Representing Trees with Constraints

Ben Curry[1⋆], Geraint A. Wiggins[2] and Gillian Hayes[1]

[1] Institute of Perception, Action and Behaviour, Division of Informatics, University of Edinburgh, Edinburgh EH1 1HN
[2] Department of Computing, School of Informatics, City University, Northampton Square, London EC1V 0HB

Abstract. This paper presents a method for representing trees using constraint logic programming over finite domains. We describe a class of trees that is of particular interest to us and how we can represent the set of trees belonging to that class using constraints. The method enables the specification of a set of trees without having to generate all of the members of the set. This allows us to reason about sets of trees that would normally be too large to use. We present this research in the context of a system to generate expressive musical performances and, in particular, how this method can be used to represent musical structure.

1 Introduction

This paper describes how constraints can be used to represent a specific class of trees that have the following properties:

Rooted - each tree has a node distinguished as the root node.
Ordered - the children of each node are distinct and cannot be re-ordered without changing what the tree represents.
Constant depth - the leaf nodes of each tree are all the same distance from the root.
Strict - at each depth, one of the nodes has at least two successors.

The number of distinct trees in this class is large for each n, where n is the number of leaf nodes. If $n \geq 10$ the set of trees described can not easily be manipulated or used within a computer system. We present here an efficient way of representing this large set of trees, using constraint logic programming, that enables us to use this class of trees in our research.

The structure of the paper is as follows. The next section explains why we are interested in representing sets of trees in the context of music. We then present some implementation details including our representation and the constraints used to specify the trees of interest. Some results are presented that illustrate the effectiveness of this method. Finally, we end with our conclusions.

⋆ Ben Curry is supported by UK EPSRC postgraduate studentship 97305827

2 Motivation: Grouping Structure

This work forms part of our research into creating an expressive musical performer that is capable of performing a piece of music alongside a human musician in an expressive manner.

An expressive performance is one in which the performer introduces variations in the timing and dynamics of the piece in order to emphasise certain aspects of it. Our hypothesis is that there is a direct correlation between these expressive gestures and the musical structure of the piece and we can use this link to generate expressive performances.

The theory of musical structure we are using is the Generative Theory of Tonal Music (GTTM) by Lerdahl and Jackendoff (1983). The theory is divided into four sections that deal with different aspects of the piece's musical structure. We are particularly interested in the *grouping structure* which corresponds with how we segment a piece of music, as we are listening to it, into a hierarchy of groups. It is this hierarchy of groups that we seek to represent with our trees.

The rules are divided into two types: *well-formedness* rules that specify what structures are possible; and *preference* rules that select, from the set of all possible structures, those that correspond most closely to the score.

The rules defining grouping structures are based on principles of change and difference. Figure 1 shows four places where a grouping boundary may be detected (denoted by a '$*$'). The first case is due to a relatively large leap in pitch between the third and fourth notes in comparison to the pitch leaps between the other notes. The second boundary occurs because there is a change in dynamics from piano to forte. The third and fourth boundaries are due to changes in articulation and duration respectively.

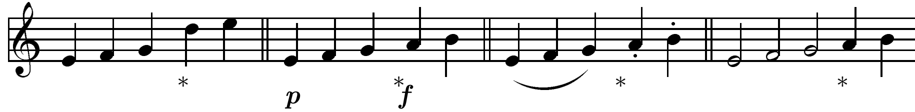

Fig. 1. Points in the score where grouping rules may apply

Figure 2 shows an example of a grouping structure for a small excerpt of music. We can see that the music has been segmented into five different groups, one for each collection of three notes. The musical rest between the third and fourth groups causes a higher level grouping boundary that makes two higher level groups which contain the five groups. These groups are then contained within one large group at the highest level.

The grouping structure can be represented with a tree. Figure 3 shows a tree representation (inverted, to aid comparison) for the grouping structure shown in Fig. 2. The leaf nodes at the top of the tree correspond to the notes in the score, and the branches convey how the notes are grouped together. This is an

example of the class of tree we are trying to represent. From this point onwards the trees will be presented in the more traditional manner, i.e. the leaf nodes at the bottom and the root node at the top.

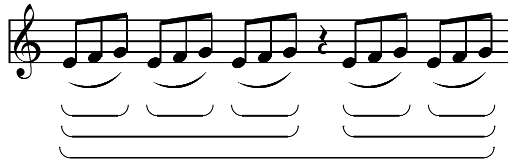

Fig. 2. An example grouping structure

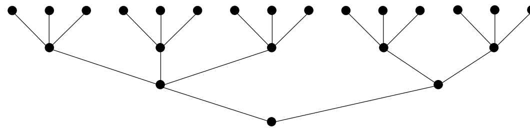

Fig. 3. Tree representing the grouping structure shown in Fig. 2

Although the GTTM grouping rules are presented formally, the preference rules introduce a large amount of ambiguity. For a particular piece of music, there are many possible grouping structures which would satisfy the preference rules. The purpose of the present research is to devise a way to represent this large set of possible structures in an efficient way so that they can be used by a computer system.

Using our hypothesis of the link between musical structure and expressive performance, one of the core ideas of our research is to use rehearsal performances by the human musician to disambiguate the large set of possible grouping trees. The expressive timing used by the musician in these rehearsals provides clues as to how the musician views the structure of the piece. A consistent pattern of timing deviations across a number of performances will enable us to highlight points in the score where the musician agrees with the possible grouping boundaries.

3 Using Constraints

This section of the paper explains how we use constraint logic programming (Van Hentenryck, 1989) to represent sets of trees. Although constraints have been used in the areas of music composition (e.g. Henz 1996) and tree drawing

(e.g. Tsuchida 1997), this research is concerned with an efficient representation of large numbers of tree structures, which is a problem distinct from these.

Constraint logic programming over finite domains enables the specification of a problem in terms of variables with a range of possible values (known as the *domain* of the variable) and equations that specify the relationships between the variables. For example if (1), (2) and (3) hold then we can narrow the domains of x and y as shown in (4):

$$x \in \{1..4\} \tag{1}$$

$$y \in \{3..6\} \tag{2}$$

$$x + y \geq 9 \tag{3}$$

$$x \in \{3..4\} \wedge y \in \{5..6\} \tag{4}$$

The following sections outline the representation and the constraints we use to specify the class of trees. We begin by discussing the representation of the nodes and then present the five types of constraints used to ensure that the trees generated belong to our class.

3.1 Representation

We know that our class of trees will be monotonically decreasing in width from the leaf nodes up to the root and, therefore, we can represent the set of trees by a triangular point lattice of nodes[1]. Figure 4 shows the point lattices for trees of width $n = 3$ and $n = 4$.

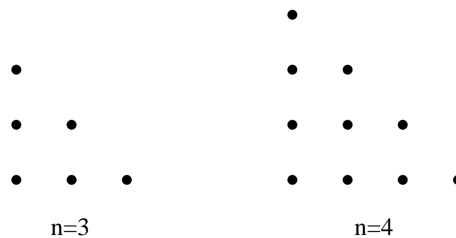

Fig. 4. Point lattices for trees of width 3 and 4

Each node has the following variables (illustrated in Fig. 5):

1. *id*: a unique identifier;
2. *uplink*: a connection to the level above;

[1] An implementation detail means that there is always a path from the highest node of the point lattice to the leaf nodes, but this highest node should not be considered the root node. The root node may occur at any height in the point lattice and is identified as the highest node with more than one child.

3. Downlink values which represent all the nodes on the level below that are connected to this one.

The *id* is specified as an (x,y) coordinate to simplify the implementation details. The *uplink* variable contains an integer that represents the x-coordinate of the node on the level above to which this node is connected i.e. node ($uplink$, $y+1$). The downlink values, specified by a lower (dl) and upper (du) bound, refer to a continuous range of nodes on the level below that may be connected to this one i.e. nodes (dl, $y-1$)...(du, $y-1$).

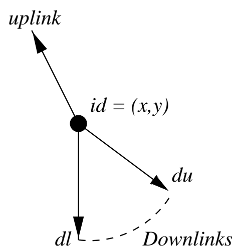

Fig. 5. A typical node

The next sections present the constraints that are applied to the nodes in order to create the specific set of trees in which we are interested. They begin by specifying the domains of the variables and then constraining the nodes so that only those trees that belong to our class can be generated.

3.2 Node Constraints

The first task is to define the domains of the variables for each node. Due to the triangular shape of the point lattice, the uplink for each node is constrained to point either upwards, or up and to the left of the current node. We constrain the downlink for each node to span the nodes directly below, and below and to the right of the current node.

The constraints (given in (5)-(8)) define the domains of the *uplink* and downlink range (i.e. dl and du) for each node[2]. The *uplink* lies in the range $\{0..x\}$ where x is the x-coordinate of the current node. The zero in the range is used when the node is not connected to the level above.

$$domain([uplink]) = \{0..x\} \tag{5}$$

$$domain([dl, du]) = \{0..n\} \tag{6}$$

$$(dl = 0) \oplus (dl \geq x) \tag{7}$$

$$du \geq dl \tag{8}$$

[2] The $\oplus$ in (7) denotes exclusive-or.

The downlink specifiers dl and du are constrained in a similar way to lie in a range from {0..n} with the added constraints that du has to be greater than or equal to dl and that dl either equals zero or is greater than or equal to x. Figure 6 shows how these constraints relate to the direction of the connections to and from each node.

Constraint (9) handles the situation of a node which is not used in a tree. If the *uplink* of the node is zero then the downlinks of the node must also be zero.

$$((dl = 0) \Leftrightarrow (du = 0)) \wedge ((dl = 0) \Leftrightarrow (uplink = 0)) \tag{9}$$

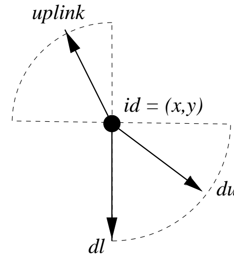

Fig. 6. Constraining the Uplinks and Downlinks

3.3 Level Constraints

To ensure that the connections between two levels do not cross, constraints (10) and (11) are applied to each pair of adjacent nodes. For a pair of nodes A and B, with A directly to the left of B, the $uplink_B$ must either point to the same node as the $uplink_A$ or to the node to the right of it or, if it is unused, be equal to zero (10).

$$(uplink_B = uplink_A) \vee (uplink_B = uplink_A + 1) \vee (uplink_B = 0) \tag{10}$$

Once one of the uplinks on a particular level becomes equal to zero, all the uplinks to the right of it must also be zero (11). This prevents the situation of an unconnected node in the midst of connected ones.

$$(uplink_A = 0) \Rightarrow (uplink_B = 0) \tag{11}$$

Figure 7 shows examples of correct and incorrect mid-sections of a tree under these new constraints. The bottom example is incorrect because it violates constraints (10) and (11).

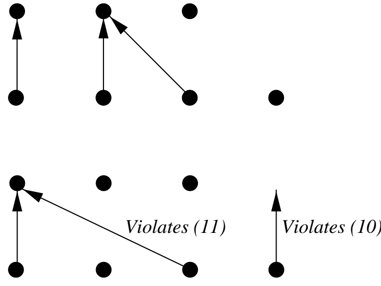

Fig. 7. A correct (*top*) and incorrect (*bottom*) mid-section of a tree

3.4 Consistency Constraints

If the current node refers to a node in the level above, the x-coordinate of this node must appear within its downlink range. Constraint (12) ensures that if this node points to a node on the level above, the downlink range of that node must include this one. Figure 8 shows how this constraint affects two nodes where the lower one is connected to the upper one.

$$(x_{above} = uplink_{this}) \Leftrightarrow ((x_{this} \geq dl_{above}) \wedge (x_{this} \leq du_{above})) \tag{12}$$

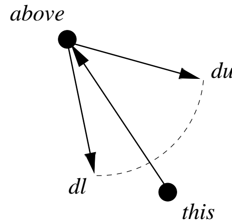

Fig. 8. Ensuring connectivity between nodes

3.5 Width Constraints

We now constrain the trees to decrease in width as we travel from the leaf nodes to the root node. The width of a level is defined as the number of nodes that have a non-zero uplink on that level. Constraint (13) deals with this situation with the precondition that the width of the current level is greater than 1. This precondition is necessary to allow situations such as the first four trees in Fig. 10 where we consider the root node to be at the point where branching begins.

$$(width_i > 1) \Rightarrow (width_j < width_i) \quad (13)$$

We want to ensure that the trees decrease in width to reduce the search space as much as possible. Figure 9 shows an example of a tree which does not decrease in width between two levels, we can remove this tree from our search space as it does not contribute anything new to the grouping structure as we move from level i to level j.

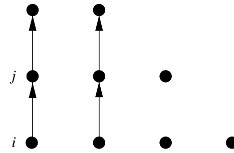

Fig. 9. A section of a tree that does not decrease in width

3.6 Edge Constraints

The last step is to ensure that the uplink of the rightmost[3] node on a level points inwards (the rightmost node in Fig. 7 is an example of this). We find the maximum x of the level above that has a non-zero *uplink* and then ensure that the *uplink* of the rightmost node points to it ((14) and (15)).

$$\mathcal{S} = \{x : id(x,y) \text{ has } uplink_x \neq 0\} \quad (14)$$

$$uplink \leq max(\mathcal{S}) \quad (15)$$

3.7 Valid Trees

The constraints given in §3.2 to §3.6 define the set of trees which belong to our class. Figure 10 shows an example set of width $n = 4$. The white nodes are ones that appear in the generated solutions but are not considered to be part of the tree since the root of the tree is the highest node with more than one child.

3.8 Using the Constraint Representation

The constraints which have been defined in the sections above describe a general class of trees. The next step is to introduce aspects of the grouping structure to

[3] By 'rightmost' we mean the node on the current level with the maximum x-coordinate that has a non-zero uplink.

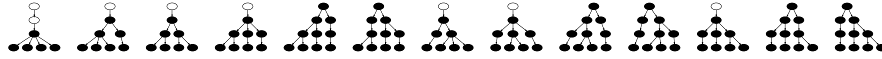

Fig. 10. All the trees of width four ($n = 4$)

reduce this large set of trees to only those trees that correspond to the piece of music being analysed.

Every point in the musical score where a grouping boundary could occur is identified, for each of these points we then measure the relative strength of this boundary against the surrounding ones. Every boundary point can then be used to determine the shape of the tree by ensuring that every pair of notes intersected by a boundary corresponds to a pair of nodes separated in the tree set.

To separate the nodes in a tree, we need to ensure that the parents of the nodes are not the same, and if we have a measure of relative strength between boundaries, we can specify how far towards the root the nodes need to be separated. The algorithm below shows how this is implemented:

$Repel(id_A, id_B, strength)$
  if $(strength \geq 1)$ **then**
    $parent(id_A) \neq parent(id_B)$
    $Repel(parent(id_A), parent(id_B), strength - 1)$
  endif

This recursive predicate takes two nodes and a strength argument and recursively ensures that the nodes are separated up to a height *strength*. Figure 11 shows an example tree where the tree is divided into two subtrees by a *Repel* constraint that is applied with $strength = 1$ between the second and third leaf nodes.

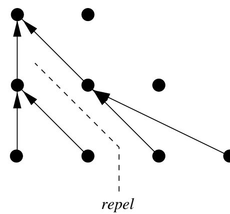

Fig. 11. How *Repel* affects the tree

4 Results

We generated all the trees up to width $n = 7$ and found a similarity with an entry in the Online Encyclopedia of Integer Sequences (Sloane, 2000). It matched a sequence discovered by the mathematician Arthur Cayley (1891) based upon this particular class of trees which has the recurrence shown in (16) and (17)[4]

This recurrence defines the number of trees that belong to our class that are of width n.

$$a(0) = 1 \tag{16}$$

$$a(n) = \sum_{k=1}^{n} \binom{n}{k} a(n-k) \tag{17}$$

Using our representation, the approximate formula, derived experimentally, for the number of constraints to represent the set of all the trees of width n is given in (18).

$$Constraints \approx \frac{2}{3}n^3 + 11n^2 - \frac{2}{3}n - 24 \tag{18}$$

The number of trees of width n grows rapidly (e.g. the number of trees of width 50 is 1.995×10^{72}). By contrast, the number of constraints it takes to represent the same number of trees is 1.1×10^5.

Figure 12 shows how the number of trees grows in comparison to the number of constraints as we increase the width of the tree. The number of trees increases at a greater than exponential rate whereas the number of constraints increases at a low-order polynomial rate.

5 Conclusions

This paper presents our research on representing a specific class of trees with constraint logic programming. Although the number of constraints needed to represent these large sets of trees is comparatively small, the computational time needed to solve the constraints is not.

The representation currently restricts the trees to have leaf nodes at the same depth; however, it does allow the addition of quite simple constraints to change the class of trees represented. For example, to restrict the trees to strictly binary trees we need only add the constraint $du = dl + 1$.

With the use of constraints we have delayed the generation of trees until we have added all the possible restrictions, this offers a great reduction in complexity and allows us to manipulate trees of greater width than would normally be possible.

[4] Where $\binom{n}{k}$ is the standard n choose k formula given by: $\frac{n!}{k!(n-k)!}$

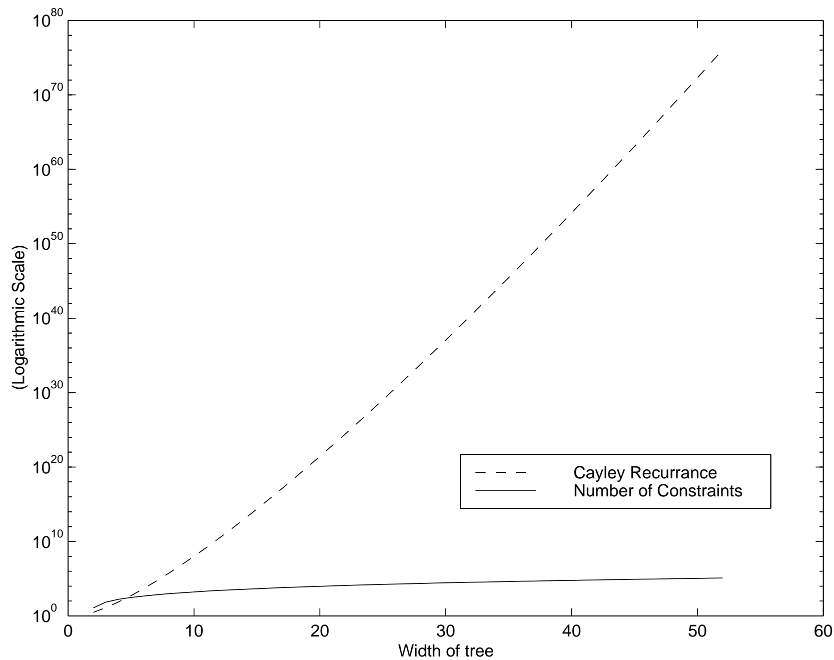

Fig. 12. A graph showing how the number of trees and number of constraints grows with the width of the tree

References

Cayley A.: On the Analytical Forms Called Trees. Coll. Math. Papers, Vol. 4. Cambridge University Press (1891)

Henz M., Lauer S. and Zimmermann D.: COMPOzE – Intention-based Music Composition through Constraint Programming. Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence, IEEE Computer Society Press (1996)

Lerdahl F. and Jackendoff R.: A Generative Theory of Tonal Music. MIT Press (1983)

Sloane N. J. A.: The On-Line Encyclopedia of Integer Sequences. Published electronically at http://www.research.att.com/˜njas/sequences/ (2000)

Tsuchida K., Adachi Y., Imaki T. and Yaku T.: Tree Drawing Using Constraint Logic Programming. Proceedings of the 14th International Conference of Logic Programming, MIT Press (1997)

Van Hentenryck P.: Constraint Satisfaction in Logic Programming. Logic Programming Series, MIT Press (1989)