A Detailed Description of MaxAnnealing

Fernando Iazzetta
Center for New Music and Audio Technologies
University of California at Berkeley
1750 Arch St. – Berkeley, CA - 94709
also from
PUC-SP – Communication and Semiotic Program
fernando@CNMAT.berkeley.edu

Fabio Kon
Department of Computer Science
Institute of Mathematics and Statistics
University of São Paulo
Cx. Postal 66281 – 05389-970
05389-970 – São Paulo (SP) Brazil
kon@ime.usp.br

Abstract

Musical composition can be roughly viewed as a search for the best solution among a finite - although huge - universe of possibilities. Some of the algorithmic compositional techniques try to simulate the act of composing doing this search automatically. However, this approach has two major problems. The first one is the hardness of depicting aesthetic concepts through mathematical rules. The second problem is the low efficiency of the exhaustive search among all possible solutions.

The "Simulated Annealing" algorithm - first proposed in [1] - presents very good results on finding the optimal solution for many combinatorial problems efficiently (in polynomial time). In this paper we present an adaptation of this algorithm to the problem of algorithmic composition. We then discuss some possibilities regarding goal functions to this algorithm and describe MaxAnnealing, a tool designed to help composers and musicologists study the possibility of defining aesthetic concepts through mathematical rules. The system is implemented in the MAX programming environment.

1 Introduction

Musical composition involves many interrelated parameters which interact with each other creating a complex structure of musical possibilities and constraints. As Minsk has said "the problem of creating a good piece of music is a problem of finding a structure that satisfies a lot of different constraints" [2]. To deal with this complexity the composer uses different strategies, some of which are very well defined and can easily be formalized while others remain so flexible and context dependent that resist to any kind of formalization. These strategies include intuition, chance, adaptation, and trial and error based choices. However when the compositional task is transferred to a computer, these strategies are not always available since they are hard to implement as a computer program.

Usually, the compositional task consists of defining a musical goal to be reached by a compositional project. In most cases, not only the compositional project is determined by the goal, but also the latter is sensitive to the constraints imposed by the former. Thus, music arises from the balance of the composer's intentions (goal) and the compelling processes of organizing musical ideas (project). In other words, composition is a constant exercise of adaptation and interaction between project and goal.

However, in the case of computer generated compositions, sometimes it is hard to come up with an efficient project which can be translated into a compositional algorithm. Although the composer can have

a clear idea of his musical intentions, he may be incapable of formalizing these ideas in order to build a program. Actually, for certain kinds of musical problems it is hard, or even impossible, to delineate an algorithm to solve them using the contemporary Computer Science tools. Usually, it can be due to the diversity of elements involved in the problem, to the complexity of parameters that affect the problem, or even to the lack of knowledge about some aspects of the system.

In his piece *Protocol* for solo piano, Charles Ames [3] proposes a compositional method that goes beyond the traditional processes strictly based on random selection or rigid determinism used in previous computer music programs. Ames formalizes his ideas in a "protocol" which is a "collection of tests where each test has been ranked according to one's preferences [...] Then by having the computer evaluate a substantial repertory of alternatives, one can direct it to search for the alternative best fitting these criteria. If there is a choice passing all tests, then systematic evaluation must find it; otherwise, the search will provide the best of imperfect choices" [3, p. 215]. The caveat of this method is that the computer must evaluate all in an enormous range of possible musical configurations to find the best one.

Another problem involving an extensive search of compositional solutions is presented by Schottstaedt in his "Automatic Counterpoint" [4]. Schottstaedt implements a program that generates five species of counterpoint based on the rules exposed by J.J. Fux in the Gradus ad Parnassum (1725). Although the rules which govern the counterpoint are very clear and well defined, there are many different solutions for the same counterpoint problem and the computational time to check each possible solution becomes a significant constraint. "If we make an exhaustive search of every possible branch of a short (10-note) first-species problem, we have 16 raised to the 10th power possible solutions (there are 16 ways to move from the current note to the next note). Even if we could check each branch in a nanosecond, an exhaustive search in this extremely simple case would take 1,000 sec (about 20 minutes)" [4, p. 203]. The solution Schottstaedt presented to this case was to start with the best first result found for each interval, that is, the one to which the program assigns the smallest penalty. But as Schottstaedt recognizes, "the first such solution may not be very good. By accepting the smallest local penalty we risk falling into a bad overall pathway" [4, p. 203].

Some computational tools have been applied in music in order to solve this kind of problem where it is necessary to find the best configuration in a large space of possibilities. Examples are the back-propagation training algorithms used to weight the connections in a neural network, or the genetic algorithms which, inspired in the selective processes that occur in a natural environment, apply successive transformations to a system that evolves toward its environmental fitness. Those tools are based on the search of the best solution of a problem through iterative processes. In these processes, a possible solution is generated and evaluated by a particular function. Then, the program generates a new possible answer which again is evaluated. The results of these successive "guesses" are compared in order to direct the search for the best result. After doing a number of these iterations the system can reach a solution which is sufficiently close to the optimal.

2 The Simulated Annealing Algorithm

Simulated annealing (SA) is a probabilistic algorithm used in the search for optimum solution first described in [1]. It has been developed after the analogy with a Condensed Matter Physics thermal process named annealing and can be used to find near optimal configurations in very large and complex systems.

The annealing process consists on heating a piece of metal until it reaches a temperature slightly above its critical homogenization temperature and then carefully decreasing the temperature until the molecules are arranged in a way so that the metal reaches its thermodynamic equilibrium. The thermodynamic equilibrium is the state in which the molecules form a structure strictly organized and the energy of the system is minimal. If the heating and cooling processes are not correctly done, the metal does not reach the thermodynamic equilibrium.

As a combinatorial optimization process, the purpose of the SA is to find the minimum or maximum values of a cost function for a specific system. The cost function or goal function measures how good a specific configuration is. The SA starts with an initial structure S which can be randomly generated and has a method for modifying this structure generating a neighbor structure. For each step in this process, the quality of the new structure is determined and, if the neighbor solution S' is better than the

current solution, then S' becomes the current solution. But if the new solution does not represent an improvement in the goal function, it still can be accepted with probability

$$e^{\frac{Quality(S')-Quality(S)}{T}}.$$

This condition, known as Metropolis criterion, helps the SA algorithm to escape from local minima. Unlike traditional local search algorithms, SA can make occasional moves in the search space which can decrease the value given by the goal function $Quality()$. The probability of acceptance of the new structure is greater if the difference between the cost of S and S' is small, even if it would represent a decrease in quality. Also, the probability of acceptance of a structure that decreases quality gets smaller as the temperature T decreases providing that the algorithm gets stabilized under a certain temperature.

It is possible to demonstrate that if the goal function and temperature lowering functions meet some constraints, the SA ends its running in polynomial time and finds the optimal solution with probability almost one [6].

3 Applying SA to Musical Composition

Figure 1 presents a version of the algorithm applied to the musical composition problem.

Procedure *Simulated Annealing*
Begin
  $S \leftarrow$ random initial song
  $T \leftarrow 1$ /* initial temperature*/
  While (Quality is increasing) do
    Repeat 1000 times
      $S' \leftarrow Neighbor(S)$
      If $Quality(S') > Quality(S)$ Then $S \leftarrow S'$
      Else $S \leftarrow S'$ with probability $e^{\frac{Quality(S')-Quality(S)}{T}}$
    $T \leftarrow T \times 0.9$ /* decreases the temperature */
  $X \leftarrow S$, the final song given by the algorithm
End.

Figure 1: *Simulated Annealing Algorithm*

3.1 Basic Data Structure

Each configuration of the solution space is called a "song" and its structure is defined by the following C language declarations.

```
typedef struct { char pitch, velocity, start; } note;
typedef note song[MaxVoices][MaxTimeUnit];
```

Thus, we see a song as an array of `MaxVoices` rows and `MaxTimeUnit` columns where each entry is a triple (pitch, velocity, start). Pitch may contain either a MIDI pitch value (from 0 to 127) or a rest (represented by -1). Velocity contains a MIDI velocity value (from 0 to 127). Start may contain either 0 or 1 representing that there is a note or rest starting at that time unit (1) or not (0).

3.2 Neighbor Function

The job of the neighbor function in the SA algorithm is to receive a configuration in the solution space and to return another configuration in the same space being slightly different from the first one.

Our implementation of this function returns a song identical to the one it receives except for one note which is randomly added.

3.3 Goal Function

The purpose of our work is to develop an algorithmic tool to be used by composers and musicologists. Our idea is to offer a simple way for composers or musicologists with programming experience write their own goal functions in C, link them to our system, and then use the resource created as a MAX external object. Any goal function which receives a song as described above and returns an evaluation for its quality can be linked to our basic system.

For those with no programming experience, we have written a goal function which can have some of its parameters defined through the cells and tables of a friendly MAX patch. In the next section we describe the implementation of this goal function which can be taken as an example by other people interested on using our system with their own goal functions.

4 Implementation of MaxAnnealing

We have created a program called MaxAnnealing which uses the SA algorithm in order to find the optimal solution for a compositional problem. The program was implemented in the OPCODE's MAX 2.5.2 environment [5] since MAX offers a series of handful tools for manipulating all the music and graphic data needed by the program. An external MAX object called `annealing` which performs the optimization was created using the ThinkC 5.0.3 compiler.

The program has three basic modules: 1) the parameters interface, which allows the user to set the parameters that will be used for the song evaluations ; 2) the simulated annealing object, which performs the search for the optimal solution; 3) the player, which receives the song produced by the annealing algorithm and plays it through a MIDI output.

The current implementation of MaxAnnealing generates a song of sixteen bars, each bar consisting of four beats, each of which having up to four subdivisions, which we call "time units". The piece is distributed by four different tracks or voices which can be assigned to four different MIDI channels. All these values can be changed through modifying the annealing object source code. The printout of the source code can be found in appendix B.

4.1 The Parameters Interface

The program starts with the user providing tree general parameters for the piece. The first one is a tension curve which determines the pitch tension at each time unit of the piece. The pitch tension measures how dissonant are the simultaneous intervals that occur at each point of the music. This parameter is set by drawing a curve where each point represents the tension for each time unit in the piece as shown in Figure 2.

A second curve determines the density parameter. In this case, each point in the curve corresponds to the number of notes that should sound at each moment.

Finally, the user assigns weights to every pitch class of the chromatic scale. Here, a weight 0 means that the note should seldom occur. This brings a generic character in the pitch domain since the user can determine which scales he wish to use and create a pitch hierarchy by assigning high weights to certain pitches and low weights to others.

4.2 The Simulated Annealing Object

All data set by the user is then given to the simulated annealing module which starts the optimization process. It begins with a piece of music composed by just one arbitrary four-note chord where the voices are distributed from the highest to the lowest note through the tracks one to four (any other initial song

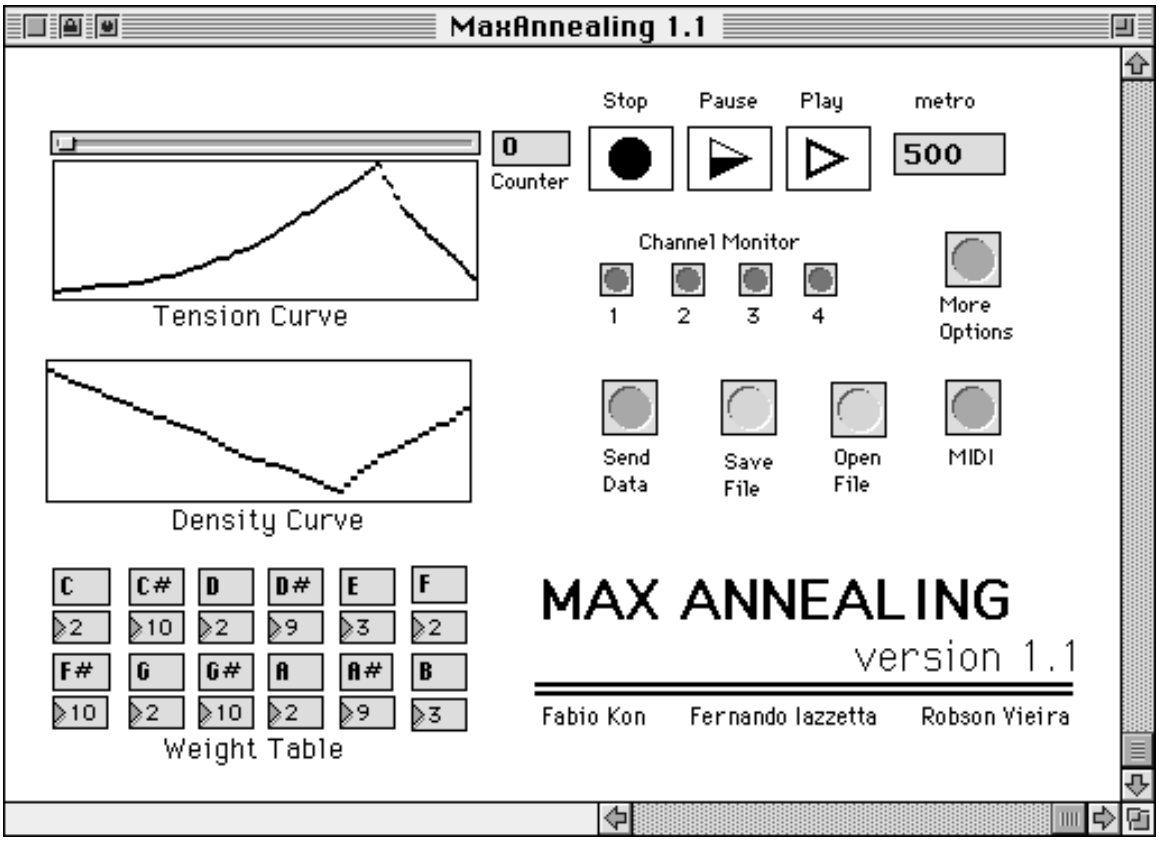


Figure 2: The MaxAnnealing Patch

would fit here). This is the first song evaluated by the algorithm. Given a song, its quality is evaluated by the goal function which is the weighted average of the five following criteria.

1. For each time unit, it calculates the tension among the intervals generated according to a previously established table of dissonance. To calculate the tension, MIDI notes are transformed in pitch classes and then the algorithm verifies all intervals that occur among the four voices for each time unit. These intervals are translated into previously established values which represent a dissonance level. The pitch tension for each time unit is calculated by adding these values.

   The closer is the pitch tension to the tension determined by the user for that time unit, the higher is its evaluation. The final evaluation for the whole song is the average of its time unit evaluations.

2. Scanning the song notes, it verifies if the density in each moment of the piece is compatible with the density curve drawn by the user. Besides, Lower pitches receive higher evaluation if they have longer durations than higher pitches. This would lead to a music structure in which lower pitches will be associated to long durations and higher pitches will be associated to short durations.

3. For each of the four voices, it evaluates the leaps between each note and its antecedent. Leaps close to a minor third receive higher evaluation than bigger and smaller ones. This will assure that the melodic contour of each voice will not have too many large leaps nor too many small intervals.

4. The more crossings between voices a song has, the lower is its evaluation. By doing so we try to avoid too many voice crossings.

5. It scans every song note adding their pitch class weights - given by the weight table set by the user. The higher is the sum of the weights, the higher is the song evaluation.

As we have seen in section 2 and 3 the value given by the goal function is compared with the quality of a previous song. Then the program decides if it takes the new song as the new temporary solution, or if it keeps the last value.

4.3 The Player

This is a simple module which receives the best result obtained by the simulated annealing and translates this data so it can be played as a four-channel MIDI sequence. It presents some standard control buttons including play, pause, stop, metro, and save and load sequence.

5 Results

We have run MaxAnnealing for several times, using different parameters to test its performance. Initially, we have set the program to search through an average of 9,200 pieces of music for each new parameter configuration. This process took about 30 seconds in a shared SUN SPARCserver 1000 and one minute in a Macintosh Performa 630. Further tests have shown that, in some cases, MaxAnnealing was able to find very good solutions after a search through only 2,000 songs, which lowers considerably the necessary time to run the program. It is worth noting that even searching through 9,200 different songs, the algorithm runs almost 2,000 times faster than that which searches through all the 2^{24} possible solutions for our particular problem.

For each test we have set different parameters in order to generate specific kinds of musical output. By assigning certain weights to the notes in the pitch table we were able to generate pieces of music based on different modes and scales. The curves also provided an easy method of control over the tension and density of events which happened at each point of the song. After only a few experiments with these controls we could make satisfactory predictions about the general characteristics of the music MaxAnnealing was going to generate as the best solution.

Although our intention was only to demonstrate the validity of using the SA technique in the solution of musical problems, and despite the simplicity of the compositional rules applied, MaxAnnealing has produced some interesting musical results. Appendix A presents the score of a composition produced by our system.

6 Conclusion and Further Work

We have introduced the use of simulated annealing algorithm as a powerful tool in the fields of musical composition and musicology. The Simulated Annealing has shown to be very effective in the search for a satisfactory solution for problems involving a large number of possible musical configurations in a very reduced time span.

With modifications in its objective function, the system would be very useful for finding the solution of many other compositional problems. Moreover, one can conceive the utilization of the SA as a practical tool in the field of musicology, which would enable the verification of relations between the formal rules which govern a piece of music and the actual effects - in terms of auditory experience - those rules generate.

As a further step on this work, we intend to test the use of other objective functions and develop the MaxAnnealing user interface to allow the generation of more complex compositional systems, which, we believe, will lead to more interesting musical results. These developments include new configuration options to be set by the user and the introduction of more elaborated compositional constraints in the program functions.

The MaxAnnealing source code and binaries are available by anonymous FTP at ftp.ime.usp.br, directory pub/macintosh/MaxAnnealing .

7 Acknowledgments

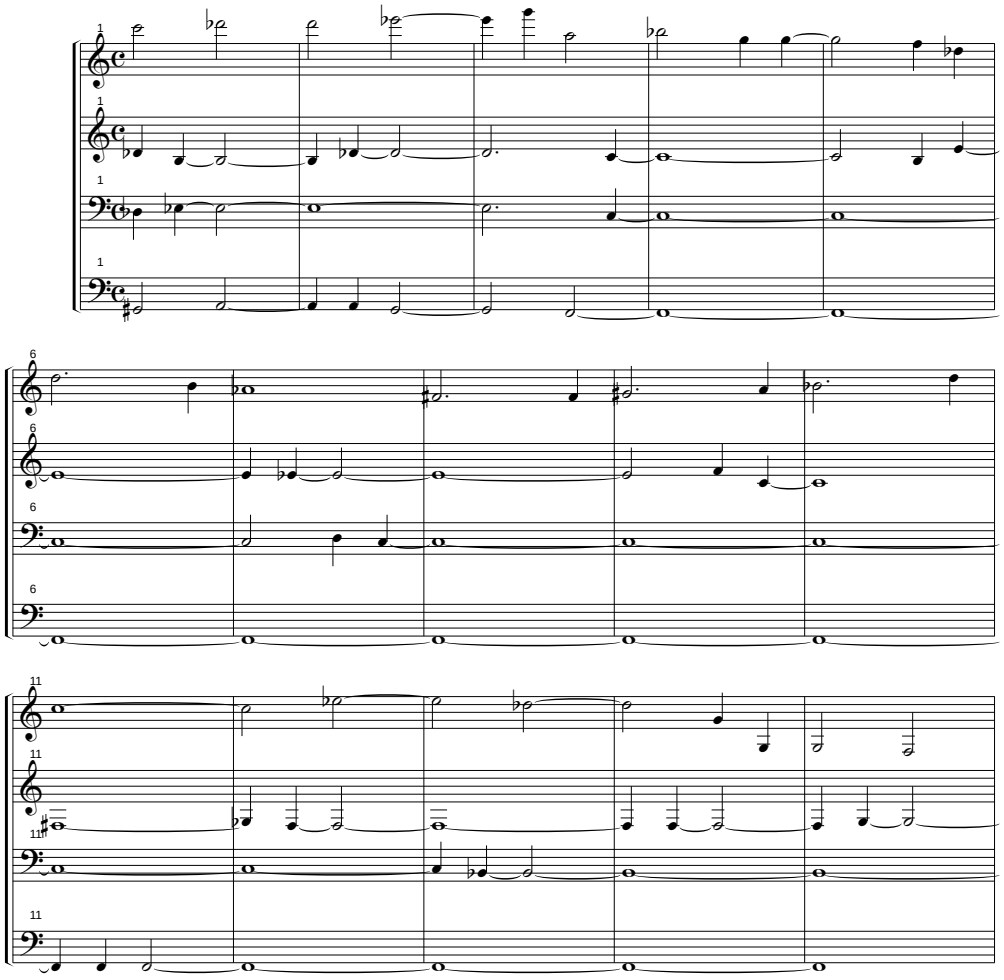
The authors gratefully acknowledge the help provided by Robson Feichas Vieira (IME/USP) throughout the development of the computational system. Alvaro L. S. Nunes provided us with his implementation of the Simulated Annealing Algorithm to a Graph Theory problem.

This work was supported by CNPq (process # 200124/94-3), FAPESP (process # 93/0603-1), and CNMAT.

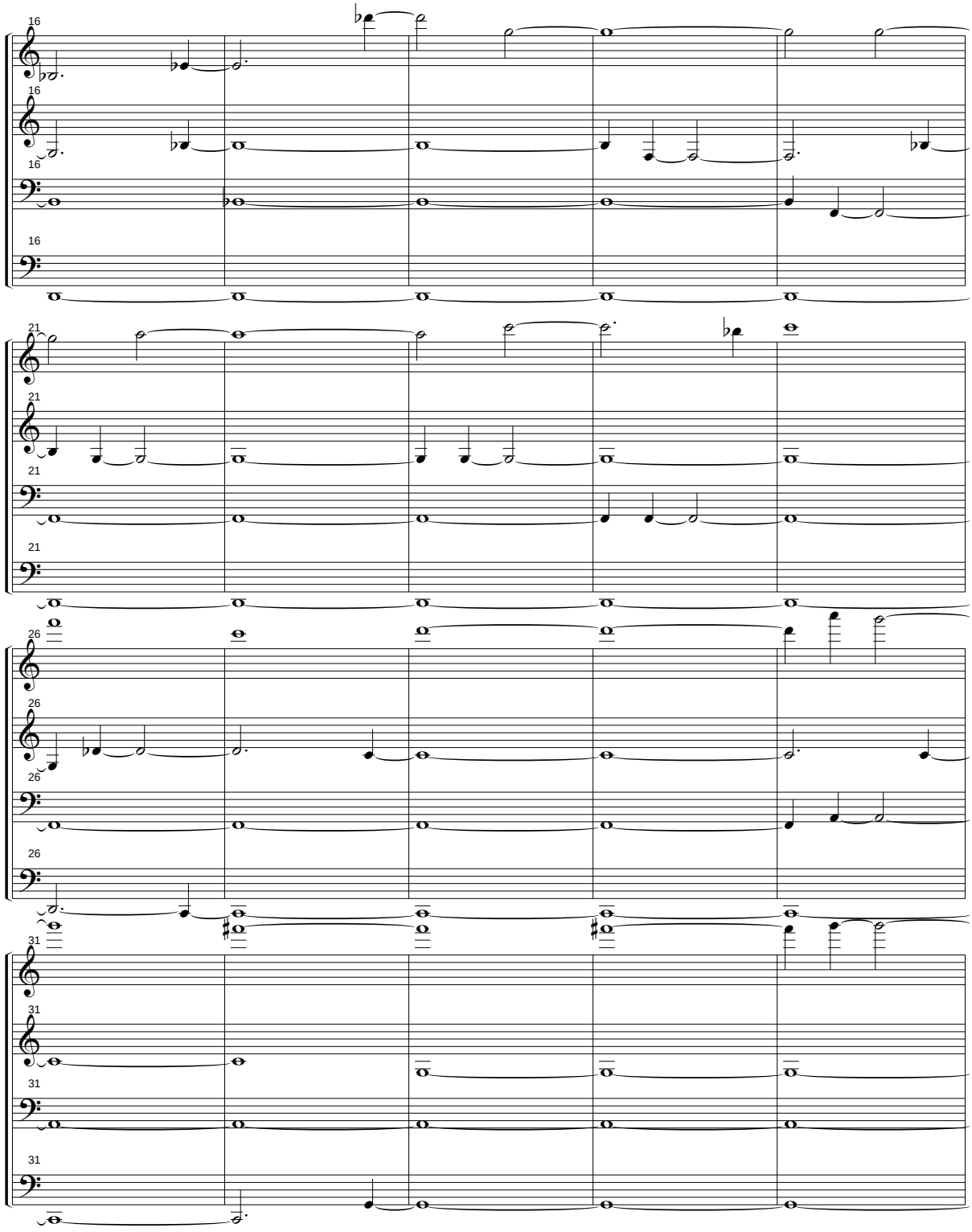
A A Musical Example

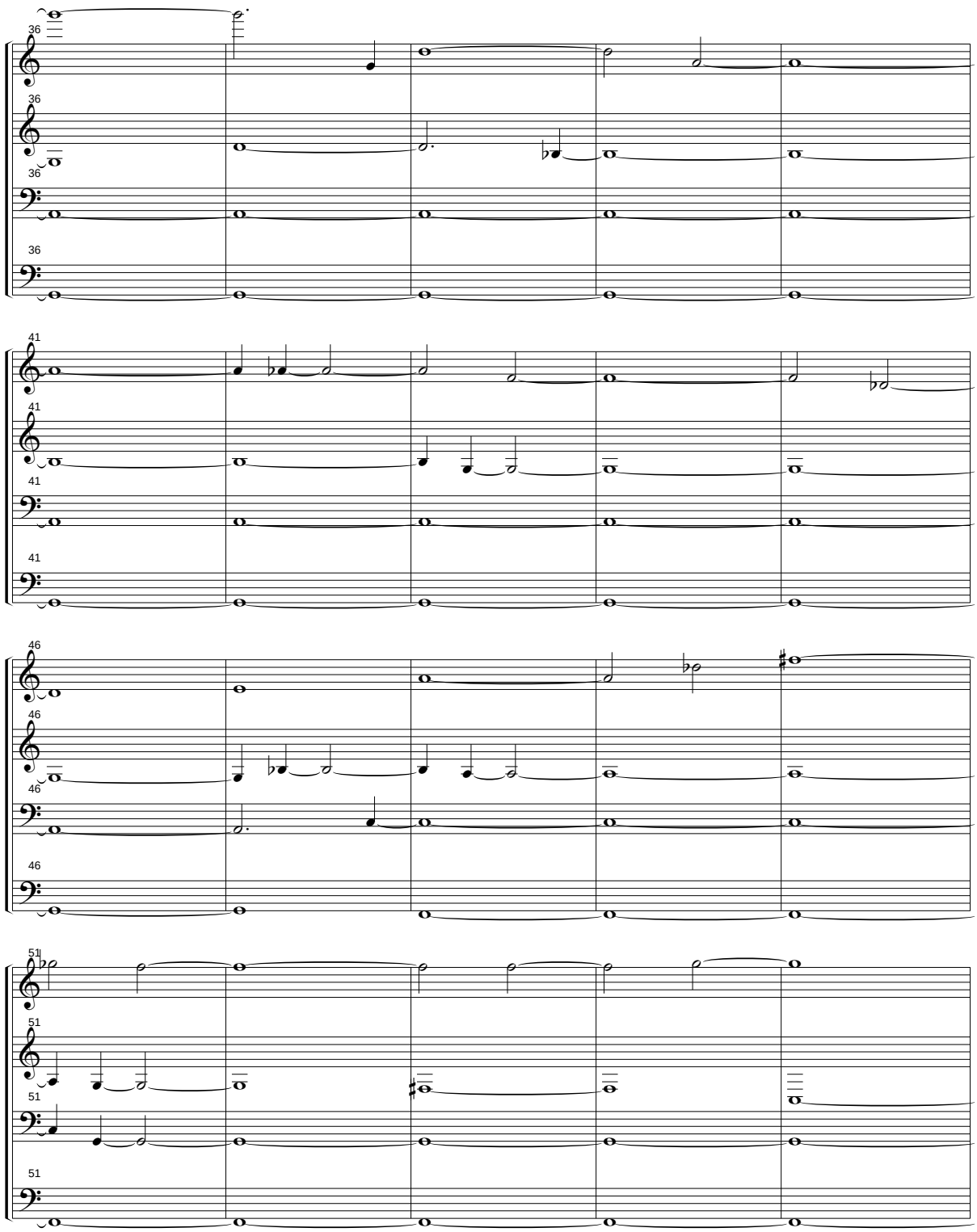
MaxAnnealing

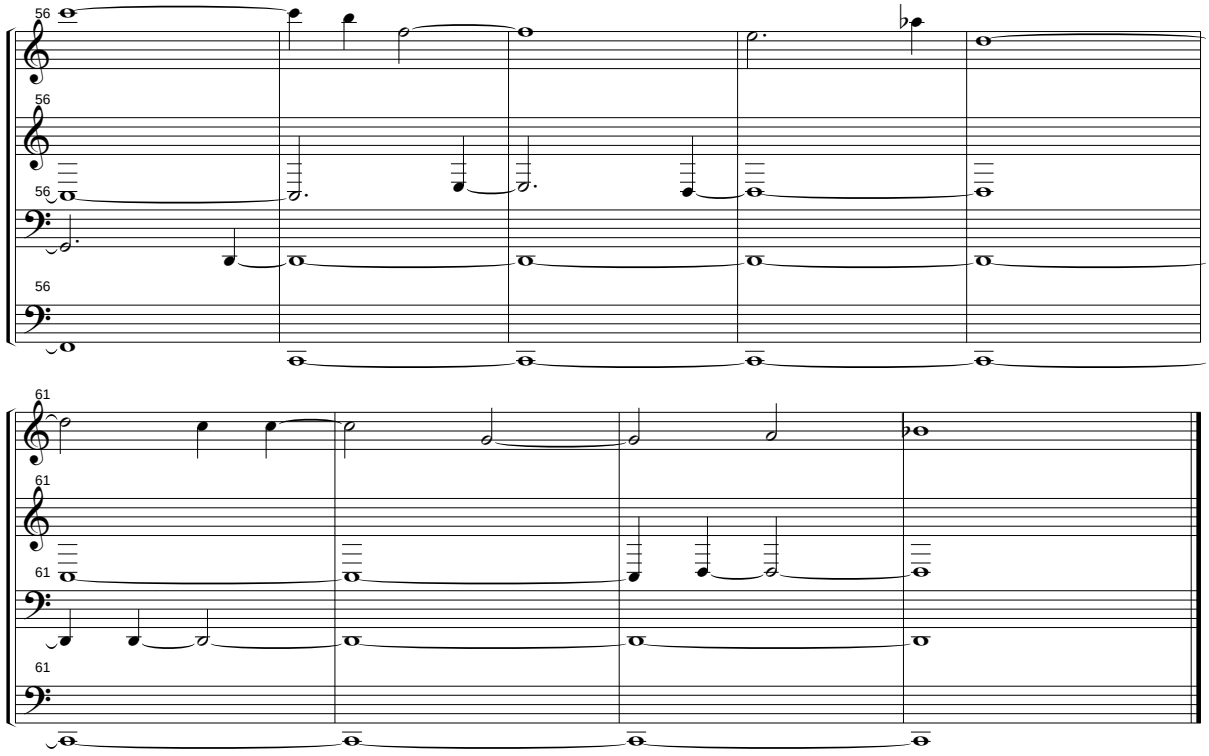
Example 1



-1-




36
36
36
36
41
41
41
41
46
46
46
46
51
51
51
51


56
56
56
56
61
61
61
61

B MaxAnnealing Source Code

```
/********************** File: maxannea.h **************************/
/* We are sorry but this is a bilingual code. */

#include "ext.h"
#include "typedefs.h"
#include "SetUpA4.h"

#define True 1
#define False 0
#define MaxIt 1000
#define MaxVozes 4
#define MaxSemiColcheias 255
#define REST -1
#define RAND_MAX 32767
#define MaxCruzamentos 6
#define MaxTensao 66

typedef struct _nota Nota;
struct _nota { int pitch, velocity, start; };
typedef Nota Partitura[MaxVozes][MaxSemiColcheias];
/* pitch and velocity are the MIDI values, start is a boolean variable that
   tells whether the note start at this moment or not. */

extern int P1, P2, P3, P4, P5;
extern int TENSAO_USUARIO[MaxSemiColcheias];
extern int qualidade(Partitura *X);
extern int DURACAO_ADEQUADA[MaxVozes][MaxSemiColcheias];
extern int pesos_das_notas[];
extern int tensao( Nota *n1, Nota *n2, Nota *n3, Nota *n4);
extern int cruzamento( Nota *n1, Nota *n2, Nota *n3, Nota *n4);
extern int aponta_escala[13];
extern fptr *FNS;

void inicializa(Partitura *);
void copia(Partitura *, Partitura *);
void vizinho(Partitura *, Partitura *);

typedef struct esqueleto {
        struct object e_ob;
        Atom e_av[MaxSemiColcheias];
        int e_ac;
        void *e_out;
} Esqueleto;

/****************** File: maxannealing.c **************************/
#include "maxannea.h"

extern Partitura X;

fptr *FNS;

void *esqueleto_new(), *MostraList();
void *class;

void out(Esqueleto *x, Partitura *score);

main(fptr f)
{
  RememberA0();
  SetUpA4();
  FNS = f;

  setup ( &class, esqueleto_new, 0L,
         (short) sizeof(struct esqueleto),
         0L, A_DEFLONG, 0);
  addmess(MostraList,"list", A_GIMME, 0);
```

```
  finder_addclass ( "Lists","maxannealing");
  post("Maxannealing Installed");
  RestoreA4();
}

void *esqueleto_new()
{
  Esqueleto *x;
  SetUpA4();
  x = (Esqueleto *)newobject(class);
  x->e_out = listout(x);
  RestoreA4();
  return (x);
}

void *MostraList(register Esqueleto *x, Symbol *s,
                 int ac, Atom *av)
{
  register int i,k;
  long n;

  SetUpA4();

  if(ac == 12)
    for (i=0; i < 12; i++) {
      if (av[i].a_type==A_LONG)
        pesos_das_notas[i] = (int) av[i].a_w.w_long;
      else post("annealing object: wrong list element type");
  }
  else if(ac == 5)
    if (av[i].a_type==A_LONG) {
          P1 = (int) av[0].a_w.w_long;
          P2 = (int) av[1].a_w.w_long;
          P3 = (int) av[2].a_w.w_long;
          P4 = (int) av[3].a_w.w_long;
          P5 = (int) av[4].a_w.w_long;
    } else post("annealing object: wrong list element type");

  else if(ac == 255)
    for (i=0; i < 255; i++) {
      if (av[i].a_type==A_LONG)
        TENSAO_USUARIO[i] = (int) av[i].a_w.w_long;
      else post("wrong list element type");
    }

  else if(ac==64)
   {
    for (k=0,i=0; i < 64; i++)
      if (av[i].a_type==A_LONG) {
        DURACAO_ADEQUADA[MaxVozes-1][k++] = DURACAO_ADEQUADA[MaxVozes-1][k++] =
        DURACAO_ADEQUADA[MaxVozes-1][k++] = DURACAO_ADEQUADA[MaxVozes-1][k++] =
        (int) 8 - av[i].a_w.w_long;
      } else post("wrong list element type");

    inicializa(&X);
    find_best_song(x);
    out(x,&X);
   }
  else post("list of wrong length");
  RestoreA4();
}

void out(Esqueleto *x, Partitura *score)
{
 int register k = 0, i, j;

 for(i=0;i<MaxVozes;i++)
  for(j=0;j<MaxSemiColcheias;j++)
```

```
   {
    SETLONG(&x->e_av[0],(long)(*score)[i][j].pitch);
            SETLONG(&x->e_av[1],(long)(*score)[i][j].velocity);
            SETLONG(&x->e_av[2],(long)(*score)[i][j].start);
            if (!outlet_list(x->e_out, 0L, (int) 3, x->e_av))
             post("error while sending data to outlet");
   }
 post("Song created");
}

/********************** File: principal.c **************************/
#include "maxannea.h"

extern fptr *FNS;
int P1, P2, P3, P4, P5;
Partitura X;
float myexp(float x);
int  myrand();

float Esfria(float Temperature)
{
  return(Temperature*0.9);
}

int Aceita(float Temperature, float Variation)
{
  float aux,P,Q;

  if (Temperature != 0.0)
    aux = Variation/Temperature;
  else
    aux = 0;
  P = 1.0/myexp(-aux);
  Q = (float)myrand();
  Q = Q/32767.0;
  return (Q < P);
}

Partitura Xmax, Xnovo;

int find_best_song(x)
Esqueleto *x;
{
  long  Qualidade_max,  Qualidade_Atual,
        Qualidade_Nova, Variacao;
  float Temperature;
  int   Cont, Iteracoes, Aumentou, i;

  P1 = P4 = 2;  /* Initial weights for the 5 criteria */
  P5 = 3;
  P3 = 1;
  P2 = 1;

  inicializa(&X);                 /* Generate initial song */
  Qualidade_Atual = qualidade(&X);
  copia(&Xmax,&X);
  Qualidade_max   = Qualidade_Atual;
  Aumentou        = True;
  Iteracoes       = 0;
  Temperature     = 100.0;

  while (Aumentou)   /* While the quality is increasing... */
    {
     Cont = MaxIt;
     Aumentou = False;
     for(Cont = MaxIt; Cont > 0; Cont--)
       {
        Iteracoes++;
```

```
        vizinho(&Xnovo, &X); /* Pick a new song in the neighborhood */
        Qualidade_Nova = qualidade(&Xnovo);
        Variacao = Qualidade_Nova - Qualidade_Atual;
        if (Variacao > 0)
          {
           copia(&X,&Xnovo);
           Qualidade_Atual = Qualidade_Nova;
           if (Qualidade_Nova > Qualidade_max)
             {
              copia(&Xmax, &Xnovo);
              Qualidade_max = Qualidade_Nova;
             }
           Aumentou = True;
          }
        else
          {
           if (Aceita(Temperature,Variacao))
            {
             copia(&X, &Xnovo);
             Qualidade_Atual = Qualidade_Nova;
            }
          }
       }/* for */
     Temperature = Esfria(Temperature);
    }
}

long semente = 1234567;
/* Since the ANSI library used to create external objects does not
   have the rand and the exp functions, we had to write our own. */

int myrand()
{
  semente =((semente*1103515245+12345)>>1)&2147483647;
  return (int) semente&32767;
}

float myexp(float x)
{
  /* Uses the Taylor Series in order to give a good aproximation of
     exp(x) if x is something between 0 and 5. We are using the
     following expression: (1+x+x*x/2+x*x*x/6+x*x*x*x/24+x*x*x*x*x/120
                           +X^6/720 +X^7/5040 +X^8/40320 +X^9/362880
                           +X^10/3628800);
  */
  return (1.0+x*(1.0+x*(0.5+x*(0.16666667+x*(0.0416667+x*(1.0/120+x*
         (1.0/720+x*(1.0/5040+x*(1.0/40320+x*(1.0/362880+x*(x/3628800)))))))))));
}

/********************** File: quali.c ***************************/

#include <stdlib.h>
#include "maxannea.h"
int TENSAO_USUARIO[MaxSemiColcheias];

int tensao( Nota *n1, Nota *n2, Nota *n3, Nota *n4)
{
  static int tensao_intervalar[12] = {0, 10, 7, 6, 5, 2, 9, 1, 4, 3, 8, 11};
  int t=0;

  if(n1->pitch != REST )
    {
      if(n2->pitch != REST) t+=tensao_intervalar[abs((*n1).pitch - (*n2).pitch)%12];
      if(n3->pitch != REST) t+=tensao_intervalar[abs((*n1).pitch - (*n3).pitch)%12];
      if(n4->pitch != REST) t+=tensao_intervalar[abs((*n1).pitch - (*n4).pitch)%12];
    }
  if(n2->pitch != REST )
    {
```

```
      if(n3->pitch != REST) t+=tensao_intervalar[abs((*n2).pitch - (*n3).pitch)%12];
      if(n4->pitch != REST) t+=tensao_intervalar[abs((*n2).pitch - (*n4).pitch)%12];
    }
  if(n3->pitch != REST && n4->pitch != REST)
    t+=tensao_intervalar[abs((*n3).pitch - (*n4).pitch)%12];
  return (t);
}

int cruzamento( Nota *n1, Nota *n2, Nota *n3, Nota *n4)
{
 int j=0;

 if(n1->pitch != REST )
   {
     if((*n1).pitch > (*n2).pitch && n2->pitch != REST) j+=n1->pitch - n2->pitch;
     if((*n1).pitch > (*n3).pitch && n3->pitch != REST) j+=n1->pitch - n3->pitch;
     if((*n1).pitch > (*n4).pitch && n4->pitch != REST) j+=n1->pitch - n4->pitch;
   }
 if(n2->pitch != REST )
   {
     if((*n2).pitch > (*n3).pitch && n3->pitch != REST) j+=n2->pitch - n3->pitch;
     if((*n2).pitch > (*n4).pitch && n4->pitch != REST) j+=n2->pitch - n4->pitch;
   }
 if(n3->pitch != REST && n4->pitch != REST && n3->pitch > n4->pitch)
   j+=n3->pitch - n4->pitch ;
 return (j);
}

int salto(int n1, int n2)
{
 if(n1 == REST || n2 == REST) return(0);
 return(abs(n1-n2));
}

int qualidade(Partitura *X)
{
  long tensao_da_musica = 0, duracao         = 0,
       duracao_voz      = 0, cruzamentos     = 0,
       saltos           = 0, soma_dos_pesos  = 0;
  int nota_atual        = 0, nota_seguinte   = 0,
      semi, dur, voz,        numero_de_notas = 0;

 /* Harmonic Analysis (vertical) */
  for (semi=0; semi < MaxSemiColcheias; semi++)
    {
      tensao_da_musica += abs(tensao(&(*X)[0][semi],&(*X)[1][semi],
                                     &(*X)[2][semi],&(*X)[3][semi])-
                              TENSAO_USUARIO[semi]);
      /* This only works if the song has 4 voices. We must generalize
      this in the future */
      cruzamentos += cruzamento(&(*X)[0][semi],&(*X)[1][semi],
      &(*X)[2][semi],&(*X)[3][semi]);
    }

  /* Horizontal Analysis */
  for (voz=0; voz < MaxVozes; voz++)
    {
     semi = 0;
     duracao_voz = 0;

     while(semi<MaxSemiColcheias)
      {
       nota_atual = (*X)[voz][semi].pitch;
       numero_de_notas++;
       dur = 1;
       semi++;
       while ((*X)[voz][semi].start == 0 && semi<MaxSemiColcheias)
         {
```

```
          dur++;
          semi++;
         }
       duracao_voz += abs(dur-DURACAO_ADEQUADA[voz][semi-1]);
       soma_dos_pesos += dur*pesos_das_notas[(nota_atual%12)+1];

       if(semi<MaxSemiColcheias)
         {
          nota_seguinte = (*X)[voz][semi].pitch;
          saltos += salto(nota_atual, nota_seguinte);
         }
      }
     duracao += duracao_voz*(voz+1);
    }

  soma_dos_pesos *=10;
  soma_dos_pesos /= 1024;                          /* between 0 and 100 */
  tensao_da_musica *=100;
  tensao_da_musica /=(MaxSemiColcheias*MaxTensao); /* between 0 and 100 */
  cruzamentos *=100;
  cruzamentos /=(MaxSemiColcheias);                /* between 0 and 100 */
  saltos *=10;
  saltos /= (numero_de_notas);                     /* between 0 and 100 */
  duracao *=100;
  duracao /= (2*MaxVozes*MaxSemiColcheias);        /* between 0 and 100 */
  return ((int)
          (P1*(100-tensao_da_musica)+
           P2*(100-cruzamentos)+
           P3*(30-labs(saltos-30))*33/10+ /* Incentivamos saltos proximos
                                             da terca menor */
           P4*(100-duracao)+
           P5*soma_dos_pesos));
}

/********************** File: buro.c ***************************/

#include <stdlib.h>
#include <string.h>
#include "maxannea.h"

int DURACAO_ADEQUADA[MaxVozes][MaxSemiColcheias];
int pesos_das_notas[13];
int melhor_nota=0;

void inicializa (Partitura *X)
{
 int i,voz,semi;

 for (i=0;i<12;i++)
   if (pesos_das_notas[i] > pesos_das_notas[melhor_nota]) melhor_nota = i;
 for (voz=0; voz<MaxVozes; voz++)
   for (semi=0; semi<MaxSemiColcheias; semi++) {
     if (semi%16==0) {
       (*X)[voz][semi].pitch = melhor_nota+(voz+3)*12;
       (*X)[voz][semi].velocity = 100;
       (*X)[voz][semi].start = 1;
     }
     else {
       (*X)[voz][semi].pitch = melhor_nota+(voz+3)*12;
       (*X)[voz][semi].velocity = 100;
       (*X)[voz][semi].start = 0;
     }
   }
 for (semi=0; semi<MaxSemiColcheias; semi++) {
   DURACAO_ADEQUADA[2][semi] = DURACAO_ADEQUADA[3][semi]<<1;
   DURACAO_ADEQUADA[1][semi] = DURACAO_ADEQUADA[2][semi]<<1;
   DURACAO_ADEQUADA[0][semi] = DURACAO_ADEQUADA[1][semi]<<1;
 }
```

```
}

void copia(Partitura *X, Partitura *Y)
{
  memcpy(X, Y, sizeof(Partitura));
}

void vizinho (Partitura *Y, Partitura *X)
/* Returns in Y, a neighbor of X */
{
 int pitch, duracao, voz, semi, oitava, k, l, i, temp = 1;

 voz     = (int) myrand()&3;
 semi    = (int) myrand()&255;
 pitch   = (int) myrand()%12;
 temp = (DURACAO_ADEQUADA[voz][semi]<<1)+1;
 duracao = (int) myrand()%temp;

 copia(Y, X);
 if (pitch != REST) {
   if (semi == 0) i = (*Y)[voz][semi].pitch;
   else           i = (*Y)[voz][semi-1].pitch; /* i is the previous note */

   if (i == REST) i = melhor_nota+(voz+3)*12;
   oitava = ( int) i/12 -1;   /* We choose a pitch near the pitch of i */
   pitch += oitava*12;        /* preserving the selected pitch class.  */
   while (abs(i-pitch) > 6)
      if ( i > pitch) pitch += 12;
      else pitch-= 12;
   while(pitch < 36) pitch +=12;
   while(pitch > 96) pitch -=12;
 }

 (*Y)[voz][semi].pitch = pitch;
 (*Y)[voz][semi].velocity = 100;
 (*Y)[voz][semi].start = 1;
 for(k=1; k<duracao && semi+k<MaxSemiColcheias; k++) {
     (*Y)[voz][semi+k].pitch = pitch;
     (*Y)[voz][semi+k].velocity = 100;
     (*Y)[voz][semi+k].start=0;
 }
 if (semi+duracao<MaxSemiColcheias && !(*Y)[voz][semi+duracao].start)
   (*Y)[voz][semi+duracao].start = 1;
}
```

References

[1] Kirkpatrick S., C.D. Gelatt Jr. and M.P. Vecchi. "Optimization by simulated annealing", *Science*, 220, pp. 671-680, 1983.

[2] Roads, Curtis, "Interview with Marvin Minsk", *Computer Music Journal*, 4 (3), 1980.

[3] Ames, Charles, "Protocol: Motivation, Design, and Production of a Composition for Solo Piano", *Interface*, 11, pp. 213-238, 1982.

[4] Schottstaedt, W. "Automatic Counterpoint", *Currents Directions in Computer Music Research*, Max Mathews and John Pierce (Eds.). Cambridge: The MIT Press, 1989.

[5] Puckett, M., D. Zicarelli, *MAX - An Interactive Graphic Programming Environment*, Opcode Systems, Menlo Park, CA, 1990.

[6] Mitra, D., Romeo, F., Sangiovanni-Vicentinelli, Alberto, "Convergence and Finite-Time Behavior of Simulated Annealing", *Advanced Applied Probability*, 18, pp. 747-771, 1986.