

The rules are categorized here according to the arity of their corresponding predicate. The variable(s) that the predicates are appropriately applied to is illustrated by a constraint graph representing counterpoint compositions of length four, the smallest such compositions that utilize all the rules of first species counterpoint. As the rules are introduced, more and more constraints are added to the constraint graph until finally it completely embodies the task of composing first species counterpoint.

Unary Constraints

The unary constraints restrict the pitch values for the notes of the compositions to those that are realizable by a single instrument. Since counterpoint is generally written for voice, we impose an arbitrary two-octave range, though others prefer that this range be even smaller (see [Hiller59]):

> *Each note must be taken from the Aeolian mode (the white keys of a piano) and must not range over more than two octaves.*

$$\text{mode}(i) \equiv i \in \{45, 47, 48, 50, 52, 54, 55, 57, 59, 60, 62, 64, 65, 67, 69\}$$

Applying this unary predicate to each note of the composition satisfies this rule (Figure 4.1).[3] Thus we see how a single rule of counterpoint is implemented by the application of multiple local constraints.

[3] In this and other constraint graphs we are treating the notes of the melody as variables.

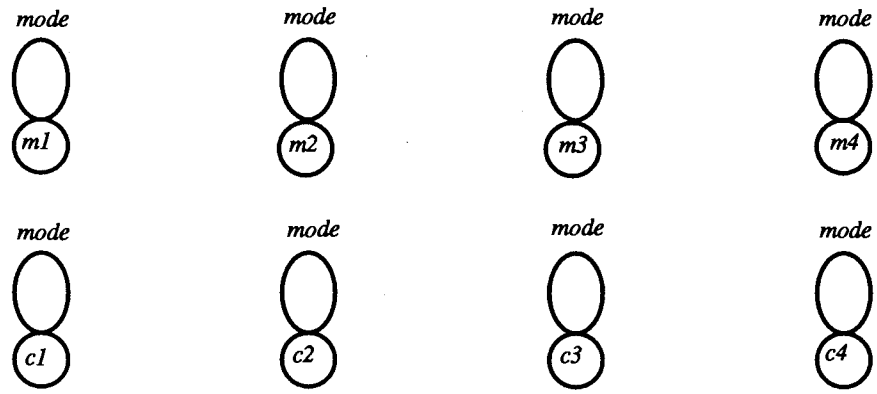


Figure 4.1: Constraints that enforce the *mode* rule.

Restricting the examples to the Aeolian mode is a simple way to avoid problems of modulation since compositions in the Aeolian mode do not normally stray from the scale [Swindale62]. The exception to this rule occurs during formation of the cadence:

The cadence in the Aeolian mode is formed with G♯.

$$\text{cadence}(i) \equiv i \in \{56, 68\}$$

This unary predicate is applied to the second to last note of the counterpoint and replaces the looser *mode* constraint (Figure 4.2).

Binary Constraints

Most of the harmonic rules governing allowable intervals are binary relationships between notes m_i and c_i. Though the counterpoint is restricted to stay within a twelfth of the melody, no restriction regarding the crossing of voices is in place. A single constraint graph containing all of the binary constraints to be introduced is given at the end of this section.

The last bar must be a perfect consonance. These are: unison, fifth, octave, and twelfth.

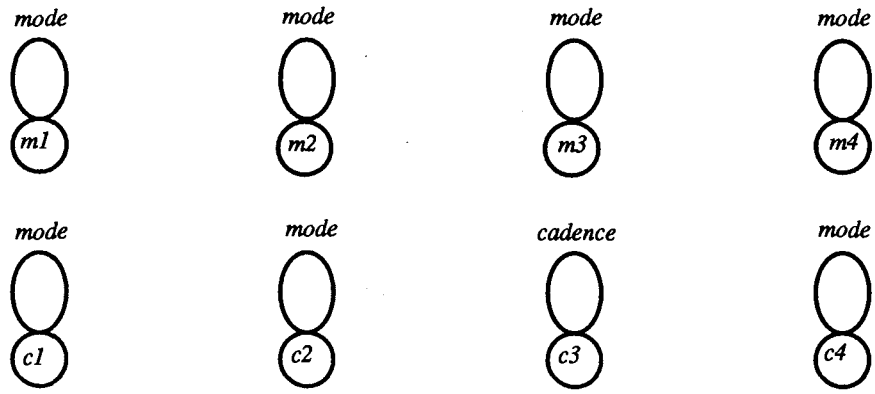


Figure 4.2: Constraints that enforce the *mode* and *cadence* rules.

$$\text{perfect}(i,j) \equiv \mid i-j \mid \in \{0,7,12,19\}$$

This predicate is thus applied to c_n and m_n, the notes that comprise the last bar of the composition.

The following rule assures that the counterpoint is in the same mode as the melody.

> *If the counterpoint is in the lower part, the first bar must be either a unison or an octave.*

$$\begin{aligned} \text{first}(i,j) \equiv \quad & \text{if } i > j \\ & \text{then } i - j = 12 \\ & \text{else perfect}(i,j) \end{aligned}$$

This higher-order predicate is applied to the notes of the first bar. The allowable harmonic relationships in all other bars is given by the following rule:

> *In all other bars, permitted intervals are the consonances up to the twelfth (excluding unison). These are: minor third, major third, fifth, minor sixth, major sixth, octave, minor tenth, major tenth, and twelfth.*

$$\text{harmonic}(i,j) \equiv \mid i-j \mid \in \{3,4,7,8,9,12,15,16,19\}$$

Instead of a unary constraint, one could define the *cadence* rule as a higher-order binary relation:

> *If the counterpoint is in the upper part, there must be a major sixth in the second to last bar. If the counterpoint is the lower part, it must be a minor third.*

$$\begin{aligned}\text{cadence}(i,j) \equiv \quad & \text{if } i > j \\ & \text{then } i-j = 3 \\ & \text{else } j-i = 9\end{aligned}$$

However, formulating the cadence in this manner creates a potential conflict with the *mode* rule, which does not allow the usually necessary accidental to occur in the counterpoint. If we assume that "...the second degree of a mode occurs always as the next to the last tone in the cantus firmus..." [Mann65, pp. 28-29] then *cadence* applied to c_{n-1} is sufficient to guarantee the appropriate cadence. This would also allow the application of a stricter unary constraint on variable m_{n-1}.

The following rule eliminates melodic intervals that, deemed difficult to sing, are strictly forbidden:

> *Melodic skips greater than a minor sixth are not allowed, nor are skips of a tritone (augmented fourth).*

$$\text{melodic}(i,j) \equiv \mid i-j \mid \leq 8 \wedge \mid i-j \mid \neq 6$$

The constraint graph with the addition of these binary constraints is given in Figure 4.3.

Ternary Constraints

Local melodic contour rules require constraining three consecutive notes of a line. For example, skip-step motion:

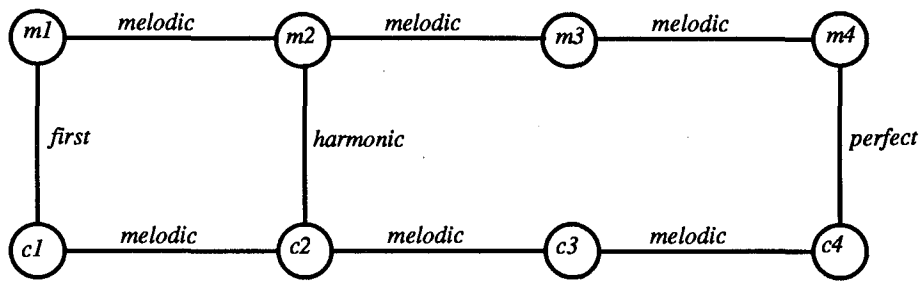


Figure 4.3: Binary constraints enforcing the *perfect, first, harmonic, and melodic* rules.

Any melodic skip greater than or equal to a minor third must be followed by stepwise motion; that is, motion less than a minor third.

$$\text{skipStep}(i,j,k) \equiv \mid i-j \mid \leq 3 \vee \mid j-k \mid < 3$$

Relationships on three successive notes are also an opportunity to promote variety by forbidding more than one successive repeat of a given note (a rule from [Hiller59]):

The same note three times in a row is not permitted.

$$\text{noThree}(i,j,k) \equiv (i \neq j \vee i \neq k)$$

Though not an explicit rule of first species counterpoint, the *noThree* rule is included in our treatment as it is a good example of how local constraints can be used to effect the global contour of a melodic line. For example, a gradual upward slope in the counterpoint could be enforced by a set of binary constraints like the following:

$$1 < i \leq n-2 : \text{climb}(c_i, c_{i+2}) \equiv c_i < c_{i+2}$$

Since the *skipStep* and *noThree* rules involve constraints on the same sets of variables, it is convenient to combine them into a single predicate:

$$\text{ternary}(i,j,k) \equiv \text{skipStep}(i,j,k) \wedge \text{noThree}(i,j,k)$$

The inclusion of the ternary constraints results in the constraint graph of Figure 4.4.

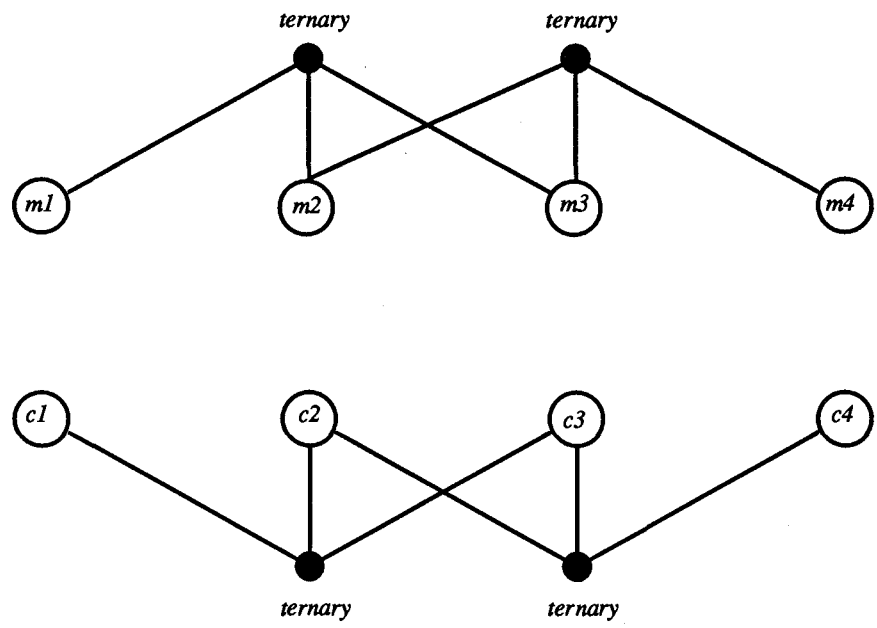


Figure 4.4: Ternary constraints enforcing the *skipStep and noThree* rules.

Quaternary Constraints

What could be considered the fundamental rule of counterpoint is enforced by a set of quaternary constraints:

Parallel motion to a perfect consonance is not permitted.

$$\text{parallel}(i,j,k,l) \equiv \neg\text{perfect}(k,l) \vee \frac{\mid i-k \mid}{\mid j-l \mid} < 0$$

The last rule to be considered is also a quaternary constraint:

Progressions to an octave by a skip are not to be tolerated.

$$\text{octave}(i,j,k,l) \equiv \mid l-k \mid \neq 12 \vee (\mid i-k \mid \leq 3 \wedge \mid l-j \mid \leq 3)$$

Again since these two predicates are to be applied to the same sets of variables it is convenient to define

$$\text{quaternary}(i,j,k,l) \equiv \text{parallel}(i,j,k,l) \wedge \text{octaveSkip}(i,j,k,l)$$

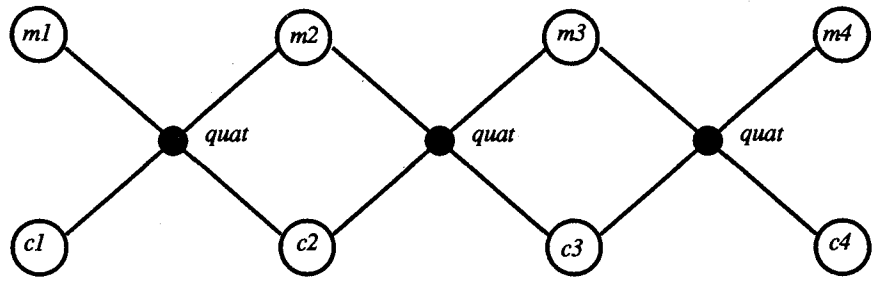


Figure 4.5: Quaternary constraints enforcing the *parallel* and *octave* rules.

Figure 4.5 is this quaternary constraint applied to the appropriate variables in order to enforce the *parallel* and *octave* rules.

4.2.2 A Constraint Graph for Counterpoint

The completed constraint graph for first species counterpoint of note against note is presented in Figure 4.6. It is, of course, in actuality a hypergraph. For the sake of clarity, the unary constraints have been removed. This graph is for compositions of length four, the shortest compositions to include all the rules of first species counterpoint, and can easily be extended for compositions of greater length.

Implementing counterpoint as a CSP using CSP-IC requires the construction of a MCG that corresponds to the constraint graph of Figure 4.6. The notes of the melody appear as variables in the constraint graph, but since they are all instantiated, some slight simplifications can be made. Certain reciprocal constraints (e.g., *harmonic* from c_2 to m_2) can be excluded and all constraints that would apply only to melody variables (e.g., *melodic* on m_i, m_{i+1}) are assumed to hold and thus excluded altogether. Figure 4.7 is the constraint graph that reflects these simplifications.

The corresponding MCG for Figure 4.7 is given as a collection of three different figures (Figure 4.8, Figure 4.9, Figure 4.10) that separately detail the binary, ternary, and quaternary constraint components respectively. For expediency, we have failed to define the various transposed predicates necessary in the actual implementation (e.g., *ternary* requires three

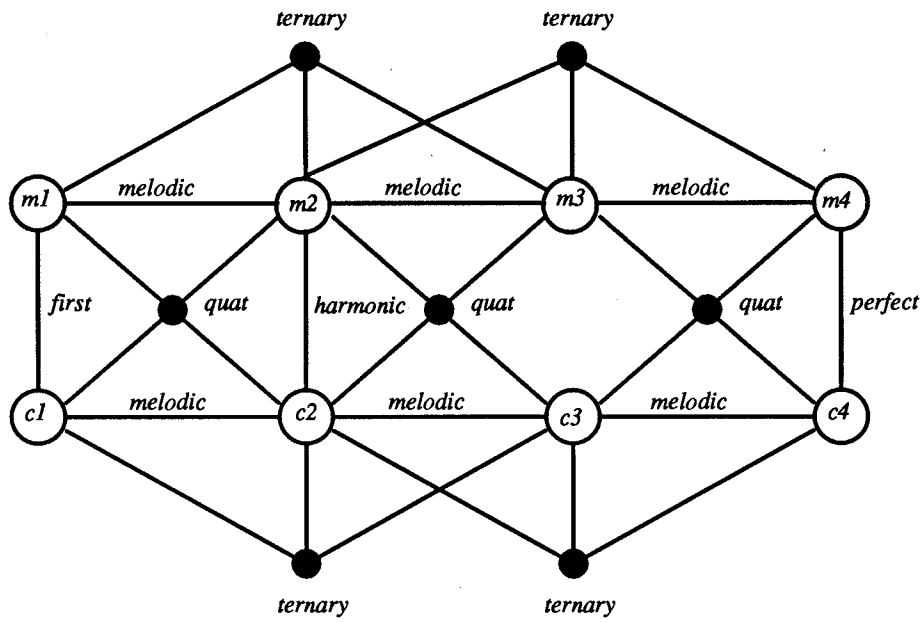


Figure 4.6: A constraint graph for first species counterpoint.

different versions depending on the variables it is applied to). The inclusion of ternary and quaternary constraints results in a rather complex MCG. However, the nearest-neighbour property of many of the constraints permits an easy extension to the size of the graph (i.e., the length of the composition). The *main.m* file that constructs the MCG for first species counterpoint is in Appendix C. Results of this implementation are discussed later in this chapter.

Intuition would suggest that first species counterpoint is not a hard problem; a standard backtracking procedure may find an acceptable solution, and if so, often without any backup. However, there is no guarantee that this will always be the case. In [Freuder82] it is shown that a sufficient condition for backtrack-free search is the application of a strong k-consistent algorithm to the constraint graph, given k is greater than the width of the graph. A backtracking search that employs a vertical order with width less than k is then guaranteed not to backtrack. The application of Freuder's theory to constraint *hypergraphs*, such as

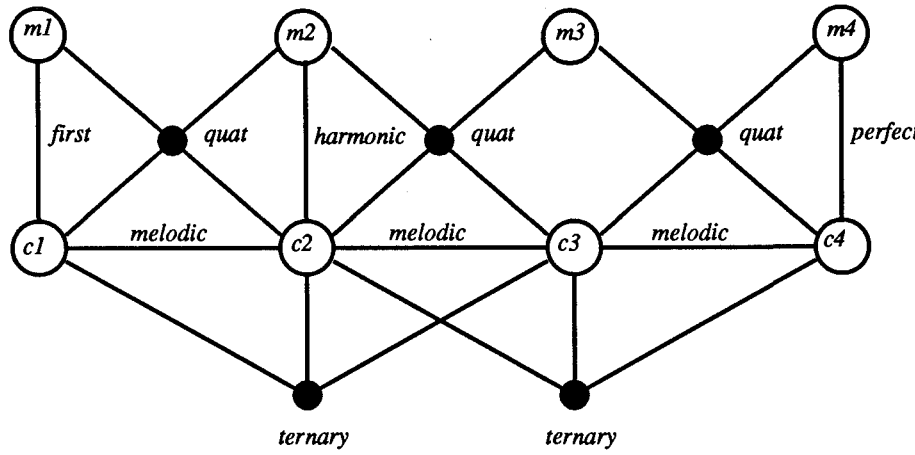


Figure 4.7: A simplified constraint graph for first species counterpoint.

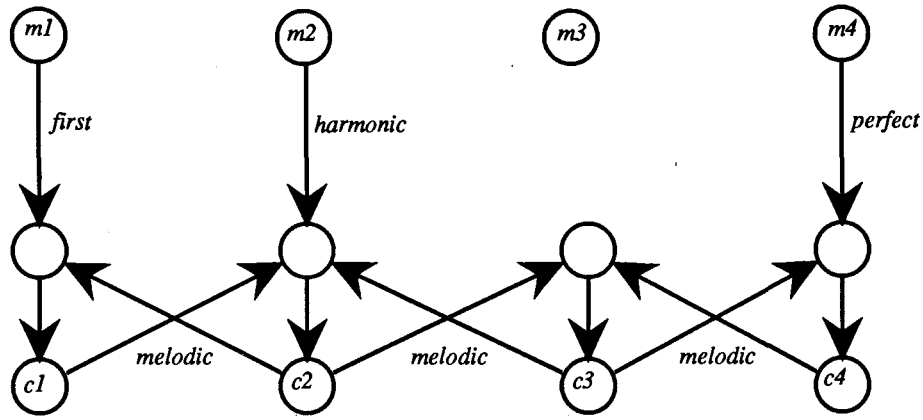


Figure 4.8: The binary constraints in the MCG for first species counterpoint.

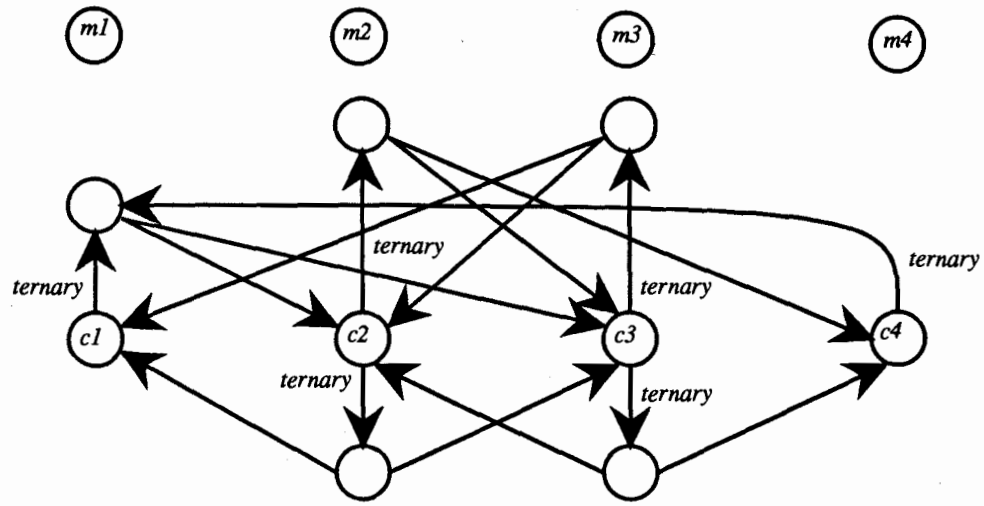


Figure 4.9: The ternary constraints in the MCG for first species counterpoint.

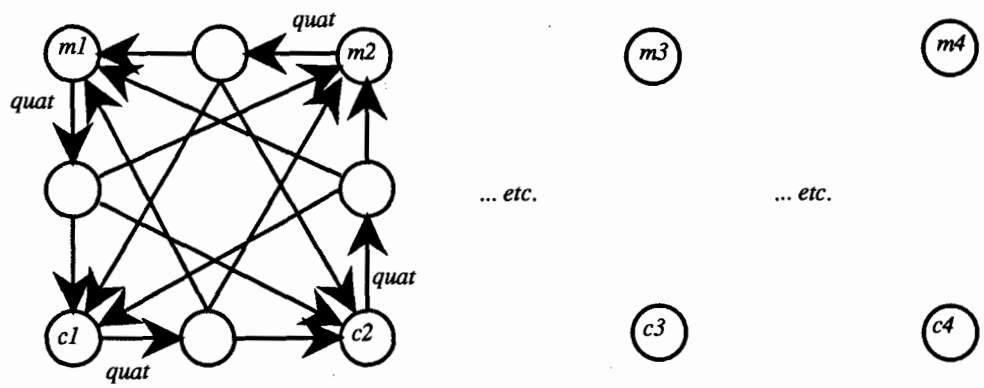


Figure 4.10: The quaternary constraints in the MCG for first species counterpoint.

```
To find the width k of a simple hypergraph:
Remove from the hypergraph all nodes not connected to any others.  Set k to 0.
Do while there are nodes in the hypergraph.
   Set k equal to k + 1.
   Do while there are nodes with degree less than or equal to k.
      Remove all nodes from the hypergraph with degree less than or equal to k.
      Generate the subhypergraph on the remaining nodes.
```

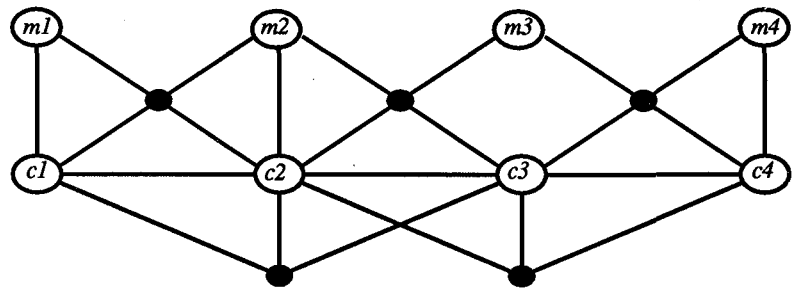
Figure 4.11: Algorithm for finding the width of a hypergraph.

needed to represent the counterpoint problem, requires a generalization of his algorithm for the determination of width (Figure 4.11).

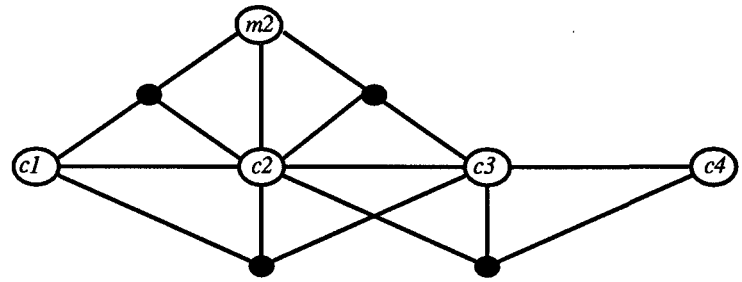
Using this algorithm it is determined that the hypergraph for first species counterpoint has width $k = 3$ (see Figure 4.12 for the details of this calculation). Thus a backtrack-free search is guaranteed if the graph is first processed by a strong 4-consistent algorithm.[4] Therefore, pre-processing with a strong 2-consistent algorithm (e.g., AC3) does not guaranteed an efficient backtracking search. A general backtracking search without any consistency techniques would certainly be prone to an exponential explosion, particularly if all solutions are to be found.

However, Freuder's result predicts nothing of the expected inefficiency of a tree search coupled with arc consistency (e.g., *findASolution* and *findSolutions*). In practise, these algorithms perform admirably well, requiring relatively few backups. Table 4.1 gives some empirical measurements of the two methods at work on generating counterpoints for a four-bar melody (Figure 4.13), seven-bar melody (Figure 4.14) and twelve-bar melody (Figure 4.15) when the graph is first processed with *makeArcConsistent*. In contrast, table 4.2 gives the same measurements when the graph is not first pre-processed. As seen from these results, pre-processing the graph with a full arc consistency algorithm is a good idea. Particularly

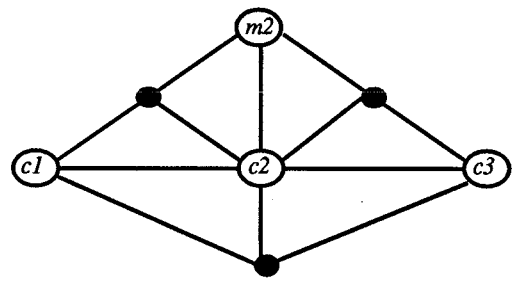
[4] Therefore, a polynomial time algorithm exists for composing a single consistent counterpoint composition of any length: make the graph 4-consistent, an $O(n^4)$ operation, and follow up with a tree search that, since it is guaranteed to be backtrack-free, will run in time linear in n.



No nodes have degree < 2. k = 2: m1, m3, and m4 have degree 2.



Now c4 has degree 2.



No other nodes have degree 2 or less. k = 3: c1 and c3 have degree 3.

Now c2 and m2 have degree 1. k = width = 3.

Figure 4.12: Finding the width of the counterpoint constraint graph.

Figure 4.13: A four-bar melody for which counterpoint is to be generated.

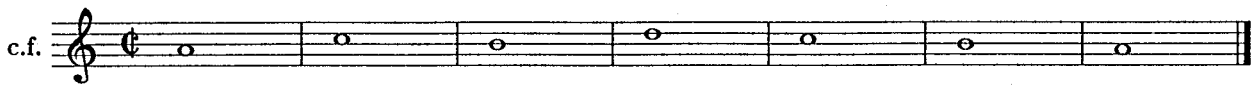


Figure 4.14: A seven-bar melody for which counterpoint is to be generated.

striking in comparison is the number of backtracks needed to find a single solution for the 12-bar melody: no backup is necessary if the graph is first made arc consistent compared to the partial exploration of 1,496 dead-ends without the pre-processing.

Sample counterpoints generated by our implementation are found in Figure 4.16. We have re-produced, as reported by the *findSolutions* method, the first and last permissable counterpoints for each of the three melodies. These correspond respectively to the lowest and highest allowable counterpoints within the two-octave range.

4.2.3 An Analysis of the Individual Rules

The constraints of counterpoint together define a space we might call the language of counterpoint and allow us to test if a sentence (composition) is a member of this language. In this sense, the constraints define the syntax of counterpoint and guide the generation of the

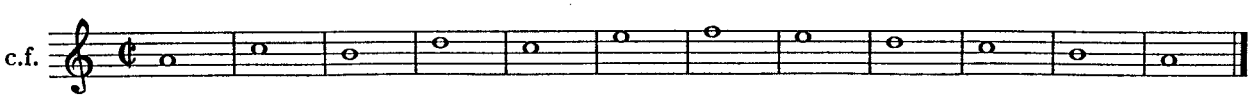


Figure 4.15: A twelve-bar melody for which counterpoint is to be generated.

	Number of Solutions	*Number of Backtracks*	*Nodes Expanded*	*Constraint Checks*
findSolutions				
4 Bar	9	0	50	2,796
7 Bar	76	6	488	29,135
12 Bar	2,746	20	17,282	818,282
findASolution				
4 Bar	1	0	8	1,736
7 Bar	1	0	14	7,726
12 Bar	1	0	24	15,782

Table 4.1: Empirical measures of *findSolutions* and *findASolution* when the graph is pre-processed with *makeArcConsistent*.

	Number of Solutions	*Number of Backtracks*	*Nodes Expanded*	*Constraint Checks*
findSolutions				
4 Bar	9	21	71	7,492
7 Bar	76	189	488	80,050
12 Bar	2,746	6,736	25,048	2,149,243
findASolution				
4 Bar	1	8	16	3,209
7 Bar	1	25	39	7,583
12 Bar	1	1,496	1,520	275,858

Table 4.2: Empirical measures of *findSolutions* and *findASolution*, without pre-processing.

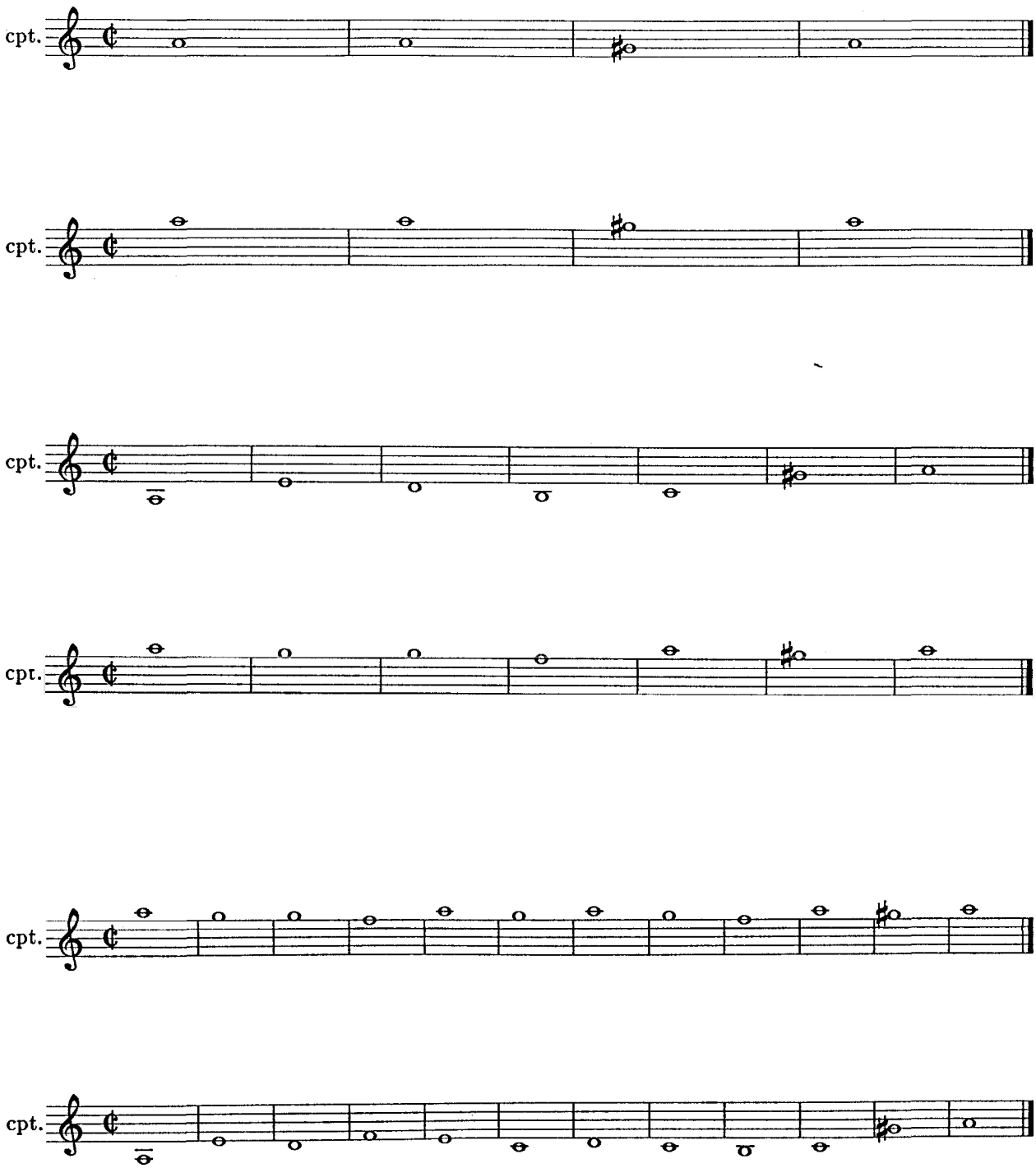


Figure 4.16: Sample counterpoints generated for each of the melodies.

correct sentences. Just as in a natural language, some of these correct sentences may be deemed more beautiful or pleasing than the others.

In a natural language, the syntax does not provide clues about the pleasantness or beauty of correct sentences: the beauty depends almost exclusively on the semantics. However, music (a closed, non-referential system) is devoid of a formal notion of semantics.[5] Thus even though music is more than just combinatorics of notes, a formal aesthetic analysis is necessarily restricted to a discourse on syntax. We thus undertake an analysis of the individual rules of counterpoint by means of measuring their respective effect of constraining the space of allowable compositions.

One metric of a constraint is its *satisfiability ratio* or *tightness* in isolation of any other constraint [Nadel88]. Nadel defines the satisfiability ratio of an n-ary constraint $c(x_1, x_2, \ldots, x_n)$ as the fraction of tuples of the cross product $D_1 \times D_2 \times \cdots \times D_n$ that are accepted by c. Search heuristics based on this metric are developed in [Nudel83]. However, satisfiability ratios are unsuitable for our analysis because we seek to examine the tightness of a *set* of constraints – not necessarily an individual constraint – that enforce a single rule.[6]

By formulating counterpoint as a CSP, it is not difficult, as we have shown, to efficiently generate every allowable counterpoint for a given melody. Enumerating all acceptable counterpoints is not only an interesting exercise, it provides insight into the power of the rules to individually and collectively reduce the solution space and thereby suggests the aesthetic merit of existing and potential constraints. Though a ludicrous notion for a performance, improvisation, or accompaniment system, the exhaustive search is useful for analysis [Ebcioglu86].

CSP-IC is ideal for such a study. Not only can it efficiently generate all solutions to a CSP, but it also permits the selective relaxation of the constraints that enforce the individual

[5]Though perhaps not semiotics nor emotion. See [Meyer56].

[6]Moreover, the satisfiability ratio is not necessarily an accurate reflection of a constraint's usefulness and importance precisely because it ignores the other problem constraints it must interact with in a given CSP. An empirical measure based on the constraint's role as one of many in a particular CSP would be more useful.

rules, thus enabling the empirical measurment of each rule's relative effectiveness. We say a constraint is *relaxed* when it is always *true*. Relaxing a rule requires either eliminating the procedural component, or changing the declarative component, of the constraints that enforce the rule (i.e., remove Constraint objects from the MCG or make the predicates always return *true*).

Using a fixed melody the number of additionally accepted counterpoints given the relaxation of the constraints comprising a single rule can be determined. We have invented the *importance ratio* (IR) of rule r for a particular CSP, which we define to be:

$$IR_{csp}(r) = \frac{y_r - z}{\sum_{k=1}^{p}(y_k - z)}$$

where y_r is the number of solutions to the CSP *csp* when all the constraints that make up rule r are relaxed, and z is the total number of solutions to *csp* satisfying all p rules that comprise the CSP.

The intuition behind the importance ratio is that the relaxation of the most important rule in a given CSP results in the largest increase in the number of solutions. A rule that is responsible for every reduction in the search space has an importance ratio of 1. A rule that is superfluous and thus has no effect on reducing the search space has an importance ratio of 0.

In Figure 4.3 the rules of first species counterpoint and some associated importance ratios are listed. We have given the IR for the four- and seven-bar problem instances, denoted by IR_4 and IR_7 respectively. The discrepencies in the IR measurements between the two problems can be attributed to the additional instances of certain constraints in the seven-bar example, particularly *harmonic*. Attaining the IR for the twelve-bar example is a highly intractable exercise given the proliferation of acceptable counterpoints when certain constraints are relaxed.

Interestingly, the *noThree* rule has no effect whatsoever on reducing the search space

Rule	IR_4 (Rank)	IR_7 (Rank)
mode	.0734 (5)	.1688 (2)
cadence	.1743 (3)	.0579 (5)
perfect	.2110 (2)	.0454 (7)
first	.2661 (1)	.0556 (6)
harmonic	.0367 (8)	.4710 (1)
melodic	.0734 (6)	.0879 (3)
skipStep	.1193 (4)	.0752 (4)
noThree	0 (9)	0 (10)
parallel	.0459 (7)	.0305 (8)
octave	0 (9)	.0076 (9)

Table 4.3: Some importance ratios for the counterpoint rules.

for the two sample problems. However, it is responsible for eliminating 340 potential counterpoints for the twelve-bar example and thus would have a non-zero IR for that problem instance.

Elimination of the unary rules (*mode* and *cadence*) does not, as one might expect, result in a drastic increase in the number of allowable counterpoints. The additional compositions, though free to modulate, are generally held in place by the higher-arity harmonic and melodic rules. Similarly, the quaternary constraints are mostly obviated by the interactions of the other rules.

The IRs, somewhat surprisingly, suggest that the vertical rules (*cadence, perfect, first,* and *harmonic*) are more important than the horizontal rules (all others). In both sample problems a vertical rule has the highest IR: *first* for IR_4 and *harmonic* for IR_7. Thus, despite the predominance of melodic considerations when writing counterpoint, the harmonic requirements are more efficient in defining the genre.[7]

Examining the set of acceptable counterpoints can suggest some useful new rules. By discovering the properties of the aesthetically poor counterpoints permitted by the current

[7]Only because the aesthetically important harmonic relationships are more readily definable in absolute terms, whereas the melodic component is necessarily a more abstract, less constrained, though no less important element of a counterpoint composition.

rule set, guidelines for the creation of new rules become evident. For example, the counterpoints listed in Figure 4.16 are generally poor because of their repeated use of the highest, or climax, note. This suggests that an appropriate rule to add would be one that prohibits the repetition of the highest note within some neighbourhood.

The usefulness of new rules can be tested and measured by merely adding clauses to existing constraints or defining new constraints altogether. For example, to test the effect of allowing melodic skips of a minor sixth only in an upward direction, the constraint *melodic* could be changed to a higher-order definition. Similarly, the implementation of a *gravity* rule whereby skips must be followed by a step in the opposite direction could be facilitated by a change to the *skipStep* constraint.

Chapter 5

Discussion

In this chapter the results of the thesis are discussed in relation to other competing approaches.

5.1 CSP-IC

In order to fairly evaluate the system we have built, it must first be properly classified. A good categorization of the tools commonly used to solve CSPs is given in [VanHentenryck89], which we paraphrase here. There are three usual approaches:

- recast the problem inside integer programming and solve using a standard algorithm (an example of this technique is in [VanHentenryck89]),
- use an AI problem-solver, like REF-ARF [Fikes70], or
- implement, in an imperative language, an ad-hoc search procedure.

To this list, we add the following:

- use a constraint programming language, like CHIP.

Van Hentenryck is critical of the first three approaches. Integer programming is seen as a computationally expensive procedure. AI problem-solvers he criticizes for not involving

the user in the problem-solving process. Because such systems are of the *black box* type, the user is prohibited from exploiting domain dependent knowledge that might make them work better for a particular problem. Finally, the ad-hoc approach is unsuitable because it requires considerable programming effort and results in programs that are difficult to modify and extend.

In contrast, Van Hentenryck sees CHIP as a viable alternative. Because of its declarative semantics, programs written in CHIP have reduced development times and are easy to modify and extend. Also, CHIP's ability to handle both numeric and symbolic constraints through consistency techniques, and its use of sophisticated labeling procedures, are seen as beneficial features not offered by any competing approach.

We now argue that CSP-IC does not suffer from the drawbacks of the first three approaches yet provides the aforementioned benefits of CHIP. Our system, a software-IC [Cox86] for the solution of CSPs, does not fit into the above categorization. CSP-IC is neither AI problem-solver, ad-hoc solution, nor constraint programming language, but it can be seen to embrace the best qualities of each.

5.1.1 CSP-IC is More Than an AI Problem-Solver

If viewed as an open architecture, our system addresses each of Van Hentenryck's criticisms of the AI problem-solvers. Allowing the user access to the implementation details of the system provides opportunity to improve performance for a given problem. In particular, the extent of constraint propagation (i.e., full, or degrees of partial, arc consistency) can be controlled by changing the manner in which *revise:using:* filters domains and propagates the constraints. Additional methods can be created to offer a choice ranging from pure backtracking (no consistency techniques) to full arc consistency.

The ability to specify how a constraint is to be used is advocated as an important feature of CHIP. Given the definition of various degrees of arc consistency through additional methods, this would also be possible in CSP-IC. The MCG could contain information as to

what procedure will enforce each particular constraint.

Alternatively, measures can be taken to give the user run-time control over the constraint propagation. The Csp class is not, strictly speaking, a necessary component of the CSP-IC. In certain applications it may be more meaningful simply to maintain consistency in a graph rather than find solutions to the CSP it models. User-directed constraint propagation and search hold interesting possibilities. Given an appropriate user-interface, the user could have run-time control over both the vertical and horizontal order of the search tree.

Further optimizations are possible. If the CSP of interest is comprised only of binary constraints, a new method could be defined in the Csp class to exploit this restriction e.g., *findSolutions* can be modified to only perform arc consistency over the appropriate subgraph at each level in the search tree (like Nadel's TSAC3).

5.1.2 CSP-IC is More Than an Ad-Hoc Implementation

The power to exploit problem features is often accompanied by inertia: the failure of many ad-hoc implementations is their inflexibility when it comes to solving other problem instances. Because they are not declarative, Van Hentenryck claims, imperative ad-hoc solutions are necessarily difficult to modify. This need not be the case. CSP-IC is not only reusable for any CSP, it is extensible to any datatype for the variables. This is accomplished through its design, which has isolated the aspects of CSP-solving (tree search and constraint propagation) that require the considerable programming effort and are most likely to be an impediment to change. The resulting framework is very close to the declarativeness of a constraint programming language. Implementing an imperative, ad-hoc solution does not in itself guarantee a non-extensible system: bad software engineering does.

5.1.3 CSP-IC is More Than a Constraint Programming Language

Though CSP-IC is not a constraint programming language *per se*, it exhibits many of the benefits of one while providing additional flexibility. Van Hentenryck argues that, because

CHIP has declarative semantics, program development time is reduced. Similarly, because the CSP-IC programmer need only concentrate on the representation of the CSP (i.e., the constraint graph), program development time is reduced. In fact, the constraint graph is often a more intuitive and natural formulation of a CSP than a CHIP program. A user of CHIP still needs a substantial logic programming skill-set in order to effectively state and solve problems with that language.

CSP-IC programs are also easily extended and modified. This became evident during the rule analysis of the previous chapter when, in order to obtain the data necessary to calculate the importance ratios, problem instances were continually redefined.

Moreover, CSP-IC can be used to solve CSPs that require both numeric and symbolic constraints. This is facilitated by its common representation scheme and consistency technique for both types of constraints. In CSP-IC, a single consistency technique (arc consistency) is sufficient for the development of sophisticated labeling procedures given the unified representation in the form of the MCG.

Interestingly enough, Van Hentenryck has wondered about the dubiousness of exploring constraint programming languages:

> Since logic programming has already proved useful for many other applications, we think that a specific language for combinatorial problems will not be of much interest [VanHentenryck89, p. 207].

Though it is true that a language limited solely to the solution of CSPs may not be of much interest, we find CSP-IC to be of great interest. Since CSP-IC is a software-IC, it can be embedded in a system where constraint propagation is seen as a useful computation mechanism. This is possible because of its separation of variable datatype from the system.

In particular, this freedom to define the domain objects and the constraints on these objects outside of the system itself is seen as an important attribute. Because the constraints are defined in the implementation language, they are computationally complete and not restricted by some notion of base- or higher-order constraints as they often are in constraint programming languages. As well, the domain objects are not forced into a single

interpretation that would restrict its applicability. The domain objects can even be other CSPs.[1]

5.1.4 CSP-IC is Not Perfect

Despite the many benefits of our design, CSP-IC has two major (though correctable) flaws. Firstly, recompilation is required if a new problem instance is to be solved. Changing the declarative part of a constraint requires the recompilation of the class that defines the Boolean method. Modifying any other component of a CSP requires recompiling the routine that constructs the modified constraint graph (usually *main.m*).

The other major drawback is the representation scheme. The MCG is an awkward and often confusing CSP representation, particularly when n-ary constraints, $n > 2$, are involved.

However, both drawbacks are surmountable. Incremental recompilation as featured in other object-oriented languages would provide a more flexible setting for the redefinition of problem instances. Therefore, implementing CSP-IC in such an OOP language would circumvent this inconvenience. Objective-C does not yet provide the type of programming environment conducive to rapid change.

Regarding the construction of the MCG, a potential solution is to create a separate program that generates the code to build the MCG based on a constraint graph as input. Since the structure of the MCG is completely determined by the constraint graph, this is a possibility. As currently implemented, no part of the constraint graph to MCG transformation is automated. An ideal environment would feature an interactive window-based program that allowed the user to graphically construct a constraint graph. The program would then generate the corresponding code to create the MCG.[2]

[1] The only structure imposed by CSP-IC is that domain objects respond to a *print* message by stating their value.

[2] This idea is in part inspired by NeXT's Interface Builder.

5.2 Composition as a CSP

5.2.1 The Intractability of Automated Composition

Chapter 4 of this thesis proposed that music composition can be viewed as a CSP. As an example, first species counterpoint was shown to be readily modeled as a CSP and using CSP-IC, counterpoint was generated. However, it is unlikely that we have discovered a practical method for automated composition. This was not our intention. Instead, the simple fact that composition has a natural formulation as a CSP does have some important ramifications for the computer music community: certain choices of control mechanism and representation will likely lead to programs with exponential time requirements.

For example, in one well-published project on automated composition [Ebcioglu84, Ebcioglu86, Ebcioglu88], a non-deterministic logic language, BSL, is used to harmonize chorales. A BSL program compiles into a backtracking procedure that attempts to find all satisfiable assignments to the existentially quantified variables (in this case, all four part harmonizations of a given melody that satisfy the specified constraints). The language does not incorporate consistency techniques, though it does utilize intelligent backtracking. Heuristics, in the form of BSL formulas, are in place to bias the search towards more musical solutions first. However, a single harmonization can require "15-30 minutes of VAX 11/780 CPU time" even despite the "efficient implementation that compiles the [BSL] into C" [Ebcioglu84, p. 84]. This is quite perplexing for Ebcioglu, particularly when he purposely avoided LISP and Prolog since

> ...the inefficiency of these languages has a tendency to limit their domain of applicability to computationally small problems, whereas the problem of generating non-trivial music appears to require gigantic computational resources... [Ebcioglu86, p. 784].

This statement incorrectly labels LISP and Prolog inefficient rather than blaming the intractable problems they are often used to solve. Ebcioglu's real problem is a failure to (at least explicitly) recognize the intractability of the problem he is attempting to solve. After

all, BSL is merely a language that solves CSPs almost as fast as a backtracking C program can; that is to say, not very fast for certain problem instances. Perhaps if he incorporated consistency techniques the exhibited exponential explosion would be diffused (at least to a degree).

Given infinite time, Ebcioglu's program would find a harmonization identical to that produced by Bach, providing his rule-base is an accurate cognitive model of Bach chorales. By examining the complete set of generated harmonizations and undertaking an importance ratio analysis, it might be possible to more tightly define his rule-base. Biasing the search towards the most musical solutions first is a good idea, but won't help to reduce the amount of backtracking.

5.2.2 Consistency Techniques Can Help

Short of a major breakthrough in NP-complete problem-solving, rule-based chronologically backtracking systems are likely to continue dominating the landscape of automated composition systems that seek to harmonize in accordance with some knowledge base (declarative or otherwise). Consistency techniques in the form of constraint propagation are an effective means of squelching the inevitable exponential explosion when these systems are applied to non-trivial tasks.

An excellent illustration comes from our counterpoint implementation. A backtracking procedure to produce counterpoint for a given melody, without consistency techniques, faces a search tree with a branching factor of at least 25: the pitch for each note of the counterpoint can come from the notes within the octave above and below the corresponding note of the melody. If, however, the search space is first subjected to a full arc consistency algorithm, a much smaller search tree results. Figure 5.1 is the twelve-bar melody of the previous chapter and its resulting counterpoint search space after *makeArcConsistent* is applied to the constraint graph. The resulting tree has a branching factor of ≈ 5.42. The trade-off is very small: only 12,181 consistency checks are needed to enact this dramatic reduction in

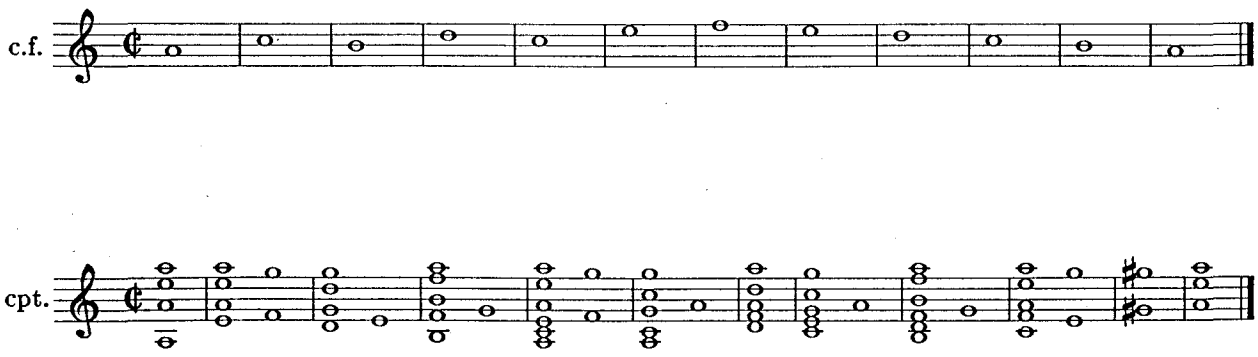


Figure 5.1: A given melody (c.f.) and its corresponding counterpoint search space (cpt.) as generated by pre-processing the constraint graph with *makeArcConsistent*

the search space.

It is not unreasonable to expect that even without consistency techniques a backtracking procedure would find one solution without serious time requirements. However, one could not predict the same for much larger tasks, say even four-part florid counterpoint. A rule analysis, like the one undertaken in the previous chapter, that requires the generation of all acceptable compositions would likely become genuinely intractable without consistency techniques.

Nadel has suggested that the optimal search procedure for underconstrained CSPs is Forward Checking [Nadel88]. For the task of generating first species counterpoint, our test results do not concur. Table 5.1 gives empirical measurements for four search algorithms given the task of generating counterpoint for the twelve-bar melody of Figure 5.1. For comparison, two new procedures, *BT* and *FC*, were implemented with CSP-IC by changing the functionality of the *revise:using:* method. The two procedures correspond respectively to straightforward backtracking with side-effect free forward checking (*revise:using:* neither filters domains nor propagates change) and Forward Checking with redundant checks (*revise:using:* filters domains but does not propagate change). These two partial arc consistency algorithms are compared with the full arc consistency algorithms *findSolutions* and *findASolution* with and without (*w/o*) *makeArcConsistent* used as a pre-processor.

	Number of Backtracks	*Nodes Expanded*	*Constraint Checks*
Report all solutions:			
BT	11,671	30,131	1,972,048
FC	11,086	29,546	1,040,626
findSolutions	20	17,282	818,282
findSolutions w/o	6,736	25,048	2,149,243
Report one solution:			
BT	2,025	2,049	223,875
FC	2,002	2,026	111,431
findASolution	0	24	15,782
findASolution w/o	1,496	1,520	275,858

Table 5.1: Comparison of four search algorithms for generating counterpoint.

As seen from the results in this table, for this task the most important consistency technique is the application of arc consistency as a pre-processor.

Chapter 6

Conclusion

In this chapter the accomplishments of the thesis are summarized and some suggestions for further research are given.

6.1 Summary of Results

In this thesis we have undertaken an examination of two ideas: the notion that object-oriented programming can be used to build a software-IC for the solution of CSPs, and that music composition has a natural formulation as a CSP.

Like many of the good ideas from artificial intelligence (e.g., rule-based production systems, resolution theorem proving), constraint propagation has emerged as a useful computation mechanism, as realized by the recent proliferation of constraint programming languages. However, to date it seems that constraint propagation is a better idea than any of these existing implementations, which too often force a restricted domain of application upon the user.

CSP-IC, as a tool for solving CSPs, is a useful alternative to existing constraint programming languages and AI problem-solvers. By defining a methodology for the conversion of an abstract model of a CSP, namely the constraint graph, into a network of co-operating objects, we have isolated some useful abstractions of constraint programming. This system

can solve CSPs involving constraints of any arity, and frees the programmer from the details of tree-search and constraint propagation.

We have also shown how music composition can be readily viewed as a CSP. This provides insight into the intractability of many approaches advocated for intelligent composition systems. The CSP model is seen as beneficial because of its exploitation of consistency techniques, which were shown to drastically reduce the search space of certain composition problems. We do not necessarily advocate the CSP model, or for that matter the rule-based approach in general, as an effective approach to automated composition. We do, however, see it as an improvement over standard backtracking methods.

The synthesis of these two contributions was the generation of contrapuntal music by modeling first species counterpoint as a CSP and implementing its solution in our constraint satisfaction system. Using this approach, we undertook an analysis of the rules of first species counterpoint by measuring their individual effect on constraining the number of compositions belonging to the genre.

6.2 Future Research and Extensions

There are many possible paths to follow in future research for both CSP-IC and music composition as a constraint satisfaction problem. The major extensions include, but are not limited to, the following.

Because it is a software-IC, CSP-IC is seen in our eyes to be a potentially convenient way to incorporate constraint propagation in existing systems. An interesting project would be to embed CSP-IC in an object-oriented expert system shell. This would provide additional insight into the appropriateness of its design, as well as hopefully suggest an alternative representation scheme. There may be a more natural mapping from constraint graph, which we see as a good representation for CSPs, to object-oriented program. Through the process of embedding CSP-IC in an existing system, a better representation might become manifest.

In particular, the development of an automated method for constructing the MCG would improve the system's ease of use.

With regards to the modeling of composition as a CSP, we would like to see an incorporation of consistency techniques in Ebcioglu's work. However, since Ebcioglu has already invested a significant amount of time in creating a substantial knowledge base in first-order logic, CSP-IC is not necessarily the best platform to do so. CHIP, as a logic language, is a more suitable vehicle for testing the hypothesis that Ebcioglu's system could expect a significant reduction in time requirements given the utilization of consistency techniques.

We note two additional future investigations of the CSP paradigm for composition. First, by extending the counterpoint example to model even more complex composition styles, we would gain a fuller understanding of the appropriateness of viewing composition strictly as a CSP. Others have expressed reservations regarding a constraint-only approach [Ebcioglu84], but we see possibilities, as does Marvin Minsky:

> If you really had to think of all the things, all the multiple constraints that were satisfied in a piece, you could never do it. But if you can just activate a lot of experts that don't have to communicate very much but just send constraints among one another saying, "Whatever you do don't put the –" [Minsky80].

Perhaps rather than the imitation of an historical style, contemporary music experimentation would be a more interesting application of the CSP paradigm.[1]

The second avenue of potential interest lies in the field of music education. A constraint graph approach coupled with consistency techniques seems particularly suited for the educational environment where it is more important that the computer prevent mistakes than generate compositions. Because consistency is maintained as each variable is instantiated, the student can make a choice for a note value and see the ramifications in the form of the resulting constraint propagation (e.g., "...if I choose a fifth for the interval in this bar, I see that it rules out the A in the previous bar..."). This is possible with CSP-IC if the Csp class is bypassed, instead letting the student guide the search and choose note values.

[1] Hiller and Isaacson similarly suggested this point of departure for their counterpoint system [Hiller59].

Explication of domain eliminations is a possible feature of such a system by empowering the Constraint objects with the ability to explain their actions.

Appendix A

The CSP Software-IC

The Objective-C class implementations listed in this appendix comprise the object-oriented constraint satisfaction system described in Chapter 3. Taken together, these classes provide a software-IC for the solution of CSPs representable by a constraint hypergraph.

Source listings are presented for each of the classes *Link, Variable, Constraint, Constraint3* (as an example of a higher-arity REVISE procedure), and *Csp*.

A.1 The Class Link

A.1.1 Link.h

```
#import <objc/Object.h>

@interface Link: Object
{
   id   node;
   SEL  label;
}

- getNode;
- (SEL) getLabel;
- setNode: aConstraintObject;
- setLabel: (SEL) aConstraintMethod;

@end
```

A.1.2 Link.m

```
#import "Link.h"

@implementation Link

- getNode
{
   return (node);
}

- (SEL) getLabel
{
   return (label);
}

- setNode: anObject
{
   node = anObject;
   return (self);
}

- setLabel: (SEL) aSEL
{
   label = aSEL;
   return (self);
}

@end
```

A.2 The Class Variable

A.2.1 Variable.h

```
#import <objc/Object.h>
#import <objc/List.h>

@interface Variable:Object
{
    List *domain;
```

```
    List *neighbours;
    STR         name;

    // for archiving... (this should be a private variable)

    List    *backupDomains;
}

- addToDomain: anObject;
- setDomain: newDomain;
- getDomain;
- printDomain;

- addToNeighbours: anObject;

- (BOOL) alertChanged;
- (BOOL) instantiate: anObject;

- save;
- restore;

- free;

+ new:(STR)varName;

@end
```

A.2.2 Variable.m

```
#import <stdio.h>
#import <objc/hashtable.h>          // for NXCopyStringBuffer
#import "Variable.h"
#import "Link.h"
#import "Constraint.h"

@implementation Variable

+ new:(STR)varName
{
   self = [super new];
   neighbours = [List new];
   domain = [List new];
```

```
    name = NXCopyStringBuffer(varName);
    backupDomains = [List new];
    return (self);
}

// Blow away the object and its instance variables,
// but not the objects in the domain Lists.

- free
{
    [neighbours freeObjects];
    [neighbours free];
    [backupDomains free];
    [domain free];
    [super free];
}

- addToDomain: anObject
{
    [domain addObject: anObject];
}

- addToNeighbours: anObject
{
    [neighbours addObject: anObject];
}

- setDomain: newDomain
{
     domain = newDomain;
}

- getDomain
{
     return (domain);
}

// notify all its neighbours in the graph that its domain has been changed
// return TRUE if the changes did not result in an empty set

- (BOOL) alertChanged
{
     unsigned i;
```

```
    BOOL     result;
    Link     *temp;

    i = 0;
    result = YES;

    while (result && i < [neighbours count])
      {
        temp = [neighbours objectAt:i];
        result = [[temp getNode] revise:self using:[temp getConstraint]];
        i++;
       }

    return (result);
 }

// print the value of each member of domain

- printDomain
{
  int   i;

  printf ("%s: ",name);
  [domain makeObjectsPerform: @selector(print)];
  printf ("; ");
}

// temporarily set the domain to be anObject and check for
// consistency; if not okay, change domain back

- (BOOL) instantiate: anObject
{
     List   *temp_domain;

     temp_domain = [domain copy];
     [domain empty];
     [domain addObject: anObject];

     if ([self alertChanged])
        {
         [temp_domain free];
         return (YES);
        }
```

```
    else          /* inconsistent */
       {
        [domain free];
        domain = temp_domain;
        return (NO);
       }
}

// The next two methods are needed for backup searches

- save
{
   List   *backupDomain;

   backupDomain = [domain copy];
   [backupDomains addObject: backupDomain];
}

- restore
{
   [domain free];
   domain = [backupDomains removeLastObject];
}

@end
```

A.3 The Class Constraint

A.3.1 Constraint.h

```
#import <objc/Object.h>
#import "Variable.h"

@interface Constraint:Object
{
    Variable *variable;
}

- setVariable:aVariable;
- (BOOL) revise:sender using:(SEL)constraint;
```

```
@end
```

A.3.2 Constraint.m

```
#import "Constraint.h"

@implementation Constraint

// initialize the outlet variable  ie. who are we constraining?

- setVariable:aVariable
{
    variable = aVariable;
    return self;
}

// Mackworth's REVISE - filter members of variable wrt the sender
// using the method 'constraint'.  This is done additively because of
// the use of the List class to model the domains.
// Return NO if changes prove to be inconsistent ie. a domain reduced to {}

- (BOOL) revise:sender using:(SEL)constraint
{
   List       *domain1, *domain2, *newDomain;
   unsigned   i, j;
   BOOL       satisfied, changed;
   id         memberDomain1;

   domain1 = [variable getDomain];
   domain2 = [sender getDomain];
   newDomain = [List new];
   changed = NO;

   // Check that each member of domain1 is satisfied by at least one member
   // of domain2.  If so, add it to the newDomain.

   for (i = 0; i < [domain1 count]; i++)
     {
     memberDomain1 = [domain1 objectAt: i];
     satisfied = NO;

     j = 0;
```

```
    do
     {
      if ( [[domain2 objectAt: j] perform:constraint with:memberDomain1] )
 satisfied = YES;
      j++;
     }
    while ((! satisfied) && j < [domain2 count]);

    if (! satisfied)
changed = YES;
    else
       [newDomain addObject: memberDomain1];
    }

 // Decide the outcome of the revision...

 if ( [newDomain count] == 0 )   // domain now {}! Don't change.
    {
     [newDomain free];
     return (NO);
    }

 if (changed)
     {
      [variable setDomain: newDomain];
      if ([variable alertChanged])    // propagate!
          return (YES);
      else
         {
          [newDomain free];
          [variable setDomain: domain1];
          return (NO);     // inconsistent propagation
         }
     }

  // no change to domain occurred at all
  [newDomain free];
  return (YES);        // everything is still consistent
}
@end
```

A.4 The Class Constraint3

A.4.1 Constraint3.h

```
#import "Constraint.h"
#import "Variable.h"

@interface Constraint3:Constraint
{
    Variable *variable2;
}

- setVariable2:aVariable;

@end
```

A.4.2 Constraint3.m

```
// For enforcement of ternary constraints.

#import "Constraint3.h"

@implementation Constraint3

// initialize the outlet variable2 ie. who are we constraining?

- setVariable2:aVariable
{
    variable2 = aVariable;
    return self;
}

// Mackworth's REVISE, but for ternary constraints.
// Filter members of variable and variable2 wrt the sender
// using the method 'constraint'.
// Over-rides the definition in the superclass.

- (BOOL) revise:sender using:(SEL)constraint
{
   List       *domain1, *domain2, *domain3,
              *newDomain2, *newDomain3;
```

```
unsigned   i, j, k;
BOOL       satisfied, changed;
id         memberDomain2, memberDomain3;

domain1 = [sender getDomain];
domain2 = [variable getDomain];
domain3 = [variable2 getDomain];
newDomain2 = [List new];
newDomain3 = [List new];
changed = NO;

// Check that each member of domain2 is
// satisfied by at least one member of domain3 and
// domain1.  If so, add it to the newDomain2.

for (i = 0; i < [domain2 count]; i++)
  {
  memberDomain2 = [domain2 objectAt: i];
  satisfied = NO;

  j = 0;
  do
    {
     memberDomain3 = [domain3 objectAt: j];

     k = 0;
     do
      {
       if ( [[domain1 objectAt: k] perform:constraint
             with:memberDomain2 with:memberDomain3] )
  satisfied = YES;
       k++;
      }
     while ((! satisfied) && k < [domain1 count]);
     j++;
    }
  while ((! satisfied) && j < [domain3 count]);

  if (! satisfied)
     changed = YES;
  else
      [newDomain2 addObject: memberDomain2];
  }
```

```
if ([newDomain2 count] == 0)
   {
   // domain now {}! Don't change.
   [newDomain2 free];
   [newDomain3 free];
   return (NO);
   }

// Check that each member of domain3 is
// satisfied by at least one member of domain2 and
// domain1.  If so, add it to the newDomain3.

for (i = 0; i < [domain3 count]; i++)
  {
  memberDomain3 = [domain3 objectAt: i];
  satisfied = NO;

  j = 0;
  do
    {
     memberDomain2 = [newDomain2 objectAt: j];

     k = 0;
     do
      {
       if ( [[domain1 objectAt: k] perform:constraint
             with:memberDomain2 with:memberDomain3] )
  satisfied = YES;
       k++;
      }
     while ((! satisfied) && k < [domain1 count]);
     j++;
    }
  while ((! satisfied) && j < [newDomain2 count]);

   if (! satisfied)
     changed = YES;
   else
      [newDomain3 addObject: memberDomain3];
  }

// Decide the outcome of the revision...
```

```
  if ( [newDomain3 count] == 0 )
     {
      // domain now {}! Don't change.
      [newDomain2 free];
      [newDomain3 free];
      return (NO);
     }

  if (changed)
      {
       [variable setDomain: newDomain2];
       [variable2 setDomain: newDomain3];
       if ([variable alertChanged] && [variable2 alertChanged] ) // propagate!
           return (YES);
       else
          {
           [newDomain2 free];
           [newDomain3 free];
           [variable setDomain: domain2];
           [variable2 setDomain: domain3];
           return (NO);     // inconsistent propagation
          }
      }

    // no change to domain occurred at all
    [newDomain2 free];
    [newDomain3 free];
    return (YES);        // everything is still consistent
}

@end
```

A.5 The Class Csp

A.5.1 Csp.h

```
#import <objc/Object.h>
#import <objc/List.h>

@interface Csp:Object
```

```
{
    List *variables;
    long bts,
sol,
nodes;
}

- addToVariables: anObject;
- findSolutions;
- findASolution;
- (BOOL) makeArcConsistent;
- free;

+ new;

// Metrics for search efficiency

- (long) getBts;
- (long) getSol;
- (long) getNodes;

@end
```

A.5.2 Csp.m

```
#import <stdio.h>
#import "Csp.h"
#import "Variable.h"

@implementation Csp

+ new
{
   self = [super new];
   variables = [List new];
   bts = nodes = sol = 0;
   return (self);
}

- free
{
   [variables makeObjectsPerform:@selector(free)];
```

```
    [super free];
}

- addToVariables: anObject
{
    [variables addObject: anObject];
}

- findSolutions2: (int) index for: (int) numVariables
{
    Variable        *aVariable;
    List            *aVariablesDomain;
    int    i,j;
    BOOL    success;

    // base case - at end of list of variables
    if (index == numVariables)
      {
       sol++;
       [variables makeObjectsPerform: @selector(printDomain)];
       printf ("\n");
       return;
      }

    // select a variable from the list so we can instantiate it
    aVariable = [variables objectAt:index];
    aVariablesDomain = [[aVariable getDomain] copy];
    index++;

    // now try and instantiate aVariable
    success = NO;
    for (i = 0; i < [aVariablesDomain count]; i++)
      {
       // first take a picture of the world
       for (j = index; j < numVariables; j++)
         [[variables objectAt:j] save];

       // instantiate
       if ([aVariable instantiate: [aVariablesDomain objectAt:i]])
           // recursively instantiate the future variables
           {
           success = YES;
           nodes++;
```

```
            [self findSolutions2: index for: numVariables];
            }

        // now unarchive each future variable for the next instantiation
        for (j = index; j < numVariables; j++)
            [[variables objectAt:j] restore];
      }

    if (! success)
       bts++;

    [aVariable setDomain: aVariablesDomain];
}

- findSolutions
{
  [self findSolutions2: 0 for: [variables count]];
}

- (BOOL) findOneSolution: (int)index for: (int)numVariables
{
    Variable        *aVariable;
    List            *aVariablesDomain;
    int    i,j;
    BOOL        success;

    // base case - at end of list of variables
    if (index == numVariables)
      {
       sol++;
       [variables makeObjectsPerform: @selector(printDomain)];
       printf ("Solution\n");
       return (YES);
      }

    // select a variable from the list so we can instantiate it
    aVariable = [variables objectAt:index];
    aVariablesDomain = [[aVariable getDomain] copy];
    index++;

    // now try and instantiate aVariable
    success = NO;
    for (i = 0; i < [aVariablesDomain count]; i++)
```

```
    {
     // first take a picture of the world
     for (j = index; j < numVariables; j++)
       [[variables objectAt:j] save];

     // instantiate
     if ([aVariable instantiate: [aVariablesDomain objectAt:i]])
         // recursively instantiate the future variables
         {
         nodes++;
         success = [self findOneSolution: index for: numVariables];
         }

     // now unarchive each future variable for the next instantiation
     for (j = index; j < numVariables; j++)
         [[variables objectAt:j] restore];

     if (success)
        break;
    }

  [aVariable setDomain: aVariablesDomain];

  if (! success)
    {
     bts++;
     return (NO);
    }
}

- findASolution
{
  [self findOneSolution: 0 for: [variables count]];
}

- (long) getSol
{
   return (sol);
}

- (long) getBts
{
   return (bts);
```

```
}

- (long) getNodes
{
   return (nodes);
}

- (BOOL) makeArcConsistent
{
   int   i;

   for (i = 0; i < [variables count]; i++)
     if (! [[variables objectAt:i] alertChanged] )
        return (NO);

   return (YES);
}

@end
```

Appendix B

A Solution to the n-Queens Problem

This appendix contains the source code as an addition to CSP-IC needed to solve the classic n-queens problem, $n \leq 10$. A class for the type of the domain objects is first introduced followed by the *main* routine needed to construct the MCG.

B.1 The Class Square

The domain objects, squares on a chess board, are encapsulated in the class *Square*. Objects of class Square have a *row* and a *col* (column) defined as instance variables. It is in this class that the constraint *noAttack:* is defined, which reports whether two squares are in the same column or diagonal.

In order that solutions may be reported, the objects comprising the domains of the variables must respond to the message *print*. In the class Square, only the column number is printed in response to this message.

B.1.1 Square.h

```
#import <objc/Object.h>
```

```
@interface Square:Object

{
   int  row,
        col;
}

- (int) getRow;
- (int) getCol;
- setSquare:(int)row andCol:(int)col;
- print;

+ (long) getCount;

// The constraints

- (BOOL) noAttack: other;

@end
```

B.1.2 Square.m

```
#import <stdio.h>
#import <math.h>
#import "Square.h"

static long  count = 0;   // for counting constraint checks

@implementation Square

- (int) getRow
{
   return (row);
}

- (int) getCol
{
   return (col);
}

- setSquare:(int)aRow andCol:(int) aCol
{
```

```
    row = aRow;
    col = aCol;
    return (self);
}

- print
{
    printf(" %d",col);
}

+ (long) getCount
{
    return (count);
}

// constraint

- (BOOL) noAttack: other
{
    int   row2, col2;

    count++;

    row2 = [other getRow];
    col2 = [other getCol];

    return ((col != col2) && (abs(row - row2) != abs(col - col2)));
}

@end
```

B.2 The *main* Routine

The *main.m* file builds the MCG, and, by sending the message *findSolutions* to the instance of class Csp, reports solutions to the problem the graph represents. The size of the chess board is given as a parameter to the program. As in most representations for the n-queens as a CSP, the n variables are each assigned to a different row, thus reducing the complexity of the problem. Comments throughout the program guide the construction of the MCG.

B.2.1 main.m

```
#import <stdlib.h>

// Import the CSP-IC...
#import "CSP/Variable.h"
#import "CSP/Constraint.h"
#import "CSP/Link.h"
#import "CSP/Csp.h"

// The domain objects' class...
#import "Square.h"

#define MAX_QUEENS  10

char *queenNames[MAX_QUEENS] = {"Q0","Q1","Q2","Q3","Q4",
                                "Q5","Q6","Q7","Q8","Q9"};

// Used to decipher the run-time argument...
int stringToInt(s)
   char   *s;
{
   int i, integerValue, result;

   result = 0;

   for (i = 0; s[i] != '\0'; i++)
     {
      integerValue = s[i] - '0';
      result = result * 10 + integerValue;
     }

   return (result);
}

void main(int argc, char *argv[])
{
   Variable     *Q[MAX_QUEENS];
   Constraint   *cQ[MAX_QUEENS];
   Square       *square;
   Link         *L;
   Csp         *csp;
```

```
int          i, j, num_queens;

num_queens = stringToInt(argv[1]);

// Create a problem instance...
csp = [Csp new];

// For each queen:
//   create a Variable object
//   add it to the problem instance 'csp'
//   create a binary Constraint object
//   point the Constraint object at the Variable object
for (i = 0; i < num_queens; i++)
  {
   Q[i] = [Variable new:queenNames[i]];
   [csp addToVariables: Q[i]];
   cQ[i] = [Constraint new];
   [cQ[i] setVariable:Q[i]];
  }

// Initialize the domain of each Variable object, and
// add one link for every other queen in the problem.
for (i = 0; i < num_queens; i++)
  for (j = 0; j < num_queens; j++)
     {
     square = [[Square new] setSquare: i+1 andCol: j+1];
     [Q[i] addToDomain: square];
     if (i != j)
        {
        L = [Link new];
        [L setNode: cQ[j]];
        [L setConstraint: @selector(noAttack:)];
        [Q[i] addToNeighbours: L];
        }
     }
// Arc consistency is ineffectual before a variable is instantiated,
// so no need to [csp makeArcConsistent];

// Find all solutions and report statistics.
[csp findSolutions];

printf ("Number solutions: %ld\n",[csp getSol]);
printf ("Number constraint checks: %ld\n",[Square getCount]);
```

```
    printf ("Number backtrack points: %ld\n",[csp getBts]);
    printf ("Number nodes expanded: %ld\n",[csp getNodes]);

    // Clean up...
    [csp free];
    exit(0);
}
```

B.3 Sample Run

```
%n_queens 6
Q5:  5; Q4:  3; Q3:  1; Q2:  6; Q1:  4; Q0:  2;
Q5:  4; Q4:  1; Q3:  5; Q2:  2; Q1:  6; Q0:  3;
Q5:  3; Q4:  6; Q3:  2; Q2:  5; Q1:  1; Q0:  4;
Q5:  2; Q4:  4; Q3:  6; Q2:  1; Q1:  3; Q0:  5;
Number solutions: 4
Number constraint checks: 3104
Number backtrack points: 2
Number nodes expanded: 26
%
```

Appendix C

The Counterpoint Application

Generating counterpoint with CSP-IC requires the definition of a Note class (where the constraints are defined) and the construction of a MCG (done in *main.m*) to reflect the problem.

C.1 The Note Class

A Note object is merely an integer in the range 1 to 127 in accordance with the pitch representation in the MIDI standard.

C.1.1 Note.h

```
#import <objc/Object.h>

@interface Note:Object

{
   int  pitch;
}

- (int) getPitch;
- setPitch:(int)pitch;
- print;
+ (long) getChecks;
```

```
// Some constraints

- (BOOL) harmonic: other;
- (BOOL) perfectHarmonic: other;
- (BOOL) perfectCfHarmonic: other;
- (BOOL) melodic: other;

- (BOOL) skip: other step: other2;
- (BOOL) skipped: other step: other2;
- (BOOL) step: other skip: other2;

- (BOOL) noParallel: x2 To: x3 Perfect: x4;
- (BOOL) noPerfect: x2 From: x3 Parallel: x4;
@end
```

C.1.2 Note.m

```
#import <stdio.h>
#import <math.h>
#import "Note.h"

// Constants (musical intervals).

#define unison 0
#define minor2nd 1
#define major2nd 2
#define minor3rd 3
#define major3rd 4
#define fourth 5
#define tritone 6
#define fifth 7
#define minor6th 8
#define major6th 9
#define octave 12
#define minor10th 15
#define major10th 16
#define twelfth 19

static long checks = 0;

@implementation Note
```

```
- (int) getPitch
{
   return (pitch);
}

- setPitch:(int)aPitch
{
   pitch = aPitch;
   return (self);
}

- print
{
   printf(" %d",pitch);
}

// constraints...

- (BOOL) harmonic: other
{
   int   interval;

   checks++;
   interval = abs(pitch - [other getPitch]);

   return ((interval == minor3rd) ||
           (interval == major3rd) ||
           (interval == fifth) ||
           (interval == minor6th) ||
           (interval == major6th) ||
           (interval == octave) ||
           (interval == minor10th) ||
           (interval == major10th) ||
           (interval == twelfth));
}

- (BOOL) perfectHarmonic: other
{
   int   interval;

   checks++;
   interval = abs(pitch - [other getPitch]);
```

```
    return ((interval == unison) ||
            (interval == fifth) ||
            (interval == octave) ||
            (interval == twelfth));
}

- (BOOL) perfectCfHarmonic: other
{
    int   interval;
    checks++;

    interval = abs(pitch - [other getPitch]);
    if (pitch > [other getPitch])
        return ((interval == unison) || (interval == octave));
    else return ((interval == unison) ||
            (interval == fifth) ||
            (interval == octave) ||
            (interval == twelfth));
}

- (BOOL) melodic: other
{
    int   interval;

    checks++;
    interval = abs(pitch - [other getPitch]);

    return ((interval <= minor6th) && (interval != tritone));
}

// ternary...

- (BOOL) noThree:other InRow:other2
{
    checks++;
    return ((pitch != [other getPitch]) || (pitch != [other2 getPitch]));
}

- (BOOL) skip:other step:other2
{
    int   interval;

    checks++;
```

```
   interval = abs(pitch - [other getPitch]);

   if (interval >= minor3rd)
      return ((abs([other getPitch] - [other2 getPitch]) <= major2nd) &&
               [self noThree: other InRow: other2]);
   else return ([self noThree: other InRow: other2]);
}

- (BOOL) skipped:other step:other2
{
   int   interval;

   checks++;
   interval = abs(pitch - [other getPitch]);

   if (interval >= minor3rd)
      return ((abs(pitch - [other2 getPitch]) <= major2nd) &&
               [self noThree: other InRow: other2]);
   else return ([self noThree: other InRow: other2]);
}

- (BOOL) step:other skip:other2
{
   int   interval;

   checks++;
   interval = abs(pitch - [other2 getPitch]);

   if (interval <= major2nd)
      return ([self noThree: other InRow: other2]);
   else
      return ((abs([other getPitch] - [other2 getPitch]) < minor3rd) &&
               [self noThree: other InRow: other2]);
}

// quaternary...

static BOOL perfect (x1, x2)
   int   x1, x2;
{
   int   interval = abs(x1 - x2);

   return ((interval == unison) || (interval == fifth) ||
```

```
          (interval == octave) || (interval == twelfth));
}

- (BOOL) noParallel: x2 To: x3 Perfect: x4
{
   // avoid division by zero
   checks +=2;
   if ([x2 getPitch] == [x4 getPitch])    // no motion
      return (YES);

   // octave constraint:
   if ( (abs([x4 getPitch] - [x3 getPitch]) == octave) &&
        ( (abs([x4 getPitch] - [x2 getPitch]) > major2nd) ||
          (abs(pitch - [x3 getPitch]) > major2nd) ))
      return (NO);

   if ( perfect([x4 getPitch], [x3 getPitch]) )
      return ((float)((pitch - [x3 getPitch])/
              ([x2 getPitch] - [x4 getPitch])) <= 0.0);
   else return (YES);
}

- (BOOL) noPerfect: x2 From: x3 Parallel: x4
{
   // avoid division by zero
   checks +=2;
   if (pitch == [x2 getPitch])    // no motion
     return (YES);

   // octave constraint:
   if ( (abs([x4 getPitch] - pitch) == octave) &&
        ( (abs([x4 getPitch] - [x3 getPitch]) > major2nd) ||
          (abs(pitch - [x2 getPitch]) > major2nd) ))
      return (NO);

   if ( perfect([x4 getPitch], pitch) )
      return ((float)(([x4 getPitch] - [x3 getPitch])/
                      (pitch - [x2 getPitch])) <= 0.0);
   else return (YES);
}

+ (long) getChecks
{
```

```
   return(checks);
}

@end
```

C.2 main.m

```
#import <stdlib.h>
#import <stdio.h>
#import <math.h>
#import "CSP/Variable.h"
#import "CSP/Constraint4.h"
#import "CSP/Link.h"
#import "CSP/Csp.h"

#import "Note.h"

#define NUM_BARS 4

char *cp_names[12] = {"cp1", "cp2", "cp3", "cp4", "cp5", "cp6", "cp7",
                      "cp8", "cp9", "cp10", "cp11", "cp12"};
char *cf_names[12] = {"cf1", "cf2", "cf3", "cf4", "cf5", "cf6", "cf7",
                      "cf8", "cf9", "cf10", "cf11", "cf12"};

void main(int argc, char *argv[])
{
   Variable     *cp[NUM_BARS],
                *cf[NUM_BARS];
   Constraint   *binary[NUM_BARS];
   Constraint3  *ternary;
   Constraint4  *quat;
   Note *note;
   Link       *L;
   Csp       *csp;
   int        i, j;

   csp = [Csp new];

   for (i = 0; i < NUM_BARS; i++)
     {
      cp[i] = [Variable new:cp_names[i]];
```

```
      [csp addToVariables: cp[i]];
      binary[i] = [Constraint new];
      [binary[i] setVariable:cp[i]];

      cf[i] = [Variable new:cf_names[i]];
      [csp addToVariables: cf[i]];
     }

  // the cantus firmus...
  note = [[Note new] setPitch: 57];
  [cf[0] addToDomain: note];
  note = [[Note new] setPitch: 60];
  [cf[1] addToDomain: note];
  note = [[Note new] setPitch: 59];
  [cf[2] addToDomain: note];
  note = [[Note new] setPitch: 57];
  [cf[3] addToDomain: note];

  // the counterpoint...
 for (i = 0; i < NUM_BARS; i++)
   {
    if (i != NUM_BARS -2)
       {
        note = [[Note new] setPitch: 45];
        [cp[i] addToDomain: note];
        note = [[Note new] setPitch: 47];
        [cp[i] addToDomain: note];
        note = [[Note new] setPitch: 48];
        [cp[i] addToDomain: note];
        note = [[Note new] setPitch: 50];
        [cp[i] addToDomain: note];
        note = [[Note new] setPitch: 52];
        [cp[i] addToDomain: note];
        note = [[Note new] setPitch: 53];
        [cp[i] addToDomain: note];
        note = [[Note new] setPitch: 55];
        [cp[i] addToDomain: note];
        note = [[Note new] setPitch: 57];
        [cp[i] addToDomain: note];
        note = [[Note new] setPitch: 59];
        [cp[i] addToDomain: note];
        note = [[Note new] setPitch: 60];
        [cp[i] addToDomain: note];
```

```
        note = [[Note new] setPitch: 62];
        [cp[i] addToDomain: note];
        note = [[Note new] setPitch: 64];
        [cp[i] addToDomain: note];
        note = [[Note new] setPitch: 65];
        [cp[i] addToDomain: note];
        note = [[Note new] setPitch: 67];
        [cp[i] addToDomain: note];
        note = [[Note new] setPitch: 69];
        [cp[i] addToDomain: note];
       }
    else
       {
       note = [[Note new] setPitch: 56];
       [cp[i] addToDomain: note];
       note = [[Note new] setPitch: 68];
       [cp[i] addToDomain: note];
       }
  }

  // binary constraints...

  for (i = 1; i < NUM_BARS; i++)
    {
     L = [Link new];
     [L setNode: binary[i-1]];
     [L setConstraint: @selector(melodic:)];
     [cp[i] addToNeighbours: L];

     L = [Link new];
     [L setNode: binary[i]];
     [L setConstraint: @selector(melodic:)];
     [cp[i-1] addToNeighbours: L];
    }

  L = [Link new];
  [L setNode: binary[0]];
  [L setConstraint: @selector(perfectCfHarmonic:)];
  [cf[0] addToNeighbours: L];

  for (i = 1; i < NUM_BARS - 2; i++)
    {
     L = [Link new];
```

```
    [L setNode: binary[i]];
    [L setConstraint: @selector(harmonic:)];
    [cf[i] addToNeighbours: L];
   }

 // There's no harmonic constraint on the 2nd last bar.

 i = NUM_BARS - 1;

 L = [Link new];
 [L setNode: binary[i]];
 [L setConstraint: @selector(perfectHarmonic:)];
 [cf[i] addToNeighbours: L];

// The ternary constraints...
for (i = 0; i < NUM_BARS - 2; i++)
  {
   ternary = [Constraint3 new];
   [ternary setVariable: cp[i+1]];
   [ternary setVariable2: cp[i+2]];
   L = [Link new];
   [L setNode: ternary];
   [L setConstraint: @selector(skip:step:)];
   [cp[i] addToNeighbours: L];

   ternary = [Constraint3 new];
   [ternary setVariable: cp[i]];
   [ternary setVariable2: cp[i+2]];
   L = [Link new];
   [L setNode: ternary];
   [L setConstraint: @selector(skipped:step:)];
   [cp[i+1] addToNeighbours: L];

   ternary = [Constraint3 new];
   [ternary setVariable: cp[i]];
   [ternary setVariable2: cp[i+1]];
   L = [Link new];
   [L setNode: ternary];
   [L setConstraint: @selector(step:skip:)];
   [cp[i+2] addToNeighbours: L];
  }

// quaternary...
```

```
for (i = 0; i < NUM_BARS - 1; i++)
  {
   quat = [Constraint4 new];
   [quat setVariable: cp[i]];
   [quat setVariable2: cf[i+1]];
   [quat setVariable3: cp[+1]];
   L = [Link new];
   [L setNode: quat];
   [L setConstraint: @selector(noParallel:To:Perfect:)];
   [cf[i] addToNeighbours: L];

   quat = [Constraint4 new];
   [quat setVariable: cf[i]];
   [quat setVariable2: cp[i+1]];
   [quat setVariable3: cf[i+1]];
   L = [Link new];
   [L setNode: quat];
   [L setConstraint: @selector(noParallel:To:Perfect:)];
   [cp[i] addToNeighbours: L];

   quat = [Constraint4 new];
   [quat setVariable: cf[i]];
   [quat setVariable2: cp[i]];
   [quat setVariable3: cp[i+1]];
   L = [Link new];
   [L setNode: quat];
   [L setConstraint: @selector(noPerfect:From:Parallel:)];
   [cf[i+1] addToNeighbours: L];

   quat = [Constraint4 new];
   [quat setVariable: cp[i]];
   [quat setVariable2: cf[i]];
   [quat setVariable3: cf[i+1]];
   L = [Link new];
   [L setNode: quat];
   [L setConstraint: @selector(noPerfect:From:Parallel:)];
   [cp[i+1] addToNeighbours: L];
  }

if ([csp makeArcConsistent])
   [csp findSolutions];
else printf ("Not consistent.\n");
```

```
 printf ("Num solutions: %ld, Num backtracks: %ld, Num nodes: %ld\n",
          [csp getSol], [csp getBts], [csp getNodes]);
 printf ("Num constraint checks: %ld\n", [Note getChecks]);

 [csp free];

 exit(0);
}
```

Bibliography

[Borning81] Borning, Alan. "The Programming Language Aspects of ThingLab." *TOPLAS*, 3 (4), pp. 353-387.

[Borning87] Borning, Alan. "Constraint Hierarchies." *OOPSLA 1987 Proceedings*, pp. 48-60.

[Cohen90] Cohen, Jacques. "Constraint Logic Programming Languages." *Communications of the ACM*, 33 (7), pp. 52-68.

[Colmerauer90] Colmerauer, Alain. "An Introduction to Prolog III." *Communications of the ACM*, 33 (7), pp. 69-90.

[Cope87] Cope, David. "An Expert System for Computer-assisted Composition." *Computer Music Journal*, 11 (4), 1987.

[Cox86] Cox, Brad. *Object-Oriented Programming: An Evolutionary Approach.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1986.

[deKleer89] de Kleer, Johan. "A Comparison of ATMS and CSP Techniques." *IJCAI 1989 Proceedings*, pp. 290-296.

[Doyle79] Doyle, J. "A Truth Maintenance System." *Artificial Intelligence*, 12, pp. 231-272.

[Ebcioglu84] Ebcioglu, Kemal. "An Expert System for Schenkerian Synthesis of Chorales in the Style of J.S. Bach." *ICMC 1984 Proceedings*, pp. 135-142.

[Ebcioglu86] Ebcioglu, Kemal. "An Expert System for Chorale Harmonization." *AAAI 1986 Proceedings*, pp. 784-788.

[Ebcioglu88] Ebcioglu, Kemal. "An Expert System for Harmonizing Four-part Chorales." *Computer Music Journal*, 12 (3).

[Fikes70] Fikes, Richard E. "REF-ARF: A System for Solving Problems Stated as Procedures." *Artificial Intelligence*, 1, pp. 27-120.

[Filman88] Filman, Robert E. "Reasoning with Worlds and Truth Maintenance." *Communications of the ACM*, 31 (4), pp. 382-401.

[Freuder78] Freuder, Eugene. "Synthesizing Constraint Expressions." *Communications of the ACM*, 21 (11), pp. 958-966.

[Freuder82] Freuder, Eugene. "A Sufficient Condition for Backtrack-Free Search." *Journal of the ACM*, 29(1), pp. 24-32.

[Garey79] Garey, Michael and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco: W. H. Freeman and Company, 1979.

[Golomb65] Golomb, S.W. and L.D. Baumert. "Backtrack Programming." *Journal of the ACM* , 12, pp 516-524.

[Guesgen87] Guesgen, H. W. *et al.* "Constraints in a Hybrid Knowledge Representation System." *IJCAI 1987 Proceedings*, pp. 30-33.

[Haralick80] Haralick, R.M. and G. L. Elliot. "Increasing Tree Search Efficiency for Constraint Satisfaction Problems." *Artificial Intelligence*, 14, pp. 263-313.

[Hayes-Roth86] Hayes-Roth, Barbara *et al.* "PROTEAN: Deriving Protein Structure From Constraints." *AAAI 1986 Proceedings*, pp. 904-909.

[Hiller59] Hiller, Lejaren A. and Leonard M. Isaacson. *Experimental Music: Composition with an Electronic Computer.* New York: McGraw-Hill, 1959.

[IMA83] International MIDI Association. *MIDI Musical Instrument Digital Interface Specification 1.0.* International MIDI Association: North Hollywood, 1983.

[Knuth74] Knuth, Donald E. "Estimating the Efficiency of Backtrack Programs." STAN-CS-74-442, August, 1974.

[Leler88] Leler, Wm. *Constraint Programming Languages: Their Specification and Generation.* Reading, Massachusets: Addison-Wesley Publishing Company, 1988.

[Levitt84] Levitt, David. "Machine Tongues X: Constraint Languages." *Computer Music Journal,* 8 (1), pp. 9-21.

[Mackworth77] Mackworth, Alan K. "Consistency in Networks of Relations." *Artificial Intelligence*, 8, pp. 99-118.

[Mackworth85] Mackworth, Alan K. and Eugene Freuder. "The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems." *Artificial Intelligence*, 25, pp. 65-74.

[Mackworth87] Mackworth, Alan K. "Constraint Satisfaction." *Encyclopedia of Artificial Intelligence*, New York: John Wiley & Sons, 1987, pp. 205-211.

[Mann65] Mann, Alfred ed. *The Study of Counterpoint from Johann Joseph Fux's Gradus Ad Parnassum.* New York: W. W. Norton, 1965.

[Meyer56] Meyer, Leonard B. *Emotion and Meaning in Music.* Chicago: University of Chicago Press, 1956.

[Minsky80] Roads, Curtis. "Interview with Marvin Minsky." *Computer Music Journal*, 4 (3), pp. 25-39.

[Montanari74] Montanari, Ugo. "Networks of Constraints: Fundamental Properties and Applications to Picture Processing." *Information Science*, 7, pp. 95-132.

[Nadel88] Nadel, Bernard A. "Tree Search and Arc Consistency in Constraint Satisfaction Algorithms." *Search in Artificial Intelligence.* L. Kanal and V. Kumar, eds. New York: Springer-Verlag, 1988.

[Nadel89] Nadel, Bernard A. "Constraint Satisfaction Algorithms." *Computational Intelligence,* 5 (4), pp. 188-224.

[NeXT89] NeXT Inc. "Object-Oriented Programming and Objective-C." On-Line Documentation, Release 1.0.

[Nudel83] Nudel, Bernard. "Consistent Labeling Problems and their Algorithms: Expected-Complexities and Theory-Based Heuristics." *Artificial Intelligence*, 21, pp. 135-178.

[Roads85] Roads, Curtis. "Research in Music and Artificial Intelligence." *ACM Computing Surveys*, 17(2), pp. 163-190.

[Rossi88] Rossi, Francesca. "Constraint Satisfaction Problems in Logic Programming." *SIGART Newsletter*, 106, pp. 24-28.

[Stallman77] Stallman, Richard M. and G. J. Sussman. "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer Aided Circuit Analysis." *Artificial Intelligence*, 9, pp. 135-196.

[Steels85] Steels, Luc. "Constraints as Consultants." *Progress in Artificial Intelligence.* Chichester, England: Ellis Horwood, 1985.

[Stefik86] Stefik, Mark J. and Daniel G. Bobrow. "Object-Oriented Programming: Themes and Variations." *AI Magazine*, 6 (4), pp. 40-62.

[Sussman80] Sussman, G.J. and G.L. Steele, Jr. "CONSTRAINTS - A Language for Expressing Almost-Hierarchical Descriptions." *Artificial Intelligence*, 14, pp. 1-39.

[Swindale62] Swindale, Owen. *Polyphonic Composition.* London: Oxford University Press, 1962.

[Taenzer89] Taenzer, David *et al.* "Object-Oriented Software Reuse: the Yoyo Problem." *Journal of Object-Oriented Programming*, 2 (3), pp. 30- 35.

[Thomas85] Thomas, Marilyn Taft. "Vivace: A Rule Based AI System for Composition." *ICMC 1985 Proceedings*, pp. 267-274.

[VanHentenryck87] Van Hentenryck, Pascal. "A Theoretical Framework for Consistency Techniques in Logic Programming." *IJCAI 1987 Proceedings*, pp. 2-8.

[VanHentenryck89] Van Hentenryck, Pascal. *Constraint Satisfaction in Logic Programming.* Cambridge, Massachusetts: MIT Press, 1989.

[Waltz75] Waltz, David. "Understanding Line Drawings of Scenes with Shadows." *The Psychology of Computer Vision.* Patrick Henry Winston, ed. New York: McGraw-Hill, 1975.

[Winston84] Winston, Patrick Henry. *Artificial Intelligence (Second Edition).* Reading, Massachusetts: Addison- Wesley, 1984.