A Constraint Programming System for Music Composition, Preliminary Results

Charlotte Truchet, Carlos Agon, Philippe Codognet

Equipe Représentation Musicales, IRCAM and LIP6, Université de Paris 6, Paris, France
email: {Charlotte.Truchet, Carlos.Agon}@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. The composer start with a predefined score and set constraint between some objects in a visual way. These constraints are translated in a specific syntax in order to be solved using an algorithm of adaptive research. Adaptive search refines the concept of cost function by taking the weight of each variable into account. We show that this algorithm is well adapted to musical problems. The first experimental results on real musical problems are satisfying, in terms of implementation and computation time.

1 Introduction

Constraint programming offers a powerful and intuitive way of describing and manipulating musical entities. Several working systems and theoretical studies have been proposed, dedicated to different musical applications : automatic harmonization (P. Ballesta. 1998) (F. Pachet and P. Roy. 1995), assistance to composition (M. Laurson 1996) (C. Rueda and A. Bonnet 1998), just to name a few. This paper is related to the use of constraint programming techniques for musical composition.

We propose a new constraint programming system for contemporary composers as part of the OpenMusic system (G. Assayag and al. 1999). 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.

-rich constraints : the system must deal with difficult global constraints such as « AllDiff », 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.

-visual programming : we must keep in mind that users of our system are composers, not necessarily experts in computer science. We prefer a visual programming interface in order to make the access to programming easier for the composer. Another reason, may be more important, is that visual graphic programs can be considered as a musical notation of the compositional process. The principal idea is to integrate visual programming and traditional music notation in a sound way.

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 different, for the constraints as well as for the data structures (chords, tempi, metrics, rhythmic or melodic patterns, harmonic structures, etc). The constraints are sometimes only arithmetic constraints, more often they contain "exists", "for all" type of constraints, and

nearly always "Alldiff", which are seldom available in existing musical software, and known to be difficult to handle. The domains are always finite.
We had then to avoid two extreme options : the first one would have been to write very user-friendly primitives, according to a particular composer, but probably useless for the others. On the other side, we could 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.
Concerning the solver, we have used 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". Atypical 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.
This paper is divided into three principal parts : section 2 details the algorithm of local search. Section 3 defines the basic grammar of our constraint language. We define also the cost-functions associated to each basic constraint. Some high level constraints are discussed in this sections. Section 3.3 shows the graphic interface used to define CSP. We emphasize the integration between music notation and visual programs. Finally, in section 4, we show two realistic examples. Principally, we detail them in terms of graphical definition and execution time. Some considerations and related future works are given in the conclusion.

2 Adaptive Search

This algorithm has been proposed by Philippe Codognet (P. Codognet D. Diaz 2001), who tested it on classical CSPs like the n-queens, magic square number partitioning and arithmetic problems. It is particularly adapted to musical contraint programming (C. Truchet, G. Assayag and P. Codognet). 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 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, vehicle routing, etc. They are basically rest on the idea of collecting information on the quality of the current configuration (assignment of variables to values of their domains), exploring a neighborhood 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, and then replaces it.

Algorithm

1. Random initialization

Repeat

2. Compute the costs of every (not Tabu) variable, and sum them to get the global cost of the current configuration

3. Select V the variable of higher cost, and explore its domain to search a better value, i.e. 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 has been reached.

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 variables) 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 does not seem 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.

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 threshold, 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.

3 Constraint Language

3.1 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), belonging to (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 meaningful and continuous, and work experimentally well, but for the alldiff constraint primitive.

3.2 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 some of them. The cost-functions given here are the ones we found experimentally the best on our CSPs.

Distance

The most usual constraint is to minimize a distance-like value on all the configuration. 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 sequence of n variables $V_1 ... V_n$. 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.

All-different constraint

Any $f_{alldiff}$ will necessarily be discontinuous, 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, rhythms 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.

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 instantiated 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 instantiated variables. Experimentally, the latter solution is faster than a alldiff cost-function.

Increasing list

We often have to deal with increasing sets of variables. For such a constraint on the variables $V_1 \dots V_n$, the cost function (on a variable V_i) can be expressed by the grammar, but this may not be the best one. It is experimentally faster to take into account the cost of each misplaced variables with regard to V_i :

$$f(V_i) = \sum_{j<i} \max(0, V_j - V_i) + \sum_{j>i} \max(0, V_i - V_j).$$

3.3 Visual CSP definition

For the moment, in OpenMusic we integrate scores in graphic algorithms as components that allow the visualization of a particular data at one moment of the computation. It means that the beginning of a composition is the algorithm and graphical editors are used in order to show results. In this paper we have extended the editors to introduce constraint programming into them. This implies a change in the procedure of composition : we start from a musical structure and then we implement programs to modify it. The composer must choose which parts of the structure will be constrained and connect them to a constraint object represented as a diamond in the score.

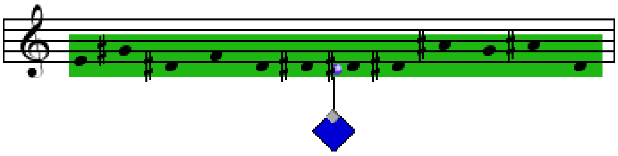

Figure 1 Constraint on a musical object.

We explain the graphic definition of a CSP using a simple example. In the Figure 1 we select a sequence of twelve notes as the object to constrain. By double clicking on the constraint object we get an editor that allows us to define the CSP. Three groups of boxes exist in order to define variables, terms and constraints. In the example of the Figure 2 we use the box *inside-vars* which defines one variable for each note in the sequence. Variables are related to the attribute *midic* (pitch) of the note and the domain for each variable is the arithmetic series from 0 to 11. We have two constraints in this problem. The first one is represented by the box *alldiff,* it forces all variables to be different.

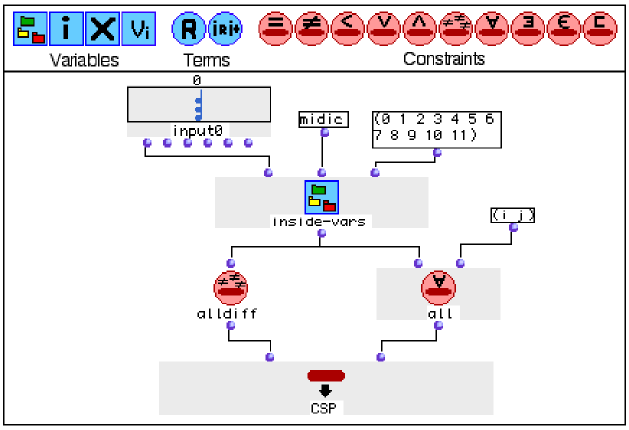

Figure 2 Constraint definition.

The second constraint is represented by the box *all*. This box takes a list of variables and defines a constraint beetwen terms built from the list of variables. Figure 3 shows the constraint definition for the *all* box.

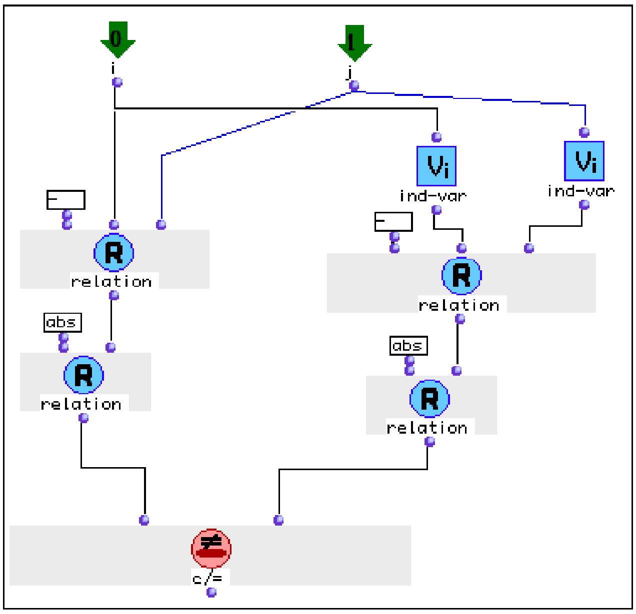

Figure 3 N-queens constraint.

In the last figure we use the *relation* box in order to build terms from the variables. These two constraints define the well know problem of n-queens for n = 12. The graphical patch is translated to the CSP grammar defined in section 3.1 :

Variables : $V_1, \ldots, V_{11}$

Domains : (0_11),…,(0_11)

Constraints :

- alldiff($V1, \ldots, V_{11}$)
- $\forall$ i,j $|V_i - V_j| \neq |i - j|$ for $i \neq j \in 0, \ldots, 11$

Once the CSP is defined, cost-functions are generated and the local search algorithm attempts to solve the problem. A possible solution for our example is the *melody* shown in the Figure 4.

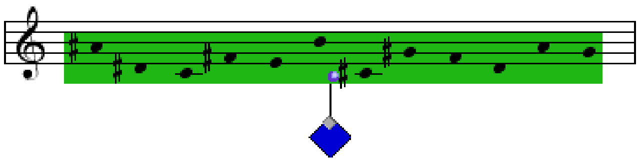

Figure 4 N-queens solution.

3.4 Visual tracing

There is a great interest to implement user-guided search strategies for constraint problems. They allow user to visualize the state of the CSP at one moment of the resolution. One of the most famous tools is OZ Explorer (C. Schulte 1997), which uses the search tree of a constraint problem as its central metaphor. Because we use an adaptive research algorithm this metaphor is not adapted, though we use the fact that we can propose an approximate solution for the problem at any moment of the resolution. We define then a parameter controlling a mechanism of incremental visual update of the musical structures. Moreover the user can associate a particular color for a constraint, so that variables (parts of the score) that do not satisfy a given constraint are drawn in the particular color associated to the constraint, when the solution is updated. User can also hold up the resolution and make modification of the score by hand. It is possible to keep a current solution as the structure associated to the editor.

4 Examples

We have tested the adaptive search implementation on about ten problems. We will detail in this paper two of them.

4.1 Sorting chords

Given a set of chords, the goal is to order them so that the total 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. In the Figure 5 we define a sequence of 10 chords as the musical object to modify.

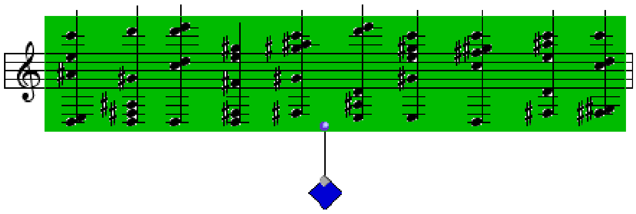

Figure 5 Sorting chords, problem definition.

Figure 6 shows the graphic definition of the CSP. A variable is defined for each chord. The domain of each variable is the set of chords. All variables must be different (*alldiff* constraint). The second constraint (box *c=)* forces the addition of common notes of chords to be equal to the total number of notes. It is clear that this constraint is satisfied only if all chords are the same.

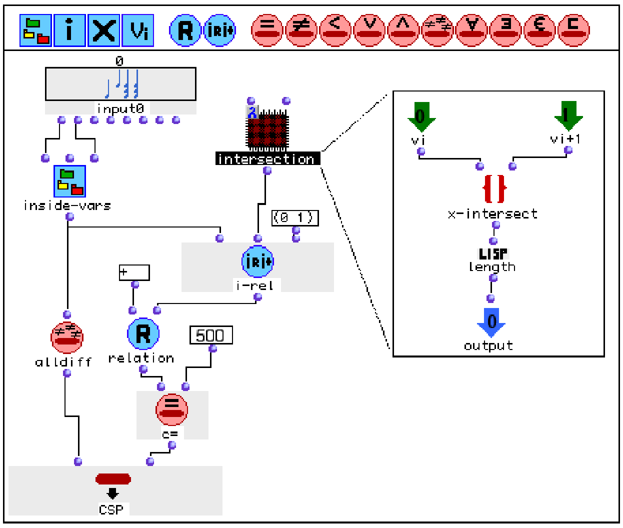

Figure 6 Constraints for sorting chords.

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 fixed 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 reasonable 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 7 shows an example of resolution by adaptive search. The number of common notes between two successive chords are (2 2 3 2 2 3 1 2 2), so 19 common notes. The execution takes about 1 second.

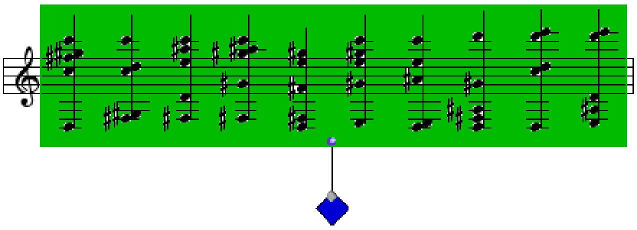

Figure 7 A solution for sorting chords problem.

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.

4.2 Rhythmical patterns

This problem has been proposed by the composer Mauro Lanza. We have n voices, each one playing repetitively a rythmical pattern of fixed length d_i. 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 $\#_i$. The constraint states that for a fixed duration D, we never hear two onsets at the same time. In the Figure 8 we define three voices to constrain (n = 3).

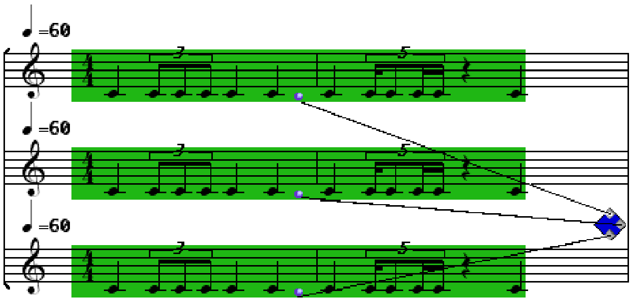

Figure 8 Rhythmical patterns, problem definition.

The patch of the Figure 9 defines variables for the problem $\#_1 = 2$, $\#_2 = 4$, $\#_3 = 3$.

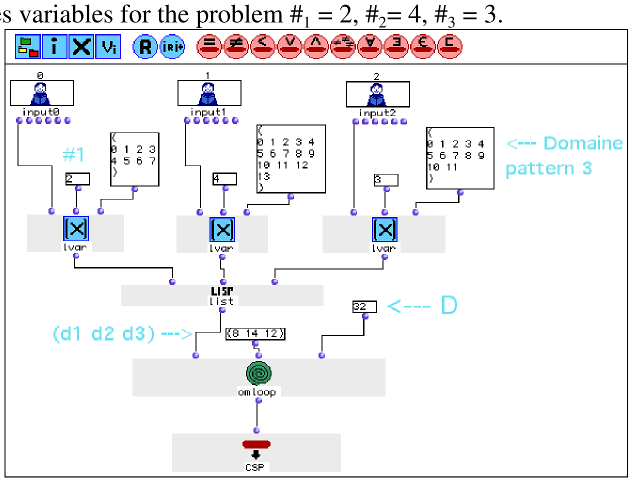

Figure 9 Rhythmical patterns, variable definition.

Using the *loop* control structure of the OpenMusic language we define a set of constraint of the form :

$$(V_{ij} + k_i d_i)\ (d_{i'}) \neq V_{i'j'}$$

where V_{ij} is the j-th note of the pattern of the i-th voice and $k \in \{0, d_i, 2d_i, .., td_i\}$ with $td_i \leq D$, $(t+1)d_i > D$. These constraints are represented in the patch of the Figure 10.

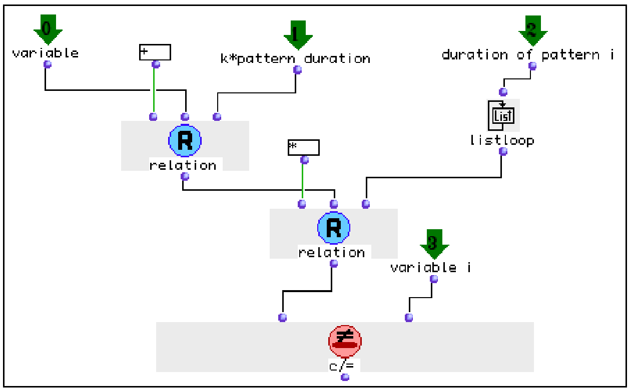

Figure 10 Constraints for rhythmical patterns.

Figure 11 shows an approximate solution for our problem, errors are framed in grey

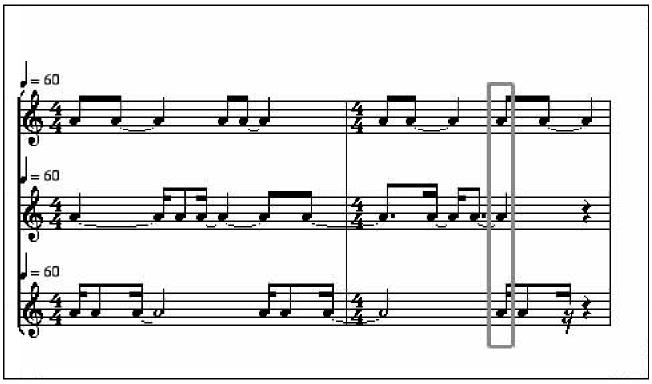

Figure 11 An approximate solution.

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

On 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, 108 seconds (1208 iterations) for n = 6 on an average of 10 runs (duration of 128 time units, average onset density of 2). We see also experimentally a high variability in the calculation times, a computation time factor of 2 or 3 is not rare.

Furthermore, this algorithms deals with the case when there is no solution whith a density less than one, for instance.

5 Conclusion

The experimental results of our system are satisfying. It is particularly obvious on hard problems, not to mention the insolvable problems where classical techniques give 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. Moreover, visual specification of CSPs provide an easier access to programming. However the use of the graphic interface needs to be tested by a more significant population of composers. The grammar also makes it possible to unify the adaptive search algorithm and a 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.

6. Acknowledgement.

Specials thanks to all composers that took part in this project.

7 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.*

P. Codognet and D. Diaz 2001. Yet Another Local Search Method for Constraint Solving. *Proceeding of the AAAI Symposium "Using Uncertainty in Computation", T. Walsh, C. Gomes (Eds.), Cape Cod, AAAI Press October 2001.*

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 A. Bonnet 1998, Un langage visuel basé sur les contraintes pour la composition musicale, *Hermès, 1998.*

C. Schulte 1997. Oz Explorer: A visual constraint programming tool. *In Lee Naish, editor, Proceedings of the Fourteenth International Conference on Logic Programming, pages 286-300, Leuven Belgium, July 1997. The MIT Press*.

C. Truchet, G. Assayag and P. Codognet, Visual and Adaptive Constraint Programming in Music, *Proceedings of the ICMC,* La Havane, 2001.