Interactive Intelligent Systems Workshop: Music Constraint Programming (4)

Torsten Anders
Interdisciplinary Centre for Computer Music Research (ICCMR)
University of Plymouth
http://cmr.soc.plymouth.ac.uk/

28 November 2007
Outline

1. Motivation: Problem-Specific Search Orderings
2. The Constraint Model Based on Computational Spaces
3. Specialising the Constraint Model for Music
4. Conclusion
Why Discussing the Search Process?

- Music constraint programming greatly simplifies the implementation of complex music theory models
  - User only specifies all constraints the solution should fulfil
  - A constraint solver finds solution(s) for a constraint satisfaction problem (CSP) via search
- However, reasonably efficient search vital to make system useful in practice
- This talk discusses how musical CSPs are solved efficiently
Variable and Value Orderings I

**Variable ordering**
- Order in which variables are visited during the search process
- Variable ordering has great impact on efficiency (size of resulting search tree)
- Suitable variable ordering highly problem dependent

**Static vs. dynamic variable ordering**
Variable ordering is either fixed before the search starts (static), or computed during the search process (dynamic)
Variable and Value Orderings II

**First-fail principle**

- Common principle for designing dynamic variable orderings
- Essence: deal with hard cases first – if failure is inevitable, better fail early
- Typical approaches: first visit variable with smallest domain, or variable with most constraints applied to it
Variable and Value Orderings III

**Value ordering**

- Order in which variable domain values are considered during the search process (speculative computation: values may fail and others may be tried later)
- Has impact on efficiency, but also on quality of first solution
- Common principle: succeed-first principle or best-first heuristic
- Example for musical CSP: good heuristic is often a randomised domain value selection (avoids uniformness)
‘Variable orderings’ in manual composition I

- In classical music education, the harmonic structure (underlying chord progression) often written before the actual note pitches
- Some contemporary composers finish rhythmical structure and aspects of instrumentation before writing note pitches
- Melody plus accompany setting: melody is often written first, and then the accompaniment
- Homophonic music: notes of outer voices (sopran and bass) are usually written before middle voices
- Contrapuctual music: composer usually progresses with all voices more or less in parallel
These observations suggest: variable orderings also play an important role for efficiently/adequately solving musical CSPs.
Variable Orderings for Musical CSPs in Existing Systems

Existing constraint systems support a single and static variable ordering: optimised for specific class of musical CSPs – but less suitable for others

- Many music constraint systems represent music simply as a sequence of score objects (e.g., Situation, PWConstraints subsystem PMC – two seminal systems)
- The static variable ordering visits the variables in the order of the sequence
Left-to-Right Variable Ordering I

Left-to-right variable ordering

Static variable ordering of Score-PMC, a subsystem of PWConstraints for polyphonic music [Laurson, 1996]

- Visit note with smaller start time more early
- If two notes share the same start time
  - Visit note of lower voice before note of upper voice
  - Visit longer note before shorter note
Left-to-Right Variable Ordering II

Left-to-right variable ordering, demonstrated at a Bach chorale (cf. [Laurson, 1996])
Left-to-Right Variable Ordering III

Advantages
- Efficient solving of polyphonic CSP
- Rhythmical structure can be arbitrarily complex

Disadvantages
- Rhythmical structure must be fully determined in CSP definition (!)
- This variable ordering hard-wired in Score-PMC: less efficient for, e.g., harmonic CSPs with complex constraints on underlying harmonic structure (causes redundant work at note pitches)
Motivation

We want to solve various different musical CSPs: harmonic, contrapuctual etc. Therefore, we want to choose a variable ordering suitable for the CSP at hand.

The following section introduces a constraint programming model which supports dynamic and user-definable variable and value orderings.
Message-Passing Concurrency: the Underlying Programming Model I

- **Partial values** (logic variables): variable can be
  - free (nothing is known about its value)
  - partially determined (e.g., it is a list with undetermined elements)
  - fully determined
  - Constraints add information about variable values (e.g., unification, numeric constraints)

- **Concurrency**: computations executed in multiple *threads* (created explicitly)
Message-Passing Concurrency: the Underlying Programming Model II

- **Synchronisation of threads on variables:**
  - Thread *blocks* if logic variables used in a statement of the thread lack required information
  - Another thread might provide this information – threads communicate via (dataflow) variables

- **First-class procedures**: procedures (abstracting computations) are first-class values, and support lexical scope

- **Ports**: communication channel for sending data between concurrent threads, including many-to-one communication (asynchronous FIFO)
Message-Passing Concurrency: the Underlying Programming Model III

**Note**

- Message-passing concurrency is highly expressive programming model: it greatly simplifies writing concurrent programs with massive number of threads
- Reason: stateless concurrency (no conflicts of shared resources can occur)

**Example: Erlang programming language**

Model of programming language *Erlang* is similar to this message-passing concurrency model. Ericsson uses Erlang successfully in several Ericsson products for telecommunication.
From Message-Passing Concurrency to Constraint Programming

- No support for search in message-passing concurrent model
- Adding *computational spaces* provides support for speculative computations and search
- In spaced-based constraint model, search is encapsulated
- Alternative to *backtracking*-based search (as in Prolog) – backtracking not feasible with concurrency and interoperating with external world
- We only study simplified view on the constraint model based on spaces
Propagate and search: a **computation space** encapsulates information available on a CSP at a certain stage during the search process.
The Constraint Store

- **Constraint store**: stores information on variable values – conjunction of basic constraints
- **Basic constraint**: representation of information on partial value of a single variable. Example for finite domain integers (FD ints): two forms possible
  - $X \in D$ means $D$ (a set of natural numbers) is *domain* of $X$, special case $X \in \{n\}$ means $X = n$ ($X$ determined to $n$)
  - $X = Y$ means $X$ and $Y$ are equal (unified) – both can be undetermined

Example constraint store

\[ X \in \{1, \ldots, 5\} \land Y = 7 \land Z = X \]
Propagator

- Any more complex constraint (non-basic constraint) expressed by propagator
- A propagator is a concurrent agent
- Propagator aims to add information (i.e. narrows variable domains) which is
  - consistent with constraint store
  - follows from constraint expressed by propagator
- Implemented by algorithm usually highly optimised for its specific constraint
Constraint Propagation II

Example: propagator $X < Y$ narrows domain of $X$ and $Y$

- Store before propagation: $X \in \{1, \ldots, 5\} \land Y \in \{1, \ldots, 5\}$
- Store after propagation: $X \in \{1, \ldots, 4\} \land Y \in \{2, \ldots, 5\}$
Note
Constraint propagation does not necessarily lead to a solution

Example: propagators $X \neq Y$, $X \neq Z$, and $Y \neq Z$ cannot reduce the domains further

$$X \in \{1, 2\} \land Y \in \{1, 2\} \land Z \in \{1, 2\}$$

Stable space
No further propagation is possible: hosting computation space is stable
Constraint Distribution I

Constraint distribution creates two child spaces which are the result of two complementary decisions (expressed by the two added constraints $C$ and $\neg C$)

parent computation space
- propagator
- [...] propagator
- constraint store
- distributor

child computation space
- propagator
- [...] propagator
- constraint store
- distributor

child computation space
- propagator
- [...] propagator
- constraint store
- distributor

constraint $C$

constraint $\neg C$
Constraint Distribution II

- **Constraint distribution** (branching): proceeds to spaces easier to solve, but with same solution set (search)

- **Distributor**: concurrent agent
  - Waits until space is stable
  - Then creates two child spaces (copies of parent space)
  - Add some basic constraint $C$ to store of one child space and its complement $\neg C$ to store of other child space
  - Important: choose such $C$ and $\neg C$ which trigger further constraint propagation
Example distribution strategy: first-fail

Select variable with smallest domain, and determine it to its left-most domain value

Combination of constraint propagation and distribution is a complete search method for solving CSPs
First-Fail Distribution: a Musical Example I

CSP all-distance series definition (length 4)

\[ Xs := \text{list of 4 FD ints, each with domain } \{0, \ldots, 3\} \]
\[ Dxs := \text{list of 3 FD ints, each with domain } \{1, \ldots, 3\} \]
\[ \bigwedge_{i=1}^{3} Dxs_i = |Xs_i - Xs_{i+1}| \]
\[ \land \; \text{distinct}(Xs) \]
\[ \land \; \text{distinct}(Dxs) \]
All-distance series: search tree for all solutions of length 4

<table>
<thead>
<tr>
<th>Space</th>
<th>Domains of elements in $X_s$</th>
<th>Domains of elements in $D_{X_s}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>${0, \ldots, 3}, \ldots, {0, \ldots, 3}$</td>
<td>${1, \ldots, 3}, {1, \ldots, 3}, {1, \ldots, 3}$</td>
</tr>
<tr>
<td>2</td>
<td>${0, \ldots, 3}, \ldots, {0, \ldots, 3}$</td>
<td>${1, {2, 3}, {2, 3}}$</td>
</tr>
<tr>
<td>3</td>
<td>${0, \ldots, 3}, \ldots, {0, \ldots, 3}$</td>
<td>${2, 3}, {1, 2, 3}, {1, 2, 3}$</td>
</tr>
<tr>
<td>4</td>
<td>${1, 2}, {1, 2}, {0, 3}, {0, 3}$</td>
<td>${1, 2, 3}$</td>
</tr>
<tr>
<td>5</td>
<td>${1, 2, 0, 3}$</td>
<td>${1, 2, 3}$</td>
</tr>
<tr>
<td>6</td>
<td>${2, 1, 3, 0}$</td>
<td>${1, 2, 3}$</td>
</tr>
</tbody>
</table>
Distribution Strategy Definition I

- Distribution strategy can be defined ‘from scratch’ (cf. [Schulte, 2002])
- More convenient: definition with a higher-level interface
- Simple interface example expects two first-class functions as arguments (see next slide)
Distribution Strategy Definition II

Order: which variable is distributed (variable ordering)
Boolean function expecting two variables. Returns true if first variable should be visited before the second.

Value: how does distribution strategy effect domain of selected variable (value ordering)
Function expecting a variable, and returning a reduced domain specification for this variable (usually a single domain value).

Example: first-fail distribution strategy definition
Order: $myOrder(X, Y) := \text{getDomSize}(X) \leq \text{getDomSize}(Y)$
Value: $myValue(X) := \text{getMinDomValue}(X)$
Variable and Value Orderings

- User can freely define distribution strategies
- Defining a distribution strategy means defining shape of search tree: i.e., a variable and value ordering
- Next distribution step is always decided only when it is required: dynamic ordering
- Distribution strategies can be changed independently of problem definition
Principles for Efficient Distribution Design

- An efficient distribution strategy results in a relatively small search tree (little amount of failure)
- Constraint propagation never causes a fail (no redundant work)
- An efficient distribution strategy keeps distribution steps at minimum, i.e. helps constraint propagation to do most of the work
- Common example: first-fail principle (see above)
Inaccessible score context

Set of score object which can not be accessed because of undetermined information

Example: if the rhythmical structure is undetermined, then the contexts of simultaneous notes are inaccessible

Note

- If inaccessible contexts are constrained, then constraints applied to inaccessible contexts can not propagate
- This occurs frequently in musical CSPs
Resolve-Inaccessible-Context Principle II

Resolve-inaccessible-context principle

- Resolve constrained inaccessible score contexts early in the search process
- A rule of thumb for designing score variable orderings, like first-fail principle
Besides propagation and distribution, the constraint model has more features – at least mentioned here:

- Constraint propagation between variables with specific domains
- User-definable distribution strategy (branching strategies): specifies search tree
- User-definable exploration strategy: exploration of search tree
- Reified constraints: constraining the truth value of other constraints (e.g., with logical connectives)
- Recomputation: trades memory for run time
- Parallel search: distribute workload of solver on multiple computers
Implementations of the Space-Based Constraint Model

- **Mozart**: implementation of the multi-paradigm programming language Oz, [http://www.mozart-oz.org/](http://www.mozart-oz.org/)
Score Distribution Strategies

- Distribution strategies usually distribute plain variables
- Instead, score distribution strategies distribute parameter objects of music representation
- Advantage:
  - Parameter objects provide access to the score object they belong to, and that way to all information in score (via bidirectional links between score objects)
  - So, a score distribution can make an informed decision
Recap: Bidirectional Links Between Score Objects

The hierarchic structure of a single note and its contained parameters (UML)

Bidirectional link: An Item references its Parameters and vice versa

Note Parameters are offsetTime, startTime, duration, ...
Recap: Bidirectional Links Between Score Objects II

The hierarchic structure of a container with several contained notes

Bidirectional links: A Container references its contained Items and vice versa

Sequential and Note Parameters are omitted for brevity
Definition: First-Fail Score Distribution Strategy

Recap: idea of first-fail distribution

Select parameter which stores the variable with smallest domain, and determine the variable to its left-most domain value

First-fail distribution strategy distributing parameters

Order:

\[
\text{myOrder}(\text{par}_1, \text{par}_2) := \\
\text{getDomSize}(\text{getValue}(\text{par}_1)) \leq \text{getDomSize}(\text{getValue}(\text{par}_2))
\]

Value: \(\text{myValue}(X) := \text{getMinDomValue}(X)\)
Application: First-Fail Distribution Strategy I

Musical example: Fuxian first-species counterpoint

http://strasheela.sourceforge.net/strasheela/doc/Example-FuxianFirstSpeciesCounterpoint.html

- All constraints can be applied directly (i.e. no inaccessible contexts in CSP definition)
- This makes it possible to apply an established general distribution strategy: first-fail
Application: First-Fail Distribution Strategy II

- Small search tree with only few failed notes (squares) until first solution is found (diamond) – i.e. constraint propagation does most of the work
- Runtime: ca. 50 msecs

\(^a\)Pentium 4, 3.2 GHz, 512 MB RAM, Linux Fedora Core 3
Resolving a Single Contexts I

- Typical example of inaccessible score context: if rhythmical structure is undetermined, then simultaneous notes are inaccessible
- One solution: determine all temporal parameters, before other parameters

**Variable ordering which determines all temporal parameters first**

**Order:** \( myOrder(par_1, par_2) := isTemporalParameter(par_1) \)
Resolving a Single Contexts II

- Variable ordering which first determines temporal parameters is only example

- Other example is harmonic CSP: explicitly represented analytical harmonic information should be determined before actual note pitches
Resolving Multiple Contexts in Order I

A variable ordering for harmonic CSPs

First determine the temporal structure, then the harmonic structure, and finally the actual note parameters in the order pitch class, octave, pitch

Order: $\text{determineInOrder}([\text{isTemporalParameter}, \text{isChordParameter}, \text{isPitchClass}], \text{isPitchOctave}], \text{isPitch}])$
Resolving Multiple Contexts in Order II

Function \( \text{determineInOrder} \) returns ordering function \( g \)

\[
\text{determineInOrder}(\text{tests}) :=
\]

\[
\text{let} \quad /* \text{Append a default test function which always returns true.} \quad \text{allTests} := \text{append}(\text{tests}, [f : f(x) := true])
\]

\[
\text{in} \quad g : g(p_1, p_2) :=
\]

\[
\text{getTestIndex}(p_1, \text{allTests}) \leq \text{getTestIndex}(p_2, \text{allTests})
\]

- Function \( \text{determineInOrder} \) expects list of boolean functions; returns variable ordering function which determines parameters in the order specified by the list of boolean functions
- Function \( \text{getTestIndex} \) expects object and list of boolean functions; returns index of first function returning true for given object
Application: Resolving Inaccessible Score Context 1

Musical example: chord progression

http://strasheela.sourceforge.net/strasheela/doc/Example-MicrotonalChordProgression.html

- Distribution strategy for CSP actually combines resolving of multiple score contexts with first-fail principle
- Example: in case of multiple chord parameters, the parameter with smallest domain is determined first
Left-to-Right Variable Ordering I

- Dynamic ordering version of variable ordering of Score-PMC (see above)
- Resolves inaccessible score context of simultaneous score objects dynamically
- Therefore, applicable for polyphonic CSP even when the rhythmical structure is undetermined in CSP definition
Recap: left-to-right variable ordering of Score-PMC
Definition: Left-to-Right Variable Ordering

A left-to-right dynamic variable ordering

Order: \( myOrder(p_1, p_2) := \)

\[
\text{let } start_1 := \text{startTime}(\text{getItem}(p_1)) \\
start_2 := \text{startTime}(\text{getItem}(p_2)) \\
isStart_1\text{Bound} := (\text{getDomSize}(start_1) = 1) \\
\text{in } \text{if } isStart_1\text{Bound} \land (\text{getDomSize}(start_2) = 1) \\
\text{then if } start_1 = start_2 \\
\text{then } isTemporalParameter(p_1) \\
\text{else } start_1 \leq start_2 \\
\text{else } isStart_1\text{Bound}
\]
Musical example: florid counterpoint

http://strasheela.sourceforge.net/strasheela/doc/Example-FloridCounterpoint.html

- Context of simultaneous notes constrained, but inaccessible in CSP definition
- CSP defines relatively complex combinatorial problem. Rules which cause particular complexity (together with standard rhythmic, harmonic, and melodic counterpoint rules):
  - Canon
  - Pitch maxima and minima of phrases must differ
Application: Left-to-Right Variable Ordering II

Runtime measurements (full CSP)

- Left-to-right variable ordering: ca. 4 secs (189 distributable spaces, 175 failed spaces, search tree depth 47)
- Distribution which first determines rhythmic structure: no solution after 1 hour!
- Left-to-right variable ordering at least 900 times faster
Application: Left-to-Right Variable Ordering III

Runtime measurements (simplified CSP: no unique maxima and minima pitches required)

- Left-to-right variable ordering: 1.7 secs (92 distributable spaces, 70 failed spaces, search tree depth 53)
- Distribution which first determines rhythmic structure: 14 secs (630 distributable spaces, 601 failed spaces, search tree depth 62)
- Left-to-right variable ordering almost 10 times faster
Application: Left-to-Right Variable Ordering IV

Result

Choice of suitable variable ordering has great influence on efficiency – also in music domain
Recommended Reading I

Computer-aided composition (CAC) in general


Recommended Reading II

Music constraint programming


Recommended Reading III

Constraint programming (CP) in general

- Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann. – explains various constraint solving algorithms
Recommended Reading IV

**Constraint programming model used by Strasheela**


- Schulte, C. (2002). *Programming Constraint Services*. Springer-Verlag. – most detailed explanation of the space-based constraint model, advanced text
OpenSound Control Interface for Oz (MSc project)

- Strasheela results can be exported in various formats for music notation and sound synthesis
- This project will add OpenSound Control output to Strasheela
- OpenSound Control (OSC) is communication protocol used by many music applications
- OSC exceeds the widespread MIDI standard (e.g., more flexibility what data is send, operates at broadband network speeds)
- Project will create an Oz interface for an existing cross-platform OSC library (C or C++ library, e.g., liblo)
OpenSound Control Interface for Oz (MSc project) II

URLS

- Strasheela: http://strasheela.sourceforge.net
- OSC: http://www.cnmat.berkeley.edu/OpenSoundControl/
- liblo: http://liblo.sourceforge.net
- Oz: http://www.mozart-oz.org
A Graphical User Interface for Strasheela (MRes project) I

- Strasheela highly expressive composition system
- Its user interface is the programming language Oz: suitable for expert users, but makes learning Strasheela hard for new users
- This project will design and implement a graphical user interface for important Strasheela functionality
- Strasheela is programming system: its interface must allow high degree of flexibility
Possible solution: visual programming language (VPL) – many successful music programming systems with VPL exist

Possible approaches

- VPL based on existing VPL system for music (e.g. PWGL or OpenMusic) – generates Strasheela code, communication via socket
- Design of new VPL, e.g., implemented with QTk, a high-level Tk interface provided by Oz
A Graphical User Interface for Strasheela (MRes project) III

**URLS**

- Strasheela: http://strasheela.sourceforge.net
- PWGL: http://www2.siba.fi/PWGL/
- OpenMusic: http://recherche.ircam.fr/equipes/repmus/OpenMusic/
- Oz: http://www.mozart-oz.org
Summary

- Motivation of problem-specific variable and value orderings
- Constraint model based on computational spaces allows for user-defined and dynamic variable and value orderings
- Score distribution strategies implement problem-specific variable/value orderings for musical CSPs. Examples
  - Common technique in general: first-fail
  - Distribution strategies for resolving inaccessible score contexts
  - Left-to-right variable ordering