A wizard's aid:
efficient music constraint programming with Oz

Torsten Anders

Music, Mind and Machine, University of Nijmegen, The Netherlands

`t.anders@nici.kun.nl`

This article proposes an efficient search strategy for constraint based computer assisted composition. The strategy uses the *propagate and distribute* technique and always proceeds in score-time, even if the rhythmic structure of the score is not known before search.

1 Introduction

Constraint programming has become increasingly popular for computer assisted composition, because it allows the composer to generate a musical score by describing a desired result. However, he does not need to specify, *how* to achieve this outcome. Using the constraint paradigm the composer can define arbitrary relations between score parameters. The system then searches for one or more solutions.

Score parameters may be interrelated by constraints on different aspects such as rhythmic rules, harmonic rules, voice leading rules etc. To find a solution fulfilling all constraints in an efficient way, the search should proceed in score-time, i. e. by completion of parallel score entities in parallel and not sequentially. However, if the search computes the rhythmic structure too, the search order cannot be determined before the search itself: the program can not guess in advance which score objects can be processed in parallel.

This article proposes a complete search strategy which uses the *propagate and distribute* technique and always proceeds in score-time, even if the rhythmic structure of the score is not known before search. The strategy is implemented in the Oz programming language [10, 13, 6].

Plan of the paper: The section 2 presents some examples of constraint programming packages for music composition and their respective search strategies. Constraint programming capabilities of Oz are introduced in section 3. Section 4 proposes an approach how to specialize this means for music composition.

2 Music constraint programming

Several working systems and theoretical studies apply constraint programming to various musical applications. The following section introduces a few systems for computer assisted composition. Special attention is paid to their respective search strategies.

Each program is integrated in a composition environment. Environment editors can be used for inspecting and manipulating input and output.

2.1 PWConstraints

PWConstraints [5] allows the user to freely declare constraints on the MIDI pitches of a given rhythmical score. Constraints are defined very concisely using a pattern matching language. PWConstraints exists as a library integrated in PatchWork [2].

The search engine of PWConstraints uses a backtracking search over finite domains of integers. Because of the predefined rhythmic structure, a search order proceeding in score-time is established by the program before search. Forward checking and heuristics (applied to the order of domains) can be used, while weighted constraints help to handle over-constraint problems.

In newer extended versions other note parameters can also be set, rhythmic values for instance. However, this program still requires a rhythmical input.[1]

2.2 Arno

The environment Arno [1] extends Common Music [11] by the means of constraint programming. In Arno arbitrary parameters of Common Music elements can be declared nondeterministically using finite domains of arbitrary values. Constraints – expressed as unary functions on a Common Music object – restrict the actual slot values of that object. To define n-ary relations the score API of Common Music and its extensions supplied by Arno can be used.

Arno applies the search engine from Screamer [9], i.e. it uses backtracking search. Back jumping and heuristics can be applied. However, the performance of Arno degrades for longer, non-monophonic examples. The program is developed to generate the rhythmic structure of polyphonic music too. It can therefore not guess in advance, which events will be simultaneous in time. Hence

---

[1]Internally, search variables are still encoded as pitches. These may be converted to change the rhythmic structure.

the search does not proceed in score-time but container-wise and performs redundant work.[2]

2.3 Adaptive search

A constraint extension for OPENMUSIC [2] based on an adaptive search is currently in development [12, 3]. The system performs a local improvement strategy which quickly computes approximate solutions. The user can observe the refinement of the search and stop the process when the current solution is adequate.

Constraints, expressed in logical form, are automatically translated into cost functions to guide the search progression.

Because adaptive search does not execute a complete search, constraints such as, e.g., `allDifferent`,[3] are hard to fulfil strictly by this strategy. However, `allDifferent` is a very important constraint for composers. Therefore the system uses a forward checking technique instead of an adaptive search to solve this constraint.

3 Constraint programming in Oz

The Oz programming language offers a rich set of programming paradigms combined in one simple model with a sound theoretical foundation [10, 13]. Supported paradigms include functional programming, logic and constraint programming, stateful programming, object-oriented programming, concurrency and distributed programming.[4]

The logic and constraint programming capabilities of Oz subsumes both search-based logic programming (*don't know* nondeterminism) and concurrent logic programming (*don't care* nondeterminism). The definition of a search problem and the definition of a search strategy are independent of each other in Oz (unlike as, e.g., in PROLOG). The language offers high level means to program search strategies.

Oz has already been applied to music composition. The projects COMPOzE [4] and COPPELIA [14] realize automated composition. Both projects are restricted to four-voiced music.

3.1 Propagate and distribute

The following subsection introduces how search-based constraint programming is done in Oz. It concentrates on finite domain constraints [8].[5]

The constraint solver of Oz uses a technique called *propagate and distribute*. This name is based on the two inference rules of the method: *constraint propagation* and *constraint distribution*.

Computation with constraints takes place in a *computation space*. The computation space contains a *constraint store* which stores a conjunction of *basic constraints* on logic variables.[6] A basic constraint of a finite domain problem takes either the form $x = n$ (n being an integer or another variable) or $x \in D$ (D is a finite domain).

Non-basic constraints (as "$x < y$" or "all values of $x_1, \ldots, x_n$ are distinct") are imposed by propagators. Propagators are concurrent agents. A propagator will narrow a variable domain by propagating a new basic constraint, if the basic constraint is entailed by both the constraint store and the propagator. It thus reduces the search space. However, constraint propagation does not necessarily lead to a solution (or a fail).
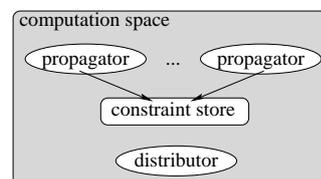
A *distributor* is a concurrent agent which waits until its hosting computation space S becomes stable (i.e. no further propagation is possible). It then creates two new computation spaces by executing a binary choice statement. The new spaces inherit all basic constraints and propagators of S. Additionally an arbitrary constraint C is added to one and its complement, $\neg C$, to the other space. Adding these constraints does not change the solution. It may however restart propagation. The combination of constraint propagation and distribution yields a complete solution method for finite domain constraint problems.

Distribution techniques differ in what constraint is added to which variable. In Oz, a few standard distribution strategies exist. The user can also define new strategies. The best (e.g. fastest) strategy depends on the problem. A typical distribution technique is *first fail*: this strategy selects resprespectively removes the leftmost domain value of a variable with the smallest domain.

Distribution defines the shape of the search tree. This tree can be explored by different search strategies to find a single, several, all or the best solution,[7] according to some criterion. The search can also be interactively guided by tools as, e.g., the Oz Explorer [7].

Propagation, complemented by distribution, has a better performance than a simple backtracking search. Depending on task, this strategy also performs better than

---

[2] For polyphonic music that can be reduced to monophonic music (e.g. canons) the performance is much better.

[3] All elements are pairwise distinct.

[4] The Mozart system – an implementation and interactive development environment of the Oz programming language – is cross-platform compatible; implementations exists for UNIX (including Linux and MacOS X) and MS Windows [6]. Mozart is open and free software, it is published under a *X11 style* license.

[5] A finite domain consists of non-negative integers in Oz, which can be mapped to arbitrary discrete data.

[6] Logic variables are single-assignment variables and bound by unification as, e.g., in Prolog.

[7] Best solution search uses a binary procedure. After finding a solution it constrains the next solution to be better than the previous according to that relation.

adaptive-search (see section 2.3). A constraint like `allDifferent` is hard to solve by means of a local search only. Constraint propagation is optimally suited to this task.

3.2 All-interval series

The following subsection illustrates constraint propagation and distribution by computing an all-interval series.[8] The figure 1 shows the full search tree for all solutions of length 4. Each tree node represents a new computation space, introduced by distribution. Solved spaces are drawn as diamonds $\diamond$, failed spaces as boxes $\square$ and distributable spaces as circles $\bigcirc$.
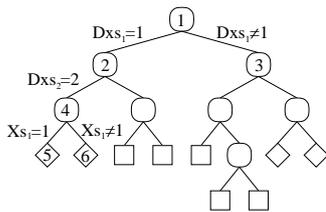
Space	Xs	Dxs
1	$[\{0\#3\} \ldots \{0\#3\}]$	$[\{1\#3\} \{1\#3\} \{1\#3\}]$
2	$[\{0\#3\} \ldots \{0\#3\}]$	$[1 \{2, 3\} \{2, 3\}]$
3	$[\{0\#3\} \ldots \{0\#3\}]$	$[\{2, 3\} \{1\#3\} \{1\#3\}]$
4	$[\{1, 2\} \{1, 2\} \{0, 3\} \{0, 3\}]$	$[1 \ 2 \ 3]$
5	$[1 \ 2 \ 0 \ 3]$	$[1 \ 2 \ 3]$
6	$[2 \ 1 \ 3 \ 0]$	$[1 \ 2 \ 3]$

Figure 1: all-interval series: search tree for all solutions of length 4

Only 17 nodes are necessary to find all 4 solutions. The table below the tree shows the basic constraints on the elements of the solution series `Xs` and the intervals `Dxs` before their respective spaces get distributed.[9]

The constraints added by the first fail distribution strategy are shown next to the tree arcs. The strategy always affects the leftmost variable with minimal domain size. In the first space it creates a choice point by either binding the first element of Dxs to its first domain value or removing this value.

After each distribution step propagators adjust the domain of the other variables accordingly. In space 2 the propagator constraining all intervals to be distinct removes the determined interval 1 from the other two interval domains. In space 4 the propagator determines the third interval to be 3, which awakes the distance propagator to reduce the domain of last two series elements to $\{0, 3\}$. This again reduces the domain of the first two series elements as well, because all elements are constrained to be different.

---

[8]An all-interval series consists of distinctive pitch classes and distinctive intervals between the pitches. For simplicity, the example avoids equal pitch distances instead of equal or complementary pitch intervals.

[9]The notation $m\#n$ is used for the finite domain $m, \ldots, n$.

4 A distribution strategy for a score

The following section proposes how a search for an arbitrary nested score can proceed in score time, even if the rhythmic structure of the score is not known beforehand. The approach uses the finite domain constraint programming facilities of Oz. It defines a distribution strategy which always distributes the domain of an undetermined score element parameter with the smallest start time.

The search tree thus results in a dynamic search order. The most serious performance draw back of ARNO is its naïve static search order (see section 2.2). The static search order is also the reason why PWCONSTRAINTS invariably uses precomputed rhythmic scores (see section 2.1).

4.1 The score distribution strategy

To ease the definition of new distribution techniques Oz offers special procedures to specify which constraint shall be added to which variable for distribution. Such procedure expects a specification of a distribution strategy and an Oz vector containing all data to be distributed. The distribution can be specified by functional arguments.

For our purpose the variables to be distributed are the parameter values of score elements (durations, pitches etc.). We therefore use a parameter data structure containing both its value and a link to the element the parameter belongs to.[10] That way we can estimate which undetermined parameter belongs to a score element with smallest start-time.

The distribution strategy is now described in terms of the functions given to the arguments *filter*, *order* etc. of the Oz procedure `FD.distribute`.

Filter: Keep only parameters with an undetermined parameter value (once filtered out parameters will never be considered for distribution again).

Order: Find a parameter of which the start-time of the related element can be or is determined (e. g. the duration of the predecessor element is known) and whose start-time is minimal. Prefer durations over all other element parameters.

Select: Access the value variable of the parameter.

Value: Select a domain value (e. g. the minimum value of the domain).

4.2 Discussion

The proposed approach constrains only the parameters of score elements (duration, pitch etc.). The formal structure of the score needs fully be determined, e. g.,

---

[10]Of course the element (e. g. note) data structure also needs access to their parameters, i.e. bidirectional links are used.

the number of notes in a voice needs to be known before search.[11]

Naturally, constraints can only be propagated between parameter values which are already know to be related. If, e. g., a harmonic relation constrains pitches of notes simultaneous in score-time, propagation can not be performed before these notes are known to be simultaneous.[12] Therefore durations need to be determined first. Constraints relating duration parameters with other parameters can thus not always be propagated but may be satisfied by distribution. Constraints between parameters whose element relations are known (e. g. elements of a voice, or elements already known to be simultaneous) can freely propagate.

A constraint may be expressed in an unary procedure on a score element. Arbitrary relations can be defined using accessors for element parameters and accessors to other score elements. Such constraints can be mapped to all elements of a score or subsets of these (e.g., all elements of a voice).[13]

The choice of the distribution technique is independent of other search issues in Oz. Therefore, the proposed distribution does not limit the expressiveness of an Oz constraint script. Particularly a best solution search can still be performed and over-constraint problems can be handled using reified constraints[14].

5   Conclusion

Constraint programming is attractive for computer assisted composition, because the composer can define arbitrary relations between score parameters, without specifying how to achieve them.

The *Propagate and distribute* technique is an efficient and complete search strategy whose search order can be tailored. This article proposed a distribution strategy for a musical score which always proceeds in score-time, even if the rhythmic structure of the score is not known before the search.

In various situations a full search is preferable over non-complete strategies, e. g. when constraints as `allDifferent` shall be strictly fulfilled, when all solutions or the best solution with regard to a certain criterion are desired.

---

[11] A more free approach may express the elements of a container by a list with an unbound tail. By and by the search process adds elements to the list. But this way constraint propagation between the list elements is at least considerably reduced.

[12] Such a constraint needs to run in a separate thread. The thread will block until the simultaneous notes are known. It will then add its constraint to the parameters of the elements.

[13] The mapping should only define redundant constraints if necessary, e. g., to reduce the search space by eliminating symmetries. Otherwise redundant propagators impair efficiency.

[14] Reified constraints constrain the truth value of other constraints and thus make e. g. disjunction, implication, and negation possible. A truth value is expressed as a variable of the finite domain $\{0,1\}$, which allows to, e. g., maximize the number of fulfilled constraints.

6   Acknowledgments

References

[1] Tosten Anders. Arno: Constraints Programming in Common Music. In *Proceedings of the 2000 International Computer Music Conference.*

[2] Gerard Assayag, Camilo Rueda, Mikael Laurson, Carlos Agon, and Olivier Delerue. Computer Assisted Composition at IRCAM: From PatchWork to Open Music. *Computer Music Journal*, 23:3, Fall 1999.

[3] Philippe Codognet and Daniel Diaz. Yet Another Local Search Method for Constraint Solving. In T. Walsh and C. Gomes, editors, *Proceedings of the AAAI Symposium "Using Uncertainty in Computation"*. AAAI Press, 2001.

[4] Martin Henz, Stefan Lauer, and Detlev Zimmermann. COMPOzE — intention-based music composition through constraint programming. In *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence*, pages 118–121, Toulouse, France, November16–19 1996. IEEE Computer Society Press.

[5] Mikael Laurson. *PatchWork, PWConstraints*. IRCAM, Paris, first english edition, Oktober 1996. Reference manual.

[6] The Mozart Programming System. `http://www.mozart-oz.org/`.

[7] Christian Schulte. Oz Explorer – Visual Constraint Programming Support. Technical report. available at `www.mozart-oz.org`.

[8] Christian Schulte and Gert Smolka. Finite Domain Constraint Programming in Oz. A Tutorial. Technical report. available at `www.mozart-oz.org`.

[9] Jeffrey Mark Siskind and David Allen McAllester. Screamer: A Portable Efficient Implementation of Nondeterministic Common Lisp. Technical report, University of Pennsylvania Insitute for Research in Cognitive Science, 1993.

[10] Gerd Smolka. The Oz Programming Model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000. Springer-Verlag, Berlin, 1995.

[11] Heinrich Taube. An Introduction to Common Music. *Computer Music Journal*, 21:1:29–34, 1997.

[12] Charlotte Truchet, Carlos Agon, and Philippe Codognet. A Constraint Programming System for Music Composition, Preliminary Results. In *Seventh International Conference on Principles and Practice of Constraint Programming, Musical Constraints Workshop*, Paphos, Cyprus, 2001.

[13] Peter van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. To appear, `http://www.info.ucl.ac.be/~pvr/book.pdf`.

[14] Detlev Zimmermann. Modelling Musical Structures. Aims, Limitations, and the Artist's Involvement. In *ECAI 98 Workshop on Constraints for Artistic Applications*, Brighton, 1998.