Visual and Adaptive Constraint Programming in Music

Charlotte Truchet, Gérard Assayag, Philippe Codognet

Equipe Représentation Musicales, IRCAM and LIP6, Université de Paris 6, Paris, France
email: {Charlotte.Truchet, Gerard.Assayag}@ircam.fr Philippe.Codognet@lip6.fr

Abstract

We propose an environment for musical constraint solving, adapted to contemporary composition, in the visual programming language OpenMusic. We describe the use of two algorithms, a complete one and an incomplete one. The latter is a local search algorithm, called adaptive search. Adaptive search refines the concept of cost function by taking into account the weight of each variable. We show that an incomplete algorithm is well adapted for musical problems. The first experimental results on real musical problems are satisfying, including in terms of computation time.

1 Introduction

Constraint programming is a major issue in the domain of Computer Assisted Composition (CAC). Expressing problems with constraints is natural in composition. Harmonization is a good example : the textbooks give rules like "no parallel fifth", "opposite motion between two voices", and so on. So to say, they are written in a declarative way. For contemporary music, the rules certainly differ, but the principle of stating rules remains, and some composers already use constraint solvers.

In the field of automatic harmonization by constraints programming, the first solver was proposed by Ebcioglu (K. Ebcioglu, 1997), followed by Tsang and Aitken whose solver was still quite slow (C. P. Tsang and M. Aitken. 1991). Philippe Ballesta wrote a solver based on Ilog Solver (P. Ballesta. 1998). Pachet and Roy used the particular structures of the tonal music to build a system with BackTalk (F. Pachet and P. Roy. 1995). All these systems are compared on a very specific problem, automatic harmonization in the choral style, and stand on a specific music theory, tonal music.

Two solvers have been developed for contemporary music, PWConstraints (M. Laurson 1996) and Situation (C. Rueda and F. Valencia 1997). Both are based on the forward checking paradigm. They are useful for certain type of problems, but still lack the efficiency, the genericity (for PWConstraints) and the ease of use that are crucial for real-size production situations.

2 Goal

Our goal is to build a constraint programming system for contemporary composers. We figure out set of requirements that cannot be found together in existing systems :

- genericity : it must be possible for the user to define, through representations and rules, his own music-theoretic background, including the classical one if needed.

- visual programming : the primitives in the system must be accessible through a visual programming, highly graphical environment.

- rich constraints : the system must deal with difficult global constraints such as « AllDiff », « It exists..such as » and with constraint hierarchies.

- completeness scalability : we want the user to be able to scale freely the system between completeness, which may involve huge calculations, and incompleteness, i.e. approximate solutions that can be extremely fast to find while very close to optimal solutions.

We present two techniques for dealing with musical constraints in OpenMusic, an OpenSource Visual Programming Language for musicians (G. Assayag *et al.*, 1999). The first one is a visual integration of a classical solver with backtracking and propagation techniques. The second one is an innovative local search algorithm, very efficient and well adapted to musical constraint problems. The ultimate goal is to gather these two solvers under a common interface, a visual declarative language for musical constraints problems. The underlying engine would then scale up and down between completeness and uncompleteness depending on the user's need of a fast (possibly real time) solution or of an exhaustive search.

We first discussed with several composers who either had already used constraints solvers, or had some constraints problems they could not solve. We observed that their Constraint Satisfaction Problems (CSPs) were all very differents, for the constraints and for the data structures as well (chords, tempi, metrics, rythmic or melodic patterns, harmonic structures, etc). The constraints are sometimes only arithmetic constraints, more often they contain "all different", "exists", "for all" type of constraints which are seldom available in existing musical softwares, and known to be difficult to handle. The domains are always finite.

We had then to avoid two extreme options : the first one would be to write very user-friendly primitives, according to a particular composer, but probably useless for another. On the other side, we can give plenty of low-level constraint primitives, so that the language would be very expressive, but users would have a hard time combining the primitives to write their own constraints... We have decided to start with the second option, which still allows us to propose an adapted constraint language once we have written the solver. Anyway, the solver has to deal with a large set of constraint types.

Concerning the solver, we will distinguish two types of needs, with which we will deal separately. Most of the composers want to find *one* solution to their CSP, maybe a few solutions. In this case, we will use an uncomplete algorithm. But more rarely, one may need to generate a musical material with constraints, something like a database. In other terms, one wants all the solutions to the CSP, however costly it migth be. In this case, we cannot avoid to search the entire domain, so we use backtracking techniques.

In the first case, we observed that the principle "wait a long time to get an exact solution" was not probably the best. A composer who uses a constraint solver may re-write the solution, changing some terms according to (non formalizable) criteria. Atypicals or surprising solutions must not necessarily be missed, even if they are not exactly correct. They may be of some interest precisely because of their originality. Finally, since we will sometimes have to deal with unsolvable problems, it makes sense to provide approximate solutions as soon as they are available.

For all these reasons, we have designed an uncomplete algorithm, called adaptive search, belonging to the family of local search methods. It works by successive improvement of an initial configuration, and allows us to deal with approached solutions. We can also display the intermediate solutions, which makes sense since each configuration is better than the previous one. The composer can thus stop the evaluation when he or she considers to be "near enough".

3 Backtracking

We have imported in OpenMusic (OM) a Lisp-written constraint solver, called Screamer (J. M. Siskind and D. A. McAllester 1993). Screamer adds indeterminism to Lisp, by two primitives, *either* and *fail*. *either(e1... en)* first evaluates *e1*, if it leads to a fail, it evaluates *e2*, and so on. Screamer introduces backtracking techniques in OpenMusic, and also has a propagation solver. OM version 4 now has a Screamer library with all the functions for backtracking and accessing the solver. We have added to OM's visual language a new visual program class (patch) for the indeterministic functions. For these patches, the code generation directly calls Screamer primitives. The constraints can now be written in the visual language, as in figure 1.

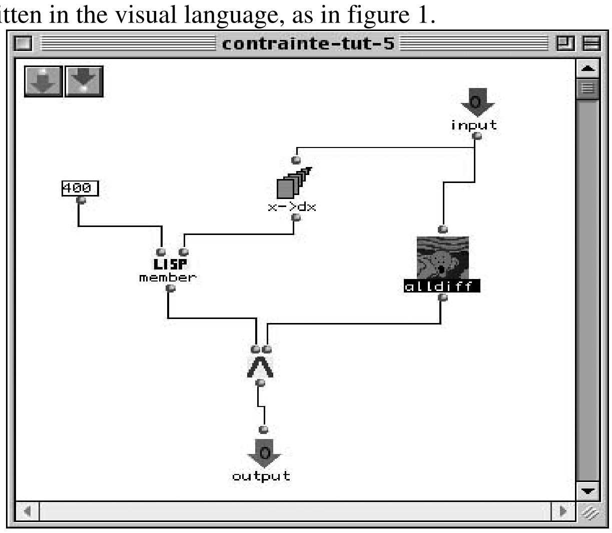

Figure 1 A patch expressing a constraint in OM 4. The constraint here states that a chord must contain all-different notes, and an interval of 400 midicents.

With Screamer, we can handle with problems for which all the solutions are needed. We did so for instance for a composer and musicologist, Jean-Michel Bardez, who wanted to find micro-intervallic musical scales with quarter tones. Of course, the whole set of these is far too big for anyone to listen to. So we added some constraints on the generated scales : symetries of various kinds, limited transposition, or a more complicated and significant one, an harmonic constraint which states that there is at least a fixed number of chord triads where all the chord intervals are not tempered.

This constraint allowed us to generate all the scales made of either semitones or quartertones (domain of size 2), with at least 6 semitones and 6 quartertones, and :

- at least eight times a "tonic chord" (0 4 7) in quartertones (calculation time 2 min 30 s, 4272 solutions)
- at least twelve times a "tonic chord" (0 4 7) in quartertones (calculation time 2 min 38 s, 126 solutions)
- at least twelve times a "dominant chord" (0 4 7 10) in quartertones (calculation time 3 min 40 s, no solution)

In this example, the important is that we want to build a database of scales, so that composers can pick in it for their own needs. So we do not need to bother with the calculation time, since the calculus will be made once for all. In cases like this, backtracking techniques are well adapted.

4 Local Search

4.1 Generalities

This algorithm has been proposed by Philippe Codognet (P. Codognet 2000), who tested it on classical CSPs like the n-queens, magic square number partitioning and arithmetic problems. It deals with problems in CSP format : variables, associated (finite) domains and constraints on the variables. Many solvers already exist for the CSP over finite domains, like GNU-Prolog (P. Codognet and D. Diaz 2000) or ILOG Solver (J.-F. Puget 1994). Departing from classical constraint solving techniques, adaptive search is a heuristic algorithm, based on the the local search family (E. Aarts and J. Lenstra 1997). These algorithms have already proven their efficiency on combinatorial problems such as the boolean satisfaction (SAT), the travelling salesman problem (TSP), job-shops, vehicule routing, etc. They basically rely on the idea of collecting information on the quality of the current configuration (assignment of variables to values of their domains), exploring a neighbourhood of slightly modified configurations and moving to the most promising one. Simulated annealing, genetic algorithms, Tabu method or the Greedy SAT algorithm are well-known instances of methods based on the local search paradigm.

In adaptive search, every constraint is represented by an error function on the variables, which gives a measure of "how much" the constraint is violated. For instance, if the constraint is "V1=V2", we could choose as a cost function abs(V1 - V2) for variables V1 and V2, and 0 for the other variables. Combining these functions in a global cost function (for instance by summing them), we can tell both the quality of the current variable-value configuration, and the cost of each variable regarding the constraints. Starting from a random assignment, the iterative process computes the costs of each variable, finds the worst variable (i.e. the most expensive one), searches its domain to find a better value that minimizes global cost, replaces it.

4.2 Algorithm

1. Random initialization

Repeat

2. Compute the costs of every (not Tabu) variable, and sum them to obtain the global cost of the current configuration
3. Select V the variable of higher cost, and explore its domain to search a better value, ie a value that minimize the global cost.
4. **If** there is no better value **then** mark V Tabu, **else** replace V by its better value.

Until the global cost is zero.

Written so, there is absolutely no reason why the algorithm should end. It does not when there is no solution, and even when there are solutions nothing guarantees the ending. To avoid this, one can count the iterations and stop when a maximum number is reached.

4.3 Local Minima

To avoid being trapped in local minima, the algorithm includes an adaptive memory, such as in Tabu Search : if a variable leads to a local minimum, it is marked Tabu and left out of the search for a fixed number of iterations. This number, the Tabu length, is an important parameter for the algorithm, intuitively it measures the stubbornness of the algorithm. A high value (the highest value being the number of variable) means that all the paths will be explored as long as it is possible. A low value means that the algorithm leaves quickly the paths that seem not promising. The user can fix the Tabu length according to his own needs : for instance, when his CSP has no solution, the Tabu length should be high.

Local minima for all the variables can also be met. All the variables will then successively be marked Tabu. In this case the algorithm is re-started randomly. There are many ways to reset : one can start again from nothing, all the variables at random. Another possibility is to reset only some variables (as suggested in C. P. Gomes, B. Selman and H. Kautz 1998). Experimentally, a good compromise is to reset the Tabu variables randomly.

4.4 Implementation

We have implemented this algorithm in OpenMusic, creating a particular solver class. The user can construct a class that inherit from any musical OM class, and from the solver class. During the evaluation, as soon as a local minimum is found, the musical object is upgraded, so that it contains constantly the best solution found so far. Two tresholds can be fixed by the user. The first one is a cost limit for the local minima to be given. The second one is the treshold for ending : the evaluation ends as soon as the global cost is lower than it. The user can also adjust the weight of the constraints, as in example 6.3 below.

This algorithm satisfies many of our goals, mainly because the assignments are directed towards improvement of the configuration. First, it deals naturally with partial solutions. The user can choose a treshold, and ask for the best local minimum reached so far to be printed, provided the global error is lower than this treshold. When there is no exact solution to the CSP, it allows to get partial solutions. With the same principle he can obtain approached solutions if he wishes.

The representation with costs, instead of boolean constraints, gives the possibility to put weight on the constraints, by multiplying their cost by a chosen weight. Thus, playing with the treshold, we can make a distinction between the constraints that must be satisfied and the constraints that may be (preferences). Finally, the global constraints are easily handled. Their only feature is their cost functions being constants, which is not a problem for the solver.

5 Constraint Language

5.1 Grammar

Switching from constraints to cost-functions makes the expression of constraints more difficult. It is out of question to let the composer write the cost-functions himself. We have written a parser for the constraint language we defined. This parser analyzes the constraints expressed in logical form and computes the associated cost functions, based on a formal transduction grammar. This language contains arithmetic (for instance - to calculate an interval, or * to convert midis in midicents), equality, inequality (a pitch value must be geater than a fixed value), appartenance (a chord must contain a precise note), logical or and and, all different (a chord must be made of different notes), existential and universal quantifiers, which allows to express all the CSP we need.

More precisely, the constraint language is made of terms and constraints. The terms *t* are simply Lisp expressions (for instance arithmetic expressions), with the only restriction that they contain no predicate symbol such as = or <. The constraints are

$$C ::= t = t \mid t < t \mid t \in t \mid C \wedge C \mid C \vee C$$
$$\mid \forall t \in t, C \mid \exists t \in t, C \mid alldiff(t...t)$$

There are many ways to define the parser. The only strong requirement for a cost-function f_C associated to a constraint C is $f_C(t) = 0 \Leftrightarrow C(t)$. Of course, we can guess that the solver will work better with good cost-functions, that really represent the cost of the constraint : this is the main idea of the algorithm. But even so there are obviously many possibilities for f_C. We have chosen the following for our parser .

$$P(t_1 = t_2) = |t_1 - t_2|$$
$$P(t_1 < t_2) = \max(0, 1 + t_1 - t_2)$$
$$P(t_1 \le t_2) = \max(0, t_1 - t_2)$$
$$P(t_1 \in t_2) = \min\{|t_1 - t|, t \in t_2\}$$
$$P(C_1 \wedge C_2) = \max(P(C_1), P(C_2))$$
$$P(C_1 \vee C_2) = \min(P(C_1), P(C_2))$$
$$P(\forall t_1 \in t_2, C(t_1)) = \max\{P(C)(t_1), t_1 \in t_2\}$$
$$P(\exists t_1 \in t_2, C(t_1)) = \min\{P(C)(t_1), t_1 \in t_2\}$$
$$P(alldiff(t_1...t_n)) = Card\{t_i = t_j, i < j < n\}$$

The cost-functions generated this way are meaningfull and continuous, and work experimentally well, but for the alldiff constraint primitive.

5.2 All-different constraint

Any $f_{alldiff}$ will necessarily be uncontinuous, according to the above definition. Of course, we cannot avoid the alldiff primitive which is crucial for musical applications, and probably the primitive the most often used by the composers we have met : chords are made of different notes, rythms are made of different onsets, scales are made of different midi values, and so on.

Since the alldiff is such a necessary constraint, and must be satisfied, we have tried other ways to handle with it. In case of an alldiff on n variables, each variable having the same domain of size n (permutation of n variables), the easiest way is to slightly modify the problem (keeping of course the same solutions) and to work on the permutations of the variables. This is the case in example 6.1 described below.

In case of an alldiff on n variables, each variable having a domain of size d, d>n, is it more efficient to modify the algorithm. Instead of searching the whole domain of the worst variable for a better value, it searches the domain minus the already instanciated values of the other n-1 variables. For classical constraint programmers, this sounds like a sort of (micro) forward checking, but very easy to perform since we already have instanciated variables. Experimentally, the latter solution is faster than a alldiff cost-function.

5.3 Example of predefined constraint

This constraint language allows to define a very large scope of constraints, and is far enough sufficient for the composers' needs. Anyway, as said before, parsing constraints based on very low-level constraint primitives is not satisfying for musical purposes. The composers' CSP are very diverse but show some similarities, which already allows to define more elaborated constraints (and cost-functions). Let's describe the most important one (apart from the alldiff constraint, to avoid trivial solutions).

The most frequent constraint is to minimize a distance-like value on all the configuration, as in example 6.1 or 6.2. The user defines its own distance d (which is usually not a mathematical distance, but this does not matter) between two variables. Then he wants to minimize of maximize it on a set of n variables $V_1 \ldots V_n$. The best cost function is then $f(V_i) = d(V_{i-1}, V_i) + d(V_i, V_{i+1})$. About half of our CSPs can be expressed this way. Musically, the idea of minimizing a distance is natural and the adaptive search works very well on this kind of problems.

6 Examples

We have tested the adaptive search implementation on a about ten problems. We will detail in this paper the three most interesting.

6.1 Sorting chords

Given a set of chords, the goal is to order them so that the number of common notes between two successive chords is maximal. It can be seen as a manner to arrange a musical material initially at random. To solve this problem exactly, we have to visit the entire domain of possibilities, with a factorial complexity. The goal is to find reasonably good solutions in a reasonable time.

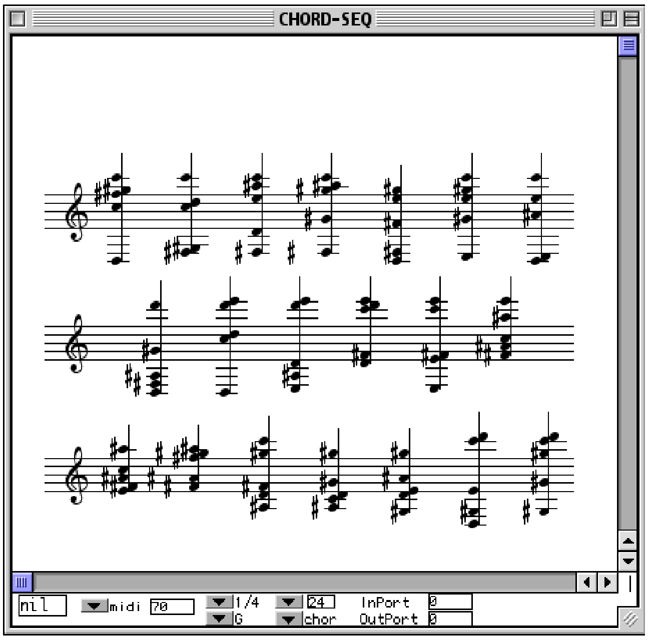

Figure 2 An solution found with adaptive search.

An algorithm based on a Travelling Salesman Problem Approximation Algorithm graph theory already exists in OM, which optimizes the global number of common notes. The trouble is that it shows local gaps : for a long sequence, the successive chords have many notes in common, then there are two chords with no shared notes. Musically, this is not satisfying even if the number of total common notesis high.

With adaptive search, we can avoid this by putting an additional constraint, with a great weight, stating "at least one shared note between two successive chords", plus the constraint on the number of common notes. There are then many strategies, since we can never know when to stop. A first possibility is to mark the local minima and to stop when a local minimum with same cost is encountered more than k times, k fiwed by the user. This works very well to find the best solution but is far too slow. A good strategy is simply to try k times, k fixed by the user. The calculation time becomes then reasonnable and the solutions found are good in average. The results are very satisfying. For 50 attempts with 8 chords of 6 notes, the best solution has been found 39 times and the other answers were only one note far from the best.

Figure 2 shows an example of resolution by adaptive search. The 20 chords are chosen at random on a domain of size 20. The numbers of common notes between two successive chords is (2 2 3 2 2 3 1 2 2 3 3 2 4 3 2 3 2 2 3), and 46 common notes (in 1,5 seconds). If we run the OM algorithm on the solution already found, we obtain for the common notes (2 3 2 2 3 2 3 4 2 2 2 1 1 3 2 1 3 1 3 1) and 43 common notes.

In this example, the interest of the adaptive search algorithm is not really to be fast but to allow to refine the constraints, or to organize them into a hierarchy.

6.2 Rythmical patterns

We have n voices, each one playing repetitively a rythmic pattern of fixed length. The time unit is fixed, and the patterns can be represented as sets of (all different) integers. The number of onsets by pattern is also fixed. The constraint states that for a fixed duration, we never hear two onsets at the same time, see figures 3 and 4. An important parameter of this CSP is the average density of onsets. A density of 2 means that we hear in average one onsets every 2 time units. Of course, it fixes the number of onsets by pattern, that is the number of variables for our CSP.

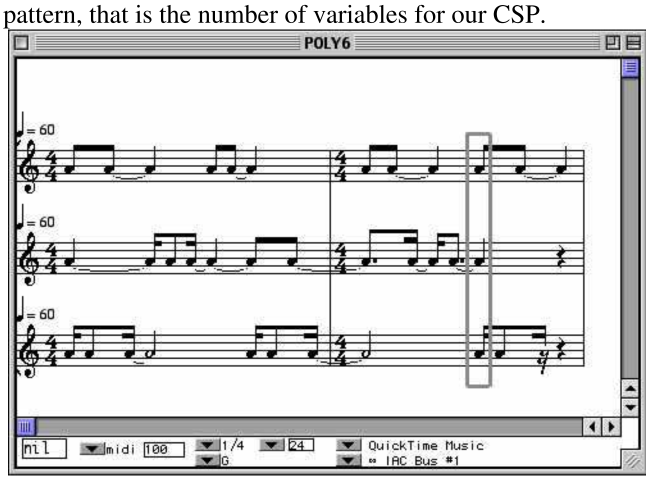

Figure 3 An approximate solution, error framed in gray.

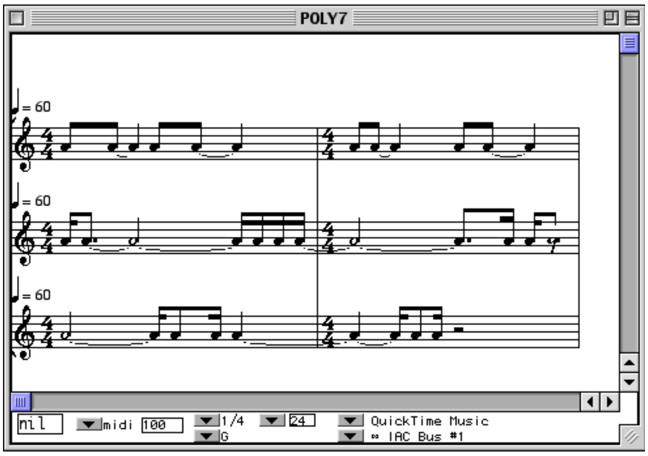

Figure 4 An exact solution.

There is obviously no solution when the density of onsets becomes too big. Even for low values, it is particularly difficult to solve.

One this CSP, the calculation time of a classical solver like PWConstraints is around half an hour for n = 6. (on a Mac G4). The adaptive search performs very well : 12 seconds (350 iterations) for n = 3 (on a Mac G4), 108 seconds (1208 iterations) for n = 6 on an average of 10 runs (duration of 128 time units, density of onsets of 2). We see also experimentally a high variability in the calculation times, a difference of 2 or 3 times is not rare.

Furthermore, this algorithms deals with the case when there is to solution for instance whith a density less than one.

6.3 Nzakara harp

This problem has been formulated by Marc Chemillier (M. Chemillier 1999). One of its most interesting properties is that it has no exact solutions because of two conflicting constraints. The goal is to construct a sequence of n bichords, each bichord in a domain of size 5. The first constraint C1 states that the lower voice of the sequence reproduces the upper voice (apart from a transposition), with a time gap of p (n and p are fixed integer). Formally, for every k < n, the lower note of the (k+p)-th bichord is determined by upper note of the k-th bichord. The second constraint C2 is to avoid trivial sequences such as (a a a…) or (a b a b a b...). It is sufficient to write it as "bichord (k+p) ≠ bichord k". Notice that the two constraints are antagonistic. An example is shown figure 5.

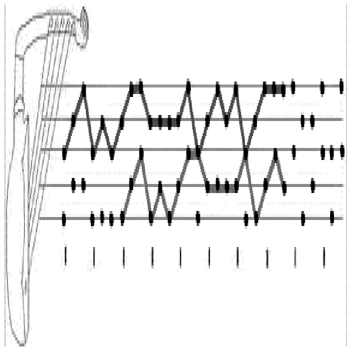

Figure 5 The lower voice reproduces the upper voice with a time gap and a translation.

A complete constraint solver like Prolog cannot deal with this kind of problem : after a huge calculation time, the answer is of course "no". The first interest of adaptive search in this case is thus to get approximate solutions, as usual. But this CSP shows also the utility of weighing the constraints. It is possible, and easy, to play with the balance of the constraints since there are only two and they are antagonistic (one can find a solution with C1, or with C2, but not with both). It is sort of a deal : how much do you want of C1, how much are you ready to release on C2. This example shows the advantage of adaptive search for over-constrained CSP. Experimentally, the adaptive solver finds good approximate solutions. Figure 6 shows a resolution with n = 30, p = 6, and zero errors for C2, seven for C1. The Nzakara make 6 errors on C1 with these values.

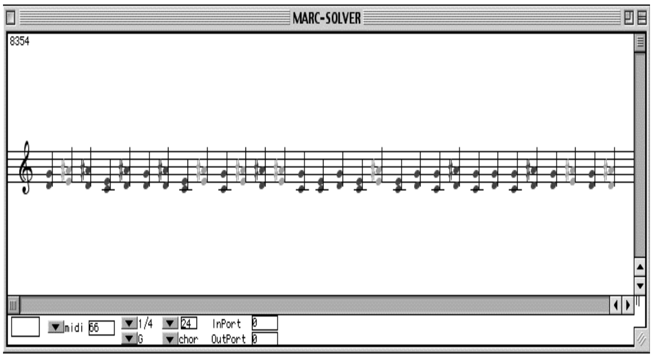

Figure 6 Bichords with an error for C1 are in light gray.

7 Conclusion

The experimental results of adaptive search are very satisfying. It is particularly obvious on hard problems, not to mention the insolvable problems where classical techniques gives no answer after a long time, while adaptive search can give approached answers quite quickly. The transduction grammar makes the tedious task of designing cost functions transparent to the user, which can express his rules as classical logical constraints. The grammar also makes it possible to unify the adaptive search algorithm and the classical CSP solver under the same visual constraint specification language.

Future works include a better implementation of the adaptive search, an intensive experimentation phase where a great number of musical problems, involving composers of different style and culture, will be submitted to the system, undoubtedly leading to new optimization needs and to new improvements. Finally, the two solvers must be integrated with the same interface and constraint language.

References

E. Aarts and J. Lenstra 1997. "Local Search in Combinatorial Optimization." *Wiley*.

G. Assayag, C. Rueda, M. Laurson, C. Agon and O. Delerue 1999. "Computer Assisted Composition at Ircam : PatchWork & OpenMusic." *Computer Music Journal* 23:3.

P. Ballesta. 1998. " Contraintes et objets, clefs de voûte d'un outil d'aide à la composition." *Editions Hermès.*

M. Chemillier 1999. "Ethnomusicologie, ethnomathématique. Les logiques sous-jacentes aux pratiques artistiques transmises oralement." *Forum Diderot*, Société Mathématique Européenne.

P. Codognet 2000. "Adaptive Search, Preliminary Results." *BOOK of the 4th ERCIM/CompulogNet Workshop*.

P. Codognet and D. Diaz 2000. "The implementation of GNU Prolog." *Proceedings SAC'OO 15th,* ACM Press.

K. Ebcioglu, 1997. " An expert System for Harmonizing Chorales in the style of J.-C. Bach." *AAAI Press.*

C. P. Gomes, B. Selman and H. Kautz 1998. " Boosting Combinatorial Search Through Randomization." *Proceedings of AAAI'98*.

M. Laurson 1996. "Patchwork : a visual programming language and some Musical applications." *Sibelius Academy*.

F. Pachet and P. Roy. 1995. " Integrating Constraint Satisfaction Techniques with Complex Object Structures." *Proceedings of the 15th Annual Conference of the British Computer Society Specialist Group on Expert Systems.*

J.-F. Puget 1994. "A C++ implementation of CLP." *Proceeding of SPICIS'94.*

C. Rueda and F. Valencia 1997. "Improving Forward Checking with delayed evaluation." *Proceedings of CLEI'97*.

J. M. Siskind and D. A. McAllester 1993. "Nondeterministic Lisp as a Substrate for Constraint Logic Programming." *Proceedings AAAI-93*.

C. P. Tsang and M. Aitken. 1991. " Harmonizing Music as a Discipline of Constraint Logic Programming." *Proceedings of the ICMC*.